Avocado: A Distributed Virtual Environment Framework

Henrik Tramberend

März 2003

Dipl-Inform. Henrik Tramberend
henrik@tramberend.de

Technische Fakultät
Universität Bielefeld

Abdruck der genehmigten Dissertation zur Erlangung des akademischen Grades Doktor der Naturwissenschaften

Gutachter:

Prof. Dr. Ipke Wachsmuth
Dr. Martin Göbel

Prüfungsausschuss:

Prof. Dr. Gerhard Sagerer
Prof. Dr. Ipke Wachsmuth
Dr. Martin Göbel
Dr. Marc Latoschik

Tag der Promotion:

6. März 2003

Referenzexemplare gedruckt auf alterungsbeständigem Papier nach ISO 9706

Avocado: A Distributed Virtual Environment Framework

Dissertation zur Erlangung des Grades
Doktor der Naturwissenschaften

von

Henrik Tramberend

Technische Fakultät der
Universität Bielefeld

März 2003

Danksagung

Marc Latoschik
Ipke Wachsmuth
Martin Göbel
Olaf Langmack
Christoph Stratmann
Frank Hasenbrink

Danke.

Contents

1. Introduction 1
1.1 Motivation 1
1.1.1 Virtual environments 2
1.1.2 Distributed virtual environments 4
1.1.3 Large-scale distributed virtual environments 6
1.2 Overview of this work 7

2. Related work 9
2.1 Non-distributed VE systems and standards 9
2.1.1 OpenGL Performer 10
2.1.2 OpenInventor 13
2.1.3 VRML97 16
2.1.4 X3D 17
2.1.5 VR Juggler 18
2.2 Distributed VE systems 19
2.2.1 RB2 19
2.2.2 MR 19
2.2.3 DIVE 20
2.2.4 Repo-3D 20
2.2.5 MASSIVE 20
2.3 Research prototypes for scalable DVE systems 21
2.3.1 SPLINE 21
2.3.2 CVE (MASSIVE-2) 22
2.3.3 HIVEK (MASSIVE-3) 23
2.3.4 RING 23
2.3.5 NPSNET 25
2.4 Clustered rendering systems 26
2.4.1 WireGL 28
2.4.2 Net Juggler 28
2.5 Summary and discussion of related work 29

3. DVE Systems - A conceptual approach 35
3.1 DVE system requirements 35
3.1.1 Distributed object and event model 35
3.1.2 Display device abstraction 36

3.1.3 Direct-manipulation user interaction 37
3.1.4 Rapid prototyping and extensibility 37
3.1.5 Platform and performance considerations 39
3.2 From concept to architecture 39
3.2.1 Object model 40
3.2.2 Event model 42
3.2.3 Distributed object model 44
3.2.4 Distributed application layout 46
3.2.5 Distributed event model 48
3.2.6 Network transport layer 48
3.2.7 Data input and device interface 50
3.2.8 Scripting language selection and binding 51
3.2.9 Target platform considerations 54
3.2.10 Execution model: Single vs. multi-threading 54
3.2.11 A base system for low-level tasks 55
3.2.12 System API structure 56
3.2.13 Extension mechanism: API and link strategy 57
3.3 AVOCADO Architecture Summary 58

4. Avocado - implementation of the framework foundation 61
4.1 The Avocado object model 61
4.1.1 Mapping the field concept to Performer 62
4.1.2 Field containers for object state encapsulation 65
4.1.3 Smart pointers and reference counting 66
4.1.4 Adaption of Performer classes through subclassing . . 68
4.1.5 Field connections for event dissemination 71
4.1.6 Scene graph node classes 73
4.1.7 Sensor objects 73
4.1.8 An external device daemon process 74
4.1.9 Integration of the ELK Scheme language 75
4.1.10 A component interface for framework extension 77
4.1.11 An interface for state object persistence 78
4.2 Display device abstraction 80
4.2.1 Basic elements of the abstraction 82
4.2.2 Configuration examples for selected devices 87
4.3 Tool-based interaction support 92
4.3.1 An interaction metaphor for the Responsive Workbench 93
4.3.2 Implementation of the interaction abstraction 96
4.3.3 Extension through specialization 101
4.4 Summary 102

5. Avocado - implementation of the distribution architecture . . . 105
5.1 Distributed object model . . . 105
5.1.1 State sharing through object replication . . . 105
5.1.2 The distribution group abstraction . . . 106
5.2 Distributed event model . . . 109
5.2.1 Distributed state change notification and event handling 109
5.2.2 Field connections in the distributed context . . . 110
5.3 Guaranteeing application state consistency . . . 112
5.3.1 The Ensemble/Maestro group communication system 112
5.3.2 Consistency through total message ordering . . . 113
5.3.3 Synchrony through view atomic message delivery . . . 113
5.3.4 Dynamic membership and atomic state transfer . . . 114
5.3.5 Distributed locking through total message ordering . . 116
5.4 A simple distributed application example . . . 121
5.5 Pacman: A complex distributed application example . . . 123
5.6 Summary . . . 130

6. Scalability in distributed virtual environments . . . 133
6.1 Introduction . . . 133
6.2 Scalability analysis . . . 134
6.2.1 An environment model for scalability analysis . . . 135
6.2.2 Analysis of existing systems . . . 136
6.2.3 Comparison of tiling vs. no scalability mechanism . . 138
6.3 Exploiting visibility for rendering . . . 139
6.3.1 View frustum culling in visual rendering . . . 139
6.3.2 Level-of-detail evaluation . . . 141
6.3.3 Hierarchical level-of-detail rendering . . . 142
6.4 Exploiting visibility for distribution . . . 144
6.4.1 Hierarchical environment partitioning . . . 145
6.4.2 Mapping distribution groups to scene partitions . . . 146
6.4.3 HLOD evaluation and the working set . . . 148
6.4.4 HLOD evaluation and view frustum culling combined 149
6.4.5 Caching of inactive nodes . . . 150
6.5 Implementation of the HLOD node . . . 151
6.6 Scalability analysis of the hierarchical distribution approach . 152
6.7 Summary . . . 155

7. Results, applications and future work . . . 157
7.1 Results of this work . . . 157
7.2 Applications built with AVOCADO . . . 158
7.2.1 Multi-modal interaction in virtual reality . . . 158
7.2.2 Oil exploration demonstrator . . . 159
7.2.3 PC-CAVE render cluster . . . 162
7.2.4 Caveland . . . 163

7.2.5 Commercial applications: Vertigo Systems and RMH . 166
7.3 Suggestions for future work 167

1. Introduction

Building Distributed Virtual Environment (DVE) applications is a complex task. Many different technologies and algorithms from various areas of computer science need to be combined to build an application. Real-time 3D graphics, data acquisition and network communication are all active research areas that contribute basic technology to the development of DVE applications.

Developers and researchers typically spend a lot of time and effort to implement well known concepts from these areas. Specialized frameworks help application developers and researchers to focus on the development and implementation of application specific functionality by factoring aspects common to all DVE applications into a consistent and reusable foundation of concepts, APIs and services.

This thesis documents research performed during the development of the AVOCADO framework for distributed virtual environment applications at IMK VE[1], where it serves as the central platform for all DVE related research projects. In this context, possible approaches to the design of a general DVE framework have been investigated and the resulting reference implementation is presented. Special attention is given to the seamless integration of distributed application semantics into the standard, stand-alone VE object and event model. Further, the limiting constraints of large-scale DVE applications are discussed and a multi-resolution approach to scalability is presented.

Some of the concepts and results presented in this work have previously been published elsewhere (see [20, 71, 43, 29, 44, 72, 47, 74, 31, 66, 28, 73] in the bibliography).

1.1 Motivation

To motivate the development of a new DVE framework a general overview of the area of VE systems is given. To understand the complexity that is introduced into a VE system by adding distribution support, it is necessary to introduce all relevant approaches that have been taken in the past and

[1] Virtual Environments group of the Institute for Media Communication at the late GMD (German National Research Center for Information Technology). IMK VE is now part of the Fraunhofer-Gesellschaft.

to discuss their shortcomings. Finally, the importance of scalability in DVE applications is underlined.

1.1.1 Virtual environments

The terms *Virtual Environment* and *Virtual Reality* are often used synonymously to describe a computer-generated, artificial 'environment' or 'reality' that is presented to a user. A virtual environment tries to evoke a strong sense of reality in the user. This is achieved by the generation of artificial input to the users visual, acoustic and haptic senses.

By interfacing some of the users articulations in the real world back into the virtual environment, the user can consciously interact with the environment. Typically, interfaces to direct-manipulation devices are used, but nowadays more advanced interaction techniques like speech and gesture recognition have become a major research interest.

The generation of high-quality visual feedback from the virtual environment is often considered the most important aspect in generating a high degree of immersion. The desire to increase the degree of immersion led to the development of sophisticated image generators and display devices. Beginning with low-resolution monoscopic CRT displays used in early flight simulators and image generators that where capable of rendering only a few hundred polygons per second, the development progressed toward todays high-resolution stereoscopic display systems like the CAVE[16] and readily available graphic cards that render a hundred million polygons per second.

Depending on the kind of application, a virtual environment software needs to drive such diverse display devices as head-mounted displays (HMD), see-through HMDs for augmented reality, active stereo projection systems with LCD shutter glasses, passive stereo projection systems using polarization for image separation or various multi-screen projection systems. The picture in figure 1.1 shows the construction principle of a CAVE[2] multi-screen display.

Parallel to the development of new display devices, image generators and input devices, various toolkits and application frameworks are developed. They provide a basic software infrastructure for the development of VE applications. The main goal of these efforts is the maximization of software reuse in order to minimize the necessary development resources for application development. Designed for different application domains, the only common nominator of most toolkits and frameworks is a *scene-graph* based object model. The provided API, the supported hardware and operating systems and the set of supported display and input devices vary greatly.

[2] The CAVE is both a recursive acronym (Cave Automatic Virtual Environment) and a reference to "The Simile of the Cave" found in Plato's Republic, in which the philosopher explores the ideas of perception, reality, and illusion.

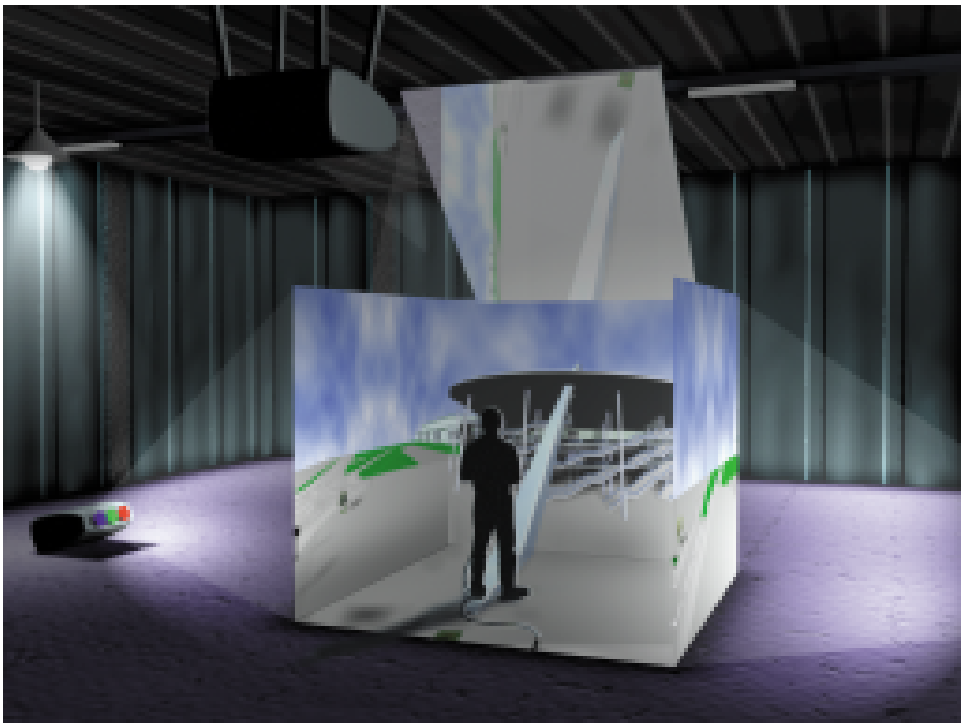

Fig. 1.1: A typical four-sided CAVE setup. The wall images are back-projected onto translucent screens, while the floor image is projected from above (picture courtesy of RMH[62]).

A closer evaluation of representative toolkits and frameworks (see section 2.1) reveals that, while each system has - depending on the application domain - promising and sometimes widely adopted solutions to specific sets of problems, none has all the necessary properties to serve as a basis for a general purpose DVE framework. A common trade-off is that between an elaborate object and event model and API and, on the other hand, rendering performance. Systems that emphasize developer friendly APIs tend to neglect support for performance optimizations, while systems optimized for performance often provide underdeveloped APIs and object models.

This thesis argues that the design of a framework for DVE application development must be based on well known and well understood design concepts from stand-alone VE toolkits and frameworks in order to be accepted by developers and to increase productivity. At the same time, maximum performance is a major requirement for the generation of interactive applications, and must not compromised by the design of the framework APIs.

Therefore, the most successful design concepts are culled from existing stand-alone VE frameworks, and are combined into the basic system design for the AVOCADO framework.

1.1.2 Distributed virtual environments

The possibility to immerse a single user into a virtual environment, creates the desire to simultaneously share the environment with multiple users, and allow them to not only interact with the environment but also with each other.

First attempts to provide such an environment where made by attaching two sets of displays and input devices to one stand-alone VE system. This effectively allows to generate two independent views into one environment. The drawback of this approach is twofold. The two persons using the system are forced to be located in close spacial proximity since the connections between a machine and the display and input devices are subject to limiting length restrictions. Therefore, the two people sharing a VE on such a system needed to be at least in the same building, if not in the same room.

The obvious solution to overcome this drawback is the use of two or more stand-alone systems that can communicate over a standard network connection in such a way that all connected systems have access to the data necessary to generate the distributed but consistent impression of *one* virtual environment that is shared between multiple users. The use of a standard network connection removes most restrictions on the relative spatial location of the users, while the use of one dedicated system per view eliminates the need to divide computing resources between views.

The generation of two views into an environment on one system is straight forward because there is one database that describes the environment according to a stand-alone object model, and it is the same for both views. Changes made to the database by either user are immediately visible to the other user because both view are generated from the same database. A centralized database is possible because the database access bandwidth available for the processes to render the views is only limited by the memory access bandwidth of the underlying hardware. In a networked setup the initial state of the environment as well as all subsequent changes to the environment need to be communicated to all connected systems to enable them to generate a consistent view of the environment for each participating user. A centralized approach is no longer possible, because network bandwidth is typically smaller than memory bandwidth by several orders of magnitude. A distributed object and event model is needed to describe the state of the networked distributed environment.

The first distributed object models in the context of DVE are extremely simple and cumbersome to work with because they place severe constraints on system configuration and setup possibilities and on the set of possible database operations in order to minimize the complexity of the networking code and to achieve a minimal level of consistency. The basic constraint is that no objects can be added to or removed from the environment and only specific object attributes can be changed at runtime. All processes

are required to load a copy of the environment database at startup and are only allowed to change specifically marked object attributes. These changes get communicated to the other processes where they are applied to the local copy of the database. Assuming that communication only starts after all processes have initially loaded the database, and the set of processes is static, there is a good chance that a consistent state can be maintained on the distributed database copies. The resulting system is static and the operations available to the application developer are very limited. On the other hand, this allows the extension of stand-alone VE systems to working distributed VE systems.

In contrast, dynamic systems should not impose any constraints on the set of operations possible on objects in a distributed environment compared to a stand-alone environment. Further, well defined consistency assertions must be given even in the presence of dynamically joining and leaving processes. Processes should not be required to bring their own copy of the environment database, but should be updated to the current state after joining the environment (A glossary of the DVE terminology used in this thesis is offered in table 1.1).

Network topology is another important area in DVE design. Basically two different approaches exist, centralized and distributed.

Centralized: A central server process maintains the environment database, while client processes connect to the server to obtain the information about the environment. Client processes do not directly communicate with each other.

Distributed: A distributed system does not rely on a central server, but consists of a set of identical peer processes that communicate with each other to exchange the necessary information about the environment. No central server is needed.

While a centralized system can more easily guarantee the provision of consistent environment views to the users, a distributed system is more reliable because there is no single point of failure. Many systems follow the centralized approach because it is less complex and is easier to implement. However, recent developments in the area of group communication provide foundations to build server-less distributed systems that are consistent and reliable.

These problem descriptions illustrate that in order to build a framework that unobtrusively and transparently supports the development of distributed VE applications, a number of issues need careful consideration:

- Design of a distributed object and event model (DOM)
- Developer APIs

Term	Meaning
User	A not necessarily human user whom a view of the environment is presented to. For example, autonomous agents that populate an environment also qualify as 'users'.
View	A view of the environment is generated under the control of one process on one machine.
Process	A process runs on one machine and handles the computational resources used to generate a view for one user. A process can be either client, server or peer.
Client	A client process can not live without a server. It communicates only with server processes.
Server	A server process feeds one or more. Sometimes, server processes communicate with each other.
Peer	A peer process does not need a server. All peers are created equal and communicate directly with other peer processes.
System	System, as in VE or DVE system, means the whole software setup, (like *operating system*). Can be meant to include the hardware as well.
Object	Objects populate the environment. The state of an object is completely described by its attributes.
Environment	The environment is entirely build from objects. A view of the environment is presented to the user.

Tab. 1.1: Each of these terms is used with any number of different meanings throughout all of computer science. However, in the context of this work they convey the stated meaning.

- Network topology
- Network transport
- DOM implementation

This thesis documents the development of a general design for a dynamic DVE framework that extends well known and established object models and APIs into the domain of distributed applications.

1.1.3 Large-scale distributed virtual environments

Scaling properties of DVE systems are best described along two dimensions: The number of users that participate in a DVE, and the size of the environment. The size of an environment can be measured in terms of extend, total number of geometric primitives or primitive density.

Increasing the number of users or processes in a DVE can lead to exhaustion of available network bandwidth due to the quadratic increase in

network traffic. In a DVE that uses one-to-one network connections, whenever an object is changed by one of N processes, update messages are sent to the other $N-1$ processes. Assuming that all processes change objects at the same rate this generates $O(N^2)$ update messages per simulation step. In a setup running at a simulation frequency of 30 Hz with an average of only 5 object changes per process and simulation step with an assumed 128 bytes per update message, a maximum of only 7-8 users can be supported by a 10 Mbit Ethernet in theory. In practice this number is reduced by half due to network protocol overhead. This simple approach does not scale well with an increasing number of users in the system. Many DVE applications exist that require far more than 7-8 user to interact at interactive rates in a distributed environment.

Increasing the size of the environment by adding complex objects eventually overloads the image generators, which have an upper limit on the number of polygons they are able to render. Increasing the number of polygons that have to be rendered for each frame at the same time reduces the number of frames rendered per second. Frame rates below 30 frames per second are generally not acceptable for an interactive VE application. Also, environment size can lead to memory problems if a client has to maintain a complete copy of the entire environment database.

Examples for DVE applications that require large environment databases include, for example, whole earth visualization projects, where a virtual globe with arbitrary zooming capabilities is presented to the user and database sizes easily exceed several gigabytes.

This thesis argues that scalability with respect to number of users and environment size is a key issue that needs to be addressed by a general purpose DVE framework. The design of a novel scalability solution is developed and discussed.

1.2 Overview of this work

The main part of this thesis is structured in 6 chapters.

Chapter 2 gives an overview of related work in the field and introduces relevant technologies. Section 2.1 describes relevant stand-alone VE frameworks and toolkits in greater detail. Section 2.2 discusses general distributed virtual environment systems, while section 2.3 describes large-scale DVE systems and the different approaches to scalability they propose. Finally, section 2.5 presents a comparative feature analysis of the discussed systems and prepares the ground for chapter 3.

Chapter 3 is the first of the four central chapters of this work. Based on the results from the previous chapter, 3.1 defines a set of requirements for the design of a general DVE framework. These requirements guide the development of the general architecture in section 3.2. Finally, section 3.3 provides a comprehensive summary of the proposed design.

Chapter 4 describes the non-distributed aspects of the Avocado framework. Based on the general design formulated in the proceeding chapter, section 4.1 defines the details of the object and event model for Avocado and describes the basic implementation strategies. Section 4.2 develops the display device abstraction that is subsequently used to model all display devices relevant to DVE applications. Along the same lines, section 4.3 presents an abstraction for tool-based direct-manipulation interaction in DVE applications.

Chapter 5 describes the distribution mechanisms used in Avocado. Section 5.1 takes the stand-alone object model from chapter 4 and extends it into a dynamic, distributed object model that forms the basis for distribution support. The implementation details in section 5.1 show how the distribution mechanism are integrated into the stand-alone infrastructure. To illustrate the validity of the approach to provide familiar APIs to application programmers, sections 5.4 and 5.5 provide an introduction into DVE development with Avocado.

Scalability in large-scale DVE applications is the third focal point of this thesis and the main theme of chapter 6. In section 6.2 the scalability properties of traditional approaches are analyzed and discussed. Section 6.3 describes how visibility based optimization techniques are traditionally used to accelerate rendering engines in computer graphics, and is followed by section 6.4 where similar techniques are proposed to improve scalability in large-scale DVE applications. Section 6.5 covers the implementation of this approach in terms of the distributed object model presented in chapter 5. Section 6.6 finally discusses the scalability properties of this new approach.

Chapter 7 concludes this thesis. Section 7.1 summarizes the results. In order to demonstrate the validity of the concepts presented in this work, section 7.2 documents a number of applications and research projects that where successfully built on top of Avocado. Finally section 7.3 proposes some directions for further research in the area of DVE frameworks.

2. Related work

The technological field covered by DVE systems is broad. To establish the proper context for this work, several technologies and research systems must be introduced. Their presentation is structured in four categories:

VE systems: Non-distributed VE systems and standards have been in development for many years now. They have been widely adopted throughout research and industry and represent the technological foundation for all VE related research.

Distributed VE systems: Existing DVE research systems implement many different approaches to distributed application development. Their strengths and weaknesses provide valuable input for the design of a new framework. At the same time, they provide the background against which new systems must be evaluated.

Large-scale distributed VE systems: Several research prototypes attack the problem of scalability from different angles. Their relevant characteristics are outlined to subsequently allow a comparative evaluation of the new approach to scalability presented in this thesis.

Clustered rendering systems: Distributed render clusters can be considered as "close cousins" to general DVE systems. Although they serve a different purpose, their technological background is similar. This close connection is underlined by the fact that the DVE framework presented in this work is also actively being used as the basis for a distributed render cluster implementation.

This chapter describes and analyzes relevant related work from the four areas. A general strategy of this work is to extract successful concepts from existing systems and at the same time avoid their mistakes. In the context of this strategy, this chapter provides important input for the formulation of the architectural design concepts in chapter 3.

2.1 Non-distributed VE systems and standards

Many toolkits for the development of stand-alone VE applications exist today. They provide the programmer with a high-level interface to represent

complex geometry in a scene graph and to render that scene graph. The programmer is shielded from the details of dealing with low-level graphics and system APIs, and can concentrate on the development of the application logic.

Two popular and widely used toolkits are OpenGL Performer and OpenInventor from Silicon Graphics (SGI). Both are introduced in detail because of their great influence on the design of Avocado. Performer is a basic building block of Avocado and is used to manage the scene graph and perform efficient rendering on high-end, multi-processor hardware. At the same time, many features originally introduced by OpenInventor have been adopted in the design of Avocado.

While OpenGL Performer and OpenInventor are proprietary systems, it has been understood early on that open standards for file formats and APIs are essential for a broad adoption of VE technologies and applications in research, development and entertainment. Therefore, two successive standardization initiatives will be described in some detail. The Web3D Consortium has coordinated the development of the VRML97 International Standard and is currently working on the X3D specification. VRML (Virtual Reality Modeling Language) specifies a file format and object model for the description of virtual environments. X3D adds extensibility via XML and a standard API to the underlying object model with language bindings for the Java programing language and the ECMAScript language.

Finally, the VR Juggler system is representative for a slightly different approach where the application itself is responsible for the representation and rendering of geometry and application data. The VE system provides merrily services for VE specific device IO.

2.1.1 OpenGL Performer

OpenGL Performer[64] is a commercial toolkit for visual simulation and 3D graphics applications that is in widespread use in the real-time simulation industry.

Existing general purpose rendering libraries like OpenGL[59] or XGL[57] provide only a very low-level, direct interface to the hard- and software rendering capabilities of the underlying graphics subsystem. As graphics workstation push rapidly into areas that have traditionally been the exclusive province of special purpose image generators, it becomes increasingly difficult and tedious for graphics programmers to squeeze the last bit of performance out of a particular machine. Performer addresses this problem by providing a software layer called `libpr` above the low-level graphics API OpenGL, that efficiently manages geometry and state information. Performers second approach to boost rendering performance is the utilization of multiprocessing for pipelined rendering.

Efficient rendering

The first prerequisite for high performance rendering is the efficient representation of geometry. Performer provides the `pfGeoSet` primitive to represent geometry. `pfGeoSets` utilize application-supplied arrays for attributes such as vertex coordinates, colors and texture coordinates.

On high-end graphics machines, care must be taken to ensure that data transfer between processor and graphics subsystem is efficiently organized, otherwise the application will starve the graphics hardware, which results in sub-optimal rendering performance. `pfGeoSets` provide specialized rendering routines, one for every conceivable combination of attribute arrays, attribute bindings and primitive types. Each routine is comprised of a specific, tightly optimized rendering loop for that particular combination. In this way, Performer can provide optimal rendering performance for a large variety of geometry representations.

Graphics state commands do not modify the frame buffer, but instead configure the graphics hardware with a particular mode (e.g. shading model) or attribute (e.g. texture) that modifies the appearance of rendered geometry. Efficient management of graphics state is required for optimal graphics performance. Performers provides state management that helps the programmer to avoid redundant mode changes while rendering the geometry.

Performers `libpr` layer provides specialized graphics primitives and graphics state management, that allows for efficient utilization of available graphics hardware without special considerations and in depth knowledge on the side of the application developer.

Database hierarchy and traversal

Above the `libpr` layer Performer positions a higher-level library, `libpf`, which adds a database hierarchy and automatic multiprocessing capabilities to `libpr`.

The Performer *scene graph* is a directed acyclic graph of *nodes*. The geometric information is located in the *leaf nodes*, while the *internal nodes* provide concepts like grouping, transformation, sequencing, level-of-detail and morphing.

Most database processing is accomplished thorough *traversal* operations on the scene graph. Typically, an applications updates the scene graph and viewing parameters for the next frame and then initiates one or more traversals to generate the next frames image. The libpf layer features an object-oriented API and is implemented as a C++ library.

During traversal of the scene graph, transformations contained in `pfSCS` and `pfDCS` nodes are accumulated on a transformation stack and applied to the geometry before rendering. Thus, following the concept of *nested transformations* [12], each subtree defines its own local coordinate system.

Additionally the scene graph defines a bounding volume hierarchy. Each node has a bounding sphere that encloses the node as well as any children it may have. These bounding volumes are automatically recomputed whenever the extent of the geometry or the scene graph topology is changed. The bounding volume hierarchy allows the efficient acceleration of intersection and culling traversals.

After the application has modified the scene graph for the next frame, three basic types of traversal can be applied:

Intersection Traversal (ISECT): The ISECT traversal processes line-segment based intersection requests for simple collision-detection and terrain-following. This traversal makes use of the bounding volume hierarchy to accelerate hit testing.

Culling Traversal (CULL): The CULL traversal rejects geometry outside the viewing frustum, computes level-of-detail switches (`pfLOD`) and sorts geometry by graphics state.

Drawing Traversal (DRAW): The DRAW traversal sends geometry and graphics commands via the low-level graphics API (OpenGL) to the graphics hardware.

The CULL and DRAW traversals are completely automatic. They are triggered when the application gives the command to render the next frame. Depending on the position of the view point and the current viewing direction, view-frustum-culling can reject the majority of the geometry in a scene, and thus substantially reduce the amount of data send to the graphics subsystem.

The CULL traversal converts the culled and sorted scene graph in to an efficient display list which eventually contains the entire frame, and hands it over to the DRAW traversal. The DRAW traversal traverses the display list generated by the CULL traversal and does nothing else but issue graphics command to the rendering hardware. Because of the preprocessing already performed, this can be done very fast and efficiently.

Multiprocessing

Performer uses a coarse grained, pipelined, multiprocessing scheme which is targeted at workstations with few processors (tens), as opposed to massively parallel machines with thousands of processors. The partitioning of work is based on processing stages. One stage is a discrete section of a processing pipeline dedicated to do a specific type of work. Performers basic units of work are the different scene-graph traversals, ISECT, CULL and DRAW. Consequently, the different multiprocessing stages are build around the different traversal operations. Together with an additional *Application Stage*

(APP) they form two kinds of processing pipeline, the *rendering pipeline* and the *intersection pipeline*.

The *rendering pipeline* consists of three stages, the APP, CULL and DRAW stage. The APP stage performs application specific manipulations of the scene graph and passes the resulting scene graph to the CULL stage. Here a CULL traversal is performed that generates the culled and sorted display list which is sent to the DRAW stage. Here a DRAW traversal is performed that sends the data to the graphics hardware. Performer implements multiprocessing by assigning each stage to a separate processor for execution.

Performers pipelined multiprocessing scheme trades throughput versus latency. A fully multi-processed rendering pipeline imposes an additional rendering latency of three frames on the application. On the other hand the throughput is also roughly threefold.

Summary

Performer is a programming toolkit for building high performance, multiprocessed graphics applications. It extracts maximum performance from multiprocessor graphics workstations by using:

- Geometric data structures optimized for rendering.
- Efficient reduction of graphics mode changes during rendering.
- Pipelined multiprocessing for parallel scene-graph traversals.
- Efficient view frustum culling.

All these specific optimization approaches are well hidden from the application programmer. He normally does not need to concern himself with the details of either of these mechanisms.

2.1.2 OpenInventor

OpenInventor[68, 83, 84] is a 3D graphics toolkit with a strong focus on interaction support and extensibility. It's scene graph structure and file format was chosen as the basis for the VRML1.0[6] standard, the predecessor of the VRML97 and X3D standards.

OpenInventor is an object-oriented toolkit aimed at developers of interactive 3D graphics applications. OpenInventors main goal is to simplify the task of writing 3D graphics applications that use direct manipulation techniques. OpenInventor supports development of interactive 3D graphics application by providing an extensible set of 3D objects that support *direct manipulations* of 3D objects, allowing the user to interact with the objects in the same window as they are displayed. This approach is common in 2D applications, but was rarely used in 3D applications before.

The remainder of this section will briefly introduce the OpenInventor *scene graph*, the *3D event model* and the concept of *manipulators*.

Scene graph - nodes and actions

Like Performer, the OpenInventor scene graph is a directed acyclic graph of nodes. Nodes are containers that store object specific information in public accessible sub-objects called *fields*. Each node class defines a number of fields, each with a specific value type associated with it. Field objects provide a consistent mechanism for editing, querying, reading and writing instance data within nodes. The collection of fields completely describes the objects state.

The scene graph can be traversed by *action objects*, to perform specific operations like rendering or bounding box calculation. An application performs an operation on a scene graph by applying the appropriate action to the root node of the graph.

The exact interaction between a particular action and a particular node is determined by lookup in a two-dimensional virtual function table. This *double dispatch*[32] ensures easy extensibility in both directions. New nodes can define a new interaction with every existing action they desire to do so, while new actions can selectively specify new interactions with existing nodes. The lookup of the appropriate interaction is based on the class type of the node and the action object involved.

When applying an action to an inner node, the standard behavior is to visit the children of the group node left to right, and apply the action successively. This results in a depth first traversal of the sub-tree rooted at the original node.

3D event model

Inventor uses a simple approach to distribute user events to manipulators and other smart nodes. Window system specific events are generated in response to some user action involving the mouse or the keyboard. A special event handling action is used to distribute the events to the 3D scene. This action takes the 2D event and performs a normal traversal of the scene graph. Any node interested in such an event may process the event and indicate that it has handled it. Each node class can define its own behavior to the event handling action. The event handling traversal is stopped as soon as a node is found to handle the event.

Manipulators

The event model described above is used to integrate manipulators and other interactive nodes into the application. The general concept behind a manipulator is best described by looking at an example (Figure 2.1).

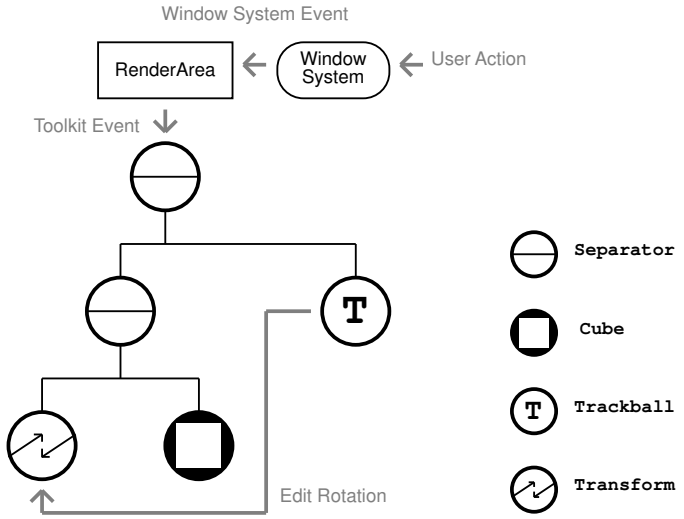

Fig. 2.1: Event model example showing a trackball manipulator applied to a cube shape (Figure from [68]).

The trackball manipulator places an invisible, but still pickable, sphere around the object it is meant to manipulate. The sphere is used to detect user interaction requests through the event model described above, and to translate mouse motion into rotational changes to the object. The trackball manipulator is implemented as a specialized separator node that registers an event handling function with the event handling action, and creates a banded sphere geometry around all of its children. The sphere is used as the handle for the interaction, while the children comprise the object that is manipulated by the trackball.

When the trackball node is encountered during a event handling traversal, it first checks whether the event is a left-mouse-down event. In that case it asks the event handling action for the object directly under the cursor. If this object is the trackball object, the event handling is grabbed. All further mouse-movement events will be delivered directly to the trackball node and there be translated into rotational movements for the manipulated object. A left-mouse-up event will terminate the event grab, and thus the manipulation of the object.

The trackball manipulator is a typical example for a *simple manipulator*, of which a variety of types are predefined in the toolkit. Inventor also supports the construction of more complex *compound manipulators* from simple ones, that are able to perform more complex interaction tasks.

Engines and field connections

The Inventor scene-graph represents a fixed, static geometry that changes only in response to user interactions by way of manipulators. To increase the expressiveness of the scene-graph Inventor introduces the concept of *engines*. Engines allow the application developer to encapsulate geometry as well as behavior into the scene-graph. Inventor engines are objects similar to nodes. The entire state of an engine object is expressed in terms of fields. Engines have an evaluation function that is executed whenever one of the fields changes. Thus, engines can be given a behavior in response to field changes. In addition to the standard fields, engines have special *output fields* that may be modified from within the evaluation function.

Engines are 'wired' into the scene-graph using connections between fields. These *field connections* copy the value of their *source field* to their *destination field* whenever the value of the source field is changed. A destination field can have exactly one source field, but a source field can have any number of destination fields. This way, engines and nodes can be wired into data-flow networks that allow the developer to model complex, animated behavior into the otherwise static scene-graph. They are first class members of the Inventor object zoo, and can be written to and read from file along with the node objects.

Summary

OpenInventor is a programming toolkit that makes it easy for developers to build rich 3D applications with direct interaction. This is achieved by providing a simple and flexible 3D event model that translates from 2D window-system events to 3D events that are delivered to objects. Based on this event model special manipulator objects are introduced, that translate the events into object manipulations. All objects are organized in a scene graph that can be traversed by action objects to perform specific functions. Through the use of engines and field connections objects with an active behavior can be represented in the scene graph.

2.1.3 VRML97

VRML97[78] is an international ISO/ISEC standard that describes the *Virtual Reality Modeling Language*, a file format for describing interactive 3D objects and worlds. Being primarily a file format definition, it does not qualify as a complete VE system, but because the underlying object and event model base on interesting concepts, VRML97 has considerably influenced the design of contemporary VE systems. The standard document ISO/ISEC 14772 describes the scope of the language as follows:

> "ISO/ISEC 14772, the Virtual Reality Modeling Language

> (VRML), defines a file format that integrates 3D graphics and multimedia. Conceptually, each VRML file is a 3D time-based space that contains graphic and aural objects that can be dynamically modified through a variety of mechanisms. This part of ISO/ISEC 14772 defines a primary set of objects and mechanisms that encourage composition, encapsulation, and extension.
>
> The semantics of VRML describe an abstract functional behavior of time-based, interactive 3D, multimedia information. ISO/ISEC 14772 does not define physical devices or any other implementation-dependent concepts (e.g., screen resolution and input devices). ISO/ISEC 14772 is intended for a wide variety of devices and applications, and provides wide latitude in interpretation and implementation of the functionality. For example, ISO/ISEC 14772 does not assume the existence of a mouse or 2D display device."

VRML97 is the direct successor of VRML 1.0 and as such draws heavily from the object model and file format defined by the OpenInventor toolkit. The objects that describe a world are organized into a scene graph. The OpenInventor field connection mechanism is adapted in form of *event routes*, which are a slight variation on field connections as they do not directly connect object fields, but can be drawn only between special event-in and event-out fields.

Another characteristic of VRML is that it is intended to be used in a distributed environment such as the World Wide Web. There are various objects and mechanisms built into the language that support multiple distributed files, including:

- in-lining of other VRML files;
- hyperlinking to other files;
- definition of an external application interface (EAI);

The EAI[82] specification defines a programming interface that is meant to be implemented by VRML97 conforming browsers in order to allow external applications to access objects in a VRML world. This would, for example, allow a VRML and HTML compliant browser to build HTML user interfaces that manipulate object in a 3D VRML scene.

2.1.4 X3D

The X3D[79] is the direct successor to the VRML97 standard and as such tries to be backward compatible by only extending but not modifying VRML97. The X3D Final Working Draft Specification describes the improvements over VRML97 as follows:

> "Extensible 3D (X3D) is a software standard for defining interactive web- and broadcast-based 3D content integrated with multimedia. X3D is intended for use on a variety of hardware devices and in a broad range of application areas such as engineering and scientific visualization, multimedia presentations, entertainment and educational titles, web pages, and shared virtual worlds. X3D is also intended to be a universal interchange format for integrated 3D graphics and multimedia. X3D is the successor to the Virtual Reality Modeling Language (VRML), the original ISO standard for web-based 3D graphics (ISO/ISEC 14772-1:1997). X3D improves upon VRML with new features, advanced application programmer interfaces, additional data encoding formats, stricter conformance, and a componentized architecture that allows for a modular approach to supporting the standard."

In particular, X3D significantly extends the VRML standard in two directions. First, it separates the definition of the data encoding from the object model and provides a space efficient binary format and an XML based file format to complement the VRML97 file format. Second, in addition to the VRML97 JavaScript interface, X3D provides several other language bindings, most notably a binding to the Java programming language, and a mechanism that allows integration with component architectures like COM and CORBA.

2.1.5 VR Juggler

VR Juggler[7, 8] is an open source VR software development environment initiated at Iowa State University.

VR Juggler is an VR application framework built around a modular *micro-kernel*. The kernel provides basic services for the implementation of *manager objects* that are the basic building blocks of the VR Juggler system. Three different classes of manager objects exist:

Internal managers: handle external device input, display configuration and user interface presentation. Specific handlers are implemented for each new I/O device or GUI toolkit that is added to the system.

External managers: handle rendering of the application state. Two implementations of a *draw manager* are provided, an OpenGL draw manager and an OpenGL Performer draw manager. The draw managers do not attempt to present a common programming interface abstraction for the supported rendering systems. Instead, an application must be targeted specifically at one of the available draw managers.

Application mangers: encapsulate the application specific functionality. An application manager depends on the available internal and external managers to implement the desired application functionality.

VR Juggler is a framework that explicitly does not depend on a specific rendering engine, but allows and requires applications to bring their own. This includes all aspects of data representation. For example, no scene graph API is provided.

2.2 Distributed VE systems

The non-distributed VE systems, that have been presented in the previous section, provide the technological background for the development of distributed VE systems. Their functionality, object models and APIs are familiar to framework and application developers alike, and are the starting point for the design of DVE frameworks and toolkits. This section presents several of the attempts to offer toolkits for distributed VE application development that have been made in recent years. The discussed systems provide various degrees of support for network based communication between the distributed processes that form an application. The discussion is focused on the distributed object models and APIs, and the network transport that is used.

2.2.1 RB2

VPL's RB2[11] system was probably the first and, in terms of market share in the early nineties, the most successful VR software ever. RB2 has support for two-user virtual reality applications, where two identical RB2 systems are linked together via a dedicated modem connection. Database duplication is not automatic and both systems must run identical copies of the database. Object attributes that are to be exchanged between the two systems have to be explicitly connected with a communication port using RB2's unique visual programming language BodyElectric. Because only two processes can communicate with each other at a time, the communication is point-to-point.

2.2.2 MR

The MR Toolkit[65] uses a simple shared memory model to allow communication between different processes. Specific memory locations can be marked as shared, and data written into these locations will be transmitted to other processes. A receiver has to explicitly receive the data and apply it to it's local database. All communication is point-to-point.

2.2.3 DIVE

DIVE's[14] distribution mechanism is based on ISIS, a fault-tolerant communication system from Cornell University. DIVE supports multi-user applications by replicating all objects in the scene graph. Objects have a standard form which comprises a simple geometry plus an optional event-driven finite state machine for behavior. More complex behavior is achieved by manipulating the objects using specifically written C code.

When a program joins a world it receives a complete copy of the current world state (all of the objects in the world) via TCP/IP. Changes to the world are propagated by update messages, which are reliably communicated to all processes in the world using the ISIS group communication toolkit. There is also a fixed set of events which includes collision, input and interaction events, plus a limited number of simple user events. These events are all broadcast to the entire world, and each process interprets them independently.

Distribution is not transparent to the application developer. DIVE applications that manipulate objects in distributed environments must use a special set of operators to do so. Because DIVE uses a group communication system, it is one of the first distributed VE systems without a central server that can give some consistency guarantees.

2.2.4 Repo-3D

Repo-3D[56] is the central component of COTERIE, a research platform for distributed augmented reality systems. It uses a replicated, shared scene graph to support distributed applications.

An interesting aspect of Repo-3D is that it is based on the Network Objects package of Modula-3 and Obliq, an interpreted language for distributed object-oriented computation. As a result, the replication and distribution abilities are part of the programming language and not part of the system.

The scene graph is built from network distributed Obliq objects. Changes to objects attributes are transparently distributed to other processes. To overcome distribution induced latency problems in interactive situations, Repo-3D allows local modifications on replicated objects that are not distributed. The local changes are accumulated and can later be transmitted to repair global consistency.

2.2.5 MASSIVE

MASSIVE[35, 36] is a distributed, multi-user VR system with a focus on scalability and heterogeneity. Distribution is based on a shared database approach that involves independent computational client processes communicating over typed peer-to-peer connections using UDP/IP. It is targeted

at virtual conferencing applications using not only graphical representations of participants but also text and live audio.

Scalability is achieved by introducing a spatial model of interaction between objects. Each object in a virtual world has an *aura* that defines the spatial extent to which interaction with other objects is possible. Interaction, and thus communication, between two objects is enabled only if their auras collide. An *aura manager* detects aura collisions and opens a peer-to-peer communications connection between the respective client processes. Modifications of object attributes are not communicated to all other objects, but only to those objects for which aura collisions exist.

Objects define several different interfaces with communication capabilities such as text, graphics or audio. Communication can only occur if compatible interfaces exist. This allows for heterogeneous combinations of clients that can communicate with the same object using the appropriate interfaces.

The combination of aura collisions and interface matching to qualify objects for communication is referred to as *spatial trading*.

2.3 Research prototypes for scalable DVE systems

The systems described in the previous section are designed to handle only moderately sized environments and a small number of users. However, many distributed VE applications require support for considerably larger environments with hundreds or even thousands of participating users. As described in section 1.1.3, the increase of environment size and number of users leads to a super-proportional increase of resource requirements. This section reviews systems that support the development of such large-scale multi-user virtual environments, and analyze their approaches to the inherent scalability problems.

