postors.

Examples of *view dependent* impostors are:

- A texture map that is pasted onto the appropriate face of an object's bounding box. This texture map is called a textured cluster when it corresponds to an image of a group of objects.
- Another view dependent texture map is also known as billboard in [6] and is obtained in the same way as texture maps. A billboard is centered at an object's center and made to rotate in such a way that it always face the observer. Since one billboard is computed for each face of the object's bounding box as the observer moves around the object a different billboard is selected to display according to the viewpoint. This impostor is useful to represent objects that are approximately rotationally symmetric such as pine trees.
- Another variant of the texture map described above is a pseudo-texture map[1]. A pseudo-texture map is a triangular mesh (or a quadrilateral strip) onto which a texture map is pasted in such a way that each pixel in the image is associated to a pair of triangles (or quadrilateral) in the mesh.

Examples of *view independent* impostors are:

- The conventional levels-of-detail, i.e., geometrically coarse versions of a given object[2].
- Boxes whose faces have the average areas and colors as the corresponding sides of the object's bounding box.
- Texture mapped boxes. This representation uses texture maps that are pasted onto each face of the object's bounding box and is useful to represent box like objects such as the Standard Oil Building in Chicago.

4.2 Impostor Selection

There are certain cases where specific impostors are more suitable than others and we can usually "suggest" to the walkthrough program which representation to display at a given point in the simulation.

For example, if the image-space size N of an object is less then a few pixels then the representation that should be used is the average box above. If N is greater then a pre-fixed maximum size then the full detail object might be required. If different LODs are present in the model, then different image space size thresholds may be used to select the appropriate LOD to be displayed.

Box-like and symmetric objects can be displayed using a texture mapped box and a billboard, respectively. Texture maps can be selected according to the obeserver's viewpoint. For example, if four texture maps are used for each face of an object's bounding box, then the appropriate texture map for a given viewpoint can be selected as follows:

1. In a pre-processing phase, associate to each texture map a number corresponding to the region it belongs as in Figure 3.

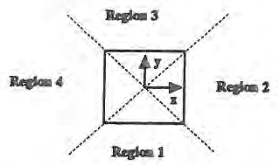


Figure 3: Possible viewpoint regions in object coordinates.

2. During the walkthrough we determine the viewpoint with respect to the object's coordinate system and therefore the region it is in.

In some situations, both a view dependent and a view independent representation are suitable. When this is the case, we can decide upon these two representations by obtaining the accuracy of each representation for the particular observer view angle using the table described in Section 3.2 and then select the representation with the highest accuracy/cost ratio. This heuristic is particularly useful in cases where the observer's line of sight is approaching a 45 degree angle with the line perpendicular to the texture map. In such a case although the texture map may have a low rendering cost, its accuracy will also have a low value which will favor the selection of a possibly more costly view dependent representation.

4.3 Formalization of the Problem

We begin by defining a meta-object abstraction to be an entity with one or more hardware drawable representations as in the framework described in Section 4.1. It is an abstraction for both conceptual objects and groups of objects.

As before, a hardware drawable representation is an entity that can be rendered by the graphics hardware to represent objects and has associated to it a rendering cost and a measure of its "contribution" to the simulation.

The model is then defined as a collection of conceptual objects at specific positions and orientations in space that forms the environment in which the user navigates.

The model hierarchy is defined to be a tree structure whose nodes are meta-objects that provide multiple representations of the model, each representing it at a given rendering time and providing the user with a given perception of it. In this hierarchy each node contains drawable representations of its children. The root contains the coarsest representations for the entire model with the lowest possible rendering cost while the leaves form the perceptually best representation of the model with the highest rendering cost.

More formally, the model hierarchy M is a tree structure that can recursively be defined by the following rules:

1. A meta-object that has no children is a model hierarchy with just one node, the root node.
2. Let $M_1, M_2 ... M_n$ be model hierarchies whose root nodes are the meta-objects $m_1, m_2 ... m_n$, respectively, that represent sets of conceptual objects and have associated with each of them the sets $r_1, r_2 ... r_n$ of drawable representations. Let m be a meta-object that represents the union of m_i and has associated to it a set r of drawable representations such that $Cost(Max(r)) < \sum_{i=1}^{n} Cost(Min(r_i))$, where $Max(r)$ is the representation that has the highest cost among those in r, $Min(r_i)$ is the representation that has the lowest cost among

[1]It can be used in machines that do not have texture mapping hardware.

[2]Some toolkits such as Performer[6] provide routines to automatically generate coarse versions of a given full-detail object.

those in r_i and $Cost(x)$ is the rendering cost of representation x. M is then defined to be a model hierarchy if m is the parent of m_i for $i = 1 \ldots n$.

Figure A shows how the model of a city would be organized to form a hierarchy in which each node has a set of impostors to represent the objects it subsumes.

Given these definitions, we state the walk-through problem as a tree traversal problem:

"Select a set of nodes in the model hierarchy that provides the user with a perceptually good representation of the model", according to the following constraints:

1. The sum of the rendering cost of all selected nodes is less than the user specified frame time.
2. Only one node can be selected for each path from the root node to a leaf node, since each node already contains drawable representations that represent all its descendant nodes.

The problem as described here is an NP-complete tree traversal problem and is a variant of the "Knapsack problem", which is not surprising since we are generalizing the system that Funkhouser and Sequin showed to be a knapsack problem. The candidate sets from which only one element will be selected to be put in the knapsack are the set of representations associated to each meta-object. The knapsack size is the frame time per frame that the selected representations must not exceed. The cost of each element is the rendering cost associated to a representation. The profit of an element is the accuracy of the representation plus the benefit of the object with which it is associated.

To solve this problem we use the framework described in Section 4.1 and develop a model hierarchy building algorithm and a heuristic representation selection algorithm.

4.4 Design of the Model Hierarchy

We partition the entire model according to our formalization of the problem, and form a tree structure in which each node contains low-cost representations for the nodes it subsumes.

The structure that we use is a variation of an octree that is a bounding volume hierarchy, that can be used to cull objects against the viewing frustum and also serves as a rendering aid, since we can draw its nodes.

This tree is constructed in a bottom-up fashion instead of the traditional top-down recursive way, so that we can see which objects are being "clustered"[3] together as described in Section 5.

The criteria used to group objects takes into account only the proximity of objects and our model hierarchy building program is designed to cluster together nearby objects first in the way illustrated in the 2D example of Figure 4.

According to a user-supplied number of divisions in x, y, and z axis of the bounding box of the entire model an initial octree cell size and therefore tree depth is specified. We start by creating a "child list" that contains all the conceptual objects in the model with their bounding boxes. This initial list will correspond to the leaves of the tree. The child list is used to generate the next level up of the tree. For each level of the tree and for each cell in that level, we get the set of objects that are completely inside the cell. If this set is empty we move on to the next cell. Otherwise we compute the bounding box of the objects in the cell and discard it if the bounding box is already in the child list, since impostor representations for that set of objects had already been created. If it is not in the list we create impostor representations for the cluster inside the cell.

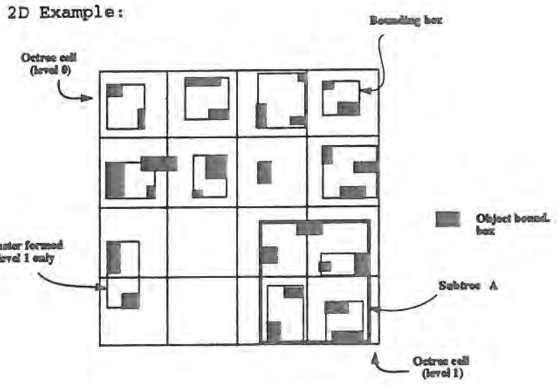


Figure 4: Generating the model hierarchy octree. Representations are generated for cells with more than one object.

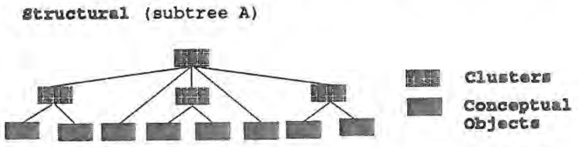


Figure 5: Subtree A as depicted on Figure 4.

In our implementation clusters are generated by creating texture maps[4] of the objects from given view angles and their generation is described in Section 5. After the impostor representations have been created, we make the cell point to its children and remove them from the child list. We then add the new cell to the end of the child list and repeat the process until we obtain a single cell with impostor representations for the entire model.

It is important to note that at each time we cluster objects we always take into account the actual objects that the cell subtends instead of previously computed clusters.

Note that cluster representations are generated only if there is more then one object totally inside each cell. Single objects inside a cell as well as objects on cell boundaries will be grouped in the next levels up in the hierarchy. Figure 5 shows the structure of subtree A depicted in Figure 4.

4.5 Traversal of the Model Hierarchy

Due to the NP-complete nature of selecting representations to render from the model hierarchy, we have devised a heuristic algorithm that quickly (in less than the frame time) traverses the model hierarchy. This algorithm selects representations to be rendered, accumulating rendering cost until the user-specified frame time is reached. When this occurs,

[3]What is meant by clustering is basically the generation of impostors for the group of objects.

[4]Actually, representations only need to obey the cost requirement stated in Section 4.3.

the algorithm stops and sends a list of representations to the graphics pipeline.

The tree traversal is top-down from the root node and first traverses the branches that contain the most "beneficial" nodes according to the benefit heuristic presented in Section 3.1.

The problem is that our per-object benefit heuristic associates benefit not to cluster representations but to representations for conceptual objects that are at the very bottom of the tree. High up in the hierarchy we do not know to which branches of the tree the most beneficial objects belong. Because of this, we have decided to break the selection of nodes to render in two phases as described below.

4.5.1 First Pass: Assign Initial Representation, Benefit, Visibility, and Cost.

In this first phase of the selection process, we recursively descend the model hierarchy in a depth-first manner and associate a benefit and visibility value with each node in the tree, and an initial drawable representation.

Since the leaves represent single objects, their benefits are computed as a weighted average of the factors intrinsic to objects as described in Section 3.1. The benefit value associated to a tree node is taken to be the maximum value of all the benefits of its children.

The visibility of nodes are computed by checking if the bounding box in eye-coordinates of the bounding box of the object intersects the viewing frustum. A node is said to be visible if at least one of its children is visible.

At a given point in the simulation a view dependent and a view independent representation for an object is selected using the criteria specified in Section 4.2. The rendering cost and accuracy of drawable representations that are stored with each representation in the model are used to select which of these two representations will be assigned to be the initial representation of the node. The representation that has the highest accuracy/cost ratio is selected to be the initial representation. In the next phase (described below), if there is still frame time left we try to improve on this initial choice.

After initial representations are selected to each of a node's children, the children's cost is stored with the node to be used in the next phase.

4.5.2 Second Pass: Best-First Tree Traversal.

In this phase, we use the information obtained in the previous phase for each node of the model hierarchy to implement an efficient 'best-first' tree traversal. The result of this traversal is a rendering list of drawable representations that is sent to the graphics hardware for rendering as shown in Figure 6.

To implement this strategy, we make use of a list of meta-objects organized in decreasing order of benefit (computed in the previous phase). We keep accumulating frame time as we select representations to render and whenever the time required to render the children of a node plus the total accumulated time so far exceeds the frame time we insert the representation for the node in the rendering list and move on to the next node.

The algorithm first explores the branches of the tree connected to the most beneficial nodes as follows: Start by inserting the root node in the list and setting the total rendering cost to be the cost of rendering the initial representation associated to the root node. We then process this list until

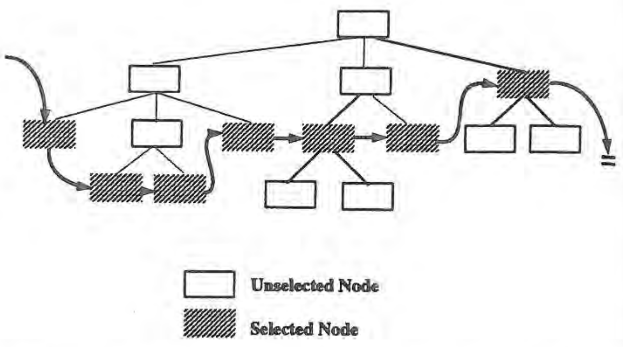


Figure 6: Tree representing the model hierarchy and the set of nodes to be rendered as a linked list.

it is empty. We remove the element in the front of the list and discard it if it is not visible.

If the node is a leaf node (containing a conceptual object) we check if there is still rendering time left to select a better representation for the object. In the positive case we select to render (insert in the rendering list) the next higher accuracy representation for the node and add its rendering time to the total accumulated rendering time.

In the case where the node contains representations for a cluster of objects, we check if instead of rendering the cluster representation we still have time to render all of its children, i.e., the total accumulated time plus the cost of rendering the node's children does not exceed the frame time. In the positive case, we insert each of its visible children in the list ordered by each ones benefit and add their cost to the total accumulated rendering time. Otherwise we insert the cluster's representation into the rendering list.

Note that at each point in this traversal, a complete representation of the scene is stored in the list of meta-objects and whenever there is frame time left to render the children of a node, before adding the cost of the children to the total accumulated cost we subtract the cost of the initial representation for the node.

4.6 Temporal Coherence

While navigating through the model the viewpoint can randomly get close or far away from "important" objects that require most of the frame time. This sometimes causes a seemingly random switch from a cluster representation to the representations of the actual objects (or vice-versa). The idea of using frame-to-frame temporal coherence as used by Funkhouser and Sequin, is used here to mininimize this effect by discouraging switching from representations for parent nodes to representations for children nodes. We keep a counter of the number of times the walkthrough program decided to switch from parent to children. The actual switching is only allowed if this counter exceeds a pre-fixed threshold. The delayed switching from children representations to cluster representations is not implemented since it would occur in a situation that most of the frame time has already been allocated and this delay would greatly reduce the frame rate.

5 Implementation

This research has resulted in the implementation of three programs on a four processor SGI Onix workstation with a RealityEngine board: the model hierarchy building and representation generation program, the cost and accuracy of representations measurement program, and the walkthrough program.

These programs are implemented in C++, use GL[8] for rendering, and have an X-Motif GUI to facilitate parameter changes for system evaluation.

5.1 Model Hierarchy Building and Representation Generation

The program that builds the model hierarchy implements the hierarchy building algorithm described in section 4.4 and opens two windows, as shown in Figure B. The right window displays the objects/clusters and compute texture maps for each of the sides of their bounding boxes while the left illustrates the process of building the hierarchy. In this image, the dots represents objects that were not "clustered" yet. The purple square with green dots is the bounding box of the objects (in green) that completely fit inside it and the "red" band is showing groups of objects already "clustered".

View dependent impostors such as texture maps are automatically obtained in the following way with the help of the graphics hardware:

1. Set up a viewpoint, a viewing direction, and an orthographic projection matrix.
2. Draw the object(s) in a completely black background and adjust the texture resolution[5] by scaling the object(s) inside the orthographic viewing volume.
3. Grab the resulting image from the window (right window in Figure B) and set the alpha component of black pixels to zero, so that if the objects have holes we can see through when they are rendered.

Average color boxes are also obtained in a similar fashion. The average color for each face is just the average of the rgb colors of all non-black pixels and the average area is the number of all non-black pixels in the face's image that is converted to an area in object coordinates.

The generation of a pseudo-texture map involves a preprocessing of the original image because if there are too many pixels on the image the rendering of the texture would require too many meshed triangles. Therefore, we successively shrink the original image by convolving it with a Gaussian filter that averages the RGB components of the pixels.

5.2 Cost and Accuracy of Representations Measurement

The cost of each representation is measured by selecting a specific representation and drawing it a number of times in order to get an average rendering time as shown in Figure C.

The accuracy of an impostor is measured using the procedure described in Section 3.2 and a table that describes how similar each of the representations is compared to the original image of the object for five directions around the object

[5]What ultimately determines the resolution of the texture map is the complexity (or granularity of details) that the object(s) exhibit(s) from a particular direction.

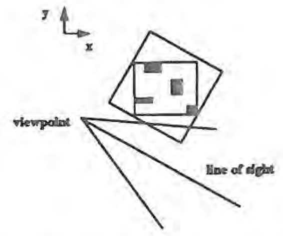


Figure 7: Checking the visibility of a set of objects against the viewing frustum.

is generated. One of the most immediate improvements we need to make is the use of more directions in this table.

5.3 Visual Navigation

The walkthrough program implements the framework described in Section 4.1 and the traversal algorithms described in Section 4.5. The computation of the representation to be rendered in the next frame is done in one processor while another one holds the graphics pipeline to render the current frame. Semaphores are used to synchronize the two processes.

The traversal algorithm assumes that visibility of bounding boxes can be determined quickly. This can be done by first computing the bounding box in eye-coordinates of the object's bounding box. We then compute its intersection with a box formed by extending the slice of the viewing frustum corresponding to the farthest z-value of this box to its nearest z-value. This visibility test can return true even though no object inside the cluster is also inside the viewing frustum as shown in Figure 7.

This problem is solved by the first phase of the traversal algorithm since it marks a cluster as visible if and only if at least one of the objects that it represents is inside the viewing frustum. If computing the visibility of individual objects are taking too much time we can use a faster test and check if spheres enclosing groups of objects intersect the viewing frustum.

5.4 Performance

Our test model has around 1.6 million polygons and during our tests we have constrained the number and size of texture maps generated by the hierarchy building program to the available texture memory of one megatexel (one million texture pixels) by selecting appropriate octree cell sizes and adjusting the resolution of the texture representation for objects and clusters.

For this model we were able to keep a frame rate of around 16 frames per second (fps) for a target frame rate of 30 fps throughout the simulation without too much degradation in image quality. Figure D shows the image seen by the observer (left) and a top view of the the same scene showing where clusters are being displayed (right).

Figure 8 shows the user mode (right) and real time (left) throughout a simulation path of the model. The user time graph shows that our estimation of cost and rendering algorithm is achieving the goal of keeping a uniform and high frame rate. The real time graph show spikes due to random interrupts and a gap with respect to the 1/30 line due to smooth LOD switching using transparency blending.

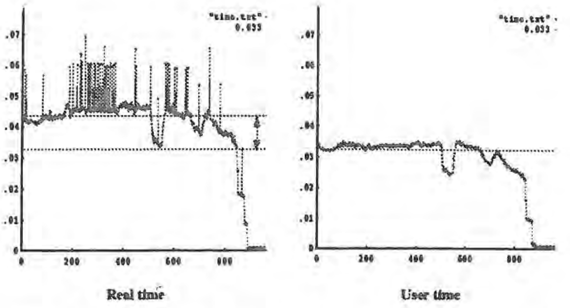


Figure 8: Plot frame versus frame time with (left) and without (right) smooth LOD switching.

These interrupts can be minimized by mechanisms such as processor isolation, interrupts redirection, processor locking and so on as described in [9].

The same model, without the model hierarchy, takes around 1 frame per second for certain viewpoints in our test path.

6 Limitations

One limitation of this system is the number of texture maps that can be used to represent objects and clusters. As the system uses more texture maps to represent clusters and individual objects, the chance of a texture-cache miss increases. A cache miss results in an unpredictable interrupt that will invariably defeat the purpose of a predictive system. Future methods of intelligent prefetch of textures that are likely to be needed could make texture cache misses much less likely, and thus allow the use of many more textured impostors.

We have not addressed the illumination of the environment. Although the illumination of a complex environment can be computed using the radiosity method in a view independent fashion the shading attributes of objects (adding specular highlights) and clusters would need to be incorporated to their representations. Instancing of objects would not be practical since two identical objects in different locations in the model would have different shading attributes. As the size of texture memory increases these problems will become less serious, but they will not go away.

The most serious limitation in our current implementation is that our tree traversal requires that a cluster know something about the benefits of its children, so all primitives are visited once per frame in the first pass of the algorithm, and therefore it is $O(n)$, where n is the number of objects. Our traversal algorithm is top-down, so there is no reason it could not be $O(\log n)$ if a more intelligent traversal algorithm is used.

7 Conclusion

We have presented a way of using clusters of objects to improve the performance of an LOD-based visual navigation system. When there are too many visible LODs to render in real-time, we render single texture-mapped cluster primitives in place of groups of individual LODs. The techniques used to generate clusters can also be used to generate a particular type of textured LODs for single primitives. We have also discussed some limitations of the object-based benefit heuristic, and extended it to account for the variability of an LOD's quality as the view angle changes.

The main lessons to be drawn from this work are that the predictive framework of Funkhouser and Sequin extends well to a hierarchical version of the LOD concept, and that precomputed visibility information is not essential for efficient visual navigation programs.

8 Acknowledgments

Thanks to Ken Chiu and Aaron Yonas for their suggestions on the draft of this paper. Thanks to Ken Chiu and Paul Bourke for the model of a tree and a town house, respectivelly, used in the color plates. Thanks to the Brazilian government agency CAPES, for providing the first author the financial support to conduct this research. Thanks to the Research and University Graduate Schools (RUGS) Research Facility Fund (RFF) and the NSF CDA-92-23008 grants that provided the graphics workstations that were used in this research. The second author was also supported by NSF RIA grant CCR-92-09457.

References

[1] John M. Airey, John H. Rohlf, and Jr Frederick P. Brooks. Towards image realism with interactive update rates in complex virtual building environments. *Computer Graphics*, pages 41–50, 1990.

[2] Kurt Akeley. Reality engine graphics. *Proceedings of SIGGRAPH'93 (Anaheim, California, August 1-6, 1993). In Computer Graphics Proceedings, Annual Conference Series, 1993, ACM SIGGRAPH*, pages 109–116.

[3] Thomas A. Funkhouser and Carlo H. Sequin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environmnets. *Computer Graphics*, pages 247–254.

[4] Thomas A. Funkhouser, Carlo H. Sequin, and Seth Teller. Management of large amounts of data in interactive building walkthroughs. *Proceedings of the 1992 Symposium on Interactive 3D Graphics (Cambridge, Massachusetts, March 29 - April 1, 1992), special issue of Computer Graphics, ACM SIGGRAPH*, pages 11–20, 1992.

[5] Paulo Maciel. Visual navigation of largely unoccluded environments using textured clusters. *Ph.D. Thesis*, January 1995. Indiana University, Bloomington.

[6] John Rohlf and James Helman. Iris performer: A high performance multiprocessing toolkit for real-time 3D graphics. *Proceedings of SIGGRAPH'94 (Orlando, Florida, July 24-29, 1994). In Computer Graphics Proceedings, Annual Conference Series, 1994, ACM SIGGRAPH*, pages 381–394.

[7] Harvey R. Schiffman. *Sensation and Perception an Integrated Approach.* John Wiley & Sons, New York, 1990.

[8] Inc. Silicon Graphics. *Graphics Library Programming Guide, Volumes I and II*, 1992.

[9] Inc. Silicon Graphics. *React In Irix: A description of real-time capabilities of Irix v5.3 on Onyx/Challenge multiprocessor systems.*, 1994.

Guided Navigation of Virtual Environments

Tinsley A. Galyean

MIT Media Lab
Cambridge, MA. 02139
tag@media.mit.edu

ABSTRACT

This paper presents a new method for navigating virtual environments called "The River Analogy." This analogy provides a new way of thinking about the user's relationship to the virtual environment; guiding the user's continuous and direct input within both space and time allowing a more narrative presentation. The paper then presents the details of how this analogy was applied to a VR experience that is now part of the permanent collection at the Chicago Museum of Science and Industry.

1. INTRODUCTION

Today's Virtual Reality (VR) technology provides us with an opportunity to have experiences that would otherwise be impossible. We can smoothly and continuously interact while immersed in environments that would be inaccessible or impossible to experience. In these environments, we are free to roam and explore. architectural walk throughs for example, scientific visualization, and even games like DOOM place us in alternative worlds while giving us methods for navigating these virtual spaces. These methods allow smooth and continuous interaction that can immediately influence the constantly changing presentation, but they rely on the user's actions and thoughts to bring structure to the experience. If any narrative structure (or story) emerges it is a product of our interactions and goals as we navigate the experience. I call this emergent narrative. In some applications this complete freedom to explore is appropriate. However, there is an alternative. This is the process of empowering the author to bring structure to the experience, which makes this medium more appropriate for applications such as teaching, storytelling, advertising and information presentation. To do this, we will need to balance the interaction (exploration) with an ability to guide the user, while at the same time maintaining a sense of pacing or flow through the experience. This type of guidance is the process of a providing narrative structure. Like a narrative presentation any solution must guide the user both temporally and spatially.

2. PREVIOUS WORK

Virtual environment navigation has mainly consisted of building new methods and technologies that allow the users to control the position and orientation of the virtual camera, through which they see the virtual world. Early work in camera control (even before the advent of VR technology) focused on specifying camera movements over a path. [1, 4] In an effort to address the needs of animation production and does not address the issue of interactive camera control.

A number of different researchers have addressed the issues behind camera control for manipulation and/or exploration applications. [2, 5] All of these methods focus on providing better ways for the user to roam free, exploring within the virtual world. It is this ability for the user to directly control his/her place in the virtual world that is so often synonymous with the words "virtual reality." While these methods couple smooth, continuous interaction with the smooth and continuous presentation available in real-time computer graphics environments, they do nothing to guide the user. There is no room for an author's intentions to influence the experience. Therefore there is no narrative structure.

Researchers in interactive narrative working with linear material like digital video have worked to unfold it in order to provide a non-linear environment for the user [3]. Shots are interactively laced together into sequences and these sequences tell a story. Because shots are the smallest building blocks available, the interaction intermittently guides how these shots are laced together.

The traditional analogy of how these types of interactive experiences are structured is often referred to as the branching analogy. Each branch represents a linear segment traversing part of the narrative space. A linear segment is played until the next node is reached. It is at these nodes where options are provided. The advantage of branching is that the experience does have a narrative structure, the interaction is guided. The disadvantage is that one can interact only at the nodes thereby chopping up both the interaction and the presentation.

The goal set forth is to find a way to marry the advantages of immersive VR experience with the advantages of narrative structure. How do we allow the smooth, continuous interaction and presentation, to coexist-exist with the structural and temporal qualities of narrative (plot and pacing)? In other words, how do we balance the notion of interaction with guidance (telling).

3. THE RIVER ANALOGY

Here I propose an alternative analogy for navigating virtual spaces. Instead of linking a sequence of branches and nodes, or giving the user free rein, I suggest that the navigation paths be more like a river flowing through a landscape. The user is a boat floating down this river with some latitude and control while also being pushed and pulled by the pre-defined current of the water. Like the branching structure this approach constrains the audience's movement through the space to interesting and compelling paths. But there are some unique advantages to this approach: the flow of the experience, the continuous input of the rudder, and multiple levels of structure.

The river analogy assures an uninterrupted flow. When in a boat you float down the river even when you are not steering. The presentation is continuous regardless of whether or not there is input. The amount of control you have over the boat varies with the properties of the river. If the rapids increase, you move faster with less room for swinging from side to side. Alternatively, the pace can slow and the river can widen giving room to steer from one bank to the other.

In the river analogy the boat's rudder can be likened to audience input. A rudder takes input continuously. The amount of influence may vary depending on the water conditions but you can always provide the input. It is also the case that the rudder may have both an immediate and a long term impact on the navigation. How the rudders are used can determine both your local position within the river, but also a more global position, such as which fork in the river your boat might take.

The river provides two levels of guiding structure. First is the local structure of the river including the water flow, rocks in the river, the width between the banks, etc. Second, is the global structure, including both the path the river flows and the forks that separate and/or rejoin. The audience input has influence on how both levels of this representation are navigated. The rudder or input can steer between the banks while the position of the boat when a fork is reached will dictate which part of the fork the audience will travel.

Like a river, a guiding navigation method should guide without interruption of the presentation. This creates a sense of interaction by constantly accepting user input and guiding it with a higher level, longer term structure.

4. APPLICATION OF THE RIVER ANALOGY

A highlight of the Chicago Museum of Science and Industry's new exhibit, *Imaging the Tools of Science*, is the virtual reality experience. The primary goal of this exhibit was to expose and educate the visitor to what VR technology is and can do. Any experience that was going to be successful, was going to be highly constrained by the issues inherent in bringing an immersive experience to a public place like the museum. In a museum setting it is necessary to limit the amount of time a person spends, provide an interface that keeps people from getting lost and frustrated, while at the same time making them aware that they have some direct and immediate control over how they move through the environment. To meet these dèmands it was decided that the experience would be between 2 and 3 minutes long with a clear beginning, middle, and end. This allowed the user to feel they had a complete experience while allowing the museum to predict how quickly they could move people through the exhibit. These constraints required the user's navigation to be guided through the virtual world, and the river analogy helped us address these issues.

In this application, the analogy of the river was taken quite literally. We defined a path through the virtual space as the river. The user was then guided through the space much like a water-skier behind an invisible boat. The boat or anchor moves along the path at a rate that varies as specified by the creator of the experience (the author). The user is then tethered to the anchor by a spring that constantly pulls them along. Meanwhile the user is free to look in any direction he or she chooses. Figure 1 shows the model we used. This model gives the user direct control over where they are looking while at the same time giving them indirect control over their local position. Looking in a given direction will impart some force in that direction and allow the user to swing over in that direction moving closer to the object they are watching. At the same time the boat continues to pull them along the journey, maintaining a sense of pacing and flow.

There are a series of parameters that can change the nature of this interface: the current and desired speed of the anchor, the amount of thrust the user is imparting, and the spring and damping constants. In this implementation, all of these values are free to change throughout the experience. The changes are encoded at

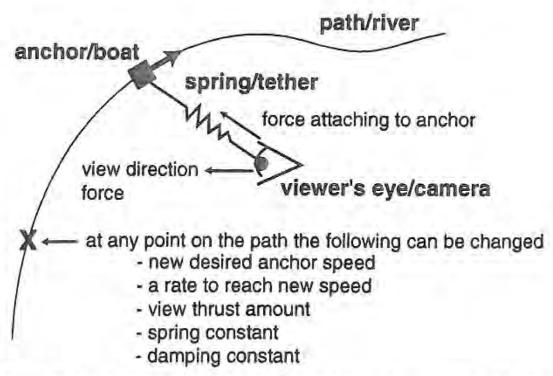

path/river
anchor/boat
spring/tether
force attaching to anchor
view direction force
viewer's eye/camera
at any point on the path the following can be changed
- new desired anchor speed
- a rate to reach new speed
- view thrust amount
- spring constant
- damping constant

Figure 1: An application of the River Analogy, consisting of a number of different parts: the anchor moving along the path, a spring attaching the user position to the anchor, a thrust imparted by the user dictated by the direction the user is looking, and a general viscous damping to prevent the user from oscillating about the anchor position.

locations along the path, allowing the author to specify over which areas of the journey the user is more or less free to roam. For example, as the user approaches a larger open space the author many choose to slow down the anchor, decrease the spring and damping constants, and increase the viewer thrust allowing the user more latitude and time to explore. Alternatively, the author might focus the experience by increasing the spring constant, speeding up the anchor, and reducing the thrust.

5. CONCLUSION

It is clear that there are VR application for which the current methods of navigation are not sufficient. Some of these applications suggest the need for a method to guide the user as s/he navigates the virtual landscape. The River Analogy provides a way of thinking about how the author's intentions can steer the interaction given by the user to create a guided navigation. This paper has presented this analogy and one particular application of this analogy to an existing public VR exhibit. This work only begins to touch on the potential of guided interaction for virtual environments.

ACKNOWLEDGEMENTS

The Art Technology Group including: Martin Friedmann, Stephen Clark, Andy Schwarz, Keith Baccki, Andy Hong, David Rose, Joe Chung, Jeet Singh. Martin in particular for outstanding work on implementation. Glorianna Davenport (Media Lab) for her help and support. Steven Drucker for the many inspiring conversations.

REFERENCES

1. Bartels, R., J. Beatty, and B. Barsky, *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*. Morgen Kaufmann, Los Angeles, 1987.

2. Brooks, F. P. Grasping Reality Through Illusion -- Interactive Graphics Serving Science. *CHI '88 Proceedings*, Special Issue of SICHI Bulletin,1988, 1-11.

3. Davenport, G., T. Aguierre-Smith, and N. Pincever, Cinematic Primitives for Multimedia, Computer Graphics & Applications, (July 1991), 67-73.

4. Shoemake, K. Animating Rotation with Quaternion Curves. Proceeding of SIGGRAPH '85. (San Francisco, California, July 22-26, 1985). In Computer Graphics 19, 3 (July 1985), 245-254

5. Ware, C. and S. Osborne. "Exploration and Virtual Camera Control in Virtual Three Dimensional Environments," Proceedings of the 1990 Symposium on Interactive 3D Graphics (Snowbird, Utah, March 25-28, 1990),special issue of Computer Graphics, ACM SIGGRAPH, New York, 1990, 175-184

Portals and Mirrors:

Simple, Fast Evaluation of Potentially Visible Sets

David Luebke and Chris Georges
Department of Computer Science
University of North Carolina at Chapel Hill

Abstract

We describe an approach for determining potentially visible sets in dynamic architectural models. Our scheme divides the models into cells and portals, computing a conservative estimate of which cells are visible at render time. The technique is simple to implement and can be easily integrated into existing systems, providing increased interactive performance on large architectural models.

Introduction

Architectural models typically exhibit high depth complexity paired with heavy occlusion. The ratio of objects actually visible to the viewer (not occluded by walls) to objects theoretically visible to the viewer (intersecting the view frustum) will usually be small in a walkthrough situation. A visibility algorithm aimed at reducing the number of primitives rendered can exploit this property. Following prior work [1,2,3], we make use of a subdivision that divides such models along the occluding primitives into "cells" and "portals". A cell is a polyhedral volume of space; a portal is a transparent 2D region upon a cell boundary that connects adjacent cells. Cells can only "see" other cells through the portals. In an architectural model, the cell boundaries should follow the walls and partitions, so that cells roughly correspond to the rooms of the building. The portals likewise correspond to the doors and windows through which neighboring rooms can view each other.

Given such a spatial partitioning of the model, we can determine each frame what cells may be visible to the viewer. By traversing only the cells in this potentially visible set (PVS), we can avoid submitting occluded portions of the model to the graphics pipeline. What cells comprise the PVS? Certainly the cell containing the viewpoint is potentially visible. Those neighboring cells which share a portal with the initial cell must also be counted as potentially visible, since the viewer could see those cells through the portal. To this we add those cells visible through the portals of these neighbors, and so on. In this manner the problem of determining what cells are potentially visible to the viewer reduces to the problem of determining what portals are visible through the portals of the viewer's cell.

luebke@cs.unc.edu (919) 962-1825
georges@cs.unc.edu (919) 962-1789
CB# 3175 Sitterson Hall; UNC, Chapel Hill, NC 27599-3175

Our system makes this determination dynamically at render time. Rather than finding the exact PVS for each cell as a preprocess, we postpone the visibility computation as long as possible. This type of dynamic evaluation of portal-portal visibility is not new. Earlier efforts have centered on precisely determining sightlines through portals; our method offers a less exact but much simpler alternative. The algorithm has been implemented on the Pixel-Planes 5 graphics computer at the University of North Carolina and provides a substantial speedup on a sample model of 50,000 polygons.

Previous Work

Jones [1] explored the subdivision of geometry into cells and portals as a technique for hidden line removal. In his algorithm, models are manually subdivided into convex polyhedral cells and convex polygonal portals. The subdivision is complete in the sense that every polygon in the dataset is embedded in the face of one or more cells. Rendering begins by drawing the walls and portals of the cell containing the viewer. As each portal is drawn, the cell on the opposite side of the portal is recursively rendered. In this way the cell adjacency graph defined by the partitioning is traversed in depth-first fashion. The portal sequence through which the current cell is being rendered comprises a convex "mask" to which the contents of the cell are clipped. If the intersection of a portal with the current mask is empty, the portal is invisible and the attached cell need not be traversed.

More recent work has abandoned the attempt to compute exact visibility information, focusing instead on computing a conservative PVS of objects that *may* be visible from the viewer's cell. The graphics pipeline then uses standard Z-buffer techniques to resolve exact visibility. Airey [2] was the first to use a portal-based approach effective in architectural environments. He described multiple ways to approach the problem of determining cell-to-cell visibility, including ray-casting and shadow volumes. Teller [3] has taken the concept further and found a closed-form, analytic solution to the portal-portal visibility problem. Using 2D linear programming to test portal sequences against arbitrary visibility beams, Teller computes a complete set of cell-to-cell and cell-to-object visibilities in a preprocess. At render time this PVS is further restricted according to which portals are actually visible. Teller's approach is mathematically and computationally complex, requiring hours of preprocess time for large models [3].

Motivation

Such a large preprocessing cost may be inappropriate for interactive applications. For example, architectural walkthroughs are often used for revision purposes. A visualization of a building under design is more valuable to an architect if inquiries of the type "What if I move this wall out ten feet?" can be answered immediately. Adding portals, moving portals, and redistributing cells boundaries will all be common operations in an interactive architectural design application. To take full advantage of the static visibility schemes mentioned above, each of these would require a potentially lengthy PVS recalculation best done off-line.

Envisioning such an application as our final goal, we decided to focus on improving the dynamic visibility determination. Jones' algorithm finds the exact intersection of 2D convex regions, requiring O(n lg n) time for portal sequences with n edges. Teller's linear programming approach computes only the *existence* of an intersection, and runs in time linear in the number of edges. We sought a dynamic solution that would also run in linear time and would integrate easily into existing systems.

Faster Dynamic PVS Evaluation

We use a variation of Jones' approach that employs bounding boxes instead of general convex regions. Our scheme first projects the vertices of each portal into screen-space and takes the axial 2D bounding box of the resulting points. This 2D box, called the *cull box*, represents a conservative bound for the portal; that is, objects whose screenspace projection falls entirely outside the cull box are guaranteed not to be visible through the portal and may be safely culled away. As each successive portal is traversed, its box is intersected with the aggregate cull box using only a few comparisons.

During traversal the contents of each cell are tested for visibility through the current portal sequence by comparing the screenspace projection of each object's bounding box against the intersected cull box of all portals in the sequence. If the projected bounding box intersects the aggregate cull box, the object is potentially visible through the portals and must be rendered. Since a single object may be visible through multiple portal sequences, we tag each object as we render it. This *object-level culling* lets us avoid rendering objects more than once per frame.

Alternatively, we can render each object once for every portal sequence which admits a view of the object, but clip the actual primitives to the aggregate cull box of each sequence. Under this *primitive-level clipping* scheme objects may be visited more than once, but since the portal boundaries do not overlap, no portion of any primitive will be rendered twice. Typically object-level culling will prove more efficient, but for objects whose per-primitive rendering cost far exceeds their clipping cost, primitive-level clipping provides a viable option.

Implementation

We have implemented this approach on Pixel-Planes 5, the custom graphics multicomputer developed at the University of North Carolina. The traversal mechanism treats portals as primitives to be rendered. Each portal consists of a polygonal boundary and a pointer to the adjacent cell; when a portal is encountered during traversal we test its axial screenspace bounding box against the current aggregate cull box. If the intersection is nonempty, we use it as the new aggregate cull box and recursively traverse the connected cell.

We feel that modeler integration is crucial to this problem of interactive model revision. If an architect wishes to move a wall or broaden a doorway, the modeling system should be able to make the change quickly and broadcast that change to the graphics system. In our system the spatial partitioning of the model into cells and portals is directly embedded in the modeler's representation. Portals are treated as augmented polygons, each tagged with the name of the attached cell. Cells are simply logical groupings in the modeler's hierarchy and need not necessarily be convex. We have found this quite convenient when constructing models; each room typically corresponds to a cell and it takes only seconds to add and move a portal, or to reshape a cell. We have already adapted two commercial modelers to our system, which speaks to the simplicity of the integration process.

Results

We have tested our system on a subset of the UNC Walkthrough project's model of Professor Fred Brooks' house, comprised of 367,000 radiositized triangles. The speedup obtained by this visibility algorithm, like the speedup obtained by similar schemes, is extremely view-and model-dependent. Over a 500-frame test path through the model, the frame rate using PVS evaluation ranged from just over 1 to almost 10 times the frame rate of the entire unculled model. For typical views the dynamic PVS evaluation culled away 20% to 50% of the model. It should be emphasized again that these numbers are specific to the model and view path, but they certainly indicate the promise of the algorithm as a simple, effective acceleration technique.

Ongoing and Future Work

Efficiency could be further increased by applying *obscuration culling* to portals [4]. This scheme tests potentially visible items against an "almost complete" Z-buffer before rendering. This would allow the 'detail' objects in each cell as well as the occluding cell walls to block portals, potentially reducing the PVS. The Pixel-Planes architecture makes obscuration culling of portals feasible, and we are currently exploring this possibility.

Teller mentions that the concept of portals may be extended to mirrors [3]. Under this scheme mirrors are treated as portals which transform the attached cell about the plane of the mirror; this has the advantage of automatically restricting the PVS seen through the mirror. Though conceptually simple, mirrors introduce many practical difficulties which require additional clipping by the rendering engine to resolve. For example, geometry behind the mirror must not appear in its reflected "world," and reflected geometry must not appear in front or to the side of the mirror.

A special case that avoids these problems can be constructed by embedding the mirror in an opaque cell boundary (for example, a wall-mounted mirror in a bathroom), and we have implemented such mirrors (Plate 1). The concept of an immovable mirror fits poorly with our goal of interactive, dynamic environments, however, so we have focused on the more general case. Clipping is complicated further by mirrors that overlap in screenspace, and further still by mirrors which recursively reflect other mirrors. At present our system allows static mirrors, which can reflect each other to arbitrary levels of recursion, or more general "hand-held" mirrors, (an example of free-moving portals), which permit one-bounce reflections. We are currently working on the dynamic, fully recursive case.

Acknowledgments

The authors would like to extend their sincere thanks to Mike Goslin, Hans Weber, Power P. Ponamgi, Peggy Wetzel, and Stump Brady. This work was supported by ARPA Contract DABT63-93-C-C048.

References

[1] Jones, C.B. A New Approach to the 'Hidden Line' Problem. The Computer Journal, vol. 14 no. 3 (August 1971), 232..

[2] Airey, John. Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations. Ph.D. thesis, UNC-CH CS Department TR #90-027 (July 1990).

[3] Teller, Seth. Visibility Computation in Densely Occluded Polyhedral Environments. Ph.D. thesis, UC Berkeley CS Department, TR #92/708 (1992).

[4] Greene, Ned, Kass, Michael, and Miller, Gavin. Hierarchical Z-Buffer Visibility. Proceedings of SIGGRAPH '93 (Anaheim, California 1993). In Computer Graphics Proceedings, Annual Conference Series, 1993, ACM SIGGRAPH, New York 1993, pp. 59-66.

Interactive Playing with Large Synthetic Environments

Bruce F. Naylor
AT&T Bell Laboratories
Murray Hill, NJ 07974

Introduction

Until recently, opportunities to experience large synthetic environments have been limited primarily to expensive training simulators. However, with the advent of "location based entertainment" at theme parks and even CD-ROM based games for PC's, these kinds of experiences are beginning to be made available to the general public as well. The constraints on the possibilities for appealing "content" arises from the technological capabilities that are possible for a given performance level on a given platform. Currently, for 3D graphics, performance is closely tied to the number of texture mapped polygons that can be rendered for each frame as well as the rate at which collisions of various kinds can be computed.

Large synthetic environments require at least tens of thousands of polygons, and could easily entail millions. However, for each image, only a small subset of these polygons are typically required to synthesize the image. Similarly, collisions between two objects, or between a viewer and the environment, involve an even smaller subset. The task then for efficient geometric computations is to, if possible, quickly identify the relevant subset. The principal methodology for finding the minimal subset of polygons is to use spatial search structures, such as regular grids, octrees, or binary space partitioning trees. In this paper, we describe briefly the current status of our efforts at using binary space partitioning trees for navigating through and playing with large environments, including rendering and collision detection, as well as permitting interactive modifications of the environment using set operations that should prove appealing for entertainment applications.

Partitioning Trees (or BSP Trees) [Fuchs, Kedem, and Naylor 80] differ from regular grids in that they are hierarchical (multi-resolution), and from octrees in that the method of space partitioning requires not only determining when to partition, but where, as well. The absence of a restriction on the planes used in partitioning trees obviates the need for a distinction between the spatial search structure and the representation of polyhedra by using the planes containing faces to partition space. A single tree, representing some rigid object for example, can be transformed with affine and perspective transformations by only transforming the plane equations; thereby not changing the tree structure. The tree provides a visibility order for rendering objects with any mix of transparent and opaque surfaces, and the near-to-far ordering that can be used for pruning away fully occluded subtrees. In addition, it can be used for efficient solid clipping with a view-volume, for computing shadows and/or global illumination, for intersecting rays with an object, and for determining the location of points (representing, for example, sprites in computer games) in an environment. Spatial relations between two objects can be computed efficiently by merging their respective trees into a single tree [Naylor, Amanatides and Thibault 90]. This provides, on the modeling side, set operations and collision/interference detection. For rendering, tree merging determines inter-object visibility, analogous to merging sorted lists in Merge Sort, which provides the proper ordering required for transparent objects whose convex hulls interpenetrate. It can also be used to discover that a moving object has become totally occluded by another object and so need not be drawn. In addition, tree merging can be used to cast shadows from one object onto another.

Visibility Culling of Large Environments

The most important computation for efficiently navigating through large environments is conceptually a rather simple and familiar one: clipping to the view-volume. However, approaching this using solely the traditional graphics pipeline for polygon clipping in an O(n) process. While any spatial search structure can be used to accelerate this computation, our use of partitioning trees has several consequences. The first is that partitioning trees provide a representation of space as a hierarchy of convex bounding volumes that is highly adaptive to the contents of the space. Thus, unlike a regular grid, the subdivision can be very fine in areas of high level of detail without compromising the representation of large open areas by a few large cells. Our method of building trees [Naylor 93], based on minimizing the expected cost of search operations, produces such trees, since large open regions are treated as being highly probable and will have short paths. This is completely analogous to Huffman codes, in which the number of bits assigned a code, i.e. the path length in the Huffman code binary tree, is inversely related to the probability of that code being used. Here, largeness is treated as being positively correlated to the probability of intersecting a region.

Given a tree representation of the environment, constructed off-line, together with a particular view, we find the subset of the environment within the view-volume by first constructing a Partitioning Tree representation of the view-volume [Naylor 92a]. Since we are using a tree for this, the view-volume can have any polyhedral geometry, and so need not be limited to a truncated pyramid. We also provide solid clipping; that is, the intersection of the view-volume with the solid environment will be displayed by polygons having the attributes of the material with which the view-volume is intersecting. We use the general tree merging machinery for view-volume clipping/culling. However, we do not produce a new tree corresponding to intersection of the environment with the view-volume. Rather we combine the view-volume intersection with the visibility priority traversal so that the polygons are transmitted to the polygon drawing stage as the intersection operation proceeds, thus obviating the need for creating an intermediate clipped tree.

Another very effective but very simple method of reducing the computational requirements is to combine simulation of fog with placement of the far clipping plane.

By setting the fog parameters so that total fog color occurs at (or slightly before) the far-plane, the presence of the far-plane is effectively obscured. We use this as a simple mechanism for maintaining a target frame rate of, for example, 10 frames per second. Whenever the frame rate is too slow we bring the far-plane closer to the viewer at a rate determined by the frame rate deficiency. So if one turns the corner from a simple to a complex view, the fog rolls in over the next dozen or so frames until the frame rate is restored. Similarly, we move the far-plane away when the frame rate will allow this. Both of these movements are constrained by min and max values.

Collision Detection in Large Environments

In the design of entertainment applications, it is often required to know whenever some moving object collides with the environment or with some other object. Such collisions can be detected by merging the partitioning tree representing an object with a "model-tree", whose initial value is (a copy of) the environment-tree. Since each tree can be interpreted as a hierarchy of convex bounding volumes, merging two trees is simply merging together two hierarchies of bounding volumes in a top-down (largest-to-smallest) and recursive manner. Whenever the region containing a subtree of one tree is found to not intersect a region containing a subtree from the other tree, no comparisons between the contents of those two subtrees need be made. Empirically, this seems to happen around 50-75% of the time. This is, of course, the mechanism that reduces the computation to be generally less than $O(n^2)$. In fact, when the objects are sufficiently separated so that the first bounded/finite regions in each tree do not intersect, then the computation can be done in O(1), and is analogous to testing two bounding volumes for intersection.

