MOVIE Model for Open Systems based High Performance Distributed Computing

Wojtek Furmanski‡, Chris Faigle‡, Tom Haupt,
Janusz Niemiec‡, Marek Podgorny, and Diglio Simoni‡
Northeast Parallel Architectures Center, Syracuse University, Syracuse NY
and
‡Department of Computer and Information Science, Syracuse University
‡Department of Physics, Syracuse University

Abstract

MOVIE (Multitasking Object-oriented Visual Interactive Environment) is a new software system for High Performance Distributed Computing (HPDC), currently in the advanced design and implementation stage at Northeast Parallel Architectures Center (NPAC), Syracuse University. The MOVIE System is structured as a multiserver network of interpreters of high-level object-oriented programming language MovieScript. MovieScript derives from PostScript and extends it in the C++ syntax based object-oriented interpreted style towards 3D graphics, high performance computing and general purpose high level communication protocol for distributed and MIMD-parallel computing. The present paper describes the overall Open Systems based MOVIE design and itemizes currently implemented, developed and planned components of the system.

1. Introduction

The MOVIE project was started at Caltech [Furm92f] within the Caltech Concurrent Computation Program (C3P) and it is now continued at NPAC at Syracuse University. The MOVIE System is currently in the advanced design and development process. The first release MOVIE 1.0 is planned at the alpha/beta level before/after summer '93 and we intend to demonstrate first MOVIE applications at the Supercomputing '93.

☐ In Chapter 2, we present the global overview of the MOVIE System, with the focus on the HPDC component, based on a network of MOVIE Servers.

☐ In Chapter 3, we discuss in more detail the structure of the MOVIE Server and we itemize its major components which are either implemented or in the development stage.

☐ Finally, in Chapter 4, we present briefly the current and planned MOVIE application projects.

2. MOVIE System Overview
2.1. Open Systems

A typical medium-to-high-end computing unit in today’s R&D labs is a UNIX workstation from one of the major vendors such as DEC, HP, IBM, SGI or Sun. These machines are typically connected by local and/or wide area networks and the distributed computing gradually becomes an integral part of the environment, with services such as distributed file systems, access to database or visualization servers, support for teamwork development and for some more advanced and automated distributed computing models.

The growing performance, reliability and popularity of computer networks, combined with emerging overall sta-
bility in the workstation hardware and software technologies seem to suggest that the high speed network of volume produced UNIX workstations, hereafter referred to simply as nodes, is very likely to represent one of the leading architectures for the advanced computing environment in '90s.

A typical lab is based on a collection of diverse systems, including personal computers, Mac/NeXT machines and UNIX workstations from various vendors. On the high end, some of these nodes may be further attached to high performance and/or capacity devices such as massively parallel machines, visualization or database servers, expert systems, neural network boards etc. In consequence, a generic high performance computer network is typically heterogeneous which raises non-trivial software development challenges for distributed computing.

The relevance of interoperability in heterogeneous distributed environments is now widely recognized and there are strong emerging standardization tendencies, usually referred to collectively as the Open Systems technology. The conformance of most major vendors to X based network-extensible windowing is one of the recent examples of such tendency.

However, the low-level standardization efforts, driven by the interoperability requirements, are often accompanied at higher levels by the vendor specific customizations, driven by the marketing mechanisms. A good example here is provided by the two major competing X based GUI toolkits – Motif and OpenLook. In the area of add-on high performance special devices, such as for example massively parallel and/or high-end graphics systems, the standardization process is even slower since the software models here are typically well behind the rapidly evolving hardware technologies.

The Open Systems software environment is also evolving and its precise content is currently the subject of vigorous discussions in the computer literature and press. Here, we adopt a pragmatic, user-oriented attitude and we consider as 'standards' the intersection of the bundled system software, offered by the individual hardware vendors.

The UNIX operating system with the socket library based network extensibility, the C programming language and the more recent X Window System are 'pure' standards within such definition.

In the area of higher level network-extensible graphics models, there is still no full consensus between vendors but the customized packages now have an increasing functional overlap and they can be naturally grouped into appropriate 'equivalence classes'. In this sense, we consider as 'standards' the Motif/OpenLook pair of X based GUI toolkits, the DPS/NeWS pair of PostScript graphics servers or PEX/GL pair of 3D graphics servers. We also include on the list of standards the AVS/Explorer pair of high-end dataflow based distributed visualization servers†.

In the area of low level programming languages, Fortran and C play the leading role, with Fortran maintaining its popularity in the number crunching sector for scientific/engineering computing and with C becoming the standard platform for system development and integration. Both languages are now being augmented by higher level constructs: the class mechanism of C++ and the index-free matrix algebra of Fortran90.

In the following, we will consider by the present Open Systems software environment the following collection of subsystems: UNIX, C/C++, Fortran77/Fortran90, X, Motif/OpenLook, DPS/NeWS, PHIGS/PEX/GL, AVS/Explorer.

† Currently, only Explorer is offered as a bundled software on SGI machines but there are also public domain packages with similar functionality (apE, Khoros) and, leaving aside the question of specific vendors/packages, one should consider the underlying visualization technology as the promising Open Systems standard candidate.
The only currently existing binding platform supporting fully consistent and complete Open Systems programming is provided by the C language, with other subsystems listed above structured as C libraries or preprocessor/compiler/linker extensions. However, the C language model itself is inadequate for ‘in large’ programming. Rapid prototyping facilities are absent and the compilation/linking time becomes a severe bottleneck for large system development. Also, there is a substantial ‘impedance mismatch’ between more modern subsystem designs and their low level C language interfaces. Typical examples include the C wrapper model for DPS/NeWS servers which destroys many attractive high level features of PostScript, or the all-but-natural C interface to object-oriented GUI toolkits and 3D graphics models. A recently popular drift from C to C++ might relieve some of these problems but C++ is still a too low level design for ‘in large’ programming.

In consequence, there is now an urgent need for a hardware vendor independent high level integration platform for the emerging volume of Open Systems software, which would accelerate the portable and scalable application development process in the distributed environment. The MOVIE System, described in the next Sections is an attempt in this direction.

2.2. Virtual Machine

The MOVIE model can be most conveniently introduced in the context of heterogeneous distributed computing as a network of Virtual Machines, custom designed to enforce network homogeneity and hence to facilitate distributed software development, while at the same time maximally conforming to established and emerging Open Systems standards.

