with additional C routines; the C language is more flexible and powerful than any higher level geometric scripting language we could design ourselves.

10 Results

We have constructed a placement editor for real-time interactive walkthrough of large building databases. One of our primary goals was to work with off-the-shelf input and display hardware, a goal which required the use of a software framework to allow the user to perform unambiguous 3D manipulation with 2D devices.

Our solution is based on *object associations*, a framework that provides the flexibility to combine pseudo-physical properties with convenient teleological behavior in a mixture tailor-made for a particular application domain or a special set of tasks. We have found that such a mixture of the "magical" capabilities of geometric editing systems with some partial simulations of real, physical behavior makes a very attractive and easy-to-use editing system for 3D virtual environments. The combination of goal-oriented alignments, such as snap-dragging, with application specific physical behavior, such as gravity and solidity, reduce the degrees of freedom the user has to deal with explicitly while maintaining most of the convenience of a good geometrical drafting program.

We found it to be practical to separate into two types of procedures the mapping of 2D pointing to 3D motion and the enforcement of the desired object placement behavior. These procedures are clearly defined and easy to implement as small add-on functions in C. Geometric and database toolkits allow high-level coding and ease of modification. Our object associations normally cause little computational overhead to the WALKTHRU system. This is an important concern, since keeping the response time of the system fast and interactive is a crucial aspect of its usability and user-friendliness [5].

The result is a technique that makes object placement quick and accurate, works with "drag-and-drop" as well as "cut and paste" interaction techniques, can provide desirable local object behavior and an automated grouping facility, and greatly reduces the need for multiple editing modes in the user interface. The resulting environment is devoid of fancy widgets, sophisticated measuring bars, or multiple view windows. To the novice user it seem that not much is happening – objects simply follow the mouse to reasonable, realistic locations. And that is how ideally it should be: any additional gimmick is an indication that the paradigm has not yet been pushed to its full potential. Some issues remain to be fully resolved, such as dealing with association loops, but our prototype demonstrates that this approach provides a simple, flexible, and practical approach to constructing easy-to-use 3D manipulation interfaces.

A prototype implementation in the context of a model of a building with more than 100 rooms has proven to be attractive and has reduced by a large factor the tedium of placing furniture and wall decorations. One of the authors has constructed scenes of rather cluttered offices with many pieces of furniture, fully loaded with books, pencils, coffee cups, etc. in five to ten minutes (see Figure C2). The implementation in our specific WALKEDIT application domain required only 5 programmer-defined procedures to fully characterize most of the desired object behavior.

References

[1] Baraff, D. Fast Contact Force Computation for Non-penetrating Rigid Bodies. *Proc. of SIGGRAPH '94* (Orlando, FL, Jul. 1994), pp. 23-34.

[2] Barlow, M. Of Mice and 3D Input Devices. *Computer-Aided Engineering* 12, 4 (Apr. 1993), pp. 54-56.

[3] Bier, E.A. Snap-Dragging in Three Dimensions. *Proc. of the 1990 Symposium on Interactive 3D Graphics* (Snowbird, UT, Mar. 1990), pp. 193-204.

[4] Borning, A. The Programming Aspects of Thinglab, a Constraint-Oriented Simulation Laboratory. *ACM Trans. on Programming Languages and Systems* 3, 4, pp. 353-387.

[5] Funkhouser, T.A. and Séquin, C.H. Adaptive Display Algorithm for Interactive Frame Rates during Visualization of Complex Virtual Environments. *Proc. of SIGGRAPH '93* (Anaheim, CA, Aug. 1993), pp. 247-254.

[6] Gleicher, M. Briar: A Constraint-Based Drawing Program. *Proc. of the ACM Conference on Human Factors in Computing Systems – CHI '92* (Monterey, CA, May 1992), pp. 661-662.

[7] Hahn, J.K. Realistic Animation of Rigid Bodies. *Computer Graphics* 22, 4 (Aug. 1988), pp. 299-208.

[8] Helm, R., Huynh, T., Lassez, C., and Marriott, K. Linear Constraint Technology for Interactive Graphic Systems. *Proc. of Graphics Interface '92* (Vancouver, BC, Canada, May 1992).

[9] Lin, M.C. and Canny, J.F. A fast algorithm for incremental distance calculation. *International Conference on Robotics and Automation*, IEEE (May 1991), pp. 1008-1014.

[10] Myers, B.A. Creating User Interfaces using Programming by Example, Visual Programming, and Constraints. *ACM Trans. on Programming Languages and Systems*, 12, 2 (Apr. 1990), pp. 143-177.

[11] Nelson, G. Juno, a Constraint-Based Graphics System. *Proc. of SIGGRAPH '85* (San Fransisco, CA, Jul. 22-26, 1985). In *Computer Graphics* 19, 3 (Jul. 1985), pp. 235-243.

[12] Nielson, G. and Olsen, D. Direct Manipulation Techniques for 3D Objects Using 2D Locator Devices. *Proc. of the 1986 Workshop on Interactive 3-D Graphics* (Chapel Hill, NC, Oct. 1986), pp. 175-182.

[13] Smith, R.B. Experiences with the Alternate Reality Kit: An Example of the Tension between Literalism and Magic. *IEEE Computer Graphics and Applications* 7, 9 (Sep. 1987), pp. 42-50.

[14] Teller, S.J., and Séquin, C.H. Visibility Preprocessing for Interactive Walkthroughs. *Proc. of SIGGRAPH '91* (Las Vegas, Nevada, Jul. 28-Aug. 2, 1991). In *Computer Graphics*, 25, 4 (Jul. 1991), pp. 61-69.

[15] Venolia, D. Facile 3D Direct Manipulation. *Proc. of the ACM Conference on Human Factors in Computing Systems – CHI 93* (Amsterdam, Netherlands, Apr. 1993), pp. 31-36.

BUNGIE - EXHIBIT 1006 - PART 10 OF 14

CamDroid: A System for Implementing Intelligent Camera Control

Steven M. Drucker
MIT Media Lab

David Zeltzer
MIT Research Laboratory for Electronics

Massachusetts Institute of Technology
Cambridge, MA. 02139, USA
smd@media.mit.edu
dz@vetrec.mit.edu

Abstract

In this paper, a method of encapsulating camera tasks into well defined units called "camera modules" is described. Through this encapsulation, camera modules can be programmed and sequenced, and thus can be used as the underlying framework for controlling the virtual camera in widely disparate types of graphical environments. Two examples of the camera framework are shown: an agent which can film a conversation between two virtual actors and a visual programming language for filming a virtual football game.

Keywords: Virtual Environments, Camera Control, Task Level Interfaces.

1. Introduction

Manipulating the viewpoint, or a synthetic camera, is fundamental to any interface which must deal with a three dimensional graphical environment, and a number of articles have discussed various aspects of the camera control problem in detail [3, 4, 5, 19]. Much of this work, however, has focused on techniques for directly manipulating the camera.

In our view, this is the source of much of the difficulty. Direct control of the six degrees of freedom (DOFs) of the camera (or more, if field of view is included) is often problematic and forces the human VE participant to attend to the interface and its "control knobs" in addition to — or instead of — the goals and constraints of the task at hand. In order to achieve *task level* interaction with a computer-mediated graphical environment, these low-level, direct controls, must be abstracted into higher level camera primitives, and in turn, combined into even higher level interfaces. By clearly specifying what specific tasks need to be accomplished at a particular unit of time, a wide variety of interfaces can be easily constructed. This technique has already been successfully applied to interactions within a Virtual Museum [8].

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
1995 Symposium on Interactive 3D Graphics, Monterey CA USA
© 1995 ACM 0-89791-736-7/95/0004...$3.50

2. Related Work

Ware and Osborne [19] described several different metaphors for exploring 3D environments including "scene in hand," "eyeball in hand," and "flying vehicle control" metaphors. All of these use a 6 DOF input device to control the camera position in the virtual environment. They discovered that flying vehicle control was more useful when dealing with enclosed spaces, and the "scene in hand" metaphor was useful in looking at a single object. Any of these metaphors can be easily implemented in our system.

Mackinlay et al [16] describe techniques for scaling camera motion when moving through virtual spaces, so that, for example, users can always maintain precise control of the camera when approaching objects of interest. Again, it is possible to implement these techniques using our camera modules.

Brooks [3,4] discusses several methods for using instrumented mechanical devices such as stationary bicycles and treadmills to enable human VE participants to move through virtual worlds using natural body motions and gestures. Work at Chapel Hill, has, of course, focused for some time on the architectural "walk-through," and one can argue that such direct manipulation devices make good sense for this application. While the same may be said for the virtual museum, it is easy to think of circumstances — such as reviewing a list of paintings — in which it is not appropriate to require the human participant to physically walk or ride a bicycle. At times, one may wish to interact with topological or temporal abstractions, rather than the spatial. Nevertheless, our camera modules will accept data from arbitrary input devices as appropriate.

Blinn [2] suggested several modes of camera specification based on a description of what should be placed in the frame rather than just describing where the camera should be and where it should be aimed.

Phillips et al suggest some methods for automatic viewing control [18]. They primarily use the "camera in hand" metaphor for viewing human figures in the Jack™ system, and provide automatic features for maintaining smooth visual transitions and avoiding viewing obstructions. They do not deal with the problems of navigation, exploration or presentation.

Karp and Feiner describe a system for generating automatic presentations, but they do not consider interactive control of the camera [12].

