Distributed Lighting Networks

Peter Kipfer
Computer Graphics Group, University of Erlangen-Nürnberg
Am Weichselgarten 9, D-91058 Erlangen, Germany
kipfer@informatik.uni-erlangen.de

Abstract

Simulating the physics of energy transport is a well established method for creating images from a description of a virtual scene. The algorithms developed over the past years, are complex and demand a lot of computing power. The Vision rendering framework provides a flexible, object–oriented architecture, based on the physical description of rendering. Its lighting network technique allows the simulation of complex lighting by composition of algorithms. This technical report presents a case study of applying an integrated approach to parallelization and distribution in a application–oriented way to the Vision rendering framework, with an emphasis on lighting calculations. The combination of a multi–threaded client–server model with asynchronous request processing, allows data–parallel work to be done, while taking advantage of functional parallelism. Using CORBA for the implementation of the distribution functions, and the existence of wrapper objects, makes distribution specific issues totally transparent to developers of traditional rendering and lighting algorithms, while providing support for programmers of advanced algorithms and for production purposes. The distributed system is completely configurable by Tcl–scripts. Despite the flexibility and general character of the distributed system, the overhead is kept small, and good speedup can be achieved.

1 Introduction

Since the beginning of computer graphics, creating images from virtual scenes has been a major research subject [Rog98]. Two main directions have evolved since then: the first one tries to approximate the illumination of a surface by applying algorithms and heuristics to local data and parameters. Though the achievable effects are limited, the purely local character of this approach permits efficient hardware implementation and easy parallelization through hardware replication.

The second approach to image creation is to simulate the physics of energy transport within the scene [SP94]. The algorithms, that have been developed for the global exchange of illumination information, are however quite complex and demand a lot of computing power. Therefore they are mostly software implementations on general purpose hardware. To reduce the execution time, parallelization and distribution of computations among multiple processing units is a current research topic.

Algorithms, computing global illumination effects are hard to integrate efficiently into traditional ray–tracing software, because of the way they access the scene data. Programs, that have been developed especially for a new illumination algorithm, on the other hand often fail to integrate traditional rendering functionality. This strong dependence on specific algorithms is a major drawback for both research and development as well as for a production environment. The Vision rendering framework [Slu96] offers a flexible, object–oriented architecture, which is strongly based on the physical description of rendering. It takes advantage of the common aspects of various rendering algorithms, while isolating their differences. The definition of frozen interfaces for the subsystems enables the rendering algorithms to be used as building blocks, that can be inserted at the appropriate places. They can be exchanged with each other without side–effects on other parts of the system.

This technical report presents an integrated approach to parallelization and distribution for the Vision rendering framework, with an emphasis on the lighting subsystem. It combines a multi–threaded client–server model with asynchronous request processing, communicating through CORBA. This allows data–parallel work to be done, while taking advantage of functional parallelism. The implementation of the distribution functions within base classes and wrapper objects makes the distribution issue totally transparent to developers of traditional rendering algorithms, while leaving it up to the developer of advanced parallel algorithms whether to use it, or not. In any case, the distributed system can be configured by Tcl–scripts to reuse traditional implementations for functional parallelism or data–parallel work. Additionally, because of it's minimal intrusion into the core of the Vision architecture, the whole distributed system extension can be removed by conditional compilation, making the Vision framework portable to non–UNIX, stand-alone environments, as long as there exists a ANSI C++ Compiler.

This technical report is organized as follows: the next section gives a very brief introduction to the Vision rendering framework and its lighting subsystem. In section 3, the system extension for distribution and parallelization is presented. A overview of the management and computation server classes is followed by a discussion of the central aspect of asynchronous communication, and how to apply it to lighting calculations. Section 4 illustrates the efficiency and flexibility of the distribution extension for various purposes with example configurations. After section 5 has discussed advantages, disadvantages and future work, section 6 concludes this technical report.

2 The Vision framework

The architecture of the Vision rendering framework closely follows what could be a description of the rendering process in natural language. The environment is divided into three major parts, containing eight categories, which describe the subsystems of the implementation. Figure 1 shows the resulting clear separation into geometrical description of the scene, simulation of global illumination exchange and sampling of the resulting lightfield.

