2014

Enabling natural interaction for virtual reality

Ryan Andrew Pavlik
Iowa State University

Enabling natural interaction for virtual reality

by

Ryan Andrew Pavlik

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements of the degree of
DOCTOR OF PHILOSOPHY

Co-majors: Human–Computer Interaction; Computer Science

Iowa State University
Ames, Iowa
2014

DEDICATION

To Katie and Amaya, for their guidance, support, and love.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENT

I would like to express my appreciation for my co-major professors, Judy M. Vance and Leslie Miller, as well as those who participated in my committee during my time at Iowa State University, Stephen Gilbert, Jonathan Kelly, David Weiss, Horea Ilies, Debra Satterfield, and Srinivas Aluru. Their guidance and input was valuable as I completed the research work performed during my assistantship and the work discussed in this dissertation.

I am grateful to my colleagues at the Iowa State University Virtual Reality Applications Center for their assistance with evaluation and feedback on my research. Thanks to Kevin Godby for his contributions to the development of the new, modern LaTeX ISU thesis class used to typeset this document. I would additionally like to thank William McNeely for technical assistance and expertise in ensuring a physically-based simulation platform.

I appreciate the support for this work through research assistantships with Judy M. Vance, and the support of the Joseph C. and Elizabeth A. Anderlik Professorship at Iowa State University. I am grateful for the recognition by my graduate program in recommending me, and to the Iowa State Foundation and the Iowa State University Graduate College for support through the Brown Graduate Fellowship. This work was performed at the Virtual Reality Applications Center at Iowa State University.

Finally, I am grateful for the support, patience, and sacrifice of Katie and Amaya Pavlik as I completed work on this dissertation.

Copyrights and Disclaimers

National Science Foundation (NSF)

This material is based upon work supported by the National Science Foundation under grants #CMMI-0928774 and #CMMI-1061458. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.[1]

Institute of Electrical and Electronics Engineers (IEEE)

[1] NSF Grantee Acknowledgment of Support and Disclaimer for grants #CMMI-0928774 and #CMMI-1061458, per NSF *Award & Administration Guide*, publications NSF 09-29 and NSF 10-1 respectively, online at `http://www.nsf.gov/pubs/policydocs/pappguide/nsf09_29/aag0929print.pdf` and `http://www.nsf.gov/pubs/policydocs/pappguide/nsf10_1/aagprint.pdf`, accessed August 8, 2014.

[2] IEEE copyright permission for use in a thesis/dissertation, per Permissions FAQ, online at `http://www.ieee.org/publications_standards/publications/rights/permissions_faq.pdf`, revised April 2013, accessed August 8, 2014.

ASME

A copyright permission request was submitted to ASME August 8, 2014. The following reply with permission grant was received August 12, 2014.

Dear Mr. Pavlik,

It is our pleasure to grant you permission to publish all or any part of the following ASME papers:

- *R.A. Pavlik and J.M. Vance, "A Modular Implementation of Wii Remote Head Tracking for Virtual Reality," in ASME 2010 World Conference on Innovative Virtual Reality (WINVR 2010), 2010, pp. 351–359. doi:10.1115/WINVR2010-3771 Copyright ©2010 by ASME*
- *R.A. Pavlik and J.M. Vance, "Expanding Haptic Workspace for Coupled-Object Manipulation," in ASME 2011 World Conference on Innovative Virtual Reality (WINVR 2011), 2011, pp. 293–299. doi:10.1115/WINVR2011-5585 That publication is Copyright ©2011 by ASME*
- *R.A. Pavlik, J.M. Vance, and G.R. Luecke, "Interacting with a Large Virtual Environment by Combining a Ground-based Haptic Device and a Mobile Robot Base," in ASME 2013 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, 2013. doi:10.1115/DETC2013-13441 Copyright ©2013 by ASME*

as cited in your letter for inclusion in a doctoral dissertation (in the form of multiple papers) to be published by Iowa State University.

Permission is granted for the specific use as stated herein and does not permit further use of the materials without proper authorization. Proper attribution must be made to the author(s) of the materials. Please note: if any or all of the figures and/or Tables are of another source, permission should be granted from that outside source or include the reference of the original source. ASME does not grant permission for outside source material that may be referenced in the ASME works.

As is customary, we request that you ensure proper acknowledgment of the exact sources of this material, the authors, and ASME as original publisher. Acknowledgment must be retained on all pages printed and distributed.

Many thanks for your interest in ASME publications.

Sincerely,
Beth Darchi
Publishing Administrator
ASME
2 Park Avenue, 6th Floor
New York, NY 10016-5990
Tel 1.212.591.7700
darchib@asme.org

ABSTRACT

This research focuses on the exploration of software and methods to support natural interaction within a virtual environment. Natural interaction refers to the ability of the technology to support human interactions with computer generated simulations that most accurately reflect interactions with real objects. Over the years since the invention of computer-aided design tools, computers have become ubiquitous in the product design process. Increasingly, engineers and designers are using immersive virtual reality to evaluate virtual products throughout the entire design process.

The goal of this research is to develop tools that support verisimilitude, or likeness to reality, particularly with respect to human interaction with virtual objects. Increasing the verisimilitude of the interactions and experiences in a virtual environment has the potential to increase the external validity of such data, resulting in more reliable decisions and better products.

First, interface software is presented that extends the potential reach of virtual reality to include low-cost, consumer-grade motion sensing devices, thus enabling virtual reality on a broader scale. Second, a software platform, VR JuggLua, is developed to enable rapid and iterative creation of natural interactions in virtual environments, including by end-user programmers. Based on this software platform, the focus of the rest of the research is on supporting virtual assembly and decision making. The SPARTA software incorporates a powerful physically-based modeling simulation engine tuned for haptic interaction. The workspace of a haptic device is both virtually expanded, though an extension to the bubble technique, and physically expanded, through integration of a haptic device with a multi-directional mobile platform. Finally, a class of hybrid methods for haptic collision detection and response is characterized in terms of five independent tasks. One such novel hybrid method, which selectively restores degrees of freedom in haptic assembly, is developed and assessed with respect to low-clearance CAD assembly. It successfully maintains the high 1000 Hz update rate required for stable haptics unlike previous related approaches.

Overall, this work forms a pattern of contributions towards enabling natural interaction for virtual reality and advances the ability to use an immersive environment in decision making during product design.

CHAPTER 1. GENERAL INTRODUCTION

This dissertation synthesizes research in a number of related areas that follow from the general theme of enabling natural interaction for virtual reality. The particular kind of "natural interaction" varies according to context, but is typically taken to mean verisimilitude: making some aspect of a computerized environment more like the real world. The motivating context for the overall work is supporting decision making in engineering. Data obtained in virtual experiences is used in the decision making process, and increasing the verisimilitude of the interactions and experience has the potential to increase the external validity of such data.

Several threads of investigation emerged in developing this work for both Human-Computer Interaction and Computer Science. These threads are intentionally multidisciplinary and at times divergent; however, they remain complementary and linked to an idea of natural interaction. The research contributions presented take varied forms, as suits an intentionally multidisciplinary approach. They include not only published advancements in the literature of their respective fields, but also software components developed during or as the result of the research process. The software components developed, often released as open source software or contributed as improvements to existing open source projects, serve a dual purpose. They represent a public advancement of the state of the art in a tangible, applied way. Furthermore, when they have been released publicly, they contribute to the integrity of our application of the scientific method. In a research field where so many results follow from software, that software becomes an important part of our methods and its availability becomes key to reproducibility of our results [5].

1.1. Dissertation Organization

This dissertation takes the "manuscript" format, in which, with the exception of this chapter and Chapter 9, each following chapter is a self-contained paper, with its own introduction, conclusion, and bibliography. The purpose of this overall introduction and of the overall conclusion in Chapter 9 is to highlight the cohesive research threads as woven through the chapters. As already noted, the title "Enabling Natural Interaction for

Virtual Reality" is the key to this cohesive whole, while the chapters interpret the title and emphasize portions of it in differing ways.

The work included starts with Chapter 2, which presents an advancement in virtual reality's potential reach, *enabling virtual reality* at a broader scale. At the time of its composition, the Nintendo Wii Remote had been recently released and was the first of a wave of low-cost, consumer-grade motion sensing devices intended for gaming. Work had hit the hobbyist scene using the infrared sensor in the Wii Remote along with a simple modification to commercially-available lighted safety glasses to provide head-tracked perspective [6], an important component of virtual reality [1], in a fairly simplistic virtual environment. The hardware required for such a setup was minimal and simple to construct. In academic research, more complex constellations of multiple Wii Remotes and/or LED infrared emitters were being investigated for use in more conventional virtual environments [7, 2]. My contribution was to join the two clusters of investigation, by designing and implementing a driver for VRPN [8] that allowed use of the Wii Remote and the simple two-point tracked safety glasses with commercial and academic virtual reality software that accepts VRPN tracker input, a de facto standard.

The software presented in Chapter 3, VR JuggLua, is the base for the work in the remaining chapters in addition to being a contribution on its own. Beyond just a Lua binding for VR Juggler, it provides for interactive virtual environment creation through an interactive code entry console during execution of an immersive application, provides an embedded domain-specific language for scene graph creation, and more. It furthers the goal of the dissertation in two ways. On one level, the domain-specific language and other features for the programmer-user provide a more natural interaction with environment construction; in this way it enables *natural interaction through usability for programmers, especially end-user programmers*. On another level, the more natural, iterative development techniques possible with VR JuggLua allow faster iteration in *development of interaction techniques for virtual reality*.

The narrower focus on virtual assembly simulations begins with Chapter 4. Virtual assembly, particularly with haptic force feedback, is a tool for simulating assembly, disassembly, and maintenance procedures through manipulation of CAD models in a virtual environment. The uses and goals of virtual assembly are many, but often include evaluating designs and capturing user knowledge as individuals interact with a given assembly situation. Thus, virtual assembly relies on natural interaction, in the sense of realistic, lifelike interaction, to arrive at correct conclusions and provide the best input to the engineering decision making process. Haptic force feedback devices add an additional sensory output beyond simply visual virtual reality, permitting the simulation of weight, inertia, and forces and torques from part-to-part or part-to-environment interaction. Including haptic force feedback supports natural interaction, but the devices required can limit it. Due to the need to deliver absolute forces, haptic devices are typically grounded by being affixed to a single spot in the real world. Their working area is often much smaller than the working area that an environment is trying to simulate. Chapters 4 and 5 address the limited workspace problem from two different angles. A method of virtually expanding the workspace of a haptic device is discussed in Chapter 4. Known as the "bubble technique," this hybrid position-rate control scheme provides a zone of 1-1 interaction in the main workspace of the haptic device, and responds to movement toward the edges (past a spherical boundary) by moving the virtual environment to shift the effective virtual workspace. It was presented in [3] in the context of another use of haptic devices: exploration of and interaction with simulated surfaces. In applying it to virtual assembly situations involving grasping and moving objects, we discovered a number of issues. This chapter presents those issues, as well as one technique to address some of them. While a virtual workspace expansion is not fully "natural," interaction with an artificially small region of the virtual environment is much less so. Thus, this work to advance virtual workspace expansion furthers *natural interaction in virtual assembly scenarios by addressing limitations of force feedback devices and techniques*.

A different approach to expanding the workspace of haptic devices is presented in Chapter 5. In this work, a novel integration of an omni-directional mobile robot base with a conventional grounded haptic device together with control algorithms and software systems are developed to physically enlarge the workspace. Using the system presented in this chapter, a user can move throughout a large-volume virtual reality system such as a CAVE while using the haptic device, and the mobile robot will transparently move to keep the base of the haptic device in a neutral position relative to the end effector. The experience is as if the workspace of the haptic device were physically enlarged greatly while maintaining the other properties of the haptic device as constructed. The contribution here is the combined fully-functioning system providing an enlarged workspace, with a number of custom software components, custom firmware on the mobile robot, and integration between the components. Clearly, a *larger workspace for haptic interaction* furthers natural interaction for virtual environments.

Chapters 6, 7, and 8 delve deeper into a particular aspect of haptic virtual assembly. One challenge in this field is handling assembly of parts with low clearances between them. Fundamentally, this challenge arises from collision detection and response algorithms that require the use of tessellations of the original precise models in order to achieve high enough performance to permit stable haptic force feedback. These tessellations result in artifacts that artificially limit the degrees of freedom that should be present between two parts. These three chapters elaborate on aspects of a hybrid technique that combines the use of a fast tessellation-based algorithm with knowledge of the precise original model. The overall goal of this work is to permit natural interaction with low-clearance CAD models loaded directly into a virtual assembly system, with no manual preprocessing steps. The work presented here builds on the basic process presented in [4], where the idea is to use an tessellated geometry algorithm for the bulk of the interaction while looking up the corresponding precise representation, recognizing geometric constraints, and responding to them to improve interaction particularly during low-clearance tasks.

An overview of the data structures and algorithms involved in my work building on this hybrid method, with a view to their computational complexity and specific optimizations applied to make the algorithm more efficient, is presented in Chapter 6. The novel design of the software for one particular step of the process, that recognizes constraints given two sets of boundary representations, forms the contribution in Chapter 7. Here again the usability is on two levels: both maintainability and extensibility for the developer, and supporting the broader hybrid method for natural low clearance assembly. Finally, Chapter 8 presents the full process as a two-level hierarchy of research tasks from a more engineering-based perspective. It also presents a fundamental distinction from [4] in how this work makes use of the knowledge of geometric constraints within the run-time portion of the hybrid algorithm. Rather than attempting to form a constraint system, solve it, and combine those results with the tessellated geometry algorithm at a global, per-colliding-part-pair level, Chapter 8 directly addresses the artificial removal of degrees of freedom incurred by the tessellation. Individually and as a group, these chapters presenting advancements to a hybrid collision detection and response algorithm work toward the goal of *realistic interaction with CAD models in a virtual environment.*

Bibliography

[1] Kevin W. Arthur, Kellogg S. Booth, and Colin Ware. Evaluating 3D task performance for fish tank virtual worlds. *ACM Transactions on Information Systems*, 11(3):239–265, 1993. ISSN 1046-8188. doi:10.1145/159161.155359. URL http://dl.acm.org/citation.cfm?doid=159161.155359.

[2] Yang-Wai Chow. Low-Cost Multiple Degrees-of-Freedom Optical Tracking for 3D Interaction in Head-Mounted Display Virtual Reality. *International Journal of Recent Trends in Engineering*, 1(1):52–56, 2009. ISSN 1797-9617.

[3] Lionel Dominjon, Jérôme Perret, and Anatole Lécuyer. Novel devices and interaction techniques for human-scale haptics. *The Visual Computer*, 23(4):257–266, March 2007. ISSN 0178-2789. doi:10.1007/s00371-007-0100-4. URL http://link.springer.com/10.1007/s00371-007-0100-4.

[4] Daniela Faas. A Hybrid Method for Haptic Feedback to Support Manual Virtual Product Assembly. In *ASME 2011 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, pages 1–8, Washington, DC, USA, 2011. ASME. doi:10.1115/DETC2011-47665. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1639771.

[5] Luis Ibanez, William J. Schoeder, and Marcus D. Hanwell. Practicing Open Science. In Victoria Stodden, Friedrich Leisch, and Roger D. Peng, editors, *Implementing Reproducible Resear*, chapter 9, pages 241–281. Chapman and Hall/CRC, 2014. ISBN 978-1466561595. URL https://osf.io/s9tya/files/.

[6] Johnny Chung Lee. Hacking the Nintendo Wii Remote. *IEEE Pervasive Computing*, 7(3):39–45, July 2008. ISSN 1536-1268. doi:10.1109/MPRV.2008.53. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4563908.

[7] Torben Sko and Henry J. Gardner. The Wiimote with multiple sensor bars: creating an affordable, virtual reality controller. In *Proceedings of the 10th International Conference NZ Chapter of the ACM's Special Interest Group on Human-Computer Interaction - CHINZ '09*, pages 41–44, New York, New York, USA, 2009. ACM Press. ISBN 978-1-60558-574-1. doi:10.1145/1577782.1577790. URL http://portal.acm.org/citation.cfm?doid=1577782.1577790.

[8] Russell M. Taylor II, Thomas C. Hudson, Adam Seeger, Hans Weber, Jeffrey Juliano, and Aron T. Helser. VRPN: a device-independent, network-transparent VR peripheral system. In *Proceedings of the ACM symposium on Virtual reality software and technology - VRST '01*, page 55, New York, New York, USA, 2001. ACM Press. ISBN 1581134274. doi:10.1145/505008.505019. URL http://portal.acm.org/citation.cfm?doid=505008.505019.

CHAPTER 2. A MODULAR IMPLEMENTATION OF WII REMOTE HEAD TRACKING FOR VIRTUAL REALITY

A paper published in the proceedings of the *ASME 2010 World Conference on Innovative Virtual Reality (WINVR 2010)*

Ryan A. Pavlik, Judy M. Vance

Abstract

Virtual reality (VR) environments based on interactive rendering of 3D computer graphics often incorporate the use of position and orientation tracking on the user's head, hands, and control devices. The Wii Remote game controller is a mass-market peripheral that can provide a low-cost source of infrared point tracking and accelerometer data, making it attractive as a PC-based virtual reality head tracking system. This paper describes the development of an extension to the Virtual Reality Peripheral Network (VRPN) software to support the use of the Wii Remote game controller as a standard tracker object in a wide range of VR software applications. This implementation permits Wii Remote-based head tracking to directly substitute for more costly commercial trackers through the VRPN and VR Juggler Gadgeteer tracker interfaces. The head tracker provides up to 100Hz of head tracking input. It has been tested in a variety of VR applications on both Windows and Linux. The discussed solution has been released as open-source software.

2.1. Introduction

Position and orientation tracking is essential to virtual reality. Head tracking, in particular, allows the calculation of an off-axis perspective projection for interactive three-dimensional (3D) computer graphics. Such a display, whether monoscopic or stereoscopic,

Ryan A. Pavlik was the primary author and researcher.
R.A. Pavlik and J.M. Vance, "A Modular Implementation of Wii Remote Head Tracking for Virtual Reality," in *ASME 2010 World Conference on Innovative Virtual Reality (WINVR 2010)*, 2010, pp. 351–359. doi:10.1115/WINVR2010-3771
Copyright ©2010 by ASME

provides enhanced depth and distance cues to the user, increasing the sense of immersion and improving task performance [2, 1]. Immersive virtual reality displays can be used in the engineering and design process to allow human interaction with computer-aided design (CAD) data sets more naturally than non-immersive applications.

Recent developments in consumer game systems, such as the Nintendo Wii, have introduced tracking peripherals and game mechanics to a broad audience. The availability of such systems has also presented academic and hobby researchers with access to the Wii Remote hardware, which integrates linear accelerometers, infrared point tracking, digital and analog game controls, and a Bluetooth wireless interface into an affordable single game controller device.

The methods presented here result in a novel modular extension to the Virtual Reality Peripheral Network (VRPN) software that supports the use of the Wii Remote hardware for virtual reality head tracking. This work enables easy implementation of Wii Remote tracking with slightly modified commercial off-the-shelf (COTS) hardware for use within research-grade virtual reality frameworks. It serves as the tracking component of a simple virtual reality system that nonetheless supports a range of existing virtual reality applications designed for more sophisticated tracking systems.

2.2. Motivation

The present research is motivated by the need for low-cost VR head tracking. As stereo televisions, stereo movies, game controllers, and 3D interaction devices become more affordable and commonly available, there is the opportunity to leverage these technological advances to support a broad range of virtual reality applications. Current magnetic, ultrasonic, and optical tracking systems are very expensive to consider for wide use. Low-cost VR head tracking would greatly enhance the immersive experience of some of these common stereo configurations.

At Iowa State University, researchers are also developing VR applications to encourage K-12 students to consider careers in science, technology, engineering and math (STEM). Outreach efforts in K-12 education require the availability of an inexpensive, portable tracking solution that can be set up and taken down quickly, easily transported from the lab to presentation sites, and operated independently of the physical configuration of the space. Another area of need is to supplement existing small desktop commercial magnetic trackers for research purposes. These trackers only provide two channels of tracking. Often, desktop VR requires three channels of tracking to accommodate two-handed tracking and head tracking. The ability to combine the existing magnetic tracker with low-cost head tracking provides a cost effective solution for three-channel requirements.

2.3. Background

Pintaric et al. [12] developed an optical position tracking system using infrared light and commercially-available hardware. Their system relies on multiple cameras that are synchronized with a pulsed infrared flash mounted on each camera. Arrangements of retro-reflective, passive markers are placed on position-tracked objects. The goal of their work is to provide broad six degree-of-freedom (6DOF) position tracking for augmented reality and immersive virtual reality, within a room-sized space.

Work by J. C. Lee into novel applications of the Wii Remote hardware served to drive a variety of further research into using the Wii Remote for position tracking. Lee wrote a number of applications, including a sample head tracking application [10], using B. Peek's C# library for access to the Wii remote data under Windows [11]. His solution tracks two IR light-emitting diodes (LEDs) attached to the sides of a pair of eyeglasses. He uses the position data to calculate an off-axis projection that he renders on the user's standard computer monitor using DirectX. Lee's software is widely available and has introduced the potential for using the Wii Remote as a position tracker to a wide audience; however, his software is limited in scope and not easily incorporated into other VR applications.

Lee's head tracking application attaches the IR LEDs to the dynamic object (the user) while keeping the remote itself static. This is unlike the intended use of the Wii game console in which a "Sensor Bar" (actually a pair of IR light sources located a known distance apart) is placed in a stationary location above or below the user's television. In such a setup, moving the Wii Remote itself drives the interaction. As Lee explains, a setup with a fixed Wii Remote and moving IR markers instead results in a system that is primarily sensitive to translation, and less sensitive to orientation, which suits the needs of desktop VR head tracking. A strength of his solution is the total ease-of-use, including the fabrication of the required hardware. Even the LED-augmented glasses are commercially available, and can be modified for infrared use by simply swapping the LEDs.

T. Sko and H. Gardner designed a configuration to use a hand-held Wii Remote as a pointing controller in a two-walled virtual reality theatre [13, 14]. They placed multiple sensor bars in the environment to provide a wider usable range for the device. Their solution uses an agent-based algorithm to continually maintain a model of the controller's pose, based on the known physical location of the multiple sensor bars. The Wii Remote can only track up to four points simultaneously, so they spaced the sensor bars to avoid greater than four visible IR points at any time.

Chow [6, 5] details a system using two Wii Remotes to provide both head tracking and controller input to a virtual environment running on a head-mounted display. As with the work by Lee, Chow uses a stationary Wii Remote observing head-mounted LED markers to provide head tracking input. In this effort, four IR markers are used to provide increased input data sufficient to drive the POSIT algorithm [7] and provide accurate pose estimation. The research also included a verification of the poses returned by the Wii Remote-based tracker by comparison with pose estimates from a commercial magnetic tracker. The system was able to produce estimates with less than one centimeter of average position difference and less than one degree of average orientation difference.

Work by Wognum [17] permits Wii Remote head tracking using VRPN on Windows only. His work ports the algorithm used by Lee to the VRPN package, using it as a versatile platform for implementing tracking software. The algorithm interacts directly with the Windows-specific WiiYourself! software library to access the Wii Remote, and uses only the IR point data returned by the controller. As a VRPN tracker driver, it provides compatibility with a variety of VR frameworks. At compile-time, the user must specify whether the remote is above or below the display, as a proxy for the orientation of the Wii Remote and its coordinate system. It requires the user to initiate a reset of the pose estimate through interaction with the tracker server if an inaccurate projection is noticed. It does not use any accelerometer or gravity measurements.

The solution presented here builds on Lee's configuration. It differs from previous work in that a standard software interface is designed which allows the Wii Remote to interface easily with existing academic and commercial VR applications built on open platforms. The input interface interacts with a Wii Remote over an existing standard protocol to provide additional flexibility. In this way, it can also accept recorded or simulated Wii Remote data, or even data from an alternate device preprocessed by a point-tracking computer vision system. This method supports the use of inexpensive hardware, coupled with the Wii Remote, to provide low-cost position tracking in place of more sophisticated commercial tracking devices.

2.4. Design Considerations

The design of the solution discussed here was primarily driven by the dual need for tracking in outreach programs and supplementary tracking for research applications. An analysis of the functional requirements led to the selection of three principles to guide the development of the solution.

2.4.1. Limited Scope

The scope was limited to head tracking only, based on the motivating needs. A magnetic tracker can handle hand tracking for research applications, and outreach applications only need head-tracking at this time. This allowed an enumeration of assumptions and constraints to simplify implementation and limit system complexity. Tracked orientations and positions could be restricted to 3DOF translation with rotation about a sensor-normal axis for presentation of valid stereo pairs. The importance of translation was emphasized. Occlusion considerations were minimized, since occlusion of the tracked glasses would imply visual occlusion of the display for the user.

2.4.2. Compatibility

In order to satisfy the dual need for Wii Remote-based head tracking, the solution needed to be compatible with existing VR frameworks, especially VR Juggler. In order to support multiple existing VR frameworks in the same manner as commercial trackers, the application-facing side of the tracker driver needed to be sufficiently abstracted so as to directly substitute for other, more sophisticated trackers. This included presenting a full rigid-body 6DOF pose, even though the basic algorithm presently only supports calculating 4DOF. Unlike the tracking software presented by Lee, this abstraction resulted in a tracker that is fully independent of the actual virtual environment being executed. Furthermore, the output of the tracking module does not necessarily need to drive a head-coupled perspective view. It can also be compared to known values for verification and tuning.

2.4.3. Flexibility and Modularity

The software design principles of modularity and loose coupling were important in the design process to ensure maximum flexibility of the final system. On the Wii Remote-facing side of the tracker driver, accessing data via a `vrpn_Analog_Remote` interface provides loose coupling. In most live uses, the provider of the data is the `vrpn_WiiMote` device that exposes the state of a Wii Remote on Windows and Linux through `vrpn_Analog` channels. However, as any data source matching this interface will suffice, there are a variety of possibilities for data sources. It permits replacement of the underlying Wii Remote access library to permit a more cross-platform implementation. Furthermore, the algorithm's implementation is not strictly tied to the Wii Remote. The Wii Remote's prevalence, high refresh rate, and built-in computer vision processing made it a convenient choice as a data source. However, a computer vision system processing web-cam input could also be implemented for use with the tracker.

The tracker operates on a number of values which lie in a known range, so an ideal solution would accept any source of suitable values. A useful implication of this is that verification of the software's latency, accuracy, and precision may be performed by substituting the source of values. A "virtual Wii Remote" may be implemented that calculates the expected sensor data for a given pose, then feeds that data into the tracker to permit comparison of the output and input.

Implementation of alternative pose estimation techniques also suggested an emphasis on modularity. Modularity permits replacement of the basic two LED tracking with a more sophisticated algorithm utilizing Kalman filtering [8], model evaluation, or more advanced techniques to provide a more robust response. The two LED system may even be substituted by a three LED tracker to disambiguate screen-orthogonal translation from rotation around a vertical axis.

Figure 2.1 shows the potential producers and consumers of the generic data interfaces manipulated by the tracking driver. The contribution described in this article is represented by the green "Tracking Driver Module" box. The primary producer of input data is the live Wii Remote. The primary consumers of tracking data are the user applications shown in orange. User applications written to use VRPN directly may directly consume the tracking output, while the input abstractions of the VR Juggler and Vizard systems serve as direct consumers for their respective user applications. The two pairs of sources and sinks connected by dotted lines are automated testing and verification capabilities made possible by the design features of the described solution.

2.5. Software Platform

This work builds on a number of existing software systems. The existing virtual assembly application motivating development is written using VR Juggler, an open-source virtual platform for virtual reality software development available at http://www.vrjuggler.org [3]. This framework provides a cross-platform method of developing virtual environments in C++ that can be run on multiple operating systems and on virtual reality hardware ranging from standard desktop workstations to immersive stereoscopic projection environments.

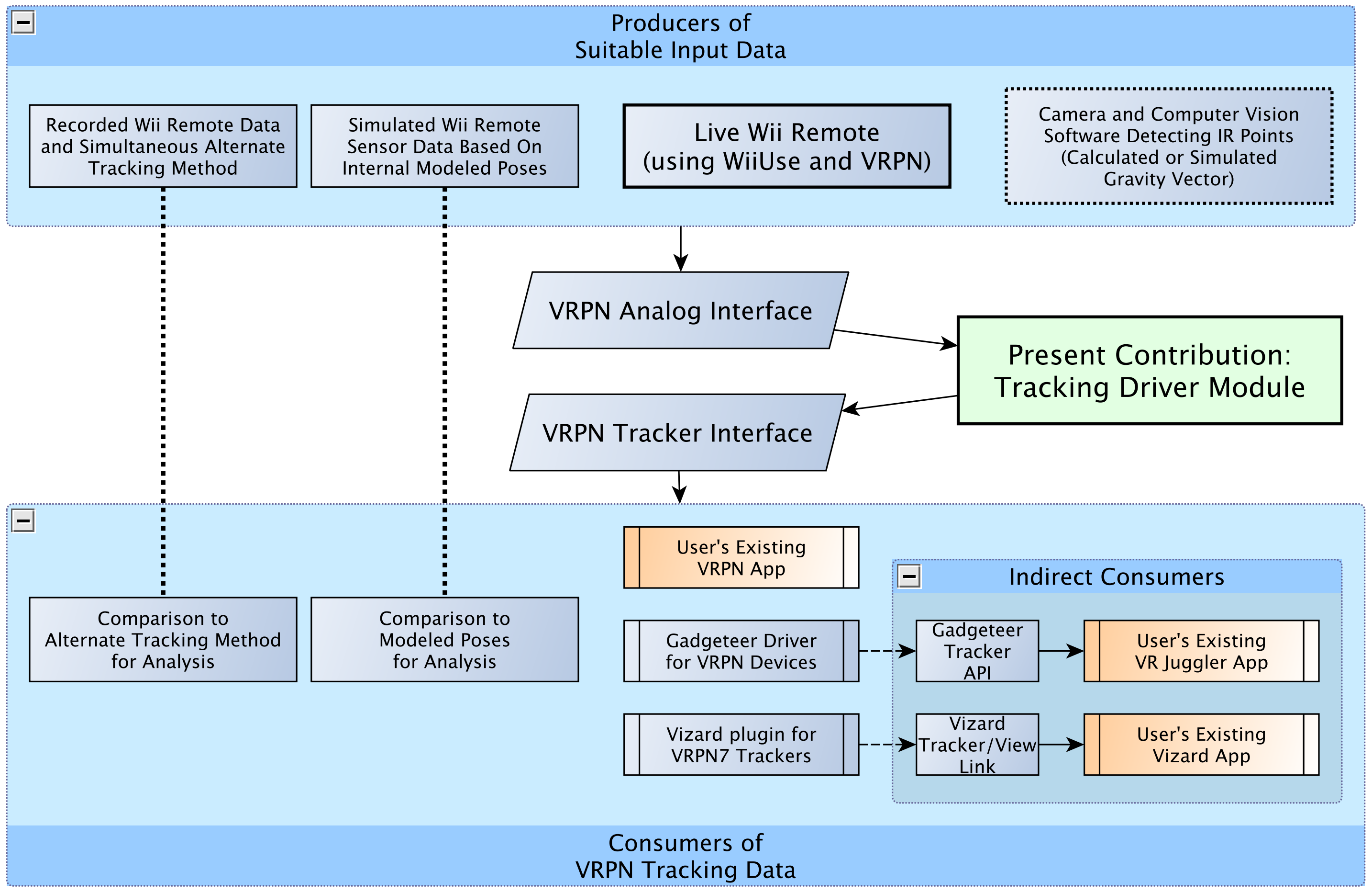

Producers of
Suitable Input Data
Recorded Wii Remote Data and Simultaneous Alternate Tracking Method
Simulated Wii Remote Sensor Data Based On Internal Modeled Poses
Live Wii Remote (using WiiUse and VRPN)
Camera and Computer Vision Software Detecting IR Points (Calculated or Simulated Gravity Vector)
VRPN Analog Interface
Present Contribution: Tracking Driver Module
VRPN Tracker Interface
User's Existing VRPN App
Gadgeteer Driver for VRPN Devices
Vizard plugin for VRPN7 Trackers
Indirect Consumers
Gadgeteer Tracker API
User's Existing VR Juggler App
Vizard Tracker/View Link
User's Existing Vizard App
Comparison to Alternate Tracking Method for Analysis
Comparison to Modeled Poses for Analysis
Consumers of
VRPN Tracking Data

Figure 2.1: Flexibility in sources and sinks for the core tracking algorithm

Vizard is a commercial virtual reality software platform similar in scope to VR Juggler. It provides virtual reality hardware access and scenegraph structures for VR. Vizard applications are written in Python, an interpreted, dynamically-typed language.

The Wii Remote head tracking software itself is implemented using C++ as a module for the Virtual Reality Peripheral Network (VRPN) software package available at `http://www.vrpn.org`. This open-source system provides a transparently networked, generic interface to a wide range of VR devices and input systems [15]. VRPN is used directly in a range of VR applications. Generic VRPN interfaces can also be accessed through a Gadgeteer driver included with VR Juggler, or through the VRPN7 extension included with Vizard. As such, any tracker, analog input, or digital input (button) device accessible using VRPN can also be used in applications built on VR Juggler.

VRPN 07.26 provides a "WiiMote" device exposing the raw data returned using the WiiUse open-source library [9] to access the Wii Remote hardware over Bluetooth in Windows and Linux. This driver provides a number of standard VRPN analog channels to convey the sensor coordinates and relative sizes of up to four IR points, as well as the gravity vector as returned by the three integrated linear accelerometers, among other data.

A sample tracker client application was developed to evaluate the performance of the Wii Remote tracker. Figure 2.2 shows the head tracker data visualized in this standalone application, with the position and orientation of the teapot matching the pose reported by the head tracker. This client was developed using VRPN directly to access tracker data, along with the GLUI and GLUT graphics tool-kits. In addition, the tracker was used to demonstrate a VR Juggler-based virtual assembly application in which a user can interact with CAD models in a natural, spatial way with haptic force-feedback.

2.6. Hardware Platform

The VR Juggler software allows the virtual assembly application to run on a variety of systems. The primary system used in this work is a desktop workstation with an Intel Xeon processor and an nVidia Quadro FX 1000 graphics card. Stereoscopic images are rear-projected by a DepthQ 3120 DLP projector displaying 800×600 resolution at 120Hz on an 80cm (32 inch) diagonal screen. The projector is used in conjunction with CrystalEyes active liquid crystal stereo shutter glasses and a Stereographics emitter, providing effectively 60Hz display refresh per eye. A Polhemus Patriot magnetic tracker provides two channels of 6DOF position and orientation tracking in the desktop workspace area.

Another hardware system used for undergraduate engineering education consists of two DLP projectors with circularly-polarized filters to provide passive stereo. A large

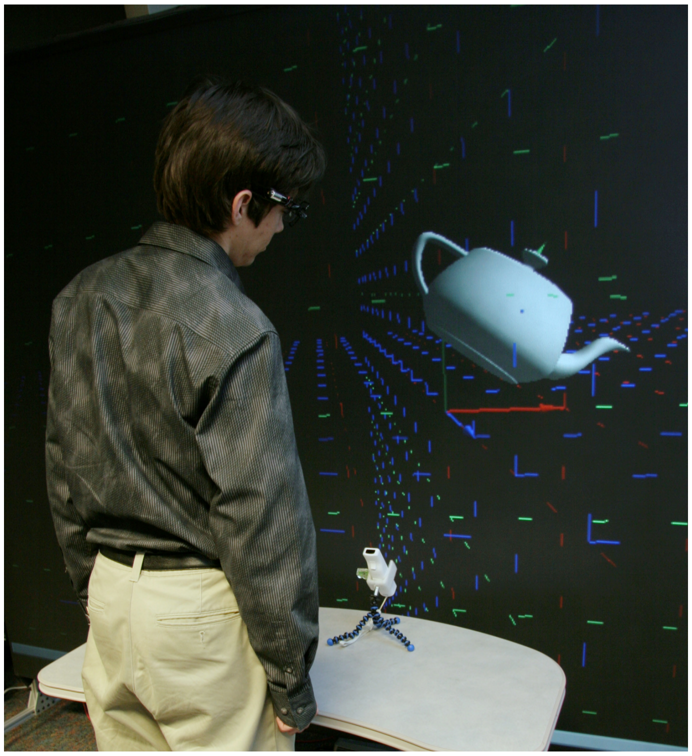


Figure 2.2: Head tracking data shown as teapot

silvered screen is used in a front-projection setup to provide a stereo viewing experience to a classroom of students. This system, also running on an Intel Xeon-based workstation with Quadro FX 3700 graphics, has no additional trackers besides the Wii Remote system described here.

A third system, consists of an Intel Xeon workstation with an nVidia Quadro FX 5800 graphics card. This system drives one of two display systems. It can display on either a 1680×1050 56cm (22 inch) diagonal widescreen LCD monitor at 60Hz providing a high-resolution mono display, or using a DepthQ 3120 projector, it can provide a stereoscopic display on a 3m (10 foot) diagonal rear-projection screen. This larger screen is used for investigating 1–1 scale interactions in virtual assembly and haptics, and presently features no pose tracking system beyond this Wii Remote system.

The present research follows Lee's approach of using a stationary Wii Remote with IR LEDs fixed to the moving object. In both the stereo and mono configuration, the Wii Remote head tracking system relies on two battery-powered infrared LEDs attached to glasses (standard eyeglasses or the CrystalEyes shutter glasses), as shown in Fig. 2.3. The distance between the LEDs, typically 15cm, must be provided in advance to the tracking software. Since this hardware configuration is similar to that used by Lee, this research extends the utility of glasses designed or modified for Lee's popular application to use in

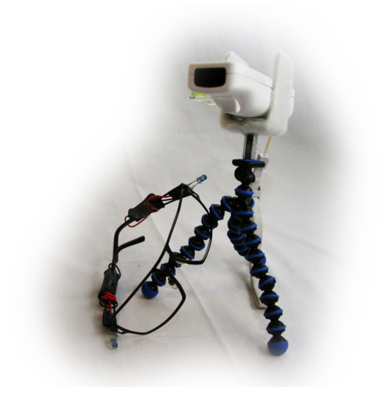


Figure 2.3: Wii Remote, tripod mount, and trackable glasses

academic and commercial virtual environment frameworks. The Wii Remote hardware itself is available commercially for about US$40. A mounting bracket was easily constructed to attach the Wii Remote to a variety of standard camera tripods.

2.7. Implementation Details

The head tracking driver is implemented as a filter device that takes, as input, any source of 15 VRPN analog channels. These channels carry the three components of the measured gravity vector and up to four 3-tuples of size and x- and y-position of a tracked point on the 1024×768 sensor. While typical usage of the driver will be in conjunction with the `vrpn_WiiMote` driver that exposes these channels among other data, this generic input interface increases the modularity of the overall tracking system by permitting substitution of the `vrpn_WiiMote` device.

Implementation of the Wii Remote head tracking driver as a VRPN module permits a variety of uses for the completed code. It can be compiled into a library for direct use by an application, if desired. A standalone server application, shown at the module level in Fig. 2.4, was produced that allows a user to start using the tracker by simply starting a single application. The server, built against the VRPN server library, creates the `vrpn_WiiMote` device that uses the WiiUse library to access the device. It then initializes the tracker module with the VRPN device name of the Wii Remote. When started in this way, the head tracker can be accessed as a standard VRPN tracker server from the local host or from across the network. VR Juggler can be configured to access this tracker device using the `VRPN_drv` driver included in the upstream software suite. The position proxy corresponding to the user's head can be set to use this driver as its data source, providing head-coupled rendering.

