Virtual Reality in Assembly Simulation — Collision Detection, Simulation Algorithms, and Interaction Techniques

Dem Fachbereich Informatik
der Technischen Universität Darmstadt
eingereichte

Dissertation

zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
von

Dipl.-Inform. Gabriel Zachmann

Referenten der Arbeit:	Prof. Dr.-Ing. Dr. h.c. Dr. Eh J. L. Encarnação
	Dr. Carolina Cruz-Neira
Tag der Einreichung:	29. Mai 2000
Tag der mündlichen Prüfung:	10. Juli 2000

To my wife Biggi

and my little daughter Mirjam

Acknowledgements

I would like to thank Professor Dr. Encarnação for his advice and support. The Fraunhofer Institute for Computer Graphics, which he is head of, has been a great work environment.

I am grateful to Dr. Cruz-Neira for accepting co-advisorship and for travelling a great distance to attend my defense.

From my work in many stimulating projects, I owe thanks in particular to Dipl.-Ing. Antonino Gomes de Sá, Dipl.-Ing. Rainer Klier, and Dipl.-Ing. Peter Zimmermann.

Special thanks go to all current and former members of our extraordinary group "Visualization and Virtual Reality", of which Dr. Stefan Müller is the department head: Dr. Peter Astheimer, Uli Bockholt, Dr. habil. Fan Dai, Dr. José Dionisio, Dr. Wolfgang Felger, Torsten Fröhlich, Dr. Thomas Frühauf, Dr. Martin Göbel (former department head, now with GMD), Dr. Helmut Haase, Elke Hergenröther, Udo Jakob, Dr. Kennet Karlsson, Christian Knöpfle, Wolfram Kresse, Stefan Lehmann, Bernd Lutz, Mimi Lux, Wolfgang Müller, Dr. Alexander del Pino, Alexander Rettig, Marcus Roth, Frank Schöffel, Dr. Florian Schröder, Andrzej Trembilski, Dr. Matthias Unbescheiden, Gerrit Voss, Jens Weidenhausen, and Dr. Rolf Ziegler. Our secretary Renate Gröpler deserves special mention, because she has shielded me from quite a bit of paper work. *A4 is simply the best*.

I also would like to thank all my research assistants and students for their efforts (roughly in chronological order): Reiner Schäfer, Andreas Flick, Axel Feix, Jochen Ehnes, Andreas Giersig, Andreas Zieringer, Van Do, Stefan Hoppe, Andreas Hess.

Last but not least, my wholehearted thanks go: to my wife Biggi, for her patience, for drawing some of the figures, and for proof-reading the whole thing (it is understood that all bugs still persisting are mine, not hers); to my little daughter Mirjam, who has witnessed the final phase and cheered it up with her cuteness; to my parents and brothers, for showing the way in life and for always being there.

Gabriel Zachmann, Darmstadt, Mai 2000

Contents

Chapter 1

Introduction

Sie sitzen schon, mit hohen Augenbrauen,
gelassen da und möchten gern erstaunen.

GOETHE, Faust, Vorspiel auf dem Theater

Although research in virtual reality has been done for over 10 years,[1] only a few years ago the non-academic world started to evaluate its use to solve real-world problems. As of this writing, virtual reality is in the process of leaving the realm of purely academic research. Among others, the automotive industry is evaluating its potential in design, development, and manufacturing processes [DR86, DFF+96]. In fact, the automotive industry has been among the first, but others, such as suppliers, have begun to evaluate VR, too.

While simulators (flight simulators in particular) have been in regular use for several decades, and Boom-like displays as well as head-tracking have been devised in the '60s [Sut68] (see Figure 1.1), it seems that the field of *virtual reality*[2] has come into existence only when the so-called "data glove" [DS77, ZLB+87, SZ94] and 6D hand (or body) tracking were invented.[3]

Some of the first research efforts predating VR have been the "Put-that-there" project at MIT [Bol80], UNC's "Walk-thru" project [Bro86], the "Virtual environment display system" at NASA Ames' [FMHR86], and Myron Krueger's more artistic "artificial reality" applications [Kru83].

[1] At VRAIS '98, David Mizell has remarked that "every computer graphics program after 1990 is a VR system".

[2] Throughout this book, I will use a *slanted* font for introducing technical terms, while I will use an *emphasized* font for emphasis.

[3] One could argue that tracking was actually invented much earlier, namely with master-slave manipulator arms in the '50s, or even earlier yet during the Renaissance with the pantograph [Pan98].

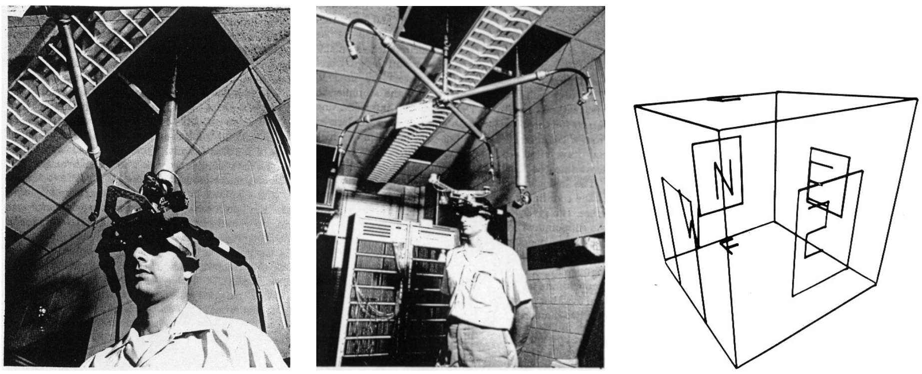


Figure 1.1: A bit of VR folklore: The first HMD was, to my knowledge, developed by Sutherland [Sut68]. It featured stereoscopic display of 3,000 lines at 30 frames/sec, line clipping, and a mechanical and ultrasonic head tracker.

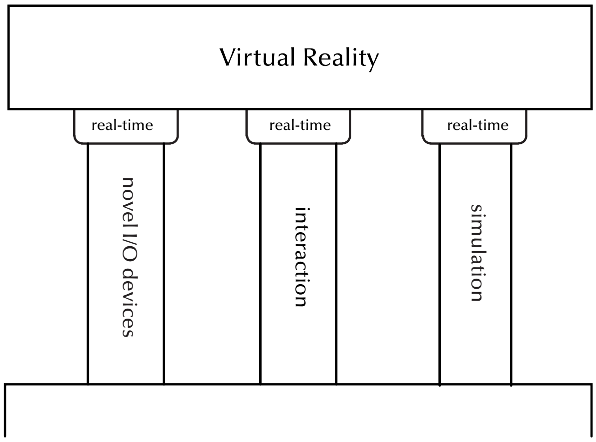


Figure 1.2: Virtual reality rests on three pillars.

In the beginning, there were a number of terms for the new field: *artificial reality*, *virtual reality*, and *cyberspace*, just to name a few. The popular press particularly favored the latter. However, early on it became clear to the scientific community that it must not allow the popular press as well as the scientific press to hype virtual reality [JBD+90] in order to avoid suffering the same fate as AI. Fortunately, the popular media has somewhat lost interest in "cyberspace" while interest from industries is constantly increasing.

From a historical point of view, virtual reality is just a logical consequence of an on-going *virtualization* of our every-day life [EDd+96]. However, the social impacts of this development are not clear at all at this point.

The definition of virtual reality involves three components (see Figure 1.2):

1. real-time interaction,
2. real-time simulation,
3. immersion and direct interaction by novel I/O devices.

From a more technical point of view, a VR system must meet three criteria:

1. interaction with the virtual environment must be immersive and intuitive,
2. rendering must be done in real-time and without perceptible lag (20 Hz for graphics, 500 Hz for haptics),
3. object behavior must be simulated in real-time.

Despite promising research efforts in the beginning of virtual reality, there were a number of unsolved problems, some of which still persist:

1. For several automotive applications rendering hardware was not fast enough. For moderate scenarios, rendering speed is still too slow by a factor of about 10. For a complete car rendering has to be faster by a factor of 100 still.

2. Creating (authoring) virtual environments for highly interactive scenarios with complex object behavior and complex interactive processes was a time-consuming task.

   Chapter 2 presents a *framework* which is a significant step towards a solution.

3. Data integration with existing IT infrastructures of large companies has not yet been solved in a satisfactory manner. This is partly due to the lack of standards meeting the needs of VR, which is partly a consequence of the current[4] graphics API confusion and very dynamic graphics hardware market.

4. Real-time physically-based simulation of non-trivial object behavior has been recognized as one of the major missing ingredients. Behavior includes rigid body dynamics, inverse kinematics, flexible objects, etc. All of these problems have been solved in theory and in non-real-time systems. However, to my knowledge, it was not possible to simulate that behavior in real-time for non-trivial object complexities and numbers (i.e., several tens or even hundreds of objects, each consisting of some 10,000–100,000 polygons).

   The most time-consuming part of many simulation problems is *collision detection*. In Chapter 3 several algorithms are developed to tackle this fundamental task.

   In Section 4.5.4, an algorithm is presented for simulation of the sliding behavior of objects being moved by the user, such that the object does not penetrate other objects while following the user's hand.

5. Human-computer interface devices were very immature. They were cumbersome, clumsy, inaccurate, and limited. This includes tracking as well as visual and haptic/tactile rendering.

   In order to improve electro-magnetic tracking and render it applicable to serious shopfloor applications, *filtering* and *correction* algorithms are developed in Section 4.3.1 and Section 4.3.2, resp.

6. From an interaction point of view the human-computer interface to virtual environments is still a field of active research. New intuitive, immersive metaphors have to be invented, because porting classical WIMP[5] metaphors has been found to be inadequate and inefficient.

   In Chapter 4, *techniques*, algorithms, and software architectures are discussed to improve interaction with virtual environments.

7. In 1995 there was no "real" application of VR on a routine basis. For 1–2 years, however, VR is being used for simple applications (such as styling or design reviews) in some productive processes.

   Based on the framework and algorithms presented in this thesis, a virtual assembly simulation application (see Section 5.2) is developed, which is being integrated in the product process of a major manufacturing company.

These were the most severe difficulties in making VR practicable for virtual assembly simulation. The goal of this work is to establish the thesis statement:

Assembly simulation using virtual reality is feasible.

Although the solutions presented in this dissertation are targeted mostly at the manufacturing domain (the automotive industry, in particular), most of them are applicable to VR systems in general.

[4] as of 1998/99

[5] Windows, Icons, Menus, Pointers

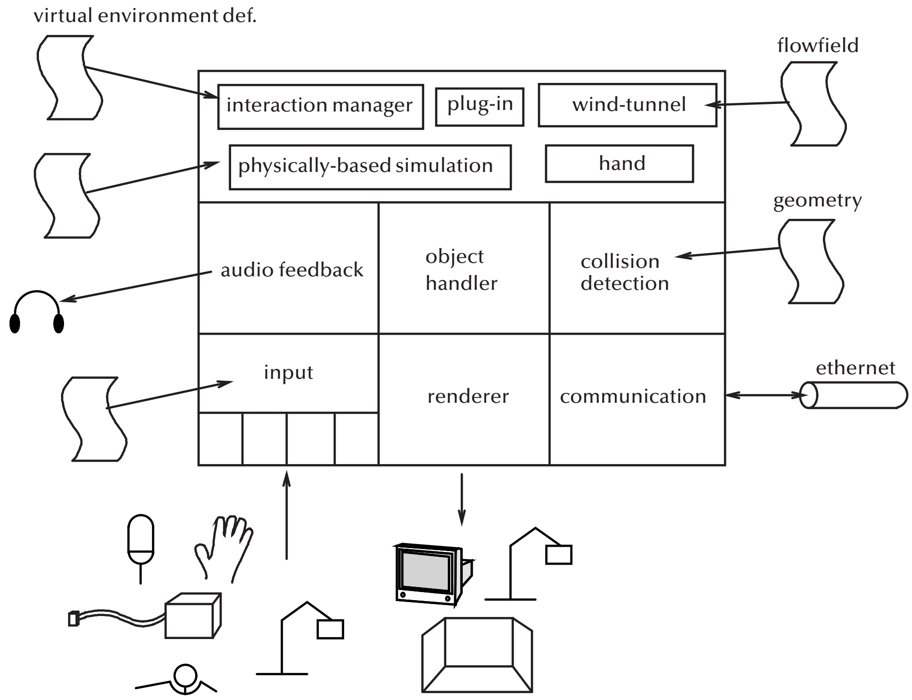


Figure 1.3: The object and scene graph manager is the central module of virtually any VR system. All other modules building on it "simulate" or render a certain aspect of the virtual environment. Some of those are controlled by the interaction module (e.g., sound renderer and device drivers); others are "peer" (e.g., physically-based simulation).

1.1 Architecture of VR systems

A complete VR system is a large software system, consisting of many modules. Every VR system contains an object manager, renderer, device drivers, communication module, navigation and interaction module, and, usually, physically-based simulation, sound rendering, scientific visualization, application-specific modules, etc. (see Figure 1.3).

The visual part of a virtual world is represented by a hierarchical scene graph. Everything is a node in this graph: polyhedra, assemblies of polyhedra, LODs, light sources, viewpoint(s), the user, etc. Most of the functionality and interaction presented below will operate on the scene graph, i.e., it will, eventually, change some attribute(s) of some object(s). There are commercial packages providing scene graph manager and renderer, for instance Performer, Inventor, and Fahrenheit. Our own VR system is currently (as of 1999) based on our own object manager [Rei94], but we plan to port it to Fahrenheit.

The modules at the top of Figure 1.3 are those "visible" to the user — they provide functionality to be invoked directly by the user. The virtual environment management system (VEMS) is responsible for most interactions with the user. It is driven by a specification for a VE which tells what action to perform when a certain input is received. This will be described in detail in Section 2.2.

At the bottom I have placed the input device layer (among others), although it does not directly provide functionality or input to the object handler (this is more to stress the fact that the object handler is very central). It provides an abstract interface to a number of input devices. This layer is used mostly by the VEMS. I will describe it in more detail in Section 4.1.1.

Close to the object handler is the collision detection module (see Chapter 3). It is responsible for detecting collisions among objects in the scene graph. In my implementation, it is integrated in the object handler, so that it notices automatically when objects have moved. This module is used mostly by physically-based simulation modules and by the interaction handler.

Plug-ins are modules which provide some application-specific behavior or functionality, or an on-line interface to other applications such as CAD systems. They can be loaded at run-time by the VR system.

Almost all modules should be able to run concurrently to each other. This is particularly necessary for real-time critical modules such as the renderer, physically-based simulation, and collision detection module. In our VR system, the renderer, collision detection, device drivers, and wind tunnel simulation modules can run concurrently, as well as some of the physically-based modules.

Of the modules described so far, I have implemented the interaction manager (VEMS), the collision detection module, and the device layer and drivers, in addition to some physically-based simulation and several plug-ins for many applications.

1.2 Overview

This section provides a brief overview of my thesis, which is organized in four main chapters. Each of them tackles a specific area of problems related to the difficulties in making VR practicable and a wide-spread tool for real-world applications, in particular virtual assembly simulation.

Chapter 2 presents an object-oriented framework for describing (authoring) virtual environments (VEs). After reviewing briefly some related approaches, a new paradigm is developed. This paradigm allows to describe VEs without programming and it is intuitive enough to be used by non-programmers. In addition, it facilitates specialized high-level graphical user-interfaces for building applications in special application domains. Furthermore, it has been designed with multi-user VR in mind. This framework has proven to be suitable, powerful, and flexible; all VR applications for manufacturing customers have been built on top of this framework.

Chapter 3 presents several algorithms for detecting collisions of polygonal objects. This is the major bottleneck of many simulations, in particular physically-based simulations. In order to achieve real-time behavior of objects, this problem must be solved. This work presents several algorithms for several classes of objects, which are among the fastest algorithms known today.[6] Algorithms for finding quickly pairs of objects possibly intersecting are presented. Several ways of parallelizing collision detection algorithms have been implemented and are evaluated. Finally, a collision detection pipeline is developed comprising all the algorithms presented earlier. In addition, issues such as robustness, concurrency and other implementation issues are discussed.

Chapter 4 deals with various issues relating to user interaction. A framework for integration of input devices is presented and lessons learnt using and

[6] As of 1998

working with it are discussed. One of the problems of electro-magnetic tracking (a common tracking technique) is noise. In order to reduce this I present a filtering pipeline which have been implemented in our VR system. Another problem with electro-magnetic tracking is distortion which leads to warped images and can cause interaction difficulties in precision tasks. A simple and fast method to correct these distortions is provided. The precision of this method is evaluated both with real data and with mathematical experiments. In addition, the amount of distortion of different tracking systems has been evaluated. In the second part of this chapter, a framework for navigation is presented (which has been implemented in the VR system), as well as a framework for the user's head. The discussion is focused in particular on practical issues of interaction paradigms.

Finally, Chapter 5 describes several applications, which have been built on top of the frameworks and algorithms presented earlier. They prove the usefulness, flexibility, and power of the algorithms and frameworks developed in this thesis, helping to make VR more practicable than it used to be.

Chapter 2

Simulation of Virtual Environments

Those who can, do.
Those who can't, simulate.
ANONYMOUS

Computer simulation is the discipline of designing a model of an actual or theoretical real system, executing the model on a digital computer, and analyzing the execution output [Fis96]. The overriding objective of any simulation is making a correct decision.[1]

In order to simulate a virtual environment, first it must be described. This desciption can then be executed by the VR system. This chapter presents a framework for authoring (describing) virtual environments and the interaction between the user and that environment.

Classifications

Virtual environments can be classified by several criteria (see Figure 2.1): whether or not they are "real" (or could be, for that matter) [Zac94a]; whether they are already existing (i.e., they reflect some existing real-world environment), or whether they will exist some time, or have ceased to exist, or will never exist; finally, they differ in being remote, local, or scaled environments.

An example of a "real", existing, yet heavily scaled VE is the NanoManipulator [TRC+93, GHT98]. Most VR training simulators create an existing,

[1] Of course, for VR there are applications where supporting a decision is not the goal, such as entertainment.

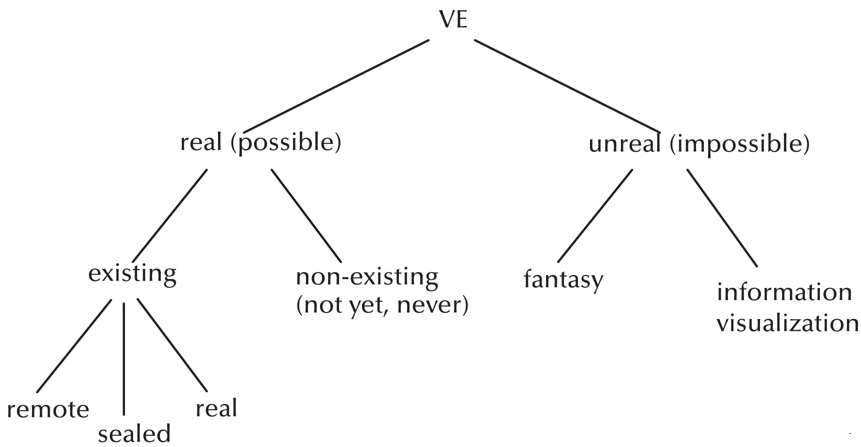


Figure 2.1: Virtual environments can be classified in a number of different types.

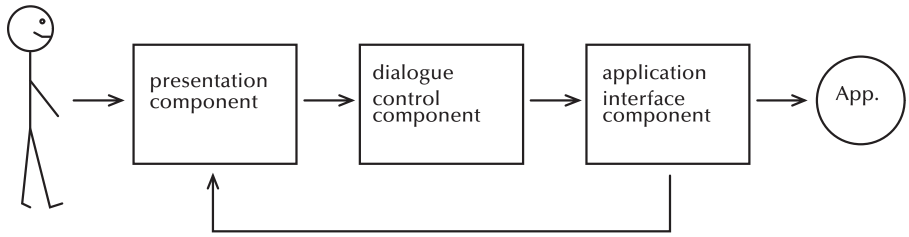


Figure 2.2: The Seeheim model modularizes UIMSs into three components.

true-scale VE, which is more remote, distributed, and multi-user (e.g., astronaut training [Lof95]), or more local and single-user (e.g., surgeon training [MZBS95]). Most VEs in entertainment (such as [AF95]) are in category "unreal and fantasy". Scientific visualization in VR usually creates a "real", already or possibly existing, true-scale VE [DFF+96]. On the other hand, information visualization in VEs creates completely "unreal" data spaces [FB90a]. The goal of virtual prototyping is to create VEs which are possible and "real", and some of which will exist in the future.

2.1 Describing human-computer interaction

One of the tasks of a virtual environment management system (VEMS) is the "dialogue" between user and computer and the maintenance of the user interface. So part of a VEMS is actually a user interface management system (UIMS). Therefore, it makes sense to investigate the possibility to use some of the results of 2D UIMSs.

Several frameworks and dialogue models have been developed to facilitate the description of traditional 2D user interfaces. I will briefly review some of them in the following.

2.1.1 User-interface management systems

A well-known model for the modularization of UIMSs is the *Seeheim model* [Gre84, Pfa85]. It divides user interfaces into three components: a *presentation component* describing the appearance of the interface and dealing with physical input from the user; the *dialogue control component* deals with the syntax and content of the user interface, which is what authoring is mostly about; and the *application interface component* (see Figure 2.2). This threefold decomposition is similar to the well-known *MVC paradigm* as set forth with Smalltalk [GR85] (see Figure 2.3).

The presentation component[2] for 3D graphics systems has been implemented in systems/library such as Performer [RH94], Inventor [Sil92], Y [Rei94], and Into [FSZ94]. The application interface model, "nomen est omen", defines the interface between the user interface and the rest of the application (this component is called *model* in Smalltalk).

Conceptually, the components communicate by passing tokens, consisting of a name (or "type") and data. In general, they will be implemented by function or method calls, callbacks, event and message queues, etc.

[2] The *view* in Smalltalk parlance

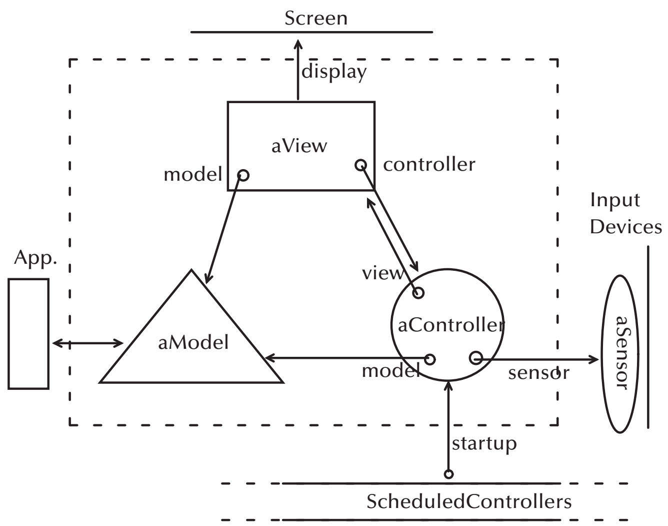


Figure 2.3: The model-view-controller paradigm of Smalltalk-80 proposes a similar decomposition as the Seeheim model.

The dialogue control component (*controller* in Smalltalk) takes the input from the user via the presentation component, performs certain actions as defined by the user interface designer, possibly affects the state of the application, and responds to the user (again via the presentation component).

Several ways of describing the "behavior" of dialogue components have been devised – a few of which I will review briefly in the following.

2.1.2 Transition networks

The transition network model is based on the notion of recursive transition networks (RTN). They are an extension of simple transition networks (STNs), the formal definition of which is a 7-tuple $M = (Q, X, A, \delta, \alpha, q_0, f)$, with

$$
\begin{array}{rl}
Q & \text{a finite set of states} \\
X & \text{a finite set of input symbols} \\
A & \text{a finite set of actions} \\
\delta : Q \times X \to Q & \text{the transition function} \\
\alpha : Q \to A & \text{the action function} \\
q_0 \in Q & \text{the initial state} \\
f \subset Q & \text{the set of final states.}
\end{array}
$$

When M starts in q_0 it receives input symbols in X representing actions performed by the user. The transition function δ determines the next state, while α and the new state determine the name of the action M is to perform. STNs can accept the same class of languages as finite-state machines.

STNs can be drawn very conveniently by digraphs (see Figure 2.4). There are variants of the definition just given which allow actions to be attached to the arcs (i.e., transitions) as well as to the nodes (i.e., states) of the graph. This does not alter the descriptive power of the STN, but it can be more convenient and reduce the number of states needed.

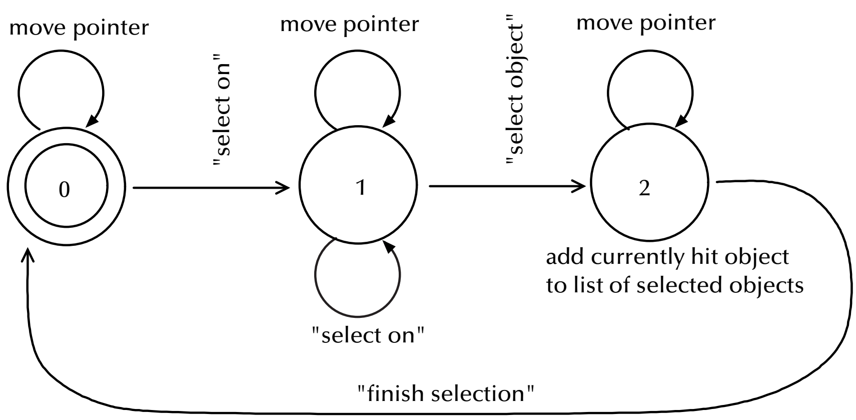


Figure 2.4: Recursive transaction networks are well suited to specify interaction with the user graphically (here, a simple transaction network is shown).

Recursive transition networks (RTNs) are an extension of STNs: arcs can be labeled additionally by *sub-diagrams* (RTNs). The definition is the same as for STNs augmented to be a 9-tuple

$$M = (Q, X, A, Z, \delta, \alpha, q_0, f, Z_0)$$

with

$$\begin{aligned} Z &= \text{the set of stack symbols, with} |Z| = |Q| \\ Z_0 \in Z &= \text{the initial symbol on the stack} \end{aligned}$$

and a slightly changed transition function

$$\delta : Q \times X \;\cup\; Q \times Z \;\longrightarrow\; Q \;\cup\; Q \times Z.$$

The analogue of a RTN in the domain of automata is the deterministic push-down automaton. RTNs can be drawn by a set of disconnected digraphs, where sub-diagrams each have their own initial state and set of final states (which are not part of the formal definition).

A further extension of RTNs are *augmented transition networks* (ATN) [Woo70]. In addition to the set of states and the stack, they are also equipped with a set of *registers* and a set of *functions* operating on the registers. Arcs are labeled by input symbols and functions. A transition can be made only when the correct input symbol has been read *and* the function attached to the arc evaluates to true.

2.1.3 Context-free grammars

The idea behind this model is that the human-computer interaction is a dialogue, which is governed by a grammar. The language of such a grammar is the set of all valid sequences of user inputs.

The formal definition of this model is the same as that for context-free grammars, with the rules augmented by *actions*. This is very similar to the way compiler front-ends are described. A problem is that the flexibility with actions depends on the parsing algorithm used: if parsing is done top-down, then actions can be attached in mid-rule. If parsing is done bottom-up, the accepted language is larger. However, actions can be attached only to the right side, i.e.,

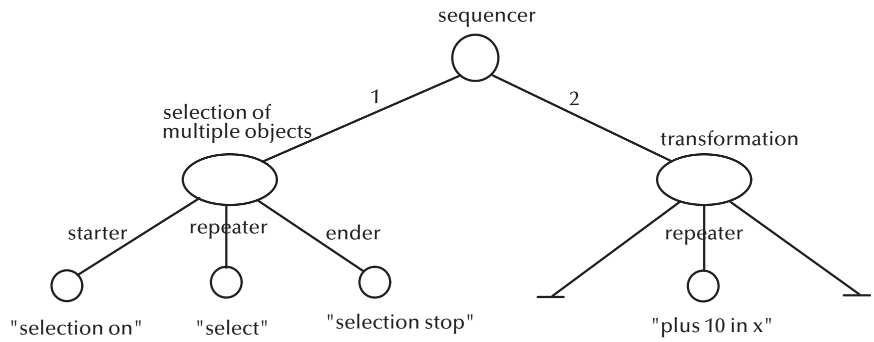


Figure 2.5: Interaction trees combine graphical specification with logical and primitive flow-control constructs.

the action can be executed only after the right-hand side of a production has been entered by the user completely.

Dialogue cells are a different approach of specifying a grammar-based dialogue.

2.1.4 Event languages

From a formal point of view, the event model is not as well established as TNs or CFGs. The event model is based on event languages. Event languages are a programming language paradigm just like object-oriented programming. Usually, event languages are general-purpose programming languages extended with a few extra constructs.

The basic building blocks of event frameworks are event handlers, written in some high-level programming language. Event handlers can be created (i.e., instantiated from a "template") and destroyed at run-time; they can send and receive events.

The input to event handlers are events. Events can be generated by the user (via input devices), and some can be generated inside the dialogue control component. Events convey some data plus a "type". They can be broadcast (to all event handlers having a response defined for it), or they can be sent to a specific event handler. In an extended framework [CCT89], they can also be sent to a certain group of event handlers, much like the multicast feature of UDP. Event handlers can be deactivated without losing their state.

In general, an event is ignored by an event handler, if it does not have a response implemented for it (even if the event has been sent explicitly to that particular handler). However, each event handler can implement an exception response, which deals with events not handled by the other procedures.

An event system provides an event queue for each of its event handlers. When input events arrive from the presentation component, they are added to the ends of each queue the event handler of which has declared an interest for. An event handler processes one event at a time taking them out of its queue. The processing of an event is viewed as an atomic operation. After an event has been processed, some queues might have more events at the end. All event handlers are viewed as concurrent processes.

2.1.5 Interaction trees

A more graphical approach for modeling user interface dialogues is the interaction tree model [Hüb90, Hd89].

The basic idea is to compose interaction dialogues from basic interaction *entities* which are combined to more complex dialogues by *structuring elements*. A dialogue is represented by a tree, in which the leaves are the basic entities and inner nodes are structuring elements. Each node can be triggered, which makes it send that trigger event up in the tree to its parent; or it can be activated by its parent, which puts it in a state where it is waiting for a trigger; or it can be inactive, i.e., waiting to be activated.

Basic interaction entities handle the input which comes from any physical input device, such as a glove or speech recognition system. There are four different types of structuring elements, each implements a different function on its children:

- *Or* gets triggered if one of its children is triggered.
- *And* is triggered when all of its children have been triggered at least once.
- *Sequence* activates and executes its children in a predefined order.
- *Repeat* has exactly three children: a *starter*, a *repeater*, and an *ender*.

For an example see Figure 2.5.

2.1.6 Expressive power of the notations

There are *two measures* for the expressive power of the notations so far: the *descriptive power* and the *usable power*.

The descriptive power of a notation is the set of user interfaces or virtual environments that can be described by the notation. Determining this set can sometimes be converted into a problem in formal language theory.

Much more interesting from a practical point of view is the usable power which is the set of user interfaces that can *easily* be described in the notation. If a certain interface or behavior or chain of actions is hard to describe, chances are that the interface designer or VE author will not do it, but instead change the design of the interface or VE. The usable power will always be a proper subset of the descriptive power. Unfortunately, there is no objective measure for the usable power.

From formal language theory we know that recursive transition networks can parse exactly all context-free languages (by way of, in general non-deterministic, push-down automata) [MAK88], i.e., from a formal point of view, the descriptive power of the transition network model and the context-free grammar model are the same.

ATNs have much more descriptive power than RTNs. Indeed, it seems that ATNs have the same power as event languages (when restricted to programming languages without subroutine calls and loops).

Languages based on the *event model* are widely regarded as the most expressive and flexible notation for the specification of user interfaces. In terms of descriptive power, it has been shown [Gre86] that all TN and CFG models can be transformed into the event model, and that there are user interfaces which can be described by the event model but not by the other two.

However, there are two reasons why event languages should be preferred. First, although transition networks (TN) and context-free grammars (CFG) have been extended, it can still be extremely difficult to handle unexpected user actions or exceptions. For TNs, this can be solved to some degree by wildcard transitions leading to states which try error recovery. Another solution are "error recovery diagrams" associated with each regular diagram. These error diagrams are used when an unexpected input is received [Ols84].

The second disadvantage of TNs and CFGs is that they do not lend themselves easily to the description of *multi-threaded dialogues*. In such a dialogue, the user can be involved in several separate dialogues at the same time.

On the other hand, transition networks do have the advantage that they can be displayed and edited graphically quite easily – at least as long as they do not exceed a certain size.

2.1.7 Scripting languages

For the past 20 years, the class of scripting languages has matured remarkably, unrealized by many people, even programming language scientists. This is a class of programming languages designed for a different task than system languages (like Fortran, C, Smalltalk, Java). While system programming languages are used to build large applications and implement complex algorithms, more or less from scratch, scripting languages are meant to "glue" components together (usually, the components will be implemented in a system language). Therefore, they are sometimes called "glue languages" or "system integration languages".

Although some scripting languages have been utilized to implement the dialogue component of user interfaces (e.g., Tcl and VisualBasic), it will become clear below why I am considering this class of languages here.

Scripting languages can come in any programming language paradigm; however, most of them are imperative or object-oriented. Although the borderline between scripting and system languages is not very sharp, they tend to differ in typing system and efficiency (in several ways).

The major difference is the type system. Other differences, like efficiency, are, to some extent, a consequence of this difference. Most high-level system programming languages are strongly typed languages with a rich type system (number of types, promotion and coercion rules, etc.). By contrast, scripting languages tend to be weakly typed or type-less. They are string-oriented: most/all values are represented by strings.

Scripting languages are much more efficient than system languages in terms of the number of lines of assembly instructions per program line [Ous98]: 1 statement of C produces about 5–10 assembly instructions compared to 100–1000 instructions/statement for Tcl.[3]

In terms of execution speed, scripting languages are less efficient than system languages. Partly, this is due to the fact that they are interpreted instead of compiled. But even when they are compiled, they tend to be slower, because objects are represented by high-level types (variable-length strings, hash tables, etc.) even if that would not be necessary.

On the other hand, interpretation increases productivity, because there are no turnaround times during development, and because code can be changed on-the-fly during run-time of the application.

[3] Lines of code seem to be the smallest unit of a programmer's mental model of a program. An evidence is the fact that programmers produce roughly the same number of lines of code per year, regardless of the language being used [Boe81]. So they are a good measure of programming efficiency as several programmer and programming language surveys have shown.

Summarizing, scripting languages are a valuable complement (not a replacement) to system languages. They are the language of choice if the task involves gluing, flexible system integration of components, or prototyping of rapidly evolving, simple functionality.

2.2 Authoring virtual environments

Describing VEs is commonly named "authoring". A VE author needs to specify (at least) two things:

1. the geometry, scene graph, and materials (the "clay"), and
2. the behavior of objects, and the user interface (the "life").

Note that the latter part not only comprises human-computer interaction, but also the behavior and properties of objects by and themselves. Often, the term "authoring" is used in a narrower meaning denoting only (2), which is what I will use in this section.

Any VR system meant to be used within an industrial process must face the fact that it is just one link in a long chain of software packages (CAD, CAE, FEM, etc.), which might impose a lot of constraints and requirements.

Although VR has been around for about 10 years in the research community, only recently it has become clear that the creation of VEs is a major bottleneck preventing the wide-spread applicability of VR. There is still a lack of tools to make VR an enabling technology in industry and entertainment. Creating virtual worlds is still a cumbersome and tedious process.

In this section, I propose a framework which increases productivity when creating virtual environments (VEs). VE "authors" can experiment and play interactively with their "worlds". Since this requires very low turn-around times, any compilation or re-linking steps should be avoided. Also, authors should not need to learn a full-powered programming language. A very simple, yet powerful script "language" will be proposed, which meets almost all needs of VE creators. As a matter of course, virtual worlds should be input-device independent.

In order to achieve these goals, I have identified a set of basic and generic user-object and object-object interactions which are needed in most applications [Zac96].

For specification of a virtual environment, there are, at least, two contrary and complementary approaches:

- *Event-based.*
  One approach is to write a *story-board*, i.e., the creator specifies which action/interaction happens at a certain time, because of user input, or any other event (see Figure 2.6).

  A story-driven world usually has several "phases", so we want a certain interaction option to be available only at that stage of the application, and others at another stage.

- *Behavior-based.*
  Another approach is to specify a set of *autonomous objects* or *agents*, which are equipped with receptors and react to certain inputs to those receptors (see for example [BG95]).

  So, overstating a little, we take a bunch of "creatures", throw them into our world, and see what happens (see Figure 2.7).

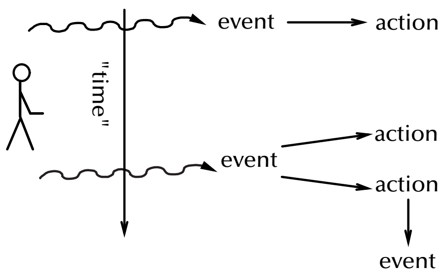


Figure 2.6: The event-based approach to authoring VEs basically specifies a story-board and conditional and temporal relations.

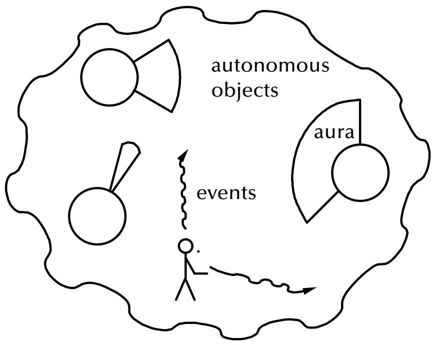


Figure 2.7: A different approach from event-based authoring as behavior based, which focuses on creating autonomous objects.

In the long term, one probably wants to be able to use both methods to create virtual worlds. However, so far the event-based approach has been quite sufficient for virtual prototyping and entertainment applications.

The way VEs are created can be distinguished also by another criterion: the *toolbox approach* versus the *scripting approach*. The toolbox approach involves programming in some high-level language using the VR system's interaction library, while scripting means specification of the VE by some very simple language especially designed for that purpose. It is my belief, that both approaches must be satisfied. For various reasons (see Section 2.1.7), the scripting approach is much less time-consuming; on the other hand, there will be always certain application-specific features (in particular, high-level features) which can be implemented better by the toolbox approach. In the following, I will explain the script based approach, since there is not much to be discussed on the toolbox approach.

All concepts and features being developed below have been inspired and driven by concrete demands during recent projects [PRS$^+$98, Zac98a]. All of them have been implemented in an *interaction-module* (see Section 1.1), which is part of IGD's VR system [ADF$^+$95, AFM93]. The first implementation was in C, while the second implementation is a complete object-oriented redesign in C++.

2.2.1 Design premises

If VR is ever to play a major role in industry, then casual users, like CAD engineers, must be able to create VEs. Therefore, one of the design premises of my description language is that even non-programmers should be able to easily learn and use it on a casual basis. Therefore, any computer science concept like state machines, grammars, flow control, type systems, etc., had to be avoided. This does not necessarily mean that these concepts must not be implemented at all — however, an architect or mechanical engineer should be able to specify as much as possible of his VEs without ever having to worry about them.

The language for specifying VEs will be very simple for several reasons: VE authors "just want to make this and that happen", they do not want to learn yet another complete programming language. Moreover, it is much easier to write a true graphical user interface for a simple language than for a full-powered programming language.

The study of programming languages has shown that in most of the successful languages, particular design goals were constantly kept in mind during the design process [Lou93]. In particular, *generality* and *orthogonality* seem to be of great importance when designing a language. Additionally, *simplicity* should be kept in mind, too.[4] Generality is achieved by avoiding special cases and by combining two closely related features into a more general one. Orthogonality means that language constructs can be combined in any meaningful way without producing "surprises". A language is uniform if similar things or constructs look similar, and, more important, different things look different.

From the point of view of discrete event simulation, the model type I have chosen for the system is the *declarative event-oriented* model, and the model technique is the *script* [Fis95].

2.2.2 Other VR systems

There are quite a few existing VR systems, some commercial some academic. Some of them I will look at briefly in the following.

Sense8's WorldToolkit follows the toolbox approach. Basically, it provides a library with a high-level API to handle input devices, rendering, simple object locomotion, portals, etc.

DIVE is a multi-user, distributed VR platform [HLS97, CH93]. The system can be distributed on a heterogeneous network (making use of the Isis library [Bir85]). New participants of a virtual world can join at any time. They will receive a copy of the current database. All behavior is specified as a (usually very simple) finite state-machine (FSM). Any FSM is part of some object's attributes. Database consistency is achieved by using distributed locks.

Division's dVS features a 2D and 3D graphical user interface to build and edit virtual worlds at run-time [duP95]. Attributes of objects are geometry, light source, sound samples, collision detection parameters, etc. Objects can be instanced from classes within the description file of a virtual world. Inheritance is supported in a simple form. Several actions can be bundled (like a function in C) and invoked by user-defined events. However, the syntax seems to be rather complicated and not really apt for non-programmers. The framework for defining behavior is built on the notions of actions and events. In dVS, however, events denote a very different concept than in my framework. There, events are more like discriminators. As a consequence, actions and triggers are not orthogonal, i.e., the user must know which event is understood by which action.

The Minimal Reality toolkit (MR) [WGS95, HG94] is a networked system, which uses a script file to describe behavior and sharing of objects. Scripted object behavior is compiled into so-called OML code which is interpreted at run-time. For each OML instance there must be an associated C++ class.

Unlike MR, I have not developed objects ("classes") with rather high-level built-in behaviors, such as Tanks, Bombs, or Hills. Instead, I will identify actions on objects on a lower, and therefore more generic, level.

AVOCADO [DEG+97] basically implements the VRML approach on top of Performer, i.e., nodes in the scene graph are augmented by fields and the scene

[4] "Everything should be made as simple as possible, but not simpler." (Einstein)

graph itself is augmented by routes. Nodes communicate and exchange data via routes. From [DEG+97] it is not clear exactly how behavior is implemented in the nodes; but it seems that it can be implemented either by Scheme scripts or by C++ code on top of AVOCADO's API. There seems to be no scripting facility appropriate for non-programmers. The system seems to be biased towards experimental and entertaining applications.

Similarly, the VR-Deck pursues the approach of communicating modules programmed in C++ [CJKL93]. Modules receive anonymous events from a pool, and produce new events and place them in the pool (this is somewhat similar to the Linda framework for distributed systems).

I believe that keeping *both* geometry and behavior (plus maybe other properties like kinematic constraints) of the virtual world in *one* file can be tedious and very inflexible. This is true in particular for application domains like virtual prototyping (see Chapter 5). Therefore, I strictly separate geometry, behavior, physical properties, acoustic properties, etc., in separate files, unlike [HG94, ACHS94, Ghe95]. This greatly facilitates developing virtual environments, because almost always the geometry will be imported from CAD systems (e.g., Catia or ProEngineer), MRI reconstruction algorithms, or animation software (e.g., Alias/Wavefront or SoftImage). During several development iterations, we usually get several versions of the geometry, while we want to keep our VE description files. In addition, for CAD engineers we need to provide a simple GUI (see Section 2.4.2) tailored to their specific application domain, e.g., assembly simulation (see Section 5.2) or styling review.

2.2.3 The AEIO paradigm

The basic idea of the event-based approach (see Section 2.2) is that the user's input creates events which trigger actions, invoke properties, or behavior. For instance, when the user touches a virtual button, a light will be switched on; or, when a certain time is reached an object will start to move. Consequently, the basic components of our virtual worlds are *inputs*, *actions*, *events*, and graphical *objects* — the *AEIO quad*[5] (see Figure 2.8).

Note that actions are *not* part of an object's attributes (in fact, one action can operate on many objects at the same time).

In order to be most flexible and in accordance to our design premises in Section 2.2.1, the action-event paradigm must meet the following requirements:

1. Any action can be triggered by any event. Any event can be fed with any input.

2. Many-to-many mapping: several events can trigger the same action; an event can trigger several actions simultaneously; several inputs can be fed into the same event; an action can operate on many objects at the same time.

3. Events can be combined by boolean expressions.

4. Events can be configured such that they start or stop an action when a certain condition holds for its input (positive/negative edge, etc.)

5. The status of actions can be input to events (loopback).

[5] In the object-oriented programming paradigm, actions, events, inputs, as well as graphical objects are *objects*. However, in the following, I will use the term *object* only for graphical objects.

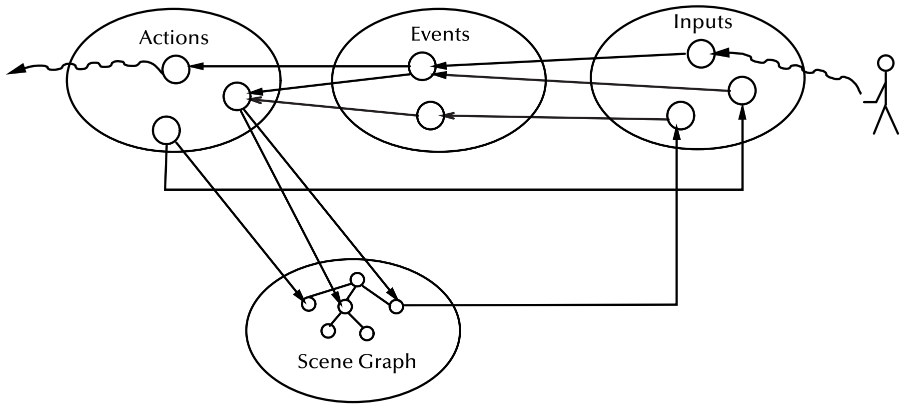


Figure 2.8: The AEIO quad (actions, events, inputs, objects). Anything that can "happen" in a virtual environment is represented by an action. Any action can be triggered by one or more events, which will get input from physical devices, the scene graph, or other actions. Note that actions are not "tied-in" with graphical objects, and that events are objects in their own right.

I do not need any special constructs (as in [MP90]) in order to realize *temporal operators*. Parallel execution of several actions can be achieved trivially, since one event can trigger many actions. Should those actions be triggered by different events, we can couple them via another event. Sequential execution can be achieved by connecting the two actions by an event which starts the second action when the first one finishes. Similarly, actions can be coupled (start-to-start or start-to-stop) with a delay.

Because of the requirements above, we need a way of referring to actions and events. Therefore they can be given a name. Basically, there are two ways to declare an action-event pair in the script:

> *action-name* : *action* . . .
> *event-name* : *event* . . .
> *action-name event-name*

or

> *action* . . . *event* . . .

where *action* and *event* in the latter form cannot be referenced elsewhere in the script.

Most actions operate on objects, and many events have one or two objects as parameters. In order to achieve an *orthogonal language*, those objects can have any type (geometry, assembly, light source, etc.) whenever sensible.

2.2.4 Semantic attributes, object lists, and object groups

Almost all manipulations within a VE eventually affect some objects. Therefore, these are the building blocks of a VE (the "virtual bricks", if you will). Almost all objects have a graphical representation, the *geometry*, and a set of *graphical attributes*. In general, this is even true for light sources or sound sources.

However, for various functions and actions graphical attributes do not suffice to describe the state of an object completely. So, all objects also have *semantic attributes*. Such attributes describe *capabilities* of objects, and there are

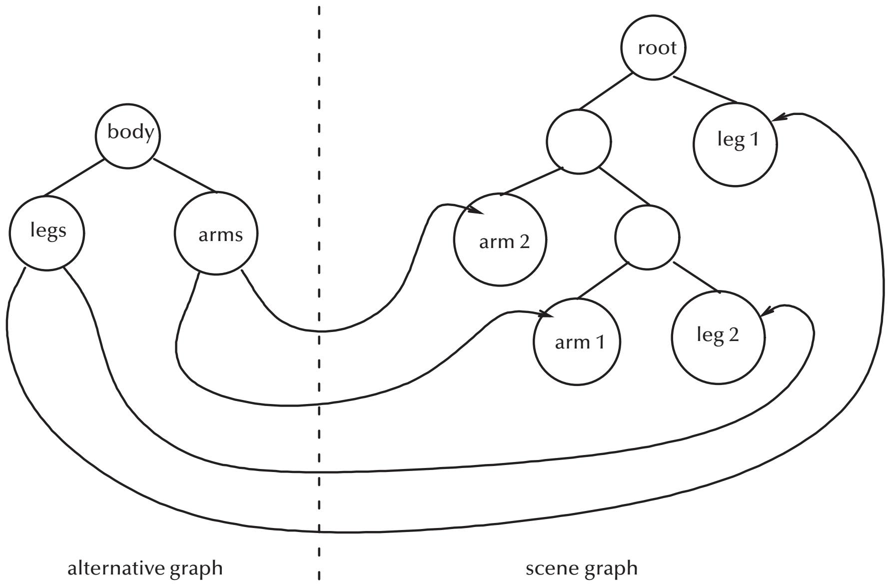


Figure 2.9: Hierarchical grouping (similar to drawing programs) establishes an alternative, user-defined scene graph.

attributes describing its *current state* (see table below). Usually, there is a state attribute for each capability attribute, but there are also state attributes per se, such as the lock status of graphical attributes.

Attribute	Meaning
grabbable	Can the object be grabbed by the user
grabbed	Is the object being grabbed currently?
movable	Can the object be moved by the user? For instance, through the `transform`-action.
selectable	Can the object be selected
selected	Is the object selected right now? (needed for `Info` action, for instance)
ghost is on	The object is colliding currently with some other part, and its ghost is being displayed additionally

Such semantic attributes can be implemented through membership in an object list (see below).

It has proven to be very convenient to be able to specify a *list* of objects instead of only one with any action which operates on objects. Such an object list can be subsumed under a new name, which provides alternate *groupings*. Object lists can be specified in the script file of the VE, and they can be changed at run-time through actions.

In addition, it is necessary to be able to specify *regular expressions* (wildcards) with object lists, in particular, if the exact names of objects are not known in advance. For example, if I want to define an action which deletes all objects whose name is `arrow...`, but those arrows will be created only at run-time.

Just like with drawing programs, assembly simulation users often would like to group parts and modify these as a whole (see Figure 2.9). For instance, such a group could be grabbed or translated only as a whole. Therefore, a mechanism for hierarchical grouping of objects must be provided. Grouping establishes an *alternative scene graph*. The leaves of such an alternative scene graph are nodes in the renderer's scene graph (not necessarily leaves in the renderer's scene graph). Inner nodes of the alternative scene graph do (usually) not correspond to inner nodes of the renderer's scene graph. An object of the scene graph can be part of a grouping. It can be part of at most one grouping. (Remember: It can belong to many object lists.) An alternative scene graph (grouping graph) is necessary for two reasons: first, in my experience it is always a bad idea to change the original scene graph as provided by the CAD system (information is lost, and other actions/modules might depend on the original hierarchy); second, with grouping graphs, we have the flexibility of maintaining several alternative scene graphs.

Actions operating on lists of objects must first perform *group closure* defined as follows. Assume an object O is member of an object list L. Let the object also be part of a user-defined grouping. Let the top-most grouping node of O be G (remember: it can be part of at most one group). The closure of L with respect to grouping results from L by replacing O by all leaf nodes of group G.

Due to the dynamic nature of the scene graph it is important that objects are referenced by *name* instead of pointer. Objects might cease to exist or the scene graph might be restructured (even by other modules of the application). If all object creations and deletions occur through an action, then it is easy to make the VR system cache object pointers. If other modules besides the interaction manager can create/destroy objects, then special mechanisms need to be implemented so that the VR system knows when its object pointer chances become invalid.

Virtual prototyping users frequently want to *exchange* geometry at run-time, i.e., from a semantic point of view, the object does not change but its spatial appearance. So, all its semantic attributes must be kept, but actions and modules (e.g., grabbing and collision detection module) dealing with its geometry must be notified, when this happens.

With polygonal rendering, the notion of *LODs* has been developed [Red96, Tur92, LT99], which can help increasing rendering speed while preserving the perceptual quality of rendered images. Basically, LODs are different graphical representations of the same geometric object. For the description of VEs, we need a similar concept: different *semantic* representations of the same object. Since most of these representations involve geometry, it makes sense to integrate them into the (graphical) scene graph. However, they will not be used for display but for other functionalities. So there might be a representation for collision detection (which does not need all the appearance attributes, and which might have a higher resolution for more precision), a representation for the hull (in order to implement safety distance checking), etc.

2.2.5 Grammar

The grammar of the scripting language is fault-tolerant and robust against variations and abbreviations (for instance, the user can write `playback`, `play-back`, `anim`, etc., for the keyframe animation action). Ordering of lines should (almost) never matter! (in the first implementation this was achieved by a multipass parser.)

For easy creation and maintenance of almost identical scripts, full C-like preprocessing is provided. This includes macros, conditional "compilation", and

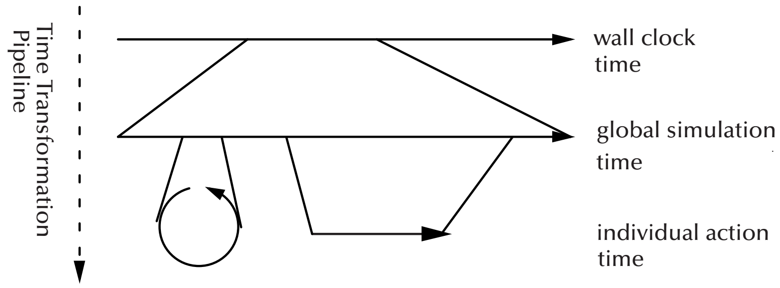


Figure 2.10: A simulation of virtual environments must maintain several time "variables". Any action can have its own time variable, which is derived from a global simulation time, which in turn is derived from wall-clock time. There is a small set of actions, which allow the simulation to set/change each time transformation individually.

including other files. The preprocessor's macro feature provides an easy way to build libraries with higher-level behavior.

2.2.6 Time

Many actions (besides navigation, simulation, and visualization) depend on time in some way. For example, an animation or sound sample is to be played back from simulation time t_1 through t_2, no matter how much computation has to be done or how fast rendering is.

Therefore, the module maintains a global *simulation time*, which is derived from wall-clock time. The transformation from wall-clock time to simulation time can be modified via actions (to go to slow-motion, for example, or to do a time "jump").

Furthermore, we need to maintain an unlimited number of time variables. The value of each time variable is derived from the global simulation time by an individual transformation which can be modified by actions as well (see Figure 2.10).

Those times can be used as inputs to events, or to drive simulations or animations. Thus, time can even be used to create completely "time-coded" parts of a virtual reality show.

In a way, the time transformation pipeline as proposed in Figure 2.10 resembles the (simplified) rendering transformation pipeline, except that it works the other way round and it lives in 1D: there is a wall-clock time (vs. device coordinates), which is transformed into the global time (vs. world coordinate system), which in turn is transformed into local "times" (vs. object coordinate systems).

2.2.7 Inputs and events

Anything that happens in our VEs does so, because there has been an input which made it happen. Inputs can be considered the interface to the real-world, although they will be used also to interface with other modules and even with the interaction manager itself. They can be considered the "sensory equipment" of actions and objects.

Events are the "nerves" between the user input and the actions, the "motors" which make things move. This analogy is not too far-fetched, since events can actually do a little processing of the input. The input to events has always one of two states: *on* or *off*. Similarly, actions can be only on or off.

Events have the form

[event-name :] trigger-behavior input parameters

where *event-name* is for further reference in the script. All components of an event are optional. When an event triggers it sends a certain message to the associated action(s), usually "switch on" or "off".

It is important to serve a broad variety of inputs (see below), but also to provide all possible trigger behaviors. A trigger behavior specifies when and how a change at the input side actually causes an action to be executed. Let us consider first the simple example of an animation and a keyboard button:

animation on as long as button is down,
animation switched on whenever button is pressed down,
animation switched on whenever button is released,
animation toggles on→off or off→on whenever button is pressed down.

These are just a few possibilities of input→action trigger-behavior. The complete syntax of trigger behaviors is

action
`on|off|switch_on|switch_off|toggle`
`while_active|while_inactive| when_activated|when_deactivated`
input

It would be possible to have the VE author "program" the trigger-behavior by using a (simple) finite state-machine (as in dVS, for instance). However, I feel that this would be too cumbersome, since those trigger behaviors are needed very frequently.

All actions can be triggered at start-up time of the VE script by just omitting the *event* and *input* parts. In order to be able to trigger actions in a well-defined order at start-up time, there is a special "input" named `initialize` with a parameter which specifies the ordering (integer or name of a phase).

In addition to the basic events, events can be combined by logical expressions, which yields a directed "event graph". This graph is not necessarily acyclic.

2.2.8 A collection of inputs

In this section I will briefly list some of the "sensory equipment" of my VE framework.

Physical inputs. These include

- all kinds of buttons (keyboard buttons, mouse, spacemouse, boom),
- flex and tracker values; since these are real-value devices, thresholds are used to convert them to binary values,
- gestures (see Section 4.2.1 for gesture recognition algorithms),
- postures; these are gestures plus orientation of the hand; examples are the "thumb-up" and "thumb-down" gesture (see Figure 4.2),

- voice input; this means keyword spotting, enhanced by a regular grammar, which, in addition, can tolerate a certain (user-specified) amount of "noise"; so far, this seems to be quite sufficient.

2D analogues. Just like for 2D GUIs, we need (sometimes) virtual buttons and menus. In VR, there are much more possibilities as to the interaction techniques (see Section 4.5.1). Suffice it to say here, that one possibility is that virtual buttons are 3D objects which are checked for collision with some pointing "device", usually the graphical object for the finger tip. Any object of the scene graph can be a virtual button.

Virtual menus are the 3D analogues of 2D menus as known from the desktop metaphor. Each menu item can be the input to one or more events. From the description file, it is quite easy to create the geometry and textures for the 3D menus (possibly, additional parameters could determine layout parameters). This can be done at load-time (as opposed to a preprocessing step as in [Jay98]). If menus are 2D overlays (see Section 4.5.1), then the generation from the file is even simpler.

Geometric inputs. These check some geometric condition. Among them are *portal*s and *collision*s.

A portal is an arbitrary object of the scene graph. The input checks whether a certain object is inside or outside a portal. By default, the center of the object's bounding box is considered (or any other point in local space). The object can be the viewpoint. Other definitions check whether a certain point has crossed a polygon (the portal); however, from a practical point of view, that is just a special case of our definition.

Portals can be very useful for switching on/off parts of the scene when the user enters/leaves "rooms" (visibility complexes or different virtual environments). This can help to increase rendering speed when otherwise the renderer would send geometry to the pipe which is within the viewing frustum but cannot be seen, because it is completely occluded.[6] Another application is chaining several actions in a way which does not depend on time (for instance, playing a sound when some object passes by).

A collision input is the "output" of the module for exact collision detection of objects [Zac95, Zac98b].

Monitors. The status of actions, objects, counters, and timers can be fed into the event network by *monitor inputs*.

Any action's status (*on* or *off*) can be fed back into the event network directly via monitor inputs. The value of counters can be compared to another counter or a constant. The result of the comparison is the input to events.

Attributes of graphical objects (boolean, integer, float, vector, or string valued) can be interrogated, so that actions can be triggered when they change, while we do not care which action (or other module) changed them. The general form of an object attribute input is

`attribute` *attribute-name* `object` *object-name comparison*

[6] This application of portals is sort of a "poor man's visibility complex". It asks the VE author to group geometry into visibility cells [TS91], and to tell the renderer when to display which cells.

Attributes are not only graphical attributes (transformation, material, visibility, etc.), but also semantic "interaction" attributes (see Section 2.2.4).

Object attributes might be set by the interaction manager itself (possibly by many different actions), or by *other* modules, so object attribute inputs can provide a kind of simple access control mechanism in some cases.

All time variables (see above) can be compared to an interval or an instant in time by one of the usual comparisons. The timer input monitors the value of a timer whether it is within the interval, or whether it has passed the instant.

2.2.9 Actions

Actions bring a virtual environment to "life". Anything that happens, as well as any object properties are specified through actions. They could be thought of "motors" executing what the network of events and inputs tell them to do.

Actions come in 4 categories:

1. plug-in actions
2. references to an existing action
3. python functions
4. a sequence of actions

There are no "built-in" actions — or rather, there is no difference between plug-in actions and built-in actions (see Section 2.4).

Actions are usually of the form

action-name : *function object-list parameters options*

All actions should be made as general as possible, so it should always be possible to specify a *list of objects* (instead of only one). Furthermore, objects can have any type, whenever sensible (e.g., assembly, geometry, light, or viewpoint node). The *action-name* is for later reference in the script.

My experience shows that it is necessary to be able to activate and deactivate actions. This is necessary when the author of a VE wants the user to perform actions in a certain chronological order. For instance, in a maintenance scenario a user should be prevented from doing something before he has done something else first.

An action is deactivated when it does not respond to messages being sent by events. This can be done by a certain action, which (de-)activates other actions. Alternatively, it is quite convenient to be able to bracket the actions you want to (de-)activate:

```
activate event
    actions
    ...
endactivate
```

Consistency. This is certainly an issue in any VR system being used for real-world applications. Here, I will not discuss the problems arising in multi-user VEs, or in systems with concurrent modules. The problem of consistency exists even within our interaction module. Some of the actions described below set transformations of objects. Obviously, those will interfere when they act at the same time on the same object. For example, in virtual prototyping applications, we could grab an object with our left hand while we stretch and shrink it with the other hand. The problem arises also, when an action takes over, e.g., we scale an object after we have grabbed and moved it.

Flushing transformations or squeezing them into one matrix is not a very good idea, since we loose the original object and we have no control over the order of transformations. I do not want to loose the transformations of earlier actions, because this is valuable information we might need in a further step "outside" the VR system.

One way to deal with that is to impose a strict sequence of transformations to be used per object, at least for objects under the control of this module. I have chosen the sequence: $\textit{scaling} \times \textit{rotation} \times \textit{translation}$. Sometimes, it is necessary or convenient for the implementation of actions to add more transformations. In that case, those actions need to re-establish that sequence after they are finished (this can be done, for instance, by squeezing all transformations into one matrix, and decomposing it again into 3 transformations [Tho91]).

Another consistency issue arises when we use levels-of-detail (LODs) in the scene graph. Since any object can be a LOD node or a level of an LOD, any action should transparently apply changes to all levels, so that the author of the virtual world does not have to bother or know whether or not an object name denotes a LOD node (or one of its children).

2.2.10 A Collection of Actions

The following will briefly describe a list of action which have turned out to be useful in general throughout my work on several virtual prototyping projects.

Navigation. Navigation is probably the most basic interaction with a VE (see Section 4.4). It is triggered by an event. With the point-and-fly paradigm, this is usually a gesture or a spoken command. Other parameters for navigation, such as direction and speed, are obtained directly from the logical input devices by the action.

The geometry of the VE. can be changed by actions for loading, saving, deleting, copying, creating, and attaching objects or subtrees.

Several actions can change object attributes like visibility, wireframe, and transformations. Others change material attributes, such as color, transparency, or texture. Transformation actions can be used to position an object at a certain place or to move objects incrementally.

Objects (and subtrees) can be scaled interactively with the "stretch" action. When it is invoked, handles will be displayed at the corners and faces of the bounding box of the object. These can then be grabbed and will scale the object(s) as they are moved. This action is quite useful to select a certain volume of the world or to create place holders from generic objects like spheres, cylinders, etc.

Grabbing. This is probably the second most important action, at least in virtual prototyping. Grabbing could be solved by using an attach action. However, in my experience, there are many situations where grabbing needs to be parameterized by a lot of options.

The event which triggers grabbing could be any of the above. Conventionally, it is a gesture (fist). However, two-point grabbing (see Section 4.5.3) can be realized as well by a boolean combination of the collision of the finger tips

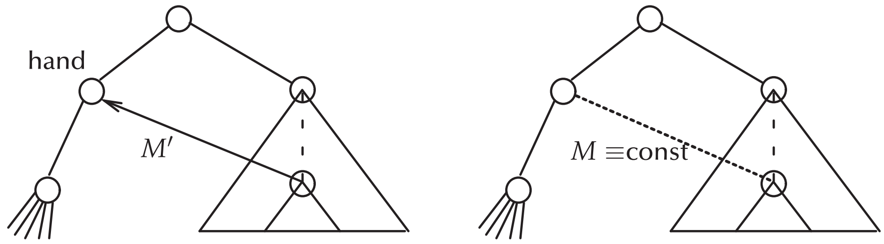


Figure 2.11: Grabbing objects can be implemented by re-linking objects in the tree, or by maintaining a transformational invariant. Either approach has its advantages and disadvantages.

with the object (e.g., thumb and one of the other fingers must collide with the object).

The grab action first makes an object "grabbable", i.e., it ensures that all objects involved are registered with the collision detection module. Then, as soon as the hand touches it, it will be attached to the hand. There are two ways this can be done (see Figure 2.11): either by linking the object to the father of the hand in the scene graph, or by maintaining a transformation invariant [RH92].

Of course, this action allows grabbing a list of sub-trees of the scene graph (e.g., move a table when you grab its leg). Options are: move other objects along with the grabbed one; move the parent in the scene graph, or the grandparent, etc.

As with navigation, there are several constraints and improvements that should be made in order to simulate and render this every-day interaction more naturally (see Section 4.5.3).

Animations. These can add a great deal of "life" to a virtual world. Almost any attribute of graphical objects can be animated. Animation actions include playback of transformations, visibility, transparency (for fading), and color from a file. The file format is flexible so that several objects and/or several attributes can be animated simultaneously.

Animations can be *absolute* or *relative* which just *adds* to the current attribute(s). This allows, for example, simple autonomous object locomotion which is independent of the current position. Of course, continuous interpolation between key-frames is possible. Animations can be time-coded or frame-coded.

Physical concepts. Physically-based simulations can increase "believability" of VEs tremendously. One of the most basic physical concepts is gravity. It has been implemented in an action "gravity" (actually, it should be called a property), which makes objects fall in a certain direction and bounce off "floor objects", which can be specified separately for each falling object.

More physical behavior would be nice to have, such as complete kinematics of rigid bodies including resting contacts and friction.

Constrained movement. Occasionally we want to constrain the movement of an object. It is important to be able to switch constraints on and off at any time, which can be done by a class of `constraint` actions.

Several constraints on transformations of objects, including the viewpoint, have proven useful:

1. Constrain the translation in several ways:
   (a) fix one or more coordinates to a pre-defined or to their current value,
   (b) keep the distance to other objects (e.g., ground) to a pre-defined or to the current value. The distance is evaluated in a direction, which can be specified.

   This can be used to fix the user to eye level, for terrain following, or to make the user ride another object (an elevator, for example).

2. Constrain the orientation to a certain axis and possibly the rotation angle to a certain range. This can be used to create doors and car hoods.

All constraints can be expressed either in world or in local coordinates. Also, all constraints can be imposed as an *interval* (a door can rotate about its hinge only within a certain interval). Interaction with those objects can be made more convenient, if the deltas of the constrained variable(s) are restricted to only increasing or decreasing values (e.g., the car hood can only be opened but not closed).

Of course, the constraints listed above are just very simple ones; for more complicated "mechanisms", a general inverse kinematics approach will be needed, like [WFB87] or [ZB94, EK89].

Object selection. There must be two possibilities for specifying lists of objects: *hard-wired* and *user-selected*.

In entertainment applications, you probably want to specify by name the objects on which an action operates. The advantage here is that the process of interacting with the world is "single-pass". The downside is inflexibility, and the writing of the interaction script might be more cumbersome.

Alternatively, we can specify that an action operates on the currently selected list of objects. This is more flexible, but the actual interaction with the world consists of two passes: first, the user has to select some objects, then specify the operation.

The list of selected objects can be communicated suitably and explicitely to other actions via object lists (see Section 2.2.4): The selection action just writes the objects into a certain object list, which other actions can read from. Using object lists, it is possible to have several selection actions write to different object lists for different purposes.

Virtual prototyping. There are several features, which have proven to be generally useful for VP applications. Since they are more tuned toward a special class of VR applications (albeit one of the most important ones today), I will discuss those in more detail in Section 5.1.

Finite state machines. The system can maintain an arbitrary number of *counters*. Those counters can be set, incremented, or decremented via certain actions. They can be used as input to events (which in turn trigger other actions).

Counters are very useful to switch from one "stage" of a "story-based" VE to the next one by the same gesture or voice command, or they can be used to build more complicated automata (a traffic light, for example).

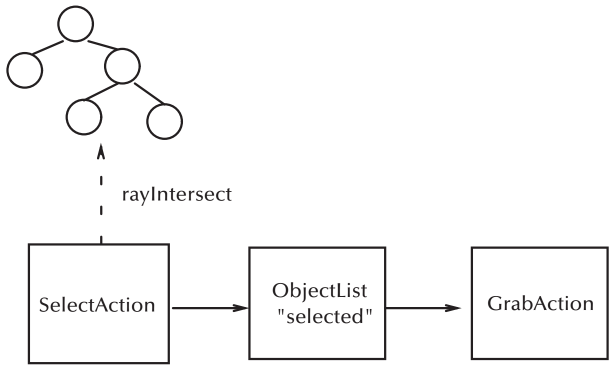


Figure 2.12: The concept of object lists is a convenient and suitable mechanism to communicate the list of selected objects to other actions, to all other actions or only to distinguished ones.

User modules. From my experience, most applications will need some special features which will be unnecessary in other applications. In order to incorporate these smoothly, our VR system offers "callback" actions. They can be called right after the system is initialized, or once per frame (the "loop" function), or triggered by an event. The return code of these callbacks can be fed into other events, so user-provided actions can trigger other actions.

These user-provided modules are linked dynamically at run-time, which significantly reduces turn-around time.

It is understood that all functionality of the interaction module as well as all data structures must be accessible to such a module via a simple, yet complete API (see Section 2.4).

User modules might also be implemented as separate programs. Or some of the programs existing under Unix might fulfil a task (e.g., taking a snapshot). So, an interfacing action to Unix commands is provided.

Miscellaneous. As described above, there are actions to set or change the time transformation for the time variables.

In addition to the actions already described, which allow giving various kinds of visual feedback to the user, there is also an action to give audio feedback.

2.3 Examples

In this section, I will give a few examples how the description of VE looks like. Some of the examples are drawn directly from actual projects.

The following example shows how the point-and-fly navigation mode can be specified.

```
cart pointfly dir fastrak 1 \
    speed joint thumbouter \
    trigger gesture pointfly
cartrev gesture pointflyback
cart speed range 0 0.8
glove fastrak 1
```

The hood of a car can be modeled by the following lines. This hood can be opened by just pushing it with the index finger.

```
constraint rot Hood neg \
   track IndexFinger3 \
   axis a b to c d \
   low -45 high 0 \
   on when active collision Finger13 Hood
```

Menu setup consists of two parts: the interaction with the menu itself,

```
menu popup MyMenu options speech "menu"
menu acknowledge MyMenu joint thumbouter
```

and the specification of actions triggered by menu selections

```
action menu button MyMenu_1
```

In order to provide acoustic feedback when action A is switched on, we can write

```
sound sound-file switch on when activated action A
```

Finally, I will present an example of a library "function" to make clocks. (This assumes that the hands of the clock turn in the local xz-plane.)

```
define CLOCK( LHAND, BHAND )
timer LHAND cycle 60
timer speed LHAND 1
// rotate little hand every minute by 6 deg in local space
objattr LHAND rot add local 6 (0 1 0) time LHAND 60
// rotate big hand every minute by 0.5 degrees in local space
objattr BHAND rot add local 0.5 (0 1 0) time LHAND 60
// define start/stop actions
Stop##LHAND : timer speed LHAND 0
Start##LHAND : timer speed LHAND 1
```

The ## is a concatenation feature of acpp. By applying the definition CLOCK to a suitable object, we make it behave as a clock. Also, we can start or stop that clock by the actions

```
CLOCK( LittleHand, BigHand )
action "StartLittleHand" when activated speech "clock on"
action "StopLittleHand" when activated speech "clock off"
```

2.4 Implementation

Since we are dealing with graphical objects augmented by semantical attributes, the system must maintain *semantic objects*. Inheritance does not seem to be flexible enough for the future, because there might be many other modules wishing to augment the scene graph by their own attributes. So I have chosen to encapsulate semantic attributes in the class of WalkObj, which also references scene graph objects.

Each module can add different representations to a group node in the scene graph. The object handler provides methods to add and retrieve these representations transparently. However, a WalkObj always references the group

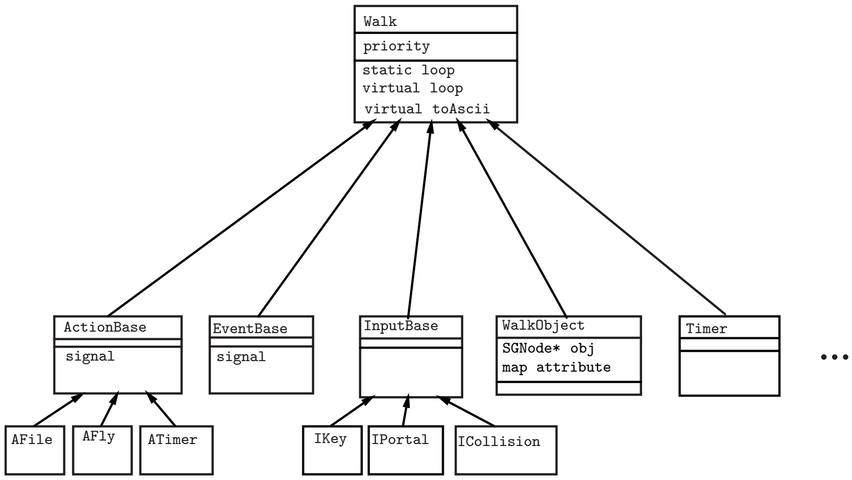


Figure 2.13: The design of the simulation module of the VR system.

node and never one of its representations (children). Likewise, it never references a certain LOD node in the scene graph, but always the LOD group node.

