View Management for Distributed User Interfaces

Blaine A. Bell

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Graduate School of Arts and Sciences

Columbia University
2005

Abstract

View Management for Distributed User Interfaces

Blaine Bell

We present a new approach to designing user interfaces based on view management. *View management* refers to layout decisions that determine spatial relationships between objects, taking into account visibility constraints that allow applications to manage what users see. Our techniques are used to satisfy these constraints by controlling what is seen in a wide range of user interfaces: from 2D desktops to 3D immersive environments, from view-only presentations to interactive techniques, and from single user to collaborative situations.

Screen space is a difficult resource to utilize properly in user interfaces. Poorly designed user interfaces, consisting of overlapping windows, unwanted popups, and unused screen space, occur on many different types of displays, such as PDAs, laptops, cell phones, and wall-sized displays. We avoid these problems by providing techniques for representing and managing unused screen space to avoid overlapping when possible. Applying these techniques to 3D user interfaces imposes visual constraints on the placement of objects, such as labels and annotations, relative to the 2D visible projections of 3D objects on the view plane. We develop techniques to handle issues such as performance and temporal stabilization, and we create guidelines to help ensure that labels and annotations behave well across a wide range of situations.

We have applied our techniques to *augmented reality*, in which information is visualized directly within the context of the real world by overlaying graphics onto what is seen. We develop tools, including an annotated situation-awareness aid based on a *world in miniature*, that make it easy to visualize information about the environment, including parts of the environment that might not be completely visible from their current location.

We extend these ideas further into distributed interactive environments. In a world encompassing a wide range of display technologies, we are interested in how people will access, use, and interact with information and each other. All of this work presented has been developed (or redeveloped) using an efficient rule-based programming technique we call *Data Programming*, which is designed for standalone and distributed development and has given us the flexibility to investigate a wide range of scenarios in these environments.

Contents

Acknowledgements

While this thesis has most definitely been the largest venture that I have undertaken so far in my life, there are so many people to whom I am grateful for helping me accomplish it. From the most detailed technical advice of suggestions, comments, and constructive criticisms to the unconditional love and support received from my family and friends, I am ever endowed with the knowledge that I have received from these people throughout the years. I would like to thank:

- My advisor, Steven Feiner, for his guidance and support throughout the entirety of this work. His natural gift to persistently strive further and his extraordinary ability for paying attention to details have forever made an impact on my work and the quality of this thesis.

- The members of my thesis committee for helpful advice, comments, and suggestions: John Kender, Peter Allen, Michelle Zhou, and Simon Julier.

- All of my colleagues associated with the Computer Graphics and User Interface Lab at Columbia University, which is where all of the work for this thesis was completed.

    - Michelle Zhou for initially introducing me to the MAGIC project and her unique perspective on the automatic generation of computer graphics which has always kept me on my toes.

    - Tobias Höllerer for collaborating with me in many projects. I am appreciative for his valuable advice, both professionally and personally.

    - Everyone who has worked in the lab and have in some way influenced my work by numerous discussions, as well as contributing to many of the demonstrations and exhibitions at many conferences, events, and on campus at an almost everyday basis: Drexel Hallaway, Hrvoje Benko, Sinem Güven, Edward Ishak, Alex Olwal, Chris Sandor, Takashi Okuma, Simon Lok, and Gábor Blaskó.

- My colleagues at the Naval Research Laboratories for also collaborating in demonstrations and helping me integrate my work into their Battlefield Augmented Reality System (BARS): Yohan Baillot, Simon Julier, Dennis Brown, Mark Livingston and Larry Rosenblum.

- My first research advisor at the Naval Observatory in Washington DC, Miran Miranian, for introducing me to research and revealing the excitement and gratification of hard work, which I has stuck with me ever since.

- The first and kindest person I have met at Columbia University, Linda Meehan. From the minute that I met her, she has been so kind, enthusiastic, and supportive

of everything I have done, during my time at Columbia, both undergrad and graduate school, and in-between.

- My boys in Brooklyn: Anson Lang, Mike Clausen, and Devin Burnam, who have given support and inspiration through their examples to help me extend my curiosities far beyond the extent of this thesis, both socially and creatively.

- My hockey boys: Rich McHugh and Greg Christopher, who have not only encouraged me to pursue athletic interests with the same enthusiasm as this thesis, but have also continually strengthened my pursuit to travel, both from their own experiences and ones we have had together.

- My brother Doug, who probably knows me the best, has encouraged me to pursue my ambitions in every step of the way. I think there is no one else better that can work together with me as a team, and there is no team better that can mutually motivate each other in just about anything we do, separately and together. From any entrepreneurial endeavor, to adventurous travels, and to the exploration of cultural and social experiences, he has always been there for me and listens as much as he speaks the truth.

Lastly, and most importantly,

- Mom and Dad. I want to express my love, respect and gratitude to them for everything they have done for me. They bore me, raised me, taught me, inspired me, and loved me. Not once did they ever hesitate to support me in anything I pursued. I am eternally grateful and to them I dedicate this thesis.

This work was supported in part by the National Library of Medicine Grants 5-R01 LM06593-01 and 5-R01 LM06593-02; the New York Science and Technology Center for Advanced Technology in Information Management, Computer Science, Medical Informatics and Genomics; Office of Naval Research Contracts N00014-97-1-0838, N00014-99-1-0249, N00014-99-1-0394, and N00014-99-1-0683; NSF Grants IIS-00-82961 and 01-21239; and gifts from Intel, Microsoft, and Mitsubishi Electric Research Labs. I was supported by an IBM Graduate Research Fellowship for the academic year of 2002-3.

For Mom and Dad

Chapter 1
Introduction

Designing and implementing a user interface (UI) is a difficult and time-consuming task. Well-designed UIs developed for specific tasks and applications are difficult to reuse, and often need to be done by an artist or designer in a case-by-case basis. On the contrary, we want to create techniques that assist UI design to make it easier to build flexible and intuitive UIs with little manual intervention by the designer. This dissertation presents the idea of *view management*, which refers to maintaining visibility constraints on the projections of objects on display surfaces, while also taking into account users' viewing perspectives. We believe this concept along with some of the techniques should exist in almost any application, whether the UI has one or many displays and/or users, whether it uses 2D or 3D projections, or whether it is a view-only presentation or an interactive application. In some specific situations, these techniques might be utilized more than others. In this chapter, we discuss the goals we have set out to accomplish (Section 1.1) and provide a summary of our research contributions (Section 1.2).

1.1 Research goals

A key part of user interface design involves determining the positions and sizes of the objects that the user sees, a task often known as *layout*. In 2D user interfaces, the geometry of 2D objects directly determines their geometry on 2D displays, and their assignment to layers directly determines which of a set of overlapping objects obscures the others. In contrast, in 3D user interfaces, 3D objects are viewed from one or more viewpoints, and projected onto 2D view planes that are mapped to 2D displays. Even in a static 3D scene, changing the viewpoint can change the positions, sizes, and visibility relationships of the objects' 2D projections. Thus, the positions and sizes of the objects' projections and their visibility relationships depend on both the objects and the viewpoint. These visibility dependencies determine what is shown on the display and have a major influence on how the user interface is designed.

Our research focuses on developing software that is aware of these visibility relationships by processing screen space to introduce and explore new automated techniques for graphical user interfaces. Unlike most traditional layout techniques, our research deals with layout decisions that take into account objects that are already displayed and possibly constrained to a portion of the screen. Instead of focusing on the initial layout of information alone, our layout decisions enhance the interface throughout the presentation, considering layout changes during the users' interactions and automatic presentation discourse without redesigning the entire interface.

Introducing these techniques by no means provides complete solutions for every application scenario. However, by exploring a wide range of applications and situations, we can offer strategies for using different algorithms and methods based on our experience. Nonetheless, many of our solutions, specifically for interactive layouts, have acceptable results that allow applications to obtain better control of what is displayed.

1.2 Research contributions

This thesis makes the following contributions:

- *Space management* (Chapter 2) allows programs to use dynamic spatial representations for designing layouts that avoid overlapping already displayed objects on the screen. We introduce representations and algorithms that provide a means for quantifying screen space, as well as other multidimensional spaces, that are used for efficient management throughout an application's duration. We have developed examples that apply techniques for using and querying these spatial representations that also provide a foundation from which all of the layout methods in this thesis are derived.

- *View management* (Chapter 3), which is based on space management, makes it possible to control what users see in 3D user interfaces by computing the 2D projections of objects on the view plane, taking into account visibility. Throughout this thesis, new ways of controlling overlaid information, such as labels and annotations, will be described for different environments, such as Virtual Reality (VR) and Augmented Reality (AR). We classify view management techniques into

three different categories (automatic, user-driven, and collaborative/distributed) and provide numerous examples of each. We also have developed guidelines to help developers understand when to use the techniques, and how to best select parameters for optimal results.

- The design of an *annotated situation-awareness aid* (Section 3.4.2), based on a "world in miniature" provides AR and VR users with a god's eye view of the environment to improve their understanding of it. This tool incorporates *user-driven* view management that is hands free and is controlled mostly by a user's head movement. We incorporate this tool into many parts of this thesis, and it is used it to present information about the environment in both virtual and augmented reality.

- *Data Programming* (Chapter 4), a new declarative rule-based programming method, is based on defining programs as persistent in-memory tabular structures that combine data and code. This new programming technique allows developers to build applications using space management and view management, as well as other complex interactive applications that are flexible and easily extensible. We extend Data Programming to build rules for controlling what is displayed in distributed graphical systems that use multiple distributed displays controlled by multiple computers. Benefits also include easy integration of complex modules and component reuse. Everything presented in this thesis has been implemented (or re-implemented) and extended in Data Programming.

1.2.1 Space management

There are many situations in user interfaces in which objects need to be introduced without overlapping other objects that are currently displayed. For example, a window in a window system might need to be placed such that it matches criteria for position, size, or aspect ratio, yet does not overlap any existing windows. In most spatial representations typically used in interactive graphics, it is easy to determine whether a conflict has occurred, either by a linear comparison with a list of allocated objects, or by more efficient approaches provided by a hierarchy of bounds or an interval tree [119]. However, these representations do not make it easy to select a new placement, modify an

existing placement, or even determine whether it is possible to position an object anywhere without overlap. The result is a plethora of applications that adopt a simplistic approach to allocating space. These are typified by window managers and browsers that position newly created windows at arbitrary positions, or that apply simple tiling or cascading strategies on demand, but which do not maintain the layout as users move and resize objects. Consequently, users often need to modify object geometry by hand. This can be especially tedious when large numbers of objects or multiple applications are involved.

To address these problems, we have developed an efficient approach (presented in Chapter 2) for representing, building, and querying *empty space*—the dual of the *full space* occupied by objects of interest to the user. This spatial representation, which uses axis-aligned shapes such as rectangles, cuboids, or other shapes of arbitrary dimension, is used to keep track of important displayed objects and to avoid overlapping these objects when more information needs to be shown in applications such as ones shown in Figure 1.1.

We describe the representation, discuss the implicit assumption of using axis-aligned shapes (Section 2.2), and provide efficient incremental algorithms for adding (Section 2.2) and deleting (Section 2.4) full-space rectangles that represent important objects, and for querying the empty-space representation (Section 2.5). By querying the representation, we show several allocation strategies and discuss performance (Section 2.6). We present example applications (Section 2.7) that apply the strategies to lay out components in three testbed applications: a window manager (Section 2.7.1), a multimedia presentation system (Section 2.7.2), and an internet browser (Section 2.7.3).

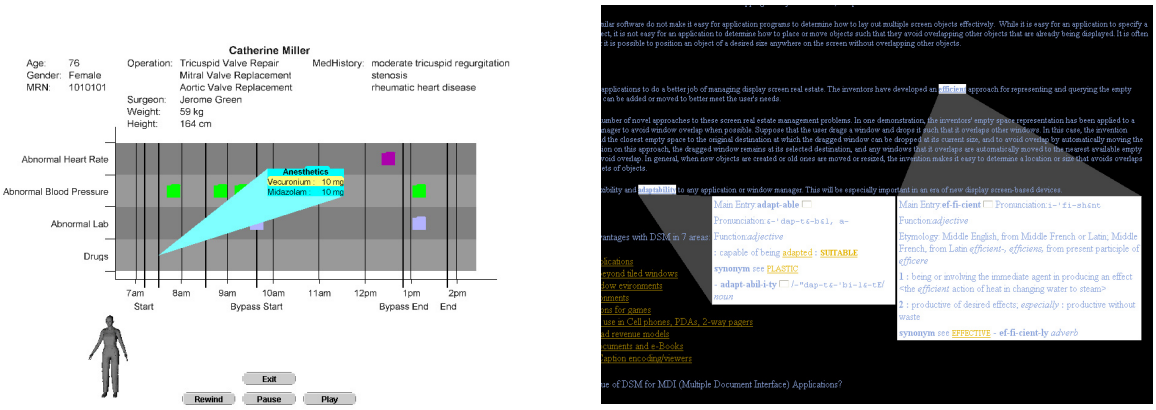


Figure 1.1 Space management applied to (a) a multimedia presentation system for showing detailed information within a timechart interface and (b) an internet web site that shows word definitions.

Figure 1.2 View management in an augmented reality meeting scenario (imaged through a tracked, optical see-through, head-worn display).

1.2.2 View management

We apply 2D space management to 3D graphics to address *view management*, which refers to the layout decisions that determine spatial relationships between 2D projections of 3D objects that obey some desired set of constraints, taking into account visibility. For example, some objects may be sufficiently important to the tasks being performed that their visibility should be guaranteed, while others may need to be rendered at some minimum size or preferred aspect ratio to improve their legibility. Furthermore, there are many situations in which the viewpoint or the geometry of certain objects may be dynamic, yet externally determined, and relatively unpredictable. For example, the viewpoint may be constrained to that of a head-tracked display, and some of the objects may be controlled by a simulation or may be part of a changing physical world viewed through the see-through display of an augmented reality system.

Figure 1.2 demonstrates how automated view management is applied to a collaborative augmented reality environment [7, 24, 28, 133]. It is photographed through a see-through head-worn display from the perspective of one user sitting across from another,

discussing a virtual model of the Columbia campus located between them. Labels are placed relative to buildings' projections for readability without overlapping or ambiguity. Additional annotations in the scene include a meeting agenda (shown at the left), a virtual copy of a building that one of the users has created, and popup information related to that building (both shown at the right). The projections of these annotations avoid other objects, including the visible user's head, to maintain a clear line of sight between users.

Figure 1.3 shows a simple interactive 3D information visualization system that allows users to inquire about buildings on our campus. Visible buildings are initially labeled to show their identity, and to convey what is selectable. When a user selects a building, more information about it is displayed, and placed close to the building's visual projection with a leader line that points at the lower right the building's projection to convey the information's relationship. A virtual avatar is shown in the model with a comic-strip speech balloon annotation, constrained at a fixed position relative to the projection of the avatar's mouth. All annotations are laid out so they do not overlap each other or any of the buildings' projections. Automated placement of these annotations by our view management module ensures that important objects in the 3D scene remain unobscured when providing the user with useful information.

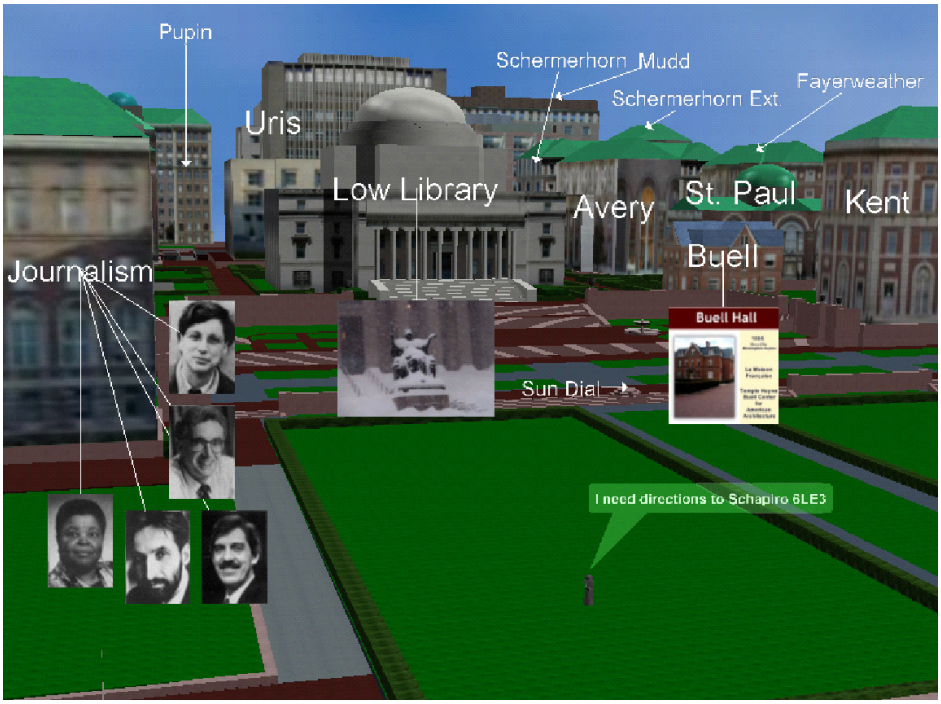


Figure 1.3 Screen shot of a 3D campus information visualization system.

In the real world, many different factors influence what we see, whether done routinely, subconsciously, or without control. For example, we push aside a centerpiece to enable eye contact with others at dinner, or we hold a city guidebook so that we can read a relevant section, while simultaneously viewing an historic building that it describes as we pass by on a bus. External elements can affect what we see as well, such as someone's head blocking the view in a movie theater or hockey game. In any of these situations, virtual and augmented reality provide the opportunity to present important information that can assist users when needed, as previously described in the examples (Figures 1.2–1.3). Depending on the application and situation, continuously changing priorities influence what users want to view, both virtually and in the real world.

In order to build a system to support displaying information in these settings, general definitions of properties and constraints are identified (Section 3.2), along with a computational process to help satisfy these constraints (Section 3.3). We then focus on describing the underlying algorithmic strategies behind animation computations of view management (Section 3.3.5), including different ways of minimizing state changes, sustaining positional stability, and using interpolation for eliminating discontinuous jumps. We follow by discussing details on how we implement the view management pipeline (Section 3.3.7).

We classify three methods of view management (Section 3.4) on the control of what users see: automatic (Section 3.4.1), user-driven (Section 3.4.2), and collaborative/distributed (Section 3.4.3). These methods are combined and used concurrently in the view management module. We describe in Section 3.5, from our experience in numerous demonstrations at conferences and within our lab, some guidelines on how to set the annotation parameters (from Sections 3.3.4–3.3.5) based on informal user feedback.

1.2.3 Annotated situation-awareness aid

By overlaying information onto a real or virtual world, an interface can assist users in understanding their environment. This is an especially important task if the user is mobile and the environment is unfamiliar: the virtual overlays need to enrich and explain, rather than clutter and confuse the user's physical surroundings. Users also want to have control over what information the system shows at any given time.

We have developed a situation-awareness aid [21] for AR that is intended to provide the user with an overview of the surrounding environment and the ability to discover, select, and inquire about objects that may not be directly visible to the user within that environment. Our aid is based on a *world in miniature* (WIM) [129] or exocentric "god's eye view" [59]: a miniature overview model of the surrounding environment is embedded within the user's view and allows user-driven view management (Section 3.4.2) by designing the interface so that it is easy for the user to control how much attention they want to devote to the aid. The work described here addresses two key issues. First, hands-free control of our aid makes it easy for the user in a mobile application to determine how much attention they wish to devote to the aid. We describe an approach for controlling the position, scale, and orientation of our aid relative to the view plane as a function of head orientation alone. Second, changes in user focus produce complementary modifications of overlaid annotations that are presented in the full-scale physical environment and in the miniature virtual aid. Figure 1.4 shows a screen shot of our annotation aid in action.

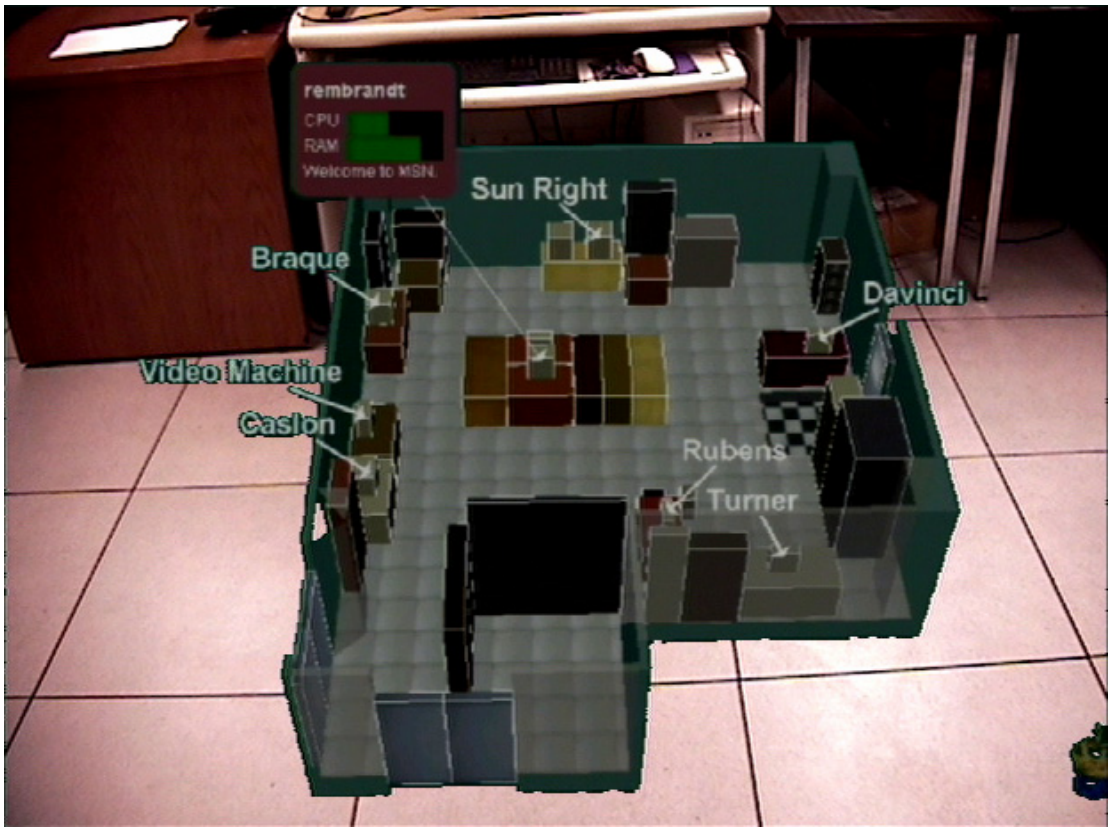


Figure 1.4 Annotated situation-awareness aid provides information about the user's environment

We also use the situation-awareness aid as a tool to exemplify collaborative/distributed view management (Section 3.4.3). Instead of switching attention exclusively to the 3D map, information moves freely between the world and the WIM without losing focus on the real world. In these types of environments, many users are continuously active, often with each other. We envision users will be frequently interested in what others are doing. Because of this, we explore some different ways in which information about other users can be visualized in the WIM. We develop rules that process information in the distributed multi-user environment to automatically choose what to selectively show in each individual's head-worn display. Sometimes a general overview of what others are doing or visualizing is important; at other times a particular event or set of events are important for the system to draw the user's attention. In all of these cases, we use the WIM to provide useful situation-awareness information.

1.2.4 Data Programming (DP)

Interactive view management requires a continuously changing representation of relationships and constraints throughout program execution. From our experience, as the descriptions of constraints that are passed to the module get more complex, it becomes more difficult to implement them using Object-Oriented Programming [26]. Declarative rule-based programming languages such as OPS5 [52], CLIPS [113], and JESS [54], allow users to define complex representations, but fail to provide efficient means to change these representations quickly. Many ways of enhancing these types of rule-based systems have been developed, such as using parallelism [66, 100, 130] and increasing working memory [103], but these still fail to deliver the performance that is needed for interactive graphical programming. The need for performance stems from the requirement of processing rules while images are rendered on the display. In order to satisfy rules interactively, certain rules need to get fired quickly (e.g., before the next frame is rendered). The slowness of rule-based declarative programming styles arises from the memory allocation/de-allocation performed each time a rule gets fired or re-evaluated. This is because in a rule-based system, each pattern match consists of facts and/or partial-orders which occupy space in the working memory and get added or removed when any dependent fact is changed. Thus, in a complex rule-based system, when many rules need to

get fired quickly, most of the processing time is spent on allocating memory (i.e., facts and partial orders) instead of programming logic.

We introduce Data Programming (DP), which differs from conventional rule-based systems and OOP by allowing programmers to control memory management explicitly. This makes programming somewhat more complex, but for interactive applications, it is important to control memory management for performance and understandability. The same types of rules as in rule-based systems can be programmed in DP while minimizing (and eliminating, in many cases) memory allocation when changes occur and rules fire. The goal in DP, as in rule-based systems, is to develop a network of data in working memory to minimize evaluations when the data changes [53]. However, for rules that occur frequently, instead of allocating memory when rules fire, DP uses persistent allocated memory, which consists of Boolean values to represent whether the rule for a particular instance is valid. DP rules consist of Boolean operations, such as ANDs, ORs, and NOTs, that evaluate and change these Boolean variables when necessary. Each Boolean variable represents the validity of an assertion, while the Boolean operations connecting variables together represent rules that behave precisely like those of rule-based systems.

DP's declarative programming style defines tables in the same way as typical main memory Relational Data-Base Management Systems (RDBMS) [40], with the addition of function columns declared within the tables that have relationships to their arguments in the same or adjacent tables. These functions include rule-based Boolean operations, mathematical operations, and any other type of user written function. In contrast to developing programs that use main memory databases as a separate tool, DP unifies the entire program, which is declared with data tables, data rows, and the relationships within and in-between tables.

Our intention for this chapter is to give a high-level description of the concepts of DP. Since we have implemented almost every example shown in this thesis in DP, we feel that it is important to describe our new programming paradigm. As interactive applications get more complex, we believe that DP is an easy way to add complex logic to existing programs without redesigning the entire system. We discuss how DP has evolved from rule-based systems (Section 4.1), followed by comparing DP with respect to previous work related to data-based and knowledge-based graphics (Section 4.4), and finally

discuss the benefits and drawbacks of this approach compared to other approaches, such as shared memory, component-based, and data flow systems (Section 4.5).

Chapter 2

Space Management

There are many examples in user interfaces in which applications should introduce information that does not obscure already displayed objects. For example, a popup window in a word processor or internet web browser ought to avoid important figures, paragraphs, and images so that the user can continue viewing them without an abrupt interruption. By placing new information in a position without overlap, the system could allow the user to observe both of the previously and newly displayed information at the same time, thus adding utility to the display by showing more information and possibly saving the users time.

When trying to implement this functionality, it is easy to determine whether an overlap has occurred, either by a linear comparison with a list of objects, or by more efficient approaches provided by hierarchy of bounds or interval trees [119]. Graphical systems commonly provide these kinds of representations, such as window trees of 2D window systems [121], and bounding volume hierarchies of 3D graphics packages and renderers [35, 81], that are typically used for efficiency in a variety of operations, such as rendering, clipping, and collision detection. However, these representations do not make it easy to determine how to modify a desired placement or select a new one to avoid the conflict, or even whether it is possible to position an object anywhere in the scene without overlap. Unfortunately, this lack of spatial representations for avoiding overlap forces most applications to introduce new information in either an arbitrary placement, such as the center of the screen or the information's last placement, or constrained to a reserved place on the screen, such as the sides of a window or screen.

For this reason, we introduce s*pace management* [17], which is an efficient approach for representing, building, and querying *empty space*—the dual of the *full space* that is occupied by objects of interest to the user. By using this spatial representation, we are able to keep track of important objects and avoid overlapping these objects by querying the empty space to find new placements when more information is introduced. For

simplicity and efficiency, we work with floating-point axis-aligned shapes, such as rectangles, cuboids, or other shapes of arbitrary dimension, within an axis-aligned workspace. Efficient algorithms are provided for incrementally adding and deleting full-spaces, and for querying the empty-space representation that is maintained. Although we have implemented multi-dimensional space management, since the algorithms are easily extendable to *N* dimensions, applications presented in this thesis use the 2D spatial representation consisting of 2D full and empty space rectangles. These full-space rectangles may represent the exact dimensions of objects (e.g., the rectangular windows of a 2D GUI) or approximations (e.g., the upright extents of the 2D projections of 3D objects). At any time, the user can query the empty-space representation, which is automatically maintained by the system. The empty-space representation consists of the largest empty-space rectangles, each of whose height and width is as large as it can be without overlapping any full-space rectangle. That is, each of the largest empty-space rectangles is bounded on the left, right, top, and bottom by an edge of a full-space rectangle or an edge of the workspace. For example, Figure 2.1 shows the four possible largest empty-space rectangles surrounding a single full-space rectangle. Note that the largest empty-space rectangles do not necessarily partition the empty space, but can overlap, as in Figure 2.1(b).

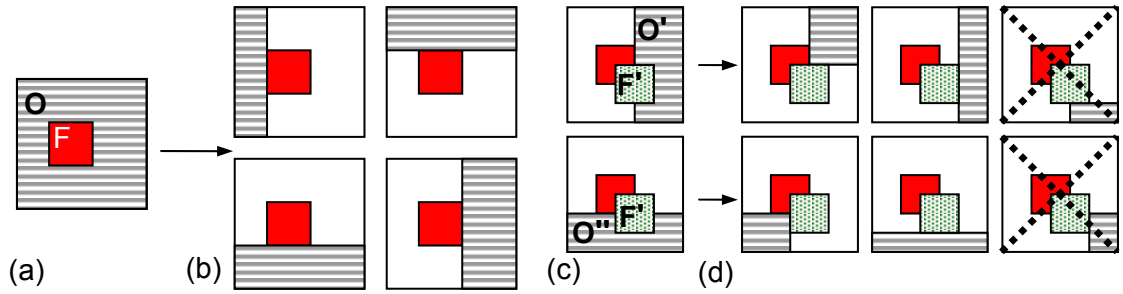


Figure 2.1 Adding a full-space rectangle F. (a) Adding a solid red full-space rectangle F to an empty scene, represented by a striped light gray single largest empty-space rectangle O, (b) Creates four new striped light gray largest empty-space rectangles, (c) Another full-space rectangle F' (dotted green) reduces two largest empty-space rectangles O' and O'' (striped light gray). (d) Four empty-space rectangles (striped light gray) are created. Dashed "X"s indicate new empty-space rectangles that are not largest empty-space rectangles, since these rectangles are wholly contained within other resulting rectangles..

This representation is well suited for determining the placement of upright rectangular objects that avoid overlapping. Any rectangle that can be placed within the empty space will be completely enclosed by at least one of the largest empty-space rectangles.

Finding a largest empty-space rectangle that satisfies simple geometric constraints on position, width, height, or aspect ratio, requires only simple comparisons with the desired full-space rectangle to be added. For this approach to be successful, the empty-space representation must be computed automatically, offering interactive performance to applications that need it.

Many prototype applications that we have developed make use of our implementation, including a window manager testbed, a knowledge-based 3D presentation designer, and an internet browser. Colleagues in our lab and at other institutions have also downloaded our software and used our representation in their applications [6, 72, 73].

2.1 Related work: Spatial representations and algorithms

A number of spatial representation schemes have been developed that explicitly represent empty space. For example, quadtrees and octrees [119] provide 2D and 3D discrete approximations of full and empty space using a hierarchy of axis-aligned cells. Corner stitching [105], the space representation for axis-aligned rectangles used in the MAGIC VLSI layout system [106], supports efficient queries of full and empty spaces adjacent to a given full or empty space. However, the basic structure of the corner-stitching algorithm tiles the plane with non-overlapping full-space and empty-space rectangles. In this data structure, each rectangle has references to its adjacent rectangles in each direction. Since there can be multiple adjacent rectangles in each direction, these pointers are chosen by the rectangles that are adjacent to the "corners" of the associated rectangle. These pointers, or "corner stitches", are then traversed to query the spatial layout. Overlapping full-space rectangles have been addressed in corner stitching by using multiple overlapping layers (each individually corner-stitched), or by decomposing overlapped rectangles into a set of non-overlapping rectangles before adding them. Thus, all these representations partition the empty space into non-overlapping pieces. Thus, they do not make it easy for the user to find empty space that can accommodate a desired rectangle that is larger than any of these individual partitions. Unlike this previous work, our representation maintains the list of overlapping largest empty space rectangles to allow the quick exhaustive search for empty space with any type of desired constraints.

Bernard and Jacquenet [22] use the same empty-space representation that we do, presenting an incremental algorithm for adding full-space rectangles. However, they do not support overlapping full-space rectangles or deletion of full-space rectangles. Both of these capabilities are important for general interactive applications, such as the ones we describe in Section 2.5.

Several tiled window managers have used empty-space representations to allocate space when non-overlapping windows are created, moved, or resized. Many tiled window managers are restricted to hierarchical layouts (e.g., [79, 135]) in which windows (i.e., full-space rectangles) are allocated in a strict hierarchy by recursively partitioning the workspace horizontally or vertically at each subdivision. Thus, no full-space or empty-space rectangle can straddle partitions. Although it is straightforward to query the empty-space representation in a hierarchical system, this is possible only because of the restrictions that prohibit overlapping empty-space rectangles. In contrast, the Siemens RTL and constraint-based RTL/CRTL tiled window managers [36] support nonhierarchical layouts, relying on corner stitching. However, adjacent empty-space partitions in corner stitching must be combined in both dimensions to determine whether they can accommodate a window of a specified width or height. Unlike any of these window managers, our representation requires a simple search through the list of rectangles to exhaustively query the entire empty space for layout strategies. Using other representations might necessitate combining smaller adjacent rectangles during the query to exhaustively check the empty space.

2.2 Axis-aligned shapes

Throughout this thesis, axis-aligned shapes (rectangles in 2D) are used for representing both full and empty spaces. Although this is a typical assumption that is used for visual objects in user interfaces (e.g., event processing and layout in toolkits [131] and window systems [121]), by restricting our representation to axis-aligned shapes, we are making certain implicit assumptions about how our representation will be used both for determining the input for arbitrarily shaped objects (i.e., full-space objects that should not be overlapped), and imposing limitations on querying our representation.

It is possible to approximate arbitrary shapes by using multiple axis-aligned shapes. Using more shapes in our representation for each object, the quality of the results increase by returning spaces that might not have been found without the improved ap-

proximations. However, since more objects are added into the data structures, it will take longer to obtain the result. Thus, for supporting interactive performance, it is important to use approximations of objects only when needed.

Querying our representation for arbitrary empty spaces has significant limitation when using axis-aligned shapes. Our representation is not well suited for exhaustively searching the empty space for the placement of concave objects. For example, the green dotted concave object in Figure 2.2(a) fits within the empty space without any overlap but not within any individual axis-aligned empty space (Figure 2.2b). To place an arbitrary using our representation, multiple spaces (i.e., all combinations of overlapped spaces) need to be evaluated. Figure 2.2(b) shows how the new green dotted concave shape fits into the union of the two largest empty spaces on the left. Even so, an exhaustive search of all possible placements of this shape is difficult to compute and represent, since it requires not only determining the translational placement but also the orientation of the arbitrary shaped object within multiple empty spaces. Although it might be feasible to compute these possibilities from our representation, it is not ideal and is much more complicated than determining placement for axis-aligned shapes, which is done by using simple comparisons. Since any novice programmer can easily iterate through a list of rectangles, our representation is appealing for quick and easy implementation using empty space.

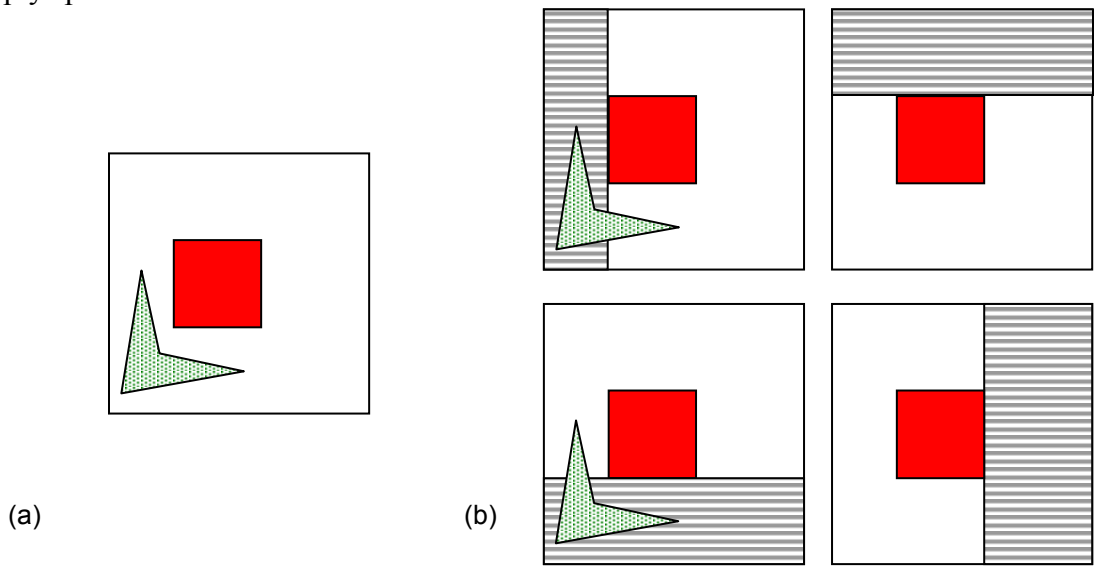


Figure 2.2 Placement for a concave object within the empty space is difficult with our representation. (a) Possible non-automated placement without overlap, (b) Non-overlapped concave object placed without being completely enclosed by at least one largest empty space.