2.3.1 SPLINE

SPLINE[5, 81] assembles a large virtual environment from disjunct spatial areas, so called *locales*. A locale is a bounded area of space that defines its own local coordinate system. Locales are completely independent from each other, there is no single global coordinate system. A relationship or link from one locale to another can be defined by specifying a coordinate system transformation that transforms between the two local coordinate systems. Each object in a virtual environment belongs to exactly one locale. Locales often correspond to distinct regions of a virtual environment, such as a room, a corridor or a vehicle. A complete virtual environment is composed by linking together a number of locales.

Spline is an inherently distributed system, that supports a large number of concurrent clients in a virtual environment. Each locale is associated with

a distinct set of multicast groups. All communication relevant to a locale takes place only in the associated multicast groups. Each locale is hosted by a server process that maintains the entire state of the objects contained in the locale, including their graphical representation.

By policy, a client that renders a view of a virtual environment has to take into account all objects that belong to its current locale and all objects that belong to topologically neighboring locales. To achieve that, the client joins the corresponding multicast groups and receives the current state (mostly graphical object description) that is needed to render the environment. When changing position, the client process detects if it crosses the border of its current locale and readjusts its current set of relevant locales by leaving or joining the corresponding multicast groups. This simple awareness management policy ensures that a client only "sees" those locales that are in its immediate vicinity, even if it moves within the environment.

The concept of locales potentially allows for very large distributed virtual environments because it effectively localizes communication between client and server processes.

2.3.2 CVE (MASSIVE-2)

CVE[33, 34] is the successor to MASSIVE. CVE tries to improve scalability with respect to the number of participating clients by introducing multicast communication to the spatial model of interaction used in MASSIVE.

CVE divides a virtual world into several different regions that have a defined spatial extent and associates a different multicast group with each region. Each object in a world belongs to exactly one region and can only be communicated with through the associated multicast group. A client visiting a virtual world defines a *focus object* that describes the region of space that the client would like to interact with. The client then joins all multicast groups belonging to regions that intersect with its focus object, and receives the geometric descriptions of all objects that are contained in those regions. Subsequent changes to object attributes are communicated to all interested clients using the appropriate multicast group.

Multicast groups can be explicitly represented by group objects. This allows the creation of multicast group hierarchies by recursively placing group objects into higher-level multicast groups. CVE proposes to use this to model object abstraction. In such a scenario a group object would be an abstract, simpler and more coarse representation of all the objects that might be contained in its sub-groups. A client can decide which level of abstraction of an object or region of world space it wants to interact with, by joining the appropriate multicast groups.

2.3.3 HIVEK (MASSIVE-3)

HIVEK[60] is the successor to CVE described in the previous section. It abandons the recursive multicast group approach that has been used in CVE to model abstraction and help scalability in favor of *locales* as introduced in the SPLINE system (see section 2.3.1).

HIVEK extends the concept of locales as introduced in SPLINE in two notable ways, *dynamic locale selection policies* and support for abstractions. SPLINE defines the set of relevant locales for a client to be the locale hosting the client plus all direct topological neighbors. HIVEK introduces the possibility to dynamically choose between different location selection policies that can make use of awareness and cost/benefit criteria.

N Step Selection: This is a generalization of the nearest neighbor criterion used in SPLINE where $N = 1$. Allowing different values for N can be used to adapt to the resources available on a client's machine. Clients on more powerful machines can increase N to fully utilize their resources, while less well equipped clients can concentrate their resources on the immediate vicinity.

N Nearest Selection: To prevent the possible explosion in the number of locales for lager N using the N step policy, the N nearest selection criterion can be used to put an upper limit on the number of locales.

N Most Aware, and Cost/Benefit Selection: This policy further improves resource utilization by weighting each locale according to its awareness and cost/benefit values relative to the clients hosting locale.

In addition to dynamic locale selection HIVEK proposes to use abstractions of locales to improve utilization of client resources. Abstractions in this case would be less detailed geometric representations of locales that need less network resources to transmit and less client resources to render.

2.3.4 RING

The RING[30] system uses a client-server design to implement large-scale multi-user VEs. The approach is specialized to densely occluded environments and is based on the work by Teller[70] to precompute line-of-sight visibility information for the shared environment.

Each client manages exactly one *entity* that can be freely navigated through the scene database and renders images from the viewpoint defined by the entity position and orientation. The entities have a geometric representation and are potentially visible to the other clients. Clients do not communicate the position of their entity directly to other clients, but communicate only with a server. Several servers can be located across a WAN as shown in figure 2.2 (left). Each client communicates with exactly one server,

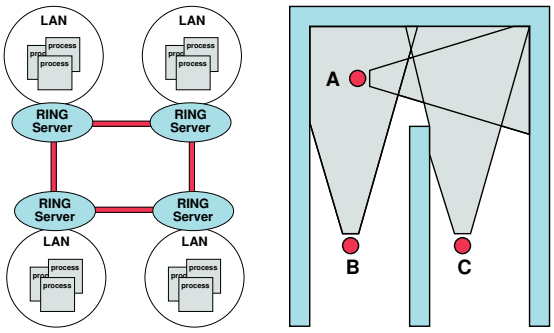

Fig. 2.2: RING client processes communicate with each other over a network of servers.

and entity update information generated by a client has to be processed by at least one server before it is delivered to other clients. The servers use the precomputed visibility information to decide which entities are visible to which other entities in the densely occluded environment as shown in figure 2.2 (right). This entity-to-entity visibility information is then used to *filter* the entity update messages between clients. Thus, clients will receive update messages only for entities that are visible to them. Because the environment is assumed to be densely occluded, this will result in a considerable reduction in overall update message traffic. By using dead-reckoning to predict entity movement, RING is able to further reduce message traffic. However, the approach has several restrictions that prevent it from being applied to a more general domain of VE applications.

- The only information that is dynamically shared between the clients is entity position and orientation. This is difficult to apply to application areas other than avatar based multi-user chat environments.
- Because the precomputed line-of-sight visibility information is static by nature, no modifications can be made to the environment at runtime, except entity movement. Every other modifications to the environment, in particular modifications to the geometry, requires recomputation of the visibility information.
- The servers do not communicate geometric information between the clients. Thus, each client needs to obtain a copy of the entire virtual environment before it can participate. This introduces a whole bunch of problems with database duplication. Besides guaranteeing that all clients obtain exactly the same version of the environment, all servers have to have access to the corresponding precomputed visibility information. Every small modification to the environment requires this information to be redistributed to all servers and clients.

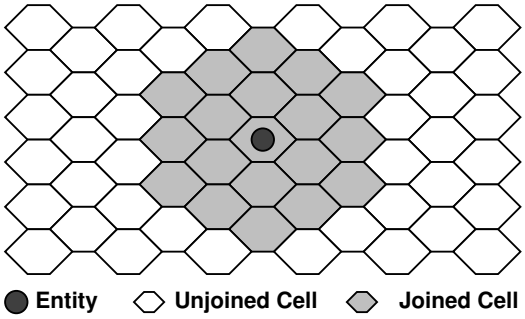

Fig. 2.3: NPSNET defines an area of interest.

- Servers have to be placed at strategic points to allow efficient operation of the system over a wide area network. Messages can only be efficiently culled, if the server connection topology closely matches the wide area network topology,

2.3.5 NPSNET

NPSNET[55, 53, 54] is a descendant of SIMNET, probably one of the first and best known large-scale VE systems. NPSNET and SIMNET both support the DIS protocol. The primary goal of both systems is the support of multi-user battlefield simulations. While SIMNET uses broadcasting to send messages between clients, NPSNET has recently introduced an extension that uses IP multicast groups to localize communication and cut down on message traffic.

The NPSNET multicast extension defines a static, hexagonal spatial partitioning of the battlefield area (see figure 2.3). Each hexagonal cell is associated with a different multicast group address. The entities that move around on the battlefield employ an area of interest manager (AOIM) that determines which multicast groups the entity process needs to join. The decision is based on the position of the entity and the radius of the area of interest. As a result an entity process only joins multicast groups that are associated with cells that are in close proximity to the entity position, and receives state update PDUs (protocol data units) only for those entities that are within its area of interest.

Similar to RING, the DIS protocol used by NPSNET is mainly used to communicate positional entity information between a large number of users. No geometric information is transmitted. Thus, all participating processes need to obtain a copy of all geometry contained in the scene before the simulation is started. This causes the same class of problems that RING has with geometry distribution.

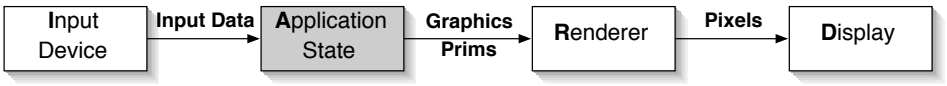

Fig. 2.4: The processing pipeline of a typical VE application: Data is gathered from *input devices* and is applied to the *application state*. Then, the application state is translated into *graphics primitives* that are transfered to the *renderer*. The renderer produces an image, the *pixels* of which are sent to the *display*.

2.4 Clustered rendering systems

Despite recent advances in accelerator technology, many graphics applications will not run at acceptable display update rates. Some application domains can be relied on to produce large datasets that defy visualization using even the most advanced and expensive graphics workstations available. For example, crash simulation applications produce geometric datasets containing several million polygons each at frame rates that easily lead to visualization requirements of several hundred million polygons per second. At the same time multi-display output devices like the Responsive Workbench or the CAVE require the simultaneous generation of up to 12 independent output signals.

Traditionally both demands have been met by using expensive graphics supercomputers like that SGI line of SMP machines, that are equipped with multiple processors and graphics subsystems. The advent of commodity PC configurations that provide graphics power comparable or even superior to a single InfiniteReality pipe at a fraction of the cost, has sparked the desire to build networked rendering cluster using readily available and affordable PC graphics workstations.

Parallel graphics system are usually classified according to the point in the processing pipeline (see figure 2.4) at which data is redistributed to multiple subsystems. The diagram shows a data processing pipeline that is typical of VE applications. Data flows from the input devices through several processing steps to the output display device. A common approach to parallelization is to duplicate a section of the pipeline and to locate those duplicated processing steps on multiple machines in a cluster, thus increasing throughput for that section of the pipeline.

Figure 2.5a shows a straight forward approach taken by some VE systems to provide simple cluster rendering capabilities. Data from the input devices is gathered on a single cluster node and broadcast to the network. Multiple cluster nodes each run the application and renderer to produce different output images. They receive the input data, apply it to the application state and render the output images in parallel. A good example for this approach is *Net Juggler* which is described in more detail in section 2.4.2.

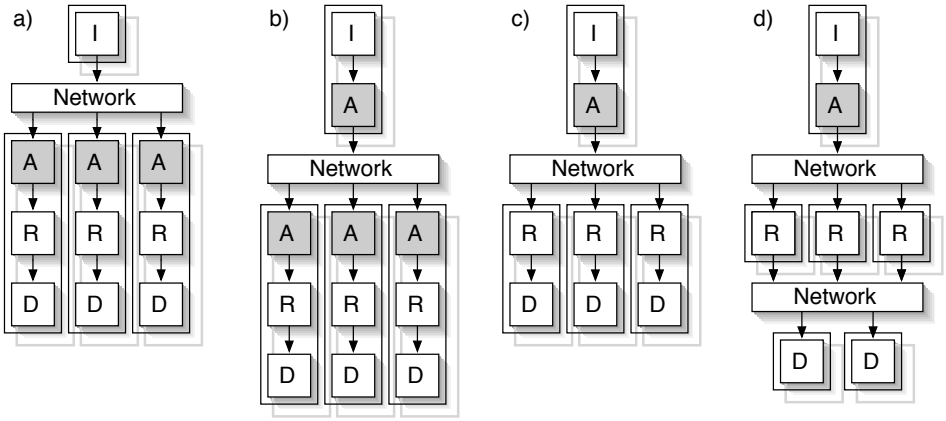

Fig. 2.5: Parallel graphics systems can be classified according to the point in the processing pipeline at which data is redistributed to multiple subsystems: a) distributed input data, b) replicated application state, c) parallel renderers, d) parallel renderers with image composition.

Consistency requires that all nodes start computation with the same initial application state, and that all application copies in the cluster react in the same way to the distributed input data. To allow for non-uniform cluster configurations, the display stages are often synchronized using swap-locking on the frame buffers and gen-locked video signals.

While this approach is simple to implement, it is only applicable to fully deterministic applications. A more flexible approach is shown in figure 2.5b. Here, the application state is replicated and distributed instead of the raw device input data. One master node in the cluster gathers the input data and applies it to the application state which is then replicated to several identical rendering nodes. The rendering nodes generate the graphics primitives from the replicated application state in parallel. Because the rendering nodes do not modify the received application state, no consistency problems arise. This approach works very well if the changes to the application state can be described and transmitted incrementally. Applications that render large time varying data sets, for example, are likely to saturate the network with the state replication messages.

If application state replication is not possible or not desirable, the master application node in the cluster can directly produce the graphics primitives and send them over the network to a set of independent rendering nodes as shown in figure 2.5c. By supplying a standard rendering API like OpenGL for the generation of the graphics primitives, parallelization of existing OpenGL applications can be achieved easily.

All three approaches described above have in common that each render-

ing node in the cluster drives a separate image output device. Such clusters can effectively be used to parallelize multi display applications such as the CAVE, the Responsive Workbench or tiled mega-displays. To allow the parallelization of single display applications using a render cluster, an *image composition network* that combines the rendered images from the cluster nodes into one image can be inserted between the render nodes and the display node as shown in figure 2.5d.

2.4.1 WireGL

An example of such a system is WireGL[42, 41, 13] that provides a parallel OpenGL interface to a render cluster built from inexpensive graphics workstations. Multiple application nodes can share the task of graphics primitive generation and concurrently feed the render cluster, while the optional use of an image composition network allows the generation of any number of display images from the output images of the clustered render nodes. WireGL has recently been transformed into an Open Source project named Chromium.

2.4.2 Net Juggler

Net Juggler[3] is based on the VR Juggler system described earlier in this chapter. It enables VR Juggler applications to run on a homogeneous PC render cluster that drives multi-screen displays like the CAVE or the Responsive Workbench.

Each rendering node in the cluster runs an identical copy of the VR Juggler application, and renders a different view of the application environment onto one of the output displays. All data input devices are managed by a dedicated cluster node that permanently reads the latest input data and broadcasts it to all render nodes in the cluster. This makes Net Juggler a concrete implementation of the cluster architecture described in figure 2.5a. The VR Juggler device managers on the render nodes are modified to not read their input directly from the devices but to receive it from the device input node in the cluster. Care is taken to assure that the single VR Juggler application does not need any knowledge about the fact that it is running in a cluster. This way, porting of existing VR Juggler applications to Net Juggler is particularly easy.

Because the rendering of different views will most probably require differing amounts of time on each render node, synchronization of the display output across all render nodes is performed. However, cross-display consistency entirely depends on the assumption that all render nodes will update their local copy of the application state identically, based on the received input data. Because VR Juggler holds the application responsible for the maintenance of the distributed application state, no implicit state sharing

between application copies is possible. If communication other than dissemination of input device data is required, the application has to implement the necessary mechanisms.

The transport layer is built around the MPI[18] (Message Passing Interface) standard implementation MPICH[37].

2.5 Summary and discussion of related work

The analysis of the VE and DVE systems provides valuable input for the development of the forthcoming DVE framework architecture. To enhance accessibility, the features of the systems are summarized and structured in three tables. Table 2.1 gives a comparative summary of the related stand-alone VE systems presented in section 2.1, while tables 2.2 and 2.3 provides a similar overview for the distributed systems from section 2.2 and the scalable distributed systems from section 2.3 respectively.

With regard to existing VE systems and especially DVE systems, a number of observations can be made:

- The scene graph is the widely accepted object model for VE toolkits and frameworks. It presents itself as the obvious foundation for the development of a DVE object model.
- Only few attempts have been made to establish a distributed shared scene graph as the object model for distributed virtual environment development. Existing systems suffer from serious problems. DIVE, for example, lacks a consistent API for the distributed scene graph, while Repo3D does not offer any provisions toward performance optimization.
- Many different event models have been used to deliver and handle events in VE applications. Traditional event dispatch methods derived from mechanism found in WIMP[1]-style user interfaces, do not apply well to the closely coupled communication needs in VE applications.
- So far, no distributed event model has been described that interfaces and integrates well with the field connection or event route approach used in stand-alone VE applications.
- Many DVE systems offer shared state abstractions to the application developer. However, using these facilities often requires significant effort. Commonly, only parts of the application state, such as transformation matrices which describe object positions, are shared between the distributed processes. Sometimes, explicit specification of communication endpoints for shared object attributes is necessary. The

[1] WIMP: Windows, Icons, Menu, Pointer

resulting problem of duplicate databases, ensuring that all processes work on consistent copies of the shared database, seriously limits the a systems applicability.

- While specialized scalability solutions exist in certain application areas or as research prototypes, none of the general purpose DVE frameworks addresses system scalability. Most existing approaches are closely tied to specific application domains and are not applicable outside their respective domain.

- The importance of rapid prototyping, and in particular script language bindings to access the object model, is undervalued in current systems. DIVE, for example, provides a partial binding to the scripting language TCL. Only a limited subset of the object model API is available and the scripting language can only be used to specify event handler responses.

- The general acceptance of a framework or a toolkit largely depends on the ability of the system to utilize existing resources and enable the development of applications with competitive performance characteristics. Although Inventor, for example, offers superior APIs and interfaces, few projects use it as a basis for immersive VE applications, because is lacks explicit support for high-performance graphics hardware. On the other hand, Performer — with its under-designed C++ API — is in wide use in both the research community and the simulation industry.

Many promising approaches to build general-purpose DVE frameworks have been described so far. However, on closer examination these systems exhibit serious drawbacks and are often not usable in real-world applications due to performance and scalability problems. The following chapter uses the presented results to develop a general architecture for a DVE framework that provides solutions to the problems found during the analysis of existing systems.

Feature	OpenGL Performer	OpenInventor	Extensible 3D (X3D)	VR Juggler
OS Platform	SGI IRIX, I386 Linux	SGI IRIX, I386 Linux	N/A	SGI IRIX, I386 Linux, Microsoft Windows
Availability	Commercial product	Open source project	Specification only, no production-quality implementation yet	Open source project
Low-Level Graphics API	OpenGL	OpenGL	N/A	Application specific
System Structure	Toolkit	Framework	N/A	Framework
Execution Model	Multiple threads	Single thread	N/A	Multiple threads
Scalability	multiple processors, multiple graphics subsystems	None	N/A	Application dependent
Display Options	Mono and/or stereo display, multiple displays	Mono or stereo display, single display	N/A	Mono and/or stereo display, multiple displays, application dependent
Object Model	Scene graph with nested transformations	Scene graph with nested transformations	Scene graph with nested transformations	None, application dependent
Event Model	None	Data-flow graph between object attributes	Event routes between objects	None, application dependent
Persistence	Scene graph persistence, Performer binary format (.pfb)	Scene graph persistence, OpenInventor file format, text or binary (.iv)	Scene graph persistence, VRML97, XML and binary file formats	None, application dependent
Data Formats	Loaders for all popular 3D file formats	OpenInventor file format, text or binary (.iv)	VRML file format, text or binary (.iv)	None, application dependent
Input Devices	Mouse, Keyboard	Mouse, Keyboard	N/A	Mouse, Keyboard, most popular VR input devices
Extension Mechanism	Subclassing	Subclassing, dynamic loading	Native object subclassing, script prototyping	Subclassing
Application Language	C, C++	C++	IDL definition, specification available for C++, Java, JavaScript	C++

Tab. 2.1: A comparative feature summary of non-distributed VE frameworks, toolkits and technologies.

Feature	Reality Built for Two (RB2)	MR Toolkit	DIVE	Repo3D	Massive
OS Platform	Mac OS (application), SGI IRIX (renderer)	SGI IRIX	SGI IRIX, Linux	Unix (HP, Sun, SGI), Windows	SGI IRIX
Availability	Commercial product, no longer available	Source code license	Binary code license	Not available	Not available
Low-Level Graphics API	Iris GL	Iris GL / OpenGL	Iris GL / OpenGL	OpenGL, Renderware	Iris GL
System Structure	Application	Set of libraries	Toolkit	Toolkit	Toolkit
Execution Model	Single thread, external render process	Single thread	Multiple threads	Multiple threads	Single thread
Scalability	None	None	None	None	None
Display Options	Mono or stereo, single display	Mono or stereo, single display	Mono or stereo, single display	Mono or stereo, single display	Mono or stereo, single display
Object Model	objects, attributes, list of objects	tagged vertex lists, state variables	objects, attributes, scene graph	objects, attributes, scene graph	objects, attributes
Event Model	data-flow network between object attributes	global events, callback subscription	global events, callback subscription	global events, callback subscription	global events, callback subscription
Persistence	Objects and relationships, no geometry	None	Servers can save environment state	No	No
Data Formats	Proprietary formats	Proprietary format, loaders for some 3d formats	Proprietary format, loaders for some 3d formats	Unknown	Unknown
Input Devices	Mouse, Keyboard, some VR input devices	Mouse, Keyboard, most popular VR input devices	Mouse, Keyboard, most popular VR input devices	Mouse, Keyboard	Mouse, Keyboard, some VR input devices
Extension Mechanism	None	Device driver interface	None	Subclassing	None
Application Language	BodyElectric (a visual programming language)	C	C, C++, limited TCL binding	Modula 3, Obliq binding	C
Distributed Object Model	Selected object attributes	Selected memory locations	Object attributes, replicated scene graph	Object attributes, replicated scene graph	Object attributes
Distributed Event Model	None	Event broadcast	State change, attachable notification handlers	None	Dedicated point-to-point connections between objects
Dynamic State Modification	Attribute modification	Variable modification	Attribute modification, object creation/deletion	Attribute modification, object creation/deletion	Attribute modification, object creation/deletion
Dynamic Membership	No	No	Yes, complete state transfer	Yes, complete state transfer	Yes
Network Transport	Modem line protocol, one-to-one	TCP, point-to-point	UDP, group communication (ISIS)	TCP, group communication (Modula 3 Distributed Objects)	TCP/IP, point-to-point
Application Layout	Peer-to-peer	Peer-to-peer	Process group	Process group	Client/server
Typ. # of Processes	2	3	5	2	5

Tab. 2.2: A comparative summary of DVE frameworks and toolkits.

Feature	SPLINE	CVE (Massive-2)	HIVEK (Massive-3)	RING	NPSNET
Distributed Object Model	Flat object set, replicated objects	Flat object set, replicated objects	Object hierarchy, replicated scene graph	Flat object set, replicates object positions only	Flat object set, replicates object positions only
Distributed Event Model	Dedicated point-to-point connections between objects	Dedicated point-to-point connections between objects	None	None	Object to object messaging
Dynamic State Modification	Attribute modification, object creation / deletion	Attribute modification, object creation / deletion	Attribute modification, object creation / deletion	Client position and viewing direction	Object position and some attributes
Dynamic Membership	Yes, complete state transfer	Yes, complete state transfer	Yes, complete state transfer	Yes, partial state transfer	Yes, no state transfer
Network Transport	TCP point-to-point, UDP multicast	TCP point-to-point, UDP multicast	TCP point-to-point, UDP multicast	TCP point-to-point	UDP multicast
Application Layout	Client / server Peer-to-peer	Client / server, Peer-to-peer	Client / server, Peer-to-peer	Client / server	Client / server, Peer-to-peer
Intended # of Processes	10^2	10^2	10^2	10^2	10^3
Partitioning Criterion	Spatial position and extend	Spatial position and extend	Spatial position and extend	Visibility	Spatial position and extend
Partition Definition	Modeling	Modeling	Modeling, run-time	Preprocessing	Modeling
Partition Selection	Topology	Spatial relationship, distance	Topology, distance	Preprocessed visibility	Distance
Recursive Partitioning	No	Yes	Yes	No	No

Tab. 2.3: A comparative summary of large-scale DVE systems. Because the presented systems are primarily research prototypes, little is known about platforms, object model, data formats, supported input devices and such. Therefore, this table only lists aspects relevant to scalability.

3. DVE Systems - A conceptual approach

Drawing from the analysis of related work in chapter 2, this chapter develops and describes an architecture for a general-purpose DVE framework. First, a set of requirements for the system is formulated. These requirements subsequently enable the identification of the major design topics and guide the necessary design decisions. Finally, a complete architecture for a DVE framework is provided.

3.1 DVE system requirements

A DVE framework has to fulfill a diverse and sometimes conflicting set of requirements. They address such specific topics as the available hardware and software infrastructure or the direct needs of the projects involved. But also more general subjects like programming paradigm and development style are important. The discussion of related systems in chapter 2 identifies a number of areas where precise requirements need to be specified. In particular these are:

- The definition of a *distributed object and event model*,
- support for a variety of *different display devices*,
- built in support for *user interaction through direct manipulation*,
- APIs and language bindings that allow *rapid prototyping* and provide *extensibility*,
- a clear specification of *target platform* and *performance requirements*.

This section formulates and rationalizes the requirements in these areas and therefore provides a foundation for the development of the DVE framework architecture in section 3.2.

3.1.1 Distributed object and event model

Support for network distributed applications is one of the key requirements for a new DVE system. The object and event model must specifically allow application designs that consist of several distributed processes that communicate over a network connection.

In general, object and event model of a VE system define four things:

- The representation of objects and state in the virtual environment.
- The mechanisms and APIs that allow modification of objects and environment state.
- The general representation of events in the virtual environment.
- The mechanisms and APIs that allow generation, delivery, and consumption of events.

In particular, a distributed object model must define an object representation that can be accessed and manipulated from network distributed processes, while a distributed event model allows the dissemination of events between those processes. Because the object model defines the application developers interface to the virtual environment, two principles must be honored to keep the learning curve flat and to increase the chance of adoption by the application developers:

Familiarity: The interfaces should enable any VE developer who has a good knowledge in VE application development using stand-alone toolkits and frameworks, to immediately develop distributed applications. Therefore, the number and complexity of new concepts to learn with respect to the distributed object and event model should be kept to a bare minimum.

Transparency: All distribution related complexity should be hidden in the object and event model, and the knowledge of further implementation details should not be necessary in order to use the distribution features.

Thus, the design must define a distributed object and event model that provides familiar interfaces for application developers and allows the creation of network transparent distributed applications.

3.1.2 Display device abstraction

The Responsive Workbench is only one of the immersive VE devices that are in use. Most notably the CAVE[16, 15]-like CyberStage is a display system that needs to supported. Recently IMK VE introduced the two-sided Responsive Workbench, which is a Responsive Workbench with a vertical back-projection screen attached perpendicular to the table-top screen. Older devices like various head-mounted displays and the BOOM[22] are also used in projects at IMK VE. As a newly conceived VE system, this framework has to support all these existing devices, while being flexible enough to adapt to any new VE output devices.

The major challenges related to display output devices are twofold. Multi-viewpoint rendering is essential for most immersive devices because

they tend to use stereo image generation to amplify depth perception. The two-sided Responsive Workbench or the CyberStage for example demand the generation of up to four different stereo-views for their display screens. A display device abstraction is needed that is expressive enough to model all these different display devices.

Besides the logistic problem of generating eight independent video signals, rendering images for the different views in real-time is a serious performance problem. Modern high-end graphics workstations tackle this problem by providing several hardware graphics acceleration sub-systems which can be used in parallel. Thus, the ability to make efficient use of multi-pipe hardware for multi-viewpoint stereo rendering is another central requirement for the AVOCADO system.

3.1.3 Direct-manipulation user interaction

All VE applications provide mechanisms for user interaction with objects in a virtual environment. For example, the Responsive Workbench provides user controlled virtual tools to manipulate objects is a natural part of the workbench metaphor. In general, the implementation of user interaction with a virtual world can be divided into two distinct steps.

Data acquisition: All user actions that describe her intentions have to be mapped into the virtual world. Input data is acquired in a variety of ways. Conventional methods include binary input devices like buttons and switches or spatial input devices that track the position of real-world objects controlled by the user. More advanced methods include voice and gesture recognition devices that recognize and translate spoken commands and articulated gestures.

Data interpretation: The raw input data that has been mapped into the virtual environment has to be interpreted and applied to the appropriate object or set of objects to evoke the desired effects.

This two step approach to user interaction needs to be supported as interaction with the virtual environment is a major feature of all VE applications.

3.1.4 Rapid prototyping and extensibility

Research in the area of VE application development is in its early stages often based on experimentation. For example, knowledge about working interaction patterns in VE applications is still very limited, and new ideas have to be verified and validated quickly. This early stage of development needs to be well supported to make a framework useful as a research tool. Developers must be enabled to very rapidly explore different ways to implement their new ideas, and need immediate feedback in that process.

The usual development cycle found in many VE systems is to long because many steps have to be performed sequentially to modify and restart an application:

Coding: This usually involves use of a text editor to compose the source code and configuration files.

Compilation: The newly edited source code is compiled into object files. Most probably a C or C++ compiler is used.

Linking: The compiled object files are linked with the VE toolkit libraries to produce an executable.

Running the application: The linked executable is run. This involves loading of application specific data like geometry and texture files.

Testing: Now the newly coded functionality can be tried out.

Quitting the application: Once the testing is done the application needs to be terminated and the cycle restarts at the coding stage.

A research system needs more direct means of prototyping new applications. For this reason, in addition to the primary compiled implementation language the framework needs a binding to an interpreted language that will eliminate the necessity to compile and link each modification, and will allow the developer to work on the running application without the need to terminate and restart it after each change. At the same time, this interpreted language must be integrated in a way that does not affect the overall performance of the application.

The IMK VE research group works on a great number of projects from very different research and application areas. The range of activities spans from entertainment applications like Caveland (Section 7.2.4) to scientific visualization prototypes like the oil exploration demonstrator 7.2.2. Other research groups work in areas like multi-modal human-computer interaction and aim to incorporate voice and gesture recognition into VR applications. To keep the amount of code and feature duplication to a minimum, it is desirable to use one software system as a basis for the implementation of all projects a group is working on.

As a result, the framework needs an extension mechanism that allows for easy addition of application specific data types, objects and functionality. It is important that adding or modifying extensions does not enforce a recompilation of the entire system or application. To properly support the rapid prototyping programming paradigm, dynamic linking and loading capabilities are desirable.

3.1.5 Platform and performance considerations

At the time of the requirements definition (1996), IMK VE used a 4 processor SGI Onyx RealityEngine[2] workstation with two graphics sub-systems to drive their immersive display devices, the CyberStage and the Responsive Workbench[46, 45, 26]. A later upgrade to a 12 processor Onyx2 InfinteReality[58] workstation with four graphics sub-systems was considered very likely at that time.

Researchers at IMK VE were normally equipped with smaller SGI desktop workstations like the SGI Indy or O2 workstations. These are single processor machines with only very moderate graphics hardware accelerators. Because they are binary compatible to the Onyx and Onyx2 supercomputers, they are well suited to serve as inexpensive VE application development workstations.

Thus, the compelling primary target platform for a VE research system at IMK VE was the entire line of Silicon Graphics workstations and supercomputers. Applications should run unmodified on all SGI workstations.

Interactive immersive VE applications need to offer a reasonable high frame update rate to be believable. For immersive display devices like the Responsive Workbench or the Cyberstage the lower limit for the frame rate should not be below thirty frames per second. In order to achieve the highest possible frame rate for interactive applications, the framework needs to fully exploit the available hardware infrastructure at IMK VE.

The main performance relevant feature of the high-end SGI graphics super-computers are:

- Multiple processors
- Multiple graphics subsystems

Thus, a very basic requirement the efficient utilization of high-end SGI graphics workstations by using the available processors and graphics subsystems to parallelize the application and image generation. Also, because the capabilities of real-time rendering systems tend to grow over time in accordance to Moore's Law, upward compatibility to new generations of graphics hardware systems is highly desirable.

3.2 From concept to architecture

The general architecture of a VE system is complex and can best be described as set of related design topics. Figure 3.1 shows the dependency relationship of the major design topics for the system. The identification of the topics in part stems from the evaluation of the related work in chapter 2 and in part is directly derived from the requirement specification in 3.1.

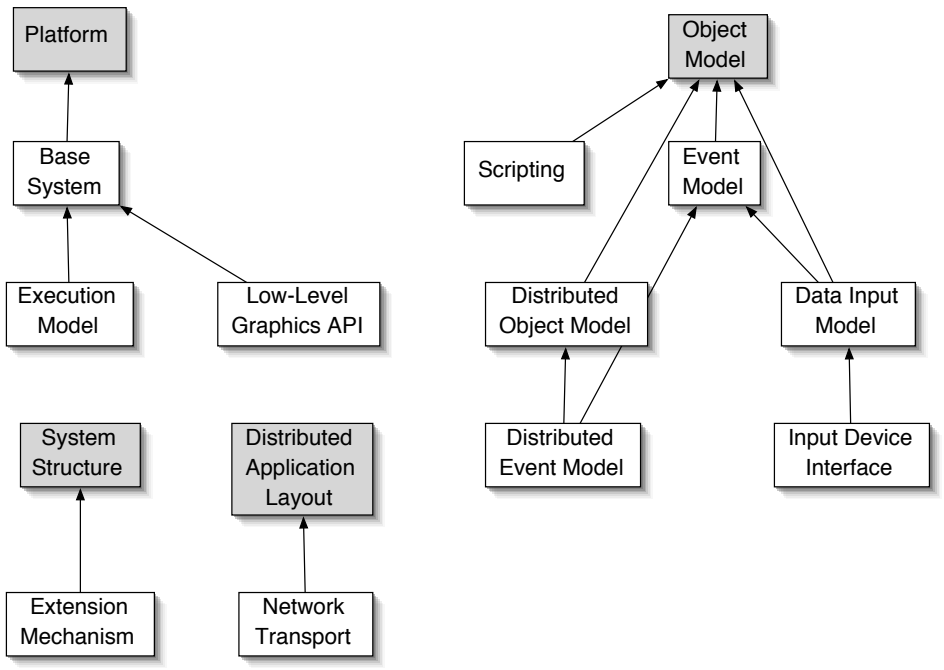

Fig. 3.1: The dependency relationship graph for the major design topics reveals the hot spots of the architecture.

The dependency analysis in figure 3.1 clearly identifies five especially relevant topics for which design decisions significantly influence the overall architecture. These are:

- Object Model
- Platform
- System Structure
- Distributed Application Layout

This section develops the overall concepts by examining each design topic in context of the requirements and the findings from related work. For each topic the possible design choices are presented and discussed. The discussion then leads to a design decision for that topic.

3.2.1 Object model

The object model of a VE system defines how the data that describes the virtual environment is represented and which interfaces and APIs developers can use to modify the environment. Because many other architectural details

directly or indirectly depend on the object model, the object model is a central part of the entire architecture.

Very few systems renounce an object oriented paradigm to describe the environment. Early versions of the MR Toolkit did use tagged vertex and polygon lists to describe geometry. This works well for virtual environments that can be described as mainly geometric in nature. In a more diverse virtual environment, more and more non geometric attributes, like behavior and other application specific attributes, are added to parts of the environment. Such virtual environments are best described as a collection of objects that have certain attributes and relationships, which directly leads to an object oriented approach.

RB2 and Massive both describe the environment as flat collections of typed objects that have certain attributes. While this is an improvement over vertex lists, flat object lists do not allow to easily aggregate objects, i.e. describe objects as a composition of sub-objects, which is a very natural and powerful metaphor to describe complex environments. A better approach, taken by systems like Performer, Inventor or DIVE, is to use a *scene graph* to relate all objects in a directed, acyclic graph.

A scene graph represents an entire environment as the aggregation hierarchy that culminates in a single *root node* which serves as starting point for recursively defined operations. Together with transformation nesting and hierarchical bounding boxes the scene graph allows the implementation of very efficient rendering algorithms, and as such is an established choice as the representation organizing structure in contemporary VE systems.

Following the object oriented paradigm, scene graph nodes are instances of *node classes*. The node classes are organized in an inheritance hierarchy, which provides powerful mechanisms for the extension of existing classes and reuse of existing functionality. Further, because common class interfaces and functionality can be factored into common base classes, the implementation of generic operations becomes possible. For example, toolkits like Performer and Inventor define generic traversal operations for a scene graph. These are implemented on the level of *plain nodes* and *group nodes* but work without modification for scene graphs built from instances of all derived node classes.

The ability to formulate generic operations for the entire representation of a virtual environment is a basic mechanism necessary to provide general extensibility, as new object classes can be added to a framework without requiring reimplementation of already existing operations. Adding *introspection* or *reflection* to the object classes further increases the expressiveness of generic operations. The Inventor toolkit, for example, provides generic persistence for the entire scene graph based on the ability to completely describe the state of any object, namely the values of its attributes, through a reflection interface. Performer on the other hand lacks the introspection ability. As a consequence the Performer implementation of persistence can handle only core object classes, application specific extensions are not auto-

matically included in Performer binary format files.

Inventor represents nodes in the scene graph as a field container which encapsulates the object state as a collection of fields. The field container interface allows generic access to all field names and values, regardless whether the class is a native Inventor class or is defined by the application. Fields are also directly accessible from the C++ API, because they are defined as standard C++ class members. Inventor demonstrates a very successful combination of the abstract field container interface that allows the definition of generic operations on objects and the class member based field interface that allows convenient access to concrete class objects from C++. This aspect of the Inventor object model is widely accepted and has been adopted by many VE systems and standards. The VRML97 standard and the proposed X3D standard are only two examples. Performer lacks this degree of sophistication in the object access API. Object access is possible via class member functions that follow a more traditional get and set pattern.

Hence, the necessary components of the object model can be characterized as follows:

Objects: The entire virtual environment is completely described as a collection of objects.

Scene graph: Object instances are organized in a directed acyclic graph. The scene graph is a familiar structure for the description of a VE and is a good basis for the implementation of performance enhancing algorithms.

Object class hierarchy: Objects are instantiated from object classes that form an inheritance hierarchy. Using classes and class inheritance to define object types allows the implementation of object based extension mechanisms while maximizing reuse of existing functionality.

Fields: Object attributes are encapsulated in typed fields. Fields are implemented as C++ data members and provide a familiar object state access interface to the C++ developer.

Field container: The field container interface allows generic access to an objects field names and values. This allows the generic implementation of operations like scripting, streaming and persistence.

More details on the Avocado object model and a description of the implementation strategy is presented in sections 4.1.1, 4.1.5, 4.1.3, 4.1.6 and 4.1.11.

3.2.2 Event model

The event model describes the representation of events in the VE system and how event generation, dissemination and consumption are handled. Events

are normally typed and carry a value that is to be delivered at some point in time. Interactive systems deliver events as soon as they are generated in order to allow the creation of low-latency feedback loops that are the basis of interactive applications. Upon delivery, events usually trigger some action, the least of which is the cause of a state change at the destination. The specification of this action is called an *event handler*.

While event generation is generally straight forward, the two important questions an event model must consider are:

- How and where are event handlers specified?
- How is an event handler bound to a specific event?

Events in a VE system can be categorized into external and internal events with respect to their origin. External events are generated outside the application environment and normally describe actions like user controlled mouse movement or spatially tracked body movements. Internal events are generated inside the virtual environment and describe things like the passing of time or object collisions. Often, internal events are generated through the consumption of external events.

Simple VE event models are derived from event delivery mechanisms traditionally used in non object-oriented WIMP user interfaces and are mostly used to deliver external events into the application. The MR toolkit, for example, specifies event handlers as global callback functions that are bound to a specific event type. The context in which the event handler is evaluated is always the entire environment, which requires that the knowledge about the events effects needs to be concentrated in one event handler.

A more object-oriented scheme allows the specification of event handlers for specific objects in the environment. Because more than one possible handler for each event type may exist, the event delivery mechanism needs to decide which handler is selected to consume the event. For example, OpenInventor defines special event handler objects that can be added to various parts of the scene graph. For each external event, the event dispatcher traverses the scene graph top-down and delivers the event to the first event handler object that will handle it. While this allows the definition of event handlers that only need a limited, local context, the relationship between event source and destination is inherently implicit.

Interestingly enough, OpenInventor introduces a second, completely explicit event dispatch mechanism that is only used for internal event delivery. By allowing type compatible fields of objects to be connected with copy-on-write semantics through *field connections*, OpenInventor builds a fine-grained event mechanism that allows the explicit specification of a direct one-to-many relationship between event source and event destinations. The event handlers are encapsulated in special engine nodes, that allow the specification of notification methods for each engine field that might receive

an event. Because this has proven to be a viable concept to express object interrelationships in complex virtual environments, the VRML97 and X3D standards both embrace the mechanism under the name *event routing*.

Because of the ability to conveniently describe the data-flow between objects in a VE application, and because the concept is familiar to many VE application developers, this design chooses field connections as the event delivery mechanism for its event model. However, it fixes two unnecessary shortcomings of the OpenInventor implementation.

First, in contrast to OpenInventor, field connections connections are used to also deliver external events to the application. This normalizes event delivery within the framework and allows the developer to implement more application specific dispatch mechanisms for external events if needed. In order to use the field connection mechanism to deliver external events to application objects, an explicit representation for the event source is necessary. Therefore *sensor objects* are introduced that map external input data onto a field based API. Sensor objects are standard field containers that expose input data values as fields. Whenever an input device changes a data value, the field value is updated accordingly. Objects that must handle input data just connect their fields to the sensors fields to receive new data values over the field connection whenever an input data value changes.

Second, OpenInventor unnecessarily requires the use of special engine objects to specify event handlers as side effects of field value changes. This enlarges the object zoo and complicates application development. Therefore, explicit specification of event notification handlers for every field container object in the environment is introduced. These handlers are invoked whenever a field on the containing object changes its value. This further normalizes the object model, as there now is no difference between an object state change through an event by way of a field connection and a direct programmatic field change through one of the available APIs, as far as object state and notification handler invocation are concerned.

A detailed description of the Avocado event model, the resulting data-flow network and its implementation is presented in section 4.1.5. The related data input model is described in section 4.1.7.

3.2.3 Distributed object model

The object model of a distributed VE system describes the representation of objects in a virtual environment which can simultaneously be accessed and manipulated by several users. This work assumes that it is important to evoke the impression of one consistent environment for all users. Therefore, the user processes need access to identical environment state information for a distributed application.

A look at the current literature reveals three increasingly sophisticated methods that attempt to present a consistent view of a shared virtual en-

vironment to several processes. Because the users are specifically allowed be spatially separated, the user processes are assumed to run on separate machines that communicate via network connections.

Variable sharing: Fragments of the environment state are synchronized between processes. Usage of these shared variables in distributed applications is comparable to the use of global variables. If one process changes the value of a shared variable, all other processes will eventually see the new value. The MR toolkit, for example, supports shared variables as a tool for the implementation of distributed applications.

Attribute sharing: A comparable but more object-oriented approach is the explicit sharing of selected object attributes. Similar to shared variables, shared object attributes have to be explicitly declared as such. This method is for example used by the RB2 and the Massive-1 systems to share the position of certain objects between processes.

Object sharing: Entire objects can be declared as shared. All attributes of a shared object are automatically shared. This includes object attributes that contain references to other shared objects, like the children attribute of scene graph nodes. Additionally, object creation and deletion are shared operations. In combination, this effectively creates a *shared scene graph* and all, even topological, changes that one process applies to the shared scene graph will be observed by all other processes. DIVE and Repo3D are examples for a shared scene graph approach to support distributed applications.

Variable and attribute sharing are not very useful as the sole distribution mechanism for general purpose DVE applications.

Because both approaches share only part of the environment state, maintenance of consistency is a responsibility of the application. In order to provide a consistent view of the shared environment all processes must obtain an identical copy of the non shared environment state at application startup. In most cases the application must provide persistent copies of the initial environment state to all participating processes. Subsequent modifications of local state elements must only be performed as a deterministic side effect of shared variable changes, as direct changes of local state elements will not be visible to other processes. Thus, only if all processes react consistently on shared variable changes with respect to their local environment copy, some degree of consistency might be achieved. Generally, both approaches do not allow additional processes to join an already running application, because the joining process would start with a virgin local application state, while already participating processes may already have performed local state modifications that the joining process can not reconstruct from the current shared variable values.