Since the tree has cells labeled as being in-cells or out-cells, collisions occur whenever one of the operands in the recursive process reaches an in-cell while the second operand is either an in-cell or a subtree containing in-cells. For each in-cell, we maintain an identifier field (an integer) which can be used to identify the entity with which the collision is occurring. We also maintain identifier fields at each internal node which is set to the identifier at the in-cells of its subtree whenever they are all the same; otherwise it is null. This permits extracting the identifier without descending the entire subtree, which it essential for obtaining sub-linear performance. Each pair of identifiers is added to a collision report list, which is maintained in sorted order to avoid adding duplicates.

Visual Effects

A number of special effects appropriate for entertainment applications can be achieve efficiently using tree merging as the basic operation. One class uses traditional set operations. We have used subtraction to blast holes in walls and buildings as a result of firing a gun, as well as to simulate tunneling with a drill. One important aesthetic consequence of this is not only the simulation of these effects, but also the complex and unexpected geometry that is created. Such user generated variety reduces the burden on a game designer to always meet the desire for new experiences; here the user gets to participate is creating his/her own variety. In principal, the other set operations, union, intersection and symmetric difference, could be employed to modify the environment in interesting ways.

Another approach we have developed is to combine transparency with tree merging to create two new effects that only temporarily modify the environment (usually for only one frame). One of these creates the effect of a force field slicing though the environment. This uses a temporary union between a very transparent object, such as an elongated cylinder, and the environment in which the object takes precedence over the environment. The part of the environment inside the transparent object is temporarily removed, while the faces corresponding to the intersection between the object's surface and the environment takes on the attributes of the environment. This gives the appearance of a force field moving through walls, for example. The second technique provides an x-ray effect. It differs from the force-field effect only in one aspect: rather than remove the portion of the environment that is inside the transparent object, these faces are also made transparent. Thus, one can look though walls while retaining the sense that the wall is still there (see color plate).

A more traditional visual effect is the use of shadows. Our current method constructs shadows for each object independently. This is achieve by creating shadows volumes for each face in priority order and adding these volumes to the object-tree [Naylor 92b]. This transforms what were formerly out-cells into partitioned regions that are homogeneous with respect to visibility of lights, visibility being an additional property of a region. Currently, each light is assigned one bit to indicate visibility of that light. When trees are merged, shadows are cast onto other objects by classifying the faces of one with respect to the other tree. This only requires extending the normal tree merging process to maintain the visibility property. So for example, whenever the recursive process reaches a cell, the visibility of a cell is transmitted to the other operand by performing an "and" operation between the cell's visibility field and those of the other operand.

References

[Fuchs, Kedem, and Naylor 80]
H. Fuchs, Z. Kedem, and B. Naylor, "On Visible Surface Generation by a Priori Tree Structures," **Computer Graphics** Vol. 14(3), pp. 124-133, (June 1980).

[Naylor, Amanatides and Thibault 90]
Bruce F. Naylor, John Amanatides and William C. Thibault, "Merging BSP Trees Yields Polyhedral Set Operations", **Computer Graphics** Vol. 24(4), pp. 115-124, (August 1990).

[Naylor 92a]
Bruce F. Naylor, "Interactive Solid Modeling Via Partitioning Trees", Proceeding of *Graphics Interface* , pp. 11-18, (May 1992).

[Naylor 92b]
Bruce F. Naylor, "Partitioning Tree Image Representation and Generation from 3D Geometric Models", Proceeding of *Graphics Interface* (May 1992).

[Naylor 93]
Bruce F. Naylor, "Constructing Good Partitioning Trees", Graphics Interface '93, Toronto CA, pp. 181-191, (May 1993).

Of Mice and Monkeys: A Specialized Input Device for Virtual Body Animation

Chris Esposito
Virtual Systems Group
Boeing Computer Services

W. Bradford Paley
JueyChong Ong
Digital Image Design, Inc.

Abstract

This paper discusses the motivation, design, implementation, and some sample applications of a new input device, called the Monkey™, that can be used for real-time control of digital human models.

1. Introduction

Software models of realistic human figures are useful in a wide variety of areas, from TV commercial animation to human factors analyses of reachability and maintenance procedures in aircraft design. A common requirement across all of these areas is the quick and easy manipulation of the figure into desired postures for keyframes or through motion sequences for interaction with the surrounding environment. This paper describes a new input device, called the Monkey, that has been designed to make these manipulations faster and easier than they have been, discusses several of the issues surrounding it's rationale, design, and prototype implementation, and briefly describes some of the applications of this device that are in progress.

1.1 Motivation

Realistic human figures are complex structures that have many joints, with each joint having 1 or more rotational degrees of freedom. Each of these degrees of freedom has an associated constraint or limit that determines how far the associated joint can flex around the specified axis. To further complicate matters, some of these constraints interact (in a non-linear way) with other constraints. For example, a rotation at the hip that brings the knee up and closer to the chest will change the allowable joint rotation limits at the knee so that the knee is forced to flex as the thigh is raised.

Several kinds of devices have been used to interactively control the posture of a human model. One of these is the ubiquitous mouse, which has been used in the Jack system from the University of Pennsylvania [1], and the Safework system from Genicom Consultants [2], to name two of several available systems. In these systems, some portion of the body is selected with the mouse and then interactively dragged around the screen following the mouse, as far as the relevant joint limits will allow. In the Safework system, for example, there are seven "handles" that are the entry points to the inverse kinematics system. Once one of these handles is selected, all translation of the handle and the associated joints and segments is done in the plane of the screen, regardless of the orientation of the figure. Similar techniques are used in computer animation software (e.g., Wavefront's Kinemation).

The benefits of being able to manipulate human models in virtual environments are sufficiently great that these systems are a success, despite some of their difficulties. The core problem is that the desired manipulations are inherently 3D with many continuous-valued degrees of freedom, but the manipulator used is a 2D device with 2 continuous-valued degrees of freedom (x,y position) and 1 discrete-valued degree of freedom (button state). Jacob & Sibert [3] describe this as a mismatch between the perceptual structures of the manipulator and perceptual structure of the manipulation task. They have demonstrated that for tasks that require manipulating several integrally related quantities (e.g., a 3D position) a device that naturally generates the same number of integrally related values as the task requires (such as a Polhemus) is provably better than a 2D positioning device (such as a mouse).

The task of interactively manipulating a human figure is more complicated than the positioning task described in [3] in two ways. First, there is a much larger number of degrees of freedom to choose from, and it is often desirable to manipulate several of them simultaneously. Second, the "posture space", or set of possible postures, is continuous and Euclidean in most places, but has pockets of unreachable areas that represent postures that a human cannot normally do.

1.2 Related Work

There are numerous examples of so-called "waldo" devices (the term comes from a Robert Heinlein story [4]), which are used to control a device it mimics. The Exos Dextrous Hand Master is an exoskeleton worn over the hand and wrist, designed initially for the teleoperation of robots.[5] Jim Henson's muppets are waldo-controlled, and include a computer - generated character named Waldo C.

Chris Esposito (chrise@atc.boeing.com)
Boeing Computer Services
PO Box 24346, MS 7L-48
Seattle, WA 98124

W. Bradford Paley (brad@didi.com)
JueyChong Ong (ong@didi.com)
Digital Image Design, Inc.
170 Claremont Ave., Suite 6
New York, NY 10027

Graphic.[6] Dave Sturman has also created a "finger-walking" puppet controlled with a DataGlove.[7]

Another approach to whole-body control, more sophisticated than the mouse, uses a real person instrumented with position trackers and a motion capture system that records the person's movements in real time. Because these systems usually have sampling rates of more than 30Hz, real-time animation is possible. The Polhemus Fastrak and the Ascension Flock of Birds are two commercially available position tracking systems that have been used for this purpose. This approach neatly solves the problems mentioned above, and a well-done implementation can capture very complex and subtle real-time movements that would be difficult to capture any other way.

However, current implementations of this approach also have problems of their own. The first problem is cost, since a fully instrumented body can easily use 10-12 sensors and such a system can easily cost $30,000. The second problem, as several animators have found, is that it makes a difference where on the body the sensors are placed, and the optimal set of positions is not obvious. The third problem is that each sensor is attached to a central unit by a long cable, and that dragging around as many as a dozen cables is sufficiently encumbering that this system really requires two people to operate, with one to run the motion capture system and one dedicated to performing the motion. The fourth problem is that the electromagnetic fields used by these systems are distorted by the presence of metal in the surrounding environment, which degrades the accuracy of the reported measurements.

In other systems, the magnetic sensors are replaced with reflectors, light sources and high speed cameras which capture the reflections as points of light that can be tracked. An overview of how these systems are used can be found in [6].

To animate facial features, Williams used a video camera to track reflective material attached to an actor's face.[8] These reflective spots are then mapped to points on a facial model. Commercial systems using this technique are now available. In another technique, the VActor uses sensors in contact with the skin of an actor to achieve similar results (effectively a facial waldo). [6]

The above systems were designed for the real-time recording of animation. Their emphasis is on collecting a continuous stream of data. The higher the update rate, the more realistic the animation is likely to be.

For non-real-time input of an animation sequence, traditional animators have applied a technique of using miniature models or armatures that are positioned with varying degrees of automation for capturing successive frames of an animation sequence. This is known as stop-motion animation. For example Tim Burton's "The Nightmare Before Christmas"[9] was animated almost entirely with this technique, each of the characters painstakingly positioned by hand for each frame of the movie. For Jurassic Park[10], Industrial Light and Magic collaborated with Phil Tippett to create miniature dinosaur armatures fitted with encoders at key joints. This allowed the model to be used by stop-motion animators, but with the keyframe data input into the computer animation system.[11] It is important to note that the earliest stages of the work described here were done in early 1992, predating disclosure of the completely independent work done for Jurassic Park.

2. A New Input Device - the Monkey

The prototype Monkey stands about 18" tall and is about 6" wide. The figure can be either freestanding or attached to another sensor at the end of a scaffold on a support stand. This allows the figure to be rotated in space with respect to the stand and then left in that position without additional support. From head to toes, there are 32 total degrees of freedom (1 per sensor) in the prototype Monkey. In addition, there are three degrees of freedom for orientation with respect to the base. The production Monkey has 35 sensors, plus 3 additional for the attachment to the stand. The Monkey body itself is proportional to a 50th percentile North American male body, with deviations only where necessary to position the sensors. Additional sources of the anthropometric data used in constructing the prototypes were [12] and [13].

Since the number of degrees of freedom in the human body is greater than the number given above, obviously not all of the body joints are instrumented. For example, a single site above the pelvis in the Monkey represents the entire spine with only three degrees of freedom.. All of the major limb joints are instrumented, although the number of sensors for a joint varies depending on how that joint moves. Each sensor has an associated rotation limiter, set beyond the normal range of human motion, that prevents damage to its associated sensor. The sensors are loose enough to allow smooth motion without excessive effort, but with enough friction so that the figure can be put into a posture and stay in that posture without further support.

The rotation sensors themselves use low-noise conductive plastic potentiometers which are low cost and provide a rotational accuracy of 3 degrees and a resolution of 0.3 degrees, which is more than sufficient for almost all of the expected uses. If more accuracy is desired, then more expensive military-grade potentiometers can be incorporated without too much effort. The use of potentiometers makes this system immune to the magnetic, acoustic and infrared interference that degrades the performance of most other trackers. The maximum lag per sensor is 2 milliseconds. For a given Monkey, there is a fixed relationship between the physical position/orientation of a sensor and the value it returns, which means that postures are precisely repeatable.

The Monkey also has eight binary inputs. These can be wired to foot switches or other binary devices and may help to increase productivity or user comfort in two ways: 1) a user does not have to go back and forth between the Monkey and a mouse or keyboard; 2) users uncomfortable with computers or the software user interface do not even have to deal with this issue. Several traditional stop motion animators we talked to informally were adamant about not wanting to deal with a computer.

2.1 From Prototype to Production

Informal testing and evaluation of the prototypes at Boeing, Digital Image Design (DID), and by thousands of attendees and passers-by at SIGGRAPH led to a number of changes from the prototype to the production models. Some of the most important ones are described below. Figure 1 is a picture of the production Monkey.

We found that even though there were plenty of places that a user could comfortably and effectively grasp the Monkey to move it (e.g. the brass links in arms and legs, the potentiometers, etc.), these places were not obvious to the first-time user. We enlarged the Monkey to be 1/3 human scale, and this allowed longer links between joints. We machined larger diameter link pieces for both arms and legs (eight in all) and put hollowed out sections at the center of each one. These sections read immediately as finger-holds, and new users need no explanation to make use of them. As anecdotal evidence of their effectiveness, no further complaints about it being untouchable have been heard.

We found that the Z-axis clavicle joint was important to allow full vertical reach, as it works with the shoulder when someone does something like pat one's head. It was also important to animators, because a shrug, a very expressive

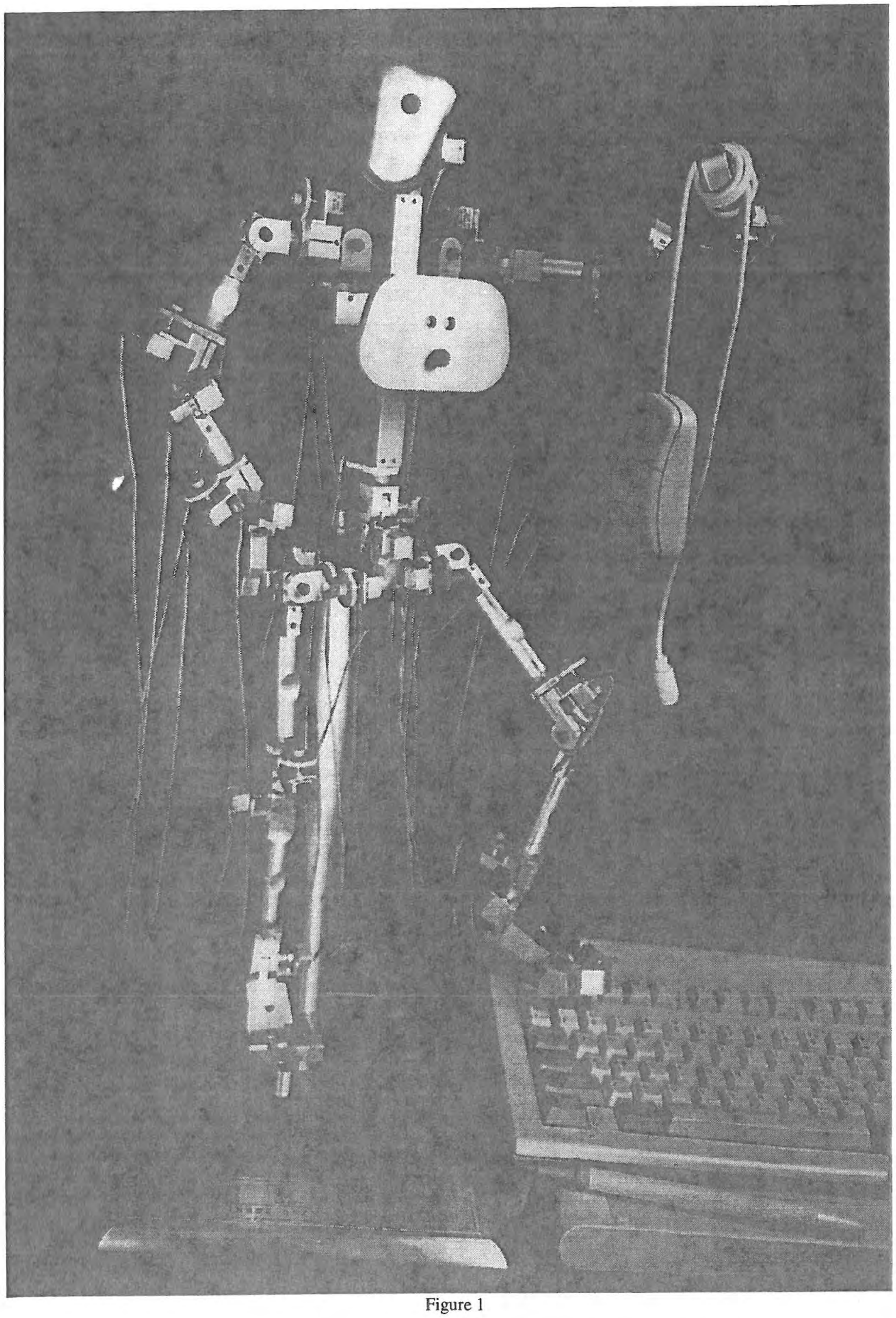


Figure 1

gesture, is almost exclusively a Z-axis clavicle rotation. This joint was added to the production Monkey.

We added a metatarsal X-axis joint to allow the Monkey to be mounted by the feet to a stand or perforated metal platform. (The metatarsal joint, and all of the joints in the leg have been made strong enough to support the weight of the whole Monkey, even standing on one toe.) This kind of mounting enables several Monkeys to be used together in a scene requiring the interaction of several figures. It also enables animators to more fully express a walk cycle, as people shift their weight from heel to toe. It also helps experienced stop-motion animators to more accurately judge and express weight and balance, things recognized as a result of a gestalt effect of combining figure and floor.

Adding a stylized, but realistic and asymmetrical, profile to the feet and hands helped in two ways. It allowed us to avoid putting mechanical limits on the swivel joints because users can immediately tell which direction to rotate a limb back towards its rest position. In the prototype, the symmetrical feet and hands left confusion, especially as to which side of the arm was the front. This slight additional anthropomorphic nuance also gives the Monkey a much more human aspect, subtly enhancing the ability of the user to look at the input device itself to sense whether the current posture is balanced, comfortable, expressive, or realistic.

The two central pieces, the torso and pelvis links, were awkward to move in the prototype. We added a breastplate that acts a handle to grasp the torso. Even more important to recreating postures involved in walking, lifting, and dancing is fluid movement of the pelvis. A permanent handle here would inhibit leg movement and interfere with the mounting post. Instead we put two holes in the front of the pelvic link as a docking site for a removable tool that provides enough leverage to move the pelvis with ease.

We did put mechanical limits on the hinge joints to prevent over-rotation that could damage the potentiometers. Mechanical stops were more important to implement on hinges because in typical use there is much more leverage available than there is for swivels.

While even the prototypes had fully variable tension on the joints, production Monkeys are more carefully balanced with regard to joint tension. The user may change the tension, but it is a laborious and non-trivial task that has a large impact on usability.

Detachable and individually replaceable wires with strain-reliefs were added to lessen the frequency and impact of the inevitable pulled cable. Now, a cable will generally just unplug if too much tension is put on it; it can easily be plugged back onto the potentiometer. Even if it breaks it can be replaced immediately, with Monkey controller and software running.

A move light was added to the Monkey controller box. This light blinks on and off with a frequency directly proportional to the speed at which a joint is moving. It gives immediate feedback to the user that this specific rotation is being interpreted by the controller. This is very useful to test the Monkey and controller without having to connect it to a computer. It is also a reassuring sign that all is functioning well even if a screen update takes several seconds (as it may if the Monkey is driving complex rendering or compute-intensive constraint systems).

2.2 The Computer-Monkey Interface

The Monkey is connected to a controller containing analog-to-digital converters and RS-232 serial communications hardware. The controller implements a simple communications protocol for sending and receiving requests and data between the computer and the controller. The controller can be set to filter out data from joints that are not of interest for a particular use. LED indicator lights on the front panel of the controller indicate power on/off and data sending/receiving.

Protocol Description

The protocol currently supports 12 commands, with each command specified as a 2-byte, unterminated string. One of the commands, `Set Active Channel Bitmask', also takes 5 bytes (40 bits) of unsigned data for specifying what channels should report data back. The most significant bit of the 1st byte is channel 0. The complete protocol description document is available on request from Digital Image Design.

The following commands are available:

- Reset the controller
- Stream mode: Switch to Stream mode and send posture data continuously
- Stream with Timestamp
- Demand mode: Switch to demand (polling) mode
- Demand: Request a single posture data record
- Demand with Timestamp
- Read the binary input channels
- Set the data channels reporting bitmask - takes 40 bits of data, 1/channel
- Set the Baud rate: selected speeds are 9600, 19200, 38400, 5760, and 115200
- Obtain the data channels reporting bitmask
- Halt stream mode (implying a switch to demand mode)
- Obtain the hardware and firmware version information

Controller Data Format

Posture data is returned in a variable length tagged record. The first byte indicates the number of data channels reported in the record. This is followed by three bytes for each channel reported: the first of the three bytes contains the data channel number (1-40), and the other two contain the value of the channel. Channel values range from 0 through 1023, although the resolution is reduced in at the extremes (approx. 0-100 and 923-1023). Finally, the last four bytes contain the timestamp if requested.

All data returned uses only seven bits per byte; the most significant bit is used as a phasing bit, where the first byte of any record has the phasing bit set, and all remaining bytes have their phasing bits cleared.

This protocol, the data format, and the phasing bit conventions are similar to those of other position/orientation tracking devices. This makes it possible to quickly convert or extend existing software for one of these devices to control the Monkey.

Initially, we were concerned about the additional bandwidth requirements imposed by the use of a tagged data format. We later decided that we could obtain sufficient throughput with the tagged format and enjoy two advantages:

1). data from each channel need not always be sent in a particular sequence, allowing greater flexibility in implementing the controller firmware.

2). it allows us to implement differential reporting modes in the future to increase throughput; in the differential stream or demand reporting modes only data channels which have changed by a certain amount set by the user will be reported. This is similar to the incremental reporting mode found in certain tracking systems. However in incremental mode, data is sent only if the sensor has changed. In differential mode, "empty" records are sent if there are no sensor updates and a demand poll or stream request is received. Recognizing the difference between a moving stream of data and "jitter" oscillations of the analog-to-digital circuitry will be an issue to be addressed.

To address the possibility of controlling several characters in a shared space, we have designed features into the controller that will enable the linking and addressing of multiple Monkeys or Monkey-complementary devices in the same way that many position/orientation sensing systems allow multiple sensors to be used in a shared space. This will be beneficial in applications which deal with multiple interacting human characters.

Controller throughput

We informally measured the throughput of the Monkey controller (firmware version 2.0) for both streaming and polling mode using a simple program which reads, decodes and stores a certain number of Monkey data records in an array. The program uses UNIX (UNIX is a trademark of AT&T Bell Laboratories) read() and write() system calls through the UNIX termio interface. gettimeofday() was called before and after execution of the read/decode/store cycle for all of the records to determine the elapsed time required to read the total number of data records into the array. The program was written in ANSI C and run on a Silicon Graphics Indigo Elan R4000 computer under a beta version of the IRIX 5.3 operating system. The serial port speed was 38,400 baud. The controller is capable of handling speeds of up to 115,200 baud.

We obtained the following results for processing 400 records:

Polling Mode (38,400 baud)

No. of channels reported	Avg. no. of reports/sec. (five trials)
40	24.94
20 (assorted*)	47.90
3 (channels 0,1,2)	49.96

Stream Mode (38,400 baud

No. of channels reported	Avg. no. of reports/sec. (five trials)
40	29.13
20 (assorted*)	53.43
3 (channels 0,1,2)	175.56

*for each trial, a different set of twenty channels was specified.

3. Color Plate

The figures on the color plate show how the Monkey might be used in computer animation using Wavefront Technologies' Kinemation software. Kinemation has a motion capture interface which allows users to write a motion capture server interfacing Kinemation with various devices (like Ascension's Flock of Birds or Polhemus' FasTrak, the Monkey and others).[14, 15]

The photographs show Bob Nicoll of Wavefront Technologies posing the Monkey in three postures for an animation sequence. After the initial capturing of the three key postures, spline interpolation and several constraints are applied. Selected frames from the resulting one hundred-frame animation are shown next.

4. Future Work

The Monkey and similar devices show promise in increasing the productivity and ease of tasks requiring the non-real-time specification of body postures. Such devices play increasingly important roles as it becomes less and less cost-effective or impossible to build a full-size mock-up or subject live actors to threatening situations.

The first author is currently engaged in writing a driver and integrating the Monkey into an existing human modeling system used at Boeing for evaluating human factors analyses of aircraft designs. Most of these analyses have to do with instrument visibility, part reachability, and validation of maintenance procedures on digital prototypes before the plane is assembled for the first time. These analyses are currently fairly time-consuming to do, but the Monkey's ease of manipulation is expected to considerably reduce the time required. A series of usability studies is planned to quantify the benefits of using the Monkey instead of the existing mouse-based system.

DID plans to produce complementary devices that will augment the capabilities of the Monkey. For instance, a hand is planned for the near future to increase the articulation capabilities that would be possible when used in conjunction with the Monkey.

DID is also continuing to investigate ways to increase the update rate of the Monkey in real applications by finding ways to reduce the rendering bottleneck in slower computers. The current Monkey server in Kinemation, for example, updates all thirty-nine joint rotations at each update regardless of whether those joints have been moved. Jittering in the analog-to-digital circuitry also causes a nervous twitching appearance of the virtual body. Some filtering of the incoming data should minimize this.

Current DID research includes developing a way to use the Monkey to add more character and expressive movement to the relatively flat-looking data that is obtained using motion capture devices. This is an important research area for animation, as motion capture alone may not satisfy audiences looking for interpretation, not mimicry. Rotoscoping was considered and discarded by Disney as early as the late 1930s for the production of Snow White. [16] Most of the experienced animators we have spoken with have remarked on the lifelessness of captured data. We intend to generate key frames to fit captured data, maintaining the weight and beauty of real physical movements, then tweak those key frames with the Monkey, bringing the movement back into the realm of fantasy.

Repositioning the Monkey to a previously input or computer-generated posture is an important task for some uses, especially in animation for adding character to relatively flat motion-capture data. This can be done on the screen by superimposing a figure in the target posture and one following the Monkey. We are of the opinion, however, that a display integral to the Monkey armature itself will greatly ease this task. Such a display is under development.

5. Acknowledgements

Jeff Taylor contributed artistry in mechanics for the Monkey armature. Genicom Consultants contributed to the early discussions about software interfaces and provided valuable anthropometric data. Rush Green and Paul Johnson from Boeing provided an animator's eye view of the Monkey specifications. Dan Ling and Tim Skelly in the Research Group of Microsoft Corporation provided encouragement and suggested more animation requirements. Bill Chernoff from Shooting Star Technologies contributed to the protocol specification. Eric Leighton gave us a careful, expert analysis of the Monkey and description of how it compares with traditional stop-motion armatures. Tony Simmerman pointed out that the Monkey allows posing to continue even if a screen update takes several seconds.

Hai Ng, Emily Hartzell and Nanther Thangarajah labored late nights to help wire the first four Monkey prototypes; Hai and Emily are currently members of the production staff. The Virtual Systems Group at Boeing Computer Services provided some of initial funding for the Monkey. Bob Nicoll and Wavefront Technologies provided much of the animation material used in the preparation of this paper's associated video. Steve Feiner gave us the references for waldos.

REFERENCES

1. Phillips, Cary. (1991) Jack 5 Users Guide, Computer Graphics Research Laboratory, University of Pennsylvania, Philadelphia.

2. Carrier, Robert. (1994) Safework 1.0 Users Guide, Genicom Consultants, Montreal, Canada.

3. Jacob, Robert., and Sibert, Linda. (1992). The Perceptual Structure of Multidimensional Input Device Selection. CHI `92 Proceedings, Monterey, California (1992).

4. Heinlein, Robert. Waldo. In Waldo & Magic, Inc., New York, Del Rey Books, 1986

5. Hand Master Controls "Smart" Machines. In NASA Tech Briefs 13, 10 (Oct 1989)

6. Robertson, Barbara. Motion Capture Meets 3D Animation. In On the Cutting Edge of Technology, Indianapolis, Indiana, Sams Publishing, 1993, 1-14.

7. Feiner, Steven. Personal electronic mail communication on December 14, 1994.

8. Williams, Lance. Performance-Driven Facial Animation. Proceedings of SIGGRAPH '90 (Dallas, Texas, August 6-August 10, 1990). In Computer Graphics 24, 4 (August 1990), 235-242.

9. Touchstone Pictures. The Nightmare Before Christmas, Tim Burton, dir. Film, 1993.

10. Universal Pictures. Jurassic Park, Steven Spielberg, dir. Film, 1993.

11. Shay, Don and Duncan, Jody. The Making of Jurassic Park, New York, Ballantine Books, 1993.

12. Diffrient, N., Tilley, A., and Bardagjy, J. (1974). Human Scale 1/2/3, MIT Press.

13. Kroemer, K., Kroemer, H., and Kroemer-Elbert, K. (1994) Ergonomics:How to Design for Ease and Efficiency, Prentice-hall, New Jersey.

14. Wavefront Technologies. Real-Time Motion Capture with Kinemation 2.1 Pre-Alpha, version 0.5. 1994.

15. Wavefront Technologies. Wavefront Motion Capture Server Library Reference Manual, version 0.1. 1994.

16. Finch, Christopher. The Art of Walt Disney: from Mickey Mouse to the Magic Kingdoms, 1983 ed., Harry N. Abrams, New York, 1983, (original edition: Walt Disney Productions, 1973).

A Virtual Space Teleconferencing System that Supports Intuitive Interaction for Creative and Cooperative Work

Mikio Yoshida Yuri A. Tijerino Shinji Abe Fumio Kishino

ATR Communication Systems Research Laboratories

Abstract

Since their advent, interaction with computers has been a very fascinating field of research. Though, we have come a long way from turning knobs and punching cards to using keyboards and pointing devices, natural language interaction has not seen widespread use as a general means of interaction. The thesis of this paper is that some application fields, specially those dealing with computer graphics, can benefit from the interaction of natural language and hand gestures. This bimodal means of interaction in computer-graphics-based task, complements the deficiencies of just applying either one of the two modes. This paper, describes current research efforts taking place at ATR for combining hand gestures and verbal descriptions for generating, modifying and manipulating 3D computer graphics objects.

1 Introduction

ATR Communication Systems Research Laboratories (ATR) has developed a Virtual Space Teleconferencing System(VSTS)[5] that makes it possible for remotely located people to meet in a computer-generated virtual workspace, giving them the impression of being at the same location, see Figure 1.

Though at first it might be tempting to say that the main contribution of the system is the fact that it allows people to "attend a meeting" without having to be there physically, we claim that the system does more than just that. That is, it provides the meeting participants with a synthetic environment that supports cooperative and creative work that allows them to perform tasks not possible or too time consuming in conventional meeting rooms or video teleconferencing.

As an illustration of how this virtual workspace can support cooperative and creative work, let's take the example of a meeting between designers of an automobile or a piece of furniture. With a VSTS such as the one we describe in this paper, the participants would be able to represent their design ideas in an intuitive manner as they evolve so that other participants can "see" them.

Generally the mental images of our ideas are vague in the beginning, but if we can transform this images into actual images(*i.e.*, in the form of virtual objects), our mental images would become clearer, not only to other people but to ourselves as well. We could then modify the actual images to either accommodate suggestions from other participants or to adjust to aesthetics or other kind of constraints such as those imposed by manufacturing or organizational requirements. This process could be repeated as many times as necessary to come up with a marketable or manufacturable product(*e.g.*, a new car model or sofa), thus rendering the meeting very productive. In order to support this kind of interactions it is indispensable that the participants be able to generate, manipulate, and modify virtual objects intuitively and with high degree of freedom. Obviously, simple manipulation, the only feature supported by most virtual environments, would not be enough because this would not give participants the ability to represent simple, not to mention complex, design ideas.

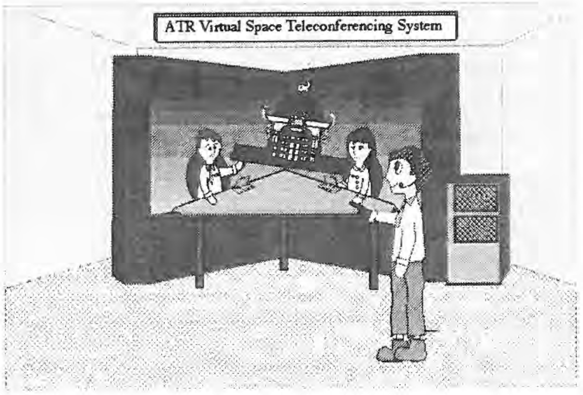


Figure 1: Illustration of the ATR Virtual Space Teleconferencing System, which supports up to three-site teleconferencing through a wide-field of view angular display.

On the other hand it would be impractical to require that participants perform operations on virtual objects in the environment through detailed graphics commands or nu-

2-2, Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-02, Japan e-mail:mikio@atr-sw.atr.co.jp, phone: +81-7749-5-1211

merical input. In this paper, we propose interaction with the objects in an intuitive way through combination of natural language and hand gestures, an area in which only little research has been done [13, 19, 6], to avoid the use of complex CAD tools currently being employed for this purpose. We also discuss the techniques used to support generation, combination and deformation of 3D virtual objects.

2 Virtual Space Teleconferencing Characteristics

The VSTS developed at ATR can support creative work because its intuitive interaface helps it to make close contact with people's natural means of expressing themselves (*i.e.*, talking, gesturing and paying attention). The VSTS also makes realistic representations of virtual participants, which is a characteristc that enhances the realism necessary to support the feeling of being there. This feeling of presence, along with the ability to perform operations on the virtual cooperative space in an intuitive manner, are some of the the most important features implied by the characteristics of the VSTS described below.

CG synthesis of the human body: While talking with a person to get some information we also pay close attention to non-verbal expressions (*e.g.*,facial expressions and other gestures). In order to take these expressions into account in our VSTS, the human image of participants are represented by means of computer graphics. Polygonal models and textures of the participants are obtained and stored in memory in advance. Only their motions are sensed and transmitted to the human 3-D CG model. The model is then updated for every motion the person does. For instance, face expressions are represented by sensing the positions of 13 control points on the face[8].

Real time collision detection: Generally it is hard to understand the precise spatial relations among objects in a virtual 3-D space. Certainly we can infer these relations from occlusions or crossing of the objects. However this is not efficient, especially when dealing with many virtual objects. This is due to the fact that we have to move the objects around and make them occlude or cross with each other at various angles and positions to grasp an idea of their spatial relations. Though, there are many methods to overcome this problem, we have adopted a real-time collision detection approach[11]. Using this method, we can detect the collisions between the triangular polygons of virtual objects in real time. The system gives the users two feedbacks, one is a collision sound and another is the color change of the collided surfaces. With this aid it is possible, to a certain degree, to infer the spatial relations of objects, even if they are occluded.

Wide-field-of-view angular display: As described above, eye and hand motion, as well as body gestures are important factors for human communication. Therefore, it is important to display participants as close as possible to real size, so that a higher degree of reality can be achieved. To accomplish this, we have developed a large wide-field-of-view display. The display combines two 70-inch back-projected screens in a seamless manner. Objects and participants in the display can be looked at from different directions and distances by a participant who wears a pair of LCD glasses fitted with a magnetic position sensor. The ability to look in 3D at objects and other participants from different perspectives in this wide-field-of-view display, gives the participants the impression of sharing a much larger environment, thus increasing the feeling of presence.

Intuitive interaction with hand gestures and speech recognition: Generally, it is difficult to operate on and with virtual objects. Positioning them precisely and deforming them, can be as difficult as the game called "pin the tail on the donkey", though in the virtual environment your eyes are opened and in the game they are closed. 3-D modeling tools (*e.g.*, AutoCad^{TM1}, Alias^{TM2}, etc.) make it possible to perform these operations precisely, but at the expense of increasing the complexity. Operators cannot operate intuitively such tools unless they translate their ideas to expressions close to the machine representations (*e.g.*, the menus supplied by the tool and/or the programming language).

This kind of tools can be very distracting because they require that the participants adopt a low-level computer representation that might not fit very well with their own mental representations. We have focused on supporting natural language (through speech recognition) and hand gestures to give the users an intuitive way to interact with and modify objects in the virtual world.

Intuitive interaction with natural language and hand gestures may free us from explicitly having to focus our attention on what the computer is doing. Instead, we can focus on what is it that we want to accomplish. On the other hand, it might be difficult for a computer program to recognize and combine natural language with hand gestures, because their relation might depend on the cultural background. In the system we developed, only a few combinations between natural language and hand gestures were explored. However, these few combinations add a great deal of intuitivity to the system as we were able to discover.

3 Techniques to Realize Creative and Cooperative Work

3.1 WYSIWYS(What You Say Is What You See) Frame work

When we read or listen about the description of an object, we tend to create an image of it in our minds. If such an object is already-known, our previous memory about it is recalled, if not we try to associate it with and already-known object. In a similar way, when describing our own mental image of an object to someone, we make use of verbal expressions as well as hand gestures. Furthermore, we assume the listener shares the same common knowledge as we do, and otherwise we try to adequate or descriptions to his knowledge about such an object or others related to it. It is necessary that the listener has this common knowledge (*i.e.*, ontology) in order to get his own mental image of the object being discussed. However, a simple missunderstanding or a lack of some piece of knowledge on the part of the listener will result in a completely different object from the one being described.

This kind of communication skill would be very practical if we could use it not only between people but also between people and computers. This suggests the advantage of using

[1] AutoCad is a trademark of Autodesk Inc.

[2] Alias is a trademark of Alias Research Inc.

natural language in combination with hand gestures in the virtual space, to express our intentions or mental images as 3-D virtual objects to computers, can be very advantageous if an underlying intermediate representation of concepts is available in the form of a 3-D ontology of objects and operations performed on them. Since the virtual objects are synthesized by means of CG to give us the visual feedback of our interactions, so we can "see" what we want to communicate to others. Such a concept has been named "WYSIWYS"(What You Say Is What You See)[18]. WYSIWYS promises to be an invaluable paradigm for numerous applications in such fields as design, art and computer imagery; and specially for virtual space teleconferencing and CSCW systems, by enhancing human-to-human communication.

3.2 Verbal Expressions and Hand Gestures

3.2.1 Variety of Verbal Expressions and an Ontology for 3-D shapes

Many verbal expressions have similar meaning and tend to be vague. Consider the case of 3-D shapes, a simple pyramid, which is a 3-D primitive, could be referred as "Pyramid", "Squared Cone", "Roof" and so on. The way it is addressed depends on the person who does it. 3-D primitives such as cubes, cylinders, spheres, cones, prisms, pyramids, etc., usually have several ways of being referred to. Based on some 3-D geometrical considerations these expressions can be classified to make them correspond to one of these 3-D primitives. It has been proven that complex objects are described as combinations of these basic primitives[7]. Consider Figure 2(b), even though we recognize it as a "car", many other expressions are also valid (*e.g.*, "automobile", "motor car", etc.), each of which depend on the spatial relations between the components(primitives).

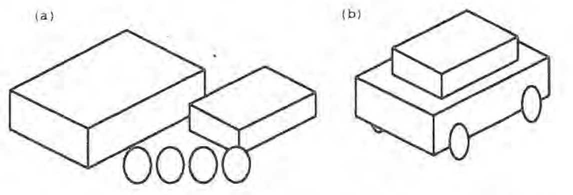


Figure 2: Basic shapes can describe more complex ones:(a)presents some basic shapes in random order without any particular meaning,(b)organizes the shapes into the simple form of a car.

In this way, the general 3D-shape conception (idea) of an object is not only related to its basic form or shape but also to the way this complex object was constructed. Furthermore, the structure of such a complex object might be stratified and/or recursive. Even though this structure conception of an object may differ from people to people, there are some good reasons to believe that there is a common sense knowledge about it that we all share [20, 16, 4, 9, 12]. Therefore, in order to permit the vagueness of natural language the system should take into account this common sense knowledge about this general structure conception of 3D-objects. Such a structure conception of 3D-objects is what we call "Knowledge-level 3-D Visual Ontology". And is one part of the system configuration, indeed a database[18]. This "Knowledge-level 3-D Visual Ontology" includes also some information (concepts) about other deformations, *e.g.*, rigid transformations, scaling, bending, twisting, tapering, rounding, swelling, sharpening, etc. Also, 3D-shapes related concepts about state and characteristic are provided here. These characteristics may be geometrical(*e.g.*, round), visual (*e.g.*, color), functional (*e.g.*, sitting down), quantitative (*e.g.*, size) or qualitative (*e.g.*, large). Similarly, states may be quantitative as well as directional (*e.g.*, 50cm. to the left), or qualitative as well as positional (*e.g.*, to the left of A). For more details about this , the reader is referred to [17].

3.2.2 Hand Gestures and Shape Modeling

There are two main problems when considering verbal expressions and hand gestures as means of a human-computer interface. One is that of getting accurate data about the hand gesture, and the other is that of transforming the virtual(graphic) objects according to the intention of the gesture. Recently, the former has reached a good level of reliability, due to increasing research efforts to quantify hand gestures by means of several devices. On the other hand, the latter has remained more passive due to some difficulties; description of graphical objects and performance issues among others.

For instance, it is somehow easy to make the computer understand that an specific gesture means to change the length of a graphical object. However, it is generally difficult to go one step further and actually change the length of the given graphical object. In the case where the geometric representation of an object is given by vertices and polygons, the polygons associated with the transformation (*i.e.*, elongate) must be determined and actually moved to achieve the required effect. This kind of determination and transformation of vertices and polygons required by polygon-based graphic representations can be very expensive and therefore cause a decrease in system performance. Therefore, we propose a better geometrical representation, that might be more appropriate to associate 3-D object deformations.

Superquadrics, which were first discovered by Hein[2], provide a useful representation for a combination of hand gestures and natural language to model 3-D objects.

Superquadrics are defined by the following vector:

$$\begin{aligned} x &= a_1 \cos^{\varepsilon_1}\alpha \cos^{\varepsilon_2}\omega \\ y &= a_2 \cos^{\varepsilon_1}\alpha \sin^{\varepsilon_2}\omega \\ z &= a_3 \sin^{\varepsilon_1}\alpha \end{aligned} \qquad (1)$$

where x, y, and z are the coordinates of points on the superquadrics surface, a_1, a_2, and a_3 are scale parameters. The parameters $\alpha \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ and $\omega \in [-\pi, \pi]$, represent the degrees of latitude and longitude, while ε_1 and ε_2 represent the squareness on the y-z and x-y plane respectively.

With this representation, if a verbal command as "elongate that object up to here" is combined with your hand placed perpendicularly to one of the axes of the superquadrics surface, only the corresponding scale parameter has to be changed to achieve the desired result.

All these parameters, as well as other local parameters for deformable superquadrics proposed by Terzopoulos[14], can be combined with different sorts of concepts dealing with hand gestures to intuitively model an object's shape.

Superquadrics are only one kind of implicit functions that can be used for representing objects generically. There might be other representations useful for this purpose as well. In our VSTS we have to yet implement superquadrics primitives as the general way of representing objects, but preliminary prototypes show that this representation may be very efficient.

4 Configuring a Japanese Portable Shrine

4.1 3-D Visual Ontology about Japanese Portable Shrine

Following the guidelines presented above, we developed a prototype of the system that can be used in the context of designing a Japanese portable shrine, which is common in Japanese festivals.

Intentions inherent in combinations of hand gestures with verbal expressions are interpreted in the following manner:

Input: *result of speech recognition*: this gives a rated list of possible candidate strings of characters.

1. translate the string with the highest rating into a concept in the prestored ontology

2. repeat 1 until a concept that makes sense in the current context is found

3. get information about gesture and position of the user (*if needed*)

4. translate to graphic commands and process it

Output: *visual feedback*

String to concept translation: Translation from a string to a concept is accomplished by referring to our "Knowledge-level 3-D Visual Ontology". This system has 2 types of Knowledge-level Ontologies; one about labels, which is shown in Table 1, and another about operations, also shown in Table 2. The later can also be classified in three subcategories: **Generation, Manipulation and Deformation.** In short, the knowledge-level ontology consists of concepts about names of 3-D primitives and names of operations (*see* Table1,2). As the reader might notice, concepts about operations and objects can be associated with several labels, thus dealing this way with several words that might have the same meaning according to the context.

Table 1: The Words about The Parts of A Japanese Shrine

Regular Word	Alias	English
Hoo	Ootori	Chinese phoenix
Tachidai	Hoo-Base	Base of Chinese phoenix
Yane-Base	Kashiranuki	Base of Roof
Yane	Okujoo	Roof
Kago	Heya	Chamber or Room
Torii	Igaki	Fence and Gate
Kamisori	Base	Base of Shrine
Katsugi-Boo	Boo	Carrying Bar
Komafuda	Fuda	Plate

Combining concepts with hand gestures and position of user: Some concepts, such as "Are(*that*)", "Ano ...(*that* ...)"and "Asoko(*there*)" require information from hand gestures. When the translation process infers that one of these words was spoken, it tries to confirm that an appropriate gesture (*e.g., pointing*)was also performed. In this case, the system requests information about the hand

Table 2: The Words about The Operations

Regular Word	Alias	English
Oku	Tsuika-Suru Haichi-Suru Naraberu Hairetsu-Suru	to put
Kuttsukeru	Tsunageru Kumiawaseru Tsunageru	to join
Nobasu	Nagaku-Suru	to lengthen
Marumeru	Maruku-Suru	to round
Idoosuru	Ugokasu	to move
Mawasu	Kaiten-Suru	to rotate

position and orientation. If any objects are in the context of the gesture an appropriate action will take place on them (*e.g.,* it will be selected if pointed at). Other, concepts such as "Mottekuru (*bring it here*)" need only to obtain the position of the hand to translate the object to that particular place. The reason that information about the position of the user is also needed is that the system allows also view dependent directional manipulations such as "Migi he (*to the right*)" or "Mae he (*to the front*)". In those cases, the direction in which the objects are moved, rotated, etc, depend on the position from which the person is viewing the scene at.

Translating to Commands In this final stage, the system maps and implements commands that reflect the interpretation of the verbal utterances along with hand gestures. Generally commands are composed of two arguments; object and operation. Some commands also require numerical data that is inferred either from hand and/or viewpoint information or given by default (*e.g.,* object coordinates, angular or scaling factors, and so on).

4.2 System Architecture

The components of this system are depicted in Figure 3. This subsection will describe the function of each module.

Gesture Recognition Module: This module manages the information about hand gestures. Specifically, this module observes hand gestures and position, and transforms this information to data. This module doesn't perform symbolic interpretation or recognition.

Speech Recognition Module: This module processes speech input and produces rated character string candidates of the recognition of a phrase. This module was developed by ATR Interpreting Telephone Laboratories[10], a sister company of our labs.

Multimodal Input Fusion Module: This module, synchronize the results of the speech recognition and the gesture recognition modules, and integrates their inputs into single interpretations.

Shape Ontology: This is a database about basic 3-D shapes. It contains both information about the shapes and operations that one can perform on each individual shape. In this module each shape can have several different labels associated with it to allow some degree of ambiguity.

Command Library: This contains the commands about operations for 3-D shapes such as generating, deforming, moving and so on. All operations that the user intended will be converted to these commands referring to the Shape Ontology described above.

Generation and Execution or Graphics Commands: This module generates the commands to operate 3-D shapes referring the data obtained from the three modules described above. The output of this module is sent to the Virtual World Managing Program.

Virtual World Managing Program: This program receives the output from the module above and the data about the position and the field of view of the virtual cooperative workers. Moreover, this module displays all changes of the virtual environment to the users.

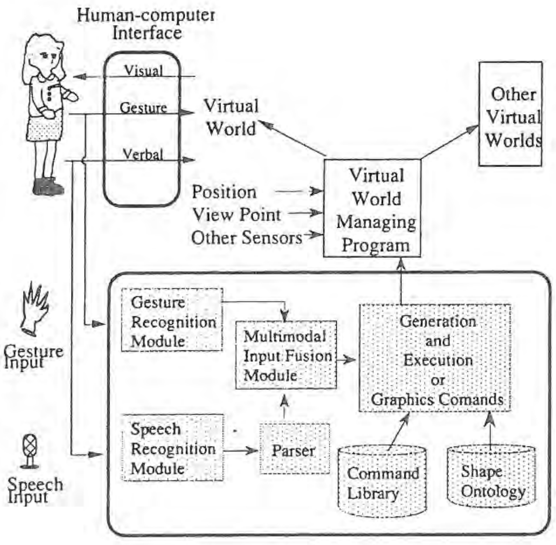


Figure 3: System Architecture

Though there are the Collision Detection module and the module for displaying human body, in this paper omit these detailed explanation, because of space limitation. Instead the reader is referred to [11] and [8] respectively and to the discussion.