Special modern software engineering techniques are required to achieve such a compromise and to assure that the Virtual Machine network design is scalable and open for new standard insertions. In our proposed solution, each node of the MOVIE network is an individual MOVIE Server, built as an interpreter of the high level programming language MovieScript. MovieScript derives from PostScript and extends it towards other graphical, computational and communication layers of the Open Systems software. The internal dynamics of the MOVIE Server is governed by preemptive multithreading which integrates intranode computation with internode communication and supports a wide range of distributed/concurrent programming paradigms.

In the context of the Open Systems software, MOVIE Server itself is a single C program, a single UNIX process and a single X client. The software integration strategy is layered, with the low level compiled C layer restricted to the internal binding and server development tasks and with the external API/GUI layers to Open Systems resources structured in the high level interpretive Virtual Machine style.

The choice of PostScript as the integration language represents a natural and in some sense a unique minimal solution. A stack based model, PostScript lends itself ideally as a Virtual Machine ‘assembler’. An interpreted high level extensible model, it provides natural rapid prototyping capabilities. A Turing-equivalent model, it provides an effective integration factor between code and data and hence between computation and communication. Finally, the graphics model of PostScript is already a de facto standard for electronic printing/imaging and part of the Open Systems software in the form of DPS/NeWS servers.

The concept of the multithreading programmable server, based on extended PostScript derives from the NeWS (Network-extensible Window System) server [GRA89], developed by Sun in late ’80s. The seminal ideas of NeWS for client-server based device independent windowing are substantially extended in MOVIE
towards multiserver based, Open System conforming, device-independent, high performance distributed computing.

2.3. MovieScript

The design process of MovieScript starts from the original PostScript language component of DPS/NeWS servers, as defined in the language manuals by Adobe Systems, Inc. [ASI87]. All basic design ideas, control flow structures and the scalable 2D graphics model are adopted 'as is' and then uniformly extended towards other Open Systems components.

This extension process is organized structurally in the form of a 2-dimensional inheritance forest which provides a novel design platform for integrating functional and object-oriented language structures.

All the original PostScript types such as array, string, dict etc. are retained and included in the topmost 'horizontal' layer of primitive types in MovieScript. This layer is further extended by new computation, graphics and communication primitives. The design objectives of this language sector are: optimized performance, structural simplicity, enforced polymorphism of the operator set. The group of primitive types within the inheritance forest plays the role of the root class in conventional object-oriented models.

At the same time, the PostScript syntax itself is also extended within the MovieScript design to support the C++-style 'true' object-oriented model with dynamic binding and multiple inheritance. The derived types, constructed via the inheritance mechanism starting from the primitive functional types extend the inheritance forest in the 'vertical' direction towards more composite, complex and abstract language structures. A finite set of primitive types is constructed in C and hardwired into the server design. Other primitive types and all derived types are constructed at run-time at the interpreted level.

Typically, we introduce a few primitive types for each component of the Open Systems software to provide binding between the appropriate C libraries and the corresponding sectors of MovieScript. For example, we introduce the xtclass and xtwidget primitive types to parametrize all XtIntrinsics based widget classes and instances within the X based GUI toolkit. Also, we introduce the field primitive type which, together with a class of polymorphic operators, provides a data abstraction for an n-dimensional formatted array with the Fortran90-style functionality, embedded in the original PostScript arithmetic.

When designing primitive types and the associated operators, we try to be 'faithful' as possible with respect to the corresponding Open Systems layers. In case of vendor specific solutions, such as Motif/OpenLook, we construct multiple support at this level, parametrized by the toolkit type. The higher interpreted level of derived types is then used in the next stage to resolve the design conflicts between subsystems and to construct uniform metainterfaces for competing commercial components.

In some other cases, such as for example within the 3D graphics domain, where the standardization process is currently at a less advanced stage, we start from the portable metainterface design already at the primitive type level and we construct C binders to the current standard candidates such as PHIGS, PEX or GL.

The basic integration strategy is to always promote as quickly as possible the content of a given C library to the interpreted level and to construct the ultimate design of the corresponding MovieScript sector in a series of application-driven experiments, using the rapid prototyping tools for the language extension.

The overall organization of the CASE tools for the server development facilitates such extension process. The C code for the server is layered in such a way that the externally visible component of MovieScript can be easily
modified during the language implementation process.

2.4. MOVIE Networks

The communication model for MOVIE networks is based on one simple uniform principle: nodes of such network communicate by sending MovieScript. This model unifies communication and computation: computing in MOVIE is when a server interprets MovieScript whereas communication is when a server sends MovieScript to be interpreted by another server on the network.

Social human activities provide adequate analogies here. One can think of MOVIE network as of a society of autonomous intelligent agents, capable of internal information processing and of information exchange, both organized in terms of the same high level language structures.

The processing capabilities of such system are in principle unlimited. Detailed programming paradigms for distributed computing are not specified initially at the MovieScript level and can be freely selected depending on the application needs. Successful computation/communication patterns with some reusability potential can be then retained within the system in the form of appropriate MovieScript extensions.

This methodology provides also the natural foundations for megaprogramming, i.e. software development techniques for building systems from systems. Our initial structural injection is the Open Systems software, reorganized in the uniform distributed object-oriented MovieScript style. The system initialized this way is now supposed to grow by addressing application projects and maintaining its reusable subsystems in the form of MovieScript types and shells.

The conceptual model outlined above doesn’t specify yet the network topology. The connectivity model is layered, starting from the lowest ‘physical’ level, supported by the node servers and matching the actual hardware topology, followed by higher, more abstract layers to be constructed at the interpreted level. In the distributed computing domain, the physical layer is given by the dynamic topology, described in Section 3.7 and implemented as the MovieScript interface to the socket library. In the MIMD-parallel computing domain, the physical layer is given by the interpreted MovieScript version of the previous compiled ‘C or Fortran + message passing’ models.

The major advantage of the MovieScript MIMD model for the physical layer as compared with the previous compiled models stems from the programmable, multithreading server capabilities of the ‘node program’ in MOVIE. We discuss in the following some natural configurations of the MIMD network of MOVIE servers.

The regular homogeneous sector of MIMD programming remains unchanged. After assigning a unique thread in each node server for synchronous processing, the usual ‘MovieScript + message passing’ techniques apply, with the selected thread script playing the role of the ‘node program’. In particular, all regular communication algorithms, constructed previously in compiled models (see e.g. [FoFu88a]) can be easily reconstructed in MovieScript and organized as appropriate language extension.