Gleicher and Witkin [10] describe a system for controlling the movement of a camera based on the screen-space projection of an object, but their system works primarily for manipulation tasks.

Our own prior work attempted to establish a procedural framework for controlling cameras [7]. Problems in constructing generalizable procedures led to the current, constraint-based framework described here. Although this paper does not concentrate on methods for satisfying multiple constraints on the camera position, this is an important part of the overall camera framework we outline here. For a more complete reference, see [9]. An earlier form of the current system was applied to the domain of a Virtual Museum [8].

3. CamDroid System Design

This framework is a formal specification for many different types of camera control. The central notion of this framework is that camera placement and movement is usually done for particular reasons, and that those reasons can be expressed formally as a number of primitives or constraints on the camera parameters. We can identity these constraints based on analyses of the tasks required in the specific job at hand. By analyzing a wide enough variety of tasks, a large base of primitives can be easily drawn upon to be incorporated into a particular task-specific interface.

3.1 Camera Modules

A camera module represents an encapsulation of the constraints and a transformation of specific user controls over the duration that a specific module is active. A complete network of camera modules with branching conditions between modules incorporates user control, constraints, and response to changing conditions in the environment over time.

Our concept of a camera module is similar to the concept of a *shot* in cinematography. A shot represents the portion of time between the starting and stopping of filming a particular scene. Therefore a shot represents continuity of all the camera parameters over that period of time. The unit of a single camera module requires an additional level of continuity, that of continuity of *control* of the camera. This requirement is added because of the ability in computer graphics to identically match the camera parameters on either side of a cut, blurring the distinction of what makes up two separate shots. Imagine that the camera is initially pointing at character A and following him as he moves around the environment. The camera then pans to character B and follows her for a period of time. Finally the camera pans back to character A. In cinematic terms, this would be a single shot since there was continuity in the camera parameters over the entire period. In our terms, this would be broken down into four separate modules. The first module's task is to follow character A. The second module's task would be to pan from A to B in a specified amount of time. The third module's task would be to follow B. And finally the last module's task would be to pan back from B to A. The notion of breaking this cinematic shot into 4 modules does not specify implementation, but rather a formal description of the goals or constraints on the camera for each period of time.

As shown in figure 1, the generic module contains the following components:

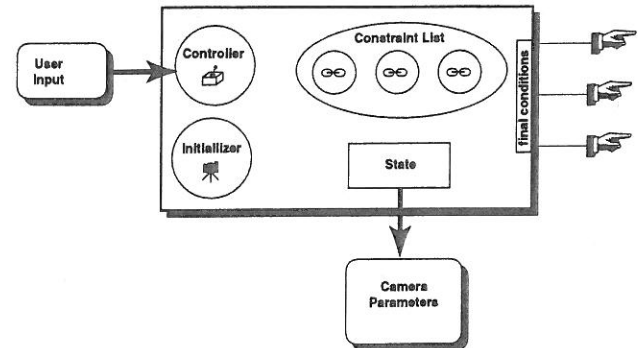

Figure 1: Generic camera module containing a controller, an initializer, a constraint list, and local state

- the local state vector. This must always contain the camera position, camera view normal, camera "up" vector, and field of view. State can also contain values for the camera parameter derivatives, a value for time, or other local information specific to the operation of that module. While the module is active, the state's camera parameters are output to the renderer.
- initializer. This is a routine that is run upon activation of a module. Typical initial conditions are to set up the camera state based on a previous module's state.
- controller. This component translates user inputs either directly into the camera state or into constraints. There can be at most one controller per module.
- constraints to be satisfied during the time period that the module is active. Some examples of constraints are as follows:
  - maintain the camera's up vector to align with world up.
  - maintain height relative to the ground
  - maintain the camera's gaze (i.e. view normal) toward a specified object
  - make sure a certain object appears on the screen.
  - make sure that several objects appear on the screen
  - zoom in as much as possible

In this system, the constraint list can be viewed simply as a black box that produces values for some DOFs of the camera. The constraint solver combines these constraints using a constrained optimizing solver to come up with the final camera parameters for a particular module. The camera optimizer is discussed extensively in [9]. Some constraints directly produce values for a degree of freedom, for example, specifying the up vector for the camera or the height of the camera. Some involve calculations that might produce multiple DOFs, such as adjusting the view normal of the camera to look at a particular object. Some, like a path planning constraint discussed in [8] are quite complicated, and generate a series of DOFs over time through the environment based on an initial and final position.

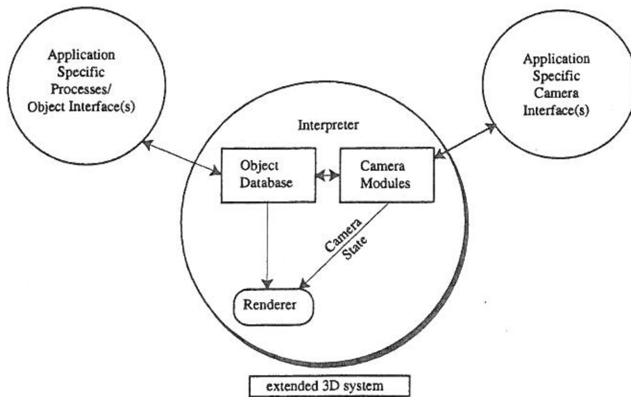

Figure 2: Overall CamDroid System

3.2 The CamDroid System

The overall system for the examples given in this paper is shown in figure 2.

The CamDroid System is an extension to the 3D virtual environment software testbed developed at MIT [6]. The system is structured this way to emphasize the division between the virtual environment database, the camera framework, and the interface that provides access to both. The CamDroid system contains the following elements.

- A general interpreter that can run pre-specified scripts or manage user input. The interpreter is an important part in developing the entire runtime system. Currently the interpreter used is TCL with the interface widgets created with TK [17]. Many commands have been embedded in the system including the ability to do dynamic simulation, visibility calculations, finite element simulation, matrix computations, and various database inquiries. By using an embedded interpreter we can do rapid prototyping of a virtual environment without sacrificing too much performance since a great deal of the system can still be written in a low level language like C. The addition of TK provides convenient creation of interface widgets and interprocess communication. This is especially important because some processes might need to perform computation intensive parts of the algorithms; they can be offloaded onto separate machines.
- A built-in renderer. This subsystem can use either the hardware of a graphics workstation (currently SGIs and HPs are supported), or software to create a high quality antialiased image.
- An object database for a particular environment.
- Camera modules. Described in the previous section. Essentially, they encapsulate the behavior of the camera for different styles of interaction. They are prespecified by the user and associated with various interface widgets. Several widgets can be connected to several camera modules. The currently active camera module handles all user inputs and attempts to satisfy all the constraints contained within the module, in order to compute camera parameters which will be passed to the renderer when creating the final image. Currently, only one camera module is active at any one time, though if there were multiple viewports, each of them could be assigned a unique camera.

4. Example: Filming a conversation

The interface for the conversation filming example is based on the construction of a software agent which perceives changes in limited aspects of the environments and uses a number of primitives to implement agent behaviors. The sensors detect movements of objects within the environment and can perceive which character is designated to be talking at any moment.

In general, the position of the camera should be based on conventional techniques that have been established in filming a conversation. Several books have dissected conversations and come up with simplified rules for an effective presentation [1, 14]. The conversation filmer encapsulates these rules into camera modules which the software agent calls upon to construct (or assist a director in the construction of) a film sequence.

4.1 Implementation

The placement of the camera is based on the position of the two people having the conversation (see figure 3). However, more important than placing the camera in the approximate geometric relationship shown in figure 3 is the positioning of the camera based on what is being framed within the image.

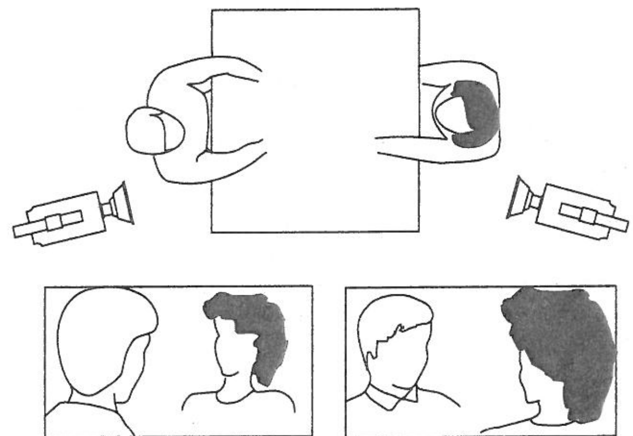

Figure 3: Filming a conversation [Katz88].

Constraints for an over-the-shoulder shot:

- The height of the character facing the view should be approximately 1/2 the size of the frame.
- The person facing the view should be at about the 2/3 line on the screen.
- The person facing away should be at about the 1/3 line on the screen.
- The camera should be aligned with the world up.
- The field of view should be between 20 and 60 degrees.
- The camera view should be as close to facing directly on to the character facing the viewer as possible.