4.3 An Example

Here we will describe a prototype of the VSTS that allows use of verbal expressions and hand gestures to configure a Japanese portable shrine.

Generation: The following example places the chamber of the shrine as shown in Figure 4, that has been previously stored as a polygon-data file, in the virtual workspace through verbal interaction only.

Speaker: Hoo-Base wo haichisuru
(*where Hoo-Base = base of Chinese phoenix and haichisuru = to place*)

The words "Hoo-Base" is translated to the concept of "Tachidai"(= Base of Chinese phoenix) referring to the "Knowledge-level 3D Visual Ontology".

Selection: The following example shows selection of the roof of the shrine.

Speaker: Kago wo
(*where Kago = chamber*)

The next example shows selection of the far most object.

Speaker: [*pointing gesture*] + Are wo

Speaker: [*pointing gesture*] + Ano Kago wo

When the user performs a pointing gesture, a line is drawn from the hand in the direction of the pointing finger. Every object in the path of this "laser" is highlighted with a white bounding box. This white box gives appropriate visual feedback so that the user knows what objects he/she is pointing at. Moreover, when an object is selected using verbal expressions or combination of verbal expressions with hand gestures, the object is highlighted with a green bounding box as visual feedback; see Figure 5. This allows the operator to know whether the recognition was successful or not. We have found this to be extremely helpful.

Position Translation: This example shows how an object that has been previously selected is moved to the right.

Speaker: [*pointing gesture*] + Are wo
+ Migi ni + Idoosuru
(*where Migi ni = to the right*
Idoosuru = to translate)

This example shows how an object is moved to a position indicated by the hand.

Speaker: [*pointing gesture*] + Are wo
[*Place a hand at a position*] + Mottekuru
(*where mottekuru = to bring*)

Figure 6 shows that an object has moved to the position indicated by the hand. This interaction allows us to get an unseen object. Moreover, when an occluded object exists and if the user knows its name, he can get it by selecting it with its name.

Joining together: The next example shows how two objects are selected and joined together(Figure 7, 8).

Speaker: [*pointing gesture*] + Are wo
[*pointing gesture*] + Kono Katsugiboo ni
Tsunageru
(*where Katsugiboo = carrying bar of*
shrine and Tsunageru = to join together)

In this initial prototype, one of the attributes of the objects is the information of where and what objects can be joined.

Scaling: The next example shows how an object is made taller(Figure 9).

Speaker: [*pointing gesture*] + Ano Yane wo
Takaku-Suru
(*where Yane = roof*
and Takakusuru = to make higher)

5 Discussion

The VSTS incorporates a high degree of reality with an intuitive interface to support creative and cooperative work space. Using our system, the meeting participants can communicate non-verbal information among themselves (*i.e.*, gaze, facial expressions, body posture, hand gestures and so on). At the same time they can combine hand gestures with verbal description to perform operations on virtual objects. We implemented these functions in order to realize high degree of realism.

In addition, the WYSIWYS environment that we implemented in the VSTS, allows this operations to be very intuitive.

5.1 Pursuit of Reality

In our system, computer graphics representations of conference participants can change their facial expression, can move their eyes and move their bodies that reflect the real movements of the participants. For instance, for facial expressions we used a technique that is based on synthesizing changes of facial expressions using smaller data by transmitting information about positions of some points on face[8]. Using such technique, we have realized an efficient way of transmitting facial expressions, a kind of non-verbal information.

Real-time collision detection is another subject which we considered important as feedback to participants to identify spatial relations between virtual objects. We realized real-time collision detection by implementing an octree-based algorithm[11] for such purpose. We believe that in the future, this technique will become the basis to implement Force-Feedback in our system.

On the other hand, there are some issues that remain to be solved. Our system, forces participants to wear special equipment that decrease the feeling of realism we want to achieve, namely LCD shutter glasses and sensor-fitted gloves. A system that does not require this type of equipment can be more believable and less obstructive to realism. At ATR we are investigating on techniques that address these issues. A stereoscopic viewing technique that does not require the use of special glasses is an example. It relies on lenticular screens that transmit the appropriate images to the right and left eye through vertical optic components[15]. Formerly, this method had the known problem that a person could see stereoscopically only in some restricted regions, but was solved by combining head tracking with mechanical movement of the scene. Moreover, a real-time hand gesture recognition using 3-D prediction model[3], which doesn't need any equipment (*e.g.*, *Cyber*Glove$^{\mathrm{TM}}$[3]), is also the subject of on going research at ATR. Force-feedback mentioned above could be one of the most intuitive interfaces to help interpret relations between virtual objects. However, force-feedback research currently relies on mechanical devices to simulate reaction forces on the hand or skin. At present, it is too difficult to suggest that we can completely get rid of all special equipment in the near future.

5.2 Intuitive Interface

The most natural way for people to communicate is through the use of gestures and natural language. In the WYSIWYS framework that we have implemented in the VSTS, people can not only use natural language and gestures among themselves, but they can also use the same means of communication to express their intentions to the computer. Therefore, the VSTS provides an environment that departs from conventional means of human-to-human communication and human-machine interaction. The WYSIWYS framework allows us to see our design intentions literally take shape as virtual objects or transformations. Because, other people can also see our design intentions as they evolve into virtual objects, communication between people becomes more efficient as well. Furthermore, other people are also able to make modifications to computer interpretations of our design intentions, because they have access to the same objects being shared in a common virtual space. This is a very important characteristic of the VSTS, because it means that besides supporting the creative process of shaping our design intentions, it can also support the collaborative process of reshaping the resulting virtual objects to accommodate other people's intentions as well.

This all may sound very nice, but there are some problems that we need to overcome. The main problem is that the computer needs to know much about common sense 3-D shapes. It needs to be able to map all these shapes as well as operations performed on them to ambiguous combinations of natural language phrases and gestures that may change from person to person. In other words, the VSTS needs a common sense 3-D shape ontology of concepts related not only to 3-D shapes, but also operations on the shapes. The question now arises of how to construct such an ontology. Perhaps we can not answer this question satisfactoryly at this time, but we can speculate that some knowledge-acquisition methods or some collaborative efforts to acquire common sense knowledge can also be applied here. The authors, have already investigated the possibility of applying at least one knowledge acquisition method, namely Personal Construct Psychology, to acquire visual descriptive concepts about cars [17]. This proved to be successful within some given constraints, but still needs further exploration to be applied in a more broad common sense domain. Because, the VSTS relies on highly sophisticated computer programs to support a high degree of intuitive interaction, speed becomes also major problem.

Finally, we can also mention the problem of recognizing gestures accurately and interpreting their meaning according to context. At present, we have only implemented pointing and grabbing gestures because our system is still on a prototype stage. However, we have found that even these two simple gestures can be interpreted in several ways if we rely only on the information given by the shape of the hand. For instance, a pointing gesture may be confused with a gesture for pushing an object. Therefore, hand gestures can not be treated as symbolic information and have to be combined with context information given by other means such as natural language. Other gestures, also depend on motion and there are current research efforts taking place at ATR to incorporate them in the VSTS as well[1].

6 Conclusion

Careful study of how hand gestures combine with verbal utterances when people communicate might lead to a new way to communicate our intentions to systems such as the VSTS developed at ATR. As computer technology improves, faster computational speed and graphic display update rates will also become available. Hence, in the future VSTS-like systems might become also very important tools for communi-

[3] *Cyber*Glove$^{\mathrm{TM}}$is a trademark of Virtual Technologies.

cation among people for creative, cooperative and constructive processes. In the simplest case, an user might point at some unfamiliar object in a virtual environment and ask more information about it (*e.g.*, a virtual museum). In a more difficult situation, the user could generate complex 3-D models by simply shaping objects that simulate complex behavior with the hands and giving qualitative features with verbal instructions.

Integration of hand gestures with verbal instructions also makes a break-through on how to avoid too much dependence on natural language processing by taking advantage of real-time computer graphics feedback. This way, people can "see" that the computer is "interpreting" from natural language. This instantaneous feedback allows people to guide the interpretation process interactively. For instance, it is difficult to describe a situation about some real physical event to a computer that depends exclusively on current natural language processing technology. However, if the computer provides a graphical environment that can be manipulated with verbal instructions and hand gestures, a person could describe a physical situation by generating graphic objects with one or both hands and detailing their attributes verbally.

It is also important to simulate physical phenomena associated with some objects in the virtual environment so that the creative process produces a better feeling of how the object might behave in the real world. However, we also leave this as a future research issue.

References

1. Altman, Edward J., Normal Form Analysis of Chua's Circuit with Applications for Trajectory Recognition. In *IEEE Transactions on Circuit and Systems*, volume 40, pp. 675–682. IEEE, October 1993.

2. Gardiner, M. The superellipse: a curve between the ellipse and the rectangle. In *Scientific American*, volume 213, pp. 222–234: 1965.

3. Ishibuchi, Koichi., Takemura, Haruo., and Kishino, Fumio. Real-time hand gesture recognition using a 3-D prediction model. In *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, pp. 324–328, 1993.

4. Johansson, G. Configurations in Event Perception. Almqvist and Wiksell, Stockholm, 1950.

5. Kishino, Fumio., Ohya, Jun., Takemura, Haruo., and Terashima, Nobuyoshi. Virtual space teleconferencing system–Real-time detection and reproduction of 3-D human images. In *Proceedings of HCI International '93*, pp. 669–674, 1991.

6. Mochizuki, Kenji., Takemura, Haruo., and Kishino, Fumio. Object manipulation and layout in a 3-D virtual space using a combination of natural language and hand pointing. In *Proceedings of 4th Sapporo International Computer Graphics Symposium, HW-1*, pp. 38–42, 1992.

7. Nishihara, H. Intensity, visible-surface and volumetric representations. In *Artificial Intelligence*, volume 28, pp. 293–331. 1981.

8. Ohya, Jun., Kitamura, Yasuichi., Takemura, Haruo., Kishino, Fumio., and Terashima, Nobuyoshi. Real-time reproduction of 3D human images in Virtual Space Teleconferencing. In *Proceedings of IEEE Virtual Reality Annual International Symposium*, pp. 408–414, September 1993.

9. Rosch, E. On the internal structure of perceptual and semantic categories. In Moore, T.E., editor, *Congnitive Development and the Acquizition of Language*. Academic Press, New York, 1973.

10. Sagayama, S., Takami, J., Nagai, A., Singer, H., Yamaguchi, K., Ohkura, K., Kita, K., and Kurematsu, A. ATREUS: a Speech Recognition Front-end for a Speech Translation System. In *Proceedings of EURO SP 1993*, pp. 1287–1290, 1993.

11. Smith, Andrew., Kitamura, Yoshifumi., Takemura, Haruo., and Kishino, Fumio. A Simple and Efficient Method for Accurate Collision Among Deformable Polyhedral Objects in Arbitrary Motion. In *Virtual Reality Annual International Symposium, North Carolina, USA*. IEEE, March 1995.

12. Stevens, S. *Patterns in Nature*. Brown Books, Boston, MA., 1974.

13. Takahashi, Tomoichi., Hakata, A., Kobayashi, Yukio., and Yamashita, K. User interface using language and visual information. In *Proceedings of Computer World'88, Kobe, Japan*, pp. 192–199, 1988.

14. Terzopoulos, Demetri. and Metaxas, Dimitri. Dynamic 3D Models with local and global deformations: deformable superquadrics. In *IEEE Transaction on Pattern Analysis and Machine Intelligence*, volume 13, pp. 703–714. 1991.

15. Tetsutani, Nobuji., Nagashima, Yoshio., Tomono, Akira., and Kishino, Fumio. Stereoscopic display method employing eye-position tracking. In *Proceedings of International Symposium on Three Dimensional Image Technology and Arts(Tokyo, Japan, February 5-7, 1992)*, pp. 101–107, 1992.

16. Thompson, D. A. *On Growth and Form*. U.K., 1942. 2nd ed.

17. Tijerino, Yuri A., Mochizuki, Kenji., and Kishino, Fumio. Interactive 3-D Computer Graphics Driven through Verbal Instructions: Previous and Current Activities at ATR. In *Comput. & Graphics.*, volume 18, pp. 621–631. Elsevier Science, Great Britain, 1994.

18. Tijerino, Yuri A. "WYSIWYS"–Interactive generation, manipulation and modification of 3-D shapes based on verbal descriptions. In *AI Review Journal*, volume 8. Kluwer Academic. unpublished.

19. Weimer, David. and Ganapathy, S.K. A Synthetic visual environment with Hand gesturing and Voice input. In *Proceedings of CHI'89*, pp. 235–240, 1989.

20. Wertheimer, M. Laws of organization in perceptual forms. In W. D., Ellis, editor, *Gestalt Psychology*. Harcourt Brace, New York, 1923.

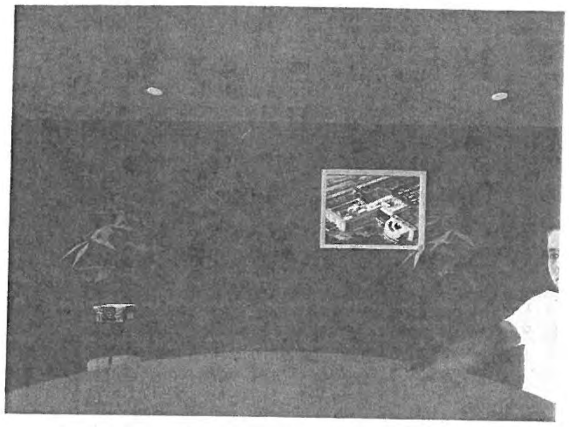


Figure 4: Example of Generation: generating a Tachidai.

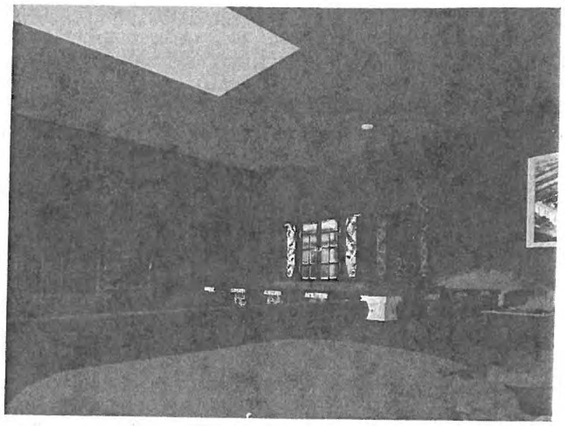


Figure 7: Example of Joining together(1)

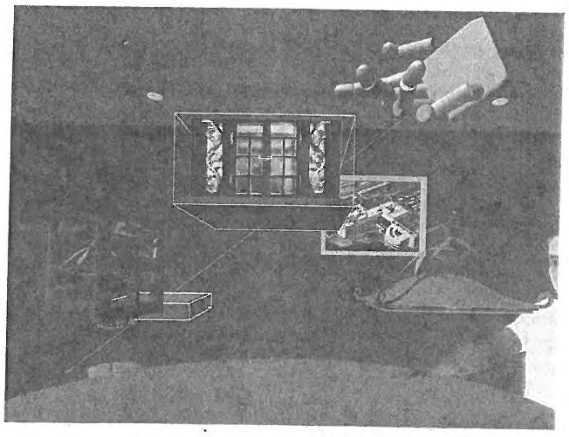


Figure 5: Example of Selection: selecting a chamber.

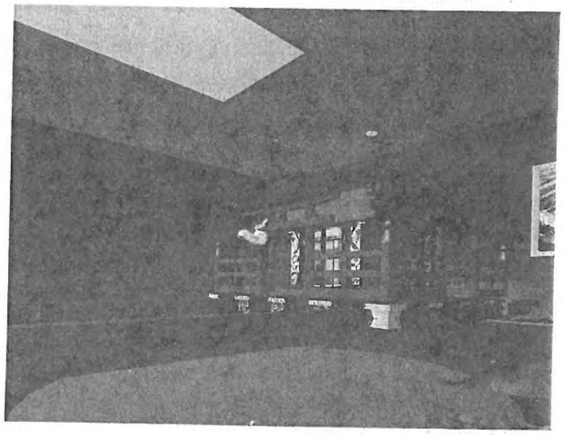


Figure 8: Example of Joining together(2)

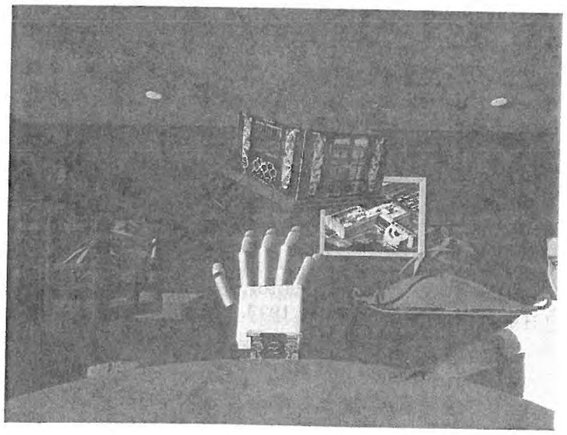


Figure 6: Example of Translation: bring a base of Chinese phoenix to hand.

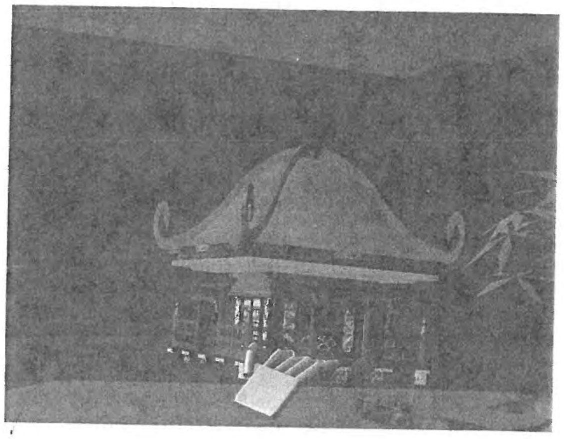


Figure 9: Example of Scaling: roof is made taller.

HAPTIC RENDERING: PROGRAMMING TOUCH INTERACTION WITH VIRTUAL OBJECTS

K. Salisbury[*], D. Brock[†], T. Massie[‡], N. Swarup[‡], C. Zilles[‡]
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139

1 Abstract

Haptic rendering is the process of computing and generating forces in response to user interactions with virtual objects. Recent efforts by our team at MIT's AI laboratory have resulted in the development of haptic interface devices and algorithms for generating the forces of interaction with virtual objects. This paper focuses on the software techniques needed to generate sensations of contact interaction and material properties. In particular, the techniques we describe are appropriate for use with the Phantom haptic interface, a force generating display device developed in our laboratory. We also briefly describe a technique for representing and rendering the feel of arbitrary polyhedral shapes and address issues related to rendering the feel of non-homogeneous materials. A number of demonstrations of simple haptic tasks which combine our rendering techniques are also described.

2 Introduction

The process of mechanically interacting with with remote and virtual objects has been of interest to researchers for a long time. Handling of distantly located objects through remotely controlled manipulators has been feasible since at least the early days of handling hazardous nuclear materials [12]. In these systems a master control device is used to control the actions of the remote manipulator. "Force reflection" is sometimes used to present to the user, through the master, forces encountered by the remote manipulator. This permits perception and manipulation of these remotely located objects. In the early 70s researchers began to simulate this type of interaction through the use of simple mechanical models of objects in the environment. By computing the forces which would be encountered in interactions with real objects and displaying them through a force reflecting interface, the sensation of touching objects could be created [10]. These "haptic" interactions with simulated objects represent one of the first instances of mechanical interaction with virtual objects.

Haptic interactions have been used to aid investigations of molecular docking [1]. This task requires the user to follow a force gradient until the molecules are interlocked. The force field the molecules move through is derived from models of inter-molecular forces. Although a realistic calculation of these forces is computationally intensive, they can be applied to the user as simple attractions or repulsions and used to find suitable docking configurations. This approach has been found to be useful for this complex, molecular level, task.

The molecular docking task does not, however, require generation of the same type of contact forces that we encounter in everyday manipulation of the objects. Forces resulting from contact, palpation, and stroking actions require generation of macroscopic forces which give rise to sensations of shape, surface hardness, texture and friction. Kilpatrick [5] found it suitable to model hard surface interactions using Hooke's law augmented with clicks when virtual contact is made. He recommended, in addition, a mechanical brake making surfaces "feel" harder, to "radically increase friction when a virtual hard surface is encountered."

Recent interest in creating and interacting with virtual environments (VEs) has begun to push these ideas to new levels of sophistication. Taking advantage of advances in graphic display, computational capability and modeling of visual representation has permitted the visual component of complex virtual environments to be rendered

[*]Principal Research Scientist, Dept. of Mechanical Engineering
[†]Research Scientist, Artificial Intelligence Laboratory
[‡]Graduate Student, Dept. of Mechanical Engineering

with good fidelity. The ability to perform mechanical or "haptic" interaction with these scenes has lagged significantly behind. The majority of VE systems in use today rely on passive devices, such as instrumented gloves and joysticks, to track user motions and permit limited interaction with virtual objects. To provide force feedback to users in these systems, a few researchers have adapted teleoperator masters and in some instances have developed dedicated haptic feedback devices. Advances in haptic interaction have been limited due to lack of high performance interface devices, and the lack of a coherent approach to object modeling and sensory display of mechanical attributes.

Our work at at MIT has begun to address this problem with integrated investigations into the science of haptics. The term haptics has come to be used by the VE and telerobotics communities to refer to the sensorimotor interactions which occur during perception and manipulation of mechanical objects. We have concentrated on methods for tracking the motion of the human finger and applying precisely controlled forces to the user's fingertip through a ground-based haptic interface, the Phantom [6]. A wide range of demonstrations have shown that our device has sufficiently clean dynamics (stiff, low-friction, backdrivable) to display a wide dynamic range of impedances with high fidelity [2]. As a result of the high sampling rate, sensor resolution, and structural stiffness of our haptic interface, the dynamic modes of the haptic interface are highly decoupled from the programmed dynamics of the virtual environment. Thus, transparency to the dynamics of the interface hardware is achieved and representation of the virtual environment dynamics is greatly facilitated. As a simplifying assumption, we have focused on point interactions. Point contacts with objects permit only pure forces (no torques) to be exerted through the contact and require only three active (powered) motions in the haptic interface to faithfully reproduce the force geometry. The minimal complexity of the system has helped achieve good bandwidth by reducing parasitic structural and actuator mass. This reflects our view that good temporal display quality is at least as important as good spatial characteristics in a haptic display. The point paradigm is not a particularly restrictive assumption in that multiple points can be combined to exert torques on objects and control their orientations as with human fingertips.

While researchers have begun to look at algorithms for generating forces resulting from contact with virtual objects [3,11], we feel that there is a great need for a more coherent approach to generating (or rendering) these sensations and modeling interactions with complex objects. Our interest is in developing a framework in which we may represent shape, surface properties, bulk properties and multiple object interactions. Such a framework should permit the representation of a wide variety of objects and object interactions, while simultaneously addressing the problems of real-time generation of appropriate sensations. We can view haptic interactions as really occuring at two levels. When a contact occurs there is a net force (vector) experienced (or generated) by the user. In addition the distribution of the forces (or tractions) which occur at each contact site are perceived through the user's mechanoreceptors, giving rise to our tactile sense. Because of the difficulty in building tactile displays which present the spatial distribution of forces at each contact, we have focused on displays which present only the net force information. We have found that if this force information is presented with sufficient bandwidth and resolution, many effects that we consider to be tactile sensations can be created. Surface shape, compliance, texture and friction can all successfully be evoked through proper modulation of the net force exerted on the human. A general framework for haptic rendering must then be able to represent and permit display of these and other basic haptic elements. These elements, in turn, must be contained in a larger framework which represents the object shape and bulk properties appropriate for rendering the larger scope of interactions that occurs during object motion and inter-object interactions.

Though our efforts to build a general haptic rendering system are still in the early stages, we have made progress in the rendering of basic contact interaction elements, macroscopic object shape properties, and bulk object properties. We describe below our progress in these areas.

3 Rendering Haptic Elements

A region in space populated with objects can be divided into volumes which represent free space and volumes which represent objects. The surfaces of these objects comprise the boundaries between the two. Perception of the shape and details of these objects is accomplished by haptic exploration in which these surfaces are palpated and stroked. We discuss below the various haptic elements which must be available to enable active exploration and perception of objects, many of which we have implemented. Taken together, these elements permit higher level tasks such as grasping and manipulation, some of which we have demonstrated.

3.1 Freespace Movement

A haptic rendering system must first be able to give the sensation of free space. To do this requires a haptic interface with intrinsic characteristics that allow it to be effortlessly moved about, with little distraction from mechanism friction, inertia and vibration. In using the Phantom interface this is enabled by the device's intrinsically low backdrive friction and inertia. In addition, the mechanism's smooth transmission characteristics and well damped high natural frequency reduce unintended vibrations to nearly below perceptible levels.

3.2 Contact Transients

At the instant of contact with a surface rapid onset of force occurs with sufficient impulse to remove momentum from the user's finger or tool. This requires good bandwidth and stiffness in the interface to provide quick stable, onset of force. We and others typically accomplished this by programming a one-sided spring function to generate repelling forces that increase with surface penetration. As discussed below, careful control of this contact impedance can be used to vary and enhance perceived material properties.

3.3 Contact Persistence

The sensation of sustained contact with a surface requires that the user be able to push into it and experience compressive contact forces of sufficient magnitude to make it feel solid without actuator saturation or instability. We have found that it is not necessary to generate huge forces to create the illusion of solid immovable walls. In fact, when performing manipulation involving motion at only the elbow, wrist, and fingers, users rarely exert more than 10 Newtons of force. The illusion of solid surfaces, is reinforced by the contrast between these contact forces and the low free-space forces imposed by the Phantom (typically less than 0.1 Newtons).

3.4 Contact Impedance

While not completely separable, we can divide the impedance of an object into two components, the local or contact impedance and the net or gross impedance of the object. The contact impedance gives rise to sensations of material properties. As other researchers have recognized [3], we have found that adding viscous damping to the characteristic equation for a constraint surface greatly enhances the user's perception of a hard surface. Perceptually, a wall simulated in our system by $f = Kx + Bv$ can be made to feel like hard plastic, whereas a wall simulated by $f = Kx$, using the same value for K, would feel spongier than a typical mouse-pad. Effectively, adding a damping term will change the coefficient of restitution between a user and the virtual surface [11].

3.5 Frictionless Surfaces

When a user only experiences forces normal to the surface being touched the sensation of a slippery or frictionless surface is evoked. Computing contact forces in this case requires only the determination surface normals and penetration depth. This is, in fact, is the easiest haptic effect to generate with our system. The same good intrinsic properties of the Phantom system which permit the sensation of free space motion contribute to the faithful rendering of frictionless motion in the 2 dimensional subspace of sliding across a surface. While using the Phantom to touch friction-free surfaces, users have described the sensation as that of "an ice cube sliding on glass."

3.6 Surface Friction

Imposing tangential forces on users while they stroke a surface adds an important sense of realness to perception of objects. In real life, we rarely experience frictionless surfaces and, in fact, heavily rely on friction in tasks involving manipulation. We have developed several techniques which approximate both stiction and Coulomb friction (static and dynamic friction). As with [11], we recognize the importance of incorporating static friction into the friction model. In our implementation the model has two states: sticking and sliding. When contact is first made we store the location of contact and begin the stiction state. If the user tries to slide along the surface, tangential forces (using Hooke's law or impedance control) are applied to restore the user back to his initial point of contact, the "stiction point". If the force required exceeds the normal force times the the coefficient of friction, then we change to the sliding state.

Unlike [11] we model the sliding state with Coulomb friction rather than simple viscosity. Coulomb friction involves applying a retarding force which is only a function of the coefficient of friction and normal force, in the direction opposite to the direction of motion. Due to the difficulty in accurately measuring small velocities in a sampled data system, we designed a robust method which requires only position measurements. When transition to the sliding state occurs, we know the displacement from the stiction point and can assume the user is moving in the direction of this displacement. To create a tangential force with the correct magnitude and direction we simply need to move the stiction point to a new place on the line which connects the user and the old stiction point. The stiction point's offset from the haptic interface point can be calculated by dividing the friction force (the normal force times the coefficient of dynamic friction) by the stiffness. Once the new stiction point is assigned we return to the stiction state.

By setting the coefficient of dynamic friction below the coefficient of static friction we have demonstrated a convincing stick-slip sensation. The vibration generated during object motion against friction modeled in this way evokes a sensation of slippage. In the BLOCKS demonstration program (program images shown at end of paper) a user is able to pick up a virtual cube with two Phantoms; if the objects slips, the user can detect this occurrence by attending to these vibration and net force direction cues. Without such a friction model, force closure grasps of virtual objects would not be possible. By using friction to enable grasps the BLOCKS program permits the blocks to be stacked, thrown, dribbled, and juggled [15].

3.7 Surface Curvature

Surface discontinuities at edges and corners are primarily perceived in humans by mechanoreceptors sensitive to curvature. However we have demonstrated that these basic curvature sensations can be convincingly be displayed by control of the normal force vector. Users will perceive a discontinuity of the normal direction as an edge or corner; one key to making smooth objects is to vary the direction of the force vector continuously. By utilizing surface normals at the vertices (defined say, by averaging ajacent facet normals), a satisfying normal force direction can be found at any point via interpolation between these vertex normals (much like Phong shading in graphics).

We have found the actual shape of an object to be rather insignificant in making objects feel smooth. Because of the inaccurate position sense that humans have, a coarsely meshed polyhedron will be perceived as smooth if a suitable surface normal interpolation scheme is used. This has been demonstrated in a pair of example programs we have written. One program, models a surface by assigning heights on a 2-D mesh. Complex surfaces including a telephone and a baboon's face have been "rendered" by interpolating height and surface normal between points in this matrix of heights. In the case of the phone rendering actual heights were measured and entered into the mesh. In the case of the baboon face, a pseudo-height map was derived from image point brightness. Though this does not really represent the true shape, it provides sensations of underlying geometry. The second program, (SMOOTH) presents the user with a smoothed rendering of a polyhedrially modeled asteroid shape. It is rendered using the constraint-based god object method described below, with the addition smooth interpolation of surface normals across edges. The result is that the previously sharp edges feel rounded.

3.8 Surface Texture

The sensation of texture results from both the effects of small shape details and friction on surfaces. In direct manipulation humans can utilize both their tactile sense (fingertip mechanoreceptors) and net force sense to perceive texture. Conveniently (since we currently lack good tactile array force displays), variations in net force applied to a user can generate texture sensations. Minsky [8] presented users with variations in tangential forces dependent on local shape variations to evoke a wide variety of texture sensations. We have also used shape-driven variations in normal force to evoke sensations of texture on a frictionless surface [6]. To be complete, variations in normal and tangential forces should be used together to simulate texture with force-based displays. The stick-slip sensation demonstrated by [15] does address part of this need in providing a purely friction dependent sense of texture. It remains to combine both shape and friction dependent force variations to display more complex texture.

We have begun to explore techniques similar to graphics texture mapping that can be used to overlay the surfaces of objects with standard textures. For example, one could define a texture map which induces slight reorientations in the rendered contact normal of a surface facet to which it is applied. Making this perturbation a function of location on the facet reflects the spatial dependence of texture, however care must be taken to not alter the spatial frequency when the texture is mapped to facets of different scale.

We have also created convincing shape dependent textures by using a height-map function applied to planar surfaces. At every point on the planar surface, the software calculates a height offset and a normal vector offset, as defined by the height-map. The texture patch is defined by a grid of heights, and is constructed to permit tiling on a bigger surface without texture discontinuity between adjacent patches. For example a suitable continuous texture patch can be defined by assigning $z = \cos(x)\cos(y)$ with x and y in the range $(0, 2\pi)$. It is interesting to note that depending on the period and amplitude of such a texture, users may perceive it as a shape, as a texture, or in the limit, as friction.

3.9 Net Object Motion

In the preceding sections, we have primarily addressed local effects in which little or no net object motion occurs. In fact, some of our efforts have investigated interaction with objects that are free to move in one or more dimensions. By tracking contact forces according to the above techniques and applying these forces to a model of the object's mass, stiffness and viscosity with respect to ground, it is relatively easy (in few-degree-of-freedom systems) to integrate the resulting accelerations and compute net object displacements. Demonstrations of spring centered switches (SLIDERS) and switches with detents (BUTTONS) have been made and suggest a rich range of virtual controls which may be constructed. A demonstration which permits pushing of masses on a frictionless surface (MULTY3) shows the ability to interact with dynamic objects and control two-degree-of-freedom motions. Two phantoms have been used together in a program (BLOCKS) which permits grasping and placing cubes which are free to move in rectilinear (three-degree-of-freedom) motion. Extending these capabilities to full six-degree-of-freedom motion including manipulation and assembly tasks is clearly a formidable undertaking but one which requires a firm understanding of the local effects we have addressed to date. Significant extensions are required to address the kinematics of articulated objects such as mechanisms and objects with transient kinematics, such as are encountered during assembly and tool interaction.

4 Shape Representation

It is desirable to not only display local surface properties, but also overall shape of objects. We have implemented a number of techniques to describe shape. An evolution of techniques is in progress, starting with vector field implementations, progressing to god object representations, and looking ahead to potential energy function representations.

Our vector field methods subdivide the volume of an object and associate a sub-volume with each surface. When the haptic interface is in a sub-volume, a force whose magnitude is a function of the distance penetrated is applied in the direction of the normal to the associated surface[6]. These vector field methods conceptually create a map of the 3-D object volume and assign a force vector to each location, so that during each servo loop the contact force can be looked up.

This method works rather well for simple geometric shapes because it is reasonably easy to construct these subspaces by hand. For planes aligned with the coordinate axes the force vector can be computed from a simple $F_x = Kx$ relation. For spheres, the direction is that of the vector pointing from the sphere's center to the haptic interfaces endpoint, and the magnitude is the distance the endpoint has penetrated the sphere's surface scaled by a constant. The simplicity of this method has allowed us to explore many aspects of haptic rendering, but it has its drawbacks. When designing more complex objects it is less obvious how to sub-divide the volume, and thin objects are susceptible to being pushed through.

The central difficulty is that the maximum stiffness of any virtual object is limited, due to the inherent mechanical compliance of haptic interface devices. This means that the user's contact point often penetrates simulated object volumes to a greater distance than would be possible in real life, leading to an ambiguity in determining which surface was entered. A better method was needed to keep track of the surface being stroked if believable forces were to be displayed robustly.

The constraint-based god object method employs a strategy to stop the haptic interface's virtual contact point from penetrating objects[14]. By concentrating on surfaces rather than volumes, we attempt to more realistically compute forces, and incidentally give ourselves access to an enormous body of objects already in existence in standard surface representations. This method keeps track of a virtual contact point (the god object) which remains on the surface when a virtual object is probed. With the location of the god object on the surface, there is no ambiguity in which force vector should be applied to the user.

Given the previous location of the god object and the current location of the haptic interface, the algorithm will identify a number of surfaces on the rendered object which are currently involved in the interaction and denote them as active. A surface is *active* if the god object is on one side of the rendered surface, and the haptic interface is on the other, and the action takes place within the boundaries of the surface. One surface can be active for each powered degree of freedom in the device.

Once this set of surfaces, or constraints, has been identified the new location of the god object can be computed. By finding the closest point on the active constraint surface to the current haptic interface point we can determine the new location of the god object (strictly, this applies to the frictionless case, but can be extended to include surfaces with friction. Since we chose planar constraints, the solution can be found by solving a set of linear equations requiring only 65 multiply or divide operations to calculate the coordinates.

This method will create a faceted object which can exhibit sharp corners; smoothed objects can also be rendered by adding a smoothing algorithm. We are currently in the process of combining the basic effects described above witht the god object renderer. We expect this to result in a fairly rich system in which arbitrarily shaped polyhedral objects may be rendered with controllable degrees of smoothing, friction, surface impedance. In the next section we address another approach to rendering complex shapes which lends itself to rendering objects with bulk material properties which are significantly non-homogeneous.

5 Rendering Non-homogeneous Materials

Although the methods described above permit a large class of objects to be rendered, they do not directly address objects composed of non-homogeneous materials. Incorporation and presentation of non-homogeneity greatly extends the class of objects that can be presented, particularly tissue surrounding the internal organs and skeletal structure of vertebrates. For instance, haptic presentation of biological objects will be an integral component in multiple modality surgical environment simulations. We describe below preliminary work which concentrates on local surface impedance properties [13].

We are concurrently developing approaches to the haptic scanning of surface property data based on force sensing, analogous to the visual scanning of pictures to produce image data. We envision mechanically probing an object at discrete surface points, capturing local surface properties through force and position measurements, and finally storing the data in a format readable by the haptic renderer. Due to the inherent sampling nature of scanning, the haptic rendering of the sampled surface data must be able to sufficiently reconstruct the original surface properties without perceptual loss of information. Hence, the techniques we use to haptically *represent* surface information are intrinsically coupled to the issues involved in haptically *scanning* surface properties.

In contrast to computer graphics which involves global environment rendering, haptics primarily involves local interactions. For a large class of objects, local interactions are decoupled from global object dynamics. Consequently, efficient computational haptic rendering algorithms should take advantage of this local nature. As the user moves their interaction point on the surface, the haptic renderer will only render the local "window" of surface representation data about that point. We have successfully demonstrated haptic rendering of non-homogeneous objects by employing this haptic window technique.

5.1 Rendering Methods

The geometric modeling technique of B-spline surfaces is utilized to interpolate discrete, spatially distributed, values of surface impedance data. B-splines are particularly appropriate to haptic rendering because they are comprised of a set of blending functions that has only local influence and are dependent on a finite number of neighboring control points[9]. Furthermore, the order of the interpolating polynomial is not affected by the number of control points. Both of these facets complement the attributes of the haptic window which only renders local properties. To ensure smooth haptic transitions across non-homogeneous sample points, C^2 continuity is imposed on the B-spline which results in cubic interpolating surfaces. As a result, a 4 x 4 patch of data points is necessary to construct the interpolation polynomial.

Although geometric interpolation of surface impedances provides an efficient and simple means for rendering surface properties, there are limitations. Primarily, geometric interpolation does not guarantee that a closed circuit interaction path with the virtual object will be conservative, hence potentially providing the sensation of an unrealistic "active" surface. It is possible to interact with the surface in a compliant area expending little work, move tangentially to an area of higher impedance, and then leave the surface with nonzero net energy transfer. In order to ensure passivity of the surface, requirements must be placed on the internal force field and boundary conditions imposed by the surface. Specifically, if the force field $\mathbf{F}$ within the surface can be described as the negative gradient of a scalar potential field Φ,

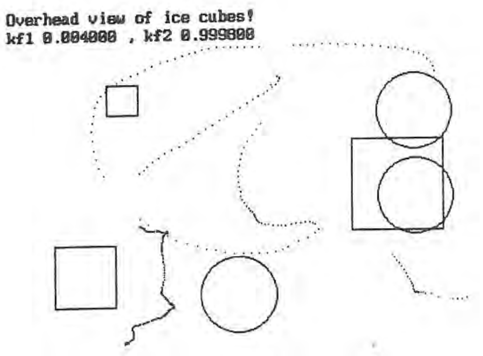
$$\mathbf{F} = -\nabla\Phi \tag{1}$$

and the potential at the surface is constrained to be constant everywhere, then *any* closed path interaction with the object will be conservative.

We are investigating potential field methods which respect the passivity requirements directly. Conceptually, we can use static electro-magnetic fields to model object properties. Representing surfaces as perfect conductors permits us to enforce equal potential at entry and exit from touching an object. Solutions of Laplace and Poisson equations, can then be used to solve for the value of forces at points internal to the object. If we then wish to set the local impedance at points within the object, we may impose further internal boundary conditions on the potential field. Thus, we may conveniently map impedances measured for real objects into the geometric model of the object's force generating function using the above relationship.

6 Haptic Demonstrations

A number of demonstrations (illustrated in screen images shown below) have been developed which use the basic haptic elements, described in the previous section, as building blocks for more complex applications. Initially simple geometric shapes, such as spheres, cubes, and polyhedra, were constructed and implemented using the vector force field approach described earlier. Dynamic objects, illustrated in Figure 1, were later developed. These simulations allowed users to push and slide objects, permitting the discrimination of virtual mass and inertia. Surface effects between objects, including stiction and Coulomb friction were also added.



```
Overhead view of ice cubes!
kf1 0.004000 , kf2 0.999000
```

Figure 1: MULTY3, a dynamic simulation which allow users to push and slide virtual objects, and permit the discrimination of mass, inertia, friction, and impact.

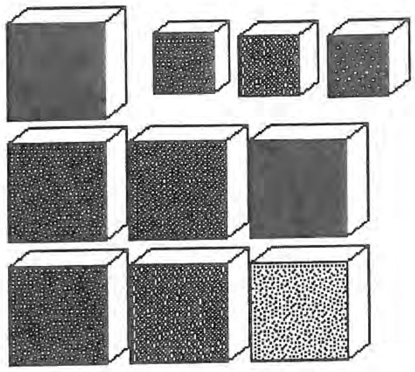


Figure 2: Virtual instrument panels include knobs, buttons, sliders, and switches, with clicks, detents, toggles, and stiffness. Illustrated is BUTTONS program.

An application which became immediately apparent, was the virtual control panel. Knobs, buttons, sliders, and switches, featuring clicks, detents, toggles, and stiffnesses, allowed users to "feel" and operate virtual instruments, as shown in Figure 2. Another application, which may have significant importance, is the simulation and rehearsal of medical procedures. Figure 3, shows the screen image of the needle biopsy simulator we developed. A magnetic resonance image (MRI), acquired from the Brigham and Women's Hospital, was segmented along a user specified line. Mechanical properties including stiffness, tear strength, and viscous friction were assigned to each layer, so that the surgeon could feel the pressure of needle against the tissue and "pop" as each layer was pierced.

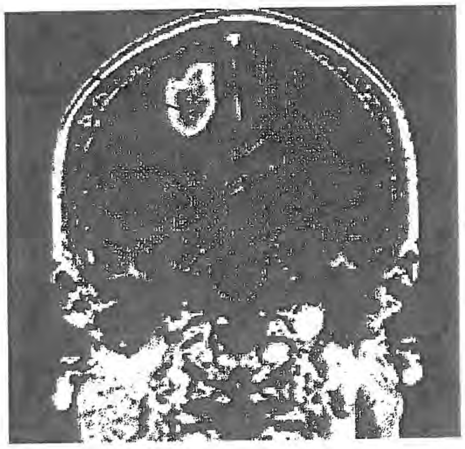


Figure 3: A needle biopsy simulator demonstration, MRI,allowed surgeons to experience the sensation of pressure of the needle against tissue, and the "pop" as each layer is piered.

Using a distributed interactive simulation approach, we created a tissue palpation demonstration. The haptic device, controlled by a 486PC, transmitted probe and tissue information via the network to a SGI Indigo^2 Extreme. Thus the user could feel the compliant surface while viewing a high-quality graphics image, as shown in Figure 4.

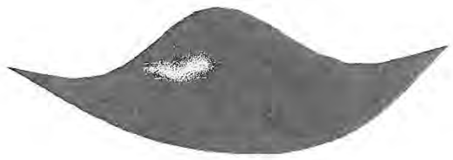


Figure 4: DEFORM, a tissue palpation demonstration using the Phantom haptic device running on a PC and a Silicon Graphics workstation to provide graphics support.

Using standard graphics file formats, we were able to haptically render arbitrary convex and concave objects. Figure 5 shows an "asteroid" imported from a .plg file and presented to the user to both push and probe. This example is prepatory to the development of standard object interchange format which will allow visual, haptic, acoustic, and functional representation. Finally, combining the objects, elements, and algorithms developed above, we build a simple "virtual world" composed of building blocks and virtual fingertips to manipulate them. The user employed two haptic devices to pick up and toss the cubes around the room, while feeling the friction, mass, inertia, and impact of these objects.

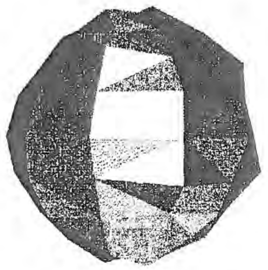


Figure 5: Standard graphics file formats were imported and rendered with the **ASTEROID** program. The program permits palpation and exploration of interior or exterior surfaces.

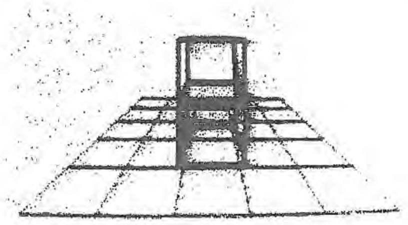


Figure 6: **BLOCKS**, a program which renders two blocks that can be grasped and manipulated using two fingertips.

7 Acknowledgments

This work is supported in part by NAWCTSD contracts, N61339-93-C-0108, N61339-93-C-0083, N61339-94-C-0087, ONR Grant N00014-93-1-1399, fellowships from the National Science Foundation Graduate Fellowship program and the National Defense Science and Engineering Fellowship program.

8 References

1. Brooks, Jr., Fred P., M. Ouh-Young, J. J. Batter, and P. J. Kilpatrick. "Project GROPE – Haptic Displays for Scientific Visualization." Proceedings of SIGGRAPH '90. Dallas, Texas. August 6-10, 1993.

2. Colgate, J. Edward, and J. Michael Brown "Factors Affecting the Z-Width of a Haptic Interface." Proceedings of the IEEE International Conference on Robotics and Automation, 3205-3210. 1994.

3. Colgate, J. Edward, P.E. Grafing, M.C. Stanley, and G. Schenkel. "Implementation of Stiff Virtual Walls in Force-Reflecting Interfaces." Proc. IEEE-VRAIS, pp. 202-208, 1993.

4. Durlach, Nathaniel I., et al. "Virtual Reality: Scientific and Technological Challenges." Report produced for the National Research Council, National Academy of Sciences. Washington D.C. December 1994.

5. Kilpatrick, P. J. "The Use of Kinestetic Supplement in an Interactive System." Ph.D. dissertation, Computer Science Department, University of North Carolina at Chapel Hill. 1976.

6. Massie, Thomas H. "Design of a Three Degree of Freedom Force-Reflecting Haptic Interface." SB thesis, MIT EECS Department. May, 1993.

7. Massie, Thomas H. and Kenneth Salisbury. "The PHANToM Haptic Interface: A Device for Probing Virtual Objects." Proceedings of the ASME Winter Annual Meeting, Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems. Chicago, IL. November 1994.

8. Minsky, Margaret M., et al. "Feeling and Seeing: Issues in Force Display." Computer Graphics, vol. 24, no. 2, pp. 235-243, 1990.

9. Mortenson, Michael E. Geometric Modelling. New York: John Wiley & Sons, Inc. 1985.

10. Noll, A. Michael. "Man-Machine Tactile Communication." Society for Information Display Journal. July/August 1972. Reprinted in Creative Computing, July/August 1978 p.52-57.

11. Salcudean, S. E. and T. D. Vlaar. "On the Emulation of Stiff Walls and Static Friction with a Magnetically Levitated Input/Output Device." Proceedings of the ASME Dynamic Systems and Control Division. Chicago, IL. Nov. 6-11, 1994.

12. Sheridan, Thomas B. Telerobotics, Automation, and Supervisory Control. Cambridge, MA: MIT Press, 1992.

13. Swarup, Nitish. SM thesis in progress. Department of Mechanical Engineering, MIT. Expected May 95.

14. Zilles, Craig and Kenneth Salisbury. "A Constraint-Based God Object Method for Haptic Display." Submitted to IEEE/RSJ International Conference on Intelligent Robots and Systems, Human Robot Interaction, and Cooperative Robots. 1995.

15. Zilles, Craig. SM thesis in progress. Department of Mechanical Engineering, MIT. Expected May 95.

Object Associations
A Simple and Practical Approach to Virtual 3D Manipulation

Richard W. Bukowski   Carlo H. Séquin

University of California at Berkeley [‡]

Abstract

This paper describes a software framework to aid in designing and implementing convenient manipulation behaviors for objects in a 3D virtual environment. A combination of almost realistic-looking pseudo-physical behavior and idealized goal-oriented properties, called object associations, is used to disambiguate the mapping of the 2D cursor motion on the display screen into an appropriate object motion in the 3D virtual world and to determine a valid and desirable final location for the objects to be placed. Objects selected for relocation actively look for nearby objects to associate and align themselves with; an automated implicit grouping mechanism also falls out from this process. Concept, structure, and our implementation of such an object association framework in the context of the Berkeley Soda Hall WALKTHRU environment are presented.

