PUC

ISSN 0103-9741

Monografias em Ciência da Computação
nº 02/93

A Programming Approach for Parallel Rendering Applications

Andre B. Potengy
Carlos J. P. Lucena
Donald D. Cowan

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, Nº 02/92
Editor: Carlos J. P. Lucena Maio, 1993

A Programming Approach for Parallel Rendering Applications*

Andre B. Potengy
Carlos J. P. Lucena
Donald Cowan**

* This work has been sponsored by the Secretaria de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

**Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada N2'L3 G1

In charge of publications:

Rosane Teles Lins Castilho

Assessoria de Biblioteca, Documentação e Informação

PUC Rio — Departamento de Informática

Rua Marquês de São Vicente, 225 — Gávea

22453-900 — Rio de Janeiro, RJ

Brasil

Tel. +55-21-529 9386 Telex +55-21-31048 Fax +55-21-511 5645

E-mail: rosane@inf.puc-rio.br

techrep@inf.puc-rio.br (for publications only)

A Programming Approach for Parallel Rendering Applications

A.B. Potengy[1,3] C.J.P. Lucena[1] D.D. Cowan[2]

[1]Depto. de Informática
Pontifícia Universidade Católica
Rio de Janeiro, 22460, RJ, Brazil
`lucena@inf.puc-rio.br`

[2]Computer Science Department
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
`dcowan@watcsg.uwaterloo.ca`

[3]Instituto de Matemática Pura e Aplicada
Estrada Dona Castorina 110, Rio de Janeiro
22460, RJ, Brazil
`potengy@visgraf.impa.br`

March 29, 1993
PUCRioInf-MCC02/93

Abstract

Solutions to many problems in scientific computing can be expressed as parallel computations that can be implemented on multi-processor systems or networks of powerful workstations. Design and implementation of these parallel programs usually is a complex effort which requires cooperation among the participating tasks in order to ensure correct operation and consistency. This paper introduces the ADV/ROI design approach that combines Abstract Data Views (ADVs) and Remote Object Invocation (ROI) and applies this approach to the implementation of parallel programs using the principles of object-oriented design. The ADV/ROI approach is applied in two steps. First, a "working" sequential version of a program is implemented and then this program is converted to the parallel version using a well-defined procedure involving multiple inheritance. We also demonstrate that consistency of the views of the computation are maintained as the program evolves from its sequential to parallel version. The ADV/ROI approach is then demonstrated by showing the design and implementation of a parallel volumetric ray tracer.

Categories and Subject Descriptors: D.1.3 [**Software**]: Programming Techniques – *Distributed Programming*; D.1.3 [**Software**]: Programming Techniques – *Parallel Programming*; D.1.5 [**Software**]: Programming Techniques – *Object-Oriented Programming*; D.2.2 [**Software**]: Software Engineering – *Tools and Techniques*; D.3.2 [**Software**]: Programming Languages – *Object-Oriented Languages*; I.3.7 [**Computer Methodologies**]: Computer Graphics – *Three-Dimensional Graphics and Realism*;

Resumo

A solução de muitos problemas de computação científica pode ser expressa como computações paralelas que podem ser implementadas em sistemas de multi-processamento ou redes de estações de trabalho poderosas. O "design" e a implementação destes programas paralelos é, em geral, um esforço complexo que requer cooperação entre as tarefas envolvidas para assegurar a correta operação e consistência. Este trabalho introduz o enfoque de "design" baseado em ADV/ROI, que combina Visões Abstratas de Dados (ADVs) e Chamada Remota de Objetos (ROI) e aplica este enfoque através do uso do "design" orientado a objetos. O enfoque ADV/ROI é aplicado em duas etapas. Primeiro, uma versão sequencial e operacional do programa é implementada e depois esta versão é convertida para se tornar uma versão paralela através do uso de um procedimento bem definido que envolve herança múltipla. O trabalho também demonstra que a consistência entre as visões das computações são mantidas à medida que o programa evolui da versão paralela para a sequencial. O enfoque ADV/ROI é demonstrado exibindo-se o "design" e a implementacão de um ray-tracer volumétrico paralelo.

1 Introduction

Current multi-processor systems and networks provide a powerful environment to support parallel applications in large-scale scientific computation, and development of complex applications using these hardware structures is becoming feasible. However, as noted in [Nash], scientific computing is heading for a software design crisis, when scientific programs of over a million source-lines and over a thousand entry-points, written by over 100 people distributed over 30 sites, are being constructed.

This *Software Crisis* [Pre92] in Scientific Computing will not be solved by just the current Computer Aided Software Engineering (CASE) tools. More fundamental work will be needed involving the creation of new software design approaches supported by a solid base of formalism and experimentation. In this context object-oriented design and implementation [Boo91] have significant implications for the scientific computing community, as many aspects of the object-oriented paradigm appear to be an appropriate solution for structuring large and complex software systems to support scientific computation.

The Remote Object Invocation (ROI) model [NOLTE92] which evolved from Remote Procedure Calls (RPC) [CK89], provides a good solution for integrating multiple machines into a single computing environment. The RPC approach is a client/server arrangement where the client is an application that issues calls and the server is the resource that handles the call. In the ROI model, clients and servers are existing objects distributed over multiple computing systems.

The Abstract Data View (ADV) approach [CILS93a] also results in a client/server structure where the ADVs are clients interacting with servers which are abstract data types (ADTs) [Hoa89]. ADVs have been used in interactive applications as objects for structuring graphical user interfaces and this work is reported in [LCP92, CILS93a, CILS93b, CILL$^+$92]. Current research on the ADV concept is extending the model to support interfaces between any two interacting objects. In this context ADV programming can then be viewed as a client/server protocol implementation activity.

Language-level support for ROI using a C++ based language extension for distributed/parallel computing systems in an object model called *dual objects*, is introduced in [NOLTE92]. The ADV approach can use similar language constructs since "likenesses" in the dual object model are views of their associated "prototypes", while ADVs are views of their associated ADTs. The main difference is that an ADV is a free-form object representing any subset of its associated object's public part, while a likeness is a fixed form projection of its prototype public part. Actually, the client interacts with a remote server through a likeness (or ADV) as if it were local. This local characteristic provides a design approach for distributed applications, which could significantly reduce development cost and complexity.

In this paper we take advantage of this local property and present the design of a distributed solution to the ray tracing efficiency problem by using the ADV approach, the dual objects paradigm and remote object invocations. The approach is applied to the implementation of a parallel volumetric ray tracer [Levoy90] using a simple domain decomposition problem. The ray tracer domain is naturally decomposed, given the low coupling of the algorithm. The volumetric ray tracer has higher cost than the ordinary surface ray tracer because each ray traced from the object space to the eye is also sampled, in order to detect volume data.

Using the ADV/ROI approach the design proceeds in two steps. In the first step we create the system as if it were a sequential system, with all objects in the same conventional process. After creating a "working" conventional system, the second step consists of distributing the objects that compose the system over a network of parallel processors, which may be locally or remotely located. In this step the entire specification for the sequential version is reused, demonstrating how the ADV/ROI approach can be used to build parallel applications from conventional ones. Consistency must exist between the state of the objects and their visual representation and this consistency must be maintained in the distributed version of the application. The ADV approach ensures that this consistency property exists for both sequential and distributed versions of the program.

Although we have demonstrated the ADV/ROI approach in the context of the volumetric ray tracer, the mechanism is general enough to be employed in a large range of adaptive domain decomposition applications, by adapting the task redistribution criteria.

