The DataPaper: Living in the Virtual World

Mark Green
Chris Shaw

Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada

Abstract

Virtual reality user interfaces are a new type of user interface based on placing the user in a three dimensional environment that can be directly manipulated. Virtual reality user interfaces have a wide range of applications in scientific computing, remote manipulation, communications, and the arts. This paper presents a cost effective hardware and software architecture for this type of user interface. Some of the hardware devices required to produce this type of user interface are discussed, along with the techniques used to combine them into a usable system. A low level and high level set of software tools are described. The low level tool set handles the interaction between the input and output devices and the user's program. The high level tool set assists with the development of virtual reality user interfaces. An example user interface developed with our tool set is described, along with suggestions for further research.

Keywords: Virtual Reality, User Interfaces, 3D Interaction

1. Introduction

Virtual reality user interfaces are a new type of user interface based on placing the user in a three dimensional environment that can be directly manipulated. The user's complete field of vision is filled with computer generated images, and he or she can use three (or more) dimensional input devices to directly interact with the underlying application. By using a head-mounted display, the user can move through the environment and have the images respond to his body movements. The main aim of virtual reality user interfaces is to give the user the feeling of directly interacting with his or her application, and not using a computer. Virtual reality user interfaces have a wide range of application in scientific computing, remote manipulation, communications, and the arts. The basic ideas behind virtual reality user interfaces are developed in [Fisher86], [Brooks Jr.86], [Krueger85] and [Green89].

As an example of this type of user interface consider a fluid dynamics computation. These computations are performed on large two or three dimensional grids, and the results of these computations are recorded at fixed intervals of time (every 0.1 seconds of simulation time for example). The results of the computation could include the position of the fluid's surface, and for each grid cell, the flow velocities, pressure, and temperature. Obviously, a printed version of these results will be very hard to interpret. Similarly static plots of the flow velocities at each time interval do not give a good impression of how the flow develops over time. Fluid flow is a dynamic phenomenon, thus a dynamic three dimensional presentation of the results is required. At each time step of the computation, a three dimensional image of the flow can be produced (two images if a head-mounted display is used). These images can be presented to the user as they are produced, giving an impression of how the flow develops over time. Given a fast enough computational engine, the images could be produced in real time allowing the user to directly monitor the computation. The user can interact with the computation through a multi-dimensional input device, such as a DataGlove. The user can interactively enter the boundary conditions for the computation, and change these conditions and other parameters as the computation evolves. In this way the user can steer the computation to the more interesting parts of the model, and thus save considerable amounts of computing time. The user can interactively slap the water with the DataGlove and watch the changes in the flow. This type of user interface is possible with today's technology, and this example is based on one of the prototype virtual reality user interfaces that we have constructed.

In this paper we describe our experiences with constructing virtual reality user interfaces. The two main purposes of this paper are to present an affordable and usable hardware and software architecture for this type of user interface, and identify some of the major research problems in this area. The second section describes the hardware that we are using to explore this type of user interface. The next section describes the low level software that lies between the hardware and the tools that we have produced for constructing virtual reality user interfaces. The typical programmer of our system doesn't directly interact with this software level.

The fourth section of this paper describes the software tools that we have produced for constructing virtual reality user interfaces. The following section describes one of the applications that has been developed using these tools. The final section of this paper presents some of the research problems that we have identified.