2.3 Adding a full-space rectangle

Given an existing list of full-space rectangles with an associated empty-space representation, when a full-space rectangle *F* is added, the set of largest empty-space rectangles needs to be modified. Only those largest empty-space rectangles that intersect *F* will be affected, since they are the only rectangles whose bounds can change because of the addition.

The fundamental operation for adding a full-space rectangle is *reduction* [22]: the creation of new, smaller empty-space rectangles from existing largest empty-space rectangles *O* that intersects full-space rectangle *F*. For each edge *e* of *F* that intersects *O* and is not collinear with any edge of *O*, a new empty-space rectangle is created. This new empty-space rectangle is bounded by *e* and the three other edges of *O*; thus it corresponds to the portion of *O* that is on the side of *e* outside of *F*. For example, if the left edge of *F* intersects *O* (as in Figure 2.1a), then *O* can be reduced to a rectangle whose left, top, and bottom edges are inherited from *O*, and whose right edge is the left edge of *F* (top left of Figure 2.1b).

Each edge in rectangle *F* can reduce rectangle *O* to create between zero and four new, smaller rectangles: one for each edge of *F* that intersects the interior of *O*. For example, Figure 2.1(b) shows all of the four empty-space rectangles created by reducing the outer rectangle *O* (Figure 2.1a) by the solid red rectangle *F* that is fully contained in *O* and touches none of *O*'s edges. In contrast, *F* could not reduce *O* at all if *O* is fully contained in *F* and touches none of *F*'s edges.

The rectangles created by reducing *O* by *F* represent the empty-space remaining after the addition of *F*. Thus, to maintain the empty-space representation, we must use *F* to reduce each largest empty-space rectangle that it intersects. Each of the original largest empty-space rectangles that intersect *F* must be removed from the list of largest empty-space rectangles. However, we cannot simply replace the rectangles that are removed by the rectangles created by reducing them—some of the new reduced rectangles may be contained within others. For example, Figure 2.1(d) shows the six empty-space rectangles created by adding another full-space rectangle *F′*, reducing the two largest empty-space rectangles that overlap *F′* in Figure 2.1(c). Two of these six new empty-space rectangles are contained inside the other four and are "X"ed out in Figure 2.1d. Therefore, we must

add only these four largest empty-space rectangles to the list of largest empty-space rectangles.

A new empty-space rectangle adjacent to a full-space rectangle's edge *e* only needs to be tested for containment against other empty-space rectangles adjacent to *e*. However, it is important to note that it must be tested against all such empty-space rectangles, including those largest empty-space rectangles that share only the one edge *e*, and which therefore cannot be reduced by this full-space rectangle which is being added. This is shown in Figure 2.3, where adding the dotted green rectangle in part (c) reduces the striped light gray largest empty-space rectangle in part (a) to create a single new empty-space rectangle on its right. However, this new empty-space rectangle is wholly contained within the striped light grey largest empty-space rectangle in (b), and therefore is not added to the list of largest empty-space rectangles.

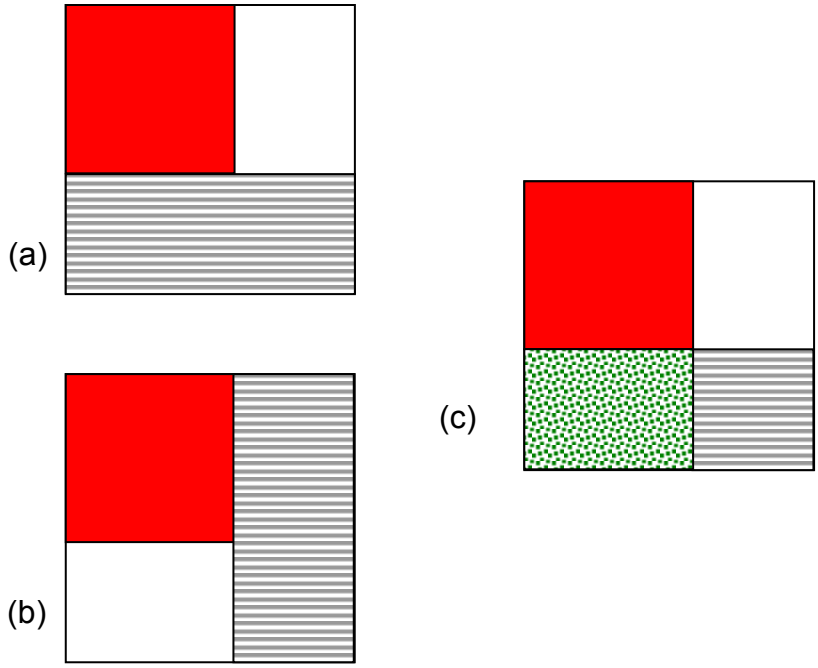


Figure 2.3 (a–b) A solid red full-space generates two striped light gray largest empty-space rectangles. (c) Adding the dotted green full-space rectangle into the lower left corner generates the striped gray empty space shown in (c) on right that must be compared with largest empty space in (b) to determine that the new empty space is not a largest empty space.

Figure 2.4 shows the pseudo-code for the incremental add of full-space rectangle *F*. First, the set of largest empty-space rectangles that intersect *F* or are adjacent to it are computed using a window query (see Section 2.5). Next, the rectangles of this set are

compared with *F*. Each rectangle that is adjacent to a side of *F* and external to *F* is added to a list of largest empty-space rectangles adjacent to that side. All other rectangles in the set are deleted from the set of largest empty-space rectangles and are reduced by *F* to create a separate list of new empty-space rectangles for each edge of *F*. Then each edge's possible empty-space rectangle list is culled to eliminate members that are not largest empty-space rectangles, and the surviving members are added to the list of largest empty-space rectangles. This algorithm is easily extended to *N* dimensional spaces by inserting comparisons and combinations to rectangles that are adjacent to edges in each additional dimension.

Our addition operation differs from that of Bernard and Jacquenet [22] in its ability to handle overlapping rectangles. Using the interval tree data structure [119], we use one query operation to obtain the set of relevant overlapping or adjacent largest empty-space rectangles and process each of them against the newly added rectangle *F*. In contrast, Bernard and Jacquenet use four query operations (one for each edge of *F*) to find all largest empty-space rectangles intersected by each edge and then reduce them by that edge. By checking only edge intersection, rather than rectangle intersection, their algorithm does not handle overlapping rectangles properly. For example, suppose that *F* fully covers a largest empty-space rectangle *O,* and *F*'s edges lie wholly outside *O*. Rectangle *O* should be deleted, but this will not occur when *F*'s edges are used for querying because *O* is not intersected by any of *F*'s edges. Bernard and Jacquenet also do not take into account the necessary comparisons for maintaining a unique set of largest empty-space rectangles, as demonstrated in Figure 2.3. The new empty-space rectangle generated by adding a new full-space rectangle should be compared with other existing empty-space rectangles that are adjacent but not changed during the operation. By excluding these extra comparisons, the algorithm results in spurious additions to the list of largest empty-spaces.

```
SpaceManager.addFullRectangle(Rectangle F){
  fullSpaceList.add(F)
  CList = existing rectangles in largestEmptySpaceList
    that intersect or are adjacent to F
  Vector possibleLESList[4] = new Vector[4]
  Vector adjacentLESList[4] = new Vector[4]
  for each empty space O in CList {
    if (O adjacent to any side e of F and is external to F)
      adjacentLESList[e].add(O)
    else {
      largestEmptySpaceList.delete(O)
      if (O.minx < F.minx)   /* Figure 1(b), top left */
        possibleLESList[LEFT].add(
          Rectangle(O.minX,F.minX,O.minY,O.maxY))
      if (O.maxX > F.maxX)/* Figure 1(b), bottom right */
        possibleLESList[RIGHT].add(
          Rectangle(F.maxX,O.maxX,O.minY,O.maxY))
      if (O.minY < F.minY)  /* Figure 1(b), bottom left */
        possibleLESList[BOTTOM].add(
          Rectangle(O.minX,O.maxX,O.minY,F.minY))
      if (O.maxY > F.maxY)/* Figure 1(b), top right */
        possibleLESList[TOP].add(
          Rectangle(O.minX,O.maxX,F.maxY,O.maxY))
    }
  }
  for each Direction D in (LEFT,RIGHT,BOTTOM,TOP){
    for each empty space P in possibleLESList[D] {
      if (P not enclosed by any space in
          adjacentLESList[D] or any other space in
          possibleLESList[D])
        largestEmptySpaceList.add(P)
    }
  }
}
```

Figure 2.4 Incremental addition of a full-space rectangle

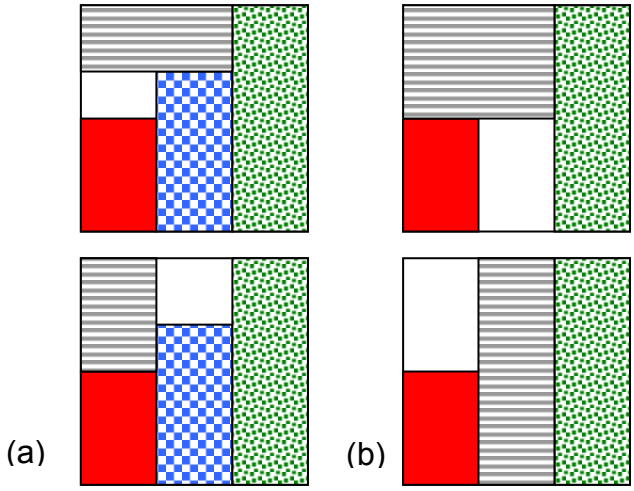


Figure 2.5 Deletion is not as simple as expanding current empty space rectangles. (a) Three full-space rectangles (solid red, dotted green, and checkered blue) within an area that defines the two striped light gray largest empty-space rectangles. (b) Deleting the checkered blue full-space rectangle should yield this new set of dashed light gray largest empty-space rectangles. Note that these largest empty-space rectangles in (b) cannot be obtained by expanding each dashed light gray rectangle in (a) in one dimension after deleting the checkered blue rectangle

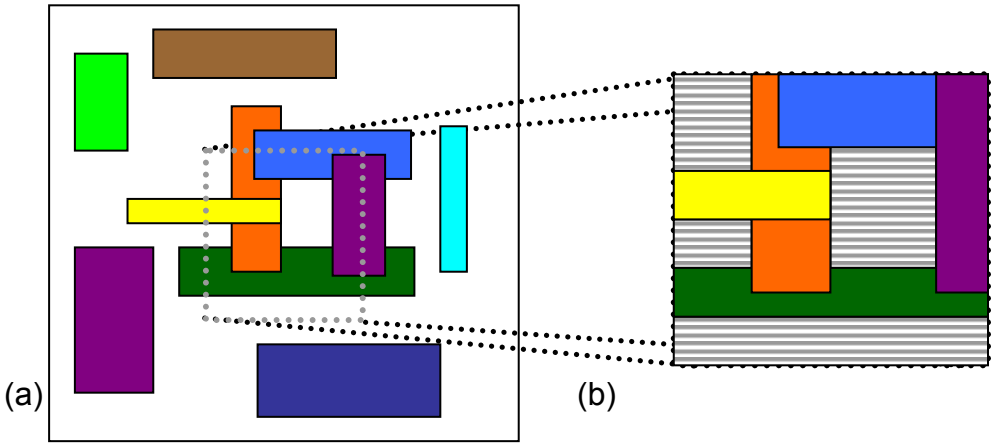


Figure 2.6 (a) The deletion of an overlapping full-space rectangle, shown as gray dotted outline. (b) Recursive space manager is created to process the area formerly occupied by the deleted rectangle, to determine the four striped light gray largest empty-space rectangles within the area.

2.4 Deleting a full-space rectangle

Deleting a full-space rectangle *F* is essentially the inverse of adding it. We need to modify the largest empty-space list to include all largest empty-space rectangles that intersect *F*'s area after it is deleted. At first it might seem that we would only have to re-

construct the largest empty-space rectangles destroyed when *F* was added, by 1D expansion (the inverse of reduction) of the empty-space rectangles created by reducing each largest empty-space rectangle *O* that intersected *F*. Unfortunately, this is not possible, as shown in Figure 2.5, because of the empty-space rectangles that would need to be expanded may have been discarded in a previous operation since they were not largest empty-space rectangles. Furthermore, the largest empty-space rectangles that will intersect *F*'s area after it is deleted may be different than those that intersected its area before it was added. In fact, full-space rectangles intersected by or wholly contained in *F* may define largest empty-space rectangles unrelated to those present before *F*'s deletion.

To delete *F*, we must find all of the largest empty-space rectangles that intersect *F*'s area after deletion. We compute these by first creating two lists: the largest empty-space rectangles external to and adjacent to *F*, and the empty-space rectangles wholly contained in *F*'s area after its deletion. The list of adjacent external largest empty-space rectangles is obtained with a window query. To find the list of empty-space rectangles within *F*'s area, we recursively invoke our space manager for the area bounded by *F*, adding to it all full-space rectangles intersecting *F*, not including *F* itself. The empty-space rectangles found within *F*'s area are thus the largest empty-space rectangles relative to *F,* as shown in Figure 2.6(a–b).

For each empty-space rectangle found within *F*'s area, we determine the set of edges of *F* to which it is adjacent. An empty-space rectangle inside *F*'s area that is adjacent to no edges of *F* can be added to the list of largest empty-space rectangles without any further processing.

An internal empty-space rectangle that is adjacent to one or more edges of *F* may be combined with zero or more of the external largest empty-space rectangles adjacent to *F* to create new largest empty-space rectangles. Therefore, we keep a list of rectangles that are possibly largest empty-space rectangles. At the end of the algorithm, we add a rectangle to the largest empty rectangle list only if it exists in the possible list and is not enclosed by any other rectangle in the list. We must also delete the empty-space rectangles that were previously adjacent to *F* if they are contained within any of the largest empty-space rectangles added.

```
Method combineSpaces(Rectangle R1,Rectangle R2) {
    if (isAdjacentInX(R1,R2))
        return (Rectangle(MIN(R1.minX,R2.minX),
            MAX(R1.minY,R2.minY),
            MAX(R1.maxX,R2.maxX),
            MIN(R1.maxY,R2.maxY)))
    else if (isAdjacentInY(R1,R2))
        return (Rectangle(MAX(R1.minX,R2.minX),
            MIN(R1.minY,R2.minY),
            MIN(R1.maxX,R2.maxX),
            MAX(R1.maxY,R2.maxY)))
}
```

(a)

(b)

Figure 2.7 (a) Combining adjacent spaces. (b) Two empty spaces (outlined) that are adjacent in Y are combined to create a third (light gray) empty space.

We combine empty-space rectangles by noting that only adjacent rectangles can be merged. If two rectangles are adjacent in one dimension, then they can be used to spawn a new rectangle whose extent in that dimension is the union of the two rectangles' extents in that dimension, and whose extent in the other dimension is the intersection of the two rectangles' extents in the other dimension, as shown in Figure 2.7(a–b). If we iteratively combine each empty-space rectangle *R* within *F* with all those adjacent to it outside of *F*, we will create the set of largest empty-space rectangles. Steven Nowick (personal communication) has pointed out that this is analogous to the consensus theorem used in digital logic to find prime implicants [80].

Figure 2.8 shows the pseudo-code for the incremental delete. We accomplish this merging operation by treating the edges of *F* in sequence, as depicted in Figure 2.9. For each edge *e* of *F*, each internal empty-space rectangle *R* adjacent to *e* is merged with the external largest empty-space rectangles adjacent to *R*. The resulting merged rectangles are added to a merge list, as are all of an edge's internal empty-space rectangles that are not wholly contained in any of the merged rectangles. When the next edge *e* of *F* is proc-

essed, the empty-space rectangles on the merge list that are also adjacent to *e* must be merged with all other empty-space rectangles that are adjacent to *e*. Taking into account the edges of *F* shared by empty-space rectangles avoids the brute force approach of merging explicitly every combination of internal and external empty-space rectangles adjacent to all edges of *F*. The delete algorithm as well as the combine function can easily be extended to *N* dimensional axis-aligned shapes by checking for adjacent shapes along the edges in the new dimensions, extending the combine function for the new dimensions, and including these new edges in the iteration loops (Figure 2.8).

```
SpaceManager.deleteFullRectangle(Rectangle F){
  remove F from fullSpaceList
  create new Space Manager S to represent area of F
  get all full-space rectangles that intersect F
    and add them to S
  adjacentEmptySpaceList  = all largest empty-space
    rectangles adjacent to and external to F
  /* Note: these are the only largest empty spaces external
    to F  that need to be processed in this algorithm */
  /* possibleLESList is a temporary list of
    empty spaces that could possibly be largest empty
    spaces; initialize it to the empty spaces in S */
  possibleLESList = S.largestEmptySpaceList
  for each edge e in (LEFT,RIGHT,BOTTOM,TOP) {
    adjList = spaces in possibleLESList adjacent to edge e of F
    for each empty space R in adjList {
      for each empty space P in adjacentEmptySpaceList  {
        if (P adjacent to edge e of R and edge e of F)
          possibleLESList .add(combineSpaces (R, P))
          /* see Figure 2.7 for function combineSpaces       */
      }
      if (any rectangle in possibleLESList  encloses R)
        remove R from possibleLESList
    }
  }
  for each empty space R in adjacentEmptySpaceList {
    if (R enclosed by a rectangle in possibleLESList)
      largestEmptySpaceList.delete(R)
  }
  for each empty space R in possibleLESList {
    if (R not enclosed by any other space in
        possibleLESList)
      largestEmptySpaceList.add(R)
  }
}
```

Figure 2.8 Incremental deletion of a full-space rectangle

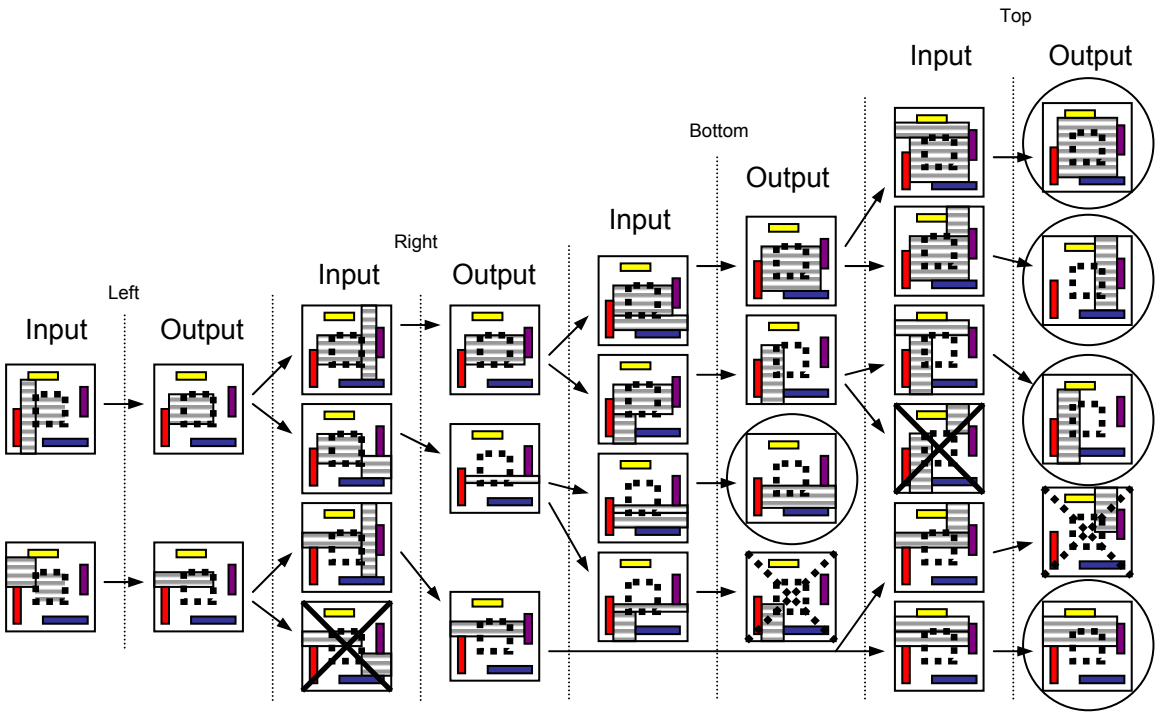


Figure 2.9 Deletion of a non-overlapping full-space rectangle (shown as heavy dashed black outline). Each box in an edge's input column shows a pair of striped light grey empty-space rectangles that are candidates for combination to create an empty-space rectangle in that edge's output column. Circled boxes contain completely processed empty-space rectangles that will be added to the space manager's list of largest empty-space rectangles when the deletion is completed. Boxes covered by a solid "X" contain pairs of spaces that cannot be combined. Boxes covered by a dashed "X" contain completely processed empty-space rectangles that are subsets of other empty spaces computed in this process, and which will be discarded.

2.5 Querying the space representation

The space manager maintains separate data structures for the full-space and largest empty-space rectangles. To optimize the worst-case time and space for querying rectangles within a given set of bounds in X and Y, a 2D interval tree for each set of rectangles is used, requiring worst-case $O(\log^2 N + M)$ time for a query and $O(N \log N)$ space for the data structure, where N is the number of rectangles in the data structure and M is the number of rectangles returned by the query [119]. We also support callbacks, so that each time the space representation changes, applications can be notified to modify their own data structures that are used for optimizing specific queries.

2.6 Space manager performance

We have implemented the space manager in Java 1.2. Performance depends on the distribution of the rectangles, especially within the trajectories of moving full-space

rectangles, and benefits greatly from using the deletion algorithm, instead of recomputing the entire space representation when an object is to be deleted. In general, when moving one full-space rectangle with more than three others in the scene, or moving two full-space rectangles with more than six others in the scene, we have found that it is better to delete and add the moving rectangles from the space manager, rather than to add all full-space rectangles from scratch.

Figure 2.10 compares the time it takes to do a move operation by deleting and adding the moving full-space rectangle (*Delete/Add*) vs. adding all full-space rectangles into an empty space manager (*Add All*). This figure graphs elapsed time for randomly generated sets of full-space rectangles on a 733 MHz Intel Pentium III processor with 256MB RAM, running Windows 2000. There are considerable differences, (e.g., for 100 full-space rectangles we recorded ~3 msec to delete and add one more full-space rectangle, vs. 143 msec to add all the full-space rectangles).

As the number of moving objects and total scene objects increase, both the *Add All* and *Delete/Add* method can become quite slow. This is especially noticeable if many objects are moved continuously, with space allocation performed at each frame. To address this problem, we have implemented an efficient undo capability that makes it possible to undo the most recent add operations quickly. In addition to the regular add operation, we provide an *undoable add* that saves the largest empty-space rectangles that it removes, and marks the new largest empty-space rectangles that it creates. An undoable add can be undone by having the space manager remove the largest empty-space rectangles it created and re-add the largest empty-space rectangles that it removed. This undoable add makes it possible to animate a selected group of objects more quickly than by repeatedly deleting and adding them. To accomplish this, the programmer first deletes the objects to be animated and adds them back with new locations or sizes by using undoable adds. Then, for each frame in the animation, the previous frame's undoable adds are undone and the modified objects are added back with undoable adds. Figure 2.11 compares the performance of *Delete/Add* with *undoable add* by interchanging these functions to delete the same full-space rectangles from the same scenes with randomly generated sets of full-space rectangles, similar to the scenes generated for Figure 2.10.

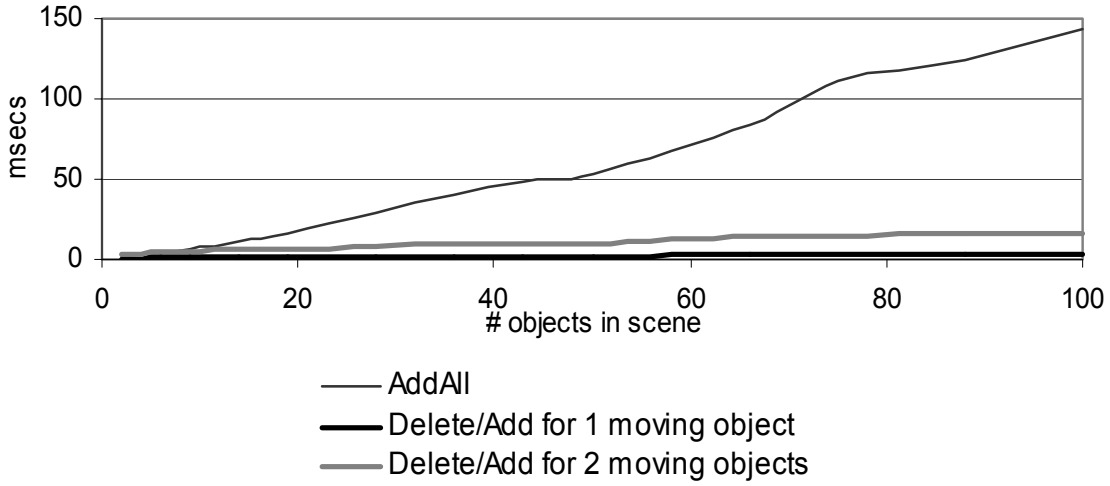


Figure 2.10 Performance comparison of Add All vs. Delete/Add operation durations for different scenes.

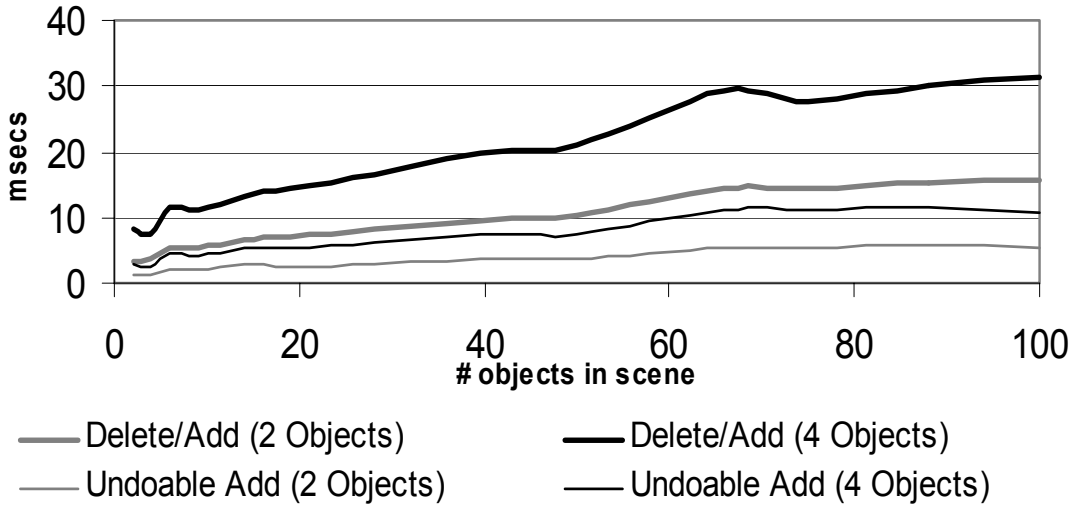


Figure 2.11 Performance comparison of Delete/Add vs. undoable add operation durations for different scenes.

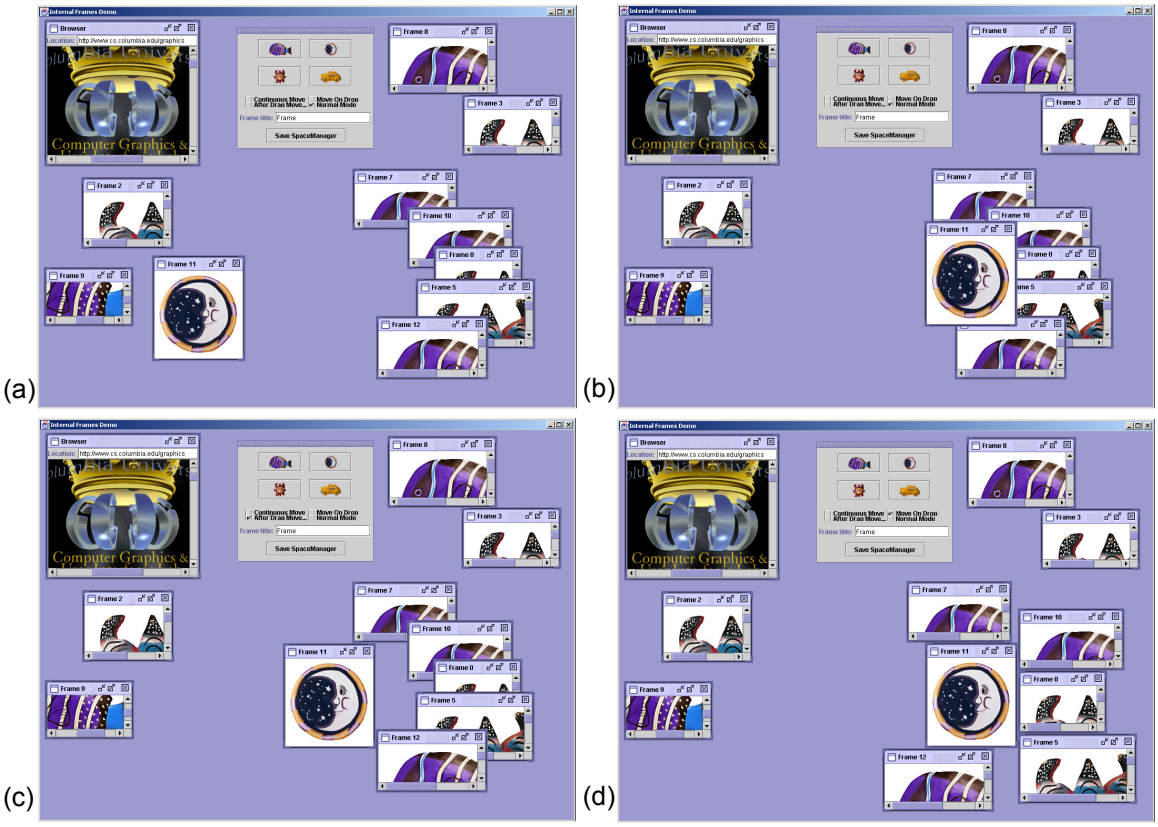


Figure 2.12 Window manager testbed. (a) Dragging the window (the moon) at the lower left, (b) conventionally results in placement where window is dropped. (c) Overlap-avoidance drag interpolates the dragged window to a sufficiently large empty-space rectangle closest to its original destination. (d) Alternative overlap-avoidance approach interpolates all windows overlapped by dragged window to sufficiently large empty spaces closest to their original locations.

2.7 Examples

In the following sections, we present three applications that apply space management strategies to layout. First, we describe how we use it to build functionality into a window manager that avoids overlapping windows when possible, both during user interactions and when introducing new windows. We follow by describing a multimedia presentation system that uses it throughout the duration of the program for information visualization. We conclude by describing strategies and priorities for space management that are used to introduce more information within an internet browser.

2.7.1 Using space management in a window manager testbed

We have used our space manager to create a testbed for window management tasks by modifying the Java InternalFrameDemo [63]. By making it easy to examine the list of largest empty-space rectangles, we can provide some of the benefits of a tiling window manager, such as the lack of window obscuration, within the framework of a desktop overlapping window manager.

Figure 2.12(a–b) shows the screen before and after a conventional window dragging operation. Here, dragging the window containing the image of the moon at the lower left of Figure 2.12(a) and dropping it at the place shown in Figure 2.12(b) causes it to overlap the windows on the right. Avoiding overlap would require more careful placement of the dragged window, or moving the other windows out of the way. We explored two new kinds of automated *overlap-avoidance* dragging. In both approaches, all windows are added to the space representation as they are created. In one approach (Figure 2.12c), a window is deleted from the representation when it is dragged. When the user releases the window at a new location, the window manager queries the largest empty-space representation to find the closest empty space that can contain the dragged window without overlap. If that space is within a parameterized distance of the window's current location, the window is added to the representation and smoothly moved there by the system; otherwise it is added at its original destination. This allows the user to move a window quickly and sloppily, overlapping it with adjacent ones at the new location, knowing that if the window is dragged to a location that is sufficiently near a large enough empty space, it will move to the closest such space to avoid overlap.

An alternative approach (Figure 2.12d) examines the multiple windows that are overlapped by the moved window, deletes each of them from the space manager, and moves them to new destinations to avoid overlap. In order to calculate the destinations, the space manager is used to find the closest empty space for each individual rectangle. However, calculating these destinations separately causes this greedy algorithm to give priority to windows that are processed first, and can fragment the space, resulting in a missed solution. (Alternatively, we could try all possible processing orders, and compute destinations that define the minimal motion across all windows without missing a possible solution.) Currently, if a solution is not found, then some of the rectangles remain in

place despite overlap. Rather than adjusting window positions only after a dragged window has been dropped, we can instead allow the motion to occur dynamically during the move itself. For example, while the dragged window is moved, each overlapped window can be moved out of the way.

In complementary work, Badros, Nichols, and Borning have explored the incorporation of a modern constraint system within a programmable window manager, called SCWM (Scheme Constraints Window Manager) [11]. This system allows the user to place classical constraints on windows; for example, to keep one window to the left of another or to keep the total height of two windows constant. While these constraints can allow interesting and useful behavior, they operate only on the representations of the windows (full-space rectangles). Thus, while a window can push a chain of windows as it moves as in our approach, it cannot move them to prevent overlap with other windows because the system has no useful representation of the empty-space. We believe that an empty-space representation, such as that described here, would be an especially useful addition to such a constraint-based window manager.

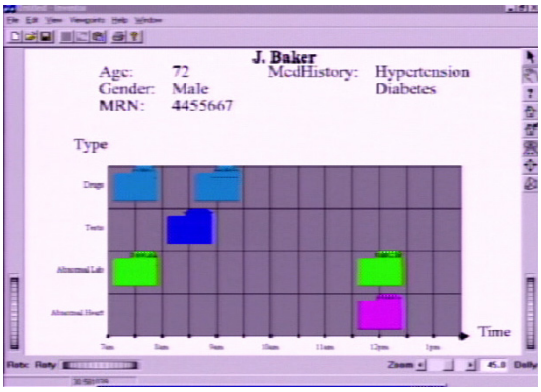
2.7.2 Multimedia presentation system

We have incorporated our space manager into IMPROVISE, an existing research system that automatically generates 3D information visualizations [143]. IMPROVISE's hierarchical-decomposition partial-order planner maps a set of communicative goals to 3D presentation objects. While this system also uses a constraint solver [61] to constrain object geometry, it originally used a space representation that included only the objects being laid out.

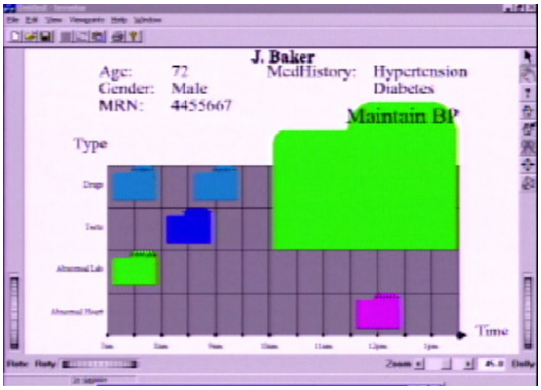
IMPROVISE was used to develop MAGIC (Multimedia Abstract Generation for Intensive Care) [37, 75, 76, 98], which is a system that automatically generates briefings of patient status after coronary bypass surgery. Figure 2.13(a) shows how MAGIC displays a timeline that presents a patient's surgery. Each of the folders in this interface contains information about the patient that the user may wish to explore in more detail. Enlarging a folder to fill the display or popping it up without regard to overlapping important objects in the scene can cause a disturbing loss of context. Instead of creating a single area dedicated solely to the display of enlarged objects, we have used the space manager to allow the system to query the available space to find a suitable largest empty-

space rectangle. For example, the query used to create Figure 2.13(b) finds the empty-space rectangle that can contain the largest possible rendition of the enlarged object at its original aspect ratio.

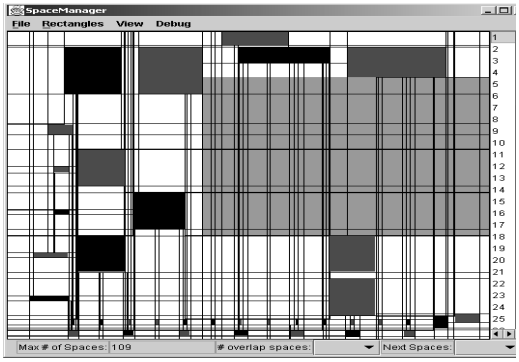
(a)



(b)



(c)



(d)

```
Method getLargestSpacewithAspectRatio(float aspectRatio){
    Rectangle largestSpace = null
    float area, largestArea = 0
    for each empty-space O in S.largestEmptySpaceList {
        if (O.height * aspectRatio >= O.width)
            /* use width to determine area */
            area = O.width / aspectRatio
        else
            /* use height to determine area */
            area = O.height * aspectRatio
        if  (area > largestArea){
            largestArea = area
            largestSpace = O
        }
    }
    return (largestSpace)
}
```

Figure 2.13 Space manager applied to automated information visualization system. (a) Folders on time-line contain data about patient's operation. (b) Visualization system uses space manager to position and scale selected folder to reach largest size possible at its original aspect ratio without overlapping important objects. (c) Space manager debugging tool shows dark full-space rectangles and edges bounding empty spaces used to calculate enlarged folder. Gridded timeline background and selected folder in (a–b) are deleted from spatial representation by visualization system to allow scaled folder to overlap them. Large light gray rectangle at right is largest empty-space rectangle selected by query to contain scaled folder. Because the folder has a fixed aspect ratio, it does not take up the entire width of the rectangle allocated to it. (d) Pseudo-code for query that selects largest empty space. Allocation strategy must also determine where to place object in selected largest empty-space rectangle.

