A SECOND STEP TOWARDS VIRTUAL REALITY: THE ENTITY MODEL AND SYSTEM DESIGN

Daniel J. Pezely, Mark T. Evenson, Michael D. Almquist and William Bricken
January 1991

ABSTRACT

At SIGGRAPH'90 in Dallas, there was a panel discussing the hip, hype, and hope of virtual reality. A true virtual reality system is still very far away. Systems available today are, for the most part, just interactive computer graphics, but this is only the first step. Here is a design model which is a possible second step. This design gives a platform which researchers will have a great amount of flexibility for creating applications, testing theories, and analyzing gathered information. One major component of the system is that various tools from the artificial intelligence branch of computer science are inherent in the design. Below is an discussion of the implementation design based upon the conceptual design of VEOS from Dr. William Bricken's team at the Human Interface Technology Laboratory at the University of Washington, Seattle.

Categories and Subject Descriptors

C.2 Computer-Communication Networks
C.2.4 Distributed Systems:
distributed applications, network operating systems, distributed databases
C.3 Special-Purpose and Application-Based Systems:
virtual environments

D.2 Software Engineering
D.2.5 Testing and Debugging
D.2.10 Design
D.4 Operating Systems
D.4.2 Storage: segmentation
D.4.7 Organization and Design

E.1 Data Structures:
Lisp packages combined with inodes

H.2 Database Management
H.2.1 Logical Design:
data models
H.4 Information Systems Applications
H.4.3 Communications Applications:
virtual conferencing

I.2 Artificial Intelligence
I.2.1 Applications and Expert Systems
I.3 Computer Graphics
I.3.2 Graphics Systems:
distributed/networked graphics

General Terms: Implementation design

Additional Key Words and Phrases:
Virtual environment platform, artificial intelligence tools, Common Lisp, complete source code to be made public at time of *SIGGRAPH '91* conference.

SYSTEM DESIGN

There has been a longing for tools which are both extremely useful and exceptionally easy to use. Such dreams are not exclusively those of computer science or those of people this century, but the desire has existed in someone's mind for as long as someone wanted to learn and understand information.

Computers have assisted in the processing of information, and with the highly interactive computer graphics available in the recent years, many people have mentioned how much more useful future systems might be.

The individuals and research teams on the quest of virtual reality (VR) have taken the first step: interactive, three-dimensional, computer graphics [3]. The interactivity comes from position-tracking equipment, head-mounted displays, and hand-held or hand-worn input devices. And, of course, very fast and very expensive computational and graphical display hardware is required. In some cases audio, tactile, and force feed-back devices are added to the system.

Some of the drawbacks of this first step are that there is little more to the system than graphical objects and there is little or no provision for multiple environments with multiple users distributed over a network. Other than the audio, tactile, and force queues used in the more advanced systems, there is often no meaning to that data, and mapping coordinates, colors, textures, and sound attributes is the job of the world-builder and not of the participant who is inside the environment. (See [1] for explanations about the notion of the *user* becoming the *participant* and what it means to be *inside* a virtual environment.)

For a possible second step, start with a level of abstraction just above graphical objects: entities. The word means *that which exists*. An entity may be thought of as information which is stored symbolically. Entities are not restricted to graphical object or sound elements but can be anything which can be stored symbolically.

Below is a description of the implementation design of Dr. William Bricken's VEOS project at the Human Interface Technology Laboratory at the University of Washington, Seattle. See [2] for descriptions of other projects at HITL.

Entities of Symbolic Information

The *Entity Model*, then, is the idea that entities dominate the virtual environment system. That is, information determines the flow of execution of the system. And, since information can come from input devices (such as 3-space position tracking devices) and information can be sent to output devices (such as rendering systems) which are both controlled by the participant, the previous virtual environment systems may be implemented within this model.

In addition, this design model allows for information originating from other sources to be mapped into the environment and to have control of entities.