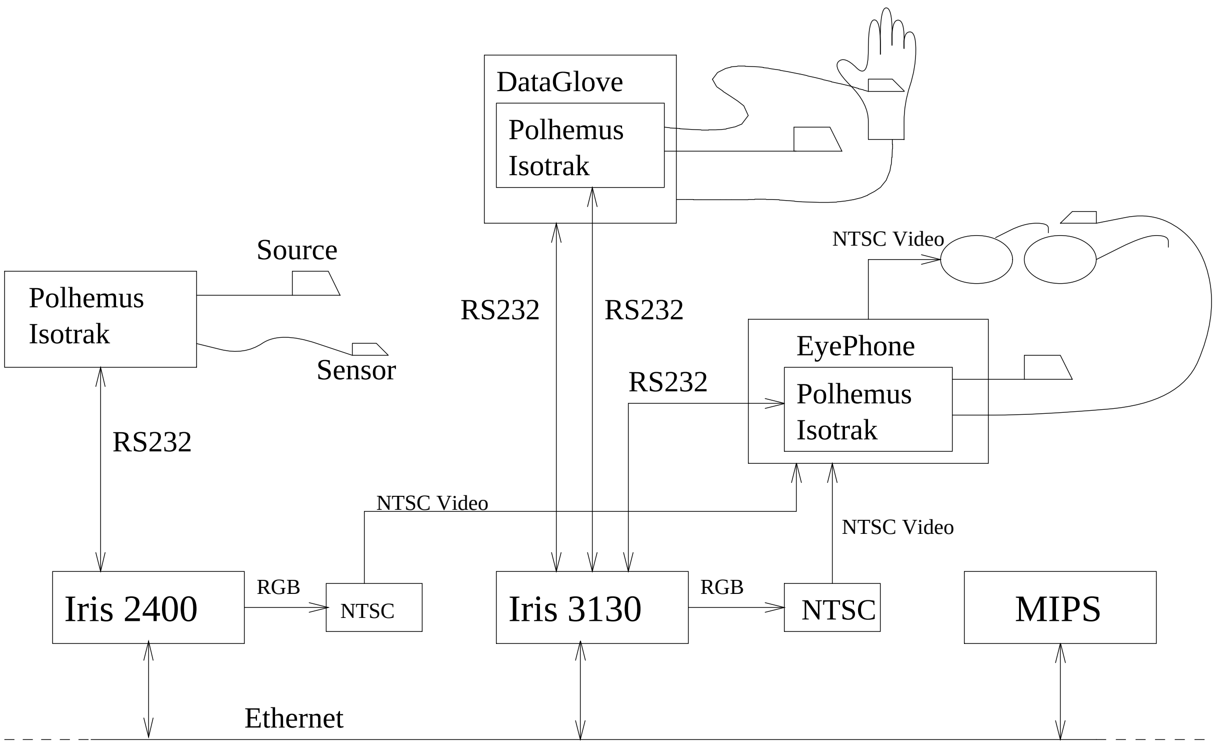

Figure 1 Current system architecture

2. Hardware Configuration

This section describes the hardware that we have assembled for investigating virtual reality user interfaces. The current laboratory configuration is shown in figure 1. Most of the hardware described in this section is available commercially, and the parts that aren't can easily be made in any electronics shop. The two key components in our hardware configuration are a head-mounted display and a DataGlove.

The head-mounted display that we are using is the VPL EyePhone. This device consists of two LCD TV's that are mounted less than two inches in front of the user's eyes. An optics system is used to expand the images so they cover most of the user's field of vision, and an optional diffusion panel is used to blur these images. Without the diffusion panel the individual red, green and blue dots in the displays are visible. The LCD's are driven by two NTSC video sources. In our configuration we use two Silicon Graphics Iris workstations to generate the video signals. One of the workstations is a 3130 and the other is a 2400. The workstations are synchronized over an ethernet, and two NTSC converters are used to convert the video signals from the workstation into NTSC for the LCD's. The two workstations are not genlocked, but using the ethernet for synchronization gives a good illusion of stereo in the applications that we have tried so far. The head-mounted displays uses a Polhemus Isotrak digitizer to determine the position and orientation of the user's head. This information is used to generate the images that are presented to the user.

The main input device that we are using is the VPL DataGlove [Zimmerman87]. This device senses the position and orientation of the hand, along with the bending of the fingers. An Isotrak digitizer is used to determine the position and orientation of the hand. Special fiber optics placed on the back of the hand are used to determine the orientation of the fingers. At the finger joints, the fibers have been treated to attenuate the light when the joint is bent. Two joints on each finger are instrumented, and can be sampled up to 60 times per second. This information can be used to determine the positions of the fingers, or the gesture that the user is making.