The ability to delete objects from the representation is an important feature. For example, the system specifies that the objects in the time chart must remain visible, while the large gridded time chart background has less of a priority and can be overlapped. Therefore, as shown in the debugging display of Figure 2.13(c), the system temporarily deletes the time chart's background from the space representation that is used to calculate the placement of the enlarged folder. This allows the folder to overlap the background,

while not overlapping the time chart's contents and the folder's original position. Figure 2.13(d) shows the query written by a visualization system implementer using the space manager to select the largest empty-space rectangle in which to display the enlarged folder.

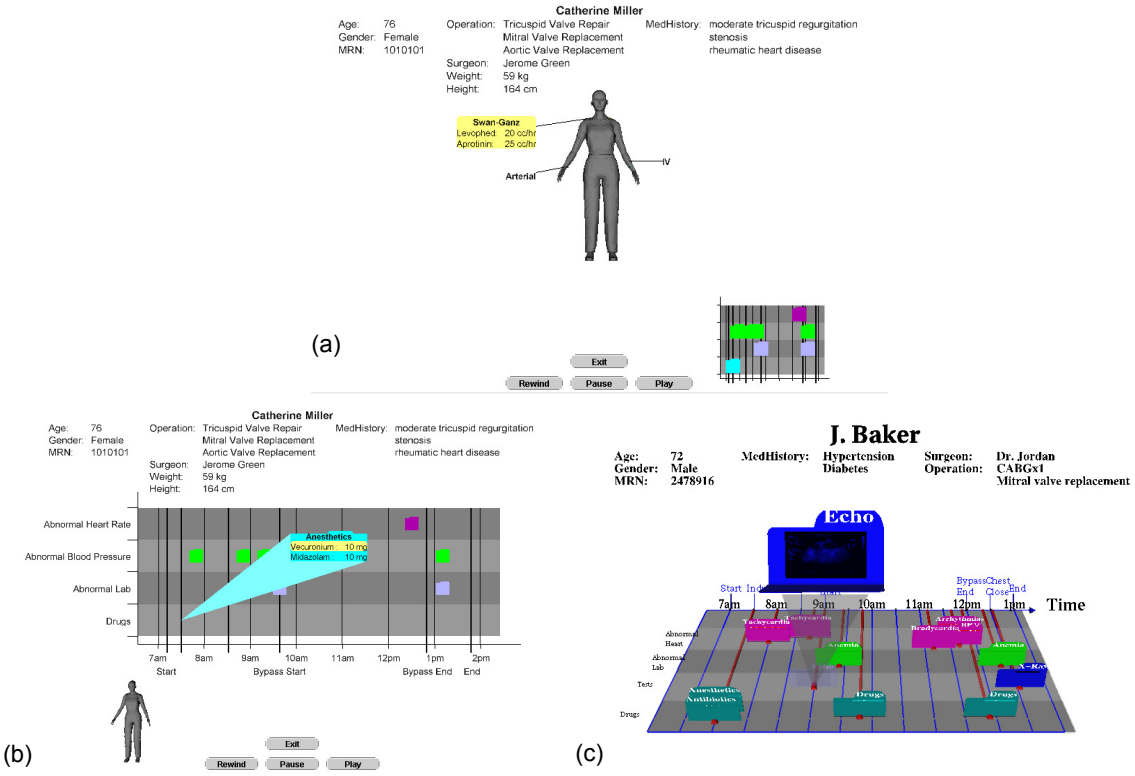


Figure 2.14 Space manager applied to automated medical information visualization system. (a) Structure diagram shows spatial related patient information. (b) Highlighted file folder in time chart automatically placed closest to original position with a given size. (c) Spatial allocation in 3D time chart uses the 2D projections of 3D objects.

Alternative queries could find the empty-space rectangle closest to the related object's original location that can accommodate a new object with some specified minimum scale factor. Figure 2.14 shows additional screen shots of the system in action. A structure diagram (Figure 2.14a) interactively shows information pertaining to physical positions of a patient's body [144]. Using the space representation, each piece of information is automatically placed close to the related body part without overlapping other important information, including the body itself. A leader line is used to show the relationship between the information and the related body part. We also show two more versions of the time chart using different perspectives of the coordinate frame, both in 2D (Figure 2.14b)

and 3D (Figure 2.14c). These examples illustrate how the enlarged folder can be graphically connected to its original position to avoid confusion. In both of these cases, when the user clicks on an element, or when the automatic presentation system decides to show more information, the element is enlarged to the closest place that can fit a rendition of the information without overlapping other important information. In the 3D version, each full-space rectangle is the upright bounding box of the 2D projection of its associated 3D object.

2.7.3 Internet browser

We have developed some web-based examples to show how our space manager can be used on the Internet for advertisements and popup word definitions. The main benefit for using our techniques inside an internet browser is that new images and text can be displayed without occluding anything else on the web page regardless of size and placement of the scrollable resizable browser window. Thus, information can be guaranteed visible at any time if there is enough space, no matter what portion of the web page the user is viewing.

Current web advertising techniques used today include popup windows and Flash [96] animations, which introduce information that overlaps other parts of the web pages and are often considered intrusive and annoying [118]. Currently, the main approach to ameliorate these problems is simply to prevent them from happening, using so called "popup blockers", which unfortunately eliminates the benefits of showing the information at all. By applying space management to web advertising, our goal is to display the same information inside the web page while avoiding the unwanted overlap when possible.

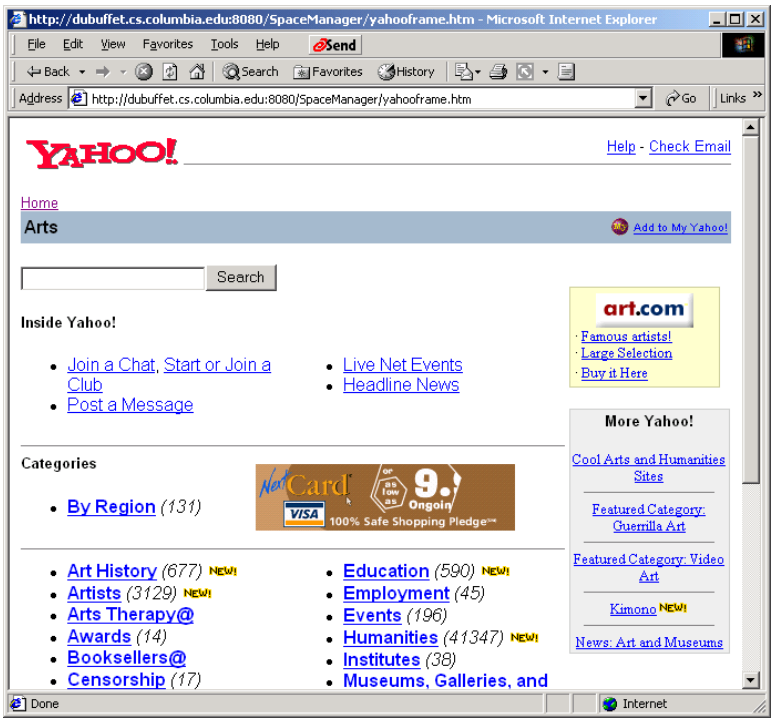


Figure 2.15 Space Management applied to an Internet Browser for advertising. The VISA advertisement is automatically placed in the middle of the screen without overlapping anything else on the web page.

Figures 2.15, 2.17, 2.19, and 2.20 show examples of using Space Management inside a web page. In these cases, the empty space of the entire scrollable region gets computed only when a page is initially loaded, or when the layout gets altered (typically when the width of the window changes). The mechanisms for acquiring screen space layout in JavaScript [1] are done using a function called "getClientRects()"* inside a script within a hidden frame, which gets a list of rectangles in pixels†. Also, since this function returns screen space values of the entire hierarchy of HTML objects, we filter out internal node objects that do not actually take up screen space (such as tables without borders and paragraphs). For security purposes, the space representation is calculated on the server to avoid public access to the code. This is implemented by first posting to the server the full-spaces which are accumulated from the hidden JavaScript frame. The server then processes the full-spaces to generate the space representation and returns the

* Only supported in Microsoft Internet Explorer 5

† We note that there are many browser dependent issues, including absent objects, such as bullet points, that are displayed on the screen but are not included. Current internet browsers are not designed to handle this. However, we hope that these problems will get solved in future versions.

results along with the logic to provide the interactive functionality. The code to query the results and place the information is run on the browser client in a new JavaScript script in the same hidden frame. We have tried variations of the type of information passed between client and server based on how much flexibility is needed versus how much functionality is constrained. For example, if the object sizes are known, the server can exclude sending the empty space rectangles that cannot fit any of the objects.

Once we have the empty space for an entire web page, we can use it for queries. Instead of using constant aspect ratios or sizes, as in the previous examples, the content in these applications, such as images and/or text, can have multiple renditions with different sizes and different aspect ratios. This makes it more difficult not only to determine whether there is a correct fit, but also how to determine which layout is best. For the predefined advertisement images, for example, we simply prioritize the banners that have the largest area. The queries that determine placement iterate through the available sizes in order of largest area until a space is found. We have tried many different strategies for introducing and moving these advertisements:

- fading in/out over time
- guaranteeing visibility inside the scrollable region
- animating within the visible scrollable region

These queries can occur after the recomputation of screen space, over a period of time, and/or while the user scrolls the page. Since advertising on the internet is an extremely complex issue [118], comprehensive user studies should be performed for further validation.

Chapter 3

View Management

The application of space management to 3D user interfaces, as described in Chapter 2, has limitations insofar that we determine the objects' projections without the capability to handle arbitrary visibility relationships and object geometry. Instead of dealing with 2D objects approximated by axis-aligned rectangles, we want to compute screen space of the projections of 3D objects on the 2D view plane. These projections can have arbitrary shapes on the view plane and depend on the visibility with respect to the point of view, projection parameters, as well as other objects in the scene that might obscure others. Our initial goal for computing the screen space of 3D projections was to enhance 3D scenes by determining layouts of overlaid information, such as labels and annotations. However, these layout strategies provide an excellent way to control avoiding overlap, guarantee visibility, and keeping an object within proximity of others.

Consider the problem of labeling objects clearly and unambiguously in a 3D scene, such as buildings in images of our campus. For example, Figure 3.1 shows the result of a simple layout approach that positions each label at the projected centroid of the object with which it is associated (Figure 3.1), and Figure 3.2 depicts a variant on this approach that suppresses any label whose object's centroid is not visible. These approaches result in overlaps and mislabeling because they do not take into account hidden parts of the buildings or conflicts in label placement. In contrast, each internal label in Figure 3.3, which is generated by an algorithm we explain below, appears within the visible projection of its building and no labels overlap. Figure 3.4 shows a simple 3D modeling utility that uses this algorithm to label the named objects in a VRML file, suppressing objects that are not visible.

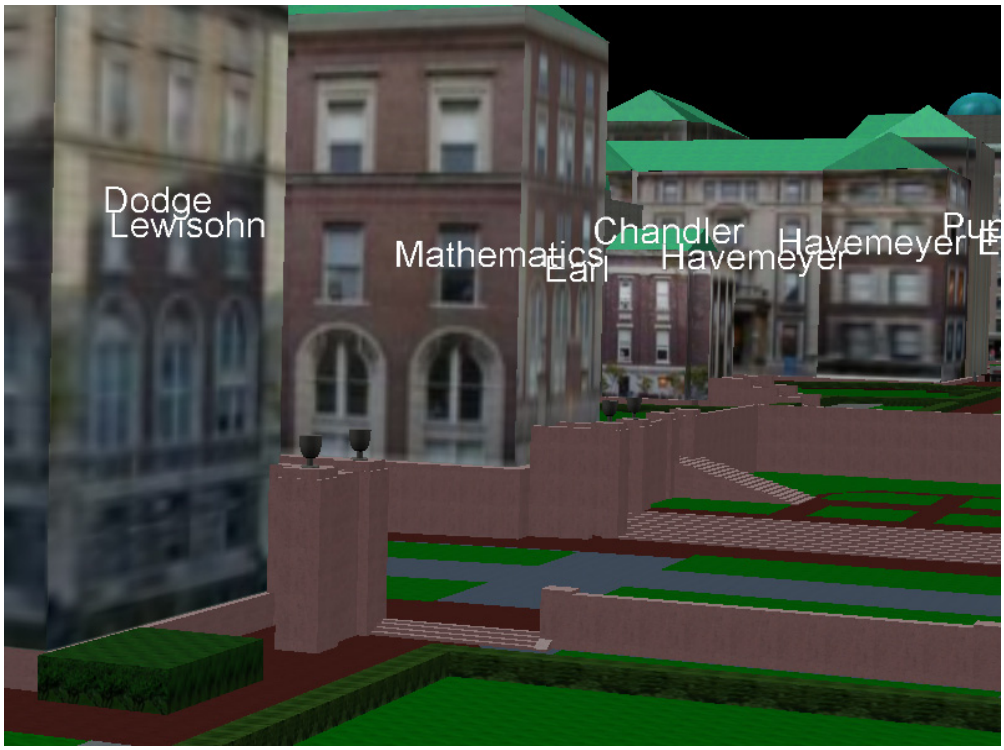


Figure 3.1 Area feature labeling. Naïve label placement at projected centroid of each building causes overlaps, and mislabeling because of buildings hidden partially or entirely.

Figure 3.2 Area feature labeling with the suppression of labels for buildings whose projected centroid is not visible.

Figure 3.3 Area feature labeling. Label placement using approach of Section 4.3.

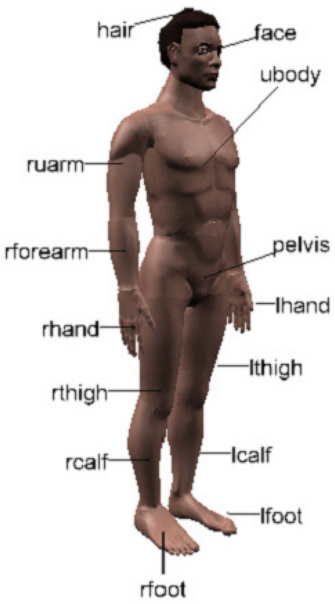


Figure 3.4 Labeling approach applied to named objects in VRML models.

We use the term *generalized view management* (Section 3.4) to refer to the techniques of managing what users see. We distinguish techniques that control what users see into three categories of view management: automatic, user-driven, and collaborative/distributed. *Automatic view management* [18] (Section 3.4.1) determines what a user sees based on the environment properties and relationships. *User-driven view management* (Section 3.4.2) allows users to choose what is seen at real-time by using predictable tools with expected behaviors. *Collaborative/distributed view management* (Section 3.4.3) manages what each user sees in a distributed environment based on information received from every other user, every display in the environment, and the environment itself.

3.1 Related work: Labeling and annotating

Researchers have often labeled and annotated objects in virtual and augmented environments without taking into account visibility constraints (e.g., [41, 43, 102, 128, 132, 141]). In contrast, graph layout algorithms [14] and label placement algorithms for point, line, and area features on maps and graphs [32] make layout decisions that have spatial dependencies and are therefore closely related to view management. Approaches to labeling point features typically explore a set of candidate positions arranged in a circle around each feature [10]. An objective function rates a solution's goodness, based on factors such as the overlap of labels with features and other labels, and the location of labels relative to their features. Christensen, Marks, and Shieber show that an exhaustive search approach to labeling point features is NP-hard [32].

In 3D user interfaces, it is possible to avoid some view management decisions by caching virtual objects out of the user's field of view (e.g., in a virtual tool belt or just above the view volume), and displaying them briefly when needed [99]. Filtering approaches, in general, can be used to limit the number of virtual objects being displayed [4, 77]. Nevertheless, there are many applications in which constraints on multiple objects need to be maintained, such as keeping related objects within relative proximity, assuring visibility, or avoiding occlusion.

A number of researchers have developed approaches for automatically avoiding 3D occlusion of selected objects. Some of these rely on controlling the viewing specification (e.g., [68, 110]), which is not possible in head-tracked environments in which the

viewing specification is slaved to the user's head. Others utilize illustrative effects, such as cut-away views and transparency [49] or line style [44, 78], without altering object geometry. Visual access distortion [29] moves objects away from the user's line of sight to a selected focus point by displacing each object perpendicular to the line of sight by a function of the magnitude of the object's distance from the line of sight. However, this approach does not actually guarantee the visibility of an object at the focus point: the size of an object's projection is not taken into account when determining how far to move it (e.g., a large object may still occlude the focus point after the move), and the summed displacements for an object that lies between lines of sight to multiple focus points may not move the object in a useful way (e.g., the projection of an object that lies along the vector average of two lines of sight may not move). In contrast, our view management approach can select positions for objects that guarantee that desired occlusion relationships with other objects are maintained.

3.2 Object properties and constraints

All of the objects in our scenes have properties, some of which may be marked as controllable or constrained by the user, the view management module, a tracker, or other components (e.g., a simulation or user interface). This means that there can be multiple sources of change, and these sources can control or constrain different properties of the same object. The view management module uses the controllable properties to help satisfy the constraints. We support a set of constraints that control the view plane layout, currently visibility, position, size, priority, and transparency of objects in a 3D scene. These constraints can be set explicitly by user interface components or by the user, or they can be defined to change based on other properties or relationships.

- *Visibility constraints* specify occlusion relationships on the view plane: those objects that a given object should not occlude, and those objects that it is allowed to occlude. Examples of the use of visibility constraints include:
  - Pavement and grass patches in the campus scene can always be occluded by other objects.
  - Objects can be overlaid by their own labels, but not by labels for other objects.
  - The collaborating user's face may not be occluded by any other objects.

- *Position constraints* specify the minimum and maximum distance to be maintained from:
  - Other objects: For example, the "speech balloon" of Figure 1.3 is constrained to be above and near an area feature (the avatar's mouth).
  - A point, area, or volume in a coordinate system, which may be screen-stabilized (relative to the user's head position/orientation) or world-stabilized (relative to the world).
  - The viewpoint: For stereo viewing, this distance will control how much the user's eyes need to converge to fuse the two separate images together.
- *Size constraints* specify a range of possible sizes. For example, labels are associated with a font size range, with preference towards the high end of the range when possible.
- *Priority constraints* specify priorities for objects. This allows the system to determine the order in which objects are included in the image, so that less important objects can be elided if adding them will violate other constraints. The guarantee of displaying objects also determines priorities between different layouts.
- *Transparency constraints* specify a range of object transparency values. For example, transparency can be modified to minimize the consequences of occluding other objects.

3.3 Computational view-management pipeline

Computing the visible portions of the projections of objects on the view plane is essential for satisfying most of the constraints described. Figure 3.5 shows a pipeline that specifies the order in which computations are done for view management [20]. To precisely maintain the specified visual constraints, this pipeline should get executed for each frame rendered. However, since some of these computations can take a considerable amount of time to compute, asynchronous execution of the pipeline for interactive applications, as in the Decoupled Simulation Model [125], can preserve interactive rendering frame rates at the expense of view management accuracy. (These tradeoffs might be preferable depending on the user's preference or task.) In this section, we discuss each pipeline step and give examples of how it is performed in our implementation.

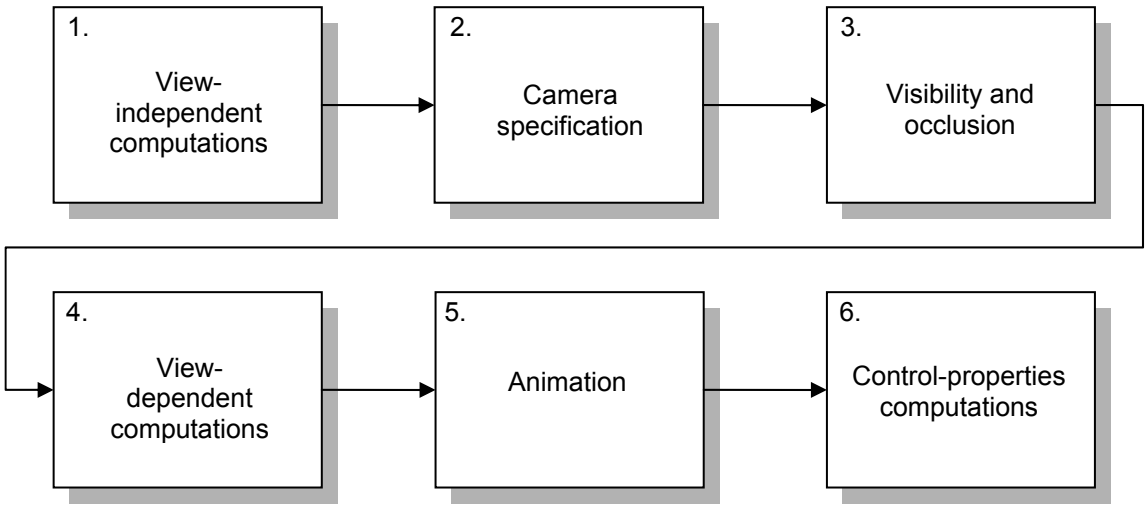


Figure 3.5 View Management Pipeline

3.3.1 View-independent computations

Computation that is not dependent on the viewing specification (step 1) can happen either at the beginning of the pipeline or asynchronously. Some applications, such as ones that use head-worn displays, must render graphics based on real-time tracker input. To minimize the system's response time (or error caused by asynchronous computation latency), we want to minimize the time between reading the tracker data (i.e., to determine the viewing specification in step 2) and setting object properties (step 6). To do this, we execute all computations that are not dependent on the viewing specification outside of these steps. These computations can include most types of logic in a graphics system, such as layout, content management, and interaction. Also, data structures and variables that are needed further down the pipeline (e.g., spatial representations and accumulation variables used in projection calculations) can be initialized at this time.

3.3.2 Camera specification

For each 3D object in the scene, we need to determine projection parameters (step 2) that include viewpoint (two for stereo), orientation, and other viewing parameters (e.g., field of view and clipping planes). For a physical simulation, all objects in the same co-

ordinate system will typically have the same relative viewing specification. Depending on the type of graphical application, determining the viewing specification for each frame can be different. We want to support applications in which the viewing specification changes interactively and unpredictably, as well as applications in which the view follows a predetermined path. Some interactive applications determine the user's viewpoint and orientation by using input from 3D trackers, along with transformations based on the position and orientation of the physical trackers and the display relative to the user. Other interactive 3D applications are controlled using 2D direct manipulation devices, such as a mouse. Applications that use predetermined viewpoints, such as video and animation playback, can be manually planned or use automated assistance [68, 110]. Determining the viewing specification may involve screen-space computations to automatically choose camera placement [25].

The situation-awareness aid we describe in Section 3.4.2 determines a viewing specification for a world in miniature that depends on the user's head pitch, which is computed from the viewing specification for the physical world. In this AR example, two viewing specifications, one for the physical world and one for the world in miniature, are computed from tracker input for rendering.

3.3.3 Visibility and occlusion

After determining the viewing specification, the constrained objects' visibility, occlusion and depth values are computed (step 3) to help satisfy the constraints discussed in Section 3.2. To satisfy these constraints efficiently, we use 2D space representations, described in Chapter 2, to encode the visible portions of the projections of 3D objects on the view plane and another instantiation of this 2D space representation to encode the complementary portions of the view plane on which no projections lie (i.e., the empty space). Each such representation of an arbitrary spatial region consists of a list of rectangles that are axis-aligned, are possibly overlapping, and are the largest such rectangles that lie wholly inside that region.

To satisfy some constraints, such as avoiding occlusion, we can search for a position for an object within the empty-space representation. For other constraints, we need to compute the visible-space representation of one object, or of a combination of objects. We have used two visible-surface-determination algorithms to generate these representa-

tions: one based on a Binary Space Partitioning Tree [56], and one based on a hardware-accelerated *z*-buffer [8, 30].

3.3.3.1 Visibility using a Binary Space Partitioning tree (BSP)

We perform visible-surface determination for view management by sorting the upright extents of the objects' projections in visibility order. We use the BSP tree algorithm [56] to efficiently produce the visibility order for an arbitrary projection of a scene. A BSP tree is a binary tree whose nodes typically represent actual polygons (or polygon fragments) in the scene. Because we need to determine the visibility order for objects, the BSP tree planes separate objects rather than individual polygons [107, 124]. We choose these planes using the heuristics of Thibault and Naylor [136]. Although BSP trees are often used for purely static scenes, dynamic objects can be handled efficiently by adding these objects to the tree last and removing and adding them each time they move [33].

Determining the visible portions of each object

For each frame that needs visibility computations, we determine the visible portions of the scene's objects by using our 2D space representation, which is described in Chapter 2. As objects are processed in a front-to-back order using the BSP tree, the space representation is populated with full-space rectangles that represent the 2D projections of the 3D scene objects. Thus, the empty space represents the 2D screen space of the view plane that is not populated by any previously processed object. By evaluating the empty space before an object is processed, the visible space for that object is determined to be the empty space within the object's 2D projection (i.e., the space that has no projections of previously processed objects). We obtain the largest visible space representation for each object by clipping the largest empty space rectangles that overlap the object's projected area and eliminating redundant rectangles that are wholly contained in any others. This visible space representation provides an efficient way to find suitable locations for annotations relative to the visible area of each object's projection.

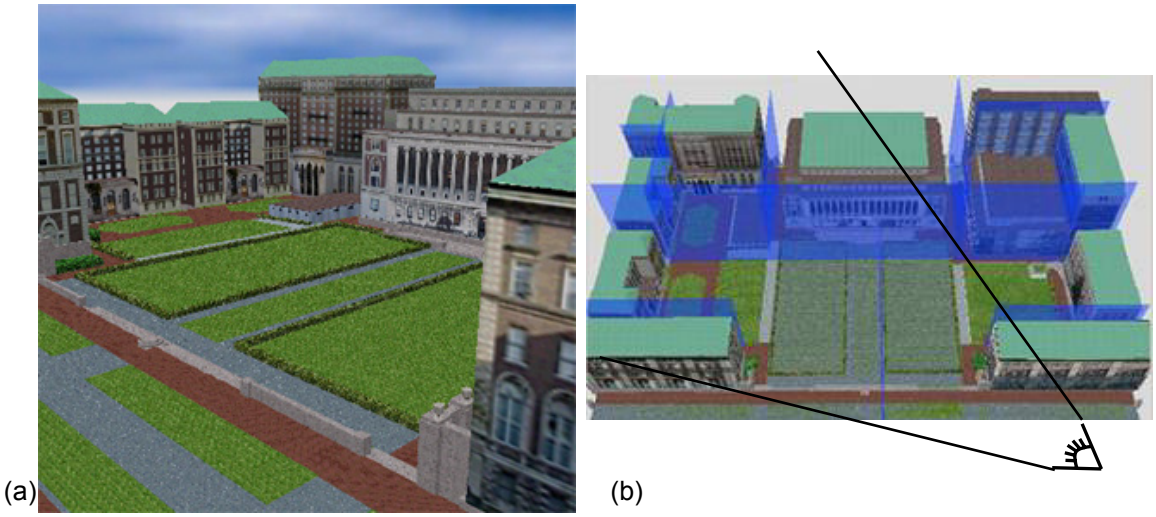


Figure 3.6 (a) 3D view of a campus. (b) Bird's eye view of the campus scene, which includes blue semi-transparent splitting planes for the BSP tree and an eye icon with lines, which indicates the user's view frustum in (a) with respect to the model.

Figure 3.6(a) shows an example of a 3D view of a campus, along with the bird's eye view of the scene in Figure 3.6(b), which includes blue semi-transparent splitting planes for the BSP tree and an eye icon () with lines that indicate the user's view frustum. For each node obtained from the BSP tree in front-to-back order, the 2D upright extent of its projection is computed by projecting each vertex and adding the encompassing axis-aligned bounding rectangle to the space representation. Each row from top to bottom in Figure 3.7 shows how each node's projection is processed in order to compute each node's visible space, from front-to-back relative to the user's viewpoint. Initially, the 2D spatial representation of the view plane is empty (Figure 3.7a, upper right).

When processing each node, the upright extent of its 2D projection is intersected with the members of the current list of largest empty-space rectangles, as shown in Figure 3.8. This can be performed efficiently because we maintain the largest empty-space rectangles in a 2D interval tree [119], allowing an efficient window query to determine the members that intersect the extent. (The intersection operation itself is trivial because all rectangles are axis-aligned.) The intersection yields a set of rectangles (not necessarily contiguous), some of which may be wholly contained within others; these subset rectangles are eliminated. The result is a set of largest rectangles whose union is the visible portion of the node (Figure 3.8b). In our example, the visible space of the first two nodes is computed as their entire projections (Figure 3.7a–b). This is because there are no previ-

ously processed projections that overlap each node's projected area (dashed outline), which both lie completely within the 2D empty space (Figure 3.7, right column).

Each subsequent object processed (Figure 3.7c–g), have projections that overlap previously processed projections, and thus, their visible space includes only a portion of the projected area. Consider the third node processed (Figure 3.7c): a zoomed image in Figure 3.9(a) shows the upright extent of its 2D projection as a dashed outline overlaid onto the 3D projection (left) and overlaid onto the 2D space representation (right). The empty space is evaluated within the dashed outline, as exemplified in Figure 3.8, and results in the one dark grey rectangle (right) that represents the node's visible space. The fourth node's projection (light blue in Figure 3.7d) is evaluated within the empty space and the visible space is determined to be the two dark grey rectangles shown on the right in Figure 3.9b. The rest of the nodes are processed in the same way (shown in Figure 3.7e–g), and the resulting empty space representation (Figure 3.7h) can be used in view-dependent computations (Section 3.3.4), for objects that must avoid the projections of all of the objects in the scene.

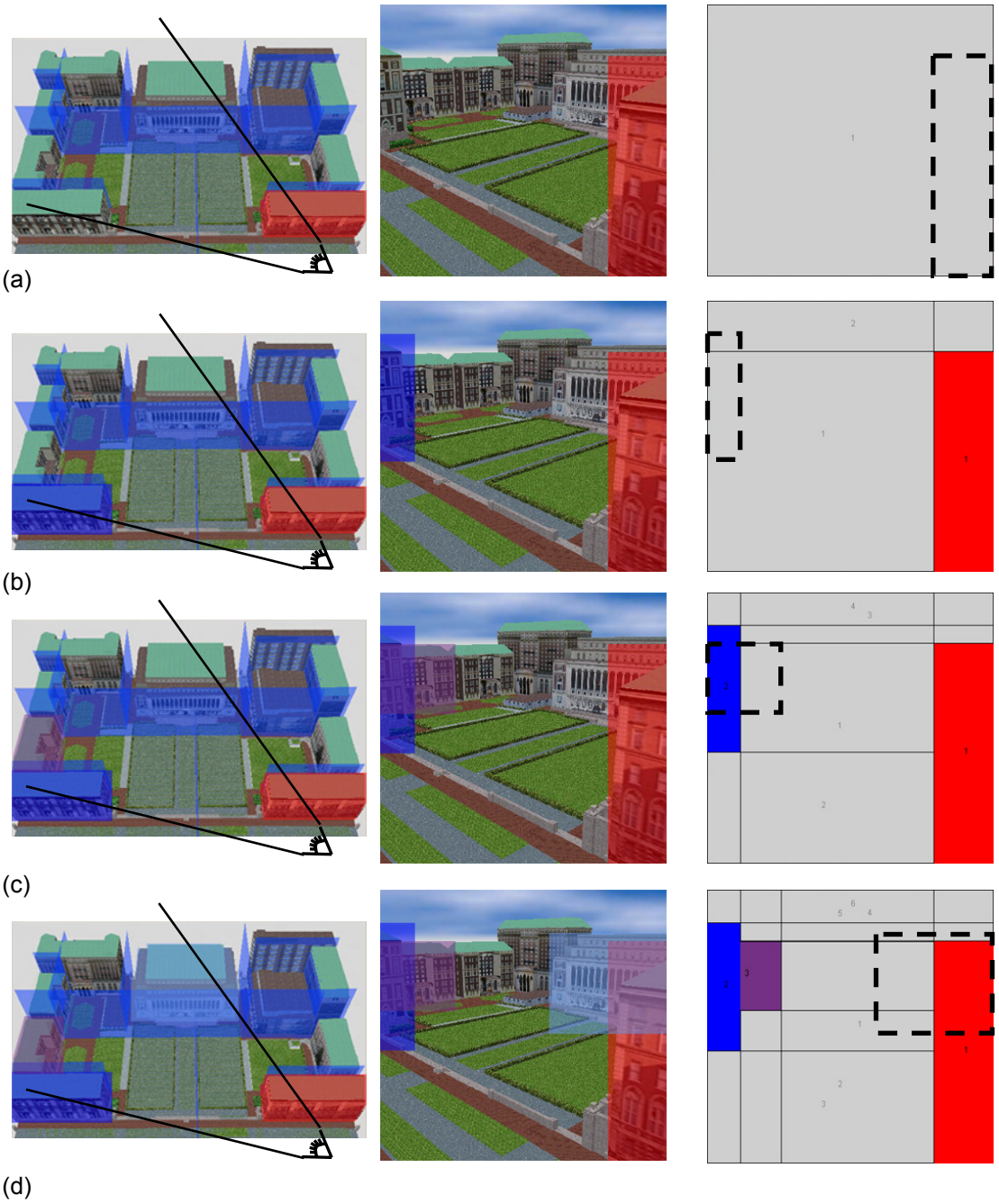


Figure 3.7 Computing screen space of each object from front-to-back relative to the user's view-point (continued on next page). Each row, from top to bottom, shows the processing of an additional object. The current BSP tree node being processed is additionally highlighted in the bird's eye view (left column), its projection highlighted in the 2D view plane (middle column), and its projection outlined with dashes in the 2D spatial representation (right column), which is used to determine visible space for the object by computing the empty space within the BSP tree node's projection. After a projection has been processed, it is added as full-space to the 2D spatial representation.

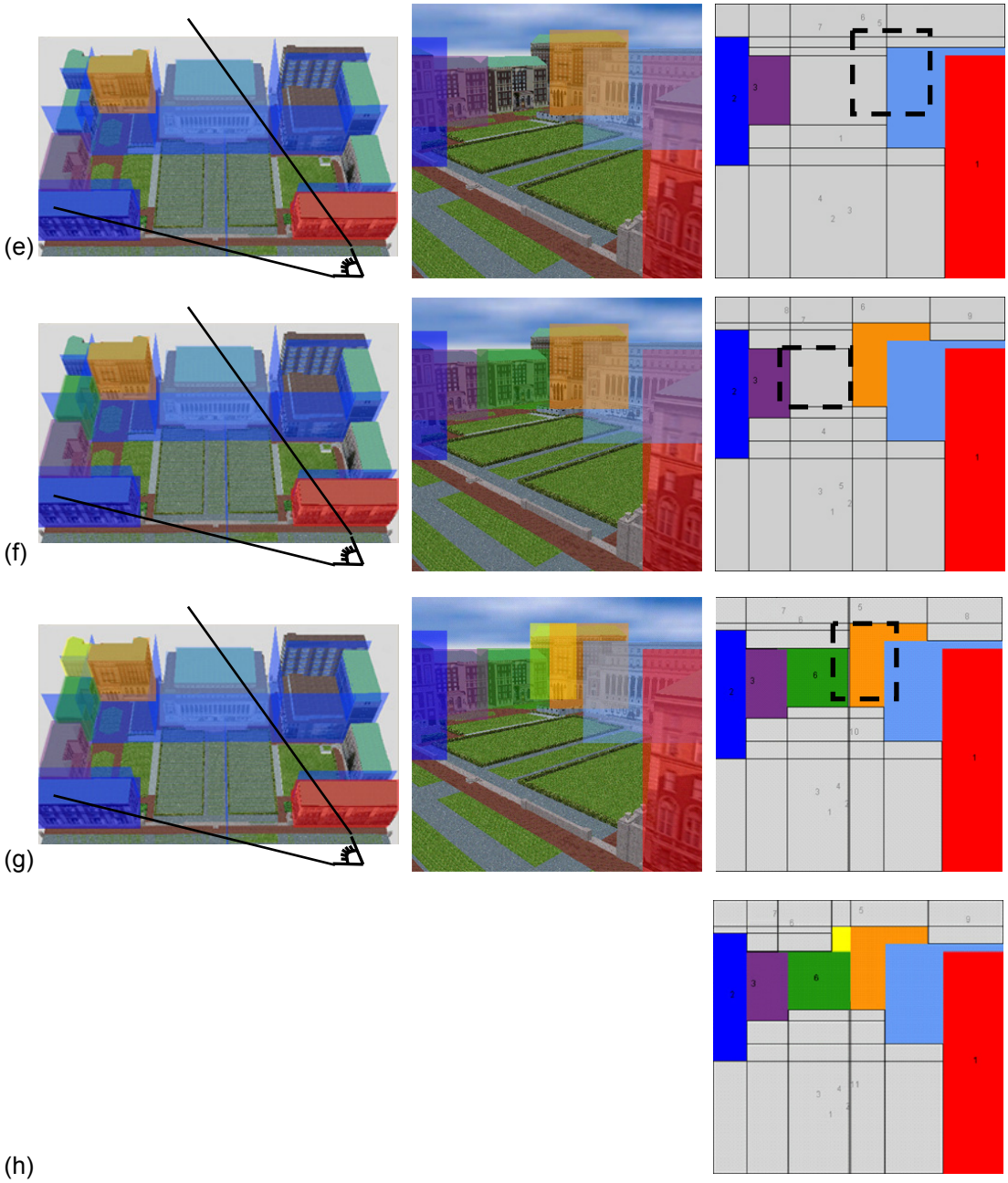


Figure 3.7 (continued) Computing screen space for the remaining objects from front-to-back relative to the user's view point. (h) Screen space representation after the last object is processed.

