Haptic Interaction with Three-Dimensional Bitmapped Virtual Environments

by

Jered Floyd

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1999

© Jered Floyd, MCMXCIX. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author . . . .
Department of Electrical Engineering and Computer Science
May 20, 1999

Certified by . . . .
Norman Margolus
Research Affiliate, MIT Artificial Intelligence Laboratory
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Haptic Interaction with Three-Dimensional Bitmapped Virtual Environments

by

Jered Floyd

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 1999, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I describe the design and implementation of a system integrating a force-reflecting haptic interface, the PHANToM, with a high-powered computer designed to efficiently compute cellular automata, the CAM-8. My goal was to build a system to allow users to interact with three-dimensional bitmapped simulations running on the CAM. Such simulations are far more computationally complex than can currently be computed on the PHANToM host. This system can also provide an intuitive user interface for manipulating data within CAM simulations. Methods for coping with network and computational latencies are described, and example applications are evaluated to explore the effectiveness of the system.

Thesis Supervisor: Norman Margolus
Title: Research Affiliate, MIT Artificial Intelligence Laboratory

Acknowledgments

I would like to expresss my gratitude to Norman Margolus; as my advisor he was always a source of creative ideas, made time to help whenever assistance was needed, and was willing to let me design and move this project in the direction that interested me most, even if that direction changed often. He was also a great help in writing the Forth code that controls the CAM-8.

I would also like to thank Brian Anthony and Mark Ottensmeyer from the Haptics Lab. Brian made space in the lab and found hardware for me to use, and both he and Mark were willing to help whenever I had questions about the PHANToM.

All of my friends at MIT, too many to name here, have provided everything from stimulating conversation to whimsical craziness during the five years of my studies, and have somehow managed to keep me optimistic, open-minded and sane.

Finally, I wish to thank my parents, who have supported me in all my endeavors no matter how complicated, were prescient enough to buy a home computer before most of the world knew they existed, and have always let me be myself.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Most computer-generated virtual environments are constructed from polygonal models of objects. This limits interaction with such systems to rules that may not accurately model a physical system. With advances in technology, however, it has become possible to compute three-dimensional bitmapped environments at rates fast enough for real-time simulations. Systems now exist that can enforce a set of rules over a large multi-dimensional space, primarily for examining physical simulations such as lattice-gas or hydrodynamic models. Interactive user interfaces for real-time manipulation of these new systems, however, have been lacking.

In this thesis, I describe the design and implementation of a system that utilizes a force-reflecting haptic interface to provide a three-dimensional haptic user interface for real-time simulations running on a cellular automata computer. A user places their finger in the haptic device, and is able to interact with objects in the virtual environment of the simulation. This provides a simple and intuitive interface for interacting with large bitmapped simulations.

Chapter 2 focuses on the design of a basic system for haptic interaction in a bitmapped environment. Chapter 3 explores coping with issues of network and computational latency when operating such a system. Chapter 4 evaluates the performance of the completed system in a number of sample applications. Finally, Chapter 5 reviews the initial design objectives and concerns, and examines future directions for associated research.

1.1 The CAM-8 Computer

The CAM-8 is a fine-grained parallel supercomputer designed for the efficient computation of cellular automata, developed by the Information Mechanics Group at the MIT Laboratory for Computer Science. It consists of an array of low-cost processors that subdivide an n-dimensional space. The CAM is designed to take advantage of good local connectivity in a uniform space to carry out certain kinds of large parallel computations. At its base, it provides a configurable n-dimensional cellular automaton which may be programmed with the desired transition rules.

Such an architecture is closely tied to the idea of modelling physical systems. Its simple, uniform layout provides insight into how in the future computations could be adapted to be performed by microscopic particles, based on fundamental spatially uniform interactions. Likewise, as a uniform architecture with configurable fundamental rules, the CAM is well suited for modeling physical interactions.

The CAM-8 has been extensively used for lattice-gas simulations, data visualization, hydrodynamic modelling, statistical mechanics modelling of processes like annealing and crystallization, and other simulations involving the processing of data in multi-dimensional bitmapped environments. To be sure, it is not limited to such simulations; as a general purpose CA machine it can efficiently run more abstract computations such as Conway's Game of Life, or digital logic simulations. The CAM is a powerful engine that provides high computing density at a comparatively low cost.

The CAM-8 connects to a host system via the CAMBus cable and protocol. At this time, there exists only a SparcStation SBus interface for CAMBus.

The CAM-8 is a remarkable machine that can easily perform real-time presentation and manipulation of three-dimensional bitmapped virtual environments; however, it suffers from the fact that there are no effective ways for a user to interact with the simulated environment. As a cellular automata machine, it is designed so that starting data are loaded into the cell matrix, and then the automata rules are repeatedly applied. The system can provide output data to a video display or back to the host computer, and can accept additional data from a video camera or the host computer, but there is currently no interface that allows a user to interact intuitively with objects within a simulated environment.

My goal in integrating a haptic interface with the CAM-8 is to have haptic interactions with the virtual environment simulated within the CAM. This allows for interfaces to CAM simulations that are easily understood and consistent with the way a user would expect the environment to react, based on their real-life haptic experience. Such a system transforms the CAM from an excellent computational engine to a powerful interactive tool.

The aim here is not to expand the existing technology to allow for new kinds of computational problems to be handled, but rather to make the existing computations more accessable to the user. Connecting a video monitor to a CAM-8 aids the understanding of a simulation by providing two-dimensional bitmapped output to the user; this project goes even further by providing a three-dimensional bitmapped bi-directional interface. Such a system should make understanding of and interaction with many simulations far more simple and intuitive than existing interfaces, and allow for a broad range of new applications.

1.2 The PHANToM Haptic Interface

The PHANToM haptic interface, originally developed by Thomas Massie at MIT, is a convenient, active device that senses the position of a user's fingertip, and can exert a precisely controlled force back upon the fingertip. It represents a substantial advance upon previous systems due to its size, simplicity, and accuracy. This device allows for high fidelity sensing and physical feedback to a user, and hence facilitates high resolution physical interactions with virtual environments.

The device is currently being used for a wide range of projects, including systems for computer-based training, medical simulation, and data visualization. It has been commercially developed by the MIT startup company SensAble Technologies, and is being successfully marketed while development on the technology continues. Most, if not all, current applications that use the PHANToM to interact with a virtual environment model their environment in much the same way as a 3-D graphical representation would be modelled; namely polygonal models of objects. On a standard workstation such as those the PHANToM is designed to interface with, a high-resolution bitmapped environment would quickly overwhelm the processing power and memory available. The CAM-8, however, with its high computing density is ideally suited for such bitmapped simulations.

The PHANToM haptic interface is able to connect to a host PC running Windows NT, or a Silicon Graphics system running Irix. Unfortunately, the PHANToM cannot be connected directly to a SparcStation, nor can the CAM-8 be connected to either a PC or SGI system. This restriction complicated my project somewhat, but also expanded its scope to include exploring topics of teleoperation in remote virtual environments.

Chapter 2

Basic System Design

I will describe basic haptic interactions with bitmapped environments in systems of increasing levels of complexity, starting from an example with the bitmapped world existing on the same machine as the PHANToM host and concluding with a client/server system with the bitmapped world existing on the CAM-8 computer. The goal is to design a system that will present the user with a realistic interface to the virtual environment. Interacting with the thimble-gimbal attachment of the PHANToM, then the user should ideally be presented with an experience indistinguishable from one where they are interacting with a real environment though a thimble. Force responses should be quick and accurate, and the tool with which the user interacts with the environment should not be larger or less manageable than a finger.

2.1 Basic Force-Reflecting Operation

The basic design of a haptic system is centered around a closed servo control loop that must operate at a very high rate. This basic haptic rendering loop consists of the following steps:

1. Retrieve user's finger position from haptic interface
2. Detect collisions of user with objects in virtual environment

3. Determine appropriate reaction forces for these collisions
4. Send forces to haptic interface for application to user

In traditional continuous environments, the most time-consuming steps are 2 and 3, detecting collisions and calculating restoring forces. To determine what objects a user has collided with in a continuous environment, it is necessary to calculate collisions with all objects in the vicinity of the user. In a bitmapped environment, however, step 2 is much simpler as it is only necessary to check whether the cells the user is in are occupied.

Hooke's law ($F = kx$) is traditionally used to calculate a restoring force after a user has intersected an object in the virtual environment. After the user has penetrated a surface in a continuous environment, the restoring force is calculated as if there were a spring between their fingertip and the surface of the object. This force is thus proportional to the depth the user has penetrated the virtual object, and perpenducular to the surface. This model works well when individual surfaces are penetrated, but the superposition of forces when multiple objects are penetrated can have unexpected results.

In a bitmapped environment, it is not possible to collide with multiple objects at the same point, and thus the superposition of forces is not a concern. Unfortunately, it is somewhat less trivial to determine the restoring force for having penetrated an object. Unless we track the user's path into occupied space, we cannot determine the surface normal or depth into the object. Instead, as the user moves about the environment, we can compute the location of a 'ghost' point, a point that unlike the user is strictly constrained by the surfaces of virtual objects [6]. After the user has penetrated an object in the environment, the restoring force can be computed as if there were a spring between the user's fingertip and the ghost point.

For an accurate haptic system that provides realistic sensations of touch, it has been found that the basic haptic rendering loop needs to run at a very high rate, at least 1 kHz. As the loop drops below this rate, the system provides a lower fidelity interface to the virtual envionment. Instead of objects that feel solid, with sharp and rapid force responses, objects become soft and unrecognizable. Lower loop rates can also lead to oscillations as the force responses become increasingly inaccurate for the current user position, and can even cause violent force responses. For these reasons, the PHANToM toolkit enforces a 1 kHz rate. If force responses are not sent at this rate, a watchdog timer disables the system. This provides only 1 ms for all computations to take place, and is the most significant limiting factor on the complexity of haptic simulations.

2.2 Single Host PHANToM Operation

The simplest system for working with haptics in a bitmapped virtual environment places the computation of the enviroment on the same computer that is hosting the haptic interface. The host computer is responsible for computing the physics for the entire virtual environment as well as the collision and force information for the user. Unlike algorithmically defined polygon environments, even very small bitmapped environments incur a huge computational cost. In an active environment transitions must be computed at each cell for every unit time. In a passive environment, any user interaction with an occupied cell could require a huge number of recalculations. This significantly limits the size of a bitmapped environment that can be simulated on a haptic interface host as the computational complexity of the environment will rapidly exceed the host's ability to compute resultant forces at the required 1 kHz rate.

2.3 Client/Server PHANToM Operation

One step further towards modelling the bitmapped virtual environment on the CAM is modelling the environment on a a remote server. This model of operation is much like the basic algorithm for haptic rendering, except that position information is sent from the PHANToM host to a seperate machine which computes the force vector to be returned. Ideally, this model is not significantly different that the previous one, however in the real world this can introduce significant communication delays into the crucial haptic loop.

2.4 PHANToM Operation with CAM Simulations

Finally, the virtual environment may be modelled on a specialized piece of hardware, such as the CAM. As a cellular automata based machine, the CAM is ideally suited for modelling bitmapped environments and can quickly compute physical laws across a large three-dimensional space. This builds upon the previous example because the CAM-8 cannot currently be hosted on the same machine as the PHANToM. The machine with the haptic interface must send the user position information to the machine hosting the CAM, which then communicates this information into the CAM simulation, extracts force information and returns this to the haptic interface host.

2.5 Concerns With This Approach

While in an ideal world this simple approach would work, the communcation and computation delays make it impossible to provide the 1 kHz haptic response rate required for a suitable user interface.

The faster the haptic rendering loop can be made, the more accurate the user experience will be; ideally it should remain purely on the client system. With current technologies it is not possible to have the closed haptic interface control loop extend beyond the PHANToM host machine and still maintain an adequate update rate. It was necessary to find a method to cope with the network and computational latencies.

Chapter 3

Network and Computational Latency

3.1 Client/Server Communication

As the PHANToM and the CAM-8 connect to different host computers, the two computers need to exchange information for the system to work. At the minimum for closed-loop operation, the PHANToM host needs to send position and perhaps velocity data to the CAM-8 host, and the CAM-8 host needs to return a force response. More complex system designs involve even more data interchange. Communication options were somewhat limited by the available hardware. Several possibilities were explored, including serial and Ethernet connections. Each communication method provided different amounts of bandwidth and latency.

3.1.1 Direct Serial Connection

The first attempt to connect the two machines so that they could communciate was simply a direct serial connection. The fastest serial rate that both systems supported was 38400 bps. This quickly proved to be inadequate. The position data from the haptic interface host consists of 3 IEEE floating point values, one for each of the dimensions. This is 12 bytes, or 96 bits, per sample not including framing data. At 38400 bps, this requires a full 2.5 ms just to be sent over the wire. Given an equal return delay for force data, the communication latency of a direct serial connection is 5 ms round trip, and the computational latency of the server must then be added on top of this. Clearly, this is completely impractical for a 1 kHz control loop, for which it would be necessary for communications and computation to be completed within 1 ms.

3.1.2 Ethernet Connection

The other communication interface that both systems share is a 10 Mb/s Ethernet connection. Given a completely quiescent network, one could expect the on the wire delay of a 12 byte data sample to be well under a millisecond, naively as short as 10 microseconds. Such a communication latency has negligible impact on the haptic control loop.

In practice, however, the latency over an Ethernet connection is much higher than 10 microseconds. Although we can arrange to have a quiescent network by connecting the two systems to a private Ethernet, the complexity of the Ethernet and IP protocols introduces considerable latency. Experimentation showed that the round trip latency over a UDP connection should be expected to be as high as 1 ms. Thus Ethernet is also unacceptable for closed-loop operation.

