© Copyright by Edward Henry Bachta, 2002

MANIPULATING IMPLICIT SURFACES IN THE CAVE™ USING PARTICLE SYSTEMS

BY

EDWARD HENRY BACHTA

B.S., University of Illinois at Urbana-Champaign, 1999

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Masters of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2002

Urbana, Illinois

To My Friends and Family

Acknowledgements

I have many people to thank for their support of my work on this project. My advisors, Dr. M. Pauline Baker and Dr. John C. Hart, provided me with invaluable knowledge, advice, and freedom of creativity, without which I could not have progressed. I also need to thank the members of the Visualization and Virtual Environments group at NCSA, many of whom helped to teach me everything I know about the CAVE, and especially Paul Rajlich, who introduced me to the team. Every one of the students who helped to develop components of the implicit surface library also deserves my thanks for their hard work. My thanks goes out to all of my new friends, as well as the old friends and family who have supported me over the years.

Table of Contents

1 Introduction 1

2 Related Work....................2

3 Implicit Surfaces....................3

4 The Advanced Surface Library 6
- 4.1 Supporting Libraries....................6
  - 4.1.1 Mathematics Libraries....................6
  - 4.1.2 The C++ Standard Template Library....................7
  - 4.1.3 The Open Graphics Library....................7
- 4.2 Surface Library Overview 8
- 4.3 The Particle System 9
  - 4.3.1 Preliminary Design 11
  - 4.3.2 Object-Oriented Design....................12
  - 4.3.3 Component-Based Design....................16

5 The CAVE Automatic Virtual Environment....................19
- 5.1 Input Devices 20
- 5.2 Output Devices 22
- 5.3 CAVElib and FreeVR 24

6 Applications 27
- 6.1 Implicit Viewer 27
- 6.2 Immersive Surface Sculptor 28

7 Future Work 34

References 35

1 Introduction

The goal of this thesis project was to develop an immersive environment for studying implicit surfaces. Recently, work has been done in our computer graphics group at UIUC to provide the research community with a library enabling further study of implicit surfaces. [15] My project has taken that work further to provide an application that enables a new method of interaction with implicit surfaces. Using the CAVE, surfaces can not only be viewed in an immersive fashion, but by tracking the hands the application provides an intuitive, hands-on mode of control.

This work began in the Advanced Surface Modeling course offered by Prof. John C. Hart in the fall semester of 2000. I contributed source code implementing some of the underlying geometric components of the library, as well as a particle system designed to visualize implicit surfaces. At the end of that semester, we had a rough implementation of our basic components, including visualization via polygonization and particles. Work then continued on the project and I developed a new version of the particle system enabling surface manipulation. I have now successfully developed an application utilizing our library that allows the user to intuitively "sculpt" implicit surfaces in a virtual environment. This paper will present some related work, implicit surfaces, the advanced surface library developed by the group (in particular the particle system that it employs), the CAVE, and the progression of application development that led to the conclusion of the project.

2 Related Work

The metaphor of sculpting has been used in many surface modeling projects. It is especially powerful when using haptic devices that provide force-feedback, creating the impression that the virtual object is resisting the user's motions, similar to clay. The developers of [19] utilize such a device to manipulate objects using a physics-based modeling technique with subdivision solids. In [22], a haptic device provides feedback for a user probing objects defined by implicit functions. Even three dimensional input devices that do not give force-feedback still offer an improvement in interaction over the standard mouse, and are utilized in [20] to sculpt objects using a free-form spline method.

Virtual environments have been used in a wide variety of disciplines to provide effective interfaces. One example very closely related to sculpting is the work described in [21], which uses the CAVE to provide a three dimensional painting canvas. The developers even went as far as constructing physical paintbrush and paint can props to make the interface more intuitive.

My work brings implicit surface modeling into a visually immersive environment, enabling new methods of representation and exploration. Similar to the CAVE painting project, this research could be taken further in the field of artistry as well.

3 Implicit Surfaces

Implicit surfaces are two-dimensional, geometric shapes that exist in three-dimensional space; they are defined according to a particular mathematical form. [1] To gain an understanding of implicit functions it is useful to start from the two dimensional domain, and then proceed to three dimensions.

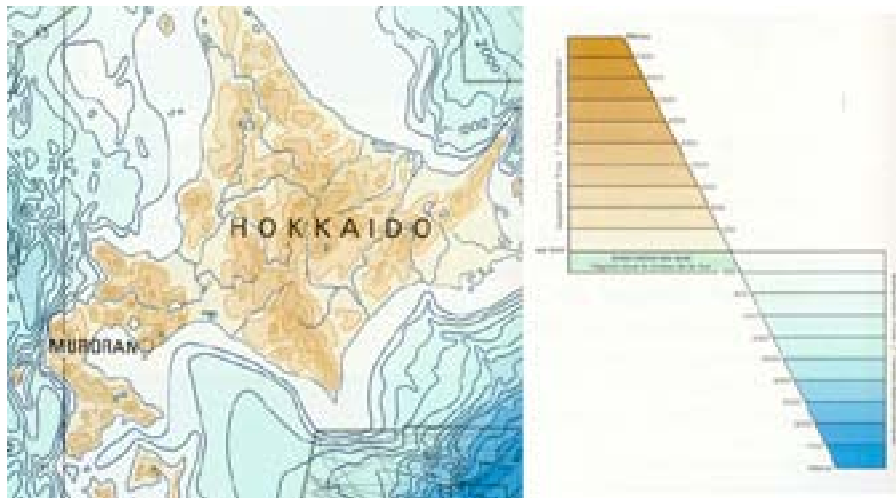

Figure 1: Bathymetric chart of Hokkaido, Japan

Let us first consider a topological map of hilly terrain. The chart above depicts the elevation at different locations on and around Hokkaido, the northernmost of the main islands of Japan. [18] An island such as Hokkaido is actually the top of an underwater landmass. Areas in tan tones are above sea level, while areas in blue tones are below, as indicated in the legend. Darker color indicates a greater difference from sea level. Implicit functions define values at all points in space much like this chart represents elevation at all coordinates on the map. The coastline is represented by a contour indicating elevation zero, which exists between the areas shaded in the lightest tan color and the areas colored white. This is analogous to the zero set of a two-

dimensional implicit function, which is represented as an implicit contour and exists between regions of positive values and regions of negative values.