A natural next step is to construct the Fortran90–style matrix algebra by using the physical communication layer and the already existing single node support in terms of the field objects, now playing the role of node sections of the domain decomposed global fields. Such construction represents the run-time interpreted version of the Fortran90D model [FHKKKTW90]. Compiler directives are replaced by ‘interpreter directives’, i.e. MovieScript tools for data decomposition which can be employed in the dynamic real-time mode. Various interface models to the compiled Fortran90D environment can be also constructed. Furthermore, since arithmetic doesn’t play any special role in the MovieScript syntax, the matrix algebra model can
be naturally further extended by new, more complex and specialized regular operators, emerging in the application areas such as image processing, neural networks etc.

Complex 'real world' applications typically involve a mixture of regular and irregular processing modules. Examples include machine vision, Command and Control or Virtual Reality, where the massively parallel regular algorithms (early vision, signal processing, rendering) are to be time-shared and often coupled by pipelines or feedback loops with the irregular components (AI, event-driven, geometry modeling). The simplest organization of such problems is achieved by spatial decomposition where the separate network sectors are assigned to individual processing modules. In many cases, however, the communication constraints, scalability or graceful degradation requirements might favor the temporal decomposition techniques in which each processing module utilizes all currently available spatial computing resources whereas the real-time coupling is achieved by time-sharing in terms of loosely synchronous concurrent multithreading. The MOVIE network of programmable multithreading servers offers a new attractive model for implementing such complex applications. Distributed time-sharing can be constructed in a natural way and the resulting code maintains modularity since it decomposes into separate tasks for individual threads.

Remote server programmability can be exploited to reduce communication overhead for irregular problems. Rather than sending repeated sequences of long messages, the sender can pre-program the receiver and then send only the short parameter packets to activate transient, remotely installed procedures. This feature is particularly effective in the distributed mode for low bandwidth communication media such as telephone lines.

Many other interesting features emerge in such model. High level PostScript messages can be dynamically created and destroyed. Dynamic point-like debugging and monitoring can be realized in a straightforward way at any time instance by sending an appropriate query script to the selected node. Longer chunks of the regular MovieScript code can be stored in a distributed fashion and broadcast only when synchronously invoked, i.e. one can work both with distributed data and code. Static load balancing and resource allocation techniques, developed for compiled models (see e.g. [FoFu88b] [FoFu88c]) apply and can be significantly enhanced by new dynamic algorithms, utilizing the thread mobility features in the distributed MovieScript environment.

### 2.5. Extensibility model

MOVIE 1.0 will represent the minimal closed design of the MOVIE server, defined as the uniform object-oriented interpreted interface to all Open Systems resources defined in Section 2.1. Such a model can be then further expanded both at the system level (i.e. by adding new emerging standards or by creating and promoting new standard candidates) and at the application level (i.e. by building MOVIE based application packages).

Two basic structural entities used in the extension process are *types* and *shells*. The type extension model is based on the inheritance forest and it was discussed in section 2.3. The shell extension model utilizes PostScript-style extensibility and is described below.

Structurally, a MovieScript shell is simply an instance of the *shell* type. Its special functional role stems from the fact that it provides mechanisms for extending the system dictionary by new types and the associated polymorphic operators. In consequence, types and shells are in a dual relationship, such as nodes and links in a network or nouns and verbs in a sentence. In a simple physical analogy, types play the role of particles, i.e. some elementary entities in the computational domain and shells provide interactions between particles. In conventional object-oriented
models, objects, i.e. particles are the only structural entities and the interactions are to be constructed as special kind of objects. The organization in MOVIE is similar at the formal structural level since MovieScript shells are instances of the MovieScript type but there is functional distinction between object-based and shell-based programming. The former is following the message passing based C++ syntax and can be visualized as 'particle in external field' type interaction. The latter is following the dataflow based PostScript syntax and can be visualized as multiparticle processes such as scattering, creation, annihilation, decay, fusion etc.

An appealing high level language design model can be constructed by iterating the dual relation between types and shells in the multiscale fashion. Composite types of generation N+1 are constructed in terms of interactions, involving types and shells of generation N. The ultimate structural component, i.e. the system-wide type dictionary is expected to be rich and diverse to match the complexity of the ‘real world’ computational problems. The ultimate functional component, i.e. some very high level language defined by the associated shells is expected to be simple, polymorphic and easy to use (‘common English’), with all complexity hidden in methods for specialized types (‘expert English’).

3. Elements of the MOVIE Server

3.1. MovieScript interpreter

MovieScript interpreter represents the central part of the MOVIE Server. The MovieScript ’machine word’ or object handle is represented as a 64-bit long C structure, referred to as item, composed of 32-bit long tag field and 32-bit long value field. The tag field decomposes into 16-bit long object identifier field and 16-bit long status flag vector. The value field contains either the object value for atomic types (such as numbers) or the object pointer for composite types (such as strings or arrays). MovieScript array objects and stacks are implemented as vectors of items.

Composite objects are handled by the custom Memory Manager. Each composite object contains the header with object attributes and (optionally) the data buffer. MOVIE memory consists of two sectors – static and dynamic – each implemented as a linked list of contiguous segments. Headers/buffers are located in static/dynamic memory. Static memory pointers are ’physical’ (time-independent), whereas buffers in the dynamic memory can be dynamically relocated by the heap fragmentation handler. Headers are assumed to be ’small’ (i.e. of fixed maximal size, much smaller than the memory segment size) and hence the static memory is assumed to never fragment in the non-recoverable fashion.

The persistence of the memory objects is controlled by the reference count mechanism. Buffer relocation is controlled by the lock counter. Each reference to the object buffer must be preceded/followed by the appropriate open/close commands which increment/decrement the lock count. Only the buffers with zero lock count are relocated during the heap compaction process.

Item, header and buffer components of an object are represented by three separate chunks of physical memory. The connectivity is provided by three pointers: item points to the header, header points to the buffer, buffer points back to the header (the last pointer is used during the heap compaction).

Both primitive and derived types within the inheritance forest are represented by the (item, header, buffer) triple. For primitives, such as string or array, the buffer contains the raw data vector in the appropriate format (characters for string, items for array). For derived types, the buffer contains the vector of pointers to the superinstance headers.

The inner loop of the interpreter is structured as a
large C switch with the case values given by the identifier fields of the object items. Some performance critical primitive operators are built into the inner loop as explicit switch cases, other are implemented as C functions or MovieScript procedures. MetaShell provides convenient tools for automatic insertion of new primitives into the inner loop switch.

A single cycle of the interpreter contains the following steps: check the software interrupt vector, take the next object from the execution stack, push it on the operand stack (if the object is literal) or jump to the switch case, given by the object identifier (if the object is executable). The interrupt vector is used to handle the system clock based requests such as thread switching, event handling or network services, as well as the user requests such as debugging, monitoring etc.