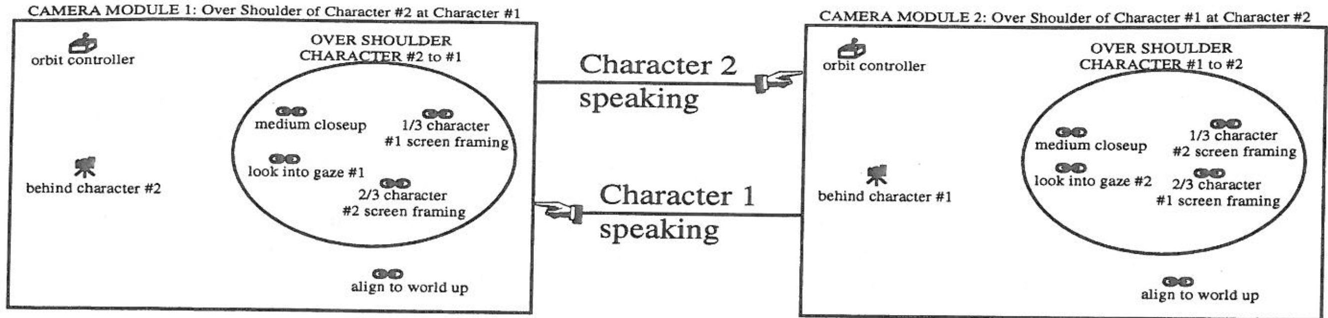

Figure 4: Two interconnected camera modules for filming a conversation

Constraints for a corresponding over-the-shoulder shot:

- The same constraints as described above but the people should not switch sides of the screen; therefore the person facing towards the screen should be placed at the 1/3 line and the person facing away should be placed at the 2/3 line.

Figure 3 can be used to find the initial positions of the cameras if necessary, but the constraint solver contained within each camera module makes sure that the composition of the screen is as desired.

Figure 4 shows how two camera modules can be connected to automatically film a conversation.

A more complicated combination of camea modules can be incorporated as the behaviors of a simple software agent. The agent contains a rudimentary reactive planner which pairs camera behaviors (combination of camera primitives) in response to sensed data. The agent has two primary sets of camera behaviors: one for when character 1 is speaking; and one for when character 2 is speaking. The agent needs to have sensors which can "detect" who is speaking and direct a camera module from the desired set of behaviors to become active. Since the modules necessarily keep track of the positions of the characters in the environment, the simulated actors can move about while the proper screen composition is maintained.

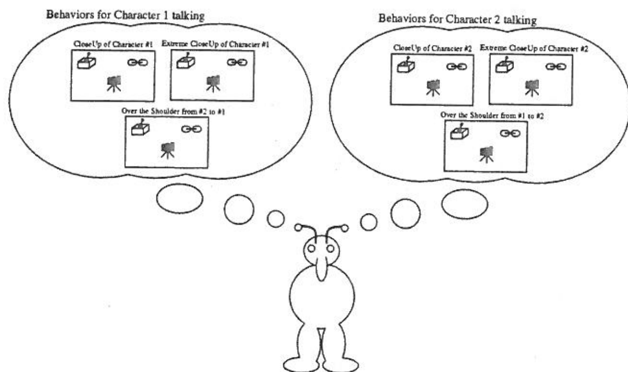

Figure 5: Conversation filming agent and its behaviors.

Figure 6 shows an over-the-shoulder shot automatically generated by the conversation filming agent.

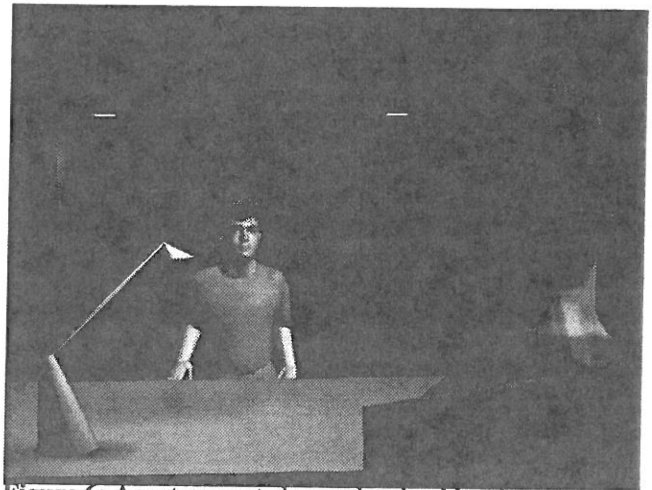

Figure 6: Agent generated over-the-shoulder shot.

5. Example: the Virtual Football Game

The virtual football game was chosen as an example because there already exists a methodology for filming football games that can be called upon as a reference for comparing the controls and resultant output of the virtual football game. Also, the temporal flow of the football game is convenient since it contains starting and stopping points, specific kinds of choreographed movements, and easily identifiable participants. A visual programming language for combining camera primitives into camera behaviors was explored. Finally, an interface, on top of the visual programming language, based directly on the way that a conventional football game is filmed, was developed.

It is important to note that there are significant differences between the virtual football game and filming a real football game. Although attempts were made to make the virtual football game realistic— three-dimensional video images of players were incorporated and football plays were based on real plays [15] —this virtual football game is intended to be a testbed for intelligent camera control rather than a portrayal of a real football game.

5.1 Implementation

Figure 7 shows the visual programming environment for the camera modules. Similar in spirit to Haeberli's ConMan [11] or Kass's GO [13], the system allows the user to connect camera modules,

and drag and drop initial conditions and constraints, in order to control the output of the CamDroid system. The currently active camera module's camera state is used to render the view of the graphical environment. Modules can be connected together by drawing a line from one module to the next. A boolean expression can then be added to the connector to indicate when control should be shifted from one module to the connected module. It is possible to set up multiple branches from a single module. At each frame, the branching conditions are evaluated and control is passed to the first module whose branching condition evaluates to TRUE.

Constraints can be instanced from existing constraints, or new ones can be created and the constraint functions can be entered via a text editor. Information for individual constraints can be entered via the keyboard or mouse clicks on the screen. When constraints are dragged into a module, all the constraints in the module are included during optimization. Constraints may also be grouped so that slightly higher level behaviors composed of a group of low level primitives may be dragged directly into a camera module.

Initial conditions can be dragged into the modules to force the minimization to start from those conditions. Initial conditions can be retrieved at any time from the current state of the camera. Camera modules can also be indicated to use the current state to begin optimization when control is passed to them from other modules.

Controllers can also be instanced from a palette of existing controllers, or new ones created and their functions entered via a text editor. If a controller is dragged into the module, it will translate the actions of the user subject to the constraints within the module. For example, a controller that will orbit about an object may be added to a module which constrains the camera's up vector to align with the world up vector.

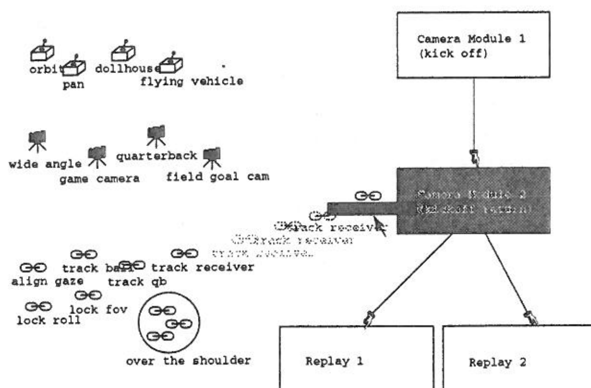

Figure 7: Visual Programming Environment for camera modules

The end-user does not necessarily wish to be concerned with the visual programming language for camera control. An interface that can be connected to the representation used for the visual programming language is shown in Figure 7. The interface provides a mechanism for setting the positions and movements of the players within the environment, as well as a way to control the virtual cameras. Players can be selected and new paths drawn for them at any time. The players will move along their paths in response to clicking on the appropriate buttons of the football play controller. Passes can be indicated by selecting the appropriate players at the appropriate time step and pressing the pass button on the play controller.

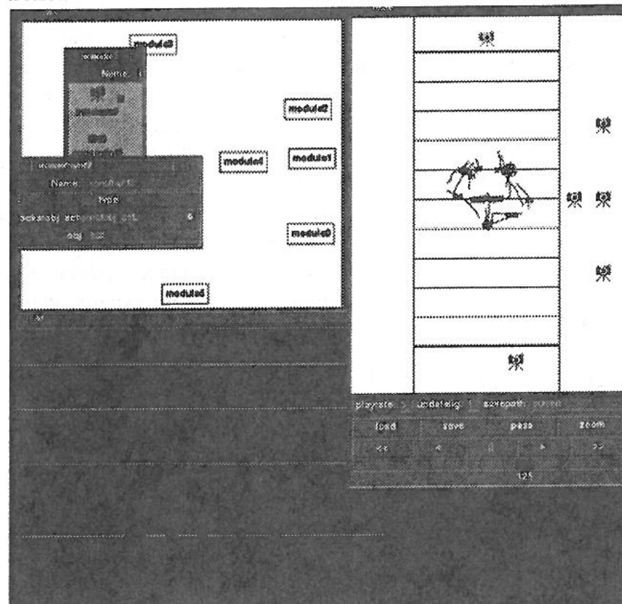

Figure 8: The virtual football game interface

The user can also select or move any of the camera icons and the viewpoint is immediately shifted to that of the camera. Essentially, pressing one of the camera icons activates a camera module that has already been set up with initial conditions and constraints for that camera. Cameras can be made to track individual characters or the ball by selecting the players with the middle mouse button. This automatically adds a tracking constraint to the currently active module. If multiple players are selected, then the camera attempts to keep both players within the frame at the same time by adding multiple tracking constraints. The image can currently be fine-tuned by adjusting the constraints within the visual programming environment. A more complete interface would provide more bridges between the actions of the user on the end-user interface and the visual programming language..