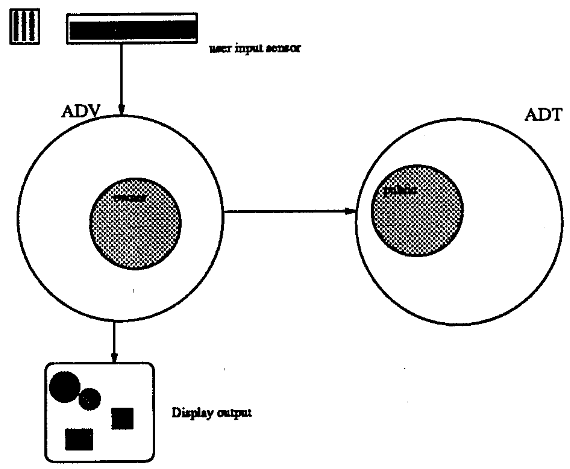


Figure 1: The ADV Architectural Model

2 The ADV Model and the Dual Object Paradigm

2.1 The ADV Model

The ADV model is a user interface model that cleanly separates the application from the user interface. A typical system based on the ADV architectural model consists of a collection of Abstract Data Types (ADTs) that manage the data structures and the state of the application, a collection of ADVs that comprise the perceivable behavior and handle both views and events, and a mapping between the ADVs and ADTs. Thus, the ADVs are clients of the server ADTs. The mapping associates the views of variables in the ADV with the state of the variables in the ADT. Thus, any change in a variable initiated from the ADV will also be modified in the ADT, and any change of a variable in the ADT will be reflected in its view in the ADV. The changes in the variables in the ADV are usually a consequence of an event that are generated by input from devices operated by the user. The diagram in Figure 1 illustrates the ADV, ADT, and the mapping that is shown as an arrow connecting the ADV to an ADT.

The ADT consists of a public interface and a body. In the body we describe the functions that are public, use public variables, as well as define variables and functions that are private to the ADT. The ADT is completely independent of the user interface, and in fact does not access any input or output. The ADV provides all the input and output functions and fully controls its associated ADT. The ADV implements all decisions related to the information exchanged between the user and the user-interface application, and all interactive objects.

The ADV model allows nesting of ADVs and ADTs, both of which can be viewed as objects. However, the model does not enforce any implementation-biased approach for the objects and the nesting mechanism. The nesting is simply the fact that an ADV object can include ADV objects, and an ADT object can include ADT objects.

In the ADV model, several ADVs can be associated with a single ADT; each can provide a different view of the ADT, or different control functionality. Since an ADT has no knowledge of input or output, it does not need to refer to any ADV. As a consequence, there is *not* a symmetrical arrangement with the ADT.

In the ADV architectural model the mapping between an ADV and an ADT is represented by the variable **owner**. The variable **owner** represents the connection between the user-interface and the non-user-interface application and is the name of the ADT instance associated with an ADV instance.

An ADV instantiation is associated with one and only one ADT instantiation. However, an ADT instantiation can be associated with several ADV instantiations. In other words, an ADV instantiation or user interface owns one and only one ADT instantiation; whereas an ADT instantiation may be owned by several ADVs instantiations or user interfaces. This kind of relationship guarantees that a certain ADT can be viewed in different ways, but the different views are consistent with the state of the single ADT.

Because the association relationship comes from the ADV, and not the other way around, the **owner** variable can be in the ADV for specification purposes. Roughly speaking, an ADV observes and manipulates the ADT which is its owner; from the point of view of the ADT, the observations are invisible and the manipulations are anonymous.

In systems based on the ADV architectural model, the control originates in the user-interface application rather than in the non-user-interface application. This placement of control is appropriate, since in highly interactive systems, the flow of control is determined primarily by the user of the system. The use of multiple ADVs, one for each independent view, makes the flow of control of each ADV quite simple: each ADV responds to a relatively small number of user-generated events and examines and manipulates one ADT.

2.2 The Dual Object Approach

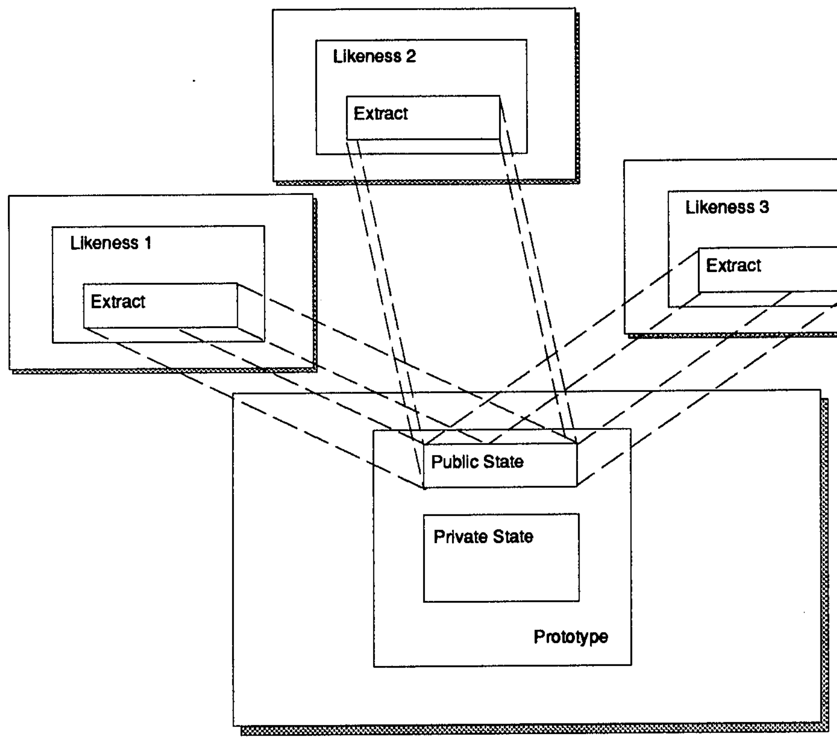


Figure 2: Dual Object Model

Dual objects [NOLTE92] provide a simple and efficient mechanism to make transparent the sharp distinction between local and remote object access. From an application's point of view, each dual object combines both the aspects of an individual local object instance as well as a globally sharable instance at the same time.

A dual object consists of two closely related but physically separated parts: the prototype and the likeness. A prototype represents the internal view of an object, whereas the likeness represents its outer appearance. Furthermore, the prototype logically has both a public and a private state. A likeness is built by extracting the public state, encapsulating this extracted state and exporting it to the requested external context. This dual object concept is illustrated in Figure 2.

Thus, an application has access to a remote object by manipulating one of its likenesses. The application can use a likeness just as if it were the prototype itself. However, when a likeness is manipulated, its public state becomes inconsistent with the prototype's state. When the prototype is invoked, before any operation on the prototype is executed, a unification mechanism is used to re-establish consistency. We call *vertical consistency* the ability of a prototype and one of its likenesses to have compatible states.

Furthermore, a dual object can have one prototype and a set of likenesses for that prototype. Once vertical consistency between a prototype and a likeness is established, this event must be propagated to the other likenesses associated with that prototype, so that vertical consistency can be established between the prototype and all the other associated likenesses. We call this constraint *horizontal consistency*, which means that the set of likenesses are compatible with the underlying prototype. To maintain this kind of consistency, the model assumes the existence of an application dependent consistency protocol.

Another interesting concept in this paradigm is the Clerk entity. Clerks are prototype managers and are analogous to the RPCserver object described later in this paper. Likenesses communicate with Clerks in order to access prototypes. All actions appear as if the user were accessing the prototype, but in fact, the user is accessing a likeness that accesses a Clerk that accesses the prototype. The Clerk is a "system entity". The programmer never makes references to Clerks in the program specifications, as it is automatically generated for each dual object class.

2.3 Relating Abstract Data Views and Dual Objects

The Abstract Data View and Dual Object models are similar, since both models provide a way to manipulate objects as dual entities. One entity corresponds to the "user interface" or client aspects of the computation, while the other entity is the server which supports the state of the computation and the processes to manipulate that state. Likenesses and ADVs which represent the "user interface" entity constitute almost the same concept, except for the fact that ADVs are free-form objects which access ADTs by making use of references, and likenesses are "projections" of their associated prototypes in an address space and access them through *extract* structures. Prototypes and ADTs represent the server and are actually the same concept.

