(54) Title: METHOD AND SYSTEM FOR SIMULTANEously CREATING AND USING MULTIPLE VIRTUAL REALITY PROGRAMS

(57) Abstract: A method and system for creating and using virtual reality computer programs. Multiple independent virtual reality programs (116) may be simultaneously presented on the user’s display device (120). The programs may operate on separate computers connected by a network (100). The central program (110) services and manages the one or more programs that operate with the system. A graphics subsystem (112) is used by the central program for creating the images comprising the virtual environment. The graphics subsystem includes graphics libraries that support the user’s display device. In addition, the central program uses a scene graph structure to maintain the dynamic virtual environment as the programs operate. An application program interface is used to facilitate communication among the programs and the system.
(84) Designated States (regional): ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, HR, HU, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— with international search report

— before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments

For two-letter codes and other abbreviations, refer to the “Guidance Notes on Codes and Abbreviations” appearing at the beginning of each regular issue of the PCT Gazette.
METHOD AND SYSTEM FOR SIMULTANEOUSLY CREATING AND USING MULTIPLE VIRTUAL REALITY PROGRAMS

FIELD OF THE INVENTION

The present invention generally relates to virtual reality systems, and more specifically relates to a method and system for simultaneously creating and using multiple distributed virtual reality programs.

BACKGROUND OF THE INVENTION

Virtual Reality

Virtual reality (VR) refers to the presentation of a three dimensional artificial environment that may be perceived as reality by the user. A user may interact with and be projected into the virtual environment with the implementation of devices that allow the system to receive signals from the user.

Effective virtual reality immerses the user in computer generated sensory data which may include audio, visual, and tactile data. Visual data is delivered to the user on various devices such as a projector screen, monitor, head mount display, retinal projection, or special goggles. Display devices may be immersive or non-immersive.

An immersive device is one which uses separate images for the right and left eyes of the user and encompasses a significant portion of the user's field of vision. An example of such a device is the CAVE (Computer Automatic Virtual Environment). The CAVE is an elaborate virtual reality system that projects images around the user (e.g., on the walls, floor and/or ceiling) not merely on a monitor. The CAVE produces separate images for the left and right eyes, resulting in a
stereoscopic effect which produces an illusion of depth. In addition, CAVE allows multiple users to experience the virtual reality simultaneously.

Non-immersive display can be accomplished using a CRT display or LCD projector. These devices do not simulate stereo vision and they cover only a small portion of the user's vision.

Input devices may be conventional devices, such as keyboard and mouse, or specialized devices, such as data glove, eye-motion detector, or voice recognition devices.

Field-Related Technologies

Virtual reality programs operate on the various input and output devices to generate the images and other sensory data perceived by the user. Manipulating the images, for example, that create the virtual reality environment requires sophisticated algorithms and computer programming technology. Virtual reality programs involve significant graphics manipulation, most of which is not uniquely specific to the program. Consistent with modern programming techniques, much of the graphics processing is performed by graphics libraries. Many of the standard graphics library, and hence VR programs, implement scene graphs.

Scene graphs are directed acyclic graph data structures for representing multi-dimensional aspects of the scene, i.e., the visual presentation of the VR environment. To describe the scene with a scene graph, the information about aspects of the scene, such as shape, transformation (location in space) and properties take the form of nodes attached to each other in a deliberate manner as to constitute a graph. The links connecting the nodes establish relationships between the aspects of the presentation that the nodes represent. Hence, for example, a virtual reality program that simulates the motion of a car may generate a scene graph containing numerous nodes branching out of a root node, where the individual nodes represent parts of the car, such as the wheels, doors, windows, mirrors, steering wheel, signals, lights, brake pedal, and acceleration pedal etc. The nodes are connected so as to establish that if the transformation (position) of the wheels move, the rest of the car also moves in corresponding fashion. The connections between
the nodes also establish that if one of the doors moves (open) the wheels do not necessarily move.

Elements of the scene graph are displayed by a process called rendering. Rendering entails traversing a scene graph to determine information corresponding to the shapes defined in the graph and the associated properties, and generating display signals in accordance with the information. To continue with the example of the car, the program traverses, in a particular order, the scene graph containing the nodes describing aspects of the car, rendering each aspect in the correct position relative to the other aspects/nodes, thereby creating the presentation of a car.

Currently a number of software libraries are available to programmers for use in conjunction with designing and running virtual reality programs. The following is a representative list of popular software libraries. These libraries vary in performance, functionality and abstraction level (i.e., manipulating pixels or manipulating objects).

OpenGL (www.opengl.org: the contents of which are incorporated herein by reference) is a library of functions that can perform basic graphics tasks such as drawing, shading, transforming, lighting, texturing, and projecting. It also includes advanced features such as mipmaps, anti-aliasing, projected textures and platform-specific extensions. OpenGL is hardware accelerated on certain platforms and can be used very effectively in virtual reality while still being applicable to the high quality work of the film industry. OpenGL does not support any hierarchical scene graph or multiprocessing model. OpenGL is written in C.

IRIS Performer (www.sgi.com/software/performer: the contents of which are incorporated herein by reference) is built directly on top of OpenGL and is considered the standard tool of the high-end visualization and simulation industry. Users range from military and government to film post-production studios and TV stations. Performer adds a hierarchical scene graph to OpenGL that allows programmers a more intuitive and efficient way of managing objects and the transformations applied to them. Performer also adds a multiprocess pipeline model to OpenGL.

This pipeline significantly improves performance in rendering and database access on
 multiprocessor platforms, and also allows for pipelining and parallelization of the tasks that normally occur in OpenGL applications. As a result, multiprocessor computers will run Performer application much faster than uniprocessor machines. Performer is supported on LINUX and IRIX operating systems and can handle a larger bandwidth of data than OpenGL alone. Performer also supports the special hardware of SGI workstations, which are the state of the art in the field (see www.sgi.com/products).

Multigen Vega (www.multigen.com/products/vega1.htm; the contents of which are incorporated herein by reference) is the military's tool of choice for creating war simulations. It is essentially identical to Performer but has extensions that simulate special effects, load special terrain databases and support various simulation specific needs. Vega is supported on IRIX.

World Tool Kit (www.sense8.com/products/wtk.html; the contents of which are incorporated herein by reference) is a clone of IRIS Performer that runs operates on Windows NT and IRIX. It also includes a client/server tool that allows users on different computers to all use the same applications.

Other VR software systems supply a development library as well as runtime components that add value outside of simply rendering images. For example, Bamboo (www.watson.net/Bamboo; the contents of which are incorporated herein by reference) Bamboo is a component (i.e. plugin) framework for developing shared VR environments. Based on the Netscape Portable Runtime platform, Bamboo is portable and somewhat language independent. Bamboo projects contain various modules that can be loaded and unloaded at runtime. The display of Bamboo based programs is accomplished with OpenGL and a scene graph.

Distributed Interactive Virtual Environments (DIVE) (www.sics.se/dive/dive.html; the contents of which are incorporated herein by reference) is an Internet based VR system tuned for use in multi-user environments. DIVE allows multiple use manipulation either via user supplied Tcl scripts or with compiled C code. Each DIVE user stores a copy of the "shared world" and
renders it individually. DIVE currently supports head mounted displays and on screen display and is limited to about three platforms, e.g., IRIX, MS Windows, and LINUX.

DEVA ([www.aig.cs.mman.ac.uk/systems/Deva](http://www.aig.cs.mman.ac.uk/systems/Deva); the contents of which are incorporated herein by reference) is geared toward developing intelligent techniques of describing behavior and mitigating metaphysical differences between these behaviors. DEVA also addresses the management of multiple users at distributed locations. DEVA is based on top of MAVERIK, an OpenGL rendering system.

