Interactive Walkthrough Environments for Simulation

by

Richard William Bukowski

B.S. (Cornell University) 1992
M.S. (University of California, Berkeley) 1995

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor Of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Carlo H. Séquin , Chair
Professor John Canny
Professor Patrick Pagni

Fall 2001

The dissertation of Richard William Bukowski is approved:

Chair Date

Date

Date

University of California at Berkeley

Fall 2001

Interactive Walkthrough Environments for Simulation

Copyright 2001
by
Richard William Bukowski

Abstract

Interactive Walkthrough Environments for Simulation
by
Richard William Bukowski
Doctor Of Philosophy in Computer Science
University of California at Berkeley
Professor Carlo H. Séquin , Chair

This thesis describes a second-generation walkthrough framework that provides extensive facilities for integrating many types of third-party simulation codes into a large-scale virtual environment model, and puts it in perspective with first-generation systems built during the last two decades. The framework provides an advanced model database that supports multiple simultaneous users with full consistency semantics, system independent storage and retrieval, and efficient prefetching and object reconstruction techniques to support second and third-generation walkthrough systems. Furthermore, our framework integrates support for scalable, distributed, interactive models with plug-in physical simulation to provide a large and rich environment suitable for architectural evaluation and training applications.

A number of third-party simulations have been integrated into the framework, including dynamic physical interactions, fire simulation, multiple distributed users, radiosity, and online tapestry generation. All of these simulators interact with each other and with the user via a data distribution network that provides efficient, optimized use of bandwidth to transport simulation results to clients as they need them for visualization. These diverse simulators provide proof of concept for the generality of the framework, and show how quickly third-party simulations can be integrated into our system. The result is a highly interactive distributed architectural model with applications in research, training, and real-time data visualization.

Finally, an outlook is given to a possible third generation of virtual environment architectures that are capable of integrating different heterogeneous walkthrough models.

_____
Professor Carlo H. Séquin
Dissertation Committee Chair

Contents

List of Figures **v**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Driving Applications . . . 3
1.2.1 Fire Safety . . . 3
1.2.2 Applied City Models . . . 4
1.3 Technical Challenges . . . 4

2 Background and Related Work **6**
2.1 First Generation Systems . . . 7
2.1.1 Outdoor Environments . . . 7
2.1.2 Indoor Environments . . . 8
2.1.3 Simulation-Enhanced Environments . . . 9
2.1.4 Shortcomings of First Generation Systems . . . 11
2.2 Citywalk: A Second Generation Architecture . . . 12
2.3 Overview . . . 12

3 Database Support **14**
3.1 Design Goals . . . 14
3.1.1 Large Model Visualization . . . 15
3.1.2 Efficient On-Line Model Updates . . . 15
3.1.3 Multiple Interactive Agents . . . 16
3.2 Why Not Use An Off-The-Shelf Product? . . . 17
3.3 Specification . . . 18
3.3.1 API Overview . . . 18
3.3.2 Basic Design . . . 21
3.3.3 Object Service Layer . . . 25
3.3.4 Modification and Transaction Semantics . . . 31
3.3.5 Locking Semantics . . . 31
3.3.6 Watch Semantics . . . 32
3.4 Programming Concerns . . . 33
3.4.1 Ease of Extension . . . 33

3.4.2 Visualization During Database Mutation . . . 35
3.4.3 Effects of Dynamic Update on Frame Rate . . . 38
3.4.4 Updates and Viewing Processes . . . 39
3.4.5 Scalability . . . 40
3.5 Performance . . . 42

4 Simulation Data Management and Control 44
4.1 Motivation . . . 44
4.2 Assumptions and Summary of Approach . . . 46
4.2.1 Integration of Visualization and Simulation . . . 46
4.2.2 Walkthru as a Model Client Environment . . . 48
4.2.3 Simulation Types . . . 51
4.2.4 Simulator Output . . . 52
4.3 Communication and Control . . . 54
4.3.1 Primitive Channels . . . 54
4.3.2 The Simulation Manager . . . 55
4.3.3 Client to Simulator Communication . . . 56
4.3.4 Simulator to Client Communication . . . 58
4.4 Real-Time Data Management . . . 62
4.4.1 "Just-In-Time" Simulation Data Management . . . 62
4.4.2 Bandwidth Management . . . 64
4.4.3 Performance Analysis . . . 66
4.4.4 Results . . . 68
4.4.5 Conclusions and Observations . . . 73

5 Simulator Integration 75
5.1 Integration API . . . 75
5.1.1 Framework Modules . . . 75
5.1.2 Choosing an Interface . . . 76
5.1.3 Simulator Component . . . 78
5.1.4 Generic Interface Components . . . 80
5.1.5 Walkthru Interface Components . . . 81
5.2 CFAST (The Consolidated Model of Fire and Smoke Transport) . . . 82
5.2.1 Overview and Capabilities . . . 82
5.2.2 Database Integration . . . 85
5.2.3 Simulation Service . . . 88
5.2.4 User Interface Module . . . 89
5.2.5 Application . . . 91
5.3 IMPULSE (Impulse-based dynamics simulation) . . . 92
5.3.1 Overview and Capabilities . . . 92
5.3.2 Database Integration . . . 93
5.3.3 Simulation Service . . . 93
5.3.4 User Interface . . . 93
5.3.5 Results . . . 94
5.4 Real-time Multiuser Walkthru . . . 95

5.4.1 Overview and Capabilities . . . 95
5.4.2 Simulation Service . . . 95
5.4.3 User Interface . . . 96
5.4.4 Results . . . 97
5.5 Tapestries: On-line Imposter Generation . . . 97
5.5.1 Tapestry Construction . . . 98
5.5.2 Tapestries in the Framework . . . 99
5.6 Radiosity on Demand . . . 100
5.6.1 Overview . . . 100
5.6.2 Incremental Radiosity Updates . . . 101
5.6.3 View-Based Radiosity Updates . . . 102
5.6.4 Radiosity Updates in a Dynamic Model . . . 103
5.6.5 Radiosity Results . . . 104
5.7 The Generic Metasimulator . . . 106
5.8 Overall Integration Experiences . . . 107

6 Model Construction with Floorsketch **108**
6.1 Motivation . . . 108
6.2 Basic Modeling with Floorsketch . . . 110
6.3 Extrusion . . . 113
6.4 Advanced Applications . . . 117
6.5 Results . . . 118

7 Discussion **120**
7.1 Architectural Analysis . . . 120
7.2 Relationship to Existing Techniques . . . 123
7.2.1 Database Techniques . . . 123
7.2.2 Communication and Interaction Techniques . . . 124
7.2.3 Distributed Simulation Techniques . . . 124
7.3 New Directions . . . 124
7.3.1 Simulation Triggering . . . 124
7.3.2 Temporal Navigation and Representation . . . 126
7.3.3 Integrated Rendering Frameworks . . . 126
7.4 Summary . . . 128
7.5 A Final Thought . . . 128

Bibliography **129**

A Library API Reference **136**
A.1 System Library (**system**) . . . 136
A.1.1 Compatibility Functions (*system.h, compat.[h,c]*) . . . 136
A.1.2 Error Reporting (*errors.[h,c]*) . . . 136
A.2 Core Class Library (**gsim**) . . . 137
A.2.1 Socket Abstraction (*rtsocket.[hpp, cpp]*) . . . 138
A.2.2 Channel Abstraction (*rtchannel.[hpp, cpp]*) . . . 138

A.2.3 Timer Facilities (*rttimer.[hpp,cpp]*) . . . 139
A.2.4 Threading Facilities (*rtthread.[hpp,cpp]*) . . . 139
A.2.5 Universal Base Class (*wkobject.[hpp,cpp]*) . . . 139
A.2.6 Packable Interface and Smart Buffers (*rtbuffer.[hpp,cpp]*) . . . 140
A.3 Database Library (**ndf**) . . . 140
A.3.1 Block Services (*dfblockservice.[hpp,cpp], dfblockserviceex.[hpp,cpp]*) . . . 141
A.3.2 Blob Services (*dfblobservice.[hpp,cpp]*) . . . 142
A.3.3 Universal Blob Server (*dfservermain.[hpp,cpp]*) . . . 143
A.3.4 Transaction Stack Object (*dftransaction.[hpp,cpp]*) . . . 143
A.3.5 Persistent Object Base Class (*dfobject.[hpp,cpp]*) . . . 144
A.3.6 Database Shell Object (*dfdatabase.[hpp,cpp]*) . . . 144
A.3.7 Smart Pointers (*dfsmartptr.[hpp,cpp]*) . . . 145
A.4 Simulator Library (**newsim**) . . . 146
A.4.1 General Architecture . . . 146
A.4.2 Summary of Optional Overloadable Classes . . . 150
A.4.3 Simulation Manager (*rtsimulationmanager.[hpp,cpp]*) . . . 150
A.4.4 Bandwidth Manager (*rtbwmanager.[hpp,cpp]*) . . . 152
A.4.5 Condition Chunk Base Class (*rtconditions.[hpp,cpp]*) . . . 153
A.4.6 Simulation Service Base Class (*rtsimulationservice.[hpp,cpp]*) . . . 153
A.4.7 Simulation Client Base Class (*rtsimulationclient.[hpp,cpp]*) . . . 153
A.4.8 Local Simulation Service Base Class (*rtlocalsimservice.[hpp,cpp]*) . . . 153
A.4.9 Remote Simulation Service Base Class (*rtremotesimservice.[hpp,cpp]*) . . . 154
A.4.10 Real-Time Simulation Components (*rtrtcomponents.[hpp,cpp]*) . . . 154
A.4.11 Real-Time Simulation Client Base Class (*rtrtsimulationclient.[hpp,cpp]*) . . . 155
A.4.12 Virtual-Time Simulation Components (*rtvtcomponents.[hpp,cpp]*) . . . 156
A.4.13 Virtual-Time Simulation Client Base Class (*rtvtsimulationclient.[hpp,cpp]*) 157
A.4.14 Simulation View Base Classes (*[rtsimview,rtbasicsimview].[hpp,cpp]*) . . . 157

List of Figures

2.1 *Snapshots from the Virtual Los Angeles Project.* 8

3.1 *First and second generation Berkeley Walkthru databases. The first generation concentrated on a homogeneous interior environment; the second generation scales up to a heterogeneous, indoor and outdoor environment.* 14
3.2 *Multiple different types of agents active within the same environment; multiple users, dynamics, and fire.* 17
3.3 *Interrelated constellations describing an object in the Citywalk database. Constellation boundaries are dashed; the master object for each constellation has a thick border.* 19
3.4 *Multiple processes connecting to a database. The blob service is replicated through a socket connection; local database objects provide namespaces in which each database object can be in an independent load and lock state.* 22
3.5 *A typical watch firing sequence. When a transaction commits, some objects are modified; these trigger watches set on that object by other processes, which are notified via a message in an asynchronous thread. How they respond to the watch is defined by the individual process. Note that neither the time to receipt nor the order of the notifications is guaranteed; processes must anticipate this and be ready to operate on incomplete information in time-critical situations.* 23
3.6 *Different referential spaces for different processes. Particular portions of the model may or may not be loaded in each space, or may be older versions awaiting update.* 25
3.7 *Assignment and identity of objects. When an object is assigned, it takes on a universal identity that it shares in all referential spaces. On assignment, the object acquires a DbRef value that identifies it to all processes (e.g. the* **Ref** *field). The first assignment also establishes version 0 of the object in the persistent store.* . . . 28
3.8 *Serialization of objects converts them to a machine independent form that can either be stored in the database or transferred to other processes via sockets. Note that references external to the constellation are stored in* **Ref** *form.* 30
3.9 *Dynamic update taking place over multiple frames. In intermediate frames, the system may render outdated information in exchange for reducing the visual impact of frame rate discontinuities.* 39

3.10 *Models can be split over multiple servers, providing scalability for viewers that can incrementally load local regions of the database. References to objects in other databases can be embedded by combining a database location with a* **Ref** *within that database.* 41

4.1 *Our model of the machines comprising the data network. Note that we assume there is only one limited-bandwidth physical link between machines, particularly to client machines, and that data and simulation servers can reside anywhere.* 47
4.2 *The output of many types of simulation can be grouped by the same volumes that partition the model for visibility. If this is the case, the viewer only needs to receive simulation data for the volumes they can see.* 49
4.3 *The just-in-time concept provides minimal latency by sending only just as much information at each timestep as the client can receive. Traditional buffering can result in large backups of data that result in high latency if the viewer suddenly needs different information while the buffers are still packed.* 50
4.4 *Dynamic updates vs. persistent updates. Dotted borders represent machine boundaries; arrows between them require network communication. Dynamic updates are much faster, due to data sharing and minimal network communications; Persistent updates are easier to closely synchronize, via database locking and transactions, and do not require simulation-specific connections.* 53
4.5 *Telemetry objects provide the simulation server with knowledge of what interest regions the client is exploring.* 58
4.6 *Intermediate nodes can merge telemetry nodes to improve bandwidth usage within the service network.* 59
4.7 *An example of the set of chunks generated over time in the dynamics simulator. The chunks generated for a particular object, which can be located in the condition set by the objects' unique ID in the subvolume field of the chunk, can show changing internal data over time, and change volume as the object moves between cells.* 61
4.8 *Chunk importance is based on proximity to the user's immediate interest and near-future lookahead interest sets.* 64
4.9 *Trace data of simulator-visualizer data transfer for three strategies: the oldest-data-first strategy (top), the visibility-guided strategy (middle), and the bandwidth-managed-importance strategy (bottom). The horizontal axis is real time; the vertical axis is simulation (i.e. virtual) time. Three functions are plotted for each strategy: the amount of simulation time completed by the simulator, the viewer's current visualization time, and the timestamp of the latest chunk that has been transmitted from simulator to visualizer. Note the vertical lines in the requested visualization time, which denote user-created time discontinuities, and the horizontal lines in the requested visualization time, which show regimes for which data is available from the simulator, but for which that data had not been transmitted in time to be viewed. The "maximum simulation time transmitted" curves give an indication of how responsive each strategy is to user movement in space and time.* 69

4.10 *Trace data of communication pipe backup (i.e. clogging) for the oldest-data-first and visibility-guided strategies. The former is much worse than the latter, although it is in the latter that it actually makes a difference. Pipe blockage in the bandwidth-managed-importance case is negligible (less than 1.5 kB/s on this graph, where the others peak at about 550 kB/s and 250 kB/s respectively), and can in fact be reduced to an arbitrarily small amount on a fast computer by increasing the manager's call-back frequency.* 70
4.11 *Trace data of the percentage of spacetime volumes visible to the user that have simulation data available, but for which that data has not yet been transmitted to the visualizer. The oldest-data-first strategy exhibits massive gaps in viewable data; the visibility-guided strategy fares better, but there is still a 40-second period where the user should be seeing smoke and flame, but instead sees nothing. The bandwidth-managed-importance case shows only brief 1- to 2-second gaps at time discontinuities (i.e. where the user unpredictably drags the time slider far into the untransmitted data).* 71

5.1 *A diagram of how the system components connect simulator to visualizer. Components in bold outline are created by the user; components in dotted outline are provided by the integration framework.* 76
5.2 *VCR controls that control the flow of "virtual" time.* 81
5.3 *The zone model finite element method used for fire simulation. Each room maintains two zones, with up to three qualitatively discrete exchange regions between volumes through doorways or windows.* 84
5.4 *CFAST's original input and output, as it is distributed by NIST. These forms are difficult for an untrained user to create and understand.* 85
5.5 *The plugin UI for setting CFAST's chemical properties.* 86
5.6 *How CFAST (bottom) maps volumes onto the world cell structure (top). The Walkthru model contains detailed geometric information, but little else; the CFAST model is geometrically much simpler, but contains chemical and materials information that Walkthru lacks.* 87
5.7 *CFAST view modes. Left, realistic mode; right, thermal mode.* 90
5.8 *IMPULSE simulating bears and balls bouncing in a laboratory.* 94
5.9 *Left, multiuser chat window. Right, avatars interact with each other and the doorways in the MIT LCS model.* 96
5.10 View location and projection surface for portal and cell tapestry construction. . . . 100
5.11 An example interaction between a user, the radiosity agent, and the tapestry agent. . 105

6.1 *The Building Model Generator (BMG) converts modified CAD floorplans into 3D Citywalk models, but not without substantial help from the user.* 109
6.2 *A basic floorplan in Floorsketch, and its components.* 111
6.3 *Portals move with the room they are in; this makes adding rooms to central hallways easier, and prevents the user from having to track two-sided entities if the floor layout is modified.* 111
6.4 *The three flavors of portal ("Door", "Window", and "Full Wall"), and how they extrude into 3D from different 2D configurations.* 112

6.5 *Rooms in Floorsketch, populated with tokens representing furniture and view frustums.* 113
6.6 *Using a JPEG image in the background, the user can more easily "trace" an existing floorplan into a 3D model.* 114
6.7 *Example of portals that lead to other volumes vs. portals that lead to the outside.* . 115
6.8 *The 12-story MIT Laboratory for Computer Science (LCS), modeled in Floorsketch from JPEG images of its floorplans in less than 1 day. Left, the stacked floors extruded from Floorsketch. Right, the exteriors of the buildings in Tech square (LCS building is circled). The interior fits inside the exterior shell to within 6 inches on all sides.* 119

A.1 *The object configuration resulting from a request to create and connect to a service on the local machine.* 148
A.2 *The object configuration resulting from a request to create and connect to a service on a remote machine.* 149
A.3 *Classes and inheritance patterns in the* **newsim** *library.* 151

Acknowledgements

This work was done under the guidance of Professor Carlo Séquin , who has been an advisor, mentor, and friend for the last 9 years. He has the patience of a saint, and his unique combination of enthusiasm for tackling new challenges and down-to-earth sensibility about what is important in life and research have been inspirational. I consider myself extremely fortunate to have been one of his students; the experience has been truly remarkable. Thanks also to his wife Greti for many wonderful evenings having fondue with friends.

I am also grateful to the entire Berkeley Walkthru research group, who, over the years, have helped make my experience here as wonderful as it has been. Seth Teller and Tom Funkhouser started the whole thing, and have never hesitated to help me understand aspects of the system. Seth in particular has gone above and beyond the call of duty and I salute him as a colleague and friend. More recently, Maryann Simmons, Laura Downs, and Mike Wittman have contributed invaluable insights and hard coding time to the Citywalk effort.

I received extensive financial support for this work from the National Institute of Standards and Technology, in particular Walter Jones of the Building and Fire Research Lab there. Their funding has enabled the bulk of this thesis, and I thank them heartily for it.

Last but certainly not least, thanks to my family, particularly my parents, Richard and Maria Bukowski; my grandparents, Guy and Virginia Agostino and Richard and Helen Bukowski; and my wife, Laura Downs, for all their support. I could not have come this far without them.

Chapter 1

Introduction

1.1 Motivation

Until a few years ago, virtual environments (VEs) were used primarily for visualization tasks. Recently, VE technology has begun to propagate into design tasks (such as AutoCAD's 3D visualization of a proposed construction, or the many "home improvement" programs you see on the shelves of your local software stores, which do "still-frame" walkthroughs of small, 1 or 2 room additions to houses, remodeled kitchens, or the like). As the technology grows in capability and our computers grow in processing power, these systems will also become useful as integrated simulation and design evaluation environments. In the future, we see a second generation of VE systems being used to train firefighters in how to combat raging building fires without ever setting foot near a flame [1], or to determine how changes in lighting, airflow, or noise will affect the comfort levels of occupants of a building [2, 3, 4].

One application domain with a particularly high expected payoff is building design evaluation, where scientists, engineers, architects, and other professionals can enter a virtual space and evaluate many of its physical properties with no danger, minimal cost, and a small investment of time. With such a system, users can preview architectural designs, evaluate their performance under various metrics, and do potentially destructive "what-if" experiments without cost or risk. To obtain useful answers to such experiments, we need to integrate good physical simulations with virtual environment interfaces. Integration of powerful simulation technology with virtual reality visualization systems affords the possibility of intuitive interpretation and visualization of the results of complex and powerful simulations via 3D computer graphics.

While the combination of virtual reality and environmental simulation constitutes a frame-

work for very powerful tools, it also raises many implementation challenges. The interactivity of the environment must be improved to allow users to affect simulations in progress by performing relevant actions, like opening doors and windows or moving objects. These dynamically changing conditions must be efficiently propagated from the client to a simulator, and the results must be transported back to the viewer efficiently. The system must integrate the simulator's results with the virtual environment, and display those results in a useful way; either symbolically, in the case of scientific visualization applications, or photorealistically, in the case of training or entertainment applications. These problems are compounded by the need to distribute both the virtual environment and the simulation over multiple computers – potentially connected by relatively high-latency, low-bandwidth networks such as the Internet – when attempting to simulate and visualize large buildings with hundreds of rooms.

Another major hurdle is the difficulty of walkthrough model construction. Building an accurate architectural walkthrough model is a challenging task. Typically, it requires the user to either model the environment with complex 3D modeling software, or acquire CAD architectural plans and use semi-automated tools to extract building structure. Proper use of these tools requires intimate knowledge of the 3D software in question (CAD or another modeling system) and the nature of the 3D visualization database toolkit. Furniture and details must also be added, often without the benefit of any sort of interactive software; for example, the first-generation Berkeley architectural walkthrough (Walkthru) system requires users to enter instance statements into a text file and use a UNIX batch-mode build process [5]. "Naive" users (e.g. those who wish to use a VE system without acquiring a degree in computer science) have a very hard time using these construction methods; this strongly limits the utility of the technology outside the academic world. Hence, one of our goals is to simplify this process to get users quickly building models that conform to the needs of both the simulation engine and our second-generation walkthrough viewer.

Finally, we would also like this framework to be flexible enough to allow programmers to rapidly integrate additional third-party simulations into the environment, such as physical simulations, acoustics simulations, or lighting simulations. This allows us to leverage the work of other research groups, add a larger spectrum of interactions to the world, and offer the technology to other classes of user, thus making the walkthrough environment more interesting and more useful.

1.2 Driving Applications

1.2.1 Fire Safety

The primary driving application behind this work has been building fire safety. The fire protection community in both the United States and the world at large is moving towards performance-based standards from the current prescription-based standards. This creates a need to use simulations such as the National Institute of Standards and Technology Consolidated Model of Fire and Smoke Transport (NIST CFAST [6]) fire simulator to evaluate performance in the design and review phase. However, these simulation programs are typically written with rudimentary input and output mechanisms, and are hard to learn and use. We believe that applying a tool based on virtual environment technology like the Berkeley Walkthru can permit us to streamline the input techniques, process the output into more useful and intuitive forms, and improve the scope and ease of interaction with the system.

Thus, CFAST was the first third-party simulation we integrated into our second-generation Berkeley Citywalk system. CFAST provides an accurate simulation of the impact of fire and its byproducts on a building environment. Integrated into Citywalk, it provides real-time, intuitive, realistic and scientific visualization of building conditions in a fire hazard situation from the perspective of a person walking through a burning building. The viewer can observe the natural visual effects of flame and smoke in fire hazard conditions. Alternatively, scientific visualization techniques allow the user to ”observe” the concentrations of toxic compounds such as carbon monoxide and hydrogen cyanide in the air, as well as the temperatures of the atmosphere, walls, and floor. Warning and suppression systems such as smoke detectors and sprinkler heads can be observed in action to help determine their effectiveness. This technology can be used to improve fire safety by helping engineers and architects evaluate a building's potential safety and survivability through performance-based standards (i.e. how well the building protects its occupants from the fire). With more development, it could also be used to help train personnel in firefighting techniques and rescue operations by presenting them with practice situations that are too risky to be simulated in the real world. Another very interesting safety application is predicting and assisting human behavioral responses under emergency conditions; these simulations could help predict response patterns of building occupants in emergencies, or, with realistic visualization tachniques, help teach residents where to go and what conditions to expect during an emergency building evacuation (e.g. “virtual fire drills”).

1.2.2 Applied City Models

The ability to explore an entire campus, or a whole city, rather than a single building is another key application driving this work. This requires a 2 to 3 order of magnitude increase in database size. The state of the art in city model construction is on the verge of taking a leap, with the advent of 3D laser scanners and automated reconstruction technologies [7, 8, 9]. These applications need a system framework that can combine the techniques used in interior architectural walkthroughs, such as cell and portal culling, with techniques used for exterior scenes, such as impostors and horizon culling, into a coherent whole. These city-scale models can then be used for new, large-scale training and maintainence applications, including military ground training, integrated maintainence models for campuses and cities, and entertainment applications. They also hold the promise of combining with wide-area wireless networks and "smart building" technology to advance the state of the art in public works, safety, and maintainence. For example, a fire starts in a building. The building systems send information about which sprinklers are active and which detectors are seeing toxic gases via a broadband network to a fire station, which has access to a networked full virtual model of the city and buildings. The model can help the crew find the way to the building, communicate situational information to the chief, while she is on the fire truck on the way to the scene. Meanwhile, integrated simulation can predict the likely future conditions to help plan where the firefighters should stage their assault on the flames to maximize both safety and firefighting effectiveness. We would like to construct a system that could support such a scenario given the emerging technologies that are coming into widespread use today.

1.3 Technical Challenges

This thesis focuses on providing a support framework that is the basis for an second-generation walkthrough system providing integration of the universe of traditional, first-generation virtual environment visualization and data management techniques, and support for rapidly integrating diverse third-party simulation engines into a unified and highly distributed virtual environment that provides for efficient, emergent behavior arising from the interaction of multiple human and simulation agents with each other and the world model. The result is a two-tiered framework combining a robust, distributed persistent database substrate emphasizing availability and a general-purpose, extensible structure, with a high-performance real-time data distribution layer that provides intelligent routing of data between interactive agents to cope with performance issues on modern

wide-area networks in the face of time-critical simulation appilcations.

Chapter 2

Background and Related Work

Over the last few decades, the evolution of powerful personal workstations equipped with advanced graphics hardware has given us low-cost systems on which fairly complex virtual worlds can be explored at interactive speeds. This capability is used for both professional design applications as well as entertainment.

The earliest such "walkthrough" systems date back to the 1970's and evolved with the goal of providing real-time flight simulators [10, 11]. Training in a virtual environment helps pilots gain skill without the physical danger and cost of actual training flights. In the late 1980's, the research community began to publish work that described and analyzed techniques by which near real-time performance could be achieved on relatively rich and complex models [12, 13, 14]. These research systems typically focused on a single issue for which they advanced the state of the art. Some systems introduced new model abstractions and culling mechanisms to improve rendering performance [12, 14, 15]. Others focused on incorporating many users or providing a high level of interactivity with the world [16, 17, 18]. A few special-purpose systems allowed the real-time exploration of specific very large and complex environments [19, 15, 20].

These early systems were struggling to address basic problems in virtual environment visualization, and they were often pushing the bounds of contemporary hardware or algorithms to achieve their goals. We refer to these systems as *first generation*. In the last few years, advances in performance and reductions in cost are finally providing enough system resources to generalize and merge these varied techniques into a coherent whole. This creates a new set of structural and theoretical problems, as these disparate techniques all have their own requirements and may interact with each other in complex ways. The promise of these *second generation* systems is to provide greatly enriched interactivity and realism with large numbers of users on very large, distributed

models.

2.1 First Generation Systems

Existing walkthrough systems can be roughly split into two general types: indoor and outdoor environments. While their application domains differ, all of these systems achieve scalable performance by partitioning the model such that only a relatively small portion of the database needs to be resident in memory at one time (visibility culling) and by applying level of detail (LOD) abstractions to those elements that are visible to reduce rendering time.

2.1.1 Outdoor Environments

The first outdoor virtual world applications were flight simulators. Their development resulted in pioneering work in levels of detail and abstraction that allowed a large, complex landscape to be loaded and simplified in real-time such that it was renderable on contemporary graphics hardware. The large scale of the environment in these systems was handled by combining locality-based databases (i.e. the world is "tiled", and users see only the tile they are over, plus the adjacent tiles, at any one time [21]) with level of detail techniques to simplify distant geometry and objects. The simplicity of the environment (typically a textured terrain map with a few detail objects) and the relatively slow speed of the user (it takes a long time to cross a tile) makes it straightforward to load terrain tiles as the user approaches them.

Recently, virtual city walkthrough projects such as the Virtual Los Angeles Project [22] explored more advanced database techniques for streaming large models to clients, as well as new simplification techniques such as impostors [23]. These systems offer much more complex world models than flight simulators; the virtual Los Angeles model, for example, contains over a terabyte of data (Figure 2.1). This complexity is needed because the user is in close proximity to the buildings, so the geometry must be much more detailed than in a flight simulator, which can get away with simply texturing the distant ground tiles. The Los Angeles system pioneered the use of a streaming media database, which enables them to stream small sections of the model for the area around the user's current location with extremely low latency. Simplification techniques such as impostors allow the database to use less disk or network bandwidth for the same level of visual detail in these large outdoor models.

Another approach to outdoor environments is simulators such as NPSNet [24], which

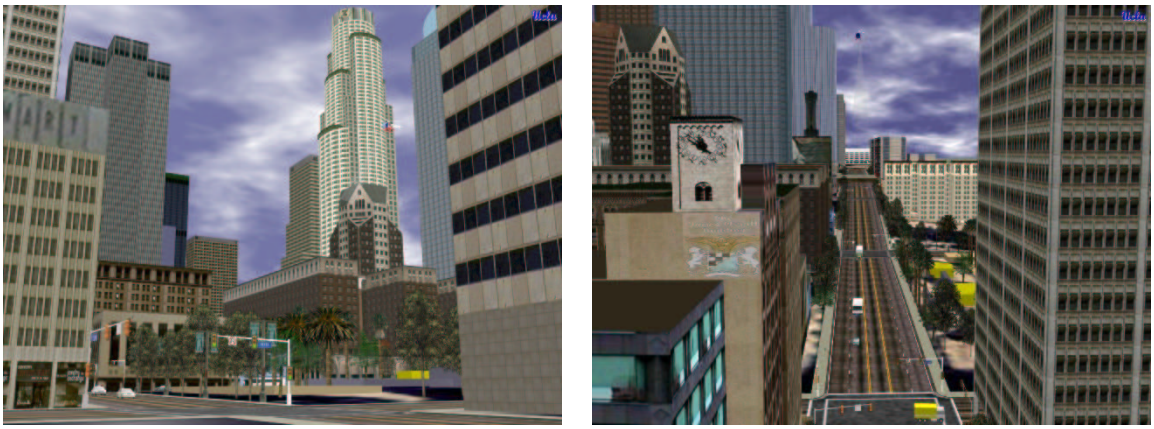


Figure 2.1: *Snapshots from the Virtual Los Angeles Project.*

focus on a large number of users in the world with high interactivity. These high-performance interactive systems provide a high-speed communication layer using IP multi-cast to provide rapid distribution of the state of users and other entities within localized cells [25]. This allows very low latency (under 100 ms) and provides realistic interactions at the speed of combat for hundreds of users at once [26]. Model scaling is done via "zones", which are terrain tiles much like those used for flight simulators, and within which the multi-cast is performed; this serves to limit the multi-cast bandwidth to a reasonable level, since clients only multi-cast to other clients that are "nearby". Recent commercial multi-player on-line computer games involving thousands of simultaneous users, such as Ultima Online or Everquest, attempt similar interactions. Due to hardware and bandwidth constraints, they are even more strongly partitioned into localized zones. These systems cannot benefit strongly from multi-cast techniques because their users are often widely distributed and use low-speed modems; they sacrifice range of visibility (often only a few dozen feet) and interaction latency (which can slow to seconds if the scene gets crowded) to achieve "reasonable" interactivity at extremely low cost compared to the military systems.

2.1.2 Indoor Environments

Indoor environments, which are primarily found in architectural walkthroughs, are treated separately because they have a densely occluded structure that lends itself to various forms of strong portal culling [27, 14]. These environments pose conceptually similar problems in database management and rendering complexity, but they rely less on level of detail and instead on a more powerful set of culling techniques that can take advantage of the densely occluded nature of the models.

Systems such as the Berkeley Walkthru program [5] and the several University of North Carolina architectural walkthrough systems [20] have proven that systems using on-disk or on-network object databases, combined with integrated level of detail abstractions, prefetching, and user motion prediction, can provide interactive (10 frame per second or better) visualization of very large, complex architectural databases, just as the virtual city projects provide fast visualization of huge outdoor databases.

Funkhouser's RING system [28] provides distributed multi-user functionality in the indoor domain, using a system of central servers with both high-speed interconnections and higher level geometric information about world structure. This allows the system to distribute the same information more intelligently, and limit the amount of data that is transferred to individual client machines based on client regions of interest. The approach also improves on the multi-cast techniques in that it works better for clients with slow and/or nonlocal network links, which is a problem for the IP multi-cast systems used in outdoor databases.

2.1.3 Simulation-Enhanced Environments

The most frequent application of virtual reality technology so far has been visualization of static spatial environments. Even where they offer simulated or interactive agents, current virtual worlds are typically nearly-static environments with a few movable objects and avatars inside. The most common applications of these systems are either peer-to-peer simulation of the user's interaction with other users or simulated entities, or systems that use physics to make the world seem more "real" to an immersed user. Some more famous examples of the former include the Iowa driving simulator [29], where the user's vehicle interacts with other independently-simulated road vehicles, and the Department of Defense's NPSNET [17, 13], where "units" of military vehicles engage in simulated combat on static terrain. Each simulated unit (or vehicle) communicates its status to all other units. Since the environment (i.e. the terrain) is fixed, the communication requirements are bounded by the number of simulation entities, not the size of the environment. Though these systems may be doing some actual physical simulations, because only a few "detail objects" in the world are actually changing, the amount of data being transferred is relatively small. Other systems are typically concerned with the physics of everyday object interaction, such as impenetrability and collisions [30, 31, 32]. They have been used to evaluate the ergonomics of environments like kitchens, automobiles, or work spaces. In these systems, simulations are typically limited to objects being directly manipulated, and the computations are simplified, so that they can be done directly

in the visualization environment without seriously loading down the computer.

On the other hand, many virtual-reality visualization systems have been built to allow the user to perform and interact with complex physical simulations, but they tend not to involve what we would consider "interactive simulation;" that is, the user is simply exploring precomputed data, without being able to interactively change the conditions under which that data was derived. NASA's virtual windtunnel [33], in which airflow around a particular object is calculated, is a well documented example of this approach. An observer can enter a "black void" in which the object is suspended, insert "ink" sources to produce streamers along flow lines, and view the airflow computations from within the air space around the object. This system visualizes a precomputed computational fluid dynamics solution, and only allows the user to explore the space of the computed solution, without the ability to interactively modify the object or wind conditions for which the solution had been generated.

The architectural community is very interested in full-scale interactive environmental simulation of planned environments from the point of view of an immersed human observer. Parameters of interest include lighting, temperature, and airflow throughout an entire building, and the computations can become very complex. Some architectural firms have constructed non-interactive, predefined video-tape visualizations comprising many moving people [3]. Realistic world simulation, where the environment itself is changing based on a reasonable subset of physical and chemical laws, and under the possible influence of user-initiated changes to the scenario set-up, is a much more difficult task. Combining such simulations with immersive visualization by one or more active observers adds particular challenges with respect to synchronization and data management.

For systems that do offer interactive, real-time scientific visualization of complex simulations, the data transmission problem is well documented [34, 35, 27]. As the simulated system grows more complex, the amount of data needed to describe the full simulation state of the system in each time step can easily exceed the available bandwidth between simulator and visualizer. Efficient encodings, even lossy compression, have been employed to alleviate this communications bottleneck [35]. Another approach is to run the visualizer on the same (super)computer that performs the simulation, thereby hopefully gaining access to any needed data for visualization on demand in less than a frame time. However, this requires that the observer be physically close to the simulation engine, or that there exist a fast video link between the visualizer and the display screen used by the observer [30]. The video link approach also requires a low-latency command line from the observer to the simulator to make the user's normal movements and interactions with the environment reasonably responsive. In such a set-up, it would be more difficult to realize a collaborative

environment in which individual observers can sign on at will from anywhere in the country at any time.

Densely occluded interior environments such as buildings, boats, planes, or caves offer certain advantages for immersive environmental simulation. They can take advantage of the same kind of preprocessing that has already been demonstrated in the context of visualization of static models [12]. Only those simulation results that affect the currently visible set of spaces need to be transmitted to the visualizer. A cell-based decomposition of the densely occluded world allows an effective estimation of a tight yet still conservative superset of the data that is absolutely necessary for visualization at any moment in time. As long as the number and complexity of the cells visible at any time remains bounded, the size of the whole world model can be, in principle, arbitrarily large – as long as there is sufficient (super)computer power to keep the ongoing simulation up-to-date.

2.1.4 Shortcomings of First Generation Systems

While first generation systems provided basic tools and solutions to many of the fundamental problems facing specific virtual environment applications, they were usually unconcerned with the way in which these tools and solutions inter-operated. For example, none of the aforementioned systems allows the model to be changed by the observer in any meaningful way at run-time. Such changes would result in a need to recompute sections of the database structure; this often involves complex precomputations which would be slow and difficult to distribute to the other affected clients. Indeed, the model environment itself was typically not even centrally served to the users; almost all of these systems use a replicated world database that must be present in its entirety at each client when the simulation starts. Only a few systems have also distributed world state [36, 27, 22] and those have not considered scaling to large models of the size of cities or buildings, nor to many hundreds or thousands of users sharing and modifying the space.

The ability to distribute multiple clients and servers provides a second level of scalability for those models that cannot fit on a single server, or have so many clients that a single server would be a performance bottleneck for the system. This functionality is addressed indirectly by virtual walkthrough systems in that they address the need to page information in real-time from a slow secondary storage system, and store only a portion of the environment in RAM at any time. One can substitute “network” for “disk” in this work, since the issues are similar. However, these systems provide read-only performance for a single user; they do not address interactivity between users, nor do they address the integration of simulators into the world model. User-user and user-

world interaction add another level of complexity; having multiple entities that interact in real-time in and with the virtual environment involves making efficient use of available bandwidth at each machine in the network. This, in turn, can impact the scale of what can be distributed and rendered in real-time.

Finally, for many applications, the environment will only be truly useful if it supports physically realistic behavior. The types of realistic behaviors that are important depend on the specific application; in a flight simulator, it is important that the flight dynamics of the aircraft be realistic, whereas in a firefighter training application, the flame and smoke spread and combustion chemistry are the important elements to get right. Research systems have not generally addressed this area; production systems have been confined to small, localized models and focus on the interaction of a single user with a small but complex model. Where multiple users are concerned, they are generally sharing the output device and are observers rather than actors.

2.2 Citywalk: A Second Generation Architecture

We have now reached the point where networks and workstation hardware can support systems that combine these techniques to provide a richer, more useful virtual world experience. To support a functional and robust fusion of these first generation approaches, we need a system that combines aspects of a distributed persistent database, that can provide model storage and support intelligent model loading and unloading, with a high-performance network layer that provides interactive speeds for time-critical aspects of agent-agent and agent-world interactions. This leads naturally to a two-tiered architecture with a tightly linked object database paired with a data distribution layer that can rapidly propagate critical information between clients over limited-bandwidth links. Practical applications involving realistic physics also demand that the system provide a framework for integrating physical simulations that can be "plugged in" to the system and act as agents in parallel with the users, reacting to user inputs as well as with each other.

2.3 Overview

This thesis presents an integrated system that attempts to address these factors. Chapter 3 discusses the database substrate that provides the foundation for the scalable, interactive, distributed world model. Chapter 4 describes the real-time simulation layer that provides the ability to integrate distributed physical simulations and clients with intelligent use of bandwidth and system resources

to provide realism and real-time interactivity. Chapter 5 describes the practical experiences with integrating multiple third-party simulations into the environment. Chapter 6 describes Floorsketch, a utility tool for rapid prototyping of databases, and chapter 7 contains a final summary, analysis of the system architecture, and concluding thoughts on the project.

Chapter 3

Database Support

3.1 Design Goals

Many first generation walkthrough database systems were designed to address the problem that an interesting virtual environment model is too large to fit in memory at once, and too large to render naively with existing hardware graphics pipelines. This issue remains relevant even in the face of order of magnitude advances in PC technology over the last decade. Though the original flat-shaded single-building databases rendered by the Berkeley walkthru program are much easier to render on modern machines, interesting world models have increased dramatically in both scope (entire cities versus individual structures) and in detail level (radiositized, fully textured models instead of flat shaded models) (Figure 3.1).

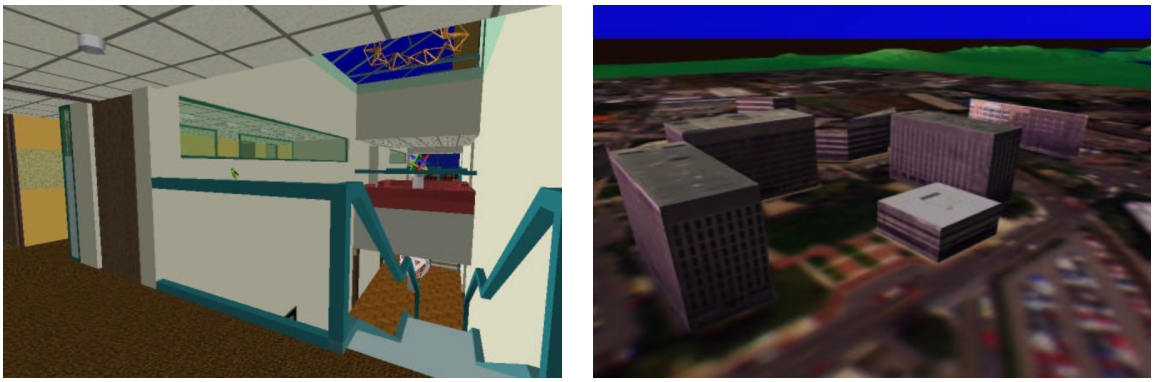


Figure 3.1: *First and second generation Berkeley Walkthru databases. The first generation concentrated on a homogeneous interior environment; the second generation scales up to a heterogeneous, indoor and outdoor environment.*

First generation systems developed a set of techniques developed to solve the model size problem for a single freely moving user in a static environment. Solutions for issues that arise in having multiple users interacting within a shared environment, or having simulation agents interacting with users and the world, have not been well integrated with these "large model" walkthrough techniques. One of our goals in developing a second generation walkthrough system was to improve upon the strengths of the first generation systems (e.g. larger, more detailed models, and more variety and speed in the types of spaces and visibility methods used) while introducing integrated techniques that support multiple simulation and viewing agents; robust, rich interactions between agents and the world; and which run efficiently on distributed platforms.

3.1.1 Large Model Visualization

The database underlying a second generation walkthrough system must subsume the functionality represented by a first generation database. Funkhouser asserts in [19] that the critical aspects of a database storage system include "1) Store very large models; 2) Support persistent addition, deletion, and modification of data; 3) Support efficient access to data by application-defined functions; 4) Allow asynchronous, application-defined memory management functions; and 5) Perform efficient transfers from disk into memory." He also asserts that "traditional" database systems provide excessive feature sets that reduce performance significantly, to wit: 1) Crash recovery, 2) Application-defined functions must be able to run outside of the database address system, and 3) General purpose queries require execution of a query engine, and copy the resulting data into buffers. As a result of this analysis, Funkhouser built the customized database engine that drove the first-generation Berkeley Walkthru program. This engine provided the ability to define "segments" that were not interpreted by the database, but could be stored or loaded as a unit.

3.1.2 Efficient On-Line Model Updates

First generation systems supported persistent addition, modification, and deletion of data in the world model database. However, most were not designed to do these things *interactively*, maintaining consistent, up-to-date views for the clients as the model is being manipulated. These systems typically add data to the model in an offline (batch) fashion; modifying the database and viewing it are mutually exclusive activities. Some systems provided rudimentary support for modifying some parts of the database (c.f. Berkeley Walkthru Editor [37]). However, interactively modifying the world structure in such a way that the visibility data structures are themselves modified

was not supported, limiting the user to adjusting details such as color and positions of furniture.

A second generation system should support *on-line* modification of *all* portions of the model, including structural or visibility elements. While it may not be possible to provide algorithmic updates to visibility structures, in cases where the computations *can* be performed in real time, the database layer should not pose an obstacle. For example, in the original Berkeley Walkthru, supporting real-time modification of the building structure (moving walls and floors) is algorithmically quite tractable, as it consists merely of readjusting a few KD-tree cell partitions and changing some coordinates. Unfortunately the database and viewer lacked a mechanism for discovering, loading, and integrating modifications of visibility structures in real-time. Thus, any change to the database would require stopping the viewer and reloading the entire visibility structure, which is impossible to do while maintaining an interactive frame rate. Furthermore, the viewer cached and pipelined information on the assumption that these structures would not change. This makes interactive changes to the visibility structure infeasible.