Turning to mathematics, in the two dimensional domain we can consider the general equation for a circle of radius r at coordinates (x_c, y_c): $(x - x_c)^2 + (y - y_c)^2 = r^2$. This equation has variables x and y spanning the domain, and the parameters x_c, y_c, and r defining the properties of the circle. The implicit function describing a circle, $F(x,y) = (x - x_c)^2 + (y - y_c)^2 - r^2 = 0$, also operates on these variables. The parameters are regarded as constants for any particular shape. Using the logic of algebra, it can be seen that the implicit function will evaluate to zero only at points that are on the circle. At points within the circle, F(x,y) will evaluate to negative numbers (much like areas below sea level in an elevation map), with the minimum at the center of the circle. And at points outside the circle the function value will be positive (like areas above sea level), with no maximum.

In the three dimensional domain we can consider the implicit equation for a sphere at the origin: $F(x,y,z) = x^2 + y^2 + z^2 - r^2 = 0$. For a given value of r, this equation will evaluate to zero exactly at the coordinates of points on the surface of a sphere of radius r. At points inside a sphere of radius r the function will have negative values, and outside it will have positive values. In this sense implicit surfaces are much like iso-surfaces, which are commonly used in scientific visualization to determine where particular values occur in a dataset that changes continuously over the domain. In fact, if the values of the implicit function were regarded as data, the iso-surface with iso-value 0 would describe the same surface.

If our function is continuous and differentiable over the three-dimensional domain, we can determine surface normals [1], enabling us to more effectively represent the surface. For example, ray-tracing methods can be used to create a high-quality shaded image of the implicit

surface. There are also other less computationally intensive methods of visualization that make use of surface normals, including the oriented particle method that will be discussed later.

Performing an inside-outside test on an implicit surface requires only as much time as it takes to compute its corresponding function. Computational methods that operate on the gradient of the function can be used to find the surface if starting at a random seed point. There are also techniques that can be used to detect or even prevent major topological changes as a surface is deformed. These are just a few useful features that help to explain why the implicit function has become an interesting shape representation for modeling applications.

4 The Advanced Surface Library

During the fall semester of 2000, the homework and projects in Professor John C. Hart's Advanced Surface Modeling course involved developing the components of the Advanced Surface Library. The motivation behind the work was to provide a basis for further investigation of implicit surfaces, both in studying their aspects and in researching new methods of representation. In this section I will provide a description of the existing libraries used, an overview of the surface library, and a detailed report of the work I did on the particle system.

4.1 Supporting Libraries

In the development of the surface library, we made use of a few existing packages. We decided it would be beneficial to use existing implementations of fundamental mathematical primitives and operations, standard data structures, and graphics rendering.

4.1.1 Mathematics Libraries

Many components of our system operate on three-dimensional vectors and systems of equations. One of the most useful libraries we took advantage of is the Graphics Gems vector library. [2] It provides classes for three and four element vectors and the matrices necessary to transform them. Also included are a collection of functions providing standard vector and matrix operations.

There are also a few components in our library that require the use of mathematical solvers. In these instances we make use of the Template Numerical Toolkit (TNT) library. [3] It is designed to assist in scientific computing applications by providing template vectors, matrices, and multidimensional arrays. This allows the programmer to use the efficient computational algorithms provided by TNT on data types specific to the application.

4.1.2 The C++ Standard Template Library

The standardization of C++ was started in 1989 and finished at the end of 1997. [8] A standard library was designed to go along with this standardization of the popular C++ library. It provides components for I/O, strings, common data structures and algorithms, computation, and internationalization.

The part of the STL we most heavily used is probably the Vector class. Vector is the STL container class for dynamic arrays, and provides an interface allowing random access to array elements. For example, a collection of particles might be held in a vector called 'particles' with each element containing a single particle. To access the nth particle, simply stating 'particles[n]' would suffice.

We also made use of the Valarray class. A valarray is similar to a vector, however its size cannot be changed without reallocating the entire structure. They are very useful when mathematical operations must be applied uniformly to all elements. In such a case, operators can be applied to valarray variables as if they were scalar values.

4.1.3 The Open Graphics Library

OpenGL is the most widely adopted computer graphics standard. [4] Because of its wide acceptance, we decided to implement any graphical components using this standard. OpenGL provides an interface to the two- and three-dimensional rendering capabilities of graphics hardware. It is designed as a streamlined, hardware-independent interface and makes use of performance optimizations such as pipelining. [14] The library does not contain any features to handle windowing tasks or user input, so most of our applications also make use of GLUT, the OpenGL Utility Toolkit. GLUT is a platform independent tool that handles the operations required to run an application in a simple window.

4.2 Surface Library Overview

The backbone of the surface library is the Implicit class and those classes derived from it. There are two basic types of derived classes, the primitives and the operators. Primitive classes implement implicit functions. Algebraic primitives implement functions that are defined by algebraic equations of varying degree, while geometric primitives implement functions of distance. Operators modify the functions of other classes to which they are attached. We have implemented binary, unary, and n-ary operators. These components of the library compose a derivation hierarchy from the Implicit base class.

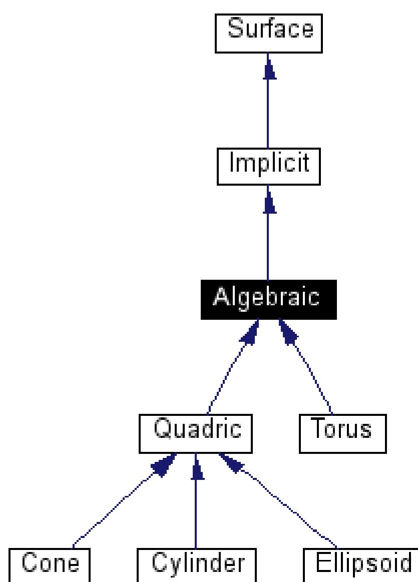

Figure 2: Inheritance graph for the Algebraic branch of Implicit

An object is created from these classes by using a combination of primitives and operators. Every class derived from Implicit provides the same basic interface that allows for interrogation of function values, gradients, and curvature information at arbitrary points in space. When an operator is attached to an object, it retains a reference that can be used to interrogate the sub-component when its functions are called. The composition of an object can be specified in a file.