**Current Systems**

Current software libraries and systems are unable to support the display of concurrent unrelated programs. Users of modern computer systems frequently use multiple unrelated programs concurrently and the ability to do so is vital to the usability of any computer system (i.e., running netscape and MS Word at the same time). Not only are current VR libraries and systems unable to support multiple programs, but also the work required to turn two programs into one is time consuming.

Some systems (DIVE, DEVA, Bamboo) allow each user to affect the virtual environment simultaneously, but do not allow a single user to execute multiple, independent programs. This is because these systems are designed specifically for multi-user environments without considerations for multi-program design. The implementation of such systems is concerned mainly with quick and efficient updating of shared or distributed databases.

Current systems have limited runtime scalability, meaning that as they are required to do more computational simulation work, they are unable to maintain acceptable performance. This can be attributed to their inability to separate the platform the program is running on from the platform the program displays on. Situations where the need for separation between a program's display and computation arise when the computation complexity for rendering and simulation become too great for one machine. Specific examples include: A complex fluid dynamics simulation running on a supercomputer displaying on an immersive device not connected to the
supercomputer itself; an application running on a PC, displaying on an immersive system; and applications running on computers distributed around the world, such as Internet routers, all displaying on a single immersive device.

Other shortcomings of current systems are platform dependency and language dependency. Systems usually support only one or two platforms for display and only one language for programming. These shortcomings greatly restrict the development of useful and widely used software for display devices. The present invention satisfies these and other needs.

SUMMARY OF THE INVENTION

The present invention is a system and method for creating and using virtual reality (VR) computer programs. Specifically, the invention allows for the simultaneous display of multiple independent VR programs by managing the VR display and other sensory output devices on which these programs operate. The system includes the capacity to display programs that are running on any machine connected to a network, e.g., LAN and/or Internet, or on the machine running the VR display device (or devices). To achieve a virtual reality environment, the system operates the graphics subsystem that creates images of the virtual environment, and services and manages the programs that operate with the system.

The system maintains a central mechanism (Construct) for processing the presentation of one or more application VR programs operating concurrently. The system acts as an interface between the various applications and the output device or devices that comprise the VR presentation to the user. The applications may be interactive or self-contained; may be operating locally or remotely over a network; and may be written in any language. The applications are limited only by the imagination of the programmers, provided the programs conform to the system API, application program interface. Each program operates as if it were an independent program, where instructions affecting the presentation (VR environment) are processed by the central mechanism. To facilitate the display of programs operating on different computers, the system combines current graphics systems with a distributed object system.
In a preferred embodiment of the invention, the system provides an API supported by at least one graphics library and uses a scene graph schema for managing the data comprising the presentation of the VR environment. Upon receipt of instructions affecting the presentation, the system updates the scene graph accordingly and realizes the change, typically by updating the display, though naturally extendable to other output mediums. The system maintains the scene graph using distributed objects and system identifiers for each node and provides the system identifiers to the application programs as needed. The applications use the system identifiers provided by the system in their instructions relating to the VR environment.

BRIEF DESCRIPTION OF THE DRAWINGS

Other objects, features and advantages of the invention discussed in the above summary of the invention will be more clearly understood from the following detailed description of the preferred embodiments, which are illustrative only, when taken together with the accompanying drawings in which:

Figure 1 is a block diagram of the preferred embodiment of the present invention;
Figure 2 is a block diagram of a Construct in accordance with the preferred embodiment;
Figure 3 is an illustration of an application program in accordance with the preferred embodiment;
Figure 4 is a flow chart showing a method of processing an application program in accordance with the preferred embodiment;
Figure 5 is a flow chart showing a method of processing by the Space Manager in accordance with the preferred embodiment;
Figure 6 is a block diagram of an Implementation portion of the Construct in accordance with the preferred embodiment;
Figure 7 is an illustration of interprocess communication in accordance with the preferred embodiment;
Figure 8 is an illustration of broadcast interprocess communication in accordance with the preferred embodiment;
Figure 9 is a block diagram of the hierarchy among types of objects in accordance with the preferred embodiment; and

Figure 10 is a block diagram of a scene graph data structure in accordance with the preferred embodiment.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

In the preferred embodiment of the present invention, a system enables users to participate in a virtual reality (VR) environment generated and manipulated by one or more independent VR application programs. The primary runtime environment, called the Construct, is the platform in which the virtual reality presentation is generated. The user operating the virtual reality session starts the Construct. Along with the Construct, the user may implement one or more application programs to participate in the same virtual reality session. The system facilitates the use of VR application programs that operate on the environment, various output devices that project the environment perceived by the user(s) and optionally various input devices that determine each user's attention and movements. The system also provides tools for programmers creating VR applications. Such tools include a framework for creating specialized space management programs, application program interface (API) and other developmental libraries.

The Construct is the central program that receives the multiple and contemporaneous inputs/outputs (influences) in the VR environment from application programs. Influences on the environment include requests for functionality from application programs operating on the environment. The system provides the capability for displaying multiple applications concurrently, sharing resources and facilitating cooperation between the applications. This allows the user to move between applications without closing them down. In addition, any aspect of the virtual reality session experienced by the user may be shared by the applications. Such interoperability is facilitated by the distributed nature of the underlying mechanics of these VR programs. Due to the distributed nature of the mechanics, the usage, implementation and interface of the programs are loosely coupled and can be located on different machines. Such distribution aids in gaining scalability, modularity and extensibility.
In order to have the capacity for processing multiple application programs concurrently, the system uses a single shared VR environment typically represented by a single scene graph. A scene graph is a directed acyclical graph data structure that represents a hierarchy of elements (termed nodes) that can be delivered to a rendering system and turned into an image on the appropriate device, e.g., immersive display. Scene graphs aid application programmers in thinking about the scenes that they build and manipulate. The format of the scene graph is conducive to efficient processing by computer graphics systems. The system supports basic scene graph libraries available in current graphics systems such as IRIS Performer, JAVA3D, SSG, WTK, and VRML. The use of different scene graph libraries allows the various graphics and scene graph systems to be interchanged without affecting the functionality or requiring recompilation of the Construct.

Each application program may be designed to generate and manipulate its own scene graph, but in operation all functions affecting the scene graph are executed at the Construct’s scene graph. The system provides a uniform API for managing functions affecting the scene graph. The API provides a common communication format between application programs and the system. Application VR programs may be designed independently of the system, then written in compliance with the API and operate seamlessly with the system. To support the API, the system uses shared graphics libraries. The API, scene graph and graphics libraries are discussed below.

In the preferred embodiment, the Construct is implemented in an object oriented programming language. As is known in the field, the Construct may be implemented in other languages and adapted for other computer platforms. According to the preferred embodiment, the Construct uses objects to generate and facilitate the VR experience. In general, an object is a collection of data and functionality that are conceptually related. For example, a typical program has an object to manage file operations and such object may be called "file object". The nodes that comprises the scene graph are objects. The scene graph is the data structure used for storing and managing the VR environment as it is to be perceived by the user. Hence, the nodes are the building blocks that comprise the scene graph and thereby the VR environment. For example, in a VR
environment having a bouncing ball, the properties, features, and functionality of that ball are collected and managed in one or more nodes. The node (or nodes) may be said to represent the ball. If the VR environment is also to have a cat, the cat is represented by another node (or group of nodes). The Construct and application programs generate and manipulate various objects, including nodes, as each proceeds to operate in a VR session. Examples of some of the objects typically used with this system are set forth below.

A space manager is used to manage the presentation of a VR session produced by the operation of multiple applications. The space manager is a program that controls the allocation of space within the VR environment and updates the environment to reflect changes requested by application programs or caused by the user. Each time a program changes its spatial state, the Construct alerts the Space Manager and adjusts parameters of the environment accordingly. Without a space manager, the Construct would otherwise execute the directions of each application program individually, without any regard for the presentations of the other applications functioning contemporaneously.

