NAVAL POSTGRADUATE SCHOOL
Monterey, California

DISSERTATION

FIDELITY OPTIMIZATION IN DISTRIBUTED VIRTUAL ENVIRONMENTS

by

Michael V. Capps

June 2000

Dissertation Supervisor: Michael Zyda

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, 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, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2000	**3. REPORT TYPE AND DATES COVERED** Doctor of Philosophy Dissertation

4. TITLE AND SUBTITLE Fidelity Optimization in Distributed Virtual Environments	**5. FUNDING NUMBERS**
6. AUTHOR(S) Michael V. Capps	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	**12b. DISTRIBUTION CODE**

13. ABSTRACT

In virtual environment systems, the ultimate goal is delivery of the highest-fidelity user experience possible. This dissertation shows that is possible to increase the scalability of distributed virtual environments (DVEs), in a tractable fashion, through a novel application of optimization techniques. Fidelity is maximized by utilizing the given display and network capacity in an optimal fashion, individually tuned for multiple users, in a manner most appropriate to a specific DVE application.

This optimization is accomplished using the *QUICK* framework for managing the display and request of representations for virtual objects. Ratings of representation **Q**uality, object **I**mportance, and representation **C**ost are included in model descriptions as special annotations. The *QUICK* optimization computes the fidelity contribution of a representation by combining these annotations with specifications of user task and platform capability.

This dissertation contributes the *QUICK* optimization algorithms; a software framework for experimentation; and associated general-purpose formats for codifying Quality, Importance, Cost, task, and platform capability. Experimentation with the *QUICK* framework has shown overwhelming advantages in comparison with standard resource management techniques.

14. SUBJECT TERMS distributed virtual environment, linear programming, computer graphics, resource management	**15. NUMBER OF PAGES** 226
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFI-CATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

Approved for public release; distribution is unlimited

FIDELITY OPTIMIZATION IN DISTRIBUTED VIRTUAL ENVIRONMENTS

Michael V. Capps
B.S., University of North Carolina at Chapel Hill, 1994
M.S., University of North Carolina at Chapel Hill, 1996
S.M., Massachusetts Institute of Technology, 1999

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2000

Author:

Michael V. Capps

Approved by:

Professor Michael Zyda, Professor of Computer Science

Ted Lewis
Professor of Computer Science

P. David Stotts, UNC
Associate Professor of Computer Science

Donald Brutzman
Assistant Professor of Applied Science

Rudolph Darken
Assistant Professor of Computer Science

Approved by:

Dan Boger, Chair, Computer Science Department

Approved by:

Anthony Ciavarelli, Associate Provost for Instruction

ABSTRACT

In virtual environment systems, the ultimate goal is delivery of the highest-fidelity user experience possible. This dissertation shows that is possible to increase the scalability of distributed virtual environments (DVEs), in a tractable fashion, through a novel application of optimization techniques. Fidelity is maximized by utilizing the given display and network capacity in an optimal fashion, individually tuned for multiple users, in a manner most appropriate to a specific DVE application.

This optimization is accomplished using the *QUICK* framework for managing the display and request of representations for virtual objects. Ratings of representation **Q**uality, object **I**mportance, and representation **C**ost are included in model descriptions as special annotations. The *QUICK* optimization computes the fidelity contribution of a representation by combining these annotations with specifications of user task and platform capability.

This dissertation contributes the *QUICK* optimization algorithms; a software framework for experimentation; and associated general-purpose formats for codifying Quality, Importance, Cost, task, and platform capability. Experimentation with the *QUICK* framework has shown overwhelming advantages in comparison with standard resource management techniques.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

My thanks to my committee for their excellent guidance. I know this work would have remained eternally unfinished were it not for the efforts of my dissertation supervisor, Dr. Michael Zyda.

The National Science Foundation supported this work with a Graduate Research Fellowship, which allowed me to focus on a single topic through multiple academic institutions.

This work would not have been completed without the aid of the faculty, staff, and students of the Naval Postgraduate School. I have been extremely impressed by this institution. Special thanks to John Locke, for his help with modeling; and to the thesis processor and graduation administrators for their understanding during a last-minute crisis.

My thanks to the many doctoral students who have influenced my graduate career. Their guidance regarding this research, and the general process of surviving the dissertation ordeal, has been invaluable. I am both relieved and saddened to leave this fraternity.

I will be eternally grateful to my wife, Laura Elizabeth Capps, for her understanding and support throughout this effort.

I. INTRODUCTION

A. THESIS STATEMENT

Resource consumption in distributed virtual environments can be optimized given specialized descriptions of user task, model complexity, model quality, and display platform capability.

B. MOTIVATION

The field of computer graphics has advanced rapidly in recent decades, but there have always been models and simulations whose complexity outstrips available technology. High-end network throughput has continued to improve, but the communications requirements of popular shared virtual reality systems exceed the capabilities of the latest networking technology.

In virtual environment systems, the ultimate goal is delivery of the highest-fidelity user experience possible. Unfortunately, users' fidelity requirements are not met currently, nor does it appear they will be met for some time. It is therefore of utmost importance that the capacities of virtual environment client resources are exploited in an optimal fashion. What is more, it is desirable to manage this optimization such that the fidelity of the user experience degrades smoothly in the face of additional system stress. That smooth degradation is the dual of system scalability, which is a primary concern in the design of Virtual Reality (VR) and Collaborative Virtual Environment (CVE) systems.

The desire for graphics optimization has led to the development of several resource management systems. However, these methods are useful for only limited domains of graphics models and applications, such as terrain datasets or 2-1/2 dimensional architectural walkthroughs. General purpose optimization techniques are needed.

Network bandwidth has been described as the single largest roadblock in deployment of CVE systems [Sandin *et al.,* 1997]. The most effective answer thus far has been to manage communications so as to only communicate information when necessary. In VR systems which store environment descriptions on distributed servers, most clients naïvely request visual descriptions for all portions of the virtual environment. Large-scale environments are rarely displayed in their entirety, a fact which can be exploited to optimize network communications.

C. APPROACH

The aforementioned efforts to optimize graphics and networking have, until now, been performed independently. The research described in this dissertation investigated the development of a unified framework for general-purpose virtual environment optimization. The results show it is possible to determine how best to display the environment, and how best to communicate its definition, with a single algorithm. This joint optimization of graphics and networking leads to systems more capable of supporting distributed collaboration in graphical environments.

This generalized optimization was performed by abstracting concepts of resource costs and limitations, such that network bandwidth was treated no differently than graphics pipeline throughput. Similarly, fidelity characteristics were abstracted to allow comparison between heterogeneous objects. Display decisions and object requests were then optimized by maximizing fidelity within the various cost thresholds.

Previous forays into cost and fidelity determination have been intended only for very limited application domains. No general-purpose display management systems use more than a single floating-point number to describe the very complex factors involved in the delivery of a high-fidelity user experience. This problem is further complicated by the fact that high-fidelity need not always correspond to the highest-quality image presentation. Accordingly, a primary effort of this thesis was the definition of the primary factors that contribute to the effectiveness of a virtual environment. These factors are divided into the following categories: quality, importance, cost, task, and platform capability.

Given these factors, it becomes possible to formulate an optimization problem for driving object display and request. One goal of this dissertation was guaranteed-correct optimization, but such algorithms were determined to have exponential time complexity. This encouraged development of approximation algorithms that run in polynomial time. While the best of these algorithms can only guarantee a solution 50% of optimal, in practice the results are usually much more useful.

The effectiveness of this optimization framework is demonstrated by a proof-of-concept implementation.

D. CONTRIBUTIONS OF THIS WORK

This dissertation claims the following contributions to the state of the art:

- combined optimization of graphical and networking resources in virtual environments
- general-purpose algorithms for exact and approximated CVE optimization
- definition of Fidelity as a function of virtual world objects and their representations
- inclusion of dynamic user task definition in the CVE optimization process
- inclusion of dynamic display platform capabilities in the CVE optimization process
- model annotation formats for codifying quality and cost
- software framework for experimentation with optimization parameters and algorithms

E. DISSERTATION ORGANIZATION

The remainder of this dissertation is organized as follows:

- **Chapter II:** To provide the reader with a background in the technical areas of this dissertation, this introduction chapter is immediately followed by a summary analysis of related work.

- **Chapter III:** This chapter presents a more formal statement of the optimization problem. This includes the description of a typical instance of the display problem, and defines a novel technique for reaching the solution of that problem: the *QUICK* framework. That display problem is then extended by a variety of complicating factors, in order to demonstrate the general applicability of the *QUICK* method.

- **Chapter IV:** This chapter discusses how platform-specific capabilities are provided to the optimization formulation. Chapter IV also explains the dramatic effect of user task upon fidelity, and how task affects the computation of each *QUICK* factor.

- **Chapter V:** With a specification for platform capability and application task, it is then possible to detail the form of the various inputs to the *QUICK* optimization algorithms. Chapter V begins this process with the Quality annotation, which captures the relative accuracy of each representation for an object.

- **Chapter VI:** This chapter follows Quality with a description of the Importance annotation, and how to compute the Importance function from the static and dynamic characteristics of a given scene object. Chapter VI also explains how the Cost of a representation is computed relative to a display platform.

- **Chapter VII:** This chapter explains the selection of graphics software libraries for the *QUICK* proof of concept implementation. It additionally introduces the *QUICK* scene graph and the *QUICK* framework's software architecture.

- **Chapter VIII:** Having defined the problems in virtual environments, as well as the structure of the *QUICK* scene graph, it is possible to define the *QUICK* optimization. This chapter begins with a discussion of the problem formulation from a scene graph instance. That is followed with a complexity analysis of the guaranteed-correct solution. The chapter concludes with a discussion of techniques for reducing the complexity of the optimization.

- **Chapter IX:** This chapter presents the details of constructing a software implementation which uses the *QUICK* optimization framework.

- **Chapter X:** Chapter X analyzes the effectiveness of the contributions of this dissertation. The primary technique is comparison to other related work, which is performed both with the systems as a whole and with their optimization algorithms taken independently.

- **Chapter XI:** The dissertation concludes with a summary of contributions and suggestions for application of those results. A number of promising avenues are provided for follow-on research.

II. RELATED WORK

A. INTRODUCTION

Graphics management systems have adopted widely disparate approaches to the display and communications problems. To some extent, this broad range of techniques reflects the relative lack of experience in developing this class of applications. Additionally, almost all approaches to date have addressed only a small subspace of the problem, usually specific to a single application.

This proposal draws heavily upon previous research results in a number of subfields of computer science. This chapter documents significant research literature in each of those subfields, with special attention paid to those that are particularly relevant or considered ground-breaking. Where appropriate, the discussion includes comparison with this work, so as to demonstrate its contribution to the state of the art.

B. FIDELITY DEFINITION AND JOB TASKS IN VIRTUAL ENVIRONMENTS

Reaching fidelity to facilitate tasks in a virtual environment is of utmost importance, but rarely is a formal definition of fidelity used. Generally designers are content with systems that maximize resolution and frame rate—i.e., deliver as realistic an experience as possible. However, the failure of virtual reality in some exercises implies that fidelity may come from symbolic representation rather than realistic presentation.

Generally job task is inherent in an application, as most optimized virtual reality applications are designed for a specific job task. Job task in the *QUICK* system is abstracted, allowing run-time task changes that in turn affect Importance and Quality. There is surprisingly little supporting research in the area of abstracting or codifying job task. Chapter IV offers examples of how task can be considered independently of application or virtual world, and includes a mechanism for task-based modification to the *QUICK* factors.

C. QUALITY AND THE TIME/SPACE INTERFACE

While human perception and noticeable difference is an active area of psychophysical research, perceptual quality for three-dimensional images in virtual environments is usually assumed to fit a standard set of simple heuristics. Microsoft's proposed Talisman graphics architecture [Lengyel and Snyder, 1997] is an excellent example of using fiducials to estimate fidelity. The Talisman system generates 2D sprites from multiple models and then composites them with appropriate back-to-front ordering. The authors suggest that this approach allows better targeting of system resources by exploiting frame-to-frame coherence with image warps.

The fiducials they suggest for comparing representations are:

- geometric: maximum point-wise distance between original and current characteristic points

- photometric: shading differences between original and current points, with adjustments to normals considered

- sampling: measures how samples are stretched or compressed

- visibility: ensures that occlusion in the eye-direction is resolved properly

These metrics of course were developed to apply specifically to the sprite-based rendering algorithms of Talisman. Surprisingly, they comprise one of the most comprehensive approaches to image quality in a graphics management system today.

The evaluation of the quality of a single object or representation is itself a complex process. To further complicate matters, the quality of a representation is affected by surrounding representations. For instance, a very-high resolution image of an building might appear to be high quality when displayed alone, but if it is included in a geometric scene generated from a slightly different angle then the unmodified high-resolution image might be distracting.

The Berkeley Walkthrough system [Funkhouser and Séquin, 1993] uses cost/benefit analysis for switching between levels of detail. That analysis includes a hysteresis factor, which reduces the benefit of switching to a new representation by an amount proportional to the difference in level of detail from the current representation.

SGI's IRIS Performer package [Rohlf and Helman, 1994] also notes the deleterious effects of switching between levels of detail, and provides two mechanisms to ease the transition: blending and morphing. Blending draws both the new and old representation simultaneously, using transparency to fade one from prominence to the other, whenever a LOD switch is required (illustrated in Figure 1).

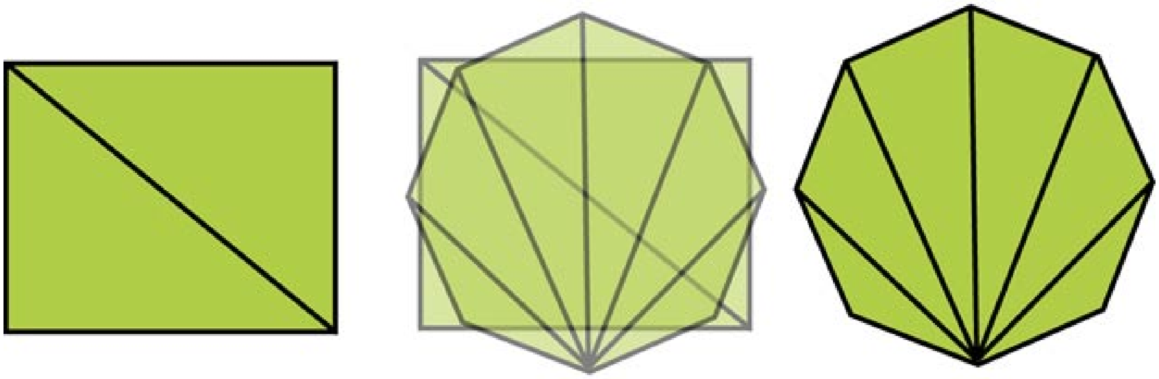


Figure 1. LOD blending in Performer.

The obvious drawback is that this method requires rendering both representations simultaneously. Performer also supports a standard geometric morphing package, which has additional computation requirements instead of rendering requirements.

D. INTEREST AND IMPORTANCE GENERATION

Using interest to determine quality choice has been used previously in several limited-domain systems. The aforementioned Virtual Planetary Explorer terrain-display system [Hitchner and McGreevy, 1993] kept a list of important, modeler-specified geometric points. Interest falls off with distance from each point; the interest of a region was the sum of importance contributed by all such points.

The Berkeley walkthrough also incorporates a limited notion of importance in the Cost/Benefit heuristic [Funkhouser and Séquin, 1993]. The Benefit of display of an object is computed from standard factors of resolution, screen size, and hysteresis effect. Then, Benefit is modified by a factor based on the type of the object; for example, walls are more important than furniture in an architectural walkthrough, and enemy robots might very important in a game. This information was planned to be computed statically, during model creation time, and there is no record of implementation of any influence of ontological description upon importance.

Francois Sillion's Ville project [Sillion *et al.*, 1997] for displaying urban models substitutes simplified triangle meshes for complex building geometry when appropriate. The system includes modifications by Sami Shalabi [Shalabi, 1998] which use city morphology to determine where best to generate those image impostors. Because the images must created from only limited number of positions, likely path points must be predicted. Possible viewpoints are reduced to the streets in the model; the street information is input during the model generation phase. Besides creating viewpoints at places where visibility undergoes major changes (for example, street intersections), an importance generator determines major landmarks such as tall buildings and city squares. Those landmarks are given additional detail, and therefore more detailed impostor information, as shown in Figure 2.

While the automatic generation of importance can be effective, the process is particularly model- and application-specific. Excepting only procedurally-generated models, development of most virtual world data requires significant human interaction. Even systems which generate models from images usually contain a significant manual image-registration step. Rather than spending inordinate programming time developing algorithms for generating importance, it is likely more sensible to have the modeler—who is already very familiar with the model and its intended use—spend an extra few minutes labeling important areas and objects. Construction time of urban models, for instance, is usually gauged

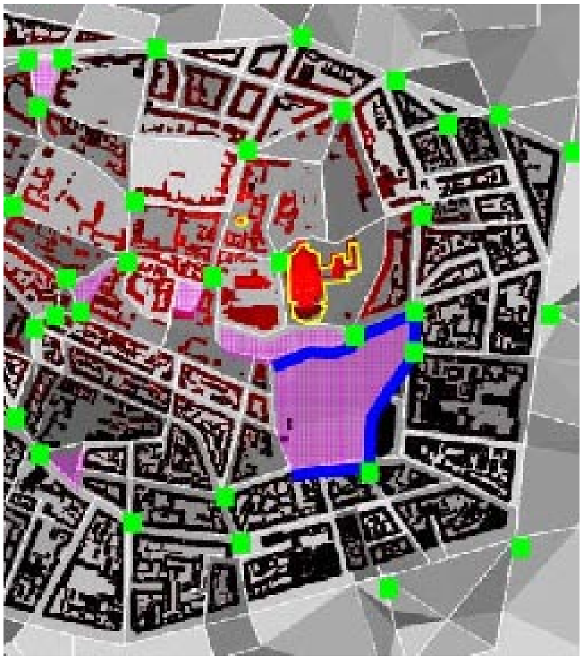


Figure 2. The VILLE system importance generator for urban scenes [Shalabi, 1998].

in modeler-months; it does not seem unreasonable to add a few modeler-minutes (or hours) for Importance annotation. Authoring tools can easily be modified to support such improvements.

E. SPATIAL SUBDIVISION

The *QUICK* system requires that virtual environments be arranged in a scene graph which is a forest of hierarchical trees. The notion of dividing virtual worlds into such hierarchies was originated by James Clark [Clark, 1976], who contended that spatially-based hierarchical object definitions, coupled with bounding volumes, can improve the process of visibility culling. Since that time, spatial partitions have been used as the basis for optimizing a number of graphics processes, including animation and ray-tracing.

The Binary Space Partition Tree, or BSP-tree [Fuchs *et al.*, 1980], is the most general hierarchical division; it can reproduce the division of other methods such as Quadtrees (regularly-divided 4-way trees) and KD-trees (axial binary trees). It does so at a higher cost of traversal and computation.

Subdivision techniques can also be combined into hybrids, such as in the overlay method of [Magillo and Floriani, 1995]. Here two hierarchical subdivisions, with varying level of detail, are used in conjunction to divide a model. This was specifically developed for terrain applications, where frequently a model comes from a variety of sources in varying resolutions. Hybrid data divisions are used in many more graphics applications, for example raytracing acceleration (linetrees with octrees) and radiosity solutions (hierarchical grids and BSP trees) [Drettakis and Sillion, 1996].

F. VISIBILITY DETERMINATION

It has long been understood that the number of polygons in a complex model far exceeds the number able to be rendered in an interactive manner. Visibility determination is the first, and probably most effective, method used to cull polygons from the set to be rendered. Consequently, nearly all modern graphics systems support hardware implementations of frustum culling algorithms.

Precomputation of visibility is particularly effective in standard models that can be subdivided spatially. Research at the University of California [Teller and Séquin, 1991] and University of North Carolina [Airey *et al.*, 1990, Luebke and Georges, 1995] was particularly successful in architectural walkthroughs. Buildings are easily divided into logical

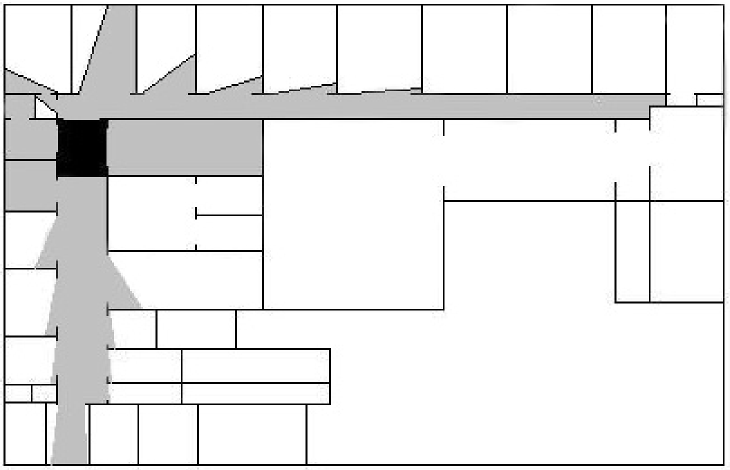


Figure 3. Cell-to-cell visibility using portal stabbing [Teller and Séquin, 1991].

spaces (rooms), and the visibility computation is constrained in the 2 1/2 dimensional space with such a high incidence of axially-aligned occluders. Though the Berkeley authors argue that their visibility algorithm can be extended to a 3D architectural model; the complexity of extending to a general-form model, however, has prevented any such an implementation. The UNC system fired random rays between two cells to determine inter-cell visibility; this system is an effective approximation but an exact answer requires an infinite number of rays. The Berkeley system determines sight-corridors between portals (doors and windows into cells); this is illustrated in Figure 3, which shows the portions of each room visible from the dark cell containing the eyepoint. This simplification was effective only because of the constraints on walls and portals, and inevitably was an expensive computation in terms of memory and processor consumption.

IdSoftware uses the portal-visibility model from the above systems in their extremely popular 3D video game Quake [IdSoftware, 1996]. Quake was best known for near-perfect utilization of PC hardware capability. Along with other accelerating measures such as BSP-trees, light maps for precomputed radiosity lighting, and texturing, Quake uses potentially-visible sets for culling. Each room in a Quake model stores an associated PVS of rooms which are visible from one or more viewpoints in the room. When rendering a scene, Quake first eliminates all rooms not in the PVS, and then uses a special angular-sweep algorithm to eliminate rooms not in the view frustum. The multi-user version of Quake uses a centralized server process; the server performs awareness management by only forwarding visible actions to players. Essentially, if an action (such as gun fire or motion) occurs outside of a player's PVS, the player is unaware the action occurred. Permanent actions (player death, door open/shut) are always communicated for model consistency. For events that cause noise, such as gun fire, each cell also has a PHS—a potentially-hearable set—so the action is properly forwarded to players who can hear the action, even though they might not be able to see it.

Yagel and Ray of the Ohio State University [Yagel and Ray, 1996] use a similar regular space subdivision into cells; they then classify those cells into interior, exterior, and wall cells. This method is particularly well suited to environments such as caves, sky-lines, blood vessel models, and the like. Cells can be discretized into a quad-tree, grid, or purely data-driven (BSP or KD tree) data structure; the model can use only one subdivision throughout. Portals are inappropriate for the intended model domain; visibility is determined using sight corridors, or if necessary, by searching for connected blocking occluders. Each cell stores a list of other cells visible from it; during the rendering stage, the visible-cell-list for the cell containing the viewpoint is the set to be rendered. Notably this system was implemented for two-dimensional models only, though the extension to three dimensions was planned.

Precomputation of visibility is not always the most effective method. Often a precomputation is prohibitively expensive; unable to be performed in advance because the model is generated dynamically; or the model does not lend itself to appropriate segmentation. For instance, it is obviously not feasible to compute visibility from all possible viewpoints. Determining exactly which polygons are visible in a given frame is likely also too complex to compute interactively. Satyan Coorg's algorithm [Coorg and Teller, 1996] determines in real-time the most significant occluding polygons in a scene, and uses only that subset to test whether other polygons are visible. (In the color version of Figure 4, major occluders are shown in black.) This conservative algorithm exploits spatial and temporal coherence between frames, making dynamic computation quite cost-effective on an amortized basis.

Researchers at the University of Genova in Italy have had success in the limited domain of terrain maps and height fields [Magillo and Floriani, 1994]. A hierarchical terrain map contains detail stored in a progressive manner, such that searching deeper into the hierarchical model's tree (with some computation) gives greater and greater detail. Using visibility for culling in this situation requires two stages—an initial computation at a given resolution level, and an update when the desired resolution is changed. [Magillo and Floriani, 1994] presents a method for directly traversing the structure to the

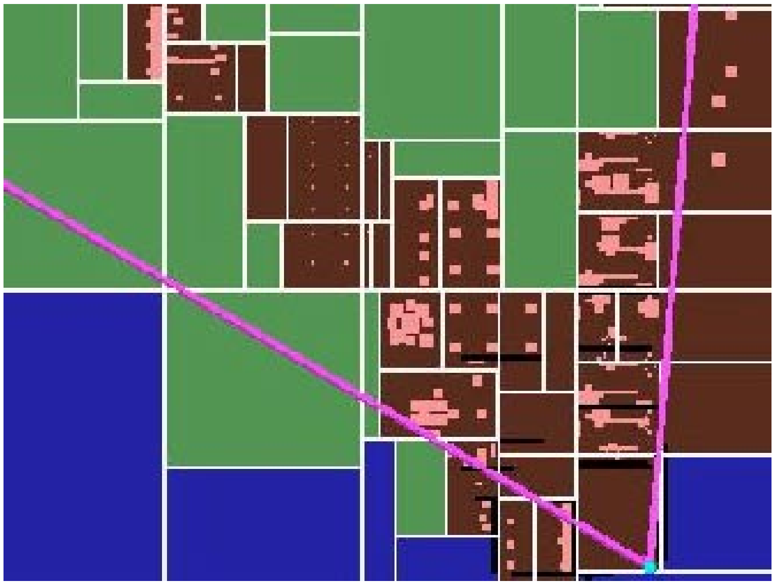


Figure 4. Dynamic visibility with the largest-occluder algorithm [Coorg and Teller, 1996].

depth of a desired resolution and computing visibility during that traversal, rather than requiring the explicit computation of the model at that resolution. Two traditional methods, sweep-line and front-to-back traversal, are extended to the hierarchical model without a significant increase to time or space complexity.

G. DISPLAY COST DETERMINATION

The true cost of displaying primitives with a graphics subsystem is a heatedly debated topic; this is demonstrated by the numerous available methods [Zyda *et al.*, 1990] for profiling graphics workstation (and PC card) performance. The *QUICK* model depends on an accurate approximation of the relative cost of rendering one representation versus another. Previous systems using cost/benefit rendering have allowed either only geometric LODs, or geometry and one alternate representation; the *QUICK* model is more general in that respect though of course each representation type will require full cost analysis. Cost

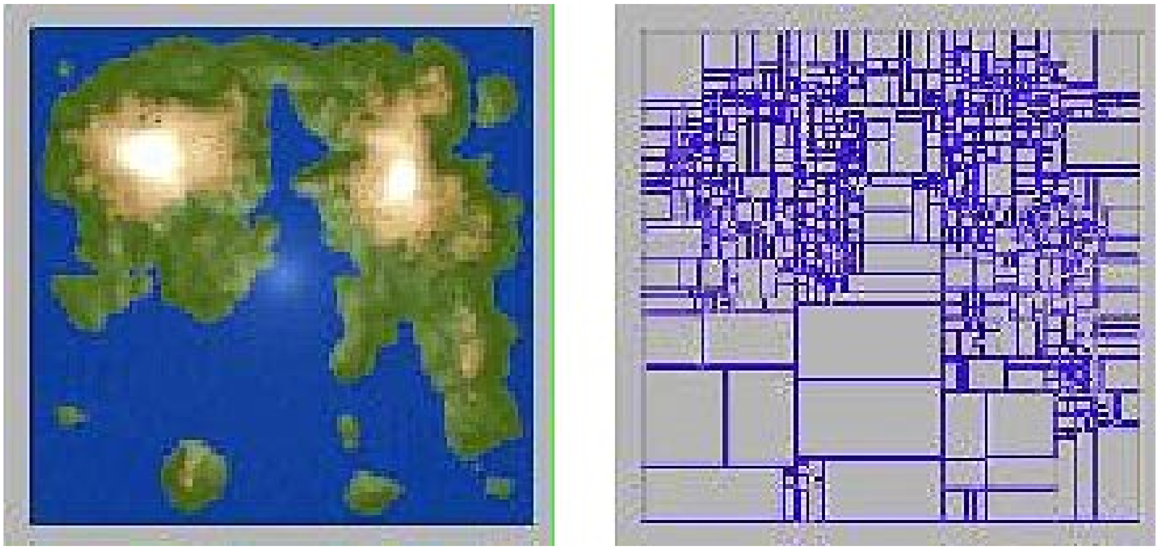


Figure 5. Spatial subdivision for hierarchical image caching [Shade *et al.*, 1996].

analysis has been especially rigorous in the fields of ray tracing and radiosity calculation, in which various approaches make narrowly-different cost/performance trade-off decisions [Appel, 1968, Speer *et al.*, 1985, Danskin and Hanrahan, 1992, Reinhard *et al.*, 1996].

The Berkeley system [Funkhouser and Séquin, 1993] also made a brief investigation into the cost of displaying geometric objects. Given an object *O*, a geometric level of detail selection *L*, and a rendering algorithm *R*, the system computed the *Cost(O,L,R)* function. With the assumption that all objects are geometric, and that Cost is equal to time spent rendering, that Cost function can be simplified to be the maximum of the per-primitive processing, per-pixel processing, and per-vertex processing times in the graphics pipeline. The function includes a constant multiplier for each subsystem, based on experimental data for the given display platform. While this is an excellent first pass at a Cost heuristic, it is not particularly appropriate for multiple display algorithms nor multiple platforms, and it allows no consideration for non-polygonal representations.

Researchers at the University of Washington and Microsoft Research have developed two management systems of major significance to this research project (the second is discussed in a later section). The first is the hierarchical image caching walkthrough [Shade *et al.*, 1996] by Jonathan Shade *et al.* This system assumes path coherence, and stores rendered images of nodes in the scene graph so they may be re-used. The scene graph is divided spatially (with a BSP-tree), and during the rendering traversal their algorithm decides what form to render. Figure 5 shows an overhead-view of a virtual environment and the corresponding spatial division. If an image has been stored and it is still appropriate, it is used. If an image is not used, the system decides if the cost of rendering an independent image of the node (and drawing the resulting image) is less than the cost of rendering the geometry, given an estimate of how long the image is likely to be applicable. An eye-point that moves slowly in a straight line, for instance, is much more likely to allow repeated use of stored images than one that moves and turns erratically.

The error metric for deciding the suitability of a cached image is simply based upon maximum angular discrepancy of the corners of the node's bounding box. Given a user-specified error threshold, it is possible to predetermine an area for which all viewpoints will be within the tolerance for angular discrepancy. When this system is used with a pregenerated path, it is simple to compute the number of frames a cached image is within error tolerance. In an interactive setting, current velocity and maximum acceleration can be used to make a worst-case estimate. Then the comparison is simply the cost of rendering geometry versus the amortized cost of a single frame of geometry (to create the image cache) plus displaying a quadrilateral with a texture-map of the cached image. The costs of each were determined experimentally for the test platform.

H. RESOURCE MANAGEMENT

Managing of resources in display systems is not a new concept, though past systems have addressed only specific application area. This section includes a discussion of complexity reduction methods, such as level-of-detail generators. Following that is a review of systems that manage complexity and quality trade-offs, first with geometric LOD models and then with limited hybrid rendering models.

1. Level of Detail (LOD) Generation

Level of detail generation has been an active area of research since the 1970's [Clark, 1976]. Lately, that research has focused more on the efficient generation of LOD representations that capture the essence of the information while reducing the cost of display as much as possible.

The simplification envelope [Cohen *et al.*, 1996] project is a joint effort between UNC and Duke University, for generating a hierarchy of level-of-detail approximations for a given polygonal model. Probably the most impressive point about the research is that an approximation is guaranteed to have its points within a user-specifiable error-bound (distance from boundary) of the original model. Their algorithms generate approximations to triangle meshes that attempt to minimize the total number of polygons required to meet the user's constraint. Conveniently, this system also automatically generates appropriate

LODs and viewing distances for display.

Researchers at Georgia Tech developed a system for generation of continuous-detail representations of terrain height-fields [Lindstrom *et al.*, 1996]. Rather than pregenerating those representations, the geometric model is generated dynamically as needed. Within this framework, minor adjustments in detail are computationally inexpensive. A viewing system built to render those models uses bounds on image quality, with standard distance and pixel-area metrics, for choosing the precision of representations. The work in [Ferguson *et al.*, 1990] is similar in that it generates continuous levels of detail for terrain models.

Generation of appropriate levels of detail is a well-explored area. Other systems include Lodestar [Schmalstieg, 1997], for generating LODs for VRML; and the view-dependent polygonal simplification method described in [Luebke and Erikson, 1997].

2. LOD Management

Switching between precomputed geometric level-of-details is the most common method for reducing display cost for a given frame. One of the first complete-solution systems was VPE, NASA's Virtual Planetary Explorer[Hitchner and McGreevy, 1993]. VPE was essentially a terrain-display system, though in this case the terrains displayed are those of entire planets. VPE's stated goal was the display of Martian terrain with a 10 Hz update rate, yet the terrain data was much too complex to render in such a fashion. The solution was multiple LOD representations for the terrain; representations were selected based upon three criteria: 1) distance from the viewpoint, 2) distance from the center of field of view, and 3) user-defined level of interest. The second criterion was based on the assumption that in a head-mounted display, the user focus is on the center of the display (and that visual resolution is highest at the focal point). For the interest criterion, the user picked certain geometric points in the model to be important, based on application scenario. The level of interest in any region was then computed as the sum of the importance lent by all such points, where the importance was attenuated by the square of the distance. The VPE system is certainly an important predecessor to the *QUICK* model, in that it incorporates ideas of importance and quality, but its scope is limited to geometric terrain data only.

Probably the most popular method for building LOD-accelerated applications is the IRIS Performer package by SGI [Rohlf and Helman, 1994]. The Performer automatically adds such effective procedures as view-frustum culling, multiprocessing, and scene-graph optimization. Relevant to this discussion, however, is the level-of-detail switching algorithms. The Performer API allows specification of multiple levels of detail for a scene node, as well as specification of distance, pixel-size, and field-of-view criteria for switching between those representations. Performer can also track the processing load on the system, and use that information to switch to less costly representations in the case of overload. The Performer toolkit is an excellent general-purpose system for optimal rendering, but it performs automatic LOD-switching in only a limited manner.

Probably the single project most influential on this research is the Berkeley walkthrough system, specifically Thomas Funkhouser's adaptive display algorithms for interactive frame rates [Funkhouser and Séquin, 1993, Funkhouser, 1993]. Using the PVS cell-to-cell visibility techniques described previously, the system was able to greatly reduce the complexity of the model portion to render. The full system also performed cell-to-object and eye-to-object visibility checks, and stored multiple levels of detail for each object. Finally, an optimized data-storage format and prediction mechanism was used to select proper representations for those objects. This system was the first to use dynamic heuristics for LOD determination; it tracked frame rate and would adjust detail to bring the frame rate in line with that desired by the user. That heuristic was a simple Cost/Benefit analysis of choosing each representation.

This system is again a limited-domain application of many of the concepts of the *QUICK* system. There is no notion of quality of representation; user fidelity is defined rudimentarily as frame rate; cost is the number of polygons; representations are only geometry; the model is limited to 2 1/2 dimensions; and importance is limited to visibility determination and distance. This is not to say that the Berkeley Walkthrough is not an excellent application, but rather, to show that its ground-breaking work has natural ramifications for future work such as the *QUICK* model.

It is interesting to note that LOD use is particularly well-accepted by the graphics community as a means of display acceleration. VRML, the specification for the primary web-based graphics format, includes **LOD**, a level-of-detail node [Pesce, 1995]. **LOD** contains an array of distances and a group of object representations; representations are switched between based on the distance from the viewpoint to the object.

The second system by the University of Washington particularly relevant to this project speeds rendering of complex environments with a spatial hierarchy. The scene is divided hierarchically into an octree, and then each octree node is associated with a “color cube” [Chamberlain *et al.*, 1996]. The color cube is an approximation of the contents, using a single color and a single level of transparency, as determined from the six axial directions. The rendering traversal algorithm determines if a given node subtends a pixel area on the screen greater than some user-specified parameter. If so, the algorithm recursively checks the node’s children; if not, then the color cube approximation is drawn instead. When a leaf is reached with size greater than the parameter, the geometry drawn normally. The paper cited above explains that this method is not effective for continuous surfaces, because the transparency value is particularly view-dependent; the test application was the rendering of a forest of trees.