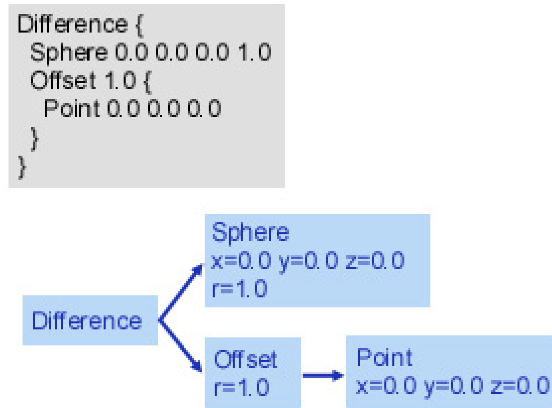

Figure 3: Object composition

Assuming Point and Sphere are of the geometric type, the implicit function defined by the file description in the figure above will evaluate to zero at all points in space. While not a very interesting object, it serves to demonstrate how object composition can be specified in a file. A pointer of Implicit type referencing the Difference object would normally be used to represent the entire function. The parameters of the object can be set by passing an eight element array using this reference. This array is then split by the binary Difference operator, which passes the first four elements to Sphere and the last four to Offset. Sphere sets its parameters normally, while Offset sets its own parameter from the first element of the array it received and passes the remaining three to Point, which also sets its values normally. Parameters are obtained from the object in a similar manner by passing an array to be filled to the Difference object. These procedures are used repeatedly by the particle system to interrogate and update the surface during interaction.

4.3 The Particle System

Rather than using polygonization [1] to visualize an implicit surface, a particle system [5] can be used. One advantage of this approach is that it requires less computation to construct a

new representation when the user has modified surface parameters. It also provides the mechanics necessary to allow for direct sculpting of the surface. In this manner the particle system method provides an effective representation for interactive surface modeling.

We have implemented the Witkin & Heckbert method [5], which uses two populations of particles. The particles in the set used to visualize the surface are called “floater” particles, because they float on the surface of the object. The particles in the other set are called “control” particles, and are used to manipulate the surface.

The floater particles are given a repulsion force that drives them away from particles with similar normals, which means they are most affected by nearby floaters on the same side of the object (surfaces could have narrow folds). To maintain performance as population increases, the particles must be either partitioned into buckets or associated with mesh connectivity. Making use of this information reduces the number of inter-particle computations. The velocity given to a floater due to repulsion is constrained by the surface normal and the current change in surface parameters. Components of the initial velocity that violate these constraints are removed, resulting in motion that keeps the particle on the surface during deformations. Particles are adaptively added and removed in a manner allowing them to quickly spread evenly over a surface. By manipulating system parameters the population can be controlled.

The user can also define the number of control particles in the system, as well as their location and velocity. However, once the number of control particles equals the number of degrees of freedom in the implicit function, the surface will be locked into a particular shape until some control particles are removed. When the user moves the control particles, they are first assigned a velocity corresponding to the intent of the user. A system of equations is then solved to determine the change in parameters that will cause minimal deviation from the desired

change in control particle motion. The actual control particle motion is then computed, and all particles are updated. The difference between the desired motion of control particles and their actual motion has earned them the reputation of being "slippery". [5]

In this section I will describe the phases of design and implementation that the particle system has gone through. We started with a very limited implementation and have been able to refine it into a much more robust and flexible system by using an iterative method of software engineering.

4.3.1 Preliminary Design

When I first implemented the particle system as my class project, the design was based on an application written by Hans Pedersen. His application used the Witkin & Heckbert method to visualize surfaces composed of blobby spheres and cylinders. The user could add new blobs, change the perspective, and modify some of the particle system parameters. However, it did not include manipulation via control particles. Hans used a three dimensional bucketing system to partition the floater particles. This locality-based partitioning scheme helped to maintain interactivity even as population increased.

This first iteration of our design was manifested in a single monolithic object that contained everything the particle system required. It contained a set of fixed-length arrays that held the particle attributes, as well as variables for other system parameters. In our implementation, a reference to the implicit surface would be given to this particle system object, and an update function would be called periodically. There were also some functions to change system parameters, but overall the interface was very simple.

There were a few problems with this first system. First, I was unable to get control particles working before the end of the semester. Second, particles on open surfaces (those with infinite

extent) would float off to infinity, making visualization via the particle method impossible. Third, we noticed that the constraint related to compensating for change in parameters was missing in our implementation. As a result, particles would lag behind when a surface was moved or deformed. This effect gave the impression that the particles were flowing over the surface from the front to the rear. Considering that we eventually want to create a mesh based on particle positions, this behavior is most undesirable. Not only would the mesh appear to flow over the object, it would be constantly re-stitched in the regions where particles were being added or removed. And fourth, we could only have up to a certain number of particles due to the fixed-length arrays.

4.3.2 Object-Oriented Design

To solve the problems we experienced with our first design, we took an object-oriented approach in the next iteration. The intent was to eliminate the restriction on the number of particles without losing performance, bound the particles within a specified region, implement the constraints in accord with [5], and to have a full implementation including control particles.

The design was based on the idea of a particle object. This object was designed to contain the current particle location, normal, and radius. We also designed a particle factory that could be used to create different types of particle objects. A factory is an object-oriented design tool that instances objects of types derived from a commonly inherited base-class. In addition to the basic type, we wanted to be able to create particle objects using shared memory for use in applications designed for multi-processor machines like those running the CAVE. Using objects created from shared memory, consistency of particle data could be maintained among processes. By using the factory concept, this would be transparent to the rest of the particle system design, allowing us to

use the same implementation in various situations by simply changing the particle factory used in the application.

Solving the boundary problem was tricky, and we are still refining our method. The idea is based on a boundary object that can tell the system whether a given point is in bounds. The two goals in our solution are keeping existing particles inside, and stopping new particles from appearing outside. The implementation of the former is simple; if a particle is given a velocity that takes it out of bounds, remove the component of the velocity causing the problem. The latter had us confused for a while, because particles would "leak" out of the bounding volume. We eventually realized that this was because a particle on the edge might spawn a new particle outside the boundary. Having no particles nearby, it would grow and eventually spawn another particle out of bounds, ad infinitum. A new particle is given a random location on the tangent plane of the surface, nearby the particle spawning it. We decided to keep choosing new locations if the spot at which to place the new particle ended up out of bounds. This resulted in infinite loops, so we decided to use a scheme that slowly increases the distance at which the new particle appears from its parent. This allowed us to begin visualizing open surfaces. However, we sometimes observe particles that float into the system from outside the boundary. This is a problem currently under investigation, and may be solved by simply aborting a spawning operation if the initial spawn location is found to be out of bounds.