According to the present invention, the VR presentation of each application is spatially centered, and conceptually stacked one on top of the other unless explicitly specified otherwise by the application. With the implementation of a space manager, each application may be allocated a distinct space in which to operate its VR presentation. The space manager may be designed with varying levels of complexity. For example a space manager may provide each application with a distinct origin in space but the presentation of each application is not confined a particular subspace. Alternatively, the space manager may allow the user to designate an origin for each application. The space manager may also allow users to modify parameters of the management or shrink programs at the users command. Naturally, to operate effectively, there can be only one operating space manager associated with any given Construct at any given time.

The system for generating the Construct may be a distributed system implemented using multiple computers. The various application programs may be implemented on different computers networked to the computer implementing the Construct. In addition, portions of the system (e.g.,
the space manager) may be implemented on separate computers. Referring to Figure 1, the Construct 114 and Space Manager 118 may be implemented on a computer 110, while application programs 116 that use the Construct may reside on the same computer 110 or another computer, e.g., computer 112. In addition, the Space Manager may be located on any machine, e.g., computer 110 or computer 112. The application programs communicate to the Construct 114 via a communication network 100, such as, a Local Area Network, Wide Area Network or the Internet. The display device(s) 120 are situated and connected to the computer 124 at the user’s location. Typically, the input device(s) 122 are also connected to the user’s computer 127.

The Construct 114 uses graphics libraries to affect the display of the scene graph manipulated by the Space Manager and application programs. Specifically, if an application program wants to change the appearance of individual graphic elements in the environment, it requests or instructs the Construct 114 to do so. The application programs themselves are independent of any specific graphics library. The application programs use the API to communicate with the Construct. The Construct 114 forwards the request to the Space Manager 118, which then places a notification on a queue accessible by the Space Manager 118. The Space Manager 118 fulfills the request by updating the scene graph. This typically involves calling one or more functions in the applicable graphics library. The Display component of the Construct proceeds to render the scene graph, effecting an updated presentation by the display or other output devices.

To use (run) the system, the user starts the system and a blank environment appears. This environment is the visible manifestation of the Construct. During system startup, the display, communication and associated management utilities are started. Once the system has completed startup, application programs on the local machine, or on any machine connected to the network, may manipulate the scene graph which in turn affects the display. Application programs are executed in a conventional manner. For instance, a user may run an application program using a command interpreter or from any other program. The Space Manager places the application program in the virtual environment and once placed, the user may interact and experience the program through the immersive display.
In the preferred embodiment, an application program establishes communication with the Construct using the Common Object Request Broker Architecture (CORBA). CORBA is a standard set by the Object Management Group for communications between distributed objects. By using this standard, application programs may be designed independently and yet interface with the Construct without customization. Each application program creates its own variables and calls functions that manipulate the runtime environment. To the programmer, a geometric object that they are manipulating may appear to exist in the program, but instead they are manipulating a representation of that geometric object that actually resides in the Construct runtime environment. This analogy is similar to abstractions in modern operating systems where programmers believe they have access to a device but in reality they are just manipulating an abstraction of that device. CORBA allows this analogy to go beyond actions on a single computer to allow programmers to manipulate local objects that actually reside remotely.

Various implementations of CORBA exist in the computer industry and are written for various platforms and languages. The API utilizes an appropriate implementation of CORBA to communicate with the Construct, which also uses an appropriate implementation of CORBA. For instance, if the Construct is running on a UNIX based machine and uses an implementation of CORBA written in C and a programmer wants to write an application program in Java on a Windows machine, the development libraries for Java use the Java-specific implementation of CORBA and interface with the Construct effectively.

The development libraries are used to write application programs for the system and define a set of graphics functionality that are carried out by the runtime environment resulting in the images sent to the user. One standard and important developmental tool is the Application Program Interface (API). Programmers writing software for use with the run-time system are required to use the system’s API. This API contains all of the functionality needed to communicate between the program and the system, create and modify elements of a scene graph, and communicate with other application programs. The system API contains abstract graphics procedures from implementation specific libraries such as OpenGL, Performer, World Tool Kit and others. The procedure formats are generic across libraries, without changing the interface to the programmer. All system API classes derive from objects which are the root of all distributed functionality.
The inheritance hierarchy of the API can be seen in Figure 9, showing how the different classes (objects in runtime) are related to one another in terms of function and data inheritance. The API may be supported under C, C++, Java, Perl and several other languages, and may be ported to new languages by implementing a CORBA object request broker for the target language.

One of the tools provided by the system for the application programmer is a framework for designing space managers. The framework includes documentation and specifications for writing a space manager. It may include a code skeleton which lays out the basic functionality requirements for space management. Any manager that fulfills the basic functionality requirements may be implemented with the present system.

**Construct**

The Construct is the conceptual center of the system, and is responsible for the management and display of application programs using the system. The Construct is responsible for displaying the visual state of the runtime environment. Application programs written with the aid of and in accordance with the API for this system, connect to the Construct which manages their display to the user. The Construct is run on the machine that controls the display device (or output devices). The applications may be distributed across different machines on the network. In this way, the system achieves separation between the displaying machine and the machines on which application programs run. The Construct may be implemented in C++ and use the Common Object Request Broker Architecture (CORBA). Other protocol standards may be used, such as, Remote Procedure Calls, Message Passing Interface, DCOM (Microsoft's version of CORBA), and SOAP (Microsoft standard for XLM RPC). Therefore, programs operating in the system only need conform to the system specifications for communication, leaving the system to be responsible for interfacing with the specific graphics libraries (e.g., Performer, WTK, and openGL).

The Construct is an interface between the applications and the display devices. Referring to Figure 2, the Construct 200 includes several functions which are conceptually divided into
components. The Map Service 212 is the component that facilitates the translation between its own identifiers and memory locations, and those from application programs. Implementation module 214, which may be a sub-component of Map Service, provides graphics functionality to the generic CORBA interfaces. Display module 216, which may also be a sub-component of Map Service, maintains the scene graph and facilitates rendition on the output devices, e.g., immersive display device 230. Optionally, the Construct may contain display hardware drivers 218 to take the visualization information from the Display module 216 and render it on the display devices 230. In addition, the Implementation and Display may be structured as components of the Construct without the intermediary Map Service component or with the Map Service as a third component.

In operation, the Construct begins with an initialization process involving test communication between various components within the system, including Map Service and CORBA utilities. If any essential component is missing, the system may report the problem and shut down. Upon completion of the initialization process, the Construct is in a state ready to accept application programs.

Also, during the initialization stage, the Construct generates a basic two-node scene graph (224 and 226). Over the course of the VR session experienced by the user, application programs generate and manipulate nodes 222 and each of these nodes are represented in Implementation 214. The Display module 216 maintains the relationships between and among the nodes, and hence the scene graph. Nodes 222 in the Implementation are added to the scene graph as additional nodes 228. Though not specifically shown, it should be appreciated that applications generate and use objects other than nodes which are also represented and supported by functionality at the Implementation component, similar to nodes 222 but not necessarily represented at the Display component. For example, a file object, enabling functionality involving file management, is an object that is not a node and is not realized in the presentation. See figure 9 and accompanying text for additional examples.
An application program may be run from any CORBA compliant application attached to the Construct directly or via a network. Referring to Figure 3, when an application 300 begins processing in its local address space, it creates a client 310 to interface with the Construct. The client establishes communication with the Construct and requests information regarding the various components of the Construct. This information is stored by the client for future use and the application program is ready to operate transparently with the system.

