Using Distributed Raytracing in Movie Production

BJÖRN LEFFLER

KTH Numerical Analysis and Computer Science

Master’s Degree Project
Stockholm, Sweden 2005

TRITA-NA-E05018
Using Distributed Raytracing in Movie Production

BJÖRN LEFFLER

TRITA-NA-E05018

Master’s Thesis in Computer Science (20 credits) at the School of Computer Science and Engineering, Royal Institute of Technology year 2005
Supervisor at Nada was Lars Kjelldahl
Examiner was Lars Kjelldahl
Abstract

This master’s thesis studies how distributed raytracing can be used for movie production purposes. To get a complete view of the problem, the author implemented a distributed raytracer, which includes typical features used in the movie industry. The numerical results show that the process scales well, which makes raytracing a more accessible technology.

Although distributed raytracing slightly decreases the total performance of a render-farm, it dramatically shortens the render time of each frame. This provides more control and overview to the end users: Frames that are difficult to render are noticed faster and artists get more direct feedback.

Sammanfattning

Detta examensarbetet studerar hur distribuerad raytracing kan användas för filmproduktion. För att uppnå en helhetsbild av problemet har författaren implementerat en distribuerad raytracer, som även inkluderar typiska funktioner som används i filmindustrin. Resultaten visar att processen skalar väl, vilket gör raytracing till en mer tillgänglig teknik.

Ett renderings-klusters totala prestanda minskar något när denna teknik används. Detta kompenserar av att resultaten anländer fortare, vilket ger användarna en bättre bild av vad som pågår. Bilder med svårigheter lokaliseras snabbare och användarna slipper vänta på resultaten.
Résumé

Ce projet de diplôme étudie comment une moteur de lancer de rayon peut être distribué afin de faciliter l’utilisation dans la production de film. Afin d’obtenir une vue complète du problème, l’auteur a implémenté une moteur de lancer de rayon distribué, qui inclue des fonctionnalités couramment utilisées dans l’industrie de film. Les résultats numériques montre que cette procédure évolue bien et rend la technologie plus accessible.

Bien que cette procedure diminue légèrement les performances d’un parc d’ordinateur, le temps de rendu de chaque image est fortement diminué, ce qui donne plus de contrôle aux utilisateurs. Les images avec difficultés sont plus vite repérées et les utilisateurs reçoivent les résultats plus vite.
Preface

This master’s thesis was written by Björn Leffler at NADA, KTH, Sweden under the guidance of Lars Kjelldahl. The experiments were carried out at the Swiss Federal Institute of Technology EPFL, Lausanne, Switzerland.

I would like to thank all the people who have helped me to complete this work and this report. In particular Lars Kjelldahl for valuable advice and for giving me so much freedom during this research. Laurent Desimone for providing access to computer resources. Rasmus Kaj and the other members of Stacken Computer Club, who gave me programming advice and finally my aunt Elisabeth Forssell for proofreading.
# Contents

1 **Introduction** ............................................. 1
   1.1 Background ........................................... 1
   1.2 Goal ..................................................... 1
   1.3 Objective ............................................... 1
   1.4 Method ............................................... 2
   1.5 Scope .................................................... 2
   1.6 Approach ............................................... 2
   1.7 Summary of results .................................... 3

2 **Raytracing** .................................................. 4
   2.1 Introduction .......................................... 4
   2.2 Basic algorithm ....................................... 4
   2.3 Global illumination .................................. 5
   2.4 Example ................................................. 5

3 **The RenderMan Interface** .................................. 7
   3.1 What is RenderMan? ................................... 7
   3.2 Example ................................................. 7
   3.3 RenderMan features .................................... 8
      3.3.1 Advanced concepts ............................... 8
      3.3.2 Outstanding image quality ....................... 9
      3.3.3 Flexible and extensible ......................... 9
      3.3.4 Performance ....................................... 9
      3.3.5 Free choice of rendering algorithm ............ 10
      3.3.6 User programmable ............................... 10
   3.4 The Shading Language ................................ 10
      3.4.1 How shaders work ................................ 10
      3.4.2 Surface Shaders .................................. 11
      3.4.3 Displacement shaders ............................ 11
      3.4.4 Light shaders .................................... 11
      3.4.5 Volume shaders .................................. 11
      3.4.6 Imager shaders .................................. 11
   3.5 Combining raytracing with shaders .................... 12
   3.6 Shader example ........................................ 12
Chapter 1

Introduction

This chapter describes the goal of this master’s thesis, the problems I try to solve and the method for solving them.

1.1 Background

Movie production is an expensive process in terms of computer resources (processing power, memory, disk usage and so on). Each frame is generated in high resolution with as many features and effects as technology will allow.

Even though raytracing has existed for a long time, it is rarely used in movie production. The reason is that the technology is simply too expensive in terms of computation power. Another reason is because light effects can often be approximated using other technologies, such as environment mapping to simulate reflections.

As today’s computers become more powerful, raytracing is becoming more accessible. Still, the necessary render times are long. One way of shortening this time is to distribute the raytracing across several processors.