3.1.3 Other Communication Options

Additional hardware would likely have afforded lower communication latency. A high-speed point-to-point link such as HIPPI or FireWire (IEEE 1394) could potentially provide delays small enough such that the client/server communcation would not itself hinder closed loop operation.

There are a number of reasons that I chose not to explore such other options. Specifically, any of these options would require further specialized hardware for both systems, and time spent on that hardware would have been better spent on a CAM-8 interface for the PC or a PHANToM interface for the SparcStation. Additionally, working with Ethernet as the communications medium offers a broader horizon of future applications. Examining methods of using haptics in bitmapped environments over a network leads directly to the possibility of using haptics in shared or distributed virtual environments. Most importantly, the computational latency introduced by the CAM outweighs the network latency introduced by Ethernet when interacting with moderately large environments. At the level of delay due to UDP/IP networking the communcation latency was no longer the limiting factor to closed-loop operation.

3.2 Stability and Response Issues

Before exploring more complicated options, I first examined the possibility of standard closed loop operation at a lower response rate. Given Ethernet communications and the speed of the CAM-8, a loop rate of 100 Hz would not be an unfeasable goal.

To determine whether or not this was a feasable direction to continue working towards, I reimplemented the simple “Hello, sphere” program that was included with the PHANToM’s SDK. This program presents a simple virtual evironment with no visual feeback. Only two primitives exist in this environment, a sphere and a floor. I modified this example so that the force computation took place on the server system. The haptic interface host sent position data at 1 kHz to the server, which would then compute the resultant force on the user’s finger and send this data back to the haptic interface host. Computation time for such a simple environment was negligible, so this served to test whether or not the network delay was acceptable for closed loop operation.

Testing revealed two serious problems with this model of operation. First, the environment presented to the user is of very low fidelity. The stiffness of items in the enviroment is determined by the stiffness of the haptic interface and the stiffness of the closed servo loop, of which the latter is the limiting factor with the PHANToM [5]. With the communication delays present on an otherwise quiescent Ethernet, the closed loop delay is long enough that objects in the world have unacceptably low stiffness, making them virtually unidentifiable.

More seriously, the network delay introduced distracting and sometimes violent instabilities into the force feedback response. The delay between the time at which the user's position is sampled and the time when the computed force is delivered is large enough such that the force may be completely incorrect from the force that should be delivered to the user at the current time. From this effect, induced oscillations have been encountered in force feedback teloperation systems and have been addressed in a number of different ways that I explored as possible solutions. Several of these methods for coping with delay-induced instabilities are summarized below.

It is evident that simple closed-loop operation is not possible with this model of haptic interaction with bitmapped environments, making it necessary to find a method to cope with the network and computational latencies while presenting a realistic haptic experience for the user.

3.3 Teleoperation

The same problems that I have encountered with communication and computational latency have previously been seen when using force feedback devices for teleoperation. The computational delays are essentially no different from additional communication delays, so the whole system can be likened to time-delayed teleoperation in a virtual environment. Existing systems have been designed to cope with delays at least as large as 1 s [8], which is far greater than my expected system latency.

The term teleoperation is used to describe systems involving two distant, coupled robots. A human operator moves the master robot, and this motion is measured and transmitted to the remote slave system. The slave system then attempts to mimic the master motion in its environment. Some level of feedback, such as video, is provided to the human operator.

Force feedback is of particular interest in teleoperation as it allows the two robots to share forces in addition to motion [8]. This allows for much more intricate interaction with the remote environment, and in ideal cases allows the human operator to act as if they were themselves at the remote location.

As with delayed operation in virtual environments, time delays in teleoperation significantly impact the sense of reality that is presented to the user. In traditional teleoperation without haptic feedback, this is at worst a distraction and a hinderance to normal operation. When an active force feedback component is added, however, the closed feedback loop that is created can become dangerous and cause the entire system to become to unstable for normal operation. Such violent oscillations that I encountered in my simple assessment of network delay are common in uncompensated time-delayed teleoperation.

For this reason many thought that haptic feedback was not suitable for time-delayed systems [8]. Work in the early 1990s, however, demonstrated that it was possible to integrate haptic feedback in a stable manner. As interacting in time-delayed computer simulations is functionally quite similar to interacting with time-delayed remote environments, I evaluated methods used to manage stability in teleoperation systems to see if they were suitable for my project.

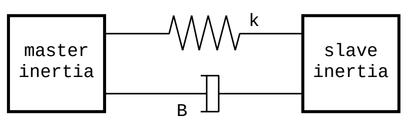

Figure 3-1: Teleoperation PD Model

3.4 Basic Force-Reflecting Teleoperation

A basic undelayed teleoperator models the master and slave devices as two inertial objects connected by a spring and damper system, in essence a P.D. (Proportional Differential) controller. (Figure 3-1) Such a system clearly consumes more power than it produces, and so intuitively is resistant to violent undamped oscillations.

Adding the time-delay to this system means that we can no longer compute

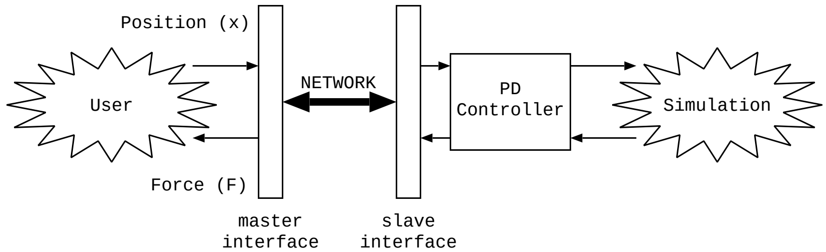

Figure 3-2: Basic Force-Reflecting Teleoperation

and apply the forces at the master and slave systems simultaneously. Instead we must either compute forces at one of the endpoints and send this to the other, or compute forces at both endpoints and only share position and velocity data. The first method is usually chosen as both have been shown to have similar stability limits and the former has been found to be more natural to operate. (Figure 3-2)

This method of coping with delay in a teleoperation system is directly applicable to coping with delay in a bitmapped virtual environment. As a teleoperation master, the PHANToM has extremely low inertia (apparent mass $< 100\,g$), and as a slave the projected endpoint in the virtual environment has no inertia whatsoever. The basic closed-loop system for haptics in bitmapped remote environments could easily be extended to give both master and slave additional effective mass. The actual user position and projection of this point into the virtual environment could be coupled in the computer by the P.D. controller. This solution would certainly improve the erratic operation of the naively designed basic system, however it is not without its disadvantages.

Even this system is not immune to instability, although with sufficient damping, this sort of system can be made stable when the time delay is limited to some upper bound. This is a reasonable restriction for operation on an isolated network, although if future work involved machines seperated by the public Internet, this restriction may be excessively limiting.

More problematic is that this solution makes the user excessively aware of the fact that the system is working around the teleoperation delay. Both master and slave are imbued with a potentially large inertia that may be completely inappropriate for the simulation at hand. Operating a time delayed system in this manner makes interaction with simulations as difficult as handling large and heavy objects. While this is not a task outside the capabilities of the user, this is not the sort of interface that is my goal. The system should attempt to present to the user an interface modelled as closely as feasable to undelayed operation.

Another serious problem is that this system is based on the assumption that the environment is passive. This is understandably necessary to prevent energy introduced by the environment from driving the feedback loop to instability, however it is extremely limiting to the scope of simulations that the user may interact with. Given the computational power available on a device such as the CAM and the goal of interacting with physical simulations, forcing these simulations to be passive to account for network and computational delay is not an acceptable solution.

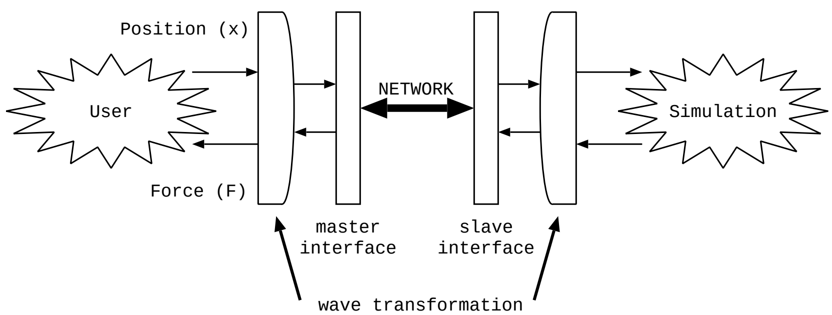

Figure 3-3: Wave Based Force-Reflecting Teleoperation

3.5 Wave-Based Teleoperation

In 1996, Gunter Niemeyer extended the traditional teleoperation model with the introduction of wave variables, single variable combinations of both velocity and force data. (Figure 3-3) Encoding the data in this wave variable form makes a standard teleoperation system impervious to instabilities due to communications delay. For large delays, oscillations may result, however these may be accounted for with methods including wave impedance matching, wave filtering, and wave controllers.

Unlike conventional teleoperation, wave-based teleoperation allows for easy tuning of the perceived virtual tool with which the user interacts with the remote environment. The fidelity of interaction is, of course, limited by the closed-loop bandwidth of the system, but this can be hidden by altering the characteristics of the virtual tool. Under a lightly delayed system, the user might interact with the environment with a light pencil, or their finger. As the delay increases, however, the virtual tool becomes heavier or softer, reducing the bandwidth requirements of the controller loop. Wave variables allow for easy adjustment of how the virtual tool under a large delay is perceived, at the extremes a large heavy drill or a soft sponge, or somewhere in between.

Wave based teleoperation has the advantage of increased stability over conventional teleoperation. It also provides a method of tuning the trade-off of clarity of haptic feedback with speed of operation, which is less readily determined by the adjustible components of conventional teleoperation. For the purposes of operating in a virtual environment, however, it has the same failures as conventional teleoperation.

While wave based teleoperation provides a level of adjustment in the interface with which the user interacts with the environment, it still presents a very different interface than undelayed operation. For the sort of simulations that I would like to interact with on the remote system, this is unfortunate. This method adds quite a bit of complexity over conventional teleoperation, with not much in the way of gain for my system. Wave-based teleoperation is also like conventional teleoperation in that it assumes a passive environment, which is not an acceptable restriction.

3.6 Teleoperation with Buffer Zones

All of the methods described above for coping with time-delayed teleoperation place cumbersome restrictions upon the operator and the environment. Much of this is due to the fact that these methods were designed to improve the use of teleoperation in real environments. These methods all assume an unknown and passive environment. Neither of these assumptions is necessarily true, one reason why none of these methods are deemed acceptable for this system. When teleoperating in a virtual environment, the entire state of the environment is necessarily known to the remote system. This opens up a huge new realm of options for coping with delay. Hopefully these options will also provide a solution which does not require a passive environment.

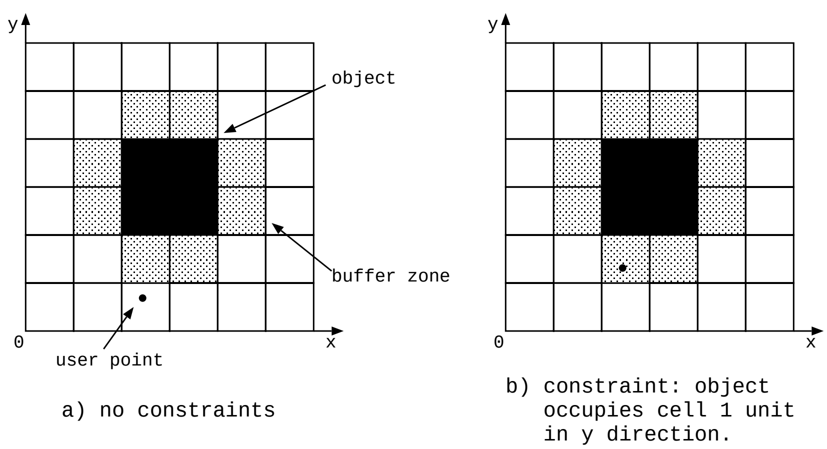

Figure 3-4: Teleoperation with Buffer Zones

The first obvious new method is to provide the master system with advanced warning when the teleoperation slave is in danger of a collision in the virtual environment. In a bitmapped virtual environment, this can be done by estabilishing a buffer zone around occupied cells in the environment, as in Figure 3-4a. The size of this buffer zone is dependent on the maximum expected velocity of the operator and the teleoperation delay; it should be large enough that the largest distance travelled by the user during one round-trip delay does not exceed the size of the buffer.

This can easily be modelled as an intangible skin around all objects in the virtual environment. While in free space the user moves about unhindered as the client sends position updates to the computation server. When the user penetrates one of the buffer cells, the server warns the client of the potentially imminent collision by sending a constraint to the client indicating the direction and distance to the occupied cell. (Figure 3-4b) The client may then monitor the user for violation of this constraint, and immediately provide a force response without having to wait an additional round-trip delay for confirmation of the collision. As the user moves from cell to cell in the environment, motion constraints are updated by the server. In a bitmapped world, the client is at most responsible for six user constraints at any given time, one for each of the six faces of the open cell the user is currently in.

This solution is substantially better than the methods described above for coping with delay, as it provides a much more accurate and realistic response to the user and is not limited to passive environments. It differs notably from all other teleoperation methods in that the closed servo loop no longer encompasses the entire system, instead it runs entirely on the local machine. The closed haptic loop may run at a 1 kHz rate to provide stiffness and resistance to instability. The user is essentially operating in a frequently updated local model of the remote environment in which they are interacting.

