DAT3 project
Aalborg University
Autumn 2002

Group members:	Trien Huy Ly	jianfai@
	Henrik Thostrup Jensen	htj@
	Jacob Mogensen	jacobm@
	Alex Vendelbo Ringgaard	talzor@
	Kristian Sørensen	ks@
	Martin Guld Thomsen	guld80@

Faculty of Engineering and Science
Aalborg University

Department of Computer Science

TITLE:

Fraja - Distributed ray-tracer

PROJECT PERIOD:

DAT3,
3rd September -
18th December 2002

PROJECT GROUP:

E2-214

GROUP MEMBERS:

Trien Huy Ly
Henrik Thostrup Jensen
Jacob Mogensen
Alex Vendelbo Ringgaard
Kristian Sørensen
Martin Guld Thomsen

SUPERVISOR:

Emanuel Fleury

NUMBER OF COPIES: 9

REPORT PAGES: ??

APPENDIX PAGES: ??

TOTAL PAGES: 77

SYNOPSIS:

This report describes the development of an extension to The Raja Project, making possible to distribute the rendering of pictures to several computers on a network, thus reducing the rendering time significantly.

The report starts with a brief description of basic ray-tracing followed by an analysis of different network aspects i.e. multicast, the concept of a master server and static versus on demand connections. Last in the analysis is a section detailing different way of distributing workloads. Next comes a design chapter that describes the extension made for Raja. This is followed by a chapter on performance analysis, which consists of benchmarks and the analysis here of. Finally we present some ideas for further development.

Authors

Trien Huy Ly

Alex Vendelbo Ringgaard

Jacob Mogensen

Kristian Sørensen

Henrik Thostrup Jensen

Martin Guld Thomsen

Preface

Throughout this report the words *Pixel collection* and *work* both refer to a specific number of pixels from the picture that is to be rendered. Grouped together in a square the coordinates

(5,3), (5,4), (5,5)
(6,3), (6,4), (6,5)
(7,3), (7,4), (7,5)

could be a pixel collection. Also *Pixel collection size*, *collection size* and *work size* are used interchangeably and refer to the side length of a pixel collection square i.e. the above example has a size of 3; and a pixel collection with a size of 9 will contain $9 \times 9 = 81$ pixels.

All classes or methods from the program are written in *italic* and all references to names including external files (e.g. Cone.raj) are written in quotation marks. Note that with .raj files, which is the encoding of the pictures in Raja, the extension is implied e.g. Cone.raj is written as "Cone".

When the name Raja is mentioned, it is referring to The Raja Project while the name Fraja refers to our extension for this program. Note that we have a second extension called Straja. Also we assume that the reader is familiar with the client-server architecture.

The code for Fraja can be found on the Internet at:
http://www.cs.auc.dk/~ks/projects/dat3.tar.gz

User guide

To make use of Fraja you must first edit line 19 the file fraja/src/server/Server.java to set the address of your master server. The same must be done in line 95 in fraja/src/raja/net/client/DistributedRenderer.java.

Next you must make sure that you have GNU make and Sun Java 2 SE 1.4 installed, now the programs must be compiled. This is done by typing "make" in the fraja directory.

To start the master server, go to fraja/lib and type "java MasterServer", likewise to start the server type "java Server". The client, on the other hand, must be started with the start script xraja from fraja/bin. Note that the client cannot run on a host that simultaneously runs a server and/or master server.

Contents

1 Introduction

1.1 Motivation

Ray-tracing is a method for creating photo-realistic pictures, but the price for this is the time it takes to produce the picture. If the picture issue is complex or if the amount of pictures to render is high, then even if you render on a very powerful state-of-the-art computer, it can take a very long time. A way to increase the rendering speed and archive a reasonable rendering time is to distribute the computation over several machines and thereby combining their capacity.

Before commencing on the analysis of the system, a brief introduction to how ray-tracing works is given and some basic concepts are explained.