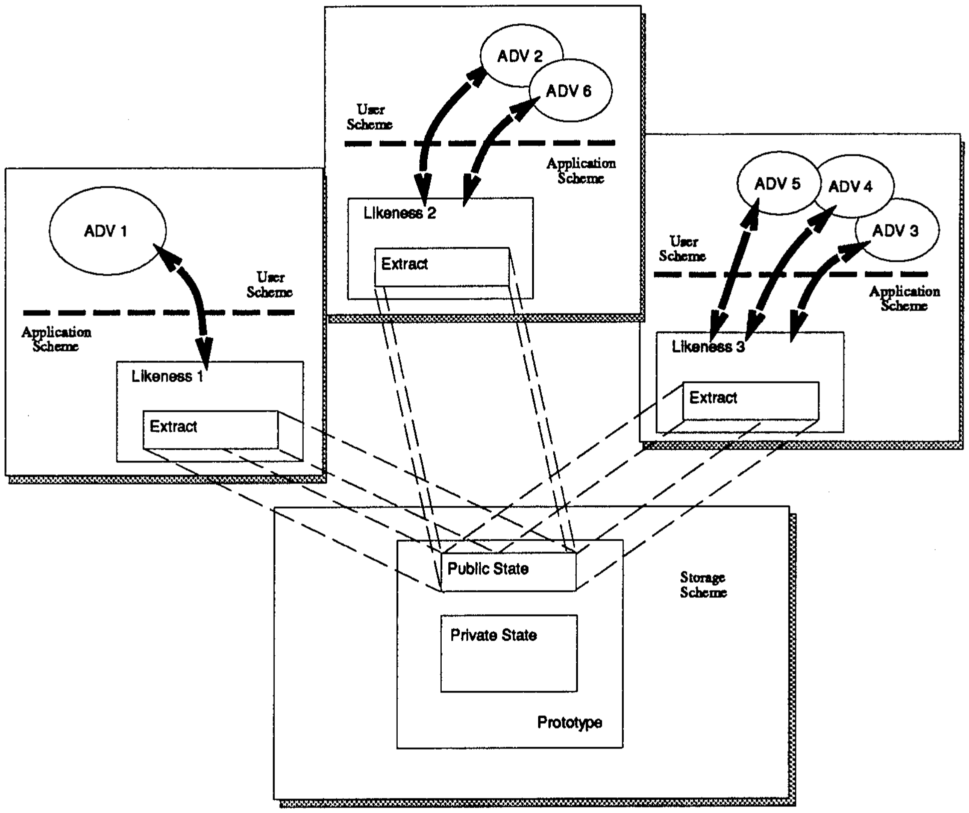


Figure 3: Object represented in multiple schemes

Although the dual object and ADV approach are similar there are a few key useful differences. Research in ADVs concentrated on user interface aspects of objects while establishing the use of object/view pair as a basic design model. Views of an object were assumed to be free-form entities, since they were meant to be customized representations. This approach leads to the concept of *flexible views* of objects.

On the other hand, the Dual Object model focuses on language support at the operating system level, where a likeness is interpreted as a *rigid view* of an object. This rigid view concept is quite useful, because rigid views can be generated automatically by the operating system and used as basic building blocks to support flexible views or ADVs. Automatic generation of rigid views greatly simplifies distributed application programming. RPC applications use a similar approach where an RPC code generator creates the low-level RPC code.

We conclude that we can combine the dual object and ADV models, by considering likenesses as rigid views of prototypes, which are ADTs. Further we can assume that these views can have other representations or flexible views in other system layers or representation domains such as a user interface. Figure 3 illustrates this point by showing an example of an object (prototype or ADT) with multiple representations in multiple schemes. The operating system is supposed to handle inconsistencies between these objects, in order to maintain their compatibility.

A relevant characteristic of the dual object model is that prototypes and likenesses are not specified as separate entities. The programmer specifies the prototype, using an "annotation" to declare whether a method is executed by the prototype or by the likeness. Then both specifications are combined. This approach does not establish a clear separation between representation schemes. Conceptually the representations are different entities, but they are combined in the specification. This is a major difference between the two approaches. The ADV model maintains the clear separation between entities at all times, while in [NOLTE92] separation is implicit.

Our research with ADVs has enforced the idea that since a view is an interface ("visual" specification) of an abstract data type, it "drives" or owns the ADT. Unfortunately, the communication between objects and their views at the implementation level is usually not trivial, if one wants to maintain vertical and horizontal consistency. At the implementation level ADTs and associated views need to share a "friendship" relation. Especially in the parallel paradigm, the model should not assume that there is a master/slave relationship, instead, the model should support cooperation.

As a consequence, in the ADV model, we must specify an object so that horizontal and vertical (H/V) consistency will be maintained with current and future views. Our solution consists of defining H/V consistency as an invariant property in the specification phase, and mapping the consistency mechanism to the implementation phase. A complete formal discussion on V/H consistency is presented in [LCP93] in terms of binary relations. One of the most important results is Theorem 2.1.

Theorem 2.1 *Given an abstract data type x, all abstract data views of x are vertically consistent with it if, and only if, any two abstract data views of x are horizontally consistent.*

```
Specification $ADV$
    Subtype of $GUIObject$
    declaration  $main\_window$: $Window$
                 $owner$: $Interactive$
    Invariant: $inv\text{-}ADV \triangleq V\_consistent(self, owner)$
    Constructor $CreateADV$ ($iobj$: $Interactive$)
    Operation $UpDate$ ()
    post    // ADV subclass dependent procedure
End $ADV$
```

Figure 4: A Model ADV Specification

3 The ADV/ROI Design Environment

The ADV/ROI design environment consists of a set of abstract classes which are used to build interactive client/server based applications. These classes include the ordinary ADV hierarchy [LCP92], and the new ones: *Client*, *Server*, *ClientHandler*, *ServerHandler*. Horizontal consistency is a consequence of the *Interactive* invariant (see Theorem 2.1). The specifications in Figures 4 through 7 describe these classes in a VDM-like notation [AI91, Ier91a] similar to a programming language, so no familiarity with VDM is required of the reader.

The Client class described in Figure 6 is a protocol definition for the interface between a *ClientHandler* and a *Client* subclass instance. Similarly, the *Server* class in Figure 6 is a protocol definition for the interface between a *ServerHandler* and a *Server* subclass instance. Horizontal consistency is maintained by issuing *UpDate* calls after writing to a *Server*. The mechanism defined here for updating purposes is a very general solution. Different mechanisms can be implemented by redefining the *UpDate* operation defined for the general object : *Client*.

The *ClientHandler* and *ServerHandler* described in Figure 7, cooperate to establish the communication between a *Client* subclass instance and a *Server* subclass instance. A *ClientHandler* is located at the same address space as its associated *Clients*, and a *ServerHandler* is located in the same address space as its associated *Servers*. Furthermore, we consider that there is only one *ClientHandler* and one *ServerHandler* per address space (note that there can exist more than one

```
Specification Interactive
    declaration   $class\_name$: $Char^*$
                  $advs$: $ADV^*$
    Invariant: $inv\text{-}Interactive \triangleq \forall view \in \text{elems } advs \cdot V\_consistent(view, self)$
    Constructor CreateInteractive ($name$: $Char^*$)
    Operation UpDate ()
    post   for $i \leftarrow 1$ to len $advs$
           do $advs[i].UpDate()$
           end
End Interactive
```

Figure 5: The Specification of the ADT Interactive Class

```
Specification Client
    declaration   $owner$: $Server$
                  $ch$: $ClientHandler$
    Invariant: $inv\text{-}Client \triangleq V\_consistent(self, owner)$
    Constructor CreateClient ($ownr$: $Server$)
    Operation UpDate ()
    post   // Client subclass dependent procedure
End Client

Specification Server
    declaration   $clnts$: $Client^*$
    Invariant: $inv\text{-}Server \triangleq \forall client \in \text{elems } clnts \cdot V\_consistent(client, self)$
    Constructor CreateServer ()
    Operation UpdateClients ()
    post   for $i \leftarrow 1$ to len $clnts$
           do $clnts[i].UpDate()$
           end
End Server
```

Figure 6: The Specification of the Client and Server

address space in the same host).

Actually, both the *Client* and *Server* are not aware of their enclosing address spaces. A *Client* communicates with a *Server* as if there is no intermediate layer between them. The handlers are responsible for communicating between address spaces and so they contain maps from their remote customers to the associated remote handlers. For example, if an instance of a class *A_Client* sends a request for an instance of a class *A_Server*, the request is intercepted by the local *ClientHandler*, which delivers the request to the correct *ServerHandler*, based on the *Server* $\times$ *ServerHandler* map. The *ServerHandler*, in turn, captures the request, forwarding it to the appropriate *Server*. If the *Server* needs to send a reply message, the *ServerHandler* intercepts the reply, delivering it to the correct *ClientHandler*, based on the *Client* $\times$ *ClientHandler* map. The *ClientHandler* will then forward the reply to the original *Client*. This communication mechanism is illustrated in Figure 8.