Partially as a result of this batch mode of database modification and lack of on-line changes, efficient updates in the presence of dynamic model changes was often a low concern in first generation systems. The first generation Berkeley database did not support efficient deletion of objects; the database file grew monotonically over time, as new segments were allocated contiguously at the end of the file. While this did improve the monolithic load performance of the database (continuous segments are faster to load than non-contiguous segments), it meant that an actively modified database rapidly grew to an unwieldy size, and contained an ever-growing fraction of "dead space." A second generation database needs to be able to support efficient modifications of the database, and efficient garbage collection.

3.1.3 Multiple Interactive Agents

Furthermore, a second generation system should support *multiple agents* operating on a database at the same time. This requires the ability to operate in a server/client mode on a database with state, as well as requiring locking and transaction functions, which are necessary to maintain database consistency in the face of simultaneous accesses.

Finally, we wish to make integration of multiple different types of agents as easy as possible for the developer. There are many groups working on real-time simulators for various aspects of reality; physics, fire, lighting, and so on. A versatile second generation database should be able to leverage the work and expertise of these other groups by allowing the integration of their code

into the world model with a minimum of conversion and rewriting overhead (Figure 3.2). The more difficult the integration process, the less likely it is that these codes will be successfully integrated into the system. Note that this was also a concern in some first-generation systems, primarily for the purpose of integrating different visibility and rendering systems. Advances in common languages (i.e. java and C++) and careful thought have allowed us to improve upon this aspect of our database design.

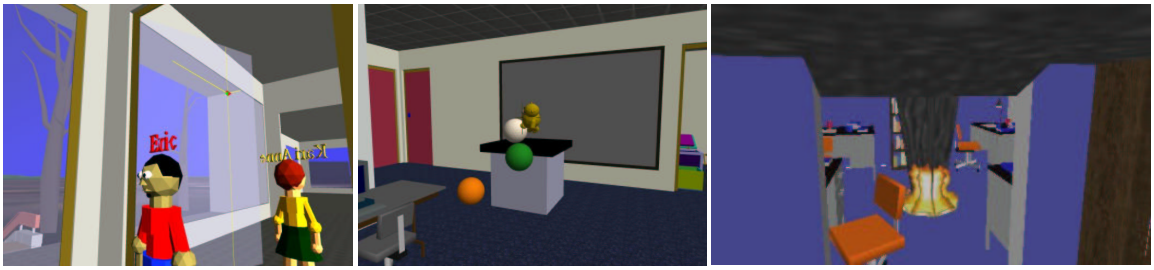


Figure 3.2: *Multiple different types of agents active within the same environment; multiple users, dynamics, and fire.*

3.2 Why Not Use An Off-The-Shelf Product?

Many of the capabilities needed for a second generation walkthrough database are provided by commercial general-purpose object database systems, such as POET, Objectivity, and Objectstore [38]. These databases all provide persistence models that are straightforward to apply to existing C++ code, allowing the programmer to add persistence to bodies of legacy code that were not designed to operate on persistent objects. They all provide server-client functionality, efficient notification mechanisms, locking and transaction semantics, and multiuser capabilities. Since they are commercial products, they are generally well optimized and provide good performance.

In the first generation walkthrough, a custom object database was used instead of a more general-purpose object database. This design decision was necessary because, at the time, the general purpose databases on the market did not offer sufficient performance on desktop workstations; the visibility system could not load objects quickly enough to support a consistent rendering rate of at least ten frames per second. In general, commercial databases have not been used for high-performance walkthrough systems because of performance concerns. These concerns are largely attributable to the fact that such databases are performing much more work than necessary, due to the fact that they support additional capabilities such as SQL query functionality and global object

indexing mechanisms. These functions slow performance without offering the walkthrough system any real benefit.

Of course, in the intervening time period workstations have become faster, and a commercial ODBMS could now support a first-generation walkthrough system at a reasonable detail level. However, a custom database is still useful for several reasons. First, it remains the case that a custom database system that does not implement functions that are not useful to the walkthrough system will provide better performance. Second, with our custom database we have been able to add useful functions that are not available in existing commercial systems, such as the ability to have nonpersistent objects that are managed by the database and have a lifetime equal to that of the client process that created them. Third, with a custom database it is possible to freely move functionality between client and server, and minimize traffic across the relatively slow network link. Finally, it is a hard reality that commercial databases are expensive to license and expensive to support, whereas a custom database requires only an investment of time on our part. Thus, while it would be possible to use a commercial ODBMS as the underlying database model, we decided to go with our own database system in order to provide optimal functionality and performance.

3.3 Specification

3.3.1 API Overview

A database object corresponds to a small set (a "constellation") of C++ objects that are arbitrarily interrelated via C++ pointers. One of the objects in the constellation is the *master* object; this object must be inherited from a persistence-capable base class that provides identity tracking, storage management, and lock and watch management for the entire constellation via a virtual interface. Constellations have a unique, universal identity that can be used by any other object to refer to the constellation. Each constellation constitutes an atomic *model component* that can be loaded or unloaded independently of other model components. For example, one KD-cell in a Berkeley architectural walkthrough model (the DBCELL class in the Walkthru code), along with its contents and adjacency lists, precomputed visibility structures, and modification information, is implemented as a single constellation, with several types of separate component objects managed by the DBCELL master object (Figure 3.3). Each DBCELL in the building model is a separate constellation, as are other major types of object like class master definitions (DBCLASS objects), geometry (DBGEOMETRY) and material definitions (DBMATERIAL), and other objects that were

stored in segments in the original Berkeley walkthru database. Thus, each individual DBCELL object may be loaded or not loaded in each client, and a cell may be loaded independently of any particular model object in the cell; but a cell may not be loaded independently of its contents list or its visibility structure.

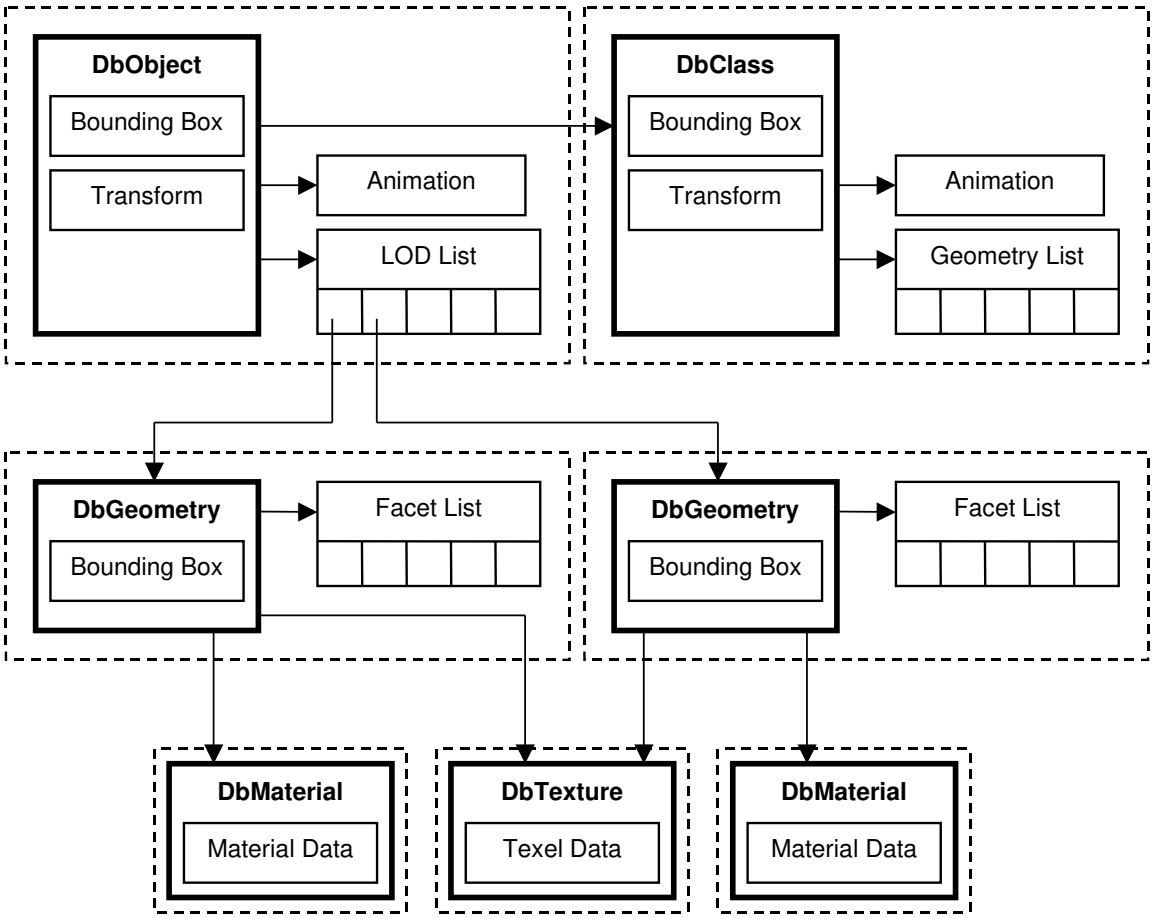


Figure 3.3: *Interrelated constellations describing an object in the Citywalk database. Constellation boundaries are dashed; the master object for each constellation has a thick border.*

Any object that has a handle to the database may request a C++ pointer to the constellation's master object by querying the database with the constellation's identity. If the constellation is currently memory resident, the database simply retrieves a pointer to it from a lookup table. If the constellation is not resident, the database must execute a load cycle to make the object resident. A constellation's memory residence status can be also be queried without forcing it to be loaded. Ejection from memory is handled by reference counting to allow sharing of objects between independent threads; dereferencing the identity into a pointer causes a reference increment. Constellation iden-

tities are assigned at the time that the object is first committed to a database. Once an identity is assigned, that identity forever specifies that particular constellation in that database, and will never be reused to refer to a different constellation. By storing an identity reference rather then a C++ pointer, constellation *A* can refer to a constellation *B* without forcing constellation *B* to be memory resident at any time that constellation *A* is resident.

A C++ object that is acting as a constellation master must implement virtual function overloads for functions that serialize the constellation to a specified buffer (i.e., turn the set of linked C++ objects into a contiguous block of data), return the current size of the constellation's serialization, or de-serialize it from a buffer. This interface is used by the database engine to pack the objects into "blobs" that can be committed to disk, transmitted across network links, or reconstituted later into the original object. The master object's packing interface is responsible for serializing the entire contents of the constellation into a form that can be uniquely reconstructed from just the contents of the buffer. To provide for system independence, our database core provides a special binary buffer class that provides XML-compatible toolkit functions for converting member variables to and from serial binary representations. As long as the object's author uses these toolkit functions to serialize member variables to and from the buffer, the constellation blobs are in "network form" and can be transmitted to and reconstituted on machines of different architectures. In practice, we use our databases interchangeably on both Silicon Graphics machines (which use MIPS processors with little-endian byte encoding) and generic Intel PCs (which use Pentium processors, which are big-endian). Moreover, we use two different operating systems on the Intel machines (Windows and Linux). All servers and clients on these machine/OS pairs are compiled from the same source code base. This system-independent nature of the database library and format has proven to be very convenient.

Our second-generation database library also allows for both *local* and *server-client* mode. In local mode, the operations take place directly to a database disk file; local mode access allows only a single user to be operating on the database to avoid file system collisions. In *server-client* mode a database server proxy operating in local mode provides the same interface to multiple clients that the local mode library does, but with full multiprocess-safe locking semantics. Each client is connected via a network socket to the server, potentially running on a separate machine. This distinction is transparent to the visualization engine.

3.3.2 Basic Design

Blob Service Layer

The core of our second generation database design is similar in many respects to the first generation database. The lowest layer is called the *blob service*. This service provides the ability to add, delete, modify, fetch, and store binary blocks (called blobs) between memory and some shared persistent storage medium. Blobs are permitted to be of any size, and are opaque to the blob service layer. When a blob is created, it is assigned a permanent reference tag within the database called a *ref*. The ref is guaranteed never to be reused for the lifetime of that particular persistent store; this prevents referring objects or processes from accidentally retrieving an object which was deleted from the database, but had its reference recycled by the blob service. Refs are not guaranteed to be unique between stores; thus, an object can be uniquely identified over all time and space by a combination of a persistent store identifier and a ref within that store. To provide for very large databases, refs are implemented as 64-bit identifiers, with a dynamically expandable binary storage representation to allow for efficient use in small databases.

Persistent stores may be accessed through two implementations of the blob service. One implementation is the more simple local file implementation; this opens (and locks) a file on a local file system and uses it as a persistent store. This corresponds to a traditional mode of access; the process reading the database is the only process that can access or write to the store, thus there is no need for locking or transaction functionality within the store. The second implementation is more interesting, and the common mode of access in a second generation implementation; that is the server-client mode implementation. The blob service library includes a server executable that can provide distributed access to a persistent store file to many simultaneous user processes. The API access to the database for either implementation is identical. In fact, the server itself is a client for an instance of a local file store, which replicates the API over a TCP/IP socket link to multiple external processes (Figure 3.4).

In addition to the basic add, retrieve, store, and delete functionality, all of which operate on refs and binary blobs, three additional basic functions are supported: Locks, watches, and transactions. There are 3 types of lock that can be applied to a blob ref; read, write, and delete. Read locks prevent other processes from acquiring write or delete locks; write locks prevent other processes from acquiring any type of lock. Delete locks are special in that they only prevent other delete locks; they are used to garantee that an object will continue to exist, though it may be changed.

Watches are used to provide efficient asynchronous notification of database changes to

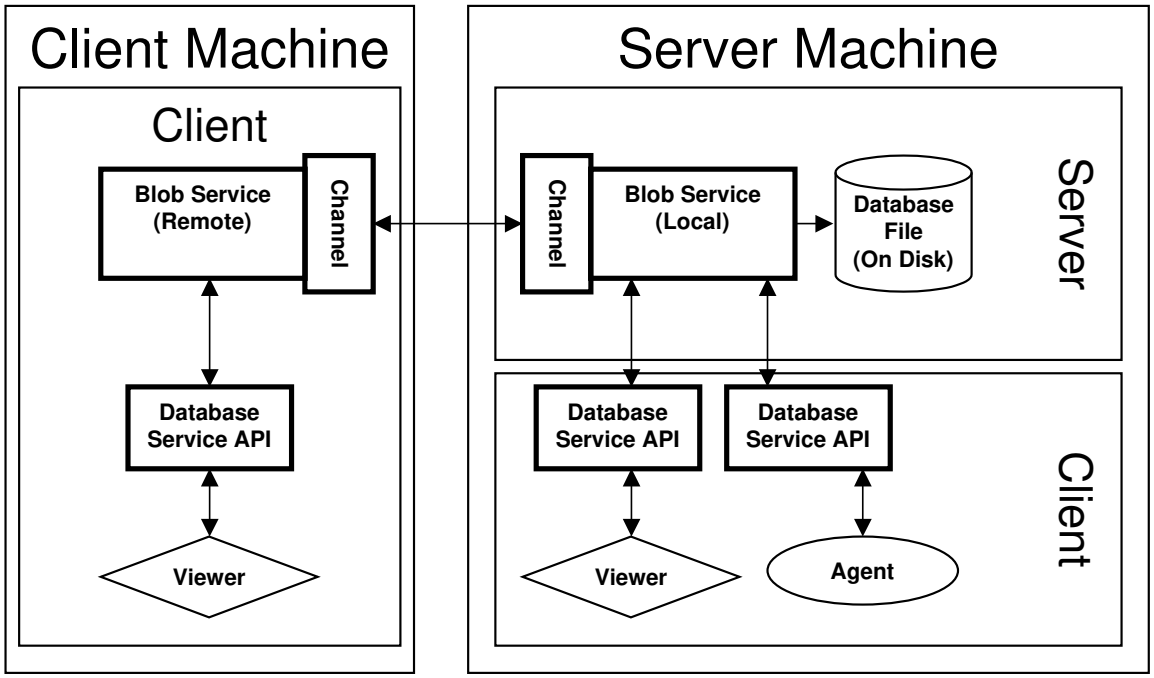


Figure 3.4: *Multiple processes connecting to a database. The blob service is replicated through a socket connection; local database objects provide namespaces in which each database object can be in an independent load and lock state.*

processes that need to do work in response to such changes. Watches can be targetted at an individual ref, or, for storage convenience, at a set of refs. Watches come in 5 types: Load watches, which fire when a process reads a blob for the target ref; Store watches, which fire when a process writes a blob for the target ref; Delete watches, which fire when a process deletes the blob for the target ref; Lock watches, which fire when a process locks the blob for the target ref; and Unlock watches, which fire when a process unlocks the blob for the target ref. When a watch is set, the setting process provides a function pointer to the database. When the watch fires, this function is called with two parameters: the type of watch fired (which may be more than one) and the ref that caused the triggering of the watch. The set of watches is stored in an associative table indexed on ref for efficient access. Note that watches are not ordered, nor do they provide any time guarantees. It is guaranteed that the watch will fire exactly once for each triggering event, but the call of the function may happen an arbitrary amount of time after the operation actually occurs, and multiple watches on the same operation will fire in an unpredictable order (Figure 3.5).

Transactions allow the grouping of sets of blob operations into atomic operations that either fail or succeed as a whole, preventing database corruption via either partial modification of

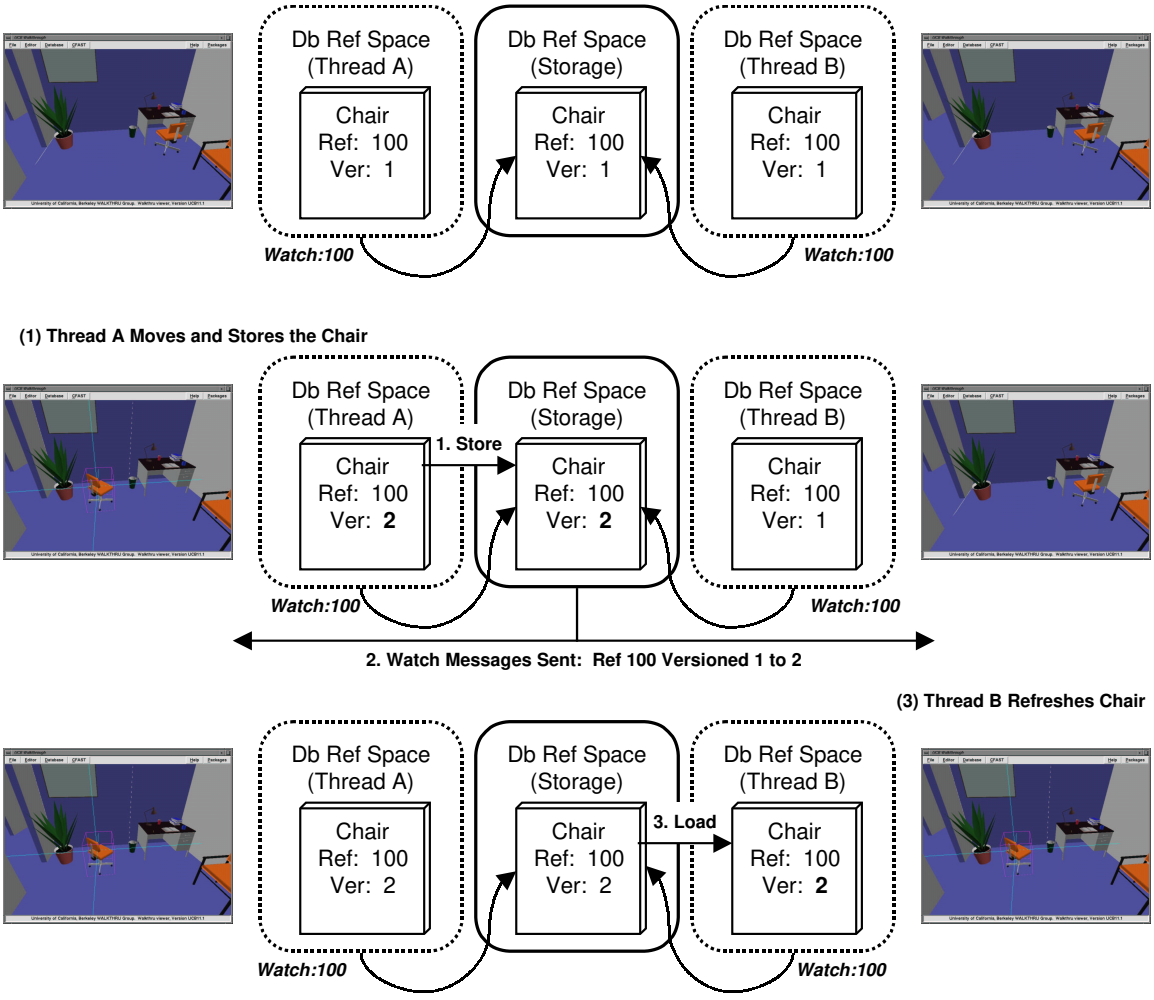


Figure 3.5: *A typical watch firing sequence. When a transaction commits, some objects are modified; these trigger watches set on that object by other processes, which are notified via a message in an asynchronous thread. How they respond to the watch is defined by the individual process. Note that neither the time to receipt nor the order of the notifications is guaranteed; processes must anticipate this and be ready to operate on incomplete information in time-critical situations.*

the database or partial writing of the disk file. Transactions have only 3 operations: *Begin*, which starts the batch; *Commit*, which causes all blob operations between the last Begin and the Commit to be either successfully executed, or not executed at all with an error generated; or *Abort*, which causes all operations since the last Begin to remain unperformed and cleared from the todo list. The implementation of the transactions is done on the file level; a multi-stage commit mechanism guarantees that the file always represents a valid database regardless of where in the commit process it may be interrupted. Any Store or Delete watches on objects which are stored or deleted during the body of the transaction will not fire until the transaction successfully commits, since the changes are not actually made to the database before that time.

Each client process that attaches to the database is provided with a unique key value which is used to apply locks, watches, and transactions to the database; this key value establishes the identity of the locking, watching, or transacting process. The tables of active locks, watches, and transactions are indexed by the ref of the blob. Note that a blob's existence in the database doesn't affect the lock and watch functions; thus, a database may have no blob for ref x, yet some process may still have active watches and locks on x. This means that locks and watches can outlive the objects to which they were applied, or can be applied before the object is actually committed to the database. This property can be very convenient for two reasons. First, due to the asynchronous nature of watches, more than one operation can occur between the event and the receipt of the watch for that event, and those events could cause problems if the watches were automatically mutated or invalidated to keep them consistent with the presence of a blob in the database. For example, an object will be deleted before its delete watch notification is received by the client. The object may even be deleted, then reinstated and deleted again. If delete locks were automatically removed without indication to the client, then this situation would be difficult to deal with, as it means that valid notifications might be lost; if the delete, reinstatement, and a modification all happened before the receipt and processing of the delete message, that process might miss notification of the modification because the watch was invalidated and the process was never given a chance to reinstate a write watch after the object had been restored. The same reasoning applies to reference counts on watches or locks, which can become out of sync if they are forcibly removed when a blob is deleted. The separation of watch and lock tracking from the blobs themselves also means that the lock and watch service can be separated from the blob service itself; if the clients are consistent about applying the appropriate type of lock before executing the database operation, then watch and/or lock traffic can be diverted to a separate server from the read/write traffic, allowing improved scalability in a multi-server system by partitioning "ref space" among one or more lock and watch

servers (Figure 3.6).

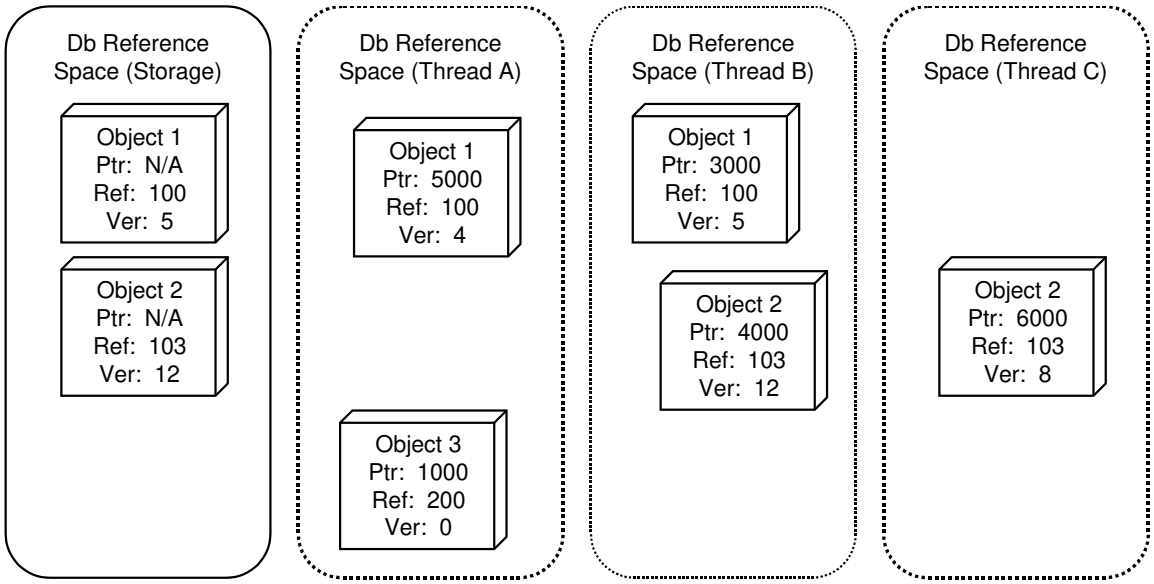


Figure 3.6: *Different referential spaces for different processes. Particular portions of the model may or may not be loaded in each space, or may be older versions awaiting update.*

The final major feature of this layer is the prefetching interface. A load may be issued as a prefetching load; this will initiate the load command to the server or file without blocking the caller, and return a value indicating that the blob is not yet present. With this call, the caller can attach an event callback which is invoked at arrival time of the blob. If additional requests come in before the object arrives, the event is simply added to the set of events waiting for the object; of course, any or all of the callers can simply block until the object arrives, as well.

Since the blobs are opaque to the database service, this layer is also capable of supporting language-independent services via the socket interface. The simple request API at the socket level can be used by any language that can make socket connections. As an example, we have successfully used the database at this level with a Java client.

3.3.3 Object Service Layer

The object service layer is built on the blob service layer and provides the C++ object-level interface to the core database functions.

Object Factory and Schema ID

In order to reconstruct objects from binary buffers, an object factory is needed. This is a global object that can construct an instance of a class on the heap, given an ID code that is universal and unique to a class over all programs that use that class. In our system, the codes are integer values that are assigned when the class is created and are never repeated. Our object base class provides virtual functions that return the precise schema ID for a given object.

Persistent Objects

The storage and reconstruction mechanism is based on a virtual interface called the packing interface. This interface implements functions for asking an object how much buffer space it needs for storage, serializing the object to a stream, and deserializing the object from a stream. Each object is also derived from a base class that stores three things: a reference count, a pointer to the database object to which the object belongs, and the database ref which is the object's identity within that database. Finally, a database-capable class must be registered with the system's object factory, with a schema ID that is unique to that class over all the client programs that will access this database. These interfaces are combined into a single base class, DfObject, from which all database-capable objects in the system must inherit and implement the appropriate virtual functions.

Assigning the object to an open database consists of simply constructing the object with a standard construction method and calling the Assign() function to make the object persistent. Assign allocates a new ref from the database, stores that ref and the database pointer in the object, sets the object's reference count to 1, and records that object as the current memory instantiation of that database ref. Once an object has an identity and a home database, it is a fully capable persistent object (Figure 3.7). Note that assignment results in a reference on the object; the function that assigns the object to the database automatically starts with an active reference to the object and is responsible for tracking that reference and eventually releasing it once the object is fully initialized (i.e. linked appropriately into the database's application structure). Note also that a newly created C++ object also has a reference count of 1; in the case of an object that is unassigned to a database, dereferencing an object to 0 will delete the object. This means that any object inherited from the DfObject base class may be tracked (referenced and dereferenced) in the same way as an assigned object would be, and the dereferencing operation does the appropriate work (i.e. deletes the object when it is no longer of interest to the program). This makes application programming and member function programming easier, as a uniform set of code may be used for both persistent

and nonpersistent data structures.

Writing a persistent object's state to the database is initiated by the user calling the member function Store(). First, the store function queries the object for its size and creates a buffer of that size, plus a small portion for storing the schema ID of the object. The ID is written at the beginning of the buffer, then the object's packing function is called on a stream adaptor attached to the rest of the buffer. This stream adaptor performs conversion of the data to network byte ordering as it is written into the buffer; this makes the data system independent. Once the write is complete, the buffer size is compared to the expected size as a sanity check; if it passes, the buffer is written to the blob layer under the appropriate ref.

Reconstructing an object from the database given its ref proceeds in a similar fashion. It is initiated by a client process with an open database object and a ref to the desired object. The first thing the database does is check if the object is already memory resident; if it is, it increments the ref count and returns the object pointer. If it is not, it calls the lower layer to retrieve the blob buffer for the specified ref. After attaching a network-byte-order converting stream adaptor to the buffer, the schema ID is read and the object factory is called to construct a generic instance of that object type. The remainder of the buffer is then passed to the deserialization function of the generic object to load the data into the object. The object is assigned a reference count of 1 and placed in the memory table of the database as the current memory instantiation of that database ref, and the new isntance is returned to the caller.

Smart Pointers

The smart pointer is an important component of the database system. When porting code or writing new code, objects will point to each other to form data structures. In the case of persistent objects, we want the ability to have a pointer reference an object without forcing the object to be memory resident. In the classic walkthru system, this was achieved by explicitly "swizzling" and "unswizzling" the pointers between actual references to valid objects and an opaque value that referred to an abstract object in the database. The client program was responsible for determining when to swizzle or unswizzle; legacy code typically required the complete data structure to be unswizzled before any operations could take place, because legacy operations assume that all pointers are either NULL or reference valid objects. In the case where an object was referenced by more than one other object, or by multiple threads, the referring objects had to explicitly maintain reference counts to make sure they didn't unload an object that was in use by another referring object.

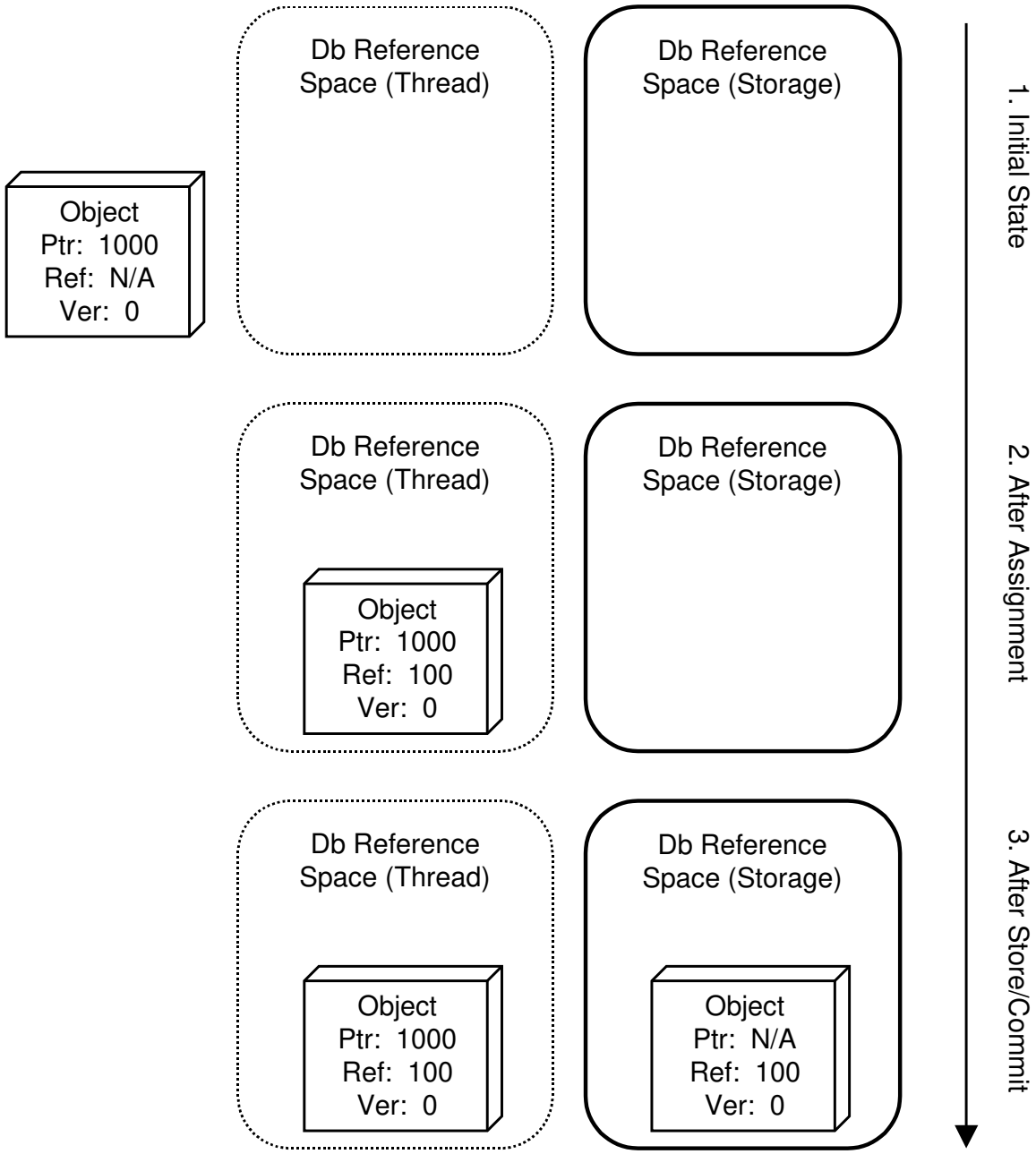


Figure 3.7: *Assignment and identity of objects. When an object is assigned, it takes on a universal identity that it shares in all referential spaces. On assignment, the object acquires a DbRef value that identifies it to all processes (e.g. the* **Ref** *field). The first assignment also establishes version 0 of the object in the persistent store.*

This structure required a great deal of bookkeeping and led to odd errors, because it was impossible to determine whether a pointer was a valid memory reference or an opaque swizzled reference.

In the second generation database, this functionality is replaced by the templated smart pointer class. The smart pointer is overloaded to behave like a regular pointer with regards to dereferencing and member access operations, which allows legacy code to operate unchanged on the objects. Internally, the smart pointer contains a database pointer and database ref that denotes the object's identity. The pointer may or may not at any given time actually have the object loaded; if it is loaded, the reference is tracked and the object is dereferenced when the smart pointer is unloaded, freeing the calling code from having to track references explicitly.

During a regular C++ access to the pointer value, the smart pointer checks to see if it has the object loaded, and loads it if necessary before returning the memory pointer. This allows legacy code to operate transparently without requiring preloading of the data structures. Furthermore, the code can go back and unload smart pointers without affecting the operation of legacy functions. Finally, a smart pointer has an access interface that allows conclusive determination of whether the object is loaded or not, as it can check whether its tracked pointer is NULL without actually losing track of the object reference.

The smart pointer implements the full packing interface, so it is trivial to include it in the packing functions of the container class. Internally, the pointer stores the database ref in the buffer, a value that is typically only as large as an integer for smaller databases.

Object Serialization

Object member data such as floating point numbers, arrays, or integers are straightfoward to serialize into a buffer; given the correct network byte ordering, the data is simply appended to the buffer, and can be read back in the same order it was written in. However, storing links between objects is slightly more complicated.

When an object contains pointers to other objects, the serialization process depends on two factors. First, the designer must consider whether the object being referenced is persistent or not. For example, the DBCELL object points to a number of other data structures that are pointed to, rather than contained in, the DBCELL itself. These include a separate contents record and a linked list containing condensed descriptions of all the visible paths from this cell to other cells in the model (Figure 3.8). Since it is not useful to load the visible path list without loading the DBCELL itself, and the list is referenced only through member functions of the cell, the list is not made persistent; it

is considered part of the DBCELL. Similarly, it isn't useful to be able to load the contents structure independently of the KD-cell description, nor can the contents structure be accessed directly from outside the DBCELL, so the contents structure is also nonpersistent. Conversely, the contents lists of the cell contain references to detail objects (i.e. furniture) that are either contained in or incident to the cell. Since it *is* useful to be able to load a furniture object independently of any particular cell that contains it, these detail objects are implemented as persistent objects (i.e. constellation masters).

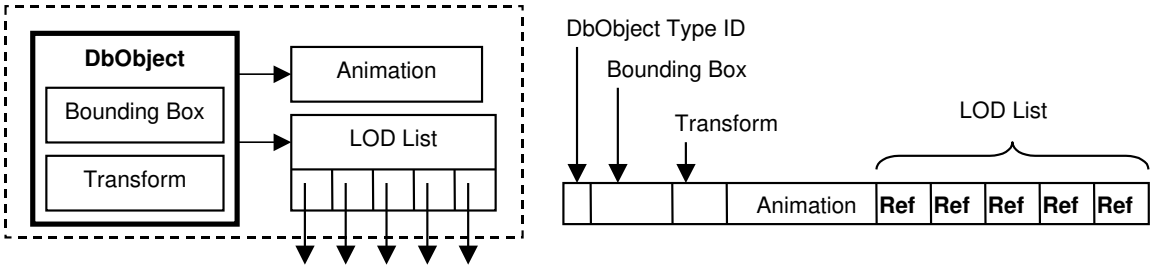


Figure 3.8: *Serialization of objects converts them to a machine independent form that can either be stored in the database or transferred to other processes via sockets. Note that references external to the constellation are stored in* **Ref** *form.*

When a nonpersistent object is referenced and maintained by a master object, but is not a part of that master object, the master object must take responsibility for storing and reconstructing the slave object when the master is stored or loaded. For example, the packing interface of the DBCELL is responsible for reserving enough buffer space to encode the size and contents of the visible path list, to write the size and contents to the buffer on a store operation, and to reallocate and reconstruct the list on a retrieve operation. In the case of a persistent object, such as a detail object in the DBCELL contents list, the designer will typically use a smart pointer object rather than a typical C++ pointer. Thus, while the DBCELL is responsible for packing the contents list itself, it is not packing individual objects; rather, it is storing a list of references to external objects which can be loaded independently of the cell itself.

Platform-Independent Persistence Model

The platform independence of the persistence model relies on the fact that the blob service layer is transmitting, storing, and retrieving opaque binary blobs by their ref tags and schema type IDs. Thus, any client of any language or architecture can use the socket API to request a blob by tag, for which it can reconstruct the class represented by the schema ID in any appropriate local

fashion by (1) creating an object of the class represented by that schema ID, and (2) Decoding the contents of the blob as appropriate into that local class. The local object can provide any desired or needed representation of the data, in any byte order or layout. At storage time, the local class is back-converted via a local API call into the common XML layout for that object type and presented back to the blob layer as an opaque unit once again. This approach even admits multiple bindings of the class in different threads, so long as the universal data representation defined by that schema ID is obeyed.

In theory, each language and machine architecture can have its own implementation of the object service layer. In practice, we have created a full implementation for C++ and a rudimentary implementation for Java. As most client architectures can compile our C++ implementation, we have been able to provide database functionality on several machine and operating system types with minimal effort, including SGI Irix, Linux, and Win32 platforms.

3.3.4 Modification and Transaction Semantics

Modifications to the database are typically executed with transaction semantics. Transactions are nestable and provide the ability to atomically update a set of database objects. This capability guarantees consistency among constellations. Additionally, the capability minimizes the damage that can be caused by bugs and crashes in individual clients. Once a transaction is begun by a process, all object modifications executed within the transaction are buffered rather than written to the database. The transaction may be committed or aborted at any time. If aborted, the database remains unchanged; if committed, the database appears to atomically update the state of all objects modified in the transaction. This capability becomes increasingly important as more clients are simultaneously using a database; without it, a single client crash or bug could render the database inconsistent and unusable, even given proper use of locking semantics. It would also be very difficult to guarantee consistent updates of the database, as any given client might see only a partial view of the changes if they were not executed atomically.

3.3.5 Locking Semantics

Object locking occurs at the level of a constellation, which can be locked by a process via its identity. Several types of lock are available, corresponding to a “many readers, one writer” locking model. A read lock is non-exclusive, allowing other processes to also obtain read locks, but preventing other objects from obtaining write or exclusive delete locks. Exclusive write locks may

be obtained only if no other process has a read lock, and prevent any other process from getting any type of lock; non-exclusive write locks allow other processes to obtain read locks, but no other type of lock. Exclusive delete locks prevent other processes from obtaining any type of lock, while non-exclusive delete locks simply prevent any other process from deleting the object.

Locks are applied to a section of the database when consistency of reading or writing across a set of more than one constellation is necessary. For example, if the editor needs to move an object from one cell to another, it must write lock both cells, ensure it has the most current version of those cells, and delete lock the object before making the change. This prevents inconsistencies caused by collisions with other edit processes. Weak locks should be used when possible so that processes that are interested only in reading the database are not blocked by the editing process; for example, if a write lock were used on the object in the above example, a reader would be blocked from reading the object even though the edit process is not actually going to change the object.

3.3.6 Watch Semantics

Constellations may also be "watched" by processes via the constellation identity. The blob service maintains a set of watches on objects in that database, indexed by the database ref. A watch request includes a mask of desired events (possible events are read, write, delete, lock, or unlock) and a call-back function. While a watch is in effect, whenever any of the specified events happens to the constellation, the call-back is invoked by the database core from a separate thread. Watches are fired asynchronously in the database core; due to the vagaries of network traffic and machine load, there are no absolute guarantees about the amount of time between the event and the time that the watch arrives at the client, nor are there any guarantees of the order of watch receipt in the case of multiple changes being made atomically, as is the case when a transaction is used. However, in practice, store and delete modifications tend to happen fairly quickly and in bursts when transactions are committed to the database, because all the store and delete operations in the transaction happen atomically at that time, which causes the set of watch callbacks for those operations to happen at once.

Watches are the mechanism by which clients are notified of changes made to the database by other clients. Because watches can be applied at constellation granularity, notification traffic can be dynamically tailored by the client to just those portions of the database of interest to that client. This provides scalability in the network traffic and client portion of the notification process. Scalability on the server itself can be provided by using remote or hierarchical watch servers. This

will increase latency between the execution of the operation and the arrival of the watch, but would be necessary in the event of a very large number of clients watching the same portion of the database.

3.4 Programming Concerns

3.4.1 Ease of Extension

The ease with which new visibility algorithms and simulation or active agent codes can be integrated with the walkthrough is vital to the success of the system as both a research platform and an integration framework. Funkhouser pointed to integration as a major challenge of these database systems, as there is a great deal of sophisticated legacy code which could contribute to an interactive world database.

Ease of integration was achieved in the first generation Berkeley Walkthru database by using a packing system whereby objects were binary-copied into segments, then a user-definable postprocessing function converted internal pointers to offsets within the binary segment. Loading the object back consisted of running the same function to convert the offsets back into pointers.

The advantages of this approach are twofold. One, object reconstruction is extremely quick, as new memory allocations are not necessary and copying of non-pointer data is kept to a minimum. Second, existing data structures could be binary copied verbatim into and out of database segments with the exact layout they had in memory; in general, elements such as KD-tree cells could be read from the database and operated on directly by the code written for the nonpersistent versions of these objects.

There are a number of disadvantages, though. First, when going to a fully object-oriented language such as C++, binary copies of objects are invalid when moved between runs of the program, as their virtual function tables and internal layouts can differ depending on inheritance and code placement in memory. Thus, the reconstructed objects are often nonfunctional. Second, an object may have internal structure such as linked lists that are presumed by the legacy code to be separate heap structures with certain invariants. Binary copying the lists into a segment, then reconstructing them by altering the pointers to point within the segment, will often lead to the legacy code trying to deallocate part of the segment that it assumes is a separate object, but is actually a portion of a larger allocated block. This leads to segmentation faults and heap corruption which can be difficult to diagnose and impossible to rectify without extensive changes to either the segment unpacking process, which will negate the performance advantage of the approach, or to the legacy

code, which is undesirable.

Third, this approach generates databases that can only be read by a particular build of the database on a particular architecture. Any change to object layouts or recompilation with a different compiler will render all existing databases invalid, and databases are not portable between machines. This is undesirable for a second generation database because it strongly limits the universe of clients that can access the database.

Conversely, the packing interface foundation of the second generation database provides an explicit unpacking of the objects via an object factory and user-defined virtual functions, operating through an XML-based byte code converter. This approach sacrifices speed of reconstruction, but addresses all of the above concerns. Since objects are validly created through an object factory, the layouts are always correct, both with respect to invisible portions of the objects (e.g. virtual function tables) and variances between compilers or different builds of the system. The reconstruction code rebuilds the internal structure exactly, including separate heap blocks if the object calls for them, which results in a reconstruction that is completely indistinguishable from the original object and is much more likely to work properly with unaltered legacy functions. The XML conversion in the unpacking process yields databases that are binary compatible with any machine architecture, thus expanding the universe of machines on which the system can run. Finally, the fact that schema IDs are explicitly embedded in the objects means that not only can older databases be read into current builds, but current builds can actually convert old databases "on the fly" to new schemas and object layouts if the programmer writes code to convert the older binary data to the new schema format.