Again, entities are information. But, to interact with entities, the atomic elements of that entity's information are mapped into the components of renderable data or from the components of acquired data. Here, the term *render* should be thought of as its real-world meaning (i.e., to make or to translate) and not just as its meaning in computer graphics.

The atomic elements of information are symbols. Symbols may be words or strings of characters or any value which may be used to identify and distinguish one piece of information from another. Since multiple symbols may refer to the same information, entities are compared by their meanings or their values.

Drawing from the artificial intelligence (AI) branch of computer science, a good way to store symbols and values is with lists. And, one of the better ways to work with lists is with the Lisp programming language.

Lisp gives a clean mechanism for storing information when that information could be data or functions for evaluation: both have symbols to identify them. In Lisp, there is no distinction between the different types of symbols at the user level [9]. Internally to the design of a Lisp system, the data structures for storing variables and functions can be the same. In this implementation design, the information about the *data objects*, in Lisp terminology, is stored in the same way a user would store the information: in lists. Therefore, the user or the user's program can access the information about the data object by accessing the list which the symbol's definition is in. All values for kernel system variables are held in lists with regular variables referencing the exact position for run-time efficiency. This way, the user may obtain any information about the system kernel which the system kernel can itself obtain.

The System As A Tool

Since using Lisp is convenient for managing the information within a system, a full Common Lisp implementation might be provided, thus allowing researchers to be able to use the system for both managing virtual environments and providing for various tools such as neural networks, inference engines, expert systems, hypertext engines, and artificial life systems.

With such tools at the disposal of anyone using this design, researchers will hopefully better understand VR, its applications, and its potential.

Not only can tools be created and used with the system, but the information can be modified while the system is running. The components of the information which get mapped into the components of the display data can vary and be dependent upon other information available to the system. With the programmability of Lisp, the data being displayed could change without having any further input or commands sent by the participant, all by functions remapping different information to the display data.

Now, the virtual environment is no longer dependent upon the participant for commands. In fact, the environment can run without any involvement of the participant, beyond starting the system. This makes for a good platform to run artificial life projects where the researchers only need to peek in and observe the organisms occasionally while the domain is active.

The environment could even be controlled by a remote system sending commands rather than having a local participant or local programs running within the environment. The remote system might be another virtual environment with another participant visiting the local environment, so there is the flexibility of multiple participants and multiple environments. The remote system could also be fed by data acquisition equipment. This idea of remote access allows for virtual environment servers.

One concept which is crucial to this design is that the system is a platform for building applications upon. This platform is similar to an operating system of a computer in that it manages resources and gives applications a higher level of abstraction to use than the underlying hardware.

However, this design provides for a system-independent platform and not just a hardware-independent platform. So, implementations on different models of computers by the same vendor will function the same, and implementations on different vendors' models of computers will function the same.

The DataSpace and Messages

Lisp controls the system's flow of execution, but the heart of the system is the *DataSpace*. This is where all information is stored internally in the form of lists. Parts of the DataSpace may be in main memory (RAM) or in secondary memory (disk drives) or on remote database servers. Some lists might have to always be located in main memory, but most lists could be located anywhere. The location is transparent to anyone or any application accessing the DataSpace since a request for access will be handled by the system without the need for specifying which piece of hardware to use. This is similar to the access methods of the UNIX file systems structures [7]. The DataSpace may be segmented to reduce the complexity of searching as well as allowing other features explained in the section called Divisions of the DataSpace, below.

The DataSpace contains all input/output (i/o) messages. Messages are lists. The added distinction of a *message* is simply to denote the fact that the list has traveled or will travel through an i/o channel.

Sources and destinations of messages could be any device or channel which can be read from, written to, or otherwise have information obtained from or sent to.

Common channels are serial devices (modems, pointing/tracking devices, data-acquisition equipment, etc), disk file systems (configuration files, script/batch files, saved-environment files, etc), sockets [7] (internet-domain communications, interprocess communications, remote procedure calls, etc), and terminals (console, graphic workstations with virtual terminals, etc).