Note that the *Client* and the *Server* are abstract classes, which means that they are realized only through their respective "concrete" subclasses. In the ADV/ROI model, the programmer can specify the remote ADTs without knowing about the lower layers of network or bus communications, or even client/server relations. These issues are postponed until the *Client* and *Server* subclasses are specified. For example, the programmer can specify a program A as an ordinary local object, and a *AView* as a simple ADV for A, by extracting its public members. To make A distributed, the programmer need only create a subclass of A and *Server* using multiple inheritance. The resulting program called *A_Server* would act as the server, and a multiple inheritance subclass of *AView* and *Client* called *A_Client* would act as the client.

This approach provides both modularity and transparency in building parallel applications. Reuse is also possible by applying to the *Client* and the *Server* domains, the same composition operations. This model enables its users to create distributed objects too. As illustrated in Figure 9, the object C is composed of two remote objects A and B, placed in different remote hosts. For the user of object C, everything happens as if C were all local.

This description of the programming process fits the model described in the previous section by including vertical consistency in the invariant definitions. Theorem 2.1 ensures that vertical consistency between all objects related by the **owner** variable is enough to achieve horizontal consistency. Thus, Clients and Servers are consistent, as well as ADVs and Interactive objects.

```
Specification ClientHandler
    declaration   $srvhs\colon ServerHandler^*$
                  $clnts\colon Client^*$
                  $ss\colon Server \xrightarrow{m} ServerHandler$
                  $msgs\colon MsgQueue$
    Constructor CreateClientHandler ()
    Function Create: $ServerType \times Host \to Server$
    Create (srvtype, hst) $\triangleq$ // ask the ServerHandler in Host hst
                                  to create a Server instance and return its id
    Operation Send (sv: Server, msg: Message)
    post    // send message in msg to the server sv
    Operation UpDate (clnt: Client)
    pre     $clnt \in$ elems $clnts$
    post    $clnt.UpDate()$
End ClientHandler

Specification ServerHandler
    declaration   $clnths\colon ClientHandler^*$
                  $srvs\colon Server^*$
                  $cc\colon Client \xrightarrow{m} ClientHandler$
                  $msgs\colon MsgQueue$
    Constructor CreateServerHandler ()
    Function Create: $ServerType \to Server$
    Create (srvtype) $\triangleq$ Create Server of srvtype and return its id
    Operation Run ()
    post    // handle incoming requests, delivering them
            // to the appropriate servers
End ServerHandler
```

Figure 7: The Specification of the Client and Server Handlers

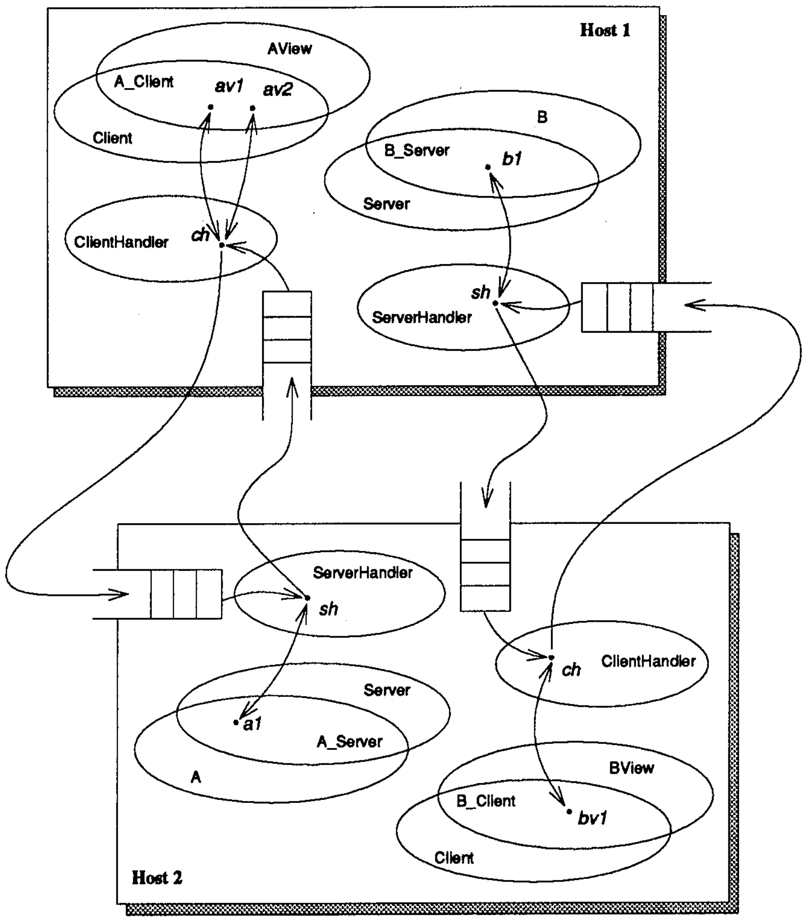


Figure 8: ADV/ROI architecture

This means that if all related objects are consistent, all objects in a process space can be considered to be local.

As shown in Figure 8, if the object *av*1 is consistent with *a*1 and *b*1 is consistent with *bv*1, then an object in **HOST 1** can use these objects as if both *a*1 and *b*1 were in its address space. Similarly any object in **HOST 2** can use these objects as if both *a*1 and *b*1 were in its address space. This transformation is illustrated in Figure 10 from the viewpoint of **HOST 1**. The transformation can be proved to be valid by showing that any view preserves its owner's invariants, once they are vertically consistent.

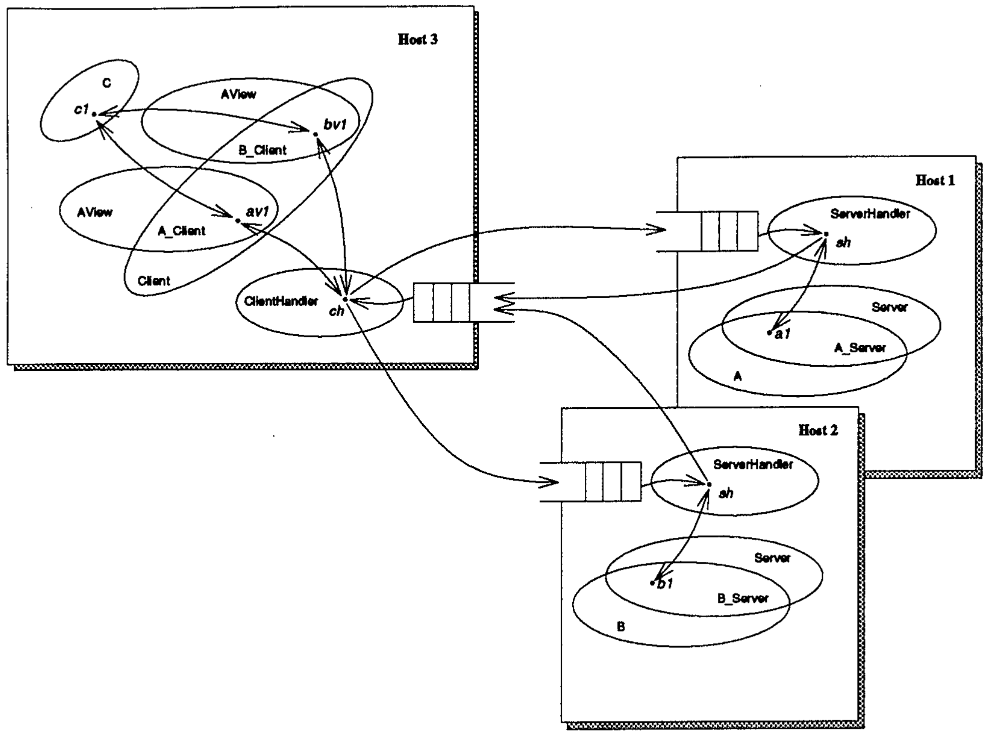


Figure 9: Distributed Object

3.1 The Implementation

The current implementation of this architecture was constructed using a set of classes created by Barry A. Warsaw[1], which handles client/server connections using remote procedure calls. This set of classes, as well as the ADV/ROI implementation, were constructed using C++, on Unix.