Nevertheless, some special purpose application scenarios for the use of variable or attribute sharing in the implementation of distributed applications exist. Networked multi-player versions of first-person-shooters like Doom, Quake or Unreal share only very small fragments of the application state between user processes, mostly spatial object positions. The major portion of the environment description, which can be rather large, is distributed on commercially available CDROMs. This is not a problem, because the game application does not modify the environment during game play.

Based on the intention to present a consistent view of one environment to any number of users, the object sharing approach is best suited as a basis for the implementation of a general purpose DVE system. Modifications to any shared object attributes will consistently be visible to all processes. Because object creation and deletion can also be supported as shared operations, and the entire environment state is shared, late joining processes can be supported without the need to magically reconstruct the current application state from a virgin copy. Further, because the interfaces and semantics of the shared scene graph that are exposed to the application developer are equal to their counterparts for a non shared scene graph, this distributed object model is very familiar to the application developer. Further, the reflection properties of the in section 4.1 already defined non-distributed object model allow an almost transparent implementation of the distribution functionality.

Therefore the chosen object model will be based on the concept of a transparently shared scene graph. A detailed description of the distributed implementation of field container objects and fields and the resulting consequences for the application developer are presented in section 5.1.

3.2.4 Distributed application layout

In distributed applications all participating processes need access to a shared application state. Because the processes most likely run on different machines, the entire application state has to be shared over network connections. Two different variants can be used to implement the necessary communication mechanism:

Client/Server semantics are commonly used by systems like CORBA[85] (Figure 3.2). The object state resides at one process only, while remote access is transparently handled via synchronous *Remote Procedure Calls* (RPC). This variant is most commonly used in system where access to remote objects is relatively infrequent. It incurs a significant communication overhead, as even read-only operations on objects require network communication. On the other hand, overall storage space is saved to a considerable amount as only one copy of each object exists in the distributed system. Further, consistency is

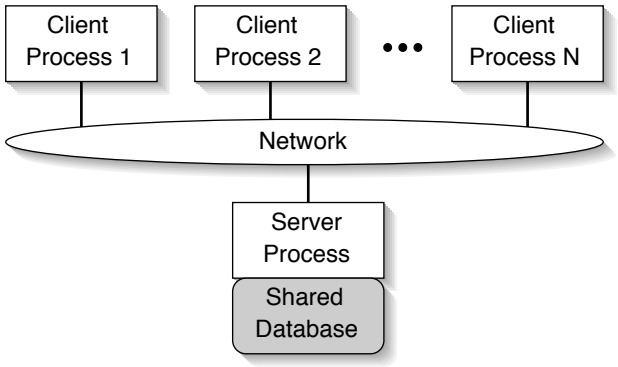

Fig. 3.2: Distributed systems with client/server semantics use a central database process that clients communicate with over a network.

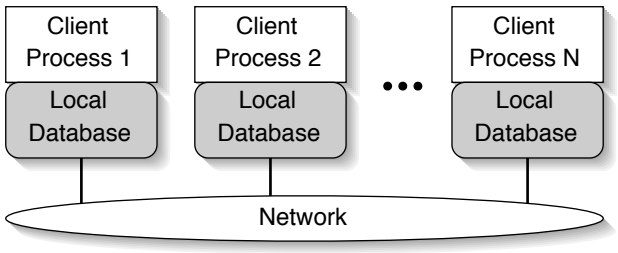

Fig. 3.3: Distributed systems with replication semantics do not use a central database. Instead, each client process holds a local copy of the database that the system keeps synchronized for all clients.

not a problem, because only one copy of an object exists in the server process. To the developer, the concept is easy to understand and to handle, because remote procedure calls are synchronous and mimic local procedure calls.

Replication semantics provide for a local copy of each object at each participating process (Figure 3.3). Object access is completely local to the accessing process, and thus can be very fast. Unlike RPC, object requests trigger communications only if the object has been modified. The replication mechanism then takes care that the object change is replicated to the other local copies of the object. Consistency has to be enforced by the implementation of the replication mechanism.

Because RPC communication is synchronous in nature and a client that requested an operation on a remote object is blocked until the request has been performed and the result is returned, the client/server model is not applicable to the domain of high-end real-time rendering, where databases of hundreds of distributed objects have to be traversed up to sixty times per

second. The high bandwidth requirements during object access for rendering necessitate that each object that is to be rendered exists in the local memory of the rendering process.

Thus, distribution support must be based on object replication because copies of the distributed objects are present in each processes local address space and can be accessed for rendering without additional communication overhead.

3.2.5 Distributed event model

The non-distributed object and event model already described in sections 3.2.1 and 3.2.2 defines event dissemination through explicit connections between object fields and provides the possibility to specify event handling in terms of general notification handlers that respond to general field value changes. Because the distributed object model guarantees that all field changes on shared objects are transparently communicated to the distributed copies, local event delivery through a field connection is automatically communicated as a field value change to all processes. Because field value changes generally - and independently of the source of the change - invoke the respective notification handler, all distributed object copies are automatically given the opportunity to react to a local field value change. Thus, events that are locally delivered to distributed objects via a field connection are effectively delivered to all participating processes.

The shared objects provided by the distributed object model, in combination with the non-distributed event model in the form of local field connections, automatically provide a seamlessly integrated distributed event model. The model is very compact and is a consistent and almost transparent extension of the non-distributed case. It does not introduce new elements to the API and therefore provides the application developer with a smooth transition path from stand-alone to distributed application development.

Section 5.2 provides details on the distributed event model and introduces a distributed locking facility, which is used to synchronize more complex interaction patterns between processes.

3.2.6 Network transport layer

The implementation of the distributed object model requires the careful selection of a network transport layer protocol. A number of different protocols are potentially available, each with different properties and performance characteristics.

Systems like Massive and MR, for example, use TCP/IP connections between processes to share application state. The TCP protocol is one-to-one connection oriented and, as a reliable protocol, guarantees data delivery even in the presence of lost packets. In a low latency requiring setup, where

each process directly communicates with each other process, the number of direct connections to be maintained grows with the square of the total number of processes. Further, if N is the number of processes, each state change requires $N-1$ identical packets to be sent over the network. This is especially undesirable if shared-media networks like Ethernet or FDDI are used. In these situations, systems that use IP multicast instead of TCP communication fare much better.

IP multicast[17, 23, 80] can be seen as a variation of the UDP protocol. Instead of being addressed to one destination, a multicast message can be addressed to a group of destinations. On LAN (Local Area Network) segments like Ethernet that support hardware multicast operation, multicasting to a group of destinations is very efficient in terms of bandwidth usage compared to sending a separate unicast message to each destination. Multicast capable routers connect LAN segments such that multicast communication is also possible across wide area networks. Because all processes listen to a common multicast address, each state change only requires one update message to be sent that is received by all processes, regardless of the total number of processes involved. Unfortunately, IP multicast is an inherently unreliable protocol. Any delivery guarantees required by an application have to be implemented on top of it. For this reason, several *group communication protocols* that are based on IP multicast have been developed.

Group communication protocols address the following problems that are not known in point-to-point communication between just two processes:

Reliability: Reliability deals with recovering from communication and site failures such as buffer overflows, missed packages and network partitions. For example, depending on the network transport used, not all processes will always receive all messages sent. This is often the case in WAN settings where the reliability of the routes to different destinations may vary greatly. Reliability is more difficult to implement for group communication than for point-to-point communications.

Consistency: Even if reliable delivery of messages is guaranteed, not all processes will necessarily receive the sent messages in the same order.

Synchrony: In the presence of group membership changes, messages may or may not be delivered to leaving or joining members. This is a source for inconsistencies.

The *process group model* [9, 10] has proven to be a good abstraction for communication between several processes. It addresses the above mentioned problems and provides reliable communication between distributed processes. The concept of a *process group* describes a group of processes that intend to communicate with each other using one-to-many semantics. Each process can send a message to the group at any time, while each message sent

to the group is received by all processes in that group. Lost messages are detected and re-sent, such that the application can regard message delivery as reliable.

Group membership is dynamic as processes can join or leave a group at any time. The current list of group members, called a *view*, is maintained and is replicated to all group members. The view is an ordered list, and the perceived order is the same for all processes. Any membership changes invalidate the current view and a newly constructed view is distributed to the group members. Along with the view, an application specific view state can be replicated to all new members during a view change. This is called *view atomic state transfer*. Within a view the state is guaranteed to be the same for all members. Messages to the group are delivered *view synchronous*. To guarantee synchrony, a message is delivered to all processes in the same view. Further, each message will be delivered in the same view it was sent in. During view changes when members leave or join the group and the member views may be temporarily inconsistent, no messages are delivered. Join and leave operations are said to be *view atomic*.

Messages delivered to a group inside a view can either be *unordered*, *FIFO-ordered* or *totally ordered*. If messages are unordered, no particular order within a view is guaranteed. Messages can even arrive in a different order at each member. FIFO-ordering guarantees that messages are received in sender order. If a sender sends message m_0 before m_1, then every member will receive message m_0 before m_1. Total ordering guarantees that all messages are FIFO-ordered and are received in the same order by every member of the group.

Group communication protocols provide strong ordering and delivery guarantees. Based on these guarantees, the implementation of a consistently shared application state becomes possible, and because IP multicast is used, it can be done very efficiently. Toolkits like DIVE and Repo-3D, for example, use group communication systems as the network layer for their implementation of object distribution. Therefore, the implementation of the object model must be based on a group communication system.

3.2.7 Data input and device interface

User input data for immersive virtual environments is usually gathered by external controller hardware that is connected via serial RS232 or network connections, e.g. space trackers, data gloves and joystick devices. Often, opening a connection to each of these devices requires the obedience of time consuming initialization protocols that lead to connection setup times of several seconds[1].

[1] The quality of the firmware of many of these devices is exceptionally bad. For example, early firmware versions of the Ascension Flock of Birds magnetic space tracker where not able to survive repeated connection attempts over the serial port without power-cycling

Many VE toolkits, like MR and DIVE for example, integrate input device connection management directly into the application. If several different input devices are used by an application, the need to explicitly open a connection to each device can lead to unacceptably long application startup times in the order of 60 seconds and more. During phases of rapid prototyping, frequent application restarting often becomes necessary. In these phases it is important that application restart times converge against zero. This is clearly not possible if the re-opening of input device connections alone takes forever.

The solution is the use of a separate, long-running *device daemon* process. When the device daemon is started, it connects to all input devices and never disconnects until it is shut down. The application does not directly connect to the input devices anymore, but connects to the device daemon instead. The device daemon gathers the input data from the device hardware and relays it to the application. The use of efficient connection protocols between application and daemon allows the connection setup time for the application to be neglected. Thus, frequent application restarts without the time penalty of device connection setup become possible.

Therefore, the AVOCADO framework will not acquire input data directly from the input devices, but utilize a long-running device daemon that manages all input device connections. Details on the device daemon implementation are presented in section 4.1.8.

3.2.8 Scripting language selection and binding

Development of virtual environment applications, especially in the research area, often follows a highly iterative approach where applications are not even fully specified until late in the development cycle. Many VE toolkits and frameworks do not account for this situation as changes and reconfigurations require recoding in C or C++ and recompilation of parts or even the whole application.

An interpreted scripting language which has a binding to all relevant high-level object interfaces in a framework can greatly reduce the burden on the application programmer and will significantly shorten the development cycle. No recompilation is necessary and often modifications can be applied to the running application.

On the other hand, scripting languages are interpreted at run-time and can be an order of magnitude slower than compiled code. Therefore care must be taken if scripting languages are used in interactive applications in order not to destroy the interactivity with slow script execution.

There are two primary usage patterns for scripting languages in interactive applications:

---

the device between attempts. The only "supported" mode of operation was to connect to the device once, and to never disconnect.

Application scripting: During startup of an application a startup script is executed that instantiates and configures application specific components.

Event handler scripting: Scripts are registered as event handlers and are executed whenever the respective event occurs. Script execution time is critical because scripts may be executed repeatedly in every frame.

The VRML97 and X3D standards, for example, define a script language binding for the object model and use Javascript as a scripting language. Both standards use scripts exclusively for the definition of event notification handlers on objects that have been defined through the PROTO extension mechanism.

JavaScript is a good choice for a scripting language in an object-oriented environment, because Javascript is an object-oriented delegation language and the VRML object model and its PROTO extension interface can be nicely mapped to Javascript objects. Unfortunately, only very few mostly incomplete and slow implementations of Javascript interpreters have initially been available. Further, because the early Javascript implementations were not primarily designed for easy integration with existing applications, the native code interface is complex and cumbersome to use.

The late DIVE versions, for example, use Tcl as a scripting language. Like VRML97 and X3D, DIVE uses scripting exclusively to define event notification handlers. The use of Tcl for the definition of event handlers in interactive applications is especially problematic, because Tcl is basically a string replacement language that is implemented as such and is therefore especially slow. On the other hand Tcl has been designed as an application extension language and therefore has a clean native code interface that conveniently supports the definition of new scripting commands in compiled code. While being easy to use, the relative simplicity of the native code interface is at the same time a major reason for the overall slowness of Tcl. While data values and objects can internally be represented as basic C data types or as opaque handles to C or C++ structures, parameters to functions can only be passed as strings. Therefore, whenever parameters are passed as function arguments, they need to be converted to a string representation and back. An especially bad but in a VE application very common example is the passing of a four by four matrix value from native code to script and back.

In Tcl this requires the conversion of 16 floating point values to a string representation and back, just to pass the matrix value without modification. While the forced string coercion of parameter values greatly simplifies the parameter passing interface, it significantly slows down the evaluation of Tcl scripts.

Inventor and Performer do not have any scripting abilities. The entire application has to be defined in a compiled language, both toolkits support C

and C++ APIs. While this approach has no inherent performance problems, the need for compilation significantly slows down the development cycle in rapid prototyping environments.

Hence, the choice of a scripting language for an interactive virtual environment framework should fulfill the following criteria:

Language features: Despite the fact that most scripts are rather short, the scripting language must be a full featured programming language that supports an adequate set of data and control abstractions.

Binding API: The binding interface must allow efficient representation of complex C or C++ data types and structures in the scripting language and support opaque data handles to be passed as parameters and return values between script language and native language.

Performance: Although it is clear that interpreted languages can never reach the performance of compiled languages, a scripting language used in an interactive application must not be unreasonably slow. An important factor, beside the binding interface, is the garbage collection method that is used.

One of the languages that fulfills all of these criteria is Scheme[1, 19], a general purpose programming language descended from Algol and Lisp. It is a high-level language, supporting operations on structured data such as strings, lists and vectors. Scheme is a fairly simple language to learn, since it is based on a handful of syntactic forms and semantic concepts. Scheme interpreters that are written in C are small, fast and provide an easy to use binding API. Many mature implementations are available.

Scheme is one of the few contemporary programming languages that use lexical scoping to resolve variable binding. Lexical scoping is especially useful for the unobtrusive specification of context for scripted event handlers. Often, callbacks and event handlers need additional context information that is not easily available at execution time and that can not be specified through the handlers interface. Normally, global variable bindings would be used. Lexical scoping achieves the same effect without pollution of the global name space and allows a finer control over the variable scope.

Therefore, Scheme is identified as the programming language of choice to provide a scripting interface to the application developer. The design uses Scheme for application scripting as well as for event handler scripting. Details on the Scheme implementation used in AVOCADO and the application development style that stems from the combination of a C++ framework with the functional programming language Scheme are presented in section 4.1.9.

3.2.9 Target platform considerations

The choice of a hardware and operating system platform is completely determined by the corresponding requirement. Because all immersive display systems at IMK VE are driven by Silicon Graphics workstations, and the majority of developers and researchers work on smaller SGI desktop workstations, SGI hardware and software is clearly the main target platform.

Even if this constraint did not exist, the performance requirement would have led to the same decision at that time (1996), because none of the competing systems was able to deliver a comparable price/performance ratio. The graphics accelerators used in workstations from Hewlett Packard and Sun Microsystems where only able to deliver a fraction of the performance at the same price, while image generators from Evans & Sutherland could deliver comparable performance at considerably higher costs.

3.2.10 Execution model: Single vs. multi-threading

The execution model determines the number of flows of control that exist in an application. There are two fundamentally different possibilities to choose from:

- One flow of control. Often called a *single threaded* execution model.
- More than one flow of control. Often called a *multi threaded* execution model.

The fundamental difference between single-threaded and multi-threaded execution is the need for explicit synchronization if more than one thread access shared data resources. While many synchronization primitives exist on systems that support multi-threading, their use adds a considerable level of complexity to the application development that often leads to a new class of interesting and hard to find programming and design errors.

Multi-threading — on the other hand — offers a direct path to application scalability on multi-processor hardware, especially on shared memory architectures. Multiple threads of control can be scheduled simultaneously on multiple processors, and make application performance scalable with the number of available processors. This makes multi-threading especially interesting, because the targeted graphics workstations have a shared memory, multi-processor hardware architecture.

OpenInventor is an example for a VE toolkit that uses a single-threaded execution model. One main-loop sequentially handles data input, preparation and rendering for each frame. This approach has two consequences. First, only one processor is utilized by OpenInventor, even if the underlying hardware makes more than processor available to the application. Thus, OpenInventor is not scalable with the number of available processors. Second, most OpenGL implementations do not allow one process to control

more than one OpenGL graphics context. Because each separate graphics subsystem maintains its own independent graphics context, an application must allocate one process per graphics subsystem in order to generate images for multi-display output devices in parallel. Thus, as an inherently single-threaded toolkit, OpenInventor can not be used to drive image generation on multi-pipe hardware. For these reasons, OpenInventor has never been used to build applications for high-end immersive display systems.

The OpenGL Performer toolkit, on the other hand, is specifically designed to be scalable with the number of available processors and graphics subsystems. Performer uses a pipelined architecture that separates sequential tasks in the main loop into pipeline stages, and thus allows efficient utilization of available processors and graphics subsystems.

Because the framework design is targeted at high-end graphics workstations and is required to deliver the highest possible rendering performance, it needs to support a multi-threaded execution model that allows the full utilization of available processor and graphics resources.

3.2.11 A base system for low-level tasks

The major reason to consider the use of existing middle-ware as a basis for the implementation for a framework is the conservation of available development resources. A VE framework is a complex system and requires the commitment of a considerable amount of development resources. Care must be taken not to waste these resources on features for which suitable implementations already exist and are available. The saved resources can better be used to develop new and unique feature not found in existing systems.

A potential candidate must fulfill a set of requirements that directly relate to the already made design decisions. While no system that might be considered can be expected to meet all requirements, it is important that if a potential base system does not support a certain feature it does at the same time not prevent its implementation.

The platform choice suggests a closer look at systems that use the OpenGL graphics API. While this is true for most relevant systems, the degree of support for the implementation of the proposed object and event model varies greatly. Support for the required multi-threaded execution model is especially sparse. It turns out, that the only promising candidates are the OpenGL Performer and OpenInventor toolkits. Both systems are general purpose graphics application toolkits are intended to be used as middle-ware systems, while specialized VE systems, like DIVE or Massive, define restrictive object and event models that are incompatible with the proposed object model.

OpenInventors object and event model has contributed some inspiration during the design of the object model and already implements many of its features, which would qualify it as a possible choice. Unfortunately,

as described in the previous section, OpenInventor implements a single-threaded execution model and does not support the efficient utilization of multiple processors and multiple graphics subsystems. It is therefore not capable to fulfill the performance requirements.

OpenGL Performer, on the other hand, is performance optimized for multi-processor and hardware with multiple graphics subsystems. It provides a compatible object model that can be extended to fulfill the specification. Because OpenGL Performer implements all aspects of in-memory object representation, hardware resource management and efficient rendering, its use can save considerable development resources. Therefore, the OpenGL Performer toolkit is a promising candidate for the implementation of the proposed framework design.

Details on how AVOCADO implementation uses OpenGL Performers data representation and rendering capabilities as the basis for the implementation of its object and event model are described in section 4.1.4.

3.2.12 System API structure

The system structure determines how the systems functionality is presented to the application developer, and determines the system semantics on the border between native system functionality and application specific functionality. There are to different philosophies in use, the *toolkit* and the *framework* approach. The main differentiation criterion between the two is the ownership of overall flow control.

Toolkit: The application defines the flow of control. Toolkit functionality is organized in a set of libraries. Applications are built against these libraries and utilize the required toolkit functionality by calling toolkit defined functions. In the case of interactive applications the application implements the main loop and thus controls the overall flow of control.

Framework: The framework defines the flow of control. It provides a generic application skeleton that is extended with application specific functionality. Application specific extensions are built against framework libraries and are registered with the framework through well defined interfaces. The frameworks application skeleton then calls the application specific code whenever appropriate[2].

When using a toolkit, application development often starts with duplication of an example application that is subsequently modified according to the application requirements. An example is the `perfly` application that ships with OpenGL Performer. Many applications that use the Performer libraries

[2] This is often referred to as *The Hollywood Principle*: "Don't call us, we call you!".

are modified copies of `perfly`. This example modification approach mimics the use of application skeletons that are provided by frameworks, without provision of a clean interface between application and toolkit supplied code. A problem occurs when a newer version of a toolkit becomes available and an application should be updated to use the newer version. Because application specific code is interwoven with, probably modified, toolkit code from the application example, and because the original example application can be expected to have changed in the new release, in most cases an automatic upgrade of the application by mere recompilation against the new toolkit libraries is not feasible. Updated fragments of the example application will have to be identified and incorporated by hand. Therefore, long lived toolkit based applications that are meant to be upgraded to new toolkit releases require considerable development resources during upgrade.

Framework based applications, on the other hand, are easier to upgrade to new framework releases, because the border between application and framework provided code is marked by clearly defined interfaces. In many cases, recompilation against a new framework release is sufficient, provided the interfaces have not been changed. The OpenInventor Toolkit is, despite it's name, a good example for a framework. The `iview` application that comes with OpenInventor is the skeleton for most OpenInventor applications without being ever modified in application specific ways.

Frameworks also support a rapid prototyping approach to application development naturally. The developer starts with a running application that is incrementally extended and, if all goes well, it's feature set gradually converges toward the specification.

Because rapid application prototyping is a key requirement, the chosen structure is that of an application framework. This, in turn, requires the existence of an extension mechanism for the implementation of application specific functionality.

3.2.13 Extension mechanism: API and link strategy

Application development using a framework is characterized by the addition of new functionality in the form of new code to the existing framework. Two major aspects need to be defined.

Extension API: The extension API defines the calling interface between the framework and the application extension.

Link Strategy: The extension link strategy defines how and when the extension code is linked with the framework code.

Modern object-oriented frameworks use inheritance and polymorphism to define an interface for application specific extensions. An extension is represented by a class that derives from a dedicated framework base class.

The base class defines virtual functions that can be overloaded to implement application specific behavior. The compiled extension code needs to be linked with the framework code in order to be used by the application. There are two different link mechanisms that can be used.

Dynamic linking: The extension code is compiled and pre-linked into a shared library and is automatically loaded into the application at startup.

Dynamic loading: The extension code is compiled into a shared library and is loaded into to the application at run-time and under explicit control of the application.

The drawback of dynamic linking compared to dynamic loading is that the framework skeleton application must be explicitly linked against the code in the shared extension libraries, and that *all* extensions are automatically loaded at application startup, regardless of whether they are used during that invocation or not. This behavior increases application load time and memory footprint.

An example for a successful combination of inheritance based framework extension and dynamic loading is the OpenInventor toolkit. OpenInventor allows extensions to derive from and specialize any relevant class that is defined in the framework. The custom application specific objects are loaded on-demand at run-time.

Hence, the AVOCADO implementation should follow this approach. Together with dynamic extension loading this provides a fine grained modular structure even for large applications while application load time and memory footprint are kept to a minimum. Details on the extension mechanism are presented in section 4.1.4.

3.3 Avocado Architecture Summary

This chapter formulated the requirements for and developed the architecture of a general-purpose DVE framework. A number of design topics where discussed in the context of the requirements and observations from related work in the field. For each topic a solution was specified and described. The resulting architecture is summarized in table 3.1.

The following chapters discuss details of central points of the architecture, and describe a complete implementation of the discussed design in form of the AVOCADO framework.

Design Topic	Definition	Avocado Solution
Object Model	Virtual environment object representation and APIs.	Object oriented, scene graph, object class hierarchy, field container, fields
Event Model	Event representation, creation, dissemination and consumption.	Copy-on-write connections between fields, change notification on field containers is event handling.
Distributed Object Model	Distributed virtual environment object representation and APIs.	Present consistent environment to all user processes. Transparently shared scene graph. All derived field container class objects can be shared.
Distributed Application Layout	Role model for application processes.	Shared scene graph implemented by object replication. Synchronized local database copies for all peer processes. No central server.
Distributed Event Model	Distributed event representation, creation, dissemination and consumption.	Field connections are local. Distributed field notification automatically distributes events.
Data Input Model	Mapping of external input data into the virtual environment.	Sensor objects map external input data to fields. Field connections deliver data to destinations.
Input Device Interface	Low-level input device handling.	Long-running device daemon process handles input device hardware. Applications communicate with daemon via shared memory.
Scripting Language	Binding to interpreted scripting language.	Script binding for all basic field data types and derived field container classes. Language is Scheme. Application and event handler scripting.
Platform	Target system platform. Processor architecture and operating system.	High-end SGI graphics workstations running SGI IRIX.
Low-Level Graphics API	Selection of low-level graphics API.	SGI OpenGL.
Execution Model	Single-threading vs. multi-threading.	Multi-processing for performance scalability with multiple processors and graphics subsystems. Pipelined execution.
Base System	Use of existing middle-ware as basis for implementation.	SGI OpenGL Performer for in-memory data representation of geometry and multi-pipe real-time rendering.
System Structure	Structure of system APIs exposed to application developer.	Framework. Application development is framework extension.
Extension Mechanism	Mechanism for application specific extension of framework.	Inheritance based extension of framework classes with dynamic loading.
Network Transport	Selection of transport layer protocols.	Multicast based group communication with strong consistency guarantees. Ensemble/Maestro from Cornell University.

Tab. 3.1: A brief summary of the proposed architecture.

4. Avocado - implementation of the framework foundation

The system architecture described in chapter 3 is the basis for the implementation of the AVOCADO DVE framework. This chapter describes the implementation of the basic framework aspects that are not directly related to distribution, while chapter 5 describes how the AVOCADO framework is extended to the domain of distributed applications. Additionally, a closer look at the implementation of the display device abstraction and the tool-based direct-manipulation framework for user interaction is presented.

4.1 The Avocado object model

The AVOCADO object and event model closely follows the design in sections 3.2.1 and 3.2.2. *Nodes* provide an object-oriented scene-graph API which allows the representation and rendering of complex geometry. All AVOCADO objects are *field containers* that represent object state information as a collection of *fields*. AVOCADO uses *field connections*, to build a data-flow graph orthogonal to the scene graph. Based on the reflective properties of the object model, all objects support a generic *streaming* interface, which allows object state information to be written to a stream, and the subsequent reconstruction of the object from that stream. The streaming capabilities are used to implement generic object persistence for all AVOCADO objects and the entire scene graph.

Following the argument in section 3.2.11, the AVOCADO implementation is based on OpenGL Performer. AVOCADO uses Performer for scene graph management and to achieve maximum performance for graphic intensive applications. Advanced rendering tasks like culling, level-of-detail switching and communication with the graphics hardware are all handled by Performer. Performers capabilities are utilized by sub-classing AVOCADO classes from Performer classes.

In addition to its C++ API, AVOCADO features a complete language binding to an interpreted scripting language. As detailed in section 3.2.8, Scheme is the language of choice. All high-level AVOCADO objects can be created and manipulated from the Scheme scripting language.

The following sections present details of the implementation and describe

the APIs that are exposed to the application developer.

4.1.1 Mapping the field concept to Performer

The efficient implementation of generic streaming and scripting interfaces for heterogeneous objects requires additional *meta* information about object attributes and their types, and a way to access those attributes without knowing the exact type of the containing object. The C++ programming language does not treat classes as first-class objects, so this meta information is not easily available on a language level.

Performer, for example, uses a member function API to access the state attributes of an object. A symmetric pair of getter and setter functions exists for each attribute. Setting one attribute may change another attribute of that object as a side effect. However, no abstract information about the number of attributes, their type and their value can be obtained from an object via the Performer API.

As discussed in 3.2.1, AVOCADO uses *field objects* as containers for object state attributes. Fields encapsulate basic data types and can be used efficiently to provide generic streaming and scripting interfaces. They are implemented as public class members and are thus inherited by derived classes. They are directly accessible by client classes and are AVOCADO's premier interface to object state attributes.

Fields can be one of two different types. *Single value fields* contain one basic data type value, while *multi value fields* contain a vector of an arbitrary number of values.

Basic field data type encapsulation

Fields are implemented using the C++ template[69] mechanism.Templates provide direct support for generic programming, that is programming using types as parameters. The C++ template mechanism allows a type to be a parameter in the definition of a class or function. Templates do not require the different types used for instantiation of a particular template to be related in any inheritance hierarchy. This makes them particularly well suited for the implementation of fields, which are used for the encapsulation of otherwise unrelated *basic field data types*.

Nevertheless, basic field data types must satisfy certain constraints to be usable as parameters for the field class templates. In particular, they must provide the following:

- A default constructor, a copy constructor and a destructor.
- The comparison operators `operator==()` and `operator!=()`
- The assignment operator, `operator=()`

C++ built-in	C++ stdlib	Performer	AVOCADO
`bool`	`string`	`avVec2`	`avBaseLink`
`int`	`complex`	`avVec3`	`avType`
`unsigned int`		`avVec4`	`avClosure`
`long`		`avMatrix`	
`unsigned long`		`avQuat`	
`float`		`avSeg`	
`double`		`avSphere`	
		`avPLane`	
		`avBox`	

Tab. 4.1: AVOCADO initially provides four categories of basic field data types. Besides the built-in C++ types, several more complex data types from the C++ standard library are supported. Further, all classes from the Performer pfLinMath package have been supplemented with the necessary functionality for scripting and streaming. Finally some AVOCADO specific types can be used.

- Two functions `av_scheme_bundle()` and `av_scheme_unbundle()` must exist for the particular basic field data type. They convert a value to and from the desired Scheme representation for the scripting interface (see section 4.1.9).
- Two operators `operator<<(avOutputStream&)` and `operator>>(avInputStream&)` must exist. They implement streaming capabilities for the basic field data type.

AVOCADO uses twenty basic types to instantiate a default set of eighty different field types. The basic types, and the resulting field types, can be classified into one of four categories (see table 4.1). The default set of available field types represents those most commonly used in general interactive 3D applications. The application developer can define new application specific field types at any time.

AVOCADO defines four class templates for the instantiation of the different field types which are available for every basic field data type.

`avSingleField<>` encapsulates one data value.

`avSingleAdapterField<>` adapts between the AVOCADO field interface and the Performer method based interface for attribute access (see section 4.1.4).

`avMultiField<>` encapsulates a vector of data values. It can carry any number of values, and is dynamically resizable.

```
template<class T>
class avField : public avTyped {
public:
  const string&            getName() const;
  avLink<avFieldContainer> getContainer() const;
  bool                     connectFrom(avField* field);
  void                     disconnect();
  avField*                 getConnectedField();
  virtual Elk_Object       getSchemeValue();
  virtual int              setSchemeValue(const Elk_Object);
};
template<class T>
avOutputStream& operator<<(avOutputStream& stream,
                           const avField& field);
template<class T>
avInputStream& operator>>(avInputStream& stream,
                          avField& field);\end{verbatim}
```

Fig. 4.1: The abstract `avField` class provides a common interface for all field specializations.

`avMultiAdapterField<>` adapts between the AVOCADO multi value field interface and the Performer method based interface for multi valued attributes (see section 4.1.4).

All field class templates are derived from the common abstract base class `avField`. The `avField` class provides a common interface for all possible field specializations. This interface provides facilities to query information about the container a particular field, connect and disconnect other fields, and the necessary functions and operators to support streaming and scripting.

As an illustration of the field class API, part of the template class definition for the single field is shown in figure 4.1. Access to a field value is provided by the `getValue()` and `setValue()` methods. These methods with their proper signature are defined by the derived classes of `avField`. For each field class a pair of stream operators exist, allowing serialization of the field value into a stream, and the reconstruction of the field value from a stream.

Single-value fields

Figure 4.2 shows part of the `avSingleField` interface. `avSingleField` is a template class and is used to instantiate single value fields on basic data types. It defines appropriate setValue() and getValue() methods according to the template class type parameter.

```
template<class T>
class avSingleField : public avField {
public:
  void          setValue(const T& value);
  const T&      getValue() const;
};
```

Fig. 4.2: `avSingleField` is a template class and is used to instantiate single value fields of basic data types.

```
template<class T>
class avSingleField : public avField {
public:
  void             setValue(const T& vector<value>);
  const vector<T>& getValue() const;
};
```

Fig. 4.3: `avMultiField` is a template class and is used to instantiate multi-value fields of basic data types.

Multi-value fields

AVOCADO makes extensive use of the Standard Template Library (STL). Multi-field values are implemented as STL vectors of the field type. Figure 4.3 shows the corresponding template class declaration for multi-value fields.

4.1.2 Field containers for object state encapsulation

AVOCADO objects are *field containers*, which represent object state as a collection of fields. A field container can be queried for the number of contained fields and their references. Relevant parts of the field container interface are shown below. This, together with the generic streaming interface of fields, allows us to provide streaming functionality at the field container level without knowing the exact type of the underlying object. This extends to all classes further derived from `avFieldContainer`.

AVOCADO's field container interface for object representation provides a solid foundation for the subsequent, generic implementation of advanced features like data-flow computing, scripting and distribution. These capabilities are inherited by any application specific extensions added by the programmer. Because AVOCADO nodes are sub-classed from Performer nodes, both node types can be freely combined to build the scene graph.

The central method of the field container interface is the getFields() method. getFields() returns a generic vector of all fields contained in a field container, regardless of the concrete type of the field container. Together with the ability of the avField class to denote the name and the value of any

```
class avFieldContainer {
public:
  int             getNumFields();
  avFieldPtrVec&  getFields();
protected:
  virtual void notify(avField& field);
  virtual void evaluate();
};
ostream& operator<<(ostream& stream,
                    avLink<avFieldContainer>& fc);
istream& operator>>(istream& stream,
                    avLink<avFieldContainer>& fc);
```

Fig. 4.4: The `avFieldContainer` encapsulates the state information of an object and represents it as a collection of fields.

field as string values, this allows the easy implementation of the streaming operators that are in turn the basis of the implementation of persistence.

4.1.3 Smart pointers and reference counting

One of the most prominent sources of serious errors in C and C++ programs is related to memory management and the use of pointers. In C++ every object instantiated from heap memory, has to be explicitly deleted in order to release that memory. As heap objects are handled via memory pointers and those can be freely copied, the question of who is finally responsible for proper deletion of an object arises if there is more than one reference to that object. This leads to three potentially fatal possible errors:

Orphaned objects: This happens if all references to an object go out of scope, without the object being deleted. There are no further references to the object and there is no way the memory used by the object can be released until program termination. This leads to memory leaks which will eventually drain the memory resources to nil.

Multiple object deletion: If more than on client calls delete on one and the same object, the integrity of the internal memory management structure is bound to be seriously damaged. This leads to subtle errors which are very hard to find without special debug tools.

Dangling pointers: There is no way to tell whether a pointer reference to an object is valid, or if the object has long been deleted. Access through the invalid pointer may easily damage the internal memory structure and most likely cause hard to find fatal errors.

One way to overcome all these problems is the use of *handle objects* (see [21, 69]). A handle encapsulates a pointer to some object class and allows

```
class avBase : public avTyped {
public:
  void reference() {_use_count++;}
  void unreference() {if (--_usecount == 0) delete this;}
private:
  unsigned int _use_count;
};
template<class Type>
class avLink : public avAnyLink {
public:
  avLink(Type* object_ptr) {
    _ptr = object_ptr;
    _ptr->reference();
   }
  ~avLink() {_ptr->unreference();}
private:
  Type* _ptr;
};
```

Fig. 4.5: The abstract base class `avBase` implements the reference counting mechanism that AVOCADO uses by way of `avLink<>` to implement its memory management scheme.

to separate the objects interface from its implementation. Handles can also be used to provide memory management for the object class. By including a *use count* on the object class it is easy to implement a *reference counting* scheme for the handle-object combination. Whenever a handle is created that references an object, the use count on the object is incremented. Thus, the object knows how many references to it exist at any time. When a handle goes out of scope, i.e. is destroyed, the use count on the object is decremented. When the use count reaches zero, there cannot possibly exist another reference to the object and it can safely be deleted.

The `avLink<>` class, as shown in figure 4.5, is used to implement *reference counting* as a memory management scheme on AVOCADO objects. The object maintains a a use count that records the number of currently active references to it. Whenever a new handle is instantiated, the use count on the referenced object is incremented via the `reference()` function. Likewise, whenever a reference is deleted, the use count is decremented. When the use count reaches zero, the object is automatically deleted.

The template class `avLink<>` defines a typed smart pointer (or *handle*) to AVOCADO objects which hides reference counting details from the programmer and greatly reduces the difficulty of writing exception-save code. Because AVOCADO uses handles exclusively to reference objects, dangling pointers and memory leaks are no problem in AVOCADO applications, which makes them extremely stable and reliable.

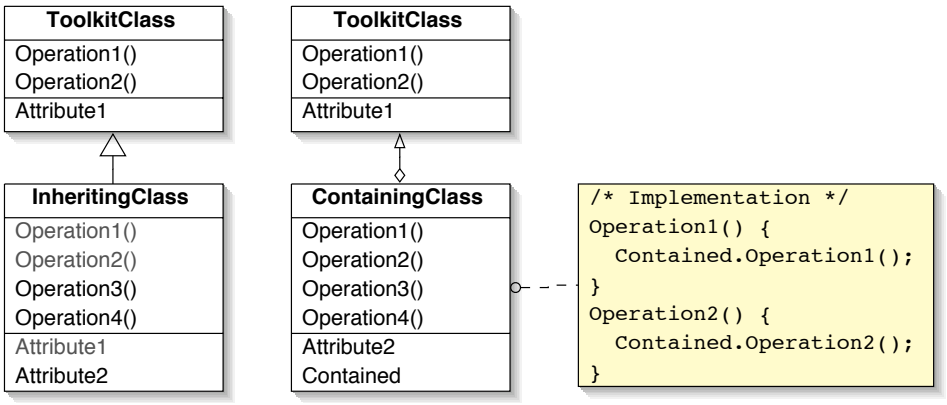

Fig. 4.6: Extension and reuse by inheritance and containment. Both methods have certain advantages and disadvantages.

4.1.4 Adaption of Performer classes through subclassing

Extension of an existing object-oriented framework or toolkit generally follows one of two possible paths, *inheritance* or *containment* (see [69]). Figure 4.6 illustrates both possibilities.

With the inheritance scheme, functionality is added to the classes of the core toolkit by subclassing. New classes are derived from the classes of the core toolkit. The extended classes still provide the interface and signature of their base classes and can be used as such, while adding new interface elements. The addition of a new interface is most conveniently implemented with multiple inheritance.

The containment scheme implements a new, *mirrored* version of the original class hierarchy. Each new class contains a pointer or reference to the corresponding class object from the core toolkit as a private member. The interface of the core class is replicated in the new class and member function calls are forwarded. Further on, core classes are no longer used directly, but only through their containing adapters. This implies, that it is no longer possible to freely mix and match instances of core classes with instances of new classes. Containment does not require the use of multiple inheritance, which from the beginning would avoid a number of problems that can arise using multiple inheritance with C++.

The use of multiple inheritance in C++ can use to phenomenons like the *Diamond of Death*, shown in figure 4.7, where a class XYZinherits from two classes Y and Z that have a common superclass X in their derivation hierarchies. This causes XYZ instances to have two or more instances of X embedded within them. This causes ambiguity with regard to the data and function members associated with X, as well as possible storage duplication. On the other hand, it makes the combination of two classes to form a third

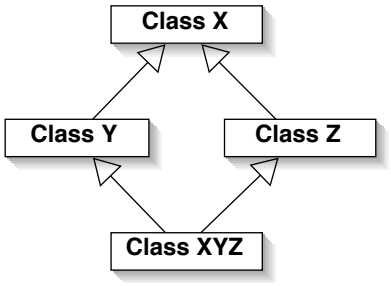

Fig. 4.7: The *Diamond of Death* is a common phenomenon that occurs when multiple inheritance is used in conjunction with poor design.

much easier.

Because of its ability to freely mix Performer class objects and AVOCADO class objects to build the scene graph, AVOCADO follows the first approach. Each concrete Performer class in the scene graph API is sub-classed, and the resulting AVOCADO class provides an AVOCADO specific interface in addition to the standard Performer interface. The AVOCADO interface is *mixed in* via multiple inheritance.

Node subclassing

By subclassing from the concrete Performer scene-graph API classes and using multiple inheritance to mix in the `avFieldContainer` class, Performer objects are provided with the field oriented AVOCADO API. In other words, the AVOCADO scene graph API inherits the structure and functionality of the Performer scene-graph API to build concrete classes on top of its abstract field container interface. To differentiate between the native Performer classes and the derived AVOCADO classes, all AVOCADO classes replace the Performer specific name prefix `pf` with the prefix `av`.

Adapter fields are used to bridge from the Performer method based API to the AVOCADO field oriented API. They will forward `getValue()` and `setValue()` requests to the appropriate getter and setter functions of the Performer API. This ensures that Performer related state information is correctly updated according to field value changes, and possible side effects are properly evaluated.

The following example shows the subclassing procedure for a `pfGroup` node. *pfGroup* has a partially inherited interface as shown in figure 4.8. An `avGroup` class is derived from `pfGroup` and `avFieldContainer` and equipped with a suitable collection of adapter fields (Figure 4.10). The `Name` field implements its `getValue()` and `setValue()` methods in terms of calls to `pfGroup::getName()` and `pfGroup::setName()`. The `Children` field shows the adaption of a multi-field to a whole range of Performer access member

```
class pfGroup {
 int         setName(const char *name);
 const char* getName(void);
 int         addChild(pfNode *child);
 int         insertChild(int index, pfNode *child);
 int         replaceChild(pfNode *old, pfNode *new);
 int         removeChild(pfNode* child);
 int         searchChild(pfNode* child);
 pfNode*     getChild(int index);
 int         getNumChildren(void);
};
```

Fig. 4.8: The Performer `pfGroup` interface uses member functions to query and alter object state.

```
class avGroup : public pfGroup,
                public avFieldContainer {
public:
  avMultiAdapterField<avLink<avGroup> > Parents;
  avSingleAdapterField<string>          Name;
  avMultiAdapterField<avLink<avNode> >  Children;
};
```

Fig. 4.9: The interface of the AVOCADO node `avGroup` is field based.

functions used to manage the children of a group node. The return type of `Children.getValue()` is a STL vector of `avLink<avNode>`.

AVOCADO objects map the member function interface for object state access to a field based interface.

Field subclassing

Creating of new field types is simple. The `avSingleField<>` and `avMultiField<>` templates are used to define new field types. The new basic type supplied to the templates must support a small set of operations:

`const type& operator=(const type& value)`: A field basic type must be assignable.

`int operator==(const type& value)`: The equality operator. C++ simple types like `int` and `float` are all comparable, while new classes must explicitly define this operator.

`istream& operator>>(istream& is, type& value)`: The stream-in operator defines how a value of the basic type is read from a stream.

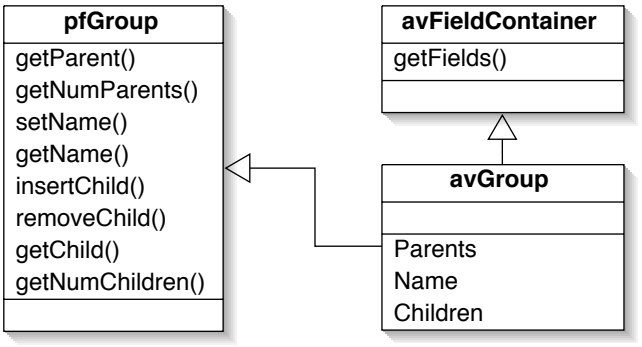

Fig. 4.10: The method based interface of the `pfGroup` class is mapped to the field based interface of the `avGroup` class.