This method is not without its problems, however. The simple six-constraint model does not easily account for cells diagonal to occupied cells, only those orthogonal to the faces of occupied cells. (Figure 3-5a) Without some changes, the user can still get arbitrarily close to an occupied cell without the client receiving any advanced warning, meaning a user could significantly penetrate a surface in the environment before the client knows to provide a force response. This can be worked around by having constraints identify what nearby cell is occupied as opposed to direction and distance, but this adds further complexity. (Figure 3-5b)

When buffer zones extend beyond one unit away from occupied cells, they may

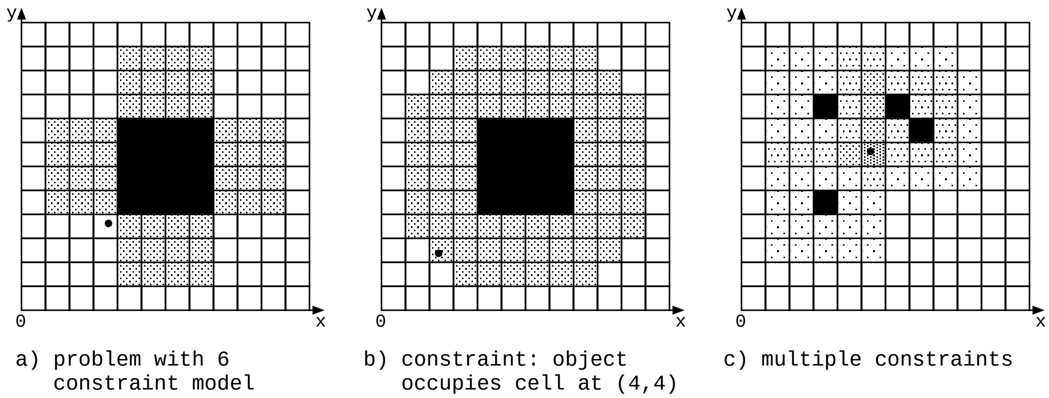

Figure 3-5: Problems with Buffer Zones

begin to overlap and provide additional constraints for the client to keep track of. (Figure 3-5c) In a system with substantial delay, the computational load on the client may become excessive.

This system cannot gracefully deal with variable delay, as wave-based teleoperation does. The upper limit of the delay it can handle is set by the size of the buffer zones around occupied cells, which are not easily modified while the simulation is running.

Additionally, this requires significant modifications to existing simluations, forcing the addition of the buffer zones. This may be a significant computational load upon active evironments, and may consume valuable data bits on the bitmapped computer.

Finally, while this may provide a reasonable sense of reality for passive, immobile environments, it is unclear whether mobile objects may overreact to user contact when pushed. The sudden change from penetrating a virtual surface to being in a buffer zone as an object slides from cell to cell could provide a ratchet-like feel to the entire environment.

For these and other reasons, buffer zones are not suitable for haptic interaction in remote bitmapped virtual environments. This method provides a number of advantages over the conventional teleoperation methods, however it introduces a host of new problems that are not easily resolved.

3.7 Time/Space Tradeoff in Virtual Environments

None of the methods described thus far for coping with latency are sufficient to meet the system design goals. To meet these goals, it was necessary to formulate a new method for coping with latency in virtual teleoperation. For the kind of high-fidelity force response desired, it is clear that the closed haptic response loop should not extend beyond the haptic interface host, and should be designed to operate as quickly as possible based on a local model of the remote virtual environment. As we relaxed some of the environment requirements, as with buffer zones, performance improved significantly. By relaxing environmental restrictions further, we are able to find an acceptable solution to the latency problem.

All of the methods described above make a fundamental assumption about the nature of the system: There is a direct spatial correspondence between the local and remote environments. That is to say, if there is a wall five inches from the neutral starting position in the remote environment, then the user will perceive it five inches from the neutral starting position with the haptic interface. Fixed objects in the remote environment are also fixed in a frame of reference about the haptic interface.

This assumption is understandable when teleoperating in a remote real environment. The slave robot physically exists in that remote environment, and when it collides with an object it encounters natural restoring forces that should be communicated back to the master. This is not, however, the case when operating in a virtual environment. The slave robot exists only inside the computer, and does not necessarily encounter forces unless we choose to compute them, and these forces can be computed without affecting the state of the environment. It is free to penetrate solid objects and violate the physics of the virtual universe if we so desire. In my research, I did not find any other similar systems that took advance of these properties.

This understanding allows us to make a simple trade-off of temporal versus spatial accuracy. We are able to provide a high-fidelity haptic response at the exact time the user penetrates an object in the virtual environment by relaxing the spatial correspondence between the haptic interface workspace and the virtual environment.

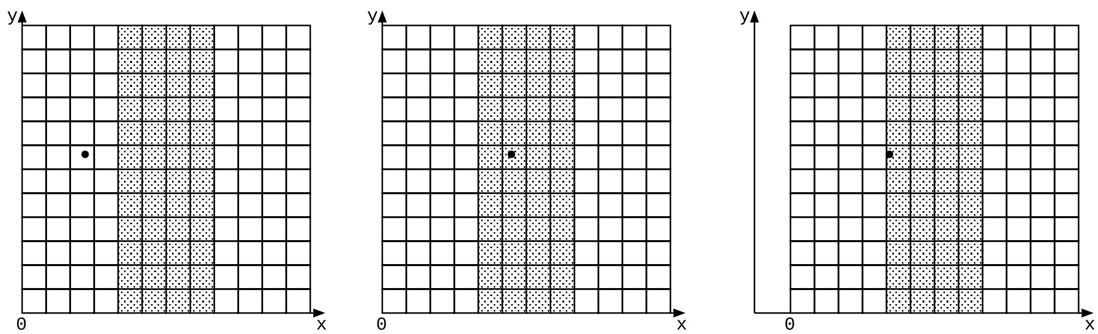

Figure 3-6: Time / Space Tradeoff

A simple example: Consider that the user is moving unimpeded through free space, and there exists a solid surface some distance in front of the user. (Figure 3-6a) As the user moves towards the surface, at some point they will penetrate the surface given the current frame of reference of the haptic interface. (Figure 3-6b) Some delay length later, the computation server will realize that the user has violated physical constraints of the environment. In a traditional teleoperation system, it would then compute a force to deliver to the user which by the time it was delivered would be incorrect, as the user may have moved significantly during the time it took to compute and communicate this force. Instead, the server tells the haptic client that the user has penetrated a surface in the environment, and where that collision occured. The client uses this information to offset the coordinate system the user is operating in so that instead of having significantly penetrated the surface the user is merely just within it, computes an appropriate force response, and caches the constraint implicit in the existance of that surface so that forces to impede further progress in that direction are computed on the client alone. (Figure 3-6c)

In other words, when a user collides with an object in a remote virtual environment, instead of immediately encountering a restoring force they frictionlessly push the coordinate system (and all objects therein) away from them for the duration of one round-trip delay. After this delay, the haptic client has learned of the environmental constraints and can provide force feedback as appropriate.

This solution builds upon the gains that teleoperation with buffer zones provides, and remedies many of the problems with that method. It provides for accurate and clear response to the user, and is not limited to passive environments. The closed haptic response loop is confined to the local machine, and needs to compute relatively few potential forces, so is free to run at a rate sufficient to provide a high level of stiffness and resistance to instability.

Unlike buffer zones, there is no need to worry about sneaking up on a surface: penetrating a surface with little to no advanced warning because of variable delay, insufficient buffer zone, or diagonal approach. The user's position is not of concern until after they have collided with an object, and so there is no need for advanced warning. There is no need to modify simulations to allow for buffer zones. All of the coordinate system offsetting can be done in the software handling communications between the haptic client and the computation server. As there are no buffer zones, there is no need to worry about overlapping zones and excessive multiple constraints, and so in a discrete environment the client is limited to three simultaneous client constraints.

The accuracy of the spatial correspondence between the haptic workspace and the virtual environment depends on the communication and computational delays, and thus, as with wave variables, variable delay may be handled transparently. A larger delay simply results in larger spatial offsets.

This method does introduce some new concerns. For example, if a user repeatedly poked at a virtual surface, each collision would result in the coordinate system being offset slightly further in that direction. It is feasible that the user would be able to push the entire active environment outside of the haptic workspace. There is a simple solution to this problem; simply have the coordinate system of the haptic workspace tend to return to its original starting position. This can be implemented by modelling the effects of a spring between the current coordinate space origin and its starting position. Such a constraint can easily be figured in to the force response on the haptic client.

This is still a method for coping with the fact that we have significant communication and computational latency; it does not mitigate the fact that we have limited bandwidth between the haptic client and the remote virtual environment. It does, however, give us a higher level of haptic accuracy – timely and sharp responses to object collisions, in exchange for inaccuracy as to where these collisions occur in the haptic workspace.

This is an acceptable trade-off. If the haptic response is inaccurate, this is eminently noticable to the user. Objects feel soft or unwieldy, and in worst cases violent oscillations and instabilities may result. The result of spatial inaccuracy, however, is far less noticable to the user of a haptic interface. As the user is operating the haptic interface in free space, they have very little in the way of a spatial frame of reference beyond their own proprioception. For small coordinate offsets, the effect is not easily perceptible to the user.

The previous assertion should be qualified somewhat. The result of spatial inaccuracy is far less noticable than haptic inaccuracy *as long as other forms of feedback match the haptic feedback.* It has been shown that when a system provides visual and haptic feedback, the visual feedback far outweighs the value of haptic feedback to the user. Thus, in a simulation providing visual feedback that indicates the user's current position, the displayed position should correspond to the user's position of one time-delay prior, so as to match their haptic feedback.

In the system as described, the client is responsible for the computation of the restoring forces to be presented to the user. When interacting with simulations where the cells that make up the environment are rather large, this would appear to require the objects in the world to have some homogeneous set of surface properties. For such small simulations, this method for coping with delay is limiting and primarily valid for rigid objects. The system could be extended such that the server provides surface properties that correspond with each directional constraint, if the computation server keeps track of such information.

As the cells of the simulation shrink to sizes we are more likely to encounter with large bitmapped environments, however, I theorize that instead of approximating material interaction by utilizing a set of surface characteristics, the physics of the environment can create realistic force responses for material properties. Each individual cell will still act as a rigid body, however the way in which these cells shift about in the simulation due to physics will give a realistic material response to the user.

Current haptic systems compute restoring forces as if the user had penetrated some hollow object and a mechanical system is trying to return them to the surface. Bitmapped virtual environments should allow restoring forces to be based on the physics of a finite element analysis of the actual solid objects.

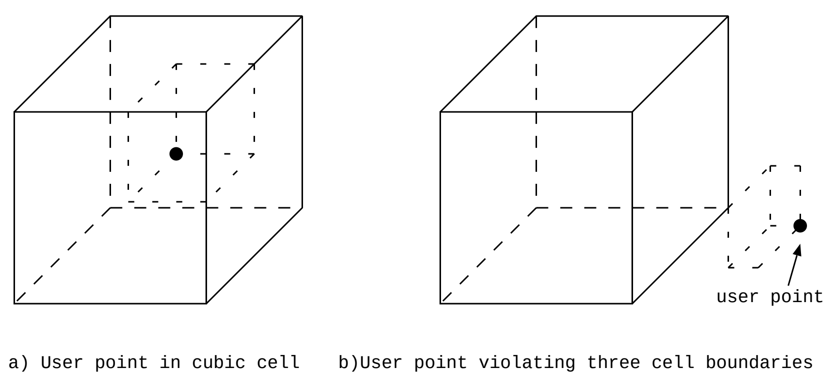

Figure 3-7: User constraints in a discrete environment

3.7.1 Concerns in Discrete Environments

There are a very limited number of user constraints when operating in a discrete environment, such as a bitmapped virtual environment, as the user is always within a cubic cell with six boundaries. At any given time, the user could potentially violate at most three of these boundaries, and thus there are at most three constraints on the user at any given time, as shown in Figure 3-7. The exact methods for handling these constraints as the user moves from cell to cell are less than trivial.

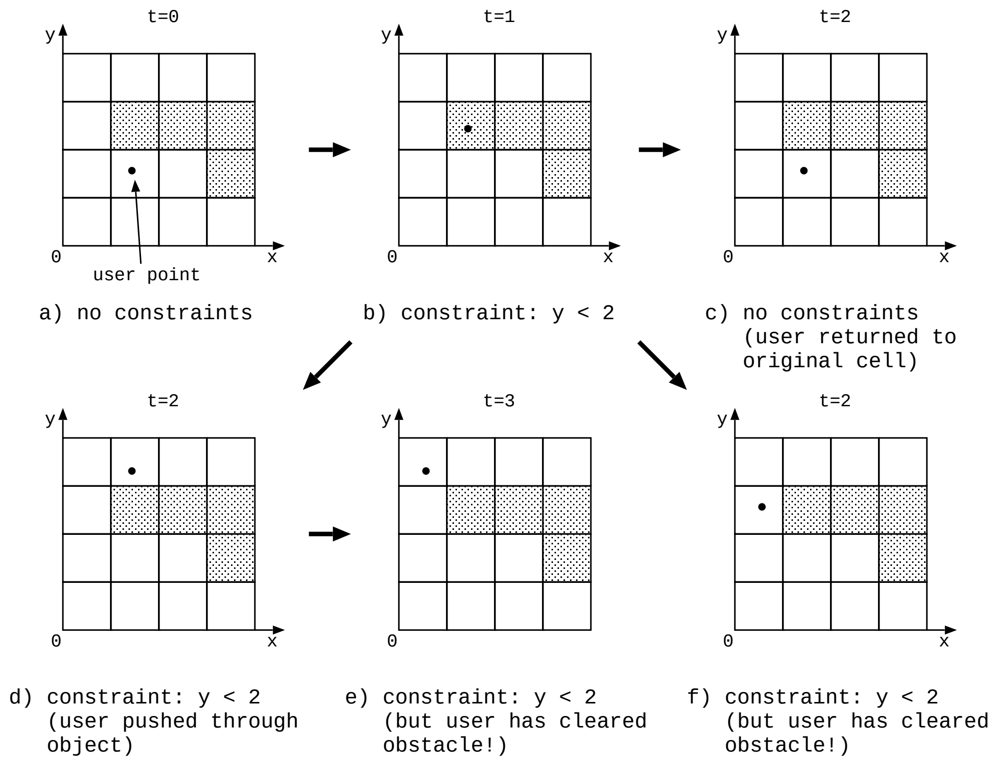

Figure 3-8: Behavior in a simple constraint model

The simplest case is when the user is in a free (unoccupied) cell. In this case, there are no constraints on the user. (Figure 3-8a) Now, suppose the user moves from a free cell into an occupied cell. In two dimensions, assume the user has moved from a cell based at (1, 1) into an occupied cell at (1, 2). This sets the constraint $y < 2$. (Figure 3-8b) Similar constraints would be set moving from a free cell into an occupied cell in any dimension.