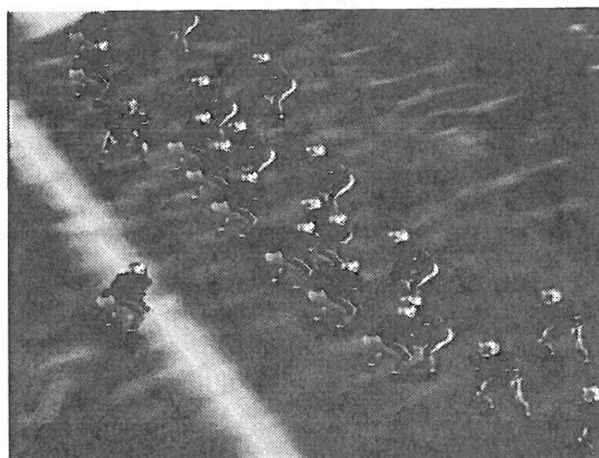

Figure 9: View from "game camera" of virtual football game.

6. Results

We have implemented a variety of applications from a disparate set of visual domains, including the virtual museum [8], a mission planner [21], and the conversation and football game described in this paper. While formal evaluations are notoriously difficult, we did enlist the help of domain experts who could each observe and comment on the applications we have implemented. For the conversation agent, our domain expert was MIT Prof.essor Glorianna Davenport, in her capacity as an accomplished documentary filmmaker. For the virtual football game, we consulted with Eric Eisendratt, a sports director for WBZ-TV, Boston. In addition, MIT Professor Tom Sheridan was an invaluable source of expertise on teleoperation and supervisory control. A thorough discussion of the applications, including comments of the domain experts, can be found in [9].

7. Summary

A method of encapsulating camera tasks into well defined units called "camera modules" has been described. Through this encapsulation, camera modules can be designed which can aid a user in a wide range of interaction with 3D graphical environments. The CamDroid system uses this encapsulation, along with constrained optimization techniques and visual programming to greatly ease the development of 3D interfaces. Two interfaces to distinctly different environments have been demonstrated in this paper.

8. Acknowledgements

This work was supported in part by ARPA/Rome Laboratories, NHK (Japan Broadcasting Co.), the Office of Naval Research, and equipment gifts from Apple Computer, Hewlett-Packard, and Silicon Graphics.

9. References

1. Arijon, D., *Grammar of the Film Language.* 1976, Los Angeles: Silman-James Press.

2. Blinn, J., Where am I? What am I looking at? *IEEE Computer Graphics and Applications,* July 1988.

3. Brooks, F.P., Jr. Grasping Reality Through Illusion -- Interactive Graphics Serving Science. *Proc. CHI '88.* May 15-19, 1988.

4. Brooks, F.P., Jr. Walkthrough -- A Dynamic Graphics System for Simulating Virtual Buildings. *Proc. 1986 ACM Workshop on Interactive 3D Graphics.* October 23-24, 1986.

5. Chapman, D. and C. Ware. Manipulating the Future: Predictor Based Feedback for Velocity Control in Virtual Environment Navigation . *Proc. 1992 Symposium on Interactive 3D Graphics.* 1992. Cambridge MA: ACM Press.

6. Chen, D. T. and D. Zeltzer. The 3d Virtual Environment and Dynamic Simulation System. Cambridge MA, Technical Memo. MIT Media Lab. August, 1992.

7 Drucker, S., T. Galyean, and D. Zeltzer. CINEMA: A System for Procedural Camera Movements. *Proc. 1992 Symposium on Interactive 3D Graphics.* 1992. Cambridge MA: ACM Press.

8. Drucker, S. M. and D. Zeltzer. Intelligent Camera Control for Virtual Environments. *Graphics Interface '94.* 1994.

9. Drucker, S.M *Intelligent Camera Control for Graphical Environments.* PhD. Thesis. MIT Media Lab. 1994.

10. Gleicher, M..A.W. Through-the-Lens Camera Control. *Computer Graphics.* 26(2): pp. 331-340. 1992

11. Haeberli, P.E., ConMan: A Visual Programming Language for Interactive Graphics. *Computer Graphics.* 22(4): pp. 103-111. 1988

12. Karp, P. and S.K. Feiner. Issues in the automated generation of animated presentations. *Graphics Interface '90.* 1990.

13. Kass, M. GO: A Graphical Optimizer. in ACM SIGGRAPH 91 Course Notes, Introduction to Physically Based Modeling. July 28-August 2, 1991. Las Vegas NM.

14. Katz, S.D., *Film Directing Shot by Shot: Visualising from Concept to Screen.* 1991, Studio City, CA: Michael Weise Productions.

15. Korch, R. *The Official Pro Football Hall of Fame.* New York, Simon & Schuster, Inc. 1990.

16. Mackinlay, J. S., S. Card, et al. Rapid Controlled Movement Through a Virtual 3d Workspace. *Computer Graphics* 24(4): 171-176. 1990.

17. Ousterhout, J. K. Tcl: An Embeddable Command Language. *Proc. 1990 Winter USENIX Conference.* 1990.

18. Philips, C.B.N.I.B., John Granieri. Automatic Viewing Control for 3D Direct Manipulation. *Proc. 1992 Symposium on Interactive 3D Graphics.* 1992. Cambridge, MA.: ACM Press.

19. Ware, C. and S. Osborn. Exploration and Virtual Camera Control in Virtual Three Dimensional Environments. *Proc. 1990 Symposium on Interactive 3D Graphics,* Snowbird, Utah, 1990. ACM Press.

20. Zeltzer, D. Autonomy, Interaction and Presence. *Presence: Teleoperators and Virtual Environments* 1(1): 127-132. March, 1992.

21. Zeltzer, D. and S. Drucker . A Virtual Environment System for Mission Planning. *Proc. 1992 IMAGE VI Conference,* Phoenix AZ.July, 1992.

3D Painting on Scanned Surfaces

Maneesh Agrawala
Andrew C. Beers
Marc Levoy

Computer Science Department
Stanford University

Abstract

We present an intuitive interface for painting on unparameterized three-dimensional polygon meshes using a 6D Polhemus space tracker as an input device. Given a physical object we first acquire its surface geometry using a Cyberware scanner. We then treat the sensor of the space tracker as a paintbrush. As we move the sensor over the surface of the physical object we color the corresponding locations on the scanned mesh. The physical object provides a natural force-feedback guide for painting on the mesh, making it intuitive and easy to accurately place color on the mesh.

CR categories: I.3.6 [Computer Graphics]: Methodology - Interaction Techniques. I.3.7 [Computer Graphics]: 3D Graphics and Realism - Color and texture; Visible surface algorithms.

Additional keywords: 3D painting, painting systems, direct manipulation, user-interface.

1 Introduction

Painting systems are a very common tool for computer graphics and have been well studied for painting on 2D surfaces. While many two dimensional techniques can be applied to painting on 3D surfaces, there are issues that are unique to 3D object painting. The most important aspect in developing a 3D painting system is maintaining an intuitive, precise and responsive interface. It is crucial that the user be able to place color on the surface mesh easily and accurately.

Many computer graphics studios (including Pixar and Industrial Light and Magic) have developed their own 3D paint programs which use a mouse as the input device. These painting systems are often used to paint textures onto the 3D computer graphics models which they will then animate. The user paints on some two-dimensional image representing the three dimensional surface and the program applies an appropriate transformation to convert the 2D screen space mouse movements into movements of a virtual paintbrush over the 3D mesh. Hanrahan and Haeberli describe such a system for painting on three-dimensional parameterized meshes using a two-dimensional input device in [5]. The main feature of this system, and one which we retain in ours, is that painting is done directly on the mesh in a WYSIWYG (What You See Is What You Get) fashion. The drawback of this system is that the transformation from the 2D screen space to the 3D mesh may not always be immediately clear.

This type of system could be extended to use a 3D input device. Movements of a sensor through space would map directly to movements of the virtual paintbrush. Such a system might be difficult to use, however, because there would be no way to "feel" when the paintbrush is touching the mesh surface. This problem could be solved by providing the user with force-feedback, the importance of which is well recognized (see [2], [10], [4]).

In our system, 3D computer models are built from physical objects, so these objects are available to serve as a guide for painting. As 3D computer graphics applications have become widespread, the demand for 3D models has lead to the development of 3D scanners which can scan the surface geometry of a physical object. Turk and Levoy have recently developed a technique for taking several scans of an object and "zippering" them together to create a complete surface mesh for the object [11]. If a surface mesh has been derived from a physical object in this way, the quickest, most intuitive method for specifying where to paint the mesh would be to point to the corresponding location on the surface of the physical object.

Our approach is based on this idea. Given a physical object we scan its surface geometry. We then use a 6D Polhemus space tracker as an input device to the painting system. As we move the sensor of the tracker over the surface of the physical object, we paint the corresponding locations on the surface of the scanned mesh. The sensor of the space tracker can be thought of as a paintbrush, providing a familiar metaphor for understanding how to use our system.

The remainder of this paper is organized as follows. Section 2 describes the organization of our painting system. Section 3 details how our system represents meshes internally. Section 4 discusses the algorithms and methods we use for painting, registration, and combating registration errors. Our results are presented in section 5. Section 6 discusses possible future directions of this work, and section 7 summarizes our conclusions about our system.