Distributed Systems Capabilities

One i/o channel which is extremely important, is calling functions within the application and among remote applications. Since the system can handle i/o from internet-domain sockets, any two computer systems on the same network can communicate with each other. And, since the internet ties so many various networks together, a world-wide virtual environment is theoretically possible, although realistically, improbable with today's network bandwidths.

To give researchers the greatest flexibility, the Lisp evaluator may be any Lisp system external to the system kernel. In fact, any interpreted language system may be used by creating a single, very simple, Lisp function which passes a list on to another program. This program may be on the local machine or on a remote machine.

The routing of messages is handled by resolving symbols to be Lisp functions which, with abstracted names of the respective devices, in turn, call routines built into the system which pass the evaluated parameters to the first function on to the i/o channel. This is known as using *stub routines*. The routing is done by the *Entity Evaluator*.

Since messages are lists, and lists can be stored information or calls to functions which control the flow of execution within an application, and multiple computer systems can communicate, the message-passing mechanism provides for a distributed system.

Message-passing and distributed systems have been seen to be a key component in the future of computer systems [5][8][10].

The Entity Evaluator

The core of the evaluation mechanism is match and substitute. Although a seemingly simple concept, match and substitute is one of the core operations of inference engines and expert systems.

Substitution might be as simple as replacing one symbol for another, or it might be replacing one symbol for a list of symbols. Substitution could also replace the matched symbol with the results of a call to a routine with the name of the matched symbol.

Remote Procedure Calls

Remote function calls (often referred to as remote procedure call or RPC's) are made through the use of stub functions: routines which send the parameters of the called function to the remote system to be evaluated. This gives the illusion of the everything being local. This is a standard technique used in today's operating systems [7][10].

Stub routines tack on additional information so the remote system can return the results back to the calling system. The receiving system deals with the trailing information by hiding it from the function which is to be called. Since lists may contain sublists, the parameters are in a sublist, and only that sublist is passed to the function for evaluation.

To decrease the overall amount of messages to be passed from one system to another, an application should make good use of the Lisp engine in the remote system. If routines with long parameter lists need to be called repeatedly or if multiple routines need to be called in succession, then the system making the remote calls should form simple routines to be created on the remote system; these routines should have all of the parameters that will not be changed already set so a more compact message can be sent. This will decrease the overall bandwidth of communication and potentially increase the response time.

Message Content

For the most part, run-time performance of this design has not been discussed. Since Lisp is an important part of the design, it may have been assumed by the reader that all messages pass characters (i.e., from the ASCII character set) which form symbols which are put in lists. For most systems, that may be the only case that needs to be dealt with.

Again, a list is information stored symbolically, and the symbols may be any value which may be used to identify and distinguished one piece of information from another. That means symbols can be characters, integers, or any bit pattern. Many interfaces for programs only allow characters which may contain letters and digits but not integers, unless those integers happen to be the ASCII equivalent of something which is allowed.

If that limitation is removed, integers, floating-point numbers, and signed values may be used as symbols. We may remove this limitation because some symbols need not be read by humans, or when a human might need to read it, a tool will be used to make the necessary translation.

The main reason for allowing the full range of values to be used as symbols is that now more types of information may be passed as messages.

[Implementation note: An important note for implementations is that symbols should stored separately from the variable which specifies the length of the symbol. In this sense, neither of the traditional Pascal or C string data types will be used so that the full symbol is used for storing values of that symbol and not a size value or a token character to specify the end. The main reason for this is that the system will be written in C or a C-derivative language, so Pascal strings would cause unnecessary complexity, and C strings which use terminating null characters may interfere with the actual symbol being stored.]

For run-time efficiency reasons, it is better to use system-dependent object code generated by compiled and optimized source code than it is to use un-optimized source code which needs to be interpreted.

One solution to that problem is to allow the transmission of system-dependent object code through messages. By reversing the suggestion for multiple remote function calls, multiple messages of object code fragments may be sent as unique lists, and then a concatenation routine can be sent which joins all of the fragments of the object file.

