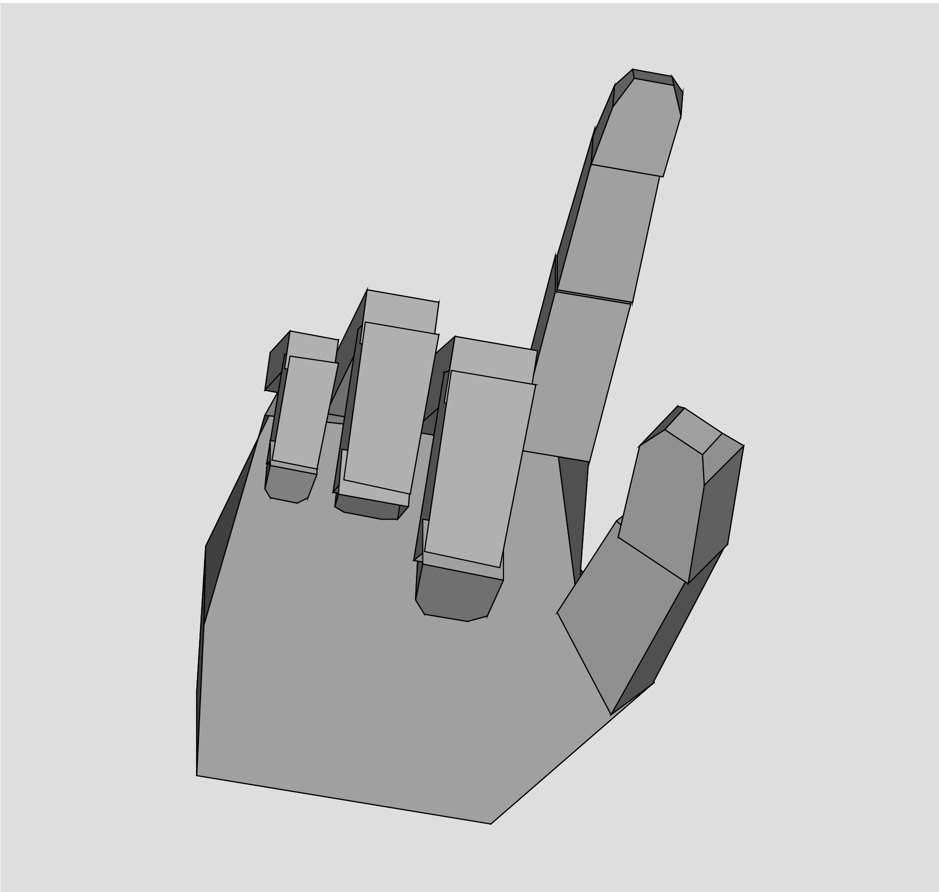
Implementing Virtual Reality



Jeremy Lee

July 17, 1992

Introduction	**1**
The Goal	**1**
The Problem	**1**
Basic concepts	**1**
Virtual Worlds	**2**
Object Definition	**2**
Process Definition	**3**
Distributed processing and Networking	**3**
Languages	**4**
Processes	**4**
Links	**4**
Memory usage	**5**
Message passing	**5**
Process Identification	**5**
Interaction standards	**5**
Touch	**6**
Sound	**7**
Vision	**7**
Ownership	**8**
Security	**8**
The Death of ’God’	**8**
Database Models	**9**
Syncronicity	**9**
Measurement	**10**
Rendering	**10**
Patch-Process relations	**11**
External Interfacing	**12**
Conclusion	**12**
Copyright Notice	**13**

Introduction

Presently, I have no catchy name, so I'll just have to stick to dry names like "The System" or "Rendering process".

The main problem in the field of VR today is the fact that everybody is doing it. And when everybody does it, then everybody has their own ideas on how things should be done, and everyone does it differently. Eventually, one (or two) standards emerge.

In the past, this hasn't been too much of a problem, and workarounds have been devised because in most cases, people have converged on a similar, if distinct, solutions. VR is not like that. No-one is even presently sure what VR is. This will lead to confusion, vastly different standards, and implementations. Once again, this is not too much of a problem in other areas, but one of the goals of VR is networking, the ability to plug dozens, hundreds, or thousands of people and realities together. If this is to be done, then some standards have to be worked out to allow this to happen. This is probably not such a standard, but it is hoped that the concepts in this paper will somehow find their way into researchers minds and leap to the fore when confronted with a particularly tricky situation.