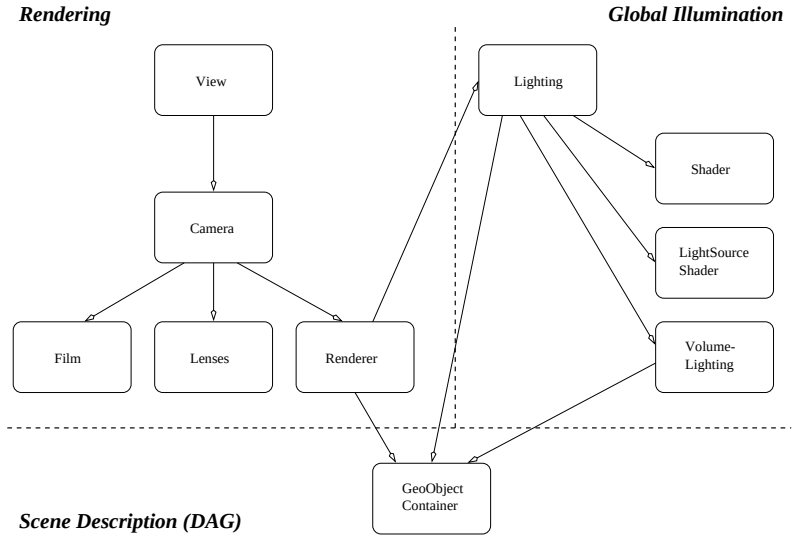


Figure 1: A class diagram [Boo94] of the subsystems of the Vision architecture and their *"uses"*–relation, taken from [Slu96].

- **GeoObject, Surface and Volume:** The abstract base class GeoObject contains the interface for the geometry subsystem. A scene description consists of objects derived from

Surface or Volume, describing primitives. Container objects can be used to build hierarchical structures. The geometry subsystem maintains the scene as a directed acyclic graph (DAG).

- **LightSourceShader:** This subsystem is able to describe the emission of light from a surface or volume.
- **Shader:** A shader describes the reflection of light at a distinct point on a surface. Using a unified description of ray–surface intersections makes this structure independent of the actual geometry of the intersected object, which can be queried from the geometry subsystem.
- **VolumeLighting:** This subsystem describes the general interactions between light and a medium. It is separated from the shader subsystem, because there are more effects to take into account, making the handling of surfaces easier to optimize.
- **Lighting:** A lighting object describes the global aspects of illumination within a scene. It uses the local descriptions of illumination provided by the above subsystems to compute the incident illumination at any point in the scene. The continuing work on Vision has brought up many implementations of the lighting subsystem. In the context of this technical report, the Lighting Networks [SSH+98] are of special interest.
- **View:** This subsystem describes a particular view of the scene. It triggers the rendering of an image with the help of a camera and a film object.
- **Camera:** With the use of the lenses system and the viewing parameters, the incident 3D–lightfield is projected onto the 2D–film.
- **Film:** With the help of the Renderer system, the 3D–lightfield is sampled and saved persistently.

2.1 Lighting Networks

Traditional rendering systems often implement lighting calculations within a monolithic algorithm. However, the effects expressed by Kajiya's rendering equation [Kaj86] can also be calculated independently. Multi–pass technology [CRMT91] is using this functional decomposition to obtain better results. This clearly is a starting point for working in parallel, but keeping track of dependencies within the result database is a problem for monolithic systems.

The lighting network [SSH+98] technology within the Vision framework provides an object–oriented way of dealing with functional decomposition for lighting calculations. It implements a lighting subsystem for Vision.

Illumination is represented to the Vision system through different local description formats (Illumination Basis — IllumBasis). A algorithm for lighting calculation can therefore be viewed as a Lighting Operator (LightOp) on a specific description. There are converter LightOps, which transform a IllumBasis into another one. To enable automatic configuration, the LightOps can be queried about the IllumBasis they support. The most common IllumBasis is the Point–Sampling Basis. The LightOps are connected to form a lighting network. Figure 2 shows a example network. In addition to a better understanding of what's going on within the lighting subsystem, this structure is simple to modify and thanks to object–oriented programming, easy to maintain[1].

