THE FERN MODEL: AN EXPLANATION WITH EXAMPLES

Max Minkoff
March 1992

Based on the FERN Model developed by:
Geoffrey Coco and Colin Bricken

The ultimate design goal of VEOS and the systems that will be built on top of it is to provide a generic structure which will not only provide a fast enough system to provide a high-quality virtual reality experience but will also support the many world features that will ultimately be desired by the diverse worldbuilders that will be using the system in the future. This in turn leads to a more basic set of design requirements that will not only provide such quality and diversity, but will also function on the current and near-term available computer systems. They are:

Generic world structure

The world must impose as little predetermined structure as possible. It should be able to support as many types of spaces, systems, and interactions as possible.

Distributed data and processing

Due to the limited number of processors available that can adequately handle a well-populated virtual world, the system must allow its needs to be distributed across many computer systems, not only allowing for multiple participants, but also for the support of complex worlds on relatively low-end computing hardware.

Reliable data management

Due to the distributed nature of the system, there must be a reliable and consistent way to handle the flow of information between entities in the world.

Fast hardware updates

A requirement in any virtual reality system, hardware updates, such as the update of viewed images as the participant moves through the world, must not be sacrificed for information flow.

Furthermore, for a system to support the type of interaction between entities that is envisioned, entities themselves must have certain features:

Autonomy

Entities must be able to support and operate themselves, without dependence on any other entity or system.

Equality

Again because of the distributed nature of the system, plus the requirement for generic worlds and autonomous entities, all entities must be equal in structure and function.

World containment

The nature of the virtual world is that the world itself is an entity, and each entity is itself a world. This must be reflected in the system design. As will be seen in the next several pages, the FERN model provides all of this functionality in a simple, concise package.

ENTITIES

An entity in the FERN Model is a single and complete set of processing functions embodying zero or more objects in the virtual world. Usually, an entity will be the whole of any one being in the world, but it is possible to have cooperating entities form a single being. Any set of objects that operate as a connected whole and have one set of behaviors and functions is in entity.

Behavior

Entity equality and autonomy is fundamental to this system. As such, behavior cannot be based on control of one entity by another, but instead must be built on controlled individual reactions to world events. In the FERN model, all behavior is reduced to reactions to perceived data regarding other entities in the world. In order to create entity actions and interactions, including anything from a reaction to a tool to hunting for food, the worldbuilder must create the corresponding set of behaviors within each entity. It is quite possible, though presumably not common, to have one entity, for example a lion, try to interact with another entity, say try to eat a cow, and have the receiving entity not respond. In this case, the cow would not make the correct response – to be eaten. It is up to each entity how it will react to the given situation, and up to the worldbuilder to create the behaviors. In subsequent sections, the underlying structure for employing these behaviors will be described.

Entity Interaction

In order to create the type of complex interactive worlds that are currently envisioned, have them run on hardware that is available currently and in the short term, and allow multiple participation from potentially remote sites, virtual world entities must be autonomous and equal. It would not do for many remote entities be dependent on each other, or dependent on some other type of facility in the world or on the network. Furthermore, giving entities autonomy means that they can interact with all other entities in ways limited only by imagination and programming techniques, not by attention and processing power.

The FERN model indeed does provide entities with both autonomy and equality. Every entity has the same basic functional structure, and operates as a peer to all other entities in the virtual universe. An entity's capabilities are only limited by the extent to which they are programmed and the speed of the processor the entity is running on. In keeping with the reactive model of entity interaction, each entity is ultimately responsible for its own actions. Every entity has the opportunity to refuse to "obey" another entity. Really, no entity would even impose a command on another entity. Instead, behaviors are created through programming the entity to want to respond to a state in the environment. Instead of a wand tool commanding an object in the world to move, the object would be programmed to move when the wand tool emitted the object's unique ID with a suggested movement vector.

One further level in the discussion of entities is the nature of virtual worlds in the virtual universe, and the relationship of a world to a contained entity. The virtual world is an entity to itself, as much as any entity contained in it is an entity. This, plus the necessary relationships between entities and between entities and hardware, leads to the need for every world to be an entity and for every entity to be a world. In the FERN model, since every entity contains both the functions for participating in a world and maintaining a world, any entity may do either or both at the same time. It is even possible to inhabit several worlds at the same time.