1.2 Ray-tracing overview[1]

Imagine that you see the world through one single tiny dot, and you are looking at the world through a small rectangular window a short distance in front of you, filled with a wire mesh. The small rectangular window is your computer screen and every hole in the screen mesh represents a pixel in the picture, while your eye is the virtual pinhole camera in the ray-tracing program. Instead of really seeing through the camera, the software mathematically projects a line from the imaginary camera through each pixel on the screen and determines if the line intersects any objects in the scene you have described. Figure 1.1 on page 15 illustrates the basic elements in ray-

[1] This section is based on [3] and [4].

tracing . If the line hits, the pixel is filled with the color of the object at that position, allowing for lights, reflections and other influences. This is called ray-tracing .

This method of rendering allows for relatively easy addition of extra features such as shadows and reflections. For shadows, when a ray from the camera hits an object, the software creates lines from that point on the object to each light you have defined in the scene. If a line does not hit anything else the point is illuminated by that light, if the line intersects other objects in advance of the point, it will be shadowed from that light.

Large images, images containing large numbers of lights, complicated objects and textures, can take a lot of time to render. An often used solution to render the large number of frames needed for a movie, in a reasonable amount of time, is a large number of computers networked together and used specifically for rendering. Such a setup is known as a render farm.

1.2.1 Terminology

Before going into details of ray-tracing , some terminology used is introduced. As already mentioned the world or scene is the place where the ray-tracer draws the image from. The scene consists of objects and light sources. The point where all the rays of light start is called the camera or eye. In front of the camera is the rectangular window filled with a wire mesh. The idea is that each hole in the mesh corresponds to one pixel in the final image, so if you want to create a picture in 1024×768 resolution, you would have a wire mesh in the view window with a grid of 1024 squares across and 768 squares down.

The name ”ray-tracing” comes from the basic idea of ray-tracing: The ray-tracer tries to simulate the way light rays move in the real world. Even though the ray-tracer tries to simulate the real world, there is one striking difference. In order to save precious CPU-cycles the ray-tracer only simulates the light that end up striking the camera. In order to do that, the process has to be reversed by tracing light rays from the camera and moves out through one of the holes in the mesh to draw the picture.

In order to calculate a color of a pixel in the final picture, the ray-tracer creates a ray of light, which it casts out to the scene though one of the holes in the wire mesh. The color of the pixel depends on the color of the object that the ray intersects in the scene, but it also depends on the light sources in the scene. There are different ways to calculate the effect of the light sources in the picture, but we will only explain cosine shading here. The special cases

with reflection and refraction with objects that are more or less transparent or act as a mirror will likewise only be briefly discussed.

1.2.2 Intersections

For each type of object in the scene the ray-tracer has an intersection routine (e.g. a cube). If the intersection routine determines that the ray intersects an object, it returns the distance of the intersection point from the camera and the normal vector at the point of intersection. The distance to the intersection point is needed because if a ray intersects more than one object, we choose the one with the nearest intersection point. The normal is used to determine the shade at the point, since it describes in which direction the surface is facing, and therefore affects how much light the point receives from the light sources in the scene.

1.2.3 Cosine shading

Cosine shading determines the brightness of an intersection point based on the normal vector received from the intersection routine and the light sources in the scene. First it calculates the vectors from the intersection point to lights sources in the scene. If the two coincide, the surface directly faces the light source, then the point has maximum brightness intensity from that light source. As the angle between the two vectors increases, the brightness fades away. The reason this method is known as cosine shading is that the cosine function matches it perfectly. It returns the value 1 when given an 0° angle, and returns 0 when given a 90° angle, when the surface and light source are perpendicular. The cosine function returns values from -1 to 0 when the surface is facing away from the light source, i.e. it does not receive any light. The value of 0 means the surface is directly perpendicular to the light source and therefore does not receive any light. When the surface faces the light source, the function returns values from 0 to 1, the higher values the more light it receives. To prevent that an intersection point not facing the light source should be completely pitch black, there should be added some ambient light to every intersection point.