1.2 Goal

The goal of this master’s thesis is to investigate how the performance changes when more computers are added to a distributed raytracing system. Should performance increase less than expected, a secondary goal is to locate bottlenecks in order to find limitations or suggest possible optimizations.

1.3 Objective

The above questions are interesting, because if the distribution process scales well, then adding more processors is a simple and universal solution to shorten render time. For some applications, this might even open the possibility of ‘realtime’
raytracing. For other applications, such as movie production, it would simply make raytracing a more accessible technology.

1.4 Method

My method of investigating the scalability of such a system is to implement one, study the encountered problems and see how close to the ideal numerical results I can get. This work should give me a clear view, better understanding and allow me to analyse the problem correctly.

I could also have chosen to improve some existing raytracing software, which probably would have been faster in terms of development time. On the other hand, this might have imposed certain restrictions on my work.

Creating a new raytracer allowed me to approach the problem in an ideal way, and to design the software for distributed use from the beginning.

1.5 Scope

In order to reach this goal, I have implemented a distributed raytracer called RenderBjorn, which includes typical and time-consuming features found in commercial renderers, such as anti-aliasing and programmable shaders. The primary purpose of RenderBjorn is not to obtain production quality output, but to illustrate the kind of scalability issues one could expect to find in movie production systems.

The reason for this limitation is that only a reasonable part of the total time available should be used for programming. RenderBjorn was designed to run on university workstations and studies both shared and distributed memory parallelism. Depending on the problems that arise, the main purpose is to locate them and their causes, not necessarily to solve them.

1.6 Approach

During the time available for this project, I am expected to use a certain procedure. This is the approach that I have chosen:

1. Study of relevant literature
2. Planning of the implementation
3. Implementation of a raytracer, prepared for distributed use
4. Implementation of the distributed part of the raytracer
5. Tests and optimisations
6. Writing this report and other administrative tasks
1.7 Summary of results

- In the ideal case, performance scales linearly with the number of processors available. This was the case in a shared memory environment.

- Performance scales almost as well in a distributed environment, at least up to a certain number of processors. The downside of using a distributed environment is that the total memory requirement of a renderfarm greatly increases. On the positive side, it may still be cheaper to purchase and operate many smaller computers rather than a few large ones.

- Distributing raytracing on an existing renderfarm does not increase the total memory requirement as compared to raytracing one frame per computer. It simply uses more memory for a shorter amount of time.

- The total performance of a renderfarm slightly decreases when using distributed raytracing instead of raytracing one scene per computer. This is greatly compensated by each frame being rendered faster, giving more control and a better overview to the users.
Chapter 2

Raytracing

This chapter introduces raytracing, a technology used in computer graphics to simulate light in different ways.

2.1 Introduction

One of the most common methods of explaining the process of image synthesis is to appeal to our intuition about the physical process of photography: Light rays are emitted from light sources, bounce around the scene and eventually hit the camera. This process of following light rays is called raytracing. In practice, rays are not traced from light sources, hoping they eventually hit the camera. Instead, rays are traced backwards from the camera until they hit a light source.

Because raytracing simulates the propagation of light, it is capable of creating realistic images with many interesting effects, such as mirror reflections, refraction through glass or volumetric effects when light rays pass through materials.

2.2 Basic algorithm

The following pseudocode illustrates a basic recursive raytracer

1:  For each pixel of the image:
2:      trace a ray from the virtual camera position
       through the position of the pixel in the image plane
3:      find the closest intersecting surface
4:      find the surface color at the intersecting position
5:      if necessary, trace another reflected ray (go to 3)
6:      if necessary, trace another refracted ray (go to 3)

At first sight, this may not look difficult, but it is computationally very expensive. Step 3 represents the fundamental operation of raytracing, which is to find the intersection of a light ray with a geometric primitive. Depending on the type of
primitive (sphere, polygon, NURBS and so on), this may require anything from a simple mathematical expression to a complex geometric approximation.

The cost of finding the closest intersecting primitive also increases with the total number of geometric primitives and this operation may also be repeated several times per pixel, depending on how the light bounces. High-resolution images with millions of primitives therefore require a lot of processing power for this operation alone.

For these reasons, a lot of effort is concentrated on optimizing this operation, for example by using bounding boxes and search trees.

2.3 Global illumination

Raytracing is also a global illumination algorithm. All objects may interact and cast shadows on all other objects. Global illumination algorithms are rarely used in movie production, because they require more processing power and large amounts of memory. From a user perspective, global illumination is very practical, as shadows and light interaction are automatically created.

2.4 Example

Figure 2.1 shows a typical example of recursive raytracing. N represents the surface normals at ray-object intersections.

- Ray 1 is traced from the camera position, through the image plane, and intersects object A.
- Ray 2 is traced towards light C. Light D is hidden.
- Ray 3 is traced as the reflection of ray 1 on object A.
- Rays 4 and 5 are traced towards lights C and D.
- Ray 6 is traced as the refraction of ray 1 on object A.
- Ray 7 is traced towards light D. Light C is hidden.
Figure 2.1. Example of recursive raytracing.
Chapter 3

The RenderMan Interface

This chapter introduces *The RenderMan Interface* [4], an open specification for scene description. This standard was chosen because it is widely used in movie production.