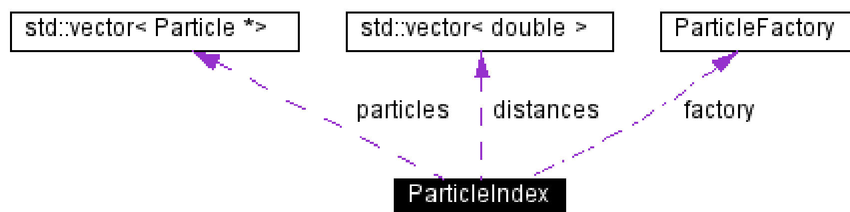

Figure 4: Contents of the ParticleIndex class

We utilized the STL Vector class to manage unbounded collections of these particle objects within a ParticleIndex object. To maintain performance, we designed a subclass using a 3D grid-based bucketing scheme very similar to the one used in the previous design, but this time using references to our particle objects (rather than array indices) to represent particles in the buckets. A different subclass might instead use mesh connectivity to enhance performance. Because of these implementation specific details, the index was designed to obtain new particles from a factory on behalf of the particle system engine so that it could perform the necessary bookkeeping (e.g. determining which bucket the new particle corresponds to, or reconnecting a mesh).

To display these particles we created a ParticleRenderer class, and an OpenGL-specific subclass. A particle renderer is simply called during the display process and given a vector of particles. It then renders the particles according to internal color and geometry settings using the graphics API in which it was implemented, in our case OpenGL.

The solution to the behavioral problem in our first system was rather straightforward. We simply needed to compute the valid components of the velocity while taking into account the current change in surface parameter values. The proper equation representing this computation is presented in [5] as Equation 5. We also made some larger modifications to the design of our computational unit. We decided to extract the elements of the system that we considered data into a separate object. This included the implicit surface, a particle index, a vector of control particles, a particle bounding volume, a set of desired parameter changes, a time step factor, and the desired population. The rest of the system would act as an engine to process this data when necessary. We could then use the same engine to operate on multiple sets of data, allowing us to

work with multiple behaviorally identical particle systems in our applications. Tweaking the settings of the engine would affect all of the particle systems.

The most significant addition to the design was support for control particles. During computation, a system of equations is created that represents the surface-particle constraints defined by control particle motion and surface parameter changes. [5] The solving of this system is a minimization that seeks to update the surface parameters while getting an effect as close as possible to the motion of the control particles. We used the Cholesky factorization method to solve this system of equations by making use of the implementation in [3]. We implemented an object based on code found in [12] to solve the system using Singular Value Decomposition. This object would be used if the Cholesky solver were to fail, as suggested in [5].

We also added a feature dubbed "flexibility". If all surface parameters are represented when solving the system of equations for surface deformation, all parameters are susceptible to change. This may not be the desired effect. In these situations, we provide the ability to specify, with a set of Boolean values, which parameters are considered flexible. In the user interface of an application there may be checkboxes next to the parameters that provide the ability to set their flexibility. When the system of equations is constructed in preparation to be solved by our system, we simply ignore the parameters that are to be constant and only update those that are flexible at the end, as suggested in [5].

Our design resulted in a fully operational particle system, ready to be used in our desktop and immersive applications. However, while beginning on those projects we decided to try one more design that had the potential to be even more useful.

4.3.3 Component-Based Design

The last design we implemented is based on the idea that a particle system has attributes that are represented as particle data, and behaviors that the system uses to interpret that data. We wanted to be able to replace or add these components of the system on the fly to get different effects. A particle system is assumed to have particle location and velocity by default, so these parameters are contained in a ParticleSystem object. A collection of attributes and behaviors is maintained by the ParticleSystem object.

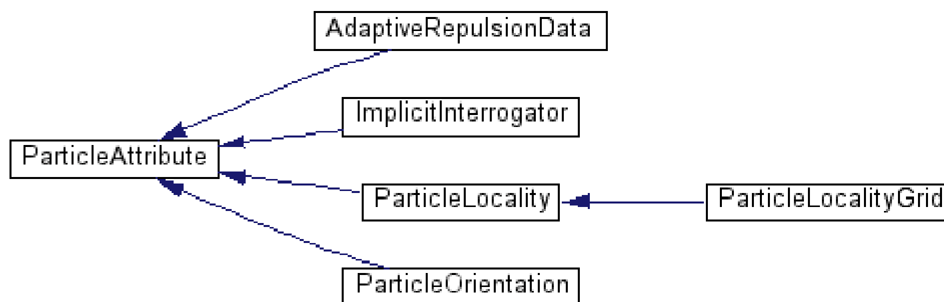

Figure 5: Inheritance graph for ParticleAttribute

Each behavior is dependant on certain attributes. For example, both ParticleRepulsion and ParticleFate operate on a ParticleRepulsionData attribute. If ParticleFate were removed, the repulsion data would persist for ParticleRepulsion. If a new behavior dependant on the repulsion data were added, it would be able to detect the existing data and use it. The attributes can be named, so that two distinct attributes of the same type can be differentiated by the behaviors. We assume that there is only one instance of any type of behavior in a system, although our implementation should be able to handle multiple behaviors of the same type. Any dependencies between behaviors are assumed to be resolvable by sharing an attribute so that there are no inter-behavior references, allowing individual behaviors to be easily added to and removed from the system.

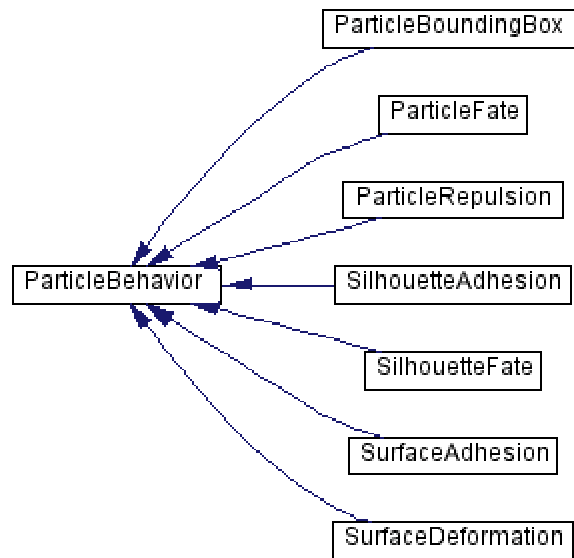

Figure 6: Inheritance graph for ParticleBehavior

Each behavior also implements four functions that are called in order. In the applyForce function, a behavior has a chance to exert a force on the particles in the system. Next, applyConstraint allows the behavior to remove components from the current velocities in order to constrain particle motion. The integration function is called to indicate that the behavior can update its internal parameters for the next time step. And finally, the cleanup function is called to do any post-integration steps to prepare for the next step. In complex systems, it is not necessarily the case that all force functions are called first, followed by constraint, integration and cleanup. They can be interleaved, just as the order in which behaviors might be called in one order for force application, and another order for constraint. For simple systems, the fullUpdate function of ParticleSystem executes these behavioral functions in the basic order.