In some cases the system kernel will be set up to dynamically link in object files while the system is running. The Free Software Foundation's GNU Dld dynamic linker handles such tasks [6].

Divisions of the DataSpace

Since there is a Lisp engine designed into the system, scoping issues need to be discussed. In Lisp, each function has its own scope for resolving symbols so that if a variable is used internally to a function which happens to have the same symbol as another variable or function name, the local symbol should be chosen.

To satisfy this and to divide the DataSpace into manageable segments, subspaces may be used.

All messages are held in the WorkSpace. When the system starts up, the WorkSpace is completely empty. The first thing the system kernel does is to translate start-up parameters from the command-line, menus, or whatever, into messages. The first message in the WorkSpace is always the start-up message which gets substituted with the sequence of commands which should be used to configure the system.

To compare this with UNIX systems, this is similar to a login shell automatically looking for a .login file and evaluating the contents of that file.

To separate the i/o messages in the WorkSpace from the system library symbols, there are SysLibSpaces: system library spaces. There may be any number of these spaces. Some SysLibSpaces might be for system kernel routines, Common Lisp routines, utility routines, etc. The routines in these spaces cannot be modified individually; however, the entire space may be purged from memory and another SysLibSpace may be loaded. This idea of a dynamically linked space is explained below.

To separate application routines from the system libraries, there are other spaces called UsrLibSpaces: user library spaces. Such spaces are identical to the SysLibSpaces with the exception of the SysLibSpaces being the system default libraries.

The space called the *UsrLibSpace* will be a list of all the spaces used for user applications. If the participant has loaded three applications, called VR-Emacs, CAD, and VideoFeed, the UsrLibSpace, will have references to three spaces called UsrLibSpace-VR-Emacs, UsrLibSpace-CAD, and UsrLibSpace-VideoFeed. The *SysLibSpace* and *SysLibSpace-* spaces work in the same fashion.

[Implementation note: When a symbol needs to be resolved, no matter which space it is in, the first space to be searched is the one in which the symbol is in. Then, the default is to search the WorkSpace, the UsrLibSpaces, and finally the SysLibSpaces in the reverse order in which they were loaded. This allows redefinitions of library routines at any level, including redefinitions of redefinitions.

A simple mechanism is provided in the design which allows absolute addressing so that the system default libraries can be reached, no matter how many redefinitions have been made. This is just the Lisp reader not allowing redefinitions of this function's symbol.]

Additional spaces are for constants and free-list management of symbols and list terms. There is the ConstantsSpace, SymbolFreeList and FreeList, respectively.

A space definition is composed of its symbol (name of the list) mentioned above, followed by the list of symbols which are in that space. From the DataSpace level, spaces can be thought of as indices which are an extra level of indirection to data, which only contain the necessary references so as to make queries less complex.

The names of these spaces are used internally to the system kernel as effectively global variables for run-time efficiency reasons. Each space is a subspace of the DataSpace, so by starting at the beginning of the DataSpace, every symbol and every list can be found, no matter which space it is in.

This provides for a very useful debugging tool for programmers who wish to modify the kernel source code or modify the system at run-time.

To further this idea of using the system as its own debugging tool, all symbols allocated by the system can be linked together in the StringSpace so as to assist in possibly finding run-time program errors. This is the one space which may be deactivated by recompiling the system kernel with certain flags omitted. The amount of memory required for this feature could, in most cases, be used for more useful things.

%%

In **Figure 1**, there is an example DataSpace of a system kernel just after it has started up. Upon initialization of the kernel, the WorkSpace is completely empty. Then, just before the main loop of the program, a *startup* message is placed in the WorkSpace. The default SysLibSpace should have a *startup* symbol for the evaluator to match on, and the respective function should get executed which might load the contents of a script file into the WorkSpace as the substitution of the *startup* message at evaluation.

%%

Dynamically Linked Spaces

By using run-time or dynamic linkers, it is possible to read in compiled object files into the data segment of the system memory and effectively execute data.