3.1 What is RenderMan?

Pixar Animation Studios developed and published the RenderMan Interface specification in 1988. The goal of the RenderMan Interface is to provide a standard mechanism for modeling and animation software to send data to rendering systems in a device-independent way. The RenderMan Interface is similar to Postscript, but in 3D.

RenderMan is sometimes used to refer to *Photorealistic RenderMan* or *PRMan* — Pixar’s own implementation of the RenderMan Interface. For the rest of this paper, RenderMan will refer to The RenderMan Interface.

3.2 Example

The RenderMan language is easy to understand to anyone with a general knowledge of computer graphics. Figure 3.1 shows the output of listing 3.1.
# Configure the "camera"
Translate 0.0 0.0 2.0
Projection "orthographic"

# Add an ambient and a directional light source
LightSource "ambientlight" 1 "intensity" 0.5
LightSource "distantlight" 2 "from" [1 1 1]

# define the scene with a red sphere with radius 1
WorldBegin
Color 1 0 0
Sphere 1 -1 1 360
WorldEnd

Listing 3.1

Figure 3.1. Example of RenderMan output

3.3 RenderMan features

3.3.1 Advanced concepts

Production renderers must make a wide variety of tricks available to the users. Effects like shadows, texture mapping of all varieties, atmospheric and volumetric effects, depth-of-field, lens flare and control information for post-rendering manipulation of images must be provided for.
RenderMan is intended as a high-level description of what to render instead of describing the details of how to render. This includes high-level primitives, such as cubic patches and NURBS, as well as abstract descriptions of attributes, such as the shading language. To this day, RenderMan is still the only open specification for scene description that includes advanced concepts like motion blur and user-definable characteristics.

### 3.3.2 Outstanding image quality

Production studios use the highest possible resolution to obtain maximum image quality. Since the final images are displayed on huge screens, even small artifacts will leave a bad impression. Artifacts of any and all varieties are to be eliminated. High-quality anti-aliasing in all dimensions, texture and shader filtering, pixel filtering and accurate geometric approximations are all required features. Anti-aliasing of shading, lighting and geometry must work consistently at all scales and all distances from the camera.

### 3.3.3 Flexible and extensible

Perhaps an even more important requirement is flexibility and controllability. The studio requires the renderer to be user-programmable so that it can be made to do everything that no one ever knew would be needed.

Instead of creating an API with a fixed set of graphics primitives and visual attributes, RenderMan was designed with extensible mechanisms built in. Similarly to other graphics APIs, such as OpenGL, RenderMan contains commands to draw graphic primitives with certain visual attributes. But unlike most of them, RenderMan was not specifically designed for current-generation hardware graphics accelerators. Instead, it was designed to be extensible, to fit future generations of hardware.

### 3.3.4 Performance

RenderMan has a powerful set of speed/quality and speed/memory trade-offs, which are intuitive to use. This is necessary in a production environment, where a large number of frames must be rendered with a limited number of resources and time. Memory usage must also be reasonable for this reason.

Unfortunately, developers cannot depend on Moore’s law (hardware performance doubles every 18 months). As computers get faster and cheaper, the demand from the users usually increase accordingly. In some cases, even faster than Moore’s law. Therefore, rendering times tend to take the same amount of time, regardless of hardware.
3.3.5 Free choice of rendering algorithm

Instead of focusing on the algorithm used to render a scene, RenderMan focuses on what is to be rendered. The choice of algorithm is left to the implementation, which may use any combination of scanline methods (Z-buffer, REYES), raytracing, radiosity, or other methods. This also allows RenderMan to evolve with future algorithms as they become available. The output quality however, is always expected to be outstanding.

3.3.6 User programmable

The RenderMan Shading Language is a programming language for extending the predefined functionalities of the RenderMan. New materials and light sources can be created using this language, which is also used to specify volumetric attenuation, displacements, and simple image processing functions. All required shading functionalities are expressed in this language. A Shading language is an essential part of a high-quality rendering program. No single material lighting equation can ever hope to model the complexity of all possible materials.

Every production renderer nowadays contains a shading language, which can be used to redefine the shading model in almost arbitrary ways. It is hard to overstate how critical this one feature of a renderer is to a production environment. Users will not accept the standard phong shading model on all objects. The users must be able to change surface properties, reactivity to light, and even the physics of light and optics itself to generate the compelling photosurrealistic image of each shot.

3.4 The Shading Language

3.4.1 How shaders work

A shader is a procedure or function, which answers the question “What’s going on at this spot?”. This is known as an implicit model, as compared to an explicit model, which would be more like “draw this at that position”.

The Shading Language is one of the most important features that RenderMan has to offer and it completely controls the calculation of the color of objects in a scene.