During this redesign we also fixed another problem we had detected in our implementation of the particle system mechanics. When a floater particle moves out into an unexplored region of space its repulsion radius is supposed to increase, causing it to speed out toward the unknown. As it slows down when it encounters other pioneering particles, its radius will shrink so that the

particles can cover the surface evenly and present an accurate visualization. We had noticed that this change in radius wasn't happening in our system, and by carefully checking our implementation of the equations related to particle repulsion in [5], we managed to fix our problem and now observe the desired effect in this latest version of our system.

This component based design has allowed us to create a set of behaviors implementing silhouette particles that can easily replace the behaviors of the Witkin-Heckbert system. These particles, rather than evenly sampling the entire surface, remain on the silhouette edge of the surface. Thus, when looking at an implicit surface defined by the equation for a sphere, the particles form a ring around the "edge". Perhaps we will soon have an application allowing us to turn a Witkin-Heckbert particle system into a silhouette particle system on the fly.

5 The CAVE Automatic Virtual Environment

Figure 7: The CAVE Automatic Virtual Environment

The CAVE is a virtual reality theatre used at the National Center for Supercomputing Applications for projects in various disciplines. “CAVE” is a recursive acronym standing for CAVE Automatic Virtual Environment. It is a projection-based system originally developed at the Electronic Visualization Laboratory (EVL) at the University of Illinois at Chicago. [16] [17] CAVE technology was unveiled to the computer graphics community at SIGGRAPH ’92 and has since been installed at many research facilities and museums throughout the world.

A suitable definition for virtual reality is “*a medium composed of interactive computer simulations that sense the participant's position and replace or augment the feedback to one or more senses -- giving the feeling of being immersed or being present in the simulation.*” [7] In the CAVE, these simulations are applications that run on an SGI Onyx2 supercomputer. Various devices are connected to this computer to handle input and output. The application makes use of some of these devices to sense the participant’s position and intent, and others to replace and

augment feedback to the senses. In this section I will describe the most common input and output devices and the two software libraries I used to create my applications.

5.1 Input Devices

The CAVE can be augmented to respond to multiple participants in an audience, but usually it is configured for a single primary user. One of the components of the CAVE is the tracking system, which continually detects the position (location and orientation) of a set of sensors. At NCSA we have a PC dedicated to this task, which can make use of electromagnetic or ultrasonic tracking hardware. Both of these components offer six-degrees-of-freedom tracking, which is sufficient to detect the position of the sensors. The CAVE tracks the head of the primary user via a sensor mounted on a pair of shutter glasses. The shutter glasses worn by the remainder of the audience are not tracked, and thus the system cannot accommodate for their perspectives. An additional sensor is commonly used to track a wand device, and our electromagnetic setup has one more tracker used in some programs to track the user's off-hand. Knowing that this tracker would be needed in the sculpting application, I used this hardware. One drawback of using the electromagnetic system is that I ran into problems with poor accuracy near the metal framework of the CAVE.

The wand is the primary source of event-based input in the CAVE system. In addition to having its position tracked, there are three buttons and a pressure-sensitive joystick mounted on the wand.

When the user decides to interact with an object in the virtual world, a common method used is to point the wand at an object and press the button corresponding to the action desired. This event is then processed by the application, which determines which object (if any) is being

pointed at and what to do as a result of the button press. There are of course other methods of interaction, and it is up to the software engineer to determine what model works best.

Figure 8: A Fakespace™ wand

The joystick is commonly used to navigate through the virtual world. Pushing up or down, for example, might translate the user in the direction that the wand is pointing, while pushing left or right performs a rotation. This model is a common “flight” model, named so because it allows a user to fly to any location in a three dimensional space. In a “drive” model, rotation and translation would be restricted to the ground plane. Because the joystick is analog, pushing harder on it can influence the speed of travel.

Acoustic input has also been used in the CAVE. When using the CAVE the participant is normally standing in an upright position, which makes it difficult to use a keyboard. Although there are single-hand devices designed for text input, it is often more desirable to use voice recognition as it is less cumbersome. Voice commands can be given via microphone to trigger events in an application, or audio channels can be utilized to enable communication with remote participants.

5.2 Output Devices

A major portion of the CAVE's output devices is composed of the projection system. The framework supporting the three upright screens is fashioned as a cube. The screens attached to this frame (actually a single sheet of material bent around the corners) cover the front, left, and right sides. Images are projected onto these screens from the rear by CRT projectors. Projecting from the rear solves the problem of casting a shadow while using the system. At NCSA, we require the use of mirrors to bend the projection path because the room housing the CAVE is not large enough to accommodate a direct throw from the projectors onto the 10'x9' screens. The floor is also used as a display surface, with the image being front projected from a projector suspended from the ceiling. In some applications the shadow produced as a result is actually a realistic feature.

SGI's Infinite Reality graphics hardware produces the images that are displayed on these screens. We use a resolution of 1024x768 that is refreshed at 96Hz. The framerate is actually halved because each screen needs to be rendered twice per "frame", once for each eye. The eyes are then fooled into seeing in stereo by Crystal Eyes active stereo LCD shutter glasses, which synchronize with the graphics hardware over an infrared interface and alternately block the left and right eye. Each eye is only allowed to see when the images intended for it are displayed on the screens. This happens fast enough that one generally doesn't notice it, but the effect can eventually cause discomfort in some individuals during extended sessions.

Figure 9: Shutter glasses

There are a number of IR emitters placed around the CAVE that provide a signal for all active shutter glasses, so that everyone (not only the primary user) wearing the glasses can perceive the stereo effect. However, the images produced are generated for the primary user's perspective, resulting in skewed perception for other participants. The degree of distortion is related to the difference in perspective, so that a participant looking in the same general direction as the primary user experiences little skew. When working with a modeling application such as the implicit sculptor it is important to be aware of this inadvertent cause of distortion.