The Goal

The basic Goal of VR is to produce an environment that is indistinguishable from reality in which certain things can be done or experienced that cannot normally be done. A commercial flight simulator is an example, in that people crowd into a mock cockpit, and a massive computer projects images onto large screens, while the entire assembly is rocked about. To the people in the cabin, the illusion is often complete, and they totally believe that they are flying a real plane. What can now be done is trial of emergency procedures, and extraordinary situations can be tested and tried, without danger of destroying property or killing the crew.

VR seeks to take this one step further by removing the physical "set" and replacing it with direct sensory input, such as vision, sound, touch, etc. Current technology has progressed to the stage where Vision and Sound can easily be simulated with sufficient computing power.

To translate this into computing terms, an interactive, multi-participant system must be built.

The Problem

The problem is that there are far far too many ways that this can be done, and everybody seems to be going at it from the wrong direction, from the screen in. Current rendering technology is being used as the basis for building a VR, and although this produces "results", it is a short-term system that will reach it's limits very quickly.

What has to be done is that an entire paradigm must be invented that takes care of all of the aspects of a VR, treating the renderer as just a part of it, indeed, as just a small part.

To begin with, we must base our system on reality to some degree. The concepts that we choose to base it on will affect the way the entire system is set up, and so they much be chosen carefully. Or, alternately, a system must be set up where these choices are made unimportant. The internal goal of the system must be to make an environment that is as flexible as possible, and one in that (almost) anything can be done.

Basic concepts

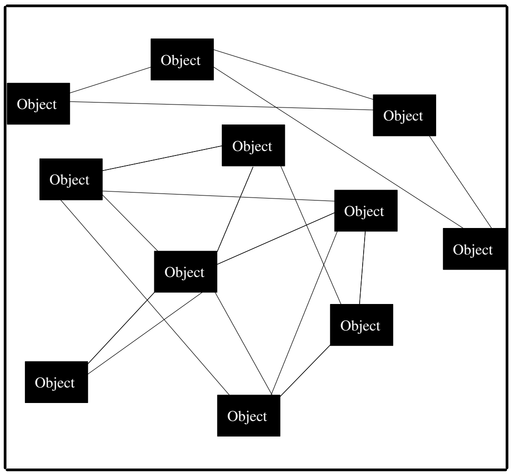
I have taken one basic concept from reality, the Object. The way that the object is viewed and heard can safely be ignored for the present. Suffice to say that each object interacts with another object in some way. This is simulated by each object opening a **link** with another object. This link is a communications channel, through which information can be passed between objects. This link is actually a logical extension of a certain component of an object, discussed later.

Everything that occurs between two objects can be considered an interaction, which again can be considered to be information passed over a link. Seeing an object is an interaction, so is touching it, smelling it (if we had the olfactory technology) or anything else.

This is the only definition. Every interactive "unit" is considered to be an object, including the (multiple) observers in the scene. Each object is capable of interacting with any other object.

Virtual Worlds

The fact that any object can interact with any other object suggests a few problems. When several thousand objects are active at one time in a networked environment, how do you partition yourself to interact with only a specific group of them? Some have suggested the concept of a "Virtual World" in which you join the world and leave the previous one, jumping between them as necessary. Others have suggested a similar idea using rooms and doors, in which leaving by a door is equivalent to moving between virtual worlds.

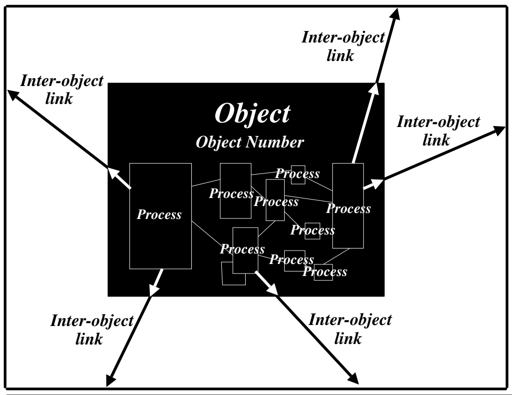


Two groups of objects share different "virtual Worlds"

I would extract the basic premise from these descriptions and say that the goal here is to choose a sub-group of objects to interact with. All this requires is for a group of objects to agree to exclusively interact with each other. there is no need for distinctions like "rooms" and "worlds" when all you have to do is decide to only interact with a small subset of all available objects. One suggestion was that the concept of "rooms" enabled security, and certain doors would be "locked". In the wider view, objects simply have the right to refuse to pass information with un-authenticated objects. If every object in the "room" shares this feature, then even if someone manages to enter the "room" then they still cannot interact with the objects. Even if an object is taken outside the room then they still cannot interact with it. Each object is responsible for it's own security.