The whole lighting network is managed by a special object called MultiLighting, that implements the lighting subsystem interface towards other Vision subsystems and behaving according to the facade design pattern [GHJV95]. A illumination request is forwarded to the LightOp "at the lower end" of the network, called the MasterLightOp. While performing his calculations, he queries his predecessor. This leads to a pull–driven data–flow through the network. The distributed system extension described in this technical report, is capable of fulfilling these communication tasks

[1] please see [SSH+98] for a profound discussion of formal requirements like problem domain decomposition, graph relaxation and BRDF compatibility issues.

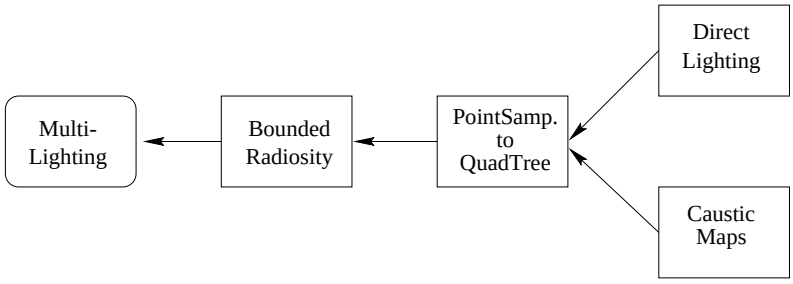


Figure 2: A example of a lighting network, correctly simulating the illumination effects, the Phong–shading algorithm tries to approximate.

asynchronously, enabling the lighting network to work with functional parallelism in the fashion of a pipeline.

3 The distributed System

The (reentrant) subsystems of the Vision framework lay the foundation to a new approach in distributed rendering. The extension focuses on the distributed management of these subsystems, while paying special attention to portability and minimal code changes of the traditional Vision classes. The basic infrastructure of the distributed Vision system [Peu98] consists of seven main components. Concurrency control and the communication infrastructure is implemented through five management server classes:

- **Session object:** This is the central control class for all activities of all objects, that work together in the creation of a image. There is exactly one Session object per scene description. It saves references to all Vision objects and implements the MultiLighting and the Session interface, following the facade design pattern, to coordinate work for all participating Renderer and Lighting objects.
- **Vision object:** It maintains the scene description, creates Lighting and Renderer objects and triggers their methods. It acts as a instance of a traditional Vision object for the Session and the HostManager objects. A Session object typically creates multiple Vision objects to make them work functionally parallel or concurrently on some data. The CORBA client launched by the user to start the system, is a Vision object executing a special control thread, which orchestrates the creation of the Session, triggers the rendering phase and controls the shutdown.
- **HostManager object:** On every participating host, a HostManager provides information about the system environment, creates Vision objects and controls access to them, in order to prevent multiple instantiation or duplication of data.
- **NetManager object:** It creates and maintains the Session object along with a list of all participating hosts, on which it can ensure the existence of a HostManager. Therefore the Session object can transparently access all Vision objects. There is exactly one NetManager per LAN domain.

To encapsulate rendering and lighting classes, there are two base classes for computation servers. The design follows the active object pattern [LS96]:

- **Lighting object:** This is the distributed equivalent of the Vision lighting subsystem. Algorithms can both implement monolithic lighting calculations, or act as a LightOp of a

distributed lighting network. The interface of the Lighting object features a pair of asynchronous request–callback methods. The object is responsible for the concurrent execution of it's own methods. Because there may be multiple Lighting objects, in the case of using a lighting network, the Session object designates the MasterLightOp and the interconnections through it's MultiLighting.

- **Renderer object:** Every Vision object is accompanied by a Renderer object, which is responsible for driving the rendering process. The communication between the Renderers and the lighting system is done via asynchronous request–callback methods.

Figure 3 shows a running distributed system. Note that host 1 does concurrent lighting calculations with a lighting network. Therefore host 1 should have multiple processors to enable functional parallelism.