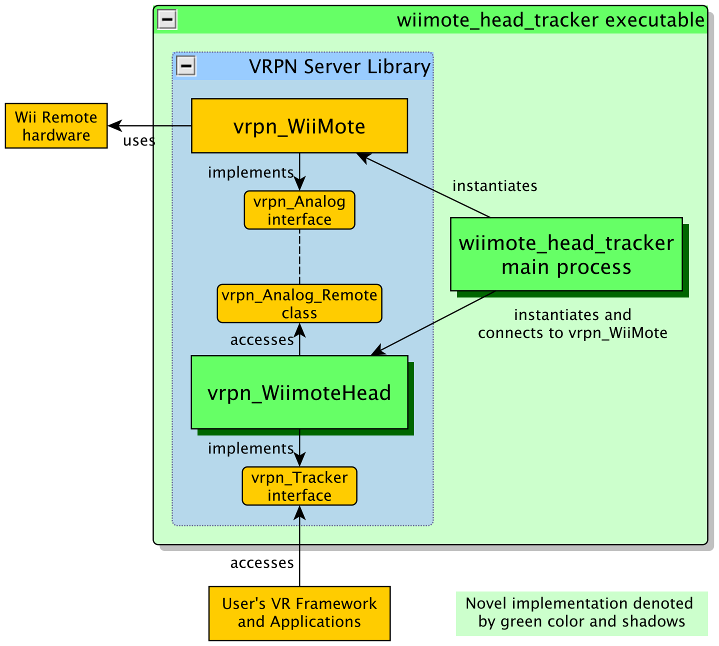

wiimote_head_tracker executable
VRPN Server Library
Wii Remote hardware
uses
vrpn_WiiMote
implements
vrpn_Analog interface
instantiates
wiimote_head_tracker main process
vrpn_Analog_Remote class
accesses
instantiates and connects to vrpn_WiiMote
vrpn_WiimoteHead
implements
vrpn_Tracker interface
accesses
User's VR Framework and Applications
Novel implementation denoted by green color and shadows

Figure 2.4: Module-level view of tracker server

Regardless of whether the tracker is standalone or part of an end-user application, when it is initialized with an analog device, it registers report callbacks for that device, so that it is notified when new data is available. When a callback is triggered, the raw data is stored in the tracker object for processing. Only the four IR points and the accelerometer data are retained: all other channels, including the buttons and any extension controllers, are ignored.

When the number of valid IR tracked points is equal to 2, calculations are performed using the fixed field of view and the known distance between the LED points to arrive at an estimate of x-, y-, and z-translation, as well as rotation about the sensor-normal z-axis. This calculation step is modular in the code, to support substitution of pose estimation techniques. Calculation proceeds in the coordinate system aligned with the Wii Remote. Later, during the VRPN main loop, the presence of new Wii Remote data or sufficient elapsed time since the last report trigger a final pose preparation and report.

To eliminate the need to adjust for the pitch and roll of the Wii Remote for every installation, an algorithm to achieve gravity correction was developed and applied before calculating the final pose. It is imperative to eliminate Wii Remote pitch and roll variability because, unlike other trackers with a more permanent installation method, the tracker base in this system is easily portable and attached to a tripod-style mount that introduces orientation variability over the long term. Implementing this correction provides a more

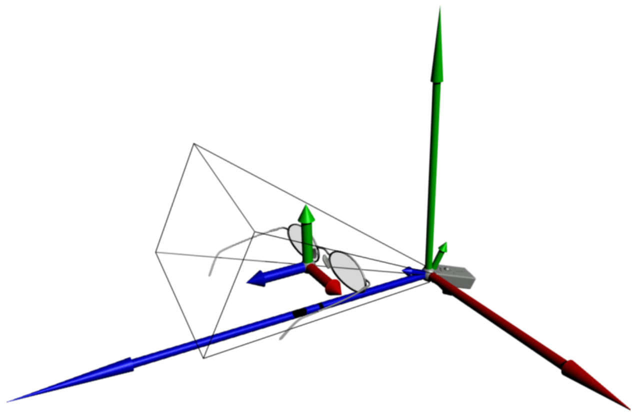


Figure 2.5: Transforms and coordinate systems

accurate tracking result without the need for a user-initiated reset, unlike the work by Wognum [17]. The stored accelerometer data is used to rotate the estimated pose to account for the pitch and roll of the Wii Remote, and by extension, its IR sensor, as shown in Fig. 2.5. The reported pose is thus relative to a virtual tracker whose z-axis is along the projection of the Wii Remote position down on to a plane perpendicular to gravity, with the y-axis directly opposed to gravity and the x-axis proceeding from their intersection in a right-hand coordinate system. Effectively, the user may consider the tracker base to be level with respect to gravity, even when it is not. This approach also permits the user to freely position the Wii Remote at an incline as needed to ensure that head movements stay within the limited field-of-view of the Wii Remote's sensor. This calculation is performed in a "just-in-time" manner to ensure that the freshest data is reported in case multiple reports from the Wii Remote were received since the last main loop execution due to system performance. Finally, the gravity-corrected pose estimate is packed and transmitted as a VRPN tracker pose update.

Figure 2.6 describes the entire tracking and VR system. The three lanes denote the three concurrent processes; the first occurs in the Wii Remote hardware, while the second and third take place on the computer's processor. The interfaces between all concurrent processes are asynchronous. The elements of the flowchart within the green box are part of the present contribution. They correspond to the steps taken by the tracking filter driver to transform the analog input into a tracker pose estimate.

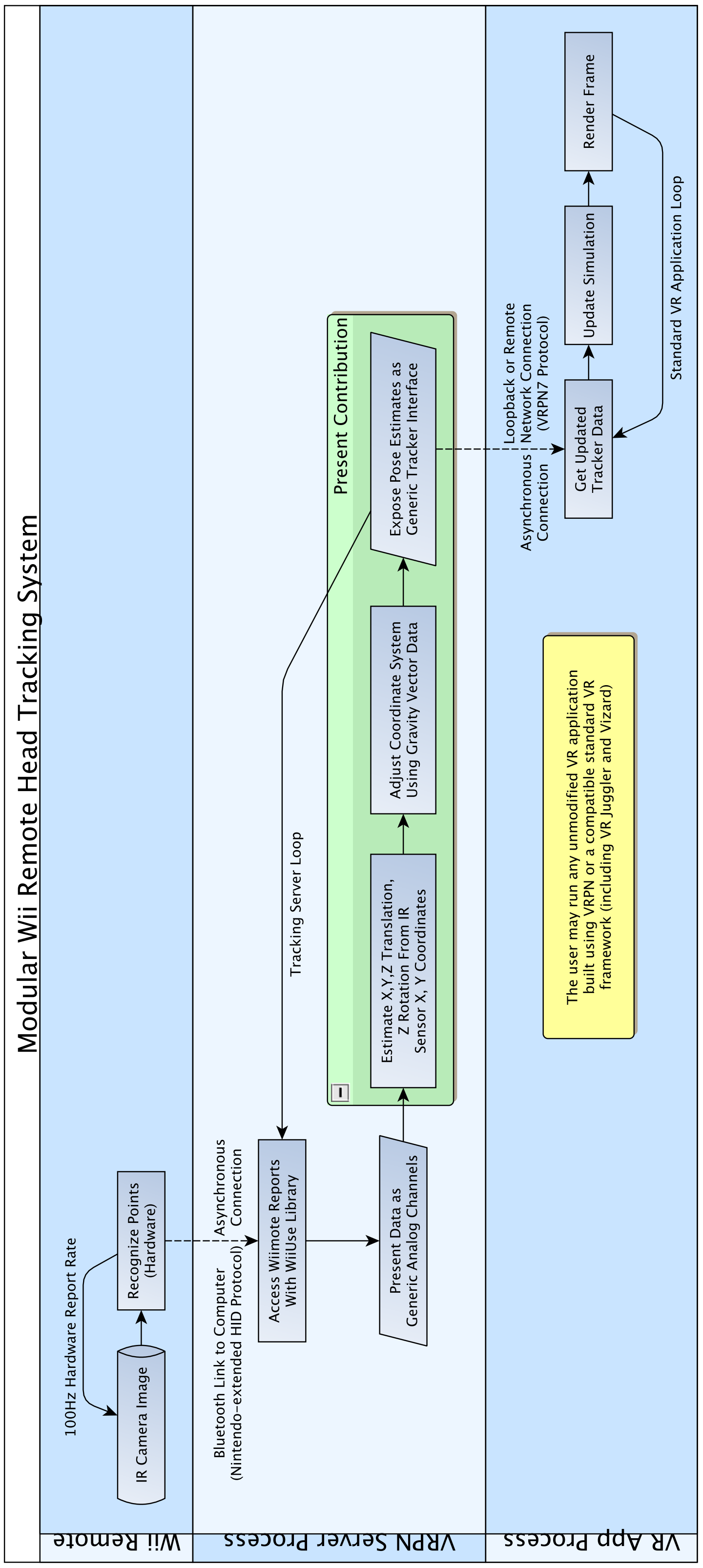

Modular Wii Remote Head Tracking System
Wii Remote
VRPN Server Process
VR App Process
100Hz Hardware Report Rate
IR Camera Image
Recognize Points (Hardware)
Bluetooth Link to Computer (Nintendo-extended HID Protocol)
Asynchronous Connection
Access Wiimote Reports With WiiUse Library
Present Data as Generic Analog Channels
Tracking Server Loop
Present Contribution
Estimate X,Y,Z Translation, Z Rotation From IR Sensor X, Y Coordinates
Adjust Coordinate System Using Gravity Vector Data
Expose Pose Estimates as Generic Tracker Interface
Asynchronous Connection
Loopback or Remote Network Connection (VRPN7 Protocol)
Get Updated Tracker Data
Update Simulation
Render Frame
Standard VR Application Loop
The user may run any unmodified VR application built using VRPN or a compatible standard VR framework (including VR Juggler and Vizard)

Figure 2.6: Flowchart showing concurrency boundaries
]

2.8. Results

The described system has been successfully implemented and used for its intended purposes. The tracking device, `vrpn_Tracker_WiimoteHead`, and the standalone server application, `wiimote_head_tracker`, have been contributed back to the VRPN community for inclusion with the package as open source software, downloadable from http://www.vrpn.org. The tracker software described is able to update its pose estimate at the full 100 Hz rate of the Wii Remote. It provides continual tracking inputs for stereo and mono view calculation by VR Juggler. The tracker's CPU usage on a wide range of desktop workstations is very low. Execution does not interfere with the applications' displays nor with a high-frequency 1000 Hz haptic rendering thread updating a force-feedback device in the test virtual assembly application. Running the head tracker server in a standalone mode results in tracking calculations taking place in a separate process from the virtual environment application, which suits the multi-processor architecture of modern personal computers. While the overall system latency has not yet been measured, anecdotal experiences using the tracker to provide head-coupled immersive display confirm the suitability of the head tracking server for the task.

The tracking system is easy to prepare and use. It generally only requires executing the tracker server application before starting the VR application. No run-time configuration or maintenance is required for the driver. Because of the gravity compensation feature, the user can configure the VR system as if the Wii Remote, as the tracker base, were completely level. The level tracker coordinate system simply needs to be located within the virtual world, as with any tracking device. The source of the tracking data should be entered as the VRPN device `Tracker0@localhost` in the most common case. On Linux, the only hardware-related setup required is for the user to press the *1* and *2* buttons on the Wii Remote to place it in visible mode immediately prior to starting the tracking application. Pairing is completed automatically. On Windows, the tested Bluetooth stacks require a more involved procedure immediately prior to each execution of the tracking server. The user must remove any existing Wii Remote pairing from the computer, press the *1* and *2* buttons on the Wii Remote, and initiate a new pairing without a passcode in a short amount of time. This slightly complicates the process of using the tracker on Windows. However, this seems to be an issue with the interaction between the Windows operating system and the modified Bluetooth Human Interface Device (HID) protocol used by Nintendo to communicate with the Wii Remote, rather than any aspect of the implementation of this tracker.

The relatively narrow field of view (FOV) of the Wii Remote sensor has been the most challenging aspect of using this system so far. As no attempts are made with the current pose estimation algorithm to update pose predictions when fewer than two IR points are observed, both the left and right extremes of the user's head must remain within the sensor's view volume or pose estimation stops until they return to view. The Wii Remote was experimentally measured to have a horizontal FOV of $\Theta_h = 43°$and a vertical FOV of $\Theta_v = 32°$. So, given a fixed distance between the LEDs d_{LED} (typically 0.15 m), and considering only translation of the user's head, the trackable area of a plane located a distance d away from the sensor and orthogonal to its view vector can be calculated, as given in eq. 2.1.

$$A(d) = \left(2d \tan \frac{\Theta_h}{2} - d_{LED}\right)\left(2d \tan \frac{\Theta_v}{2}\right) \tag{2.1}$$

The volume of the tracker's effective region between two such planes (e.g., a minimum and maximum distance) d_1 and d_2, where $d_1 < d_2$, can be calculated as the volume of the view frustum with base of $A(d_2)$ truncated at d_1, as given in eq. 2.2.

$$V = \frac{A(d_2) \cdot d_2 - A(d_1) \cdot d_1}{3} \tag{2.2}$$

The limited field of view is mitigated by the gravity auto-correction feature of the implemented driver, as the user may adjust the tilt of the Wii Remote at run-time without further configuration in order to track a greater percentage of likely head poses if loss of tracking in normal motion is noted. This works well in practice in both desktop and large projection situations.

Depending on the specific IR LEDs chosen for the glasses, the emission angle and alignment of the LEDs can somewhat impact performance of the tracker. When battery levels become low, LEDs with a narrow field of view may be less consistently detected by a Wii Remote far above or below the user's head. In practice, manual adjustment to aim the LEDs toward the Wii Remote, as well as replacement of nearly-depleted batteries minimize the impact of this limitation.

2.9. Future Work

In addition to (x, y) coordinates for each sensor point, the Wii Remote returns a size value, with 16 possible values (4 bits). This information is not presently used in the estimation. Integration of filtering, such as a single-constraint-at-a-time (SCAAT) filter [16, 4], may permit improved pose estimation through use of all available data in standard cases, as well as continued tracking updates in case the user moves outside the sensor's field of view.

The addition of a third LED to the glasses, forming a triangle, can be used to determine the difference between a rotation around a vertical axis and a translation along a sensor-normal axis. When using only the sensor location of two LEDs, both of these transformations are observed by the sensor as a decreased distance between IR points and therefore are indistinguishable motions. However, their impact on an off-axis projection rendered using the tracking data is quite distinct. Implementation of a additional tracker module that handles this three LED case is currently being pursued.

The relative utility of the sensor's view volume varies based on the position and orientation of the Wii Remote. Accordingly, analysis of head-tracking data recorded with an alternate tracking system during similar virtual reality tasks on a similar display could be used to create a tool that suggests an optimal position for the Wii Remote to capture the most probable positions and orientations accurately, based on user-input data about the physical configuration of the workstation.

Finally, multiple Wii Remotes are accessible simultaneously over the same Bluetooth protocol. The low cost of the hardware means that even a multi-view setup can be cost-effective for widespread use. An extension to the described filter driver can be implemented that takes two Wii Remote-like inputs and calculates a more accurate pose estimation for the tracked glasses.

2.10. Conclusion

A dual need for low-cost head position and orientation tracking for use in virtual reality applications led to the design of a loosely-coupled software system specialized for desktop VR head-tracking with cross-platform capabilities and transparent compatibility with existing VR software frameworks. The implementation of a tracker based on the use of IR sensors and accelerometers on a stationary Wii Remote and two IR LEDs mounted on glasses worn by the user builds on the hardware designs and basic calculations from Lee's widely popular work in Wii Remote-based head tracking. The VRPN-based filter module provides for loose coupling on both the hardware-facing and application-facing side. This design enables the system to be used to provide high-rate head tracking input inexpensively to a wide range of existing VR applications compatible with VRPN, including applications based on Vizard or VR Juggler. The standalone Wii Remote head tracker server builds on VRPN and the filter module to provide a tracking data source in a single step, and has been tested to function smoothly on both Windows and Linux platforms.

Acknowledgment

The authors are grateful to Daniela Faas of Iowa State University for assistance in the integration of these results into the virtual assembly research platform; to Meisha Rosenberg and Max Perkins of Iowa State University for support in hardware design, fabrication, and evaluation; and to the anonymous reviewers for their helpful feedback on a draft of this paper. This work was performed at the Virtual Reality Applications Center at Iowa State University, and was funded by NSF award CMMI-0928774.

Bibliography

[1] Roland Arsenault and Colin Ware. Eye-hand co-ordination with force feedback. In *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '00*, pages 408–414, New York, New York, USA, 2000. ACM Press. ISBN 1581132166. doi:10.1145/332040.332466. URL http://portal.acm.org/citation.cfm?doid=332040.332466.

[2] Kevin W. Arthur, Kellogg S. Booth, and Colin Ware. Evaluating 3D task performance for fish tank virtual worlds. *ACM Transactions on Information Systems*, 11(3):239–265, 1993. ISSN 1046-8188. doi:10.1145/159161.155359. URL http://dl.acm.org/citation.cfm?doid=159161.155359.

[3] Allen Bierbaum, Christopher Just, Patrick Hartling, Kevin Meinert, Albert Baker, and Carolina Cruz-Neira. VR Juggler: a virtual platform for virtual reality application development. In *IEEE Virtual Reality Conference (VR 2001)*, pages 89–96, Los Alamitos, CA, USA, 2001. IEEE Comput. Soc. ISBN 0-7695-0948-7. doi:10.1109/VR.2001.913774. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=913774.

[4] Gary Bishop, Gregory F. Welch, and B. Danette Allen. Tracking: Beyond 15 minutes of thought. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH '01*, 2001. URL http://www.cs.unc.edu/~tracker/ref/s2001/tracker/.

[5] Yang-Wai Chow. The Wii Remote as an input device for 3D interaction in immersive head-mounted display virtual reality. In *Proc. IADIS International Conference Gaming 2008: Design for Engaging Experience and Social Interaction*, pages 85–92, 2008. URL http://www.iadis.net/dl/final_uploads/200815L011.pdf.

[6] Yang-Wai Chow. Low-Cost Multiple Degrees-of-Freedom Optical Tracking for 3D Interaction in Head-Mounted Display Virtual Reality. *International Journal of Recent Trends in Engineering*, 1(1):52–56, 2009. ISSN 1797-9617.

[7] Daniel F. Dementhon and Larry S. Davis. Model-based object pose in 25 lines of code. *International Journal of Computer Vision*, 15(1-2):123–141, June 1995. ISSN 0920-5691. doi:10.1007/BF01450852. URL http://www.springerlink.com/index/10.1007/BF01450852.

[8] Rudolph Emil Kalman. A New Approach to Linear Filtering and Prediction Problems. *Journal of Basic Engineering*, 82(1):35, 1960. ISSN 00219223. doi:doi:10.1115/1.3662552. URL http://fluidsengineering.asmedigitalcollection.asme.org/article.aspx?articleid=1430402.

[9] Michael Laforest, Ryan A. Pavlik, and WiiUse Community. WiiUse, 2008. URL http://github.com/rpavlik/wiiuse.

[10] Johnny Chung Lee. Hacking the Nintendo Wii Remote. *IEEE Pervasive Computing*, 7(3):39–45, July 2008. ISSN 1536-1268. doi:10.1109/MPRV.2008.53. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4563908.

[11] Brian Peek. WiimoteLib - .NET Managed Library for the Nintendo Wii Remote, 2007. URL http://www.brianpeek.com/page/wiimotelib.

[12] Thomas Pintaric, Hannes Kaufmann, and Gabriel Zachmann. Affordable Infrared-Optical Pose Tracking for Virtual and Augmented Reality. In *IEEE VR Workshop on Trends and Issues in Tracking for Virtual Environments*, pages 44–51, 2007. URL http://www.ifs.tuwien.ac.at/node/13003.

[13] Torben H. Schou and Henry J. Gardner. A Wii remote, a game engine, five sensor bars and a virtual reality theatre. In *OZCHI '07: Proceedings of the 19th Australasian conference on Computer-Human Interaction*, pages 231–234, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-872-5. doi:10.1145/1324892.1324941.

[14] Torben Sko and Henry J. Gardner. The Wiimote with multiple sensor bars: creating an affordable, virtual reality controller. In *Proceedings of the 10th International Conference NZ Chapter of the ACM's Special Interest Group on Human-Computer Interaction - CHINZ '09*, pages 41–44, New York, New York, USA, 2009. ACM Press. ISBN 978-1-60558-574-1. doi:10.1145/1577782.1577790. URL http://portal.acm.org/citation.cfm?doid=1577782.1577790.

[15] Russell M. Taylor II, Thomas C. Hudson, Adam Seeger, Hans Weber, Jeffrey Juliano, and Aron T. Helser. VRPN: a device-independent, network-transparent VR peripheral system. In *Proceedings of the ACM symposium on Virtual reality software and technology - VRST '01*, page 55, New York, New York, USA, 2001. ACM Press. ISBN 1581134274. doi:10.1145/505008.505019. URL http://portal.acm.org/citation.cfm?doid=505008.505019.

[16] Gregory F. Welch and Gary Bishop. SCAAT: incremental tracking with incomplete information. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques - SIGGRAPH '97*, pages 333–344, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co. ISBN 0-89791-896-7. doi:10.1145/258734.258876. URL http://dl.acm.org/citation.cfm?doid=258734.258876.

[17] Wouter Wognum. Wii headtracking in VR Juggler through VRPN. 2008. URL http://www.xs4all.nl/~wognum/wii/.

CHAPTER 3. BEYOND THE BINDING: VIRTUAL REALITY REPL AND APPLICATION DEVELOPMENT WITH VR JUGGLUA

A paper submitted to IEEE *Transactions on Computer Graphics and Visualization*

Ryan A. Pavlik, Judy M. Vance

Abstract

VR JuggLua is a software library, virtual reality framework, and virtual environment development system originating as Lua bindings of VR Juggler and OpenSceneGraph. This combines a high-level, high-performance interpreted language for application development with the extensive configurability and virtual reality hardware support of VR Juggler, as well as the established interactive graphics and rendering power of OpenSceneGraph. It has been extended from this base by the development of an interactive code execution GUI that permits REPL-like interactivity while the frame loop of the virtual reality system continues to execute, even executing new code in parallel and maintaining consistent state across a cluster. An embedded domain-specific language has been built in Lua to provide for more declarative scene graph construction without introducing new file formats or syntax rules. Additionally, to ease development and maintenance of virtual environments, coroutines have been employed to provide the programmer with the ability to write functions that appear to run linearly and in parallel, containing a draw-like call and controlling the timing of drawing, while transparently maintaining state between frames and hiding the repeated

Ryan A. Pavlik was the primary author and researcher.
Based on an earlier paper presented as:
R.A. Pavlik and J.M. Vance, "VR JuggLua: A Framework for VR Applications Combining Lua, OpenSceneGraph, and VR Juggler," **Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS) in IEEE Virtual Reality**, Singapore: 2011. Published as doi:10.1109/searis.2012.6231166.
That previous publication is Copyright ©2012 IEEE. Reprinted with permission. With regard to the material from that publication:
Personal use of this material is permitted. Permission from IEEE must be obtained for all other users, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works for resale or redistribution to servers or lists, or reuse of any copyrighted components of this work in other works.

update callback idiom. The full system is open source and freely available online, and has been applied in a number of capacities.

3.1. Introduction

Virtual reality software frameworks span a wide range of capabilities and areas of focus. Each virtual reality (VR) framework provides some subset of the following features: operating system portability layer, input device abstraction, display view-port configuration, VR system simulation, cluster support, three-dimensional (3D) scene data structures, event system, and scripting. Frameworks that emphasize the systems level provide little or no higher-level content authoring support. Conversely, frameworks that explore the experience of content creation generally fall short in system independence and compatibility with complex or high-end virtual reality systems. This gap limits the ability of experience designers and researchers to both develop real-time interactive environments using high-level constructs and run their environments on a broad range of VR computing systems.

This research builds upon existing mature software to produce a framework supporting rapid development and iteration of virtual environments (VEs) with the potential to run on the broadest possible range of VR systems. VR Juggler was selected as the basis for this development. The VR Juggler open source virtual reality software platform ([10]) supports a broad range of VR systems, including large multi-node clusters such as the 49-node C6 facility. VR Juggler also supports Windows, Mac, and Linux. The dynamically-typed Lua programming language ([29, 30]) was selected for integration both as a scripting language to extend C++ applications and as a fully-capable development language for building complete immersive applications. Lua's clear and minimal syntax, ease-of-use for end-user programmers, and ease of interoperability with C++ supported this choice.

The resulting framework, dubbed VR JuggLua, supports the same wide range of VR systems as its VR Juggler core. It uses the OpenSceneGraph[1] (OSG) graphics library to provide scene organization, model loading, and rendering support. Using VR JuggLua, VR applications can be written entirely in Lua, or use Lua in a lesser capacity in an application written using the VR JuggLua C++ API. On top of the basic binding layer, functionality for interactive programming during the execution of a virtual reality application was added. A domain-specific embedded language built within Lua was developed to provide a more intuitive method of scene graph creation while maintaining compatibility with the full functionality of the raw binding of OpenSceneGraph. The programming language construct of coroutines was applied to allow programmers to perceive a linear control flow

[1] http://www.openscenegraph.org/

despite the frame loop of the underlying software, as well as to run effectively parallel tasks without simultaneous access concerns.

The paper proceeds as follows. Relevant literature in three distinct areas is discussed in section §3.2. The basic design of the framework and binding of VR Juggler and OSG functionality to Lua, including design challenges overcome, is detailed in section §3.3. Developments beyond the binding of software libraries to Lua that provide valuable functionality for virtual reality application development are presented in section §3.4. A few instructive examples demonstrating the value and capability of VR JuggLua follow in section §3.5.

3.2. Background

3.2.1. Virtual reality frameworks

A wide variety of software frameworks for building virtual reality applications have been developed. The CAVE Library initially developed for use with the CAVE Automated Virtual Environment ([16, 37]) is an example of early work in the systems category of virtual reality frameworks. It has evolved into a commercial solution integrating clustering support and focusing on multi-screen application development. VR Juggler introduced a highly modular architecture for VR applications to provide a “virtual platform” for development and execution on diverse systems ([10, 11]). Later development extended its use from high-end graphics systems to commodity computer clusters ([2, 11]). The FlowVR platform was developed based on experience in using VR Juggler in a clustered environment, and emphasizes a data-flow model for distributed real-time interactive computation with high modularity ([3, 4]). The Syzygy system presents multiple frameworks for VR application development, and was developed with an explicit focus on clustered execution ([43]).

Other frameworks focus more on the content authoring experience, often integrating an interpreted scripting language for rapid development. Colosseum3D integrates OpenSceneGraph, physics capabilities, and audio rendering, and combines the use of C++, a custom object-description format, and Lua scripting ([6]). Colosseum3D generates bindings of its C++ classes using the tolua++ utility. The commercial VR authoring environment Virtools[2] integrates a custom scripting language, VSL, for content creation. AVANGO/NG applies a generic field and field container programming interface to a scene graph based on OpenSceneGraph, with Python scripting support ([31]).

A programming model more closely linked to the use of an interpreted language has also found success in creating several varieties of immersive interactive experiences.

[2] http://www.virtools.com/

WorldViz Vizard[3] is a commercial application framework, using the Python language with a custom integrated development environment (IDE) to create experiences rendered using OpenSceneGraph. However, it has limited clustering support when compared to some of the systems-focused frameworks designed explicitly for distributed execution. TINT is an augmented reality (AR) and mixed reality (MR) framework designed to present a pure Python programming interface, with optional interaction with C++ modules compiled for improved performance ([19]). By delegating computationally-intensive tasks to compiled code, the bulk of applications can be written using Python for development efficiency and still achieve interactive performance. The HECTOR platform takes a similar approach integrating compiled code and interpreted Python, with an event-driven architecture for virtual reality applications ([48]).

3.2.2. Domain-specific languages

The work presented here builds a domain-specific language within Lua. Domain-specific languages, or DSLs, ([47]) became widely known and used as the many "little languages" developed in early computing and expanded by UNIX utilities ([8]). These little languages allow programmability of software in terms of the problem domain, rather than the general-use programming language they interact with, by abstracting the implementation. These particular languages feature their own parser and interpreter, often generated with tools such as Lexx and Yacc. A common reason for using a domain-specific language is to solve a problem in a given application domain with concise, clear code. The DSL might be developed and maintained by a software engineer working with knowledge of the domain, with programs in the DSL written or verified by domain experts themselves. Such code can be easier to maintain and trace to domain requirements. In many cases, the creation of a DSL parallels the process of developing a software product line family, where programs written in the DSL specify members of the family ([46, 24, 38]).

A specific kind of DSL is a domain-specific embedded language, or DSEL ([26, 27]), referred to by some as an internal domain-specific language to avoid confusion with embedded device programming ([22]). A DSEL differs from the "little languages" by building upon and within an existing programming language, rather than starting from scratch. A clear advantage of DSELs over external DSLs is that the developer is spared the overhead of developing a syntax, parser, and lexer entirely from scratch, reducing the startup cost of the DSL approach. Additionally, tooling for the existing language can be re-used. This lowers the startup cost of using a domain-specific language as compared with developing applications for a problem domain in a general purpose language ([27]).

[3] http://www.worldviz.com/products/vizard/

Some languages are particularly well-suited to hosting DSELs. Common LISP idioms have been considered DSELs, and the functional language Haskell is also considered well-suited to DSEL development ([26]). Notably, Lua was developed to replace an earlier data description language (a limited type of DSL) called SOL, and features minimal base functionality with syntactic sugar targeted toward development of DSELs that resemble the bibliographic data file format BibTeX ([30]).

3.2.3. Read-Eval-Print loop (REPL)

Many commonly-used modern interpreted languages, including Python, Ruby, and Lua, provide a REPL-style interactive shell as a development tool. The concept of a "read-eval-print loop" provides an interactive programming environment by reading user input, evaluating the input as an expression, printing results, then looping to allow further execution in the same context. The history of REPLs traces back to the origins of the programming system LISP (LISt Processor) ([33]). In LISP, all data, including programs themselves, were represented as nested structures based on lists. Printing such structures was an early first step in the development of LISP, and for the sake of data persistence beyond a single session reading list structures soon followed. As even programs were represented in this way, development of the `eval` function produced the first interpreter, which could be called in an infinite loop ([34]). An interactive interpreter could then be implemented with just the functions `read`, `eval`, `print`, and `loop`, which led to the term commonly used today. It is difficult to trace when exactly these function names became combined into a name for all such similar systems, but Sandewall does use the phrase in a publication from 1978 when describing the "incrementality" requirement of an interactive programming system ([42]).

The concept of incrementality and interactive computing (what would now be referred to as interactive programming) dates at least back to 1964. Lombardi and Raphael conceive of an "interactive computer," whose focus is on evaluating expressions, as distinct from executing stored code ([32]). The implementation of these ideas as REPL environments for both programmer support and pedagogical purposes were well-established by the DrScheme environment ([21]). Scheme, a LISP dialect, was used as a teaching programming language in a number of computer science curricula, often based on or inspired by the well-known "Structure and Interpretation of Computer Programs" (SICP) text developed at MIT ([1]). By providing enhanced features on top of a bare REPL, DrScheme provided a useful environment to experiment and incrementally develop programs in Scheme. Based on the merits of a REPL environment for teaching programming, DrJava was developed providing a similar experience built on a Java interpreter instead of Scheme ([5]). While not

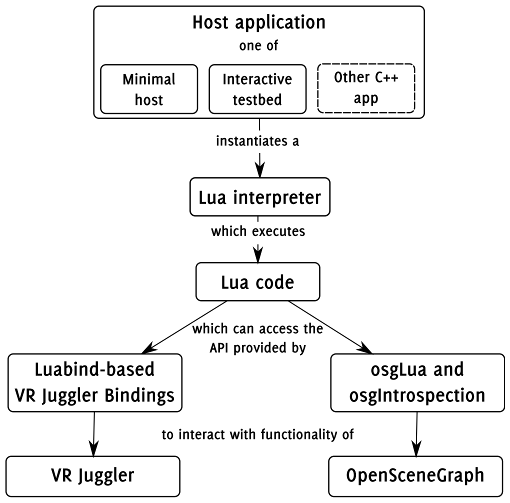


Figure 3.1: System diagram

all interpreter interactive shells rise to the level of these two teaching-focused programming environments, they do provide a useful tool for both beginning and experienced developers. They may not meet the theoretical requirements of a LISP REPL as strictly constructed, but common usage refers to them as such, and as the example of DrJava demonstrates, they provide a number of the benefits of interactive computing.

3.3. System Design

This section describes the implementation of the VR JuggLua framework starting from the foundation of existing software and extending and continuing toward higher levels of the platform. Section 3.3.1 discusses the base levels of existing software used in this framework. Section 3.3.2 addresses integrating these systems and presents a coherent, logical interface for application development.

As a full framework, VR JuggLua encompasses its foundational software, bindings for this software to Lua, the Lua interpreter library itself, and basic host applications. A typical application will have only one Lua interpreter state with access to all bindings included in VR JuggLua. A VR JuggLua application uses both the osgLua module and the VR Juggler bindings included in the VR JuggLua framework to access a complete set of virtual reality functionality from Lua (Fig. 3.1).

3.3.1. Foundational software

The VR Juggler software framework is a "virtual platform" for development of VR software that can be used on a wide variety of VR computing systems ([10]). It consists of several components that together allow virtual reality applications to be written in C++ and executed using various hardware configurations. The VR Juggler library provides display management and transfers control during specific periods of the frame loop to application objects. Application objects are the highest-level of content authoring interface presented by VR Juggler. Specializations of the base application object are included that support using scene graph libraries, including OpenSceneGraph and OpenSG ([41]). The VR Juggler kernel, however, is intentionally independent of any particular scene system, and can even support DirectX graphic rendering in addition to OpenGL and OpenGL-based scene-graphs.

Binding to Lua

The Lua language is a high-performance language designed for embedding and extension ([29, 30]). The Lua language must always be tied to a host application. A minimal host application that presents a basic Read-Eval-Print loop (REPL) as well as script execution is included with the standard Lua implementation. The canonical Lua source code is included in the VR JuggLua source tree and built as a library during the software build.

On top of Lua, the Luabind[4] library provides an intuitive method of wrapping C++ classes, methods, and functions for access from Lua. It uses template metaprogramming techniques to generate appropriate Lua C API calls for binding at compile-time, which allows it to automatically deduce function signatures in most cases and compile directly to a binding in a single step. VR JuggLua applies Luabind to create bindings to VR Juggler components. These bindings function like any other Lua module, extending the functionality of any interpreter state in which they are loaded.

OpenSceneGraph and osgLua

OpenSceneGraph (OSG) was selected as the graphics subsystem of VR JuggLua. It is a mature scene graph, supported in VR Juggler, with good interoperability across platforms and import plug-ins for a wide variety of image and model file formats. Importantly, reasonably up-to-date bindings for OpenSceneGraph to Lua were available, in a package called osgLua.[5] Rather than manually creating bindings for all of OpenSceneGraph, or preprocessing the OSG headers, osgLua uses the osgIntrospection library to provide access to nearly all OSG classes. As a part of OSG 2.8.x, osgIntrospection loads wrapper dynamic libraries generated automatically from the source. By dealing only with osgIntrospection

[4] http://www.rasterbar.com/products/luabind.html

[5] http://svn.pplux.com/lab/osgLua/

types, values, and methods, rather than statically binding to specific types and methods, osgLua is able to avoid falling behind upstream OSG development and offer nearly complete coverage of the library's capabilities. Though public development of osgLua seems to have stalled in late 2007, this introspection-based approach allows it to work on newer versions with only minor updates.

Developing the VR JuggLua software framework also resulted in developing a large number of improvements to osgLua and osgIntrospection, to both fix errors and extend functionality. Among the improvements include direct access to object properties without using set/get functions, providing a more natural and Lua-like interface. The introspection-based binding was also augmented with generated code to specifically recognize the vector, matrix, and quaternion data-types, selectively defining additional Lua metatable methods for these values to allow the direct use of the normal math and comparison operators in Lua code.

Connecting osgLua and Luabind

The distinct representations of Luabind-wrapped objects and osgLua-managed objects in Lua state presented a challenge during VR JuggLua implementation. A key insight is that once osgLua is loaded, OSG types can effectively be considered "native types" in Lua. Luabind provides a template-based system allowing seamless conversion between C++ string types and Lua strings, C integer and floating point types and Lua numbers, and so on. Luabind has a public `native_converter_base` interface to allow developers to provide similar converters for their own specialized classes wrapping these basic data-types. This converter interface was applied to allow osgLua-managed objects to be passed to and returned from methods bound with Luabind.

OpenSceneGraph types can be divided into two groups: reference types, which are always allocated on the heap and passed by pointer, and value types, which may be allocated on the stack. Templated subclasses of the Luabind native converter template base class were made to handle these two categories of data types. The Boost Type Traits library was used to selectively enable a specialized default converter when a given type inherits from `osg::Object`. This approach results in the need for only a few-line class to specify the type name for each OSG value type is involved in the Luabind-wrapped method. C preprocessor macros were employed to reduce this to a single line per OpenSceneGraph value type. When VR JuggLua is compiled, it invokes the macros for the common OSG types that it uses. If a client application written in C++ wishes to bind functions to Lua and requires support for additional OSG types in the binding, the header can be included and the preprocessor macros can be invoked for any available type. This solution allows OSG types to be passed seamlessly between Lua and Luabind-bound C++ code.

3.3.2. Low-level API

The approach taken to binding the VR Juggler components to Lua was to keep the interface simple and allow the most common use cases to be written entirely in Lua. From the application's point of view, interaction with the VR Juggler kernel is limited to specifying the jconf configuration files, and starting and stopping the application thread. In the C++ API, all kernel interactions take place with the singleton ([23]) instance of the kernel. In the Lua binding, then, the singleton kernel instance is implied, and small free functions were bound that look up the singleton pointer and call the method.

Access to device input takes place through a variety of device interface classes in Gadgeteer, the input device management library of VR Juggler. These classes were bound one-to-one, but with slight modifications. The need to separately call an `init` function with the name of the device alias, mandated in C++ by the smart-pointer pattern implemented by these device interfaces, was eliminated in favor of a parameter to the constructor. Getter methods are used in the C++ interface, while in Lua, the input device data can be easily presented as directly-accessible properties. VR Juggler uses the GMTL matrix and vector math template library[6] that, while suiting the purposes of VR Juggler applications without a scene graph system, does not directly inter-operate with the equivalent types in OpenSceneGraph. The Lua binding offers the opportunity to standardize on the OpenSceneGraph types, so positions and transforms are accessible as OSG vector and matrix types, using meters as the units.

To provide a fully-featured VR software framework, VR JuggLua also includes Lua bindings to the main data types of Sonix, the VR Juggler sound library. As with the Gadgeteer device interfaces, the Lua binding exposes read-only or read-write properties instead of getter/setter methods where feasible. Sounds can be configured either externally in a jconf configuration file, or at run-time in Lua code, and triggered by Lua code when applicable. The ability to keep sound triggering code in Lua improves the clarity of C++ simulation code by separation of concerns ([28]).

To complete the binding of VR Juggler to Lua, a method for creating application objects, the basic unit of the VR application, was needed. Application objects implement a C++ interface specifying action to take during initialization and each of the steps in the kernel frame loop: `preFrame`, `latePreFrame`, `draw`, `intraFrame`, and `postFrame`. In applications based on VR Juggler and OpenSceneGraph, the `osgApp` specialization of the application

[6] http://ggt.sourceforge.net/

object interface contains an implementation of the draw method to render the scene graph. Most application logic is called during the `preFrame` or `latePreFrame` stages, which can update the scene graph based on newly-received input device data.

To allow an application to be written entirely in Lua, an implementation of the `osgApp` interface was needed. To allow kernel calls to application object methods to invoke Lua functions, an application object proxy class was created, using a synthesis of the proxy and delegation design patterns ([23]). The proxy class derives from the VR Juggler `osgApp` class. Lua code can instantiate this application proxy and pass a Lua table data-structure to it, which serves as an application object delegate. If this table has function elements whose names match the methods in the application object interface, the application proxy will call those functions during the appropriate phase of the kernel frame loop. Defining an application object this way is an application of latent or "duck typing" [7] as popularized by the Python programming language ([20]). If a Lua table has one or more method that an application object would have, it can be considered an application object, without requiring the introduction of a specialized type.

Luabind does permit binding of classes with virtual methods and the sub-classing of those classes entirely in Lua, so a strict typing approach to creating Lua application objects is possible. However, the application proxy object approach taken in VR JuggLua has several advantages over direct sub-classing in Lua. For instance, the application proxy object can perform some error checking. If an application delegate has not been set by the time the kernel requests application object and scene initialization, a useful error message can be produced and execution can be stopped. Similarly, if a delegate has been set, but no forwarded calls have succeeded in an entire frame loop, the application proxy can assume that a logic error has occurred and stop execution. The application proxy layer also allows simplifying standards to be implemented. For example, despite display configuration taking place in meters, the default projection with VR Juggler produces a foot unit-based scaling. As VR JuggLua standardizes on meters for positional device data, the application proxy creates a root scaling transform node to produce an apparently meter-based display setup for VR JuggLua applications.