In the first generation database, inter-segment references were encoded into 32-bit values that fit precisely in the space taken by the actualy object pointers, so that those pointers could be swizzled and de-swizzled in place. This created a limitation that the database could only hold 2^{32} segments. For future expandability, it was desirable to extend these segment identifiers to more than 32 bits; however, this was infeasible in the first generation database. To remove the 32-bit limitation would require either abandoning the binary copy scheme and going to a more complex unpacking method, or altering the objects to place pads or unions wherever the objects have pointers, which would increase the memory footprint of the objects and create potential incompatibilities in legacy functions. In the second generation database, which uses 64-bit extensible identifiers, the explicit unpacking scheme means we can store 64-bit values but leave the 32-bit pointers in the objects. In the case where we want to replace the pointers with on-demand references rather than always-on references, the smart pointer class provides excellent compatibility with legacy code by overloading

standard pointer functions such that they operate indistinguishably from regular C++ pointers.

In practice, our conversion of the system has shown excellent compatibility with legacy code. Adapting a nonpersistent object to a persistent form is generally simply a matter of adding an inheritance from the persistent base class, overloading the packing functions, and replacing interobject pointers with their smart pointer equivalents. In fact, in porting codes from the first generation walkthrough database to the second, most of the required modification of the old data structures were the removal of reference counting and load managing structures that had been added into the original, nonpersistent objects to provide the same functionality that our smart pointer classes provide automatically. The conversion process actually reverted these objects to be more like they were in their original, nonpersistent form, with suitable replacement of pointers with their smart pointer equivalents. The other major issue that comes up involves understanding the layout of a legacy code's data structures well enough to write packing and unpacking routines for that code. This issue is shared with the first generation database system; in fact, in many cases we used the old pointer-swizzling functions as templates for their counterparts in the new system. With a preprocessing step such as the one used by POET, it would probably be possible to automate this process for most legacy codes; however, we have not implemented such an approach.

In addition to providing a simpler, more effective upgrade path to converting nonpersistent legacy objects to persistent objects in the new database, the new objects are much easier to troubleshoot. The unpacking routines are now "safe" in the sense that no assumptions are made about runtime layout, which makes it easier to debug the unpacking process itself, as standard debugging tools can decode the data structures at any step of the process. The smart pointers are also superior to the "swizzling" method for two reasons. First, a smart pointer can be definitively checked at run time to determine whether the object is loaded or not without consulting extra indexing structures, which is impossible with a swizzled pointer. Second, a smart pointer can provide on-demand object loading with no intervention or modification of legacy code, which is also impossible with a swizzled pointer. It has been our experience that the conversion process is easier to comprehend and easier to troubleshoot with the smart pointer tools.

3.4.2 Visualization During Database Mutation

In a first generation system, the preparation of a frame for display is typically initiated by a trigger function. This trigger function is called either by the UI (when the user manipulates the mouse such that the viewpoint changes) or by one of the installed packages, such as the editor (when

the package has made alterations to the scene or needs to render a new frame, so that it can add to or modify the current display via the display call-backs in the frame procedure). Once a frame is initiated, the cull process determines the set of visible cells, the objects in those cells are added to the display list, and the display list is rendered by the draw process.

This process is insufficient for a system that incorporates distributed editing and modification. There is no way for external processes to invoke the trigger function in a particular client, nor is there any way for an external process to determine whether the changes that have been made will affect any particular client. Even if such information was available, there is a severe scalability issue when each client must obtain, in real time, enough state information from all other client processes to determine when those client processes must update their displays. Thus, updates must be handled by each client's interaction with the database itself, rather than by interactions with other clients.

The solution to this issue is found in the watch and notification system. A client is conceptually partitioned into two sets of processes; processes that display database contents (display processes), and processes that alter database contents (editing processes). Note that there may be many modules (editors, simulators, etc.) within one "client" that each have one or more display or editing processes.

For editing processes, the job is actually simplified from first-generation systems. The editing process is now freed from any responsibility to notify display processes of changes in database conditions. It uses a transaction to make its changes, and must only observe proper locking semantics to ensure that the database is not corrupted; but otherwise it is written to behave as if the display process were not there at all.

The display process is responsible for using the watch mechanism provided by the database to keep track of of any database modifications that may affect the display, and to update the database image that it is currently rendering when those notifications are triggered.

For example, consider the UCB Walkthru rendering system [5]. In this environment, a subsystem maintains a view window and a user view frustum in the database, moves the frustum in response to user input, and renders frames corresponding to what is visible from that frustum. Internally, a display list is constructed for each frame given the frustum by the following basic process:

1. Locate the eye point of the frustum in the KD tree for the space.

2. Given the corresponding cell, traverse adjacent cells via portals, adding those cells to the visible cell list, until the visible region has been exhausted, indicating that no additional cells

are visible.

3. Traverse the contents lists of the visible cells and add the objects to the display list.

Understanding this process gives us enough information to write the notification mechanism for the rendering subsystem. Ideally (under the full UCB Walkthru prefetching and memory management system [27]), this subsystem will be maintaining its own memory resident image of a portion of the database consisting of:

1. the database root, which points to...
2. the KD tree for eye point location, which points to...
3. the set of potentially visible cells from the eye point for a certain span of time, given a maximum user velocity and a lookahead time, which point to...
4. the set of detail objects and geometry contained in those cells.

The remainder of the database is swapped out; i.e. it has no memory image in the process' address space. As the user frustum moves through the database, elements (2) and (3) in the above list change as cells and their contents are swapped in and out, as the user moves away from cells behind them and towards cells in front of them.

Clearly, since these database elements are exactly those that affect the rendered view, we must update the view in response to an alteration to any of them. Thus, we apply watches to all database structures in that list. As long as the database is not altered, we may proceed normally, rendering frames and swapping elements in and out as the user frustum moves and the visible sets dictate which new elements must be swapped in, and which are far enough away to swap out. When an element is swapped in, a watch is dynamically applied to that element; likewise when an element is swapped out, the watch is removed.

When a notification is received, we know that some element of the portion of the database that is of interest to the current view has been modified by some external process. Because we are maintaining local images of these objects, we have enough information to complete any current render pass without being concerned about objects changing under the display list and rendering it inconsistent. Thus, we gather notifications in a set until the current frame is finished. Because the update process is completely asynchronous with the editing process that triggered the update, neither the display process nor the editing process needs to block on the other, except for relatively short refresh periods during transaction commit or during the refresh of a resident set.

In the next frame interstice, we update the resident set, proceeding in order from earlier elements in the list to later elements (i.e. KD-cells to detail objects to geometry) . The refresh operator is applied to each constellation for which we have received a notification. Note that this may result in ejection of certain database elements, as KD cells may contain different database cells, objects may have switched parents or been removed entirely, and so on. It may also result in instantiation of completely new objects if elements have been added to cells in the resident set.

Changes that are made outside the resident set are not of interest to the update process for this client, because the objects are not resident in memory. Should we need to access those objects in the future, the standard load process that causes the object to become resident will instantiate the most current version of the object. This provides a scalable solution to updating any number of clients at the client level, as any particular client's watch set is only as big as its resident set, and it has been shown [39] that the resident set is of small, fixed size relative to the overall database size for Walkthru databases.

3.4.3 Effects of Dynamic Update on Frame Rate

Even though updating the resident set during the frame interstice results in correct behavior, it can result in an irregular frame rate, because the time required to update a portion of the model via database access is unpredictable and will generally be high relative to the desired frame rate. This is a natural consequence of the fact that potentially large changes need to be propagated more quickly than the communication channel between clients allows.

To bridge this gap, we continue to render frames while the update is taking place. These frames render the visual representations of the "old" objects while they are being refreshed (Figure 3.9). This requires the old visual representation to be cached. We maintain a cache of visual representations indexed by database ref; if an object is encountered during a frame rendering traversal of the scene, and that object is currently undergoing a refresh operation, the frame renderer draws the cached representation. Once the object is refreshed consistently with the rest of the scene, the cache is invalidated during a frame interstice and the next frame render will display the new view of the object. This cache also retains client-specific information such as the knapsack counters used by the rendering algorithms from the first-generation walkthrough rendering engine. This approach prevents incomplete frames, and prevents the entire visualization engine from "hanging up" while an update occurs. If an object passes out of the visualization working set before it is updated, its cached representation will be destroyed, and queued update requests for that object can be discarded

without processing.

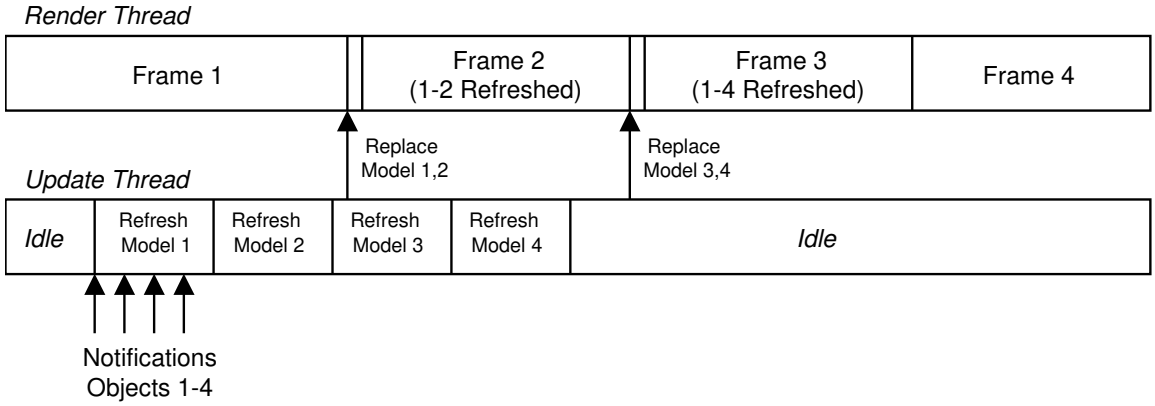


Figure 3.9: *Dynamic update taking place over multiple frames. In intermediate frames, the system may render outdated information in exchange for reducing the visual impact of frame rate discontinuities.*

3.4.4 Updates and Viewing Processes

One of the key concerns in a multiuser system is how to quickly and accurately convey changes made by one user to all other interested observers who might be affected by those changes. We will deal separately with the issues of changing the database and with viewing those changes. When the database is not being changed, rendering proceeds as in any first-generation walkthrough system. However, the protocols to handle database object updates must be made more powerful to handle the presence of distributed, autonomous update processes typical for second-generation walkthrough systems.

Communications between processes can occur in two different modes, one indirect, the other direct. The indirect mode occurs via changes made to database objects (i.e. committed transactions). Assuming that all interested clients have proper watches on the objects being committed, this communication channel provides a nonblocking, asynchronous means of modifying the world state seen by any appropriately connected viewers. Direct communication occurs through a simulation process manager, where the client process can send messages to and receive messages from any simulator to which an active client connection has been established. These messages can consist of anything from a one-byte command code, all the way up to an arbitrary constellation that implements the packable interface.

All communication travels along an abstraction layer that provides control over how band-

width is split between the various simulation client connections and the database update process. This gives the user a measure of control over which elements (database updates or simulations) are most important to have up-to-date in the client view. Each process is limited by the communications subsystem to a certain bandwidth budget per time slice. Packets are dispatched on receipt in a separate dispatch thread, and the receiver is expected to deal with the incoming data immediately.

3.4.5 Scalability

The core principle behind scalability in the walkthrough is that the process has only to maintain and operate on a small subset of the total database at any given time. The cell-and-portal culling scheme, along with the densely occluded nature of the building model and a database layer that allowed a tight superset of the visible set to be loaded at any time, achieved this goal for viewing the static world database.

In the second generation database, the ability to modify the world on the fly, as well as the ability to have many users operating in a server-client mode with a remote database, created the need for consistency across users and the propagation of updates from the database back to clients. The database structures that support these abilities must also follow the scalability principle; e.g. the newly added operations and structures cannot take time proportional to the database size. They must operate in time proportional to the visible sets of the client processes.

Clients interact with each other via two mechanisms; writes to the database, which are propagated to other clients via watch notifications, and direct data distribution via the simulation subsystem, which is addressed in chapter 4. The database system must provide scalable service given the assumption that the world model is structured such that the visible set of world model elements is small relative to the entire world model. Cell and portal culling is an example of a structure that meets this criterion for architectural world models.

If the model is structured in that fashion, then each client will have a small resident set of database elements that it is interested in at any given time. Each client will be setting and removing watches as they move through the database and their resident set changes. Assuming the clients are distributed relatively evenly throughout the world (if they are not, the problem itself fails to scale, since clearly each client must receive updates from each other client's actions), the server's performance will scale well if the work to propagate modifications to an object to the set of clients that are interested in that object is proportional to the number of interested clients. Furthermore, any single server that contains any portion of the world model will not scale well if too many of the

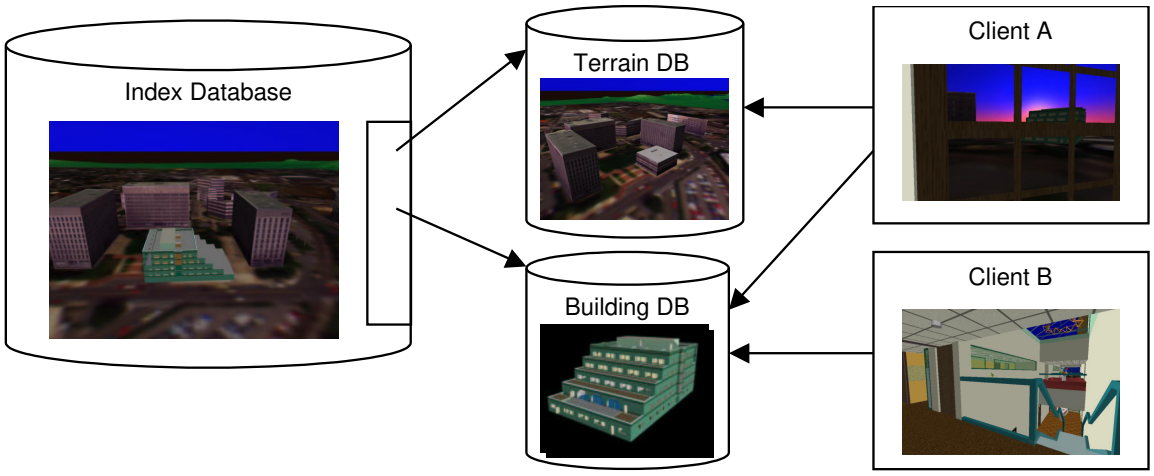


Figure 3.10: *Models can be split over multiple servers, providing scalability for viewers that can incrementally load local regions of the database. References to objects in other databases can be embedded by combining a database location with a* **Ref** *within that database.*

clients are interacting continuously with that server (Figure 3.10).

Our database design admits partitioning of the database services along two axes to help the system's network traffic scale with the size of the problem. First, the world model can be partitioned among several servers. If we partition the model spatially among several servers, then the client traffic will naturally partition among those servers, since clients' working sets will focus on the server that they are "in", potentially plus a few nearby servers [28].

Within a server, there are three major sources of traffic; database reads and writes, lock request traffic, and watch notification traffic. A database modification transaction and its resulting updates typically follow this pattern:

1. Write lock requests are made by the modifying client.
2. Writes are performed by the modifying client.
3. Watches are sent to the receiving clients, and write locks are removed by the modifying client.
4. Read lock requests are made by the receiving clients.
5. Reads are performed by the receiving clients.
6. Read locks are removed by the receiving clients.

Since we are using transactions to modify the database, steps (3) through (6) are quite "bursty" in nature. Many objects will be changed atomically in the database, and all those notifications go out at that moment to many clients, who will all try to refresh those objects at the same time. This causes a big momentary load on the server as it attempts to service all of those lock and watch requests, as opposed to the more distributed load of the clients roaming through the data space. This can impact the frame rate of unrelated clients, since now the "roaming" clients must contend with the impaction of the server caused by this burst of watch notifications.

To mitigate this problem, we can separate the server into two separate subservers, which should be tightly linked to each other with a high-bandwidth link. The primary subserver serves locks, transactions, and object data; the second subserver serves watch applications and watch notifications. When a transaction commits in the primary subserver, instead of dispatching the watches directly, it dispatches the transaction data to the second subserver, which aggregates the notifications for each client together into a single package that also contains the updated object data for each watched object from that client for that transaction. These packages are then sent to the clients from the secondary subserver. This transfers the notification load entirely to the second subserver, leaving the primary subserver to continue handling requests from roaming clients, at the expense of adding the latency of a transmission between the two subservers to the latency of the watch notification.

3.5 Performance

The conversion of the first generation visualization engine to the new database resulted in a small degradation of performance. This was expected, since the new database provides more functionality; however, the impact was relatively minor, since the bottleneck is still the speed at which data can be streamed from the disk or network connection. To assess performance relative to an off-the-shelf database, we tested basic read and write performance versus the POET commercial database on a typical target system (a Pentium 3/850 PC running Windows 2000). The benchmarks showed our database outperforming POET by about 10 percent on similar operations. This is attributable to POET functionality that is absent in our API specification.

It is important to note that the retrofit of the first generation Berkeley walkthrough functionality on the second generation substrate resulted in immediate and substantial gains in database load time. For example, when operating on the full Soda Hall model database (a 360 megabyte model), the load time decreased from several minutes to a fraction of a second. This "fast start"

ability is attributable to the smart pointer abstraction, which allowed the system to load objects on demand as the legacy visibility code needed them, rather than having to conservatively load large sections of the visibility structures before invoking visibility functions.

Chapter 4

Simulation Data Management and Control

4.1 Motivation

Evaluating the performance of architectural designs is one of the most interesting applications of virtual environments. The purpose of the first-generation Berkeley Walkthru system was to help evaluate how well the proposed structure of Soda Hall met its design criteria, and to allow aesthetic evaluation of the design. Projects such as the UNC walkthrough [40] and various virtual city projects [22, 41] have similar purposes, and are also geared towards evaluating architectural spaces. These evaluations require only the basic ability to have a user interactively "walk through" the space; there was no need for physical behavior in the model. The visual quality, size, and layout of the space can be evaluated without being able to move furniture or light fires.

The next logical step is to use the model to evaluate more interesting performance properties of the environment. For example, we could evaluate airflow, sound propagation, or radio transmissions and reflections through the structure. We could evaluate how well the structure performs in hazardous situations, such as earthquakes or fires. We could see how objects in the environment react to physical interactions between each other and users, or how large numbers of users interact with each other in the space.

Simulations such as these tend to fit well into the data model of a walkthrough system, in that they simulate the propagation of a physical effect through the space, similar to the way a traditional walkthrough simulates the propagation of light and visibility. Though these simulators

can describe more complex interactions, they are also strongly or completely constrained by the structure of the building. For example, all building-scale fire simulators operate on either a cell-and-portal or finite element model that maps closely or directly onto the spatial structure the walkthrough system builds for visibility purposes. This principle is true for all of these simulators; they all can make use of a spatial partition like the one the Berkeley Walkthru uses for visibility computation, database partitioning, and prefetching.

One application domain with a particularly high expected payoff is building design evaluation, where scientists, engineers, architects, and other professionals can evaluate a proposed or existing structure without handling or damaging physical materials. With such a system, users could preview architectural designs, evaluate their performance with various metrics, and do simulations and potentially destructive "what-if" experiments, such as fire safety studies, cheaply and with no risk. The visualization system can also contribute a more intuitive visualization of the results of complex simulations. Finally, a realistically reactive virtual environment could also be used for training of personnel for scenarios that are too dangerous, difficult, or expensive to simulate in real life.

It typically takes many years, a large budget, and a great deal of specialized expertise to author a simulation engine that provides high-quality, verifiably accurate output. It is unreasonable to assume that the planners of a new building or the authors of a virtual environment system will create these simulators on their own; they already have their hands full with interactive rendering and visualization of large environments in real time. Moreover, it would be highly uneconomical to ignore the huge body of tried and tested simulation code that already exists. Unfortunately, these simulators were not designed to work as interactive agents within a virtual world. It is thus very useful to provide a framework into which simulations can be tightly coupled into a virtual world model with a minimum of effort and of code rewriting. This approach leverages the work already done by physicists and engineers in designing the simulators with the latest approaches in computer graphics to provide a rich and productive user experience. Designing a general-purpose framework that allows these disparate systems to work well together is the challenge we are addressing in this portion of the work.

One driving application behind this work was to realize some of these advantages for the benefit of fire safety in architectural environments. Our initial target simulator was the National Institute of Standards and Technology's (NIST) Consolidated Model of Fire and Smoke Transport (CFAST) [6]. CFAST currently provides the world's most accurate real-time simulation of the impact of fire and its byproducts on a building environment. CFAST as a standalone package suffers

from unintuitive simulation setup and very rudimentary graphical output. Integrated into the walk-through system, it can provide real-time, intuitive, realistic, scientific visualization of building conditions in a fire hazard situation from the perspective of a person walking through a burning building. The viewer can observe the natural visual effects of flame and smoke in fire hazard conditions; alternatively, scientific visualization techniques allow the user to "observe" the concentrations of toxic compounds such as carbon monoxide and hydrogen cyanide in the air, as well as the temperatures of the atmosphere, walls, and floor. Warning and suppression systems such as smoke detectors and sprinkler heads can be observed in action to help determine their effectiveness. This technology can be used to improve fire safety by helping engineers and architects evaluate a building's potential safety and survivability through performance-based standards.

Throughout this chapter, we will be using the CFAST simulator as a running example to illustrate how a physical simulator that was not designed for a virtual environment is integrated into our framework.

4.2 Assumptions and Summary of Approach

4.2.1 Integration of Visualization and Simulation

We assume one or more simulators and a visualization client, each of which operates on a cell-and-portal style environment database. This database may be arbitrarily large, i.e., we could be operating on a building that will not fit into memory, and each of the two component systems can deal with the paging problem in its own way. However, due to occlusion, the visible "working set" of volumes will be tractable for any observer position. There is a mapping between the volumes of the visualization database and the simulation database, but the two are not be expected to be the same (i.e. a simulator "cell" might cover multiple visualizer "cells", or vice versa). Presently, we do not support arbitrarily complex geometric mappings between the two databases; we assume that one or more visualizer cells correspond to one simulator cell. We assume that the visualizer will transmit any setup information needed to begin simulation before issuing the start command. Furthermore, the visualizer may provide a front end by which the scenario being simulated may be changed on-the-fly. For example, the user may start a wastebasket fire in some room and then explore how the spread of the fire is influenced by opening or closing various doors or windows in the visualizer, thus repeatedly changing the situation being simulated. In such a case, the visualizer must transmit such an environment update to the simulator in real time, and the simulator must then recalculate

previously computed simulation results that are affected by the change, as well as alter the course of the simulation in progress. For simulators that do not explicitly support recomputation of parts of the data based on changes in the environment, we can stop the simulator engine, reset the internal state to the world state as stored in the simulation data set for the time the change was made, and restart the engine from that point. This solution is rather brute-force, as it requires complete recomputation of all conditions from that point forward. Hopefully in the future, simulators designed to work in interactive environments will directly support interactive modification without requiring discarding all simulation results beyond the time of the change.

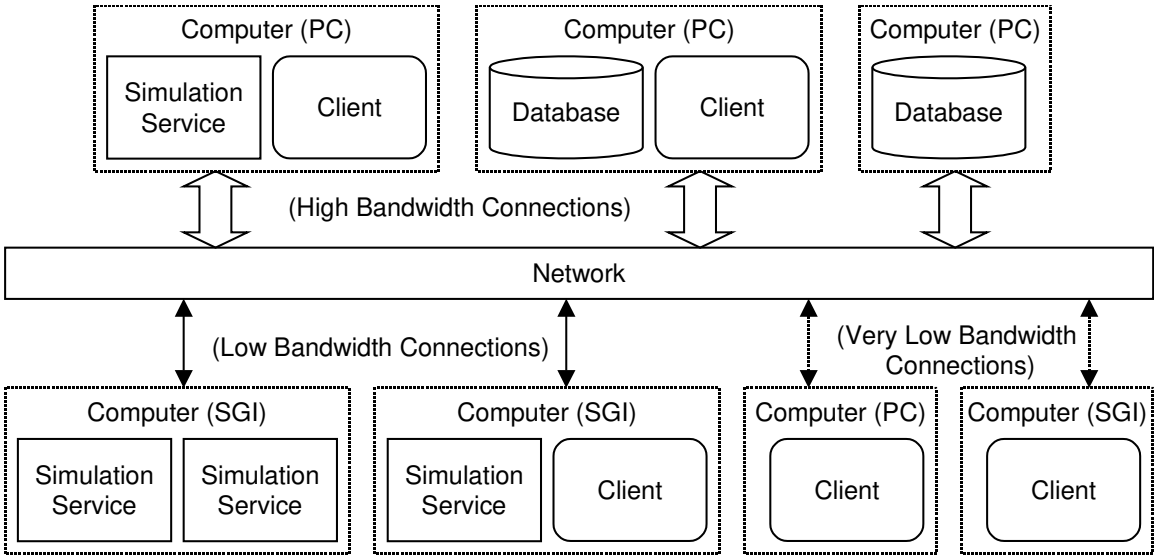


Figure 4.1: *Our model of the machines comprising the data network. Note that we assume there is only one limited-bandwidth physical link between machines, particularly to client machines, and that data and simulation servers can reside anywhere.*

Either one or both of the two component systems may be distributed, and may be operating on computers connected by anything from a LAN to a potentially high-latency, low-bandwidth network such as the Internet (Figure 4.1). We would also like to be able to attach and detach visualizers to a simulation in progress, to allow multiple observers to independently observe different portions of the data from the ongoing simulation. The simulator generates data about subsequent world states observing relevant dependencies. Most existing simulators operate with a fixed time step and produce results in time slices that span all volumes in the database. Since the viewer can only see a portion of the model in any given frame, only a subset of that information will be of relevance to the visualizer at any particular time (Figure 4.2). We refer to a discrete piece of simulated

information that is associated with one time slice and one spatial cell, a simulation "chunk." These chunks might be generated in different order depending on the demands of the visualizer.

The bottleneck in getting simulation data to the visualizer for rendering in real time may be in one of two places: either the simulator is too slow to generate data in real time, or the communication process between the simulator and visualizer has insufficient bandwidth to transmit the necessary chunks in a timely fashion. The simulation speed bottleneck is likely to hold for single-CPU simulations of reasonably sized databases; CFAST on a single 150MHz R4400 can only simulate about 16 cells (depending on degree of interconnection and density of furniture) in real time. Our goal in this situation is to increase the simulator's potential effectiveness by letting it know what areas of the world are of current interest to the visualizer. Specifically, the visualizer will inform the simulator of the currently visible cells and of the cells that may become visible in the very near future. The simulator can then concentrate on calculating and shipping the corresponding chunks with higher priority. In the future, we expect simulator technology to improve; simulators will become faster, and their designs will evolve to provide better support for interactive visualization. Recent work has shown that this can be a promising approach for modeling the dynamics of physical structures [42].

For the case where communication bandwidth is the bottleneck, the framework provides mechanisms that are easy to use and that optimally exploit the available bandwidth, while hiding communications concerns from the simulation designer. Of course, it is not possible to guarantee that all needed simulation chunks will be at the visualizer in time: the user might jump to a different part of the building or suddenly advance the time slider far into the future. To minimize the visible discontinuities associated with such a switch, we use a "just-in-time" chunk transmission scheme (Figure 4.3). Our scheme keeps the communication channel in a state of near-starvation, allowing unanticipated "emergency" chunks to be sent through a nearly-empty transmission queue. This approach minimizes latency in a time-critical situation, while still transmitting chunks at the highest possible rate for the channel.

4.2.2 Walkthru as a Model Client Environment

The Berkeley Walkthru program was designed to support real-time interactive visualization of large (several million polygons), densely occluded building models at interactive frame rates (greater than 10 frames per second). To accomplish this goal, the Walkthru subdivides the "world" into rectilinear *cells*, connected by *portals*. In a preprocessing step, the system associates with each

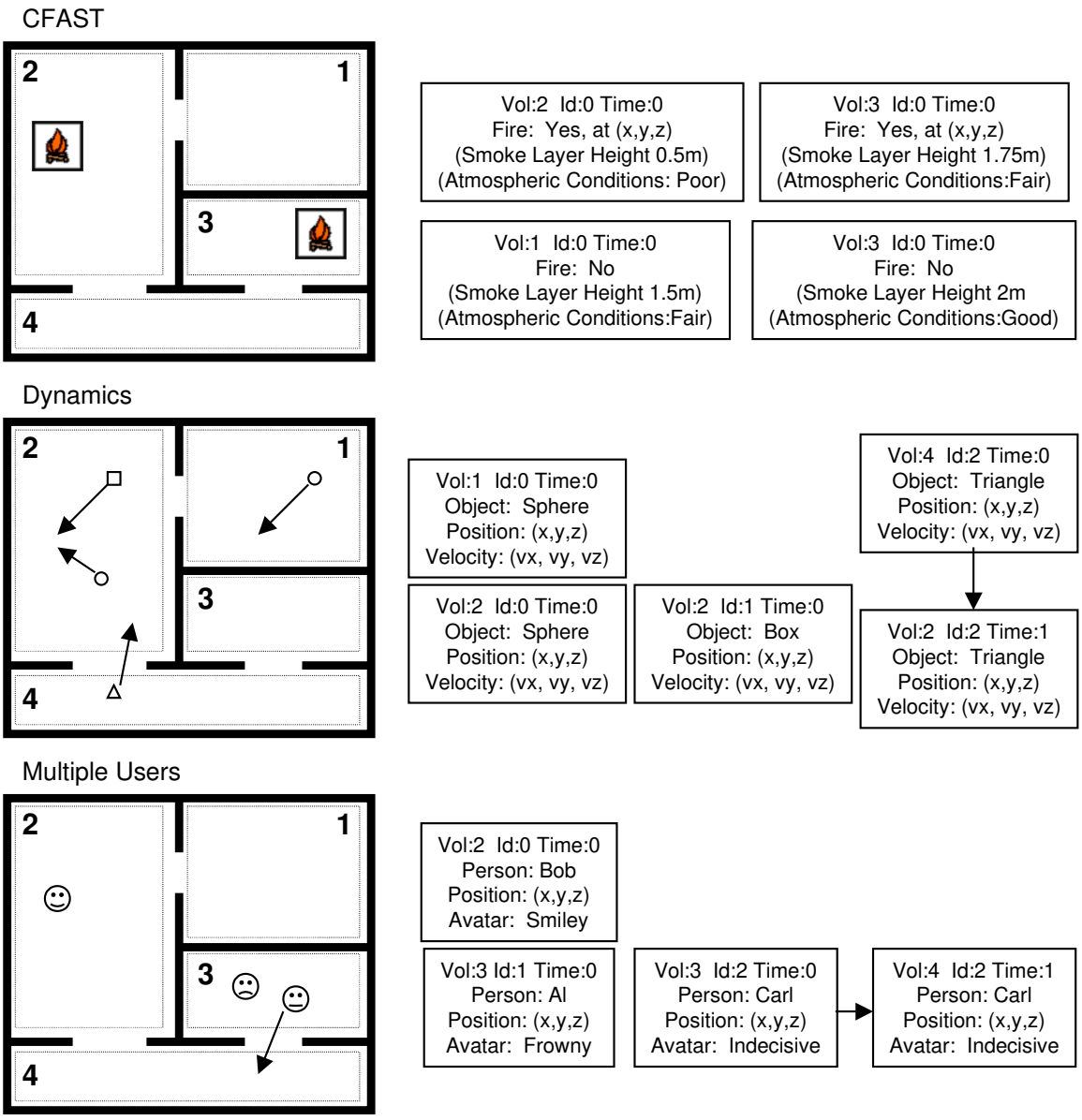


Figure 4.2: *The output of many types of simulation can be grouped by the same volumes that partition the model for visibility. If this is the case, the viewer only needs to receive simulation data for the volumes they can see.*

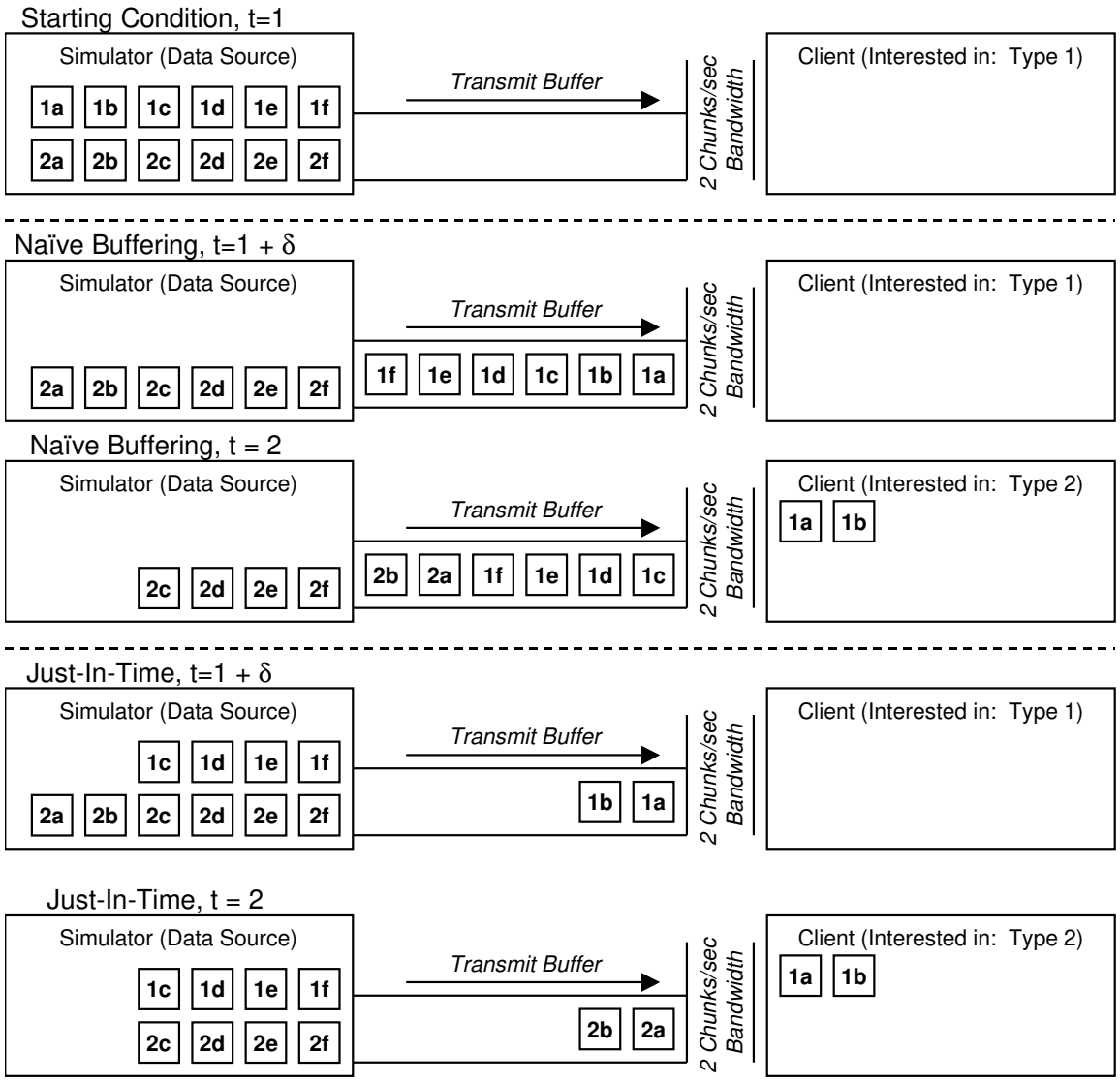


Figure 4.3: *The just-in-time concept provides minimal latency by sending only just as much information at each timestep as the client can receive. Traditional buffering can result in large backups of data that result in high latency if the viewer suddenly needs different information while the buffers are still packed.*

cell the set of all other cells that can be seen by an observer from any point within that cell. From this information, plus constraints on how quickly the observer can move through the database, the Walkthru can compute, for each frame, a set of cells that tightly, but conservatively bound the set of cells visible in the next few frames. There are only two types of object in the Walkthru: "major occluders," which are two-dimensional wall, ceiling, or floor polygons, whose planes define cell boundaries; and "detail objects," which are 3D models of building contents (such as furniture and light fixtures), and which are associated with the cells that intersect the object's bounding box. During each frame, the detail objects and major occluders incident to any visible cell are drawn, and visibility is reevaluated from the new position. If the user wishes to voluntarily disallow changes in major occluders by the database editor and any in-use simulators during a visualization run, many visibility relationships can be precomputed for the database. Otherwise, the update rate of the visibility computations is still quick enough to support relatively small-scale changes in the visibility structure of the world (i.e. punching some new holes in walls, or opening a new shaft in the floor or ceiling). In the last few years, Walkthru has provided a testbed for several applications including database construction [37], large scale radiosity computation [43], and scalable distributed walkthroughs with up to thousands of simultaneous users [5].

4.2.3 Simulation Types

Although all simulations that we are interested in integrating into this framework share the property that their results can be partitioned spatially into chunks that tile the environment, there are two distinct ways they treat time. Some simulators, such as fire or dynamics simulators, operate on a *virtual* time scale; that is, simulation time does not necessarily run one to one with real time. A user may wish to pause, rewind, play slowly, or jump around within the time axis defined by the simulation. Other simulators operate in *real time* rather than virtual time. This is defined as a simulator for which there is one "current" set of data across the entire set of volumes, and the user is uninterested in being able to look either ahead or behind the current time. The canonical example of a real-time simulator is the multiuser simulator; each client is interested in where all the avatars of visible users are at the moment, but is not interested in where they were in past times.

Real time can be treated as a special case of virtual time, where neither the user nor the system has any control over the visualization time; the visualization time moves in lockstep with real time. However, the distinction between virtual-time and real-time admits two optimizations that can be applied to data transmission for real-time simulators. First, the simulation data set may discard

any chunk for which the set has a more current chunk (i.e. "old" data may be discarded without worrying about the user moving the time slider backwards or putting it into reverse). Second, chunks for these types of simulation may be transmitted via a protocol that does not guarantee delivery or sequencing. Since we are only interested in the most recent world data, we can transmit updates continuously and not worry about getting acknowledgements of old data, since by the time the acknowledgement was received, the data would be out of date anyway.

4.2.4 Simulator Output

There are two basic ways for the simulation to produce output. First, it can modify the world database to which the clients are attached, and let the normal update mechanisms render the results. The second is to directly feed additional information to the clients through a connection that is external to the database (Figure 4.4).

Each approach has advantages and disadvantages, and is appropriate in different situations. Modifying the world database can be appropriate for real-time simulators that want to make persistent changes in the environment, such as lighting simulations; making regular editing-type modifications to the database is easy to code, and if the results are intended to be permanent anyway, it is a very appropriate mechanism to use. However, this adds a lot of latency to the process of transmitting results to the simulation client; a typical update cycle involves making the modification, having a notification propagate to the clients, and, if the notification message itself did not carry the necessary information for the update, having the clients request the changed information from the database. Each of these steps requires its own set of two-way network communication between the database server and a client process; thus, the entire process involves at least 3, and possibly 5, cycles of network transmissions and acknowledgements, compared with the single cycle needed when transmitting the results directly to the client. Furthermore, in this scenario all communications are routed through the database server, which is, most likely, already heavily loaded with visualization traffic.

Another issue with direct database modification is that of the virtual time axis. Many simulations do not make a single, permanent modification to the database; they create a time profile of conditions that the user wishes to interactively explore. None of the world databases we have used have any notion of time; the model world is an environment that changes in real time as the user modifies it, but there is no way to look into the past or branch users out into multiple temporal spaces, as you may want to do if you are doing comparison studies of different simulation scenarios

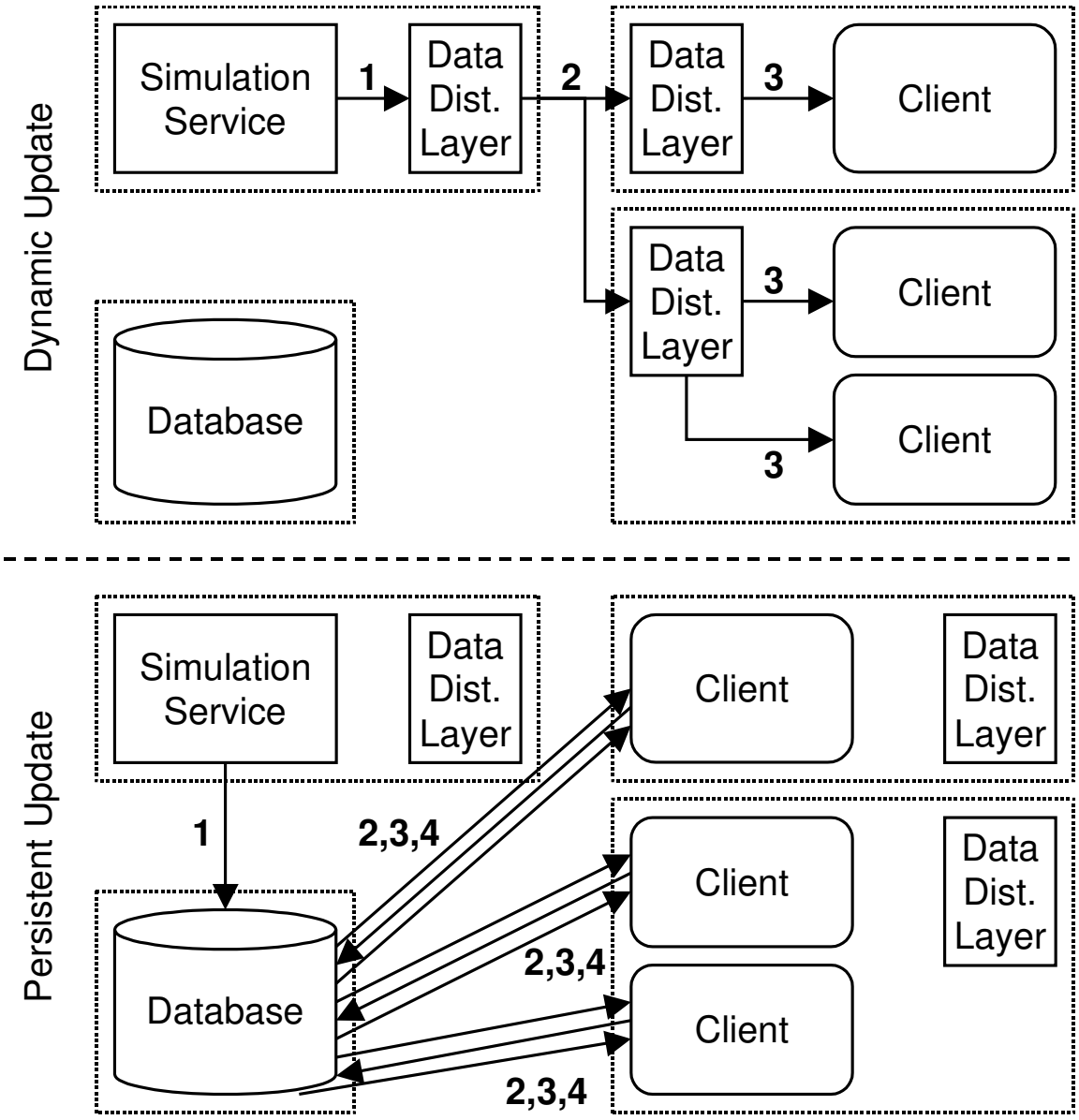


Figure 4.4: *Dynamic updates vs. persistent updates. Dotted borders represent machine boundaries; arrows between them require network communication. Dynamic updates are much faster, due to data sharing and minimal network communications; Persistent updates are easier to closely synchronize, via database locking and transactions, and do not require simulation-specific connections.*

within the same environment. In the case of a virtual time simulation, our data management system must serve as a temporal database of conditions evolving over many timelines at once, give the user the ability to control where their visualization is in virtual time, and provide efficient delivery mechanisms for data that can cull the output in time. The basic world database acts as a repository for elements that do not change; the simulation network acts as a database and delivery mechanism for deltas in world conditions over many disparate timelines.

4.3 Communication and Control

Communication and control between database servers, simulators, and visualization clients all go through a unified communication layer that provides more control over system communications than the raw socket abstraction does. This section describes those primitives.