Both the MOVIE memory and the inner loop of the interpreter are performance optimized and supported by internal caches e.g. to speed-up the systemdict requests or small object creation. MOVIE Server is faster than NeWS or DPS servers in most basic operations such as control flow or arithmetic, often by a factor 2 or more.

3.2. APL/Fortran90/SQL-style computing

In spite of all optimizations efforts, each interpreted model must be slower than its compiled implementation language. In our case, MovieScript is slower than C by a factor 5 or more for elementary operations such as loop overhead or scalar arithmetic. The natural strategy, often adopted in interpreted language models is to build therefore a more elaborate support for the vector arithmetic and to enforce the vectorized programming techniques.

Historically, the first interpreted model for the index-free vector algebra was offered by the APL language [BPP88]. Recently, the most popular restricted implementation of the APL ideas can be found in the Fortran90 model. Also, database query languages such as SQL can be viewed as vector models, operating on table components such as rows or columns.

In MovieScript, the numerical computing is implemented in terms of the following primitive types: number, record and field. MovieScript numbers extend the PostScript model by adding the formatted numbers such as Char, Short, Double, Complex etc. The original PostScript arithmetic preserves value (e.g. an integer result is converted to real in case of overflow) whereas the extended formatted arithmetic preserves format as in the C language.

Record is the interpreted abstraction of the C language structure. The MovieScript interface is similar to that for dictionary objects. The memory layout of the record buffer coincides with the C language layout of the corresponding structure. This feature is C compiler dependent and it is parametrized in the MOVIE Server code in terms of a few typical alignment models, covering all currently popular 32–bit processors.

Field is an n-dimensional array of numbers, records or object handles. All scalar arithmetic operators are polymorphically extended to the field domain in a similar way as in Fortran90. This basic set of field operators is then further expanded to provide vectorial support for domains such as imaging, neural nets, databases etc.

Images are represented as 2–dimensional fields of bytes and the image processing algorithms can be typically reduced to the appropriate field algebra. Since the interpreter overhead is negligible for large fields, MovieScript offers natural rapid prototyping tools for experimentation with the image processing algorithms and with other regular computational domains such as PDEs or neural networks.

A table in the relational database can be represented as a 1–dimensional field of records, with the record elements used as column labels. Most of the basic SQL commands can be expressed again in terms of the suitably extended
field algebra operators.

PostScript syntax provides flexible language tools for manipulating field objects, facilitating operations such as constructing sections (regions) or building multidimensional fields.

3.3. X/Motif/OpenLook interface

When building the MovieScript interface to Xlib and GUI toolkits such as Motif or OpenLook, we face the following design problems:

a) integrate Xlib model for 2D pixel graphics with DPS/NeWS model for 2D vector graphics and with PEX/GL model for 3D graphics,

b) integrate X and MovieScript event handling,

c) embed the XtIntrinsics based object-oriented model in the MovieScript based object-oriented model

d) design uniform metainterface to Motif/OpenLook models

The problem a) is partially solved and is to be completed after the DPS/NeWS and 3D interface designs are finished. The problem b) is solved by writing the custom version for the X event handler and incorporating X events into the MovieScript scheduling. Again, this part of the design will be completed after the scheduling sector is finished. Right now, MOVIE server is configured simply as a regular X client, i.e. the program control is passed to the default event handler provided by Xlib and the communication between X and MOVIE servers is realized by the toolkit callback requests, implemented as the MovieScript procedures. The problem d) will be solved after c) is finished both for Motif and OpenLook. Currently, c) is designed for a generic toolkit and explicitly implemented for Motif [Furm92e]. The OpenLook implementation is in progress.

The object-oriented model of XtIntrinsics is based on static binding and single inheritance. As such, it doesn’t contain enough dynamics and functionality to motivate the faithful embedding in terms of derived types in MovieScript. Instead, we implement the widget classes in the form of parametric modules in terms of a few primitive MovieScript types such as xtclass (widget class), xtwidget (widget instance), xtattr (widget attribute), xtcallback (widget callback). The types xtclass and xtattr play the role of static containers of the corresponding Xlib information and they are supported only by a set of query/browse methods. The types xtwidget and xtcallback are dynamic, i.e. their instances are created/destroyed in the run-time.

The operator xtwidget creates an instance of the widget class, taking as input two objects: the parent widget and the array of attribute-value pairs. Attributes are specified by literal MovieScript names, coinciding with the corresponding Motif names. The Motif attribute set is suitably extended. For example, the widget class name itself is a special attribute, to be specified first in the attribute-value array. The associated value is the widget instance name as referred to by the X Resource Manager. Another special attribute is represented by the MovieScript atomic item $ which indicates the nested child widget. Its corresponding value is the attribute-value array for this child widget.

Mechanisms are provided for combining various toolkit components into the global GUI toolkit. The minimal set of components consists of the XtIntrinsics subtree provided by the X Consortium and the vendor-specific subtree such as Motif or OpenLook. This 2-component model can be further extended by new user-provided components. Each toolkit component is implemented as individual MovieScript shell. In particular, the shell Xi defines the intrinsic widgets, the shell Xm defines the Motif widgets etc. There is also a toolkit integration shell Xt which provides tools for combining toolkit components (e.g. $Xt = Xi + Xm$).
The implementation of OpenLook interface in this model is reduced to specifying the shell \( Xo l \) with the OpenLook widgets and building the full toolkit \( Xt = Xi + Xol \). This work is currently in progress.

### 3.4. DPS/NeWS interface

The X/Motif based graphics/GUI model, useful as it is, lacks some essential features like a transformable coordination system, text scaling and rotation, dithering, Bezier curves, advanced font technology, and image manipulation. This functionality is provided in MOVIE via a MovieScript interface to PostScript servers such as NeWS or DPS.

In both NeWS and DPS, numerous extensions to the basic PostScript operator dictionary has been implemented to support windowing environment. In both cases, extensions are provided to support interaction between the user and the on-screen image, but these extensions are completely incompatible.

Given the circumstances, we designed the NeWS/DPS interfaces to MOVIE so as to address only the basic graphic functionality of the PostScript extension to the servers. Interaction with the windowing system has been left in the domain of the X/Motif/Open Look interface. We also deliberately refrained from explicitly supporting multithreading within the NeWS/DPS servers. However, since MOVIE supports remote server programming for NeWS/DPS, MOVIE users can still access both the language extensions and the multi-threading capabilities.