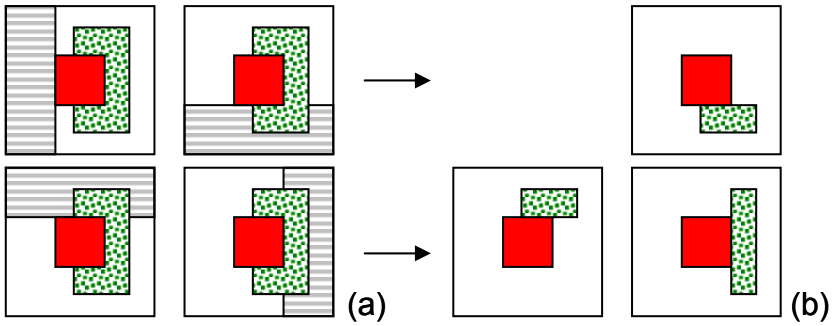


Figure 3.8 Adding a new object to the view-plane representation. (a) New object extent (dotted green) is added to original object (solid red), shown separately with each of the four original largest empty spaces (striped light grey). (b) Resulting visible largest spaces of new object extent (dotted green).

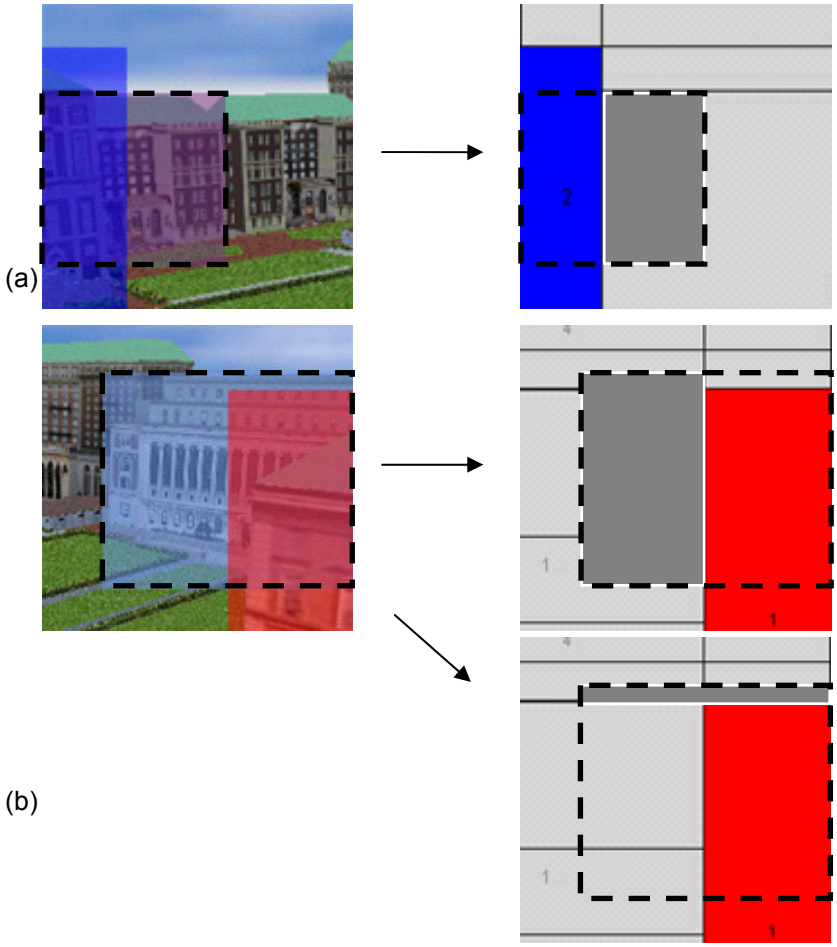


Figure 3.9 The projections (dashed outline) of the third (a) and fourth (b) visible BSP tree nodes (from Figure 3.7c–d), which are overlaid onto the 3D projection on the left, are processed. The empty space is evaluated within the area of these projections to result in the largest visible space rectangles (dark grey), shown on the right, that represent the visible spaces for the nodes.

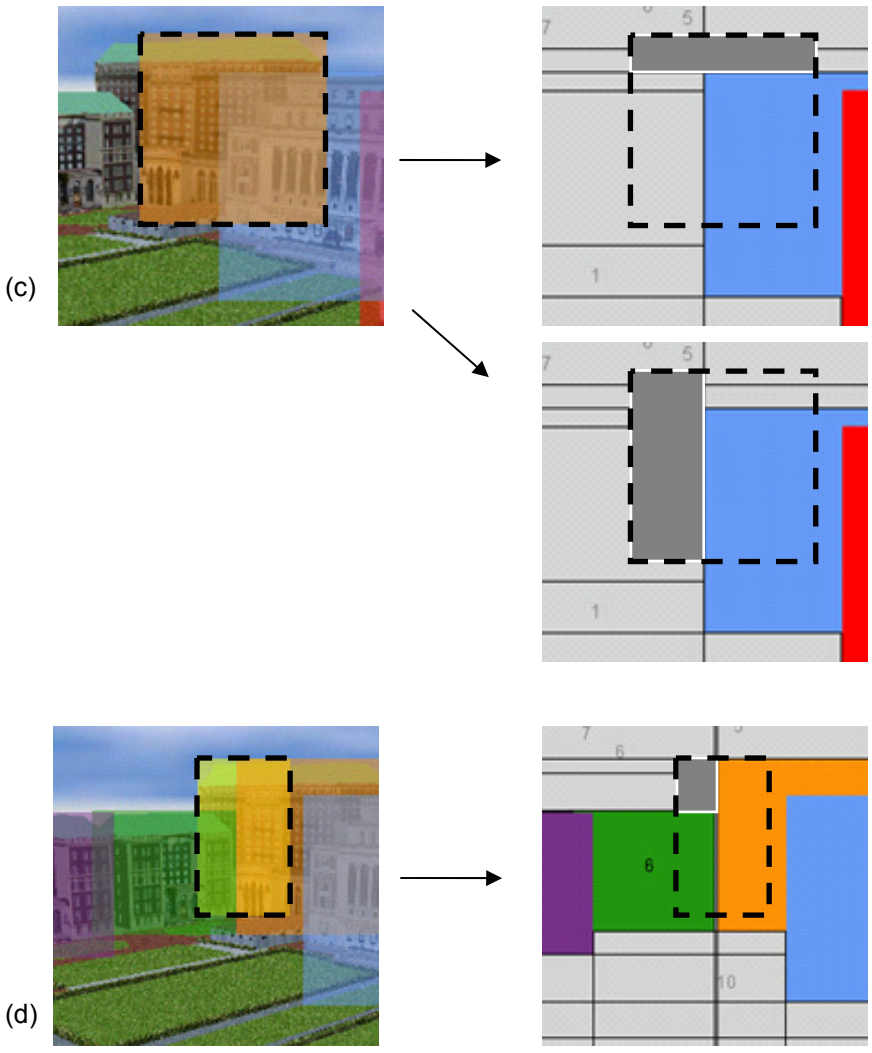


Figure 3.9 (continued) The projections (dashed outline) of the fifth (c) and seventh (d) visible BSP tree nodes (from Figure 3.7e and g), which are overlaid onto the 3D projection on the left, are processed. The empty space is evaluated within the area of these projections to result in the largest visible space rectangles (dark grey), shown on the right, that represent the visible spaces for the nodes.

Some objects are represented by a single BSP tree node, while others are split across multiple nodes. Consider Figure 3.10, where objects are placed so that it is not possible to create a plane that will partition the scene without splitting one or more objects. In this case, we continue to use the heuristics of Thibault and Naylor [136] to minimize the amount of splitting in the BSP tree. However, there are certain objects, specifically concave objects, which often do not get split well using the heuristics we use. This can result in large screen space approximations for concave objects, including empty space, which can inappropriately increase the area allocated to them on the view plane.

To address this, we create splitting planes manually to assure that the BSP tree has multiple parts for some of the objects. This allows our system to support concave objects, which could alternatively be supported automatically by using convex decomposition algorithms [90, 97].

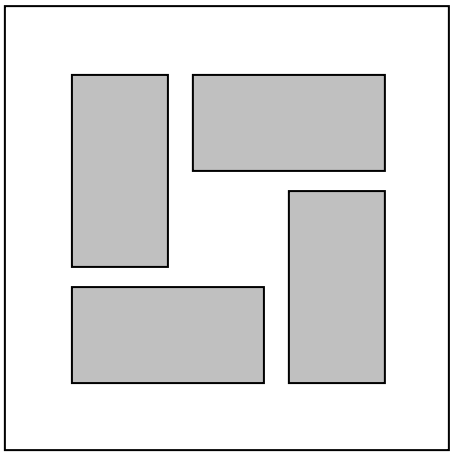
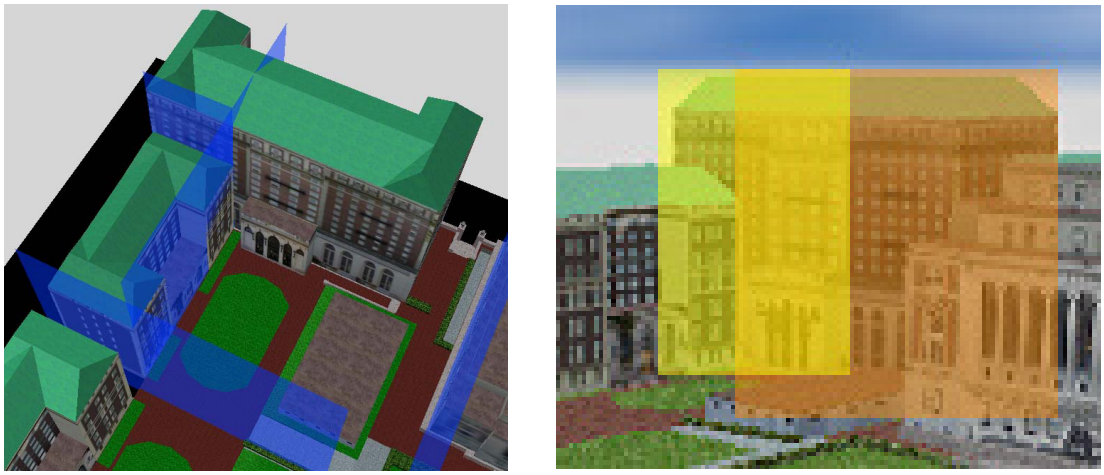


Figure 3.10 Object layout (in 2D or in 3D as a plan view of a scene) that requires a BSP plane to split at least one of the objects. There is no plane that will partition the group of objects without cutting an object in parts. This requires us to split an object in the object BSP tree into multiple parts that are used to compute 2D screen space individually. These visible parts are then combined together to form the visible space of an entire object.

Figure 3.11 shows an example of an object split into multiple parts by a BSP tree plane, along with the approximation of each part's projection transparently overlaid. For each part of an object, the largest visible rectangles are computed, as described previously in this section, and are shown for these specific parts in Figure 3.9(c–d). Once the visible rectangles have been computed for each part of an object, they must now be combined to obtain the visible space rectangles for the entire object. To do this, each part of the same object is coalesced into an accumulated list of visible rectangles, which represents the screen space for the entire object. In the next section, we describe how the visible space rectangles of two parts of an object are coalesced. This algorithm can be used sequentially on multiple parts to obtain one unified visible space representation for an entire object.



(a) (b)

Figure 3.11 A building that has two BSP tree object parts. (a) Building in the top part of the image is split by the plane. (b) Projections of both parts are shown in a close up of a portion of Figure 3.6 and are used to determine the visible space for this object. The yellow rectangle represents the projection of the left building part, and the overlapping orange rectangle represents the projection of the right building part.

Coalescing lists of largest visible rectangles

To coalesce two lists of largest visible rectangles, we merge each visible rectangle from one list with the visible rectangles of the other list. This operation is similar to the space management deletion algorithm (Section 2.4), in which the largest empty-space rectangles that are revealed when a covering rectangle is deleted are combined with all adjacent largest empty-space rectangles. Unlike the deletion algorithm, we combine lists of largest visible rectangles incrementally, which means that some pairs of rectangles being processed may overlap. To handle these cases, the function that combines two largest visible rectangles must support overlap, creating up to two new largest visible rectangles, as shown in Figure 3.12. Thus for each rectangle in one list, we find all ways in which it can combine with the rectangles in the other list, and remove those that are enclosed by any other, to determine the largest visible rectangles in the union of the two lists. Figure 3.13 shows an example of coalescing the two projected BSP tree object parts from Figure 3.11. Each part consists of the visible space rectangles, which are the results from processing the projections shown in Figure 3.9(c–d). The horizontal yellow striped rectangle from the left part is combined with the two vertically orange striped rectangles from the right part, which results in the three circled rectangles that represent the largest visible spaces for this entire object.

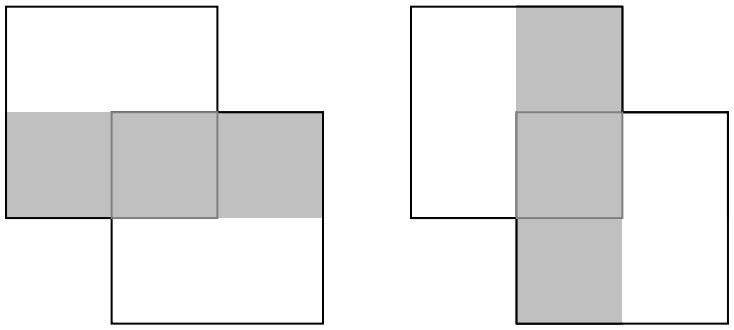


Figure 3.12 Two overlapping largest rectangles (white boxes) create an additional largest rectangle (grey boxes) in each dimension in which their combined extent is greater than that of either original rectangle.

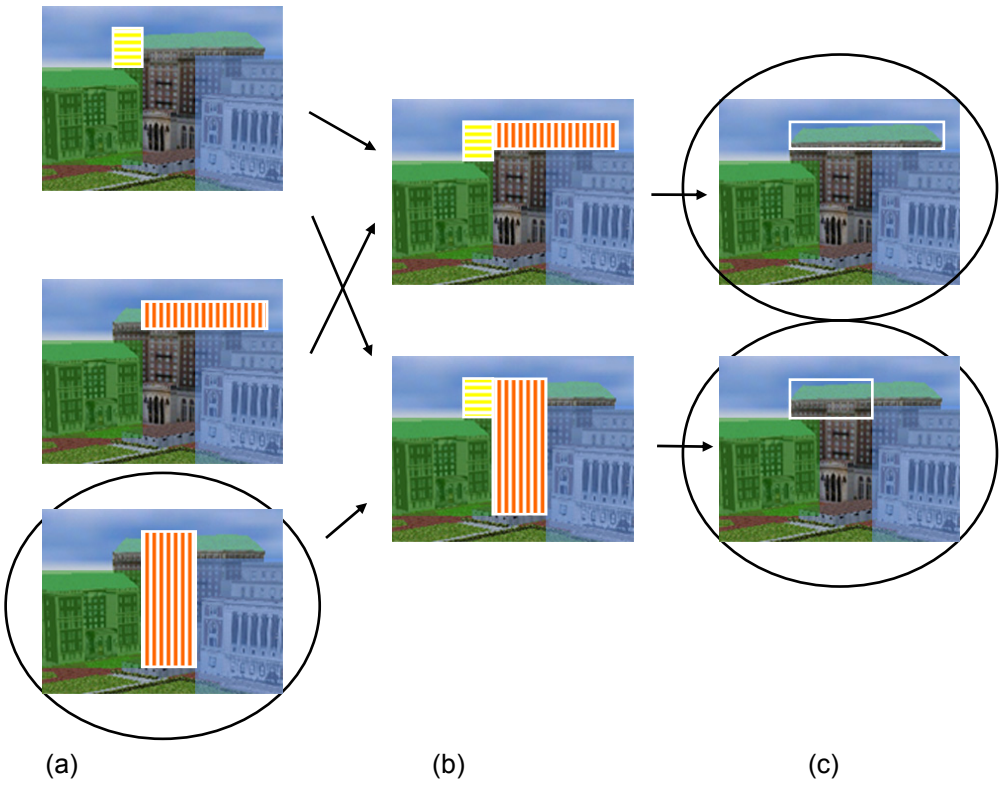


Figure 3.13 Coalescing lists of visible space rectangles of two BSP tree parts (from Figure 3.11) of the center building, which is partially occluded by the buildings highlighted in semi-transparent green and light blue (left and right). (a) Initial visible spaces of both parts include the horizontal striped yellow rectangle from one BSP tree part and two vertically striped orange rectangles from the other BSP tree part (results from computations in Figure 3.9c–d). (b) The rectangle (horizontal striped yellow) from one part's visible representation is combined with the two adjacent rectangles (vertically striped orange) of the other part. (c) Results of the two combine operations. The three circled rectangles represent the largest visible spaces of the building. The two original rectangles in (a) that are not circled are not included in the largest visible spaces because they are wholly contained within at least one resulting circled rectangle.

The resulting representation supports a variety of layout strategies. Each object is represented by the largest rectangles contained within the visible parts of the object's projection. Thus, an appropriate position and size can be selected for a new object within any object's visible projection. This makes it possible to perform the equivalent of area feature labeling [101] in a 3D environment where objects can partially occlude each other; in this situation, we need to determine which parts of an object are visible, and select that one that would be best to contain a label.

When laying out a new object, a rectangle can be selected that lies within one of a set of objects. Alternatively, the coalescing algorithm can be used to merge the largest rectangles of multiple objects to create a meta-object that corresponds to the union of those objects. This enables the system to lay out a new object whose projection spans the space of multiple objects. Objects can either be coalesced as they are added to the representation or after the representation is built. If both the original objects and the coalesced objects are desired, the coalesced objects can be created in a second space manager representation.

Since full space objects are represented the same way as empty space, both can be coalesced; for example, to treat certain objects as empty space (e.g., the grass, pavement, and sky shown in the campus view of Figure 3.6). Note that coalescing full space with empty space is *not* the equivalent of simply not processing the objects when constructing the representation: some objects might obscure parts of others. For example, in our model, the grass and pavement obscure underground infrastructure that would be treated as visible if the grass and pavement were not added.

3.3.3.2 Visibility using z-buffer

Alternatively, we can compute the 2D spatial representation of the visible regions of objects in a 3D scene using a *z*-buffer algorithm [30]. By rendering each object in a distinct color with the current viewing specification, we can use the frame buffer as an *object buffer* [8], in which each pixel's color encodes the object visible at that pixel.

We use a sweep algorithm to convert the resulting pixel representation to the spatial representation that is needed to satisfy our constraints (illustrated in Figure 3.14 with pseudo-code shown in Figure 3.15). Since it is likely that adjacent pixels will have the same value, we evaluate each pixel, starting at one corner of the buffer, across the largest

dimension of the image, keeping track of the last position at which the pixel value has changed and the current object value. Once a change is detected or the end of the scan line is reached, a rectangle is created from the last to the current position (with unit height) and combined into the current object's visible-space representation. The current object and last position are then reset, and the algorithm continues. Since the only rectangles that can be adjacent to a rectangle on the current scan line are those combined from the previous scan line, a 2D interval tree of all rectangles from the previous scan line is kept for fast lookup. Any rectangle that is not a result of a combine operation from the last scan line is considered part of the final result and does not need further processing. The set of largest visible space rectangles for each visible region consists of all resulting rectangles, including the rectangles that have been combined on the last scan line.

Figure 3.14 shows the Z-buffer visibility sweep algorithm at the stage of processing a new rectangle (outlined in dotted black and grey) within a small pixel area, where the numbers associate each square pixel to a related object in the scene. Each pixel is evaluated in the sweep direction (left to right on rows from top to bottom) to find new rectangles for objects, which are found when the next pixel's associated object is different from the previous pixel's associated object or the end of row is reached. In this example, the new rectangle in Figure 3.14(a) is processed because the next pixel to the right of the pointer is labeled "3". To process the new rectangle, it is added to visible space associated with the same object labeled "2", which is highlighted in grey in Figure 3.14(a). The visible space for each object in the scene is accumulated as a list of largest empty space rectangles throughout the sweep process.

In order to add the new rectangle, it is compared with each rectangle in the same object's visible space representation that overlaps it in the sweep direction (in this example, the comparisons are executed on the four solid grey rectangles shown on the left in Figure 3.14(b–e)). The grey highlighted rectangle in Figure 3.14(b) is not adjacent to the current row (solid double line) and is therefore added to the visible space of the object, since it is no longer needed for processing. (Note: this rectangle could have been detected as no longer needed for processing at the end of the previous line, at an expense of extra processing at the end of each row.) The grey highlighted rectangles in Figure 3.14(c–e), are adjacent to the new rectangle, thus, are combined with the new rectangle in the vertical direction to create the three new grey rectangles highlighted in Figure 3.14(f–

h), respectively. The new combined rectangle Figure 3.14(g) is a subset of the rectangle highlighted in Figure 3.14(f) and is therefore is discarded (indicated by an "X"). After rectangles in Figure 3.14(c–d) are processed to produce the proposed rectangles Figure 3.14(f–g), they can no longer be adjacent to any more processed space and therefore are added to the visible space of the object (shown in the pseudo-code in Figure 3.15 lines 34–37). All other computed rectangles Figure 3.14(e, f, h) and the new rectangle (outlined in dotted black and grey) are kept for this object for further processing.

Note that we must be careful to avoid the problems that can occur when two adjacent polygons in the same object fail to share a vertex on a shared edge, potentially causing missed pixels [51]. This can result in a large area being improperly split into a set of smaller ones. To prevent this, we add the necessary extra vertices to polygons in our models.

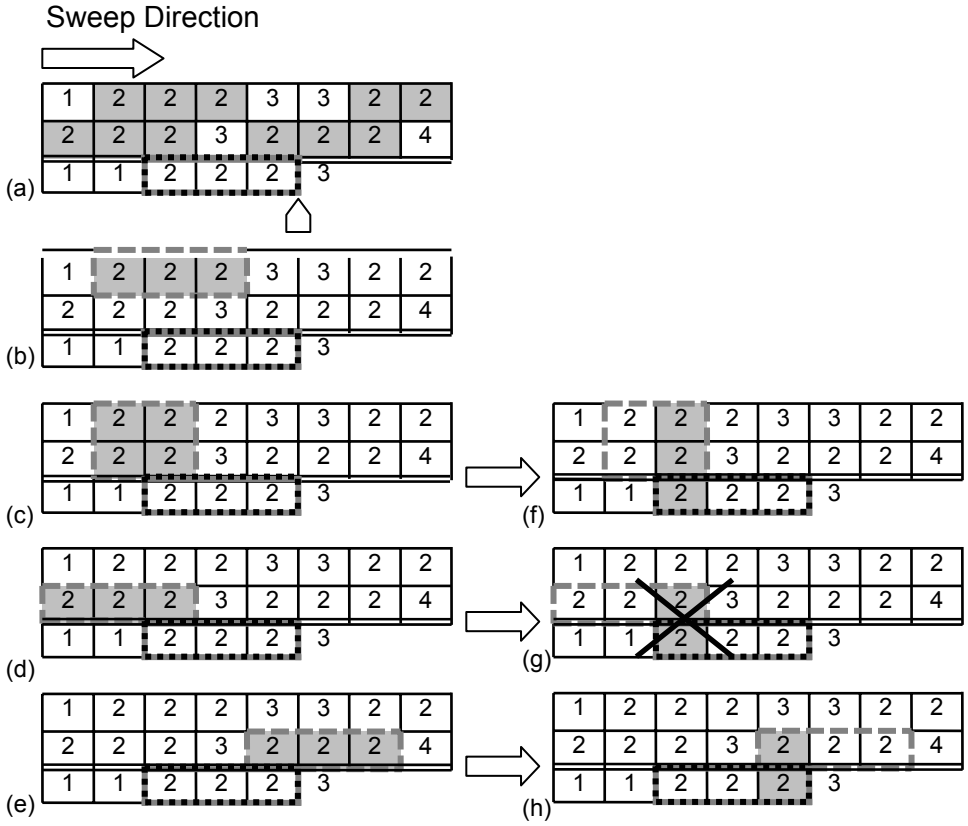


Figure 3.14 Z-buffer visibility sweep algorithm at the stage of processing a new rectangle (outlined in dotted black and grey) that adds to the visible space for the given object labeled "2". (a) Visible space for object "2" highlighted in grey. Four highlighted grey rectangles in (b–e) are processed and if adjacent, are combined with the new rectangle. The results of these combinations consist of the three highlighted grey rectangles in (f–h). Rectangle in (g) is a subset of the rectangle in (f) and is therefore discarded (indicated by an "X"). After the entire Z-buffer is processed, the accumulated final visible space representation for each visible object consists of a list of largest space rectangles.

```
computeViewPlaneRectanglesForEachObjectFromObjectZBuffer(){
1    IntervalTree2D objectViewPlaneRepresentation[] = new IntervalTree2D[N]; /* resulting representation */
2    IntervalTree objectXIntervalTrees[] = new IntervalTree[N];/* IntervalTree in X direction for each object */
3    int objectBuffer[] = renderAndCreateObjectBuffer();
4    int place=0, startX, startY, lastX, lastY, previousPixelObjectValue = -1;
5    for (y = 0; y<height; y++){
6       for (x=0; x<width; x++){
7          place++;
8          int currentPixelObjectValue = objectBuffer[place];
9          if (previousPixelObjectValue <0){
10            /* No object in previous pixel */
11            startX = lastX = x;
12            startY = lastY = y;
13            previousPixelObjectValue = currentPixelObjectValue;
14         } else if (x!=(width-1) and previousPixelObjectValue== currentPixelObjectValue){
15            /* pixel is same as previous, increment last and continue */
16            lastX = x;
17            lastY = y;
18         } else {
19            /* change is detected or last pixel in row,
20               combine region to all adjacent spaces for object */
21            newRectangle = new Rectangle(startX, startY, lastX+1, lastY+1);
22            possibleNewRectangles = new List();
23            possibleNewRectangles.add(newRectangle);
24            eachRect =
25               objectXIntervalTrees[previousPixelObjectValue].windowQueryInX(newRectangle);
26            for each rectangle R in eachRect {
27               if isAdjacentInY(R, newRectangle){
28                  newCombinedRectangle = combineSpacesInY(R, newRectangle);
29                  if isEnclosedBy (R,newCombinedRectangle)
30                     objectXIntervalTrees[previousPixelObjectValue].remove(R);
31                  removeAnyRectangleInEnclosedBy (possibleNewRectangles, newCombinedRectangle);
32                  if (newCombinedRectangle not enclosedby any in possibleNewRectangles)
33                     possibleNewRectangles.add(newCombinedRectangle);
34                  if (R.maxX <= lastX + 1) {
35                     objectViewPlaneRepresentation[previousPixelObjectValue].add(R);
36                     objectXIntervalTrees[previousPixelObjectValue].delete(R);
37                  }
38               } else {
39                  objectViewPlaneRepresentation[previousPixelObjectValue].add(R);
40                  objectXIntervalTrees[previousPixelObjectValue].delete(R);
41               }
42            }
43            for each rectangle R in possibleNewRectangles
44               objectXIntervalTrees[previousPixelObjectValue].add(R);
45         }
46      }
47   }
48   /* all resulting spaces in interval tree are spaces for objects */
49   for each object O from 0 to N
50      objectViewPlaneRepresentation[O].addAll(objectXIntervalTrees[O])
 }
```

Figure 3.15 Pseudo-code for the sweep algorithm to convert discrete object buffer representation to our space representation.

3.3.3.3 Visibility computation performance

Since most current graphics hardware implements the *z*-buffer algorithm, the time needed to compute the pixel representation is low. However, the conversion to our representation is more computationally intensive. Generating the object buffer at a lower resolution than used for rendering the user interface accelerates the algorithm at the expense of accuracy.

In the BSP tree approach, accuracy can also be traded for performance. Since objects are approximated by the upright rectangular extents of their BSP tree nodes, better approximations can be produced if more nodes are generated at the cost of increased computation.

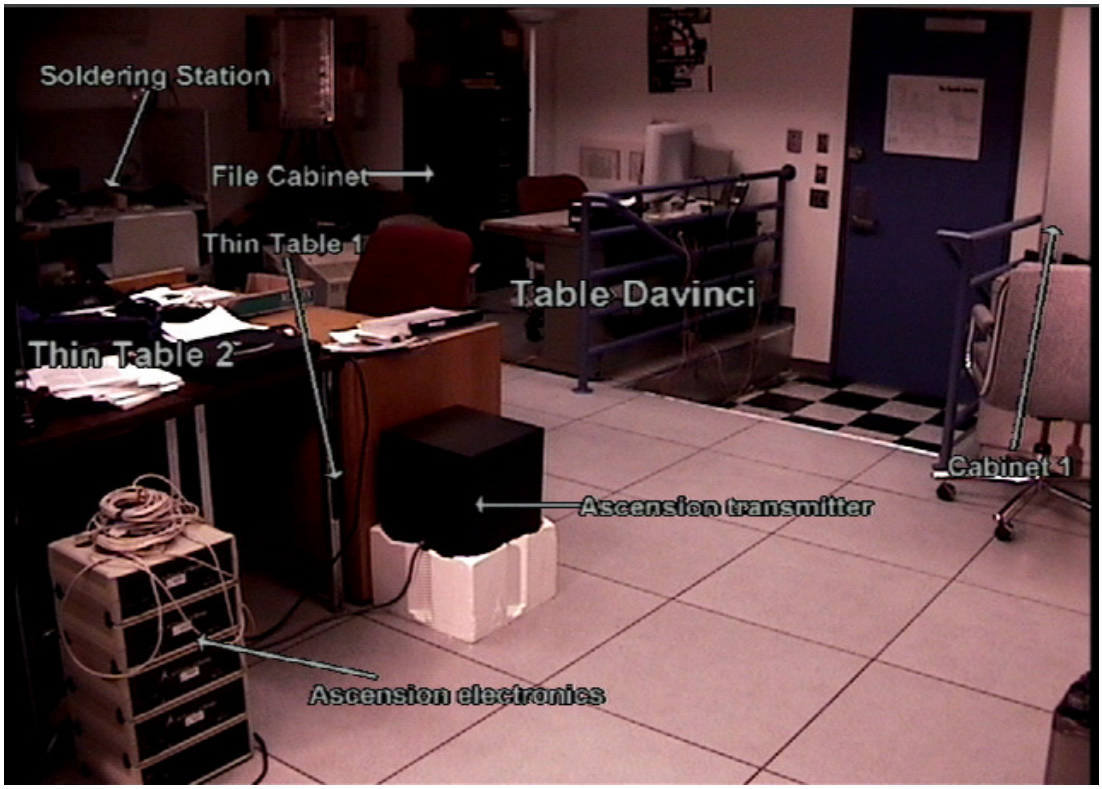


Figure 3.16 Browsing an augmented reality environment.with labels placed relative to the visible portions of the objects in the scene.

3.3.4 View-dependent computations

The resulting visibility representations, consisting of the largest rectangles of single objects, multiple objects, and empty spaces (i.e., background), can support a variety of layout strategies. Queries on these visible regions (step 4) can be designed for choosing appropriate positions and sizes for new objects within these areas. For placing text, this is equivalent to map area feature labeling [117]; however, in a 3D environment objects can partially occlude each other. In this situation, we can place the label relative to the visible portion of the related object. The visibility representations also allow proximity queries that can constrain the resulting position of an object to be inside a region and closest to a point or another projected object. These queries are satisfied by searching the region's list of largest visible rectangles to find a resulting rectangle that best fits the criteria needed, similar to the queries used in space management.

We describe here some examples of strategies used for satisfying specific constraints for application tasks such as labeling, annotating, and placing other information. The results from these computations can be used to place objects directly or as input to animation computations (Section 3.3.5) that produce the final placement.

There are many cases in 3D user interfaces in which the application needs to communicate information to the user. For example, Figure 3.16 shows an augmented reality environmental browsing mode in which the system tries to label a number of objects that have been determined to be of interest to the user, making sure that the labels are not ambiguous. In this case, we use the two-tiered approach that was applied to Figures 1.2–1.3, creating both internal and external labels, and ensuring that no label occludes any part of an object except the object it labels. In contrast to Figure 1.2, Figure 3.16 is imaged using the video see-through display, in which video mixing hardware is used to combine a camera view of the real world with synthesized graphics.

3.3.4.1 Internal label placement

For each object to be labeled we determine the largest visible rectangle internal to the object's projection (similar to the query shown in Figure 2.13) that can contain the largest copy of the object's label, given a user-settable font and size range, and taking into account an additional buffer around the label to keep adjacent labels from appearing

to merge. This buffer can also help control the amount of the related object's projection that is blocked by the label if the object has a high priority. For efficiency, the dynamic size of a label is approximated by using its aspect ratio at a certain font size, texture mapped to a polygon, and then scaled, based on the result of the query.

3.3.4.2 External label placement

If no visible space for an object is large enough, the object will not be labeled internally, but could instead be labeled externally. External labels can be processed in any desired order. We currently use the front-to-back order; however, since the allocation algorithm is greedy, if an importance metric were available for labels, it could be sorted instead. To lay out an external label, the system queries the empty-space representation for a rectangle that can contain the label within a user-settable range of sizes, and within a user-settable distance from the object. If the label is allocated within the empty space, it will neither be occluded by nor occlude any other object. If no such rectangle can be found, then the label is not allocated.

Since external labels are potentially ambiguous, we also generate a leader-line arrow from the label to the interior of its object. Providing the leader line helps mitigate possible misidentifications. Because internal labels occlude parts of the object that they label, we also support tagging an object to indicate whether or not certain parts should be internally labeled (e.g., internal labels may be suppressed for faces). Each external label is added to the spatial representations as it is created, so that objects added later do not occlude it.

Figure 3.17 Image taken directly through an optical see-through head-worn display of restaurant and related popup information

3.3.4.3 Other placements

Applications may need to show additional information about objects in the scene (Figures 1.2–1.3, and 3.14). This information can be placed external to related objects, much like external labels, possibly with different size and aspect ratio constraints.

Some information, such as the agenda located on the left side of Figure 1.2, does not refer to any objects that are visible in the scene. To place the agenda on the screen so that it doesn't overlap any important objects, the empty-space region is queried after all other annotations are placed. There are also situations, as the AR neighborhood restaurant guide of Figure 3.17, where there is not enough room to place the information. In this case, we detect the limited space by the negative results of external placement. Another query is used to place the popup information so that it avoids overlapping the restaurant, but could overlap other objects, such as the building above it. As last resort, if this query fails to result in a placement, which can happen if the object is very close to the user, if the user's field of view is small, and/or the annotation has a large size, a reserved placement is used, typically close to the bottom and side of the screen.

3.3.5 Animation

Thus far, we have described our system without taking into account temporal continuity. However, if the layout of objects in sequential frames is computed independently, discontinuous changes in object position and size occur, which may be annoying in some applications. Therefore, in some real time applications, we need to take into account decisions made during previous frames when determining current placement (step 5). First, we introduce hysteresis in the discrete state changes that can take place (Section 3.3.5.1). Second, we try to make sure that objects being laid out occupy roughly the same position relative to their defining object (or to their screen position if screen-stabilized) as they did in the previous frame (Section 3.3.5.2). Third, we interpolate between certain kinds of discrete changes (Section 3.3.5.3).

Based on informal user feedback, when the viewing specification is controlled by a mouse, animation is typically not preferred—when the user stops controlling the view, the viewing specification stops changing, but the annotations would keep moving if animation were used. In contrast, when the viewing specification is controlled by a head tracker, it is always changing, if only slightly, potentially resulting in drastic changes in the results of our layout queries between frames. Thus, to avoid jitter and excessive movement of annotations, it is necessary to provide animation and temporal continuity.

3.3.5.1 State hysteresis

There are several situations in which objects may change state, resulting in a discrete visual change, such as from an internal label to external one, or from being displayed to not being displayed. Some UIs use three constraints on the size of an object being displayed: minimum size, maximum size, and preferred size (e.g., Java 2D [63]). As shown in the state diagram of Figure 3.18, we borrow this approach, modifying the definition of minimum size by defining both an absolute minimum size (absMin) and a sustained minimum size (min) to avoid oscillation by minimizing jumps between states when the size is close to the boundary conditions.