3.4. Extensions Beyond the Binding

To this point, a fairly direct binding of VR Juggler to Lua has been described, with some novel work to connect distinct Lua binding systems and a relaxing of strict type requirements for application objects as posed by C++. This work forms the foundation for

[7] Originating in a quotation attributed to nineteenth-century poet James Whitcomb Riley: "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

the development of a number of advanced capabilities above and beyond simple creation of VR Juggler applications in Lua code resembling the equivalent C++ code. Taken together, the following extensions beyond the binding provide useful, novel capabilities that both lower the barrier of entry for creating virtual experiences and improve the developer experience across the spectrum of expertise.

3.4.1. A “Run Buffer” for REPL in event loops

Despite the interpreted nature of Lua and its stock host REPL environment, the basic binding described loses this development interactivity and essentially imposes an “edit-compile-test” cycle with an application restart substituted for the compile step. This is due to the “don’t call us, we’ll call you” design of VR Juggler ([25]), which is common to many GUI event loop architectures. Effectively, once an application completes a few startup steps, it turns over control to an external event loop which calls it back to handle events. Once the control transfer statement starting the event loop is executed, it may not return until the application exits. This removes the interactivity produced by a REPL such as the basic Lua interactive interpreter. In the specific case of VR Juggler, the function to start the kernel event loop (known in this case as the frame loop) returns after spawning a thread. However, in practical usage, once the kernel is started by Lua, the Lua thread must block until kernel exit, since any attempt to interpret additional new code from outside the kernel thread would result in concurrent threading problems as the kernel thread and the initial thread both attempt to interact with a single Lua interpreter state simultaneously.

In extending the REPL to this type of architecture, consider that in event loop/frame loop programming, idle operation consists of a continuously executing event loop, rather than a blocking input call as found in a command-line REPL-type application. As such, a separate user interface (UI) thread is the simplest way to implement the “read” portion of a REPL independent from the idle loop. The UI can effectively block waiting for the user’s input of code to evaluate. The evaluation of new code input, however, must still take place within the event loop. The “print” portion of such a REPL consists both of optional text print output (logically directed back to the UI) as well as a change in the continually-running environment’s state. In VR Juggler, the `latePreFrame` step of the kernel loop is the most logical place to evaluate new code, since the application state at that point corresponds neatly to a mental model of interactive execution: accessing device interfaces will return the most recent data, and changes to the graphical state are possible and will be immediately reflected in the display in the subsequent `draw` step. A delay of at most a frame-length between code entry and evaluation does occur due to the asynchronous nature of the GUI console and the kernel frame loop, though this is imperceptible.

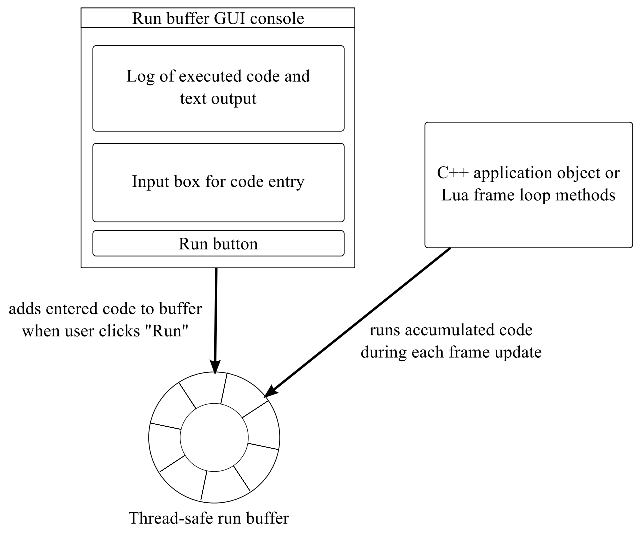
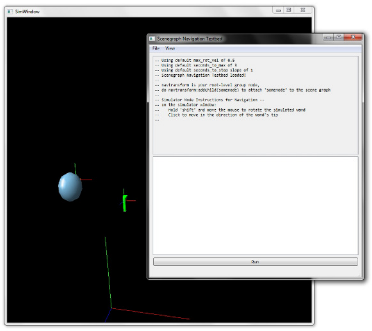

Run buffer GUI console
Log of executed code and text output
Input box for code entry
Run button
C++ application object or Lua frame loop methods
adds entered code to buffer when user clicks "Run"
runs accumulated code during each frame update
Thread-safe run buffer

Figure 3.2: Thread-safe run buffer for interactive execution

Based on these concepts, VR JuggLua includes a thread-safe run buffer system supporting interactive code execution during application run-time, illustrated in Fig. 3.2. Code can be added to this circular buffer at any time from C++ (including an interactive console) or Lua. A single method call on the buffer runs all contents, in order. This run buffer method call is bound to Lua and can be placed in the application delegate function corresponding to the `latePreFrame` state. An interface for an interactive GUI console, with text-based stub, FLTK[8], and Qt[9] implementations, exposes this code-entry functionality to a user. Any VR JuggLua-based application can use this GUI console as a drop-in component, supporting not only REPL-style code entry and display of print output from Lua, but also additional functionality built into the GUI such as script file loading/saving and drag and drop handling.

Applying the run buffer and GUI console, an interactive virtual reality REPL capability was created. An application initially designed to serve as a testbed for scene creation and manipulation, referred to as "NavTestbed," is the immersive parallel of the minimal Lua host REPL. All details of setting up a VR Juggler application are handled behind the scenes. A minimal Lua application object provides navigation capability and runs the code accumulated in the run buffer. An empty scene and console are presented on start-up, and user code is executed interactively and apparently immediately. This interactive console allows learning of syntax to proceed more rapidly than the edit-compile-run cycle of C++ or even the edit-run cycle of a bare Lua VR JuggLua application. Lua errors are presented immediately in the GUI console, and by default do not halt the execution of

8 `http://www.fltk.org/`
9 `http://qt-project.org/`



```
-- Using default max_rot_vel of 0.5
-- Using default seconds_to_max of 3
-- Using default seconds_to_stop slope of 1
-- Scenegraph Navigation Testbed loaded!
--
-- navtransform is your root-level group node,
-- do navtransform:addChild(somenode) to attach 'somenode' to the scene graph
--
-- Simulator Mode Instructions for Navigation --
-- in the simulator window:
--    Hold 'shift' and move the mouse to rotate the simulated wand
--    Click to move in the direction of the wand's tip
--
```

Figure 3.3: Interactive testbed application with console and simulator window

the application. The user is thus encouraged to try the code again, with modifications as errors would point out. In anecdotal experience, the console serves well to localize errors in longer, more complex virtual environments: if the full script does not produce the desired results, users quickly learn to try pasting code incrementally. In effect, the debugging behavior of stepping through problem code arises spontaneously as a user works with the environment. The testbed application does not impose any specific structure on application code developed interactively. Executed code, interspersed with text output formatted as Lua comments, is logged and available for saving to a script file or copying and pasting into a text editor. While this application was initially intended to be a minimal testing host, when combined with the control structures discussed in 3.4.3, it has become the host for nearly all VR JuggLua-based development and operation.

Figure 3.3 shows this "NavTestbed" application running on Windows 7, on a desktop system in simulator mode. The window in the background is the simulator view, showing a representation of the room coordinate system as well as the simulated head and wand position. VR Juggler simulator mode allows keyboard and mouse inputs to be translated into immersive device inputs, such as head and wand position tracking and wand button presses. Though simulator mode loads by default, configuration files for an immersive VR system can also be loaded, allowing experimentation with virtual environment design to take place in the actual hardware system used for running completed applications. The GUI console can either float above the windows rendering the immersive display or be moved to an additional non-immersive display.

The solution described to this point provides an effective REPL for the VR Juggler event loop system in VR JuggLua when run on a single node. However, many target environments for VR Juggler include a cluster of render nodes in addition to an interactive head node. VR Juggler provides a system for synchronize user data across a cluster along with the input device data every frame. By registering the run buffer with this user data synchronization mechanism, code entered interactively on the head node is synchronized and run simultaneously on all nodes of the cluster. By only running the code that had accumulated in the buffer by the time of the pre-frame data synchronization during that frame, consistent execution state down to a per-frame resolution can be guaranteed. Essentially, the code entered by the user/developer between the start of consecutive pre-frame states is held and run on all nodes only once it has been synchronized. When the GUI console is launched in a cluster environment, render nodes can either replace the GUI console with a stub text console, or allow the GUI console to automatically detect its cluster state and disable code input on that node.

3.4.2. Embedded domain-specific language for scene graph description

Lua allows rapid development of experience authoring techniques, primarily due to its concise "constructor" syntax and table data structures that allow for development of data description and domain-specific languages. The osgLua library provides a fairly direct translation of the C++ API of OpenSceneGraph to Lua. While this approach allows access to the full potential of the library, it can make common tasks repetitive and unclear. For instance, using pure osgLua syntax, the following code would be used to load a model, attach it to a transform, and attach this transform to a root scene graph node.

```
t = osgLua.loadObjectFile("teapot.osg")
xform = osg.PositionAttitudeTransform()
xform:setPosition(osg.Vec3(1, 0, 0))
xform:addChild(t)
root:addChild(xform)
```

Lua allows tables, which are a data structure like associative arrays, to be created in-line with `{}`, and function calls passing a single table argument may be made simpler by abbreviating `functionCall({data, data})` as `functionCall{data, data}`, which is known as the constructor syntax. Clearly-named functions designed for constructor syntax can replace the scene graph creation code listed above with this simpler, yet equivalent alternative:

```
root:addChild(

    Transform{

        position = {1, 0, 0},
        Model("teapot.osg")

    }

)
```

Here, `Transform` is a function taking some named arguments specifying property values for a `PositionAttitudeTransform`, as well as any number of unnamed arguments corresponding to OSG nodes to add as children. It is being called with a `position` argument, as well as the results of a call to `Model`, a simple wrapper around `osgLua.loadObjectFile` to load a given file and report an error if the load is not successful. The `PositionAttitudeTransform` node created and returned by `Transform` is passed directly to the original C++-style `addChild` call to connect it to the scene graph root.

This DSEL more clearly indicates the values assigned to node properties, and also directly conveys the nesting of the model node within the transform node, an important aspect of the scene's organization that might be missed in the more procedural original code. In a sense, the code resembles the data structure it represents. In this way, an embedded domain-specific language was developed for the most commonly-used scene graph components, including `Transform`, `MatrixTransform`, `Group`, `Switch`, and `Geode` (a geometry node). By building this language within Lua instead of externally such as in Colosseum3D ([6]), it could be developed incrementally, and it avoids introducing an additional set of syntax rules and a new file format. Since each DSL function returns the standard osgLua data type, interoperability is maintained between the DSL and C++-style osgLua code. By incrementally creating wrapping functions for the existing API, the DSEL in VR JuggLua can be characterized as applying the API-to-DSL notation decision pattern ([35]).

3.4.3. Coroutines for parallel linear control flow

Coroutines are a control abstraction that allow control flow to re-enter at the point where it exited, essentially suspending and resuming an independent thread ([15, 18, 17]). Note that here, thread does not necessarily imply concurrent execution, or the synchronization and locking concerns that come with it. Rather, Lua coroutines provide multiple threads of control, with full lexical scoping and independent call stacks, in a single operating system thread. The application of coroutines to virtual environment creation was reinforced by anecdotal experience working with a novice developer. When considering how one might make an element of a scene continuously rotate, the first reaction of the new developer was to volunteer the idea of an infinite loop. More experienced developers familiar with event loops will quickly recognize that this is not feasible in the traditional way, as the update callback must return for the system to begin drawing. One implementation might transform this concept by preserving state between update callbacks in a member variable or similar structure that persists between calls. However, coroutines transparently preserve execution state between suspend and resume. This provides the basics to effectively inverting the control structure, allowing a user to write code that appears to control drawing and execution.

This is implemented in VR JuggLua as follows. A collection of coroutines resumed each frame, called "frame actions," is managed by the application delegate, which resumes each one in turn during every `latePreFrame` state. A developer can create a "frame action" procedure that appears to run with linear control flow, in parallel with other such procedures, by passing a function to a function called `Actions.createFrameAction()`, which creates a coroutine for that function and adds it to the collection of frame actions. Frame actions may freely call other functions, interact with state, and create additional frame actions. VR JuggLua provides number of wrappers around the `coroutine.yield()` Lua function for use within frame actions, the simplest being called `Actions.waitForRedraw()`. When each still-active frame action is resumed by the application delegate, the duration between the previous frame and the current frame is passed. The result is that `waitForRedraw()` and its peers appear to pause the calling procedure to draw, then return the amount of time that they paused. It is important that this duration be returned, so that frame-rate-independent animation or simulation can be performed in the frame action.

With this system in place, the combination of the run buffer and the frame action system in the application delegate provides a base for application development without the need for developing additional application delegates. Pure Lua VR JuggLua applications are typically constructed with some immediately-executed code (often setting up the scene graph) and typically one or more frame actions, handling interactions, navigation, and other behaviors. The run buffer and REPL allows additional code to be added during runtime, whether it executes immediately, creates a new frame action, or both. No expressive power is lost by avoiding the use of custom update functions in an application delegate: the idiom of repeated calls to an update function can be emulated by a frame action containing a loop with just an update call and a wait for redraw. The configuration of the frame action collection is similar to the description of the audio synthesis scheduler in Lua presented in [45], also implemented in Lua using coroutines.

3.5. Examples

Immersive applications have been successfully written using the VR JuggLua system, both in pure Lua and in a combination of C++ and Lua. Students in a graduate-level introduction to virtual reality course have used it across several semesters to produce final projects. VR JuggLua has been the platform for research in virtual reality and related fields leading to a number of publications ([39, 12, 36, 9, 13, 40, 14, 7]). This section will highlight a few samples of the results achieved using VR JuggLua notable for illustrating particular aspects of the framework's design.

3.5.1. Learning Virtual Reality Interactively

The interactive testbed application was applied in an unstructured undergraduate learning environment which focused on concepts of scene-graphs and 3D virtual reality. A sample task of scene design was assigned, with the goal of prototyping a more sophisticated application. A reasonably-complex scene was built from multiple models, sourced internally as well as from the Google 3D Warehouse[10]. An iterative process on typical laptop and desktop computers was observed, with rapid iterations of the application script tested interactively using the testbed application. The script constructed in this way was then launched in a single-machine two-walled immersive environment for more thorough testing. It was ultimately demonstrated in the C6, a six-wall high-resolution CAVE-like system powered by a 49-node cluster. The application performed smoothly and as designed.

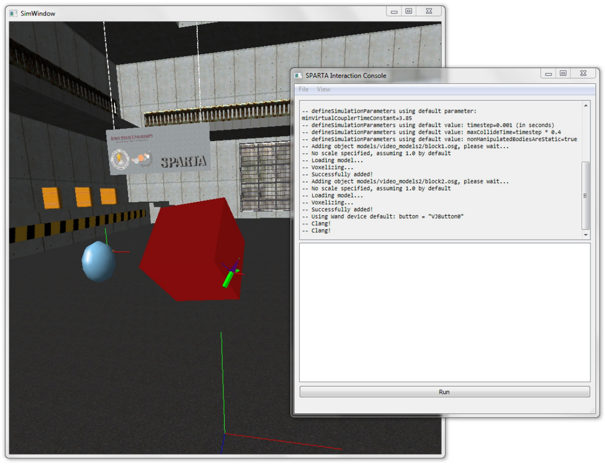
3.5.2. Testing Navigation Techniques

In the course of a summer program for undergraduates, a scenario was developed for testing navigation in a user study in the C6 environment. An application was written, entirely in Lua, by undergraduate and graduate students. The application loaded sophisticated models, and supported comparison of two navigation techniques based on device input. The necessary transforms and manipulations to display the externally-sourced models were developed on desktop machines using the interactive testbed. The navigation techniques interpreted analog and positional data from sensors on an instrumented real object, to provide an on-screen registered virtual version of the object and to allow movement in a natural way. Logging of navigation data was implemented, and a successful user study was completed, in a limited time frame, eventually leading to publication ([12]). High performance of the application was observed, despite the use of an interpreted language and very complex graphical model. This is made possible due to the delegation of graphics rendering to the C++-based OpenSceneGraph. Lua code traversed and modified models at load time and updated transforms during run-time, but the actual rendering code in a VR JuggLua application remains part of OpenSceneGraph.

3.5.3. Integrating with C++ Simulations

Research into applications of virtual reality technology to manufacturing engineering led to the development of virtual assembly simulations with haptic force-feedback capability. The *Scriptable Platform for Advanced Research and Teaching in Assembly* (SPARTA) is the

[10] http://sketchup.google.com/3dwarehouse/



```
-- defineSimulationParameters using default parameter:
minVirtualCouplerTimeConstant=3.85
-- defineSimulationParameters using default value: timestep=0.001 (in seconds)
-- defineSimulationParameters using default value: maxCollideTime=timestep * 0.4
-- defineSimulationParameters using default value: nonManipulatedBodiesAreStatic=true
-- Adding object models/video_models2/block1.osg, please wait...
-- No scale specified, assuming 1.0 by default
-- Loading model...
-- Voxelizing...
-- Successfully added!
-- Adding object models/video_models2/block2.osg, please wait...
-- No scale specified, assuming 1.0 by default
-- Loading model...
-- Voxelizing...
-- Successfully added!
-- Using Wand device default: button = "VJButton0"
-- Clang!
-- Clang!
```

Figure 3.4: SPARTA in simulator mode showing Lua console

successor to the *System for Haptic Assembly and Realistic Prototyping* (SHARP) as developed by Seth et al. ([44]). SPARTA is an application built on VR JuggLua in which C++ code performs physically-based simulation of interactions between part models, rendering corresponding haptic force feedback to haptic devices such as the Geomagic® Touch™ and the Virtuose™ 6D35-45 by Haption at a rate of 1000 Hz.

Classes in SPARTA representing the physics simulation, physical bodies, and manipulator devices are bound for Lua access by Luabind. Lua code executed using the GUI console and run buffer is used to configure interaction devices and techniques, load parts to interact with, and start the physics simulation. Lua scripts performing these tasks are used in place of configuration files, offering extended functionality for complex configurations and eliminating the task of writing a configuration file parser. Scripts are either loaded from the command line, or interactively using the GUI console, which has been included as a "drop-in" component and allows incremental development of SPARTA configurations akin to the incremental development of VR JuggLua applications enabled by the interactive testbed application. Figure 3.4 shows SPARTA running in simulator mode with the GUI console visible along with the simulator window. Of course, like all VR JuggLua applications, SPARTA can be run in a fully-immersive mode without any changes beyond configuration files.

Furthermore, while C++ code performs the physics computations and high-rate simulation in SPARTA, the visual and audio feedback is written entirely in Lua. A simple application object delegate handles updating the positions and orientations of models in the scene graph based on the current simulation state. Collision statistics are monitored by separate Lua code to trigger appropriate sounds using the Sonix binding in VR JuggLua. Frame actions are used for additional functionality, as well as automation of user study procedures. In effectively providing a superset of VR JuggLua's capabilities, most Lua-only functionality developed in VR JuggLua can be used directly in SPARTA.

The case of SPARTA illustrates a high-end application of VR JuggLua: it is a sophisticated application taking advantage of the VR JuggLua C++ API and binding its own internal objects to Lua. It uses Lua code to configure the C++ core and translate simulation state into visual and audio displays. The Lua interface provides an extension point for investigating interaction techniques.

3.6. Conclusions and Future Work

VR JuggLua began as a binding of VR Juggler and OpenSceneGraph to Lua. Subsequent research and development have extended its potential in three major ways. Interactive code execution in the manner of a REPL was re-introduced using a cluster-capable, thread-safe run buffer. An embedded domain-specific language for scene graph description in Lua was built. Finally, coroutines were applied to invert apparent control flow to permit more linear execution and formulation of virtual environment behavior. The software is developed under an open-source license and both source and Windows binaries are provided online.[11]

The interactive GUI prompt adds code to a run buffer asynchronously from the frame loop. The run buffer concept allows interactive code execution simulating the behavior of a blocking REPL while the frame loop continues. Just syncing the buffer when device data is synchronized is not sufficient to extend this to a cluster because it's possible to enter code after synchronization but before the contents of the buffer are run. This problem was solved by only executing those run buffer entries that have been synchronized, and delaying code entered between sync and run until the next frame. This allows interactive code execution, in a running frame loop, across a cluster maintaining consistent state on all nodes.

Building bindings for VR Juggler and providing a base application shell eliminates the boilerplate required for a simple application. Between new bindings written for VR Juggler and the enhanced introspection-based OpenSceneGraph bindings, a large proportion of the underlying functionality is available. However, this functionality was primarily available through imperative-style code strongly resembling its C++ equivalent. By building an embedded DSL for scene graph description in Lua, common scene graph construction tasks can now be written in a way that reflects the structure in a more declarative style. By employing the API-to-DSL notation decision pattern, the DSL is seamlessly compatible with the fullness of functionality accessible through the imperative C++-style API. The DSL

[11] `https://github.com/vancegroup/vr-jugglua`

simplifies common code in a way that is both useful to novice VR programmers learning about scene graphs and helpful to experienced programmers working on development and maintenance of a virtual environment.

Typical control flow in VR software involves relinquishing control to an event loop that will periodically call a particular function for application-specific logic and updates. In the context of developing a virtual environment, it can be more natural to think of program logic proceeding somewhat linearly, rather than as repeated calls to an update function. Coroutines provide a model and method of multiple parallel, non-concurrent threads of control that can pause and resume execution. By implementing a collection of coroutines resumed each frame during the update function, and wrapping a "yield" (return and suspend) call as a "wait for redraw," a system of "frame actions" was developed for providing more linearly-structured execution with state transparently preserved across multiple frames. The frame action coroutine system can trivially emulate the per-frame "update callback" idiom, so it loses no expressive power.

Future work includes collaborative work to broaden usage of VR JuggLua outside of Iowa State University courses and research. Some technical changes could be made to more easily permit use of third-party Lua modules for orthogonal functionality. While the package runs on Windows, Linux, and Mac OS X, some additional work is needed to provide an easily-distributable application bundle on Mac. Formal evaluation of the system's usability with novice programmers is also being considered.

Acknowledgment

Thanks to Leif Berg, Patrick Carlson, Meisha Rosenberg, Adam Carlson, Alicia Fleege, Andrew Moore, and Jane Peters for using VR JuggLua in diverse real-world applications and providing valuable feedback. This paper builds upon work initially presented at the Fourth Workshop on Software Engineering and Architectures for Realtime Interactive Systems at IEEE VR 2011. This work was performed at the Virtual Reality Applications Center at Iowa State University. This material is based upon work supported by the National Science Foundation under grant #CMMI-0928774.

Bibliography

[1] Harold Abelson, Gerald Jay Sussman, and Jul Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2 edition, January 1996. ISBN 978-0-262-01153-2. doi:10.2307/3679579. URL `http://mitpress.mit.edu/sicp/`.

[2] Jérémie Allard, Valérie Gouranton, Loick Lecointre, Emmanuel Melin, and Bruno Raffin. Net Juggler: Running VR Juggler with Multiple Displays on a Commodity Component Cluster. In *IEEE Virtual Reality Conference (VR 2002)*, volume 2002, pages 273–274, Los Alamitos, CA, USA, 2002. IEEE Comput. Soc. ISBN 0-7695-1492-8. doi:10.1109/VR.2002.996534. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=996534.

[3] Jérémie Allard, Clément Ménier, Edmond Boyer, and Bruno Raffin. Running Large VR Applications on a PC Cluster: the FlowVR Experience. In Eric Kjems and Roland Blach, editors, *Proceedings of EGVE/IPT 05*, pages 59–68, Aalborg, Denmark, 2005. Eurographics Association. ISBN 3-905673-21-5. doi:10.2312/EGVE/IPT_EGVE2005/059-068.

[4] Jérémie Allard, Jean-Denis Lesage, and Bruno Raffin. Modularity for Large Virtual Reality Applications. *Presence: Teleoperators & Virtual Environments*, 19(2): 142–161, April 2010. ISSN 1054-7460. doi:10.1162/pres.19.2.142. URL http://www.mitpressjournals.org/doi/abs/10.1162/pres.19.2.142.

[5] Eric Allen, Robert Cartwright, and Brian Stoler. DrJava: a Lightweight Pedagogic Environment for Java. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education - SIGCSE '02*, pages 137—-141, New York, New York, USA, March 2002. ACM Press. ISBN 5581134738. doi:10.1145/563340.563395. URL http://portal.acm.org/citation.cfm?doid=563340.563395.

[6] Anders Backman. Colosseum3D – Authoring Framework for Virtual Environments. In Eric Kjems and Roland Blach, editors, *Proceedings of the 9th Int. Workshop on Immersive Projection Technology, 11th Eurographics Workshop on Virtual Environments, IPT/EGVE 2005*, pages 225–226. Eurographics Association, 2005. ISBN 3-905673-21-5. doi:10.2312/EGVE/IPT_EGVE2005/225-226.

[7] Sara Behdad, Leif P. Berg, Deborah Thurston, and Judy M. Vance. Leveraging Virtual Reality Experiences With Mixed-Integer Nonlinear Programming Visualization of Disassembly Sequence Planning Under Uncertainty. *Journal of Mechanical Design*, 136(4):041005, February 2014. ISSN 1050-0472. doi:10.1115/1.4026463. URL http://mechanicaldesign.asmedigitalcollection.asme.org/article.aspx?doi=10.1115/1.4026463.

[8] Jon Bentley. Programming pearls: Little Languages. *Communications of the ACM*, 29(8):711–721, August 1986. ISSN 00010782. doi:10.1145/6424.315691. URL http://portal.acm.org/citation.cfm?doid=6424.315691.

[9] Leif P. Berg, Sara Behdad, Judy M. Vance, and Deborah Thurston. Disassembly Sequence Evaluation Using Graph Visualization and Immersive Computing Technologies. In *ASME 2012 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, pages 1351–1359, Chicago, Illinois, August 2012. ASME. ISBN 978-0-7918-4501-1. doi:10.1115/DETC2012-70388. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?doi=10.1115/DETC2012-70388.

[10] Allen Bierbaum, Christopher Just, Patrick Hartling, Kevin Meinert, Albert Baker, and Carolina Cruz-Neira. VR Juggler: a virtual platform for virtual reality application development. In *IEEE Virtual Reality Conference (VR 2001)*, pages 89–96, Los Alamitos, CA, USA, 2001. IEEE Comput. Soc. ISBN 0-7695-0948-7. doi:10.1109/VR.2001.913774. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=913774.

[11] Aron Bierbaum, Patrick Hartling, Pedro Morillo, and Carolina Cruz-Neira. Implementing Immersive Clustering with VR Juggler. In Osvaldo Gervasi, Marina Gavrilova, Vipin Kumar, Antonio Laganà, Heow Lee, Youngsong Mun, David Taniar, and Chih Tan, editors, *Computational Science and Its Applications - ICCSA 2005*, volume 3482 of *Lecture Notes in Computer Science*, pages 1119–1128. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-25862-9. doi:10.1007/11424857_120. URL http://www.springerlink.com/content/xl12urj4lea3ycyn.

[12] Patrick Carlson, Carl Kirpes, Ryan A. Pavlik, Judy M. Vance, Livien Yin, Terrence Scott-Cooper, and Troy Lambert. Comparison of Single-Wall Versus Multi-Wall Immersive Environments to Support a Virtual Shopping Experience. In *ASME 2011 World Conference on Innovative Virtual Reality (WINVR 2011)*, pages 287–291, Milan, Italy, 2011. ASME. ISBN 978-0-7918-4432-8. doi:10.1115/WINVR2011-5582. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1623673.

[13] Juan Sebastián Casallas, James H. Oliver, Jonathan W. Kelly, Frédéric Merienne, and Samir Garbaya. Towards a model for predicting intention in 3D moving-target selection tasks. In *Engineering Psychology and Cognitive Ergonomics (Part of HCI International)*,

pages 13–22. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-39360-0_2. URL http://link.springer.com/chapter/10.1007/978-3-642-39360-0_2.

[14] Juan Sebastián Casallas, James H. Oliver, Jonathan W. Kelly, Frédéric Merienne, and Samir Garbaya. Using relative head and hand-target features to predict intention in 3D moving-target selection. In *IEEE Virtual Reality Conference (VR 2014)*, pages 51–56. IEEE, March 2014. ISBN 978-1-4799-2871-2. doi:10.1109/VR.2014.6802050. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6802050.

[15] Melvin E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, July 1963. ISSN 00010782. doi:10.1145/366663.366704. URL http://portal.acm.org/citation.cfm?doid=366663.366704.

[16] Carolina Cruz-Neira, Daniel J. Sandin, and Thomas A. DeFanti. Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques - SIGGRAPH '93*, pages 135–142, New York, New York, USA, 1993. ACM Press. ISBN 0897916018. doi:10.1145/166117.166134. URL http://portal.acm.org/citation.cfm?doid=166117.166134.

[17] Ana Lúcia de Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems*, 31(2):1–31, February 2009. ISSN 01640925. doi:10.1145/1462166.1462167. URL http://portal.acm.org/citation.cfm?doid=1462166.1462167.

[18] Ana Lúcia de Moura, Noemi Rodriguez, and Roberto Ierusalimschy. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, 2004. doi:10.3217/jucs-010-07-0910. URL http://www.jucs.org/jucs_10_7/coroutines_in_lua/de_moura_a_l.pdf.

[19] Ulrich Eck and Christian Sandor. TINT: Towards a Pure Python Augmented Reality Framework. In *Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS) in IEEE Virtual Reality*, 2010. URL http://www.magicvisionlab.com/pub/eck_searis10/paper.pdf.

[20] Bruce Eckel. Strong Typing vs. Strong Testing. In Joel Spolsky, editor, *The Best Software Writing I*, pages 67–77. Apress, 2005. ISBN 978-1-4302-0038-3. doi:10.1007/978-1-4302-0038-3_11. URL http://www.springerlink.com/index/g575t57k372hth60.pdf.

[21] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Programming Languages: Implementations, Logics, and Programs*, pages 369–388. Springer, 1997. doi:10.1007/BFb0033856. URL http://www.springerlink.com/index/w280n4560526w967.pdf.

[22] Martin Fowler. A Pedagogical Framework for Domain-Specific Languages. *IEEE Software*, 26(4):13–14, July 2009. ISSN 0740-7459. doi:10.1109/MS.2009.85. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5076452.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, volume 206. Addison-Wesley, Reading, MA, 1995.

[24] N.K. Gupta, L.J. Jagadeesan, E.E. Kouteofios, and D.M. Weiss. Auditdraw: generating audits the FAST way. In *Proceedings of ISRE '97: 3rd IEEE International Symposium on Requirements Engineering*, pages 188–197. IEEE Comput. Soc. Press, 1997. ISBN 0-8186-7740-6. doi:10.1109/ISRE.1997.566869. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=566869.

[25] Patrick Hartling. VR Juggler: The Programmer Guide, 2010. URL http://vrjuggler.org/docs/vrjuggler/3.0/programmer.guide/programmer.guide.pdf.

[26] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196–es, December 1996. ISSN 03600300. doi:10.1145/242224.242477. URL http://dl.acm.org/citation.cfm?id=242477.

[27] Paul Hudak. Modular domain specific languages and tools. In *Proceedings. Fifth International Conference on Software Reuse*, pages 134–142, Victoria, BC , Canada, 1998. IEEE Comput. Soc. ISBN 0-8186-8377-5. doi:10.1109/ICSR.1998.685738. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=685738.

[28] Walter L. Hürsch and Cristina Videira Lopes. Separation of Concerns. Technical report, College of Computer Science, Northeastern University, 1995. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.5223.

[29] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua – An Extensible Extension Language. *Software: Practice and Experience*, 26(6):635–652, June 1996. ISSN 0038-0644. doi:10.1002/(SICI)1097-

024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P. URL http://doi.wiley.com/10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P.

[30] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of Lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages – HOPL III*, pages 2–1—-2–26, New York, New York, USA, 2007. ACM Press. ISBN 978159593766X. doi:10.1145/1238844.1238846. URL http://portal.acm.org/citation.cfm?doid=1238844.1238846.

[31] R. Kuck, J. Wind, K. Riege, and M. Bogen. Improving the AVANGO VR/AR framework – Lessons learned. In *Workshop Virtuelle und Erweiterte Realität*, pages 209–220, Magdeburg, 2008. URL http://www.avango.org/raw-attachment/wiki/Res/Improving_the_AVANGO_VR-AR_Framework--Lessons_Learned.pdf.

[32] Lionello A. Lombardi and Bertram Raphael. Lisp as the language for an incremental computer. Technical report, MIT Sloan School of Management, Cambridge, MA, USA, 1964. URL http://hdl.handle.net/1721.1/48305.

[33] John McCarthy. Recursive functions symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, April 1960. ISSN 00010782. doi:10.1145/367177.367199. URL http://doi.acm.org/10.1145/367177.367199.

[34] John McCarthy. History of LISP. In Richard L Wexelblat, editor, *History of Programming Languages I*, pages 173–185. ACM, New York, NY, USA, June 1978. ISBN 0-12-745040-8. doi:10.1145/800025.1198360. URL http://dl.acm.org/citation.cfm?id=800025.1198360.

[35] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, December 2005. ISSN 03600300. doi:10.1145/1118890.1118892. URL http://dl.acm.org/citation.cfm?id=1118892.

[36] Mike Oren, Patrick Carlson, Stephen Gilbert, and Judy M. Vance. Puzzle assembly training: Real world vs. virtual environment. In *IEEE Virtual Reality Conference (VR 2012)*, pages 27–30, Orange County, California, March 2012. IEEE. ISBN 978-1-4673-1246-2. doi:10.1109/VR.2012.6180873. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6180873.

[37] Dave Pape. A hardware-independent virtual reality development system. *IEEE Computer Graphics and Applications*, 16(4):44–47, July 1996. ISSN 02721716.

doi:10.1109/38.511852. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=511852.

[38] David Lorge Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976. ISSN 0098-5589. doi:10.1109/TSE.1976.233797. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1702332.

[39] Ryan A. Pavlik and Judy M. Vance. Expanding Haptic Workspace for Coupled-Object Manipulation. In *ASME 2011 World Conference on Innovative Virtual Reality (WINVR 2011)*, pages 293–299, Milan, Italy, 2011. ASME. ISBN 978-0-7918-4432-8. doi:10.1115/WINVR2011-5585. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1623675.

[40] Ryan A. Pavlik, Judy M. Vance, and Greg R. Luecke. Interacting with a Large Virtual Environment by Combining a Ground-based Haptic Device and a Mobile Robot Base. In *ASME 2013 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, Portland, Oregon, 2013. ASME. doi:10.1115/DETC2013-13441. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1830274.

[41] Dirk Reiners, Gerrit Voß, and Johannes Behr. OpenSG: basic concepts. In *Proc. of OpenSG Symposium 2002*, 2002. URL http://www.lsi.usp.br/~mkzuffo/PSI5787/opensg/Reiners_Basics.pdf.

[42] Erik J. Sandewall. Programming in an Interactive Environment: the "Lisp" Experience. *ACM Computing Surveys*, 10(1):35–71, January 1978. ISSN 03600300. doi:10.1145/356715.356719. URL http://dl.acm.org/citation.cfm?id=356715.356719.

[43] Benjamin Schaeffer and Camille Goudeseune. Syzygy: native PC cluster VR. In *IEEE Virtual Reality Conference (VR 2003)*, volume 15, pages 15–22. IEEE Comput. Soc, 2003. ISBN 0-7695-1882-6. doi:10.1109/VR.2003.1191116. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1191116.

[44] Abhishek Seth, Hai-Jun Su, and Judy M. Vance. Development of a Dual-Handed Haptic Assembly System: SHARP. *Journal of Computing and Information Science in Engineering*, 8(4):044502, 2008. ISSN 15309827. doi:10.1115/1.3006306. URL http://computingengineering.asmedigitalcollection.asme.org/article.aspx?articleid=1401373.

[45] Wesley Smith and Graham Wakefield. Real-time multimedia composition using Lua. In *Proceedings of the Digital Art Weeks 2007*, 2007. URL http://www.mat.ucsb.edu/Publications/07_SmithWakefield_DAW_Real-time_MultimediaCompositionUsingLua.pdf.

[46] S.A. Thibault, R. Marlet, and C. Consel. Domain-specific languages: from design to implementation application to video device drivers generation. *IEEE Transactions on Software Engineering*, 25(3):363–377, 1999. ISSN 00985589. doi:10.1109/32.798325. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=798325.

[47] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000. ISSN 03621340. doi:10.1145/352029.352035. URL http://portal.acm.org/citation.cfm?doid=352029.352035.

[48] Gordon Wetzstein, Moritz Göllner, Stephan Beck, Felix Weiszig, Sebastian Derkau, Jan P. Springer, and Bernd Fröhlich. HECTOR – Scripting-Based VR System Design. In *ACM SIGGRAPH 2007 posters*, volume 143, San Diego, California, 2007. ACM. doi:10.1145/1280720.1280876. URL http://doi.acm.org/10.1145/1280720.1280876.

CHAPTER 4. INTERACTING WITH GRASPED OBJECTS IN EXPANDED HAPTIC WORKSPACES USING THE BUBBLE TECHNIQUE

A paper submitted to the ASME *Journal of Computing and Information Science in Engineering*

Ryan A. Pavlik, Judy M. Vance

4.1. Abstract

Haptic force-feedback can provide useful cues to users of virtual environments. Body-based haptic devices are portable but the more commonly used ground-based devices have workspaces that are limited by their physical grounding to a single base position and their operation as purely position-control devices. The "bubble technique" has recently been presented as one method of expanding a user's haptic workspace. The bubble technique is a hybrid position-rate control system in which a volume, or "bubble," is defined entirely within the physical workspace of the haptic device. When the device's end effector is within this bubble, interaction is through position control. When the end effector moves outside this volume, an elastic restoring force is rendered, and a rate is applied that moves the virtual accessible workspace. Publications have described the use of the bubble technique for point-based touching tasks. However, when this technique is applied to simulations where the user is grasping virtual objects with part-to-part collision detection, unforeseen interaction problems surface. Methods of addressing these challenges are introduced, along with discussion of their implementation and an informal investigation.

Ryan A. Pavlik was the primary author and researcher.
Based on an earlier paper presented as:
R.A. Pavlik and J.M. Vance, "Expanding Haptic Workspace for Coupled-Object Manipulation," in *ASME 2011 World Conference on Innovative Virtual Reality (WINVR 2011)*, 2011, pp. 293–299. doi:10.1115/WINVR2011-5585
That publication is Copyright ©2011 by ASME

4.2. Introduction

Virtual assembly involves manipulation of computer-aided design (CAD) models to simulate assembly processes. Haptic force-feedback augments visual and audio feedback to provide physical feedback indicating the interaction between grasped objects and the environment [13, 2]. However, the limited physical workspace of ground-based haptic devices reduces the utility of these devices in large virtual environments. The bubble technique has been presented as one method of allowing a user to interact within a large virtual environment using a grounded haptic device with a smaller workspace. This technique holds great promise for expanding the potential of haptic interaction in virtual environments. This paper proposes improvements to the bubble technique to support force-feedback for users when they are manipulating grasped objects in large virtual environments. A discussion of current methods to increase the haptic workspace is followed by specific evaluation and algorithmic development of improvements to the bubble technique.

4.2.1. Increasing the Haptic Workspace for Ground-Based Devices

Ground-based haptic force feedback devices result in a limited workspace. Some devices, such as the SPIDAR, are specifically designed for large workspaces without modification [5]. Other techniques involve physical modifications to the haptic device which allow the device to travel within the virtual environment [17, 4, 11, 22, 12, 14, 21]. The focus of this paper is on increasing the haptic workspace of a fixed-base haptic device, such as the Haption Virtuose™ 6D35-45 in a CAVE with displays on two walls plus the floor as shown in Fig. 4.1. One technique of increasing the haptic workspace is to apply scaling in position control. As proposed by Fischer and Vance, the ratio of virtual workspace size to physical workspace size can be used to ensure an entire virtual volume is reachable [10]. Scaling increases the reachable workspace and eases coarse manipulation, but makes fine manipulation more difficult.