Referring to Figures 2 and 3, as the application program 300 operates, it creates and manipulates objects 312 representing various elements of the VR environment. When an application program creates an object, the Map Service 212 must be informed in order to keep track of all the objects representing elements in the VR environment. The API automatically interfaces with the Map Service without explicit directions from the application programmer. The Map Service registers the new object and returns a system ID number for the newly created object to the application. Thereafter, the application program interacts with the Implementation 214 via the API which in turn affects the Display 216 which controls the immersive display device sending images to the user. The client 310 created by the application program is associated with the other objects created by that application, and contains shared information that may be required by the other objects.

Referring to Figure 4, a VR session begins at step 400, with the initialization of the Construct which generates a blank scene graph. Substance is added to the blank scene graph by the operation of application programs. When an application program starts, it creates a client object (step 420), and establishes communication with the Construct, (step 422). Typically, steps 420 and 422 are performed once for each application program as it joins the VR session.

While the session may involve a variety of VR functionality, the general process involves creation of a variety of objects, and the manipulation of those objects. To start a scene, the application creates a scene object which is communicated to the Construct. The scene object contains general information (and optionally functionality) about the application and its presentation that may be used by other objects or processes. At step 430, the Construct determines whether the application seeks to create an object or manipulate an object. If the
application is creating an object the Construct then determines at step 432 whether the object to be created is a scene object. If the object is a scene object, the process continues to step 440, where the application program creates a scene object that is mirrored at the Construct. The application program provides the scene object with attributes that describe its presentation within the session. These attributes may indicate, for example, whether the presentation must be close to the user and whether it may intersect with the presentation from other programs. The process then continues to step 442 where step 432, the scene object generates a Pseudo Root which is also communicated to the Construct. The Pseudo Root is the root node of the scene graph from the perspective of the application. At the construct the application’s scene graph is only a subgraph. At step 444, the Space Manager recognizes the Pseudo Root and at step 446 interprets its attributes. The Space Manager may determine whether translation (moving in space) or scaling (changing size) are required. At step 448, the Space Manager attaches the Pseudo Root to the main scene graph. According to the attributes specified in the scene object.

Once a scene is started, other objects may be created and associated with the scene. To accomplish this, the API informs the Construct that the application program seeks to create an object at step 440, and that it is not a scene object at step 442. The application program then creates an object at step 450. The application program provides the object with properties including its relationship within the scene. The API informs the Construct to create the object at step 452 and, at step 454, the Map Service creates an object at Implementation. These steps are repeated anytime an application seeks to create another object. Process identifiers may be used to track the objects and the associations between them, as is known in the field.

In addition to creating objects, the application program manipulates objects according to its design. At step 440, the API instructs the Construct that the application program seeks to manipulate an object, for example, a node of a scene graph. The application program manipulates the object, performing some function, at step 460. At step 462 the request for functionality defined by the object is communicated to the Construct where, at step 464, the Implementation proceeds to realize the functionality, possibly referencing the appropriate graphics library.
Map Service & Implementation

The Map Service which is, a component of the Construct, is responsible for object creation and management. Objects are uniformly referenced by their system object identifier assigned by the Map Service. The Map Service also provides the mapping between the system object identifiers and the memory at the Construct. The functionality of the objects, though "controlled" or "executed" by the applications, are realized at the Construct, where the scene graph and graphics libraries are located. By using the system API, the applications are insulated from the specific details of implementation at the Construct. The Map Service receives the system object identifier from an application and the Map Service proceeds to realize the functionality via Implementation. The Implementation is composed of an interface which declares the expected functionality and the graphics-library-specific implementation which supplies the functionality to the interface using a specific graphics library. To change the graphics implementation that the system uses to generate images, the entire Implementation is replaced when the Construct is compiled.

The API defines the appropriate interface for each kind of object. Referring to Figure 6, each different kind of object 614 is defined by a set of functions 616, generically composed of responsibilities 622. The responsibilities of the various objects are fulfilled by a graphics library 618. The graphics libraries support the functionality defined for the objects and the library may be easily substituted with another graphics library. During operation of the system, objects are created using the previously set definitions and hence supported. However, many instances of the same kind of object may be created. Where there are, for example, two instances of the same kind of object 614, arrows point to the same set 616 to indicate that both objects have the same functionality (responsibilities), by definition. Conventionally, during runtime, objects are called instantiated objects.

During operation, the Map Service assigns system object identifiers to each new instantiated object created by an application program. Referring to Figure 6, when the object subject to the task is new, an object is created, assigned a memory address, and the memory address is sent to the respective graphics library 618 to visually realize the addition of a new object. While the
application program knows the local memory address of the object, it does not know the system memory address located at the Construct. However, the application program uses the system object identifiers to reference the objects when interfacing with the Construct. Hence, when the application calls/executes a responsibility, the application provides the system object identifier to the Map Service which "translates" the identifier into a (system) memory address at the Construct. The Map Service proceeds to relay the address and other information to the graphics library 618. In short, the information about the object being added begins as a memory address in an application program, is mapped to a system identifier and is then mapped to a memory address in the Construct.

For example, a typical function/responsibility 622 of the Transform object, 614, is adding a node to a scene graph, called addChild. AddChild accepts as a parameter an identifier of the node to be added to the scene graph. For example, to add a Text node, addChild is called with the identifier of the Text node. This may be denoted "Transform::addChild(Text)". The Text node is assigned a system object identifier by the Map Service when the Text node is created. In operation, the Transform object calls addChild using the system object identifier. The Map Service then provides the memory address corresponding to the system identifier for the Text node. Finally, addChild 622 references the graphics library 618 to realize the addition of the Text node identified by its memory address to the scene graph.

Display Module

The Display module provides an interface between the applications and the output devices, along with the libraries associated with those devices. This means that to support a new device or library, the Display is replaced with a suitable one. Application programs do not need to know what display device they will present their scene on at runtime. This allows one application to be written for a variety of devices instead of requiring a different application to be written every time the target display hardware changes. This flexibility is possible because of the Display interface, which can be implemented with a variety of different libraries.
Hardware devices supported by the Display module include, CAVEs, BOOMs, HMDs, a variety of tracking devices and flat-screen monitors. Many of the hardware implementations are provided by specific libraries such as VRCo's CAVELib and Ohio State's VRJuggler.

**Space Manager**

To allow application programs to operate and move elements in space without a priori knowledge of their surroundings (the operations of other applications), knowledge of the environment is maintained by a central entity, the Space Manager. The Space Manager stores this knowledge and manipulates the scene graph accordingly.

An important characteristic of the Space Manager is that it is a nonessential part of the system that is implemented with the same tools (API) that are used to make user-level programs for the system. This allows the Space Manager to be removed, changed and restarted without affecting other parts of the system and allows it to exist on any computer able of communicating with the system (Figure 1: 110, 112). The Space Manager is also able to recognize a scene graph previously managed by a different space manager and internalize information about the state of that graph.

As application programs operate to generate VR environments or change the existing one, they create or alter local scene graphs which are really subgraphs of the scene graph in the Construct (managed by the Space Manager). The Space Manager connects the received sub-graphs to the main graph through the root node or navigation node. The root node represents the origin or center of the environment; (0, 0, 0) in an X, Y, Z Cartesian coordinate system. The other nodes are called the navigation nodes. Anything attached to the root node appears to remain stationary and anything attached to a navigation node appears to move with the user's coordinate system. This is because instead of moving the user in the scene, the scene moves around the user. Typically, interface mechanisms (such as a virtual menu for the user to manipulate parameters) are connected to the root node. Since they do not move, they thus remain accessible to the user regardless of the user's position in the environment. On the other hand, typical elements of the
VR environment are connected to a navigation node directly or through other nodes forming a path from the navigation node so that they move naturally as the user navigates the environment.