The RenderMan Shading Language is a C-like language with access to information, such as geometric data, surface parameters and incoming light. The shaders can use a variety of useful geometric, mathematical and noise functions. In addition, custom functions can be added to simplify or provide new functionalities of any kind.

There are 5 different types of shaders in the RenderMan Shading Language:
3.4.2 Surface Shaders

Surface shaders are attached to geometric primitives and describe how the surface reacts to the light that shines upon them. This is how the optical properties of materials are modeled. The shader computes the light reflected in a particular direction by summing up the incoming light and considering surface properties. The shader may also calculate any function, use textures, add noise, and so on.

3.4.3 Displacement shaders

Displacement shaders add perturbations to a surface, such as wrinkles or bumps. These small-scale surface changes are insignificant to the geometric model and change only the surface position and surface normal. Compared to ordinary bump mapping, where only surface normals are modified, displacements shaders actually deform the geometric model. This also affects the visible borders of the primitive and the shadows falling from the object.

3.4.4 Light shaders

Light shaders calculates the color and intensity of the light emitted from a light source towards a point being illuminated. A light will typically have a direction, a distribution and a fall-off with distance.

3.4.5 Volume shaders

Volume shaders control how light is affected as it passes through a participating medium, such as smoke or transparent materials.

Volume shaders are used in three different ways:

1. Atmospheric shaders
   Control the atmospheric contribution, such as fog. This type of volume shader is applied between the camera and the first geometric primitive.

2. External shaders
   Control how light changes when traveling from one geometric primitive to another.

3. Internal shaders
   Control how light changes when traveling through a transparent material such as colored glass. Volumes are defined as the inside of solid objects.

3.4.6 Imager shaders

Image shaders are used to program simple pixel operations that are done before the image is quantized and output.
3.5 Combining raytracing with shaders

Whenever a ray travels through space, its color and intensity is modulated by the volume shader attached to that region of space. If that region is inside a solid object, the volume shader is the one associated with the interior of that solid. Otherwise, the exterior shader of the spawning primitive is used. Whenever an incident ray intersects a surface, the surface shader attached to that graphic primitive is invoked to control the spawning of new rays and to determine the color and intensity of the outgoing rays and the material properties of the surface. Finally, whenever a ray is cast to a light source, the light source shader associated with that light source is evaluated to determine the color and intensity of the light emitted.

3.6 Shader example

If shaders were used in figure 2.1, they would be evaluated as follows:

- Before other shaders, displacement shaders are evaluated.
- At each ray-object intersection, the corresponding surface shader is evaluated.
- The atmosphere shader affects ray 1.
- The light shader of light C affects rays 2 and 4.
- The light shader of light D affects rays 5 and 7.
- The exterior volume shader of object A affects ray 3.
- The interior volume shader of object A affects ray 6.
Chapter 4

Distributed raytracing

This chapter describes the algorithms used to distribute the raytracing over several processors.

4.1 General

Processors may share the same memory space of a computer, or be separated in different computers, each having its own memory space. Only these two cases are studied, but a combination of both (several multiprocessor computers) is also possible.

4.2 Shared memory architecture

The easiest way to parallelize raytracing is to use a computer with several processors and a single shared memory. All processors then have access to the same information. Information stored in memory by one processor is immediately accessible to all other processors. Several threads are used for the raytracing part. These threads are hopefully scheduled by the operating system, so that they execute simultaneously on different processors. Building shared memory computers is a difficult task. There are limitations to bus performance, size, cooling, etc...

4.3 Distributed memory architecture

When distributing the raytracing on several independent computers, each computer has its own system resources, including memory. All information must therefore be sent over the network, using a communication protocol. This external communication is a lot slower than the internal communication within a computer. The scene description must be available to all processors and in all memory spaces. This greatly increases the total memory requirement.
4.4 Roles

RenderBjorn can be used in three different ways:

1. **Standalone**
   - Local

2. One client and one rendering server.

   ![Diagram](image)

3. One client, one coordinator and many workers.

   ![Diagram](image)

To clarify how the communication works and to specify which computer does what in the distributed mode, I have defined a number of roles. Of course, nothing prevents one computer from having several roles.

- **Local**
  A standalone version: all work is done on the same machine without network communication. Multiple threads can be used to take advantage of a multiprocessor computer.

- **Client**
  A client connects to a server (a Master or a Coordinator), which then executes the raytracing. From the Client’s point of view, the server is a “black box”: it doesn’t matter whether it’s a Master or Coordinator. A Client never performs any raytracing and has no notion of RenderMan. It simply forwards the information and waits for the result. Then the server tells the client all it needs to know. The purpose of this is to create “lightweight” clients with a graphical user interface.

- **Master**
  A Master is a network server that receives information about the scene to render, performs the raytracing and sends back the result over the network.

- **Worker**
  A Worker does the same thing as a Master, except it only renders parts of the total scene. A Coordinator tells the Worker which parts.
• **Coordinator**  
  A Coordinator coordinates the workers. Initially, it forwards the scene information to the Workers. Then the Coordinator tells each Worker which pixel areas it should render. The result is forwarded to the Client. A Coordinator only manages the workers and does not perform any raytracing itself.