The basic model for PostScript graphics support in MOVIE is implemented as follows: all operators contained in the MovieScript dictionary except those flagged as PostScript graphic operators execute within the MOVIE server. When a graphic action PostScript operator like translate, lineto, fill, or currentgray is encountered, the MovieScript interpreter figures out the number and type of its input and/or output arguments and arranges for their transfer to or from the actual NeWS/DPS context, updating the MovieScript stack accordingly. Transmissions to NeWS/DPS are asynchronous. For operators that request a response from NeWS/DPS the MOVIE Server sends an appropriately structured PostScript wrap and waits for the answer before execution is resumed.

PostScript server programmability allows for an attractive dynamic RPC model for inter-server communication. In the MOVIE-NeWS/DPS case, this link is handled by MovieScript operators gop and gdef. The syntax of gop is the following:

\[
key \ #_in \ code \ #_out \ gop \Rightarrow \ key \ rop
\]

where \( key \) is the literal name, \( #_in \) and \( #_out \) are numbers, \( code \) is an MovieScript object capable of defining some remote PostScript code and \( rop \) is the MovieScript operator (with the prefix ‘r’ standing for ‘remote’). \( gop \) installs the user-defined graphics operator (implemented as a PostScript procedure) in the remote PostScript server and it also creates the local MovieScript operator \( rop \) associated with this remote operator. Both local and remote operators are associated with the common name, specified by \( key \). The \( code \) object can be a MovieScript procedure or string. The execution of \( rop \) consists of sending \( #_in \) arguments from the MOVIE operand stack to the NeWS/DPS operand stack, executing remote procedure in NeWS/DPS, associated with \( key \) and previously installed in NeWS/DPS by \( gop \), finally transporting back \( #_out \) output objects from NeWS/DPS to MOVIE.

The associated gdef operator is simply a sequence: \{ gop def \} i.e. it installs \( rop \) in the local dictionary under the name \( key \). In other words, the action of gdef is fully symmetric on local (MOVIE) and remote (NeWS/DPS) servers. gop output format can be used to handle \( rop \) differently, e.g. by installing it as an instance method within the MovieScript class model.

We illustrate the use of the gdef operator for installing
the rectangle drawing operator box:

```
/bbox % x y w h box => draw box of size (w,h) at (x,y)
4 { newpath 4 2 roll moveto
   dup 0 exch rlineto exch 0 rlineto
   neg 0 exch rlineto
   closepath stroke } 0 gdef
```

After installing box both in MOVIE and NeWS/DPS as above, one simply uses it in MovieScript as the usual local procedure. For example, the MovieScript code:

```
512 512 16 16 box
```

sent to the MOVIE Server will draw the 16x16 rectangle in the middle of the screen controlled by NeWS/DPS.

The NeWS interface was constructed first in the Sun environment [Furm92d] and the model has been ported to DPS in the DEC environment [Podg92b].

### 3.5. AVS/Explorer-style extensibility

A new model for visual distributed computing is proposed by the present generation of high-end dataflow based visualization systems such as AVS from AVS, Inc. (former Stardent Computer, Inc.), Explorer from SGI or public domain packages such as apE from OSC or Khoros from UNM. We focus here on AVS which is currently most popular and was evaluated recently by the MOVIE group in a series of application projects (see e.g. [Haup92]) but the following comments apply to other packages as well.

The computational model of AVS is based on a collection of parametric modules, i.e. autonomous building blocks which can be connected to form processing networks. Each module has definite I/O dataflow properties, specified in terms of a small collection of data structures such as field, colormap or geometry. The Network Editor, operating as a part of AVS kernel, offers interactive visual tools for selecting modules, specifying connectivity and designing convenient GUIs to control module parameters. A set of base modules for mapping, filtering and rendering is built into the AVS kernel. User extensibility model is defined at the C/Fortran level – new modules can be constructed and appended to the system in the form of independent UNIX processes, supported by appropriate dataflow interface.

From the MOVIE perspective, we see AVS-type systems as providing the interesting model for 'coarse grain' modular extensibility of MovieScript, augmenting the native 'fine grain' extensibility model discussed in Section 2.6. An AVS module interpolates between the concepts of a PostScript operator (since it 'consumes' a set of input objects and 'produces' a set of output objects) and a class instance (since it also contains GUI-based 'methods' to control internal module parameters). In consequence, AVS-style modules can be used to extend both the functional and object-oriented layers of MovieScript towards the UNIX domain in the form of user-provided independent UNIX processes. Also, any 'third party' source or user-constructed older software package can be converted to the appropriate modular format and appended to the MOVIE system in terms of similar interface libraries as developed for AVS modules.

The advantage of AVS extensibility model is maximal 'external' software economy due to easy connectivity to 'third party' packages. The advantage of the MOVIE model, based on the MovieScript language extensibility, is maximal 'internal' software economy within the native code volume, generated by MOVIE developers. The merge of both techniques is particularly natural in the MovieScript context since PostScript itself can be viewed as a dataflow language.

There are several possible approaches for building MOVIE interface to AVS which are currently in the design or development stage:

- MOVIE Server can be structured to behave as a collection of AVS modules, with MOVIE playing the role of a computer server. This allows for fast
development of visual AVS applications using the interpreted MovieScript environment.

- MOVIE server can be structured as an AVS coroutine. Coroutines often require more dynamic and versatile GUI tools than those provided for parameter control by the AVS model. Such interface could be then constructed in MOVIE and hence both systems would be visible on the screen and play a balanced role of coupled visual compute servers.

- Finally, one can also explore the AVS Command Language Interpreter (CLI). MovieScript wrappers to the CLI scripts can be constructed and used as language extension commands, either in the coroutine mode discussed above, or by utilizing the Server Option of AVS which allows to send directly the CLI scripts to AVS from a remote process. In this configuration, AVS is used as a passive graphics device, driven by MovieScript commands.

3.6. 3D graphics model

The need for computer visualization for commercial, scientific, and for user interfaces has grown at a logarithmic rate. Several solutions, such as using high-end graphics workstations (e.g. SGI, HP), dedicated graphics hardware (e.g. PixelPlanes), normal low-end workstations, or massively parallel machines (e.g. CM-2) have all become standard practice. While many of the solutions use some well-defined standard, like PHIGS, PEX, or GL, some use no standard at all. To date it has been impossible to effectively write an application with one source code, and run it on any of the above platforms.

We are developing in MOVIE a 3D graphics platform that can run on top of any of the standards and/or distributed across a heterogeneous network, which can contain workstations and/or massively parallel machines. This section discusses the design of the model, the rendering pipeline, and the current status of the implementation. Discussion of application of the system to VR, animation, and scientific visualization is beyond the scope of this document, but for each, a set of tools will be developed on top of the 3D model using MovieScript, and then these tools used to develop complete applications.