2 System Configuration

The block diagram in figure 1 depicts our overall system configuration. Before we can paint, we must create a mesh representing a physical object. We use a Cyberware laser range scanner to take multiple scans of an object and combine them into a single mesh using the zipper software. The Polhemus Fastrak space

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
1995 Symposium on Interactive 3D Graphics, Monterey CA USA
© 1995 ACM 0-89791-736-7/95/0004...$3.50

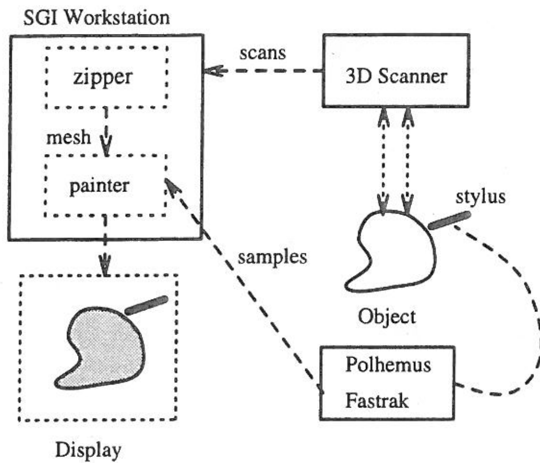

Figure 1: 3D Painting System Configuration

tracking system tracks the location of a stylus as it is moved over the physical object. The painter application maps these stylus positions to positions on the zippered mesh.

The Cyberware Scanner uses optical triangulation to determine the distance of points on the object from the scanning system. A sheet of laser light is emitted by the scanner. As the object is passed through this sheet of light, a camera, located at a known position and orientation within the scanner, watches the object. The scanner triangulates the depths of points along the intersection of the object and the laser sheet based on the image captured by the camera. As the object passes through the laser sheet, a mesh of points representing the object as seen from this point of view is formed.

The Polhemus Fastrak tracking system reports the 3D position and orientation of a stylus used to select the area on the mesh to paint. A field generator located near the object emits an AC magnetic field which is detected by sensors in the stylus to determine the stylus's position and orientation with respect to the field generator. The painter application continously polls the tracker for the stylus' poisiton and orientation at about 30 Hertz.

3 Data Representation

Previous work in 3D painting has only allowed painting on parameterized meshes, or on meshes that have texture coordinates previously assigned at each mesh point. Paint or surface properties applied to these meshes can be stored in a texture map, in the former case using the parameter values at points on the mesh as texture coordinates. While Maillot, Yahia, and Verroust have developed a method for parameterizing smooth surface representations[9], there are no general techniques for parameterizing arbitrary surface meshes.

Although a single Cyberware scan results in a parameterized triangle mesh, suitable for use by other 3D painting systems, such a mesh is generally not a complete description of the object. This incompleteness is due to self-occlusions on the object, making some points on the object invisible to a rotational scan. By combining data from multiple scans, Turk and Levoy's zippering algorithm [11] produces a more complete mesh for the object. However, the resulting mesh is irregular and unparameterized, so we lose the ability to store surface characteristics in texture maps.

To paint on unparameterized meshes, we store surface characteristics (e.g. color and lighting model coefficients) at each mesh vertex. When painting on the object, these surface characteristics are changed only at the mesh vertices. We render the mesh using the SGI hardware Gouraud shading to interpolate the color between the vertices of triangles composing the mesh. Because we do not require regular or parameterized meshes, our algorithm works with meshes acquired from many different kinds of scanning technologies, including hand digitizers, CT scanners and MRI scanners. CT and MRI scanners produce volume data rather than a surface mesh and so an algorithm like marching cubes [8] would be required to convert the volume data set into a suitable mesh representation.

Since we only have color information at the vertices of the mesh polygons, the polygons should be small enough to avoid sampling artifacts when displaying the mesh. As Cook, Carpenter and Catmull point out in their description of the REYES rendering architecture [3], this is possible when polygons are on the order of a half pixel in size. Due to memory constraints we typically paint on meshes in which triangles are about the size of a pixel when the mesh is displayed at a "reasonable" size (e.g. a quarter of the size of the monitor). We have implemented controls for scaling the display of the mesh so that it is always possible to reduce its display size to achieve subpixel color accuracy.

Since we would like to use a mesh with small triangles, the number of triangles in a typical mesh may be quite large. We therefore need to augment the triangle mesh with a spatial data representation that will allow us to find mesh vertices quickly. To facilitate this, we uniformly voxelize space. Associated with each voxel is a list of vertices on the mesh that are contained in that voxel. Storing these voxels in a hash table gives us nearly constant-time access to any vertex on the mesh, given a point close to it. Alternatively we could have used a hierarchical representation such as an octree for storing the spatial representation.

We do not use a simple 3D array indexed by voxel location because most meshes will contain large empty regions in voxel space. By using a hash table, we do not explicitly store the empty regions of voxel space, which results in a tremendous reduction in memory usage.

4 Methods

4.1 Object–mesh registration

When painting an object with our system, the user places the object on a table in front of the workstation. Before we can paint the mesh, we need to determine a transformation between positions reported by the tracking system in the coordinate space of the physical object and points in the coordinate space of the mesh. We would also like this transformation to ensure that relative orientations of the physical stylus and the virtual paintbrush are the same. We can accomplish this by finding an affine, shear-free transformation between the two coordinate spaces. We use a method developed by Horn [6] for obtaining such a transformation.

Horn's method determines a translation, rotation, and scaling that will align points in one coordinate system to corresponding points in another coordinate system, while minimizing the total distance between the sets of points. The two sets of points may be collected as follows. First, the mouse is used to select a point on the mesh. Then, the stylus is used to point to the corresponding point on the object, thus specifying a correspondence pair. Horn's method requires three or more of these correspondence pairs to determine the registration transformation.

There are several sources of error in collecting the two sets of points including inaccuracies in the tracking system, and inaccuracies in matching the points on the mesh to points on the object. However, as the number of correspondence pairs is increased, small alignment errors in individual pairs are averaged out and the total alignment error decreases. Unfortunately, specifying correspondence pairs is tedious and time consuming.

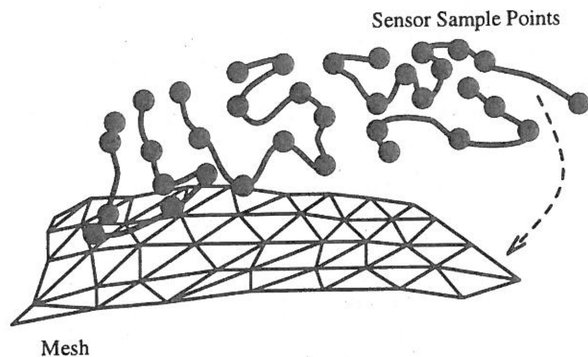

Figure 2: A large set of sensor sample points is collected by running the sensor of the space tracker randomly over the surface of the object. These sensor points are roughly hand-aligned with the mesh, and then Besl's algorithm is used to obtain a more precise alignment.

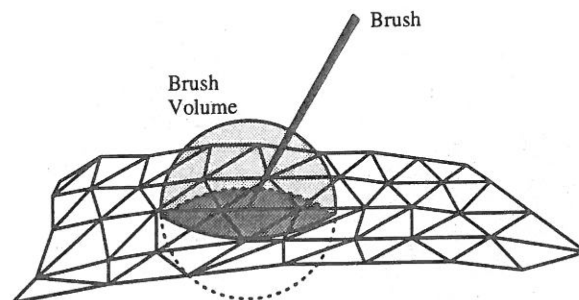

Figure 3: Paint is applied to all mesh vertices falling within the brush volume. Here the vertices in the dark gray region are painted.

An algorithm developed by Besl[1] overcomes this problem. Although two sets of points are still required, it is not necessary to specify the point-to-point correspondences between them. We collect a large set of points in tracker space by sampling the position of the stylus while randomly moving it across the surface of the physical object. We use a subset of the mesh vertices as the other set of points. Besl's algorithm determines the best transformation between the two sets of points by iterating on the following steps. First an approximate correspondence between the two sets of points is computed, based on their proximity in space. Then, Horn's method is applied to these pairs of points to align them more closely. On each successive iteration of the algorithm, the proximity-based correspondence improves, which in turn improves the transformation generated by Horn's method.

Besl's algorithm is guaranteed only to find a locally optimal alignment, not a globally optimal one. Therefore, we need to ensure that the sensor samples and the mesh are initially aligned such that the globally optimal solution can be found. The initial alignment is done by hand as (see figure 2), and is often a difficult and time consuming process. To speed this process, we have added the ability to easily generate a rough alignment of the sensor samples to the mesh. Once we have collected the large set of sensor samples, we ask the user to specify three or more correspondence pairs as described at the beginning of this section. From these pairs we calculate the scale factor between the sensor samples and the mesh. We also translate the centroid of the sensor correspondence points so that it is aligned with the centroid of the mesh. This produces a rough alignment of the sensor samples to the mesh which can then be hand-refined to produce the initial alignment required for Besl's algorithm.

Our registration scheme is summarized as follows:

1. The user collects many samples of the physical object's surface by running the stylus over the object.
2. The user selects three or more points on the mesh, and points to their corresponding locations on the physical object with the stylus. These correspondence pairs are used to compute a rough alignment of the sensor samples collected during step 1 to the mesh.
3. If necessary, the user makes further hand adjustments to the rough alignment of the sensor samples to the mesh using the mouse to bring them into initial alignment.
4. Besl's algorithm is run to refine the alignment of the sensor samples to the mesh.