3. Hybrid Display Management

Hybrid display technology had its real start in the raytracing community, where ray-tracing would be used in concert with other methods to generate images either more quickly or with more realistic lighting effects [Arvo and Kirk, 1990]. Other raytracing efforts traversed multiple representation types simultaneously, for example volume-arrays and polygons in [Levoy, 1990].

The *QUICK* model is primarily intended for interactive graphics techniques, rather than as another method for accelerating raytracing. As such, this section looks at systems which have been successful in rendering multiple representation types in a single coherent image.

The hierarchical image caching project mentioned previously [Shade *et al.*, 1996] is a particularly relevant management system for hybrid rendering technology. For each scene node, the rendering algorithm chooses between two representations based upon a quality metric. Additionally, the system actually has the ability to create new representations when it is cost effective to do so.

Researchers at the University of North Carolina extended their previous work in architectural walkthroughs by adding image warping [Rafferty *et al.*, 1998]. Given a partition of a building into cells, their system renders the nearest cells with geometry and farther cells as static images. At each portal to a cell, a set of images is pregenerated. In any given frame, the most relevant images are composited with image-warping techniques to generate the final scene. This resulted in significant acceleration of frame rate due to the polygonal complexity of the model.

Paul Debevec at the University of California at Berkeley developed a system to use geometry and photographs for both modeling and rendering [Debevec *et al.*, 1996]. In the limited domain of architectural geometry, photogrammetric modeling is possible to recover the basic geometry of a scene. The technique uses stereo pairs of images to determine accurate depth readings at various pixels in an image. The rendering phase dynamically generates the textures for the base geometry by mapping the photograph taken from the nearest point to the viewpoint. The authors point out that the depth-image information extracted in the model-based stereo algorithm can be useful in image-warping renderers as well.

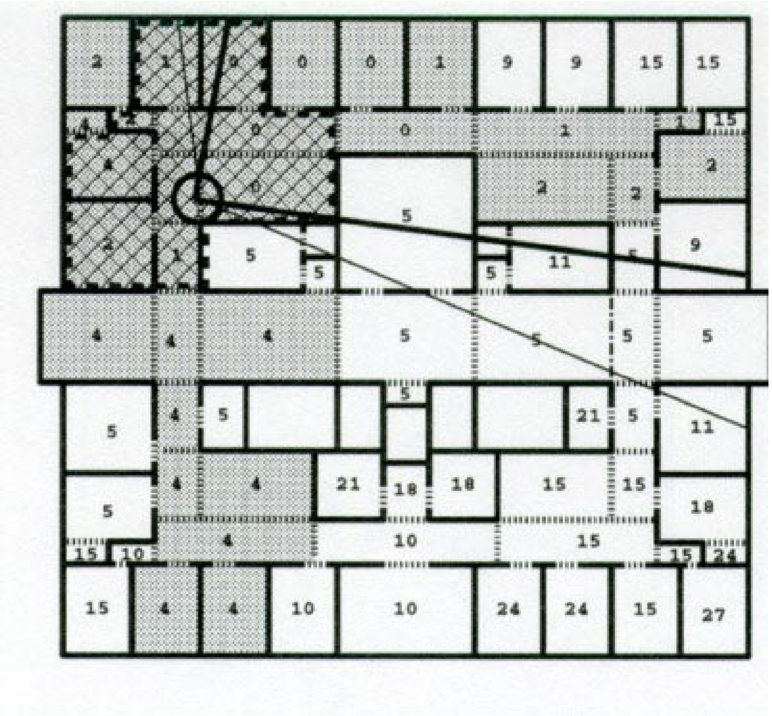


Figure 6. Motion prediction in the Berkeley walkthrough [Funkhouser, 1996].

I. MOTION PREDICTION

Motion prediction is not a major focus of this work, and a simplifying set of “motion classes” will be used. The Berkeley Walk-through [Funkhouser, 1996] used a known limitation of foot speed, and a user-specified frame rate, to determine the length of time a user would need to reach rooms in a 2.5 dimensional architectural model. Figure 6 shows the number of time steps required to reach each room in Soda Hall; those rooms reachable within five time-steps are shaded. In a model with such tightly constrained user paths as a building, this is an effective mechanism for culling objects from the list of objects to be prefetched. Similar work, applied to path-planning for robots in a geometric environment, can be found in [Canny and Lin, 1993].

J. LOCAL CACHING IN GRAPHICS SYSTEMS

The *QUICK* system employs a novel series of inputs in order to make decisions in the management of a distributed graphics cache. Disk cache management techniques have been used to excellent effect in graphics systems in which the extent of a local model outstrips core memory storage capacity. For instance, the original NPSNET system [Falby *et al.*, 1993] used a hierarchical data cache for swapping between terrain tiles. The SPLINE system [Waters *et al.*, 1997] uses region-based segmentation for caching; at any given time, only the current region and neighboring regions are in main memory. Even early entertainment software used such techniques, in order to stay within the very tight memory constraints of early personal computer technology. For example, the first Castle Wolfenstein software title could only store a single (two-dimensional) room of the castle in memory; moving through a portal resulted in a cache miss and disk load delay.

K. DISTRIBUTED GRAPHICS SYSTEMS

Computer-supported collaboration, and distribution of graphical data, are mature areas of computer science. A number of previous efforts share some portion of the goals of this project, but no system to date has embodied all of its objectives.

1. Research Systems

Many systems have a notion of shared graphical objects and communication of state changes to those objects. The Reality Built for Two system [Blanchard *et al.*, 1990], for example, allowed collaboration between two users; NPSNET [Macedonia *et al.*, 1995] allowed loose collaboration between thousands. Each of these projects takes a different approach to the distribution of initial object state, network topology, and collaboration paradigms, but all assume homogeneous client software. The Distributed Interactive Simulation (DIS) [DIS, 1993] and High-Level Architecture (HLA) [Kuhl *et al.*, 1999] standards enable cooperation between heterogeneous clients, as long as they follow a set of network protocols. Nearly all of these systems could benefit from asset prioritization of the sort described in this thesis.

A review of networked virtual environment architectures, and a tutorial for these standard methods of information sharing, can be found in Singhal and Zyda's 1999 text [Singhal and Zyda, 1999]. A subset of these systems are discussed in detail below.

a. DIVE

The DIVE system [Carlsson and Hagsand, 1993] from the Swedish Institute of Computer Science is a landmark tool for virtual collaboration and interaction. DIVE was one of the first to include clients for multiple machine architectures (RS6000, SGI, Sun), which contributed to its popularity. Each user in DIVE has a replica of a shared database, which is distributed using the ISIS [Birman *et al.*, 1985] distributed locking mechanism; applications appear to only be accessing shared memory, which is transparently updated by ISIS. A DIVE universe is partitioned into multiple worlds, which are associated with ISIS process groups; switching between worlds is permitted, but a user can only be aware of a single world at a time. DIVE uses no loading priority when transferring a virtual world description. There is support for world segmentation, with scene graph subdivision;

additionally the application can perform session management over these segments. There is no documented case of these facilities being used in combination for asset prioritization.

b. MASSIVE

The family of Aura applications [Benford and Fahlen, 1993] atop the DIVE system used the intersection of invisible geometrical volumes around objects and avatars to trigger actions and connections; for example, avatars within a certain range might have an audio chat channel begun between them. The MASSIVE system from the University of Nottingham [Greenhalgh and Benford, 1995] greatly expanded the model of those volumes and used their intersection to define awareness between objects. The aura, which can be any description of a spatial volume, is used to determine if there should be any interaction at all between two participants (similar rules can be used with objects); if the auras intersect, a connection is created between the two participants.

Then a finer grain of granularity takes over, based on additional volume functions. Observers have a focus, which is a function defining their region of interest, and a nimbus, which is a measure of their projection's likelihood to be noticed by other observers. Generally, the auras will be simple functions whose intersection is easy to compute, such as spheres. Once a connection is created, each participant determines the amount of intersection that exists between their focus and the other's nimbus, and that implies a level of awareness.

These functions can be attributed to different media, so for instance a visually striking but very quiet participant might have a large visual nimbus but small audio nimbus. Of particular note is that awareness need not be equivalent in each direction; many users might be aware of a loud participant, who could herself have the impression of solitude. Due to the server-less nature of MASSIVE, however, she would continue to receive constant updates on the other participants because of the aura intersection; the information would be discarded at the application layer.

It is entirely possible for an observer to have no focus at all for various media, and this is used as an excellent method to allow logical heterogeneity. A participant with a full-featured graphical display and no audio simply has a focus size of 0 for audio; a participant with a text-only console could use a size-0 focus for the visual medium and simply place an ASCII character in their position in a two-dimensional map.

The level of awareness determined from the amount of focus/nimbus intersection, can be used to good effect during rendering. For instance, low visual awareness can be translated into display of lower-detail geometry. This might also be used for prioritization of state transfer. Similar to DIVE, the world description is segmented, and it does offer internal feedback facilities that would make such prioritization simple to support.

c. SPLINE

SPLINE is Mitsubishi Electric’s Scalable Platform for Large Interactive Networked Environments [Anderson *et al.*, 1995], the initial implementation of OpenCommunity. SPLINE facilitates CVE development by providing a shared world model that is shared transparently across multiple clients. Applications are then able to interact with each other by making changes to objects they own, and observing changes in remotely-owned objects. Objects are represented in the world in a hierarchical fashion, such that each object has a parent and zero or more children. Positions in the world are carried through this relationship, such that if a parent object is translated all of its children are translated in a like manner. Objects can also have a locale as a parent. Locales are atomic awareness regions which correspond to an area in the virtual universe.

A typical application might subscribe to a locale, by connecting to its server and joining that locale. Objects in that locale are placed in the application's world model, and it begins receiving updates on those objects. The application can publish new objects in the locale, which are in turn shared among other applications aware of that locale. Any modifications made by the application are reflected to remote applications as well. When an object is moved across a locale's boundary, the locale is queried to see if a neighboring locale exists in that direction. If so, the object is moved to the new locale. Because object positions act as an offset from the center of its locale, the object's position is modified (by a special transformation representing the locale crossing) to be appropriate for the new locale.

Locales are an efficient method of solving problems of data flow by breaking up a virtual world into chunks that can be described and communicated independently. Locales divide the world based on three key features: each locale has a separate address, its own coordinate system, and a list of locally-neighboring locales.

Locale-based relevance serves as a highly-efficient culling mechanism. The standard awareness model in SPLINE makes a user aware of the locale which contains its avatar, plus the locale's immediate neighbors. Local coordinate systems for locales allow high positional precision, even in galaxy-sized virtual worlds; small memory representations of position can be highly accurate. Storing only local neighboring relationships in a locale facilitates combination of locales from different designers and sources. Separate worlds need not be designed with each other in mind. Even when differing wildly in size and shape, they can be combined painlessly. Also, the combination of independent coordinate systems and locally-defined neighboring relationships allows the representation of non-Euclidean virtual spaces: one-way doors and spaces larger on the inside than outside are simple examples.

All objects in SPLINE are associated with a single locale. A virtual world can contain thousands of locales, with each locale having knowledge of only its immediate neighbors. Yet applications need a way to query about objects in the virtual universe, to find other users, and the like.

SPLINE solves this with beacons. A beacon is an object with two special fields: a tag, and a locale address. The beacons of a virtual world act as a content-addressable index from tags to locale addresses. Beacons are stored in the world model normally, as they are associated with some locale, but they also are tracked by a special beacon server process. SPLINE can find those servers by hashing on the beacon's tag. So, with just the tag, an application can contact a beacon server and ask for information about all beacons with a certain tag.

These tags are used by world creators to mark special objects that need to be found. For instance, if an author wanted to ensure that police stations could be found easily, she could add a beacon with a police tag as a child of each police station object. Then, by publishing the tag in a public forum (such as the application's help files, or a WWW page), users could use it to find all the beacons with police-tags (and thereby the police stations). Beacons can also be used for temporary situations. For instance, one might add a beacon to a moving object to be able to track it, or users might tag themselves so friends might find them.

(1) Diamond Park. Diamond Park was the first large-scale virtual world and application built using SPLINE. The park is a square mile of landscape, with buildings, lakes, and simple terrain which makes up sixty-two locales. Users interact while riding computer-controlled exercise bicycles, and conversing via an audio channel. The design of some Diamond Park structures shows the power and flexibility of a locale-based world, and they are discussed in detail.

The Desert House is a small building within Diamond Park containing a much larger desert terrain. The desert locale was in fact designed separately from Diamond Park, and placed within to illustrate composability. Two difficulties arise in the addition of the Desert House: first, the polygonal complexity of the interior was such that most client hardware could draw little else at interactive rates; and second, viewing across the doorway gives an inconsistent view due to the difference in scale factor. Both problems were easily solved by adding a vestibule to the entrance of the house, such that two locales were between the exterior and interior. Because the world model in SPLINE consists of the current locale and its immediate neighbors, at no time are the Desert House and the Diamond Park exterior both in the world model. Additionally, the border-locales are situated such that there is no sight line that contains both the exterior and interior.

Diamond Park contains twenty-two obelisks which act as a method to quickly move about the park—without biking a mile each time! The obelisks appear small from the outside, but upon entering the user sees a room with twenty-two archways leading out of the other obelisks. This does cause awareness of a large portion of the model. To avoid an inconsistent view across a boundary between two differently-scaled locales, each archway is filled with a static pre-generated picture of the exterior of each obelisk.

d. Shared Scene-Graph Systems

The Distributed OpenInventor (DIV) project [Hesina *et al.*, 1999] uses the scene graph as a shared memory structure, and it encourages the authoring of graphical applications that are distributed in a manner nearly transparent to the programmer. The system also includes excellent high-performance networking facilities. GMD's Avocado system [Tramberend, 1999] similarly distributes data by transparent replication of the scene graph, in this case that of the Performer graphics library, on sgi systems. The Scene Graph as Bus approach [Zeleznik *et al.*, 2000], part of the National Tele-Immersion Initiative, is a proposed mechanism for mapping between heterogeneous scene graphs, in a cross-platform manner.

2. Internet-Based Graphics Technologies

As processing and bandwidth capacity has increased across the Internet, the possibility of Internet-based graphics has emerged. The *QUICK* framework is specifically targeted for the client-server model which is the norm for the World Wide Web, and later chapters investigate the applicability of *QUICK* to web-based graphics technologies. The following sections give a brief overview of some standard formats for Internet-based three-dimensional graphics.

a. Virtual Reality Modeling Language (VRML)

The Virtual Reality Modeling Language [VRM, 1997] is a file format for describing interactive three-dimensional objects and worlds. It was designed to be deployed on the Internet, and from the very first has had HTTP hyperlink capability embedded in objects. VRML's simplicity has led its growth as a universal interchange format for three dimensional datasets, as nearly all applications can read and write the VRML ASCII file format. In addition to this simplicity, the ability to embed dynamic behaviors offers significant expressivity, and VRML is used for applications from medical visualization to multi-user worlds.

Though VRML is not itself a virtual environment system, this discussion considers VRML-based worlds and browser applications as a whole. Most VRML applications require that the virtual world be downloaded in its entirety before interaction is allowed. Author control of this step is permitted using Switch and LOD nodes. VRML worlds often consist of multiple VRML files, linked via World Wide Web locations; most browsers resolve these links and fetch all included files before passing control to the user. VRML files already contain excellent inherent model subdivision: each file represents a standard tree-based scene graph, and files can contain internal switch and Level of Detail nodes that divide the files further. This indicates that VRML is an immediate possibility for application of *QUICK* concepts. In fact, the *QUICK* file format (discussed in section VII.C is a non-standard extension of VRML. Those extensions could be similarly accomplished using VRML's PROTO capability, albeit in a fashion which does not lend as well to efficient computation in Java3D VRML-parsing software.

b. Extensible 3D (X3D)

Often heralded as the next generation of VRML, X3D [X3D, 2000] is an XML-based file format for 3D scene description. The X3D specification will be split into a very small core functionality and profiles atop that core; the intention is that simple browsers can support only the core, and that more advanced browsers can support additional extensions. While X3D is not yet complete, it shows much promise; a major design consideration is the inclusion of an asset prioritization scheme, and it appears that a *QUICK* X3D profile could be integrated into advanced performance-conscious browsers.

c. Streaming Geometry

One method to combat the initial delay in interactivity common in networked virtual environments is to stream geometry. In this approach, representations are sent in a very low detail at first, and then progressively refined. The user is able to interact with the scene while this refinement process occurs. These representations are considered *continuous* in that they provide a large number of options for display detail. Continuous representations can significantly reduce the complexity of fidelity optimization; possibilities are discussed further in the future work section at the end of the dissertation.

d. QuickTime Virtual Reality (QTVR)

QuickTime VR [Chen, 1995] is an image-based format which gives the impression of immersion in a virtual scene. Panoramic cameras are used to generate wide-angle images, which are stitched together to create a cylindrical image centered on the viewer's position. The user is then able to rotate in place; minor zoom capability is offered via image-warping techniques. QTVR scenes can consist of multiple cylindrical nodes, which the user can then navigate between interactively. There is no notion of asset prioritization in QTVR; however, loading is performed progressively, and the user is able to navigate partially-loaded scenes during download. Despite these extensibility limitations, *QUICK* annotations might be integrated in content prior to generating QTVR scenes, thereby offering an adaptive resolution control mechanism for otherwise-static fidelity.

3. Multi-User Entertainment Software

The release of id Software's entertainment game Quake [IdSoftware, 1996] was a quantum leap in the availability of distributed virtual reality on the desktop. In 1997, in fact, their product was hesitatingly labeled the state of the art in the entire field of networked virtual environments—including research systems [Capps and Stotts, 1997]. In the multi-player version, each participant connects to a single centralized server. Motion and action updates are communicated via the server to other players. The server stores the current state of the virtual environment, in order to provide support for late-comers. The original game comes with a limited number of maze and building maps to play; new environments can be found on the web, or dynamically downloaded when first joining a session in that environment.

However, this latter method exposes a major weakness of the network architecture. Most Quake players connect to the server by modem; the application of a number of advanced techniques in awareness management and client-side simulation make possible play with such limited bandwidth. A client connecting to an unfamiliar environment automatically requests the environment description, which is usually about one megabyte in size. This process nominally takes five minutes on a 28.8kbps modem, but usually requires closer to fifteen minutes due to the server's double duties. Game play does not begin until the entire model has been acquired; interestingly, most servers run a game for ten to fifteen minutes before cycling to a new map. Therefore it is quite possible for a participant to be stuck in a cycle where each environment file is moot before its download is complete.

Quake environments are purposely divided into rooms with limited connectivity, so as to allow precomputation of visibility between spaces. This reduces the computation required for the physics and rendering engines, as in the Berkeley Walk-through system [Funkhouser *et al.*, 1992]. This division is exactly the sort of subdivision required for asset prioritization: rooms can and should be downloaded in order of importance. Yet Quake allows absolutely no interaction during the download process—fidelity is zero.

L. SUMMARY

This chapter presented work related to the design and implementation of an optimization scheme for virtual environments. Overview summaries were provided for graphics, human factors, virtual environments, and networking issues germane to this effort. Virtual environments research builds upon the foundations of these and many other disciplines, and it is therefore neither appropriate nor possible to provide an exhaustive literature review. Key surveys, as well as more complete bibliographies, are available in [Durlach and Mavor, 1994] [Singhal and Zyda, 1999] [Keshav, 1997] [Foley *et al.*, 1990] [Baecker and Buxton, 1987] and [Baecker *et al.*, 1995].

The review presented in this chapter shows that creation of a general-purpose optimization system for distributed virtual environments has not been previously proposed or attempted. However, many previous efforts have faced issues similar to those that constitute this research; the chapters that follow show how such previous results can be integrated into the larger scope of this dissertation.

III. EXPANDED PROBLEM STATEMENT

In order to present a general-form optimization for display selection, it is necessary to characterize a generic form of the model display problem: **“Optimal display is characterized by the selection of a visual representation for scene nodes in a virtual world, such that the combined display of those selections provides the highest-fidelity user experience on a given display platform.”**

Though the terms of this statement are familiar, their usage bears definition:

- **scene node**: A denotable unit in a scene graph, usually a single artifact, group of artifacts, or virtual object represented by visual representations. The terms “scene node” and “virtual object” are used interchangeably in this document.

- **scene graph**: A hierarchical structure representing a virtual world or scene, divided either spatially or logically, consisting primarily of scene nodes.

- **visual representation**: A computer-parsable graphical description, such as polygons, triangles, images, etc. A single scene node may have multiple representations, for example, graphical Levels of Detail (LODs). A scene node must contain at least one visual representation. Therefore, the display selection for any scene node involves a minimum of two possibilities—the single representation or no representation at all.

- **combined display**: Visual presentation of each scene node’s chosen representation.

- **highest-fidelity user experience**: The highest-fidelity user experience is one that gives the best performance, as defined by the user or model author. A standard acceptable approximation of "best performance" is a high-resolution view, with a refresh rate sufficiently rapid to avoid distraction or eye-strain, that includes all appropriate scene nodes. There exist complex simultaneous trade-offs between those features—usually user- , model-, and platform-dependent—which this dissertation explores in detail.
- **display platform**: A combination of software, computer processor(s), and graphics display hardware.

Mathematically, this optimization problem can be illustrated as follows. Let S_W be the set of all selection states for drawing the nodes in a virtual world W. That is, for each selection state $s \in S_W$, all nodes $n \in W$ have associated with them a choice of representation r. Each node representation can be null, meaning node n is omitted and not rendered, or can be one of the r available representations in node n. $s(n, r)$, then, is the choice r for any given node $n \in W$.

The display cost of any particular selection is a function of the display platform d and the representation choice: $c(d, s(n, r))$. The total cost C for a given selection state sums across all of the scene nodes, as shown in equation III.1. The fidelity function is similar, as shown in equation III.2.

$$C(s, W, d) = \sum_{n \in W} c(s(n, r), d) \tag{III.1}$$

$$F(s, W, d) = \sum_{n \in W} f(n, s(n, r), d) \tag{III.2}$$

The optimization function is to choose a selection set s_0 such that fidelity is maximized:

$$(\exists s_0 \in S_W)(\forall s \in S_W)[F(s_0, W, d) \geq F(s, W, d)] \ \wedge \ [C(s_0, W, d) \leq T_d] \tag{III.3}$$

and cost does not pass a given threshold T_d of the display platform. Chapter VIII shows how to build a problem model from an instance of the optimization problem, and how to reach a solution using linear optimization techniques.

This dissertation postulates that Fidelity is a direct function of the quality of each representation and the importance of the object that it represents. That is the fidelity contribution f of a particular representation choice is:

$$f(n, s(n), d) = q(n, s(n), d) \times i(n) \tag{III.4}$$

where the quality function q is a factor of the node, representation choice, and display; and the importance i is a function of the object's impact on the virtual world.

It is therefore possible to optimize display and request in a virtual world given the following information:

- **Quality** rating of each representation
- **Importance** rating of each associated scene node
- **Cost** rating for rendering each representation

Hereafter this general framework is referred to as the *QUICK* model, where QuICk stands for **Q**uality, **I**mportance, and **C**ost.

This relationship implies that all scene nodes have the highest-quality representation in the case where there is no constraint from limited computational resources. When resources are limited, the greatest possible Quality can be chosen in the most Important scene nodes. Boundary cases are logical as well: for example, there is no contribution to scene fidelity by any node with the null representation or a node with zero importance, regardless of the chosen representation.

A. THE STANDARD DISPLAY PROBLEM

The *QUICK* framework is best explained by describing its application to specific problem types. The first of these is a typical display problem, with the following characteristics:

- single display platform
- model is available locally
- model fits entirely within main memory
- representations are polygonal geometry with color information
- multiple representations for a scene node are geometric Levels of Detail
- highest-quality representations of all objects can not all be drawn simultaneously
- fidelity is defined as visual accuracy

Even for the standard display problem, the computation of a guaranteed-optimal selection set is NP-complete (a proof is available in Chapter VIII). Constructing the optimization model is straightforward, given the Quality and Importance inputs. However, determining the appropriate content inputs for the display function is non-trivial. Generation of each of the three q, i, and c functional inputs is discussed in turn below, with special attention to the simple display problem stated above.

1. Quality

The quality of a representation is a subjective notion that can vary significantly between users, applications, and display platforms. It is possible to record with each representation all pertinent information about its rendered result: geometric precision, geometric accuracy, color accuracy, and so forth. These values are combined at run-time with platform-specific factors to compute the possible Quality contribution of each representation. Static platform factors, such as display hardware resolution, are determined during the program initialization phase. Dynamic factors are significantly more expensive as they must be tested repeatedly, and recomputed after any change.

Gauging the relative quality of multiple geometric level-of-detail representations is straightforward, and simple to record in this system. Quantifying the difference between functional accuracy and visual accuracy is much more complex. *QUICK* depends on subjective author annotations for such values, and provides a framework for experimentation in that open research area.

Chapter V contains a much more detailed discussion of the quality factor.

2. Importance

It is possible to reduce the complexity of a scene without significantly reducing the viewing fidelity by dropping detail only from unimportant areas. For example, in a virtual painting gallery the paintings might have a very high relative importance, while floor tiles, benches, and the like might be low. Likely a user viewing this world would ignore such accouterments anyway, and definitely would prefer that in a resource-limited situation that the paintings' nodes were the last to be degraded. Other common heuristics for detail elision, such as screen size and virtual distance, can also be included in the Importance factor. Further details on the definition and computation of Importance are available in Chapter VI.

3. Cost

In a model where each representation is a list of indexed face set polygons, an appropriate cost approximation is the number of polygon vertices. If the display platform is polygon-limited, optimization to the threshold is straightforward. A number of graphics systems have explored complex cost evaluations that include multiple related resources such as rendering hardware, texture memory, and central processing. The characterization and consumption of these resources is left to the graphics hardware community, and note that *QUICK* can easily incorporate any such approach. Further details on the cost factor are available in Chapter VI.

B. COMPLEX DISPLAY PROBLEM

The *QUICK* model is sufficient for the solution of more complex cases of the display problem as well. The complex display problem is defined with the following characteristics, in addition to those from the standard display problem:

- single display machine with entire model available
- display platform capabilities change during execution
- model cannot necessarily fit entirely within main memory
- multiple, dynamic user tasks
- representation display can require multiple independent resources
- considerable visual occlusion of model from some viewpoints

In this situation, *QUICK* factors are now multi-dimensional; for example, the resource Cost of a representation involves both its polygonal processing requirements and its memory footprint. Additionally, the resource limitations set by the display platform for those Costs also vary dynamically. For example, in a multi-tasking system, available memory might be reduced by allocations in unrelated processes. The addition of new resource constraints adds no asymptotic complexity to the optimization step, but does make the optimization formulation slightly more involved.

The major difference between the complex display problem and the previous is the allowance of user tasks that do not necessarily require visual realism. In *QUICK* , user tasks define their own computations for the Quality and Importance factors. Through this process, tasks specify what comprises a high-fidelity user experience. The *QUICK* optimization then maximizes Fidelity within resource limitations, according to the task's definition of Fidelity, without any modifications to the optimization algorithms.

A brief example of a task-specific Quality computation serves to illustrate these concepts. A color-perception task might consider color resolution the only major factor in the Quality of a representation. Such a task might compute Quality as the color depth of a representation's textures, divided by the maximum color depth, with a maximum value of 1.0. The maximum color depth is a static platform-specific factor determined by the display software and hardware. On a platform that supports only 16-bit color, the Quality of 16-bit textures would be 1.0, the same as for a 24-bit texture. Likely, the optimization would choose the 16-bit representation, since it offers the same Quality with reduced memory-storage and display complexity. Note this task ignores the issues of geometric accuracy considered paramount for a standard walkthrough application.

Further information about task definition, with more detailed examples, is presented in Chapter IV.

C. DISTRIBUTED-MODEL DISPLAY

With only minor modifications, the *QUICK* model can be used to optimize the actions of a client in a distributed graphics system. The distributed case is defined as an extension of the complex display problem, in which:

- the virtual environment definition is stored on a special server machine
- that server is different from the display platform, and is reachable by a network connection

The clients still must solve their local display problem, but now face a considerable delay between the time an unavailable representation is requested and the time it can be displayed. This distributed-cache management is essentially the same issue as that faced in the complex display problem; namely, unloaded representations arrive via some limited-bandwidth transfer path, with a (generally) predictable delay.

Supporting transfer ordering with the *QUICK* framework requires only minor modification to the optimization formulation. At each stage after initialization, the optimization process has access to the characteristics of all nodes in memory, and some nodes which have not been requested. (Chapter VII explains the process by which annotations and nodes are requested and cached in the *QUICK* software package.) The display optimization is performed as if unrequested nodes were available; their transfer costs are kept below the network capability threshold, and their storage costs are included in the primary storage allocation. Once a working selection set is generated, the missing nodes are requested. The display optimization is then repeated with only the currently available nodes; with memoization techniques, the second computation is greatly accelerated.

To support transfer ordering and optimization, Cost information must also include memory footprint and bandwidth consumption. This same information is required for objects in secondary storage; disk and network transfer paths are functionally equivalent. In conjunction with a specification of machine capability threshold, these values are used to optimize consumption of the network and disk resources. Memory footprint values are vital to local cache management, as well as for computing the cost of a cache fetch action.

IV. PLATFORM AND APPLICATION

A. INTRODUCTION

Quality and Cost cannot be computed without detailed knowledge of the capabilities of the display platform. A representation easily rendered on one platform might present a major obstacle to real-time interaction for another. The difference between two textures might be stunning on a high-resolution platform, but imperceptible in low resolution.

All applications, and adjustments to applications such as the *QUICK* optimization, are best judged by task performance. The exact user task can often be difficult to ascertain, as the user's intent may often transcend the original design of an application. For instance, a terrain-display application might be used for both mission rehearsal and for navigation training. The user's purpose is the only true means for evaluating the effectiveness of any optimization process. Accordingly, Task has a profound effect upon the input factors of the *QUICK* framework.

This chapter discusses the *Client Specification*, which contains all of the platform-specific information needed for the *QUICK* optimization process. Also included are the means by which user task defines subjective performance of an application. All *QUICK* factors can vary by platform and task, so this chapter also explains methods for encoding such data into the optimization.

B. CLIENT SPECIFICATION

Each display platform has myriad properties which dictate its ability to manage and display virtual environments. The *QUICK* optimization attempts to select a subset of the the virtual environment that maximizes fidelity and can be managed within the constraints of the given display platform. The *QUICK* Client Specification, also referred to as the *ClientSpec*, contains the details of these constraints.

The method for determining the ClientSpec values is forced by the particulars of the software implementation. Some values can be tested by the software, often by querying the operating system or the graphics library. Some values should be provided by the user; this can be done statically, in the form of start-up arguments, or dynamically as the user's tolerance for resource consumption varies.

The remainder of this section describes a set of system capabilities included in the ClientSpec, which are divided into categories of *Display*, *Rendering*, and *Storage/Transfer*. This list is not exhaustive, nor is it likely to be sufficient for all types of hardware or representation formats. However, these values have been found to offer sufficient information for the *QUICK* optimization process in the implementation described in Chapter IX.

1. Display

The Display values are those related to graphical presentation of the virtual world. The Display category specifically omits values of rendering capability, such as polygons per second, that are affected by the complexity of chosen representations. Instead, these values describe the capability of the hardware display device, its drivers, and its current settings. These constant values can affect the rendering pipeline; for example, monitor settings with high color depth can significantly slow rendering. Not all Display values are static; for example, display resolution is affected by the virtual field-of-view, which some applications change during program execution.

The Quality chapter explains how many of these values are used in the Quality computation (see section V.D.1).

a. Display Resolution

The hardware display resolution sets the upper limit for useful precision in the virtual environment. This is particularly useful when computing the Quality of a representation, because often the screen resolution will be too low for noticeable differences between two high-precision representations.

This value can be stored in many formats; the most useful thus far has been a ratio of screen pixels to the field-of-view angle, in both horizontal and vertical directions. The window size in pixels is stored in the client specification, and the display resolution is recomputed whenever the viewing field of the virtual environment changes. That ratio is compared at run-time with the precision of a representation and its subtended screen angle. The lower ratio of the two is chosen for the Quality computation.

This formulation is not dependent upon the type of display device. Head-mounted displays and monitors have similar viewing characteristics, except for the distance between pixels and the eye. For small pixels, human eye precision can be inadequate; in such cases, it is appropriate to include viewing distance and pixel size as a similar ratio.

b. Display Update Rate

Modern display hardware updates the screen at a constant rate, regardless of the graphics processing pipeline. *QUICK* assumes that a double-buffering solution is applied to allow construction of an image across multiple frame updates. The display update rate is stored as the maximum possible refresh speed; drawing the scene graph more quickly has no visible effect.

c. Stereoscopy

The ability to present stereoscopic image pairs offers a more immersive sense of three-dimensional object placement, usually at the trade-off of halving the display update rate. This value does not present a platform constraint; rather, it is included to specify a platform's capabilities. A review of the benefits of stereoscopy in virtual environments is available in [Hodges, 1992].

d. Color Depth

The Color Depth value reflects the current display settings for color resolution. The value is stored as an integer three-tuple which holds the number of bits of precision for red, green, and blue color values. When determining Quality, representations with color precision greater than the display platform are limited to the platform specification.

e. Alpha Depth

Most displays restrict the precision of transparency settings, similar to color depth. This value stores the number of bits of precision available for declaring transparency, and is treated similarly to Color Depth for Quality computations.

2. Rendering

The Rendering factor includes values that reflects a platform's capability to display virtual environments, especially its ability to scale to larger data sets. These values are usually determined in a preprocessing stage by evaluating performance over a series of computational and display tasks. Performance benchmarks are a well-explored area; standard benchmarks are available from organizations such as the Standard Performance Evaluation Corporation.

Chapter VIII explains how these Rendering specifications are used, in conjunction with Cost computation, for the optimization process.

a. Polygonal Rendering Performance

Certainly the single most important display platform is its capability to render geometric primitives. The fact that this value is constrained, and usually beneath the amount needed to display complex scenes at interactive rates, is a primary motivation for the *QUICK* system.

Polygonal performance can be measured with industry standard benchmarks such as SPEC viewperf and SPEC glperf (SPEC benchmarks are available online through http://www.spec.org). Alternatively, this value can be a fixed value representing the number of primitives that can be drawn at an acceptable frame rate. Such values can be determined empirically with simple test programs by choosing a target frame rate and increasing scene complexity until the target is missed. Initialization in the *QUICK* implementation offer similar functions that can be executed at run-time, but their accuracy of course is lower than that available in full test suites.

Because rendering performance and frame rate are so central to the optimization, the user will frequently desire more direct control of those constraints. The user interface in the sample implementation described in Chapter IX includes sliders for interactively adjusting the maximum allowable polygons. In this way, the complexity/speed trade-off can be made much more accurately.

Depending on hardware characteristics, rendering performance may require division into subcategories. For instance, image texture processing capability might be best treated as its own system constraint. The *QUICK* test implementation uses a single value for Rendering Performance, and it has proven to be much more effective than competing scene management systems (as shown in Chapter X).

b. Computational Performance

All display platforms offer general-purpose computational resources in addition to the graphical rendering pipeline. While traditional polygonal representations are usually fed directly to the graphics pipeline, other representation formats can require preprocessing. For example, fractally-defined geometry requires dynamic computation of appropriate detail. First, this value indicates the number of physical processing units. Second, processing performance is be measured with standard benchmarks such as SPEC CPU2000, which measures floating-point and integer operation performance. While those benchmarks are proprietary, results for almost all hardware/operating system combinations are publicly available. Similar to polygonal performance, the *QUICK* implementation includes initialization functions that can test computational performance dynamically with reduced accuracy.

The *QUICK* optimization treats processor and polygonal performance as independent values. This is a deliberate over-simplification; most platforms use the main processor in the graphics pipeline for geometric transformations and lighting. Fortunately, commodity graphics hardware designs are evolving towards a “graphics processing unit” in which all rendering-related functions take place in the graphics subsystem.

3. Storage/Transfer

The Storage/Transfer values represent a platform’s performance as a node in a distributed cache system. These values reflect the capability for retaining objects in the local cache, whether in memory or on disk, as well as the capability to move objects between those caches and networked repositories. While these values can remain static for simplicity, network conditions and available memory will often change during the execution of an application. Still, a static configuration file with average values is often sufficient.