The use of an Isotrak in the DataGlove was a source of problems. First, the DataGlove doesn't allow us to directly interact with the Isotrak. The DataGlove configures the Isotrak to report roll, pitch and yaw angles, which is not the best way to report the orientation of an object (this is discussed further in section 3). This problem can be solved by detaching the Isotrak from the DataGlove hardware and driving it directly. Second, having more than one Isotrak in the same room requires special care (we have three in our current configuration, one in the DataGlove, one in the head-mounted display, and one standalone Isotrak digitizer). By default the Isotrak samples 60 times per second and it requires the complete 1/60 second to process the sample. Since the Isotrak broadcasts an electro-magnetic field, only one device can sample at a time. Multiple devices can be handled by time multiplexing the sampling. In our case each of our three Isotraks are sampled 20 times per second. Using this sampling rate doesn't seem to adversely effect the quality of the interaction. The DataGlove electronics provides the sync signals required for three Isotraks, so this problem can be easily solved.

Another problem that occurs with the Isotrak is its limited range of accuracy. The Isotrak digitizer consists of two main components, which are a source that produces the electro-magnetic pulse and a sensor that detects this pulse. In normal operation the source is kept at a fixed position and the sensor moves about the environment. Polhemus Navigation Systems claim that the digitizer has an accuracy of 0.25 inch RMS when the sensor is up to 30 inches away from the source, and "reduced accuracy" at separations up to 60 inches. The accuracy of the Isotrak is affected by metal objects and other sources of EM radiation that lie close to the source or sensor, such as computers, fluorescent lights, and display screens. In a computer graphics laboratory this 60 inch range will never be achieved. In practice we have found that noise in the readings becomes noticeable when the source and sensor are separated by more than 24 inches, and a high level of noise occurs when the separation reaches 50 inches. In the case of positioning with the DataGlove this noise can be a major problem, but the user can partially adjust to it through the use of visual feedback. On the other hand, it is a serious problem for the head-mounted display, since the noise in the Isotrak causes the images to jiggle. When the user's head is not moving there can be significant changes in the images presented to him, which can be very disturbing. We have solved this problem by filtering the data produced by the Isotrak as described in the next section.

The only piece of custom hardware that we are using is a device we call a navigation box. This device has four buttons and three potentiometers and is contained in a small box that can be attached to the user's belt. The navigation box plugs into a port on the DataGlove electronics box, which has three spare A/D converters and four spare digital inputs. The use of the navigation box is described in section 5.

3. Low Level Software

As one might expect, the Isotrak and DataGlove devices need driver software to integrate them into our environment. These devices communicate with the external world via a serial data link, and are entirely controlled by commands sent via the serial line. This section describes the software structure required to effectively drive these devices, and the steps that were taken to best utilize the data gathered by them.

3.1. DataServers

Our first attempt at a driver was a simple library of routines that polled an Isotrak for data, waited for the response, and returned the properly interpreted data to the application program. The advantage of this scheme was that it minimized the time lag between request and use of data. A simple polygon mesh viewer which used this library gave good response, that is, movement of the polygon mesh corresponded almost exactly with movement of the 3D sensor with little more than a 1/30 second lag.

The source of this lag is the sum of the time to send the poll request, plus the time required by the Isotrak to format its data, plus the time to send the data across the serial link to be interpreted by the host computer. To nullify this sampling lag, we reversed the order of polling such that a poll request was sent before the polygon drawing operation, and the data was collected after the mesh was drawn and the frame buffer was swapped. If the polygon mesh was small enough, this scheme worked quite well, since the time to draw the model was less than the time to gather the next data sample. In essence, sampling and drawing occurred simultaneously, speeding up the drawing operation and increasing the data sampling rate. However, if the polygon mesh was too large, each drawing operation required several times the sampling period to complete, and thus by the time the program collected the sampled data, the data was no longer up-to-date. This could be seen as a small lag in the program's response.

To combat the problem of sampling latency, we built a software system based on the client-server model. The server program, called `isotrakd` (for Isotrak Daemon), is the sole manager of the serial port that communicates with the Isotrak. While a client is connected, `isotrakd` samples the Isotrak continually, and returns the latest position and orientation values to the client program upon request. This arrangement has numerous advantages, the first being that the sampling latency between `isotrakd` and the Isotrak is relatively low, while the sampling latency between client and `isotrakd` is nearly zero. Since `isotrakd` is a separate process, time that would otherwise be wasted in the monolithic polling scheme while waiting for a frame buffer swap can be better spent processing Isotrak data.