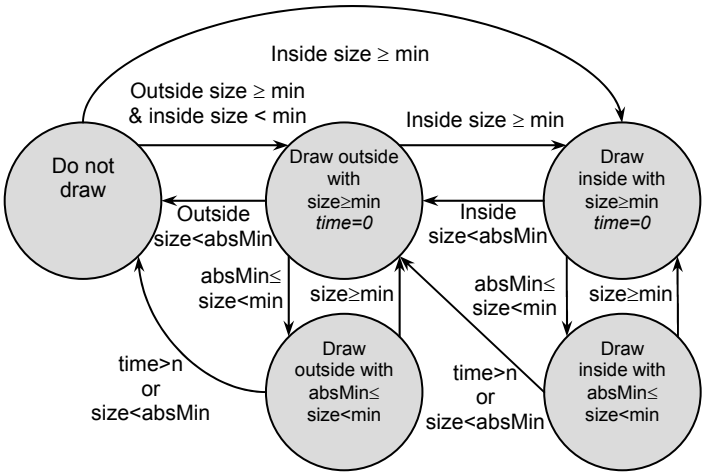


Figure 3.18 State hysteresis in area feature labeling. Area features are preferentially labeled inside rather than outside. Directional arrows mark the criterion that changes the label from one state to another. Labels are promoted from left to right through comparison with the minimum size (min) and demoted from right to left through comparison with the absolute minimum size (absMin). The time interval (time) measures how long a label has been in the same state, and can influence the change of a label's state when greater than a time duration (n).

For this technique, an object's visual size is guaranteed to be the sustained minimum size at least once within a settable time interval *n* or else it will not be shown. The system displays the object only when there is enough space for the sustained minimum size, and removes it when it is below the absolute minimum size. Furthermore, if the object is already being displayed and there is only enough space for an object within the absolute and sustained minimum sizes for time > *n*, the object is removed. Otherwise, the object is displayed at the largest possible size no greater than its maximum size. The state diagram of Figure 3.18 employs similar tests to make an additional distinction between placing an object inside (preferred) vs. outside another object, which we use for displaying area feature labels. The values selected for *n* and the difference between the absolute minimum and sustained minimum sizes help avoid visual discontinuities.

3.3.5.2 Positional stability

We address the problem of stabilizing the annotations' positions using two techniques that can also be used simultaneously for optimal results: 1) using the previous placement and visibility information as input to the queries for determining the next placement, and 2) computing the placement intermittently based on screen space. Both of these techniques intend to minimize the changes in placement over time and are similar to traditional key frame animation [86]: the results of these computations are consid-

ered key frames that are used for determining placement destinations, while interpolation (Section 3.3.5.3) computes the animation between the key frame placements ("in-betweening"). Instead of introducing movement, this technique results in the annotations being continuously rendered at the key frames for positional stability, while in-between frames are rendered minimally (only when movement between key frames is necessary).

Using previous placements and visibility information

To compute whether the current frame is a "key frame" for a particular object, it is necessary to find out whether the computed position (Section 3.3.4), which is considered the optimal placement for the current frame, has moved from the object's relative position in the last computation. To do this, an additional operation, computed in pipeline step 4, finds the closest current layout to the previous layout using the current visibility information and previous placements (Figure 3.19). For example, if the label was an inside label in the previous frame, a proximity query would be used on the visible region of the object. However, since the annotation's related object could have moved because of view specification change (e.g., due to head movement), the last placement is not valid for finding the closest layout. Instead, we compute the change relative to the projection of the related object by transforming the last frame's placement from the 2D coordinate system of the projected object in the last frame, to the 2D coordinate system of the projected object in the current frame, where the centroid of each 2D rectangle is its own origin. The result, transformed back into pixel coordinates, is used as the parameter to query for the closest relative placement from the previous frame inside the current frame. This result is compared with the optimal layout and is used if both results are the same within a threshold; if they are different, we set the placement relative to the previous placement, start a timer and, if the optimal and closest fail to coincide after a set amount of time, the optimal position is used and the timer is reset.

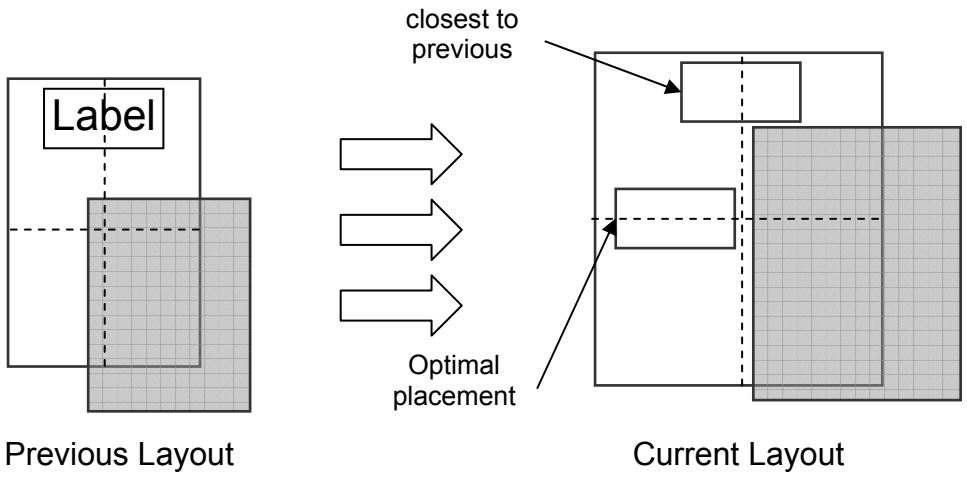


Figure 3.19 An example of how the previous layout (left) is used to determine the current layout (right) for an inside label. The gray rectangle is an object that partially occludes the object that is labeled. The dotted lines represent the 2D reference coordinate system of the projected object which is used to determine the closest placement to the previous frame.

Stabilizing screen space placement in 3D

We introduce a new way of determining annotation placement by dispersing the same screen space calculations temporally, stabilizing the annotations in 3D between calculations, and interpolating between the results by combining 2D and 3D techniques. There are two main benefits from using this technique: minimizing computation for performance and positional stability. Instead of computing visibility, visible space representations, and placements every frame, we recompute these intermittently based on when the placements need to be updated.

After the screen placement has been computed, as described in Section 3.3.4, the label's 2D center, along with arrow pointer tip (if shown) is projected from the 2D screen space back into the relative 3D scene coordinate system. In order to do this, a depth value needs to be used. Depending on which type of visibility computations has been done (Section 3.3.3), it is sometimes easier to accumulate depth values within the visibility calculations step than it would be to compute depth values when needed (i.e., for the BSP tree calculations, each vertex is projected into screen space, which includes depth in screen space). Depending on the application, the depth values used include the closest, farthest, middle, and depth of the centroid's projection. However, from our experience, these decisions only influence the quality of the placements when the camera specifica-

tion changes drastically. And in these cases, if the camera movement is drastic, but then stops, the labeling mechanism corrects itself on the next iteration. If the camera movement is continuous, as in a precomputed video, it is better to compute screen space placements frequently and at every frame.

Determining when to compute the screen space placement is difficult because it requires a delicate balance between the amount of processing and the accuracy (or quality) of the layout. Essentially what this hybrid strategy allows us to do is to disperse screen space computations and only use them when necessary, given their high computation requirements. We compute the placements at regular intervals (e.g., every second). This produces good results, and saves computation time that is proportional to the interval length between computations.

There are also discrete events that require immediate placement based on the application. When an annotation changes its relative coordinate system (from physical to WIM, as in Figure 1.4), we immediately compute the placement to allow the transition to occur. However, when an object becomes visible, it is not clear whether we should immediately compute the placement, since there is a chance a placement will not be found, or the placement will not be optimal (e.g., an external label for an object that is becoming more visible).

To determine the best strategies to use in these situations, a closer understanding of the application is needed, including the type of interface, the kind of information shown, and the purpose for showing the information in the interface. For example, if the interface is an interactive application and the annotation is a label that is of low priority, it might be better to wait until the label can fit inside the visible portion of the object, while lowering computations and eliminating unwanted movement that might occur. However, if the application is a video and it is important to show the annotation, then the strategy might detect the visibility change (at a computational cost, since visibility information would be calculated every frame) to determine immediate placement inside or outside the visible portion of the object.

Using this hybrid layout strategy lowers the need for computing visibility every frame. However, like the example described, there might be certain annotations that determine placement sporadically, causing visibility computations to occur irregularly.

Since performance is a priority, it is useful to consider synchronizing the placements of annotations to occur at the same frame to minimize the visibility computations.

These layout strategies present a number of these kinds of issues regarding the tradeoff of accuracy vs. performance. Some of the decisions result in subtle changes, while others have more obvious effects.

3.3.5.3 Interpolation

To minimize the negative effects of the jumps between annotation placements, we move and scale annotations between the placements using interpolation. We use multiple interpolation techniques in 2D and 3D, including a new hybrid 2D/3D version of positional interpolation that is used in conjunction with the new layout strategies described above. In this section, we discuss the different interpolation techniques we have used, their associated behaviors and how they can be used in applications to achieve optimal results.

Linear interpolation

Simple linear interpolation can be used for changing the 2D scale of the annotations and the 3D reference points (both for annotations and pointers) in the hybrid screen space placement (Section 3.3.5.2). When computing interpolation, since both of the start and ending values have been computed in the past, the interpolation starts when the ending value has been computed. This requires the time duration to be chosen, typically based on when the next key frame can be computed. If this is known, as in the time-dependent hybrid strategy, then the time duration is used for interpolation. However, if it isn't known, another interpolation based on a distance threshold (next section) is used.

For 2D positional interpolation, when interpolating the raw 2D placements of the key frames, the results are not satisfactory since camera movement (rotation/translation) is not taken into account. When camera movement occurs, labels appear to be "floating" on the view plane trying to catch up to the correct placement. To account for this problem, we interpolate the 2D placements relative to the projection of the related object (regardless of visibility), similar to how the key frames are generated when using previous placements and visibility information (Section 3.3.5.2). By using the related object as reference, even when the object moves drastically on the screen (both translation and

scale/distance), the annotations reside close to the projections of the objects with reasonable results.

Distance threshold

There are many situations we have described in which we want to interpolate values, both position and scale, but do not know the time durations since the destination values can be changed at any given time. In these cases, instead of interpolating between the values, we define the maximum distance an interpolation can move for each frame, or a *distance threshold*. For each frame, we compute the vector and distance between the current position and the destination value. If the distance is less than the threshold, the next value simply becomes the destination value. If it is greater than the threshold, the new value is moved from the current value along the vector (towards the destination) by the distance of the threshold value. Thus, if the destination value changes at any time, we maintain continuity in the position of the annotation while continually imposing movement towards the optimal placement. Pseudo-code for this computation is found in Figure 3.20, and is executed in every frame in which the destination is different than the current position. As an alternative, the distance threshold could be set relative to the length of the last frame's time duration or relative to a predicted time duration of the current frame to assure the constant speed of the object's movement.

```
/* function returns the current placement for the next frame */
Point computePlacementUsingThreshold (Point previousPlacement, Point destination)
   Point movementVector = destination – previousPlacement;
   double distanceBetween = movementVector.length();

   if (distanceBetween > DISTANCE_THRESHOLD){
        movementVector.normalize();
        movementVector = movementVector * DISTANCE_THRESHOLD;
        return (previousPlacement + movementVector);
   } else {
        return (destination);
   }
}
```

Figure 3.20 Pseudo-code for computing the distance an object can move every frame by limiting it by a threshold, which is used to uphold consistant interpolation for continuously changing values.

Hybrid 2D/3D interpolation

This new strategy applies a 2D distance threshold to a 3D position interpolation. When constraining the threshold distance in 3D, the resulting distance on the screen depends on how far it is away along with its angle relative to the view plane. These unwanted dependencies will give inconsistent perceived interpolation results. Instead, we define the distance threshold in 2D and compute the distance and vector of the projected 3D points in the 2D image plane. Once we determine the next position based on the 2D distance threshold, we project the 2D placement back into 3D to maintain the resulting 3D position. This results in more consistent 2D behavior than 3D position interpolation.

3.3.6 Control properties computation

When all view management pipeline computations have completed, the properties of the rendered objects must be updated. Depending on what graphics package is used (e.g., OpenGL, Java3D, or OpenInventor), different variables and functions are set in order to control the associated graphics system. Furthermore, certain graphics systems require different control variables. When creating a view manager, the results of the computations are stored separately from the graphics-system–dependent variables. This allows the view manager to not only handle any graphics implementation, but also run asynchronously, as described at the beginning of Section 3.3, while maintaining consistent results. Therefore, this step cannot run asynchronously, since all controlled object properties need to get set before rendering begins, assuming that any object can affect another object's projection.

3.3.7 Pipeline implementation

The pipeline is implemented exactly as described: the sequential processing of the individual steps (Section 3.3) in the rendering loop. Figure 3.21 shows pseudo-code of the view management computations that occur for each execution of the pipeline. For each pipeline step, parameters in the objects, as described in Section 3.2, control what computations should be included in the rendering loop and in what order they should be done. There are certain objects that are used in different steps of the pipeline that are connected. For example, if a controllable object needs to be placed relative to a 3D ob-

ject, then the 3D object's visibility is computed (step 3) before the placement of the controllable object is determined (step 4).

The state diagram (Figure 3.18) is implemented with state values within the controllable objects. Depending on the state, different values are computed and different comparisons are made to determine whether the state is changed, as shown in Figure 3.18. For example, if the annotation is currently placed inside but below the minimum size (lower right state in Figure 3.18), the system first determines the inside placement. If the size enlarges above the minimum size, then the state changes to the upper right state. If the size shrinks below the minimum size, or if the accumulated time within this state is longer than the time specified, then the state of the annotation changes to the upper middle state. For each state, the computations and comparisons specified in the diagram are executed. If the state changes, then the associated computations are added to what is computed within the rendering loop, while the old state computations are removed.

```
                                                                    Steps
for each frame {
    for each object not dependent on the view                       1. View-independent
        compute layout for object                                      computations

    set camera specification (tracking, pre-processed,
                                computed, mouse)                    2. Camera
                                                                       Specification
    compute visibility and occlusion,
        which consists of the view-plane representation             3. Visibility and
        for all objects needed to satisfy constraints                  occlusion

    for each controllable object in priority order {
        determine object position and size, based on constraints
        if object has visibility constraints involving controllable 4. View-dependent
            objects that have not yet been processed then              computations
                add object to view-plane representation
    }
    for each object that needs animation
        compute current placement for object                        5. Animation

    for all shown objects                                           6. Control-properties
        update control properties dependent on implementation          computations
}
```

Figure 3.21 View Management pseudo-code. Pipeline steps (Section 3.3) shown on the right are related to the code shown on the left.

Ordering is also an important part of designating priorities within our view management system. If an object is deemed important, we determine its placement first. We implement this by using variables within the controllable objects that set the order in which its placement is determined. This is somewhat of an ad-hoc method, where priorities between a few objects, such as the other users head in Figure 1.2, and the hand-held display in Figure 3.22(b). A better way to manage these constraints, which we have not had time to implement, would be to use a constraint hierarchies [27], that could automatically determine the ordering of the computations between many objects within our pipeline based on their constraints and relationships. As the view management becomes more complex, this will be important to implement properly.

3.4 Generalized view management

So far, the main focus of view management (Chapter 3) has been described as architecture, including the descriptions of the view-plane representation and the view-management pipeline. Using view management in applications consists mainly of controlling users' viewing priorities, and how they change over the run-time of an application. In general, users want information from the system based on their situation and what they are going to do next.

We characterize view-management decisions into three types: automatic, user-driven, and collaborative/distributed. We will use this taxonomy of view management to distinguish different graphical techniques that help users make their own decisions in AR and VR. These characterizations are based on how view management priorities are controlled to determine what users see. We discuss these three types of view management, show examples of how they are used to assist users in specific applications, and discuss how each category will influence new applications in 3D user interfaces.

3.4.1 Automatic view management

We define *automatic* view management as the layout decisions computed directly from environment properties, relationships, or users' goals. This includes the automatic generation of presentations based on an understanding of users' situation and needs [143]. Examples of strategies described so far include satisfying specific constraints for application tasks such as labeling, annotating, and placing other information. In these

cases, the system automatically shows information based on the user's situation: Figure 3.22(a) show a situation where the user is browsing 3D objects in our lab, and the automatic view management system tries to place labels relative to all objects the user can see. There are other situations in which the user is only interested in certain types of objects. For example, a network administrator who is coming to our lab to fix problems is only interested in the computers and their locations. In this situation, settings in the system, controlled either manually by the user or from user preferences, determine that the labels for the computers should be shown.

Figure 3.23 shows some screen shots of a system that uses physical devices to control virtual objects displayed on the screens [120]. (This work was demonstrated at ISMAR 2004 and was done in collaboration with Chris Sandor, Alex Olwal, and Nick Temiyabutr). We use the overlaid information in our AR system to show the relationships between the physical devices and the related control objects. When the user operates the devices, our system automatically shows each relationship as a connection along with an icon that represents the functionality that occurs with each device (Figure 3.23a). Currently, this icon is placed relative to the 2D projection of the 3D center of the line between the centroids of the objects. The functionality of the devices can also be changed interactively by the user with a tracked wand, similar to the Lincoln Wand [114], but using commercial tracking equipment. Figure 3.23(b) shows a user automatically changing a relationship. In order to change a relationship between a physical device and a virtual object, three attributes need to be chosen in any order: the function, the physical device, and a target virtual object. The physical device and target virtual object are selected by moving the wand close to the device or the target's projection on the screen. The function is selected by moving the wand close to one of the menu items on the table in Figure 3.23(b).

Our system also shows the connection between a physical device and a virtual object while they are changed by the user. Figure 3.24 represents the possible states that occur when the user is doing the interaction: the three input variables (target, device, and function) are mapped to the virtual representation of the connection, which consists of the line between the begin point, end point, and whether the icon is shown. Figure 3.23(b) shows an example when both the target object (highlighted on the screen) and the function (displayed as the icon) are selected, but the user still needs to select the physical de-

vice. In this case, as shown in the table's row (Figure 3.24, highlighted in grey), the visual representation of the relationship consists of the line between the wand's tip (twand) and the virtual object (target), and the appropriate icon to indicate the function selected (i.e., horizontal translation). The visual representations of the relationships, defined in Figure 3.24, are appropriately updated by the system while the user performs the interactions.

Figure 3.22 (a) Browsing an augmented reality environment (b) Tracked handheld display takes priority over labels.(imaged through a tracked video see-through display)

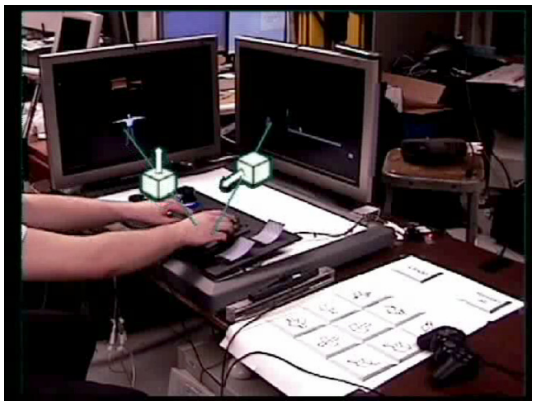
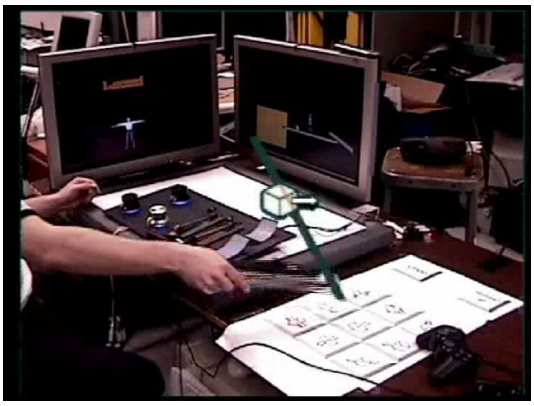
(a)  (b) 

Figure 3.23 Automatic view management shows device functionality. (a) Overlay shows relationships between physical devices in operation and controlled target objects as lines from the devices on the table to the target objects on the displays. Each line is labeled with an iconic representation of its function, in this case translation. (b) The system's state in which the user has selected both the target object and function without choosing the device (i.e., the row pointed out in Figure 3.24). The relationships are shown as a line between the wand's tip and the target object, labeled with an iconic representation of the function selected (i.e., horizontal translation).

Input			Line Output		
Target	Device	Function	Begin	End	Icon
		✓	bwand	twand	✓
	✓		twand	device	
	✓	✓	twand	device	✓
✓			twand	target	
✓		✓	twand	target	✓
✓	✓		device	target	
✓	✓	✓	device	target	✓

Legend

bwand	beginning of wand
twand	the wand's tip
device	physical device
target	Virtual object

Figure 3.24 Table shows how the visual representation of the relationship, consisting of the line and possible icon, is generated based on which attributes (i.e., Target, Device, and/or Function) have been selected. From these attributes, both the beginning and end points of the line are generated from a selection from the beginning of the wand, the wand's tip, the centroid of the physical device, or the virtual object (abbreviations are defined in the legend on the right). A ✓ in the table represents an existing attribute. The state of the system shown in Figure 3.23(b) is the highlighted row with a light grey background.

3.4.2 User-driven view management

In many AR applications, we would like to assist users interacting with and obtaining information about the physical environment. We define *user-driven* view management as the user interactions that predictably influence layout decisions for user control. In our testbed, users can control annotations and other user interface components by

indirect manipulation; a mouse or other wireless device controls a menu to select filtering modes or to manipulate object properties and constraints. Selecting a label reveals more information, which can present buttons or other widgets to offer additional options for obtaining additional information.

The distinction between automatic and user-driven view management is best explained with the example of showing relationships between physical devices and control objects, already described in section 3.4.1 (and shown in Figure 3.23). Even though users' actions directly impact what is displayed and how it is displayed, the rules that are created to show the information do not depend on any users' actions. Thus, we do not consider this to be user-driven because the user does not predictably influence or choose dynamically what is displayed.

Our user-driven interaction techniques focus on allowing users to control virtual information by manipulating physical objects and by using head movement. These direct manipulation techniques are used to create user interfaces that demand less user attention. Shneiderman [126] states the essential ingredients of direct manipulation include immediate response to actions, incremental changes, and reversible effects. These techniques, when applied to AR, can help the user balance their attention between the real world and virtual overlaid objects, and can interactively control what is seen virtually.

3.4.2.1 Related work: User assistance for environmental awareness

AR allows the opportunity to visualize information directly within the context of the real world by overlaying graphics on what is seen. Using overlaid information can assist a user in understanding their environment. This is an especially important task if the user is mobile and the environment unfamiliar: the virtual overlays need to enrich and explain, rather than clutter and confuse, the user's physical surroundings.

One tool in Virtual and Augmented Reality uses the concept of a miniature "god's eye view" model of the user's environment. Furness first sketched this idea embedded within the full-scale environment in a description of the user interface for the Air Force Visually Coupled Airborne Systems Simulator [59]. Similarly, Darken and Sibert [38] place a scaled map of the user's environment at a fixed position at the bottom of the user's view in a head-tracked environment intended to assist in search tasks, and experi-

ment with controlling the map's yaw (*y*-axis orientation) so that it is always aligned with the environment.

Stoakley, Conway, and Pausch [129] expand these ideas to create a *world in miniature* (WIM), a miniature model of the user's environment that is affixed to a 6DOF tracked clipboard held in the user's non-dominant hand and manipulated with a 6DOF device held in the dominant hand. Their WIMs are viewed in context of a full-scale virtual environment and used to support selection and manipulation of objects, including a representation of the user that makes possible rapid locomotion. To avoid the visual confusion that could result when moving the user's representation in a WIM, Pausch and colleagues [109] defer the viewpoint change in the full-scale environment until the manipulation is complete, and then interpolate the user's current viewpoint into the new viewpoint within the WIM itself. Fukatsu and colleagues [57] take an alternative approach in which a handheld 6DOF tracker is used to manipulate a bird's eye view inset into the user's first person view of a virtual environment. Head orientation alone has been used to orbit the user around an object [34], and to view such objects as a WIM, while using head or hand position to move the camera closer to or farther from the center of rotation [85]. In another approach to hands-free interaction, LaViola and colleagues [87] control a world-stabilized WIM on the floor of a CAVE by walking, toe and heel clicking, and leaning. Unlike previous work, our tool uses head orientation in a way that makes it easy for the user in a mobile, (mostly) hands-free application to determine how much attention they wish to devote to the aid. We use view-management techniques so that overlaid annotations enhance the informative display of the WIM by following the user's attention between the full-scale physical environment and the miniature WIM.

3.4.2.2 Using head movement: Controlling a situation-awareness aid

We have developed a situation-awareness aid, based on a world in miniature that provides the user with an overview of the surrounding environment and the ability to discover, select, and inquire about objects that may not be directly visible within that environment. The user's head pitch controls the aid's position, scale, and orientation. This makes possible user initiated view-management control: by using head movement, users can easily change their focus of attention between the real world and the virtual aid.

The aid's yaw (*y*-axis orientation) is constrained to be aligned with the environment, as in [38]. Figures 3.24–3.25 show screen shots of the aid in our laboratory environment. In Figure 3.26(a), the user looks slightly upward and the unannotated aid stays close to the bottom of the viewport, pitched towards the view plane by a default angle (22.5˚) about the user's position in the aid. In Figure 3.26(b), the user looks slightly downward and the aid is scaled to be slightly larger, angled slightly closer to parallel to the view plane, and translated up slightly higher in the viewport. In Figure 3.26(c), the user looks nearly straight down, and in response receives a zoomed-in, annotated view that is nearly parallel to the view plane, with the user's position in the aid located at the center of the viewport. Figure 3.27 shows a version of the aid in our outdoor AR system that uses both texture-mapped aerial photography, as well as 3D models of individual buildings.

The roll component of the aid's orientation is kept at zero, so that its ground plane is always parallel to the bottom of the viewport, but not necessarily parallel to the ground plane in the physical world. Note that the only effect of the user's position on the aid is to control the "you are here" marker, rendered as a red sphere, about which the aid orients (in yaw and pitch). This marker is positioned at the center of the viewport's width and at a head-pitch–determined location relative to the viewport's height.

Figure 3.25 shows how the user's head pitch determines the aid's pitch, scale, viewport *y*-position, and labeling status. The parameters shown were used to make the images in Figures 3.26–3.28. The viewport height is *yres*, and the *max_s* scale factor is computed automatically for a given environment so that the aid fills the viewport when centered and scaled fully. The other parameters were determined pragmatically through informal tests with visitors to our lab. A threshold value determines when annotation is triggered (annotations are turned on in the aid when they are turned off in the real world). For each of the other three variables (pitch, scale, and y position in the viewport), there is a set of four parameters: min and max values and two head pitch trigger values. We linearly interpolate the dependent variables from min to max for head pitch values in between these triggers. Note that these mappings are independent of each other, even though we noticed during our tests that it makes sense to correlate some of the trigger values for WIM animations, as shown in Figure 3.25.

No head pitch trigger value is set above zero degree head pitch. That means that as long as the user looks straight ahead or up, the aid occupies the smallest possible amount of screen space allowed by the parameters and stays near the bottom of the view-port. Both the scale and the vertical position of the WIM are set to reach their maximum values when the user looks down at a 45˚ angle.

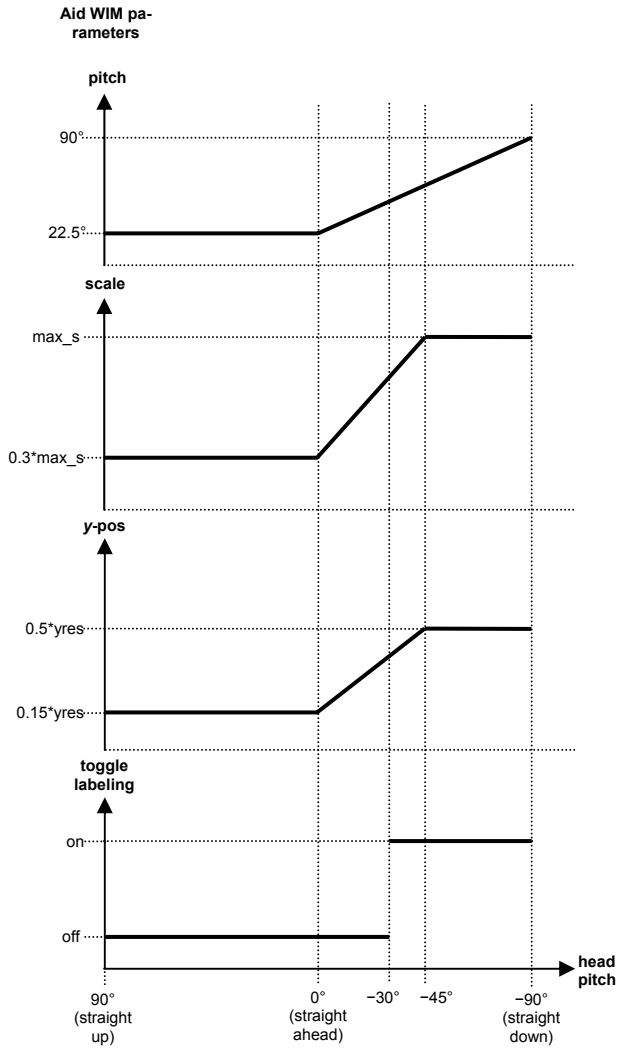


Figure 3.25 Mapping user head pitch (x-axis) to the aid's placement and labeling parameters (y-axis) from the top to bottom: pitch, scale, y-position on the screen, and whether the labels are toggled relative to the aid.

(a)

(b)

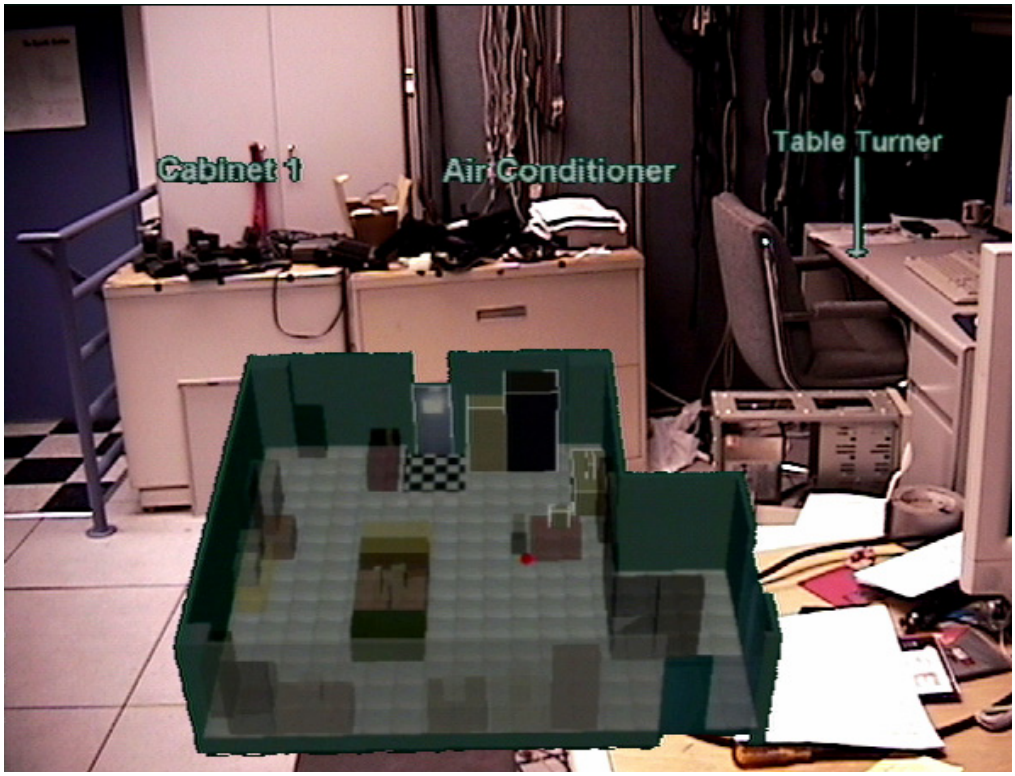


Figure 3.26 Annotated situation-awareness aid. (a) Original position looking slightly upward. (b) Looking slightly downward.

(c)

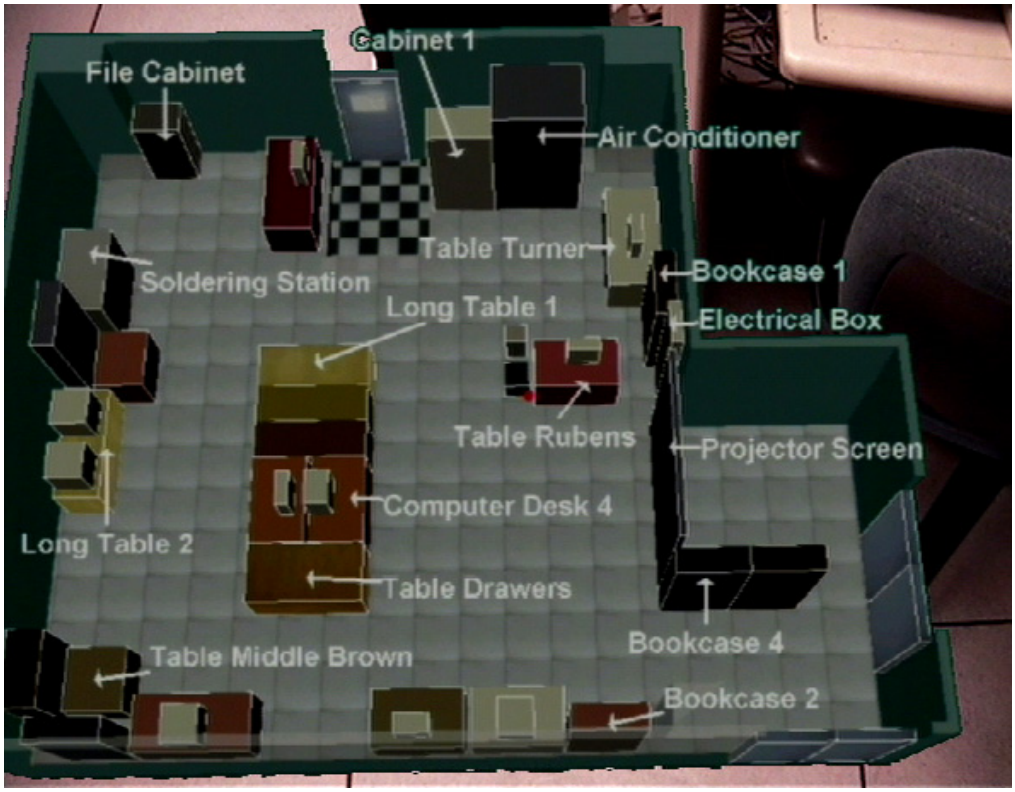


Figure 3.26 (continued) Annotated situation-awareness aid. (c) Looking nearly straight down.

Figure 3.27 Outdoor version of the annotated situation-awareness aid.

3.4.2.3 Showing more information: Annotation assistance

The aid provides us with another set of capabilities for providing the user with more information. In Figure 3.27, labels are placed relative to the projections of the virtual objects in the aid in the same way we annotate the physical objects. A threshold value (set at looking down 30°) determines when the labels change between annotating the real world and labeling the aid.

Popup annotations behave in the exact same way. Figure 3.28(a) shows an annotation reporting on a computer's CPU and memory usage, placed relative to the projection of its associated computer's physical monitor, while avoiding other important objects, including other physical objects, labels, and the aid. As the user looks down at the aid, the annotated physical monitor is no longer visible, but the popup remains displayed. As the aid scales up, its labels appear, and the popup's arrow points to the monitor's virtual representation in the aid, shown in Figure 3.28(b). As the user looks up, the popup is dissociated from the aid and recaptured by the physical monitor in the full-scale physical environment. Thus, popup annotations can move freely between the physical world and its scaled virtual representation. Furthermore, virtual objects in the aid can be selected to view information about physical objects that might not be currently visible in the real world from where the user is currently positioned.

(a)

(b)

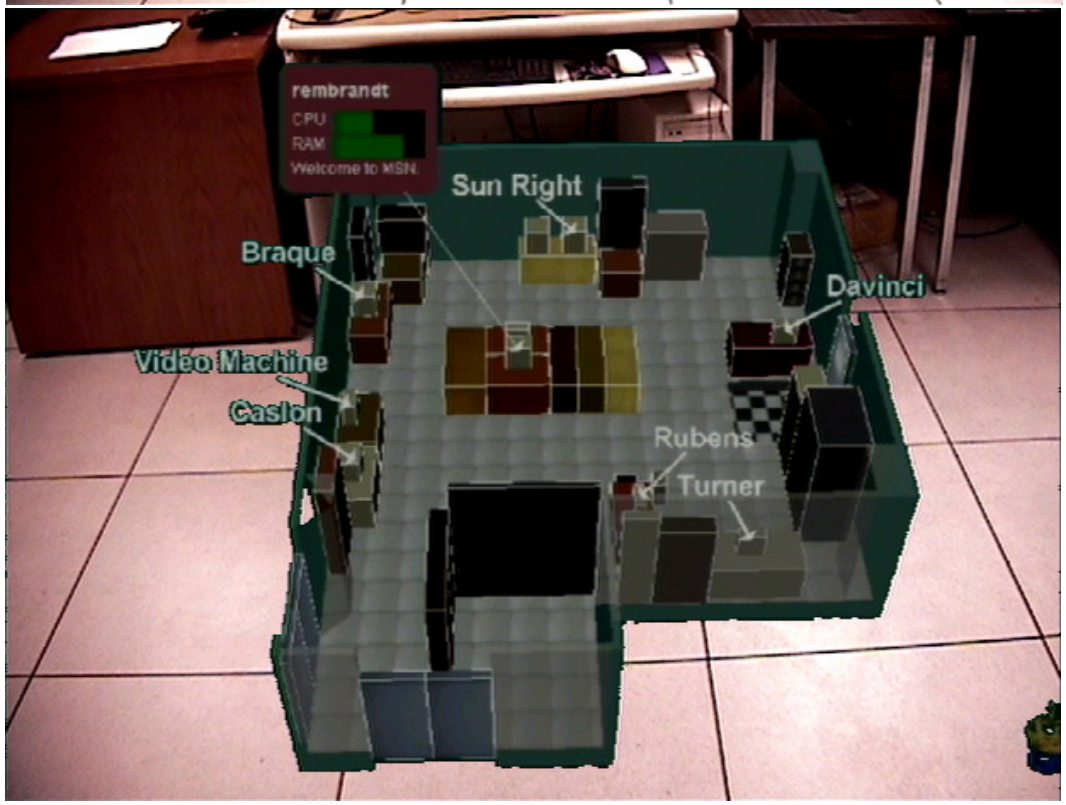


Figure 3.28 Transfer of annotations between the physical world and virtual aid. (a) Popup annotation provides information about the computer monitor to which its arrow points at the center. (b) As user looks down, annotations are enabled in the aid, and the popup's arrow now points at the monitor's representation in the aid.