The CAVE is also outfitted with a surround sound system. There are eight speakers that are mounted at the corners of the CAVE. The speakers belonging to the front bottom corners can be removed if not required, as they occlude the screens. The signals going to these speakers are generated by a sound server that runs on a supporting SGI workstation. The signal then runs through a mixer and on to the speakers.

5.3 CAVElib and FreeVR

The CAVE is driven by a 12 processor SGI Onyx2 supercomputer using four Infinite Reality graphics rendering pipes. A CAVE application running on this system is composed of a handful of interdependent processes that communicate with each other via shared memory. The software library commonly used to manage these processes and acquire information from the CAVE hardware is called CAVElib [9], a proprietary CAVE library designed at EVL. The FreeVR library [10], currently under development by Bill Sherman at NCSA, implements the same functionality with additional features.

Both of these VR libraries contain functions allowing the user to query the system for the button presses, valuator values, and tracker positions required to respond to the user's position and intent. They also provide the ability to present the user with feedback through the virtual world by rendering graphics using the OpenGL or SGI Performer libraries. Other types of feedback can also be provided by making use of additional libraries, but CAVElib and FreeVR assist the programmer by managing the state of the graphics pipeline. For this reason there are OpenGL and Performer versions of these libraries.

The VR libraries also manage the processes and inter-process communication that comprise the software state of the system. When an application is started, it will initialize and configure shared memory for the application and establish a reference to a location in shared memory where the tracking system places sensor data. Once shared memory has been prepared, the main process forks compute, display, and networking processes.

In both CAVElib and FreeVR the compute process is actually the post-fork main application process. Generally, this process enters a loop that is terminated by pressing the ESC key at the console or by indicating a desire to exit the application from within the virtual

environment. Within this loop, the process performs computation necessary to update the virtual world at each time step. In a scientific application, for example, this might be the application of systems of equations to data that generate geometry representing specific phenomena. The scientist can then interact with the application to investigate these phenomena in the data (which may have been obtained as a result of computational runs on NCSA's supercomputing clusters). The main process also responds to the input gathered from the CAVE's input devices.

In the OpenGL versions of our two libraries, one display process is created for each wall. These processes all continuously execute the draw function of the application. Common multi-process issues arise, and the use of mutual exclusion locks as well as barriers can be useful. In a deeper investigation of the implementation these libraries use, I found that the classic readers-writers problem occurs [13]. The approach taken to solve the readers-writers problem in both libraries results in starvation for the writers. As a result, when the amount of geometry in a scene increases, it becomes difficult for the main process to obtain a write lock and perform its update. Of course, the bias toward readers is understandable, as framerate would suffer if the display processes were to starve when obtaining read locks. Performer operates in a similar fashion, but much of the inter-process communication details are handled by the library.

The networking process is used to share important data with remote collaborators in CAVElib. For example, in a shared virtual environment, avatars are often used to represent participants so that they can interact more easily with one another. For this, tracking data for the head and props such as the wand are required. Participants can then easily observe the glances and gestures of others. Additional application-specific data can be shared across the network as well. For collaborative environments requiring large amounts of shared data, EVL is developing

a library called CAVERNSoft [11] that includes network databases and quality-of-service techniques to provide optimum performance.

Although these VR libraries have much in common, there are certainly differences. One of the ways in which FreeVR differs from CAVElib is that it is an open source project. The benefits of being open source are that other developers can verify the implementation, as well as propose and develop additional features. FreeVR is also much more configurable than CAVElib, allowing an application to work with many kinds of virtual reality systems. In FreeVR the notion of a system is a collection of processes utilizing a number of devices and inputs. Different systems can be established using a configuration file that is read by the application when executed. The library then operates in a manner specific to the configuration while satisfying the requests of the application. A feature not yet implemented allows the user to remap inputs in the configuration file. If the user interface works better when two buttons are swapped, or a different input device provides better interaction, a few simple modifications to the configuration file might be all that is needed. Although FreeVR does have these advantages, there is no built-in support for networking, and it is still in its developmental stages.

6 Applications

We developed a number of applications using our surface library. For the desktop, I created a surface visualization program to test correctness while implementing the particle system. Once I had confidence in the ability of our particle system to visualize a surface, I started working on the immersive sculpting application using the CAVE at the NCSA, where I have been working as a research assistant. In this section I will describe these applications.

6.1 Implicit Viewer

I originally designed ImpView (Implicit Viewer) to debug the particle system during development. When executing ImpView, the user specifies a file describing the implicit surface to be visualized. The application then reads the description from file and constructs an object. A particle system is created, assigned to the object, and its surface is visualized. The particles are drawn as discs, colored such that their pink sides face “outward” and their blue sides face “inward”. The user is able to rotate and zoom to inspect the surface. There is also a feature included to perform a rough polygonization.

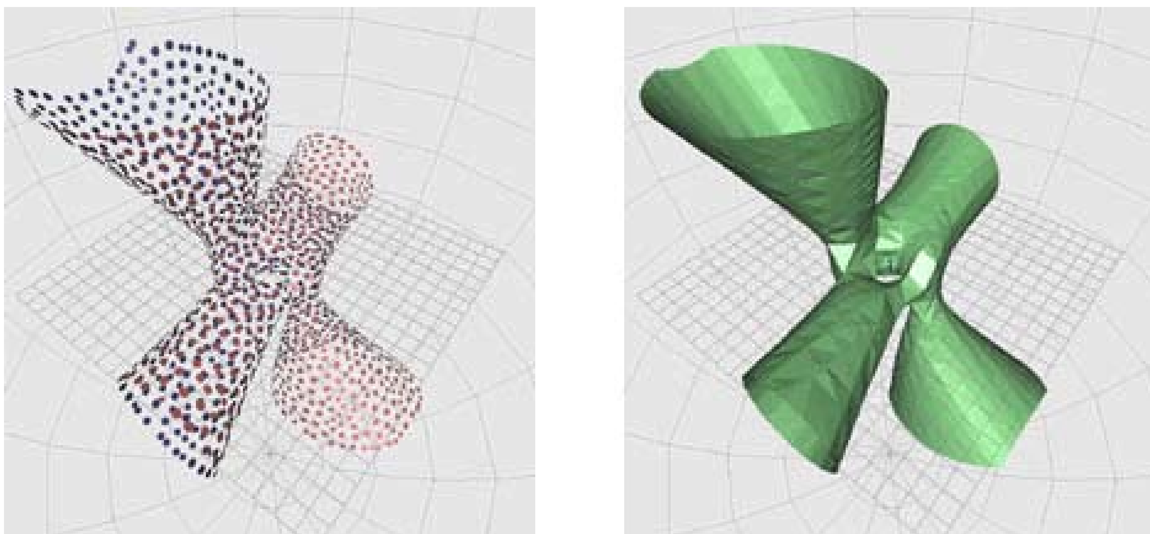

Figure 10: Particle (left) vs. marching (right) comparison using ImpView

The application can be compiled on the same platforms as the surface library, as the only additional library used is GLUT, which implements a platform-independent windowing API. ImpView has been tested in IRIX, Linux, and Windows and we still find it a useful tool.