Chapter VIII explains how these specifications are used as limitations in the optimization process.

a. Available Disk Storage

Disk space usually far outstrips the size of virtual environment models, so the available file cache size is rarely a constraint. However, for very long-lived or complex scenes, this can be a concern. Disk space must be considered a dynamic value. In multitasking operating systems, such as Windows and UNIX, other processes (or even other computers) may be sharing the disk storage resource.

b. Available Memory

The price of memory modules has dropped significantly in recent years, with a resulting increase in the capacity of main memory in the average workstation. Conveniently, growth of virtual environment model descriptions has out-paced that capacity increase, leaving a need for cache management systems like *QUICK*. To optimize request and deletion of representations, the *QUICK* optimization must have up-to-date information on memory allocation limitations—especially in multiprocessing systems, in which memory availability is particularly volatile.

c. Latency to Server

Latency information is critical when making prediction-based object requests, as the accuracy of prediction techniques usually drops exponentially with time (see section VI.C for more detail). While this value is included in the client specification, it is difficult to consider without representation-specific information. In the worst case, each representation is served from a different network location with individual network delay. In the optimal case, servers containing representations being considered for request could be *ping*ed for latency. Since limitations on network bandwidth usually affect latency more than round-trip communication times, a single average network delay value has been sufficiently accurate in practice.

d. Available Bandwidth

The client specification includes the available network bandwidth, in both directions, from the display platform to the Internet. This value is necessarily myopic in scope, since network throughput between client and server is usually limited by the lowest-bandwidth connection on the path between them. Determining current throughput between two points on the Internet usually requires more traffic than a representation transfer, so such detail is only useful on a frequently-accessed server. The Total Entertainment Network, a closed client-server system, used such evaluation techniques to improve networked game interactivity.

The Available Bandwidth value can also include internal bandwidth, especially between the secondary and tertiary cache (main memory and disk storage). While internal bandwidth is usually not a factor in networked virtual environments, it should be considered when navigating large local datasets that require significant paging. The Berkeley Walkthrough offers an excellent introduction to the issues involved in disk database management [Funkhouser, 1996].

C. DYNAMICISM OF TASK

User task is both highly variable and highly subjective. The *QUICK* framework is able to capture that variability in the virtual environment optimization process. This section shows that user task and intent cannot be extrapolated from knowledge of the virtual environment world model, or even of the application interacting with that model.

```
Define QIC for Lamp:

switch (Task) {
  case Hide-and-seek: {
    set Quality = q’
    set Importance = i’
  }
  case Lighting-visualization {
    set Quality = q’’
    set Importance = i’ * .5
  }
Cost = c
```

Figure 7. Task-based step-function technique.

A virtual environment model can be used for a variety of user tasks; examples abound. For example, SGI’s Performer library is packaged with a city model, known as PerformerTown. That town, and its derivatives, have been used for performance testing, vehicular-navigation training, and even large-scale military exercises. This reuse is even more prevalent with smaller graphical models: a lamp designed for a VRML virtual office design program might well be found populating databases used for a variety of other applications.

Originally, a task-based step-function approach was considered, as illustrated with the pseudo-code below. In such an approach, every virtual object contains different *QUICK* annotations for each planned task. But the reuse patterns of objects indicate that it is not always possible to know all tasks for which a model might be used.

A second approach considered was to break down each task into component parts, and define *QUICK* factors for each of those components. A given task might, for example, be a mix of “fast fly-through” and “precision targeting”. Brief exploration was convincing that no such breakdown is likely to exist; and, if those categories were to exist, they would likely be analogous to the standard *QUICK* factors themselves.

It is evident that a single virtual object model can be used in multiple applications, and therefore, for multiple tasks. Additionally, a single application may be applied to multiple tasks, and those tasks may change during a single incarnation of the application. Complicating matters is the fact that only the user has an accurate understanding of task at any given instant—and that the user may be engaged in more than one task at that instant.

The goal of *QUICK* is to optimize with respect to the current task. The first step towards that goal is to inform the optimization system constantly of that task. Since only the user has that information, the application must provide an interface for the capture of the tasks and their priority. It is generally possible, in designing an application, to presuppose what general tasks it will enable; a list of those common tasks is then included in the interface. Certain classes of applications might simply force task changes, without direct user input; for example, a plot-point in a computer game might necessitate a change in task from “navigate” to “avoid detection.”

The second step towards the optimization goal is to use tasks in asset prioritization. The next section gives examples of how task might modify quality and importance factors.

D. TASK COMPUTATION

In the *QUICK* system, the Task (note capitalization) is defined as an algorithmic representation of user preferences and application priorities. The current value of each *QUICK* factor (Quality, Importance, and Cost) is computed at run-time as a combination of model annotations, application state, and platform state. The algorithms for this combination process are defined within the Task specification.

An explanation of how this fact is incorporated into the optimization computation must wait until the *QUICK* factors and optimization are explained in following chapters. However, it is still possible to justify the discussion of task via anecdotal evidence. The following two sections illustrate the reliance of quality and importance upon task.

1. Task and Importance

A change in task is most noticeable with the Importance metric. Importance reflects the contribution to fidelity that can be made by any virtual object. When a task does not require a given object, its presence or absence has little impact on fidelity and consequently the object has equally little importance.

A virtual museum yields an excellent example in which task can have tremendous impact upon Importance. A likely task would be a sight-seeing walk-through of the museum's various exhibits. In that case, the user would require high-fidelity viewing of (for instance) colonial furniture exhibits, while other patrons of the museum would have no importance to the task. A switch of task to finding an art thief would likely invert that relationship; suddenly, detail of the museum patrons would be essential, and the furniture is needed only for its properties of visual occlusion. It is clear that properly generating the Importance of scene objects requires current information on user task.

In most systems, the Importance of a scene object is based upon simple heuristics such as distance from the viewpoint or the area of pixels the object subtends. (Chapter VI will demonstrate that these techniques alone are insufficient.) Task is the factor such heuristic-based systems ignore. In the case of distance, a sniper training exercise would likely rank a faraway target as far more important than a nearby rock. Similarly, for pixel-area, a virtual bird-watcher would find a small bird on a tree limb much more important than the much larger tree. Yet a system such as Performer would prioritize geometric detail for the tree under the assumption that fidelity is most easily increased with large objects. Clearly, task overwhelms factors such as distance and screen-area subtention.

2. Task and Quality

Quality is also dependent upon user task, though in a manner that is both less noticeable and less suitable for computation. As in the previous section, this dependency is demonstrated by giving examples of tasks which would the invert priority ordering implied by standard heuristics. For instance, the real-time rendering engine in the forthcoming PC video game “Vampire” uses multiple representations for anthropomorphic figures. Representation choice is made based on using simple distance to determine Importance, and polygon count to determine Quality. Low-polygon models in this system assume an anterior view, so special care is given to keep that view constant across the various representations. (This assumption is valid for general game play, wherein anthropomorphic characters usually face the player.) The slightest change in the user task invalidates the polygon-based Quality value. For instance, a task such as silhouette identification (from all perspectives) would require most low-polygon models have a Quality of zero, since their silhouette information is not only imprecise but occasionally fully misleading.

Misleading information seems to be a theme in task-adjusted Quality ratings. Most virtual environment systems equate visual realism with fidelity, and therefore assign highest Quality ratings to those representations with the most visual complexity. But in some cases, there is an unintuitive need for less-precise models. For instance, research at the Naval Postgraduate School [Goerger, 1998] has shown that visual detail can have a negative impact on some training tasks; mental correlation between virtual representation and real object can be confounded by misleading precision. That research found that, at least for a virtual environment of a real space, that the use of inaccurate high-detail models to represent real-world objects caused confusion in the user's ability to correlate virtual and real objects.

These findings imply that fidelity can stem from symbolic representation as well as realistic presentation, which points to the need for some codification of the purpose an object serves in a virtual world.

E. ONTOLOGICAL REPRESENTATION

The previous sections of this chapter demonstrate the need for task-specific adjustment of *QUICK* factors. Hard-coding all possible tasks into a virtual world description is not a candidate method, as it is impossible to extrapolate all user tasks for which any virtual object will be used. In fact, it is equally foolhardy to presuppose all future uses for a single virtual environment application. (In the case that an environment is designed expressly for a particular purpose, task information can be included, but this should not interfere with general use.)

First it is assumed that the application can determine the current user task(s), or be informed by the user of the task(s). This puts the responsibility on the application to query virtual objects about their function, such that task-based adjustments to Quality and Importance can be made. For this reason, it is necessary to include a virtual object's functional definition in its description.

Functional definition requires a precisely defined common terminology; the combination of terminology and definitions is known as an ontology. This is the well-explored area of knowledge representation, and is generally acknowledged to be unsolvable except in limited domains. The *QUICK* framework makes no claims to original work in ontology, but rather is designed to incorporate outside research with ease. There exists excellent prior work, such as the Stanford Knowledge Systems Laboratory [Farquhar *et al.*, 1995] online ontological databases, and a recently proposed ontology for virtual world objects [Soto and Allongue, 1997], that can and should be integrated.

In the *QUICK* proof of concept system discussed later in this thesis, virtual object files include a simple array of zero of more textual descriptions. For example, a virtual apple object might include:

```
Plant:Tree:Fruit:Apple
MassedObject:0.25kg
Food:Fruit:Apple
```

This information is used by tasks to adjust *QUICK* factors; for example, a “foraging” task might increase the Importance of all Food objects. This simple mechanism is sufficient for demonstrating the need for task-based asset prioritization, though plainly would need to be replaced before for general-purpose use.

The Extensible Markup Language (XML) was designed for conveying structured data [Consortium, 1998]. As explained in Chapter II, the X3D graphics format is based upon XML. There exists an opportunity to integrate an XML-based ontological system into X3D object descriptions, which could then feed directly into the *QUICK* optimization.

F. SUMMARY

The capabilities of the display platform dictate both the resources available for presentation of a virtual environment and the limitations on precision of perception. Therefore, *QUICK* includes a mechanism known as the client specification, or ClientSpec, for defining those capabilities.

Fidelity is not always defined by visual accuracy; a user may prioritize objects or presentation differently, based upon their goals for the application. In the *QUICK* framework, this profile information is stored in the Task. The Task contains the algorithms by which the current Quality, Importance, and Cost are computed from available annotation and application state information.

V. QUALITY DETERMINATION

A. INTRODUCTION

This chapter provides a more detailed description of the composition and computation of the *QUICK* Quality factor. This discussion is limited to the visual domain, as that is the primary media for virtual environment clients, but *QUICK* should be equally applicable to other media.

This chapter begins with an annotated list of the Quality factor components. The next section shows how Quality is computed, by integrating specifications of the display platform, application task, and application state. This also includes a discussion of relative and absolute Quality, and the problems with building a virtual world with representations from heterogeneous sources.

The Quality computation can be greatly complicated by inter-representation interaction. While such issues are specifically excluded from the initial *QUICK* implementation, they are explored briefly at the end of this chapter for completeness.

B. RELATIVE VS. ABSOLUTE QUALITY

Outside of this optimization, the term “quality” is generally applied as a relative measure between two comparable items. In the *QUICK* system, the quality factor must serve as both absolute and relative measure. If only one can be eaten, apples and oranges must be compared; the fruit chosen should be that most appropriate to the situation. Any comparison between two apples would certainly be simpler, but both comparisons can be performed in deterministic fashion if the needs and tastes of the diner are known.

In graphical terms, the Quality factor is applied in two ways. First, given two representations for the same object, the higher fidelity representation should have a higher Quality rating. Second, given two representations for different nodes, the most appropriate representation should have a higher Quality rating. The techniques for computing that Quality rating, incorporating application task and display platform, are discussed in the remainder of this chapter.

For the first task, comparing two representations for the same object, it is reasonable to suppose there exists an objective test for determining relative accuracy. However, this is only the case if the two representations are labeled in quality order. That is, if representation 1 is labeled of higher quality, then the quality of representation 2 should be a factor of its deviance from representation 1. Without an *a priori* ordering, the determination is impossible; though one representation may have higher precision, or greater Cost, it is not necessarily more accurate. Fortunately, most secondary representations of models are generated from an original by repeated application of polygonal simplification techniques. Therefore, advance knowledge of the most accurate representation is rarely required; for homogeneous representations , accuracy generally increases monotonically with Cost and precision.

C. QUALITY COMPONENTS

The Quality factor describes the visual accuracy of an individual representation of an object. Representations with average Quality are those that adequately describe the intended object. Low Quality representations give only a general impression of the object, or include significant error. High Quality representations are the best available visual descriptions, and often contain original data. Two representations with equal Quality are implied to be interchangeably appropriate for the given application. Frequently, equality is an indication that the human eye cannot discern any differences between them on the given display platform.

When describing a representation, values generally fall into two categories—those that record the precision of the representation, and those that record the accuracy of the representation. (Precision ¿ considered as the total amount of information available, and accuracy ¿ only being the significant part of that information.) Quality components originally incorporated values from both categories, but it has since been determined that only accuracy values are needed. When comparing a certain facet of two representations, the precision has no bearing except when it limits denotable accuracy. When computing Quality for a certain display platform, the issue is not whether the platform can convey all of the precision information in the representation. Rather, the task is to determine whether the platform can convey all of the *significant* information in the representation. Precision information is indirectly recorded in the Cost factor (as discussed in Chapter VI) since additional precision is usually reflected in higher representation Cost values.

It should be noted that this is not an exhaustive list, but rather an acceptable generalization for the subset of representations used in the initial *QUICK* implementation. representations. Many changes and additions to this list will likely be required as different representation types and platforms are incorporated into *QUICK* .

The Quality information for a representation includes the following components:

1. Geometric Accuracy

The primary metric for Quality of standard representations is geometric accuracy. This component reflects the spatial difference, if any, between two representations. It consists of two values: the average error for any point on the surface, and the standard deviation in that error. Both error values are recorded in meters. Meters are the standard unit for most web-based graphics formats, and nearly all other formats provide conversion routines that yield data in meters.

Measuring the error between two geometric models can be a time-consuming procedure. Likely the best method is to avoid measurement altogether and create levels of detail with known accuracy values. Many Level of Detail generators, such as the Simplification Envelopes algorithm [Cohen *et al.*, 1996], accept the geometric error tolerance as a parameter.

Complete analysis of geometric error for externally-generated representations can be intractably difficult, as it requires total matching between distinct topologies. Instead, error is usually accomplished by subset sampling, either using a fixed number of points or enough points to generate an acceptable estimate of error. One method is to choose a set of characteristic points on both representations and to determine the point-wise differential in their positions, similar to the geometric fiducials of Talisman [Lengyel and Snyder, 1997]. Matching characteristic points on both surfaces usually requires either human intervention or *a priori* knowledge of the generating algorithm.

While there are techniques to determine geometric error without a human in the loop, they are useful in only limited cases. One method is to cast a ray radially outward from the center of each representation and determine the distance at which the object surface was crossed. (For concave objects, or those of genus greater than 0, multiple crossings might occur.) Differences between the intersection distances for the two representations would indicate geometric error. This can indicate false error unless all differences between the two objects are radial. In Figure 8, point Q has been deleted in the lower-detail representation; the error distance on the (dashed grey) radial arrow shows a significant error distance. However, the desired value distance is shown magnified in the rightmost figure.

This suggests the possibility of measuring average surface distances, rather than radial error. Sample points on the surface of one representation are selected randomly, or distributed evenly using a relaxation algorithm similar to that in [Turk, 1991]. For each point, the distance to the closest surface in the other representation is computed. Those values are averaged to yield the geometric error. This method generally yields more reasonable results than ray cast sampling. However, it can miss large errors by corresponding a point with an incorrect surface, as shown in Figure 9.

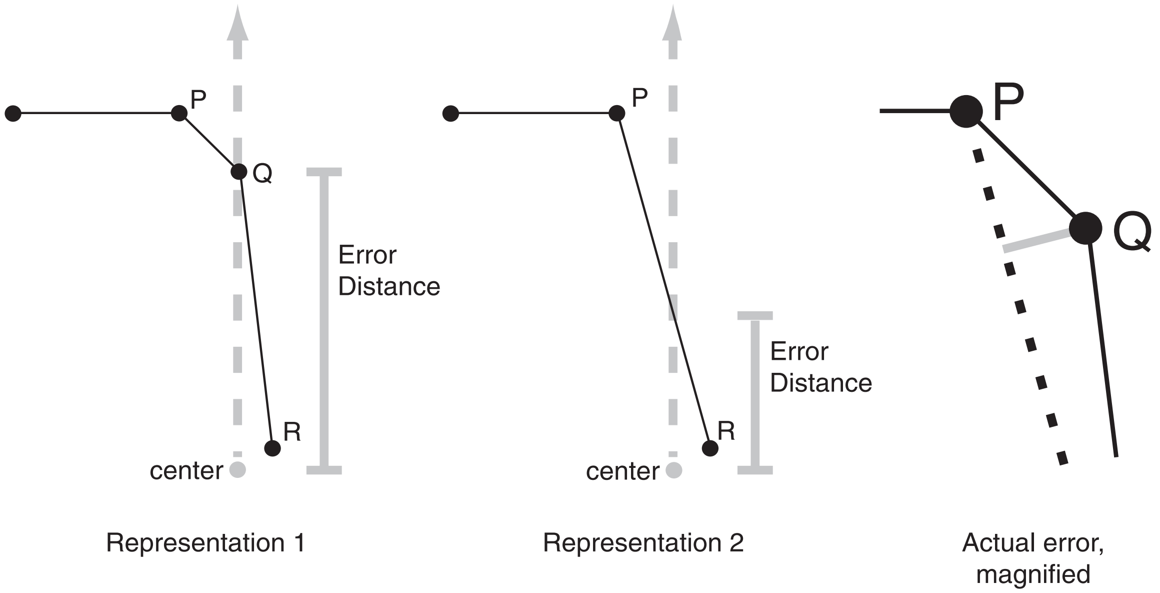


Figure 8. Error calculation using radial sampling.

2. Color Accuracy

Geometry has no intrinsic visual description; geometric surfaces generally have an associated coloration. That color can be specified with widely varying precision, usually with between 2^2 and 2^{32} possible values. That precision is an upper bound on the accuracy, which can often be less than available precision. Depending on the authoring technique, a

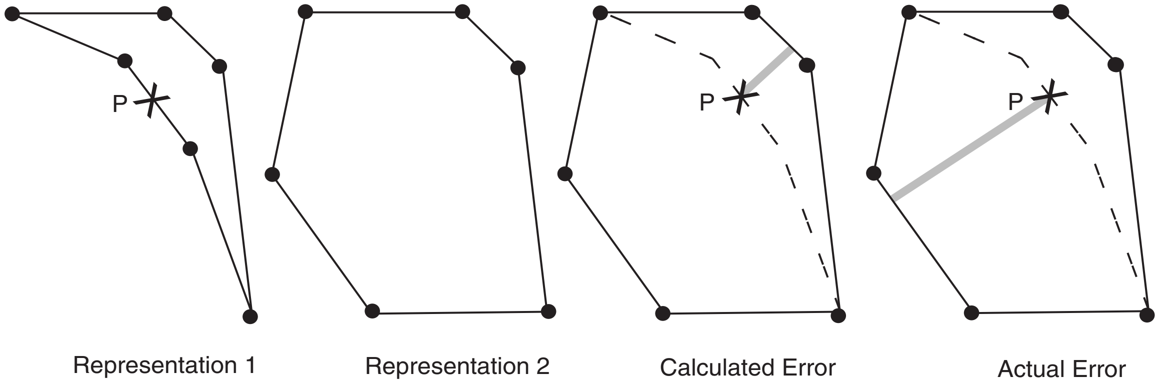


Figure 9. Error calculation using surface distances.

high-precision color may be down-sampled into a smaller color space, or a low-precision indexed color may be translated into a larger color space.

The Quality annotation contains an integer value for color depth. This specifies the number of bits of color accuracy, and is independent of (but bounded by) color precision. Similarly, the annotation contains an integer value for alpha-channel depth, which specifies the number of bits of transparency accuracy.

3. Texture Resolution

Color can be replaced or blended with image textures to give the impression of additional geometric detail. The resolution of such textures is an important factor in the visual quality of a representation. This value is stored in the Quality annotation as a single integer, the number of pixels in the texture image. In the case of multiple-resolution textures such as a mip-map, the highest resolution is used. If multiple textures adorn a single representation, the pixel count for the lowest-resolution image is used.

4. Subjective Quality

While the above (and other) values can measure model accuracy, they cannot always convey the subtle differences in visual impact between two representations. This indicates there is not always a direct relationship between geometric accuracy and representation Quality. Research such as the view-dependent geometry project [Rademacher, 1999] shows that accurate geometry can in some cases even reduce display fidelity. Artists build careers around the process of conveying information, and it is impossible to capture that knowledge

Representation:	1	2	3	4
Triangle count:	603	1184	1816	2360
Avg. Geometric Error:	.189m	.169m	.051m	0m
Subjective Quality	65%	90%	95%	100%

Table I. Subjective quality for the “truck” representation set (see Appendix B.

in a handful of numerical values. Extensive research has been performed to determine the capability of the human eye and brain to process visual information—which has shown that visual capabilities can vary extremely depending on the nuances of situation. For example, minor differences in color accuracy can be both obvious and impossible to detect, depending on the portion of the color spectrum and the luminosity [MacDonald, 1999].

Given this, it has been convenient in practice to incorporate human judgment into the Quality factor. A single floating-point value is inserted into the annotation which reflects the author’s estimation of the “visual perfection” of the representation. Traditional LOD management systems behave as if the cost ratio between two representations dictates the Quality ratio. However, an object can often be adequately described with significantly less detail, and the Subjective Quality value can be useful in that situation. Table I shows an example set of LOD representations for an object, with Subjective Quality values included.

One major drawback of subjective labeling is consistency among model authors, which is needed when constructing virtual environments from distributed sources. This limits its utility in the distributed case. Still, on display platforms with few technical limitations (e.g., a high-resolution, true color display) this percentage value has been sufficient for use as the final Quality value with no computation. This experience is discussed further in Chapter X.

D. COMPUTING QUALITY

This section describes the process by which the Quality value is computed. Annotation values alone can be adequate for determining the actual Quality of a representation—not unlike a clock that is correct twice a day. In the general case, however, factors external to the description of a virtual environment can have significant influence upon the perceived Quality.

Each Task includes its own algorithm for computing Quality as a function of the annotation values, client specification, and application state. Most Tasks assume a human sensor, so the Quality determination frequently includes human capability as a factor, which is discussed below.

1. Platform and Human Factors

Most visual-quality metrics are specific to a certain display platform. For instance, while doubling the resolution of an image would normally have a significant impact on perceived Quality, there might be no noticeable difference between a high- and low-resolution texture on a low-resolution display device. Systems such as head-mounted displays typically offer low screen resolution, and therefore additional geometric detail may offer little benefit.

The practical result of this is that when computing Quality, the annotation values are modified for the display platform. For example, if the geometric accuracy for a representation is higher than can be detected with the resolution in the ClientSpec, the accuracy is reduced to reflect that limitation. Similarly, the color accuracy annotation is limited by the color depth of the display; there is exactly zero visual difference between representations accurate to 24 or 32 bits when the display supports only 8-bit color.

Similarly, human capacity for detecting color and detail offer additional upper bounds on the amount of useful representation detail. In general, available display technology rarely is able to present detail undetectable by the human eye. However, one can envision a high-resolution display presented at a large distance from the eye, such that the ability to resolve detail is constrained not by the screen resolution but the visual angle. Another example is detection of color variation; if the human threshold is less than the difference in color accuracy between two representations, then that difference is not a factor in their Quality difference. Human color variation detection thresholds vary significantly by the spectral qualities of the color. In general, these constraints are not needed for Quality computation due to hardware limitations. For more information on display design for the human eye, see [Banks and Weimer, 1992].

2. Task Factors

Each Task includes its own algorithm for computing the Quality value, because different Tasks may have widely different needs in a representation. For example, while a representation with high-resolution texture and simple geometry may be considered high-quality for a predominantly visual task, it would be nearly useless for a Task requiring highly precise haptic feedback. Another Task might raise the computed Quality for representations modeled in a certain theme—for instance, those labeled “Cartoonish”—that matched the application. Section IV.D.2 addressed in more detail how and why Tasks might influence a given Quality computation.

3. Dynamic Factors

There is no general correspondence between geometric accuracy and screen resolution. These data must be related with a geometric transformation between the virtual environment space and screen space. That information is only available during the execution of an application, based upon the eye position in the virtual world. Therefore, for proper incorporation of screen resolution, Quality must be continuously recomputed at run-time.

Distance attenuation of Fidelity is incorporated into the default computation for Importance (see section VI.C). Therefore, distance-sensitive computation of Quality is often omitted in the default Quality computation.

E. HYSTERESIS

The Quality of a representation can also be affected by its spatial and temporal interfaces with other representations. For instance, the well-known hysteresis effect occurs when swapping between representations of a scene node—even between various LODs of geometry. Popping between low and high detail versions can be detrimental to the user experience, *even if the change results in greater view realism*.

The interface in space is equally important to the user experience. Two scene nodes that join seamlessly in an original high-resolution version will likely have distracting discontinuities if presented in varying resolutions. The discontinuity is even more pronounced if the representations are of varying form, for instance, when a building is drawn with a geometric half and a warped depth-image half. Proper division of a model into scene nodes can ameliorate this problem in some instances, but rarely in all possible instances.

The Quality of each representation can be adjusted dynamically based on its interaction with other representations. Issues such as thrashing, where an object oscillates between two representations, can be prevented by increasing the Quality of the currently selected representations. Unfortunately, the optimization process is already NP-complete (see Chapter VIII); incorporating Quality changes based upon previous or neighboring representation selections would increases the optimization complexity tremendously.

VI. IMPORTANCE AND COST DETERMINATIONS

A. INTRODUCTION

This chapter gives a detailed presentation of the Importance and Cost *QUICK* factors. These factors are presented together because their specification and computation is considerably less complex than for Quality. In fact, the Cost computation rarely includes any application-specific or dynamic factors, and is based purely upon the platform specification. Similarly, the Importance computation is only rarely affected by the display platform, instead relying on the state of the virtual world.

For each factor, this chapter first presents the components that make up the factor. It then shows how a Task combines those components (with application state and display platform where appropriate) to compute a single final value. When no Task is specified, the default computation is used; each factor's default algorithms are explained here. Finally, the annotation and computation processes can often be automated, and so each factor's description concludes with suggestions for that procedure.

B. IMPORTANCE COMPONENTS

The Importance factor describes the impact an object has upon a virtual world scene. An object with very low Importance has little effect upon the overall Fidelity of a scene, so therefore unimportant objects are usually represented by low Quality versions. Objects with high Importance are essential to the integrity of a scene, and therefore are usually represented by the highest Quality possible.

In *QUICK* , the Importance annotation for an object is given as a single floating-point number between zero and one. That value represents the relative Importance of an object within a world, with one being the highest possible value. No single absolute value indicates “important” or “not important”; rather, it is the difference between Importance values that impacts optimization selections for a scene. The value is clamped in the range $[0..1]$ to simplify the computation of Fidelity. Since Fidelity is computed by multiplying Quality and Importance together, objects with zero Importance offer zero Fidelity no matter the Quality of the chosen representation.

It is intended that the chosen Importance values be consistent throughout a virtual world. However, there are no facilities for normalization in the case of independently-authored world components. The default value for Importance is .5; recommended practice suggests that Importance values follow a bell curve distribution around .5, with standard deviation of .1, to ensure that extreme values are very rare.

C. COMPUTING IMPORTANCE

The annotation described above makes up just one part of the final Importance value. Similar to the Quality factor, a number of issues external to the world description can influence the Importance computation. While the platform capability plays only a small role, the application task and state quite nearly obviate the need for any Importance annotation. In fact, while the Importance computation is the simplest of the three *QUICK* factors, its significant dependence upon dynamic application state information makes it the most costly computation in terms of run-time system resources.

The major contribution to Importance comes from the application Task, combined with the ontological object description. This reflects the fact that the information which is essential to the user varies by application task. (An explanation of these issues, with example scenarios, is available in Chapter IV).

Each Task uses its own algorithm for combining object description, annotation, and application state to compute Importance. The following section describes the dynamic application state information which is made available by the *QUICK* library for that computation.

1. Dynamic Factors

The spatial arrangement of objects and viewpoint in a virtual world has a major impact on the Fidelity contribution made by any object. Most LOD management systems depend solely upon spatially-based heuristics to make representation decisions. *QUICK* makes the results of similar computations available to the application so that they can be combined as appropriate for the current task. This section explains how each of those variables is determined; the Task defines how these variables are combined in the Importance computation.

a. Distance Attenuation

Simple LOD management systems, such as VRML and Java3D, use proximity as the sole measurement for object importance. Traditionally, LOD node definitions include a series of distances that indicate which representation should be chosen, as shown

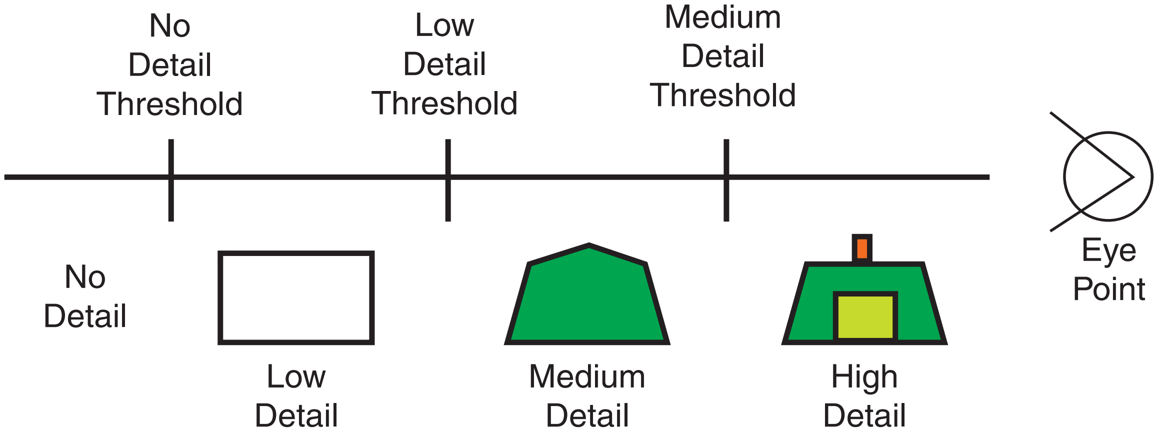


Figure 10. LOD selection by threshold distance.

in Figure 10. When the object is less distant than the first distance, the highest-detail object is selected; as the object moves farther from the viewpoint, representations with less detail are selected. This mimics the real-world effect of angular resolution.

These arbitrary distance settings are constant regardless of task or surrounding virtual environment. While such techniques have proven adequate for a singular purpose, they negatively impact the composability of virtual world content. (A full comparison of *QUICK* and traditional resource management systems can be found in Chapter X.)

Essentially, the desired outcome is attenuation of Importance over distance. This attenuation can be modeled with a step function, as in Figure 10, or as a continuous polynomial. The Virtual Planetary Explorer project [Hitchner and McGreevy, 1993], for example, determined importance by summing the square of the distances from certain fiducial points.

In *QUICK* , the distance attenuation function is incorporated in a Task definition rather than embedded in each object description. Tasks can query the current distance between an object and the viewpoint, and then adjust the Importance as desired.

b. Screen Position

The difference in acuity in the human eye between foveal and peripheral perception is striking. Rich Gossweiler's dissertation [Gossweiler, 1996] included a framework that used such psychophsycial metrics to make decisions of rendering complexity. In the absence of eye-tracking hardware, that systems and others generally assume that eye focus is on the center of the screen and optimize appropriately. Accordingly, *QUICK* offers functions to determine screen coordinates for virtual objects. Without eye-tracking capabilities, this information is rarely useful and is therefore omitted from the default Importance computation.

c. Subtended Screen Area

Distance attenuation attempts to reflect the change in subtended visual angle caused by object motion. However, it does not account for the fact that objects can vary significantly in size. For example, an object at distance $2d$ with view-perpendicular cross-section size $3s$ subtends 1.5 more visual angle than an object at distance d with cross-section length of s (see Figure 11).

Arguably, large objects make a significant impact upon the fidelity of the scene, regardless of their distance from the viewpoint. Of course, the cost of displaying those objects is equally significant, especially in display platforms limited by pixel-fill. The *QUICK* system is able to determine the number of pixels covered by an object (or, more cheaply, the object's bounding volume) if that information is required by a Task.

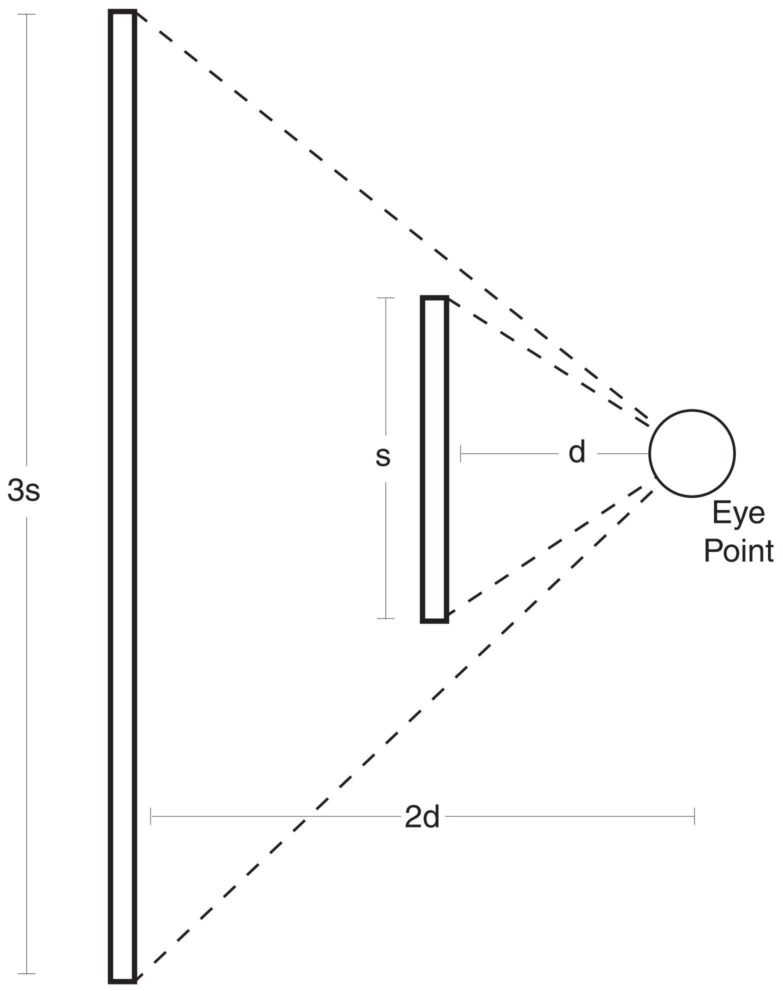


Figure 11. Importance effects of size can outweigh distance.

d. Visibility

Using visual occlusion to reduce graphics processing load is an active area of research in computational geometry. Determination of visibility is a complex operation (general-form exact visibility is considered to be an $O(n^9)$ problem). Therefore, point-to-object visibility is often determined in a precomputation stage, such as was used in the Berkeley Walkthrough [Funkhouser and Séquin, 1993]. In the *QUICK* model, such informatin can be used by adjusting a node's Importance if it is occluded.

It is worthwhile to note that most existing visibility engines return only boolean information, stating simply whether an object is or is not visually occluded. Values of a continuous nature would be more effective in combination with *QUICK*. For instance, when appropriate to a Task, an object's Importance could be multiplied by its visibility; an 80% visible object would have its Importance reduced by 20%. Any such opportunity to add information to the *QUICK* inputs invariably results in added expressivity for application Task programmer.

The Graduated Visibility Set (GVS) determines a "percentage" of visibility between two spaces in a model [Capps and Teller, 1997]. The GVS could be adapted for inclusion in the *QUICK* framework, though it is best suited for virtual environments in which the set of possible viewpoints is constrained.

Occlusion determination is often used in conjunction with visibility culling, which is significantly less expensive to compute. Most modern graphics hardware incorporates frustum culling, in which objects are culled if they exceed a distance from the eyepoint or are outside the viewing area. View-frustum culling is usually excluded from Importance determinations because changes in viewing direction can occur more rapidly than optimization passes. However, facilities are available for determining whether an object is within the viewing frustum.

e. Motion Prediction

Optimization in *QUICK* is used both for display decisions and representation request decisions. While a change of representation choice is evident within at most two frame display cycles, a representation request may not be evident for considerably longer. A request incurs round-trip network latency to begin the transfer, and then the transfer itself is constrained by available network bandwidth. The representation file is parsed into a scene graph in memory, and that graph is attached to the virtual world between draw traversals. For large representations requested over a poor network connection, this delay can take seconds.