1.2.4 Reflection and refraction

If the ray of light intersects an object that is reflective, a new ray is reflected from the point of intersection toward the direction of reflection. If the in-

tersected object is transparent, a new ray is refracted into the surface. If the materials on either side of the surface have different degrees of refraction, such as air on one side and water on the other, then the refracted ray will be bent. If the same medium exists on both sides of the surface then the refracted ray travels in the same direction as the original ray. This can quickly complicate the calculations for each pixel. The new ray from the first intersection can also hit a reflective or transparent object and so on. In theory there could be an infinite number of rays for every single pixel in the picture, which of course is inconvenient. The normal way to handle this is to set a maximum to the number of recursive rays, so when a ray for example intersects the 10th reflective object it does not create a new ray.

1.3 Raja

Having the background knowledge in place we will carry on with project which is based on the existing open source ray-tracer Raja, which stands for RAy-tracer in JAva. The Raja Project intends to build a complete modern ray-tracer using the Java language. On the Raja Projects homepage[2], you can download the latest source files, an older and perhaps more stable version of Raja or even try it in a browser without the need to install it. The authors are Grégoire Sutre and Emmanuel Fleury.

[2]http://raja.sourceforge.net

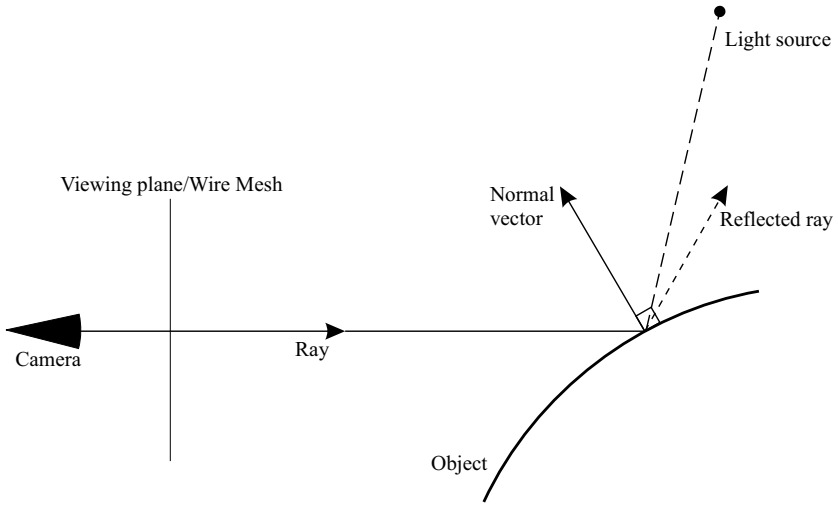

Figure 1.1: Ray-tracing .

2 Analysis

2.1 Object oriented analysis

To understand this section, it is required to know the basics of OOA&D[1], which can be achieved through page 21 to 112 in [2].

Important aspects of the project will be discussed in this chapter. The method applied during the analysis and design chapters is inspired by OOA&D. The result of this chapter is to give a firm understanding of the model of the problem domain. The model is conceived from the system definition which fulfills the "FACTOR" criteria. At first a "rich picture" will be presented in order to introduce how the system is perceived from an outsider's point of view.

2.1.1 Rich picture

Rendering an image by ray-tracing is very processor demanding and thereby very time-consuming. Distributing the rendering task to a group of servers is a way to get this work done faster. This idea of work distribution is encapsulated by the rich picture on figure 2.1 on the next page.

The image to be rendered is a setup containing a description of the objects in the image. This setup is called a scene depicted as the 3D coordinate system. The scene is the first thing that the client sends. Afterward each server receives a portion of the image to be rendered, which the disordered cubic symbolizes. Each server renders its portion of the image and sends