### 4.5 General algorithms

Two well-known algorithms from parallel programming are used in the following ways:

- **Producer and Consumer**  
  The *consumer* (a Client) tells the *producer* (a Master or a Coordinator) which task to perform by sending the RenderMan input. The producer then sends the result back to the consumer.

- **Bag of Tasks**  
  The main job is split into smaller *tasks* by the Coordinator. These tasks are performed one by one by the Workers. The Coordinator tells each Worker which task to carry out, until all tasks are done.

### 4.6 Specific algorithms

The following section outlines the specific algorithms explaining how the different roles cooperate. The distributed parts of RenderBjorn are *not* specific to raytracing and can be used with any rendering algorithm. In addition, the distribution protocol is *not* RenderMan specific and can easily transfer other scene descriptions.

#### 4.6.1 General

In general, the rendering is done by splitting the final image into rectangular *pixel areas*, then rendering each area until the final image is complete. Splitting the frame into smaller areas facilitates the distribution of work and the intermediate results give a better overview to the user. Although the following algorithms only describe how a single frame is rendered, the algorithms are easily generalized to several consecutive frames.

#### 4.6.2 Local role

- Input render information, usually from files.
- Split image into pixel areas.
- Render each pixel area.
- When all pixel areas are rendered: save final image to file and/or display image on screen.
4.6.3 Client role

- Send render information to server (a Master or a Coordinator).
- Receive every pixel area of the frame from the server.
- Put all parts together
- Save the resulting frame to a file and/or display on the screen.

4.6.4 Master role

- Receive render information from the Client.
- Split image into pixel areas.
- Render each pixel area and send it back to the Client.

4.6.5 Worker role

- Receive render information from the Coordinator.
- Receive request from Coordinator
- Render the requested pixel area and send it back to the coordinator.

4.6.6 Coordinator role

- Receive render information from the Client.
- Split image into pixel areas.
- Order Workers to render all pixel areas
- Forward result to the Client
Chapter 5

Implementation

This chapter explains the procedure for creating the distributed raytracer. Development was done in iterative steps, each time making sure every step worked fine before moving to the next.

5.1 Basic raytracer

To begin, a basic raytracer was implemented. This was straight-forward as raytracing is a well-documented process. Common algorithms, such as ray-primitive intersection algorithms are discussed in [1] and [3]. A specific vector and matrix library was developed, which includes useful functions, such as reflections and rotations.

The largest problem at this stage was to know if the vector operations and camera projections were correct. Images were often upside-down or reversed. This problem was overcome by creating the same scene in OpenGL for reference. At this stage, input was hardcoded into the raytracer. Later, when the RIB parser was implemented, the result was compared to other RenderMan compatible renderers. Being able to compare results in this way is one of the major advantages of using the RenderMan interface for input.

Though not RenderMan specific, the geometric primitives, lights, cameras, etc. could all be reused later. For fun, the famous Utah teapot was also included. The fish-eye camera was actually created by mistake when I tried to implement the projection camera.

5.2 RenderMan features

After creating the basic raytracer, the possibility of attaching surface shaders to geometric primitives and light shaders to light sources was added. The next major task was to implement the RenderMan shading language, which proved to be quite a challenge.
5.2.1 Standard shaders

The RenderMan Specification contains a number of standard shaders, which are present in all RenderMan implementations. Implementing these shaders was a good preparation for the next part.

5.2.2 Programmable shaders

Programmable shaders are written in the RenderMan Shading Language. Once compiled, they can be dynamically loaded when needed.

Implementing these shaders was a difficult task. Ordinary renderers use a custom compiler to create byte-code, which is executed in an interpreter at run-time. This was not an option, as it would have taken too much time to implement. Instead, I opted for another solution. By creating a parser, the shading language could be transformed into C++. This C++ shader is then compiled into a shared library file, which can be loaded into RenderBjorn.

A secondary problem was that only C, but not C++ can be loaded as a dynamic library. A wellknown workaround was used to solve this problem.

5.3 Distributed framework

The distributed version of RenderBjorn was created using the roles presented in section 4.4.

5.3.1 First phase: client and master

The initial raytracer was modified to act as a client or a master. The client sends the input (RIB files) to the master, which renders the image in small segments (called pixel areas) and sends them back to the client. This phase took a lot more time than the following because the network protocol had to be defined and implemented. Only smaller changes to the initial raytracer were required to add networking.

5.3.2 Second phase: coordinator and worker

Implementing the distributed server was easy and fast, with only minor changes to the network protocol. The coordinator acts as a master to the client and as a client to the workers. The only difference is that the workers receive requests of which pixel areas to render from the coordinator. From the client side, a master and a coordinator behave identically. The client part was therefore left unchanged.
5.4 Network protocol

RenderBjorn has its own network protocol, which uses TCP/IP. Being a connection-oriented protocol, TCP delivers messages in a FIFO (first in, first out) way, needed to avoid data corruption. Messages are never lost in transit as TCP guarantees delivery and resends the message if necessary. Actually, two TCP connections are used, one to send control messages and one for binary data. This is similar to the FTP (file transfer protocol) and facilitates message parsing. In case of network congestion, this may also favor the control connection as it carries less data. The connections are opened in the direction of the server and will work even if the client side is placed behind a firewall.