All events, actions, and semantic objects must have a unique name, even if the script does not specify one. This is necessary for distributed VEs (multi-user environments), so that remote systems can communicate the event triggers to each other.

During parsing of a VE description file, the left hand side of a statement is turned into an action object (either by creating a new one, or by retrieving an existing one) and a message, which will be sent to that object by the right hand side (an event). Usually, the message is the standard message "on/off", sometimes it is an action specific message, e.g. "add/delete" for the selection action. The left hand side might look like a new action although it refers to an existing action, e.g. `selection add/delete`.

Figure 2.13 shows the most important classes of the VR system in UML notation. Figure 2.14 shows the basic sequence diagram.

All actions, events, and input classes are implemented as individual DSOs. That way, there is no difference between "built-in" actions/inputs and application-specific ones. All DSOs are loaded on demand only. This makes the treatment of actions/inputs uniform. The greatest advantage probably is during development. The system can replace all instances of an action/input by a new implementation *at run-time*: it destroys all instances of a certain class (= DSO), closes the DSO, then waits for the user's ok, and finally loads the new DSO and instantiates all former instances with the new class.

In order to achieve this dynamic loading/instantiating, classes must register themselves with a factory, and instances must be created by that factory. Registration must be done at load-time of the DSO through class variables, which are constructed at load-time.

It is easy to change parameters of actions/inputs/events at run-time by a GUI. In order to facilitate a standardized and simple GUI implementation, each class provides a description of its parameters and their types. That way, the GUI can create a standardized form at run-time when the user wants to change a parameter of some action. After that, the GUI has to call the `paramChanged` method of the action, of course.

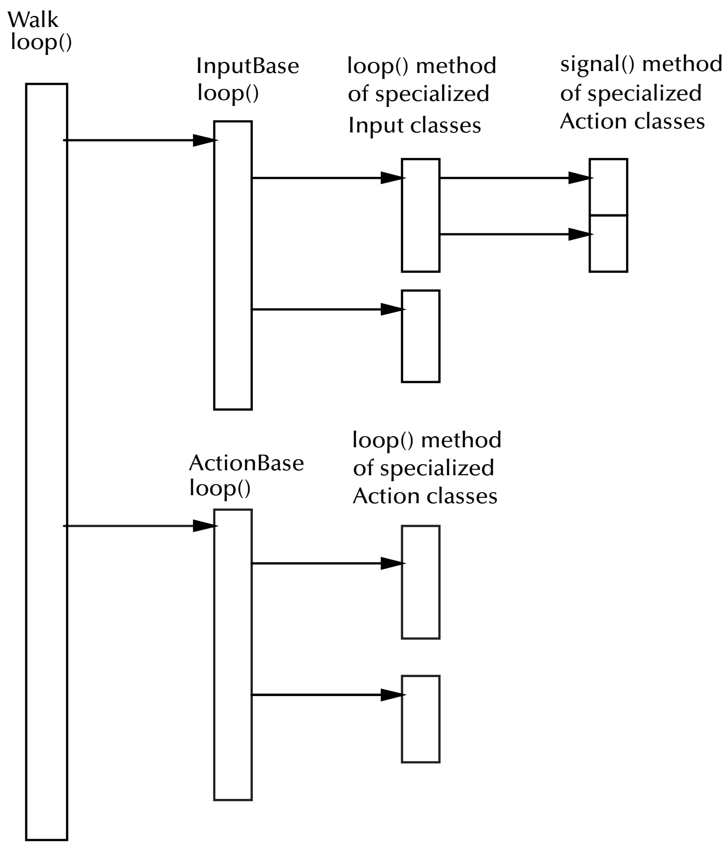


Figure 2.14: Basic sequence diagram of the interaction manager.

The parser can be written in a similar standardized way. Each class provides the parser with a map of keywords, types, and options, which tells the parser how to parse an action/input/event. The parser can then create the instance through the node factory. So, the burden of parsing is "out-sourced" from the classes to the parser, although the parser has no *a priori* knowledge about what attributes/parameters each class can have. With this mechanism, there is no difference between "built-in" classes and application-specific ones. So it is possible to use application-specific classes in the VE script file, even though the parser does not know about them (because, basically it does not know about *any* classes).

2.4.1 Distributing the system

The architecture described above is suitable for multi-user VR (distributed VR). I am not concerned here with the issues of latency, large-scale distribution (hundreds or thousends of participants), network topology, or floor control.

When a new user registers, she has to receive a complete copy of the current scene graph and the behavior graph (actions, inputs, and events). Then she will make her own additions to that scene graph and the behavior graph (for instance, the avatar representing herself). These additions will then be transmitted to all other participants.

In a distributed VR system, input objects come as one of two variants: the one assumed and discussed so far, or they are just a *proxy*. In non-distributed

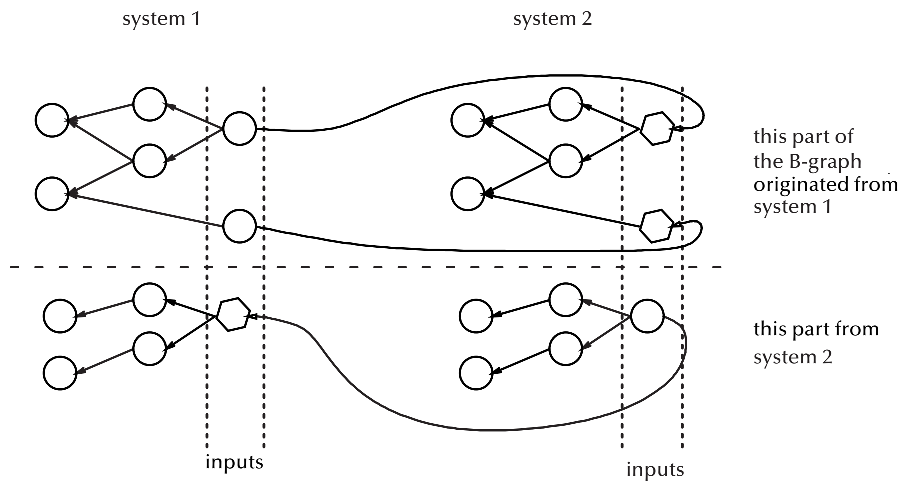


Figure 2.15: Behavior graphs can be distributed easily by replicating all actions/events/inputs and the scene graph. Proxy inputs provide a transparent mechanism to broadcast messages to all VR systems. In the figure, the octagonal input nodes are proxy inputs which do not compute anything themselves but replicate the message computed on the remote VR system and feed it into the local system.

VR, all inputs are "real" inputs. In distributed VR, all inputs created by the local system are "real" inputs, too. All input objects which have been received as part of the behavior graph transmitted from other systems are instanced as mere proxy inputs. A proxy input is one which does not really generate messages by itself. Instead, when the real input (located on a remote VR system) generates a message, this message will be broadcast to all other VR systems. They will take this message and "pretend" it has been generated by the respective local systems. Thus, distribution is transparent to the rest of the system (see Figure 2.15).

The main loop needs to be augmented a little bit to provide for multi-user VR:

```
main loop for distributed VR

loop:
   clear global list new_messages
   for each local input:
      compute message (if any)
      and store it in a global list new_messages
   broadcast all messages in new_messages
      to all other VR systems
   receive new messages from other VR systems
      and add them to new_messages   {for the proxy inputs }
   for each message ∈ new_messages:
      call signal() of filter or action
```

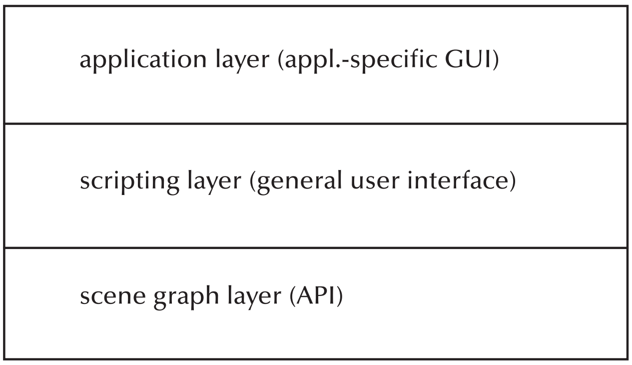


Figure 2.16: I have identified three layers of increasing abstraction and specialization, on which authoring of a VE takes place.

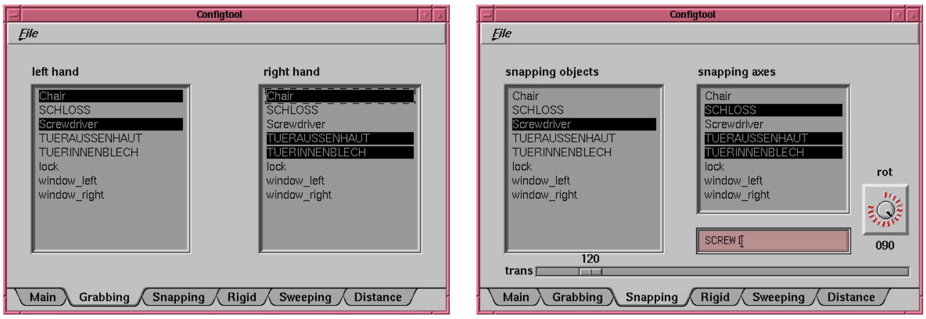


Figure 2.17: For each application domain a specialized VE application builder is needed, I believe, which provides the high-level functionality specific to that area. These high-level functions can be mapped on the script level.

2.4.2 The three layers of authoring

In order to make VR an efficient tool to save time, it must be easy to "build" a virtual environment which represents, for instance, part of a car and simulates part of its physical behavior (see also Section 5.1). It must be at least as easy as designing with a CAD system.

I have developed a three-layer framework of authoring such VEs; each layer provides a certain level of abstraction and specialization. It has proven to be flexible and powerful (see Figure 2.16).

The bottom layer is the *scene graph*: it deals mostly with geometry and rendering optimizations. Some scene graph APIs, such as VRML2.0 or Inventor, also provide very low-level scripting features (*routes*, *engines*).

The next level is the AEO scripting framework (see above). The functionality provides general building blocks to construct VEs, each of which with higher-level, yet general "story-board driven" functionality.

End-users working in a certain application domain (such as assembly simulation) will specify scenarios at the *application layer*, which provides a graphical user-interface (see Figure 2.17) and specialized, very high-level functionality (e.g., the user tells the system which objects are tools).

Scenario templates

If parts had standard names, then a large portion of VEs could be derived from standard "scenario templates" specific to the application domain, e.g., "front door", "tail light", "gear box", etc., or "office building", "shopping mall", "city", etc. So, for a VR session with a different geometry, a VE author would only have to modify one of those templates.

However, it is not clear to us yet, whether designers will ever design all the VR-relevant attributes. Some of them are geometric, like visible material, thickness of metal sheets, and the like. So far, a lot of authoring time is spent basically on specifying the non-geometric (semantic) attributes of parts, such as the function of objects (screw, tool, etc.), non-geometric materials (flexibility, smoothness), the order of tasks in the (dis-)assembly process, etc.

Chapter 3

Collision Detection

And when you have reached the mountain top,
then you shall begin to climb.

KAHLIL GIBRAN

One of the main goals of using a VR system for assembly simulation is the potentially high degree of "reality" which can be experienced when immersed in a VE. In order to achieve this, the VR system needs to be able to simulate realistic and natural object behavior at interactive frame rates.

Other tasks of a VR system in the context of virtual prototyping are geometrical and spatial analyses. In a fitting simulation a designer might want to check interactively, if a slightly larger, different part would fit in the place of the original part. Or he might want to scale or shift a part while the system checks all relevant safety distances. In order to check serviceability of a part, the VR system has to track the work space necessary for the disassembly path and for the tools, and it has to report both intentional and "forbidden" collisions. During an assembly or disassembly simulation it is often necessary to simulate kinematics in order to perform a sensible design study.

Real-time collision detection of polygonal objects undergoing rigid motion is of critical importance in many interactive virtual environments. In particular, simulation algorithms, utilized in virtual reality systems to enhance object behavior and properties, often need to perform several collision queries per frame. It is a fundamental problem of dynamic simulation of rigid bodies and simulation of natural interaction with objects. For example, in virtual prototyping, parts should be rigid and slide along each other. Especially for haptic rendering of force-feedback, very fast collision detection is needed, as the haptic rendering loop must run at 500 Hz, at least.

It is very important for a VR system to be able to do all simulations at interactive frame rates. Otherwise, the feeling of immersion, the "believability" of the virtual environment, and even the usability of the VR system will be impaired severely.

3.1 The setting

3.1.1 The simulation loop

With collisions there are two tasks to be handled: *collision detection* and *collision handling*. The former is the general problem of determining whether or not objects penetrate (i.e., "something happened"), while the latter is the problem of determining appropriate steps based on the current collision status, which is usually handled by a simulation module. Although both parts pose interesting problems, I will focus only on the collision detection part in this chapter. For further reading on collision response see [Bar94, MW88, BV91, Hah88a].

In general, collision detection is integrated into simulation loops almost always in the following way:

```
collision detection within a simulation loop

loop forever
   move objects by simulation
   check collisions
   take appropriate actions based on collision reports
   (e.g., highlight objects, do back-tracking,
    or calculate forces, ...)
```

3.1.2 Requirements and characterization

The requirements for collision detection algorithms can be very demanding. Especially for virtual prototyping, since CAD data has two characteristics: the geometry is very large in terms of polygon counts (10,000–100,000 polygons per object), because the tessellation error should not be larger than 0.5 mm; secondly, the geometry is not well-formed, i.e., there are gaps between polygons, polygons could be duplicate or degenerate, and the geometry is not 2-manifold in general — hence such geometry is named *polygon soups*.

In spite of the geometrical difficulties, collision detection must be fast enough under all circumstances, so that real-time simulations can perform several iterations per frame. For instance, a rigid body simulation could need 10–20 iterations per loop (see Section 4.5.4). And force-feedback needs at least 500 collision queries per second in order to be able to render "aliasing-free" forces.

Furthermore, the algorithm cannot make any assumptions or estimations about the position of moving objects in the future when they are being moved by the user (noise, jitter).

The following is a wish list of features which the ultimate algorithm would satisfy completely:

- Given a certain form of representation, it should be able to handle the *largest possible class* of objects (e.g., deformable "polygon soups" in the polygonal B-rep). An important subset is the set of convex polytopes. Other classes can be found in [Tou88].
- It should be *fast* (goal: $2 \times n \times 100,000$ polygons in $\leq$ 1 millisecond checked).
- In case of a collision it should return a *witness*, i.e., a pair of intersecting polygons plus a point of intersection; alternatively, it should be able to return *all* pairs of intersecting polygons.
- Given the current and previous position, it should be able to compute the *exact* time and position of collision. Exact time and position is needed by simulation algorithms, in particular rigid body simulations (see Section 4.5.4).
- It should not need very large *auxiliary data structures*. In particular, construction of these data structures should not take too long ($\leq$ 1 second per object). Otherwise, those data structures would have to be constructed in a preprocess and saved with the original geometry. This can be a problem for the acceptance of VR in large IT infrastructures of companies.

As of today, no algorithm is known which meets *all* requirements. Therefore, algorithms expose some characteristic features or restrictions:

- *Exact* collision detection is generally harder than *approximate* collision detection. The latter is usually biased, i.e., the algorithm tends to favor one answer over the other (see Section 3.4.3). The bias is caused by using some sort of simplification or a probabilistic algorithm. For instance, for collision avoidance an approximate algorithm with a bias for "intersection" can be sufficient [CAS92].

- *4-dimensional* approaches take time into account [Hub93, Can86, ES99]. The time dimension can be used to compute the exact time and position of a collision.

  "Timeless" approaches consider all objects only at a certain time (they still keep in mind that objects probably move — unlike approaches in computational geometry). If timeless approaches have to provide the time of collision more accurately, they will resort to some kind of back-tracking method.

  This class of collision detection algorithms is sometimes called *dynamic*, while the "timeless" one is called *static*.

- *Incremental.* Some algorithms try to exploit temporal coherency in order to speed up the collision detection procedure. This can increase memory costs significantly, since they need additional data structures. Furthermore, they need to store results from earlier collision detection queries. (see Sections 3.4.3 and 3.4 for example).

  These algorithms are also called *dynamic*, while non-incremental algorithms are called *static*.

- *Hierarchical* algorithms exploit spatial coherence by divide-and-conquer techniques.

  Hierarchies can be built above object level or on polygonal level, by space subdivision (see Section 3.8), or by plane sweep.

- *Restricting the domain* of input objects can sometimes allow for more efficient algorithms. Examples are the restriction to rigid bodies and/or convex polytopes.

- Many applications can do with *off-line* (i.e., not real-time) collision detection, because the application is not driven by real-time input like in VR environments, for example, path planning in robotics or physically-based simulation for animation.

3.1.3 Object Representations

The internal representation of graphical objects has great impact on the choice of algorithms, not only for collision detection, but also for rendering, modeling, and many other parts of an interactive graphical system.

Several different approaches to object representation have been developed. They can be distinguished into *boundary-based* versus *volume-based*. Boundary-based object representations are the classical polygonal b-rep, parametric surfaces (B-Splines, NURBS, etc.), or "augmented" octrees [CCV85, NAB86, FK85]. Some volume-based object representations are the well-known octree [YKFT84, TS84, FA85, NAB86, Dye82, TKM84], BSP [PY90, TN87b, NAT90b, Tor90, Van91], and CSG. Not so well-known representations are sphere splines

[MT], which approximate an object by moving a sphere along some spline curve while varying its radius; ray-reps [MMZ94], which represent an object by a collection of parallel line segments of varying length and position; and the H-P model, which approximates an object by slices of a sphere [CM87].

All of these object representations have been devised to suit special needs; some of them are still in use today. BSPs have been devised for hidden-surface removal without z-buffer hardware. It is also fairly straight-forward to compute boolean operations on them. Parametric surfaces are used in CAD systems to model curved surfaces with higher continuity. Octrees are also used in solid modelers and for representing volume data.

Octrees and BSP trees are well suited for intersection computations of polyhedral objects; however, we are not really interested in the construction of the intersection of two polyhedra. On the other hand, octrees cannot provide for exact collision detection algorithms (unless augmented octrees are used), and a huge amount of memory is needed to achieve a fairly reasonable accuracy. Finally, both will probably never be the native representation for VR applications (although octrees sometimes *are* being employed for scientific visualization).

The advantage of b-reps is that they can easily hold topological information about the geometry, such as adjacency and incidence of features (i.e., vertices, edges, polygons). Octrees are better suited for computing mechanical properties like mass, volume, inertia tensors, etc. [LR82, TKM84].

Sphere splines and sphere coverings (the discrete variant of sphere splines [OB79]) are also only approximate representations, thus they cannot provide for exact collision detection algorithms.

Research has been done on real-time rendering of NURBS [AES94, Pet94, KML95]. I believe that, eventually, polygonal rendering on high-end graphics workstations will be supplemented (if not replaced) by NURBS. However, NURBS rendering will not become an option before several years.

3.1.4 Definitions

Depending on the class of objects being checked, there are different possible definitions of the term *collision*. Let P, Q be two objects:

1. arbitrary objects: $\exists\, e \in E\, \exists\, p \in F : \; e \cap p = x \in \mathbb{R}^3$

2. closed objects:

$$\exists x \in \mathbb{R}^3 : \; x \in P \wedge x \in Q$$

   or

$$d(P, Q) := min\{\, |p - q| \; : \; p \in P, q \in Q \,\} > 0$$

   i.e., their distance does not vanish.

3. convex objects:

$$\forall\, e \in E : \; e \cap P \neq \varnothing$$

   or

$$\exists\, \text{plane}\, w \in \mathbb{R}^4 : \quad \forall\, v \in P : \; v \text{ is left of} w \;\wedge\; \forall\, v \in Q : \; v \text{ is right of}\, w$$

   (other definitions are possible)

Sometimes, the term *collision detection* is understood as the problem of *constructing* the intersection $P \cap Q$. Of course, the *detection problem* can be reduced to the construction problem. However, since the latter is generally not needed in VR applications, and since it is by orders of magnitude harder, I will consider only the *detection problem*.

3.2 The basic operation

With almost all collision detection algorithms the basic operation is polygon/-polygon intersection, or edge/polygon intersection. Basically, all collision detection algorithms are concerned with trying to avoid as much polygon/polygon tests as possible.

For the sake of completeness, I will quickly review some of the polygon-polygon intersection algorithms. Most of the algorithms gain their efficiency by *reduction of dimensionality*.

In [Zac94b] I have presented an efficient implementation of edge/polygon intersection tests. Only one division is involved; before executing that division there are many pre-checks based on sign comparisons and other tests, which can reject many non-intersecting edge/polygon pairs trivially. Should the actual intersection be necessary the point/in-polygon test is reduced to 2D.

Polygon/polygon intersection tests can be reduced trivially to edge/polygon intersection tests by testing each edge of one of the polygons against the other polygon, and vice versa.

Other efficient polygon/polygon intersection tests have been presented by [Möl97, Hel97]. The idea of the former is to compute the intersection of the supporting planes, which is a line. Then, each polygon is intersected with that line, which yields two intervals. The polygons can intersect only, if the intervals on the line intersect. The idea of the latter is to compute the intersection of one of the polygons with the supporting plane of the other one, which yields a line segment. Then, the line segment is tested for intersection with that polygon in 2D.

The method presented in [Möl97] works only for triangles, as does the method in [Hel97], although the latter could be extended to work for arbitrary convex polygons. My method works for arbitrary, not necessarily convex, polygons.

I have carried out extensive timing comparisons between [Möl97] and my method. It turns out that for triangles the algorithm [Möl97] is faster. For 5-gons and higher, my algorithm is faster than a triangulation and repeated triangle intersection. The break-even point are quadrangles. Table 3.1 substantiates that.

The column "Möller" denotes the algorithm of [Möl97]. When Möller is applied to quadrangles, each of them must be split into 2 triangles and then up to 4 pairs of triangles are tested. Splitting a quadrangle involves finding a concave vertex if the quadrangle is non-convex. This was done because I combined both algorithms mine and "Möller", and depending on the case one of the two are called. So, in order to determine the break-even point, all preliminary tests had to be taken into account.

Although it is desirable to have a polygon intersection test at hand which is as fast as possible, the importance lessens significantly as the number of polygon/polygon tests is reduced by more sophisticated algorithms. The algorithm in the next section (3.3) still needs to do a lot of those tests. However, algorithms

# triangles	Mix	Möller	Zach
2 triangles	non-intersecting	2.3	5.7
	intersecting	2.9	3.4
	mix, 3.5% intersect	2.3	5.6
2 convex quadrangles	non-intersecting	7.6	7.8
	intersecting	6.7	5.4
	mix, 42% intersect	7.3	6.8
2 concave quadrangles	non-intersecting	7.4	7.5
	intersecting	7.7	5.0
	mix, 5% intersect	7.5	7.4
2 concave 5-gons	non-intersecting	11.3	11.0
	intersecting	7.9	7.6
	mix, 29% intersect	10.4	10.0

Table 3.1: Comparison between my polygon-polygon intersection algorithm and the one presented by Möller [Möl97]. 1 million random polygons were tested. The figures are in microseconds for testing one pair; they were obtained on a 194 MHz R10000.

such as the hierarchical ones do much less of them. Of course, the right balance has to be found between the effort of avoiding polygon/polygon tests and the costs involved.

3.3 Bounding-box pipelining

The brute-force algorithm is "so trivial that it is not even worth a literature reference" [MP78]. It goes as follows. Check every edge of polyhedron P if it intersects any of the polygons of polyhedron Q, and vice versa. (It is not sufficient to check only the edges of P against polygons of Q; and, it is not sufficient to check whether there are some vertices inside the other polyhedron, provided they are closed). Alternatively, one can check polygons against polygons.

From this very simple algorithm, it is obvious that edge/polygon or polygon/polygon intersection tests are the basic operation of (almost) all collision detection algorithms. Actually, most algorithms use polygon-polygon intersection tests as fundamental operation, which is more convenient from a practical point of view. In the following, I will do so, too.[1]

By adding several pre-checks in a pipelined way such that the computationally cheaper tests come first, we can achieve a considerable improvement.

Trivially, if a polygon $f \in F^P$ is not (partially) in the bounding volume of Q then f needs not be considered any further. Similarly, we observe, that a polygon $f \in F^P$ can intersect a polygon $g \in F^Q$ only if g is (partially) in the bounding volume of P. So, before we check polygons of P against polygons of Q, we collect, in a pre-phase, all polygons of Q which are in the bounding

[1] If we assume that a polygon/polygon intersection test is computationally as expensive as 2×4 edge/poygon intersection tests, then it might seem inefficient to do polygon/polygon tests instead of edge/polygon tests, because the brute-force algorithm basically tests each edge twice. However, with optimized algorithms, polygon/polygon tests are performed very rarely. In addition, an object has about $2|F|$ many edges (assuming quadrangles). So, any algorithm which basically operates on bounding volumes of polygons and edges needs to process about twice as many edge bounding boxes than face bounding boxes.

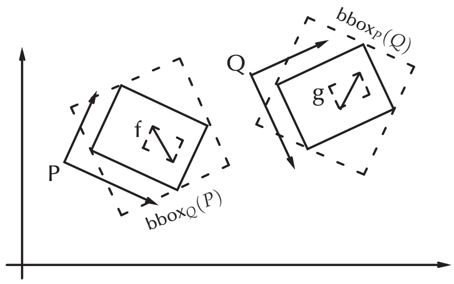


Figure 3.1: Doing bounding box checks in the appropriate coordinate system is an application of the principle of problem transformation.

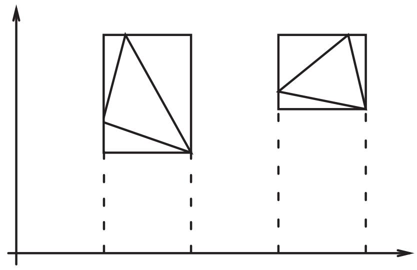


Figure 3.2: Lazy evaluation of bounding boxes saves a lot of computations, too.

volume of P; these will be denoted by the set $\bar{F}^Q$. Then, polygons of P are checked only against polygons of $\bar{F}^Q$.

The pre-checks explained so far are bounding volume checks. Since such pre-checks must be very efficient in order to gain anything, I have chosen bounding boxes; other bounding volumes might be suitable as well.

It is important that the bounding box checks are done in the most efficient coordinate system, i.e., by choosing the coordinate system carefully, we can save a lot of transformations. So, the collect-phase for $g \in F^Q$ should be done in Q's coordinate frame, while bounding box tests for $f \in F^P$ should be done in P's coordinate frame (see Figure 3.1). That way, no vertex transformations are needed to compute bounding boxes of polygons.

A second technique is *lazy evaluation* of bounding boxes. At some point in the bounding box "pipeline", it will be necessary to actually compute the bounding box of polygons (unless the object is stationary, this has to be done every frame). The naive approach would transform the vertices of a polygon into world space, then compute the bounding box, and finally test the bounding boxes. However, it is more efficient to first compute only the x-coordinates of the vertices involved, then the x-slab of the bounding boxes, and then compare those (see Figure 3.2). Only if they overlap, the y-slabs and z-slabs should be computed.

At first glance, it might seem inefficient to compute face bounding boxes from scratch every time the object has moved. One might think that it would be much faster to store a face's bounding box (in object coordinates) and transform it later on. However, in (almost) all practical cases, polygons have only 3–5 vertices, and a bounding box can be found by merely comparing points, while transforming a box costs at least 36 FLOPS (see [Gla90] and [Zac94b, p. 84]).

The "inner" loop can be made more efficient by sorting the polygons in $\bar{F}^Q$ along the x-axis, for instance (this requires only the x-slabs to be valid). Let $f \in F^P$ be the polygon to be checked against polygons of Q. Let $\text{bbox}^h_x(f)$ denote the upper x-coordinate of the bounding box of f (I will call this a *right bracket* — *left brackets* are defined analogously). Assume we do a scan over $\bar{F}^Q$ from left to right (low to high) along the x-axis, and assume furthermore that we consider only left brackets. Since all slabs are sorted, we can stop the scan after we have passed $\text{bbox}^h_x(f)$. Additionally, $\forall g \in \bar{F}^Q : \text{bbox}^l_x(g) \in \text{bbox}_x(f)$ we do not have to test the x-slabs for overlap. Only for those g, whose left

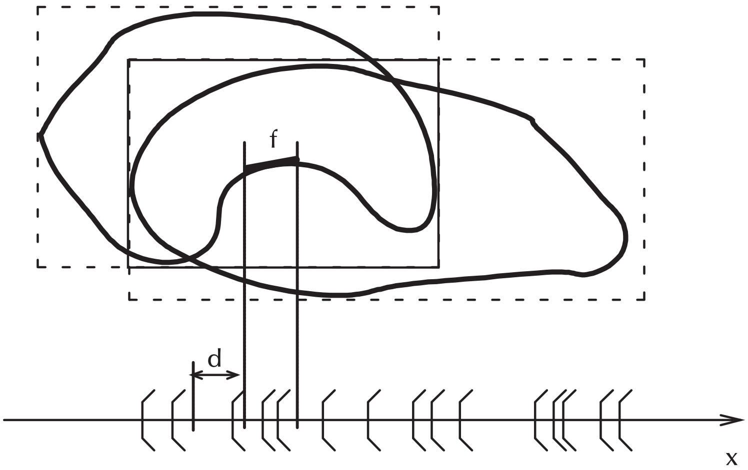


Figure 3.3: By sorting brackets of all polygons, one can significantly reduce the number of polygons to be visited.

bracket is left of f's left bracket, we need to test $\text{bbox}^h_x(g) \geq \text{bbox}^l_x(f)$. All of the above tests happen in Q's space.

In general, polygons are small compared to the size of the object. We can exploit this in order to get rid of almost all of the above comparisons. So, if the largest of all x-slabs in $\bar{F}^Q$ is d wide, then it is sufficient to consider only left brackets of $\bar{F}^Q$ in the range $[\text{bbox}^l_x(f) - d, \text{bbox}^h_x(f)]$, and only those g with $\text{bbox}^l_x(g) \in [\text{bbox}^l_x(f) - d, \text{bbox}^h_x(f)]$ have to be tested by comparison (see Figure 3.3). An upper bound for d is the maximum size of all polygons. Alternatively, d could be determined with each collision query. We can find the first left bracket just right of $\text{bbox}^l_x(f) - d$ by a combination of binary and interpolation search[2] [AHU74].

Finally, we do not need to consider all $f \in F^P$ in the outer loop. By sorting F^P with respect to $\text{bbox}^l_x(f)$ (in P's space) we can find quickly f_i such that $\text{bbox}^l_x(f_{i-1}) < \text{bbox}_P(Q) - e \leq \text{bbox}^l_x(f_i)$, where e is the maximal length of polygons of P.

So, the refined algorithm looks like:

<u>`Pipelined BBoxes`</u>

sort $\{\text{bbox}^l_x(f) \mid f \in f^P\}$ $\qquad$ {*needs to be done only once* }

$\bar{F}^Q := \{g \in Q | \texttt{bbox}(g) \cap \texttt{bbox}_Q(P) \neq \varnothing\}$
sort $\{\min_x(g) \mid g \in \bar{F}^Q\}$

find $f_k \in F^P$ *such that*
$\quad\min_x(f_{k-1}) < \texttt{bbox}_P(Q) - e \leq \min_x(f_k)$
`l = k ...`*until* $f_l > \texttt{bbox}_P(Q)$:
$\quad\texttt{bbox}_P(f_l) \cap \texttt{bbox}_P(Q) = \varnothing$
$\quad\quad\longrightarrow$ *next* f_l
$\quad\texttt{bbox}_Q(f_l) \cap \texttt{bbox}_Q(Q) = \varnothing$ $\qquad$ {$bbox_Q(f_l)$ *is needed anyway* }
$\quad\quad\longrightarrow$ *next* `l`

[2] Interpolation search is a variant of binary search which performs better by a factor if the data is evenly distributed.

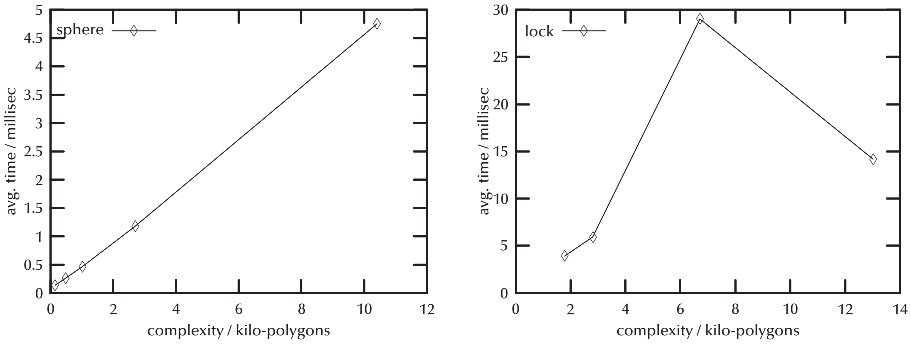


Figure 3.4: Performance of the BBox pipeline algorithm. Two identical objects have been tested using the procedure outlined in Section 3.5.10. The number of polygons is for one object. Left: sphere; right: door lock.

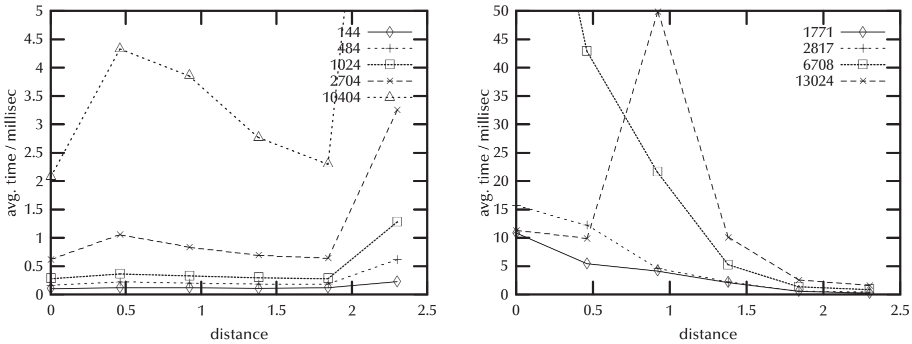


Figure 3.5: Performance of the BBox pipeline algorithm with respect to distance. Figure 3.4 has been derived from these data by averaging over distance. The number of polygons is for one object. Left: sphere; right: door lock.

find $g_i \in \bar{F}^Q$ *such that* $\{$*bboxes are Q's space* $\}$
  $g_{i-1} < \min_x(f_l) - d \le g_i$
`j = i` ...*until* $g_j > \max_x(f_l)$:
  $\texttt{bbox}_y(g_j) \cap \texttt{bbox}_y(f_l) = \emptyset \longrightarrow$ *next* `j`
  dito with z
  g_j `and` f_l `intersect`
    $\longrightarrow$ `return` "*intersection*"

This algorithm is well suited for parallelization (see Section 3.10)

3.3.1 Good and bad cases

Although the algorithm is still $O(n^2)$ in the worst case, it performs remarkably well in practical cases (see Figures 3.4 and 3.5), especially since it does not need any additional data structures.[3]

I believe this is due to two complementary effects:

- If objects do not collide and are not too close to one another, then there will not be many polygons in $\bar{F}^Q$, and not many polygons $f \in F^P$ pass the first bounding box test.

[3] More plots can be found at `http://www.igd.fhg.de/~zach/coldet/index.html#bbox-pipeline`

- If objects do intersect "fairly" (to be discussed in a moment), then chances are good that the algorithm will find a pair of intersecting polygons early.

The above are two "good" cases for almost all collision detection algorithms. There are also two bad cases:

- Both objects do not overlap but almost touch each other. In that case, there are many polygons in $\text{bbox}(P) \cap \text{bbox}(Q)$.
- Both objects overlap by a large amount, i.e., $\text{bbox}(P) \cap \text{bbox}(Q)$ is large compared to the volume of both objects. In that case, there are so many pairs of polygons to be tested, that it takes a long time until the algorithm finds one.

3.4 Convex polytopes

The restriction of a geometric problem to convex polytopes quite often results in more efficient algorithms. The same is true with collision detection.

Convex polyhedra allow one to view the problem in many more different ways than arbitrary objects could offer: one can consider a convex polyhedron not only as a collection of polygons, edges, and vertices with certain properties, but also as the intersection of half spaces, or as the convex hull of its vertices. These are not different internal representations of convex objects, but merely different ways to *look* at the data.

The general usefulness of convex collision detection algorithms by themselves is questionable,[4] at least in the area of virtual prototyping and physically-based simulation. However, convex hulls can be utilized earlier in the collision detection pipeline for filtering out unnecessary exact collision checks (see Section 3.9), and they might be useful for special applications like simulation of wires composed of cylinders [HD00].

Basically, there are two types of representation of convex hulls: *vertex representation* and *facet representation*. The construction problems are named *vertex enumeration* and *facet enumeration* problem, resp. Sometimes, the former is also called the *irredundancy problem* [Sei97], because finding the vertex enumeration is equivalent to finding the vertices which are not redundant.

The efficient construction of convex hulls has been a long-time area of research in the field of computational geometry. Two classes of algorithms can be distinguished: *graph-traversal* algorithms and *incremental* algorithms. The former are also called *static* or *off-line*, while the latter are also called *dynamic* or *on-line* algorithms. Graph-traversal algorithms conceptually consider all vertices at any time. They construct "the right" convex hull from scratch. On the other hand, incremental algorithms consider one vertex after the other. They usually start with the convex hull of a trivial subset of the vertices, which is expanded gradually. Some famous graph-traversal algorithms are the "gift-wrapping" [CK70] algorithm and [Sei86, AF92]. Examples of incremental algorithms are the "beneath-beyond" method [Kal84] and [CS88, Cha93, BDH93]. I used the latter, because an implementation is readily available [BDH96, BH97].

While many of the earlier algorithms have complexity $O(n^2)$ (including Qickhull), there are also algorithms with complexity $O(n \log n)$, such as the divide&conquer algorithm presented by [PH77]. However, it seems that Quick-

[4] Imagine one of the objects of Figure 3.42 decomposed into convex pieces, not to speak of non-closed objects.

hull is output sensitive in practice.

In the following I will discuss several algorithms for collision detection of convex objects.

3.4.1 Static algorithms

Computational geometry

When computational geometers directed their attention to the problem, the goal at first was to *construct* the intersection of two polytopes. Only later on, researchers realized that the *detection* problem is interesting by itself and can be solved in fact more efficiently than the construction problem. The main goal of research in this area has been to find algorithms with optimal asymptotical worst-case complexity. All results apply only to the strictly static case, and most algorithms consider geometrical features of an object to be given in world coordinates.

The first to present a construction algorithm which has an asymptotical complexity below the trivial $O(n^2)$ were [MP78]. Like many linear programming algorithms, the algorithm consists of two phases: the first one searches for a point in the intersection of the two polyhedra, the second phase then constructs the actual intersection (if any) by taking the dual of the two polyhedra, forming the union of these duals, and finally computing the dual of the result again, which yields the intersection. The overall complexity is $O(n \log n)$.

Another approach computes the distance of convex polytopes by a hierarchical boundary representation [DK85]. The bottom of any hierarchy is a tetrahedron. A separating slab of two convex polytopes can be found by the following procedure: construct a separating slab for the two tetrahedra, then maintain this slab as the algorithm moves upward in the hierarchy.

This algorithm has been improved by [MS85]. They present an algorithm which constructs the intersection of two polyhedra one of which can be non-convex. They also use the hierarchical representation of polyhedra and achieve the same upper bound.

The best upper bound for the detection problem, to my knowledge, is given by [DK83]. The algorithm has worst-case complexity of $O(\log^2 n)$, $n = |V|$. However, the algorithm seems to be very involved, and no implementation is known to me. Besides, it assumes the polyhedra to be pre-processed in a certain way which does not lend itself directly to moving objects. Other works on the detection problem are [CD87, Rei88].

Clipping

Collision detection between two convex polytopes can be solved by clipping. Clipping at convex polytopes is solved by the Cyrus-Beck algorithm. Again, clipping is the construction problem, and the detection problem ("does an edge have a common point with a convex polytope?") can be solved more efficiently [Kol97]. The idea is that an edge is supported by a pencil of planes. The edge does not intersect the polytope, if there are two planes which do not intersect the polytope. Checking planes for intersection with the polytope can be done more efficiently than actually clipping lines. By transforming the problem into *dual space* it can be done even more efficiently. This is because planes become points, convex polytopes become a pair of convex, piecewise linear, continuous functions, and the problem is to determine whether or not the point (i.e., plane) is contained in either of the convex functions (i.e., a point location problem).

I compared the Cyrus-Beck clipping algorithm (including several bounding box pre-checks) with the algorithm of Section 3.3. The result is that clipping is much slower (up to 2 times slower for objects with only 250 polygons)! This is because with clipping much less pre-checks can be done.

3.4.2 Incremental convex algorithms

It seems that convex objects are particularly well suited for incremental algorithms. Several algorithms have been presented exploiting temporal coherence.

The first one seems to have been presented by [LC92, CLMP95], which tracks *closest features*. [vdB99] has improved the GJK algorithm ([GJK88]) by turning it into an incremental version using the notion of a separating axis. A different approach at computing a separating axis has been presented by [Chu96]. However, their proof for convergence seems to be wrong ([vdB99]).

3.4.3 Separating Planes

Inspired by neural networks (more precisely, by perceptrons [HKP91]) I have developed an algorithm which does not require any preprocessing of the polytopes except its adjacency map.

If we consider a convex polyhedron to be the convex hull of its vertices, we can use the notion of linear separability: P and Q do not intersect iff there is a plane h such that all vertices of P are on one side, and all vertices of Q are on the other side (see Figure 3.6). Such a plane is called a *separating plane*, and P and Q are called *linearly separable*. Let $P = \{p^1, \ldots, p^n\}, Q = \{q^1, \ldots, q^m\} \subseteq \mathbb{R}^3$. Then

$$\begin{aligned} &P, Q \text{ are } \textit{linearly separable} && :\Leftrightarrow \\ &\exists w \in \mathbb{R}^3, w_0 \in \mathbb{R}\, \forall i,j : p^i \cdot w - w_0 > 0\ ,\ q^j \cdot w - w_0 < 0 \end{aligned}$$

Translating the above into projective space we get

$$\begin{aligned} &P, Q \text{ are } \textit{linearly separable} && \Leftrightarrow \\ &(p^i, -1) \cdot (w, w_0) > 0\ ,\ (q^j, -1) \cdot (w, w_0) < 0 \Leftrightarrow \\ &(p^i, -1) \cdot (w, w_0) > 0\ ,\ (-q^j, 1) \cdot (w, w_0) > 0 \end{aligned}$$

If P and Q do intersect, the algorithm presented above will not terminate. This can be fixed in two ways: One way would be to stop the loop after a certain amount of iterations. If it has not found a separating plane, we will just assume that P and Q are not linearly separable. This would turn the algorithm into a *probabilistic* one, biased towards “not linearly separable”.

There is a very simple algorithm to compute w for perceptrons:

<u>`Perceptron learning rule`</u>

`input:` $Z = \{z^k\} := \{(p^i, -1), (-q^j, 1)\}$
`guess an arbitrary start vector` w^0*`, say,`* $(0, 0, 0, 1)$
`loop`
  $\exists z : z \cdot w^l < 0$
  $\longrightarrow w^{l+1} := w^l + \eta \cdot z$

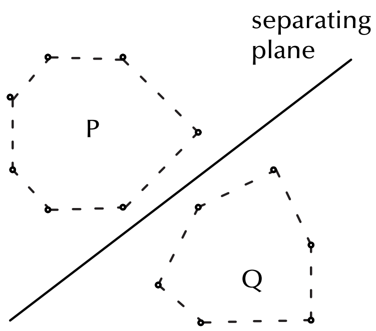


Figure 3.6: Two convex polyhedra do not intersect if they are linearly separable.

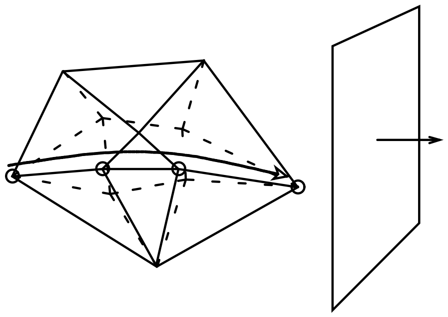


Figure 3.7: With hill climbing, the separating plane algorithm needs to visit only a fraction of all the points.

$\forall z : z \cdot w^l > 0$
$\longrightarrow \uparrow w$ `is separating plane`

The idea of this algorithm is to "turn" the separating plane a little bit whenever it finds a point z which is still on the "wrong" side of it.[5]

In order to avoid calculating $z \cdot w^l$ for all points, I use hill climbing to find the global minimum. If the start point is a long way from the minimum, hill climbing will walk on a "trail" toward that minimum calculating the dot product only for a fraction of all the points (see Figure 3.7). If the plane was not a separating plane, it is moved, but the new minimum with respect to the new plane is probably not far from the old minimum.

The algorithm above assumes that the points of P and Q are given in the same coordinate system. However, since one of them or both are moving, this is not true. So, the points first have to be transformed into world space.[6] We can save that transformation by transforming the plane w into P's and Q's coordinate system, resp.: $w \cdot z^\mathsf{T} = w \cdot (p \cdot M)^\mathsf{T} = (w \cdot M^\mathsf{T}) \cdot p^\mathsf{T}$, where M is P's and Q's transformation matrix, resp. That way, we actually maintain the plane in both coordinate systems.

There is another possibility to make the algorithm terminate always: we can establish an upper bound $l_{\max}$ for l, the maximum number of loop iterations needed to find a separating plane for two linearly separable polyhedra. This upper bound is independent of the number of vertices; it depends only on the distance D between P and Q. The mathematical details can be found in [Zac94b, HKP91].

However, computing the distance is not really what we want to do, because then we do not need to find a separating plane anymore. And since we plan to use convex collision detection only as a pre-check in the collision detection pipeline, it does not really matter if the test occasionally returns a wrong answer *as long* as it never dismisses a pair of objects which actually do collide.

Simulated annealing. The algorithm above is an optimization algorithm. The goal is to find the global maximum of $D(w) = \frac{1}{|w|} \min\{wz^i\}$. Step by

5 A movie showing the algorithm in action can be found at `http://www.igd.fhg.de/~zach/coldet/index.html#movies`

6 We could instead choose P's or Q's coordinate system, too, but here this wouldn't improve performance.

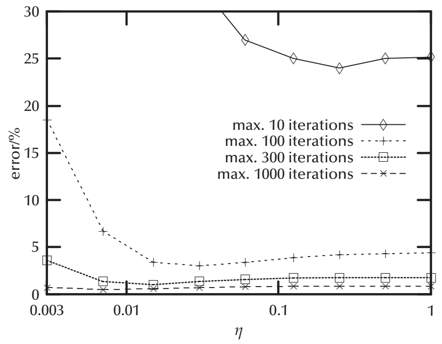


Figure 3.8: Impact of η and the max. number of iterations on the correctness of the separating planes algorithm. The error is the number of times where the separating planes algorithm returned with "collision", but there was none, in fact.

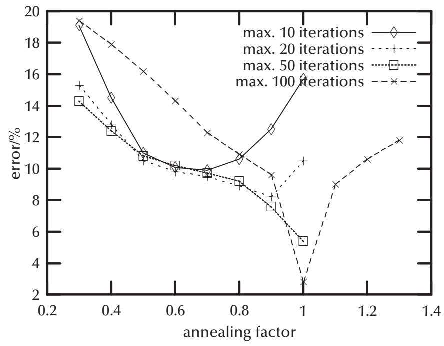


Figure 3.9: Optimal annealing factor for η with various maximum iterations; the initial η is the optimal one when doing no annealing. A factor of 1.0 means no annealing is being done.

step, the algorithm tries to move further into the direction of that maximum. So, like other optimization algorithms, I tried simulated annealing to increase the convergence. A "temperature" factor of 0.97 turned out to be optimal.

Another way to increase convergence could be to base η on the "badness" of the current (not yet separating) plane with respect to the current point being on the wrong side. Then, an update would be computed by $w^{l+1} := w^l + \frac{w^l z}{|w^l||z|} \cdot z$. However, it is not clear (without actually trying) that this approach is really faster, since the update equation is computationally much more expensive than the original one.

Incremental collision detection. The separating planes algorithm can be easily extended to an incremental method (which I have implemented): for each pair of objects we store the plane which the algorithm ended up with, and the two points realizing the global maximum of the hill-climbing step. When the same pair of objects is checked for collision the next time, we use this plane for the initial "guess", and the two points for the starting points of the hill-climbing. If objects have not moved very much since the last collision query, this initial guess will probably be a good guess. If the two objects did not collide the last time, and they do not collide this time, chances are good that it suffices to check that the two cached points still realize the global maximum. If they are, then the hill-climbing step will just visit all their neighbors.

Results. Several tests were carried out to evaluate the feasibility of the approach.[7]

The first test was designed to find out the impact of the two parameters: the maximum number of iterations, and the update parameter η (no annealing); see Figure 3.8 for the results.

I also tried to measure the effect of how the initial plane is computed. Three methods were considered: the trivial plane $w = (1, 0, 0, 0)$; the plane through

[7] Scenario: two cones bouncing off each other.
Invocation: `movem -x 10 -t 1000 co -e 1.5 -p <eta> <max.iter.>`.

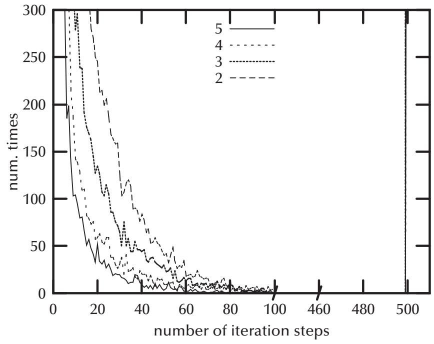


Figure 3.10: The maximum number of steps can be chosen fairly precisely such that neither too many false collisions are reported nor too many futile iterations are performed. In the plot, each graph corresponds to a different extent of the environment, which consists of 10 objects bouncing off each other.

the midpoint between the two barycenters of P and Q, and the normal of the plane is the line through the two barycenters; and, finally, like the previous method but using the barycenters of the bounding boxes of P and Q.

initial plane	error/% (max. num. iter.)			improvement		
	iter. ≤ 10	iter. ≤ 100	iter. ≤ 1000			
	$\eta = 0.25$	$\eta = 0.03$	$\eta = 0.007$			
trivial	24	3.0	0.50	1.0	1.00	1.00
barycenters	16	2.8	0.37	1.5	1.07	1.35
bbox centers	15	2.8	0.37	1.6	1.07	0.37

This shows that we can save some time by choosing a non-trivial estimate of the separating plane. In addition, it shows that a very simple estimate serves well enough.

The last test's purpose was to find out the optimum factor for the annealing of η. Again, the test was done for various maximum numbers of iterations. For each of these, the optimum initial η was chosen. The initial guess for the separating plane was done with the third method, i.e., the plane between the barycenters of the bounding boxes. See Figure 3.9 for the results. It is not clear to me why annealing helps only in the cases where 10 or 20 iterations are the maximum.

Finally, I tried to determine a "good" value for the maximum number of iterations. The smaller that number is, the more collisions might be reported in cases where the objects are only almost touching; and the larger it is, the longer the algorithm needs until it can assume a collision in those cases where the objects are really intersecting. Fortunately, based on Figure 3.10 a fairly good choice of the maximum number of steps can be made. It is a histogram of the number of iterations the algorithm needed until it found a separating plane, depending on the density of the environment. The numbers were determined by having 10 convex objects bounce off each other in a cube of the given size. The maximum number of iterations was set to a very large value.

Adaptation to the future by feedback. The separating planes algorithm can adapt to the constellation of the two objects by feedback from the exact collision detection: when the exact collision detection has found a collision, it is very

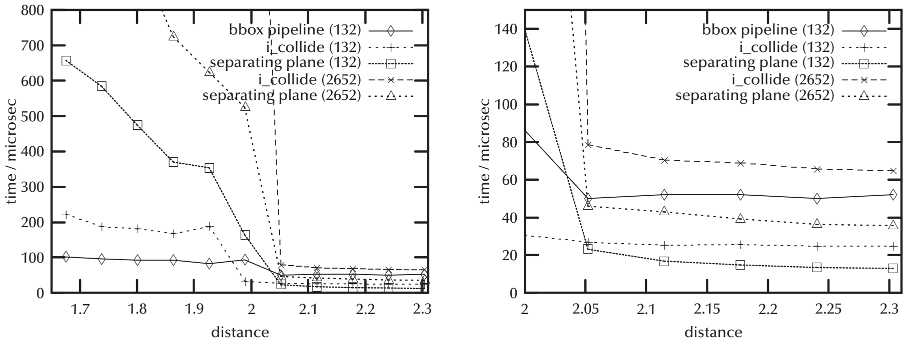


Figure 3.11: Dependency of two convex incremental collision detection algorithms on distance. The numbers in parentheses denote the number of polygons of each object. It is remarkable that for small polygon counts in the case of intersection, the bounding box pipeline algorithm is faster than the incremental ones.

likely that there will be a collision during the next frame. In that case, it is futile to try to find a separating plane for the next frame. So we can as well decrease the maximum number of iterations.

Should there be no collision, although the separating plane algorithm has taken the maximum number of iterations, then we increase the maximum number of iterations. In such a situation, we might have prevented doing an exact collision detection if the separating plane algorithm had done more iterations.

This adaptation tries to predict the future and do less work since the result is probably known already. Only when the prediction has failed, the algorithm has to spend more work in the current frame (and a little bit more in the future).

Comparison between separating planes and Lin-Canny. I compared my implementation of the separating planes algorithm with the Lin-Canny algorithm as implemented in I_collide [CLM+]. That implementation was ported onto the same data structures as the separating planes algorithm lives on.

With incremental algorithms, several dependencies should be examined. How does the average collision detection time depend on the number of polygons (it should be sub-linear), the distance, and rotational velocity? In addition, the usual dependencies should be checked, in particular, how does it perform when objects collide?

For the following benchmarks, I have used the scenario as described in Section 3.5.10, except that for fairness, the moving object does not spin around its center, instead it orbits around the stationary one (if it would only spin, then the separating plane would never have to be moved again once a valid plane has been found). Two spheres have been chosen for producing the following benchmarks, because they are "uniform". For the separating planes algorithm, the maximum number of steps was set to 300. Times are averaged over 5,000 samples for each distance. At all distances, there were only $0 - -3$ wrong results, except for distance 2.000060 which yielded 1256 wrong results.

The plots in Figure 3.11 show how collision detection time depends on distance between two objects (each time is the average over many samples and rotational velocities). It is remarkable that for small polygon counts in the case of intersection the bounding box pipelining algorithm (see Section 3.3) is faster than any of the incremental algorithms. Furthermore, I am surprised that the

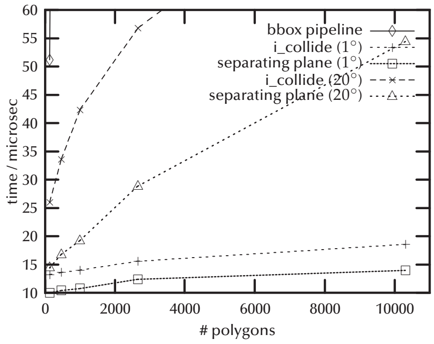


Figure 3.12: Collision detection time seems to be sub-linear in the number of polygons, where the hidden constant depends on the rotational velocity.

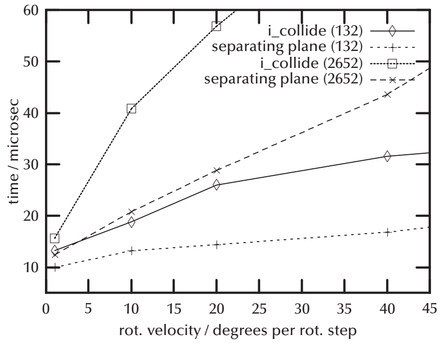


Figure 3.13: The complexity of I_collide seems to be sub-linear in rotational velocity while my separating planes algorithm seems to be linear, but with a much smaller constant.

separating planes algorithm depends very little on distance. One would suspect that with almost touching distances (such as 2.05) the algorithm would need significantly more time to find a separating plane.

Figure 3.12 shows how the performance of the two algorithms depends on the number of polygons and on the rotational velocity. Here, time has been averaged over distances > 2. It is no surprise that the "almost $O(1)$" claim ([LC92]) holds true only for small angular velocities (like 1 degree per step).

It is obvious that collision detection time depends (almost) linearly on the angular velocity, with the factor determined by the number of polygons (see Figure 3.13).

Robustness. It seems that the separating plane algorithm is much more robust than I_collide. I am fairly positive that during the port of I_collide to our data structures I have not impaired its robustness.

I observed that with certain rotational velocities, the algorithm reported collisions although the two spheres had a distance > 2.5 (even with moderate polygon counts). In addition, with polygon counts $> 5,000$ the algorithm ran into cycles with length > 200 but < 1000, even with distances well above 2. This means that the algorithm considered over 200 feature pairs before it finally detected a cycle (and thus assumed that a collision has occurred).

Maybe these problems persist because of the "flaring" of Voronoi regions. By flaring, some planes are turned "in", some "out", and the amounts are chosen such that afterwards all Voronoi regions overlap slightly. So, it could be that with large polygon counts (i.e., small dihedral angles), even non-adjacent Voronoi regions overlap.

Maybe it is not the flaring itself causing the problems, but the fixed amount (in degrees). On the other hand, it might be non-trivial to calculate the right amount numerically stable.

3.4.4 A simplified Lin-Canny algorithm

To my knowledge, the algorithm presented by [LC92, CLMP95] was the first one to exploit temporal coherence. It has expected sub-linear running time, depending on the relative rotational speed of the object pair (see also previous section).

The idea is to track the minimal distance between two objects. That distance is realized by two *features*, where a feature denotes a vertex, an edge, or a polygon. A pair of features realizing the minimal distance between two objects is called *closest features*. If a pair of features has been closest features during the last frame, then it is likely that the same pair is also the pair of closest features in the current frame. If it is not, then the new pair of closest features is "nearby", depending on how much the two objects rotated relative to each other.

Given a pair of features, a new pair of closest features can be found efficiently by kind of a "steepest decent" method. This is motivated by the following neat lemma based on the outer Voronoi diagram of polyhedra:

Lemma 1 (Closest features, [LC92])
Given two features f, g of P and Q, resp.; let V_f, V_g be their outer Voronoi regions, resp. Let p, q be two points on f, g, resp., realizing the distance $d(f, g)$. Then

$$\begin{gathered} f, g \text{ are closest features } \Leftrightarrow \\ p \in V_g \text{ and } q \in V_f \end{gathered}$$

So, if a pair of features does *not* fulfil the condition of closest features, then p, for instance, is "behind" one of the planes enclosing V_g. It is easy to see that the feature g' associated with the Voronoi region "behind" that plane is closer to f than g. This is already the algorithm: start with any two features, test the "closest feature" condition, then move to the next pair of features based on the Voronoi plane which is being violated (if any).

There are two issues that must be taken care of in an implementation: If the objects penetrate, then the algorithm might be caught in a cycle and never find a pair of features with distance 0. Therefore, it must maintain a list of features already visited. The second issue is numerical stability: adjacent Voronoi regions share a plane and one of p or q might lie exactly (within the machine's precision) on such a plane (because the edge, for instance, intersects that plane).

The Voronoi diagram is easy to construct in this case. We only need to construct the outer regions, and these regions are simple in that their boundary planes are perpendicular to edges and polygons.

It is possible to do closest-feature tracking without Voronoi regions. This is an advantage, because constructing the Voronoi diagram still takes some time at start-up of a VR system, and it takes a considerable amount of memory. Without a Voronoi diagram, the algorithm could even handle deforming objects, as long as the stay convex and the topology does not change.

The idea is to replace the "closest-feature" criterion and to consider only faces.

The distance $d(f, g)$ between two faces f and g is realized by two points which can either lie on their interior, or on the interior of an edge, or on a vertex.

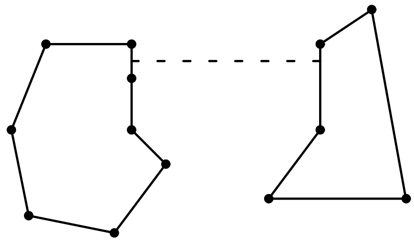


Figure 3.14: My Voronoi-less algorithm for convex objects can handle this pathological case, too.

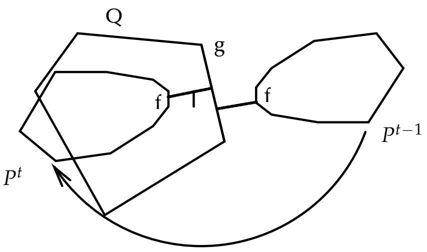


Figure 3.15: When a pair of closest faces has been found, they must be checked for being on the inside of the other object.

Lemma 2 (Closest faces)
Given two faces f, g of P and Q, resp. Then

$$f, g \text{ are closest faces } \Leftrightarrow$$
$$d(f,g) = \min_{f,g}\{d(f',g') \mid f' \text{adjacent to } f,\ g' \text{adjacent to } g\}$$

So, the algorithm is really quite simple: given a pair of faces, check the distance of all pairs of adjacent faces. If any of them has smaller distance than the current pair, "go" to that one. If the algorithm is to implement a steepest decent (or "ascent", if we want to call it "hill-climbing"), then it has to check all adjacent pairs, and then go to the one with the smallest distance.

There is a pathological case (see Figure 3.14). It can occur only, if both convex objects have adjacent coplanar faces, and then only, if the objects happen to be placed so that those faces are parallel. The algorithm can be modified slightly so that it can deal with such a situation: if there are pairs with the same distance as the current one, then they are visited, unless they have already been visited. An alternative might be to "joggle" the vertices slightly ([BH95, BS90]), so that the faces are no longer coplanar, while making sure that the object stays convex and the faces stay planar.

The advantages of the Voronoi-less algorithm are that it does not need any Voronoi diagram, and it is much easier to implement numerically stable (using simple ϵ-arithmetic). In addition, it can do a steepest decent, which is not trivial to do with the Voronoi-based approach, and which, to my knowledge, has not been done in the widely used reference implementation I_collide [CLM$^+$].

On the other hand, the Voronoi-based algorithm has to compute the distance for only one pair of features and has to do two point-in-Voronoi-region tests, in order to proceed to the next pair of features.

Both algorithms have to take special measures in order to detect intersection. In case of an intersection, the Voronoi-based approach will be caught in a cycle, although the length is usually quite short (in my experience, mostly 2 or 4).

With my algorithm, intersection is checked after a pair of closest faces has been found. Of course, if during hill-climbing a pair with distance 0 is found, then this is an intersection, and the algorithm is done. Otherwise, we need to check that the two faces are outside the other polyhedron. Figure 3.15 shows an example where an intersection is found only with that check: at time $t-1$, (f,g) are the closest faces; then, object P is rotated. When the algorithm tries to find a new pair of closest faces at time t, all pairs adjacent to (f,g) have a greater distance.

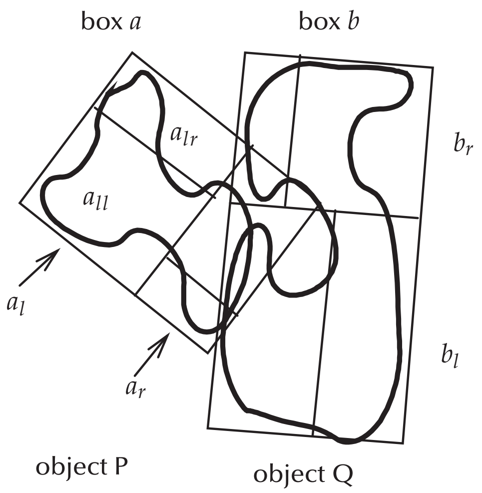


Figure 3.16: The basic idea of hierarchical collision detection: only faces and edges of overlapping boxes have to be checked for intersection.

3.5 Hierarchical collision detection

Before a pair of objects is tested for intersection, a bounding box test is usually done. This is a conservative test of "non-intersection" on both sets of polygons as a whole. By taking this idea further, applying it recursively to polygon subsets of both objects, and using various types of bounding volumes, we arrive at the notion of bounding volume hierarchies and hierarchical collision detection. Basically, this is a *divide-&-conquer* approach.

The idea is to construct at initialization time a BV tree for each object. During run-time, when a collision query is to be processed for a given pair of objects, the transformations of the objects are applied to their BV trees (which can be more or less expensive computationally, depending on the type of BV hierarchy). By traversing the BV trees, regions of interest can be found more quickly; these regions are those parts of the objects which are "close" to one another.

3.5.1 Outline of hierarchical algorithms

The goal of any hierarchical collision detection scheme is to discard quickly many pairs of polygons which cannot intersect. For the sake of simplicity, I will illustrate the basic idea with the example of boxes (see Figure 3.16): assume boxes A and B overlap; if box A_1 does not overlap with box B_1, then we do not need to check any polygon enclosed completely by A_1 against any polygon enclosed completely by B_1. analogously, we can use the other 3 box-box tests to prevent checking unnecessary polygon pairs. Of course, this is done recursively, which yields a *simultaneous traversal* of two bounding volume hierarchies.

The following pseudo-code outlines the basic scheme of collision detection algorithms based on a bounding volume hierarchy:

```
Simultaneous traversal of BV trees

a = bounding volume of A's tree,
b = bounding volume of B's tree
a[i], b[i] children of a and b, resp.
```

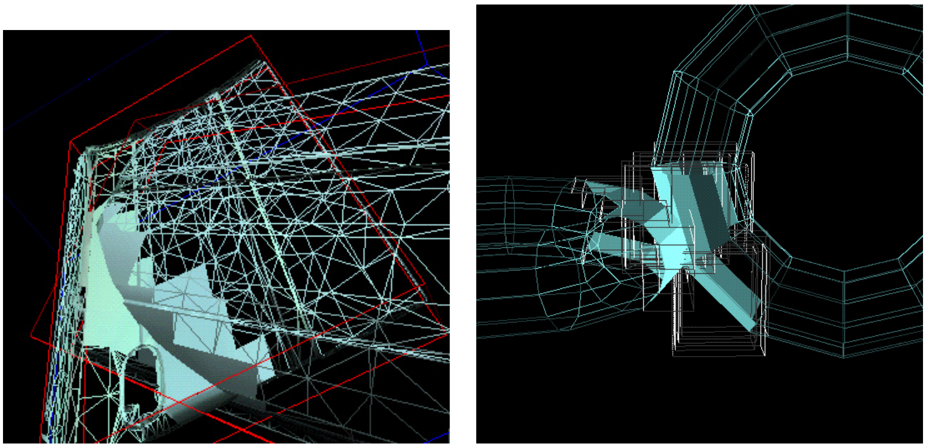


Figure 3.17: All polygons which are considered for intersection in the worst-case are rendered solid. The rejection of polygons is based on a hierarchical BV data structure. Depending on the tightness of the BV, the set of overlapping leaves approximates more or less the region of intersection.

```
traverse(a,b):
a or b is empty ⟶ return
b leaf ⟶
   a leaf ⟶
      process primitives enclosed by a and b
      return
   a not leaf ⟶
      forall i:
         a[i],b overlap ⟶ traverse(a[i],b)
b not leaf ⟶
   a leaf ⟶
      forall i:
         a,b[i] overlap ⟶ traverse(a,b[i])
   a not leaf ⟶
      forall i:  forall j:
         a[i],b[j] overlap ⟶ traverse(a[i],b[j])
```

With some BV schemes, it can be more efficient sometimes, if one (or more) of the children of a BV are empty. This can help to approximate the object better. The test between an empty and a non-empty BV is trivial.

Usually, leaf BVs will contain exactly one polygon (or primitive). However, this is not inherent to the idea of hierarchical BV trees or simultaneous traversal.[8]

Figure 3.17 visualizes the set of polygons considered in the worst-case for exact intersection calculations. All hierarchical BV schemes produce similar images.

So, the problem of hierarchical collision detection is basically to find a type of BV which can be tested for overlap efficiently even if tumbling through space. In addition, the BVs should enclose their associated set of polygons as tightly as possible in order to provide "optimal" BV trees (see Section 3.5.2).

Besides collision detection, BV trees could be used for other functions, too: the only part that would have to be re-defined is the "*process primitives*" step,

[8] In fact, with an early BV scheme I have found that it is better to have more than one polygon enclosed by a BV. This was due to quite expensive BV overlap tests.

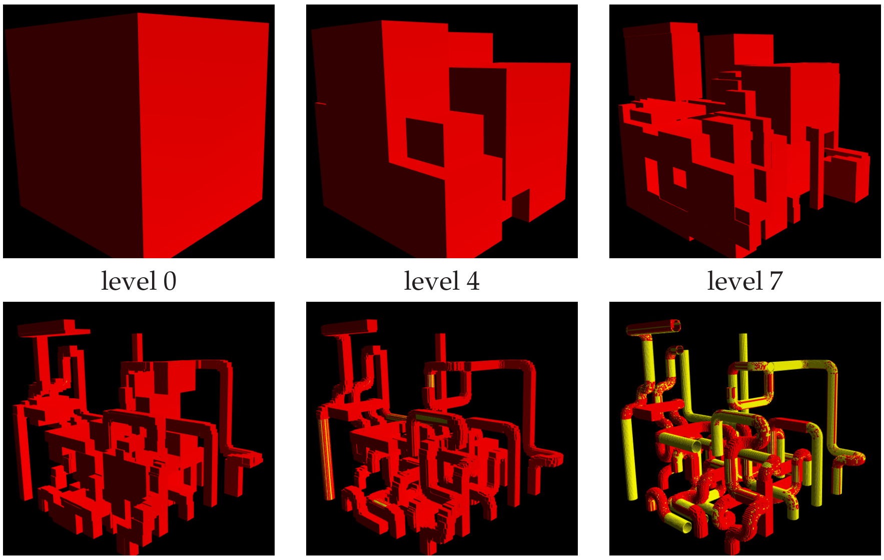


Figure 3.18: A BV hierarchy of boxes.

which provides the "semantics" of the overall operation (see [NAT90a] for a similar point of view regarding BSP trees).

A different view on bounding volume hierarchies

So, the data structure associated with hierarchical collision detection is *bounding volume trees*. Each node in such a tree is associated with a subset of the primitives of the object, together with a bounding volume (BV) that encloses this subset. Given two objects and the roots of their associated BV trees, a simultaneous traversal of the two trees recursively checks all pairs of their children BVs for overlap. If such a pair does not overlap, then the polygons enclosed by them cannot intersect.

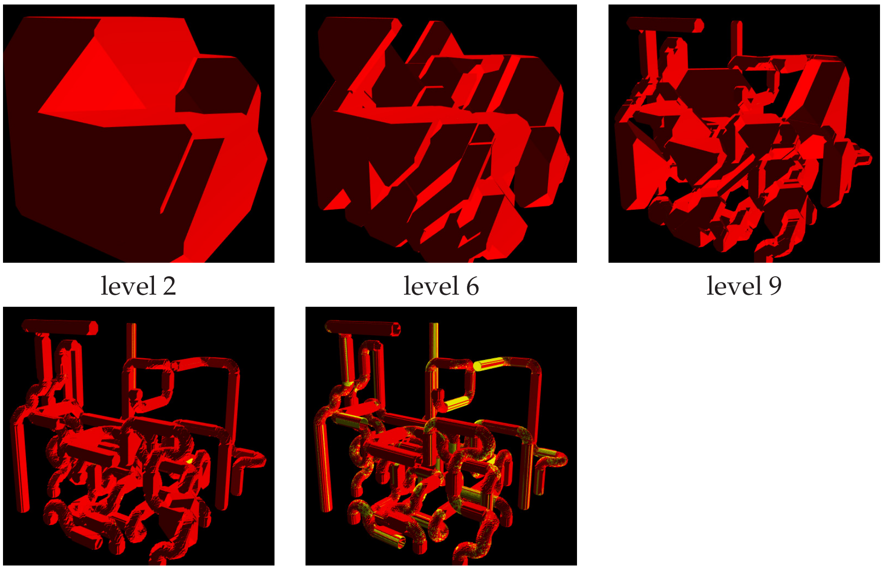
BV hierarchies can be viewed as hierarchies of successive refinement, or as levels-of-detail (LODs) for geometry — which are not suitable for rendering, of course. Figures 3.18 and 3.19 show different levels of a BV hierarchy. Figure 3.18 has been built using boxes as bounding volumes, while Figure 3.19 uses DOPs.