THE FERN MODEL

As noted in the previous section, all entities are equal in functional structure. Every entity in the virtual universe operates in the exact same way. Each entity performs three functions:

Database

Each entity maintains a database of personal attributes, sibling attributes, and children attributes. It using the searching language NANCY, a basic part of the VEOS system, to accomplish this task.

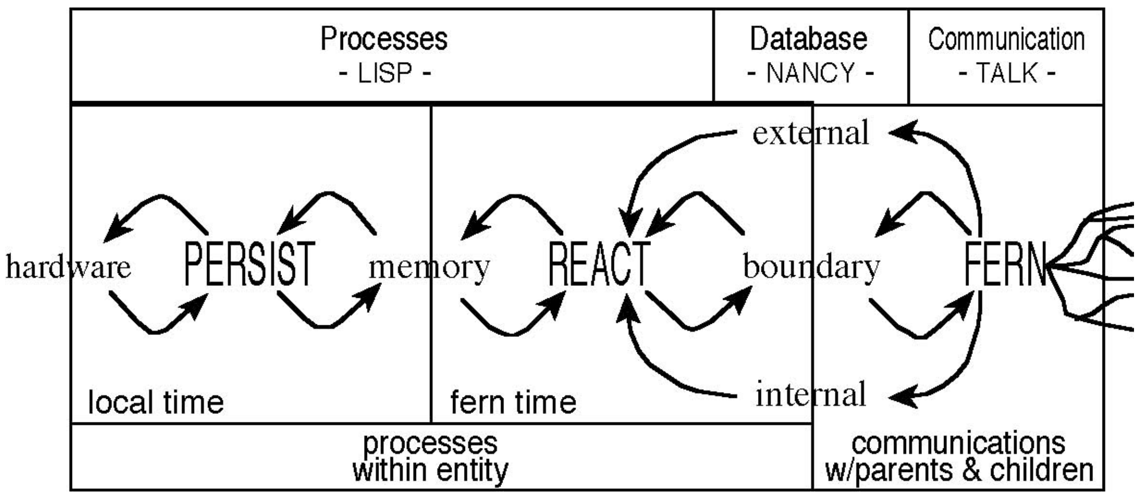

Figure 1: The FERN Model

Behave

Each entity has two function loops that process input data, both from the entity's own personal states and the states of other entities in the world, into state changes of both attributes that are displayed to other entities, such as position and color, and those that are only known to the entity itself, such as degree of fatigue. These processes are accomplished through programming in the LISP language.

Communicate

Communications with "parent" and "children" entities are the backbone of the system. It is the only way for the entity to be aware of the state of the worlds both within and without the entity. The communication process is facilitated by the use of Talk, another major part of the VEOS system.

RESOURCES

The data used by the entity's processes are stored in areas called resources. They are both the sources and the sinks for the data used and created by the entity. The two types of resources used by the entity are those that are maintained in the entity's grouple space and interfaced with the Nancy database searching language, and those that are stored and used elsewhere.
Nancy-maintained resources