4.2 Painting

To paint a three-dimensional surface we must determine where new paint is to be applied. The tip of our paintbrush has a 3D shape associated with it which defines the volume within which paint is applied (see figure 3). In general this brush volume can be any 3D shape. The most straightforward painting algorithm would be to paint every vertex that falls within the brush volume. We can think of this approach as filling the entire brush volume with paint using a 3D scan-line algorithm to step through all the voxels within the volume. The drawback of this approach is that the mesh is likely to be relatively flat within this volume, therefore not filling much of it. This volume-fill algorithm would search through many empty voxels.

Our approach is to first find a vertex on the mesh that is within the brush volume. We then perform a breadth-first flood fill of the mesh from this seed point. The vertex on the mesh closest to the ray extended along the brush direction from the sensor position is used as the seed, as depicted in figure 4.

Although we poll the tracker for the position of its sensor at about 30 Hertz, the sampling rate is not fast enough to produce a smooth stroke as the brush is swept along the object. For the paint to be applied smoothly, without gaps, we need to fill the surface with paint along a stroke. The flood fill idea can be modified to account for this, coloring vertices within the volume defined by sweeping the 3D brush shape along a stroke connecting successive sensor positions. In our system, we connect successive positions using a linear stroke. Thus, for a sphere brush we would sweep out a cylindrical volume with spherical end caps along the stroke.

One problem for the flood fill algorithm is that it can not correctly handle all surface geometries. Consider a surface with a small indentation. If we place the brush directly above the indentation we should be able to paint the surfaces on either side of it. However, the flood fill brush will only paint one side of it, because it floods out along the mesh surface from the seed point as shown in figure 5(A). This problem could be prevented by performing a volume-fill within the brush geometry, as in figure 4, rather than flood filling out from the seed point along the mesh surface. In practice, we have never encountered a surface geometry for which the surface flood fill causes noticeable anomalies.

Another problem with this algorithm is that mesh triangles which are occluded to the paintbrush may be painted. The correct solution to the problem would be to do a complete visibility test before painting a vertex to ensure that the vertex was visible to the brush. Because this test is very expensive and would hinder interactive performance, we only check that the dot product of the

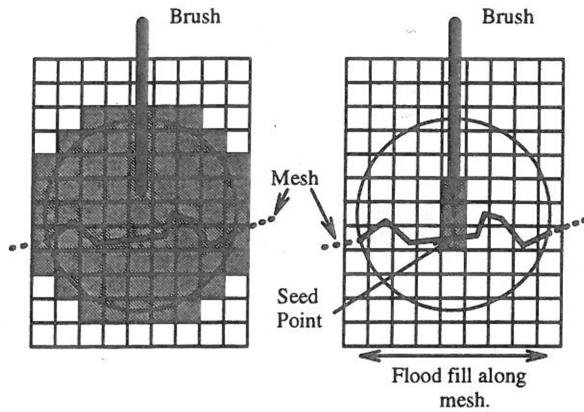

Figure 4: Two methods for determining where to apply paint within a spherical brush volume. The scan-line algorithm walks through every voxel within the brush volume. The flood-fill algorithm extends a ray from the brush tip to the surface and then floods paint out along the surface.

vertex normal and the brush orientation is negative. This ensures that we only paint vertices that are facing the brush, but there are still some cases where we might paint occluded triangles, as shown in figure 5(B). In this case the flood fill seed point falls on the left side of Peak B. As color floods out from the seed point along the left side of Peak B, points that are occluded by Peak A will be painted. The volume-fill approach would be no better than the flood-fill approach at handling this mesh geometry. Both methods fail because they do not check for occlusions between the tip of the brush and the mesh surface.

With hundreds of thousands of polygons in a typical mesh it would be impossible to redraw the entire mesh after each paint stroke and maintain interactive performance. Instead, we only redraw the triangles in which at least one vertex was painted. By using the surface flood fill algorithm in combination with this lazy update scheme we can interactively paint large meshes.

4.3 Brush effects

We have implemented several different brush volumes including a sphere, cylinder and cone, and several different brush effects. The sphere brush paints all vertices within a sphere centered at the brush tip. The cylinder paints all vertices within a cylinder centered at the brush tip and oriented in the direction of the brush. The cylinder brush is typically used to fill large areas by stroking it lengthwise along the surface. The cone brush paints all vertices within a cone, with its apex at the brush tip and oriented in the direction of the brush. By tilting this brush as we paint we can achieve the effect of painting with an airbrush.

Another effect we implemented was to modulate the application of color using 3D solid textures and 2D image textures. To apply solid textures, we use the vertex location as an index into a texture map and apply the corresponding texture color. For 2D textures we define a plane on which the texture resides and perform an orthogonal projection of the unparameterized 3D mesh points into the texture plane. This gives a mapping from the mesh points into the texture. The user can control the position, orientation and scale of the 2D texture plane through a mouse-driven interface.

We have also implemented several compositing filters that are applied to the paint as it is laid down on the surface. The simplest filter is the "over" filter. Using this filter, the paint from the brush replaces the paint at each affected vertex. The "blend" filter has a slider-selectable parameter α and performs standard alpha blending between the old mesh color and the new paint color. The "distance" filter is a special case of the blend filter for which alpha is proportional to the distance of each affected vertex from the tip of the brush.

Each of the brushes we have described so far only affects the surface characteristics of the mesh. We can also change the geometry of the mesh using a displacement brush. Our displacement brush pulls mesh vertices within the brush geometry in the direction of the brush. Although this is an effective way to change the surface geometry, it undermines the use of the physical object as a painting guide. In practice, however, we have found that if we apply small displacements, the physical object can still be used as a guide. A problem with the current implementation is that it is possible to produce objectionably long, thin triangles as we pull the surface. We could alleviate this problem by re-polygonalizing the triangles as we elongate them during the displacement.

4.4 Combating registration errors

The accuracy of the registration between the sensor and the mesh depends on several factors. The Polhemus Fastrak is only accurate to within 0.03 inches, and the magnetic field generated by the Polhemus is distorted by metallic objects as well as other electromagnetic fields in the work area. Furthermore, Besl's registration algorithm is dependent on an initial hand-alignment of the sensor samples to mesh vertices. If this initial alignment is poor, the registration transformation produced by Besl's algorithm may not be globally optimal. Registration errors can cause the virtual brush tip to lie some distance away from the mesh even when the Polhemus stylus is physically touching the object surface. In this case it would be difficult to paint the surface with small brush volumes.

One approach to overcome this would be to use a long, thin cylindrical brush. The problem with this approach is that painting

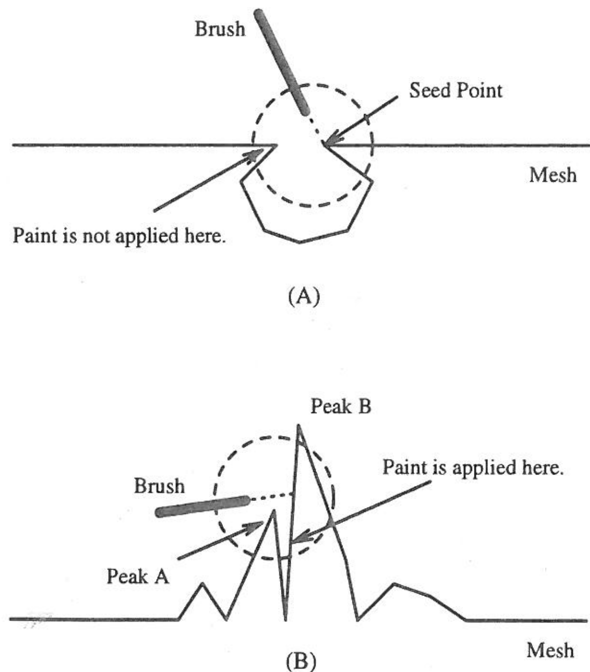

Figure 5: Mesh geometries which cause problems for the painting algorithm.

a fine line with such a long, thin brush would force us to ensure that the brush is perpendicular to the mesh throughout the stroke. Slight changes in brush orientation would change the size of the area painted on the mesh.

An alternative approach is to give the user the option of "gluing" the brush to the mesh. When painting, the location of the brush is constrained to be the closest point on the mesh to the sensor, rather than the sensor's location itself. We can think of this as extending the tip of the brush so that it always touches the mesh surface. Since the brush's position is now forced to lie on the surface, we can paint with very small brush shapes, even in the presence of registration errors.

5 Results

We have been able to paint detailed textures on several different meshes including the bunny and the wolf-head, shown in color plates 2-8. The bunny mesh was created by zippering 10 Cyberware scans of the ceramic bunny shown in plate 1; the final mesh contains 69,451 triangles. Plate 2 shows sensor sample points in the process of being initially aligned with the bunny mesh in preparation for running Besl's registration algorithm. The purple crosses represent sensor sample points.

A 3D checkerboard texture and 2D image texture of an orchid were applied to the bunny shown in plate 3. While the triangles in the original bunny mesh were about the size of a pixel, we found that a finer mesh was necessary to capture fine detail in the image texture. We refined the original bunny mesh by simply splitting each triangle into four smaller triangles.