This allows for applications to be precompiled for run-time efficiency and/or security of program source code. One application may be read-in as a single space or even as multiple spaces.

Whenever spaces are read-in, information about where the subspaces are located in the object file is required or a key symbol is needed to be searched for. So, this feature of the system design is a slight modification of the available dynamic linkers.

The imaginative reader would notice that when dynamic spaces are joined with message passing where the message content is object files, this provides for a clean way to *remotely* upgrade the system for a new command library since the system kernel need not be shutdown and restarted locally.

Data Structures

The composition of the *Head* of the list and the list formed by *Term* structures may be thought of as a file system if the head structure is compared to the *inode* of BSD UNIX. The structures may evolve completely into a file system structure since the DataSpace is to be shared among multiple kernels or multiple readers and/or multiple parsers in a single kernel running on a multiprocessor computer [7].

To see the kernel data structure for lists, refer to Figure 2.

Graphical Applications

The discussion of the system up to this point has dealt with the design of the kernel of the system, and such a design may be compared to that of an operating system: on top of this operating platform or environment is where the applications will reside.

Applications, from the most basic to the more complex, will include computer graphics so people can see and understand the information which they are examining. There are many reasons for this.

%%

```
typedef struct _Head    Head,  List;
typedef struct _Term    Term;

struct _Head  {
      unsigned short    links;      /* Num refs to list */
      unsigned short    flags;      /* Locked?, Modified?, etc */
      Term              * head;     /* List: chain of Terms */
      Term              * tail;     /* End of list */
      Head              * next;
};

struct _Term {
      union _Form  {
            unsigned    * symbol;   /* Name or value */
            Head        * list;     /* Nested list */
            int         (*fn)();    /* function pointer */
      }                 form;       /* Form of this term */
      int               type;       /* Which Form to use */
      unsigned int      size;       /* Array of any form.xx */
      Term              * next;
};

extern List * DataSpace, * WorkSpace, * UsrLibSpace, * SysLibSpace;
```

Figure 2: This list structure is a slight variation on the internal representation of a Lisp list.

%%

For a long time, as can be seen in the root of virtual reality and computer graphics, humans have liked to see things pictorially. From the early days of graphics, researchers have experimented with head mounted displays to obtain yet a better understanding of the images presented on the graphical displays. So, it is no wonder that steps are being taken towards VR's total immersion concepts.

Graphical Objects

Getting back to the notion that everything is an *entity* and that all lists of information are entities, the information which the rendering system deals with is graphical components of entities, lists of data to be displayed which came from other lists, which may also be considered entities.

A list of data to be displayed is a sublist of an entity within the virtual environment. The sublist, which is actually a list, since lists are nestable, may be generated by functions within an entity. This sublist is often called a *graphical object* in the scheme of the Entity Model to distinguish it from a more expanded *entity* which might be more than just graphical information.

Again, an entity may have graphical information within it, but graphical information is not necessarily an entity. This distinction is extremely important for the Entity Model.

Since the graphical objects will most likely be changing constantly--- changing scale, position, shape, location, or any attributes like color--- a representation for polygons and geometric objects needs to be used in place of arrays of points with arrays of indices referencing points of polygons. Although the array model might be good for transmission and storage due to its compactness, modifications to the data may be too complex for the needs here.

Many of the changes which will be taking place inside a virtual environment may not be observed by a participant because the environments may be far too complex to observe everything at once (e.g., the participants may be in another part of the virtual environment and cannot see the object which changed). Instead, the changes will be made and then stored. There may be many changes made before a participant has a chance to make any observations.

One current design for handling planar polygons and geometric objects has a *head* vector which locates the origin of an object from some other origin and n vertices from the origin of the object to each vertex of that object.

Various data structures and animations techniques are being studied and implemented and will be benchmarked for overall performance, flexibility, and transmission qualities. After all, the graphical objects will be sent over a high traffic computer network connecting the virtual environment server with the rendering system.