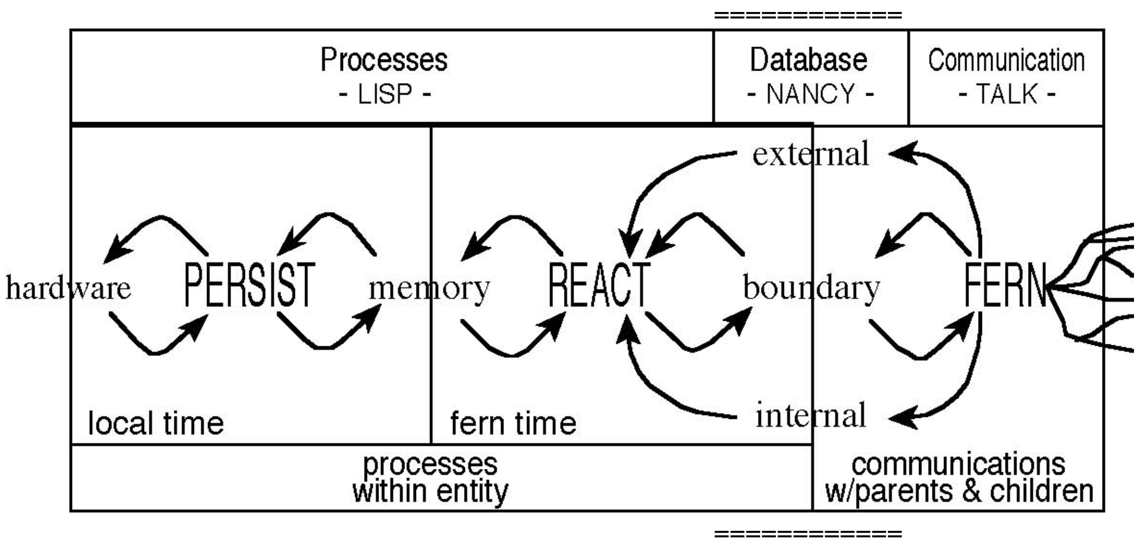

Figure 2: Grouple Space Resources

There are three resources that are specifically stored in the entity's grouple space and accessed through NANCY, the VEOS data searching language. They are the three sets of data that pertain to matters external to the entity. This includes both the entity's own external state and the external states of other entities in the system. They are:

Boundary

Data about the self that is meant to be communicated with the parent and thus shared with as many siblings as are interested is stored in the boundary. This area of grouple space is readable and writable. An entity would read the boundary to get its current information and write to it to change its state (through a reaction. An example might be reading the entity's current position, applying a movement vector to it to create a new position, and writing that new position out to the boundary.

External

The external partition of the grouple space contains information regarding the entity's siblings in all the worlds that the entity is a member of. Data comes from the boundaries of other entities and is distributed and updated by the parent of each world that the other entities are part of. The entity can set perceptual filters to tell the parent how to filter all the world data into only that information the entity is able to react to. Unlike the boundary, this area is readable only. Since this contains only information about other siblings, there is never a reason to write to it.

Internal

The internal is comprised of all data collected from the boundaries of contained entities. While it cannot be written, it is meant to filtered and communicated to all contained entities. It is the internal partition of the parent entity that is filtered and sent to the externals of all contained entities.