4.3.1 Primitive Channels

The lowest level primitive of our communication model is the *channel*. This is a 2-way, buffered, asynchronous mechanism that can operate in either a nonblocking polling or an interrupt-driven mode. Each channel has one or more *bands*. Bands can be allocated on the fly by system components once the channel has been opened to another machine. Each band can be set up as either a sequenced, guaranteed communication line (based on Internet Transmission Control Protocol [TCP] [44]), or as a nonsequenced, nonguaranteed communication line (based on User Datagram Protocol [UDP] [45]). The former can be used for setup, control, and non-real-time components, while the latter can be used by real-time components that require the performance of UDP. The channel is instrumented to track the latency and bandwidth usage of the individual bands, so that client processes can estimate how much bandwidth is being used by each part of the system. Each band also has a bandwidth manager object associated with it; the manager may be instructed to limit the bandwidth available to the band to a given value, and can be instructed to issue a callback at a specified interval if there is "spare" bandwidth that has not been used in a given time period.

This abstraction has two major advantages. First, it provides a machine-independent interface to write the client processes against; this improves the portability of the system, as the channel is the only abstraction that needs to be rewritten for a different architecture. Second, the centralized channel object allows instrumentation and tracking of latency and bandwidth usage as well as the ability to apportion bandwidth to particular bands from a central location. Since multiple socket

connections between machines are always multiplexed onto a single physical line anyway, there is no loss of performance inherent in adding this abstraction layer to the socket connection, and the additional statistics and control it provides allows us to better balance communication resources between Citywalk components.

A general-purpose server class is provided that allows the process as a whole to open a server port on a machine and wait for connections, as well as to provide for services to be provided on each band. Channels can be opened locally or over a network; the appropriate low-level protocol is automatically selected by the system when it connects. Our typical server process allocates bands dedicated to both database service and simulation management services.

4.3.2 The Simulation Manager

The simulation manager is the central management thread that controls the simulation agents running in a particular process. Each running process in the network has one global instance of the simulation manager, which runs in its own thread. When a pair of processes are linked in order to provide for distributed simulations, the simulation managers establish two bands of communication within the channel connecting the two processes, one data band and one control band. The control band is used by the managers to execute high-level commands such as launching instances of simulations, establishing links between clients and running simulations, and describing which simulations are available and running on which machines in the network. The data band is where all actual communication between clients and simulators takes place, and is allocated as a TCP or UDP band depending on simulator type.

Once started, the simulation manager provides an indexing and launching service to any machines that are connected to the established network of managers. A client on any machine can request and receive lists of simulators that are available on any other machine in the network, as well as lists of actual running simulations on those machines. The client can then use that information to either remotely launch new simulations or to attach to running simulations in the same database space. Once a client is attached to a simulation, all communication between the client and that simulation is routed through the manager's data channel to the target machine.

This centralized architecture provides three important benefits over an architecture that allows simulators and clients to connect directly to each other. First, it allows monitoring and control of aggregate bandwidth between machines. Without such a centralized communication mechanism, there is no way to control or optimize how much data particular client-simulation pairs

are transmitting between each other; if there are many links in the network, the data may overload the available hardware link and add arbitrary amounts of latency to each band. Second, it allows for the elimination of redundancy in framework communication. For example, if a client on machine a is running three simulations s_1 through s_3 on machine b, those three simulations each need to know the visible and lookahead sets of the client. If the client had established three separate connections, it would have to communicate these sets three times, using three times the bandwidth. In our scheme, however, the client transmits the sets only once, and that information is distributed to each simulation by the manager. This works the other way as well; if two clients on one machine are accessing the same simulation on another machine, these client visible sets are unioned *before* they are transmitted. Thus, any overlap between the two clients' visible sets results in bandwidth savings over the channel in both directions; overlapping visible set entries are transmitted only once, and corresponding data chunks are only transmitted back to the clients' machine once. The latter savings can be substantial, particularly in simulations with large data sets; this scheme is much like the one used in Funkhouser's RING system [28]. Third, the system can launch and terminate simulations independently of the establishment of a channel between machines, and can index simulations begun by other clients on different machines.

The simulation manager itself has a view window that provides the controls for connecting to other simulation managers on other machines, displaying a catalog of simulations available and simulations that are running, launching new simulations, and connecting to running simulations. By entering an IP address and port, a new connection can be established to a server that serves one or more types of simulation or running simulations. The user is presented with a list of provided and running simulations and an index of the machines they are available on. From this list the user may select a simulation and connect to it with a particular type of view. This creates a local instance of a client and a view that are attached to the Walkthru view window. The view will automatically close the client connection if the view's UI window is closed.

4.3.3 Client to Simulator Communication

Control Traffic

Any client can communicate with the service asynchronously by handing command codes, raw packets, or packable objects to the interface on the client machine. This data is then encoded and transmitted through the communication subsystem at the first opportunity. These methods are primarily used for higher-level control of the simulation (for example, applying a user-specified

force during a physics simulation, or opening or closing a door during a fire simulation) and for the initial simulation setup phase to transmit case information to the service (for example, sending the trigger fire location when starting up a fire simulation). The server receives the data annotated with the ID of the client who sent it.

Client Telemetry

During a simulation run, the client must maintain an up-to-date record of which spacetime regions are of interest; without this information, the server cannot prioritize transmission of data to the client, as it cannot determine which chunks are most important. Thus, simulation clients maintain a data structure that is kept up-to-date on the server by a reversed version of the data distribution method used for real-time simulators. This is referred to as *client telemetry* (Figure 4.5). There are two default telemetry types; the first is used by real-time simulators, and consists of a currently visible volume set and a potentially visible set (e.g. set of volumes that may become visible in the near future if the user moves). The second type is actually a subclass of the first type; it adds a current viewing time and time velocity of interest. This second type is used by virtual time simulators, as they require knowledge of what the client visualization time is as well as their visible sets. Though it is straightforward to overload these telemetry classes to provide additional information, only one of our simulators currently takes advantage of this; that being the multiuser simulator, which adds the currently selected avatar model ID and the client's current location and velocity to the client telemetry object.

When a client changes its telemetry object (e.g. when the visible set changes, or the VCR control is manipulated), the framework transmits an update to the server. This allows the service to access a current local image of the telemetry object for each client attached to it.

Telemetry objects can also define a merge function, which allows interest regions to be combined at internal nodes in the network (Figure 4.6). This increases efficiency by allowing the service to only process volumes in overlap regions once. Telemetry can also be merged across different simulators if the telemetry objects are of compatible types; thus, only one set of telemetry is necessary for a given visualization client, even though more than one simulator may be running on that client. This further reduces bandwidth usage and increases efficiency at internal network nodes.

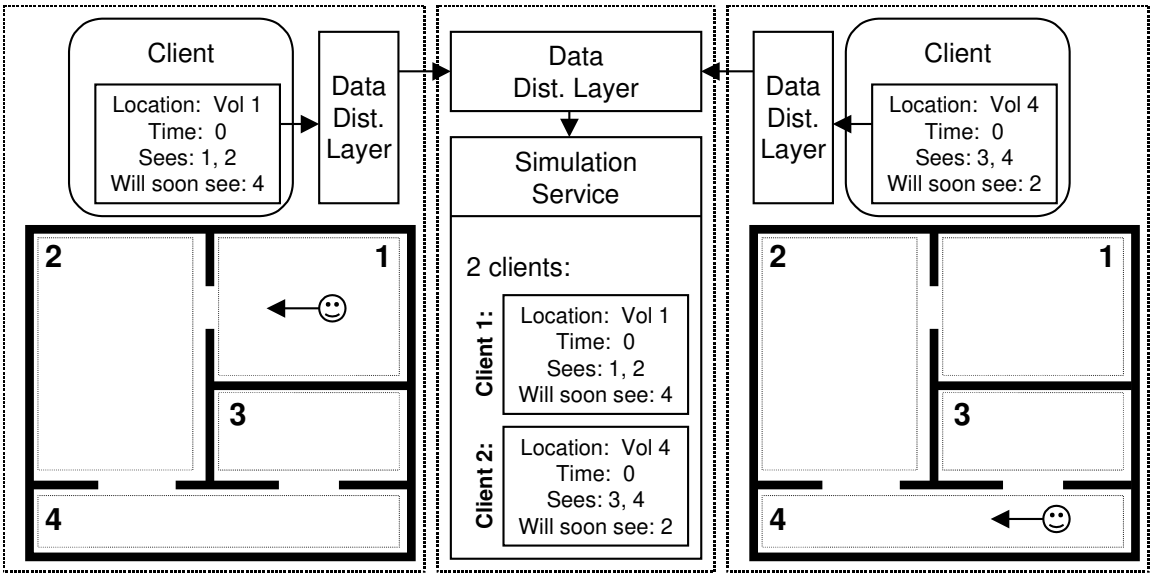


Figure 4.5: *Telemetry objects provide the simulation server with knowledge of what interest regions the client is exploring.*

4.3.4 Simulator to Client Communication

General Communication

There are three mechanisms for the server to transfer information to its clients. The first two are indirect and are handled automatically by the framework; that is, communication via changing the database directly (which is propagated to clients via watches), and communication via the real-time data distribution subsystem (discussed in the remainder of this section). The third type is direct transmission of packets. The server interface provides functions to transmit data packets or packable objects to a specific client (by client ID) or broadcast to all current clients. This type of manual communication is typically used for "one shot" data such as the chat channel for the multiuser service (which receives lines of text from the clients and rebroadcasts them to everyone currently in range of the sender).

Real-Time Data: The Simulation Data Set

In order to provide efficient data exchange between simulator and visualizer, we need a general structure for simulation data that can be easily managed and which is flexible enough to accommodate any information that a particular simulator may want to convey to the visualizer. This structure, called the *simulation data set*, is designed to hold and index simulation results from one

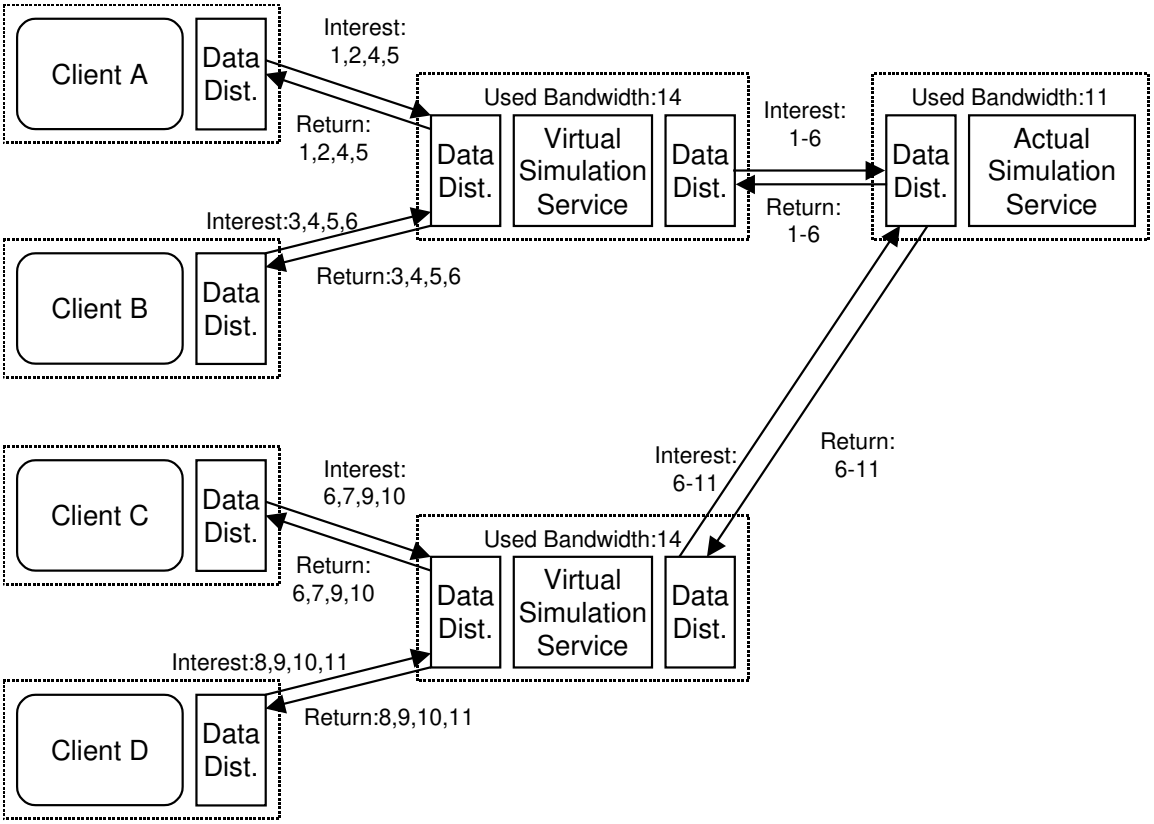


Figure 4.6: *Intermediate nodes can merge telemetry nodes to improve bandwidth usage within the service network.*

run of one simulator.

The data nodes being indexed are the aforementioned "simulation chunks;" they are variable-size data structures that represent part of the simulation output for a particular volume at a particular simulation time. The structure of a chunk is user-definable, so it can be easily modified to accommodate different simulator models. Chunks are subclassed from the database-capable base class defined in the second-generation Walkthrough database subsystem, and implement the packing API; thus, they are machine-independent and can be trivially transmitted across the network or stored in a persistent database by the client. Chunks are indexed by three major values, all three of which must be defined for each chunk: the volume tag, the subvolume tag, and the time value.

The simulation data set assumes that the world is partitioned into a set of volumes that are understood by both the visualizer and the simulator. The volumes are assigned integer tags by the designer. The framework uses these tags to identify data for a particular volume, but makes no assumptions about the spatial relationship of the tags to each other; that meaning is determined by the simulator and client. For example, in the case of the fire simulator, the tags are 1 through n, corresponding to the set of n rooms in the CFAST input data file. The Walkthru maintains a mapping from these tags to 3D bounding boxes in the building model's coordinate system; these bounding boxes allow mapping from a 3D offset into a CFAST room to a 3D coordinate and cell pointer in the Walkthru model's coordinate system. Note that one simulator "room" may map to more than one Walkthru cell, or vice versa. Another example is the dynamics simulator, which uses the KD-tree tags as the volume tags. This tag set has the advantage of being "native" to the walkthrough; each tag corresponds to exactly one visibility cell, and can be found quickly in the KD tree. The volume tags are used by the framework to identify visible and soon-to-be visible volumes. During a run of the visualizer, the framework extracts visibility information from the Walkthru's culling engine to determine which volume tags represent volumes that are visible in the current frame, and which represent volumes that may be visible within a certain lookahead time. Because the system has a known mapping between simulator volume IDs and Walkthru cells, the visualizer can transmit desired simulation time and cell visibility information to the simulator, allowing the latter to determine exactly which chunks are needed for rendering both the current frame and near-future frames. The fact that these tags are opaque to the framework make them useful for many kinds of simulators rather than being tailored to a specific simulation.

In addition to the volume tag, the designer may also define a subvolume tag. The subvolume tag is another opaque, user-defined integer value that is only used by the framework as a searchable subkey for data indexing. It is not interpreted in any way, so it can be used by the de-

signer to represent any axis of data they wish. For example, in the CFAST interface, subvolume tag 0 within a volume is used to store the basic information about the conditions in the volume for each of the two gas layers; the temperature, the pressure, and so forth. Subvolume tags 1 through n, where n is the number of pieces of furniture in the room, describe the burn state of each of these pieces of furniture. If a piece of furniture is not burning in a volume at a particular time, there will be no data chunk corresponsing to that {volume, furniture ID} pair at that timestep. For the multiuser simulator, on the other hand, these subvolume tags correspond to the user IDs of the users logged into the virtual environment. If a particular user is in a volume, that volume will contain a chunk tagged with a subvolume ID equal to that user's ID, which contains information about the position, velocity, and avatar status of that user (Figure 4.7).

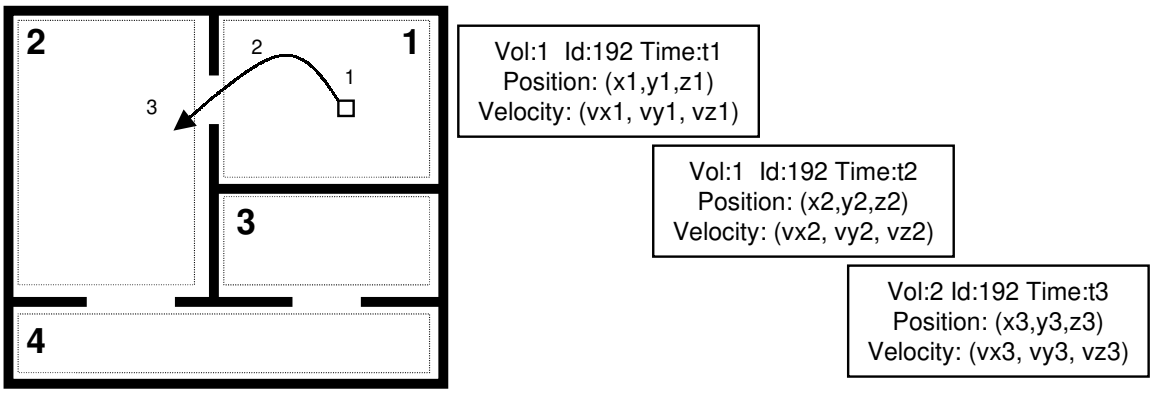


Figure 4.7: *An example of the set of chunks generated over time in the dynamics simulator. The chunks generated for a particular object, which can be located in the condition set by the objects' unique ID in the subvolume field of the chunk, can show changing internal data over time, and change volume as the object moves between cells.*

The three keys (volume, subvolume, and time) are used within the simulation data set to index the chunks in two ways. The first index uses the keys in the order: *volume, time, subvolume*. the second index uses the keys in the order: *time, volume, subvolume*. These two indices are necessary to support the two types of query that are most often asked of the simulation data set:

1. Given a particular volume at a particular time, produce the set of chunks that correspond to that volume at that time. This is asked by the visualizer to retrieve the set of chunks that must be drawn for each visible volume in the current frame.

2. Given a particular time, produce the set of chunks that are defined for a particular set of volumes within that time. This is asked by the framework to retrieve the set of chunks that

are most urgently needed for visualization, given the current visible and lookahead sets and the current simulation time being requested by the client.

As an example, consider simulation chunks from the fire simulator, which come in two types. The first type contains temperature, energy output, location, and fuel conversion rate of one particular fire; there can be many of these in one spacetime volume, corresponding to active fires from individual fuel sources such as pieces of furniture. The second type, of which there is only one per spacetime volume, contains the chemical concentrations of nine different gases, fuel concentration, atmospheric pressure, toxicity level, and smoke interface height for the volume as a whole.

Note that parallelizeable simulations would work very well within this data model, since the sets of data chunks generated by the separate simulator threads are easily recombined via simple unions of their simulation data sets. Furthermore, since the thread that is controlling the simulation knows how the problem is distributed, it should also be able to appropriately distribute the visibility lookahead data provided by the simulation manager.

4.4 Real-Time Data Management

4.4.1 "Just-In-Time" Simulation Data Management

In order for the visualization manager to ensure that the appropriate simulation chunks are either already present or en route from the simulation machine, it has to provide the remote simulation manager with enough information to determine which chunks are most critically needed. To do this, we define an "importance function" over spacetime, in which the chunks associated with spacetime cells of higher importance will be transmitted to the visualizer earlier. Clearly, the spacetime cells that are visible to the user at the current visualization time are the most important ones, and are needed immediately by the visualizer. Given the user's location, maximum velocities in space and time, the current visualization time, the current visualization time velocity, and the preprocessed volume visibility information from the viewer's cull process, we can compute for each spacetime cell the earliest real time in the future in which the user might be able to see that cell. This defines the desired function; smaller "earliest-possible-time-to-visibility" values correspond to higher importance. The information needed to compute this function is available to the visualization manager, which is directly linked to the visualizer; one of the visualization manager's tasks is to transmit this information to the simulation manager, which evaluates the importance function over

the set of chunks generated by the simulator, and thereby determines which unsent chunks are most important at any given time.

Our current system does not support the full computation of this function. We implement a heuristic approximation by maintaining a *visibility set* and an up-to-date visualization time at the simulation manager. The visibility set contains the set of Walkthru cells that are either currently visible to the observer, or may become visible in the next several frames. This information is normally computed as part of a Walkthru frame. The visualization manager monitors the visibility set and transmits an update to the simulation manager when the set changes between frames. Similarly, the visualization time and time velocity are updated when the user alters the time velocity or moves the time slider. Note that, though the visualization time changes as real time passes, the simulation manager can keep accurate track of the current visualization time without continuous updates from the visualization manager; updates are only necessary if the user manipulates a control setting.

The simulation manager then assigns highest importance to the transmission of chunks that are in the visible set and whose time is closest to the current visualization time in the direction of the current time velocity. The next highest importance is assigned to chunks in the visibility set in the *opposite* direction of the current time velocity, since the user often wants to review preceding time slices in the current location to find out how the situation has evolved. All other chunks are of tertiary importance. This corresponds to an approximation of the "ideal" importance function discussed above for very high values of time velocity; it can be computed quickly and does not require the full visibility information of the Walkthru's visibility processing. The simulation manager uses the communications channel to transmit those chunks that have not already been sent and are of highest importance as denoted by the heuristic function (Figure 4.8).

A sudden change in the time, time velocity, or visibility set can result in a need to get a new set of chunks to the visualizer as quickly as possible. If the user has been visualizing simulation time $t_s = 10$, for example, and the time slider is moved to $t_s = 200$, the simulation manager may have this data, but it is unlikely that the data has been transmitted already. In this case, the simulation manager *immediately* evaluates the most critical chunks to be sent to the visualizer, and transmits those chunks as soon as possible.

It is interesting to note that limiting the user's maximum "time acceleration" (i.e. disallowing direct manipulation of the time slider, allowing the user to move in time only with the VCR buttons) has the effect of allowing us to compute a "time lookahead" to go along with the visibility lookahead. This means that we can establish a tight superset of the number of spacetime chunks that might be visible in the next few seconds of real time. Without such a bound, the potentially

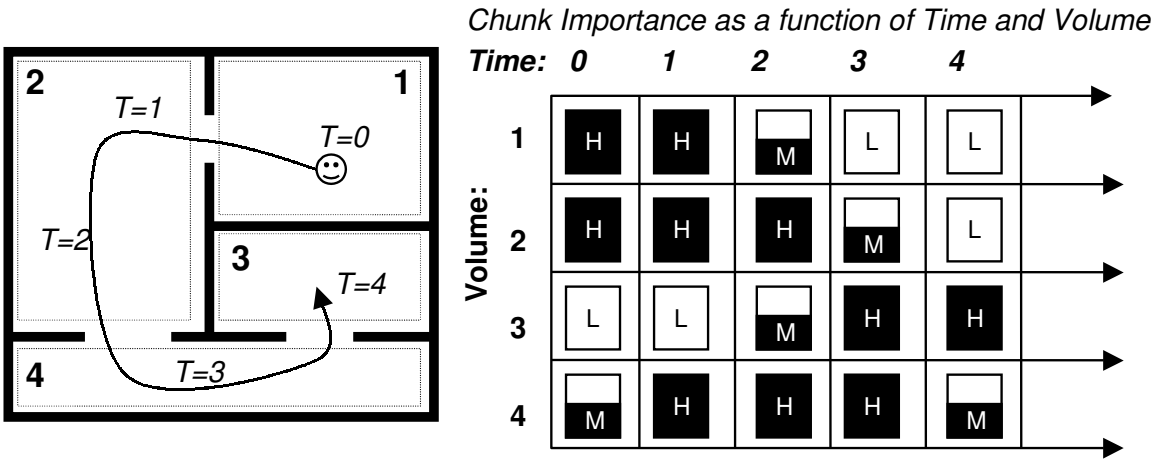


Figure 4.8: *Chunk importance is based on proximity to the user's immediate interest and near-future lookahead interest sets.*

visible set from one frame to the next includes the set of potentially visible cells for *all* timeslices of simulation data available, because the user can drag the time slider from any point to any point within one frame time. With such a bound, and a bound on the number of chunks that will be submitted per spacetime volume (which is easy to derive for most simulators, including CFAST), we can compute a minimum required bandwidth so that we can *guarantee* that all of the needed chunks will be available if there has been at least enough time since the chunk's submission to overcome the latency of the communication channel.

If memory is limited on the visualization machine, it is possible for our system to run the visualization manager as a *cache*, rather than as an *accumulator* of the entire simulation data set. In this case, the visualization manager is allowed to "throw out" old or not recently used chunks. The visualization manager reports to the simulation manager which chunks have been discarded, so that they may be retransmitted if they need to be viewed again,. In the case of very large precomputed data sets, the simulation manager can also be run on a local machine, managing access to a huge disk file instead of an active simulation, while the visualization manager manages the set of simulation data being cached in memory.

4.4.2 Bandwidth Management

Given only the importance function on the set of simulation chunks that have been submitted, there is no indication of *how much* data should be sent by the simulation manager per unit time. Because the channel is buffered, if no bandwidth usage constraint is enforced, then every time

some conditions are submitted by the simulator, all of that data could be queued for transmission through a channel that will not be able to actually finish transmitting that data for quite some time. In a priority situation, when the importance function has changed due to user input, and a different set of chunks are needed *immediately*, queued "old" chunks would delay the transmission of urgent data until those older chunks had drained through the pipe. This "clogging" reduces or eliminates the system's ability to respond to sudden changes in visibility or time. Unfortunately, with most physical simulators, this situation would occur fairly often; physical simulators, including CFAST, tend to exhibit "bursty" output, corresponding to sets of solutions for conditions across a slice of time for the entire model. If we use our importance function to determine which chunks are to be sent, the situation becomes even worse; sudden changes of the user's time or position generate even larger spikes, as new, potentially huge sets of chunks become highly important when the user walks into a new region of the database.

An early solution we tried for this problem is to include a priority bypass that provides the ability to interrupt the channel's normal input queue with a second queue of chunks that are to be transmitted first. In an interactive system, this priority bypass often proves ineffective, due to the fact that *two* of the aforementioned sudden changes in the importance function could cause the system to send priority data down an already busy priority channel, and the more recent priority packets, which are now more critical, are delayed in the same fashion that the one-channel strategy delays the first set of priority packets. The situation is made worse in larger databases; in the unmanaged condition the size of these spikes grows with the size of the database. Adding bypasses on top of bypasses quickly becomes unwieldy and inefficient; once all of the data is sitting in multiply-bypassed queues, control of transmission order becomes impossible, the amount of storage needed for redundant queues quickly becomes prohibitive, and the work needed to override a chunk that has been regenerated by the simulator grows without bound.

The core of the problem is the inherent buffering of data in the communication channel. This buffering is unavoidable due to its ubiquity in the low-level communication structures provided by the operating system and the network itself, which use buffering to optimize throughput. Unfortunately, the more buffering there is in the channel, the larger the potential latency for a high-priority packet to be transmitted through the channel; since guaranteed-receipt network protocols guarantee arrival in order of transmission, every bit of buffered data in the channel must clear the channel before our high-priority packet can get through. Thus, we would like to operate the channel in a near-starvation mode, which simultaneously minimizes buffering while using all or nearly all of the bandwidth to transmit useful chunks as quickly as possible. This job is handled by our *bandwidth*

manager, which closely controls the speed at which the simulation manager is allowed to transmit chunks to the visualization manager. Available bandwidth is currently specified to the bandwidth manager in total kilobytes per second (kB/s). The bandwidth number should be selected to closely approximate real bandwidth (i.e. on two machines on an Ethernet, bandwidth might be on the order of 1 MB/s, whereas two machines connected by 28.8 kb/s modem would only be able to manage about 3 kB/s). Several times a second, the bandwidth manager "wakes up" and gives the simulation manager permission to transmit another x kB worth of simulation chunks on the data stream, where x is the given bandwidth divided by the manager's wakeup frequency (typically 5 to 10 Hz). When this happens, the simulation manager selects x kB worth of chunks from the unsent chunk pool in order of importance, and gives those chunks to the channel for immediate transmission. By the time the manager wakes up again, all of the submitted chunks should have cleared or nearly cleared the channel; thus, if an emergency situation happens while the manager is asleep, when the manager next wakes up, the most important chunks will be transmitted on a nearly empty channel, which minimizes the transmission latency for those chunks. At the same time, if no emergency occurs, the channel is still being utilized at nearly its maximum capacity, with the next most important set of chunks being sent "just-in-time" for the channel to have completed transmitting the last set; clogging cannot occur if the bandwidth estimate is accurate or conservative.

In our current system, the bandwidth manager's settings are provided by the user. This is generally effective since the user knows how "wide" their connection to the server is (modem, LAN, etc., and the speed in MB/s). Multimedia video on demand systems faced with a similar problem have shown that the system can determine the speed of the connection in real time, and scale the system's notion of available bandwidth appropriately [46]. This approach is also applicable in our framework, and should respond even better to changes in network conditions; we have not implemented this approach, since we have been satisfied with the performance of a fixed allowance.

4.4.3 Performance Analysis

We make the following assumptions:

There is a simulator generating b bytes per spacetime chunk over s spaces at a rate of n times real time (i.e. n seconds of data are generated across all volumes per second of real time). The visible plus lookahead set is of size s_v, $s_v << s$. The channel bandwidth is B bytes per second.

Given that $B < snb$, in the naive first-in-first-out case, data will accumulate behind the buffer at a rate of $snb - B$ per second. Since this method is unresponsive to user input, the viewer

will progressively lag further and further behind the current state of the simulation. Ironically this may not be apparent to the user unless the system is designed to display the difference between the time that the simulator is "currently" working on versus the largest simulation time in the local cache.

In the case where we transmit the data based on visibility, but still do not account for channel bandwidth, the accumulation rate of data in the queue is $B - s_v nb$, which may be zero if the visible set is small enough. However, the moment that the viewer moves such that s_v is changed to s_v^1, the system must begin transmitting the data for the missing volumes. If that data is again transmitted in a first-simulated-first-transmitted sense, we see an immediate insertion of $(|s_v^1 - s_v|nb)t_s$ bytes into the channel, where t_s is the current simulation time. This data must be removed from the queue before further changes in s_v can be processed. Furthermore, the data needed for viewing is not immediately present in the pipeline; it must wait for (1) the remainder of the data for the unused portion of s_v, $(s_v nb)(t_s - t_v)$, plus (2) the information from $t = 0$ to $t = t_v$ for the newly visible regions, $(|s_v^1 - s_v|nb)t_v$, to clear the channel before the needed data is present for the current frame. Additional changes in s_v during the transmission of this block of data will be further delayed by the remainder of this block that is left in the channel when the change in s occurs.

Finally, consider a case where we partition available bandwidth into small, fixed size timeslices. Within each timeslice $t_t s$, we have $Bt_t s$ bytes of bandwidth available. At the beginning of each timeslice, we determine what will be most important to transmit to the client over the next $t_t s$ seconds, and queue that information for transmission. Thus, assuming that we have up-to-date information about the client's most critically needed information, and assuming that we have enough bandwidth to transmit that information, we can guarantee that the client always has the needed data regardless of viewer motion over time. In the case that the client's needs change suddenly, and assuming it takes $t_c n$ time for the client to notify the server of the changed conditions, we have a worst-case maximum latency of $t_c n + t_t s + 2l + s_v nb$, where l is the latency inherent in the socket, between the establishment of a completely new visible set and the completion of transmission of the data necessary to support that new set. Note that, because of the just-in-time nature of the algorithm, there is never any unneeded backup in the socket past that of the last timeslice, or $Bt_t s$ bytes. Thus, this maximum latency holds regardless of how quickly the viewer's position and visible set changes, and the socket is guaranteed to be transmitting useful data within $t_c n + l + t_t s$ of any change in the interest set.

4.4.4 Results

Figures 4.9 through 4.11 show a typical example and comparison of the performance of three strategies for data management. The most basic is the naive, *oldest-data-first* strategy (figure 4.9A) which simply queues timeslice data into the communication channel as it becomes available. The second is the *visibility-guided* strategy (figure 4.9B), in which simulation data is transmitted only for visible or almost-visible volumes (i.e. in order of the basic heuristic importance function), but with no bandwidth management, so that it queues *all* unsent available data for the visible volume set after a change in visualization time or the visible set. The third strategy is our full *bandwidth-managed-importance* strategy (figure 4.9C), which incorporates all of the subsystems mentioned in this paper. The data was gathered from our instrumented RTC package during identical prerecorded runs of both the visualizer and simulator, in which all simulation data generation, user motion, and manipulation of the time slider and VCR controls were recorded and reproduced in exactly the same way for each run. The communication bandwidth was artificially reduced to 3 kB/s for these runs in order to demonstrate the difference between the strategies; at present, our largest test case is insufficient to stress the switched Ethernet in our office. The reader may wish to note that this bandwidth was selected to correspond to that available from a 28.8 kb/s modem link.

The three graphs in figure 4.9 compare three functions of real time for each strategy: how far the simulator has progressed through the simulation, labeled *Simulated Up To*; the latest simulation time for which simulation data has actually been sent to the visualizer (i.e. the maximum possible viewable simulation time at the visualizer), labeled *Max. Simulation Time Transmitted*, and the simulation time currently being requested by the user within the Walkthru, labeled *Requested Visualization Time*. If bandwidth were infinite, the user should be able to "see" simulation results whenever the requested visualization time is less than the simulated-up-to time, and the maximum simulation time transmitted would be identical to the simulated-up-to time (which is the most recent simulation time available from the simulator). Under bandwidth limitations, however, it may be that the requested visualization time is less than the simulated-up-to time, but the data is not yet available (i.e. the maximum simulation time transmitted is *less* than the requested visualization time) due to failure of the communication channel to transport the needed data. The most visible evidence of this in the graphs is where the requested visualization time becomes a horizontal line, indicating that the autopause mechanism has engaged due to the visualizer not having the needed data (resulting in a zero time velocity and unchanging visualization time). Many such flats are seen in the case of the *oldest-data-first* strategy; the channel is far too narrow at 3 kB/s to transmit the data in time.

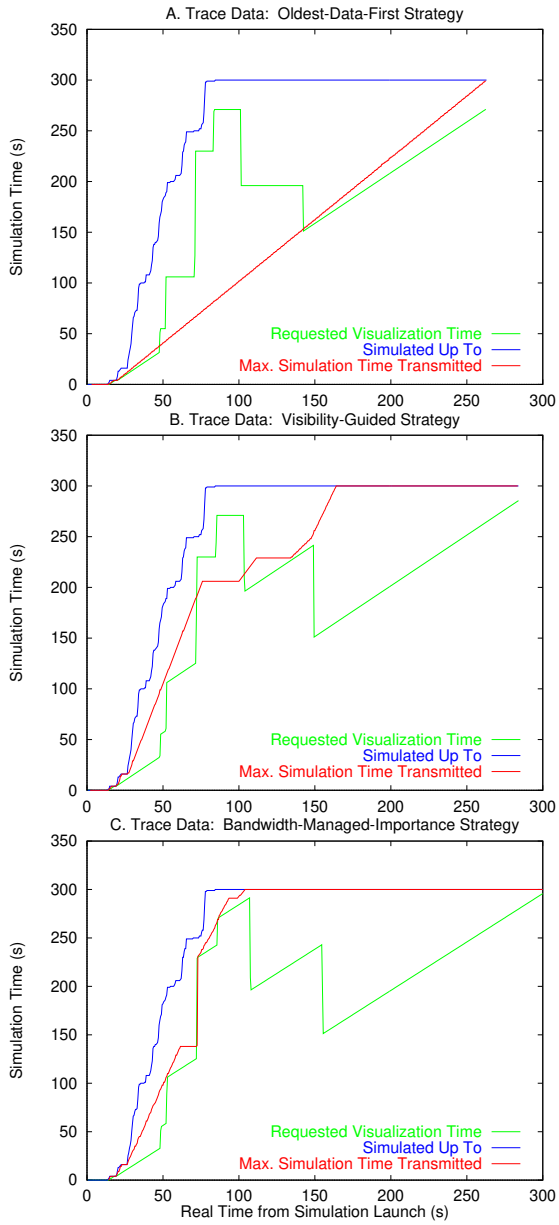


Figure 4.9: *Trace data of simulator-visualizer data transfer for three strategies: the oldest-data-first strategy (top), the visibility-guided strategy (middle), and the bandwidth-managed-importance strategy (bottom). The horizontal axis is real time; the vertical axis is simulation (i.e. virtual) time. Three functions are plotted for each strategy: the amount of simulation time completed by the simulator, the viewer's current visualization time, and the timestamp of the latest chunk that has been transmitted from simulator to visualizer. Note the vertical lines in the requested visualization time, which denote user-created time discontinuities, and the horizontal lines in the requested visualization time, which show regimes for which data is available from the simulator, but for which that data had not been transmitted in time to be viewed. The "maximum simulation time transmitted" curves give an indication of how responsive each strategy is to user movement in space and time.*

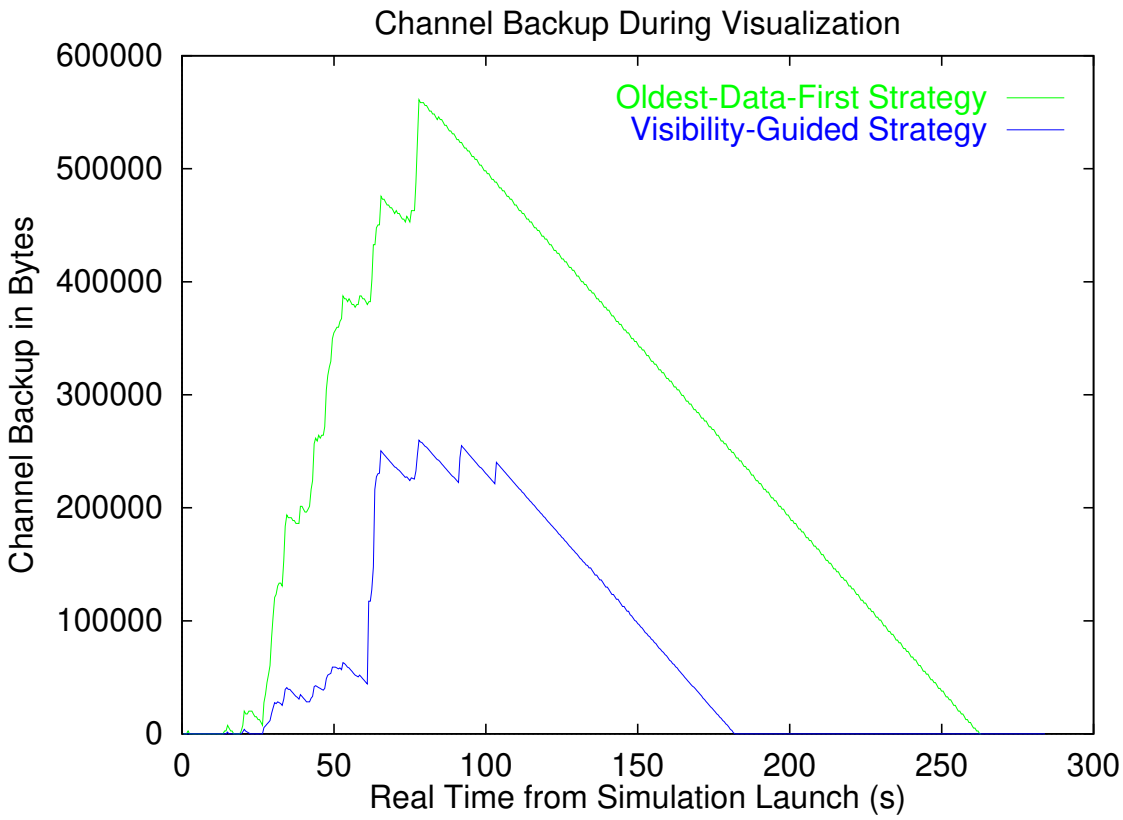


Figure 4.10: *Trace data of communication pipe backup (i.e. clogging) for the oldest-data-first and visibility-guided strategies. The former is much worse than the latter, although it is in the latter that it actually makes a difference. Pipe blockage in the bandwidth-managed-importance case is negligible (less than 1.5 kB/s on this graph, where the others peak at about 550 kB/s and 250 kB/s respectively), and can in fact be reduced to an arbitrarily small amount on a fast computer by increasing the manager's callback frequency.*

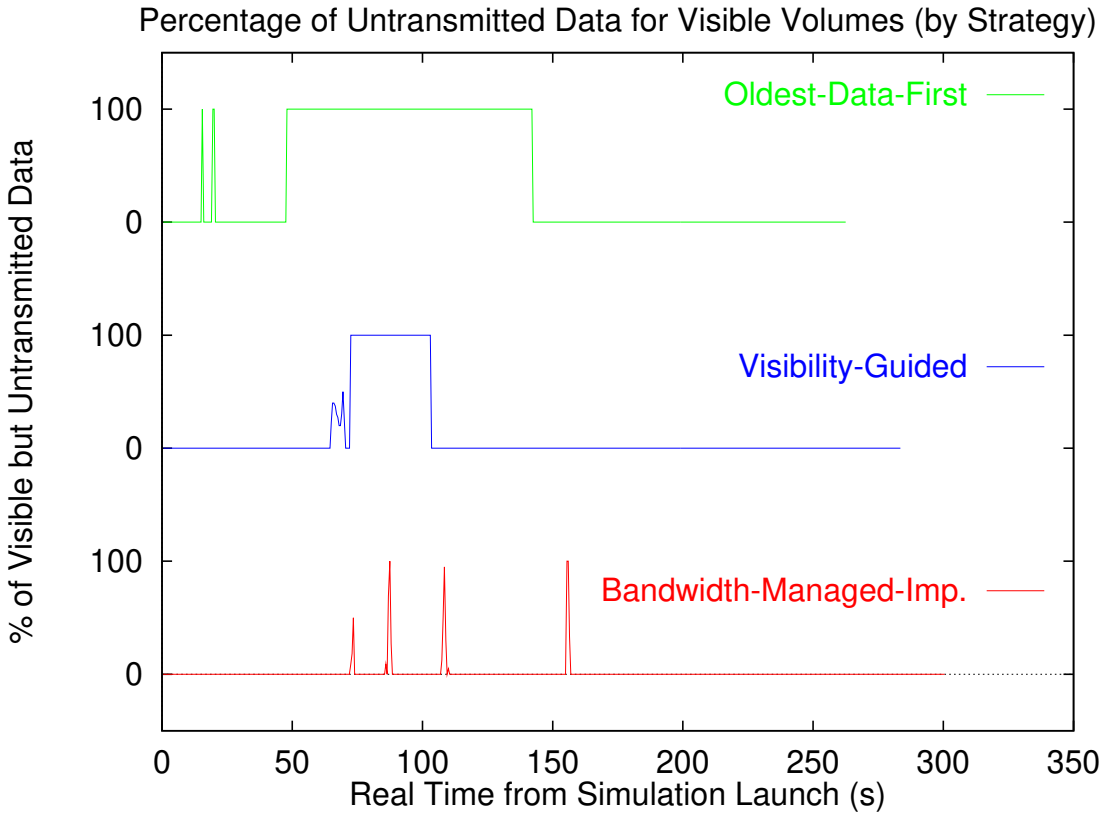


Figure 4.11: *Trace data of the percentage of spacetime volumes visible to the user that have simulation data available, but for which that data has not yet been transmitted to the visualizer. The oldest-data-first strategy exhibits massive gaps in viewable data; the visibility-guided strategy fares better, but there is still a 40-second period where the user should be seeing smoke and flame, but instead sees nothing. The bandwidth-managed-importance case shows only brief 1- to 2-second gaps at time discontinuities (i.e. where the user unpredictably drags the time slider far into the untransmitted data).*

The *visibility-guided* strategy does better toward the beginning of the run, where the fact that few volumes are visible allows it to get more timesteps to the visualizer, since those steps contain fewer volumes. However, when the user walks out into the hallway at time 70, the new set of visible volumes results in a deluge of newly important data being queued into the channel. When the user proceeds to move the time slider at time 83, the channel is clogged with the (as yet unsent, but already obsolete) "priority" data from the hallway transition, and the system is unable to respond, resulting in a period of no data being visible. A graph of the communication channel blockage per unit real time (figure 4.10) shows that a large "clog" occurs at time 70 in the visibility guided case; this clog is what prevents the adaptation to the time discontinuity at time 83. The *oldest-data-first* strategy shows much more extensive clogging. Ironically, since the naive strategy has no ability to adapt to changing conditions anyway, the clogging is somewhat moot.

In the *bandwidth-managed-importance* case, the system adds the data to the pipeline a little at a time, never adding enough to clog it for more than a fraction of a second; when the user enters the hallway, the system immediately switches to transmitting the needed data for the hallway, and when the time slider is moved, a similar switch is performed that sends the data for the new time. Since the pipe is never clogged, the needed data can be transmitted quickly in subsequent emergency situations.