Together with the stream out-operator it defines the stream representation of the type.

`ostream& operator<<(ostream& os, const type& value)`: The stream-out operator defines how a value is written to a stream.

`Scheme av_scheme_bundle(const type& value)`: This function converts a value into a suitable scheme representation. Simple types like `int` and `float` are converted to their scheme counter-parts. More complex types that have no comparable scheme representation, can be implemented as opaque scheme objects, such that they can be handled but not directly manipulated from scheme.

`int av_scheme_unbundle(type& value, Scheme)`: This function converts a scheme value to a value of type `type`. This function can fail if both types are not compatible.

4.1.5 Field connections for event dissemination

As pointed our in 3.2.1Avocado uses the concept of *field connections*. Fields of compatible type can be connected such that whenever the value of the *source field* changes, it is immediately forwarded to the *destination field*.

Data-flow graph

Using field connections, a data-flow graph can be constructed which is conceptually orthogonal to the scene graph. Avocado utilizes this mechanism to specify additional relationships between nodes, which cannot be expressed in terms of the standard scene graph. This facilitates implementation of interactive behavior and the import of real world data into the scene graph.

```
class avField {
public:
  int connectFrom(avField* field);
  void disconnect();
  avField* getConnectedField();
};
```

Fig. 4.11: The field connection API of the *avField* class consists of three methods.

Figure 4.11 shows the field connection API that is part of each field class interface.

A field can be the receiving end of at most one field connection, while any number of fields can receive their values from a single field. In other words, the *fan-in* of a field is 1, the *fan-out* of a field is n.

Evaluation

The evaluation of the data-flow graph is performed once per rendering frame. The starting point of the evaluation are the fields on the sensor objects that incorporate data from real-world devices into an application. If the user, for example, presses a button on a workbench stylus, the corresponding `Trigger` field on the tracker sensor object changes value and initiates an evaluation. The new field value propagates along the field connections until all connected fields have received the new value. Field connections forward value changes immediately, so that there is no propagation delay for cascaded connection paths in the graph.

Loop detection

A data-flow network can contain loops. These must be detected and properly resolved in order to prevent infinite notification loops that would effectively hang the entire application. To achieve this, a field will only receive and forward at most one new value for each evaluation cycle. If a loop existed, value forwarding would stop in the moment the first field in the loop receives the forwarded value for the second time.

Notification

A field container can implement side effects on field changes by overloading the `notify()` and `evaluate()` member functions. Whenever a field is set to a new value, the `notify()` method is called on the field container with a reference to the changed field as an argument. The field container can do whatever is necessary to achieve the desired effect, including the manipulation of other fields. After all notifications for all fields on all field containers

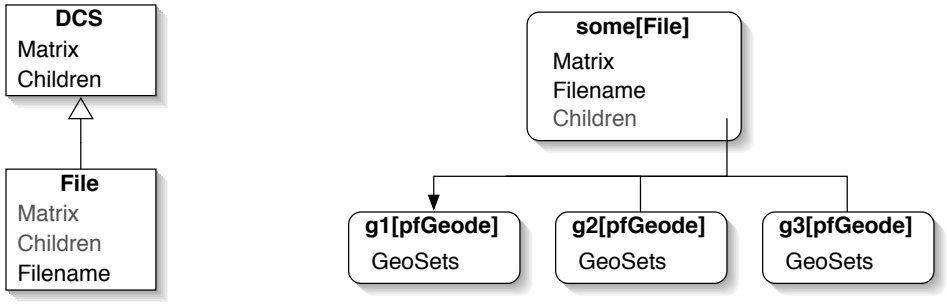

Fig. 4.12: The `avFile` node.

have been made for one frame, the `evaluate()` method is called on each field container which had at least one of its fields notified. This allows the field container to perform actions which depend on more than one updated field value for each frame.

4.1.6 Scene graph node classes

As pointed out in section 4.1.4, field container adaptions exist for all Performer node classes (`pfGroup`, `pfDCS`, `pfLOD`, ...) and most of the Performer object classes (`pfGeoState`, `pfMaterial`, `pfTexture`, ...), which together represent the Performer scene-graph API.

An example for a node extension: `avFile`

The ability to mix Avocado nodes with Performer nodes to construct the scene graph can be conveniently used to define new nodes with additional functionality. The `avFile` node, for example, is derived from the adaption node `avDCS`. It inherits the interface of `avDCS`, which basically consist of a `Children` and `Matrix` field. The `avFile` node adds a `URL` field of type `string`. With an overloaded `notify()` method, `avFile` reacts to changes of the `URL` field by retrieving a geometry from the specified URL and adding that geometry to its list of children (See figure 4.12).

Thus, `avFile` imports the geometry into the scene graph, and can be regarded as an opaque handle to it. Subsequent changes to the `URL` field replace the old with the newly specified geometry.

4.1.7 Sensor objects

Sensors represent Avocado's interface into the real world. They are derived from the `avFieldContainer` class, but not from any Performer node, and thus cannot be inserted into the scene graph. Sensors encapsulate the code necessary to access input devices of various kinds. The data generated by a device is mapped to the fields of the sensor (Figure 4.13). Whenever a

```
class avTrackerSensor : public avDeviceSensor {
public:
  avSingleField<string>   Station; // inherited
  avSingleField<avMatrix> Transform;
  avSingleField<bool>     Button;
};
```

Fig. 4.13: The AVOCADO sensor classes map data values from external devices to fields.

device generates new data values, the fields of the corresponding sensor are updated accordingly. Field connections from sensor fields to node fields in the scene graph are used to incorporate device data into an application.

AVOCADO uses the term *sensor* with a different meaning than it is used in OpenInventor (see section 2.1.2). OpenInventor uses the term *sensor* to describe objects that can be attached to other objects to monitor attribute changes on those objects and trigger application defined actions. Even arbitrary changes to scene graph subtrees can be monitored using sensors. ivview, a simple OpenInventor viewer, for example, attaches a sensor to the root node of the scene graph. The sensor fires whenever the scene graph is changed by the application and initiates a redraw of the scene. In AVOCADO the term sensor to refer to objects that interface to the real world and incorporate real-world data into the application.

An example for a sensor object: `avTracker`

The `avTrackerSensor` class, for example, provides an interface to a six-degree-of-freedom space-tracker like the Polhemus Fastrak or the Ascension Flock of Birds.

4.1.8 An external device daemon process

AVOCADO utilizes a device daemon process, which handles the direct interaction with the devices via serial line or network connections. The daemon updates the device data values into a shared memory segment, where the `avDeviceSensor` classes can access them. A *station* name is used to identify the desired device data in the shared memory segment, and every `avDeviceSensor` class specifies this identifier in it's `Station` field. After connecting to the device daemon, the `avTrackerSensor` class provides the current position and orientation information from the selected tracking device represented as a matrix in its `Transform` field. By connecting the `Matrix` field of an `avDCS` node to the `Transform` field, the subtree rooted by the `avDCS` node will move according to the tracker movement reported from the selected station. Many `avSensor` classes have been defined to support a

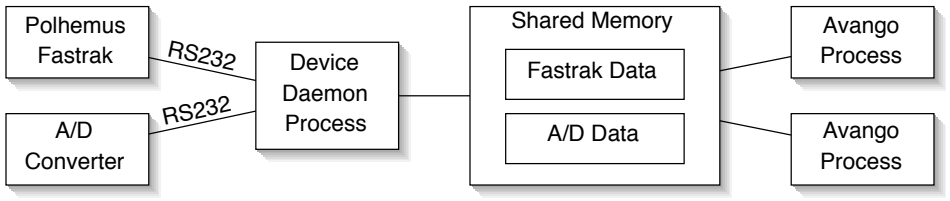

Fig. 4.14: The device daemon process is started during the boot phase of the operating system and provides low latency data from the various devices to its clients on a permanent basis. Clients processes can connect and disconnect at any time.

```
(make-ident-matrix)
(make-trans-mat    tx ty tz)
(make-scale-mat    sx sy sz)
(mult-mat          mat1 mat2 ... matn)
(invert-full-mat   matrix)
(set-mat-row       matrix row value-list)
(get-mat-row       matrix row)
```

Fig. 4.15: The Scheme binding to the `avMatrix` class allows creation and manipulation of `avMatrix` values from Scheme. The `make-...` functions create a new matrix value, while the other functions perform manipulations on provided matrix values.

great variety of input devices, such as mouse, keyboard, trackers and gloves.

4.1.9 Integration of the ELK Scheme language

As pointed out in 3.2.8, AVOCADO implements a complete language binding to the interpreted language Scheme. AVOCADO uses the ELK [51] Scheme implementation, which is a small and elegant Scheme interpreter and is especially suited to serve as an extension language for C and C++ programs. The AVOCADO scheme binding is based on the field and field container APIs of the AVOCADO objects.

For all basic data types which are used to instantiate field classes a scheme representation with all necessary accessor functions exists. The basic data types are passed by value to and from the Scheme interpreter and can be handled like any other built-in Scheme type. Figure 4.15 shows a subset of the Scheme binding for the `avMatrix` basic type.

AVOCADO objects like nodes and sensors are handled by reference. This is implemented by providing a binding for the `avLink` class. `avLink` values are again passed by value to the Scheme interpreter so that references to AVOCADO objects are properly reference counted. Scheme uses a garbage

```
;; the objects
(define tracker (make-instance 'avTrackerSensor))
(define file    (make-instance 'avFile))

;; configure the tracker
(av-set-value tracker 'Station "head-tracker")

;; load some geometry into the file node
(av-set-value file 'Url "http://viswiz.gmd.de/head.iv")

;; build the scene graph, and connect the tracker
(av-set-value scene-root 'Children (list file))
(av-connect-from file 'Matrix tracker 'Transform)
```

Fig. 4.16: An example scheme script that loads an Inventor geometry, adds it to the scene graph and draws a connection to a tracker sensor.

collector to reclaim memory from objects which can no longer be accessed by the interpreter. When an `avLink` value gets garbage collected, the reference count on the associated AVOCADO object is decremented properly.

AVOCADO objects can be created from Scheme by providing the name of the desired object class as an argument to the `(make-instance class)` function. The object is instantiated, and a reference is handed back in form of a `avLink<>` value.

A set of accessor functions allows access to the field container and the field interfaces of AVOCADO objects. Figure 4.16 shows an example script which instantiates an `avFile` nodes and connects it to an `avTrackerSensor`.

An `avFile` nodes and an `avTrackerSensor` are instantiated. The tracker is configured to read it's input from a tracking device called *head-tracker*. The file node loads an OpenInventor file from the specified URL. By making the file node a child of the *scene-root* the associated geometry gets rendered. After connecting the file node to the tracker sensor, the loaded geometry will follow the tracker movement.

Because ELK Scheme is an interpreted language, every AVOCADO application provides a command-line interface, where Scheme commands can be given at run-time. All AVOCADO objects which have been defined from a scheme script can be accessed and manipulated while the application is running.

The AVOCADO scripting interface suggests a two tracked approach to application development. Complex and performance critical functionality is implemented in C++ by subclassing and extending existing AVOCADO classes. The application itself is then just a collection of Scheme scripts which instantiate the desired AVOCADO objects, set their field values and define relationships between them. The scripts can be written and tested

while the application is running. This leads to very short turnaround times in the development cycle and provides a very powerful environment for rapid application prototyping, which was proposed in the requirements.

4.1.10 A component interface for framework extension

A framework that offers a component interface encourages developers to design a complex application as a collection of objects that perform simple, well-defined tasks. It allows the framework to be kept small while additional functionality can be added to an application in a controlled and convenient way. The probably most (in)famous example is Microsoft's COM[63] (Component Object Model). Component interfaces enforce strict encapsulation of functionality, encourage clean interface design and foster reuse, which is generally a good thing. A good component interface does not just expose internal structures, but exposes functionality on a higher level of abstraction that describes *what* can be done, not *how* it is accomplished.

A component is a collection of code that exposes a set of interfaces that define object properties, methods and events. A few requirements must be fulfilled by a component interface to make it useful:

- A component can be attached to the application at run-time. It need not be available at compile-time.
- Interfaces are unique and immutable within the realm of the component model.
- Objects can be instantiated from the component at will. They adhere to one or more of the components interfaces.
- Each object can be queried for its set of supported interfaces.
- Each interface can be queried for the set of properties, methods and events it defines (reflectivity).

Based on the field container class Avocado defines a simple component interface that fulfills those requirements. An object interface description, with respect to the component model, consists of a unique name and an unordered set of fields. Additional methods or events cannot be exposed. Each object is of type `avFieldContainer` and can only be manipulated through the `avFieldContainer` interface and by modifying the objects fields. Because of the fully reflective properties of the avFieldContainer interface, all component objects can be queried for their type name and for the names, types and values of the contained fields.

A component is represented by a dynamic shared object (DSO) that can be mapped into the address space of an application at runtime and contains the components code and data segments. By convention, each component

provides exactly one type of object that can by created through a factory function. The component DSO file carries the same name as the type of the components object.

The global function

```
avLink<avObject> makeInstanceByName(const string\& type);
```

is the key to generic object instantiation from components. First, the type registry is check whether a factory object for a type of this name exists. If yes, a new object of the desired type is created and returned. If not, a set of predefined locations on disk is searched for a component DSO file with the same name as the type of the object to be constructed. If such a DSO is found, it is mapped into the address space of the application and the initialization function is resolved and called. During initialization of the component a factory object for the component type is registered with the type registry. This factory object is now used to create an object of the desired type and the object is returned to the caller.

Because of the full reflectivity of the objects created through the component interface, they have all properties of normal Avocado framework objects with respect to such important features as streaming, persistence and script-ability. This is especially important with respect to the intended development style when using the Avocado framework. Common and CPU intensive functionality can be encapsulated in reusable, compiled components and is accessible through the associated objects and interfaces, while application specific initialization and association of components is implemented using the interpreted scripting language.

4.1.11 An interface for state object persistence

The streaming interfaces for fields and field containers have already been discussed in sections 4.1.1 and 4.1.2. The ability to serialize fields and field containers into a stream of bytes or characters is now used to implement persistence not only for single objects, but also for references between objects and thus the entire scene graph.

Writing the state of a single field container object to file is straight forward to implement. Figure 4.17 shows a code segment that writes a set of fields to file. It is the core of the implementation of the stream-out operator on `avFieldContainer`. It consists of a loop over all fields of the field container. Each field is written as the field name followed by the string representation of the field value followed by a newline character.

If the field has a field connection from some other field, this field connection has to be made persistent as well. This shows that it is not sufficient to write out single field container objects, the relationships between objects need to made persistent. There are two different kinds of relationships between objects that need to be regarded:

```
void
avFieldContainer::write(avOutputStream& os)
{
  avFieldPtrVec& fields = getFields();
  avFieldPtrVec::iterator i;
  for (i=fields.begin(); i!=fields.end(); i++) {
    os << (*i)->getName() << ' ' << (*i)->write(os) << endl;
    (*i)->writeConnection(os);
  }
}
```

Fig. 4.17: The core of the stream-out operator on `avFieldContainer`

- Field connections
- Object references via fields of type `avLink<>` and `vector<avLink> >`

Field connections and object references define a general, non-directed, probably cyclic graph that needs to be serialized. The solution is to find a traverser that guarantees traversal of the entire graph in a limited amount of time and to prevent objects from being written to a stream more than once.

The Avocado traversal algorithm exploits the fact that objects are organized in a scene graph that is guaranteed to be directed and acyclic. A top-down, depth-first traversal of the scene graph along the edges defined by the parent-child relationship between objects will eventually pass all objects in the scene-graph at least once.

During the traversal an object that is passed for the first time is assigned a traversal sequence number and its state is written to the stream. If the object contains references to other objects, either through field connections or object references, it is checked whether the referenced object has already been written to the stream or has been encountered for the first time.

If it has already been streamed the reference is represented by writing the sequence number of the referenced object to the stream. Also the traversal does not continue with the referenced object.

If the referenced object is encountered for the first time, a new sequence number is generated and together with the newly encountered object is written to the stream. The traversal then continues with the referenced object.

This way, all objects get written to the stream exactly once and all references between objects are properly represented in the stream. Figure 4.19 shows the serialization of the simple scene graph consisting of three objects that can be seen in figure 4.18.

Note how the sequence numbers are appended to the object classes separated by a colon. A reference to an already written object is represented by

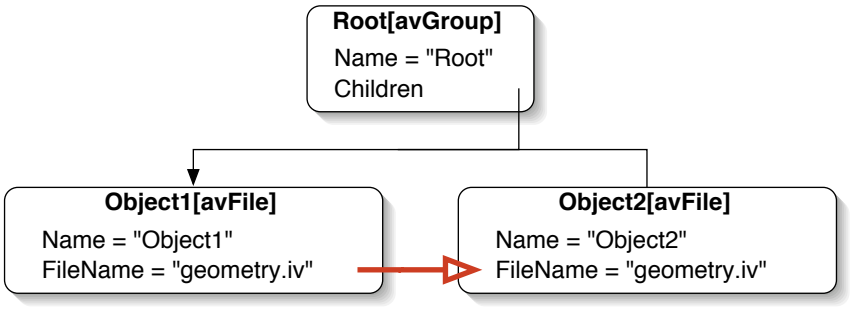

Fig. 4.18: The serialization of this simple scene graph is shown in figure 4.19

```
avGroup:1 {
  Name "RootNode"
  Children [
    avFile:2 {
      Name "Object1"
      FileName "geometry.iv"
      = avFile:3 {
          Name "Object2"
          FileName "geometry.iv"
        }
    }
    #3
  ]
}
```

Fig. 4.19: This is the result of the serialization of the simple scene graph shown in figure 4.18.

a hash mark followed by the sequence number of the object. A field connection is represented by an equal sign following a field value. The "Object2" is referenced twice in this example and is written to the stream only once. The second reference is represented using the sequence number.

The resulting file format is not intended as a public data exchange format, but is meant to be a framework internal format. Therefore it is not explicitly documented and the modification of the resulting files with a text editor or other framework external tools is not recommended.

4.2 Display device abstraction

A major requirement for the Avocado framework is to enable the use of a huge variety of display devices. Possible devices can range from a simple graphics window on a workstation monitor to sophisticated, head-tracked stereo, multi-screen devices like the Cyberstage and the Responsive Work-

bench. To support a specific device it may be not sufficient to provide a suitable rendered image of the scene. Head-tracked devices like the BOOM, for example, require that the tracking data from the device be incorporated into the imaging model. Further, some of the more interesting devices additionally require the provision of 8 or more different images of the scene to compose visual output for one rendering frame.

A display device abstraction must be mighty enough to model all those different kinds of display devices, while at the same providing maximum performance through the efficient use of multi-pipe and multi-processor hardware. The concepts used in the abstraction should be few in number and easy to grasp for the application developer.

Devices can belong to one or more of the following categories:

Monoscopic Display: A monoscopic, "one eyed" view of the scene is rendered.

Stereoscopic Display: A stereoscopic "two eyed" view of the scene is rendered. Actually, two monoscopic images are rendered from slightly different view points, one for the left eye, one for the right eye. The device supports some means of separating the two images when presented to the viewers left and right eye (e.g. shutter glasses or polarizing glasses).

Multiple Screens: The display consists of more than one physical display area. Usually the physical screens are pieced together to provide the impression of one large, not necessary flat, screen. Separate images are rendered for each physical display screen.

Static Symmetric Frustum: The viewing frustum is static and the position of the projection plane relative to the viewer is constant. The frustum itself is symmetric in order to simulate the behavior of a real life camera optic.

Static Asymmetric Frustum: The viewing frustum is static and the position of the projection plane relative to the viewer is constant. The frustum is asymmetric in order to accommodate for the position of the physical viewer relative to the physical screen.

Dynamic Frustum: The frustum is recalculated for every frame in order to accommodate for the position of the physical viewer relative to physical screen, even if that viewer is moving relative to the screen. These devices require some means of measuring the position of the physical viewer relative to the physical screen.

To illustrate the enormous flexibility required from a general purpose display device abstraction, a few of the more common existing display devices have been categorized in Table 4.2.

	Monoscopic Display	Stereoscopic Display	Multiple Screens	Static Symmetric Frustum	Static Asymmetric Frustum	Dynamic Frustum
Monitor	•			•		
Stereo Monitor		•			•	
Wall Projection	•			•		
CONE		•	•		•	
Stereo Projection		•			•	
CAVE, Cyberstage		•	•			•
Workbench		•	•			•
Head-Mounted Display		•	•		•	
A+C Explorer Panel	•					•

Tab. 4.2: Categorization of common display devices. The CONE is a curved, 4-screen surround projection with edge blending hardware. The Art+Com Explorer Panel[67] is a screen-tracked LCD flat-panel device.

The Avocado approach to the problem is based on a small number of components that the application programmer can link together to customize his application to any number of different display devices or combinations thereof. The following section will describe each of the components in detail.

4.2.1 Basic elements of the abstraction

A class diagram showing the relationships between the display component classes is given in Figure 4.20.

The `avEye` class

The `avEye` class is derived from the standard `avDCS` class. It is meant to be part of the scene graph, and marks a possible eye point in the graph. Additionally, `avEye` has a the `NearFar` field which is used to specify the positions of the near and far clipping planes to be used for renderings from that eye point. The position and orientation of the `avEye` node in global world coordinates are used to determine the camera position and orientation for renderings from that eye point. As a normal member of the scene graph, an `avEye` node will inherit all transformations specified on its path to the scene root.

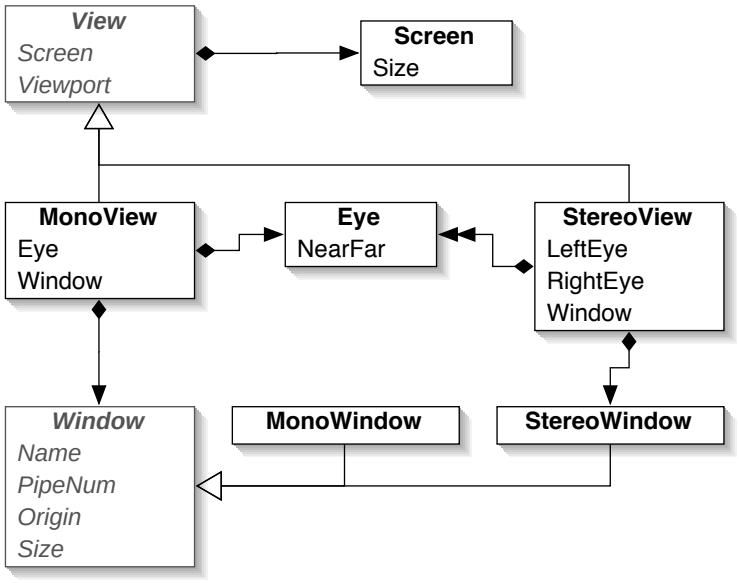

Fig. 4.20: Display class diagram. `avEye` and `avScreen` instances are part of the scene graph. The others are part of the application environment.

The `avScreen` class

The `avScreen` class is derived from the standard `avDCS` class. It is meant to be part of the scene graph, and marks a possible (virtual) projection screen in the graph. The `Size` field determines the horizontal and vertical extend of the virtual screen on the projection plane The virtual screen is defied to be axis aligned within the projection plane coordinate system. By convention, the projection plane is defined as the XZ-coordinate plane in the local coordinate system of the `avScreen` node. The position and orientation of the `avScreen` node in global world coordinates are used to determine the position and orientation of a projection plane and the extend of the virtual screen on that plane. As a normal member of the scene graph, an `avScreen` node will inherit all transformations specified on its path to the scene root.

The `avView` class

The `avView` class is an abstract class and is directly derived from `avFieldContainer` and is therefore not part of the scene graph. Its concrete subclasses can be seen as part of the application environment surrounding the scene graph.

The calculation of the viewing frustum is the main responsibility of the `avView` class. By using the function `avNode::getWorldTransform()`, `avView` determines the transformation from the local coordinate system of

the `avEye` node into the world coordinate system. The function traverses the scene graph upward to the root node, and accumulates the transformation matrices it finds in `avDCS` or `avSCS` nodes on the way up. The same is done for the `avScreen` node. Let the `avEye` transform be the 4×4 matrix E, and S the one for the `avScreen` node. Further, let w be the horizontal width of the screen, and h the vertical height, both expressed in the local coordinate system of the screen and obtained from the `Size` field of the `avScreen` node. The frustum is specified in terms of the distances n and f of the near and far clipping planes from the eye point, and the distances of the intersection lines of the four defining, axis aligned side planes of the frustum with the near clipping plane from the origin (Se figure 4.21). Let l be this distance for the left side plane, and r, b, t for the right, bottom and top planes respectively. This frustum definition is in *eye space*, where the eye point is at the origin, and the viewing direction is the positive Y-axis (by convention of the underlying graphics system). First, the projection of the eye point onto the screens XZ-plane, $\mathbf{e}^{S_{xz}}$, is determined as:

$$\mathbf{e}^{S_{xz}} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \mathbf{ES}^{-1}$$

The distances of the near and far clipping planes in eye space , n^E and f^E are specified in the `NearFar` field of the `avEye` node. They are used as absolute values and are not transformed. Now the frustum definition can be completed as:

$$\begin{aligned} n &= n^E \\ f &= f^E \\ l &= \frac{(-\frac{w}{2} - \mathbf{e}_x^{S_{xz}})n}{\mathbf{e}_y^{S_{xz}}} \\ r &= \frac{(\frac{w}{2} - \mathbf{e}_x^{S_{xz}})n}{\mathbf{e}_y^{S_{xz}}} \\ b &= \frac{(-\frac{h}{2} - \mathbf{e}_z^{S_{xz}})n}{\mathbf{e}_y^{S_{xz}}} \\ t &= \frac{(\frac{h}{2} - \mathbf{e}_z^{S_{xz}})n}{\mathbf{e}_y^{S_{xz}}} \end{aligned}$$

Although the frustum definition is done in eye space, by first transforming the eye point into screen space and the projecting it onto the screen plane, and then transforming it back into two dimensional eye space, all

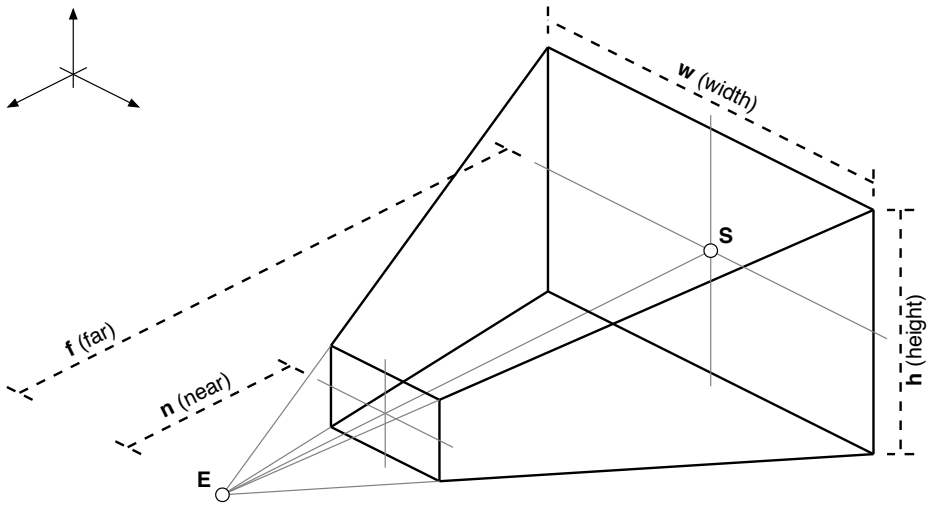

Fig. 4.21: The relative position and orientation of the screen and the eye coordinate system E and the screen coordinate system S defines the view frustum that is used for rendering. (For visual clarity, the distance between E and S is exactly f, which is not mandatory and not normally the case)

rotational components are conveniently eliminated from the eye point specification.

Beside the calculation of the viewing frustum, the avView class is responsible for the mapping from the virtual screen to a graphics output window. The positioning of the projected image within the graphics window is specified via the `Viewport` field. Its four values are used to position the image relative to the four window borders.

The `avMonoView` class

The `avMonoView` class is a concrete subclass of the `avView` class. Its `Eye` and `Screen` fields are of type `avLink<avEye>` and `avLink<avScreen>` respectively, and are used to reference the particular `avEye` and `avScreen` nodes that are to be used for view frustum calculation. The `Window` field references a particular monoscopic graphics window to which the output should be directed.

The `avStereoView` class

The `avStereoView` class is a concrete subclass of the `avView` class. Its `LeftEye`, `RightEye` and `Screen` fields are of type `avLink<avEye>` and `avLink<avScreen>` respectively, and are used to reference the particular

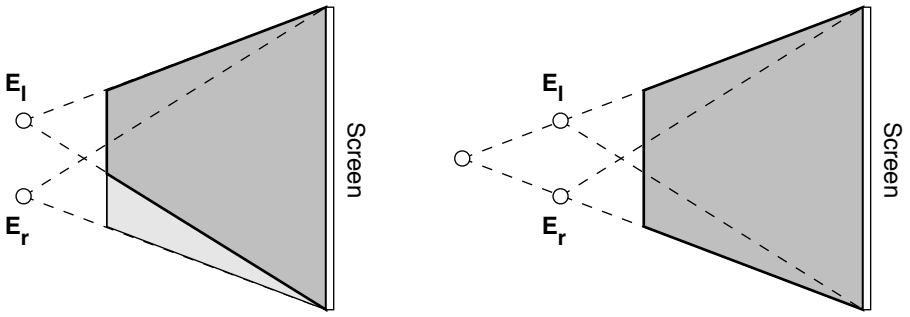

Fig. 4.22: A combined view frustum for left and right eye is used to optimize cull performance. Instead of two cull passes only one is needed at the expense of a slightly larger cull frustum.

`avEye` and `avScreen` nodes that are to be used for the calculation of *two* related viewing frustra for stereo rendering. The `Window` field references a particular stereoscopic graphics window to which the output should be directed.

The only reason for the existence of dedicated mono and stereo versions of the `avView` class is optimization of rendering performance. As mentioned in Section 2.1.1, Performer makes aggressive use of view frustum culling in the CULL rendering stage to reduce the amount of geometry sent to the DRAW stage for rendering. Typically, the viewing frustra for the left and right eye view of a stereo image pair differ only by a small percentage, producing almost identical results in the CULL stage (see figure 4.22).

The avStereoView class exploits this, by culling only once for each stereo image pair. The cull is performed against the union of the view frustra for the left and right eye. So, instead of using two CULL stages and two DRAW stages for the generation of a stereo image pair, `avStereoView` only uses *one* CULL stage and two DRAW stages to produce the same result. This can reduce processor utilization by up to 25%.

The `avWindow` class

The `avWindow` class is an abstract class and is directly derived from `avFieldContainer` and is therefore not part of the scene graph. Its concrete subclasses can be seen as part of the application environment surrounding the scene graph. The main task of an `avWindow` is to identify the graphics window that an `avView` should render to. The `PipeNum` field specifies the id of the graphics hardware subsystem, the window will be opened on. The `Origin` and `Size` fields specify the relative position and the extend of the window. The `Name` field specifies a string that will be shown in the windows title bar.

The `avMonoWindow` class

The `avMonoWindow` class is a concrete subclass of the `avWindow` class. It will open a window of the specified size and origin on the specified hardware graphics pipe. The window will be opened in mono mode, providing only a front and a back buffer for double buffered rendering.

The `avStereoWindow` class

The `avStereoWindow` class is an abstract subclass of the `avWindow` class. It is used as a base class for all stereo window classes. The `Window` field on the `avStereoView` class can accept references to all concrete subclasses of the `avStereoWindow` class.

The `avNewStyleSW` class

The `avNewStyleSW` class is a concrete subclass of the `avStereoWindow` class. It will open a window of the specified size and origin on the specified hardware graphics pipe. The window will be opened in quad buffer stereo (also called "stereo in a window") mode, providing a left and a right version of the normal front and a back buffer for double buffered stereo rendering. Quad buffer stereo supports field sequential stereo output devices and is normally used in conjunction with synchronized LCD shutter glasses for image separation.

The `avOldStyleSW` class

The `avOldStyleSW` class is a concrete subclass of the `avStereoWindow` class. It will open a window of the specified size and origin on the specified hardware graphics pipe. The window will be opened in mono mode, providing only a front and a back buffer for double buffered rendering. The assumption is, that the entire graphics subsystem is in "old-style stereo mode". This has to be assured by the user running the AVOCADO application. In "old-style stereo mode" the upper half of the entire screen is used to display the video image for the left eye, while the lower half will display that for the right eye. The video scan-conversion logic will produce a field sequential stereo signal from the two images. A `avOldStyleSW` normally opens a full screen window and directs the image for the left eye to the upper half of that window, and the image for the right eye to the lower half.

4.2.2 Configuration examples for selected devices

This section gives six common examples to show just how the display components enable the AVOCADO application developer to support a wide variety of different display output devices.

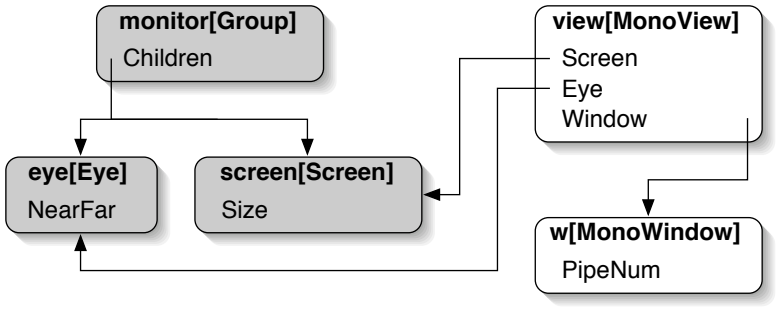

Fig. 4.23: A configuration for a simple monitor device.

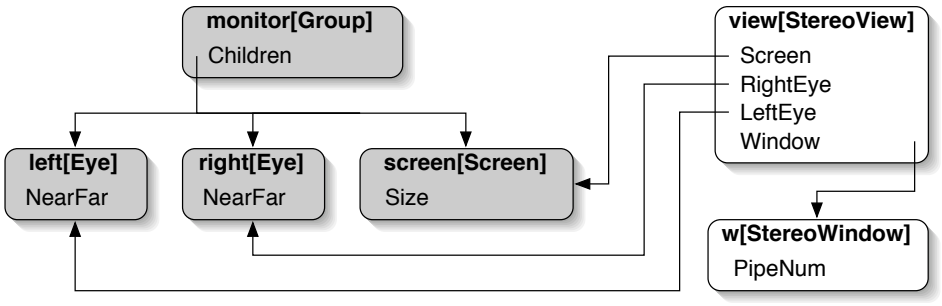

Fig. 4.24: A configuration for a stereo capable monitor device.

Monitor

A single graphics window on a workstation monitor is the most simple device that is supported by Avocado (see figure 4.23). It uses an `avMonoView` object in conjunction with an `avMonoWindow` to open a window on the specified graphics display. One `avEye` node and one `avScreen` node are grouped together using an `avGroup` node. Their relative positions and orientations completely specify the viewing frustum for rendering. They are both referenced by the `avMonoView` object. The `avGroup` node can be placed anywhere in the scene graph to determine the eye position and orientation.

Should the application require the eye point to move (which is likely to happen in most applications), it just needs to modify a suitable transformation node above the group node to alter its absolute position and orientation in world coordinate space. The relative position of the avEye and avScreen node, and thus the viewing frustum, will remain constant. The viewer will be moved through the scene without modification of the camera parameters.

Stereo monitor

The configuration for a stereo capable monitor (see figure 4.24) is very similar to the configuration for the simple monitor introduced above. Instead

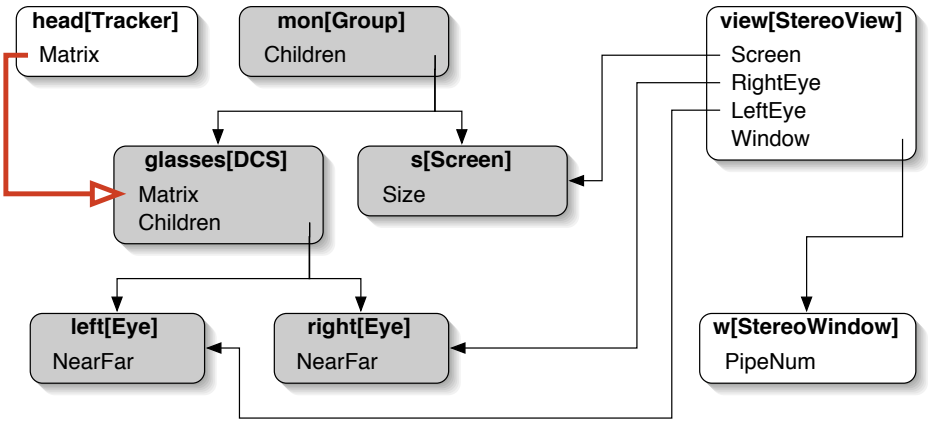

Fig. 4.25: A configuration for a head-tracked stereo capable monitor device.

of the mono versions of the view and window objects, the stereo versions `avStereoView` and `avStereoWindow` are used. Stereo images need two eye points, one for the left and one for the right eye view. The use of two `avEye` nodes reflects this. Both eye points project their images onto the same virtual screen, so only one `avScreen` node is used. Eye nodes and screen nodes are referenced by the `avStereoView` object. As with the monitor configuration, the group node can be added to the scene graph at any position. The absolute world space position of the `avEye` nodes specifies the eye points for left and right eye, while the relative positions of eye nodes and the screen node determine the two frustra.

Head-tracked stereo monitor

With a slight modification the stereo monitor configuration can be altered to support head-tracked stereo (see figure 4.25). An `avDCS` node is inserted above the two `avEye` nodes. Its `Matrix` field is connected from the `Matrix` field of a tracker sensor (see section 4.1.7). The tracker sensor will deliver data from a real world sensor that is attached to the users head (or, most probably, to the stereo glasses she is wearing). If the user moves her head in front of the monitor, the two `avEye` nodes below the `avDCS` nodes automatically follow that movement. As in the real world, where the monitor stays fixed, the `avScreen` node is not moved.

Thus, the two frustra used for image rendering change along with the the eye point movement, providing a high-quality stereo image of the objects in the scene. Because the frustra are recalculated for each new position of the users head, the viewed objects appear to be suspended in space, the user can look at them from different angles just by moving his head. Still, the whole setup can be moved freely through the scene by modifying

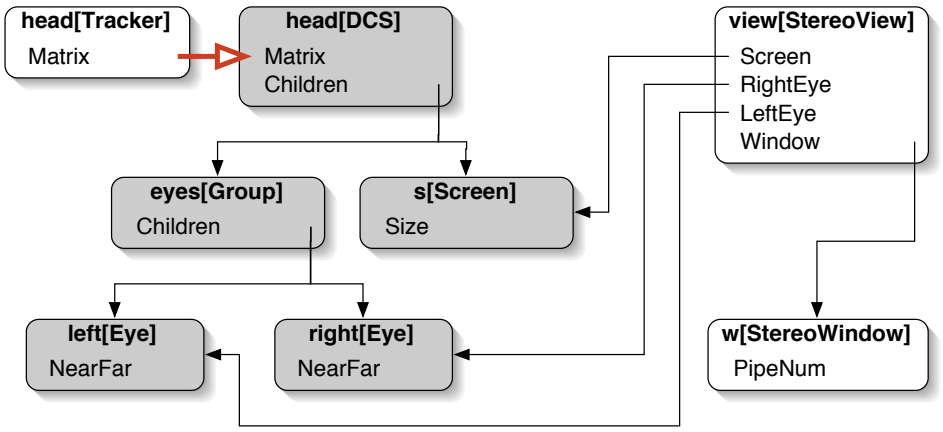

Fig. 4.26: A configuration for a head-mounted display with 100% overlap.

any transformation node above the `avGroup` node. The user will have the impression, his monitor where a window into the scene that he can virtually move around with.

Head-mounted display with 100% overlap

The configuration for a head-mounted display with 100% overlap (see figure 4.27) is surprisingly similar to the head-tracked stereo monitor configuration. The position of the `avGroup` and `avDCS` nodes in the scene graph have been swapped. Thus, not only the eye points, but also the virtual screen moves along with the head movement of the user. This reflects the physical configuration of a head-mounted display, where the physical screens are attached to the users head, and the entire assembly is tracked. There is still only one `avScreen` node, so that this configuration only works for head-mounted displays with 100% overlap.

Head-mounted display

The previous configuration examples used only one `avEye` object, and thus only one `avView` object. This was possible because, even in the stereo configurations, all the eye points used would project their images onto the same virtual screen. For head-mounted displays with less than 100% (see figure 4.27) overlap this is no longer true. A second `avScreen` node is needed because the two stereo images created here do only partially overlap. This is a common technique, used to increase the field-of-view. As a result, stereo vision is only available in the overlapping area of the two views, and the two viewings frustra are not only different with respect to the eye positions, but also with respect to the screen position and orientation.

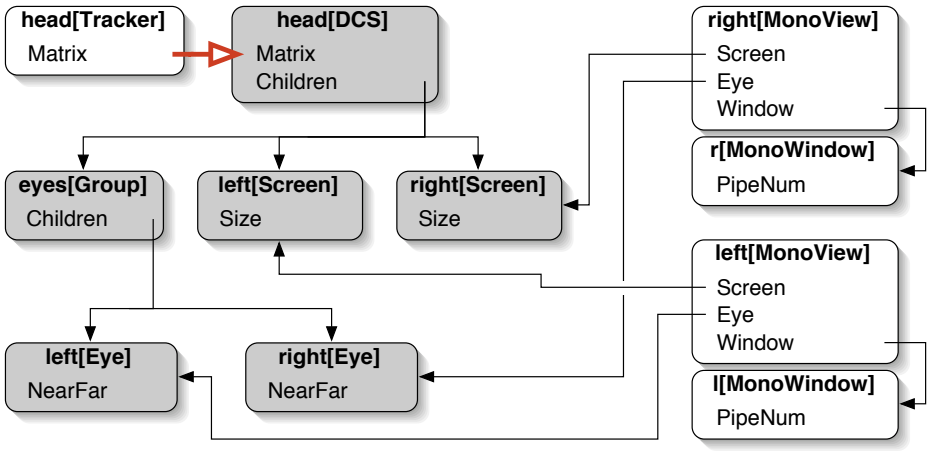

Fig. 4.27: A configuration for a head-mounted stereo display with less than 100% overlap.

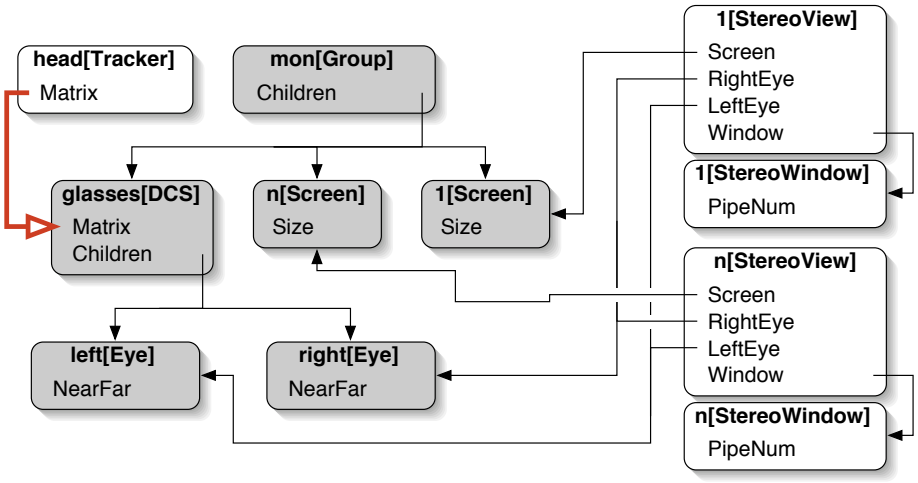

Fig. 4.28: A configuration for a two-sided head-tracked workbench device. The same configuration with N screens and stereo views would be used for a N-sided CAVE.

Instead of one `avStereoView` object, two `avMonoView` objects with associated `avMonoWindow` objects are used. Each view object references one `avEye`, `avScreen` pair to independently produce one of the two stereo images.