Once a program is placed in the scene graph at the Construct, the Space Manager need not continually monitor the scene for changes that it will need to act on. Instead, the changes occurring in the application programs register notifications into a queue that the Space Manager monitors. The Space Manager uses this queue to refresh its representation of only the changed parts of the scene. This optimization allows the Space Manager more time to do its most important job, managing space.

The Space Manager maintains an internal model of the scene composed of boundary representations. For example, the boundary representations may be in the form of a sphere or box. The Space Manager uses the boundary representations in its calculations for intersection and in its representation of occupied space. Then the Space Manager updates the scene graph to reflect the calculations.

The Space Manager is an independent program that may be used in conjunction with the Construct to enhance the VR presentation. When the Space Manager is executed, it must first determine whether there is a space manager currently or previously associated with the Construct. The Space Manager process begins with initialization. Referring to Figure 5 at step 502, the Space Manager queries the Construct to determine whether the scene / VR environment is being managed by another space manager. Since the Construct may use or associate with only one space manager at any given time, if there is another space manager in operation, the incoming Space Manager exits the system, at step 503. Provided no other space manager is active, at step 504, the Construct sets up the Space Manager for use with the system. At step 506, the Space Manager queries the Construct to determine whether the environment was previously managed and if so, at step 507, the Space Manager assimilates the scene graph previously generated, generating appropriate internal representations.
Once the initialization steps are completed, the Space Manager's general operations are driven by signals from the applications. At step 510, the Space Manager waits for a signal from the Construct or application program. When a signal is received, the process continues to step 512, where the Space Manager determines what type of signal is received. If the signal indicates the creation of a new pseudo root, the process continues to step 520, where the Space Manager receives the new pseudo root, and interprets its attributes (step 522). At step 524, the Space Manager positions the pseudo root within the central scene graph and at step 526, the pseudo root is attached to the scene graph. If at step 512 the Space Manager determines that the received signal indicates changing a node, the process instead continues to step 530 where the Space Manager changes the node accordingly and at step 532 recalculates the scene graph respectively. After step 526 or step 532, the process returns to step 510 to await another received signal.

The Space Manager makes its management decisions based on attributes that programs choose. Attributes may be divided into groups, for example, intersection, locality and attachment. The intersection attribute describes whether the program can intersect with other programs and may take the values of "exclusive" (preventing the intersection of programs) or "inclusive" (allowing intersection of programs). The locality attribute describes approximately how dense a program's space usage is and may take the values of "environmental" (allowing the program to move throughout the scene graph) or "localized" (limiting the program to move within an area smaller than the entire scene graph). The attachment attribute describes the location of the space in reference to other spaces and may take the values of "attached" (indicating a particular reference point), "detached" (indicating the absence of a reference point), "user_attached" (indicating reference with respect to the user), "x_aligned", (indicating reference with respect to the x-axis rather than a point) "y_aligned" (indicating reference to with respect to the y-axis) and/or "z_aligned" (indicating reference with respect to the z-axis). Typically, the default combination is "exclusive", "localized", "detached".
Interprocess Communication

System-to-program and program-to-program communication is essential for a dynamic and flexible system. These communications are known as Interprocess Communication (IPC).

IPC is required because processes within the system do not share the same address space or even the same platform. Without IPC, it would be extremely difficult for disparate processes to work together, share resources and communicate. Problems are avoided by making the protection of local data the responsibility of local processes, relieving the system of synchronization responsibilities across distributed processes. IPC may be used for one-to-one communication as well as one-to-many (also called broadcast) communication.

One-to-One IPC is implemented using properties, signals, events, and fat pipes. All of these are implemented within the confines of the CORBA run-time system. One-to-one IPC most often takes place with a confirmation that the communication took place, implemented as reliable TCP. A client makes a request to a common interface ORB (object request broker) which directs the request to the appropriate server that contains the object. For low-latency operations, a non-returning call can be used, implemented within the ORB implementation as unreliable UDP (user datagram protocol).

Figure 7 illustrates four generic forms of interprocess communication that are supported by the system: Property, Signal, Event and Fat Pipe. These concepts are somewhat different in principle and implementation from the paradigms in modern operating systems that bear similar names. As described with reference to Figure 3, each program generates a client object to handle general operations and processing. The client objects of the running application programs send and receive messages among each other to achieve interprocess communication. Properties are communications sent by one program 710 to describe itself to another program 712. Signals are communications sent from one program 714 to another program 716 concerning system conditions or instructions to take an action. Events are general communications between
programs (718, 720), indicated with a bidirectional arrow. Fat pipes are bidirectional communications used to transfer large files or set up data streams between programs (722, 724).

**Properties** A property is a distributed attribute that can be accessed by another process. A property contains data that one program offers to other programs. Programs possess data values, which they then export by way of properties. Once a data value is exported, a change in that local value updates the property as well. Properties are often simple types of data structures, e.g., integers, floating point numbers, but may be complex aggregate data structures as well, e.g. lists, tables. An example of a property could be color and a value of that property could be yellow. A property has an associated value and is accessible to any distributed application through the "pull" data model which allows consumers of the shared data to access it at their convenience upon request. The local IPC implementation associates each exported property with a unique string identifier.

**Signals** are used for system related information. Signals are used to transmit information and instructions about termination, relocation, execution, suspension and other process level functionality. Signals have a higher priority than events (described below). A signal is a message targeted to notify a process of a system condition. Received signals are interpreted by the receiving end, which calls the appropriate function according to the signal received. The system defines a standard set of signals and the user cannot define any additional signals.

Signals, like other IPC mechanisms, can be executed asynchronously. Signal operations that change shared data are expected to provide their own mutual exclusion to prevent data corruption. (Typically one process may not access the shared data while another process is about to change the data value.) Signals are push-based, meaning that an application can receive one without any warning. Applications with the proper permissions may generate signals. One process initiates the signal and the other process receives and performs the defined operation. Signals typically do not return a value.
An event is a targeted, definable message that is sent by one program to another program or set of programs. Events have varying delivery types such as guaranteed, unreliable and "best-guess" transmission. Events may be used for non-system related message passing and are freely expandable and usable by applications. An event is a message targeted to notify a process of a user-defined condition. Application programmers may define or even standardize sets of events that their applications recognize and/or send. The runtime system does not define events. Received events are interpreted by the receiving end, which calls (executes) an appropriate function according to the information received in the event. An event that cannot be interpreted by the receiving process results in a null operation - to avoid making both the application and runtime system unstable.

Events and signals are nearly identical in their implementation. Both are carried out using handlers implemented at the Object level which listen to events or signals that come across the network. For every event or signal, its ID is translated into a handler that may or may not be registered with the system. IDs that are received and do not have a registered handler result in a null operation (an operation that effectively does nothing).

The difference between events and signals is conceptual. Signals are used exclusively by the runtime system to inform application processes of essential, critical information. For this reason, signals have a higher priority than events. If multiple events are queued in the handler for execution, a newly arrived signal will take precedence. Signals are generally guaranteed to arrive at their destination(s). The handler will generally execute the defined operation unless fatal behavior occurs between the time of receipt and the time of execution. Signals cannot be defined by an application.

Events, on the other hand, are used for non-critical, application message passing. Any application can define any arbitrary amount of events. These events may be standardized across a set of applications such as "window" managers or may be transient for the lifetime of the application and published to a central authority. Events may also be assigned priorities. Thus a newly arrived event with a higher priority than all currently queued events will be executed first.
The highest event priority is generally not higher than the lowest signal priority since, in general, signals have precedence over events. Events can be defined to ensure delivery and execution or they can be defined to make a best effort at delivery. Best-effort delivery is often useful for non-critical operations such as animation transform updates. Event execution may be deferred to execute an arriving signal because signals have a higher priority than events. Depending on the outcome of signal execution, event execution may or may not continue. After matching an event ID with the associated routine, the handler checks to see if any signals have arrived in the queue. If a signal has arrived, the handler first removes the signal from the queue and processes the signal. If no signal has arrived, or if the intervening signal has non-fatal behavior with respect to the process, the event routine is executed.