The second advantage of `isotrakd` is that the application software is simplified, since the client needs only to make a few calls to initialize service, and only one sampling call per screen update to gather data. Also processes on other machines can access the Isotrak across the network, which means that the Isotrak can be used with a wider range of applications, and it can be used when the host machine's display is occupied by other unrelated tasks. This network capability will be expanded in the future by dedicating a small, fast processor with many serial ports solely to the task of managing all the serial-controlled devices.

The client-server interface allowed us to experiment cleanly with the idea of adding more intelligence to `isotrakd`. The most obvious addition was to make `isotrakd` robust enough to detect device resets and anomalous conditions in the Isotrak itself. The second addition was to filter the Isotrak data as it was received, which solves part of the noise problem that occurs when the sensor is moved to the outer parts of its range. The basic filtering scheme used was an N-stage FIR filter, where the filter's output is the weighted sum of the most recent N samples. If N is too large, the filtered data shows a noticeable lag between actual motion and filtered output. If N is too small, the noise-reduction effect is not apparent. Currently, we find the filter with the best trade-off between noise reduction and low lag is a 5-stage box filter, which is a classic low-pass filter. Each data element is filtered separately, so each of X, Y, and Z position are filtered independently, and each of the four quaternion elements is filtered independently with a quaternion normalization step afterwards.

One unexpected effect of the client-server model is the decoupling of the `isotrakd`'s rate of sampling the Isotrak versus the client sampling `isotrakd`. The result of this decoupling is that `isotrakd` usually samples the Isotrak at between two and four times the screen update rate in our simple polygon mesh viewer. What this means is that the filtering can be performed at essentially no real cost.

Given the success of the Isotrak server, we also built a server called `gloved`, or Glove Daemon. The purpose here was essentially the same: package the interface to the DataGlove such that the client program need only make a small number of calls to interact with the hardware. Filtering is supported in `gloved` also, since the flex sensors on the glove are not terribly accurate, and the navigation box is also quite noisy.

To properly use the DataGlove, one must first calibrate the DataGlove electronics to the maximum and minimum finger bend angles produced by the user's hand. During the calibration process, the host computer uses the raw fibre-optic intensity values to generate a function which maps intensity to bend angle. When calibrated, the DataGlove uses these mapping functions in table lookup form to return bend angles. `gloved` manages this process, and saves old calibrations on `gloved`'s file system for later retrieval if the user wishes. By default, the user is determined by having the client pass the user id of the client process.

3.2. Orientation Data

As mentioned in section 2, the Euler angles of Yaw, Pitch and Roll are not the best way to express orientations, for three reasons. Euler angles express an arbitrary orientation as a series of three rotations about one of the three coordinate axes. There are twelve possible orderings of the axes, which results in unnecessary ambiguity when stating orientation. Aeronautics and graphics uses the Z, Y, X or Yaw, Pitch, Roll ordering, while physics uses Z, X, Z ordering.

Independent of order, the second problem with Euler angles is a phenomenon called "gimbal lock", which is the loss of one degree of rotational freedom. Gimbal lock occurs when the first and last Euler axes of rotation are coincident or nearly coincident. For example, a ship's compass uses a mechanical gimbal to ensure that the compass needle rotates about an axis which is normal to the tangent plane on the earth's surface. A gimbal will fail at certain "impossible" orientations, such as when a ship is stood on its bow or stern, and in this situation, compass readings are meaningless because the ship is not pointing in any direction given by the compass needle. In fact, the ship points along the needle's axis of rotation.

Unfortunately, Euler angles are used for orientation by the Isotrak and DataGlove, and are implicitly used in the viewing operations in most graphics packages. We discovered the last point by accident on one or our workstations. We noticed in one of our viewing programs that the environment would suddenly turn upside down when we moved into certain areas. A simple test program was constructed that performed a 360 degree rotation about the Z axis in 1 degree increments. When the viewing direction approached the positive or negative Y axis the environment would perform a 360 degree rotation about the viewing direction, due to gimbal lock along this orientation. Obviously, Euler angles cannot be used in virtual reality user interfaces, since there is no guarantee that gimbal-locked orientations will not occur.