The implementation has lead us to create some abstract classes on top of Warsaw's structure, in order to create an environment compatible with our design approach. A significant part of this system was taken from examples of *rpcgen* generated code (which is available with RPC). The first implementation was developed on Sun SPARCstations and was ported to IBM RISC System 6000 machines, allowing parallel applications to be distributed over a network of different machines.

[1]Warsaw's implementation is free software that can be redistributed and/or modified under the terms of the GNU General Public License as published by the Free Software Foundation. Warsaw can be reached at bwarsaw@cen.com.

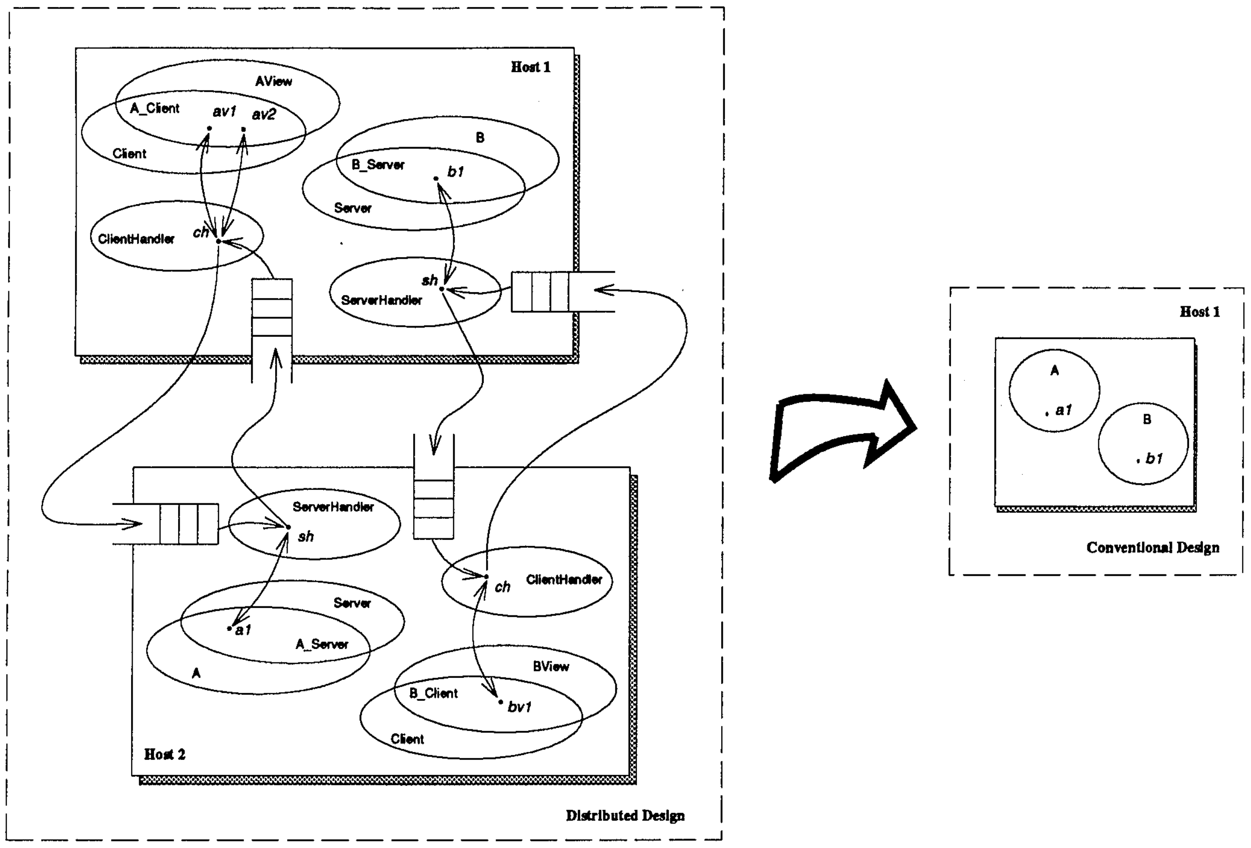


Figure 10: Transformation from distributed system to local system

4 An Application: A Simple Parallel Ray Tracer

This section describes some important aspects of a simple parallel Ray Tracing algorithm discussed earlier. The design notation used for this example is an extension of the notation defined in [CRC92] to include concurrency. The definition of an object-oriented notation supporting concurrency is a subject of current research.

The algorithm described next is based on the work of Marc Levoy [Levoy90] in volumetric rendering. Implementations of the algorithm can be naturally decomposed into independent subproblems that can be distributed over a network of renderers. The method for distributing subproblems is presented in the Section 4.2.1. An image produced by the example is shown in Figure 13.

The following subsections present two versions of a volumetric ray tracer. First, a sequential ray tracer is specified. Then a parallel version of that ray tracer is described, by adding some client/server subclasses to the classes defined in the sequential version. All the specification for the sequential version is smoothly reused, showing how the ADV/ROI approach can be used to build new parallel applications from conventional ones.