Now imagine that the user returns to cell (1, 1) on their own accord, or due to the restoring force. The constraint $y < 2$ is no longer violated, and so there are no constraints on the user. (Figure 3-8c)

This, however, is not the end of the story. As the user is not interacting with actual objects but instead with a force feedback device of limited response power and stiffness, they may pass entirely through virtual objects in the environment. Consider the case where the user has encountered the constraint $y < 2$, and then pushes on to the cell (1, 3). (Figure 3-8d) This cell is not occupied; the user has completely penetrated the virtual surface. Although the cell (1, 3) is unoccupied, the constraint $y < 2$ is still valid for the user.

One way of implemeting this would be to not relax constraints until they have been satisfied; that is to say, enforce the constraint $y < 2$ until the user has returned to a cell with a y coordinate less than 2. This is not entirely valid. Consider the examples in figures 3-8e and 3-8f. In each of these cases, there should be no further constraints on the user because they have cleared the obstacle by motion in other dimensions, however with our naive implementation above, there would still be the now bogus y constraint.

A solution to this problem is to use the sort of constraint-based God-objects that Zilles and Salisbury described in 1995 [16]. Instead of just tracking constraints and the current user position, we also keep track of where the user would actually be if they were not able to penetrate virtual objects, and calculate constraints for restoring forces to that point. This God-object, or 'ghost point' as Massie calls it, is *strictly* constrained by the calculated constraints.

As the user moves in free space, their ghost point tracks their actual position exactly. When they penetrate an object, however, it remains strictly constrainted to the surface. As user motion drags the ghost point around in unconstrained directions, user constraints are re-evaluated whenever the ghost point enters a new cell. In figure 3-9c, for instance, as soon as the user moves to cell (0, 3), the ghost point is dragged to cell (0, 0), constraints are re-evaluated, and the $y < 1$ constraint is lifted. Figure 3-9 demonstrates the effectiveness of this method in two dimensions. Is it trivial to extend this to three (or more) dimensions.

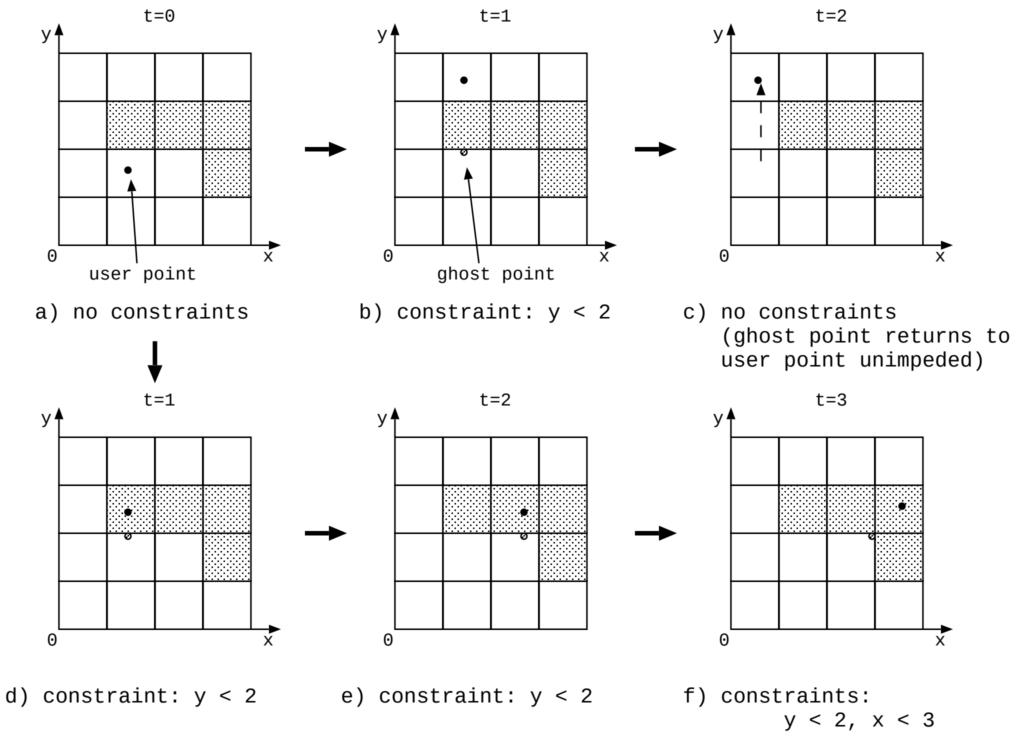

Figure 3-9: Behavior in a ghost point constraint model

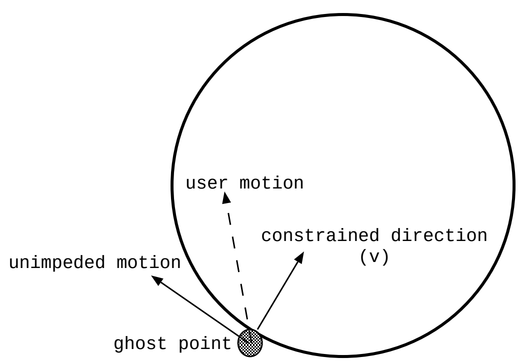

Figure 3-10: User constraints in a continuous environment

3.7.2 Continuous Environments

Though not applicable to bitmapped virtual environments, I find it useful to point out that this sort of constraint-based teleoperation is also applicable to traditional continuous environments. Upon intersecting a virtual object, constraints can be set to impede user progress in the $\vec{v}$ direction that is the inverse surface normal of the penetrated object, as in Figure 3-10. On the server, the calculations necessary are essentially those of a local simulation with constraint-based ghost points, so this is only useful for a particularly powerful remote compute server for physics in an active environment, or for shared virtual environments. The number of constraints is not limited to the three provided in discrete environments, and so communication and calculation requirements may go up dramatically in such a system.

Chapter 4

Application Adaptation and Results

4.1 Hello, voxel!

To validate the method for coping with computational and network latency described in the previous chapter, I first implemented a simple client / server haptics system with a three-dimensional bitmapped environment computed on the remote SparcStation, without using the CAM. I constructed a remote environment consisting of 3 by 3 by 3 cells, each 30 mm on a side. This fits nicely within the 5” cube within the PHANToM’s workspace, and allows for some additional buffer space for offsets.

This example is very much like Thomas Massie’s first ’box’ example program, implemented in a discrete space as opposed to a continuous one [5]. All cells except the center cell are occupied; the user starts in the center cell. Exploring this environment, the user should get the impression that their finger is confined to the space within a small cubic box. To explore the effects of larger latencies, the main loop of the server was designed to allow an arbitrary delay to be added to the computations.

I explored interacting with the simulation at various computation delays. The average spatial displacement upon encountering obstacles, and the computed average velocity based on this, are detailed in Table 4.1 for a number of latencies. At the observed average velocity of 0.2 m / s, a delay of 200 ms represents the upper

Table 4.1: Spatial displacement observed for varying latencies

Approx. Latency	Update Rate	Avg. Spatial Offset	Average Velocity
1 ms	(1000 Hz)	<0.5 mm	
20 ms	(50 Hz)	4 mm	0.20 m/s
50 ms	(20 Hz)	12 mm	0.24 m/s
100 ms	(10 Hz)	23 mm	0.23 m/s
200 ms	(5 Hz)	40 mm	0.20 m/s
500 ms	(2 Hz)	(full range)	

end of the range where this method is reasonable for interacting with remote simulations. A larger workspace and slower user movements could conceivably make this method usable for up to one second delays.

At low latencies, up to approximately 50 ms, the user experience is very sharp and precise. It is difficult, if not impossible, to tell that there is a significant delay in the computation of new user constraints. As the latency increases, the workspace and objects within it appear to stretch out over more space, particularly when the user makes quick motions. Objects in the environment remain in the same locations with respect to one another, however bouncing quickly between two opposite sides of the box may make the box feel larger than it is in the remote environment.

I extended this simulation to allow for larger remote environments, and found, as expected, that performance did not degrade for static environments as large as 64 by 64 by 64 cells. The SparcStation server does not have the computational power to update 256,000 cells per cycle at any appreciable rate, so this system is restricted to purely static simulations where the user is the only moving object.

4.2 CAM-based Simulations

4.2.1 Static Pieces

Having verified that the system as designed could cope with quite large computational latencies, I moved on to have the virtual environments simulated on the

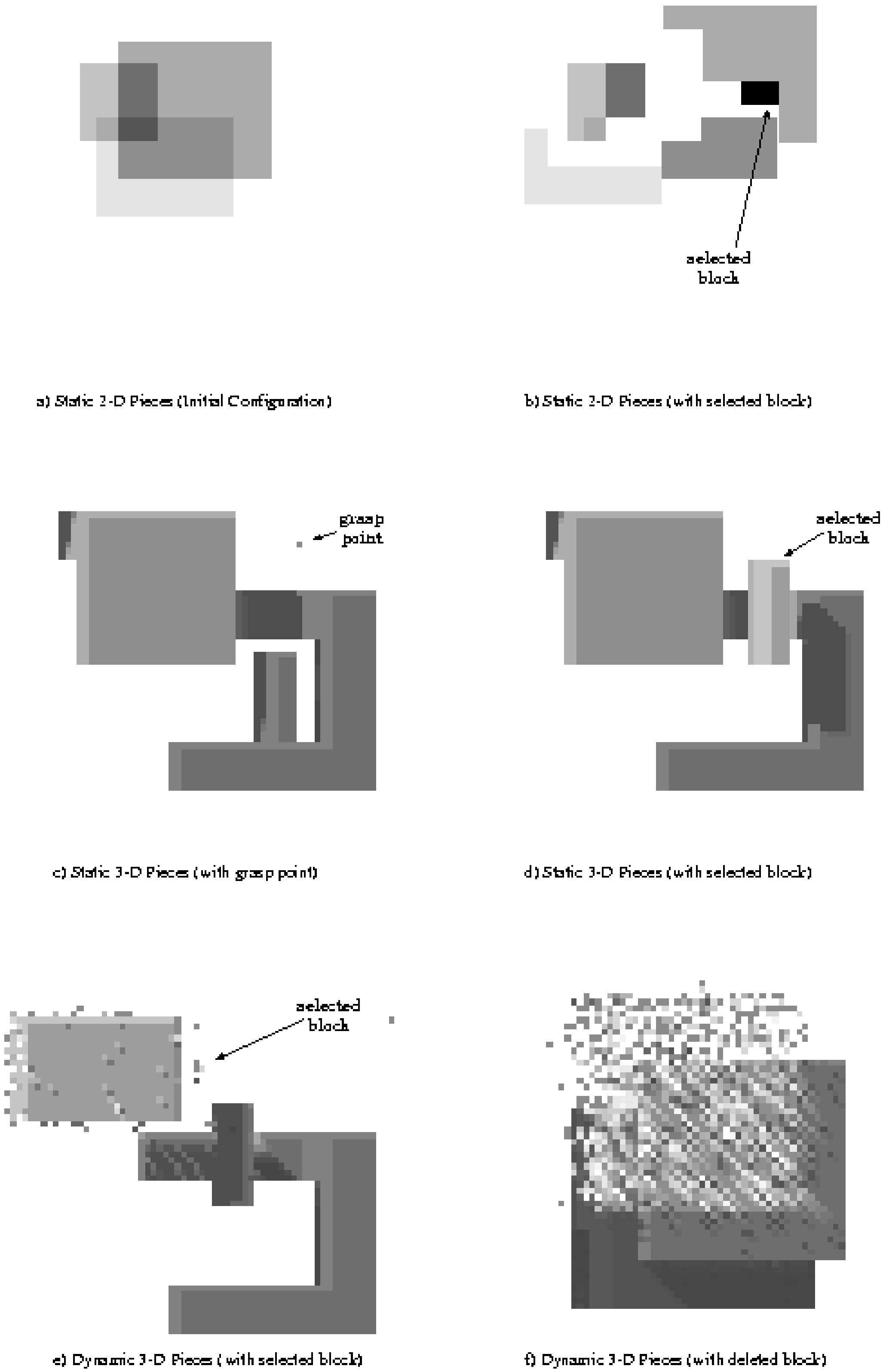

Figure 4-1: Images of CAM Simulations

CAM-8 instead of the SparcStation. The first CAM-based simulation was a modification of an existing demo application.

The original application consists of a 512 by 512 by 1 space in which there exist up to 63 solid pieces. At any given time, one of these pieces may be selected, and moved in any of the eight cardinal directions. Whenever there is an overlap of the selected piece with any of the other pieces, the number of overlapped cells is noted. This simulation is controlled by an external C program that attempts to randomly move the pieces apart until they can all be moved independently.