1 Introduction

Creating a fully equipped model of a large, furnished building for virtual walkthroughs is an arduous task. Even assuming the availability of a good interactive 3D geometry editor with a friendly and efficient user interface, such tasks are inherently much more difficult than drafting and editing in only two dimensions. The problem with a 3D world is that it is impossible to exactly control all six degrees of freedom (DOF) at once with only 2-dimensional input and display devices. Typically, software solutions are used to map 2D cursor motion to limited 3D object space motion [12]. These can be cumbersome to use in complex environments, and do not address the fact that objects often require positioning with respect to objects around them. High-tech solutions such as the "SpaceBall" [2], "DataGlove," 3D mice [15], or virtual 3D displays do not solve the problem either; precise placement of objects in three dimensions is hard – even in the real world – unless we get help from the physical interactions of the objects we want to place. Consider positioning a picture frame one millimeter in front of a wall without touching the wall with the frame or with your hands; visual feedback alone cannot do a satisfactory job.

As part of the Berkeley WALKTHRU Project we have built a prototype version of an object manipulation system, called "WALKEDIT," tailored to populating large building models with furniture, personal items, books, coffee cups, and various trimmings and details that make such a building model look real and interesting (see Figure C1). Our approach is based on a system of "object associations," a software framework that supports simple and practical manipulation of 3D objects with 2D I/O devices via two special types of programmer-supplied procedures and an implicit grouping behavior. It gives the programmer the ability to specify object-dependent methods of disambiguating 2D gestures in a 3D world and allows association of suitable local behavior with database objects to make precise default placement easy. These associations usually fall somewhere between physical simulations and mathematical constraints, but can be less formal and more flexible than either.

2 Interactive Building Environments

In the process of developing an editor for our Soda Hall WALKTHRU program, we examined different methods for helping the user to move objects in 3D with 2D devices. We wanted the process of moving furniture in a 3D virtual environment to be as quick and easy as moving cut-out cardboard pieces on a floorplan. However, it should also be possible to force objects to align themselves nicely to walls and to one another, if the operator chooses such an option.

There is no single correct answer to the question of what "ideally" should happen in response to a user dragging an object across the 2D display screen. There is at least one uncontrolled degree of freedom (DOF) due to the third coordinate of the virtual world. Choosing the "right" value to be assigned to this coordinate becomes a contention between *realistic* (physically correct) and *teleological* (goal-oriented) models for the virtual world, and is strongly dependent on the specific application domain. Traditional tools tend to take an extreme stand on one or the other end of the spectrum.

Most 2D drafting tools provide an idealized goal-oriented behavior. Selected shapes freely follow the cursor "across" other objects and snap nicely into alignment with other features if *grids* or *gravity* have been turned on. Setting up these extra controls requires some overhead for activating alignment manifolds, setting up tugboats and orientation frames, changing editing modes, or grouping and un-grouping objects. In 3D virtual worlds, the situation is even worse; ob-

[‡]Computer Science Division, Soda Hall, Berkeley, CA 94720-1776; bukowski@cs.berkeley.edu and sequin@cs.berkeley.edu.

jects now have twice the number of DOFs to be controlled. However, the real world being modeled can often provide disambiguating clues or implicitly desirable alignments. By exploiting these application-specific expectations, some of the teleological control overhead can be automated. In a simulation of a *physical* environment, it seems natural to exploit gravity and solidity to disambiguate the projection of the 2D input parameters into the 3D virtual world. By providing automated alignment with the surfaces on which objects come to rest, extraneous DOFs are removed, and the cognitive burden of specifying them is removed from the user. A complete, accurate, physical simulation, on the other hand, may be counter-productive to efficient 3D editing; we all know how hard it is to move real furniture through a small apartment.

Since we are working in a *virtual* world, we can adopt selectively some of the desirable characteristics, while ignoring others, and add some useful non-physical behavior on top. Such "magic" behavior can be *more* convenient than the behavior of real-world objects [13]. It is easier to move furniture without concern for temporary physical obstruction or inter-penetration; the notorious task of moving a piano through a staircase is no problem in our WALKEDIT environment, and pictures can hang on walls without physical hooks. On the other hand, extra physical or non-physical constraints can be imposed where they simplify manipulation tasks: e.g., pictures can be forced to hang perfectly level at all times, or chairs in a classroom can be made to snap into nicely aligned rows. Object associations provide the structure and the encouragement for the interface programmer to set up this balance between realism and virtual-world magic.

Some of the key paradigms of 3D manipulation and some of the behavioral aspects of objects in a building that we found desirable when populating our Soda Hall model with furniture are summarized below, together with a reference to the object associations that provide the corresponding behavior:

- User-selected objects should follow the mouse pointer, so that "point and place" becomes an integral, intuitive operation. The *relocation procedure* (to be discussed below) provides the main mechanism for this behavior.
- Objects typically should not float in mid-air but rest on some supporting surface. If the cursor points to the surface of a desk or to a bookshelf, it can be implied that the user wants to move the selected object to that particular surface; an *association procedure*, "pseudo-gravity," supports that goal.
- Alternatively, many things, such as picture frames or light fixtures are attached to walls or other vertical surfaces; another *association procedure*, "on-wall," generates the desired behavior.
- Such implicit associations of objects with reference objects should be maintained even when the reference object moves or is changed in other ways; however, they must also be breakable so that objects can be lifted off a surface easily and moved somewhere else. An automatic *dynamic grouping mechanism* built into the object association framework provides that service.

We have also found visibility information to be an important tool. In our WALKEDIT environment, it is natural for the user to move in such a way that the destination point of the motion is visible and the object's final position can be defined precisely by direct pointing. Therefore, we find it acceptable to restrict object manipulations to locations that one can see, avoiding the complexity of user interfaces with which one can reach behind other objects (e.g. systems based on DataGloves or other 3D devices). This simplifies considerably the task of mapping 2D pointing to 3D motion.

3 Object Association Framework

3.1 Background

Our approach borrows heavily from several paradigms developed in the realm of interactive computer graphics over the last decades. It first has notions of ***snap-dragging*** [3], but without the need of explicitly dealing with visible alignment manifolds; most alignments are provided automatically by the association procedures rather than explicitly by the user. Second, while it can emulate some of the behavior of a *physical simulation* of the objects in the environment [1, 7], it can be less constraining than our every-day world; objects can pass through one another and remain in physically impossible non-equilibrium positions under the control of suitable associations, which may be application-specific or may depend on the editing mode. Third, while some associations could readily be described as *constraints*, our system does not require the rigid formality and associated solution machinery that one would find in a mechanism editor based on an underlying constraint system [11, 4, 10, 8, 6].

A novel feature that emerges naturally from our approach is an automated *implicit grouping mechanism*; it uses the relationships established between objects as they reposition themselves with respect to their environment.

3.2 Two-Phase Approach

The generic editing move in an interactive environment is to "grab" an object and then to "place" it (and any objects grouped with it) somewhere else. In our "WALKEDIT" program, a user clicking on an object selects both the object itself and a *selection point* on the object which makes a natural *handle* for further manipulation. Once the object and its selection point have been established, the user can apply either a *local* motion by dragging the mouse pointer, or a *remote* operation such as "picking up" the object and "placing" it at a different location. Control of these motions and final placement is handled by the procedures of the object association.

When dragging the object to some desirable final position, the intermediate path of the object should be as direct as possible to follow the goal-oriented directive of the user, yet the final position should be "realistic" within the defined simulation constraints of the virtual environment. This extracts the best of the two competing approaches discussed in Section 2; the user can move the object anywhere in a true teleological way, but then the object realigns itself to satisfy some of the physical realities of the environment. This leads to a two-phase approach to moving an object. During a first *relocation phase*, the object follows a trajectory free of physical or behavioral restrictions and which is a suitable disambiguation of the 2D path specification in screen space into a 3D motion in world space. During a second *association phase* the object uses its association rules to determine a good nearby position which best satisfies the stated behavioral conditions of the object in a rest state.

Object associations are thus based on two types of small procedures that are invoked when an object is selected. Each object is assigned one *relocation procedure* but may have a number of prioritized *association procedures.* The relocation procedure is used during local, interactive motion to disambiguate gestures made with the mouse pointer; it defines a mapping of incremental 2D mouse motion to incremental 3D object motion. Association procedures are used for both local and remote placement; they apply additional motion components to an object, based on the other objects in the area, with the goal to preserve the desired object behavior. In addition, objects will dynamically link themselves to the reference objects with respect to which they have aligned themselves; they will typically follow any movements of these reference objects.

3.3 Associations used in WALKEDIT

In WALKEDIT we are primarily concerned with keeping objects supported against gravity, having them attached – and thus properly aligned – to the ceiling, to walls, or to vertical surfaces of other objects, or having objects aligned with respect to each other. All this can be achieved with a remarkably small set of primitives. Different objects carry by design one or more (ordered) association attributes. The user can add or remove extra association attributes from the existing set to selected objects during the interactive walkthrough mode. When such an object is selected, its attributes will determine which relocation procedure applies to the object and which association procedures are used to determine the final placement of the object.

So far we have implemented two relocation procedures: the *on-horizontal* procedure is designed for objects that move primarily horizontally, while the *on-vertical* procedure is for moving along vertical surfaces. In both routines, the object moves along a piecewise continuous, polyhedral 2D manifold in space. The left mouse button *translates* the object along the manifold without changing its orientation relative to the manifold. The middle button *rotates* the object about a line through the center of its bounding box normal to the manifold section on which the object rests.

In addition, there are three association procedures: the *pseudo-gravity* procedure, the *anti-gravity* procedure, and the *on-wall* procedure. Pseudo-gravity simulates objects that normally rest on a supporting surface. Anti-gravity is used for attaching light fixtures, smoke detectors, sprinklers, and other such objects to a ceiling. On-wall is used for pictures, white boards, wall clocks, and other objects that hang on vertical surfaces. All of our WALKEDIT association searches use the same type of ray-based probing mechanism to find alignment objects. These ray-probes determine which nearby objects affect the alignment of the selected object.

We cannot expect that these few simple procedures will take care of *all* editing needs in our building environment. The goal is to make 90% of the typically encountered operations easy and natural. For special needs we still can access traditional editor functions via pull-down menus. If one needs an exact rotation by 45 degrees, one opens the *rotation operator* menu; if one wants to create a perfect row of 20 chairs, the familiar *replicate* menu is perfectly appropriate. If, on the other hand, one finds that one often has to do a special task that is not well supported by classical editor menu commands, such as pushing furniture into corners, then it pays to write a new association procedure "in-corner." This procedure probes in all 4 directions, finds the *two* closest objects, and then does on-wall alignments in two directions, trying to satisfy them both at the same time.

If this is not good enough, because one frequently wants to crowd furniture together in less regular formations, then it is time to develop a more or less accurate pseudo-physical collision detection mechanism and add it to the collection of association procedures. Depending on the types of objects that need to be manipulated, this may simply be based on bounding boxes (good enough for file cabinets) or may use a more sophisticated algorithm that can handle concave objects (needed for grand pianos). We are currently experimenting with a prototype implementation of such a collision detection routine based on the Canny-Lin algorithm that quickly finds closest features in pairs of convex shapes [9, 1].

We have integrated these procedures with the user interface layer that controls all the major editing functions: selection, dynamic grouping, dragging, and detailed placement. In the following sections we review these tasks in detail and discuss our implementation of the procedures that constitute the object association framework.

4 Selection and Dynamic Gathering

In WALKEDIT, selection is performed by shift-clicking the object. There may be other objects that have been previously associated with the selected object; these other objects were positioned with respect to the selected object when they were last moved. For example, the reference object identified by the pseudo-gravity association is the surface on which the selected object came to rest. Since the position of the reference object influenced the position of the selected object, it makes sense to implicitly group the latter with the former and maintain that relative positioning when the reference object is moved. This means that all of these associated objects must be found and grouped with every new object selected; this grouping is maintained for the duration of the motion. An object can have multiple associations; it will then move when any of its reference objects moves.

Associations are *not* permanently maintained constraints; they are applied to the object that is currently being moved. Moving an object can cause other associations to disappear. Doing the group search dynamically ensures that each time an object is picked, the group that gets assigned to it is the right one at that point in time. Because associations are determined from a selected object towards potential reference objects, but are used in the opposite direction, valid associations between two objects may change by the motion of a third, unrelated object. For example, an alignment association between two concave objects may leave space between the two into which a third object can be inserted, thereby breaking the previous association. To allow for such changes and to ensure robust behavior of the object association framework, every time an object is selected we perform a local search for associated objects dynamically in real time and store them in a separate data structure. For efficiency, likely candidates (that is, those objects that were known to be associated with the selected object previously) are checked first. Then, a general search is started in the vicinity of the selected object, relying on our cell-based spatial subdivision structure used for visibility precomputation and observer tracking [14]. The association procedures (see below) are called for all objects incident to the subdivision cells occupied by the selected source object to see if they are associated with it; each object returns a set of association links, and all of these links together form a graph on the objects in that region. The search efficiently calculates a local closure on this graph to obtain the group of objects linked, directly or indirectly, to the selected object.

To keep the virtual environment interactive and the response to any mouse-directed motions instantaneous, we do not delay the interactive manipulation of the original selected object; we carry out the association search in the background. As soon as an associated object is found, it is subjected to the cumulative set of manipulation transformations applied so far to the source object. This approach has the somewhat startling effect, that when the user grabs and moves a fully loaded desk, some of the objects on the desk may at first remain behind, suspended in mid-air, and will then catch up with the new desk position within a few seconds as they are found to be associated with the desk. We found that most users quickly accept this behavior. To minimize this effect, the association closure graphs, once constructed, are cached in memory, so that any further moves of such a group of objects can be truly instantaneous. The closure process may be safely interrupted before closure is complete if the user decides not to move the chosen object but instead selects a different one. The cache holds whatever portion of the graph was completed, and this potentially useful work is saved; the next time an object in the area is selected, the system will simply pick up the search where it was left off.

This implicit grouping mechanism replaces both the explicit grouping mechanism found in many 2D editors and the inherent grouping resulting from setting constraints between objects. Our mechanism keeps the user focused on the actual positioning of the desired object, while automatically making many of the grouping connections the user would have to make by hand with either of the classical methods. Furthermore, breaking a connection between objects that have been implicitly associated is as simple as grasping the dependent (associated) object and moving it to a new location, at which point the association with the old reference object is broken and a new one is established. Of course, we also give the user the power to override the automatic grouping mechanism by turning it off, or to perform grouping manually by *alt*-clicking objects to explicitly add or subtract them from the current group. The two grouping mechanisms can be active simultaneously; adding an object to a group by *alt*-clicking will then also add any associated objects to that group.

5 Dragging with Relocation Procedures

The *local* motion paradigm – dragging the object with the mouse – is the basic editing move for fine-tuning the position of an object, or for moving objects over short distances; the user selects the object, then moves the mouse pointer in the desired direction. To generate each frame of the motion animation, the *relocation procedure* is first called to convert the cursor position into a constrained position on a suitable auxiliary manifold that depends on the type of association carried by the selected object. The relocation procedure moves the object along the manifold in such a way that the selection point maintains coincidence with the cursor. After the relocation procedure determines the base motion, any relevant *association procedures* are run to determine additional motions that the object must perform to maintain its desired behavior. The association procedures will normally move the object in degrees of freedom not controlled by the mouse; however, if a more constraining motion is desired, it may further restrict the motion on the surface of the 2D manifold. For instance, the association procedure may force an object to move along a 1D path as if dragged by an invisible rubber band between the mouse and the selection point.

When the user initiates an interactive motion by holding down some shift/control key and clicking a mouse button, the relocation procedure is called with arguments corresponding to the current screen coordinates of the mouse, the user's view frustum, the particular drag mode being used (translate or rotate), the selection point on the object, and the original mouse screen coordinates where the object was selected. It first makes an *apriori* selection of one or two preferred DOFs that can be controlled directly and unambiguously with a mouse or with another 2-parameter input device, and which most naturally reflect the basic motion of the selected object. A simple, invisible, auxiliary 2-dimensional manifold, such as a plane, cylinder, or sphere, is established through the current selection point; the only requirement for the auxiliary manifold is that its projection into the view window maps points on the screen 1:1 onto points on the manifold. The object is then moved under mouse control in such a way that its selection point stays on the manifold. The mapping between the cursor motion on the screen and the relocation of the selection point in the 3D virtual world is obtained by intersecting the cursor ray from the eye point with the auxiliary manifold. This gives an intuitive behavior for direct control; the object, grabbed by the user-selected handle, will follow the projection of the mouse movement on a reasonable restricted manifold. In general, these manifolds should be piecewise continuous so that the object will move in a predictable local way for small movements of the mouse.

The manifold used in our *on-horizontal* procedure is simply a horizontal plane through the selection point. In the translation mode, the eye-cursor ray is intersected with the plane equation $z = s_z$, where s is the original coordinate of the selection point. The ray-plane intersection returns some point i; the procedure returns translation vector $i - s$. In the rotation mode, the eye-cursor ray is ignored; the x offset of the mouse pointer on the screen is used as an angle. A rotation by that angle about the plane normal is returned.

On-vertical uses a more complex manifold, composed of piece-wise planar offset surface segments situated in front of the faces of the visible walls in the scene. In the translation mode, the procedure uses the geometric database to intersect the eye-mouse ray with the first surface it hits. If this surface is a vertical one, the intersection point i of the ray with the surface is determined, and the translation vector $i - s$ is returned (where s is, again, the initial coordinate of the selection point). However, the algorithm also computes the rotation angle between the manifold's surface normal at the selection point and at the new point, and returns that rotation to maintain the orientation of the object's "back" with respect to the manifold. This makes wall hangings follow the changes in wall orientation; if a wall hanging is moved around a corner, the rotation causes it to turn its back toward the new wall as it moves. The on-vertical rotation mode simply rotates the object about the normal of the manifold.

After sliding the object along the alignment manifold, the relocation procedure returns a 3D *offset vector* in space, representing the difference between the original pose of the object when it was selected and the new pose indicated by the mouse motion; this represents the fundamental motion intended by the user. This offset position is what is passed on to the *association procedures* for the object.

6 Placement with Association Procedures

At the offset position, the association procedure needs to find the closest valid rest pose for the moving object, given that the latter is supposed to obey some particular behavior.

The first step is to find the possible candidates for alignment. All of our association procedures currently rely on ray projections. Pseudo-gravity and anti-gravity cast rays vertically downward and upward from the selection point, respectively; the objects that these rays hit are the objects with respect to which the selected object's position is adjusted, falling down or up respectively. The on-wall association casts rays in the major horizontal axis directions of the original definition of the object; the closest object in those four directions is the one used for alignment, as the object "falls" sideways against the closest vertical surface.

In these simple procedures, the object does not change its orientation. It is assumed that the object was suitably defined in its local coordinate system, i.e., in a horizontal, aligned position, so that by simply translating it, say, downwards onto (typically horizontal) floors, it will come to rest in the intended position.

Here is the pseudo-gravity procedure in pseudo code:

```
1. While the object O has changed height in the last
   iteration, do:
   (a) Project a ray from the selection point S on object
       O downward to hit some face F of some object A;
   (b) Determine if S is within the bounding box of
       some object B (the smallest bounding box if
       there is more than one);
   (c) if (B is NULL) or (B==A) or (S is visible), drop
       the bottom of O's bounding box to the height of
       F; else, lift the bottom of O's bounding box to
       the height of the top of B's bounding box;
2. Return the total motion of O and associate O with A;
```

In general, this procedure will place the selected object on top of another one that the user points at by using a combination of visibility cues and interference tests (see section 8.1 for discussion of visibility issues). The anti-gravity procedure, used for objects that stick to ceiling surfaces, is identical to pseudo-gravity with the vertical directions reversed ("upward" instead of "downward" and "bottom" for "top"). The on-wall procedure makes some additional assumptions. For an object to attach itself to a wall, it needs to have some notion of a "back-side" which is moved to be coincident with the closest vertical support surface. Since the Soda Hall object descriptions do not carry such a notion explicitly, we assume that the object is defined with its back's surface-normal in one of the major horizontal axis directions. These four directions are then checked for the closest vertical surface, and the pseudo-gravity algorithm is then run along the corresponding axis. Thus when the user first brings such an object into the Soda Hall environment, it needs to be placed close to some wall with its one side that is supposed to act as its back-side.

For every move generated from an offset vector along the relocation manifold, the association procedures decide what local fix-up motions must be made at the new position to implement the desired local behavior for the object (e.g., falling to a supporting surface, in the case of gravity). Each association procedure computes local components of the overall motion, commensurate with the desired object behavior. The motion generated by the association procedures may also cause the object to change from one supporting manifold to another, such as when the motion generated by the relocation procedure would move the object beyond the edge of the current support or into another solid object.

Once the association has determined what local objects and forces affect the motion of the selected object, the offset vector from the relocation procedure is modified to reflect the local motion, and the new vector is returned from the association procedure. The procedure may also optionally return a set of one or more new local associations of the selected object with other objects in its new environment. When the user finalizes the motion by "releasing" the selected object, these new associations replace the original associations that were in effect when the object was selected.

Objects can be placed into the scene directly out of a *knapsack*. This is a standard inventory mechanism based on a temporary buffer with which users can pick up, put down, cut, copy, or paste the currently selected object or group. In case of such a direct placement from a knapsack, the user designates a destination point, but there exists no original object handle location in 3-space from which an offset vector can be calculated; thus, the relocation procedure is bypassed. In these cases, the eye-to-cursor ray is intersected with the first object that it hits, and a previously determined selection point of the object in the knapsack (or the center of the bounding box, as a default) is brought into coincidence with that 3D location. Normally the object to be placed will now be in an inconsistent physical state with respect to objects at the target position; the association procedure(s) for the selected object are called to correct the positioning in an appropriate way, such as lifting it to the surface of the target object under the influence of "pseudo-gravity" or pasting it to the target face if the primary association of the object to be placed is "on-wall." The object can now be re-selected and further fine-adjusted with local dragging motions.

7 Multiple Associations

Multiple association procedures may come into play for single objects. For example, objects like book cases are supposed to obey *pseudo gravity* and simultaneously fit snugly against walls. This may reduce the DOFs of an object to just one or even zero. In the latter case, the object may jump from one desirable location to the next one as the user moves the mouse pointer and the association procedure selects the closest location that fits the desired behavior.

Multiple associations attached to an object type are explicitly ordered. The corresponding procedures are called in a chain, each one receiving the cumulative associations and offsets generated by the one before. A systems programmer assigning combinations of associations to certain types of objects must consider their possible interactions. The interactions can potentially be very complicated since associations are described functionally rather than mathematically; an association procedure can conceivably do anything. Because of this, it is difficult, if not impossible, for the object association framework to generically resolve conflicts between all combinations of procedures. The associations implemented in WALKEDIT are simple and orthogonal and are particularly tailored to the rectilinear, axial environment of Soda Hall; thus, their interactions are easy to predict and not very problematic. The individual adjustments of all associations are gathered into a single cumulative transformation which is then uniformly applied to the selected object and all its dependent associated objects in the dynamically found group.

Figure 1 shows the flow of control, from the inputs to the object association mechanism to its output for an object with a relocation procedure and two association procedures. On the input side, the user selects the object (upper box) and then moves it with the mouse pointer (lower box). Selecting the object launches the implicit grouping search, which proceeds simultaneously with the other operations. The original position of the object and the motion of the mouse are sent into the relocation procedure, which uses the initial position and the mouse motion to determine an offset which is sent through the chain of association procedures. Each association procedure modifies the offset and sends it to the next procedure, while outputting associations. The last procedure also outputs the final motion of the object in 3D space, which is applied to the list of objects output by the implicit grouping search.

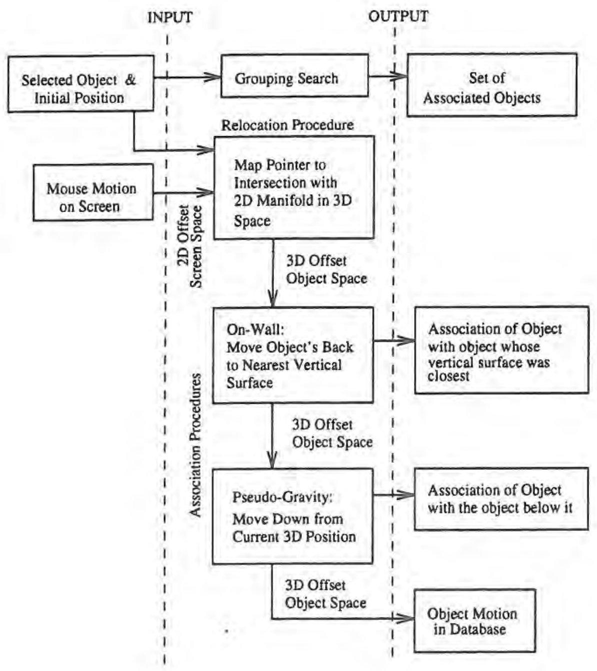

INPUT
OUTPUT
Selected Object & Initial Position
Grouping Search
Set of Associated Objects
Relocation Procedure
Mouse Motion on Screen
Map Pointer to Intersection with 2D Manifold in 3D Space
2D Offset Screen Space
3D Offset Object Space
On-Wall: Move Object's Back to Nearest Vertical Surface
Association of Object with object whose vertical surface was closest
Association Procedures
3D Offset Object Space
Pseudo-Gravity: Move Down from Current 3D Position
Association of Object with the object below it
3D Offset Object Space
Object Motion in Database

Figure 1: A flowchart showing the various procedures at work for an object that obeys on-wall and pseudo-gravity (for example, a bookcase).

An interesting algorithmic question is raised by cyclic constraints arising from the mutual associations of several objects. Imagine placing two "on-vertical" objects back-to-back in the middle of a room. Each object will associate with the other, thus forming a cycle. If object A is selected, object B will dynamically group with it, and will want to rigidly follow the motion of object A; however, object A will want to move along the surface of object B, because its association sees B as the closest vertical surface. Thus the two objects can never again be moved away from their joint back-to-back alignment plane. A similar situation could arise if an "on-ceiling" light fixture is attached to the underside of a table obeying pseudo-gravity. Our current solution involves breaking loops - once they have been detected - at the point where a large object would associate itself with a smaller object. This seems to provide the right feel in a building environment, but may not be a general enough answer.

8 User Interface Issues

While we can start from a few desirable paradigms (see Section 2) to define the user interface for object manipulation in a 3D virtual world, there will always be situations that will put some of these principles in conflict with one another and where there seems to be no obvious "right" answer. A few such tricky problems are raised in this section and our current solutions are discussed.

8.1 Use of Visibility Information

One of the main cues used to disambiguate the depth coordinate during object manipulation is the intersection of the cursor ray with a visible support surface. Thus when moving an object obeying pseudo-gravity, one would typically grab it near its "foot" while looking downwards onto the supporting surface. This establishes a relocation manifold with a reasonable intersection angle with the cursor ray and gives the user good interactive control over the motion. It raises the issue what should happen when the object is dragged beyond the visible range of the support surface or outside the extent of the support altogether. It also raises the issue how one can ever lift an object *off* such a support surface, e.g., to place a book onto a higher shelf.

Figure 2 illustrates a first typical situation. It should be possible to slide a coffee cup underneath a table; thus, we can not simply lift it to the top of the table when the bounding boxes of the cup and of the table start to intersect. Here we use visibility information and our *pointing* paradigm to resolve the issue. As long as the cursor ray clears the table top, the cup stays on the floor. Since no part of the table is between the cup and the floor, and the cup is not actually intersecting the table, the association procedure has no difficulties settling the cup in a valid position on the floor. However, when the ray intersects any part of the table, *and* the bounding boxes of the cup and the table intersect, the cup gets lifted to the top of the table.

Another critical situation is shown in Figure 3. When the cup is dragged beyond the edge of the table top, a non-physical situation occurs. This could be resolved in two ways. The system could try to place the cup where the cursor ray hits a valid support surface. Since the ray may still hit the table top, or perhaps end in a vertical surface, this will not always lead to a useful answer. Thus we have

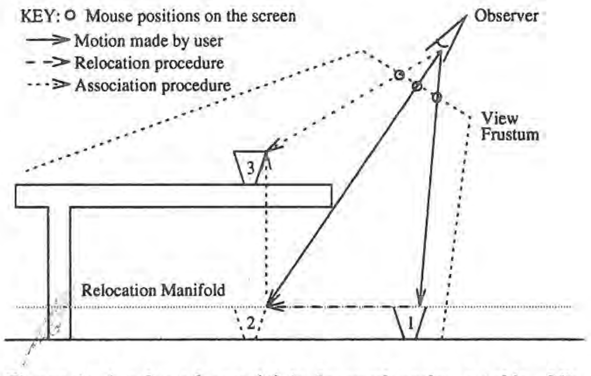

KEY: ○ Mouse positions on the screen
⟶ Motion made by user
- -> Relocation procedure
- - -> Association procedure
Observer
View Frustum
3
Relocation Manifold
2
1

Figure 2: A selected cup (1) is dragged under a table. Visibility information is used to determine when it rises to the tabletop (2); the association procedure modifies both the object position and mouse cursor position (3).

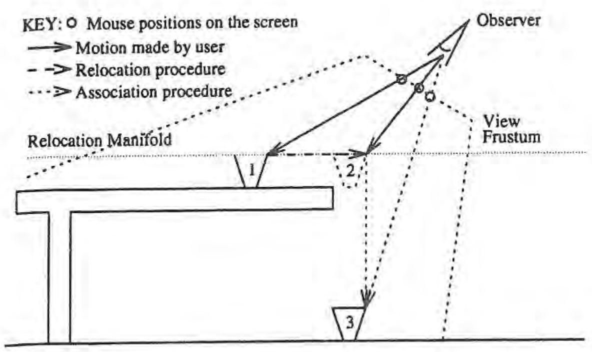


Figure 3: A selected cup (1) is dragged off a table's supporting surface. The cup falls (2) onto the lower surface (3).

found that it makes more sense to give priority to the physical view of the world and drop the cup straight down from the spot where it left the table top to the floor, which then acts as its new support surface.

In all these situations we have an interesting interplay between the teleological and the physical view of our virtual world; visibility information and the intersection of cursor ray with a particular objects are used as additional cues to infer the intent of the user.

8.2 Mouse-Cursor Correspondence

Another key paradigm of the desired user interface is that the object should follow the cursor as directly as possible. This principle needs to be violated necessarily in situations such as the ones above, where establishing a physically valid position may result in a dramatic (vertical) adjustment. As long as the association procedure doesn't add any motion to the object, the relocation procedure usually maintains correspondence. However, the association procedure has no responsibility to maintain the connection between the mouse pointer and the selection point. This then raises the issue whether in such situations the cursor should stay where the user last moved it, or should be "warped" along with the extra motion given to the object by the association procedure. While it is generally preferable to keep the cursor point attached to the handle established at the selection point on the object, this has the consequence that the cursor - and the object itself - may disappear from the screen altogether. Consider the situation in Figure 4 where the cup is moved beyond the *back* end of the table, and where the cursor ray hits no suitable support. The gravity procedure will drop the cup behind the table and possibly out of sight, and the cursor may vanish with it if the floor lies below the lower edge of the viewport. If the fall happens too quickly, the user might not know where the cup has gone and what should be done to bring it back. We have introduced several remedies for this unacceptable situation. First, the cup is made to fall slowly, to imitate reality to some degree and to give the user time to see what is happening. Second, the cursor never disappears entirely from the screen; in the above situation it would be clamped at the lower edge of the viewport. Third, we maintain three axial lines through the selection point on the object to give the user better insight into its position in 3-space. In the above case, the user would thus still see a vertical line emanating from behind the table, giving a clear cue of where the cup currently lies.

To bring the object back into view, the user can move the cursor so that the (invisible) cup moves into the bounding box of the table, whereupon it jumps back to the table top. Alternatively, the user may go to a new location from where the cup is visible, and then continue moving it from its current location on the floor behind the table. Finally, if the object seems totally lost, it can readily be brought into the knapsack while it is still selected, and from there it can be placed directly at the current cursor position. A keyboard shortcut permits to "warp" the object directly from any (possibly hidden) position to the cursor position with a single *ctrl*-click. This operation is also a very efficient way to quickly populate a room with furniture. It takes three mouse operations to place an object in a desired spot: one click to select it, a *ctrl*-click to warp it into the neighborhood of the desired spot, and one *shift*-click-and-drag operation to fine-tune the final position.

9 Software Engineering Concerns

Providing desired object behaviors in 3D virtual worlds is in principle not an easy task. Many nitty-gritty problems concerning data structures and efficient representations must be addressed in order to keep the environment truly interactive. Creating a cohesive framework of object associations is our attempt at keeping this overhead concentrated in one place, so that it can be amortized more easily by the systems programmer with each new object behavior introduced, and so that the user can be given the flexibility of easily choosing the types of behaviors for each object that are most appropriate for the manipulation tasks at hand.

The descriptions of the association and relocation procedures used in the Soda Hall walkthrough look very simple in pseudo-code. It is important to note that the pseudo-code is very close to the level of the actual C code used for the implemented procedures. This is because the WALKTHRU program system provides a rich set of libraries including a complete geometric computation package that operates on vectors, rays, points, planes, and other objects. It also provides the mechanisms to easily search the local area of an object for other objects, to find the objects whose bounding boxes contain a given point, and to quickly find the first object intersected by some space ray. Thus, most lines of pseudo code convert to a few lines of actual C code, making implementation rather straightforward. In such an environment, object associations are most naturally implemented

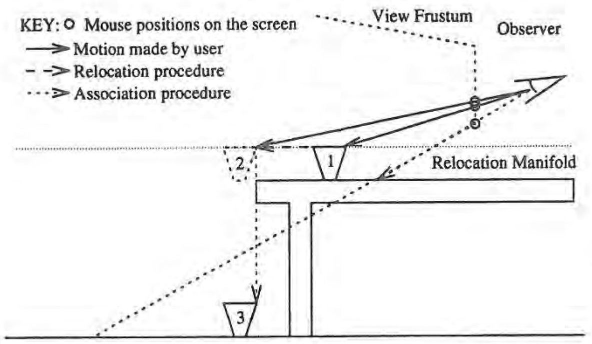


Figure 4: A selected cup (1) is moved off the back of a table (2), falling completely out of the view window (3). The cursor is clamped to the lower edge of the window.

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.

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].

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:

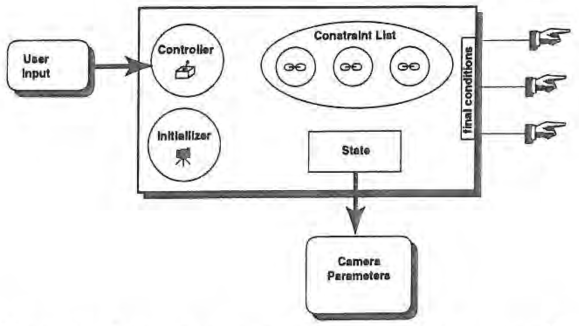


Figure 1: Generic camera module containing a controller, an initializer, a constraint list, and local state

- 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.

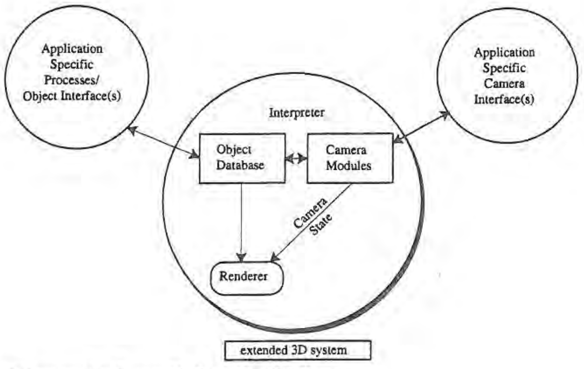

Application Specific Processes/ Object Interface(s)
Application Specific Camera Interface(s)
Interpreter
Object Database
Camera Modules
Camera State
Renderer
extended 3D system

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.

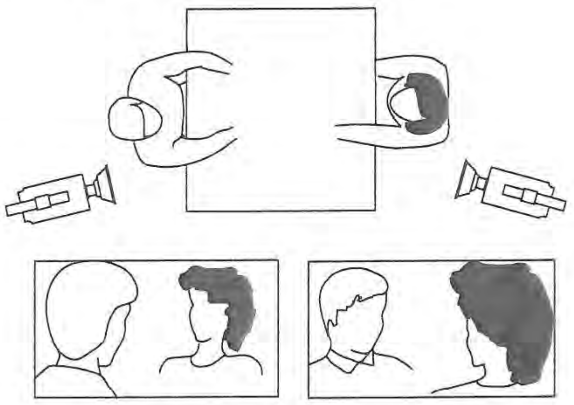


Figure 3: Filming a conversation [Katz88].

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.

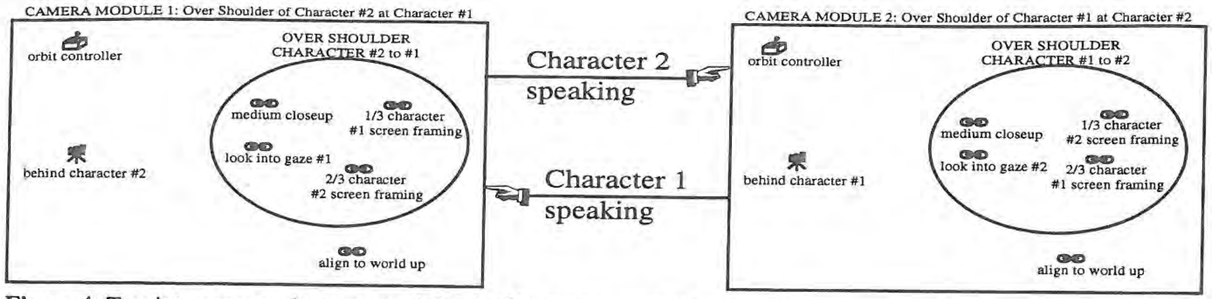

CAMERA MODULE 1: Over Shoulder of Character #2 at Character #1
orbit controller
OVER SHOULDER CHARACTER #2 to #1
medium closeup
1/3 character #1 screen framing
look into gaze #1
2/3 character #2 screen framing
behind character #2
align to world up
Character 2 speaking
Character 1 speaking
CAMERA MODULE 2: Over Shoulder of Character #1 at Character #2
orbit controller
OVER SHOULDER CHARACTER #1 to #2
medium closeup
1/3 character #2 screen framing
look into gaze #2
2/3 character #1 screen framing
behind character #1
align to world up

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.

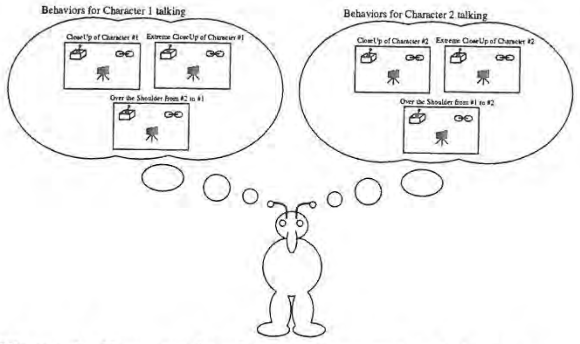

Behaviors for Character 1 talking
CloseUp of Character #1
Extreme CloseUp of Character #1
Over the Shoulder from #2 to #1
Behaviors for Character 2 talking
CloseUp of Character #2
Extreme CloseUp of Character #2
Over the Shoulder from #1 to #2

Figure 5: Conversation filming agent and its behaviors.

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

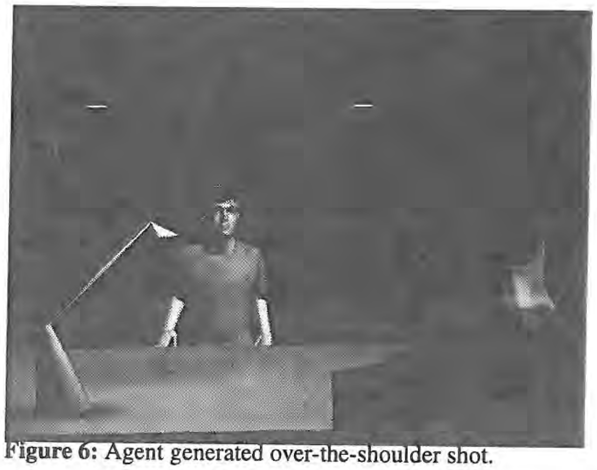


Figure 6: Agent generated over-the-shoulder shot.

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.

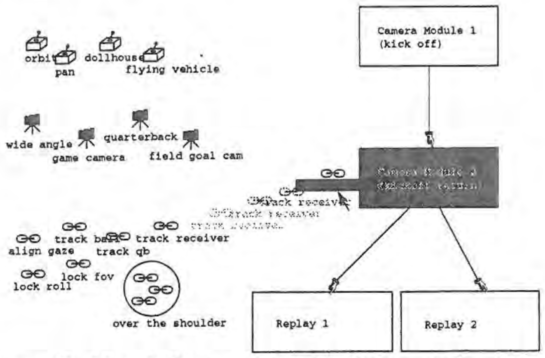


Figure 7: Visual Programming Environment for camera modules

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.

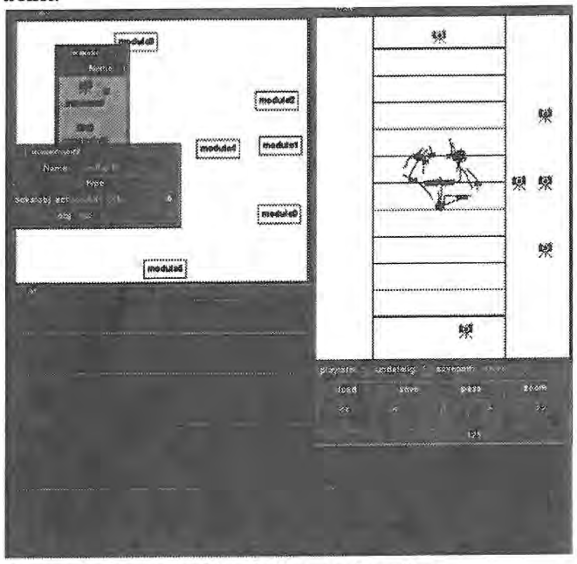


Figure 8: The virtual football game interface

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..

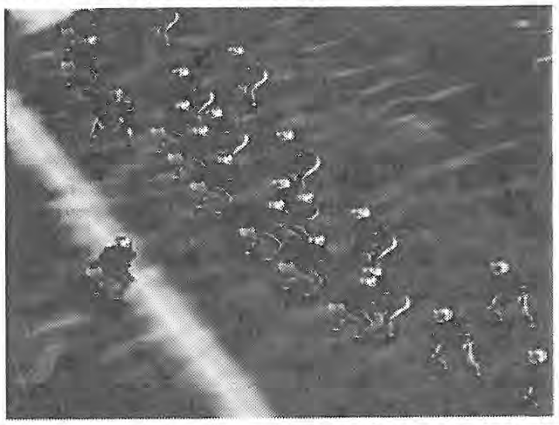


Figure 9: View from "game camera" of virtual football game.

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

1995 Symposium on Interactive 3D Graphics, Monterey CA USA
© 1995 ACM 0-89791-736-7/95/0004...$3.50

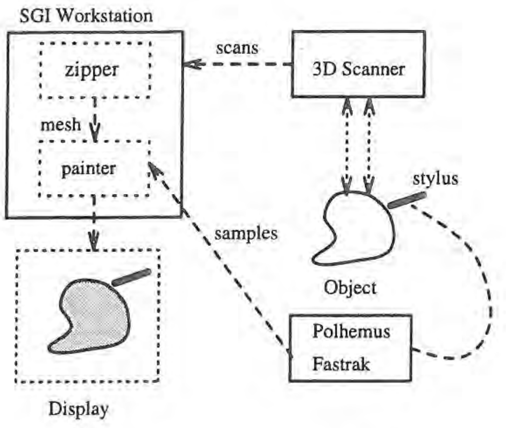


Figure 1: 3D Painting System Configuration

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.

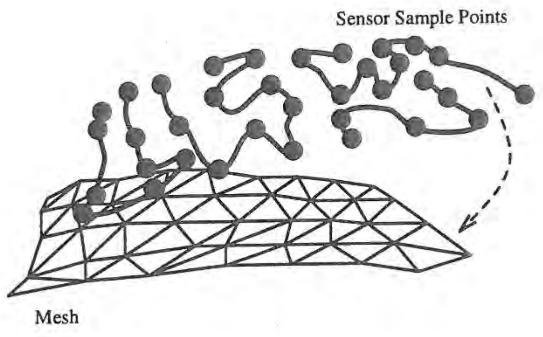


Figure 2: A large set of sensor sample points is collected by running the sensor of the space tracker randomly over the surface of the object. These sensor points are roughly hand-aligned with the mesh, and then Besl's algorithm is used to obtain a more precise alignment.

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.

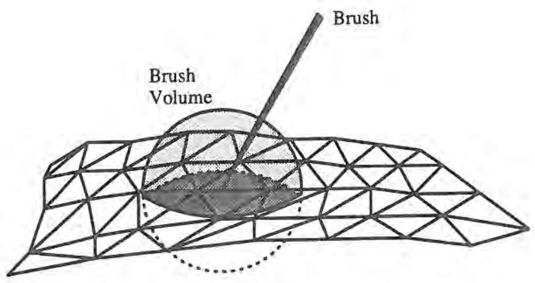


Figure 3: Paint is applied to all mesh vertices falling within the brush volume. Here the vertices in the dark gray region are painted.

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

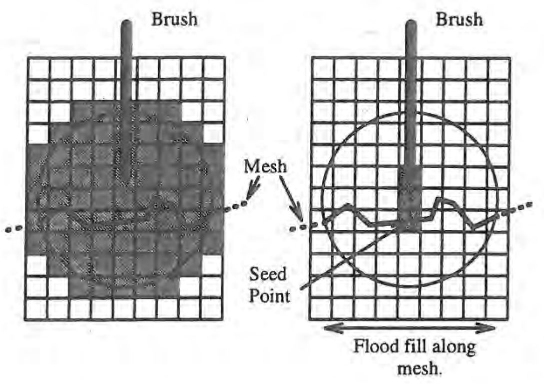


Figure 4: Two methods for determining where to apply paint within a spherical brush volume. The scan-line algorithm walks through every voxel within the brush volume. The flood-fill algorithm extends a ray from the brush tip to the surface and then floods paint out along the surface.

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

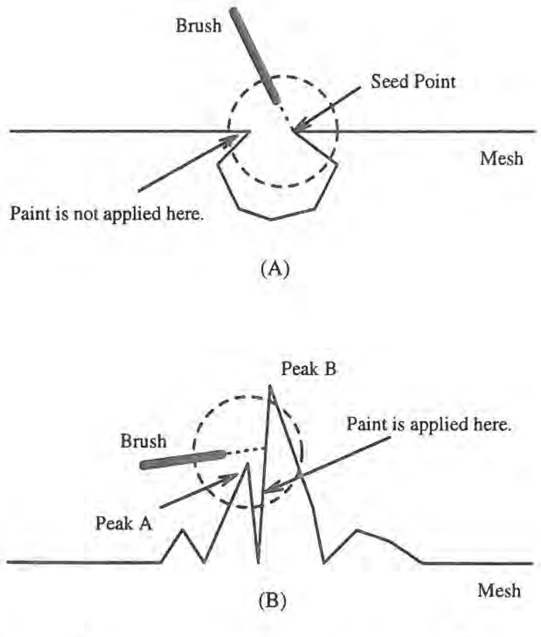


Figure 5: Mesh geometries which cause problems for the painting algorithm.

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.

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

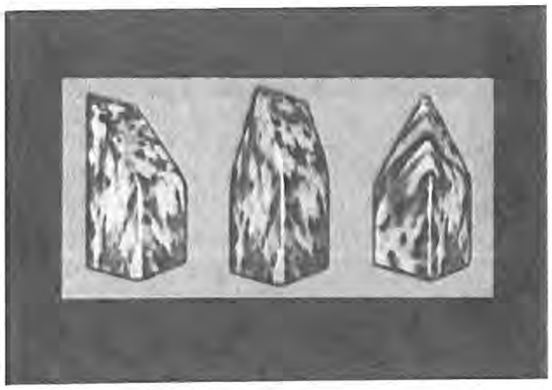


Figure 1: Commonly used carving tools

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: .