Most of the standards require implementations to provide a minimum set of features, like Gouraud shading, but not other features, like Phong shading, double-buffering, z-buffering, a-buffering, radiosity, or shadow calculations. The 3D model for MOVIE requires a minimal set of functionality, like flat shading of polygons, since most of the other features affect only the quality of the rendering, and not the model. A model may look better rendered by one hardware platform than another, but it will still be the same model. It is the responsibility of the user to provide the hardware capability of their desired set of features, or to use MOVIE's own rendering pipeline, which may or may not have all of the above features, as it develops, and again may require certain machines, or enough power to use a feature. The model does, by design, allow all of the above; it simply does not require them.

We did not want to force the developer to deal with his objects on a vertex-by-vertex basis for rendering; rather he should deal with graphical objects as objects at that level. Thus our approach is to maintain each object as a series of attributes and points, or aliases to other objects. Some objects contain many references to other objects. Thus objects are built much like class hierarchies in OOP, and this fits in naturally with the overall design of MOVIE.

All of the objects can, however, be manipulated by the developer at a low level, since the objects are maintained by the servers, even if they have been posted to a standard. For example, these objects, when then model is used on top of PHIGS, are kept by the MOVIE server and are also translated into PHIGS structures for PHIGS to
render, but a heuristic will be used to minimize the amount of work done at the structure level. Objects which are sufficiently changed after having been posted to PHIGS will be reposted, rather than edited. When we develop the internal MOVIE 3D standard, we will provide mechanisms for distributing the objects throughout the network when using the internal MOVIE standard.

In addition to using already defined standards, MOVIE will have its own rendering pipeline. At the developer level, the identical MovieScript commands will be used for this pipeline as for PHIGS, PEX or GL, and thus will not be seen as another standard, since the pipeline level will not be managed by the developer. At the server level, however, things will be very different. Rendering will be accomplished by executing a series of tasks, which define stages in the pipeline. Not every stage in the standard geometry rendering pipeline will necessarily be its own task; some stages may be grouped together to form a single task for a polygon or pixel. The exact form of the task will be decided at run time, based upon the actual environment available to the MOVIE servers. These tasks will themselves be written in MovieScript, and therefore will be executable by any of the MOVIE servers, on any machine, including workstations and massively parallel machines. The tasks will take advantage of some of MOVIE’s nicer features, such as index-free (FORTRAN-90 style) matrix algebra, MOVIE’s communication features, and in particular polymorphic operators.

At the time of this writing, the model has been designed and is partially implemented. The model exists in its own shell within the MOVIE server, and has libraries for each of the first three models (PHIGS, PEX, and GL). This will enable us to compile the MOVIE server with any of the three, or to completely remove the 3D graphics shell if needed. Some initial design concepts were reported in [Faig92a] and the current status is documented in [Faig92b].

3.7. Communication and scheduling

The MOVIE system is being constructed as a network of multithreading servers. MOVIE servers may reside on a heterogenous network of loosely-coupled machines as well as on homogenous, tightly-coupled MIMD nodes. Each server can perform its task independently, as a stand-alone computational environment, or it may cooperate with other servers to enhance the processing efficiency of the multiserver system.

The core of a MOVIE server is constituted by the MovieScript interpreter including the local scheduler, the Network Manager and the Message Passing Manager. We call this entity the kernel of the MOVIE server. The notion of a program in execution encompasses an interpreted thread which is scheduled within the server memory boundaries. Commonly, threads are regarded as a low-overhead units of scheduling. This advantage is amplified in the interpreted MOVIE environment where the simplest context switching requires only an exchange of two pointers.

Threads are scheduled according to statically assigned priorities, in preemptive fashion. All processes with the same priority are grouped in one of several priority queues. The scheduler controls two kinds of such queues: the real-time queues and computational queues. The threads performing time critical operations are placed in the real-time queues. A real-time thread is being executed until it either blocks itself, or until another higher priority thread preempts it. When another thread in the same priority queue is ready to run, the currently executing thread that has exhausted its time quantum may be descheduled. All non real-time threads have lower priorities, and they are scheduled by time slicing. Threads placed in low priority computational queues are scheduled to run less often. This scheduling policy enables building the interactive, dynamic environments where the demands for intensive
computation and good response time can be simultaneously satisfied.

The interpreted threads may need to exchange information. The interprocess communication is carried out using dynamically created and destroyed mailboxes. The abstraction of mailboxes enables us to have uniform message passing mechanism as well within one MOVIE server as between MOVIE servers running on different nodes. The MOVIE servers residing on a heterogeneous network communicate using appropriate network protocols. Since the UNIX domain has been our prime target, the network communication model is based on BSD UNIX sockets. The socket library offers an access to two Internet protocols — TCP and UDP. These protocols are used for multiserver communication model, and we will treat them as communication links. Servers connected by these links constitute a network architecture for our distributed environment.

The network topology is partially connected, but it can be dynamically extended to fully connected. At any time, a MOVIE server can join the already existing network or leave it. Although the network is configured from identical MOVIE servers, some of them have special jobs to perform. These nodes are called host nodes, and they manage the activity of the whole net. Any of MOVIE servers is prepared to take such role. A server becomes a host either when the network does not yet exist or when a running host node demands it. Host nodes are contacted to get network parameters of other existing MOVIE servers or to perform other global services. Since the server’s network has a very dynamic nature, the host node plays a role of the global database. In addition, such features as synchronized operations, broadcast of messages or fault tolerance can be supported by host nodes. The number of host nodes may depend on the total number of servers, the desired efficiency of computation and communication, and the requested fault tolerance.

All participating network nodes are connected by three kinds of links; host–host, host–server, and server–server. The last two kinds of links are established using the reliable Internet protocol TCP. The update of information shared between host nodes is done using less reliable but also less expensive UDP/IP protocol. All TCP connections are bi-directional. When the communication bridge has been created, the two connected nodes act in a symmetrical way, sending and receiving data. However, the functional distinction is still present since every request generates a client and a server that must serve the orders. Since one link between two servers must serve communications between many threads on both ends of the channel, information between servers is sent in packets in spite of the fact that TCP channels are streams of data.

A thread can communicate over the network in three different ways: using the transparent message passing mechanism, specifying the receiving thread and the name of the server, or using the exact network address of the remote server’s machine accompanied by the thread’s parameters. The last two choices have lower overhead and are preferably used for network-oriented applications or for the testing and debugging purposes. The first method requires only the name of a globally declared mailbox. All other operations are performed automatically by the local Network Manager. The desired mailbox might not be known to the Manager in which case the host node is consulted. If the requested mailbox is declared and thus known to the host node, its instance is generated on the server’s local side and further requests to this mailbox can be easily served.