The third problem is that Euler angles have a discontinuity in its coordinate space at two of the poles. On the Isotrak, these poles are at Pitch = +/- 90 degrees. In fact, pitch is restricted to lie in the range from -90 degrees to +90 degrees, which means that when the sensor is oriented such that the Pitch angle crosses the pole, the Yaw and Roll are rotated by 180 degrees to generate a Pitch angle of less than 90 degrees. If the data is noisy, this double rotation is noticeable when the orientation nears the poles. Moreover, filtering the Euler angles worsens the problem.

The solution to these problems is to use the unit quaternion, which is a four element vector that expresses orientation as a rotation about an arbitrary axis. Quaternions are usually written in the following way:

$$q = [w,(x,y,z)] = [\ \cos\frac{\theta}{2}\ ,\quad \sin\frac{\theta}{2} \times (x',y',z')\ ] \qquad 1$$

$$w^2+x^2+y^2+z^2 = 1$$

$$x'^2+y'^2+z'^2 = 1$$

Quaternions were originally developed as an extension to complex numbers by Hamilton [Shoemake85], and as such are two-part numbers with a real scalar part w, and an imaginary vector part (x,y,z). Euler showed that any orientation can be stated in terms of a single rotation about a reference vector, much like any position can be stated as a translation from a reference point. The unit quaternion can be interpreted as a rotation by $\frac{\theta}{2}$ degrees about the arbitrary axis (x,y,z), with the unit quaternion being restricted to a magnitude of 1. Equation 1 shows the unit quaternion's definition, and further shows that the vector (x',y',z') is also restricted to a magnitude of 1. When `isotrakd` filters quaternion data, the magnitude of the filter output quaternion is likewise forced to 1. The value $[1,(0,0,0)]$ is the multiplicative identity quaternion. To catenate rotations, simply multiply their quaternions, as in equation 2.

$$[s_1,v_1][s_2,v_1] = [(s_1s_2 - v_1 \cdot v_2),(s_1v_2+s_2v_1+v_1 \times v_2)] \qquad 2$$

This formulation of rotation catenation takes 16 multiplies and 12 adds, while the 4×4 homogeneous matrix form takes 64 multiplies and 48 adds.

One of the advantages of quaternions is that they are continuous for all orientations, so gimbal lock does not occur.

The quaternion can be used to generate a 3D (non-homogeneous) rotation matrix directly, at the cost of 9 multiplies and 15 adds [Shoemake85], as shown in equation 3.

$$M = \left\{ \begin{matrix} 1-2y^2-2z^2 & 2xy+2wz & 2xz-2wy \\ 2xy-2wz & 1-2x^2-2z^2 & 2yz+2wx \\ 2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 \end{matrix} \right\} \qquad 3$$

This matrix can be used in one of two ways. For the DataGlove and stand-alone Isotrak applications, the current transformation matrix is multiplied by this rotation matrix to place the hand or the object being manipulated in the proper orientation. For the head-mounted display, the transpose (i.e. inverse) of this matrix is multiplied onto the transformation stack as the rotation component of the viewing transformation. In both cases, the results are quick, efficient, and continuous at all orientations.

The standard DataGlove restricts access to its internal Isotrak, and only reports the Euler angles for orientation. A stand-alone Isotrak, including the one that comes with the EyePhone, reports quaternions when so requested, so we have re-wired the DataGlove box to have separate paths to both the DataGlove and its internal Isotrak.

4. Software Tools

Producing a virtual reality user interface can require a considerable amount of programming. This suggests that a set of software tools should be developed to assist in the development of this type of user interface, in the same way that we have produced software tools for other types of user interfaces (for example see [Green85] or [Singh89a, Singh89b]). The tools that we have developed so far address two types of applications. The first type of application involves adding a visualization component to an existing program. We are using an application skeleton to address this problem. The other problem is the development of three dimensional environments that the user explores and interacts with. This type of application is supported by an interactive modeler and viewing program. Both of these software tools are described below.