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

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:

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

where

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

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

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:

$$A \ \mathit{diff} \ B = A - A\,B. \tag{4}$$

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

$$A \cup B = A + B - A\,B \tag{5}$$

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:

$$\text{for } all \ \alpha - R < i < \alpha + R, \tag{6}$$

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

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

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

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

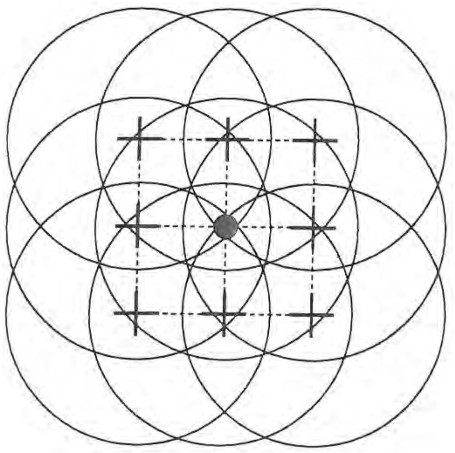


Figure 2: 2D splatting of a density point to its eight neighborhood. • represents the density point (α, β, γ), and + represents the neighboring sample points.

of the filter support and it attenuates linearly to zero at a distance equal to the radius of the filter support, R. Furthermore, the total contribution from the filter should be equal to one. Formally, the hypercone filter is defined as:

$$h(r) = \begin{cases} \dfrac{(R-r)\times w_{\max}}{R} & 0 \le r \le R \\ 0 & otherwise \end{cases} \qquad (7)$$

where

$$\int_r \int_\theta \int_\phi \frac{(R-r)\times w_{\max}}{R}\, dr\, d\theta\, d\phi = 1. \qquad (8)$$

In our implementation, the sawing tool volume is generated by splatting the binary voxelized discrete tool volume using Equation 6.

To speed up the splatting process, the contribution to the sample points in the neighborhood is precomputed and stored in a lookup table. The splat lookup table consists of $(2R-1)^3$ entries, corresponding to the $(2R-1)\times(2R-1)\times(2R-1)$ splat neighborhood samples. Since the splat lookup table is neither object- nor feature-dependent, various shapes and sizes of tools can be generated on the fly using this technique.

6. Rendering

In our system a ray casting and compositing approach (e.g., [7]) is used to render the volume-sampled objects which are represented in a 3D volume raster of voxels. That is for each pixel on the image plane, a continuous parametric ray is cast toward the raster to determine the pixel color. Commonly, samples are taken uniformly along the ray to test for ray-object intersection. For accuracy, our ray-object intersection point is calculated analytically by using the trilinear function and the equation of the line. In this section we describe an extended ray casting technique that takes advantages of our volume-sampled models to eliminate image space aliasing. In addition, localized ray casting method is presented for interactive image updating during sculpting.

Thomas et al [13] developed a technique for antialiased ray tracing of surface-based geometric primitives. In their algorithm a pair of proximity covers are built to enclosed each object in the scene. These covers are used to detect when a ray is near the proximity of an actual surface. Once the proximity of a silhouette edge is detected, the distance between the ray and the silhouette edge is used for edge filtering. Our antialiased ray casting algorithm utilizes the filtered surfaces that surrounded our volume-sampled objects. These filtered surfaces have the property of diminishing density as one travels from the interior of the object to the empty space. This smooth transition of density from object to empty space enables one to easily approximate the distance, ray_dist, of the ray from an actual surface. More precisely, as one steps through each sample point a, along the parametric ray, the sample-object distance is approximated by the formula:

$$dist(a) = \begin{cases} > R & \text{if } f(a) = 0 \\ 0 & \text{if } f(a) \ge isovalue \\ R\,(1 - \dfrac{f(a)}{isovalue}) & \text{if } 0 < f(a) < isovalue \end{cases} \qquad (9)$$

where R is the radius of the filter support and $f(a)$ represents the density at the current sample point a. Then, the closest ray-object distance can be approximated by

$$ray\_dist = \min(dist(a)), \text{ for } all\ a \in ray. \qquad (10)$$

Once it is known that a ray is near the silhouette of an object, that is its ray_dist is greater than 0 and less than R, the filtered color of the pixel is approximated by

$$C_{pixel} = (1 - \frac{ray\_dist}{R})\, C_{object} + (\frac{ray\_dist}{R})\, C_{backgrd.} \qquad (11)$$

Therefore, a 2D antialiased image of the projection is produced without the need for image space supersampling.

Furthermore, by taking advantage of the fact that a typical sculpting action only modifies a small region of the object volume, we only need to cast new rays at those pixels which are affected by the modified region. Consequently, interactive rendering speed is achieved. However, if the object volume is rotated or translated, then the entire image needs to be regenerated, preferably using a progressive refinement or low-resolution ray casting approach. In the progressive refinement approach, the image is generated in a multi-pass fashion by pixel sub-sampling. Thus, a low-quality image is immediately available and the user can stop the rendering process if necessary. Alternatively, the low-resolution ray casting method is a one-pass algorithm which performs image space interpolation, in addition to pixel sub-sampling. Our localized ray caster has advantages over the localized Marching Cube algorithm in that ray casting employs an image-order technique to determine view point visibility. Thus, image updating of the modified regions is trivially done. On the other hand, the object-order approach of Marching Cube rendering complicates the local image updating process, since it is difficult to determine which hidden polygons will become visible after a sculpting operation.

7. Results and Implementation

The VolVis system [1] provides an ideal framework for our volume sculpting system. VolVis, developed at SUNY Stony Brook, is a comprehensive, diversified, and high performance volume visualization system, which is use extensively around the world. It supports manipulations of multiple volumes and a variety of rendering algorithms, ranging from highly accurate volumetric ray tracing to fast rough approximation. The metaphor of sculpting a volumetric solid has been found to be intuitive, and its ease of use has allowed even a novice user to learn the system in minutes. Our sculpting tool has been successfully used to create many realistic objects. Figure 3 illustrates the process of sculpting a chair from a block of wood. The original block of wood is of $75 \times 125 \times 75$ resolution. The 3D volume wood texture has been applied to the object to give it the realistic appearance. Note that, unlike traditional surface graphics, applying texture in our volume graphics representation does not introduce additional processing cost since texture color is pre-stored within each voxel as a view independent attribute. In Figure 4 a chair and a table, generated with our system, are placed in a room and rendered with volumetric ray tracing [12]. The lamp and the goblet are each generated by revolving a volume-sampled curve around a circular base. There is not a single polygon in this figure. Additional sculpted objects include the cello and chair in Figure 5, and a gazebo and bench in Figure 6. The sculpted windmill shown in Figure 7 is placed on a synthesized volumetric fractal terrain. The smoke from the chimney is a simulated data of a ventilated air flow.

8. Acknowledgements

Special thanks to Rick Avila and Lisa Sobierajski for their 3D textures and volumetric ray tracer.

9. References

1. Avila, R., He, T., Hong, L., Kaufman, A., Pfister, H., Silva, C., Sobierajski, L. and Wang, S., "VolVis: A Diversified Volume Visualization System", *Visualization '94 Proceedings*, Washington, DC, October 1994, 31-38.

2. Coquillart, S., "Extended Free-Form Deformation: A Sculpting Tool for 3D Geometric Modeling", *Computer Graphics (Proc. SIGGRAPH)*, **24**, 4 (August 1990), 187-196.

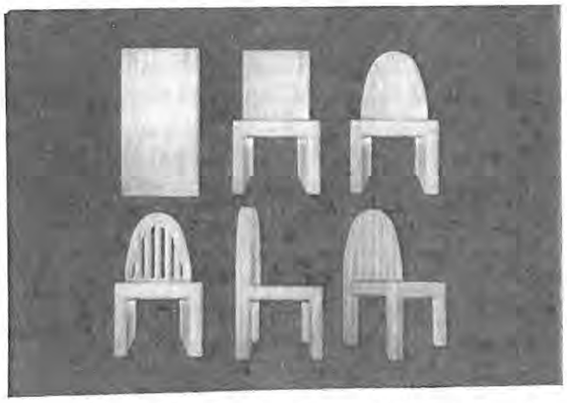


Figure 3: Sculpting of a chair from a block of wood.

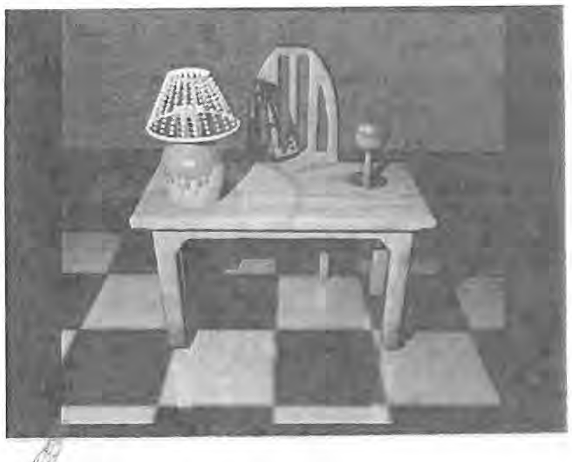


Figure 4: A volumetric ray traced scene of a room which includes a sculpted chair and table.

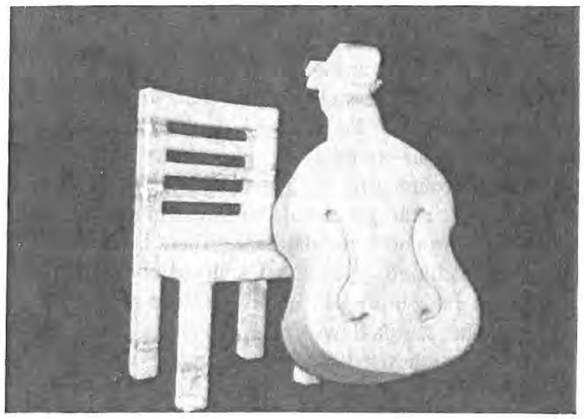


Figure 5: Sculpted cello and chair.

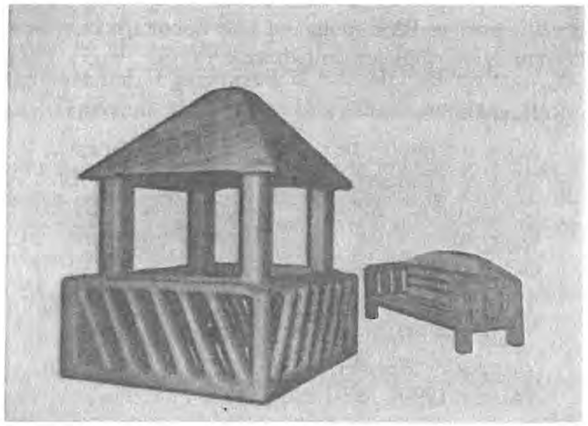


Figure 6: Sculpted gazebo and bench.

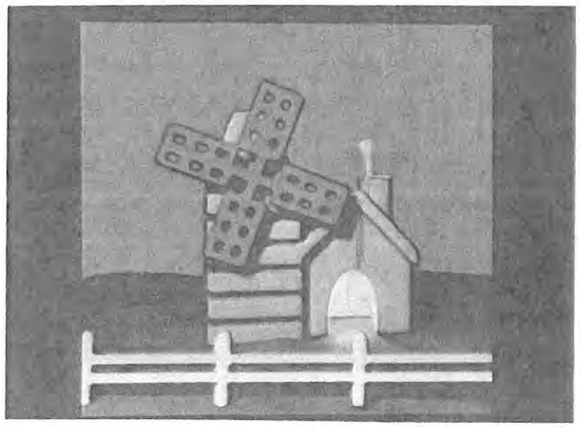


Figure 7: Sculpted windmill on a fractal volumetric terrain.

3. Dubois, D. and Prade, H., *Fuzzy Sets and Systems: Theory and Applications*, Academic Press, 1980.
4. Galyean, T. A. and Hughes, J. F., "Sculpting: An Interactive Volumetric Modeling Technique", *Computer Graphics (Proc. SIGGRAPH)*, **25**, 4 (July 1991), 267-274.
5. Kaufman, A., "The *voxblt* Engine: A Voxel Frame Buffer Processor", in *Advances in Graphics Hardware III*, A. A. M. Kuijk, (ed.), Springer-Verlag, Berlin, 1992, 85-102.
6. Kaufman, A., Cohen, D. and Yagel, R., "Volume Graphics", *IEEE Computer*, **26**, 7 (July 1993), 51-64.
7. Levoy, M., "Display of Surfaces from Volume Data", *Computer Graphics and Applications*, **8**, 5 (May 1988), 29-37.
8. Naylor, B., "SCULPT: An Interactive Solid Modeling Tool", *Graphics Interface '90*, May 1990, 138-148.
9. Pentland, A., Essa, I., Friedmann, M., Horowitz, B. and Sclaroff, S., "The ThingWorld Modeling System: Virtual Sculpting By Modal Forces", *Computer Graphics*, **24**, 2 (March 1990), 143-146.
10. Saito, T. and Takahashi, T., "NC Machining with G-Buffer Method", *Computer Graphics (Proc. SIGGRAPH)*, **25**, 4 (July 1991), 207-216.
11. Sederberg, T. W. and Parry, S. R., "Free-form Deformation of Solid Geometric Models", *Computer Graphics (Proc. SIGGRAPH)*, **20**, 4 (August 1986), 151-160.
12. Sobierajski, L. and Kaufman, A., "Volumetric Ray Tracing", *Volume Visualization Symposium Proceedings*, Washington, DC, October 1994.
13. Thomas, D., Netravali, A. N. and Fox, D. S., "Anti-aliased Ray Tracing with Covers", *Computer Graphics Forum*, **8**, (1989), 325-336.
14. Van Hook, T., "Real-Time Shaded NC Milling Display", *Computer Graphics (Proc. SIGGRAPH)*, **20**, 4 (August 1986), 15-20 .
15. Wang, S. W. and Kaufman, A. E., "Volume-Sampled 3D Modeling", *IEEE Computer Graphics & Applications*, **14**, 5 (September 1994), 26-32.

The Tecate Data Space Exploration Utility

Peter Kochevar*
Digital Equipment Corporation

Len Wanger
San Diego Supercomputer Center

Abstract

A new prototype, interactive visualization system is described which is designed to allow anyone to browse for and then visualize data within general data spaces. The prototype, called Tecate, capitalizes on the architectural strengths of current scientific visualization systems, network browsers like Mosaic, database management system front-ends, and on virtual reality systems. Tecate is able to browse for data contained in databases managed by database management systems, and it can browse for information contained in the World Wide Web. In addition, Tecate dynamically crafts user-interfaces and interactive visualizations of selected data-sets with the aid of an intelligent system. This system automatically maps most kinds of data into a virtual world that can be explored directly by end-users.

1 Introduction

With the proliferation of computer networks, the number, size, and accessibility of *data spaces* has increased dramatically. A data space is any data source or repository whose access is controlled via a well-defined software interface. Some examples of data spaces are a database managed by a database management system, the World Wide Web, and any data object, in the object-oriented sense, that resides in a computer's main memory whose components are accessible via the object's methods.

In order to learn, conduct commerce, and entertain in a world that is increasingly being abstracted away as a collection of data spaces, a general, interactive tool is needed to foster data space exploration. This tool should allow any end-user to both browse the contents of data spaces as well as allow them to inspect, measure, compare, contrast, and identify patterns in selected data-sets. Combining both tasks into one tool is both elegant and utile in that end-users need only learn one system to seamlessly pass back and forth between browsing for data and then assimilating it.

To best perform its function, a tool for exploring data spaces should be able to

- interface to general data spaces
- saliently visualize most any kind of data whether it be scientific data or the listings in a telephone book
- dynamically craft user-interfaces and interactive visualizations based on what data is in hand, who is doing the visualizing, and for what reason
- be highly interactive.

There are systems available today that have some of these capabilities but no one system possesses all of them. Data visualization systems, such as AVS [1], Khoros [11], or Data Explorer [8], are very capable of visualizing scientific data but they are very poor at interfacing to general data spaces, provide only limited interactivity within visualizations themselves, and require visualizations to be crafted by hand by knowledgeable end-users. Network browsers like Mosaic are good at fetching data from certain types of data spaces but they are very limited in the variety of data they can directly visualize, and they offer a very restricted type of interactivity. Finally, front-ends to database management systems provide very elaborate querying mechanisms for selecting data from a database but they do not have sophisticated means for visualizing and further exploring query results.

To address the need for a comprehensive tool that will effectively explore the informational content of data spaces, a prototype system called *Tecate* has been created. The architecture of Tecate borrows from that of visualization systems, network browsers, and database management systems as well as from virtual reality systems like Alice [18] and MR/OML [6]. A major contribution of Tecate is the incorporation of the architectural strengths of these systems into one coherent whole.

From the outset, Tecate was conceived as an object-oriented system where objects are imbued with behaviors that can aid in data exploration and knowledge creation. In particular, Tecate enables the browsing for data in a database management system or the World Wide Web via user-interaction with graphical renditions of objects that represent data features. To Tecate, the results of any database query or Web fetch issued while browsing is data that requires appropriate visualization. To help in this regard, Tecate has within it

* *Contact address:* Peter Kochevar, DEC/San Diego Supercomputer Center, P. O. Box 85608, San Diego, CA 92186-9784, kochevar@sdsc.edu.

an intelligent system that automatically maps most kinds of data into a virtual world that can be explored directly by an end-user.

2 The Abstract Visualization Machine

The heart of Tecate is an object management system called the *Abstract Visualization Machine (AVM)*. All major components of Tecate as well as entities appearing in virtual worlds created by Tecate are *objects* that communicate with one another via message passing. The AVM is responsible for creating, destroying, altering, rendering, and mediating communication between these objects. The two major components of the AVM are the *Object Manager* and the *Rendering Engine* (see Figure 1).

2.1 Tecate's Object Model and the Abstract Visualization Language (AVL)

Tecate uses the delegation model of inheritance [25] in which no distinction is made between classes and instances as in languages like C++. In Tecate, there is a single object creation operation, *clone*. Any object in the system can serve as a *prototype* from which an exact copy can be made through the clone operation. A clone inherits properties from its prototype by copying the prototype's properties, but any such property can be altered or removed so that a clone can take on an identity of its own.

Tecate objects possess four classes of properties:

- Attributes that affect an object's visual appearance such as geometric and topological structure, color, texture, material properties, etc.
- Behaviors determined by the set of messages objects send and to which they respond.
- A collection of variables whose values represent an object's state.
- A list of sub-objects that are *parts-of* a given object just as a wheel is a part-of a car.

Although most users of the system uniformly see communicating objects, a distinction is actually made between two kinds of objects based on how they are implemented. *Resource objects* are implemented primarily as external processes using some compilable general purpose programming language such as C or Fortran. Objects that have compute-intensive behaviors or whose behavior executions are time-critical are generally implemented as resource objects. For instance, most Tecate objects that provide system services, such as rendering, are implemented as resource objects.

Objects populating virtual worlds that represent data features are implemented differently than resources using an interpreted programming language called the *Abstract Visualization Language*. Such objects are termed *dynamic objects* because they may be created, destroyed, and altered on-the-fly as a Tecate session unfolds. Nonetheless, the ability to dynamically add, remove, and alter object properties is not solely endemic to dynamic objects. Resource properties may also be changed on-the-fly because resources are actually implemented with a dynamic object that "fronts" for the portion of the resource that is implemented as an external process.

The Abstract Visualization Language (AVL) that is used to specify dynamic objects is the "native" language of the AVM. It is built upon the Tcl embeddable command language [15], but AVL extends Tcl by adding object-orientedness, 3-D graphics, and a more sophisticated interaction handler. AVL is a typeless language that is capable of performing arbitrary computations. Through the language, object properties are specified and manipulated, and object behaviors are invoked by sending messages from one object to another. To give a flavor of programming in AVL, Figure 2 depicts a code fragment that when interpreted implements a 3-D icon representing a World Wide Web (WWW) site. In general, AVL may be viewed as an example of a virtual reality markup language (VRML) [17, 19], an analog of the hypertext-based HTML that underlies network browsers like Mosaic.

```
clone Hyperlink Visual

add Hyperlink {
   appearance {
     shape {sphere}
     diffuseColor {0.0 0.0 1.0}
     repType {surface}
   }
   behavior {
     # Initialize hyperlink
     init {url desc} {
       addstate url $url
       addstate description $desc
       add scene1 "subobject [getself]"
       send window1 addEvent
         "[getself] {Button-1 {openUrl {}}}"
     }

     # Open the URL
     openUrl {} {send www fetch "[getstate url]"}
   }
}

clone scene1 Visual
clone window1 CameraWindow
send window1 init {scene1}

clone hlink1 Hyperlink
send hlink1 init {"http://gopher.sdsc.edu/Home.html"
     "SDSC home page"}

# Use the SDSC model geometry
add hlink1 {appearance {shape {AliasObj "sdsc.tri"}}}
```

Figure 2: An implementation of a WWW icon in AVL.

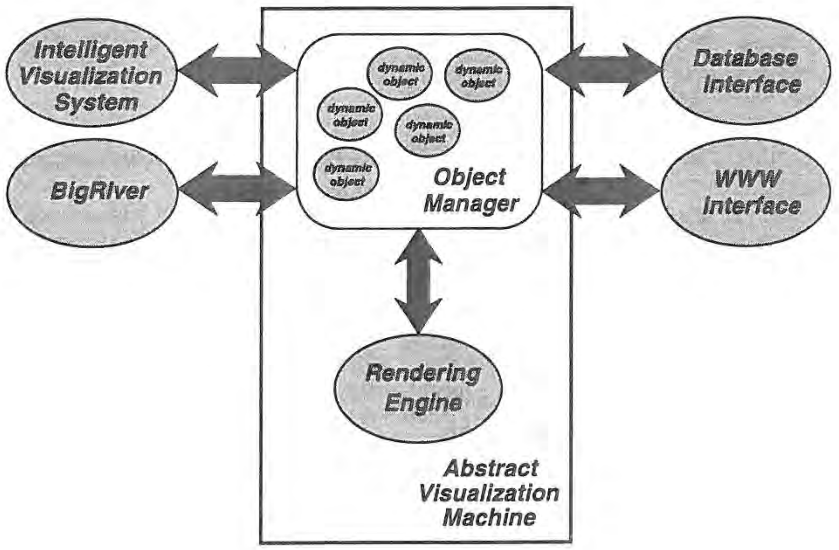


Figure 1: An architectural diagram of the Tecate data exploration utility.

2.2 The Object Manager

The Object Manager is the primary workhorse within the AVM. It is responsible for interpreting AVL programs, managing a database of objects, mediating communication between objects, and interfacing with input devices. The Object Manager is itself a resource object that is distinguished by the fact that all other resource objects are spawned from this one object. In addition, the Object Manager is responsible for creating a distinguished dynamic object, called *Root*, from which all other dynamic objects can trace their heritage via prototype-clone relationships.

The Object Manager is implemented upon a simple, custom-built *thread* package. Each object within Tecate can be thought of as a process that has its own thread of control. Threads can either be implemented as "lightweight processes" that share the same machine context as the Object Manager's operating system process, or they can be implemented as their own operating system process separate from that of the Object Manager. Within Tecate, dynamic objects are implemented as lightweight processes while resource objects are implemented as "heavyweight" operating system processes which may or may not have an attached lightweight process as an adjunct. A low-level function library is provided to handle the creation and destruction of threads, and to handle inter-thread communication between two threads regardless of how they are implemented.

2.3 The Rendering Engine

The Rendering Engine is a special resource object wholly contained within the AVM. It is responsible for creating a graphical rendition of a virtual world that is specified by AVL programs interpreted by the Object Manager. When interpreting an AVL program, the Object Manager strips off appearance attributes of objects and sends appropriate messages to the Rendering Engine so that it can maintain a separate display list representing a virtual world. These display lists are represented as directed, acyclic graphs whose connectivity is determined by object-subobject relationships that are specified within AVL programs.

The present Rendering Engine implementation makes use of the Doré graphics package [13] running on an Alpha workstation augmented with a Kubota Denali graphics processor. The display lists that are created by invoking behaviors within the Rendering Engine are actually built up and maintained through Doré. The set of messages that the Rendering Engine responds to represents an interface to a platform's graphics hardware which is independent of both the device and graphics package.

3 Object-Object Interactions and Input/Output

The primary focus within Tecate is on object-object interactions. These interactions occur primarily by sending messages from one object to another, and objects can send messages to themselves which has the effect of making a local function call. Unlike systems such as OpenInventor [24], rendering is not a central activity within Tecate but is rather just a side-effect of object-object interactions. In this sense, Tecate is more like VR programming systems like Alice and MR/OML although Tecate is far more flexible.

In Tecate, objects can create and destroy other objects, and alter the properties of existing objects on-the-fly. Presently, all of an object's properties are visible to any other object and hence those properties can be manipulated externally. In the future, some form of selective property hiding may prove useful so that designated properties of an object would not be alterable by other objects.

A powerful feature of Tecate is its ability to dynamically establish object-subobject relationships among objects. This feature provides a mechanism with which to build assemblies of parts much like what is done in classical hierarchical graphics systems like Doré or OpenInventor. But, this feature also provides the capability to create sets or aggregates of objects that share some trait such as being highlighted. Tecate, through AVL, allows all objects within a set to be treated en masse by providing a means to selectively broadcast messages to groups of objects. A message that is sent to an object can be forwarded to all of the object's subobjects. Thus one object can serve as a "container" for all other objects that are highlighted, the highlighted objects merely being subobjects of the container. If one wants to unhighlight all highlighted objects, only one unhighlight message needs to be sent to the container object which then forwards the message to all its subobjects. In general, an object can be the subobject of any number of other objects and thus simultaneously be a member of many different sets.

The handling of user-input within Tecate is intended to appear no different than ordinary object-object interactions. All physical input devices known to Tecate have an "agent" object associated with them that acts like a device handler. All objects wishing to be informed of a particular input event register themselves with the appropriate agent. When an input event occurs via a user-interaction, all registered objects are sent a message notifying them of the event. Complex events, such as the occurrence of event A and event B within a specified time period, can easily be defined by creating new handler objects. These handlers register to be informed of separate events, but then turn around and inform other objects of the conjunction of the events.

To aid in simulating physical processes or to help in performing animations, Tecate provides a predefined *clock* object that pulses every millisecond. If objects wish to be informed of a clock pulse, those objects register themselves with the clock object just like objects register themselves with input device agent objects. The default clock object can be cloned and each clone can be instantiated with a different clock period. Any number of clock can be ticking simultaneously during a Tecate session. Since new clocks can be created dynamically, and objects can register and unregister to be informed of clock pulses on-the-fly, clocks can be used both as timers and triggers as well as pacesetters.

Besides a clock, Tecate also provides predefined objects that represent windows, lights, and cameras. These objects are considered "abstract" objects in the sense that they are not intended to be used directly but rather they are to serve as prototypes from which clones can be created. It is these clones that are used to illuminate and render given virtual worlds.

4 Application Resources

Tecate comes predefined with a number of resource objects that aid in interactively visualizing data. These objects include the Intelligent Visualization System, the Database Interface, the World Wide Web Interface, and a visualization programming system called BigRiver (see Figure 1). Additional system resources can be added easily by an application programmer using tools provided with the base Tecate system. A new resource can be built around either a user-written program or a commercial off-the-shelf application.

4.1 The Intelligent Visualization System

The Intelligent Visualization System (IVS) is provided to dynamically build interactive visualizations and user-interfaces, and to aid non-expert end-users in exploring data spaces. This knowledge-based system is similar in concept to other expert visualization systems [5, 9, 14, 21], and it has been described in detail elsewhere [2, 12]. The IVS automatically crafts virtual worlds based on a *task specification* and on a description of the data that is to be visualized. A task specification represents a high-level data analysis goal of what a user hopes to understand from the data. For instance, a user may wish to determine if there is any correlation between temperature and the density of liquid water in a climatology data-set.

From the data description and task specification, a *Planner* within the IVS produces a data-flow program which when executed builds an appropriate virtual world that represents a selected data-set. The Planner makes use of a collection of rules, definitions, and relationships that are stored in a *Knowledge Base* when building a visualization that addresses a given task specification. Contents of the Knowledge Base include knowledge about data models, user tasks, and visualization techniques. The Planner functions by constructing a "sentence" within a data-flow language defined by a context-sensitive graph grammar. Presently, the Knowledge Base is implemented using the Classic knowledge representation system [20] while the Planner is implemented in CLOS [22].

4.2 BigRiver

The data-flow program produced by the Intelligent Visualization System is written in a scripting language that is interpreted by BigRiver, a visualization programming system similar to AVS [1] or Khoros [11]. BigRiver consists of a collection of procedures called *modules* each of which has a well-defined set of inputs and outputs. Functional specifications for these modules represent some of the knowledge contained in the Intelligent Visualization System's Knowledge Base. Visualization scripts that are interpreted by BigRiver specify module parameter values and they dictate how the outputs of chosen modules are to be channeled into the inputs of others.

BigRiver modules come in one of three varieties: I/O, data manipulators, and glyph generators. All modules make use of self-describing data formats based on the mathematical notion of *fiber bundles* [3, 4, 7]. One format is used for manipulation within memory while the other is an "on-the-wire" encoding that is meant for transporting data across a network. An input module is responsible for converting data stored in the on-the-wire encoding into the in-memory format. The data manipulator modules transform fiber bundles of one in-memory format into those of another. The glyph generators take as input fiber bundles in the in-memory format and produce AVL programs which when executed build virtual worlds containing objects that represent features of selected data-sets. A single display module takes as input AVL code and passes it to the Abstract Visualization Machine where the appearance attributes of objects are used to create an image of a virtual world containing the objects via the Rendering Engine.

4.3 The Database Interface

The Database Interface (DBI) provides the means to interact with a database management system (DBMS), which in the current version of Tecate can either be Postgres [23] or Illustra [10]. Database queries are sent to the DBI by Tecate objects where they are handed off to a DBMS server for execution. Query results are returned from the server to the DBI which then packages them up as an on-the-wire encoding of a fiber bundle buffered on local disk. A description of the fiber bundle and the location of the buffer are sent back to the object that made the query request of the DBI. That object might then request the Intelligent Visualization System to structure a virtual world whose image would appear on the display screen by way of BigRiver and the Rendering Engine. Objects in the virtual world can be given behaviors that are elicited by user interactions which might then result in further database queries and so on. Chains of events such as these provide a means for browsing databases via direct manipulation of objects within a virtual world.

4.4 The World Wide Web Interface

The World Wide Web Interface functions similarly to the DBI but rather than access data in a database, the Interface provides access to data stored on the Web. Messages containing Universal Resource Locators (URLs) are passed to the Web Interface which then fetches the datafiles pointed to by the URLs. In retrieving data from the Web, the Web Interface uses Mosaic's Web software library.

Once a datafile is fetched, the Web Interface attempts to translate its contents into an AVL program which is then passed to the Object Manager for interpretation. The AVL either specifies the creation of a new virtual world representing the datafile's contents, or it specifies new objects that are to populate the current world being viewed. If the fetched datafile contains a stream of AVL code, the Web Interface merely forwards it to the Object Manager. If the datafile contains general data in the form of an on-the-wire encoding of a fiber bundle, the Web Interface appeals to the Intelligent Visualization System to structure an appropriate virtual world. If the datafile contains a stream of HTML code, the Web Interface invokes an internal HTML interpreter which produces an AVL program that is then interpreted by the Object Manager. This interpreter actually understands an extended version of HTML that supports the direct embedding of AVL within HTML documents.

5 Putting It All Together

When a Tecate session is initiated, a rendition of a base virtual world is presented through which Tecate's various services can be utilized. This virtual world, and possibly others, is specified by an AVL program that is stored in a file which is automatically read in at system start-up. This program is completely arbitrary and configurable but it should serve to initiate the browsing of data spaces and the subsequent visualizations of individual data-sets. In Tecate, the present default virtual world is an abstract landscape populated with icons that when selected, allow end-users to initiate either the browsing of data within databases or the WWW.

5.1 Visualizing Data in a Database

When an end-user decides to browse for data in a database, a new virtual world is overlaid upon the default one. This world is built up from a toolkit of user-interface widgets where each widget is a Tecate dynamic object. Because there is not yet a comprehensive 3-D widget set within Tecate, some widgets still rely on 2-D constructs provided by the Tk widget set [16].

In the database world, the *MapQuery Tool* is provided so that graphical queries can be made for Earth science datasets whose geographical extents and timestamps fall within user-specified constraints. The tool is built around a world map upon which regions of interest can be specified (see Figure 3). Once a user marks a region of interest on the map and selects a temporal range, a query message is sent to the Database Interface. The result of the query is returned to the Mapquery Tool which then forwards it to the Intelligent Visualization System, accompanied by a **select** task directive, so as to structure an appropriate visualization. The ensuing script produced by the IVS is eventually executed by BigRiver where a stream of AVL code is produced that is sent to the Abstract Visualization Machine for interpretation.

This AVL program creates a new virtual world consisting of a collection of 3-D icons each corresponding to one dataset that was returned as the result of the initial query. Each icon is a Tecate object whose physical appearance is a function of data-set type. The Intelligent Visualization System also builds in two behaviors for each icon. Depending on how an icon is selected by a user, either the meta-data associated with the data-set represented by the icon is displayed in a separate window, or a query message is sent to the Database Interface requesting the actual data. In the latter case, the query result is again forwarded to the Intelligent Visualization System after prompting the user for a task specification. Yet another virtual world containing objects representing data features is created and displayed with the aid of BigRiver and the Abstract Visualization Machine, and so on (see Figures 4 and 5). In general, data exploration proceeds this way by creating and discarding virtual worlds based on interactions with objects populating prior worlds.

5.2 Browsing the World Wide Web

If an end-user chooses to browse the World Wide Web, the default virtual world is supplanted by a new one that depicts a map of the Earth arrayed in 3-D. Select Web sites are positioned in the world as 3-D icons as can be seen in Figure 6. Each icon is cloned from a single "Hyperlink" prototype object that uses a state variable to store a URL. Each Web site icon inherits a behavior that causes a datafile pointed to by its URL state variable to be fetched when the icon is picked.

Once selected, the home page for a Web site is visualized on the base of an inverted pyramid whose apex is centered on the chosen icon (see Figure 7). The text and imagery for the home page appears similarly as it would when visualized using a hypertext-based browser like Mosaic. Highlighted text has a hyperlink associated with it that can be followed by picking the text. Such highlighted text is a Tecate object that also is cloned from the same Hyperlink prototype object as the Web site icons. If upon following a hyperlink another HTML document is retrieved, that document is viewed upon the base of another inverted pyramid whose apex rests on the selected text, and so on (see Figure 8). Rather than having to page back and forth between hypertext documents as with Mosaic, in Tecate an end-user need only "fly" about the virtual world to gain an appropriate viewpoint from which to view a desired document.

6 Conclusion

Tecate is an ambitious system which seeks to bring together into one package useful features found in visualization systems, network browsers, database front-ends, and virtual reality systems. As a first prototype, Tecate was created using a "breadth-first" development strategy. That is, it was deemed essential to first understand what components are needed to build a general data space exploration utility, and then determine how those components interact. This development strategy traded off functionality of individual components for the completeness of a fully running visualization system.

In the future, "depth" needs to be added to the Tecate system components. For instance, the present Knowledge Base within the Intelligent Visualization System now only contains very limited knowledge of visualization techniques that can be used to transform data into a virtual world. In addition, the basic module set within the BigRiver resource now only consists of about ten functions. Consequently, Tecate can only construct very crude visualizations in its present form.

There is a long list of new features and enhancements that will hopefully be included in succeeding generations of Tecate. The management of objects needs to be reworked so that thousands of objects can be efficiently handled simultaneously. Although Tecate now builds virtual worlds, virtual reality gadgetry has yet to be included within the system. AVL needs new features, and it needs to be streamlined. Tecate can also greatly benefit from a toolkit of 3-D widgets that can be used to interact with objects within virtual worlds. Finally, the Doré graphics system that is used within Tecate needs to be replaced with a more mainstream system like OpenGL.

In practice, Tecate has proven to be an exciting system to use, and it is an excellent foundation from which to pursue further research and development in the exploration of general data spaces. Tecate advances the state-of-the-art by demonstrating a way to both graphically browse for data and then interactively visualize data-sets that are selected. Tecate accomplishes these tasks by exploiting the flexibility of an interpreted, object-oriented language that describes virtual worlds.

Acknowledgements

This work was supported by the Digital Equipment Corporation, the University of California, and the San Diego Supercomputer Center as part of the Sequoia 2000 Project. Special thanks to Mike Kelley, Jonathan Shade, and Colin Sharp for their help in constructing the Tecate prototype.

References

[1] Advanced Visual Systems, Inc. *AVS User's Guide*, May 1992.

[2] Zahid Ahmed et al. An intelligent visualization system for earth science data analysis. *Journal of Visual Languages and Computing*, December 1994.

[3] D. M. Butler and M. H. Pendley. A visualization model based on the mathematics of fiber bundles. *Computers in Physics*, pages 45–51, Sep/Oct 1989.

[4] David M. Butler and Steve Bryson. Vector-bundle classes form powerful tool for scientific visualization. *Computers in Physics*, 6(6):576–584, Nov/Dec 1992.

[5] Stephen M. Casner. A task-analytic approach to the automated design of graphic presentations. *ACM Transactions on Graphics*, 10(2):111–151, April 1991.

[6] Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada. *Object Modeling Language (OML) Programmer's Manual*, 1992.

[7] R. B. Haber, B. Lucas, and N. Collins. A data model for scientific visualization with provisions for regular and irregular grids. In *Proceedings Visualization '91 Conference*, 1991.

[8] IBM, Corp. *IBM Visualization Data Explorer: User's Guide*, 1992.

[9] Eve Ignatius and Hikmet Senay. Visualization assistant. In *Proceedings IEEE Visualization Workshop on Intelligent Visualization Systems*, October 1993.

[10] Illustra Information Technologies, Inc. *Using Illustra*, June 1994.

[11] The Khoros Group, Dept. of Electrical and Computer Engineering, University of New Mexico. *Khoros User's Manual*, 1992.

[12] Peter Kochevar et al. An intelligent assistant for creating data flow visualization networks. In *Proceedings of the AVS '94 Conference*, 1994.

[13] Kubota Graphics Inc. *Doré Programmer's Guide*, 1994.

[14] J. D. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 5(2):110–141, 1986.

[15] John Ousterhout. Tcl: An embeddable command language. In *Proceedings of the 1990 Winter USENIX Conference*, 1990.

[16] John Ousterhout. An X11 toolkit based on the Tcl language. In *Proceedings of the 1991 Winter USENIX Conference*, 1991.

[17] Anthony Parisi and Mark Pesce. Virtual reality markup language (VRML). Available via Mosaic, http://www.wired.com/vrml/, June 1994.

[18] Randy Pausch et al. Alice: A rapid prototyping system for virtual reality. In *Course Notes #2: Developing Advanced Virtual Reality Applications*. ACM Siggraph '94 Conference, 1994.

[19] David Raggett. Extending WWW to support platform independent virtual reality. Available via Mosaic, http://www.wired.com/vrml/, June 1994.

[20] Lori Resnick et al. *CLASSIC Description and Reference Manual for the Common LISP Implementation*. AT&T Bell Laboratories, Murray Hill, New Jersey, 1993.

[21] Hikmet Senay and Eve Ignatius. VISTA: A knowledge based system for scientific data visualization. Technical Report GWU-IIST-92-10, George Washington University, March 1992.

[22] Guy L. Steele Jr. *Common LISP: The Language, 2nd Edition*. Digital Press, 1990.

[23] M. Stonebraker and G. Kemnitz. The POSTGRES next generation database management system. *Communications of the ACM*, pages 78–92, October 1991.

[24] Paul Strauss and Rikk Carey. An object-oriented 3D graphics toolkit. In *Proceedings of Siggraph '92 (Chicago, Illinois, July 26-31, 1992)*, New York, 1992. ACM Siggraph.

[25] David Ungar and Randall Smith. Self: The power of simplicity. *Sigplan Notices*, 22(12):227–241, December 1987.

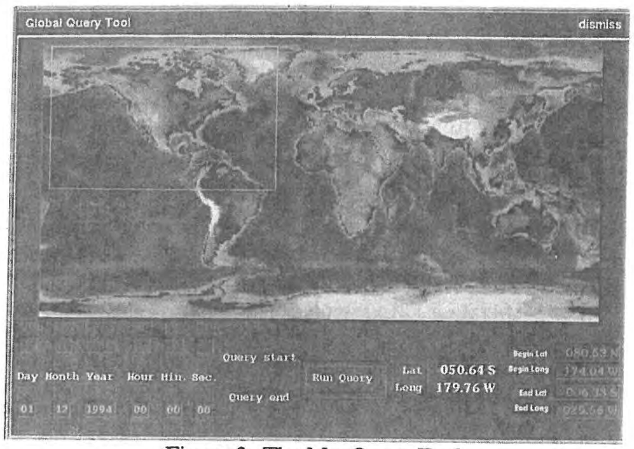


Figure 3: The MapQuery Tool.

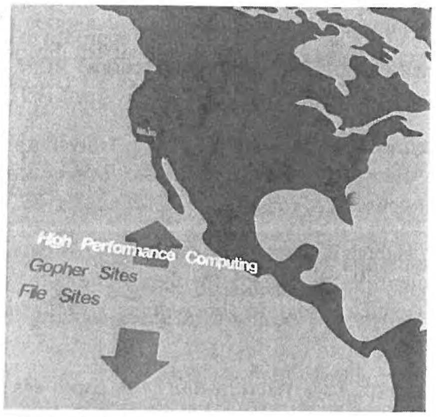


Figure 6: WWW sites depicted as 3-D icons on a world map.

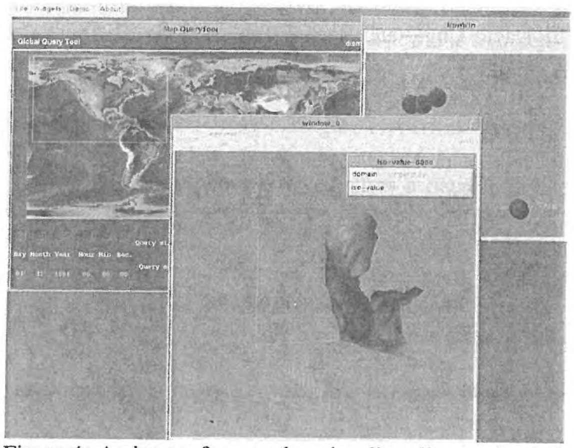


Figure 4: An iso-surface used to visualize climate simulation data.

Figure 7: Viewing a Web site's home page.

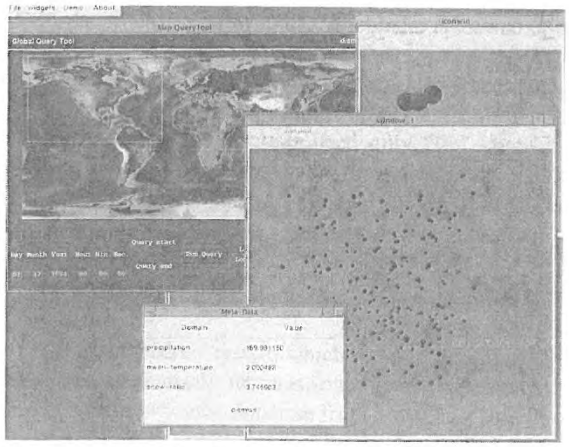


Figure 5: A visualization of data from hydrological measurement stations.

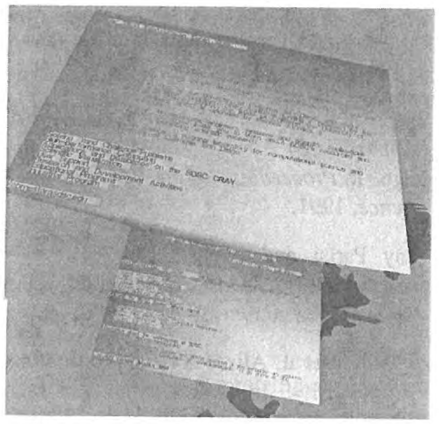


Figure 8: The result of following a hyperlink from a hypertext document.

An Environment for Real-time Urban Simulation

William Jepson[1], Robin Liggett[2], Scott Friedman[3]
Department of Architecture, UCLA

ABSTRACT

Drawing from technologies developed for military flight simulation and virtual reality, a system for efficiently modeling and simulating urban environments has been implemented at UCLA. This system combines relatively simple 3-dimensional models (from a traditional CAD standpoint) with aerial photographs and street level video to create a realistic (down to plants, street signs and the graffiti on the walls) model of an urban neighborhood which can then be used for interactive fly and walk-through demonstrations.

The Urban Simulator project is more than just the simulation software. It is a methodology which integrates existing systems such as CAD and GIS with visual simulation to facilitate the modeling, display, and evaluation of alternative proposed environments. It can be used to visualize neighborhoods as they currently exist and how they might appear after built intervention occurs. Or, the system can be used to simulate entirely new development.

SIMULATION INTERFACE

Work at UCLA has focused on creating a user interface for viewing and interacting with a 3-dimensional environment which has been designed specifically to meet the needs of the planning and design professions. This interface and simulation software runs on a Silicon Graphics Onyx workstation with Reality Engine graphics hardware allowing extensive use of real-time texture mapping. The simulation software was developed using Silicon Graphic's IRIS Performer application development environment. Using a Motif/X-Windows standard, the UCLA interface to the simulation includes a well-defined set of functions that most users will find sufficient for loading and viewing models without additional programming effort. However, the interface has been designed in such a way that it is easy to custom tailor the simulation to a particular application.

The simulation interface includes fly/drive controls so that the user can travel anywhere and view any part of the model from a digitally accurate perspective. Dynamic objects (such as moving vehicles or pedestrians) can be included in the scene. The user has the option of attaching to any of these objects as they are moving through the model allowing specific paths to be followed and evaluated.

A separate mode of interaction allows three-dimensional selection ("picking") of objects in the scene. Once selected, an object can be removed from the scene (simulating, for example, the removal of a building from a lot), or highlighted in the scene (as if a colored spot light were focused on it). More importantly from a design perspective, alternative models can be substituted for the object. This latter function is useful for displaying design options for a particular site, or showing a sequential set of options (for example, models that show the development of a site over time or the growth over time of newly planted foliage).

Another key option in "pick" mode allows an associated data base to be queried for object attributes. This option provides the capability for dynamic query and display of information from an existing data base (for example, a Geographic Information System (GIS)) in a real time 3-dimensional format.

THE MODELING PROCESS

The simulation component of the system does not include capabilities for building the basic model geometry, rather it is used only for interactive viewing, manipulation and querying of the 3-dimensional model and associated databases. Software System's MultiGen is the primary 3-dimensional modeler used in the modeling process. MultiGen, traditionally used for military applications, has the ability to quickly model an urban scene by the application of photographic images ("textures") to highly simplified geometric models of objects such as buildings, trees, streets, etc.

The model creation process begins with plan view aerial photographs which are a quick, easy and accurate way to obtain up-to-date information on street widths, building foot prints, foliage, etc. These photos are scanned into the computer and appropriately scaled and rotated to fit into the California State Plane grid coordinate system that is used for all Los Angeles projects. Using these photos as a base, streets and blocks are quickly identified, outlined and inserted into the database using MultiGen. In many cases detailed street, parcel and building plan data already exists in DXF format (for many areas of the City of Los Angeles, for example), which can significantly shorten this phase of the modeling. If a DXF file is available, the aerial photograph can be calibrated to the DXF plan.

Generally modeling the 3-dimensional geometry of the existing buildings requires only simple rectilinear extrusions to the building

[1]William Jepson, (310)825-5815, bill@ucla.edu
[2]Robin Liggett, (310) 825-6294, robin@gsaup.ucla.edu
[3]Scott Friedman, (310)825-6294, scott@gsaup.ucla.edu
Department of Architecture, School of Arts and Architecture
1317 Perloff Hall, UCLA, Los Angeles, CA 90024-1467

1995 Symposium on Interactive 3D Graphics, Monterey CA USA
© 1995 ACM 0-89791-736-7/95/0004...$3.50

heights, although more detailed model construction is possible. The photorealism of the model comes from the application of photographic textures to the simple 3-D forms. Textures are captured by video taping each building facade in the study area. This video information is fed directly into the computer, perspective and color corrected, and saved in a texture database.