To adapt this for use as a haptics simulation, it was necessary to make a number of changes to the application. Ideally the server would obtain the new user position from the client, and ask the CAM to attempt to move the piece to the new position. The server would then return new boundaries based on whether or not it was possible to make the desired move. Unfortunately, this would not allow for ghost point tracking as described in Chapter 3, and so the user could potentially tunnel through small objects, or have difficulty sliding along surfaces. Instead, the server software keeps track of the location of the user ghost point, and asks the CAM to attempt to move the current selected piece to the new position. The CAM steps a cell at a time, moving the selected piece and checking for overlap with other pieces. The CAM returns to the server the actual distance moved, from which boundaries can be computed and returned to the client.

In the completed simulation, the PHANToM endpoint exists as a single cell 'grasp point'. This grasp point attempts to track the PHANToM's motion, and with it the user may explore the environment and feel the pieces therein. When in contact with a piece in the environment, the user may choose to select it. At this point, the user controls the motion of the selected piece in the environment. The selected piece can be moved about, restricted in motion by the other pieces in the environment, and dropped in any open space. Figure 4-1a shows the initial configuration of the pieces, and Figure 4-1b shows the simulation with pieces reorganized and one selected.

Experimentation with this simulation yielded an average spatial offset of 50 mm.

Given the 0.2 m / s average velocity estimated by the 'Hello, Voxel!' program, this corresponds to a 250 ms delay, or an update rate of 4 Hz, on a half-populated CAM-8. As expected from the previous results, this system is quite usable, although distances often feel somewhat larger than they actually are in the simulation.

There are a number of things that account for this slow update rate. Most importantly, this program makes terribly inefficient use of the CAM. The routines that attempt to move the grasp point to the desired position make use of the CAM in a serial manner, moving the grasp point only one cell per computation update of the entire simulation space. Additionally, the manner in which collision with the environment is detected is rather inefficient – it involves a transition at each cell and the summing of overlap information across the entire space, as well as the manipulation of this data on the host computer. This particular simulation is also not a particularly impressive use of the CAM, as the pieces in the simulation are completely static, so most cells have no interesting computation occuring. Additionally, the CAM-8 that this simulation ran on was only half-populated, with 4 processor boards. With an additional 4 processor boards, a times two speed improvement should be expected.

Later experimentation with a fully-populated CAM showed the expected speed improvement. An average spatial offset of 25 mm was measured, which corresponds to a 125 ms delay or an update rate of 8 Hz.

4.2.2 Static Piece Improvements

I was able to make a number of improvements to the above simulation that yielded a much faster update rate. Instead of attempting to move the grasp point one cell at a time, the simulation was modified so that the CAM attempts to move the grasp point four cells ahead. If this attempt fails, it does a binary search to determine the furthest the piece may be moved. This improvement alone should lend a nearly times four speed improvement, but it imposes the restriction that all pieces must be at least four cells thick.

The simulation was also modified to update the display less often. When updating the display, the CAM waits for the next video frame boundary, which is on average 33 ms away. By updating the display after every seven position updates, previously lost time is saved while still maintaining accurate video feedback to the user.

With these improvements, the average spatial offset fell to 7 mm, which corresponds to a 35 ms latency (29 Hz update rate) on a fully-populated machine. This provides a very solid and usable simulation. Another factor that contributes to the accuracy of this simulation is that the average spatial offset is a somewhat misleading value. Offsets are usually larger when the user is moving about in free space searching for an object. When exploring an object in detail motions are smaller; thus offsets are smaller and the simulation is more accurate.

As mentioned before, tracking the user motion in the simulation is unfortunately a serial operation, and as such is very time consuming. In the 512 by 512 by 1 simulation described above, cells were approximately 0.2 mm on a side in the x and y dimensions. Even small motions result in the CAM tracking the user through a large number of cells. In three-dimensional simulations, for example a 64 by 64 by 64 environment, cells are larger. User motions involve far fewer cell transitions, and so the update rate is further improved.

4.2.3 3-D Pieces

Now that I had shown that it was possible to interact effictively with two-dimensional simulations on the CAM, the next step was to extend the simulation to three dimensions. The static piece simulation was modified to represent a space 64 by 64 by 64 cells in size; this has the same number of cells as the 512 by 512 by 1 space and thus should run at the same rate. The routines that calculate user and piece motion were extended to support the third spatial dimension which was a simple modification as the as the z dimension is not functionally different from the first two. Appendices A through C include the source code for this project. Appendix

A, `voxel_client.c`, contains the source code for the client program that runs on the PHANToM host. Appendix B, `cam_server.c`, includes the server-side routines responsible for network communication and latency management. These routines are run-time linked into the Forth code that controls the CAM-8, included in Appendix C.

The existing 3-D display renderers only supported displaying all surfaces as a single color. While this is sufficient for many simulations, this display mode made it difficult to differentiate between pieces in this simulation. With the assistance of my advisor, Norman Margolus, we were able to extend the display routines to support three bits of color information for surfaces. This new display mode works extremely well for the 3-D pieces simulation.

This completed simulation demonstrates the success of my system. Exploring the environment and grasping pieces within it with the PHANToM are intuitive tasks, and the haptic and visual displays provide consistent feedback to the user. Figure 4-1c shows a configuation with three unselected pieces, with the grasp point in the upper right of the screen. Overlapping pieces in the projection and the shadows created by the rendering routine both provide depth information visually to the user. Figure 4-1d shows the smaller piece selected and moving about in front of the other objects.

4.2.4 Dynamic 3-D Pieces

The three-dimensional pieces experiment is an impressive simulation, but this experiment still does not make very effective use of the CAM processor power. As a simple demonstration of what more complex simulations are possible, this experiment was extended to include some additional dynamics, namely diffusion. When a piece is selected in this new simulation, it starts to send particles out into the environment. It behaves as if it were perhaps a frozen object being grasped by the user with hot tongs; particles steam off from it and then diffuse away.

This behavior can be seen in figure 4-1e. The piece in the upper left of the screen is currently selected, and so is surrounded by a cloud of randomly diffusing particles. The next image displays the system immediately after a large piece has been selected and then deleted; this allows the mass of particles to be seen before they diffuse entirely away.

Chapter 5

Conclusion

5.1 Review of Implementation Issues

I initially expected that a fair amount of work will be required to integrate the PHANToM haptic interface and the CAM-8 system. Here are the main issues I defined that would require significant research and decisions to be made.

5.1.1 CAM-8 Availability

When I started, there were only 3 assembled CAM-8 systems at MIT, all of which were under constant use. For my research, I needed a CAM-8 available for my use a great deal of the time. There are a large number of CAM-8 parts at the AI Lab, enough to construct at least several more working systems. I worked with Norm Margolus over the summer and fall to assemble additional machines. This work was fairly uneventful. We encountered some small bugs in the newest revision of the CAM-8 interface board, but were able to fix them.

5.1.2 Network Communication

Unfortunately, the CAM-8 system connects only to Sun SparcStations, while the PHANToM haptic interface does not. The PHANToM primary supported environment is Microsoft Windows NT. To handle this problem, a network protocol needed to be devised for passing the necessary information for physical interactions between a client system running Windows NT and a SparcStation server acting as the CAM-8 host.

This issue grew to encompass the bulk of my project. The simplest solution would have the NT machine acting as a thin client, doing little processing on its own and passing all data to the SparcStation, however as I describe in depth in Chapter 3 this was not sufficient. I explored a number of different options for handling the issues introduced by network and computational latency, and settled upon the constraint-oriented time/space tradeoff described above.

5.1.3 Data Format Reconcilliation

The GHOST toolkit provided for interfacing the the PHANToM is primarily based on high-level primitives such as polyhedral objects and polygon meshes. I was intially concerned as to how easily that could be reconciled with the pure bitmapped environment within the CAM. The Basic I/O library for the PHANToM provided me with a more amenable interface, allowing me to extract position data and return force data directly to the PHANToM.

5.2 Review of Goals

My goal in this project was to design a system to provide for intuitive user interfaces to bitmapped simulations. I hoped that this system would make understanding and interaction with many simulations far more simple than existing interfaces. I believe that the system as implemented achieves these goals, but this it is not without room for improvement. The current hardware puts this project just on the edge of possibility, where a proof-of-concept system was feasable and more complex systems will be possible soon.

The single point interface that the PHANToM provides is somewhat limiting in that it does not entirely allow users to interact with simulations the same way they interact with the world. The user is restricted to poking and tracing around the environment, and the inability to grasp objects with multiple fingers is distracting. The lack of tactile, in addition to haptic, feedback provides an interface different from what the user is familiar with, although it is a close approximation to interacting with the environment through a thimble. These issues can be addressed by more complex haptic interface hardware, and future advances in technology.

My goal of a general purpose interface to bitmapped simulations was not as simple as I first expected. It is often necessary to craft the haptic interface to the simulation at hand, because of the properties of the different simulations. For example, it was trivial to interact with the block-based simulations described in Chapter 4 because the pieces all had smooth surfaces and could slide past one another frictionlessly. If the pieces were more complex, however, it might not be easy for the user to manipulate the pieces effectively, particularly if the surfaces have interlocking boundaries. Such a simulation would require more complex rules for haptic interaction.

Additionally, the CAM-8 is a powerful machine, but it is still rather limited when it comes to real time simulations such as those described in Chapter 4. Realistically, a 64 by 64 by 64 simulation is the largest that is feasable with the current generation of CAM hardware, as the update rate demanded by haptic interaction is much higher than previous applications. The next generation CAM system currently being designed should be several orders of magnitude faster, and thus allow for substantially more complex simulations.

5.3 Future Directions

5.3.1 Single-host operation

For convenience, cost, and reliablity reasons, it would be desirable to have the PHANToM and CAM-8 both hosted on the same machine. This would reduce the need for expensive, under-utilized hardware, and obviate the need for coping with network latency and reliability. As the network latency is far outweighed by computational latency in current systems, this is not necessarily a high priority. As SBus is no longer a supported bus type for Sun hardware, and the Solaris drivers for the CAM exist only for SunOS 4, it would be best to build a PCI interface card for the CAM. Such a card could be used in modern SparcStations, as well in PC systems that host the PHANToM. On the PC, I would suggest future work take place on a more developer-friendly operating system than Windows NT, namely Linux or a BSD variant.

5.3.2 Effective CAM Usage

The current method for tracking the user within the simulation space is terribly inefficient due to the serial manner in which it makes use of the CAM. It may be possible to move the user through the simulation with a method other than ghost point tracking, though this may have the tradeoff of being able to tunnel through thin surfaces. Other methods should be explored to see how they effect simulation performance. Additionally, not all of the information from the simulation is currently used. When detecting collisions, the CAM returns to the server software not only if a collision occured, but also the number of cells that have overlapped. This information could potentially be used to aid the calculation of restoring forces.

5.3.3 PHANToM Thermal Display

A device to present thermal stimuli to a user of the PHANToM haptic interface has been developed at the MIT AI Lab [9]. A small stylus attached to the end of the PHANToM arm contains a thermo-electric heat pump and a water-cooled heat sink, and is able to present temperatures over the entire range of comfortable sensation. This device allows for the presentation of another dimension to the PHANToM user, which could be useful for interacting with CAM simulations. For example, many physics-based simulations on the CAM involve the concept of temperature, and this information could be passed along to the user through the thermal display.

5.3.4 Time/Space Tradeoff in Continuous-space Teleoperation

As described in section 3.7.2, the sort of constraint-based teleoperation I have used to reduce the effects of network and computational latency is also applicable to traditional continuous environments. The computations required on the server are very similar to the existing computations required for single-host operation. It would be interesting to explore this further, as it easily enables simulations involving multiple users on seperate systems interacting with a shared remote virtual environment.

5.3.5 Reapplication to Teleoperation in Real Environments

Coping with communucation delays has always been a serious concern with teleoperation systems, and new methods for compensating are constantly being explored. Although this current implementation of the time/space tradeoff method is somewhat odd in that it only shares position information between the master and slave systems, it should be possible to extend the system to share force information as well. If this were done, this method for coping with latency could be reapplied to teleoperation in real environments, providing a new tool for stabilizing force-reflecting teleoperation systems with significant delay.

5.4 Other Applications

Beyond the applications that the PHANToM haptic device is already being used for, facilitating physical interactions with a bitmapped environment such as that provided by the CAM-8 greatly expands the scope of environments that can be haptically manipulated. Simulations that were once unavailable due to the sheer computational complexity of providing real-time information on the environment would now be possible. Instead of just virtual environments with coarse-grained physics simulated at the object level, users would have the option of interacting with virtual environments simulating the physics of the environment at a much finer-grained particle level. Some potential applications of this technology would allow haptic interactions with material crystallization simulations, manipulation of large-scale molecular simulations, and other interactions with environments modelled with finer-grained physics allowing for more accurate simulations.

Besides such physics simulations, the CAM-8 supercomputer is very effective at computing the progress of a wide range of cellular automata in a large number of dimensions. Physical metaphors could be developed for haptic interactions with two-dimensional computations, such as Conway's Game of Life, or for simulations in four or more dimensions. The opportunity to physically manipulate a four-dimensional object, accounting for the still unavailable dimension in some manner, might provide a more intuitive visualization method for introductory purposes.

Not all of these example applications may be realizable with the computational power in the current CAM systems; howevern with the interface now developed, it should be straightforward to adapt it to work with future CAM systems with far greater computational power than the CAM-8. The possible applications of the PHANToM / CAM pairing are virtually endless.

5.4.1 Crystallization and Annealing

The CAM-8 has been used extensively for large statisical simulations such as those used to model annealing and crystallization. As the simulation runs, the CAM also provides a ray-traced display of the progress of the material system. Like most CAM simulations, however, there is not a clear way to interact with the virtual environment as the simulation is running. By inserting the user into the virtual environment by use of the PHANToM device, however, the user could touch the simulated crystals, poke holes in structures with their finger tip, and examine the haptic properties of the virtual surfaces.