Pioneered by Dominjon et al. [8, 9], the bubble technique is a hybrid haptic control technique for expanding the haptic workspace. It supports fine manipulation as well as access to a larger effective working volume by moving the workspace under some conditions. The bubble technique uses pure position-control within a spherical volume, referred to as the “bubble.” The bubble is centered in the device’s physical workspace and sized to encompass the specific haptic device’s working volume. Movement of the haptic end effector outside of the bubble applies a velocity to the workspace, effectively moving the workspace to another position within the virtual scene. The user feels a slight elastic restoring force as he/she pulls the bubble to the new location. Once the bubble reaches its

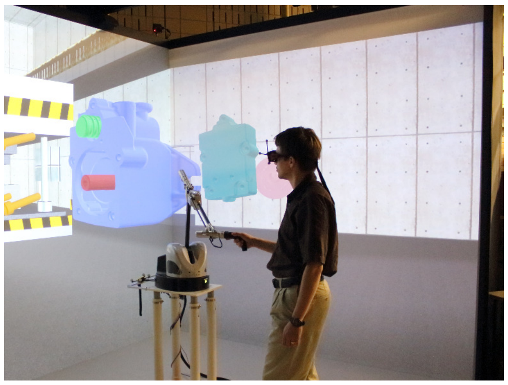


Figure 4.1: Haption Virtuose™ 6D35-45 in a large workspace virtual environment

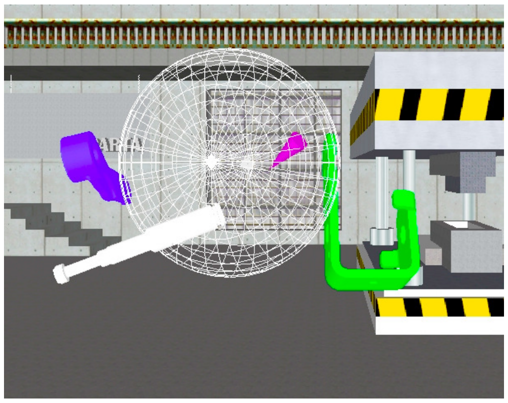


Figure 4.2: Wire-frame workspace display in virtual assembly application SPARTA

new location, the user freely operates the haptic device in that area of the virtual scene, feeling appropriate forces. For a device like the Haption Virtuose™ 6D35-45 that has a relatively large working volume, the bubble technique provides users with the ability to feel forces from interactions in areas of the virtual scene that are beyond the device's physical working volume. For smaller desktop-size devices like the Sensable Phantom Omni®, which has a limited working volume, this technique provides a valuable method to increase the overall haptic working volume, provided frequent movement outside the bubble is acceptable. The original research [8, 9] discussed a number of variations on the original bubble technique. These include the presence or absence of a visual indication of the workspace, such as the wire-frame sphere shown in Fig. 4.2, the implementation of a small scaling factor to control the quality of fine manipulation incrementally, and the potential of moving the camera (the user's viewpoint) with the bubble, effectively permitting navigation based on haptic interaction with the environment. Another related hybrid position-rate control technique, which is designed for a two-dimensional, non-haptic input device, is RubberEdge [6].

In Dominjon et al. [9], the bubble technique is described and studied in the context of a point-touching application, without grasped objects or object-to-object interaction. The implementation as found in the VirtuoseAPI allows application of the bubble technique to both point-touching and grasped object simulations. However, we have found that the use of the bubble technique with grasped object manipulation presents interaction challenges. This paper outlines those challenges and describes proposed solutions.

4.2.2. Interaction Device Design and Bubble Restoring Forces

Zhai and Milgram [25, 24] enumerate a few dimensions of the design space for input devices, of which *transfer function*, typically ranging from position to rate controlled, and *controller resistance* (isotonic through isometric) are most applicable to the current discussion. Isotonic devices allow muscle contraction and movement with low resistance: essentially the default mode of haptic devices as well as desktop computer mice. Isometric resistance refers to contraction with high resistance and little or no movement, *e.g.* Spaceball/SpaceMouse™-type devices or pointing sticks found on laptop computer keyboards. The range between these two extremes, in which the sensor provides some stiffness, includes elastic resistance (varying with displacement), viscous resistance (varying with velocity), and inertial resistance (varying with acceleration). Zhai and Milgram studied the performance of various devices in a two-dimensional design space. As the examples of desktop mice and laptop pointing sticks might suggest, isotonic resistance devices performed best for actions where position control was critical and isometric resistance devices performed best where rate or velocity control was most critical. Their data analyses showed a clear interaction of these two dimensions. Applying the bubble technique effectively turns a haptic device (typically an isotonic-position control device) into a device with both position and rate control transfer functions. It follows, then, that when rate control is active, the user would be well-served to feel isometric device resistance, which can be simulated by the force output capabilities of the haptic device. In the bubble technique, when the user moves the haptic end effector outside of the bubble, an elastic restoring force transforms the haptic device from an isotonic to an isometric device. This serves an important purpose in not only signaling to users an exit from the bubble, but also pulling the user back out of rate control into position control. Though later discussion will reveal problems related to the inclusion of an elastic restoring force, it serves an important function in the overall bubble technique and its removal is not a practical option.

4.3. Investigation of Interaction Challenges

Three interaction challenges were observed when applying the bubble technique in a virtual assembly application. Two challenges involved the experience of object-object collision while outside of the bubble. The elastic restoring forces of the bubble and the forces due to the manipulation of objects cannot be distinguished. This can confuse the user when object collisions occur when the end effector is outside of the bubble. In addition, the movement of the bubble when objects are colliding can result in a perceived "stickiness" when attempting to separate colliding objects. Finally, the visualization of the reachable workspace as a spherical volume or "bubble" can distract from the other visual feedback provided by the simulation.

4.3.1. Implementation Platform

The present work was implemented in SPARTA, the *Scriptable Platform for Research and Teaching in Assembly* [19]. This application, the successor to SHARP [23], provides a virtual reality environment where arbitrary computer-aided design (CAD) models can be loaded and manipulated using physically-based modeling and haptic force feedback. It builds on the VR Juggler open-source virtual reality software framework to support a wide variety of hardware and software platforms [3]. Model loading, triangulated data-structures, and graphics rendering is provided by the OpenSceneGraph library[1] working in concert with VR Juggler. The VR JuggLua framework, which extends VR Juggler with Lua scripting capabilities [20], maintains the visual and audio feedback, and provides rapid prototyping of immersive interaction in the simulation. SPARTA itself uses configuration scripts written in Lua code to load models, connect and configure devices, and launch the simulation. A run-time Lua console allows interactive re-configuration of the simulation.

Collision detection and physically-based modeling is based on the Voxmap PointShell™ (VPS) software developed by McNeely et al. [15, 16] and licensed from Boeing. VPS permits collision detection and force rendering involving arbitrary geometries at very high rates. It operates by performing discretization of input geometries into voxels, and performing voxmap sampling to detect collision and compute forces. SPARTA incorporates software that connects VPS to OpenSceneGraph, allowing arbitrary portions of the scene-graph to be voxelized at run-time without a separate preprocessing step.

The implementation of interaction devices in SPARTA is modular [18] and polymorphic, with wand, glove, and haptic devices all presenting a generic device interface that hides the details of individual device types from the simulation. SPARTA also provides for

[1] `http://www.openscenegraph.org`

objects that behave as virtual "filter" devices, modifying input and output rather than corresponding directly to physical hardware. Instead, while these filter objects present a device interface for use in the simulation, they also take a generic device object as input. The base class for such virtual filter devices provides method implementations that directly forward calls to the contained device, providing pass-through behavior by default. Derived implementations then selectively override these default methods to produce specific effects as desired. For instance, SPARTA includes one such virtual filter device type that scales up position and velocity reports by a user-supplied coefficient and scales down forces correspondingly. These filter devices are akin to transformation nodes in a scene-graph data structure, except that their more general formulation permits them to apply a range of effects beyond spatial transformations.

In SPARTA, the bubble technique is implemented as a virtual filter device that takes as input an existing device object, the radius and center of the desired bubble with respect to the device's workspace, the elastic stiffness for the bubble's restoring force, and details of the rate control function. This permits a single implementation of the bubble technique to be used with all supported device types. It also permits the combination of scaling with the bubble technique, which has been useful in when working with desktop devices such as the Phantom Omni in this research. Figure 4.2 is a screenshot of the SPARTA environment showing three geometry models, the bubble represented as a wire-frame sphere and the location of the haptic end effector as indicated by the virtual cone object.

Distinguishing Bubble Force from Collision-Related Force

Understanding the various forces that are calculated as a result of manipulation and/or collision is important in understanding issues that occur when using the bubble technique. When haptically manipulating grasped objects in a virtual environment, the primary source of forces rendered to the haptic device is the spring-damper system of the "virtual coupler." Initially proposed by Colgate et al. [7] and further investigated by Adams and Hannaford [1], the virtual coupler connects the virtual representation of the haptic end effector's position (also known as the haptic handle) to the grasped virtual object by a critically-damped spring-damper system (see Figure 4.3). The model of the virtual coupler contains both linear and torsional components. Stiffness values are assigned to the linear and torsional springs of the virtual coupler. The stiffness constants are determined empirically and related to the haptic device's capabilities and time step. When using the haptic device to perform free movement of objects in the virtual scene, displacement of the haptic handle results in the calculation of a reaction force based on the mass and inertia of the grasped object, which is then used to calculate the object's new position. The reaction force is also

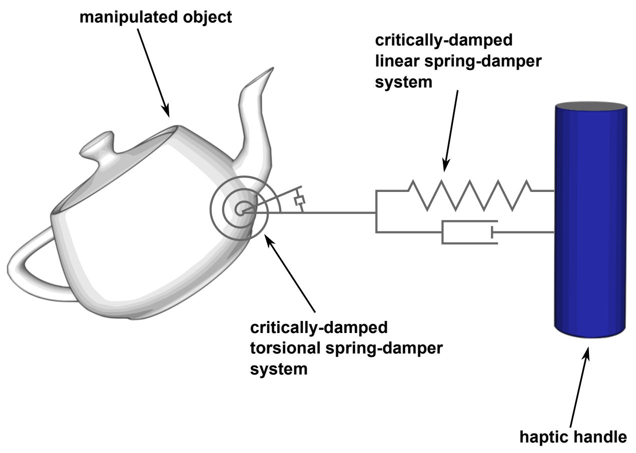

manipulated object
critically-damped linear spring-damper system
critically-damped torsional spring-damper system
haptic handle

Figure 4.3: Conceptual view of the virtual coupler

rendered to the haptic device, which conveys a sense of comparative mass. Because the system is critically-damped, the new computed location of the grasped object tends to achieve the position and orientation of the haptic handle as rapidly as permitted by the specified stiffness, without oscillations and other instabilities. This presents to the user as if he/she were grasping the object with the haptic end effector and moving it in virtual space. Use of the virtual coupler helps ensure overall system passivity. The virtual coupler is conceptually illustrated in Fig. 4.3, with conventions that will be used throughout this work: a cylinder as the haptic handle, linked by a spring-damper system to a teapot as the grasped object. The distance between the haptic handle and grasped object is exaggerated for clarity in this figure. The result of using the virtual coupler is that forces of collision are not directly transmitted to the haptic device, but are felt because of an increase in the virtual coupler spring displacement. Due to the high update rate, the virtual coupler can convey fairly detailed information about the shapes of colliding objects, such as initial contact, ridges, etc.

In the bubble technique, when the end effector moves outside of the bubble, the haptic workspace moves in the direction of the end effector motion and an elastic restoring force is rendered to the haptic device that pulls the haptic handle back in the direction of the bubble. Recall that this restoring elastic force is desirable as Zhai and Milgram found that rate control using an isotonic (non-elastic) device resulted in poor performance [25]. When holding an object and moving outside of the bubble, as illustrated in Fig. 4.4, the grasped object moves in the same direction that the bubble moves. When the grasped object collides with another object, collision forces due to object-to-object interactions will be generated on the haptic handle. These collision forces will be in the same direction as the restoring elastic force. Since net forces are the summation of all forces acting on a body, forces due

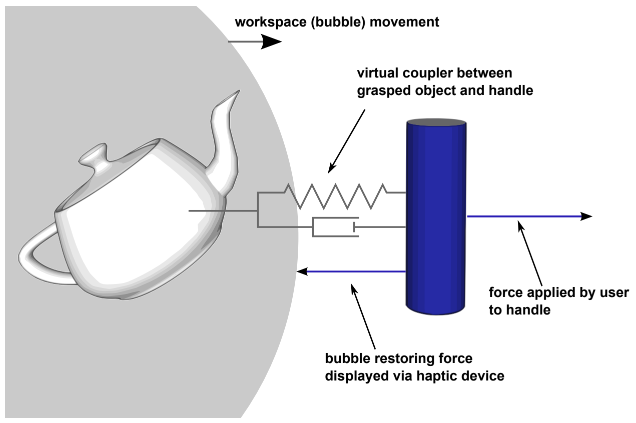

workspace (bubble) movement
virtual coupler between grasped object and handle
force applied by user to handle
bubble restoring force displayed via haptic device

Figure 4.4: Grasping object while moving bubble

to collision and forces due to the bubble technique cannot be distinguished. This weakens the haptic cues provided by collision during assembly. When the haptic handle is already experiencing a force, the addition of a collision force appears incremental. Furthermore, current haptic devices do not have the ability to produce large reaction forces, so a collision may go entirely unnoticed by the user if the elastic restoring force saturates the device's capabilities for force output. When the workspace velocity is proportional to the distance outside the bubble, users tended to seek high velocities by "pushing through" the bubble. This situation results in high velocities of movement and high elastic restoring forces which are likely to saturate the device's output.

A combination of techniques can be used to convey to the user when a collision has occurred. One approach to distinguishing these two sources of forces is to decrease the elastic stiffness of the bubble itself by setting a low stiffness constant. By decreasing the intensity of the bubble forces, the user's ability to move fast enough to saturate the force-rendering capabilities of the device during bubble movement will decrease, leaving capacity for rendering increased forces upon collision. However, this presents a trade-off between available force capacity for collision and clarity of the elastic re-centering cues during rate control. The use of additional sensory cues, both visual and audio, is a promising solution. Even a simple sound effect played upon starting collision draws attention to the transition between free-space movement and collision when manipulating an object in the rate-control area of the workspace, yet it does not physically change the feel of the collision. The metallic clang sound effect used in SPARTA clearly indicates a change in the virtual environment, despite potentially unchanged or minimally changed force output.

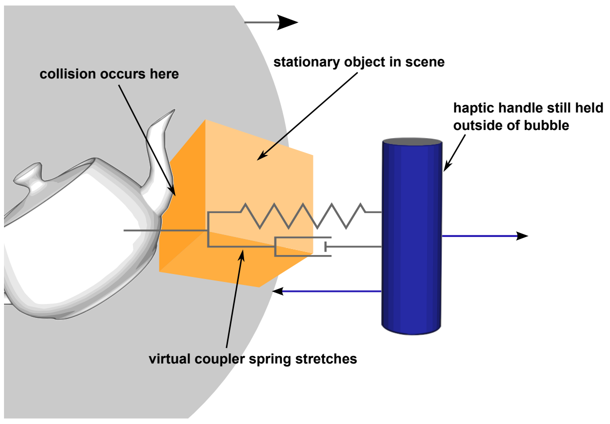
Similar visual changes, such as color or transparency are effective cues that a collision has occurred.

Bubble Movement During Collision

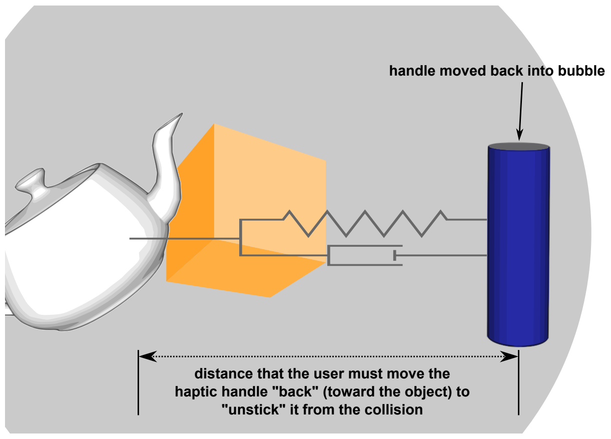
There is another issue that occurs when the grasped object collides with another object in the virtual environment. In a pure position-control system, moving the haptic device away from a colliding object quickly moves the grasped object. In turn, this decreases forces rendered because no collision impedes the restoration of the grasped object to the pose of the haptic handle, so the spring displacement (due to the virtual coupler) between the object and handle can quickly approach zero. However, when using the bubble technique to move the workspace, a "stickiness" occurs during object-to-object collisions when the end effector is outside of the bubble.

As the bubble and the grasped object move, the object may collide with other objects in the scene. Collision forces on the object prevent it from moving through the obstruction, normally generating a haptic cue by rendering increased forces through the haptic device. However, as discussed earlier, sustained, swift movement of the bubble may actively produce large bubble restoring forces that mask the forces related to the collision. When users fail to feel the collision of objects, they continue to apply enough force to the device to counter all forces rendered, keeping the haptic handle outside of the bubble. As a result, the bubble continues to move, thus continuing to move the effective position of the haptic handle in the virtual environment, as illustrated in Fig. 4.5a.

Moving the end effector back within the bubble is expected to produce a decrease in forces felt due to the elimination of bubble restoration forces. The user also expects the grasped object will move away from the colliding object. In fact, neither takes place. While bubble restoring forces are eliminated, the large displacement between the handle and the grasped object, resulting from bubble movement, continues to provide collision forces. Furthermore, since the virtual location of the workspace has moved, the user now must move the end effector in the opposite direction (and of equal magnitude of the bubble's movement during collision) before the colliding objects are pulled apart. These two connected phenomena are illustrated in Fig. 4.5b. The subjective experience of this situation is of a "stickiness" that prevents a user from being able to easily separate objects once they have collided. Attending to the displayed location of the haptic handle and its changed relationship with respect to the grasped object would reveal the true state of the simulation and how to disentangle the objects. However, since the handle's visual representation is generally less prominent and less subjectively meaningful than the visualization of the grasped object itself during manipulation, this remains a frustrating challenge to users.


collision occurs here
stationary object in scene
haptic handle still held outside of bubble
virtual coupler spring stretches

(a) Colliding objects while moving bubble


handle moved back into bubble
distance that the user must move the haptic handle "back" (toward the object) to "unstick" it from the collision

(b) Handle returns within bubble

Figure 4.5: Behavior of virtual coupler when colliding and moving bubble

Addressing this problem is complex. One technique is to stop the bubble from moving during collision, even if the haptic handle is outside of the bubble. This is not a very viable option. The nature of the collision-detection computation actually results in oscillation between states of collision and non-collision. This requires developing a state rule to determine when to stop bubble movement yet avoid oscillation. One solution would be to determine a new collision by comparing the current collision count to a short-term maximum collision count. Usually, this will avoid the cycle problem. However, stopping the bubble during collision poses new problems (when and how should it be re-started?) and reduces a worthwhile aspect of the bubble's effect: the bubble also represents the region of the device with the highest fidelity feedback so keeping the handle within it has merit on its own.

Another approach is to change the rate control law governing the bubble displacement such that large movements outside the bubble are discouraged and therefore, force saturation is less likely to occur. The bubble rate and elastic force are along the same direction as the vector from the center of the bubble to the end effector. Thus, the end effector position, elastic force, and bubble rate can be represented as scalar distances, forces, and rates along this vector. As previously formulated [8], R is the radius of the bubble, and D is the distance between the center of the bubble and the end effector. The distance of the end effector outside of the bubble, x, is given in equation (4.1).

$$x = D - R \tag{4.1}$$

The restoring force, F, is modeled as a linear spring force as given in equation (4.2), where k is a constant.

$$F = -k \cdot x \tag{4.2}$$

Dominjon et al. propose a cubic, monotonic relationship between distance outside the bubble and rate of bubble movement as illustrated in equation (4.3), where V is bubble velocity and K' is a constant.

$$V = K' \cdot x^3 \tag{4.3}$$

A quadratic, monotonic relationship has also been implemented which produced similar results as the cubic relationship of equation (4.3).

An alternate rate control law that reduces the workspace rate after reaching a peak has been implemented and evaluated. This control law is referred to as a "peak ring" function. In essence, the function allows increasing velocity beyond the surface of the bubble until a predefined distance is achieved. At this "peak" distance, the peak velocity of the bubble is achieved. When a user pushes the end effector beyond this peak distance, the velocity of the bubble is reduced so that no additional advantage is gained by the user by moving farther away from the bubble. By producing the highest rate of bubble motion at a single peak just outside of the position-control region of the bubble, and quickly tapering off to a constant rate beyond this peak, the greatest bubble movement that a user will encounter will be confined to the area when the bubble renders a relatively small elastic force. This method reduces the user's tendency to continue to push harder against the bubble.

Choose x^* as the distance outside the bubble at which the peak rate, v^*, is achieved. This distance can be a function of the bubble radius, R. Let $0 < \beta < 1$ specify the percent of the total peak velocity that is achievable at all distances farthest away from the bubble surface. Define a quadratic function of x with its global maximum of $v^* = f(R \cdot x^*)$ as follows:

$$f(x) = -\frac{v^*}{x^{*2}}\,(x)\,(x - 2R \cdot x^*) \tag{4.4}$$

In the interval $x \in [0, 2x^*]$, $f(x)$ is positive. The velocity is determined by $f(x)$ in what can be called the "peak zone," a subset of $x \in [0, 2x^*]$ defined by the predicate

$$P(x) = (x < R \cdot x^*) \vee \left((x > R \cdot x^*) \wedge (f(x) > \beta \cdot v^*)\right) \tag{4.5}$$

In the tested implementation, parameter values of $R = 0.45$ (in meters), $x^* = 0.15$, $\beta = 0.3$, and $v^* = 1.5$ were chosen. The bubble rate for some distance outside of the bubble x is a

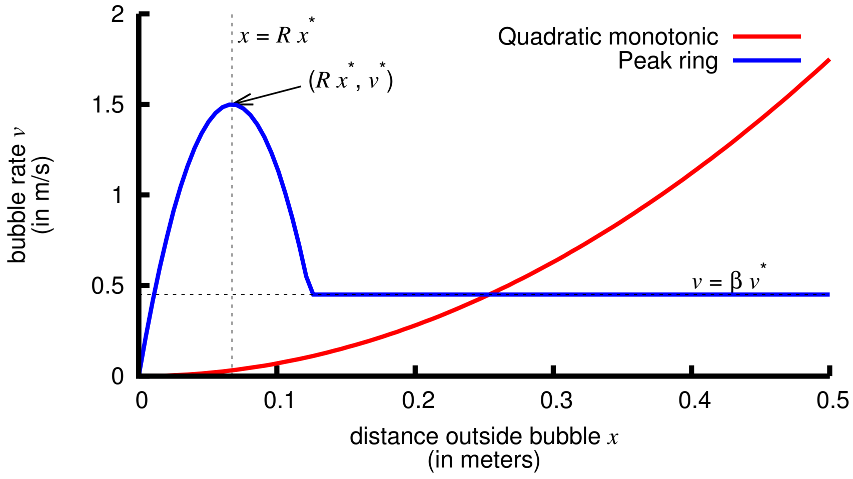


Figure 4.6: Control laws as investigated

piecewise function defined as

$$V = \begin{cases} f(x) & \text{if } P(x) \text{ is true} \\ \beta \cdot v^* & \text{otherwise} \end{cases} \tag{4.6}$$

Figure 4.6 shows a monotonic quadratic control law ($V = K' \cdot x^2$ with $K' = 7$) and this "peak ring" with parameters set as above. The specific values are not as relevant as the overall trends. Whereas the original bubble technique produces workspace movement in response to pushing out of the bubble, this modified control law can be described as producing movement by touching just outside the bubble.

This contributes to resolving the issues with grasped object manipulation in two ways. Since the most efficient movement occurs at a slight distance outside the bubble, the elastic restoring force is relatively small. With a small elastic force from the bubble, the device is less likely to be saturating its force-rendering ability with just the bubble force alone. If the manipulated object collides with another object, the collision effects transmitted through the virtual coupler will be more clearly felt with a lower "background level" of force from the bubble. Secondly, as this peak rate is located physically near the pure position-control area of the workspace, a user's action to move a grasped object away from a collision will result in the device leaving the rate-control zone in a short distance and short time. As implemented, this "peak ring" bubble rate function anecdotally improved the perception of collision forces during bubble movement, often resulting in the user stopping the movement of the bubble once collision occurred. This simple sample control law demonstrates the principle of finite peak velocity for the bubble technique in grasped object manipulation.

4.4. Bubble Visualization

The sphere-shaped volume of the workspace providing direct position-control has previously been visualized as a semi-transparent sphere [8, 9]. Dominjon et al. assert that dual-display of the spherical bounding volume (haptic and visual) is important and supports association of the physical and displayed workspace. SPARTA's bubble technique module includes three visualization modes: a semi-transparent sphere, a wire-frame sphere, and a no-visualization option. An informal evaluation of these different visualizations of the bubble while assembling CAD models was performed. Display of the wire-frame bubble (Fig. 4.2) seems to serve as a useful tool to support explanation of how the bubble technique works. However, assembly of complex CAD geometry appeared to be impeded by display of the bubble. The semi-transparent sphere obscures the geometry when opaque enough to clearly visualize the workspace volume. The wire-frame sphere does not occlude the geometry, but it appeared visually distracting and cluttered. In contrast, when display of the bubble was disabled, use of the haptic device to perform virtual assembly was natural with little conscious attention paid to the detail of the hybrid control.

[9] concluded that visual display of the sphere aided users in interacting in the virtual environment. Two hypotheses may explain the seeming contradiction with the assertion of the sphere display's importance. A virtual assembly application may present a higher task load than point-touching applications such as the one studied in [9]. Visualization of the workspace may impose a continued cognitive awareness of the hybrid position-rate control scheme, presenting difficulties in completing the original task. A second hypothesis is that visualization of the bubble during object manipulation presents a challenge of divided attention, with the sphere visuals serving to distract from the features of the manipulated geometry that facilitate or impede assembly.

In light of these findings, run-time-switchable display of the workspace bounding sphere has been implemented. This allows explanation of the bubble technique with the workspace clearly visualized, and subsequently allows actual use of the environment and completion of assembly tasks to proceed unobstructed with the bubble display disabled.

4.5. Conclusions and Future Work

Haptic interaction devices provide valuable cues in virtual reality simulations, but their physical workspace is often limited by the mechanics required to render stiff, realistic forces.

One particularly promising way of extending the workspace of ground-based haptic devices is to implement a hybrid position-rate control scheme rather than purely position-control. Investigation into extending the haptic workspace during grasped object manipulation, as needed for virtual assembly tasks, has identified several additional challenges. The lack of distinction between bubble and collision forces, and the fact that bubble movement may proceed even during collision, can result in inaccurate perceptions of force and an uncomfortable perceived stickiness between the colliding objects. A promising approach for addressing these issues is to use a non-monotonic rate control scheme for the bubble movement. A user can be discouraged from pushing too hard/fast against the bubble force by creating a peak bubble velocity a short distance outside of the position-only area of the bubble, rather than having the velocity continually increase with increased distance from the bubble. A implementation of such a "peak ring" control has been devised which combines quadratic and constant functions. Initial investigation of this method shows promise.

In the future, other methods of improving the bubble technique will be explored. An additional technique for distinguishing bubble and collision forces is to render an augmented "bump" effect upon the start of collision. The nature of penalty-based physics, where an object in collision is modeled as cycling in and out of collision rapidly, requires a careful detection of the start of a high-level collision event. Implementing an augmented bump would affect the user's hand position during subsequent time-steps, leaving less margin for error in determining the start of collision. Implementing and evaluating this force augmentation is planned.

Acknowledgment

The authors are grateful to William McNeely for technical assistance in ensuring a physically-based simulation platform; Haption S.A. for hardware support of the Virtuose™ device; to Patrick Carlson and Meisha Rosenberg of Iowa State University for their assistance with evaluation. This paper expands upon work initially presented at WINVR 2011. This work was performed at the Virtual Reality Applications Center at Iowa State University. This material is based upon work supported by the National Science Foundation under Grant No. CMMI-0928774.

Bibliography

[1] Richard J. Adams and Blake Hannaford. A two-port framework for the design of unconditionally stable haptic interfaces. In *IEEE/RSJ International Conference on*

Intelligent Robots and Systems, volume 2, pages 1254–1259, Victoria, BC, Canada, 1998. IEEE. ISBN 0-7803-4465-0. doi:10.1109/IROS.1998.727471. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=727471.

[2] Paolo Belluco, Monica Bordegoni, and Samuele Polistina. Multimodal Navigation for a Haptic-Based Virtual Assembly Application. In *ASME 2010 World Conference on Innovative Virtual Reality (WINVR 2010)*, pages 295–301. ASME, 2010. ISBN 978-0-7918-4908-8. doi:10.1115/WINVR2010-3743. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1617289.

[3] Allen Bierbaum, Christopher Just, Patrick Hartling, Kevin Meinert, Albert Baker, and Carolina Cruz-Neira. VR Juggler: a virtual platform for virtual reality application development. In *IEEE Virtual Reality Conference (VR 2001)*, pages 89–96, Los Alamitos, CA, USA, 2001. IEEE Comput. Soc. ISBN 0-7695-0948-7. doi:10.1109/VR.2001.913774. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=913774.

[4] Diego Borro, Joan Savall, Aiert Amundarain, Jorge Juan Gil, Alejandro Garcia-Alonso, and Luis Matey. A large haptic device for aircraft engine maintainability. *IEEE Computer Graphics and Applications*, 24(6):70–74, November 2004. ISSN 0272-1716. doi:10.1109/MCG.2004.45. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1355895.

[5] Laroussi Bouguila, Masahiro Ishii, and Makoto Sato. A large workspace haptic device for human-scale virtual environments. In *First International Workshop on Haptic Human-computer Interaction*, pages 86–91, Glasgow, UK, 2000. URL http://www.dcs.gla.ac.uk/~stephen/workshops/haptic/papers/bougilia-poster.pdf.

[6] Géry Casiez, Daniel Vogel, Qing Pan, and Christophe Chaillou. RubberEdge: reducing clutching by combining position and rate control with elastic feedback. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology - UIST '07*, page 129, New York, New York, USA, 2007. ACM Press. ISBN 9781595936792. doi:10.1145/1294211.1294234. URL http://portal.acm.org/citation.cfm?doid=1294211.1294234.

[7] J. Edward Colgate, Paul E. Grafing, Michael C. Stanley, and G Schenkel. Implementation of stiff virtual walls in force-reflecting interfaces. In *Proceedings of IEEE Virtual Reality Annual International Symposium*, pages 202–208. IEEE, 1993. ISBN 0-7803-1363-

1. doi:10.1109/VRAIS.1993.380777. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=380777.

[8] Lionel Dominjon, A. Lecuyer, Jean-Marie Burkhardt, Guillermo Andrade-Barroso, and Simon Richir. The "Bubble" Technique: Interacting with Large Virtual Environments Using Haptic Devices with Limited Workspace. In *First Joint Eurohaptics Conference and Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, volume i, pages 639–640. IEEE, 2005. ISBN 0-7695-2310-2. doi:10.1109/WHC.2005.126. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1407038.

[9] Lionel Dominjon, Jérôme Perret, and Anatole Lécuyer. Novel devices and interaction techniques for human-scale haptics. *The Visual Computer*, 23(4):257–266, March 2007. ISSN 0178-2789. doi:10.1007/s00371-007-0100-4. URL http://link.springer.com/10.1007/s00371-007-0100-4.

[10] A. Fischer and Judy M. Vance. PHANToM haptic device implemented in a projection screen virtual environment. In *Workshop on Virtual Environments 2003 - EGVE '03*, pages 225–229, New York, New York, USA, 2003. ACM Press. ISBN 3905673002. doi:10.1145/769953.769979. URL http://portal.acm.org/citation.cfm?doid=769953.769979.

[11] Florian Gosselin, Claude Andriot, Florian Bergez, and Xavier Merlhiot. Widening 6-DOF haptic devices workspace with an additional degree of freedom. In *Second Joint EuroHaptics Conference and Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems (WHC'07)*, pages 452–457. IEEE, March 2007. ISBN 0-7695-2738-8. doi:10.1109/WHC.2007.127. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4145216.

[12] Florian Gosselin, Claude Andriot, Joan Savall, and Javier Mart. Large Workspace Haptic Devices for Human-Scale Interaction: A Survey. In *EuroHaptics 2008*, pages 523–528, Madrid, Spain, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-69057-3_66. URL http://link.springer.com/chapter/10.1007/978-3-540-69057-3_66.

[13] Chang E. Kim and Judy M. Vance. Using VPS (Voxmap PointShell) as the Basis for Interaction in a Virtual Assembly Environment. In *ASME 2003 Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, pages 1153–1161, Chicago, Illinois, 2003. ASME. doi:10.1115/DETC2003/CIE-48297. URL

http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1585721.

[14] In Lee, Inwook Hwang, Kyung-Lyoung Han, Oh Kyu Choi, Seungmoon Choi, and Jin S. Lee. System Improvements in Mobile Haptic Interface. In *World Haptics 2009 - Third Joint EuroHaptics Conference and Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, pages 109–114. IEEE, March 2009. ISBN 978-1-4244-3858-7. doi:10.1109/WHC.2009.4810834. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4810834.

[15] William A. McNeely, Kevin D. Puterbaugh, and James J. Troy. Six degree-of-freedom haptic rendering using voxel sampling. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques - SIGGRAPH '99*, pages 401–408, New York, New York, USA, 1999. ACM Press. ISBN 0201485605. doi:10.1145/311535.311600. URL http://portal.acm.org/citation.cfm?doid=311535.311600.

[16] William A. McNeely, Kevin D. Puterbaugh, and James J. Troy. Voxel-based 6-dof haptic rendering improvements. *Haptics-e*, 3(7), 2006. URL https://haptics-e.org/Vol_03/he-v3n7.pdf.

[17] Norbert Nitzsche, Uwe D. Hanebeck, and Günther Schmidt. Design Issues of Mobile Haptic Interfaces. *Journal of Robotic Systems*, 20(9):549–556, September 2003. ISSN 0741-2223. doi:10.1002/rob.10105. URL http://doi.wiley.com/10.1002/rob.10105.

[18] David Lorge Parnas, Paul C. Clements, and David Mandel Weiss. The Modular Structure of Complex Systems. *IEEE Transactions on Software Engineering*, SE-11(3): 259–266, March 1985. ISSN 0098-5589. doi:10.1109/TSE.1985.232209. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1702002.

[19] Ryan A. Pavlik and Judy M. Vance. Expanding Haptic Workspace for Coupled-Object Manipulation. In *ASME 2011 World Conference on Innovative Virtual Reality (WINVR 2011)*, pages 293–299, Milan, Italy, 2011. ASME. ISBN 978-0-7918-4432-8. doi:10.1115/WINVR2011-5585. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1623675.

[20] Ryan A. Pavlik and Judy M. Vance. VR JuggLua: A Framework for VR Applications Combining Lua, OpenSceneGraph, and VR Juggler. In *Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS) in IEEE Virtual Reality*, Singapore, 2011. IEEE. doi:10.1109/SEARIS.2012.6231166. URL http://dx.doi.org/10.1109/SEARIS.2012.6231166.

[21] Ryan A. Pavlik, Judy M. Vance, and Greg R. Luecke. Interacting with a Large Virtual Environment by Combining a Ground-based Haptic Device and a Mobile Robot Base. In *ASME 2013 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, Portland, Oregon, 2013. ASME. doi:10.1115/DETC2013-13441. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1830274.

[22] Angelika Peer, Yuta Komoguchi, and Martin Buss. Towards a Mobile Haptic Interface for bimanual manipulations. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 384–391. IEEE, October 2007. ISBN 978-1-4244-0911-2. doi:10.1109/IROS.2007.4399008. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4399008.

[23] Abhishek Seth, Hai-Jun Su, and Judy M. Vance. Development of a Dual-Handed Haptic Assembly System: SHARP. *Journal of Computing and Information Science in Engineering*, 8(4):044502, 2008. ISSN 15309827. doi:10.1115/1.3006306. URL http://computingengineering.asmedigitalcollection.asme.org/article.aspx?articleid=1401373.

[24] Shumin Zhai. *Human performance in six degree of freedom input control*. Ph.d. dissertation, University of Toronto, 1995.

[25] Shumin Zhai and Paul Milgram. Human performance evaluation of manipulation schemes in virtual environments. In *Proceedings of IEEE Virtual Reality Annual International Symposium*, pages 155–161. IEEE, 1993. ISBN 0-7803-1363-1. doi:10.1109/VRAIS.1993.380784. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=380784.

CHAPTER 5. INTERACTING WITH A LARGE VIRTUAL ENVIRONMENT BY COMBINING A GROUND-BASED HAPTIC DEVICE AND A MOBILE ROBOT BASE

A paper published in the proceedings of the *ASME 2013 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*

Ryan A. Pavlik, Judy M. Vance, Greg R. Luecke

Abstract

Ground-based haptic devices provide the capability of adding force feedback to virtual environments; however, the physical workspace of such devices is very limited due to the fixed base. By mounting a haptic device on a mobile robot, rather than a fixed stand, the reachable volume can be extended to function in full-scale virtual environments. This work presents the hardware, software, and integration developed to use such a mobile base with a Haption Virtuose™ 6D35-45. A mobile robot with a Mecanum-style omni-directional drive base and an Arduino-compatible microcontroller development board communicates with software on a host computer to provide a VRPN-based control and data acquisition interface. The position of the mobile robot in the physical space is tracked using an optical tracking system. The SPARTA virtual assembly software was extended to 1) apply transformations to the haptic device data based on the tracked base position, and 2) capture the error between the haptic device's end effector and the center of its workspace and command the robot over VRPN to minimize this error. The completed system allows use of the haptic device in a wide area projection screen or head-mounted display virtual environment, providing smooth free-space motion and stiff display of forces to the user

Ryan A. Pavlik was the primary author and researcher.
R.A. Pavlik, J.M. Vance, and G.R. Luecke, "Interacting with a Large Virtual Environment by Combining a Ground-based Haptic Device and a Mobile Robot Base," in *ASME 2013 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, 2013. doi:10.1115/DETC2013-13441

throughout the entire space. The availability of haptics in large immersive environments can contribute to future advances in virtual assembly planning, factory simulation, and other operations where haptics is an essential part of the simulation experience.

5.1. Introduction

The goal of this work is to expand the workspace of a ground-based haptic device to encompass the full area of a multi-wall projection screen or large area position tracked virtual reality facility. The result will support user interaction with force feedback within a large virtual reality facility. To illustrate, the large workspace Haption Virtuose™ 6D35-45 haptic device is designed to provide forces and torques within a cube-shaped workspace 45 cm on each side. Therefore, using this device, only the collision of virtual objects within the 45 cm cube-shaped workspace can provide force feedback to the user. Other small ground-based haptic devices have smaller workspace areas. With today's availability of large area position tracking systems, virtual reality facilities can be constructed of any size. Methods are needed to support using haptic devices in a large area position tracked virtual environment that could potentially include several square meters of floor space. The assembly scenarios of interest take place in a typical work cell of 2 m to 3 m square. Thus, expanding the effective workspace of the haptic device permits richer, more realistic simulations of assembly scenarios by allowing the user to walk around in the available physical space while still receiving haptic force and torque feedback. The approach is to mount a commercially-available haptic device on a powered omni-directional mobile robot, further develop a control scheme originally designed for smaller devices, and produce modular software that provides this functionality in a way that is easy to use. The major hardware components are commercial, off-the-shelf (COTS) components. It is the integration of the hardware into a full system, the extension of the control system and the software to integrate it all that is novel. The prototype system consists of a Haption Virtuose™ 6D35-45 haptic device, mounted on a stand which is placed on an omni-directional Mecanum-style mobile robot drive base.

5.2. Background

Body-based haptic devices are inherently mobile since they are grounded to the user; however, these devices can only provide relative forces, such as grasp forces. Ground-based haptic devices are, as the name implied, fixed to the ground so these devices are able to display absolute forces, such as weight. Our desire is to provide the ability to feel absolute forces and torques in a large area virtual reality facility so our research in focused on the use of ground-based devices only.

There are several existing approaches to extending haptic force feedback to a large working area. One approach involves simply building the haptic device as large as the physical virtual facility work area. Tensed-cable devices are based on a system of cables and motorized pulleys, starting with the pioneering SPIDAR [9], and proceeding on to its family of successors [15] and commercial adaptations of the design [7]. The cables are fastened to a fixed structure and are controlled with motorized pulleys. Within the virtual reality facility, one or more handles are attached to the cables. There is typically one more cable per handle than degrees of freedom in feedback. As the user moves the handle, the pulley motors actuate the cables to provide force and torque feedback and also to encode the position of the handle. One of the strengths of this design is that the size of the frame determines the extent of the force and torque feedback area, and the frame can be built to encompass very large spaces. However, these systems can be complex to control, and the same cables that offer such freedom of workspace size also present singularities which limit the orientations that are reachable and provide feedback [6].