6.2 Immersive Surface Sculptor

The immersive surface sculpting application started out as nothing more than an immersive version of ImpView. Even so, once implemented we were able to experience the benefit that this prototype of the immersive application provided. On two dimensional display surfaces such as the painter's canvas and the desktop monitor, shading and perspective are cues that are commonly used to convey a sense of depth. Even using these techniques, it is difficult to interact with a three dimensional virtual world. The CAVE and other stereo display systems provide an additional visual depth cue known as parallax. When observing an object, each eye sees the object and creates an image from a different perspective. The brain perceives depth in the world by combining these two differing perspectives. Stereoscopic displays fool the brain into thinking it is seeing three dimensional objects by creating a separate image for each eye. The result is a much more convincing sense of depth, which in the early version of our sculpting application gave us the sense that we could touch or reach through objects with our hands.

After completing the first version of this application, I added the ability to create, delete, and move control particles. At every time step, I used the object-oriented version of our particle system to calculate surface parameter changes based on the motion of these control particles. Because of the mechanics of the system, the control particles often would not end up exactly at the location to which the user dragged them. This is part of the slippery nature of this particle scheme. To provide an intuitive interface, it was important to not let that slipperiness hinder the ability of the user to interact.

The interaction method at this stage used only the wand and operated as follows. The joystick is used to navigate around the surface using the flight mode of travel, and the right button resets navigation. To place a control particle, the user presses the middle button near the surface. A particle then appears at the location on the surface closest to the wand. A press of the middle button at a location close enough to an existing particle deletes it. To drag a control particle, the user presses the left button with the wand near the chosen particle and then moves the wand while holding the button down. When the button is released, so is the particle. The particle continues to be in the grabbed state during dragging even if it is repositioned by the particle system such that it no longer rests near the wand location. During dragging, a velocity is given to the particle in the direction of the wand. The magnitude of this velocity is proportional to the distance to the wand. During short and slow dragging motions, this method is very intuitive because wand motion and particle motion do not differ very much. However, during longer and faster motions the particle does lag behind, giving the interface a viscous feel. Even so, it provides a very hands-on and intuitive interface for interacting with implicit surfaces.

I have taken the level of intuition one step farther by using the natural gesture of pinching in my final implementation of this interface. To do this, I'm using the PinchGlove™ device from Fakespace. This device obtains information from two trackable gloves and communicates with the host computer via a serial connection. Each glove has a mount, upon which a sensor such as those used by our tracking system can be affixed. This allows the application to take the position of the hand into account. The gloves have a conductive material sewn into the fingertips. When two or more fingertips make contact, a circuit is completed and the device can register which fingers are pinching. It can handle any combination of pinches, but in my application I only used thumb-finger contacts on one of the hands. This allows the user to handle the wand in the off

hand to travel. There are methods of implementing travel using the gloves, but as I am only studying the use of grasping for interacting with surfaces, I decided to use the simple and common flight travel paradigm once again.

Figure 11: A PinchGlove™ with attached tracker

Figure 12: Flying through an implicit valley in a quartic surface

Pinching the thumb and middle finger places a control particle on the surface, and pinching the thumb and ring finger removes a control particle, replacing the functionality of the middle

wand button. Pinching with the index finger will grab an existing control particle, or create a temporary particle if the user pinches an area on the surface where there is no control particle. The control particle is given a velocity in the same manner used in the previous method while the user maintains this contact. When the fingers are separated the particle is released, resulting in deletion of the particle if it was temporary. The use of these temporary particles enables swift tweaking, while particles placed with the middle finger act as stable anchors.

One of the first questions I sought to answer was whether I needed some sort of virtual object to represent the glove in the virtual world (an avatar), or if I could simply use the hand itself. Being able to use just the hand would give the user a first-person perception of their hand in the virtual world, and would be most convincing. To clarify, if I were to reach out with my hand and grab a particle just as I would expect to grab a marble in the real world, it would give me a strong feeling that I am holding the particle. On the other hand, using an avatar would not be as convincing because I would have a second-person perception of my action.

To give the proper effect when using only the hand, I would need to have a representation of the actual location of the pinch in the application. I decided to test my idea in an application that calibrates the pinch locations of a user. The application provides a small sphere shaped target and requires the user to pinch at its location to calibrate the thumb-forefinger pinch displacement for each finger of the left hand. Knowing the location of both the glove sensor and the target in the virtual world, the application calculates offset vectors. These vectors are maintained by the application and applied to the tracker information to arrive at the pinch location for each finger. Tiny spheres are rendered in a color-coded fashion corresponding to each finger at the pinch locations to represent the knowledge of the application. Once the user is satisfied with the calibration, a button press switches the application into a test mode in which there are a number

of spheres randomly positioned in the interaction space and colored to correspond to the fingers. Colored points are drawn at the pinch locations. Pinching on a sphere with the appropriate finger allows the user to drag the sphere around the virtual space.

If there were no tracking error, this first-person method of interaction would probably work well. However, tracking is most accurate in the center of the CAVE, and error increases as a sensor moves toward the screens. This is evident because the positions of the pinch indicators relative to the glove change dramatically as the hand is brought away from the center of the space. Without the indicators or some other sort of avatar, it is unlikely that a user would be able to grasp an object lying near the screens of the CAVE.