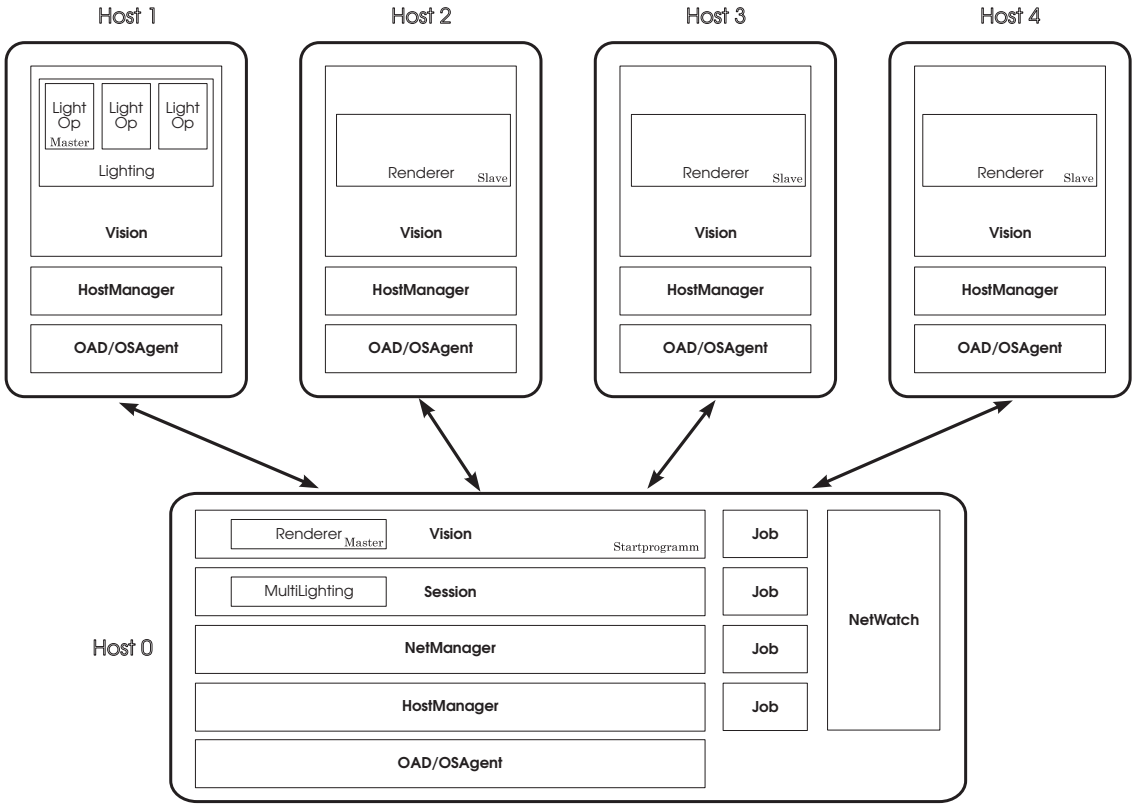


Figure 3: Example of a running distributed system. The `NetWatch` application currently monitors the activities of objects on host 0. Information about objects on other hosts can be accessed by instantiating their Observer interface with a Job object.

The operating system functions are accessed via the portable operating system adaption layer interface of the ACE library [Sch94]. The communication and remote object creation is done using the CORBA implementation VisiBroker of Inprise Corp. [VG98]. To facilitate further development and maintenance, the design of the base classes follows the guidelines of several design patterns [GHJV95] [CS98] [LS96] [SHP97] [McK95].

3.1 Points of asynchronous communication

The class definitions of the distributed system above, allow multiple points of parallelization and distribution. The system is configured by Tcl–Scripts that are parsed by a Tcl–Interpreter in every

Vision object. The integration of the CORBA interface into the base classes of both Renderer and lighting objects, allows all implementations to access remote CORBA objects through the asynchronous request–callback interface. The whole system is started by a small C–program launching the first Vision object and registering it with the CORBA ORB.

The first possibility to work data–parallel, is within the Renderer object. The Session object configures one of them to act as MasterRenderer, who distributes the work asynchronously in a data–parallel way, and collects the results. All the slave Renderers compute intersections with objects of the scene and call the MasterLightOp to perform lighting calculations at the intersection points. The partitioning of the image by the MasterRenderer is implemented by subclasses of the Renderer base class, which allows different scheduling strategies.

Within the setup phase of a LightOp, some implementations need to preprocess the geometry: for example, finite element algorithms subdivide surfaces, calculate visibility or compute energy transfer between surfaces and volumes. This can be quite time consuming. When using a lighting network, the distributed system offers a second possibility of parallelization: it calls all setup and preprocessing methods in parallel on all LightOps.