Constant frame rate

Sometimes, a VR system needs to sustain a constant frame rate, in order to achieve a certain "smooth feel".[9] Games are notorious for this need for a constant 60 Hz frame-rate.

So, if the collision detection module is part of the main loop, it can spend only a certain, limited amount of time on collision detection. Even if it is concurrent, it can be desirable to make it sustain its own constant "frame-rate".

[9] With high frame-rates (25–60 Hz), changes in frame-rate become much more noticeable than in the low range (8–20 Hz). A user will notice such a change as a sudden "jerk". I suspect this is because frame-rates can be only multiples of the video refresh rate, e.g.: 60, 30, 20, 15, 12, ... So, in the high range, frame rates can change only about a factor 2 or 1.5, while in the lower range, the factor gets closer to 1.



level 2 level 6 level 9 level 13 level 16

Figure 3.19: A BV hierarchy of 14-DOPs.

Hierarchical algorithms are, due to their recursive refinement nature, well suited for sustaining a constant frame rate [Hub95, OD99]. The basic idea is that traversal of the hierarchies is interrupted when the allocated amount of time is over. There are several strategies to get the most information out of the time available.

When traversal could be interrupted, one has to choose between depth-first and breadth-first traversal. If the application wants to bias the collision detection more towards the conservative side, then one would probably choose breadth-first traversal and return "collision" if traversal has been interrupted. In addition, if there are several pairs to be checked for collision, they should probably be sorted according to the time they took to check during the last loop. A pair which took little time during the last frame (and, therefore, must have been non-colliding), will probably take little time in the current frame, too.

3.5.2 Optimal BV hierarchies

It is obvious that any hierarchical collision detection algorithm can be only as good as its associated algorithm for constructing the hierarchies. Therefore, it is important to develop algorithms for constructing "optimal" BV hierarchies in the following sense: a simultaneous traversal will find polygons "close" to each other as quickly as possible for any position of the objects relative to each other. "Good" hierarchical data structures for ray-tracing are characterized by a low stabbing number [BCG+96]. Similarly, "good" BV trees for collision detection are characterized by a low *overlap number* (but see below).

To my knowledge, it remains still an open problem as to exactly what an optimal BV hierarchy is. Is it possible to find a local characterization for optimal BV trees, i.e., is there some characterization of a BV and its father and children such that a BV tree consisting only of optimal BVs will be optimal, too? Maybe, there is no such local characterization; then the question would be, is there a global characterization of the tree itself, which can be computed using only the geometry of the tree itself. Is minimum total volume such a characterization? or minimum depth? or with largest polygons at the topmost leaves?

It is not clear yet, whether there is a single measure which should be optimized during the construction of a BV hierarchy in order to achieve an optimal tree for collision detection. Obviously, the following criteria should guide the construction algorithm:

- The total volume of all BVs should be small [BCG+96].
- The tree should be balanced in terms of polygon counts.
- The volume of overlap of two siblings should be small.
- The tree should be fairly balanced in terms of polygon count. If there are several possible subtrees with approximately same balancedness, then the one is to be preferred which yields the tighter bounding volumes.
- "Sphere-like" BVs should be preferred, because they minimize the total volume of the BV hierarchy.

There are three general ways to construct a BV hierarchy: *insertion* methods [GS87, BKSS90], *bottom-up* methods [BCG+96], and *top-down* methods. Insertion methods start with a single polygon and a trivial tree; then, one after the other, each polygon is inserted at the top of the tree and sifted down by certain criteria. Bottom-up methods enclose each polygon by an elementary BV; then, in each step two (or more) of them are selected and merged. Top-down methods proceed just the other way round: the set of all polygons is associated with the root of the tree; then, the set of polygons is split into two (or more) sets according to certain criteria, and associated with the children.

So far, I have chosen the top-down method for all my BV hierarchies.

3.5.3 The cost of hierarchies

The performance of any collision detection based on hierarchical bounding volumes depends on two conflicting constraints:

1. the tightness of the BVs, which will influence the number of BV tests, and
2. the simplicity of the BVs, which determines the efficiency of an overlap test of BVs.

This can be turned into a cost equation [GLM96]:

$$T = N_b T_b + N_p T_p \tag{3.1}$$

where T is the total time, T_b and T_p are the time to check one bounding volume and primitive, resp., and N_b,N_p are the number of bounding volume and primitive tests, resp.

Tightness of a bounding volume

One of the open questions is if there is a *best* bounding volume. Probably, this question cannot be viewed independently from the algorithm, because the algorithm is often tailored for a certain BV.

One criterion might be the ratio V/A of a BV (V = volume, A = area). However, for box, sphere, and DOP, this is not such a suitable criterion. If we assume a cube, a sphere, and a dodecahedron to be the best-case BVs for box-, ellipsoid-, and DOP-tree, resp., then that ratio is the same for all of them: $\frac{V}{A} = \frac{1}{6}d$. (The diameter of a cube is the length of its side, the diameter of a dodecahedron is the diameter of its in-sphere.)

3.5.4 The BoxTree

Inspired by BSP trees, k-d trees, and balanced bipartitions (known in the area of VLSI layout algorithms), I explored the usefulness of axis-aligned boxes. This type of BVs offers some advantages: it is fairly easy to construct the BV tree (see Section 3.5.2), and the simplicity of the BV hopefully makes it efficient to test two of them for overlap.

They are called *axis-aligned* boxes, because they are aligned to the axes of the object's coordinate frame. Of course, when objects move, they are no longer aligned with respect to the world's coordinate frame. So, when using this type of BVs, the task is to devise algorithms for efficient overlap tests of tumbled bounding boxes.

Since each node of the tree is a box, I have named this BV hierarchy a *BoxTree* [Zac95, Zac97b]. It is a binary tree. For reasons which will become clear in a moment, the child boxes are arranged in a certain way with respect to their father box: They have exactly the same extents except for one coordinate along the *splitting axis*: if a_l, a_r are left and right child box of a, resp., then $\text{bbox}(a) = \text{bbox}(a_l)$ and $\text{bbox}(a) = \text{bbox}(a_r)$ *except* $\text{bbox}^h_\alpha(a_l) \le \text{bbox}^h_\alpha(a)$ and $\text{bbox}^l_\alpha(a_r) \ge \text{bbox}^l_\alpha(a)$, where $\alpha \in x, y, z$ is the splitting axis.

Octrees are a special case of BoxTrees. However, BoxTrees allow for finer control on the balancedness.

Each node in the BoxTree stores only: the name α of the splitting axis, two positions $\text{bbox}^h_\alpha(a_l)$, $\text{bbox}^l_\alpha(a_r)$, two pointers (to the left and right sub-box), and, for leaves only, a pointer to the polygon(s) associated to the leaf (which can be stored in one of the child-pointers).

3.5.5 BoxTree traversal by clipping

For this BoxTree traversal algorithm, I constrain the child boxes further to be a true partition of the father box, i.e., $\text{bbox}^h_\alpha(a_l) = \text{bbox}^l_\alpha(a_r) = c^a_\alpha$. Although this reduces the flexibility during construction of BoxTrees, it allows the traversal algorithm to re-use more calculations across recursions.

The intersection test of two boxes could be done by the Liang-Barsky algorithm [LB84]. However, exploiting the special geometry of boxes allows a much more efficient intersection test for two boxes: we can clip all box-edges parallel to one another at the same time. This will enable us to re-use many computations during one box-box check. Due to the special arrangement of child boxes within a father box, we can re-use *all* of the arithmetical computations when descending one level in the BoxTree. Special features of boxes are: the faces form three sets of two parallel faces each, the edges form three sets of four parallel edges each, when a box is divided by a plane perpendicular to an edge, all edges retain their entering/leaving status.

One step of the traversal algorithm corresponds conceptually to splitting one box of a pair of boxes (a, b) (see Figures 3.20 and 3.21) and calculating the overlap status of the two new pairs of boxes. Such a step can be performed with at most 72 multiplications and 72 additions.

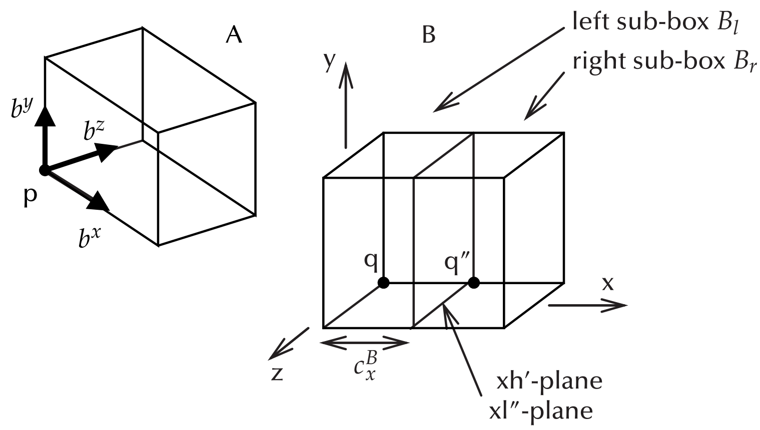


Figure 3.20: Splitting box B perpendicular to its x-edges bounds the line intervals of edges of A.

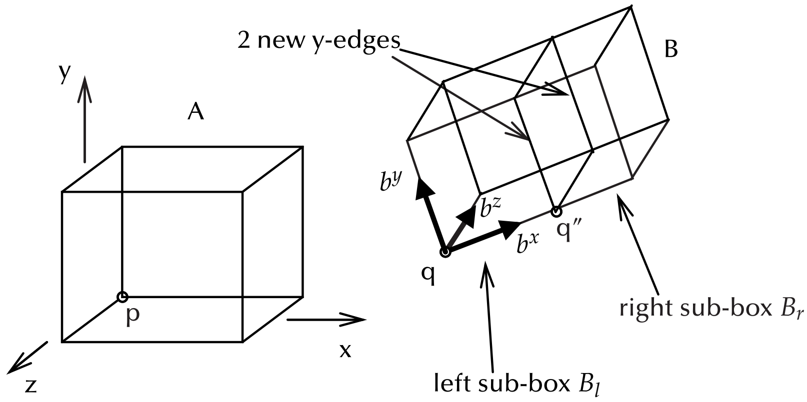


Figure 3.21: Splitting box B perpendicular to its x-edges yields 2 new y-intervals and 2 new z-intervals. All other intervals can be re-used.

For a given pair (a, b) of boxes, all the information on their intersection status is given by two sets of 3×4 line parameter intervals for the edges of a and b, resp. If all intervals of one object are empty, then (a, b) do not overlap.

The simultaneous traversal has two phases: an initialization phase, and the traversal phase. I will briefly describe them in the following — the mathematical details can be found in [Zac95].

Initialization phase. This phase computes the initial intervals for the root boxes of two objects P and Q. Conceptually, we have to set up 2×3 tables, each with 3×4 entries. Each entry in those tables can be calculated by at most one multiplication and one addition. Each column of a table yields the line parameter interval of one edge.

The calculations of this phase can be done by the same routines which are needed for the traversal phase.

Traversal phase. The basic step of the traversal is the test "(a, b_l) *intersect*" and "(a, b_r) *intersect*". We will do this by bisecting the box b into its left and

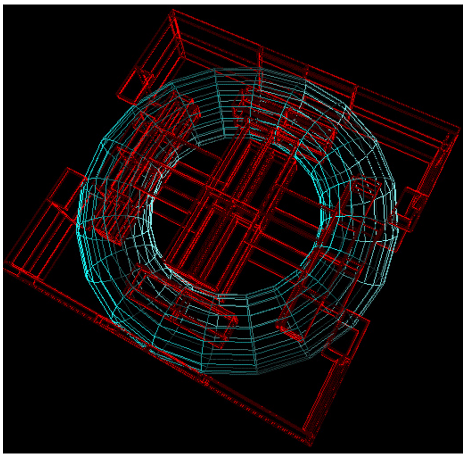


Figure 3.22: This shows all the empty boxes of the BoxTree for a torus. With common complexities, 40–60% of the bounding box' volume are covered by empty boxes, typically.

right sub-box, which is equivalent to computing two new sets of $2 \times (3 \times 4)$ line parameter intervals, one describing (a, b_l), the other describing (a, b_r).

This seems like a lot of computational work; however, half of the information stored in a set of intervals for (a, b) can be re-used for (a, b_l), the other half for (a, b_r) (see Figures 3.20 and 3.21).

Parallel edges and polygons Although rare in real-world scenarios, parallelism must be dealt with in order to achieve a robust algorithm.

Like many other properties or values, parallelism is preserved during simultaneous traversal through the BoxTree. Again, the special geometry of boxes reduces the number of significant edge-face comparisons: if an edge is parallel to a face, then all edges of the same family are parallel to all faces of the other family.

During bisecting a box, we might discover that an edge is parallel to a plane, by which we were about to clip it. Two things could happen: either, the edge is on the "wrong" side of the plane, in which case the interval will be empty and we are finished; or, the edge is on the "right" side, in which case we just proceed to the next plane.

Timing

For timing tests I chose the following scenario: two objects move inside a "cage". Initial positions and translational and rotational velocities are chosen randomly at start-time. When the two objects collide, they bounce off each other based on simple heuristics (e.g., by exchanging translational and/or rotational velocities). The size of the cage is chosen so as to "simulate" a dense environment, i.e., most of the time there are only "almost-collisions", which is the "bad" case for most algorithms. Also, this excludes any side-effects, e.g., by bounding box checks. The test objects were regular ones, like spheres, tori, tetra-flakes, etc., and real-world data (e.g., an alternator). Rendering is always switched off, of course.

Figure 3.27 shows a comparison between the BoxTree algorithm (using optimal parameters for the tree construction as determined in Section 3.5.7) and the

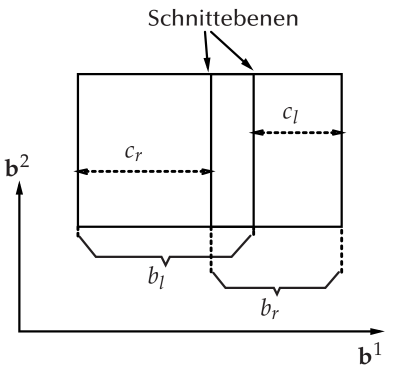


Figure 3.23: Each (inner) node in the tree needs to store only one short integer (denoting one of the three axes which the two cutting planes are perpendicular to), and two reals c_f, c_r, one for each cutting plane.

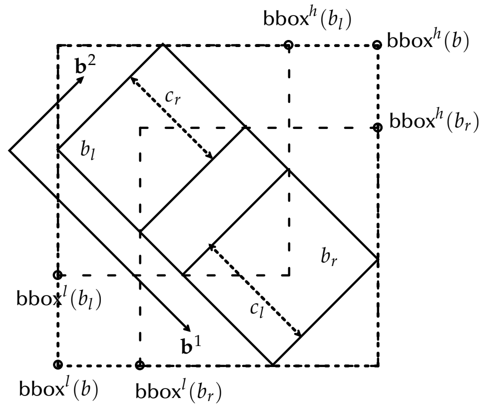


Figure 3.24: The aligned boxes enclosing the two children B_f and B_r share 3 of their min/max values each with the aligned box enclosing B.

simpler algorithm as described in Section 3.3. The same scenario as above was used. Each sample is an average over 20×2000 frames. The tests were run on an 200 MHz R4400. The figure was obtained with two tori, but similar results have been obtained for all other object types.

As expected, BoxTrees are much faster when object complexity is above a certain threshold, but slower for small objects. The threshold (for tori) is about 100 polygons, below which a simple algorithm out-performs the sophisticated one.

3.5.6 BoxTree traversal by re-alignment

Bounding volume hierarchies are always constructed in object space. When objects move, the BVs need to be transformed as well, conceptually. The problem of developing fast hierarchical collision detection can be regarded as finding BVs which are

1. invariant under rigid motions,
2. fast to test for overlap, and
3. tight.

These conditions seem to be contradictory: either they are tight, or they can be tested quickly; and, if they are fast to test for overlap, then they are usually not invariant under rigid motions.

In the previous section, I have described an overlap test for boxes which are not aligned in world-space, but which are all aligned in object-space. The test gains some speed from certain constraints on the arrangement of child boxes with respect to their father.

Still, testing non-axis-aligned boxes for overlap is expensive compared to the test for axis-aligned boxes. Actually, axis-aligned bounding boxes offer probably the fastest overlap test among all bounding volumes. So, instead of testing the boxes of Box-Trees directly, they can be enclosed by an axis-aligned box. This can be done very fast, which will be described in the next subsection.

One recursion step

We must be able to compute the axis-aligned boxes fast, otherwise we do not gain speed in the overall overlap test. In this section, we can allow $\text{bbox}^h_\alpha(a_l) \neq \text{bbox}^l_\alpha(a_r)$ (see Section 3.5.5), because (1) we would not gain anything by keeping it, and (2) this relieves us of the burden of handling "crossing" polygons. For the sake of simplicity, I will denote those values by c_l and c_r, resp. Note that here, $\text{bbox}^h_\alpha(a_l)$, and $\text{bbox}^l_\alpha(a_r)$ denote offsets for the two cutting planes from the *opposite* sides of the father box (see Figure 3.23)!

Let us assume we know that the two axis-aligned boxes enclosing a and b do overlap. Let us assume further that both a and b have two children (without loss of generality).

Now we want to find out whether or not the four pairs of sub-boxes, (a_l, b_l), (a_l, b_r), (a_r, b_l), (a_r, b_r), overlap. In a naive approach, one would compute the corners of each non-axis-aligned box $a_l, \ldots, b_r$, take the min/max of each, and then do an axis-aligned box-box test. However, this is too expensive, because it would discard all information gathered so far.

Notice that, by way of the special construction of Box-Trees, only 3 of the min/max values of each sub-box have to be computed — the other 3 can be copied from the "father"-box.

Let $\mathbf{b}^1$, $\mathbf{b}^2$, $\mathbf{b}^3$ be the coordinate frame of box b in common (world-)space (see Figure 3.24). Assume that the cutting planes are perpendicular to $\mathbf{b}^1$, without loss of generality. Then, the x-coordinate min/max values of the axis-aligned box enclosing b_l are

$$\text{bbox}^h_x(b_l) = \begin{cases} \text{bbox}^h_x(b) - c_l \mathbf{b}^1_x & , \quad \mathbf{b}^1_x > 0 \\ \text{bbox}^h_x(b) & , \quad \mathbf{b}^1_x \leq 0 \end{cases}$$

and

$$\text{bbox}^l_x(b_l) = \begin{cases} \text{bbox}^l_x(b) & , \quad \mathbf{b}^1_x > 0 \\ \text{bbox}^l_x(b) - c_l b^1_x & , \quad \mathbf{b}^1_x \leq 0 \end{cases}$$

Similarly, the x-coordinate min/max values of the axis-aligned box enclosing b_r are

$$\text{bbox}^h_x(b_r) = \begin{cases} \text{bbox}^h_x(b) & , \quad \mathbf{b}^1_x > 0 \\ \text{bbox}^h_x(b) + c_r \mathbf{b}^1_x & , \quad \mathbf{b}^1_x \leq 0 \end{cases}$$

and

$$\text{bbox}^l_x(b_r) = \begin{cases} \text{bbox}^l_x(b) + c_r \mathbf{b}^1_x & , \quad \mathbf{b}^1_x > 0 \\ \text{bbox}^l_x(b) & , \quad \mathbf{b}^1_x \leq 0 \end{cases}$$

Quite analogously the y-coordinate values and the z-coordinate values of the axis-aligned bounding box can be computed. Although Figure 3.24 is 2D, it is easy to verify that the formulas above hold in 3D as well.

We will not only save half of the calculations for the aligned boxes, but also half of the comparisons of the overlap tests of aligned boxes. A naive aligned box overlap test would compare 6 coordinates — however, notice that we need to check only 3 of them, since the status of the other 3 has not changed. So again, we re-use information from the recursion step before.

More precisely, assume we know that $\text{bbox}(b)$ and $\text{bbox}(a)$ overlap. Now we want to know whether or not $\text{bbox}(b_l)$ and $\text{bbox}(a)$ overlap. Since only 3 values of $\text{bbox}(b_l)$ differ from $\text{bbox}(b)$, we need to compare only those with $\text{bbox}(a)$. For example, the x-coordinate comparison is

$$\left.\begin{array}{l} \mathbf{b}^1_x > 0 \ \wedge \ \text{bbox}^h_x(b_l) < \text{bbox}^l_x(a) \\ \mathbf{b}^1_x \leq 0 \ \wedge \ \text{bbox}^l_x(b_l) > \text{bbox}^h_x(a) \end{array}\right\} \ \Rightarrow \ \text{bbox}(b_l) \text{ and } \text{bbox}(a) \text{ do } \textit{not} \text{ overlap}$$

(again, assuming that the cutting plane is perpendicular to $\mathbf{b}^1$). Note that the decision $\mathbf{b}^1_x \lesseqgtr 0$ has been made already when computing $\text{bbox}^h_x(b_l)$ or $\text{bbox}^l_x(b_l)$, respectively.

Summarizing, the number of floating point operations for one box-box overlap test takes at most 1.5 multiplications, 2 additions, and 2.5 comparisons.[10]

3.5.7 Constructing the BoxTree

Since the BoxTrees in the previous section and Section 3.5.5 are very similar, the same algorithm and (almost) the same heuristics can be applied.

In addition to the criteria set forth in Section 3.5.2, the following have to be considered as well to guide the partitioning:

- Due to the constraints of the boxes with respect to their father, nodes of a BoxTree (except the root) do not necessarily bound the associated set of polygons tightly. Therefore, it is sometimes better not to divide the set of polygons at all, but pass it on to one of the two sub-nodes, which can bound it more tightly, while the other sub-node remains empty. This can be viewed as "splitting off" an empty sub-box. (see Figure 3.22).

- For the traversal algorithm of Section 3.5.5: the number of "crossing" polygons should be minimized. This does not apply for the algorithm of Section 3.5.6.

Given a set of polygons which is completely inside a bounding box, the algorithm determines the largest possible *empty* sub-box. If it is larger than a certain, pre-defined threshold, then it "splits off" that empty box. Otherwise, it considers each of the three splitting axis orientations and determines the optimal partitioning of the set of polygons along that axis. Then it chooses the one which produces the best partitioning among all three possibilities.

Given a set of polygons F and a certain splitting axis orientation, how do we find quickly two "optimal" sub-sets of polygons $F^l, F^r \subseteq F$ such that $F^f \dot\cup F^r = F$ and the criteria above are met as close as possible? I have taken a "greedy" approach: the algorithm starts with two empty sets F^l and F^r with associated empty sub-boxes. It considers each polygon in F and puts it into F^l or F^r whichever sub-box will be extended by the least amount. If neither of the two sub-boxes would be extended, or if both would be extended by the same amount, then it puts the polygon into the smaller set.

Exactly how the optimal splitting axis is determined will be described below in Section 3.5.7. First, I will derive the complexity of this algorithm. (It does not depend on the method how the splitting axis is chosen.)

[10] The number of operations per recursion step is 6 multiplications, 8 additions, and 10 comparisons. (3×2mult. $+ 3 \times 2$ add. for enclosing the two children of a box by an axis-parallel box, and $2 \times 3 + 2 \times 2$comp. for the overlap tests.) This is for the worst case: both boxes have two children each, and all four pairs of those children overlap.

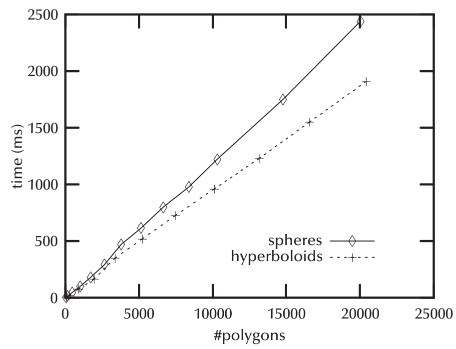


Figure 3.25: Experiments confirm that building boxtrees is in $O(n)$ average running time, and that the hidden constant is small enough for practical purposes. The graph shows timings for building the BoxTree for spheres and hyperboloids. Timing was done on a 200 MHz R4400.

Complexity

This partitioning algorithm is very fast, because it involves only bounding box comparisons.

Under certain assumptions the complexity of constructing a BoxTree is in $O(n)$, where n is the number of polygons. This is supported by experiments (see Figure 3.25).

Let us assume that cutting a box takes a constant number of passes over all polygons associated with that box. Every cut will split the box's polygons into 2 sets F_1, F_2, with $|F_1| + |F_2| = |F| = n$. Let us assume further w.l.o.g. that $|F_1| \leq |F_2| \leq \alpha n$, with $\frac{1}{2} \leq \alpha < 1$. So, for depth d of a BoxTree $n = (\frac{1}{\alpha})^d$.

Let $T(n)$ be the time needed to build a BoxTree for n polygons. Then,

$$
\begin{aligned}
T(n) &= cn + T(\alpha n) + T((1-\alpha)n) \\
&\leq c \sum_0^d 2^i \alpha^i \\
&\leq c\alpha 2^{d+1} \\
\Rightarrow T(n) &\in O(n)
\end{aligned}
$$

Geometrical robustness

This issue is of great importance (as I learnt the hard way). This is especially true for polygonal objects which are computer-generated and expose a high degree of symmetry, like spheres, tori, extruded and revolved objects, etc. These objects usually have very good splitting planes, but if the splitting routine is not robust, the BoxTree will not be balanced at all.

The problem is: when do we consider a polygon to be on the left, the right, or on both sides of a plane? Because of numerical inconsistencies, many polygons might be classified "crossing" even though they only *touch* the plane (see Figure 3.26). The idea is simply to give the plane a certain "thickness" 2δ. Then, we will still consider a polygon left of a plane c, even if one of its edges is right of c, but left of $c + \delta$. All the possible cases are depicted in Figure 3.26.

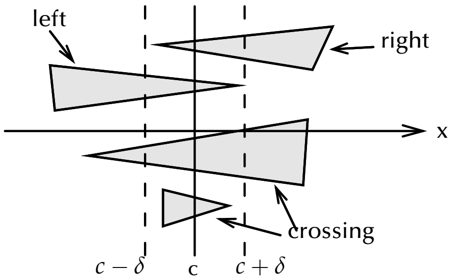


Figure 3.26: For splitting a set of polygons by a plane, geometrical robustness can be achieved by giving the plane a certain “thickness”.

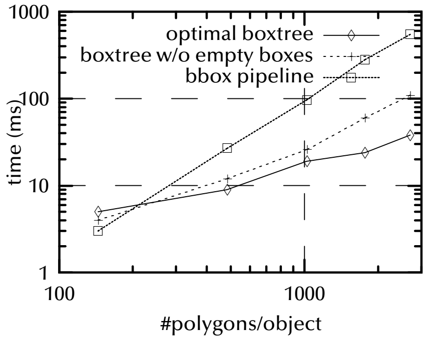


Figure 3.27: Comparison of the boxtree algorithm with the non-hierarchical algorithm of Section 3.3. Other object types (sphere and tetra-flake) yielded similar results with slightly different thresholds.

Optimal BoxTree parameters

Before any timing can be reasonable, *optimal parameters* for BV tree construction have to be determined. In this case, the question is: when should an empty box be split off?

To answer this, I ran several tests with different objects and different choices of those parameters. Fortunately, the “near-optimal” range for this parameter seems to be fairly broad. I also checked experimentally that empty boxes do actually yield some speed-up (see Figure 3.27).

It also turned out (fortunately), that optimal BoxTree parameters do not depend much on the type of the object.

Optimal criteria

As outlined above, for each of the three axes the set of polygons is split into two possible subsets. The task then is to determine which axis (i.e., which split) is the best. The split criterion is basically a penalty function $f(x|y|z,\ldots)$ which takes the size of the box to be split, the sizes of the candidate sub-boxes, and the corresponding numbers of polygons (or any other variables). So, the best axis is α with $f(\alpha,\ldots) = \min_{x|y|z}\{f(x|y|z,\ldots)\}$ (some criteria realize the maximum).

Various reasonable criteria come to mind. However, it is not obvious which one yields the best BoxTree in terms of collision detection time. In order to choose the best criterion, I have benchmarked 14 different criteria by the following procedure. Each criterion was applied to a suite of 19 test objects (sphere, torus, cone, hyperboloid, and 15 automotive objects), which ranged from 5000 polygons to 100,000 polygons. With each criterion and each object the average collision detection performance was determined by the benchmark procedure described in Section 3.5.10.

Let t_{ko} be the average collision detection time with criterion k applied to object o. Then the relative performance is $t'_{ko} = t_{ko}/t_{2o}$ (w.l.o.g., criterion 2 was chosen for no particular reason). The relative performance of each criterion is shown in Figure 3.28. The average performance $t_k = \frac{1}{n}\sum t_{ko}$ renders a similar

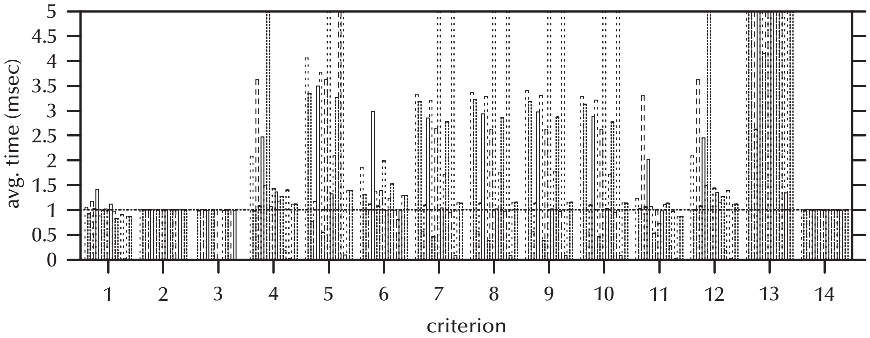


Figure 3.28: Relative performance of the splitting criteria with respect to collision detection time.

picture (see Figure 3.29). If one defines the *rank* of a criterion as $r_k = \frac{1}{n}\sum t'_{ko}$, we get the following table:

criterion	1	2	3	4	5	6	7
rank	0.89	1.00	0.99	1.91	3.37	1.62	2.51

criterion	8	9	10	11	12	13	14
rank	2.54	2.54	2.50	1.09	1.91	198.7	1.00

Let $(\mathbf{b}^l, \mathbf{b}^h)$ be the box to be split; let $d = \mathbf{b}^l - \mathbf{b}^h$; let $d' = \mathbf{b}'^l - \mathbf{b}'^h$, where $(\mathbf{b}'^l, \mathbf{b}'^h)$ is the bounding box of the polygons associated with the box to be split (remember that $(\mathbf{b}'^l, \mathbf{b}'^h)$ can be smaller than $(\mathbf{b}^l, \mathbf{b}^h)$); let $(\mathbf{c}^{l\,1|2\,x|y|z}, \mathbf{c}^{h\,1|2\,x|y|z})$ be the 3 candidate sub-box pairs. The 14 penalty functions were:

1. $f(\alpha) = \mathbf{c}_\alpha^{h\,1\,\alpha} - \mathbf{c}_\alpha^{l\,2\,\alpha} - d'_\alpha$

   Minimize the overlap of the two sub-boxes; take into account the extent of the parent-box.

2. $f(\alpha) = \mathbf{c}^{h\,1\,\alpha} * \mathbf{c}^{l\,2\,\alpha}$

   Minimize the volume of the overlap of the two sub-boxes.

3. $f(\alpha) = \frac{\mathbf{c}_\alpha^{h\,1\,\alpha} - \mathbf{c}_\alpha^{l\,2\,\alpha}}{d'_\alpha}$

   Minimize the overlap of the two sub-boxes relative to the extent of the parent-box (this is a variant of criterion 1).

4. $f^*(\alpha) = d'_\alpha$

   Try to generate "cube-like" boxes.

5. $f(\alpha) = |n^{1\,\alpha} - n^{2\,\alpha}|$

   Minimize the unbalancedness in terms of polygon count.

6. $f(\alpha) = \mathbf{c}_\alpha^{h\,1\,\alpha} - \mathbf{c}_\alpha^{l\,2\,\alpha}$

   Minimize the overlap of the two sub-boxes; do *not* take into account the extent of the parent-box (this is a variant of criterion 1).

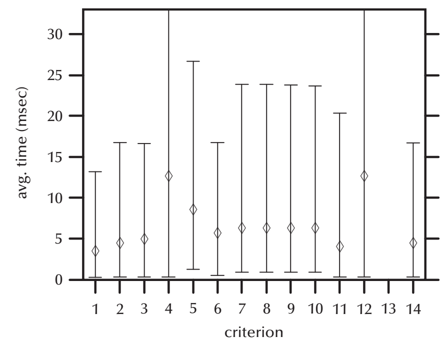


Figure 3.29: Average performance of the splitting criteria. Criterion 1 creates the best BoxTrees.

7. $f(\alpha) = (f^5(\alpha), f^1(\alpha))$

   Combination of criteria 5 and 1: first minimize unbalancedness, then, if both sub-boxes are perfectly balanced, try to minimize overlap.

8. $f(\alpha) = (f^5(\alpha), -d'_\alpha)$

   Combination of criteria: first minimize unbalancedness, then try to create "cube-like" boxes.

9. like 8, but with d instead of d'

10. like 7, but with d instead of d'

11. $f(\alpha) = \text{recursion depth} \mod 3$

    This is a "bogus" criterion. It serves as a counter-check for the others.

13. $f^*(\alpha) = \mathbf{c}_\alpha^{h\,1\,\alpha} - \mathbf{c}_\alpha^{l\,2\,\alpha}$

    This is like criterion 6, except the penalty function is maximized instead minimized.

The starred functions are maximized, all others are minimized.

3.5.8 Oriented boxes

Axis-aligned boxes are simple bounding volumes but sometimes they cannot approximate objects or polygons very well. Especially for objects with a strong diagonal orientation, the volume of axis-aligned boxes contains a lot of "dead space" (see Section 3.5.3). Therefore, it is natural to look at *oriented bounding boxes* (OBB), i.e., boxes with unconstrained orientation.

Bounding volume hierarchies consisting of oriented boxes place no constraints on the orientation or position of the BVs, except that they enclose their associated set of polygons. So, it might happen that child boxes are not completely enclosed by their father box (see Figure 3.30).

Basically, two issues must be solved with OBBs: (1) a fast overlap test, and (2) construction of the optimal OBB for a given set of polygons. With top-down construction of OBB trees, a third issue is: given a set of polygons, how do we split it such that the optimal OBBs for both subsets yield an optimal OBB tree?

In [GLM96] Gottschalk et al. have given a neat solution for (1) and an algorithm for (2) and (3), for which they also provide experimental evidence that it is at least near-optimal.

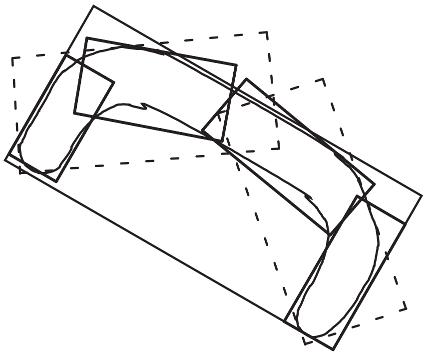


Figure 3.30: The OBB tree is a bounding box hierarchy where each box is placed such that it is (almost) minimal.

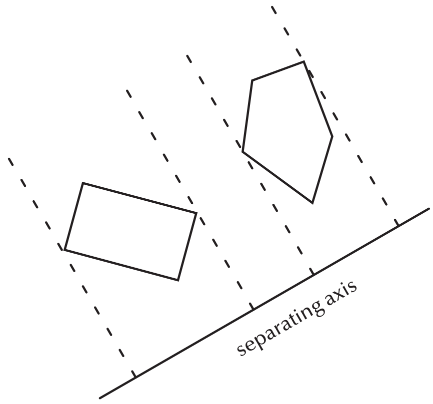


Figure 3.31: A fast overlap test for oriented boxes is based on the separating axis theorem.

Overlap test. The basic idea is the *separating axis theorem*: Given two convex objects. These objects do not overlap if and only if there is a line between both objects, such that the projection of the two objects onto that line yields two disjoint intervals. Such a line is called a separating axis (see Figure 3.31).

The proof is quite simple. If there is such a line, then we can find trivially a separating plane. If the two objects do not overlap, then there is a separating plane and the line perpendicular to that plane is a separating axis.

We know that if there is a separating plane, then there is also a separating plane, which is parallel to one of the faces of one of the object, or which is parallel to one edge of each object. So, for boxes there are only 15 potential separating lines to be tested.

Each test amounts to evaluation of a certain inequality involving the transformation matrices of both objects, the box orientations, and the box radii. If the number of possible box orientations and radii is small, then many terms of the inequality can be precomputed [HK97].

Tight fitting OBBs. The minimum-volume enclosing OBB for a set of n points can be computed in $O(n^3)$ [O'R85]. However, this is not practical for large objects (in terms of vertices).

An approximation of the minimum-volume enclosing box can be found by computing the covariance matrix of a set of triangles, with each triangle weighted by its area. Two of the eigenvectors of that matrix are the axes of the minimum and maximum variance. So, taking the eigenvectors as the axes of the OBB will align it with the geometry.

Constructing OBB trees. Given a set of polygons, [GLM96] compute the axes of the OBB, centered at the mean of the vertices. The set of polygons is split in two halves based on their centers, compared to the mean along the longest OBB axis. This yields balanced trees. However, my experiments indicate that balanced trees are usually *less* than near-optimal.

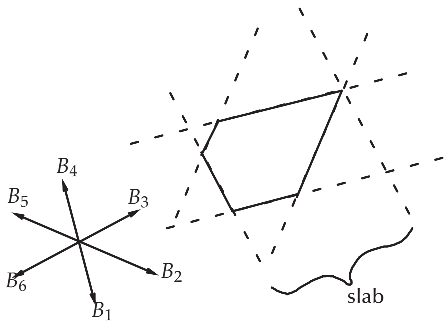


Figure 3.32: k-DOPs can be viewed as intersection of $k/2$ slabs.

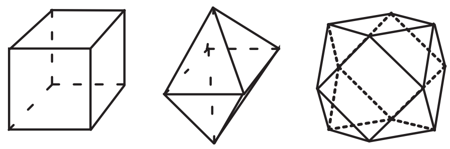


Figure 3.33: Three examples of DOPs: box, octahedron, and the unification of both.

3.5.9 Discretely oriented polytopes

Boxes and spheres are "shape-specific" bounding volumes in that there are certain shapes which they can bound tightly, while there are other types of shapes which they bound inherently badly.

The following generalization of boxes can overcome that problem. This type of BV has been used before by [KK86] for ray queries in ray-tracing. BV trees with this type of bounding volume have been used for collision detection before by [HKM96]. However, they seem to use a more general kind and to do hill-climbing to compute the axis-aligned DOPs from the "tumbled" ones, which is probably less efficient than my method [Zac98b].

Definitions

Discrete orientation polytopes (DOPs) are convex polytopes whose faces can have only normals which come from a fixed small set $\mathcal{B}$ of k orientations (hence k-DOPs). Probably the fastest overlap check for axis-aligned boxes is the well-known interval test. In order to be able to apply such a test to DOPs, we further restrict the set of orientations such that for each orientation of the set there is also an anti-parallel one;[11] the planes supported by two corresponding faces form what is commonly known as a *slab* (see Figure 3.32). Additionally, each plane must not be *redundant*, otherwise the overlap test based on interval-tests can return wrong answers. A plane is redundant when it does not pass through any of the vertices of the convex hull of the DOP.[12]

This special kind of k-DOPs can be viewed as a generalization of axis-aligned boxes. By increasing k, DOPs can approximate the convex hull of objects arbitrarily close (see Figure 3.33 for three examples of DOPs).

Being the intersection of k half-spaces

$$H_i : \mathbf{B}_i x - d_i \leq 0, \quad 0 \leq i < k$$

a k-DOP can be represented by the point $\mathbf{d} = (d_0, \ldots, d_{k-1}) \in \mathbb{R}^k$, where $\mathbf{B}_i$ are

[11] In order to make a DOP overlap test as fast as possible on average in the case of non-overlapping DOPs, I arrange the orientations in a list such that orientation B_i is "as perpendicular" as possible to all previous orientations $B_0, \ldots, B_{i-1}$.

[12] If the plane passes through exactly 1 or 2 vertices, then it does not contribute new vertices to the hull by itself. Still, we will not call such a plane redundant, because a stricter definition is not necessary.

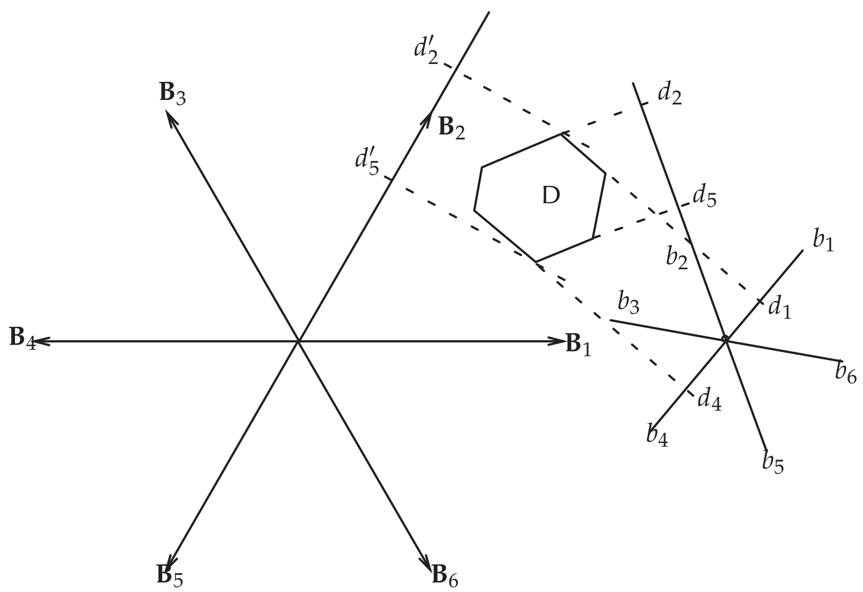


Figure 3.34: A rotated DOP can be enclosed by an aligned one by computing new plane offsets d'_i. Each d'_i can be computed by an affine combination of 3 $d_{j^i_l}$, $1 \leq l \leq 3$ (2 in 2-space). The correspondence j^i_l depends only on the affine transformation of the associated object and the fixed orientations $\mathbf{B}_i$.

the k fixed orientations. We will call $\mathbf{d}$ the *plane offsets* for that DOP.

So, at each node of a DOP tree we need to store only k floating point numbers (since the orientations are the same for all DOPs) plus 2 pointers (assuming binary trees).

Aligning DOPs

Given two DOP-trees O and Q, the basic step of the simultaneous traversal is an overlap test of two nodes. In order to apply the simple and very fast interval overlap test to DOP-trees, they must be given in the same space. However, at least one of the associated objects has been transformed by a rigid motion, so the DOPs of its DOP-tree are "tumbled" — in fact, in any other than the object's coordinate system the DOPs are no longer DOPs in the strict sense.

The idea is to enclose a tumbled DOP by another, "axis-aligned" DOP. I call this process *(re-)aligning*. The re-aligned DOP is, of course, less tight than the original one. On the other hand, the overlap test between two aligned DOPs is much faster than between non-aligned ones.[13]

By choosing O's object space (w.l.o.g.), we need to re-align only Q's nodes as we encounter them during traversal. I will show that this can be done by a simple affine transformation of the DOP's plane offsets. I would like to remark that, except for the interval overlap test of DOPs, the alignment algorithm works in the case of general DOPs as well.

Assume we are given a (non-aligned) DOP D of Q's DOP-tree, which is represented by $\mathbf{d}$. Assume also, that the object associated with Q has been trans-

[13] Of course, an incremental hill-climbing overlap test, which saves closest features, could be applied to non-aligned DOPs. However, this would incur a lot of additional "baggage" in the data structures. In fact, [HKM96] have reported that it is still less efficient than brute-force re-alignment.

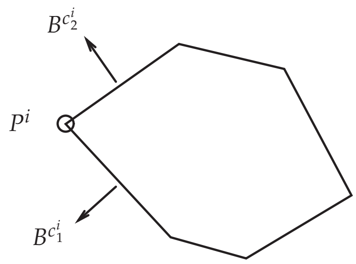


Figure 3.35: The correspondence between coefficients of tumbled DOPs and enclosing aligned DOPs is established in two steps at initialization time and at the beginning of a traversal.

formed by a rotation M and a translation $\mathbf{o}$, with respect to O's reference frame. Then D is the intersection of k half-spaces

$$h_i : \mathbf{b}_i\mathbf{x} - d_i + \mathbf{b}_i\mathbf{o} \le 0,$$

where $\mathbf{b}_i = B_i M^{-1}$ (see Figure 3.34).

Now suppose we want to compute the d'_i of the enclosing DOP D' of D. There is (at least) one extremal vertex P_i of the convex hull of D with respect to $\mathbf{B}_i$. This vertex is the intersection of 3 (or more) half-spaces $h_{j_l^i}$, $1 \le l \le 3$. It is easy to see that

$$d'_i = \mathbf{B}_i \begin{pmatrix} \mathbf{b}_{j_1^i} \\ \mathbf{b}_{j_2^i} \\ \mathbf{b}_{j_3^i} \end{pmatrix}^{-1} \begin{pmatrix} d_{j_1^i} \\ d_{j_2^i} \\ d_{j_3^i} \end{pmatrix} + \mathbf{B}_i\mathbf{o} \qquad (3.2)$$

The correspondence established by j_l^i is the same for all (non-aligned) DOPs of the whole tree, if the following condition is met: DOPs must not possess any *completely redundant* half-spaces, i.e., all planes must be supported by at least one vertex of the convex hull of the DOP. (We do allow *almost redundant* half-spaces, i.e., planes which are supported by only a single vertex of the convex hull.) Fortunately, this condition is trivially met when constructing the DOP-tree.

The correspondence j_l^i is established in two steps: First, we compute the vertices of a "generic" DOP, constructed such that each vertex is supported by three planes (i.e., no degenerate vertices). In an intermediate correspondence c we store with each vertex P^i the three orientations $\mathbf{B}_{c_1^i}, \ldots, \mathbf{B}_{c_3^i}$ of the three supporting planes (see Figure 3.35).[14] In the second step, another correspondence is calculated telling which P^i does actually support a plane of the new axis-aligned DOP D' (not all P's will do that). The first and the second correspondence together yield the overall correspondence j.

The first intermediate correspondence c has to be computed only once at initialization time,[15] so a brute-force algorithm can be used. The second intermediate one has to be computed whenever one of the objects has been rotated, but it is fairly easy to establish: at the beginning of each DOP-tree traversal, we transform the vertices of a generic DOP (see below) by the object's rotation. Then, we combine these to establish the final correspondence j_l^i.

[14] Because of my loose definition of *redundancy* for planes, it could happen that more than 3 planes pass through one point in space. This is no problem, though, because this just means that several vertices of the DOP will be coincident. Each of them corresponds to exactly three planes.

[15] I chose not to hard-code it, so I could experiment with different sets of orientations.

To establish the intermediate correspondence c, we can choose any DOP satisfying the additional condition that all planes do support a non-degenerate face (i.e., all planes are non-redundant in the strict sense). I construct such a DOP in the following way: Start with the unit DOP $\mathbf{d} = (1, \ldots, 1)$. Then check that each plane satisfies the condition. If there is a plane which does not, increase its plane-offset. This algorithm should be made probabilistic so as to avoid cycles. Of course, it could still run into a cycle, but this has not happened so far in countless runs.

Brute-force alignment

There is another way to realign DOPs [KHM+98]. The idea is to represent DOPs by their vertices. Then, an aligned enclosing DOP of a tumbled DOP can be found trivially by transforming the vertices of the tumbled DOP and then computing the min/max of all vertices along each orientation.

Enclosing a non-aligned DOP by an aligned one takes

	affine transformation	vertex transformation
FLOPs	$6k$	$6k^2 + 17k$

where multiplications, additions, and comparisons count equal (the constant terms have been omitted). Both methods represent the resulting aligned DOP in "slab form". For the estimation of the brute-force method, I have assumed that DOPs consist only of triangles.

Generalization

So far, I have silently assumed that for both objects we use the same set of orientations. Futhermore, I have assumed that the set of orientations used for constructing the re-aligned DOPs is the same as that used for constructing the DOP-trees in object space.

Both assumptions are there just to keep things simple.

The DOP-trees of the two objects can be built using completely different sets of orientations, because when checking for overlap they will really be enclosed by DOPs constructed with yet another set of orientations.

When constructing the DOP enclosing a tumbled one, they do not necessarily have to share the same set of orientations. Actually, they do not even need to have the same number of orientations. The knowledge of *which* local-space orientations are mapped to aligned orientations is entirely encapsulated in the correspondence j_l^i. It could happen that some local orientations do not even occur in the correspondence (if the number of world-space orientations is much smaller than the number of local-space orientations).

This generalization adds more freedom to the construction of DOP-trees. However, it also adds more parameters to the optimization of them. And it is not yet clear to me, how they can be determined efficiently (without trial-and-error).

Building DOP-Trees

For virtual prototyping applications it is important, that the construction can be done at load time. Otherwise, this would be another preprocessing step which had to be done in order to prepare a VE. The acceptance for any preprocessing steps is extremely low in the manufacturing industries (probably in all industries). Another problem is, that each additional class of files needed for the specification of VEs causes some serious hassle with the product data management systems (and their maintainers).

Therefore, the construction must be fast. So, we make use of several heuristics and estimations which try to emulate the criteria listed in Section 3.5.2.

The input to the construction algorithm is a set $\mathcal{F}$ of k-DOPs (each of them encloses one of the polygons of the object) and a set $\mathcal{C}$ of points which are the barycenters of the polygons.

First, the algorithm finds $c_i, c_j \in \mathcal{C}$ with almost maximal distance.[16] Then it determines that orientation which is "most parallel" to $\overline{c_i c_j}$. Now we sort $\mathcal{C}$ along that orientation, which induces a sorting on $\mathcal{F}$.

After these preliminaries (which are done for each step of the recursion), we can split $\mathcal{F}$ in two parts $\mathcal{F}_1$ and $\mathcal{F}_2$. We start with $\mathcal{F}_1 = f_i$, $\mathcal{F}_2 = f_j$, where f_i, f_j are associated to c_i, c_j, resp. Then, we consider all other $f \in \mathcal{F}$ in turn and assign them to $\mathcal{F}_1$ or $\mathcal{F}_2$, whichever BV increases less in volume. If both BVs of $\mathcal{F}_1$ and $\mathcal{F}_2$ would increase by the same amount (in particular, if they would not increase at all), then f is added to the set which has fewer polygons so far.

It can happen, that one of the sets gets all the polygons, because its DOP grows creepingly. This happens particularly, because polygons tend to come "sorted" in some way. To avoid this, we take candidates f from $\mathcal{F}$ alternatingly from its lower and its upper end.

Computing the volume of a DOP is not trivial, since we are given only the plane offsets $\mathbf{d}$. If we had the vertices, then the volume would be the sum of the volumes of the tetrahedra formed with an inner point. There are also estimations for the volume based on the number of vertices, edges, and faces, such as

$$\begin{aligned} &\frac{1}{3} e \sin(\frac{f}{e}\pi)[\tan^2(\frac{f}{e}\frac{\pi}{2})\tan^2(\frac{v}{e}\frac{\pi}{2}) - 1] r^3 \\ &\qquad \le V \le \\ &\qquad\qquad \frac{2}{3} e \cos^2(\frac{f}{e}\pi)\cot(\frac{v}{e}\frac{\pi}{2})[1 - \cot^2(\frac{f}{e}\frac{\pi}{2})\cot^2(\frac{v}{e}\frac{\pi}{2})] R^3 \end{aligned}$$

with f, e, v the number of faces, edges, and vertices, resp., and r, R the radius of the insphere and circumsphere, resp. Since these numbers would be the same for all DOPs in almost all practical cases (no redundant planes, all vertices with degree 3), this would be an interesting option, if I had a way to calculate or estimate the radius R of the circumsphere (the radius r of the insphere is the minimal distance between slabs).

Other heuristics have been implemented and tested, but the one described above has produced the best results.

Therefore, I tried several simple estimations of the volume or the increase in volume for DOPs. The first one is a simple generalization of the volume of boxes:

$$\text{vol} = \prod_{i=0}^{k/2} \delta_i$$

[16] This is an $O(n^2)$ problem. There are algorithms with better complexity, but they are quite involved, and in my opinion, the optimal pair is not really necessary. So, I compute only a near-optimal pair by a simple $O(n)$ heuristic.

depth	10	11	12	13	14	15	16
sphere	0	0	1	375	8853	13233	642
car door	92	740	1639	659	162	42	26

Table 3.2: Histogram of the depth of the leaves of the 6-DOP tree. The sphere has approximately 20,000 polygons, while the car door has approximately 3,300 polygons.

where δ_i is the space between slab i. For boxes, this is the true volume, for larger k, the volume will be overestimated. I also tried to use an estimate of the increase in volume

$$\Delta\text{vol} = \prod_{i=0}^{k/2} \delta'_i$$

where δ'_i is the increase of d_i or 1 if there is none in the direction i. Instead of the product I also tried the sum. Finally, I tried an even simpler estimate

$$\Delta\text{vol} = \delta''_{l_1} + \delta''_{l_2}$$

where δ'' is the increase of d_i (or 0) and l_1, l_2 is the "most parallel orientation". This last estimate produced the best results so far.

Overall, the algorithm seems to produce good trees. In particular, they are fairly well balanced. Therefore, the average depth is almost the same for all sets of orientations (which I have verified by experiments). Table 3.2 shows a histogram of the depth of the leaves of the 6-DOP tree for two objects. Other heuristics have been implemented and tested, but the one described above has produced the best results.

Geometric robustness and accuracy

The construction algorithm is geometrically robust and can be applied to all unstructured models. No adjacency information is required. There are no connectivity restrictions and the faces can be degenerate (a line segment or a point), which happens frequently in CAD data.

The overlap test is very robust, since it involves only multiplications, additions, and comparisons. A small ϵ-margin guards against arithmetic round-off errors. This also adds more robustness in the case of degenerate DOPs, where one or more planes are redundant or which have no volume. Actually, that margin can be applied to the DOPs while the DOP-tree is being constructed (inflating them a little), so during traversal, no ϵ-additions/subtractions need to be done. No error accumulation can occur during traversal of the tree.

The optimal number of orientations

Obviously, there are two contradictory effects when the number of orientations of DOP trees is increased: on the one hand, they can better approximate the convex hull of the set of polygons enclosed by them; on the other hand, an overlap test between them is more expensive.

Three different sets with 6, 8, and 14 orientations have been tested: the faces of 6-DOPs have the same normals as a cube, 8-DOPs have normals of an octahedron, and 14-DOPs have normals of the union of the former two. These DOPs will be called *standard DOPs*.

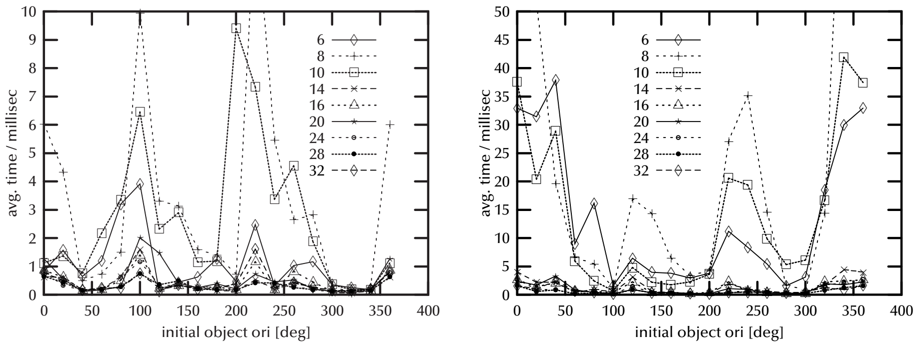


Figure 3.36: The effect of an object's coordinate frame on collision detection performance can be significant. Left: two car bodies (60,000 polygons each), right: door locks (43,000 polygons each). The plots have been generated by the benchmark procedure outlined in Section 3.5.10. The time shown is the collision detection time averaged over "interesting" distances. The locks are different in that most of the polygons are in the interior, because it has many parts on the inside. This might explain the "inversion" of the shape of the curve.

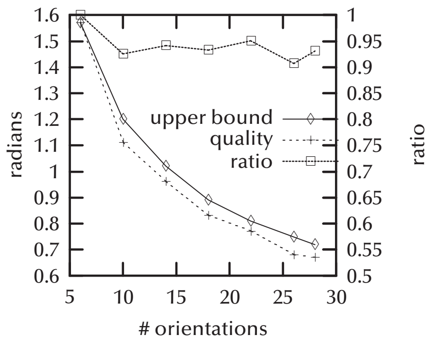


Figure 3.37: The quality of the DOPs generated by my simulated annealing process seems to be almost optimal.

Besides the three standard DOPs, I also investigated the whole range from $k = 6 \ldots 32$. These DOPs were created by the following simulated annealing process: k points are distributed randomly on the unit sphere, under the constraint, though, that for each point $\mathbf{p}$ the point $-\mathbf{p}$ is also in the set. This constraint is an invariant throughout the process. According to their distance, each point pushes off all others. All the forces are summed for each point (considering that $\mathbf{p}$ and $-\mathbf{p}$ are linked together by a rigid "bar"), which make the points move on the surface of the sphere. In order to reach an equilibrium, the repelling forces are weighted with a factor (the "temperature") which is decreased continuously. To my knowledge, no optimal "density" of packing points on a sphere is known yet [Slo98] (so there is no algorithm to construct an optimal one). The 6-, 8-, and 14-DOPs found by this algorithm are not necessarily "axis-aligned" in the sense that their orientations are parallel to one of the coordinate axes. This algorithm was motivated by the observation that parallel orientations are redundant, while orthogonal orientations are good.

I tried to estimate the *quality* of the DOPs generated by the above simulated annealing process. The following is but one way to measure a DOP's quality: a k-DOP is defined by a set of k points $\mathbf{p}_i$ on the unit sphere. Let

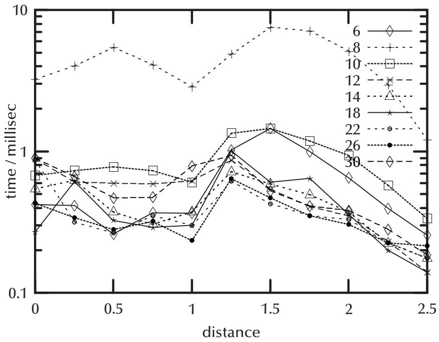


Figure 3.38: Collision detection time for two tori, each with 10,000 polygons. The distance is, as always, measured between their centers. Each graph corresponds to a particular k.

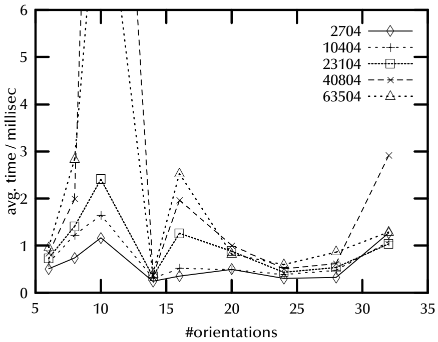


Figure 3.39: The same data as in the figure to the left (each graph corresponds to a certain distance). This graph provides some evidence that the optimal k does not depend on the distance between objects. For all other objects in the "suite", similar graphs were obtained.

$d = \min_{i \neq j}\{\arccos(p_i p_j)\}$. This is the smallest distance on the surface of the sphere between two points. An upper bound for this is $d_{\max} = 2\frac{2\pi}{n\sqrt{3}}$.[17] Figure 3.37 shows the quality of the DOPs obtained by simulated annealing compared to this upper bound.

Some geometry has unevenly distributed polygon normals (note that the "pipes" object, for instance, has a lot of polygon normals in one of the coordinate planes). Figure 3.36[18] show the effect of different object coordinate frames on collision detection performance: before the DOP-tree has been built, the object was rotated. One can conclude that 8-DOPs and 10-DOPs yield the worst performance; in particular, they seem to be most susceptible to "wrong" object orientations. However, it is not quite clear to me, why $k = 8, 10$ should perform so much worse than $k = 6$.

Figures 3.38, 3.39, and 3.40 show the effect of different numbers k of orientations in various ways. Although there is no k which is optimal for *all* objects and distances, it seems that $k = 24$ is the optimum. There seems to be no significant gain for larger k, and with some objects there is even a performance loss. In addition, all k seem to behave similarly with respect to the varying distances between the two objects.

---

[17] This upper bound can be derived as follows. In the plain, the densest packing of discs is achieved if the centers are arranged in a hexagonal lattice. With this packing, the ratio $v = \frac{\text{area covered by discs}}{\text{total area}} = \frac{\pi}{2\sqrt{3}}$ (this is the ratio of the area of a disc enclosed by a hexagon). So, for small discs on a sphere, the area covered by discs $A' \approx \frac{2\pi^2}{\sqrt{3}}$. Therefore, each disc can have at most radius $r_n = \sqrt{\frac{2\pi}{n\sqrt{3}}}$ ($n =$ number of discs on the sphere).

Since this upper bound is derived from planar geometry, it is not a tight bound for small n. However, for larger n it becomes fairly tight.

[18] All data for the plots in this section have been obtained on a 194 MHz R10000.

Similar plots for the complete test suite can be downloaded from `http://www.igd.fhg.de/~zach/coldet/index.html#dop-opt-ori`

From these figures it becomes clear why comparing collision detection algorithms is difficult: depending on the situation and the particular objects chosen for the tests, the collision query times can vary by an order of magnitude.

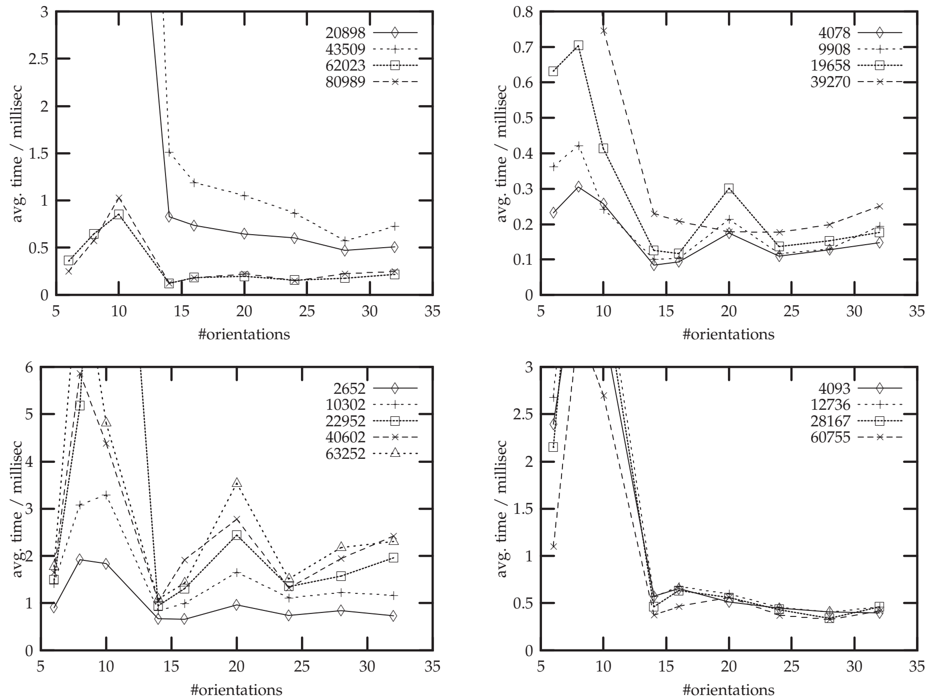


Figure 3.40: Average collision detection time for various objects of the "suite" depending on different k (from top left: door lock, hose, sphere, car body). The average was calculated over the "interesting" distance range and over all object coordinate frames. Each object has a "radius" of 1.

Figure 3.41 shows that collision detection time increases[19] only very slowly with complexity (provided the optimum number of orientations has been chosen).

I also compared the memory requirements and construction time among DOP-trees using 6, 8, or 14 orientations. As expected, construction time increases slightly as the number of orientations increases (see Table 3.3). Also, the amount of memory required increases slightly (see Table 3.4). This is due to the fact that the depth of trees does not change with different sets of orientations.

Comparison of DOP-trees and OBB-trees

In this section I look at the comparison of DOP-tree versus OBB-trees with respect to memory usage and tree construction time. An explanation of the benchmark procedure and a comparison with respect to collision detection time can be found in Section 3.5.10. I used 6-DOPs to compare DOP-trees with OBB-trees.

Table 3.5 shows the amount of memory required by DOP-trees and OBB-trees. For both, memory usage depends linearly on the number of polygons, of course. At each node, DOP-trees store only 6 floats whereas an OBB-node stores (at least) 18 floats. The table indicates that DOP-trees need about 4–8

[19] I am not sure as to why sometimes collision detection time even *decreases* with increasing complexity.

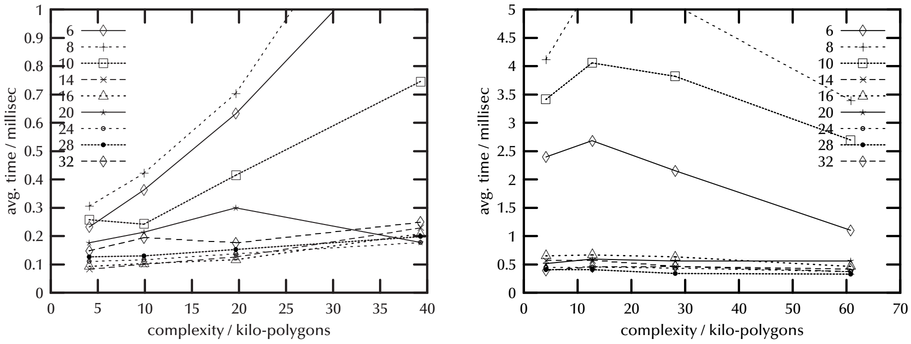


Figure 3.41: With the optimal number of orientations, collision detection time increases only slowly with increasing complexity.

		time / sec		
		# orientations		
	#pgons	6	8	14
door	3391	0.14	0.17	0.19
lock	20898	1.3	1.4	1.8
car	60755	4.2	4.7	6.4
pipes	124736	9.4	10.3	13.8

Table 3.3: Construction time of the BV hierarchy for various CAD and synthetic objects. The time increases by about 50 percent when 14-DOPs are used instead of 6-DOPs.

		MB		
		# orientations		
	#pgons	6	8	14
door	3391	0.1	0.1	0.1
lock	20898	1	1	2
car	60755	5	6	9
pipes	124736	11	13	19

Table 3.4: The amount of memory required by a DOP-tree with various orientation sets. 14-DOP-trees need about 80 percent more memory than 6-DOP-trees.

times less memory. (For the memory comparison, I have changed all `doubles` to `floats` in the Rapid code [Got97], because my DOP trees are implemented with `floats`. And, of course, the memory required by the list of vertices and polygons is taken into account, too.)

Table 3.6 summarizes the comparison of the construction time of the data structures for DOP- and OBB-trees.

The number of possible correspondences

Profiling has shown that a significant amount of time in the computation of equation 3.2 is spent fetching indices from correspondence j. The compiler is generally pretty good at arranging float and integer operations such that there are as few wait cycles as possible. However, in equation 3.2 there are about twice as many integer operations as float operations.

Since there are only finitely many correspondences, we can determine all of them and write specialized code for each of them to calculate equation 3.2. So, each of the correspondences can be hard-coded in a special function. At initialization of a simultaneous tree traversal, this function has to be determined, based on the correspondence. Then during traversal, this function can be called by function-pointer.

How many correspondences are there for a given set of orientations? For 6-DOPs as depicted in Figure 3.33, we can calculate that number by imagination. A cube has 8 vertices. Imagine the cube being suspended by one of its vertices

		MB	
	#pgons	6-DOP	OBB
door	3391	0.1	1
lock	20898	1	8
car	60755	5	21
pipes	124736	11	44
tori	12544	0.1	9
spheres	22952	1	17
tori	73984	6	60
spheres	97032	9	73

Table 3.5: A comparison of memory requirements indicates that DOP-trees need about 4–8 times less memory then OBB-trees.

		time / sec	
	#pgons	6-DOP	OBB
door	3391	0.1	0.4
lock	20898	1.3	4.5
car	60755	4.2	10.5
pipes	124736	9.4	18.4
tori	12544	0.6	2.4
spheres	22952	1.3	5.4
tori	73984	4.8	16.1
spheres	97032	6.7	25.6

Table 3.6: Comparison of the construction time of the BV hierarchy for various CAD and synthetic objects. The construction of DOP-trees is about 2–4 times faster.

by a thread. It can rotate around the vertical axis. Imagine a plane being moved from the right close to the cube such that it touches exactly. Imagine another plane touching from the front. Exactly 6 different vertices can be incident to the right plane. For each of them, there can be exactly two vertices touching the front plane. So, for cubes there are $8 \cdot 6 \cdot 2 = 96$ possible correspondences.

I have written a program which computes all possible correspondences. It takes the generic DOP (see Section 3.5.9) and samples SO_3 (the space of all orientations) by a large number of discrete orientations. For each of them, a correspondence is computed and added to the table, if not already there. Finally, C code is produced from that table.

For 6-DOPs the speed-up is summarized by the following table:

object	faces	speed-up
torus	10,000	1.5
sphere	10,000	1.2
cylinder	10,000	1.2–2.3
car body	60,000	1.2

We can determine the asymptotic number of correspondence by an argument similar to the one above. The generic k-DOP has $v = 2k - 4$ vertices. We pick one vertex of the generic DOP, attach it to the plane of one of the orientations, and suspend the DOP by that vertex. There are v possibilities to pick the vertex. Then we pick any other plane and make it touch the generic DOP while it is rotating. At most $v - 2$ different vertices can touch the second plane. How many different vertices can touch a third plane while the vertices touching the first two planes remain the same? I suppose, this can be only a small number. This would yield (k^2) as an upper bound for the number of possible correspondences.

Since the number of correspondences is growing rapidly as k increases, I believe that hard-coding the correspondences in separate functions is not reasonable for $k > 10$. For instance, I have found empirically that $k = 14$ yields 16,255 correspondences. That would turn into about 4.5 MB source code.

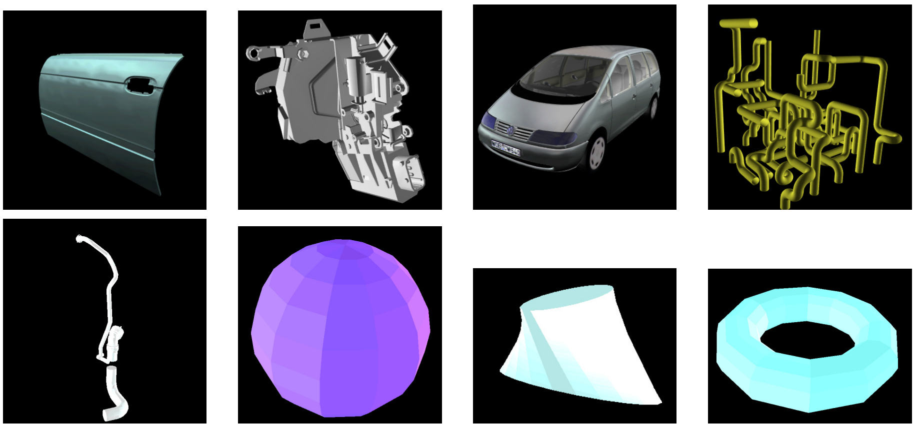


Figure 3.42: A suite of test objects. They are (left to right): the sheat metal of a car door (≈ 3,000 polygons), the lock of a car door (≈ 20,000–80,000 polygons), body and seats of a car (≈ 4,000–60,000 polygons) a section of pipes (≈ 120,000 polygons), a hose (≈ 4,000–40,000 polygons), sphere (≈ 3,000–90,000 polygons), hyperboloid (≈ 5,000–180,000 polygons), torus (≈ 3,000–90,000 polygons). (Data courtesy of VW and BMW)

Splitting DOPs

One might wonder how an algorithm similar to splitting of boxes would work for DOPs.[20] That way, not all plane offsets would have to be calculated when enclosing a tumbled child-DOP by an axis-aligned one.

However, since the orientations are not linearly independent (when $k > 2n$ in n-space) we cannot just cut a DOP in two parts by a plane perpendicular to one of the orientations: in general the two DOPs produced by that cut will have *redundant* planes!

Another idea to save some computations when calculating enclosing DOPs is based on the observation that not all d's of (tumbled) child-DOPs are different from the d's of its parent-DOP. For 6-DOPs, in general only 3 of the d's will be different. So, during a traversal of the DOP-tree we would need to compute only those d' which are affected by one of the "new" d's.

Unfortunately, with 6-DOPs 5 (3) d' have to be computed if only 3 (1) d are changed. So, for 6-DOPs this is probably not worth the effort. However, for larger k this might still be interesting, because the relation $\frac{1}{2} < \frac{\#\,\mathrm{new}d'}{\#\,\mathrm{new}d} < 1$ gets closer towards $\frac{1}{2}$.

3.5.10 Comparison of four hierarchical algorithms

This section compares four hierarchical collision detection algorithms with respect to detection speed: my Boxtree algorithm (see Section 3.5.4), my DOP-tree (see Section 3.5.9), OBB-trees [GLM96], and QuickCD [KHM+98]. For OBB trees, I used the Rapid implementation [Got97], and for QuickCD I used [KHM99].