LISP Resources

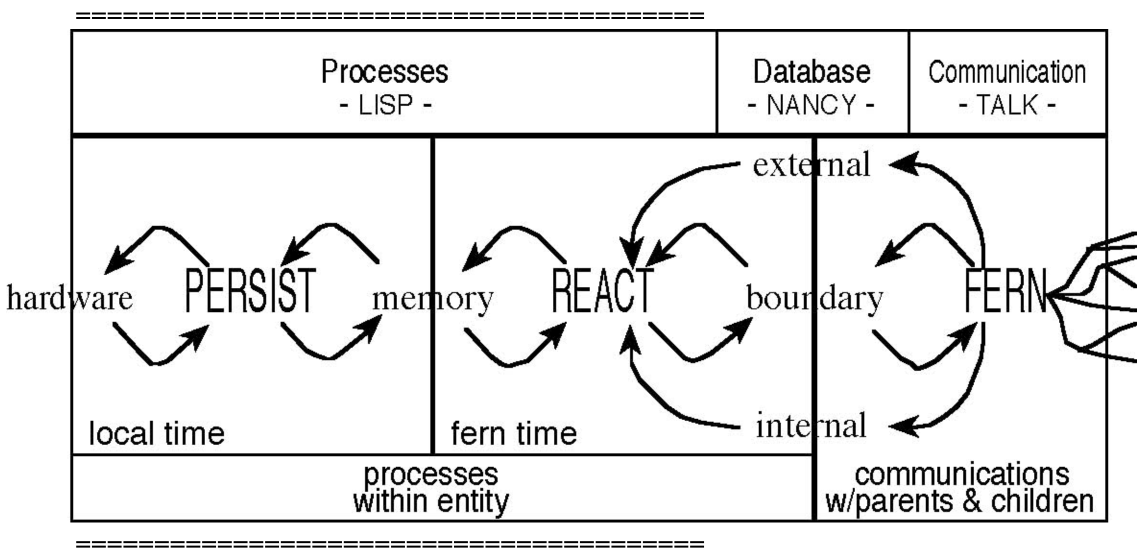

Figure 3: LISP Resources

The other two resources contain data about the entity that is never passed to the rest of the world, but instead are involved with the ongoing maintenance of the entity in relation to itself and to the physical world ("real" reality).

Memory

This resource contains data pertaining to states that are not communicated to other entities, at least not directly. They are only pertinent to the entity itself and its persistence through time. Unlike the mandatory grouple space partitions mentioned above, memory can exist in either the entity's grouple space, LISP space, or a combination of both, depending on the needs of the entity and the structure of the data.

Hardware

This is data produced and/or used by an entity's related hardware. Data can be read in from a hardware device or written to it. An example of a device that would be read in would be a position tracker, providing both position and rotation in the world with regard to the transmitter. Data might be written to the hardware resource, for example, to enable the imager to create the proper images. This particular example is one where two pieces of hardware are connected to the same virtual entity - the eye. In this way, position and rotation data can be attached directly to the imager to provide the fastest refresh rate to the participant. Otherwise, this data would have to be communicated from one entity to another and be dependent on the speed of the processors maintaining the dependent entities and the network between them.

PROCESSES

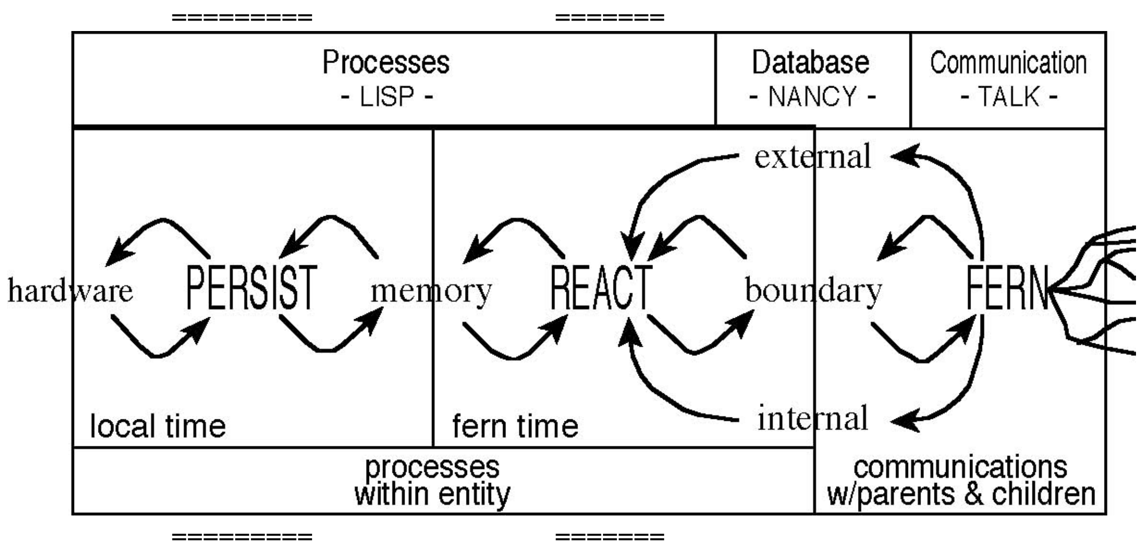

Figure 4: Behavior Processes

Processes are the operational functions the entity uses to exist in the virtual world. There are two types of processes that occur. The processes that provide reactions based on input stimuli are the behaviors, PERSIST and REACT. The other process is the FERN, which maintains all necessary communications with other entities in the virtual universe.

PERSIST

These are behaviors that are independent of any factors outside the entity. They only rely on data known only to itself: data that does not relate the entity to other entities in the universe, but only to itself and to time. PERSIST behaviors read and write to both hardware and memory only. Since they do not depend on or affect on data related to other entities, it would never need to access any other resources than these two.