4.1. Application Skeleton

The application skeleton addresses the problem of adding a virtual reality user interface to an existing application. There are certain aspects of a virtual reality user interface that are essentially the same in all applications. For example, all applications must synchronize the workstations that are generating the images, perform viewing and stereo projections based on the viewer's current position and orientation, and provide navigation through the environment. The application skeleton handles these aspects of the user interface and provides a framework for the application programmer's code. This skeleton can be used with existing applications, or form the basis of new applications.

To use the skeleton the application programmer must provide a procedure that produces an image of his data (the skeleton computes the viewing transformations). A call to this procedure must be placed at the appropriate place in the skeleton (this point in the code is indicated by a comment). After compiling the new version of the skeleton on both workstations, the user can interact with his data. The skeleton provides a default user interface that allows the user to walk around his data and navigate in the environment. Either a mouse or the navigation box can be used to move about the environment. The navigation box is the default device if it is connected, otherwise the mouse can be used for navigation. If the navigation box is used the three potentiometers on it are used to change the origin of the user's coordinate space. All the user's head motions will now be in respect to this new origin. If button one on the navigation box is pressed, the potentiometers can be used to scale the user's movements. By default, the scale factor is 1.0 in all three dimensions. Thus, if the user moves an inch in his space, he will move an inch in the model space. If the mouse is used, the three mouse buttons are used to move in the x, y and z directions, the scaling option is not available with the mouse.

In many ways the skeleton provides a minimal virtual reality user interface, but it provides a very quick way of getting something running. If the programmer already has a procedure that can draw his data or model, then a virtual reality user interface for that model can be constructed in less than an hour. Thus, the skeleton provides a good prototyping tool, and a convenient mechanism for evaluating different display techniques.

4.2. Interactive Modeler and Viewer

One of the main applications of virtual reality user interfaces is exploring and interacting with three dimensional environments. These environments could represent the design of a new building [Brooks Jr.86], the environment that a remote manipulator is operating in [Fisher86], or a work of art. In all of these examples the environment consists of a number of objects that the user wants to explore or interact with. These objects are either static or have relatively simple motions (in comparison to fluid dynamics or other physical simulations). The design of this type of application consists of three main activities, designing the geometry of the individual objects, placing these objects in the environment, and designing their behavior. We have designed a simple set of tools for constructing this type of virtual reality user interface. This tool set is not intended to be a production tool, its main purpose is to identify the problems that need to be solved in order to produce usable tools.

The tool set is divided into three main components. The first component is an interactive modeling program. This program is used to create the objects in the model and place them within the environment. The second component is the interactive viewing program that allows the user to explore the environment. The third component is an environment compiler which converts the output of the modeling program into the format required by the viewing program.

The modeling program is based on two key concepts, which are primitives and masters. A primitive is a template that is used to create an object. The set of primitives in the modeling program can be extended at any time. At the present time a programmer must write a C procedure for each new primitive, but we are in the process of designing a special primitive definition language that will simplify the construction of primitives. Most of the primitives in the modeler are parameterized. When the user creates an instance of the primitive, he specifies the parameter values for it using a set of graphical potentiometers. Each of the parameters has a default value that produces a reasonable object. For example, one of the default primitives is a floor. The parameters for this object are whether the floor is tiled, and the size of the tiles.

A master is a collection of objects that is treated as a unit. The objects in a master can be either primitives, or instance of other masters. When a object is added to a master the user interactively positions, scales, and orients the object within the master's coordinate system. An environment is produced by creating instances of the masters. When an instance is created, the user can position, scale and orient it in the same way that objects are handled within masters. The transformations used to position the instances within the environments can be interactively edited.

The viewing program is based on BSP trees [Fuchs80, Fuchs83]. A BSP tree is a binary tree that has a polygon at each of its nodes. The left subtree of the current node contains polygons that are behind the polygon at the node, and the right subtree contains polygons that are in front of the polygon at the current node. The normal to the polygon is used to determine its front and back sides. When a BSP tree is displayed, the eye position is compared to the polygon at each node as it is visited. The coordinates of the eye can be inserted into the plane equation for the polygon to determine whether the eye is in front or behind the polygon. If the eye is in front of the polygon, the left subtree is displayed first, then the polygon at the current node followed by the right subtree. Otherwise, the right subtree is displayed first, followed by the polygon at the current node and the left subtree. The BSP tree is independent of the eye position, each time the viewer's eye changes the BSP tree is traversed. Thus, BSP trees are an efficient solution to the hidden surface problem for static environments. For large environments the construction of a good BSP tree can be quite time consuming, therefore, the scene compiler is used to pre-compute the BSP trees.