_Fat Pipes_ are stream-like connections that can be established between a program and the system or between two programs. These streams are used to transfer large amounts of raw data, such as files or network streams. A fat pipe operates to transfer large blocks of data between different processes. The most common use for this mechanism is transferring 3D models between one location and another. The fat pipe mechanism provides a disk cache management system for processes that wish to temporarily acquire a model for addition to the scene graph for the lifetime of the process. The fat pipe implementation, on application request, acquires the model, caches it, and purges it when either the use of the model is discontinued or when the application is terminated.

The fat pipe operates by formatting the binary data and then transferring it over a pre-defined CORBA channel. The fat pipe also defines quench and suspension operations for use by algorithms managing the efficient flow of traffic on the network. The fat pipe can be told to quench (drop the transfer rate) a large bulk transfer, or suspend the transfer entirely. This is useful when other, prioritized operations such as important signals and events must occur. The fat pipe can either use TCP or reliable UDP to transmit data.

The fat pipe is implemented as a pull-based mechanism. To use a fat pipe, a process must first negotiate the file transfer with another process. This involves requesting a file, communicating
about its availability, transmitting the file, performing compression and checksumming and closing down the pipe. Fat pipes that are used to transmit more than two large data sets between the same processes are kept open until a time-out to eliminate the repeated cost and complexity in creating the connection.

A fat pipe "writes" its data to the disk cache object, which is then responsible for writing the data temporarily to disk or to a memory location. The fat pipe is not concerned with where (disk or memory) the cache writes data. Applications may choose to use in-memory databases and stores to improve performance. If the cache fills, the pipe receives a suspend operation until the cache can resolve the problem, either by a preset behavior or by notifying the user.

All operations on the transmitted data are provided through interfaces. Thus, compression, encryption and binary-to-text encoding can all occur based on specific implemented methods defined at an execution site. A standard binary-to-text implementation is provided as a default behavior.

**Broadcasting**

Figure 8 illustrates the broadcast interprocess communication (IPC) that is supported by the system. Broadcasting allows one program to send data to multiple other programs simultaneously and solves the problem of transmitting identical data to many targets. Broadcasts are critical system events that must reach all processes, or a subset thereof.

One-to-many IPC allows shared communication to all processes (Figure 8). The one-to-many model requires features not included in CORBA, and therefore, this is implemented by way of a UDP network component which is added to the system instead of making modifications to the COBRA specifications.

Another problem with broadcast implementation is that it must be reliable. When the system is shutdown and sends a termination broadcast to all processes, all processes must be guaranteed to
receive it. To address this issue, a set of UDP network objects with simple reliability algorithms is used to accomplish the task of transmitting the broadcast.

**Programs and Programming**

Programmer control of the system is accomplished via creation and manipulation of objects in distributed programs. For instance, a visualization of web-server activity in a CAVE may be implemented by writing a program for the web-server that parses server logs and uses that information to manipulate a scene graph. The manipulated scene graph is a sub-graph of the system wide scene graph used by the Construct, which may be displaying on a CAVE, to render images for the user. This means that a change in web-server activity changes parsed information which, under control of a programmer, affects the display of the visualization in the CAVE.

A set of distributed programs, such as the web-server visualization program discussed above, manipulates a scene graph using a defined set of functionality. This set of functionality is called the Application Programming Interface, sometimes also referred to as development libraries.

Referring to Figure 9, objects are conceptually arranged in a class hierarchy according to functionality. An object 950 may represent any data-functionality combination. There are many different types of objects, for example, file 960, bound 962, color 964, and font 968 are objects with functionality relating to their names. Nodes 970 are objects that may comprise the scene graph, with each object having some functionality in common (making it a node) and some unique according to its name. For example, transform 988, geometry 986, text 984, and light 982 are different types of nodes. With respect to an element in the environment, the geometry node may represent its shape, the transform node may indicate changes in modeling the geometry, the text node may represent text to appear in the environment, and the light node may represent shading and color of the element. The following is a list of typical functionality available to the programmer in generating and changing a scene graph.
**Bound** (962): A boundary representation (sphere or box) which is used in intersection testing, and visibility testing to speed processing. The Bound is used in the Space Manager.

**Color** (964): Color in RGBA (Red, Green, Blue, Alpha) format. This may be used to designate a single color or list of colors.

**Coordinate** (978): A coordinate in Cartesian space (X, Y, Z). This is used to designate points, and in building more complicated geometry such as triangles.

**File** (960): Represents a file which may be used to store complex geometry information, for example, the geometry and color information for a geothermal data visualization.

**Font** (968): A text font which is used in conjunction with Text.

**Geom** (986): A Node which is a collection of Points, Lines, Triangles, Normals, Textures and/or VertexLists which may or may not be indexed. This is the basic building block for creating visible material in the scene graph. Geoms can be created using pre-existing files or alternatively created in real-time.

**Light** (982): A Node which represents a light source of type POINT, SPOT or OMNI. The Light also has color and orientation. The default orientation is at the origin along the -Z axis.

**Node** (970): The basic building block of the scene graph. This object can have children and can be a child. Any object which can be part of the scene graph must be descended (in an Object Oriented sense) from the Node object.

**Normal** (972): A type which stores a vector normal to the surface of a Triangle.

**Scene** (974): The root of the application program's scene graph.

**Text** (984): A Node which when combined with a Font can produce text in the VR environment.

**Texture** (not shown): Image data in RGBA format which can be applied to Geom data.

**Transform** (988): A Node that can translate, scale, rotate and shear its child Nodes.

**Triangle** (not shown): A Node that is a triangle defined by a 3-member VertexList. Triangles can be used to build up complicated geometry and modify the geometry dynamically.

**Vec3** (976): A vector in 3-D Cartesian space (X, Y, Z)

**VertexList** (980): An ordered list of Coordinates, used to build Triangles and Texture Coordinates.
Figure 10 shows an example of an application program 900 generating a sub-graph that may be added to the environment. Building the scene graph may be accomplished with C++ code. The code is illustrated in the lower portion of Figure 10, while the scene graph generated by the code is presented in the upper portion of Figure 10. On line 1, the program begins by creating a Scene (R) 912 which represents the root of the scene graph sub-graph but is only a subtree in the system wide scene graph. Then, on lines 2-3, the program creates various nodes 916 and 918 that are to be used. The nodes created and used in this example are arbitrary; in practice, the creation of specific nodes is dictated by the particular program being designed. Once the initialization is complete, the program adds nodes to the scene graph repeatedly using the addChild routine. In this example, on lines 5-6, two transform nodes T1, T2 are added to the root node. On lines 11-13, three transform nodes T3, T4, T5 are added to the second transform node T2. On lines 8, 10, 15-17, a single geometry node is added with a link to each of the transform nodes T1-T5. Alternatively, five geometry nodes may be added, one geometry to each transform, or some mixture thereof.

**System Performance and Scalability**

The topology of the system is most accurately characterized as a star topology with requests fanning inwards from programs to the Construct. This fanning-in topology is a design requirement that plays a pivotal role in determining the scalability and performance of the system.

The scalability of the system is typically limited by the memory and processing power of the machine that runs the Construct. The system can scale in various dimensions such as:
- complexity of geometry in the system,
- number of active programs in the system,
- frequency of function calls in the system,
- frequency of communication between programs (IPC).