Responsive Workbench, Cyberstage and CAVE

The top of the crop are clearly large scale, immersive displays like the Cyberstage or the Responsive Workbench. Their similarity to head-mounted displays, with respect to the goal of user immersion, shows in the similarity of the to display device configurations (see figure 4.28). The basic setup is like that for the head-mounted display with less than 100% overlap. For each physical screen there is one `avScreen` object connect to one dedicated `avView` object. Because CAVEs use shutter-glass stereo, an `avStereoView` object with associated `avStereoWindow` is used. The `avGroup` node and the `avDCS` node are again reversed such that the tracker only controls the eye positions, not the virtual screens. The two `avEye` nodes are shared by all `avStereoView` objects, reflecting the assumption that there is only one head-tracked viewer in the CAVE at any one time.

Again, because the tracker controls the eye points relative to the virtual screen, the whole setup can be freely moved around in the virtual world by modifying any suitable transformation above the *avGroup* node. The user can look at the CAVE as a vehicle that she is driving (or flying) around the virtual world.

Summary

AVOCADOs display device abstraction relies on a few components that can be easily combined to adapt to any conceivable display device configuration. Because of its modularity it can not only adapt to the device configuration, but also efficiently utilize multi-pipe graphics hardware on multi-processor machines to achieve the best possible performance for any configuration. By separating the frustum definition into two independent components, the eye point and the virtual screen, devices with head-tracked viewers can be conveniently modeled. Moreover, by integrating the eye point and virtual screen definition into the scene graph, even complex camera control mechanisms can easily be implemented. By using special optimizations for stereo views, the rendering performance can be increased by up to 25%.

4.3 Tool-based interaction support

Direct user interaction with objects in a VE on devices like the Responsive Workbench is a natural part of the interface metaphor of such devices. This section describes a framework that provides the necessary services for the application level implementation of WIMP-style user interfaces, which are well known from two-dimensional WYSIWYG[1] applications.

Although being intended for the somewhat special case of stylus based interaction on the Responsive Workbench, the framework should provide

[1] WYSIWYG: What You See Is What You Get

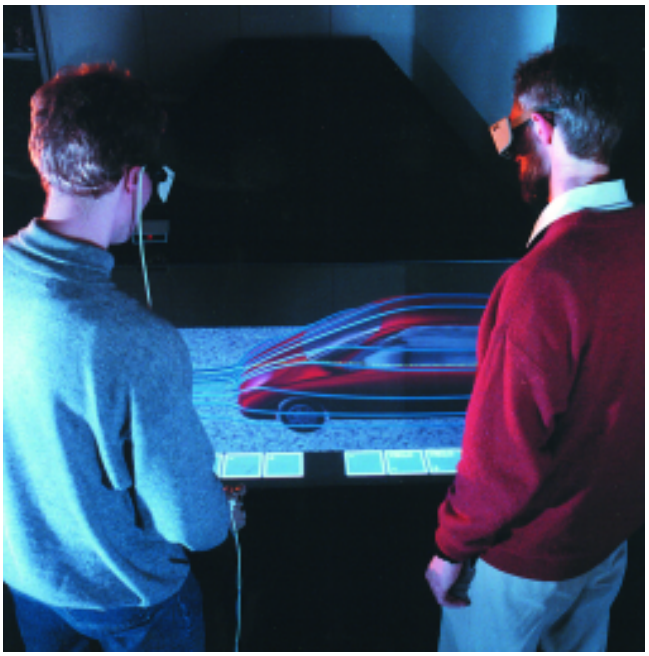

Fig. 4.29: Two users in front of a Responsive Workbench, working on a flow-field simulation of a car.

maximum overall flexibility to the application developer in the definition of object/pointer interaction patterns. Especially the question of where to specify the action should not be answered by the framework, but should be resolved by the application. Also, it should be possible to mix both styles within one application.

Also the interaction framework must be extensible such that new applications can implement specific interaction patterns without modification of the framework. This can be achieved by representing the necessary abstractions as AVOCADO classes that can be extended using the standard AVOCADO component interface (see also section 4.1.10).

4.3.1 An interaction metaphor for the Responsive Workbench

The Responsive Workbench[26] is a horizontal back-projection tabletop display. The height of the projection display corresponds to the height of a real-world workbench or table. The user stands upright in front of the Responsive Workbench and looks at the stereo projection through a pair of head-tracked goggles. Left and right image separation is achieved by either using image sequential projection and synchronized LCD-shutter glasses, or by using two polarizing image projectors and matching polarized glasses.

A commonly used input device is a space tracker probe in form of a stylus that the user can grab with her hand and move around above the

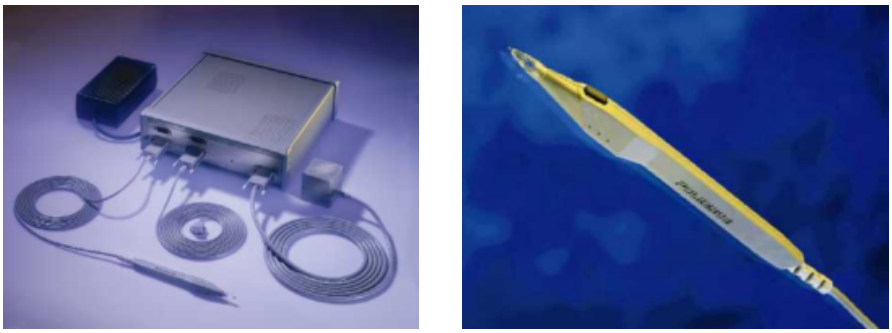

Fig. 4.30: A Polhemus Fastrack with the Stylus input device.

Workbench. The stylus is equipped with a push button. The VE software can use the stylus position and orientation together with the status of the push button to deduct the possible intention of the user with respect to the context of the VE.

The virtual environment that is presented to the users often consists of complex objects that appear to be "lying" on the tabletop display. The stylus will then be used to directly manipulate those objects, as for example in a assembly testing application. The metaphor that is used to facilitate the interaction between the stylus and an object in three dimensions is directly related to the well known case of interaction between a mouse pointer and two dimensional objects in a normal WYSIWYG application.

The stylus corresponds to the mouse. The user holds the stylus with his dominant hand and moves it around above the workbench display. A visual marker object is used inside the VE to visualize the corresponding position of the real world stylus in the virtual environment. This marker is the equivalent to the mouse pointer in a 2d user interface. Whenever the marker intersects with an object in the virtual environment, an interaction is possible.

The marker resembles an infinitely long light saber[2] to overcome potential reach problems. Objects can appear in positions on the workbench display that the user can not reach with his hand. This is, for example, always the case if the object appears *under* the workbench tabletop. The one dimensional extend of the marker still allows precise interaction with objects out of reach in a laser beam pointer like fashion.

Using the marker to identify an object for a possible interaction, the button on the stylus is used to initiate an interaction. This is classic *point and click*, just like in 2d user interfaces using a mouse and an on-screen pointer.

[2] A popular general purpose tool in the Star Wars universe.

Point, click and drag

A user can perform three different basic actions on an object using the stylus.

Point: The pointer intersects one or more objects. No interaction takes place. Instant visual feedback through the marker allows for easy selection of objects for interaction.

Click: The user clicks the button on the stylus. An interaction between the pointed to object and the pointer is initiated. It can potentially last until the stylus button is released.

Click and Drag: The user clicks the button on the stylus. An interaction between the pointed to object and the pointer is initiated. During the button down period the spatial position of the stylus is available as one parameter of the interaction.

Interaction between a pointed to object and the pointer can occur when the stylus button is clicked. During pure pointing, when the stylus button is not pressed, no interaction between object and stylus is supported. A more general approach would allow interactions between the pointer and an object even if the button is not pressed. For performance reasons it was decided to not allow this case, as a simple implementation of a continuous detection of intersections between the stylus and all object in the scene turned out to significantly impact overall system performance on small, one-processor graphics workstations.

Action specification

After an object has been selected for interaction by pointing and clicking, it needs to be specified what kind of action should be performed. As the object and the pointer are the only two entities participating in the interaction, either the object or the pointer must define the details of the interaction. A closer look at common interaction patterns found in WIMP applications reveals good reasons for both possibilities.

Object-based Action Specification: The action is associated with the object. The pointer is generic and just triggers the action that is defined on the object. Different objects can define different actions. A good example is a standard push button in a WIMP application, where the mouse pointer behaves the same for all buttons, while each button typically performs a specific action. The definition of a specific action is solely associated with each button.

Pointer-based Action Specification: The action is associated with the pointer. All objects behave the same with respect to the pointer. As an example for this case consider a 2D drawing application. Most drawing

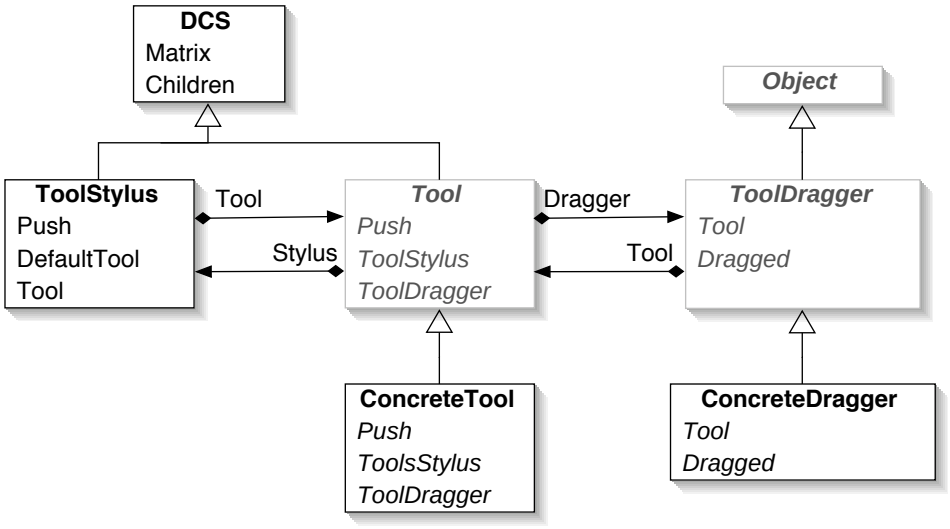

Fig. 4.31: The abstraction used to represent interaction patterns between objects and the workbench stylus consist of three classes.

applications provide a variety of user selectable pointers that perform specific operations on objects. If the user, for example, selects the "move pointer", clicking and dragging any object will move the object around on the canvas, regardless of the concrete type of the object. The definition of the action is solely associated with the pointer.

4.3.2 Implementation of the interaction abstraction

The abstraction used to represent interaction patterns between objects and the workbench stylus consist of three classes (see figure 4.31), the `ToolStylus`, the `Tool` and the `ToolDragger`. The `ToolStylus` is a concrete class that describes a stylus, while `Tool` and `ToolDragger` are abstract classes that must be specialized to perform application specific tasks.

The `ToolStylus` class

The `ToolStylus` class represents the physical stylus in the virtual environment. It is derived from `DCS` and thus inherits a `Matrix` field. By connecting positional input from a tracker to the `Matrix` field, the `ToolStylus` moves along with the real-world stylus. The `ToolStylus` also monitors the status of the stylus button. If desired, a 3D representation for the `ToolStylus` can be provided by adding some geometry as child nodes.

The main purpose of the `ToolStylus` is to relate the user controlled stylus movements to tools that facilitate manipulation of objects in the virtual environment. Objects of class `Tool` can be connected to the `Tool` field. On

connection the `Tool` node is re-parented to the `ToolStylus` and for the duration of the connection moves along with the stylus. The stylus button status is also forwarded to the tool and is available through the `Push` field. Tools can be exchanged by the application at any time, making the `ToolStylus` a generic handling device for the various `Tool` objects an application offers to the user.

The `Tool` class

The `Tool` class is the abstract base class for all tools. Each concrete subclass represents a tool that can be attached to a `ToolStylus` and implements a specific method of interaction with objects in the virtual environment. If a tool requires a 3D representation, the suitable geometry can be added as child nodes.

A tool does not manipulate an object directly, but does so by way of a *ToolDragger* object.

The `ToolDragger` class

The `ToolDragger` class is the abstract base class for all draggers. Draggers represent the interface between the objects in the scene graph and the tools that want to manipulate the scene graph and its objects. Together with the tool currently attached to the tool stylus, the `ToolDragger` objects define whether an interaction is possible at all, and if so, what the exact effect of the interaction on the relevant objects will be.

Every object that is part of the scene graph is derived from the `avNode` class and thus inherits the `Dragger` field. It is a multi-field that can contain any number of references to `ToolDragger` objects.

An example for a simple `ToolDragger` is the `MatrixDragger`. The Interface that a `MatrixDragger` exposes to a tool consists of a single matrix field.

```
class MatrixDragger : public ToolDragger{
public:
  MatrixDragger();
  avSFMatrix  Children;
};
```

A tool that interacts with this dragger modifies the `Matrix` field according to its own position. The `Matrix` field of the dragger, in turn, connects to the `Matrix` field of a `DCS` node in the scene graph. The dragger has two responsibilities: first, it marks an object for possible interaction with a certain class of tools, and second, it mediates between the interacting tool and object and possibly filters modification requests.

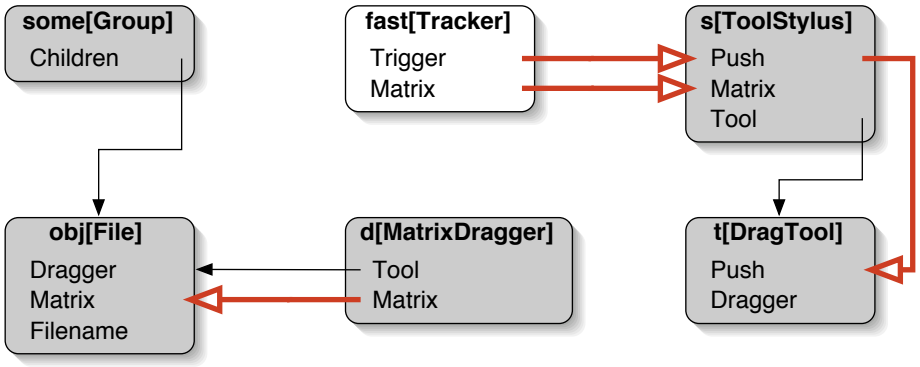

Fig. 4.32: A simple interaction setup in its initial state before the user pushes the stylus button.

A simple interaction example

The implementation is further documented along the lines of a simple interaction example. Figures 4.32, 4.33 and 4.32 show various stages of an interaction.

Figure 4.32 describes the initial state of the system, everything is set up but interaction is yet performed. The `MatrixDragger` *obj* is connected to an `avFile` node that is part of the scene graph. The file node references the dragger while its `Matrix` field is connected from the `Matrix` field of the dragger. A `Tracker` is used to feed the `Push` and `Matrix` fields of the `ToolStylus` *s* with the current button status, position and orientation of the real-world stylus. The tool stylus holds the `DragTool` *t* and forward the value of its `Push` field to the tool.

Now, the user points the stylus at the geometry that is represented by the `File` node *obj* and pushes and holds the stylus button (Figure 4.33). The tool stylus now performs a *pick* operation and determines whether the stylus points at an object that has a dragger connected that is compatible to the current tool. The process of picking and dragger matching is further detailed in sections 4.3.2 and 4.3.2. In this example the pointed to object is the `File` node *obj*, and the `MatrixDragger` *d* is compatible to the `DragTool` *t*. Now *d* and *t* are connected by referencing each other with their `Tool` and `Dragger` fields respectively. This connection between tool and dragger is held up as long as the user keeps the stylus button pushed.

After the initial setup, whenever the user moves the stylus, the `Matrix` field of the `Tracker` *fast* reflects this change and propagates it through the field connection to the tool stylus *s* (Figure 4.34). The tool stylus now calculates its new global position and makes it available to the connected tool *t*. The tool in turn calculates the Δ transformation from its old position to the new position and communicates that to the temporarily connected

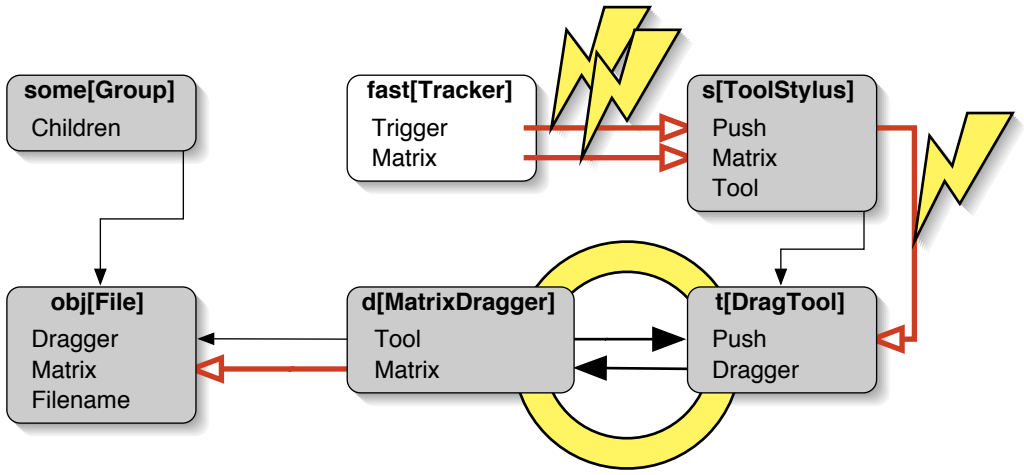

Fig. 4.33: The user has pushed the stylus button and after some searching and matching the relevant tool and dragger are temporarily connected.

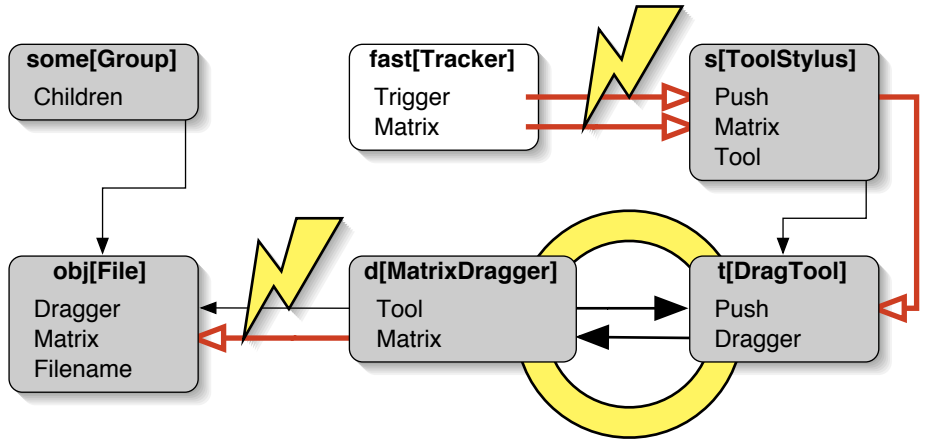

Fig. 4.34: The connection between the tool and the dragger is established, and as long as the user holds the stylus button, the stylus movements are applied to the *obj* node (The lightning symbolizes an active field connection).

dragger *d*. The dragger now applies this Δ transformation to the `Matrix` field of the `File` object *obj*. As a consequence, the geometry represented by the `File` node *obj* will move along with the stylus movement in the real-world, as long as the user pushes the stylus button.

When the user releases the stylus button the temporary connection between the dragger *d* and the tool *t* is removed and the geometry below the

`File` node *obj* remains at its last position.

Picking

Picking is initiated in the `ToolStylus` once the user has pushed the button on the stylus. The desired result of this stage is the object that intersects with the laser pointer beam of the tool stylus. This test is performed in two steps:

1. The pick segment is calculated in global coordinates and is described as a line segment that extends from the global stylus position into the direction of the laser beam pointer. First, the transformation from the local stylus coordinate system into global coordinates is calculated by upward traversal of the scene graph and accumulation of all transformations found on the way. Then the local stylus position and direction is transformed into global space and used to construct the pick segment.

2. In a top-down, depth-first traversal of the scene graph, the pick segment is tested against the object geometry. If an object geometry intersects with the pick segment, the object and the path to the object in the scene graph are returned. If more than one object intersects with the pick segment, only the one closest to the stylus is returned.

Dragger matching

Every object returned from picking is now checked for a dragger that matches the currently selected tool and can be interacted with. If no such dragger can be found, no interaction is possible. Because each object can associated with any number of draggers, the list of draggers is checked in the order of specification. The first dragger that matches the tool is used for the interaction. The remaining draggers are not considered.

If an object has no association with a matching dragger, the test is repeated with the parent object as specified by the pick path for the original object. This extension of the search to all parent objects allows the specification of single draggers for complex, compound objects by attaching the dragger to the top most group node of the object. Pick hits on subordinate geometric detail are then automatically propagated to the top most group node. By using the explicit pick path, it is guaranteed that the upward hit propagation works correctly in the presence of multiple instancing.

The decision whether a tool and a dragger can interact is made based on a two dimensional dispatch table that has an entry for each tool dragger combination that can interact (see figure 4.35). If an entry exists for a tested tool and dragger combination, the tool and the dragger can be connected to perform the actions that are necessary to conduct the interaction.

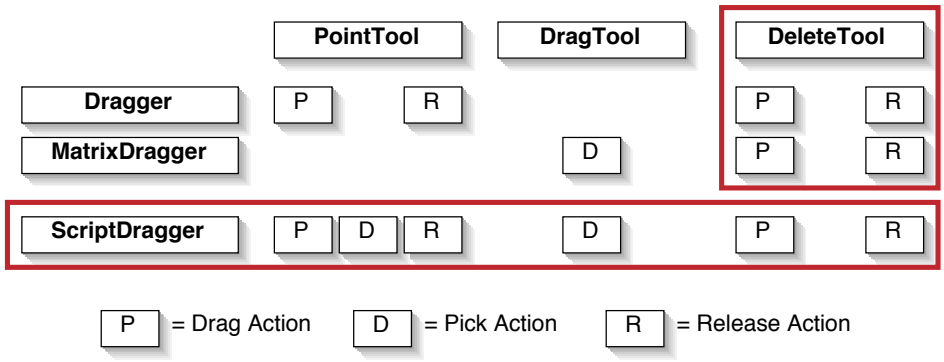

Fig. 4.35: The dispatch table has an entry for each tool dragger combination that can interact.

Specification of action functions

Each entry in the dispatch table specifies a set of three, possibly empty, action functions that are called during different stages of the interaction between a tool and a dragger. These action functions completely specify the effects of an interaction. References to the tool and the dragger are passed as parameters to each of the three functions. Possible action function are:

Pick Action: The pick action function is called immediately after a dragger and a tool are connected. It is called once at the beginning of an interaction.

Drag Action: The drag action function is called each time the position or orientation of the tool stylus changes during an interaction. It is called only if a tool and a dragger are currently connected.

Release Action: The release action function is called at the end of an interaction right before the connection between the participating tool and dragger is released.

The signature is the same for all three action functions.

```
typedef void (*ToolCallback)(Tool*, ToolDragger*);
```

The action functions implement the interaction entirely by only using the public interfaces of the tool and the dragger object. If state information has be passed between different invocations of the action functions, it can be associated with either the tool or the dragger object.

4.3.3 Extension through specialization

The action function entries into the dispatch table are defined during the definition of either a new tool or a new dragger class. Each tool or dragger

class has an initialization function that performs the entries into the dispatch table when the first tool or dragger of that class is instantiated.

Figure 4.35 shows a an example of a dispatch table and illustrates the extension mechanism. Assume the dispatch table initially consists of two rows and to columns holding the action functions for the `Dragger`, `MatrixDragger`, `PointTool` and `DragTool` classes. The `PointTool`, for example, defines a *Pick* and a *Release* function for objects of type `Dragger`. The *Pick* function highlights the geometry associated with the dragger, while the *Release* function removes the highlight. No *Drag* function is specified so stylus movements will have no effect.

Now a tool is added that interacts with both existing draggers. The new `DeleteTool` defines a *Pick* and a *Release* function and registers them with all existing draggers. If the *Release* function is called it checks whether the stylus still points at the same object it pointed to when the *Pick* function was called, and if so, it deletes the object from the scene graph. This way the `DeleteTool` allows deletion of any object that has a `Dragger` or a `MatrixDragger` connected. In the same way, new `Dragger` classes can be added to the system, like in this example, the `ScriptDragger`. Corresponding to a newly added dragger, a new tool specifies action functions for all previously existing tools.

Interaction summary

AVOCADO provides a simple and elegant framework for the application specific implementation of user interaction in virtual environments. Basic direct-manipulation interaction on devices like the Responsive Workbench can easily be modeled with a small number of classes and objects.

4.4 Summary

In this chapter details of the non-distributed aspects of the AVOCADO implementation have been presented. The implementation closely follows the general VE architecture that has been described in section 3.2, and as such fulfills all requirements formulated in section 3.1: A comprehensive object and event model, generic support for various display device, direct manipulation interaction support, a rapid prototyping development style, extensibility and maximum performance.

The main points of the AVOCADO implementation can be summarized as follows:

- AVOCADO is based on OpenGL Performer. The Performer facilities are responsible for data representation and high-performance rendering.
- The object model follows the field container paradigm and implements the necessary interfaces by subclassing from performer node classes.

- Nodes and sensors are the two main object classes. Nodes represent the objects used to build the scene graph, while sensors are AVOCADO s abstraction for external data input.
- The Elk Scheme implementation is used as the scripting language. A complete binding for all field container APIs is implemented. Scheme scripts are used for application scripting and event handler scripting.
- The component interface uses the field container interfaces to allow the creation of run-time loadable extension node classes through subclassing from native AVOCADO classes.
- The fine grained display device abstraction describes all contemporary display devices commonly used for virtual environment visualization.
- The interaction support framework provides the necessary abstractions and services for the implementation of direct manipulation interaction using tools and draggers.

Based on the presented AVOCADO implementation, the following chapter describes the realization of the distributed object and event model postulated in sections 3.2.1 and 3.2.2.

5. Avocado - implementation of the distribution architecture

In chapter 3 a general design for a distributed object and event model for DVE systems was presented. It provides a transparently shared scene graph and an integrated event distribution mechanism to the application developer. The relevant implementation details and APIs are presented in this chapter. Further, the implementation of the underlying transport layer based on the Ensemble/Maestro group communication system is described. The reliability and ordering guarantees of the transport layer allow the robust implementation of dynamic group membership changes and consistent state replication as described in section 3.2.6.

5.1 Distributed object model

The implementation of the distributed Avocado object model closely follows the design described in section 3.2.1.

5.1.1 State sharing through object replication

As rationalized in section 3.2.4 distribution support must be based on object replication. This way, copies of the distributed objects are present in each processes local address space and can be accessed for rendering without additional communication overhead, To provide a consistent view of the application state, local modifications of the objects are transmitted and applied to all other object copies.

Objects are represented as field containers that encapsulate object state in a set of fields (see section 4.1.2). The streaming interface of the field and field container classes allows a very elegant implementation of these distributed object semantics. Whenever a field value on a distributed object is locally changed, the new state of the object is serialized to a network buffer which is subsequently sent to all processes. Here the serialized state change information is reconstructed into the appropriate object copy, and thus distributed state consistency is reestablished.

As stated in section 4.1.1, the parent child relationship between objects in the scene graph is represented by a multi-field of reference values on the parent node. Because all field values, including `avLink<>` (see section

4.1.3) types, are serializable, the distribution scheme described above will not only distribute and synchronize singular objects, but also parent child relationships between them. As a result, it will automatically replicate the entire scene graph to all processes in a distribution application.

By default, all fields of a node are distributed. However, this is not always necessary or even desirable. For example, the values of `Parents` fields on `avGroup` and `avDCS` nodes (see section 4.1.6 for details on these nodes) are never distributed, because the values of these fields are redundant with respect to the `Children` fields. A simple example will illustrate that: Whenever a node A is made a child of a node B, the `Parents` field of A is automatically updated to include a reference to node B. Under distribution, all replicated copies of A become children of the matching replicated copies of node B as well, and their `Parent` fields are locally updated to include a reference to node B. Thus, the `Parent` field needs not to be distributed.

5.1.2 The distribution group abstraction

The replication of the scene graph needs an application specific context in order to allow several independent distributed applications to coexist on the same network or even in the same process without interference. This context is provided by a *distribution* group.

A process can attach itself to one or more distribution groups (Figure 5.1). Objects can than be instantiated either locally as *local objects*, or in one of the attached to distribution groups as *distributed objects*. A local object exists only in the address space of the creating process, while for a distributed object copies will transparently be instantiated in the address space of each process attached to the distribution group. A distributed object belongs to no or to exactly one distribution group.

Distributed object creation in Avocado is a two stage process. First, a local object is created, which is then, in a second step, migrated to the desired distribution group. The migration involves the announcement of a new object to the distribution group and the dissemination of the current object state to all group members. Subsequent changes to an object's state will be forwarded to all distributed copies of the object.

State transfer to joining members

The ability to replicate object state and keep it synchronized between a constant set of processes is not enough to guarantee consistency in a dynamic application setting where processes join and leave distribution groups arbitrarily.

To illustrate this, consider the case of a distribution group with two member processes A and B. Both processes have already created several distributed objects in that group. Now a third process C joins the group.

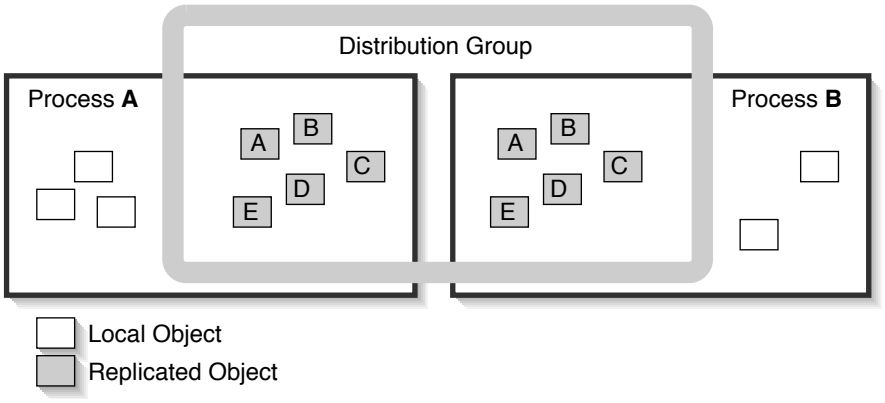

Fig. 5.1: AVOCADO objects can either be created local to a process, or they can belong a distribution group, and thus be replicated over all attached processes.

From now on, all three processes will be notified of *future* object creations and manipulations, but process C will not know of the objects that A and B had created *before* it joined.

This problem is solved by performing *state transfer* to every joining member. When a new process joins an already populated distribution group, one of the older group members takes the responsibility to transfer the current state of the distribution group to the new member. This involves sending all objects currently distributed in the group, with all their field values to the newcomer. After the state transfer, the new member will have the proper set of object copies for this distribution group. To prevent consistency problems, the state transfer is performed as one atomic action by suspending all other communication during the process.

The replication of the entire scene graph, paired with the state transfer to joining members, effectively provides a consistent view of the application state to all processes. New members can join an existing distribution group at any time and will immediately receive their local copy of the scene graph constructed so far in the distribution group. Furthermore, the application programmers do not need to concern themselves with distribution details. They can take the scene graph for granted on a per-process level, and can concentrate on the semantics of the distributed application.

The distribution group node `avNetDCS`

The group membership for the processes is managed by the `avNetDCS` class. The `avNetDCS` class is derived from `pfDCS` and `avPerformerNode` and is similar in functionality to the `avDCS` node. It has three additional fields and two additional methods that form the distribution group API. It is used

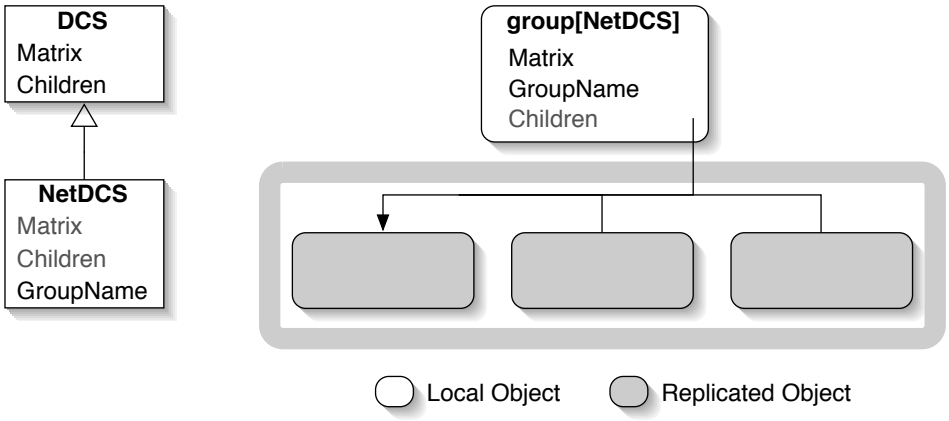

Fig. 5.2: The `avNetDCS` node represents a subtree of distributed geometry, just like the `avFile` node represents a subtree of geometry loaded from a file.

just like a normal node. `avNetDCS` is similar to the `avFile` file node, as it represents the geometry which is shared in the distribution group associated with the value of the `Groupname` field (figure 5.2). If the `avNetDCS` is added to the local modeling hierarchy, the subtree of distributed nodes which is managed by it will become part of the modeling hierarchy. Each process that wants to join a distribution group has to instantiate a `avNetDCS` node and set the appropriate distribution group identifier in `Groupname`.

The `Groupname` field identifies the distribution group the nodes represents. An empty string disables distribution for this node. This is the default value. If the `Groupname` field is changed, all nodes which have been associated with the former distribution group identifier will be deleted from the hierarchy. Any nodes present in the new distribution group will be added to the hierarchy.

The signatures of additional free functions for the C++ and Scheme bindings are shown in figure 5.3. `distribute_object()` takes a link to a locally created Avocado node and turns it into a distributed node in the distribution group represented by the `avNetDCS`. The node is immediately distributed, and copies will start to exist in any process currently attached to the group. The node will not automatically be part of the modeling hierarchy, and will have to be added to the hierarchy below the `avNetDCS` node.

`distribute_object()` takes a link to a currently distributed Avocado node and removes it from the distribution group. All distributed copies will be deleted with respect to their reference count (if a replicated copy still has a positive reference count, it will not be deleted, but there will be no further updates from the net).

```
C++:
  void distribute_object(const avLink<avDistributed>& obj);
  void undistribute_object(const avLink<avDistributed>& obj);

Scheme:
  (distribute-object obj)
  (undistribute-object obj)
```

Fig. 5.3: The distribution API.

5.2 Distributed event model

As described in section 4.1.5 the scene graph is augmented by a data-flow graph for event processing. Field connections can be drawn between object fields. Field value changes propagate along the connection from the source to the destination field and trigger a notification function on the destination field.

The general object notification scheme used to implement actions on objects in response to field value changes needs to be revised and adapted to the case where field value changes are performed on distributed objects.

5.2.1 Distributed state change notification and event handling

The notification scheme has to distinguish between *local side-effects* and *distributed side-effects* with respect to the semantics of the notification functions `notify()` and `evaluate()`.

The following example illustrates the situation. In figure 5.4, three processes A, B and C have joined a distribution group and a small scene graph has been constructed. Now, process A changes the value of field $f1$ on node a^A (a^A shall denote the distributed copy of node a living in process A). Because the node is distributed, the new value is communicated to field $f1$ on the copies a^B and a^C of the node.

As described in section 4.1.5 the `notify()` and `evaluate()` methods will be eventually called on node a^A allowing the implementation of side effects to the change in value. This may involve the modification of other field values from within the `notify()` and `evaluate()` functions. Lets assume the `notify()` function on node a will set a new value to field $f2$ on node a as a result of a change to field $f1$.

If now `notify()` was called on each copy of A as a result of the distributed field change to $f1$, each invocation would apply the change to field $f2$ on node A. This change would in turn be distributed to all copies. Thus field $f2$ would be updated exactly N times on each copy of node A, where N is the number of distributed copies involved, in this case three. This is clearly not desirable.

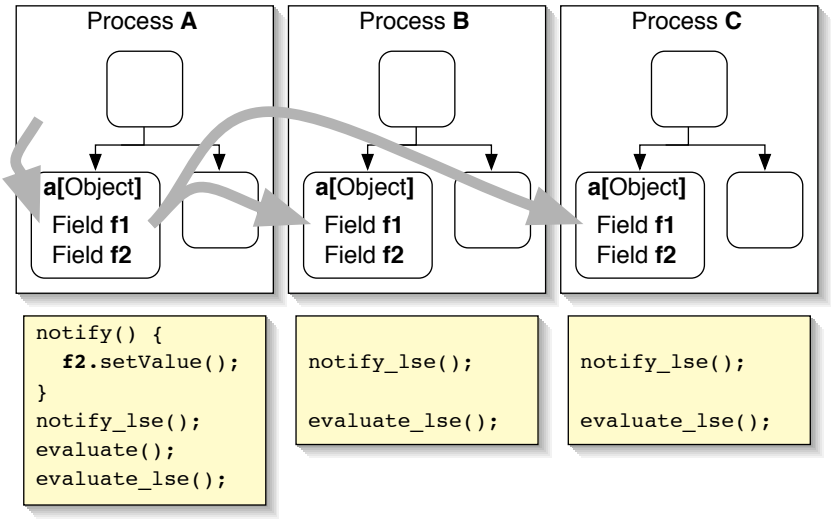

Fig. 5.4: The notification protocol is different for the originating field container and the distributed copies.

If neither `notify()` nor `evaluate` was called on the copies of the modified field container, it would be for example impossible to implement the `avFile` node, introduced in section 4.1.6, as a distributed node. The `avFile` node (see also section 4.1.6) loads specified geometry as a side-effect of changes to it's `Url` field. If one process changes the `Url` field on a distributed `avFile` node, this change must be applied to all distributed copies of the node, so that all copies represent the same geometry.

To resolve this problem, the notification function is split into two parts, one for *local side-effects* and one for *distributed side-effects*. The distributed side-effect versions `notify()` and `evaluate()` are only called on *one* distributed copy of the node. These functions may perform modifications to other fields without destroying the distributed notification scheme.

On the other hand, the local side-effect versions `notify_lse()` and `evaluate_lse()` will be invoked on *all* distributed copies of the node. They can be used to implement side-effects which have to be executed unconditionally, regardless of whether a field has been changed locally, or as result of a distributed change, as in the case of the `avField` node. The only restriction is that the local side-effect versions must not modify other field values.

5.2.2 Field connections in the distributed context

Based on a working notification scheme for field changes on distributed nodes, the behavior of field connections, introduced in section 4.1.5, in a distributed environment can now be investigated.

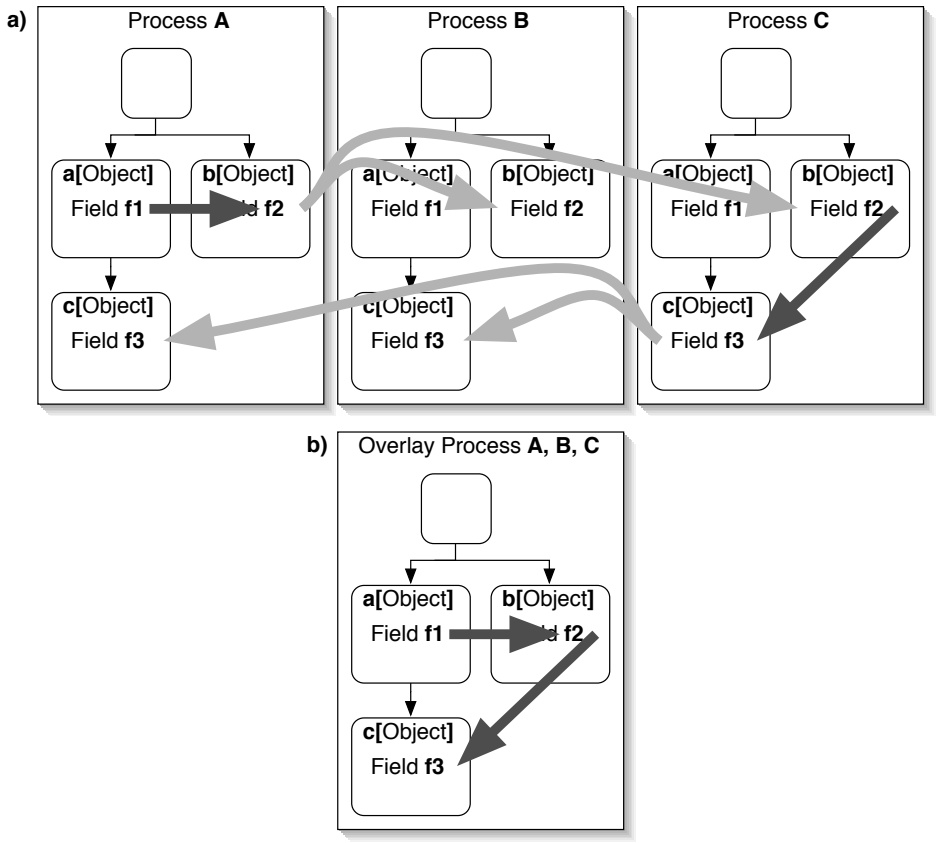

Fig. 5.5: The distributed data-flow graph (a) exhibits the same behavior as a stand-alone version with identical connections (b) would.

The basic assumption of the distributed event model is that field connections are local constructs that are not distributed. They only exist in the process which connected the involved fields. Consider the example given in figure 5.5a. Three processes A, B and C are attached to a distribution group and share a small scene graph of four nodes. Process A has connected field $f1$ on node a^A to field $f2$ on node b^A, while process C has connected field $f2$ on node b^C to field $f3$ on node c^C.

If A writes a new value to field $f1$ on node a^A, field $f2$ on node b^A will immediately receive the new value, due to the field connection between $f1$ and $f2$. Then, all distributed copies of node b will receive the new value for field $f2$. Because of the field connection in process C, the field $f3$ on node c^C is immediately notified of the change, and will receive the new value. This, in turn, will trigger an update of $f3$ on all distributed copies of node c^C. All field changes have become visible to all processes.

Without any further provisions, the behavior of the distributed data-flow graph matches the behavior of a stand-alone graph with identical connec-

tions. This effect can be illustrated by overlaying the three scene graphs in figure 5.5a to obtain figure 5.5b. If field $f1$ is changed, the new value propagates via field $f2$ to field $f3$.

From the application programmer's point of view, building interactive distributed applications is now just as straight forward as building standalone applications.

5.3 Guaranteeing application state consistency

The goal of the distributed object model is to provide identical copies of the application state to networked processes. For performance reasons, a local state changes never initiates a network wide update of the entire application state, but instead only the relevant state fragment is updated. Because synchronization messages require a certain amount of time to travel the network and because network response times are generally not deterministic, the mere presence of the network in context with the incremental nature of state update messages is a serious source of potential state inconsistencies.

State inconsistencies are especially annoying if processes make a local decision based on the inconsistent distributed state copy they maintain. Because of the inconsistency, the processes will probably come to different conclusions. For example, if object positions are inconsistent across processes, a test for object collisions will likely return different results for each process. As discussed in section 5.1.2, arbitrarily joining and leaving distribution group members are also a potential source of inconsistencies.

To maintain a consistent application state under these circumstances, AVOCADO employs the Ensemble [39] system from Cornell University.

5.3.1 The Ensemble/Maestro group communication system

The Ensemble system[39] is a high-performance, reconfigurable plug-and-play architecture intended for adaptive group communication applications. The basic functionality is to track membership of groups and to provide communication support among group members. Ensemble is a reimplementation of Horus [75] and as such implements the *process group model* [9] to provide reliable multicast communication between distributed processes. It is a successor to the ISIS system that implemented group communication with point-to-point messages and is used for the implementation of the transport layer in the DIVE system.

Ensembles architecture revolves around the notion of a protocol stack. Such a stack is constructed from simple micro-protocol modules, which can be stacked in a variety of ways to meet the communication demands of an application. Ensembles micro-protocol modules implement, among other things, basic sliding window protocols, fragmentation and reassembly, flow

control, signing and encryption, failure detection and recovery, group membership, and message ordering.