By the time a requested representation arrives, it may no longer be pertinent, and in fact never be selected for display. In that case, the memory, network bandwidth, and parsing resources have all been wasted—hardly an optimal strategy. The standard approach for avoiding such wasteful operations is to request representations such that they will be needed at the time of their arrival. This requires knowledge of the optimal world state at a time in the future, which requires a prediction algorithm.

Prediction of world state can be performed with varying degrees of accuracy. For an animated path, the prediction can be made with perfect certainty. Constraining the possible paths in a virtual environment increases prediction accuracy. Controlling the intrinsic navigational motion range (velocity, acceleration, and rotational velocity and acceleration) has a similar effect. The Berkeley Walkthrough system allows only human-range motion, inside an architectural space, so tolerable motion prediction was possible. Even with such constraints, accuracy of motion prediction techniques usually drops exponentially with increasing time, due to the ever-increasing space of options.

In the *QUICK* framework, motion prediction can be used when determining Importance, as that value is highly proximity-dependent. As discussed above, motion prediction is a function of both the virtual environment and the navigation method, and general-purpose motion prediction techniques are generally not useful. Therefore, all motion prediction models are incorporated into specialized Tasks, and then used when computing distance attenuation and visibility for Importance.

2. Default Computation

It is strongly suggested that application programmers write Task specifications for each significant use of their application. The *QUICK* framework offers a standard Task that offers reasonable performance for general-purpose applications. The default computation for Importance is straightforward: the annotation value is modified for object distance only.

The Importance value I is computed by:

$$I = i * \left(\frac{far - d}{far} \right)^2 \tag{VI.1}$$

where i is the annotated Importance value, far represents the far clipping distance, and d is the object's distance from the eyepoint.

The other factors discussed above are not incorporated for a variety of reasons. Screen area is closely related to distance, and should therefore be needed only for special purpose tasks or environments. Visibility is much too expensive to compute dynamically and so is not included in the default case. Visibility preprocessing is not feasible, or even useful, for arbitrary models which are not completely available locally. For similar reasons, motion prediction is not useful for general-purpose systems. In the default case, there is no path constraint, since collision between avatar and environment is not supported. Additionally, the user motion model allows near-infinite rotational acceleration and velocity, which makes prediction highly inaccurate.

D. IMPORTANCE ANNOTATION STRATEGIES

Generating Importance information should be a trivial addition to the authoring process. In most scenes, the majority of nodes have average importance. Some objects would be annotated as varying from average if they were especially important (or unimportant) to the intended usage of the scene. A model author cannot possibly foresee all possible applications of a scene, which is why the author annotation information is used in only the most general-purpose systems.

Automatic Importance generation methods usually hinge upon visibility and sightlines; for instance, landmarks might be identified as those objects which can be seen from many places in the virtual environment. Certainly the visibility preprocessing discussed above is a form of automated Importance generation. The Ville project, mentioned in section II.D, uses morphological analysis to determine areas of interest in city models. It is important to note that any of these mechanisms can be incorporated into the *QUICK* framework by building a Task which knows how to apply that information appropriately in generating an up-to-date Importance for a scene object. While *QUICK* includes several common mechanisms for generating Importance, it has been designed as a framework for the exploration of existing and new algorithms rather than a definitive library of techniques.

E. THE COST FACTOR

The remainder of this chapter describes the components of the Cost annotation. The *QUICK* Cost factor is a multi-dimensional value that reflects a representation's consumption of the various limited system resources. The available amounts of each of these resources for a given display platform are described by its client specification. The optimization process selects the highest-fidelity representations whose summed resource costs are below the specified limitations.

1. Cost Components

The Cost tuple consists of two primary sections: storage requirements and processing requirements. Storage requirements relate to memory footprint and file storage, while processing costs are those related to rendering a representation. It should be noted that while the components of these costs are discussed individually below, many new and different system limitations will likely become important as new types of representations and platforms are incorporated into *QUICK* .

The storage cost of a representation includes the following factors:

- **Disk footprint.** Text-based graphics file formats are generally designed for readability rather than compression. Accordingly, the file size is included as a separate resource Cost. Available disk file-cache space is rarely a constraint, but can be important for very large environments or long-lived sessions. This can be determined by simple inspection of the completed file.

- **Memory footprint.** Each representation has a memory space requirement after it has been parsed into a scene graph and geometric description. An exact value requires knowledge of the display platform and graphics library. Sinking memory costs have reduced the likelihood of main memory constraints, but knowledge about storage size is required for cache management for large environments. This information is usually determined by the author in an experimental application, or the disk footprint is used as the default.

- **Network footprint.** The transmission size of a graphics file is generally the same as the disk footprint. This component can be different if a chosen file format includes any sort of network compression. Network bandwidth is frequently a tightly-constrained resource, and the network footprint is used to prioritize network requests.

- **Texture size.** Most modern graphics hardware systems include special-purpose cache memory for storing textures. Exceeding the limitations of that cache will often significantly degrade performance by requiring additional bus transfers between main memory and the graphics subsystem. This information can usually be determined with modeling tools.

The processing cost of a representation includes the following factors:

- **Primitive Count.** For traditional graphics hardware, the primary limitation on scalable virtual environments is polygon throughput. Polygon flow reduction has been a primary research focus since the onset of computer graphics. While lit triangles are certainly no longer the only way to describe three-dimensional geometry, they are still the primary standard for benchmarking hardware performance. While this value is a simplification which does not include optimization information (such as the organization of the primitives, which can greatly enhance throughput), primitive count is still the most effective gauge of the processing requirements for a model. This information can be determined with a variety of public-domain modeling tools.

- **Pixel area.** Graphics systems can also be limited by their capability to rasterize triangles into filled pixels on the screen. The pixel area gives the number of pixels that must be filled to display a representation. Pixel area can be estimated by transforming the representation's bounding volume to the appropriate distance and projecting to screen space. This information can only be ascertained during execution, when the object's position is available, so this Cost component is often omitted from the optimization process.

Non-standard representations, such as fractally-defined geometry, require computations that cannot be performed with graphics hardware. The Cost annotation originally included a FLOPS (float-point operations) component which specified the amount of processing needed to generate displayable geometry from the memory description. The great variety of possible representations, and the equally great variety of algorithms for their computation, made that component's use infeasible. There is currently no way to specify what graphics library will be used to process a representation, and without that information format-specific processing estimates are not useful. This topic requires additional investigation, and is discussed further in Chapter XI.

2. Computing Cost

Because the Cost factor is a vector instead of a single value, there is usually no need for a computation step. When formulating the optimization problem, each representation requires a certain amount of each system resource. The client specification gives the limitation for each resource, and therefore, the limitation to the cost constraints in the optimization.

The default computation of Cost does not perform any computation. Tasks can override this behavior if desired. For instance, Cost components can be dependent upon dynamic application state; pixel area is a prime example, which requires updated viewpoint information. In general, Tasks should avoid excessive computation in the Cost determination stage, as it affects the system processing load but cannot be included or omitted from the optimization process.

VII. SOFTWARE DESIGN

A. INTRODUCTION

This chapter explains the software implementation of the *QUICK* framework. It begins with an discussion of available graphics software libraries, and a rationale for the selection of Java and Java3D. Following is a description of the scene graph file format, which combines geometric descriptions of representations with the *QUICK* annotations. The chapter concludes with a review of the software architecture for managing the model cache, that is, the process by which models are loaded, parsed, and displayed.

B. SOFTWARE LIBRARIES

The choice of graphics library software is complicated by the availability of a number of effective but disparate solutions. Choosing a particular graphics library brings concomitant choices of scene graph format, available high-order geometric representations, hardware and operating system choices, and more.

This section describes the *QUICK* system's requirements of a graphics library, as well as the reasons for the selection of graphics library for the primary *QUICK* implementation.

1. Requirements

Because the selection of graphics software library has such pervasive effects on the system architecture, a list of requirements were established at an early stage:

- **Cross-Platform:** The *QUICK* system is intended to be a general form solution which reduces client display platforms to a set of important characteristics. Therefore, the implementation itself should support heterogeneous platforms. Cross platform windowing support is not a requirement, but is preferred.

- **Free, or Ubiquitous:** *QUICK* itself is intended to be distributed freely, so it is appropriate that the chosen graphics subsystem be widely, or freely, installed.

- **Extensible:** No scene graph or library will contain all possible representation types. Most, but not all, graphics libraries are extensible.

- **Multi-threaded:** Support for concurrent access to the scene structure is required in order for *QUICK* to perform optimizations while drawing. Single-threaded execution would lead to a notable lack of interactivity.

- **High-level:** A library with its own high-level scene graph gives an excellent starting point for *QUICK* development. Additionally, the benefit of a low-level only graphics API (flexibility) is not necessarily helpful in this instance.

2. Selected Software

Initially, the creation of a new scene graph library was considered. That option was discarded because it would likely negatively affect the use of *QUICK* as either a system foundation or learning tool. Therefore, a number of graphics libraries were investigated for use in the *QUICK* framework. This section summarizes the findings of that investigation.

The Performer, Fahrenheit, and Direct3D Retained-mode libraries were all rejected due to lack of portability. Performer currently is available for only SGI Irix and Linux platforms; the Linux release has only limited functionality. Fahrenheit and Direct3D are available only for Microsoft Windows platforms.

OpenInventor is implemented upon a number of platforms, though for some platforms there is a fee for third-party implementations. However, OpenInventor is by nature a single-threaded application, which makes it infeasible for real-time applications with *QUICK* .

At the time of this decision, the Fahrenheit and X3D libraries were not fully specified, so they were not fully considered as options.

OpenGL meets many of the needs for *QUICK* , in that it is widely-available, freely distributed, high-performance, and cross-platform. OpenGL does not support both Immediate and Retained mode rendering. Therefore it has no high-level scene-graph interface. Many scene-graph libraries (such as Inventor, Performer, and Java3D) sit atop OpenGL and those choices seemed preferable.

PLIB [PLI, 2000], a cross-platform library similar to Performer, was seriously considered. It offers reasonably high-performance, and is in Open Source. The Java3D library [Sowizral *et al.*, 1997] is similarly cross-platform, and has a much more active development community. Java3D is written atop Sun's Java programming language, whereas PLIB is a C++ library. Java is generally preferred over C++ when rapid prototyping and development is more of a concern than run-time performance, so it is naturally preferred for implementing a thesis proof-of-concept system. Because of the language difference, and its more supportive development community, Java3D was selected for the prototypical implementation of the *QUICK* framework.

C. QUICK SCENE GRAPH AND FILE FORMAT

To contain the *QUICK* annotations, and store the relationships between objects and their representations, it was necessary to create a number of special scene graph nodes. This section describes those nodes, the syntax for their specification, and their semantic interactions. Sun's Java3D graphics library was used for the *QUICK* software implementation (see Chapter X for an explanation of that decision). Although nodes in the Java3D scene graph cannot be directly modified, subclassing is allowed to permit extension and variation.

1. Scene Graph Elements

Each individual object in the virtual environment is represented in the *QUICK* scene graph by a QSwitch node. In a Java3D scene graph, Group nodes are interior tree nodes that include an ordered set of children. The Java3D Switch node extends Group by adding the ability to designate which of the children are included in traversals. That designation can include zero, all, or any combination of the child subtrees. The *QUICK* QSwitch node extends the Java3D Switch with the Importance information for its related virtual object.

Each representation of an object in a virtual environment is included in the scene graph with a QNode. The QNode is an extension of the Java3D TransformGroup, which is simply a Group node that includes a geometric transformation which is applied to all children. The QNode contains Cost and Quality annotations in special data structures; these are included as nodes in the file format, but are not scene graph nodes included in the traversal. The geometric data for a representation is stored in the children of the QNode. This information is often not available at initialization, but is instead kept in a separate file to allow demand-based loading. Each QNode includes a location field, which is a string representation of a (possibly networked) file location, which is used to locate the geometry. Because that geometric data for a QNode is usually stored in a separate file, it is incumbent upon the author of the QNode to ensure that the each representation of a virtual object share physical characteristics (size, position, etc.). QNode extends TransformGroup, and therefore contains its own transformation, to facilitate that process.

The geometric data stored beneath a QNode is often similar or identical across multiple occurrences of objects. To prevent repeated storage cost for each use, the Java3D scene graph supports *instancing* for repeated lightweight reuse of nodes. A subgraph can be loaded once into memory, and then symbolically linked into multiple points in the scene

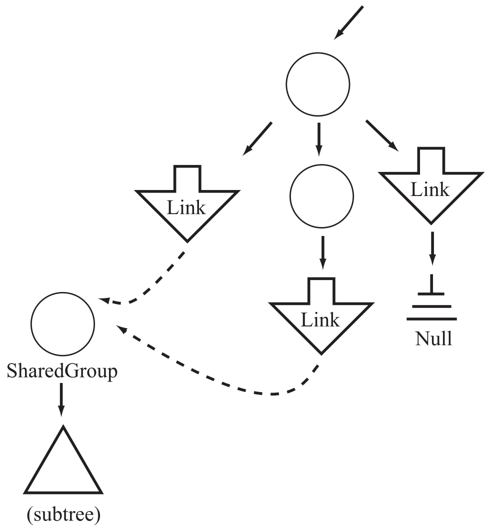


Figure 12. Java3D Link and SharedGroup nodes.

graph (see Figure 12). Java3D uses the SharedGroup node to mark the root of a sharable subgraph. The Link node is a special Group node that allows exactly one child, which must be a SharedGroup.

Each QNode contains a single Link node which points to the SharedGroup containing the representation geometry. The *QUICK* system defers loading that geometry until it is needed, so at initialization a QNode usually will have not have a subgraph. The proper procedure would be to add a Link to the SharedGroup when the geometry becomes available, but this is not permitted by the graphics library. In order to accelerate rendering, Java3D puts strong restrictions on run-time modifications to scene graph structure. To reliably circumvent this restriction, the *QUICK* implementation uses a special 'null' SharedGroup node. Each Link is initialized to point to the null node, which has no effect on the draw traversal; the Links are adjusted when their geometry becomes available.

Both the QNode and the QSwitch nodes include an array of strings which serves as the functional description. This information is required for task-based adjustment of the *QUICK* factors, as discussed in Chapter IV section E. Most objects serve a variety of roles in a virtual world, and therefore any given task might gauge the Importance of an object differently. The utility of a content description to describe the roles of a scene object (and its related QSwitch) is obvious. Less clear is the need for a content description of an individual geometric representation (the QNode). Actually, the capability to annotate a representation with qualitative remarks gives great power of expression. For example, there is no straightforward method for comparing the Quality of a artist's non-photorealistic representation of a hotel with the Quality of a geometric CAD model. Depending on the user or task, either might be considered the superior. Labeling each a representation as "cartoonish" or "dreary" can adequately inform a task for proper discrimination. (Use of ontological descriptions in fidelity computation was discussed in Chapter IV.)

The structure of the *QUICK* scene graph is tightly constrained in order to minimize the complexity of the optimization process. These topographical constraints do not cause any loss in generality for scenes which can be depicted, because the topology of a scene graph does not need to be related to visual arrangement. These constraints are listed and explained below; additionally refer to Figure 13.

- *QSwitch allows only QNode children.* For simplicity, *QUICK* assumes that only QNodes will be attached to a QSwitch grouping node. Each child of a QSwitch is assumed to be a different representation of the same virtual object. Allowing any

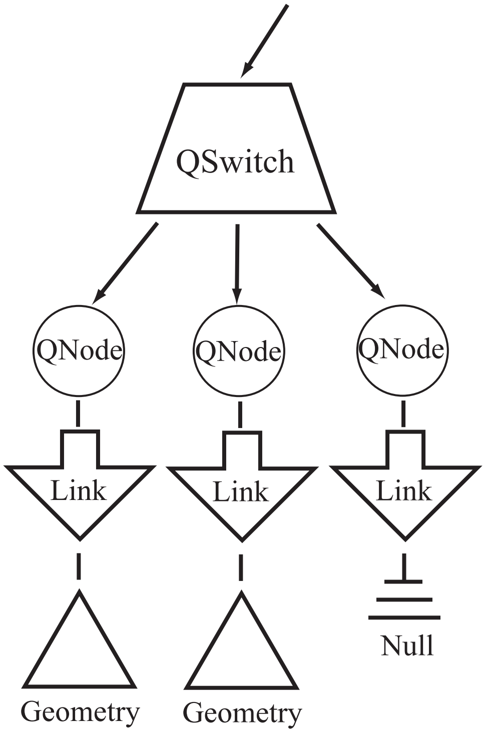


Figure 13. A legal QSwitch node has only QNode children, which each contain a single Link child.

other type of node as a child implies that the *QUICK* system would not have the annotation information needed for the linear optimization model. A single child without those annotation is enough to make optimal child selection impossible.

- *Only one QSwitch allowed on any path.* Allowing nested QSwitch nodes greatly increases the complexity of the computation. Nested decision points would require solution of optimization sub-problems in the overall optimization, increasing the already-exponential complexity of an n-QSwitch optimization by a factor of $n!$. Therefore, only one QSwitch is allowed on a path from the scene root to any leaf.

- *Qnode has one Link child.* QNode supports only a single child, which is a Link node as discussed above. When the geometry for a representation is not in memory, the Link points to the null node. Any other children of the QNode are ignored by the *QUICK* engine, and their presence could cause unwanted behavior. Accordingly, the file parsing system rejects files with more than one child in a QNode; these are syntactically correct, but semantically flawed. Chapter VIII contains a more detailed discussion of this issue.

- *No extraneous Link nodes.* To identify the top-down inherited state at any given node, it is necessary to trace upwards to the scene root. Most scene graphs are simple hierarchical trees, meaning that exactly one path exists from the root to any node. Link nodes and instanced SharedGroups add variability to the structure of a scene graph. To define a root-to-node path uniquely, it is then necessary to include each Link node on that path. The QSwitch node, and therefore the QNode, is constrained to not be nested. This limits the number of Links on any path to one, making the problem of tracking node paths much less complex. Since most *QUICK* path queries (such as world-coordinate position of an object) point to the QSwitch or QNode, no Link is included in the path at all. To simplify the path generation process, *QUICK* requires that the scene graph not include Link nodes from other sources. The VRML97 loader for Java3D does not use instancing, so this constraint does not restrict the authoring process.

2. File Format

This section describes the *QUICK* file format, and includes examples of the special *QUICK* control nodes. The *QUICK* file format is a derivative of the Virtual Reality Modeling Language (VRML) 1997 ISO standard [VRM, 1997]. The selection of VRML is a straightforward decision, for a number of reasons:

- *ASCII file format.* VRML models are traditionally expressed in plain-text, facilitating *QUICK* modifications to pre-existing VRML files. This also simplified file processing, as Java includes excellent functions for reading and parsing text.

- *Ubiquitous acceptance.* VRML is the *lingua franca* of three-dimensional models; almost every major authoring package includes a VRML export facility. Most web browser applications include a VRML browsing module, or offer one as an option. *QUICK* optimization techniques might have a tremendous impact on 3D on the Internet through VRML. By initially proving *QUICK* 's effectiveness with practical testing on VRML models, it is more likely that the recommendations of this thesis might be applied to that domain.

- *Free model libraries.* VRML's popularity led to the construction of many thousands of models. Many of these models are publicly available on the World Wide Web; in the absence of copyright restrictions, any can be annotated and included in a *QUICK* virtual environment.

- *Inherently networked.* VRML was designed from the beginning for client/server networking on the Internet. VRML's Inline node, which contains a web location for another VRML file, gives world authors the flexibility to incorporate models which are distributed across the Internet. *QUICK* is most effectively used with models segmented in exactly this fashion.

- *Java3D loader.* The Java3D & VRML Working Group of the Web3D Consortium established interoperability between the VRML format and the Java3D API. The program source for the loader is publicly available. Further development continues via that Consortium's X3D and Source Task Groups.

The VRML standard allows for extension with new node types, using the PROTO (prototype) and EXTERNPROTO (externally-defined prototype) nodes. The *QUICK* annotations and additional nodes are defined within the VRML97 standard using these constructions. PROTO-handling in the Java3D VRML97 loader does not lend itself to the *QUICK* optimization process. Therefore, for convenience, the initial *QUICK* implementation uses a special extension of VRML97 with non-standard node definitions. *QUICK* node definitions using the PROTO construction are included below for completeness.

The format for each of the new *QUICK* nodes is discussed in turn below. Each line of these specifications includes the field type, the field tag, and the default value for the field. Field types are given in the same format as the VRML97 specification [VRM, 1997], and the reader is strongly recommended to consult that document. (Briefly, the prefixes "SF" and "MF" indicate a single field and a multiple-member field, respectively. "Vec3f"

```
QNode {
  # fields common to the VRML Group and Transform nodes:
  SFVec3f     bboxCenter       0.0 0.0 0.0
  SFVec3f     bboxSize         -1.0 -1.0 -1.0
  MFNode      children         []

  # fields used in the VRML Transform node:
  SFVec3f     center           0.0 0.0 0.0
  SFRotation  rotation         0.0 0.0 1.0 0.0
  SFVec3f     scale            1.0 1.0 1.0
  SFRotation  scaleOrientation 0.0 0.0 1.0 0.0
  SFVec3f     translation      0.0 0.0 0.0

  # new fields:
  MFString    contents         []
  SFString    url              ""
  SFNode      cost             NULL  # a QCost node
  SFNode      quality          NULL  # a QQuality node
}
```

Figure 14. QNode file format.

indicates a vector containing three floating-point numbers, and “Rotation” is an axis-angle representation analogous to a quarternion vector.)

The QNode representation format using VRML is given in Figure 15 (the modified VRML version used in the *QUICK* implementation is given in Figure 14). Most fields are inherited from its base Transform node. The VRML Transform node is in turn a subclass of the Group node, so those fields are listed as well. The *children* node list is used when the geometry for a representation is included in the same file. Generally, it is preferred to use the *url* string to specify where to find that geometry, because this gives the *QUICK* framework the option to defer loading and parsing. In the case of small geometric descriptions,

```
PROTO QNode [
  # fields for the VRML Transform node:
  field        SFVec3f     qbboxCenter       0.0 0.0 0.0
  field        SFVec3f     qbboxSize         -1.0 -1.0 -1.0
  exposedField SFVec3f     qtranslation      0.0 0.0 0.0
  exposedField SFRotation  qrotation         0.0 0.0 1.0 0.0
  exposedField SFVec3f     qscale            1.0 1.0 1.0
  exposedField SFRotation  qscaleOrientation 0.0 0.0 1.0 0.0
  exposedField SFVec3f     qcenter           0.0 0.0 0.0
  exposedField MFNode      qchildren         []

  # new fields:
  MFString    contents          []
  SFString    url               ""
  SFNode      cost              NULL  # a QCost node
  SFNode      quality           NULL  # a QQuality node
]
{
  Transform {
    bboxCenter IS qbboxCenter
    bboxSize IS qbboxSize
    translation IS qtranslation
    rotation IS qrotation
    scale IS qscale
    scaleOrientation IS qscaleOrientation
    center IS qcenter
    children IS qchildren
  }
}
```

Figure 15. QNode file format, using standard VRML PROTO.

```
QSwitch {
  # fields from the VRML Switch node:
  SFInt32     whichChoice      -1
  MFNode      choice           []

  # new fields:
  SFFloat     importance       .5
  MFString    contents         []
}
```

Figure 16. QSwitch file format.

it is often preferable to include the information directly as a child of the QNode, to avoid the overhead of restarting the parsing engine.

The *url* field is a character-string containing an Internet URL or a local file system reference. This field is ignored if the *children* field is not null. The *contents* field is a list of strings, as specified in the previous section and in Chapter IV, which describe this QNode’s representation. The *cost* field contains a single node, which must be a QCost node; similarly, the *quality* field contains a single QQuality node. If either field is left blank, the correct node will be created and initialized to its default values.

The QSwitch description is given in Figure 16. The QSwitch is a simple extension of the VRML Switch node, with two added fields. The VRML Switch includes an array of children, similar to a Group, with the added *whichChoice* field to designate which of the children should be initially drawn. The default value is to display none of the children, which is the preferred setting when authoring a *QUICK* model. The *whichChoice* setting is only used as the initial setting for a QSwitch; any subsequent optimizations may change the

```
PROTO QSwitch [
  # fields from the VRML Switch node:
  exposedField SFInt32     whichChild      -1
  exposedField MFNode      children           []

  # new fields:
  exposedField SFFloat     importance       .5
  exposedField MFString    contents         []
]
{
  Switch {
    whichChoice IS whichChild
    choice IS children
  }
}
```

Figure 17. QSwitch file format, using standard VRML PROTO.

rendered child without regard to that value. The *importance* value is a single floating-point number, whose purpose is described in Chapter VI. Lastly, the QSwitch contains a *contents* field for task-based optimization. Figure 17 gives the same description in a more standard VRML PROTO format.

The QQuality node indicates the Quality for a QNode representation. The format given in Figure 18 includes only a workable subset of the possible values that could be included in a Quality computation. *QUICK* is intended to serve as a framework for exploration in that area; this research does not purport to offer a general-purpose formulation for Quality, which can vary by application task. The QCost node is designed similarly (see Figure 19); it does not necessarily include all possible costs of a QNode representation, but it does allow sufficient flexibility for most models. All fields in the QQuality and QCost

```
QQuality {
  SFFloat     geomError           -1.0
  SFFloat     geomStdev           -1.0
  SFInt32     colorDepth          -1
  SFInt32     textureResolution   -1
  SFInt32     alphaDepth          -1
  SFFloat     subjective          -1.0
}

PROTO QQuality [
  exposedField   SFFloat     geomError           -1.0
  exposedField   SFFloat     geomStdev           -1.0
  exposedField   SFInt32     colorDepth          -1
  exposedField   SFInt32     textureResolution   -1
  exposedField   SFInt32     alphaDepth          -1
  exposedField   SFFloat     subjective          -1.0
]
{
  WorldInfo {
    # There is no standard VRML scene node
    # analog for QQuality, so a comment
    # node is added.
  }
}
```

Figure 18. QQuality file format, and its associated PROTO format.

nodes default to -1, which is recognized by the *QUICK* framework to mean that the value should not be included in the optimization formulation. Note that the PROTO forms of the QQuality and QCost nodes add only a comment node to the VRML scene graph. All field access is performed directly through the PROTO.

The example file in Figure 20 shows all of these nodes used in combination. It is important to note that, in VRML files, the field ordering within a node is insignificant. The

```
QCost {
  SFInt32     triangles        -1
  SFInt32     flops            -1
  SFInt32     filesize         -1
}

PROTO QCost [
  exposedField   SFInt32     triangles        -1
  exposedField   SFInt32     flops            -1
  exposedField   SFInt32     filesize         -1
]
{
  WorldInfo {
    # There is no standard VRML scene node
    # analog for QCost, so a comment
    # node is added.
  }
}
```

Figure 19. QCost file format, and its associated PROTO format.

file contains a P-38 airplane with two representations, one with full geometry and the other just a simple box. The airplane object is modeled with a QSwitch to allow the *QUICK* system to decide between these two representations; each representation is placed in a QNode child of the QSwitch. The ordering of the QNodes in the QSwitch is not important, and is ignored in the optimization process. (The second QNode is the higher resolution model in this case.) The two representations do not have the same orientation or scale, so the second QNode uses its Transform capability to make those adjustments before loading.

```
QSwitch {
  importance 1.0
  contents [
    "Vehicle:Air:Plane:P38"
  ]
  choice [
    QNode {
      quality QQuality {
        textureResolution 0
        alphaDepth 0
        geomError  5.5
      }
      cost QCost {
        triangles 12
        filesize 154
      }
      url "http://vr.edu/quick/models/box.wrl"
      contents [
        "Geometry:Box"
      ]
    } # end QNode
    QNode{
        quality QQuality {
          alphaDepth 8
          textureResolution 0
          geomError .1
        }
        cost QCost {
          triangles 2404
          filesize 34552
        }
        rotation 1 0 0 -1.57
        scale 15 15 15
        url "p38.wrl"
    } # end QNode
  ] # end choice
} # end QSwitch
```

Figure 20. Example *QUICK* file using all special extension nodes.

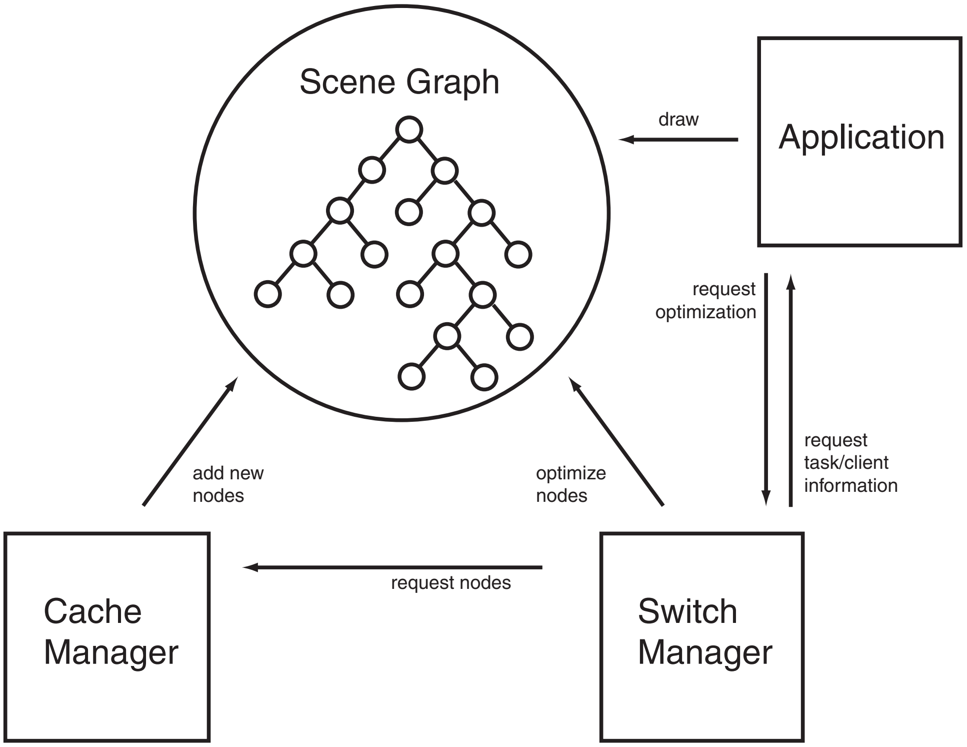


Figure 21. Primary functional components in the *QUICK* framework.

D. SOFTWARE ARCHITECTURE

This section explains the architecture of the components of the *QUICK* framework software system. This architecture is presented in a language- and implementation-independent manner to facilitate additional implementations. This architecture for *QUICK* optimization was designed to be general enough for application to any graphical browser paired with a scene graph offering thread-safe access. Details of the Java/Java3D proof-of-concept implementation built for this dissertation can be found in the following section.

The architecture consists of four major modules, as shown in Figure 21. The Application maintains, and possibly updates, the client specification and task definition. It also contains the user's graphical interface to the virtual environment. The visual data for the virtual environment is contained in a hierarchical scene graph, which all other modules can access or modify concurrently. The *SwitchManager* is attached to the scene graph to control display. The SwitchManager chooses which QNode child of each QSwitch is to be displayed. One method for making that choice is linear optimization, but SwitchManagers can be based on standard heuristics as well. The SwitchManager is also responsible for requesting new representations via *CacheManager*, the final module. The CacheManager controls the local store of objects; it handless all access to objects in secondary disk storage and the network. When a node is requested, the CacheManager locates, loads, and parses the node and inserts it into the scene graph.

1. Application Design

The Application module contains the graphical display engine which handles navigation of the virtual environment. A typical *QUICK* application can be built atop a pre-existing walkthrough program, adding two Manager modules and giving them partial access to the scene graph.

The Application holds task and client specification information, and must offer access to the SwitchManager module. *QUICK* applications designed for a specific purpose may keep the task static, whereas others may allow the user to switch between multiple tasks as the situation warrants.

Each type of Task is represented by a separate program class responsible for computation of the *QUICK* factors. When the SwitchManager performs an optimization, it requires up-to-date Quality and Cost for each QNode and Importance for each QSwitch.

The algorithm for determining these values is dependent upon the application goal, so there is no useful method that can suffice in all cases. Therefore, each Task class embeds its own program code for computing the *QUICK* factor values. The SwitchManager delegates all computations to the current Task, so that it can return values that are properly modified.

2. CacheManager Design

The CacheManager module must manage all of the multiple sources and stores for representations—including the network, local disk, and main memory. The CacheManager does not necessarily make any decisions about which files to request; it only needs to carry out the commands of the SwitchManager. The CacheManager can be charged with selecting nodes for deletion when necessary. The deletion process can be optimized in nearly the exact same fashion as the request process; a combination of the Least-Recently-Used strategy, with lowest-Importance / highest-storage-Cost, seems appropriate.

The CacheManager consists of a number of subcomponents which help with storage and network access. Those components, shown in Figure 22, operate as follows:

- *CacheManager*: The CacheManager component provides the disk and network interface to the SwitchManager. It contains a LoadManager and a buffer of nodes to be returned to the SwitchManager.

- *LoadManager*: This component offers a sparse interface to the CacheManager for nodes: Load(), Unload(), and Delete(). It handles the platform-specific details of

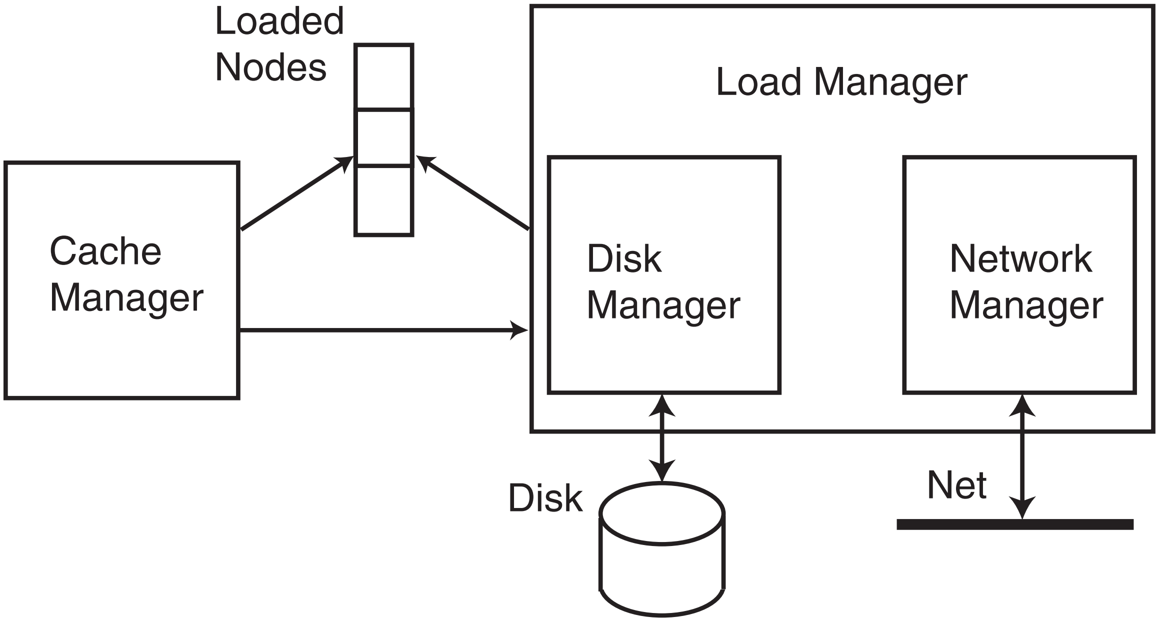


Figure 22. Cache management components.

loading files with the DiskManager and NetworkManager. It additionally contains the parsing elements for building scene graphs from files.

- *DiskManager*: The DiskManager controls transfer of nodes to and from the local disk; these can be either files on the local drive or files cached locally from previous network activity. The simple API includes the following: Load(), Save(), Delete(), and a test to see if a node is already in the disk cache.

- *NetworkManager*: The NetworkManager implements a single Fetch() method used to download a node from a network location. NetworkManager, DiskManager, and LoadManager need to observe the Singleton pattern; that is, only one instance of each can exist in any process space.