The viewer has essentially the same user interface as the skeleton application. Each time that the user moves his head, the BSP tree is traversed to give the current view of the environment. The user can use either the mouse or the navigation box to change the origin of the user's coordinate system and to change the scale of his motions. With our current hardware configuration environments with a few hundred polygons give reasonably good response (between 5 and 10 updates per second). The upper limit for this configuration is about 500 polygons. Our experience indicates that most interesting environments will have several thousand polygons, and this should be possible with current workstation technology (one of the workstations we are using in our current configuration is 5 years old).

The scene compiler converts the data structure used by the modeler into a set of BSP trees. Basically, each of the objects in the model is converted into a BSP tree. Using a set of BSP trees instead of one BSP tree for the entire environment solves two problems. First, the size of the BSP tree grows quite quickly as the number of polygons in the environment increases. When each node of the BSP tree is constructed, the remaining polygons in the model must be divided into two disjoint groups depending on whether they are in front of or behind the current polygon. The plane of the current polygon usually cuts several of the polygons in the environment, thus each of these polygons will give rise to two or more polygons in the BSP tree. As the number of polygons in the environment increases, the number of polygons that will be cut also increases (there is a greater probability that any given polygon will be cut). Thus, for large environments it is hard to construct good BSP trees. For a convex polyhedron, none of the polygon planes cut any of the polygons in the polyhedron. Most of the objects that occur in the environments are basically convex, therefore, good BSP trees can easily be built for most of these objects. Thus, building a BSP tree for each object, and then ordering them correctly for each view point, will result in fewer polygons to display, and fewer tree nodes to visit.

The second reason for building a separate tree for each object is to allow for simple motion in the environment. If each object has a separate BSP tree, it can be moved without recomputing all of the BSP trees in the environment. The only thing that will change is the order in which the trees must be displayed. We are currently developing algorithms and heuristics that will accommodate these simple motions.

5. Applications

The hardware configuration and software tools described in the previous sections have been used to produce a number of virtual reality user interfaces. The development of one of these interfaces for fluid dynamics is described in this section.

The starting point for the development of this user interface was an existing fluid dynamics program [Bulgarelli84]. This program is written in Fortran and runs on two of the computers in our department (a MIPS M/1000 and a Myrias SPS-2). The main aim of the exercise was to add a virtual reality user interface to this program with minimal changes to the existing program code. The two main reasons for choosing this program were the availability of the source code, and that it appeared to be a typical scientific program.

The fluid dynamics program runs on one of the processors mentioned above and the virtual reality user interface runs on the Iris workstations. The two programs communicate over the ethernet, with the user interface part of the application responsible for starting the fluid dynamics code. This particular configuration minimizes the changes that must be made to the fluid dynamics program. The following changes have been made to the fluid dynamics program. First, the program now reads all of its boundary conditions from a file (the original version of the program read some of the boundary conditions from a file, and others were set with assignment statements). Second, the format of the output was changed slightly so it was easier for the user interface component to read. Third, the output routine was modified to read a packet of data each time a packet of results was sent to the user interface. This packet exchange is used to synchronize the two programs, and to pass data from the user interface to the fluid dynamics program. The data in the packet indicates changes in the boundary conditions for the computation. Fourth, the routines that enforce the boundary conditions were modified to accept data from the user interface. These modifications involved less than 5% of the fluid dynamics program code.

The development of the user interface started with the application skeleton described in section 4.1. The first step in developing the user interface was to produce images of the fluid flow. This was done by using the height of the water surface at each point on the computation grid to define a polygonal mesh representing the fluid surface. The polygons in this mesh are displayed in back to front order based on the position of the viewer's eye (since the mesh is rectangular and located at a fixed position in space, this is fairly easy to do). With this version of the user interface the user could view the flow as it developed over time, and he could walk around, or through, the flow to obtain different views of it.