3.4.2.4 Manipulating physical objects: Moving a handheld display

We can also provide user-driven view management by allowing users to manipulate physical objects, such as a handheld display, to influence what the user sees. Figure 3.22(b) shows a user interactively controlling a six-degree-of-freedom-tracked, handheld computer, whose screen we would not like to have obscured by labels. To accomplish this, the screen is defined as having a higher priority than any of the objects in Figure 3.22(a), and no associated label. Consequently, since much of "Table Davinci" is obscured by the handheld computer in Figure 3.22(b), relative to Figure 3.22(a), the "Table Davinci" label changes from a large internal label to a smaller external one; the "File Cabinet" and "Ascension Transmitter" labels and arrows move up and down, respectively, out of the handheld computer's way; and the "Thin Table 1" label disappears.

Alternatively, imagine replacing the tracked handheld computer with a transparent tracked frame. Instead of having the system show labels for all objects that are not obscured by the handheld computer, the system could act like a 3D magic lens [137], showing labels only for objects that fall within the frame.

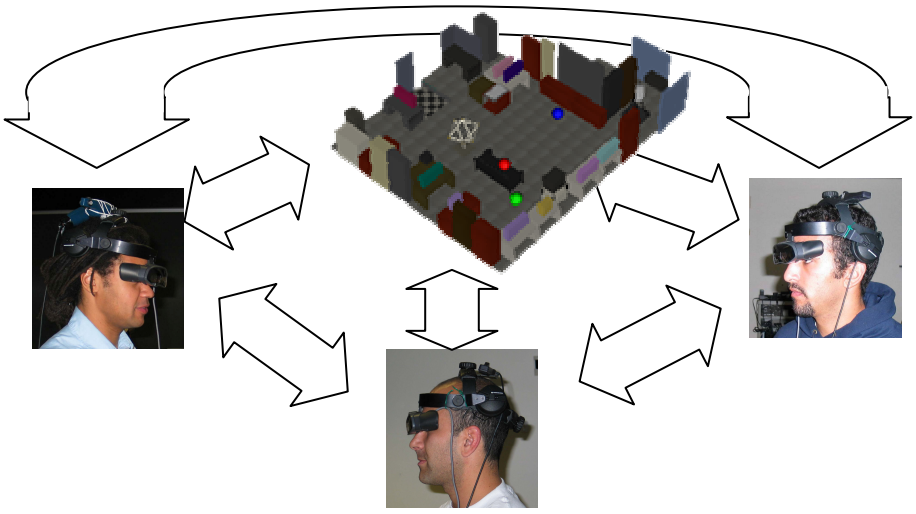


Figure 3.29 Each user sends and receives information to and from every other user and display in the environment and the environment itself.

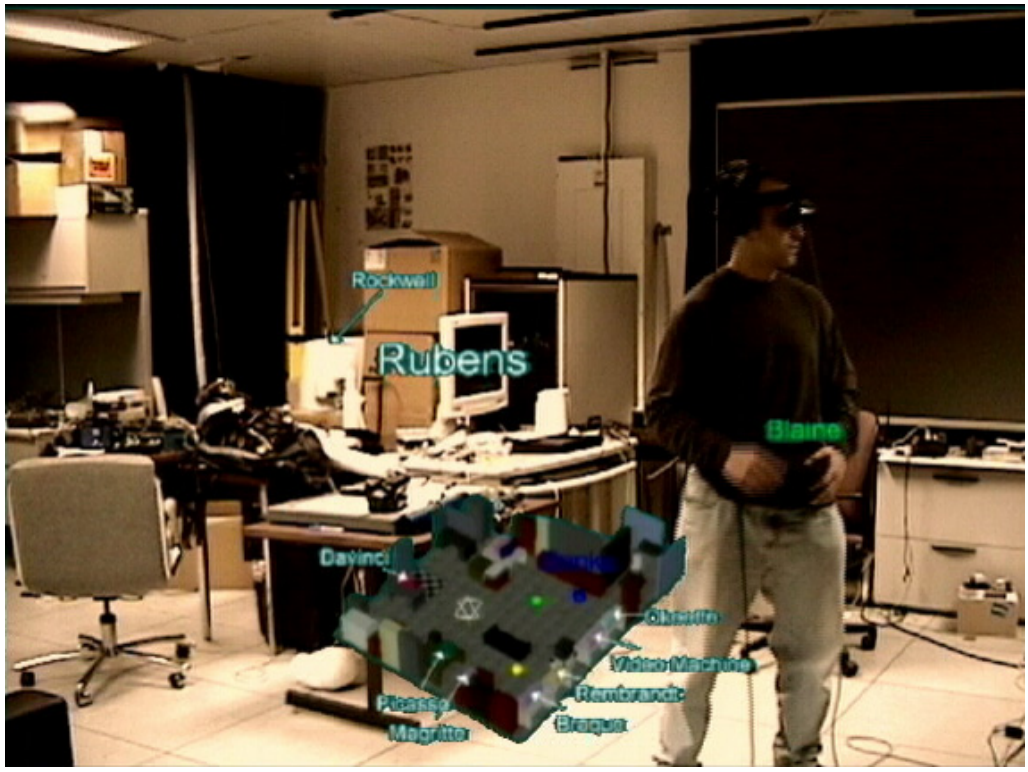
3.4.3 Collaborative/distributed view management

There are many ways that a collaborative system and its users can perform view management for multiple views. Figure 3.29 shows how information can move between different entities in our system: users can send information to and receive information from any user or display in the environment, as well as the environment itself. The current work implemented includes some examples on how information displayed reacts to other information in the environment.

3.4.3.1 Keeping track of important objects

We may wish to keep track of certain objects (e.g., other team members), independent of whether they are being seen by us or by others. Figures 3.30–3.31 shows views of a single AR user. Other users and objects that are directly visible are labeled in the user's view of the real world; all remaining users are labeled in the user's situation awareness aid, in which each is represented by a colored sphere. As users move into and out of view, the system automatically moves their labels between the real world and the aid, using animation to maintain coherence. Labels are laid out using our view-management algorithms (Chapter 3).

(a)



(b)

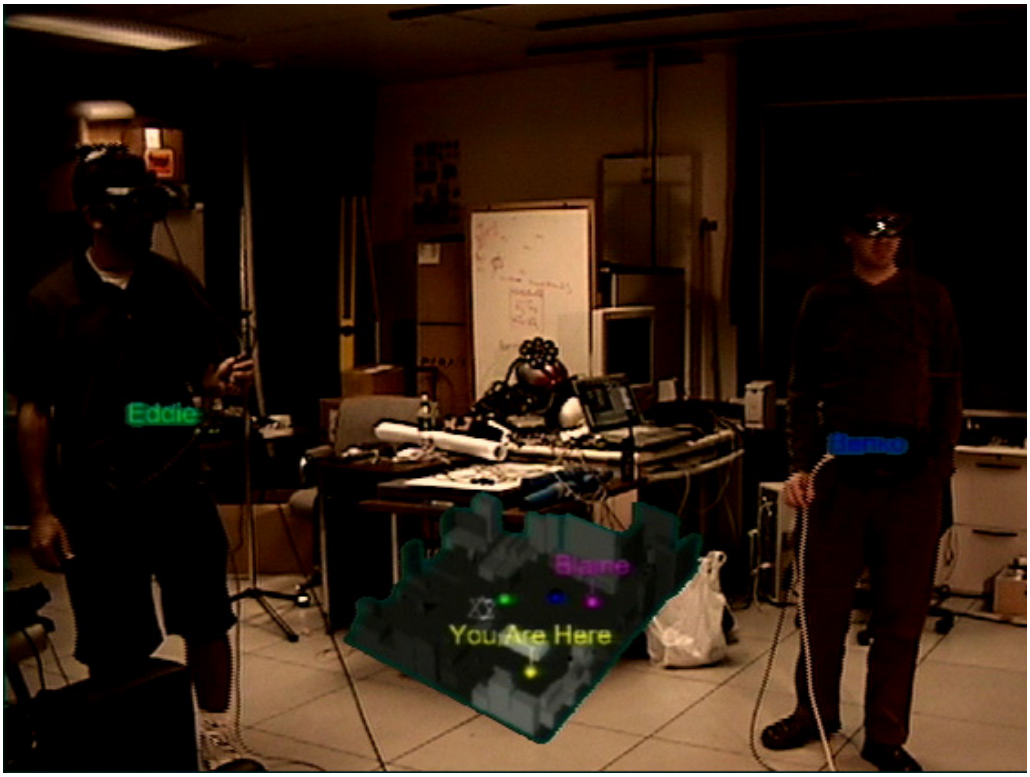


Figure 3.30 Viewing a collaborative augmented reality environment (imaged through a tracked video see-through display). (a) Users and additional objects labeled, (b) Situation-awareness aid being used to monitor what other collaborating users are viewing.

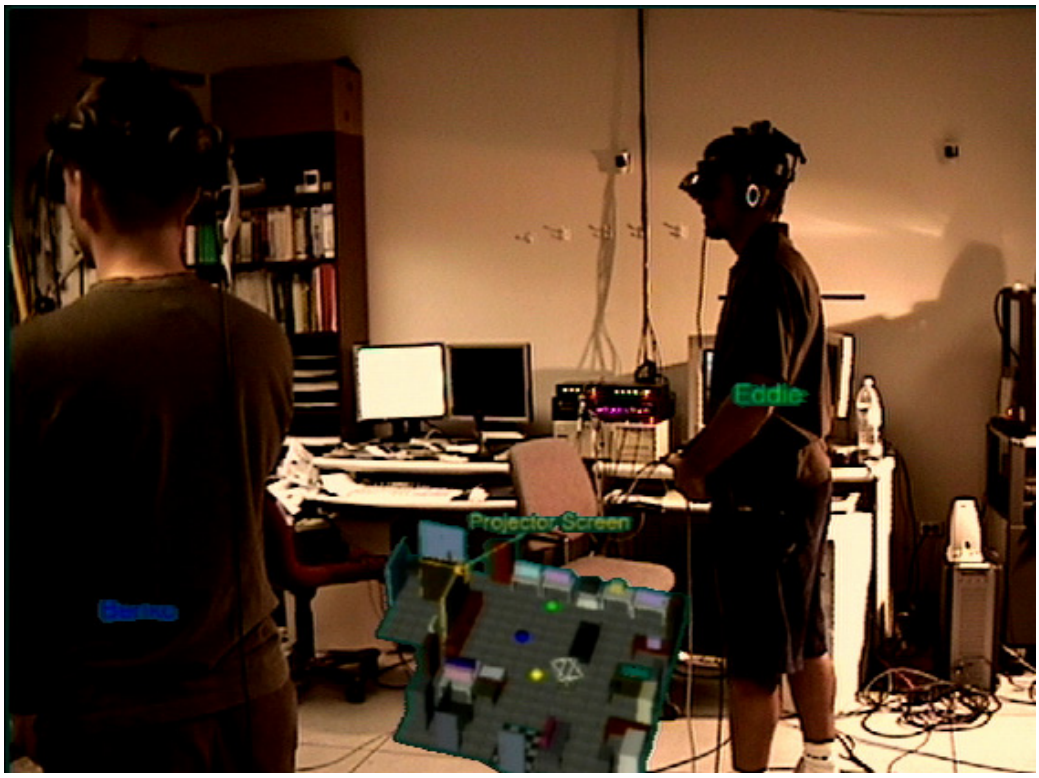


Figure 3.31 A notification rule that points out (imaged through a tracked video see-through display) any display that two users are viewing and is not in the local user's field of view. In this case, the Projection Screen fits the rule, resulting in the display of a related label in the aid.

3.4.3.2 Monitoring other users' views

We may wish to visualize what others are currently viewing. To make this possible, we have built on the situation-awareness aid described in Section 3.4.2.2. Presenting an overview of the environment in the aid could help the user get the big picture, while focusing on a particular object in the aid could provide details about it. Figure 3.30(b) shows a collaborative environment in which the virtual representations of real-world objects in the situation-awareness aid are colored based on the number of users that currently have that particular object in their field of view: the more users viewing an object, the brighter it becomes. This visualization might be useful to a team that needs to maximize coverage of an area being monitored, or to determine which objects are currently attracting the attention of others. In our implementation, each user's own computer determines the visibility of their environment, based on their position and orientation, and sends this information to any user that has requested it. A variation of this approach ac-

cumulates visibility information over time, and uses it to color an object to represent the number of different users that have seen it or the amount of time that it has been in view.

3.4.3.3 Notifying users of other users' actions

Notification mechanisms are commonplace in current software systems; for example, informing us about incoming email, instant messaging, software updates, and bug fixes. We are exploring the use of notifications in our collaborative environment, in which a user might want to be notified whenever other users do certain things. For example, Figure 3.31 shows the result of a request for notification when at least two other users are viewing a display in the real world and the local user is not. In this case, the system highlights the projection screen in the situation-awareness aid, and labels it to point it out. (The annotation could also show what the other users are viewing on the screen.)

User

User	Is Local
Blaine	TRUE
Eddie	FALSE
Benko	FALSE
Alex	FALSE

Visibility

User	Object	Visible
Blaine	Screen	FALSE
Eddie	Screen	TRUE
Benko	Screen	TRUE
Alex	Screen	FALSE
Blaine	Table	TRUE
Eddie	Cabinet	FALSE
…	…	…

Objects

Object	Number of Lookers	Is Local Looking
Screen	2	FALSE
Cabinet	0	FALSE
Table	1	TRUE

Figure 3.32 Data tables that represent information used to develop rules. Incremental updates to the Visible column in the Visibility table are distributed to other users throughout each example. The Number of Lookers *and* Is Local Looking *fields are computed values, based on information in the User and Visibility tables. In each example, different rules use this information to control program functionality.*

3.4.3.4 Distributed view management implementation

The distributed view management examples in this section are based primarily on computing visibility information for all collaborating users in the environment. We use a rule-based approach implemented in Data Programming (Chapter 4) to specify how in-

formation is computed and communicated to each of the users. It is difficult to implement these examples using typical programming techniques, such as object-oriented programming, since it would require developing a protocol that would communicate the necessary information from each computer to all of the others. When designing this protocol, complicated design decisions arise even for these simple applications, such as 1) it is not clear whether source and destination computers should be specified in the protocol for error checking, 2) whether visibility information for multiple objects should be included in the protocol, since changes might occur at the same time, and 3) which variables should be included into the protocol. In Data Programming, instead of focusing on protocol design, the logic is developed into the design of a set of distributed tables, which simplifies the implementation and avoids the need to make these difficult decisions about the protocol. Performance and functional enhancements are made by manipulating the distributed tables without continually changing or adding new protocols.

Figure 3.32 shows simplified versions of some data tables (i.e., User, Visibility, and Objects tables) for the local user whose view is seen in Figures 3.30–3.31. The *User* table has a row (i.e., record) for each user, with a column *Is Local* that represents which user is local to this computer. The *Visibility* table encodes which objects are seen by which users, where each computer has visibility information for all users in the distributed environment. The *Objects* table indicates the number of users looking at each object and whether the local user is among them.

Each table has certain rows that are distributed from each user's computer using one-way replication. Each user replicates the user's own row in the *User* table (without sending the *Is Local* column) to every user in the distributed system. The distribution strategy for the *Visibility* table reflects the source of the visibility computations. For our examples, since each user's computer calculates their own visibility, each user distributes a row in the *Visibility* table for each object to every other user in the environment. The visibility computations incrementally update the *Visible* column that is based on the distributing user's own visibility calculations.

The *Objects* table keeps track of information about the objects and is replicated from a server which publishes information about the environment. Two fields in the *Objects* table are not distributed since they can be computed directly from the two other tables (*Visibility* and *User*): *Number of Lookers,* which represents the number of users

looking at each object, and *Is Local Looking*, which represents whether the local user is looking at each object. To compute the *Number of Lookers* field, we set up a links between each *Visibility* row and its respective *Objects* and *User* rows. A local computation in the *Visibility* table increments the *Number of Lookers* by one when an object becomes visible to any user, and decrements it by one when it stops being visible to any user. Therefore, as updates to the *Visible* column are received by remote users, *Number of Lookers* will also be updated. To set the *Is Local Looking* column in the *Objects* table, a link between each object row and the related *Visibility* record for the local user is used to connect the *Visible* column in the Visibility table into the *Is Local Looking* column in the *Objects*. Thus, whenever the visibility changes for local objects, the *Is Local Looking* column in the *Objects* table is changed as well.

In the examples previously shown, there are also computations that use this distributed in formation to set application variables, such as rendering color and label annotations. In Figure 3.30, the *Number of Lookers* field is used to set the grey scale of the objects in the WIM (i.e., for simplicity, dividing the number of lookers by the number of users in the system). In the example in Figure 3.31, there is also a computation that determines whether the two computed fields, *Number of Lookers* and *Is Local Looking*, match the rule. If they do, then the rule sets a Boolean value that fires all the computations needed to place a label relative to the virtual representation of that object. We present a detailed description of how these computations are developed using Data Programming in Chapter 4.

3.5 Parameter guidelines based on informal user feedback

Throughout the duration of this work, we have performed many AR and VR demonstrations, from the original Emerging Technologies demonstration [45] at SIGGRAPH 2001, to outdoor mobile AR demonstrations at ISAR 2001 [46], and to informal laptop VR demonstrations at UIST [18, 19]. From the many users who have experienced our system, we have received feedback that has helped assist us to set the annotation parameters properly. We have found that our animation parameters, defined in Section 3.3.5, need to be changed based on how the different viewing specifications, defined in Section 3.3.2, move the camera with respect to the annotated objects, so that our system can provide satisfactory annotation results for every situation.

View	State Hysteresis	Positional Threshold	Interpolation
Mouse Control	Low/ Not needed	Low/ Not needed	Fast/None
Pre-determined			
Spinning	Low/Moderate	Very Low	Slow
Sweeping	Low/Moderate	Very Low	Slow
HMD			
Walking	High	Low	Moderate
WIM	High	High	Moderate

Figure 3.33 Parameter guidelines for different view specifications (left column). Parameter columns relate to state hysteresis (Section 3.3.5.1), positional stability (Section 3.3.5.2), and interpolation (Section 3.3.5.3).

Figure 3.33 shows how we characterize different viewing specifications (left column) into mouse control, an HMD view for walking around a regular-sized world (i.e., either in VR or AR), an HMD view onto our virtual aid based on a WIM, and two different types of pre-determined paths: spinning around an environment while keeping focus on the same position, and sweeping from one side of an environment to another while rotating the view. These pre-determined paths have been typically used for presentational videos of our techniques. In each of these cases, we describe guidelines (in the three parameter columns in Figure 3.33) for determining programmatically the parameters for the three types of annotation animation: state hysteresis, positional threshold, and interpolation.

When the viewing specification is controlled by a mouse (as stated previously Section 3.3.5), animation is typically not preferred (Figure 3.33, first row). This is attributed to the interaction behavior we have built into the mouse: once the user presses down on the mouse button, the movement of the mouse either orbits around a focus point [85], changes distance from the focus point, or translates the focus point until the mouse button is released. These parameters are specific to this mouse behavior, and are not generalized to all mouse behaviors.

The fields in Figure 3.33 relate to how the discrete values are set for the algorithms described in Section 3.3.5. The state hysteresis, ranging from low to high, relates to how much the system tries to keep the annotation at the same state (i.e., inside, outside, or no show) with respect to its previous state. The parameters to control this relate to Figure 3.18: the time interval (time) and the difference between absolute minimum and minimum size (absMin-min). For situations in which it becomes more important for annotations to keep the same state, such as when using HMDs, both of these values increase. The highest time values were typically around .3 seconds, while the annotation sizes depends on the content and screen resolution. The largest difference between the absolute and minimum sizes was approximately 20% lower than its minimum size (e.g., a font size of 25 had the absolute font size of 20).

The positional threshold, also ranging from low to high, relates to how much the system tries to keep the annotation at the same position. The algorithm, described in Section 3.3.5.2, compares previous placements with the optimal current placement. In order to control the range of the positional threshold for different situations, we change the distance that evaluates whether the closest to previous layout is the same as the optimal placement (Figure 3.19). As the increase of positional threshold is needed, we have increased this distance to attempt to keep annotations in the same position for a longer period of time. The distance is dependent on pixel screen size, but for our examples, the largest distance we have used to get reasonable results was 20 pixels on an 800x600 size screen. Using high values is particularly useful when the current optimal placement of the annotation changes frequently, but within a small area. Thus, higher values produce reasonable results when using an HMD view with a WIM, since the objects projections are typically small.

The interpolation parameter, which ranges from none to moderate, relates to how fast annotations move from one position to another once it determines that it should move. We use the distance threshold (Section 3.3.5.3) to change the fastest speed that an annotation can move from its previous value, and use pixels per second to set this threshold. For the moderate situations when using an HMD, users seemed to be comfortable with 300 pixels per second on an 800x600 display.

Chapter 4

Data Programming (DP)

Many of the examples described in Chapters 2 and 3 were initially implemented in JAVA [63]. As more functionality was built into these systems, development became more difficult and required redesigning the JAVA objects and their relationships. Not only did they take a long time to redesign, it also wasn't clear which design was the best. What is the best usually depends on what functionality is to be added next, which is not often known. In this section, we describe a new software development technique we call *Data Programming* (DP), which allows programmers to build complex interactive applications that are flexible and easily extensible. The technique is similar to main-memory database management systems [40, 88], which use fast accessible program memory for data, but with an important difference: DP incorporates an entire program's functionality into the tables, rows, and joins, which include not only data, but code as well.

DP allows programmers to focus on both programming logic and memory use. There are two important aspects of DP: (1) the system that supports DP (analogous to a database server), and (2) the way in which programmers use DP (analogous to a database programmer using a database server). Both of these aspects use the fundamental programming structure *data type record*, which is essentially a sequential array of pointers in memory, and a *data type*, which defines how data type records are allocated. Together, these serve as the backbone of DP.

4.1 Rationale

We developed DP through the evolution and integration of two projects that both use space management and view management: MARS (Mobile Augmented Reality Systems) [47, 71] and MAGIC (Multimedia Abstract Generation for Intensive Care) [75, 76, 98]. From the initial implementation using JAVA, we started the conversion into a rule-based system using the rule-based expert system called JESS [55]. JESS uses the Rete pattern matching algorithm [53], which allows user-defined rules to continuously evalu-

ate and fire on facts that are asserted and retracted in the knowledge base. Our system was developed in conjunction with Tobias Höllerer, who developed Ruby: a rule-based architecture for MARS UI management [70]. In the Ruby system, JESS rules were created for exploring different display modes in AR, and for developing logic that adapts to different situations in AR, such as tracking accuracy and user context.

A long-standing problem with using a rule-based system in an interactive graphics environment is that it is too slow to handle certain rules [9]. For example, if we wanted to impose a rule that specifies that a virtual object should get rendered differently in user *A*'s environment depending on whether the object is visible to user *B*, then a fact needs to get updated whenever an object changes visibility to user *B*. In turn, continuous re-evaluation of this rule asserts and retracts the facts representing the rendering attributes of the related object. This can happen quite frequently, possibly before every frame rendered, and results in poor performance because the amount of memory that is allocated and de-allocated depends on the number of changing facts and partial-orders of pattern matches in working memory (i.e., asserted and removed facts). Furthermore, when rule bases become more complex, the additional rules with the same dependencies perform additional evaluations, further hindering performance.

To cope with this problem, since JESS is integrated into JAVA, instead of using facts to define variables, we decided to use facts to *instantiate* variables, such as a Boolean variable for visibility and a floating-point variable for transparency. Instead of asserting and retracting a fact that represents whether an object is visible, a Boolean variable would get changed, eliminating the need to allocate/de-allocate memory at run-time. Since the knowledge base allows the freedom to access and create relationships between any of these variables, it is easy to instantiate more data relationships. For example, typical operations could include defining a Boolean variable for whether the rule should be enforced, and some code to set the transparency to a specific value only if the rule is enforced *and* the visibility is true.

As the system evolved, more variables were created in the knowledge base, and the system took longer to initialize. Many variables were instantiated at the same time for the same reasons. Because of this, the *data type*, a template for creating multiple instances at the same time inside an array, called a *data type record,* was created to minimize the number of facts in the knowledge base and to reduce startup time. Although the

rule-based system still determined when data type records were allocated, the number of rules and records increased quickly throughout development and it became difficult to manage and expand the rule base. By moving the functionality from the rule-based system into data type records, the entire JESS component was eliminated. Thus, the entire logic of a DP program consists of tables, columns, relationships between columns (i.e., function arguments), and rows (i.e., data type instances).

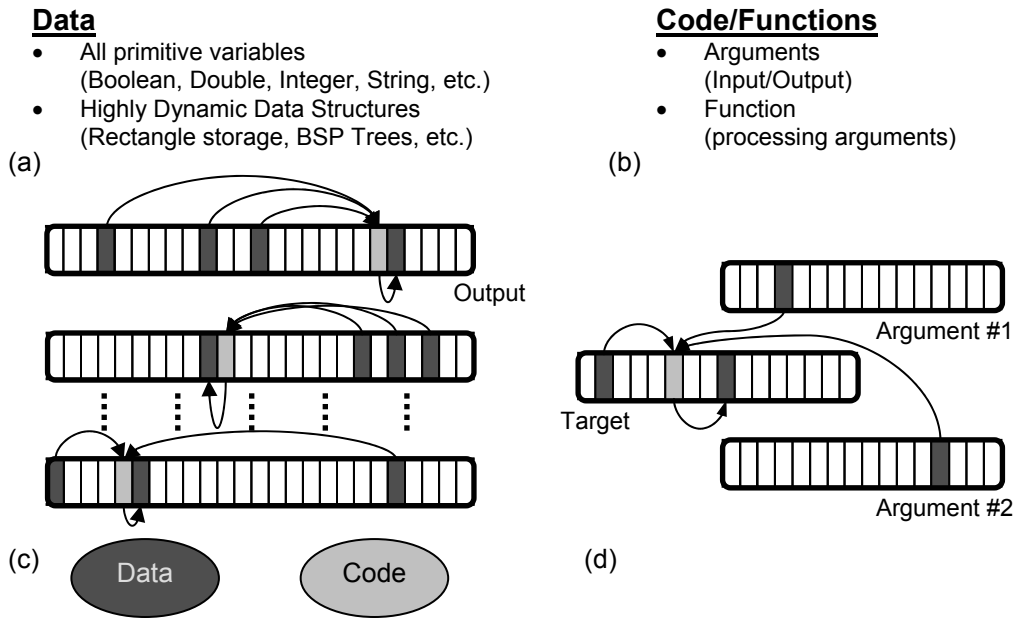


Figure 4.1 Data types (i.e., tables) have two types of slots: data and code (i.e., functions). (a) Data slots consist of primitive variables and highly dynamic data structures, and (b) function slots consist of the code itself and its arguments, which are relationships with other slots. (c) Internal arguments are relationships between slots within the same data type, and (d) external arguments are relationships between slots in related data types. Arrows indicate the relationships between arguments and their functions.

4.2 Data types

A *data type* represents the definition of a *table* and its relationships between other tables. It is used to instantiate each *data type record* (i.e., a row), which is essentially an array of pointers residing sequentially in program memory. Instead of separating an object's member variables and functions, as in Object Oriented Programming, the *data type* defines each column (i.e., a *slot* or field) as either a piece of data or a function (i.e., code). A *data slot* consists of either primitive variables, such as Boolean, double, integer, or a

string value, or dynamic data structures, such as a BSP Tree, a rectangle list, or a sorted list. A *function slot* is defined by code (or a pointer to code), along with the relationships between the slot's input and output arguments. Boolean functions, such as 'And' and 'Or', have an arbitrary number of arguments, while other functions, such as computational functions, have a fixed number of arguments, which consist of any number of inputs and outputs.

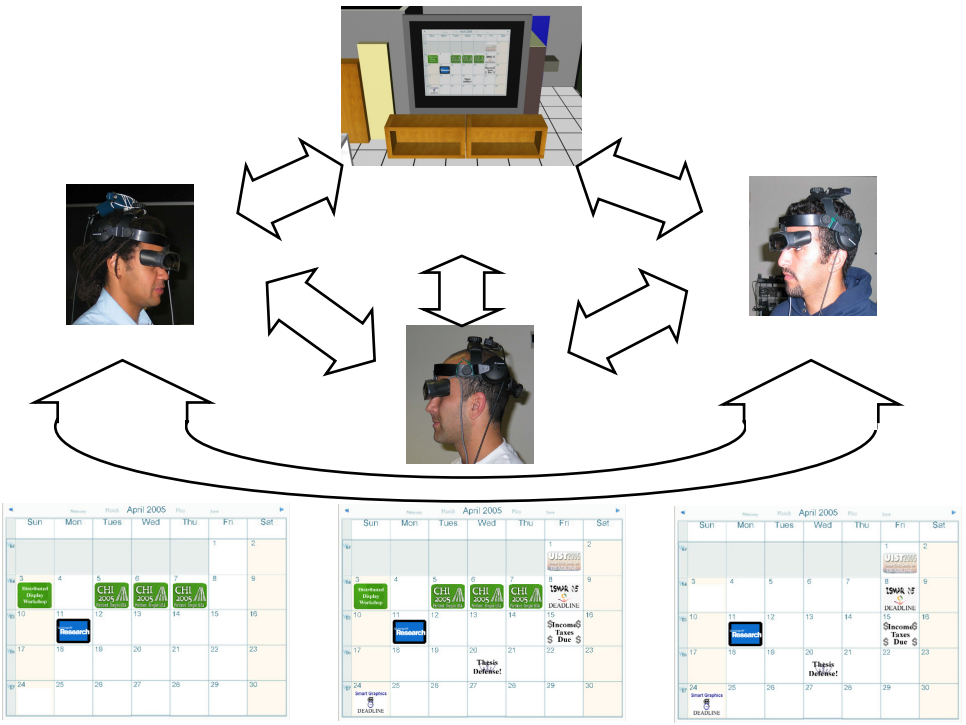
By defining *data types*, programmers can dynamically develop arbitrary, multi-dimensional data structures, along with computations that process them. The *data types* allow related data to be placed close together in memory, so that functions have direct access (a constant array offset) to their internal arguments (Figure 4.1c). Functions can also have external arguments that come from other records of *data types* (shown in Figure 4.1d). More complex relationships can also be defined between slots (both internal and external) using system tables that make it easier to develop relationships between multiple functions and arguments. This is useful when creating groups of slots with the same relationships, or patterns, to develop similar functionality that resides in the same or different data types.

Function arguments can also have special arguments, such as an entire *data type record* (either the target or an external argument), an offset value of a particular *slot*, or a constant value. This allows a function to access arbitrary slot values based on other arguments, typically done for flexibility. This, along with data structures within slots, such as a hash table or an ordered list, makes it possible to easily access data anywhere in the system from any *function slot*. This is similar to many rule-based and knowledge-based systems; however, the tabular structures provide accessibility in a systematic structured way.

Slots can be generated using a *data attribute*, which represents a way for programmers to define one or more slots in one or more *data types*. Some attributes can be defined in all data types, such as the Boolean value named "is-initialized" that represents whether the *data record* has been initialized. Other slots can be defined in groups of data types, and others can be defined in only one data type. Each attribute can also have an *attribute instance type* that defines how many *slots* are allocated for a particular attribute. For example, a value that can be animated might have three *instance* values: before ani-

mation, after animation (or current), and the start value of the animation, where the current value is computed by interpolating between the before animation and start values.

(a)



(b)

Calendar Event Type	User 1	User 2	User 3
Solid green	✓	✓	
White		✓	✓
Black outlined	✓	✓	✓

Figure 4.2 Three users wearing head-worn displays in an environment with a shared wall-mounted display. (a) Each user has their own view of calendar information that can be shown on either the physical wall-mounted display or the user's head-worn display. (b) We categorize the calendar events into three types (Solid green, White, and Black outlined) and define each user's view by these types (checks "✓" in the table represent that a user can see the particular type).

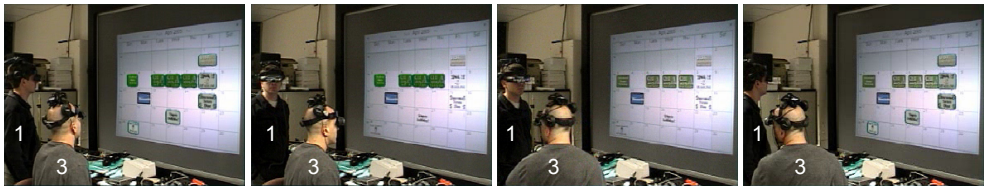
4.3 Example: Distributed Display Environment

Consider the following example of how DP can be used to program logic in a distributed environment with multiple users and displays. Figure 4.2(a) shows an environment with three users wearing head-worn displays and a shared wall-mounted display. We define a rule that shows each user their own view of some calendar information defined in Figure 4.2(b) using both the head-worn display and shared display:

If *A* is a graphical object
and
A is included in the view of everyone who is looking at the wall-mounted display
Then
Show *A* on the wall-mounted display
Else
Only show it virtually on those users' head-worn displays

Figure 4.3 shows an example situation: as users change their view of the wall-mounted display (shown in Figure 4.3a), the graphical objects move between the head-worn displays and the wall display, based on the rule and views defined for each user (Figure 4.2b). The benefit of our design in DP is that it is flexible enough to account for an arbitrary number of users with an arbitrary number of computers and displays, and each user's view (defined for each user, as in Figure 4.2b) can dynamically change at run-time while the system recomputes the rule interactively. We describe the implementation for the distribution tables that allow specification of the users' views on a per-object basis, and the results, which consist of values distributed across the system on every computer, determine how to correctly render the graphical objects on each local display. This example distributes information in tables using the peer-to-peer configuration that is similar to the examples shown in Section 3.4.3. We first describe the data type definitions, followed by the computed slots and how they are determined, and conclude by describing the distribution mechanisms.

User 2's View:



(a)

	State 1	State 2	State 3	State 4
Visibility: is the user looking at the physical wall-mounted display?				
User 1	TRUE	FALSE	FALSE	TRUE
User 2	TRUE	TRUE	TRUE	TRUE
User 3	TRUE	TRUE	FALSE	FALSE

(b)

On Shared Display				
Solid green	FALSE	FALSE	TRUE	TRUE
White	FALSE	TRUE	TRUE	FALSE
Black outlined	TRUE	TRUE	TRUE	TRUE

(c)

Figure 4.3 Situation shown in Figure 4.2 that upholds the rule in Section 4.3 that controls each user's view of calendar information and places each piece of information on the shared display when possible, based on where users are looking. (a) View through User 2's head-worn display at four different states, with Users 1 and 3 labeled for this figure. (b) Table indicating which users are looking at the shared display, and (c) the results of the rule (i.e., whether the event type is placed on the shared display). If the information is not on the shared display and is in the user's view, then it is placed virtually on the head-worn display. For example, since User 2 sees every event type, if the event isn't on the shared display, then it is shown virtually on User 2's head-worn display.

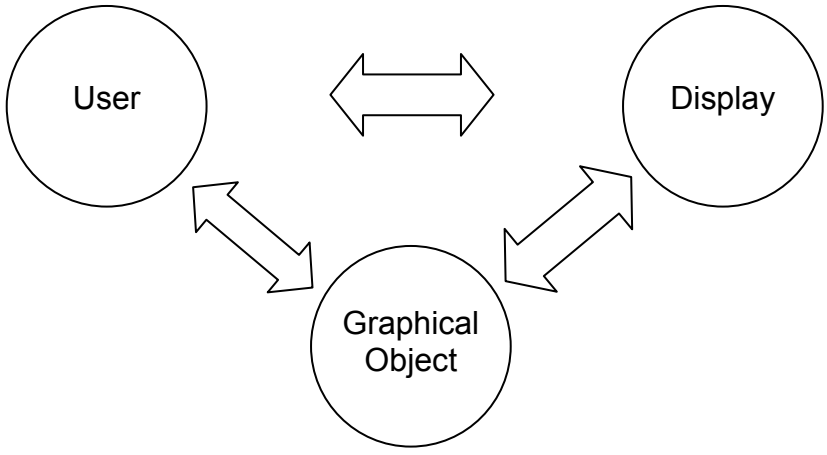


Figure 4.4 Relationships between main entities Users, Displays, and Graphical Objects, form the basis for designing tables for this distributed display example.