Third, the communication between the Renderers and the MasterLightOp is asynchronous too, as is the communication between two LightOps. This allows requests to be forwarded very easy at points, where the graph of a lighting network splits, by just reassigning the header of the request packet. Because the Vision framework encapsulates data in "Smart Objects" [Slu96, section 5.3.4], speculative or lazy evaluation can be performed. The asynchronous request–callback communication methods support this feature, by returning the length of the servers request queue. Clients therefore can choose which server to ask, if there is more than one offering the service. Alternatively they can adjust the size of request packets, or switch from speculative to lazy evaluation through using another type of Smart Object for the query.

3.2 Distributed Lighting Networks

The "natural" encapsulation of lighting algorithms and data within a lighting network enables the use of this functional decomposition for distribution purposes. Besides the speedup potential, putting the LightOps on different hosts allows each one to use the whole memory of the system. Basically, there are three ways to make a lighting algorithm known to the distributed system.

1. The lighting subsystem does not try to do functional distribution of the lighting calculations. The constructor of this special lighting object just serializes and forwards all requests in a synchronous manner to the traditional lighting object, that was instantiated by the scene parser, a Tcl–Script or some other default resource. This can be a single traditional LightOp or the MasterLightOp of a traditional lighting network. To speed up computation, the execution is done multi–threaded on multi–processor machines.

   The benefit of this approach is, that developers of traditional LightOps can get a speedup when testing their implementation. Additionally, the creation of preview images with low complexity (local) lighting is accelerated by distributing the intersection calculation of the Renderers.

2. A lighting algorithm is completely re–implemented, in order to take advantage of internal multi–threading, special hardware or communication to third–party CORBA objects. Developers can reuse all traditional Vision classes as well as all facilities of the distributed system extension.

   This is the most powerful possibility of creating a lighting object for the distributed system. It does however require knowledge about its design and behavior.

3. The setup script instructs the Vision object to wrap a traditional LightOp implementation with a special lighting object, which adds the ability to act as a distributed lighting object. This is accomplished by a special implementation of a distributed Smart Object, wrapping traditional Smart Objects to form "the upper half" of the wrapper, through which

the traditional LightOp calls it's predecessor in the lighting network. Therefore, it's totally transparent to the traditional LightOp, whether he's situated in a traditional lighting network, or in a distributed one.

This is the easiest way of building a distributed lighting network, as it requires no programming effort. Furthermore, it is very flexible, in that the whole configuration is done within Tcl–scripts at system startup. Developers of lighting algorithms, that don't offer much chance to be parallelized successfully, need not to know anything about the distributed system in order to use it.

4 Results

This section demonstrates the flexibility of the distributed Vision framework, by discussing some example configurations and the asynchronous communication paradigm. Using the base classes described above, several distributed LightOps have been implemented for specific purposes. In order to reuse the traditional LightOp implementations efficiently, two Multiplexer helper classes are available. They can multiplex request packets onto a pool of identical (distributed) LightOps, making them look like a single LightOp to the rest of the lighting network. There are different scheduling strategies available for the management of the request pool. Figure 4 shows the scene used for the measurements throughout this report.

Figure 4: This scene was used for all the measurements in this report.

4.1 Efficiency of asynchronous communication

Asynchronous request–callback communication combined with multi–threaded clients and servers allows a maximum of parallelism for the LightOps of a distributed lighting network. Table 1 compares a small network, using asynchronous requests, with a equivalent network, using wrapped traditional LightOps. The latter call their predecessor through the synchronous interface for traditional (non–distributed) lighting networks. Both examples use the same host configuration, which is done at Session setup. The last line shows the total execution time, measured at the command prompt.

Table 1: Efficiency of asynchronous communication

wallclock seconds for	**asynchronous LightOps**	**wrapped LightOps**
Session Setup	22.26	23.37
Parsing Scene	5.80	5.67
Lighting Setup	1.56	1.68
Renderer Setup	0.30	0.34
Render Frame	1,922.06	2,916.95
Total	**1,977.36**	**2,974.92**
	66 %	100 %

The main reason for the speedup is the low number of 210 CORBA method calls over the LAN in the case of asynchronous communication, compared to 128,070 synchronous invocations in this example. Both networks transfer 22.7 MBytes of request data through CORBA marshaling. The overhead of the synchronous method invocations however doesn't slow down the communication that much, as the $100\frac{MBit}{s}$ Ethernet has enough resources left[2]. It's the synchronous protocol, that blocks the client until the server has completed the method call. Therefore, the probability for the client to wait, drops with the number of remote calls, but makes the server less responsive. As mentioned in section 3.1, the client implementation can adjust the size of the request packets, to balance this.