3. SwitchManager Design

The SwitchManager module performs the optimizations that drive the modifications to the scene graph. It offers both single-pass and ongoing optimization, depending on the needs of the application. Internally, it traverses the scene graph (in-order) and runs special helper functions whenever QNode or QSwitch nodes are encountered. The SwitchManager usually needs up-to-date *QUICK* factor information for these helper functions. To compute those values, it queries the Application for the current Task and delegates the computation as desired. The resulting *QUICK* values are cached whenever possible; for example, if the Task and client specification have not changed, and the Quality algorithm is not sensitive to application-state (such as user's head position), those values need not be recomputed.

Different classes of SwitchManagers might exhibit radically different behavior on the same scene graph. One might request all unloaded QNodes when it encounters them, while another might compute an optimal pre-caching request order based upon a predicted navigation path. The key to these differences lies in the implementation of the QNode and QSwitch processing functions that are invoked during traversal. In the example in which all nodes are automatically requested, the QSwitch processing function would be empty, and the QNode processing function would request the QNode's representation if not already available.

An optimal draw process is slightly more complex, as is illustrated in Figure 23. In this case, the optimization function creates a linear programming problem instance, then uses the traversal process to add the variables and constraints to the problem. At each

```
optimize:
  create a new optimization problem instance;
  traverse tree;
  solve problem;
  where result differs from current,
    change the displayed QNode;

to process QSwitch:
  compute Importance for this node;
  add new QSwitch and its Importance to problem;

to process QNode:
  compute Quality;
  compute Cost;
  inform problem to add this QNode to the current
    QSwitch, with its Quality and Cost;
```

Figure 23. Pseudocode for optimal drawing algorithm.

QNode and QSwitch, the *QUICK* factors are dynamically computed and submitted to the optimization problem. After the traversal is complete, the problem is solved, and its results are applied by changing the drawn QNode where directed.

VIII. OPTIMIZATION PROCESS

The optimization problem can be stated as multiple instances of the following questions:

- **Display.** Given a series of QSwitch nodes, and associated QNode children, which available QNodes should be displayed?
- **Child request.** Given a QSwitch node, which QNode children (if any) should be loaded into memory, and in what order?

The discussion below demonstrates that these problems can be reduced to the same problem, given the special constraints on *QUICK* scene graph construction.

Display. Each QNode node in the scene graph has associated with it *QUICK* annotation information. Given a constraint on total allowable cost (which is based on the capability of the display platform), the Display problem is a straightforward linear optimization to maximize fidelity. The programming model for that optimization is discussed further in section VIII.A below. The result yields a selection set which chooses zero or one QNodes for display at each QSwitch.

Child request. To perform asset prioritization for virtual world transfer, the system must create a preference ordering for the unloaded subtrees of each QSwitch node. This process cannot be performed in an optimal manner without *QUICK* annotations for each node in each (as yet unloaded) subtree. The decision to download a subtree must certainly be made in advance of making the download; an optimal decision may not include loading the subtree at all. Even downloading a skeleton of the subtree's scene graph, including *QUICK* annotations but omitting geometry, is not possible for some instances of the problem; for a large database, the skeletal subtree can itself be too great for local replication.

One logical approach is to record summary annotation information at each level of the scene graph hierarchy, and to fetch only the summary information at each level. Unfortunately, this is difficult to support because there is no straightforward method for summarizing the annotations. For instance, given three nodes with very different Quality annotations, there is no way to give a summary that is both accurate enough for optimization and smaller than a complete listing.

To make the optimization problem tractable, *QUICK* scene graphs are constrained to have no more than one QSwitch and one QNode, on any path from scene root to any scene leaf. In practice, this constraint is not overly restrictive. Multiresolution models traditionally do not contain multiresolution submodels; resolution selections are usually internally complete. This indicates that a QSwitch subtree will generally be homogeneous in Quality; that it represents one version of the object denoted by its parent QSwitch object, so it can be represented by the QSwitch's Importance; and its homogeneity allows its Cost to be aggregated as well.

This restriction on scene graph construction thus reduces the Child Request problem to be similar to the Display problem. First the Display problem is solved over the set of available representations. Then the Display problem is recreated, but QNodes holding both available and unavailable representations are included in the formulation. If the result of this new optimization is the same as previous, no nodes need fetching into the cache. If the result differs, all unavailable nodes that were chosen in the optimization needs to be considered for request. Those requests can be prioritized by transfer cost, fidelity contribution, or whatever manner a given optimization scheme prefers given the current availability of resources.

A. PROBLEM FORMULATION

The formulation of the *QUICK* optimization model is performed in three steps:

1. Build maximizing objective function
2. Add total cost constraint
3. Add object constraints

The following simple example illustrates the process of building an optimization problem from a small scene graph. Figure 24 shows a scene graph with two objects. Each object is represented by a QSwitch (trapezoid); at the time of the optimization, Obj1 has a dynamically computed Importance value of 0.5, and Obj2 has an Importance of 0.7. Each object has four possible representations, or QNodes, shown as the circular "Reps" in the graph. The Quality (Q) and polygonal Cost (C) of each representation has already been computed, and are also included in the graph.

The given optimization task is an instance of the display problem, within a polygonal cost of 30. That is, all four representations for each object are already in memory, and

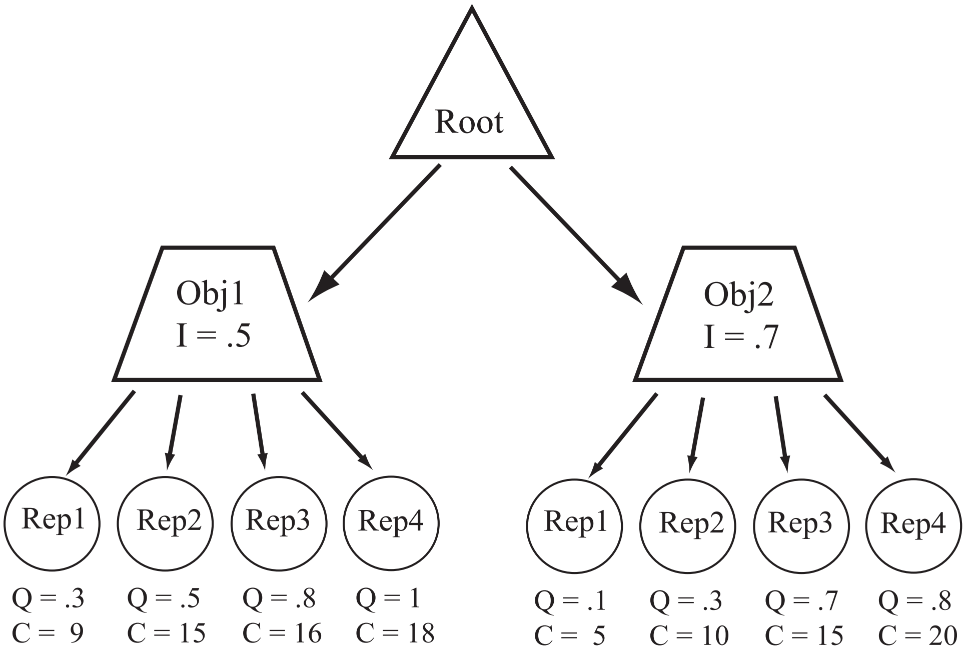


Figure 24. A simple scene graph with two objects with different importance values and representations.

the optimization will be used only to decide which representations to display. The steps enumerated above are used to build the linear optimization model. There is one variable for each representation, where each variable is boolean and can be set to 0 (do not draw) or 1 (draw). The representation choice vector is labeled X, consisting of variables $x_{i,j}$ where i is the QSwitch and j is the QNode child of QSwitch i.

Step 1: Build maximizing objective function.

To maximize total Fidelity, it is first necessary to determine the Fidelity contribution of any particular representation choice. Most of the computation of the *QUICK* factors has already been completed; the only remaining step is multiplicative combination of Quality and Importance. This step includes the “empty” representation for each object, to allow the possibility that an optimal situation could include no representation for a given object. The

Fidelity for the possible representations is given below.

That yields the following objective function for this instance:

$$.15x_{1,1} + .25x_{1,2} + .4x_{1,3} + .5.x_{1,4} + .07x_{2,1} + .21x_{2,2} + .49x_{2,3} + .56x_{2,4} \tag{VIII.1}$$

Note that variables with 0.0 coefficients, namely the empty representations, have been omitted from equation VIII.1. The general-form equation is shown below in VIII.2. Functions I and Q are the Importance and Quality functions, respectively; n is the number of QSwitches and k is the variable number of QNodes for each QSwitch.

$$I_1Q_{1,1}x_{1,1} + I_1Q_{1,2}x_{1,2} + \ldots + I_1Q_{1,k}x_{1,k} + \ldots + I_nQ_{n,1}x_{n,1} + \ldots + I_nQ_{n,k}x_{n,k} \tag{VIII.2}$$

which equates to maximizing the summation

$$\sum_{i=1}^{n}\sum_{j=1}^{k} I_iQ_{i,j}x_{i,j} \tag{VIII.3}$$

Step 2: Add total cost constraint.

The cost constraint includes each variable with its cost as a coefficient. The empty representations have no cost, so again they are omitted.

$$9x_{1,1} + 15x_{1,2} + 16x_{1,3} + 18.x_{1,4} + 5x_{2,1} + 10x_{2,2} + 15x_{2,3} + 20x_{2,4} \leq 30 \tag{VIII.4}$$

The general-form is similar to the general-form objective function:

$$C_{1,1}x_{1,1} + C_{1,2}x_{1,2} + \ldots + C_{1,k}x_{1,k} + \ldots + C_{n,1}x_{n,1} + \ldots + C_{n,k}x_{n,k} \leq MaxCost \tag{VIII.5}$$

which equates to the summation

$$\sum_{i=1}^{n}\sum_{j=1}^{k} C_{i,j}x_{i,j} \leq MaxCost \tag{VIII.6}$$

In instances where more there is more than one type of limited resource, this step will generate multiple cost constraints.

Step 3: Add object constraints.

The last step is to constrain the values of the variables to ensure exactly one representation is selected for each object. Each object yields a separate constraint of the form:

$$x_{i,0} + x_{i,1} + \ldots + x_{i,k} = 1 \tag{VIII.7}$$

This constraint would still allow for fractional combinations of the variables, or combinations of positive and negative coefficients. It is assumed that all variables have already been constrained to $\{0, 1\}$; this is discussed further in the complexity analysis in the next section.

The optimal solution for the simple problem instance discussed above has a Fidelity of .74. That value is reached by selecting variables $x_{1,2}$ and $x_{2,3}$, which have fidelity of .25 and .49 respectively, and a total cost of 30.

B. COMPLEXITY ANALYSIS

This section gives a complexity analysis of the optimization problem encountered in the *QUICK* framework. For a full discussion of time and space complexity theory, the reader is urged to consult [Sipser, 1997, Garey and Johnson, 1979].

Most standard linear optimization problems are known to be solvable in polynomial time (P-time) [Bertsimas and Tsitsiklis, 1997]. Unfortunately, linear optimization problems that constrain variables to integer values, known as *integer programming* problems, often require significantly more computation to solve. The variables in the *QUICK* optimization problem (hereafter labeled Q_{opt}) are each associated with a certain representation, and dictate whether it is selected in an optimal subset. Since representations are either chosen or not chosen, those variables are all constrained to integer values in $\{0, 1\}$, where 1 indicates those representations to be included in the optimal set. Q_{opt} is therefore an instance of the *zero-one integer programming* problem (commonly called ZOIP).

Any optimization problem has two closely-related corresponding problems: *evaluation* and *recognition* [Bertsimas and Tsitsiklis, 1997]. The evaluation problem is to determine the value of the objective function, that is, the value of the optimal assignment of variables. A solution to the evaluation problem specifically does not yield the preferred assignment of the variables. The recognition problem is a slight simplification of the evaluation problem; it determines whether the value of the objective function meets or exceeds a given threshold, and does not even yield the actual value of the objective function.

A P-time solution to the optimization problem guarantees a P-time solution to the evaluation problem, since the value of the objective function can be computed in P-time from the variable assignments that result from the optimization solution. Similarly, a P-time solution to the evaluation problem leads to a P-time solution of the recognition problem, since the evaluation result must only be compared to the threshold in an $O(1)$ operation.

When applied to the *QUICK* optimization, these problems can be stated as follows:

- Q_{opt} = { $< G >$ $|$ determine assignment of variables X which yields the maximum fidelity, given the scene-graph optimization problem G }

- Q_{eval} = { $< G >$ | determine the fidelity f of the optimal solution to the scene-graph optimization problem G }

- Q_{recog} = { $< G, f >$ | determine whether there exists a solution to the scene-graph optimization problem G whose fidelity is $\geq f$ }

The *QUICK* optimization library uses an exponential-time algorithm to maximize the fidelity of a given scene graph, which indicates that the problem scalability is less than would be desirable. In fact, it is highly unlikely that a faster solution to Q_{opt} exists, since it can be shown to be an NP-complete problem. A proof follows; it begins by showing that Q_{recog} is NP-complete, and then extending that result to show Q_{opt} is also NP-complete.

To show Q_{recog} is NP-complete, it is necessary to show that Q_{recog} is in the class NP, and that it is NP-hard.

1: Show $Q_{recog} \in NP$.

A language is in NP if and only if it is decided by some nondeterministic polynomial time Turing machine, or equivalently, has a polynomial-time verifier. Consider Turing machine T_R which nondeterministically branches on each representation variable, such that for each possible assignment of variables, one computation branch computes the cost and fidelity for that assignment. The cost and fidelity computations for a single representation requires $O(1)$ time, and therefore each branch of computation would require $O(n)$ time where n is the number of representations. Thus, T_R runs in polynomial time, and Q_{recog} is in NP.

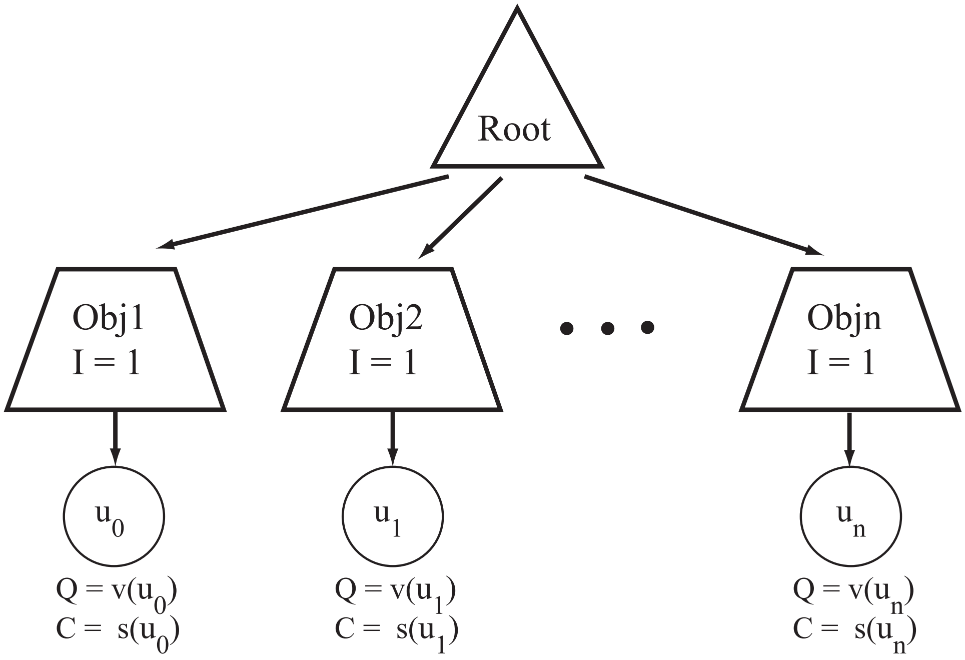


Figure 25. An instance of the 0-1 Knapsack problem converted to an instance of the *QUICK* recognition problem.

2: Show Q_{recog} is NP-hard.

This proof is accomplished by polynomial-time reduction from the 0-1 Knapsack problem. An instance of the 0-1 Knapsack problem is defined as positive integers B and K, a finite set U, and functions $s(u)$ and $v(u)$ over U such that $s(u) \in Z^+$ and $v(u) \in Z^+$. The problem is to determine whether there exists a subset $U' \subseteq U$ such that $(\sum_{u \in U'} s(u)) \leq B$ and $(\sum_{u \in U'} v(u)) \geq K$.

The related optimization problem is stated more colloquially as the thief's dilemma; given the desire to maximize his gain, and a knapsack of limited capacity, and varyingly-valued items to steal, which items should the thief place in his knapsack. The version of the problem generally called 0-1 Knapsack is the recognition problem, namely whether there is a way to fill the knapsack that gives the desired value within a limited capacity. The

“0-1” refers to the binary-constrained problem, where the solution allows only zero or one of each item.

0-1 Knapsack [Garey and Johnson, 1979] is a problem widely known to be NP-complete. Any instance of 0-1 Knapsack can be easily transformed to an instance of Q_{recog} in polynomial time (see Figure 25). For each $u \in U$, the transformation creates a QSwitch with Importance = 1, and a single QNode child with Quality = $v(u)$ and Cost = $s(u)$. The Cost limit is set to B, and the fidelity minimum f is set to K. A solution to this instance of Q_{recog} is exactly analogous to a solution of the original instance of the 0-1 Knapsack problem. Moreover, the transformation of the problem instance can be completed in polynomial time (specifically, $O(n)$ time). Therefore, since 0-1 Knapsack is NP-complete, and Q_{recog} can be used to solve 0-1 Knapsack, Q_{recog} must be NP-hard.

Similar tactics can be used with the 0-1 Knapsack recognition problem to show that Q_{opt} and Q_{eval} are NP-hard. An instance of the 0-1 Knapsack problem is polynomially transformed to a *QUICK* scene graph. Solving Q_{eval} yields the optimal fidelity, which is compared with K to give the solution to the Knapsack problem. Similarly, the variable assignments resulting from Q_{opt} can be evaluated in P-time to determine if the total fidelity is greater than K.

All three classes of *QUICK* problems have been shown to be NP-hard, and Q_{recog} has been proven NP-complete. The following steps use these conclusions to prove the NP-completeness of the evaluation and optimization problems.

As before, to show that Q_{eval} is in NP, it is necessary to show the existence of a nondeterministic Turing machine that solves the evaluation problem in P-time. Turing machine T_E accepts an instance G of the *QUICK* scene graph. In its first step, T_E computes the maximum possible fidelity F of any variable assignment. This can be determined easily, in $O(n)$ time, by summing the fidelities of the highest-fidelity representation from every QSwitch. In the second step, T_E makes a binary search of the possible solution space, from 0 to F, using the Turing machine T_R (which was earlier shown to solve Q_{recog}) as a subroutine. On each invocation of T_R, T_E eliminates half of the remaining possible results of the objective function, so it calls the T_R subroutine $\lceil \log F \rceil$ times. Since T_R has already been determined to run in polynomial time, T_E must also run in polynomial time. Therefore Q_{eval} must be in NP; since it has already been shown to be NP-hard, Q_{eval} is therefore NP-complete.

Armed with this knowledge, it is at last possible to prove that Q_{opt} is NP-complete. Turing machine T_O accepts an instance G of the *QUICK* scene graph problem. The machine's first step is to create a modified instance of G, labeled G', in which the first representation is constrained to 0. Turing machine T_E is run as a subroutine on both G and G', and the fidelity results are compared. If the results are the same, then clearly that representation can be removed from the optimization problem without loss of optimality. If the results are different, then the optimal variable assignment must include that representation set to 1. So in either case, the representation can be removed from G. This process is repeated for each variable (that is, for each representation) until no variables remain. This process requires

$O(n)$ invocations of the T_E machine, which has been shown to run in polynomial time. So, nondeterministic Turing machine T_O solves Q_{opt} in polynomial time, indicating that Q_{opt} is in NP. Since Q_{opt} has been previously shown to be NP-hard, Q_{opt} is therefore NP-complete. This coincides with general knowledge about the complexity of integer programming problems and zero-one integer programming problems [Garey and Johnson, 1979].

C. SIMPLIFICATION TECHNIQUES

By definition, NP-complete algorithms do not scale to large data sets. This section presents techniques for for simplifying the optimization process, either through approximation techniques or by constraining the problem. An introduction to dynamic programming, approximation algorithms and greedy algorithms can be found in [Cormen *et al.*, 1990].

1. Dynamic Programming

Dynamic programming solves optimization problems by solving its subproblems; it can be applied only to problems which exhibit optimal substructure traits. The 0-1 Knapsack problem, for instance, can be reformulated as a series of subproblems which determine the maximum value for a subset of the possible objects. The solution to those subproblems can be combined to determine the maximum value over the whole set of objects. Each subproblem can be computed in $O(|U|s(u)_{max})$ where $|U|$ is the size of the set of objects and $s(u)_{max}$ is the maximum cost value for any object. Since there are $|U|$ subproblems, the total running time for the dynamic programming algorithm is $O(|U|^2 s(u)_{max})$. For all but very large values of $s(u)_{max}$, this is a significant improvement.

2. Approximation Algorithms

Approximation algorithms provide a suboptimal solution to an optimization problem in polynomial time. They include a guarantee of their maximum error; some such algorithms even yield customizable speed/accuracy trade-offs. The 0-1 Knapsack algorithm, for example, can be approximated by reducing the cost values in the $s(u)$ function. Since the complexity of the dynamic programming solution hinges upon $s(u)_max$, reducing that value yields an equivalent reduction in running time. By scaling down the values in $s(u)$, some optimization accuracy is of course lost, but the solution complexity is reduced to $O(\frac{n^3}{\sigma}$ where σ is the bound on the error ratio. A full discussion of this algorithm is available in [Bertsimas and Tsitsiklis, 1997].

The *QUICK* implementation includes an approximation algorithm based on the "greedy" technique. For each optimization pass, the representations are sorted by a benefit-to-cost ratio; in this case the ratio is Fidelity to Cost. In the (standard) case of multi-dimensional Cost, multiple ratios are recorded. By using the merge sort algorithm, the worst-case and average-case running time of this greedy algorithm is $O(nlogn)$. Scene coherency can improve the expected running time further, since merge sort runs more quickly on nearly-sorted lists. This requires that the sorted list is stored between optimization passes, that few representations change Fidelity / Cost ratios, and that few representations are added or deleted.

Even this worst case of $O(nlogn)$ is a substantial improvement over the dynamic programming solution, even for low values of $s(u)_{max}$. The difference, of course, is that the greedy algorithm cannot guarantee an optimal solution. According to Garey and Johnson, the similar greedy approximation algorithm for the 0-1 Knapsack problem can guarantee a relative error no better than 2 [Garey and Johnson, 1979].

3. Continuous Representations

The complexity of the *QUICK* optimization stems from the integer constraint on representation selections. In the common case, the optimization task is to select from a set of discrete levels of detail. However, representations that offer continuous levels of detail do exist. Progressive meshes and fractal geometry, for instance, both can be dynamically computed to a exact level of accuracy. The available precision is usually limited by the geometric description technique (triangles, for instance).

This flexibility completely changes the optimization formulation. Instead of a list of static representations, each QSwitch would contain the maximum accuracy supported, and a function specifying the Fidelity/Cost ratio. The *QUICK* problem is then reduced to the fractional knapsack problem; each continuous representation must only be set to the complexity that maximizes overall Fidelity. This can be optimally solved by the greedy technique, by choosing the maximum allowable accuracy for those representations with the highest Fidelity/Cost ratio. Therefore, constraining objects to continuous representations allows optimization in $O(nlogn)$ time.

IX. SOFTWARE IMPLEMENTATION

The *QUICK* architecture discussed in Chapter VII has been implemented in Java in a proof-of-concept system which demonstrates the effectiveness of the optimization framework. Java was selected primarily because of the Java3D scene graph library. Java is also a natural choice for networked applications, as it was designed with web-based data, code transport, and portability in mind. Additionally, Java's simple memory and thread management facilities significantly reduced the programming burden.

This chapter describes the various packages that comprise the *QUICK* system, as well as their high-level interactions. It follows with a detailed examination of the classes in each package and their relationships. Diagrams of class relationships are given using the Unified Modeling Language (UML); a primer for UML can be found in the short reference book, *UML Distilled* [Fowler *et al.*, 1999]. Actual class names are given in `fixed-width` font. In color-printed versions of the diagrams, pure abstract interfaces are drawn in red, abstract classes are drawn in salmon-orange, and standard classes are drawn in yellow.

The *QUICK* system consists of a series of Java packages, each labeled (in the standard Java form) with an Internet domain and the system name. Because this is a Java3D implementation of *QUICK* , the names are of the form "edu.vr.quick.j3d.[package name]". The packages are:

- *edu.vr.quick.j3d* contains the core classes needed for any Java3D *QUICK* application, such as `QNode` and `QSwitch`.

- *edu.vr.quick.j3d.cache* contains the `CacheManager` and `LoadManager` classes that manage the *QUICK* cache of representations, including the memory cache, disk cache, and the scene graph.

- *edu.vr.quick.j3d.chooser* contains the `SwitchManager` classes that decide when and how to modify the application scene graph.

- *edu.vr.quick.j3d.opt* contains the classes which formulate and solve the zero-one integer programming problem from a *QUICK* scene graph.

- *edu.vr.quick.j3d.opt.lpsolve* contains classes for solving general-form linear optimization problems.

- *edu.vr.quick.j3d.opt.test* contains tests for both the lpsolve and opt packages.

- *com.sun.j3d.loaders.vrml97.impl* contains the classes needed to modify the standard Java3D loader to load and parse *QUICK* files.

- *edu.vr.quick.j3d.util* contains miscellaneous classes that do not fit in any other package.

- *edu.vr.quick.j3d.app* contains the main application components, including the GUI elements and the Java3D scene graph.

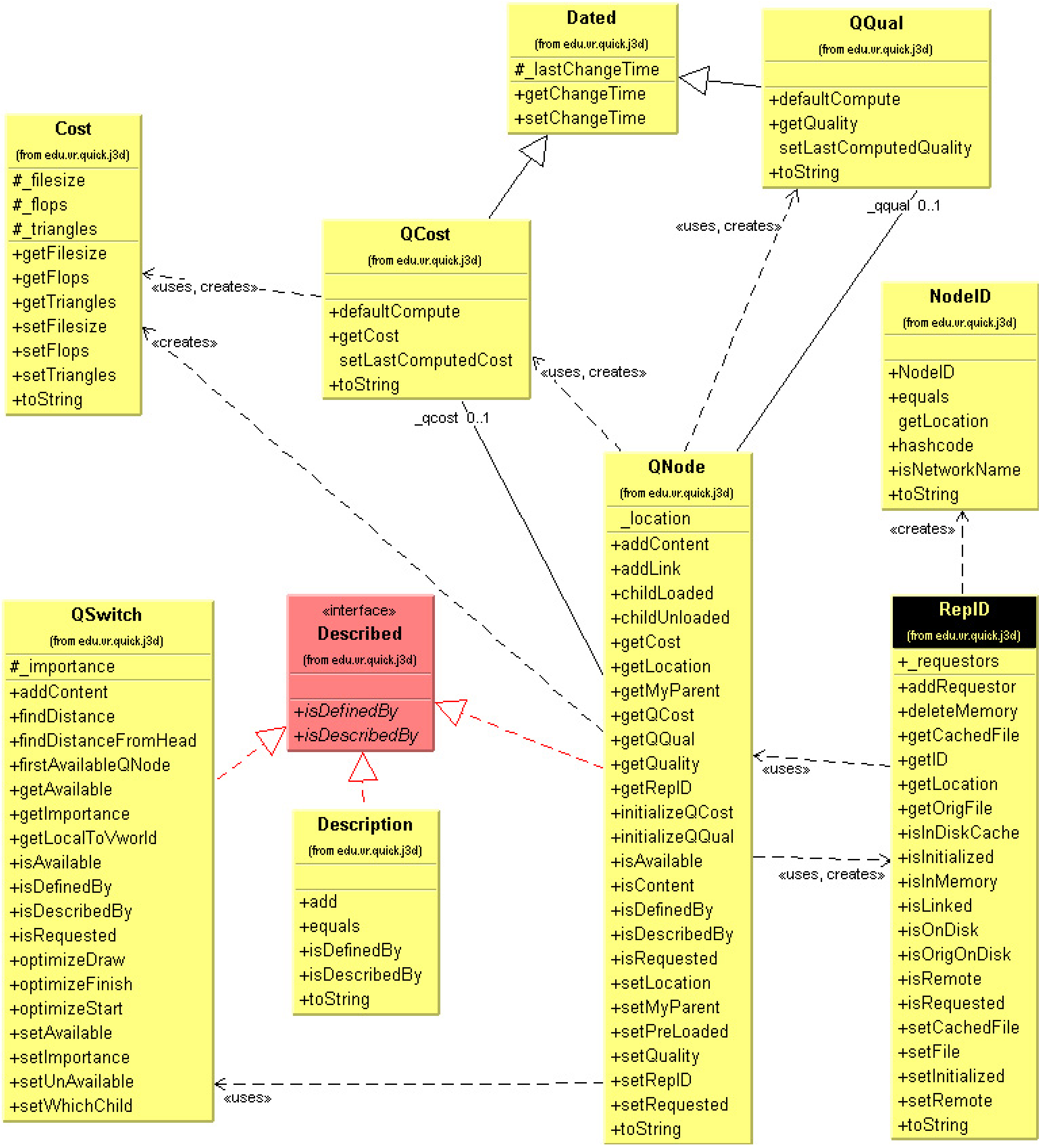


Figure 26. Class hierarchy diagram of scene graph-related classes in the edu.vr.quick.j3d package.

A. CORE PACKAGE

This package contains the core classes needed for any Java3D QUICK application. It holds the basic scene graph elements, *QUICK* annotations, and central elements for building Applications. Figure 26 shows the *QUICK* scene graph nodes, `QNode` and `QSwitch` and their included classes. The `RepID` class, contained by `QNode`, serves as the primary identifier and handle for representations in the virtual world. A `RepID` is constructed upon first discovering a representation's URL. After loading, it is used to find that representation in memory or the file cache; it is also used by the `CacheManager` and `SwitchManager` classes to identify a representation for loading or unloading. The `RepID` class in turn contains a `NodeID`; this is currently only the Internet URL, but other environments may use different globally-unique identifier schemes.

The `QQual` and `QCost` classes contain *QUICK* annotation information, as discussed in section C; their values are computed dynamically based upon the current client situation. (QQual is the name of the class which implements QQuality functionality.) The content description information for task-based modification of *QUICK* values is stored in a `Description` instance. The `QSwitch` and `QNode` classes each store a `Description` via the `Described` interface. Interface indirection is used because the `Description` class uses a Java Vector to store the definition terms; another implementation would be needed for better-than-linear search and insertion times.

The `Dated` class marks an object whose computed value ages with time. In the `QCost` class, this is used to track the time since the last dynamic cost computation. When the current cost is requested, the time of the last update is compared against the last change times to the current task and client specification. If the cost has been computed more recently, the cached value is used and no additional computation is required. This technique is used throughout the core and cache management packages.

The other classes in this package are provided for application development. The aptly-named `Application` class provides the basis for all *QUICK* applications. (The review of the edu.vr.quick.j3d.app package shows its relationships in the prototype system.) The `Application` instance contains the task and client specification, which describe the application's platform and current state. The `Task` abstract class includes methods for computing Quality, Importance, and Cost given a client specification. This diagram includes two concrete subclasses of `Task`: `StandardTask` and `FlyTask` (from the edu.vr.quick.j3d.app package). The `StandardTask` simply uses default calculation methods for the *QUICK* factors, while the `FlyTask` computes a special `Cost` instance for optimization purposes. The client specification is contained in the `ClientSpec` class, which contains fields for all of the various display platform characteristics important to the optimization process. The `ClientSpec` is a `Dated` class, like `QCost` and `QQual`, to encourage re-computation of *QUICK* factors when platform characteristics change during program execution.

B. CACHE PACKAGE

The cache package contains the implementation of the architectural components outlined in Chapter VII, section D.2 above; its primary internal interactions are shown in

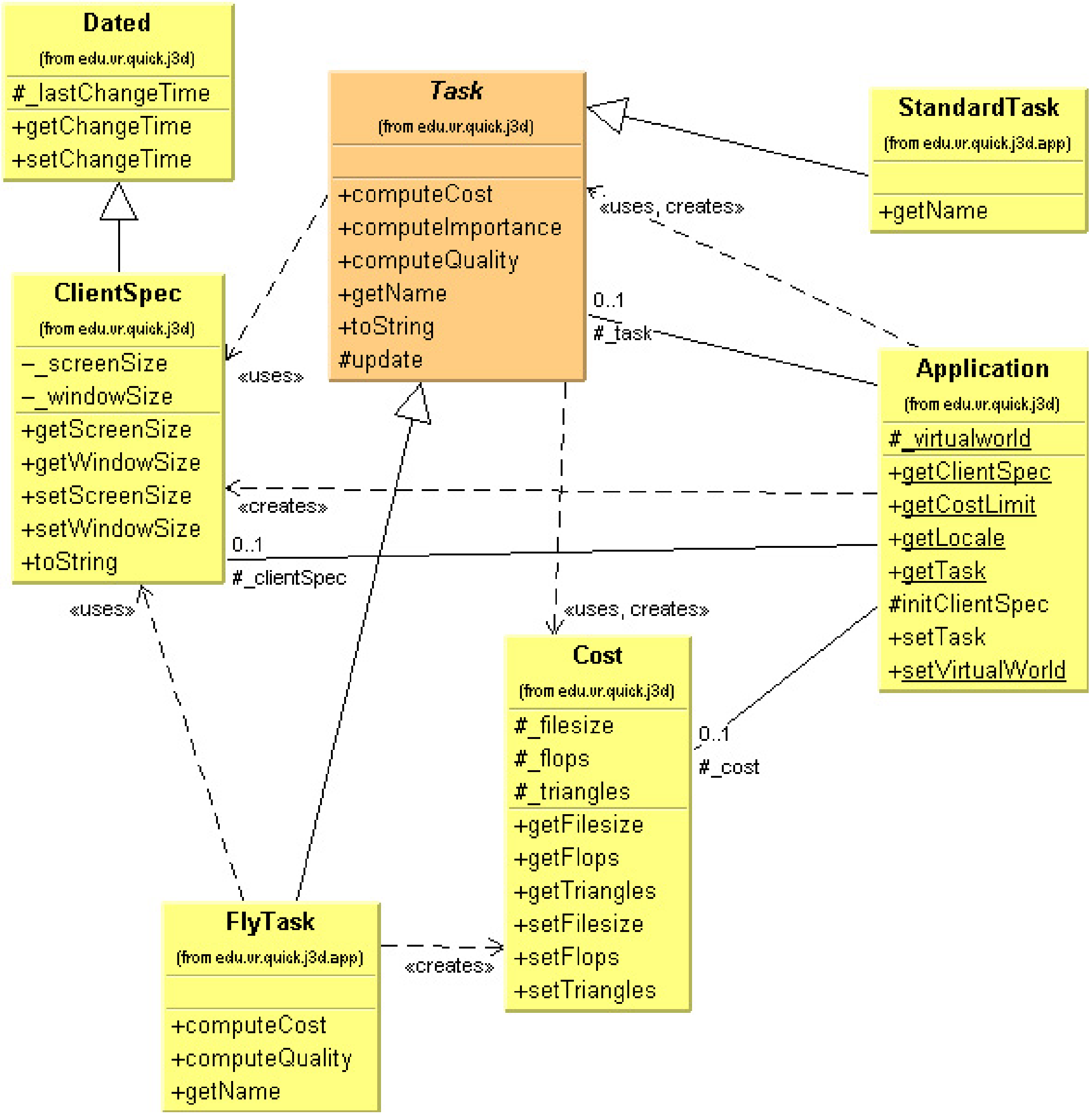


Figure 27. Class hierarchy diagram of application classes in the edu.vr.quick.j3d package.

Figure 28. The `CacheManager` is the only class which is externally visible, and it is used by the *QUICK* `Application` package. The `CacheManager` contains a reference to a `LoadManager`, which is an interface (a pure-virtual abstract class) for loading, unloading, and deleting representations. Interface indirection is used frequently in this *QUICK* implementation to encourage decoupling in design. The implementation of the `LoadManager`, `LoadMgrImpl`, contains an instance of a class implementing the `DiskManager` and `NetworkManager` interfaces, thereby giving access to disk and network resources for loading files. Those interfaces are implemented by the `LoadMgrImpl` and `DiskMgrImpl` classes respectively.