Other approaches consist of the addition of redundant axes to ground based haptic devices to expand the workspace and overcome workspace singularities. Gosselin et al. suspended the Haption Virtuose 6D35-40 six degree-of-freedom haptic (6-DOF) from an overhead beam structure mounted to the floor. A DC motor and a back-drivable cable capstan reducer powered translation of the haptic device along the beam [5]. This extra degree of freedom expanded the haptic workspace in one direction along the length of the overhead beam. Borro et al. [3] took a similar, but different, approach. The LHIfAM consists of a custom designed haptic device attached to a vertically oriented gantry structure. The structure allows linear translation in the horizontal direction along a cross beam that extends between two fixed vertical end supports, and vertical translation as the cross beam slides up and down along the fixed vertical end supports. Both the Gosselin device and the LHIfAM are well suited to the application situation where a single projection screen displays virtual components for assembly and maintenance evaluation; however, this solution only extends the haptic workspace in one direction or the 2D vertical plane, which is not sufficient for reaching into a more general large scale virtual reality facility such as a CAVE™. Further development of Gosselin's research led to a commercialized device offered by Haption called Scale 1. This consists of a floor mounted structure that contains three orthogonal motorized booms which move the Virtuose 6D35-40 in 3D space [8]. The support structure is positioned outside of a multi-wall immersive environment with one of the supporting motorized beams extending into the space. Ueberle et al. developed a specialized device called the VISHARD10 to eliminate workspace singularities present in standard non-redundant 6DOF haptic devices [18]. VISHARD10 is a admittance controlled robotic arm with 10 degrees of freedom and a cylindrical workspace measuring 1.7 m in diameter and 0.6 m in height. It has the ability to produce large forces within the workspace; however, the cylindrical workspace in this configuration is height limiting and not able to support the standard reach height of a human. Like all the devices presented so far, its ground-based design requires permanent, or at least semi-permanent, installation and the device itself still limits the bounds of the haptic workspace.

One way of adding redundant axes to an existing haptic device to provide a theoretically limitless range of motion is through integrating the haptic device with a mobile robot, to produce what Nitzsche et al. call a "mobile haptic interface," or MHI [10]. The particular MHI discussed in Nitzsche et al. couples an omni-directional mobile base with a Sensable Phantom Premium 1.0 device. Later the mobile base was used with the VISHARD7 [14], the mobile-targeted successor of the VISHARD10. Barbagli et al. explored mounting Sensable Phantom Premium 1.5 devices on two different mobile robot bases [2]. They investigated the performance of those bases and the functioning of the combined control scheme through the use of a simulated user input to the haptic device. Despite the results of the work of Nitzsche et al. and Barbagli et al., Gosselin et al. de-emphasized the category of haptic devices with mobile bases in the general search for large-workspace haptic interaction in a 2008 review of the field [6], citing problems with slip and the negative visual impact of using such devices in a projection-screen environment. However, alternate robot types and control schemes for mobile haptics interfaces have yet to be explored, thus providing motivation for the current research presented in this paper.

5.3. Design and Implementation

Transparency when using a haptic device is highly desirable [10]. The user should be free to move about in the space whether or not a virtual object is being manipulated. Additionally, the user should feel appropriate collision forces throughout the space when manipulating an object. An example of the desired experience using the virtual assembly scenario follows. A user holds on to the end effector of the haptic device and walks within the physical confines of the virtual reality facility to the location of a virtual object of interest displayed in the virtual reality system. The user reaches out and virtually "grabs" the part by selecting it using the end effector. The user can then move elsewhere in the space, such as to a workbench, and "release" or place the virtual object with the rest of the virtual objects on the bench. Throughout these interactions, the mobile base and haptic device work together transparently.

To avoid confusion, the following nomenclature will be adopted for the remainder of this work. “Haptic device” shall refer to a ground-based haptic interaction device, such as a Haption Virtuose™ 6D35-45 or a Sensable Phantom Omni®. A coordinate system is defined at some fixed location on the haptic device referred to as the “base.” “Mobile robot” shall refer to the powered omni-directional mobile robot.

On a technical level, the system works as follows. As the user manipulates the end effector of the haptic device, the mobile robot moves the haptic device base to follow movements of the user through the space. This expands the usable workspace of the haptic device from the reachable volume of the haptic device to the entire position tracked area in the virtual reality (VR) facility. The mobile robot carries a position tracking target. The tracking system in the VR facility reports the position of the mobile robot relative to the room. Composing the room-to-base and base-to-end-effector position transforms results in the obtaining the overall position and orientation of the end effector in the room. This combined system effectively functions as a haptic device with a physical workspace encompassing a substantial portion of the volume in the virtual reality system, much larger than the physical workspace of the haptic device on its own.

The haptic device communicates with the computer simulation to report its end effector position and receive force commands, just as if it were on a fixed base. The optical tracking system in the VR system provides the location of the base of the haptic device in the physical space to the simulation. The mobile robot’s velocity (translational on the plane of the floor) is driven proportionally to the error between the haptic device’s end effector position and a configured neutral position relative to the base, both within the coordinate system of the haptic device. This control system is based on the scheme described by Garlington [4].

5.3.1. Hardware

The system’s hardware consists of three separate conceptual parts: a haptic device, a tracking target in the virtual reality system, and a mobile robot base. The experimental system described here uses a Haption Virtuose™ 6D35-45 haptic device, ART optical tracking, and an omni-directional mobile robot. The Virtuose provides substantial force capabilities with six actuated degrees of freedom (force and torque feedback) and a workspace roughly equivalent to a human arm pivoting at the shoulder. While this provides the desired experience and large workspace, the general system is not dependent on this specific device. For example, a Sensable Phantom Omni® device was mounted on the mobile robot early in the development process so software and controls integration could proceed concurrent with the work to develop the physical mount for the Virtuose device. Phantom Omni devices have three actuated (force feedback) and three passive (rotation sensing without torque feedback) degrees of freedom and a workspace roughly comparable to a human hand pivoting at the wrist. The conceptual integration and most of the software described in this work can be applied to many haptic devices, subject to the physical limitations of the Mecanum-style drive base used. The combination of the haptic device and the mobile robot needs to be tracked by the virtual reality system, so the method used varies according to the virtual reality system in which the mobile robot will operate. This experimental system was used within the Multimodal Experience Testbed and Laboratory ("METaL") immersive facility[1] at Iowa State University's Virtual Reality Applications Center. This $4\,\mathrm{m} \times 3\,\mathrm{m} \times 3\,\mathrm{m}$ CAVE™, with three projected surfaces (two walls and the floor), uses an ART TrackPack4 optical tracking system. The base of the Virtuose is tracked within METaL by attaching a "Claw" tracking target to it. This passive optical tracking target consists of four 20 mm diameter retro-reflective spheres rigidly arranged, which allows the four camera TrackPack4 system to determine the base's complete position and orientation within the room. The Virtuose is rigidly bolted to a stand holding it at a comfortable working height above the mobile robot. The entire setup in METaL can be seen in Fig. 5.1. One of the tracking cameras is visible in this photo above the corner between the screens, and the tracking target is partially visible just left of center, attached to the black base of the Virtuose. Some early testing of the software was done using a Phantom Omni instead of the Haption Virtuose and with another ART optical tracker as well as an InterSense IS900 hybrid ultrasonic-inertial tracking system.

The omni-directional mobile robot selected, pictured in Fig. 5.2, is based on the "4WD Mecanum Wheel Robot Kit," model 10011, from Nexus Robot.[2] This robot has four 100 mm diameter Mecanum wheels, which each have 9 rubber rollers at a 45° angle to the axle. By controlling the wheel directions and velocities individually, the Mecanum-style drive can move by translation in the plane in any direction, as well as rotation in the plane with or without simultaneous translation. Each wheel is driven by a Faulhaber 2342L012CR coreless, brushed DC motor, running on nominally 12 VDC, with a no-load speed of 8100 RPM and a recommended top speed of 7000 RPM. Each motor is mated to a Nexus-built 12 CPM optical quadrature encoder, as well as to a 64:1 planetary gearhead which drives the wheel. Driving the wheels at the recommended top speed results in an overall theoretical rate of the robot of 0.57 m/s.

[1] http://www.vrac.iastate.edu/METaL/

[2] http://nexusrobot.com/product.php?id_product=67

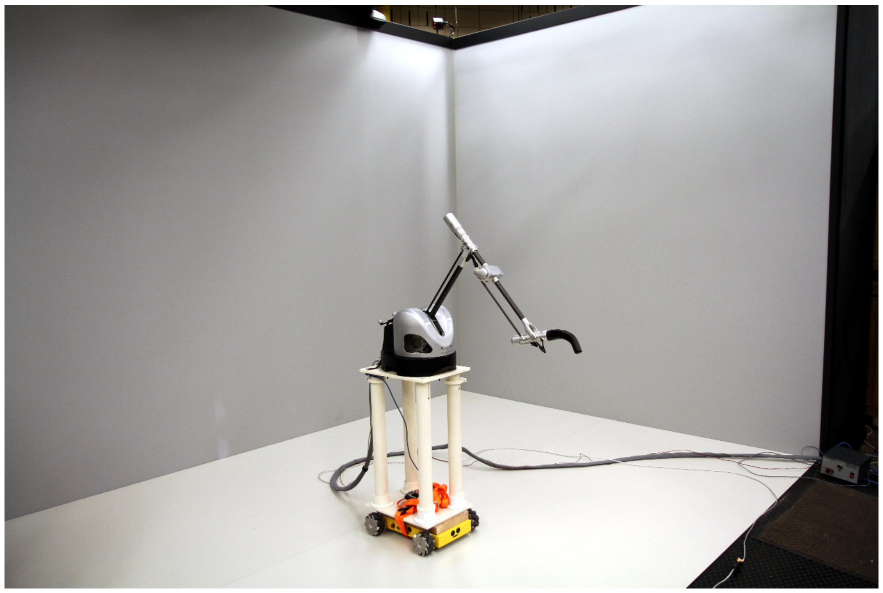


Figure 5.1: Haption Virtuose on Mobile Robot in METaL Virtual Reality System

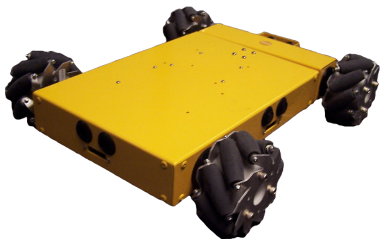


Figure 5.2: Omni-directional Mobile Robot with Mecanum-style Drive

The robot kit also includes a custom Arduino™-compatible Atmel AVR ATmega328p-based microcontroller unit (MCU) development board and input/output expansion board. The Nexus development boards stray from the design of the Arduino Duemilanove board primarily by their integration of four L298-based motor driver channels. Each motor driver channel takes as input one pulse-width modulation (PWM) pin on the MCU, as well as an additional digital output pin to indicate direction/sign, and outputs $[-12, 12]$ VDC to the connected motor. Each motor's encoder provides an A and B phase output, which are connected to digital input pins on the MCU so that the embedded software may monitor the behavior of the motors. The hardware serial interface (UART) provided by the MCU is both directly accessible via TTL-level pin-outs as well as usable through an integrated USB-serial converter chip. Either UART access allows for communication between the MCU and a computer, and the USB interface also allows in-circuit programming using an Arduino-compatible bootloader and the `avrdude` programming software.

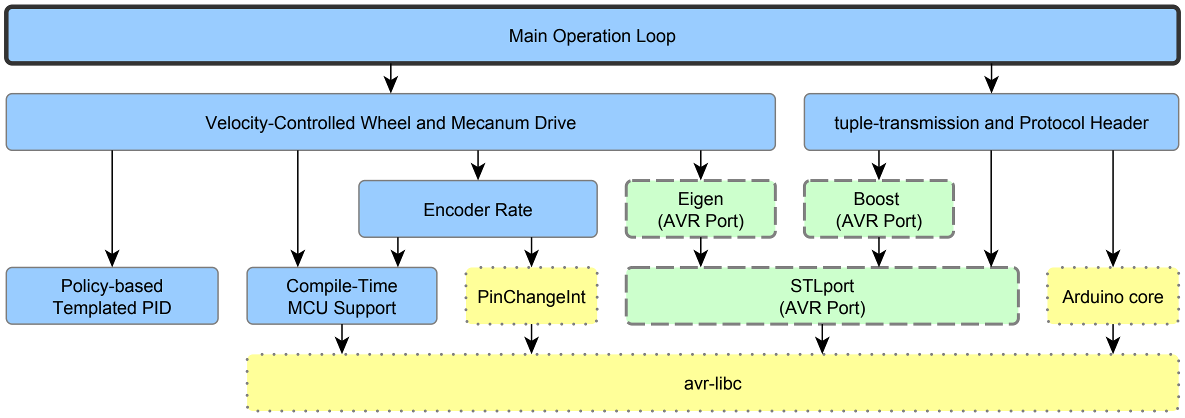


Figure 5.3: Embedded Software Layer Diagram

5.3.2. Software

Three separate software applications are involved in this integrated solution. Embedded software running on the mobile robot provides velocity control of the robot's four wheels, as well as a translation and rotation interface to a connected computer along with performance data. On a computer connected over a serial channel to the robot (either via USB or over Bluetooth RFCOMM), a "robot server" runs and adapts the serial communications with the robot into a network-transparent collection of VRPN analog and analog output servers. Finally, the SPARTA virtual assembly software, extended with adaptations to handle a device with a moving base and to output error measurements to the VRPN robot server, permits the use of the overall system as effectively one very-large-workspace haptic device capable of taking advantage of the full range of virtual assembly and interaction features.

The following sections describe and build on software layer diagrams shown in Figures 5.3 and 5.4. These diagrams share a common notation for distinguishing novel components from incremental improvements to existing software and stock software components used as-is. In both these figures, nodes represent software components or libraries, and edges show their dependencies in lower layers. Nodes with a solid border (colored blue) represent new software developed in this work, as an advancement of the state of the art. Nodes with a dashed border (colored green) are third-party libraries that were modified, typically to enable use in embedded processors, in this work and also represent advancements. Nodes with a dotted border (colored yellow) are stock third-party libraries used relatively as-is, and are shown to describe the base upon which this work builds.

Mobile Robot Software

A custom embedded software stack was developed for the Arduino-compatible MCU in the mobile robot. Initially, the Arduino integrated development environment (IDE) was used for development and testing, later being mostly supplanted by the arduino-mk Makefile system. The software was written in C++ using a toolchain consisting of `avr-gcc` 4.7, `avr-libc` 1.8.0, and `binutils-avr` 2.20.1. In the interest of developing a modular, maintainable codebase [11], the MCU software consists of a small main application calling into a number of libraries. Figure 5.3 is a layer diagram showing the libraries used and their dependencies, with the appearance of each node in the diagram holding the significance discussed earlier. In particular, in this diagram, the top node, with the thick border, represents the main loop of the embedded software.

A discussion of these components in more detail is warranted to describe the role they play and the contributions involved. A subset of the Arduino 1.0.2 core library, modified for GCC 4.7 compatibility, was used for some basic functionality. Registration and dispatch of pin-change interrupt handlers, used to monitor the encoders, was performed using the PinChangeInt 1.7[3] Arduino library developed by Lex Talionis and Michael Schwager. Three open-source third-party libraries were modified to support use on AVR microcontrollers during the course of this work. The STLport C++ standard library implementation was ported to AVR, and the modified version is publicly available[4] with instructions for use in the Arduino environment. Some AVR-specific porting and Arduino convenience modifications were also made to the header-only portions of the Boost C++ libraries version 1.51.0, released[5], and used in the embedded software. Additionally, the Eigen C++ template library for vector and matrix math was modified to avoid name collisions with `avr-libc` and remove assumptions about type sizes that do not hold when compiling for a 16-bit address space, and made available along with corresponding Arduino convenience headers[6]. These libraries form the base upon which the higher-level functionality for robot control was built.

In the interest of modularity, software written from scratch for use in this project was also divided up into logical libraries. The “tuple-transmission” library uses typelists and C++ template metaprogramming techniques to generate efficient code for serializing and de-serializing finite, known collections of messages. Template metaprogramming allows substantial portions of the code to be specialized and optimized automatically at compile

[3] http://arduino-pinchangeint.googlecode.com
[4] https://github.com/vancegroup/stlport-avr
[5] https://github.com/vancegroup/arduino-boost
[6] https://github.com/vancegroup/EigenArduino

time, rather than invoked with branches incurring performance costs at run-time. The tuple-transmission library, by design, is used on both the MCU and the computer to permit two-way communication. The full protocol used is defined in a single header file, designed for use in identical form on both ends of communication, allowing more efficient code and minimal overhead by not requiring message contents to be self-describing. The tuple-transmission library has been open sourced under the Boost Software License and published online[7], and the protocol header used is also publicly available[8].

Some additional libraries were written to encapsulate the details of elements of the system. A custom C++ microcontroller support library provides, among other features, a virtual "signed analog output" port for motor control, where a signed output value is turned into a PWM duty cycle and a direction bit on a pair of physical output pins. A template-based, layered proportional-integral-derivative (PID) controller library is used for velocity PI control of each individual wheel. A library for reporting rates based on quadrature encoder input, built using policy-based design principles [1], handles pin-change interrupts for all four quadrature encoders and computes instantaneous motor rotational velocity. This library successfully handles the $7000\,RPM \times 12\,CPR \times 4$ transitions per click $\times\, 4$ wheels = 1344000 interrupts per minute, or 22400 interrupts per second, generated during maximum recommended velocity robot motion. Finally, a library encapsulating velocity-controlled individual wheel drive and, from that, four-wheel Mecanum drive, provides the highest level interface to mobile robot motion to the embedded software. This permits not only the "production firmware," but also a number of verification and testing applications, to be built with concise, expressive high-level code.

Robot Server Software

The robot server is the simulation computer's counterpart to the embedded software. It communicates using the same tuple-transmission library and protocol headers as the embedded software, and serves to interface the established serial protocol of the robot with network-transparent VRPN (Virtual Reality Peripheral Network) [17] server devices. Figure 5.4 shows the layers of software libraries involved in the robot server, using the same conventions as Fig. 5.3 to indicate component origin. Two library components appear here that did not appear earlier. The first, TCLAP, is an open-source command line option parsing/handling library[9]. Internally, the server application is actually a front-end to an internal "vrpn-error-server" library providing a common subset of robot server features.

7 https://github.com/vancegroup/tuple-transmission
8 https://github.com/vancegroup/NexusRobotProtocol
9 http://tclap.sourceforge.net/

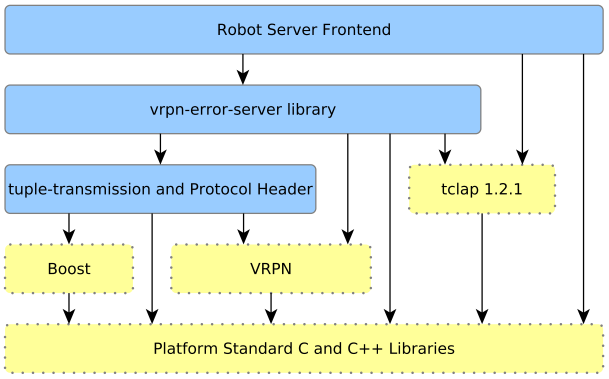

Robot Server Frontend
vrpn-error-server library
tuple-transmission and Protocol Header
tclap 1.2.1
Boost
VRPN
Platform Standard C and C++ Libraries

Figure 5.4: Robot Server Software Layer Digram

The front-end application used with the haptic device system provides control of the robot by a `vrpn_Analog_Output` server set up to receive two-dimensional, floating-point error vectors, scale them by a command-line-configurable proportional gain, then send them to the mobile robot as signed integer velocity commands. As with the embedded software, the creation of a library layer with a thin application layer over it permits additional applications for testing and verification to be easily built. The entire package, which has been built and tested on both Linux and Windows, is available as open-source software[10].

SPARTA Simulation Software

The final piece of the integration puzzle is the actual virtual assembly simulation software. In this work, SPARTA, the *Scriptable Platform for Advanced Research and Teaching in Assembly* [12], was extended with two modules to enable the use of the haptic device on the mobile robot. SPARTA, as the successor to the SHARP family of applications [16], provides a virtual environment for physically-based, haptic interaction with computer-aided design (CAD) models. Its physics simulation and range of haptic device drivers are written in object-oriented C++, while the graphics, audio feedback, and high-level interactions are written in Lua, taking advantage of the VR JuggLua framework [13] based on OpenSceneGraph and VR Juggler. Haptic device configuration, as well as virtual assembly scenario creation, is done with a domain-specific language built within Lua.

A particularly relevant aspect of SPARTA's design is the generic interaction device interface defined and used by the simulation. As a first step, this permits different types of input devices to present a uniform API, including haptic devices from different vendors, devices without haptic feedback, and so on. Adding a device driver to SPARTA by implementing this interface for a given device makes it immediately usable with all applicable function-

[10] https://github.com/vancegroup/vrpn-error-server

ality of the software: the design of the simulation is very loosely coupled to particular interaction devices. Furthermore, the generic device interface can also be implemented by “virtual” or “filter” devices: software objects that behave like input devices, but do not correspond directly to physical hardware, instead wrapping another input device and observing or modifying the data flow. It follows that supporting interaction with a haptic device on a mobile robot in SPARTA can be reduced to including a driver for the haptic device and producing one or more filter devices to interact with the mobile robot and account for its effects on the data reported by the haptic device. SPARTA already included support for the Haption Virtuose, as well as the Phantom Omni used for early testing, before this research began. The remaining additions to SPARTA are neatly split into two filter devices. The first, called `TrackedTransform`, is not strictly limited to use with a powered mobile robot base. Its function is to appropriately transform data both going to and coming from the simulation, based on the position of a tracker target assumed to be fixed to the base of its contained device.

The remaining task is to control the movement of the mobile robot based on data from the haptic device. In this research, based on the control scheme investigated by Garlington [4], the error (vector) between the position of the haptic device’s end effector and a predefined neutral position (roughly centered in the physical workspace of the haptic device) drives the robot. The control seeks to minimize that error and move the base so the end effector is neutrally located at all times. The SPARTA filter device for driving the robot is configured by providing the contained device and its corresponding neutral end effector position, as well as the device name of a `vrpn_Analog_Output` device that should receive the error vector. `VRPNMobileBase` does not modify any of the data flows, but on each position update it computes an updated error, connects to the robot server (which may be running on the same computer), which applies a gain to the error and passes it on to the mobile robot embedded software as a goal velocity.

5.4. Results

The full system as described successfully provides haptic interaction within a large workspace by having a mobile robot drive a ground-based haptic device around within the tracked area. Figure 5.5 shows the system in use, with the same hardware as pictured in Fig. 5.1. Before this image was captured, the user grasped the white pin (currently seen on the right side of the image) with the purple cursor (representing the end effector location in the simulation) and removed it from the rest of the assembly (green object and blue object). In the photographed moment, the user is now physically walking across the space while grasping the pin in order to set it down on the other side of the room, and the mobile robot is moving the base of the haptic device to keep up and allow this single movement to span a wider area than could be reached with the haptic device alone. Haptic feedback of colliding and sliding parts was felt during the removal of the pin, and now the simulated mass of the pin is all that is felt during movement of the haptic device.

The modularity and loose coupling described in the preceding sections produced a highly robust system: any one or more of the SPARTA simulation, robot server, VR system tracker, or mobile robot motor power can be shut down and started up again without bringing down any other part of the system. The ability to restart the robot server application was particularly useful during tuning of the proportional gain used to compute velocity from end effector positional error, since the gain was specified as a command-line argument to the robot server. Stopping it and starting it again with a different value while leaving the rest of the system operational and ready to use supported a very short test cycle time.

Following implementation of all components, the gain was found interactively by increasing the gain incrementally until the experience of moving a grasped object in space began to produce undesired haptic artifacts. A very high gain was able to be used, resulting in the mobile robot quickly "following" the movement of the end effector and keeping up with movement throughout the space. Free space movement in the entire area is possible and the user feeling is subjectively light and transparent. With the Haption Virtuose 6D35-45 device, capable of 35 N peak force and 10 N continuous, mounted on the mobile robot, simulated collision forces are stiff and crisp, and subjectively comparable to the experience of using the device on a fixed base. As intended, the full tracked area in the virtual reality system was made usable for haptic interaction with this work, allowing direct interaction with a larger, fuller simulation than with the ground-based haptic device alone.

5.5. Conclusions and Future Work

The full system is operational and suitable for further use, analysis, and development. Because the design of the SPARTA software allowed the necessary parts of the solution to be interposed essentially invisibly between the haptic device hardware and the simulation, the "haptic-device-on-powered-mobile-robot-base" can be used with all the scenarios and interactions built on SPARTA already; Fig. 5.5 is in fact one such example.

There are many opportunities for future work. One area of improvement involves implementing a method to allow the user to operate in a virtual world that is larger than the physical workspace of the optical tracking system. We are currently exploring various

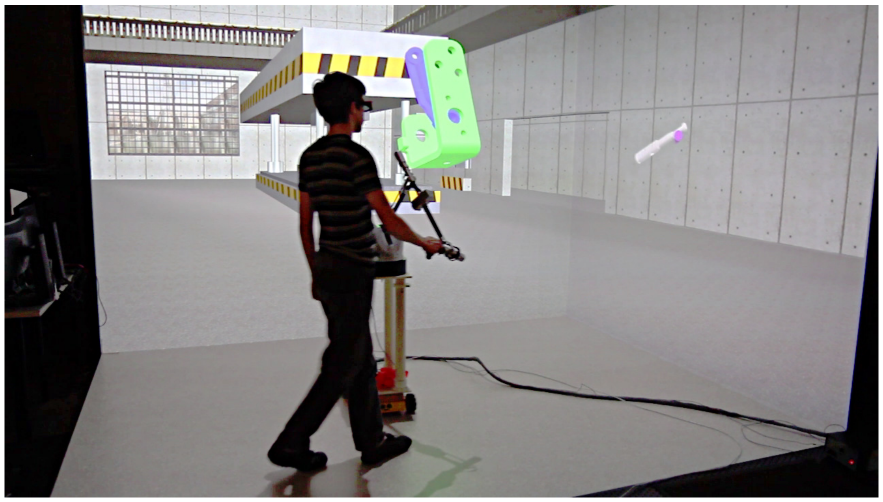


Figure 5.5: System in use simulating disassembly in METaL virtual reality system with Haption Virtuose

methods of navigating within a large virtual environment in which the user has haptic capabilities. Navigation in this sense refers to interacting with environments larger than the physically tracked area in a given VR system by moving the physical room around in the virtual space. Combining navigation and this system would allow, for example, factory or large assembly line walk-throughs, while still retaining the ability to interact haptically in the full physical workspace due to the mobile base.

Another area of future work involves exploring how to limit the mobile robot to avoid collisions in the physical space. In a virtual reality system of any sort, there are always physical limits, whether they are projection-screen walls or walls of an area with wide-area tracking and head-mounted displays. Physical object avoidance by the mobile robot needs to be implemented. Possibilities include use of on-board sensors on the mobile robot to detect limits and modify the velocity commands received accordingly, as well as computer-side approaches using the tracked position of the base to modify velocity commands before they are sent. There are presently no specific safeties, other than operator interaction, integrated into this system to prevent the mobile robot from contacting the projection screens or the walls in the space.

There also exist conditions in the projection-screen environment where the haptic device occludes the user's view of the virtual environments. This is only an issue in projection-screen based systems, as full head mounted display systems do not project the real environment onto the user's view. One approach would be to place the robot behind the user but this would create other issues, such as needing a larger floor area than the projected floor area. We will continue to explore the possibility of this configuration.

Cable management for the haptic device could also be improved. Presently no explicit cable management beyond bundling/looming and the inherent stiffness of the cables is being performed; however, the stiffness of the cable effectively kept it away from the mobile robot wheels. Future work will include exploring cable management techniques.

Additional work is needed on the control scheme to counteract reaction torques at the interface between the floor and the mobile robot. In the current setup, the system assumes that the mobile robot begins and moves aligned with the axes established for the room. This is generally true, however, as noticed particularly during extended testing, a powerful or extended simulated collision with the haptic device's arm extended produces a torque about the axis between the mobile robot and the haptic device's base. This can result in some rotational slippage of the mobile base so that it is no longer aligned with the room axes, which violates assumptions presently made at some layers of the system. Work on a rotational control scheme to keep the robot aligned with the axes could improve the experience. Applying rotational control to the mobile robot based on the rotation of the end effector, the behavior of the user, or both, could improve usability by keeping the haptic device base both appropriately located and conveniently oriented. This would also require appropriately handling rotation of the base at all layers in the combined configuration, revisiting assumptions.

Acknowledgment

The authors would like to thank Mylene Campana (ENS Cachan Antenne de Bretagne, France) and Adam Carlson for their work in developing familiarity with the Mecanum-style drive, and to Luke Swanson and Maria Blanek for the physical mount design and load testing. This research was performed at the Virtual Reality Applications Center at Iowa State University. This material is based upon work supported by the National Science Foundation under grant #CMMI-1061458.

Bibliography

[1] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0-201-70431-5.

[2] Federico Barbagli, A Formaglio, M Franzini, A Giannitrapani, and D Prattichizzo. An Experimental Study of the Limitations of Mobile Haptic Interfaces. In Marcelo H. Ang Jr. and Oussama Khatib, editors, *Experimental Robotics IX*, volume 21 of *Springer*

Tracts in Advanced Robotics, pages 533–542. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-28816-9. doi:10.1007/11552246_51. URL http://dx.doi.org/10.1007/11552246_51.

[3] Diego Borro, Joan Savall, Aiert Amundarain, Jorge Juan Gil, Alejandro Garcia-Alonso, and Luis Matey. A large haptic device for aircraft engine maintainability. *IEEE Computer Graphics and Applications*, 24(6):70–74, November 2004. ISSN 0272-1716. doi:10.1109/MCG.2004.45. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1355895.

[4] Isaac Garlington. *Expanding the usable workspace of a haptic device by placing it on a moving base*. Masters thesis, Iowa State University, 2012. URL http://lib.dr.iastate.edu/etd/12715/.

[5] Florian Gosselin, Claude Andriot, Florian Bergez, and Xavier Merlhiot. Widening 6-DOF haptic devices workspace with an additional degree of freedom. In *Second Joint EuroHaptics Conference and Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems (WHC'07)*, pages 452–457. IEEE, March 2007. ISBN 0-7695-2738-8. doi:10.1109/WHC.2007.127. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4145216.

[6] Florian Gosselin, Claude Andriot, Joan Savall, and Javier Mart. Large Workspace Haptic Devices for Human-Scale Interaction: A Survey. In *EuroHaptics 2008*, pages 523–528, Madrid, Spain, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-69057-3_66. URL http://link.springer.com/chapter/10.1007/978-3-540-69057-3_66.

[7] Haption S.A. Haption Inca 6D datasheet. Technical report, 2007. URL http://www.haption.com/site/pdf/Datasheet_Inca.pdf.

[8] Haption S.A. Scale 1 Haption Motorized Boom datasheet. Technical report, 2010. URL http://www.haption.com/site/pdf/Datasheet_Scale1.pdf.

[9] Yukihiro Hirata and Makoto Sato. 3-dimensional Interface Device For Virtual Work Space. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 2, pages 889–896. IEEE, July 1992. ISBN 0-7803-0738-0. doi:10.1109/IROS.1992.594498. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=594498.

[10] Norbert Nitzsche, Uwe D. Hanebeck, and Günther Schmidt. Design Issues of Mobile Haptic Interfaces. *Journal of Robotic Systems*, 20(9):549–556, September 2003. ISSN 0741-2223. doi:10.1002/rob.10105. URL http://doi.wiley.com/10.1002/rob.10105.

[11] David Lorge Parnas, Paul C. Clements, and David Mandel Weiss. The Modular Structure of Complex Systems. *IEEE Transactions on Software Engineering*, SE-11(3): 259–266, March 1985. ISSN 0098-5589. doi:10.1109/TSE.1985.232209. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1702002.

[12] Ryan A. Pavlik and Judy M. Vance. Expanding Haptic Workspace for Coupled-Object Manipulation. In *ASME 2011 World Conference on Innovative Virtual Reality (WINVR 2011)*, pages 293–299, Milan, Italy, 2011. ASME. ISBN 978-0-7918-4432-8. doi:10.1115/WINVR2011-5585. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1623675.

[13] Ryan A. Pavlik and Judy M. Vance. VR JuggLua: A Framework for VR Applications Combining Lua, OpenSceneGraph, and VR Juggler. In *Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS) in IEEE Virtual Reality*, Singapore, 2011. IEEE. doi:10.1109/SEARIS.2012.6231166. URL http://dx.doi.org/10.1109/SEARIS.2012.6231166.

[14] Angelika Peer, Yuta Komoguchi, and Martin Buss. Towards a Mobile Haptic Interface for bimanual manipulations. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 384–391. IEEE, October 2007. ISBN 978-1-4244-0911-2. doi:10.1109/IROS.2007.4399008. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4399008.

[15] Makoto Sato. Development of String-based Force Display: SPIDAR. In *8th International Conference on Virtual Systems and MultiMedia (VSMM 2002)*, Gyeongju, Korea, 2002.

[16] Abhishek Seth, Hai-Jun Su, and Judy M. Vance. Development of a Dual-Handed Haptic Assembly System: SHARP. *Journal of Computing and Information Science in Engineering*, 8(4):044502, 2008. ISSN 15309827. doi:10.1115/1.3006306. URL http://computingengineering.asmedigitalcollection.asme.org/article.aspx?articleid=1401373.

[17] Russell M. Taylor II, Thomas C. Hudson, Adam Seeger, Hans Weber, Jeffrey Juliano, and Aron T. Helser. VRPN: a device-independent, network-transparent VR peripheral

system. In *Proceedings of the ACM symposium on Virtual reality software and technology - VRST '01*, page 55, New York, New York, USA, 2001. ACM Press. ISBN 1581134274. doi:10.1145/505008.505019. URL `http://portal.acm.org/citation.cfm?doid=505008.505019`.

[18] Mark Ueberle, Nico Mock, and Martin Buss. VISHARD10, a novel hyper-redundant haptic interface. In *12th International Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems, 2004. HAPTICS '04.*, pages 58–65. Ieee, 2004. ISBN 0-7695-2112-6. doi:10.1109/HAPTIC.2004.1287178. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1287178`.

CHAPTER 6. MEMOIZATION AND REDUNDANCY IN MULTIPLE GEOMETRIC REPRESENTATIONS FOR CONSTRAINT RECOGNITION IN HAPTIC VIRTUAL ASSEMBLY

A paper in preparation for submission to ASME *Journal of Computing and Information Science in Engineering*

Ryan A. Pavlik, Leslie L. Miller

Abstract

Haptic virtual assembly applications allow users to interact with CAD models in a realistic way in a virtual environment, with haptic force feedback to complement the visual display. Haptic display requires performing collision detection and response at a very high rate, so methods that use tessellations of original exact CAD geometry predominate. We discuss a hybrid method combining tessellated geometry collision detection and response with precise geometry. Several steps of look-ups and computations are required to obtain actionable geometric constraint information automatically given the results of a tessellated geometry algorithm, traversing multiple redundant geometric representations. After discussing the algorithms, we present the "hash-consing" data structures and memoization that, in concert with facts about the input arising from multiple geometric representations and interactive collision detection, provide a much more efficient solution than an initial analysis would suggest. Implementation details are discussed, and the C++ "hash-consing" class template is made available.

6.1. Introduction

Virtual assembly is a process that involves manipulation of a computerized representation of a part in a virtual environment that allows some form of part–part interaction. Among other purposes, it can be used as a tool to develop manufacturing and assembly

Ryan A. Pavlik was the primary author and researcher.

processes, train workers, and evaluate a part, assembly, or product design for ability to be assembled, disassembled, or serviced. Haptic force-feedback in a virtual assembly simulation provides another sensory output besides visual display, conveying more information and making the simulation more similar to real life. It maintains the relationship between the control (user's hand grasping a device) and the display when collision detection and response are available, so the user can feel, not only see, when objects in the environment hit each other and are prevented from moving. However, simulating assembly with collision detection and response at haptic rates is difficult, particularly with low clearances between parts to be assembled. Haptic force-feedback requires simulation at a much higher rate (typically 1000 Hz) than purely graphical feedback (often 15 Hz to 120 Hz), and virtual assembly can involve complicated models that must be simulated with detail rather than as simplified primitive shapes.

6.1.1. Scope and Contributions

This paper presents a new explicit high-level decomposition of a "hybrid" collision detection/response algorithm incorporating fast discretized representations as well as precise b-reps into distinct components. Additionally, the theoretical complexity (time and space) and data structures are discussed. Due to the origin of the data involved, we were able to incorporate a number of optimizations, so performance differs greatly from the theoretical worst case performance. These optimizations are presented both in general and with reference to a C++ implementation.

6.2. Background

The context of this work assumes physically-based modeling of collisions between arbitrary three-dimensional models. These models are not necessarily convex, and correspond to parts of mechanical assemblies designed with CAD software. Models provide the shape information for the rigid bodies simulated in the physics engine. A penalty method of collision response is chosen. Collision detection detects a small degree of interpenetration between bodies, which is resolved by a penalty force to each of the bodies involved proportional to the depth of interpenetration and directed to eliminate it.

6.2.1. CAD modeling, tessellations, and approximations

Computer-aided design (CAD) software used to develop these parts represents them in mathematically-precise ways. Constructive solid geometry (Figure 6.1, left) is one such method of modeling. It involves combining basic solid forms (rectangular prisms, cylinders,

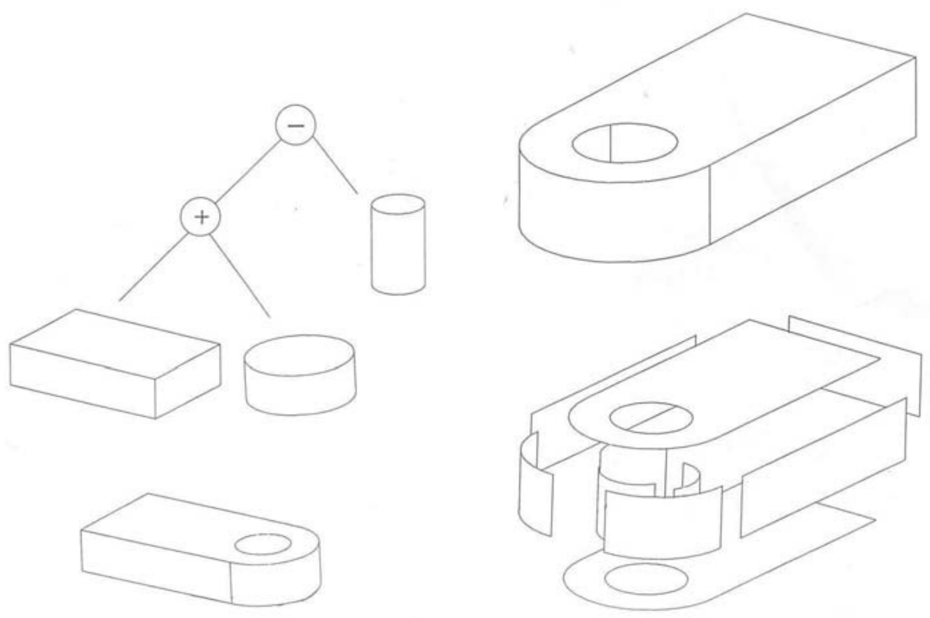


Figure 6.1: A computer-modeled object and its original precise definition (constructive solid geometry, left, or parametric boundary representation, right) [12]

etc.) by Boolean operations such as difference, intersection, and union. The method of CAD modeling considered in the context of the present work is solid or surface modeling with parametric boundary representations, or b-reps (Figure 6.1, right). Parametric b-reps are the mathematical representation of the bounding faces and edges defining an object. B-reps can vary in complexity, but many consist of commonly-understood elements such as spherical faces, planar faces, cylindrical faces, linear edges, and circular edges, of particular dimensions and bounds. On the simple models shown in Figure 6.2, some of these b-reps are pointed out. These common elements disclose how parts can interact with each other. For instance, a straight pin in a through-hole can rotate freely and move along its axis, which is a cylindrical constraint. Several similar geometric constraints arise in this way from the definition of part geometry.