Figure 4.11 shows the crucial data of concern; that is, the percentage of volumes per unit real time that are visible to the user and for which simulation data has been generated, but for which that simulation data has not been transmitted yet. The top curve shows that the *oldest-data-first* strategy spends almost half of the simulation run in this state; the viewer is looking at blank volumes when they should be seeing smoke. The center curve shows that the *visibility-guided* strategy does well until the second discontinuity at $t = 83$, at which point it breaks down as well; however, it recovers just after $t = 100$, whereas the *oldest-data-first* strategy doesn't recover for another 20 seconds. Finally, the *bandwidth-managed-importance* strategy is shown at the bottom. The spikes are 1 to 2 seconds long, and are only present at gross visualization time discontinuities (compare the locations with the vertical jumps in visualization time in figure 4.9). This corresponds closely with the minimum response latency for an emergency; it takes about 1 second just to transmit the data for a timestep at 3 kb/s, and the bandwidth manager makes new bandwidth available once every 0.5 seconds in our system, so the total response latency in our test case has a lower bound of 1 second, and an expected time of 1.25 seconds if the channel had absolutely no latency (which is the ideal case). Thus, our bandwidth manager approaches ideal performance under these conditions. If we bound the user's time velocity, we can further reduce the frequency of these spikes; with a

bandwidth of about 30 kb/s, we could eliminate them entirely for any user manipulation of the VCR controls (i.e. any time velocity under 10 virtual seconds per real second). This claim cannot be made for the other two strategies.

4.4.5 Conclusions and Observations

The purpose of this data management method is to provide data in a timely fashion to a networked client, assuming the network connection is too slow to simply send all of the simulation results to the client as they are generated. We have demonstrated a technique and ability to achieve this goal. Of course, it is reasonable to ask if this is even a consideration given that most desktop workstations are currently wired into networks that can provide up to one gigabit of local bandwidth. Simulations that can run in real time will probably not be able to generate a gigabit of data, thus they will not be able to saturate the network even if they naively transfer all simulation results to the client as they are generated. So why are we worried about conserving bandwidth between simulator and client?

Even a gigabit network might be saturated in two cases that are interesting to potential users. First, there may be multiple clients viewing the results of one particular simulation. With a naive strategy, providing the data to n clients requires n times the bandwidth. Ten users have already reduced the available maximum bandwidth by a factor of ten. Second, simulations are not the only target environment for this project. We also wish to be able to transport sensor data associated with rooms to a central location. Modern "smart building" programs are placing more and more different environmental sensors throughout the building. These sensor networks are used to regulate building systems such as the HVAC system, the elevators, and public lighting systems in order to provide more efficient use of the building. In emergency situations, these sensor networks also afford the ability to gather situational information for response crews. The quantity of data being generated in real-time for a high-rise building may be very large depending on what types of sensor are installed. Thus, the amount of information being generated may be quite large, even for a hardwired local network. We also need to account for the fact that any one particular client is not the only user of that network; they are sharing the bandwidth with other users, who may be consuming large amounts of it at any given time. Furthermore, we would like to be able to combine many different simulations to enhance our environment. With more than one simulator active at once, plus the remote access to the environmental database itself, the bandwidth is divided again that many ways.

The next consideration is the cases where we may not have a fast network. Even though

local networks are fast, wide area access over the Internet may not be. It is common to be unable to even saturate a one megabit link to a nonlocal site on the Internet (i.e. a site outside of the building). One of our goals is to integrate databases and simulation systems at disparate sites; thus, these WAN limitations may be telling.

There has been some effort in the emergency response community aimed at getting situational information to crew chiefs as early as possible; this would mean giving them information not only at the fire house, but en route to the site, since in an emergency they could not afford to spend any time sitting at a terminal before they get onto the transport to the site. The most apparent technology for this is radio or cellular modem technology such as the Ricochet network, which already covers many major metropolitan areas. This makes it ideal for these applications, since its coverage is large enough to practically assume its availability for response crews in large cities, and its commercial availability means that costs are relatively low. These networks operate an order of magnitude slower than wired networks; Ricochet offers 128 kb/s transfer rate, or 13 kB/s, which is considerably slower than many simulations and sensor networks generate output data for a building. This makes bandwidth an important consideration for such applications.

Finally, cost considerations come into play. Both embedded building systems and equipment for first response and fire research personnel are typically subject to strong financial constraints which may prevent them from having the types of gigabit network that computer scientists are accustomed to having access to. It is not unreasonable to assume that the "common" technology for these personnel is a generation behind the state of the art (e.g. they are still using 28.8 kb/s or 56 kb/s modem links); thus, a practical system designed for immediate use may have to assume a fast modem rather than a broadband network connection. In these cases, 3 or 4 kB/s is a fair estimate of available bandwidth.

It is also reasonable to assume that the inevitable growth in these transfer rates (as technology improves over time) will probably be accompanied by a corresponding growth in the power and scope of the simulations available to be run in real time, and thus the size of the output data we need to transfer to the client. This implies that the problem will likely be an issue for future applications as well.

Chapter 5

Simulator Integration

We have applied our simulation framework to integrate several physical simulators and interactive agents into the second generation walkthrough platform. All but one of these simulations were authored outside of the Berkeley Walkthru group; the most advanced of them, the CFAST fire simulator, has been under continuous development for the last few decades. In this chapter, we discuss each of the agents, how they were integrated into the framework, and the results and experiences of the integration processes.

In addition, we have developed two new programs, one for radiosity based lighting and the other for imposter generation, which have been given the same abstract interfaces of the other simulation agents; those will be described in more detail.

In this chapter, we will look at integrating a simulator into the framework from a user's perspective and describe the process for the simulators that we have integrated over the course of the project.

5.1 Integration API

5.1.1 Framework Modules

Our integration framework is extremely modular; there are many modules and structures that can be overloaded by the programmer if the default behavior is unsatisfactory, but for most simulators, only two major pieces of code need to be written; the simulation service itself (e.g. the simulator "glue" class that translates commands and data between the simulator itself and the framework) and the simulator's user interface plugin for the Citywalk visualizer (which adds com-

mands to the UI that allow the simulation to be controlled, and render the simulation data to the GL pipeline on each frame) (see Figure 5.1).

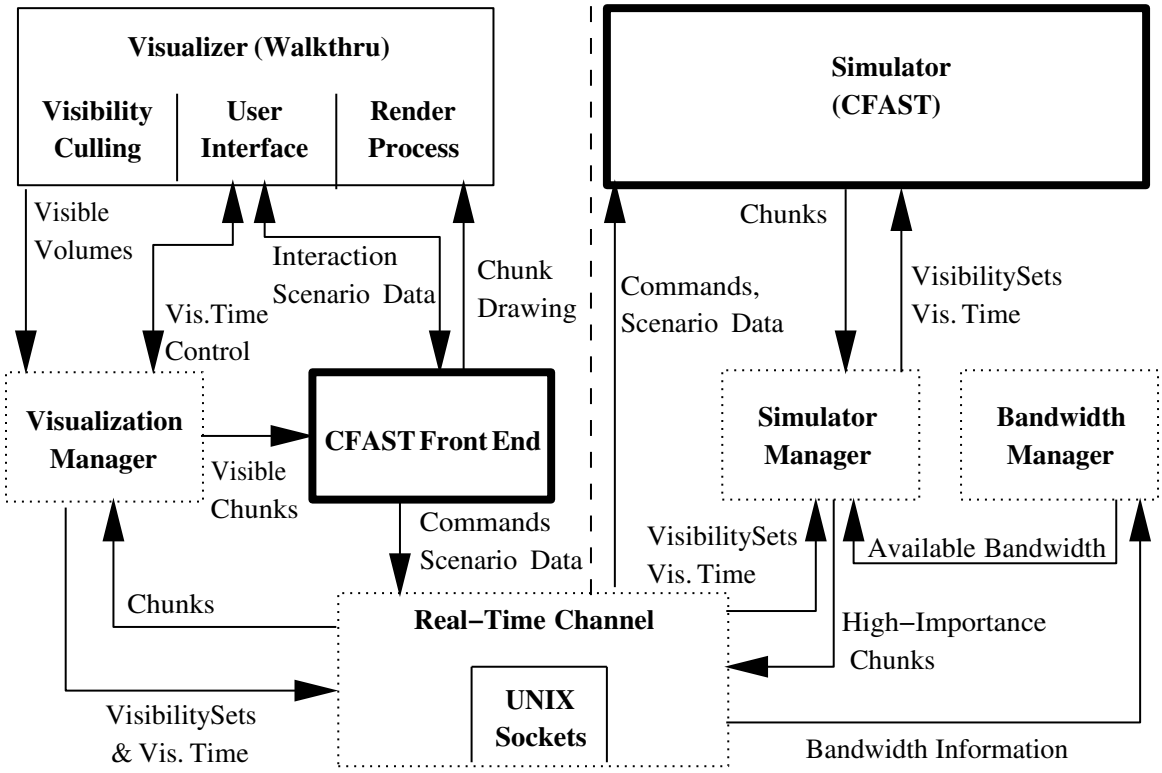


Figure 5.1: *A diagram of how the system components connect simulator to visualizer. Components in bold outline are created by the user; components in dotted outline are provided by the integration framework.*

5.1.2 Choosing an Interface

Simulation agents have available to them a connection to the database to retrieve information about the current user environment, as well as access to each client's visible set and potentially visible set. They generate information over time that can be either propagated persistently to the database ("persistent updates"), or environmental data which is stored in dynamic run-time structures and propagated to the clients on demand, based on the clients' areas of interest in space-time ("dynamic updates" [1]).

The client's area of interest consists of three elements: a visible set of cells, a potentially visible set (defined by the set of cells the user could possibly reach in a given time span), and a current time of interest, defined by a VCR-style time controller that the user can access for each

agent they are attached to as a client. Persistent database changes, such as radiosity computations, are propagated directly to the database in the same way that any other editing process would modify the database. However, the type of access that the agent has to the clients' areas of interest allows the agent to focus its efforts on particular regions of the database that the user is observing or editing in real time. For example, the radiosity agent can perform gathers preferentially on the visible set of cells, reducing the amount of time between agent startup and the time that the user sees reasonable results in the room in which she is standing.

In the case of dynamic updates, the agent updates a local *condition set* of data chunks, sorted by database cell and time stamp; each chunk describes the physical conditions generated by the simulator for the cell at the given time. This condition set may be asynchronously added to as the agent computes more and more results. A separate process in the database manager propagates this dynamic data to the client based on a prioritization of the data and the bandwidth limitations imposed by the communication substrate. Each cycle, the manager fills the available bandwidth with the most important (i.e. the "closest" in space-time to the viewer's area of interest) portion of the condition set and transmits that to each client. Each client maintains a local mirror of the condition set, which is gradually copied over from the server process to the client process, from the most interesting data down to the least interesting based on the interest set. If the agent stops generating data for a period of time, eventually the client's mirror will completely synchronize with the agent's local set. At any time, though, the agent can update regions of its condition set, at which point the propagation process will restart and run until they are again synchronized. The local copy of the condition set is used by the client to update displays and renderings of the simulation data in real time.

Dynamic updates are normally used by agents whose information content is time-critical in the real world, such as the multiuser service, or agents whose data sets explicitly include time as a fourth dimension. For example, both the physics simulator and the fire simulator primarily use dynamic updates, because they both generate space-time data; in the former case, object paths over time, and in the latter case, environmental conditions over time. The dynamic update method offers the additional advantage of being able to provide different "branches" of space-time to networked clients; i.e. the user could run two fire simulators on the same area of the database under different suppositions, and view both side-by-side in 2 different viewers (or even overlaid in the same viewer) to perform case studies or comparative analysis. An agent may mix dynamic and persistent updates; it is not limited to one or the other. For example, the physics simulator operates primarily dynamically, allowing the user to run time forward and backward and view the time profiles of objects;

but the user can at any time instruct the agent to propagate the "current" simulation state to the persistent model, causing a real, permanent change in object positions.

Note that, regardless of whether the agent chooses the persistent or dynamic update methods, it is unaware of the location of the client processes (i.e. whether they are on the local machine or on a remote machine). Thus, the agents and clients may be distributed dynamically in any way that the walkthrough framework or the user chooses with the goal to yield an appropriate workload partition.

Note also that simulation agents can themselves set watches on constellations in the environment. Because the agents have access to the database, these watches can be used to trigger updates in either persistent or dynamic data that are propagated appropriately. For example, several radiosity renderers could be assigned to different rooms. Each would watch the elements in nearby rooms looking for changes. If a chair is altered (moved by an editing process, or has its lighting changed by a radiosity process), the agent can identify that change via the watch and perform local updates by loading the changed objects, changing other objects, and committing those changes. In turn, this may cause additional agents to "wake up" via their watches, which then may make further modifications of their own. Alternatively, moving a chair with an editing process might notify a simulator that is simulating a fire on that chair. This may cause the simulation to dynamically compute the consequences of the chair's new position with respect to the flame spread. Any change in the database can cause a cascade of updates, all executing asynchronously on separate machines, to yield an appropriately updated database and dynamic world model. One example of this is the interaction between the radiosity agent and the tapestry generation agent (see figure 5.11 in section 5.6.5).

5.1.3 Simulator Component

The simulator component is typically comprised of three parts: The framework simulator component library, which is provided to the user by the Walkthru framework; the simulator code itself, which is presumed to come from an external source; and the glue code to link these components together, which must be written by the user.

The glue code is implemented by subclassing an object from the base class **RtLocalSimulationService**. This base class includes virtual functions for:

1. Receiving direct communications from a particular client, in the form of a binary block, packable object, or an integer command message;

2. Sending direct communications to a particular client or broadcast to all clients, again either as a binary block, packable object, or command message;

3. Running a simulation.

4. Clients attaching or detaching from the running simulation.

5. The manager receiving client state information from a client. This state information is accessible via separate function calls that take the identity of a client and return that client's latest visible set and lookahead set.

The subclassed element is compiled together with the framework library, the server libary, and the simulation code. This produces an executable that, when run, becomes a network server that serves the simulation. Alternatively, the same element plus the simulation code can be compiled in with the visualization client; this gives a simulator that can be run "locally" on the visualization client, with no changes in code.

During a typical run, the simulation agent progresses through three states. Simulation instances are created in a *setup* state. In this state, the initiating client uses the API functions to provide the simulator with enough information to begin the simulation. When the simulator has enough data, it places itself into the *run* state. In this state, it composes sets of chunks as it computes them and submits them to the simulation manager as results for the simulation. A simulation enters the *idle* state when it has computed the results of the initiating event to closure, either to a new static world state, or to an "end time" specified in the simulation setup. If world conditions are changed by the client, the simulator may return to the run state from the idle state until the consequences of the change have been computed, at which point it reenters the idle state. If a simulation agent ever has no clients attached, the simulation manager will halt its thread and deallocate it, regardless of its current state.

The thread in which the simulation agent's main function executes allows the simulator to generate simulation data as rapidly as possible without worrying about how that data is being transmitted to the visualizer. Once the main function is engaged, the simulator simply generates data, and "submits" the data (in the form of sets of spacetime chunks) to the simulation manager. Typically, the simulator produces chunks in batches that correspond to a slice across the volume set at a given simulation time; however, submissions may be made for any timeslice or volume ID, including space/time IDs that have been submitted previously. If a simulation chunk is submitted for the same time, volume ID, and subvolume ID as a previously submitted chunk, it supersedes

the older chunk. In this way, a simulator may modify any subportion of the previously generated data that is incorrect or that was generated as a quick approximation to be improved later. This will generally happen when the user makes a change in the environment at a particular simulation time, rendering many or all of the chunks after that time invalid. The simulator can also "retract" chunks, in which case the chunk is marked "dirty/remove" and will be removed from the client data sets as necessary.

During the run, the simulator may receive messages from the client that may change how the simulation plays out, including retroactive changes. For example, the client might request that a door be closed at simulation time t, even though the simulator has already simulated well past time t. In this case, the simulator must recompute values starting at t and retract or replace chunks indexed from time t forward.

If the simulator has the necessary capabilities, it may request the current set of visible volumes and the current visualization time from the simulation manager, and selectively generate or improve the corresponding simulation data to ensure that the visualization can proceed without pausing. We believe that this will be an important feature of future simulators that intend to provide visualization data in real-time while operating on very large databases.

5.1.4 Generic Interface Components

There are two additional classes that can be subclassed to provide the front-end capabilities required for a simulation. These classes are the *simulation client* class (**RtLocalSimulationClient**) and the *simulation view* class (**RtSimulationView**).

The simulation client base class performs all of the framework data management functions that are necessary on the client side. The user may subclass the client, if they need to do specialized setup operations on the client side for the simulator (i.e. to transmit case information to the simulation during the setup state), if they need to receive communications from the server, or if they wish to provide convenience functions for composing different types of messages for the server. These functions are handled by overloading the *setup* or *message receipt* virtual functions defined in the client base class. The client also provides functions for specifying and changing a "current" client visible set and lookahead set of volume tags.

The simulation view is subclassed to provide a user interface to a running simulation. If desired, the system provides a standardized simulation controller automatically within the simulation view window which controls the simulation time being rendered in the visualizer. This control

consists of a panel with simulator connection status, a time slider bar that shows the time range of the currently running simulation, and a set of VCR-style controls (play, reverse play, fast forward, reverse, and pause) that allows the user to control the rate at which time passes (Figure 5.2). The slider bar may be directly manipulated to change the current viewing time to any desired value; the VCR controls alter the "time velocity" of the user in simulation time, in units of seconds of simulation per second of real-time (Play is velocity 1, Fast Forward and Rewind are 10 and -10, respectively, Pause is velocity 0, etc.). The portion of the slider corresponding to data that has been computed by the running simulation is colored green; the portion corresponding to the as yet un-simulated timespan is colored red. This provides immediate feedback to the user about how far the simulation has progressed. The slider is prevented from entering the red region. The user can add additional displays or controls to the view that interact with the simulation client (which can be a generic client if the extent of the interaction is to send generic messages to the simulation). Multiple types of view may be defined for a given type of simulation.

Figure 5.2: *VCR controls that control the flow of "virtual" time.*

5.1.5 Walkthru Interface Components

If the framework is being used within the Walkthru, the user will generally also wish to overload a Walkthru *module* that provides a menu control to identify and launch connections to simulators, retrieve the associated clients, and apply desired views to those clients. The framework does not depend on the walkthru, however; this step is unnecessary if the visualization is standalone. The template module provided with the framework can generally be used with only a few changed lines if the additional state needed by the simulation (i.e. the state not represented explicitly in the

virtual environment) is small.

We also provide a special simulation view subclass (**RtWalkthruSimulationView**) which provides automatic propagation of the visible set from the current frame being rendered to the attached simulation client, so the user need not worry about computing that set. Thus, when using this special base class, the selection of which volumes are being visualized is determined simply by "walking" to the appropriate area.

The Walkthru view base class also includes a tool intended to mitigate the inherent "burstiness" of most simulations, including CFAST. This tool, called the *autopause* mechanism, will automatically "pause" the visualization time in two situations. At the beginning of the simulation run, autopause engages to allow the simulator to get a certain distance ahead of the current visualization time. This provides a buffer of data that allows the visualization to proceed smoothly if the simulation output becomes bursty. Furthermore, at any point where the visualization time "catches up" to the simulator, the autopause is engaged in the same fashion. In either case, after the simulation has provided enough of a buffer, the pause will automatically be removed and visualization time will once again move forward.

The Walkthru view class also includes a rendering function that is called on each frame. This function is typically overloaded by the user. It takes as input a Walkthru database cell and a set of chunks describing the current conditions in the cell, and renders the chunks' contents into the GL window. During each frame, it is called with all visible cells in the frame that have simulation chunks associated with them at the current visualization time. It is never called for a cell or simulation chunk that is not visible in the current frame; this provides efficient, rapid rendering of simulation conditions. The front-end is also provided with hooks into the visualizer's event processing system and is required to interpret any user interactions that might affect the ongoing simulation scenario. If such an interaction happens, the necessary changes to the scenario are transmitted to the simulator, and, by default, all simulation chunks from that simulation time forward are invalidated. The simulator then has the option to either invalidate or regenerate any portion of that data.

5.2 CFAST (The Consolidated Model of Fire and Smoke Transport)

5.2.1 Overview and Capabilities

CFAST [6, 1] is one of the best available fire chemistry and physics simulators. Created by the National Institute of Standards and Technology in the 1980s, it provides a description of how

atmospheric conditions in a building will evolve in the early stages of a fire (before major structural damage occurs that changes the qualitative nature of the building model). It has been thoroughly tested against physical experiments and is accurate enough that CFAST simulations have been used as evidence in court cases and legal investigations.

A CFAST building model consists of a set of rectilinear volumes connected by "vents," which is a generic category for connections between volumes. A vent can be horizontal or vertical, and has a specified height, width, and offset from the floor. Volumes are numbered from 1; volume 0 is a special volume indicating "outside" and constitutes an infinite heat and oxygen source and sink. So, for example, a door between rooms is a horizontal vent; a window is also a horizontal vent, but connecting to volume 0. An elevator shaft or fireplace flue is an example of a vertical vent. The user can also specify HVAC ducting between volumes; these are like vents but have a run length and can form a network with fans and outlets that provide certain airflows to the rooms. Each volume specifies a material for the floor, walls, and ceiling; these materials are chosen from a library file that contains material descriptions of common building elements such as drywall or wood paneling.

Within a volume, the user can place furniture or fire detection and supression systems. Furniture does not have a geometric description; it is "placed" at a 3-D point offset from the corner of the room, and each type of furniture carries with it an ignition heat flux and a set of output functions that describe the heat and chemicals generated by the burning furniture as a function of time from ignition. Detection and supression systems comprise smoke and heat detectors and pre-modeled supression systems like sprinklers; they are placed in the same way as the furniture. During the simulation, these elements are dormant until the system computes that the heat at the location of the object is high enough to trigger either the igniton of a piece of furniture, or the activation of a detector or sprinkler. From the activation time, the object then contributes to the conditions in the volume by either generating a secondary fire (in the case of furniture) or supressing existing fires in the room (in the case of supression systems). Detector activations don't contribute to the simulation, but they can be used to determine when people in the house would be alerted to the fire. The user can also specify particular vents being opened or closed at certain times in the simulation; this represents doors or windows being opened or closed during the incident.

The simulation itself is essentially a finite element solution, with a special twist. CFAST is a "zone model" simulation, which means that instead of partitioning each room into many tiny fixed cells, it is separated into only two cells by cutting the volume with a horizontal plane at some height. The height is not fixed; it begins at the ceiling, and as smoke and combustion products build up in the upper layer, the interface moves down, growing the upper subcell and shrinking the lower

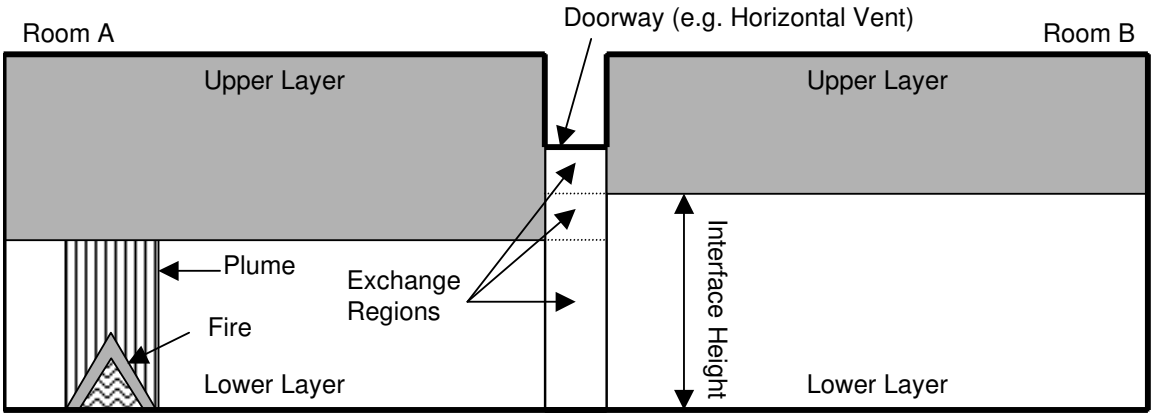


Figure 5.3: *The zone model finite element method used for fire simulation. Each room maintains two zones, with up to three qualitatively discrete exchange regions between volumes through doorways or windows.*

one. This is a special optimization that is a very good approximation for real fires, in which the hot air, which contains an even mixture of the combustion byproducts (i.e. smoke and soot), tends to form a largely homogeneous layer at the top of the room, separate from the cool and clear air in the lower part of the room. At the boundaries of the layers, heat and combustion products are exchanged, and the fires draw oxygen from certain layers and inject heat and combustion byproducts into others. The whole system forms a set of differential equations, and the heart of CFAST is a differential equation solver. The solution is advanced until a qualitative change occurs that requires modifying the equations (such as a door opening, or a layer moving down such that it can now vent through a window, or a piece of furniture reaching ignition temperature), at which point the equation set is updated and the solution continues. The equations relate the transfer of heat, pressure, and twelve other quantities including various gas and particulate (soot) concentrations between layers and volumes. The result is a set of time functions of these quantities in each layer of each volume and the position of the layer boundary (Figure 5.3).

CFAST by itself is not an interactive program. Input is composed with a text editor; the user then runs the simulator in a batch mode on the input file. This file contains rows of values denoting the dimensions of the volumes and vents, locations of furniture and detector elements, and the chemical and heat profiles of the "main" fire. The simulator then runs as a batch process and outputs a raw file. This file is postprocessed with a separate program to extract graphs of desired quantities over time. The user can also specify a small number of graphs in the text input file; these graphs can be drawn in real time as the simulation runs. Both the simulator itself and the

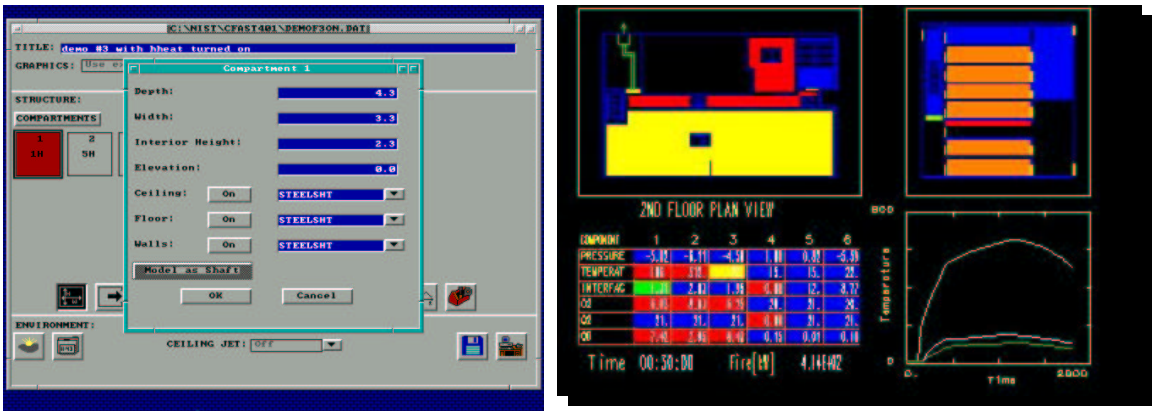


Figure 5.4: *CFAST's original input and output, as it is distributed by NIST. These forms are difficult for an untrained user to create and understand.*

postprocessing program are written in FORTRAN and are linked with a third party user interface and with a graphing package licensed by NIST (Figure 5.4).

5.2.2 Database Integration

In order to use CFAST with Citywalk, the user interface module for the simulator needs to be able to generate a CFAST input case given the state of the virtual world. While the geometric data needed by the simulator can be derived from the model geometry, the raw Citywalk model lacks the necessary material data (e.g. structural composition of the building, and burn properties of the furniture). Furthermore, a fire simulation requires a fire to be set at some point in the world, and the fuel and chemical properties of that fire must be specified (Figure 5.5).

Geometrically, CFAST required the environment to be partitioned into volumes with interconnected portals. While the basic form of a Citywalk database could be naively mapped directly to a CFAST input, Citywalk volumes are much more highly partitioned than a CFAST input needs to be. For example, a room might have a pillar in one wall that causes a spatial subdivision of the room into three cells instead of one. A naive translation would generate three simulator volumes from these cells, with corresponding (very large) portals between them. While technically correct, this results in a simulation that runs more slowly and provides quantitatively very similar results to one that considers the room one volume and ignores the minor volume intrusion of the pillar. To combat this inefficiency, we allow the user to specify a set of bounding volumes in 3D space that define the set of volumes to be considered for CFAST. Each cell that is contained in a bounding volume is considered part of that room, and the volume of the bounding volume is the room volume

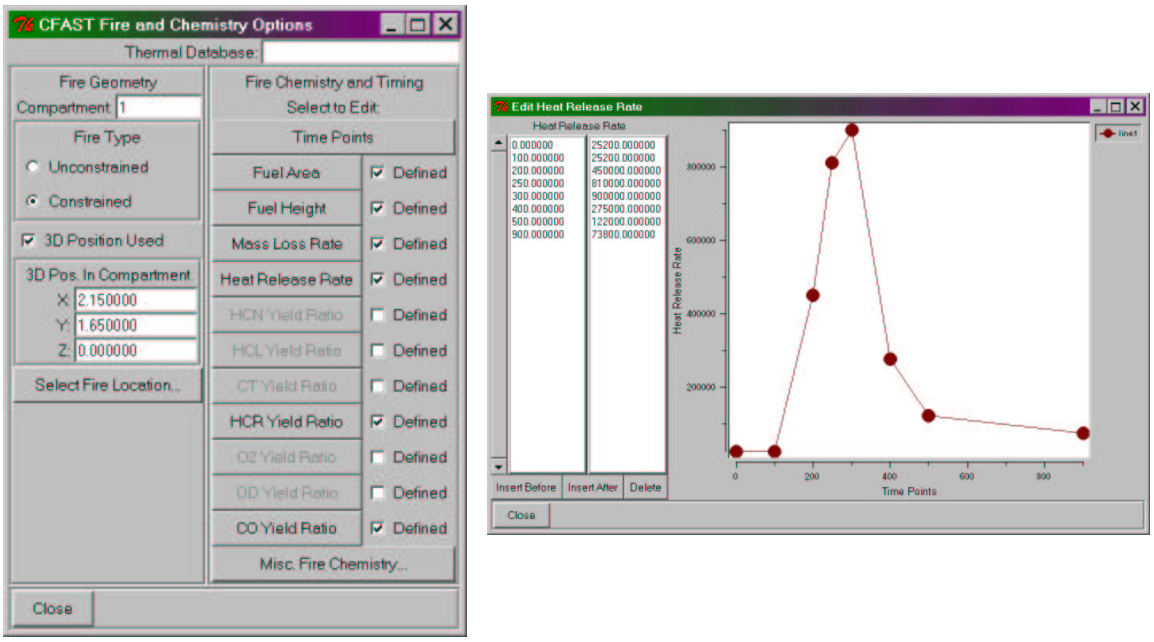


Figure 5.5: *The plugin UI for setting CFAST's chemical properties.*

for the simulator. Vents between these volumes are derived from paths through cells that connect cells tagged with a volume ID (Figure 5.6). When rendering graphics for a cell, the conditions for the volume containing the cell are rendered within the cell. Locations within a bounding volume can be easily mapped to locations in a Citywalk cell, and vice versa.

The material data was straightforward to integrate into the second-generation database. In the case of structural elements (walls, floor, and ceiling), we defined a CFAST-specific data object describing the materials of the structure that was accessible to the UI module when it exported the information. This object associated material types from a library provided with CFAST with the floor, walls, and ceiling of the volume to which the object was attached. In the case of detail objects (furniture and active elements such as sprinklers and smoke detectors), a second new data object was attached to Citywalk object classes that again referenced a furniture burn profile from a library provided with the CFAST distribution. These mappings are controlled by list dialogs accessible from the UI module for the simulator.

The fire chemistry data, on the other hand, is not directly related to the world model, so there is no obvious attachment point of the case data to the world model. We implemented this as an imported process from existing CFAST models, where the fire alone (without the world geometry) could be imported from a CFAST case. This data is basically a set of fuel supply and chemical

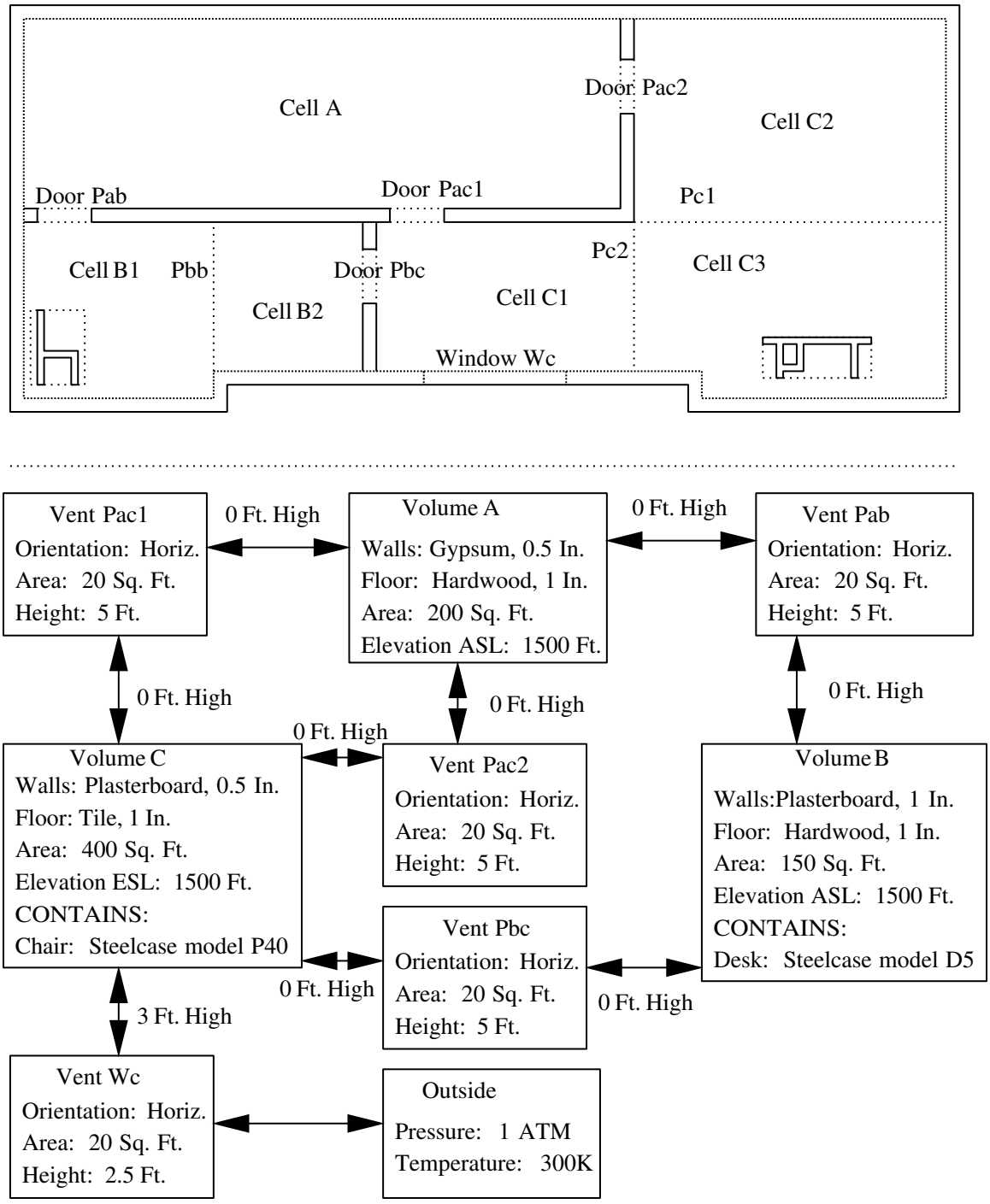


Figure 5.6: *How CFAST (bottom) maps volumes onto the world cell structure (top). The Walkthru model contains detailed geometric information, but little else; the CFAST model is geometrically much simpler, but contains chemical and materials information that Walkthru lacks.*

functions of time that describe the main fire; this information is passed on to CFAST unchanged when the simulation is initiated.

5.2.3 Simulation Service

The CFAST simulator module is a *virtual time* dynamic agent; its primary output is a set of values for various quantites (interface height, temperatures, pressure, chemical composition, etc.) for each volume over a span of time. Thus, the primary data chunk for each volume at each timestep contains a simple array of each of these tracked values. In addition, the primary chunk contains an array of detector or sprinkler activations, if any have been triggered in that volume. If a detector or sprinkler's ID is in the activation array, it is active for that time in the simulation. Finally, a second array contains records for all fires that have happened in the volume as a result of furniture elements igniting, as well as the main fire if it is in that volume. Each record in this array contains the output values for that particular fire at that timestep (e.g. energy output, fuel burn rate, etc) and a fire ID that is either the object ID of the furniture that is burning or a special value for the main fire.

When the user launches a CFAST simulation service, it goes into a setup state where it waits to receive all the information about the situation from the front-end module of the initiating client. The set of volumes and their interconnectivity, the set of objects and their class types and locations within the volumes, the main fire chemistry, and the ignition point are all transmitted by the receiver and compiled as a regular CFAST input case. When all the data is received, it is fed into a standard CFAST data file which is then read back in to the FORTRAN engine by the CFAST input routines, and the differential equation solver is launched.

Only one modification to the computation engine was needed; the insertion of a callback function after each timestep has been computed. This callback returns to the simulation service, where the world state for that timestep is read out of the FORTRAN data structures and converted into spacetime chunks. The chunks are then fed back into the simulation manager as output. At this time, any commands that have been received from clients are processed, such as modifications to the world state (e.g. doors opening or closing). These changes are propagated into the FORTRAN data structures, and control is returned to the differential equation solver for another iteration.

If a change is made in the past of the simulation (i.e. at a timestep that has already been generated), the simulation must be restarted entirely at that older time, because intermediate results may be changed by the new conditions. To do this, the system halts and resets the simulator, and reconstructs the input case from the spacetime chunks stored for that time. The simulation can then

progress normally, and any new chunks being generated will overwrite previously generated chunks as necessary.

When the simulation ends, the service enters an idle state. When all clients are disconnected, the service terminates. If, on the other hand, a client comes in with a command to make a change at some simulation time, the simulation restarts at that time and new results can be generated. Additional clients can connect and disconnect from a running simulation from anywhere in the network and view the results just as the initiator of the simulation does; this facilitates sharing results and cooperative exploration of the data.

5.2.4 User Interface Module

Control Section

A large section of the UI module for the CFAST service is dedicated to managing the mappings of material, volume IDs, and furniture classes to structures in the Citywalk model. Given a Citywalk database decorated with materials information and a loaded main fire profile, the only remaining element of the process is to place the main fire and begin the simulation. Fire placement is as simple as clicking on a "set fire" button (which is a UI function that can be mapped to any key), and clicking a spot on the floor. This information, together with world geometry, the material attachments for the structure, and the class attachments for objects constitute a full description of the input case for CFAST.

At this point, the UI module queries the simulation manager for a CFAST service available to the machine. If more than one are available (for example, one on the local machine and one on the network), the user can select which one they want to use. The module then launches the service, compiles the input case from the model data, and starts the service running.

While a simulation is running, certain actions taken by the user on the environment can affect the state of the simulation. For example, closing a door or moving a piece of furniture can affect gas flow or the time of ignition of that furniture, respectively. To identify these conditions, when a piece of furniture is added to the input case, a watch is applied to that furniture element. If a user modifies the element, that information is noted by the watch function and the change is sent to the service.

Rendering Section

The 3D rendering callback for an active CFAST service iterates over the cell list in the frame, converts the cell to a volume ID, looks up the conditions for that volume in the condition set, and renders the conditions according to the current visualization mode. The user can select from a number of modes with a UI panel. Each mode draws the conditions into the frame in a different way; the modes are selected to help visualize particular aspects of the conditions in the volume. For example, the user can elect to draw the conditions in a pseudo-realistic fashion, which overlays smoke and flame animations that represent "realistic" views of the environment. Alternatively, a schematic view is available that uses transparent, colored polygons to represent smoke and flame; this mode conveys similar information but doesn't obscure the view. Another mode is the thermal imaging mode, where no smoke is drawn at all, but the walls are colored according to the temperature of the volume; this mode conveys normally non-visible information in a visible way (Figure 5.7). Many more modes are conceivable; the task of choosing which ones convey information the best is a large research task in its own right, and we hope that the framework will be used to create more types of visualization in the future.

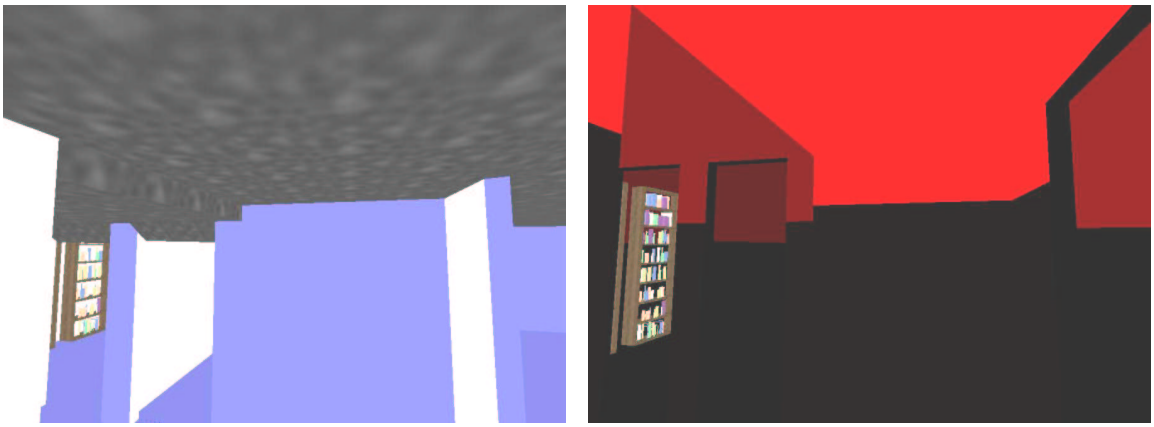


Figure 5.7: *CFAST view modes. Left, realistic mode; right, thermal mode.*

If individual model elements like fires, ignited furniture, or activated sprinklers are present in a volume in the view, and are also present in the condition set for that volume in the frame, special graphics are drawn at their locations as well. For example, an ignited piece of furniture will show a fire plume coming off its surface, and a sprinkler will show a water jet if it is activated.

In addition to the rendering plugin for the 3D view, the CFAST service provides qualitative output in the form of "probes." The user can place a probe anywhere in the model, and get a

numeric reading of any of the simulated physical quantities (atmosphere composition, heat, etc) at that location at the simulation time. A probe attached to the user position moves with him and reads conditions at the viewpoint. This process simply adds the probe locations to the visible set on each frame, and updates the dialog box values accordingly.

We also implemented a completely separate rendering module for CFAST that plugs into the Floorsketch program (described in chapter 6). The result is the ability to display simulation results directly on the schematic 2D floorplan; rooms turn yellow and red as the conditions worsen, and icons are drawn to represent fire locations and sprinkler activations. This module demonstrates the orthogonality of the simulation data framework; it can operate completely independently of the Walkthru 3D visualization. The user can create cases and run simulations entirely within the simple 2D floorplan. The CFAST input geometry can be derived directly from the 2D plan, and a set of dialogs specific to the Floorsketch-CFAST front-end plugin allows the user to set the non-geometric properties of the simulation, such as chemistry and fuel values. This module can run independently as a client of a simulation anywhere in the network.

5.2.5 Application

The CFAST module in the Citywalk simulation framework successfully leverages the simulation power of CFAST with the visualization mechanisms of Citywalk to provide a powerful simulation tool. The system has been tested and works well, even over very low bandwidth links (e.g. modem links). It has been well received in the fire research community; interest has come from teachers, lawyers, and researchers who wish to use the system to more easily communicate the results of fire simulation. This simulator subsystem could also be used in a "cybernetic building," where the building has a network backbone that unifies control of its services and sensors. Using real-time sensor output in the condition sets would be trivial; such a network could provide earlier warning of real fire situations, and provide information through the Internet to remote clients in the firefighting and emergency response services. Display panels in the lower floors of the building and remote panels in fire trucks networked with wireless WAN technology could give crews better information earlier and help to save lives and property.

One area that has not been well explored, but shows promise, is the ability to extend the capabilities of the simulator with the Citywalk framework. For example, CFAST does not simulate inter-volume heat transfer through walls because its text-file-based input mechanism is too difficult to use. The Citywalk model has enough information to compute these form factors without any

extra effort on the part of the user; with this information, CFAST could improve the quality of the simulation with little extra coding.

The CFAST service was the first service created in the Citywalk simulation framework; as such, it acted as the testbed for many of the protocols and simulation support services. As such, the design of the module does not make use of some of the more recent features of the network. Were we to redesign the module today, we would have the client simply identify the model database and the chemistry information to the service, and have the service build the input case directly by reading the database. This would make the setup faster and more efficient, as the client would not have to transmit all of the case information to the server at startup time.