[20] Actually, in the beginning I attempted to carry results and ideas from Section 3.5.6 over to DOP trees.

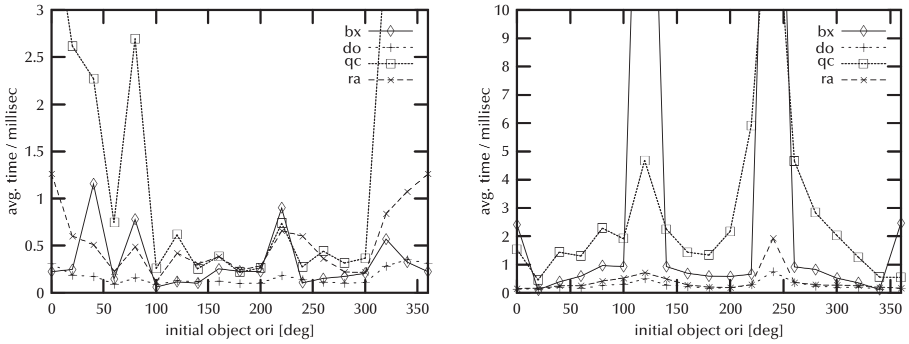


Figure 3.43: Initial object orientation (before constructing the BV tree) can have a significant effect on collision detection time. Left: door lock with 20,000 polygons, right: torus with 10,000 polygons.

General benchmarking procedure and results

It is extremely difficult to evaluate and compare collision detection algorithms, because in general they are very sensitive to specific scenarios, such as the relative size of the two objects, the relative position to each other, the distance, etc. I propose a simple benchmarking program which eliminates these effects. It has been kept very simple so that other researchers can easily reproduce my results and compare their algorithms.[21]

The test scenario involves two identical objects which are positioned at a certain distance $d = d_{\text{start}}$ from each other. The distance is computed between the centers of the bounding boxes of the two objects. Then, one of them performs a full revolution around the z-axis (which is pointing towards the viewer in Figure 3.42) in a fixed, large number of small steps (here 2000). With each step a collision query is done, and the average collision detection time for a complete revolution at that distance is computed. Then, d is decreased, and a new average collision detection time is computed, which yields graphs such as those shown in Figure 3.38.[22] Since the initial orientation of an object with respect to its object frame can have a significant impact on the efficiency of the BV-tree (see Figure 3.36), this procedure is repeated for several different initial object orientations (i.e, before the BV-tree is constructed). When plotting the average collision detection time, I have averaged over all "interesting" distances and all initial object orientations. Here, I have chosen the range from 2 distance steps *before* the contact distance through 2 steps *after* that point. I believe this reflects representative usage of collision detection. Initial object orientation means its orientation with respect to its reference frame, which could make a difference when constructing the BV tree.

I have carried out extensive experiments[23] using this benchmark procedure with different objects, both synthetic and real-world CAD data 3.42. All objects are scaled uniformly to fit in a cube of size 2^3. All timings include vertex and normal transforms. Except if otherwise noted, times are in milliseconds.

---

[21] The source code of the "main loop" and some synthetic objects can be ftp'ed from `http://www.igd.fhg.de/~zach/coldet/index.html`. The CAD objects can be obtained via the author (for scientific purposes only).

[22] The complete set of plots can be retrieved from `http://www.igd.fhg.de/~zach/coldet/index.html#dop-opt-ori`

[23] All tests have been done on a SGI R10000 194 MHz).

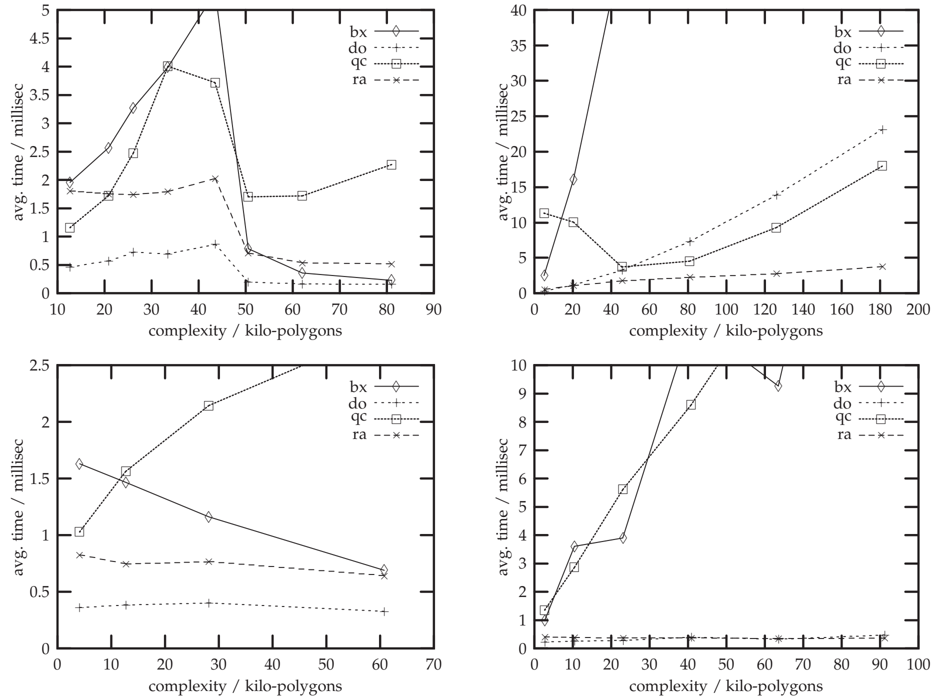


Figure 3.44: Comparison of the 4 hierarchical collision detection algorithms. The objects from top left: door lock, hyperboloid, car body, torus.

Results

From Figure 3.43 it is obvious that an object's orientation with respect to the object frame does make a difference in collision detection time. This is probably because the quality of some BV trees changes significantly when an object's initial orientation is changed.

Figure 3.44 shows that the Boxtree and QuickCD perform relatively poor compared to DOP-tree and OBB-tree.[24] Sometimes, my DOP-trees perform better than OBB-trees (in the Rapid implementation), sometimes vice versa.

It is not clear to me, why there is an "inversion" in the plot of the door lock, and why the DOP-trees perform so exceptionally poor with the hyperboloid object.

3.5.11 Incremental hierarchical algorithms

Since incremental collision detection has proven to be so effective for convex objects, it seems logical to pursue a similar approach for non-convex objects and combine it with hierarchical methods. Similarly, in the area of radiosity, hierarchical and incremental methods are being combined [DS97].

To my knowledge, only two papers have been published presenting an algorithm in this category: [LC98] construct sort of a "pair tree", the nodes of which represent a pair of BVs, one of each object's BV tree on the same level. In [PML97], an incremental hierarchical sweep-and-prune algorithm is presented. However, no results regarding performance have been presented.

[24] The complete set of plots can be downloaded from `http://www.igd.fhg.de/~zach/coldet/index.html#comparison`

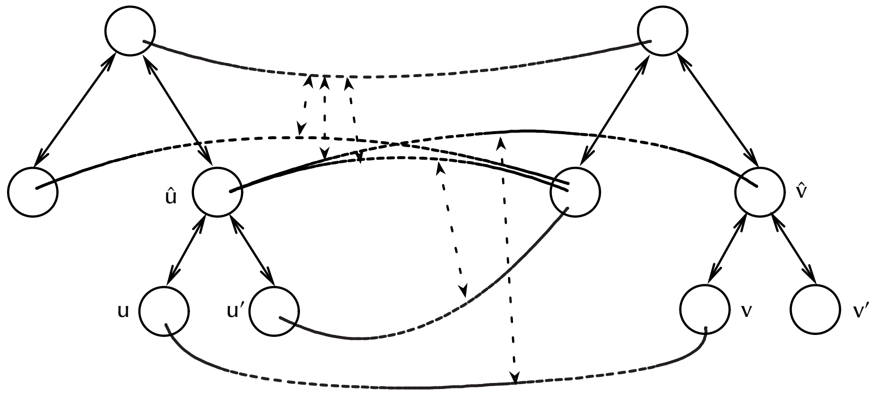


Figure 3.45: The link tree (dashed) connects two BV hierarchies (solid). It is created and maintained during top-down and bottom-up traversals. Bottom links are marked bold. Note that links can connect nodes at different levels in their respective hierarchies. If there is a link between two nodes, then the associated BVs have overlapped at some earlier time.

For my implementation I have used DOP trees. However, most of the following algorithm and discussion applies to all hierarchical BV schemes, such as sphere trees or OBB trees.

The new traversal scheme

Any collision detection scheme conceptually produces a list of pairs of polygons which need to be checked for intersection, because they are "sufficiently" close to each other (this list is not necessarily exhaustive). However, in traditional hierarchical collision detection algorithms this knowledge is discarded with every new collision test, even if those lists are almost the same in subsequent frames, because objects have moved only very little. This is because the BV hierarchies are always traversed from top to bottom.

Therefore, I introduce a new traversal scheme: a mixed bottom-up and top-down scheme. The idea is to resume (conceptually) at pairs of BVs where a previous collision test has left off. Then we check whether these pairs are still overlapping; if so, we proceed further down; if not, we back up (conceptually) and try to find other pairs where we can go down again.

Data structure

Let tree(P) denote the BV tree of object P. In order to save the information obtained during a traversal of two BV trees, I introduce a new data structure which I call the *BV link tree*: if two nodes $u, v, u \in$ tree$(P), v \in$ tree(Q), overlap, then we create a link(u, v) between the two (see Figure 3.45). In the following I will call leaves of the link tree *bottom links*. These links either connect two leaf nodes of the BV trees, or all 4 pairs of BVs of their child nodes do not overlap. In other words, bottom links are created when the top-down traversal terminates.

As we will see below, links will be created only between pairs of BVs which are visited during a top-down traversal. Note that links do not necessarily connect nodes only on the same level. However, if they do connect nodes on different levels, then at least one of the BVs connected is a leaf node.

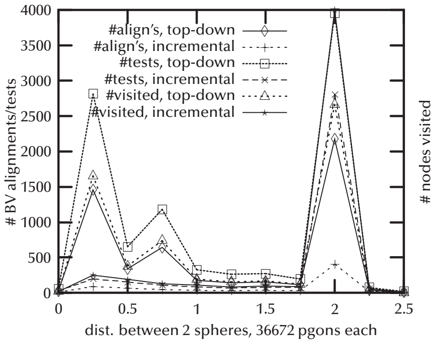


Figure 3.46: The number of BV test, DOP alignments, and nodes visited depends on the distance between the objects.

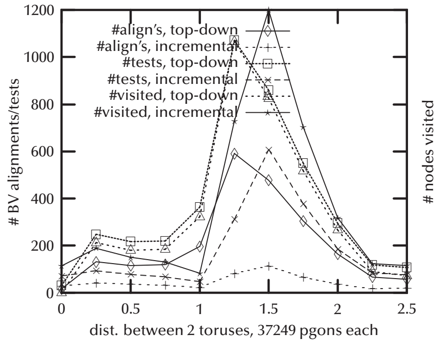


Figure 3.47: It also depends very much on the particular objects involved.

In the following, I will develop an algorithm to maintain such a link tree while performing collision detection. However, we will not necessarily visit all links (i.e., nodes) of the link tree during a collision detection query. Therefore, if there is a link(u, v), then at some earlier "frame" the BVs u and v overlapped.

Other properties of the link tree are obvious: if there is a link(u, v), then there is also a link$(\hat{u}, \hat{v})$, with $\hat{u}, \hat{v}$ the parents of u, v, resp. And, there will be a bottom link which is a child of link(u, v) (or itself).

The algorithm

Let us assume we are given the BV trees of two objects P and Q. We are also given a link tree between the two BV trees, which could have been obtained by an ordinary top-down traversal.

Assume that a BV pair (u, v) overlaps; then we know that all "parent pairs" overlap, too. Therefore, we can traverse the hierarchies *upward* and do not need to test BVs for overlap.

So, the idea is to maintain a list of bottom links and a link tree. For each bottom link, we will first check whether it is still a valid link (i.e., whether the associated BVs still overlap), and whether we should further traverse the BV trees down. This might already produce an intersection, and in that case we are done. Otherwise, we traverse the link tree (and the BV trees) up.

During up-traversal, we have to check all pairs of children of a pair of BVs and possibly traverse down the respective sub-trees. Of course, if a pair of children is already connected by a link, then we do not need to traverse down into that sub-tree, because we will pass that pair later during an up-traversal (or find an intersection before).

So, basically the algorithm resumes where a previous run has stopped. Probably, it resumes several times at different nodes of the trees. During up-traversal it "spawns" down-traversals to explore some parts of the trees which have not been visited in a previous run, but which need to be visited now, because their BVs overlap.

The algorithm for up-traversal works as follows:

up($\hat{u}$, $\hat{v}$, u, v)

link$(u, v,)$ *has been visited already*
    $\longrightarrow$ return

```
û,v̂ overlap                              {trivial if u,v overlap }
    ⟶
    forall (u',v') ∈ children(û,v̂)
       if there is no link(u',v')
       ⟶ down(u',v')
û,v̂ don't overlap
    ⟶
    forall (u',v') ∈ children(û,v̂)
       clear all links between sub-trees
          rooted at u',v'
up( parent(û,v̂), û,v̂ )
```

Here, `down()` denotes the conventional top-down traversal, while parent() and children() should be obvious.

Note that with this implementation a link(u, v) means only that at some time the BVs at nodes u and v did overlap – it does not necessarily mean that this was true the last time a collision was checked.

So far, the algorithm "only" saves BV overlap tests. Since at present I am using my DOP tree algorithm, I also want to minimize the number of DOP alignments: during the conventional top-down traversal, we need to enclose one of the two DOPs so that we can perform an overlap test quickly [Zac98c]. Although this can be done by an affine transformation, it is still one of the most time-consuming operations during traversal.

The number of alignments can be reduced by deferring them (i.e., by *lazy evaluation*). When we find out that the BV pair u, v does not overlap, then we do not know anything about the status of the parent pair $\hat{u}, \hat{v}$. In order to find out that, we need to compute the aligned DOP of $\hat{u}$. If $\hat{u}$'s other child u' ($u \neq u'$) has some link, then we know that $\hat{u}$ will be visited later by another up-traversal via u' (maybe that has already happened; then, of course, we do not defer $\hat{u}$'s alignment). When $\hat{u}$ is visited again eventually (via u'), we can compute its aligned DOP by a simple unification of the aligned DOPs of u and u'. This is not only much faster, but also the resulting DOP is tighter than the one obtained by an affine transformation of the "tumbled" DOP.

So, during traversal we will maintain a list of "deferred links", i.e. links which have been visited already but whose overlap status has not been checked yet. So, the initial loop over all bottom links is followed by a loop over all deferred links:

```
incremental collision detection

forall bottom links (u,v)
    down( u, v )                   {checks polygons in leaves }
    if intersection
       return "intersection"
    up( û, v̂, u, v )

while there are deferred links (u,v)
    up( û, v̂, u, v )
```

Note that more links can be deferred while one of the links deferred earlier is being processed.

A node can be involved in many links (see Figure 3.45). Therefore, it can be visited many times by different up-traversals. Therefore, DOP alignments can be reduced further by saving the aligned DOP with the node.

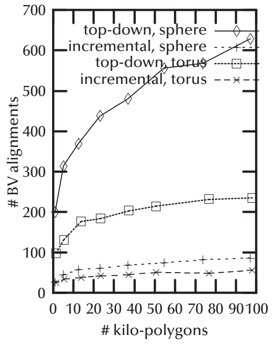


Figure 3.48: The average number of BV alignments for the incremental and the top-down algorithm.

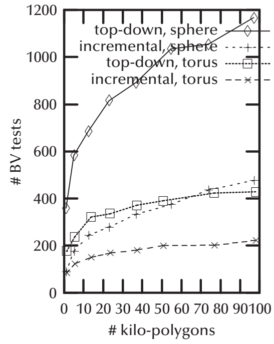


Figure 3.49: The average number of BV tests.

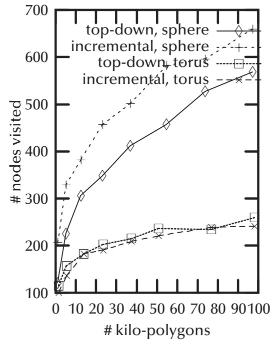


Figure 3.50: The average number of nodes visited. It is slightly larger for the incremental algorithm.

Note that in the bad case (“no intersection”), still the same number of nodes have to be visited during simultaneous traversal of the BV trees. In the good case (“there is an intersection”), it should find a witness early on. In the bad case, the number of DOP alignments and BV tests will be decreased significantly (see below).

Results

For the evaluation of my incremental algorithm and for its comparison with the non-incremental version, I have used the procedure described in Section 3.5.10. All results have been obtained on a 195MHz R10000 with 2×32 kBytes primary cache and 4 MBytes secondary cache.

The test scenario results in graphs such as Figures 3.46 and 3.47. I have investigated several characteristic numbers, namely the number of BV alignments[25], the number of BV overlap tests, and the number of nodes visited.

As one can see from the plots, the number of explicit BV alignments and the number of BV tests have been reduced significantly with the incremental algorithm. Experiments with other polygon numbers (I have done 5,000 through 100,000 polygons) show that the relationship stays constant: the incremental algorithm does about 5 times less BV alignments than the top-down algorithm. Similarly, the relationship between the incremental and the top-down algorithm remains constant for the number of BV tests for different numbers of polygons.

Because my algorithm can defer the verification of a link (i.e., BV alignment), a node can be visited several times – each time has been counted separately, even if nothing is done except adding a link to the list of deferred links. This is why the number of nodes visited is larger for the incremental algorithm than for the conventional top-down method.

When averaged across all distances, we can evaluate the algorithm’s dependence on the number of polygons. Figures 3.48, 3.49, and 3.50 show the characteristic numbers for spheres and tori of different sizes, respectively.

[25] This is one step of a BV overlap test – other hierarchical BV algorithms might involve similar steps, such as the transformation of a sphere.

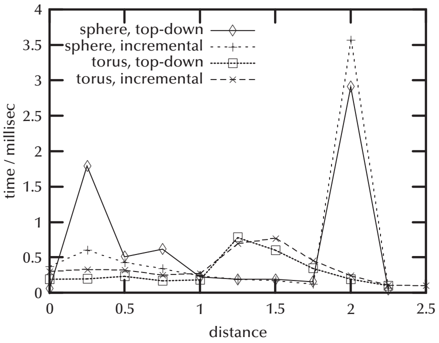


Figure 3.51: The collision query time depends on the distance of the two objects (approx. 23,000 pgons/object). I believe that the incremental algorithm does not perform better due to a significantly higher number of secondary level cache misses.

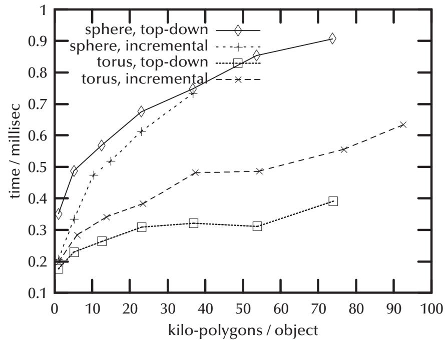


Figure 3.52: The performance of the incremental and the non-incremental top-down algorithm compared for several polygon counts.

Of course, the most important measure is the collision query time. Plots similar to the previous ones are shown in Figures 3.51 and 3.52 for the time. Unfortunately, the timing results are not as expected: both algorithms take about the same time.

I believe that this cannot be explained satisfactorily by the algorithm's repeated visits of some nodes. (In the case of the torus, the number of nodes visited by the incremental algorithm is actually the same as for the top-down algorithm!) Profiling showed that the incremental algorithm produces about 6 times more secondary cache misses then the conventional top-down algorithm. I suspect that this is fatal, because "leaving" the secondary cache can incur a performance penalty of factor 4–10 (see Section 3.11.3).

Still, I believe that the results on the characteristic numbers are quite promising. What needs to be done is a better up-traversal scheme which does not need to defer the alignment of BVs, yet utilizes lazy evaluation when computing them.

Discussion

The algorithm's complexity in the worst case is still the same as for the conventional top-down algorithm. However, the results are quite satisfactory in terms of the reduced number of BV tests and DOP alignments. This shows the algorithm's potential for improving the query time in the case of successive collision tests with only slightly moved objects.

Under certain assumptions, [LC98] have pointed out that the maximum speed-up by an incremental algorithm is a factor of 2. However, I believe that these assumptions hold true only for a particular type of algorithm.

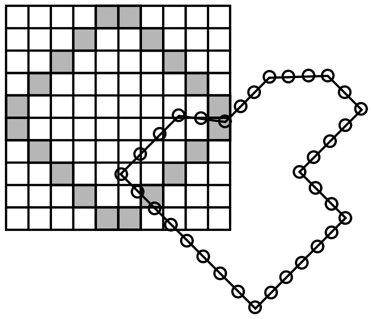


Figure 3.53: Other representations could make faster collision detection possible. For instance, the basic step with voxels and point clouds is very fast.

3.6 Non-hierarchical algorithms

3.6.1 Points and Voxels

So far, the basic operation has been the intersection of two polygons (or an edge and a polygon), or the overlap test of two bounding volumes. The goal of almost all algorithms so far was to reduce the number of these basic operations. But instead of looking at new ways to further reduce it, we should also look at other object representations.

In contrast to checking polygons against polygons, we can also check points against voxels. In a sense, this means that we represent objects (for the purpose of collision detection only) by different primitives. This idea has been presented by [MPT99] in the meantime.

So the idea is to represent one of the objects of a pair by a point cloud. The points should be distributed equidistantly on the surface of the object. The other object is decomposed in a volume of voxels, in this case just binary voxels. If the object is closed, then we could mark also those voxels black which are completely inside the object — this depends on the definition of "collision" we want to adopt.

Let us assume that object P is approximated by a point cloud[26] and object Q is approximated by a voxel volume (see Figure 3.53). A brute-force algorithm would then transform each point of P into the coordinate system of Q, calculate indices, and access the voxel volume of Q.

The plain voxel grid is very memory-inefficient. If Q's bounding box is a cube of size s and it is voxelized by an n^3 grid, then the approximation error is $\varepsilon \leq \sqrt{3}\frac{s}{n}$. We need 1 bit per voxel, so the memory needed for voxeling Q is $\geq (\frac{\sqrt{3}s}{2\varepsilon})^3$. For instance, to approximate an object with bbox size 100 mm and 1 mm resolution we need about 0.6 MB, and for 0.1 mm resolution we need about 620 MB. Let us assume that the area of the surface of an object with bounding box of size s is $2\frac{4}{3}\pi(\frac{s}{2})^2$ ($2\times$ the area of a sphere). If any point on the surface is no more than ε away from one of the clouds (measured *on* the surface), then the surface must be covered completely by disks of radius ε. Suppose the total area is covered by 130% of the area of all disks. Then we need to approximate one object by a point cloud of $\frac{1.3\times 4}{3}(\frac{s}{2\varepsilon})^2$ many points.

[26] In general, we will want to approximate that object by a point cloud which is the smaller one. For a given resolution, this will result in fewer points.

With the same example as above, we need about 44,000 points or 0.5 MB for an approximation error of 0.1 mm.

In order to reduce memory or increase resolution, one of the octree techniques [Sam90c, CCV85, GA93, WSC+95] can be used (see Section 3.8). Resolution can be increased further by an exact octree represent [CCV85]. Of course, this decreases speed significantly. As with hierarchical collision detection, an efficient trade-off has to be found which minimizes a cost equation similar to 3.1.

In order to increase speed, the point sets should be stored hierarchically. Several methods could be employed, such as k-d trees [Ben90], octrees for points [PS85], or even BoxTrees or OBB-trees (see Sections 3.5.4 and 3.5.8).

If objects can be represented *exactly* by polygons (which is generally the case if they have been modeled using polygons), then the point cloud and voxel representations introduce an approximation error. However, in virtual prototyping applications polygons are always approximations themselves. In order to keep the error low, one should, of course, create the point clouds and voxels from the CAD model directly.

For covering polygonal models by point clouds, [Tur92] have presented an algorithm.

This representation has the advantage that it becomes fairly easy to compute the minimal distance between P and Q. To that end, Q's voxel volume has to hold scalars, which will increase the memory usage (by a factor 32 if the scalars are floats). Each voxel contains the minimal distance from that voxel to Q. Such a distance field can be pre-computed.

3.7 Flexible Objects

Flexibility of objects complicates the collision detection problem significantly, because precomputations are generally invalidated by a deformation. On the other hand, it seems non-trivial to devise algorithms not depending on precomputations yet still as fast as the ones depending on them, such as hierarchical algorithms.

Collision detection of flexible objects can even be complicated further if self-intersections have to be checked, too [MW88]. It cannot be reduced to general collision detection of flexible objects. It is more complicated than general collision detection, because the algorithm has to filter out those pairs of polygons which are adjacent. These polygons always touch or intersect each other.

A different representation is used by [Gas93]: objects are modeled by a skeleton and a field function. Thus every object is closed. Collision detection is done by sampling the surface and testing each point if it is inside or outside.

[Sny95] use parametric or implicit curved surfaces and an interval Newton method to find the minimum of a certain function, which describes the conditions for a collision.

In order to speed up collision detection of deformable polygonal models, [VT94] assume smoothly discretized models. Based on that they develop a criterion for pooling manu adjacent polygons, so that they can be checked simultaneously.

The approach taken by [vdB99] is simply to "re-fit" an existing box tree by traversing it bottom-up. This approach works for AABBs and seems to be faster than rebuilding OBB trees for reasonable polygon numbers. It would also work

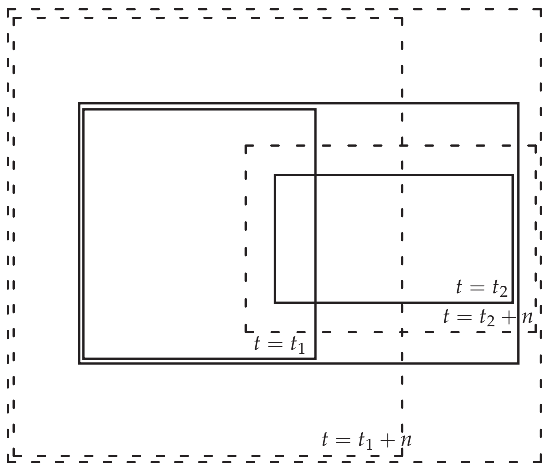


Figure 3.54: DOPs are updated during withdrawal of the "grow-shrink" algorithm. Otherwise they grow each frame according to the temporal coherence assumption.

for DOP-trees. [SAK+95] basically rebuild an octree for the faces of all objects participating in collision detection.

3.7.1 The "grow-shrink" algorithm

Even though objects are flexible, it would be nice to be able to still utilize hierarchical collision detection. A brute-force way to do that would be do rebuild the hierarchy each time the object has deformed. This approach would probably be slower than the algorithm of Section 3.3 for most practical complexities. However, simply re-fitting the whole tree ([vdB99]) seems to be overkill, because many nodes of the tree are not involved in an overlap test, especially those at the lower levels.

The idea is to retain the structure of the BV hierarchy and *re-fit* its node when necessary. So, we need a criterion when to update nodes. The brute-force method would traverse the complete hierarchy of both objects before doing the collision test. However, by making an additional assumption we can refine the criterion: let us assume that any vertex of object P does not move farther than Δ^P (in object space). This assumption captures *temporal coherence* of the deformation of objects. It should be valid especially in virtual environments.

Let us assume that we are given two DOPs[27] $\mathbf{d}^P$ and $\mathbf{d}^Q$ (with the definitions of Section 3.5.9). Let us further assume that f frames have passed since they were computed exactly the last time. Then, the following DOP overlap test tells us if the polygons enclosed cannot intersect:

$$\exists\, i : \; d_i^P + f\Delta^P < -(d^Q_{i+\frac{k}{2}} + f\Delta^Q) \quad , \; 0 \le i < \frac{k}{2}$$
$$\vee$$
$$\exists\, i : \; -(d_i^P + f\Delta^P) < d^Q_{i+\frac{k}{2}} + f\Delta^Q \quad , \; \frac{k}{2} \le i < k$$

[27] Of course, the algorithm works with any type of bounding volume which can be expanded inexpensively.

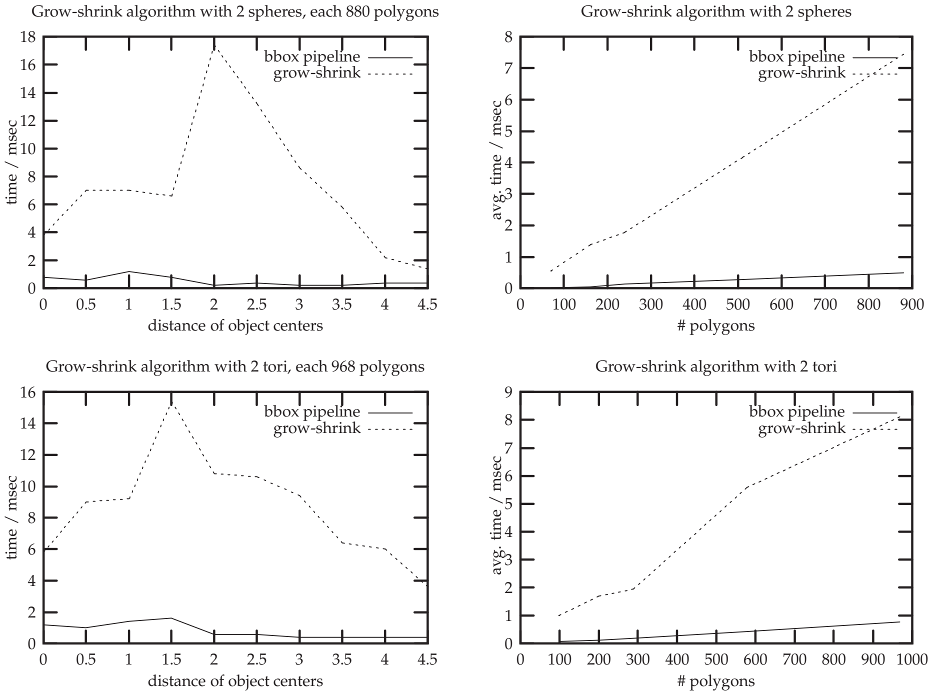


Figure 3.55: The grow-shrink algorithm seems to perform always worse than the bounding box pipeline algorithm. This might be because the algorithm will traverse the hierarchy to a larger depth.

In other words, the idea is to "grow" a DOP so that it still contains all the polygons assigned to it, even if they have moved as far as possible under the assumption. Of course, the overlap test really has to be done in world space by applying the algorithm from Section 3.5.9.

Obviously, if grown DOPs do not overlap, then we do not need to traverse further into that part of the tree. Every frame this happens, the DOPs will grow further by one shell. So, eventually they will overlap (this will happen the sooner, the closer they are). At this point, they need to be "shrunk". This is done by a union of the children of the DOP during withdrawal (see Figure 3.54).

Each node in the BV tree has one additional parameter, namely a time-stamp telling for which time it is "exact". With this time-stamp the algorithm knows how much to grow the DOP during traversal. Analogously to Section 3.5.1, the following pseudo code outlines the algorithm:

```
Grow-and-Shrink hierarchical collision detection

a = bounding volume of P's tree,
b = bounding volume of Q's tree,
f = frame time
a[i], b[i] children of a and b, resp.
growshrink( a, b, f ):
test grown(!)  (a,b) for overlap
no overlap
⟶ return 0
a and b are leaves ⟶
   shrink DOPs of a and b
   test (a,b) for overlap
```

```
      no overlap
      ⟶ return 0
      test polygons of a and b for overlap
      return result
   a or b is not leaf ⟶
      forall (a_i,b_j) ∈ children(a,b):
         growshrink(a_i,b_j)
         (a_i,b_j) intersect
         ⟶ break
      shrink DOPs of a and b                         {if possible }
      return result
```

<u>Shrink a DOP</u>

```
   a = bounding volume
   f = current frame time

   a is leaf ⟶
      compute new extent of a
      t := f
   a not leaf ⟶
      if min_i{t(a_i)} > f ⟶
         t := min_i{t(a_i)}
         a := ⋃_i a_i
```

Unfortunately, this algorithm seems to perform worse than the bounding box pipeline (see Section 3.3) in all practical cases. Figure 3.55 shows the behavior of the algorithm in detail for tori and spheres. Although benchmarks for larger complexities have not been carried out, the trend suggests that this algorithm is always slower than the bounding box pipeline algorithm.

It seems that this algorithm traverses many more levels than expected. Of course, the algorithm will traverse the bounding volume hierarchies to increasingly deeper levels as the BVs "grow". But as soon as leaves overlap, BVs on higher levels shrink to their minimal size. Maybe the bounding box pipeline is more efficient than one would think it is, i.e., maybe it is very good at reducing the number of pairs of polygons.

3.7.2 Sorting

Basically, collision detection is the problem of finding pairs of overlapping boxes. If we consider axis-aligned boxes, this problem can be reduced to finding overlapping one-dimensional intervals (along three axes). In the static case, several data structures have been invented to solve that problem. Among them are *segment trees* and *interval trees* [Ove88b, SW82], and R^*-trees [BKSS90]. Actually, R^*-trees are suitable for n-dimensional queries. However, it is not clear how these data structures can be adapted to the dynamic case in which *all* boxes move continuously as well as change their size.

When the boxes move relative to the object's coordinate system, it is not trivial to maintain hierarchical data structures of boxes with less effort than rebuilding them from scratch each time. Still, we would like to take advantage of temporal coherence of the motion of the boxes.

Suppose we are given two sets of boxes, one set of red boxes, and one set of blue boxes. The idea is to project the boxes on all coordinate axes. Along

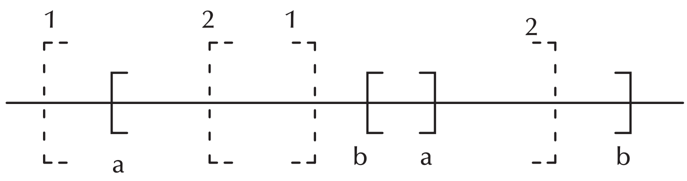


Figure 3.56: An incremental collision detection algorithm can be derived from maintaining sorted red and blue brackets on three axes.

each coordinate axis, we are then given a set of red and blue intervals (see Figure 3.56). Each interval consists of two *brackets*: an *opening* bracket (or *left* bracket) and a *closing* (or *right*) bracket. The goal is to find all overlapping intervals, one of which is red and the other one blue. This can be done by three *sweeps* across each axis, with the interval endpoints sorted along them.

During the sweep, we keep a list of *active* intervals. An interval is active, if we have encountered an opening bracket, but not yet its closing bracket. When the sweep encounters a closing bracket, the interval is removed from the list of active intervals. When the sweep encounters an opening bracket, two things happen: we increment a counter for all pairs of red/blue intervals, where one of them is the interval currently selected by the sweep, the other interval is from the list of active intervals; after that, the current interval is added to the list of active intervals.

For each red/blue interval pair, a counter is maintained in an *overlap* matrix. After all 3 sweeps have been finished, this counter tells whether or not the two intervals overlap.

Temporal coherence can be utilized by incrementally sorting the intervals along each axis. Once the brackets are sorted along an axis, they should be "almost" sorted in the next frame. I use Bubblesort for updating the ordering of the brackets.

The overlap matrix could become very large for moderate polygon sizes, even if a single char is used for each entry. In general, it is a sparse matrix. In particular, if a pair of intervals does not overlap along the first axis, then we won't need to create an entry for it during any further sweeps. Therefore, the overlap matrix is implemented as a list of lists. Furthermore, the algorithm can be optimized by the following observation: if two intervals do not overlap on one of the axes, then we do not have to test the corresponding pair of intervals on any of the other axes.

Unfortunately, this algorithm performs even worse than the one in the previous subsection. One problem is that the contents of the matrix are computed from scratch each time. In addition, profiling has shown that accessing the sparse matrix takes a lot of time. In addition, I suspect that each single access causes several cache misses (see Section 3.11.3).

In order to speed up this kind of algorithm, several problems would need to be solved: a fast implementation of sparse matrices, and a method that allows to re-use the contents of the overlap matrix. The idea is to update only those matrix entries that need to be changed, because many intervals that overlapped during the last frame are likely to do so in the current frame. This would require to utilize both the intervals as of last frame sorted along each axis and the current position of all intervals. However, as polygons become smaller and smaller the benefit of this is questionable. In addition, housekeeping for such an update algorithm seems to me to become too difficult to yield any speed-up.

Ideally, one would like to completely do away with the overlap matrix.

3.8 The object level

Some VEs and some simulations need to calculate the behavior of a huge number of moving objects, or of several objects moving in an environment of a huge number of stationary ones. These environments pose the *all-pairs problem* on a global level — just like the all-pairs problem on the face level.[28] Of course, before doing any collision check between two objects, we first check their bounding volumes for intersection. Still, the complexity is quadratic, i.e., with n moving objects we have to do $\sim \frac{n^2}{2}$ bounding volume checks. Usually, the objects are distributed fairly evenly about the volume of interest (the "universe"), so even in the case where one object moves among n stationary ones, checking n bounding boxes for overlap is too expensive.

Although the problem bears some similarity to collision detection on the polygon level, the methods developed there do not necessarily apply as well to the global problem. The difference is that here, the basic primitives (i.e., objects) maintain no relationship to one another. In particular, all of them can move with respect to one another. Even methods developed for collision detection of flexible objects might not be applicable to the global situation, in particular if they utilize some smoothness or topological or hull property of the set of polygons (or subsets).

There are basically two methods to solve the problem, both of which exploit *space coherency*: most regions of the universe are occupied by only one object or empty. Consequently, each object has a very small number of "neighbors" (in some sense); only these neighbors have to be tested for collision with the object itself. One class of methods is *space indexing* which builds on space partitioning data structures (possibly hierarchical ones), the other class of methods is "object-oriented" in the sense that objects are maintained in a hierarchy or sorted in some order.[29]

One might wonder when it does actually pay off to determine neighbors by a global method, i.e., is it always worth to try to avoid $\frac{n^2}{2}$ bounding box tests? Let

$$\begin{aligned}
P_b(n) &= \text{number of neighbor pairs according to BV,} \\
P_s(n) &= \text{number of neighbor pairs according to space indexing} \\
T_b &= \text{time needed for one BV neighbor test} \\
T_s(n) &= \text{time needed for space indexing neighbor test} \\
T_p &= \text{time needed for one exact collision test (average)}
\end{aligned}$$

Then, space-indexing is more efficient if

$$\begin{aligned}
P_b(n)T_p + P_s(n)T_b + T_s(n) < P_b(n)T_p + \frac{n^2}{2}T_b \qquad &\Rightarrow \\
P_s(n) < \frac{n^2}{2} - \frac{T_s(n)}{T_b}
\end{aligned}$$

---

28 In some sense, these problems and levels are similar to the levels addressed by *hierarchical radiosity* and *clustering*. The only difference is that in radiosity *polygons* are usually considered the basic "objects". Hierarchical radiosity tries to solve the all-pairs problem for a pair of objects (i.e., polygons), by building hierarchies of patches (which are the basic entities an object consists of), while clustering tries to solve the all-pairs problem for the complete scene by building hierarchies of objects.

29 Here, one might draw the analogy to the concepts of *screen precision* and *object precision*, or *screen space* and *object space*.

If there was no additional BV test after the space-indexding, then the break-even point would be

$$P_s(n) - P_b(n) < \frac{\frac{n^2}{2}T_b - T_s(n)}{T_p}$$

This fits well with my experience: if there are only a few, large objects, a global neighbor-finding method is not worthwhile; on the other hand, for a large number of objects consisting of only a few polygons, a global method is necessary.

3.8.1 Other approaches

Sweeping plane. The basic idea by [CLMP95, BF79a, PS90] is to sweep a plane along the x-axis; whenever this plane intersects more than two boxes at a time, we check all the rectangles obtained by the actual intersection of the sweep plane at the current position with those boxes. This is now a similar test in two dimensions, which can be performed by the same method, now with a sweep line instead of a plane. This method is well suited for exploiting temporal coherence.

Since there are only finitely many boxes, we can first sort the set of x-values of all boxes. Then, the sweep along the x-axis can be done by "hopping" from one x-value to the next in the sorted list. During the sweep, we maintain a list of y-values and z-values. At each x-value found in the list we either *enter*, or *leave* a box; correspondingly, we add or remove its y- and z-values from the y- and z-value arrays. Since the sweep along the y-plane will need the y-array to be sorted, too, we will add and remove y- and z-values by insertion sort (which involves at most one pass over the two lists).

Bounding-volume hierarchy. The idea is to exploit the scene graph hierarchy which is already present in the scene, or, if there is none yet,[30] to create such a hierarchy ([YW93]). The idea is similar to bounding box hierarchies of polygons. The differences are: here, we are given only one tree, and we have to find overlapping pairs within that single tree; and, second, all leaves are, potentially, moving.

For ray-tracing, bounding box hierarchies work very well [KK86], because the scene is entirely static; so, the hierarchy can be built once at start-up time. In order to yield any significant improvements compared to the quadratic all-pairs case, the hierarchy must be well modeled and fairly deep.

However, with this approach there are two problems:

1. In dynamic environments, the hierarchy would have to be rebuilt every n-th frame in order to maintain a fairly optimal correlation between spatial locality and tree locality. Since ray-tracers have to build the hierarchy only at start-up time, they can create a much finer and better optimized hierarchy on polygon level.

   Imagine an assembly line scenario with many objects moving through large volumes and many robots moving within their stationary working

[30] Scenes in virtual prototyping applications are not yet modeled hierarchically. This will probably change in the next decade for automotive applications, as car makers are building product data management systems, tools, and design guidelines to create so-called "process data". These will contain, among others, hierarchical model information.

volume. In such a scenario, any bounding box hierarchy will soon become useless, because there will be no correlation between geometrical locality and topological locality.

2. Usually, there are many objects of the scene which do not have to be checked for collision (see Section 3.9). In that case, the scene graph is sparse in terms of collision objects. Therefore, large parts of the scene graph will be traversed just to "find" a single collision object.

3.8.2 Bounding Volumes

In order to gain any speed, we will deal only with bounding volumes throughout this whole section. So, whenever I use the term "object" in this section, I mean its bounding volume. Of course, the bounding volume must be much simpler an object than the object it bounds. At the same time, this is the general disadvantage of bounding volumes: depending on the geometry of the object "inside", they can contain very much "empty" space. Some of the desirable characteristics of bounding volumes are:

- easy to compute,
- little memory requirements,
- fast transformable,
- simple overlap check,
- tight fitting.

There are a few very simple, commonly used bounding volumes: *axis-aligned bounding box* (AABB), *sphere*, *DOP*. AABBs are probably the optimal BV with respect to fast overlap tests. Unfortunately, they are not invariant under rigid motions. An enclosing AABB of a transformed AABB can be up to 2 times as large in volume as the original bounding box. They can be computed by 45 FLOPs (18 mult. + 18 add. + 9 comp.) [Zac94b, Gla90].

The geometry of spheres is probably the simplest one, which makes them attractive. Spheres are invariant under rigid motions. However, an overlap test between two spheres is not quite as inexpensive as an overlap test between two axis-aligned boxes. Computing the optimal bounding sphere is not nearly as easy as computing an AABB. The brute-force method is in $O(n^4)$. For exact optimal bounding spheres, the algorithm usually chosen is linear or quadratic programming [Meg83, EH72]. Computing an almost-optimal solution seems to be more feasible [Wel91, Rit90, Wu92].

DOPs seem to have been considered first for ray tracers [KK86]. They can be considered a generalization of AABBs. The advantage is that one can trade tightness for overlap test speed.

Cylinders and prisms seem to be simple, too [BCG$^+$96]. However, they do not seem to be too useful, even with static scenes, probably because their geometry is already too complicated [WHG84]. For curved surfaces, such as splines, curved BVs such as spherical shells seem to be a good choice [KGL$^+$98, KPLM98].

3.8.3 Space-indexing data structures

The problem of finding neighbors (in some sense) of an object among a set of similar objects is sometimes called *space indexing*, *space partitioning*, *range* *search*, *space covering*, etc. Many data structures have been developed in the past to help find neighbors, determine proximity, and many similar problems. Computational geometry has recognized the *range search* problem very early [BF79b].

All of these methods exploit *space coherency* and the fact that most regions of the "universe" are occupied by only one object or they are empty. Consequently, each object has a small number of neighbors, compared to the total number of objects.

As mentioned before, most of the data structures developed so far (especially those in the field of computational geometry), assume a static environment. However, the serious challenge is the dynamic quality of the environment. While the only criterion with static environments is *fast retrievability* and *fast neighbor-finding*, the criterion with dynamic environments is, in addition, *fast updating* for moving objects.

BSP. Binary space partitioning was developed to partition a whole scene (a set of polygons), so as to solve the hidden surface problem [FKN80]. For visualization of dynamic scenes, the data structure has been augmented by so-called "auxiliary planes" [Tor90]. These try to divide space without cutting objects.

Despite the simplicity of the basic idea, there is the annoying problem of cut objects. In general, this cannot be avoided, and the size of a BSP tree can get rather large — the lower bound is $\Omega(n^2)$ for n objects. Only recently an algorithm has been presented which can construct the optimal BSP (see [PY90]).

It seems to me that BSPs have been mainly used for solid modeling, where objects can be represented and operated on by using BSPs [TN87a, NAT90a].

Cell subdivisions. Cell-based neighbor-finding implies the following definition: given a set of cells, each of which can be "occupied" by one more object(s). Two objects are neighbors, if there is a cell occupied by both of them.

There are two basic classes of cell decompositions of space: uniform and hierarchical. Usually, boxes are used (which yield a space subdivision), but other BVs can be used as well, such as spheres (which yield a space covering).

In contrast to BSPs, uniform cell subdivisions are not "object oriented" but space oriented, i.e., the data structure itself does not depend on the arrangement of objects. Instead, this data structure is built once at start-up time; later on, cells are just "filled" with the objects they contain.

Hierarchical cell decompositions try to overcome this shortcoming by recursive subdivision of cells, based on criteria on the objects' distribution. They can better handle uneven object distributions. The most common scheme are *octrees* [Sam90a, Sam90b]. They have been heavily used for ray-tracing [GA93, MSH+92, Sun91], and solid modeling [TKM84, FK85, NAB86]. Octrees have been combined with B-reps in order to combine fast algorithms for boolean operations and exact object representation [CCV85].

The most basic cell subdivision are uniform *grids*. Grids are such a simple data structure that there is not too much literature on them [Ove88c, Ove88a, HT92]. For ray tracing, regular grids have proved to be the second-best data structure [MSH+92].

Grids are used in a dynamic environment by [ZOMP93]. However, they use it in a very large environment where objects are very small compared to grid cells. In fact, they are so small, that objects are approximated by points. Furthermore, cells are large enough so that objects cannot traverse more than one cell per frame. In molecular modeling, [Tur89] has used grids to speed up collision detection time among spheres. [CAS92] applied octrees for collision avoidance with tele-operation in a dynamic context.

I have developed algorithms for updating grids and octrees quickly in a dynamic environment (see below).

Macro-regions are based on a (fixed) grid and, like octrees, the method tries to group large areas of contiguous empty voxels [Dev89]. They are more flexible, though, because they do not superimpose other grid layers. Instead, these areas of empty voxels (called "macro-regions") are rectangular, and as large as possible. They may (and usually will) overlap.

Adaptive grids have been developed by [KS97] for accelerating ray tracing. They are a hybrid approach, combining hierarchical space partitioning with the benefits of regular grids.

Field trees are somewhat similar to adaptive grids, in that they comprise several layers [FB90b]. Unlike adaptive grids, however, they seem to be more suitable for uniform object distributions and densities.

3.8.4 Octrees

In the following I will describe algorithms for quickly updating octrees in a dynamic environment of moving objects (boxes). First, I will describe basic insertion; then, how to update an octree efficiently.

The fundamental step for finding neighbors is *insertion*. A simple algorithm to insert all objects in the octree. Then, we can simply enumerate all cells occupied by an object by just executing the insertion algorithm once more. When objects move between frames, the octree has to be updated. Again, this can be done naively by using the insertion algorithm: we make one insertion with the old position to remove any reference to the object to be moved, then we make a second pass with the new position to insert it again.

Basic insertion algorithm

The basic algorithm for inserting an object B is quite simple:

```
Octree insertion
input:  box B, octant o

o is leaf ⟶
   add B to o's list
   return
o completely inside B ⟶
   add B to o's list
   return
i = 1...8:
B intersects with sub-octant o_i
⟶ insert B in o_i
```

Since an octree is (conceptually) a multi-layer uniform grid, we can use integer arithmetic to do the box-octant comparisons. This defines a voxel to be a "half-open" cube $[v_x, v_x + \delta_x] \times [v_y, v_y + \delta_y] \times [v_z, v_z + \delta_z]$.

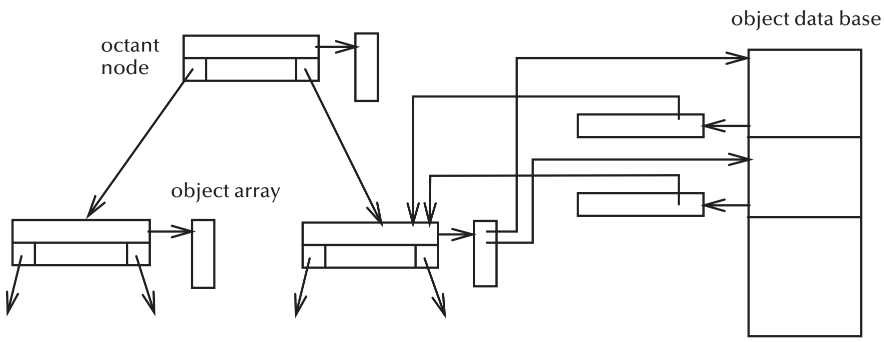


Figure 3.57: Octree data structure enhanced with octant arrays.

Finding neighbors

After all objects have been inserted in the octree, the next step is to find all "nearby" objects to a given one. This can be done very simply by just using the insertion algorithm again; except instead of inserting the object, we just read the list of "occupants" attached to each cell.

The only difference to insertion is the case when an octant is completely inside the object. With insertion, we are finished with recursion at this point. For finding neighbors, however, we have to descend further down, since there might be objects which are attached only to some octant at a deeper level.

We can do a little better by enhancing the data structure by *octant arrays*. Every object has its own octant array. This array contains references to every octant whose occupant list has got a reference to the object in question (see Figure 3.57).

When trying to find neighbors of an object, we do not have to traverse the octree from the top. Instead, we can start traversal at those nodes which are referenced by the object's octant array. (Most of them will be leaves; see below).

Moving an object

When an object has been moved by the application, the octree has to be updated accordingly. The naive way to do this is first to remove all references to this object from any octree cell, then to insert it again. This could be done by two traversals through the octree. With octant arrays used for neighbor-finding, we can remove object references faster (the insertion phase remains the same).

In most real-time applications, however, objects move only a small distance, and rotate probably a little bit (*temporal coherence*). Therefore, their new bounding box is "almost" the same as of one frame before. With two traversals for updating the octree, most of the references to an object would be removed only to be inserted immediately again by the next traversal.

We cannot speed up the object removal phase very much. All we can do is to scan the object's octant array and remove the object from all those cells which are no longer in the new bounding box.

Now we only have to insert the object in those cells which have not been occupied before. Given the "new" box B^{new} and the "old" box B^{old}, we partition the difference $B^{\text{new}} \setminus B^{\text{old}}$ into (at most) six boxes, which I will call *entering boxes* E_i:

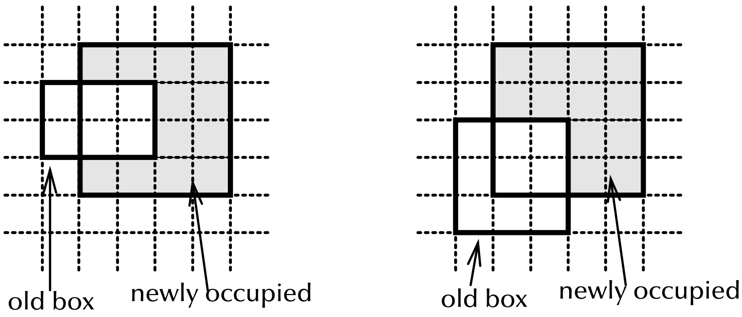


Figure 3.58: Symmetric set difference of "new" and "old" bounding boxes. The "new" part is partitioned into rectangular boxes. When moving an object, an octree traversal has to be done for only those cells which are in the box-difference.

$$
\begin{aligned}
E_0 =& [B^{\text{new}}_{\text{xmin}}, B^{\text{old}}_{\text{xmin}} - 1] \times [B^{\text{new}}_{\text{ymin}}, B^{\text{new}}_{\text{ymax}}] \times [B^{\text{new}}_{\text{zmin}}, B^{\text{new}}_{\text{zmax}}] \\
E_1 =& [\max(B^{\text{old}}_{\text{xmin}}, B^{\text{new}}_{\text{xmin}}), B^{\text{new}}_{\text{high}}] \times [B^{\text{old}}_{\text{ymax}} + 1, B^{\text{new}}_{\text{ymax}}] \times [B^{\text{new}}_{\text{zmin}}, B^{\text{new}}_{\text{zmax}}] \\
E_2 =& [\max(B^{\text{old}}_{\text{xmin}}, B^{\text{new}}_{\text{xmin}}), B^{\text{new}}_{\text{high}}] \times [B^{\text{new}}_{\text{ymin}}, B^{\text{old}}_{\text{ymin}} - 1] \times [B^{\text{new}}_{\text{zmin}}, B^{\text{new}}_{\text{zmax}}] \\
E_3 =& [\max(B^{\text{old}}_{\text{xmin}}, B^{\text{new}}_{\text{xmin}}), B^{\text{new}}_{\text{high}}] \times [\max(B^{\text{old}}_{\text{ymin}}, B^{\text{new}}_{\text{ymin}}), \min(B^{\text{old}}_{\text{ymax}}, B^{\text{new}}_{\text{ymax}})] \times \\
& [B^{\text{new}}_{\text{zmin}}, B^{\text{old}}_{\text{zmin}} - 1] \\
E_4 =& [\max(B^{\text{old}}_{\text{xmin}}, B^{\text{new}}_{\text{xmin}}), B^{\text{new}}_{\text{high}}] \times [\max(B^{\text{old}}_{\text{ymin}}, B^{\text{new}}_{\text{ymin}}), \min(B^{\text{old}}_{\text{ymax}}, B^{\text{new}}_{\text{ymax}})] \times \\
& [B^{\text{old}}_{\text{zmax}} + 1, B^{\text{new}}_{\text{zmax}}] \\
E_5 =& [B^{\text{old}}_{\text{xmax}} + 1, B^{\text{new}}_{\text{xmax}}] \times [\max(B^{\text{old}}_{\text{ymin}}, B^{\text{new}}_{\text{ymin}}), \min(B^{\text{old}}_{\text{ymax}}, B^{\text{new}}_{\text{ymax}})] \times \\
& [\max(B^{\text{old}}_{\text{zmin}}, B^{\text{new}}_{\text{zmin}}), \min(B^{\text{old}}_{\text{zmax}}, B^{\text{new}}_{\text{zmax}})]
\end{aligned}
$$

Of course, these calculations are done using integer coordinates. Mostly, 3 of the entering-boxes E_i are empty; all six are non-empty only if the box has grown but moved very little.

Then we do an insertion traversal with each non-empty of the entering boxes E_i (see Figure 3.58). This should result in much fewer cells being visited.

Implementation

An efficient implementation of the basic insertion algorithm for octrees has to take advantage of the following simple conditions:

$$
\begin{aligned}
B^{\min}_x > o^{\text{mid}}_x &\rightarrow \text{ do not consider the 4 left sub-octants at all} \\
B^{\max}_x > \mathbf{o}^{\max}_x &\rightarrow \text{ do not consider the 4 right sub-octants at all}
\end{aligned}
$$

where o^{mid} is the center of the current octant o. Similar tests with y and z yield a decision tree of depth 3. For each leaf of the decision tree we can use a different sub-octant traversal scheme which minimizes assignments (see Figure 3.59). On some architectures, this yielded a speed-up of 20%.

Including an octant's bounds and its midpoint in the node structure does not seem to speed-up the insertion. So we can as well compute the bounds of sub-octants at recursion time, which saves about half of the memory required by the plain octree.

The octree backbone, i.e., all octant nodes, are created at initialization time. They remain throughout the whole run time. This is done so as to minimize

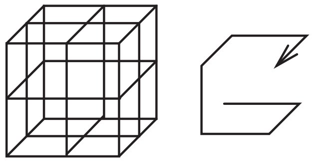


Figure 3.59: For the case where all sub-octants have to be visited, this traversal scheme minimizes assignments of octant bounds. Similar traversal schemes are used for the cases where less sub-octants have to be visited.

memory allocation and deallocation. Occupant lists are also created at initialization time, but only very small ones. Their size can grow and shrink, depending on how many objects are in an octant. However, this growing/shrinking is controlled by some hysteresis, so the number of memory reallocations is kept low. Tests showed that by increasing this hysteresis from 5 to 10, the number of memory reallocations was decreased by a factor 3. However, the overall running time did not decrease significantly — apparently, memory reallocations are fast compared to an octree traversal.

Octree results

The first test was designed to measure pointer distribution with respect to octree depth. The octree spanned a 10^3-cube, containing 100 objects, each of size 1^3 (in local coordinate system).

octree depth	#object pointers at depth (%) 1	2	3	4	5
2	0	100			
3	0	0	100		
4	0	0	0.6	99.4	
5	0	0	0.03	3.5	96.4

This means that with an octree of depth 5, say, 96% of all object pointers of the whole octree are stored in leaves. (A depth 5 octree has got $32 \times 32 \times 32$ voxels.) It seems that an octree is an efficient space subdivision only if objects are large in comparison to the voxel size.

The same set-up was used to measure memory usage and actions.

octree depth	usage (kBytes)	#mem. actions/frame malloc	realloc	free
2	8	9	78	9
3	16	114	78	114
4	264	839	55	839
5	2264	4821	14	4818

Memory usage includes the octree backbone plus all object arrays. Profiling revealed that memory allocation or freeing does not consume significant time.

Timing. Figure 3.60 shows the results of the improvements over the naive algorithm, proposed in this section. Obviously, the algorithm which visits only those nodes it really has to is the fastest. The tests were run on an SGI Onyx (R4400, 150 MHz).

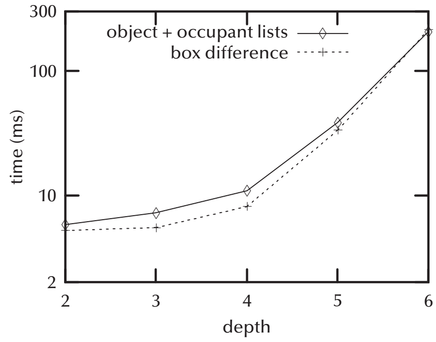


Figure 3.60: Performance gain by incremental maintaining of octrees of dynamic scenes. A depth of 0 corresponds to one cell only. Both variants use occupant lists, but one of them visits only cells which are in the box difference.

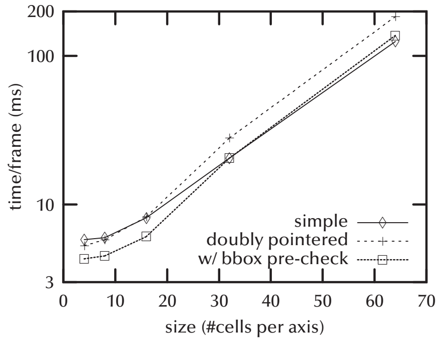


Figure 3.61: Comparison of the simple and the doubly-pointered grid data structure. Grid updating plus generation of "close" object pairs with 100 objects. The third curve is with the additional pre-check whether the new and the old bounding boxes (in integer coordinates) are the same.

3.8.5 Grids

The general idea of grids is extremely simple: every cell maintains a list of objects which are (partially) in that cell. When moving an object in the grid, we can use the same box-difference technique as above for octrees, in order to visit only those cells which have to be changed. The performance of the grid data structure is shown in Figure 3.61.

In order to reduce the number of cells an object occupies I tried to use its non-axis-aligned bounding box. In order to find the occupied cells, a stabbing algorithm was implemented. A box is stabbed by a set of parallel rays which is just dense enough. This yields a set of columns of cells. However, this method is faster only for grids with more than 45^3 cells.

Timing. When there is only one moving object among several stationary ones, the threshold of benefit of a grid is reached with smaller object numbers. The scenario here was: one moving object (20 polygons) among 500 stationary objects, each with 52 polygons (altogether 25,000 polygons). Exact collision detection was done.

without grid	1.0 msec/frame
with 8^3 grid	0.2 msec/frame

Grids seem to provide the optimum speed-up (compared to the all-pairs test) with a size of $5 \times 5 \times 5$ through $10 \times 10 \times 10$ voxels. This is in the same order as for octrees in static settings.

3.8.6 Comparison of grid and octree

Octrees and grids have been compared with each other and with the n^2-method (see Figure 3.62). The scenario is n objects moving inside a cube without exact collision detection. So objects can pass through each other, but this should not affect the timing. In order to keep the density constant as the number of objects

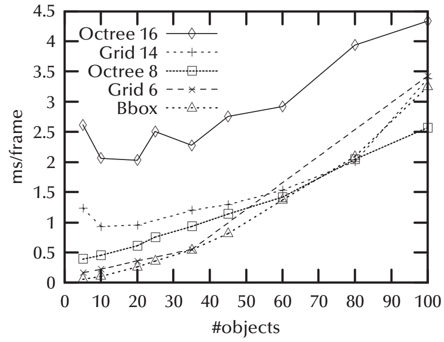


Figure 3.62: Comparison of grid, octree, and $\frac{n^2}{2}$ bounding box tests. The graph labeled "grid 14" corresponds to a grid with 14^3 cells; similarly for the other graphs.

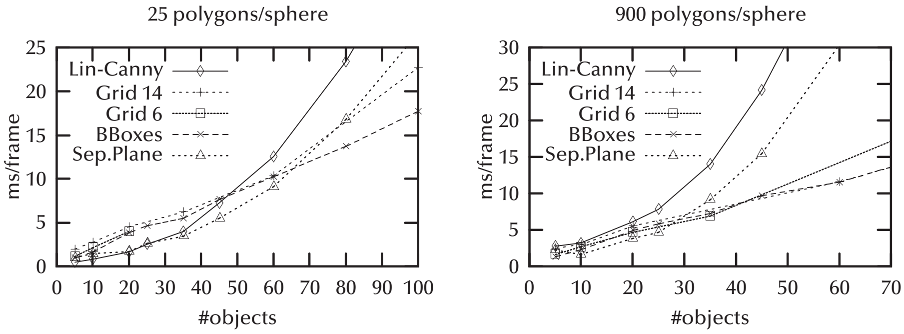


Figure 3.63: A comparison of various neighbor-finding algorithms. Exact collision detection (DOP-Tree) has been performed if a pair was found to be "close enough", and it is part of the times obtained. Scenario: n spheres bouncing off each other inside a cube. The density (defined as the number of collisions per frame) is constant through all object counts.

increases, the size of the cube is increased accordingly. Density is defined as the average number of collisions per frame.

With very small numbers of objects, the n^2-method performs better than the grid. In contrast to static environments, octrees are always less efficient than grids. This is in contrast to results obtained by [MSH+92] for ray tracing, which suggest octrees in favor over grids.

I believe that, in general, a space indexing data structure for dynamic environments has to be much less complex than a data structure for static scenes. Complex data structures tend to be "precomputation-biased": the computational burden is shifted to their updating, so that queries can be more efficient. In highly dynamic environments, I believe, the simpler data structure is superior to complex data structures in terms of speed, because they need to be updated constantly.

3.8.7 Comparison of grid and separating planes

It is not trivial to compare grids and the separating planes algorithm. It is easy, of course, to compare them to each other by themselves. However, since we want to use them for nearest-neighbor finding, there are a lot of side effects.

One characteristic property is *complexity*, in the context of nearest-neighbor finding: grids are essentially in $O(n)$, while the separating planes algorithm and I_collide are in $O(n^2)$ ($n =$ number of objects). On the other hand, there is *quality*: by this we understand how often a neighbor-finding algorithm passes a pair of objects to the exact collision detection although they do not collide. Obviously, the grid has a lower quality than convex hull based algorithms.

Another property is the *dependence* on object complexity. The grid is independent of object complexity (unless tight-fitting bounding boxes are used), while the convex-hull-based algorithms depend on it, because usually the complexity of the convex hull depends to some degree on the number of vertices of the objects.

So, if the number of polygons is large or the number of objects is large, then a grid is probably more efficient. On the other hand, if the number of objects is small or each object has only few polygons, then the separating planes algorithm is probably better. Exactly where the break-even point is depends, again, on several factors, such as how fine the grid is, and how fast the exact collision detection algorithm is.

Figure 3.63 substantiates these considerations. Clearly, there is a break-even point, which depepnds on the number of polygons per object. The *BBox* graph shows the performance of $\frac{n^2}{2}$ bounding box checks. Obviously, this brute-force method has a very small constant factor, so that it will exhibit the n^2-characteristic only with many more objects than 100.

Depending on the number of objects, either the separating planes algorithm alone should be used, or it should be combined with a grid, so as to form 2 stages of the collision detection pipeline (which could be considered a filtering pipeline). The idea is that the grid serves as an $O(n)$ prefiltering stage while the separating planes algorithm performs a more precise proximity check on those pairs that have passed the grid. That way, I assume, the good properties of both neighbor-finding algorithms can be combined.

3.8.8 Combining grid and separating planes

As the previous section has shown, the grid and another algorithm based on convex hulls, such as the separating planes algorithm, should be combined to perform neighbor "filtering". That way, the grid ensures basically $O(n)$ performance, while the separating planes algorithm ensures that only those pairs of objects are checked for collision which are very close to each other.

This scheme has not yet been implemented in the collision detection module, but I am planning to do so in the near future.

3.9 The collision detection pipeline

A fundamental concept in computer graphics is the *pipeline*. It is interesting to realize that this concept is appropriate for the implementation of many modules in a computer graphics system. Probably the first pipeline identified was the rendering pipeline ([AJ88]), which is still valid, although somewhat modified, even in modern architectures [Bar97, MEP92]. Other pipelines are the visualization pipeline [Fel95] and the haptic pipeline [Zie98].

Collision detection can be regarded as a pipeline of successive filters. The input is a set of objects (namely all objects in the scene graph), while the output is a set of polygons (namely the ones overlapping). The difference to the rendering pipeline is that in the collision detection pipeline pairs of entities are passed down.

In previous sections I have described all parts necessary to implement a collision detection module. In this section I will briefly describe the collision detection pipeline.

I will not discuss the issue of dynamically pre-fetching objects and auxiliary collision detection data for walk-throughs (and similar applications), if the complete environment does not fit in main memory. A system solving this problem has been described by [WLML99].

The collision interest matrix. In a VR system the collision detection module is used by many different high-level modules (like interaction manager, inverse kinematics, rigid body dynamics, etc.). Each module is interested in different collisions. In addition, there are usually a lot of object pairs which no module is interested in. Finally, sometimes a module just "knows" that certain pairs of objects cannot collide at all, or that a pair of objects will collide all the time.

The collision detection module, on the other hand, needs to save certain data with each pair of objects to be checked for collision, such as callbacks, cached information obtained during the last collision check, time stamps, etc.

Conceptually, we need a collision interest matrix to hold all these per-pair information. The data structure is usually a diagonal matrix.[31] Other data structures could be hash tables, sparse matrix, or lexicographically ordered lists. In my experience, however, a diagonal matrix is satisfactory.

The pipeline. The data flow in the collision detection pipeline is conceptually as follows (see Figure **??**). The object handler tells the collision detection module about any objects that have been moved by any other module. These are added to a list buffer. Eventually, the buffer is emptied, and any global neighbor-finding data structure is updated according to the new positions. Then, a list of pairs is generated from the collision interest matrix. This list of pairs is filtered by one or more neighbor-finding algorithm (e.g., grid and/or collision detection of their convex hulls).

This produces an intermediate list of pairs of objects "close" to each other. For each of these an exact collision detection is performed. This finally produces a list of colliding objects.

Finally, for each colliding pair one or more callbacks are executed.

[31] Of course, only for occupied cells memory is allocated. All unoccupied cells just hold one pointer.

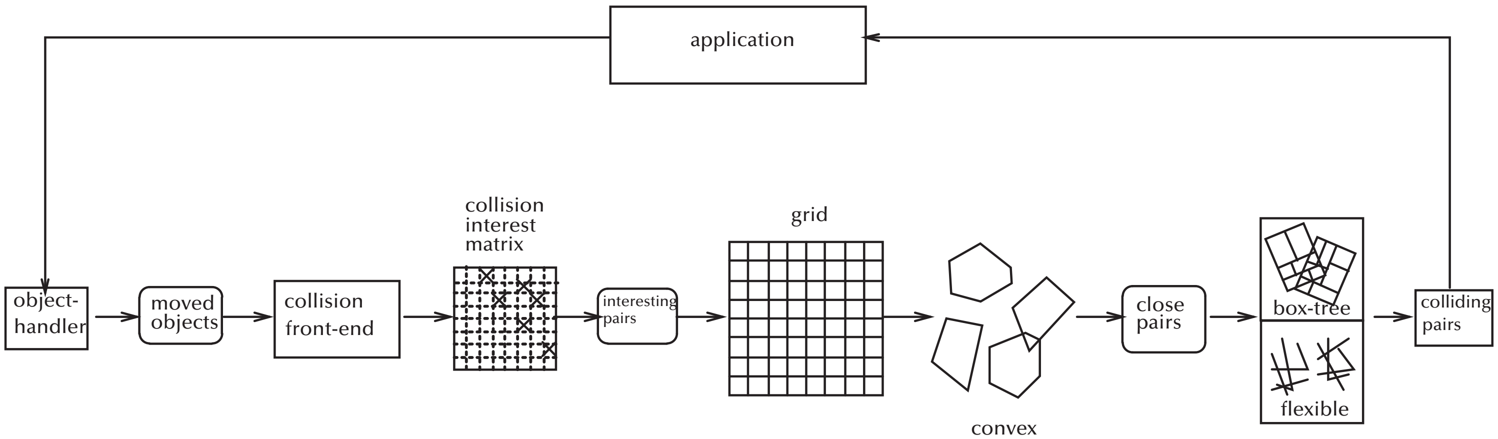


Figure 3.64: The collision detection module implements this pipeline.

3.10 Parallelization

In general, there are several models of parallelization. One of them is the *producer-consumer* model. A pipeline can be viewed as a sequence of producers and consumers, each consumer being the producer for the next one.

Another model is the *master-slave* model. Here, one master doles out pieces of work to the slaves. The slaves return the results to the master. The master is responsible for balancing the work-load.

Yet another model is the *anarchist model* (such as Linda [Car98]). The idea is that every producer puts work-pieces in a work-pool. The workers take a piece out of this pool. When a piece of work is finished, they put the result back into the pool. [32]

These models are mostly helpful for designing coarse-grain and distributed parallelization. Basically, they are ways to think about load-balancing. On the fine-grain and shared-memory level, a different perspective is more helpful. On this level, we have a continuous spectrum of models.

Given a task, which can be partitioned into arbitrarily-sized "chunks". If we partition it into as many chunks as there are processes, then we have *static load balancing*. Assume that we partition the task into many small chunks, so that we have much more chunks than processes. Then, each process can work on as many chunks as possible, until all chunks are finished; this is *dynamic load-balancing*.

At the other end of the spectrum, there is what I call the "stride-based" approach. When all the chunks have elementary (or, atomic) size, then a static load-balancing scheme is needed, otherwise synchronization overhead would be overwhelming. The scheme works in an "interleaved" fashion: all elements are stored in an array; then each process takes elements id, $n + \mathrm{id}$, $2n + \mathrm{id}$, $\ldots$ (where id is the process' ID, starting from 0).

I have implemented parallelization on both levels coarse-grain and fine-grain. In the following I will discuss these efforts in more detail.

3.10.1 Coarse-grain parallelization

The coarsest-grain parallelization is concurrency between the collision detection module and all the other modules. The appropriate model is producer-consumer, where all modules are both. This will be discussed in more detail in Section 3.11.4.

If one thinks about the pipeline-nature of the collision detection module (see Section 3.9), the usual pipeline parallelization (as implemented in CPUs) comes to mind. However, as we all know from CPUs, it is crucial that the pipeline be balanced. With the collision detection pipeline, this is definitely not the case. All the heavy work is in the back-end, while the space-indexing stage has to do very fairly "light-weight" work (usually), and all other stages are more of a organizational or glue nature.

In addition, it would not be trivial to parallelize the grid. Access to grid cells would need to be exclusive. If synchronization would lock the whole grid at once, then most processes would be waiting all the time. On the other hand, providing each cell with its own lock would probably make access to cells slow.

Therefore, I have not parallelized the pipeline.

[32] Internet-wide distributed projects like "Seti@Home" or "Crack RSA" seem to share characteristics with both the master-slave and the anarchist model.