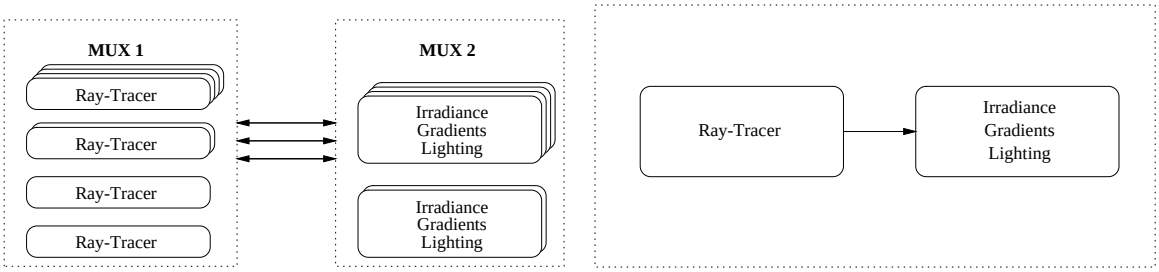
4.2 Optimizing standard rendering

To optimize rendering times in the case of calculating previews, as mentioned in section 3.2, the following example uses 4 hosts with a total of 8 processors:

		SGI Onyx	**SGI Onyx**	**SGI O2**	**SGI O2**
processor	number	4	2	1	1
R10000	MHz	196	195	195	195
Vision object	instances	1	1	1	2
	CORBA client				×
	Renderer	×	×	×	×
	Lighting	×	×		

The lighting hosts execute a traditional implementation of a Irradiance Gradients LightOp. Configuring this system, required just to name the hosts and the Lightop with it's parameters in a standard resource file. The Tcl–scripts for system setup took care of distributing the objects. The Session object uses a Multiplexer to pool the lighting hosts. This distributed system is compared to the traditional Vision system (single thread of control), running on the fastest machine and calculating lighting with the same LightOp implementation.

[2] the example network transfers $\frac{23MByte}{3{,}000s} \approx 64\frac{kBit}{s}$ net data



(a) distributed System

(b) traditional Vision

wallclock seconds for	**distributed System**	**traditional Vision**
Session Setup	31.91	-
Parsing Scene	5.61	-
Lighting Setup	0.14	-
Renderer Setup	0.36	-
Render Frame	317.03	2,359.20
Total	**387.41**	**2,380.15**
	16 %	100 %

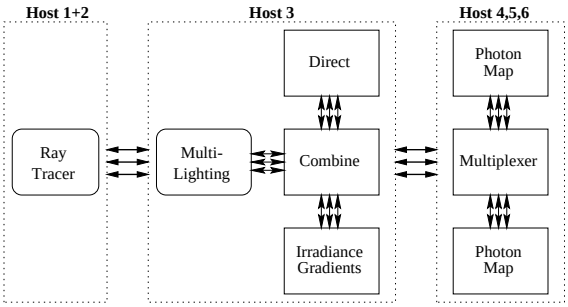
The speedup obtained is quite near the theoretical maximum of 12.5 %. The overhead of ≈ 90 seconds consists of 30 seconds Session setup, 5 seconds of additional parsing on the CORBA startup client and another 5 seconds delay for allowing the hosts to clean up the CORBA objects, before the main CORBA startup client shuts down all Vision instances. This results in a overhead of ≈ 13 % during the rendering phase for the distributed system.

4.3 Optimizing complex lighting

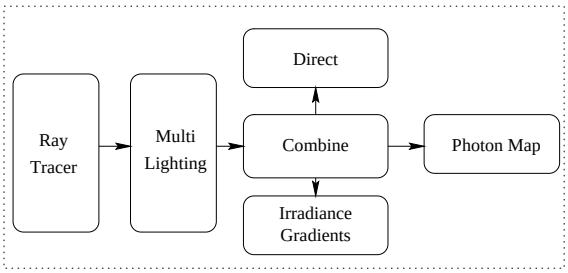
The functional decomposition of a lighting network offers the biggest potential for distribution and parallelization, at the risk of high communication costs. As shown earlier, the asynchronous request–callback communication paradigm is able to provide a solution for that problem. The following example uses 6 hosts with a total of 11 processors.