You will notice that at no point has the concept of a central controller been raised. This is because one is not necessary, in an object centred view. Each object must be capable of taking care of itself.

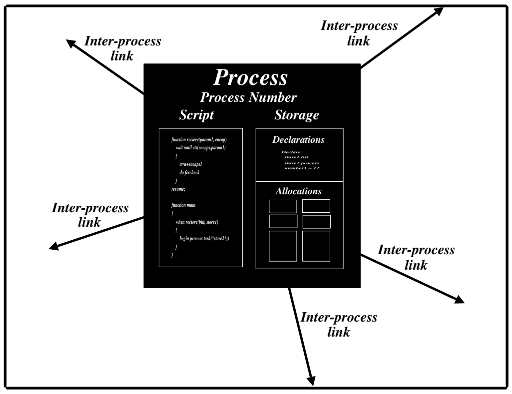
Object Definition



An object is defined as being comprised of a number of processes. Each process sends messages to other processes. Inter-object links are simply extensions of inter-process links that occur between objects.

The grouping of certain processes into an object is really arbitrary, and one could supposedly do away with the idea of objects entirely (which in fact don't really exist outside of the definition that they are composed of a number of processes) except that they are a convenient method of grouping processes into functional units.

Process Definition



A process is basically a task, executing in some named language. In fact, a task could be defined in any language as long as a few criteria are met:

1. Each process is capable of establishing communication links with other processes.
2. Each process is capable of starting another object/process
3. Each process can be moved, mid-execution, onto another processor.

The first two requirements are simple. The last one is not, and to my knowledge, has not been investigated before.

Can a compiled language be shifted, mid-execution, to another processor? And I don't necessarily mean a processor of the same type. I am talking from an IBM to a SparcStation. In a multiprocessor environment, an object may be copied, or moved, onto another processor. While copying the object could involve re-compiling and restarting the process, moving it requires that all internal structures/variables, and execution state be preserved.

Example: If each process was a Unix process running in C:

In this example, each process would be capable of establishing a communication link with other processes via library functions and standard unix calls. If the definition of a process extended to include the source code, then a process could start another process by issuing a command to compile it and start the process, passing it some initial information. It would even be capable of copying and starting itself on another processor but it would fail the last requirement of being able to be moved, while running, to any other processor.

This requirement of being moved also creates a few other problems, which can be solved with a little work. First, during the move, a process will be inactive, and incapable of responding to messages. A queue will have to be set up to store these messages. But also, once the object has moved, then messages passed to the original processor have to be passed on to the new processor, and all other sending processes should be notified that the process has been moved. We have the same situation were someone is moving house, and something has to happen to their mail.

Why do we need this third requirement, which is making life so difficult?

In a distributed system, each object is comprised of many processes. To distribute the load evenly. processes have to be shifted and moved among the processors available to the network. This allocation is done transparently and separately to the individual processes, so that distribution algorithms can be improved and site-specific parameters can be taken into account.

If processes cannot by dynamically distributed amongst the multiple processors available, then processes will get "stuck" on certain processors, resulting in an uneven distribution and wasted processing time. And what happens when a processing node has to be shut down? The object **has** to be moved then.

Distributed processing and Networking

The fundamentals for networking are already in place, and no doubt specialist networks will be evolved for VR in time. Any networking details will be taken for granted.

Likewise, since processes are defined as concurrent tasks, which communicate in a very

OCCAM like way, the basics for fine-grained multiprocessing are in place. The complex tasks of process-processor allocations and linking is left to the site/machine dependant VR operating system that individual sites are using. This area has already been extensively researched, and an appropriate method is no doubt currently sitting on someone's shelf waiting to be dusted off and put into place.

Languages

Any language that the VR supports must be capable of performing the above mentioned tasks. I can only guess at what such a VR language will look like.

It will probably not be a current language. The requirements are beyond the specifications for any current language. The saving grace of the system is that multiple languages can be defined and implemented, as long as they can communicate, start other encapsulated processes, and be encapsulated. Of course, if one machine cannot handle a process defined in a particular language, then it will have to be run on another that can support it. Eventually on or two languages should emerge.