When a pixel area is sent from the server to the client side, its size and position are announced over the control connection. The binary data is then sent over the data connection. Sometimes the binary data arrives before the announcement. The data may also be split into several network packets. For these reasons, a buffer system is needed to receive and keep track of all binary data.

The communication is event-based and mostly asynchronous: The receiver does not explicitly confirm received messages. The sender simply sends the messages it considers necessary and these messages may be received at any time. This is an important step to reduce idle waiting. A coordinator, for example, can send orders to workers in advance.

The communication is failsafe, in the sense that a malfunctioning worker will be removed and its work rescheduled to another worker. New workers may also be added as the they become available. This flexibility is a major advantage as compared to MPI communication, where the number of peers are constant and fault-tolerance is more difficult to achieve. Using MPI also requires using a special compiler. This was a second restriction I wanted to avoid.

5.5 System architecture

RenderBjorn was created using standard C++ and the GNU g++ compiler. The network protocol was designed using TCP/IP with a C++ socket library [8]. POSIX threads and locks were used to control parallel processing. The software libraries LibTIFF[5], LibJPEG[6] and LibPNG[7] were included to handle image formats.

This platform-independent combination of software can be compiled on any platform and has successfully been built on Mac OS X, Linux and Solaris.
5.6 Problems encountered

Implementing the programmable shaders was a difficult task. The mechanism described in section 5.2.2 must be implemented using the only available method, which is to create a stand-alone C library.

5.6.1 Compiling programmable shaders

This part was straight-forward, but as there is no exact grammar for the Shading Language, the parser had to be changed numerous times, as new language constructs were discovered by testing different shaders. The parser creates a C++ class for each shader.

5.6.2 Loading programmable shaders

The C++ compiler can only produce dynamically loaded C libraries. The well-known workaround consists in declaring C functions, which can create an instance of the C++ shader class and return a pointer to that instance. It must then be released using another library function.

5.6.3 Dependencies of programmable shaders

All functions and data types used by the shader must be included in the dynamically loaded file. This was not a problem for the standard shaders, since everything is included in the executable file anyway, but to the dynamically loaded files, this meant adding half the code of RenderBjorn. The solution to this problem was to create a special shader environment, which contained only the minimum requirements for the Shading Language. This solution worked, but felt like a lot of trouble only to satisfy the program linker.

5.6.4 Size of compiled programmable shaders

The shaders now actually worked, but the compiled shaders files were too large. Over 1 MB for the simplest shader due to all the extra information included. This problem remains unsolved. One solution might be to place the shader specific environment in another external library and link all compiled shaders to that library.

5.6.5 Distributed Performance

At first, the distributed mode took at least twice as long as the single-processor version. The solution was to rewrite the coordinator algorithm. Instead of doing everything in one thread, a specific thread was assigned for client communication and another thread was assigned for worker communication. The workers also received one extra pixel area to avoid idle waiting.
Chapter 6

Numerical results

This chapter describes the numerical results obtained and compares them to the ideal results, which could be expected from a distributed raytracer.

I have derived the following equations myself. To my surprise, the same equations were later presented during the summer course in high performance computing, given by PDC, the Center for Parallel Computers at KTH.

6.1 The graphics pipeline

Before analyzing the numerical results, it is important to understand how the graphical pipeline works. I have chosen to split it into 4 different steps:

- **Load**: the initial loading of the necessary elements.
  (example: read scene descriptions, load textures, shaders and so on)
- **Preprocessing**: Any processing before raytracing can start.
  (example: creating optimisation trees, displacement shaders, ...)
- **Raytracing**: raytracing, including shading and other effects.
- **Postprocessing**: Any processing after raytracing is complete.
  (example: imager shaders)

These steps are executed sequentially. On a single-processor computer, the total time is $A + B + C + D$. Since raytracing is the most time-consuming part, it is the most interesting to distribute and run in parallel.
6.2 Lower bound

In RenderBjorn, the load, pre- and postprocessing are neither executed in parallel, nor distributed. So even if raytracing could be done in no time, the minimum time would be \( A + B + C \).

In addition, if the conditions are optimal when using only one processor, using \( m \) processors is at most \( m \) times faster.

6.3 Ideal results

Let’s assume that:
- Raytracing requires 100 times more time than any of the other steps. (\( A = B = D = 1 \) and \( C = 100 \))
- The raytracing part scales linearly with the number of processors.
- Processing power is the bottleneck in the system.

In order to compare results, we introduce Speed-up, the performance gain compared to \( T_0 \), the execution time on a single-processor computer. In this case,

\[
\text{Execution time} = \frac{A + B + C + D}{A + B + C + D} \times T_0 = \frac{3 + \frac{100}{m}}{103} \times T_0
\]

\[
\text{Speed up} = \frac{T_0}{\text{Execution time}} = \frac{103}{3 + \frac{100}{m}}\]

Figures 6.1 and 6.2 show the expected execution time and speed-up of the above example.