Ensemble is written in Objective Caml[52], a dialect of the functional programming language ML[38]. Ensemble is freely available from the Cornell website and is provided without fees or restrictions. Maestro[76] is a an object-oriented high-level interface to the Ensemble group communication system. The conceptual design of Maestro follows the group programming model of the Ensemble system.

Maestro offers a framework for applications to implement a state transfer to new group members. Three state transfer safety levels are supported:

Free State Transfer: Normal message exchange between group members can proceed during a free state transfer.

Protected State Transfer: During protected state transfer only *safe* messages can be send between group members. A message is considered safe, if it does not result in modification of the group state. The application has to explicitly mark messages as safe.

Atomic State Transfer: Only messages that are part of the state transfer protocol will be delivered during a state transfer. All other messages are delayed until after the state transfer is finished.

Several properties of Ensemble/Maestro group communication system are used to guarantee a consistent application state across processes. They are discussed in the following sections.

5.3.2 Consistency through total message ordering

Like other distributed 3D systems, for example Repo-3D or DIVE, which provide a consistent shared application state between group members, Avocado relies on total message ordering to guarantee consistency.

Maestro allows totally ordered messages, such that every group member will receive all messages sent to the group in exactly the same order. This guarantees that each process sees and applies all incremental state update messages, and that after a certain message has been applied by all processes, the global application state is consistent.

Although total ordering introduces additional network latency because a sequencer is used, it is a convenient and powerful way to guarantee consistency.

5.3.3 Synchrony through view atomic message delivery

Maestro manages the processes which communicate in one group as a list of group members, called a *view*. Whenever a new member joins the group, or an old member leaves the group, the view is updated accordingly. View

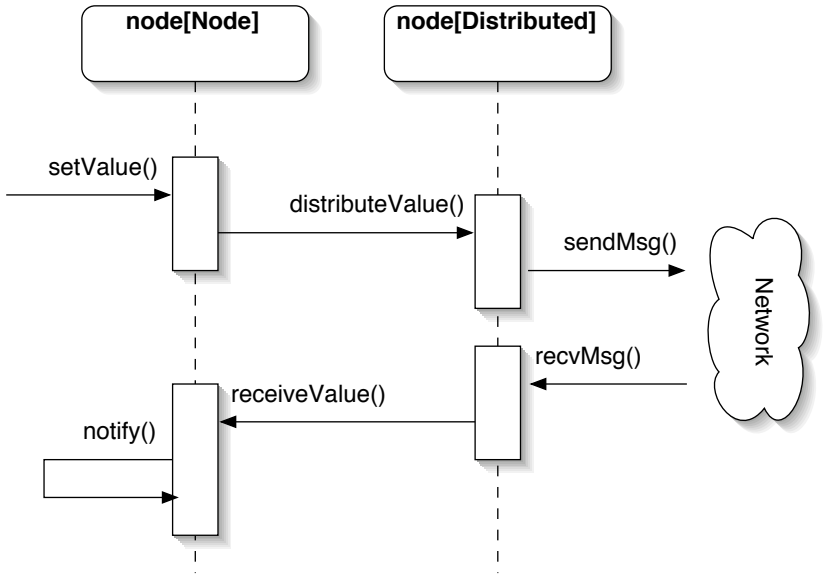

Fig. 5.6: To maintain consistency, all state update messages have to be totally ordered. Therefore, local updates to distributed objects are passed through the network layer in order to be properly sequenced.

changes are announced to all members, so that every member always has an up-to-date list of all other members in the group.

Messages to the group are sent *view atomic*, such that messages sent in one view are guaranteed to be delivered to all members in that same view, and *only* to those members. New members joining a group will not see any messages from old members until the new view is installed and is available to all members.

Taking into account the total ordering properties of the communication, the replicated application state is guaranteed to be consistent immediately after a new view has been installed. At that point, all old members have received exactly the same messages in exactly the same order, while new members have received no messages at all.

At this point it is safe to suspend all normal messaging between members and initiate a special *state transfer* phase in order to provide new members with a copy of the current application state.

5.3.4 Dynamic membership and atomic state transfer

Immediately after joining a group, new members have no knowledge about the history of the group, i.e. they do not possess a copy of the shared application state. Because all old members share a consistent state at this point, an atomic state transfer from one of the old members to the new

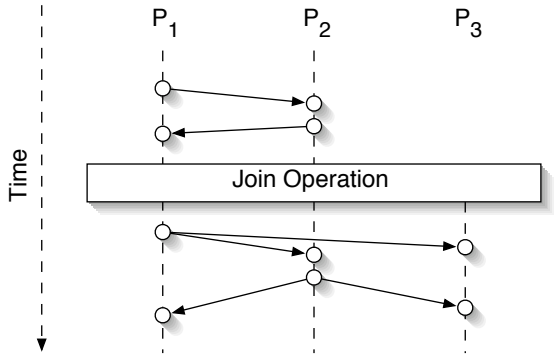

Fig. 5.7: Each join operation initiates an atomic state transfer to the joining member. Normal update message traffic is suspended during the state transfer.

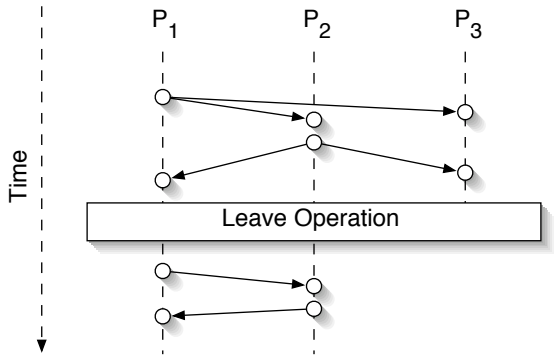

Fig. 5.8: Each leave operation initiates view change.

member is sufficient to bring the new member up to date. During the state transfer all other communication in the group is suspended. After the transfer the new member has exactly the same state information as the old members, and normal operation can resume.

The atomic state transfer allows for addition of new members to a group at any time without destroying consistency. The apparent drawback of this approach is the suspension of normal communication during the state transfer. If the application state takes a considerable amount of time to transfer, this will be noticeable to the user, as the application 'freezes' with respect to updates from the network.

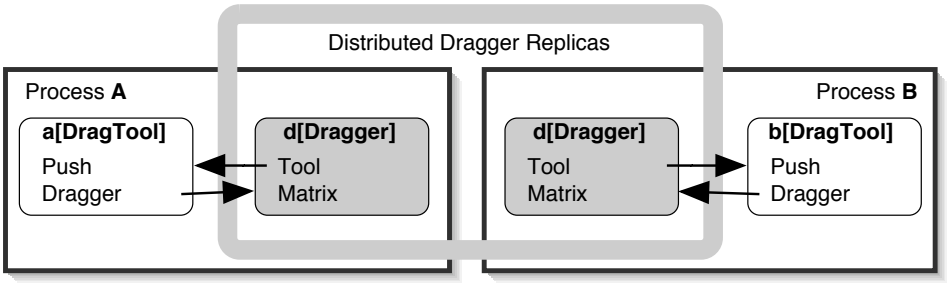

Fig. 5.9: The dragger that is attached to the object is shared.

5.3.5 Distributed locking through total message ordering

One of the challenges of applications that have multiple threads of control is concurrency management. In distributed VE applications each participating process can access and modify any object that is part of the environment. If multiple processes choose to manipulate the same object at the same time, this can lead to undesirable and uncontrollable results.

An example for this situation is the tool-based direct interaction presented in section 4.3. Consider a situation where two users, each controlling a tool, try to move around an object at the same time. Figure 5.9 illustrates that the dragger that is attached to the object is shared and thus replicated into the local address space of each process, while the tool that is controlled by each user is not shared and exists only in the local address space of each object. Because interaction between tool and dragger is handled locally by each process, conflicting manipulations of the two associated dragger replicas may occur.

Without explicit concurrency management, the implicit policy to resolve access conflicts could be called *the last one wins*: Whichever tool accesses the object last will take control of the object, regardless of whether the object was previously controlled by another tool or not. A desirable solution is to grant exclusive object access to one tool during an interaction, effectively preventing any interference from other tools.

A simple approach would be to extend the tool to perform a test whether an object is currently being controlled by another tool before taking control. In a non-concurrent, single-threaded environment this would work, in a concurrent environment with multiple threads of control and with multiple tools performing this test at the same time, it will lead to *race conditions* that will easily break the interaction protocol between tools and draggers. Too overcome this problem, the test for availability of the dragger and the acquisition of the dragger needs to be one atomic operation that can not be interrupted by other processes. This area of code is called a *critical section*.

To prevent a critical section from being executed by more than one process at a time, the existence of a locking mechanism, often called a *lock*,

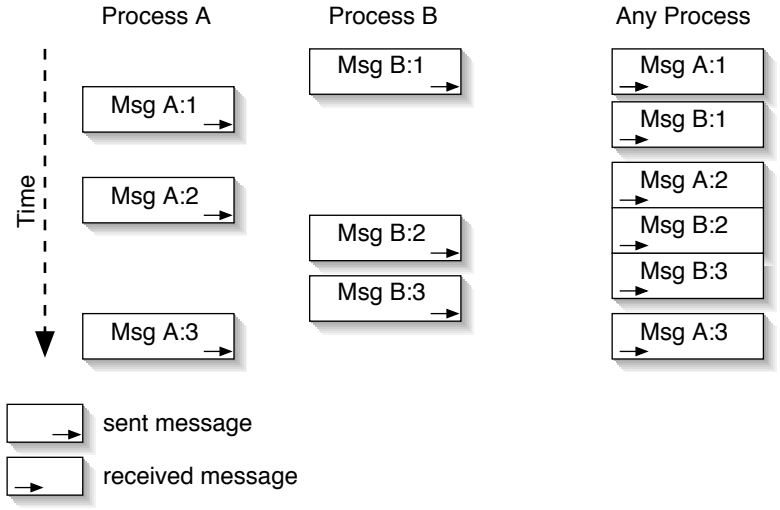

Fig. 5.10: Under total ordering constraints, all messages sent to the group are received in the same order by all group members, including the processes that sent the messages.

is required. A lock can only be acquired by one process at a time, and until the process that is holding the lock releases it, no other process can successfully acquire it. The important property of locks is that an attempt to acquire a lock can *never* lead to a race condition. To implement race-free dragger control, a tool would acquire a lock, test the availability of the a dragger, seize the dragger if available and release the lock. Because the critical section of code is guarded by a lock, no concurrency problems would occur.

A distributed lock

Avocado uses the total ordering property of its transport layer to implement a distributed locking mechanism. Total ordering guarantees that the order of received messages is the same for all processes in a communication group (see figure 5.10). In particular, this is true for all object field update messages that are sent to synchronize the fields of distributed object replicas in a distribution group.

In a communication group with total message ordering, a very simple distributed locking mechanism can be built. Assume that the distributed lock l is free. A process that wants to acquire lock l sends a corresponding message to the entire group. Figure 5.11 shows processes A, B and C sending acquisition requests for lock l to the group at roughly the same time. Because of the total ordering properties of message delivery, all processes will receive the three acquisition messages in the same order. This means, that the

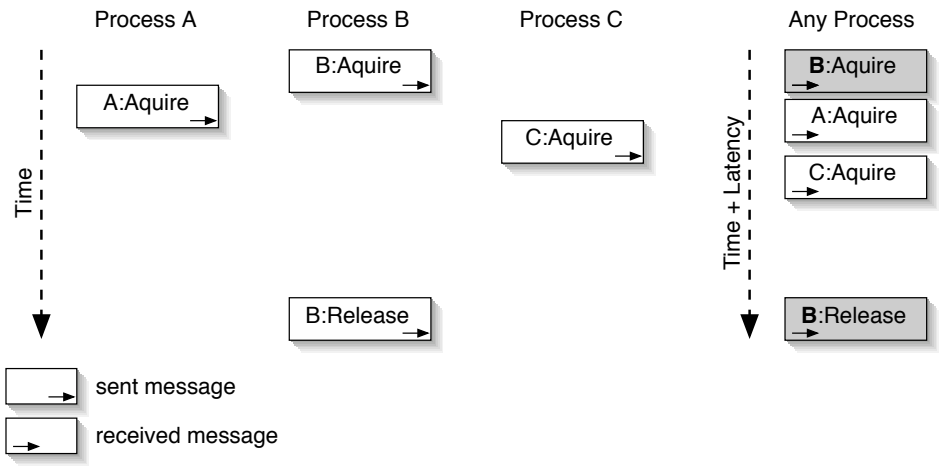

Fig. 5.11: All processes can decide locally which process should be granted the lock, because all lock messages are received in the same order. Here, process B wins the race for the lock.

race condition for lock l can be resolved locally by each of the participating processes and does not require further communication. Each process will locally grant the lock to the process thats acquisition message is received first.

In this example (Figure 5.11) the lock will be granted to process B. All other acquisition messages for lock l will be discarded until process B is done with the lock and sends a corresponding release message. Process B knows it has won the lock when it receives its own acquisition message without having received any prior acquisition messages from processes A and C. Process A and C know they lost the race for the lock when they receive the acquisition message from process B without having yet received their own messages. No process in the group will send a new acquisition message for lock l until it received the corresponding lock release message from process B.

The `avLock` field type

Based on the previous results, AVOCADO defines the `avLock` field type that is meant to be used as a locking facility for distributed objects and applications that need it. First, interface and implementation of `avLock` will be described and then an usage example will be given by incorporating locking into the tool-based interaction framework introduced in section 4.3.

The `avLock` class defines a field type that accepts values of type `String` via the standard `setValue()` and `getValue()` methods (see section 4.1.1 for details on fields). The embedding field container is notified of field value

`setValue("acquire")`	Request the lock for the calling process. If the lock is not available, the calling process is queued.
`setValue("request")`	Request the lock for the calling process. If the lock is not available, the calling process is *not* queued.
`setValue("release")`	Release the lock the calling process is holding.
`getValue() == "granted"`	The lock is held by the calling process.
`getValue() == "occupied"`	The lock is held by another process.
`getValue() == "free"`	The lock is currently not held by any process.

Fig. 5.12: Possible operations on an `avLock` object.

changes by a call to the standard `notify()` method. The possible value related operations on an `avLock` object are described in figure 5.12.

If the containing object of an `avLock` field is not distributed, it behaves as though the lock where always held. A call to setValue() has no further side effect, and directly calls notify() on the containing object. Likewise, getValue() always returns `"granted"`. This prevents objects that use locking in a distributed environment from breaking in a non-distributed environment.

The `notify()` method of the embedding field container is called whenever a status change of the lock is of potential interest to the particular process. If the application requested the lock via `setValue("acquire")` or `setValue("request")`, the `notify()` method will be called on the corresponding field container when the status of the lock has changed. No polling of the lock value is necessary. In the `notify()` method the application can learn whether the lock was granted or not by examining the lock value with `getValue()`. If the lock value returned is `"granted"`, the lock has been granted, while a value of `"occupied"` indicates that the lock has been acquired by some other process.

The implementation of the locking mechanism relies on the fact that synchronization of field values on the object replicas is conducted by totally ordered messages. Further, the group-wide unique end-point-id (*eid*) of the processes is used to identify the lock requests. The mapping from the field interface to the distributed locking algorithm developed in the previous section is as follows:

`setValue("acquire" | "request")`: Send the *eid* of the calling process as

a request to acquire this lock to the group. This is performed as part of the normal object update message for the embedding object.

`setValue("release")`: Send the *null eid* as a request to release this lock to the group. This is performed as part of the normal object update message for the embedding object. It is only executed if the calling process currently holds the lock.

`notify(eid)`: An update message for the lock field containing a valid *eid* has been received by the network layer. If any process is currently holding the lock and the message was generated by `setValue("acquire")`, the message is added to the end of the request queue. Otherwise, if the received *eid* equals the *eid* of the process, the lock is considered granted to the requesting process, `notify()` is called on the embedding field container and further application calls to `getValue()` will return `"granted"`. If request eid and process eid are not equal, the requesting process is also notified, but `getValue()` will return `"occupied"`.

`notify(0)`: An update message for the lock field containing the *null eid* has been received by the network layer. The lock is now considered released. If not empty, the next eid entry will be popped from the request queue and be handled as if received by `notify(eid)`.

The use of the request queue with `setValue("acquire")` has one consequence that application developers must be aware of. If a process requests a lock, this lock *will* eventually be granted, but there is no way for the process to know how long this will take (and yes, *for ever* is an option). Because request and grant are completely asynchronous, an application may decide to stop waiting for a once requested lock. To avoid being subsequently granted the lock in a probably unsuitable moment, the application can and should cancel the acquisition request by calling `setValue("release")`. In that case the process will be notified as if the lock had been denied.

Concurrent interaction with locking

To prevent race conditions in a concurrent interaction where two or more tools try to manipulate the same dragger at the same time, the dragger base class `avToolDragger` is equipped with an `avLock` field.

Before a tool and a dragger are connected at the beginning of an interaction sequence, the tool stylus has to acquire the lock on the dragger to assure exclusive access to the dragger. This enforces, that while one tool stylus holds the lock on the dragger, no other tool stylus can invoke a concurrent interaction between that dragger and another tool. At the end of the interaction, tool and dragger are disconnected and the lock is released. Because `avField` locking is an asynchronous process, the tool stylus maintains a field connection to the lock in order to be notified of a successful

lock acquisition. The field connection is established after the lock has been requested and released after the lock has been granted. If the user releases the push button before a lock is granted, the tool stylus revokes the lock request and disconnects from the lock field.

This extended interaction protocol between tool stylus, tool and dragger works unmodified in undistributed as well as in distributed environments and guarantees exclusive and race free interaction.

5.4 A simple distributed application example

This section presents a SCHEME example on how to write a distributed AVOCADO application.

A *server application* joins a distribution group and loads a simple geometry. A `avMatrixDrager` is attached to the geometry. The default point tool is replaced with a drag tool, so that the user can drag around the geometry. Any number of '*client applications* can join the same group and then immediately see the geometry the server has already loaded.

Note that the terms *client* and *server* are used on the application level, not on the distribution transport level. One process, the server, loads a geometry, which will be seen by all other group members, the clients. In this sense, every process in the group can be a client and a server at the same, as every process can add geometry to the group, and see the geometry the other processes provide.

The server code

First, an instance of the `avNetDCS` class is created. The `Groupname` field is set to the name of the desired distribution group, *test*. Then the node is added to the `aview` hierarchy (figure 5.13).

Then a `avFile` node and a `avMatrixDragger` are instantiated and are registered with the distribution group.

Then the hierarchy is constructed. The now distributed `file` node is added to the `avNetDCS` node as a child, and the also distributed `dragger` is attached. These operations are all field value manipulations and thus happen for all copies of these nodes at all processes which are members of the distribution group *test*.

Finally the filename for the geometry is set. All distributed copies of the `file` node will now load the geometry found in the file *unit-cube.iv*.

Now the default point tool on the `aview` stylus is replaced with a drag tool, so that the user will be able to drag the geometry around. Because the drag tool operates by changing the `Matrix` field on the dragger, and the dragger in turn changes the `Matrix` field on the `file` node, any manipulations done by one user, will be visible to all other users in the distribution group.

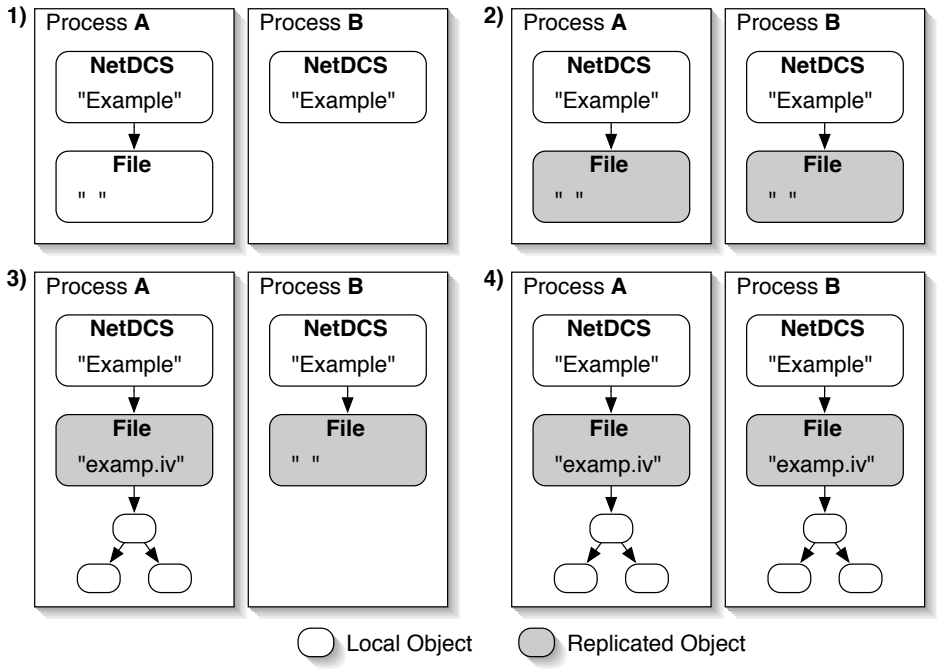

Fig. 5.13: (1) The `avNetDCS` node is created in both processes separately and the distribution group is joined by assigning the `GroupName` "Example". (2) The `avFile` node is created in the server process. (3) The `avFile` node is registered as a distributed node and parented to the `avNetDCS`. (4) The `avFile` node is replicated to the client side automatically. Setting the the `Filename` field on the server will load some geometry. (5) The field change is replicated to the client. (6) The client loads the same geometry as the server.

The client code

The client creates an instance of the `avNetDCS` class, joins the *test* group, and adds the node to the local `aview` hierarchy (see figure 5.15). If the server has already added the `file` node to the distribution group, the cube will show up immediately. The client also replaces the default point tool with a drag tool.

There can be as many client processes as needed. Every user can now see and manipulate the geometry loaded by the server process. Furthermore every user can load some other geometry and add it to the same group.

```
;; create distribution group node and add to scene graph
(define net-group (make-instance-by-name "avNetDCS"))
(av-set-value net-group 'Groupname "test")
(av-add net-group)

;; create avFile and avMatrixDragger nodes and add to group
(define file    (make-instance-by-name "avFile"))
(define dragger (make-instance-by-name "avMatrixDragger")))
(-> net-group 'distribute-object file)
(-> net-group 'distribute-object dragger)

;; add file node to group and connect dragger
(av-set-value net-group 'Children (list file))
(av-set-value file      'Dragger  (list dragger))

;; load geometry
(av-set-value    file  'Filename "unit-cube.iv")

;; instantiate a tool and connect to tool stylus
(define tool (make-instance-by-name "avDragTool"))
(av-set-value av-stylus 'Tool tool)
```

Fig. 5.14: The server code.

```
;; create an avNetDCS and join the group
(define net-group (make-instance-by-name "avNetDCS"))
(av-connect-from net-group 'TimeIn    time-sensor 'Time)
(av-set-value net-group    'Groupname "test")
(av-add net-group)

;; replace the point tool with a drag tool
(define tool (make-instance-by-name "avDragTool"))
(av-connect-from tool 'TimeIn time-sensor 'Time)
(av-set-value av-stylus 'Tool tool)
```

Fig. 5.15: The client code.

5.5 Pacman: A complex distributed application example

The distributed Pacman game was inspired by the ATARI ST game classic MidiMaze, which borrows the main character from the Namco arcade game *Pac-Man*, and was built as a distributed proof of concept application that has served as a test-bed for the evaluation of different implementation strategies for the distribution support in AVOCADO. This variant of Pacman is rather aggressive as the objective for each pacman is to run around in the maze and shoot red bullets at the other pacmans (see figure 5.16).

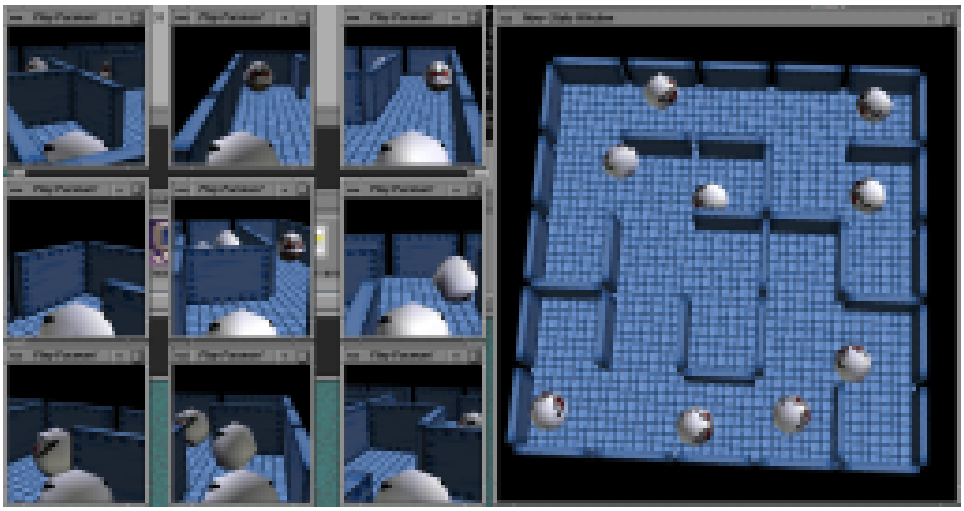

Fig. 5.16: The distributed Pacman game is situated in a maze that is randomly generated by a dedicated maze process. The maze is placed into a distribution group. Player processes join the maze and let each user interactively control the actions of one pacman.

The application consists of one *maze process* and any number of *pacman processes*. The maze process generates and distributes the maze geometry in the pacman distribution group, and thus provides the playing field for the pacmans. A pacman process can join the maze process in the distribution group and while doing so adds a pacman to the maze. The pacman is controlled by the user via mouse and keyboard and can move around freely within the maze. A pacman can fire bullets at the other pacmans. If a pacman is hit by a bullet, it is deflated and paralyzed for a small amount of time (see figure 5.18).

The implementation of Pacman demonstrates a number of techniques that are useful during the development of distributed Avocado applications. In particular:

Compound objects: An class describes the properties of an compound object as a collection of fields. Upon instantiation, an internal object representation that reflects the property values of the object.

Abstract distribution: Because the field values of a compound object completely describes the objects state, the internal representation does not need to be distributed, but can be reconstructed by each process. This can significantly reduce the amount of bandwidth needed to replicate a complex geometric object as only a few field values need to be transmitted instead of a completed geometric description of the object.

Local behavior: An objects behavior evaluation ultimately results in one or

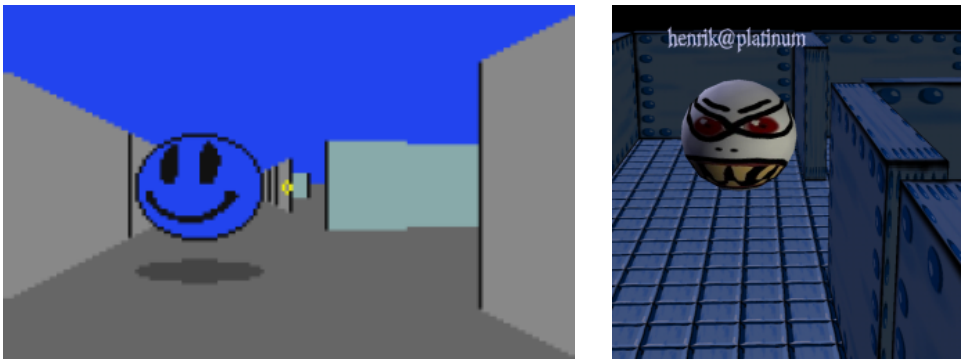

Fig. 5.17: Original and Fälschung: The Pacman game (right) is loosely based on the ATARI ST game MidiMaze (left), one of the first 3d networked multi-player first-person shooters. The text above the pacman shows user and host name of the controlling process.

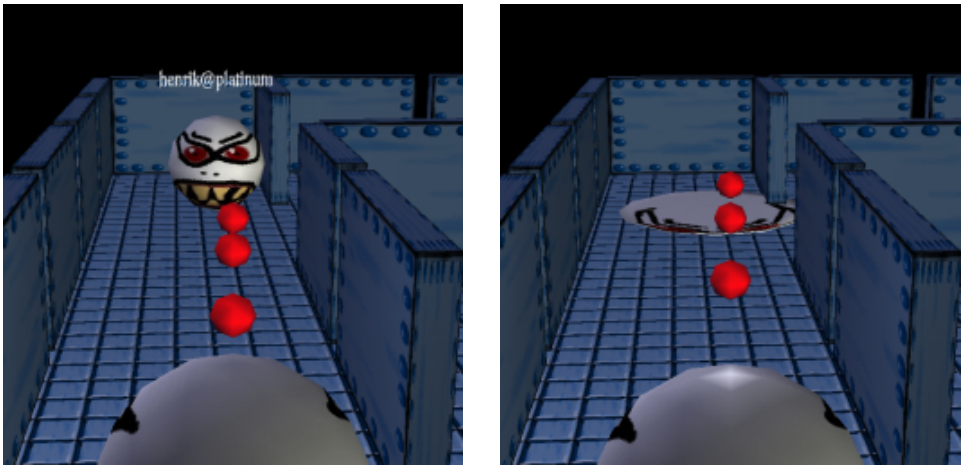

Fig. 5.18: The pacmans can move freely through the maze and shoot at each other (left). Once hit, a pacman is flatted and disabled for a period of time (right)

more field value changes. In a distributed environment the local behavior is only evaluated locally by one process for one object copy and the result is automatically shared via field replication. The result of the behavior is distributed, not the behavior.

Distributed behavior: The behavior of the internal representation of an compound object executed at each copy of the object. The behavior evaluation is distributed, not the results.

Figure 5.19 shows the node class diagram for the pacman application. The pacman classes are derived from avDCS and represent the different

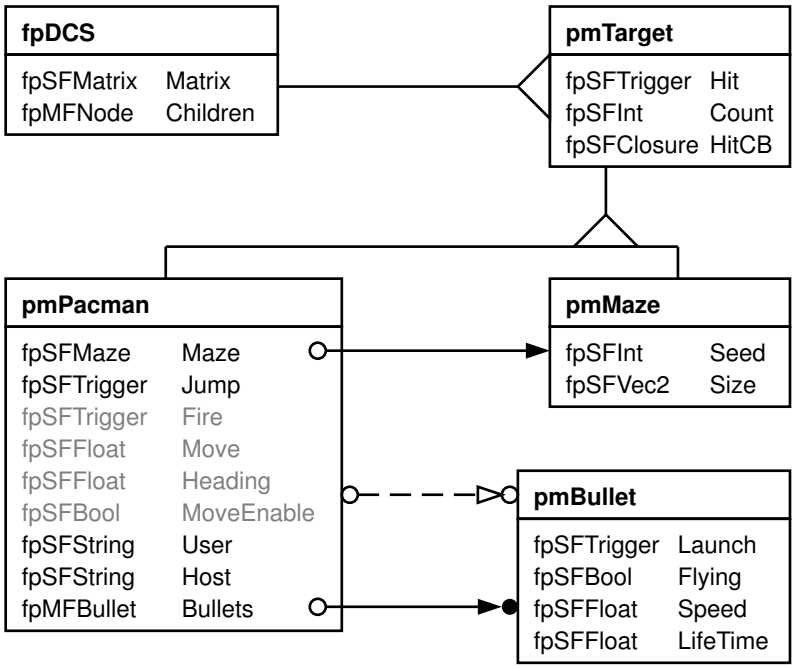

Fig. 5.19: The class diagram of the pacman example. The game is build from a maze , the pacmans and any number of bullets. The inheritance graph reflects that the maze and the pacmans are targets, which means they can be hit by bullets.

components that make up the game: the maze, the pacmans and the bullets.

The `pmTarget` class

The `pmTarget` class is an abstract class that is meant to be sub-classed if the instances of a class need to be hittable by a bullet. The collision detection performed by the `pmBullet` class only recognizes bullet collisions with objects derived from `pmTarget`, all other objects are ignored. The `Hit` field of `pmTarget` is triggered for each hit, and a derived class can define appropriate behavior based on the notification. Additionally, a scheme closure can be defined in field `HitCB` that is evaluated for each hit. The `Count` field provides the number of hits that the particular target has suffered during it's live time.

The `pmMaze` class

The pmMaze class (see figure 5.20) defines the geometry of the playing field. pmMaze is a compound node that exposes a concise field interface and assembles its internal representation from several sub-nodes. A maze' walls are build from a shared instance of an `avFile` node that loads the wall

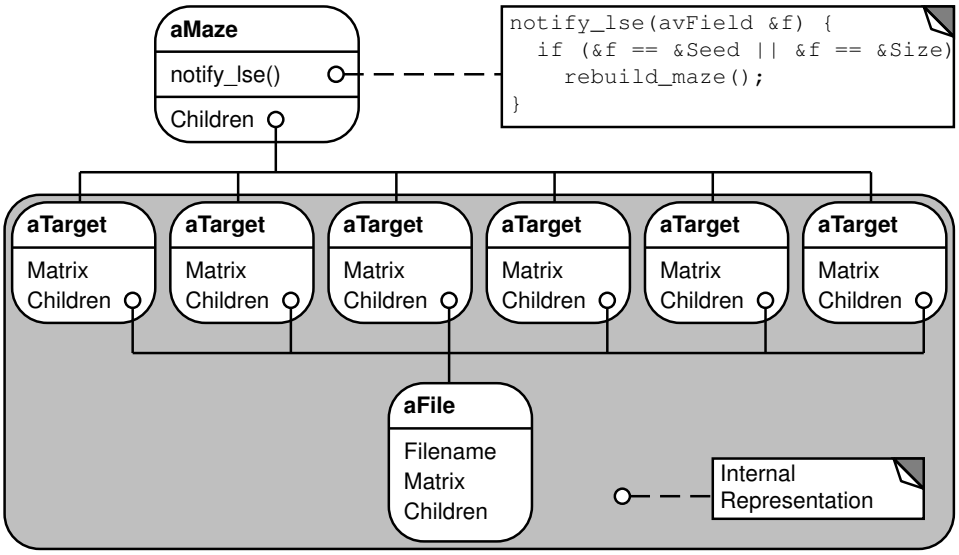

Fig. 5.20: The maze walls are build from a shared instance of an `avFile` node that loads the wall geometry. The `Seed` field value is used to seed the random number generator that controls the the layout of the maze.

geometry. This node is multiply referenced by `avTarget` nodes that define the position and orientation of each wall segment and identify the walls as targets. Because the walls are derived from `avTarget` they will block bullets on collision.

The `Seed` field value is used to seed a random number generator that controls the the layout of the playing field. Because all `avMaze` nodes use the same maze generation algorithm, and thus every distributed instance of `avMaze` creates the exact same maze for a given seed value, the value of the `Seed` field together with the value of the `Size` field completely determines the layout of the maze. The distribution of the `avMaze` node is extremely efficient, only the values of `Size` and `Seed` need to be communicated over the network. Each distributed copy rebuilds the internal maze representation from these two values. Because it will almost always be faster to generate a complex geometry locally than to transmit a serialized representation of that geometry over a network connection and then reconstruct it, the abstract distribution of compound objects is an important strategy to build high-performance distributed applications.

The `pmPacman` class

The pmPacman class also defines a pacman as a compound node. On instantiation, `avPacman` creates a sub-tree of nodes that represent the pacman object as shown in figure 5.21.

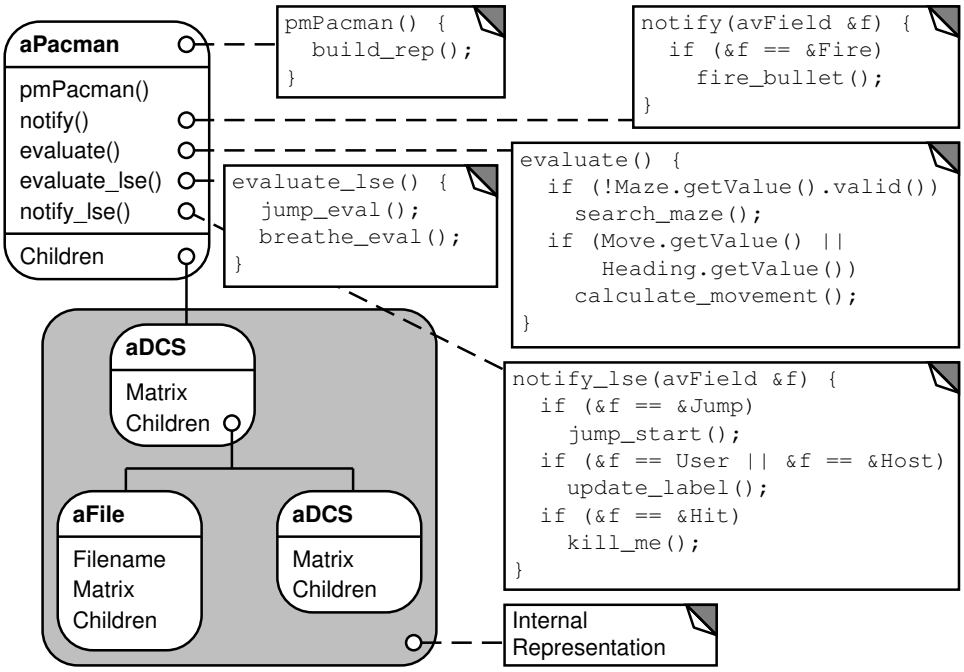

Fig. 5.21: The pmPacman class defines a pacman as a scene graph node that uses an internal hierarchy of nodes to implement the pacman appearance. The behavior is implemented in the four notification functions `evaluate()`, `notify()`, `evaluate_lse()` and `notify_lse()`.

Action	**Field**	**Behavior**	**Method(s)**
move	`Move, Heading`	local	`evaluate()`
jump	`Jump`	distributed	`notify_lse()`, `evaluate_lse()`
get hit	`Hit`	distributed	`notify_lse()`, `evaluate_lse()`

Tab. 5.1: The behavior of a pacman is defined by three actions that are triggered by field value changes.

A pacman exhibits a rather complex behavior that depends on external user interaction and on the internal application state. The user controls the pacman using mouse and keyboard. The `Move` and `Heading` fields are connected to the vertical and horizontal mouse movement offset respectively. The `Fire` trigger field is connected to the left mouse button, while the `Jump` trigger field is connected to the middle mouse button. Whenever a bullet detects a collision with a pacman, the Hit field on the pacman is triggered to notify the hit. Based on these five input Fields, an avPacman object performs three different actions as shown in table 5.1.

The jump and the hit behavior are defined as distributed behaviors and hence are implemented in the `evaluate_lse()` and `notify_lse()` variants

of the field value change notification functions. As a result, the local modification of the `Jump` field on the pacman in response to a mouse button click, for example, is distributed to all copies of the pacman object and the animation of the jump is calculated and performed locally by each pacman. Using distributed behavior is a good strategy to conserve bandwidth. In this case only the value change of the `Jump` trigger field is transmitted, while the successive modifications of the relevant matrix field during the animation of the jump are performed locally at each process and are not transmitted over the network. Distributed behavior can be used safely with respect to consistency if the result of the behavior is not accumulating.

In contrast, the move behavior is defined as a local behavior and is implemented in the `evaluate()` function of the pacman. `evaluate()` is executed only by one process, usually the one that created the corresponding object. Therefore, the calculation of the new pacman position is performed only by one process and the resulting modification of the pacmans transformation matrix is replicated to all distributed copies. Each evaluation of the move behavior modifies the position of the pacman relative to its previous position. Implementing the movement of the pacman as a local behavior assures the overall consistency of the pacman position because each new position is communicated as an absolute value. If behaviors with accumulating results are implemented as distributed behaviors, small differences in the results locally evaluated by the different processes will also accumulate, leading to an inconsistent application state. In this case, after some time the processes would see differing values for the position of a certain pacman.

The `pmBullet` class

Each pacman is armed with four bullets that can be fired at other pacmans. The `pmBullet` class (see figure 5.22) implements bullet movement and collision detection as distributed behaviors. The movement is triggered by the `Launch` field, while the `Speed` and the `LifeTime` fields control the rate and the duration of the movement. During flight the bullet checks collisions with other geometry and, if it hits an object of type `avTarget`, triggers the `Hit` field on that object.

Because movement and collision detection for a bullet are distributed behaviors, they are evaluated by the one process that created a particular bullet instance. Thus, the potentially time consuming calculation of object collisions for all bullets is effectively distributed over all processes in the distribution group, because each process calculates collisions only for those bullets it created. In contrast to the popular central gaming server approach found in on-line FPS (First Person Shooter) style games like Quake and Unreal, the distributed collision detection ensures much better scalability with respect to available processing resources.

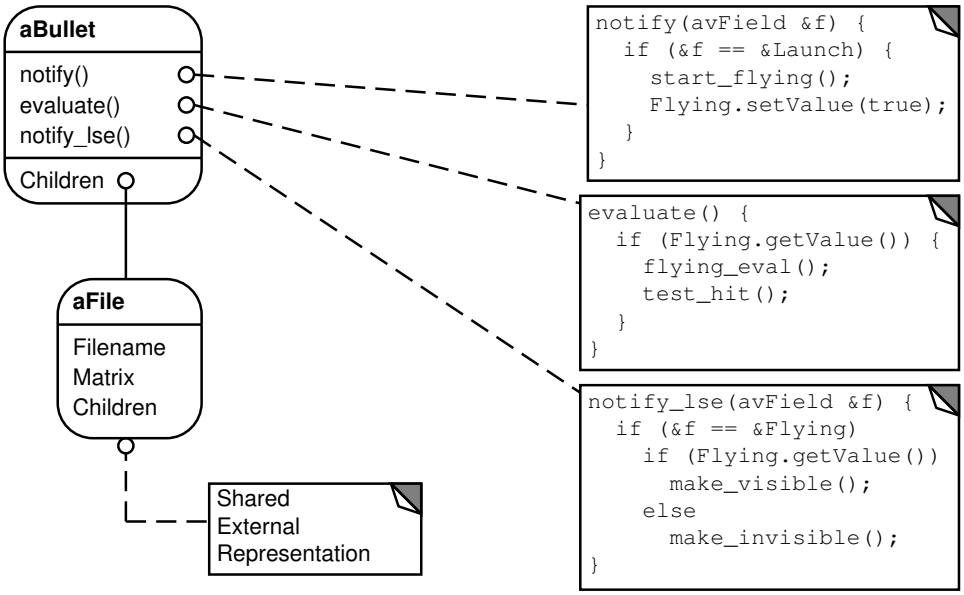

Fig. 5.22: The bullet is launched by the pacman and autonomously performs the collision detection to determine possible target hits.

Pacman Summary

The AVOCADO implementation of the pacman game has served as a test case for distributed application development. It demonstrates how the distributed AVOCADO object model is used and introduces two design patterns that lead to well balanced scalable distributed applications:

- object distribution at different levels of abstraction.
- local and distributed behavior evaluation.

5.6 Summary

In this chapter, details of the distributed aspects of the AVOCADO implementation have been presented. The implementation closely follows the general DVE architecture that has been described in section 3.2, and as such fulfills all requirements formulated in section 3.1.

The main points of the AVOCADO distribution implementation can be summarized as follows:

- The distributed AVOCADO object model provides programmers with the concept of a shared scene-graph that is accessible from all participating processes of a distributed application. The scene-graph is transparently replicated such that each process holds a local copy of the scene-graph.

- The unification of distributed field change notification and field connection event handling gracefully extends the field connection concept to distributed applications. This new approach provides the same evaluation characteristics in distributed applications as in stand-alone applications. It effectively simplifies the development of distributed interactive applications.

- The Ensemble/Maestro group communication toolkit is used to implement the AVOCADO network transport layer. Ensemble/Maestro provides reliable multicast communication primitives with strong message ordering guarantees that are used to implement the following AVOCADO distribution features:

  - Reliable and consistent state replication
  - Fully dynamic group membership changes
  - Automatic state transfer to joining members
  - Distributed locking

- As described in section 4.1.10, the AVOCADO component model supports application specific extensions through subclassing. All subclasses automatically inherit the distribution properties and are fully usable in distributed applications.

The successful implementation of distribution functionality in the AVOCADO framework validates the applicability of the general DVE framework design that has been presented in chapter 3. However, the question of scalability has not been addressed so far. The following chapter analyzes the general scalability properties of DVE systems and proposes a scalability solution for the AVOCADO framework.