A typical load process is initiated by a `SwitchManager` invoking the `CacheManager.request` method. The `CacheManager` checks to make sure the representation hasn't already been loaded, or previously requested, and then calls the `LoadManager.loadRep` method. The `LoadMgrImpl` ensures that the designated file isn't already in the disk cache, and then determines whether the URL refers to a network or disk location. If it does refer to a local disk file, the `DiskMgrImpl` loads and parses the file and returns a Java3D scene graph to the `CacheManager` via the requesting `LoadMgrImpl`. If the file is located remotely, the `NetMgrImpl` class fetches the file and writes it into the filecache; the file is then handled as if it had been a local file originally. This "download, write, parse" scheme ensures that the secondary cache (those representations in memory but not in the scene graph) is always a subset of the tertiary cache (the local disk).

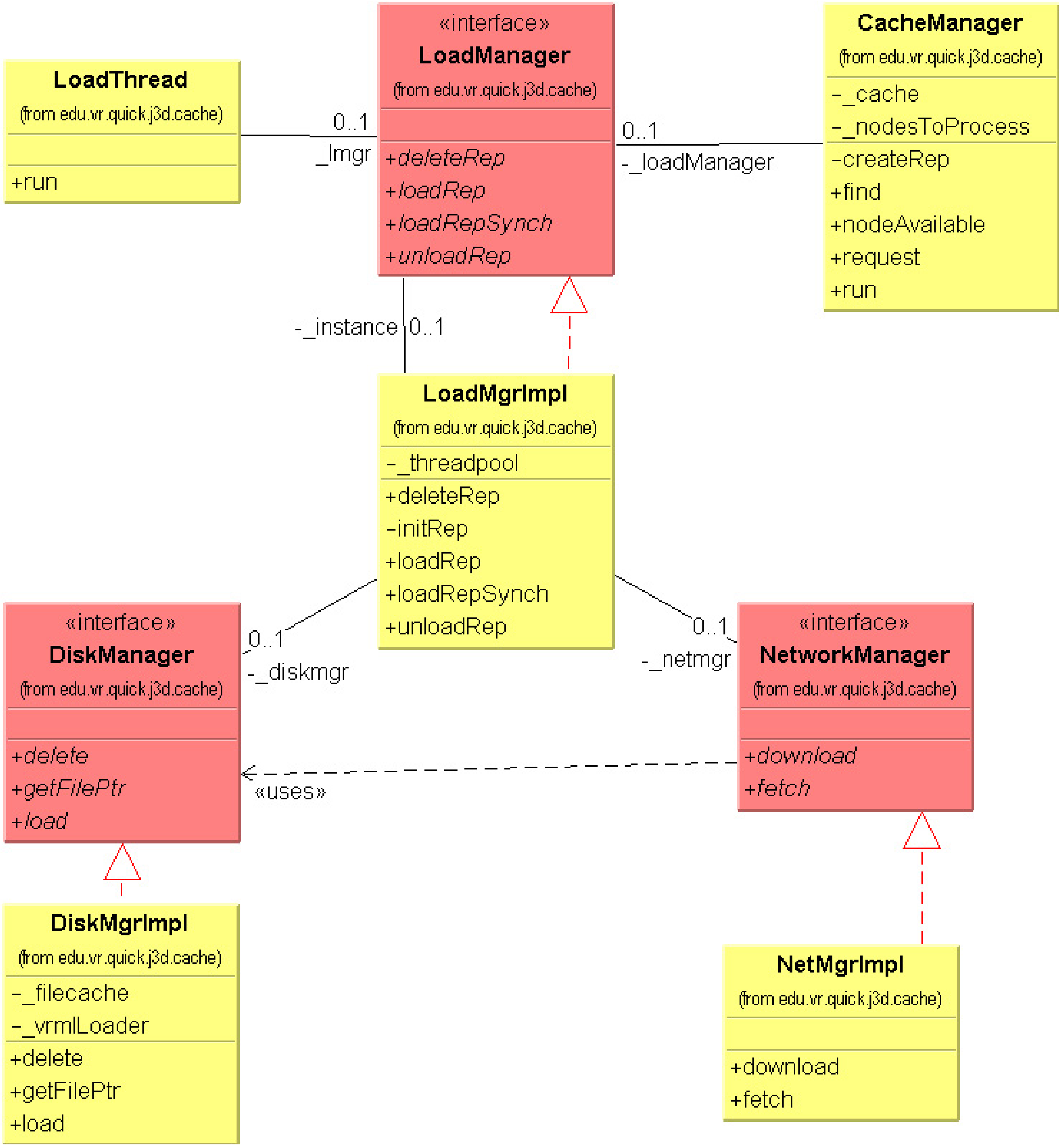


Figure 28. Class hierarchy diagram of the edu.vr.quick.j3d.cache package.

C. SWITCHING PACKAGE

The chooser package contains the implementation of the SwitchManager module discussed above in section VII.D.3. The abstract base class, conveniently named `SwitchManager`, can be extended to build managers for any purpose. The class is `Runnable` and can therefore be spawned into its own thread of execution; otherwise, the `pulse` function can be used for a single optimization pass. Either method uses the `SwitchManager.traverseTree` function, which walks through the scene graph processing the *QUICK* relevant control nodes (QNode, QSwitch, and Link). `SwitchManager` also contains a reference to the cache manager for requesting or deleting representations.

Figure 29 shows the contents of the package, which includes a number of concrete subclasses of `SwitchManager`. For example, the `LoadAllMgr` class overrides the `SwitchManager.processQSwitch` method such that each time a QSwitch is encountered on a traversal, any unavailable children are automatically requested. The more complex `DrawOptMgr` class overrides handlers for both QNode and QSwitch nodes, and uses them to construct a linear programming problem instance (member `_problem`). `DrawOptMgr` is in turn extended by the `DrawMaxMgr` class, which draws the highest Fidelity children of each QSwitch regardless of Cost. This is accomplished by using the structure of `DrawOptMgr`, building the optimization problem in the exact same manner, but at the last using an infinitely large Cost constraint.

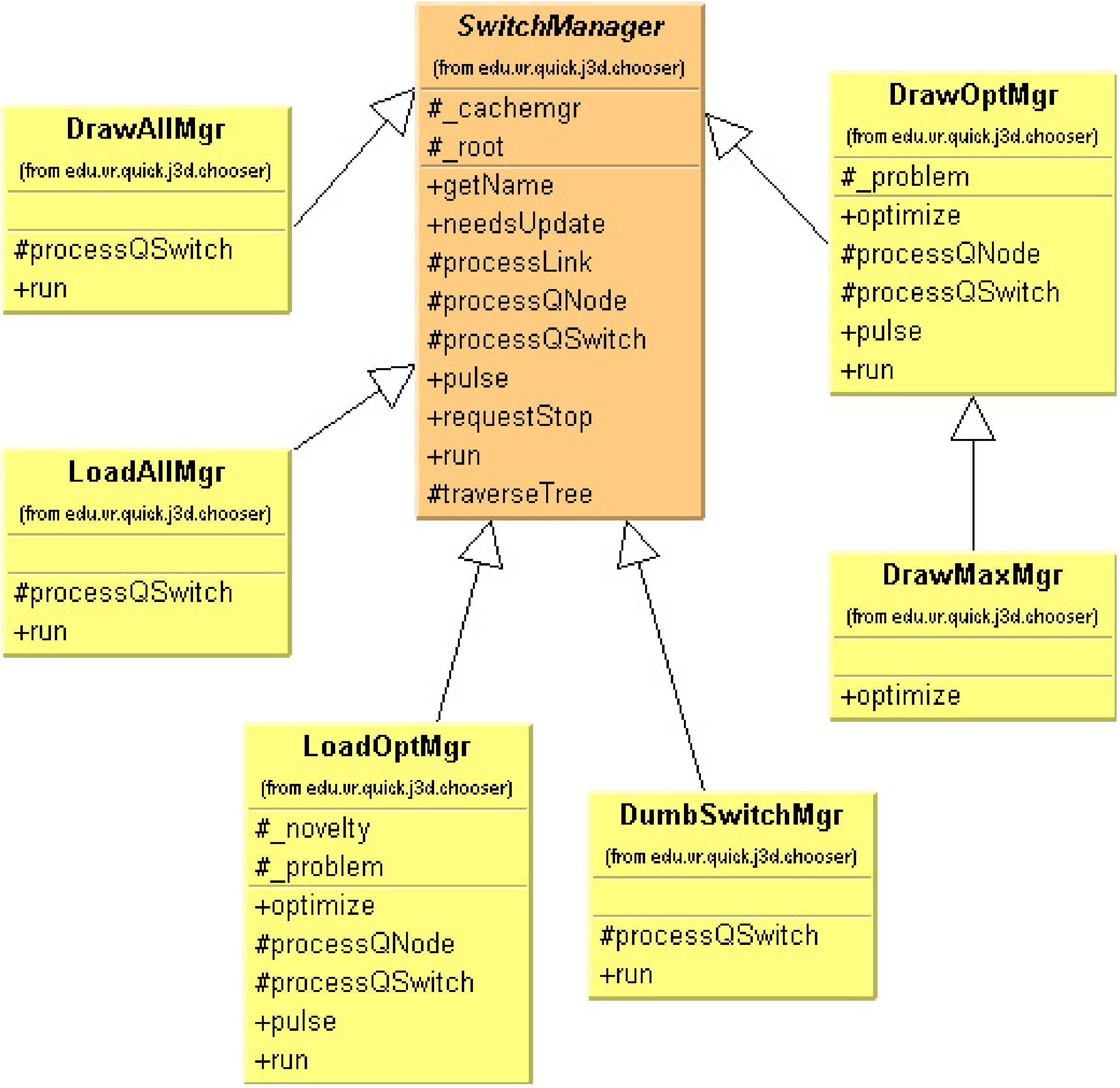


Figure 29. Class hierarchy of sample classes in the edu.vr.quick.j3d.chooser package.

D. OPTIMIZATION PACKAGES

This package contains the classes for building a linear programming problem from a *QUICK* scene graph. The `lpsolve` package is a Java port of the popular C linear programming library, LP_SOLVE. The port was performed by the Java group at Washington University at St. Louis; the code is available via http://www.cs.wustl.edu/ javagrp/help/LinearProgramming.html. This library was chosen primarily because the source code was freely available; this decision was fortuitous because modifications to the code were required to allow access to the final variable coefficients for the objective function. Additionally, the algorithms of the LP_SOLVE package have undergone significant community testing, and are considered to be more robust and scalable than most.

Figure 30 shows the relationship between those two packages and an optimizing switch manager. `DrawOptMgr` contains an instance of `QProblem`; as it traverses the scene graph, it calls the `registerQSwitch` and `addRep` methods to add the *QUICK* control nodes to the problem formulation. When adding a QSwitch, the `setImportance` method is used; the function for adding a QNode representation, `addRep`, expects arguments which indicate the computed Quality and Cost. When the problem formulation is complete, `DrawOptMgr` calls the `QProblem.solve` method and then one of the `apply*` methods to apply the new optimization to the scene graph.

During the traversal process, the `QProblem` class internally builds a `switches` Vector of `SwitchEntry` instances. These are used both for translating the optimization data into the linear programming matrix, and for translating the solution vectors back into changes to *QUICK* nodes. When `QProblem.solve` is invoked, the LP_SOLVE class from the lpsolve package is created. `solve` is the API for the problem formulation, and offers methods for adding constraints, constraining variables to integer values, and setting the optimization objective. An instance of `lprec` is passed to all of `solve`'s problem-building functions, and it contains the matrix representing the problem constraints.

E. PARSING PACKAGE

These classes take advantage of Java's guaranteed file loading order to interpose a slightly-modified parser into the Java3D VRML97 loading library. By placing this version of the library earlier in the CLASSPATH, certain classes can be made *QUICK* -conversant without replacing the entire package. Figure 31 shows a portion of the modified `Parser`'s interface. The `Parser` encounters node names in a text file and delegates the text contents of that node to a special class-specific parser of the same name. Therefore, the only modification needed was to register the four *QUICK* node names: `QNode`, `QSwitch`, `QCost`, and `QQual`. Since the `Parser` creates these classes indirectly, through their class names, their relationships are shown as a dashed line in the diagram instead of a standard "Creates" relationship.

The `QSwitch` parsing class shares many functions of the other grouping nodes, and so it inherits from the unmodified `GroupBase` class as shown. `QNode` is a superset of the VRML Transform node, and so its parser inherits from the unmodified `Transform` parsing class.

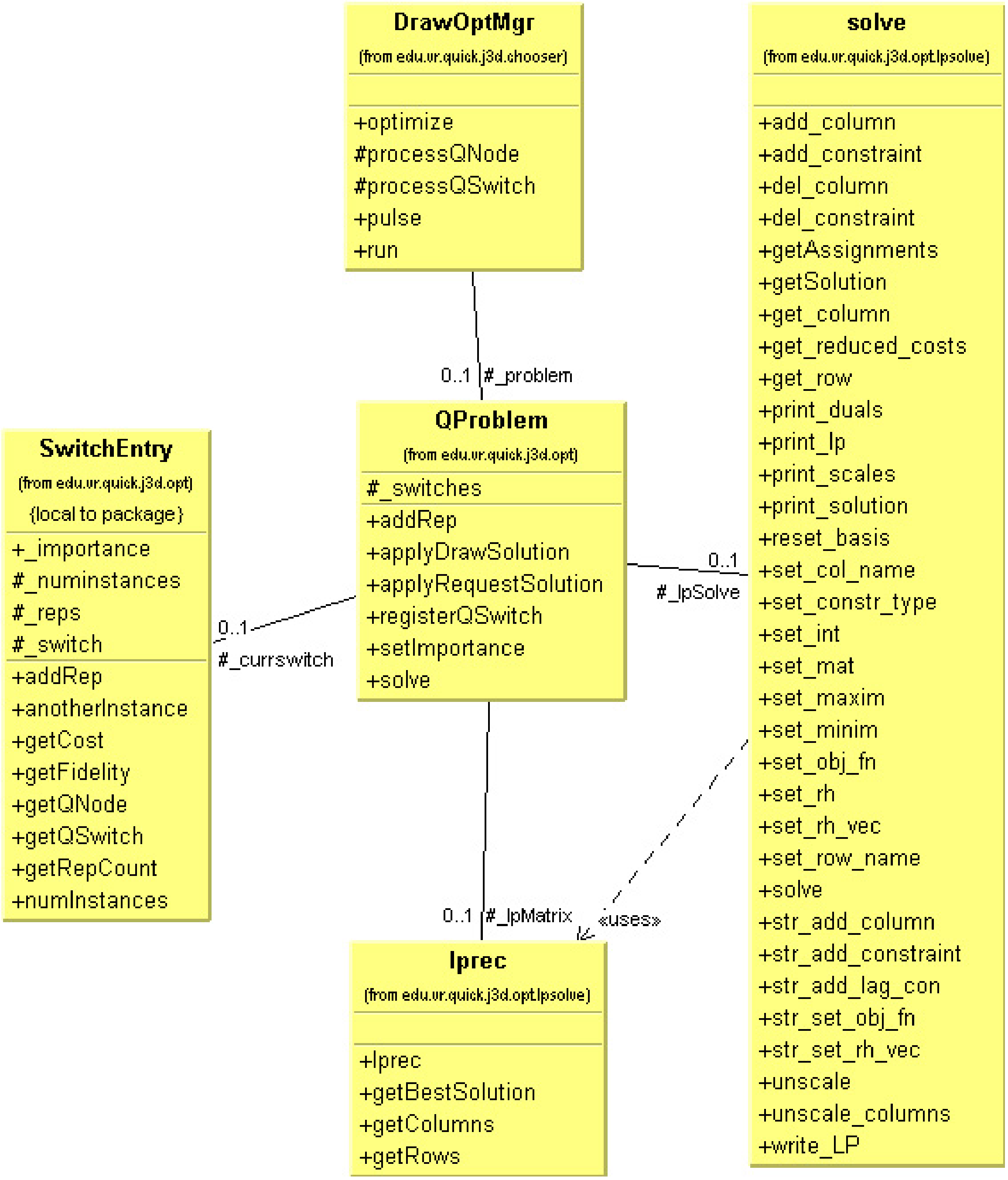


Figure 30. Class hierarchy diagrams for the edu.vr.quick.j3d.opt and ...opt.lpsolve packages.

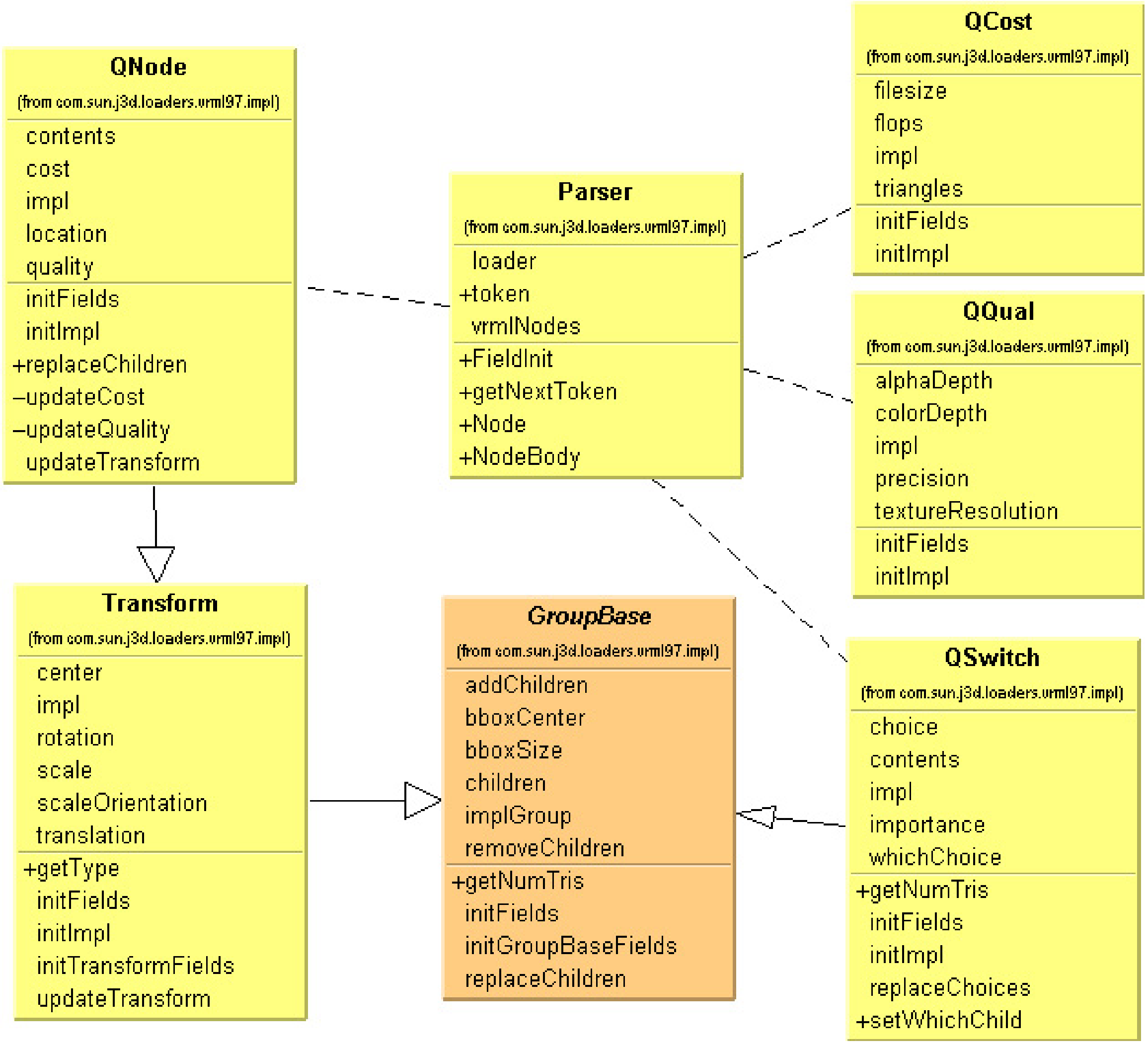


Figure 31. Added and rewritten classes in package com.sun.j3d.loaders.vrml97.impl.

F. UTILITY PACKAGE

This package contains utility and convenience classes used in packages throughout the system, shown in Figure 32. The `PushOnlyStack` is a special interface for a stack data-structure that does not allow "pop" actions. The special `Stack` class in this package is empty, but both implements the `PushOnlyStack` interface and extends the standard Java Stack class. This allows the creator of such a stack to use all normal stack functions,

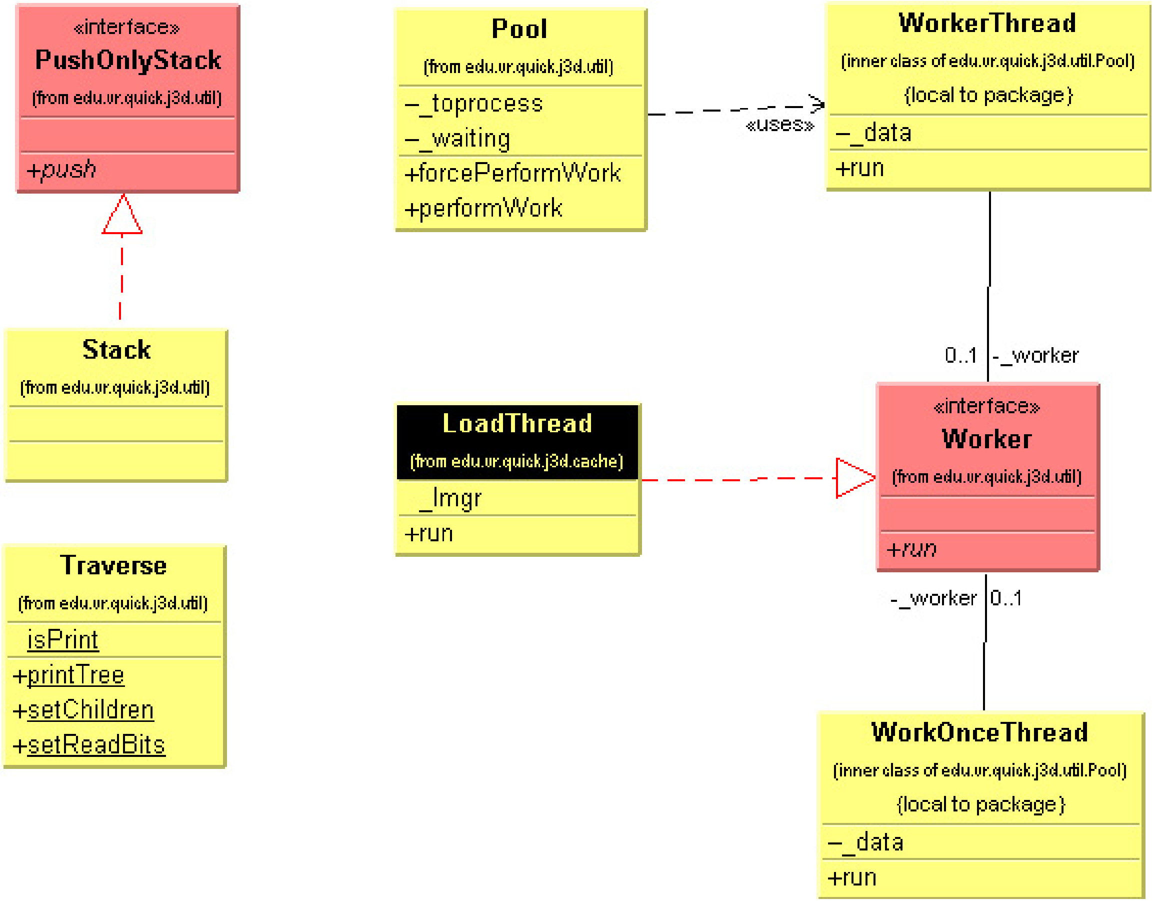


Figure 32. Class hierarchy diagram for edu.vr.quick.j3d.util.

but also to control access of client classes to the internals by offering only the restricted interface.

The `Traverse` class offers scene-graph traversal methods for standard tasks, such as printing the nodes in a tree. Another example, the `Traverse.setReadBits` method, searches a scene graph and makes the children of all group nodes accessible (which is not the default in Java3D).

An early version of *QUICK* made use of Java's thread facilities inefficiently. Rapid creation and lapsing of execution threads requires significant overhead that can be avoided if the computation needs are understood in advance. The loading and parsing functions occur in separate threads of execution, which reduces lapses of interactivity when waiting on a network or disk response. The `LoadManager` class now uses the `Pool` class to manage these loading threads. The `Pool` begins empty, and new threads are created as needed up to a certain maximum. When that maximum is reached, thread requests are placed in a FIFO queue; as threads become available, they take up the work requests in the queue. A review of threads and related operating system concepts is available in [Silberschatz and Galvin, 1994].

The threads in the `Pool` are `WorkerThreads`, which are created internally and not exposed to the application programmer. The application programmer creates a subclass of the `Worker` interface, such as `LoadThread` of the cache package. To initiate a `Worker`, it is passed an object argument of the data to operate upon; in the case of the `LoadThread`, a `RepID` identifier is passed in and the `LoadThread` runs the representation-loading process.

G. APPLICATION PACKAGE

This final package contains the application-specific classes for presenting a virtual environment client optimized for a specific task. The `VirtualWorld` interface contains methods for accessing the environment scene graph. All parts of the system can reach the singleton `Application` instance, and it contains a reference to the current `VirtualWorld`, so all parts of the system have read-only access to virtual world data.

In this case, `VirtualWorld` is implemented by the Java graphical user interface

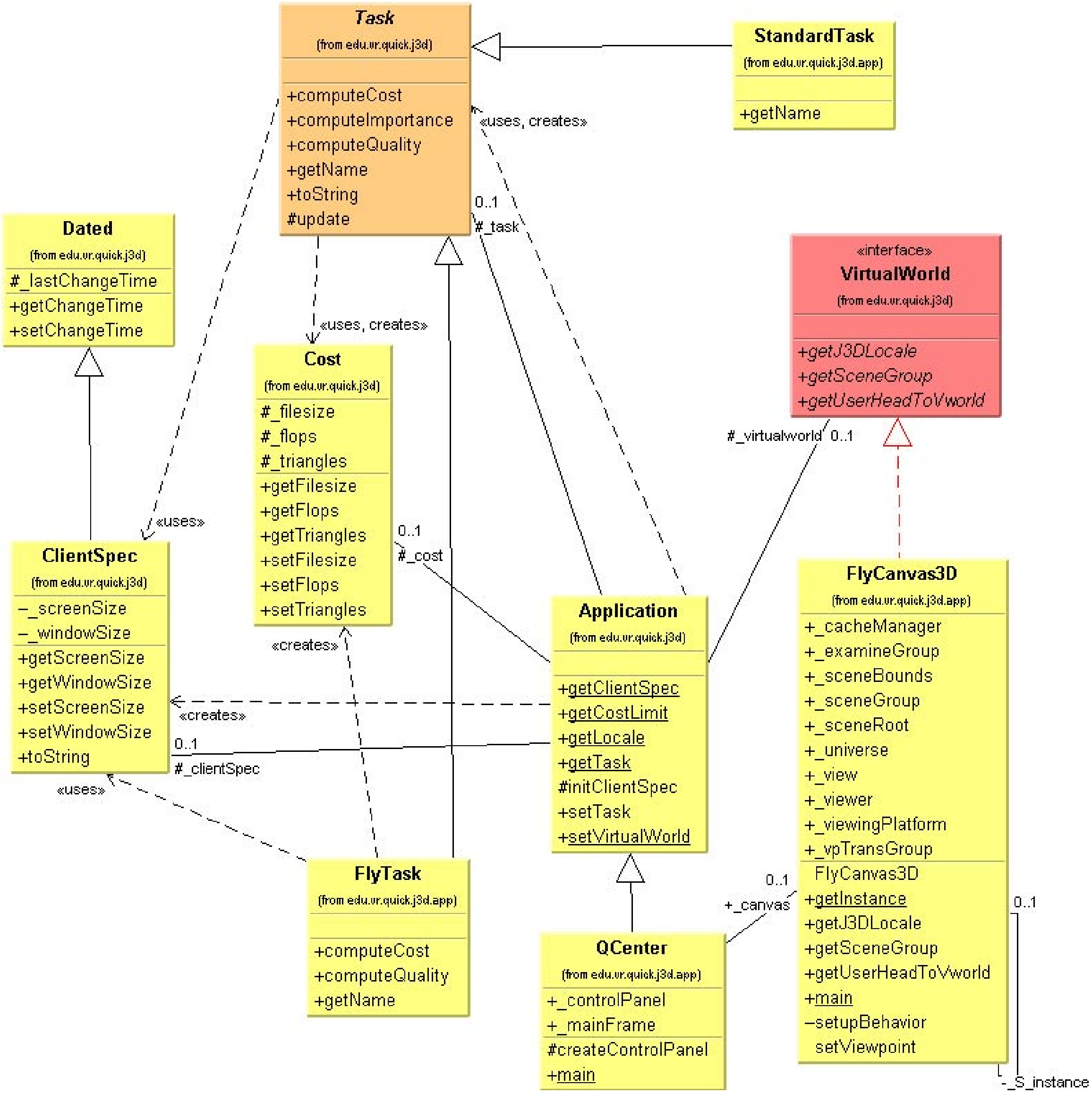
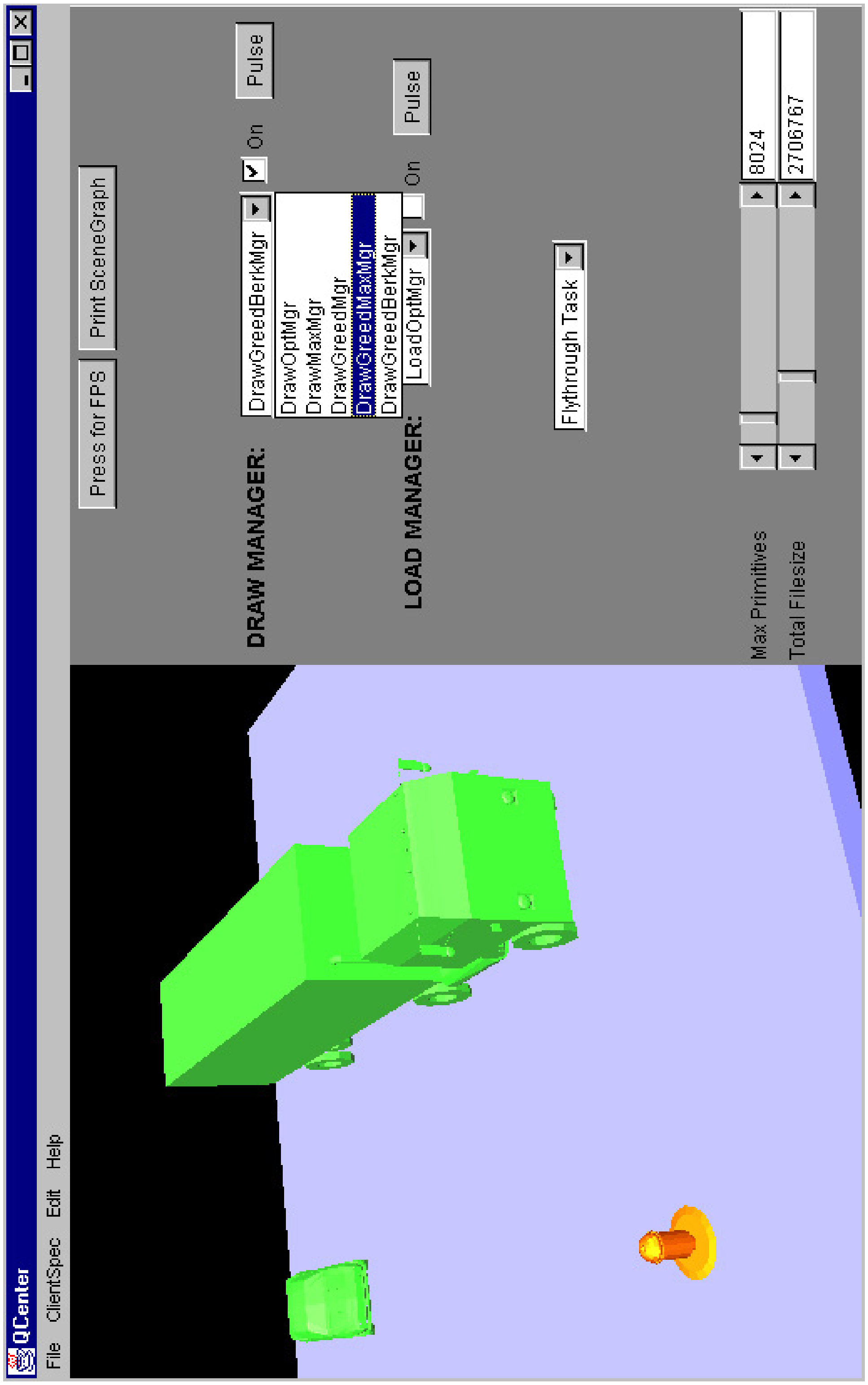


Figure 33. Class hierarchy diagram for application-building classes in edu.vr.quick.j3d.app.

component which contains a Java3D canvas: the `FlyCanvas3D`. That class contains its own main loop, so it can be run as the basis for an independent application, but its default behavior does not include any loading or switching capabilities. `FlyCanvas3D` controls access to the scene graph; it also contains user interface components such as navigation, frame-rate reports, and the like.

The `QCenter` class is the primary application for this *QUICK* implementation. It contains a control panel which allows the user to change almost all facets of the system during runtime—load managers, drawing optimizations, user task, cost thresholds, and even client specification. This design supports the experimental nature of this proof-of-concept system by offering a simple mechanism for adding new selections in those categories. Figure 34 shows a screen capture of the `QCenter` user interface.

A comparative analysis of the effectiveness of this implementation is available in Chapter X.



DRAW MANAGER: DrawGreedBerkMgr ☑ On

LOAD MANAGER: LoadOptMgr ☐ On

Flythrough Task

Max Primitives 8024

Total Filesize 2706767

Figure 34. QCenter screen capture.

X. ANALYSIS OF EFFECTIVENESS

A. INTRODUCTION

This chapter compares the *QUICK* system to other resource optimization systems by analyzing the complexity and effectiveness of their algorithms and implementations. This section contains a brief description of each analyzed system; most systems have been introduced in previous chapters. The next section contains a discussion of the drawing and request optimization processes. Each system is evaluated in turn with regards to both complexity and correctness. These evaluations are combined into recommendations for both preferred core algorithms and available implementations.

The analysis in this chapter focuses on the following six techniques, which were selected both for optimization effectiveness and to ensure adequate coverage of the technology space. The four-letter codes below are used throughout the chapter to designate both the systems and their resource-management algorithms.

PERF: SGI's Iris Performer [Rohlf and Helman, 1994] is a toolkit for building virtual environments that take advantage of SGI hardware rendering. Performer used a closed feedback loop to manage display resources.

BERK: The Berkeley Walkthrough [Funkhouser and Séquin, 1993] was the first project to investigate optimization for virtual environments. A Cost/Benefit heuristic was used to make display and cache request decisions. Further details on the Berkeley Walkthrough and Iris Performer are available in section II.H.

J3DV: Sun's Java3D [Sowizral *et al.*, 1997] graphics library, which serves as the basis for the initial *QUICK* implementation, has been described throughout this work. Java3D uses the same techniques as most VRML browser technology, so Java3D and VRML management techniques are combined into this single category. Further description is available in sections II.K and VII.C.

SPLN: Mitsubishi Electric's SPLINE [Anderson *et al.*, 1995] was designed for efficient navigation of distributed virtual environments. A user in a SPLINE environment navigates between multiple connected locales; management techniques operate on locales at the high level, and similarly to VRML at the lowest level. Further description is available in section II.K.

QGRD: The *QUICK* framework includes a Greedy optimization algorithm, as discussed in section C, which selects representations based on their Fidelity to Cost ratio.

QOPT: The final *QUICK* algorithm is the linear optimization method, as discussed at length in Chapter VIII.

B. ANALYSIS OF OPTIMIZATION EFFECTIVENESS

This section looks at the effectiveness of the *QUICK* optimization for managing the draw and request processes. It includes the exact computation using linear optimization, as well as the approximating algorithms from Chapter VIII. The discussion begins with a definition of correctness, which provides a basis for comparison between these disparate resource management systems. A description of the structure and complexity of the optimization techniques follows. Where applicable, those techniques are evaluated with respect to the given definition of correctness. Finally, this section draws upon these evaluations to provide an analysis of the comparative effectiveness and merit of the *QUICK* system.