4.1 Sequential Ray Tracer

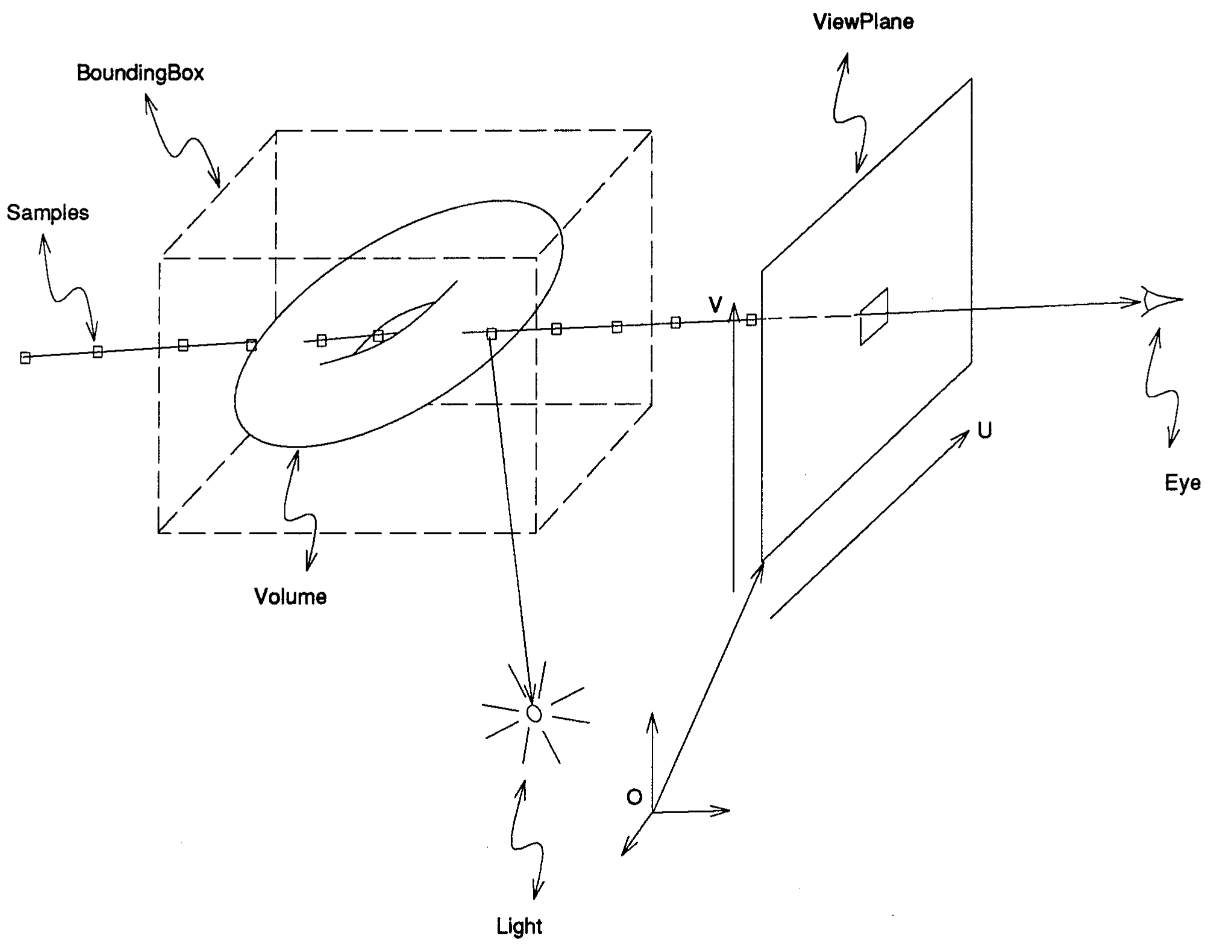


Figure 11: Volumetric Ray Tracer Objects

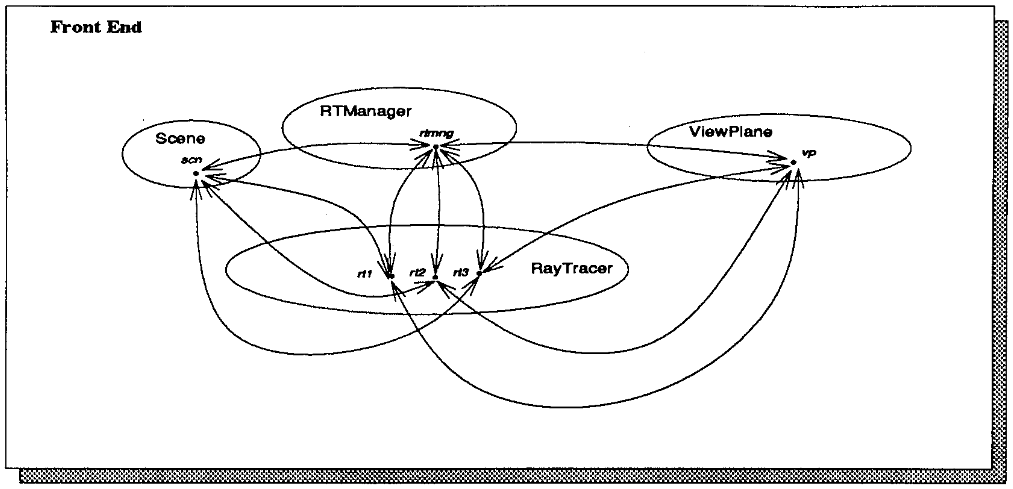


Figure 12: The Conventional Ray Tracer

The basic structure of the system is illustrated diagrammatically in Figure 12. At this level we do not make any reference to the ROI objects. Distribution is achieved by creating subclasses of these objects and the ROI objects. This step will be described in the Section 4.2.2. The basic classes that constitute our simple ray tracer system are:

RTManager. The actual program that manages the entire ray tracer system is called *RTManager*. In the distributed parallel system, the *RTManager* is a client of a set of remote *RayTracers*.

RayTracer. The *RayTracer* is composed of a set of simple objects such as the ones shown in Figure 11. *RayTracer* is responsible for the rendering calculations.

ViewPlane. The *ViewPlane* is the finite projection plane containing the vectors in the Euclidian space, which correspond to pixels in the resulting image. This plane is defined by three vectors. The H vector is the plane "lowest left corner". The U and V vectors define the plane "grid" as well as the plane direction. The Ray Tracer algorithm uses this plane by requesting the Euclidian three-dimensional space coordinates for its points, based on two-dimensional normalized coordinates, by activating the *ViewPlane vector* function. The number of uniform intervals in the normalized coordinates, combined with U and V magnitudes, give the point sampling for the rendering.

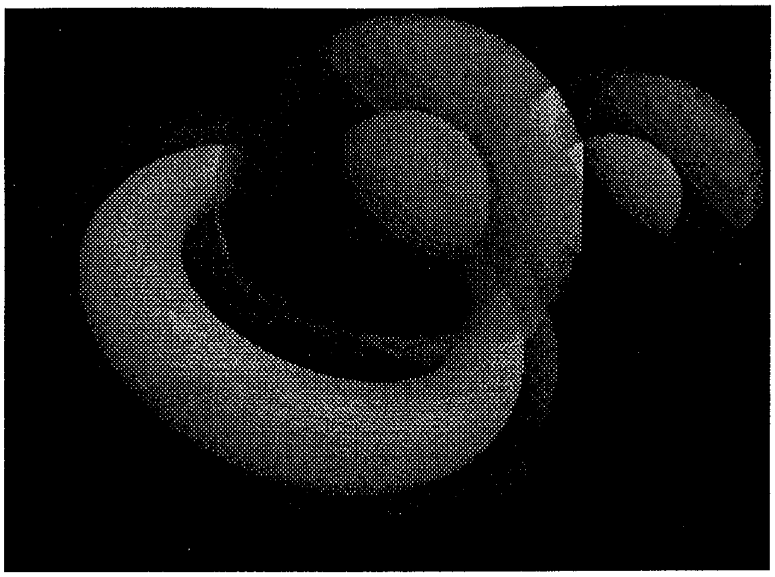


Figure 13: Image produced by the parallel ray tracer

Volume. A *Volume* is an abstract entity. It specifies the functions required by the ray tracer to render an object (or a *Volume* subclass). Figure 13 shows two *Volume* subclasses: two instances of a sphere volume, and one instance of a torus volume. These objects have non-constant opacity in their interiors.

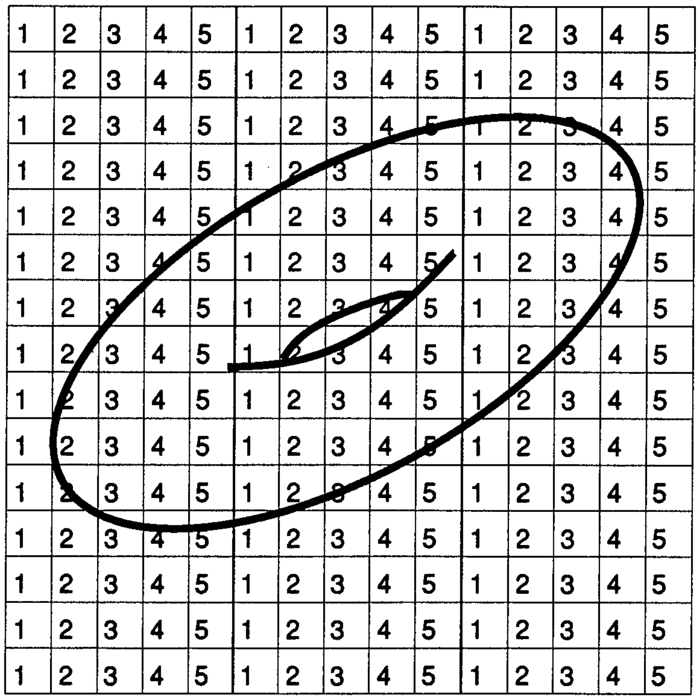
SceneVolume. A *SceneVolume* is an object containing a list of *Volumes*, a *Light* source, and some attributes required for Phong [Phong75] shading. A *SceneVolume* is itself a *Volume* subclass, so that the ray tracer is able to render it. The ray tracer does not differentiate between a *Volume* and a *SceneVolume*.

4.2 A Parallelization Method

4.2.1 Domain Decomposition

Domain Decomposition is a method for dividing an algorithm domain into subdomains and distributing these subdomains over the system processes. In the case of Ray Tracing, the domain is composed of the set of "pixels" on the ViewPlane in a Scene. There are two problems associated with this method. One problem is generating a distribution that is as uniform as possible. The other is decomposing the domain such that the communication between processes is at as low a level as possible. Normally there will not be a simultaneous solution for both problems. Actually, this method provides improvement in both areas when the decomposition is adaptive.

In the Ray Tracing scheme, the processes will be represented by the RayTracer objects. Each RayTracer will receive a set of messages, requesting a calculation over a ViewPlane subdomain. Once all RayTracers need to manipulate the same Scene, the objects will be shared[2] by them. The adaptive domain decomposition is managed by an RTManager, which controls the allocation of RayTracers and the task distribution across the system.