Though CAD software maintains the definition of a part using precise b-reps, the model is often converted to other formats for use. For instance, display of 3D models is often optimized for meshes of triangles (tri-mesh or tessellated data). CAD programs may even internally tessellate a part for interactive preview during the modeling process, while maintaining the precise model as ground truth for modification. Interactive physics simulation with b-reps is also difficult because of poor performance. Thus, for interactive physical simulation, one of the model formats (often the tessellated model) is converted into a representation usable by a high-performance physics engine. In the broader research that comprises this work, a common physical model format uses voxels. Voxels are small cube-

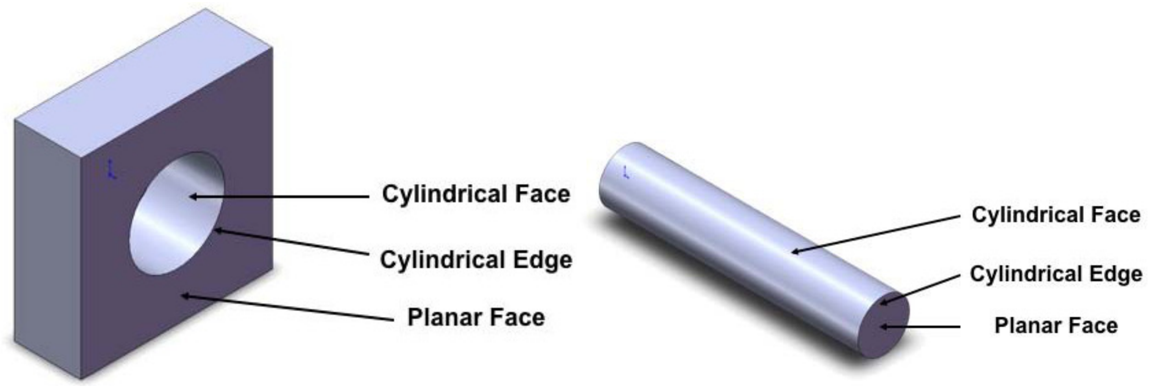


Figure 6.2: B-reps that can easily be inspected to gain insights during collision detection [5]

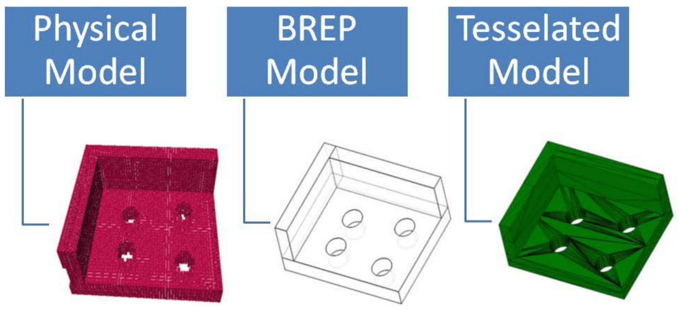


Figure 6.3: Multiple representations of a single object: left to right, in collision engine (voxels), original precise b-rep, and tessellated for graphics or physics [5]

shaped "volume elements" (analogous to 2D pixels—"picture elements") that represent the discretized volume of the model. In Figure 6.3, a b-rep model is shown with its boundaries in wire-frame. To the right is the same part tessellated into a tri-mesh. To the left is the voxelized version used in physics simulation. The holes are only truly circular in the precise b-rep: the tessellation turns circles into n-gons, and the voxels approximate the curve with cubes. This is one example of how the other model representations are approximations. These approximations may result in physical simulation arriving at results and behaviors that are not specified by the b-rep, particularly during fine manipulation tasks [14]. For instance, with small enough clearances between a circular pin and hole, neither approximation will permit free rotation, and will sometimes prohibit insertion due to collisions detected in the tessellated model that do not exist in the precise b-rep.

6.2.2. Virtual assembly

The specific tessellated geometry collision detection algorithm used in the implementation of this work is VPS (Voxmap PointShell) [8, 9], a software package licensed from Boeing and subsequently modified to support this approach. VPS represents bodies as

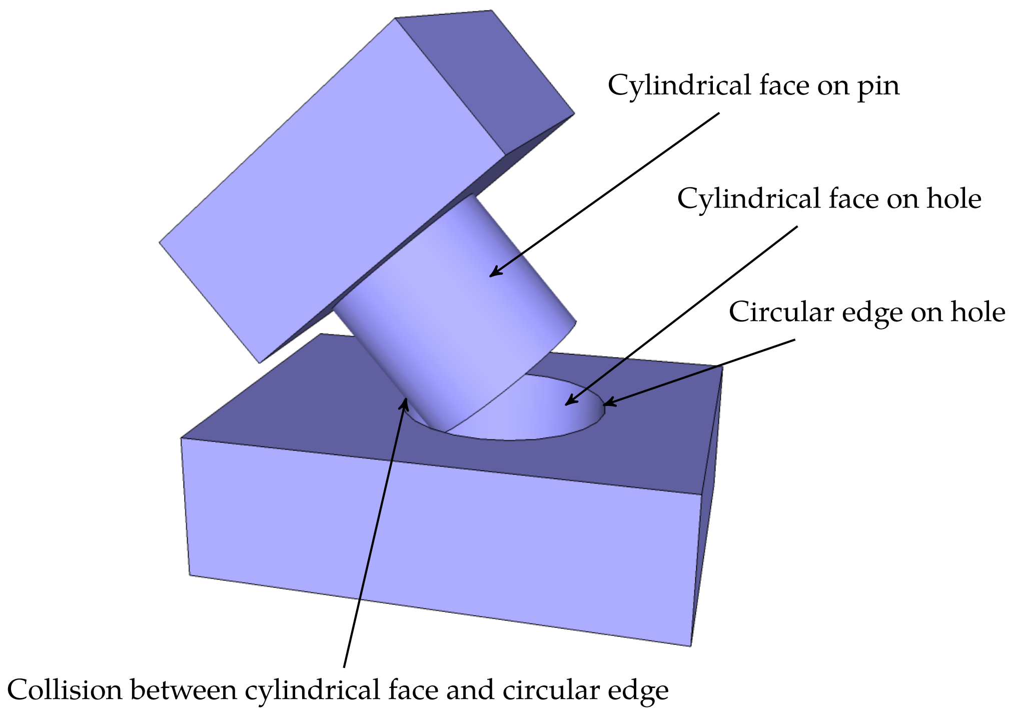


Figure 6.4: Collision between two objects and the associated b-rep information

a voxel map and a pointshell derived from center points of surface voxels. At the basic level, points in the pointshell of one body are transformed and tested against the voxel map of a possible colliding body to determine depth of penetration and associated response force. As implemented, the basic VPS algorithm is augmented with additional features such as distance maps, pre-collision braking forces, geometric awareness, and temporal consistency to ensure the preservation of a physically-based modeling environment without interpenetration or object tunneling, while maintaining a 1000 Hz operation rate to support haptic interaction.

The hybrid collision detection/response algorithm discussed in this work was initially presented in Faas [4]. It fuses information from the b-rep model with the results from the tessellated geometry to permit low-clearance virtual assembly. Based on the collisions detected between discrete collision units (for instance, voxels), it looks up the original b-reps, and automatically recognizes constraints to modify the force generation. For example, in Figure 6.4, several faces and edges are labeled. Collision between voxels on the left edge of the pin and the left edge of the hole would recall sets of b-reps that would include cylinders and circles, and a cylindrical constraint could thus be inferred.

Other approaches to low-clearance assembly include Tching et al. [13]. It provides for haptic “guide planes” corresponding to specific assembly situations, such as peg-in-hole. Upon collision of the corresponding part with the appropriate guide planes, a constraint is incrementally enabled and the tessellated geometry algorithm is locally disabled. However, models must be manually (interactively) augmented with these guide planes: it is not a fully automatic process.

Seth [10, 11] presents a method of performing virtual assembly entirely using precise b-reps. The proprietary D-Cubed geometry kernel was used to perform collision detection queries and compute collision responses. In early revisions, users manually specified constraints between colliding bodies through voice commands in the virtual environment. Further work automatically recognized constraints between colliding b-reps and enforced the corresponding constraint until the involved bodies were released by the user. However, this system could not sustain an update rate sufficient for haptic force-feedback.

6.3. Structure of the Hybrid Method

The method presented here builds upon the work of Faas et al. [4, 5]. We begin by first describing the whole hybrid method from an algorithmic point of view.

The overall hybrid method is split into two parts. The first part, shown in Algorithm 1, is offline, unsupervised preprocessing of the CAD model. It is automatic and executes when a model is initially loaded in order to set up data structures for the second part. The second part, shown in Algorithm 2, is the "run-time" algorithm. It executes within the tight loop of the physics engine, under intense performance constraints. When choosing among performance trade-offs, it is acceptable to allow slower tasks to occur down the offline part in order to obtain better performance for the run-time part. The algorithms as written here refer to voxels, but can be generalized to other discrete collision units in other physics engines, as long as it is possible to associate a number with each such unit and receive those numbers along with the collision detection results.

```
Input: A model with both b-rep and tessellated (triangular mesh, or tri-mesh)
       representations available
begin
    breps ← getBoundaryRepresentations(model)
    trimesh ← getTrimesh(model)
    voxels ← voxelizeTrimesh(trimesh)
    foreach voxel ∈ voxels such that voxel is a surface voxel do
        subsetbreps ← computeBoundaryRepresentationSetForVoxel(breps, voxel)
        storeBrepsForVoxel(subsetbreps, voxel)
    end
end
```

Algorithm 1: Model loading procedure for a single model

First consider Algorithm 1. It starts with simple data access. Line 4 contains a call, `voxelizeTrimesh`, to the tessellated geometry physics engine to generate its model representation and return its elements. In line 5 a loop over each surface voxel in the body begins. The purpose of this loop is to establish the association between voxels and b-reps, or more specifically, between each (surface) voxel and a subset of the set of b-reps in that body. There are two distinct parts inside the loop. The software computes the subset of b-reps for a voxel (`computeBoundaryRepresentationSetForVoxel` in line 6), which is considered to be very time-consuming. These expensive results are then stored (`storeBrepsForVoxel` in line 7) in a way that enables high-speed look-up during run-time. There are a number of possible implementations of `computeBoundaryRepresentationSetForVoxel`, such as presented in [5], and implementation of this method is out of scope of this paper. The storage of the b-rep subsets, however, is discussed further in this work, and is the first place where redundancy inherent in geometric data can be applied to provide optimized performance.

```
1  foreach pair of bodies (body_1, body_2) do
2      collidingVoxelPairs ← getCollidingVoxelPairs(body_1, body_2)
3      foreach pair of colliding voxels (voxel_1 ∈ body_1, voxel_2 ∈ body_2) ∈ collidingVoxelPairs
       do
4          Compute penalty force based on interpenetration.
5          subsetbreps_1 ← lookupBrepsForVoxel(voxel_1)
6          subsetbreps_2 ← lookupBrepsForVoxel(voxel_2)
7          getConstraintFromBrepSets (subsetbreps_1, subsetbreps_2)
8          Interpret the constraint returned, if any, to modify penalty force.
9          Compute corresponding torques from this voxel pair.
10         Accumulate forces and torques.
11     end
12 end
```

Algorithm 2: Run-time procedure executed every collision detection frame, which take place at a rate of 1000 Hz

While the first algorithm focused on a single model and thus a single simulated body at a time, Algorithm 2 considers pairs at a time (line 1). For an arbitrary n-body simulation, where any body may interact with any other body, this is the required general format. In fact, much of the body of this loop directly corresponds to the underlying penalty-based tessellated geometry collision detection and response simulation. Line 2 is a call to that algorithm's collision detection routine. Line 3 begins a loop iterating through the results of collision detection, computing the penalty forces and associated torques for collision response. Lines 5 to 8 are the portions unique to this hybrid collision simulation algorithm.

The calls to `lookupBrepsForVoxel` in lines 5, and 6 are the counterparts to the call to `storeBrepsForVoxel` in Algorithm 1, line 7. They perform the first step of bringing in more precise information to the simulation, by using voxels to determine colliding b-reps. The two subsets returned are subsets of the b-reps of their respective bodies. Line 7 calls the function `getConstraintFromBrepSets`, shown in Algorithm 3 and discussed in more detail below. Following this call, the b-reps retrieved have been mapped to an actionable geometric constraint when possible. The `getConstraintFromBrepSets` call may return a null geometric constraint, in which case the response of the overall algorithm for that voxel is the tessellated geometry algorithm's response unaltered. The subsequent step at line 8 is phrased generally because there are a number of possible algorithms to incorporate the geometric constraint data into the initially-computed collision response.

```
Input: b-rep subsets $\mathsf{subsetbreps}_1, \mathsf{subsetbreps}_2$
Output: A constraint, possibly null.
1 begin
2     foreach pair of b-reps $(\mathsf{brep}_1, \mathsf{brep}_2) \in \mathsf{subsetbreps}_1 \times \mathsf{subsetbreps}_2$ do
3         Detect constraint, if any, between $\mathsf{brep}_1$ and $\mathsf{brep}_2$
4     end
5     Reduce set of detected constraints to a single constraint.
6     return result of reduction
7 end
```

Algorithm 3: getConstraintFromBrepSets

The function `getConstraintFromBrepSets` (Algorithm 3) performs constraint recognition between sets of b-reps using a method that performs constraint recognition between individual b-reps. Line 2 begins iteration over the Cartesian product of the two input sets, to determine if any pair of b-reps involved between these two voxels results in a constraint. Line 3 involves a call to an automatic constraint recognition engine. Implementation of such an engine is out of scope for this paper, though it will be considered to be a process of non-negligible but constant time complexity. One such implementation is presented in Chapter 7. The loop implements a map operation, which is followed by a reduction to at most a single geometric constraint (line 5).

6.3.1. Considering the task in terms of sets and functions

More formally, let R_i be the set of all boundary representation entities (Algorithm 1, line 2) in some body B_i, and let V_i be the set of voxels (Algorithm 1, line 4) in the same body. Then, we define the family of functions

$$r_i : V_i \to \mathbb{P}(R_i) \tag{6.1}$$

that map each voxel in a body to the subset of b-reps that correspond to it in the original model. The function for a particular body is initially computed by Algorithm 1, line 6. The range of this function for a given body is the power set of the body's b-reps, so potentially $2^{|R_i|}$ distinct subsets of b-reps might be associated with voxels of the body.

At run-time during the simulation, consider a collision detected between bodies B_i and B_j (Algorithm 2, line 1). The collision detection engine returns a set of pairs of voxels,

$$H = \{(v_i, v_j) \mid v_i \in V_i, v_j \in V_j, v_i \text{ and } v_j \text{ are colliding}\} \tag{6.2}$$

which can be called the collision pairs (Algorithm 2, line 2).

We can formalize the interface of an automatic geometric constraint recognition engine, as applied in line 3. It defines C as the set of recognizable constraints, including the null/unrecognized constraint $\emptyset$. On execution it applies the function

$$g : \{(x, y) \mid x \in R_i, y \in R_j\} \to C \tag{6.3}$$

determining if the interaction of two boundary representation entities from distinct bodies results in a geometric constraint.

Each element (v_i, v_j) of a collision set is a pair of voxels that are interacting. Each such element implies a potential interaction between a b-rep subset $r_i(v_i)$ and a b-rep subset $r_j(v_j)$. The constraint recognition engine directly considers only pairwise interactions between individual b-reps, thus it must be invoked on all elements of $RP = \{(x, y) : x \in r_i(v_i), y \in r_j(v_j)\}$. This set RP is the Cartesian product seen in `getConstraintFromBrepSets` (Algorithm 3, line 2). Constraint recognition is applied as a "map" operation on these b-rep pairs. The first part of a trivial corresponding "reduce" operation can be described by defining the set of non-null detected constraints

$$DC = \{c \mid c \in g(RP), c \neq \emptyset\} \tag{6.4}$$

The trivial "reduce" operation currently applied returns a single constraint if and only if only one non-null constraint was recognized. Thus, forming a layer over the constraint

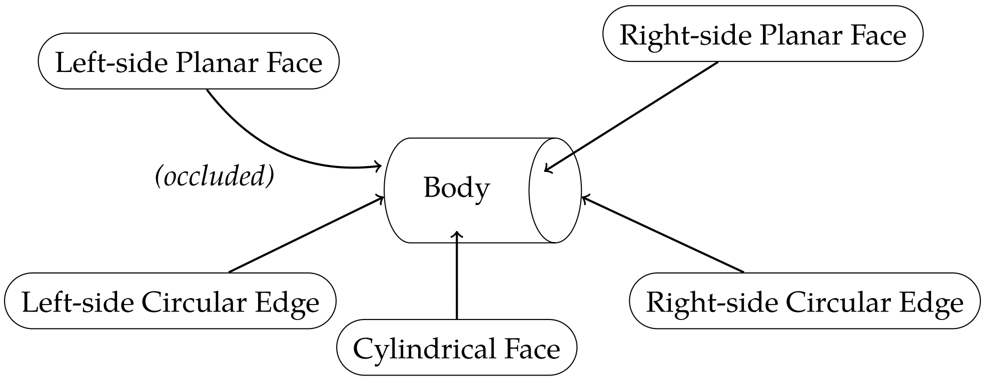


Figure 6.5: A cylinder and its b-reps, uniquely labeled

detection engine is the function comprising both map and reduce steps,

$$h(r_i(v_i), r_j(v_j)) = \begin{cases} c \in DC & |DC| = 1 \\ \emptyset & \text{otherwise} \end{cases} \tag{6.5}$$

We compute the run-time of `getConstraintFromBrepSets`, full constraint detection between sets of b-reps as follows. For simplicity, we will assume that the size of both $|R_i|, \left|R_j\right| \sim \mathcal{O}(n)$ for some n. Similarly, the b-rep subsets corresponding to a single pair of colliding voxels $|r_i(v_i)|, \left|r_j(v_j)\right| \sim \mathcal{O}(n)$. The constraint recognition engine is thus run on a set of size $|RP| \sim \mathcal{O}(n^2)$ (the Cartesian product in Algorithm 3).

6.4. Optimizations

The computational (time and memory) complexity of this problem appears to be high in theory, but facts about the context provide useful opportunities for optimization. For instance, the family of b-rep subsets defined by the range of r_i for a given body is likely to be much smaller than the theoretical $2^{|R_i|}$ because discretization resolution (voxel resolution in this case) is chosen to be high enough to capture the details of a model for rough collision detection. Typically, voxel size must be set to smaller than the smallest feature of a body involved in collision. If more than a small fraction of the b-reps of a body were encompassed by a given voxel, substantial detail would be lost in the discretization process and the voxel size would be considered too large for the model. Thus, each voxel maps to at most a few b-reps under r_i. For illustration, consider a cylindrical pin and its b-reps, as shown in Figure 6.5. A voxel on the edge of such a cylindrical pin, as shown in Figure 6.6 contains two faces and an edge. A voxel located at the corner of a cube is likely to be the worst common case, since it may refer to six b-reps: three edges and three faces. Furthermore, as also seen

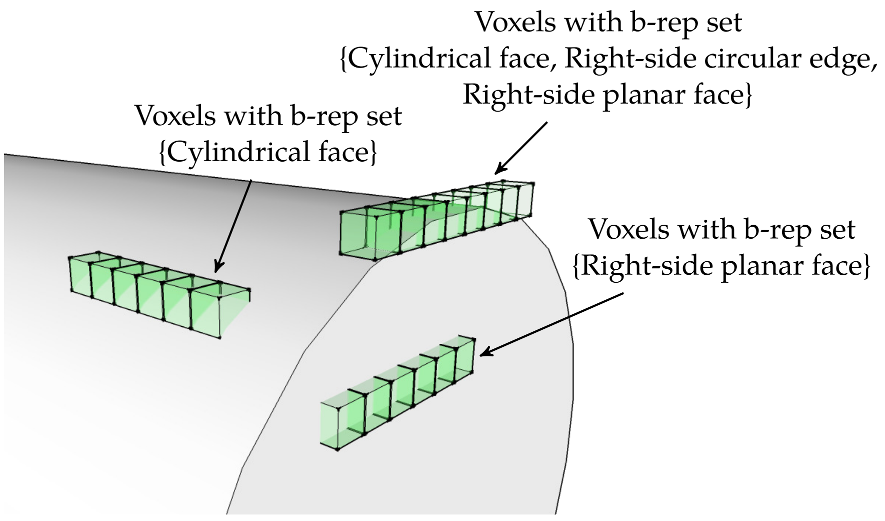


Figure 6.6: Some sample voxels from Figure 6.5 and their sets of associated b-reps

in Figure 6.6, many voxels will map to the same set of b-reps- all voxels along a given edge, all voxels on a face away from an edge, etc. Together this implies that the number of unique b-rep subsets in a body at a reasonable voxelization resolution is small: smaller than the number of voxels and much smaller than $2^{|R_i|}$. In the diagram, 20 voxels are shown, yet they share just 3 unique b-rep subsets. Though the voxels shown are just a fraction of all the surface voxels in the model, the full model would contain only 5 unique b-rep subsets. These 5 sets would comprise the 3 shown, plus the set containing the left-side face, and the left-side equivalent of the set (of size 3) associated with the edge voxels pictured.

This small number of unique b-rep subsets per body suggests that constraint recognition between any two given b-rep subsets may be a repeated computation. This is reinforced by temporal and spatial consistency. Voxels colliding this frame are likely to be colliding (or near to those colliding) next frame. Voxels colliding in a single frame are likely to be located near each other. (Remember, voxels near each other are likely to have the same b-rep subset.)

6.4.1. Data types

Space for a small integer in the voxel representation is all that is needed to store the index to the b-rep subset within each voxel representation. Taking advantage of the low number of unique sets in the range of r_i, a container named `CountedUniqueValues` was designed to store b-rep subsets for each body. The principle is to collapse the storage space needed by storing each unique value only once, and to assign a constant integer ID to each stored value allowing rapid recall later. This is an application of the "hash-

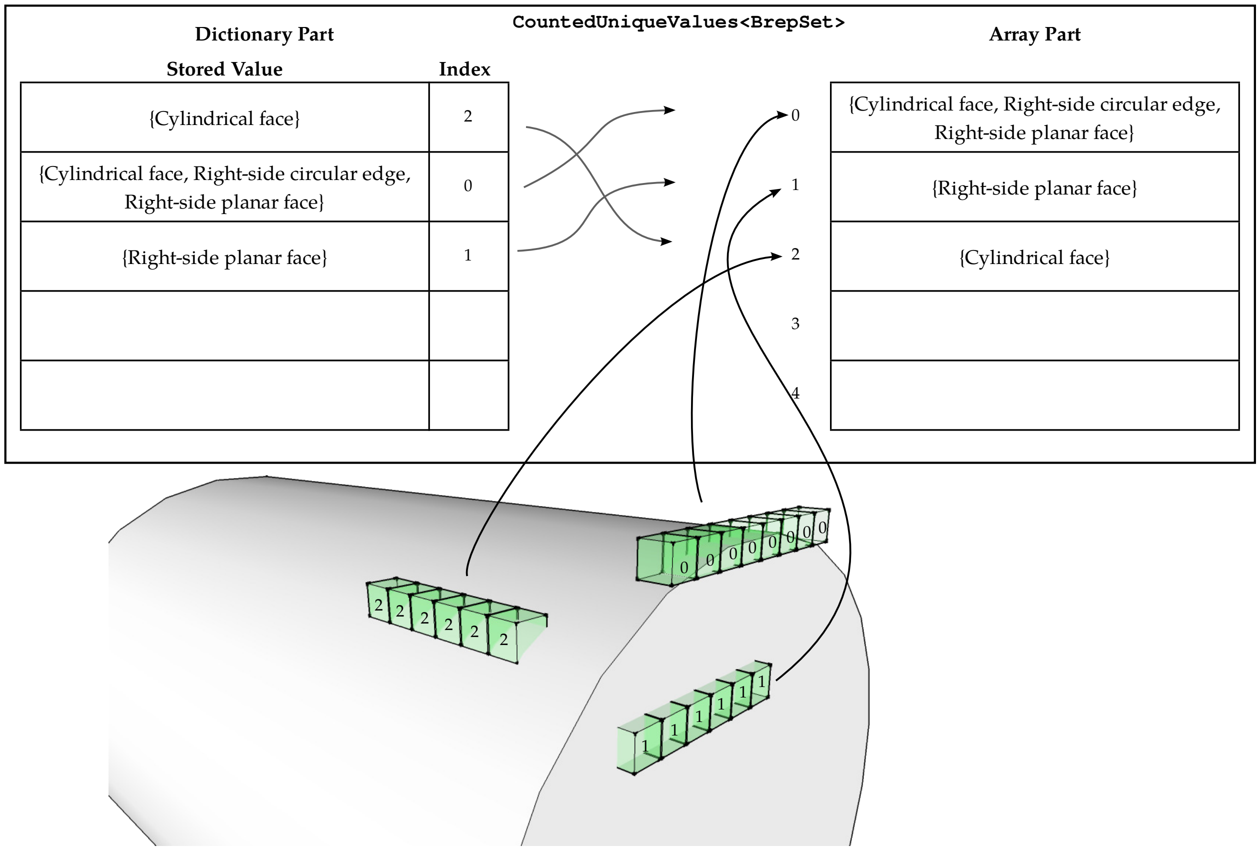


Figure 6.7: The `CountedUniqueValues` data structure with values from Figure 6.5

consing" technique [3, 6], most widely discussed in Lisp literature, which involves keeping only one copy of each unique value in an environment. While the subsets might not be very expensive to construct or particularly space-intensive, they may be too large to fit within space provided for expansion in the voxels themselves. Furthermore, merging identical values means that comparisons can be based on identity, rather than equality. The integer identity of each subset is fast to compare and easy to use as a part of a dictionary key, which will be important later. Note that for this application, storing a value is not performance-critical, since this only happens at startup, but retrieving a value must be very fast, since this happens in the inner loop. The design of `CountedUniqueValues` combines a dynamically-sized array of values and a dictionary mapping values to integers, as shown in Figure 6.7. A store operation (Algorithm 4) checks the dictionary first to see if the value has already been stored, simply returning the existing ID if it has been. If the value has not yet been stored, it is appended to the array and its index in the array is stored in the dictionary and returned. In both cases, an ID is returned that can be used to recall the value with just an array index (Algorithm 5). As the figure illustrates, this small integer ID can be directly associated with voxels, with the index identifying the unique subset of b-reps associated with a given voxel.

The run-time complexity of this compound data structure depends on the characteristics of its contained dictionary and array. Here, the term array refers to an indexed container with constant time retrieval and amortized constant time appends. This provides `CountedUniqueValues` with constant-time retrieval, as well, which suits our need for high performance in the simulation loop. By keeping track of the size of the array, all array operations in the structure are constant time or amortized constant time. Stores are then dominated by the selected dictionary type's find, get, and insert complexity.

```
Data: A CountedUniqueValues structure (a dictionary and an array)
Input: A value $x$ to store
Output: An integer ID to use as a retrieval key
1 if $dict[x]$ exists then
2     $i \leftarrow dict[x]$
3 else
4     $i \leftarrow$ size of $array$
5     append $x$ to the array
6     $dict[x] \leftarrow i$
7 end
8 return $i$
```

Algorithm 4: CountedUniqueValues::Store

```
    **Data**: A CountedUniqueValues structure (a dictionary and an array)
    **Input**: An integer ID $i$ (from the Store method)
    **Output**: A value
1   **return** *array[i]*
```

Algorithm 5: CountedUniqueValues::Retrieve

6.4.2. Applying memoization

Applying the `CountedUniqueValues` data structure collapses multiple objects representing the same set of b-reps into just a single object, referred to by a number. In addition to the storage benefits of using this structure, these numeric identifiers can be used as opaque, unique identities of the theoretical b-rep subsets suitable for equality comparison as well as usage as part of a dictionary key. Each body contains a `CountedUniqueValues` structure, storing its b-rep subsets. For each pair of bodies, a "collision link" object is maintained to compute collision detection and response between those bodies. During a collision event, the results of `getConstraintFromBrepSets` (modeled in equation (6.5) as the function h) are needed for each colliding voxel pair. The collision link contains a structure that implements a memoization layer for `getConstraintFromBrepSets`. It is provided with two b-rep subset identities from `CountedUniqueValues`, and before invoking `getConstraintFromBrepSets`, it looks up that pair of IDs in a dictionary structure containing cached, already-computed results. If the given pair has been computed previously, the constraint can be returned immediately without further computation. Otherwise, the constraint recognition is performed, and the results are added to the dictionary for future use before being returned. The fruitful combination of hash-consing and memoization has been noted in other applications, as well [1, 2].

We selected a dictionary without any cache-expiry logic to use as our memoization structure, since the inputs and outputs stored are small and the number of entries are low. To elaborate, recall that there are theoretically $2^{|R_i|}$ possible b-rep subsets in a body, so in a pair of bodies, there might be up to $2^{|R_i|+|R_j|}$ unique inputs to the function h. However, as discussed earlier, the geometry and discretization resolution limit the number of unique b-rep subsets actually found in a body. Furthermore, due to physical constraints of interaction, not all combinations of b-rep subsets between the two bodies can be the result of a collision. Thus, if n and m are the number of unique b-rep subsets actually associated with a collision element on two bodies, and p is the maximum number of unique pairs of b-rep subsets actually physically capable of interacting in the simulation, we know that

$$\begin{aligned} n &\ll 2^{|R_i|} \\ m &\ll 2^{|R_j|} \\ p &< n \cdot m \end{aligned}$$

The dictionary may be a hash table, with $O(1)$ look-up, or as slow as $O(\lg p)$ for look-ups in a binary search tree. We consider the constraint-detection code to be costly, dominating even the $O(\lg p)$ naive dictionary look-up cost. The advantage of hash-consing here as described for `CountedUniqueValues` is that during the time-sensitive collision operations, retrieval of values is a known $O(1)$ operation with a small constant (an array index). Memoization of the constraint-detection results provides increased performance since a given result is likely to be needed for multiple collision units during a single frame, again for physical reasons. Any additional computations with the constraint result that are based on the bodies' positions and orientations in space can also be cached for at least one frame. Additionally, due to temporal consistency, the same results are likely to be needed in the following frame, indicating the value of preserving the cache.

6.5. Implementation and Results

The hash-consing data structure `CountedUniqueValues` was implemented as a templated container class, parameterized by its stored value type.[1] As implemented, a C++ `std::map<value_type, size_t>` is used as the dictionary, with a `std::vector<value_type>` as the array. All the operations on `std::map` that are used in the `CountedUniqueValues` implementation are mandated by the C++ standard to be of logarithmic complexity in the size of the structure [7, cross-ref map.access]. The dictionary look-up dominates the cost of Store, resulting in $O(\lg n)$ run-time where n is the number of unique values stored. The run-time cost of Retrieve is $O(1)$, since this is an array indexing and implementation independent. In the C++ implementation developed, as a consequence of the selection of `std::map` as the dictionary type, it is important that the `value_type` have value (not reference/pointer) semantics and provide equality and comparison overloads for use by the dictionary.

In the proof of concept implementation of these techniques in the context of the hybrid collision detection/response algorithm, which uses VPS as the tessellated geometry collision detection method, some 12 bits of storage are made available in the opaque data

[1] Source code available at `https://github.com/vancegroup/util-headers`

structure per voxel. These 12 bits are used to store the small integer identifying the corresponding b-rep subset, as stored by a per-body `CountedUniqueValues` structure. An object is created and retained per pair of bodies (a `CollisionLink`) that is responsible for performing collision detection and response. This was a natural location to store the memoization data structure for constraint recognition. Here again, a `std::map` was used for a first implementation. It performs well enough in practice, primarily due to the small number of entries, even though a hash table would have better asymptotic performance.

As an additional level of computation reduction, we considered that constraint response computation might be expensive. As such, constraint detection results are stateful objects. Upon their retrieval, the transformation matrices (pose) of the associated bodies are submitted as input to a method. The method caches the two poses and computes constraint response outputs. If the input poses are the same as the cached poses, such as the case where multiple collision pairs in a frame result in the same constraint, no redundant recomputation is performed. This implicitly behaves as a memoizing function object, where constraint results and the corresponding response computations are the function objects.

In operation, debug statements were inserted at key points to determine the effectiveness of the combined hash-consing and memoization. Even in the first frame with collisions, cache hits for the memoized constraint recognition function greatly outnumbered cache misses (required computations). In immediately subsequent frames, essentially all constraint detection results had already been computed and could be returned from the memoization cache, with new operations occurring only when colliding geometry interacted in a novel way for the first time.

The performance of these techniques was dramatic: the full b-rep set look-up and constraint recognition process was only required in 4 out of $7,273,610$ collisions during a long sample trial with two bodies interacting (that is, 4 cache misses). In the remaining events, previously-calculated results could be used. Of the 4 unique computation events, 3 resulted in no constraint, while one resulted in exactly one constraint. Note that the count of $7,273,606$ cache hits is essentially unbounded as surfaces that had already interacted may freely interact again without requiring full computation.

6.6. Conclusion

The hybrid method of collision detection and response for haptic virtual assembly, initially presented in Faas [4], addresses the difficulties posed by low-clearance CAD assemblies. We presented a concise description of the algorithms involved in a method patterned after [4], as well as the theoretical complexities in the absence of optimizations that may have contributed to performance issues seen in that previous work. The limited nature of the input data, namely redundant geometric representations including fine discretizations, permitted the development of several layers of de-duplication of data and computation. Combining a data structure based on the principle of hash-consing to store each unique set of b-reps just a single time, with memoization to eliminate repeated computation of functions with identical input data, eliminated a substantial proportion of the computation required in the time-sensitive portion of the hybrid method. An implementation in C++ was developed, with the `CountedUniqueValues` hash-consing template data structure released publicly. Run-time verification indicated that the layers of optimizations performed as desired, dramatically reducing the total amount of computation required to perform collision detection and response combining an tessellated geometry method with precise b-rep and constraint knowledge.

Acknowledgment

This work was performed at the Virtual Reality Applications Center at Iowa State University. This material is based upon work supported by the National Science Foundation under Grant No. CMMI-0928774. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Bibliography

[1] Henry G. Baker. A tachy "TAK". *ACM SIGPLAN Lisp Pointers*, V(3):22–23, August 1992. ISSN 10453563. doi:10.1145/147135.147145. URL `http://portal.acm.org/citation.cfm?doid=147135.147145`.

[2] Robert S. Boyer and Warren A. Hunt. Function memoization and unique object representation for ACL2 functions. In *Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications - ACL2 '06*, page 81, New York, New York, USA, 2006. ACM Press. ISBN 0978849302. doi:10.1145/1217975.1217992. URL `http://portal.acm.org/citation.cfm?doid=1217975.1217992`.

[3] A. P. Ershov. On programming of arithmetic operations. *Communications of the ACM*, 1(8):3–6, August 1958. ISSN 00010782. doi:10.1145/368892.368907. URL `http://portal.acm.org/citation.cfm?doid=368892.368907`.

[4] Daniela Faas. A Hybrid Method for Haptic Feedback to Support Manual Virtual Product Assembly. In *ASME 2011 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, pages 1–8, Washington, DC, USA, 2011. ASME. doi:10.1115/DETC2011-47665. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1639771.

[5] Daniela Faas and Judy M. Vance. BREP Identification During Voxel-Based Collision Detection for Haptic Manual Assembly. In *ASME 2011 World Conference on Innovative Virtual Reality (WINVR 2011)*, pages 145–153, Milan, Italy, 2011. ASME. ISBN 978-0-7918-4432-8. doi:10.1115/WINVR2011-5524. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1623634.

[6] Eiichi Goto. Monocopy and associative algorithms in extended Lisp. Technical report, University of Tokyo, 1974.

[7] ISO JTC1/SC22/WG21. ISO/IEC 14882:2011 – Programming Languages – C++ (C++11 Final Draft N3242). Technical report, ISO, 2011. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf.

[8] William A. McNeely, Kevin D. Puterbaugh, and James J. Troy. Six degree-of-freedom haptic rendering using voxel sampling. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques - SIGGRAPH '99*, pages 401–408, New York, New York, USA, 1999. ACM Press. ISBN 0201485605. doi:10.1145/311535.311600. URL http://portal.acm.org/citation.cfm?doid=311535.311600.

[9] William A. McNeely, Kevin D. Puterbaugh, and James J. Troy. Voxel-based 6-dof haptic rendering improvements. *Haptics-e*, 3(7), 2006. URL https://haptics-e.org/Vol_03/he-v3n7.pdf.

[10] Abhishek Seth. *Combining physical constraints with geometric constraint-based modeling for virtual assembly*. Masters thesis, Iowa State University, Ann Arbor, 2007. URL http://search.proquest.com/docview/304857108?accountid=10906.

[11] Abhishek Seth, Judy M. Vance, and James H. Oliver. Combining Dynamic Modeling With Geometric Constraint Management to Support Low Clearance Virtual Manual Assembly. *Journal of Mechanical Design*, 132(8):081002, 2010. ISSN 10500472. doi:10.1115/1.4001565. URL http://mechanicaldesign.asmedigitalcollection.asme.org/article.aspx?articleid=1450105.

[12] Ian Stroud. *Boundary Representation Modelling Techniques*. Springer London, London, 2006. ISBN 978-1-84628-312-3. doi:10.1007/978-1-84628-616-2. URL `http://link.springer.com/10.1007/978-1-84628-616-2`.

[13] Loïc Tching, Georges Dumont, and Jérôme Perret. Interactive simulation of CAD models assemblies using virtual constraint guidance. *International Journal on Interactive Design and Manufacturing (IJIDeM)*, 4(2):95–102, April 2010. ISSN 1955-2513. doi:10.1007/s12008-010-0091-7. URL `http://www.springerlink.com/index/10.1007/s12008-010-0091-7`.

[14] Judy M. Vance and Georges Dumont. A Conceptual Framework to Support Natural Interaction for Virtual Assembly Tasks. In *ASME 2011 World Conference on Innovative Virtual Reality (WINVR 2011)*, pages 273–278, Milan, Italy, 2011. ASME. ISBN 978-0-7918-4432-8. doi:10.1115/WINVR2011-5570. URL `http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1623669`.

CHAPTER 7. AUTOMATIC GEOMETRIC CONSTRAINT RECOGNITION FOR HAPTIC ASSEMBLY APPLYING DISCRIMINATED UNIONS, MULTI-METHODS, AND TEMPLATE SPECIALIZATION

A paper in preparation for submission to ASME *Journal of Computers and Information Science in Engineering*

Ryan A. Pavlik, Judy M. Vance

Abstract

The research goal was develop a new software system for recognizing geometric constraints arising from the interaction of pairs of boundary representations, for use in a haptic virtual assembly collision detection and response algorithm. The problem was modeled as a multi-method operating on a pair of b-reps. Each used b-rep type was designed as a concrete class, then applied in a discriminated union type using the Boost.Variant library for generic handling. The core of the recognition engine consists of template specializations for pairs of b-rep types that lead to constraints. The applied C++ metaprogramming techniques automatically handle commutativity of arguments for constraint recognition, avoid explicit conditionals and case statements, and allow easy extension and maintenance. These are zero-overhead abstractions that are inlined and optimized away at compile time. The resulting constraint recognition engine performs well, with per-pair recognition performance scaling with the number of specializations included in the engine, which is smaller than the total number of cases considered due to commutativity. The modern C++ techniques and the overall software design resulted in an an automatic constraint recognition engine suitable for use in a high-performance physics simulation application.

Ryan A. Pavlik was the primary author and researcher.

7.1. Introduction

Geometric constraints are an important element of CAD assemblies. Many CAD systems generate or use them in the modeling process. Virtual assembly applications permit a user to interact with models originating in a CAD package, typically in a physically-based manner. Such interaction may take place to investigate or evaluate assembly and disassembly of the parts, or provide training for the same [22]. However, in virtual assembly applications, constraint data is often not available to be loaded from CAD-specific model formats, or alternately it may be used as the only source of part-part interaction simulation. When constraint-aware collision detection and response is used, it may not complete quickly enough to support the kind of interactive experience desired. When the user is given freedom of movement to manipulate parts, this allows for non-designed part-part interactions, whether mis-assembly or simple exploration. These interactions may nevertheless logically imply a constraint.

This work forms part of an effort to augment high-performance but lower-precision collision detection and response algorithms with constraint-based precision. As a part of the process, collision detection identifies parametric boundary representations (b-reps) from the original CAD models (e.g., cylindrical faces, circular edges, planar faces). Any constraints arising from the interaction of those b-reps may affect part-part interaction and might thus affect the collision response. This requires a constraint-recognition engine to automatically recognize constraints. This work presents a novel design and implementation of such an engine applying modern C++ programming techniques.

7.2. Previous Work

Previous virtual assembly applications have exploerd the use of constraints to aid assembly. Virtual assembly applications could be considered as primarily constraint-based or primarily physically-based. Early work relied on knowing the assembled position of parts and sub-assemblies within an assembly to provide a form of constraint interaction. They provided a "snap-to-fit" experience for the user [9, 21]. The goal in those works was to aid in the creation of assembly procedures, so the absence of constraint-enhanced interaction for mis-assembly was not a substantial weakness. However, snap-to-fit necessarily sacrifices some of the physical verisimilitude of the simulation by providing an unnatural environment behavior.