1. Correctness

Defining the correctness of a subset of nodes selected for display has proven a frustrating experience. There is no definitive notion of what constitutes the correct assignment for switch-based scene graph elements. Generally, the highest-fidelity version is assumed to be the preferred selection for rendering—unless there are constraints on display resources. When resources are limited, lower-cost (concomitantly, these are usually lower-fidelity) nodes must be selected. Similarly, the preferred behavior for request management is to immediately request all available objects. When transfer bandwidth or local storage are limited, some representations must be omitted. Correctness in either case requires an absolute priority order that dictates the appropriation of the limited resource.

No such absolute priority order exists in the general case. Any scheme must account for the user task and current application state; a change in either can invalidate the priority ordering. This is exactly the lesson of the preceding chapters describing the *QUICK* framework: correctness cannot be obtained without incorporating dynamic factors. Correctness cannot be generalized accurately.

QUICK is the first customizable virtual environment management system designed to address the problem of correctness. This makes validation of the *QUICK* framework difficult, as there is little basis for comparison to previous work. *QUICK* incorporates factors omitted from other optimization methods, so it is trivial to find a problem configuration for which *QUICK* outperforms other algorithms. For example, many tasks yield priority orderings which are different from an ordering based on straightforward visual accuracy. One contrived example is an Obfuscation application, in which the user must guess about environment details from artificially-limited data. While that task can easily be factored into the *QUICK* optimization, general-purpose systems would fail by incorrectly striving for visual accuracy.

Therefore, this work postulates that the best definition for correctness is likely that which results from a properly-informed *QUICK* optimization. This is the only known technique which incorporates notions such as subjective quality and user task with objective information such as geometric precision and platform capabilities. In an effort to make fair comparisons with previous technology, the analysis below involves partially-disabled versions of *QUICK*. Complexity analysis for *QUICK* computations assumes that task-dynamicism is disabled, and that the default computations are used for each *QUICK* factor.

2. Optimization Techniques

The resource management strategies listed in the introduction use widely varying means for maximizing resource consumption. This section explains the drawing and request optimization processes in each of those systems, as well as the complexity of those processes. In all systems below that do perform request optimization, the optimization algorithm is the same as is used for drawing optimization.

Complexity results are given in terms of the number of scene objects, n, and the total number of representations, r. The r is generally larger than n, but since objects with no representations are legal, these values are reported separately below.

These complexity analyses are broken into four phases:

- **Precomputation Phase.** Some systems depend on a preprocessing step before execution. While this does not directly affect rendering times, the significant complexity of precomputation can often play an important role in algorithm selection. Generally, no precomputation phase is necessary, and its discussion is therefore omitted for many of the systems below.

- **Initialization Phase.** The setup phase in which the problem is formulated. Determining coefficients in constraints might require only a straightforward memory access, or may involve some computation such as in the case of distance-attenuation. Generally, the more exact the optimization, the longer the initialization phase.

- **Optimization Phase.** This is the process that decides which objects are included in the display set, as well as which representation will be used.

- **Application Phase.** The final phase is to apply the results of the optimization phase to the display set, or to request new nodes from the environment server. This is usually an $\Theta(n)$ operation, and is only included in the descriptions below if there is significant variance from that complexity.

For the considered systems, the optimization complexity is as follows:

a. PERF

Performer LOD nodes each include distance values similar to that shown in Figure 10 in Chapter VI. Each representation has an associated distance from the eye at which it is displayed. The application specifies a target frame rate; if that frame rate is not met, the draw load is reduced by modifying LOD transition distances. The initialization phase, which requires determining the distance to the eye from each object, is $\Theta(n)$. The optimization phase takes $O(r)$ time because transition distances can overlap, so more than one representation may be drawn per object.

Performer does not support networked environments, so it does not include request optimization. It does support paging between disk and memory for large models.

b. BERK

The Berkeley Walkthrough makes LOD decisions based on a Cost/Benefit ratio similar to (and inspiration for) the *QUICK* Greedy algorithm. The Walkthrough uses a multi-step approach to determine the benefit gained from any given representation. The first step is the removal of objects not within the potentially visible set (PVS), which is determined in a precomputation step. Static cell-to-object visibility is combined with the current viewing frustum to find all visible objects. For each visible object, the Cost and Benefit are determined in a manner quite similar to the *QUICK* factor computation. Cost is based upon number of primitives and pixels; Benefit is based upon nearness to the screen center (to approximate focus), precomputed model accuracy, and screen area.

The precomputation phase is costly; in experimentation, building environments able to be rendered in real time took hours to preprocess. Given a division of a model into c cells, cell inter-visibility is an $O(c^3 logc)$ computation, followed by an $O(clogn)$ determination for cell-to-object visibility. Because cells are generally created for a fixed number of objects, this equates to $O(n^3 logn + nlogn) = O(n^3 logn)$. These steps presuppose the existence of the cellular spatial subdivision of the environment, an extremely complex operation. The runtime initialization phase requires screen position and size information, as well as memory accesses for precomputed descriptions of each representation, yielding a total running time of $\Theta(n + r) = \Theta(r)$. Of course, if the visible object set is small, there is a significant constant factor reduction.

The computation phase uses a greedy algorithm, which sorts the representations by Cost/Benefit ratio in a manner similar to the **QGRD** algorithm. Representations are selected by ratio; any remaining Cost is used to replace original selections with representations that give higher benefit. The optimization phase is $O(rlogr)$; additionally, some coherence in values between optimization passes means that this value is usually lower in practice.

The Berkeley Walkthrough made a number of advances to the state of the art in database management for large-scale virtual environments. Such environments require precaching of objects and asynchronous disk management to prevent lapses in interactivity. By combining tightly-constrained environments, precomputed visibility, and motion parameters the system is able to predict the minimum time until an object could possibly be within view. The request process computes the shortest path to each cell and combines the computed prediction times with the Cost/Benefit optimization. The shortest path computation uses Dijkstra's method, hence the complexity of $O(c^2) = O(n^2)$.

c. J3DV

Java3D and VRML both offer distance-based LOD selection similar to Performer. However, neither system incorporates any adaptation to changing resource availability. There is no initialization phase, as there are no dynamic variables in the decision. The draw optimization phase is $\Theta(r)$, since the distance interval for an object is determined by linear search of the representation distance values. These systems usually load networked resources (Inline nodes) immediately upon discovery, with no real decision process—hence a $O(1)$ running time in the table.

d. SPLN

SPLINE is included in this chapter because of its network management; it offers little in the way of display optimization. It uses a visibility step similar to that in the Berkeley Walkthrough, but at a the much higher granularity of environment regions rather than rooms. That step is combined with VRML LOD processing for each object in those worlds, similar to **J3DV** above. The initialization phase is an $O(1)$ adjustment to top-level scene graph branch nodes; when a locale is not visible, all of its constituent objects are removed from the display subset in single step. The optimization phase complexity is the same as for **J3DV**, $\Theta(r)$ for draw and $\emptyset(1)$ for request.

e. $\boldsymbol{QGRD\ and\ QOPT}$

The complexity for these algorithms has been discussed in Chapter VIII, and is only summarized here. Only the default computation is included in the complexity, in an attempt to normalize the comparison with other systems. The two *QUICK* algorithms share a precomputation phase, which is the annotation process for representations; since there is no interaction between representations at this stage, it is considered $\theta(r)$. Similarly, they share an initialization phase; the primary dynamic component is distance attenuation, which is computed on a per-object basis, yielding complexity $\Theta(n)$. The optimization phase for **QGRD** is the same as **BERK**, $O(rlogr)$ for the sort prior to the greedy algorithm. **QOPT** is NP-complete, and therefore its running time is exponential: $O(2^n)$. While the request optimization can incorporate motion prediction algorithms, the default task computation for request is identical to the drawing optimization.

3. Experimental Results

Because of the hidden constant factors, any complexity comparison between these optimization algorithms would be improved by comparing implementation performance. However, a direct computational comparison of program execution with identical models on identical architectures is not possible. The Berkeley Walkthrough, for instance, has only

ALGORITHM	PERF	BERK	J3DV	SPLN	QGRD	QOPT
Precomputation Phase:	n/a	$O(n^3 logn)$	n/a	n/a	$\Theta(r)$	$\Theta(r)$
Initial Phase:	$\Theta(n)$	$\Theta(r)$	n/a	$O(1)$	$\Theta(n)$	$\Theta(n)$
Draw Optimization:	$O(r)$	$O(rlogr)$	$\Theta(r)$	$\Theta(r)$	$O(rlogr)$	$O(2^n)$
Request Optimization:	n/a	$O(n^2)$	$O(1)$	$O(1)$	$O(rlogr)$	$O(2^n)$
Application Phase:	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Table II. Comparison of drawing optimization complexity.

been used on the Soda Hall model which was modified expressively for that system. The code is no longer actively maintained, and all published execution times were recorded on SGI machines which are no longer in production. SPLINE is limited to the Microsoft Windows platform. While it uses VRML models, giving some basis for comparison, it too is no longer supported.

The remaining systems all are capable of displaying VRML models. In fact, Iris Performer is an actively-developed commercial product which has been optimized for the SGI platform for nearly ten years. It has been performance-tuned for the SGI Irix operating system, threading model, and graphics pipeline. All of Performer's libraries and applications are natively-compiled C++.

Java3D is available on many platforms, SGI included; however, the SGI implementation is an unoptimized preliminary release. No portable Java program can compare in run-time efficiency to natively compiled libraries, especially when it requires frequent access to system resources. In this case, the gap in implementation effort has an even greater impact: for years, SGI hardware has been designed specifically to accelerate Performer, while the SGI port of the Java3D library has not yet reached full functionality.

Therefore, while Performer and Java3D share a platform and a model format, there is little to be gained by directly comparing their application performance. The *QUICK* proof-of-concept implementation is based upon Java3D, so *QUICK* and Performer execution times are not compared for similar reasons.

a. Execution Times

Asymptotic complexity gives a useful basis for comparison, and as previously stated, the only possible basis for comparing these resource management systems. However, it is possible to directly compare computation times of the multiple *QUICK* algorithms in the Java3D implementation. This section compares the **QOPT** and **QGRD** algorithms, as well as a third **QFST** algorithm. The **QGRD** algorithm sorts representations by their Fidelity/Cost ratio, and then makes selections with replacement to maximize usage of available resources. The **QFST** ("QUICK-FAST") algorithm does not allow replacement, so it stops when a valid representation has been chosen for each QSwitch, regardless of any remaining available resources.

All timing results were determined on an SGI 320 WindowsNT workstation, with dual 450Mz Pentium II processors, 96MB of graphics memory,and 160 MB of main memory. Missing timing values for **QOPT** are due to memory limitations; those limitations were usually a factor only after the running time had exhibited exponential growth. In all cases, only the display optimization phase is included in the timing results, since initialization is identical for the three algorithms.

Number of QSwitches	Zero Resources	Average Resources	No Constraints
100	80	1120	80
200	140	4090	130
500	380	23200	390
1000	1220	n/a	1220
2000	4770	n/a	4860
3000	11190	n/a	11280

Table III. Running times for QOPT (in milliseconds), varying resource availability.

The *QUICK* problem has far too many free variables to allow testing of all possible instances. However, some variables have little influence on algorithm running time, so it is possible to simplify this comparison by picking representative values in those cases.

The first set of experiments explores the effects of resource availability on computation time. Table III shows the running times, in milliseconds, of the **QOPT** algorithm. In the experiment, each QSwitch was given four associated representations, with varying Fidelity and Cost values. The "zero resources" and "no constraints" cases allowed no and any representations to be selected, respectively. The "average resources" case included more than enough for one representation to be chosen for each object, but not enough for the costliest to be chosen in each case. Even though the implementation could not compute the running times for all instances, the graph in Figure 35 shows a clear difference between the average and boundary cases. This difference is expected with branch-and-bound linear optimization techniques such as are used in this implementation; prediction of running time for a given instance is in itself an NP-complete problem.

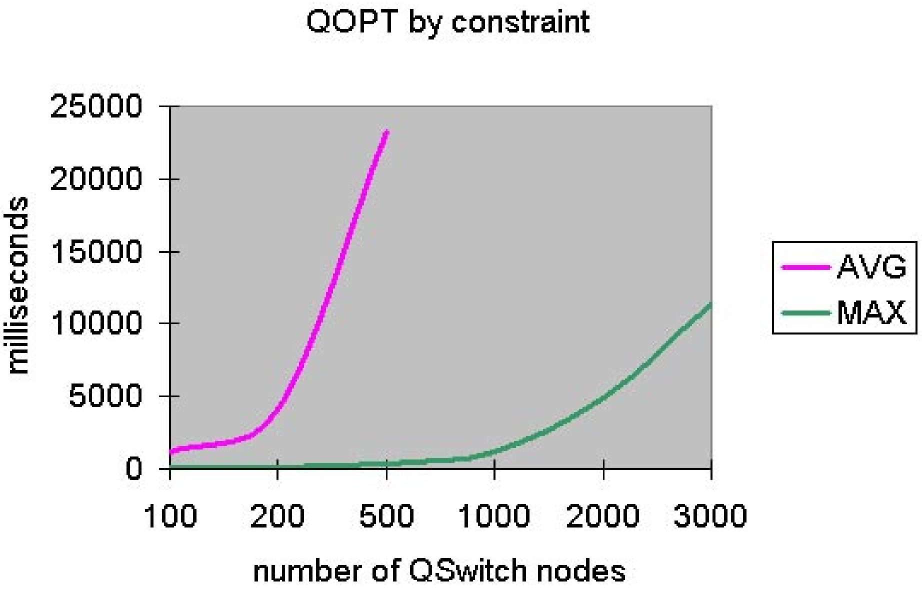


Figure 35. QOPT running times with average and maximum resources.

In testing running times for the **QGRD** and **QFST** algorithms, it was necessary to use much larger problem instances to have statistically significant timing information. Table IV and Figure 36 show a small but noticeable difference between the two algorithms, even though the asymptotic complexity for both algorithm is $O(rlogr)$. However, the sorting step hides the $O(r)$ replacement step in **QGRD**, which clearly has a high constant coefficient. The important result for this data, though, is that there is no major impact on computational complexity from variance in resource availability.

Combined with the data regarding **QOPT**, it now seems appropriate to choose an average resource complexity for direct comparison of these algorithms. Each of the algorithms is run with three cases:

Algorithm, # QSwitches	Zero Resources	Average Resources	No Constraints
QGRD,10000	480	520	540
QGRD,20000	810	890	930
QMAX,10000	470	480	490
QMAX,20000	820	820	830

Table IV. Running times for QGRD and QMAX (in milliseconds), varying resource availability.

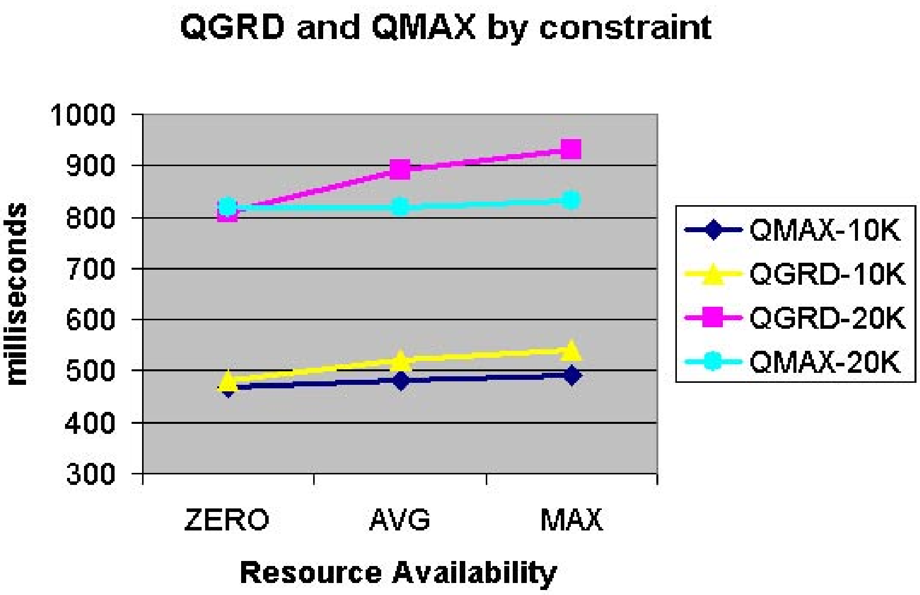


Figure 36. QGRD and QMAX running times with average and maximum resources.

- One object, with a varying number of representations
- A varying number of objects, each with one representation
- A varying number of objects, each with four representations

The last case is the most typical instance, in which each object has a small number of possible representations. In each case, variation is done by exponential steps

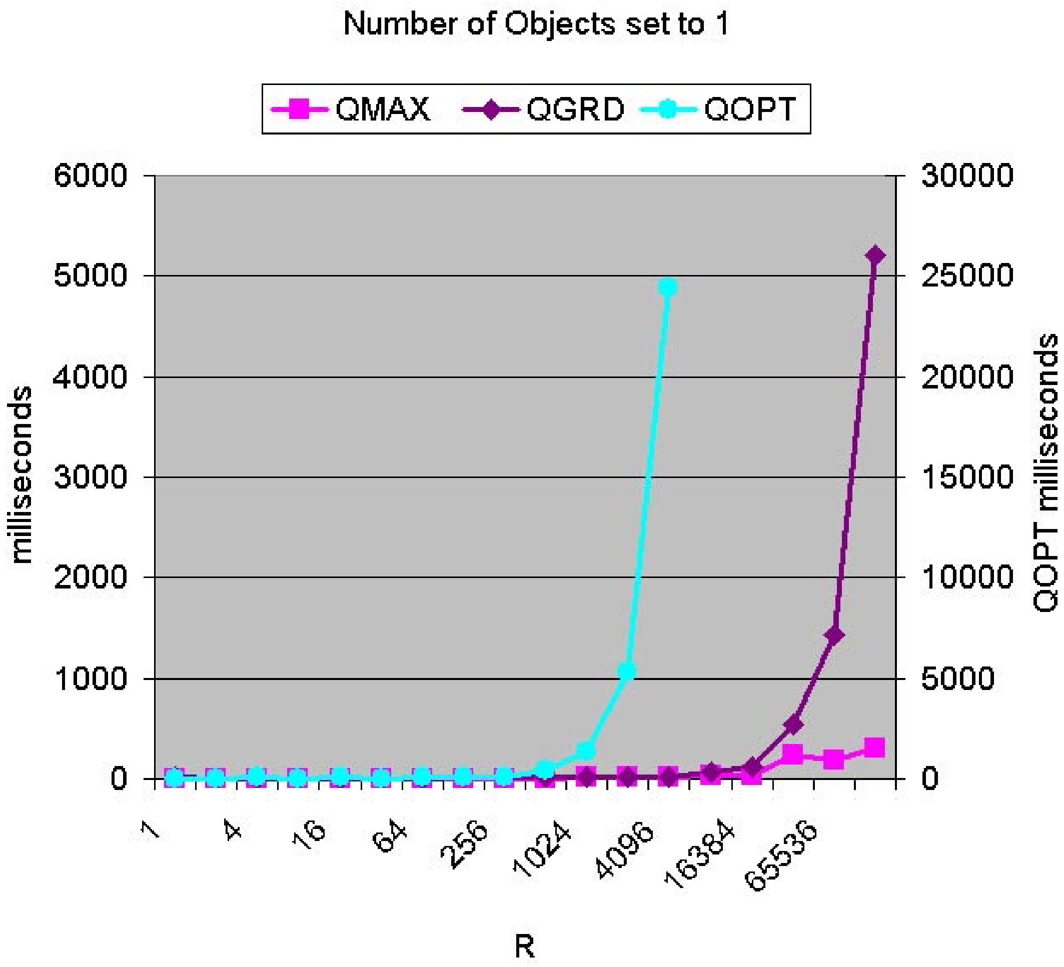


Figure 37. Running times compared with N=1 and R=2^i.

over the values 2^0 to 2^{17}. As is expected, the **QOPT** algorithm was rarely able to provide values for the most complex problems in each case—either due to memory restrictions, or the limitations of a human life-span.

In the graphs, two oddities merit mention. The first is a reminder that the x-axis increases logarithmically. The second is that, in order to combine values, the **QOPT** algorithm is graphed against the right-most y-axis. Therefore, **QOPT** complexity outpaces the other algorithms much more rapidly than a casual glance would indicate.

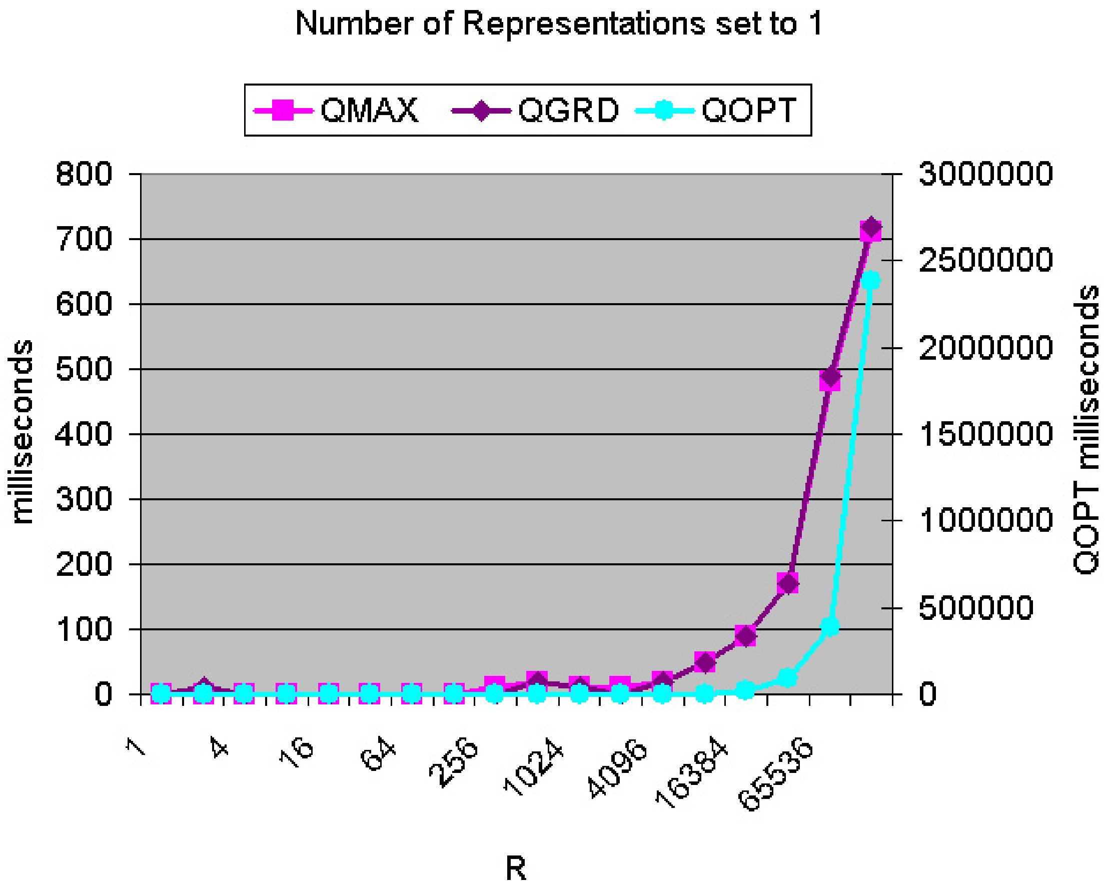


Figure 38. Running times compared with N=2^i and R=1.

These results have a number of indications for the use of the *QUICK* framework. For instance, the **QGRD** and **QMAX** algorithm perform identically in the case where there is only one representation per object. This reflects the fact that both algorithms must visit every representation after sorting. While it is not surprising that the **QOPT** algorithm does not scale well, given its NP-complete nature, it is heartening to see that problem instances with one thousand objects or more can be optimized interactively. This led to changes in the current implementation to support adaptive algorithm selection.

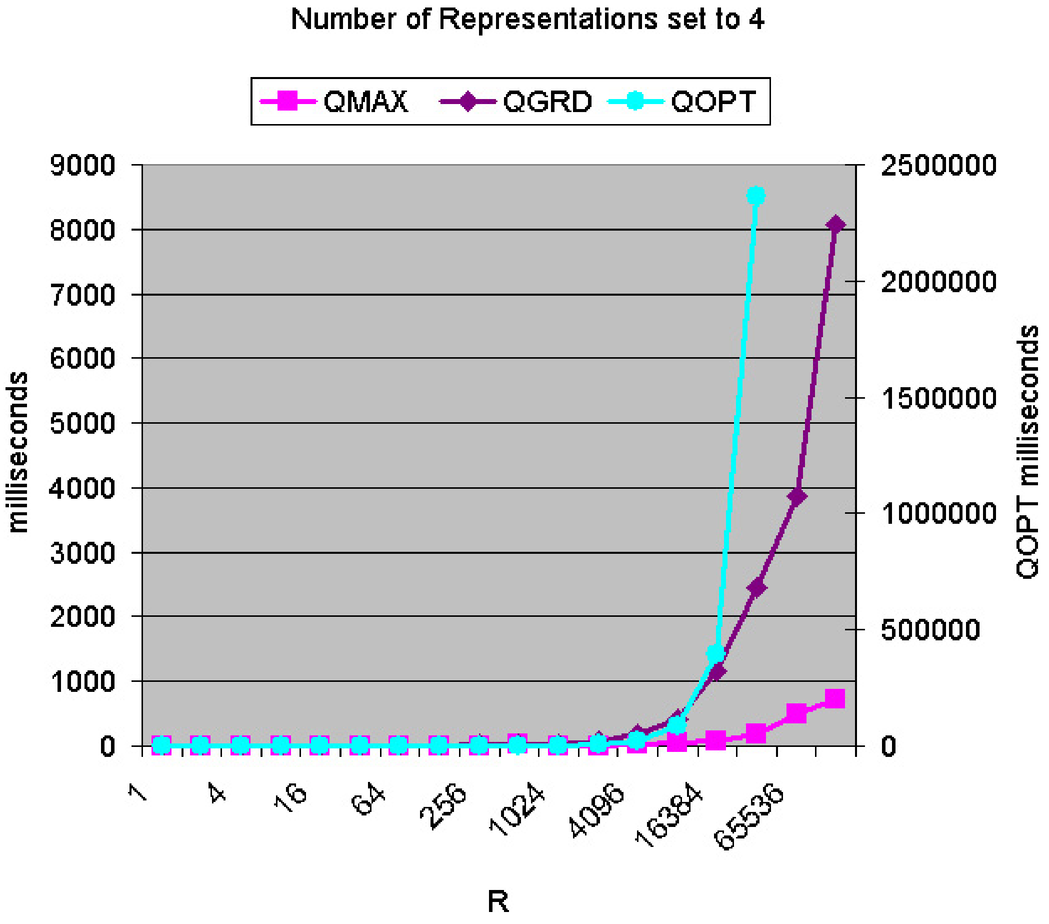


Figure 39. Running times compared with N=2^i and R=4.

4. Conclusions

Given the analysis above, it is possible to consider the effectiveness of the *QUICK* framework. First, the resource management algorithms are considered independently of their implementations; the systems as a whole are compared later.

a. Algorithm Comparison

The **PERF** algorithm has the best asymptotic running time of any of the algorithms considered, and in fact runs as quickly as systems which do not incorporate resource load. However, the Performer algorithm bases all of its optimization decisions on a single floating-point value for each representation. These distances are a pale indication of a cost/quality trade-off, and are insensitive to client capability. The use of a total ordering for representations, defined by the viewing distance thresholds, assumes that Quality and Cost scale directly. While this is often true in polygonally-defined environments, this dissertation has demonstrated a large space of problems in which Quality and Cost are not related.

The Berkeley Walkthrough (**BERK**) algorithm provides excellent run-time interactivity: $O(r + nlogn)$ for draw optimization, and $O(n^3)$ for request optimization. However, it is limited to environments filled with axial occluders. That, coupled with the requirements for preprocessing, make it inappropriate for use in a distributed system with general-form environments. If taken independently of the complete Berkeley system, the **BERK** Cost/Benefit algorithm is essentially a functional subset of **QGRD**. For example, **BERK** includes platform capability in the Cost determination, but those values are computed statically prior to execution. Similarly, the algorithm does not provide for task adaptability; visual realism was always the primary intent of the Berkeley Walkthrough.

The **J3DV** algorithm, shared by Java3D and most VRML browsers, is recommended only when bare simplicity is needed. The model annotations it requires are the same as those needed for **PERF—**which has similar asymptotic complexity but yields a resource-conscious optimization. The SPLINE algorithm is the same as **J3DV** from a rendering perspective.

By comparison, the linear-optimization *QUICK* algorithm offers the most customization choices. It is sensitive to all major factors which impact display and request correctness, and all of those factors can change during execution if necessary. The initial problem formulation is not significantly more expensive than those of the **PERF** or **BERK** algorithms. However, the worst-case complexity of the **QOPT** optimization phase is prohibitive for interactive display of very large models. The **QGRD** reduces that running time to tractable levels, at the cost of reduced accuracy in the optimization. Still, for approximately the same running time, the accuracy of **QGRD** is greater than **BERK** or **PERF**, as it has more data to guide the optimization.

In summary (assuming constant complexity), the algorithm recommendations are:

- When correctness is the primary concern, use the *QUICK* linear optimization algorithm (**QOPT**)
- When correctness and speed are both important, use the *QUICK* greedy algorithm (**QGRD**)
- When speed is vital, especially when no annotation information is available, use the Performer algorithm (**PERF**)

b. Implementation Comparison

A comparison of the available implementations of these algorithms requires a separate discussion. Separating algorithm from implementation is most cases straightforward; reimplementing the algorithms is certainly not.

Both the Berkeley Walkthrough and SPLINE systems are no longer supported, nor are they publicly available. Iris Performer requires a license fee, and is limited to the SGI platform, but as previously mentioned the implementation is well tuned for performance. Performer has a large support base and extensive documentation is available.

Binaries and source code for Java3D and *QUICK* are freely available, as is the VRML specification. The *QUICK* implementation is a super-set of the Java3D VRML library, and therefore contains all functionality of **J3DV** described above. For optimal performance, *QUICK* requires additional annotation information; it relies on Java3D for scene management of unannotated VRML files. Because *QUICK* is a functional superset of **J3DV**, it is recommended in all instances over plain Java3D or other open-source VRML browsers such as blaxxun's contact.

The *QUICK* implementation was designed in a modular fashion to simplify incorporation of new algorithms. Any of the algorithms discussed above could be added to the *QUICK* implementation, much more quickly than by designing a complete system. For instance, the **PERF** algorithm could be used by adding a distance threshold annotation to each representation (QNode), and writing a special task that would query resource consumption before each optimization pass. Other algorithms could be incorporated with similar effort.

In summary, the implementation recommendations are:

- When licensing fees are not a factor, model annotation is not possible, or robustness and support are of primary concern, use the Performer implementation (**PERF**)
- When extensibility or source code are required, correctness is paramount, or network support is required, use the *QUICK* implementation (either **QGRD** or **QOPT** depending on the situation)

XI. CONCLUSIONS AND EVIDENT EXTENSIONS

This final chapter provides a summary of the findings presented in this dissertation. The first section highlights the major contributions of this work, with special attention to results and implications relevant to other virtual environment resource management systems. This is followed by a discussion of the practical impact of this dissertation, and strategies for how these techniques might be applied in production systems.

The worth of a research effort of any magnitude can be judged both by the problems it solves and the new questions it raises. Accordingly, this chapter concludes with an annotated list of recommended extensions and avenues of further inquiry.

A. CONTRIBUTIONS

The *QUICK* framework offers a fundamentally new approach to resource management for virtual environment display and transfer. The underlying concept is simple: to maximize representation Quality for the most Important objects, while keeping the total representation Cost within defined constraints. Allowing the computation process for Quality, Importance, and Cost to vary during run-time allows tremendous expressivity in the resulting optimization.

It is uninteresting, however, to claim universality by merely including a programming interface for customization. The *QUICK* framework is so named because it defines the conventions necessary to make customizing optimization a straightforward process. The annotations recommended herein are practical and demonstrated, and are needed to determine the three *QUICK* factors.

Traditional resource management techniques attempt to support a single overriding application purpose—the user task. The *QUICK* framework allows dynamic modification of user task parameters, thereby encouraging reusability of algorithms and software. Similarly, *QUICK* tracks display platform capabilities during execution, so that updated constraints can be incorporated into the optimization. The combination of the two yields a new class of flexibility in virtual environment applications. *QUICK* defines conventions for specifying both user task and platform capability, as well as general-form ontological content description to support task computations.

The more data available for an optimization, the higher the accuracy of the result (assuming the optimization formulation and data are correct). The closest predecessor system, the Berkeley Walkthrough, uses only a fraction of the *QUICK* data set for its cost/benefit algorithm—and most of those values are not allowed to vary during execution. *QUICK* yields more accurate results, with equivalent or less time, than any competing algorithm. For a large portion of the problem space (generally, any tasks in which visual accuracy is not the sole concern), *QUICK* is the only viable algorithm available.

This dissertation includes the description of an architecture, and associated implementation, for virtual environment optimization. It includes a linear optimization algorithm which guarantees correct assignment (and slow computation), as well as faster approximation algorithms. This initial implementation was designed for experimentation with new types of tasks, annotations, optimization algorithms, and platforms. It is therefore hoped that the public release of this fully documented application framework will spur follow-on research.

B. APPLICATION

During the history of computer graphics, the growth of desired model complexity has generally out-paced improvements in rendering technology. Yet while this dissertation effort was accomplished, the polygon processing capability of commodity graphics hardware has increased more than anyone could have foreseen—by two orders of magnitude. Some argue that algorithms which trade accuracy for speed (such as level of detail techniques) will soon become unnecessary.

The utility of *QUICK* for optimizing consumption of the rendering resource will likely diminish over time, except in narrow problem spaces such as the visualization of very large graphical databases. Availability of network bandwidth and other vital resources are not increasing as quickly, however, so *QUICK* is therefore expected to remain a useful method for management of distributed virtual environment systems.

For client-server systems such as VRML environments, the primary hurdle for adoption of *QUICK* is content annotation. Chapters V and VI explained how *QUICK* annotations can be determined automatically to modify pre-existing content. Even automated processing is inconvenient given the many and varied VRML models already in existence. An intelligent browser might determine much of the annotation information during runtime after requesting a file, but that implies that the *QUICK* optimization cannot be used for object request.

For distributed worlds, decoupling annotations from the files they describe can lead to synchronization problems. This is especially true in the case of heterogeneously authored environments. Content inclusion in VRML worlds is normally performed by specifying solely an Internet location; there are no guarantees that the contents of that location will remain unchanged between sessions. In such an uncontrolled Web-based architecture, it is appropriate to store annotations within the files they describe, and make a query for those characteristics during execution. Further work in the creation, usage, and maintenance of CVE databases (and meta-databases) is warranted.

Modifying VRML to support *QUICK* annotations is possible with the PROTO node format, but is inconvenient and inefficient. This dissertation does not recommend general use of the modified version of VRML used in the *QUICK* implementation. Rather, the next generation of VRML (X3D) allows incorporation of multiple execution profiles for exactly this purpose. X3D is specified in XML, which additionally lends itself to communication of structured data of the sort needed by *QUICK* .

C. FUTURE WORK

As with most dissertation efforts, the original expectations for this project were higher than was realistic for timely completion. Also, issues arose during this research that were out of the project scope but merited further exploration. This section lists both suggestions and plans for future efforts in this area.

1. Extensions for Display

The first set of extensions pertain to the display optimization, both for improving its results and increasing its utility.

a. Annotations

The set of model annotations often grew or changed in response to the addition of new tasks. The *QUICK* framework currently includes approximately ten different task computations. Additional tasks will likely lead to further refinement of the annotation set.

b. Quality from Human Performance

While no exact specification of human capability exists, sufficient psychometric testing has been performed in narrow application domains. The process of military vehicle spotting, for example, involves a combination of visual and semantic information which leads to identification. Through experimentation, the United States military was able to determine the distances at which a vehicle's type, nationality, or even model might be identified [O'Connor *et al.*, 1996]. Incorporation of such information into the *QUICK* framework might provide a scientific, quantitative basis for Quality.

c. Semantic World Rules

By their very nature, virtual environments are not constrained to mimic physical reality. World rules define the action and interaction of objects and entities in a virtual environment; examples range from altered gravity and inelastic collisions to context-sensitive social rules. The definition and implementation of such semantic interactions is an open problem for all but the most limited domains. Such information, when available, could significantly enhance the *QUICK* Importance generation process.

d. Graduated Visibility Set

The Graduated Visibility Set (GVS) is a technique similar to the Potentially Visible Set: it determines visual occlusion between two finite geometric spaces. The Graduated Visibility Set is so named because it stores visible nodes in graduated levels—full visibility, totally occlusion, and steps in between. *QUICK* optimizations are best performed on continuous data values, rather than the binary on/off information of a PVS. The additional granularity of the GVS facilitates improved dynamic Importance determination.

e. Hybrid Representations

The original impetus for this work was to extend the hierarchical image caching efforts of the University of Washington, which combined billboarded textures with geometric representations, by adding additional representation types. In approaching that larger problem, it became clear that too many unresolved issues still remained in the management of geometric representations alone. *QUICK* gives the foundation upon which management of hybrid representations may be possible. This would require the factor computations to be individualized to each representation type. Also, the issue of spatial interface between representations becomes much more vital in the hybrid case.

f. Computational Representations

Commodity hardware has recently moved transformation and lighting to the graphics hardware, removing any computational burden from the CPU when drawing polygonal representations. In contrast to this are those representations, such as fractals, progressive meshes, and subdivision surfaces, which require computation before transmission to the graphics pipeline. These formats, which here are termed computational representations, also offer continuous (or nearly continuous) display options. Additional representations usually increase the complexity of the optimization. However, a continuous range of options (or a representation with enough options to simulate continuity) reduces the guaranteed-correct optimization problem to tractability.