1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
1	2	3	4	5	1	2	3	4	5	1	2	3	4	5

Figure 14: ViewPlane decomposition in 15 × 15 subdomains for 5 RayTracers (5 processes)

The decomposition depends on the pixel resolution and the distribution depends on the number of RayTracers available. For example, in Figure 14, the ViewPlane is decomposed into 15×15 pixels, which are distributed over 5 Ray Tracers. The first distribution is done so that the process load is

[2]Once our implementation considers a MIMD structure, the Scene must be replicated in all nodes, in order to achieve sharing.

as uniform as possible. In the ray tracer decomposition example, line granularity is a reasonable solution, since pixel granularity increases interprocess communication unnecessarily. Thus a set of scan-lines are sent to each process. However, since our environment assumes a network of different machines, and the different sets of scan-lines may cause different amounts of processing[3], some processes may become idle. In that case, adaptive decomposition should redistribute processing tasks.

4.2.2 Distributing Objects

The second step in parallelization is distributing the subdomains created in the *domain decomposition* step over a network of objects which perform tasks on the respective subdomains. Consider the network of objects as being a distributed parallel system. In the ADV/ROI model it is possible to build a parallel system in two other substeps. The first is creating the system as if it were all a sequential system, with all objects in the same conventional process. After creating a "working" conventional system, the second substep consists of distributing the objects that compose the system over a network of parallel processes that may be locally or remotely located.

The quality of the parallel system depends on the degree of parallelism and communication between entities resulting from the system configuration. The next section suggests a simple distributed configuration for the volumetric ray tracer defined previously.

4.3 Parallel Ray Tracer

The classes defined in this section are the result of distributing the objects defined in the sequential ray tracer specification. The parallel ray tracer scheme is illustrated in Figure 15.

SrvrRT. This object is responsible for rendering a subset of a Volume by rendering equally spaced sample lines of a viewing plane.

RTView. This is just a simple interface to an actual ray tracer. It can be seen as an ADV for a *RayTracer* object.

[3]One scan-line requires more calculations than another if the plane defined by the former line and the observer intercepts a larger area, in the visualized volume, than the one defined by latter line and the observer.

VPView. This is just a simple interface to an actual ViewPlane. It can be seen as an ADV for a *ViewPlane* object.

ClntRT. This is a client of a remote ray tracer. The *ClntRT* is used to send commands to the *SrvrRT* which performs the volume rendering.

ClntVP. The ViewPlane Client (*ClntVP*) is a local representation of a remote ViewPlane.

SrvrVP. The *SrvrVP* is the object that actually stores the image produced by several *SrvrRTs*. It contains a reference counter, so that it stores the image only when the last *ClntVP* asks it to close. Before that, the *SrvrVP* keeps the image in its buffers.

MasterRTManager. The actual program that manages the entire ray tracer system is called *MasterRTManager*. In the distributed system, the *MasterRTManager* is a client of a set of remote *RTManagerServices*.

RTManagerService. An instance of this class is responsible for handling one *RTserver*, which handles a *TClntVP*.

Actually, RTManager is not an ADV, it is the application itself. The RTManager visualizes the servers through their views. The ViewPlane is not an ADV, it's an ADT. The ADV for the ViewPlane is VPView, which combined with ROIClient forms a ClntVP. The RayTracer (ADT) uses VPViews (ADVs for a ViewPlane) to write pixel colors on them. The writing to VPViews activates the V/H-consistency mechanism for the dual object ViewPlane, so that the ViewPlane will contain the union of the pixel colors from all VPViews. The volumetric ray-tracer will not affect the current structure. It will only change the way rays are traced into the scenes, and the way scenes respond to that tracing.

One of the big advantages of using ADVs is that there is no mixing of representation levels. If we were not using ADVs, the application clients would have to worry about RPC layers of communication to communicate directly with the servers. They would have, for example, to know the server locations, connection channels and so on. Furthermore, the clients would have to handle horizontal and vertical consistency themselves. Using ADVs we encapsulate V/H-consistency management in the ADV x ADT communication layer, to which the application client does not need to have access.

In this example the class ClntRTView, which is a subclass of RTView and Client, and the class ClntVP, which is a subclass of VPView and Client, are likenesses. They are built by extracting the public state of their correspondent servers. Specifically they are built by creating an ADV, which is built by extracting the public state of its ADT, and using it as a baseclass, together with the class Client, for composing a likeness. We can say that an ADV built in this manner and composed with a Client by inheritance, generates a likeness (in our implementation). Similarly a pure ADT, composed with a Server by inheritance, generates a prototype.

Note that there is no graphical user interface in this example. This application is supposed to generate a file in some image format. An interesting GUI for that application would be another type of ADV for the ViewPlane, which would not be the extraction of its public state, but the graphical representation of its pixels on a window on the screen. However, if we use a standard format, there is no need for a GUI, since there are probably many image viewers on the network.

Since vertical and horizontal consistency is supposed to be maintained by the underlying ADV/ROI environment, the correct transition from the "conventional" sequential version of the ray tracer into the parallel version can be proved by determining the correct configuration of the **owner** relationships. The transformation of the structure in Figure 12 into the one in Figure 15 is valid, considering that the objects $srt1$, $srt2$ and $srt3$ are consistent with $crt1$, $crt2$ and $crt3$(so that they can be seen as $rt1$, $rt2$ and $rt3$ residing in the front end), and that $cvp1$, $cvp2$ and $cvp3$ are consistent with svp (so that they can be seen as sv residing in the front end also).

To achieve load balancing, SrvrRT objects contain state flags indicating whether they are "idle" or "busy". The ClntRT objects can be notified when SrvrRTs change to an idle state through the use of semaphores. When a SrvrRT becomes "idle", the consistency mechanisms ensure that all its associated ClntRTs also become "idle". When this event happens to a ClntRT, it is put into a *Ready Queue* in the corresponding MasterRTManager scheme. The MasterRPManager in turn, redistributes the problem domain to ClntRTs included in that queue.

A SrvrRT may own more than one ClntRT. Thus, when the MasterRTManager makes one of the clients "busy" by assigning a task, the consistency mechanism propagates this event to the owner

SrvrRT, and then to the other owned ClntRTs. The ClntRTs become "busy" and are removed from the corresponding *Ready Queues*.

Even though we describe the "idle" state modification mechanism in the context of the volumetric ray tracer, the mechanism is general enough to be employed in a large range of adaptive domain decomposition applications, by adapting the task redistribution criteria.

5 Related Work

A similar approach for developing distributed applications is discussed in [GKM93]. Applications are designed in two main steps. First, the objects are designed without considering how they are to be distributed, and then the objects are distributed in the second step. The difference is that the ADV/ROI approach provides support for design based on formal specifications, so that the second step can be proved to be correct.

In the work cited in [GKM93], the local representation of remote objects are called "surrogates", which are similar to the "likenesses" in the Dual Object paradigm. Also, instead of Remote Object Invocations, Remote Method Invocations which access objects states by using method invocations, are used as the basic communication mechanism. We believe our model presents a more general approach, since it allows local representations of remote objects to have directly accessible states.

6 Conclusions

This article presents a new approach to parallel programming using the Abstract Data View approach combined with remote object invocations. A simple concurrent environment was described by means of its design specification and corresponding implementation. The environment was successfully implemented on a network of workstations with different architectures. Object-Oriented Programming is assumed to be the correct underlying paradigm for the model.

The parallel volumetric ray tracer was implemented based on the design described in this document. Given the low coupling of the tasks in the algorithm[4], a significant improvement in

[4]Using domain decomposition, scan lines were distributed to the servers for ray tracing, and in the volumetric ray tracer algorithm, pixels are all independent from each other.

performance was detected. The two step approach in developing the system proved to be a natural way of creating parallel applications. Systems can be designed with no reference to their distribution. Then the resulting design can be naturally (but methodically) extended, using ADV/ROI, to include concurrency.

The correctness of the design extension can be proved by observing that the consistency properties are kept valid during the design implementation. If all objects connected by the **owner** relationship are consistent (see Theorem 2.1), then all objects in each process space, which are local representations of remote objects, can be *used* as if they were the remote object itself. This means that, once vertical consistency is established for all related objects, each process space can be considered to be a "conventional" system. Finally, we conclude that consistency properties force the existence of a set of mappings from the parallel system design to its corresponding "conventional" design, so that both designs can be proved to be compatible.