The performance of the system is limited only by the processing power of the graphics subsystem and interconnect bandwidth/latency. The graphics subsystem performance directly affects the user's experience asynchronously from the rest of the system. For instance, if the graphics performance is fast and the system is overburdened with requests, the user will still experience
the environment without jerky response, but changes to the environment will appear jerky. The interconnect between application programs and the Construct (be it co-located or connected via Ethernet) determines the performance of the programs and the user’s experience of the programs. Hence, performance of the graphics subsystem and interconnect are independent.

5

**Hardware Support and Included Technologies**

The system works with a variety of virtual reality hardware. The immersive display devices are fed images by special purpose graphics workstations. These workstations are capable of generating realistic images at a frequency such that the users perceive the images to be a coherent experience instead of a sequence of images.

Aside from display hardware, the system is flexible and supports a wide range of virtual reality related hardware, and is not constrained to specific display technologies, for example, motion trackers. The motion tracker most frequently uses an electromagnetic device that knows its location and orientation in space. These trackers transmit the location and orientation of the user’s head to the program that is rendering the images for the immersive system. This information allows the user to move around within the virtual environment. The system is also designed for use on non-immersive displays such as current monitor technologies.

The system works with a variety of virtual reality hardware because the support depends on the graphics library that is in use. For instance, if a specialized device with a modified version of openGL is used, then the system uses that version of openGL to support the hardware.

Other examples of related hardware with which the system may be used include: Voice Recognition systems such as IBM ViaVoice and Dragon Naturally Speaking; Gesture Recognition using neural networks where a user’s gestures can be interpreted and transformed into functional actions or events; Wireless Tracking Technologies involving stereo matching algorithms providing a wireless solution to the problem of dangling wires in virtual environments; Neurological Electrical Signal Input devices which interpret facial muscle
movement and brain wave activity; Distributed Sensor Data Collection devices for seamless remote data collection and the integration of this collected data into applications; Eye Tracking devices which monitor the movements of the eyes and leverage this capability to improve interface technologies; Computer Vision techniques which generate volumetric models of the user and determine location of various parts of the user's body; Tactile Feedback/Haptics technologies which generate force feedback and physical stimulation; and Audio servers may be connected to the system.

While the invention has been particularly shown and described with reference to preferred embodiments thereof, it will be understood by those skilled in the art that various changes in the form and details may be made therein without departing from the spirit and scope of the invention.
WHAT IS CLAIMED IS:

1. A method for operating a virtual reality system including a central program and one or more applications, the method comprising the steps of:
   initializing a virtual reality environment;
   receiving at the central program a signal sent by the one or more applications indicating a change in the virtual reality environment;
   updating the virtual reality environment; and
   realizing the updated virtual reality environment using an output device.

2. A method for providing a virtual reality environment in which a plurality of independent virtual reality programs are concurrently operating, the method comprising the steps of:
   a. generating the virtual reality environment to reflect a current status of each of the plurality of virtual reality programs;
   b. receiving an instruction from one of the plurality of virtual reality programs; and
   c. updating the environment in accordance with the instruction, and in accordance with the current status of each of the plurality of virtual reality programs.

3. The method of claim 2, further comprising the step of displaying the environment on at least one of a non-immersive screen display and an immersive screen display.

4. The method of claim 2, wherein the plurality of virtual reality programs includes at least one interactive program, the method further comprising the step of receiving an input from at least one input device wherein the input relates to said interactive program.

5. The method of claim 2, wherein one or more of the plurality of virtual reality programs are remotely located, the method further comprising the step of receiving instructions from the one or more of the remotely located virtual reality programs via a communications network.
6. The method of claim 2, further comprising the step of incorporating an additional virtual reality program during the operation of the plurality of virtual reality programs.

7. The method of claim 2, further comprising the step of assigning a corresponding origin in the environment to each one of the virtual reality programs.

8. The method of claim 7, wherein the assigning step is performed in accordance with a user designation, and the method further comprising the step of receiving the designation of the origin from a user with respect to one of the plurality of virtual reality programs.

9. The method of claim 2, wherein at least one of the plurality of virtual reality programs is written in a programming language different from the language in which the construct process is written.

10. A method for providing a virtual reality environment in which a plurality of independent virtual reality programs are concurrently operating, and utilizing a stored representation of the virtual reality environment comprising a plurality of objects, the method comprising the steps of:
   a. generating the stored representation of the virtual reality environment to reflect a current status of each of the plurality of virtual reality programs;
   b. receiving an instruction from one of the plurality of virtual reality programs;
   c. determining the object involved in executing the instruction;
   d. processing the object in accordance with the instruction and in accordance with the current status of each of the plurality of virtual reality programs, thereby updating the stored representation; and
   e. updating the environment by rendering the stored representation.

11. The method of claim 10, further comprising the steps of:
   a. determining whether the instruction involves a new object or a manipulation of an existing object;
b. if the instruction involves a new object, then adding the new object to the stored representation; else

c. if the instruction involves the manipulation of an existing object, then manipulating the object in the stored representation.

12. A method for providing a virtual reality environment in which a plurality of independent virtual reality programs are concurrently operating, and utilizing a scene graph comprising a plurality of objects, the method comprising the steps of:

a. generating the scene graph to reflect a current status of each of the plurality of virtual reality programs;

b. receiving an instruction from one of the plurality of virtual reality programs;

c. determining the object involved in executing the instruction;

d. processing the object in accordance with the instruction in accordance with the current status of each of the plurality of virtual reality programs, thereby updating the scene graph; and

e. updating the environment by rendering the scene graph.

13. A method for rendering a plurality of virtual reality programs in a virtual reality environment comprising the steps of:

a. generating a scene graph comprising a plurality of objects representing the virtual reality environment;

b. receiving an instruction associated with one of the plurality of virtual reality programs, wherein the instruction relates to a presentation of the environment;

c. utilizing a space manager process to determine how the instruction affects the scene graph;

d. updating the scene graph according to the determination of the space manager process; and

e. rendering the scene graph on an output device resulting in the presentation of the virtual reality environment.
14. A method for rendering a plurality of virtual reality programs in a virtual reality
environment comprising the steps of:
utilizing a construct process to interface between the plurality of virtual reality
programs and at least one output device, and wherein the construct process
operates to:
a. receive an instruction associated with one of the plurality of virtual reality
   programs, wherein the instruction relates to a presentation of the
   environment;
b. process the instruction, wherein the processing includes updating a stored
   representation of the environment; and
c. render the stored representation on the at least one output device resulting
   in the presentation of the virtual reality environment.

15. The method of claim 14, wherein the construct process, further operates to interact with
a space manager process, the construct process operates to:
   generate an update request based on the received instruction; and
   place the update request on a queue accessible to a space manager process;
   and wherein the step of processing the instruction is performed by the space
   manager process, the space manager process operates to:
   remove the update request from the queue; and
   process the update request by updating the stored representation of the
   environment according to the update request.

16. The method of claim 15, wherein the stored representation includes a scene graph
comprising a plurality of objects, the method further comprising the steps of:
a. receiving an object representing the update request for a change in the
   environment;
b. adding the object to the scene graph in accordance with the update request; and
c. updating the environment by rendering the scene graph to include the added
   object.
17. The method of claim 16, further comprising the steps of:
   a. determining whether the update request involves a new object or manipulation of
      an existing object;
   b. if the update request involves a new object, then adding the new object to the
      scene graph; else
   c. if the update request involves revision of an existing object, then manipulating the
      object in scene graph.

18. The method of claim 16, further comprising the step of determining boundaries of
    elements of the environment, wherein the space manager process obtains, from the scene graph,
    data for determining the boundaries.

19. The method of claim 15, further comprising the step of processing the update requests in
    a predefined order when the queue contains multiple requests.

20. The method of claim 19, wherein the predefined order is the order in which the update
    requests were received.

21. The method of claim 15, further comprising the step of processing the update request in
    accordance with attributes associated with the virtual reality program from which the request
    originated.