5.4.2 n-body Simulations

As the CAM provides such a high computational density, it is able to accurately and quickly compute the interactions of large n-body problems, as might be found in a lattice-gas simulation, a model of the forces involved in molecular interactions, or any other similar problem. Such problems are computationally intractable for real-time work on the platforms with which the PHANToM can currently interact. With the linked PHANToM and CAM, however, users would be able to reach into such simulations and feel the forces at work. For example, with the appropriate CA rules one could take a simulation of a protein molecule and examine the forces that occur as it interacts with other molecules, or is folded.

5.4.3 Real-Time Video Manipulation

The CAM-8 has an integral video input which can feed data directly into a simulation. The video data can then propagate through the system as determined by the rules of the CA. Many optical effects can be easily simulated in real time with this facility – arbitrary lens effects, for example. In another example, the video data could appear in the CAM on a simulated flexible membrane stretched across the field of simulation. With the PHANToM, a user could deform this membrane and instantly see the visual effects of that deformation.

5.4.4 Interaction Metaphor

It remains to be decided exactly how the user will interact with certain kinds of simulated environments. Overall design for three-dimension environments is quite clear, inserting the user's finger point into the simulated environment and monitoring the forces on those cells is appropriate for most simulations. Interactions with environments with greater than three dimensions are more puzzling. Obviously it is necessary to give the user an n-dimensional finger so that they can meaningfully interact with the simulation, but how do we then reconcile the results with the PHANToM's 'mere' 3-D interface? Haptic interaction with greater than three-dimensional simulated spaces is an intriguing topic for future research.

5.4.5 Virtual Instruments, and Other Artistic Endeavors

Perhaps a more fanciful example of what should be possible in the future, would be physically simulating a musical instrument within a CAM-like supercomputer. Given a stringed instrument such as a guitar or a zither, the effects of the vibrations of plucked or strummed strings upon sounding board, surrounding air, and adjacent strings could be simulated in such a way that a resulting audio output could be produced based only on the simulated physical interactions of the components and not on higher level approximations of instrument sounds. Using the haptic interface, a user could play such a virtual instrument as if it were a real one. Since the instrument only exists within the computational world of the CAM, it could have properties not possible in the real world. One could experiment with the musical effects of using materials in instrument construction not available in the real world, or even with the effects of playing the instrument in a universe with different fundamental laws.

This is just one example from a huge field of artistic and entertainment related possibilities for the linked PHANToM/CAM system. Others include simulated sculpting with realistic interactions, new concepts in 'fingerpainting', and interactive virtual artwork. While realistically the expense of such a system would prevent it from being widely used for such endeavors in the near future, the commodity component design of the CAM and the mass-market appeal of the PHANToM, now being sold commercially by SensAble Technologies, make such a tool not entirely out of reach for a great many users. While some might consider the research value in such a system dubious, this is just one possible application of the system, and it is vital to remember that such creative and whimsical ideas are integral to the learning process.

5.5 Conclusion

The PHANToM haptic interface and the CAM-8 supercomputer are both innovative devices that complement each other well; the CAM-8 is a powerful processing system without a real-time interactive user interface for the physical simulations at which it excels, and the PHANToM is a unique interface for interacting with such simluations. This system in which the PHANToM can interact with three-dimensional bitmapped virtual environments has a large number of exciting applications, and advances the accessibility of both technologies.

Appendix A

voxel_client.c

```c
#define WIN32_LEAN_AND_MEAN
#include <winsock2.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>

/* phantom includes */
#include "phantom.h"
#include "os_extender.h"

/* constants */
#define DEFAULT_PORT 5001
#define DEFAULT_SERVER "watt.ai.mit.edu"

/* stiffness of voxels in N/mm */
#define STIFFNESS 0.8
#define DAMPING 0.01
#define COORD_SPRING (float)0.05

/* hysteresis in spatial offsetting, to prevent 'jitter' between cells.
   When a user penetrates an occupied cell and the coordinate system
   is shifted, how far into the occupied cell do they remain? */
#define HYSTERESIS (float)0.5

#ifndef TRUE
#define TRUE 1
#define FALSE 0
#endif

float constraint[3];
float adjpos[3];
unsigned char constdir, oldconstdir;
```

```
CRITICAL_SECTION PositionCriticalSection, ConstraintCriticalSection;

/********
**
** local function headers
**
********/
SCHEDULER_CALLBACK servoLoopCallback(void *userData);

void Usage(char *progname) {
        fprintf(stderr,"Usage:\n%s [-h] [-s server] [-p port]\n",progname);
        fprintf(stderr,"Where:\n\t-h     displays this usage message\n");
        fprintf(stderr,"\tserver is the IP address or name of the server\n");
        fprintf(stderr,"\tport   is the UDP port to communicate on\n");
        WSACleanup();
        exit(1);
}

/*  Main
        Initializes OSExtender and PHANToM.  Resets PHANToM.  Starts servo loop
        and waits for user to hit key to end program.
 */
void main(int argc, char** argv)
{
  char *server_name = DEFAULT_SERVER;
  unsigned short port = DEFAULT_PORT;
  unsigned int addr;
  struct sockaddr_in server;
  struct hostent *hp;
  WSADATA wsaData;
  SOCKET conn_socket;
  char buffer[256];
  long *lptr = (long *)buffer;
  float *fptr= (float*)buffer;
  int i, retval, phantom_id;
  float position[3];
  long *longptr;

  /* start OS Extender (must be done first) */
  initOSExtender();

  /* Process command line arguments */
  for (i=1; i<argc; i++) {
      if ((argv[i][0] == '-') || (argv[i][0] == '/')) {
          switch (tolower(argv[i][1])) {
          case 'h':
              Usage(argv[0]);
              break;

          case 's':
              if ((i+1) < argc)
                  server_name = argv[++i];
              else Usage(argv[0]);
              break;
```

```
        case 'p':
            if ((i+1) < argc)
                port = atoi(argv[++i]);
            else Usage(argv[0]);
            if (port == 0)
                Usage(argv[0]);
            break;

        default:
            Usage(argv[0]);
            break;
        }
    }
}

/* Initialize the phantom */
phantom_id = init_phantom( "phantom.ini");
if (phantom_id < 0) {
    printf("Error initializing PHANToM\n");
    getchar();
    exit(1);
}

/* Tell user to hold phantom in reset position */
printf("Hold PHANToM in neutral position and press enter\n");
printf("Do not use reset arm on PHANToM model 3.0\n");
getchar();

/* reset the phantom - this calibrates the sensors on the PHANToM.
   It must be done before reading position or writing forces.  This function
   must be called while the PHANToM is in the reset position otherwise the
   positions read from the PHANToM will be incorrect. */
if (phantom_reset(phantom_id)) {
    disable_phantom(phantom_id);
    printf("Error reseting PHANToM\n");
    getchar();
    exit(1);
}

/* Initialize the network */
if (WSAStartup(0x202, &wsaData) == SOCKET_ERROR) {
    fprintf(stderr, "WSAStartup failed with error %d\n", WSAGetLastError());
    disable_phantom(phantom_id);
    WSACleanup();
    exit(1);
}
/* Get host address */
if (isalpha(server_name[0])) {
    hp = gethostbyname(server_name);
} else {
    addr = inet_addr(server_name);
    hp = gethostbyaddr((char *)&addr,4,AF_INET);
}
```

```
if (hp == NULL) {
    fprintf(stderr,"Client: Cannot resolve address [%s]: Error %d\n",
            server_name,WSAGetLastError());
    disable_phantom(phantom_id);
    WSACleanup();
    exit(1);
}

/* Build sockaddr_in structure */
memset(&server,0,sizeof(server));
memcpy(&(server.sin_addr), hp->h_addr, hp->h_length);
server.sin_family = hp->h_addrtype;
server.sin_port = htons(port);

/* Open the socket */
conn_socket = socket(AF_INET, SOCK_DGRAM, 0);
if (conn_socket < 0) {
    fprintf(stderr,"Client: Error Opening Socket: Error %d\n",
            WSAGetLastError());
    disable_phantom(phantom_id);
    WSACleanup();
    exit(1);
}

/* Associate a connection with this socket */
if (connect(conn_socket, (struct sockaddr*)&server, sizeof(server))
    == SOCKET_ERROR) {
    fprintf(stderr,"connect() failed: %d\n",WSAGetLastError());
    disable_phantom(phantom_id);
    WSACleanup();
    exit(1);
}

/* Handshake with server maybe? */
fprintf(stderr,"Contacting CAM server...\n");

/* Intialize mutexes */
InitializeCriticalSection(&PositionCriticalSection);
InitializeCriticalSection(&ConstraintCriticalSection);

/* start the servo loop - it does all the work:
   updates phantom, calculates reaction forces
   and sends forces to the phantom.  We use os_extender
   function startServoLoop so that this will run in real time.
   Pass startServoLoop phantom id as user data
   it will need for other iolib functions. */
startServoLoop(servoLoopCallback, (void *) &phantom_id);

/* while servo loop is running we wait for user to hit retrun to exit */
printf("Interact with the remote environment.\n");
printf("Hit enter to exit.\n");

while (1) {
    /* Acquire position mutex */
```

```
    EnterCriticalSection(&PositionCriticalSection);
    /* Get the position from memory */
    memcpy(position, adjpos, 12);
    /* Release position mutex */
    LeaveCriticalSection(&PositionCriticalSection);

    /* Send the position over the to the CAM host */
    /* These are floating point values. Luckily, both x86 and Sparc use
     * the IEEE floating point representation, modulo endianess. This is
     * NOT NECESSARILY THE CASE for other platforms! If this is ported to
     * some other platform, it may be necessary to use xdr or some sane
     * standard float representation.
     */
    longptr = (long *)position;
    for (i=0; i<3; i++)
        longptr[i] = htonl(longptr[i]);
    retval = send(conn_socket, (const void *) position, 12, 0);
    if (retval == SOCKET_ERROR) {
        fprintf(stderr,"send() failed: error %d\n",WSAGetLastError());
        disable_phantom(phantom_id);
        WSACleanup();
        exit(1);
    }

    /* Receive new constraints */
    retval = recv(conn_socket, buffer, sizeof(buffer), 0);
    if (retval == SOCKET_ERROR) {
        fprintf(stderr,"recv() failed: error %d\n",WSAGetLastError());
        closesocket(conn_socket);
        WSACleanup();
        exit(1);
    }
    if (retval != 13) {
        fprintf(stderr, "Weird packet...size %d\n",retval);
        continue;
    }

    /* Acquire constraints mutex */
        EnterCriticalSection(&ConstraintCriticalSection);
    for (i=0; i<3; i++) {
        lptr[i] = ntohl(lptr[i]);
        constraint[i] = fptr[i];
    }
    constdir = buffer[12];

    /* Release constraints mutex */
        LeaveCriticalSection(&ConstraintCriticalSection);

    /* TODO - Check for user interrupt */
}
getchar();

/* user has exited stop the servo loop, disable the phantom and quit */
stopServoLoop();
```

```
  disable_phantom(phantom_id);
  closesocket(conn_socket);
  WSACleanup();
}

/*      servoLoopCallback
*       Called at 1khz by the OS Extender to read phantom position and
*       set phantom forces.  Calculates intersections of phantom point
*       with objects in the scene and sets appropriate forces.
*/
SCHEDULER_CALLBACK servoLoopCallback(void *userData) {
    int phantom_id = *((int *) userData);
    int error;
    unsigned char newv;
    float position[3], vel[3], force[3];
    static float offset[3];
    static int first_time = TRUE, dcount = 0;

    /* the time first through the loop we need to enable the PHANToM forces.
       We don't do this before starting the servo loop because the
       hardware timeout circuit will disable the forces before the servo loop
       begins. */
    if (first_time) {
        if (error = enable_phantom_forces(phantom_id)) {
            disable_phantom(phantom_id);
            printf("Error enabling PHANToM forces\n");
            getchar();
            exit(1);
        }
        first_time = FALSE;
        zero_vector(force);
        zero_vector(offset);
        zero_vector(constraint);
        constdir = 0; oldconstdir = 0;
    }


    /* Acquire position mutex */
        EnterCriticalSection(&PositionCriticalSection);
    /* Update the PHANToM
       This updates the current state (position, velocity etc...)
       of the PHANToM.  Must be called once and only once every servo loop
       otherwise values returned by get_phantom_pos, get_phantom_vel, etc..
       will not be correct */
    if (update_phantom(phantom_id)) {
        disable_phantom(phantom_id);
        printf("Error updating PHANToM\n");
        getchar();
        exit(1);
    }

    /* Get the position from memory */
    get_phantom_pos(phantom_id, position);
        get_phantom_vel(phantom_id, vel);
```

```
/* Calculate adjusted position */
adjpos[0] = position[0] + offset[0];
adjpos[1] = position[1] + offset[1];
adjpos[2] = position[2] + offset[2];
/* Release position mutex */
LeaveCriticalSection(&PositionCriticalSection);


/* Acquire constraints mutex */
EnterCriticalSection(&ConstraintCriticalSection);

/* Offset new boundary violations if > hysteresis */
newv = constdir - (constdir & oldconstdir);
oldconstdir = constdir;
if (((newv & 0x01) && (adjpos[0] > constraint[0] + HYSTERESIS)) ||
    ((newv & 0x02) && (adjpos[0] + HYSTERESIS < constraint[0])))
    offset[0] = constraint[0] - position[0] + HYSTERESIS;

if (((newv & 0x04) && (adjpos[1] > constraint[1] + HYSTERESIS)) ||
    ((newv & 0x08) && (adjpos[1] + HYSTERESIS < constraint[1])))
    offset[1] = constraint[1] - position[1] + HYSTERESIS;

if (((newv & 0x10) && (adjpos[2] > constraint[2] + HYSTERESIS)) ||
    ((newv & 0x20) && (adjpos[2] + HYSTERESIS < constraint[2])))
    offset[2] = constraint[2] - position[2] + HYSTERESIS;

/* Calculate adjusted position */
adjpos[0] = position[0] + offset[0];
adjpos[1] = position[1] + offset[1];
adjpos[2] = position[2] + offset[2];

/* Calculate phantom forces based on constraints */
zero_vector(force);
if (((constdir & 0x01) && (adjpos[0] > constraint[0])) ||
    ((constdir & 0x02) && (adjpos[0] < constraint[0])))
  force[0] += (float) STIFFNESS * (constraint[0] - adjpos[0])
                 - (float) DAMPING * vel[0]
             + (float) COORD_SPRING * offset[0];

if (((constdir & 0x04) && (adjpos[1] > constraint[1])) ||
    ((constdir & 0x08) && (adjpos[1] < constraint[1])))
  force[1] += (float) STIFFNESS * (constraint[1] - adjpos[1])
                 - (float) DAMPING * vel[1]
             + (float) COORD_SPRING * offset[1];

if (((constdir & 0x10) && (adjpos[2] > constraint[2])) ||
    ((constdir & 0x20) && (adjpos[2] < constraint[2])))
  force[2] += (float) STIFFNESS * (constraint[2] - adjpos[2])
                 - (float) DAMPING * vel[2]
             + (float) COORD_SPRING * offset[2];

/* Attempt to restore coordinate system */
/* Really should merge this with previous, since conditions are same */
```

```
    /* if not blocked, move back small amount; otherwise
     * add restoring force */
    if ((dcount % 10) == 0) {
      if (((constdir & 0x01) && (adjpos[0] > constraint[0])) ||
          ((constdir & 0x02) && (adjpos[0] < constraint[0])))
          /* Coordinate spring contribution moved above */;
      else if (fabs(offset[0]) > 4 * HYSTERESIS)
        offset[0] -= (HYSTERESIS / 2) * ((offset[0] > 0) ? 1 : -1);

      if (((constdir & 0x04) && (adjpos[1] > constraint[1])) ||
          ((constdir & 0x08) && (adjpos[1] < constraint[1])))
          /* Coordinate spring contribution moved above */;
      else if (fabs(offset[1]) > 4 * HYSTERESIS)
        offset[1] -= (HYSTERESIS / 2) * ((offset[1] > 0) ? 1 : -1);

      if (((constdir & 0x10) && (adjpos[2] > constraint[2])) ||
          ((constdir & 0x20) && (adjpos[2] < constraint[2])))
          /* Coordinate spring contribution moved above */;
      else if (fabs(offset[2]) > 4 * HYSTERESIS)
        offset[2] -= (HYSTERESIS / 2) * ((offset[2] > 0) ? 1 : -1);
    }

    /* Release constraints mutex */
    LeaveCriticalSection(&ConstraintCriticalSection);

    dcount++;
    if ((dcount % 100) == 0)
      printf("%X Force: %f %f %f  Offset: %f %f %f\n",constdir,
             force[0], force[1], force[2],
             offset[0], offset[1], offset[2]);
    /* Send the latest force information to the PHANToM and error check */
    if (send_phantom_force(phantom_id, force)) {
        disable_phantom(phantom_id);
        printf("Error sending PHANToM forces\n");
        getchar();
        exit(1);
    }
}
```