User

User	Name	isLocal
1	Alex	FALSE
2	Blaine	TRUE
3	Eddie	FALSE

Display

DisplayID	Is HMD	IsWallMounted	IsLocal
1	TRUE	FALSE	FALSE
2	TRUE	FALSE	TRUE
3	TRUE	FALSE	FALSE
4	FALSE	TRUE	FALSE

GraphicalObject

ObjectID	Type	AnyUser Looking CanNot See*	LocalUser CanSee Object*	Show On HMD*	Show On Wall*
1	Solid green	TRUE	TRUE	TRUE	FALSE

MainMemory

Local Is HMD*	Local Is Wall*
TRUE	FALSE

GraphicalObjectToUser

ObjectID	UserID	UserCan See Object	ImposeOn LocalUser Can SeeObject*
1	1	TRUE	FALSE
1	2	TRUE	TRUE
1	3	FALSE	FALSE

UserToDisplay

User	Display
1	1
2	2
3	3

GraphicalObjectToDisplay

ObjectID	DisplayID	IsRelative
1	1	FALSE
1	2	FALSE
1	3	FALSE
1	4	TRUE

GraphicalObjectToUserDisplay

ObjectID	User	DisplayID	User Looking At Related Display*	ImposeOn AnyUser Looking CanNot See*
1	1	4	FALSE	FALSE
1	2	4	TRUE	FALSE
1	3	4	TRUE	TRUE

*Figure 4.5 Tables used for the distributed display example derived from the entities in Figure 4.4. Tables represent entities and relationships. Data within tables show the DP database on User 2's head-worn display for a solid green object State 2 of Figure 4.3. In this case, the graphical object is shown on the user's HMD. "*****" denotes a computed value.*

4.3.1 Example: Data type definitions

The data type design is based on the relationships between three main entities: Users, Displays, and Graphical Objects (Figure 4.4). Tables and slots (shown in Figure 4.5) are created for each entity and the three relationships, and reside on all the machines in the peer-to-peer system. The relationships between tables are created by populating slots in one table with the unique values (i.e., keys) of the other related tables, similar to

the way how joins work in relational databases [5, 101]. For example, the table "GraphicalObjectToUser" in Figure 4.5 uses two slots to represent the relationship between the GraphicalObject and User (i.e., the relationship table named "GraphicalObjectToUser" in Figure 4.4): ObjectID and UserID. These values are used to access the related instances in the entity tables GraphcalObject and User, respectively, and can be concatenated to create a unique value for the data type.

We also include another table (i.e., GraphicalObjectToUserDisplay) that represents the relationship between each User, Display, and Graphical Object. This table has columns for each entity (i.e., User, DisplayID, and ObjectID, respectively) that uniquely identifies each row, which are created for every possible combination of these entities. This table is important for creating variables that represent a relationship between all three entities, as described in this example.

The data tables and relationships are created to represent the entities with the correct number of instances. For example, variables that describe the Users or Displays are located in their respective tables, such as the User's name (User.Name), and whether the Display is a head-worn display (Display.isHMD) or a wall-mounted display (Display.isWallMounted). More complicated variables need an instance for every instance of a relationship, such as the information in Figure 4.2(b). In this case, in order to define each user's view, a Boolean value is needed for every GraphicalObject for every User. Thus, we define the Boolean slot "UserCanSeeObject" in the GraphicalObjectToUser data type, which is used to impose the rule defined for this example and can also be changed interactively at run-time.

4.3.2 Example: Computed slots

Each computed slot, shown with an "*" next to its name in Figure 4.5, represents values that are computed from one or more pieces of data within the database and is used to solve rules. Figures 4.6–4.7 show the Boolean functions that set the values of the computed slots from other slots within the same or adjacent tables, where slots are specified by the data type name and slot name separated by a period (".”). DP implements these functions as Boolean operations (e.g., "And", "Or" and "Not") between the variables within one or many adjacent related tables. For each of these operations, if any input changes, the operations are automatically triggered to set the result. These Boolean

functions are similar to rules in a rule-based system, except, instead of rules being fired, the functions set the output variables to either TRUE or FALSE, based on the evaluation of the inputs, without the need to allocate new memory.

Figure 4.6 shows how two slots in the Entity table "GraphicalObject" are set from slots in the related tables "GraphicalObjectToUser" and "GraphicalObjectToUserDisplay". Since there is an *N-to-1* relationship between relationship tables and the entity tables, the result is set by any number of values in the related input table by using an "Or" function that has *N* arguments, where *N* can change if more relationships are created. This functionality is expressed in Figure 4.6 using the function "anyValueOf". For example, the first function in Figure 4.6 sets the slot AnyUserLookingCanNotSee in the table GraphicalObject, which is set to TRUE in Figure 4.5 for ObjectID=1, because out of the three related rows in the table GraphicalObjectToUserDisplay, there is one row (i.e., the row with ObjectID=1, User=3, and DisplayID=4) that has a value TRUE in the column named ImposeOnAnyUserLookingCanNotSee.

Figure 4.7 shows Boolean functions that set computed columns, where the inputs come from the same or related tables. The first function defines when the graphical object should be shown on an HMD: if the local display is an HMD, and any user who is looking at the wall-mounted display should not be able to see it, and the local user can see it, then show it on the HMD. The second function defines when the graphical object should be shown on the wall-mounted display: if the local display is wall-mounted, and if no user looking at the screen can not see the object (i.e., not GraphicalObject.AnyUserLookingCanNotSee), then show it on the wall-mounted display. The next two functions are used to set the variables that are used for input in Figure 4.6.

1) GraphicalObject.AnyUserLookingCanNotSee =
   anyValueOf(GraphicalObjectToUserDisplay.ImposeOnAnyUserLookingCanNotSee)

2) GraphicalObject.LocalUserCanSeeObject =
   anyValueOf(GraphicalObjectToUser.ImposeOnLocalUserCanSeeObject)

Figure 4.6 Two rules that set computed columns in entity tables from the relational tables. Slots are specified by the data type name and slot name separated by a period (".."). The function "anyValueOf" indicates each target slot (i.e., specified on the left side of the "=") is set by any number of values in the input slot (i.e., the argument of the "anyValueOf" function) of the related instances by using an "Or" function that has N arguments, where N can change if more relationships are created.

```
1)    GraphicalObject.ShowOnHMD =
            MainMemory.LocalIsHMD
            And
            GraphicalObject.AnyUserLookingCannotSee
            And
            GraphicalObject.LocalUserCanSeeObject

2)    GraphicalObject.ShowOnWall =
            MainMemory.LocalIsWall
            And
            Not(GraphicalObject.AnyUserLookingCanNotSee)

3)    GraphicalObjectToUser.ImposeOnLocalUserCanSeeObject =
            User.IsLocal
            And
            GraphicalObjectToUser.UserCanSeeObject

4)    GraphicalObjectToUserDisplay.ImposeOnAnyUserLookingCanNotSee =
            GraphicalObjectToUserDisplay.UserLookingAtRelatedDisplay
            And
            Not(GraphicalObjectToUser.UserCanSeeObject)
```

Figure 4.7. Rules that set computed columns. Slots are specified by the data type name and slot name separated by a period (“.”). Target slots (i.e., left side of the “=”) are set by Boolean functions (i.e., right side of the “=”), which can be between any number of specified input slots. Boolean functions get reevaluated when any of the inputs change.

4.3.3 Example: Distribution

Our implementation of DP consists of a one-way distribution mechanism that allows data types and columns to be sent to other computers in the peer-to-peer system. The User and Display data types have the same replication strategy as the User table defined in Section 3.4.2.3: each computer that has a user and/or display inserts rows that are distributed to every other computer in the distributed system. Every User and Display slot is replicated, except the “isLocal” slot, which is initially set to “TRUE” on the local computer and “FALSE” on all remote computers. No other columns are needed for distribution; however, the visibility flag “UserLookingAtRelatedDisplay” in the “GraphicalObjectToUserDisplay” table represents whether a User is looking at a particular object, just like the “Visible” slot in the Visibility table in Section 3.4.2.3 (Figure 3.32). Therefore, instead of distributing this slot, we set this value to the “Visible” slot in the related Visibility table row (i.e., looking it up with the UserName and ObjectID), since this information might already be distributed (if any of the rules defined in Section 3.4.3 are currently being used). If it isn’t being distributed, since the Visibility algorithms are

executed on each User's computer, all computers are notified to send Visibility information for this particular Object.

4.4 Related work: Data-based and knowledge-based graphics

Researchers have explored several different approaches to implementing graphical systems based on tabular representations of relational data. Many of these systems were introduced over 20 years ago, but have been developed further. Some of this previous work claims that these techniques simplify programming, improve developer productivity, allow seamless integration of programs and data, and permit interactive creation and modification of applications [138, 139]. By separating the data representation that describes what is rendered, applications can independently specify what is rendered by adding, changing, and modifying a Relational DataBase Management System (RDBMS). The systems then provide mechanisms to convert the information in the database to the underlying graphics implementation to generate the display. Feiner et al. [48] applied this technique to store relational information in a database about graphical documents and their contents to support authoring and interactivity.

Using the mechanisms provided by RDBMS can also enhance the functionality of the system. Spooner [127] improves the control of the database by developing an interface that uses a general purpose database language for the flexible control of high-level graphical entities without the need to know the underlying graphics system. Garrett and Foley [60] integrate an in-memory production rule system with an RDBMS to support integrity constraints and triggers for maintaining data dependencies. This allows programmers to explicitly control the dependencies between data, including relationships between the application and graphical data.

Knowledge-based systems have powerful representation capabilities that are flexible enough to allow developers to make subtle changes in the user interface to explore the differences in benefits between UI designs with the same functionality [50]. Complex characterizations of data have been developed for exploring functionality in user interfaces that have some automatic design properties [95, 142]. More specific representations have also been applied to AR [15, 44, 70, 93, 122]. By storing data in relations within the same memory space, which allows the design of arbitrary multi-

dimensional data structures, applications gain flexibility, functionality, and understandability.

Since DP exemplifies the analogy between programming systems and databases, much database research also applies to DP. Active Database Systems [31, 39, 108] focus on different mechanisms to build complex logic into database systems, such as triggers and transactions, rules that support integrity constraints, and event-condition-actions (ECA). Database design [13, 82] is well understood and will help DP programmers balance the amount of memory used against the need for application performance.

4.5 Comparing DP to other distributed AR systems

There have been quite a few frameworks proposed for distributed AR systems. These systems provide benefits for developing distributed systems, but they also present difficulties for developing the real-time examples shown in this thesis. We compare our DP implementation with other systems, discuss the differences, and evaluate some of the benefits and drawbacks of using DP.

4.5.1 Shared memory systems

Most distributed systems that share memory use either Distributed Shared Memory (DSM) [89] or Distributed Object Memory (DOM) [12]. Both present an illusion of one large shared entity distributed among many computers, either at the memory level (DSM) or programming language object level (DOM). These systems determine what kind of replication strategy is needed based on program functionality (reading/writing data): if remote systems need to read data frequently, then the system duplicates data throughout the distributed system for easy local access. This enhances the speed of data reads, while degrading the performance of the changes (i.e., writes) needed to provide accurate synchronization, since each copy must be updated. These decisions can be determined by programmer specification, static compiler analysis, and/or automatically at run-time using statistics [12].

A good way of looking at the pitfalls of distributed shared memory is to examine data replication, a well studied problem in database research [64]. It is well known that an *update-anywhere-anytime-anyway* replication strategy is unstable. No matter what type of update strategy is used, whether it is using eager or lazy propagation methods, group, master, or two-tier methods of ownership, each strategy creates unwanted results,

such as deadlocks, longer waits, and system inconsistencies [65]. The only correct solutions, suggested by Gray et al. [64], deal with using semantic tricks that impose on design, such as using timestamps (which some shared memory systems use to provide consistency) and commutative transactions (in which the writer has no knowledge of a shared memory system).

The intention of providing a high-level abstraction (or an illusion of shared memory) to each machine in a distributed network is to allow programmers to focus on application logic rather than data communication or synchronization. This is extremely useful for creating simple distributed examples quickly and easily, but it is more difficult to program distributed behaviors [93]. In these programs, there are two types of variables defined: shared (between all computers) and local variables (specific to each individual computer). To develop programs that can logically control what is displayed on multiple computers from distributed information, shared variables can be created for each computer, but there are no mechanisms to easily create logic (and code) with this information. Each situation needs to be carefully programmed so that the local information depends on the computer-specific information that is shared.

Combining shared memory models into graphical systems has led to a number of distributed software architectures. In Coterie [94], distributed virtual environments are built using Shared Objects with objects in a structured graphics library based on scene graphs, called Repo-3D [92]. Similarly, the Studierstube Augmented Reality Project [122] enhances a popular scene graph implementation Open Inventor [69] for distributed use, which makes it easy for developers to move into distributed development using a familiar programming style and environment. Although these systems provide easy ways to develop distributed graphical systems, they still have the same issues that exist in shared memory systems when developing complex distributed logic and replicating updates.

In contrast, since our distribution mechanism in DP is one-way, it is extremely easy to create computer-specific information (e.g., a local addition to a table that is distributed to all), but a little bit more complicated to create shared information, since this requires defining local variables that can be set from the distributed information received from all of the computers. However, this makes it much easier to programmatically control shared information and implement different strategies, such as security and reserva-

tion control. In DP, since each piece of distributed data is limited to one master source, it is also much easier to maintain consistency throughout the distributed system: there is no dependence on each computer's time clock (hence, no distributed clock synchronization is necessary), and it is much easier to determine data cleanup without the need for complex garbage collection that requires further reference analyzing. Instead of using reconciliations to resolve conflicting transactions, such as the joint checking account problem [64], DP deals with collisions programmatically by requiring the design of tables, rows, and possibly distribution strategies, to have the flexibility for determining the resolutions.

4.5.2 Component-based message systems

Component systems [23, 91] are frameworks that manage services that reside in separate processes or threads in a distributed environment. The DWARF (Distributed Wearable Augmented Reality Framework) project [15] focuses on understandability, dynamic distributed reconfiguration, and modularization by organizing separate services, or processes that are "units of independent production, acquisition, and deployment" [134]. DWARF is a large-scale software development project that includes graphical development tools and provides distributed functionality such as service configurations, system monitoring, data flow configuration, and modeling of multimodal interactions.

These architectures are efficient for systems that are easily modularized into components that are distributed and need to communicate with each other. Once this modularization is developed, the system and architecture can dynamically adjust configurations based on hardware resources and system loads. In component systems, a graphics rendering component is typically implemented for each display, since each component is a process and it is difficult to combine graphics from multiple processes on one display. As more complex development occurs, it is more difficult to modularize the graphical component. Thus, the graphical systems do not benefit from the component architecture.

Message systems can also create excessive message traffic. Components residing on the same machine send messages rather than share memory between them. Since only one DP instance needs to reside on each computer, all of the processing on one machine shares memory. This allows DP development to focus on programmatically minimizing messages between computers, not processes, based on distributed hardware architecture design for real-time performance. It is more difficult to develop real-time component

services that do not know anything about the distributed architecture because of the excessive message passing when more information in the system needs to be shared.

From a memory usage perspective, having separate address spaces block components from sharing information, while also creating a fake sense of abstraction or modularization. Components can be implemented simply as a function, or as complex as an entire program. Introducing new functionality into a component system requires decisions that are made based on what data is used for computation. Programmers need to choose between creating a new component (occasional computations done with messages), adding an existing component (frequent computations within a component), and/or combining components (frequent computations between components). Instead of making decisions that fight component modularization, in DP all program variables are accessible, so that new functions can focus on the logic without these partitions.

4.5.3 Data-flow systems

Modeling the relationships between inputs and outputs of functions into a data-flow is a common topic in programming [74], and has been used by many researchers for developing systems, such as visual programming languages [84] and designing flexible interactive graphical systems [67]. Significant work has been done building software architectures focusing on processing input devices for developing interaction techniques [104], processing input devices for virtual and augmented reality environments [111], and for building visualization toolkits (e.g., VTK) [123].

In DP, data-flow is used extensively by allowing programmers to specify input-dependent slots to connect inputs and outputs of functions. For these types of functions, whenever the input values change, the function is re-executed. While many of these techniques on visualizing data-flows can be applied to DP, DP also provides two additional concepts to the data-flow paradigm: the tabular structure for organizing large data-flows, and sequential processing that use action/broadcast slots for functions that need to be executed frequently, such as for every frame rendered in a graphical system. This is much more difficult to represent in a data-flow, since executions do not depend on the inputs or outputs of the functions.

Chapter 5

Conclusions and Future Work

In this dissertation, we have developed a new approach to designing and enhancing graphical user interfaces based on the space management and view management. We have applied this new approach to many different graphical applications to explore its potential, demonstrate its benefits, and outline a framework for future use and research.

In this chapter, we first give a summary of our contributions and how these novel ideas have come to fruition. We then describe some high-level ideas that give some insight to the underlying concepts of view management. It is important to understand the scope of the work done, what types of applications will likely use these techniques, and how these techniques can be applied further to applications outside of the scope of this thesis. View management is an extremely general concept, and similar work has been done in previous research, as referenced in Section 3.1. However, our techniques, based on controlling what users see, give flexibility to graphical interfaces that greatly surpass the previous work.

Finally, we discuss some of the future work that can build on our contributions. There are many levels of improvements for our work: from low-level data structures and implementations, to high-level application-based decisions. We describe a number of these possible improvements and how they could influence the design of future user interfaces. Since our work has been published, there has also been some interesting related work in the field. We will discuss how this work applies to and influences our future directions.

5.1 Summary of contributions

Throughout this thesis, each successive contribution described follows a natural progression of our research on view management, from 2D interactive presentations, to 3D distributed immersive and interactive systems. We present these specific research results:

- A dynamic multidimensional spatial representation and wide range of techniques to use it for choosing desired layouts that avoid overlapping already existing objects (i.e., Space Management).

- The view-management computational pipeline, which describes how we apply our techniques in detail to any type of 3D user interface.

- An annotated situational-awareness aid, based on a "world in miniature," used in many parts of this thesis as a hands-free tool for providing AR and VR users a god's eye view of an environment for an improved understanding.

- A categorization of generalized view management based on how users' priorities are controlled, to determine what users see.

- Data Programming (DP), a new interactive rule-based system designed for the development of complex interactive distributed applications that are flexible and easily extensible.

The source of these contributions and the basis of many ideas start with the first contribution on space management. This spatial data structure was initially developed when working on the MAGIC multimedia presentation system, developing the 3D timeline shown in Figure 2.14c. I was developing numerous mock-ups of time charts by hand, in both 2D and 3D, contemplating questions such as "why is 3D important?" and "how am I going to do this automatically?" Lots of work and several iterations later, I realized that I was moving objects in a 3D VRML file based on their 2D projections, thus requiring me to go back and forth between changing the definition file and loading the file in the viewer until I had the correct layout. In this case, logically I decided to project the objects to the view plane to mimic what I was doing manually, but still had a problem determining layout for the informative file folders. The nature of this particular data structure, which uses upright rectangles to represent arbitrary spaces, was perfect for what our system needed to do: determine a place on the screen, specified in 3D coordinates, in which information could be placed without overlapping other projections of objects that were deemed important.

Although the final revision of the presentation system was in 2D, shown in Figure 2.14b, it was not soon after that I started applying this spatial data structure to general 3D environments, both virtual and augmented reality. By combining this space representation with typical computer graphics techniques in visibility algorithms, we were able to

generate our first attempt at labeling and annotating 3D environments. This lead to the view management paper [18], along with the Emerging Technologies demonstration [46] held at SIGGRAPH 2001 in Los Angeles in collaboration with NRL (Naval Research Labs), InterSense, and Eyeled.

A main focus of this SIGGRAPH 2001 demonstration was to show through AR labels and annotations about other demonstrations at the conference. From the demonstration layout on the conference floor, we noticed that many of the booths were not visible from our booth, and therefore most of the labels and annotations for the booths would not be shown relative to the physical world. For this reason, I created a virtual model of the conference floor, which evolved into the situated-awareness aid that allowed users to easily browse and inquire about all of the booths in the exhibition. Users could transfer their attention between the aid and the physical world using head movement alone, as described in Section 3.4.2.2.

Although the demonstration was successful and our system worked well, our software infrastructure was designed in an ad-hoc method. Menu systems, programmable components, and distribution mechanisms were constrained and did not work well for extensibility. Also, the need to build more functionality pressed us to redevelop parts of the system using rule-based systems. The benefit of using JESS, the rule engine for the java platform [54], is that it is tightly integrated into JAVA, and we did not need to completely redesign our system from scratch. Instead, we choose the necessary variables for which we wanted to develop rules, while keeping the rest of the design the same. One drawback to doing this is that we eventually built rules for all variables in the system; thus it would have been the same to redesign from scratch. Our rule-based system evolved into our DP infrastructure mainly because of performance and program organization, as described in Section 4.1. During this transformation, our system was also used for the MAGIC multimedia presentation system, developed for the New York Presbyterian Hospital. This was the beginning of the development of a software infrastructure that had the flexibility to support both a complex automatically generating multimedia presentation system and our AR system. This was beneficial for building our view-management component, which is used for developing and providing common functionality that can be employed in any type of graphical application, as described in Section 3.3.

Throughout the development of DP, we made an effort to explain it in simple terms so that others can benefit by using it. By developing simple applications with functionality that is difficult to implement with current programming styles, we tried to make it easy to understand. First, I started developing extremely simple graphical applications, but the simplicity did not help since it was easy to see how to develop using other programming styles. Then, by extending DP for distributed use, the importance of developing network programs without continually redesigning protocols has been accentuated.

Since distributed DP provides functionality to control logic on many computers, the next logical step seemed straightforward: to apply this control to view management. It is important to consider a ubiquitous environment where head-worn displays can be used in conjunction with desktop, wall-mounted, and handheld displays. By generating view-management techniques independent of which computers are connected to which displays, the complexity increases along with the implementations. We have successfully developed DP into a real-time distributed graphical system that provides many directions for future work.

5.2 Screen space as a resource

Screen space is like any other resource in computers, such as Input/Output (IO), Random Access Memory (RAM), Disks, or Processors. Most of these other resources have well known management techniques, whether it involves compiler design, operating system thread management, or database cache strategies. Most graphical layout techniques previous to ours have little control over changing what is displayed (i.e., resource allocation) without completely replacing it. For example, the default behavior of the most frequently used interface (i.e., web browser) completely reloads the entire page every time a link is chosen by the user. Both popup windows and changing tasks in a window manager drastically introduce information into the screen real estate without any knowledge of what screen space (i.e., screen resources) is currently used. Other techniques for screen space management, such as zoomable interfaces [16] and fisheyes [58] reserve space for specific behaviors that can only be used for certain types of applications.

By drawing this analogy between a computer resource and screen space, the fundamental concepts of view management are formed. If the user needs to get information

from the system, the system allocates screen space to show it. If the user no longer needs to see information from the system, or if the system needs to allocate space that is not currently available, then the system makes logical decisions to remove information (i.e., deallocate screen space). By matching these simple rules with the users' priorities dependent on the situation and application, many of the view management decisions shown in this thesis are defined.

However, these operations only include satisfying the instantaneous need for information during the users' perceptual and immediate response times, as described in the cognitive coprocessor by Robertson et al. [115]. We also want to impose view management on other types of situations in designing user interfaces. One most notable situation is when multiple objects are initially laid out, and the layout between the objects define a structure to help users understand relationships between the objects, such as graphs [95], cone trees [116], and time lines [112]. These frameworks are typically predesigned and used for conveying specific information in applications, such as the structure diagram and timeline in our presentation system shown in Figure 2.14. To allow view management in these situations, these frameworks were specifically designed with available screen space, so additional information could be placed throughout the presentation using our spatial layout strategies.

5.3 Target usage: What applications need view management?

Most of the examples in this thesis focus on using view management to introduce information to an already existing user interface. Some information is understood by simply glancing and reading, such as labels. Other information is more complex, and takes time for users to cognitively process not only the information displayed, but also its related information, both on the display and off. To make the situation more difficult, users might be physically doing something while information is being displayed, which can distract them from understand.

A good example of this complexity is in the MAGIC system, described in Section 2.7.2, which was completed in the Cardio Thoracic Intensive Care Unit at New York Presbyterian Hospital. This multimedia presentation system shows information to nurses in an intensive care unit about the patient's status after surgery, right before the patient's arrival. Although most of the popup information on the structure diagram and inside of

the file folders has little data, usually consisting of a list of drugs with dosages, patient condition and measurements, or an image, this medical information has complex relationships that are used by the nurse to get prepared to attend the patient. At the same time, since the nurse is actively preparing the drugs needed to care for the wounded patient, his/her attention is switching between the presentation and the drug containers, syringes, and drips. Instead of using a head-worn display to show the nurse the presentation independent of there head position, the presentation uses voice to describe the information shown graphically. Thus, speech synthesis can be used as an alternative, or in conjunction with view management to assist the users in receiving information.

Our framework presented in this thesis has allowed space management and view management be used in any 2D or 3D graphical application. There are already many commercial endeavors that could use our techniques in a worthwhile manner, including applications with many windows [2], web sites that use simple techniques for popping up information inside of a web page [42], and mapping software that could present data such as traffic conditions and restaurant locations [140]. There is also a major drive to capture models of the entire earth (both 2D and 3D) [83] and integrate it with information from the internet into one universally accessible interface [62]. In these kinds of applications, view management is essential to the development of general purpose interfaces that provide embedded information in the context of an already structured interface, map, or image.

To explore what applications best fulfill the need for view management, we must target situations where people require continuously updated data for success. It is difficult to envision the application without understanding what action (or possible actions) users will do when reacting to information. This is why our applications are heavily dependent on the information: an ideal situation where a user needs to make a decision based on some information displayed, which forces the system to display more information based on the user's actions and priorities, and possibly those of multiple users in a distributed environment. However, this ideal behavior cannot always be possible in highly specialized fields and immersive environments (e.g., the intensive care unit system), where the system doesn't have enough information to infer exactly what the user needs to do. Thus, we are left to display information to assist users in making their own

decisions, but it is important to make sure the system chooses the correct information at any given time.

5.4 Future work

In this dissertation, we have developed a comprehensive set of techniques based on view management. Along the way, each contribution has provided new dimensions for solving problems and introducing new functionality, some of which have not been completely drawn out, whether it has been for complexity or lack of time. In this section, we describe some of the possibilities for future work.

5.4.1 Space representation

We made the decision to represent both full and empty space as upright rectangular shapes for convenience and practicality. We were able to get good results by using the data structures and queries described. However, a number of variations on our system are possible. Some of these involve the way in which the system is used, without modifying the system itself:

- *Approximations to hierarchy.* The user can support hierarchical containment by creating a number of space managers. Each of the space managers would then be treated by the user as managing a set of objects associated with a containing object at a higher level of the hierarchy. Alternatively, recall that any full-space rectangle wholly contained within another full-space rectangle has no effect on the empty-space representation, regardless of the order in which the contained and containing objects were added. Thus, if the user adds a full-space rectangle *F* that completely covers a set of full-space rectangles already in the representation, the largest empty-space rectangles that were contained within the area of *F* are deleted from the empty-space representation (although the full-space rectangles remain). Removing *F* will add any largest empty-space rectangles covered by *F*. Thus, adding and deleting a covering rectangle can be used to approximate some of the computational benefits of a true hierarchy.

- *Unbounded workspace representation.* The space manager is currently initialized by the user to work within a bounded rectangular workspace. Alternatively, to support

infinitely zoomable user interfaces [16], it can be used as an unbounded manager, whose workspace ranges from $-\infty$ to $+\infty$ in each dimension (e.g., represented as the largest negative and positive floating point values). When computing window dimensions during a query, the user could optionally clip these to her own, possibly changing, finite min/max bounds.

- *Using 3D spatial representation*. As described in Chapter 2, we have extended the 2D spatial representation to support 3D axis-aligned cuboids, but have not created good examples in which we use this representation to allocate 3D space. One potential use in a collaborative environment could be to find a 3D place that is within view of multiple users. This would require incorporating the 3D spatial representation together with a way to exclude space, possibly by using each user's clipping volumes, to result in a 3D spatial representation that is queried to find results. This might be important if a user wants to introduce a shared 3D virtual object for collaborative use in AR or VR.

- *Rectangular approximations*. By approximating shapes and spaces as upright rectangles, our representation makes a tradeoff between quality and performance. If the shapes are not aligned with the space representation, or they are not exactly rectangular, then it is possible to use multiple rectangles to approximate an object's shape (Figure 5.1). This, in turn, will closer approximate the shapes at a cost of performing more computations on the data structures. In our 3D applications, when using BSP trees for visibility (Section 3.3.3.1), we have experienced drastic effects with certain objects, like fences, where its approximation is a long rectangular shape (similar to Figure 5.1c). In these immersive environments, a subtle head tilt will cause the projection to grow enormously in area on the screen. To prevent this, the angle between the view plane and horizontal length of the object could be used to determine the number of rectangles needed for the approximation.

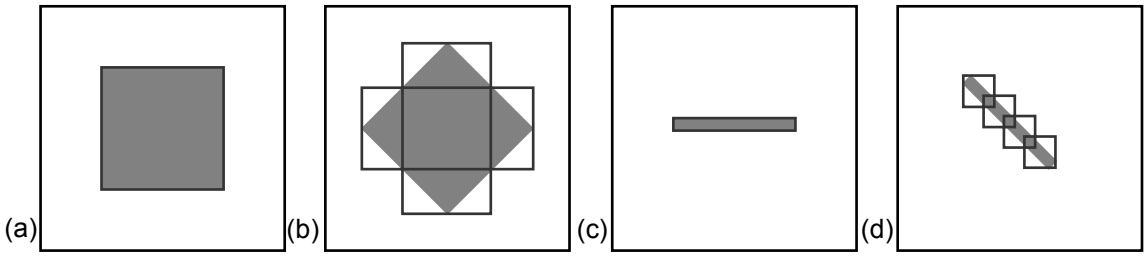


Figure 5.1 Non-upright rectangular objects approximated by multiple rectangles.

Implementing an explicit object geometry modification function or implicitly recognizing groups of deletes and adds that effectively modifies a set of objects, could result in increased efficiency. These variations of our space representation could improve the modification operations on the representation:

- *Area differences in small movements for operations*. When changes in the positions or sizes of a set of full-space objects are sufficiently small, changes needed in the empty space representation are minimal, involving only changes in the coordinates of edges, rather than the addition and removal of largest empty spaces currently caused by a delete/add pair. Similarly, suppose that a modified object's new geometry intersects its original geometry, and the intersection covers a large number of other objects. To avoid processing these other objects during the deletion and addition, the portion of the original geometry outside the intersection may be deleted and the portion of the new geometry outside the intersection added, leaving the intersection untouched.

- *Spaces use pointers to full-space rectangles for edges.* Instead of representing the space rectangles as four boundary numbers (minx, miny, maxx, maxy), each rectangle could be stored with four pointers, one for each edge, to bounded full-space rectangles. When a full-space rectangle is moved, the update would include changing the values in the full-space rectangle, and empty-spaces would only need to get added and deleted if the spatial relationships changed. Unfortunately, it is possible for an empty space to be bound on one side by multiple full-space rectangles, which makes this implementation more difficult. However, if the objects are moving considerably without changing the overall special layout, this strategy could benefit performance in both maintaining and querying the data structure.

New types of spatial data structures based on our representation are useful to explore as well. Some possibilities we have explored include:

- *Supporting arbitrary simplexes.* Eliminating the constraint of approximating objects by upright rectangles could have a huge impact on the quality of results. We believe a good direction for supporting arbitrary shapes would be to use set operations on polyhedra [136], which can represent the empty space using partition planes. Although this representation does not allow an easy way to exhaustively search the empty space for placement, it could provide a way to process the empty space into a form that would be useful and easy to query.

- *Wrap-around spherical space.* Instead of representing the rectangular view plane, the view plane of a 360 degree perspective would require our representation to wrap around its edges, either like a sphere in which the top regions meet at a pole, or like a cylinder, where only the side edges meet. Both of these representations would require the development of new operations and queries. In an immersive 3D environment, these structures represent users' overall perspective view without any field of view limitation, and could be used for enhancing labeling/annotation strategies by minimizing recomputations of layout.

5.4.2 View management

Throughout this thesis, the general idea of view management has been defined, categorized, and used to develop many different examples. Future work on view management will be guided by what types of applications are most important for using new display technologies, and could help certain technologies come to fruition. By combining the types of view management defined in this thesis, applications will be able to improve information visualization and quicker fulfillment of users needs for information in difficult, possibly life threatening situations. We discuss possible applications and how view management could be used:

- *Distributed Display Environments.* It is important to explore ways to present information with distributed displays that include head-worn displays and displays embedded within the environment. While we are becoming surrounded by an increasing

number of viewable displays, from public displays on sidewalks, buildings, and transportation vehicles to indoor displays that include wall-mounted displays, hand-held displays, and television sets, head-worn displays are growing smaller and more practical. Our work is directed toward developing ways of integrating these displays in distributed graphical systems to address contention for each user's limited field of view and attention. As we have exemplified, decisions of which display to use for different kinds of information can be made automatically by the system or controlled interactively by users. In order for these systems to become practical, it is essential that they provide good user interfaces that are easy to understand and control.

- *Improved Layout Strategies.* While we have provided many different layout algorithms in this thesis, along with guidelines on how to use them, this does not rule out the possibility of a new application having layout priorities that are different. These improvements in layout quality would consist of incorporating additional constraints to correspond to the new priorities for the desired results.

- *Pre-rendered Layout Strategies.* Although we have experimented with layouts of pre-rendered material, it has not been a main focus of this thesis. Using pre-rendered material, including material recorded from head-tracked displays, could help us improve many of our current algorithms by finding patterns between view movement and screen movement. For example, if a user's head is moving fast, different strategies might be used compared to when head movement is minimal. Layouts of pre-rendered material would also allow experimentation with look-ahead algorithms and with algorithms that are more computationally intensive compared to what is feasible in a real-time environment.

- *Usability Studies.* The algorithms and demonstrations that we have presented create a wide range of different behaviors in many environments, including 2D desktops, VR and AR. To determine what will work best for users engaged in different tasks, usability studies can be used to measure users' performance using different techniques. Our goal is to compare behavior with and without different versions of the view-management component on modeling tasks in cluttered environments that would appear to be good candidates for its use.

5.4.3 Annotated situation-awareness aid

There are many additional possibilities for using "worlds in miniature" for displaying information about surrounding and remote environments. Some variations we save for future work:

- *Multiple aids.* Exploring how multiple aids are controlled and used to show information about the environment would be useful for when the system wants to point multiple things out in the environment that cannot be seen using one perspective. For example, showing two entrances on opposite sides of a building might need two versions of our aid.

- *Selective distortion.* Showing directions or other kinds of information on the aid could require some modifications and distortions to the 3D model in the spirit of work done on 2D maps by Agrawala and Stolte [3]. By selectively distorting parts of the model that are important, such as turning points and landmarks, the interface can become more understandable by filtering out parts that are not important.

.

5.4.4 Data Programming (DP)

We have already applied DP to a range of different graphical applications including 2D, 3D, distributed, and web-based applications. The future of DP will continue to build on a common infrastructure for many different types of applications, focusing on some of these topics:

- *Graphical development and debugging interface.* The current interface for developing and debugging DP programs is a text-based command-line system. We have begun to create a simple graphical debugging tool that allows visual feedback on tables and views on tables. It is possible to continue this development to create an interactive graphical development environment for DP that is similar to many relational database tools and would allow run-time development and debugging in distributed environments.

- *Better database support.* Although we have created some applications with database communication, the infrastructure for loading and synchronizing data in a database is primitive. Better database support would consist of mapping DP tables to database tables, providing mechanisms for loading parts of a database, synchronizing data asynchronously, and supporting transactions.

Bibliography

[1] JavaScript,http://www.mozilla.org/js/.

[2] Adobe,http://www.adobe.com.

[3] M. Agrawala and C. Stolte, Rendering effective route maps: improving usability through generalization in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* ACM Press, 2001 pp. 241-249

[4] C. Ahlberg, C. Williamson, and B. Shneiderman, Dynamic Queries for Information Exploration: An Implementation and Evaluation, In *Proc. ACM CHI '92*, 1992. pp. 619–626.

[5] A. V. Aho, C. Beeri, and J. D. Ullman, The theory of joins in relational databases *ACM Trans. Database Syst.* , vol. 4 pp. 297-314 1979

[6] J. Ambite, Y. Arens, W. Bourne, P. T. Davis, S. Feiner, E. H. Hovy, J. L. Klavans, A. Philpot, S. Popper, K. Ross, J.-L. Shih, P. Sommer, S. Temiyabutr, and L. Zadoff, A Portal for Access to Complex Distributed Information about Energy, In *National Conference for Digital Government*, Los Angeles, LA, May 2002 2002

[7] K. Arthur, T. Preston, R. Taylor, II, F. Brooks, Jr., M. Whitton, and W. Wright, Designing and Building the PIT: A Head Tracked Stereo Workspace for Two Users, in *Second Int. Immersive Projection Technology Workshop*. Ames, IA, 1998.

[8] P. R. Atherton, A Method of Interactive Visualization of CAD Surface Models on a Color Video Display, In *Proc. SIGGRAPH '81*, 1981. pp. 279–287.