As long as they adhere to the rules that the OS lays down, there should be no problem.

Processes

As mentioned earlier, processes are defined in some language, and must perform a few basic tasks such as linking to other processes, starting new processes, and being encapsulated for transmission.

Whatever language it is, must be available on **all** VR machines that are going to want to support the object in any way. Multiple or alternate languages may be produced, but objects defined with processes of this type will not be portable (or in some cases viewable) on machines that do not support this language. Any language produced must be implemented in the vast majority of VR systems before it begins widespread use, as incompatibility problems will definitely eventuate with older systems.

Of course, there will be cases when specific processes will be tied to particular machines. For example, processes that control external pheriperals like gloves, speakers, and displays. These processes must be marked as "immovable" in some way. No doubt, some people will take advantage of this facility to define immovable processes that do specific things, but objects built from these processes can never be copied, or moved. Again, this is fine for some applications.

Links

Links are object pointers that can be changed without the processes being completely aware. An object can open a link (in much the same way as opening a file) and then later redirect or close the link, or open new ones. All subsequent code needs to know is the internal handle of the link (in the same way that file code uses and internal file handle that is created from opening a named file) to do operations.

Opening a link does not necessarily mean that during the life of the link, that information is continually flowing. The operating system does not continuously check the validity of the link (although it will do it's best to make sure it is current and correct) although the objects are perfectly at liberty to do so. It is even possible to open a link, and then close it, without a single byte being sent.

To open a link requires knowledge of the object/process number of the process at the other end. Links are one-way, and if the remote object wishes to respond, then it must open a link of it's own. Information is sent down the link, and the sending object must not expect a reply, but should be able to act on one. Deadlocking in such a distributed system may be difficult to correct.

Memory usage

In a VR system, how is memory allocated? First, there is no concept of a file. All available disk space should be dedicated to use as one large Virtual Memory. If you require the same functionality as a file, then create a process/object that acts simply as a data storage block, and sends/receives this data on request.

All objects are active at any one time, but standard VM algorithms should ensure that relatively unused portions of processes, such as large memory stores, stay relatively inactive on disk.

Message passing

When a message is passed down a link, what form should it take? It will basically be one large block of data, roughly equating to a spooled file or a pipe, for the remote process to do with as it will. Traditional handshaking and error checking will be done transparently by the OS.

This method is by far the most flexible and future-proof. Processes can encode these blocks in whatever format they require, but every object will be capable of sending, receiving, and passing on these blocks, even if they are unaware of their content. Of course, most languages will have standard commands to insert and extract data from these variable length message blocks, most likely following similar methods as standard file operations. Most languages should also treat each block as one largish variable or string, which is dynamically resizeable.

Process Identification

Each object is somehow given a unique ID number, that no other object can possibly share. I still don’t know how this can be done, and any suggestions are welcome.

Within each object, processes are given unique process numbers, which is easy. If a process wants to open a link inside the object, the it simply uses the process number. If it wants to open a link with another process outside the object, then it also has to give the object number. Process numbers should not change during the lifetime of a process, and are reusable.

Note that process numbers may repeat across objects, but if the object numbers are unique, then there should be no problems.

Interaction standards

So far I have talked about how the objects are built, and how they pass messages. What most people want to know is how to draw things on the screen and play with them with their powerglove. This section deals with that aspect.

Each object is a completely independent entity, containing all the necessary information about itself. It, however, must also broadcast information via links to other objects that may be interested. The other objects that may be interested constitute the "world" as far as the original object is concerned (see "Virtual Worlds") When something happens to an object, it generates messages that get sent to the other objects that "need to know". In the past the other objects may have requested to be notified of this information.

The great problem is that you are never sure how the other objects are defined. Some may be defined as triangular facets, others a b-splines, others as CSG models. They make be texture or bump mapped. They may be transparent, reflective, light emitting. There are even new types of graphics objects appearing, and no doubt other ones will appear in future. The point is that we must not make assumptions in the VR system about what types of "physical" object representations are supported. We must be able to support any and all types. If proper process encapsulation is provided, then this is not difficult, because only the object itself needs to know exactly how it is defined.

Touch

This occurs when two objects "collide".

When the "physical" aspect of an object changes, then a message is sent to the other objects detailing a minimum piece of information with which the other objects can respond, such as the centre and bounding sphere of the modified object. If other objects detect that this bounding sphere intersects with their own bounding spheres, then further messages will pass between them.