Appendix B

cam_server.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>

#define DEFAULT_PORT 5001

/* haptic workspace size in mm*/
#define X_WORKSPACE 90.0
#define Y_WORKSPACE 90.0
#define Z_WORKSPACE 90.0

/* Axis indices */
#define X 0
#define Y 1
#define Z 2

/* Globals */
int DIM_X, DIM_Y, DIM_Z;
float LEN_X, LEN_Y, LEN_Z;
int user[3], ghost[3], offset[3];
struct sockaddr_in from;
int fromlen;
int msgsock;

/* cam_startup – Intialize interface to CAM simulation
 *
 * Called from Forth at entry into simulation loop
 *
 * Accepts: ints containing x, y, z dimensions of simulation
```

```
 */
void cam_startup(int xdim, int ydim, int zdim,
                 int sx, int sy, int sz) {
    char *interface= NULL;
    struct sockaddr_in local;
    int listen_socket;

    /* Intialize simulation size */
    DIM_X = xdim;  LEN_X = X_WORKSPACE / DIM_X;
    DIM_Y = ydim;  LEN_Y = Y_WORKSPACE / DIM_Y;
    DIM_Z = zdim;  LEN_Z = Z_WORKSPACE / DIM_Z;

    /* hmm, but Phantom starts at center of workspace... */
    user[X] = ghost[X] = DIM_X - sx;
    user[Y] = ghost[Y] = DIM_Y - sy;
    user[Z] = ghost[Z] = DIM_Z - sz;

    local.sin_family = AF_INET;
    local.sin_addr.s_addr = (!interface)?INADDR_ANY:inet_addr(interface);
    local.sin_port = htons(DEFAULT_PORT);

    listen_socket = socket(AF_INET, SOCK_DGRAM, 0);
    if (listen_socket == -1){
        fprintf(stderr, "socket() failed with error %d\n", errno);
        exit(1);
    }
    if (bind(listen_socket, (struct sockaddr*)&local, sizeof(local)) == -1) {
        fprintf(stderr, "bind() failed with error %d\n", errno);
        exit(1);
    }

    msgsock = listen_socket;
    fromlen = sizeof(from);
}


void cam_shutdown() {
    close(msgsock);
}


/* get_desired_offset - Wait for next position update from client
 *                      and return desired offset from current position
 *
 * Called from Forth in main event loop
 *
 * Returns: Pointer to array of int containting x, y, z offsets
 *
 * Modifies: user, offset
 */
int *get_desired_offset() {
    char Buffer[128];
    long *longptr = (long *)Buffer;
    float *position = (float *)Buffer;
```

```
    int retval, i;

    retval = 0;
    /* wait for properly formed packet */
    while (retval != 12) {
        retval = recvfrom(msgsock, Buffer, sizeof(Buffer), 0,
                          (struct sockaddr *)&from, &fromlen);

        if (retval == -1)
            fprintf(stderr, "recvfrom() failed: error %d\n", errno);

        /* Is this a properly formed packet? */
        if (retval != 12)
            fprintf(stderr, "Malformed packet!\n");
    }

    /* Convert from network byte order */
    for (i=0; i<3; i++)
        longptr[i] = ntohl(longptr[i]);

    /* User position (mm) in position[] */
    /* Find user's cell */
    user[X] = (position[X] + LEN_X * DIM_X / 2) / LEN_X;
    user[Y] = (position[Y] + LEN_Y * DIM_Y / 2) / LEN_Y;
    user[Z] = (position[Z] + LEN_Z * DIM_Z / 2) / LEN_Z;

    /* Check for out-of-range cells */
    if (user[X] < 0) user[X] = 0;
    if (user[Y] < 0) user[Y] = 0;
    if (user[Z] < 0) user[Z] = 0;
    if (user[X] > (DIM_X - 1)) user[X] = DIM_X - 1;
    if (user[Y] > (DIM_Y - 1)) user[Y] = DIM_Y - 1;
    if (user[Z] > (DIM_Z - 1)) user[Z] = DIM_Z - 1;

    /* Calculate offset for simulation */
    offset[X] = user[X] - ghost[X];
    offset[Y] = user[Y] - ghost[Y];
    offset[Z] = user[Z] - ghost[Z];

    return offset;
}


/* calculate_constraints - Calculate what constraints are active on the
 *                         ghost point given it and the user's current
 *                         positions.
 *
 * This function calculates the X, Y and Z constraints now active on the
 * ghost point.  It must be called after move_ghost_to_user() or the results
 * may be incorrect.
 *
 * Accepts: pointer to user triplet
 *          pointer to ghost triplet
 *          pointer to constraint triplet
```

```
 *            pointer to constraint direction triplet
 *
 * Modifies: constraint triplet
 *             constraint direction triplet
 */
void calculate_constraints(int *user, int *ghost, int *constraint,
                           int *constdir) {
    /* X direction */
    if (ghost[X] < user[X]) {
        constraint[X] = ghost[X]+1;
        constdir[X] = -1;           /* less than */
    } else if (ghost[X] > user[X]) {
        constraint[X] = ghost[X];
        constdir[X] = 1;            /* greater than */
    } else
        constdir[X] = 0;            /* no constraint */

    /* Y direction */
    if (ghost[Y] < user[Y]) {
        constraint[Y] = ghost[Y]+1;
        constdir[Y] = -1;           /* less than */
    } else if (ghost[Y] > user[Y]) {
        constraint[Y] = ghost[Y];
        constdir[Y] = 1;            /* greater than */
    } else
        constdir[Y] = 0;            /* no constraint */

    /* Z direction */
    if (ghost[Z] < user[Z]) {
        constraint[Z] = ghost[Z]+1;
        constdir[Z] = -1;           /* less than */
    } else if (ghost[Z] > user[Z]) {
        constraint[Z] = ghost[Z];
        constdir[Z] = 1;            /* greater than */
    } else
        constdir[Z] = 0;            /* no constraint */
}


/* set_actual_offset - Set actual offset user could move, calculate
 *                       constraints, and inform client
 *
 * Called from Forth in main event loop
 *
 * Accepts: ints containing x, y, z offsets
 *
 * Modifies:
 */
void set_actual_offset(int actualx, int actualy, int actualz) {
    int constraint[3];
    static int constdir[3] = {0, 0, 0};
    unsigned char cdir;
    char Buffer[128];
    long *longptr = (long *)Buffer;
```

```
    float *fconst = (float *)Buffer;
    int retval, i;

    /* Move ghost to user */
    ghost[X] += actualx;
    ghost[Y] += actualy;
    ghost[Z] += actualz;

    calculate_constraints(user, ghost, constraint, constdir);

    fconst[X] = constraint[X] * LEN_X - LEN_X * DIM_X / 2;
    fconst[Y] = constraint[Y] * LEN_Y - LEN_Y * DIM_Y / 2;
    fconst[Z] = constraint[Z] * LEN_Z - LEN_Z * DIM_Z / 2;

    cdir = (constdir[X] < 0)   |
        (constdir[X] > 0) << 1 |
        (constdir[Y] < 0) << 2 |
        (constdir[Y] > 0) << 3 |
        (constdir[Z] < 0) << 4 |
        (constdir[Z] > 0) << 5;

    /* Send constraints back to client */
    for (i=0; i<3; i++) {
        longptr[i] = htonl(longptr[i]);
    }
    Buffer[12] = cdir;
    retval = sendto(msgsock, Buffer, 13, 0,
                    (struct sockaddr *)&from, fromlen);
    if (retval == -1)
        fprintf(stderr,"sendto() failed: error %d\n", errno);
}
```

Appendix C

pieces-3d.exp

```
\ 3−D haptic blocks

\* We allow objects with up to 63 component parts, numbered 1 to 63.
A part is selected, and that part is copied into the "moving−part"
plane, and the "origin" plane.  The original is then erased.  The
moving body is then shifted around as desired, with the amount of
overlap with other parts returned after each motion.  When the moving
body is released, if it overlaps anything, it is moved back to it's
origin position.  If it doesn't overlap, it's copied to it's new
position. *\


new−experiment  64 by 64 by 64 space   load step−rend−color.fth


\ ***********************
\ Definition of cell bits
\ ***********************

 1  2 == moving−part             \ 1 = part of shifting bitmap
        1  1 == moving−shape
        2  2 == grasp−point
 3  3 == overlap                 \ 1 where moving body overlaps other parts
 4  9 == part#                   \ up to 63 parts  (0 = background)
10 15 == select#                 \ external input used to select a part
10 15 == show#                   \ bits shown by 3D display


63 constant max−part#

\ ***********************
\ Part Manipulation Rules
\ ***********************
```

```
\* To grasp a part, the grasp point must overlap the part.  The part
number will be copied into the select# at that point, 0 goes into
select# elsewhere.  When select# is non-zero, we've selected a part. *\

: grasp-part-rule

                grasp-point 0<>
        if
                part# -> select#
        else
                0 -> select#
        then
;
create-lut grasp-part-table
?rule>table grasp-part-rule grasp-part-table


\* When we select a part, we mark its position in the "moving-shape"
plane, and erase it from the "part#" planes.  "select#" is a given
external parameter which is set in all cells.  *\

: show-part
        part# -> show#
        moving-shape 0<> if max-part# -> show# then
;

: select-part-rule

                select# part# =
        if
                1 -> moving-shape       0 -> part#
        else
                0 -> moving-shape
        then
                show-part
;
create-lut select-part-table
?rule>table select-part-rule select-part-table


\* We can later deposit the "moving-shape" back into the "part#" planes in
it's new position. *\

: deposit-part-rule

        grasp-point -> moving-shape
        moving-shape if select# -> part# then
        show-part
;
create-lut deposit-part-table
?rule>table deposit-part-rule deposit-part-table
```

```
\* While the part is moving, we monitor the overlap. *\

: overlap-rule

        part# 0<> moving-shape 0<> and -> overlap
        show-part
;
create-lut overlap-table
?rule>table overlap-rule overlap-table


\ *******************
\ Running the dynamics
\ *******************

\* We normally run using the "overlap-rule".  Whenever we run a step
with another rule, we'll reset the table to continue with the
"overlap-rule". *\

: init-tables

        site-src  lut
        lut-src   site
        lut-data        overlap-table
        switch-luts
        step
;
        this is when-starting

: run-table

        kick
        run             new-table       \ run with a new table and then
        switch-luts                     \ switch back to old table,
        lut-src         site            \ and leave lut-src set to site
;

define-step grasp-part-step

        lut-data        grasp-part-table
        run-table
end-step


: select-part-step (s part# -- )

        lut-data        select-part-table
        lut-src         site
                        select# field fix
        run-table
        step
;

: deposit-part-step (s part# -- )
```

```
        lut-data        deposit-part-table
        lut-src         site
                        select#  field  fix
        run-table
        step
;

: shift-part (s x y z -- )

        kick      moving-part  field   z  y  x
        run
;


define-step  shift00step    0   0   0  shift-part   end-step
define-step  shiftx+step    1   0   0  shift-part   end-step
define-step  shiftx-step   -1   0   0  shift-part   end-step
define-step  shifty+step    0   1   0  shift-part   end-step
define-step  shifty-step    0  -1   0  shift-part   end-step
define-step  shiftz+step    0   0   1  shift-part   end-step
define-step  shiftz-step    0   0  -1  shift-part   end-step

define-step  2shift00step   0   0   0  shift-part   end-step
define-step  2shiftx+step   2   0   0  shift-part   end-step
define-step  2shiftx-step  -2   0   0  shift-part   end-step
define-step  2shifty+step   0   2   0  shift-part   end-step
define-step  2shifty-step   0  -2   0  shift-part   end-step
define-step  2shiftz+step   0   0   2  shift-part   end-step
define-step  2shiftz-step   0   0  -2  shift-part   end-step

define-step  4shift00step   0   0   0  shift-part   end-step
define-step  4shiftx+step   4   0   0  shift-part   end-step
define-step  4shiftx-step  -4   0   0  shift-part   end-step
define-step  4shifty+step   0   4   0  shift-part   end-step
define-step  4shifty-step   0  -4   0  shift-part   end-step
define-step  4shiftz+step   0   0   4  shift-part   end-step
define-step  4shiftz-step   0   0  -4  shift-part   end-step

\ *************************
\ Initialization and control
\ *************************

: Initial.test.pattern

        "" pieces-3d-shifted.pat  file>cam   show
;
press I  "Initialize test pattern."