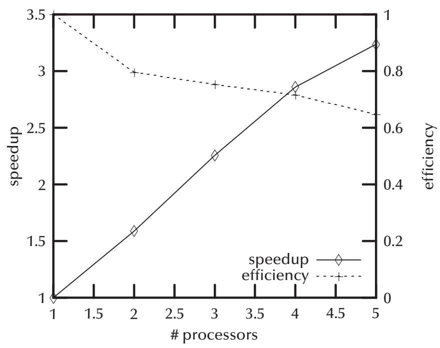


Figure 3.65: Coarse-grain parallelization of the back-end yields good speed-ups, if there are enough object pairs to be checked with each collision frame.

The back end, on the other hand, is almost trivial to parallelize. After the list of possibly colliding pairs has been filled, each process takes out one pair and checks it for collision. Here, load balancing must be completely dynamic, because collision detection times can vary between pairs by orders of magnitude (depending on their complexity and relative position).

A parallel back end can yield almost linear speed-up, because there is almost no synchronization overhead (see Figure 3.65[33]). Of course, this can be achieved only if there are enough pairs in list of possibly colliding objects *and* if those pairs are really close enough. It seems that this is true only for very dense environments.

Pretty often, only one pair "survived" the grid, the collision interest matrix, and the convex hull test. In order to take advantage of several processors in that case, fine-grain parallelization is needed.

3.10.2 Fine-grain parallelization

The algorithm of Section 3.3 lends itself very well to fine-grain parallelization. Various phases can be parallelized: the collect-phase, transformation of arrays of vertices, and the outer loop.

It is very important to make sure that a parallel implementation does not invalidate the cache too often (see Section 3.11.3)![34] So, a "chunk-wise" parallelization is often best. On the other hand, polygons and vertices tend to

[33] Measured on a 6× 194 MHz R10000.

[34] Here is a live example. In my first attempt of fine-grain parallelization of the BBox-pipeline algorithm (see Section 3.3), I parallelized the transformation of the vertices into world space like this: each process would *stride* through the vertex array. The stride length was the number of processes. Unfortunately, two processors needed about twice as much total time, although profiling showed that no time was spent at barriers!

Now, the memory layout of the vertex data structures is such that the vertex' coordinates in local space and those in world space are always next to each other (they are both in the same struct).

The problem with the stride-through approach, together with this special memory layout, is, that whenever a process writes to memory, the caches of all other processes become invalid in that area. Because of the striding, many other processes are *bound* to read from that area. And since all processes would stride from bottom to top through the array, a lot of cache misses occurred!

A much better approach here is the "chunk-wise" approach, which assigns each process one range of vertices up front.

In this example, the "stride-wise" parallelized transformation of about 10,000 vertices took 26 milliseconds, while the "chunk-wise" approach took only 10 milliseconds! (2 processes, so each took 13 and 5 milliseconds, resp.)

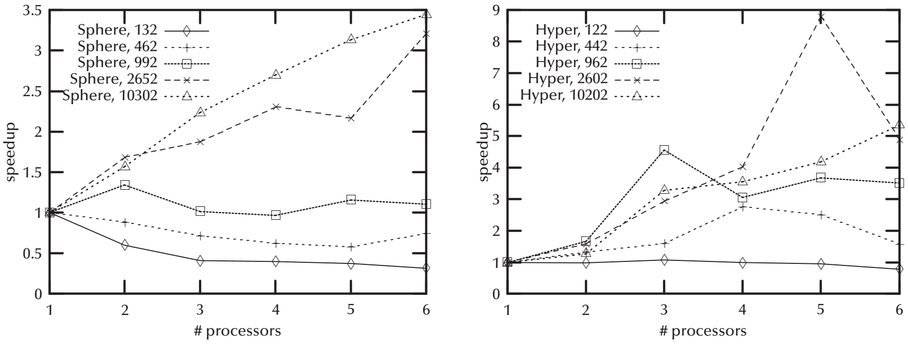


Figure 3.66: The algorithm presented in Section 3.3 lends itself well to fine-grain parallelization as the speed-up shows. On the left, two spheres were used, on the right two hyperboloids were used.

be stored contiguously in memory, in the sense that polygons close in memory tend to be close in space, too (this is a consequence of how tesselation and striping algorithms work). So, when checking polygons for intersection, "stride-wise" parallelization is better. With chunk-wise parallelization, the load would be poorly balanced, because some chunks will be completely outside the bounding box of the other object, and some chunks will be completely inside.

Although parallelizing one algorithm is fine-grain parallelization, this works well for the BBox-pipeline algorithm. Figure 3.66 shows the speed-up with two different objects. It is not quite clear to me, why there is practically no speed-up with the "nice" sphere object. With less "nice" geometry (such as a hyperboloid), parallelization yields fairly good speed-ups.

It remains to be seen whether parallelization of the Boxtree algorithm (see Section 3.5.4) or DOP-tree algorithm (see Section 3.5.9) is feasible.

3.11 Implementation issues

In this section I will describe some of the more technical issues that have occurred during my implementing the collision detection module.

This is not the end of the story. At one point, the non-parallelized version of the algorithm calculates a tight bounding box. Profilings showed that this took an insignificant amount of time. So, in my first attempt, I didn't parallelize this section of the algorithm, resulting in

```
start parallel section
   transform chunk of vertices
end parallel section
calculate tight bounding box
```

Calculating the bounding box took 9 milliseconds total, while in the non-parallelized version it took only 6! Again, this was caused by cache misses, because after the parallel section, the cached vertices of process 0 (which participated in the parallel section) were invalid except for its own chunk. Having realized that, the cure was obvious: move the calculation of the bounding box into the parallel section:

```
start parallel section
   transform chunk of vertices
   calculate tight bounding box for own chunk
end parallel section
merge bounding boxes
```

3.11.1 Requirements

During my work on the collision detection module and its application (such as described in Sections 4.5.4 and 5.2), several requirements on such a module have emerged.

The collision detection module must provide an API, so that modules using collision detection can optionally specify the order in which the collision detection module executes callbacks. Reason: Sometimes a module has registered several pairs of objects for collision detection. It can happen that more than one pair is colliding during the same collision frame. However, if a certain pair is colliding, then all other collisions are "uninteresting" to that module. While each module could implement this kind of feature itself, the right place to implement it is the collision detection module for several reasons.

Each object registered with the collision detection module must have a flag whether or not it could move, and whether or not it is flexible. That way, the collision detection module can select the algorithm appropriate for each pair of objects. In addition, each pair of objects must have a flag whether or not some module wants to know *all* intersecting polygons. Finally, each pair must have a flag telling if some module wants the collision detection module to calculate the minimal distance. In that case, a different algorithm must be used.

It should be possible to disable temporarily any collision checks with an object. Still, the collision detection module should remember whether or not the object has moved during its "hibernation", so that when the object is activated again the collision detection module will check it, even if it has not moved after its activation.

A module using collision detection should be able to disable collision callbacks of any other module for an object. This is necessary when a module needs several collision queries (such as physically-based simulations) which are not of interest to other modules, because they represent a kind of "intermediate" state.

There are different types of requests to a collision detection module (collision detection query, additions, callbacks, etc.). These must be entered in a *single* queue in front of the collision detection module's front-end.

3.11.2 Time-stamping

One does not want to compute things more often than necessary. Often in computer graphics, the "things" are arrays of entities such as points, normals, face bounding boxes, etc. With collision detection, often only some of the entities of an array need to be computed or transformed. So, the idea is to store the derived results for those entities being actually computed/transformed, and later consult a flag (each entity having its own flag) to find out whether that result has been computed already.

The problem is that whenever the input changes, all flags need to be reset. With collision detection, in almost all cases this happens when an object has been moved, when it changes geometry, or just before the next collision loop. However, it is way too expensive to visit all flags of an array of entities which have become invalid.

Here, time-stamps help. We introduce a counter for each array of entities (e.g., one counter for all face normals of an object). This counter is the "timer" for its associated array. When all of the entities of an array are changed, we simply increment that counter. When we transform/compute one of the enti-

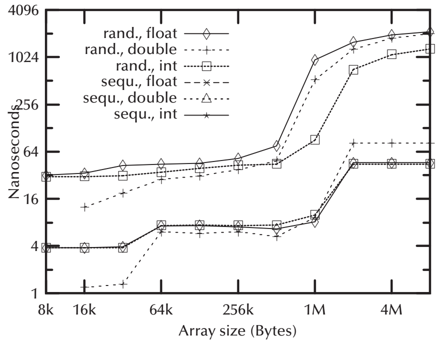


Figure 3.67: Memory access patterns cannot be neglected with the design of collision detection algorithms.

ties, we copy the value of the counter (the global "time") to the entity's flag. Two examples are given below.

For instance, the algorithm of Section 3.3 spent 30% of the total CPU time with clearing all the flags for face normals, without the time-stamping technique!

3.11.3 The CPU cache

In my experience, a careless implementation of collision detection algorithms can mar performance completely. In several cases, I have seen a performance gain by a factor 2 when the implementation was improved.

With some algorithms, even during their design the architecture of the machine has to be taken into account. I have noticed that during profiling of the algorithms presented in Section 3.5.5 and in Section 3.5.11.

One of the most important features of computer architectures is the *cache*.[35] It has great impact on "bus-intensive" algorithms, i.e., algorithms reading and writing data from/to memory very often compared to the time these data spend in CPU registers. If such algorithms show a linear memory access pattern, then a cache improves performance a lot. However, for algorithms with a random pattern often accessing memory parts "far away" from previously accessed parts, a cache does not help and might even slow down performance.[36]

Therefore, I investigated the impact of the primary-level and secondary-level data caches on R10000 architectures. The instruction cache, of course, could have considerable impact, too. However, in my experience, implementations of collision detection algorithms show a naturally high code locality, so they already benefit from the instruction cache.

The cache test I performed involves a simple operation on a large array. The size of the array is varied, so that a larger or smaller part of the complete array fits into the caches. For each array size, a completely random and a completely linear access pattern was executed on the array.

[35] Another important question is whether or not the CPU's floating-point unit is as fast as the integer unit.

[36] An early version of the algorithm of Section 3.5.11 was actually consistently slower than the algorithm of Section 3.5.9. Profiling revealed that this was due to a highly non-linear memory access pattern, whereas the algorithm of Section 3.5.9 has a much more linear access pattern since it is a depth-first tree traversal.

The tests were carried out on a 194 MHz R10000 equipped with with 32 KB primary data cache and a 1 MB secondary unified instruction/data cache size. Figure 3.67 shows that there is a considerable performance loss when the primary cache's limit is reached, and that there is an even higher performance loss when the secondary cache cannot hold the array anymore. The time shown in the plots is for two accesses to the array, one load and one store.

This proves that with current hardware architectures the RAM access time and the memory access pattern of an algorithm can no longer be neglected. Especially when comparing the access times with other operations, for instance, any float arithmetic takes approximately 30 nanoseconds. See also Section 3.10 for a discussion of the effect of cache invalidations on parallelized algorithms.

3.11.4 Concurrent collision detection

Collision detection is one of the major CPU "hogs" in many virtual prototyping applications, particularly those which do some sort of physically-based simulation. So, in VR systems, this module must run concurrently to all the other processes. Otherwise, the VR system's response to the user's movements and the frame-rate would be seriously impaired, which would disturb the feeling of immersion.

The model chosen in my implementation is the classic *producer-consumer* model, except that the buffer in-between is not, as usually, a FIFO but a double-buffer, for reasons explained below. The conventional producer-consumer model has to be modified as follows. In VR systems, the producer must never be kept waiting. So, should the buffer become full, entries must be thrown away. In my excperience, this has never happened, though. Second, the collision detection module does not wait for the front-buffer to become full; it starts checking collisions as soon as the back-buffer is empty and all collision callbacks have finished.

With concurrent collision detection, the programmer of collision detection callbacks must be aware that the callback itself is part of the collision detection process (although the code is part of the application module). As a consequence, should the callback communicate with the rest of the application module via shared data structures, these probably must be implemented as double-buffers, too.

Chapter 4

Interacting with Virtual Environments

I hear and I forget.
I see and I remember.
I do and I understand.
CONFUCIUS

Interaction in general can be approached from two points of view [FvDFH90]:

- *Task-* or *feature-oriented*.
  Some basic recurrent tasks in 2D as well as 3D are selection, grasping, creating, and destroying objects, text input, etc. In 3D, navigation is another basic task.

- *Technique*-oriented.
  This point of view focuses on the interaction paradigms and metaphors being employed to realize or implement the interaction task or feature.

  To some degree, the interaction paradigm depends necessarily on the input devices being used — not every paradigm can be implemented with every device, and some devices suggest or imply a certain paradigm.

  For instance, in 2D the paradigm of choice for *selection* is the *pointer*, which can be implemented very naturally with a *mouse* and its mouse buttons. Of course, other devices could be utilized, for example the cursor keys of the keyboard, or eye tracking combined with speech recognition.

Interestingly, in 3D the distinction between the feature-oriented and the technique-oriented point of view becomes much clearer. The "3D button" *feature* (see Section 4.5.1) can be realized by several different *techniques*: collision detection, ray-object intersection, or cone-object intersection; the ray might emanate from the finger, or from the eye through the finger; etc.

4.1 VR devices

Virtual reality basically was "born" with the invention of new input and output devices and their becoming affordable by research institutes. Of course, some applications and visions existed before that: Flight simulators clearly belong to the field of VR, although nobody called them that way (and still nobody does). As early as 1965, when computer graphics was just born, Ivan Sutherland expressed his vision in the often-cited paper "The ultimate display" [Sut65], where he proposes ideas such as eye tracking, force-feedback, and speech recognition. His ultimate display is essentially a "holodeck". Probably, he was also the first to realize a head-mounted display (HMD) [Sut68]. Also, the military utilized some of the "classic" VR devices (HMDs for exam-

type	advantages	disadvantages
fish-tank VR	best resolution and least distortion; familiar and easy-to-use; fairly inexpensive.	low immersion; stereoscopic violation because of clipping; small range of user's movements.
head-coupled VR	best immersion because of large field-of-view, all-surrounding view, and almost no stereoscopic violation; fairly large range of user's movements; affordable.	either heavy or low resolution; large distortion because of wide-angle optics; not easy-to-use (intruding interface).
projection-based VR	high resolution, large field-of-view; high degree of presence, because user can see himself; easier to share; easy-to-use.	needs more graphics pipes for more walls; possible stereoscopic violation, because user's limbs always occlude virtual objects; requires a lot of space; not so easy to maintain.

Table 4.1: The different characteristics of input and output devices imply different types of VR, distinguished mainly by the (characteristics of the) output device.

ple) a long time before they became widely available and affordable. However, only when devices such as trackers, HMDs, and gloves became available and affordable the field of VR came into being and attracted a lot of research.

As of 1998, it is clear that current VR devices are still in their infancy [CN97]. It is also understood that there is no single combination of devices which is best for *all* applications. Certain characteristics are inherent to the devices, at least they will prevail for a long time. These imply several different types of VR, distinguished mainly by the (characteristics of the) output device (see Table 4.1).

By fish-tank VR I understand desktop-based VR, i.e., a stereo monitor and shutter glasses. Head-coupled VR is based on the use of HMDs or booms (possibly strapped to the head). Projection-based VR utilizes 1 up to 6 stereo screens; this includes devices such as the workbench (1 wall as a table [Ros97]), 2-screen workbenches, panorama walls (curved or flat), and the cave (3 to 6 walls) [CSD93, Dee92]. In almost all environments, the user's head is tracked, mostly the hand, too, and sometimes the user wears a glove or other input device.

4.1.1 Input device abstraction

Although the author of virtual environments usually wants to specify the input devices to be used for interaction,[1] my experience has shown that an abstraction

[1] For instance, the author defines the "sentence" a user has to speak in order to invoke a certain action, and the author wants to specify exactly which tracking sensor is to control the user's hand.

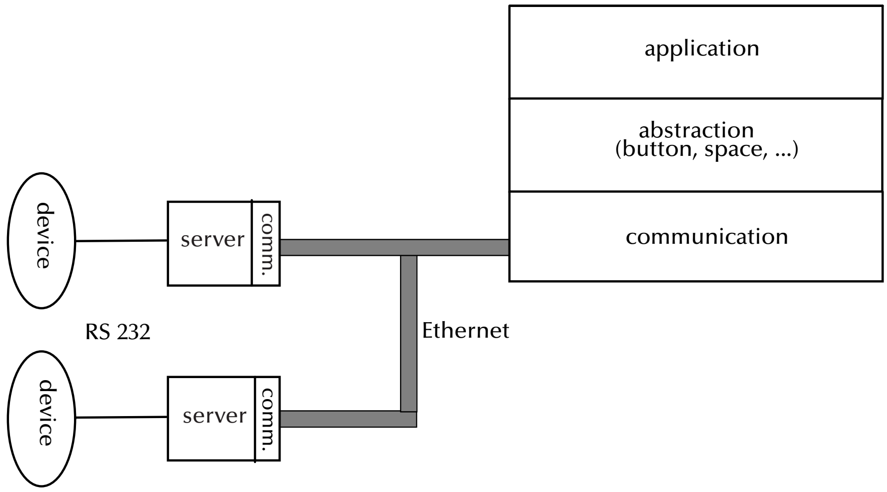


Figure 4.1: The framework for integration of input devices consists of an abstraction layer, a communication layer, and a server layer. That way, devices can be connected to any host in the LAN, and development of the interaction handler is simplified.

of input devices does help with the implementation of a VR system [Fel95, FSZ94]. In addition, an architecture which allows flexible configuration of the hardware has proven to be extremely helpful if not necessary for development. I will call this module *device handler*.

I have implemented such a framework and abstraction layer. It consists of two parts: the abstraction layer itself, and a set of device servers which provide basic services for talking to the devices. Those two layers are connected to each other by a communication layer, e.g., via ethernet, (see Figure 4.1). The servers are separate applications running concurrently and asynchronously as fast as they can. This helps reduce system lag. In addition, they can perform CPU-intensive data processing, such as filtering and correction (see Sections 4.3.1 and 4.3.2).

Potentially, a server can run on any host in the LAN. This can be very helpful for development. Of course, for serious VR applications all servers should run on the same (multi-processor) host where the main VR system is running, too. In that case, communication should be performed via shared memory, so that the renderer can read the latest viewpoint data right before it starts culling.

The abstraction layer supports *logical input devices*, so that the exact type of physical input device needs to be known only at very few places in the overall VR system. In my experience, it is sufficient to provide the following logical devices: button (binary), space (6D), and hand (flex vector). Logical buttons can be mapped on physical buttons (keyboard, mouse, spacemouse, etc.) and on speech recognition (see Section 4.2.2), spaces can be mapped on trackers, spacemouse, mouse, etc. In general, logical buttons will be used to trigger an action, while spaces will be used for all kinds of continuous motions.

Devices like location (3D), orientation (3D), value (1D), choice (1-of-n), or picker are unnecessary. Location and orientation can always be extracted from a space, and there is no benefit in implementing those "sub-devices". A picker

Of course, an author has to provide alternatives for different hardware (e.g., fully immersive or desktop).

is basically used for selecting objects in the scene graph. It is not directly connected to any input device (see Section 4.5.2). In addition, a VR system might want to use different metaphors for selection in different modes (e.g., our system uses a different metaphor for selecting annotation markers than for selecting parts). So, experience has shown that from a system architecture point of view selection should be implemented in the interaction module, which has been done in our VR system.

It has been argued that the coordinates of 6D physical devices should be normalized by the server before delivering them to the application (e.g., in the range $[\pm 1]$; [Fel95]). However, in my experience this is not a good idea. There are basically 2 types of 6D devices: *absolute* and *relative* devices. Absolute devices provide absolute coordinates, relative to some origin in the real world; such devices are electromagnetic and optical tracking systems, the boom, etc. Relative devices do not have an origin in real world; they include spacemouse, spaceball, mouse, etc. For relative devices, the raw output is converted into absolute coordinates by integration. Therefore, there is no canonical way to convert those absolute coordinates into a normal range. For absolute devices there would be a canonical way.[2] However, this is different for each device; it can be different even for the same device but with different configuration (e.g., a long-range transmitter instead of the standard one). Eventually, the (dimensionless) coordinates must to be converted back to meaningful units (e.g., millimeters for manufacturing applications). So, the application would need to know about the exact type of device at many more places.

The abstraction layer should provide both relative and absolute logical devices no matter on which physical device they are being mapped. So, the application should be allowed to request a relative or an absolute space device, no matter whether it is mapped on a spacemouse, say, or a Polhemus tracker. This is very convenient, since for navigation both types are needed, depending on whether the space is used for steering the cart or the camera (see Section 4.4). The integration over time necessary to convert relative into absolute data should be done by the servers, because they can run with a constant, known "loop rate". It could be done by the abstraction layer, but temporal aliasing will be much more noticeable, because the frame rate (or "loop rate") can vary by a factor 3 or more. Differentiation (computation of deltas) should be done by the servers, too; however, the abstraction layer must accumulate all deltas until the application polls the (relative) logical device. The communication layer cannot do that, because, in theory, several logical devices might be mapped on the same physical device, but the application might choose to poll the logical devices at different frame times.

It has been proposed to implement a *polling mode* as well as an *event mode* for logical devices. When in event mode, a logical device executes a callback whenever it receives new data from a server, while in polling mode it just waits to be polled by the application and then returns the latest data. I have implemented and used both modes; however, for the sake of a uniform design of the event generation mechanisms in the interaction handler (see Section 1.1), the polling mode should be used only. It might be helpful to provide the event mode for other, less complex, applications.

In order to keep communication overhead to a minimum, servers transmit data only when they have changed. For continuous data, a threshold can be

[2] Divide the coordinates by the maximum range of that device.

specified which tells when data are to be considered sufficiently different from old ones.

It has turned out that fault-tolerance (in a certain sense) is an important issue here. Many things can go wrong at start-up time or during run-time: the device configuration file is wrong; the device does not respond, because it was left in a bogus state by the application before; a data record got scrambled on the serial line from the device to the host, or the host dropped a part of the record (this does happen); the device did start fine, but later it quits.[3] Therefore, each server must be able to handle all of these conditions. This requires, that it can detect such a situation, and either handle it gracefully (in the case of drop-outs), or give appropriate feedback to the device abstraction layer. In the case of a faulty device configuration file, the abstraction layer should provide feedback to the user and present a configuration editor. Then, the server can re-read the configuration file, and try again; or, the device handler has to start the server anew on a different host.

Finally, the current implementation also can provide visual feedback for some devices. For instance, the logical device handling glove input can render a virtual hand, and a logical button device can be mapped on a graphical object. However, in my experience this has turned out to be unpractical for various reasons. From a system design point of view, this is not an elegant solution, because it makes the device handler dependent on the object handler and renderer, while it is really a peer module.[4] In addition, it is less flexible when more complex simulations are to be done with the glove input in order to determine the position of the virtual hand (see Section 4.5.3). Therefore, the device handler will be implemented as a strictly input module in the redesign of our VR system.

4.1.2 The data pipeline

There are (at least) two streams of data in any VR system: data generated by local input devices (see Figure 4.4) and data coming in from other participants. These data are of two kinds: discrete and continuous. Eventually, both streams of data will be output to the display and/or other participants.

There are many sources of lag in both streams: latencies in input devices, the network, filtering, expensive computations, video refresh rates, etc. All latencies can build up to large lags in the feedback to the user, which could cause "simulator sickness" [Dit97] or impair user performance.

4.1.3 Dealing with lag

Several general techniques and strategies have been devised to overcome the lag in the pipeline.

With continuous data, one can try to predict future data by extrapolation of current and past data. In some cases, this can be combined with filtering (see Section 4.3.1). Of course, the predicting algorithm needs to know the time delta it has to "see" into the future. This is easiest if that delta does not change, i.e., if the frame rate of the renderer is constant over time. This can be achieved by various level-of-detail techniques [Red96, Tur92, LT99, FS93]. Of course,

[3] This happens with Ascension systems fairly often, in particular if one of the sensors gets too close to heavy metal. But even under perfect conditions, this system quits in about 2–3%.

[4] The device handler should be strictly for input, while the renderer is strictly an output module.

data kind	lag handling	network	data structure
continuous	"better never than late"	UDP	shared memory
discrete	"better late than never"	TCP	queue

Table 4.2: The handling of lag in the streams of the two kinds of data in a VR system.

there are techniques to estimate the time to render the next frame. However, introducing such feedback loops for all places where prediction is needed complicates the overall system design, especially if they have to go back through some network. Prediction of the continuous data is complicated especially if there is a network between renderer and predictor, because this would add more uncertain latency to the estimated time the next continuous data are due.

Another way to deal with delayed continuous data can be applied if they can be generated faster than the frame rate. This is called *better never than late*, i.e., whenever the renderer is ready to draw another frame, only the most recent datum of the continuous stream will be used; all earlier data will be thrown away. This scheme can be applied to tracking data, continuous transformations, etc. The rule of thumb is that if the data can be sent over UDP, then this scheme can be applied to get the most up-to-date images.

With discrete data, the situation is different: here the principle *better late than never* is prevalent. This is true especially for data which bear a "trigger" semantics, i.e., data which generally switch some action or properties on or off. The analogy from a IP point of view is that these are data which must be sent via TCP. See Table 4.2 for a comparison.

4.2 Processing input data

Some input data sent from device servers need to be processed further in order to be useful to the interaction manager. Two cases are posture data and voice recognition data.

4.2.1 Posture recognition

A dataglove has traditionally been used to give commands to the computer through some sort of "sign language". This consists of a set of *postures*, if only one data set at a time is considered, and it is a set of *gestures*, if a sequence of postures is considered. Sometimes, postures are referred to as *static gestures*, whereas gestures are called *dynamic gestures*.

Postures can be recognized fairly reliably, while the recognition of gestures bears the same problems as that of natural speech recognition, such as noise, different time scaling, and different amplitude for the same gesture.

From my experience, the number of postures being used in a VR system should be kept at a minimum and as intuitive as possible [Zac94a] (two for navigation, one for grasping, and one for opening/closing a menu, say, seems about the maximum for an occasional user).

For the same reason, I do not think that dynamic gestures should be used in VR systems to be employed in manufacturing industries.

Therefore, I will consider only static postures in this section. From the discussion above it is clear that gesture recognition to be utilized in a traditionally CAx-based environment should fulfil the following requirements:

- Robustness, in the sense that postures are recognized correctly, only these are recognized, and transitory postures do not trigger any action;
- User-independence;
- Training is acceptable at most once per glove.

In order to reduce training, the glove server performs self-calibration, by mapping joint values on a predefined range. Since the range of raw joint angles can vary significantly across users for the same glove (and even from day to day for the same user), the server constantly keeps the range of raw joint values. This range is adjusted as new values are read. In order to avoid excessive "stretching" of this range by outliers, the range is narrowed gradually over time, if the minimum/maximum has not been observed again.

Different postures can be pretty close to each other in posture space. This could cause a "transitory" hand posture to be recognized, which in turn might trigger some action unintentionally. This is even more a problem when postures are defined in the interior of the posture space. A solution is to create a trigger only when the same posture has been recognized for a certain duration (half a second seems to be a good value).

Postures can be defined as ellipsoids in $\mathbb{R}^d$ (where $d = 18$ or $d = 22$, typically). Then, they can be recognized by a simple point-in-ellipsoid test

$$\sum_{i=0}^{d} (\frac{f_i}{r_i})^2 \leq r^2,$$

where $\mathbf{f}$ is the flex vector. In order to save CPU time, other norms could be utilized as well, for instance the l^∞-norm.

Experience showed that this approach still required each user to train the recognition for himself. Plus, recognition was not as reliable as necessary for use in every-day work.

Another approach turned out to meet the requirements set forth above. It exploits the fact that all postures our VR system understands are located near the "border" of $[0, 1]^d$, i.e., some of their flex values are (almost) maximal or minimal[5] (see Figure 4.2).

The idea is to discretize a flex vector $\mathbf{f} \in [0, 1]^d$ into $\mathbf{f}' \in \{-1, 0, +1\}^d$ (0 means the corresponding flex value is indeterminate, i.e., neither close the minimal nor maximal value). Then, $\mathbf{f}'$ is compared with each sample $\mathbf{g}$ from the database. If there is one for which

$$||\mathbf{f}' \cdot \mathbf{g}|| = ||\mathbf{g}||$$

then $\mathbf{f}'$ is recognized. The database should be constructed such that $\forall \mathbf{g} \; \nexists \mathbf{g}' :$ $||\mathbf{g} \cdot \mathbf{g}'|| = ||\mathbf{g}||$.

Although I have not performed a formal user study, our experience from several years is that this approach is extremely robust and reliable. Our database usually consists of 5 postures, which has been sufficient for all applications.

[5] In fact, even in our every-day life there are only very few postures which are in the interior of that posture space.

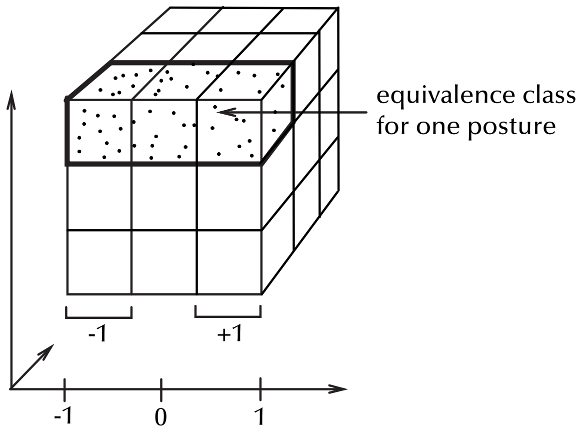


Figure 4.2: Joint value vectors of valid postures tend to be close to the border of posture space.

The database needs to be constructed only once. It suffices to calibrate the glove once for one user's hand; starting from that, the glove server does an automatic flex value adjustment on-line, in order to make the range of flex values close to $[0, 255]$ [ZLB+87]. Thus posture calibration has never been necessary for other users.

Other approaches. Posture recognition has been a field of research at least since data gloves have been available [DS77, ZLB+87, SZ94]. Biomechanical analysis of the human hand had been performed earlier [ACCL79].

Automatic scaling and filtering transitions have been described by [ZLB+87].

Back-propagation networks (perceptrons) [VB92] are quite well suited for gesture recognition. Still, a good glove calibration is essential; also, each user should have their own gesture definition data or neural network. Findings by [WS99] suggest that it might be possible to achieve user-independent gesture recognition by training a 3-layer back-propagation network with training sets of several different hands.

An interesting approach is the application of fuzzy set theory [TS98]. They represent postures by vectors of polar coordinates (each finger tip is represented by (ϕ, θ, r) relative to its proximal joint), which are discretized in $\{0, \ldots, 3\}^5 \times \{0, 1\}^3$. Their success rate is 98% with a database of 20 postures.

Dynamic gesture recognition has been applied to sign language recognition by [LO96, NW96]. Both of them propose hidden Markov models, which have been proven to perform quite well for speech recognition. Often, dynamic gestures are defined as the trajectory of any pointing device, such as a tracker or the mouse. The recognition approach of [SGS97] is based on discretization and three levels of representations for gestures. The success rate is about 90-95%. Fuzzy logic can also be applied to dynamic gesture recognition [Bim99],

Ultimately, one would like to get rid of any kind of data glove. This idea has been pursued by posture recognition research based on computer vision. However, it seems to me that reliable vision-based posture recognition will not be achieved in the near future.

An intermediate solution to untethered posture recognition might be optical tracking of markers on the wrist, palm, and each finger joint. Occlusion, and markers too close to each other are always a problem with optical tracking

— even more so with "optical" posture recognition. For instance, with the fist posture, many markers would be invisible to all cameras, and most others would be very close to each other on the cameras' frames.

Augmented postures. Since the number of intuitive postures is very limited, I implemented an augmented type of posture. Their parameter set comprises joint angles plus one dominant direction reflecting where the thumb of the hand is pointing. This direction can assume one of 6 different values.

With the notion of augmented postures, the posture input space is increased significantly. For example, in order to control the playback of an object path, the hitch-hike augmented posture can be used: pointing the thumb up means "stop", while pointing right or left denotes "forward" and "backward", respectively.

4.2.2 Voice input

Voice input is one of the most natural ways of human communication. Since speech recognition has become quite robust and almost real-time, it is a matter of course to use this input channel for interacting with virtual environments.

I have developed a simple grammar for specifying speech commands (sentences). This grammar is used in specifications of VEs in order to trigger actions. The general syntax is

$$w_{11}|w_{12}|\ldots\ n_1\ w_{21}|w_{22}|\ldots\ n_2\ \ldots$$

This sentence will trigger when the word w_{11}, or w_{12}, etc., is received, followed by at most n_1 "noise words", followed by the word w_{21}, or w_{22}, etc. The concept of noise words allows to discard unimportant utterances of users (e.g., "path [go to] [the] next position").

Some commands include a parameter which must be conveyed to the action. For instance, the command "rotate by n degrees about axis x" contains two parameters. Therefore, instead of specifying a word (or several alternative words), the grammar also allows to specify a (formal) parameter. When the sentence has been matched, the corresponding actual parameters are delivered to the action. The sentence matching engine does not perform any "type checking" (it would require that words do have a type); this is done by the action in a simplistic manner.

Because of parameterized sentences, sentence matching cannot be done by the device server. In my first implementation, sentence matching was done by the device abstraction layer. However, this has turned out to complicate matters (because of parameters). So, sentence matching is done by the interaction manager's class for speech input (see Section 2.4).

It would be possible to parse the user's utterances by the speech recognition engine. Most of them offer a grammar-driven recognition mode anyway. However, like [EWQ99] we have found that parsing the input on the application side offers much more flexibility (in that case, the speech recognition engine runs in "list" mode, i.e., it just gets a dictionary of words it should recognize). First, grammars usually need to be compiled; secondly, my "grammar" is much simpler and can be learned by non-programmers in a matter of minutes; finally, with the grammar based approach, each application would need its own grammar — with my approach, the grammar is basically specified in the description file of the virtual environment.

During the course of my work with voice input in VEs, several guidelines have emerged, which have also been reported by others.

Utilize speaker-*independent* speech recognition, although the recognition rate is slightly less than with a speaker-*dependent* one. As of today (1999) the recognition rate is about 95%. Therefore, there must be always a *fallback* input mode, should the system not recognize some words (for instance, the recognition rate can drop significantly when the speaker is under stress). That fallback will usually be the keyboard or menus.

Develop an easy-to-remember *command language*. This is more natural than keyword spotting, because, for robustness reasons, the speech-recognition is not continuous anyway. Additionally, this helps keeping the interface simple and efficient. Furthermore, speech recognition systems generally get more reliable when provided with a simple unambiguous grammar. The interface should be tolerant for variations of commands (e.g., synonyms), so that users do not have to remember the exact form. This is in accordance with [JFH92].

The number of functions in a VR system will continue to grow. Right now, our virtual assembly simulation application understands over 170 speech commands (not counting synonyms). It is impossible to remember all of them, especially for irregular users. Therefore, it is important to offer other input metaphors by which the user can find the command. Menus are such a metaphor; they have proven to be quite useful. It is important that menus are organized in such a way that the user can derive the corresponding speech command canonically. As explained in Section 4.5.1, they should be 2D menus.

Modes should be avoided, which is true for all user interfaces in general. It is particularly true for the speech input mode, because there is no persistent way to keep the user informed about the current mode. Also, the system should always provide some sort of immediate *feedback* whenever it has recognized a speech command. Sometimes, this is an action performed on the scene so that it can serve as the feedback itself. Sometimes, just a simple acknowledgement should be issued, e.g., a short sound or voice playback.

Data entry can increase user performance considerably [JFH92]. This is in accordance to the wish of users of virtual prototyping applications in the field for the possibility to position objects by a certain numerical translation or rotation in order to investigate alternatives and proposals (see Section 5.2).

Provide a simple switch so that users can turn on and off speech recognition quickly and easily. [OCW94] have pointed out that off-line speech contains 1,200 times more unintelligible words than on-line speech directed to the system.

The findings of [ODK97] indicate that users tend to strongly claim to prefer to interact multimodally, but they actually interact most of the time unimodally, which corresponds to my experience. They also find that the majority (63% in their application) of commands were done using speech only.

4.3 Tracking

By tracking we understand the capability to determine the position and/or orientation of certain points in the real world. This is one of the enabling technologies for VR. It is utilized mostly to track the position and orientation of a user's hands and head, or to track instruments such as an endoscope and scissors. They are also utilized in real-time motion capture systems to track a set of key points and joints of the human body.

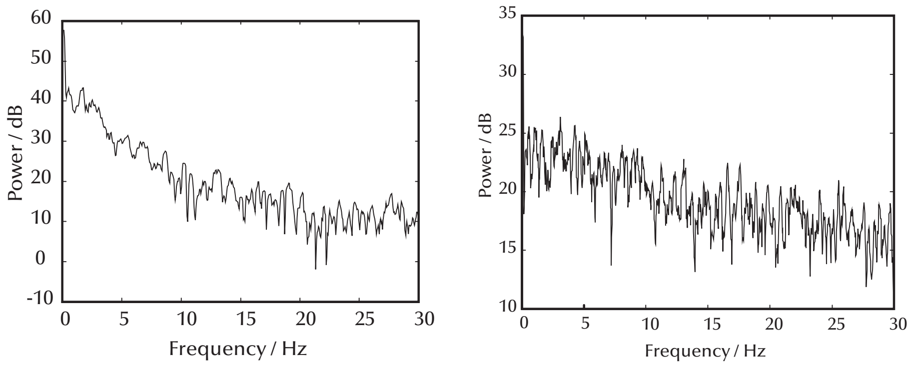


Figure 4.3: Power density spectrum of Polhemus tracker signals of translation (left) and orientation (right). Obviously, this is (almost) white noise, so it cannot be filtered by notch or low-pass filters. (The sensor was about 1.5 m away from the normal-range transmitter.)

Virtual reality always includes humans in the real-time simulation and visualization loop (which would be called a "steering-loop" in scientific visualization [ES88]). This is one of the great benefits of VR. On the other hand, depending on the degree of immersion, it requires *high-fidelity tracking*, in the sense that there is no latency, no noise, and no spatial tracking error (in an ideal system). For styling reviews (in a cave or in front of a large-screen stereo wall) spatial accuracy and noiselessness is most important, for assembly tasks (with glove and HMD) low latency and noiselessness are more important. Augmented reality makes even higher demands on tracking fidelity. [Hol97] has analysed the objective registration error in see-through HMDs; one of the results is that system delay causes more registration error than all other sources combined.

After a brief review of tracking technologies, I will describe some solutions to all three problems related to tracking. They are tailored for electro-magnetic tracking systems, because these are still the prevailing technology with VR setups.

4.3.1 Filtering

Almost all real-world VR environments comprise computers, monitors, loudspeakers, and projectors, etc. All of them radiate electro-magnetic fields which disturb electro-magnetic tracking systems.[6]

Noise in the tracking data of a user's hand can be annoying, and even make difficult assembly tasks impossible, while noise in the position data of a user's head can cause eye strain, or dizziness, and break even the feeling of immersion. Noise in the head sensor's orientation data is most easily perceptible, because a slight rotation causes the rendered image to change much more than a slight translation.

Therefore, the output of electro-magnetic tracking systems must be filtered. As explained above, it is important to filter both translations and orientations. We will see that filtering can solve two of the problems mentioned above at the

[6] Polhemus (and Ascension?) provide hardware support for synchronization with monitors in order to reduce the interference, but this is not always doable, and it does not account for other sources of interference.

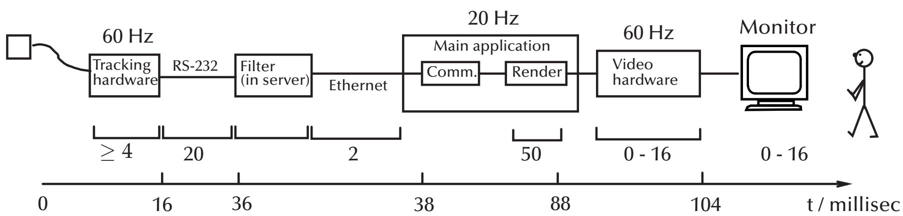


Figure 4.4: There are many stages in the data flow pipeline of any VR system where latency is introduced inherently. The pipeline shown is typical, although shortcuts could be implemented. The delays on the serial line and on the ethernet wire have been measured by myself.

same time, namely noise and latency. In fact, both *must* be solved by the same approach, otherwise one of them deteriorates while the other one is improved.

Traditional filters used in audio signal processing, such as notch filter or low pass filter, will not suffice, because the noise in tracker data is almost white, i.e., all frequencies are present. Figure 4.3 shows the power density spectrum of two signals (Polhemus and Ascension).

An inherent disadvantage of all types of filters is that they introduce additional *latency*. However, latency is already inherent in the data pipeline of any VR system. See Figure 4.4 for an overview of the stages introducing latency in a typical system. This matches well the findings of other researchers [LSG91, WO94]. There are many sources of delay: the tracker itself, serial communication, ethernet communication, image generation, video sync, and video display. Some of the delays can be reduced: for instance, the serial communication can be done at 115 bps; the tracker device can be run in "continuous mode"; when rendering concurrently to the main application, the rendering processes can retrieve the latest tracking data just before they start filling the pipe; when rendering in mono, a higher display rate can be used. However, there will always be some delay, no matter which tracking technology is being used or how fast the rendering can be done.

Since filtering has to compensate for its own latency anyway, it can as well compensate for other lag in the system. On the other hand, even if there was no noise in the tracking data, we would still need to extrapolate them in order to compensate the lag. So, filtering is a by-product of lag compensation, and system lag compensation is a by-product of filtering.

Related work

The problem of noise and latency has been recognized by several researchers.

Internal tracker latency has been examined extensively for Polhemus and Ascension by [AJE96]. They found that the internal lag is about 8 msec, and that the measurement error increases as sensors move faster (much more pronounced with Ascension's Flock of Birds).

Overall system lag has been identified as the number one source of registration errors by [Hol97]. Although I have not done a formal study, it is my experience that static tracker measurement error can introduce even more mismatch between the image displayed by the computer and the situation perceived by the user (through his eyes or kinesthetic senses; see Section 4.3.2).

Kalman filtering has been proposed by [LSG91, FSP92]. It can do optimal linear prediction of some state vector. Usually, the state vector is the current (linear and angular) position, velocity, and acceleration. Like with other filters, the longer the prediction is made, the more noise and overshooting is produced by the filter itself. The overshoot of a Kalman filter with 90 msec look-ahead can be as much as 5 cm [FSP92]. In addition, Kalman filters have quite a few parameters, such as the system noise model and a model for the spectral density. Finding optimal parameters is not straight-forward at all [FSP92], and they are not necessarily the same for all sensors (e.g., head vs. hand sensor, [LSG91]).

The ad-hoc filter based on averaging, differencing, and extrapolation using the differences, exposes similar overshoot problems like the Kalman filter. In addition, under certain cirtcumstances it introduces a lot of noise [FSP92].

For smoothing data, moving-window weighted-average filters are used frequently (most common is $(\frac{1}{4}, \frac{1}{2}, \frac{1}{4})$). Windowing can also be done on the FFT of the time-series; see [PFTV88] for an implementation of tapered windowing.

In signal processing, digital FIR (finite-impulse response) low-pass filters are employed often. They can be constructed fairly easy by doing an FFT of the transfer function and some manual tweaking of the coefficients [CM91]. However, they do not lend themselves to prediction naturally. In addition, the noise level should be at most -60 dB requiring high-order filters, which would introduce too much latency. More powerful are IIR (infinite-impulse response) filters, and they can be used for linear prediction (not to be confused with linear extrapolation). However, they perform only well at predicting smooth and oscillatory signals, and it is not trivial to find coefficients which yield a stable filter.

An approach using grey system theory has been presented by [WO94]. However, the quantitative results are not quite clear.

Requirements

From my experience, a filter for tracker data to be used in VR should possess the following properties:

- It must be predictive. As we have seen above, a VR system can delay the data by as much as 100 milliseconds. So, this is the prediction length a filter should be able to handle.

- It must be very responsive when the position of the tracker suddenly changes. A user can turn his wrist about 90° within about $\frac{1}{4}$ sec without any effort. So, with a tracker update rate of 60 Hz, we get only about 15 samples for the whole wrist turn.

- When the tracker is motionless, then the filter output should be motionless, too. While a user will never be able to hold his head perfectly still (apart from the noise in the signal), it is extremely annoying when the image is jittering even though the user *thinks* it should be still.

Filter pipeline

In order to meet the requirements, I propose a hybrid filter pipeline (see Figure 4.5). The first stage is a *spike filter*. This has proven to be necessary in some environments, where occasional spikes or outliers are present in the data. Then, the data are inserted in a ring buffer. A number of the latest data items are used to feed the monotonicity detector. If it has detected a monotone trend in the most recent data, then the whole buffer is used to feed the *polynomial*

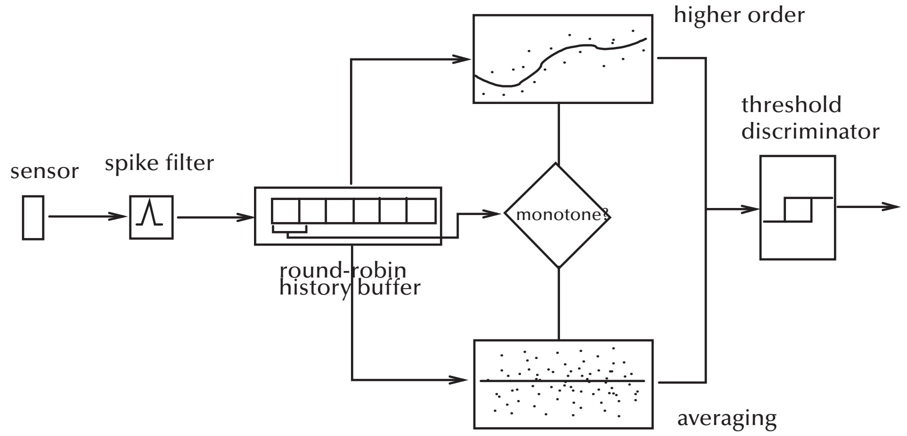


Figure 4.5: I propose a hybrid filter pipeline. It provides predictive filtering in the case of user motion, and optimal filtering in the motionless case.

filter. Otherwise, the *averaging* filter is used. Finally, a *threshold* filter prevents minimal drifts.

The monotonicity detector tries to determine whether or not the tracker is in motion. The idea is that when the tracker is actually still, then prediction is not necessary and we should use the best filter for stationary signals. I assume the noise to be Gaussian; therefore, the average is the best estimate of the true signal.

If the tracker is in (significant) motion, then the polynomial filter steps in. It does polynomial least-squares approximation of the whole data in the buffer. Then, the polynomial is evaluated at some point in the “future” so as to obtain a prediction.

Monotonicity is determined for each coordinate separately. If there is one coordinate with monotone data, then the tracker is assumed to be in motion. Only a small subset is considered for this test, because it must be decided very quickly. If too many samples were considered, then the averaging filter would be used too long before the detector would switch over to polynomial filtering; in other words, the graphical response would lag behind too long.

When the pipeline switches back to averaging, the buffer is flushed and then successively filled with new data. This avoids discontinuities when the user stops his hand, say, from a motion. Flushing is not necessary when switching from averaging to polynomial prediction, because discontinuities are not noticed when the user starts a sudden motion.[7]

If the user performs very slow motions, then the monotonicity test might fail to recognize it. However, this is not a problem, because in that case prediction is not necessary anyway.

The purpose of the threshold filter at the back-end of the filter pipeline is to make the output perfectly still in the motionless case. It compares the newest datum with the one before. As mentioned above, it is annoying when the user “means” to hold his head motionless, still the viewpoint is wobbling. In addition, it prevents spurious position data to be presented to the interaction manager, so there will be no unnecessary computations such as collision detection.

7 Gradual acceleration or deceleration seems to be rare when humans perform every-day tasks.

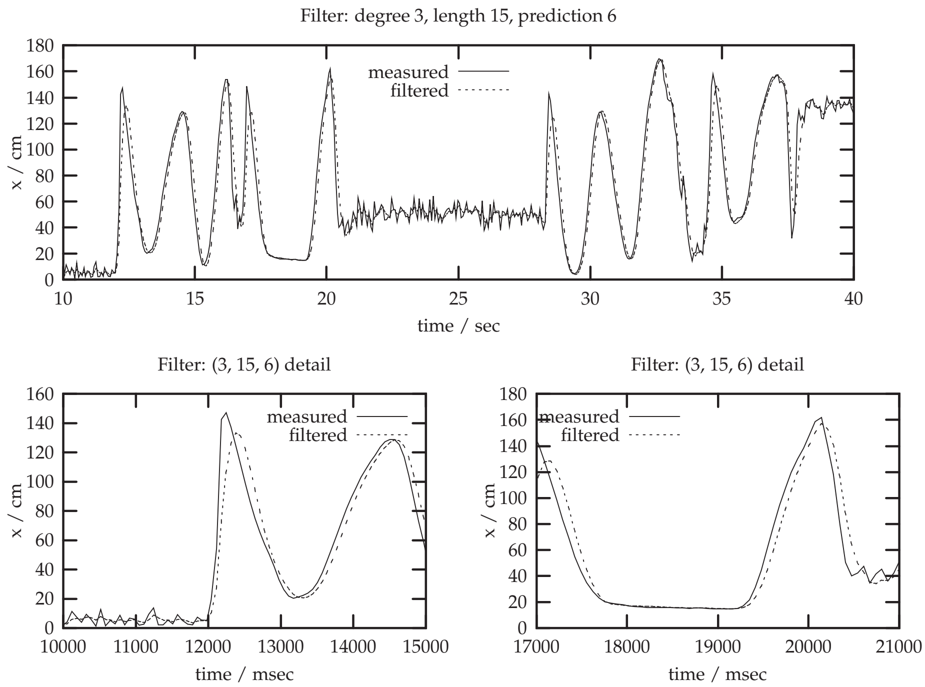


Figure 4.6: The polynomial filter (with degree 3, length 15, and look-ahead 6) applied to real tracker data. The two bottom graphs are details of the top graph. Although the time look-ahead can vary (as opposed to the sample look-ahead), this has not been noticed by any user.

The filter pipeline proposed above has the following parameters:

- Length of monotonicity test; this should be significantly less than filter length.
- Length and degree of polynomial approximation. That length is also the length of the buffer and the length of the average filter.
- Look-ahead for polynomial extrapolation.
- Threshold of the back-end.

Fortunately, good parameters can be found interactively without too much experimentation, because they are fairly independent. Although not necessarily optimal, I have found the following tuples to be good parameters: $(4, 33, 3, 14.5, 2 \times 10^{-5})$, $(4, 33, 2, 15.0, 2 \times 10^{-5})$, $(4, 15, 3, 6.0, 3 \times 10^{-5})$.

The result of the polynomial filter is plotted in Figures 4.6 and 4.7. A minor disadvantage is that the look-ahead *time* can vary a little bit, because the look-ahead evaluation remains constant. However, this has never been noticed by us or customers. The advantage is that there is no overshoot. This is very important, because overshoot is noticed very easily and users are annoyed by overshoot even more than by lag.

In order to find optimal parameters for the polynomial filter, a hill-climbing algorithm can be applied. A synthetic signal (s_i^0) is constructed which resembles real tracker motions (this can be found by “reverse engineering”). Noise is added to this signal synthetically which yields the signal (s_i). The algorithm

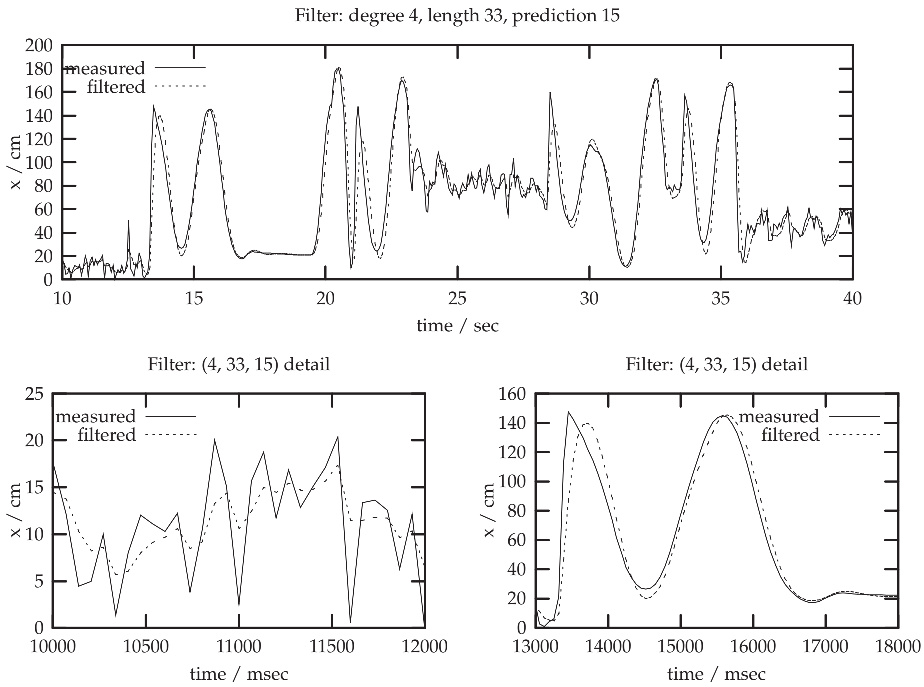


Figure 4.7: The polynomial filter with degree 4, length 33, and look-ahead 15.

starts with several good parameter tuples found empirically. For a given fixed time look-ahead $t \in \mathbb{N}$, it calculates the penalty function

$$E(d, n, l) = \sum_i (f_{(d,n,l)}(s_i) - s^0_{i+t})^2$$

where d =degree, n =length, and l =look-ahead of filter f.

Filtering orientations

This can be done very similarly to filtering positions by using quaternions. The monotonicity test works exactly the same: if the tracker is in rotational motion, so are the quaternions on the unit sphere. Averaging on the unit sphere is done by ordinary Cartesian averaging with subsequent normalization. Analogously, polynomial extrapolation is performed. I am aware of the fact that with this kind of extrapolation the look-ahead cannot be very large. However, this problem has not occurred in my experience.

There is only one pitfall which (sometimes) needs to be addressed: the quaternions $\mathbf{q}$ and $-\mathbf{q}$ represent the same orientation. So, some tracking devices or algorithms restrict their quaternion output to one hemisphere only. In other words, a "path" on the quaternion unit sphere might have a discontinuity (when it would otherwise cross the border of the hemisphere), while the orientations it represents does not.

Instead of quaternions, matrices could be used as well. This might be advantageous if this is the standard representation of orientations throughout the VR system. Matrices can be averaged by successive linear interpolation. Or, they could be averaged component-wise, and then re-orthogonalized.[8] Determin-

[8] From a mathematical point of view, this is not really allowed, but in my experience, it works well.

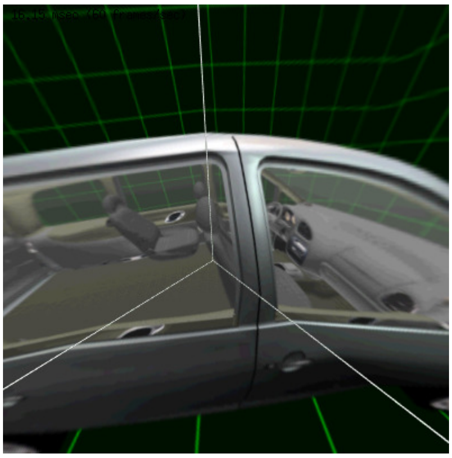


Figure 4.8: Without correction of the magnetic field tracking, an off-center viewer in a cave will see a distorted image. The effect is even worse when the viewer moves, because objects seem to move, too.

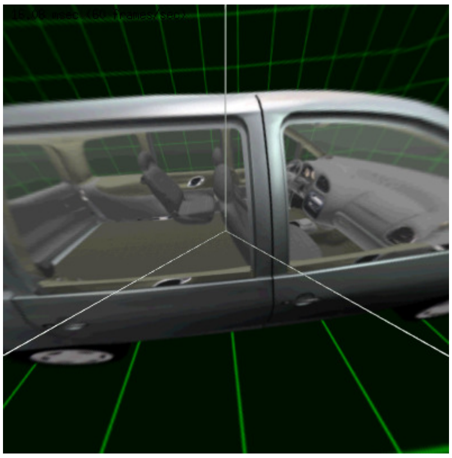


Figure 4.9: With my method, the perspective is always correct, even if the viewer moves. Data courtesy of Volkswagen AG.

ing monotonicity of matrices can be just like the test on quaternions, since each row/column of a matrix must be on the unit sphere. By applying the same sort of "dirty" math, we can do polynomial approximation and extrapolation: we just do it component-wise; the resulting matrix is then re-orthogonalized.

4.3.2 Correction of magnetic tracking errors

Electro-magnetic trackers have become the most wide-spread devices used in today's VR systems ([Zac94a, AFM93]). Commercial optical tracking systems are getting more mature; still, for several reasons electro-magnetic tracking systems will prevail for a few years. The main advantage is that they will be more inexpensive than optical systems for the next few years.

Unfortunately, there is one big disadvantage of electro-magnetic trackers: the electro-magnetic field itself, which gets distorted by many kinds of metal. Usually, it is impossible to banish all metal from the sphere of influence of the transmitter emitting the electro-magnetic field, especially when using a long-range transmitter: monitors contain coils, walls, ceiling, and floors of a building contain metal trellises and struts, chairs and tables have metal frames, etc. While tracking systems using direct current seem to be somewhat less susceptible to distortion by metal than alternating current systems, all ferro-magnetic metal will still influence the field generated by the transmitter.

A distortion of the magnetic field directly results in mismatches between the tracking sensor's true position (and orientation) and the position (orientation) as reported by the tracking system. Depending on the application and the set-up, mismatches between the user's eye position (the *real viewpoint*) and the virtual camera's position (the *virtual viewpoint*) impair more or less the usability of VR. For example, in assembly tasks or serviceability investigations, fine, precise, and true positioning is very important [DFF+96]. In a cave or at a workbench, a discrepancy of 7 cm (3 in) between the real viewpoint and

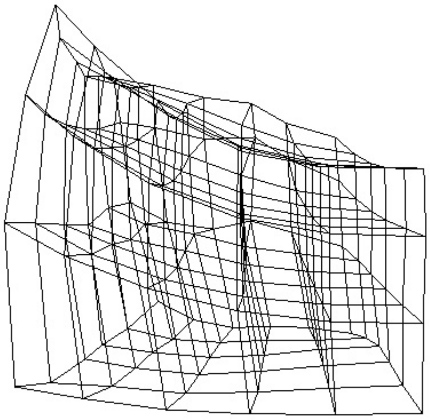


Figure 4.10: Visualization of the distortion of the field: the nodes of the lattice are measured points of a uniform lattice. The discrepancy can be as much as 40–50 cm (15–20 in).

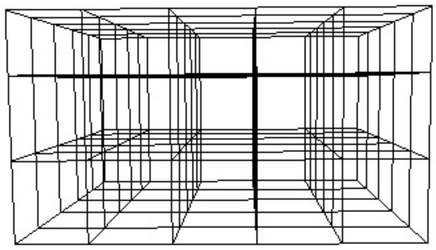


Figure 4.11: The lattice of sampling points. At each of those points, the tracking sensor's value has been averaged and recorded, which produces the field's "snapshot" on the left.

the virtual viewpoint leads to noticeable distortions of the image,[9] which is, of course, not acceptable to stylists and designers. For instance, straight edges of objects spanning 2 walls appear to have an angle (see Figure 4.8), and when the viewer goes closer to a wall, objects "behind" the wall seem to recede or approach (see [HD93] for a description of some other effects). Mismatches are most fatal for Augmented Reality with head-tracking in which virtual objects need to be positioned exactly relative to real objects [OY96].

In order to overcome these adverse effects, I have developed an algorithm which can correct these distortions of the magnetic tracking field [Zac97a]. The algorithm is based on measured data which relate the true position to the position reported by the tracking system at several points within the volume of interest. At run-time of a VR session, the algorithm interpolates these *a priori* measured values with the currently reported position of the sensor.

Since the distortion of the magnetic field is captured by a set of points, a fundamental assumption of my method is that the field does not change over time. Fortunately, in our labs this seems to be true — I could not find any significant changes (see below). Of course, if the set-up is changed, then the magnetic field has to be measured again; the field changes, for instance, when a nearby projector is moved, speakers are installed, or the whole set-up is moved to another place.

My algorithm has several desirable qualities which makes it very suitable for VR systems. First of all, the algorithm is very fast, so it does not introduce any latency into the VR system. Second, the set of measured points can be chosen arbitrarily (it does not need to have a lattice-like arrangement), and more points can be added to it at any time. The remaining error of corrected points does not depend on distance from the transmitter, nor does it depend on the amount of

[9] This is just a rule of thumb, of course. The threshold at which a discrepancy between the real and the virtual viewpoint is noticeable depends on many variables: expertise, distance from the cave wall or projection screen, size of the cave, etc.

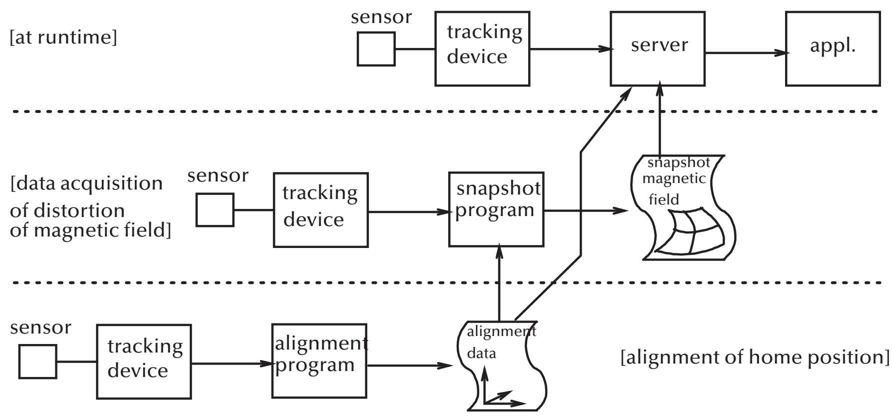


Figure 4.12: The flow of tracking data including alignment and distortion correction.

local "warp" in the magnetic field. Finally, it does not cause any additional latency in the VR system, and it is easy to implement.

In my implementation, the tracking device server has been burdened with the task of correcting tracking errors. So the data flow, as far as the device server is concerned, is: get data from the device (usually via serial line), convert to calibrated coordinate system, filter, correct distortion error, convert to application's coordinate system, send to application (see Figure 4.12).

While this section reports on measurements carried out using commercial systems, the results reported are not to be taken as a characterization of these systems. Except where otherwise noted, the Polhemus Fastrak with a long range transmitter was used to generate all the measurements.

Repeatability

By repeatability I understand the constancy of the magnetic field's distortion. It is a fundamental assumption in all static correction methods, and the accuracy of the interpolation algorithm can be no better than the constancy of the field.

The good news is that the distortion of the field remains sufficiently constant over time (see Figure 4.13). On the same site, the magnetic field has been measured two times as described in Section 4.3.2, the second time 6 weeks after the first time. The distribution of the distance between corresponding lattice nodes shows that the distortion of the field does not change significantly over time.

Another measure of constancy is the traditional correlation function. However, it is not clear what a correlation value of $c \in [0, 1]$ maps to in terms of (maximum or average) absolute error (with units of centimeters or inches).

Comparison of Polhemus and Ascension

Ascension claims that its Flock-of-Birds™ is less susceptible to ferro-magnetic metal than Polhemus' Fastrak™. In my experience, however, the Flock-of-Birds still has large distortions. Some have reported that Ascension is less susceptible when the sensor is mounted directly on a metal device. However, at least on various sites I have found the tracker error due to distortion unacceptably large (see Figures 4.14, 4.15).

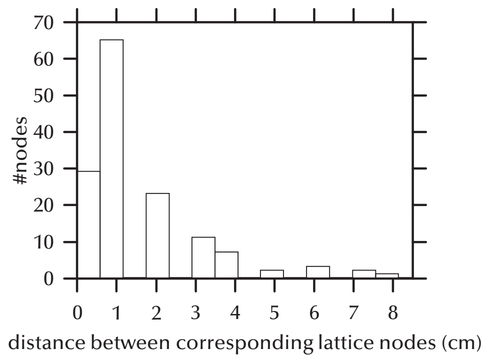


Figure 4.13: The distribution of the distances between corresponding lattice nodes of two snapshots of the magnetic field on the same site. The second snapshot was taken 6 weeks after the first. Most of the distances are within the measuring accuracy of the data. This verifies the assumption of a static distortion of the magnetic field (at least on our site).

The set-up

The first step of the method is to capture the distortion of the magnetic field so that it can be corrected subsequently based on this data. The measuring needs to be done only once per site and set-up. Obviously, it is still quite desirable that it can be done in a minimum amount of time.

For that reason, I have chosen a regular lattice for the set of measuring points, even though my correction algorithm can take an arbitrary point cloud. A lattice was chosen, because it allows to generate the true positions of the measured points automatically, thus minimizing the measuring time. Furthermore, I use four sensors, which further reduces the time. Of course, I take the average of many values (usually 50) when measuring one position sample, because the quality and precision of the data will affect the correction later-on.

In order to take snapshots of real fields, a device is needed for precise positioning of sensors to well-defined points in space. So I devised a wooden apparatus, as depicted in Figure 4.16. Several sensors are strapped to wooden blocks precisely oriented. The apparatus and strappings do not contain any metal. In order to position the sensors at well-defined x- and z-coordinates, we fixed paper to the floor (wide wall-paper, for example) and marked those coordinates with a pen.

With a set-up as described above, we are able to measure the field at 144 points ($= 6 \times 6 \times 4$) in about 20 minutes. Four positions are recorded at a time, which takes about one second for 4×50 samples (for averaging). Measuring time could be reduced further if more sensors would be used simultaneously.

Like [GAS$^+$95], I have tried to use an ultrasonic measuring device to obtain the true positions of the sensors. The advantage is that no "hardware" is needed to guide the positioning of sensors. In addition, the data matrix obtained is the inverse of the one obtained by my method, because the sensor is guided by visual aids (e.g., cubes). So the grid of measured positions is (almost) a regular one (in particular, it is rectangular), while the grid of true positions is warped. However, there are more disadvantages: The ultrasonic device introduces too much measuring error (they report an error of about 1% of the measured distance, my experience is about 2%, i.e., 4 cm if the device is 2 m from a wall). Also, a display device is needed, so the person placing the sensors in

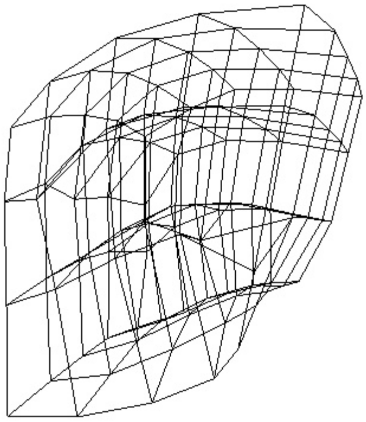


Figure 4.14: In one of our customers' VR lab, the distortion of the Ascension's field is about 50 cm max. The volume measured is about 2x2x1 m^3, with a long-range transmitter positioned 2 m above the floor and 3 m below the ceiling.

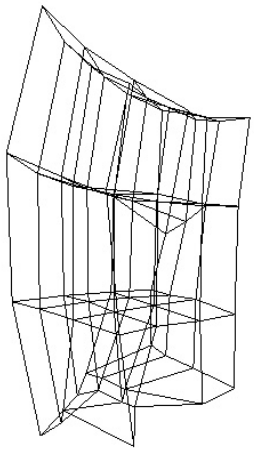


Figure 4.15: Another real-world field of one of our demo sites. The volume measured is about 1x1x2 m^3, the long-range transmitter was about 1 m above the ground.

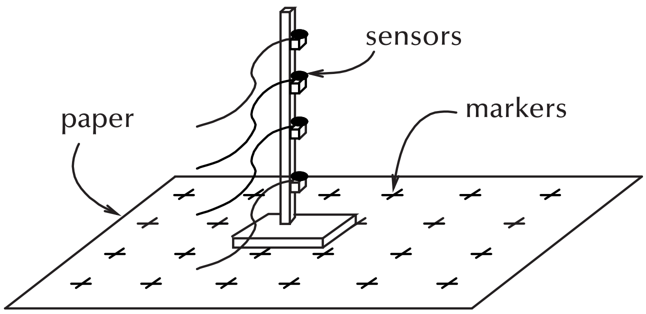


Figure 4.16: A schematic view of the apparatus I have used to take snapshots of electromagnetic fields.

space has to wear an HMD, Boom, or the Cave has to be switched on. Furthermore, it is not clear to me how the procedure suggested in [GAS$^+$95] could be modified such that orientational data can be measured too with the necessary precision. And finally, positioning several sensors precisely at the same time seems impossible to me, because they must be positioned completely independently.

There are many ways of visualizing the measured distortion. While [Bry92, GAS$^+$95] use error vectors, I believe that visualizing the measured lattice provides more insight into the data. One such measured lattice is shown in Figure 4.10 with the sampling points shown in Figure 4.11. The long-range transmitter is located near the upper right corner in the front. One can see clearly that the distortion tends to increase with the distance from the transmitter, although there are also regions closer to the transmitter that also have a large distortion. The influence of one of the projectors (which is very close to the cave due to space limitations) can be seen in the back of the lattice.

4.3.3 Scattered data interpolation

The problem we are faced with can be stated as the well-known interpolation problem in 3-dimensional space: given two sets of points $\mathcal{P} = P^i \subset \mathbb{R}^3$ and $\mathcal{Q} = Q^i \subset \mathbb{R}^3$, we are looking for a function $f : \mathbb{R}^3 \to \mathbb{R}^3$ such that $f(P^i) = Q^i$. Furthermore, we want the function to be sufficiently "smooth": it should be at least $\mathcal{C}^1$-continuous, and it should interpolate "intuitively nice", in particular, there should be no oscillations.

I will call $\mathcal{P}$ the *measured points* in *tracker space*, or, a *snapshot* of the magnetic field; $\mathcal{Q}$ will be called the *true points* in *true space*.

All interpolation schemes can be classified into two categories: global and local. Global methods take all P^i into account, while local methods consider only some P^i within a certain neighborhood. One might also consider the approximation problem. However, I opted for interpolation.

Many of the scattered data interpolation methods are not *affinely invariant*, i.e., $\mathbf{y} = f(\mathbf{x}) \Rightarrow \Phi\mathbf{y} = \Phi f(\mathbf{x})$ does *not* hold. [NF89] has described a method to modify interpolants such that they become affinely invariant. It can be applied to all methods which are based on evaluation of a metric (e.g., the Euclidean distance function).

In general, the following *characteristics* are desirable for scattered data interpolation methods: it should be general, i.e., it should not depend on a certain topology of the data to be interpolated (such as a grid-like topology); it should be at least $\mathcal{C}^1$; it should be affine invariant; and, the effects of free parameters should be understood, so that acceptable defaults can be provided.

Polynomial interpolation, approximation, and look-up tables

To my knowledge, little work has been done on the specific problem of examining and correcting the errors of magnetic tracking devices. All methods presented so far are mostly some sort of simple local interpolation.

[Bry92] has carried out some experiments on tracker error and noise. Three algorithms for correction were presented: polynomial approximation and local interpolation with two different weight functions. Although all three algorithms were evaluated, it is not clear to me whether the local interpolation methods define continuous functions. The error of those correction algorithms is in the range of 2–10 cm. The evaluation seems to indicate that accuracy decreases with increasing distance from the transmitter.

Local interpolation methods involve a look-up of the closest points or the enclosing cell. This needs some additional data structures if the look-up is to be exact or if it is to be fast. One way or another, there is always some computational burden imposed by the look-up itself, which is another reason why I have chosen a global method.

Of course, if the data matrix is acquired such that the grid of measured positions is regular, then the look-up is negligible. Then polynomial local interpolation can be done trivially ([GAS$^+$95, LS97] do trilinear interpolation). However, for reasons discussed above, the measurement method required for that was not an option to me.

The approach of [Kin99] is very similar, except that he uses global polynomial approximation. He chooses degree 3 polynomials. Orientations are represented by Euler angles. I believe this choice might cause problems when a "wrap-around" of angles occurs.

Interpolation using B-Spline-Volumes

B-Spline volumes are defined just like B-Spline surfaces [HL92, Far90], except that the dimension of the domain is the same as that of the image. In our case, since we are looking for an interpolating function $f : \mathbb{R}^3 \rightarrow \mathbb{R}^3$, we define a tensor product *B-spline volume* as

$$X(u, v, w) = \sum_{i=0}^{p} \sum_{j=0}^{q} \sum_{k=0}^{r} \mathbf{d}_{ijk} N_i^l(u) N_j^m(v) N_k^n(w) \tag{4.1}$$

with the *knot volume* (analogously to the knot vector in the 1D case),

$$\begin{aligned}
T &= T_u \times T_v \times T_w \\
T_u &= (u_0 = \ldots = u_{l-1}, u_l, \ldots, u_p, u_{p+1} = \ldots = u_{p+l}) \\
T_v &= (v_0 = \ldots = v_{m-1}, v_m, \ldots, v_q, v_{q+1} = \ldots = v_{q+m}) \\
T_u &= (w_0 = \ldots = w_{n-1}, w_n, \ldots, w_r, w_{r+1} = \ldots = w_{r+n})
\end{aligned}$$

The $N_i^l(t)$ are the *basis functions* defined on the knot vector T_u. For a given knot vector the N_i^l are a basis for the vector space of piecewise polynomial functions of degree $l-1$ on the range $[u_{l-1}, u_{p+1}] \times [v_{m-1}, v_{q+1}] \times [v_{n-1}, v_{r+1}]$.