5.3 IMPULSE (Impulse-based dynamics simulation)

5.3.1 Overview and Capabilities

IMPULSE is an object-level simulation of physical dynamics in the presence of forces in the environment. Input to IMPULSE is a scenario consisting of a number of objects, each of which is a rigidly connected assembly of convex bodies, each of which has physical information including mass and moments of inertia. The user can also specify fixed forces such as gravity. IMPULSE then simulates the positions and orientations of the objects as a function of time. It combines the Lin-Canny algorithm (to rapidly compute collision times) with an impulse-based physics model (i.e. objects never stay in continuous contact with each other). The result is a simulation that can run relatively quickly, and is also quite accurate; several tests performed by the authors verified its accuracy against both statistical and qualitative measurements of complex real-life dynamics phenomena [47].

The basic IMPULSE system uses a combination of input files and hard-coded force functions and rules to generate its simulations. Simple simulations involving only a mixture of fixed and free objects under an initial velocity and a gravitic force are directly supported. Other simulation types, such as pendulums, rotating disks, or special force functions, are hard-coded into the IMPULSE code base and activated with special codes in the input file that attach C force functions to specific objects. Given the input file, the system then generates an OpenGL animation of the object behavior as it is simulated. These animations can be stored and played back as movies.

5.3.2 Database Integration

Much like CFAST, IMPULSE can infer part of its input from the existing Citywalk database structure (e.g. the shapes and positions of the objects in the environment), but also needs extra information that is not provided by default (e.g. the mass and inertial properties, and the convex decompositions of the objects in the environment).

For detail objects (e.g. furniture) this information was added to the database as an IMPULSE-specific attachment to the Citywalk object class. The added information is generated with an external program that performs a convex decomposition of an object, and computes its mass and intertia properties. This results in an IMPULSE profile that can be attached to a Citywalk object class. Fixed structural elements, such as floors and walls, are exported to IMPULSE as extremely thin (fraction of an inch) slabs that are fixed in space. These slabs are generated on the fly by extruding the wall patch away from the room cell.

5.3.3 Simulation Service

The IMPULSE simulation agent is a *virtual time* dynamic agent like CFAST. The condition set for a volume for a timestep is a set of object IDs that are in that volume at that time, plus a transformation of that object from its pose at the start of the simulation. The database ref is used for the object ID; this allows multiple different clients to identify the same object in the simulation output. The service also provides an extra input; any client may send a message to the server that assigns a force vector to an object for a period of virtual time. This results in the user being able to "hit" an object from the UI and see the reaction in the simulation.

When the simulation is launched, it enters a setup state similar to the CFAST setup state, where the client transfers case information to the server. The client provides a set of fixed slabs which represent walls, and a set of object definitions and poses that describe the objects to simulate. The service then enters run mode and provides condition sets (e.g. simulated object poses) as fast as it can. When the last client disconnects from the service, it terminates.

5.3.4 User Interface

Control Section

The IMPULSE input case is completely derivable from the world model annotated with physics information. Thus, aside from the mapping function that allows pairing physical definitions

with object classes, there is little to the setup UI than the "start" button, which initiates the transfer of the world state and the identification and launch of the IMPULSE service from the simulation manager.

There is one runtime function, the "kick" function. This is implemented as a mouse event where the user can right-click on an object and kick it; this sends a force event to the simulator and results in the object having a sharp force applied with a vector equal to the mouse-to-object vector (i.e. the kick is away from the viewer).

Rendering Module

The rendering module hooks into the object drawing callback from the render engine. If the object to be drawn is present in the condition set for a rendered volume, the transformation is applied before drawing the object. If there are other objects present in the conditions that are not being drawn by the current view, they are located in the database after the frame and drawn into the frame with the appropriate transformation.

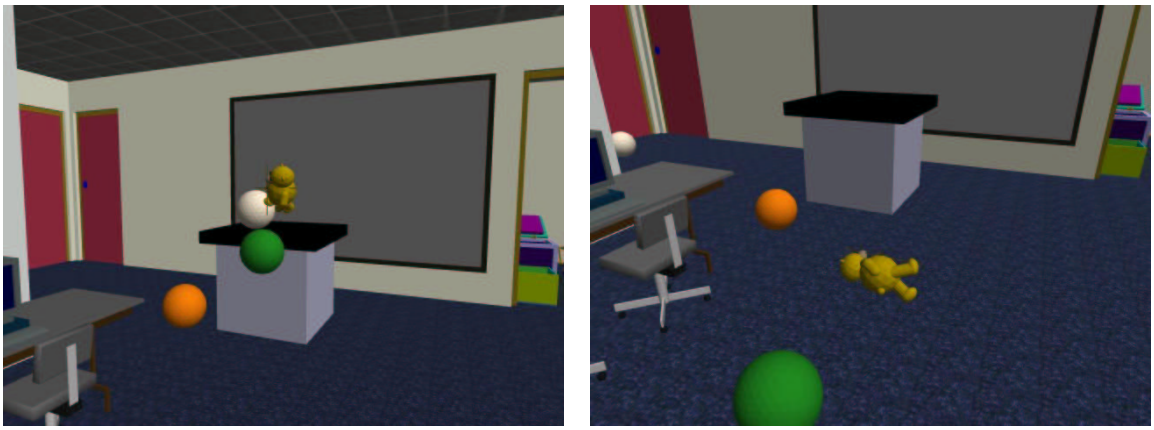


Figure 5.8: *IMPULSE simulating bears and balls bouncing in a laboratory.*

5.3.5 Results

While IMPULSE is often too slow to provide true real-time physics performance, it still provides some physically realistic behavior for objects in the world, and for small collections of objects it can provide real-time physics in Citywalk (Figure 5.8). A serious issue with IMPULSE is the fact that most objects in a virtual building model are in rest states; that is, sitting on other objects

in a stable fashion. IMPULSE is not well suited to these objects as they stay in a state of constant collision. We tried to mitigate this with some code that caused objects that were not moving much to anchor themselves in the world, and thus remove themselves from the simulation, but this approach had a number of problems with stability and accuracy.

An interesting possible use for this simulation agent is to partition the world into sections by volume and have one agent on a separate CPU responsible for simulating the behavior in each volume. This approach would limit the load on each agent. Database watches would allow agents to determine when an object has entered their "field of view;" ephemeral database attachments describing current velocities would allow them to remain consistent across boundaries. This methodology could provide seamless, large-scale physical simulation across a large model.

This agent was the second one implemented in our framework, and it is interesting to note that it took only two weeks from receiving the IMPULSE source code for the first time to running simulations in the Citywalk model.

5.4 Real-time Multiuser Walkthru

5.4.1 Overview and Capabilities

The Multiuser agent [28] distributes "avatars" of walkthrough clients to each other so that viewers can see and interact directly with other viewers in the space. Each user contributes one avatar to the environment; this avatar resides in one volume at any given time, corresponding to the location and orientation of the view frustum on that user's client. Other users can see that avatar moving as the original user moves, and can communicate directly with everyone they can see via a localized chat mechanism. Users can select which database model is used for their avatar; this selection is reflected on the other client machines as part of the visualization.

5.4.2 Simulation Service

This is the first example of a *real-time* simulation agent; clients are interested only in the most current state of the other clients in their region of interest, so there is no time control aspect of the simulation, and "old" data is useless to transmit.

The multiuser service is unique in that it does not actually perform "simulation" per se. When a client connects to the service, it is allocated a unique ID. This unique ID is an integer, allocated by the simulation server, used in the subvolume ID field of the condition set; the condition

set for a volume contains one record for each client in the volume. The definition of a "volume" is up to the client system; in Citywalk, database cells are the natural choice, since the database stores a unique integer ID for each cell. This record is a spacetime chunk with a subvolume ID equal to the ID of the client; the body of the chunk contains the avatar ID, pose, and motion prediction information about the client. If no clients are moving, the service sits idle. If a client does begin moving, it begins sending periodic updates of the avatar ID, pose, and motion prediction to the service. The service receives this information and updates the appropriate chunk, if necessary moving it from one volume to another (deleting the chunk in the original volume and adding it to the new volume). The data distribution layer then propagates the latest information back to each client that is interested in those volumes.

If a client wishes to "speak," they send a talk message to the service with the desired text. This text is rebroadcast to each client for whom the volume is in their interest set (Figure 5.9). Thus, the user can "hear" anyone who is in their visible or lookahead sets.

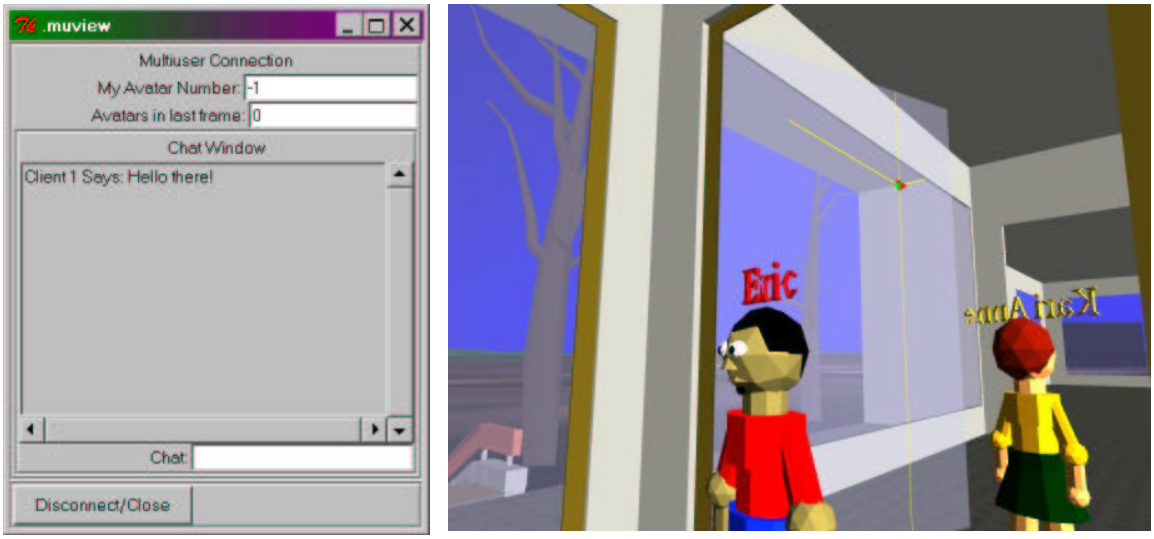


Figure 5.9: *Left, multiuser chat window. Right, avatars interact with each other and the doorways in the MIT LCS model.*

5.4.3 User Interface

The UI for this simulator is very simple. For setup, a menu entry allows specification of which avatar model the current client wants to use; the only other control is to locate and connect to a multiuser service and begin interacting. The rendering module simply iterates the active chunks

in all the visible volumes, and renders the records with the appropriate avatar model in the specified pose. The motion prediction information, combined with the timestamp on the chunk and the system clock, allows interpolation of frames between updates from the service. The chat box is a separate dialog that is part of the module, and presents an input box and an output box that displays all chat messages received from the server.

5.4.4 Results

This was the simplest module written for the framework; it only took one afternoon to write, and provides a very usable and efficient multiuser implementation within the framework. The resulting functionality is similar to the RING system [28], but using a star topology rather than the inner ring - outer ring topology. In fact, implementing the ring topology would be fairly straightforward within the framework; if we started multiple services, each responsible for one set of volumes, and had the client UI drop and reattach connections as that client moved from server to server, the functionality would be identical to that of RING.

An interesting extension of this service would be propagating sound based on a more interesting function than lookahead sets; e.g. a physical sound propagation function [4]. This could have aesthetic and usability applications for evaluating spaces.

5.5 Tapestries: On-line Imposter Generation

In order to achieve fast, interactive frame rates, first-generation walkthrough systems utilize a combination of model-based visibility culling, prediction of user behavior, and suitably chosen imposters or lower levels of detail (LOD) for some parts of the model. To produce optimal displays and to keep the frame rate as constant as possible, most objects are stored at various levels of geometric complexity. In each frame, the display manager can select a combination of object representations that produces the best image possible based on an estimation of the available resources [39, 23]. To guarantee constant frame rates, the lowest LOD may have to correspond to not displaying an object at all.

In scenes of high depth-complexity many objects, or portions of objects, that are not visible to the current view may be sent through the rendering pipeline. In order to achieve interactive frame rates and visual quality in such environments, it is imperative to render only those portions of the scene that are actually visible. One approach to address this problem is to generate view-

dependent image-based LOD representations for large masses of objects [23, 48, 49], as well as for individual objects. Such context-dependent display representations can better exploit available rendering resources. Another strategy is to utilize 2.5D textured depth meshes either as the primary rendering primitive [50] or to provide the background behind other parts that are displayed with full 3D geometry [20, 51]. Of course, any pre-computation of such representations will become invalid if the underlying environment changes. Some image-based techniques generate image imposters on line by caching results from the frame buffer [52, 53, 54]. Our approach extends existing imposter generation techniques to incorporate samples from multiple views and to support automated on-line generation and update in an interactive walkthrough environment.

A tapestry is a textured mesh constructed from an on-line sampling of the environment. The sampling is done from a collection of adjacent views, resulting in a representation of the surfaces in the environment visible from those views. The tapestry imposters can be regenerated on-line if the underlying environment changes. In our implementation, the Tapestry Simulator automatically generates imposters given a cell-portal based environment. A tapestry is associated with each relevant portal and represents the portion of the environment visible through that portal when viewed from a particular region of space. A similar dynamic technique is presented in [55] using textured rectangles as portal imposters. In the following sections, we discuss how a tapestry is constructed from a given environment, and then how this functionality is incorporated into a simulation engine in the new framework.

5.5.1 Tapestry Construction

A tapestry is a Delaunay mesh with vertices corresponding to sample points. With a relatively dense sampling, a subset of the samples corresponding to important visual features is chosen as vertices. In addition, explicit edges are specified at apparent discontinuities in the sample image and incorporated into a constrained Delaunay triangulation. The set of sample color values is stored as a texture and mapped onto the triangle mesh. This basic approach has been utilized to generate imposters for urban environments [51].

To generate a tapestry for a given view, the environment is first rendered and the resulting image is stored as a texture. World space coordinate values are derived for each sampled pixel location. Each pixel is then processed and labeled as a depth- or normal-discontinuity if the depth or estimated normal values, respectively, differ significantly from any of its eight nearest neighbors. Such “discontinuity pixels” are then chained into edges. A collection of pixels is approximated by

an edge, if the line segment formed by the end points of the chain is a reasonable approximation to the set of pixels, both in 3D and in the 2D projection in the current view. These edges and vertices are then incorporated into a Delaunay mesh. The vertices store the world-space location of the sample, resulting in a 2.5D representation of the part of the environment visible from that view. The pixel location of the corresponding sample is used as the vertex texture coordinate.

The resulting mesh contains enough geometric information to produce appropriate parallax effects when viewed from nearby viewpoints. In general, there will be areas that were not visible in the initial sampling. In these locations, the mesh triangles incorrectly interpolate between two disjoint surfaces. The further the observer deviates from the generating view, the more apparent these "skins" will become. In order to minimize these visual artifacts, we perform sampling from additional nearby views, and incorporate this information into the mesh.

5.5.2 Tapestries in the Framework

We have implemented a tapestry-based simulation agent that automatically generates tapestries in batch mode for a walkthrough environment and updates them on-line, when the environment changes.

In batch mode, the agent traverses the portal list and generates a tapestry for each portal. Given a portal, a initial viewpoint must be chosen, and then only the geometry visible behind that portal should be rendered to generate the tapestry. The tapestry should incorporate all geometry visible through the portal. To this end, we choose a viewpoint on the portal normal ray, at a distance that results in the viewing frustum conservatively covering the portal. To capture a wide view angle of samples without distortion, a spherical projection surface is used instead of a plane as the manifold for the 2D triangulation. This also allows us to generate tapestries for large cells with complex periphery geometry (such as a large room with much clutter along the walls) by placing the initial viewpoint in the center of the room and generating samples for the full 4π angular range about the viewpoint. Figure 5.10 illustrates this setup schematically.

In order to generate the appropriate view, we utilize the cell-and-portal cull traversal to generate the list of visible objects resident beyond the portal. We start a cull traversal given the initial view, but do not add any objects to the display list until the specified portal is reached. The traversal then continues as usual from that portal through any subsequent visible portals. The resulting list of visible objects is rendered to produce the input to the tapestry generator.

The tapestry is attached to the portal and added to the database. At rendering time the

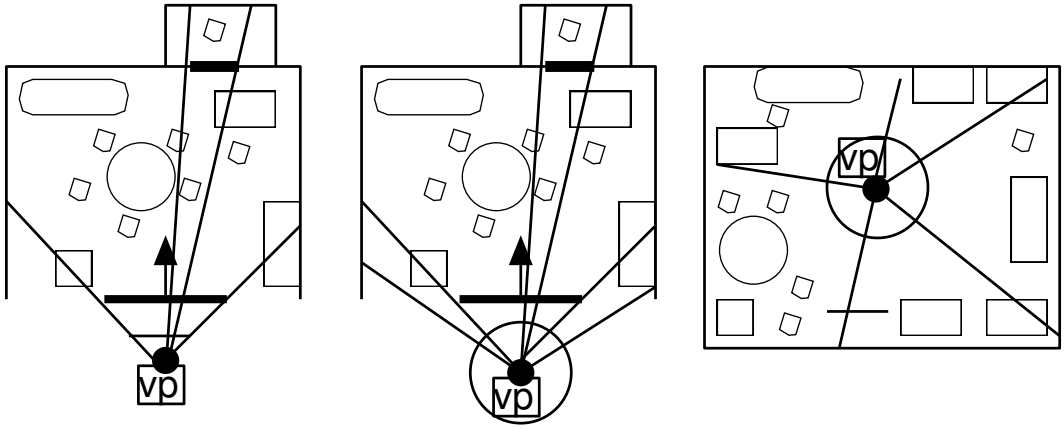


Figure 5.10: View location and projection surface for portal and cell tapestry construction.

tapestry will be used for viewpoints that are greater than a specified distance from the portal. If the cull traversal encounters a portal with a tapestry and the viewer is far enough away, the tapestry is placed on the display list and the traversal is terminated.

The agent also supports dynamic tapestry updates. Each object that is represented by a tapestry imposter stores a reference to the tapestry. Each cell also maintains a list of adjacent tapestries. The simulation agent maintains a watch on all tapestry objects. If an object moves, the appropriate tapestries are regenerated. If an object's surface appearance changes, then only the texture maps associated with the geometry need to be regenerated.

As the agent may be running independently on another machine, it does not cause contention for rendering resources. Such a simulation could therefore also be used without the initial batch calculation by incorporating a just-in-time look-ahead capability into the simulation agent. Tapestries can then be generated on-line only for those portions of the environment that the user is exploring.

5.6 Radiosity on Demand

5.6.1 Overview

To provide for realistic lighting conditions for walkthrough building models, a number of researchers have developed radiosity solvers within first-generation walkthrough systems [43, 56]. These solvers generally calculate and store radiosity shading on individual model surfaces, which are loaded and rendered at model view time. For large models, the computation time required for

several complete, global radiosity iterations can be very large. For example, performing three global iterations of radiosity on a building model with 300 rooms and a total of 40,000 polygons required nearly six full days to compute on the most powerful million-dollar, multi-CPU systems available in 1994. Even with the massive increases in computer power since then, the same computation requires nearly three days today on a fast desktop workstation (costing a few thousand dollars). Newer approaches such as [57] further decrease the amount of time necessary to generate a lighted model, but the computational expense still makes it difficult to employ such "global" solvers in an interactive, dynamic environment.

By developing a radiosity solver as a simulator plug-in that operates incrementally, we hope to provide support for interactively visualizing the effects of changing the lighting in a dynamic walkthrough environment. This simulator refines partial shading solutions on a surface-by-surface basis by focusing computational resources on areas of greatest visual importance to the currently connected simulation clients.

5.6.2 Incremental Radiosity Updates

We assume that most changes made to our virtual environment at any one time are small and concern only a few objects or rooms. It is thus natural to assume that a previously calculated radiosity solution is a good starting point for calculating the new, adjusted radiosity solution. Such incremental adjustments to existing radiosity solutions have been discussed previously in the literature [58, 59, 60, 61, 62].

A second means by which we hope to achieve a performance improvement is by taking advantage of the fact that while any change may cause the illumination value to change on many polygons, we are generally more interested in reducing the shading error visible to a particular observer at a particular time, rather than globally refining the entire model at once. Past work in importance regions has demonstrated how to provide solutions with a bias for increased accuracy near a preferred viewpoint [63, 64]. Several techniques have been presented for adaptive re-computation of radiosity in a changing environment, such as [65] which presents a method for online radiosity updates using a radiosity renderer running concurrently with a modeling system and communicating through shared memory.

We have tried to take the best of these many ideas and integrate them into our dynamic, second generation walkthrough environment to see how close we can come to a "real-time" radiosity update when objects are moved around or tumble through a room under the influences of a physical

force simulator.

5.6.3 View-Based Radiosity Updates

In the event that the radiosity process is started on a model for which no previous solution has been computed, the observer locations in the model can be used to guide the global solution dynamically to provide more "immediate" results to the observers. As we mentioned previously, the standard simulator plug-in interface provides each simulator with a visible and potentially visible set for each attached client. By taking the union of the cells in these sets, we generate a single global visible set that represents the areas of interest for all users within a model. These cell sets are then passed to the radiosity solver to guide the order of computation. Using priorities determined by the parent set for each cell, based on the proximity of the closest viewer and number of viewers to the cell, as well as the number of full gathers to the cell (the number of full gathers is the number of gathers to the object contained in the cell with the minimum gather count), the radiosity solver can order the cells in a priority queue. Selecting the next object to undergo a radiosity gather is then a simple matter of iterating through the objects contained within the highest priority cell until one is found with the same number of gathers as the full gather count of the cell. If all objects within the cell have a greater number of gathers than the full gather count of the cell itself, then the cell's full gather count is incremented, it is reinserted into the priority queue, and the process repeats with the next highest priority cell.

Obviously, the priorities assigned to cells are the key to calculating the radiosity in this scheme in a way that is satisfying to users. While we would like to concentrate as much computation as possible on the objects in the visible set, since that is the only one that users are seeing at any given time, it is also important to provide some computation to the objects in the lookahead set so that users find the other parts of the model reasonably well lit when they move around. To accomplish this, we assign priorities to the visible set cells of twice the number of full gathers to those cells, while the cells of the lookahead set receive priorities equal to the number of full gathers to those cells. We also provide a slight additional increase in priority to the visible cells, so that they receive the first gathers in an unlit model prior to the lookahead set. This priority choice seems to work well in practice, lighting those areas visible to the viewer, while not ignoring other areas nearby.

5.6.4 Radiosity Updates in a Dynamic Model

The previous section assumes a static world which we are trying to dynamically shade to convergence given a set of viewers moving through the model. Since we wish to support concurrent simulation and editing of model contents, it will often be the case that the radiosity solution in progress, or the solution that has been previously computed, is invalidated by objects moving or changing material properties. It is wasteful to recalculate the radiosity solution for an entire model from scratch when only a few objects have been altered, so we instead look to support selective correction to the existing solution. By making use of the database watch mechanism, the radiosity simulator can monitor all objects on which a solution has been computed. If any of these objects should change, the simulator is notified of the change and can analyze the object to determine an appropriate dynamic response.

In the situation where the object's material properties have changed, we can make use of the standard method of shooting a correction for the changed material properties back into the environment[66]. The walkthrough visibility system provides an efficient mechanism for determining which objects are visible to the changed object; this is simply a slight modification of the observer's display list computation. This set is used to accelerate the computation of form factors, and can be precomputed and cached at model generation time. These caches are updated when objects are moved, added, or deleted.

By shooting radiosity corrections to all objects in the changed object's visible set, the error in direct illumination is corrected efficiently. For each object receiving correction, we may further shoot a new correction to the objects in that object's visible set, and so on, until the correction factor becomes small, or we have matched the number of light bounces previously computed. Once this happens, we merge the correction with the existing radiosity on all objects with corrected radiosity, resulting in a model that is updated and correct for the new material properties of the object.

In the situation where the object's geometry has changed, we can also use the standard method of shooting corrections into the environment based on the change in form factors. Although a naïve method for finding form factor changes would require a computation on every object in the model, we are fortunate to have two shortcuts provided through the standard functionality of our system. Using the point location facility provided by a KD tree we can quickly find the cell location of the changed object's new position, and from that we can quickly compute the object's new set of visible objects. Using the union of the object's new and old set of visible objects, we can compute the form factor differences between each of those objects and the changed object in its new position.

We also compute the form factors of each pair of objects in the changed object's old visible set and new visible set to determine changes caused by the object acting as a blocker. After discovering all possible form factor differences, we can then shoot radiosity corrections into the environment in much the same way as for the case where the object's material properties have changed. Having done this and merged the correction with each object's radiosity, our model's radiosity is correct for the new geometry position.

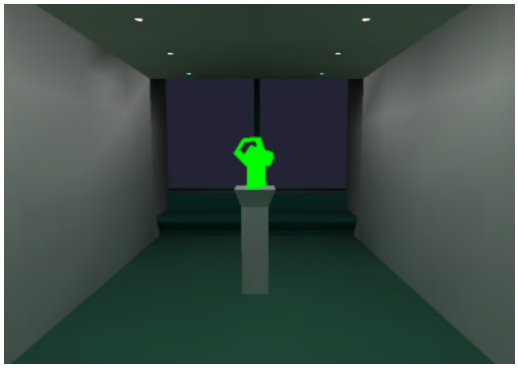
5.6.5 Radiosity Results

While we did not expect to get "real-time" performance due to the fairly loose coupling of the radiosity simulator to each client's rendering system, we did see performance that was acceptable for interactive purposes in many instances. With the described techniques in their current state of implementation – without having done any code optimization – we see radiosity updates every few seconds, with the final solution taking on the order of a few minutes. A reasonable looking intial solution takes about one minute. At this point the user can get a rough impression of how the lighting change will affect the overall appearance of the environment.

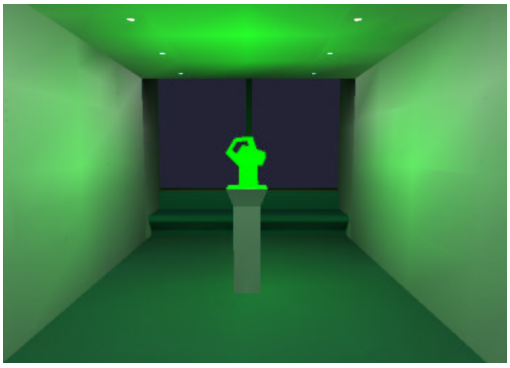
Figure 5.11 shows an example of the radiosity and tapestry simulators interacting via persistent updates. The user begins in a Soda Hall model that has been solved for radiosity and local tapestries; the alcove at the end of the hallway is dark due to a lack of lighting. The user places a brightly emissive green sculpture on a pedestal in the dark alcove. This editing action creates in a database transaction which adds the emissive sculpture to the alcove's spatial cell contents list. The radiosity agent and tapestry agent have both placed watches on this cell, and receive updates that cause them to move from the idle state to the active state, and begin reprocessing radiosity and tapestry solutions, respectively, on that cell and nearby cells. After a short time, the radiosity agent begins committing transactions of its own that alter the lighting conditions on wall surfaces near the sculpture, reflecting the new green light being emitted in the alcove. This, in turn, triggers further watches directed at the tapestry agent, which begins incorporating the newly green-tinted surfaces into the tapestries that show those wall surfaces. Eventually, the two agents go dormant again, after the lighting effects of the new sculpture are fully integrated into the cells, surfaces, and tapestry abstractions near the alcove.

From our observed results, it is apparent that making use of a radiosity solver as a simulator is a useful technique for supporting dynamic lighting updates in an interactive walkthrough environment. Although we do trade some performance to enable the integration of the radiosity

(a) We approach the alcove to be edited. At this distance the alcove is represented by a tapestry.



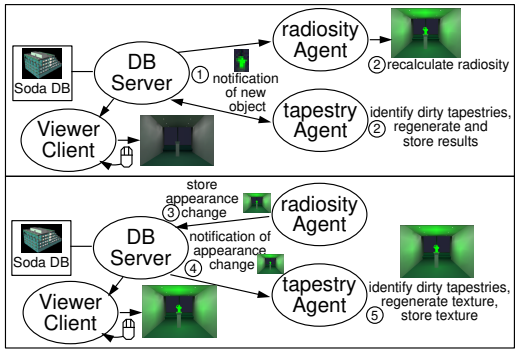
(b) We deposit a brightly emissive sculpture on the pedestal; this triggers watches in both agents.



(c) We observe the radiosity update. After a minute, the radiosity agent commits changes to the wall colors, causing watches to fire in the viewer. The viewer reloads the changed surfaces, making the new lighting visible.

(d) Moving back, we see the tapestry that was committed by the tapestry agent while the radiosity agent was computing the first gather. The bright lighting is not yet present in the tapestry.

(e) After 3 gathers the radiosity solution has converged, and the final watch causes the last updated tapestry to be displayed in the viewer.



(f) A diagram illustrating the flow of information between the database server, viewer client, and simulation agents.

Figure 5.11: An example interaction between a user, the radiosity agent, and the tapestry agent.

solver into our heterogeneous system, the resulting system as a whole provides a richer interactive experience.

5.7 The Generic Metasimulator

It is often useful to be able to persistently store the results of a simulation. The output can be reviewed later, shown to others, or retrieved and compared to simulation results computed at different times for "what-if" comparisons. Providing this ability requires a mechanism for interacting with a simulation that has been run in the past; the data must be stored, retrieved, and played back to the client or clients. Since we already have an interface designed to help interact with simulations, it seems like an obvious choice to use all of the interfaces and data distribution mechanisms for real-time simulations, applying them to data that has been stored rather than obtaining the data on the fly from a running simulator.

Since the Citywalk simulation framework is designed to store and forward arbitrary simulation results, and is tightly coupled with the database communication code, it is a trivial manner to simply assign simulation data to a database in addition to storing it in the real-time data distribution buffer. Once they have been stored, we can create a "virtual simulator" that, rather than generating the data itself, simply retrieves the data from the database and feeds it back into the data distribution mechanism. We call this subsystem the *metasimulator*.

The metasimulator can act as a simulator of any type supported by the system. Since the data needs to be interpreted properly by the client, running the metasimulator on a data set requires the client to have the proper front-end module for the original simulator installed, to provide the UI and rendering callbacks for that specific type of data. To facilitate this, the metasimulator checks via the simulation manager on the client whether the client has the appropriate view class to render the data. However, the metasimulator itself is perfectly generic; it does not understand the data it is providing to the clients, but it can understand the volume and timestamps on the data chunks and forward them accordingly. The metasimulator has also been used as a testing tool; it can simulate both an arbitrarily fast simulator of the specified type (by feeding the chunks into the data distribution layer at full speed) or a slower, more realistic simulator (by waiting a specified amount of time before introducing the data chunks for the next timeslice).

In practice, the metasimulator has proven valuable for many tasks. For debugging and demonstration purposes, we have used the metasimulator to generate movies of the same simulation under various visualizations, and to store interesting, but very large, simulation cases to be played

back in "real time." It has also been used to simulate system response under various network conditions while providing a control for the input of the simulator into the system. To the simulation user, it allows storage and later review of interesting cases, and allows comparison of newer simulations with older simulations of the same type, under different conditions.

5.8 Overall Integration Experiences

Many individuals in two geographically separated research groups have participated in the development of this framework. Systems components of the participants' own former research efforts in this area, as well as code and models made available by others were used in the composition of the new overall environment. The basic walkthrough code and the whole database and communications infrastructure have been developed by the two aforementioned research teams via a common code base shared over the Internet. Individual code modules were properly checked out from the code control system, modified, tested and then checked back into the source code control system.

The resulting framework also runs over the Internet and may involve many different simulation machines and viewing stations. These can be of different hardware type (Silicon Graphics or PC's) and run different operating systems (IRIX, UNIX, Linux, or Windows variants).

In our experience, integration of a new simulator program that previously ran in standalone mode takes from a day to a few weeks. Much of that time is spent writing mappings and translations between the data structures used in the walkthrough environment and the data structures used by the simulators. In contrast, the communication and control processes provided by the framework are typically sufficient to support the agent with little modification.

Whether we can achieve "real-time" performance for any particular set-up depends heavily on what kind of simulations we are attempting to run, how much compute power is available for each of the simulations, and, of course, on the complexity of the world itself. The primary factor tends to be the simulator itself; a simulation that can be run in real time on a dedicated machine can generally be run in real-time in a multiuser, distributed form within our framework, since we can always give it its own machine as a simulation service. Even so, a slower simulation can often benefit from integration by taking advantage of the area of interest information available to framework plug-ins, which sometimes allow a previously non-interactive application (like radiosity) to significantly improve its response characteristics by focusing its efforts on more time-critical areas of the environment.

Chapter 6

Model Construction with Floorsketch

6.1 Motivation

Interactive visualization of large virtual models is a very challenging problem, but not a very interesting one in the absence of large, complex, and useful models. Unfortunately, construction of large databases is itself a very difficult problem, and one that has historically received little attention in the research community. Constructing and populating the original Berkeley Walkthru model of Soda Hall required two man-years of effort for a single building. The initial assumption was that it would be straightforward to translate the architectural CAD model, created by the architects as part of their design efforts, into a 3D virtual building model. The Walkthru group rapidly discovered that this approach was badly flawed. First, CAD models are made to be human readable, but are often geometrically malformed, and automatic cleanup is very difficult and error-prone. Semantic information, such as room numbers, are often not present in a computer-readable format, and different CAD primitives are used to represent the same building elements. For example, in the Soda Hall CAD plans, several different types of line, multi-line, 2D polygon, and 3D volumetric primitives were used to represent interior walls, in no particular pattern. Certain elements may also be represented in a topologically inconsistent manner, such as windows being "painted" on wall surfaces with polygons, rather than actually penetrating the wall geometry. Initially, these inconsistencies were resolved by hand and with custom software specific to the Soda Hall CAD models. Later work by Lewis on the Building Model Generator system (BMG) [67] specifically addressed the CAD-to-model conversion problem. The result is a multi-stage pipeline that resolves many of these inconsistencies (Figure 6.1), but even this work was unable to completely automate the process. BMG requires a human to go through the CAD plans and make specific partitions and layer

assignments in the file before the pipeline can handle it, and requires a fair amount of "handholding" and manual fixup during the various stages of semantic partitioning and extrusion.

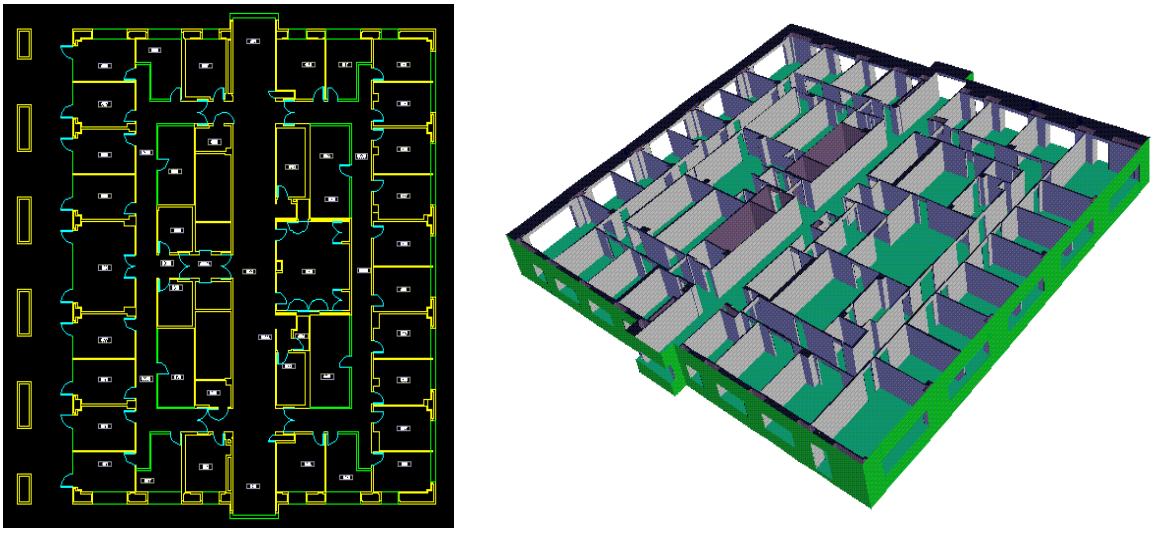


Figure 6.1: *The Building Model Generator (BMG) converts modified CAD floorplans into 3D Citywalk models, but not without substantial help from the user.*

Even if the pipeline worked perfectly, consistently, and with no human interaction, the CAD-to-model approach is impractical in many common situations. First, a CAD model needs to be available in the first place; this is not always the case, as many times we will have either a blueprint or a description of a model, as opposed to actual CAD files. Second, we need a CAD program available and a person with both CAD expertise and expertise with our software tools. CAD program licenses can be prohibitively expensive for single users, and CAD expertise can be lacking in many groups of potential users.

We have observed that, for many practical applications, extreme precision (e.g. to within an inch) is unnecessary. Real buildings are seldom that precisely related to their plans anyway (hence the large and lucrative "as-built" industry, which specializes in creating models of buildings as they stand, as opposed to what the plans say). If we had a method by which a user could "sketch" floorplans in a way similar to sketching floorplans on a sheet of paper, prototyping new models would become much easier, and more models could be created with less effort.

The Citywalk Floorsketch program is a result of these observations. Floorsketch is designed to allow modeling by approximating sketching floorplans symbolically in 2D. It assumes no knowledge of CAD, and can be used effectively with nothing more than a text description or a GIF photograph of the floorplan in question. This chapter discusses the design and implementation of

Floorsketch, and describes some results of its use with Citywalk. Floorsketch is not designed to be a complete modeling solution; rather, it is intended to be useful in situations where CAD extrusion is difficult or impossible due to poorly formed or missing CAD files, or in situations where rapid prototyping of a model is more important than high accuracy; e.g. in conceptual studies, or for conducting simulations where rapid, qualitative exploration of the problem is more important than extremely precise quantitative results.

6.2 Basic Modeling with Floorsketch

Floorsketch was designed from the ground up with two principles in mind. First, it should be as easy to create models with the program as it is to sketch floorplans on a paper. Second, the floorplan models that are generated with floorsketch should be inherently well-formed; that is, the user simply cannot create a model in floorsketch that could not be efficiently modeled in the Citywalk system. Where these two principles conflict, ease of use is chosen over strict enforcement of well-formed models, with the caveat that any malformation can be easily found, and communicated to the user or resolved by the system at extrusion time.

Cell and Portal Construction

The basic modeling operation is creating a room. This is accomplished with a single button press, which inserts a rectilinear volume into the model with a default width and height, which can be changed by the user via the preference menu. Volumes can be dragged with the mouse, or resized via their sides and corners. Dimensions can also be set via a dialog box to precise values. Volumes snap to alignment with nearby volumes to facilitate placement (Figure 6.2). They cannot be rotated, because Citywalk cannot deal with non-axis-aligned rooms.

A volume automatically has a *wall inset*. This inset represents the half-thickness of the wall between rooms; e.g. when two rooms are placed side by side, their insets add together to form the wall thickness. It is impossible to make a wall with a zero inset; thus, the user cannot create badly formed (e.g. infinitely thin) walls.

Portals are created by double-clicking on the wall inset region of a room. This creates a portal in the room against the specified wall. This portal stays inside the room where it was created, and can only be slid along the wall or widened or narrowed via the handles on its sides. Rooms are not explicitly connected via these portals; at extrusion time, the portal will automatically “punch

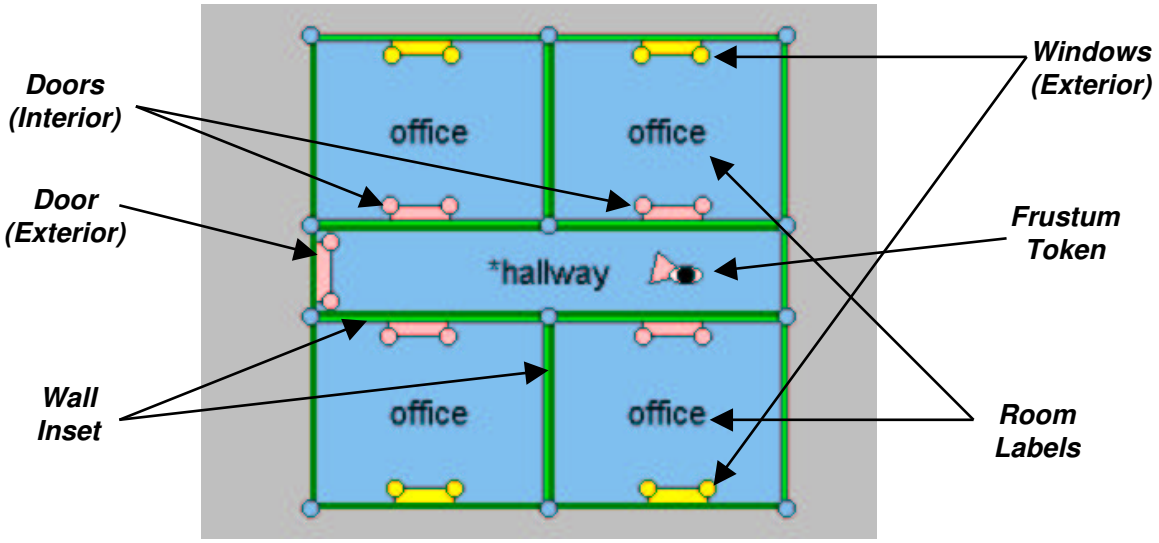


Figure 6.2: *A basic floorplan in Floorsketch, and its components.*

through" the wall to any volume that is immediately adjacent. This removes the need for the user to join rooms explicitly, and makes the interconnection process much easier. Which room the user creates the portal in is up to them; it depends on which room they would rather have the portal "travel with" when they copy or move the volumes involved. For example, in the case of a hallway with many attached rooms, the user would most likely place the doorframes in the rooms. That way, if another room was copied and nestled into the row, it would have its own door and automatically connect to the hallway. If the portals were in the hallway, they would have to be manually reshuffled to make space for the new room (Figure 6.3).

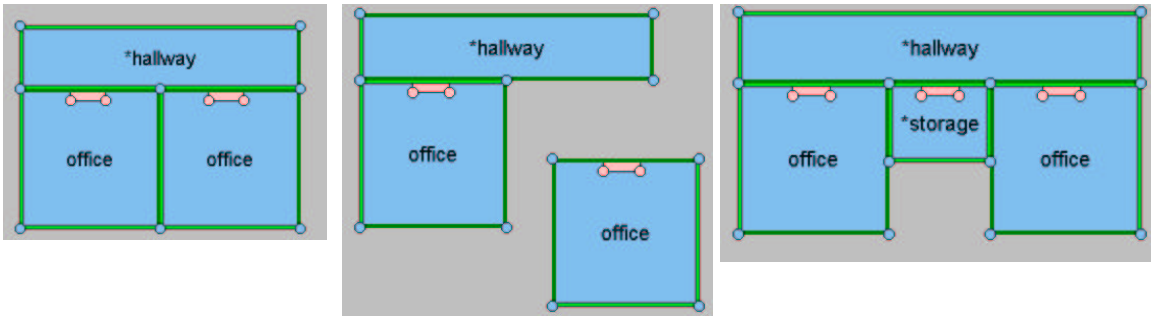


Figure 6.3: *Portals move with the room they are in; this makes adding rooms to central hallways easier, and prevents the user from having to track two-sided entities if the floor layout is modified.*

Portals come in three semantic types (Figure 6.4). Windows are the generic type, with a

sill and soffit height and position on the wall. These appear in yellow on the floorplan. Doors are different only in that their sill is at floor height; these portals appear in red on the plan. Finally, there is a special portal type called the "full wall" portal. This portal is the exact dimension of the inset wall in both sill, soffit, and width; the effect is to "punch out" the entire wall seamlessly into the adjacent volume. Full wall portals appear in dark purple on the floorplan, and are used to combine multiple rectilinear basic volumes into a larger, non-rectilinear volume, such as a hallway that turns several corners.