: overlap-count (s -- overlap# )  *count* overlap field count-field
;
```

```
: .overlap overlap-count . cr ;


\* Given desired move, go as far as possible without overlap *\

0 constant actualx
0 constant actualy
0 constant actualz

0 constant desirex
0 constant desirey
0 constant desirez

false constant free-motion?

: move1x
        desirex actualx <>
    if
        desirex 0< abs {{ shiftx-step shiftx+step }}
    then
;

: move1y
        desirey actualy <>
    if
        desirey 0< abs {{ shifty-step shifty+step }}
    then
;

: move1z
        desirez actualz <>
    if
        desirez 0< abs {{ shiftz-step shiftz+step }}
    then
;

\ This function creates the requirement that all objects must be at
\ least 4 cells in each direction!
: ?move4x     (s -- flag)

    \ Try moving part 4 cells
    desirex 0< abs {{ 4shiftx-step 4shiftx+step }}

        overlap-count 0<>
    if
        \ If overlapping, move back 2 cells
        desirex 0< abs {{ 2shiftx+step 2shiftx-step }}

            overlap-count 0<>
        if
            \ If still overlapping, move back 1 cell
            desirex 0< abs {{ shiftx+step shiftx-step }}

                overlap-count 0<>
```

```
                if
                        \ If still overlapping, move back to original cell
                        desirex 0< abs {{ shiftx+step shiftx-step }} false
                else
                        desirex sig actualx + is actualx                true
                then
        else
                \ Free, try moving forward 1 cell
                desirex 0< abs {{ shiftx-step shiftx+step }}

                        overlap-count 0<>
                if
                        \ Overlapping, moving back one
                        desirex 0< abs {{ shiftx+step shiftx-step }}
                        desirex sig 2* actualx + is actualx         true
                else
                        desirex sig 3 * actualx + is actualx        true
                then
        then
    else
        desirex sig 4* actualx + is actualx                  true
    then
;

\ This function creates the requirement that all objects must be at
\ least 4 cells in each direction!
: ?move4y     (s -- flag)

    \ Try moving part 4 cells
    desirey 0< abs {{ 4shifty-step 4shifty+step }}

        overlap-count 0<>
    if
        \ If overlapping, move back 2 cells
        desirey 0< abs {{ 2shifty+step 2shifty-step }}

                overlap-count 0<>
        if
                \ If still overlapping, move back 1 cell
                desirey 0< abs {{ shifty+step shifty-step }}

                        overlap-count 0<>
                if
                        \ If still overlapping, move back to original cell
                        desirey 0< abs {{ shifty+step shifty-step }} false
                else
                        desirey sig actualy + is actualy                true
                then
        else
                \ Free, try moving forward 1 cell
                desirey 0< abs {{ shifty-step shifty+step }}

                        overlap-count 0<>
                if
```

```
                    \ Overlapping, moving back one
                    desirey 0< abs {{ shifty+step shifty-step }}
                    desirey sig 2* actualy + is actualy        true
                else
                    desirey sig 3 * actualy + is actualy       true
                then
            then
        else
            desirey sig 4* actualy + is actualy                true
        then
;

\ This function creates the requirement that all objects must be at
\ least 4 cells in each direction!
: ?move4z      (s -- flag)

        \ Try moving part 4 cells
        desirez 0< abs {{ 4shiftz-step 4shiftz+step }}

            overlap-count 0<>
        if
            \ If overlapping, move back 2 cells
            desirez 0< abs {{ 2shiftz+step 2shiftz-step }}

                overlap-count 0<>
            if
                \ If still overlapping, move back 1 cell
                desirez 0< abs {{ shiftz+step shiftz-step }}

                    overlap-count 0<>
                if
                    \ If still overlapping, move back to original cell
                    desirez 0< abs {{ shiftz+step shiftz-step }} false
                else
                    desirez sig actualz + is actualz           true
                then
            else
                \ Free, try moving forward 1 cell
                desirez 0< abs {{ shiftz-step shiftz+step }}

                    overlap-count 0<>
                if
                    \ Overlapping, moving back one
                    desirez 0< abs {{ shiftz+step shiftz-step }}
                    desirez sig 2* actualz + is actualz        true
                else
                    desirez sig 3 * actualz + is actualz       true
                then
            then
        else
            desirez sig 4* actualz + is actualz                true
        then
;
```

```
: ?move1x    ( s -- flag )
             desirex actualx <>
      if
             desirex 0< abs {{ shiftx-step shiftx+step }}
                    overlap-count 0<>
             if
                    desirex 0< abs {{ shiftx+step shiftx-step }} false
             else
                    desirex sig actualx + is actualx           true
             then
      else
             false
      then
;
: ?move1y    ( s -- flag )
             desirey actualy <>
      if
             desirey 0< abs {{ shifty-step shifty+step }}
                    overlap-count 0<>
             if
                    desirey 0< abs {{ shifty+step shifty-step }} false
             else
                    desirey sig actualy + is actualy           true
             then
      else
             false
      then
;
: ?move1z    ( s -- flag )
             desirez actualz <>
      if
             desirez 0< abs {{ shiftz-step shiftz+step }}
                    overlap-count 0<>
             if
                    desirez 0< abs {{ shiftz+step shiftz-step }} false
             else
                    desirez sig actualz + is actualz           true
             then
      else
             false
      then
;
: ?movex     (s -- flag)
             desirex actualx - abs 4 >=
```

```
    if
        ?move4x
    else
        ?move1x
    then
;

: ?movey    (s -- flag)

        desirey actualy - abs 4 >=
    if
        ?move4y
    else
        ?move1y
    then
;

: ?movez    (s -- flag)

        desirez actualz - abs 4 >=
    if
        ?move4z
    else
        ?move1z
    then
;

0 constant show-counter
7 constant show-ratio

: move-xyz  ( dx dy dz -- dx' dy' dz' )

      is desirez    is desirey    is desirex
    0 is actualz  0 is actualy  0 is actualx

        free-motion?
    if
        0 desirex -
        0 desirey -
        0 desirez -
        kick      moving-part field  z y x
        run
        step

        desirex is actualx
        desirey is actualy
        desirez is actualz
    else
        begin
        begin ?movex not until
        begin ?movey not until
        begin ?move1z not until
        ?move1x ?move1y ?move1z or or not until
    then
```

```
        show−counter 1+ is show−counter
                show−counter show−ratio mod 0=
        if
                show
        then
        actualx actualy actualz
;

false constant part−selected?

0 constant last−selected−part#

: pick−up−part

        move1x move1y move1z

        grasp−part−step  *count*
        select# field count−field

             dup 0=
        if
             ." Nothing there! " cr drop
        else
             dup is last−selected−part#
             select−part−step  show
             true is part−selected?
        then
;

: deposit−part

             shift00step  overlap−count  0<>
        if
             ." No room! " cr
        else
             last−selected−part#
             deposit−part−step  show
             false is part−selected?
        then
;

: Delete−part

             part−selected?
        if
                0
                deposit−part−step  show
                false is part−selected?
        else
                ." No part selected! " cr
        then
;
press D "Delete selected part."
```

```
: Select/Drop

          part-selected?
     if
          deposit-part
     else
          pick-up-part
     then
          shift00step show
;
press S  "Select or drop a part."

: Free-toggle

     free-motion? not is free-motion?
;
press F   "Free motion (objects are insubstantial) toggle"

\ : Single-step-alias  Single-step
\ ;
\ press S  "(Space is rebound)"


\ ***********************
\ Optimized control from C
\ ***********************

\* You can use precompiled shifts in a loop with occasional display to
get desired shifts, or you can try using "shift-part" with variable
arguments to perform dynamic shifts.  In this case you might
precompile a selection of small shifts for efficiency. *\

"" cam_server.o clink

: phantom_startup (s xdim ydim zdim sx sy sz -- )
     6 reverse _cam_startup 3drop 3drop ;
: phantom_shutdown _cam_shutdown ;
: phantom_get_desired_offset _get_desired_offset ;
: phantom_set_offset (s ax ay az -- ) 3 reverse _set_actual_offset 3drop ;

33 constant user_x
33 constant user_y
33 constant user_z

false constant phantom-initialized?

: Start-phantom

          phantom-initialized? not
     if
          X Y Z user_x user_y user_z phantom_startup
          true is phantom-initialized?
          ." Please start the phantom client. "
```

```
        then
;
press A   "Start the Phantom server."

: Stop-phantom

                phantom-initialized?
        if
                phantom_shutdown
                false is phantom-initialized?
        then
;
press Z   "Shutdown the Phantom server."

: phantom-bye  Stop-phantom cam-bye ;    this is bye


: dx _offset @ ;
: dy _offset la1+ @ ;
: dz _offset 2 la+ @ ;

: phantom-server

        phantom-initialized? not  if exit then

        phantom_get_desired_offset
        dx dy dz move-xyz ( dx' dy' dz' )
        phantom_set_offset
;
this is while-stopped


\ Start-phantom
init-tables
Initial.test.pattern
when-starting
```

Bibliography

[1] Hans D. Hoeg. Development of virtual objects of arbitary shape. SB thesis, Massachusetts Institute of Technology, Department of Mechanical Engineering, May 1994.

[2] Norman Margolus. A bridge of bits. In *PhysComp '92*, pages 253 – 257. IEEE Press, 1993.

[3] Norman Margolus. CAM-8: a computer architecture based on cellular automata. In Lawriczak and Kapral, editors, *Pattern Formation and Lattice-Gas Automata*. Addison-Wesley, 1996.

[4] Norman Margolus. Crystalline computation. In *Feynman and Computation*. Addison-Wesley, 1998.

[5] Thomas H. Massie. Design of a three degree of freedom force-reflecting haptic interface. SB thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1993.

[6] Thomas H. Massie. Initial haptic explorations with the phantom: Virtual touch through point interaction. MS thesis, Massachusetts Institute of Technology, Department of Mechanical Engineering, February 1996.

[7] Thomas H. Massie and J. Kenneth Salisbury. The PHANToM haptic interface: A device for probing virtual objects. In *ASME International Mechanical Engineering Exposition and Congress*, Chicago, November 1994.

[8] Gunter Niemeyer. *Using Wave Variables in Time Delayed Force Reflecting Teleoperation*. PhD thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, September 1996.

[9] Mark P. Ottensmeyer and J. Kenneth Salisbury. Hot and cold running vr: adding thermal stimuli to the haptic experience. In *Proceedings of the Second PHANToM Users Group Workshop, A.I. Technical Report No.1617*, Massachusetts Institute of Technology, December 1997.

[10] K. Salisbury, D. Brock, T. Massie, N. Swarup, and C. Zilles. Haptic rendering: Programming touch interaction with virtual objects. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, Monterey, California, 1995.

[11] K. Salisbury and T. Massie. The PHANToM haptic interface. In *Proceedings of the AAAI Spring Symposium Series: Toward Physical Interaction and Manipulation, Working Notes*, Stanford University, March 1994.

[12] K. Salisbury and M. Srinivasan. Phantom-based haptic interactions with virtual objects. *IEEE Computer Graphics and Applications*, pages 6–10, September-October 1997.

[13] Adam Seeger, Jun Chen, and Russell M. Taylor II. Controlling force feedback over a network. In *Proceedings of the Second PHANToM Users Group Workshop, A.I. Technical Report No.1617*, Massachusetts Institute of Technology, December 1997.

[14] SensAble Technologies, Inc., Cambridge. *Basic I/O Programming Library for the PHANToM Haptic Interface*, 1997.

[15] Tommaso Toffoli. Fine-grained parallel supercomputer. Final Report PL-TR-95-2013 7, Step Research, Step Research, Cambridge, MA, July-October 1984.

[16] C. B. Zilles and J. K. Salisbury. A constraint-based god-object method for haptic display. In *Proceedings of the 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Pittsburgh, August 1995.