For an interpolation problem the number of points M must match the number of segments, i.e., $M = (p+1)(q+1)(r+1)$. Since $p \geq l-1$ is required, we need at least $M \geq 64$ sample points for an interpolant of degree $3 \times 3 \times 3$ (if $M = 64$ then we would get a single Bézier surface).

The interpolation (or approximation) problem generally tackled with B-Splines is: given a set of points $Q_i \in \mathbb{R}^3$, find a function $f : [0,1] \times [0,1] \rightarrow \mathbb{R}^3$ such that all $Q_i \in f([0,1] \times [0,1])$. The main issue is to find a good *parameterization* for the Q_i which means: find "good" parameters $P_i = (u_i, v_i)$ such that $f(P_i) = Q_i$. A good parameterization is one which yields a "nice" f, which means that f should not oscillate "unnecessarily". For the 1D interpolation problem (i.e., $f : \mathbb{R}^1 \rightarrow \mathbb{R}^d$) there are fairly straight-forward parameterization techniques such as chord length, centripetal, or the Foley parameterization. For the 2D case (i.e., $f : \mathbb{R}^2 \rightarrow \mathbb{R}^d, d > 2$) parameterization becomes much more "heuristic" and involved; also, for almost any method there are reasonable input data which make the parameterization method fail.

In the case of our interpolation problem $f(P_i) = Q_i, i = 0 \ldots M-1$, we are already given a parameterization by the $P_i =: (\mu_i, \nu_i, \xi_i)$. So the interpolation problem amounts to solving the following linear equations:

$$\begin{bmatrix} n_{000}^0 & n_{00r}^0 & \cdots & n_{pqr}^0 \\ n_{000}^1 & \cdots & & n_{pqr}^1 \\ \vdots & & & \vdots \\ n_{000}^{M-1} & \cdots & & n_{pqr}^{M-1} \end{bmatrix} \begin{bmatrix} \mathbf{d}_{000} \\ \vdots \\ \mathbf{d}_{pqr} \end{bmatrix} = \begin{bmatrix} \mathbf{Q}_0 \\ \vdots \\ \mathbf{Q}_{M-1} \end{bmatrix} \tag{4.2}$$

with $n_{ijk}^s := N_i^l(\mu_s) N_j^m(\nu_s) N_k^n(\xi_s)$. The coefficient matrix of this set of linear equations is quadratic and of size $M \times M$. If M is large compared to l, m, n, then it will be a sparse matrix, because each N does not vanish only on an interval which is small compared to the whole domain T. So, one of the methods specially devised for sparse matrices should be utilized to solve the linear equations [Gv89]. In the 2D case, the coefficient matrix can be re-arranged so that it becomes block diagonal.

Remark: Since the parameters P_i are not arranged in a rectangular grid, we cannot exploit the tensor product nature to compute the deBoor points d_{ijk} more efficiently (ironically, the Q_i *are* on a rectangular grid when we measure a field by the set-up described above). Let us assume for a moment that we are given the following interpolation problem

$$X(\mu_a, \nu_b, \xi_c) = Q_{abc}, \quad a = 0 \ldots A, b = 0 \ldots B, c = 0 \ldots C$$

Then we can rewrite equation 4.1 as

$$X(\mu_a, \nu_b, \xi_c) = \sum_{i=0}^{p} N_i^l(\mu_a) \cdot \underbrace{\sum_{j=0}^{q} \sum_{k=0}^{r} \mathbf{d}_{ijk} N_j^m(\nu_b) N_k^n(\xi_C)}_{\mathbf{d}'_i(j,k)} = Q_{abc}$$

So, in the first iteration we can solve for the $\mathbf{d}'_i(j,k)$ by the linear equations

$$\begin{bmatrix} N_0^l(\mu_0) & \cdots & N_p^l(\mu_0) \\ \vdots & & \vdots \\ N_0^l(\mu_A) & \cdots & N_p^l(\mu_A) \end{bmatrix} \begin{bmatrix} \mathbf{d}'_0(0,0) & \cdots & \mathbf{d}'_0(B,C) \\ \vdots & & \vdots \\ \mathbf{d}'_p(0,0) & \cdots & \mathbf{d}'_p(B,C) \end{bmatrix} = \begin{bmatrix} Q_{000} & \cdots & Q_{0BC} \\ \vdots & & \vdots \\ Q_{A00} & \cdots & Q_{ABC} \end{bmatrix}$$

The coefficient matrix is a band diagonal matrix. Also, by using a back-substitution scheme, the LU-decomposition has to be done only once. Similarly, this scheme can be done recursively until we get the $\mathbf{d}_{ijk}$. Thus the work needed for this kind of interpolation problem is by orders of magnitude less than for the general problem.

Back to our more general interpolation problem. Before we start setting up the linear equations 4.2 we still need to determine a knot volume T such that there will be no gaps in the coefficient matrix. We can achieve this by a simple recursive procedure, which we'll outline here for u:

```
Calculate knot vector for (μi)
assume sequence of (μi) is sorted

split (μi) sequence at median in two halves
calculate knot vectors for the two halves
```

We need to expand the domain $[u_{l-1}, u_{p+1}]$, since B-splines are constant zero outside. Otherwise, the correction would not be able to extrapolate if the user moves a sensor outside the captured region. This happens quite frequently, though, because in general it is not possible to capture the whole volume to be tracked due to time limitations or constraints of the capturing device and the environment.

Shape functions

The problem of interpolation frequently arises in the area of scientific visualization. It is often very convenient to do calculations in computational space rather than in physical space [Frü97, Bat82, Bun07]. Usually, some physical data (like pressure, stress, etc.) are given on a set of sample points in physical space. Often, this is a curvilinear 3D grid (grid cells are called *elements*, grid nodes are called just *nodes*). However, visualizing the data by displaying stream lines, for example, or doing ray casting for volume visualization is

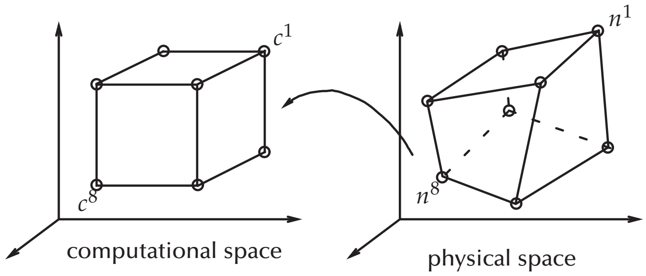


Figure 4.17: Shape functions are used in scientific visualization to transform between physical and computational space.

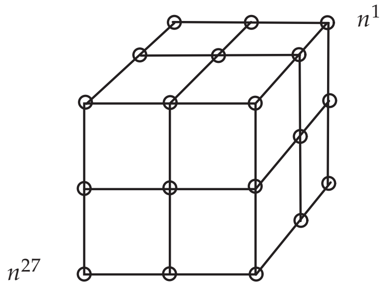


Figure 4.18: Higher order elements can be used to increase accuracy.

computationally much cheaper for regular grids (see Figure 4.17). Again, this basically involves a bijective mapping in 3-space, such that the curvilinear grid is mapped to a regular grid (of course non-scalar tensors given in the nodes need to be mapped, too).

Conversion of points from computational space (a rectangular element) into physical space is trivial (trilinear interpolation). However, in order to convert a point given inside a certain element in physical space into computational space, the notion of *shape functions* has been introduced. Suppose we are given a point $p \in \mathbb{R}^3$ inside the element specified by the nodes $n^1, \ldots, n^8 \in \mathbb{R}^3$ in physical space.[10] The corresponding element in computational space is given by $\mathbf{c}^1, \ldots, \mathbf{c}^8 \in \mathbb{R}^3$ (see Figure 4.17). Then the shape function f for that element is defined as

$$c = f(p) = \sum_{l=1}^{8} f^l(p_x, p_y, p_z) \cdot n^l \tag{4.3}$$

Generally, linear shape functions are considered, therefore

$$f^l(x, y, z) = f_1^l + f_2^l x + f_3^l y + f_4^l z + f_5^l xy + f_6^l xz + f_7^l yz + f_8^l xyz$$

So, a linear shape function f for a 3D element is defined by 8×8 coefficients. It changes linearly when moving on an edge of the hexahedron.

[10] I will not consider hexahedron elements with more than 8 nodes (edge centered and/or face centered nodes), since they are not relevant for scattered data interpolation.

If $f^l(\mathbf{n}^k) = \delta_{lk}$ (δ being the Kronecker symbol), then f does interpolate the $\mathbf{c}^k$. So, the coefficients can be obtained by solving 8 sets of linear equations $Af^l = e^l, l = 1\ldots8, e^l = (\delta l1, \ldots, \delta_{l8})$, or in expanded form

$$\begin{bmatrix} 1 & n_x^1 & n_y^1 & n_z^1 & n_x^1 n_y^1 & n_x^1 n_z^1 & n_y^1 n_y^1 & n_x^1 n_y^1 n_z^1 \\ \vdots & & & \vdots & & & & \\ 1 & n_x^8 & n_y^8 & n_z^8 & n_x^8 n_y^8 & n_x^8 n_z^8 & n_y^8 n_y^8 & n_x^8 n_y^8 n_z^8 \end{bmatrix} \begin{bmatrix} f_1^l \\ \vdots \\ f_8^l \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ 0 \\ \vdots \end{bmatrix}$$

In order to increase accuracy of the interpolant, higher-order shape functions can be considered. However, memory usage increases exponentially with the order. In the quadratic case (see Figure 4.18), elements can be specified by up to 27 nodes and $f^l(x, y, z) = \sum_{ijk=000}^{222} f_{ijk}^l x^i y^j z^k, l = 0\ldots26$. (The 20-node hexahedron element is quite common.) Thus, a quadratic shape function for one 3D element is defined by up to 27^2 coefficients.

The faces of hexahedral elements in physical space are not necessarily planar. In order to determine whether a point is inside an element physical space, it can be transformed into the unit element in computational space (after checking the bounding box). If it is inside the unit element, then it must also be inside the element in physical space.

Hexahedron elements can be used only if the data are arranged in a grid topology. One way to overcome this problem is to use *tetrahedron* elements. The grid composed of tetrahedra is called an "unstructured grid" in FEM parlance.

Let $n^1, \ldots, n^4$ denote the nodes of a tetrahedron element. The shape function for the unit tetrahedron $T^0 = (x, y, z)|x \geq 0, y \geq 0.z \geq 0.x + y + z \leq 1$ is trivial:

$$\begin{aligned} f^1(p) &= p_x \\ f^2(p) &= p_y \\ f^3(p) &= p_z \\ f^4(p) &= 1 - (p_x + p_y + p_z) \end{aligned} \tag{4.4}$$

Shape functions for an arbitrary tetrahedron T can be obtained by concatenation of 4.4 with the affine transformation mapping that tetrahedron onto T.

In order to do scattered data interpolation via shape functions, the "unstructured grid" approach seems to be most promising. Given a set of data points, a Delaunay triangulation has to be computed. This yields a triangulation such that each tetrahedron's circumsphere does not contain any other point. Also, it is that triangulation which maximizes the minimum angle of all tetrahedra.

For large data sets ($> 10,000$ points), shape functions are better suited because of better numerical stability. However, they have a few drawbacks: most of them provide only C^1-continuity, additional data structures storing topological information is needed, and there is always a little overhead in order to look up the appropriate shape function.

Here, consecutive interpolation points are *temporally coherent*, i.e., they will usually be close to each other. In order to speed up the *point location problem*, we can exploit this by saving the enclosing tetrahedron. The enclosing tetrahedron for the next point is probably close to the old one (if not the same). In general, it can be found in a few steps by a simple hill climbing procedure (with

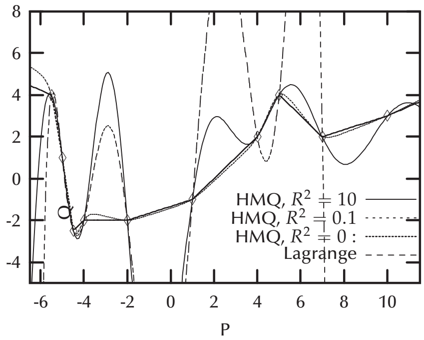


Figure 4.19: 1D examples of HMQ interpolation functions through 10 points with various R^2 parameters. For $R^2 = 0$ the function is piecewise linear. The Lagrange interpolant exposes fatal oscillations.

each step the algorithm goes to the neighbor tetrahedron which is closer to the point).

4.3.4 Hardy's Multiquadric

Hardy's Multiquadric method (HMQ) [HL93, FN94] can be used to construct an interpolation function $f : \mathbb{R}^n \to \mathbb{R}^m$, with arbitrary m, n. The general form of the interpolation function for $m = n = 3$ is

$$f(P) = \sum A_i \omega_i(P) \quad , \qquad P, A_i \in \mathbb{R}^3$$

with

$$\omega_i(P) = \sqrt{(P - P_i)^2 + R^2} \quad , \quad R > 0$$

Requiring $f(P_i) = Q_i$ leads to three sets of linear algebraic equations with a symmetric matrix. All three matrices have size $N \times N$, $N =$ number of measured points.

We cannot use the Cholesky decomposition, because the matrix is not positive definite, since the upper-left 2×2 sub-determinant is negative:

$$\begin{vmatrix} |R| & \beta_{12} \\ \beta_{21} & |R| \end{vmatrix} = R^2 - \beta_{12} < 0$$

with $\beta ij = \sqrt{(P_i - P_j)^2 + R^2} > R^2$, provided $P_1 \neq P_2$. LU decomposition [PFTV88] is, therefore, a natural choice for solving these equations, which is what I do.

The HMQ method does not tend to oscillate as polynomial interpolation schemes do (e.g., Newton or Lagrange interpolation), since the degree of the interpolating function does not depend on the number of sample points. Instead, the smoothness of the HMQ interpolation function depends on the parameter R^2. It can be shown that $f \in \mathcal{C}^\infty$ for $R^2 > 0$ [HL93]. Figure 4.19 compares the HMQ method to Lagrange interpolation (in 1D), and shows the effect of various R^2's. Figure 4.20 compares the HMQ method to plynomial approximation (see above).

There are other scattered data interpolation functions such as *Shepard* interpolation, or natural Hermite spline interpolation [Fra82]. However, all of these

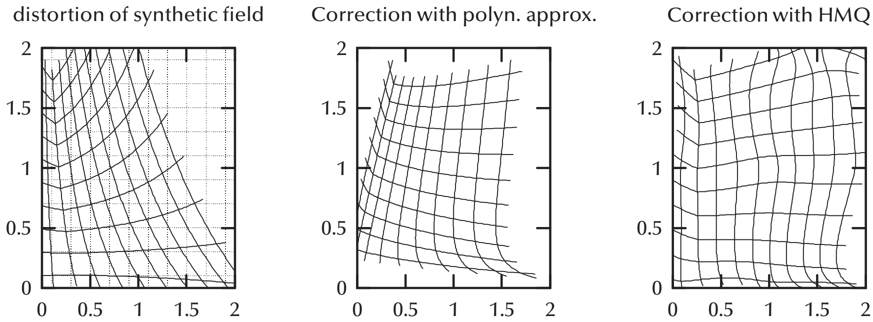


Figure 4.20: Comparison of HMQ interpolation with polynomial approximation. The graph on the left shows a synthetic distortion function, while the graph in the middle and on the right show the correction using polynomial approcimation and HMQ, resp. The interpolation functions were computed with the grid points shown in the graph pon the right, while the interpolation in the middle and on the right were done on the lines right between those grid points.

do not seem to be better than HMQ (sometimes much worse) while being much more involved [NT94].

I have also tried a close variant of HMQ, the reciprocal multiquadric (RMQ), which is defined by basis functions

$$\omega_i(P) = \frac{1}{\sqrt{(P - P_i)^2 + R^2}} \quad , \quad R > 0$$

Although I expected the RMQ to produce fairer interpolations, this is not true. Also, selection of the optimal R^2 seems to be more critical than with HMQ.

The more general form of multiquadrics is defined on basis functions

$$\omega_i(P) = ((P - P_i)^2 + R_i^2)^{\mu_i}$$

However, as [Fra82, Dyn87] point out, results are best when $\mu_i = \mu$ and $R_i = R$. I have experimented with $\mu_i = \frac{1}{4}, \frac{1}{2}, 2$, and $\| \cdots \|$. I found, that interpolation is best for $\mu_i = \frac{1}{2}$. For some of the other exponents a good R^2 is very critical and the coefficient matrix might even be near-singular.

A solution to the multiquadric equations exists and is unique, with or without polynomial precision.

The HMQ method by itself is not affine invariant, i.e., the interpolant is not invariant under affine transformations of the input data. More precisely, the HMQ is not invariant under uniform scalings (it is invariant under rotations and translations, though). Since my implementation does not include the method of an invariant metric as described above, I investigated its performance always in our problem domain (i.e., input data are given in units of cm and in the range ± 300 approximately).

Polynomial precision refers to a method's ability to reproduce a polynomial of a given order. By augmenting the MQ method by a polynomial term it can achieve polynomial precision, too: While this might be important in other fields, such as computer-aided graphic design, polynomial precision has no significance in our application, since we do not know anything about the input data. [CF91, FT96] found that, unless the surface may be closely approximated

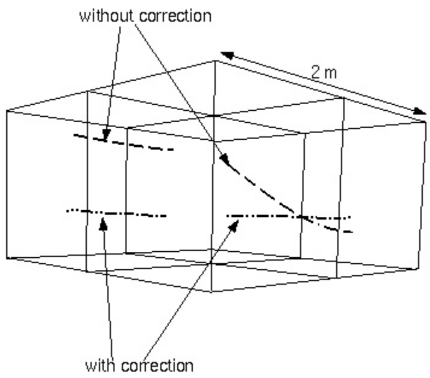


Figure 4.21: The dashed lines show the locus of points as reported by the tracker when the sensor is moved along a perfectly straight line. The dash-dotted lines show the same lines with correction. The error of the corrected positions from the true positions is less than 4 cm.

by a low-order polynomial, the MQ basis should not be augmented (sometimes not even by a constant), because it may lessen the quality of the MQ fit.

It is well-known that the solutions to radial basis functions can be problematic for large numbers of points. The condition number of the coefficient matrix rises with N. This can be remedied to some extent by scaling the data to the unit square. The condition number is usually better for the MQ method than for the thin plate spline method. The scaling strategy is known to allow for the solution of problems involving about 1000–2000 points.

The multiquadric method proved to be one of the best methods for interpolating over a set of different “known" surfaces from (not too) scattered observations [Fra82].

The optimal parameter R^2

Admittedly, the HMQ method does have one “magic” parameter (R^2) in the basis functions $\omega_i(P)$. It has considerable influence on the “smoothness” of the interpolation function (see Figure 4.19). The bad news is, no simple and robust formula is known to determine an optimal R^2 [HL93]. It depends on the number of points, the diameter of their circumcircle, and the values $f(P^i)$. [Fra82] have used $R = 1.25\frac{D}{\sqrt{N}}$, with $D =$ the diameter of the circumcircle of all measured points and $N =$ the number of points. [CF91] proposed an algorithm which produces near-optimal values for R^2.

Therefore, I developed a program to investigate the effects of R^2 interactively. Given a snapshot of the field, a few lines of tracker data, and their true positions, we can determine an optimal R^2 within a few minutes. The good news is that with 100–200 measured positions, $R^2 = 10 \ldots 1000$ is optimal, and within that range the exact value has very little impact on the quality of the interpolation.

Accuracy

I have tested my method with very satisfactory results. With a data set of 144 measured points within a volume of (2.4 m)^3 (see Figure 4.10), the corrected points are usually within 4 cm ($\approx$ 1.5 in) of the true points, while some of the measured points (without correction) are displaced by over 40 cm ($\approx$ 15 in; see Figure 4.21).

With my algorithm, the error of the corrected points from their true positions does not depend on the distance to the transmitter, nor does it depend on the amount of local "warp" of the magnetic field.

One of the evaluations of the accuracy of the HMQ method was done by moving a sensor along several well-defined straight lines within the measured volume. Then, for all points on the same line, two out of three coordinates of the corrected points should remain constant and have a known (true) value. Accuracy here is the maximum deviation of these coordinates from the true ones. The proposed algorithm and field measuring method reduce the error of a tracking sensor's position to 2–5 cm, which remedies any distortions in the images projected on our cave walls (see Figure 4.9).

There are a few factors affecting the quality of the corrected position: accuracy of the measured field data on which the interpolation is based, the sampling density, constancy of the magnetic field, and the parameter R^2 (see above).

I feel that it is very difficult to achieve an accuracy of better than 1 cm when acquiring the sample data of the magnetic field, especially when measuring a large volume such as a cave. In order to improve the accuracy of the data, a much more precise way of positioning the sensors within the volume would be needed.

Mathematical experiments

In order to find out the theoretical limits of the HMD method, I devised 6 "distortion" functions which warp 3-space. These functions are evaluated at grids, and the images are regarded as snapshots of a distorted "field". With $\mathbf{p} = (x, y, z)$, the functions $f_i : \mathbb{R}^3 \to \mathbb{R}^3$ are:

$$f_0(\mathbf{p}) = \mathbf{p}$$

$$f_1(\mathbf{p}) = 0.9 \begin{pmatrix} 0.4x + \frac{x}{d(\mathbf{p})} + \frac{y}{6} \\ 0.4y + \frac{y}{d(\mathbf{p})} \\ 0.4z + \frac{z}{d(\mathbf{p})} \end{pmatrix} \quad \text{with} \quad d(\mathbf{p}) = 0.4 \left(\frac{\mathbf{p} + (30, 30, 30)}{\mathbf{s}} \right)^2 + 1$$

$$f_2(\mathbf{p}) = 0.6 \left(\mathbf{1} + \mathbf{s} \cdot \left(e^{\frac{\mathbf{p}-\mathbf{1}}{\mathbf{s}}} - 1 \right) \right)$$

$$f_3(\mathbf{p}) = (0.7, 1.3, 1) \left(\mathbf{1} + \mathbf{s} \cdot \sin \left(\frac{\pi}{2} \left(\frac{\mathbf{p} - \mathbf{1}}{\mathbf{s}} + 1 \right) \right) \right)$$

$$f_4(\mathbf{p}) = f_1(\ f_2(\mathbf{p})\ )$$

$$f_5(\mathbf{p}) = f_1(\ f_3(\mathbf{p})\ )$$

$$f_6(\mathbf{p}) = \begin{pmatrix} 0.9\, x \\ 1.2\ (\ (y - l_y) + 200\ )\ \sin(\alpha(y, z)) \\ 0.7\ (\ h_z - (y - l_y) + 200\ )\ \cos(\alpha(y, z)) + \frac{x}{10} + \frac{s_z}{2} \end{pmatrix}$$

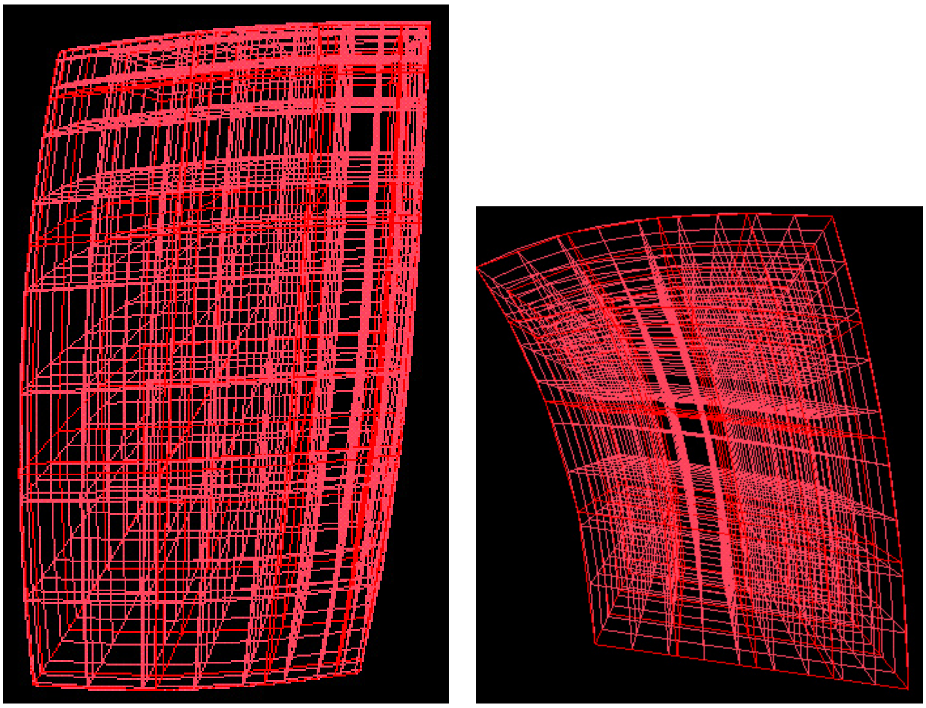


Figure 4.22: Two of the mathematical "distortion" functions used in the mathematical experiments to assess the theoretical accuracy of the HMQ method.

with

$$\alpha(y,z) = \arccos\left(\frac{y - l_y + 200}{\sqrt{(y - l_y + 200)^2 + (z - l_z)^2}}\right) - 0.2$$

where vector multiplication, division, exponentiation, and sin/cos of vectors are all meant component-wise. $\mathbf{l} = l_{x,y,z}$ is the lower corner of the domain of the synthetic field (i.e., the working volume measured), $\mathbf{h} = h_{x,y,z}$ is the upper corner of that domain, and $\mathbf{s} = \mathbf{h} - \mathbf{l} = s_{x,y,z}$ is the extent of that domain. Many of the functions above (and below) contain some "magic constants". These are just there for scaling the output so that the distortion is not too small or too large when applied to the set $[-120, 120]^3$, which is a typical range of our working volumes (in particular, of our cave).

Functions 1–4 produce pretty "nasty" distortions while functions 5 and 6 look more like real-world distortions (see Figure 4.22).

The overall distortion of the functions can be seen in Figure 4.23. Figure 4.25 shows the remaining error after HMQ interpolation with an optimal R^2 for translations and different "grid" sizes. Of course, in order to determine the residual error the functions are evaluated at points "between" the lattice points used for "training" the HMQ.

Optimal R^2 values have been determined by an exhaustive search for each function and each grid size (see Figure 4.26). It is not clear to me why for most of the functions there seems to be no optimal R^2. Furthermore, I do not have a satisfactory explanation as to why the "well-behaved" functions 5 and 6 do not show a reasonable optimum.

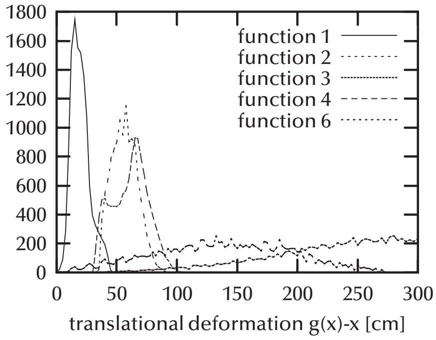


Figure 4.23: Histogram of distortions produced by the functions $f_1, \ldots, f_6$.

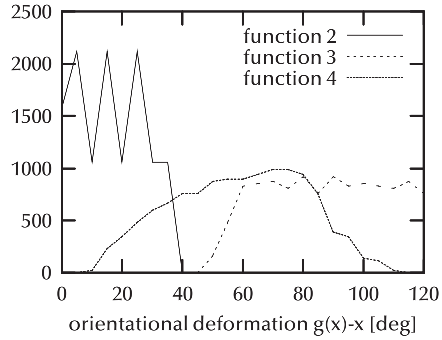


Figure 4.24: Histogram of distortions produced by the functions $o_2, \ldots, o_4$.

For orientations, I have carried out similar experiments. The distortion functions were

$$o_1(x,y,z) = \big((1,0,0),\ (0,1,0)\big)$$

$$o_2(x,y,z) = \begin{pmatrix} cos(\alpha(y)) & -sin(\alpha(y)) & 0 \\ sin(\alpha(y)) & cos(\alpha(y)) & 0 \end{pmatrix} \qquad \text{with} \qquad \alpha(y) = 0.8\frac{\pi}{2}\frac{y}{s_y}$$

$$o_3(x,y,z) = \begin{pmatrix} s(y)O_{00} - c(y)O_{02} & O_{01} & c(y)O_{00} + s(y)O_{02} \\ s(y)O_{10} - c(y)O_{12} & O_{11} & c(y)O_{10} + s(y)O_{12} \end{pmatrix}$$

$$\text{where} \quad O = o_2(x,y,z),\ s(y) = \sin(\alpha(y)),\ c(y) = \cos(\alpha(y))$$

$$o_4(x,y,z) = \begin{pmatrix} \sin(\alpha(z)) & \cos(\beta(y))\cos(\alpha(z)) & -\sin(\beta(y))\cos(\alpha(z)) \\ 0 & \sin(\beta(y)) & \cos(\beta(y)) \end{pmatrix}$$

with

$$\alpha(z) = \frac{\pi}{2}\log\left(1 + 1.4\frac{z - l_z}{s_z}\right) \qquad \text{and} \qquad \beta(y) = \frac{\pi}{2}\log\left(1 + 1.4\frac{y - l_y}{s_y}\right)$$

No translational distortion was done so as to isolate the quality of the correction for orientations. A histogram of their distortion can be seen in Figure 4.24. The quality of correction is plotted as histograms in Figure 4.27.

It seems that orientational errors can be corrected much better than translational ones. It is not quite clear to me why that is. One reason might be that for orientations the interpolation function lives in a higher dimensional space ($\mathbb{R}^6$) than for translations ($\mathbb{R}^3$).

Timing

Computation of $f(P)$ is linear in the number of measured samples P_i. Interpolating one 3-dimensional position by $f(P)$ with $i = 144$ (i.e., 144 samples) takes 0.5 milliseconds on a 250 MHz R4400 processor. Since the correction of the field's distortion takes so little time, no additional latency is introduced into the VR system. In fact, a transmission of one complete data record from the tracking system to the host takes much longer.

Solving the three sets of linear equations in order to compute the coefficients A_i is on the order of seconds; for 144 sample points, it takes about 2 sec (250 MHz R4400).

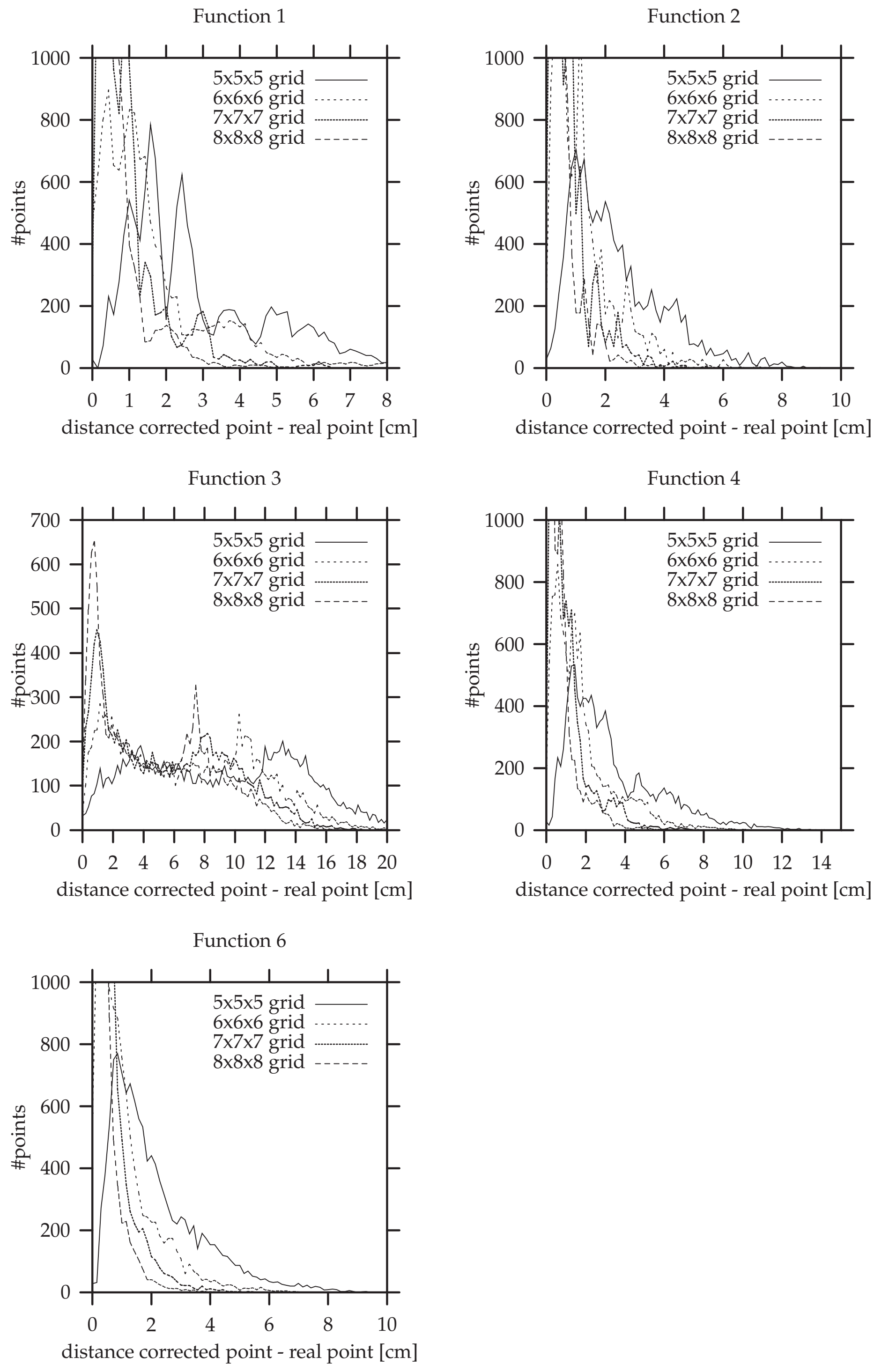


Figure 4.25: Histograms of the translational error after HMQ interpolation with optimal R^2 for different grid resolutions ($5 \times 5 \times 5 \ldots 8 \times 8 \times 8$.

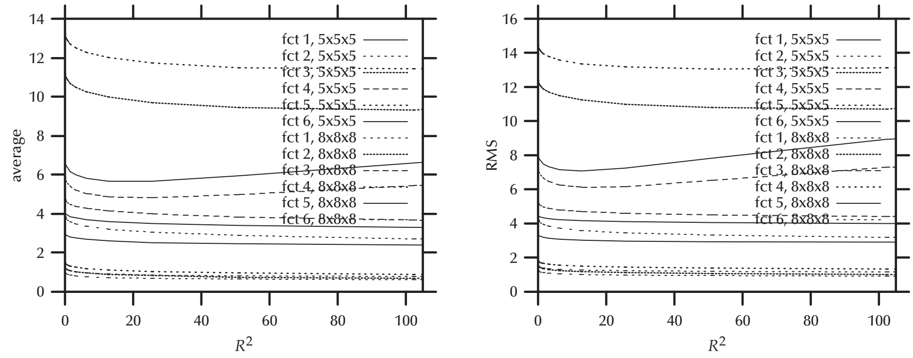


Figure 4.26: The dependence of the remaining distortion error on the parameter R^2 (left: average error; right: RMS).

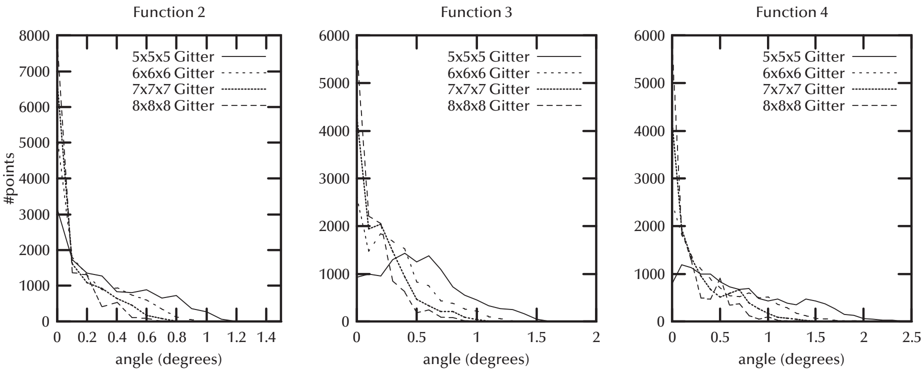


Figure 4.27: The rotational error after HMQ interpolation with optimal R^2.

"Boxed" HMQ

In some cases there can be a considerable "shift" (or translational bias) in the snapshot compared to the true positions (see Figure 4.28). Since the HMQ interpolation is invariant under uniform scalings, I presumed that it might be worthwhile to scale/translate the snapshot to better fit the true grid positions before calculating the HMQ interpolation coefficients.[11]

The idea was to compute an outer bounding box and an "inner" bounding box of the snapshot (see Figure 4.29). Then the scaling/translation is determined such the bounding box of the true grid is contained evenly between the transformed outer and inner bounding boxes of the snapshot.

It seems that, unfortunately, this transformation has not significantly improved the residual error of the correction — with an optimal R^2-value the error of the corrected positions of the synthetic "fields" (see above) is reduced by only 2–3%. With non-optimal R^2's the improvement is a bit larger, of course, but I feel this not relevant.

Correcting orientations

In a cave or at a workbench, a skewed orientation of the viewpoint is not as fatal as an offset, since the orientation determines only the parallax between the

[11] Thus, in order to correct a measured position at run-time, a similar scaling/translation has to be done, then the interpolation, then the inverse scaling/translation.

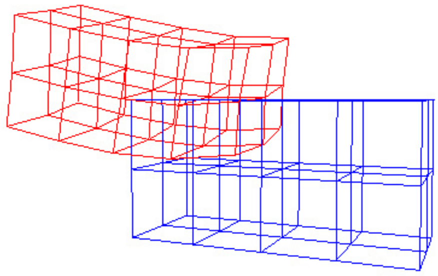


Figure 4.28: In some cases the snapshot can contain a significant amount of translational bias.

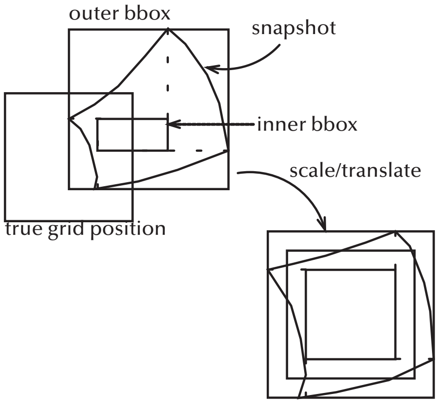


Figure 4.29: Scaling/translating the snapshot to better fit the true grid positions has not improved the residual error of the correction.

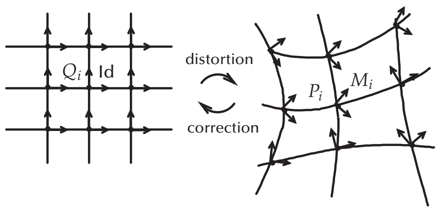


Figure 4.30: Orientations can be viewed like a smooth space tangential to positional 3-space.

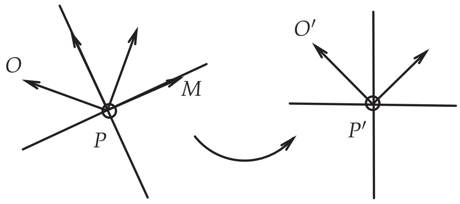


Figure 4.31: Orientations can be corrected similar to positions.

two images for the left and the right eye, respectively. A skew will, of course, introduce *vertical* parallax, which can break the stereoscopic effect altogether. But in our experience, the human eye seems to be highly tolerant of vertical parallax.

When tracking the hand of a user who is trying to perform a virtual assembly task, however, a mismatch between the true and the virtual hand's orientation might lead to confusion and frustration. Especially in a cave, a distorted orientation of the virtual hand becomes very obvious. The most demanding application with respect to tracking is AR. In my experience, a tracking error of 1 deg can be annoying.

Orientations can be corrected similar to positions. There is a difference in that they are like a smooth space tangential to positional 3-space [Lau60, Str64], i.e., each point carries its own coordinate system (see Figure 4.30). Therefore, orientations cannot be interpolated directly, that is, we cannot construct a correction function $f : \mathbb{R}^6 \to \mathbb{R}^6$ or $f : \mathbb{R}^4 \to \mathbb{R}^4$ (depending on how orientations are represented).

One of the assumptions made here is that the orientational space associated with each location is *orthogonal*. If this were not true, then a rotation of the sensor about n degrees (with constant location) would produce a tracker output of a rotation $\neq n$ degrees. In other words, the orientational distortion would depend on both position *and* orientation. In that case, we would have to mea-

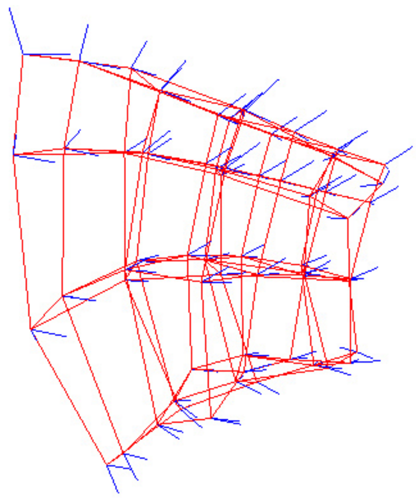


Figure 4.32: Orientations are distorted like positions, but they are not tangential to the distorted field of positions. In addition, they tend to be slightly less distorted.

sure several orientations at the same location, and the interpolation function would be $f : \mathbb{R}^{3+6} \to \mathbb{R}^6$ (assuming 2-vector representation of orientations). Fortunately, the orientational space *is* orthogonal, which I have verified by experimentation. This is in contrast to the findings of [LS97], but maybe that is due to different tracking systems.

With orthogonal orientational space, the measurement procedure is pretty much the same as for correction of positions only. The only difference is that the sensor(s) must be always positioned in null orientation. This is the orientation in which calibration has been done, i.e., it is the orientation where the tracker *should* return always the identity matrix. During measurement, the values returned by the tracker for position P^i *and* orientation M^i will be averaged and recorded in the snapshot. The true orientation does not have to be recorded, because it is known to be the identity matrix.

It turns out that the distorted null orientations are not tangential to the distorted positions (see Figure 4.32). Also, it seems that orientations get not as much distorted as positions and that there is no direct correlation between the amount of orientational distortion and positional distortion.

Assume we are given a measured point P and an orientation O together with that point. In order to correct orientations, it suffices to construct an interpolation $M = g(P)$. The corrected orientation is $O' = M^{-1}O$ (see Figure 4.31). I have used the matrix representation for orientations; I will explain the reasons below. More precisely, I used the 2-vector representation, so the function $g : \mathbb{R}^3 \to \mathbb{R}^6$, i.e.,

$$g(x, y, z) = \begin{pmatrix} m_x \\ m_y \end{pmatrix}$$

and $m_z = m_x \times m_y$. Then,

$$O' = \begin{pmatrix} \vdots & \vdots & \vdots \\ m_x & m_y & m_z \\ \vdots & \vdots & \vdots \end{pmatrix} O$$

We still need to construct g. The interpolation problem is a little bit different from the one in Section 4.3.3. Here we need to find g such that

$$\forall i : \; g(P^i) = M^i$$

and

$$\forall P \in \mathbb{R}^3 : \; g(P) \in \mathrm{SO}_3$$

All interpolation methods described above do not necessarily meet that constraint, whether we choose matrices or quaternions to represent orientations. So I chose to drop that constraint and apply the standard HMQ interpolation. Afterwards, the resulting matrix is orthogonalized and normalized. The results have been satisfactory.

Correction of dynamic fields

My approach assumes a static distortion of the magnetic field. However, in set-ups comprising a Boom plus magnetic tracking (e.g., Boom plus glove), the magnetic field is changed by the Boom. Unfortunately, the “shape” of the distortion depends significantly on the position and orientation of the Boom (I learned that the hard way). I am not sure whether or not this kind of dynamic distortion is “repeatable” in the following sense: given a certain position and orientation of the Boom, the measured position of a tracking sensor remains constant over time. If the assumption is true, then this kind of dynamic distortion can be corrected by an interpolation function $f : \mathbb{R}^{3+3+6} \rightarrow \mathbb{R}^3$, i.e., $P' = f(P_{\text{tracker}}, P_{\text{boom}}, M_{\text{boom}})$. If the “shape” of the distortion depends not only on the Boom’s position but also on its orientation (which it probably does), the process of measuring the field is probably very time-consuming.

Automatic field measurement

Because the HMQ method does not rely on a certain topology of the field snapshot, it is well-suited for automatic field measurement as suggested by [Kin99]. The idea is to set up a more precise tracking system (such as an expensive optical tracking system) temporarily. Then, during regular VR sessions (possibly with uncorrected tracking), the measurements from the precise tracking system are recorded as true positions/orientations, together with the actual measurements from the imprecise system. When enough samples have been recorded, the precise tracking system can be removed again.

Using a precise tracking system, one can, of course, also do manual field measurement. The advantage then is that this can be done in a matter of minutes. The tracker just has to be moved about the volume of interest — this could even be done in a chaotic manner.

Of course, before performing the HMQ interpolation, the samples should be “weeded” out, so that the distribution becomes as uniform as possible.

4.4 Navigation

Navigation is probably *the* most fundamental interaction technique in VEs, which can be found in all VR systems and applications. It is also one which can often not be modeled after the real world for practical reasons (how does one navigate through a virtual city efficiently?). Because of that, a wide range of navigation metaphors has been investigated [GMPP95, CW92, MCR90, WO90, RH92, Han97].

By navigation I understand any method of controlling the viewpoint in a virtual environment. This is not to be confused with the cognitive process of *wayfinding*, by which a user determines his own position and the direction of the target position based on his mental map of his (virtual) environment. Obviously, navigation by itself should burden the user with as little cognitive load as possible.

A taxonomy of navigation has been presented by [BKH98]. It is three-dimensional:

1. Target:
   - selection of direction: this includes pointing in some direction by a (tracked) body part, e.g., the hand, the torso, the head (gaze direction);
   - discrete selection of target: such as selecting a (named) position from a menu, selecting a target object (the system will place the user nearby), or by typing or speaking the coordinates;
   - selection on a map: maps can be conventional 2D maps, 3D maps like WIM, inset views from different viewpoints, etc.
2. Velocity (or acceleration):
   - constant;
   - automatic by the system (e.g., adaptive to distance from selected target);
   - selected by the user, for instance through a menu, speech commands ("faster"), or a joint's flex value.
3. Trigger (i.e., what input conditions start/stop navigation):
   - no trigger needed, because navigation is always on;
   - start and/or stop, e.g., gesture, speech command ("forward", "left", etc.), joystick button, etc.;
   - automatic by the system.

Most techniques can be mapped on the *flying carpet* metaphor. The idea is that the user is riding a cart while he can look around (see Figure 4.33). This corresponds to a subtree of the scene graph as shown in Figure 4.34. Conceptually, both the cart and the camera are controlled by a 6-DOF input device. By using the abstraction of logical input devices, all navigation modes are completely device-independent (see Section 4.1.1).

In the subtree of Figure 4.34, there are a few nodes more than might seem necessary. However, they are needed when one wants to attach other objects to the cart, the viewpoint, or the hands. None of the navigation modes must rely on the assumption that the cart is an immediate child of the root. We might wish to make the cart a child of another object, e.g., in a crash-test visualization, we might want to attach the cart to the car.

I learnt, by experience, that it is highly desirable that all navigation modes can be mapped on all possible configurations of input devices. While some combinations of navigation mode and input devices will be utilized much more often than others, a general mapping scheme comes in handy now and then.

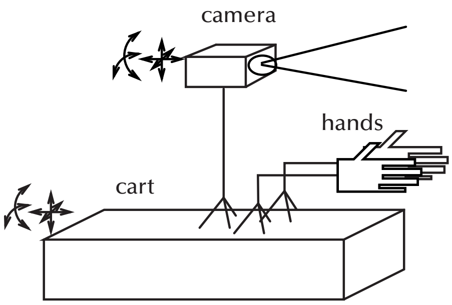


Figure 4.33: All navigation modes can be deduced from the flying carpet model. Not all modes utilize all of the "devices" shown.

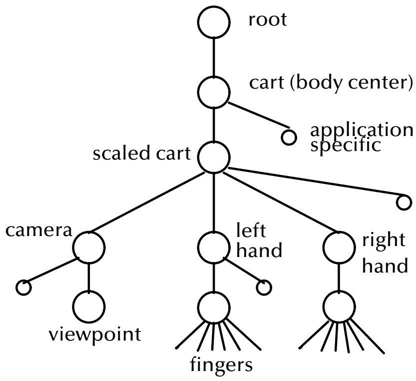


Figure 4.34: The flying carpet model is implemented by this subtree of the scene graph.

The model of the (virtual) user, as far as navigation is concerned, is comprised of navigation speed, size and offset of the hands, scaling of head motion, the head model (see below), eye separation, zero parallax distance, etc.

4.4.1 Controlling the cart and camera

Most navigation modes can be distinguished by the way they control the position of the cart. In general, there is a 6-DOF input device whose value will be used to determine the cart's position (e.g., an electro-magnetic sensor, a Boom, or a virtual trackball); I will call this device the *cart controller*. Another controller (sometimes the same as the cart controller) determines the position of the camera. Usually, the camera controller's values are fed directly into the viewpoint while the cart controller's values are processed before feeding them into the cart's transformations.

One of the most common modes is *eyeball-in-hand*: this paradigm is implemented by feeding the controller's output directly to the viewpoint and the viewing direction, while the cart remains fixed. This technique is most appropriate for close examination of single objects from different viewpoints.

In *point-and-fly* mode the user moves the cart by pointing the controller in the desired direction. The motion is only started when a certain event triggers. An additional one-degree device can be used to control the speed of the motion.

A more sophisticated variation of this mode has been suggested by [MCR90]: it might be named *object-centered* point-and-fly. The user points with a ray at some object he wants to approach. The system scales the speed by the distance to the object, which yields exponential speed. The advantage is accelerated navigation without losing fine control. In addition, the path can be determined by the system, based on the object's surface normal at the intersection point with the ray. In this case the cart will be moved sideways as well such that the user will look straight onto that part of the object. The disadvantage is that users first need to identify an object as the navigation center, which can become quite tedious. Sometimes there are simply no objects to select, for instance in immersive scientific visualization (e.g., galaxy collisions, vortices in CFD visualization).

Contrary to eyeball-in-hand is the *scene-in-hand* mode. Depending on the application, this can be quite useful for orientation or object placement. Especially in "fishtank" (i.e., desktop) VR many users find it more intuitive to rotate the scene rather than the viewpoint.

Sometimes it is desirable to be able to control the viewpoint "without hands". In that case, *speech recognition* can be used in order to move the cart by uttering simple commands such as "turn left", "stop", etc. This has become feasible with today's user-independent speech recognition systems and fast processors. Actually, this mode of navigation can be generalized to what I call the *start-stop* mode: motion in any direction is turned on and off by (different) events. Several orthogonal directions can be combined. This type of navigation is useful to provide very simple and robust navigation for very inexperienced users. I would like to mention an implementation detail so as to avoid confusion: although the cart is controlled and moved, the direction is derived from the viewpoint's coordinate system.

Teleportation is probably the simplest navigation mode. The user just selects a location (via 3D menu or spoken commands) from a menu of possible locations provided by the system. In order to overcome the limitation of a predefined list of possible destinations, [PBBW95] invented the notion of handheld miniatures of the world (WIM's). A WIM is a miniature copy of the VE (a building, say). It could be carried about at the user's left hand, at a "tool belt", or it could be popped up like a 3D menu. The user can then jump to a certain location by pointing at it in the miniature world.

4.4.2 Human factors

The task of navigation comprises two components: a *cognitive* task, i.e., building a mental map of the environment, and an *action*, i.e., giving input to the computer which makes it perform the locomotion by transforming the viewpoint in the desired direction. The mental map is constantly updated during locomotion; the navigation technique and constraints have an impact on the user's ease of building that map.

Depending on the application, the cognitive task can be more or less difficult; for instance, when navigating through large terrains or cities, a map can be very helpful for the cognitive task. In order to improve user performance with navigation, several techniques have been developed, such as non-linear scaling, and world-in-miniature.

Depending on the application, the locomotion component of navigation can be important by itself, e.g., in training fire-fighters or soldiers in VEs, it is important that navigation be as natural as possible.

4.4.3 Constraints

Sometimes it is desirable to constrain the cart. Constraints can be rotational or translational or both.

Translational constraints are commonly used to fix the height of the cart (although the other coordinates can be fixed quite similarly and simultaneously). A simple constraint just keeps one or more coordinates of the carts translation fixed at a certain value, e.g., the height. This can be used to keep the virtual user's position at *eye level*, which increases a natural feeling of certain VEs, for example in a virtual city. A more sophisticated translational constraint is *terrain-following*, which also tries to keep the user's eye level constant while he moves about. In general, this is implemented by shooting a ray in a certain direction and adjusting the specified coordinate of the cart's translation. This can be used to facilitate navigating terrains, or while riding on a piston head or in an elevator.

Rotational constraints are often used to aid inexperienced or irregular users in navigating. It can be difficult for those users to prevent tumbling and spinning.[12] In that case, the horizon can be kept level by disallowing "roll".[13] This can be constrained even further by keeping the cart's up-vector always parallel to the world's up-vector (no roll and no pitch, only yaw allowed).

Another constraint, which is not a translational or rotational constraint *per se*, is the *wall constraint*. This mode prevents the viewpoint from "trespassing" across the boundary of an object (the "wall"). It can be applied to either the cart alone or the viewpoint. When applied only to the cart, the viewpoint is still unconstrained by itself, so that the user can still "stick his head" through walls. When constraining the viewpoint, this cannot occur anymore; however, since the user's head cannot be constrained, "popping" of the viewpoint might occur when the user traces a path in and out of a wall.[14]

4.4.4 A model of the head

For immersive display devices (HMD, Boom, CAVE), in order to decouple the viewing direction from the navigation direction, and in order to provide motion parallax (for large-screen stereo projections), head tracking must be utilized. This is why my navigation model also comprises a (virtual) camera in addition to the cart.

There are 3 coordinate systems involved: the *tracker*, the *real world*, and the *virtual world* coordinate system. They are reflected by several coordinate frame transformations and calibrations. There is only one way to map head tracking input on the virtual camera.[15] Figure 4.35 shows the model of a user's head while Figure 4.36 shows the associated transformations. The position of the virtual camera is given by

$$R_e T_e = R_s(s_K T_s)(s_K T_{sr}^{-1}) R_{sr}^{-1} T_{l|r} R_{er} T_{er}$$

where

R_{sr}, T_{sr}	= head sensor in resting position
R_{er}, T_{er}	= dito for the (assumed) cyclops eye of the user
R_s, T_s	= current head sensor position
R_e, T_e	= current eye position
$T_{l\|r}$	= translation for left|right eye
s_K	= scaling of tracker values ("camera speed")

The resting position is usually defined to be in the origin of the tracking coordinate system, looking down at the $-z$ axis. The rotation R_{er} is normally the identity, however, it is quite convenient for the specification of a virtual endoscopy, which has a tilted "head".

[12] Of course, this is not a problem when there is no rotation of the cart at all, which is generally true when using point-and-fly with HMD or Cave.

[13] Care must be taken in the implementation to avoid gimbal locking when doing loopings.

[14] This popping can be alleviated by a fade-in/fade-out technique. Still, there will be some motion of the viewpoint which is not directly related to the user's head motion.

[15] Other mappings could be devised and might be fun to investigate, but the canonical one seems to be the only one relevant in applications.

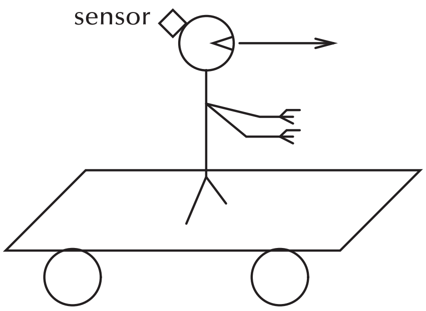


Figure 4.35: Model of a user's head.

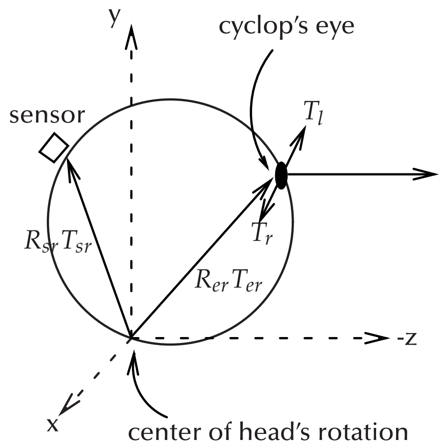


Figure 4.36: Transformations associated with the head model.

Some tasks in certain VEs necessitate large scale navigation as well as fine control navigation. For instance, in immersive visualization of CFD simulations, one might want to navigate through the field quickly but still be able to investigate fine details of the field. If the cart and the camera can be controlled independently, this can be achieved trivially by assigning different "speeds" to them.

However, if only the camera can be controlled by the user (through head-tracking, say), a technique similar to non-linear point-and-fly can be utilized [SN93]. The idea is to scale the head-motions when outside a certain "domain of interest", which is usually sphere around the tracker's origin (and thus around the cart). So, the scaling s_K becomes

$$s(T_s) = \begin{cases} s_K T_s & \text{if } ||T_s|| \le d_0, \\ s_K \frac{1}{||T_s||} e^{d-d_0} T_s & \text{otherwise.} \end{cases}$$

Within the sphere of radius d_0, the motion of the camera is linear, outside it is scaled exponentially with the distance from the origin (see Figure 4.37). The radius d_0 can be made zero and s_K can be made small so as to allow "microscopic" and "macroscopic" motions. The function $||s(T_s)||$ is $\mathcal{C}^1$-continuous in $||T_s||$. Of course, other scaling functions can be used as well; they should just be $\mathcal{C}^1$-continuous, so that the transition is smooth.

Non-linear scaling techniques for the camera work only, unfortunately, if there does not have to be a registration between real and virtual viewpoint. This is the case for HMD, Boom, or scene-in-hand paradigms. However, in a Cave or in front of a large screen with head-tracking, this method does not work, because the images will appear distorted when there is a mismatch between real and virtual viewpoint (see Chapter 4.3.2).

4.4.5 Implementation

Practically all navigation modes can be implemented by the following procedure:

`Navigation`

get new deltas $\Delta R, \Delta T$ *from controller*

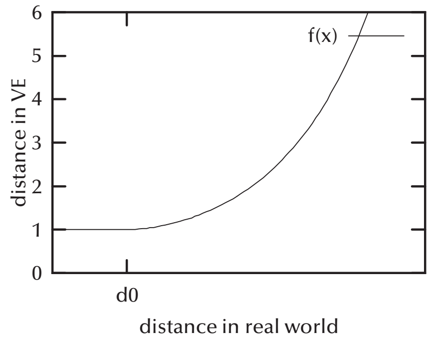


Figure 4.37: Scaling the head controller's translation can allow for "microscopic" and "macroscopic" motion.

Figure 4.38: With a Boom for interior design, the eyeball-in-hand navigation technique is used.

if there are no new deltas, use the old ones
scale $\Delta R, \Delta T$ by rendering time and user-specified speed
$T' := T \pm \Delta T \times R$, $R' := \Delta R^{\pm 1} \times R$
  constrain T' and R' (independently)
$T := T', R := R'$

In order to ensure a smooth motion the cart must be moved every frame even if there are no new data from the input device. Also, rendering time varies with each frame; in the worst case, a frame might take twice as long as the previous one (60 Hz versus 30 Hz)! Therefore, we need to acquire "deltas" from the controller and scale them such that a smooth motion is obtained. Scaling rotations can be done nicely via the "axis+angle" representation.

The difference for point-and-fly is that the controller provides absolute instead of relative values, and these absolute values are treated like relative deltas. Similarly, for start-stop ΔR and ΔT are constant values depending on the command which has been triggered.

4.5 Interaction techniques

4.5.1 Virtual buttons and menus

Sometimes it can be convenient to trigger an action in the VE by pressing the analogue of a 2D button in VR. This is called a "virtual button".

There are many techniques how virtual buttons can be implemented. By plain collision detection between a finger and the button object, by bounding box tests, by ray-object intersection, or by cone selection.

Virtual menu can be realized as an array of virtual 3D buttons. This seems to be appealing, because that way menus are just a generalization of buttons (and a lot of code can be re-used). In addition, when a 3D menu is just a sub-tree of the scene graph, it can be modeled and rendered like any other geometry.

Two main problems arise with menus: (1) where to place them, and (2) how to select an entry. Another (minor) issue is the steps needed on behalf of the user to pop up the menu and to close it again. It is not easy to place menus such that entries can be selected with ease, yet do not obstruct the view. This

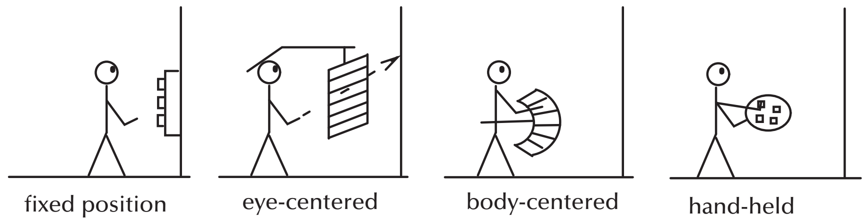


Figure 4.39: A lot of metaphors exist to guide the placement of menus in VEs.

is especially difficult when they must be kept around all the time while doing other tasks.

In our first implementation, the user selected the button of a 3D menu by touching it with the index finger. However, experience has shown that such menus are very troublesome to use in VR.

My next implementation utilized ray intersection. In this paradigm, the user sees a ray emanating from his palm or finger. A menu button is selected when it is hit by that ray. That way, more buttons can be put in a menu and the placement is much more flexible. Although this has been a great improvement, one still needs a steady hand, just like with general selection.

Finally, I implemented an implicit ray intersection technique: the ray is defined by the user's viewpoint and the tip of the index finger (or thumb). This gives much finer control, and the paradigm is closer to 2D menus and pointers, which is easier to explain to users accustomed to the desktop paradigm. This seems to be the best choice so far.

In order to address the problem of positioning the menu, a number of techniques have been tried (see Figure 4.39):

1. The simplest way is to have the creator of the geometry, or the user, decide where to put a menu. The menu remains stationary all the time, unless moved by the user. It could be visible all the time, or it can be switched visible when needed.

   Of course, this requires that the (approximate) position of the user be known in advance to the person modeling the scene. With that knowledge, however, one can ensure that the menu is visible for the user and that it does not obstruct other geometry (if possible).

   Examples for that kind of "stationary menu" are objects which are part of the VE by themselves, such as elevator buttons or remote controls.

2. If the menu is merely an "auxiliary object" of the VE, then the VR system must decide where to put the menu when being popped up by the user.

   One way is *head-centered positioning*. For instance, it could be put it in front of the viewpoint and within a distance such that it is completely visible and as close as possible. The menu can be placed right in front of the viewer or a little bit on the side or below. It can be positioned only when it is popped up, or continuously while the viewer is turning his head.

   There are a number of problems with this approach. With stereo-viewing, the positioning algorithm additionally has to take into account eye separation and zero parallax distance, so that the parallax of the menu does not get strainful on the eyes.

No matter if continuous or pop-up-time positioning is done, the position of the menu is never quite right: either it obstructs part of the scene, or it is not completely visible.

3. In order to overcome the problems of head-centered positioning, *body-centered positioning* has been investigated. Analogously to [BBDM98, MBS97], menus can be positioned relative to the user's body. If body tracking is not available, the body's position can be estimated based on head tracking.

   The idea is to attach the menu to the user's "belt". For instance, several tools' icons can be arranged around the user's waist in resemblance to a tool-belt. If there are more icons than would fit comfortably on the front of the belt, then they can be scrolled, i.e., the belt can be turned (scroll icons).

   With body-centered positioning, the problem of obstruction is reduced a lot. However, selection of the icons is a bit more difficult.

4. If both hands of the user are tracked, then *hand-held* (i.e., hand-centered) positioning can be utilized. In this metaphor, the user holds the menu on his left hand, while selecting entries (e.g., icons) with the right hand. The idea is that the menu is "carried" by the left hand like a painter's palette or like a tray.

   The hand-held metaphor is very appealing, because it solves the positioning problem. The user can place the menu himself in a comfortable position; stereo parallax is not a problem, because stereo parameters must be set such that the hands can be viewed without eye-strain. Finally, selection is efficient, because the menu can be brought close to the right hand, and human factors research has shown that positioning the right hand on an object can be done much more precisely, if the left hand touches the same object (e.g., keyboard). This is due to kinesthetic feedback.

   This positioning metaphor has been applied to many other interaction techniques [PNW98, PBBW95, WMB98].

In addition to the problems already mentioned, 3D menus seem to be incapable of holding as many entries as their 2D counterparts. This is mostly due to limited display resolution, size and positioning problems. However, I suspect that there are also some human factors issues involved.

At the moment, I believe that if one must use menus in VR, then 2D menus should be used. Our VR system provides a general mechanism to describe menus and trigger actions from them. The text of such menus is displayed as an overlay on the scene.[16] Menu items can be selected either by augmented postures (e.g., "thumb-up" moves the highlighting bar up one position, "thumb-down" moves down; see Section 2.2.8), by sending a ray from the eye through the menu to the finger, by moving the hand up and down (which seems to be most efficient), by speech recognition, or by head tracking [MBS97]. If a joystick is being used as pointing device, then its buttons can be used for controlling the menu.

[16] Menus will occlude everything if displayed as overlays (whether in a heads-up manner or as solid overlays). Therefore, in the case of stereoscopic rendering, they should be displayed with negative parallax, so they would appear to be floating in front of the VE. It is not trivial to compute the perfect parallax, but heuristics seem to be quite sufficient.

In order to retain at least a 3-dimensional appearance of menus, one can resort to "M-cubes" [WS94], or spheres or circles. However, I believe that such representations are inappropriate when a large number of items has to be presented in one menu, which is the case for virtual prototyping applications.

An interesting approach are "mark menus" [Kur93], which are somewhat the 2D analogue of sphere or cube menus. Using a mouse or tablet, user efficiency can be increased by a factor 3.5. When a user has become an expert, he can use such menus "blindfold", i.e., without actually popping up the menus, just by "marking" the pencil or mouse motions. However, users can access only 4×4 or 8×2 menu items by "blindfold" marking.

My findings are somewhat in contrast to the work of [Jay98], who propose 3D menus for virtual prototyping applications.

4.5.2 Selection

Like menus, selection is one of the very frequent interaction tasks. Most tasks involving manipulation of objects require the user to first select the object. Only in rare cases the object itself can be "hard-wired" or inferred by the system. As with menus, there are a number of difficulties when 2D-techniques for selection are implemented in 3D. However, a lot of research and experimentation is still going on about which metaphor combines best with each input device.

In 3D a common paradigm for selection is the *ray* which emanates from a 3D (or 6D) pointer. This metaphor has two parameters: the direction of the ray and its lateral "extent". When the ray is a line, selecting objects far away can be a problem with this metaphor: it can be difficult to hit the right object, especially when the tracker signal is noisy. Instead of using a ray, a cone can be used. However, this does not help to select objects very far away and/or very tiny, because with a cone ambiguities can arise. Another modification of the ray metaphor is how the direction of the ray is specified by the user. Instead of using the pointer's all 6 dimensions, only its position can be used; the ray is sent from the user's viewpoint through the pointer's position (for instance, the tip of the finger of the virtual hand). The advantage is that the ray's direction is less susceptible to noise in tracker data.

Another metaphor are "windowing" techniques. The window is a certain (rectangular) region on the screen. Objects are selected by assuming a viewpoint such that their projection is within that window. As with cone selection, ambiguities arise when several objects are inside the window. This can be solved by picking the one which is closest to the center of the window. There are different ways how the window can be specified by the user. One way is not to have him specify it at all, i.e. it is fixed. Another way is to enclose the region by the user's (or the virtual hand's) forefingers and thumbs, which requires two gloves, or by spanning the region's opposite corners by one thumb and forefinger.

It might be desirable to select objects by speech alone. However, this presents some non-trivial problems. If objects are to be selected by their name, then they must have "easy" names. However, this is not the case with manufacturing scenarios, where objects usually have names like `E38_E-Geraet2_08` or `ee_-04.pklappe_gro`. So, in order to use object names for selection, a map would have to be constructed with every scenario. In addition, the distinction among several instances of the same type of object could become tiresome (e.g., when there are 10 light bulbs in the scenario, or 100 screws). Furthermore, this kind of selection requires speech recognition which does not need to be trained, even for these very specific words.

A hybrid selection scheme based on speech recognition and menus might be practical. The system would present a 2D menu of all the objects to the user. Then the user could select one of these menu entries as described above (see Section 4.5.1), for instance by saying "item five". If there are 10's of objects or even hundreds of objects in the scene, the menu has to be organized hierarchically; which can be non-trivial if the system has to do this automatically.

Another hybrid technique integrates AI with VR. A user can select objects by "imprecise", context-dependent, situative descriptions [Sch96, LW98]. Such descriptions could contain attributes like color, size, place, for instance, "select small red part on the right". The user could point approximately in the direction where the object is, in order to narrow down the set of matching objects, and utter a command containing a deictic term like "select this tool". This approach might soon require the full gamut of speech recognition, sentence analysis, knowledge representation, etc.[17] [ODK97] have found that selection was done only very rarely in a multimodal manner. However, I believe this is due to their application which already features an efficient pen-based selection mechanism.

So far, the selection methods involved speech recognition or an abstract device like a ray or window. If the graphical echo of the user's hand or pointing device is to be used, then some mechanism is needed so that he can reach any object (at least all of those relevant to his task). This can be achieved by nonlinear scaling of the tracking data when in selection mode [SN93]. There are many possibilities to choose the scaling function. One of them is an exponential function, i.e., the further the hand is moved away from the body the larger the scaling. That way, the user can touch any object which selects it. When the "interaction radius" is made too large, the same problems as with ray methods arise: precise positioning of the virtual hand/pointer becomes difficult.

Of course, it is necessary to be able to select several objects, so that all of them can be processed (e.g., for distance computation). This can be achieved by making selection a 3-step interaction: enter selection mode, select each object, exit selection mode.

4.5.3 Grasping

Grasping objects is one of the most fundamental interaction techniques in VEs, which comes as no surprise since it is also one of the most frequent activities in the real world. Because of that, I have implemented grasping as an elementary action (see Section 2.2.10).[18]

The grasping interaction can be realized in (at least) three different ways: single-step, two-step, and naturally. Single-step grasping attaches the object to the hand at a certain event (e.g., a spoken command like "grab thing").

Two-step grasping comprises the following interaction steps:

1. Some event (e.g., a gesture or the spoken command "grab") switches grasping on; at this point, the hand is made "sticky".

[17] A user monologue like the following might happen: "select transparent small object over there", "the red one, not the blue one", "the one behind that one".

[18] As pointed out in Section 2.2.10, the grasping technique can be considered much more general; should that be necessary in the scenario of a VE, this can easily be implemented by "hand".

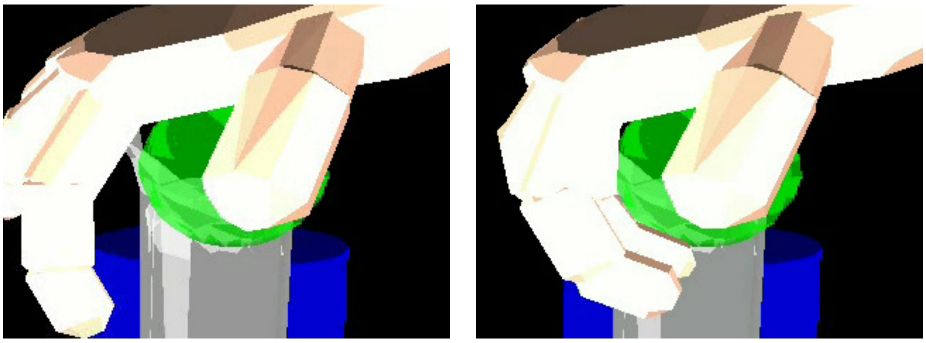


Figure 4.40: Natural grasping is basically a minimization problem for the flex vector under the constraint that finger-joints (and palm) must not penetrate the object.

2. Then another event actually attaches the thing to the hand. This is usually triggered by a collision between the hand and some object (see Section 2.2.10). Of course, other events such as the selection with a ray are sensible as well.

Releasing the object will still be done by one interaction step only, of course, which is usually triggered by the same event as step 1.