The prototype for the ADV/ROI environment presented in this document is just the beginning of our work with distributed objects. The current implementation was developed on Sun SPARCstations and IBM RS6000, allowing concurrent execution of object oriented client/server applications on a network of different machines.

Multitasking on the client and server sides was tested but not encapsulated in the *ROIClient* and *ROIServer* entities. Some successful experiments were made using *fork* and other related system calls. However, the *fork* call makes a complete copy of the parent process which is not a very good policy.

A better solution could use *Light Weight Processes* (LWP), which make selected copies of the state of the parent process through a shared-memory approach. This seems to be the correct technique for implementing multitasking on clients and servers. In the current implementation, this capability can be implemented in the *ROIClient* and *ROIServer* subclasses. Finally, Nolte's solution for declaring different types of invocations (in, out, global, trigger ...) appears to be the correct way of selecting the invocations' behavior, thus it will be included in the ADV/ROI implementation.

References

[AI91] D. Andrews and D. Ince. *Practical Formal Methods with VDM.* McGraw-Hill, 1991.

[CILS93a] D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, and T. M. Stepien. Abstract Data Views. *Structured Programming,* 14(1):1–13, January 1993.

[CILS93b] D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, and T. M. Stepien. Application Integration: Constructing Composite Applications from Interactive Components. *Software Practice and Experience,* 23(3):255–276, March 1993.

[CILL$^+$92] L.F. Barbosa, S.B. de Oliveira, R. Ierusalimschy, C.J.P. Lucena, D.D. Cowan. *A Program Design Using Abstract Data Views.* Technical Report, Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, Rio de Janeiro, RJ, Brazil, July 1992.

[Boo91] Grady Booch. *Object Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[CRC92] C.H.C. Duarte, R. Ierusalimschy, C.J.P. Lucena. *On The Modularization of Formal Specifications: The NDB Example Revised.* Technical Report, Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, Rio de Janeiro, RJ, Brazil, July 1992.

[CK89] C. Manson, K. Thurber. *Remote Control.* Byte, pp. 235–249, July 1989.

[DL93] D.D. Cowan, C.J.P. Lucena. *Abstract Data Views as a General Design Approach* Technical Report, University of Waterloo, Computer Science Department, Waterloo, Ontario, February 1993.

[GKM93] M. Guttman, J.A. King and J. Matthews. *A Methodology For Developing Distributed Applications.* Object Magazine, pp. 55–59, Jan-Feb 1993.

[Hoa89] C. A. R. Hoare. Proof of Correctness of Data Representations. In *Essays in Computer Science,* pages 103–115. Prentice-Hall, 1989.

[Ier91a] R. Ierusalimschy. *A Method for Object-oriented Specifications with VDM.* Technical report, Monografias em Ciência da Computação, PUC-Rio, February 1991.

[Levoy90] M. Levoy. *Volume Rendering by Adaptive Refinement* The Visual Computer, Springer-Verlag, pp. 2–7, 1990.

[LCP92] A.B. Potengy, C.J.P. Lucena, D.D. Cowan. *A Programming Model For User Interface Compositions* Anais do V Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens (SIBGRAPI'92), Águas de Lindóia, SP, Brazil, November 1992.

[LCP93] A.B. Potengy, C.J.P. Lucena, D.D. Cowan. *Distributed Abstract Data Views – Design and Implementation* Technical report (to appear), University of Waterloo, Computer Science Department, Waterloo, Ontario, February 1992.

[Nash] T. Nash. *Event Parallelism: Distributed Memory Parallel Computing for High Energy Physics Experiments.* Technical Report, Advanced Computer Program, Fermi National Accelerator Laboratory, Batavia, IL, USA.

[NOLTE92] J. Nolte. *Language-Level Support for Remote Object Invocations.* Arbeitspapiere der GMD, Gesellschaft Für Mathematik und Datenverarbeitung MBH, June 1992.

[Phong75] P. But-Tuong. *Illumination for Computer-Generated Pictures.* CACM, pp. 279–285, June 1975.

[Pre92] R.S. Pressman. *Software Engineering: A Practitioners Approach.* McGraw-Hill, third edition, 1992.

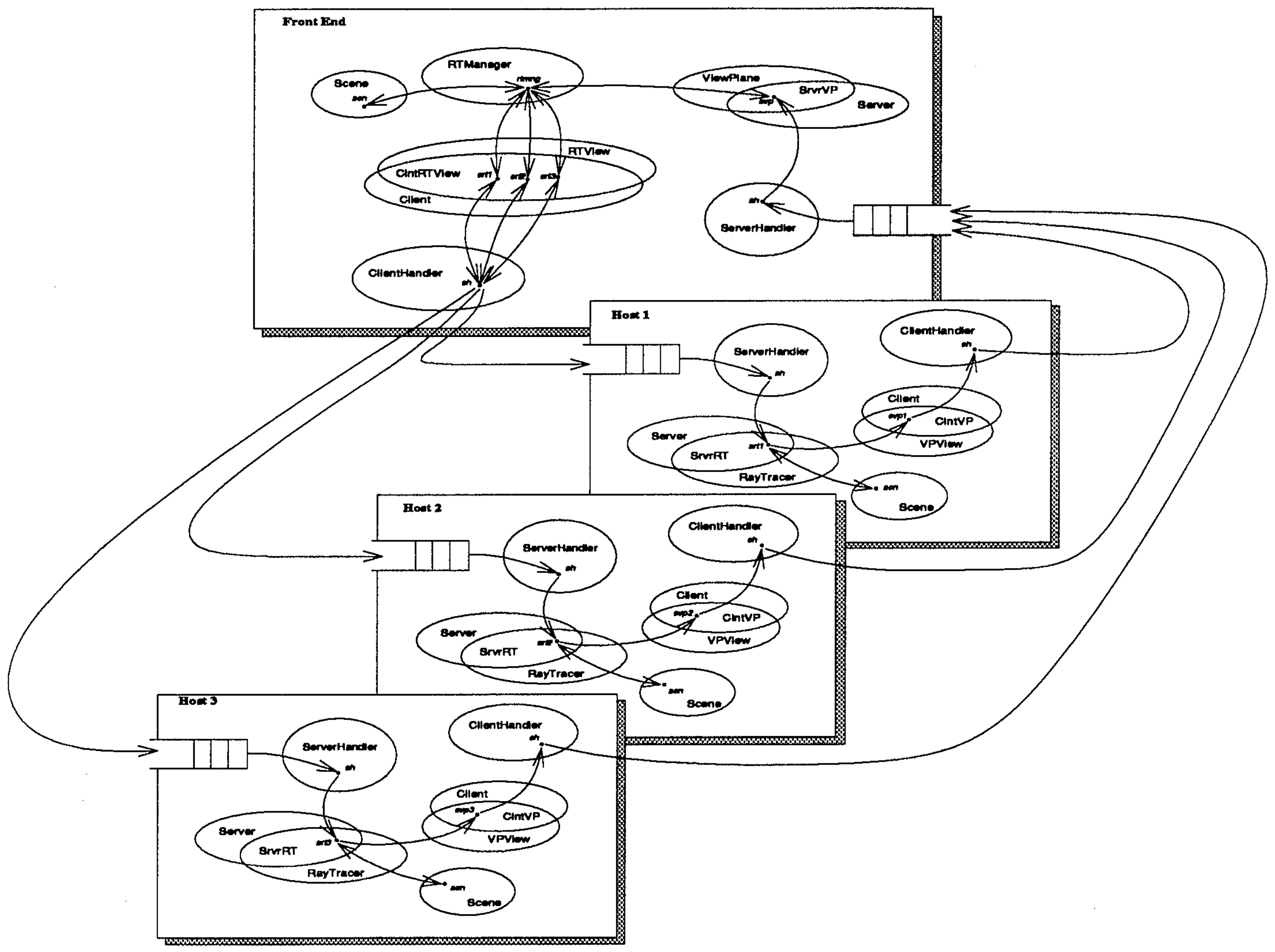


Figure 15: The Parallel Ray Tracer