The physical data base is organized spatially by partitioning the region into tiles. We find it is most efficient to use street intersections as the basic organizing units. Intersections give a convenient way to reference the locations in the database (for example, the intersection of Wilshire and Vermont). Thus one spatial tile includes an intersection and about a quarter of each of the four adjacent blocks. The blocks can be divided along parcel lines so that some data integrity is maintained, however, there is no direct link in the data structure between the separate block segments. While this partitioning scheme facilitates the modeling process (it is more convenient to split the blocks into parts than to divide the streets down the center lines), it causes additional complexity when linking the physical model to a GIS database where normally a block would be stored as a coherent entity.

APPLICATIONS

Currently there are several real-world projects underway that make use of this simulation technology and help focus the development effort. These projects, which range from architectural context studies to medium and large scale urban design and planning, emphasize different aspects of the system for design/decision making.

One project is using this visual simulation technology to provide a local, neighborhood level community-based planning and communications tool to aid in redevelopment of the Pico Union area of Los Angeles. This area was badly damaged by the 1992 riots and the 1994 Northridge earthquake. For the initial project a base area of about eighteen square blocks of the community has been modeled. Using only simple interactive features of the user interface, planners have been able to experiment with demolishing a number of existing buildings and reclaiming street areas to bring park and green space to the neighborhood. Planners in the community are particularly interested in linking the virtual reality model to a GIS data base for displaying information on parcel level characteristics such as building ownership, type and the willingness of the owner to work with the community in the redevelopment process. Accessing such data in real-time as part of the visual simulation will allow immediate identification of parcels which would be most suitable for change.

In another project, which was carried out for the Los Angeles Metropolitan Transportation Authority, an area around the Wilshire & Vermont subway transit station was modeled. This model of the existing neighborhood was then used to provide the context for evaluating development alternatives for the site above the station. One feature of the Urban Simulator interface which was particularly useful for this project is the ability to display development of a site over time. The proposed MTA development is intended to be built in five stages. By simply moving a slider on the user interface, the phases of development are added to the model. While the model has proved valuable to the MTA staff for generating conceptual plans, ultimately it is expected to be used to provide a context for interested community groups to experience alternative proposals for development around the station site.

While these projects look at redevelopment occurring in existing areas, another project focuses on conceptual modeling of a new

Pico Union Neighborhood Model

mixed-use, master planned community located near the Pacific coast approximately two miles north of the Los Angeles International Airport. To date the proposed Master Plan site layout with the surrounding natural bluff features and major existing buildings in the area have been modeled. This site model currently consists of the system of streets, lot boundaries and open spaces and will ultimately include the proposed landscaping. Once the infrastructure model is complete, articulated models of typical buildings will be placed on the lots based on a detailed set of zoning criteria from the Master Plan. Alternative forms which meet the criteria can be explored and textured with images taken from existing developments which have a similar feeling to those proposed. Rendered images of actual designs can also be tested when the project moves into this phase.

CONCLUSION

An earlier "proof of concept" prototype model focusing on a riot-torn portion of South Central Los Angeles recently won the top award in the education and academia category of the 1994 Computerworld Smithsonian Award Program. Actual experience using the system on real projects continues to validate this development effort. The system has proven to be an extremely useful tool for exploring potential design solutions. It is possible to evaluate alternatives rapidly and in more detail than through more traditional analysis. Results of the planning/design process are illustrated visually, allowing the client or community to view a proposed environment in a realistic fashion and become informed participants in the decision process.

To facilitate community participation in the simulation process, the UCLA Urban Simulation Team recently received a CalREN (California Research and Education Network) grant for a 155 megabit/second ATM connection to a wide area GTE and Pac Bell ATM net. Additional commitments from Bay Networks for ATM equipment and AT&T for their enhanced Multimedia Interface (EMMI) will enable the real-time transmission of the keyboard and mouse information in one direction and the video generated by the simulation in the other. This will allow the display of the simulation (in real-time) at any other CalREN or similarly connected site (high school, library, community center, etc.). It is not difficult to envision a time when the much prophesied 500 station interactive cable networks will support community member's connection to local city halls (at night when the machines are normally idle) where they will use this technology to interactively evaluate and comment on plans for the community.

Mathenautics: Using VR to Visit 3-D Manifolds

Randy Hudson[1]
University of Illinois
Charlie Gunn[2]
Technical University, Berlin
George K. Francis[3] Daniel J. Sandin[4] Thomas A. DeFanti[4]
University of Illinois

Abstract

In most virtual reality applications, 3-d space is a passive, ambient continuum in which the objects of study are placed. When the 3-d space itself is the object of study, as with mathematical manifolds, VR is especially important as a visualization medium. We describe the visualization of such spaces in the CAVE virtual environment.

1 Introduction

Computer graphics has become instrumental in new discoveries in several domains of mathematics. For example, in the study of minimal surfaces computer graphics was indispensable in advancing several proofs and conjectures regarding a new minimal surface [1]. Depending on the emphasis, these new techniques are called either visual or experimental mathematics. Several centers have been founded to further research in this direction, such as the Geometry Center at the University of Minnesota and the SFB-288 lab at Technical University Berlin.

One area of visual mathematics where computer graphics can be helpful is the classification of 3-dimensional manifolds (*3-manifolds*). According to a conjecture of William Thurston (we omit some technical conditions), any 3-manifold can be classified by modeling it on one of eight model geometries (see [4] and related literature). In this paper, we will be concerned with three of these model geometries (the classical cases which exist in every dimension): euclidean (E^3), spherical (S^3) and hyperbolic (H^3). We will use the term manifolds loosely, to include the related spaces known as *orbifolds*, which may contain singular points. For a fuller discussion of the mathematical background and its computer-graphical implementation followed here, see [3]. The next section contains a very abbreviated version of this article.

[1]Academic Information Technologies, Culver Hall, 1025 E. 57th, University of Chicago, rhudson@eecs.uic.edu

[2]Special Research Project 288, "Differential Geometry and Quantum Physics", Math 8-5, Strasse des 17 Juni 136, Technical University, 10623 Berlin, Germany, gunn@sfb288.math.tu-berlin.de

[3]National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, gfrancis@math.uiuc.edu

[4]University of Illinois at Chicago

2 Previous work on manifold visualization

The visualization of 3-manifolds is not as straightforward as that of 2-manifolds. The 2-torus can be visualized directly as a familiar doughnut-shaped surface by embedding it in a higher (third) dimension. This form of representation can be called the *outsider's* view. However, living within 3-dimensional space, we have no fourth, directly-visible dimension in which to embed or immerse 3-manifolds.

An alternative method for visualizing manifolds which solves this problem is the *insider's* view. This is the view we would see if we were to live inside the manifold. It is constructed as a tessellation of the model geometry by non-overlapping copies of a single tile, or fundamental domain. One copy of this tile represents the underlying manifold; the other copies represent the different ways that light can travel in the manifold to reach the observer's eye. In the case of our two-dimensional torus, the corresponding tessellation covers the euclidean plane with copies of a parallelogram. (To get the outsider's view we take one copy of this parallelogram and roll it up in 3 dimensions to make the torus.) The fundamental domain is replicated via the application of (*a discrete group of*) isometries, in this case, translations in two independent directions. Likewise, the 3-torus is visualized from the inside by applying translations in three independent directions of E^3. (The discrete groups we study can contain other kinds of isometries: glide-reflections, screw motions, rotations and reflections.) Figure 1 shows the insider's view of the 3-torus where the fundamental domain – a cube-like polytope – has been shrunk to improve visibility.

3 Introduction to GeomCAVE

The immersive nature of the insider's view is what makes a virtual environment a better visualization medium for 3-manifolds than the graphics workstation. The Geometry Center has developed an interactive viewer *geomview* based on the graphics library OOGL, capable of visualizing the three model geometries considered here on a workstation. The second author, while at the Geometry Center, developed a related tool, *maniview*, for visualizing the insider's

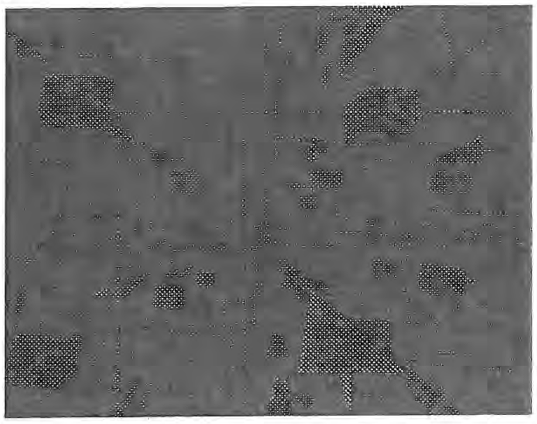


Figure 1: Tessellation of euclidean 3-space by 3-torus.

view of 3-manifolds modeled on these geometries[5]. We have adapted these tools to the CAVE virtual environment developed at the Electronic Visualization Laboratory at the University of Illinois of Chicago [2]. We call this hybrid tool GeomCAVE; using it, the observer can for the first time actually travel within these 3-manifolds and see them from the inside. (After GeomCAVE was developed, more modest alternatives for traveling through hyperbolic and spherical space using the standard CAVE libraries were developed by Ulrike Axen, Glenn Chappell, Chris Hartman, Joanna Mason, Paul McCreary and the fifth author for the Post-Euclidean Walkabout at SIGGRAPH '94. Stuart Levy and Tamara Munzner, from the Geometry Center, have recently expanded this code to read a subset of the OOGL formats directly into the CAVE.)

Modules for the CAVE generally take the form of a single draw routine consisting of GL function calls, which is called regularly from the main CAVE program. Since OOGL maintains its own graphics context, including transform and appearance stacks, which objects rely on when rendering themselves, GeomCAVE had to be slightly more sophisticated. We had to make some CAVE states available to the OOGL context (e.g., which wall is currently being drawn); and we also channeled the navigation data (walking and flying data) through OOGL routines to generate non-euclidean isometries.

The user of GeomCAVE is provided with a menu of icons representing different manifolds; choosing one brings him into the tessellation for that space. The fundamental tile is by default provided by a Dirichlet domain for the underlying group; it is represented once at full scale in wire frame and once at reduced scale as a shaded solid, with corresponding faces a unique color. The reduction in scale was necessary to provide visibility of the whole tessellation, while the coloring provides important information of the structure of the manifold. The observer is represented in the scene by a small dart-shaped object which is also tessellated (figure 2). It points in the direction of the observer's gaze, and its motion and orientation with respect to the fixed geometry of the tessellation provides further structural information. Some of the example spaces contain singular axes; approach to these is signaled by the convergence of multiple copies of the dart to a single point.

The observer can navigate through the space either by physically walking within the CAVE or by flying in the direction of a hand-held wand. He can reset himself to the origin if he gets lost, or can return to the icons to select another space to visit.

[5]geomview and maniview are available via ftp from geom.umn.edu in pub/software

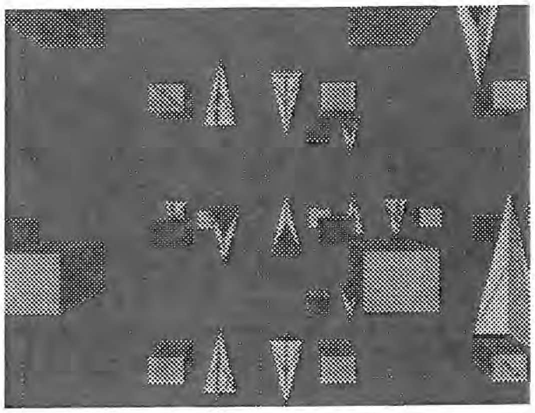


Figure 2: Tessellation of euclidean 3-space by group containing rotation axes.

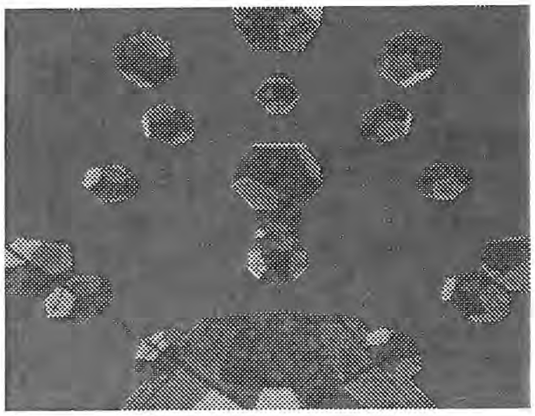


Figure 3: Dodecahedral tessellation of spherical 3-space.

3.1 The example manifolds

Two of the non-euclidean examples are tessellated by regular dodecahedra, a construction impossible in E^3. Because the sum of a triangle's angles in H^3 is less than 180°, a regular dodecahedron with right dihedral angles is possible. H^3 can be tiled with these right, regular dodecahedra in a variety of ways. In GeomCAVE, the observer can verify this experimentally by flying or walking to the common corner of eight dodecahedra and examining these right angles directly. See [3] for an illustration of an H^3 tessellation.

In the same way, it is possible to have a regular dodecahedron in S^3 which has 120-degree dihedral angles (figure 3).

The euclidean 3-manifolds featured in GeomCAVE are both based on the tessellation of euclidean space by a cube, but the discrete groups which perform the tessellation are different. See Figures 1 and 2. It is also possible to explicitly provide geometry to be tessellated instead of using the Dirichlet domain (figure 4).

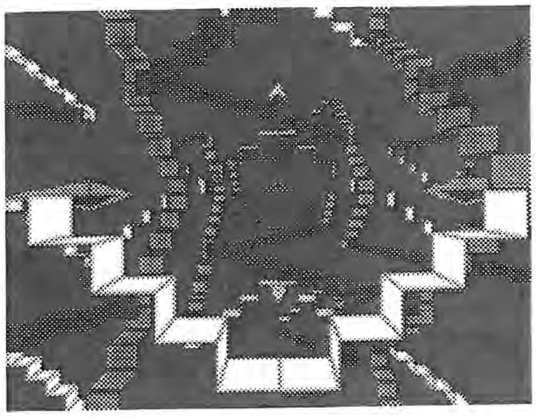


Figure 4: Alternate tessellation of 3-space by the group in Figure 2.

4 GeomCAVE implementation challenges

We encountered several challenges in the implementation of GeomCAVE:

- Conflicting viewing paradigms,
- Non-euclidean navigation in GeomCAVE,
- Manifold navigation in GeomCAVE,
- Stereo, and
- Efficiency measures.

4.1 Conflicting viewing paradigms

OOGL's on-axis perspective projection is applicable in a head-mounted display VR system, where the view planes move with the viewer's eyes, but not in the CAVE, where off-axis projection must be used because the view planes are stationary. This problem applied equally to all geometries. The solution required us to replace the OOGL camera object with an alternative means of transforming from world coordinates to screen coordinates based upon off-axis projection from the observer's position within the CAVE.

4.2 Non-euclidean navigation in GeomCAVE

Euclidean 3-space can be trivially modeled in the euclidean 3-space of the CAVE interior. We discuss the more difficult challenge of mapping hyperbolic geometry into the CAVE; similar remarks apply to the spherical case.

The model of H^3 we use in GeomCAVE the *Beltrami-Klein* (or *projective*) model [3]. In this model, hyperbolic space is modeled as the interior of the unit ball in E^3. Measurements of distance and angle are computed using a different metric; the result is that the unit sphere lies an infinite (hyperbolic) distance from any point within the ball. Hence the unit sphere is called the *sphere at infinity* in this model. Figure 5 shows two views of a tessellation of the two-dimensional hyperbolic plane in this model.

These facts have two consequences for navigation within GeomCAVE. The first follows from the requirement that the observer should not be allowed to leave hyperbolic space by walking (see end of this section). If we want to prevent the navigator from leaving H^3 then it is clear that we must map the CAVE's physical coordinates into the projective model so that it lies entirely within the unit ball. We settled on a scaling value of 0.1, yielding typical corner coordinates of .5, .5, .5, which keeps the CAVE well within the unit sphere. At this value, the dihedral angles between the cave walls, measured hyperbolically, are around 70°. Shrinking the cave yields angles approaching the right angle of Euclidean measurement.

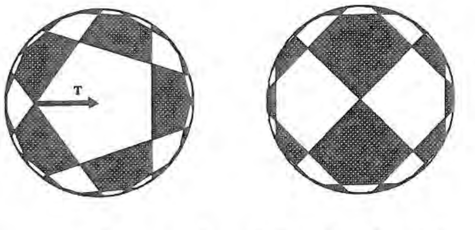


Figure 5: Tessellation of hyperbolic plane by alternately-colored, regular, right-angled pentagons, in the Klein model. Right figure represents a hyperbolic translation of the left by the vector T.

The second consequence affects how navigation is conceptualized. The most natural way to think of flying or walking is that the observer moves through the scene. However, if we follow this model when we implement hyperbolic movement, the result is incorrect. The mistake occurs in the standard construction of the off-axis perspective transformation, which typically contains an implicit euclidean translation to move the observer to the origin. This euclidean translation naturally results in an incorrect image. The solution is to hyperbolically translate the scene past the observer, rather than vice-versa.

In contrast to H^3, which is modeled on the interior of the unit ball, the projective model of S^3 contains all the available points. There is an implicit restriction on the size of the CAVE in model coordinates, since the intrinsic metric of S^3 is finite. The CAVE can not be made larger than a certain size; beyond that scale the size of the cave begins to shrink, just as a circle on the 2-sphere attains a maximum size at the equator and beyond that point gets smaller.

We made the decision to represent H^3 as an inhabitant would experience it, which prohibited the navigator from traveling outside the space. We would like to explore the possibility of euclidean exploration of H^3, so that mathematicians can see how this model "sits" within ordinary space (see Section 6).

4.3 Manifold navigation in GeomCAVE

Though not new with GeomCAVE ([3]), navigation in manifolds is worth mentioning here. One of the challenges unique to manifold exploration involves "staying centered" in the tessellation. Since E^3 and H^3 are infinite (as opposed to S^3, which is a finite space), a complete tessellation would be of infinite extent. Since we can only create a finite tessellation, the possibility exists that the navigator might fly beyond the computed tessellation. The solution adopted here is to "cage" the navigator within the central Dirichlet domain of the group. That is, if in the motion of walking or flying, a wall of the central Dirichlet domain passes by the observer (fixed at the origin), then the observer is moved to an equivalent point lying within the central fundamental domain. That is, the cumulative navigation isometry is multiplied by the group element associated with the crossed face. The resulting isometry maintains the origin within the central Dirichlet domain. With respect to the manifold, this new transform is equivalent to the original isometry, since multiplication by group elements leave the manifold invariant. However, this multiplication may be detectable in our finite implementation: some copies on the edge of the tessellation may appear or disappear. In the ideal implementation (requiring more computer power) these copies are barely visible, either being too small or too foggy.

The alternative, to translate the tessellation to follow the observer, quickly leads in the hyperbolic case to severe numerical problems in the action in the group elements. The result is that the fourth, "homogeneous" coordinate of the transformed vertices grows exponentially large and the dehomogenization operation loses precision. This is avoided by the method outlined above.

4.4 Stereo

Modeling stereo vision presented challenges in the non-euclidean case. We first describe the more familiar solution available in the euclidean setting. The observer and the CAVE have a fixed physical reality which should be mirrored in the models we apply to them. That is, the model coordinates for the navigator are the same as the model coordinates of the CAVE. In particular, the interocular separation of the observer stays at a fixed ratio to the CAVE size. We found empirically that an interocular distance of about 1/100 that of the diagonal of the CAVE is small enough to assure fusion. This translates to a distance of about 2 inches, roughly corresponding to human anatomy. In euclidean space, making the CAVE larger is equivalent to shrinking the scene while keeping the CAVE a constant size. However, in non-euclidean settings, this equivalence no longer holds! In these spaces, there is no change of size without also changing shape. Consequently, it is the CAVE and observer that changes size (and shape!) while the scene remains the same. Of course there is no guarantee of fusion; it may become difficult if the observer in H^3 becomes too large while standing near the fixed geometry; but the danger is no different from the physically observed difficulty of fusing stereo when you move your hand closer to your eyes in everyday life.

The pair of images for the stereo effect is produced by rendering each eye separately as described above by hyperbolically translating the scene to locate the given eye at the origin.

4.5 Efficiency measures

To maintain the frame rate required in VR, we needed to disable the software lighting and shading for non-euclidean scenes (OOGL does lighting in software because of the different metrics of the non-euclidean geometries). We kept the model of the tessellation simple – a wireframe, with simple, solid tiles inside. The discrete group software in OOGL automatically culled the copies of the tessellation which lay outside the viewing frustum of a given wall of the CAVE. Also, we kept the number of layers of the tessellation great enough to produce a sense of depth, but small enough to maintain an adequate frame rate.

5 Evaluation

We have combined the discrete group capabilities of OOGL with VR, the only visualization paradigm for an immersive, direct experience of mathematical spaces, to extend the power of interactive 3-d visualization of such spaces. Access to 3-manifolds via a virtual environment is a significant addition to the tools available for mathematical research and education. For example, as pointed out in section 3, GeomCAVE allows direct observation of interesting properties of non-euclidean spaces, such as the right angles of dodecahedra in hyperbolic space. GeomCAVE immediately makes features of OOGL available in VR, such as a collection of geometric models and discrete group operations. Thus, a mathematician who has built a manifold for viewing in maniview would be able to also explore it in GeomCAVE.

6 Further work

- Implement mixed mode navigation in H^3 (see conclusion of Section 4.2).
- Add more features of maniview:
  - Control over the size and shape of the Dirichlet domain.
  - Control over the depth of the tessellation.
  - As hardware improves, re-activate the software shading and fog effects.
- More sophisticated tools for mathematicians:
  - Connections with existing manifold software (such as *snappea* ([5]).
  - Finer interactive control of the discrete group: selecting subgroups, use of color, deformation of the group.
  - Simulation of dynamical systems in non-euclidean spaces.
  - Extend the coverage to the other five Thurston geometries.
- Experiment with audio tessellation along with the geometric data. The resulting echo patterns could distinguish differently-shaped manifolds.

7 Acknowledgements

We would like to extend special thanks to Stuart Levy of the Geometry Center for his help. Thanks are also due to Mark Phillips and Tamara Munzner, also of the Geometry Center, as well as Louis Kauffman, of the University of Illinois at Chicago.

REFERENCES

[1] Callahan, M.J., Hoffman, D. and Hoffman, J.T. Computer Graphics Tools for the Study of Minimal Surfaces. *Communications of the Association for Computing Machinery 31*, 6 (1988), 648-661.

[2] Cruz-Neira, Carolina, Sandin, Daniel J., DeFanti, Thomas A., Kenyon, Robert V. and Hart, John C. The CAVE: Audio Visual Experience Automatic Virtual Environment. *Communications of the Association for Computing Machinery 35*, 6 (June, 1992), 65-72.

[3] Gunn, Charlie. Discrete Groups and Visualization of Three Dimensional Manifolds. *Computer Graphics 27* (July, 1993), 255-262. Proceedings of SIGGRAPH 1993.

[4] Thurston, William. Three Dimensional Manifolds, Kleinian Groups and Hyperbolic Geometry. *BAMS 19* (1982), 417-431.

[5] Weeks, Jeff. snappea — a MacIntosh application for computing 3-manifolds". (available from ftp@geom.umn.edu).

Tracking a Turbulent Spot in an Immersive Environment

*David C. Banks, Institute for Computer Applications in Science and Engineering

‡Michael Kelley, Information Sciences Institute

ABSTRACT

We describe an interactive, immersive 3D system called *Tracktur*, which allows a viewer to track the development of a turbulent flow. *Tracktur* displays time-varying vortex structures extracted from a numerical flow simulation. The user navigates the space and probes the data within a windy 3D landscape. In order to sustain a constant frame rate, we enforce a fixed polygon budget on the geometry. In actual use by a fluid dynamicist, *Tracktur* has yielded new insights into the transition to turbulence of a laminar flow.

1 Introduction

Simulating the evolution of a turbulent spot has consumed thousands of CPU hours (on a Cray 2, Cray YMP, and YMP C-90 over the course of 2.5 calendar years) [1]. We wish to animate 230 time steps produced by the simulation, which are archived as hundreds of gigabytes of data. How does one visualize this large amount of time-varying data at interactive speeds?

A new technique for locating vortices in an unsteady flow [2] compresses the volumetric flow-data by a factor of more than a thousand. This amount of compression seemed to promise interactive visualization of a massive time-varying dataset. We therefore developed a visualization system, *Tracktur*, that uses the compressed vortex representation to help track the development of a turbulent flow [3]. *Tracktur* uses a graphics workstation, 3D tracking, and a stereoscopic display to create a virtual 3D environment populated by time-varying vortex tubes.

2 The Interactive Environment

Our target user is the theoretical flow physicist who produced the time-varying dataset. From his perspective, the significant features of the simulation include the flat plate, the fluid flowing over it, the vortex structures, and the units of the computational domain (both spatial and temporal). The combination of a plane with a continual flow over it suggested to us a windy landscape.

*ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, Virginia 23681. 804/864-2194 (banks@icase.edu).

‡Information Sciences Institute, 4350 N. Fairfax Drive, Suite 400, Arlington, VA 22203. 703/243-9422 (kelleym@arpa.mil).

One of our early design decisions was to make generous use of texture maps to enrich the virtual world. A grid-texture was an obvious choice for the ground plane, with stenciled textures added to denote streamwise units of the domain. To indicate the free-stream velocity, we animate a cloud-texture on two distant walls. Textures denote the upstream and downstream directions. Surrounded by a textured landscape, a viewer is given persistent reminders of the spatial context he is operating within. The 3D widgets in the environment are also textured to eliminate the cartoon quality that constant-colored polygons convey.

In an actual wind-tunnel experiment, the vortex structures would be only millimeters in size and the free-stream velocity would be about 30 meters per second. The lifetime of the turbulent spot would be less than a second. *Tracktur* displays the 3D animation at more human scales: the geometry is larger and the simulation lasts longer, each by about three orders of magnitude.

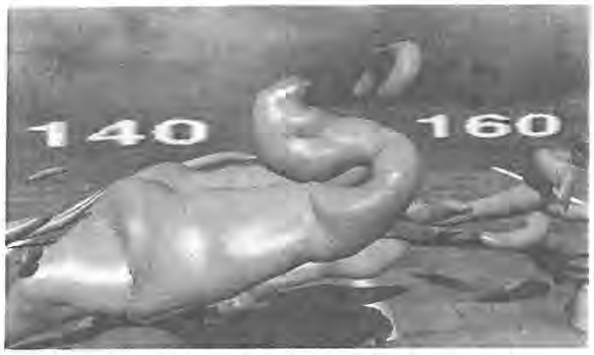
We want to help the scientist comprehend the spatial evolution of a turbulent spot; since the spot convects downstream, we let the viewer be convected along with it to keep it in the field of view. Widgets are convected downstream with the viewer to remain within reach. A time-slider advances to mirror the current time step in the animation; alternatively, the viewer can set the current time step by adjusting the slider. Shadows on the ground plane provide a depth cue at only a small penalty in performance [4]. The viewer can select surface, wire-frame, or fat-line representations of the geometry. The fat-line segments (through the core of the vortices) are given widths to match the thickness of the tube and are illuminated as one-dimensional fibers [5] in order to convey shape from shading.

We also want to permit routine measurements of flow quantities. The viewer is given a data probe – a ray emanating from the pointing device in the virtual environment. *Tracktur* locates the nearest point on a vortex core to the probe ray, then displays attributes (such as spatial position of the point) in a pop-up panel.

3 3D Toolkits

Tracktur is constructed from several component libraries, including public-domain toolkits. The Minimal Reality toolkit [6] provides the basis of a through-the-window interface that uses stereoscopic display and 3D tracking for the head and hand. The CAVE version of the application [7] uses code developed by the Electronic Visualization Laboratory [8].

We developed a custom toolkit to implement 3D menus (using Hershey fonts), buttons, and sliders. We also developed a calibration tool for the 3D trackers to determine the proper matrix transforms. The user interactively aligns coordinate axes (displayed on the screen) to establish the correct rotation matrix. The various transformations are written to a file and need not be recomputed unless the equipment is moved.



A backward-tilted S-shaped vortex head that develops in the late stages of transition from a laminar flow to a turbulent spot.

4 The Fixed Polygon Budget

A difficult aspect of developing an interactive system is preserving a fixed frame rate. Our scene-updates are typically dominated by the time spent drawing the vortex tubes, so we budget a fixed number of polygons with which to model them. The turbulent spot increases in geometric complexity as the simulation progresses: a single vortex tube at time 28 develops into about 150 tubes at time 221. An SGI Onyx with RealityEngine2 graphics sustains about 15 frames per second with a fixed count of 9000 polygons.

In the early stages of the simulation, the polygon budget allows a finer resolution than we have computed. We therefore resample the vortex skeleton at a higher spatial resolution in order to exhaust the supply of polygons. But in the late stages of the simulation it is imperative to dole out the polygons in a miserly fashion. The vortex skeletons are down-sampled according to a set of heuristics designed to preserve significant geometric features. The resampling works as a filter on the original skeletal representation of the vortex core. The first sample-point is always retained. After a point is retained, subsequent points along the skeleton are rejected unless any of the following hold:

- the arclength exceeds a threshold;
- the integrated curvature exceeds a threshold;
- the radius of the cross-section changes quickly.

Sometimes a vortex skeleton enters a small spiral from which it never exits. To guard against wasted samples, we reject points on the skeleton where the ratio of the skeleton's radius to its radius of curvature exceeds a threshold (we use the constant 0.7). These heuristics maintain a reasonable amount of geometric detail at the late stages of the simulation.

5 What Has Been Learned

The scientist who generated the dataset (Dr. Bart Singer) agreed to use the system to study how a turbulent spot develops. He has learned two new things about the evolution of the turbulent spot. In order to place them in their context, we give a brief descriptive summary of the spot's development.

First, Singer discovered a backwards-tilted S-shaped vortex head in the late stages of transition (see figure). The vortex is similar in shape to a structure seen in experimental data for a similar flow. Singer had not observed this feature in his dataset until he used our system. Evidently, the interactivity permitted him to select the right combination of a particular viewpoint and a particular time step. This could, in principal, have been discovered with the visualization system he was a accustomed to using, but its more limited interactivity made the feature much harder to find.

Secondly, the visualization system gave Singer his first view of the dynamic behavior of "necklace" vortices, which define the outer extent of the turbulent spot. They eventually shred into pieces, curling into horseshoe and hairpin vortices. Without *Tracktur*, Singer had been unable to track the necklace vortices through their entire history. These findings are initial evidence that the system can assist in the research task.

6 Conclusions

Visualization tools can certainly *communicate* research results, but it is not yet clear how well they help *produce* research results. We have created an interactive 3D visualization system, called *Tracktur*, and put it into the hands of the scientist. *Tracktur* provides a textured environment for examining the onset of turbulence. The viewer can navigate through the landscape and interact with a turbulent spot through 3D menus, buttons, sliders, and a data probe. In the hands of a fluid scientist, the system has yielded new insights into the development of a turbulent spot.

Acknowledgments

This work was supported under NASA contract No. NAS1-19480. We thank Bill von Ofenheim and the Data Visualization Lab at NASA Langley Research Center for use of their stereo glasses. We thank Jonathan Shade at the San Diego Supercomputer Center for help in creating transparent texture maps.

Bibliography

[1] Singer, Bart A. and Ron Joslin, "Metamorphosis of a hairpin vortex into a young turbulent spot." *Physics of Fluids A*, Vol. 6, No. 11 (Nov. 94).

[2] Banks, David C. and Bart A. Singer, "Vortex Tubes in Turbulent Flows: Identification, Representation, Reconstruction." *Proceedings of Visualization '94.*

[3] "The Tracktur Home Page," World Wide Web URL http://www.icase.edu/~banks/tracktur/vortex/doc/tracktur.html.

[4] Blinn, Jim, "Me and My (Fake) Shadow." *IEEE Computer Graphics & Applications* (Jim Blinn's Corner), January 1988, pp. 82-86.

[5] Banks, David C., "Illumination in Diverse Codimensions." *Proceedings of SIGGRAPH '94* (Orlando, Florida, July 24-29, 1994). In *Computer Graphics* Proceedings, Annual Conference Series, 1994, ACM SIGGRAPH, New York, pp. 327-334.

[6] "MR Toolkit," World Wide Web URL http://web.cs.ualberta.ca/~graphics/MRToolkit.html.

[7] Banks, David C., "The Onset of Turbulence in a Shear Flow Over a Flat Plate." [Demonstration] SIGGRAPH '94 VROOM Exhibit. In *Visual Proceedings: The Art and Interdisciplinary Programs of SIGGRAPH 94*, Computer Graphics Annual Conference Series, 1994, ACM SIGGRAPH, New York, p. 235. Also in "Fluid Mechanics," World Wide Web URL http://www.ncsa.uiuc.edu/EVL/docs/VROOM/HTML/PROJECTS/23Banks.html.

[8] "CAVE User's Guide," World Wide Web URL http://www.ncsa.uiuc.edu/EVL/docs/html/CAVEGuide.html.

Behavioral Control for Real-Time Simulated Human Agents

John P. Granieri, Welton Becket,
Barry D. Reich, Jonathan Crabtree, Norman I. Badler

Center for Human Modeling and Simulation
University of Pennsylvania
Philadelphia, Pennsylvania 19104-6389
granieri/becket/reich/crabtree/badler@graphics.cis.upenn.edu

Abstract

A system for controlling the behaviors of an interactive human-like agent, and executing them in real-time, is presented. It relies on an underlying model of continuous *behavior*, as well as a discrete scheduling mechanism for changing behavior over time. A multiprocessing framework executes the behaviors and renders the motion of the agents in real-time. Finally we discuss the current state of our implementation and some areas of future work.

1 Introduction

As rich and complex interactive 3D virtual environments become practical for a variety of applications, from engineering design evaluation to hazard simulation, there is a need to represent their inhabitants as purposeful, interactive, human-like agents.

It is not a great leap of the imagination to think of a product designer creating a virtual prototype of a piece of equipment, placing that equipment in a virtual workspace, then populating the workspace with virtual human operators who will perform their assigned tasks (operating or maintaining) on the equipment. The designer will need to instruct and guide the agents in the execution of their tasks, as well as evaluate their performance within his design. He may then change the design based on the agents' interactions with it.

Although this scenario is possible today, using only one or two simulated humans and scripted task animations [3], the techniques employed do not scale well to tens or hundreds of humans. Scripts also limit any ability to have the human agents react to user input as well as each other during the execution of a task simulation. We wish to build a system capable of simulating many agents, performing moderately complex tasks, and able to react to external (either from user-generated or distributed simulation) stimuli and events, which will operate in near real-time. To that end, we have put together a system which has the beginnings of these attributes, and are in the process of investigating the limits of our approach. We describe below our architecture, which employs a variety of known and previously published techniques, combined together in a new way to achieve near real-time behavior on current workstations.

We first describe the machinery employed for behavioral control. This portion includes perceptual, control, and motor components. We then describe the multiprocessing framework built to run the behavioral system in near real-time. We conclude with some internal details of the execution environment. For illustrative purposes, our example scenario is a pedestrian agent, with the ability to locomote, walk down a sidewalk, and cross the street at an intersection while obeying stop lights and pedestrian crossing lights.

2 Behavioral Control

The behavioral controller, previously developed in [4] and [5], is designed to allow the operation of parallel, continuous behaviors each attempting to accomplish some function relevant to the agent and each connecting sensors to effectors. Our behavioral controller is based on both potential-field reactive control from robotics [1, 10] and behavioral simulation from graphics, such as Wilhelms and Skinner's implementation [20] of Braitenberg's *Vehicles* [7]. Our system is structured in order to allow the application of optimization learning [6], however, as one of the primary difficulties with behavioral and reactive techniques is the complexity of assigning weights or arbitration schemes to the various behaviors in order to achieve a desired observed behavior [5, 6].

Behaviors are embedded in a network of *behavioral nodes*, with fixed connectivity by links across which only floating-point messages can travel. On each simulation step the network is updated synchronously and without order dependence by using separate load and emit phases using a simulation technique adapted from [14]. Because there is no order dependence, each node in the network could be on a separate processor, so the network could be easily parallelized.

Each functional behavior is implemented as a subnetwork of behavioral nodes defining a path from the geometry database of the system to calls for changes in the database. Because behaviors are implemented as networks of simpler processing units, the representation is more explicit than in behavioral controllers where entire behaviors are implemented procedurally. Wherever possible, values that could be used to parameterize the behavior nodes are made accessible, making the entire controller accessible to machine learning techniques which can tune components of a behavior that may be too complex for a designer to manage. The entire network comprising the various sub-behaviors acts as the controller for the agent and is referred to here as the *behavior net*.

There are three conceptual categories of behavioral nodes employed by behavioral paths in a behavior net:

perceptual nodes that output more abstract results of perception than what raw sensors would emit. Note that in a simulation that has access to a complete database of the simulated world, the job of the perceptual nodes will be to realistically limit perception, which is perhaps opposite to the function of perception in real robots.

motor nodes that communicate with some form of motor control for the simulated agent. Some motor nodes enact changes directly on the environment. More complex motor behaviors, however, such as the *walk motor node* described below, schedule a motion (a step) that is managed by a separate, asynchronous execution module.

control nodes which map perceptual nodes to motor nodes usually using some form of negative feedback.

This partitioning is similar to Firby's partitioning of continuous behavior into active sensing and behavior control routines [10], except that motor control is considered separate from negative feedback control.

2.1 Perceptual Nodes

The perceptual nodes rely on simulated sensors to perform the perceptual part of a behavior. The sensors access the environment database, evaluate and output the distance and angle to the target or targets. A sampling of different sensors currently used in our system is described below. The sensors differ only in the types of things they are capable of detecting.

Object: An object sensor detects a single object. This detection is global; there are no restrictions such as visibility limitations. As a result, care must be taken when using this sensor: for example, the pedestrian may walk through walls or other objects without the proper avoidances, and apparent realism may be compromised by an attraction to an object which is not visible. It should be noted that an object sensor always senses the object's current location, even if the object moves. Therefore, following or pursuing behaviors are possible.

Location: A location sensor is almost identical to an object sensor. The difference is that the location is a unchangeable point in space which need not correspond to any object.

Proximity: A proximity sensor detects objects of a specific type. This detection is local: the sensor can detect only objects which intersect a sector-shaped region roughly corresponding to the field-of-view of the pedestrian.

Line: A line sensor detects a specific line segment.

Terrain: A terrain sensor, described in [17], senses the navigability of the local terrain. For example, the pedestrian can distinguish undesirable terrain such as street or puddles from terrain easier or more desirable to negotiate such as sidewalk.

Field-of-View: A field-of-view sensor, described in [17], determines whether a human agent is visible to any of a set of agents. The sensor output is proportional to the number of agents' fields-of-view it is in, and inversely proportional to the distances to these agents.

2.2 Control Nodes

Control nodes typically implement some form of negative feedback, generating outputs that will reduce perceived error in input relative to some desired value or limit. This is the center of the reactivity of the behavioral controller, and as suggested in [9], the use of negative feedback will effectively handle noise and uncertainty.

Two control nodes have been implemented as described in [4] and [5], *attract* and *avoid*. These loosely model various forms of *taxis* found in real animals [7, 11] and are analogous to proportional servos from control theory. Their output is in the form of a recommended new velocity in polar coordinates:

Attract An *attract* control node is linked to θ and d values, typically derived from perceptual nodes, and has angular and distance thresholds, t_θ and t_d. The *attract* behavior emits $\Delta\theta$ and Δd values scaled by linear weights that suggest an update that would bring d and θ closer to the threshold values. Given weights k_θ and k_d :

$$\Delta\theta = \begin{cases} 0 & \text{if } -t_\theta \leq \theta \leq t_\theta \\ k_\theta(\theta - t_\theta) & \text{if } \theta > t_\theta \\ k_\theta(\theta + t_\theta) & \text{otherwise} \end{cases}$$

$$\Delta d = \begin{cases} 0 & \text{if } d \leq t_d \\ k_d(d - t_d) & \text{otherwise.} \end{cases}$$

Avoid The *avoid* node is not just the opposite of *attract*. Typically in *attract*, both θ and d should be within the thresholds. With *avoid*, however, the intended behavior is usually to have d outside the threshold distance, using θ only for steering away. The resulting avoid formulation has no angular threshold:

$$\Delta\theta = \begin{cases} 0 & \text{if } d > t_d \\ k_\theta(\pi - \theta) & \text{if } d \leq t_d \text{ and } \theta \geq 0 \\ k_\theta(-\pi - \theta) & \text{otherwise} \end{cases}$$

$$\Delta\theta = \begin{cases} 0 & \text{if } d > t_d \\ k_d(t_d - d) & \text{otherwise.} \end{cases}$$

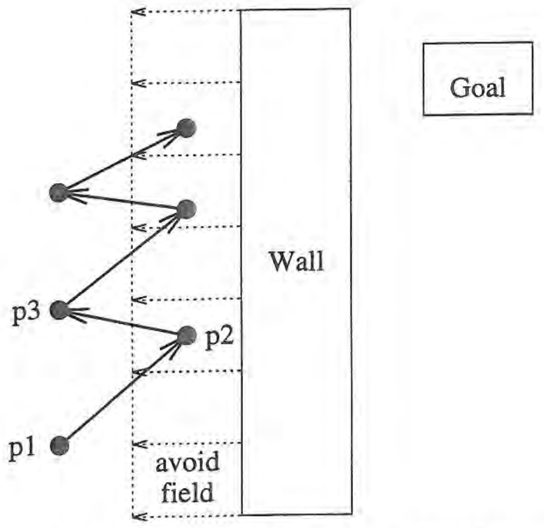


Figure 1: Sawtooth path due to potential field discontinuities

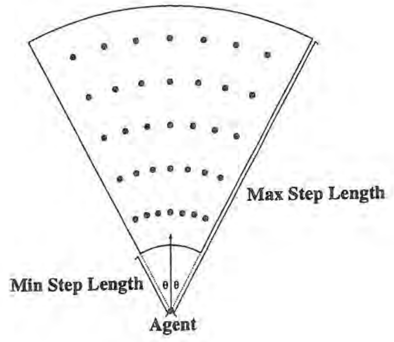


Figure 2: The fan of potential foot locations and orientations

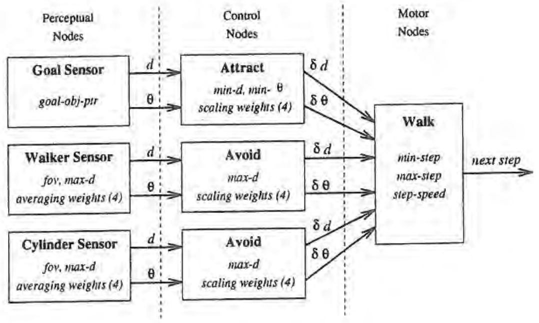


Figure 3: An example behavior net for walking

2.3 Motor Nodes

Motor nodes for controlling non-linked agents are implemented by interpreting the Δd and $\Delta\theta$ values emitted from control behaviors as linear and angular adjustments, where the magnitude of the implied velocity vector gives some notion of the urgency of traveling in that direction. If this velocity vector is attached directly to a figure so that requested velocity is mapped directly to a change in the object's position, the resulting agent appears jet-powered and slides around with infinite damping as in Wilhelms and Skinner's environment [20].

2.3.1 Walking by sampling potential fields

When controlling agents that walk, however, the motor node mapping the velocity vector implied by the outputs of the control behaviors to actual motion in the agent needs to be more sophisticated. In a walking agent the motor node of the behavior net *schedules* a step for an agent by indicating the position and orientation of the next footstep, where this decision about where to step next happens at the end of every step rather than continuously along with motion of the agent. The velocity vector resulting from the blended output of all control nodes could be used to determine the next footstep; however, doing so results in severe instability around threshold boundaries. This occurs because we allow thresholds in our sensor and control nodes and as a result the potential field space is not continuous. Taking a discrete step based on instantaneous information may step across a discontinuity in field space. Consider the situation in Fig. 1 where the agent is attracted to a goal on the opposite side of a wall and avoids the wall up to some threshold distance. If the first step is scheduled at position p_1, the agent will choose to step directly toward the goal and will end up at p_2. The agent is then well within the threshold distance for walls and will step away from the wall and end up at p_3, which is outside the threshold. This process then repeats until the wall is cleared, producing an extremely unrealistic sawtooth path about the true gradient in the potential field.

To eliminate the sawtooth path effect, we sample the value of the potential field implied by the sensor and control nodes in the space in front of the agent and step on the location yielding the minimum sampled 'energy' value. We sample points that would be the agent's new location if the agent were to step on points in a number of arcs within a fan in front of the agent's forward foot. This fan, shown in Fig. 2, represents the geometrically valid foot locations for the next step position under our walking model. This sampled step space could be extended to allow side-stepping or turning around which the agent can do [3], though this is not currently accessed from the behavior system described in this paper. For each sampled step location, the potential field value is computed at the agent's new location, defined as the average location and orientation of the two feet.

2.4 An example behavior net

The example behavior net in Fig. 3 specifies an overall behavior for walking agents that head toward a particular goal object while avoiding obstacles (cylinders in this case) and each other. The entire graph is the *behavior net*, and each path from perception to motor output is considered a *behavior*. In this example there are three behaviors: one connecting a goal sensor to an attraction controller and then to the walk node (a goal-attraction behavior), another connecting a sensor detecting proximity of other walking agents to an avoidance controller and then to the walk node (a walker-avoidance behavior), and a final behavior connecting a cylinder proximity sensor to an avoidance behavior and then to the walk node (a cylinder-avoidance behavior).

Each node has a number of parameters that determine its behavior. For example, the walker sensor and the cylinder sensor nodes have parameters that indicate how they will average all perceived objects within their field of view and sensing distance into a single abstract object. The Attract and Avoid nodes have scaling weights that determine how much output to generate as a function of current input and the desired target values.

The walk motor behavior manages the sampling of the potential field by running data through the perceptual and control nodes with the agent pretending to be in each of the sampled step locations. The walk node then schedules the next step by passing the step location and orientation to the execution module.

Note that this example has no feedback, cross-talk, or inhibition within the controller, though the behavioral controller specification supports these features [5]. Although this example controller itself is a feed-forward network, it operates as a closed-loop controller when attached to the agent because the walk node's scheduling of steps affects the input to the perceptual nodes.

Our use of *attract* and *avoid* behaviors to control groups of walking agents may appear on the surface like Ridsdale's use of *hot* and *cold* tendencies to control agents in his *Director's Apprentice* system [18]. However, his system was not reactive and on-line as our behavioral controller is, it did not limit perception of agents, it had no structured facilities for tuning behavior parameters, and it did not take advantage of developments in reactive control and behavioral simulation. His system focused on the use of an expert system to schedule human activity conforming to stage principles and used hot and cold tendencies to manage complex human behavior and interaction. We limit the use of behaviors to reactive navigation and path-planning, using parallel transition networks rather than one large expert system to schedule events, and we look to symbolic planning systems based on results in cognitive science, such as [3, 8, 16], to automate high-level human behavior and complex human interactions.

3 Parallel Automata

Parallel Transition Networks (PaT-Nets) are transition networks that run in parallel with the behavior net, monitor it, and edit it over time [8]. They are a mechanism for scheduling arbitrary actions and introducing decision-making into the agent architecture. They monitor the behavior net (which may be thought of as modeling low level instinctive or reflexive behavior) and make decisions in special circumstances. For example, the agent may get caught in a dead-end or other local minimum. PaT-Nets recognize situations such as these, override the "instinctive" behavior simulation by reconfiguring connectivity and modifying weights in the behavior net, and then return to a monitoring state.

In our pedestrian example we combine object and location sensors (in perceptual nodes) with *attract* control nodes, and proximity and line sensors (in perceptual nodes) with *avoid* control nodes. Pedestrians are attracted to street corners and doors, and they avoid each other, light poles, buildings, and the street except at crosswalks.