Plates 4-8 show several complete paintings we created with our system. Most of the paintings took several hours to complete. The wolf-head mesh in plate 8 contains 58,104 triangles while the higher-resolution wolf-head mesh used in plates 6 and 7 contains 232,416 triangles. The bunny head mesh in plate 5 is a piece of the high-resolution bunny mesh, while the low-resolution bunny mesh was used in plate 4.

In creating the bumpy wolf shown in plate 7 we used almost every painting tool we implemented. The bumps were created by applying the displacement brush with a spherical brush volume to the mesh. The distance filter was used in coloring the bumps as they were extruded from the mesh. As in plates 3 and 6, the orchid is a 2D image that was texture mapped onto the mesh.

6 Future Directions

One of the drawbacks of our system is that there is a non-trivial amount of set-up time required to register the physical object to the mesh. Registration can take several minutes and must be done every time the user wants to paint an object. Furthermore, if the object is moved after it has been registered, it must be re-registered. The most time-consuming aspect is doing the final hand alignment of the registration points to the surface mesh.

One solution to this problem would be to register the physical object as it is being scanned by the 3D scanner. Assuming the scanner always creates a mesh in the same coordinate system for each scan, we can preregister the tracker coordinate system to this mesh coordinate system using Besl's algorithm. Then, scanning any new object will automatically register it to the tracking system. However, this approach fails when we combine multiple scans using the zipper software, because the physical object must be moved between scans and so we lose the correspondence between the mesh and the object.

Ensuring that the object does not move once it has been registered is can make painting awkward and unnatural. Allowing the object to be moved would let the user to paint more comfortably. One way to permit such object movement would be to attach another sensor of the space tracker to the object and then track the movement of the object in addition to the movement of the brush.

A disadvantage of our approach is that we can only paint meshes for which we have a corresponding physical object. Thus, we can not directly paint a mesh created with a modeling or CAD program for example. However, several new rapid prototyping technologies have recently been developed for synthesizing 3D objects directly from computer models [7] [12]. Although it would be a considerable expense, with such a prototyping system we could create a physical object representing almost any mesh and then use it as a guide for painting on the mesh.

Another problem is that the user is moving the sensor along the physical object while paint is only being applied to the mesh on the monitor. Thus, the user must look at two places at once to see where the paint is being applied. This problem is reduced by placing the physical object in front of the monitor while painting.

One of the problems with polygon meshes is that they are hard to animate. Many animators are used to manipulating the control points of curved surface patches, not the vertices of an irregular mesh. Furthermore, they want to manipulate only a few control points, not the 100,000's of vertices in our typical mesh. One solution we are investigating is to fit NURBS patches to our meshes. The boundaries of these patches would be specified by tracing them using our system. In this case we would replace our space-filling brushes with an algorithm that chains together mesh vertices lying along the path traced out by the stylus.

7 Conclusions

We have developed an intuitive 3D interface for painting on 3D computer models, using the sensor of a Polhemus 6D tracker as a paintbrush. The fundamental feature of our system is that a physical object provides a force feedback guide for painting. Our system is fast enough to paint a mesh in real time as the sensor is moved over the physical surface, giving the user a sense of directly painting on the mesh. With this system there is no need to perform a transformation from 2D input space to the 3D mesh surface, as is required by other 3D painting systems that use a 2D input device. Also unlike other 3D painting systems, the meshes we paint do not need to be parameterized in any way. With our system an artist who is experienced with painting on 3D physical objects can almost directly apply that experience to painting on surface meshes.

8 Acknowledgments

We would like to acknowledge the many individuals who helped us with this project. In particular we would like to thank Greg Turk for his insightful discussions and for providing meshes for us to paint. George Dabrowski and Michael Zyda especially encouraged us to develop this project. Bill Chapin, Nat Bletter and Dan Goldman wrote the original interface for the Polhemus tracker. Alan Baronkey and Bill Chapin provided access to crucial hardware required for this project. Finally we would like to thank Lincoln Hu for giving us permission to use the physical model of the wolf head created by Industrial Light and Magic.

REFERENCES

[1] Paul J. Besl. A Method for Registration of 3D Shapes. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 14(2):239–255, February 1992.

[2] Frederick P. Brooks, Jr., Ming Ouh-Young, James J. Batter, and P. Jerome Kilpatrick. Project GROPE — Haptic Displays for Scientific visualization. In Forest Baskett, editor, *Proceedings of SIGGRAPH '90*, volume 24, pages 177–185, August 1990.

[3] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes Image Rendering Architecture. In Maureen C. Stone, editor, *Proceedings of SIGGRAPH '87*, pages 95–102, July 1987.

[4] Tinsley A. Galyean and John F. Hughes. Sculpting: An Interactive Volumetric Modeling Technique. In *Proceedings of SIGGRAPH '91*, volume 25, pages 267–274, July 1991.

[5] Pat Hanrahan and Paul Haeberli. Direct WYSIWYG Painting and Texturing on 3D Shapes. In *Proceedings of SIGGRAPH '90*, volume 24, pages 215–223, August 1990.

[6] Berthold K.P. Horn. Closed-form Solution of Absolute Orientation Using Unit Quaternions. *J. of the Optical Society of America*, 4(4):629–642, April 1987.

[7] N.F. Kinzie. Three-Dimensional Printing: a Tool for Solid Modeling. In *Conference Proceedings of NCGA '91*, pages 812–821, April 1991.

[8] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In Maureen C. Stone, editor, *Proceedings of SIGGRAPH '87*, volume 21, pages 163–169, July 1987.

[9] Jérôme Maillot, Hussein Yahia, and Anne Verroust. Interactive Texture Mapping. In James T. Kajiya, editor, *Proceedings of SIGGRAPH '93*, volume 27, pages 27–34, August 1993.

[10] Margaret Minsky, Ming Ouh-young, Oliver Steele, Frederick P. Brooks, Jr., and Max Behensky. Feeling and Seeing: Issues in Force Display. In Rich Riesenfeld and Carlo Sequin, editors, *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, volume 24, pages 235–243, March 1990.

[11] Greg Turk and Marc Levoy. Zippered Polygon Meshes from Range Images. In *Proceedings of SIGGRAPH '94*, pages 311–318, July 1994.

[12] T.T. Wohlers. Developments in 3D Printing and Rapid Prototyping. In *Conference Proceedings of NCGA '91*, pages 249–259, April 1991.

Volume Sculpting

Sidney W. Wang and Arie E. Kaufman
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

Abstract

We present a modeling technique based on the metaphor of interactively sculpting complex 3D objects from a solid material, such as a block of wood or marble. The 3D model is represented in a 3D raster of voxels where each voxel stores local material property information such as color and texture. Sculpting is done by moving 3D voxel-based tools within the model. The affected regions are indicated directly on the 2D projected image of the 3D model. By reducing the complex operations between the 3D tool volume and the 3D model down to primitive voxel-by-voxel operations, coupled with the utilization of a localized ray casting for image updating, our sculpting tool achieves real-time interaction. Furthermore, volume-sampling techniques and volume manipulations are employed to ensure that the process of sculpting does not introduce aliasing into the models.

1. Introduction

In this paper we present a free-form interactive modeling technique based on the concept of sculpting a voxel-based solid material, such as a block of marble or wood, using 3D voxel-based tools. There are two motivations for this work. First, although traditional CAGD and CAD have made great strides as design tools in many engineering disciplines, modeling topologically complex and highly-detailed objects are still difficult to design in most traditional CAD systems. Second, sculpting tools have shown to be useful in scientific and medical applications. For example, scientists and physicians often need to explore the inner structure of their simulated and sampled datasets by gradually removing material to reveal a section of interest.

Authors' email: swang@cs.sunysb.edu ari@cs.sunysb.edu
This work has been supported by the National Science Foundation under grant CCR-9205047.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
1995 Symposium on Interactive 3D Graphics, Monterey CA USA
© 1995 ACM 0-89791-736-7/95/0004...$3.50

The use of the sculpting metaphor for surface-based geometric modeling has been studied extensively [2, 8, 9, 11]. However, the concept of sculpting a volumetric object is relatively recent. Galyean and Hughes [4] generalized the 2D painting metaphor to volume sculpting by extending the 2D canvas to 3D volumetric clay. They employed marching cube polygons as an intermediate model representation and presented a novel localized marching cube rendering algorithm for achieving real-time interaction. The merit of their localized rendering algorithm is discussed further in Section 6. Although their algorithm is quite nice for generating clay or wax like sculptures, it cannot generate realistic looking objects such as those appeared in our daily environment. The use of numerically controlled (NC) milling machine as a volume sculpting tool has also been investigated [10, 14]. NC milling produces extrusion cut volume based on some geometric attributes of the rendered image or NC paths. Thus, the resulting model can only be viewed from the direction from which the rendered image was generated.

We developed a volume sculpting tool that is easy to use. A user does not need to possess the mathematical knowledge of surface modeling using CAGD techniques or solid modeling. Furthermore, models generated with our technique are free-formed and can be topologically complex. Although our models lack the precision that is required for accurate product manufacturing, such as a crankshaft, they are adequate as first-pass model designs or for applications where model precision is not greatly important, such as furnitures. Our interactive volume sculpting tool employs a ray casting algorithm for rendering. It achieves real-time interaction by employing a localized ray casting for image updating. Hence, a shape designer might be compelled to convey his/her design idea by sculpting a 3D model rather than drawing it on a 2D sketch board. In this way, during modeling the designer is able to transform (e.g., rotate) the object in space and see the design from different angles. In addition, our volume sculpting tool is suitable for manipulating sampled and simulated datasets. Furthermore, volume-sampling and volume-manipulations are used to ensure that the process of sculpting does not introduce aliasing into the models.