Each object much be able to tell whether any part of it is present in a given volume of space, and whether that part is surface or interior (in solid models as opposed to surface models).

Once the two objects are convinced that a possible intersection has taken place, possibly though the exchange of a more accurate bounding model, or on past experience (the object were intersecting just before, and now they have moved closer as opposed to further away.) then the following recursive algorithm is executed.

An initial bounding box is calculated, most likely encompassing the intersection of the two bounding spheres. This bounding box is then recursively divided into two sub-boxes, by division along the edge that is longest, or by choosing a random edge if there are two or more edges that are equally the longest.

For each sub-volume, both of the object determine if they occupy it. If both do, then that sub-volume is also recursively divided. If neither or only one object is present in the volume, then no intersection is possible within that volume, and it is discarded. This recursive division continues until the volume of the bounding box is indistinguishable from a point. (ie. when one of the objects decides that they are definitely probably intersecting as far as they are concerned.)

By this stage there will most likely be quite a number of intersection points. This will provide a large enough sample group to decide at what angle the objects are intersecting. Another brief conversation should happen about the relative velocity, momentum, spin etc. of the objects, and they should then agree on what happens next.

"What happens next" is completely unknowable. Since each objects is an independent entity. One may decide to follow the normal laws of newtonian physics and "bounce" off the other. The other object may decide to ignore and continue doing whatever it was doing. One object may decide to emit a sound ("Ouch" perhaps?) or dent, or disappear entirely. Note that since the objects have come in contact, they are now free to record this fact and use it to pass messages in future. Since any message may be passed when the objects have decided they they have come in contact, then anything is possible. Some "rules of object etiquette" should be formulated to let the objects interact in a cultured and friendly manner, such as giving suggestion to the other object about what should happen next. If a "hand" object grabs an object, then the other object is better to respond to the directions of the hand object and smoothly follow it then jerkily bounce off the fingers that hold it.

Also, to speed the process up, the bits of code that do all the checking may be sent to the other process as an encapsulation, where they can be started up (as described below), and all the collision checking can be done within one object! However, unless the hardware controlling one object is significantly faster, this will not be terribly beneficial. (as one machine is now doing all the work instead of two.)

Sound

There are two approaches to sound. When two objects are aware that they are in the same "world", then they may simply send packets of sound information to the other objects by establishing links between the sound generation processes in each object.

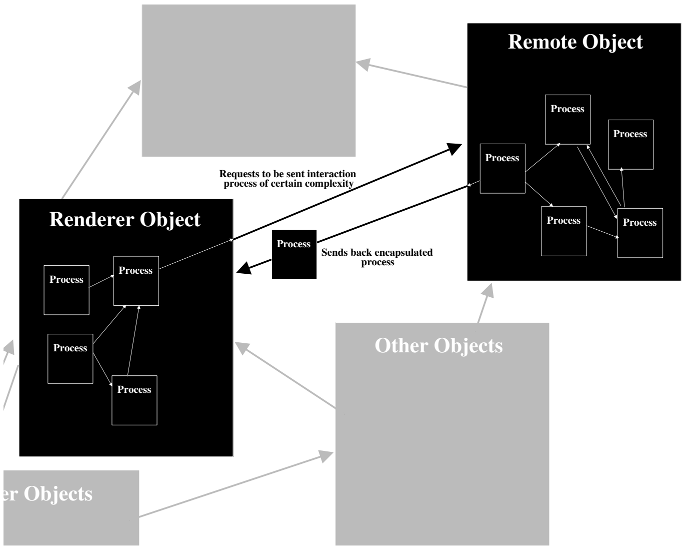
With the second method, when two objects become aware of the other, then they may exchange encapsulated processes that generate the same sound. If one object regularly generates a "Beep" sound, then instead of sending the same "beep" sound sample every time, then it can send an encapsulated process in a message for the other object to start up, which will form a link with the original object. when a "beep" is needed, then the object sends a control message to the process that is now embedded in the other object, which produces the same effect.

Why this logical shuffling? Because processes that constitute an Object are more likely to be more closely related, and have faster communication links. When shifting around large data blocks, this can be important.

Vision

This is done in a similar method to the sound generation method above.

Once objects become "aware" of each other, then they exchange encapsulated processes that render them. This allows, in many cases, the rendering procedure to be done in the one VR processor, with minimal communication with other objects that simply update their embedded processes with the new position/orientation/shape of the object. All the hard rendering is kept within the one object, and the advantages are obvious, since you can control your own object much more easily than another.