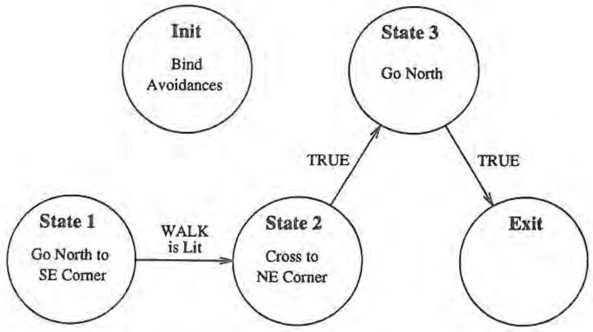


Figure 4: **North-net**: A sample **ped-net** shown graphically

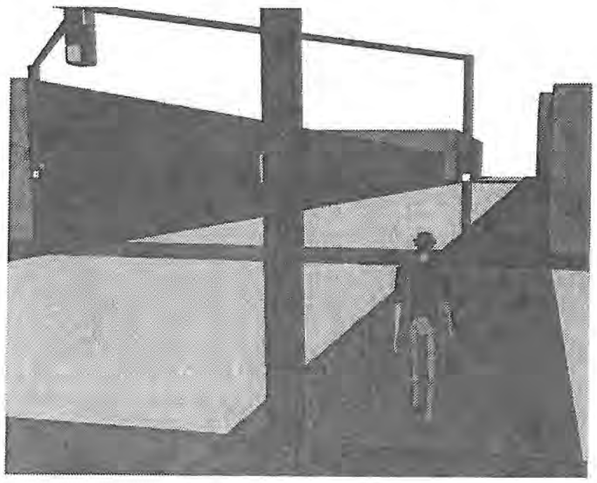


Figure 5: A pedestrian crossing the street

We use PaT-Nets in several different ways. **Light-nets** control traffic lights and **ped-nets** control pedestrians. **Light-nets** cycle through the states of the traffic light and the *walk* and *don't walk* signs.

Fig. 4 is a simple **ped-net**, a **north-net**, which moves a pedestrian north along the eastern sidewalk through the intersection. Initially, avoidances are bound to the pedestrian so that it will not walk into walls, the street, poles, or other pedestrians. The avoidances are always active even as other behaviors are bound and unbound. In State 1 an attraction to the southeast corner of the intersection is bound to the pedestrian. The pedestrian immediately begins to walk toward the corner avoiding obstacles along the way. When it arrives the attraction is unbound, the action for State 1 is complete. Nothing further happens until the appropriate walk light is lit. When it is lit, the transition to State 2 is made and action *Cross to NE Corner* is executed. The agent crosses the street. Finally, the agent heads north.

Fig. 5 shows a pedestrian controlled by a **north-net**. The transition to State 2 was just made so the pedestrian is crossing the street at the crosswalk.

4 Real-Time Simulation Environment

The run-time simulation system is implemented as a group of related processes, which communicate through shared memory. The system is broken into a minimum of 5 processes, as shown in Fig. 6. The system relies on IRIS Performer [19] for the general multiprocessing framework. Synchronization of all processes, via spin locks and video clock routines, is performed in the CONTROL process. It is also the only process which performs the edits and updates to the run-time visual database. The CULL and DRAW processes form a software rendering pipeline, as described in [19]. The pipeline improves overall rendering throughput while increasing latency, although the two frame latency between CONTROL and DRAW is not significant for our application. Our CONTROL process is equivalent to the APP process in the Performer framework. We have used this framework to animate multiple real-time human figures [12].

4.1 CONTROL Process

The CONTROL process runs the main simulation loop for each agent. This process runs the PaT-Nets, and underlying behavior net for each agent. While each agent has only one behavior net, they may have several PaT-Nets running, which sequence the parameters and connectivity of the nodes in the behavior net over time (as shown in Fig. 6).

By far the costliest computation in the CONTROL process, for the behaviors modeled in this example application, is the evaluation of the **Walk** motor node in the behavior net, and specifically the selection of the next foot position. Since this computation is done only once for every footfall, it usually runs only every 15 frames or so (the average step time being about 1/2 second, and average frame rate 30Hz). If the CONTROL process starts running over its allotted frame time, the **Walk** nodes will start reducing the number of points sampled for the next foot position, thereby reducing computation time. The only danger here is described in Section 2.3.1, the potential for a sawtooth path. If many agents are walking at similar velocities, they can all end up computing their next-step locations at the same frame-time, creating a large computation spike which causes the whole simulation to hiccup. (It is visually manifested by the feet landing in one frame, then the swing foot suddenly appearing in mid-stride on the next frame.) We attempt to even out the computational load for the **Walk** motor node evaluation by staggering the start times for each agent, and thereby distributing the computation over about 1/2 second for all agents.

Another computational load in the CONTROL process comes from the evaluation of the conditional expressions in the Pat-Nets, which may occur on every frame of the simulation. They are currently implemented via LISP expressions, so evaluating a condition involves parse and eval steps. In practice, this is fairly fast as we precompile the LISP, but as the PaT-Nets increase in complexity it will be necessary to replace LISP with a higher performance language (i.e. compiled C code). This may remove some of the generality and expressive power enjoyed with LISP.

Another technique employed to improve performanc , when evaluating a large number of Pat-Nets and behavior nets, is to have the CONTROL process spawn copies of itself, with each copy running the behavior of a subset of the agents. This works as long as updates to the visual database are exclusive to each CONTROL process. (In practice this is the case, since the current behavior net for one agent will not edit any parameters for another agent in the visual database.) Of course, the assumption in spawning more processes is that there are available CPUs to run them.

The CONTROL process also provides the outputs of the motor nodes in the behavior net to the MOTION process. These outputs, in the case of the walking behavior, are the position and orientation of the agent's next foot fall. It also evaluates the motion data (joint angles) coming from the MOTION process, and performs the necessary updates to the articulation matrices of the human agent in the visual database.

4.2 SENSE Process

The SENSE process controls and evaluates the simulated sensors modeled in the perceptual nodes of the behavior net. It provides the outputs of the perceptual nodes to the CONTROL process, which uses them for the inputs to the control nodes of the behavior net. The main computational mechanism the sensors employ are intersections of simple geometric shapes (a set of points, lines, frustums or cones) with the visual database, as well as distance computations. This process corresponds to an ISECT process in the Performer framework.

The major performance parameters of this process are the total number of sensors as well as the complexity and organization of the visual database. Since it needs read-only access to the visual database, several SENSE processes may be spawned to balance the load between the number of sensors being computed, and the time needed to evaluate them. (These extra processes are represented by the dotted SENSE process in Fig. 6.) There is a one frame latency between the outputs of the perceptual nodes and the inputs to the control nodes in the behavior net (which are run in the CONTROL process), but this is not a significant problem for our application.

4.3 MOTION Process

Once the agent has sensed its environment and decided on on appropriate action to take, its motion is rendered via real-time motion generators, using a motion system that mixes pre-recorded playback and fast motion generation techniques.

We use an off-line motion authoring tool [2, 13] to create and record motions for our human figures. The off-line system organizes motion sequences into *posture graphs* (directed, cyclic graphs). Real-time motion playback is simply a traversal of the graph in time. This makes the run-time motion generation free from frame-rate variations. The off-line system also records motions for several levels-of-detail (LOD) models of the human figure. (Both the bounding geometry of the figure, as well as the articulation hierarchy (joints) are represented at several levels of detail.) The three levels-of-detail we are using for the human figure are:

1. A 73 joint, 130 DOF, 2000 polygon model, which has articulated fingers and flexible torso, for use in close-up rendering, and fine motor tasks (*Jack*®),

2. A 17 joint, 50 DOF, 500 polygon model, used for the bulk of rendering; it has no fingers, and the flexible torso has been replaced by two joints,

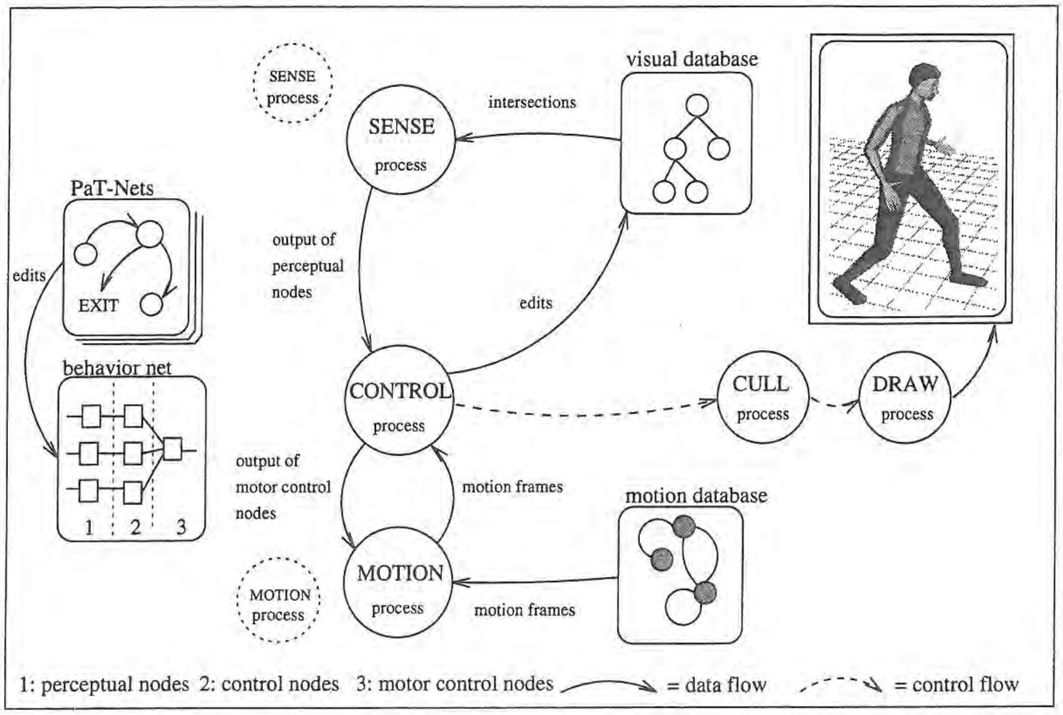


Figure 6: The multiprocessing framework for the real-time behavior execution environment

3. An 11 joint, 21 DOF, 120 polygon model used when the human agent is at a large distance from the camera.

This process produces a frame of motion for each agent, then sleeps until the next frame boundary (the earliest any new motion could be needed). It provides the correct motion frame for the currently active LOD model in the visual database. For certain types of sensors modeled in the perceptual nodes, this process will also be requested to provide a full (highest LOD) update to the visual database, in the case where a lower LOD is currently being used, but a sensor needs to interact with the highest LOD model.

The motion database consists of one copy of the posture graphs and associated motion between nodes of the posture graph. Each transition is stored at a rate of 60HZ, on each LOD model of the human agent. This database is shared by all agents. Only a small amount of private state information is maintained for each agent.

The MOTION process can effectively handle about 10-12 agents at update rates of 30Hz (on a 100MHz MIPS R4000 processor). Since the process only has read-only access to the motion database, we can spawn more MOTION processes if needed for more agents.

4.4 Walking as an example

A MOTION process animates the behaviors specified by an agent's motor nodes by playing back what are essentially pre-recorded chunks of motion. As a time-space tradeoff, this technique provides faster and less variable run-time execution at the cost of additional storage requirements and reduced generality. The interesting issues arise in how we choose a mapping from motor node outputs to this discrete representation; it plays a significant role in determining how realistic the animated agents will be.

The primary motor behavior to be executed is walking. Our full walking algorithm combines kinematics with dynamic balance control and is capable of generating arbitrary curved-path locomotion [15]. In order to reduce computational costs, however, we have not incorporated the algorithm directly into our run-time system. Instead, as implied by the preceding discussion, we record canonical "left" and "right" steps generated by the algorithm (which **is** a component of our off-line motion authoring system) and then play them back in an alternating fashion to produce a continuous walking motion.

The input to the appropriate MOTION process's walking subsystem consists of the specification of the desired next foot position and orientation (for the swing foot). This input is itself already discretized, as the motor node responsible (the **Walk** motor node) for evaluating how desirable it is for the agent to be at particular positions only computes the desirability criteria at a set number of points (in Fig. 2). However, even given that there are only n possibilities for the placement of the swing foot on the next step, this would still require us to record order n^2 possible steps, since the planted foot could be in any one of the n different positions at the start of the step (determined by the **last** step taken) and any one of the n at the end.

Without recording all n^2 distinct steps it is necessary to choose the best match among those that we do record. One of the most important criteria in obtaining realistic results is to minimize foot slippage relative to

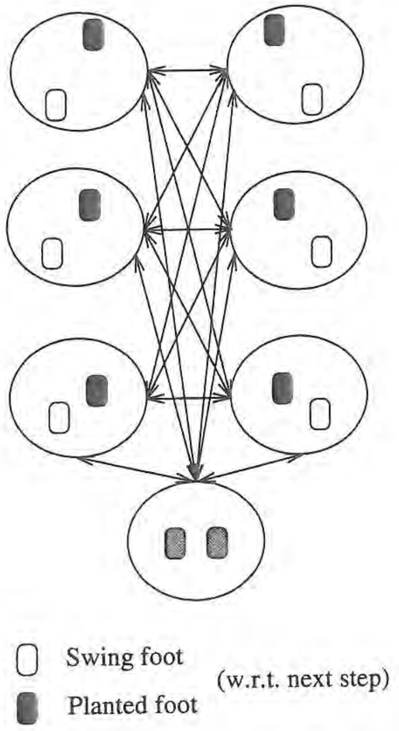


Figure 7: Posture graph for variable step length walking (3 step sizes)

the ground; foot slippage occurs when the pre-recorded movement (in particular its amount and direction) does not match that specified by the **walk** motor node at run time. On the basis that translational foot slippage is far more evident than rotational slippage (at least from our informal observations), we currently adopt an approach in which we record three types of step: short, medium, and long. Turning is accomplished by rotating the agent around his planted foot smoothly throughout the step. Having three step sizes significantly increases the chances of being able to find a close match to the desired step size, and, in fact, the **walk** motor node can be constrained to **only** consider the three arcs of the next foot location fan (see Fig. 2) that correspond exactly to our recorded step sizes. Doing so eliminates translational slippage, but has the sawtooth hazard.

The posture graph for all possible step-to-step transitions is shown in Fig. 4.4. Notice that even with only three kinds of straight-line walking there are many possible transitions, and hence numerous motion segments to be recorded. However, allowing for variable step length is very important. For instance, an attract control node can be set to drive the agent to move within a certain distance of a goal location; were there only a single step size, the agent might be unable to get sufficiently close to the goal without overshooting it each time, resulting in degenerate behavior (and possible virtual injury).

One thing worthy of mention with respect to the number of different walking steps required to reproduce arbitrary curved-path locomotion is that while there are theoretically order n^2 of them, the similarities are significant. It is thus possible that it will prove feasible to store a single full set of steps along with a little more information to represent how those steps can be modified slightly to realistically turn the agent left or right, and make it sufficiently fast for our real-time applications.

5 Conclusions and Future Work

We have designed a multiprocessing system for the real-time execution of behaviors and motions for simulated human-like agents. We have used only toy examples to date, and are eager to push the limits of the system to model more complex environments and interactions amongst the agents.

Although our agents currently have limited abilities (locomotion and simple posture changes), we will be developing the skills for interactive agents to perform maintenance tasks, handle a variety of tools, negotiate terrain, and perform tasks in cramped spaces. Our goal is a system which does not provide for all possible behaviors of a human agent, but allows for new behaviors and control techniques to be added and blended with the behaviors and skills the agent already possesses.

We have used a coarse grain parallelism to achieve interactive frame rates. The behavior net lends itself to finer grain parallelism, as one could achieve using a threaded approach. Our system now is manually tuned and balanced (between the number of agents, the number of sensors per agent, and the complexity of the visual database). A fruitful area of research is in the automatic load balancing of the MOTION and SENSE processes, spawning and killing copies of these processes, and doling out agents and sensors, as agents come and go in the virtual environment. Results in real-time system scheduling and approximation algorithms will be applicable here.

6 Acknowledgments

This research is partially supported by ARO DAAL03-89-C-0031 including U.S. Army Research Laboratory; Naval Training Systems Center N61339-93-M-0843; Sandia Labs AG-6076; ARPA AASERT DAAH04-94-G-0362; DMSO DAAH04-94-G-0402; ARPA DAMD17-94-J-4486; U.S. Air Force DEPTH through Hughes Missile Systems F33615-91-C-0001; DMSO through the University of Iowa; and NSF CISE CDA88-22719.

References

[1] Ronald C. Arkin. Integrating behavioral, perceptual, and world knowledge in reactive navigation. In Pattie Maes, editor, *Designing Autonomous Agents*, pages 105–122. MIT Press, 1990.

[2] Norman I. Badler, Rama Bindiganavale, John Granieri, Susanna Wei, and Xinmin Zhao. Posture interpolation with collision avoidance. In *Proceedings of Computer Animation '94*, Geneva, Switzerland, May 1994. IEEE Computer Society Press.

[3] Norman I. Badler, Cary B. Phillips, and Bonnie L. Webber. *Simulating Humans: Computer Graphics, Animation, and Control.* Oxford University Press, June 1993.

[4] Welton Becket. *Simulating Humans: Computer Graphics, Animation, and Control*, chapter Controlling forward simulation with societies of behaviors.

[5] Welton Becket and Norman I. Badler. Integrated behavioral agent architecture. In *The Third Conference on Computer Generated Forces and Behavior Representation*, Orlando, Florida, March 1993.

[6] Welton M. Becket. *Optimization and Policy Learning for Behavioral Control of Simulated Autonomous Agents.* PhD thesis, University of Pennsylvania, 1995. In preparation.

[7] Valentino Braitenberg. *Vehicles: Experiments in Synthetic Psychology.* The MIT Press, 1984.

[8] J. Cassell, C. Pelachaud, N. Badler, M. Steedman, B. Achorn, W. Becket, B. Douville, S. Prevost, and M. Stone. Animated conversation: rule-based generation of facial expression, gesture and spoken intonation for multiple conversational agents. In *Proceedings of SIGGRAPH '94. In Computer Graphics*, pages 413–420, 1994.

[9] Thomas L. Dean and Michael P. Wellman. *Planning and Control.* Morgan Kaufmann Publishers, Inc., 1991.

[10] R. James Firby. Building symbolic primitives with continuous control routines. In *Artificial Intelligence Planning Systems*, 1992.

[11] C. R. Gallistel. *The Organization of Action: A New Synthesis.* Lawrence Elerbaum Associates, Publishers, Hillsdale, New Jersey, 1980. Distributed by the Halsted Press division of John Wiley & Sons.

[12] John P. Granieri and Norman I. Badler. In Ray Earnshaw, John Vince, and Huw Jones, editors, *Applications of Virtual Reality*, chapter Simulating Humans in VR. Academic Press, 1995. To appear.

[13] John P. Granieri, Johnathan Crabtree, and Norman I. Badler. Off-line production and real-time playback of human figure motion for 3d virtual environments. In *IEEE Virtual Reality Annual International Symposium*, Research Triangle Park, NC, March 1995. To appear.

[14] David R. Haumann and Richard E. Parent. The behavioral test-bed: obtaining complex behavior from simple rules. *The Visual Computer*, 4:332–337, 1988.

[15] Hyeongseok Ko. *Kinematic and Dynamic Techniques for Analyzing, Predicting, and Animating Human Locomotion.* PhD thesis, University of Pennsylvania, 1994.

[16] Micheal B. Moore, Christopher W. Geib, and Barry D. Reich. Planning and terrain reasoning. In *Working Notes - 1995 AAAI Spring Symposium on Integrated Planning Applications.*, 1995. to appear.

[17] Barry D. Reich, Hyeongseok Ko, Welton Becket, and Norman I. Badler. Terrain reasoning for human locomotion. In *Proceedings of Computer Animation '94*, Geneva, Switzerland, May 1994. IEEE Computer Society Press.

[18] Gary Ridsdale. *The Director's Apprentice: Animating Figures in a Constrained Environment.* PhD thesis, Simon Fraser University, School of Computing Science, 1987.

[19] John Rohlf and James Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. *Computer Graphics*, pages 381–394, 1994.

[20] Jane Wilhelms and Robert Skinner. A 'notion' for interactive behavioral animation control. *IEEE Computer Graphics and Applications*, 10(3):14–22, May 1990.

Impulse-based Simulation of Rigid Bodies

Brian Mirtich [*]
John Canny [†]
University of California at Berkeley

Abstract

We introduce a promising new approach to rigid body dynamic simulation called impulse-based *simulation. The method is well suited to modeling physical systems with large numbers of collisions, or with contact modes that change frequently. All types of contact (colliding, rolling, sliding, and resting) are modeled through a series of collision impulses between the objects in contact, hence the method is simpler and faster than constraint-based simulation. We have implemented an impulse-based simulator that can currently achieve interactive simulation times, and real time simulation seems within reach. In addition, the simulator has produced physically accurate results in several qualitative and quantitative experiments. After giving an overview of impulse-based dynamic simulation, we discuss collision detection and collision response in this context, and present results from several experiments.*

1 Introduction

The foremost requirement of a dynamic simulator is physical accuracy. The simulation is to take the place of a physical model, and hence its utility is directly related to how well it mimics this model. A second important requirement is computational efficiency. Many applications (e.g. electronic prototyping [9]) benefit most from interactive simulation; others (e.g. virtual reality) demand real time speeds.

This paper discusses a new approach to dynamic simulation called impulse-based simulation, founded on the twin goals of physical accuracy and computational efficiency. The initial results from our impulse-based simulator look very promising, both from speed and accuracy standpoints. In this paper we give an overview of the impulse-based approach, then discuss collision detection and resolution and results from several experiments.

[*] *mirtich@cs.berkeley.edu*, Department of Computer Science, 387 Soda Hall, University of California, Berkeley, CA 94720. Supported in part by NSF grant #FD93-19412.

[†] *jfc@cs.berkeley.edu*, Department of Computer Science, 529 Soda Hall, University of California, Berkeley, CA 94720. Supported in part by NSF grant #FD93-19412.

1.1 Related work

Moore and Wilhelms give one of the earliest treatments of two fundamental problems in dynamic simulation: collision detection and collision response [14]. Hahn also pioneered dynamic simulation, modeling sliding and rolling contacts using impact equations [8]. His work is the precursor of our method, although we extend the applicability of impulse dynamics to resting contacts, and model multiple objects in contact with impulse trains as well. These early approaches all suffered from inefficient collision detection and unrealistic assumptions concerning impact dynamics (e.g. infinite friction at the contact point).

Cremer and Stewart describe *Newton* [7, 17], probably the most advanced general-purpose dynamic simulator in use today. Newton's forte is the formulation and simulation of constraint-based dynamics for linked rigid bodies, although the contact modeling is fairly simplistic. Baraff has studied multiple rigid bodies in contact [1, 2], and shown that computing contact forces in the presence of friction is NP-hard [3]. A summary of his work in this area appears in [4].

There are few full treatments of frictional collisions. Routh [16] is still considered the authority on this subject, and more recently, Keller gives an excellent treatment of frictional collisions [10]. Our analysis is extremely similar to that of Bhatt and Koechling, who independently derived the same key equation for integration of relative contact velocities during impact. They give a classification of frictional collisions, based on the flow patterns of tangential contact velocity [6].

Wang and Mason have studied two-dimensional impact dynamics for robotic applications, based on Routh's approach [18]. Finally, a number of researchers have investigated several problems and paradigms for dynamic simulation and physical-based modeling [5, 19, 20].

2 The impulse-based method

One of the most difficult aspects of dynamic simulation is dealing with the interactions between bodies in contact. Most of the work which has been done in this area falls into the category of constraint-based methods [4, 5, 7, 19]. An example will illustrate the approach. Consider a ball rolling along a table top. The normal force which the table exerts on the ball is a constraint force that does no work on the ball, but only enforces a non-penetration constraint. In the Lagrangian constraint-based approach, this force is not modeled explicitly, but is accounted for by a constraint on the configuration of the ball (here, its z-coordinate is held constant). Alternatively, one may model the forces explicitly, solving for their magnitudes using Lagrange multipliers. However this still requires complete, exact knowledge of the instantaneous state of contact between the objects, since that determines where and when such forces can exist.

A problem with this method is that as a dynamic system evolves, the constraints may change many times, e.g. the ball may roll off the table, may hit an object on the table, etc. Determining the correct equations of motion for the ball means keeping track of these changing constraints, which can become complicated. Moreover, it is not even always clear what type of constraint should be applied; there exist at least two models for rolling contact which in some cases predict different behaviors [11]. Finally, impacts are not easily incorporated into the constraint model, as they generally give rise to impulses, not constraint forces present over some interval. These collision impulses must be handled separately, as in [1].

In contrast to constraint-based methods, impulse-based dynamics involves no explicit constraints on the configurations of the moving objects; when the objects are not colliding, they are in ballistic trajectories. Furthermore, all modes of continuous contact are handled via trains of impulses applied to the objects, whether they be resting, sliding, or rolling on one another. Under impulse-based simulation, a block resting on a table is actually experiencing many rapid, tiny collisions with the table, each of which is resolved using only local information at the collision point.

Now consider the case of a ball bouncing along the terrain shown in figure 1. Under constraint-based simulation, the

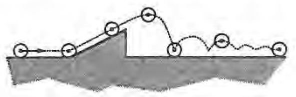


Figure 1: *A nightmare for constraint-based simulation.*

constraints change as the ball begins traveling up the ramp, leaves the ramp, and settles into a roll along the ground. All these occurrences must be detected and processed. Impulse-based simulation avoids having to worry about such transitions. In this sense, it is a more physically sound treatment since it does not establish an artificial boundary between, for example, bouncing and rolling, but instead handles the entire continuum of contact between these phases.

We do not wish to discredit constraint-based methods of dynamic simulation; indeed, there are many situations for which they are the perfect tool. We believe the impulse-based method is better suited to simulating many common physical systems, especially those which are collision intensive, or that have many changes in contact mode. We examine the possibility of using both methods of simulation together, combining the strengths of each, in section 6.

Two obvious questions concerning impulse-based simulation are: (1) Does it work, i.e. does it result in physically accurate simulations?, and (2) Is it fast enough to be practical? We defer more thorough answers to these questions to section 5, but for now state the following: impulse-based dynamic simulation *does* produce physically accurate results, and the approach is extremely fast. Simulations can certainly be run interactively with our current implementation, and we believe real time simulation is a reachable goal.

3 Collision detection

Impulse-based dynamic simulation is inherently collision intensive, since collisions are used to affect all types of interaction between objects. Hahn found collision detection to be the bottleneck in dynamic simulation [8], and efficient data structures and algorithms are needed to make impulse-based simulation feasible.

Currently in our simulator, all objects are geometrically modeled as convex polyhedra or combinations of them. The polyhedral restriction is not at all severe, because our collision detection system is very insensitive to the complexity of the geometric models, permitting fine tessellations. Indeed, some of the simulations described in section 5 use polyhedral models with over 20,000 facets, with negligible slowdown.

3.1 Prioritizing collisions

Obviously, checking for possible collisions between all pairs of objects after every integration step is too inefficient. Instead, collisions are prioritized in a heap (see figure 2). For

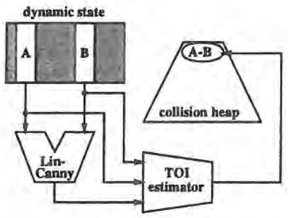


Figure 2: *Prioritizing collisions in a heap.*

each pair of objects in the simulation, there is an element in the heap, which also contains a lower bound on the time of impact (TOI) for the given pair of objects. The heap is sorted on the TOI field, thus the TOI field of the top heap element always gives a "safe" value for the next collision free integration step.

After an integration step, the distance between the objects on the top of the heap (call them A and B) must be recomputed. In our implementation, we use the Lin-Canny closest features algorithm [12]. This is an extremely efficient algorithm which maintains the closest features (vertices, edges, or faces) between a pair of convex polyhedra. It is fastest in applications like dynamic simulation, when the objects move continuously through space and geometric coherence can be exploited.

Collisions are declared when the distance between objects falls below some threshold ε_c. First suppose the distance between A and B lies above ε_c. In this case, the dynamic states of A and B along with the output of the Lin-Canny algorithm are used to compute a new conservative bound on the time of impact of A and B. The A–B heap pair is updated with this new value, possibly affecting its heap position, and the integrator is ready for another step.

If the distance between A and B is less than ε_c, a collision is declared. The collision resolution system computes and applies collision impulses to the two objects, changing their dynamic state. At this point the TOI is recomputed for these objects as before, however another step is necessary: the TOI between all object pairs of the form A–x and B–x must also be recomputed. The reason is that the TOI estimator uses a ballistic trajectory assumption to bound the time of impact for a pair of objects. Applying collision impulses to objects violates this assumption, and so every previous TOI involving such an object becomes invalid. Note that this is an $O(n)$ update step.

3.2 Further reducing collision checks and TOI updates

The strategy described above reduces collision checks significantly, especially between objects which are far apart or moving slowly. However, the number of collision checks is still $O(n^2)$ because they are performed periodically between every pair of objects. A more serious problem is the $O(n)$ TOI update step that must be performed every time a collision impulse is applied to an object. What the heap scheme misses is the fact that some objects never come near each other, and collision checks as well as TOI updates for such pairs of objects are unnecessary.

To alleviate this problem, we employ a spatial tiling technique based on Overmars' efficient point-location algorithms in fat subdivisions [15]. For each object i in the simulation, one can easily find an enclosing, axis-aligned rectangular volume B_i which is guaranteed to contain the object during the next integration step. This is possible because of the ballistic trajectory assumption.

The idea is to keep track of which objects are near each other, by keeping track of which bounding boxes overlap. To this end, the physical space is partitioned into a cubical tiling with resolution ρ. Under this tiling, Coordinates in physical space are mapped to integers under the tiling map τ:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \xrightarrow{\tau} \begin{bmatrix} \lfloor x/\rho \rfloor \\ \lfloor y/\rho \rfloor \\ \lfloor z/\rho \rfloor \end{bmatrix}. \tag{1}$$

Let S_i be the set of tiles which B_i intersects. We store i in a hash table multiple times, hashed on the coordinates of each tile in S_i. Clearly objects i and j can only possibly collide during the next integration step if i and j are both present in some hash bucket. Only in this case do we keep object pair i-j in the collision heap. Furthermore, if object i experiences a collision impulse, TOIs need only be recomputed for object pairs i-k, where object k shares a hash bucket with object i.

This scheme tremendously reduces the number of collision checks and TOI computations that must be performed, since most objects are generally in the vicinity of only a small subset of the set of all objects. Collision detection is still $O(n^2)$ in the worst case, but almost always better. Consider for example the case of simulating a vibratory bowl feeder sorting hundreds of small parts. Since the number of parts near another part can be bounded by a constant, the number of collision checks are $O(n)$.

One added wrinkle is that one must actually employ a hierarchy of spatial tilings and hash tables of varying resolutions, in order to prevent having to hash a sofa according to tiles the size of ice cubes. The hierarchy is needed to keep the rate of bucket updates small. See Overmars for more information on this multiple resolution hashing scheme [15].

3.3 Time of impact estimator

The time of impact (TOI) estimator takes the current dynamic state (pose and velocity) of two objects as well as the closest points between them, and returns a lower bound on the time of impact for those two objects. We assume the objects are convex; concavities are handled by convex decomposition.

Let $\mathbf{c}_i$ and $\mathbf{c}_j$ be the current closest points between two objects i and j on a collision course. Let $\hat{\mathbf{d}}$ be a unit vector in the direction of $\mathbf{c}_i - \mathbf{c}_j$, and d be the distance between $\mathbf{c}_i$ and $\mathbf{c}_j$. A convexity argument shows that no matter where the ultimate contact points are located, these contact points must cover the distance d in the direction of $\hat{\mathbf{d}}$ before collision can occur. From this one obtains a conservative bound on the time of collision:

$$t_c \geq \frac{d}{(\mathbf{v}_j - \mathbf{v}_i) \cdot \hat{\mathbf{d}} + r_i\omega_i + r_j\omega_j}, \tag{2}$$

where $\mathbf{v}$ denotes center of mass velocity, r denotes maximum "radius," ω denotes maximum angular velocity magnitude, and the subscripts refer to the body. This bound assumes both objects are ballistic, so that gravitational effects cancel out. If, for instance, object i is a fixed table top, then the gravitational acceleration of j must be accounted for.

The conservation of momentum can be used to bound the angular velocity magnitude of a body in a ballistic trajectory:

$$\omega_{max} \leq \frac{\|(J_x\omega_x, J_y\omega_y, J_z\omega_z)^T\|}{\min(J_x, J_y, J_z)}, \tag{3}$$

where $\mathbf{J}$ is the vector of diagonal elements of the diagonalized mass matrix, and ω is the current angular velocity.

4 Computing collision impulses

When two bodies collide, an impulse $\mathbf{p}$ must be applied to one of the bodies to prevent interpenetration; an equal but opposite impulse $-\mathbf{p}$ is applied to the other. Once $\mathbf{p}$ and its point of application are known, it is a simple matter to compute the new center of mass and angular velocities for each body. After updating these velocities, dynamic state evolution can continue, assuming ballistic trajectories for all moving objects. The point of application is computed by the collision detection system, and hence the central problem in collision resolution is to determine the collision impulse $\mathbf{p}$. Accurate computation of this impulse is critical to the physical accuracy of the simulator. We now discuss how $\mathbf{p}$ may be computed; a more detailed discussion can be found in [13].

4.1 Assumptions for collisions

For impulse-based simulation, it is not feasible to make gross simplifying assumptions such as frictionless contacts or perfectly elastic collisions. Our approach for analyzing general frictional impacts is similar to that of Routh [16], although we derive equations which are more amenable to numerical integration. Keller also gives an excellent treatment [10], and Bhatt and Koechling's analysis is quite similar to ours [6]. There are three assumptions central to our analysis:

1. Infinitesimal collision time
2. Poisson's hypothesis
3. Coulomb friction model

The infinitesimal collision time assumption is commonly made in dynamic simulation [10]. It implies that the positions of the objects can be treated as constant over the course of a collision. Furthermore, the effect of one object on the other can be described by an impulse, which unlike a normal force can instantaneously change velocities. This assumption does *not* imply that the collision can be treated as a discrete event. The velocities of the bodies are not constant during the collision, and since collision (frictional) forces depend on these velocities, it is necessary to examine the dynamics during the collision. In short, a collision is a single point on the time line of the simulation, but to determine the collision impulses which are generated, one must use a magnifying glass to "blow up" this point, examining what happens inside the collision.

Poisson's hypothesis is an approximation to the complex deformations and energy losses which occur when two real bodies collide. Trying to explicitly model these stresses and deformations is too slow for interactive simulation; Poisson's hypothesis is a simple empirical rule that captures the basic behavior during a collision. A collision is divided into a compression and a restitution phase, based on the direction of the relative contact velocity along the surface normal. The boundary between these phases is the point of maximum compression, at which point the relative normal contact velocity vanishes. Let p_{total} be the magnitude of the normal component of the impulse imparted by one object onto the other over the entire collision, and p_{mc} be the magnitude of the normal component of the impulse just over the compression phase, i.e. up to the point of maximum compression. Poisson's hypothesis states

$$p_{total} = (1+e)p_{mc} \tag{4}$$

where e is a constant between zero and one, called the coefficient of restitution, that is dependent on the objects' materials.

Our final assumption is the Coloumb friction law. At a particular point during a collision between bodies A and B, let $\mathbf{u}$ be the contact velocity of A relative to B, let $\mathbf{u}_t$ be the tangential component of $\mathbf{u}$, and let $\hat{\mathbf{u}}_t$ be a unit vector in the direction of $\mathbf{u}_t$. Let $\mathbf{f}_n$ and $\mathbf{f}_t$ be the normal and tangential (frictional) components of force exerted by B on A, respectively. Then

$$\mathbf{u}_t \neq 0 \quad \Rightarrow \quad \mathbf{f}_t = -\mu\|\mathbf{f}_n\|\hat{\mathbf{u}}_t \tag{5}$$

$$\mathbf{u}_t = 0 \quad \Rightarrow \quad \|\mathbf{f}_t\| \leq \mu\|\mathbf{f}_n\| \tag{6}$$

where μ is the coefficient of friction. While the bodies are sliding relative to one another, the frictional force is exactly opposed to the direction of sliding. If the objects are sticking (i.e. $\mathbf{u}_t$ vanishes), all that is known is that the total force lies in the friction cone.

4.2 Initial collision analysis

A possible collision is reported whenever the distance between two bodies falls below the collision epsilon, ε_c. This is only a *possible* collision, because the objects may be receding. If the normal component of the relative velocity of the closest points has appropriate sign, no collision impulse should be applied. Note we are assuming the existence a normal direction; polyhedral objects have discontinuous surface normals, however reasonable surface normals can always be found.

Establish a collision frame with the z-axis aligned with the collision normal, directed towards body 1. Let $\mathbf{u} = \mathbf{u}_1 - \mathbf{u}_2$ be the relative contact velocity between bodies 1 and 2. When $u_z < 0$, a collision impulse must be applied to prevent interpenetration; it is necessary to analyze the dynamics of the bodies during the collision to determine this impulse. We use γ to denote the collision parameter; that is, γ is a variable which starts at zero, and continuously increases through the course of the collision until it reaches some final value, γ_f. All velocities are functions of γ, and $\mathbf{p}(\gamma)$ denotes the impulse delivered to body 1 up to point γ in the collision. The goal is to determine $\mathbf{p}(\gamma_f)$, the final total impulse delivered.

Initially, one might choose γ to be time since start of impact, but in fact this is not a very good choice. If the dynamics are studied with respect to time, the collision impulses are computed by integrating force. Unfortunately, the forces generated during a collision are not easily known; one can assume a Hooke's law behavior at the contact point, begging the question of how to choose the spring constants. Nonetheless, a variety of "penalty methods" do attempt to choose such spring constants.

A way of avoiding this problem is to choose a different parameter for the collision, namely $\gamma = p_z$, the normal component of the impulse delivered to body 1. The scalar p_z is zero at the moment the collision begins, and increases during the entire course of the collision, so it is a valid parameter. Let $\Delta\mathbf{u}(\gamma)$ denote the total change in relative contact velocity at point γ in the collision, and $\mathbf{p}(\gamma)$ be the impulse delivered to body 1 up to this point. Straightforward physics leads to the equation

$$\Delta\mathbf{u}(\gamma) = M\mathbf{p}(\gamma) \tag{7}$$

(see [13] for a detailed analysis). Here, M is a 3×3 matrix dependent only upon the masses and mass matrices of the colliding bodies, and the locations of the contact points relative to their centers of mass. By our infinitesimal collision time assumption, M is constant over the entire collision. It is useful to differentiate equation 7 with respect to the collision parameter γ, obtaining

$$\mathbf{u}'(\gamma) = M\mathbf{p}'(\gamma). \tag{8}$$

4.3 Sliding mode

While the tangential component of $\mathbf{u}$ is non-zero, the bodies are sliding relative to each other, and $\mathbf{p}'$ is completely constrained. Let $\theta(\gamma)$ be the relative direction of sliding during the collision, that is $\theta = \arg(u_x + iu_y)$.

Lemma 1 *If the collision parameter γ is chosen to be p_z, then while the bodies are sliding relative to one another,*

$$\mathbf{p}' = \begin{bmatrix} -\mu\cos\theta \\ -\mu\sin\theta \\ 1 \end{bmatrix}. \tag{9}$$

Proof: $p'_x = \frac{dp_x}{dp_z} = \frac{dp_x}{dt}\frac{dt}{dp_z} = f_x\frac{dt}{dp_z}$, where $\mathbf{f}$ is the instantaneous force exerted by body 2 on body 1. Under sliding conditions, $f_x = -(\mu\cos\theta)f_z = -(\mu\cos\theta)\frac{dp_z}{dt}$. Combining results gives $p'_x = -\mu\cos\theta$. The derivation for p'_y is similar. Finally, $p'_z = \frac{dp_z}{dp_z} \equiv 1$. □

It is now clear why p_z is a good choice for the collision parameter. By applying the results of lemma 1 to equation 8, with θ expressed in terms of u_x and u_y, we obtain:

$$\begin{bmatrix} u'_x \\ u'_y \\ u'_z \end{bmatrix} = M \begin{bmatrix} -\mu\frac{u_x}{\sqrt{u_x^2+u_y^2}} \\ -\mu\frac{u_y}{\sqrt{u_x^2+u_y^2}} \\ 1 \end{bmatrix}. \tag{10}$$

This nonlinear differential equation for $\mathbf{u}$ is valid as long as the bodies are sliding relative to each other. By integrating the equation with respect to the collision parameter γ (i.e. p_z), we can track $\mathbf{u}$ during the course of the collision. Projections of the trajectories into the u_x-u_y plane are shown in figure 3 for a particular matrix M; the crosses mark the initial sliding velocities.

The basic impulse calculation algorithm proceeds as follows. After computing the initial $\mathbf{u}$ and verifying that u_z is negative, we numerically integrate $\mathbf{u}$ using equation 10. During this integration, u_z will increase[1]. When it reaches zero, the point of maximum compression has been attained.

[1]Baraff and others have noted that it is possible to construct cases for which u_z decreases as p_z increases [3]. However, this situation seems to be extremely rare; it has not occurred in any of our simulations.

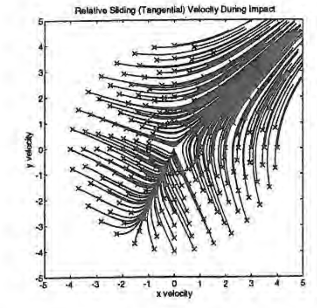


Figure 3: *Solution trajectories of equation 8 projected into the u_x-u_y plane.*

At this point, p_z is the total normal impulse which has been applied during compression. Multiplying this value by $(1 + e)$ gives the terminating value for the collision parameter, γ_f. The integration then continues to this point, to obtain $\Delta\mathbf{u}(\gamma_f)$. Inverting equation 7 then gives the total collision impulse $\mathbf{p}(\gamma_f)$.

4.4 Sticking mode

When the relative tangential velocity vanishes, the direction of the frictional force is not known a priori, and lemma 1 no longer applies. We assume like Routh that if the frictional force is strong enough to maintain the sticking condition, it will do so. To see if this is the case, we set $u'_x = u'_y = 0$ in equation 8, and solve for $\mathbf{p}'$. There is a unique solution for which $p'_z = 1$, say $\mathbf{p}' = (\alpha, \beta, 1)^T$. If

$$\alpha^2 + \beta^2 \leq \mu^2, \tag{11}$$

the friction is sufficient to maintain sticking, and so $u_x = u_y = 0$ and $\mathbf{p}' = (\alpha, \beta, 1)^T$ for the remainder of the collision.

If $\alpha^2 + \beta^2 > \mu^2$, the friction is not sufficient to maintain sticking, and sliding will immediately resume. Equation 10 is not valid when $u_x = u_y = 0$, and so is of no help in predicting the initial direction of sliding. In the case depicted in figure 3, there is a unique sliding direction leaving the origin; sliding must resume along this direction. It can be proven that the trajectories of equation 10 projected into the u_x-u_y plane never spiral around the origin, and we conjecture that in cases when the friction is not sufficient to maintain sliding there is always exactly one sliding direction away from the origin. Once u_x or u_y is nonzero, equation 10 again applies.

Our previous algorithm for computing collision impulses must be slightly modified to account for possible sticking. If at any point during the integration of $\mathbf{u}$, u_x and u_y both vanish, the integration halts. If the criterion given by equation 11 is met, sticking is maintained for the duration of the collision and both $\mathbf{u}$ and $\mathbf{p}$ vary along a straight line. Otherwise, we solve a quartic equation to determine the inward and outward sliding directions for the collision, and take the next integration step along the (conjectured unique) outward sliding direction. Once the sliding has resumed, the normal integration can continue;

Figure 4 illustrates some of the possible trajectories of $\mathbf{u}$ for different collisions. Path A represents a collision under low friction, in which the tangential component of relative contact velocity never vanishes, and the two objects slide on one another during the entire collision. Path C corresponds to a collision in which the frictional forces bring the sliding contact to a halt; as the object rebound off each other there is no relative sliding velocity. Finally, path B corresponds to a case in which sticking occurs momentarily, but the friction is insufficient to maintain this condition and sliding resumes.

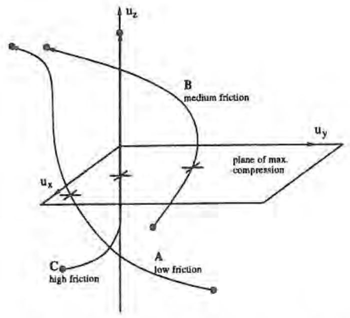


Figure 4: *Trajectories through relative contact velocity space for three different collisions.*

4.5 Static contact and microcollisions

The collision resolution method described thus far is suitable for resolving colliding contacts, but is not enough to model continuous contact as objects come to rest upon one another. In this case, the collisions must not produce an energy loss in the colliding objects, since they are modeling static forces which do no work.

Two important questions are: how can this static situation be detected using only local information at the contact point, and how should the collision model be modified to give the correct macroscopic behavior? Certainly the initial relative normal velocity at the contact point must be small; static contact only occurs as objects begin to settle onto one another. We define "small" precisely with the threshold v_e, the velocity an initially resting object acquires as it falls through the collision envelope:

$$v_e = \sqrt{2g\varepsilon_c}, \tag{12}$$

where g is the acceleration of gravity. If the relative normal velocity is below this threshold, a check is made to see if the impulse required to reverse the initial relative velocity lies within the friction cone, and if so, it is applied to resolve the collision. Such a collision is called a *microcollision*. One can show that microcollision impulses do no work on the object, just like the physical static contact forces that they model.

Microcollisions also solve another problem. Consider a block sitting on a shallow ramp with high friction, and modeling the contact as an impulse train. Even though the friction is sufficient to bring the sliding velocity to zero at every collision, the block will tend to creep down ramp because of the time it spends in a ballistic phase. However, the elastic nature of microcollisions will negate the effect of gravity during the intervening ballistic phases, by giving the block a small "kick" back up the ramp, once the tangential contact velocities become small enough. Figure 6 shows that microcollisions can bring the block to a complete stop.

The question arises as to whether microcollisions are not just some ad-hoc modification necessary to make impulse-based dynamics work. After all, one of the attractive features of the impulse-based method is that one need not enforce strict continuous contact constraints between obstacles.

Are microcollisions just such a constraint in disguise? The answer is no. As objects settle on one another, they experience a number of small collisions, none of which are initially microcollisions. Gradually, microcollisions account for a larger and larger fraction of total collisions, until eventually all collisions are microcollisions. In other words, there is a smooth transition between colliding and continuous contact. Moreover, the decision to apply a microcollision is based solely on local information at the contact point, not on some global information about the state of the system.

5 Results and Analysis

We have tested our simulator on a wide variety of problems. We now describe some qualitative and quantitative results.

5.1 Pool break

This simulation involved breaking a rack of fifteen pool balls with a high velocity cue ball. Constraint-based simulators have trouble with this example because of the large number of mutual contacts between the racked balls. Baraff has shown that the problem of finding a set of contact forces that instantaneously obey the Coulomb friction law at every contact point is NP-hard [3]. Furthermore, the contact constraints are quite transient, making it difficult to integrate along equations of motion derived from them.

The impulse-based method avoids these problems by treating the contacts as a series of closely spaced collisions. The racked balls (of standard size) were initially placed 0.1 millimeters apart. This distance is below ε_c, and thus when the cue ball strikes the rack, many collisions occur before the balls even begin to roll. Figure 5 show the high number of collisions that occurred during this simulation, especially at the point of the initial break. However, the simplicity of the collision model still permits fast simulation (see table 1). After the break, the collision rate stabilizes at roughly 3 kHz; these collisions are primarily between the balls and the table.

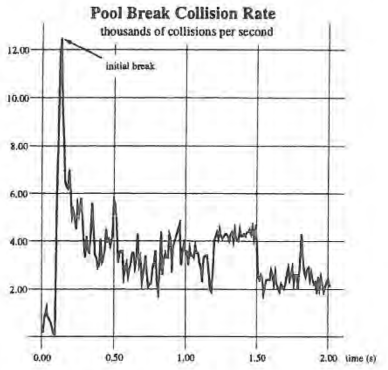


Figure 5: *Collision rate during a pool break.*

5.2 Block on ramp experiments

A good set of benchmarks for the physical accuracy of the collision model are "block on ramp" tests, involving a block sliding down a ramp with friction. We used a 20° ramp; the critical coefficient of friction at which the frictional force exactly resists the tangential component of gravity was $\mu_c = \tan 20° = 0.37$.