Other constraint-based interaction required and used constraint data from the CAD system combining snap-to and more free motion [29]. A rule-based constraint recognition engine is described in part in [19], incorporating thresholds for tracker error and notifying the user with opportunity to cancel before applying the effect of a detected constraint. Seth et al. [22, 23] investigated the use of geometric constraints in a more free-form virtual assembly environment. Initial work required the user to manually identify constraints in-simulation, while later work performed some automatic constraint recognition to enforce constraints during the duration of a user's grasp of an object. Most closely related to the present work, Faas et al. [10, 12, 11] investigated combining a physically-based virtual assembly simulation providing haptic force-feedback with automatic constraint recognition and solution. Providing haptic feedback poses a particular problem for virtual assembly development, as much higher simulation rates are required to provide a stable environment than with visuals only. Physically-based modeling based on discretized model data (i.e., not the original b-reps) is essential to operating at haptic rates with arbitrary geometry, but results in difficulties with low-clearance assembly. A hand-written series of conditionals considering the b-reps interacting performed the constraint recognition. In an alternate approach to combining physically-based interaction with constraints for haptic virtual assembly, Tching et al. [25] introduced the concept of guide planes. During interaction, collision of specific points with the guide planes would engage a constraint (with two intersecting planes defining a cylindrical constraint), replacing the physically-based simulation for that part.

The goal was to create a constraint-recognition engine that could flexibly recognize that two boundary representation elements, if found to be interacting, would give rise to a geometric constraint. More formally, an implementation was needed for a function from the set of all pairs of boundary representation elements to the set of geometric constraints including a null constraint. As this is to be used in conjunction with existing simulation techniques, exhaustively recognizing all constraints is not necessary, and adding additional constraints to the recognition engine to improve the output is expected in the life cycle of the software. Performance is important.

A technique based on the C++ type system with static typing and avoiding explicit runtime branches (`if`/`switch` statements) and inheritance-based polymorphism was chosen for reasons of maintainability and performance. The problem can be considered as invoking a function whose implementation varies based on the type (broadly construed) of both of its two arguments. This suggests the need for multi-methods.

7.2.1. Discriminated unions and multi-methods

In computer programming, a union is a data type that can hold any one of several, predefined data types. The C language provides a simple implementation of unions, in which each possible contained type appears like a field in a `struct`, but sharing the same memory [16, 17]. Usage of the C `union` facility requires the programmer to keep track of which field was last stored to (and thus what type of data the union contains). Reading different fields has been used as a way of re-interpreting data (type-punning) but can also result in undefined behavior, depending on the compiler and language specification version adhered to. This undefined behavior can lead to not just logic errors but wholesale invalidation of the containing code, or "causing demons to fly out your nose" as C programming lore holds [28].

Tagged unions, in which storage sufficient for several different data types is accompanied by a separate field indicating the currently-stored data type, are both a general data structure concept and a method of more safely using the `union` facility in C for non-type-punning applications. Tagged unions in the conceptual sense are also known as variant types or discriminated unions. They can be used to provide polymorphic behavior in a different way than typical inheritance-based polymorphism. Values held in discriminated unions can be passed around indiscriminately, yet acted upon uniquely based on their contained value type, just as virtual functions implemented in derived classes allow derived instances to be passed around by a pointer to the base object while providing distinct, specialized behavior. Careful, clever use of the C++ type system can permit implementation of discriminated union facilities in ways that retain and enforce type safety at the interface. Early work made substantial use of the free store (heap allocations) and did not bound the types a variant could contain [8], producing a facility that more closely resembles the modern purpose and use of the Boost.Any library [14, 15] than a typical variant type. Another approach, motivated by serialization requirements, used a function-static map from instance pointers to typed values within a member function template to permit arbitrary type "holding", also without bounding the set of containable types [24]. These two early techniques provide compile-time type-safety over a C union, but require an idiom resembling the following to provide polymorphic behavior:

```
void handleVariant(variant v) {
  if (v.isType<std::string>()) {
    handleString(v.get<std::string>();
  } else if (v.isType<int>()) {
    handleInt(v.get<int>());
  } // etc...
}
```

This requires the user of the variant type to make sure that their list of cases (effectively equivalent to a `switch` on an `enum` union type tag as in C) is complete and consistent.

The C++ compiler can enforce type-safety of access, but cannot enforce completeness of polymorphic behavior over a set of types with this idiom. Furthermore, since the two implementations discussed so far have unbounded contained types, no approach for applying polymorphic behavior would allow compile-time checking of completeness.

The 2001 book “Modern C++ Design” by Andrei Alexandrescu pioneered more advanced C++ techniques, among them compile-time abilities through use of templates [2]. Typelists are compile-time, template-based structures whose values are types. A discriminated union’s bounded contained types, then, could be enumerated in a typelist. Through C++ metaprogramming, explored through the “Loki” library introduced in [3] and widely disseminated through the Boost Metaprogramming Library (MPL) [1], the types in the typelist could be used to produce specific variant data types capable of containing only those types and permitting access in a type-safe way. Alexandrescu [2, 4, 5] built on the earlier examples discussed and these compile-time techniques to develop such an implementation. Notably, Alexandrescu applied the Visitor pattern using a template- and typelist-driven generic implementation based on Loki. This allows the idiom above to be replaced with a visitor class with methods accepting each of the possible types in the variant. In situations where all cases should be handled, the visitor class can be derived from a “StrictVisitor” base so that missing cases become compile-time errors [6]. This implementation inspired the design of the Boost.Variant library in the Boost library collection [13]. It similarly accepts a list of types as its template parameter, provides for type-safe storage and retrieval, and provides strict visitor functionality. It relies more on template metaprogramming, and even many of its internal operations are implemented as visitors, rather than with the “fake vtable” in [2]. By using placement-new for value construction, it holds values in stack (not heap) allocations in most circumstances for improved performance.

Standard variant visitation, as well as virtual function calls for inheritance-based polymorphism, are known as “single dispatch” - their behavior is polymorphic on one object (in the virtual function case, `this`). However, behavior that varies polymorphically based on the most specific types of two objects is not available natively in C++. This is what’s known as the binary method problem, or more generally, multiple dispatch or multi-methods [7, 20]. The visitor pattern is one way of applying binary multi-methods in C++, if the visitor itself is considered to be one of the arguments. In traditional use of the visitor pattern, the visitor derives from a base visitor class, and a polymorphic child visited class “accepts” a visitor object and calls the visitor’s “visit” method on itself. These two indirections allow recovery of derived class static type, and resultant specialized implementation selection, for both the visited and visitor object. Relevant disadvantages of the single visitor pattern include the inherent asymmetry between visitor and visited objects. If the visitor is a binary operation outside the visited classes and performs behavior dependent on the most specialized types of two arguments, as discussed in [7], it could be considered triple dispatch. The more general term multi-method will be used to avoid confusion between double dispatch, binary methods, and binary methods performed by effectively triple dispatch.

A useful feature of the Boost.Variant library not found in earlier C++ variant types is its inclusion of not only single visitors, but binary visitors. These are function objects that can be invoked polymorphically on the contained types of two variant objects. Binary visitors provide multi-methods that can take two variant arguments, and keep the visitor-specific functionality out of the visited types. Just as variant visitation eliminates manual `switch` or `if` statements, binary variant visitation can eliminate manually-coded nested `if` statements. Beyond just maintenance considerations, compile-time generic and generative programming in this way is well suited to scientific programming due to its ability to shift computation from run-time to compile time and accommodate diverse algorithms and implementations [27].

The use of discriminated unions, rather than a single class or a polymorphic base class, as a general container type is an important part of the present work's distinct approach. They provide the ability for different types to be passed in as a parameter to a single API method, much as a base class pointer, but they require no common base class and have semantics that are ideal for types whose instances are treated as values, rather than referenced entities. The compile-time metaprogramming functionality is also important to this work.

7.3. Methodology

In this work, instead of a class hierarchy based on inheritance, the individual types of boundary representation (b-rep) entities of interest were represented as unrelated types. Generic behavior and polymorphism was provided by the use of the Boost.Variant library. The `boost::variant` class template specialized on the list containing every boundary feature type is declared with a `typedef` as the generic, effectively polymorphic type `BoundaryRep`. The binary visitor technique provides the compile-time multi-method machinery to allow the basics of this work to proceed: to be able to have a function called with two generic `BoundaryRep` objects and returning a generic `Constraint` object, without inheritance or hand-written branching.

For modularity and functionality, the binary visitor function object invoked in the constraint recognition engine does not directly contain constraint recognition logic. The entire constraint recognition engine logic starting with the binary visitor is structured in three steps, as shown in Figure 7.1. The first step is the binary visitor, whose function-call operator is a method template which captures the specific static types contained in the two passed `BoundaryRep` objects. Armed with this full static type information, it invokes a function template `accumulateConstraint` shown in step two. When possible, this function will proceed to invoke code for a constraint recognition procedure specific to the two types. The design and rationale of this second step is described in a separate section below for clarity.

Each non-null constraint recognition procedure is defined by explicitly specializing a class template called `ConstraintFromBrepPair` with the two boundary representation types and implementing a static method, which receives the two b-rep objects and returns a newly-created generic constraint. These specializations are step three in Figure 7.1. If any run-time checks are needed before actually creating the constraint, such as size compatibility of cylindrical peg and hole, they are performed in this method, with the option of returning a null constraint. Each of these explicit specializations, or "recognized pairs," forms a self-contained unit of code dependent only on the forward declaration of the `ConstraintFromBrepPair` template, the two b-rep types it manipulates, and any constraint type it may create. As such, the code for each recognized pair is located in its own header file in a directory of similar headers, keeping the code clean and readable. This design provides high cohesion (all recognized pairs in one directory, with a header file for each) as well as loose coupling (minimal dependencies required by each recognized pair, each header containing only the code relevant to a particular pair of b-rep types).

7.3.1. Step two: the `accumulateConstraint` function

Returning to the middle level of indirection, the `accumulateConstraint` step is made separate to provide commutativity for the arguments of the recognition multi-method. That is, it does not matter which body has which b-rep; detecting a constraint given the b-rep pair (A, B) implies that a constraint should be detected given the b-rep pair (B, A). To eliminate the risk of missed cases, handling commutativity is built-in at step two as shown in Figure 7.1, at a level above the actual constraint recognition definitions. This makes it automatic, and the design of this step in fact makes it compile-time. There are four function templates named `accumulateConstraint` and thus potentially part of the overload resolution set, which is the set of functions considered at compile time for resolution of calls to a function by that name. However, through the use of the "substitution-failure-is-not-an-error" technique (SFINAE) [18, 26], all but one of them is removed from the overload resolution set for any given b-rep pair, making each call unambiguous. Compile-time meta-

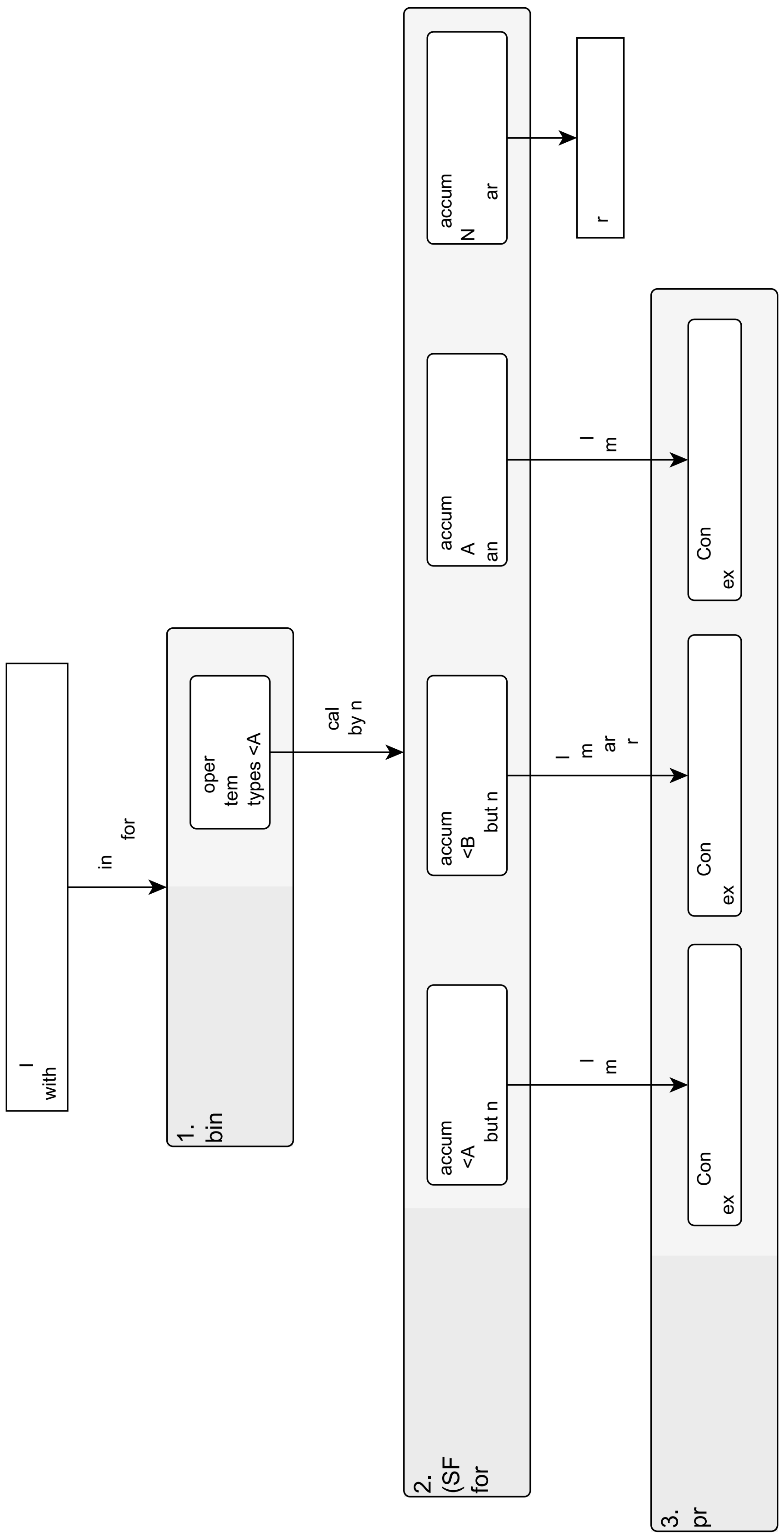


Figure 7.1: Structure of constraint recognition engine code

functions called `IsRecognized<T, U>` and `NotRecognized<T,U>` are used as additional, dummy arguments to select the appropriate overload of `accumulateConstraint` through application of the SFINAE technique. These metafunctions look to a given specialization of `ConstraintFromBrepPair` at compile time to determine if the default implementation, implying no recognized constraint, is being applied, or if an explicit specialization, implying a possible recognized constraint, is available. The metafunctions and the SFINAE technique operate as follows. In the case of `IsRecognized<T, U>`, if for the b-rep types `T` and `U`, `ConstraintFromBrepPair<T, U>` exists as an explicit specialization, that is considered a recognized pair and a nested type definition `IsRecognized<T, U>::type` is made available. To make a function template only available if some given `T` and `U` are recognized, a dummy argument of type `typename IsRecognized<T, U>::type*`, with a default value, is added to the end of the argument list. During overload resolution, to determine which function to invoke when the overloaded name `accumulateConstraint` is called, the compiler tries substituting in the templated types to each candidate, or "overload." If the pair (T, U) are not recognized, the nested `typedef` in the metafunction will not exist, making the argument of invalid type and resulting in what is known as "substitution failure." Rather than issuing a compiler error, the compiler simply eliminates that overload from consideration and moves on, generating an error only if no overload exists that does not result in substitution failure, or if multiple overloads remain that match the parameter list equally well, resulting in an ambiguous call. A similar process occurs in the use of the opposite metafunction `NotRecognized<T,U>`. Adding multiple dummy arguments results in a logical AND of the metafunctions, requiring all of them to be true to avoid substitution failure. The four `accumulateConstraint` overloads contain combinations of these metafunctions in them so that for any given pair of b-rep types, only one of them remains available. The four cases are:

1. For b-rep types `T` and `U`, if `NotRecognized<T,U>` and `NotRecognized<U,T>`, then return a null constraint: the given b-reps do not result in a constraint in either order.

2. For b-rep types `T` and `U`, if `IsRecognized<T,U>` and `NotRecognized<U,T>`, then call the recognition code in `ConstraintFromBrepPair<T,U>`. The b-reps are recognized as a constraint in their given order. (The seemingly-redundant `NotRecognized` condition here and in case 3 are to prevent this method from being available in the case where `T` and `U` are the same type.)

3. For b-rep types `T` and `U`, if `NotRecognized<T, U>` and `IsRecognized<U,T>`, then call the recognition code in `ConstraintFromBrepPair<U,T>`, flipping the order of the

arguments, and passing a flag to indicate that the second body has b-rep `T` and the first body has b-rep `U`. This handles the commutativity case.

4. For arguments that are both of b-rep type `T`, if `IsRecognized<T,T>`, then call the recognition code in `ConstraintFromBrepPair<T,T>`. This is required in conjunction with the `NotRecognized` conditions of cases 2 and 3 to ensure only one overload is available for pairs of identical b-rep types.

While this is described as though it were implemented with `if` statements, it is important to note that selection of one of these four cases happens entirely at compile time, for every possible b-rep type pair. It is also useful to note that these four conditions cover all possible valid cases without any overlap. If faulty conditions are introduced during the addition of a new constraint recognizer, they will often be caught at compile time. Duplicate recognizers would trigger a compiler error indicating a duplicate template specialization. Introducing a recognizer for (B, A) when (A, B) is already recognized and $A \neq B$ would cause all overloads of `accumulateConstraint` to be eliminated, resulting in a compiler error at that step. Compare this to an alternate implementation using `if` statements, in which no indication would be given about duplicated conditions, which may either be ignored or produce unexpected results. At most, a `switch` implementation might trigger a compiler warning if warning levels are high enough. No notice would be given by an alternate implementation if a commutative case failed to be handled, an error noted in the behavior of existing research code. In this design, commutativity is automatically handled as it is implied by the problem definition and accommodated in step two of Figure 7.1.

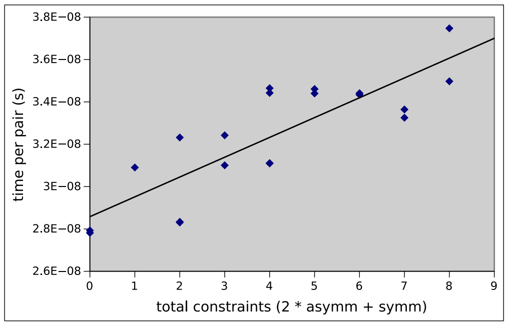
The explicit specialization provided by each recognized pair must be available in the translation unit containing `accumulateConstraint` for the compiler to generate code to select it. Therefore, enabling a recognized pair simply requires adding the corresponding header to the list of `#includes` in the implementation file. The template-driven, compile-time style of the code results in most code existing in header files, that are eventually included and their templates instantiated in a single translation unit (`.cpp` file) with the generic entry-point function as the only API exposed to code using the constraint recognition engine.

Due to compiler inlining, the additional layer of indirection used by `accumulateConstraint` is eliminated during compilation and has no run-time overhead. In the binary visitor, the compiler generates specializations of `accumulateConstraint` and `ConstraintFromBrepPair` for every combination of b-rep types that can be held in the `BoundaryRep boost::variant`. The full constraint recognition engine is thus generated in the compiler at the same time, in a single translation unit, and optimizations like inlining are performed which eliminate the overhead of function calls within the engine. The three steps within the recognition engine are what's known as "zero-overhead abstractions," a hallmark of modern C++ style. The compiler can choose the branching implementation that it judges most efficient given all the information about the possibilities, since it is responsible for generating the equivalent of the large `switch` statement that this code externally appears to implement.

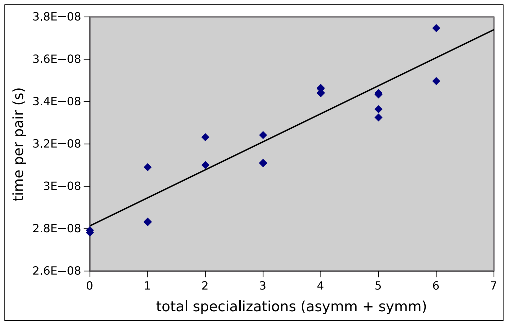
7.4. Performance Characterization

The interaction between the compiler's optimizations and the metaprogramming within the `boost::variant` visitor results in a generated algorithm performing the decision making based on types, which is difficult to characterize by hand. A number of testing runs were performed to determine asymptotic behavior experimentally. In each test, a test harness first ran the task for a set number of iterations (1,000,000). Two additional times, the time-per-iteration was computed and used to choose a new iteration count for a run estimated to take 3-seconds. The results from the third and final run were used. This procedure was used to account for unknown run-time while measuring over a sufficiently long period, as well as to ensure relatively uniform running conditions, including, for instance, warm instruction and data caches. Trials were conducted on a computer running Linux, with a 2.4 GHz Core 2 Duo processor and the software compiled with GCC 4.6.3.

The timed task consisted of running the constraint recognition engine on two sets, each containing one of each type of b-rep. The independent variables, configured at compile time, were the number of symmetric and asymmetric pairs with a specialization to recognize a constraint. A symmetric pair refers to a constraint recognizer operating on a pair of types that are the same, while an asymmetric pair refers to a constraint recognizer operating on a pair of types that are not the same. Asymmetric pairs, due to commutativity, result in effectively two constraints recognized (the explicit one and its counterpart with arguments in the other order). The number of recognizers included was manipulated by simply disabling the inclusion of individual `ConstraintFromBrepPair` headers, starting from an "all enabled" state. We anticipated run-time to increase proportional with either the number of specializations (symmetric + asymmetric) or constraints (2 times asymmetric + symmetric). The reason for the latter case is that each specialization that defines an asymmetric constraint recognition implies an auto-generated handling of the reverse-ordered case due to commutativity.



(a) Time vs. constraints enabled



(b) Time vs. specializations enabled

Figure 7.2: Time per pair, with lines of best fit for two choices of independent variable

7.4.1. Results

The raw timing data obtained from trials of disabling and enabling the various constraint recognizers can be found in Table 7.1. Two linear regression models were fitted to the data, one with the explanatory variable "total constraints" (2 times asymmetric + symmetric) plotted in Figure 7.2a, and one with explanatory variable "total specializations" (symmetric + asymmetric) plotted in Figure 7.2b. For the first model, $R^2 = 0.7075$, while for the second model, $R^2 = 0.8517$. Thus, the performance of the variant-based constraint recognition engine appears to depend on the total number of specializations, rather than the number of constraints in total. This is an interesting finding, that suggests that the visitor algorithm used by `boost::variant`, when combined with an optimizing compiler and the described implementation of handling cases of asymmetric constraints, does not impose a performance penalty for detecting reversed-order arguments. This compares favorably with a naive handwritten generic version, which would have to consider each constraint individually, as opposed to being able to combine that run-time cost for a pair of constraints that differ only in order.

7.5. Discussion and Conclusion

The method of constraint recognition described here provides a high-performance constraint recognition engine that is easy to maintain. It represents different b-rep elements as distinct, data-oriented types without a common base class. Generic code for handling b-reps is enabled by using `boost::variant` to create a discriminated union type that can contain any of the b-rep element types defined. Constraint recognition code is kept out of the b-rep classes, and defined as a multi-method applied by a binary visitor treating both arguments equally. Run-time constraint recognition is split into minimal routines that

Table 7.1: Recognition timing results

Asymmetric specializations	Symmetric specializations	Total specializations	Total constraints	Time (ns) per pair	Pairs per ms
0	0	0	0	27.928	35 806.3592
0	0	0	0	27.813	35 954.4098
0	1	1	1	30.9023	32 360.0509
0	2	2	2	32.3186	30 941.9344
0	3	3	3	32.4213	30 843.9205
0	4	4	4	34.646	28 863.3608
0	4	4	4	34.4224	29 050.8506
1	0	1	2	28.3047	35 329.8215
1	0	1	2	28.3356	35 291.2943
1	1	2	3	31.0053	32 252.5504
1	2	3	4	31.108	32 146.0718
1	2	3	4	31.0936	32 160.9592
1	3	4	5	34.3989	29 070.6970
1	3	4	5	34.6075	28 895.4706
1	4	5	6	34.4081	29 062.9241
1	4	5	6	34.337	29 123.1034
2	3	5	7	33.6412	29 725.4557
2	3	5	7	33.2519	30 073.4695
2	4	6	8	34.9714	28 594.7946
2	4	6	8	37.4733	26 685.6669

handle a given pair of b-rep types through template specialization. Additional recognizers can be added by creating a new header file with an appropriate specialization, and enabled by including this header in a main header file. Commutativity of arguments to the recognizers is handled automatically and transparently through template metaprogramming, and without impacting run-time performance. Erroneous conditions, such as adding a duplicate constraint recognizer or a recognizer for arguments already recognized in the reverse order, are diagnosed at compile-time by a compiler error, easing maintenance and further development. By making all three internal layers of the recognition engine available to the compiler in the same translation unit, the compiler can fully apply inlining and other optimizations resulting in optimal code generation, with the abstractions introducing zero run-time cost over an equivalent fully hand-written monolithic method. The end result is a constraint recognition engine that is easy to develop and maintain, and easy to use with the workings of the metaprogramming engine compartmentalized behind a single-method public API. Future work involves applying this constraint recognition engine in the context of a virtual assembly application to perform localized constraint recognition following physically-modeled collision detection, allowing an improved collision response based on constraint knowledge.

Acknowledgment

This work was performed at the Virtual Reality Applications Center at Iowa State University. This material is based upon work supported by the National Science Foundation under Grant No. CMMI-0928774. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Bibliography

[1] David Abrahams. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005. ISBN 0321227255.

[2] Andrei Alexandrescu. An Implementation of Discriminated Unions in C++. In *OOPSLA 2001, Second Workshop on C++ Template Programming*, Tampa Bay, 2001.

[3] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0-201-70431-5.

[4] Andrei Alexandrescu. Generic Programming: Discriminated Unions (I). *C/C++ Users Journal*, 20(4), 2002. URL http://erdani.com/publications/cuj-04-2002.php.

[5] Andrei Alexandrescu. Generic Programming: Discriminated Unions (II). *C/C++ Users Journal*, 20(5), 2002. URL http://erdani.com/publications/cuj-06-2002.php.

[6] Andrei Alexandrescu. Generic Programming: Discriminated Unions (III). *C/C++ Users Journal*, 20(6), 2002. URL http://erdani.com/publications/cuj-08-2002.php.

[7] Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On Binary Methods. *Theory and Practice of Object Systems*, 1(3):221—-242, 1995.

[8] Fernando Cacciola. An Improved Variant Type Based on Member Templates. *C/C++ Users Journal*, 18(10):10–21, October 2000. ISSN 1075-2838. URL http://www.drdobbs.com/an-improved-variant-type-based-on-member/184401293.

[9] Richard G. Dewar, Ian D. Carpenter, James M. Ritchie, and John E. L. Simmons. Assembly planning in a virtual environment. In *Innovation in Technology Management. The Key to Global Leadership. PICMET '97*, pages 664–667. IEEE, 1997. ISBN 0-7803-3574-0. doi:10.1109/PICMET.1997.653557. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=653557.

[10] Daniela Faas. *A hybrid method for haptic feedback to support manual virtual product assembly*. Ph.d. dissertation, Iowa State University, 2010. URL http://lib.dr.iastate.edu/etd/11480/.

[11] Daniela Faas. A Hybrid Method for Haptic Feedback to Support Manual Virtual Product Assembly. In *ASME 2011 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, pages 1–8, Washington, DC, USA, 2011. ASME. doi:10.1115/DETC2011-47665. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1639771.

[12] Daniela Faas and Judy M. Vance. BREP Identification During Voxel-Based Collision Detection for Haptic Manual Assembly. In *ASME 2011 World Conference on Innovative Virtual Reality (WINVR 2011)*, pages 145–153, Milan, Italy, 2011. ASME. ISBN 978-0-7918-4432-8. doi:10.1115/WINVR2011-5524. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1623634.

[13] Eric Friedman and Itay Maman. Boost Variant, 2003. URL http://www.boost.org/doc/html/variant.html.

[14] Kevlin Henney. Valued Conversions. *C++ Report*, 12(7):37–40, 2000. ISSN 1040-6042. URL http://www.two-sdg.demon.co.uk/curbralan/papers/ValuedConversions.pdf.

[15] Kevlin Henney. Boost Any, 2001. URL http://www.boost.org/doc/html/any.html.

[16] ISO JTC1/SC22/WG14. ISO/IEC 9899:1999 + TC1 + TC2 + TC3, Information technology – Programming languages – C (C99 Final Draft N1256). Technical report, ISO, 2007. URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

[17] ISO JTC1/SC22/WG14. ISO/IEC 9899:2011, Information technology – Programming languages – C (C11 Final Draft N1570). Technical report, ISO, 2011. URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

[18] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Concept-controlled Polymorphism. In *Proceedings of the 2nd International Conference on Generative Programming*

and Component Engineering, GPCE ’03, pages 228–244, New York, NY, USA, 2003. Springer-Verlag New York, Inc. ISBN 3-540-20102-5. doi:10.1007/978-3-540-39815-8_14. URL http://link.springer.com/chapter/10.1007/978-3-540-39815-8_14.

[19] Zhenyu Liu and Jianrong Tan. Constrained behavior manipulation for interactive assembly in a virtual environment. *The International Journal of Advanced Manufacturing Technology*, 32(7-8):797–810, March 2006. ISSN 0268-3768. doi:10.1007/s00170-005-0382-5. URL http://link.springer.com/10.1007/s00170-005-0382-5.

[20] Peter Pirkelbauer, Yuriy Solodkyy, and Bjarne Stroustrup. Open Multi-Methods for C++. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, pages 123–134, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-855-8. doi:10.1145/1289971.1289993. URL http://doi.acm.org/10.1145/1289971.1289993.

[21] James M. Ritchie, Richard G. Dewar, and John E. L. Simmons. The generation and practical use of plans for manual assembly using immersive virtual reality. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, 213(5):461–474, May 1999. ISSN 0954-4054. doi:10.1243/0954405991516930. URL http://pib.sagepub.com/content/213/5/461.abstract.

[22] Abhishek Seth, Judy M. Vance, and James H. Oliver. Combining Geometric Constraints With Physics Modeling for Virtual Assembly Using SHARP. In *ASME 2007 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages 1045–1055. ASME, 2007. ISBN 0-7918-4803-5. doi:10.1115/DETC2007-34681. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1604120.

[23] Abhishek Seth, Judy M. Vance, and James H. Oliver. Combining Dynamic Modeling With Geometric Constraint Management to Support Low Clearance Virtual Manual Assembly. *Journal of Mechanical Design*, 132(8):081002, 2010. ISSN 10500472. doi:10.1115/1.4001565. URL http://mechanicaldesign.asmedigitalcollection.asme.org/article.aspx?articleid=1450105.

[24] Volker Simonis. Chameleon Objects, or how to write a generic, type safe wrapper class. *C++ Report*, 12(1), January 2000. ISSN 1040-6042. URL http://www.progdoc.de/papers/chameleon/chameleon.htm.

[25] Loïc Tching, Georges Dumont, and Jérôme Perret. Interactive simulation of CAD models assemblies using virtual constraint guidance. *International Journal on Interactive Design and Manufacturing (IJIDeM)*, 4(2):95–102, April 2010. ISSN 1955-2513. doi:10.1007/s12008-010-0091-7. URL `http://www.springerlink.com/index/10.1007/s12008-010-0091-7`.

[26] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0201734842.

[27] Todd Veldhuizen. Techniques for Scientific C++. Technical report, Indiana University Computer Science, 2000. URL `http://www.cs.indiana.edu/pub/techreports/TR542.pdf`.

[28] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Undefined behavior: what happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems - APSYS '12*, pages 1–7, Seoul, South Korea, 2012. ACM. ISBN 9781450316699. doi:10.1145/2349896.2349905. URL `http://dl.acm.org/citation.cfm?doid=2349896.2349905`.

[29] Yong Wang, Shaikh Imtiyaz, Uma Jayaram, and Sankar Jayaram. Methods and Algorithms for Constraint-based Virtual Assembly. *Virtual Reality*, 6(4):229–243, August 2003. ISSN 1359-4338. doi:10.1007/s10055-003-0106-9. URL `http://link.springer.com/10.1007/s10055-003-0106-9`.

CHAPTER 8. IMPROVING PHYSICALLY-BASED COLLISION RESPONSE BY RESTORING DEGREES OF FREEDOM

A paper in preparation for submission to IEEE *Transactions on Haptics*

Ryan A. Pavlik, Judy M. Vance

Abstract

The long term goal of our research is to improve product design through virtual assembly. Our vision is to be able to assemble any arbitrary set of CAD models using immersive technology with haptic feedback. Assembling parts that mate with very low clearances is a challenge for traditional haptic simulations because the collision models are generally tessellations or voxelizations of the exact CAD geometry which result in unrealistic and infeasible assembly situations. Implementing snap-to or constraint-based methods artificially takes the user out of the loop and relies on preprogrammed assembly configurations. This paper presents a hybrid technique that combines a voxel-based collision detection and force response algorithm with constraint knowledge derived from the original precise geometry of the CAD models. This approach relates b-rep data to voxel data and uses the b-rep data when automatically identifying potential constraints. Next, the constraint knowledge is applied in a new way, by selectively restoring degrees of freedom at a local (voxel) level. This approach sidesteps difficulties with constraint solving, avoids creating phantom sources of force, and allows the algorithm to provide enhanced haptic interaction with low-clearance assemblies. The results show that this method correctly identifies various constraints and runs at speeds fast enough to support smooth and stable haptics.

8.1. Introduction

Haptic force-feedback poses a special problem for physics engines due to the need to maintain a computation rate of 1000 Hz for haptic stability. Exact parametric boundary

Ryan A. Pavlik was the primary author and researcher.

representations (b-reps) of CAD geometry are approximated, typically as triangulated meshes, for graphic display, and further discretization and approximation is needed to perform high-rate collision detection and response. However, the collision detection and response results from discretized approximations of geometry do not necessarily reflect the nature of the original geometry. In particular, degrees of freedom are easily lost when collisions are detected between tesselations of the original b-reps. If the clearance between objects exceeds some multiple of the discretization level of detail, traditional virtual assembly methods can be used. However, real-world CAD models often have smaller clearances. This makes assembly artificially impossible. In the general case, a model can be created with b-reps suitable for assembly that nevertheless cannot be assembled in any tessellated geometry-based simulation. This paper outlines a method that enables low-clearance haptic assembly of CAD models through a hybrid usage of both a tessellated geometry-based collision method (in coarse movement) and the original b-rep data (in fine manipulation and assembly).

8.2. Previous Work

Virtual assembly systems can be generally categorized into two types: those that operate primarily based on constraints, and those that use physically-based modeling [14]. Systems based on constraints, including snap-to-fit (positional constraint) and preprogrammed assembly constraints, allow all foreseen assembly to take place, but take the user out of the loop by providing guidance beyond physical simulation. Physically-based modeling does not provide nonphysical guidance, but computational limitations introduce challenges for low-clearance virtual assembly at haptic rates.

Anantha et al. [1] describes a system for generating textual assembly plans and recognizing degrees of freedom based on constraint recognition and satisfaction. Common types of constraints are identified and used to define the relationship of parts in an assembly during an initial CAD modeling process. The process involves a two-part representation scheme where b-reps and part features are separate and re-united later in the process. Difficulties arise in processing full constraint sets for constraint satisfaction. Further, in the case of under-constrained systems, the degrees of freedom which are not constrained are transformed into kinematic joints representing allowable motion under the constraints. The system does not involve a virtual environment.

Fernando et al. [6] presents a virtual assembly application that exclusively uses constraint-based modeling instead of physically-based modeling. The OpenGL Optimizer scene graph maintains precise and tessellated geometry representations in the same graph structure.

Using only constraint-based modeling can artificially guide the user into a preprocessed assembly state, reducing direct user interaction and influencing subsequent evaluations of assembly suitability.

The Virtual Assembly Design Environment (VADE) [7] is a physically-based modeling simulation with constraints imported from design intents created in the original CAD software. It provided interactive, two-way communication between a tightly-coupled commercial CAD package (Pro/Engineer) and the virtual assembly simulation.

Zachmann and Rettig [18] described "natural interaction" as too challenging, and at times unnecessary, as a goal of overall virtual reality applications in the engineering process. They described "intuitive interaction," of which "natural interaction" is considered a subset, as the proper standard for virtual reality simulations, and noted that virtual assembly, by virtue of benefiting from natural (not just intuitive) interaction, was among the most challenging virtual reality applications. Constraints are described as a way to facilitate precise positioning given free-space six degree of freedom user input, presuming the absence of haptic force feedback. Zachmann and Rettig considered abstract constraints from CAD packages, tool use-related constraints for accessibility studies, a specialized handler for sliding contact, and a model for precise grasping.

Liu and Tan [8] describes a "constrained behavior manager" (CBM) for virtual assembly software. This method assumes the absence of haptic force-feedback and associated ability to affect the human interaction through any mode other than visual display. It imports geometric constraints, as well as dimensional constraints and assembly sequence constraints, from the CAD software package.

Yang et al. [17] presents a virtual assembly system partially based on imported constraints. A bounding box check, then a position-orientation proximity check, is used to determine whether free manipulation or constrained movement between two parts should be active. It presents a correspondence between a taxonomy of degrees of freedom and constraints, as well as rules for reduction of the degrees of freedom from two constraints. It describes a different sort of constraint recognition that focuses on designed constraints with planes and lines.

Seth et al. [13] aimed at recognizing the colliding precise geometry and, as further detailed in [12], identifying constraints automatically from the b-reps. While constraint recognition and virtual assembly was possible, the process was too slow to be able to provide haptic force feedback.

8.2.1. Two Stage Virtual Assembly

A theoretical treatment of assembly simulation as consisting of a coarse and a fine stage was presented in Vance and Dumont [16]. In the coarse manipulation stage, parts may interact on a large scale as a user performs general positioning and alignment. In the fine assembly stage, where difficulty with low clearances is found, precision in movement and simulation is required. This research was based on a number of related projects in the low-clearance assembly field that seemed to arrive at a similar conclusion from different directions.

Tching et al. [15] combined an tessellated geometry method for coarse assembly with a precise, constraint-based method for fine assembly. It detected and responded to imminent fine assembly by intersection with "constraint planes." Interaction with these planes would eventually locally override the tessellated geometry method to provide smooth low-clearance assembly. However, these features had to be manually predefined for each CAD model in a scenario as a separate preprocessing step. Faas [3] presented a hybrid method that relied on automatically identifying constraints based upon b-rep data, determining the level of constraint alignment, and generating a combined force that blended the existing contact forces to arrive at constraint satisfaction.

For the scenario in consideration, it can be reasonable to assume that the parts as provided are possible to assemble. Faas and Vance [4] made use of this assumption to attempt to improve low-clearance haptic assembly. That method automatically retracted the volume of the tessellation (here, a pointshell for use with a voxmap) to increase clearances dynamically at run-time during the fine assembly stage.

Initial work by Faas and Vance [2, 5, 3] on combining a tessellated geometry algorithm (Voxmap PointShell, or VPS [9, 10]) with precise knowledge from b-reps was implemented in an application known as SHARP-2. The work described in this paper builds conceptually on Faas, and also uses VPS as the tessellated geometry algorithm, but is otherwise independent. This paper describes new work implemented and tested in modular software components and in the virtual assembly application SPARTA, the *Scalable Platform for Advanced Research and Teaching in Assembly* [11].

8.2.2. General Hybrid Algorithm

We decomposed the hybrid method presented in Faas [3] into several separate steps in a two-level hierarchy. The first category of tasks takes place only once during a simulation, and is thus not time-critical. In some ways, it can be considered a form of preprocessing. However, since it is unsupervised preprocessing, it is not preprocessing of the type required by Tching et al. [15]. It can be performed at the time of model loading. The second category consists of those steps that take place repeatedly at run-time during collision events, so-called "run-time operations." These steps are time-critical since their execution time must be sufficiently fast to fit in the 1 ms allotted for each physics iteration.