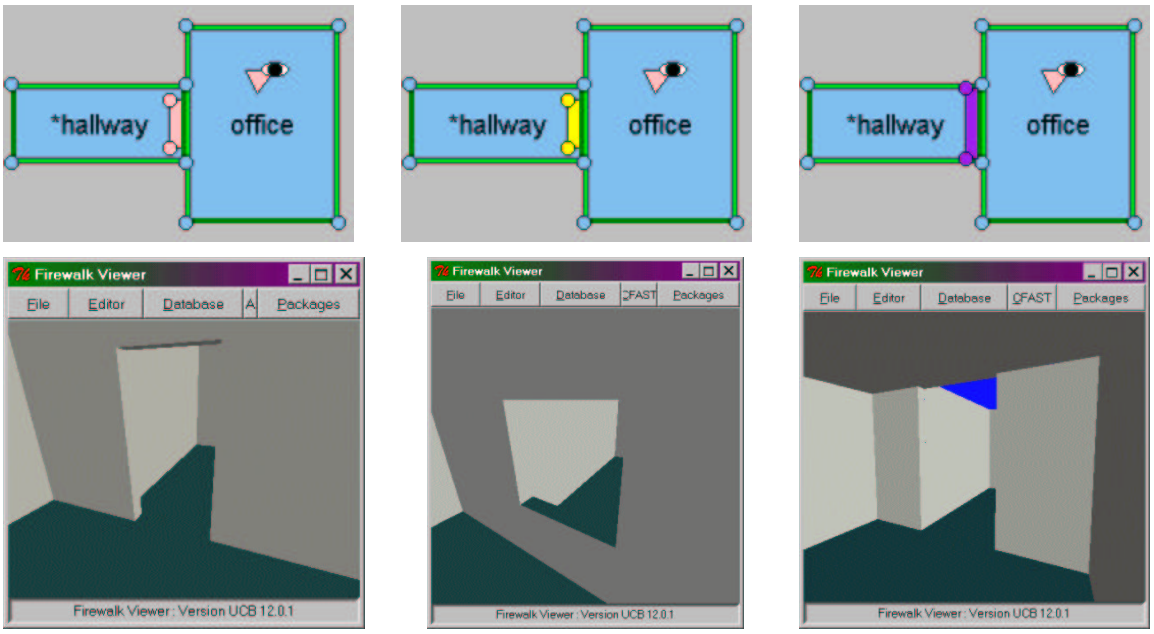


Figure 6.4: *The three flavors of portal ("Door", "Window", and "Full Wall"), and how they extrude into 3D from different 2D configurations.*

Tokens

Floorsketch provides the ability to define *tokens* that can be instanced and placed in volumes on the floorplan (Figure 6.5). Tokens have an associated 2D icon as well as 3D geometry, and can be dragged and dropped on or between rooms in the plan. They cannot be dropped outside of a room, and they move with the room they are in if that room is dragged elsewhere. Tokens normally correspond to furnishings, but there is also a special token that cooresponds to the user's eyepoint, which can be placed in the model to indicate where the user should start when loading the model for the first time.

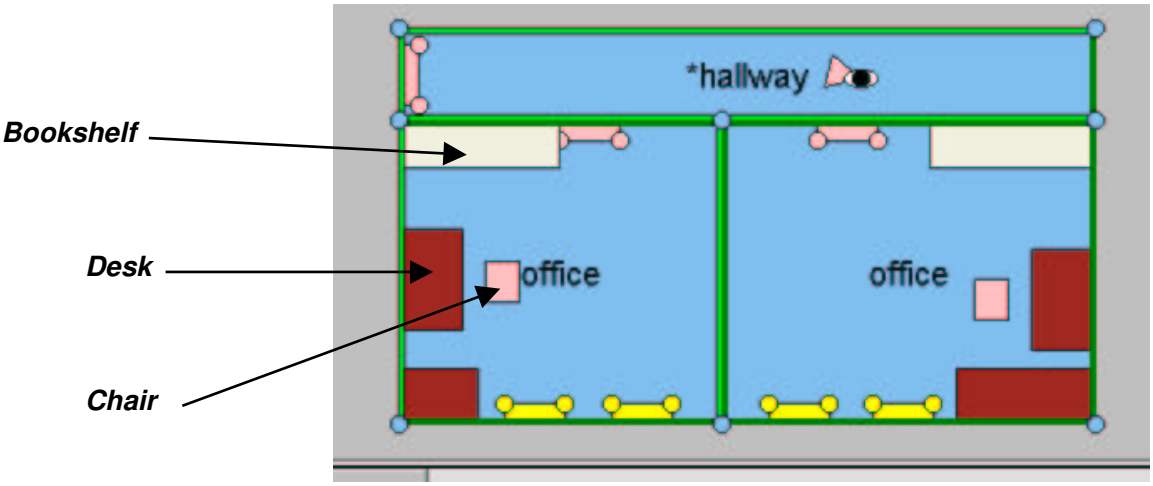


Figure 6.5: *Rooms in Floorsketch, populated with tokens representing furniture and view frustums.*

Backgrounds

The ability to create and manipulate cells and portals is useful for rapidly sketching out simple plans or for creating plans from written descriptions of rooms and their interconnections. However, it is often the case that the user has a digitized picture of a floorplan (in GIF or JPEG form, for example) and wishes to turn that floorplan into a Citywalk model.

In this case, rather than painstakingly measuring the size of volumes from the floorplan, the user can load the image into Floorsketch as a background. This includes scaling the image to an appropriate size to calibrate the measurement units in pixels to the scale of the floorplan. Once this is done, the user can simply sketch out volumes and portals directly onto the image, lining them up visually to easily create a fairly accurate model of the building (Figure 6.6).

6.3 Extrusion

Verifying Floorplans

There are only two ways for a floorplan to be invalid in Floorsketch. The first way is if two volumes overlap; allowing temporarily overlapping volumes was deemed a necessary compromise, because during floorplan construction it is often convenient to "pile up" a bunch of rooms, or to have temporary overlaps when the user has adjusted one room but not the adjacent ones. Enforcing non-overlap would unnecessarily constrain intermediate configurations in annoying ways. The

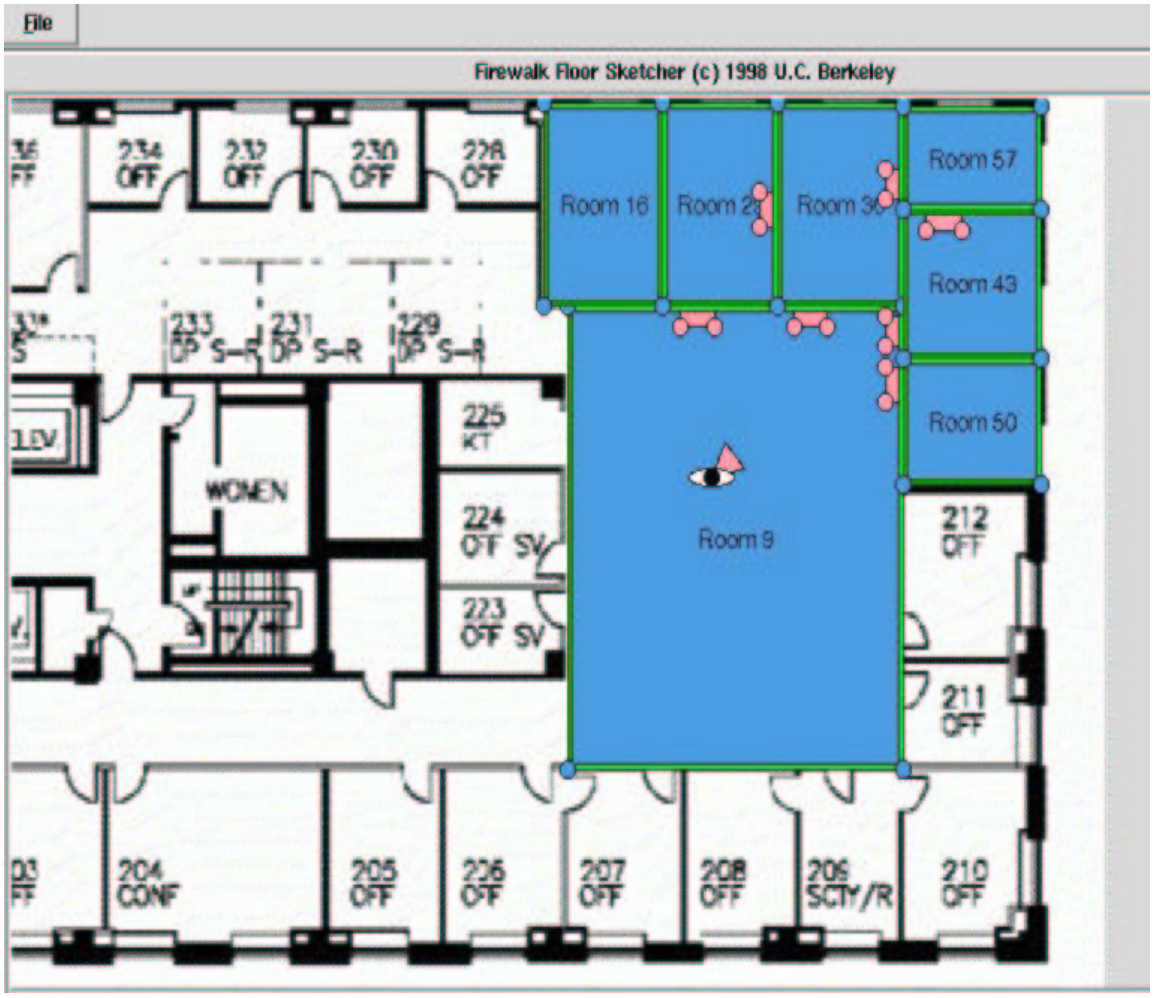


Figure 6.6: *Using a JPEG image in the background, the user can more easily "trace" an existing floorplan into a 3D model.*

second way to make an invalid plan is to have a portal in one room that, when punched through the intervening wall, overlaps a portal, wall inset, or volume border in the adjacent room. Again, this was allowed because enforcing the rule during intermediate stages of construction would be annoying to the user.

In order to guarantee that the plan is well formed at extrusion time, Floorsketch provides a "verify" button that checks for instances of these two cases. If it finds one, it tells the user exactly which room and portal caused the error and highlights it on the sketch, making it easy to fix.

Inferences and Adjacency

Several aspects of the resulting model are inferred from the adjacency of volumes in the floorplan at extrusion time. The most obvious example of this is the portals, which do not actually connect two volumes until the extrusion takes place. In addition, any section of volume wall that does not have an adjacent volume is considered to be bordering on the "outside" (Figure 6.7). Thus, those sections of wall are tiled with outward-facing polygons in the resulting model, and any portals that punch through to outside are connected with these exterior walls.

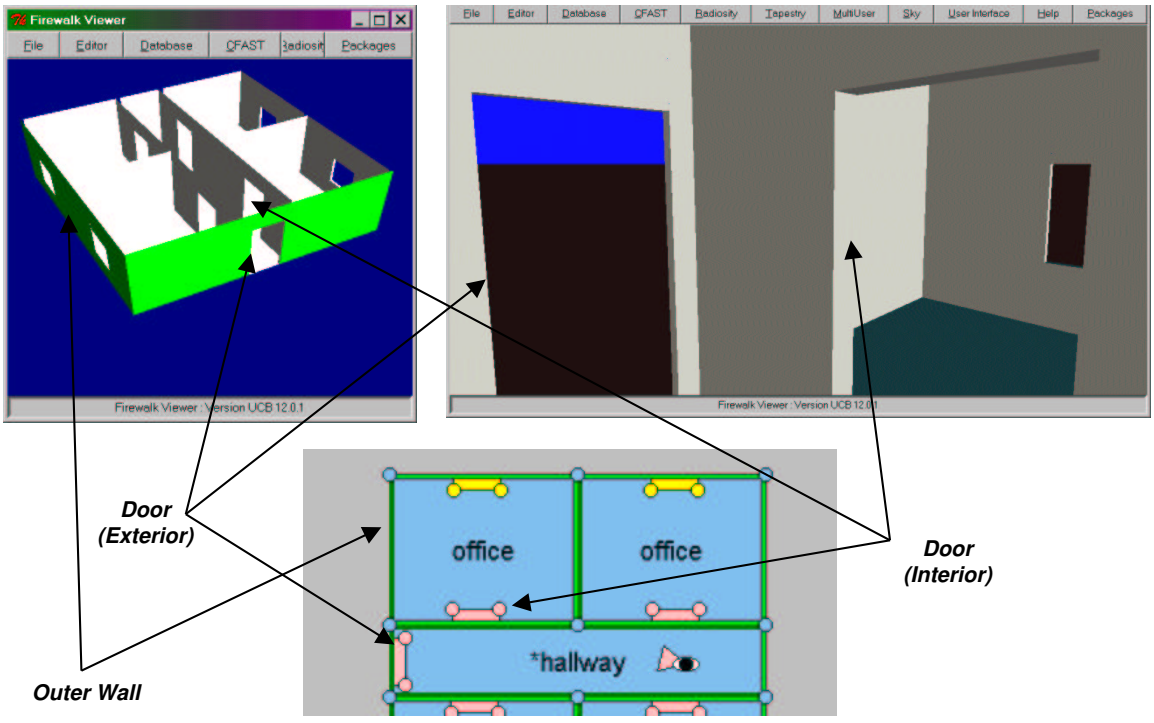


Figure 6.7: *Example of portals that lead to other volumes vs. portals that lead to the outside.*

Positioning the Model in World Space

Often an extruded floorplan is being generated to be inserted into a larger model. To facilitate this, Floorsketch takes a global transformation as input that positions the extruded model in an arbitrary world coordinate system. This is used, for example, to allow construction of multistory buildings; each floor is globally transformed in Z so that it lines up vertically with adjacent floors, and the X and Y transformations and rotation position the entire stack in world coordinates.

Tokens and Insets

Each token type is assigned a Citywalk model name in the Floorsketch configuration. At extrusion time, tokens on the floorplan are instanced into the output file with the appropriate transformation to position them in the model as they appear on the floorplan. The bottom Z coordinate of the token is aligned with the floor of the volume.

Each portal can also specify an inset model. These models refer to a master instance in the output file, appropriately transformed, rotated, and scaled such that they fit exactly in the volume of the portal when the model is compiled. Inset models can be composite models; for example, our standard inset door model has two parts, the frame and the door panel, and when inserted into the model, the appropriate object associations are activated so that the door can be naturally opened and closed about its frame hinges.

Extrusion Output

The result of the extrusion process is a set of Unigrafix (UG) files describing the floor, suitable for compilation with the Citywalk model compiler into a binary 3D model database. The user has the option of generating a "build file" along with the UG file; this is a Citywalk-readable script that will execute all the steps to build the floor when run from the Citywalk console. The extruder can also output a *semantic description* file; this file contains the user-assigned names and bounding boxes of the rooms on the floorplan, so that if these volumes are split by the visibility engine, an application can rederive the set of cells that corresponded to an original room in the floorplan.

6.4 Advanced Applications

Multi-Story Structures

Floorsketch is sometimes used to create multi-story structures, and provides several facilities to assist in that process. One of the global settings in a floorplan is a *ceiling flange* and a *floor flange*. Any volume can be instructed to leave off the floor or ceiling, so that that volume can form a vertical shaft with either or both of the floors adjacent to it. When a ceiling or floor is removed, the walls are extended vertically by the ceiling or floor flange height, respectively. Exterior wall segments are also extended vertically with the ceiling and floor flanges. These flanges allow the floors to stack properly, with a nonzero interstitial space between them, while having exterior walls and vertical shafts seamlessly interconnect with each other vertically.

Of course, it is critical that the shafts line up with each other to properly mesh, so Floorsketch provides for the ability to "pin down" volumes once they are created. A pinned volume cannot be dragged or resized. Typical usage of Floorsketch is to either create the vertical shafts on a floor first, then pin them down and copy the file for use as a template for the adjacent floors; or, alternatively, completely build one floor, pin down the vertical shafts, then copy the file and erase or edit the adjacent floors appropriately. This guarantees that shafts will properly meet.

Once the floors are extruded individually, creating the entire building is a simple matter of concatenating all of the Unigrafix files together and compiling them as one unit. This results in a single model that contains all the floors; if they are properly stacked with the global offset function, they will mesh perfectly in the output model.

Using Floorsketch for Visualization

In some cases, it is useful to view the output of a simulation or the position of the user's eyepoint in a model in real-time on a 2D version of the floorplan; Floorsketch provides a convenient interface for doing so. When Floorsketch is running alongside a Citywalk client, the user can load a model of the floorplan they are currently visualizing and connect the Floorsketch instance to it. This results in an animated version of the User Eyepoint token on the floorplan that shows the user's eye position in the model in real time. If the model was generated directly from Floorsketch output, they are automatically aligned properly.

Furthermore, it is possible to write a simulation view client that renders to the Floorsketch model rather than rendering to the frame display of the 3D viewer. We have demonstrated this

capability with a CFAST visualization client that runs from Floorsketch and displays heat levels and detector activations in real time on the Floorsketch model rather than drawing the conditions into the 3D view.

6.5 Results

Floorsketch was not intended to be a complete solution to model construction; it was meant to make model prototyping easier in a set of very specific circumstances. These included building a model on limited information (such as a simple list of rooms and interconnections with rough dimensions), building quick prototypes to perform simulations in, and quickly generating approximately correct floorplans without CAD files. Small, single-floor models are very quick and easy to create; people often make models with a dozen rooms in a matter of minutes. Larger models have shown an equal degree of success; we were able to construct an interior model of MIT's 10-floor Laboratory for Computer Science (LCS) in about 30 minutes per floor, or about 5 hours (Figure 6.8). This model was created by using scans of the LCS blueprints as backgrounds in Floorsketch, using the punch-out and stacking abilities to model stairwells and elevator shafts, and using the global positioning mechanism to place the model within a reconstructed rectangular shell on a terrain model calibrated to world coordinates via GPS. The shell was built from photographic data and was only used to get the "cornerstone" coordinate for the global positioning of the Floorsketch model. The result was a stacked interior model of the building that fit within the shell to within 6 inches on all sides, populated with openable doors. In order to perform the same task with the older Walkthru tools, the AutoCAD models of the floors would have been necessary (and these were not readily available to us) and extensive processing, both human-assisted and automated, would have been necessary to massage the CAD file into a 3D floorplan.

Floorsketch requires no setup, no CAD model or available CAD program (which can be very expensive), and no knowledge of CAD. This makes it invaluable in cases where limited source data is available or a model is needed quickly. The Floorsketch models may not be a completely accurate representation of the underlying plan; they are typically several pixels off, due to user positioning error, typically corresponding to an inch or two. On the other hand, if the building wall centers lie on a fixed grid, then the Floorsketch model could be *more* accurate, given cursor snapping and the inaccuracies in the CAD model. Floorsketch also cannot represent angular, off-axis, or more complex structural geometry; this shortcoming reflects the fact that it was designed specifically to construct Citywalk models, which need primarily rectilinear geometry due to the

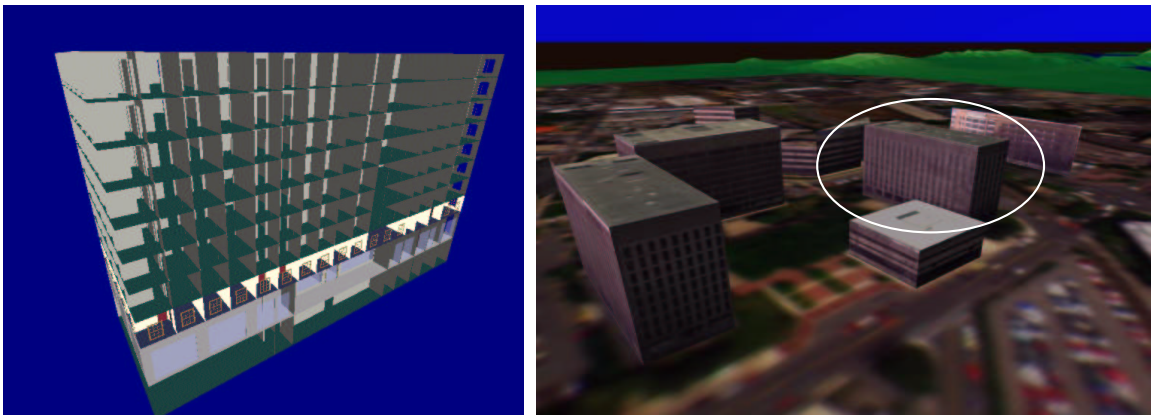


Figure 6.8: *The 12-story MIT Laboratory for Computer Science (LCS), modeled in Floorsketch from JPEG images of its floorplans in less than 1 day. Left, the stacked floors extruded from Floorsketch. Right, the exteriors of the buildings in Tech square (LCS building is circled). The interior fits inside the exterior shell to within 6 inches on all sides.*

underlying KD-tree based cell structure.

The one major problem we have had with Floorsketch is the difficulty of maintaining a network of non-rectangular rooms that represent a more complex room. In these cases, it would be easier if Floorsketch provided the ability to do arbitrarily shaped axis-aligned geometry for volumes, rather than just rectilinear geometry. Were we to undertake a redesign of Floorsketch, we would most likely attempt to represent more complex rectilinear geometry for volumes.

Chapter 7

Discussion

7.1 Architectural Analysis

There were two major goals for this project. First, we wanted to extend the functionality and scope of the Berkeley Architectural Walkthru project by providing a foundation for much larger, distributed, multiuser systems, which could combine multiple different types of model, different visibility approaches, and different forms of data into one database while maintaining the interactive performance the first generation walkthru provided. Second, we wanted to provide a framework by which physical simulators written by other research groups could be integrated into the virtual environment to enhance the utility of both systems.

The resulting system is essentially a two-tier model that dynamically distributes environment information via two channels: a high-performance, intelligent, direct channel that automates management of time-critical data (e.g. the simulation data distribution stratum), and a database channel that provides persistence and fine grained data sharing abilities at the cost of additional latency (e.g. the database layer).

Why the two tier architecture?

Given an object database server that both simulation client and simulation agent can communicate with, it would be possible to have the simulator simply write simulation results directly into the database and have the client read the data back out as it needed it. This approach would work with a sufficiently fast database and network, and sufficient prefetching of data by the client, a simulator that works sufficiently "ahead of time," and a sufficiently fast network. However, this

approach greatly increases the amount of traffic required of all the participants. First, the fact that there is an additional node in the communication (e.g. the database server) adds at least one "unit" of latency to the process. Second, the "dumb" server approach requires additional communication between the database server and client, because the client needs to actively request not only the data itself, but also the indexing structures that allow it to discern which data it needs. Conversely, a smarter server (such as the simulation manager) can proactively send data it knows the client needs by processing the indexing structures locally. Third, this increases processing time on all participants; the client must now process the indexing information and database notifications as well as the simulation results, the simulation server must construct indexing structures that allow the client to perform this task efficiently, and the database server must process the entire set of interactions with both agents. Fourth, this approach is less scalable as it is locked into a star topology with a database server as the center; intelligent simulation managers can operate in other topologies, such as the ring topology that has been established to be better for simulations in many cases (e.g. Funkhouser's RING work).

Simulation Coupling

The two-tier architecture effectively addresses high speed, read-only coupling and low speed, interlocked coupling, with free interaction of data between the layers (for example, the physics simulator creates dynamic information that is normally distributed through the high-speed direct layer, but that information can also be committed to the database and thus become visible through the lower layer to clients not directly connected to the simulator). However, we have not directly addressed interactions that require real time performance and tight interaction between agents. A (somewhat contrived) example might be to have two physics simulators operative in the same volume on different object sets, then have those objects interact with each other "across" the two simulators. Each system's computation speed would be limited to the latency of sending each step of data to the other system. In the worst case, such interactions between complex simulations would likely lead to instabilities or very poor performance.

Unfortunately, it is not clear that there is any general-purpose solution to this problem. There is an entire field of study that focuses on partitioning computations efficiently between processors, and that field has many and varied approaches to such partitioning. As such, we are forced to leave simulations that require very tight coupling to the user who is writing to our API; they will need to select the phase of computation at which they can afford to hand the data over to our

framework for general distribution.

In this vein, it is interesting to note that, for the simulators that we have integrated thus far, such partitionings can actually be done relatively efficiently in our system, even replicating efficient techniques known in the literature for those problems. For example, in closed environment radiosity, one can achieve a good solution between separate radiosity agents by partitioning the problem along the boundaries between contiguous sets of volumes, each set of volumes having a dedicated radiosity agent, and having the agents treat the adjacent volume's results as constant for several iterations of their local solution. This is very compatible with having each agent act as a simulation client to neighboring agents, updating the "constant" state of the other room periodically via simulation data updates, and generating internal updates quickly within their volumes [43]. Similarly, for physics agents, a similar partition would lead to objects being "handed off" between agents as they cross through portal regions between local sets of volumes. These two problems, distributed radiosity and distributed physical simulation, would admit similar optimizations and a similar style of simulator-to-simulator coupling through our framework. We believe these examples speak to the generality of the partitioning of data by volume, which is the basis of our real-time data distribution framework.

Why not use existing common object protocols?

A fairly obvious question to ask is why we didn't simply use off the shelf database and distributed object frameworks to build our system. We covered the technical reasons for using our own database instead of a commercial one in chapter 3. In brief, we feel that the additional capabilities and nominally optimized nature of the commercial databases do not offset the cost of using a foreign and unmodifiable code base.

There are two major common object protocol standards in existence that provide somewhat similar functionality to the network tier of our system; COM (the Common Object Model, from Microsoft, and in common use in the various versions of Windows [68]) and CORBA (The Common Object Request Broker Architecture, which is an open standard under development [69]. These systems provide distributed objects with remote procedure call (RPC) and interface abstractions, and allow for platform independence of objects (and, in some cases, language independence). However, they are relatively "dumb" protocols, and as such would only serve to replace the communication and object serialization portions of our framework. Furthermore, that serialization and communication process is "hidden" behind the COM or CORBA API, and as such we would nei-

ther be able to use the packing and marshalling mechanisms to store the objects in a persistent store, nor would we have high-level control over the bandwidth usage of the system. Finally, there are the practical issues; using COM would tie us exclusively to Windows platforms, and there are no good implementations of CORBA in existence, since it was a "design by committee" system with no party responsible for the actual implementation.

7.2 Relationship to Existing Techniques

An important measure of success of a framework system is how well we can replicate the functionality of existing visualization and modeling systems efficiently within this second-generation system architecture.

7.2.1 Database Techniques

Basic architectural walkthrough functionality, as provided by the first-generation Berkeley Walkthru and UNC walkthrough projects, is well incorporated into the system. However, the database is strictly superior to that of the first-generation Walkthru, incorporating not only the ability to do arbitrary swapping of model components into and out of memory, but also incorporating the tools needed to manage simultaneous viewing and editing of the model (e.g. locks, watches, and transactions).

Real-time city walkthroughs can incorporate such a large quantity of data as to require databases to be distributed among multiple large servers that can stream data to the clients [22]. Our second-generation architecture is designed to fully support this mode, with event-driven prefetching of objects from remote databases, and the ability to connect any client to any number of database services simultaneously.

In general, there are three possible approaches in sharing a world model: replicate the model entirely on each client and disallow changes to it [24, 28], "centralize" the model and have all clients attach to it [22], or use a truly distributed environment with clients each acting as local databases, replicating aspects of the world model to other nearby clients [70, 71]. The first of these approaches is simple but yields minimal interactivity; of course, replicating a database at each client is simple for any system, and requires only the simplest database technology. The second approach requires a database that provides full multiuser semantics (locking and watches) which is provided by our second-generation database. The last approach requires both multiuser semantics and the

ability to attach, detach, and interact with many different databases at once, on different machines, simultaneously. We also support this mode of operation with server-client mode and the ability to provide database services from any node in the network to any other node. Thus, we can claim to support all of these modes of operation with our second-generation architecture.

7.2.2 Communication and Interaction Techniques

Communication techniques used by various shared environments include both point-to-point and multicast techniques. The former is more common; the latter is typically usable only with tightly coupled clients [24]. Our second-generation system simulation data network API explicitly provides for multicast semantics to system clients, so that in our data distribution network can support multicast approaches to provide low-latency distribution of local model elements.

Consistency can be maintained loosely, with voting approaches [70], or strictly enforced with more traditional locking mechanisms. Again, we can support both approaches with global object identifiers provided by our database layer and point-to-point and local multicast provided by the data layer, and local locking and watch semantics provided by the second-generation database.

7.2.3 Distributed Simulation Techniques

Most work in this area has been concerned with simulations that are highly localized to peer-to-peer interactions between small groups of actors [24, 29]; as far as we are aware, Citywalk is the first system to address distribution of large-scale environmental data. However, a simple extension of the volume tagging scheme would result in the Citywalk data distribution network providing the same functionality as a peer-to-peer system; indeed, our multiuser simulation agent replicates a peer-to-peer system quite well already.

7.3 New Directions

7.3.1 Simulation Triggering

In the current Citywalk system, the user must explicitly trigger simulations through some mechanism provided by the UI plugin for the desired simulator. Of course, the real world doesn't work that way; physics are always present, and one only needs to take an appropriate triggering action (e.g. pushing a book off the desktop, or throwing a smoldering match into a wastebasket full of paper) for the "simulation" to begin. It seems natural to ask the same of the virtual environment;

when an action "appropriate" to a given simulator is performed, that simulator should transparently begin running and presenting its results in the current environmental view.

While such behavior would certainly contribute to the *realism* of the environment, it poses a number of difficult issues. We have discovered in the process of designing a general-purpose framework that the type of data required by and generated by the simulator cannot be predicted *a priori*. Fire simulators require materials and chemical information; dynamics simulators need mass and moment of intertia. Fire simulators produce atmospheric conditions and thermal information; dynamics simulators move objects in the same way a user might. In response to these varying needs, both the incoming and outgoing information for a simulator type must be specified by the user who is integrating the simulation into the framework, and the framework assumes very little about the form of the data. Simulation triggering has the same problem. The type of event that would or should trigger a simulation varies with the type of simulation as well as the situation the user is in. Throwing an object onto another object may or may not be a fire event, based on the composition and state of the objects in question (a lit match onto paper triggers a fire; an unlit match onto paper does not; a book onto a table does not, unless the book is already on fire and the table is flammable, etc.). The biggest question is, can we abstract anything out of the process of triggering a simulation that would allow us to provide any assistance to the simulation integrator at all? If the answer is no, then there is no way to improve upon the current situation, which is to have the UI module for a simulator or the simulator itself monitor world state via watches and client callbacks for any events that interest it (the simplest possible case being watching for the user to press a "start fire" button).

A second obstacle is the fact that there are also conditions under which a simulation may be called for in a strict physical sense, but may not be desired in a *teleological* sense [37]. That is, the user may be moving a burning match through an intermediate position when it contacts a flammable surface, and the user does *not* want a simulation to trigger, because they are setting up an entirely different situation. This observation proved critical to the implementation of the Object Associations editing framework, and has serious implications for an automated triggering mechcanism. Simulations can be very resource-intensive, and starting one against the user's wishes can be costly in terms of system response. At the very least, the user requires a means of deactivating certain types of triggers while they are in intermediate states of world manipulation.

7.3.2 Temporal Navigation and Representation

Representation of time is interesting from both a data representation point of view and a navigation point of view. Navigation in space is both natural to humans, and widely explored in the HCI literature and in industry. Navigation in time is more difficult; the only intuitive analog most people have to temporal navigation is recording devices such as VCR's and tape decks. Unfortunately these interfaces are one-dimensional and provide little interface to having multiple time axes available at once to the user, as is the case when the user may have simultaneous access to many virtual-time simulations, both ongoing and previously recorded. Navigating multiple time axes at once is useful for the purposes of comparing the results of simulations as well as synchronizing different simulators operating simultaneously. We need better mechanisms and interfaces for such navigation.

This leads naturally to the representation and interaction between these multiple timelines. Consider a user opening a door in the Citywalk environment. Do they mean to physically simulate the interactions of the door with the environment? What about the interactions of the newly opened door with running simulators? Should those simulations be restarted with the door open, or should the door be considered to be opened at the virtual time being viewed at the moment? What if the simulation is paused or being directly manipulated? These questions all relate to how the time axis is handled in a simulation-enriched model, and we do not have good, general answers at this point.

7.3.3 Integrated Rendering Frameworks

The system that we have described and implemented allows us to navigate and run simulations in a collection of buildings that fit into a single homogeneous database. One clear avenue for future evolution of walkthrough systems is an extension to integrated systems that involve models of different kinds and an expansion to models of wider scope and larger scale. Many groups have independently built virtual worlds with very sophisticated machinery for visibility culling, LOD selection, efficient collision detection, and other simulation tasks. This machinery is often tied very closely to the internal structure of the particular walkthrough system. For example, NPSNet [24] uses different basic structures from the downtown LA model [22], the UNC coal-firing plant [20], or the Berkeley Soda-Hall Walkthru model [5].

Conceptually, the simplest approach to combining such models into a virtual world would be to convert all data into a single walkthrough model format and to use one set of tools to navigate it. However, such an import task could be impractically large, and there might be primitives that

translate poorly and structures that would be lost. It is thus preferable to use these models as they were designed, with their abstractions and machinery in place.

We propose to introduce another level of abstraction to the world model. Rather than merely considering scene graph objects, cells, and actors, we need to add the concept of a walkthrough space, which has its own visibility culling and rendering methods, and which may reside on a remote system. We will let the individual walkthrough systems handle the rendering of their model worlds, since they have the appropriate machinery, and we will create a new layer of integration via a general communication interface designed to handle rendering queries between these different heterogeneous systems.

The simplest form of a query will consist of a view frustum relative to the coordinate frame of the child space within the integrated heterogeneous model. A single view of a scene will result in a recursion into all spaces that are visible in the view. Each space will collect up visible geometry from its contained spaces, which will be returned in the query response to its parent space. The rendering program will gather up all of the geometry and impostors from all spaces that are visible on each frame and render them together.

The simplest interface offers occlusion culling only in the most rudimentary form: that derived from the view frustum. Almost every virtual reality system provides more advanced mechanisms for culling hidden objects. If we want to provide such mechanisms at the highest level of abstraction, our interface must be capable of transmitting occlusion information. In order to achieve this goal, we must devise a format to describe occlusion information in the form of a generic visibility structure, for example, as a portal tree plus occluders. Information in this format may then be transmitted for a particular view across multiple types of walkthrough models. For example, we may use a cell and portal visibility scheme within a building [14, 5], and a cull horizon for looking across a city [72]. A single view from within a building may include occlusion specifications from both of these mechanisms at once.

So far, we have only described an interface for *rendering* of heterogeneous models. Clearly, before too long people will demand all the same capabilities that are now available in second-generation systems: simulation and interaction. This will necessitate transmittal of corresponding information (forces, temperatures, light flux, etc.) through the interface. This is an open grand challenge: how to deal with interactions that go across the seams of such a heterogeneous world. Opening up simulation across the boundaries of these models may require a very large bandwidth of communication, and interactive simulations may be difficult to achieve at interactive rates with the long feedback delay inherent in distributed systems.

7.4 Summary

We have presented the salient features of Citywalk that we believe to be representative of second generation virtual walkthrough environments. Our environment combines many of the techniques that were individually developed on several different first generation walkthrough systems. Its architectural framework is built on top of an object-oriented database management system and makes use of an intelligently buffered communication layer. These abstractions enable a fairly platform independent system and make it easier to distribute its functionality over many different computing sites. Different interaction and simulation engines can be added to the environment in a modular manner, as demonstrated by the six agents described in this paper.

We conclude with a vision of a third generation framework that would allow us to combine walkthrough systems with different organizations into a heterogeneous world model, in which the various model spaces communicate through a standardized interface that handles suitably abstracted and extended rendering requests. We also propopse continuing work in navigation, user interfaces, and more intuitive and useful triggering of simulation agents in the framework.

7.5 A Final Thought

Much of the power of the framework we have built is in its expandability, as a basis for future work in large virtual environments. There has been a large amount of recent work in acquisition of very large city models, and we believe that integration of the many visualization techniques that are available in the research literature, as well as physical behaviors as explored in this thesis, will be necessary to fully realize the utility of these large models. Integration is not a very well-explored issue in the community, as it leans more towards "engineering" than "science," and so suffers from being, in a sense, less "publishable." Integration of these various approaches into a seamless whole is a large problem, and one that we believe is worthy of study in its own right. Hopefully we have provided the groundwork for both this research group, and our collaborators at MIT and here at Berkeley, to continue this process and to produce some truly large-scale and highly integrated virtual environments that provide both realism and a high level of utility for real applications.

Bibliography

[1] Richard W. Bukowski and Carlo H. Séquin. Interactive simulation of fire in virtual building environments. In *Computer Graphics (Proceedings of SIGGRAPH 1996)*, August 1997.

[2] G. Ward. The radiance lighting simulation and rendering system, 1994.

[3] Taisei Corporation. Yebisu garden palace. Video, 1994.

[4] T. A. Funkhouser, I. Carlborn, G. Elko, M. Sondhi, and J. West. A beam tracing approach to acoustic modeling for interactive virtual environments. In *Computer Graphics (Proceedings ACM SIGGRAPH '98)*, pages 21–32, July 1998.

[5] Thomas A. Funkhouser, Seth J. Teller, Carlo H. Séquin, and Delnaz Khorramabadi. UCB system for interactive visualization of large architectural models. *Presence: Special Issue on Teleoperators and Virtual Environments*, 5(1):13–44, Winter 1995.

[6] P. Reneke et. al. R.D. Peacock, G.P. Forney. CFAST, the consolidated model of fire and smoke transport. 1993.

[7] G.T. Chou and S. Teller. Multi-level 3d reconstruction with visibility constraints. In *Proceedings of Image Understanding Workshop*, 1998.

[8] S. Coorg and S. Teller. Extracting textured vertical facades from controlled close-range imagery. In *Proceedings of CVPR*, pages 625–632, 1999.

[9] M. Bosse, D. De Couto, and S. Teller. Eyes of argus : Georeferenced imagery in urban environments. In *GPS World*, pages 20–30, April 2000.

[10] Bruce J. Schachter. Computer image generation for flight simulation. *IEEE Computer Graphics & Applications*, 1(5):29–68, 1981.

[11] B. J. Schachter. *Computer Image Generation*. John Wiley, New York, 1983.

[12] Seth Teller. *Visibility Computations in Densely Occluded Polyhedral Environments*. PhD thesis, Dept. of EECS, University of California at Berkeley, October 1992.

[13] M.J. Zyda, D.R. Pratt, J.G. Monahan, and K.P. Wilson. Npsnet: Constructing a 3d virtual world. In *ACM SIGGRAPH Special Issue: 1992 Symposium on Interactive 3D Graphics*, march 1992.

[14] John M. Airey. *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations*. PhD thesis, Dept. of CS, U. of North Carolina, July 1990. TR90-027.

[15] Frederick P. Brooks, Jr. Walkthrough — A dynamic graphics system for simulating virtual buildings. In Frank Crow and Stephen M. Pizer, editors, *Proceedings of 1986 Workshop on Interactive 3D Graphics*, pages 9–21, 1986.

[16] R. Deyo, J. Briggs, and P. Doenges. Getting graphics in gear: Graphics and dynamics in driving simulation, 1988.

[17] M.R. Macedonia, D.P. Brutzman, and M.J. Zyda et al. Npsnet: A multi-player 3d virtual environment over the internet. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 93–94, April 1995.

[18] Christer Carlsson and Olof Hagsand. DIVE — A platform for multi-user virtual environments. *Computers and Graphics*, 17(6):663–669, November–December 1993.

[19] Thomas Funkhouser. *Database and Display Algorithms for Interactive Visualization of Architectural Models*. PhD thesis, Dept. of EECS, University of California at Berkeley, September 1993.

[20] D. Aliaga, J. Cohen, A. Wilson, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stuerzlinger, E. Baker, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. MMR: An interactive massive model rendering system using geometric and image-based acceleration. In *Proceedings Symposium on Interactive 3D Graphics*, pages 199–206, April 1999.

[21] J. Barrus, R. Waters, and D. Anderson. Locales and beacons: Precise and efficient support for large multi-user virtual environments. *Proceedings of VRAIS'96, Santa Clara CA*, pages 204–213, 1996.

[22] W. Jepson, R. Liggett, and S. Friedman. An environment for real-time urban simulation. In *Proceedings Symposium on Interactive 3D Graphics*, pages 165–166, 1995.

[23] P. W. Maciel and P. Shirley. Visual Navigation of Large Environments Using Textured Clusters. In *Proceedings of the Symposium on Interactive 3D Graphics*, pages 95–102, 1995.

[24] M. J. Zyda, D.R. Pratt, J.G. Monahan, and K.P. Wilson. NPSNET: Constructing a 3d virtual world. In *Proceedings Symposium on Interactive 3D Graphics*, 1992.

[25] M. Macedonia, M. Zyda, D. Pratt, D. Brutzman, and P. Barham. Exploiting reality with multicast groups: A network architecture for large-scale virtual environments. *Proceedings of VRAIS'95*, 1995.

[26] Mingyu Lim and Dongman Lee. Improving scalability using sub-regions in distributed virtual environments.

[27] T. A. Funkhouser, C. H. Séquin, and S. J. Teller. Management of Large Amounts of Data in Interactive Building Walkthroughs. In *Proceedings Symposium on Interactive 3D Graphics*, pages 11–20, March 1992.

[28] Thomas A. Funkhouser. Ring: A client-server system for multi-user virtual environments. *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 95–92, April 1995.

[29] J. Cremer, J. Kearney, and H. Ko. Simulation and scenario support for virtual environments. *Computers and Graphics*, 20(2):199–200, 1996.

[30] S. Doi, T. Takei, and Y. Akiba et. al. Real-time visualization system for computational fluid dynamics. 37(1):114–123, January 1996.

[31] J. Cohen, M. Lin, D. Manocha, and M. Ponamgi. I-collide: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 189–196, April 1995.

[32] B. Mirtich and J. Canny. Impulse-based simulation of rigid bodies. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 181–188, April 1995.

[33] S. Bryson. The virtual windtunnel: A high-performance virtual reality application. In *IEEE Virtual Reality Annual International Symposium*, pages 20–26, 1993.

[34] S. Bryson. Virtual reality in scientific visualization. *Communications of the ACM*, 38(5):62–71, May 1996.

[35] T.A. Defanti, D.J. Sandin, and G. Lindahl et. al. High bandwidth and high resolution immersive interactivity. In *Very High Resolution and Quality Imaging*, pages 198–204, 1996.

[36] G. Singh. Bricknet: Sharing object behaviors on the net. In *Proceedings of the IEEE Virtual Reality Annual Symposium*, pages 19–25, 1995.

[37] Richard Bukowski and Carlo Séquin. Object associations: A simple and practical approach to virtual 3d manipulation. In *Proceedings of Symposium on Interactive 3D Graphics*, pages 131–138, April 1995.

[38] Akmal B. Chaudhri and Peter Osmon. A comparative evaluation of the major commercial object and object-relational dbmss: Gemstone, o2, objectivity/db, objectstore, versant odbms, illustra, odapter and unisql.

[39] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Computer Graphics (Proceedings ACM SIGGRAPH)*, pages 247–254, August 1993.

[40] M. Mine and H. Weber. Large models for virtual environments : A review of work by the architectural walkthrough project at unc. *Presence: Teleoperators and Virtual Environments*, 5(1):136–145, 1995.

[41] M. Kofler, H. Rehatschek, and M. Gruber. A database for a 3d gis for urban environments supporting photo-realistic visualization. *International Archives of Photogrammetry and Remote Sensing (ISPRS)*, 31, 1996.

[42] S. Chenney and D. Forsyth. View-dependent culling of dynamic systems in virtual environments. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, pages 55–58, April 1997.

[43] Seth Teller, Celeste Fowler, Thomas Funkhouser, and Pat Hanrahan. Partitioning and ordering large radiosity computations. In *Computer Graphics (Proceedings of SIGGRAPH 1994)*, pages 443–450, July 1994.

[44] J. Postel. RFC 793: Transmission Control Protocol, 1981.

[45] J. Postel. RFC 768: User Datagram Protocol, 1980.

[46] K. Gong and L.A. Rowe. Parallel mpeg-1 video encoding. In *Proceedings of the 1994 Picture Coding Symposium*, September 1994.

[47] Brian Mirtich. *Impulse-Based Dynamic Simulation of Rigid Body Systems*. PhD thesis, University of California, Berkeley, 1996.

[48] D.G. Aliaga. Visualization of Complex Models Using Dynamic Texture-based Simplification. In *Proceedings IEEE Visualization*, pages 101–106, October 1996.

[49] G. Schaufler. Nailboards: A rendering primitive for image caching. In *Rendering Techniques*, pages 151–162. Springer-Verlag, 1997.

[50] L. Darsa, B. Costa, and A. Varshney. Navigating static environments using image-space simplification and morphing. In *ACM Symposium on Interactive 3D Graphics*, pages 25–34, 1997.

[51] F. Sillion, G. Drettakis, and B. Bodelet. Efficient impostor manipulation for real-time visualization of urban scenery. In *Computer Graphics Forum (Proceedings Eurographics)*, pages 207–218, 1997.

[52] G. Schaufler and W. Stürzlinger. A three-dimensional image cache for virtual reality. In *Computer Graphics Forum (Eurographics 96)*, pages 227–236. 1996.

[53] J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In *Computer Graphics (Proceedings ACM SIGGRAPH)*, pages 75–82, August 1996.