6. Scalability in distributed virtual environments

This chapter reviews the approach to distribution support presented in the last chapter with a closer look at it's scalability properties. The inherent scalability problems are identified and a solution to these problems is developed. A prototypical implementation, implemented as an Avocado extension, is described.

6.1 Introduction

The transparently distributed scene graph presented in the previous chapter provides a convenient and intuitive abstraction for the development of distributed applications. It's conceptual strength, the distribution of the entire scene graph to all participating processes, is at the same time it's greatest weakness when it comes to scalability. The complete replication scheme is not easily applicable to large scene graphs and a great number of processes, mainly for the following reasons: process memory, network bandwidth and heterogeneous processing and rendering capabilities.

Process Memory: Because process memory is a limited resource the total size of the in-memory representation of the shared scene graph is limited by the amount of memory that is available to the smallest process in a distribution group. Thus, scalability abruptly ends if one process in a distributed application reaches its memory limits and is not able to hold the entire scene graph in main memory.

Network Bandwidth: The available network bandwidth may also be a serious constraint for scalability. For every process that joins a distribution group the entire state of the distributed applications, i.e. the shared scene graph, has to be distributed. Additionally, the normal update message traffic tends to increase in a linear fashion with the number of processes. Once the network is saturated with state transfer messages to new members and update messages between old members, the overall performance of the distributed application drops to almost zero.

Processing Capabilities: Every process needs to process every state transfer message and every update message from every other process in the distribution group. With an increasing number of processes in the group this at some point saturates input buffer space and/or processing capabilities of the process. The process can no longer process all incoming messages and the entire group needs to slow down in order to maintain consistency of the shared state.

Rendering Capabilities: A heterogeneous collection of machines with different processing and rendering capabilities may also present a problem if the scene graph increases in size. While more capable machines may be able to render a scene with an acceptable frame rate, the frame rate may drop below any acceptable limit on smaller machines.

Section 6.2.1 presents a scalability analysis of two existing approaches to address the described scalability problems, and discusses their scalability behavior with respect to the aforementioned problems.

The more complex the scene graph becomes and the more processes participate in a distributed application, the less likely it becomes that every process is interested in the entire scene graph and thus in every state transfer message and every update message from every other participating process. This is due to the fact that the *area of interest* for each rendering process in a virtual reality application is closely related to the concept of visibility. In a typical VR application the area of interest is mainly a function of viewing direction and viewer position.

The concept of visibility has been exploited in different ways do solve another closely related problem associated with real-time rendering applications, performance. Section (6.3) introduces two more common real-time rendering acceleration techniques, *view frustum culling* and *level-of-detail evaluation*. Both techniques accelerate rendering by identifying and processing only those parts of the scene graph which are potentially visible to the user. They define a visibility based area of interest.

Section 6.4 proposes to apply these techniques to the scalability problem for distributed applications. The use of visibility information to define an area interest for each participating process which allows an effective reduction of the amount of information that needs to be exchanged between processes in large distributed multi-user virtual reality applications.

Section 6.5 describes a prototypical implementation of AVOCADOs approach based on the built-in distribution support which has been introduced in the previous chapter.

6.2 Scalability analysis

To assess the effects of scaling for the different approaches to distribution, the development of a simple reference model is necessary. Based on this

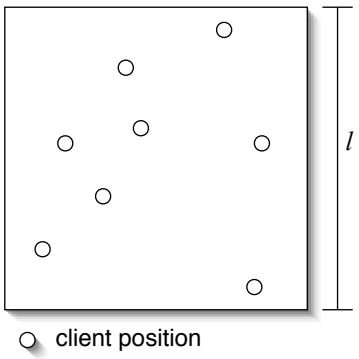

Fig. 6.1: The environment is described as a flat, square, 2-dimensional plane.

model the scalability properties of the approach developed in this thesis are compared against characteristics of existing approaches.

6.2.1 An environment model for scalability analysis

The model describes the environment as a flat, square, 2-dimensional plane. The size of the environment is determined by the side length of the square l_{env}. The geometry of the environment is entirely described using polygons, no additional representations like texture or volumetric data is used. The average polygon density d_{poly} describes the overall complexity of the environment. To eliminate the need to specifically handle border conditions, a torus topology is assumed for the environment.

One process acts as server that generates the polygonal description of the environment and makes it available to a number of client processes. Clients are evenly distributed over the environment, the average client density is d_{client} (see figure 6.1). A multicast transport is used, such that each state update message needs only be sent once and reaches all other clients.

Scalability is measured by the effects that the increase of relevant system parameters has on the systems resource requirements. Significant increase of resource consumption results in low scalability, while resource requirements that are decoupled from scalability parameters are signs for a highly scalable system.

The scalability parameters that are considered are:

Environment Size: Many application areas demand high scalability with respect to the size of the environment. Examples are military battlefield simulations that are staged in environments with a potentially very large extend, and on-line role-playing games that are situated in environments of ever increasing dimensions.

Polygon Density: The current trend toward more realistic rendering of virtual environments requires increasingly detailed environment descriptions. In the proposed model, the average polygon density directly corresponds to the environment complexity.

Number of Client Processes: Many DVE applications require high scalability in the number of client processes participating in the application. The aforementioned military simulations and on-line role-playing games need to accommodate several hundreds or even thousands of participants simultaneously.

To evaluate how the system responds to the scaling of above parameters, the following resource requirements are determined:

Client Process Memory: The amount of physical memory that a client can use for the representation of the environment is limited by the total amount of physical memory available to the client process. This hard upper limit must never be exceeded. Any increase in environment size or polygon density potentially increases the amount of memory needed for the representation.

Client Update Bandwidth: All processes can modify the environment. These modifications are communicated to all other processes. For each view, a processes needs to receive all state update messages that are generated by the other processes. The client update bandwidth is the minimum required bandwidth to do that.

Rendering Performance: Increasing environment size or polygon density potentially increases the requirements for rendering performance. In order to present a realistic view of the environment suitable for interactive applications, an acceptable rendering frame rate must be achieved. The number of polygons that can be rendered per second is the limiting factor for the achievable frame rate.

6.2.2 Analysis of existing systems

The following analysis further underlines that scalability needs to be addressed by all DVE systems and provides a reference for the discussion of the scalability approach presented in this work. With respect to scalability, two different classes of DVE systems are considered. Systems like DIVE or Repo3D that do not include any mechanism to handle scalability and systems like NPSNET Terrain or SPLINE that use tiling to address scalability issues.

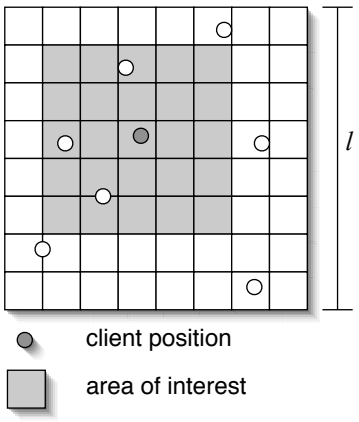

Fig. 6.2: The environment is subdivided into smaller tiles. Only tiles within the area defined by the maximum visibility are needed for rendering.

No scalability mechanism

Without support for scalability, each client maintains a copy of the entire environment. Let l_{env} be the side length of the environment, d_{poly} the average polygon density of the environment and m_{poly} the amount of memory needed to store one polygon description. The amount of process memory necessary to store the environment representation C_{memory} is approximated as

$$C_{memory} = l_{env}^2 \times d_{poly} \times m_{poly} \tag{6.1}$$

Let n_{client} be the number of participating clients, $n_{changes}$ the number of changes that one client applies to the environment for each new view, m_{msg} the average size of a single update message and r_{view} the average number of views that are generated per second. The total bandwidth C_{bwidth} required by each client to send and receive all state update messages is defined as

$$C_{bwidth} = n_{client} \times n_{changes} \times m_{msg} \times r_{view} \tag{6.2}$$

The number of polygons that a client needs to render per second is defined as

$$C_{render} = l_{env}^2 \times d_{poly} \times r_{view} \tag{6.3}$$

Tiling for scalability

The environment is tiled and each client defines a limited area of interest that is significantly smaller than the entire environment. Only the tiles that belong to that area around each client is considered to be interesting (see

Asymptotic Complexity		
N	No Support	Tiles
	$C_{memory}(N)$	
l_{env}	$O(N^2)$	$O(1)$
l_{vis}		$O(N^2)$
d_{poly}	$O(N)$	$O(N)$
	$C_{render}(N)$	
l_{env}	$O(N^2)$	$O(1)$
l_{vis}		$O(N^2)$
d_{poly}	$O(N)$	$O(N)$
	$C_{bwidth}(N)$	
n_{client}	$O(N)$	$O(1)$

Tab. 6.1: The asymtotic complexity of the cost functions depending on the considered scalability parameters.

figure 6.2). Let l_{vis} be the desired range of visibility that defines the area of interest for each client. The amount of process memory necessary to represent the area of interest C_{mem} is then defined as

$$C_{memory} = 4l_{vis}^2 \times d_{poly} \times m_{poly} \tag{6.4}$$

C_{memory} does no longer depend on the size of the entire environment, but depends on the size of the area of interest.

Likewise, only update messages for environment modifications inside the area of interest need to be received by a client. Let d_{client} be the average client density with respect to the environment size. The total bandwidth required by each client to send and receive all state update messages relevant to the area of interest is defined as

$$C_{bwidth} = 4l_{vis}^2 \times d_{client} \times n_{changes} \times m_{msg} \times r_{view} \tag{6.5}$$

Analog to equation 6.1 the number of polygons a client needs to render is defined as

$$C_{render} = 4l_{vis}^2 \times d_{poly} \times r_{view} \tag{6.6}$$

6.2.3 Comparison of tiling vs. no scalability mechanism

To compare the characteristics of the two approaches it is helpful to look at the asymptotic complexity of the cost functions for the three scalable parameters (see table 6.1).

It is immediately obvious that the tiling strategy provides major scalability improvements. It effectively decouples client memory and rendering

requirements from environment size. The complexity of the corresponding cost functions is significantly reduced from $O(N^2)$ to $O(N)$. Further, the linear dependency of the process bandwidth on the total number of processes is removed.

The tiling approach limits visibility and removes the dependency on overall environment size. Similar improvements can be seen for the dependency of the bandwidth required for a single process on the total number of participating processes.

However, tiling does not address two important scalability issues. First, there is still a quadratic increase of memory and rendering requirements, now depending on the visibility parameter l_{vis} that determines the size of the area of interest. Second, the linear dependency of both requirements on the average polygon density d_{poly} is not handled by the tiling approach. Section 6.6 shows that the hierarchical distribution approach to scalability resolves both issues.

6.3 Exploiting visibility for rendering

This section surveys basic visibility techniques that are used to increase rendering performance for large scene in real-time graphics applications.

Normally, rendering are performed for the entire geometry that is part of the scene graph. For large scenes this can consume considerable processing resources and generally results in low frame rates. Visibility based acceleration techniques rely on the assumption that in many cases only a small portion of the entire scene is visible in the final rendering. Thus, a lot of processing an rendering resources are wasted on parts of the scene that are not even visible in the final rendering. By predicting which parts of the scene graph will visible for a particular viewpoint and viewing direction and then rendering only those parts, the whole process of rendering would be accelerated.

View frustum culling and *level-of-detail evaluation* are techniques that classify all parts of the scene graph as either potentially visible or definitely invisible prior to rendering a frame. They are introduced in more detail in the following sections.

6.3.1 View frustum culling in visual rendering

The projection parameters can be represented as a *viewing frustum* that is positioned and oriented in world space coordinates. All geometry that is contained inside the viewing frustum is potentially visible on the screen for that particular projection. Geometry outside the frustum is not visible.

View frustum culling takes advantage of the hierarchical structure of the scene graph to efficiently decide which part of the scene is visible and has to be rendered. All nodes in the scene graph are extended with an

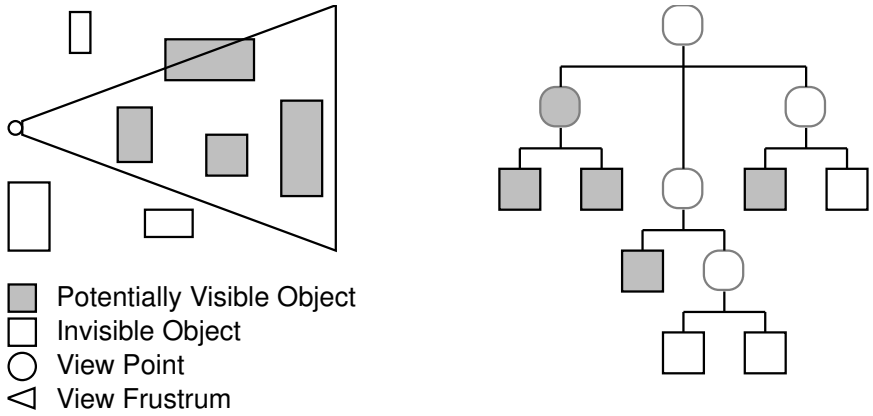

Fig. 6.3: Only object that intersect the viewing frustum are considered for rendering. The bounding box inclusion hierarchy allows for efficient testing.

additional attribute that describes a simple *bounding volume* of the node. The bounding volume of an inner node completely contains all the geometry that is contained in the subgraph rooted at that inner node. In case of a leaf node, the bounding volume is guaranteed to enclose all the triangles contained in the leaf node. Whenever the structure of the scene graph, a triangle list of a leaf node or a transformation matrix of a transformation node is changed, the bounding box of all nodes that are affected by this change are recomputed.

Prior to the culling and rendering traversal, the viewing frustum is computed from the current viewing parameters. During traversal the viewing frustum is intersected with the bounding volumes of each node. Based on the result of this intersection test, one of three possible actions is taken.

- The bounding volume lies completely outside the viewing frustum. The entire subtree enclosed by the bounding box can not possibly by visible and is *pruned*, i.e. the traversal does not continue into this subtree. The subtree is neither processed nor is the contained geometry rendered.

- The bounding volume lies completely inside the viewing frustum. The entire geometry contained in the subtree is potentially visible (except maybe for occlusion). The rendering traversal is continued into the subtree, but no further bounding box tests are performed.

- The bounding volume and the viewing frustum intersect in some non trivial way. No definite inside/outside decision can be made. The traversal continues normally into the subtree and further bounding volume tests are performed on the children of the node.

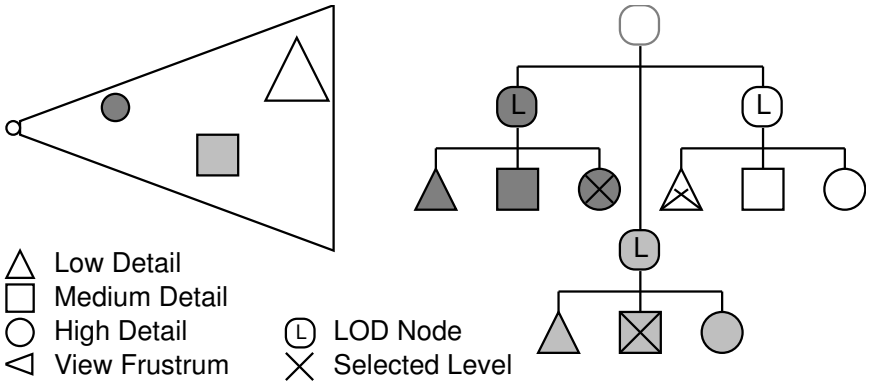

Fig. 6.4: The evaluation of a LOD node selects the appropriate object representation for rendering based on viewer distance.

View-frustum culling can save considerable processing resources during rendering and can speed up the rendering process significantly. However, the effectiveness of view frustum culling depends strongly on the quality of the bounding volume hierarchy. Culling results are best if the structure of the hierarchy closely corresponds to the spatial structure of the scene.

6.3.2 Level-of-detail evaluation

The culling technique described in the previous section classifies objects in a scene by visibility and reduce resource consumption by rendering only the visible objects. Level-of-detail (LOD) evaluation tries to further accelerate the rendering of the visible objects (see also [24]).

Because a perspective projection is used to map objects from world-space to screen-space the projected screen area of an object decreases with increasing distance of the object from the viewer position. The smaller the screen area of an object, the less object detail is visible in the final image. Rendering complex objects at great viewer distances is a waste of resources, because the original object detail is not perceptible in the image. A less complex representation of the same object would be enough to achieve the same perceptible detail, while saving considerable processing and rendering resources.

Level-of-detail evaluation exploits this effect. For each object several different versions with decreasing complexity and triangle count are supplied. For the representation of the various levels of detail for an object a new inner node is introduced to the scene graph. The *LOD node* manages its children as different versions of one and the same object. Each child is associated with a range value that determines which version is to be rendered at what object - viewer distance. During the rendering traversal, the distance between an object and the viewer is calculated and used to select the ob-

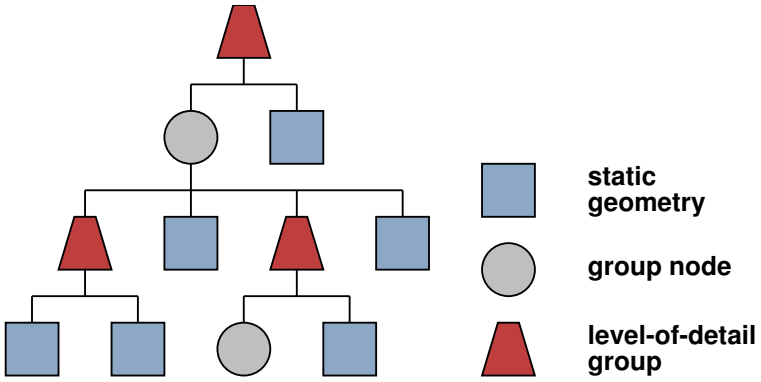

Fig. 6.5: The structure of a hierarchical level-of-detail tree.

ject representation that has the least complexity while still providing a good perception of detail on the screen. Only this version of the object is then rendered. Thus, an object at great distance consists of fewer polygons and consume less rendering resources than the same object at a closer distance.

Level-of-detail evaluation performs best if the objects in the scene are consistently modeled with multiple levels of complexity. This necessity increases the modeling effort because several different versions of each object have to be build. Automatic level-of-detail generation is a difficult problem to solve for general objects. For specific classes of objects however, usable reduction algorithms exist[40]. Level-of-detail evaluation is very suitable to accompany view frustum culling and/or occlusion culling.

6.3.3 Hierarchical level-of-detail rendering

Hierarchical level-of-detail is a variation of the LOD evaluation scheme described in the previous section. The original LOD scheme provides different levels of details for simple objects. The hierarchical LOD scheme provides different levels of detail for entire subtrees that, in turn, contain LOD nodes themselves. Figure 6.5 shows a schematic view of a hierarchical LOD scene graph.

Each LOD node has only two children, the second child for the coarse representation of that part of the scene graph, the first child for the more detailed representation. The coarse representation is a normal geometry node, while the fine representation is a complete subtree that contains further LOD nodes as its children. Thus the hierarchical LOD scheme can be used to structure the scene graph in a way that resembles adaptive multiresolution space partitioning schemes like a quad-tree or an octree. Figure 6.6 illustrates how a LOD hierarchy interacts with the viewing frustum during rendering. The example is a typical scenario that can be found in terrain based simulators.

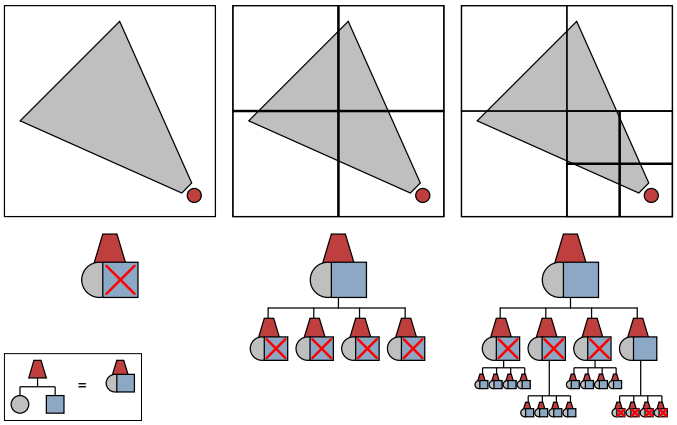

Fig. 6.6: Due to the hierarchical LOD structure of the scene graph, those parts of the surface that are close to the viewer are rendered at full resolution, while parts further away from the viewer are rendered at lower resolution.

To provide more visual and conceptual clarity in our figures, we introduce a new inner node to the scene graph. This new HLOD node is a combination of three other nodes:

- A standard LOD node with exactly two children that represent the same area of space in two different levels of detail, the *coarse* representation and the *fine* representation. The LOD node selects the appropriate child node depending on its range attribute and the distance to the viewer position. It is symbolized by a trapeze.
- A standard group node whose children can be in turn HLOD nodes. The subtree rooted at this group node is the fine representation of the HLOD node. It is symbolized by a circle.
- A standard geometry node that holds the geometric description of the fine representation of the HLOD node. It is represented by a square.

The scene graph in figure 6.6 represents a surface patch on which the viewer is located. In figure 6.6(a) the entire patch is represented by one geometric object. In this case the entire surface patch has to be rendered at full resolution to produce the image for the viewer. In figure 6.6(b) the surface patch has been subdivided into four single patches that are located on level two of a LOD hierarchy. Level one represents the same surface area with only one surface patch at a quarter the resolution using only a quarter of the triangles. This subdivision step is reapplied in figure 6.6(c) and produces a multi-resolution representation of the surface that is three levels deep. During the rendering traversal the LOD nodes in the tree are evaluated. Due to the hierarchical LOD structure of the scene graph, those

parts of the surface that are close to the viewer are rendered at full resolution, while parts further away from the viewer are rendered at lower resolution.

The hierarchical LOD scheme allows real-time rendering of almost arbitrarily large databases. However, a database must exhibit a strong spatial coherence, the structure of the scene graph must match the spatial structure of the database. This makes the hierarchical LOD scheme perfectly suitable for terrain rendering application. Further, the required multi-resolution hierarchy can be automatically generated from the raw surface data. Popular examples are whole earth visualization systems like TerraVision[61] and T-Vision[4]. Those projects use satellite remote-sensing data to construct and render the entire world in form of a globe. The user can freely navigate above the globes surface, while the hierarchical LOD evaluation imperceptibly elides all unnecessary detail from the rendered scene and guarantees real-time rendering frame rates.

6.4 Exploiting visibility for distribution

This section introduces AVOCADOs approach to distribution in large scale virtual environments. Visibility based culling and LOD schemes, which are very similar to those described in the previous section, are used to reduce the amount of communication necessary for state transfer and update, and to implement a hierarchical partitioning scheme that overcomes the scalability problems described at the beginning of this chapter.

The main assumption is that in a large scale virtual environment the area of interest of a participating process can be based on visibility. A process is assumed to be only interested in parts of the scene that are visible for the current viewing parameters. For large scale VEs this may be only a small portion of the entire scene. Thus it is not necessary for the process to have a complete copy of the scene to render the current view. A dynamic *working set*, the subset of visible objects in the scene, is enough. Further, because parts of the scene outside the working set are of no interest, the process does not need to receive state transfer and update messages for objects from outside the working set. However, the working set needs to be constantly updated according to changing viewing parameters. New parts of the scene are added if they become visible, while old parts are released from the working set if they become invisible. In addition to changing viewing parameters, moving geometry in the scene may also affect the current working set. If new geometry becomes visible it is added to the working set and geometry that becomes invisible is removed accordingly.

The following sections explain how a scene is structured into hierarchical partitions that can be added and removed from the working set of a participating process while maintaining overall consistency of the globally shared application state. A method that uses visibility information that is freely available from the rendering traversal to calculate which partitions need to

be included in the current working set is presented. Further, the distribution concepts introduced in 5 are used to to map the hierarchical partition of the scene to a set of distribution groups. The use of multi-cast based distribution groups enables the construction of large-scale VE applications that can be efficiently operated over wide area networks using the Internet Mbone infrastructure.

A prototypical implementation of this approach is based entirely on the work introduced in chapter 5. Details of the implementation are explained and a short, proof-of-concept example is given.

6.4.1 Hierarchical environment partitioning

The hierarchical LOD scheme described in Section 6.3.3 enables real-time rendering of very large scenes based on visibility and distance considerations. The hierarchical multi-resolution structure of the scene graph suggests itself as a good starting point for the necessary partitioning of the scene. It has two important properties that can be utilized to efficiently calculate and maintain a working set of visible and thus important objects.

- Strong spatial coherence. The scene graph structure directly corresponds to the spatial position of objects in the scene. This enables the efficient use of view-frustum culling techniques to select potentially visible partitions for inclusion in the working set.

- Multi-resolution information. Objects and groups of objects exist in different levels of detail. The calculation of the working set can not only be based on potential visibility but also on distance. Partitions of the scene that are located at a greater distance can be included at a low resolution version, while partitions that are closer to the viewer can be included at a higher resolution.

Further, the hierarchical LOD scheme can also be used to accelerate rendering without any additional effort because AVOCADO fully supports Performers LOD rendering capabilities and thus also hierarchical LOD evaluation during rendering.

Figure 6.7(a) illustrates how the hierarchically structured scene graph is divided into hierarchical partitions. It shows a scene graph that structurally resembles a BSP tree. The HLOD nodes introduced in section 6.3.3 are used to represent the scene graph structure. Each HLOD has two children, that are in turn HLODs and together represent the same geometry as the parent node at a finer level of detail. Each HLOD defines a partition of the scene, that consists of the coarse geometry contained in the HLOD node and the references to the partitions that are defined by the child HLOD nodes. The geometry in the child HLOD nodes does not belong to the partition formed by the parent node. This allows the selection of a partition at a coarse level

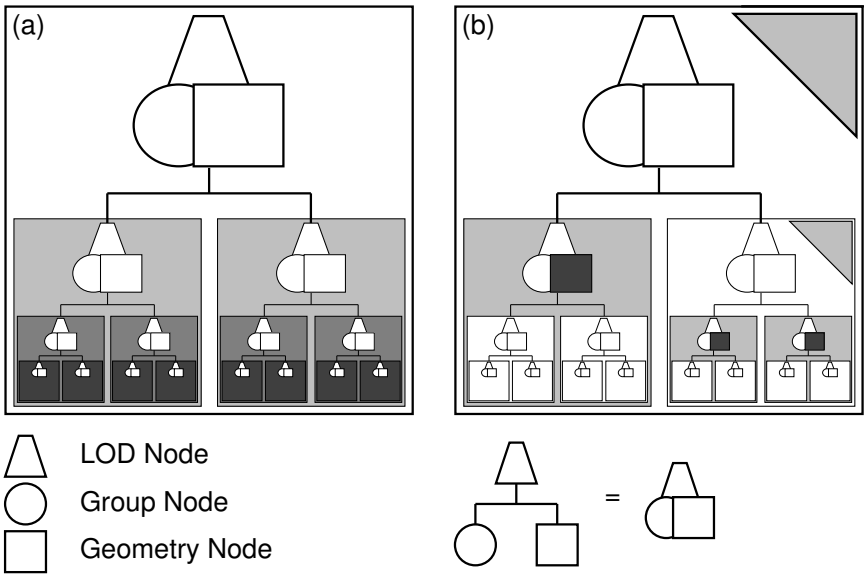

Fig. 6.7: Figure (a) illustrates how a hierarchical LOD scene graph is subdivided into hierarchical partitions. Figure (b) shows an example of a possible working set. The gray partitions are the selected working set partitions. The parent partitions are marked with a symbol. They also belong to the working set, but are currently not active. The marked HLOD geometry is the only geometry that is rendered for this working set.

of detail without automatically including the finer versions of that partition as well.

Figure 6.7(b) shows a selection of three partitions that form a possible working set (selected partitions are gray). The selected working set is a complete medium to low resolution representation of the entire scene graph, and it contains only about 20% of the geometry found in the scene graph.

The hierarchical scene graph partitioning scheme introduced in this section allows the selection of a working subset of the scene based on visibility information. This scheme is now extended to the distributed case such that every participating process can define its own working set and receive and send state update information only for those parts of the scene contained in its working set.

6.4.2 Mapping distribution groups to scene partitions

The distribution mechanism as described in Section 5.1.2 uses the concept of a distribution group to provide a consistently replicated copy of the entire scene graph to all participating processes in a distributed application. The scalability problems inherent in this approach can now be overcome by using

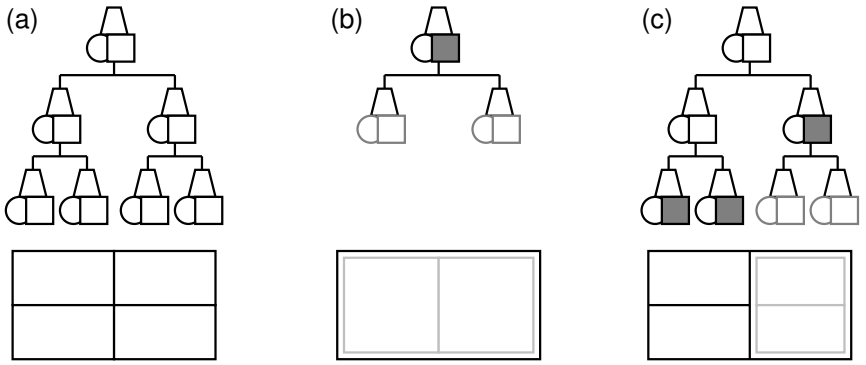

Fig. 6.8: Figure (a) shows a scene graph that is created by a process of a distributed application. It is divided into seven partitions. Each partitions HLOD node is associated with an own distribution group. Figure (b) shows the scene graph of another process that has joined only the distribution group associated with the root HLOD node of the scene graph. In Figure (c) this process has selected a more detailed representation of the scene graph by joining additional distribution groups.

more than one distribution group to distribute the scene graph.

A one-to-one mapping between scene graph partitions and distribution groups is established. To achieve this, the HLOD node of each partition of the scene graph is associated with its own distribution group. This enables processes in the distributed application to precisely select only those partitions they are interested in, by joining only the distribution groups associated with those partitions. Because all communication in a distribution group is local to the members in this group, processes only receive messages that concern their current working set of partitions. Thus, the number of update messages received by a process does no longer grow with the absolute size of the entire scene, but now only depends on the size of the current working set of the process. This allows the creation of shared scenes of potentially unlimited size because the size of the working set does not depend on the size of the entire scene.

The HLOD node introduced in 6.3.3 has to be extended to provide this additional functionality. A new *groupname* attribute identifies the distribution group that is associated with the HLOD node. The associated distribution group contains the geometry node that describes the coarse representation of the HLOD node and the child nodes which describe the fine representation. The HLOD node can now be in one of two different states:

Inactive: The distribution group specified in the groupname attribute is not joined. Thus neither the coarse geometry nor the finer representation is available. The only information available is the bounding volume of

the HLOD node. The scene graph effectively ends here. An inactive HLOD node is denoted by the following symbol:

Active: The distribution group specified in the groupname attribute is joined. This implies that the coarse geometry defined by the node is available for rendering and manipulation. Also available in this distribution group are the child HLOD nodes that define the fine representation of the HLOD node. They may in turn be either active or inactive. An active HLOD node is denoted by the following symbol:

Figure 6.8 illustrates how a process can selectively define its working set by activating the desired HLOD nodes. Sub-figure (a) shows a scene graph that has been created by a process A. It consists of seven HLOD nodes and thus defines seven partitions and their associated distribution groups. All HLOD nodes are active, so that the contents of the partitions is potentially accessible by other processes. The scene graph is organized like a BSP tree and presents a multi-resolution representation of a rectangular surface patch.

In Figure 6.8(b) a second process B, that wants to share the representation of the surface patch has joined the distribution group defined by the root HLOD node. The process receives copies of the coarse representation and the two child HLOD nodes. These are by default not activated, so that at this point only the least detailed representation of the surface patch is available to process B. If process B needs a more detailed representation, it can activate the child HLOD nodes and receive more detailed partitions of the surface patch as shown in Figure 6.8(c). This can be recursively continued until either the detail requirements are satisfied or the most detailed level of partitions has been reached.

By associating scene graph partitions with distribution groups it becomes possible to selectively share only parts of the scene graph between processes in a distributed application. Additionally, because the partitions are structured as a multi-resolution hierarchy, partitions can be selected at different levels of detail. The following section describes how a process can utilize the visibility information available from the rendering traversal to decide which partitions need to belong to the working set and how the HLOD nodes are evaluated to achieve this.

6.4.3 HLOD evaluation and the working set

As described in Section 6.3.3, HLOD evaluation is performed during rendering traversal. Depending on the distance between the HLOD node and the viewer, the traverser decides whether the coarse representation is rendered or the traversal is continued into the child nodes which reveal the finer representation. This is applied recursively until either the detail requirements are met or the finest available resolution is reached.

Because the scene partitioning scheme directly corresponds to the HLOD structure of the scene graph, the definition of the working set can now be refined in terms of the HLOD evaluation results from the rendering traversal. The working set shall be defined to consist of all scene partitions, whose corresponding HLOD nodes have been traversed during the last rendering traversal. This criterion implies that the working set always contains exactly those partitions of the scene graph that are needed to render the current view.

The traversal starts at the root HLOD of the scene graph. This node is assumed to exist and be active, i.e. the process has already joined the associated distribution group. Based on the range attribute, the viewer distance and the activation state of the node, one of two possible actions has to be taken during traversal:

Distance > Range: The coarse representation is selected. The contained geometry is traversed and rendered. Possible active children of the HLOD node are inactivated and not regarded for traversal. They are effectively removed from the working set.

Distance <= Range: The fine representation is selected. The geometry described by the coarse representation is not rendered. Possible inactive children of the HLOD node are activated. This initiates the transfer of the coarse and fine representation of the node by joining the associated distribution group. The scene partitions described by the child HLOD nodes now belong to the working set. The traversal continues into the possible HLOD children, that describe the fine representation of the node.

The selection of the working set is based on viewer distance. Because HLOD evaluation is automatically performed as part of the rendering traversal, working set determination is computationally inexpensive and almost automatic. The working set is guaranteed to contain all scene partitions at only the minimal level of detail that is required for rendering of the current view.

6.4.4 HLOD evaluation and view frustum culling combined

The working set can be further minimized if not only distance is considered as the criterion for HLOD evaluation, but also visibility. As described in section 6.3.1 view frustum culling is performed during rendering traversal. Only those parts of the scene graph that are contained in the view frustum or intersect with it are traversed. Subtrees that lie outside the frustum are not traversed and, as a consequences, not rendered.

This also true for HLOD nodes. If view frustum culling is performed, HLOD nodes outside the frustum are never rendered and thus never evalu-

ated. Because HLOD evaluation is automatically performed as part of the rendering traversal, the working set is automatically reduced to contain only those nodes that survive culling against the current view frustum.

6.4.5 Caching of inactive nodes

As detailed above, the view frustum defines the current working set. If the viewing position or orientation is changed, the frustum changes accordingly. This in turn potentially modifies the working set and as a result some HLOD nodes may be added to the working set and be activated while some old ones are removed and deactivated.

Each activation or deactivation of a HLOD node means that the process joins or leaves the corresponding distribution group. During a join operation, a state transfer is performed and at least the joining HLOD node and it's internal representation have to be transmitted over the network. In case of a fast or frequently moving viewer, the number of group membership changes that a process has to perform in a short period of time is likely to exceed the available resources to do so. Another effect, called *thrashing*, can often be observed if a particular HLOD is repeatedly, and thus unnecessarily, activated and deactivated.

To prevent resource saturation and thrashing a caching mechanism is introduced that delays the deactivation of HLOD nodes that are removed from the working set. These nodes are not rendered because they are outside the viewing frustum, but they are still active members of their respective distribution groups. If such a cached HLOD node is again promoted into the working set, no further operations are necessary because the node is still active.

The cache size can be measured in many different ways, the determination of the optimal one is certainly application dependent. One possibility is the size of the memory occupied by the cache. If a predefined amount of memory is exhausted, a cache clean-out is performed prior the activation of new HLOD nodes. Other cache size criteria could be the total number of cached HLOD nodes.

The optimal cache clean-out strategy is most likely also application dependent. Possible strategies include the cache classic LRU (Least Recently Used) or could be based on the age, i.e. the amount of time an HLOD has been active, or the amount of time of time spent in the working set.

Because caching prevents thrashing and reduces the resource consumption incurred by group membership changes, it is a necessary mechanism that enables efficient implementations of the HLOD distribution scheme.

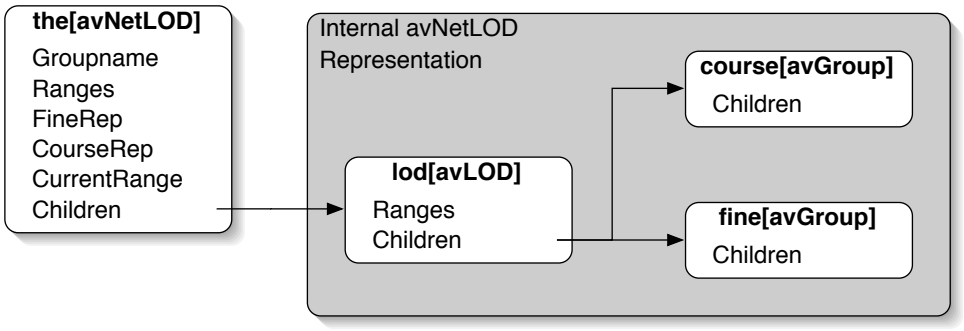

Fig. 6.9: The `avNetLOD` is derived from a standard `avNetDCS` node and is implemented using an internal representation build from distributable AVOCADO nodes.

6.5 Implementation of the HLOD node

The HLOD node, as the single component that allows a scene-graph to be hierarchically shared over an network, is implemented as a compound node based on the `avNetDCS` node (see figure 6.9).

It has the following fields, some of which are inherited from the standard `avNetDCS` class:

Groupname: When the group name is set to an non-empty string, the appropriate multicast group for that node is joined. The groups IP address is calculated from the group name using a hash function. The internal nodes are distributed in the newly joined group along with all their children.

Ranges: This is a vector of three floating point values that determine at which distance from the viewer this particular `avNetLOD` node switches between it's fine and it's coarse representation. Depending on the distance, four different cases can occur:

Distance d	**Representation**
$d < Ranges[0]$	none
$Ranges[0] < d < Ranges[1]$	fine
$Ranges[1] < d < Ranges[2]$	coarse
$Ranges[2] < d$	none

CourseRep: A single-field of type `avNode`. The subtree that is attached here defines the coarse representation of the node. Internally it is re-parented to the *coarse* group node.

FineRep: A multi-field of type `avNetLOD`. The nodes that are attached here define the fine representation of the node. Internally the nodes are

re-parented to the *fine* group node. The coarse representation always consists entirely of other `avNetLOD` nodes.

The decision whether the course or the fine representation of the node should be rendered, is based on Performers built in level-of-detail mechanism. The HLOD node is implemented as a compound node that internally uses a standard `pfLOD` to automatically perform the range evaluation as part of the CULL traversal.

The implementation of the HLOD node uses the standard AVOCADO distribution mechanisms that have been described in chapter 5. Although it exhibits a rather complex functionality, it has been implemented in only a few hundred lines of C++ code. Therefore, besides adding the ability to efficiently describe virtual environments in a multi-resolution hierarchy, it can be regarded as a successful demonstration of the extensibility and versatility of the AVOCADO distributed object model.

6.6 Scalability analysis of the hierarchical distribution approach

Similar to the tiling approach analyzed in section 6.2.1, each client defines a visibility based area of interest that is significantly smaller than the entire environment. Only the polygons contained in that area are considered for rendering. Again, the environment is tiled, but this time the tiles are structured in a hierarchical multi resolution hierarchy. Only tiles that lie within the area of interest are considered and the level-of detail for each tile decreases with viewer distance (see figure 6.10). To compare the proposed approach to the previously analyzed methods, the cost functions C_{memory}, C_{bwidth} and C_{render} are defined for the hierarchical approach.

Let l_{vis} be the desired range of visibility that defines the area of interest for each client. Further, let $n_{tilepoly}$ be the average number of polygons contained in a tile of the highest level-of-detail. The total number of high-detail tiles contained in the area of interest n_{hres} is then defined as

$$n_{hres} = (4l_{vis}^2 \times d_{poly})/n_{tilepoly} \tag{6.7}$$

The number of levels needed to build the aggregating level-of-detail hierarchy n_{lods} is then

$$n_{lods} = \log_4 n_{hres} \tag{6.8}$$

The amount of process memory necessary to represent the area of interest C_{memory} directly relates to the number of polygons needed to represent the are of interest. Because of the aggregating method of tile construction, the number of polygon per tile $n_{tilepoly}$ is the same for the tiles from all levels of detail. Thus, the problem is reduced to the determination of the number of tiles needed to cover the area of interest. Looking at Figure 6.10 this number can easily be specified.

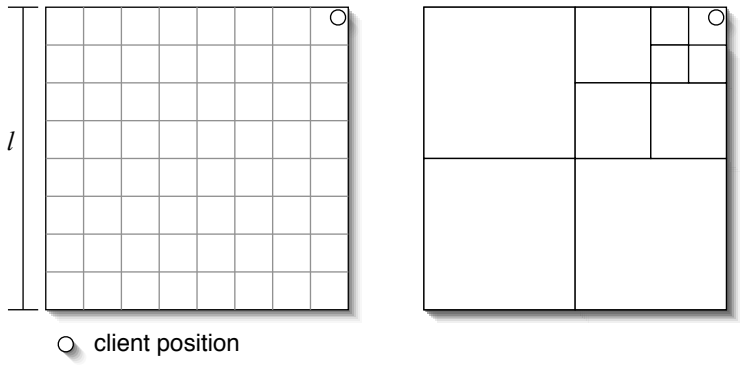

Fig. 6.10: The size of the tiles that cover the plane increases with distance form the client position, while the number of polygons per tile remains constant.

For simplicity, the calculation is done for only one quadrant. From the highest level of detail three tiles are taken to cover three quarters of the are of interest. The remaining quarter is filled from the next higher level of detail following the same method. Thus, the total number of tiles used to cover the area of interest is defined as

$$\begin{aligned} n_{aoitiles} &= 4((4-1) \times (n_{lods} - 1) + 1) \\ &= 12n_{lods} - 8 \end{aligned} \tag{6.9}$$

Using equations 6.7, 6.8 and 6.9, C_{memory} is defined as

$$\begin{aligned} C_{memory} &= n_{aoitiles} \times n_{tilepoly} \times m_{poly} \\ &= (12n_{lods} - 8) \times n_{tilepoly} \times m_{poly} \\ &= (12 \log_4 n_{hres} - 8) \times n_{tilepoly} \times m_{poly} \\ &= (12 \log_4((4l_{vis}^2 \times d_{poly})/n_{tilepoly}) - 8) \\ &\quad \times n_{tilepoly} \times m_{poly} \end{aligned} \tag{6.10}$$

C_{memory} does not depend on the size of the environment, but depends on the size of the area of interest.

As described for in section 6.2.2, only update messages for environment modifications inside the area of interest need to be received by a client. Let d_{client} be the average client density with respect to the environment size. The total bandwidth required by each client to send and receive all state update messages relevant to the area of interest is defined as

$$C_{bwidth} = 4l_{vis}^2 \times d_{client} \times n_{changes} \times m_{msg} \times r_{view} \tag{6.11}$$

Asymptotic Complexity			
N	No Support	Tiles	Hierarchy
	$C_{memory}(N)$		
l_{env}	$O(N^2)$	$O(1)$	$O(1)$
l_{vis}		$O(N^2)$	$O(\log N)$
d_{poly}	$O(N)$	$O(N)$	$O(\log N)$
	$C_{render}(N)$		
l_{env}	$O(N^2)$	$O(1)$	$O(1)$
l_{vis}		$O(N^2)$	$O(\log N)$
d_{poly}	$O(N)$	$O(N)$	$O(\log N)$
	$C_{bwidth}(N)$		
n_{client}	$O(N)$	$O(1)$	$O(1)$

Tab. 6.2: The asymtotic complexity of the cost functions depending on the considered scalability parameters.

Analog to equation 6.10 the number of polygons a client needs to render per second is defined as

$$C_{render} = (12\log_4((4l_{vis}^2 \times d_{poly})/n_{tilepoly}) - 8) \times d_{poly} \times r_{view} \quad (6.12)$$

Comparison

To compare the characteristics of the different strategies to handle scalability, it is helpful to look at the asymptotic complexity of the cost functions for the three scalable parameters (see table 6.2).