SGI	Octane	Onyx	Onyx	Octane	2× O2
# processors	2	2	4	1	2
R10000 MHz	250	195	196	175	195
Vision instances	2	1	1	1	2
Renderer					2
Lighting	PhotonMap	PhotonMap	Irrad. Grad., Direct, Combine	PhotonMap	-

In this setup, the reconstruction method of the Photon Map LightOp takes much more time to process a request, than the other LightOps of the lighting network. Consequently, a multiplexer is used to distribute this LightOp onto 3 hosts. In contrast, the three other LightOps are executed on a multi-processor machine, because their reconstruction method is fast and the communication between them can be optimized, if the CORBA implementation supports object collocation. In order to drive this complex lighting subsystem, two hosts execute rendering objects controlled by a multiplexer in a data-parallel way.



(c) distributed System



(d) traditional Vision

wallclock seconds for	**distributed System**	**traditional Vision**
Session Setup	28.05	-
Parsing Scene	7.47	-
Lighting Setup	3.20	-
Renderer Setup	0.28	-
Render Frame	3,026.28	5,096.62
Total	**3,081.86**	**5,186.07**
	59 %	100 %

The speedup obtained by this setup is not as good as one would expect. This mainly due to process idle times if for example the calculation of one upstream LightOp is sufficiently delayed. Since the underlying Lighting Networks is entirely pull-driven, the pipeline is blocked. We try to cope with that problem to some extent by allowing the asynchronous interface to drive three streams at a time.

This example shows that there are cases where the full transparency of the distribution infrastructure cannot hide inherent limitations due to the communication patterns of existing objects. Note however, that this behavior is mostly a problem of the non distribution aware algorithms of the lighting network, and not so much a general drawback of the distribution framework. However, even with the very limited success, we still get some speed-up without any change to the application logic.

Apart from that, one has also to take into account, that while a traditional system performs quite well in this case in terms of execution speed, it is severely limited by the host's memory resources. Especially the PhotonMap LightOp needs to store many photons that have been shot into the scene when working with large scene descriptions. The distributed PhotonMap LightOps in this example have the memory of three hosts to their disposition. Furthermore, the initial shooting of particles is done in parallel, reducing the Lighting setup time needed to one fifth (there are 5 processors on the three hosts), which is of great value when simulating high quality caustics.

Although there certainly is a price to pay for the flexibility of our distribution strategy, we obtain a great flexibility for configuring the distribution strategies and adapt the system to the challenges of a specific lighting network.

5 Discussion

The distribution extension imposes no limitations on the possibilities of the Vision framework. This precludes optimizations of global system state and data transport strategies, based on knowledge

about the implementation of a Lighting object. For example, the sequence of configuration calls made during the setup phase is fixed, and can only proceed step by step on completion by all LightOps. Because a lighting network employs many different LightOp implementations, chances are, that each one needs a long time to process in a different step, without depending on others. Future work is targeted to examine this problem, and provide a automatic on–demand locking of the general program path by the LightOp, that depends on global state information.

The current implementation of the distribution extension uses the Basic Object Adapter (BOA) on the server side. The new Portable Object Adapter (POA) seems to offer substantial improvements. There is ongoing research, to change the distribution extension to use the POA. This will reduce programming complexity and provide improved support for persistent servers, which are able to continue calculations in case of crashes, or reuse lighting information for image sequences, if the scene description hasn't changed. Additionally, switching to TAO [TAO97] promises to speed up communication, and to integrate other UNIX dialects.

The biggest benefit of the distribution extension is the capability to provide support for both distribution and parallelization for programmers of advanced lighting algorithms, while being completely transparent to developers of traditional LightOps. The configuration via Tcl–scripts offers the most flexible and fastest way to experiment with the composition of lighting algorithms, or with testing a new implementation. The multi–level building blocks structure allows each user to view the system at a granularity, convenient for his purposes: someone concerned with geometrical issues or intersection computation, can forget about all what's behind the lighting subsystem interface, and vice–versa. When testing a new algorithm, the distribution system can be configured to wrap the implementation. This makes the distributed Vision framework unique in the field of physically based lighting simulation.