Since different machines are more/less powerful than others, then the "type" of process you request when your "eye" object becomes aware of other objects will vary.

This leads on to "multiple views"

The term is a misnomer, because it is not only vision that will require this. Sound will also benefit from this, and also other interaction methods to come.

When an object requests to be sent a process to handle vision/sound for the remote object, it may also specify what "level" of handler to send back. For example, a slow machine may request to remote object to send a process that renders it as a wireframe. A faster machine may request to be sent processes that render the remote object as a raytraced b-spline construct. An intelligent machine may choose to render some in one form, an some in another depending on distance, or how much processing bandwidth is available. If more objects are added to it’s "world view", then it may begin rendering some of the original objects in a less intensive format so that the frame rate remains steady. Also, if you begin "physically" interacting with an object, you may change it's representation to either a higher form (to show exactly what you are doing) or a lower form. (to save communication bandwidth)

You might perceive a problem if a scene becomes too complex, and the local process cannot handle the number of processes that it would be necessary to load. The local process can always fall back on leaving the process in the remote object (probably increasing net traffic) or taking other steps, such as only rendering those objects that are closest.

The local object can also make intelligent decisions, like rendering an object as a lower level graphic when it is further away, or if it past a certain point, or too small, not rendering it at all.

The same applies to sound, or any other interaction medium. If, for example, an object did not respond to a request to supply a sound process, then it can be assumed that it won't be making any sounds.

This means that each "interested" object builds up a dynamic library of remote objects' processes that enable each object to independently render the remote objects with minimal input from them. Once a process is no longer needed, it is terminated, and it's links severed. (In that order.)

Ownership

In this model, there is no real concept of ownership. Each object is self-owning, and no other object has control over it except itself. Of course, each user has control over their own machines, and if an object is in residence, then they have the power to do whatever they wish to it. This is how object building and testing would be done. It is expected that there will be a core routine in each object that responds to commands from outside, authenticated sources to perform actions, such as self modification. So, although no other object controls it directly, it can send commands for the object to perform on itself.

Security

If an object wishes to remain secure, then a pre-requisite to communicating with that object may be engaging in a challenge-response authentication system, or sending some sort of code along with every message.

Since messages are contained within a standard data block, then there is no reason why it cannot be encrypted before transmission. The secure object may even send a process to the other object that is used to decrypt the data.

Refusing to respond to messages from certain objects also means that the secure objects will remain "invisible" to them.

The Death of 'God'

'God' is a term that I coined for an omniscient, all powerful central controller, which seems to have passed into general use.

Using the object based model described here, it is easy to see that there is no longer any concept of a God. A central controlling process would not necessarily make things more efficient, and could even degrade performance. A central server destroys the goal of a truly distributed system, and forces a situation where all messages and commands must be forced through a central machine.

'God' was invented in order to solve the problem of partitioning Virtual Worlds into manageable chunks, with a central controlling process for each world. As you have seen in the section "Virtual Worlds", this is not necessary, as each object is capable of making the distinction on it's own. In fact, this allows greater flexibility as some objects are now able to communicate with objects that are "outside this world", and provide services equivalent to doors and telephones. You can envisage a problem where half of a world is interacting with objects that the other half doesn't know about, but effective communication standards where objects share common data should prevent this. As always, it can be though of either as a feature or a bug.

Database Models

A lot of VR systems currently work on a database system, where each object is represented as a collection of data describing many attributes of the objects. This system requires a large central database manager to make the objects appear as active things. Once again we run across the problem of centrality, which gives rise to many problems, while solving others.

The main issue is that you inevitably end up with a lot of the problems associated with multi-users and traditional Database systems, in this case with possibly several hundred users trying to get at the same information at the same time. Also, a Database model incorporates some restrictions that are almost impossible to get around if future expansion is wanted.

Also, the Database will be difficult to distribute amongst parallel processors and retain all the advantages that the traditional Database approach solves. If you keep it all on the one platform, then you will eventually hit a ceiling where even the fastest computers can't handle any more.

Then, when you want to plug several databases together, you will end up either with a horrible mess, or a system that will seem surprisingly like the one listed here. Once can think of my object centred model as a large number of small databases which communicate using a built-in protocols, so in effect what I am talking about is an intelligent distributed DBMS.