Threads communicate among themselves using synchronous and asynchronous message passing. The synchronous message passing blocks the threads doing send/receive when nobody waits for it. The asynchronous method attaches new messages to a mailbox and the
thread owning this mailbox retrieves these messages when it needs them. The asynchronous message passing offers better overall performance of the system, because there is no explicit synchronization - it means no synchronization penalty. Taking into account the additional delays introduced by the network, it is very desirable feature. The synchronous communication is used for the synchronization of threads activity as well as for testing and debugging.

While for the network of heterogenous UNIX nodes the communication is based on Internet protocols, for MIMD architectures some particular, machine dependent message passing library is used to establish the connections. Since threads may only use the mailbox abstraction to communicate, the offered programming environment is highly portable. Only the kernel of each MOVIE server must be aware about the underlying hardware and software layers. Such uniform approach creates a great opportunity of the best dynamic utilization of hardware resources accessible through the network. It also makes it possible to assign the most appropriate kinds of tasks to a particular machine.

Our current activity is concentrated on embodying the preemptive scheduling, interprocess communication and network communication into the existing MOVIE environment. Some design concepts and the current status are documented in [Niem92a] and [Niem92b].

4. Planned MOVIE Applications

4.1. Terrain Map Understanding

Analysis of terrain maps, digitized as (noisy) full color images is the first MOVIE application, built in parallel with the base system development project. The problem, posed by the DMA, consists of two major steps: a) to reduce the 24–bit color images (inflicted with noise due to the cartographic and digitization processes) to clean separated (segmented) images, containing only a small set (typically 8) of the original base colors, and b) convert the color separated images to the high level database with all characters and symbols recognized and with all elements and patches such as roads, rivers, urban or vegetation areas uniquely identified.

This map understanding task involves diverse computational domains such as image processing, pattern recognition and AI, and it provided the initial driving force for developing the general purpose MOVIE system based support. At the current stage, we have completed the implementation of a class of early/medium vision algorithms for the separation stage, based on zero-crossings for edge detection and RGB clustering for segmentation. Our conclusion from this stage is that further quality improvement in the separation process can be achieved only by coupling the low level pixel-based techniques with the high level approaches, based on symbolic representations, and by providing the feedback loop from the recognition layers to the separation layer.

From the computational perspective, the currently implemented layers are based on the MOVIE field algebra support for image processing. Two trial user interfaces, constructed so far were based on the X/Motif interface for 2D graphics and on the AVS interface for 3D graphics. In preparation is a more complete tool, based on uniform 2D and 3D graphics support in MOVIE and providing the testbed environment for evaluating various techniques, employed so far to handle this complex problem. For more detailed discussion of MOVIE approach to Map Separates, see [Furm92].

4.2. Neural Networks

Recent years have seen a dramatic increase in the exploration of the collective computational properties of neural networks [Gros88]. The structure of such systems ranges in complexity from neurophysiologically correct reconstructions of single neuronal structures to com-
plex multiple-layer networks of abstract neural units. It would be highly desirable to have a single complete self-consistent simulation environment which addresses the algorithmic and solution requirements of various types of models.

Although in general most neural networks are highly parallel, most current implementations are still sequential. The Nnsim simulator from Stuttgart University as well as the Rochester Connectionist Simulator (RCS) are simply collections of C functions which include routines for communication between nodes, whose state is updated in a sequential and deterministically synchronous fashion. The P3 system from the PDP group at UC San Diego contains tools for network description and construction but is limited by its sequential, deterministic implementation in Lisp. The Nacre simulator environment, which forms part of the Pygmalion project, uses an object oriented network specification language called Slogan, which currently runs on a Lisp machine. This system is entirely asynchronous, and also allows node connectivity and execution in a probabilistic manner.

The Aspirin/MIGRAINES simulator built by MITRE Corporation is designed around a “black box” concept. Network characteristics are defined through an ad-hoc language which produces standard C code which is then linked to MIGRAINES interface routines in order to build complete stand-alone simulations.

Finally, the Genesis/XODUS system built at Caltech runs under UNIX and X windows and is perhaps the most complete system available today for building simulations of biological neural networks. The system is based on a modular approach similar to that in Aspirin, but centers on a higher level of detail and also has an interpreted simulation control language which allows the user to build entire simulations and manage both the network’s operation and the user interface.

There are fundamental issues which must be addressed in the design of a complete neural network simulator. Noting that not all user needs can possibly be determined a priori, the system must be both easy to extend and modify as well as to port (even to parallel machines). Network specification is of major importance since the connectivity of a network is a major variable determining the degree of complexity of the computations done by the ensemble. Therefore, the system should include not only a powerful graphical interface but also a high-level network specification language.

The use of MOVIE’s interfaces to X/Motif/OpenLook and of MovieScript as the language of choice fulfills these requirements appropriately. Implementation of the systems of the type described by Hopfield [Hopf82], Kohonen [Koho84] or Rumelhart et.al [RuMc86] which are based on manipulation of homogeneous layers of simple processing units could benefit directly from MOVIE’s Fortran-90 style index-free algebra capabilities through the use of instances of the assembly and adaptive_filter classes and associated methods. Applications which require one to focus this conceptual microscope deeper into the microstructure of more realistic models of neural ensembles could benefit directly from MOVIE’s multithreading capabilities through the use of instances of the neuron and channel classes and associated methods. Moreover, although virtually all neural network models have in the past assumed synchronous updating, asynchronous updating has advantages in terms of ease and speed of computation.

We propose a MOVIE shell which combines the parallel computing support and fast prototyping, multitasking, multiserver properties of MOVIE with the modular approach of Genesis and Aspirin in order to construct efficient simulations with ease. This work is planned within the new Computational Neuroscience Program at Syracuse University. MOVIE will be coupled with Genesis, Aspirin and
perhaps some other packages. The whole environment will be customized to serve diverse needs of the neuroscience community, and would be organized in the form of portable parallel neural network algorithms [NFB89].