6 Conclusion

This technical report presents a case study of an integrated and application-oriented approach to distribution and parallelization for rendering and lighting computation on the Vision rendering framework. The use of CORBA and the implementation of the distribution functions within base classes, makes distribution issues totally transparent for a specific implementation. Using the lighting networks technique, functional parallelism of the LightOps is possible. Several implementations of distributed LightOps have been created.

The distribution system extension has proved to be stable, with well defined interfaces, without imposing any limitations on the possibilities of the Vision framework. Distributed lighting networks can be constructed and configured by Tcl–scripts. The flexible structure allows the configuration of a distributed system for different purposes, ranging from speeding up previews to experimenting with complex lighting networks.

Object–oriented programming, the use of design patterns and of the ACE library for encapsulating system calls, make the Vision framework and the distribution extension portable to a wide range of platforms. The flexibility of the Vision framework makes it a ideal tool for research and education. The distributed system extension adds efficiency for production environments.

References

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, Redwood City, CA, second edition, 1994.

[CRMT91] S. E. Chen, H. E. Rushmeier, G. Miller, and D. Turner. A progressive multi-pass method for global illumination. *Computer Graphics*, 25(4):165–174, July 1991. SIGGRAPH '91 conference proceedings.

[CS98] Chris Cleeland and Douglas C. Schmidt. External Polymorphism — An Object Structural Pattern for Transparently Extending C++ Concrete Data Types. *C++ Report Magazine*, September 1998. `http://www.cs.wustl.edu/~schmidt/C++-EP.ps.gz`.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addision-Wesley, Reading, MA, first edition, 1995.

[Kaj86] James T. Kajiya. The rendering equation. *Computer Graphics*, 20(4):143–150, August 1986. SIGGRAPH '86 conference proceedings.

[LS96] R. Greg Lavender and Douglas C. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. In James O. Coplien, John Vlissides, and Norm Kerth, editors, *Pattern Languages of Program Design 2.* Addison-Wesley, Reading, MA, 1996. `http://www.cs.wustl.edu/~schmidt/Active-Objects.ps.gz`.

[McK95] Paul E. McKenney. Selecting locking primitives for parallel programs. Technical report, Sequent Computer Systems, Inc., 1995. `http://c2.com/ppr/mutex/mutexpat.html`.

[Peu98] Thomas Peuker. Distribution and Parallelization of Rendering Computations — Verteilung und Parallelisierung von Bildsyntheseberechnungen. Masters thesis (Diplomarbeit), Computer Graphics Group, University of Erlangen-Nürnberg, Germany, 1998.

[Rog98] David F. Rogers. *Procedural Elements for Computer Graphics.* McGraw–Hill, second edition, 1998.

[Sch94] Douglas C. Schmidt. The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software. In *Proceedings of the* 12^{th} *Annual Sun Users Group Conference*, pages 214–225, San Francisco, CA, June 1994. SUG. `http://www.cs.wustl.edu/~schmidt/SUG-94.ps.gz`.

[SHP97] Douglas C. Schmidt, Timothy H. Harrison, and Nat Pryce. Thread-specific storage for C/C++. In *Pattern Languages of Programming '97 conference proceedings*, September 1997. `http://www.cs.wustl.edu/~schmidt/TSS-pattern.ps.gz`.

[Slu96] Philipp Slusallek. *Vision — an Architecture for Physically-Based Rendering.* PhD thesis, Computer Graphics Group, University of Erlangen-Nürnberg, Germany, 1996.

[SP94] F. X. Sillion and C. Puech. *Radiosity & Global Illumination.* Morgan Kaufmann, 1994.

[SSH+98] Philipp Slusallek, Marc Stamminger, Wolfgang Heidrich, Jan-Christian Popp, and Hans-Peter Seidel. Composite lighting simulations with lighting networks. *IEEE Computer Graphics and Applications*, 18(2), March/April 1998.

[TAO97] Computer Science Department, Washington University at St. Louis. *Real-time CORBA with TAO (The ACE ORB)*, 1997. `http://www.cs.wustl.edu/~schmidt/TAO.html`.

[VG98] Inprise, Corp. *Visigenic Products*, 1998. `http://www.inprise.com/visibroker`.