Furthermore, as will be seen, most of the time that a PERSIST loop will be occurring, data with regard to other entities will no longer be fresh anyway. These processes can, though, indirectly affect the boundary by setting variables that REACT behaviors will depend on, changing the boundary. It is rare, though, that data from memory will be directly passed to the boundary. More likely, data in the boundary will be manipulated with respect to data stored in memory, such as a vector stored in memory, affecting the entity's current position stored in the boundary.

The PERSIST loop happens in local time - as often as the hardware will allow. Unlike REACT, which must wait for an update from the parent to occur, PERSIST will occur as often as computationally possible. The is due to the fact that PERSIST behaviors are independent of data from the rest of the universe.
An example of a process that occurs in the PERSIST loop is the one that was mentioned earlier as an example of reads and writes to hardware - the update of the image due to the position and rotation of the head. New head position and rotation values would be read by hardware and the images redrawn in the PERSIST loop. Current head position and rotation may be stored in memory, so that on the next REACT pass, the head’s position and rotation may be updated in the boundary.

REACT

These are behaviors that both depend on and affect factors both inside and outside the entity. As such, it can both read and write to both the boundary and memory, and also reads the external and internal partitions of the grouple space. The REACT loop only takes place as new updates to the boundary and external partitions are made. After the REACT loop is processed and the boundary updated, the new boundary is sent to the parent. The next REACT loop will then not occur until the parent responds with an update to the external partition. This is what is meant as FERN time, as opposed to local time. It is completely dependent on the speed of the network and of the parent, and therefore of the processor that the parent is running on.

An example of a set of behaviors:

Hunger

As time passes from the last time an entity eats (assuming it needs to), the entity gets hungrier and hungrier. This would be
processed in the PERSIST loop such that hunger would be a
function of time since the last meal.

Eat

Due to a high hunger value in memory, a reaction may be to move toward food. A movement vector in memory would be set by this behavior.

Wind The entity may realize that it's in the middle of a strong wind, and after reading the wind's direction from the external partition, may combine the wind's affects with the movement vector determined by the eat function.
Move Finally, the movement function would combine the movement vector determined by previous functions affecting it with the current position read from the boundary.

Collision

If the entity's nature is such that it cannot occupy the same space as another entity at the same time, it may check its new position in the external to see if any other entity occupies the space. If it does, it will again modify the current position, and possibly the movement vector as well, to account for the collision, before the REACT loop is completed.