The Render Entity

The Render Entity is the rendering system as seen by the virtual environment server. The components of the Render Entity are called entities, children of the Render Entity, and are actually programs which handle the various stages of the rendering pipeline [4]. This allows for parallel and distributed processing of the graphical objects inside the virtual environment.

Note that the rendering system will include facilities for non-display output such as tactile feed-back, audio output, etc, but only the graphic objects are discussed here.

The Render Entity does not have any knowledge of the input devices or even any knowledge of the participant. The Render Entity is known to the virtual environment server and is told what graphical objects it has, when those objects are modified, and when to render the objects. If the participant has multiple points-of-views for stereoscopic views, multiple Render Entities must be present.

To allow inexpensive hardware to be used for the rendering workstation, the Render Entity will only be told about the objects which are in the field-of-view of the participant. This also allows laws of physics to operate on the entities within the virtual environment and keeps the rendered image from displaying potentially false information.

As far as the Render Entity is concerned, the graphical objects are entities since they are lists of information; however, these entities should not be confused with the entities described elsewhere, for they are restricted to only graphical information. Again, there are also renders and objects for non-graphic display systems as well.

Although the rendering system will still have to perform clipping and 3D to 2D transformations, the objects which should not be seen by the participant will be filtered out by the virtual environment server. Although this increases the complexity of the server's job, the decrease in bandwidth will eventually make this a necessity.

By keeping the complexity down, very low-end equipment may be used in the pipeline. Benchmark results should be available in the summer of 1991 to evaluate performances of various configurations.

SUMMARY

Everything is an *entity*. Entities are lists of information which are stored symbolically. Some entities have explicit executable and functional capabilities while others may be used strictly for data values. The Entity Model for a second step towards a true virtual reality system provides for parallel and distributed processing, allows for multiple virtual environments with multiple and remote participants, and gives researchers a full implementation of a Common Lisp interpreter and possibly use of any source code which can be recompiled for dynamic linking. Just as the Lisp engine may be completely external to the system kernel, so too can the applications, rendering system, and input device controllers. The only requirement for communication between entities (and hence device drivers, remote systems, etc) is compliance with the protocol which is the syntax of the Common Lisp programming language with a few additional keywords.

BIBLIOGRAPHY

[1] Bricken, M.: *Virtual Worlds: No Interface To Design*, To be published in *Cyberspace*, MIT Press, 1990.

[2] Bricken, W.: *Virtual Reality: Directions of Growth*, Notes from the *SIGGRAPH '90* Panel: *Hip, Hype, and Hope---The Three Faces of Virtual Worlds*, Human Interface Technology Laboratory internal document, 1990.

[3] Foley, J.D.: *Interfaces for Advanced Computing*, *Scientific American*, 1987.

[4] Foley, J.D., Dam, A. van, Feiner, S.K., Hughes, J.F.: *Computer Graphics* Second Edition, Addison-Wesley, 1990, pp. 201-282.

[5] Forin, A., Barrera, J., Young, M., Rashid, R.: *Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach*, To appear in the *1988 Winter USENIX* conference, 1988.

[6] Ho, W.W., Olsson, R.A.: *An Approach to Genuine Dynamic Linking*, Included with the GNU Dld system, Division of Computer Science, University of California at Davis, 1990.

[7] Leffler, S.J., McKusick, M.K., Karels, M.J., Quaterman, J.S.: *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Addison-Wesley, 1989, pp. 13, 203-208, 281-310.

[8] Pike, R., Presotto, D., Thompson, K., Trickey, H.: *Plan 9 from Bell Labs*, To appear in the 1990 *UKUUG* Conference, 1990.

[9] Steele, G.L.,Jr.: *Common Lisp* Second Edition, Digital Press, 1990, pp. 12-13, 238-246, 247-287.

[10] Tanenbaum, A.S., Renese, R. van, Stavern, H. van, Sharp,, G.J., Mullender, S.J., Jansen, J., Rossum, G. van: *Experiences with the Amoeba Distributed Operating System*, *Communications of the ACM*, vol. 33, no. 12, 1990, pp. 46-63.