Notice how the actual speed-up diverges from the curve of linear speed-up. Distributing a single frame over 10 processors is actually 23\% less efficient in terms of total performance, compared to rendering 1 frame per processor. Using 20 processors is 35 \% less efficient. The maximum speedup in this example is

\[
\lim_{m \to \infty} \frac{103}{3 + \frac{100}{m}} = \frac{103}{3} = 34.33
\]

This shows that there is a trade-off between render time and total performance. This choice will depend upon the characteristics of the scene (A,B,C and D) as well as the expectations of the users. Also note that using more processors than pixel areas cannot increase performance, as the exceeding processors are idle.

6.4 Test image

Figure 6.3 shows the image used for testing. Although the test image is very simple, it has the advantage of being uniformly complicated: all pixel areas take a similar
Figure 6.1. Expected execution time with $T_0 = 100$.

Figure 6.2. Expected speed-up.
amount of time to render, which facilitates benchmarking. Using short render times also saved me time, when making many repeated measures. In all tests, pixel areas of 40x40 pixels were used and the standard “constant” shader was applied on all surfaces.

6.5 Numerical results

Each of the following tests was executed several times and the numerical value is the lowest value obtained.

6.5.1 Shared memory results

Figure 6.4 shows the numerical results on a 4-processor Sun computer when rendering the test image at a resolution of 800x800 pixels. The numerical results are close to the ideal results, showing that the distribution scales well. These results were consistent and stable. The computer has four 450 MHz Sparc processors and 1GB RAM per processor.
6.5.2 Distributed memory results

The times in the distributed memory case vary strongly, compared to the shared memory case. Especially when many processors are used. This is due to slower communication between nodes, synchronization problems or performance issues on individual nodes. Since all raytracing must finish before continuing to the postrendering step, a slow node may delay the final result. The experiments were carried out using identical computers from Sun Microsystems running Solaris. These have a 500Mhz SparcII processor with 256 MB RAM and are interconnected by 100 Mbps Ethernet.

Figures 6.5 and 6.6 present the numerical results at a resolution of 800x800 and 1600x1600. The lower curve represents the lower bound as defined in section 6.2. These graphs show that the distribution scales well, at least up to a certain number of processors. Beyond this point, the coordinator becomes a bottleneck, which I believe is due to my implementation, and not the general case.

Also notice how this number of processors increase with the size of the frame. Figure 6.6 requires 4 times more processing power than figure 6.5.
Figure 6.5. Render time in a distributed memory environment at 800x800.

Figure 6.6. Numerical results in a distributed memory environment at 1600x1600.
Figure 6.7. Efficiency in a distributed memory environment at 800x800.

6.6 Efficiency

In the next chapter, we will approximate the efficiency when distributing raytracing. In the shared memory situation, we will assume that distribution scales linearly at 100% efficiency. In the distributed case, we will assume that it is 90% efficient based on figures 6.7 and 6.8. Efficiency is defined as

\[ Efficiency = \frac{Speed\ up}{number\ of\ processors} \]
Figure 6.8. Efficiency results in a distributed memory environment at 1600x1600.
Chapter 7

Discussion

The following chapter is intended as a general study of the total time needed to render a large number of frames on a renderfarm. The simple strategy of rendering one frame per computer only is compared to the distributed strategy of rendering each frame on several computers. The discussion is based on the numerical results of RenderBjorn and not necessarily specific to raytracing.

7.1 Assumptions

This chapter assumes that:
- A process is a UNIX or POSIX process which is multithreaded to take advantage of multi-processors computers.
- Process scheduling is fair and does not favor any process.
- A single computer may run several rendering processes. This is sometimes used to take advantage of shared resources. A dual-processor computer could for example run two rendering processes.
- Processing power is the only bottleneck in the system.
- Rendering scales linearly in shared memory situation.
- Rendering scales at 90% of linear performance in a distributed memory situation.

For the sake of simplicity:
- All computers have identical hardware configurations.
- All computers run the same number of rendering processes.
- Computers do not run out of physical memory (RAM) as this dramatically decreases performances. This translates to two conditions:
  - A single process never uses more memory than the available RAM.
  - If several processes operate on the same computer, their total memory usage never exceeds the total RAM available.
7.2 Nomenclature

\( F \) number of frames to render
\( T \) total render time
\( T_1 \) total render time (simple strategy)
\( T_2 \) total render time (distributed strategy)
\( n \) number of computers
\( m \) number of processors per computer
\( s \) number of simultaneously rendered frames
\( p \) number of rendering processes per computer
\( d \) numbers of computers on which a single frame is rendered
\( t \) average render time per frame
\( t_a \) average render time per frame using one processor
\( t_1 \) average render time per frame (simple strategy)
\( t_2 \) average render time per frame (distributed strategy)