As mentioned in section 6.2.1, tiling decouples client memory consumption and rendering requirements from the environment size. While this increases scalability by reducing the costs from $O(N^2)$ to $O(N)$, it does so by shifting the cost dependency from environment size to the visibility range. If visibility is increased, resource consumption still increases quadratically.

This problem is addressed by the proposed multi-resolution approach. The dependency on the visibility range is reduced from $O(N^2)$ to $O(\log N)$. This is a significant decrease in resource requirements. Compared to tiling, it allows a much large visibility range at considerably less costs.

Further, tiling does not address the $O(N)$ dependency on polygon density. Increasing the polygon density still leads to a linear increase of resource requirements.

The multi-resolution approach reduces the cost complexity for memory and rendering requirements depending on polygon density from $O(N)$ to $O(\log N)$. This means that the distributed virtual environment becomes

scalable with the complexity of the environment representation. Additionally, the environment complexity is dynamically adjustable for each client and allows adaption to the clients rendering capabilities, by trading visual complexity against resource requirements.

In conclusion, the hierarchal multi-resolution approach dramatically improves scalability by reducing the dependency between resource requirements and environment size and complexity from $O(N)$ to $O(\log N)$

6.7 Summary

The hierarchical distribution strategy described in this work provides scalability with respect to the size and geometric complexity of a virtual environment and the number of users that simultaneously participate in it. It is based on well known optimization methods from the area of real-time rendering.

A multi-resolution hierarchy describes the environment at different geometric levels of detail. To achieve scalability, not the entire environment is shared between the processes, but each process subscribes only to a fragment of the scene graph based on it's processing capabilities and it's viewing parameters. The segmentation of the scene graph is marked with newly introduced hierarchical level-of-detail nodes that each associate a fragment of the multi-resolution scene graph with a separate multicast communication group.

This solution compares favorably to existing general-purpose scalability approaches. Hierarchical partitioning is clearly superior to flat tiling methods, as it reduces the resource complexity for increased visibility from $O(N^2)$ to $O(N)$.

7. Results, applications and future work

7.1 Results of this work

Immersive virtual environments and, in particular, distributed immersive environments are becoming an increasingly popular research area. However, building VE applications is a complex task and requires the combination of many different technologies from various research areas.

The goal of this work was to develop a design and an implementation for a distributed VE framework, that can serve as a basis for VE research and for application development. Special attention was payed to the usability of the distributed object and event model on one hand, and competitive performance on high-end systems on the other. As the main contributions of this work an architecture for a general purpose DVE framework has been developed and implemented (Chapters 3, 4 and 5) and a novel approach to scalability for large-scale distributed virtual environment frameworks (Chapter 6) has been presented.

The results of this work do not only apply to the described AVOCADO architecture and implementation, but are generally applicable to the design of scalable distributed virtual environment frameworks. They can be summarized as follows:

- The combination of a comprehensive and flexible distributed object model with consistent APIs and scalable performance is both, highly desirable and possible. Many VE toolkits and frameworks focus on one aspect while totally neglecting the other. The described architecture demonstrates the synthesis of a fully developed object and event model and scalable performance in one system.

- The distributed object and event model provides a transparently shared scene graph to the application developer. The API used to handle objects and scene graph is almost identical to the stand-alone APIs. In contrast to other approaches, that introduce new concepts to handle distributed event delivery, the presented architecture tightly integrates event handling and distributed object notification into a unique and unobtrusive distributed event model.

- Application state replication is firmly based on the process group model. Unlike most other approaches, AVOCADO explicitly sup-

ports highly dynamic distributed applications with frequent and unannounced membership changes, and at the same time provides very strong consistency guarantees.

- To ensure scalability with respect to environment size and complexity and the number of processes, a novel hierarchical multi-resolution approach has been presented. It is based on the level-of-detail concept well-known from visual rendering. The use of visibility as a selection criterion for required object detail conforms with user expectations. Hierarchical distribution maps well to the group communication paradigm used in AVOCADO, and solves the problem of heterogeneous distribution groups that consist of peers with differing processing and rendering capabilities.
- A distributed locking facility is integrated into the distributed object model and communication protocol and provides the essential functionality that is needed to implement race free application-level synchronization across processes in distributed applications.
- The framework architecture in combination with the extension APIs and the dynamic loading capabilities has proved to be a solid and highly customizable platform for the development of application specific extensions. The rapid prototyping style of application development that is encouraged by the comprehensive scripting interface has been well received by application developers.
- Direct user interaction with objects in the virtual environment is formalized in a tool-based interaction framework.
- A display device abstraction allows the adaption to all common immersive multi-screen displays. The display components support head-tracking and stereo output, and use all available hardware options to optimize the rendering performance. The explicit representation of the display model abstractions as part of the application scene graph has not been described before, and allows the description of complex, dynamic display setups with a small number of primitives.

The results of this work have been validated by numerous successful research projects and applications that use AVOCADO as a basis. The following section presents a representative selection.

7.2 Applications built with Avocado

7.2.1 Multi-modal interaction in virtual reality

It's rapid prototyping and scripting abilities qualify AVOCADO as a good foundation for the prototypical implementation of VR related research re-

sults.

The Artificial Intelligence Group at the University of Bielefeld works on a demonstration platform for virtual-reality-based prototyping using gesture and speech. They operate a three-sided CAVE using a PC cluster and the AVOCADO framework to develop an integrated framework for recognition, interpretation, and rendition of gesture and speech based interactions in VR applications.

In this context, Latoschik[49, 50] uses the AVOCADO framework as a basis for the prototypical implementation of his work on the combination of gesture and voice recognition based methods for user interaction. His system recognizes voice and gesture articulations of the user and combines them to deduct the users intentions with regard to the virtual environment. It then modifies the scene graph accordingly. The prototype allows the user to assemble a vehicle from a number of parts using a combination of voice commands and gestures. A typical compound utterance of the user would be:

> "Pick up this wheel *[Simultaneous pointing gesture at the referenced part]* and attach it to the left end of the red axle."

Gesture and voice recognition is performed by a network of interconnected modules (see figure 7.1). The modules perform subtasks of the recognition process and communicate with each other to reach a conclusion. The modules are implemented as field container nodes that communicate via field connections. Actuator nodes create a stream of data from input devices that is channeled to the appropriate recognizer nodes.

The system demonstrates a powerful use of AVOCADO's field connection mechanism to build an easily configurable and modular recognition system that is tightly integrated into the VR application. The system is configured through the Scheme scripting interface. Reconfiguration is fast and does not require any re-compilation. The use of field connections to implement modular data-driven application components demonstrates the versatility of AVOCADO's data-driven event model.

7.2.2 Oil exploration demonstrator

IMK VE used AVOCADO to build a demonstrator[25] for distributed immersive geo-scientific exploration on the Responsive Workbench. The application allows the distributed visualization of geo-science data. The data sets consist of seismic data and well log data and in its complexity presents a challenge for interactive visualization. From the paper[25]:

Seismic data: "Seismic surveys are carried out by sending acoustic shock waves into the ground where they are reflected and refracted, following the physical principles of wave-motion in layered media. The amplitude and travel time of acoustic waves returning to the surface are

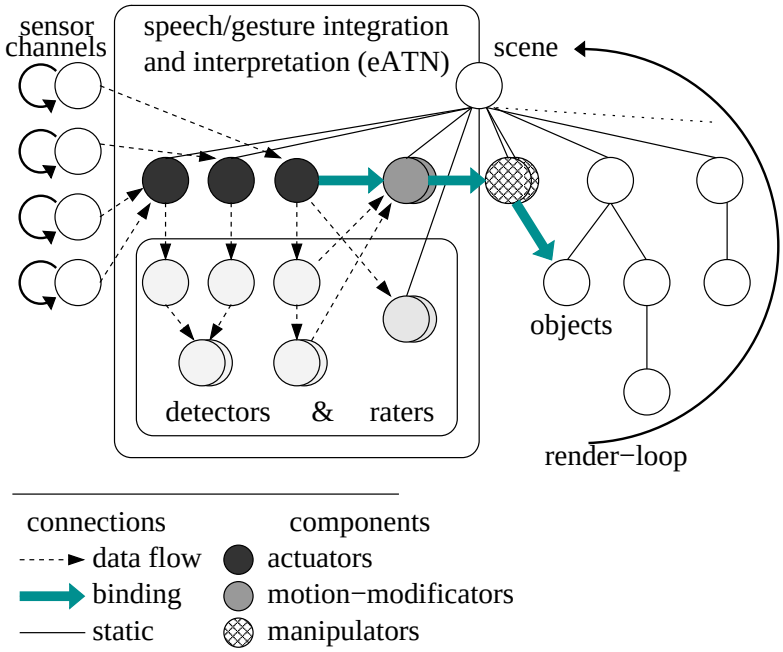

Fig. 7.1: Gesture and voice recognition is performed by a network of interconnected modules. (Figure taken from [48])

measured and processed into regular three-dimensional scalar grids. Strong coherent reflectors and other structures can be analyzed from these data volumes, which represent a block of the earth subsurface that may be kilometers on a side. The seismic cube is the central data structure for most exploration and interpretation tasks. Subsurface structures like horizons and faults are defined relative to the seismic cube and typically displayed as polygonal models."

Well log data. "Well log data is gathered by lowering instruments down an existing drill-hole to measure physical properties such as gamma radiation, neutron density, bulk density, electrical conductivity and many others. Measurements can be made at centimeter interval over hundreds of meters, so well log data is high resolution, dense and multivariate. Subsequent processing can be done to produce vector data and data representing surfaces along the drilling path."

The exploration application allows the user to interactively visualize these volumetric datasets on the Responsive Workbench using a variety of visualization tools, like perpendicular cutting planes, volumetric lenses and various probes for value data extraction.

For the intuitive manipulation of dataset and visualization tools a new input device has been developed. The CubicMouse[27] consists of a cube-

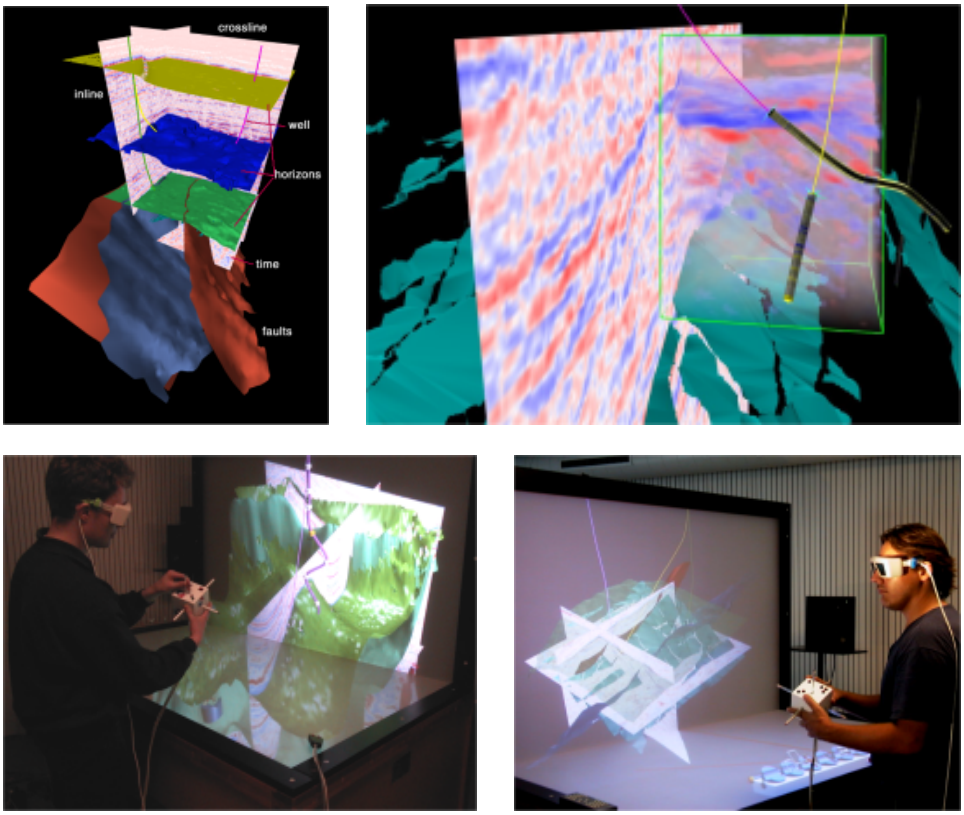

Fig. 7.2: The upper row of screen shots shows typical geo-science datasets and visualization tools. The bottom row of photographs shows users that perform a distributed visualization of a shared dataset on two remote responsive workbenches. Both use a CubicMouse for manipulation.

shaped box with three perpendicular rods passing through the center and buttons on the top for additional control. It controls the position and orientation of the virtual model and the rods move the three orthogonal cutting planes through the dataset.

The application is designed to be used in a distributed scenario, where two users on remote workbenches visualize a shared dataset. The users each use a CubicMouse to control the view on the dataset and the visualization tools.

The implementation of this application uses a number of AVOCADO features and techniques and demonstrates the validity and the versatility of the approach that AVOCADO has taken toward the development of distributed applications. In particular:

- Distribution of the application state, including datasets and visualiza-

tion tools, is performed in one distribution group using a `avNetDCS` node. The potentially very memory consuming dataset nodes are implemented as compound nodes (see section 5.5) to optimize distribution behavior.

- The representation of dataset and tools is implemented as a set of custom nodes that use the AVOCADO extension mechanism and provide the desired functionality.
- User interaction and tool handling is based on the interaction framework presented in section 4.3.
- Display setup for the one and two-sided Workbench is specified using the display device abstraction described in section 4.2.

7.2.3 PC-CAVE render cluster

As mentioned in section 2.4, at IMK VE AVOCADO has also been used to build a PC render cluster that drives stereoscopic multi-display devices like the CyberStage and the CONE.

With the advent of cheap consumer oriented AGP graphics cards for PCs, the desire to replace the high priced SGI graphics supercomputers arose. Unfortunately, the PC bus architecture is not very scalable with respect to the number of CPUs and graphics boards used in one PC. SGI Onyx like setups with 12 processors and 4 graphics subsystems are not yet feasible on a single PC.

The obvious solution is to use a cluster of networked PCs that consists of one or two CPUs and one graphics card each. However, while this approach promises easy scalability by just adding more PCs to the cluster, it also introduces a number of additional problems:

- Dissemination of application state to the cluster PCs. Each PC in a cluster renders a different view of the application state. All information about what to render needs to be transfered to each PC for each frame.
- Because of the loose coupling of networked PCs, synchronization of the output image generation is often necessary to suppress visual discontinuities on moving objects that span several displays. This is especially important in multi-display active stereo setups like the CyberStage or the CONE.

The cluster configuration used at IMK VE uses one master PC that controls four slave PCs. The master handles user input and manipulates application state, while each slave PC renders one view of the application state and provides video output for one display segment. Master and slave

PCs all run AVOCADOThe entire application state is transparently shared between the master and the slaves using AVOCADOs shared scene graph. This setup basically makes the cluster transparent to the application developer and provides good $O(1)$ scalability due to the use of IP multicast at the transport layer.

The cluster is mainly used to drive CONE and CAVE-like displays using active stereo. Therefor synchronization of the render slaves is necessary. The graphics cards directly support video gen-locking while swap-lock is implemented with custom networking hardware that uses the control signal lines of the PC parallel printer ports to provide low-latency synchronization of the frame buffer swapping.

The cluster consists of HP X4000 workstations with two XEON4 processors and a ATI FireGL4 graphics card each. The PCs are networked with using standard giga-bit Ethernet and as switching hub. Each render slave outputs a 96HZ stereo signal at 1600 times 1460 pixels, and is capable of delivering seven million polygons per second. This places the overall cluster performance in the range of multi-million dollar graphics supercomputers like the SGI Onyx InfiniteReality.

Although AVOCADO was never intended to be used as a communication middle-ware for PC based render clusters, nevertheless this installation shows that it is completely up to the task. This can be interpreted as another proof of the versatility of AVOCADOs approach to implement distributed virtual environments.

7.2.4 Caveland

The first application for the CyberStage that was build with AVOCADO is the Caveland production. It was presented to the public as the main attraction on GMD's booth at Cebit 1997 in Hannover. Caveland takes the user on a 10 minute tour through a virtual environment that is assembled from five different worlds (see figure 7.3 for some screen shots from the environment):

Caveland: A fantasy fair ground that serves as the basis for explorations into the other worlds.

Iceland: A frost bitten world that presents a number of stunning visual effects to the CAVE visitor.

Fireland: An underground landscape that explores the borders of visual perception and the fragility of the human sensomotoric system by presenting views into the world from disturbing perspectives.

Sound Spheres: An interactive composition that is controlled by the user through the manipulation of colored spheres floating in space and that is rendered visually as well as audibly.

Fig. 7.3: Screen shots from the 1997 production Caveland that was shown as the main attraction on the GMD booth at Cebit.

The user is guided through these worlds by two animated characters that explain the various attractions in comedic dialog. The characters are build from polygonal meshes that are deformed in real-time during rendering. The characters are animated by performers using traditional character animation techniques like motion capturing. Playback of the captured animation and real-time deformation of the polygonal meshes is performed parallel to rendering on a separate processor and thus, despite being computationally expensive, did not decrease the overall rendering frame rate.

The presentation is structured as a guided tour to allow supervision by untrained personnel during the extensive hours of operation. Caveland was on display for the entire duration of the exhibition. A new show was started every 15 minutes for about 8 hours a day. This results in approximately 256 shows in eight days.

Because of the size and the number of polygons of the virtual environment, the scene is segmented into separate worlds in order to allow rendering at interactive frame rates. The transition between these worlds is implemented using automatic portals that ensure that, while the details of the transition are imperceptible to the user, the system never needs to render two worlds at the same time. Further, the segmentation of the world

allowed several teams to work on the application in parallel and thus helped to reduce development time.

In addition to the visual display provided by four stereoscopic projections the CAVE was equipped with eight independent loudspeakers. The loudspeakers were driven by a dedicated sound server that was controlled by the AVOCADO framework and that was capable of producing spatially localized sound effects for almost any number of sound sources in real-time. To further increase the degree of immersion, the CyberStage floor was fitted with a low frequency resonator that would add tactile feedback to the user experience.

Caveland was developed by a team of about thirteen people over a period of three months. The major part of the development was performed on SGI Indy and Indigo desktop workstations. The final application was deployed on a four-pipe SGI Onyx InfiniteReality supercomputer.

Besides being a complete success as an eye-catcher to the more serious exhibits on display at GMD's Cebit booth, Caveland served as a proof-of-concept for the entire AVOCADO framework design. It successfully demonstrates several key features of the framework:

Multiple displays: Caveland was presented in a four sided CAVE, resulting in eight views being rendered simultaneously. The views were generated in parallel on four InfiniteReality graphics subsystems.

Interaction: The presentation was controlled entirely by the guiding operator from within the CAVE using a combination of specialized stylus and joystick input devices.

Rapid prototyping: The script based rapid-prototyping ability led to a very short development cycle, providing fast and early feedback. Because of the inherently modular extension mechanism, a large team was able to work on the production in parallel with a minimum of interdependencies. Scripting made regression testing of modules and module integration easy.

Extensibility: Several complex and application dependent extensions have been implemented. Most notably these are:

- Vertex based character animation.
- 3d localized surround sound.
- Portal based scene segmentation.
- Presentation control system.

Although these extensions have been developed specifically for Caveland, they are immediately reusable in other application contexts.

Performance and stability: The Caveland presentation shows a highly complex detailed virtual environment at frame rates that exceed 20 HZ.

The presentation was on display for 8 days, running one show every 15 minutes without any major software stability problems.

7.2.5 Commercial applications: Vertigo Systems and RMH

The Vertigo Systems GmbH[77] and the RMH New Media GmbH[62] were founded as GMD spin-offs in 1998. Since then, the two companies have produced several commercial CAVE applications that were showcased on fairs and exhibitions around the world. All productions are based exclusively on the the Avocado framework and draw from the experiences made during the development of the Caveland presentation. Since 1998, the following presentations have been produced (see also figure 7.4 for pictures from selected events):

- *CargoMaster:* A virtual reality game for the the i-CONE. Presented at the Hannover Fair and the China Coal and Mining 2001 in Beijing.
- *Avandia: Insulin Resistance Inside:* An interactive CAVE presentation for GalaxSmithKline that illustrates the mechanism behind a new diabetes drug. Demonstrated in 2000 as part of a road show at over 40 locations in Germany.
- *Siemens Innovation City:* An interactive CAVE presentation for Siemens AG that introduces telecommunication technologies to the visitors. A major attraction on the Siemens booth at the Telecom 1999 exhibition in Geneva.
- *Office interior planning for the German Foreign Minister:* An interactive planning system for interior design. Built in 1999 to assemble and present different versions of his new office to the minister.
- *Hostalen Erlebniswelt:* A virtual product presentation in the CAVE. Produced for the Kunststoffmesse 1998 in Dsseldorf.
- *RAG Erlebnisreise:* A virtual company and product presentation for RAG, the former Ruhrkohle AG. Presented in a CAVE on Hannover Fair 1998.
- *Virtual Anima:* A virtual CAVE based presentation of Andre Heller's new concepts for a theme park. The presentation was given to potential investors and later opened to the public.

The successful work of Vertigo Systems and RMH further stresses the versatility and real-world usability of the AVOCADO framework.

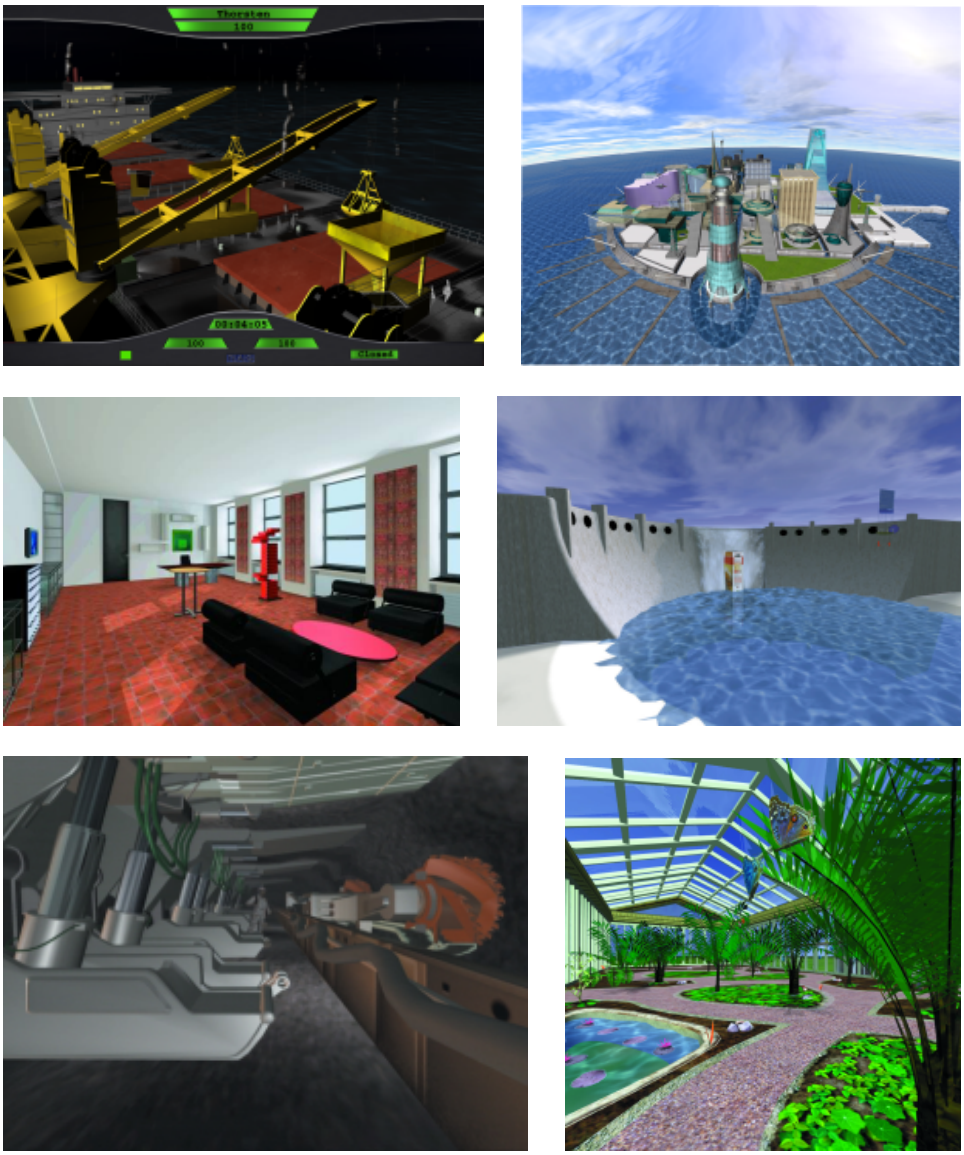

Fig. 7.4: Screen shots from selected commerial productions (CargoMaster, Siemens, Foreign Minister, Hostalen, RAG, Heller).

7.3 Suggestions for future work

Although the AVOCADO architecture and implementation are reasonably complete and have been validated by great number of real-world applications, there are still many areas where further research seems worthwhile:

- The presented approach to distribute object state and event information does not consider timing issues. While it guarantees that the

causality relationship between events is always preserved in a distributed system, it does not develop any provisions to preserve the timing relations between events. Further research is necessary to investigate how the real-time layout of events can be preserved in a distributed VE system without increasing the overall system latency.

- The use of total message ordering to guarantee consistency comes at a price, as it introduces an additional source for communication latency. Depending on the application, consistency requirements for certain state attributes might be less demanding. For example, if the consistency requirements for the spatial position attributes of a fast moving object could be relaxed to a degree where total message ordering would no longer be necessary without damaging overall application state consistency, update latency for the object position could be effectively decreased. Research into this direction would lead into the area of reliable group communication protocols that support dynamic *quality of service* (QoS) parameterization on a per message, or at least a per message class, basis.

- Currently distributed objects are owned exclusively by one process. As a consequence, objects that have been created and shared by one process will never survive that process' termination. As soon as that process dies, the replicated copies of the shared object will also be destroyed. Introducing the concepts of *shared ownership* and *ownership migration* offers an interesting starting point for research into high-availability VE server clusters. Pursuing this direction of research would also lead to a more thorough evaluation of the failure detection and failure recovery mechanisms of the underlying group communication layer.

- All processes that join a hierarchical DVE currently must at least join the top-level root group. The process group model requires each group to maintain a list of all current members. As the number of members increases the resource consumption due to member list maintenance may become a problem. An investigation into dynamic re-rooting of processes in the distribution hierarchy could further elevate the upper limit to the number of participating users.

- Currently, new object classes can only be defined using the C++ APIs. The script layer can instantiate and modify objects, but offers no mechanism to define new classes from script. It would be interesting to research how scriptable class definitions can be integrated into the distributed object model. The current assumption, that all class definitions are compiled and available to all clients, would no longer

hold, as new object classes could be dynamically defined at run-time and would need to be distributed to all processes.

- A closely related topic is the role of event handler scripting in distributed environments. Currently, only Scheme function definitions can be distributed as values between processes. Any global variable bindings contained in the Scheme environment of the transfered function definition are lost. Some first thoughts and a simple solution have already been published in [66]. A more general approach would thoroughly analyze the applicability and the semantic behavior of lexical scoping in replicated distributed environments.

The list of suggestions shows the potential for further research in this area. As results of this work, the described advances in DVE framework architecture and design contribute to the ongoing effort to contain and handle the complexities of the development of distributed virtual environments. Considering the increasing availability of network connectivity in general and the current developments in the area of on-line-gaming in particular, the technology behind distributed virtual environments will be an important research topic for years to come.

Bibliography

[1] Abelson, H., Adams, N. I., Bartley, D., Brooks, G., Clinger, W., Dybvig, R. K., Friedman, D. P., Halstead, R., Hanson, C., Haynes, C. T., Kohlbecker, E., Oxley, D., Pitman, K. M., Rees, J., Rozas, G. J., Sussman, G. J., and Wand, M. Revised[4] Report on the Algorithmic Language Scheme. Technical Memo AIM-848b, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Nov. 1992.

[2] Akeley, K. RealityEngine graphics. *Computer Graphics 27*, Annual Conference Series (1993), 109–116.

[3] Allard, J., Gouranton, L. L. V., Melin, E., and Raffin, B. Net juggler guide. Tech. Rep. RR-LIFO-2001-02, LIFO, Orlans, France, June 2001.

[4] Art+Com. T-vision. http://web.archive.org/web/19970727134454/-www.artcom.de/.

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

[6] Bell, G., Parisi, A., and Pesce, M. The virtual reality modeling language version 1.0 specification, May 1995. http://www.vrml.org/-VRML1.0/vrml10c.html.

[7] Bierbaum, A. *VR Juggler: A Virtual Platform for Virtual Reality Application Development.* Ms thesis, Iowa State University, 2000.

[8] Bierbaum, A., Just, C., Hartling, P., Meinert, K., Baker, A., and Cruz-Neira, C. Vr juggler: A virtual platform for virtual reality application development. In *Proceedings of IEEE VR 2001* (March 2001).

[9] Birman, K. P. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM Computer Science Technical Report, 91-1216 (revised), Cornell University, Jan 1992 36*, 12 (1993), 36–53.

[10] Birman, K. P. *Building Secure and Reliable Network Applications.* Prentice Hall, 1997.

[11] Blanchard, C., Burgess, S., Harvill, Y., Lanier, J., Lasko, A., Oberman, M., and Teitel, M. Reality built for two: A virtual reality tool. In *Computer Graphics (1990 Symposium on Interactive 3D Graphics)* (Mar. 1990), R. Riesenfeld and C. Sequin, Eds., vol. 24, pp. 35–36.

[12] Blinn, J. F. Jim blinn's corner: Nested transformations and blobby man. *IEEE Computer Graphics and Applications 7*, 10 (Oct. 1987), 65–69.

[13] Buck, I., Humphreys, G., and Hanrahan, P. Tracking graphics state for networked rendering. In *Proceedings of the 2000 SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (N. Y., Aug. 21–22 2000), S. N. Spencer, Ed., ACM Press, pp. 87–96.

[14] Carlsson, C., and Hagsand, O. DIVE — A Platform for Multi-user Virtual Environments. *Computers and Graphics 17*, 6 (Nov.–Dec. 1993), 663–669.

[15] Cruz-Neira, C., Sandin, D. J., and DeFanti, T. A. Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE. In *Computer Graphics (SIGGRAPH '93 Proceedings)* (Aug. 1993), J. T. Kajiya, Ed., vol. 27, pp. 135–142.

[16] Cruz-Neira, C., Sandin, D. J., DeFanti, T. A., Kenyon, R. V., and Hart, J. C. The CAVE: Audio visual experience automatic virtual environment. *Comm. of the ACM 35*, 6 (June 1992), 64.

[17] Deering, S. E. RFC 1112: Host extensions for IP multicasting, Aug. 1989.

[18] Dongarra, J., Otto, S. W., Snir, M., and Walker, D. An introduction to the MPI Standard. Technical report CS-95-274, University of Tennessee, Knoxville, Knoxville, TN 37996, USA, Jan. 1995.

[19] Dybvig, R. K. *The Scheme Programming Language: ANSI Scheme*, second ed. P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1996.

[20] Eckel, G., Göbel., M., Hasenbrink, F., Heiden, W., Lechner, U., Tramberend, H., Wesche, G., and Wind, J. Benches and caves. In *Proceedings of the 1st International Immersive Projection Technology Workshop* (Vienna, 1997), H. Bullinger and O. Riedel, Eds., Springer.

[21] Eyre-Todd, R. A. The detection of dangling references in C++ programs. *ACM Letters on Programming Languages and Systems 2*, 4 (Mar. 1993), 127–134.

[22] Fakespace. Boom 3c: A head-coupled display. http://-web.archive.org/web/19990302041751/www.fakespace.com/hcd-boom3c.html.

[23] Fenner, W. RFC 2236: Internet Group Management Protocol, version 2, Nov. 1997.

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

[25] Fröhlich, B., Barrass, S., Zehner, B., Plate, J., and Göbel, M. Exploring geo-scientific data in virtual environments. In *IEEE Visualization '99* (San Francisco, 1999), D. Ebert, M. Gross, and B. Hamann, Eds., IEEE, pp. 169–174.

[26] Fröhlich, B., Grunst, G., Krüger, W., and Wesche, G. The responsive workbench: A virtual working environment for physicians. *Computers in Biology and Medicine 25*, 2 (1995), 301–308.

[27] Fröhlich, B., and Plate, J. The cubic mouse: A new device for three-dimensional input. In *Proceedings of ACM CHI 2000 Conference on Human Factors in Computing Systems* (2000), vol. 1 of *3D Input*, pp. 526–531.

[28] Fröhlich, B., Tramberend, H., Beers, A., Agrawala, M., and Baraff, D. Physically-based manipulation on the responsive workbench. In *Proceedings of the 2000 IEEE Conference on Virtual Reality (VR-00)* (Los Alamitos, CA, Mar. 18–22 2000), S. Feiner and D. Thalmann, Eds., IEEE, pp. 5–12.

[29] Frölich, B., and Tramberend, H. Physikalische simulation in einer virtuellen umgebung. In *VDI Berichte 1435: Prozessketten für die virtuelle Produktentwicklung in verteilter Umgebung* (1998), VDI.

[30] Funkhouser, T. A. RING: A Client-Server System for Multi-User Virtual Environments. In *1995 Symposium on Interactive 3D Graphics* (Apr. 1995), P. Hanrahan and J. Winget, Eds., ACM SIGGRAPH, pp. 85–92. ISBN 0-89791-736-7.

[31] Göbel, M., Barrass, S., Eccles, J., Eckel, G., Hasenbrink, F., Lechner, U., Mostafawy, S., Toth, S., Tramberend, H.,

and Unnützer, P. Digital storytelling - creating interactive illusions with avocado. In *Proceedings of the 9th International Conference on Artificial Reality and Teleexistance* (1999), H. Tachi, Ed.

[32] Gould, D. Double dispatch with an inverted visitor pattern. *C++ Users Journal* (May 1998).

[33] Greenhalgh, C. Dynamic, embodied multicast groups in massive-2. Technical Report NOTTCS-TR-96-8, Department of Computer Science University of Nottingham, 1996.

[34] Greenhalgh, C. Spatial scope and multicast in large virtual environments. Technical Report NOTTCS-TR-96-7, Department of Computer Science University of Nottingham, 1996.

[35] Greenhalgh, C., and Benford, S. MASSIVE: a collaborative virtual environment for teleconferencing. *ACM Transactions on Computer-Human Interaction 2*, 3 (Sept. 1995), 239–261.

[36] Greenhalgh, C., and Benford, S. MASSIVE: A collaborative virtual environment for teleconferencing. *ACM Transactions on Computer-Human Interaction 2*, 3 (1995), 239–261.

[37] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing 22*, 6 (Sept. 1996), 789–828.

[38] Harper, R. W., MacQueen, D. B., and Milner, R. Standard ML. Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, Edinburgh, UK, 1986. Also CSR-209-86.

[39] Hayden, M. The Ensemble System. Technical Report TR98-1662, Cornell University, Jan. 1998.

[40] Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., and Stuetzle, W. Mesh optimization. *Computer Graphics – Proc. SIGGRAPH '93* (Aug. 1993).

[41] Humphreys, G., Buck, I., Eldridge, M., and Hanrahan, P. Distributed rendering for scalable displays. In *SC2000: High Performance Networking and Computing. Dallas Convention Center, Dallas, TX, USA, November 4–10, 2000* (New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2000), ACM, Ed., ACM Press and IEEE Computer Society Press, pp. 60–60.

[42] Humphreys, G., Eldridge, M., Buck, I., Stoll, G., Everett, M., and Hanrahan, P. WireGL: A scalable graphics system for clusters. In *SIGGRAPH 2001, Computer Graphics Proceedings* (2001),

E. Fiume, Ed., Annual Conference Series, ACM Press / ACM SIGGRAPH, pp. 129–140.

[43] Klimenko, S., Nikitin, I., Göbel, M., and Tramberend, H. Visualization in topology: assembling the projective plane. In *Visualization in Scientific Computing '97* (1997), W. Lefer and M. Grave, Eds., Eurographics, Springer-Verlag Wien New York, pp. 95–104.

[44] Klimenko, S., Nikitin, I., Urazmetov, V., Göbel, M., and Tramberend, H. Visualization of topologically nontrivial objects: The projective plane. *Programming and Computer Software*, 4 (1998), 191–194.

[45] Krüger, W., Bohn, C.-A., Fröhlich, B., Schüth, H., Strauss, W., and Wesche, G. The Responsive Workbench: A Virtual Work Environment. *IEEE Computer* (July 1995), 42–48.

[46] Krüger, W., and Fröhlich, B. The Responsive Workbench. *IEEE Computer Graphics and Applications* (May 1994), 12–15.

[47] Lalioti, V., Hasenbrink, F., Tramberend, H., and Göbel, M. Immersive telepresence in responsive virtual environments. In *Proceedings of the 9th NEC Research Symposium* (Yokohama, 1998).

[48] Latoschik, M. E. A general framework for multimodal interaction in virtual reality systems: PrOSA. In *The Future of VR and AR Interfaces - Multimodal, Humanoid, Adaptive and Intelligent. Proceedings of the Workshop at IEEE Virtual Reality 2001, Yokohama, Japan* (Sankt Augustin, march 2001), W. Broll and L. Schäfer, Eds., no. 138 in GMD report, GMD-Forschungszentrum Informationstechnik GmbH, pp. 21–25.

[49] Latoschik, M. E. *Multimodale Interaktion in Virtueller Realität am Beispiel der virtuellen Konstruktion.* PhD thesis, Technische Fakultät, Universität Bielefeld, 2001.

[50] Latoschik, M. E. A component-based framework for modelling multimodal interaction in virtual reality. In *IEEE Virtual Reality 2002 conference proceedings* (2002).

[51] Laumann, O., and Bormann, C. Elk: The Extension Language Kit. In *Computing Systems, Fall, 1994.* (Berkeley, CA, USA, Fall 1994), USENIX Association, Ed., vol. 7, USENIX, pp. 419–449.

[52] Leroy, X. The Objective Caml system: Documentation and user's manual, 2000.

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

[54] Macedonia, M. R., Brutzmann, D. P., Zyda, M. J., Pratt, D. R., Barham, P. T., Falby, J., and Locke, J. NPSNET: A multi-player 3D virtual environment over the internet. In *1995 Symposium on Interactive 3D Graphics* (Apr. 1995), P. Hanrahan and J. Winget, Eds., ACM SIGGRAPH, pp. 93–94. ISBN 0-89791-736-7.

[55] Macedonia, M. R., Zyda, M. J., Pratt, D. R., Barham, P. T., and Zeswitz, S. NPSNET: A network software architecture for large-scale virtual environment. *Presence 3*, 4 (1994), 265–287.

[56] MacIntyre, B., and Feiner, S. A Distributed 3D Graphics Library. In *Proceedings of SIGGRAPH '98 (Orlando, Florida, July 19–24, 1998)* (July 1998), ACM SIGGRAPH.

[57] Mercurio, P. J., and Cohen, P. S. Software development using XView and XGL. In *Proceedings of the Sun User Group Technical Conference* (Brookline, MA, USA, June 1991), Sun User Group, Inc., pp. 17–28.

[58] Montrym, J. S., Baum, D. R., Dignam, D. L., and Migdal, C. J. Infinitereality: A real-time graphics system. In *SIGGRAPH 97 Conference Proceedings* (Aug. 1997), T. Whitted, Ed., Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 293–302. ISBN 0-89791-896-7.

[59] Neider, J., Davis, T., and Woo, M. *OpenGL Programming Guide.* Addison-Wesley, Reading MA, 1993.

[60] Purbrick, J., and Greenhalgh, C. Extending locales: Awareness management in MASSIVE-3. In *Proceedings of the 2000 IEEE Conference on Virtual Reality (VR-00)* (Los Alamitos, CA, Mar. 18–22 2000), S. Feiner and D. Thalmann, Eds., IEEE, pp. 287–287.

[61] Reddy, M., Leclerc, Y., Iverson, L., and Bletter, N. TerraVision II: Visualizing massive terrain databases in VRML. *IEEE Computer Graphics and Applications 19*, 2 (Mar./Apr. 1999), 30–38.

[62] RMH. http://www.rmh.de/.

[63] Rogerson, D. *Inside COM.* Microsoft Press, 1996.

[64] Rohlf, J., and Helman, J. IRIS Performer: A High Performance Multiprocessing Toolkit for Real–Time 3D Graphics. In *Proceedings*

of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994) (July 1994), A. Glassner, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, ACM Press, pp. 381–395. ISBN 0-89791-667-0.

[65] Shaw, C., Green, M., Liang, J., and Sun, Y. Decoupled Simulation in Virtual Reality with the MR Toolkit. *ACM Transactions on Information Systems 11*, 3 (1993), 287–317.

[66] Springer, J., Tramberend, H., and Frölich, B. On scripting in distributed virtual environments. In *Proceedings of the 4th Immersive Projection Technology Workshop* (2000).

[67] Stratmann, C. Virtual Car. In *SIGGRAPH 99. Proceedings of the 1999 SIGGRAPH annual conference: Conference abstracts and applications* (New York, NY 10036, USA, 1999), ACM, Ed., Computer Graphics, ACM Press, pp. 282–282.

[68] Strauss, P. S. IRIS Inventor, A 3D Graphics Toolkit. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages and Applications* (Washington, DC, USA, Sept.26 Oct.–1 1993), A. Paepcke, Ed., ACM Press, pp. 192–200.

[69] Stroustrup, B. *The C++ Programming Language: Third Edition.* Addison-Wesley Publishing Co., Reading, Mass., 1997.

[70] Teller, S. J., and Sequin, C. H. Visibility preprocessing for interative walkthroughs. In *Computer Graphics (SIGGRAPH '91 Proceedings)* (July 1991), T. W. Sederberg, Ed., pp. 61–69.

[71] Tramberend, H. Toolbased interaction in virtual environments. In *Tutorial Notes of the 1997 Eurographics Conference* (1997).

[72] Tramberend, H. Avocado: A distributed virtual reality framework. In *Proceedings of the 1999 IEEE Conference on Virtual Reality (VR-99)* (Los Alamitos, CA, 1999), IEEE, pp. 14–21.

[73] Tramberend, H. A display device abstraction for virtual reality applications environments. In *Proceedings of the Afrigraph 2001 Conference* (2001), African Graphics Association.

[74] Tramberend, H., Frölich, B., and Hasenbrink, F. Tools, mediators, and interaction operators. In *Proceedings of the 3rd Immersive Projection Technology Workshop* (Vienna, 1999), Springer, pp. 77–79.

[75] van Renesse, R., Birman, K. P., and Maffeis, S. Horus, a flexible Group Communication System. *Communications of the ACM* (Apr. 1995).

[76] Vaysburd, A. Building reliable interoperable distributed objects with the maestro tools. Technical Report TR98-1678, Cornell University, Computer Science, Apr. 24, 1998.

[77] Vertigo. http://www.vertigo.de/.

[78] Vrml97 international standard, 1997. VRML Consortium.

[79] X3d final working draft specification, July 2002. VRML Consortium.

[80] Waitzman, D., Partridge, C., and Deering, S. E. RFC 1075: Distance vector multicast routing protocol, Nov. 1988.

[81] Waters, R., Anderson, D., Barrus, J., Brogan, D., Casey, M., McKeown, S., Nitta, T., Sterns, I., and Yerazunis, W. Diamond park and spline: Social virtual reality with 3D animation, spoken interaction and runtime extendability. *Presence 6*, 4 (1997), 461–481.

[82] External authoring interface, 2002. Web3D Consortium.

[83] Wernecke, J. *The Inventor Mentor*. Addison-Wesley, 1994.

[84] Wernecke, J. *The Inventor Toolmaker*. Addison Wesley, Apr. 1994.

[85] Yang, Z., and Duddy, K. CORBA: A platform for distributed object computing. *OSR 30*, 2 (Apr. 1996), 4–31.