The other change to the skeleton program was adding the interaction with the DataGlove. If the hand comes in contact with the surface of the flow, the boundary conditions must be changed. There are two types of contact possible, one type is to slap the fluid, which involves momentary contact with the fluid and a transfer of velocity (really momentum, but in the computations it turns into a velocity boundary condition) to the part of the flow that the hand comes in contact with. This type of contact can be handled by sampling the position of the DataGlove at fixed intervals of time. If the position of the DataGlove intersects the flow, then its velocity is computed from previous positions and sent to the fluid dynamics program as a change in velocity boundary condition. The second type of contact occurs when the hand is placed in the flow and held at that position. This type of contact results in a new position boundary condition. The fluid dynamics program has not be modified to handle this type of boundary condition, therefore, this type of contact has not been added to the user interface.

6. Conclusions

In this paper we have described our experiences with developing virtual reality user interfaces. This experience has suggested a number of research directions, which we will briefly outline.

Fast graphics hardware is important for virtual reality user interfaces. Our experience indicates that displays capable of drawing 20,000 polygons per second are required. This type of display technology is available commercially, but it is expensive and two displays are required for stereo. We need cheap displays that can draw large numbers of polygons. We also need devices that can produce sound and tactile feedback. On the input side we need better devices for interacting with the environment.

One of the main problems with developing virtual reality user interfaces is an almost total lack of good software tools. The tools that we have discussed are a start in this direction, but much more needs to be done. Our tools do not address the problem of dynamic objects in the environment. To handle this type of object we need efficient hidden surface algorithms, and ways of modeling and describing their motions. The description of their motion must include how they interact with other objects in the environment. We also need better tools for adding user interfaces to existing applications and producing good visualizations of their data.

Brooks Jr.86.F. P. Brooks Jr., Walkthrough - A Dynamic Graphics System for Simulating Virtual Buildings, *Proceedings 1986 Workshop on Interactive 3D Graphics*, Chapel Hill, North Carolina, 1986, 9-21.

Bulgarelli84.U. Bulgarelli, V. Casulli and D. Greenspan, *Pressure Methods for the Numerical Solution of Free Surface Fluid Flows*, Pineridge Press, Swansea, UK, 1984.

Fisher86.S. S. Fisher, M. McGreevy, J. Humphries and W. Robinett, Virtual Environment Display System, *Proceedings of ACM 1986 Workshop on Interactive 3D Graphics*, Chapel Hill, North Carolina, 1986, 77-87.

Fuchs80.H. Fuchs, Z. Kedem and B. Naylor, On Visible Surface Generation by A Priori Tree Structures, *Siggraph'80 Proceedings*, 1980, 124-133.

Fuchs83.H. Fuchs, G. Abram and E. Grant, Near Real-Time Shaded Display of Rigid Objects, *Siggraph'83 Proceedings*, 1983, 65-69.

Green85.M. Green, The University of Alberta User Interface Management System, *Siggraph'85 Proceedings*, 1985, 205-213.

Green89.M. Green, Artificial Reality: A New Metaphor for User Interfaces, *International Conference on CAD & CG, Beijing, China*, 1989.

Krueger85.M. Krueger, VIDEOPLACE--An Artificial Reality, *Human Factors in Computing Systems CHI'85 Conference Proceedings*, 1985, 35-40.

Shoemake85.K. Shoemake, Animating Rotation with Quaternion Curves, *Siggraph'85 Proceedings*, 1985, 245-254.

Singh89a.G. Singh and M. Green, A High-Level User Interface Management System, *Human Factors in Computing Systems CHI'89 Conference Proceedings*, 1989.

Singh89b.G. Singh and M. Green, Generating Graphical Interfaces from High-Level Descriptions, *Proceedings of Graphics Interface'89*, 1989.

Zimmerman87.T. G. Zimmerman, J. Lanier, C. Blanchard, S. Bryson and Y. Harvill, A Hand Gesture Interface Device, *CHI + GI 1987, Human Factors in Computing Systems and Graphics Interface*, 1987, 189-192.