7.3 General

The total render time increases with the number of frames and average render time, but decreases with the number of simultaneously rendered frames.

\[ T = \frac{Ft_a}{s} \]

7.4 Simple strategy

When each of \( p \) processes render one frames on one computer, the number of simultaneously rendered frames are

\[ s = np \]

We have assumed the scheduling is fair and that performance scales linearly in a shared memory situation. Therefore, the average render time is

\[ t_1 = \frac{t_a p}{m} \]

The total render time is therefore

\[ T_1 = \frac{Ft_1}{s} = \frac{F}{np} \frac{t_a p}{m} = \frac{F t_a}{nm} \]

Note that \( p \) (the number of rendering processes per computer) has no necessary influence on the final render time, probably because of the assumptions of section 7.1 being restrictive.
7.5 Distributed strategy

Compared to the simple strategy, the number of simultaneously rendered frames is lower:

\[ s = \frac{n \cdot p}{d} \]

But the render time of each frame is also shorter:

\[ t_2 = \frac{t_{ap}}{dm} \cdot \frac{1}{0.9} \]

Once again, \( p \) has no influence on the total render time, which equals

\[ T_2 = \frac{Fd}{np} \cdot \frac{t_{ap}}{dm} \cdot \frac{1}{0.9} = 1.11 \cdot \frac{Ft_a}{nm} \]

Compared to the simple strategy, the total render time of the simple strategy is slightly shorter than the total render time of the distributed strategy.

\[ T_1 = \frac{Ft_a}{nm} < 1.11 \cdot \frac{Ft_a}{nm} = T_2 \]

7.6 Comments

- Distributing the rendering process does not decrease the total render time, but it greatly reduces the render time of each frame. In a production environment, this gives the users more control and overview of the rendering process. Users can view the results faster and rendering problems can be noticed faster. Knowing the daily rate of rendered frames might facilitate planning of the production schedule.

- The problem of effectively distributing rendering is probably more complicated than we have assumed. We also assumed that no memory problems will arise, which is not necessarily the case. There is also a communication cost, which increases with the total number of messages sent between nodes.

- One could think that distributed raytracing uses more memory than rendering one frame per node. This is not necessarily true as more memory is simply being used during a shorter amount of time. The total memory required is approximately the same.
Chapter 8

Conclusion

Distributed raytracing is an effective tool for movie production, where adding more processors becomes a universal solution to reduce render times and possibly production schedules.

In terms of general performance, the process of distributing raytracing on many processors scales well, both in shared memory and distributed memory environments. Although my numerical results show that the process only scales well up to a certain level, this is probably not the general case. First of all, the test scenes were very simple. Secondly, the RenderBjorn implementation needs more optimization. Either way, the numerical results show that the scalability increases with the complexity of the scene.

In terms of render times, distributing raytracing is very efficient. Scenes that are difficult to render are quickly noticed and any scene can be previewed and rendered a lot faster. Choosing the number of processors on which to distribute raytracing is a tradeoff between total renderfarm performance and render time of individual frames.

In terms of memory usage, distributing raytracing over a number of computers does not increase total memory requirement, as compared to rendering one frame per computer. It simply uses more memory during a shorter amount of time. Unfortunately, the entire scene description must be available to every processor. This is not a problem in a shared memory environment, but in a distributed environment, it greatly increases the total quantity of memory.

The choice between a shared and a distributed memory situation (or a combination of both) will depend upon the total cost of processing power versus the total cost of memory.
Chapter 9

References

Throughout the project, I have used books [1] to [4] so thoroughly that I do not find it necessary to refer to specific chapters.

   An introduction to raytracing. 
   Morgan Kaufmann. 
   ISBN 0-122-86160-4

   Advanced RenderMan: Creating CGI for Motion Pictures. 
   Morgan Kaufmann. 

   Advanced Animation and Rendering Techniques. 
   Addison-Wesley Publishing Company. 
   ISBN 0-201-54412-1

   The RenderMan Interface, version 3.2. 
   https://renderman.pixar.com/products/rispec/ 
   Last visited September 6, 2004.

[5] LibTIFF. 
   http://www.libtiff.org 
   Last visited September 6, 2004.

[6] LibJPEG. 
   http://www.ijg.org 
   Last visited September 6, 2004.

[7] LibPNG. 
   http://www.libpng.org 
   Last visited September 6, 2004.
http://www.libsockets.net
Last visited September 6, 2004.
Appendix A

Terms used

Computer  A single computer has a shared memory and a shared internal communication bus. The computer can have one or several processors.

Performance When referring to a single frame, high performance means short render time. When referring to a renderfarm, high performance means a large number of frames per time unit.

Pixel area A rectangular segment of an image.

Raytracing See section 2.1

Renderer A program capable of creating images from a 3D scene description.

Renderfarm A collection of computers, whose only task is to render images.

Rendering The process of transforming input data into color information. Raytracing and shading are part of rendering.

RenderBjorn The name of the renderer created for this master’s thesis.

RenderMan See section 3.1

RIB RenderMan Input Bytestream. The file format used to transfer RenderMan information.