To support these computational representations, new Quality and Cost functions and annotations will be required. It is likely environments will combine these formats with standard polygonal representations. The naïve combination of the 0-1 and fractional knapsack problems is still NP-complete; some reformulation will be in order to benefit from the reduced complexity.

g. Object Elision

Two standard methods for reducing the set of visible objects are fog effects and the finite view frustum. Accordingly, experienced users of virtual environment systems are accustomed to the elision of far-field objects. In the *QUICK* system, however, any object can be omitted. From a resource conservation standpoint, object elision is appropriate whenever global Fidelity would be reduced by selecting any of that object's representations. As has been demonstrated in this dissertation, Fidelity is not always related to distance. The effect of this is that distant objects may be rendered and near-field objects removed.

The Fidelity/Cost ratio is usually highest for low resolution representations, so such elision is rare in practice. The option can be removed completely by severely reducing the Quality of the “empty” representation. Still, this near-field elision technique merits further investigation, likely in the form of user studies to determine the deleterious effects, if any, of its use.

h. Annotation Tools

The annotation process would be much more convenient if the appropriate analysis tools were included in modeling packages. While most of the annotation information is already available in such programs, output formatted for *QUICK* would be especially useful.

i. Optimized Cache Management

Display management and cache requests both take advantage of the *QUICK* algorithms. In the case of networked transfer, cache requests must be made predictively—otherwise the requested representation may no longer be pertinent by the time of its arrival. Such prediction can be accomplished easily by modifying Importance to reflect future values; however, this has been accomplished in only a rudimentary fashion thus far in the *QUICK* implementation. This could be improved easily by using current predictive fetching algorithms in Importance generation.

The other half of cache management, cache deletion, is currently performed with a standard Least Recently Used algorithm in the *QUICK* implementation. Depending on the algorithm used, the optimization process can yield a list of both the representations offering the most Fidelity and the representations offering the least Fidelity. Information of that type could be used to optimize clearing of cache memory.

2. Extensions for Networked Environments

A second set of extensions pertain specifically to improved integration with, or novel application to, networked virtual environments.

a. System Integration

A primary claim of this dissertation is that virtual environment traffic can be optimized by designing the client around an intelligent caching system. The initial implementation supports the theoretical grounds of that claim, but for true validation a full system design is needed. The Naval Postgraduate School's NPSNET-V [Capps *et al.*, 2000] is a Java-based virtual environment system which supports dynamic content and protocols. The architecture includes only rudimentary object request management, as it is intended that *QUICK* will serve that purpose. This will provide an excellent practical test of the framework's capabilities.

b. X3D Profile

With the lessons learned from the NPSNET-V integration, it will be possible to design an X3D profile to support *QUICK* annotations and processing. The componentized design of X3D encourages the incorporation of pervasive additions of this sort. The design of that profile will necessarily require an X3D-friendly XML specification of the *QUICK* annotations. Additional work is needed to ensure that this methodology is implemented in a manner compatible with other meta-data and annotation conventions, such as the forthcoming Resource Description Framework recommendation of the World Wide

Web Consortium.

c. *Intelligent Service*

Client-side optimization can improve transmission characteristics in a distributed virtual environment. However, modern large-scale virtual environments repeatedly find themselves constrained not by client bandwidth but by the capability of the server to process requests. Therefore, it seems logical for the serving process to optimize allocation of its resources amongst its multiple clients. This global optimization requires the client specification information from those clients; therefore a format and protocol for communicating up-to-date platform capability is required.

Server-side optimization does create new possibilities, such as factoring world state into the model service process. For instance, if a certain object is requested very frequently, or by nearly all users, the server can assume that delivering a representation for that node is especially useful for the user experience, and can adjust its Importance accordingly. Another example is that a server being used for a virtual chat area might temporarily increase the Importance of objects with avatars in close proximity, under the assumption that inhabited areas are more Important that uninhabited ones.

d. *Awareness Management*

QUICK is additionally applicable to filtering of inter-entity communications in a collaborative virtual environment (CVE). The *QUICK* system can integrate with, or even contain as a subset, algorithms for awareness management. While this capability is a primary motivation for the development of the *QUICK* system, this thesis specifically does not include proof of the applicability of *QUICK* to CVE communications.

For the purposes of this study, a CVE is defined as a shared environment in which many participants fetch models from servers and communicate special messages to each other. These messages can represent a variety of occurrences, such as avatar position changes, simple actions (such as a firing event in a military simulation), or complex actions (such as a introducing a new object and its behavior into the world). Central to making such systems scalable is managing the awareness each participant has of these messages. Broadcasting each message to all participants is convenient, but the bandwidth consumption required in large-scale systems makes it infeasible.

Computationally, selectively forwarding communications to participants is similar to the display serving problem. In this case, rather than having multiple representations of scene objects, there are multiple classes of service for entities acting in the virtual world. These classes of service for an avatar might be a combined position, velocity, angular velocity, and acceleration update thirty times a second—or just heartbeat messages sent every five seconds. When interacting closely with an avatar, the high update rate is needed, but such detail about an avatar a mile away in a fogged valley is not useful. And of course, similar to occluded areas in a model, no visual position updates are required for an avatar on the other side of an opaque wall.

The only new computation in this case is at the communications server. Given the communications capabilities and interests of its clients, and complete (highest class of service) information about entity actions, it determines what information is needed and how to forward it to the participants. The local display problem is unchanged; the only difference at the client is that object state may affect (or be affected by) interaction with the communications server. The model server also operates the same as before, either totally by request or with the traffic optimization discussed above.

(1) Quality. Defining Quality for classes of service for communications is an open and current area of research. Quality depends very closely upon the possibilities for informing a participant of an action, and upon the action itself. Some assumptions can be made, such as that Quality increases directly with update rate. Still, the strong analogy to Quality and Cost for rendering indicates caution before drawing general trends. It may be possible to develop some standard classes of service for common actions like physical motion; in general, however, Quality ratings will likely be task-specific.

(2) Importance. In the shared virtual environment case, the notion of Importance is similar to the Interest factor used in Awareness and Interest Management systems. Several different methods for determining and expressing interest have been incorporated into state of the art systems. A full review is available in Singhal and Zyda's *Networked Virtual Environments* text [Singhal and Zyda, 1999].

(3) Cost. The Cost of transmission for a class of service is the network capacity consumed per second. Network bandwidth is the primary resource limitation. The central processor resource is also consumed by processing many incoming messages, but CPU is rarely the bottleneck at the client.

D. SUMMARY

This chapter highlights the major contributions of this work: a definition of dynamic fidelity in distributed virtual environments, and a framework for maximizing fidelity through resource management. Significant opportunities for future work remain—both for the practical application of this optimization, and for the extension of its detail and scope.

APPENDIX A. ACRONYMS

2D/3D	Two-Dimensional / Three-Dimensional
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BSP	Binary Space Partition
CAD	Computer Aided Design
CPU	Central Processing Unit
CVE	Collaborative Virtual Environment
DIS	Distributed Interactive Simulation
DIV	Distributed OpenInventor
DIVE	Distributed Interactive Virtual Environment
DVE	Distributed Virtual Environment
FLOPS	Floating Point Operations
GMD	German National Research Center for Information Technology
GVS	Graduated Visibility Set
HLA	High-Level Architecture
HTTP	Hyper Text Transfer Protocol
KD	K-Dimensional [Tree]
LOD	Level of Detail
NP	Non-Polynomial
NPSNET	Naval Postgraduate School NETworked environment
QUICK	Quality, Importance, and Cost
PC	Personal Computer
PHS	Potentially Hearable Set
PVS	Potentially Visible Set
QTVR	QuickTime Virtual Reality
SGI	Silicon Graphics, Inc.
SPEC	Standard Performance Evaluation Corporation
SPLINE	Scalable Platform for Large Interactive Networked Environments
UML	Unified Modeling Language
UNC	University of North Carolina at Chapel Hill
URL	Uniform Resource Locator
VPE	Virtual Planetary Explorer
VR	Virtual Reality
VRML	Virtual Reality Modeling Language
WWW	World Wide Web
X3D	Extensible Three-Dimensional [Model, specification]
XML	Extensible Markup Language
ZOIP	Zero-One Integer Programming

Acronyms appropriate only within this dissertation:

BERK	Berkeley Walkthrough
J3DV	Java3D and VRML
QGRD	QUICK algorithm using greedy approximation
QMAX	QUICK algorithm using greedy approximation without replacement
QOPT	QUICK algorithm using linear optimization
PERF	Iris Performer
SPLN	SPLINE

APPENDIX B. EXAMPLE SCENES WITH ANNOTATIONS

This appendix contains a complete description of the 18-wheeler truck model used in many of the scenes in this dissertation. Many other example models, including all those used in test scenes, are available electronically as part of the *QUICK* software distribution.

The truck model file contains a single QSwitch that contains four level-of-detail nodes, with annotations. For visual clarity in demonstrations, the geometry is colored according to its detail. Colors are selected on a spectrum from green to red, with green for highest quality, yellow for median representations, and red for lowest quality. Figure 40 shows the four versions of the model side-by-side.

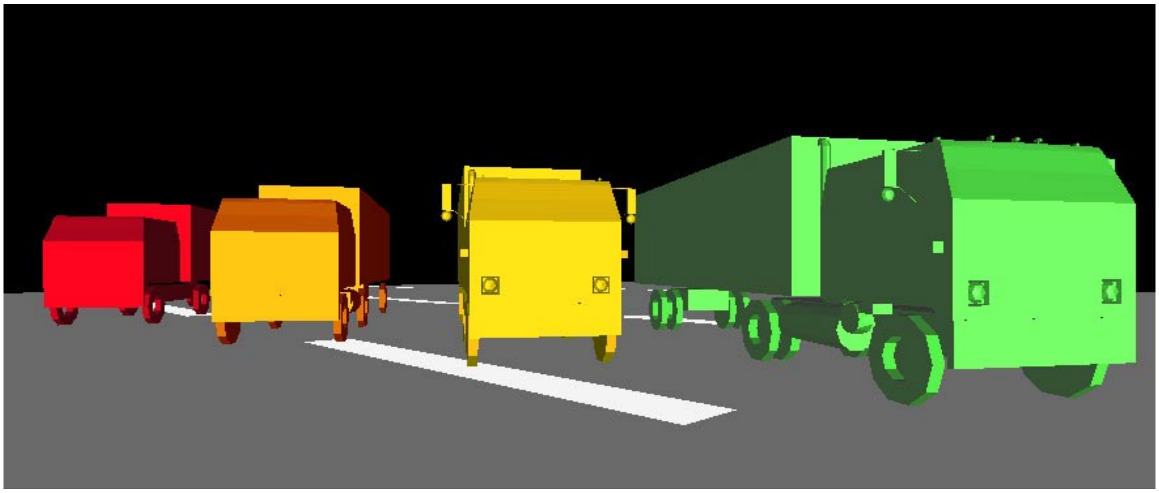


Figure 40. Truck Levels of Detail.

1. *QUICK* FORMAT

This model uses the non-standard *QUICK* extensions to VRML. This version was used with the initial *QUICK* implementation for convenience. The PROTO version, which follows, is generally preferred.

```
#QUICK V1.0 utf8
# contains a QSwitch that incorporates
# four LODs for an 18-wheel cargo truck

QSwitch {
  contents [
    "Vehicle:Ground:Truck:18_Wheeler"
  ]
  choice [
    QNode {
      quality QQuality {
        subjective 1     # author annotation
        colorDepth 4     # number of significant bits
        alphaDepth 1     # number of significant bits
        geomError 0      # error in meters
        geomErrorMax 0   # maximum error in meters
      }
      cost QCost {
        triangles 2360   # number of triangles
        filesize 133259  # ASCII uncompressed
      }
      url "18Wheel_1_2.4k.wrl"
    }
    QNode {
      quality  QQuality {
        subjective .95
        colorDepth 4
        alphaDepth 1
        geomError 0.05086  # missing wheels
        geomStdev 0.1036
      }
      cost QCost {
        triangles 1816
        filesize 118078
      }
      url "18Wheel_2_1.8k.wrl"
    }
    QNode {
      quality  QQuality {
        subjective .9
        colorDepth 4
        alphaDepth 1
        geomError 0.16949
        geomStdev 0.19098
      }
      cost QCost {
        triangles 1184
        filesize 58123
      }
      url "18Wheel_3_1.2k.wrl"
    }
    QNode {
      quality  QQuality {
        subjective .75
        colorDepth 4
        alphaDepth 1
        geomError 0.18882
        geomStdev 0.19628
     }
      cost QCost {
        triangles 603
        filesize 51582
      }
      url "18Wheel_4_0.6k.wrl"
    }
  ]
}
```

2. VRML97 *QUICK* PROTO DEFINITIONS

This section gives the VRML97 file which defines the PROTO nodes needed for *QUICK* .

```
PROTO QCost [
  exposedField   SFInt32     triangles        -1
  exposedField   SFInt32     flops            -1
  exposedField   SFInt32     filesize         -1
]
{
  WorldInfo {
    # There is no standard VRML scene node
    # analog for QCost, so a comment
    # node is added.
  }
}
PROTO QQuality [
  exposedField   SFFloat     geomError           -1.0
  exposedField   SFFloat     geomStdev           -1.0
  exposedField   SFInt32     colorDepth          -1
  exposedField   SFInt32     textureResolution   -1
  exposedField   SFInt32     alphaDepth          -1
  exposedField   SFFloat     subjective          -1.0
]
{
  WorldInfo {
    # There is no standard VRML scene node
    # analog for QQuality, so a comment
    # node is added.
  }
}
PROTO QNode [
  # fields for the VRML Transform node:
  field         SFVec3f     qbboxCenter       0.0 0.0 0.0
  field         SFVec3f     qbboxSize         -1.0 -1.0 -1.0
  exposedField SFVec3f     qtranslation      0.0 0.0 0.0
  exposedField SFRotation  qrotation         0.0 0.0 1.0 0.0
  exposedField SFVec3f     qscale            1.0 1.0 1.0
  exposedField SFRotation  qscaleOrientation 0.0 0.0 1.0 0.0
  exposedField SFVec3f     qcenter           0.0 0.0 0.0
  exposedField MFNode      qchildren         []

  # new fields:
  MFString    contents         []
  SFString    url              ""
  SFNode      cost             NULL  # a QCost node
  SFNode      quality          NULL  # a QQuality node
]
{
  Transform {
    bboxCenter IS qbboxCenter
    bboxSize IS qbboxSize
    translation IS qtranslation
    rotation IS qrotation
    scale IS qscale
    scaleOrientation IS qscaleOrientation
    center IS qcenter
    children IS qchildren
  }
} # end PROTO QNode
PROTO QSwitch [
  # fields from the VRML Switch node:
  exposedField SFInt32     whichChild      -1
  exposedField MFNode      children           []

  # new fields:
  exposedField SFFloat     importance       .5
  exposedField MFString    contents         []
]
{
  Switch {
    whichChoice IS whichChild
    choice IS children
  }
} # end PROTO QSwitch
```

3. EXTERNPROTO FORMAT

```
#VRML V2.0 utf8
# contains a QSwitch that incorporates
# four LODs for an 18-wheel cargo truck
# includes QUICK PROTOs using EXTERNPROTO mechanism
EXTERNPROTO QCost [
  exposedField   SFInt32     triangles
  exposedField   SFInt32     flops
  exposedField   SFInt32     filesize
] "http://.../quick.wrl#QCost"

EXTERNPROTO QQuality [
  exposedField   SFFloat     geomError
  exposedField   SFFloat     geomStdev
  exposedField   SFInt32     colorDepth
  exposedField   SFInt32     textureResolution
  exposedField   SFInt32     alphaDepth
  exposedField   SFFloat     subjective
] "http://.../quick.wrl#QQuality"

EXTERNPROTO QNode [
  field        SFVec3f     qbboxCenter
  field        SFVec3f     qbboxSize
  exposedField SFVec3f     qtranslation
  exposedField SFRotation  qrotation
  exposedField SFVec3f     qscale
  exposedField SFRotation  qscaleOrientation
  exposedField SFVec3f     qcenter
  exposedField MFNode      qchildren
  MFString    contents
  SFString    url
  SFNode      cost
  SFNode      quality
] "http://.../quick.wrl#QNode"

EXTERNPROTO QSwitch [
  exposedField SFInt32     whichChild
  exposedField MFNode      children
  exposedField SFFloat     importance
  exposedField MFString    contents
```

```
] "http://.../quick.wrl#QSwitch"

QSwitch {
  contents [
    "Vehicle:Ground:Truck:18_Wheeler"
  ]
  children [
    QNode {
      quality QQuality {
        subjective 1     # author annotation
        colorDepth 4     # number of significant bits
        alphaDepth 1     # number of significant bits
        geomError 0      # error in meters
        geomErrorMax 0   # maximum error in meters
      }
      cost QCost {
        triangles 2360   # number of triangles
        filesize 133259  # ASCII uncompressed
      }
      url "18Wheel_1_2.4k.wrl"
    }
    QNode {
      quality QQuality {
        subjective .95
        colorDepth 4
        alphaDepth 1
        geomError 0.05086  # missing wheels
        geomStdev 0.1036
      }
      cost QCost {
        triangles 1816
        filesize 118078
      }
      url "18Wheel_2_1.8k.wrl"
    }
    QNode {
      quality QQuality {
        subjective .9
        colorDepth 4
        alphaDepth 1
        geomError 0.16949
        geomStdev 0.19098
      }
      cost QCost {
        triangles 1184
        filesize 58123
      }
      url "18Wheel_3_1.2k.wrl"
    }
    QNode {
      quality QQuality {
        subjective .75
        colorDepth 4
        alphaDepth 1
        geomError 0.18882
        geomStdev 0.19628
     }
      cost QCost {
        triangles 603
        filesize 51582
      }
      url "18Wheel_4_0.6k.wrl"
    }
  ]
}
```

APPENDIX C. SOFTWARE AVAILABILITY AND DOCUMENTATION

All documentation for the *QUICK* software implementation is in a hypertext format, which does not lend itself to flat printing. Additionally, the software is projected to be under continuous development. Therefore, the material is included in this dissertation only by reference.

Readers interested in the *QUICK* software are encouraged to visit the following Internet address:

`http://npsnet.org/quick`

LIST OF REFERENCES

[Airey *et al.*, 1990] John M. Airey, John H. Rohlf, and Frederick P. Brooks, Jr. Towards image realism with interactive update rates in complex virtual building environments. volume 24, pages 41–50, March 1990.

[Anderson *et al.*, 1995] David Anderson, John Barrus, John Howard, Charles Rich, Chia Shen, and Richard Waters. Building multi-user interactive multimedia environments at merl. *IEEE Multimedia*, 1995.

[Appel, 1968] Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conf.*, volume 32, pages 37–45, 1968.

[Arvo and Kirk, 1990] James Arvo and David B. Kirk. Particle transport and image synthesis. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH ’90 Proceedings)*, volume 24, pages 63–66, August 1990.

[Baecker and Buxton, 1987] Ronald Baecker and William Buxton, editors. *Readings in Human-Computer Interaction: A Multidisciplinary Approach*. Morgan-Kaufmann, Los Altos, CA, 1987.

[Baecker *et al.*, 1995] Ronald Baecker, , Jonathan Grudin, William Buxton, and Saul Greenberg, editors. *Readings in Human-Computer Interaction: Towards the Year 2000*. Morgan-Kaufmann, Los Altos, CA, 1995.

[Banks and Weimer, 1992] William W. Banks and Jon Weimer. *Effective Computer Display Design*. Prentice Hall, 1992.

[Benford and Fahlen, 1993] Steve Benford and L. Fahlen. A spatial model of interaction in large virtual environments. In *Proceedings of ECSCW ’93*, September 1993.

[Bertsimas and Tsitsiklis, 1997] Dimitris Bertsimas and John Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, Belmont, MA, 1997.

[Birman *et al.*, 1985] Kenneth Birman, Thomas Joseph, T. Rauechle, and A. El Abbadi. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, SE-11:502–508, June 1985.

[Blanchard *et al.*, 1990] Charles Blanchard, S. Burgess, Young Harvill, Jaron Lanier, and A Lasko. Reality built for two: A virtual reality tool. In *Proceedings of the 1990 Symposium on Interactive 3D Graphics*, March 1990.

[Canny and Lin, 1993] John F. Canny and Ming C. Lin. An opportunistic global planner. *Algorithmica Special Issue on Computational Robotics*, 10(2-4):102–220, August 1993.

[Capps and Stotts, 1997] Michael Capps and David Stotts. Research issues in developing networked virtual realities. In *Proceedings of the Sixth Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 205–211, Cambridge, MA, June 1997.

[Capps and Teller, 1997] Michael Capps and Seth Teller. Communications visibility in shared virtual worlds. In *Proceedings of the Sixth Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 187–192, Cambridge, MA, June 1997.

[Capps *et al.*, 2000] Michael Capps, Don McGregor, Don Brutzman, and Michael Zyda. Npsnet-v: A new beginning for dynamically extensible virtual environments. *IEEE Computer Graphics and Applications*, 2000.

[Carlsson and Hagsand, 1993] C. Carlsson and Olaf Hagsand. Dive: A multi-user virtual reality system. In *Proceedings of the IEEE Virtual Reality Annual International Symposium*, pages 394–401, September 1993.

[Chamberlain *et al.*, 1996] Bradford Chamberlain, Tony DeRose, Dani Lischinski, David Salesin, and John Snyder. Fast rendering of complex environments using a spatial hierarchy. In Wayne A. Davis and Richard Bartels, editors, *Graphics Interface '96*, pages 132–141. Canadian Information Processing Society, Canadian Human-Computer Communications Society, May 1996. ISBN 0-9695338-5-3.

[Chen, 1995] Shenchang Eric Chen. Quicktime VR - an image-based approach to virtual environment navigation. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 29–38. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

[Clark, 1976] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, October 1976.

[Cohen *et al.*, 1996] Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks, and William Wright. Simplification envelopes. In *Proceedings of SIGGRAPH 96*, pages 119–128, New Orleans, LA, August 1996.

[Consortium, 1998] World Wide Web Consortium. Extensible markup language (xml) 1.0 recommendation, 1998.

[Coorg and Teller, 1996] Satyan Coorg and Seth Teller. Temporally coherent conservative visibility. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 78–87, 1996.

[Cormen *et al.*, 1990] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990.

[Danskin and Hanrahan, 1992] John Danskin and Pat Hanrahan. Fast algorithms for volume ray tracing. *1992 Workshop on Volume Visualization*, pages 91–98, 1992.

[Debevec *et al.*, 1996] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 11–20. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[DIS, 1993] IEEE standard for information technology–protocols for distribute dsimualtion applications: Entity information and interaction. IEEE Standard 1278-1993. New York: IEEE Computer Society, 1993.

[Drettakis and Sillion, 1996] George Drettakis and François Sillion. Accurate visibility and meshing calculations for hierarchical radiosity. In Xavier Pueyo and Peter Schröder, editors, *Eurographics Rendering Workshop 1996*, pages 269–278, New York City, NY, June 1996. Eurographics, Springer Wein. ISBN 3-211-82883-4.

[Durlach and Mavor, 1994] Nathaniel I. Durlach and Anne S. Mavor, editors. *Virtual Reality: Scientific and Technological Challenges*. National Academy Press, 1994.

[Falby *et al.*, 1993] John S. Falby, Michael J. Zyda, David R. Pratt, and Randy L. Mackey. NPSNET: Hierarchical data structures for real-time three-dimensional visual simulation. *Computers & Graphics*, 17(1):65–70, 1993.

[Farquhar *et al.*, 1995] Adam Farquhar, Richard Fikes, Wanda Pratt, and James Rice. Collaborative ontology construction for information integration. Technical report, Stanford University, 1995.

[Ferguson *et al.*, 1990] R. L. Ferguson, R. Economy, W. A. Kelley, and P. P. Ramos. Continuous terrain level of detail for visual simulation. In *Proceedings of the 1990 Image V Conference*, pages 145–151. Image Society, Tempe, AZ, June 1990.

[Foley *et al.*, 1990] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, 1990.

[Fowler *et al.*, 1999] Martin Fowler, Kendall Scott, and Grady Booch. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. The Addison-Wesley Object Technology Series. Addison-Wesley, 2nd edition edition, 1999.

[Fuchs *et al.*, 1980] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. volume 14, pages 124–133, July 1980.

[Funkhouser and Séquin, 1993] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 247–254, August 1993.

[Funkhouser *et al.*, 1992] Thomas A. Funkhouser, Carlo H. Sequin, and Seth J. Teller. Management of large amounts of data in interactive building walkthroughs. In David Zeltzer, editor, *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, volume 25, pages 11–20, March 1992.

[Funkhouser, 1993] Thomas Funkhouser. *Database and Display Algorithms for Interactive Visualization of Architectural Models*. PhD thesis, Computer Science Division (EECS), University of California, Berkeley, 1993.

[Funkhouser, 1996] Thomas A. Funkhouser. Database management for interactive display of large architectural models. In Wayne A. Davis and Richard Bartels, editors, *Graphics Interface '96*, pages 1–8. Canadian Information Processing Society, Canadian Human-Computer Communications Society, May 1996. ISBN 0-9695338-5-3.

[Garey and Johnson, 1979] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[Goerger, 1998] Simon Goerger. Spatial knowledge acquisition and transfer from virtual to natural environments for dismounted land navigation. Master's thesis, Naval Postgraduate School, Monterey, CA, 1998.

[Gossweiler, 1996] Richard Gossweiler. *Perception-Based Time Critical Rendering*. PhD thesis, University of Virginia, January 1996.

[Greenhalgh and Benford, 1995] Chris Greenhalgh and Steve Benford. Massive: a collaborative virtual environment for teleconferencing. *ACM transactions on CHI*, 2(3), September 1995.

[Hesina *et al.*, 1999] Gerd Hesina, Dieter Schmalstieg, Anton Fuhrmann, and Werner Purgathofer. Distributed open inventor: A practical approach to distributed 3d graphics. In *Proceedings of ACM VRST '99*, London, England, December 1999.

[Hitchner and McGreevy, 1993] Lewis Hitchner and Michael McGreevy. Methods for user-based reduction of model complexity for virtual planetary exploration. In *SPIE Vol. 1913*, pages 622–636, 1993.

[Hodges, 1992] Larry Hodges. Time-multiplexed stereoscopic computer graphics. *Computer Graphics and Applications*, 12:20–30, 1992.

[IdSoftware, 1996] IdSoftware. Quake software package, 1996.

[Keshav, 1997] S. Keshav. *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network*. Addison-Wesley, 1997.

[Kuhl *et al.*, 1999] Frederick Kuhl, Richard Weatherly, and Judith Dahmann. *Creating Computer Simulation Systems*. Prentice Hall, 1999.

[Lengyel and Snyder, 1997] Jed Lengyel and John Snyder. Rendering with coherent layers. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 233–242. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[Levoy, 1990] Marc Levoy. A hybrid ray tracer for rendering polygon and volume data. *IEEE Computer Graphics and Applications*, 10(2):33–40, March 1990.

[Lindstrom *et al.*, 1996] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hughes, Nick Faust, and Gregory Turner. Real-Time, continuous level of detail rendering of height fields. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 109–118. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[Luebke and Erikson, 1997] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 199–208. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[Luebke and Georges, 1995] David Luebke and Chris Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In Pat Hanrahan and Jim Winget, editors, *1995 Symposium on Interactive 3D Graphics*, pages 105–106. ACM SIGGRAPH, April 1995. ISBN 0-89791-736-7.

[MacDonald, 1999] Lindsay W. MacDonald. Using color effectively in computer graphics. *Computer Graphics and Applications*, 19(4):20–35, July/August 1999.

[Macedonia *et al.*, 1995] Michael Macedonia, Donald Brutzman, Michael Zyda, David Pratt, Paul Barham, John Falby, and John Locke. Npsnet: A multi-player 3d virtual environment over the internet. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 9–12, Monterey, California, April 1995.

[Magillo and Floriani, 1994] Paola Magillo and Leila De Floriani. Computing visibility maps on hierarchical terain maps. In *Proceedings of the 2nd ACM Workshop on Advances in Geographic Information Systems*, pages 8–15, Gaithersburg, Maryland, December 1994. ACM Press.

[Magillo and Floriani, 1995] Paolo Magillo and Leila De Floriani. Maintaining multiple levels of detail in the overlay of hierarchical subdivisions. Technical report, University of Genova, December 1995. hierarchical structure with two subdivisions overlayed, at diferring resolutions– good for GIS where overlapping two maps can happen.

[O'Connor *et al.*, 1996] John O'Connor, Barbara O'Kane, Christopher Royal, Kathy Ayscue, David Bonzo, and Beth Nystrom. Recognition of human activities using handheld thermal systems. Technical report, U.S. Army CECOM Research Night Vision and Electronic Sensors Directorate, Ft. Belvoir, VA, April 1996.

[Pesce, 1995] Mark Pesce. *VRML—Browsing and Building Cyberspace*. New Riders, Indianapolis, IN, 1995.

[PLI, 2000] Plib portable graphics library. http://plib.sourceforge.net, 2000.

[Rademacher, 1999] Paul Rademacher. View-dependent geometry. In *Proceedings of SIGGRAPH 99*, Los Angeles, CA, 1999.

[Rafferty *et al.*, 1998] Matthew Rafferty, Daniel Aliaga, and Anselmo Lastra. 3d image warping in architectural walkthroughs. In *Proceedings of VRAIS '98*, pages 228–233, Atlanta, GA, Month 1998.

[Reinhard *et al.*, 1996] Erik Reinhard, Arjan J. F. Kok, and Frederik W. Jansen. Cost prediction in ray tracing. In Xavier Pueyo and Peter Schröder, editors, *Eurographics Rendering Workshop 1996*, pages 41–50, New York City, NY, June 1996. Eurographics, Springer Wein. ISBN 3-211-82883-4.

[Rohlf and Helman, 1994] John Rohlf and James Helman. IRIS performer: A high performance multiprocessing toolkit for real–Time 3D graphics. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 381–395. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[Sandin *et al.*, 1997] D. Sandin, G. Olson, and Michael Macedonia. Panel: Distributed, interactive collaboration–where is it? In *Proceedings of 1997 Symposium on Interactive 3D Graphics*, Providence, RI, April 1997.

[Schmalstieg, 1997] Dieter Schmalstieg. Lodestar: An octree-based level of detail generator for VRML. In Rikk Carey and Paul Strauss, editors, *VRML 97: Second Symposium on the Virtual Reality Modeling Language*, New York City, NY, February 1997. ACM SIGGRAPH / ACM SIGCOMM, ACM Press. ISBN 0-89791-886-x.

[Shade *et al.*, 1996] Jonathan Shade, Dani Lischinski, David Salesin, Tony DeRose, and John Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 75–82. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[Shalabi, 1998] Sami Shalabi. High performance visualization of complex urban scenes. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1998.

[Silberschatz and Galvin, 1994] Abraham Silberschatz and Peter Galvin. *Operating System Concepts*. Addison Wesley, 1994.

[Sillion *et al.*, 1997] François Sillion, George Drettakis, and Benoit Bodelet. Efficient impostor manipulation for real-time visualization of urban scenery. In D. Fellner and L. Szirmay-Kalos, editors, *Computer Graphics Forum (Proc. of Eurographics '97)*, volume 16, pages 207–218, Budapest, Hungary, September 1997.

[Singhal and Zyda, 1999] Sandeep Singhal and Michael Zyda. *Networked Virtual Environments*. Addison-Wesley, 1999.

[Sipser, 1997] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, Boston, MA, 1997.

[Soto and Allongue, 1997] Michel Soto and Allongue. Semantic approach of virtual worlds interoperability. In Michael Capps, editor, *Proceedings of IEEE WET-ICE '97*, Cambridge, MA, June 1997. IEEE Press.

[Sowizral *et al.*, 1997] Henry Sowizral, Kevin Rushforth, and Michael Deering. *The Java 3D API Specification*. Java Series. Addison Wesley, December 1997.

[Speer *et al.*, 1985] L. R. Speer, T. D. Derose, and B. A. Barsky. A theoretical and empirical analysis of coherent ray-tracing. In M. Wein and E. M. Kidd, editors, *Graphics Interface '85 Proceedings*, pages 1–8. Canadian Inf. Process. Soc., 1985.

[Teller and Séquin, 1991] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 61–69, July 1991.

[Tramberend, 1999] Henrik Tramberend. Avocado: A distributed virtual reality framework. In *Proceedings of IEEE Virtual Reality '99*, pages 14–21, Houston, Texas, March 1999.

[Turk, 1991] Greg Turk. Generating textures for arbitrary surfaces using reaction-diffusion. In *Proceedings of SIGGRAPH 91*, pages 289–298, July 1991.

[VRM, 1997] The virtual reality modeling language. International Standard ISO/IEC 14772-1:1997, 1997.

[Waters *et al.*, 1997] Richard Waters, David Anderson, John Barrus, D. Brogan, M Casey, S McKeown, T Nitta, and William Yerazunis. Diamond park and spline: Social virtual reality with 3d animation, spoken interaction and runtime extendability. *Presence*, 6(4):461–481, 1997.

[X3D, 2000] X3d task group. http://www.web3d.org/, 2000.

[Yagel and Ray, 1996] R. Yagel and W. Ray. Visibility computation for efficient walkthrough of complex environments. *Presence*, 5(1):45–60, Winter 1996.

[Zeleznik *et al.*, 2000] Bob Zeleznik, Loring Holden, Michael Capps, Howard Zbrams, and Tim Miller. Scene-graph-as-bus: Collaboration between heterogeneous stand-alone 3-d graphical applications. In *Proceedings of Eurographics 2000*, Interlaken, Switzerland, August 2000.

[Zyda *et al.*, 1990] Michael J. Zyda, Mark A. Fichten, and David H. Jennings. Meaningful graphics workstation performance measurements. *Computers and Graphics*, 14(3/4):519–526, 1990.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Road., Ste 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. CAPT Steve Chapman, USN 1
   N6M
   2000 Navy Pentagon
   Room 4C445
   Washington, DC 20350-2000

4. George Phillips 1
   CNO, N6M1
   2000 Navy Pentagon
   Room 4C445
   Washington, DC 20350-2000

5. Assistant Professor Don Brutzman, Code UW/Br 1
   Undersea Warfare Academic Group
   Naval Postgraduate School
   Monterey, CA 93940

6. Research Assistant Professor Michael Capps, Code CS/Cm 5
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93940-5000

7. Assistant Professor Rudolph Darken, Code CS/Da 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93940-5000

8. Professor Ted Lewis, Code CS/Le 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93940-5000

9. Professor Michael Zyda, Code CS/Zk 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93940-5000

10. MOVES Research Center, Code CS/Fa 1
MOVES Academic Group
Naval Postgraduate School
Monterey, CA 93940-5000

11. Associate Professor Kevin Jeffay 1
UNC Computer Science
CB #3175, Sitterson Hall
Chapel Hill, NC 27599-3175

12. Brian C. Ladd 1
Mathematics Department
Valentine 118
St. Lawrence University
Canton, NY 13617

13. Justin Legakis 1
Laboratory for Computer Science
NE43-247
Massachusetts Institute of Technology
Cambridge, MA 02139

14. Henry Sowizral 1
Distinguished Engineer
Sun Microsystems, Inc.
901 San Antonio Road, MS MPK27-101
Palo Alto, CA 94303-4900

15. Associate Professor P. David Stotts 1
UNC Computer Science
CB #3175, Sitterson Hall
Chapel Hill, NC 27599-3175