[54] W. R. Mark, L. McMillan, and G. Bishop. Post-rendering 3d warping. In *Proceedings Symposium on Interactive 3D Graphics*, pages 7–16, April 1997.

[55] D.G. Aliaga and A. Lastra. Architectural Walkthroughs Using Portal Textures. In *Proceedings IEEE Visualization*, 1997.

[56] Thomas A. Funkhouser. Coarse-grained parallelism for hierarchical radiosity using group iterative methods. In *Computer Graphics (Proceedings of SIGGRAPH 1996)*, pages 343–352, August 1996.

[57] Andrew Willmott, Paul Heckbert, and Michael Garland. Face cluster radiosity. *Eurographics Rendering Workshop 1999*, 1999.

[58] Shenchang Eric Chen. Incremental radiosity: An extension of progressive radiosity to an interactive image synthesis system. *Computer Graphics (Proceedings of SIGGRAPH 90)*, pages 135–144, August 1990.

[59] David W. George, François X. Sillion, and Donald P. Greenberg. Radiosity redistribution for dynamic environments. *IEEE Computer Graphics & Applications*, 10(4):26–34, July 1990.

[60] Stefan Müller and Frank Schöffel. Fast radiosity repropagation for interactive virtual environments using a shadow-form-factor-list. *Fifth Eurographics Workshop on Rendering*, pages 325–342, June 1994.

[61] David A. Forsyth, Chien Yang, and Kim Teo. Efficient radiosity in dynamic environments. *Proceedings Eurographics Workshop on Rendering*, pages 313–323, June 1994.

[62] George Drettakis and François X. Sillion. Interactive update of global illumination using a line-space hierarchy. In *Computer Graphics (Proceedings ACM SIGGRAPH)*, pages 57–64, August 1997.

[63] Philippe Bekaert and Yves D. Willems. Importance-driven progressive refinement radiosity. *Proceedings Eurographics Workshop on Rendering*, pages 316–325, June 1995.

[64] Attila Neumann, László Neumann, Philippe Bekaert, Yves Willems, and Werner Purgathofer. Importance-driven stochastic ray radiosity. *Proceedings Eurographics Workshop on Rendering*, pages 111–122, June 1996.

[65] F. Schöffel. Online radiosity in interactive virtual reality applications. *ACM Symposium on Virtual Reality Software and Technology*, September 1997.

[66] François X. Sillion and Claude Puech. Radiosity and global illumination. 1994.

[67] Richard Lewis. *Generating Three-Dimensional Building Models from Two-Dimensional Architectural Plans*. PhD thesis, Dept. of EECS, University of California at Berkeley, May 1996.

[68] Don Box. *Essential COM*. Object Technology Series. Addison-Wesley, 1998.

[69] Alan Pope. *The CORBA Reference Guide*. Addison-Wesley, 1998.

[70] A. Frcon, H. J-Aro, and S. Stenius. Dive - the distributed interactive virtual environment - dive files description for dive version.

[71] R.C. Waters, D.B. Anderson, and J.W. Barrus et. al. Diamond park and spline: A social virtual reality system with 3d animation, spoken interaction, and runtime modifiability. *MERL*, January 1996.

[72] Laura Downs, Tomas Möller, and Carlo Séquin. Occlusion horizons for driving through urban scenes. In *Proceedings of Symposium on Interactive 3D Graphics*, pages 21–25, 2001.

Appendix A

Library API Reference

This appendix provides an overview of the APIs, classes, and functions implemented for this thesis.

A.1 System Library (system)

The **system** library provides low-level, C++ language-based compatibility functions to bridge gaps between operating systems and compilers, and provides enhanced error reporting facilities.

A.1.1 Compatibility Functions (*system.h, compat.[h,c]*)

These files are concerned with providing defines and function stubs that provide missing functions like bcmp/bzero (which was used extensively in the original SGI walkthrough, but is not available on Windows) to the code when compiled on different architectures.

A.1.2 Error Reporting (*errors.[h,c]*)

These error reporting functions are invaluable as a cross-platform tool, as **stderr** is not well defined across non-UNIX operating systems (e.g. Windows does not provide a standard console for all processes), and as a tool for logging errors from multiple batch runs of the program.

WKReportError, WKPrintToConsole

These functions take argument lists identical to *printf*, except that they route their output to locations other than the console. These extra routings are controlled by the additional functions, defined below.

WKReportError allows the specification of an additional error type tag as the first argument. There are several error types defined in *errors.h*, ranging from warnings, to errors, to fatal errors. Extra behavior is produced by these tags:

1. WKERR_NONFATAL: Causes the error to not only be written to the specified target locations, but it will also pop up an immediate modal dialog box that informs the user of the problem.
2. WKERR_FATAL: Will do everything NONFATAL does, plus force an immediate system exit.
3. WKERR_WARNING, WKERR_COMMENT: Prepend prefix strings to the output ("WARNING:" and "Comment:" respectively) which allows these types of message to more easily be stripped from the error log file produced by *WKSetErrorScriptFile*.

WKSetErrorScriptFile

Causes copies of the error messages to be printed to the specified stream. There is only one stream maintained at any time.

WKSetConsolePrintCallback

Causes the specified callback function to be invoked with a copy of each error message as they are generated.

wkprintf, wkeprintf

These functions are aliases for *WKPrintToConsole* and *WKReportError* respectively; they are a bit easier to type and read.

A.2 Core Class Library (gsim)

The **gsim** library provides two major sets of functions. First, it provides higher-level abstractions of common operating system facilities, including multithreading (which includes locking

and events), sockets and socket connections, and timing facilities (in floating-point seconds rather than system-specific integer fixed point formats). Second, it provides a global base class from which C++ objects can be inherited; this base class provides greatly enhanced runtime type identification (RTTI), supporting a global class factory and unique, universal integer IDs for all classes, and a serialization interface (corresponding "smart buffer" class) that provide for a common, machine-independent serialization API across all objects.

A.2.1 Socket Abstraction (*rtsocket.[hpp, cpp]*)

This code provides an abstracted, OS-independent socket interface. The two major classes are **RtSocket** and **RtSocketPort**, which are the socket object and the port object (to which incoming sockets connect) respectively. The semantics of these sockets differ slightly from the standard UNIX form; they default to an asynchronous mode of operation, allowing the user to specify a callback function which is called from a dispatch thead whenever incoming data is detected, and another callback which is invoked if an error condition is detected on the socket. This socket code does not provide any additional buffering; data must be consumed by the callbacks as it comes in.

The **RtSocketPort** object is created to listen on a port on the current machine. The user can either poll for connections, or instruct the port to launch a new thread each time a new socket connects; in this case, the user provides a callback function to be invoked in the thread with the newly created socket.

A.2.2 Channel Abstraction (*rtchannel.[hpp, cpp]*)

The **RtChannel** is an extra abstraction layer that can be wrapped around a socket. The channel provides the concept of allocatable *bands* on a socket, thus partitioning the socket into many independent two-way bands of communication that can be monitored as a group. These bands each operate via asynchronous callbacks like the socket itself, but they each maintain their own buffer, and a central dispatch thread automatically reads incoming socket data and routes it to the specified band buffer. The channel also specifies two different forms of message: a *command* message consisting of a single integer, or a *data* message, which is a standard variable-length packet.

The channel also provides several utilities, including the ability to register callbacks for data flow (to provide for benchmarking and logging) and functions to estimate latency on the socket.

Finally, the channel abstraction removes any size restriction on packets; it has the ability to split up and recombine large packets transparently, removing any requirement for higher level

operations to worry about maximum packet sizes.

A.2.3 Timer Facilities (*rttimer.[hpp,cpp]*)

This package provides OS-independent timing and interval callback facilities. Class **RtTimer** serves both these functions; as a class, each instance provides a separate timer that can be paused, restarted, and have a different time velocity (e.g. ratio of timer time to real time, which can be negative to cause time to flow backward). These objects inherit from the packable base class and implement the packing interface, so they can be transmitted to other entities on the network to provide timer synchronization. The class also contains several static functions which can return the current system time, and provide a global interval callback mechanism (e.g. the user can register callback functions to be invoked at a specified interval).

A.2.4 Threading Facilities (*rtthread.[hpp,cpp]*)

The **RtThread** package provides OS-independent threading and synchronization primitives. The three component classes are **RtThread**, which acts as a both thread launcher and the controller object of the launched thread; *RtThreadSemaphore*, which provides locking across threads; and *RtThreadEvent*, which provides signallable event queues. There are currently three separate implementations of this package, one for SPROC threads (e.g. Irix basic threads), one for PTHREAD threads (used in Linux and later versions of IRIX for better performance), and one for Win32 threads.

A.2.5 Universal Base Class (*wkobject.[hpp,cpp]*)

WkObject is a base class from which all packable and transmissible objects must be inherited. This is because **WkObject** provides an extended form of RTTI that is needed to provide different processes, on different machines, compiled with different compilers, with the ability to identify and reconstruct classes that are specified by other processes. To implement this, the user must identify a unique integer ID with each class that is ever integrated into the system; we maintain a central list of the already-used IDs, and have typically in the past assigned ranges of IDs to students and users who are adding classes to the system to prevent collisions. Once these IDs are assigned, a class must include the SUPPORT_RTTI and RTTI_DEFINE macros in its implementation (the header and source files, respectively) which bind the compiled code for the class to the specified integer ID.

Any **WkObject** can be asked what its class ID is, via a virtual function call. This ID, in turn, may be passed to the global **WkClassRegistry** object to provide the following functions:

1. Ask whether one class ID inherits from another class ID.
2. Construct a new instance of the specified class on the heap.

A.2.6 Packable Interface and Smart Buffers (*rtbuffer.[hpp,cpp]*)

RtBuffer is an extensive and flexible buffer class that supports:

1. Heap allocated or static buffers;
2. Rotating/Circular buffers;
3. Buffer references (e.g. "pointers" into buffers);
4. Buffer locking and producer/consumer queueing;
5. Byte-order-independent writing and reading;
6. Loading or storing buffers to and from files

This header also defines the packable interface, **RtPackableIfc**, which should be inherited from to implement packable objects (e.g. those that can be transmitted across the network or to persistently stored in databases). **RtPackableIfc** inherits from **WkObject**, so you need only inherit from the former. The function signatures in the packable interface read from and write to buffer references (**RtBuffer::Ref**), which is why these classes are together in this file.

A.3 Database Library (ndf)

The database library is called the **ndf** library, and can be found in *src/ndf*. This library implements the classes necessary for manipulating databases, database objects, and schemas in server and client mode. This library depends on **system** and **gsim**.

A.3.1 Block Services (*dfblockservice.[hpp,cpp], dfblockserviceex.[hpp,cpp]*)

The lowest level services are the block service classes, which abstract a linearly addressed, flat file space containing a number of blocks of a specified size. Blocks can be added, removed, read, and written. The block service reads and writes to a file; it is not network enabled. The API interface for the base class can be found in class **DfBlockService** in *dfblockservice.[hpp,cpp]*.

The virtual function API provides for *transaction* functionality at the block service level. This means that a transaction may be begun, operations performed on the file, and the transaction can be committed atomically. If the commit fails or the program crashes, the next time a block service is opened on that file, the file is transparently repaired to conform to the state before the transaction began. If the commit succeeds, the physical state of the file is brought in line with the operations performed. The base class, **DfBlockService**, does not implement this functionality (e.g. the virtual functions return **fail** for all transaction calls); however, there is an additional class, **DfBlockServiceEx**, defined in *dfblockserviceex.[hpp,cpp]*, which does implent the transaction operations. It does this by opening, in turn, a normal block service, but taking over and hiding the "zero" block in the file, where it stores transaction information. When a transaction is begun, **DfBlockServiceEx** buffers all operations in memory until an abort or commit is performed. If an abort is performed, the buffers are simply flushed. If a commit is performed, a sequence of operations (including writing a recovery record to the end of the file (past the last "valid" block), making modifications, then tagging and deleting the recovery record) is performed such that at no time is the physical disk image inconsistent. If the commit completes, the physical image is the same as the memory image and the memory image is flushed.

A particular file can only ever be opened as one of the two types, **DfBlockService** or **DfBlockServiceEx**. Opening the same file later as the other type will at best cause errors and at worst corrupt the database. At this time, all files are being created and opened as the extended form, **DfBlockServiceEx**.

The blocks themselves are referenced with a block reference class, **DfBlockId**, which is defined in this header. Block IDs are 64-bit values implemented as a pair of integers; they may be incremented, decremented, compared, etc. The block services take and return these IDs rather than integers; this abstracts away a 64-bit file space, getting away from the 2 GB file size limit imposed by many operating systems, a consequence of taking 32-bit integers to the various file I/O functions. **DfBlockId**s are packable and can be converted to and from byte-order-independent values for transmission or storage.

A.3.2 Blob Services (*dfblobservice.[hpp,cpp]*)

The blob service sits atop a block service, and abstracts efficient storage and retrieval of Binary Large OBjectS (e.g. blobs). The blob service provides server/client functionality, where a "universal blob server" can be attached to a block service on a local disk, then an adapter blob service can connect to the universal server; clients on the adapter machine can then access data in the remote machine via the exact same API as if the data were on the local machine. The blob server, class name **DfBlobService**, provides the following operations:

1. Open with either a local block service, or to a server port on which a universal blob server is running;
2. Allocate a new blob ID (e.g. a **DfRef**, also called a Ref);
3. Store a binary blob referenced by a particular Ref;
4. Retrieve a binary blob referenced by a particular Ref;
5. Lock or unlock a blob referenced by a particular Ref;
6. Watch or stop watching a blob referenced by a particular Ref.

Locking is done by Ref, and is a control mechanism for multiple clients attaching to the same blob service. A lock specifies an ID, which is used to identify the same "client" for future lock requests. The semantics of this ID are up to the client itself; the blob service does not interpret it. Watches allow the client to specify a callback function to be invoked when the Ref in question is manipulated in a particular way via the API. When in local mode, the lock and watch tables are simply kept in the blob service object. In server-client mode, the adapter service passes the requests via a channel control band to the universal server, which executes the command locally and returns the result via the same band. Watches and locks are aggregated at the adapters and passed along to the server as single requests, using the adapter's local pointer as its ID, then dispatched to local processes via local routing tables when the server responds. Watch responses are executed in a special watch thread, started by the server when it runs.

The blob service also supports nested transaction functionality. A stack of *transaction buffers* (**DfTxnBuffer**) are maintained; whenever a transaction begin is issued, a new buffer is pushed onto the stack. Writes to the database are routed to the top buffer, where they are stored pending a commit. Reads are also routed to the top buffer; if the buffer contains a cached write,

that value is returned, else it recurses to the next element in the stack, eventually reaching a read from the database itself. A commit writes the cached blobs in the topmost buffer to the next buffer down, and pops the buffer; if the buffer is the last one in the stack, the commit is performed on the database itself. An abort simply pops the top buffer, deleting those writes. Note that watches are triggered only on that last commit to the database itself. The lowest level buffer uses the transaction functionality provided by the block service to make atomic changes to the database file.

A.3.3 Universal Blob Server (*dfservermain.[hpp,cpp]*)

The file *dfservermain.cpp* provides the main routine for the universal blob server. To create the server, simply compile and link the **ndf** library with *dfservermain.cpp*; this will create the server executable. The server takes two arguments; the local file on which the block service will be opened, and the port number on which the server is to listen.

A.3.4 Transaction Stack Object (*dftransaction.[hpp,cpp]*)

The transaction stack objects, **DfReadTransaction** and **DfTransaction**, are utility objects intended to automate common tasks. It is very common, in the scope of a function, to read some database objects, acquiring read or write locks, operate on them, and then release them before exiting. However, it is easy to lose track of locks or object references in such functions due to a return from the middle of the function, or an exception being thrown from a lower scope. Explicitly unlocking and dereferencing all the temporary objects every time you call return or throw an exception is hard to remember and makes the code hard to read.

DfReadTransaction and **DfTransaction** help by being declared locally, in the function scope, on the stack. Instead of requesting an object directly from the database, the user requests it via the transaction (e.g. instead of database-¿Get(object), the user calls transaction.Get(database, object)). The **DfReadTransaction** automatically applies a read lock and keeps track of the reference for the user; since it is a stack object, any return or exception thrown from the scope will automatically cause the transaction destructor to be called, which causes the object to be unlocked and dereferenced properly. **DfTransaction** inherits from **DfReadTransaction**, so it provides read tracking as well. However, it also provides write and transaction tracking; simply declaring a **DfTransaction** automatically begins a transaction on the database, and if the user does not explicitly commit the transaction in scope, the object will abort the transaction upon destruction. This guarantees proper stacking and closure of transactions within scopes, again preventing subtle bugs from

sneaking into the code.

A.3.5 Persistent Object Base Class (*dfobject.[hpp,cpp]*)

All persistent objects are inherited from the **DfObject** base class, defined in *dfobject.hpp*. This class inherits from the **RtPackableIfc** base class, inheriting schema identification, a class factory, and serialization interfaces. The **DfObject** adds the concept of the object "belonging" to a database with a **DfRef** object identification. This is not to say that a **DfObject** *must* be assigned to a database; the user can create an object of this type that does not belong to a database, in which case its database pointer and ref are both NULL. Such an object still responds to the reference counting interface (e.g. the user can still Remember() or Forget() an unassigned **DfObject**; however, if the object is unassigned, this process becomes simple reference counting, with the final Forget() resulting in a call to C++ delete).

An unassigned object may be assigned to a **DfDatabase** via the AssignToDatabase(db) call, at which point it acquires a single reference count (belonging to the assigning function), a newly allocated non-NULL **DfRef**, and a database pointer. At this point, the object may be Store()d into the database and retrieved by other processes via its ref. Typically, some other object in the database will be modified to have the new Ref added to it in some way; other processes learn of the existence of the new object by reading the other object, thus finding the new Ref, and then issuing a load for that Ref to get the new object.

A.3.6 Database Shell Object (*dfdatabase.[hpp,cpp]*)

The **DfDatabase** object provides the final layer of abstraction atop the database stack (**DfDatabase** accesses a **DfBlobService** which accesses a **DfBlockService**). This layer presents C++ objects to the code above, and translates those objects to and from BLOBs to the blob service.

The user requests **DfObject**s from the **DfDatabase** by their **DfRef**s. The **DfDatabase** checks for the existence of the object in the current memory space; if it is already there, it is simply referenced and returned. If it is not there, the **DfDatabase** passes the Ref along to the blob service, which returns the binary block stored for that Ref (from disk or network, depending on what operating mode the blob service is in). The **DfDatabase** reads the object's class ID from the beginning of the block, uses the class factory to construct a C++ object of that type, and applies the object's unpacking routine to the remainder of the buffer to reconstitute the object. The object is then added to the lookup table for the local memory space, referenced, and returned.

Storing an object proceeds in reverse. The Store() call on an object causes the object to write its class ID and call PackTo() function on a buffer, which is then passed to the **DfDatabase**, which stores the buffer in the blob service under the specified Ref.

Locking and watching are passed largely unchanged through to the blob serivice; however, the blob service requires a local ID for locks and watches. The **DfDatabase** object simplifies the process by acting as the ID for the caller; it passes its own memory pointer along as the ID for the lock or watch, preventing the user from having to worry about ID allocation.

A.3.7 Smart Pointers (*dfsmartptr.[hpp,cpp]*)

DfSmartPtr is a template class that automates tracking and loading of database objects. A very common operation in database objects is to have a reference to another database object which can be loaded or unloaded dynamically as the needs of the program dictates. The "raw" way of doing this is to keep a **DfRef** and a pointer to the object, where at any given time the pointer can be NULL even though the Ref is not. This requires all pieces of code which use the pointer value to check if it is loaded and load it into the pointer if not. This, in turn, requires mutexing if the object can be accessed in multiple threads. All of this bookkeeping can make the code unnecessarily complex, is prone to errors, and can require modifying legacy code that operated on the pointers.

The **DfSmartPtr** class alleviates many of these problems. The class encapsulates a pointer value and a Ref value; the pointer is to the type templated into the class instance. The smart pointer can be assigned from either a Ref directly, or from an object (in which case the pointer loads the ref and database from the object, if it is assigned). The smart pointer can be used by operators in the same way a direct pointer would be used; if the object is not loaded, the smart pointer will transparently load the object before executing the operation. All normal pointer operations are overloaded on the smart pointer to perform this service. Furthermore, loading and unloading the pointer are single function calls (to Load() and Unload()), which perform all necessary tracking, untracking, and pointer tests. Legacy code can usually operate properly on a smart pointer to type T with no changes, where it was expecting a direct pointer to T. This class greatly improves readability and portability of code.

A.4 Simulator Library (newsim)

A.4.1 General Architecture

The simulation subsystem relies on a central controller object, the **RtSimulationManager**, of which there is one per machine. The purpose of the manager is to establish and maintain *simulation services* and communication between those services and their clients.

When a process is started, a simulation manager is created for that process. This is done only once; the first call to **RtSimulationManager::GetGlobalManager** creates the global service, and that service is the same for that process from then on. After the global service is created, a number of types of service may be registered with the manager within the process. This registration is done via the **RtSimulationDescription** object; for each type of simulator in the process, the process calls **RtSimulationManager::RegisterService** with a service descriptor for the service.

The service descriptor is basically a holder object that contains universal class IDs describing which class is used for a given position in the pipeline for that service. The main types and elements that *must* be defined in a service description are:

1. A local simulator class (derived from **RtLocalSimulationService**);
2. A class of conditions generated by that simulator (derived from **RtConditions**); and
3. A description string that can be displayed in a dialog box, describing what service this represents (in English text).

Optional elements that may be overloaded in a description, but do not generally have to be overloaded, consist of:

1. A local client class (derived from **RtLocalSimulationClient**);
2. A new client telemetry type (derived from **RtPackableClientState**);
3. Remote client and service classes (derived from **RtRemoteSimulationClient** and **RtRemoteSimulationService** respectively); the remote client defines the just-in-time priority function for bandwidth usage, and the remote service acts as a local surrogate for the actual service on the remote machine;
4. A new intermediate condition manager type (derived from **RtSimServiceConditionManager**), which defines the policy of how new conditions are integrated into the condition set;

5. Capability flags that describe what general capabilities the service has.

These optional classes are ones that, for most simulators, work fine with the default, base classes, and do not commonly need overrides.

Registering a simulator does not actually create any running instances; however, simulation managers automatically notify "neighbor" managers of what services are available on the local machine, so registering a service on a machine will cause that service descriptor to show up as a "remote service" on all other machines in the network.

The client process may request a list of available descriptors from its local simulation manager; this will return two sets of services, one set of *available* services (e.g. those for which new instances can be launched) and a set of *running* services (e.g. specific running examples of a simulation). The latter descriptors are actually derived from a subclass of **RtServiceDescription** called **RtRunningServiceDescription**. There are extra, private fields in both types that allow the local manager to route requests to other managers or specific instances of simulations on other managers, in the case of the running services.

Once the process has a descriptor, it can request either a new instance of the service be launched on the provider machine (in the case of a simple descriptor), or a new connection to an existing (running) service (in the case of a running descriptor). Typically the former is followed immediately by the latter; note that immediately after launching a service, a new **RtRunningServiceDescriptor** will be available representing the new instance.

Successfully connecting to a service will result in the return of a new object derived from **RtLocalSimulationClient**. This client object is the interface through which all communication with the service is performed. Figure A.1 shows the configurations of objects and machines that will be created if the request is made for a service registered on the local machine; figure A.2 shows the configuration resulting from a request for a service on a remote machine. Note that, to the requesting process, the result is the same; a pointer to a local **RtSimulationClient**, and to the specific launched simulator, again, the result is the same, a pointer to an **RtSimulationClient**. In the remote case, however, there may be one or more jumps through intermediate, virtual clients and servers, that behave as a local proxy of the remote object.

During simulation, the actual service (derived from **RtLocalSimulationService** feeds conditions of the type specified in the service description into its local **RtConditonSet**. All local clients of the simulation can simply read directly out of this set for maximal performance.

Some of these clients, however, may be remote clients, representing interested processes

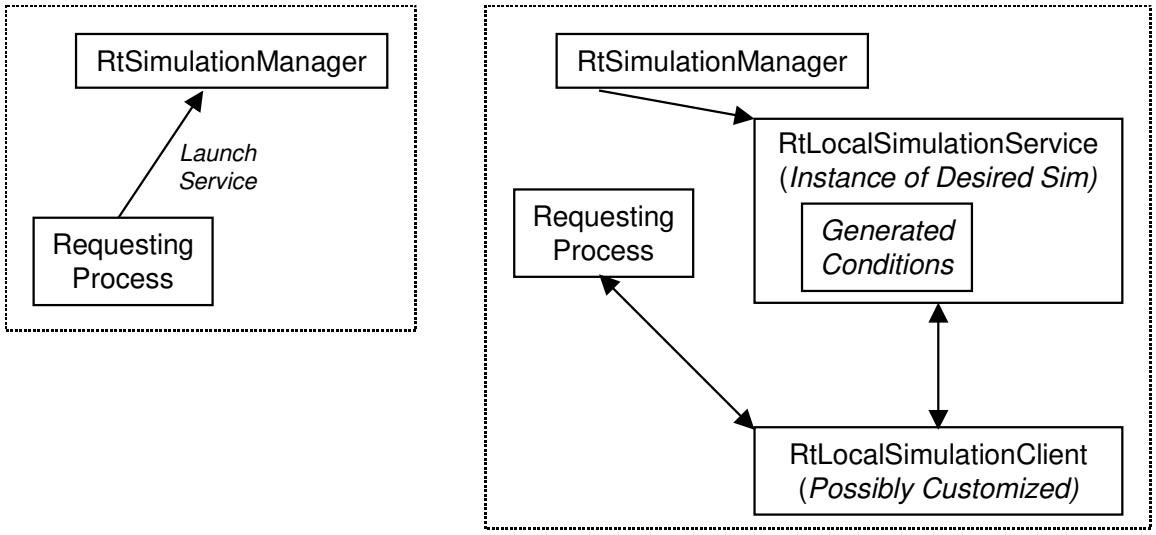


Figure A.1: *The object configuration resulting from a request to create and connect to a service on the local machine.*

on other machines. These remote clients act as surrogates for the virtual service on the client machine. A **RtRemoteSimulationClient** implements the just-in-time priority function that decides which conditions are the most important to transmit at a given time. Every so often (typically one or two tenths of a second), the simulation manager performs a callback on all **RtRemoteSimulationClient** objects for currently running simulations, instructing those clients to gather a certain amount of data (specified by the **RtBandwidthManager** for the channel, which provides a certain amount of bandwidth available for the timeslice; this bandwidth is evenly split among all active remote simulation clients). The job of the condition manager is to select the most important conditions from the service it is connected to, up to an amount of data equal to the amount specified by the bandwidth manager, and transmit those conditions to the remote simulation service on the other end of the channel. In this way, conditions make their way in a just-in-time fashion from the actual service to the "real" client on the remote machine, potentially several hops away.

In order to decide which elements are most important, the **RtRemoteSimulationClient** must have access to client telemetry (defined in chapter 4). This telemetry is an object subclassed from **RtPackableClientState**; one such object is maintained at each local client, and these objects are propagated from local clients to remote server to remote client to local server in the reverse direction from the conditions, but via the same mechanism (e.g. the simulation manager also allocates bandwidth to the **RtRemoteSimulationClient** objects it is managing to allow them to propagate changed telemetry to the services they are attached to). Given the telemetry state at the remote ser-

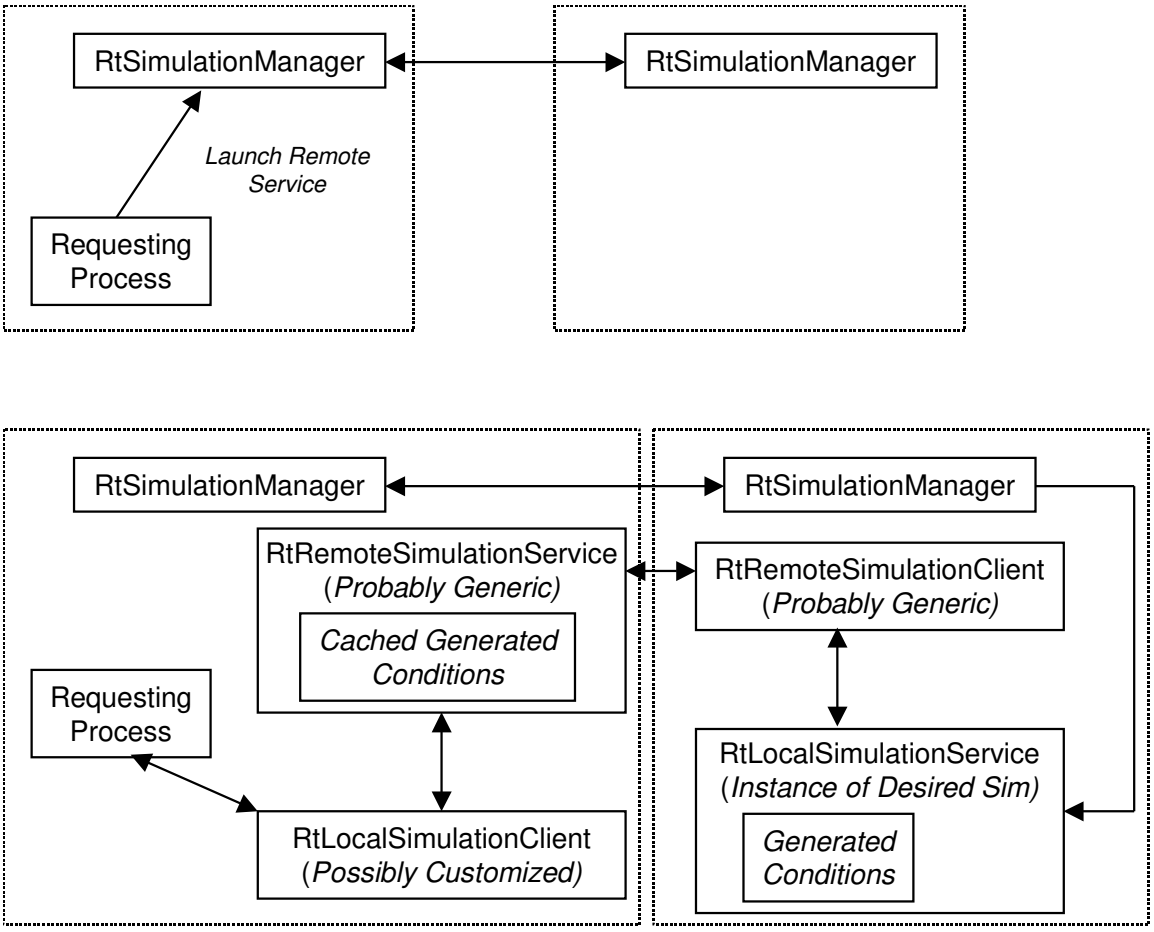


Figure A.2: *The object configuration resulting from a request to create and connect to a service on a remote machine.*

vice node in which it lives, the simulation service condition manager can decide what data is most important for transmission.

All of the optional components have overloads already defined for the two common cases of the generic virtual time simulation and the generic real time simulation. In practice, the defaults for all classes except the **RtLocalSimulationService** (e.g. the simulation itself) and the condition set type (e.g. the simulation data) can be used directly by deriving the service from either **RtLocalRtSimulationService** (for real-time simulations) or **RtLocalVtSimulationService** (for virtual-time simulations), and inheriting the simulation descriptor from **RtRtServiceDescription** or **RtVtServiceDescription** (for real-time or virtual-time services respectively). This will automatically select appropriate overloads for all the other classes which implement the telemtry types (e.g. visible and lookahead sets) and packing algorithms (e.g. closest visible conditions first) described in the body of this thesis.

Figure A.3 shows the inheritance patterns of the classes implemented in the simulation base library. Users should typically be either overloading or directly using leaf classes in this hierarchy.

A.4.2 Summary of Optional Overloadable Classes

Overload	.. In ..	.. To ...
RtLocalSimulationClient	**RtServiceDescription**	Provide simulator-specific telemetry changing calls, or simulator-specific communication utility functions
RtPackableClientState	**RtRemoteSimulationClient** subclass	Provide additional telemetry data specific to the simulation
RtRemoteSimulationClient	**RtServiceDescription**	Allow you to define a new just-in-time packing function, and specify the packable client state class.
RtRemoteSimulationService	**RtServiceDescription**	Allow you to overload the simulation service condition manager
RtSimServiceConditionManager	**RtRemoteSimulationService** subclass	Allow you to define a new override policy for submissions of new conditions.

A.4.3 Simulation Manager (*rtsimulationmanager.[hpp,cpp]*)

This file defines the simulation manager and its auxiliary classes. There should be one simulation manager per process; processes can retrieve this manager by including the header and calling **RtSimulationManager::GetGlobalManager**. If the manager is to be able to receive con-

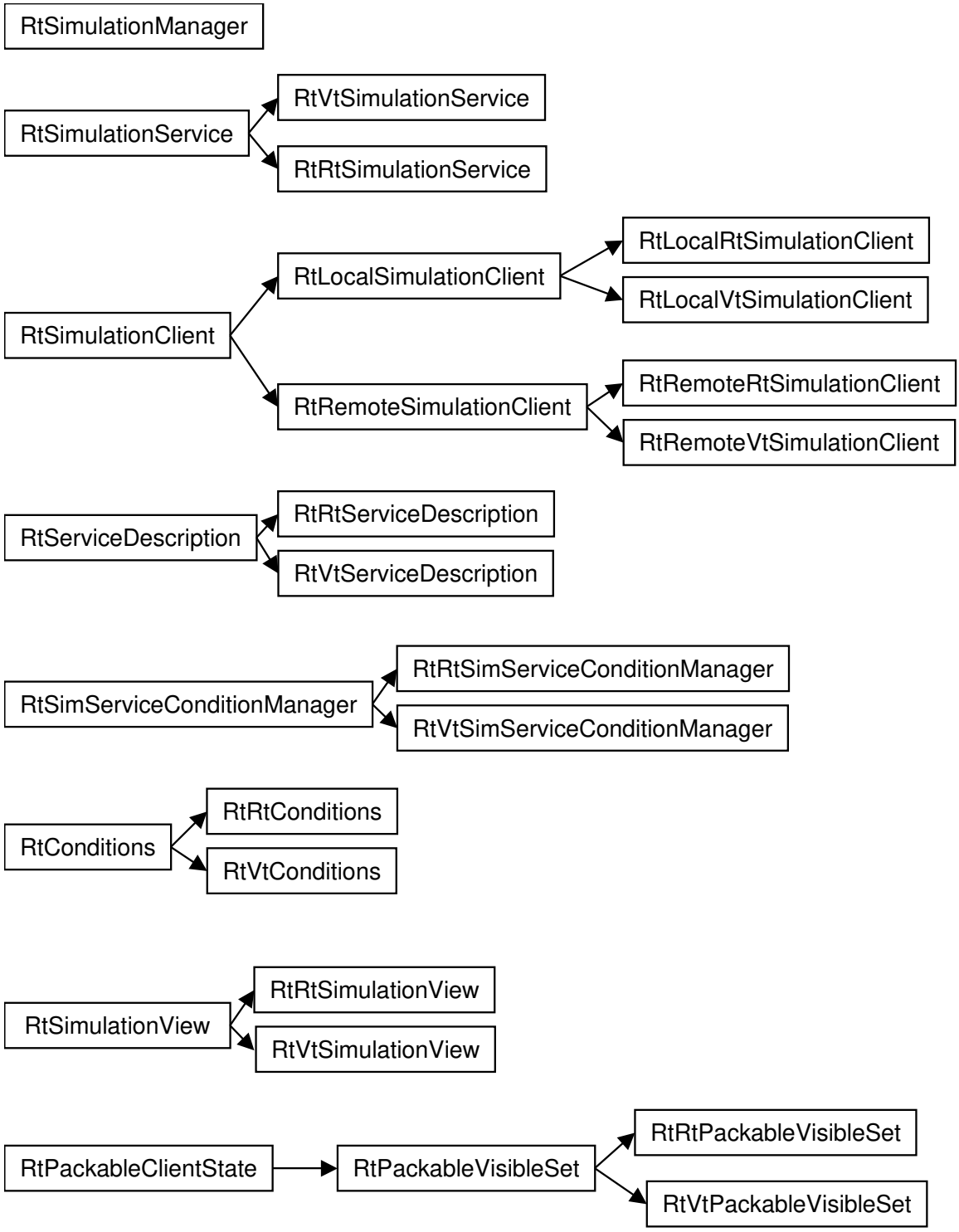


Figure A.3: *Classes and inheritance patterns in the* **newsim** *library.*

nections from other managers, the process must call **RtSimulationManager::StartServerThread** and give it a port number for the manager to use on the local machine to listen for connections from other managers. The process then registers any local simulators with the manager, and connects the manager to other managers (if a network is desired) by calling **RtSimulationManager::-ConnectToOtherManager**.

Once simulations are registered and connections established, the process can get descriptions of local and remote, running or provided simulators via the two calls **GetProvidedServices** and **GetRunningServices**. Each returns a set of services that are available. A provided service may be launched by passing its description back to **LaunchService**, and a running service may be connected to by passing the running service description to **ConnectToService**. Each of these functions returns a subclass of a **RtLocalSimulationClient**, which the caller is then responsible for "hanging up" when they are done with it.

There is a callback event interface, provided by the **RegisterForEvents** call, which allows the caller to be notified by the manager when certain events happen:

1. A new service is registered somewhere on the network.
2. A service is taken off line.
3. A new simulation has been run somewhere on the network.
4. A new client has been created on the local machine.
5. A connection to another simulation manager has been severed.
6. A simulation is shutting down.
7. A client is being closed on the local machine.

A.4.4 Bandwidth Manager (*rtbwmanager.[hpp,cpp]*)

The bandwidth manager is a utility class used by **RtSimulationManager** to monitor and control traffic on its **RtChannel**. The bandwidth manager is allotted a certain amount of bandwidth per second, and is given a callback frequency. The bandwidth manager then notifies the simulation manager when bandwidth is available, and how much; the simulation manager can then partition this bandwidth among its client objects and allow them to communicate in a controlled fashion.

A.4.5 Condition Chunk Base Class (*rtconditions.[hpp,cpp]*)

The **RtConditions** class is the base class for all simulated output. The base class defines virtual comparison functions between condition objects, and keys conditions by a spatial key and subkey. This file also defines the condition *set*, **RtConditionSet**, which is a set of conditions that defines various utility functions, the ability to lock and unlock the set, and an optional efficient secondary key lookup function (needed by some operations; e.g. some operations need efficient lookup by space, whereas other operations need efficient lookup by time. In this case, the secondary key function would be the time-based lookup).

A.4.6 Simulation Service Base Class (*rtsimulationservice.[hpp,cpp]*)

The simulation service base class provides derived classes with the ability to track and talk to the simulation manager, and provides storage space for a condition set and a simulation service condition manager. It also provides calls to communicate with clients (in either broadcast or single-client mode), and callback functions for incoming messages from clients. These functions are common to all subclasses of service, from actual services to remote stubs.

Typical users will need only interact with the callback functions to receive messages and state change requests from clients, and the interface to submit new or changed conditions for the running simulation.

A.4.7 Simulation Client Base Class (*rtsimulationclient.[hpp,cpp]*)

Similar to the service base class, this class provides space for a telemetry object, utility functions to communicate with the simulation manager and the attached simulation service, and provides callbacks for incoming messages from the service. It handles tracking the telemetry object's dirty state and whether or not it needs to be updated.

Typical users will only need to interact with the interface for receiving direct communications from the server, the interface for sending direct communications to the server, and the call to update the telemetry object.

A.4.8 Local Simulation Service Base Class (*rtlocalsimservice.[hpp,cpp]*)

This is an intermediate class and should not be directly used; the user must derive their local service class from either **RtLocalVtSimulationService** or **RtLocalRtSimulationService**,

which are both derived from this class. This class exists primarily to group those two subclasses together under one class ID.

A.4.9 Remote Simulation Service Base Class (*rtremotesimservice.[hpp,cpp]*)

This class is the base class for **RtRemoteRtSimulationService** and **RtRemoteVtSimulationService**, which in turn are used as proxies for a service that is connected to, but actually resides on another machine. This class should not be overloaded directly.

A.4.10 Real-Time Simulation Components (*rtrtcomponents.[hpp,cpp]*)

This is one of two files that contain the "grab bag" of overloads of the optional classes that implement the real-time simulation functions. It includes the following:

Class	Overloads	Purpose
RtRtConditions	**RtConditions**	Real-time conditions; adds time stamp, and sets the sort order to {Key, Time, Subkey}
RtRtServiceDescription	**RtServiceDescription**	Selects **RtLocalRtSimulationClient** as the local client type, **RtRemoteRtSimulationClient** as the remote client type, and **RtRemoteRtSimulationService** as the remote service type.
RtRtSimServiceConditionManager	**RtSimServiceConditionManager**	Specifies the real-time replacement function for new conditions; e.g. if new conditions are submitted for a volume, they override the existing conditions for that volume.
RtRemoteRtSimulationService	**RtRemoteSimulationService**	Forces use of the **RtRtSimServiceConditionManager** as the condition manager.
RtLocalRtSimulationService	**RtLocalSimulationService**	Base class for actual real-time services.

A.4.11 Real-Time Simulation Client Base Class (*rtrtsimulationclient.[hpp,cpp]*)

This is the other package of overloads for real-time simulation. It includes the following:

Class	Overloads	Purpose
RtRtPackableVisibleSet	**RtPackableClientState**	Real-time simulator telemetry class; provides a visible set of spatial keys, and a lookahead set of spatial keys.
RtLocalRtSimulationClient	**RtLocalSimulationClient**	Declares **RtRtPackableVisibleSet** as the client telemetry type, and provides utility functions for accessing and setting the telemetry set.
RtRemoteRtSimulationClient	**RtRemoteSimulationClient**	Implements the real-time importance algorithm; e.g. transmit elements that are visible, have not been transmitted yet, and have not yet been overloaded by more recent conditions. Also, declares **RtRtPackableVisibleSet** as the client telemetry type.

A.4.12 Virtual-Time Simulation Components (*rtvtcomponents.[hpp,cpp]*)

This is one of two files that contain the "grab bag" of overloads of the optional classes that implement the virtual-time simulation functions. It includes the following:

Class	Overloads	Purpose
RtVtConditions	**RtConditions**	Virtual-time conditions; adds time stamp, and sets the sort order to {Time, Key, Subkey}
RtVtServiceDescription	**RtServiceDescription**	Selects **RtLocalVtSimulationClient** as the local client type, **RtRemoteVtSimulationClient** as the remote client type, and **RtRemoteVtSimulationService** as the remote service type.
RtVtSimServiceConditionManager	**RtSimServiceConditionManager**	Implements the replacement policy for virtual time conditions.
RtRemoteVtSimulationService	**RtRemoteSimulationService**	Forces use of the **RtVtSimServiceConditionManager** as the condition manager.
RtLocalVtSimulationService	**RtLocalSimulationService**	Base class for virtual time simulators.

A.4.13 Virtual-Time Simulation Client Base Class (*rtvtsimulationclient.[hpp,cpp]*)

This is the other package of overloads for virtual-time simulation. It includes the following:

Class	Overloads	Purpose
RtVtPackableSpaceTimeState	**RtPackableClientState**	Virtual-time simulator telemetry class; provides a visible set of spatial keys, a lookahead set of spatial keys, and a current time and time velocity for the viewer.
RtLocalVtSimulationClient	**RtLocalSimulationClient**	Declares **RtVtPackableSpaceTimeState** as the client telemetry type, and provides utility functions for accessing and setting the telemetry object.
RtRemoteVtSimulationClient	**RtRemoteSimulationClient**	Implements the virtual-time importance function; e.g. transmit elements that are visible, haven't been transmitted yet, and are closest to the user's current desired visualization time; also, declares **RtVtPackableSpaceTimeState** as the client telemetry type.

A.4.14 Simulation View Base Classes (*[rtsimview,rtbasicsimview].[hpp,cpp]*)

The view classes are basically utility classes that can be attached to an **RtSimulationClient** and a graphics engine, and provide two major functions. First, they provide a standard UI window, in some cases with standard widgets (e.g. the VCR controller in the virtual time view class), to help control the simulation. Second, they automate the updating of the client telemetry object by hooking into frame callbacks and setting the new visible and lookahead sets if the user's viewpoint has changed.

There is a basic view class that simply provides a name for the subclasses, **RtSimulationView**, and the ability to hook into a generic UI window in the Citywalk system. **RtBasicSimView** adds the ability to automatically hook into the frame callback for Citywalk and propagate visible

set information automatically in the telemetry set; **RtVtSimulationView** adds the VCR widget and adds time and time velocity tracking to the telemetry. All view classes also track the client itself for the user; e.g. when the view object is destroyed, it automatically dereferences and frees the simulation client.