Syncronicity

In a distributed system, it is difficult to make sure that everything that was supposed to happen synchronously does. This is basically a network problem, and will hopefully be solved as networks get faster.

Should timestamps be put on messages? Yes. What is done with those timestamps depends on the object that receives them.

We also encounter a problem when a process begins sending messages to another process at a greater rate than the receiving process can cope with. This will lead to a huge queue, and eventually something will break. Most likely the OS will just drop the messages. Again, this is a problem that can only be solved by intelligent coding of the objects, perhaps in the form of handshaking.

But, as networks go at different speeds, might not you have the problem that causality will seem to be violated in certain cases, when multiple viewers are out of synch with each other? Unfortunately yes, but there is little that can be done. A similar thing happens within the framework of general relativity, and the real universe seems to cope there.

Measurement

How do we measure the various aspects of a Virtual World? Time will obviously be measured in seconds, or fractions of a second. Date/time stamps already are well understood and in standard use. Most interactive tasks between human users and objects will proceed at an average pace, but when object-object interactions are concerned, the speed of the interaction, whatever it's form, is only limited by the speed of the hardware. If the user wants it to slow down, then they will have to find a way to instruct the objects of that request.

One distracting question is how to measure spatial co-ordinates. 16 bit numbers are insufficient, and floating point numbers suffer from decreasing accuracy the further away you get from the origin.

The problem is that you may wish to model objects and events of any size, from sub-atomic quarks to galaxys. Some have suggested a "world scaling" index, but as there is no central server, this may be difficult to enforce. also, what if you want an object the size of a quark along with an object the size of a galaxy in the same world? To say "This should not happen" is to set an artificial limit on a VR.

The best solution is to use a numbering system that can cope with the two extremes with constant accuracy. Floating point numbers, although they can cope with the range, are subject to variable accuracy. The best solution then is to use a large integer, and the use of 128 bit integers has been suggested, as they can express the range of events present in the "real universe"

If these numbers are used to define each of the x,y,z axes, then a large cube results. At the boundarys, simply wrap around. Again a similar thing happens in the real universe. If you go far enough in one direction, then you end up where you started.

Some have challenged this on two fronts. First, that for most machines it would be faster and more efficient to process 16 or 32 bit integers, but as already said, these are insufficient, and anyway, a scaling factor would also have to be used, resulting in a degradation of performance. Also, since the numbers are integers, operations performed on them should take less time than equivalent operations on floating point numbers, which are also in widespread use.

Second, some say that the use of such large numbers will increase memory usage and network bandwidth. this is true, but Virtual World co-ordinates need not necessarily be used at all times. There will be only a limited set of times when number of this precision will be needed, generally to specify an overall position for the object. After this, smaller offset values can be intelligently supplied as needed.

Also, most of the traffic in absolute positions will be in messages notifying other objects of a movement, and network packets carrying this information will generally be larger than the amount of the data being transmitted anyway.

Rendering

The front end of VR, the rendering interface (which is often mistaken for the entire VR) has now been defined in an almost infinitely flexible way, which is not what most people want.

In most rendering systems, there are two broad rendering models, view dependant and view independent.

Diffuse lighting takes into account ambient light, radiated light from other sources, and shadows. These are the view-independent parts of any lighting model.

Opposite to this are the specular lighting models, and reflection, which make up the view dependant sections of any viewing model

The best way to deal with the view-independent model is to use a radiosity algorithm, which will quite happily generate the relevant information. This information is then incorporated into the objects themselves, by changing the colour/intensity of the visible parts that make them up. As you might have guessed, this can be implemented by each objects computing it's own hemicube and form factor information, and then passing messages. The job of radiosity computation is distributed amongst the objects that make up the world. Intelligent decisions can also be made dependant on the distance between objects, and whether certain objects only emit or reflect or absorb light.

The view dependant parameters have to be calculated by the rendering object that is "viewing" the scene. This is obviously done with the help of processes loaded from the other objects in the scene. The simplest way to handle this lighting model is to use raytracing. Unfortunately, raytracing is still a fairly CPU intensive process, and may not be practical for low end machines.

If a raytracing algorithm is chosen for final rendering of the radiosity model, then a good algorithm to choose is the Wallace, Cohen, and Greenberg two-pass approach [SIGGRAPH 87, p211] which uses reflection frustrums to combine the models.