22. The method of claim 21, further comprising the step of utilizing attributes to assist in
    placing and sizing elements of the environment.

23. The method of claim 22, wherein the attributes include an intersection attribute that
    indicates whether elements of the environment associated with the program may intersect with
    elements associated with other programs.
24. The method of claim 22, wherein the attributes include a locality attribute that indicates whether elements of the environment associated with the program are dense or sparse.

25. The method of claim 22, wherein the attributes include an attachment attribute that indicates a reference point or axis from which elements of the environment associated with the program move.

26. The method of claim 15, wherein a presence parameter is set for each virtual reality program, the presence parameter indicating whether the program may be presented in the foreground or background of the environment, the method further comprising the step of processing the update request in accordance with the presence parameter for the virtual reality program from which the request originated.

27. The method of claim 14, further comprising the step of determining whether a space manager process is currently operating with the construct process.

28. The method of claim 27, further comprising the step of replacing the space manager process with an alternate space manager process, such that the starting state of the environment for the alternate space manager process is the last state of the environment for the space manager process.

29. A method for providing a virtual reality environment in which a plurality of independent virtual reality programs are concurrently operating, the method comprising the steps of:
   a. receiving at a construct process an instruction from one of the plurality of programs wherein the instruction identifies an element of the virtual reality environment to which the instruction relates;
   b. providing graphics functionality in accordance with the instruction;
   c. utilizing a map service process for providing a memory address for the element identified in the instruction; and
   d. utilizing a display process for presenting the environment on a display device.
30. The method of claim 29, further comprising the step of utilizing the implementation process to reference one or more graphics libraries.

31. The method of claim 30, further comprising the step of referencing one or more graphics libraries wherein the referencing is part of the process of executing the instruction.

32. The method of claim 30, further comprising the step of replacing the one or more graphics libraries with one or more alternate graphics libraries.

33. The method of claim 30, wherein the one or more graphics libraries are associated with a plurality of functions, the method further comprising the step of replacing the one or more graphics libraries with one or more alternate graphics libraries such that the one or more alternate graphics library are associated with the same plurality of functions.

34. The method of claim 29, further comprising the steps of:
   a. receiving at the map service process a request from one of said plurality of virtual reality programs with respect to an object;
   b. assigning a system identifier to the object such that system identifier is associated with the memory location at which the object is stored; and
   c. providing the virtual reality program with the system identifier.

35. A method for managing a scene graph for a virtual reality environment produced by concurrent operation of a plurality of virtual reality programs, the method comprising the steps of:
   a. initializing a construct process, which in turn initializes the scene graph;
   b. receiving a client object generated by one of said plurality of virtual reality programs;
   c. determining whether the virtual reality program is generating a scene object with attributes, and if so then generating the scene object and attaching the scene
object to the scene graph in accordance with the client object and attributes
associated with the scene object; else
d. determining whether the virtual reality program is generating an object associated
with the scene object, and if so then allocating a location in memory for the object,
assigning a system identifier for the object, and associating the object with the
scene object; else
e. determining whether the virtual reality program is manipulating an existing object,
and if so then locating the object using the system identifier and manipulating the
object with the changes as specified by the virtual reality program, thereby
updating the scene graph; else
f. determining whether the virtual reality program is deleting an existing object, and
if so then locating the object using the system identifier and releasing the memory
for the object from the scene graph; and
g. updating the environment by rendering the scene graph.

36. The method of claim 35, wherein the manipulating step further comprises at least one of
the following steps:
a. selectively adding the object to the scene graph;
b. selectively changing the object in the scene graph; and
c. selectively removing the object from the scene graph.
Fig. 1

LAN / WAN / Internet

Program

Space Manager

Program

CONSTRUCT

Program

Display output

Input device

User

Input device

Program

Program

Program

...
Fig. 3

Program

Client

Object

Object

Object

Object
Fig. 4

1. Initialize Construct

2. Application initialization creates Client object

3. Client object contacts Construct

   - Delete
     - Object informs MapService
     - MapService deletes implementation object
     - Nodes removed from parents
     - Object removed from Map

   - Manipulate
     - Application manipulates Object
     - Object contacts Implementation
     - Implementation realizes functionality

   - Create
     - Scene
       - Application creates Scene with attributes
         - Implementation of Scene generates PseudoRoot
         - Space Manager notified of new PseudoRoot
         - Space Manager interprets attributes
         - Space Manager attaches PseudoRoot to main scene graph
     - Other
       - Application creates Object
         - Object contacts MapService
         - MapService creates new Object in Implementation
Fig. 7

Client

request

eProperty

Client

Property

Client

notify

take action

Client

Signal

Client?

transfer

file

Client

Event

file

Client

transfer

file

Client

Fat Pipe
Fig. 8

- Client
  - notify
  - take action

- Client
  - notify
  - take action

- Client
  - notify
  - take action

- Signal
Fig. 10

Scene Graph

1: eScene R;
2: eTransform T1, T2, T3, T4, T5;
3: eGeom G;
4: 
5: R.addStaticChild(T1);
6: R.addChild(T2);
7: 
8: T1.addChild(G);
9: 
10: T2.addChild(G);
11: T2.addChild(T3);
12: T2.addChild(T4);
13: T2.addChild(T5);
14: 
15: T3.addChild(G);
16: T4.addChild(G);
17: T5.addChild(G);
INTERNATIONAL SEARCH REPORT

A. CLASSIFICATION OF SUBJECT MATTER
   IPC(7) : G06T 17/00
   US CL : 346/418, 497
   According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED
   Minimum documentation searched (classification system followed by classification symbols)

   Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

   Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

<table>
<thead>
<tr>
<th>Category</th>
<th>Citation of document, with indication, where appropriate, of the relevant passages</th>
<th>Relevant to claim No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>US 5,625,576 A (MASSIE et al) 29 April 1997, col.24, line 43 through col.27, line 41.</td>
<td>1-36</td>
</tr>
<tr>
<td>A</td>
<td>US 5,734,805 A (ISENSEE et al) 31 March 1998, all</td>
<td>1-36</td>
</tr>
<tr>
<td>A</td>
<td>US 6,005,548 A (LATYPOV et al) 21 December 1999, all</td>
<td>1-36</td>
</tr>
<tr>
<td>A</td>
<td>US 6,054,991 A (CRANE et al) 25 April 2000, all</td>
<td>1-36</td>
</tr>
<tr>
<td>A</td>
<td>US 6,057,856 A (MIYASHITA et al) 02 May 2000, all</td>
<td>1-36</td>
</tr>
</tbody>
</table>

Further documents are listed in the continuation of Box C.

* Special categories of cited documents:
  "A" document defining the general state of the art which is not considered to be of particular relevance
  "B" earlier document published on or after the international filing date
  "L" document which may throw doubts on priority claimed(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
  "O" document referring to an oral disclosure, use, exhibition or other means
  "P" document published prior to the international filing date but later than the priority date claimed

X Inter document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

See patent family annex.

Date of the actual completion of the international search
20 OCTOBER 2001

Date of mailing of the international search report
14 FEB 2002

Name and mailing address of the ISA/US Commissioner of Patents and Trademarks
Box PCT
Washington, D.C. 20231
Facsimile No. (703) 525-5820

Authorized officer
CLIFF NGUYEN VO
Telephone No. (703) 525-5844

Form PCT/ISA/910 (second sheet) (July 1998)
<table>
<thead>
<tr>
<th>Category</th>
<th>Citation of document, with indication, where appropriate, of the relevant passages</th>
<th>Relevant to claim No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>US 6,064,389 A (BERRY et al) 16 May 2000, all</td>
<td>1-36</td>
</tr>
</tbody>
</table>