[9] A. Avritzer, J. P. Ros, and E. J. Weyuker, Estimating the CPU utilization of a rule-based system, In *Proceedings of the fourth international workshop on Software and performance*, Redwood Shores, California, 2004. pp. 1–12.

[10] R. T. Azuma, Evaluating label placement for augmented reality view management, In *Second IEEE and ACM International Symposium on Mixed and Augmented Reality*, Oct. 7-10 2003. pp. 66–75.

[11] G. J. Badros, J. Nichols, and A. Borning, SCWM — An Intelligent Constraint-Enabled Window Manager, in *Proc. AAAI Spring Symposium on Smart Graphics*. Stanford, CA, 2000.

[12] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, Orca: a language for parallel programming of distributed systems, *Software Engineering, IEEE Transactions on*, vol. 18, pp. 190-205, 1992.

[13] D. S. Batory and C. C. Gotlieb, A unifying model of physical databases, *ACM Trans. Database Syst.*, vol. 7, pp. 509–539, 1982.

[14] G. D. Battista, P. Eades, R. Tamassia, and I. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs*. USA: Prentice Hall, 1999.

[15] M. Bauer, B. Bruegge, G. Klinker, A. MacWilliams, T. Reicher, S. Riß, C. Sandor, and M. Wagner, Design of a Component-Based Augmented Reality Framework, In *Proc. ISAR '01 (IEEE and ACM Int. Symposium on Augmented Reality)*, New York, NY, October 29--30 2001. pp. 45–54.

[16] B. B. Bederson and J. D. Hollan, Pad++: a zooming graphical interface for exploring alternate interface physics, in *Proc. UIST '94*. Marina del Rey, California, United States: ACM Press, 1994, pp. 17--26.

[17] B. Bell and S. Feiner, Dynamic Space Management for User Interfaces, In *Proc. UIST '00*, San Diego, CA, November 5–8 2000. pp. 239–248.

[18] B. Bell, S. Feiner, and T. Höllerer, View Management for Virtual and Augmented Reality, In *Proc. UIST '01*, Orlando, FL, November 11–14 2001. pp. 101–110.

[19] B. Bell, S. Feiner, and T. Höllerer, Information at a glance, *Computer Graphics and Applications, IEEE*, vol. 22, pp. 6–9, 2002.

[20] B. Bell, S. Feiner, and T. Höllerer, Maintaining Visibility Constraints for View Management in 3D User Interfaces, in *Multimodal Intelligent Information Presentation*, O. Stock and M. Zancanaro, Eds.: Kluwers, 2005, pp. 255–277.

[21] B. Bell, T. Höllerer, and S. Feiner, An Annotated Situation-Awareness Aid for Augmented Reality, In *Proc. UIST (ACM Symp. on User Interface Software and Technology) '02*, Paris, France, October 27–30 2002. pp. 213–216.

[22] M. Bernard and F. Jacquenet, Free space modeling for placing rectangles without overlapping, *Jnl. Universal Comp. Sci.*, vol. 3, pp. 703–720, 1997.

[23] C. Best, M.-A. Storey, and J. Michaud, Designing a component-based framework for visualization in software engineering and knowledge engineering in *Proceedings of the 14th international conference on Software engineering and knowledge engineering* Ischia, Italy ACM Press, 2002 pp. 323-326

[24] M. Billinghurst, S. Weghorst, and T. Furness, III, Shared Space: An Augmented Reality Approach for Computer Supported Collaborative Work, *Virtual Reality*, vol. 3, pp. 25–36, 1998.

[25] J. Blinn, Where am I? What am I looking at?, *Computer Graphics and Applications, IEEE*, vol. 8, pp. 76-81, 1988.

[26] G. Booch, *Object-Oriented Analysis and Design with Applications (3rd Edition)*: Addison Wesley Longman Publishing Co., Inc., 2004.

[27] A. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer, and M. Woolf, Constraint hierarchies in *Conference proceedings on Object-oriented programming systems, languages and applications* Orlando, Florida, United States ACM Press, 1987 pp. 48-60

[28] A. Butz, T. Höllerer, S. Feiner, B. MacIntyre, and C. Beshers, Enveloping Users and Computers in a Collaborative 3D Augmented Reality, in *Proc. IWAR '99 (Int. Workshop on Augmented Reality)*. San Francisco, CA, 1999, pp. 35–44.

[29] M. S. T. Carpendale, D. J. Cowperthwaite, and F. D. Fracchia, Extending Distortion Viewing from 2D to 3D, *IEEE Computer Graphics and Applications*, vol. 17, pp. 42–51, 1997.

[30] E. Catmull, A Subdivision Algorithm for Computer Display of Curved Surfaces. Ph. D Thesis, University of Utah, Salt Lake City, UT, 1974.

[31] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca, Automatic generation of production rules for integrity maintenance, *ACM Trans. Database Syst.*, vol. 19, pp. 367–422, 1994.

[32] J. Christensen, J. Marks, and S. Shieber, An Empirical Study of Algorithms for Point-Feature Label Placement, *ACM Transactions on Graphics*, vol. 14, pp. 203–232, 1995.

[33] Y. Chrysanthou and M. Slater, Computing dynamic changes to BSP trees, in *Computer Graphics Forum (Proc. Eurographics '92)*, vol. 11, no. 3, 1992, pp. 321–332.

[34] J. C. Chung, A comparison of head-tracked and non-head-tracked steering modes in the targeting of radiotherapy treatment beams, in *Proceedings of the 1992 Symposium on Interactive 3D Graphics*. Cambridge, Massachusetts, United States: ACM Press, 1992, pp. 193-196.

[35] J. H. Clark, Hierarchical geometric models for visible surface algorithms *Commun. ACM* vol. 19 pp. 547-554 1976

[36] E. S. Cohen, E. T. Smith, and L. A. Iverson, Constraint-Based Tiled Windows, *IEEE Computer Graphics and Applications*, vol. 6, pp. 35–45, 1986.

[37] M. Dalal, S. Feiner, K. McKeown, D. Jordan, B. Allen, and Y. AlSafadi, MAGIC: An experimental system for generating multimedia briefings about post-bypass patient status, In *Proc. AMIA Annual Fall Symp*, Washington DC, October 26–30 1996. pp. 684–688.

[38] R. P. Darken and J. L. Sibert, A Toolset for Navigation in Virtual Environments, In *Proc. UIST '93*, New York, NY, November 1993. pp. 157–166.

[39] U. Dayal, M. Hsu, and R. Ladin, Organizing long-running activities with triggers and transactions, In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, Atlantic City, New Jersey, 1990. pp. 204–214.

[40] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood, Implementation techniques for main memory database systems, In *Proc. ACM SIGMOD international conference on Management of data*, Boston, Massachusetts, 1984. pp. 1–8.

[41] K. M. Fairchild, S. E. Poltrock, and G. W. Furnas, SemNet: Three-Dimensional Graphic Representation of Large Knowledge Bases, In *Cognitive Science and its Applications for Human-Computer Interaction*, Hillsdale, New Jeresy, U.S.A., May 1988. pp. 201–233.

[42] Farlex, TheFreeDictionary,http://encyclopedia.thefreedictionary.com/.

[43] Feiner, B. MacIntyre, M. Haupt, and E. Solomon, Windows on the World: 2D Windows for 3D Augmented Reality, in *Proc. UIST '93 (ACM Symp. on User Interface Software and Technology)*. Atlanta, GA, 1993, pp. 145–155.

[44] Feiner, B. MacIntyre, and D. Seligmann, Knowledge-based augmented reality, *Communications of the ACM*, vol. 36, pp. 52–62, 1993.

[45] S. Feiner, B. Bell, E. Gagas, S. Güven, D. Hallaway, T. Hoellerer, S. Lok, N. Tinna, R. Yamamoto, S. Julier, Y. Baillot, D. Brown, M. Lanzagorta, A. Butz, E. Foxlin, M. Harrington, L. Nai-

mark, and D. Wormell, Mobile Augmented Reality Systems (Demo), In *SIGGRAPH Conf. Abstracts and Applications*, Los Angeles, CA, August 12–17 2001. pp. 129.

[46] S. Feiner, B. Bell, S. Güven, D. Hallaway, T. Hoellerer, S. Lok, A. Olwal, J. Tang, N. Tinna, and R. Yamamoto, Mobile Augmented Reality Systems (Demo), In *IEEE and ACM ISAR 2001 (International Symposium on Augmented Reality),*, New York, NY, October 29–30 2001

[47] S. Feiner, B. MacIntyre, T. Höllerer, and A. Webster, A Touring Machine: Prototyping 3D Mobile Augmented Reality Systems for Exploring the Urban Environment, in *Proc. ISWC '97 (First Int. Symp. on Wearable Computers)*. Cambridge, MA, 1997, pp. 74–81.

[48] S. Feiner, S. Nagy, and A. v. Dam, An experimental system for creating and presenting interactive graphical documents *ACM Trans. Graph.* , vol. 1 pp. 59-77 1982

[49] S. Feiner and D. Seligmann, Cutaways and Ghosting: Satisfying Visibility Constraints in Dynamic 3D Illustrations, *The Visual Computer*, vol. 8, pp. 292–302, 1992.

[50] J. Foley, C. Gibbs, and S. Kovacevic, A knowledge-based user interface management system, In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Washington, D.C., 1988. pp. 67–72.

[51] J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics: Principles and Practice, 2nd Ed. in C*. Reading, MA: Addison-Wesley, 1996.

[52] C. L. Forgy, On the efficient implementation of production systems., 187, 1979.

[53] C. L. Forgy, Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, *Artificial Intelligence*, vol. 19, pp. 17–37, 1982.

[54] E. Friedman-Hill, JESS: The Rule Engine for the JAVA Platform, http://herzberg.ca.sandia.gov/jess/docs/index.shtml, 1995.

[55] E. Friedman-Hill, Jess: The Java Expert System Shell. Sandia National Laboratories, Livermore, CA, 1998.

[56] H. Fuchs, Z. Kedem, and B. Naylor, On visible surface generation by a priori tree structures, In *Proc. SIGGRAPH*, Seattle, WA, July 14–18 1980. pp. 124–133.

[57] S. Fukatsu, Y. Kitamura, T. Masaki, and F. Kishino, Intuitive Control of "Bird's Eye" Overview Images for Navigation in an Enormous Virtual Environment, in *Proc. VRST '98*. Taipei, Taiwan, 1998, pp. 67–76.

[58] G. W. Furnas, Generalized Fisheye Views, In *SIGCHI Bulletin: Proc. ACM Conf. Human Factors in Computer Systems, CHI*, Boston, Massachusetts, U.S.A., 13--17~ April 1986. pp. 16--23.

[59] T. Furness, The Super Cockpit and Its Human Factors Challenges, in *Proc. Human Factors Society 30th Annual Meeting*. Santa Monica, CA, 1986, pp. 48–52.

[60] M. T. Garrett and J. D. Foley, Graphics Programming Using a Database System with Dependency Declarations, *ACM Trans. Graph.*, vol. 1, pp. 109–128, 1982.

[61] M. Gleicher and A. Witkin, Snap Together Mathematics, In *Proc. Eurographics Workshop on Object-Oriented Graphics*, 1990. pp. 21–34.

[62] Google, Google, Inc.,http://www.google.com.

[63] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*: Addison Wesley, 1997.

[64] J. Gray, P. Helland, P. O'Neil, and D. Shasha, The dangers of replication and a solution, In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, Quebec, Canada, 1996. pp. 173–182.

[65] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco, CA: Morgan Kaufmann, 1993.

[66] A. Gupta, C. Forgy, and A. Newell, High-speed implementations of rule-based systems, *ACM Transactions on Computer Systems (TOCS)*, vol. 7, pp. 119–146, 1989.

[67] P. E. Haeberli, ConMan: a visual programming language for interactive graphics in *Proceedings of the 15th annual conference on Computer graphics and interactive techniques* ACM Press, 1988 pp. 103-111

[68] L. He, M. Cohen, and D. Salesin, The virtual cinematographer: A paradigm for automatic real-time camera control and direction, in *Proc. SIGGRAPH '96*. New Orleans, LA, 1996, pp. 217–224.

[69] G. Hesina, D. Schmalstieg, A. Fuhrmann, and W. Purgathofer, Distributed Open Inventor: A Practical Approach to Distributed 3D Graphics, In *Proc. VRST '99*, N.Y., Decemeber 20--22 2000. pp. 74–81.

[70] T. Höllerer, Ph.D Thesis. Columbia University, New York City, 2003.

[71] T. Höllerer, S. Feiner, T. Terauchi, G. Rashid, and D. Hallaway, Exploring MARS: Developing Indoor and Outdoor User Interfaces to a Mobile Augmented Reality System, *Computers and Graphics*, vol. 23, pp. 779--785, 1999.

[72] D. R. Hutchings and J. Stasko, QuickSpace: new operations for the desktop metaphor, In *CHI '02 Extended Abstracts on Human factors in Computing Systems*, Minneapolis, Minnesota, USA, 2002. pp. 802–803.

[73] E. W. Ishak and S. K. Feiner, Interacting with hidden content using content-aware free-space transparency, in *Proceedings of the 17th annual ACM symposium on User Interface Software and Technology*. Santa Fe, NM, USA: ACM Press, 2004, pp. 189-192.

[74] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, Advances in dataflow programming languages *ACM Comput. Surv.* , vol. 36 pp. 1-34 2004

[75] D. Jordan, K. McKeown, K. Concepcion, S. Feiner, and V. Hatzivassiloglou, Generation and evaluation of intraoperative inferences for automated health care briefings on patient status after bypass surgery, *Journal of the American Medical Informatics Association*, vol. 8, pp. 267–80, 2001.

[76] D. Jordan, G. Whalen, B. Bell, K. McKeown, and S. Feiner, An evaluation of automatically generated briefings of patient status, In *Proc. Medinfo*, San Francisco, CA, September 7-11 2004. pp. 227–231.

[77] S. Julier, M. Lanzagorta, Y. Baillot, L. Rosenblum, S. Feiner, T. Höllerer, and S. Sestito, Information Filtering for Mobile Augmented Reality, In *Proc. ISAR '00 (Int. Symposium on Augmented Reality)*, Munich, Germany, October 5–6 2000. pp. 3–11.

[78] T. Kamada and S. Kawai, An enhanced treatment of hidden lines, *ACM Transactions on Graphics*, vol. 6, pp. 308–323, 1987.

[79] E. Kandogan and B. Shneiderman, Elastic Windows: Evaluation of Multi-Window Operations, In *Proc. CHI '97*, New York, 1997. pp. 29–38.

[80] R. H. Katz, *Contemporary Logic Design*. Redwood City, CA: Benjamin Cummings/Addison Wesley Publishing Company, 1994.

[81] T. L. Kay and J. T. Kajiya}, Raytracing complex scenes, *Computer Graphics (SIGGRAPH '86 Proceedings)*, vol. 20, pp. 269--278, 1986.

[82] W. Kent, A simple guide to five normal forms in relational database theory, *Commun. ACM*, vol. 26, pp. 120–125, 1983.

[83] Keyhole, Keyhole Inc.,http://www.keyhole.com.

[84] J. Kodosky, J. MacCrisken, and G. Rymar, Visual programming using structured data flow, In *Visual Languages, 1991., Proceedings. 1991 IEEE Workshop on*, 1991. pp. 34-39.

[85] D. R. Koller, M. R. Mine, and S. E. Hudson, Head-Tracked Orbital Viewing: An Interaction Technique for Immersive Virtual Environments, in *Proc. UIST '96 (ACM Symposium on User Interface Software and Technology)*, 1996, pp. 81–82.

[86] J. Lasseter, Principles of traditional animation applied to 3D computer animation, In *Proc. SIGGRAPH '87*, 1987. pp. 35–44.

[87] J. LaViola, Jr., D. Feliz, D. Keefe, and R. Zeleznik, Hands-free multi-scale navigation in virtual environments, In *Proc. Int. Symp. on Interactive 3D Graphics*, March 2001. pp. 9–15.

[88] T. J. Lehman and M. J. Carey, Query processing in main memory database management systems, In *Proc. SIGMOD International Conference on Management of Data*, Washington DC, 1986. pp. 239–250.

[89] K. Li and P. Hudak, Memory coherence in shared virtual memory systems *ACM Trans. Comput. Syst.* , vol. 7 pp. 321-359 1989

[90] J.-M. Lien and N. M. Amato, Approximate convex decomposition in *Proceedings of the twentieth annual symposium on Computational geometry* Brooklyn, New York, USA ACM Press, 2004 pp. 457-458

[91] D. H. Lorenz and J. Vlissides, Designing components versus objects: A transformational approach, in *Proceedings of the 23rd International Conference on Software Engineering*. Toronto, Ontario, Canada: IEEE Computer Society, 2001, pp. 253-262.

[92] B. MacIntyre, Exploratory Programming of Distributed Augmented Environments. Columbia University, New York City, 1999.

[93] B. MacIntyre and S. Feiner, A Distributed 3D Graphics Library, in *Computer Graphics (Proc. ACM SIGGRAPH '98) Annual Conference Series*. Orlando, FL, 1998, pp. 361–370.

[94] B. MacIntyre and S. Feiner}, Language-Level Support for Exploratory Programming of Distributed Virtual Environments, in *Proc. UIST '96*. Seattle, WA, 1996, pp. 83–94.

[95] J. Mackinlay, Automating the design of graphical presentations of relational information, *ACM Trans. Graph.*, vol. 5, pp. 110–141, 1986.

[96] Macromedia, Flash,http://www.macromedia.com/.

[97] A. P. Mangan and R. T. Whitaker, Partitioning 3D surface meshes using watershed segmentation, *Visualization and Computer Graphics, IEEE Transactions on*, vol. 5, pp. 308-321, 1999.

[98] K. McKeown, D. Jordan, S. Feiner, J. Shaw, E. Chen, A. Shabina, A. Kushniruk, and V. Patel, A study of communication in the cardiac surgery intensive care unit and its implications for automated briefing, In *Proc AMIA Symp*, Los Angeles, CA, 2000

[99] M. Mine, R., F. Brooks, P. , Jr., and C. Sequin, H., Moving Objects in Space: Exploiting Proprioception In Virtual-Environment Interaction, in *Proc. ACM SIGGRAPH '97*. Los Angeles, CA, 1997, pp. 19–26.

[100] D. Miranker, A new and efficient match algorithm for AI production systems. Ph. D dissertation, Columbia University, New York, Jan. 1987.

[101] P. Mishra and M. H. Eich, Join processing in relational databases *ACM Comput. Surv.* , vol. 24 pp. 63-113 1992

[102] T. Mori, K. Koiso, and K. Tanaka, Spatial Data Presentation by LOD Control Based on Distance, Orientation, and Differentiation, in *Proc. of Int'l Workshop on Urban Multi-Media/3D mapping (UM3'99)*. Tokyo, Japan, 1999, pp. 49–56.

[103] K. Oflazer, Partitioning in parallel processing of production systems. Ph. D. dissertation, Carnegie-Mellon University, Pittsburgh, 1986.

[104] A. Olwal and S. Feiner, Unit: modular development of distributed interaction techniques for highly interactive user interfaces in *Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and Southe East Asia* Singapore ACM Press, 2004 pp. 131-138

[105] J. K. Ousterhout, Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools, *IEEE Trans. on Comp.-Aided Design of Integrated Circuits & Sys.*, vol. CAD-3, pp. 87–100, 1984.

[106] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor, Magic: A VLSI layout system, In *21st Proceedings of the Design Automation Conference on Design automation*, Albuquerque, New Mexico, 1984. pp. 152–159.

[107] M. S. Paterson and F. F. Yao, Optimal binary space partitions for orthogonal objects in *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms* San Francisco, California, United States Society for Industrial and Applied Mathematics, 1990 pp. 100-106

[108] N. W. Paton and O. Diaz, Active database systems, *ACM Comput. Surv.*, vol. 31, pp. 63–103, 1999.

[109] R. Pausch, T. Burnette, D. Brockway, and M. Weiblen, Navigation and Locomotion in Virtual Worlds via Flight into Hand-Held Miniatures, In *Proc. SIGGRAPH '95*, 1995. pp. 399–401.

[110] C. B. Phillips, N. I. Badler, and J. Granieri, Automatic Viewing Control for 3D Direct Manipulation, In *Proc. SIGGRAPH '92*, Cambridge, MA, March–April 1992. pp. 71–74.

[111] W. Piekarski and B. H. Thomas, Tinmith-evo5 - a software architecture for supporting research into outdoor augmented reality environments, University of South Australia, Technical report 2002.

[112] C. Plaisant, B. Milash, A. Rose, S. Widoff, and B. Shneiderman, LifeLines: visualizing personal histories in *Proceedings of the SIGCHI conference on Human factors in computing systems: common ground* Vancouver, British Columbia, Canada ACM Press, 1996 pp. 221-ff. .

[113] G. Riley, CLIPS: A Tool for Building Expert Systems,http://www.ghg.net/clips/CLIPS.html, 1985.

[114] L. G. Roberts, The Lincoln Wand, In *Proceedings of the Fall Joint Computer Conference*, San Francisco, California, November 1966 1966

[115] G. Robertson, S. K. Card, and J. D. Mackinlay, Information Visualization Using 3D Interactive Animation, *Communications of the ACM*, vol. 36, pp. 57--71, 1993.

[116] G. G. Robertson, J. D. Mackinlay, and S. K. Card, Cone Trees: animated 3D visualizations of hierarchical information in *Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology* New Orleans, Louisiana, United States ACM Press, 1991 pp. 189-194

[117] J. Roessel, An Algorithm for Locating Candidate Labeling Boxes within a Polygon, *The American Cartographer*, vol. 16, pp. 201–209, 1989.

[118] C. Rohrer and J. Boyd, The rise of intrusive online advertising and the response of user experience research at Yahoo!, In *Extended abstracts of the 2004 conference on Human factors and computing systems*, Vienna, Austria, 2004. pp. 1085–1086.

[119] H. Samet, *The Design and Analysis of Spatial Data Structures*. Reading, MA: Addison-Wesley, 1990.

[120] C. Sandor, B. Bell, A. Olwal, S. Temiyabutr, and S. Feiner, Visual end user configuration of hybrid user interfaces (Demo), In *Proceedings of the 2004 ACM SIGMM workshop on Effective telepresence*, New York, NY, 2004. pp. 67–68.

[121] R. W. Scheifler and J. Gettys, The X window system *ACM Trans. Graph.* , vol. 5 pp. 79-109 1986

[122] D. Schmalstieg, A. Fuhrmann, G. Hesina, Z. Szalavári, L. M. Encarnação, M. Gervautz, and W. Purgathofer, The studierstube augmented reality project, *Presence: Teleoper. Virtual Environ.*, vol. 11, pp. 33–54, 2002.

[123] W. J. Schroeder, K. M. Martin, and W. E. Lorensen, The design and implementation of an object-oriented toolkit for 3D graphics and visualization, In *Proceedings of the 7th conference on Visualization '96*, San Francisco, California, United States, 1996. pp. 93-ff.

[124] R. A. Schumaker, P. Brand, M. Gilliland, and W. Sharp, Study for Applying Computer-Generated Images to Visual Simulation, U.S. Air Force Human Resources Laboratory AFHRL-TR-69-14, 1969.

[125] C. Shaw, M. Green, J. Liang, and Y. Sun, Decoupled simulation in virtual reality with the MR toolkit, *ACM Transactions on Information Systems (TOIS)*, vol. 11, pp. 287–317, 1993.

[126] B. Shneiderman, Direct manipulation: A step beyond programming languages, *IEEE Computer*, vol. 16, pp. 57–69, 1983.

[127] D. L. Spooner, Database support for interactive computer graphics, In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, Boston, Massachusetts, 1984. pp. 90–99.

[128] T. Starner, S. Mann, B. Rhodes, J. Levine, J. Healey, D. Kirsch, R. Picard, and A. Pentland, Augmented Reality through Wearable Computing, *Presence*, vol. 6, pp. 386–398, 1997.

[129] R. Stoakley, M. Conway, and R. Pausch, Virtual Reality on a WIM: Interactive Worlds in Miniature, In *Proc. CHI '95*, Denver, CO, May 7–11 1995. pp. 265–272.

[130] S. Stolfo and D. Shaw, DADO: A tree-structured machine architecture for production systems., In *Proc. National Conference on Artificial Intelligence*, Picttsburgh, PA., 1982. pp. 242–246.

[131] Sun Microsystems, Abstract Window Toolkit (AWT),http://java.sun.com/products/jdk/awt/.

[132] I. Sutherland, A Head-Mounted Three Dimensional Display, In *Proc. FJCC 1968*, Washington, DC, 1968. pp. 757–764.

[133] Z. Szalavari, D. Schmalstieg, A. Fuhrmann, and M. Gervautz, Studierstube: An Environment for Collaboration in Augmented Reality, *Virtual Reality*, vol. 3, pp. 37–48, 1998.

[134] C. Szyperski, *Component-Oriented Software, Beyond Object-Oriented Programming*: Addison-Wesley, 1997.

[135] W. Teitelman, A Tour Through CEDAR, *IEEE Software*, vol. 1, pp. 44–73, 1984.

[136] W. Thibault, C. and B. Naylor, F., Set operations on polyhedra using binary space partitioning trees, In *Proc. SIGGRAPH '87*, 1987. pp. 153–162.

[137] J. Viega, M. J. Conway, G. Williams, and R. Pausch, 3D Magic Lenses, In *Proc. UIST '96*, Seattle WA, 1996. pp. pp. 51–58.

[138] D. Weller and R. Williams, Graphic and relational data base support for problem solving, In *Proceedings of the 3rd Annual Conference on Computer Graphics and Interactive Techniques*, Philadelphia, Pennsylvania, 1976. pp. 183–189.

[139] J. W. Wendorf, A simply extended and modified batch environment graphical system (SEMBEGS), *Commun. ACM*, vol. 21, pp. 897–904, 1978.

[140] Yahoo, Yahoo Maps,http://maps.yahoo.com.

[141] S. You, U. Neumann, and R. Azuma, Hybrid Inertial and Vision Tracking for Augmented Reality Registration, in *Proc. IEEE Virtual Reality '99*. Houston, TX, 1999, pp. 260–267.

[142] M. Zhou and S. Feiner, Data Characterizaton for Automatically Visualizing Heterogeneous Information, In *Proc. IEEE Symp. on Information Visualization*, San Francisco, CA, 1996. pp. 13–20.

[143] M. Zhou and S. Pan, Automated authoring of coherent multimedia discourse in conversation systems, In *MULTIMEDIA '01: Proceedings of the Ninth ACM International Conference on Multimedia*, Ottawa, Canada, 2001. pp. 555–558.

[144] M. X. Zhou, The Representation and Usage of a Visual Lexicon for Automated Graphics Generation, In *Proc. International Joint Conference. on AI)*, Nagoya, Japan, August 23–29 1997. pp. 1056–1062.

Appendix A Space Management API

In Chapter 2, we describe an efficient approach for representing, building, and querying empty space. Here we provide some insight into the design of the Space Management source library, which has been distributed to many colleagues. We describe three classes that build on each other through sub-classes:

1) SpaceManager. This class is responsible for computing and maintaining a list of the largest rectangles that represent the empty-space. Data structures include separate interval trees for both the full and empty space rectangles, as well as listeners to allow external classes, including sub-classes, to maintain separate data structures.
2) SpaceManagerWrapper implements SpaceManager. This class adds additional functionality including queries, a sorted list of empty space rectangles to improve performance for certain queries, and a mechanism that implements undoable-adds. A Hashtable for the full-space objects has also been added to ensure each full-space rectangle has a unique id.
3) SpaceManagerDisplayWrapper implements SpaceManagerWrapper. This class adds an additional ordered list of full-space rectangles and is used to uphold the consistency of the drawing order for the full-space objects in the demonstration application.

```
public class SpaceManager {
      IntervalTree spaceRectangles;
      IntervalTree objectRectangles;

      /* Listener Mechanism */
      /**
       * The callbacks for listeners that want to be notified each time
       * a empty-space gets added or deleted from the space manager.
       * Since each operation (add and delete) adds and deletes a list
       * of empty spaces, it notifies the callbacks when lists get added
       * and deleted as well.
       */
      Vector spacemanagerlisteners;
      public interface SpaceManagerListener {
            /**
             * Function is called when a list of empty-space rectangles
             * are added and deleted to SpaceManager representation when
             * listening.
             *
             * @param fr Full-space rectangle added or deleted
             * @param added The list of empty-space rectangles added.
             * @param deleted The list of empty-space rectangles
             *                deleted.
             */
            public void addDeleteSpaceRectangles(FullRectangle2i fr,
                                                 Collection added,
                                                 Collection deleted);
      }
      public void addSpaceManagerListener(SpaceManagerListener sml);
      public void removeSpaceManagerListener(SpaceManagerListener sml);
```

```
}
```

```
public class SpaceManagerWrapper extends SpaceManager
      implements SpaceManager.SpaceManagerListener {
      /**
       * spaceTrees: Spaces that are sorted by area
       */
      TreeSet spaceTrees;
      /**
       * objectHash: Lookup which provides direct access
       *          to full-space rectangles given a unique id.
       */
      Hashtable objectHash;

      /**
       * RedoAction - This class keeps track of the changes to the
       * empty-space representation during an add operation of a full-
       * space object.  This is used to do "undoable" adds, so that the
       * delete operation does not need to be executed for full-space
       * rectangles that are deleted either right after it has been
       * added, or in an "undo/redo" stack of successive
       * addition/deletions of full-space rectangles.
       */
      public class RedoAction {
            /**
             * Rectangle that has been added, and will be deleted
             * if the RedoAction occurs
             */
            FullRectangle2i rectangle;
            /**
             * List of empty-spaces that has been added to the
             * representation during the operation.
             */
            Vector addedSpaces;
            /**
             * List of empty-spaces that has been deleted to the
             * representation during the operation.
             */
            Vector removedSpaces;
            /**
             * Constuctor for RedoAction.
             * @param fr the full-space rectangle that has been added.
             * @param added the list of empty-spaces that were added
             *          during operation.
             * @param deleted the list of empty-spaces that were
             *    deleted during operation.
             */
            public RedoAction(FullRectangle2i fr, Collection added,
                              Collection deleted);
      }
      /**
       * doRedoAction()  - This action "undos" an add operation to the
       * representation. It takes the added empty-spaces and removes
       * them, and adds the empty-spaces that were deleted. It also
       * removes the full-space.  Warning: If this function gets
       * called and the RedoAction is not "undone" correctly in order,
```

```
 * the empty-space representation will not be correct.  Only if
 * the add operation was the last operation that occurred, or if a
 * sequence of addition operations have been "undone" in the
 * opposite order they were added.
 *
 * @param ra RedoAction that represents an addition operation that
 *          is being undone.
 */
public int doRedoAction(RedoAction ra);

/**
 * SpaceManagerWrapperListener - provides a callback for the
 *    SpaceManagerWrapper each time empty-spaces are added
 *    and deleted from the representation.  The empty-space
 *    data-structure gets maintained.
 */
public interface SpaceManagerWrapperListener {
      /**
       * addRedoRectangles - Function is called when a list of
       *          empty-space rectangles are added and deleted
       *          to SpaceManager representation when listening.
       *
       * @param fr Full-space rectangle added or deleted
       * @param added The list of empty-space rectangles added.
       * @param deleted The list of empty-space rectangles
       *                deleted.
       */
      public void addRedoRectangles(RedoAction ra);
}

/**
* Constructs a SpaceManagerWrapper that represents a specified
* area.
*
* @param  rs the area in which the Space Manager manages.
*/
public SpaceManagerWrapper(Rectangle2i rs);

/**
 * Adds a full-space to the Representation. When this method gets
 * called, the empty-space representation gets updated by having a
 * number of largest empty-space rectangles added and deleted.
 *
 * @param  rs the full-space rectangle specified.
 *
 */
 public FullRectangle2i addRectangle(FullRectangle2i rs);
/**
 * Delete a full-space from the representation.  When this method
 * gets called, the empty-space representation gets updated by
 * adding and deleting a number of largest empty-space rectangles
 * to and from it.
 *
 * @param rs the Full-space rectangle being deleted
 *
 */
 public int deleteRectangle(FullRectangle2i rs);
```

```
        /*
         * QUERIES
         *
         */
        /**
         * Returns all of the empty-space rectangles sorted by Area.
         */
         public Iterator getSpacesOrderedByArea();

        /**
         * Returns the Largest-Empty Space with the largest area.
         *
         */
         public Rectangle2i getLargestSpace();

        /**
         * returns the closest empty-space from specified rectangle.
         * @param rect rectangle specified
         */
        public Rectangle2i getClosestSpace(Rectangle2i rect);

        /**
         * Returns the largest rectangle that lies within the empty-space
         * with at least the given proportions.
         *
         *  @param prop specified minimum proportions
         */
        public Rectangle2i getLargestProportion(Dimension prop);

        /**
         * getClosestWithAtleastSizeAspectRatioClosestToBoth -
         * This query is used for finding the closest rectangle for a
         * popup object to a given:
         * screen object: screenRect and its
         * last position: prevDest given its minimum and
         * maximum width and height it can have (minx, miny, maxx, maxy),
         * its aspect ratio (ratioxtoy) and
         * the buffer between the screen object (buffer)
         */
    public Rectangle2i getClosestWithAtleastSizeAspectRatioClosestToBoth
                        (Rectangle2i screenRect, Rectangle2i prevDest,
                         int minx, int miny, int maxx, int maxy,
                         double ratioxtoy, int buffer);

        /**
         * Returns the closest rectangle from specified rectangle with
         * given dimensions.
         * Note: There can be a tie for closest rectangle, in this case
         *    we take the first one we come across, meaning the one that
         *    is enclosed with the empty-space that has the largest area.
         *    In this case, distance is measured by computing the distance
         *    between the closest points that lie within the two
         *    rectangles.
         * @param rect specified rectangle
         * @param dw specified height of returned rectangle
         * @param dw specified width of returned rectangle
```

```
     */
    public Rectangle2i getClosestWithExactDimension
                                    (Rectangle2i rect,int dw, int dh);

     /**
      * Returns the closest rectangle from specified rectangle with at
      * least the given height and width.
      * Note: There can be a tie for closest rectangle, in this case
      * we take the first one we come across, meaning the one that is
      * enclosed with the empty-space that has the largest area.
      * Also, if any of the rectangles overlap the specified
      * rectangle, then we take the rectangle that overlaps the most
      * area of the specified rectangle.
      *
      * @param rect specified rectangle
      * @param dh specified height of returned rectangle
      * @param dw specified width of returned rectangle
      */
    public Vector getClosestWithAtLeastDimension
                             (Rectangle2i rect,double dw,double dh);

    /**
     * Returns whether any space overlaps the specified rectangle
     * @param rect specified rectangle
     */
    public Boolean isAnySpaceInsideRectangle(Rectangle2i rect);

    /**
     * Returns a list of rectangles that represents the space inside
     * the given rectangle.
     *
     * @param rect specified rectangle
     */
    public Vector getSpacesInsideRectangle(Rectangle2i rect);

    /**
     * Returns a empty space inside the specified rectangle that has
     * the largest area
     *
     * @param rect specified rectangle
     */
    public Rectangle2i getLargestAreaInsideRectangle
                                               (Rectangle2i rect);

    /**
     * Returns whether any empty space encloses the specified
     * rectangle.
     * @param rect specified rectangle
     */
    public Boolean isCompletelyEnclosedBySpace(Rectangle2i rect);
}
```

```
public class SpaceManagerDisplayWrapper extends SpaceManagerWrapper {
      Vector objectVector;
      /**
       * Constructs a SpaceManagerDisplayWrapper that represents a
       * specified area
       *
       * @param  rs the area in which the Space Manager manages.
       */
       public SpaceManagerDisplayWrapper(Rectangle2i rs);

      /**
       * Adds a full-space to the Representation. When this method gets
       * called, the empty-space representation gets updated by having a
       * number of largest empty-space rectangles added and deleted.
       *
       * @param  rs the full-space rectangle specified.
       *
       */
      public FullRectangle2i addRectangle(FullRectangle2i rs);

      /**
       * Adds a full-space to the Representation. When this method gets
       * called, the empty-space representation gets updated by having a
       * number of largest empty-space rectangles added and deleted.
       *
       * @param  rs the full-space rectangle specified.
       * @param  place the place of the full-space rectangle in ordered
       *          list that is returned
       *
       */
      public FullRectangle2i addRectangle(FullRectangle2i rs,int place);

      /**
       * Delete a full-space from the representation.  When this method
       * gets called, the empty-space representation gets updated by
       * adding and deleting a number of largest empty-space rectangles
       * to and from it.
       *
       * @param rs the Full-space rectangle being deleted
       */
       public int deleteRectangle(FullRectangle2i rs);

      /**
       * Returns the closest full-space rectangle (i.e. rectangle drawn
       * last) for display purposes.
       *
       * @param rectarg rectangle specified
       *
       */
      public FullRectangle2i getClosestRectangle(Rectangle2i rectarg);

      /**
       * Deletes the full-space rectangle that is drawn last and
       * overlaps the rectangle specified. This is used to find the
       * full-space rectangle to remove from the representation when a
       * user clicks on the screen in delete mode.
```

```
         *
         * @param rectarg rectangle specified to check for overlapped
         *                full-spaces
         */
        public int deleteRectangleWindow(Rectangle2i rectarg);
}
```