For the first test, the coefficient of friction was set to $\mu = 0.5 > \mu_c$, and the block was given an initial velocity down the ramp of 125 cm/sec. The theoretical and simulated velocities of the block down the ramp are shown in figure 6. The jaggedness of the simulated velocity curve is due

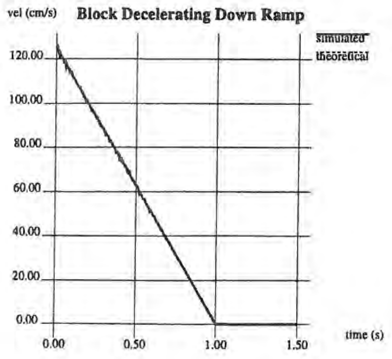


Figure 6: *Block velocity, $\mu = 0.5 > \mu_c$.*

to the discrete impulse train modeling the contact, however the average simulated velocity and the simulated position (figure 7) closely agree with theory.

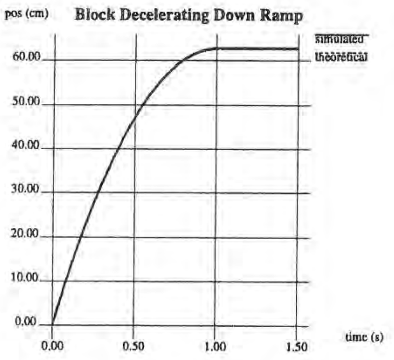


Figure 7: *Block position, $\mu = 0.5 > \mu_c$.*

In a second test, the coefficient of friction was lowered to $\mu = 0.25 < \mu_c$, with the block beginning at rest. The theoretical and simulated velocities and positions are shown in figures 8 and 9, respectively. There is close agreement between simulation and theory; the slopes of the two velocity curves are nearly identical, indicating that the impulse-based model predicts the correct frictional force on the block.

5.3 Measuring the strike pocket

We used our simulator to study the effect of a hooking ball on the width of the "strike pocket" in standard tenpin bowling. The best place for the ball to hit the pins is between the head pin and a second row pin; good bowlers throw a hooking ball, which hits the pins moving toward the center of the arrangement.

How does a hooking ball affect the chances of bowling a strike? The chaotic nature of the system makes a mathematical analysis nearly impossible, and it is also difficult to perform real experiments with sufficient control over conditions. In short, the problem is ideal for stochastic simulation. It is also a perfect application for impulse-based dynamics—the

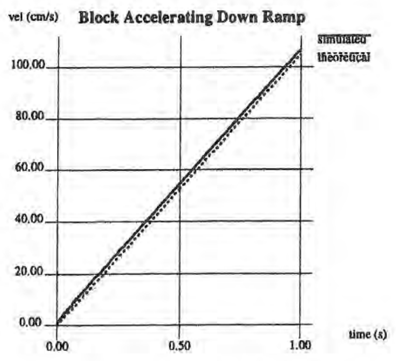


Figure 8: *Block velocity, $\mu = 0.25 < \mu_c$.*

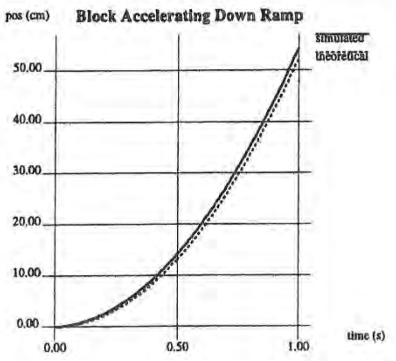


Figure 9: *Block position, $\mu = 0.25 < \mu_c$.*

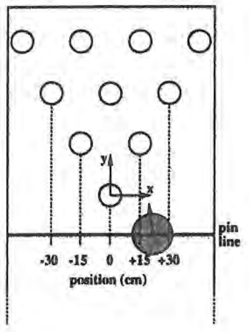


Figure 10: *Set up for the measurement of the strike pocket.*

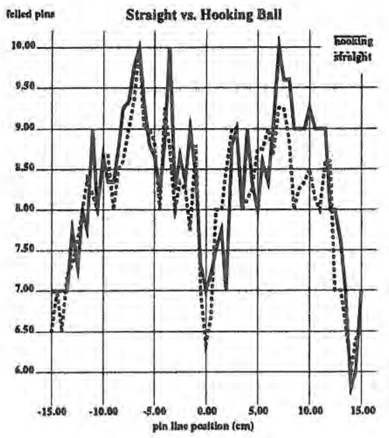


Figure 11: *Results from the strike pocket study.*

evolution is collision intensive, with many transient contacts between objects, and there is a gradual change in contact mode between the ball and the alley (bouncing to sliding to rolling). For our simulations, we used accurate physical dimensions for the alley, ball and pin sizes and masses, pin spacing, etc.; a slight approximation was made in the shape of the pins.

In the first batch of simulations, a straight ball was thrown down the alley by launching the ball with zero angular velocity, and a center of mass velocity in the $+y$ direction (see figure 10). We performed 320 trials, keeping the initial ball velocities constant, but varying the initial x-coordinate of the ball's center of mass over a 40 centimeter window, recording the number of felled pins for each trial. In a second batch of 320 trials, the initial ball velocity conditions were altered to produce a right-hander's hooking ball: angular velocity of -12 rad/s in the $+y$ direction and a linear velocity at an angle of $-2°$ from the y-axis.

Figure 11 shows the number of felled pins versus the ball position as it crossed the pin line (ordinates are averaged over 5 mm wide abscissa windows). The hooking ball is slightly better than the straight ball at most positions along the pin line, and is significantly better over a range between the head pin and rightmost second row pin (+6 to +12 cm on the pin line). This agrees with the accepted wisdom that a right-handed bowler's best strategy is to throw a hooking ball between these two pins. The plots also illustrate the dip in felled pins due to splits, when the ball hits the head pin dead on.

We could improve our model by more carefully specifying the shape of the pins and location of the ball's center of mass, which is not in general at the geometric center. However, this experiment demonstrates the feasibility and utility of using impulse-based dynamics for modeling a complex system and generating physically accurate results.

5.4 Other simulations

We briefly mention several other simulator problems we have tried, and summarize the execution time results for our simulator (see figure 12 for simulation snapshots).

Ball on spinning platter. This simulation involves a ball rolling on a disc that is spinning at high velocity. The example is interesting because of the nonholonomic constraint between the ball and disc, and in fact there are two classical models for this rolling contact which predict different behaviors! Experimental results show that the ball rolls in circles of gradually increasing radii, eventually rolling off the platter [11]. Our impulse-based simulator produces this result, demonstrating correct macroscopic behavior from the impulse-based contact model.

Block dropped on block. One block is dropped onto another, the former coming to rest on the latter.

Dominos. A line of seven dominos is set in motion by bumping the lead domino.

Chain of balls. Five balls the are placed next to each other in a straight line, and a rolling ball strikes the chain on one end. The momentum is transferred to the other end of the chain, launching the end ball.

Coins. Eight coins are tossed onto the same general area of a flat plate, and come to rest with some partially on top of others. This simulation is a good test case for all the contact modes: colliding, sliding, rolling, and resting.

Balls in dish. Seven balls are dropped into a shallow dish approximated by planar wedges. The balls come to rest in the physically accurate minimum energy configuration: one ball at the center of the dish, surrounded by the six other balls.

Table 1 gives the simulation times for all of the experiments. *Virtual time* is the length of time which passed in the simulation, *real time* is the actual time needed to compute the simulation[2], and *slowdown* is the ratio of the latter to the former (a 1.0 slowdown corresponds to real time simulation).

simulation	virtual time (s)	real time (s)	slow-down
pool break	3.0	95	32
dec. down ramp	1.5	41	27
acc. down ramp	1.0	23	23
bowling a strike	5.0	167	23.9
ball on platter	40	129	3.2
block drop	0.78	6.7	8.6
dominos	1.2	17	14
chain of balls	2.5	7.3	2.9
coins	3.6	50	14
balls in dish	7.8	95	12

Table 1: *Simulation times for experiments.*

6 Conclusions

We have described the impulse-based approach to dynamic simulation, and reported results from several simulation problems. Interactive simulation speeds have already been attained, and we believe real time simulation is ultimately possible. Also encouraging is the wide variety of physical systems that we have successfully simulated; no special tweaking was performed for any of the simulations we have described. One important efficiency point is that the impulse-based approach is highly parallelizable. Because there are no global constraints on the state of the system, the dynamic integration of an n body system is neatly decomposed into n small pieces. Such a decomposition is not possible when there are explicit constraints between the states of different bodies.

The issue of physical accuracy is also an important one to consider. Modeling a rock sitting on a table through a series of impulses seems at first questionable. However, we are not making the claim that the rock is actually experiencing microcollisions, only that by modeling the contact in this way, the correct macroscopic behavior is affected. Our simulator has produced physically plausible results for many problems. Furthermore, quantitative results withstand scrutiny when compared to theoretical models. More study is needed here, but the initial results are encouraging.

As stated previously, we do not intend impulse-based dynamics to be a complete replacement for constraint-based dynamics. A perfect application for the latter is the modeling of a hinge joint. In principle, one could model the joint in an impulse-based way, enforcing the hinge constraint through collisions between the hinge pin and sheath. However, the impulse-based approach is clearly the wrong tool for this natural constraint-based problem. We are currently adding a multibody capability to our simulator, in order to model linked rigid body structures. We are using a hybrid approach: constraint-based methods are used to enforce joint constraints, while impulse-based dynamics are used to model contact between bodies not connected via joints. We are optimistic that using the right tool for the right problem can greatly extend the frontier of dynamic simulation.

[2]Real times were computed by averaging over several trials. All simulations were performed on an *SGI Indigo I*.

References

[1] Baraff, David. Analytical Methods for Dynamic Simulation of Non-penetrating Rigid Bodies. Computer Graphics, 23(3):223–232, July 1989.

[2] Baraff, David. Curved Surfaces and Coherence for Non-penetrating Rigid Body Simulation. Computer Graphics, 24(4):19–28, August 1990.

[3] Baraff, David. Coping with Friction for Non-penetrating Rigid Body Simulation. Computer Graphics, 25(4):31–40, August 1991.

[4] Baraff, David. Issues in Computing Contact Forces for Non-penetrating Rigid Bodies. Algorithmica, 10:292–352, 1993.

[5] Barzel, Ronen and Barr, Alan H. A Modeling System Based on Dynamic Constraints. Computer Graphics, 22(4):179–188, August 1988.

[6] Bhatt, Vivek and Koechling, Jeff. Classifying Dynamic Behavior During Three Dimensional Frictional Rigid Body Impact. In International Conference on Robotics and Automation. IEEE, May 1994.

[7] Cremer, James F. and Stewart, A. James. The Architecture of Newton, a General-purpose Dynamics Simulator. In International Conference on Robotics and Automation, pages 1806–1811. IEEE, May 1989.

[8] Hahn, James K. Realistic Animation of Rigid Bodies. Computer Graphics, 22(4):299–308, August 1988.

[9] Hopcroft, John E. Electronic Prototyping. Computer, pages 55–57, March 1989.

[10] Keller, J. B. Impact with Friction. Journal of Applied Mechanics, 53, March 1986.

[11] Lewis, A. and M'Closkey, R. and Murray, Richard. Modelling Constraints and the Dynamics of a Rolling Ball on a Spinning Table. Technical report, California Institute of Technology, 1993. Preprint.

[12] Lin, Ming C. and Canny, John F. A Fast Algorithm for Incremental Distance Calculation. In International Conference on Robotics and Automation, pages 1008–1014. IEEE, May 1991.

[13] Mirtich, Brian and Canny, John. Impulse-based Dynamic Simulation. In K. Goldberg, D. Halperin, J.C. Latombe, and R. Wilson, editors, The Algorithmic Foundations of Robotics. A. K. Peters, Boston, MA, 1995. Proceedings from the workshop held in February, 1994.

[14] Moore, Matthew and Wilhelms, Jane. Collision Detection and Response for Computer Animation. Computer Graphics, 22(4):289–298, August 1988.

[15] Overmars, Mark. Point Location in Fat Subdivisions. Information Processing Letters, 44:261–265, 1992.

[16] Routh, Edward J. Elementary Rigid Dynamics. 1905.

[17] Stewart, A. James and Cremer, James F. Algorithmic Control of Walking. In International Conference on Robotics and Automation, pages 1598–1603. IEEE, May 1989.

[18] Wang, Yu and Mason, Matthew T. Modeling Impact Dynamics for Robotic Operations. In International Conference on Robotics and Automation, pages 678–685. IEEE, May 1987.

[19] Witkin, Andrew and Gleicher, Michael and Welch, William. Interactive Dynamics. Computer Graphics, 24(2):11–22, March 1990.

[20] Witkin, Andrew and Welch, William. Fast Animation and Control of Nonrigid Structures. Computer Graphics, 24(4):243–252, August 1990.

I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments

Jonathan D. Cohen Ming C. Lin * Dinesh Manocha Madhav Ponamgi
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
{cohenj,lin,manocha,ponamgi}@cs.unc.edu

ABSTRACT:
We present an exact and interactive collision detection system, I-COLLIDE, for large-scale environments. Such environments are characterized by the number of objects undergoing rigid motion and the complexity of the models. The algorithm does not assume the objects' motions can be expressed as a closed form function of time. The collision detection system is general and can be easily interfaced with a variety of applications. The algorithm uses a two-level approach based on pruning multiple-object pairs using bounding boxes and performing exact collision detection between selected pairs of polyhedral models. We demonstrate the performance of the system in walkthrough and simulation environments consisting of a large number of moving objects. In particular, the system takes less than 1/20 of a second to determine all the collisions and contacts in an environment consisting of more than a 1000 moving polytopes, each consisting of more than 50 faces on an HP-9000/750.

1 INTRODUCTION

Collision detection is a fundamental problem in computer animation, physically-based modeling, computer simulated environments and robotics. In these applications, an object's motion is constrained by collisions with other objects and by other dynamic constraints. The problem has been well studied in the literature. However, no good general collision detection algorithms and systems are known for interactive large-scale environments.

A large-scale virtual environment, like a walkthrough, creates a computer-generated world, filled with real and virtual objects. Such an environment should give the user a feeling of presence, which includes making the images of both the user and the surrounding objects feel solid. For example, the objects should not pass through each other, and things should move as expected when pushed, pulled or grasped. Such actions require accurate collision detection. However, there may be hundreds, even thousands of objects in the virtual world, so a brute-force approach that tests all possible pairs for collisions is not acceptable. Efficiency is critical in a virtual environment, otherwise its interactive nature is lost [24]. A fast and interactive collision detection algorithm is a fundamental component of a complex virtual environment.

The objective of collision detection is to report all geometric contacts between objects. If we know the positions and orientations of the objects in advance, we can solve collision detection as a function of time. However, this is not the case in virtual environments or other interactive applications. In fact, in a walkthrough environment, we usually do not have any information regarding the maximum velocity or acceleration, because the user may move with abrupt changes in direction and speed. Due to these unconstrained variables, collision detection is currently considered to be one of the major bottlenecks in building interactive simulated environments [20].

Main Contribution: We present a collision detection algorithm and system for interactive and exact collision detection in complex environments. In contrast to the previous work, we show that accurate, interactive performance can be attained in most environments if we use coherence to speed up pairwise interference tests and to reduce the actual number of these tests we perform. We are able to successfully trim the $O(n^2)$ possible interactions of n simultaneously moving objects to $O(n+m)$ where m is the number of objects *very close* to each other. In particular, two objects are very close, if their axis-aligned bounding boxes overlap. Our approach is flexible enough to handle dense environments without making assumptions about object velocity or acceleration. The system has been successfully applied to architectural walkthroughs and simulated environments and works well in practice.

The rest of the paper is organized as follows. In Section 2, we review some of the previous work in collision detection. Section 3 defines the concept of coherence and describes an exact pairwise collision detection algorithm which applies it. We describe our algorithm for collision detection between multiple objects in Section 4 and discuss its implementation in Sections 5 and 6. Section 7 presents our experimental results on walkthrough environments and simulations.

*Currently at NC A & T State University, Greensboro

2 PREVIOUS WORK

The problem of collision detection has been extensively studied in robotics, computational geometry, and computer graphics. The goal in robotics has been the planning of collision-free paths between obstacles [15]. This differs from virtual environments and physically-based simulations, where the motion is subject to dynamic constraints or external forces and cannot typically be expressed as a closed form function of time [1, 3, 11, 18, 20, 21].

At the same time, the emphasis in the computational geometry has been on theoretically efficient intersection detection algorithms [22]. Most of them are restricted to a static instance of the problem and are non-trivial to implement. For convex 3-polytopes [1] linear time algorithms based on linear programming and tracking closest points [10] have been proposed. More recently, temporal and geometric coherence have been used to devise algorithms based on checking local features of pairs of convex 3-polytopes [3, 17]. Alonso et al.[1] use bounding boxes and spatial partitioning to test all $O(n^2)$ pairs of arbitrary polyhedral objects.

Different methods have been proposed to overcome the bottleneck of $O(n^2)$ pairwise tests in an environment of n bodies. The simplest of these are based on spatial subdivision. The space is divided into cells of equal volume, and at each instance the objects are assigned to one or more cells. Collisions are checked between all object pairs belonging to a particular cell. This approach works well for sparse environments in which the objects are uniformly distributed through the space. Another approach operates directly on four-dimensional volumes swept out by object motion over time [4, 14].

None of these algorithms adequately address the issue of collision detection in a virtual environment which requires performance at interactive rates for thousands of pairwise tests. Hubbard has proposed a solution to address this problem by trading accuracy for speed [14]. In an early extension of their work, Lin and Canny [16] proposed a scheduling scheme to handle multiple moving objects. Dworkin and Zeltzer extended this work for a sparse model [7].

3 BACKGROUND

In this section, we highlight the importance of coherence in dynamic environments. We briefly review the algorithm for exact pairwise collision detection and present our multi-body collision detection scheme, both of which exploit coherence to achieve efficiency.

3.1 Temporal and Geometric Coherence

Temporal coherence is the property that the application state does not change significantly between time steps, or frames. The objects move only slightly from frame to frame. This slight movement of the objects translates into geometric coherence, because their geometry, defined by the vertex coordinates, changes minimally between frames. The underlying *assumption* is that the *time steps are small enough* that the objects to do not travel large distances between frames.

[1]We shall refer to a bounded d-dimensional polyhedral set as a convex d-polytope, or briefly polytope. In common parlance, "polyhedron" is used to denote the union of the boundary and of the interior in E^3.

3.2 Pairwise Collision Detection for Convex Polytopes

We briefly review the Lin-Canny collision detection algorithm which tracks closest points between pairs of convex polytopes [16, 17]. This algorithm is used at the lowest level of collision detection to determine the exact contact status between convex polytopes. The method maintains a pair of closest features for each convex polytope pair and calculates the Euclidean distance between the features to detect collisions. This approach can be used in a static environment, but is especially well-suited for dynamic environments in which objects move in a sequence of small, discrete steps.

The method takes advantage of coherence: the closest features change infrequently as the polytopes move along finely discretized paths. The algorithm runs in *expected constant time* if the polytopes are not moving swiftly. Even when a closest feature pair is changing rapidly, the algorithm takes only slightly longer (the running time is proportional to the number of feature pairs traversed, which is a function of the relative motion the polytopes undergo). The method for finding closest feature pairs is based on Voronoi regions. The algorithm starts with a candidate pair of features, one from each polytope, and checks whether the closest points lie on these features. Since the polytopes and their faces are convex, this is a local test involving only the neighboring features of the current candidate features. If either feature fails the test, the algorithm steps to a neighboring feature of one or both candidates, and tries again. With some simple preprocessing, the algorithm can guarantee that every feature has a constant number of neighboring features.

3.3 Penetration Detection for Convex Polytopes

The core of the collision detection algorithm is built using the properties of Voronoi regions of convex polytopes. The Voronoi regions form a partition of space outside the polytope. When polytopes interpenetrate, some features may not fall into any Voronoi regions. This can at times lead to cycling of feature pairs. To circumvent this problem, we partition the *interior space* of the convex polytopes. The partitioning does not have to form the exact internal Voronoi regions, because we are not interested in knowing the closest features between two interpenetrating polytopes, but only detecting such a case. So instead we use pseudo-Voronoi regions, obtained by joining each vertex of the polytope with the centroid of the polytope [21].

Given a partition of the exterior and the interior of the polytope, we walk from the external Voronoi regions into the pseudo-internal Voronoi regions when necessary. If either of the closest features falls into a pseudo-Voronoi region at the end of the walk, we know the objects are interpenetrating. Ensuring convergence as we walk through pseudo-internal Voronoi regions requires special case analysis and will be omitted here.

3.4 Extension to Non-Convex Objects

We extend the collision detection algorithm for convex polytopes to handle non-convex objects, such as articulated bodies, by using a hierarchical representation. In the hierarchical representation, the internal nodes can be convex or non-convex sub-parts, but *all* the leaf nodes are convex polytopes or features [21].

Beginning with the leaf nodes, we construct either a convex hull or other bounding volume and work up the tree, level by level, to the root. The bounding volume associated with each node is the bounding volume of the union of its children; the root's bounding volume encloses the whole hierarchy. For instance, a hand may have individual joints in the leaves, fingers in the internal nodes, and the entire hand in the root.

We test for collision between a pair of these hierarchical trees recursively. The collision detection algorithm first tests for collision between the two parent nodes. If there is no collision between the two parents, the algorithm returns the closest feature pair of their bounding volumes. If there is a collision, the algorithm expands their children and recursively proceeds down the tree to determine if a collision actually occurs. More details are given in [21].

4 MULTIPLE-OBJECT COLLISION DETECTION

Large-scale environments consist of stationary as well as moving objects. Let there be N moving objects and M stationary objects. Each of the N moving objects can collide with the other moving objects, as well as with the stationary ones. Keeping track of $\binom{N}{2} + NM$ pairs of objects at every time step can become time consuming as N and M get large. To achieve interactive rates, we must reduce this number before performing pairwise collision tests. The overall architecture of the multiple object collision detection algorithm is shown in Fig. 1.

Sorting is the key to our pruning approach. Each object is surrounded by a 3-dimensional bounding volume. We sort these bounding volumes in 3-space to determine which pairs are overlapping. We only need to perform exact pairwise collision tests on these remaining pairs.

However, it is not intuitively obvious how to sort objects in 3-space. We use a *dimension reduction* approach. If two bodies collide in a 3-dimensional space, their orthogonal projections onto the xy, yz, and xz-planes and x, y, and z-axes must overlap. Based on this observation, we choose axis-aligned bounding boxes as our bounding volumes. We efficiently project these bounding boxes onto a lower dimension, and perform our sort on these lower-dimensional structures.

This approach is quite different from the typical space partitioning approaches used to reduce the number of pairs. A space partitioning approach puts considerable effort into choosing good partition sizes. But there is *no* partition size that prunes out object pairs as ideally as testing for bounding box overlaps. Partitioning schemes may work well for environments where N is small compared to M, but object sorting works well whether N is small or large.

4.1 Bounding Volumes

Many collision detection algorithms have used bounding boxes, spheres, ellipses, etc. to rule out collisions between objects which are far apart. We use bounding box overlaps to trigger the *exact collision detection* algorithm.

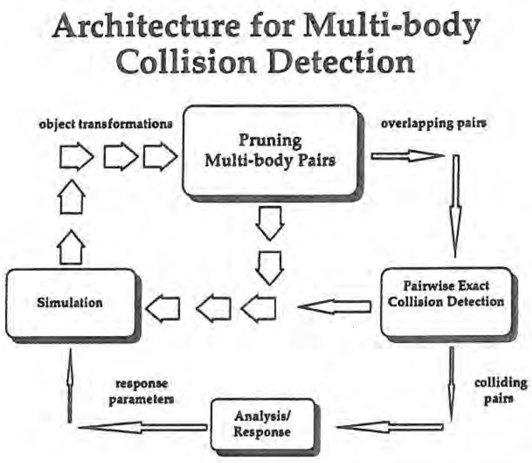


Figure 1: Architecture for Multiple Body Collision Detection Algorithm

We have considered two types of axis-aligned bounding boxes: fixed-size bounding cubes (fixed cubes) and dynamically-resized rectangular bounding boxes (dynamic boxes).

- **Fixed-Size Bounding Cubes:**

We compute the size of the fixed cube to be large enough to contain the object at *any* orientation. We define this axis-aligned cube by a *center* and a *radius*. Fixed cubes are easy to recompute as objects move, making them well-suited to dynamic environments. If an object is nearly spherical the fixed cube fits it well.

As preprocessing steps we calculate the center and radius of the fixed cube. At each time step as the object moves, we recompute the cube as follows:

1. Transform the center using one vector-matrix multiplication.
2. Compute the minimum and maximum x, y, and z-coordinates by subtracting and adding the radius from the coordinates of the center.

Step 1 involves only one vector-matrix multiplication. Step 2 needs six arithmetic operations (3 additions and 3 subtractions).

- **Dynamically Rectangular Bounding Boxes:**

We compute the size of the rectangular bounding box to be the tightest axis-aligned box containing the object at a *particular* orientation. It is defined by its minimum and maximum x, y, and z-coordinates (for a convex object, these must correspond to coordinates of up to 6 of its vertices). As an object moves, we must recompute its minima and maxima, taking into account the object's orientation. For oblong objects rectangular boxes fit better than cubes, resulting in fewer overlaps. This is advantageous as long as few of the objects are moving, as in a

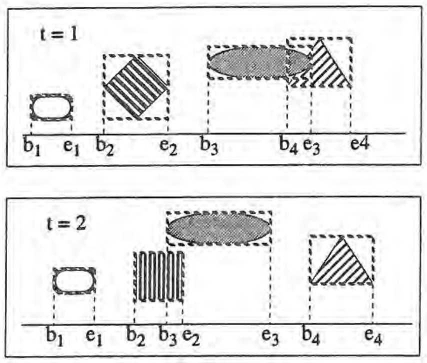


Figure 2: Bounding Box Behavior

walkthrough environment. In such an environment, the savings gained by the reduced number of pairwise collision detection tests outweigh the cost of computing the dynamically-resized boxes.

As a precomputation, we compute each object's initial minima and maxima along each axis. It is assumed that the objects are convex. For non-convex polyhedral models, the following algorithm is applied to their convex hulls. As an object moves, we recompute its minima and maxima at each time step as follows:

1. Check to see if the current minimum (or maximum) vertex for the x, y, or z-coordinate still has the smallest (or largest) value in comparison to its neighboring vertices. If so we are finished.

2. Update the vertex for that extremum by replacing it with the neighboring vertex with the smallest (or largest) value of all neighboring vertices. Repeat the entire process as necessary.

This algorithm recomputes the bounding boxes at an expected constant rate. Once again, we are exploiting the temporal and geometric coherence, in addition to the locality of convex polytopes.

We do not transform all the vertices as the objects undergo motion. As we are updating the bounding boxes new positions are computed for current vertices using matrix-vector multiplications. We can optimize this approach by realizing that we are only interested in *one* coordinate value of each extremal vertex, say the x coordinate while updating the minimum or maximum value along the x-axis. Therefore, there is no need to transform the other than coordinates in order to compare neighboring vertices. This reduces the number of arithmetic operations by two-thirds.

4.2 One-Dimensional Sweep and Prune

The one-dimensional sweep and prune algorithm begins by projecting each three-dimensional bounding box onto the x, y, and z axes. Because the bounding boxes are axis-aligned, projecting them onto the coordinate axes results in intervals (see Fig. 2). We are interested in overlaps among these intervals, because a pair of bounding boxes can overlap *if and only if* their intervals overlap in all three dimensions.

We construct three lists, one for each dimension. Each list contains the values of the endpoints of the intervals corresponding to that dimension. By sorting these lists, we can determine which intervals overlap. In the general case, such a sort would take $O(n \log n)$ time, where n is the number of objects. We can reduce this time bound by keeping the sorted lists from the previous frame, changing only the values of the interval endpoints. In environments where the objects make relatively small movements between frames, the lists will be nearly sorted, so we can sort in expected $O(n)$ time, as shown in [19, 3]. *Insertion sort* works well for previously sorted lists.

In addition to sorting, we need to keep track of changes in overlap status of interval pairs (i.e. from overlapping in the last time step to non-overlapping in the current time step, and vice-versa). This can be done in $O(n + e_x + e_y + e_z)$ time, where $e_x, e_y,$ and e_z are the number of exchanges along the $x, y,$ and z-axes. This also runs in expected linear time due to coherence, but in the worst case $e_x, e_y,$ and e_z can each be $O(n^2)$ with an extremely small constant.

Our method is suitable for dynamic environments where coherence is preserved. In computational geometry literature several algorithms exist that solve the static version of determining 3-D bounding box overlaps in $O(n \log^2 n + s)$ time, where s is the number of pairwise overlaps [12, 13]. We have reduced this to $O(n + s)$ by using coherence.

4.3 Two-Dimensional Intersection Tests

The two-dimensional intersection algorithm begins by projecting each three-dimensional axis-aligned bounding box onto any two of the x-y, x-z, and y-z planes. Each of these projections is a rectangle in 2-space. Typically there are fewer overlaps of these 2-D rectangles than of the 1-D intervals used by the sweep and prune technique. This results in fewer swaps as the objects move. In situations where the projections onto one-dimension result in densely clustered intervals, the two-dimensional technique is more efficient. The interval tree is a common data structure for performing such two-dimensional range queries [22].

Each query of an interval intersection takes $O(\log n + k)$ time where k is the number of reported intersections and n is the number of intervals. Therefore, reporting intersections among n rectangles can be done in $O(n \log n + K)$ where K is the total number of intersecting rectangles [8].

4.4 Alternatives to Dimension Reduction

There are many different methods for reducing the number of pairwise tests, such as binary space partitioning (BSP) trees [23], octrees, etc.

Several practical and efficient algorithms are based on uniform space division. Divide space into unit cells (or volumes) and place each object in some cell(s). To check for collisions, examine the cell(s) occupied by each object to verify if the cell(s) is(are) shared by other objects. Choosing a near-optimal cell size is difficult, and failing to do so results in large memory usage and computational inefficiency.

5 IMPLEMENTATION

In this section we describe the implementation details of I-COLLIDE based on the Sweep and Prune algorithm, the exact collision detection algorithm, the multi-body simulation, and their applications to walkthrough and simulations.

5.1 Sweep and Prune

As described earlier, the Sweep and Prune algorithm reduces the number of pairwise collision tests by eliminating polytope pairs that are far apart. It involves three steps: calculating bounding boxes, sorting the minimum and maximum coordinates of the bounding boxes as the algorithm sweeps through each list, and determining which bounding boxes overlap. As it turns out, we do the second and third steps simultaneously.

Each bounding box consists of a minimum and a maximum coordinate value for each dimension: x, y, and z. These minima and maxima are maintained in three separate lists, one for each dimension. We sort each list of coordinate values using insertion sort, while maintaining an overlap status for each bounding box pair. The overlap status consists of a boolean flag for each dimension. Whenever all three of these flags are set, the bounding boxes of the polytope pair overlap. These flags are only modified when insertion sort performs a swap. We decide whether or not to toggle a flag based on whether the coordinate values both refer to bounding box minima, both refer to bounding box maxima, or one refers to a bounding box minimum and the other a maximum.

When a flag is toggled, the overlap status indicates one of three situations:

1. All three dimensions of this bounding box pair now overlap. In this case, we add the corresponding polytope pair to a list of active pairs.

2. This bounding box pair overlapped at the previous time step. In this case, we remove the corresponding polytope pair from the active list.

3. This bounding box pair did not overlap at the previous time step and does not overlap at the current time step. In this case, we do nothing.

When sorting is completed for this time step, the active pair list contains all the polytope pairs whose bounding boxes currently overlap. We pass this active pair list to the exact collision detection routine to find the closest features of all these polytope pairs and determine which, if any, of them are colliding.

5.2 Exact collision detection

The collision detection routine processes each polytope pair in the active list. The first time a polytope pair is considered, we select a random feature from each polytope; otherwise, we use the previous closest feature pair as a starting point. This previous closest feature pair may not be a good guess when the polytope pair has just become active. Dworkin and Zeltzer [7] suggest precomputing a lookup table for each polytope to help find better starting guesses.

5.3 Multi-body Simulation

The multi-body simulation is an application we developed to test the I-COLLIDE system. It represents a general, non-restricted environment in which objects move in an arbitrary fashion resulting in collisions with simple impulse responses.

While we can load any convex polytopes into the simulation, we typically use those generated by the tessellation of random points on a sphere. Unless the number of vertices is large, the resulting polytopes are not spherical in appearance; they range from oblong to fat. The simulation parameters of the polytopes were their number, their complexity measured as the number of faces, their rotational velocity, their translational velocity, the density of their environment measured as the ratio of polytope volume to environment volume, and the bounding volume method used for the Sweep and Prune (fixed-size or dynamically-resized boxes).

The simulation begins by placing the polytopes at random positions and orientations. At each time step, the positions and orientations are updated using the translational and rotational velocities (since the detection routines make no use of pre-defined path, the polytopes' paths could just as easily be randomized at each time step). The simulation then calls the I-COLLIDE system and receives a list of colliding polytope pairs. It exchanges the translational velocities of these pairs to simulate an elastic reaction. Objects also rebound off the walls of the constraining volume.

We use this simulation to test the functionality and speed of the detection algorithm. In addition, we are able to visually display some of the key features. For example, the bounding boxes of the polytopes can be rendered at each time step. When the bounding boxes of a polytope pair overlap, we can render a line connecting the closest features of this polytope. It is also possible to show all pairs of closest features at each time step. These visual aids have proven to be useful in indicating actual collisions and additional geometric information for algorithmic study and analysis. See Frame 1 at the end for an example of the simulation.

5.4 Walkthrough

The walkthrough is a head-mounted display application that involves a large number of polytopes depicting a realistic scene. The integration of our library into such an environment demonstrates that an interactive environment can use our collision detection library without affecting the application's real-time performance.

The walkthrough creates a virtual environment (our video shows a kitchen and a porch). The user travels through this environment, interacting with the polytopes: picking up virtual objects, changing their scale, and moving them around. Whenever the user's hand collides with the polytopes in the environment, the walkthrough provides feedback by making colliding bodies appear red.

We have incorporated the collision detection library routines into the walkthrough application. The scene is composed of polytopes, most of which are stationary. The user's hand, composed of several convex polytopes, moves through this complex environment, modifying other polytopes in the environment. Frames 2-4 show a sequence of shots from a kitchen walkthrough environment. The pictures show images as seen by the left eye. Frames 5-6 show the user in a porch walkthrough.

6 SYSTEM ISSUES

To use I-COLLIDE, the application first loads a library of polytopes. The file format we use is fairly simple. It is straightforward to convert polytope data from some other format (perhaps the output of some 3D modelling package) to this minimal format for I-COLLIDE. After loading the polytopes, the application then chooses some polytope pairs to activate for collision detection. This set of active pairs is fully configurable between collision passes. Inside the application loop, the application informs I-COLLIDE of the world transformation for each polytope as it moves around. At any point, the application may call the collision test routine. I-COLLIDE returns a list of all the colliding pairs, including a pair of colliding features for each. The application then responds to these collisions in some appropriate way.

6.1 Space Issues

For each pair of objects, I-COLLIDE maintains a structure that contains the bounding box overlap status and the closest feature pair between the objects. These structures conceptually form an upper-triangular $O(n^2)$ matrix. We access an entry in O(1) time by using the object id numbers as (row, column) entries. If only a few pairs of objects are interacting, then the $O(n^2)$ can be reduced at the expense of slightly larger access time. For example, we can traverse a sparse matrix list to access an entry.

6.2 Geometric Robustness

In practice there are several types of degeneracies or errors that can occur in the convex polytope models: duplicate vertices, extraneous vertices, backfacing polygons, tracking error, non-planar faces, non-convex faces, non-convex polytopes, disconnected faces, etc. We have written a pre-processor to scan for common degeneracies and correct them when possible.

6.3 Numerical Issues

Numerical robustness is an important issue in the exact collision detection code. There are many special case geometrical tests in this module, and it is difficult to ensure that the algorithm will not get into a cycle due to degenerate overlap. We deal with this by performing all of our feature tests to some tolerance. Without such a tolerance, floating point errors might allow some of the feature tests to cycle infinitely. We have not observed this in practice so far, and have been careful to make the tests stable in the presence of small errors.

The multi-body sweep and prune code is also designed to resist small numerical errors. The bounding box of each polytope is extended by a small epsilon [2] in each direction. In addition to insulating the overlap tests from errors, this precaution also helps give the exact collision detection test a chance of being activated before the objects are actually penetrating.

[2]This quantity is a function of velocity between the object pairs.

6.4 Generality

While the multi-body pruning code works well with the exact collision detection routine, it functions independently of the underlying collision detection routine. This second level collision routine might or might not be exact, and it certainly need not be limited to handling convex polytopes.

7 PERFORMANCE ANALYSIS

We measured the performance of the collision detection algorithm using the multi-body simulation as a benchmark. We profiled the entire application and tabulated the CPU time of only the relevant detection routines. All of these tests were run on an HP-9000/750. The main routines involved in collision detection are those that update the bounding boxes, sort the bounding boxes, and perform exact collision detection on overlapping bounding boxes. As described in the implementation section we use two different types of bounding boxes. Using fixed cubes as bounding boxes resulted in low collision time for the parameter ranges we tested.

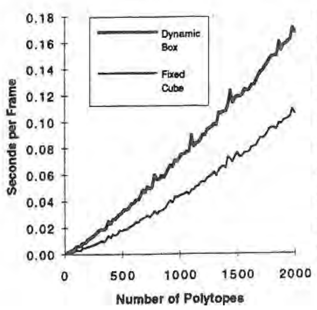
In each of the first four graphs, we plot two lines. The bold line displays the performance of using dynamically-resized bounding boxes whereas the other line shows the performance of using fixed-size cubes. All five graphs refer to "seconds per frame", where a frame is one step of the simulation, involving one iteration of collision detection without rendering time. Each graph was produced with the following parameters, by holding all but one constant.

- *Number of polytopes.* The default value is a 1000 polytopes.
- *Complexity of polytopes*, which we define as the number of faces. The default value is 36 faces.
- *Rotational velocity*, which we define as the number of degrees the object rotates about an axis passing through its centroid. The default value is 10 degrees.
- *Translational velocity*, which we define in relation to the object's size. We estimate a radius for the object, and define the velocity as the percentage of its radius the object travels each frame. The default value is 10%.
- *Density*, which we define as the percentage of the environment volume the polytopes occupy. The default value is 1.0%.

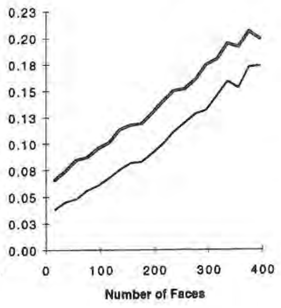
In the graphs, the timing results do not include computing each polytope's transformation matrix, rendering times, and of course any minor initialization cost. We ignored these costs, because we wanted to measure the cost of collision detection alone.

Graph 1 shows how the number of seconds per frame scales with an increasing number of polytopes. We took 100 uniformly sampled data points from 20 to 2000 polytopes. The fixed and dynamic bounding box methods scale nearly linearly with a small higher-order term. The dynamic bounding box method results in a slightly larger non-linear term because the resizing of bounding boxes

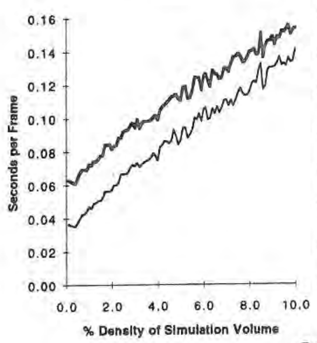
Graph 1



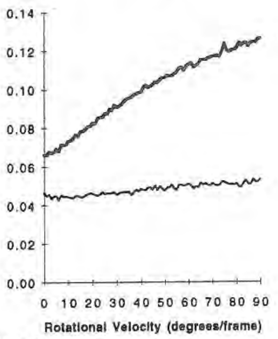
Graph 2



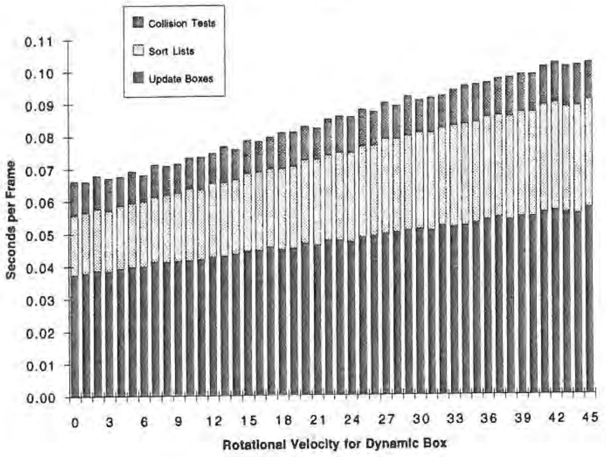
Graph 3



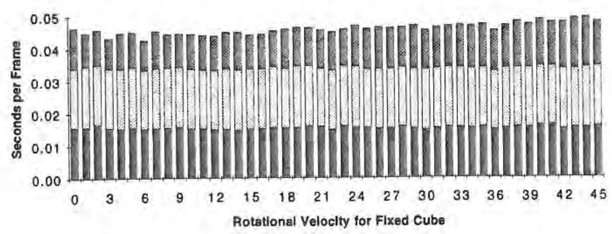
Graph 4



Graph 5



Graph 6



causes more swaps during sorting. This is explained further in our discussion of Graph 5. The seconds per frame numbers in Graph 1 compare very favorably with the work of Dworkin and Zeltzer [7] as well as those of Hubbard [14]. For a 1000 polytopes in our simulation, our collision time results in **23 frames per second** using the fixed bounding cubes.

Graph 2 shows how the number faces affects the collision time. We took 20 uniformly sampled data points. For the dynamic bounding box method, increasing the model complexity increases the time to update the bounding boxes because finding the minimum and maximum values requires walking a longer path around the polytope. Surprisingly, the time to sort the bounding boxes decreases with number of faces, because the polytopes become more spherical and fat. As the polytopes become more spherical and fat, the bounding box dimensions change less as the polytopes rotate, so fewer swaps are need in the sweeping step. For the fixed bounding cube, the time to update the bounding boxes and to sort them is almost constant.

Graph 3 shows the effect of changes in the density of the simulation volume. For both bounding box methods, increasing the density of polytope volume to simulation volume results in a larger sort time and more collisions. The number of collisions scales linearly with the density of the simulation volume. As the graph shows, the overall collision time scales well with the increases in density.

Graphs 4 through 6 show the effect of rotational velocity on the overall collision time. The slope of the line for the dynamic bounding box method is much larger than that of the fixed cube method. There are two reasons for this difference. The first reason is that the increase in rotational velocity increases the time required to update the dynamic bounding boxes. When we walk from the old maxima and minima to find the new ones, we need to traverse more features.

The second reason is the larger number of swapped minima and maxima in the three sorted lists. Although the three-dimensional volume of the simulation is fairly sparse, each one-dimensional view of this volume is much more dense, with many bounding box intervals overlapping. As the boxes grow and shrink, they cause many swaps in these one-dimensional lists. And as the rotational velocity increases, the boxes change size more rapidly.

Graph 6 clearly shows the advantages of the static box method. Both the update bounding box time and sort lists time are *almost* constant as the rotational velocity increases.

All of our tests show *exact* collision detection in demanding environments can be achieved without incurring expensive time penalties. The architectural walkthrough models showed no perceptible performance degradation when collision detection was added (as in Frame 2 to 5).

8 CONCLUSION

Collision detection has been considered a major bottleneck in computer-simulated environments. By making use of geometric and temporal coherence, our algorithm and system detects collisions more efficiently and effectively than earlier algorithms. Under many circumstances our system produces collision frame rates over 20 hertz for environments with over a 1000 moving complex polytopes. Our walkthrough experiments showed no degradation of frame rates when collision detection was added. We are currently working on incorporating general polyhedral and spline models into our system and extending these algorithms to deformable models.

9 ACKNOWLEDGEMENTS

We are grateful to John Canny and David Baraff for productive discussions and to Brian Mirtich for his help in implementation of the convex polytope pair algorithm. The kitchen and porch models used in the walkthrough applications were designed by the UNC Walkthrough group, headed by Fred Brooks. This work was supported in part by DARPA ISTO order A410, NSF grant MIP-9306208, NSF grant CCR-9319957, ARPA contract DABT63-93-C-0048, ONR contract N00014-94-1-0738 and NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization, NSF Prime Contract 8920219.

References

[1] A.Garica-Alonso, N.Serrano, and J.Flaquer. Solving the collision detection problem. *IEEE Computer Graphics and Applications*, 13(3):36–43, 1994.

[2] D. Baraff. Curved surfaces and coherence for non-penetrating rigid body simulation. *ACM Computer Graphics*, 24(4):19–28, 1990.

[3] D. Baraff. *Dynamic simulation of non-penetrating rigid body simulation.* PhD thesis, Cornell University, 1992.

[4] S. Cameron. Collision detection by four-dimensional intersection testing. *Proceedings of International Conference on Robotics and Automation*, pages pp. 291–302, 1990.

[5] S. Cameron. Approximation hierarchies and s-bounds. In *Proceedings. Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 129–137, Austin, TX, 1991.

[6] J. Cohen, M. Lin, D. Manocha, and K. Ponamgi. Interactive and exact collision detection for large-scaled environments. Technical Report TR94-005, Department of Computer Science, University of North Carolina, 1994.

[7] P. Dworkin and D. Zeltzer. A new model for efficient dynamics simulation. *Proceedings Eurographics workshop on animation and simulation*, pages 175–184, 1993.

[8] H. Edelsbrunner. A new approach to rectangle intersections, Part I. *Internat. J. Comput. Math.*, 13:209–219, 1983.

[9] J. Snyder et. al. Interval methods for multi-point collisions between time dependent curved surfaces. In *Proceedings of ACM Siggraph*, pages 321–334, 1993.

[10] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between objects in three-dimensional space. *IEEE J. Robotics and Automation*, vol RA-4:pp. 193–203, 1988.

[11] J. K. Hahn. Realistic animation of rigid bodies. *Computer Graphics*, 22(4):pp. 299–308, 1988.

[12] J.E. Hopcroft, J.T. Schwartz, and M. Sharir. Efficient detection of intersections among spheres. *The International Journal of Robotics Research*, 2(4):77–80, 1983.

[13] H.Six and D.Wood. Counting and reporting intersections of *D*-ranges. *IEEE Transactions on Computers*, pages 46–55, 1982.

[14] P. M. Hubbard. Interactive collision detection. In *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality*, October 1993.

[15] J.C. Latombe. *Robot Motion Planning.* Kluwer Academic Publishers, 1991.

[16] M. Lin and J. Canny. Efficient collision detection for animation. In *Proceedings of the Third Eurographics Workshop on Animation and Simulation*, Cambridge, England, 1991.

[17] M.C. Lin. *Efficient Collision Detection for Animation and Robotics.* PhD thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, December 1993.

[18] M. Moore and J. Wilhelms. Collision detection and response for computer animation. *Computer Graphics*, 22(4):289–298, 1988.

[19] M.Shamos and D.Hoey. Geometric intersection problems. *Proc. 17th An. IEEE Symp. Found. on Comput. Science*, pages 208–215, 1976.

[20] A. Pentland. Computational complexity versus simulated environment. *Computer Graphics*, 22(2):185–192, 1990.

[21] M. Ponamgi, D. Manocha, and M. Lin. Incremental algorithms for collision detection between solid models. Technical Report TR94-061, Department of Computer Science, University of North Carolina, Chapel Hill, 1994.

[22] F.P. Preparata and M. I. Shamos. *Computational Geometry.* Springer-Verlag, New York, 1985.

[23] W.Thibault and B.Naylor. Set operations on polyhedra using binary space partitioning trees. *ACM Computer Graphics*, 4, 1987.

[24] D. Zeltzer. Autonomy, interaction and presence. *Presence*, 1(1):127, 1992.

Author Index

Page numbers in *italics* indicate color plates.