Attaching an object to the virtual hand can be done in two different ways: either by re-linking the object in the scene graph making it a child of the hand node, or by maintaining a transformation invariant between the hand and the object. If the object is re-linked, then a new transformation must be set for it, so that it does not "jump". The new transformation is $M'_o = M_h^{-1} M_o$, where M_h is the transformation of the hand at the time of re-linking. If the object is not re-linked, then the transformation invariant $M = M_o M_h^{-1}$ must be maintained [RH92]. This can be done by updating M_o every frame with $M_o^{t+1} = M_o^t (M_h^t)^{-1} M_h^{t+1}$.

Natural grasping

The way of grasping described in the previous section is simple to implement, but not really intuitive. On the one hand, the user has to *know* that she first has to issue a speech or gesture (usually the fist) command; on the other hand, the object being grabbed "sticks" to the hand in an unnatural manner.

What one really would like to do is just close one's fingers around an object (see Figure 4.40). The system would need to make sure that fingers do not penetrate the object, and it would determine when an object is grabbed firmly so that it can be moved. So, the user would not need to remember a command, and objects cannot be grabbed by the back of the hand.

Like with force-feedback devices, we need to distinguish between (at least) 4 types of grasping:

1. *Precision grasping* with three sub-types [Jon97]: tip pinch, three-jaw chuck, and key grasp,
2. *cigarette grasping*,
3. *3-point pinch grasping*,
4. *Power grasping* (or just *grasping*),
5. *Gravity grasping* (or *cradling*).

Precision grasping involves 2 fingers, usually the thumb and one of the other fingers; it is used for instance to grasp a screw. Cigarette grasping involves two neighboring fingers; it is usually used to "park" long thin objects, such as a cigarette or pencil. 3-point grasping involves three fingers (one of them being the thumb), giving the user a fairly firm grip, and allowing him to rotate the object without rotating the hand. Power grasping involves the whole hand, in particular the palm. With this type of grasp the object is stationary relative to the hand. Gravity grasping is actually a way of carrying an object.

Related work

There are several papers describing related work. A system for automatic creation of grasping animations was presented by [ST94]. The system is based on a predefined set of grasping postures (similar to the above list) and a classification of the object into a small set of basic shapes (sphere, cylinder, etc.). A similar system was presented by [RG91]; they also look at different levels of the task to be synthesized. Algorithms for precision manipulation of objects by a virtual hand was presented by [KH95]. Their algorithm is based on the notion of a "finger-tip triangle", the motion of which determines the transformation for the object being grasped. Their system also distinguishes between 3 modes (free, push, and grasped). [RBH$^+$95] present a simple automaton consisting of 3 states and a simple top-down joint locking algorithm. This work has been taken further in [BRT96], in which they sample a virtual hand by several "collision sensors". A simple automaton is associated with each finger, controlling whether it should move the object or slide along its surface.

In the following I will describe an algorithm for solving natural power grasping and precision grasping. These are the grasp types most often needed in assembly simulation. My algorithm does not need to switch between modes. In addition to grasping, my algorithm can determine a *push*, which moves the object being touched into the direction the hand is moving. Precision handling, like turning a screw between two finger tips, is not yet implemented.

The algorithm

For power grasping, the algorithm consists of two simple parts: clasping the fingers around the object (see Figure 4.41), and analyzing the contact. The former will be done by an iteration, while the latter is implemented by a simple heuristic.

The position of the hand is completely specified by (M, F), where M is a matrix specifying the position of the hand root, and F is the *flex vector* (usually 22-dimensional). Given a new target hand position (M^n, F^n), the goal is to minimize $(|MM^{n-1}|, |F - F^n|)$ such that (M, F) is collision-free. Note that the position of a finger-joint depends on its flex value and all flex values higher up in the chain and the position of the hand root. Therefore, I suspect that there are several local minima, even if we only consider flex values during optimization (and keep the position fixed). However, this should not be a problem if the minimizatin process is fast enough, so that consecutive collision-free hand positions are not too "distant" from each other.

Minimization must not be done using the visible model of the hand; otherwise, the user would "witness" the process (because the renderer runs concurrently). So, a copy of the hand tree is used for collision detection, and only after minimization has finished, the new position/flex values are copied to the

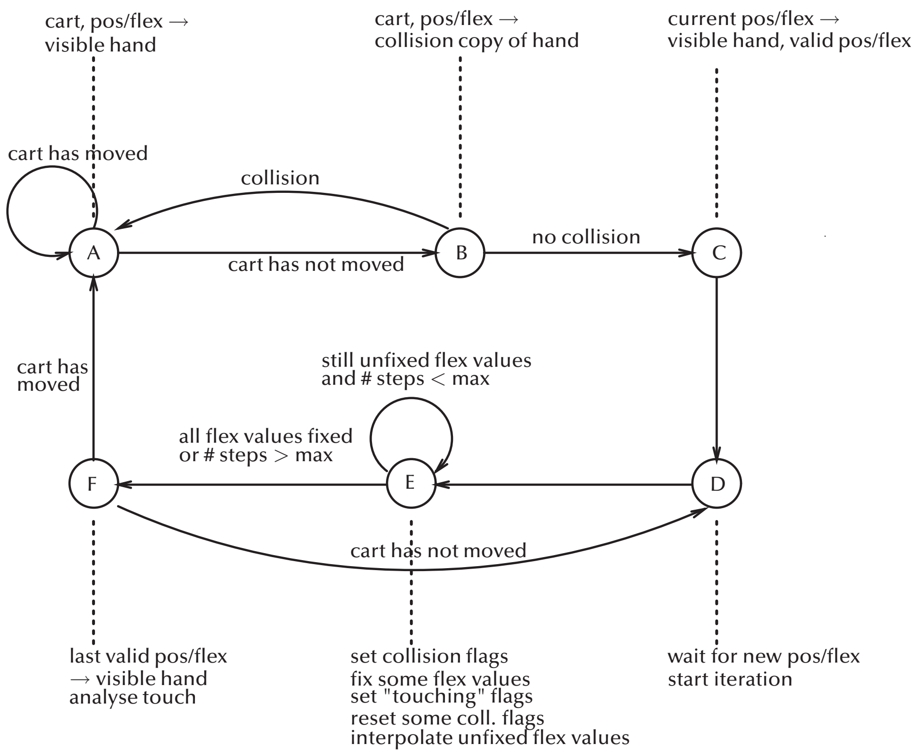


Figure 4.41: A simplified overview of the algorithm for simulating natural grasping.

visible hand. This minimization should be as fast as possible, so it should run as a concurrent process; otherwise, the user might notice considerable latency.

In order to find the optimal flex vector, I use an iteration process interpolating non-colliding (i.e., valid), and colliding (i.e., invalid), flex values. Here, a flex value is *colliding* if its associated finger-joint is colliding *or* any finger-joint depending on it. A finger-joint J' is *depending* on a finger-joint J, if it is further down the kinematic chain, i.e., if J moves, then J' moves, too. Note that a finger-joint can have many depending finger-joints. (In this context, the palm is a "finger-joint" like all the others.)

During the iteration process, the position M of the hand is treated like any other flex value, i.e., it is interpolated. The only differences are that interpolation is done on matrices instead of single real numbers. The "joint" associated with it is usually the palm or the forearm.

After a few iteration steps, some flex values will be approximated "close enough" (when the range between valid and invalid flex value is small enough). Then, they will be *fixed*. Depending flex values must be considered for fixing, too: they may or may not be close enough. So, several flex values in a row may become fixed at the same time. As long as a flex value is not fixed, it will be interpolated *and* all its depending flex values. (An alternative would be not to interpolate depending flex values, but since depending finger-joints need to be checked for collision anyway, we can as well interpolate them, too. Thus, we probably achieve an optimum faster.)

During iteration, the algorithm has to mark all finger-joints (including the palm) which are *touching* the object. If a finger-joint is not touching the object, I will call it *free*. At the beginning of the iteration, all finger-joints are free. When a finger-joint collides, the algorithm sets a collision flag for it. When that flex value gets fixed, the finger-joint is marked as "touching" if the collision flag is set (the collision may have happened several iterations earlier). After a flex value has been fixed, the collision flag of all depending flex values will be cleared again. This is because possible collisions of depending finger-joints are due to motions of up-chain finger-joints and not because the depending finger-joint is touching.

After all flex values have been fixed, the touching analysis tries to determine the type of grasp. While the clasping algorithm is in general applicable to any hierarchical kinematic chain, the analysis algorithm needs to know more about its "semantic", i.e., it has to know about a palm, it needs to know which finger-joints belong to the same finger, etc. The heuristic I have implemented is very simple:

1. only one finger-joint or palm is touching → push;
2. several finger-joints are touching, and none of them is part of the thumb, and the palm is not touching → push;

   This part of the heuristic would need to be more sophisticated if cigarette grasping should be recognized. However, this type of grasp is not needed for virtual assembly simulation.
3. one or more finger-joints is touching, one or more thumb-joints is touching, and the palm is not touching → precision grasp;
4. one or more finger-joints (possibly a thumb joint) and the palm are touching, and at least one of the finger-joints is a middle or outer joint → power grasp;
5. one or more finger-joints and the palm are touching, but all finger-joints are inner joints → push.

In my algorithm, motion of the cart is handled specially. A cart motion indicates that the user's (virtual) body is changing place (see Section 4.4). Since the hand is attached below the cart, a cart motion always brings on a motion of the hand. In that case, the algorithm does not try to clasp the hand tightly around an object, because that might cause the hand to be left behind, which is probably not what the user wanted. Unfortunately, navigation might cause the hand to end up in an invalid (i.e., colliding) place, so after navigation has stopped, the clasping algorithm cannot begin until the whole hand has been moved to a collision-free place by the user.

Future work

The next step should be to implement 3-point grasping, so that a user can turn and rotate objects without turning his hand. A little bit more difficult is probably the possibility to turn objects with 2 fingers only (precision grasp). This type of interaction (and probably others, too) relies heavily on the skin, in particular its deformability and its high friction.

The human hand is a very complex "mechanical" device. I believe it will take a few years of research to simulate object manipulations like juggling Qigong-balls in one hand or turning a screw with two fingers.

Moreover, the human hand is able to perform several grasps at the same time, such as: holding a screw with a precision grasp between index finger and thumb while pushing another object with the little finger; or, a precision grasp involving index finger and thumb while holding another object with a gravity grasp; or, handling a pair of scissors with thumb and index finger while holding a comb with a cigarette grasp between little finger and ring-finger; etc. It remains to be seen whether or not the approach taken here is suitable to simulate these complex tasks. On the other hand, alternative approaches like physically-based ones also have yet to show their suitability for solving such complex simulations.

Although grasping is more natural than before, I feel that more work needs to be done to improve the user's interaction efficiency. This is supported somewhat by [JJWT99].

4.5.4 Sliding

Being able to grasp objects is, for some application domains, not enough (see Section 5.2.2). What the user really wants is that the hand and objects being grabbed interact with other objects as in the real world, i.e., they must not penetrate other objects, and "forcing" the object while colliding with others should make it slide or glide along a path around that other object such that they touch but do not penetrate each other along that path.

As of this writing, force-feeback systems and algorithms are still not mature to provide this kind of object behavior. It is possible today to render forces for a single point or with reduced accuracy at interactive rates. However, for virtual assembly simulation force-feedback for complex objects consisting of large polygon counts and for a large work volume is needed. But even when such force-feeback systems will become doable, algorithms similar to the one described below must be implemented in order to render forces.

This is a variant of grasping where the tranformation from hand coordinate system to object coordinate system is no longer invariant. Instead, the motion of the object is determined by physically-based simulation. For the sake of clarity, let us assume that the object is collision-free at the time when it is being attached to the hand. Let us assume further that at that moment we make a copy of the object which is allowed to penetrate all other objects and which is being grasped firmly by the hand. I will call this copy the "*ghost*" of the object. It marks the position where the object would really be if there was no collision. There are, at least, three metaphors for guiding the simulation:

- The rubber band metaphor: the object is connected to the ghost by a rubber band. This tries to pull the object as close to the ghost as possible without penetrating.
- Rubber band and spiral spring: like the plain rubber band metaphor, but the object is also connected by a spiral spring to the ghost (this is a little bit difficult to picture). In the plain rubber band metaphor, the user has no control over the orientation of the object — it is completely determined by the simulation.
- Incremental motion: when the ghost has moved by a certain delta the object will try to move about the same delta (starting from its current position). If there is a collision during that delta, then the simulation will determine a new direction. So, alternatingly the object is under the control of the user and under simulation control.

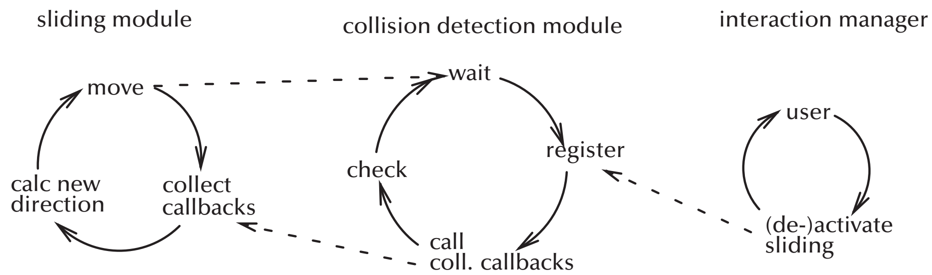


Figure 4.42: The physically-based simulation module for sliding runs concurrently to the other two main loops of the collision detection module and the interaction manager. Dashed arrows mark rendezvous points.

I have implemented the latter so far, but maybe the "rubber band and spring" might be more intuitive.

In the discussion below, I will explain the algorithm in more detail. The reader should keep in mind, that the goal was *not* to make the sliding behavior of objects as physically correct as possible. Readers interested in physically correct simulations should refer to the wealth of literature, for instance [Bar94, GVP91, Hah88b, SS98, BS98].

Instead, the goal was to develop an efficient algorithm which helps the user to move the object exactly where he wants it, and which helps the user achieve that in minimal time even in closely packed environments (such as the interior of a car door).

The main loop

First of all, I will describe the "big picture" of the simulation loop. In subsequent sections, I will go into more depth.

From a very high-level point of view, the main loop is just

```
move object check collisions determine new direction
```

This a very gross simplificaion in twofold respect: the collision detection module runs concurrently, and calculation of new directions consists of several phases.

Because the collision detection module runs concurrently, we need to implement the physically-based simulation so that it can handle that. Therefore, it does not complicate things much more, if we implement the simulation module such that it runs concurrently to the main loop of the interaction manager itself. So we can think of the three loops communicating with each other as depicted in Figure 4.42.

Parallelization does really gain something when collision detection is faster than rendering. If the simulation would be part of the interaction manager process, then the simulation could make only one transition per frame in the state machine. This is not acceptable, considering that the simulation has to make several steps in order to find a good approximation of the exact contact point (see below).

In the simulation, there is a so-called *collision object* (short *collobj*) which is invisible.[19] It is used to check intermediate position for collisions. The *visible*

[19] The geometry of the collision object is, of course, exactly the same as that of the visible object. If the scene graph API allows for it, they can share their geometry.

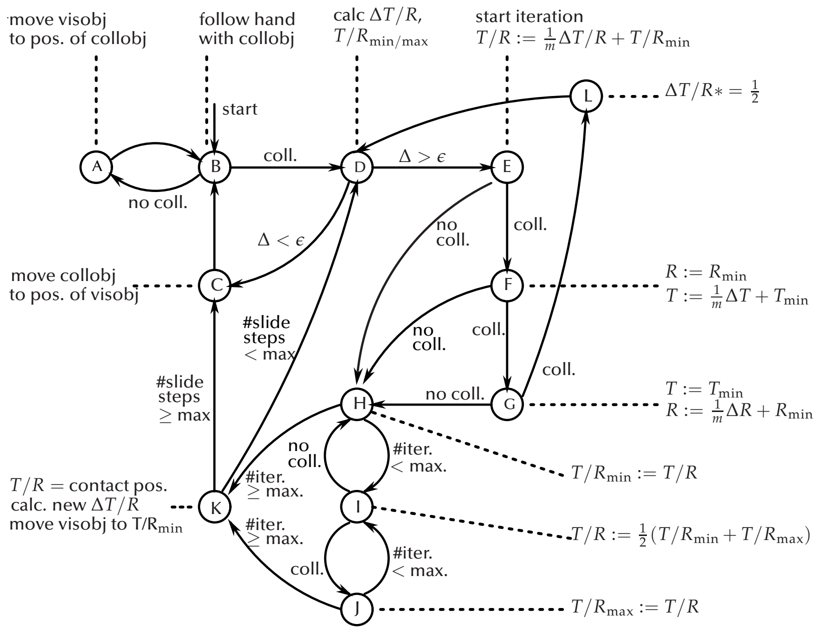


Figure 4.43: The main loop of the sliding simulation is a finite state machine.

object is the one users are really seeing. It is never placed at invalid (i.e., colliding) positions. So the user only sees a valid, i.e., collision-free path of the object.

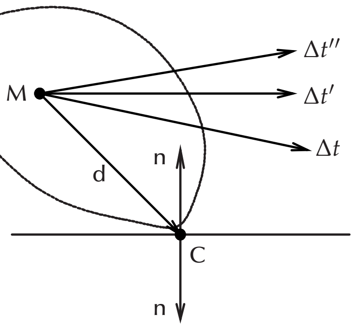
The algorithm works, simply put, as follows:

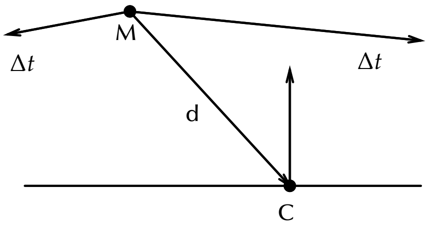
```
Sliding simulation

loop:
   while no collision
      move visible and coll. object
      according to hand motion
   {now the coll.-object is penetrating}
   approximate exact contact point
   classify contact
   calculate new direction
```

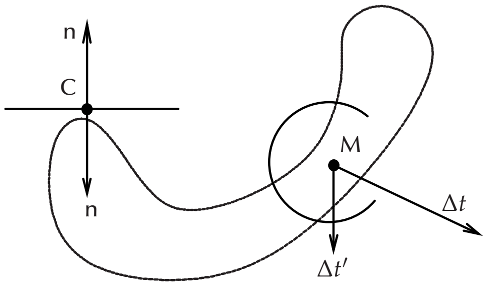
The main loop of the simulation module is actually a finite state machine. Figure 4.43 shows that in a little bit more detail. It is still omitting certain (pathological) cases which can occur in practice, but for the sake of clarity I will not go into too much detail here.

When approximating both T and R of the exact contact position simultaneously, sometimes there is no collision-free position closer than $T/R_{\min}$. But it would be possible to find a position "closer" (at least in an intuitive sense) by approximating only the rotation, for instance. That is why the state diagram contains states F and G. State E finds out whether or not a simultaneous approximation is feasible.

I do the contact approximation by interval bisection and a number of static collision checks. [ES99] propose a dynamic collision detection algorithm. However, it is not clear that this would really speed up the simulation in this case,







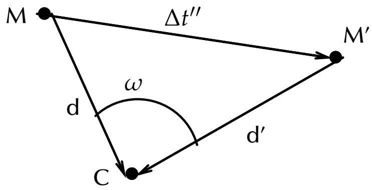


Figure 4.44: With one contact point, calculation of the direction of the new translational velocity is the same whether or not the contact normal points to the "right" side.

Figure 4.45: We can estimate the new rotational velocity based on the assumption that C must be stationary during the next step.

since the dynamic algorithm takes about 3–5 times longer and an exact contact point is usually not needed here.

New directions

Let us assume that we have the exact contact position. Let us further assume that we need to handle only practically relevant contact situations. Then we will need to deal only with the following cases: 1 contact point, 2 contact points, and ≥ 3 contact points, which I will discuss in the following.

In each case, we must be able to deal with "wrong" contact (or rather, "wrong" surface) normals. In general, polygonal geometry imported from CAD programs has "random" surface normals in the sense that the vertex order is *not* consistent across adjacent polygons. But even if it were, we would have to be able to deal with such a situation, because unclosed objects (like sheet metal) does not have "inside" and "outside". With such "sheet objects" we might be colliding from either side.

My implementation of the sliding algorithm presented here allows for arbitrarily pointing normals. They can even be "inconsistent" in the sense that adjacent polygons' normals can point on different side.

In the following, let Δt be the current translational "velocity", let n be the contact normal, let M be the center of mass, let C be the contact point.

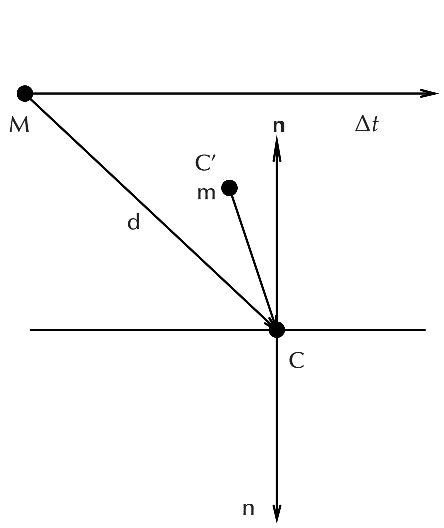


Figure 4.46: Taking the history of the contact point into account is a more robust criterion for choosing the "right" contact normal.

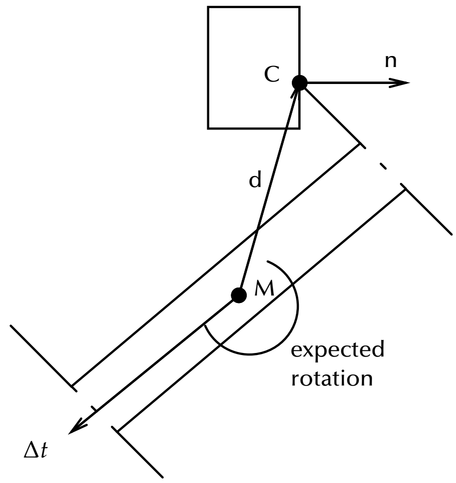


Figure 4.47: This counter example rules out a simpler but unreliable criterion for choosing the right sign of the contact normal.

1-point contact. With 1 contact point, the new translational velocity $\Delta t' = n \times (\Delta t \times n)$ (see Figure 4.44 left). Here, we do not need to check whether or not n points to the "right" side.

Experience has shown that it is much user-friendlier if we let the object slide on kind of an *air-cushion*. That means that $\Delta t'$ should be turned a little bit away from the contact plane. So, we really use $\Delta t'' = \Delta t' R(\pm\Delta t \times n, \alpha)$. Because n could point to the "wrong" side, we choose the sign of the rotation axis of R such that $\Delta t''$ and Δt enclose the larger angle.

Figure 4.44 shows on the right that sometimes it can happen that Δt points away from the contact surface and we still get a contact (because of rotation). In that case, it is more appropriate to make the object move further away from the contact surface, i.e., choose $\Delta t''$ parallel to n.

The rotation axis of a 1-point contact is simply $R = d^0 \times n^0$ (see Figure 4.45). This means that I ignore friction completely (see below for a discussion), so that a sphere on an inclined plane would just skid and not rotate.

We can estimate the rotational speed as follows. During the next simulation step, the object should rotate about R through C, i.e., C is stationary. This gives an estimate for the rotational velocity ω (see Figure 4.45 right). As with translational velocity, we should allow for some looseness by increasing ω a little.

How do we find out the correct sign of the collision normal? Making the sign of n such that $dn < 0$ does not always produce the correct result — see Figure 4.47. I have implemented a criterion based on the "history" of the contact point C: let C' be the last free position just before the first colliding one; then, I choose the sign of the normal such that $(C - C')n < 0$ (see Figure 4.46).

2-point contact. For 2 contact points, the new translational velocity is $\Delta t' = n_1 \times n_2[(n_1 \times n_2)^0 \Delta t]$ (note that $[\ldots] < 0$ if $n_1 \times n_2$ points in the "wrong" direction) and $\Delta t'' = \Delta t' R(\pm\Delta t \times (n_1 \times n_2), \alpha)$ (see Figure 4.48).

The new rotation axis for a 2-point contact is $R = \pm(C_1 - C_2)$ (see Figure 4.48). We determine the sign of R so that $M + \Delta t$ is on the positive side

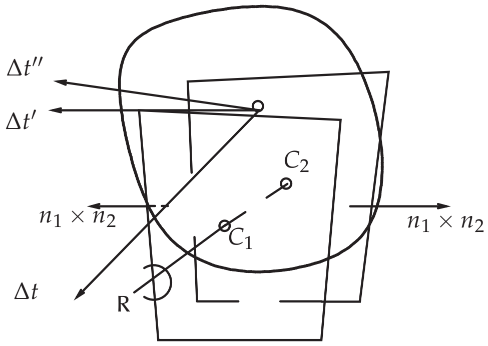


Figure 4.48: New translational and rotational velocity for a 2-point contact.

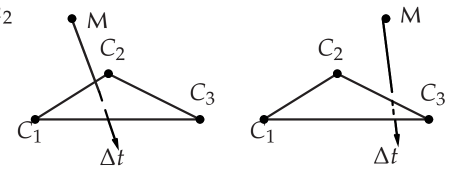


Figure 4.49: With 3-point contacts, 2 different situations must be distinguished: when looking along the old translational velocity, M can be "inside" the convex polygon spanned by C_1, C_2, C_3 or "outside".

of the plane given by $[C_1, R \times (M - C_1)]$. The new rotational velocity can be estimated as above (see Figure 4.45 right), except that we need to use the projection of M on R instead of C_1 or C_2.[20]

3-point contact. With 3-point contacts, there are two different situations (see Figure 4.49): when looking along the (old) translational velocity, M can be "inside" the convex polygon spanned by C_1, C_2, C_3 or "outside". If it is "inside", then this is a stable configuration; so, there will be only a translation parallel to C_1, C_2, C_3. If M is "outside", then we can treat this like a 2-point contact.

More than 3 contact points are similar to the 3-point situation.

Classifying the contact

In theory, we need to handle only two contact situations: vertex/face and edge/edge. Given one pair of touching polygons (p, q), we can determine the contact situation by the following simple procedure: let n_p be the number of polygons adjacent to p and touching q; define n_q analogously. If $n_p = n_q = 1$, then we have the edge/edge case. Otherwise, either n_p or n_q must be > 1 (but not both), and we have the vertex/face (or face/vertex) case.

The advantage of this way of classification is that the collision detection algorithm can stop as soon as it has found one pair of intersecting polygons (provided there is only one contact per pair).

In practice, a few more cases can happen. Partly, this is due to the mere approximation of the contact position, partly, it is due to non-closed geometry.[21]

If $n_p = 0$, for instance, then this polygon might be at the rim of the object. In order to decide that, we need to check if any edge of p intersects q (see Figure 4.50). If so, we have got the edge/edge situation. If not, then it is the vertex/face situation. Note that both $n_p = n_q = 0$ is possible.

[20] Using the projection of M on R is still an approximation, because the projection of $M + \Delta t'$ on R does not necessarily coincide with the projection of M on R.

[21] Non-closed geometry is fairly frequent in virtual prototyping: all sheet metal is non-closed. Now imagine the possible contact situations when a pipe is to be fitted into a hole of sheet metal.

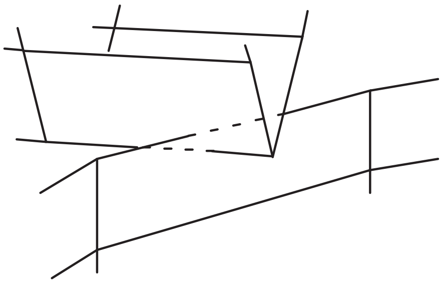


Figure 4.50: Contact classification can be done by looking at adjacent polygons and counting edge/face intersections.

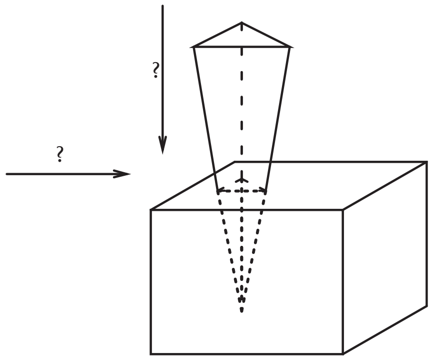


Figure 4.51: Is this an edge-edge contact, or a vertex-face contact? If discretization steps are large compared to the size of objects/polygons, then the objects' history should be taken into account.

It is not sufficient to just check edge/face intersections of the two intersecting polygons (counter-example: two intersecting wedges).

However, sometimes it seems that the history (i.e., the path) of polygons must be evaluated to decide the kind of contact (see Figure 4.51). Such an ambiguity occurs always when the motion discretization is large compared to surface discretization. It is not clear to me yet whether or not this occurs often in practice.

When to stop

The sliding algorithm does not move the object as close as possible to the ghost object. Instead it moves it about the same delta as the ghost has moved. Therefore we need a criterion when to stop motion (see Figure 4.52).

The trivial stop criterion is, of course, # slide steps $>$ max. This prevents cycles.

For the following criteria, I define a distance measure between two positions P_1, P_2, $P = (R, T)$: $d(P_1, P_2) = (\omega(R_1 R_2^{-1}), |T_1 - T_2|)$. An alternative would be $d(P_1, P_2) = (|R_1 - R_2|_F, |T_1 - T_2|)$ [Zik98].[22] I define two "less-than" comparisons on pairs: $(d_1, d_2) <_2 (d'_1, d'_2) :\Leftrightarrow d_1 < d'_1 \wedge d_2 < d'_2$ and $(d_1, d_2) <_1 (d'_1, d'_2) :\Leftrightarrow d_1 < d'_1 \vee d_2 < d'_2$.

With our distance measure, we can define more stop criteria:

1. $d(O^{t+1}, O^t) <_2 \min$
   Basically, this checks whether the simulation has arrived at a dead end.

2. $d(O^{t+1}, O') >_2 d(O^t, O')$
   This checks whether the object is moving further away from the target position.

3. $d(O^{t+1}, O') >_1 d(O^t, O') \wedge d(O^{t+1}, O^t) <_1 \min$.
   Both $>_1$ and $<_1$ check different components.

[22] In both cases, unfortunately, the distance measure is *not* independent of the coordinate system in which an object has been modeled (this is because the order of transformation is: first rotation, then translation).

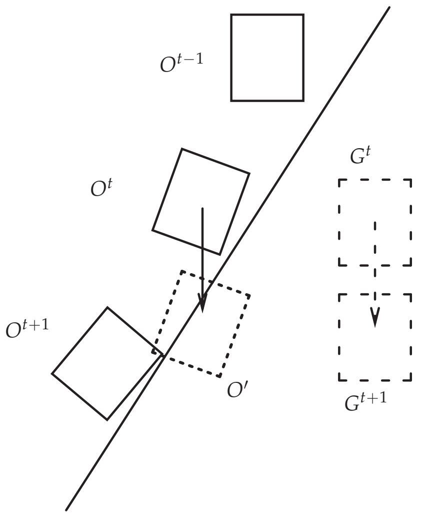


Figure 4.52: Several criteria determine when to stop moving the object with the "incremental motion" metaphor.

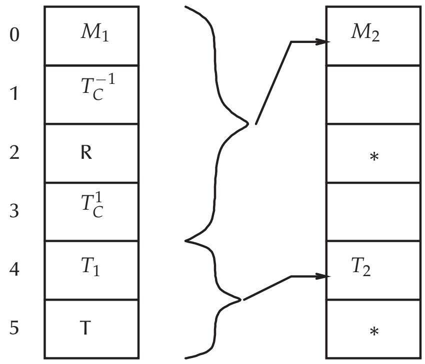


Figure 4.53: The set-up of transformations for the sliding simulation. The slots on the right marked with an arrow are the only ones that are set and changed during the approximation.

This condition means that, for instance, the ghost object just translated, but the simulation has only rotated the object.

Implementation

It is convenient to establish and maintain a set of transformations for the coll.-object, so that during approximation of the contact position the simulation loop only needs to overwrite the relevant transformations. After new velocities have been calculated, a new set has to be set up (see Figure 4.53).

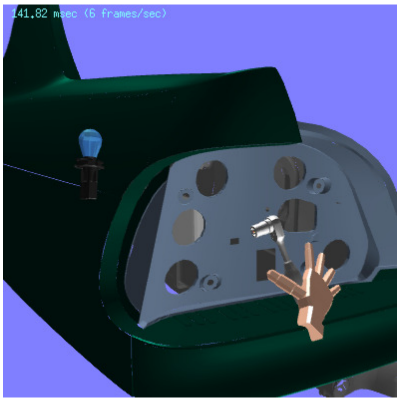
During contact position approximation, the algorithm moves the coll.-object back and forth, which will sometimes collide, sometimes not. At least the max. position at the beginning of the iteration is colliding. The algorithm has to save the position of the coll.-object whenever it collides during that iteration, because that one is needed for later classification of the contact position. More precisely, the polygons intersecting each other at the last colliding position are needed for the classification.

Sometimes, the algorithm finds 2 contact points very close to each other (relative to the distance from M). In that case, I currently unify those 2 points and treat the contact as a 1-point contact, because otherwise the direction of the rotation axis of the new rotational velocity is pretty random.

Timing

Experiments have shown that collision detection is still the bottleneck of the simulation. The test scenario and the timing is shown in Table 4.3.[23] Timing was done on a 194 MHz R10000. The DOP-tree algorithm of Section 3.5.9 was used for collision detection.

[23] You can watch the gliding simulation with some real-world objects by downloading a movie from `http://www.igd.fhg.de/~zach/coldet/index.html#movies`



num. obj.	num. pgons	max. # contacts	col.-det. time	sim. time
2	7800	1	2.2	0.27
4	11000	3	4.4	0.28

Table 4.3: Collision detection time is still by a factor 10 more time-consuming than simulation calculations (in this case). The numbers have been obtained by the scenario on the left. All times are in milliseconds, averaged over 3000 frames.

The bottom line is that collision detection still takes about 10 times as much time as the simulation.

Future work

Given only an approximation of the exact contact position, it is not trivial to classify the contact correctly under all circumstances. I would like to investigate further how this can be done. In particular, although face/face, vertex/vertex, parallel edge/edge, and parallel edge/polygon are extremely unlikely in practice (given the exact contact position), I believe they should be classified correctly in order to improve the robustness of the simulation (with unexact contact positions).

It might be necessary to consider the history (trajectory) of intersecting polygons in order to disambiguate some situations.

In order to reduce the number of collision detection tests, the penetration depth should be estimated. Unlike multibody dynamics [EJ97], it might be sufficient to compute just a part of the intersection polygon instead of computing the intersection volume (in general, the intersection polygon will consist of several disjoint parts, each of which is not necessarily simple nor closed).

Another problem are multiple contact points between the *same* pair of objects. In reality, several polygons intersect each other, all belonging to the same "smeared" contact point. Such a *contact region* might even be non-contiguous, I suspect. A pragmatic solution might be the following: consider all intersecting pairs of polygons as a distinct contact point; unify all those contact points "close" to each other (where "close" means just that the Euclidean distance is below some threshold).

Multiple contact points between the same pair of objects can be found only if the collision detection algorithm does *not* stop after the first pair of intersecting polygons has been found. This might slow down simulation considerably.

Perhaps, the problem of multiple contacts is the reason for the difficult situation depicted in Figure 4.54. So far, the algorithm does not seem to allow easy disassembly in situations where a worker in the real world would *force* the object, and turn and yank it eventually out.

The less exact the contact position is being approximated, the more likely it is that more than 2 contact points are found at that position. However, the

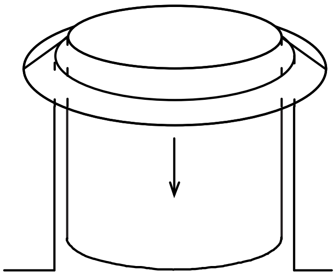


Figure 4.54: Situations where a real worker would "force" the part to come out, are still somewhat tedious in VR.

current implementation cannot deal with such situations (it handles them by falling back to the 2-point case). I would like to improve that, so that it can handle an arbitrary number of contact points.

I would like to experiment with other interaction metaphors, such as the "rubber band and spring" metaphor explained in the beginning of this section. This would probably allow the user to make the object translate by exercising enough torque on it. However, I am not sure yet if there is a canonical way how the user-specified delta and the simulation-specified delta should be "blended".

So far, I have ignored friction. As said before, I am concerned here only about suitability for virtual assembly simulation. It is not clear yet, whether users need friction in order to assemble more efficiently a part in crowded environments. At the time of this writing, the bottleneck of collision detection affects the simulation much more, so this should be improved first.

It might be interesting to explore a contrary approach: how would users like object behavior if friction was infitely large except in the pure translational case? Technically speaking, that would mean that during 1-point and 2-point contacts there would be only rotation (about the contact points). During 3-point contacts, there would be only translation ("sliding" parallel to the 3 contact points) or rotation (rotating about 2 out of 3 contact points).

Chapter 5

Applications

Technique always develops
from the primitive via the complicated
towards the simple.

ANTOINE DE SAINT EXUPÉRY

The Fraunhofer-IGD VR system "Virtual Design" incorporates all the concepts and algorithms presented so far. Many applications have been implemented with it, mainly for automotive companies [DFF$^+$96], architecture, landscaping, and historical projects.

Our VR system is being used by two types of customers: end-users customers and application-building customers. Most of the applications are virtual prototyping applications, but there are also some architectural and historical applications, and in the early years, several shows have been done with it. In the following, I will define and discuss virtual prototyping in more detail. Then, some of the applications will be described.

5.1 Virtual prototyping

Automotive industries seem to be among the leaders in applying virtual reality for real-world, non-trivial problems. After all, this is only natural, since they have been also among the first who applied computer graphics.

In the product development process, prototyping is an essential step. Prototypes represent important features of a product, which are to be investigated, evaluated, and improved. They are used to prove design alternatives, to do engineering analysis, manufacturing planning, support management decisions, and to get feedback on a new product from prospective customers (product clinic).

Markets are becoming more and more dynamic and quick-paced. In order to stay competitive, companies must deliver new products with higher quality and/or less costs in a shorter time. Additionally, they must provide customers with a broader variety of versions at minimum costs. Therefore, rapid prototyping and virtual prototyping (VP) are quickly becoming interesting tools for product development.

While some automotive companies have already begun to routinely use VR as a tool in styling and design reviews in the concept phase, it has not been clear that VR can be an efficient tool in assembly/disassembly simulations and maintenance verifications. Assembly simulations are much more difficult in that they involve a lot of interaction and real-time simulation. However, [BD83] revealed that the assembly process often drives the majority of the cost of a product. [Pra95, Ull92] point out that up to 70% of the total life cycle costs of a product are committed by decisions made in the early stages of design.

Although there are already several commercial 3D engineering tools for digital mock-up (and the number continues to grow), all of them lack one thing:

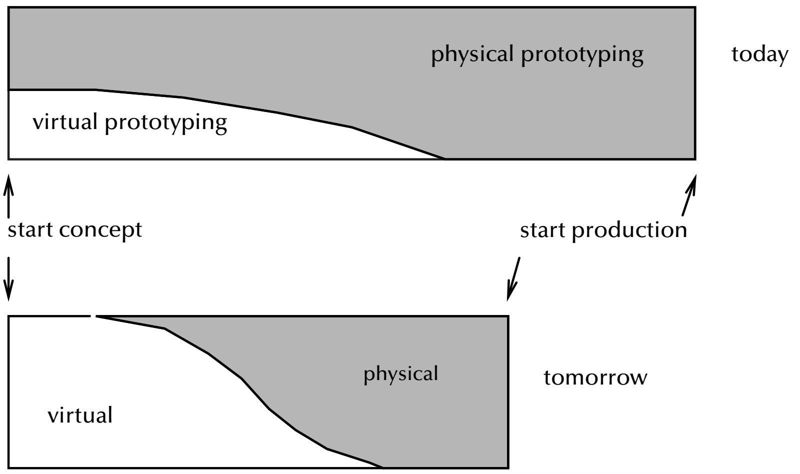


Figure 5.1: The goal of virtual prototyping is to reduce significantly the amount of hardware prototypes during conception, design, and evaluation of new products. The effect will be a reduction in time-to-market.

intuitive, direct manipulation of the digital mock-up by the human. Therefore, they are inherently inferior to VR.

In general, there is (hopefully) a twofoldbenefit of virtual prototyping:

1. shorter time-to-market, because the *design→evaluate→change* cycles are shorter[1] (see Figure 5.1);

2. cheaper products, because costs can be evaluated much earlier when designs are still in a rough stage.

Of course, crashing fewer real cars to meet federal safety regulations saves some money, too. A milled data check model of a complete car body costs about half a million DM [Dai98].

I am not sure if we will, one day, be able to do without any physical prototypes at all. But, this is the vision and the goal.

5.1.1 From rapid prototyping to virtual prototyping

Before and concurrently to the development of VR, other methods have been devised to speed up the design and prototyping of new products. One of these methods is *rapid prototyping* (RP) which has been in use for about 10 years. With RP real prototypes can be manufactured from CAD data.

As with VP, RP helps to increase quality, reduce costs, and reduce the time. Time savings can be up to 70%. According to a poll in the US, the main areas utilizing RP are transportation, aerospace, electronics, automotive, and machine manufacturing companies. But there are other more exotic areas where RP can make sense, such as surgery planning or archaeology.

There are several techniques for creating rapid prototypes, but all of them have in common that they are made out of layers which are added one at a time.[2] The techniques can best be distinguished by the material which they

[1] Building a physical prototype takes 3–13 weeks [Dai98].
[2] Most other manufacturing techniques are subtracting or deforming material.

use. Some use fluid base material, some use powder, some solid or molten material.

Although RP has been a great improvement over manual prototyping, it is still in some cases too limited with respect to the number of different variants, the size of prototypes, or the functionality. It is still not possible to produce some ten variants of the body of a car in 1:1 scale in one piece, because it is too large and it would take too long.

The most important shortcoming of RP is that an interactive "what-if" approach to the design/evaluation iteration is just not possible. The designer cannot interact with the rapid prototype and change some feature or property "on-line".

5.1.2 Definitions of virtual prototyping

There seem to be two different understandings of exactly what VP is: the "computer graphics" and the "mechanical engineering" point of view.[3]

The *computer graphics definition* of virtual prototyping (VP_{CG}) is the application of virtual reality for prototyping physical mock-ups (PMUs). The VR system simulates and renders all characteristics relevant to the particular context as precise and realistic as possible in an immersive environment. The criteria of a VR system must be met by such a VP system (see Chapter 1).

In the *mechanical engineering definition* of virtual prototyping (VP_{ME}), the idea is to replace physical mock-ups by software prototypes. This also includes all kinds of geometrical and functional simulations, whether or not involving humans. For instance, simulation of assembly lines, FEM crash tests, CFD simulations of air flow in the interior, etc., are VP_{ME} activities, too.

Digital mock-up (DMU) is a realistic computer representation of a product with the capability of performing all required functionalities from design/-engineering, manufacturing, product service, up to maintenance and product recycling [DR96]. In a sense, DMU can be viewed as the medium through which stylists, designers, testers, manufacturers, marketing people, customer supporters, etc., exchange information about a new product [ZPR+98].

So, immersive virtual prototyping is but one technique for implementing the DMU strategy:

$$VP_{CG} \subset VP_{ME} \subset \text{DMU}$$

5.1.3 The right display

Many applications of virtual prototyping in the automotive business are concerned with human tasks, visualization, or just viewing, in a very local, spatially restricted area (namely the car, or a part of it). For all of these applications, stereo viewing is essential because it enables the human brain to assess depth in the range up to about 2–3 meters away from the eye. Within that range, stereoscopic viewing is the second-most important depth cue. In fact, it has been shown that human performance in assembly tasks is significantly impaired by monoscopic viewing [Ros93]. However, another study has shown that when using HMDs, users tend to underestimate distances and lengths consistently [HF93]. Presumably, this is due to lens distortions (Booms probably

[3] Actually, the term "virtual prototyping" is also used in other areas such as electronics and VLSI chip design. In one example, HP has been able to cut the development time of PC motherboards from 9 to 2 months [Sch98].

have a similar shortcoming). It is particularly noticeable with wide field-of-view HMDs. On the other hand, there is some evidence that for certain types of applications immersion improves user performance by about 40% [PPW97].[4]

To my knowledge, there are no HMDs or Booms with a wide field-of-view and negligible distortion. There are rendering techniques to remedy this, namely multi-pass rendering with a "cushion"-like textured polygon so as to compensate for the distortion. In my opinion, though, they are not acceptable for virtual prototyping applications because all the graphics power available must be spent on achieving high polygon numbers and other multi-pass rendering effects such as real-time shadows.[5]

Other display devices like cave and projection screen have much less distortion.[6] For styling reviews they are probably the display device of choice, and maybe even for scientific visualization tasks. But for tasks where users have to move their view point (i.e., their head) a lot, I believe the HMD (or maybe a Boom) must be used.

5.1.4 Other VP applications

Some of the applications mentioned here are in the VP area according to the computer graphics definition, some are in the VP area according to the mechanical engineering definition, and some are even in the much broader DMU area.

A lot of development for utilizing VR for VP is being realized by automotive and aerospace companies. Many efforts, however, are still feasibility studies.

Practically all automotive companies investigate the use of VR for styling reviews and other mere walk-through applications. Some of them already employ it for daily work. Usually, the model is rendered on a large-screen stereo projection or in a cave. Variations can be compared on the fly with realistic colors and illumination effects [FE97]. At Daimler-Benz, the body of a car can be reviewed in an immersive virtual environment by the aid of zebra lighting [Buc98].

Since VR provides an intuitive and immersive human-computer interface, it is perfectly suited to do ergonomics studies. Consequently, many projects capitalize on this advantage of VR. Ford employs virtual prototypes with several proposed dashboard configurations to verify visibility and reachability of instruments.

Researchers at Caterpillar Inc. use VR to improve the design process for heavy equipment. Their system [LD97] allows them to quickly prototype wheel loader and backhoe loader designs to perform visibility assessments of the new design in a collaborate virtual environment. Further the engineers can simulate the operation of the equipment and evaluate visual obstructions.

Volkswagen has incorporated some useful applications in the vehicle development process. They have coupled a commercial human inverse kinematic package with VR to investigate different ergonomic features. They also visualize the results of FEA crash computations in VR interactively. The virtual

[4] They also found that practising a task at the desktop can have a *negative* effect on user performance when doing the same task in VR, while practising a task *first* in VR can have a *positive* effect on desktop performance.

[5] One has to keep in mind, that 2 million polygons must be spent at least on the car body and the car interior before a stylist will look at it.

[6] The beamer electronics compensates for the lens distortions.

product clinic avoids faulty developments and helps assess customers' wishes [PRS+98].

Chrysler launched a project to study the process of virtual prototyping, to investigate the steps required for the creation of a virtual representation from CAD models, and for the subsequent use of the prototype in immersive VR [FE97].

The Boeing 777 was the first plane to be designed entirely on computers. Boeing claims they have not built a single physical prototype before assembling the first plane. According to a key note speech at ASME'97, they have not saved time, though, compared to conventional prototyping. However, by using VP they claim to have designed a better product, i.e., a plane which needs less fuel for instance.

A prototype virtual assembly system is described in [JCL97]. However, compared to the application described in this section it is less powerful and mature. In addition, they do not describe the process of authoring a virtual environment. It seems to me that they pursue the toolbox approach, i.e., the system is a monolithic program on top of a set of libraries, while my approach is the scripting approach.

Systems for designing in VR are presented by [CDG97, ZHH96]. My approach is to use VR only for investigation and simulation. No geometry can be designed by our system, because I do not feel that this would take advantage of the strengths of VR. A factory simulation for immersive investigation has been presented by [KV98]. Although no direct manipulation with objects is possible, problems can be identified earlier than through the aid of charts and graphs produced by conventional simulations.

At Rutherford Appleton Laboratory, based on dVS, applications have been implemented for immersive investigation of CFD data and assembly simulation of parts of a particle collider [LB98]. Special about the CFD data visualization is the possibility to "repair" the grid interactively while being immersed.

A system featuring functionality similar to ours has been developed by [JJWT99]. It does not get quite clear as to how a virtual environment is actually being authored (see Section 2.2) in their system. They have performed a user survey, too; unfortunately, it does not get quite clear what kind of tasks these users perform during their daily work.

5.1.5 The virtual seating buck

In 1995 we developed a "Virtual Seating-Buck" for BMW. This project focused on the challenge of using VR to create a tighter integration between design and engineering analysis functions in the development process of automotive interiors (Figure 5.2).

It was necessary to address graphic display quality as well as functionality, and interaction techniques in order to provide the user with a convincing feeling of immersion into the virtual environment. To further increase this effect, a physical mock-up consisting of seat, steering wheel and foot pedals was built (Figure 5.3). Other hardware components included a tracking system, data glove and Boom.

One important aspect in order to intensify the user's feeling of immersion was the precise registration between real objects and virtual objects. This was achieved by calibrating the virtual steering wheel with its physical counterpart (held by the user) and implementing a virtual feedback, such that the virtual steering wheel rotates simultaneously with the physical one. Correction of dy-

Figure 5.2: The virtual seating-buck scenario. Data courtesy of BMW AG.

Figure 5.3: The physical equipment of the BMW virtual seating-buck.

Figure 5.4: Visualization of air flow in a car's interior.

namic tracking errors was not done, because the Boom changed the distortion when moved (see Section 4.3.2).

Another point of interest was the embedding of CAE simulation results into the virtual environment. This was demonstrated by visualizing the air flow in the passenger compartment caused by the air conditioner. Particles can be traced interactively by the user. (see Figure 5.4). Finally we addressed the use of VR for maintenance access verification and configuration studies. Using real-time collision detection on a large scale, conditions such as system location, space allocation and stay-out envelopes could be interactively evaluated taking the air condition/heating unit as an example.

The system has been in prototypical use at BMW's R&D facility since November 1995 running on an SGI RE2 computer with two independent graphics subsystems. The results as well as user responses are promising.

5.1.6 Exchanging an alternator

A vision of virtual prototyping was developed within the ESPRIT project AIT (Advanced Information Technologies in Design and Manufacturing; Project partners were many European automotive and aerospace companies, IT suppliers, and academia) [DR96, DFF$^+$96, DR86].

One of the key features was real-time collision detection for clash and clearance checks. Others are volume tracing and constraints (to model the hood of the car). Flexible objects (like a hose) were implemented by an application-specific module.

This was one of the first attempts to simulate (among other things) a complete service maintenance of a car's alternator by a digital mock-up: the user wearing a head-mounted display and data-glove interacts with a scene of about 40,000 polygons representing the front of the engine compartment, which is rendered at about 20 frames/sec. He has to open the hood of the car first. Then he has to accomplish the following steps in order:

1. remove the fan,
2. tilt the oil filter,
3. push the cooling hose to the side,
4. unscrew the fixing wheel of the V-belt,
5. grab the alternator and take it out.

Although this is still a rather simplified scenario of a real maintenance operation, the VR system has to provide quite a few functionalities for object manip-

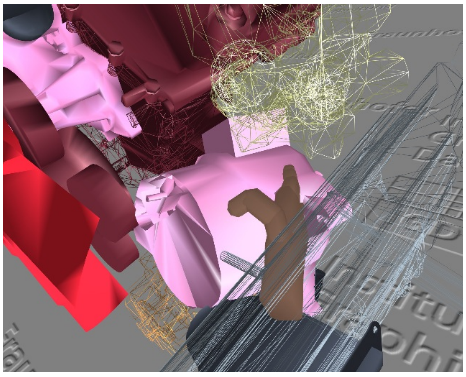


Figure 5.5: During an interactive fitting simulation in a virtual environment, the system highlights all objects colliding with the alternator (data courtesy AIT consortium).

ulation and object behavior. Each step and each functionality including the car hood involves collision detection.

Variants of parts can be tried and fitted interactively in place of the original ones. Figure 5.5 shows an example: all objects colliding with the new part will be highlighted on-line by switching their rendering to wireframe.

Volume tracing is needed in order to determine the space needed for a maintenance operation. The "sweep" action traces out the volume when the user moves an object by replicating (simplified) copies of it. This was used to check serviceability.

5.2 Assembly simulation

Assembly/disassembly verification has several goals. The final goal, of course, is the *assertion* that a part or component can be assembled by a human worker, and that it can be disassembled later-on for service and maintenance. However, other questions need to be addressed, too: is it "difficult" or "easy" to assemble/disassemble a part? How long does it take? How stressful is it in terms of ergonomics? Is there enough room for tools? Could it be done by a lay person?

On behalf of BMW, I have developed an application for assembly/disassembly simulation on top of the frameworks and algorithms presented so far [Zac98a]. In this section, I will present two scenarios which have been chosen as examples for the development of an assembly simulation application, then I will describe the functionality needed for assembly investigations.

5.2.1 Scenarios

Together with our customers, we have chosen two scenarios in order to assess a first set of functionalities needed for assembly tasks in VR; one of them is a simple one, the other is one of the most difficult.

The tail-light

The first scenario is the disassembly of the tail-light of the BMW 5 series (Figure 5.6). First, the covering in the car trunk must be turned down, in order to

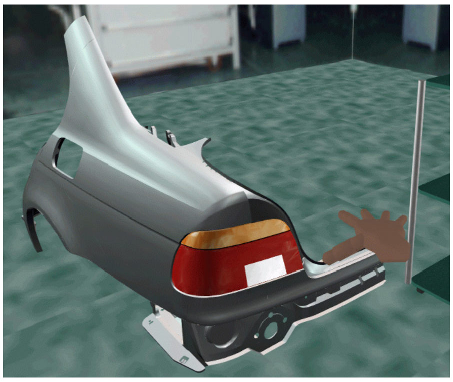


Figure 5.6: Overview of the tail-light scenario. The tail-light is to be removed.

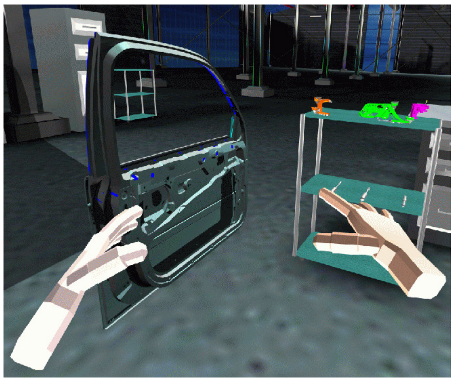


Figure 5.7: The door scenario. Two hands and several tools are necessary to perform the assembly.

get access to the fastening of the lights (Figure 5.12). To reach the screws fixing the tail-light, the fastening needs to be pulled out.

Then the tail-light itself can be unscrewed by a standard tool. After all screws are taken out, the tail-light cap can be disassembled by pulling it out from the outside.

The door

This scenario is much more complex and more difficult in that both hands and various tools must be utilized (Figure 5.7).

The first task is to put the lock in its place in the door. This is quite difficult in the real world, because it is very cramped inside the door and the lock cannot be seen very well during assembly. Screws have to be fastened while the lock is held in its place (Figure 5.8).

Next, the window-regulator is to be installed (Figure 5.9). This task needs both hands, because the window-regulator consists of two parts connected to each other by flexible wires. After placing the bottom fixtures into slots, they must be turned upright, then the regulator screws can be fixed.

Finally, several wires must be laid out on the inner metal sheet, clipped into place, and connected to various parts. However, this part of the assembly was not performed in VR.

5.2.2 Interaction Functionality

In this section, I will describe an array of techniques most of which have proven to be helpful in verification of assembly simulations. They enable inexperienced users to work with virtual prototypes in an immersive environment and help them experiment efficiently with CAD data.

Multi-modal interaction. It is important to create an efficient human-computer interface, because the tasks to be performed in virtual prototyping can be quite complex. Many of the “classic” VR interaction techniques can be utilized, such as (static) gesture recognition, 3D menus, selection by beam casting, etc. Each technique must be implemented in a very robust and user-independent manner, otherwise users will become irritated and disapproving of VR.

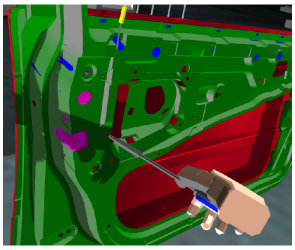


Figure 5.8: Tools snap onto screws and are constrained. Also, they are placed automatically at an ergonomic position within the hand by the system.

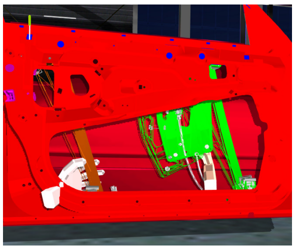


Figure 5.9: The window regulator has to be installed with two hands; the "ghost" paradigm signals collisions.

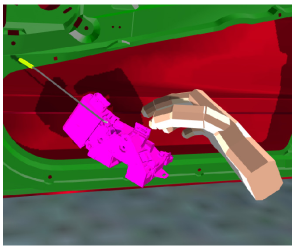


Figure 5.10: The object-on-the-lead paradigm allows to verify assembly. The object is not linked rigidly to the hand.

While grasping two objects with both hands, a user must still be able to give commands to the computer. This can be achieved most intuitively by voice recognition. Also, multi-sensory feedback (see below) plays an important role in multi-modal interaction.

An on-line service manual. I believe that VR could eventually become an efficient means for training service personnel and creating an interactive service manual. Service manuals could be disseminated in the form of VRML environments, which can be viewed and interacted with on a PC-based "fish-tank" VR system. However, augmented reality-based systems might be necessary, especially in larger and more complex vehicles, such as aircrafts and submarines.

In our environments we have implemented an interactive service manual as well as an interactive training session. First, a trainee *learns by watching* the service manual; this is basically an animation of the assembly process. While the animation is being played back, the trainee can move freely about the environment and watch from any viewpoint.

When the trainee is ready to *learn by doing*, he will perform the task step by step. After each step is completed the system will point him to the part or tool he will need for the next step and tell him what to do with it. For instance, after all screws for the door lock have been fastened, the system highlights the window regulator (by blinking) and instructs him how to assemble it. The instructions have been pre-recorded and are played back as sound files.

So far, the virtual service manual and the interactive training session are hand-crafted via manual scripting. However, it should be straight-forward to extract them from a PDM system, *if* the process data are there in a standardized form.

Getting help from the system. When the number of functions becomes large in the VR system, occasional users cannot remember some commands (in our current system there are about 40 functions with a total of 170 speech commands). Similar to 2D applications, we additionally provide hierarchical menus. I have experimented with several forms of 3D menus, but 2D menus seem to be best in terms of usability and legibility (see Section 4.5.1). In my experience, 3D menus are to be considered only as an auxiliary interaction technique, because it is more difficult to select menu entries in VR than it is in 2D.

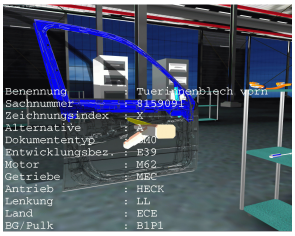


Figure 5.11: Administrative data stored in the PDM about parts can be displayed during the VR session.

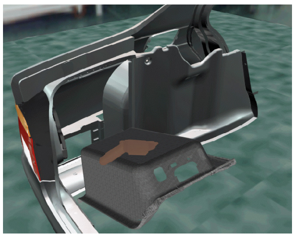


Figure 5.12: Inverse kinematics is needed for "door-like" behavior of parts.

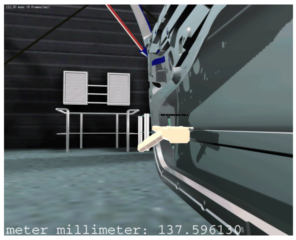


Figure 5.13: With the virtual yard-stick distances can be measured in the VE.

Investigation tools. In order to make the correct decisions, it is important that the user can get information about the parts involved in the virtual prototype currently being investigated. Administrative information about parts can be displayed in a heads-up fashion by pointing at objects with a ray (see Figure 5.11). Of course, any other selection paradigm can be used as well.

A tool which has been requested by designers is the *clipping plane*. It can help to inspect "problem areas" more closely. When activated, the user "wears" a plane on his hand; all geometry in front of that plane will be clipped away in real-time. Optionally, the part clipped away can be rendered transparently. The plane can be released from the hand and grabbed again, so that the user can move freely while the clipping plane remains motionless. Sometimes it can be necessary to restrict the motion of the plane so that it is always perpendicular to one of the world coordinate axes. By utilizing an OpenGL feature clipping can be done at interactive frame rates with a geometry of about 60,000 polygons.

Another tool to inspect assembly situations and the mechanical design is the *user size*. This parameter can be controlled by simple speech commands, which in turn affect all parameters by which a virtual human is represented, in particular navigation speed and scale of position tracking. This way, a user can comfortably "stick his head" inside some narrow space.

In order to measure distances we have implemented two options: A user can select two objects, then the system will compute the *minimal distance* between the two and display it in the heads-up display. Or, the user can grab a *virtual yard stick* (see Figure 5.13). While grabbed, the yardstick adjusts its length in both directions so that it just touches the closest geometry. Additionally, its length is shown on the heads-up display. Another way would be to select two points on the geometry and have the system display the length of that line.

Physically-based simulation. Many mechanical components have some articulated parts. These could be simple "door-like" mechanisms (see Figure 5.12), i.e., permanent joints with one rotational degree of freedom (DOF), such as hoods, lids, etc. Other very simple ones are sliding mechanisms (one translational DOF), for example the seat of a car. Inverse kinematics of these and other articulated chains can be simulated on-line.

For complicated kinematic simulations, such as the working conditions of a complete chassis, we have pursued a different approach: the VR system loads the results of an off-line simulation by a commercial package, such as Adams™. The user can then interactively steer the visualization, for example by turning the steering wheel or by speech commands.

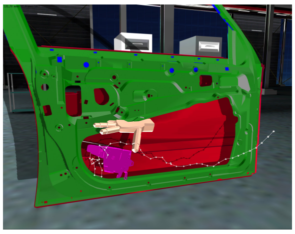


Figure 5.14: During assembly, the path of any part can be recorded, edited, and stored in the PDM system.

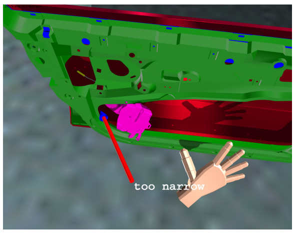


Figure 5.15: Annotations can be put into the scene by voice commands.

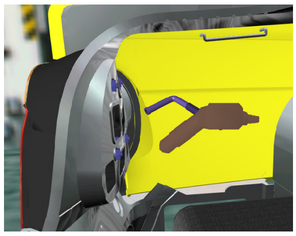


Figure 5.16: Violations of safety-distance are highlighted by yellow, collisions are red.

A lot of the parts in a vehicle are flexible: wires, hoses, plastic tanks, etc. It is still a major challenge to simulate all these different types of flexible parts with reasonable precision and at interactive rates. In particular, simulation of the interaction of flexible objects with the surrounding environment and the user's hands by a general framework is, to my knowledge, still unsolved.

We have implemented hoses and wires in our VR system; the wires or hoses are attached at both ends to other, non-flexible parts, and they can be pushed or pulled by a user's hand.

Verification without force-feedback. In my experience (substantiated by a user survey [Zac98a]), assembly tasks are more difficult in VR than in the real world, because in VR there is no force and haptic feedback. Humans can even perform quite complicated tasks without seeing their hands or tools merely based on auditory, haptic and kinesthetic feedback. Therefore, I have provided a lot of interaction aids trying to compensate for the missing force feedback.

In order to help the user place parts, I have developed two kinds of *snapping* paradigms: the first one makes objects snap in place when they are released by the user and when they are sufficiently close to their final position. The second snapping paradigm makes tools snap onto screws when sufficiently close and while they are being utilized (see Figure 5.8). The second paradigm is implemented by a 1-DOF rotational constraint which can be triggered by events.

The major problem is: how can we verify that a part can be assembled by a human worker? A simple solution is to turn a part being grasped into what I call a *ghost* when it collides with other parts: the solid part itself stays at the last valid, i.e., collision-free, position while the object attached to the user's hand turns wireframe (see Figure 5.9).

However, invalid positions can be "tunneled". Therefore, I have developed the *object-on-the-lead* paradigm: the object is no longer attached rigidly to the virtual hand; instead, it "follows" the hand as far as it can go without penetrating any other parts (see Figure 5.10). I have implemented a physically-based simulation (see Section 4.5.4), so that the object can slide along other parts; in my earlier implementation, there was no sliding, which caused the object-on-the-lead to get stuck in congested environments. So, at any time it can assume only valid positions. Of course, exact and fast collision detection is a prerequisite [Zac98b].

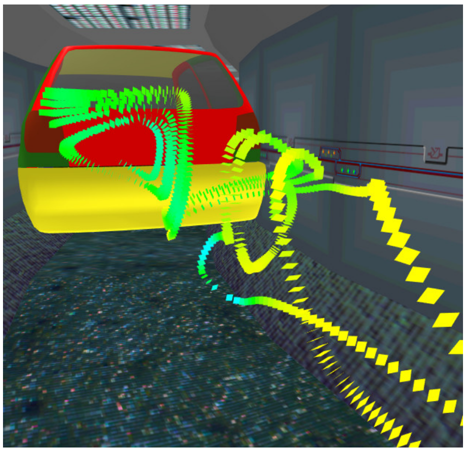


Figure 5.17: Immersive CFD investigation: color-coded particles indicate the flow and pressure around a VW Polo. Data courtesy of Volkswagen AG.

This is only a first step. A completely reliable verification will check the virtual hand for collisions as well. Also, the hand and/or part should slide along smooth rigid objects to make assembly easier for the user.

Feedback to the user. Any VR system should be as responsive as possible, especially for occasional, non-expert users. The users targeted for immersive VP will probably not use VR every day. Therefore, multi-sensory feedback is important to make them feel comfortable and in control.

Therefore, the system acknowledges all commands, in particular those invoked via voice recognition. Currently, this is done by pre-recorded audio or speech. Eventually, we will utilize speech synthesis.

During the assembly simulation, a variety of feedbacks can be combined which will be given if the user tries to move an object at an invalid position: acoustic feedback, tactile feedback by a Cybertouch^{TM} glove, and visual feedback. Visual feedback comes in several flavors: whole parts can be highlighted (see Figure 5.16), or the polygons which would have intersected at the invalid position can be highlighted.

Documentation. If a certain assembly task cannot be done, then the result of the verification session should be a precise as well as intuitive understanding why that is. A number of techniques have been implemented in order to investigate and document a possible failure of assembly.

During assembly/disassembly the path of any part can be recorded and edited in VR (see Figure 5.14). Saved paths can then be stored in the PDM system.

While parts are being moved, the sweeping envelope can be traced out. It does not matter whether the part is moved interactively by the user or on an assembly path.

Problems can be annotated by placing 3D markers (we have chosen 3D arrows). Then, verbal annotations can be recorded and displayed textually next to the marker (see Figure 5.15). Note that all interaction is done by the user via speech recognition, except for placing the marker. Eventually, the markers and annotations can be exported and stored with the parts in the PDM system.

5.3 Immersive Investigation of CFD Data

I have helped integrate a flow-visualization module with our VR system [DFF+96]. The interactive analysis of the flow around a Volkswagen Polo was demonstrated at the Hannover Messe '95 (see Figure 5.17). A different and more advanced visualization was shown at IAA '95 (see below).

Our particle tracing module allows the release of particles at sources which are either coupled to the user's hand or which can be positioned freely in the 3D scene by the user. The latter alternative allows to position a fixed particle source and walk around in the scene to watch the traces from different points of view. The system can produce consecutive particles, streamlines, and streaklines. All forms can be color-coded according to a scalar.

5.4 Shows

An early application of our integrated VR system was shown at the "Industry Exhibition Hannover" in Spring 1995 (see above). At the show, we tried to point out some possible applications of VR in the automotive industry.

For Volkswagen, we built the inside of a diesel engine as a virtual environment, which was shown at the IAA auto show in Frankfurt, Germany, in August 1995, and at the Detroit Auto Show in 1996.

A user could fly inside the combustion chamber and watch several combustion cycles while interacting with the air flow field. One of the key features there was the visualization of time-variant flow-fields in the swirl port and in the combustion chamber. We animate the piston as well as the valves during the flow visualization. The movement of diesel particles is visualized, too. Isosurfaces of the temperature serve as an indicator of the momentary location of the flames when the air/diesel mixture is burning.

All data, geometry, animations, and flow fields have been imported from simulation packages. This is an application where the importance of a concept of time becomes evident: all animations (such as piston head, valves, temperature color, and diesel particles positions) must coincide with the visualization, even though most of them are not specified on the complete cycle. Furthermore, global simulation time must be set sometimes to a certain value, or the "speed" of simulation time must be slowed down or stopped altogether.

Chapter 6

Epilogue

The Road goes ever on and on
Out from the door where it began.
Now far ahead the Road has gone,
let others follow it who can!

TOLKIEN, The Lord of the Rings,
p. 1024

In this chapter I will summarize the main contributions of this thesis and I will venture to describe avenues for future work in the area of virtual assembly simulation and virtual reality in general.

6.1 Summary

In 1995, only a few VR systems were commercially available and a few more in the academic domain. None of these was mature at the time, nor had any of them been deployed in the field for everyday work. Some commercial and most academic systems were not so much a self-contained VR system, but rather a set of libraries which application programmers could build upon.

In particular, VR was not ready for use for industry application. Problems persisted in the following areas (among others): electro-magnetic tracking, high-level specification of virtual environments, efficient interaction metaphors and frameworks, and real-time collision detection and response.

This thesis has made contributions, to all of these areas. Almost all of the algorithms, applications, and frameworks presented in this thesis have been integrated into the VR system *Virtual Design II*, which has been developed by the department I am with, and which is now commercially available through the spin-off VRCom[1].

Framework for authoring virtual environments

Creating virtual environments involves two steps: first, it has to be described, then it must be simulated. The former is known as *authoring*, while the latter basically means that the description is executed by a VR system.

After a discussion of frameworks for computer-human interaction in related areas (see Section 2.1), such as user-interface management systems, Smalltalk, and programming languages, Section 2.2 proposes a framework for authoring virtual environments. This framework is the basis of our VR system *Virtual Design II*.

The main premise for the proposed framework is that it should be easy for non-computer scientists (like architects or mechanical engineers) to author virtual environments. This excluded full-fledged programming languages or state

[1] `www.vrcom.de`

machines. Therefore, this thesis introduces the concept of *actions*, *events*, *inputs*, and *objects* (AEIO). These entities can be combined to virtual environments by the *event-based* approach; the behavior-based approach was deemed inappropriate for application to virtual prototyping.

In each class of entities, a set of entities has been identified such that a significant part of an application can be implemented using these. Each set has been chosen such that its expressive power is maximized while its number of entities is minimized.

Application-specific capabilities can be plugged in at run-time, thereby augmenting the set of actions and inputs. In order to facilitate short turn-around times during development of a plug-in, the following development cycle is supported: the system can delete all actions and/or inputs instantiated from a plug-in; the programmer changes the implementation of the plug-in; the system re-loads that plug-in and re-instantiates all former instances.

Based on the AEIO framework, a *scripting language* has been designed such that its concepts can be easily grasped by non-programmers. It is orthogonal in the sense that any input can trigger any action. In addition, a many-to-many mapping between inputs and actions can be established; actions can be combined to more complex actions; similarly, inputs can be combined by logical operators. Simple constructs allow the VE author to combine actions or inputs into more complex features. The above mentioned events are basically *filters* which can detect transitions in inputs (or other events).

Besides the conventional scene graph several *alternative* object graphs can be maintained. Due to the dynamic nature of a VR scene graph, references in that graph are symbolic, allowing for dynamic changes "behind the scene". Alternative scene graphs are also used to map *semantic attributes* not present in the conventional scene graph.

Three layers of creating virtual environments can be identified (see Section 2.4.2). The scene graph is on the lower level, while the AEIO framework is on the middle level. On the top level are specialized authoring tools for specific application domains (such as assembly simulation), which are better described as configuration tools. On that level, the author merely specifies "roles" of objects, such as "tool". Using high-level authoring tools, creation of VEs (in that specific application domain) is a matter of minutes.

Collision detection

Collision detection is one of the enabling "technologies" for all kinds of physically-based simulation in VR and other applications. The algorithms presented in this section assume a polygonal object representation.

After a brief discussion of the requirements of collision detection algorithms (see Section 3.1.2, one of the basic operations of collision detection is analyzed (see Section 3.2). The comparison of several algorithms yielded a simple rule for choosing the optimal algorithm. A collision detection algorithm wishing to maximize performance should implement at least two algorithms for that basic operation and choose either of them on a case-to-case basis according to this rule.

Based on that, I developed an algorithm for checking any type of object, even deforming ones (see Section 3.3). The basic idea is a pipeline of bounding box checks in carefully chosen coordinate frames combined with sorting of bounding boxes. By sorting, one can find quickly a range of polygons within a certain region. The sorted list of polygons can be updated quickly between successive frames, because usually deformation between frames is small. This algorithm can check two spheres with 10,000 polygons each within about 4.5 milliseconds on average and two door locks with 12,000 polygons each within about 15 milliseconds.

Convex objects present themselves for incremental collision detection. Based on the concept of linear separability and simulated annealing, I developed a very fast algorithm which depends linearly on polygon count and rotational velocity with very small hidden constants (see Section 3.4.3). The algorithm gains additional efficiency by hill-climbing on the convex hull and by maintaining the separating plane in two different coordinate frames simultaneously. Collision detection time is of the order of microseconds when objects do not intersect. A comparison with the renowned Lin-Canny algorithm showed that the algorithm presented in this thesis is about 2 times faster. In addition, it seems to be more robust numerically.