Low end machines will simply ignore any of the extra information present, and will stick to their wireframes. Any other processes viewing them will likewise see only the highest visual model that has been assigned to them. If the creator has gone to the trouble to define solid surfaces, despite the fact that they may not be able to see them, then higher powered machines will still be able to use them. Also, only the objects that are interested will be able to exchange messages to do radiosity. Objects that are not interested will simply have solid-colour surfaces, and will not respond to lighting models.

It is suggested that only radiosity light objects are defined, and point sources are ignored. If an object cannot handle being illuminated by an area light source, then it must render itself as a solid colour.

Of course, the method that each renderer uses depends on what sort of rendering process it requests from the remote objects. If it supports only wireframes, then the requested processes must be of the lowest level, which only draws lines. The next step up may involve requesting processes that draw into a Z-buffer. Each object in turn should be able to supply a number of different rendering processes.

The way that objects request rendering processes from other objects is arbitrary, and follows similar guidelines to the way in which it requests processes to "render" other interactions such as sound, or touch.

If a particular user does not like the way in which a particular object is rendered, then they are perfectly at liberty to change the object to suit their tastes.

Patch-Process relations

I define a "patch" as any graphics/"physical" object that the system supports. From a rendering perspective, this means a quadratic patch, or a polygon patch, or any other graphics object.

I suggest allocating one process to each of the patches that objects are built from. Simple pipelines allow the other processes in the object to alter the patch, but all the work regarding rendering the patch (in all it's forms) and collision detection is left up to that process. This process can be copied (for use in other objects) or replaced entirely with a newer process, which extends the functionality with the same interface to other processes.

For example: You may define a patch/process pair (really the patch only exists as a data structure within the process) for a triangular facet, with flat shading. This process is capable of setting and rendering all the aspects of this patch, including encapsulating and sending the process of the appropriate level to another object, and maintaining contact with that other process. It is also capable of detecting if the facet is within a specific volume of space to aid in collision detection. This process can be copied a number of times to produce a number of patches. A controlling process simply has to send gross feature updates such as a change in colour, or new vertices co-ordinates.

At some future time, it is decided to replace this process with one that implements a radiosity method. As far as the controlling process is concerned, it still only tells the patch process gross feature information, but now the patch process does a lot more work in terms of communicating with other patch processes to evaluate it's final surface values from the radiosity methods. The radiosity method may also require that the patch subdivide, and the process will handle this transparently (perhaps starting up new processes). As far as the controlling process is concerned, nothing has changed, and the remote rendering process also sees an unchanging interface (although it may now be doing slightly more work).

An extension may allow a patch process to detect when it has intersected with another object, and send another message when this happens. the patch will then act like a "button". When it is pressed, then an action occurs. Again, the fundamental method of rendering the patch may change drastically, but the same final result is produced when it is pressed.

As the machine(s) that an object is running on become more powerful, then these path processes may be transparently upgraded without having to change any of the rest of the object. This in effect leads to a highly modular structure within the object.

External Interfacing

Within each language, there must be some way to communicate with the machine running the process, in order to communicate with external peripherals and conventional programs that may be running on the machine. For instance, one process that constitutes a user object may read the information coming from a dataglove. This process cannot be moved from the processor on which it is running, and this condition must be flagged. Another object may be in contact with another, conventional program, or with the operating system itself, so that manipulations of that object are equivalent to sending commands to the OS.

Conclusion

There are a lot of people trying to fit round old concepts into the new square hole of VR. Static processes and OO design is one such area. For various reasons, a traditional OO approach will not work, due to the fact that each object is an independent entity, and a hierarchal model is vastly impractical. Things such as the concept of "Rooms with impenetrable walls" is a workable solution to the problem of overload, but vastly limiting. Even the basic concept of how a program relates to it's processor has been redefined.

There is also much to be said on the social and societal impacts of VR, in both directions. The culture that will be using VR is already in place, waiting for the technology. The first group to get their hands on real VR will be the inhabitants of the Global Village. Also, expectations of VR are far outpacing even the technological advancement, leading to the sort of collapse that befell AI and left it floundering without funding for a decade.

Hopefully, I have provided some new thoughts on the matter, and provided a little new knowledge and light on the vast area of VR. There is so much more to say, specifically on the exact implementation details of specific sub-sections such as vision, sound, and the definition of a real language, but I think I will leave that for future papers.

Copyright Notice

Jeremy Lee, 16th July, 1992

Bond University,
Gold Coast, Australia
s047@sand.sics.bu.oz.au

or contact the Uni.