2. Object Representation

Unlike a traditional CAD model which commonly consists of a collection of surface patches, we employ the volume graphics approach [6] by modeling every object as voxel data, represented as a 3D volume raster. The volume raster grid is uniformly spaced along each of the three orthogonal axes, but the grid might be anisotropic, that is, the spacing constant might be different for the different axes. By simply changing the spacing constant of the model raster grid, one can alter the physical size of the 3D model. Since sample points in the volume raster are defined only at discrete locations in space, a reconstruction process is needed to reproduce the original continuous model. Commonly, the reconstruction is performed in a piecewise fashion by defining a trilinear interpolation function $f(x,y,z)$ over the eight neighboring grid points, $(\lfloor x \rfloor, \lfloor y \rfloor, \lfloor z \rfloor)$, $(\lfloor x \rfloor, \lfloor y \rfloor, \lceil z \rceil)$, $(\lfloor x \rfloor, \lceil y \rceil, \lfloor z \rfloor)$, $(\lfloor x \rfloor, \lceil y \rceil, \lceil z \rceil)$, $(\lceil x \rceil, \lfloor y \rfloor, \lfloor z \rfloor)$, $(\lceil x \rceil, \lfloor y \rfloor, \lceil z \rceil)$, $(\lceil x \rceil, \lceil y \rceil, \lfloor z \rfloor)$, $(\lceil x \rceil, \lceil y \rceil, \lceil z \rceil)$. Of course, one can achieve better reconstruction by employing a larger interpolation neighborhood and use a higher-order interpolation function.

Multiple volumes are supported in our volume sculpting system. Thus, a sculptor can walk in a gallery of multiple unfinished sculptures and work on different pieces in one session. In addition to having a world coordinate system for the entire scene, a local coordinate system is associated with each volume in the scene. Transformation between the world coordinate system and each volume coordinate system is facilitated by conversion matrices stored within each volume. Translations and rotations of each volume are performed by simply concatenating the new transformation matrix to the one stored in the volume, thereby defining a new current local coordinate system and new conversion matrices.

This modeling approach is an alternative to conventional surface-based graphics and has advantages over the latter by being able to store a view independent model and its attributes such as texture and antialiasing information, and is suitable for the representation of 3D sampled data such as those acquired from medical scanning devices. More importantly, for volume sculpting applications, it supports the visualization of internal structures, and lends itself to the realization of block operations and constructive solid modeling.

3. User Interaction

The following is a typical interaction sequence of our system:

1) User loads in the initial volume data.
2) User positions and orients the object to the desired view.
3) System projects the 3D object onto a 2D image from the selected view.

Repeat 4) and 5):
4) User moves a 3D tool to the desired region.
5) System performs the actual action locally.

Like a sculptor, the user first selects the approximate size and shape of the material. Our database contains a variety of geometric primitives and also a set of sampled and simulated datasets. The geometric datasets were synthesized from geometric descriptions into their volume graphics representation using the volume-sampling technique [15]. The process of volume-sampling bandlimits the continuous object by convolving it with a radially symmetric 3D filter. As a result, the surface of the filtered object has a smoothly varying density function from object to empty space. Hence, the corresponding discrete representation is free of object space aliasing.

Once the loaded volume data is rotated and oriented, it is projected using a volume rendering algorithm of ray casting. The rendering process is explained in detail in Section 6. On the projected image, the user either moves a 3D tool to the desired region for carving the object, or draws out the desired region for sawing. Carving is the process of taking a pre-existing tool to chip or chisel the object bits by bits, while sawing is the process of removing a whole chunk of material at a time, much like a carpenter sawing off a portion of a piece of wood. For sawing, the user first needs to draw out the desired sawing region directly on the projected image. Then, this 2D region is extruded in the direction perpendicular to the view plane to form a volume. A slider bar is provided to the user to specify the depth of extrusion.

From our experience, using a 2D input device such as a mouse is easier for the user to grasp than a 3D input device, such as an Isotrack. Furthermore, unless a collision detection is implemented, the position of the tool specified by the 3D input device can penetrate the surface of the solid model. Although the penetration of object surface is fine for a heat-gun metaphor, such as the one used in [4], it is inappropriate for our click-and-invoke carving and sawing metaphor.

4. Carving

In our system, a set of carving tools are available to the user. Each of these carving tools is pre-generated using a volume-sampling technique [15] and stored in a volume raster of $20 \times 20 \times 20$ resolution. Figure 1 illustrates three commonly used tool volumes. The user can adjust the physical size of these tools by changing the constant spacing of their raster grid. Rotation and translation of the

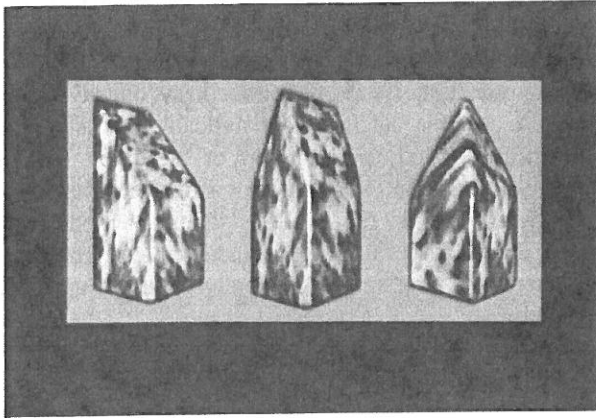

Figure 1: Commonly used carving tools

tool volume requires a simple matrix multiplication to update the object coordinate system with respect to the world coordinate system. Since both the object and the carving tool are represented as volume rasters, the process of carving involves positioning the tool volume with respect to the object in 3D space and performing a boolean subtraction between these two volumes.

As we have mentioned, the user specifies the position of the tool on the 2D projected image. This 2D position is then mapped onto a 3D position in the world space by using the projection depth, *depth*, of the pixel. This projection depth is essentially the z-buffer information. Hence, the 3D world position (x, y, z) corresponding to the 2D view position (u, v) is calculated as: .

$$(u, v, depth)_{view} \rightarrow (x, y, z)_{world}. \quad (1)$$

Next, the carving tool volume, which is volume-sampled, is subtracted from the object volume. Our algorithm does not require these two volumes to be aligned with respect to each other or having the same grid resolution. The algorithm starts by first determining the cubiodal sub-volume of the object that is overlapped by the tool volume. Then, for each grid point within that cubiodal sub-volume, a new data value is computed and assigned to reflect the affected region. Specifically, for each grid point (i, j, k) within the cubiodal sub-volume, the new sample value $f(i, j, k)$ is computed from:

$$f_{object}(i, j, k) = f_{object}(i, j, k) \ \ diff \ \ f_{tool}(x', y', z') \quad (2)$$

where

$$(i, j, k)_{object} \rightarrow (x', y', z')_{tool}, \quad (3)$$

$$i, j, k \in Integer, \quad x', y', z' \in Real.$$

Since our volume-sampled model is a density function $d(x, y, z)$ over R^3, where d is 1 inside the filtered object, 0 outside the filtered object, and $0 < d < 1$ within the soft region of the filtered surface, the boolean *diff* operator is based on algebraic sum and algebraic product [3, 15], which is employed to preserve continuity on the sculpted model:

$$A \ \ diff \ \ B = A - A\ B. \quad (4)$$

Note that instead of the *diff* operator, an $\cup$ operator defines as:

$$A \cup B = A + B - A\ B \quad (5)$$

can be used to add the tool volume to the object. This is a nice feature when one needs to patch up a hole or add some details to the model. Other set operations between the tool volume and the object are also possible, thereby implementing full voxblt (3D bitblt) capabilities [5]. Since point (x', y', z') does not usually fall on a grid point of the tool volume, a reconstruction method similar to the ones discussed in Section 2 is needed to interpolate $f_{tool}(x', y', z')$.

5. Sawing

Unlike carving, sawing requires the additional process of generating the tool volume on the fly. In our system, the user is able to draw any size circle, polygon, and Bezier curve to indicate the region to be sawed. Then, this 2D region is extruded to form the tool volume. However, to prevent object space aliasing, proper sampling and filtering must be used in generating this tool volume. Although we have previously developed a volume sampling technique [15] to accomplish this, it requires a time consuming 3D convolution process. To achieve interactive speed, we have developed a new volume sampling technique. Instead of gathering contribution from the portions of the tool that fall under the filter kernel when the kernel is centered over a sample point, the density of each point of the tool is splatted in 3D space to the affected neighboring sample points. If R is the radius of the splat kernel, then each splat affects a region of $(2R-1) \times (2R-1) \times (2R-1)$ neighborhood in 3D. An analogous 2D splatting is shown in Figure 2. The splatting of a density point (α, β, γ) to its neighboring sample grid points is formulated as:

$$\text{for } all\ \alpha - R < i < \alpha + R, \quad (6)$$

$$\text{for } all\ \beta - R < j < \beta + R,$$

$$\text{and for } all\ \gamma - R < k < \gamma + R,$$

$$f_{tool}(i, j, k) = h(\| i - \alpha,\ j - \beta,\ k - \gamma \|)$$

where h is a hypercone filter centered at (i, j, k). The hypercone filter has a spherical filter support and is weighted such that its maximum contribution is at the center