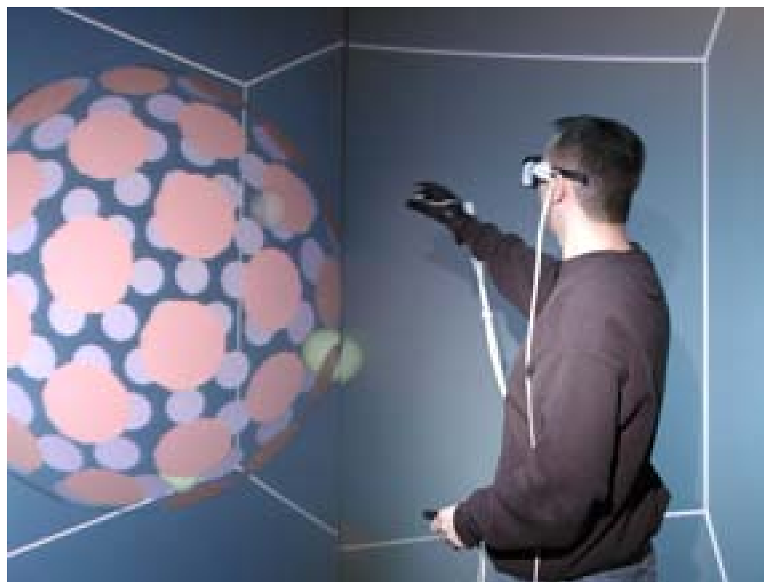

Figure 13: Manipulating an implicit sphere

Based on my observations using this program, I decided to use an indicator in the surface sculpting application that would represent the location at which pinch interactions would occur. I did not want this object to be occluded by the hand, so I placed it a fair distance away from the tracked position of the glove. I also did not want this indicator to occlude other objects in the virtual world. Since it is an abstract indicator rather than a more representative avatar such as a

hand model, it appears in the application as a small transparent grey sphere (this can be seen in the figure above).

The result is a truly intuitive user interface with a hands-on feel. It operates at interactive rates when simply visualizing a surface as well as during manipulation. The performance will vary a bit depending on the number of parameters in the function defining the surface and the number of particles used. For one of our simplest primitives, the geometric sphere, the system visualizes the surface using 30 to 100 particles at a computational rate of 120Hz. While being manipulated with up to 4 control particles, the rate drops to 90Hz. This is even better than our graphics framerate. For quartic algebraic surfaces, our most complex primitive, the system visualizes using 200 to 500 particles at around 14Hz. When manipulated with 10 control particles, this drops to 10Hz, which is still interactive.

Figure 14: Folding a quartic surface

7 Future Work

This work could be taken further in the future in a number of ways. First of all, the system could be analyzed further to provide an understanding of how the various parameters of the implicit surfaces and particle systems affect performance. Second, the existing system could be enhanced. The newest version of the particle system could be integrated, retiring the wand and using both gloves would allow for investigation of two handed pulling motions, and other methods of manipulating multiple particles at once could be tested. Third, additional visualization methods could be researched. Particle size or color could be varied to represent the surface curvature. Volumetric methods to visualize the values of the implicit function inside a surface, or streamers interrogating the gradient field would give insight into how other implicit function properties change during surface deformation. And fourth, collaborative sculpting could be investigated. A teacher might better explain topology to his class by working with them in an exploratory virtual environment. Designers or artists in different parts of the world could work together on projects. Creating a collaborative application that could handle the real-time demands of this system would be an interesting challenge.

References

[1] Jules Bloomenthal. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers, 1997.

[2] Alan Paeth.*Graphics Gems V*. Academic Press, 1995.

[3] Roldan Pozo. *The Template Numerical Toolkit Home Page*. National Institute of Standards and Technology, August 2000. http://math.nist.gov/tnt

[4] *OpenGL – High Performance 2D/3D Graphics*. http://www.opengl.org

[5] Andrew Witkin and Paul Heckbert. "Using Particles to Sample and Control Implicit Surfaces." *Computer Graphics (SIGGRAPH '94 Proceedings)*, pp. 269-278, July 1994.

[6] Kalev Leetaru. *The CAVE™ at NCSA*. National Center for Supercomputing Applications, October 8th 2001. http://cave.ncsa.uiuc.edu

[7] William Sherman and Alan Craig. *Understanding Virtual Reality*. Morgan Kaufmann Publishers, 2002.

[8] Nicolai M. Josuttis. *The C++ Standard Template Library*. Addison Wesley, 1999.

[9] Dave Pape. *The CAVE Programmer's Guide to CAVELib*. Electronic Visualization Laboratory, August 16th 2000. http://www.evl.uic.edu/pape/CAVE/prog

[10] William Sherman. The FreeVR HomePage. February 22nd 2002. http://www.freevr.org

[11] Jason Leigh. *CAVERNSoft G2*. Open Channel Foundation, 2002. http://www.openchannelsoftware.org/projects/CAVERNsoft_G2/

[12] William Press, Brian Flannery, Saul Teukolsky, William Vetterling. *Numerical Recipes in C*. Cambridge University Press, 1988. http://www.nr.com/

[13] Avi Silberschatz, Peter Galvin, Greg Gagne. *Applied Operating System Concepts*, p192-197. John Wiley & Sons Incorporated, 2000.

[14] OpenGL ARB, Mason Woo, Jackie Neider, Tom Davis. *OpenGL Programming Guide*. Addison Wesley, Second Edition, Version 1.1, 1997.

[15] John Hart et al. *Surface – an advanced surface library*. University of Illinois at Urbana-Champaign, 2002. http://graphics.cs.uiuc.edu/projects/surface

[16] Carolina Cruz-Neira, Daniel Sandin, Thomas DeFanti, Robert Kenyon, John Hart. "The Cave – Audio Visual Experience Automatic Virtual Environment." *Communications of the ACM*, Volume 35 #6, pp. 64-72, June 1992.

[17] Carolina Cruz-Neira, Daniel Sandin, Thomas DeFanti. "Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE." *Computer Graphics (SIGGRAPH '93 Proceedings)*, pp.135-142, August 1993.

[18] *General Bathymetric Chart of the Oceans*. International Hydrographic Organization, Fifth Edition, 1984.

[19] Kevin McDonnel, Hong Qin, and Robert Wlodarczyk. "Virtual Clay: A Real-time Sculpting System with Haptic Toolkits." *Proceedings of the Symposium on Interactive 3D Graphics*, 2001.

[20] Kevin McDonnel and Hong Qin. "Dynamic Sculpting and Animation of Free-form Subdivision Solids." *Visual Computer*, Volume 18, Number 2, pp. 81-96, April 2002.

[21] Daniel Keefe, Daniel Feliz, Tomer Moscovich, David LaidLaw, Joseph LaViola Junior. "CavePainting: A Fully Immersive 3D Artistic Medium and Interactive Experience." *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, March 2001.

[22] Thavida Maneewarn, Blake Hannaford, Duane Storti, Mark Ganter. "Haptic Rendering for Internal Content of an Implicit Object." *ASME Winter Annual Meeting Haptics Symposium*, November 1999.