Discretized collision units (DCUs) is a generic term used here for the tessellated geometry data structures made available at collision time, with some member suitable for use as a form of look-up key. In this work, surface voxels in VPS are the collision units, but the method is not limited to use of voxelized data structures. The terms "voxel" and "DCU" could be interchanged in this paper in nearly all cases except for the specific implementation details specific to the use of VPS.

Before run-time, such as during model loading, the following steps, the first category of subproblems, take place:

1. The model is loaded and voxelized.

2. The set of b-reps corresponding to each voxel are determined.

3. The result of the b-rep–voxel association is stored for fast run-time look-up.

The first step is part of the normal process used in the case where only the tessellated geometry algorithm is being used. The second step is the initial computation of the association of discretized collision units with original boundary representations. This step necessitates the extraction of both the original b-reps from a model, as well as its transformation into DCUs, such as through voxelization, to provide input data. In the case of voxelization, only the surface voxels may be involved in collisions, so only they need to be associated with precise representations. Informally, the second step answers the question "if something collides with this voxel, what b-reps is it colliding with?" However, since this is a computationally-expensive step, taking several minutes for complete test models in previous work, it is not suitable for run-time. Thus, the results are pre-calculated and, in the third step, stored in a data structure that allows the look-up from voxel to set of b-reps to occur quickly during run-time. In the specific case of applying this work with VPS, each opaque voxel data structure used by VPS contains 12 bits of so-called "private data" that may be used by the application, set in advance and returned during collision detection. We store an ID used to perform a look-up to a set of b-reps in this private data. Step three forms the first half of the "voxel to b-rep look-up problem."

At run-time, during each frame of simulation, collision detection is performed between each pair of bodies by the tessellated geometry collision detection algorithm and response forces are computed. The second category of tasks accomplishes this process. Rather than performing collision detection and response in a single step as is typical with the VPS software, the following steps take place in the hybrid method.

1. Once a collision is detected, a list of collision pairs is returned. Each entry includes the voxel locations in both bodies, the computed penalty force to apply at those voxels, and the “private data” value for the two interacting voxels.
2. For each entry in the list:
    (a) The private data values are used to look up the sets of b-reps corresponding to the two colliding voxels.
    (b) The b-rep sets are submitted to an automatic constraint recognizer that performs all possible pair-wise comparison of b-reps looking for any pairs that would result in a constraint.
    (c) If a constraint is recognized, a measure of constraint alignment or engagement is computed for use in varying the intensity of constraint response.
    (d) The initially-computed response force is modified incorporating the additional knowledge imparted by the results of the constraint recognition and its precise geometry knowledge to compute a final local collision response force.
    (e) Finally, the appropriate torque is computed for the response force applied at the voxel locations.
3. Response forces and torques, possibly modified from the tessellated geometry algorithm output, are summed, and at the end of the frame used to compute the new position and velocity of the colliding objects.

8.3. Analysis of Tasks

We identified five independent tasks that together form a particular hybrid method of collision detection and response. Rather than a single algorithm, we propose that there is a class of algorithms or methods with common structure, but that differ in their solutions to these individual subproblems.

8.3.1. Voxel to B-rep Initial Association

The purpose of this task is to determine which b-reps in the original model give rise to each particular discrete collision unit (voxel) in turn, so that at run-time, when colliding voxels are detected, they can be traced back to the precise b-reps. This initial association is separate from "voxel to b-rep look-up" because known and foreseen implementations are very time intensive. This step as implemented in [5] loaded the original model in b-rep form as well as a voxel-sized cube into a commercial CAD modeling kernel, and asked the kernel for intersecting b-reps as the voxel cube was moved iteratively to the position of each of the surface voxels in the voxelized model. Other methods of implementing this step are possible, but are not explored in the scope of this work.

8.3.2. Voxel to B-rep Look-up

This task must cache the results of the initial association in a space-efficient manner so that at run-time, look-up from a colliding voxel to its set of b-reps is very fast. In the present implementation using VPS, each opaque voxel structure managed by the VPS library offers 12 bits of user-accessible storage. We use that storage for an integer index into a look-up table containing the sets of b-reps. This provides constant-time retrieval at run-time. This step also provides opportunities for further optimization, which are explored in Chapter 6.

8.3.3. Automatic Constraint Recognition

Given the b-rep sets of two colliding voxels, we want to detect if a geometric constraint exists. In this case, detecting all possible constraints is not necessary. Identifying additional constraints can further improve the quality of interaction by increasing the number of instances in which constraint knowledge can be beneficially applied. However, since these geometric constraints are being used in combination with a physically-based tessellated geometry algorithm, we do not need to find all constraints to prevent object interpenetration.

Existing work in [3] is largely based on [12], with hand-coded conditionals used to recognize constraints. We have implemented a more robust, flexible, and scalable constraint recognition engine that eliminates most hand-coded conditionals with a novel design. The design uses template metaprogramming and generative programming to provide for a more declarative style of coding constraint recognition components, which provides for simpler expansion of the engine with additional constraints. The design also automatically handles the inherent commutativity of arguments to an abstract "constraint recognition function." It is discussed more fully in Chapter 7.

8.3.4. Constraint Alignment Measure

This task's purpose may not be immediately apparent. It arises from the need to know how close to fully aligned the constraint is–and thus how strongly to apply the constraint knowledge. It returns a value in the range $[0, 1]$ where 1 means the constraint is fully aligned. Previous work in [3] proposed the formula

$$0.5 \cdot e^{-\theta/\theta_0} + 0.5 \cdot e^{-d/d_0} \tag{8.1}$$

where θ is the angle between axis or vector aspects of the b-reps in the two bodies, while d is the shortest perpendicular distance between those axes. The value θ_0 is a dimensionless scale factor found by dividing the voxel size by some measure of insertion depth, while d_0 is the voxel size. We have modified this equation in the present work. The modified constraint alignment measure used in this work, for the example of a cylindrical constraint, takes the form

$$0.5 \cdot e^{-\theta} + 0.5 \cdot e^{-d/d_c} \tag{8.2}$$

The first exponent was replaced simply with the unscaled $-\theta$. For a cylindrical constraint, the dimensionless second exponent was computed using the clearance $d_c = r_1 - r_2$, where r_1 is the radius of the hole (larger) cylinder and r_2 is the radius of the pin (smaller) cylinder, as well as an empirically-determined constant scale. Similar measures are used for other constraints, and there are opportunities for improving these measures.

8.3.5. Application of Constraint Knowledge

A method for the "application of constraint knowledge problem" was implied in Faas [3], which used a proprietary geometric modeling kernel and constraint solver. In that work, a "constraint force" was computed as the force and torque of a 6-DoF spring between a body's current position and the position reported as the solution to the constraint problem. The overall response force and torque was the linear combination of the force and torque from the discretized algorithm and the constraint force and torque. The coefficients of the linear combination served to interpolate between these two collision response sources, using the constraint alignment measure as a parameter. However, the computational overhead of the constraint solver resulted in the update rate dropping from 1000 Hz to 100 Hz during assembly. An alternate approach to the "application of constraint knowledge problem" forms a key part of this paper's contribution.

8.4. Methods of Applying Constraint Knowledge

When proceeding from a disassembled state directly toward a correct assembly of parts, such as when seeking to develop assembly plans, the constraints found and used are likely to match those originally designed in the product. However, in a free-form "open world" interaction environment, collisions take place that might not be expected during

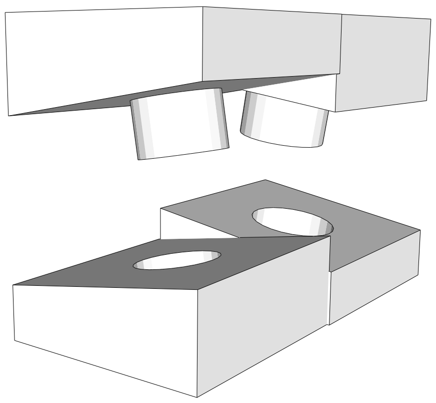


Figure 8.1: Conflicting cylinders. Incompatible constraints within a single part.

a precise assembly, yet would be expected to produce appropriate force feedback. It is particularly in these cases of "mis-assembly" or exploration that the approach of computing the "precise" response by constraint solving shows some distinct weaknesses. To begin, the following examples highlight cases that are not easily handled by algorithms that rely on constraints only. They are artificial examples but represent "corner cases" that are possible with application of virtual assembly to non-artificial test scenarios.

8.4.1. Challenges for Constraint-Based Algorithms

Some situations result in constraints that cannot be simultaneously satisfied. Figure 8.1 shows one artificial instance: both of the cylinders of the top part can be in collision with their appropriate holes at the same time; however, since they are not curved (they are angled cylinders), they cannot be assembled. In this case, the nature of a single part's geometry (two cylinders with axes that are not parallel) forbid simultaneous engagement of a particular set of constraints. In a constraint-solving approach with a global (per part or per colliding part pair) perspective, a decision would have to be made in implementation on how to handle such conflicting constraints in such a way that is physically accurate, not allowing full assembly, while providing smooth feedback for whatever partial assembly interaction is possible. A constraint-solving-based approach to the hybrid method that did not explicitly consider situations such as this would result in unknown and likely unexpected behavior when presented with such a scenario.

In Figure 8.2, though the pins of the top block could be inserted into some theoretical block with holes, and some theoretical block with pins could fit in the holes of the bottom block, this particular combination of parts won't fit together: the distance between the pins is different than the distance between the holes. Yet, as illustrated, cylinder–cylinder collisions could be detected on both pins at once, thus producing two cylindrical constraints.

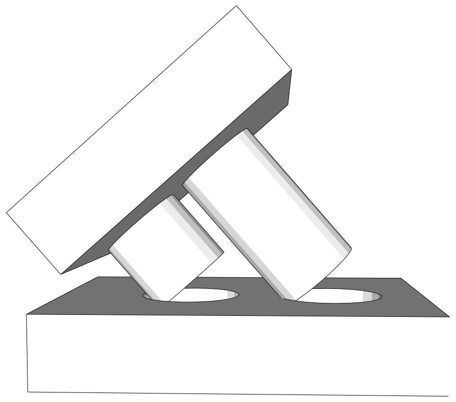


Figure 8.2: Incompatible parts. Conflicting constraints detected between two parts.

Detecting this situation is a different problem than detecting incompatible constraint sources within a single part like in Figure 8.1. A constraint-solving approach would have to establish how to detect this situation, and how to handle it. One approach might be detecting and disabling both constraints when they are both recognized for a frame. This would still allow insertion of a single pin with hybrid method, since during a feasible pin insertion only one of the two cylindrical constraints would be recognized. However, as noted previously, this also presents a case that would need explicit handling in a solution.

8.4.2. Alternate Approach: Restoration of Degrees of Freedom

The above examples demonstrate the difficulty involved in arriving at a meaningful overall collision response from the hybrid method when using constraint solving and force blending as in [4, 3] to solve the "application of constraint knowledge" task. However, returning to the original motivating problem, there is an insight that permits an alternate approach. The impetus for using this hybrid method is that algorithms using only a tessellated geometric representation *artificially remove degrees of freedom* from a system of interacting bodies. During interaction, the collisions detected and the computed responses are known at a fine-grained, local level (DCU, e.g. voxel).

Instead of focusing on constraint solving, we instead use the constraint information to reduce/eliminate the components of the forces induced by the tessellated geometry that are countering known degrees of freedom. For instance, in plane-plane sliding, contacting surfaces should slide freely. Therefore, any component of the force vectors impeding movement along the surface of the plane could be reduced or zeroed. An advantage of this approach is that it can be performed local to each DCU, and thus does not require resolving situations of multiple constraints as discussed above. Furthermore, it completely avoids constraint solving. This avoids ambiguity in under-constrained systems, as well as the generation of an artificial "constraint force" as in Faas. By operating at a local level, degree of freedom restoration (DoF-restoration) resolves the problem of inaccurate collision response at the first opportunity during the collision step, by modifying the individual forces from the penalty method that are summed to produce the global collision response.

In all recognized constraint situations, in addition to details about the geometric constraint, we take as input:

$\mathbf{x}$	The coordinates of the point (center of voxel) under consideration
$\vec{f}$	The collision response force at $\mathbf{x}$ computed by the tessellated geometry collision algorithm
α	A blending factor in $[0, 1]$ where a value of 1 implies full alignment of the constraint and full restoration of appropriate degrees of freedom (the result of the "constraint alignment measure" task)

All input, both this common input and the constraint-specific input, is transformed into the world coordinate system before performing subsequent calculations. The recognized constraint determines what components of the force vector would oppose a degree of freedom that should be present. The output of the computations is a modified collision response force vector $\vec{f}'$ in which those components are scaled down or eliminated based on α. This $\vec{f}'$ is used in place of the original $\vec{f}$ in the remainder of the collision response algorithm for computing torque and accumulating overall collision response force.

The full list of constraints recognized in the present implementation is as follows.

- Circle-Cylinder: Cylindrical constraint
- Cylinder-Cylinder: Cylindrical constraint
- Plane-Plane: Planar constraint

Anticipated extensions include:

- Cone-Cone: Cylindrical constraint, unidirectional
- Sphere-Cylinder: Linear annular constraint
- Sphere-Sphere: Spherical constraint

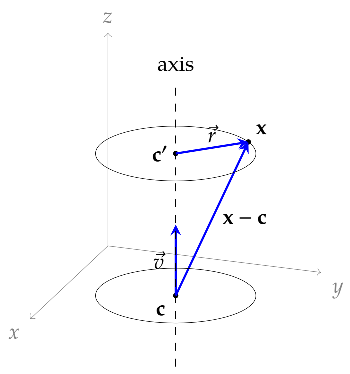


Figure 8.3: Restoring rotational DoF - three-dimensional view

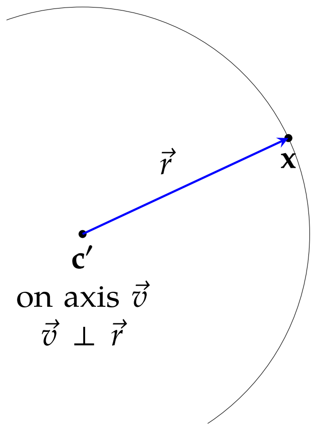


Figure 8.4: Restoring rotational DoF - two-dimensional section view of cylinder at DCU point

Response to a cylindrical constraint

A cylindrical constraint provides the following parameters, which partially characterize a cylinder: a unit vector axis $\vec{v}$ and a known point along that axis $\mathbf{c}$. Cylindrical constraints should permit rotation about the axis $\vec{v}$, as well as translation along it. Finding a modified $\vec{f}$ that restores those freedoms requires scaling down all forces that exist tangent to the surface of the cylinder described by $\vec{v}$, $\mathbf{c}$, and $\mathbf{x}$. The nearest point $\mathbf{c}'$on the axis to $\mathbf{x}$ is found by projecting $\mathbf{x} - \mathbf{c}$ on to $\vec{v}$ (see Figure 8.3):

$$\mathbf{c}' = \mathbf{c} + (\vec{v} \cdot (\mathbf{x} - \mathbf{c}))\, \vec{v} \tag{8.3}$$

The vector $\vec{r}$

$$\vec{r} = \mathbf{x} - \mathbf{c}' \tag{8.4}$$

is an outward-pointing normal at $\mathbf{x}$ on the cylinder (see Figure 8.4), and thus $\vec{n}$ computed as

$$\vec{n} = \frac{\vec{r}}{|\vec{r}|} \tag{8.5}$$

is the unit normal vector. It is also the only direction in which the full forces of the tessellated geometry collision algorithm are considered accurate and should be fully preserved. The force is then decomposed into forces along the direction of the normal, $\vec{f}_{||}$, and the forces perpendicular to the normal, $\vec{f}_{\perp}$.

$$\vec{f}_{||} = \left(\vec{f} \cdot \vec{n}\right) \vec{n} \tag{8.6}$$

$$\vec{f}_{\perp} = \vec{f} - \vec{f}_{||} \tag{8.7}$$

With decomposition complete, we compose the modified force, $\vec{f}'$, by scaling the forces that oppose desired degrees of freedom.

$$\vec{f}' = \vec{f}_{||} + (1 - \alpha)\vec{f}_{\perp} \tag{8.8}$$

Response to a planar constraint

The planar constraint provides input $\vec{n}$, a unit normal vector for the plane. Response to a planar constraint is similar to response to a cylindrical constraint, in that the goal is restoration of movement along directions perpendicular to the normal. A planar constraint offers the added consideration that forces along the normal in the direction of the normal (that is, toward the other plane) should also be eliminated. A non-negative magnitude of $\vec{f}$ along $\vec{n}$ reveals erroneous collision response force *toward* the other plane, and the whole $\vec{f}$ can be scaled. Otherwise, if the magnitude is negative, the input force is decomposed as in equation (8.6) and equation (8.7) and scaled and composed as in equation (8.8). For efficiency, the case of zero magnitude along $\vec{n}$ is included with the positive magnitudes since it results in a simpler, but ultimately equivalent, computation. Overall, the resulting force is computed by the piece-wise function

$$\vec{f}' = \begin{cases} (1 - \alpha) \cdot \vec{f} & \vec{f} \cdot \vec{n} \geq 0 \\ \vec{f}_{||} + (1 - \alpha)\vec{f}_{\perp} & \vec{f} \cdot \vec{n} < 0 \end{cases} \tag{8.9}$$

where the component vectors $\vec{f}_{||}$ and $\vec{f}_{\perp}$ are computed as in equation (8.6) and equation (8.7) respectively.

Response to other constraints

By combining elements of the two above responses, appropriate responses to the remaining constraints under consideration can be implemented. This is made substantially easier by the fact that DoF-restoration is performed on forces at a per-voxel level; torques are determined based on the forces, and thus there are only three degrees of freedom for each voxel's response to consider. For instance, the linear annular constraint of a sphere in a cylinder restores the same degrees of freedom as a cylindrical constraint: the additional rotational degree of freedom is already handled by the along-axis translational freedom restoration.

8.5. Results

The described algorithm has been implemented in the SPARTA virtual assembly application. The hybrid method implementation builds on the existing VPS-based virtual assembly functionality in SPARTA. Bodies with associated b-rep data are a subtype of standard VPS bodies. When adding a body to the simulation, the collision link creation methods determine if both bodies contain b-rep data, in which case a specific hybrid method collision link is created that performs the collision detection and response as described in this paper. This technique allows combining bodies with b-rep data and presumably a need for accurate low-clearance assembly in the same simulation as bodies without this data, transparently falling back to the pure tessellated geometry algorithm in cases where a b-rep-containing body collides with a standard body.

To permit modification of response forces at a per-voxel level, effectively splitting collision detection and collision response, the VPS software in use was modified and partially refactored. The existing `VpsPbmCollide` method was split into a `VpsPbmCollideX` advanced API method that returns a container with collision pairs and a method to transform and sum the forces listed in those collision pairs to arrive at overall response forces and torques. The `VpsPbmCollide` API used previously was retained as a wrapper around these two refactored procedures for applications that do not require the force modification that the hybrid method requires—essentially in the case of using the tessellated geometry algorithm. In the hybrid collision link, the two procedures are called individually, with the constraint detection and degree-of-freedom restoration as described here occurring in between.

8.5.1. Results

The improvements described here result in a hybrid collision detection and response method that can maintain required high rates for haptic interaction. The simulation was instrumented to count the number of overruns: occasions when greater than 1.5 ms (1.5 times the time step) passed between simulation frames. The simulation results in a few of these in normal operation of the tessellated geometry collision detection method without negative consequences. Tests were performed on a workstation with a quad-core 3.2 GHz Intel Xeon processor, 6 GB of RAM, an NVIDIA Quadro FX 5800 graphics card, a Sensable Phantom Omni haptic device, and Windows 7 64-bit edition. Even though the Omni only provides 3-DoF haptic feedback, full 6-DoF forces and torques were computed during the test. In a representative trial manipulating and inserting a pin through a block with a through-hole, a total of 45,833 frames were simulated, corresponding to a run-time of 45.8 s. In 11,053 of those frames, a collision was detected and responded to. With the hybrid method active, only 57 overruns were recorded, roughly 0.5% of collision frames and 0.1% of total simulation frames. This indicates that the simulation kept up at the 1000 Hz rate effectively continuously throughout the manipulation. This represents a substantial improvement over [3], which reported large, consistent slowdowns below 500 Hz and frequently as low as 100 Hz during insertion. The haptic insertion with the solution presented here is qualitatively smooth and free of unnatural force artifacts when tested using the Omni.

Furthermore, initial validation was performed on the suitability of the algorithm for improving the ability to assemble low-clearance objects. A 4.5 mm voxel size was used with a cylindrical hole and pin with radius 75 mm and 70 mm respectively. The block with the hole was 200 mm thick, and the pin was 400 mm long. Input from the Omni was scaled by 4 to provide access to a larger workspace. With the hybrid method disabled, it was very difficult to align the pin, insert it in the hole, and move it through the block. Any mis-alignment was likely to result in the pin being jostled back out of the hole, providing unnatural resistance to assembly. In a recorded demonstration, an experienced user was only able to pass the pin through the hole 3 times in 28 s. After turning on the hybrid method in the same setup, the pin passed through the hole 5 times in 10 s, with little unnatural resistance.

8.6. Conclusion

Physically-based modeling for haptic virtual assembly requires high-speed collision detection and response. Existing solutions result in incorrectly limiting degrees of freedom because of geometric tesselation, operate too slowly to support haptic feedback, or require manual intervention in a preprocessing step. A hybrid collision detection and response algorithm first presented in [3] is promising. We broke down the hybrid method presented in that work into five research tasks: voxel to b-rep initial association, voxel to b-rep look-up, automatic constraint recognition, constraint alignment measure, and application of constraint knowledge. Instead of solving constraints, then generating and blending an artificial "constraint force" with the tessellated geometry algorithm results, we investigated the direct, local restoration of degrees of freedom. Since the difficulty with tessellated geometry algorithms is that they result in local forces that are artifacts of the tessellation and artificially limit degrees of freedom expected in the original model, we use the constraint knowledge computed within the overall hybrid method to locally modify collision response forces. We scale down components of these forces that are acting counter to a known degree of freedom. The described solution has been implemented in a virtual assembly software system using a modified version of VPS, and functions to provide expected movement in low-clearance assembly simulations. Its performance greatly exceeds that of earlier related work, maintaining a haptic rate of 1000 Hz, while improving the ability to smoothly assemble low-clearance models.

Acknowledgment

This work was performed at the Virtual Reality Applications Center at Iowa State University. This material is based upon work supported by the Joseph C. and Elizabeth A. Anderlik Professorship at Iowa State University, and by the National Science Foundation under Grant No. CMMI-0928774. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Bibliography

[1] Ram Anantha, Glenn A Kramer, and Richard H Crawford. Assembly modelling by geometric constraint satisfaction. *Computer-Aided Design*, 28(9):707–722, September 1996. ISSN 00104485. doi:10.1016/0010-4485(96)00001-2. URL http://linkinghub.elsevier.com/retrieve/pii/0010448596000012.

[2] Daniela Faas. *A hybrid method for haptic feedback to support manual virtual product assembly*. Ph.d. dissertation, Iowa State University, 2010. URL http://lib.dr.iastate.edu/etd/11480/.

[3] Daniela Faas. A Hybrid Method for Haptic Feedback to Support Manual Virtual Product Assembly. In *ASME 2011 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, pages 1–8, Washington,

DC, USA, 2011. ASME. doi:10.1115/DETC2011-47665. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1639771.

[4] Daniela Faas and Judy M. Vance. Assessment of Pointshell Shrinking and Feature Size on Virtual Manual Assembly. In *ASME 2010 World Conference on Innovative Virtual Reality (WINVR 2010)*, pages 211–218. ASME, 2010. ISBN 978-0-7918-4908-8. doi:10.1115/WINVR2010-3765. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1617279.

[5] Daniela Faas and Judy M. Vance. BREP Identification During Voxel-Based Collision Detection for Haptic Manual Assembly. In *ASME 2011 World Conference on Innovative Virtual Reality (WINVR 2011)*, pages 145–153, Milan, Italy, 2011. ASME. ISBN 978-0-7918-4432-8. doi:10.1115/WINVR2011-5524. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1623634.

[6] Terrence Fernando, Prasad Wimalaratne, and Kevin Tan. Constraint-based virtual environment for supporting assembly and maintainability tasks. In *ASME 1999 Design Engineering Technical Conferences & Computers in Engineering Conference*, pages 11–15, 1999.

[7] Sankar Jayaram, Uma Jayaram, Yong Wang, Hrishikesh Tirumali, Kevin Lyons, and Peter Hart. VADE: a Virtual Assembly Design Environment. *IEEE Computer Graphics and Applications*, 19(6):44–50, 1999. ISSN 02721716. doi:10.1109/38.799739. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=799739.

[8] Zhenyu Liu and Jianrong Tan. Constrained behavior manipulation for interactive assembly in a virtual environment. *The International Journal of Advanced Manufacturing Technology*, 32(7-8):797–810, March 2006. ISSN 0268-3768. doi:10.1007/s00170-005-0382-5. URL http://link.springer.com/10.1007/s00170-005-0382-5.

[9] William A. McNeely, Kevin D. Puterbaugh, and James J. Troy. Six degree-of-freedom haptic rendering using voxel sampling. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques - SIGGRAPH '99*, pages 401–408, New York, New York, USA, 1999. ACM Press. ISBN 0201485605. doi:10.1145/311535.311600. URL http://portal.acm.org/citation.cfm?doid=311535.311600.

[10] William A. McNeely, Kevin D. Puterbaugh, and James J. Troy. Voxel-based 6-dof haptic rendering improvements. *Haptics-e*, 3(7), 2006. URL https://haptics-e.org/Vol_03/he-v3n7.pdf.

[11] Ryan A. Pavlik and Judy M. Vance. VR JuggLua: A Framework for VR Applications Combining Lua, OpenSceneGraph, and VR Juggler. In *Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS) in IEEE Virtual Reality*, Singapore, 2011. IEEE. doi:10.1109/SEARIS.2012.6231166. URL http://dx.doi.org/10.1109/SEARIS.2012.6231166.

[12] Abhishek Seth. *Combining physical constraints with geometric constraint-based modeling for virtual assembly*. Masters thesis, Iowa State University, Ann Arbor, 2007. URL http://search.proquest.com/docview/304857108?accountid=10906.

[13] Abhishek Seth, Judy M. Vance, and James H. Oliver. Combining Geometric Constraints With Physics Modeling for Virtual Assembly Using SHARP. In *ASME 2007 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages 1045–1055. ASME, 2007. ISBN 0-7918-4803-5. doi:10.1115/DETC2007-34681. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1604120.

[14] Abhishek Seth, Judy M. Vance, and James H. Oliver. Virtual reality for assembly methods prototyping: a review. *Virtual Reality*, 15(1):5–20, January 2010. ISSN 1359-4338. doi:10.1007/s10055-009-0153-y. URL http://www.springerlink.com/index/10.1007/s10055-009-0153-y.

[15] Loïc Tching, Georges Dumont, and Jérôme Perret. Interactive simulation of CAD models assemblies using virtual constraint guidance. *International Journal on Interactive Design and Manufacturing (IJIDeM)*, 4(2):95–102, April 2010. ISSN 1955-2513. doi:10.1007/s12008-010-0091-7. URL http://www.springerlink.com/index/10.1007/s12008-010-0091-7.

[16] Judy M. Vance and Georges Dumont. A Conceptual Framework to Support Natural Interaction for Virtual Assembly Tasks. In *ASME 2011 World Conference on Innovative Virtual Reality (WINVR 2011)*, pages 273–278, Milan, Italy, 2011. ASME. ISBN 978-0-7918-4432-8. doi:10.1115/WINVR2011-5570. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1623669.

[17] Run Dang Yang, XiuMin Fan, DianLiang Wu, and JuanQi Yan. Virtual assembly technologies based on constraint and DOF analysis. *Robotics and Computer-Integrated Manufacturing*, 23(4):447–456, August 2007. ISSN 07365845. doi:10.1016/j.rcim.2006.05.008. URL http://linkinghub.elsevier.com/retrieve/pii/S0736584506000755.

[18] Gabriel Zachmann and Alexander Rettig. Natural and Robust Interaction in Virtual Assembly Simulation. In *Proceedings of the 8th ISPE International Conference on Concurrent Engineering*, Anaheim, California, 2001.

CHAPTER 9. GENERAL CONCLUSION

The work included in this dissertation has spanned a range of topics related to enabling natural interaction for virtual reality. In Chapter 2, the "missing link" software between low-cost, readily-available hardware for head tracking and standard commercial and academic virtual reality applications was presented. The software discussed in that chapter has been integrated into the upstream VRPN software system for several years and is now widely distributed. It works toward natural interaction for virtual reality by enabling virtual reality on an individual scale, with head tracking experiences at every desk rather than just in VR-dedicated spaces.

VR JuggLua, presented in Chapter 3, contains a number of contributions. It permits virtual reality application development to take place in Lua, a very high-level language that is easy to learn, while not sacrificing the broad system support of VR Juggler [1]. It provides an embedded domain-specific language for creation of the most common scene graph structures, while not limiting access to the full power of OpenSceneGraph. In the run buffer structure, the interactivity of live code execution in a REPL is restored to the event loop environment of a running virtual reality application. Through the careful synchronization algorithm of this structure, the clustering capabilities of VR Juggler are amplified by enabling a live REPL with frame-consistent state across a many-node cluster. The development of the coroutine-based "frame action" idiom in the VR JuggLua provides both novice and experienced developers with a powerful tool for creating interactivity in virtual environments through an apparent inversion of control. These facilities have been used both by researchers working in the same team as myself as well as students in a graduate virtual reality class and outside researchers.

The SPARTA (Scriptable Platform for Advanced Research and Teaching in Assembly) virtual assembly software, the backdrop and implementation framework for the following chapters, builds on VR JuggLua by adding a powerful physically-based modeling simulation engine tuned for haptic interaction. SPARTA effectively provides a simulation core, as well as a domain-specific language for specifying virtual assembly simulations. It contains essentially a super-set of the functionality of VR JuggLua, which has permitted the rapid exploration of a number of techniques for natural interaction. SPARTA replaced existing legacy virtual assembly software (SHARP/SHARP-2 [7]) in our research group with a more flexible, easier-to-maintain system well-suited as a platform for further research.

In Chapter 4 and Chapter 5, the problem addressed is the limited workspace of haptic devices in virtual assembly simulations. The "bubble technique" is a recently-developed position-rate control method for virtually expanding the haptic workspace [3]. Chapter 4 contributes observations and analysis of problems arising from the interaction of virtual assembly simulation and the bubble technique. It also presents a modification of the bubble technique specifically designed to counter these problems. In Chapter 5, the workspace of a large haptic device is made physically even larger by a hardware-software-controls integration. The combination of an omni-directional mobile robot base, a commercial haptic interaction device, a velocity control algorithm for the mobile base, and the seamless integration of these elements into a virtual assembly application (SPARTA) as effectively a single very-large-workspace haptic device is the novel advancement contributed in that work.

Chapters 6, 7, and 8 present contributions based on work advancing the state of the art in low-clearance haptic virtual assembly. They build conceptually on initial work [4, 6, 5] toward a hybrid method combining an tessellated geometry collision detection and response library, VPS, with knowledge of original precise b-reps, constraint recognition, and constraint solving to enable low-clearance assembly of CAD models with haptic force-feedback. Chapter 6 contributes layers of de-duplication and redundant computation elimination to the general hybrid method. It contributes a data structure, complete with C++ implementation, based on the principle of "hash-consing." The data structure stores only one copy of each unique value and returns logical references to the values as integer look-up keys with constant-time retrieval. The use of this data structure dovetails with the application of memoization, a technique for caching the results of pure functions for given inputs so that the computation on an input value is only performed once. A basic, naive implementation of the hybrid method given just previous work and the general algorithm descriptions that open this chapter may be possible. However, it is the optimizations contributed here that greatly bring down the computational load of implementing the algorithm, enabling higher performance and greater scalability.

In Chapter 7, a novel design for one of the components of the hybrid method is presented. Given b-reps from two bodies that are known to be colliding, the method needs to know if a geometric constraint arises from those b-reps that might be used to improve the quality of interaction. Such an automatic constraint recognition engine is presented in this chapter. The novel design avoids explicit coding of nearly all conditional statements by the researcher-developer through application of type-driven C++ metaprogramming techniques. The constraint recognition problem is modeled as a multi-method, with commutativity. Multiple dispatch is achieved directly and without boilerplate code or polymorphic inheritance through use of the `boost::variant` discriminated union class template and its type-safe visitor facilities. Recognition of new pairs can be added in what resembles a declarative (rather than imperative or procedural) style, through explicit template specialization. A metaprogramming layer applying a compile-time conditional construct automatically handles commutativity of arguments and converts researcher-developer errors such as duplicated recognizers into compile-time errors, helping avoid run-time logic errors. The zero-overhead abstractions applied result in maintainable software that the compiler transforms into an efficient recognition engine with internal workings inlined into the main procedure.

Finally, Chapter 8 takes a second look at the procedures of the hybrid method and decomposes them into a number of distinct research tasks. A new solution for one of them, the automatic constraint recognition problem, was already presented in Chapter 7. The bulk of Chapter 8 focuses on the "application of constraint knowledge" task. The chapter presents a novel approach to applying constraint knowledge acquired during the hybrid method, focusing on degrees of freedom at a local level rather than a constraint system at a more global (per-part-pair) level. The fundamental insight behind this alternate approach is that the motivation for incorporating constraint knowledge into the collision detection and response algorithm is that the purely tessellated geometry algorithms artificially remove degrees of freedom that are captured in the constraint knowledge computed. The steps leading up to application of constraint knowledge inform the algorithm about geometric constraints, and their corresponding degrees of freedom, expected from the simulation. Then, at a local (per-voxel or similar discrete collision unit) level, the initial penalty response forces computed can be modified in the presence of a constraint, by selective reduction or removal of "erroneous" response forces acting counter to a known degree of freedom. This local degree-of-freedom restoration avoids the problems with constraint solving highlighted in this chapter, as well as the sizable performance impacts of the constraint solver. It serves as a parallel to the per-voxel tessellated geometry collision detection and response algorithm that produces an overall physically-modeled response, as a per-voxel correction based on local constraint knowledge that overall improves the quality of haptic interaction.

Overall, the works in this dissertation represent distinct contribution to the state of the art individually. Taken as a whole, they form a substantial pattern of contributions toward enabling natural interaction for virtual reality. The low-cost virtual reality work highlights the changes and opportunities available in the field as technology advances and economies of scale bring hardware supporting virtual reality into lower price ranges, more workplaces, and our homes. The virtual reality software engineering work reveals the benefits of bringing the tools from other areas (domain-specific languages, REPL environments, scripting languages) to bear on virtual reality development, as well as the potential for additional system-level advancements in virtual reality frameworks. The chapters addressing the haptic workspace highlight that not all haptic techniques are applicable directly to virtual assembly, but novel combinations and integrations can result in improved reachable workspace for any haptic application. The work on the hybrid method highlights the depth of the low-clearance assembly problem, as well as the diversity of full solutions that could be conceived to implement "the hybrid method."

9.1. Recommended Directions for Future Research

The VR JuggLua software is quite complete in its present state, but additional development and research is possible. Its extensive OpenSceneGraph binding coverage relies on a C++ introspection library based on dynamically-loaded modules compiled from code generated by header-parsing software, which only needs to be run by a developer when the OpenSceneGraph library API changes. This introspection wrapper generation software, which uses Doxygen's XML output mode to analyze classes, is complex and resource-intensive, requires an extensive configuration file to work around flaws in Doxygen and limitations in the wrapper generator, and has not been fully updated to work with versions of OpenSceneGraph released after it was removed from the general OpenSceneGraph distribution after the 2.8 stable version series. The approach that seems best at this time is to replicate the functionality of the wrapper generator in a new piece of software based on the Clang compiler front-end libraries [2]. By using the actual compiler code as a library, a Clang-based wrapper generator can be assured of accurate class information for use in producing introspection wrappers. It would reduce the amount of code and configuration needed in the actual generator significantly. In the long term, that introspection library itself might be replaced by one based on embedding components of Clang and LLVM. Additionally, explicit user studies on the usability of the VR JuggLua system, and further developments improving the functionality of the integrated code console would be valued contributions.

The modifications to the bubble technique merit further evaluation. Anecdotal evidence suggests differing effectiveness of the modifications depending on the device used and its native workspace. A gain-tuning procedure for the rate-control region of the original or modified bubble technique would also be a valuable development.

The omni-directional mobile base for a haptic device would benefit from a characterization of its run-time capabilities and the effects, if any, that it has on the quality of force display. The addition of rotational control to keep the haptic device out of the line of sight of the user would be valuable. The control software layer would benefit from the addition of physical boundary awareness, for more transparent use in facilities such as CAVEs that have fixed walls.

There is still substantial research-worthy material in the general hybrid collision detection and response technique. Chapter 8 divides the method into five tasks. In addition to the more general contributions in those chapters, Chapter 6, 7, and 8 each present a new solution to one of the tasks, leaving two tasks at the previous level of research. They are the initial voxel to b-rep association and the constraint alignment measure . The previous solution to the initial voxel to b-rep association task, based on the use of intersection detection between b-reps in the proprietary geometric modeling kernel D-Cubed, did work conceptually, but the software library brought to bear on the problem results in unnecessary overhead and challenges for other researchers attempting to reproduce the research. The underlying operation, determining which subset of b-reps pass through a cube of given size and location, seems likely to have a simpler, faster implementation possible. The previous solution to the constraint alignment measure problem, on the other hand, is less satisfactory. It depends on the distance and angle from full alignment, in addition to the voxel size and a "penetration depth" of unknown derivation. The fundamental weakness here is that a few degrees of misalignment for a cylindrical pin-in-hole conceptually suggests a very different alignment factor than the same number of degrees of misalignment between two planes. An alternate measure of constraint alignment unique to each constraint type would be a valuable addition. A more quantitative evaluation of the performance of a newer instance of the hybrid method would also merit research attention.

Finally, developments in virtual reality hardware and software systems and usability reminds us that there is a history, and a future, of more hobbyist and consumer-grade virtual reality capable hardware in addition to academic and industrial progress. It is in the best interest of academic VR research to neither desert their existing work for each new release, nor to ignore the exciting developments. Instead, researchers should remain engaged with these communities, pointing out the lessons already learned in academia about virtual environments, and learning the new lessons discovered by broader audiences. There is much future work in building new generations of the "missing link" systems connecting new hardware to the substantial base of existing knowledge and software that is available in more conventional virtual reality, as well as improving the usability of conventional VR hardware and software for developers and end-users.

Bibliography

[1] Allen Bierbaum, Christopher Just, Patrick Hartling, Kevin Meinert, Albert Baker, and Carolina Cruz-Neira. VR Juggler: a virtual platform for virtual reality application development. In *IEEE Virtual Reality Conference (VR 2001)*, pages 89–96, Los Alamitos, CA, USA, 2001. IEEE Comput. Soc. ISBN 0-7695-0948-7. doi:10.1109/VR.2001.913774. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=913774.

[2] Clang and LLVM Developers. Clang: a C language family frontend for LLVM. URL http://clang.llvm.org/.

[3] Lionel Dominjon, Jérôme Perret, and Anatole Lécuyer. Novel devices and interaction techniques for human-scale haptics. *The Visual Computer*, 23(4):257–266, March 2007. ISSN 0178-2789. doi:10.1007/s00371-007-0100-4. URL http://link.springer.com/10.1007/s00371-007-0100-4.

[4] Daniela Faas. *A hybrid method for haptic feedback to support manual virtual product assembly*. Ph.d. dissertation, Iowa State University, 2010. URL http://lib.dr.iastate.edu/etd/11480/.

[5] Daniela Faas. A Hybrid Method for Haptic Feedback to Support Manual Virtual Product Assembly. In *ASME 2011 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, pages 1–8, Washington, DC, USA, 2011. ASME. doi:10.1115/DETC2011-47665. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1639771.

[6] Daniela Faas and Judy M. Vance. BREP Identification During Voxel-Based Collision Detection for Haptic Manual Assembly. In *ASME 2011 World Conference on Innovative Virtual Reality (WINVR 2011)*, pages 145–153, Milan, Italy, 2011. ASME. ISBN 978-0-7918-4432-8. doi:10.1115/WINVR2011-5524. URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1623634.

[7] Abhishek Seth, Hai-Jun Su, and Judy M. Vance. Development of a Dual-Handed Haptic Assembly System: SHARP. *Journal of Computing and Information Science in Engineering*, 8(4):044502, 2008. ISSN 15309827. doi:10.1115/1.3006306. URL http://computingengineering.asmedigitalcollection.asme.org/article.aspx?articleid=1401373.