Actually, the current implementation of this algorithm is a probabilistic Monte-Carlo algorithm. Although convex objects are not encountered in our scenarios and applications, such algorithms can still be valuable earlier in the collision detection pipeline (see Section 3.9).

Said Lin-Canny algorithm is based on a closest-feature criterion and makes use of Voronoi regions. I have given a much simpler criterion which does not need Voronoi regions (see Section 3.4.4). So, objects could even deform, provided they stay convex. This algorithm has been implemented on Cosmo/Optimizer. However, it is, as of this writing, not clear yet how it compares to the original one.

Another class are hierarchical algorithms, which are well-suited for non-deformable objects with large polygon counts. In this class, three algorithms have been presented.

The first one is based on clipping and bisecting non-axis-aligned boxes (I have called the associated data structure a box-tree). The algorithm takes advantage of the special geometry of boxes and certain assertions about the arrangement of boxes within a box-tree. In addition, recursion makes use of all the calculations done on the previous level.

The second algorithm uses a data structure very similar to the first one. It efficiently encloses non-aligned boxes by axis-aligned ones (see Section 3.5.6). Again, recursion makes complete use of the calculations on the previous level. This algorithm is much faster than the first one.

A "good" bounding volume hierarchy is crucial for efficient hierarchical collision detection (see Section 3.5.7). Yet, the construction of such a hierarchy should be fast enough so that it can be done at load time. The algorithm for constructing box-trees developed in this thesis is, under certain assumptions, in $O(n)$ time, which has been supported by experiments. By evaluating many different criteria guiding the construction process, the optimal criterion (in the sense of fast collision detection) among them was determined (see Section 3.5.7).

The third algorithm for hierarchical collision detection makes use of discrete oriented polytopes (DOPs) (see Section 3.5.8). For this type of bounding volume, I have found an elegant way to enclose non-axis-aligned DOPs by axis-aligned ones which is in $O(k)$ while the previously proposed method is in $O(k^2)$ (k being the number of orientations). The optimal number of orientations (in the sense of fast collision detection) seems to be 24; it has been determined by extensive experiments (see Section 3.5.9). This algorithm can check a pair of door locks, 50,000 polygons each, within 4 milliseconds; two car bodies, each 60,000 polygons, can be checked within $\frac{1}{2}$ millisecond.

The collision detection algorithm based on DOP-trees can be generalized to allow different numbers of orientations and even different sets of orientations. As for the box-tree, an efficient algorithm for constructing DOP-trees was developed. It works with any number of orientations.

The box-tree and the DOP-tree algorithms were compared with implementations of two other hierarchical algorithms, namely Rapid and Quick_CD (see Section 3.5.10). Experiments have shown that the DOP-tree algorithm is about as fast as Rapid (in certain cases slower while faster in other), which is, to my knowledge, the fastest hierarchical algorithm to date.

Because incremental algorithms perform so well for convex objects, incremental hierarchical collision detection is discussed. An algorithm based on DOP-trees has been implemented (see Section 3.5.11), and its performance is evaluated in detail.

In the class of flexible objects two algorithms were implemented (see Sections 3.7.1 and 3.7.2). The first one is a hierarchical algorithm, based on the assumption that deformation is small between successive frames. During traversal, the algorithm takes the maximum "drift" of polygons into account. The second one is based on maintaining a sorted list of polygons. Overlapping polygon bounding boxes are found by three sweeps, each along one axis. Unfortunately, both of them perform worse than the bounding box pipeline algorithm in all practical cases (see Section 3.3).

So far, all collision detection algorithms dealt with the polygon level. If a virtual environment consists of many moving objects, then the n^2-problem is encountered at the object level, too.

At the object level (see Section 3.8), octrees (Section 3.8.4) and grids (Section 3.8.5) were discussed. Both have been implemented such that only those cells are being visited which actually need to be updated. According to experiments involving various complexities, grids are better suited for highly dynamic environments (see Section 3.8.6), which is in contrast to more static applications, such as ray-tracing. My experiments suggest that for more than 30–40 objects both a grid and the separating planes algorithm should be used for neighbor-finding, because grids are in $O(n)$, while convex hulls are tight bounding volumes. For environments with less than 30 objects, the separating planes algorithm should be used. However, these numbers seem to depend also on the number of polygons, to a some extent.

Finally, I have proposed and implemented a collision detection pipeline (see Section 3.9). This pipeline consists of several stages, each of which can be considered a certain type of filter. The pipeline has been integrated as a module in the VR-System Virtual Design II.

Collision detection lends itself well to parallelization. The collision detection module features concurrency, coarse-grain, and fine-grain parallelization (see Section 3.10). Experiments demonstrate the efficiency of the implementation.

Interaction

Interaction in virtual environments comprises many different aspects: device handling, processing input data, navigation, interaction paradigms, and physically-based object behavior.

Navigation is the most basic interaction in virtual environments. A general framework has been proposed allowing for a broad range of navigation paradigms and input devices (see Section 4.4). Other basic tasks include menus (Section 4.5.1), which are needed to organize the multitude of functions, posture recognition (Section 4.2.1), which is needed to invoke frequently used functions, and utterance recognition (Section 4.2.2). All of these have been integrated in the VR system.

Electro-magnetic tracking poses at least two problems: noise and distortion. In real-world sites, electro-magnetic noise is produced by all kinds of electrical devices, and distortion is caused by ferro-magnetic material in floors, ceiling, speakers, etc. Both noise and distortion can render a VR system useless, especially for virtual assembly simulation, because they would compromise precise positioning. In addition, distortion can lead to artifacts when the environment is rendered on a cave or powerwall.

The problem of noise is addressed by a filtering pipeline, designed to meet the special requirements of VR (see Section 4.3.1). Tracker distortion is greatly reduced by an algorithm developed in Section 4.3.2. Extensive tests verify its quality and performance. The algorithm has been integrated in several device servers of Virtual Design II.

Another basic interaction task is grasping. In fact, with virtual assembly simulation the user's hand is the most important tool. Traditionally, grasping has been implemented rather unnatural through posture recognition. One of the reasons is that there is (as of yet) no force feedback to the user's real hand. I have developed an algorithm for natural grasping, which presses the virtual hand's fingers to an object and grasps it based on the analysis of the contact (see Section 4.5.3).

When trying to assemble an object, it should not penetrate other parts. Instead it should behave similar to objects in the real world and somehow "slide" along the surface. This kind of behavior has been implemented by a physically-based algorithm presented in Section 4.5.4. It does not try to be physically correct, but to be as fast as possible while still providing intuitive and plausible behavior. Experiments have shown that my algorithm needs about 300 microseconds per contact point (not counting collision detection time).

Applications

The frameworks, concepts, and algorithms described above have been implemented as several modules within our VR system *Virtual Design II*. The system has been used for many applications. A lot of them were and are being developed for customers from automotive industries (*virtual prototyping*). Other application domains include edutainment, cultural heritage, and immersive scientific visualization.

In this thesis, virtual prototyping is being defined as the application of virtual reality instead of physical mock-ups. This definition is stronger than the one prevailing throughout manufacturing industries. Their weaker definition usually means the application of software in general instead of physical mock-ups.

Virtual assembly simulation is one of the applications I have developed on top of our VR system. In Chapter 5, virtual prototyping is discussed in general, while the virtual assembly simulation application is described in Section 5.2 in more detail. It is now being introduced in the product process of a large automobile manufacturer.

6.2 Future directions

I believe that virtual assembly simulation is one of the most challenging applications of virtual reality. Although it has matured in that it can be used in the product process, there are still a lot of things to be improved.[2]

Probably the most important missing feature is force-feedback. Especially in virtual assembly and maintenance simulations, acoustic and visual feedback turned out to be not sufficient to meet the demands of the users. Mechanics "see" with their hands, particularly in narrow and complex environments or when they cannot see their hands and/or tools. Therefore, force feedback would add significantly to the degree of immersion and usability, and it would give a natural and expected cue how to resolve collisions. Furthermore, it would prevent a deviation of the (real) hand and the virtual one.

While this poses non-trivial problems by itself, it also poses additional problems in areas dealt with in this thesis, namely collision detection and interaction techniques. A haptic device for virtual assembly simulation must be integrated into the system so that the user will be able to use it intuitively (like a familiar tool) and safely. Requirements on collision detection are very demanding: under all circumstances, query times must be less than 2 milliseconds for checking at least one object against all the environment.

Another important feature is the simulation of flexible parts. I feel that considerable research is still required in order to be able to interact with hoses, wires, bundles, plastic tanks, etc., in real time.

Collision detection. Since collision detection is an enabling technology not only for virtual assembly simulation, I am positive that this area will be scientifically active for many years to come. My timing experiments with the sliding simulation have shown that collision detection is still by a factor of about 10 more time consuming than the simulation algorithm itself. There are algorithms and object representations allowing faster collision detection queries [MPT99] than the algorithms presented in this thesis. However, they have other drawbacks, such as limited accuracy. So, there is still a need for much faster algorithms checking a pair of objects in close proximity.

In particular, three main directions will get greater attention: collision detection of flexible parts, incremental collision detection for polygon soups, and application-specific algorithms, such as collision detection for force-feedback. Algorithm developers will need to keep in mind cache coherency and memory access patterns. Otherwise, in my experience, an algorithm superior in theory might lose in practice.

I believe that collision detection algorithms are suitable for implementation in hardware. However, I am not sure that the approach taken by [BWS99, MOK95, SF91] is the right one. It has several disadvantages: it can handle only convex objects (and similarly "simple" objects), it is not exact, and obtaining a witness might not be straight-forward.[3] The reason for these problems is that rendering architectures are being "misused" to do something they were not designed for.

---

[2] In the article "What's Real about Virtual Reality?" in the November/December '99 issue of IEEE CG&A, Fred Brooks remarked ironically: "*VR that used to* almost *work now* barely *works*" [Bro99].

[3] Currently, reading the stencil or frame buffer is a bottleneck, but that could be overcome by a true implementation in hardware (by a "stencil buffer OR" operation).

To my knowledge, it still remains an open problem if there is a global characterization for the quality (in the sense of fast collision detection) of bounding volume hierarchies which can be computed using only the geometry of the tree itself and without prior construction of the tree. If there is such a global characterization, the next question would be, whether it can be "localized", so that it involves only a bounding volume and its father and child bounding volumes. Such a local characterization would lead to an optimal BV tree construction algorithm.

Interaction. In the area of interaction techniques, natural manipulation of objects still needs considerable research. In particular, the three types of precision grasps of objects by the user's virtual hand in a robust and stable manner is, to my knowledge, unsolved as of this writing. Challenging examples include some common and (seemingly) simple operations: juggling two balls in one hand, turning a screw between index finger and thumb, or wiggling a pencil between index finger and middle finger.

Virtual reality in general always needs some form of tracking. As of yet, the user is almost always tethered by some device, either the head-mounted display, an electro-magnetic sensor, or the data glove. However, virtual reality eventually must become "untethered". While the technology is available already to solve this problem, it is either not commercially available or too expensive.

Future directions for virtual prototyping. The result of a user survey, performed with the application described in Section 5.2, indicates that the use of VR for virtual prototyping will play an important role in the near future in automotive (and probably other manufacturing) industries [Zac99]. In particular, the response of the surveyed users has been very encouraging and optimistic that VR does have the potential to reduce the number of physical mock-ups and improve overall product quality, especially in those steps of the business process chain where humans play an important role.

VR is the best tool (today) to obtain quick answers in an intuitive way in the concept phase of the business process of a product, because in that phase data change often and are available only in a "rough" and preliminary preparation. However, a formal cost/benefit analysis at this time has, to my knowledge, not yet been performed, which might be due to the fact that virtual prototyping has been integrated in the daily productive work environment only for a very limited period and only in very few development processes.

However, VR will not become a wide-spread tool in manufacturing industries before it is seamlessly and completely integrated into the existing CA and IT infrastructure.[4] This is not only a question of conversion and preparation tools: a lot of the data needed for a complete digital mock-up are just not there yet, such as process information, semantical (non-geometric) properties, visual properties of material, etc. This can be solved only in a shift in the design process: design guidelines have to be established with virtual prototyping in mind. All automotive and aerospace companies have realized that and are working on implementing solutions. However, this does not only involve technical aspects of the design process but also a tremendous shift in corporate culture.

[4] For instance, in order to create a complete VE for immersive assembly simulation 70% of the time is needed to find and prepare the CAD data and only 30% for authoring the VE.

Bibliography

[ACCL79] K. N. An, E. Y. Chao, W. P. Cooney, and R. L. Linscheid. Normative model of human hand for biomechanical analysis. *J. Biomechanics*, 12:775–788, 1979.

[ACHS94] Magnus Andersson, Christer Carlsson, Olof Hagsand, and Olov Ståhl. *DIVE — The Distributed Interactive Virtual Environment*. Swedish Institute of Computer Science, 164 28 Kista, Sweden, 1994.

[ADF+95] Peter Astheimer, Fan Dai, Wolfgang Felger, Martin Göbel, Helmut Haase, Stefan Müller, and Rolf Ziegler. Virtual Design II – an advanced VR system for industrial applications. In *Proc. Virtual Reality World '95*, pages 337–363. Stuttgart, Germany, February 1995.

[AES94] S. S. Abi-Ezzi, and S. Subramaniam. Fast dynamic tessellation of trimmed NURBS surfaces. *Computer Graphics Forum*, 13(3):107–126, 1994. Eurographics '94 Conference issue.

[AF92] D. Avis, and K. Fukuda. A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra. *Discrete Comput. Geom.*, 8:295–313, 1992.

[AF95] P. Astheimer, and W. Felger. An interactive virtual world experience – the sbg cyberspace roadshow. In *Second Eurographics Workshop on Virtual Environments '95*, M. Göbel, Ed., pages 199–210. Springer-Verlag, Wien, February 1995.

[AFM93] Peter Astheimer, Wolfgang Felger, and Stefan Müller. Virtual design: A generic VR system for industrial applications. *Computers & Graphics*, 17(6):671–677, 1993.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[AJ88] Kurt Akeley, and Tom Jermoluk. High-performance polygon rendering. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, John Dill, Ed., vol. 22, pages 239–246, August 1988.

[AJE96] Bernard D. Adelstein, Eric R. Johnston, and Stephen R. Ellis. Dynamic response of electromagnetic spatial displacement trackers. *Presence: Teleoperators and Virtual Environments*, 5(3):302–318, 1996.

[Bar94] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Andrew Glassner, Ed., Computer Graphics Proceedings, Annual Conference Series, pages 23–34. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[Bar97] Anthony C. Barkans. High-quality rendering using the talisman architecture. In *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, Steven Molnar and Bengt-Olaf Schneider, Eds., pages 79–88. ACM SIGGRAPH / Eurographics, ACM Press, New York City, NY, August 1997. ISBN 0-89791-961-0.

[Bat82] Klaus-Jürgen Bathe. *Finite Element Procedures in Engineering Analysis*. Prentice-Hall, 1982.

[BBDM98] Mark Billinghurst, J. Bowskill, N. Dyer, and J. Morphett. An evaluation of wearable information spaces. In *Proc. of IEEE Virtual Reality Annual International Symposium; VRAIS '98*. Atlanta, Georgia, March 1998.

[BCG+96] Gill Barequet, Bernard Chazelle, Leonidas J. Guibas, Joseph S. B. Mitchell, and Ayellet Tal. BOXTREE: A hierarchical representation for surfaces in 3D. *Computer Graphics Forum*, 15(3):387–396, August 1996. Proceedings of Eurographics '96. ISSN 1067-7055.

[BD83] G. Boothroyd, and P. Dewhurst. Design for assembly – a designer's handbook. Tech. rep., Department of Mechanical Engineering, University of Massachusetts at Amherst, 1983.

[BDH93] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The Quickhull algorithm for convex hull. Technical Report GCG53, Geometry Center, Univ. of Minnesota, July 1993.

[BDH96] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22 (4):469–483, December 1996.

[Ben90] J. L. Bentley. *K*-d trees for semidynamic point sets. In *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, pages 187–197, 1990.

[BF79a] J. L. Bentley, and J. H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, December 1979.

[BF79b] J. L. Bentley, and J. H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11:397–409, 1979.

[BG95] Bruce M. Blumberg, and Tinsley A. Galyean. Multi-level direction of autonomous creatures for real-time virtual environments. In *Siggraph 1995 Conference Proc.*, Robert Cook, Ed., pages 47–54, August 1995.

[BH95] Bradford Barber, and Hannu Huhdanpaa. *Qhull manual*. The Geometry Center, formerly with the Universiy of Minnesota, 1995. URL `http://www.geom.umn.edu/software/qhulll/qh-man.htm`.

[BH97] Bradford Barber, and Hannu Huhdanpaa. Qhull version 2.4, April 1997. URL `http://www.geom.umn.edu/software/download/qhull.html`. Software.

[Bim99] Oliver Bimber. Continuous 6-DOF gesture recognition: a fuzzy logic approach. In *7th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media (WSCG '99)*, Vaclav Skala, Ed. University of West Bohemia, Plzen, Czech Republic, February8–12 1999.

[Bir85] Ken Birman. Replication and fault-tolerance in the isis system. *Proc. of the 12th ACM Symposium on Operating Systems*, pages 79–86, 1985.

[BKH98] D. A. Bowman, D. Koller, and L. F. Hodges. A methodology for the evaluation of travel techniques for immersive virtual environments. *Virtual Reality*, 3:120–131, 1998.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 322–331, 1990.

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Englewood Cliffs, New Jersey: Prentice Hall, 1981.

[Bol80] R. A. Bolt. Put-that-there: Voice and gesture at the graphics interface. *Computer Graphics*, 14(3):262–270, July 1980.

[Bro86] Frederick P. Brooks, Jr. Walkthrough — A dynamic graphics system for simulating virtual buildings. In *Proceedings of 1986 Workshop on Interactive 3D Graphics*, Frank Crow and Stephen M. Pizer, Eds., pages 9–21, October 1986.

[Bro99] Frederick P. Brooks, Jr. What's real about virtual reality? *IEEE Computer Graphics & Applications*, 19(6):16–27, November, December 1999.

[BRT96] Ronan Boulic, Serge Rezzonico, and Daniel Thalmann. Multi-finger manipulation of virtual objects. In *Proc. of the ACM Symposium on Virtual Reality Software and Technology (VRST '96)*, pages 67–74. Hong Kong, July1-4 1996.

[Bry92] Steve Bryson. Measurement and calibration of static distortion of position data from 3D trackers. In *Siggraph '92, 19th International Conference On Computer Graphics and Interaction Techniques, Course Notes 9*, pages 8.1–8.12, 1992.

[BS90] I. Beichl, and F. Sullivan. A robust parallel triangulation and shelling algorithm. In *Proc. 2nd Canad. Conf. Comput. Geom.*, pages 107–111, 1990.

[BS98] Matthias Buck, and Elmar Schömer. Interactive rigid body menipulation with obstacle contacts. *The Journal of Visuazlization and Computer Animation*, 9:243–257, 1998.

[Buc98] Matthias Buck. Immersive user interaction within industrial virtual environments. In *Virtual Reality for Industrial Applications*, Fan Dai, Ed., Computer Graphics: Systems and Applications, chapter 2, pages 39–59. Springer, Berlin, Heidelberg, 1998.

[Bun07] Pieter G. Buning. Numerical algorithms in CFD post-processing. *van Karman Institute for Fluid Dynamics*, Lecture Series:1–20, 1989-07.

[BV91] W. Bouma, and G. Vanecek, Jr. Collision detection and analysis in a physical based simulation. In *Eurographics Workshop on Animation and Simulation*, pages 191–203, 1991.

[BWS99] George Baciu, Wingo Sai-Keung Wong, and Hanqiu Sun. Recode: an image-based collision detection algorithm. *The Journal of Visualization and Computer Animation*, 10(4):181–192, October - December 1999. ISSN 1049-8907.

[Can86] John Canny. Collision detection for moving polyhedra. *IEEE Transactions an Pattern Analysis and Machine Intelligence*, PAMI-8(2):200–209, March 1986.

[Car98] N. Carriero. An implementation of Linda for a NUMA Machine. *Parallel Computing*, 24(7):1005–1021, 1998.

[CAS92] Gregory M. Herb Clifford A. Shaffer. A real-time robot arm collision avoidance system. *IEEE Transactions on Robotics and Automation*, 8(2), April 1992.

[CCT89] Niels Vejrup Carlsen, Niels Jorgen Christensen, and Hugh A. Tucker. An event language for building user interface frameworks. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, User Interface Structures II, pages 133–140, 1989.

[CCV85] I. Carlbom, I. Chakravarty, and D. Vanderschel. A hierarchical data structure for representing the spatial decomposition of 3-D objects. *IEEE Computer Graphics and Applications*, 5(4):24–31, April 1985.

[CD87] Bernard Chazelle, and D. P. Dobkin. Intersection of convex objects in two and three dimensions. *J. ACM*, 34(1):1–27, January 1987.

[CDG97] Chi-Cheng P. Chu, Tushar H. Dani, and Rajit Gadh. Multi-sensory user interface for a virtual-reality-based computer-aided design system. *Computer-Aided Design*, 29(10):709–725, 1997.

[CF91] R. E. Carlson, and Th. A. Foley. The parameter r^2 in multiquadric interpolation. *Computers & Mathematics with Applications*, 21:29–42, 1991.

[CH93] Christer Carlsson, and Olof Hagsand. DIVE — A platform for multi-user virtual environments. *Computers and Graphics*, 17(6): 663–669, November–December 1993. CODEN COGRD2. ISSN 0097-8493.

[Cha93] Bernard Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete Comput. Geom.*, 10:377–409, 1993.

[Chu96] Kelvin Chung. Quick collision detection of polytopes in virtual environments. In *Proc. of the ACM Symposium on Virtual Reality Software and Technology (VRST'96)*, Mark Green, Ed., pages 125–131, 1996.

[CJKL93] C. Codella, R. Jalili, L. Koved, and B. Lewis. A toolkit for developing multi-user, distributed virtual environments. *Proceedings of VRAIS'93*, pages 401–407, 1993.

[CK70] D. R. Chand, and S. S. Kapur. An algorithm for convex polytopes. *J. ACM*, 17(1):78–86, January 1970.

[CLM+] John Cohen, Ming C. Lin, Dinesh Manocha, Brian Mirtich, M. K. Ponamgi, and John Canny. I_COLLIDE. URL `http://www.cs.unc.edu/~geom/I_COLLIDE.html`. Software.

[CLMP95] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *1995 Symposium on Interactive 3D Graphics*, Pat Hanrahan and Jim Winget, Eds., pages 189–196. ACM SIGGRAPH, April 1995. ISBN 0-89791-736-7.

[CM87] Yong C. Chen, and Catherine M. Murphy. H-P model — A hierarchical space decomposition in a polar coordinate system. In *Computer Graphics 1987 (Proceedings of CG International '87)*, Tsiyasu L. Kunii, Ed., pages 443–459. Springer-Verlag, 1987.

[CM91] Edwin W. Cook, and Gregory A. Miller. Digital filtering: Background and tutorial for psychophysiologists. Technical report uiuc-bi-cns-91-03, The Beckman Institute, University of Illinois, Urbana, IL, 61801, 1991.

[CN97] Carolina Cruz-Neira. Introduction to virtual reality. In *Applied Virtual Reality*, Carolina Cruz-Neira, Ed., Siggraph 1997, Course 15 Notes, part 2, pages 2–1 – 2–15. Los Angeles, August 1997.

[CS88] K. L. Clarkson, and P. W. Shor. Algorithms for diametral pairs and convex hulls that are optimal, randomized, and incremental. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 12–17, 1988.

[CSD93] Carolina Cruz-Neira, Daniel J. Sandin, and Thomas A. DeFanti. Surround-screen projection-based virtual reality: The design and implementation of the CAVE. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, James T. Kajiya, Ed., vol. 27, pages 135–142, August 1993.

[CW92] Dale Chapman, and Colin Ware. Manipulating the future: Predictor based feedback for velocity control in virtual environment navigation. In *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, David Zeltzer, Ed., vol. 25, pages 63–66, March 1992.

[Dai98] Fan Dai. Virtual prototyping — principles, problems, solutions. In *Tutorial Notes of IEEE Virtual Reality Annual International Symposium; VRAIS '98*. Atlanta, Georgia, March 1998.

[Dee92] Michael F. Deering. High resolution virtual reality. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, Edwin E. Catmull, Ed., vol. 26, pages 195–202, July 1992.

[DEG+97] Dr. Ping Dai, Dr. Gerhard Eckel, Dr. Martin Göbel, Frank Hasenbrink, Dr. Vali Lalioti, Uli Lechner, Johannes Strassner, Henrik Tramberend, and Gerold Weschke. Virtual sapces – vr porjection system technologies and applications. In *Eurographics '97 Tutorial*. Blackwell Publishers, August 1997.

[Dev89] Olivier Devillers. The macro-regions: an efficient space subdivision structure for ray tracing. In *Eurographics '89*, W. Hansmann, F. R. A. Hopgood, and W. Strasser, Eds., pages 27–38. Elsevier / North-Holland, September 1989.

[DFF$^+$96] Fan Dai, Wolfgang Felger, Thomas Frühauf, Martin Göbel, Dirk Reiners, and Gabriel Zachmann. Virtual prototyping examples for automotive industries. In *Proc. Virtual Reality World*. Stuttgart, February 1996.

[Dit97] Mary Lynne Dittmar. Psychological and physiological effect of immersive environments. In *Applied Virtual Reality*, Carolina Cruz-Neira, Ed., Siggraph 1997, Course 15 Notes, part 6, pages 6–1 – 6–11. Los Angeles, August 1997.

[DK83] D. P. Dobkin, and D. G. Kirkpatrick. Fast detection of polyhedral intersection. *Theoret. Comput. Sci.*, 27(3):241–253, December 1983.

[DK85] D. P. Dobkin, and D. G. Kirkpatrick. A linear algorithm for determining the separation of convex polyhedra. *J. Algorithms*, 6:381–392, 1985.

[DR86] Fan Dai, and P. Reindl. Enabling digital mock-up with virtual reality techniques - vision, concept, demonstrator. In *ASME Design for Manufacturing Conferences*. Irvine, CA, August 1986.

[DR96] Fan Dai, and Peter Reindl. Enabling digital mock up with virtual reality techniques – vision, concept, demonstrator. In *Proceedings of 1996 ASME Design Engineering Technical Conference and Computers in Engineering*. ASME, Irvine, California, August 1996.

[DS77] T.A. DeFanti, and D.J. Sandin. Final report to the national endowment of the arts. Tech. Rep. US NEA R60-34-163, University of Illinois at Chicago Circle, Chicago, Illinois, 1977.

[DS97] George Drettakis, and François Sillion. Interactive update of global illumination using A line-space hierarchy. In *SIGGRAPH 97 Conference Proceedings*, Turner Whitted, Ed., Annual Conference Series, pages 57–64. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[duP95] Pierre duPont. Building complex virtual worlds without programming. In *Eurographics '95 State of the Art Reports*, Remo C. Veltkamp, Ed., pages 61–70. Maastricht, August 1995.

[Dye82] C. R. Dyer. The space efficiency of quadtrees. *Computer Graphics and Image Processing*, 19:335–348, August 1982.

[Dyn87] N. Dyn. Interpolation of scattered data by radial functions. In *Topics in Multivariate Approximation*, F. I. Uteras C. K. Chul, L. L. Schumaker, Ed., pages 47–61. Academic Press, 1987.

[EDd$^+$96] J.L. Encarnação, F. Dai, A. del Pino, H. Haase, U. Jakob, M. Unbescheiden, and G. Zachmann. Grenzen der Virtualisierung. Talk at Münchner-Kreis KongreßM"unchen, November 1996.

[EH72] J. Elzinga, and D. Hearn. The minimum covering sphere problem. *Manage. Sci.*, 19:96–104, 1972.

[EJ97] Peter Eberhard, and Shoushan Jian. Collision detection for contact problems in mechanics with a boundary search algorithm. *Mathematical Modelling of Systems*, 3(4):265–281, 1997. URL `http://www.mechb.uni-stuttgart.de/Leute/Eberhard/eberhard_publikationen.html`.

[EK89] M. Eppinger, and E. Kreuzer. Systematischer Vergleich von Verfahren zur Rückwärtstransformation bei Industrierobotern. *Robotersysteme*, 5:219–228, 1989.

[ES88] José L. Encarnação, and Wolfgang Straßer. *Computer Graphics*. Oldenbourg Verlag, 3 ed., 1988.

[ES99] Jens Eckstein, and Elmar Schömer. Dynamic collision detection in virtual reality applications. In *Proc. The 7-th International Conference in Central Europe on Computer Graphics, Visualization, and Interactive Digital Media '99 (WSCG'99)*, pages 71–78. University of West Bohemia, Plzen, Czech Republic, February 1999.

[EWQ99] S. S. Everett, K. Wauchope, and M. A. Péez Quiñones. Creating natural language interfaces to VR systems. *Virtual Reality*, 4(2): 103–113, 1999.

[FA85] W. R. Franklin, and V. Akman. Building an octree from a set of parallelepipeds. *IEEE Computer Graphics and Applications*, 5(10):58–64, October 1985.

[Far90] Gerald Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, 1990. 464 pp.

[FB90a] Steven Feiner, and Clifford Beshers. Visualizing n-dimensional virtual worlds with n-vision. In *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, Rich Riesenfeld and Carlo Sequin, Eds., vol. 24, pages 37–38, March 1990.

[FB90b] A. U. Frank, and R. Barrera. The fieldtree: A data structure for geographic information systems. In *Proceedings of the 1st Symposium SSD on Design and Implementation af Large Spatial Databases*, A. Buchmann, O. Günther, T. R. Smith, and Y.-F. Wang, Eds., vol. 409 of *LNCS*, pages 29–44. Springer, Berlin, July 1990. ISBN 3-540-52208-5.

[FE97] William P. Flanagan, and Rae Earnshaw. Applications: Meeting the future at the University of Michigan Media Union. *IEEE Computer Graphics and Applications*, 17(3):15–19, May/June 1997. CODEN ICGADZ. ISSN 0272-1716.

[Fel95] Wolfgang Felger. *Innovative Interaktionstechniken in der Visualisierung*. Springer, 1995. Reprint of the dissertation.

[Fis95] Paul A. Fishwick. A taxonomy for simulation modeling based on programming language principles. *IIE Transactions on IE Research*, 1995. URL `http://www.cis.ufl.edu/~fishwick/tr/tr95-019.html`.

[Fis96] Paul A. Fishwick. Computer simulation: The art and science of digital world construction. *IEEE Potentials*, pages 24–27, February/March 1996. URL `http://www.cis.ufl.edu/~fishwick/introsim/paper.html`.

[FK85] Kikuo Fujimura, and Tosiyasu L. Kunii. A hierarchical space indexing method. In *Computer Graphics Visual Technology and Art (Proceedings of Computer Graphics Tokyo '85)*, Tsiyasu L. Kunii, Ed., pages 21–33. Springer-Verlag, 1985.

[FKN80] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. In *Computer Graphics (SIGGRAPH '80 Proceedings)*, vol. 14, pages 124–133, July 1980.

[FMHR86] S. S. Fisher, M. McGreevy, J. Humphries, and W. Robinett. Virtual environment display system. In *Proceedings of 1986 Workshop on Interactive 3D Graphics*, Frank Crow and Stephen M. Pizer, Eds., pages 77–87, October 1986.

[FN94] Thomas A. Foley, and Gregory M. Nielson. Modeling of scattered multivariate data. *Eurographics State of the Art Reports*, pages 39–59, 1994.

[Fra82] R. Franke. Scattered data interpolation: test of some methods. *Math Computation*, 38:181–200, 1982.

[Frü97] Thomas Frühauf. *Graphisch-Interaktive Strömungsvisualisierung*. Beiträge zur graphischen Datenverarbeitung. Springer-Verlag, Berlin Heidelberg New York, 1997.

[FS93] Thomas A. Funkhouser, and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, James T. Kajiya, Ed., vol. 27, pages 247–254, August 1993.

[FSP92] M. Friedmann, T. Starner, and A. Pentland. Device synchronisation using an optimal linear filter. *SIGGRAPH Symposium on Interactive 3D Graphics*, pages 57–62, 1992.

[FSZ94] Wolfgang Felger, Reiner Schäfer, and Gabriel Zachmann. Interaktions-toolkit. Tech. Rep. FIGD-94i002, Fraunhofer Institute for Computer Graphics, Darmstadt, January 1994.

[FT96] D. N. Fogel, and L. Tinney. Image registration using multiquadric functions. Tech. Rep. 96-01, National Center for Geographic Information and Analysis, University of California, Santa Barbara, 1996.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1990. Overview of research to date.

[GA93] I. Gargantini, and H. H. Atkinson. Ray tracing an octree: Numerical evaluation of the first intersection. *Computer Graphics forum*, 12 (4):199–210, 1993.

[Gas93] Marie-Paule Gascuel. An implicit formulation for precise contact modeling between flexible solids. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, James T. Kajiya, Ed., vol. 27, pages 313–320, August 1993.

[GAS+95] Morteza Ghazisaedy, David Adamczyk, Daniel J. Sandin, Robert V. Kenyon, and Thomas A. DeFanti. Ultrasonic calibration of a magnetic tracker in a virtual reality space. In *Proceedings of the IEEE Virtual Reality Annual International Symposium (VRAIS '95)*, pages 179–188, March 1995.

[Ghe95] Steve Ghee. dVS – a distributed VR systems infrastructure. In *Course Notes: Programming Virtual Worlds, SIGGRAPH '95*, Anselmo Lastra and Henry Fuchs, Eds., pages 6–1 – 6–30, 1995.

[GHT98] B. Grant, A. Helser, and R. Taylor. Adding force display to a stereoscopic head-tracked projection display. In *Proceedings of the IEEE Virtual Reality Annual International Symposium (VRAIS '98)*. Atlanta, Georgia, March 1998.

[GJK88] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between complex objects. *Internat. J. Robot. Autom.*, 4(2):193–203, 1988.

[Gla90] Andrew S. Glassner, Ed. *Graphics Gems*. Academic Press, San Diego, CA, 1990.

[GLM96] Stefan Gottschalk, Ming Lin, and Dinesh Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. In *SIGGRAPH 96 Conference Proceedings*, Holly Rushmeier, Ed., Annual Conference Series, pages 171–180. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[GMPP95] Stephen Ghee, Mark Mine, Randy Pausch, and Kenneth Pimentel. *Course Notes: Programming Virtual Worlds, SIGGRAPH '95*. 1995.

[Göb95] M. Göbel, Ed. *Virtual Environments '95*, Eurographics. Springer-Verlag Wien New York, 1995. Proc's Eurographics Workshop, Barcelona, Spain, 1993, and Monte Carlo, Monaco, 1995.

[Got97] Stefan Gottschalk. Rapid library, 1997. URL `http://www.cs.unc.edu/~geom/OBB/OBBT.html`. Vers. 2.01.

[GR85] Adele Goldberg, and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, 1985.

[Gre84] M. Green. Report on dialogue specification tools. *Computer Graphics Forum*, 3(4):305–314, December 1984.

[Gre86] Mark Green. A survey of three dialogue models. *ACM Transactions on Graphics*, 5(3):244–275, 1986.

[GS87] Jeffrey Goldsmith, and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, May 1987.

[Gv89] G. H. Golub, and C. F. van Loan. *Matrix Computations*. John Hopkins University Press, Baltimore, 2nd ed., 1989.

[GVP91] Marie-Paule Gascuel, Anne Verroust, and Claude Puech. Animation and collisions between complex deformable bodies. In *Proceedings of Graphics Interface '91*, pages 263–270, June 1991.

[Hah88a] James K. Hahn. Realistic animation of rigid bodies. *Computers & Graphics*, 22(4):299–308, August 1988.

[Hah88b] James K. Hahn. Realistic animation of rigid bodies. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, John Dill, Ed., vol. 22, pages 299–308, August 1988.

[Han97] Chris Hand. A survey of 3D interaction techniques. *Computer Graphics Forum*, 16(5):269–281, 1997. ISSN 1067-7055.

[Hd89] Wolfgang Hübner, and Manuel de Lancastre. Towards an object-oriented interaction model for graphics user interfaces. *Computer Graphics Forum*, 8(3):207–217, September 1989.

[HD93] Larry F. Hodges, and Elizabeth Thorpe Davis. Geometric considerations for stereoscopic virtual environments. *Presence*, 2(1):34–43, winter 1993.

[HD00] Elke Hergenröther, and Patrick Dähne. Real-time virtual cables based on kinematic simulation. In *Proc. WSCG '2000, The 8-th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media 2000*, pages 402–409. University of West Bohemia, Plzen, Czech Republic, February 2000.

[Hel97] Martin Held. ERIT: A collection of efficient and reliable intersection tests. *Journal of Graphics Tools*, 2(4):25–44, 1997.

[HF93] Daniel Henry, and Tom Furness. Spatial perception in virtual environments: Evaluating an architectural application. In *IEEE Virtual Reality Annual International Symposium*, pages 33–40, September 18–22 1993.

[HG94] Sean Halliday, and Mark Green. A geometric modeling and animation system for virtual reality. In *Virtual Reality Software and Technology (VRST 94)*, Gurminder Singh, Steven Feiner, and Daniel Thalmann, Eds., pages 71–84, August 1994.

[HK97] Taosong He, and Arie Kaufman. Collision detection for volumetric models. In *IEEE Visualization ́97*, Roni Yagel and Hans Hagen, Eds., pages 27–34. IEEE, November 1997.

[HKM96] Martin Held, James T. Klosowski, and Joseph S.B. Mitchell. Real-time collision detection for motion simulation within complex environments. In *Siggraph 1996 Technical Sketches, Visual Proceedings*, page 151. New Orleans, August 1996.

[HKP91] John Hertz, Anders Krogh, and Richard G. Palmer. *Introduction to the Theory of Neural Computing*. Addison-Wesley, 1991.

[HL92] Josef Hoschek, and Dieter Lasser. *Grundlagen der geometrischen Datenverarbeitung*. B.G. Teubner, Stuttgart, 2 ed., 1992.

[HL93] Joseph Hoschek, and Dieter Lasser. *Fundamentals of Computer Aided Geometric Design*. A K Peters, 1993. ISBN 1-56881-007-5.

[HLS97] Olof Hagsand, Rodger Lea, and Märten Stenius. Using spatial techniques to decrease message passing in a distributed VR system. In *VRML 97: Second Symposium on the Virtual Reality Modeling Language*, Rikk Carey and Paul Strauss, Eds. ACM SIGGRAPH / ACM SIGCOMM, ACM Press, New York City, NY, February 1997. ISBN 0-89791-886-x.

[Hol97] Richard L. Holloway. Registration error analysis for augmented reality. *Presence*, 6(4):413–432, August 1997.

[HT92] Ping-Kang Hsiung, and Robert H. Thibadeau. Accelerating ARTS. *The Visual Computer*, 8(3):181–190, March 1992.

[Hüb90] Wolfgang Hübner. *Entwurf Graphischer Benutzerschnittstellen. Ein objektorientiertes Interaktionsmodell zur Spezifikation graphischer Dialoge*. Springer Verlag, 1990.

[Hub93] Philip M. Hubbard. Interactive collision detection. In *IEEE Symposium on Research Frontiers in VR, San José, California*, pages 24–31, October 25–26 1993.

[Hub95] P. M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1 (3):218–230, September 1995. ISSN 1077-2626.

[Jay98] Sankar Jayaram. Creating and managing virtual menus in immersive environments. In *Proc. of the 1998 ASME Design Engineering Technical Conferences*. Atlanta, Georgia, September 1998. paper no. DETC98/CIE-5534.

[JBD+90] Bob Jacobson, John Barlow, Esther Dyson, Timothy Leary, William Bricken, Warren Robinett, and Jaron Lanier. Hip, hype and hope — the three faces of virtual worlds. In *Computer Graphics (SIGGRAPH '90 Panel Proceedings)*, Alyce Kaprow, Ed., vol. 24, pages 10.1–10.29, August 1990.

[JCL97] Sankar Jayaram, Hugh I. Connacher, and Kevin W. Lyons. Virtual assembly using virtual reality techniques. *Computer-aided Design*, 29(8):575–584, 1997.

[JFH92] Dylan M. Jones, Clive R. Frankish, and Kevin Hapeshi. Automatic speech recognition in practice. *Behaviour and Information Technology*, 11(2):109–122, 1992.

[JJWT99] Sankar Jayaram, Uma Jayaram, Yong Wang, and Hrishikesh Tirumal. VADE: A virtual assembly design environment. *IEEE Computer Graphics & Applications*, 19(6):44–50, November, December 1999.

[Jon97] Lynette Jones. Dextrous hands: Human, prosthetic, and robotic. *Presence*, 6(1):29–56, February 1997.

[Kal84] M. Kallay. The complexity of incremental convex hull algorithms in R^d. *Inform. Process. Lett.*, 19:197, 1984.

[KGL+98] S. Krishnan, M. Gopi, M. Lin, Dinesh Manocha, and A. Pattekar. Rapid and accurate contact determination between spline models using shelltrees. In *Computer Graphics Forum, Proc. of Eurographics '98*, N. Ferreira and M. Göbel, Eds., vol. 17, pages 315–326. Blackwell Publishers, 1998. ISSN 1067-7055.

[KH95] R. Kijima, and M. Hirose. Fine object manipulation in virtual environments. In Göbel [Göb95], pages 42–58. Proc's Eurographics Workshop, Barcelona, Spain, 1993, and Monte Carlo, Monaco, 1995.

[KHM+98] James T. Klosowski, Martin Held, Jospeh S.B. Mitchell, Henry Sowrizal, and Karel Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, January 1998.

[KHM99] Jim Klosowski, Martin Held, and Joe Mitchell. QuickCD, software library for efficient collision detection, 1999. URL `http://www.ams.sunysb.edu/~jklosow/quickcd/QuickCD.html`. Vers. 1.00.

[Kin99] Volodymyr Kindratenko. Calibration of electromagnetic tracking devices. *Virtual Reality*, pages 139–150, 1999.

[KK86] Timothy L. Kay, and James T. Kajiya. Ray tracing complex scenes. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, David C. Evans and Russell J. Athay, Eds., vol. 20, pages 269–278, August 1986.

[KML95] Subodh Kumar, Dinesh Manocha, and Anselmo Lastra. Interactive display of large-scale NURBS models. In *1995 Symposium on Interactive 3D Graphics*, Pat Hanrahan and Jim Winget, Eds., pages 51–58. ACM SIGGRAPH, April 1995. ISBN 0-89791-736-7.

[Kol97] Ivana Kolingerová. Convex polyhedron-line intersection detection using dual representation. *The Visual Computer*, 13(1):42–49, 1997. ISSN 0178-2789.

[KPLM98] S. Krishnan, A. Pattekar, M. Lin, and D. Manocha. Spherical shells: A higher order bounding volume for fast proximity queries. In *Proc. 3rd Workshop Algorithmic Found. Robot.*, page to appear, 1998. URL `http://www.cs.unc.edu/~dm/collision.html`.

[Kru83] M. W. Krueger. *Artificial Reality*. Addison-Wesley, 1983. 312 pp.

[KS97] Kryzsztof S. Klimansezewski, and Thomas W. Sederberg. Faster ray tracing using adaptive grids. *IEEE Computer Graphics & Applications*, 17(1):42–51, January-February 1997. ISSN 0272-1716.

[Kur93] Gordon Paul Kurtenbach. *The Design and Evaluation of Marking Menus*. PhD dissertation, University of Toronto, Graduate Department of Computer Science, 1993. URL `http://reality.sgi.com/gordo_tor/papers/PhdThesis/PhDthesis.html`.

[KV98] Jason J. Kelsick, and Judy M. Vance. The VR factory: discrete event simulation implemented in a virtual environment. In *Proc. of 1998 ASME Design Engineering Technical Conferences / Design for Manufacturing*. Atlanta, Georgia, September 1998.

[Lau60] Detlef Laugwitz. *Differentialgeometrie*. B. G. Teubner, Stuttgart, 1960.

[LB84] Y.-D. Liang, and B. A. Barsky. A new concept and method for line clipping. *ACM Trans. Graphics (USA)*, 3:1–22, January 1984.

[LB98] L. Lakshmi, and D. R. S. Boyd. Virtual environments for engineering applications. *Virtual Reality*, 3(4):235–244, 1998.

[LC92] Ming C. Lin, and John F. Canny. Efficient collision detection for animation, September 1992.

[LC98] Tsai-Yen Li, and Jin-Shin Chen. Incremental 3d collision detection with hierarchical data structures. In *Proc. VRST '98*, pages 139–144. ACM, Taipei, Taiwan, November 1998.

[LD97] Valerie D. Lehner, and Thomas A. DeFanti. Projects in VR: Distributed virtual reality: Supporting remote collaboration in vehicle design. *IEEE Computer Graphics and Applications*, 17(2):13–17, March/April 1997. CODEN ICGADZ. ISSN 0272-1716.

[LO96] Rung-Heui Liang, and Ming Ouhyoung. A sign language recognition system using hidden markov model and context sensitive search. In *Proc. of the ACM Symposium on Virtual Reality Software and Technology (VRST '96)*. Hong Kong, July1-4 1996.

[Lof95] R. Bowen Loftin. Virtual reality links astronaut training. *Real Time Graphics*, 4(4):4–5, October/November 1995.

[Lou93] Kenneth C. Louden. *Programming Languages – Principles and Practice*. PWS-Kent, Boston, Massachusetts, 1993.

[LR82] Y. T. Lee, and A. A. G. Requicha. Algorithms for computing the volume and other integral properties of solids. I. Known methods and open issues. *Commun. ACM*, 25:635–641, 1982.

[LS97] Mark A. Livingston, and Andrei State. Magnetic tracker calibration for improved augmented reality registration. *Presence*, 6(5): 532–546, October 1997.

[LSG91] J. Liang, C. Shaw, and M. Green. On temporal-spatial realism in the virtual reality environment. *Proceedings ACM UIST'91 4th Annual ACM Symposium on User Interface Software and Technology*, pages 19–25, 1991.

[LT99] Peter Lindstrom, and Greg Turk. Evaluation of memoryless simplification. *IEEE Transactions on Visualization and Computer Graphics*, 5 (2):98–115, April - June 1999. ISSN 1077-2626.

[LW98] Marc E. Latoschik, and Ipke Wachsmuth. Exploiting distant pointing gestures for object selection in a virtual environment. In *Gesture and Sign Language in Human-Computer Interaction*, Ipke Wachsmuth and Martin Fröhlich, Eds., Lecture Notes in Artificial Intelligence, pages 185–196. Springer Verlag, vol. 1371 1998.

[MAK88] Robert N. Moll, Michael A. Arbib, and A. J. Kfoury. *An Introduction to Formal Language Theory*. Springer, Berlin, 1988. ISBN 0-387-96698-6.

[MBS97] Mark R. Mine, Frederick P. Brooks, Jr., and Carlo H. Séquin. Moving objects in space: Exploiting proprioception in virtual-environment interaction. In *SIGGRAPH 97 Conference Proceedings*, Turner Whitted, Ed., Annual Conference Series, pages 19–26. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[MCR90] Jock D. Mackinlay, Stuart K. Card, and George G. Robertson. Rapid controlled movement through a virtual 3D workspace. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, Forest Baskett, Ed., vol. 24, pages 171–176, August 1990.

[Meg83] N. Megiddo. Linear-time algorithms for linear programming in R^3 and related problems. *SIAM J. Comput.*, 12:759–776, 1983.

[MEP92] S. Molnar, J. Eyles, and J. Poulton. Pixelflow: High-speed rendering using image composition. *Proc. SIGGRAPH '92 Computer Graphics*, 26(2):231–240, July 1992.

[MMZ94] Jai Menon, Richard J. Marisa, and Jovan Zagajac. More powerful solid modeling through ray representations. *IEEE Computer Graphics and Applications*, pages 22–35, May 1994.

[MOK95] Karol Myszkowski, Oleg G. Okunev, and Tosiyasu L. Kunii. Fast collision detection between complex solids using rasterizing graphics hardware. *The Visual Computer*, 11(9):497–512, 1995. ISSN 0178-2789.

[Möl97] Tomas Möller. A fast triangle-triangle intersection test. *Journal of Graphics Tools*, 2(2):25–30, 1997.

[MP78] D. E. Muller, and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoret. Comput. Sci.*, 7:217–236, 1978.

[MP90] Roberto Maiocchi, and Barbara Pernici. Directing an animated scene with autonomous actors. *The Visual Computer*, 6(6):359–371, December 1990.

[MPT99] William A. McNeely, Kevin D. Puterbaugh, and James J. Troy. Six degrees-of-freedom haptic rendering using voxel sampling. *Proceedings of SIGGRAPH 99*, pages 401–408, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.

[MS85] K. Mehlhorn, and K. Simon. Intersecting two polyhedra one of which is convex. In *Proc. Found. Comput. Theory*, L. Budach, Ed., vol. 199 of *Lecture Notes Comput. Sci.*, pages 534–542. Springer-Verlag, 1985.

[MSH+92] M. D. J. McNeill, B. C. Shah, M.-P. Hébert, P. F. Lister, and R. L. Grimsdale. Performance of space subdivision techniques in ray tracing. *Computer Graphics forum*, 11(4):213–220, 1992.

[MT] Martin Mellado, and Josep Tornero. On the spherical splines for robot modeling.

[MW88] Matthew Moore, and Jane Wilhelms. Collision detection and response for computer animation. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, John Dill, Ed., vol. 22, pages 289–298, August 1988.

[MZBS95] Wolfgang Müller, Rolf Ziegler, André Bauer, and Edgar Soldner. Virtual reality in surgical arthroscopic training. *Journal of Image Guided Surgery*, 1(5):288–294, 1995.

[NAB86] I. Navazo, D. Ayala, and P. Brunet. A geometric modeler based on the exact octree representation of polyhedra. *Computer Graphics Forum*, 5(2):91–104, June 1986.

[NAT90a] B. Naylor, J. A. Amanatides, and W. Thibault. Merging BSP trees yields polyhedral set operations. *Comput. Graph.*, 24(4):115–124, August 1990. Proc. SIGGRAPH '90.

[NAT90b] Bruce Naylor, John Amanatides, and William Thibault. Merging BSP trees yields polyhedral set operations. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, Forest Baskett, Ed., vol. 24, pages 115–124, August 1990.

[NF89] G. Nielson, and T. Foley. *An affinely invariant metric and its applications*. Academic Press, 1989.

[NT94] Gregory M. Nielson, and John Tvedt. Comparing methods of interpolation for scattered volumetric data. *Siggraph '94, Course Notes 4*, pages 99–123, 1994.

[NW96] Yanghee Nam, and Kwang Yun Wohn. Recognition of space-time hand-gestures using hidden Markov models. In *Proc. of the ACM Symposium on Virtual Reality Software and Technology (VRST '96)*. Hong Kong, July1-4 1996.

[OB79] J. O'Rourke, and N. I. Badler. Decomposition of three-dimensional objects into spheres. *IEEE Trans. Pattern Anal. Mach. Intell.*, PAMI-1: 295–305, 1979.

[OCW94] S. L. Oviatt, P. R. Cohen, and M. Wang. Toward interface design for human language technology: Modality and structure as determinants of linguistic complexity. *Speech Communication*, 15:283–300, December 1994.

[OD99] Carol O'Sullivan, and John Dingliana. Real-time collision detection and response using sphere-trees. In *15th Spring Conference on Computer Graphics*, pages 83–92. Budmerice, Slovakia, April 1999. ISBN 80-223-1357-2.

[ODK97] Sharon Oviatt, Antonella DeAngeli, and Karen Kuhn. Integration and synchronization of input modes during multimodal human-computer interaction. In *Proceedings of ACM CHI 97 Conference on Human Factors in Computing Systems*, vol. 1 of *PAPERS: Speech, Haptic, & Multimodal Input*, pages 415–422, 1997. URL `http://www.acm.org/sigchi/chi97/proceedings/paper/slo.htm`.

[Ols84] Dan R. Olsen, Jr. Pushdown automata for user interface management. *ACM Transactions on Graphics*, 3(3):177–203, July 1984.

[O'R85] Joseph O'Rourke. Finding minimal enclosing boxes. *Internat. J. Comput. Inform. Sci*, 14:183–199, June 1985.

[Ous98] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998. URL `http://www.scriptics.com/people/john.ousterhout/scripting.ps`.

[Ove88a] M. H. Overmars. Efficient data structures for range searching on a grid. *J. Algorithms*, 9:254–275, 1988.

[Ove88b] M. H. Overmars. Geometric data structures for computer graphics: an overview. In *Theoretical Foundations of Computer Graphics and CAD*, R. A. Earnshaw, Ed., vol. 40 of *NATO ASI Series F*, pages 21–49. Springer-Verlag, 1988.

[Ove88c] Mark H. Overmars. Computational geometry on a grid: An overview. In *Theoretical Foundations of Computer Graphics and CAD*, R. A. Earnshaw, Ed., vol. F40 of *NATO ASI*, pages 167–184. Springer-Verlag, 1988.

[OY96] Takashi Oishi, and Susumu Yachi. Methods to calibrate projection transformation parameters for see-through head-mounted displays. *Presence*, pages 122–135, winter 1996.

[Pan98] Pantograph. In *Britannica Online*, Encyclopædia Britannica, Ed. Encyclopædia Britannica, Inc., 1998. URL `http://search.eb.com/bol/topic?eu=59755&sctn=1`.

[PBBW95] Randy Pausch, Tommy Burnette, Dan Brockway, and Michael E. Weiblen. Navigation and locomotion in virtual worlds via flight into Hand-Held miniatures. In *SIGGRAPH 95 Conference Proceedings*, Robert Cook, Ed., Annual Conference Series, pages 399–400. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

[Pet94] John W. Peterson. Tessellation of NURB surfaces. In *Graphics Gems IV*, Paul Heckbert, Ed., pages 286–320. Academic Press, Boston, 1994.

[Pfa85] G. E. Pfaff, Ed. *User Interface Management Systems*. Academic Press, Springer, 1985.

[PFTV88] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, 1988.

[PH77] F. P. Preparata, and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Commun. ACM*, 20:87–93, 1977.

[PML97] Madhav K. Ponamgi, Dinesh Manocha, and Ming C. Lin. Incremental algorithms for collision detection between polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(1): 51–64, January – March 1997. ISSN 1077-2626.

[PNW98] I. Poupyrev, T. Numada, and S. Weghorst. Virtual notepad: handwriting in immersive vr. In *Proc. of IEEE Virtual Reality Annual International Symposium; VRAIS '98*, pages 126–132. Atlanta, Georgia, March 1998.

[PPW97] Randy Pausch, Dennis Proffitt, and George Williams. Quantifying immersion in virtual reality. In *SIGGRAPH 97 Conference Proceedings*, Turner Whitted, Ed., Annual Conference Series, pages 13–18. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[Pra95] M. J. Pratt. Virtual prototypes and product models in mechanical engineering. In *Virtual Prototyping – Virtual environments and the product design process*, Joachim Rix, Stefan Haas, and José Teixeira, Eds., chapter 10, pages 113–128. Chapman & Hall, 1995.

[PRS+98] F. Purschke, R. Rabätje, M. Schulze, A. Starke, M. Symietz, and P. Zimmermann. Virtual reality — new methods for improving and accelerating vehicle development. In *Virtual Reality for Industrial Applications*, Fan Dai, Ed., Computer Graphics: Systems and Applications, chapter 6, pages 103–122. Springer, Berlin, Heidelberg, 1998.

[PS85] F. P. Preparata, and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[PS90] F. P. Preparata, and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 3rd ed., October 1990. ISBN 3-540-96131-3.

[PY90] M. S. Paterson, and F. F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete Comput. Geom.*, 5:485–503, 1990.

[RBH+95] S. Rezzonico, R. Boulic, Z. Huang, N. M. Thalmann, and D. Thalmann. Consistent grasping in virtual environments based on the interactive grasping automata. In Göbel [Göb95], pages 107–118. Proc's Eurographics Workshop, Barcelona, Spain, 1993, and Monte Carlo, Monaco, 1995.

[Red96] M. Reddy. SCROOGE: Perceptually-driven polygon reduction. *Computer Graphics Forum*, 15(4):191–203, 1996. ISSN 0167-7055.

[Rei88] M. Reichling. On the detection of a common intersection of k convex polyhedra. In *Computational Geometry and its Applications*, vol. 333 of *Lecture Notes Comput. Sci.*, pages 180–186. Springer-Verlag, 1988.

[Rei94] Dirk Reiners. High-quality realtime rendering for virtual environments. Master's thesis, TU Darmstadt, 1994.

[RG91] Hans Rijpkema, and Michael Girard. Computer animation of knowledge-based human grasping. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, Thomas W. Sederberg, Ed., vol. 25, pages 339–348, July 1991.

[RH92] Warren Robinett, and Richard Holloway. Implementation of flying, scaling, and grabbing in virtual worlds. In *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, David Zeltzer, Ed., vol. 25, pages 189–192, March 1992.

[RH94] John Rohlf, and James Helman. IRIS performer: A high performance multiprocessing toolkit for real–Time 3D graphics. In *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Andrew Glassner, Ed., Computer Graphics Proceedings, Annual Conference Series, pages 381–395. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[Rit90] Jack Ritter. An efficient bounding sphere. In *Graphics Gems*, Andrew S. Glassner, Ed., chapter V, pages 301–303. Academic Press, San Diego, CA, 1990.

[Ros93] Louis B. Rosenberg. The effect of interocular distance upon operator performance using steeoscopic displays to perform virtual depth tasks. In *IEEE Virtual Reality Annual International Symposium*, pages 27–32, September 18–22 1993.

[Ros97] Lawrence J. Rosenblum. Projects in VR: Applications of the Responsive Workbench. *IEEE Computer Graphics and Applications*, 17 (4):10, July/August 1997. CODEN ICGADZ. ISSN 0272-1716.

[SAK+95] Smith, Andrew, Kitamura, Yoshifumi, Takemura, Haruo, Kishino, and Fumio. A simple and efficient method for accurate collision among deformable polyhedral objects in arbitrary motion. In *Virtual Reality Annual International Symposium*, pages 36–145. North Carolina, USA, 1995.

[Sam90a] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990. ISBN 0-201-50300-X.

[Sam90b] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990. ISBN 0-201-50255-0.

[Sam90c] Hanan Samet. *Applications of Spatial Data Structures*. Addison-Wesley, Reading, Massachusetts, 1990.

[Sch96] Reiner Schäfer. Kopplung virtueller Umgebungen mit wissensbasierten Systemen. Diplomarbeit, Technische Hochschule Darmstadt, Fachbereich Informatik, June 1996.

[Sch98] Artur P. Schmidt. Physische Modelle aus dem Computer. Neue Zürcher Zeitung, 19. 8. 1998.

[Sei86] R. Seidel. *Output-size sensitive algorithms for constructive problems in computational geometry*. Ph.D. thesis, Dept. Comput. Sci., Cornell Univ., Ithaca, NY, 1986. Technical Report TR 86-784.

[Sei97] R. Seidel. Convex hull computations. In *Handbook of Discrete and Computational Geometry*, Jacob E. Goodman and Joseph O'Rourke, Eds., chapter 19, pages 361–376. CRC Press LLC, Boca Raton, FL, 1997.

[SF91] M. Shinya, and M.-C. Forgue. Interference detection through rasterization. *The Journal of Visualization and Computer Animation*, 2(4): 132–134, October–December 1991. CODEN JVCAEO. ISSN 1049-8907.

[SGS97] Xiao Yan Su, Leslie M. Goldschlager, and Bala Srinivasan. Integrating gestures into the user-interface management system. *The Visual Computer*, 13(1):168–183, 1997. ISSN 0178-2789.

[Sil92] SiliconGraphics. *Iris Inventor Programming Guide*, 1992.

[Slo98] N. J. A. Sloane. The sphere packing problem. In *DOCUMENTA MATHEMATICA III, Proceedings International Congress Math.*, pages 387–396. Berlin, 1998. URL `http://www.research.att.com/~njas/packings/`.

[SN93] Deyang Song, and Michael Norman. Nonlinear interactive motion control techniques for virtual space navigation. In *IEEE Virtual Reality Annual International Symposium*, pages 111–117, September 18–22 1993.

[Sny95] John M. Snyder. An interactive tool for placing curved surfaces without interpenetration. In *SIGGRAPH 95 Conference Proceedings*, Robert Cook, Ed., Annual Conference Series, pages 209–218. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

[SS98] Jörg Sauer, and Elmar Schömer. A constraint-based approach to rigid body dynamics for virtual reality applications. In *Proc. VRST '98*, pages 153–161. ACM, Taipei, Taiwan, November 1998.

[ST94] R. M. Sanso, and D. Thalmann. A hand control and automatic grasping system for synthetic actors. *Computer Graphics Forum*, 13 (3):167–177, 1994.

[Str64] Karl Strubecker. *Differentialgeometrie I + II*. Walter de Gryuter & Co., Berlin, 1964.

[Sun91] Kelvin Sung. A DDA octree traversal algorithm for ray tracing. In *Eurographics '91*, Werner Purgathofer, Ed., pages 73–85. North-Holland, September 1991.

[Sut65] I. E. Sutherland. The ultimate display. In *Proceedings of IFIPS Congress*, vol. 2, pages 506–508. New York City, NY, May 1965.

[Sut68] I. E. Sutherland. A head-mounted three-dimensional display. In *AFIPS Conference Proceedings*, vol. 33, pages 757–764, 1968.

[SW82] H.-W. Six, and D. Wood. Counting and reporting intersections of D-ranges. *IEEE Trans. Comput.*, C-31:181–187, 1982.

[SZ94] David J. Sturman, and David Zeltzer. A survey of glove-based input. *IEEE Computer Graphics and Applications*, 14(1):30–39, January 1994. CODEN ICGADZ. ISSN 0272-1716.

[Tho91] Spencer W. Thomas. Decomposing a matrix into simple transformations. In *Graphics Gems II*, James Arvo, Ed., pages 320–323. Academic Press, 1991. ISBN 0-12-064480-0. URL `ftp://ftp-graphics.stanford.edu/pub/Graphics/GraphicsGems/`.

[TKM84] M. Tamminen, O. Karonen, and M. Mantyla. Ray-casting and block model conversion using a spatial index. *Computer Aided Design*, 16: 203–208, July 1984.

[TN87a] W. C. Thibault, and B. F. Naylor. Set operations on polyhedra using binary space partitioning trees. *Comput. Graph.*, 21(4):153–162, 1987. Proc. SIGGRAPH '87.

[TN87b] William C. Thibault, and Bruce F. Naylor. Set operations on polyhedra using binary space partitioning trees. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, Maureen C. Stone, Ed., vol. 21, pages 153–162, July 1987.

[Tor90] Enric Torres. Optimization of the binary space partition algorithm (BSP) for the visualization of dynamic scenes. In *Eurographics '90*, C. E. Vandoni and D. A. Duce, Eds., pages 507–518. North-Holland, September 1990.

[Tou88] G. T. Toussaint. Some collision avoidance problems in the plane. In *Theoretical Foundations of Computer Graphics and CAD*, R. A. Earnshaw, Ed., vol. F40 of *NATO ASI*, pages 639–672. Springer-Verlag, Berlin, West Germany, 1988.

[TRC+93] Russell M. Taylor, II, Warren Robinett, Vernon L. Chi, Frederick P. Brooks, Jr., William V. Wright, R. Stanley Williams, and Eric J. Snyder. The Nanomanipulator: A virtual reality interface for a scanning tunnelling microscope. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, James T. Kajiya, Ed., vol. 27, pages 127–134, August 1993.

[TS84] Markku Tamminen, and Hanan Samet. Efficient octree conversion by connectivity labeling. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, Hank Christiansen, Ed., vol. 18, pages 43–51, July 1984.

[TS91] Seth J. Teller, and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, Thomas W. Sederberg, Ed., vol. 25, pages 61–69, July 1991.

[TS98] E. K. H. Tsang, and H. Sun. An efficient posture recognition method using fuzzy logic. *Virtual Reality*, 3:112–119, 1998.

[Tur89] Greg Turk. Interactive collision detection for molecular graphics. Master's thesis, University of North Carolina at Chapel Hill, 1989.

[Tur92] Greg Turk. Re-tiling polygonal surfaces. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, Edwin E. Catmull, Ed., vol. 26, pages 55–64, July 1992.

[Ull92] D. G. Ullman. *The Mechanical Design Process*. McGraw-Hill, 1992.

[Van91] G. Vanecek, Jr. Brep-index: a multidimensional space partitioning tree. *Internat. J. Comput. Geom. Appl.*, 1(3):243–261, 1991.

[VB92] Kaisa Väänänen, and Klaus Böhm. Gesture driven interaction as a human factor in virtual environments. In *Proc. Virtual Reality Systems*. University of London, May 1992.

[vdB99] Gino Johannes Apolonia van den Bergen. *Collision Detection in Interactive 3D Computer Animation*. PhD dissertation, Eindhoven University of Technology, 1999.

[VT94] Pascal Volino, and Nadia Magnenat Thalmann. Efficient self-collision detection on smoothly discretized surface animations using geometrical shape regularity. *Computer Graphics Forum*, 13(3): 155–166, 1994. Eurographics '94 Conference issue.

[Wel91] Emo Welzl. Smallest enclosing disks, balls and ellipsoids. Report B 91-09, Fachbereich Mathematik, Freie Universität Berlin, Berlin, Germany, 1991.

[WFB87] Andrew Witkin, Kurt Fleischer, and Alan Barr. *Topics in Physically-Based Modelling*, chapter Energy Constraints On Parameterized Models. ACM SIGGRAPH, 1987.

[WGS95] Qunjie Wang, Mark Green, and Chris Shaw. EM – an environment manager for building networked virtual environments. In *Proc. IEEE Virtual Reality Annual International Symposium*, 1995.

[WHG84] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1):52–69, January 1984.

[WLML99] A. Wilson, E. Larsen, Dinesh Manocha, and Ming C. Lin. Partitioning and handling massive models for interactive collision detection. In *Computer Graphics Forum, Eurographics'99*, vol. 18, pages 319–330. Blackwell Publishers, September 1999. ISSN 1067-7055.

[WMB98] G. Williams, I. E. McDowell, and M. T. Bolas. Human scale interaction for virtual model displays: A clear case for real tools. In *Proc. of The Engineering Reality of Virtual Reality*. SPIE, January 1998.

[WO90] Colin Ware, and Steven Osborne. Exploration and virtual camera control in virtual three dimensional environments. In *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, Rich Riesenfeld and Carlo Sequin, Eds., vol. 24, pages 175–183, March 1990.

[WO94] J.-R. Wu, and M. Ouhyoung. Reducing the latency in head-mounted displays by a novel prediction method using grey system theory. *Computer Graphics Forum*, 13(3):503–512, 1994. Eurographics '94 Conference issue.

[Woo70] W. A. Woods. Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10):591–606, October 1970. CODEN CACMA2. ISSN 0001-0782.

[WS94] John A. Waterworth, and Luis Serra. Vr management tools: Beyond spatial presence. In *Conference Companion CHI '94*, pages 319–320. Boston, Ma, April 1994.

[WS99] Jhn Weissmann, and Ralkf Salomon. Gestre recognition for virtual reality applications using data glves and neural networks. In *Proc. of the 1999 International Joint Conference on Neural Networks*. Washington, DC, July 1999. URL `http://www.virtex.com/applications/zurich_paper.pdf`.

[WSC+95] K. Y. Whang, J. W. Song, J. W. Chang, J. Y. Kim, W. S. Cho, C. M. Park, and I. Y. Song. Octree-R: An adaptible octree for efficient ray tracing. *IEEE Trans. Visual. and Comp. Graphics*, 1:343–349, 1995.

[Wu92] X. Wu. A linear time simple bounding volume algorithm. In *Graphics Gems III*, David Kirk, Ed., chapter VI, pages 301–306. Academic Press, San Diego, CA, 1992.

[YKFT84] K. Yamaguchi, T. L. Kunii, K. Fujimura, and H. Toriya. Octree-related data structures and algorithms. *IEEE Computer Graphics and Applications*, 3:53–59, January 1984.

[YW93] Ji-Hoon Youn, and K. Wohn. Realtime collision detection for virtual reality applications. In *IEEE Virtual Reality Annual International Symposium*, pages 415–421, September 18–22 1993.

[Zac94a] Peter Astheimer, Fan Dai, Martin Göbel, Rolf Kruse, Stefan Müller, and Gabriel Zachmann. Realism in virtual reality. In *Artificial Life and Virtual Reality*, Nadia Magnenat-Thalmann and Daniel Thalmann, Eds., pages 189–210. Wiley & Sons, 1994.

[Zac94b] Gabriel Zachmann. Precise and high-speed collision detection in interactive real-time visualization systems. Master thesis, Darmstadt University of Technology, Germany, Department of Computer Science, 1994. URL `ftp://ftp.igd.fhg.de/pub/doc/techreports/zach/colldet-thesis.ps.gz`.

[Zac95] Gabriel Zachmann. The BoxTree: Enabling real-time and exact collision detection of arbitrary polyhedra. In *Informal Proc. First Workshop on Simulation and Interaction in Virtual Environments, SIVE 95*, pages 104–112. University of Iowa, Iowa City, July 1995.

[Zac96] Gabriel Zachmann. A language for describing behavior of and interaction with virtual worlds. In *Proc. ACM Conf. VRST '96*. Hongkong, July 1996.

[Zac97a] Gabriel Zachmann. Distortion correction of magnetic fields for position tracking. In *Proc. Computer Graphics International (CGI '97)*. IEEE Computer Society Press, Hasselt/Diepenbeek, Belgium, June 1997.

[Zac97b] Gabriel Zachmann. Real-time and exact collision detection for interactive virtual prototyping. In *Proc. of the 1997 ASME Design Engineering Technical Conferences*. Sacramento, California, September 1997. Paper no. CIE-4306.

[Zac98a] Antonino Gomes de Sá, and Gabriel Zachmann. Integrating virtual reality for virtual prototyping. In *Proc. of the 1998 ASME Design Engineering Technical Conferences*. Atlanta, Georgia, September 1998. paper no. DETC98/CIE-5536.

[Zac98b] Gabriel Zachmann. Rapid collision detection by dynamically aligned DOP-trees. In *Proc. of IEEE Virtual Reality Annual International Symposium; VRAIS '98*. Atlanta, Georgia, March 1998.

[Zac98c] Gabriel Zachmann. VR techniques for industrial applications. In *Virtual Reality for Industrial Applications*, Fan Dai, Ed., chapter 1, pages 13–38. Springer, 1998.

[Zac99] Antonino Gomes de Sá, and Gabriel Zachmann. Virtual reality as a tool for verification of assembly and maintenance processes. *Computers & Graphics*, 23(3):389 –403, 1999.

[ZB94] Jianmin Zhao, and Norman I. Badler. Inverse kinematics positioning using nonlinear programming for highly articulated figures. *ACM Transactions on Graphics*, 13(4):315–336, 1994.

[ZHH96] Robert C. Zeleznik, Kenneth P. Herndon, and John F. Hughes. SKETCH: An interface for sketching 3D scenes. In *SIGGRAPH 96 Conference Proceedings*, Holly Rushmeier, Ed., Annual Conference Series, pages 163–170. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[Zie98] Rolf Ziegler. *System zum integrierten Einsatz von haptischen Displays in virtuellen Umgebungen*. Shaker-Verlag, 1998. Dissertation.

[Zik98] Karel Zikan. Communication by email, April 1998.

[ZLB+87] Thomas G. Zimmerman, Jaron Lanier, Chuck Blanchard, Steve Bryson, and Young Harvill. A hand gesture interface device. In *Proceedings of Human Factors in Computing Systems and Graphics Interface '87*, J. M. Carroll and P. P. Tanner, Eds., pages 189–192, April 1987.

[ZOMP93] Micheal J. Zyda, William D. Osborne, James G. Monahan, and David R. Pratt. Npsnet: Real-time vehicle collisions, explosions and terrain modifications. *The Journal of Visualization and Computer Animation*, 4(1):13–24, 1993.

[ZPR+98] P. Zimmermann, F Purschke, R. Rabätje, M. Schulze, M. Symietz, and O. Tegel. Virtual reality — Forschung and Anwendung bei Volkswagen, 1998.

This book was set in Palatino for the body text, Pazo Math for mathematics, and Optima Medium for the captions.
Typesetting was done with LaTeX/dvipdfm/dvips. One LaTeX/dvips run took 37 / 50 sec (user / real time) on an Onyx R10000 195 MHz (not counting the time needed to create the custom format file, with all files on a remote file server).
Programs used were LaTeX (`www.tug.org`), vim (`www.vim.org`), xfig (`www.xfig.org`), and gnuplot (`www.gnuplot.org`); the latter two with patches from the author.
Drawings and graphs were included from LaTeX using the EEPIC package (which issues TPIC macros).

An online version of this thesis (with color plots and fully hyperlinked) is available at `http://www.gab.cx`, or
`http://www.geocities.com/gabriel_zachmann/`.

The author can be reached (hopefully) at `Gabriel.Zachmann@gmx.net`.