4.3. Databases

The support for database processing is one of the near term planned MOVIE applications. We see MOVIE as an attractive software environment to address several new trends in the domain of data-intensive programming in large, such as:

**Parallel Databases:** A parallel Oracle system will be installed soon on the NCUBE2 hypercube at NPAC as part of the joint NPAC/JPL database project within the ASAS TECHBASE program. Current parallel implementations of relational database models, based on a mesh of SQL interpreters, provide a natural link with the MIMD MOVIE model. One possible MOVIE-Oracle interaction pattern can be constructed by splitting the machine into say even/odd subcubes, dedicated to MovieScript and SQL interpreters, correspondingly, and coupled in parallel by the lowest hypercube channel. Independently, MovieScript provides tools for emulating table layouts of relational databases in terms of MOVIE objects (fields of records) which allows to extend the parallel Oracle model towards the SIMD-style synchronous processing for large distributed tables.

**Object-Oriented Databases:** There is now a growing tendency to enhance the relational techniques by the object-oriented database models and management systems. The need for more robust and versatile object-oriented interfaces is particularly strong in complex application domains such as CAD packages, CASE tools or OIS (Office Information Systems). The data structures in such problems contain usually complex interconnections, subject to a set of complex constraints which are often beyond the capabilities of conventional relational DBMSs. The characteristic tendency in object-oriented database models is towards enforcing the high-level general purpose interpreted language support for data manipulation. The borderline between databases and applications gets diffused and the application scripts are often considered as part of database or/and vice-versa. The MovieScript model can accommodate in a natural way the language requirements imposed by the modern object-oriented database systems. It can be used both for building smooth interfaces to existing object-oriented models and for designing new custom database models. The data-code equivalence of the underlying PostScript model provides a natural support for integrating data-intensive application codes with its databases.

**Distributed Databases:** The relevance of distributed databases and the associated requirement of interoperability in the heterogeneous environment is a particularly stringent subset of our general discussion in Section 2.4. There is a growing list of the new generation large scale data-intensive problems such as Global Change, GIS, Environmental Modeling, Command and Control or Supercollider High Energy Physics where the underlying databases are diverse and distributed due to the very nature of underlying domains or projects and where successful design of a high level database layer, providing the abstraction of user-level uniformity in the distributed systems will turn out to be critical factor for these projects’ successes. Some domains, such as for example the Global Change, will soon face the formidable data storage, acquisition and processing demands, all measured in appropriate Tera-units which raises unprecedented challenges in all associated technology areas [RPSS91] [StDo91]. We see the MOVIE model being able to provide benefits here, such as system integration, data visualization and prototyping the high performance distributed object-oriented database management and information systems.

**Intelligent Databases:** A new challenging software cate-
gory developed recently in consequence of merging database technology with expert systems, object-oriented programming, hypermedia and on-line information retrieval. Intelligent Databases [PCKW89] are expected to play a crucial role as modern information systems, processing vast amount of real-time electronic information and handling efficiently information overflow problems. This discipline, in a similar way as Virtual Reality, discussed in Chapter 4.6, can be viewed as one of the asymptotic goals, i.e. as the global integration platform for various natural functionalities of the ultimate MOVIE model such as powerful visual interfaces, versatile high level language design tools and symbolic processing capabilities.

4.4. Expert Systems

So far, expert systems were usually developed in the Lisp based environment and they were typically structured as the stand-alone, expensive commercial packages. As the technology is now mature and its relevance is being recognized in various application domains, there is the growing tendency to build the bridge between expert systems techniques and more conventional programming models. One class of suggested integration techniques goes in the direction of C/C++ based support for AI [Hu89] which would allow to incorporate expert systems directly into this most popular programming environment. However, the C++ model has various deficiencies in this area such as the lack of dynamic binding, run-time class formation or rapid prototyping.

Within our approach of developing MovieScript as an integration language, the natural strategy is to build expert system support in this environment as well. Since PostScript derives from Lisp, its proper extension within MovieScript towards symbolic processing, combined with the dynamic object-oriented capabilities offers an adequate set of tools for building knowledge bases, inference engines and expert system shells. We are currently in the early stage of such design, with the initial focus on two near term application areas:

4.5. Virtual Reality

Virtual Reality [Krue91] [MIS91] is a new human-machine interface technology, offering the sensation of participant full ‘immersion’ in the computer generated Virtual World. Current hardware implementations are based on a set of exotic peripherals such as goggles for the wide solid angle 3D video output, head-position trackers and gloves for sensory input and tactile feedback. The VR projects cover the wide range of technologies and goals, such as high–end scientific visualization (UNC), high-end space applications (NASA Ames), base research and technology transfer (HIT Lab) and low-end consumer market products (TI/AGE). The VR domain grows vigorously and already reached the mass media. The positive response is a clear signal that the technology is a promising candidate for the ‘ultimate’ human-computer interface models.

VR poses a true challenge for the underlying software environment, usually referred to as the VR operating shell. Such system must integrate real-time 3D graphics, in large object-oriented modeling and database techniques, event-driven simulation techniques and the overall dynamics based on multithreading distributed techniques. The emerging VR operating shells, such as Trix at Autodesk, Inc., VEOS at HIT Lab or Body Electric at VPL, Inc. share many common design features with the MOVIE system. A multiserver network of multithreading interpreters of high-level object-oriented language seems to be the optimal software technology in the VR domain.

There is a campus-wide interest in multimedia/VR at Syracuse University, with a series of planned VR application focus areas such as high-end VR research for C³I applications in the area of the group decision support in collaboration with Rome Laboratories, GIS and Space applications in collaboration with Virtual Reality, Inc., multi-
media research in the CASE Center, high-end parallel VR at NPAC, vision research within the new Computational Neuroscience Program at SU.

We expect MOVIE to play important role in these projects since the system is capable of providing both the overall infrastructure (VR operating shell) as well as the high performance computational model for addressing new challenges in computational science, stimulated by VR interfaces. In particular, we intend to address the research topics in machine vision in association with the high performance support for the 'non-encumbered' VR interfaces [Krue91], or the neural network research topics in association with the tracking and real-time control problems, emerging in VR environments [Simo92b].

The most attractive from the planned MOVIE based VR environments is the high-end system constructed as a network of new parallel machines at NPAC, coupled by the fast HIPPI interconnect and dedicated to individual VR components such as geometry modeling on CM-5, parallel Oracle database on NCUBE2, pixel rendering and imaging on DECmpp, finally the overall synchronization and user interface on the SGI front-end. Such an environment would provide us with truly unique capabilities in the domain of high-end parallel/distributed VR.

Acknowledgments

We are grateful to Geoffrey Fox for his encouragement and continuous support during the course of the MOVIE project.

This work is sponsored by the ASAS grant under contract #353–6216.

The use of any trademark in this document is not intended in any way to infringe on the rights of the trademark holder.

References


[Furm92f] Furmanski, W., “MOVIE – Multitasking Object-oriented Visual Interactive Environment”, in Fox, G.C.,