FERN

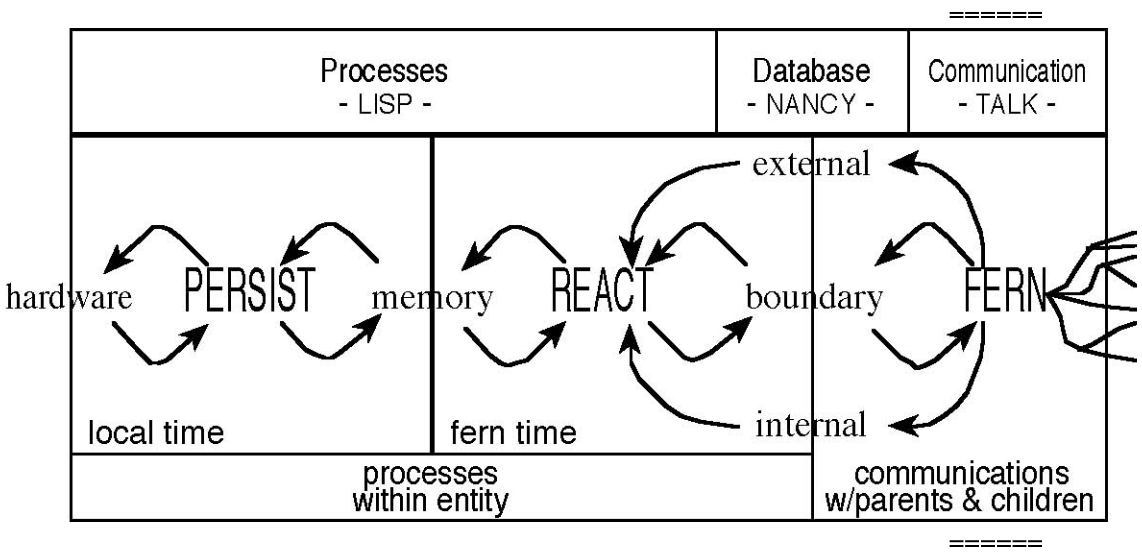

Figure 5: The FERN Process

The FERN (Fractal Entity Relativity Node) provides accurate and consistent world data management. The FERN function handles all communication with the parent and children entities.

As a parent, or world, the FERN keeps track of all entities in the world. It accepts updated boundaries from each entity and stores it in its internal grouple space partition. In turn, the parent updates each internal entity's external partition with the current state of the world, in accordance with the entity's perceptual filters, as often as the entities send updates themselves.
As a child, after a REACT loop is completed, the FERN communicates an updated boundary to the parent and receives back an updated external partition.

EXAMPLES

Learning

Entities, if predisposed to do so, can learn new behaviors by receiving new function definitions and then adding the new function call to their behavior loop. A behavior giving entity would simply include new functions in its boundary. Entities that are sensitive to these new behaviors would execute the code in the boundaries of the behavior-giving entities. Such code would likely include both the function definition and the command to add the command to either the entity's REACT loop or PERSIST loop. More intelligent entities might build in the ability to discern different types of code and make an informed decision about what code to incorporate and from whom.

Entering a world

To enter a new world, an entity will send its boundary to the entity containing the world. The parent will then add that entity to its internal partition and send back the appropriate information regarding other entities in the world. Subsequent updates to other entities within that world will include information about the recently-joined entity.

Move with wand

The wand, like any tool entity, will read in its hardware values, such as rotation and amount the trigger is depressed, process them into relevant values, such as a vector in this case, and “publish” them to its boundary. An entity that wants to use the values, such as the virtual body using the vector to move, will be sensitive to these published values when they come into its external partition. When the virtual body notices that the wand has published a movement vector with a magnitude larger than 0, it may use it to modify its current position, thus creating a movement independent of other factors. Instead, the virtual body may choose to operate other functions, such as wind, collision, etc. on the vector before the combination movement is finally made.

Follow

When maintaining a constant distance from another object, such as a bird maintaining a certain distance from other birds in a flock, the bird will add its distance vector to the other birds in question, producing a new movement completely dependent on the other entities in the world, but by no means controlled by them. The following behavior is produced completely within the entity itself.

Inhabit

Inhabiting is an easy function to accomplish. Simply, the inhabiting entity will use the inhabited entity’s relevant boundary information as its own, thus creating the same view and movements as the inhabited entity.

Portals

If an entity is sensitive to portals that can move it to another world, or another place in the current world, upon colliding with the portal the entity will change its boundary attributes into the position, rotation, and inhabited space values given off by the portal. It would be at the entity's option whether or not the previous world the entity occupied would be exited as well.

CONCLUSION

The FERN model embodies all of the design criteria identified in the first section of this paper. It provides for entities that are both autonomous and equal, and the reliable maintenance of each world's data. Furthermore, it is an extremely flexible system in that it allows any entity to have any functionality either implemented directly by the worldbuilder or learned from other behavior-giving objects in the world. It not only facilitates all communications within and between entities, it does it in a simple, elegant manner. Complex entity interactions are capable with a minimum amount of programming, and through the open-ended configuration and manipulation of the LISP language, entities will be able to grow and learn as new world capabilities are created and exchanged. The implementation of the FERN now available is a clean, stretched canvas ready to be painted with behaviors and interactions created by worldbuilders.

REFERENCES

This body of knowledge was not gleaned from a series of scholarly articles or academic texts, but from many hours of discussion and thought on the edge of the creation of this model. I thank everyone for this interaction, and especially Geoff Coco and Colin Bricken, the developers of the FERN model, for their many explanations and their openness to input, questions, and concerns.