[1] OOA&D: Object Orientated Analysis & Design

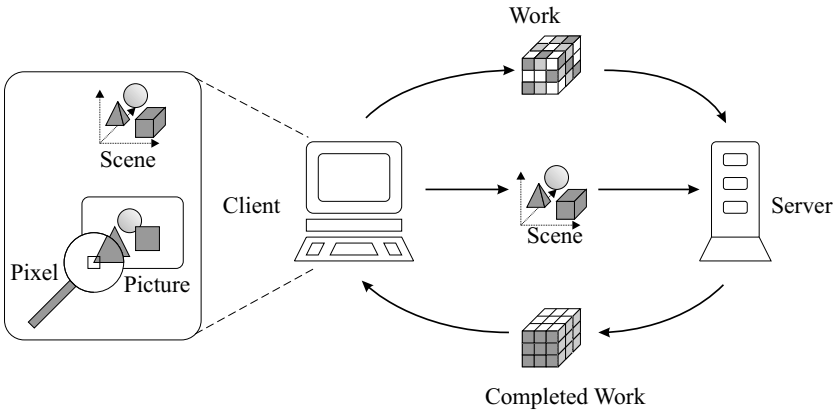

Figure 2.1: Rich picture.

the result of the completed work back to the client. The completed work is depicted as the solved cubic. The client keeps sending portions of the image to the servers until the whole image has been rendered.

2.1.2 FACTOR

The FACTOR criteria are used to make sure that the system definition covers certain important criteria, which are all listed below.

Functionality: *The main functionality that the system offers for supporting tasks in the application domain.*
Distribution of workload when doing ray-tracing in Raja to several servers. The servers are responsible for processing the received distributed work. Once the work has been processed the result will be sent to the client.

Application domain: *One or more organizational units' administration, surveillance or control of the problem domain.*
A Raja render providing work to be distributed across a network of servers.

Condition: *The conditions that determine the system's use and further development.*
Extend the existing Raja with a module responsible for distribution of the rendering task to several servers.

Technology: *The underlying technology for the system.*
The distributed Raja is implemented in Java with the use of a client-server architecture. The developer is expected to be familiar with Java and distributed systems, especially the client-server architecture.

Objects: *A future user's perception of a problem domain.*
Client, server, work and processed work.

Responsibility: *The system's responsibility.*
Ensure distribution of workload and thereby render a 3D image faster.

2.1.3 System definition

The FACTOR criteria result in the following system definition:

When rendering a 3D image the system ensures effective distribution of workload from a client to available servers. The system is an extension of Raja's rendering function. Unlike Raja the system distributes the rendering task over a network of computers. Each computer is responsible for rendering a given part of the 3D image. Once a rendering task has been carried out the result will be sent back to the client. As Raja is implemented in Java so will the extension.

2.1.4 Object structure

Now a description of the structural connection between objects will be presented. The objects are depicted on figure 2.2 on page 21 with the appropriate associations.

Client and server have three associations each. They are both associated to work and result respectively, and to each other. A client can be connected with zero to many servers in order to distribute the workload. Each server is connected to zero or one client in order to provide the rendering service.

The client divides the rendering of a 3D image into work objects, thus the client can have zero to many work objects. Each server is associated with zero to one work objects, thus the server is most of time the busy processing a work object. When processing the second work object, it will have received another work object that is stored in its buffer. In the other direction each work object can only be associated to one client and one server as no servers should process the same work object.

When each server is done processing a work object they send the result back to the client. From this we can infer that each server can produce either zero or one result at a time. This is consistent with the fact that each server processes one work object at a time. As a client can be connected to multiple servers it will also receive multiple results - one from each server. This is indicated in the figure where the client is associated with zero or more result objects. In the other direction each result object can only be associated with one client and one server, which also was the case for work objects.

2.1.5 Behavioral patterns

The object structure described above forms the foundation for the behavioral patterns for the system. The behavioral patterns are illustrated by the state chart diagrams for client and server in figure 2.3 and 2.4 on the next page.

The client starts by sending a scene to the server that it is going to use for the distributed rendering. After sending the scene the client sends out work and it will receive a result from the server. The client stops the distribution of work when the final results have been received.

The server first receives a scene from the client and for each received work the server sends the processed work back to the client as a result.

The server then starts working on the next work object, if such is present in its buffer. If not it will wait for the client to send it more work.

2.2 Server list maintenance

Before a client can start sending out work for the servers, it must acquire a list of servers so that it knows where to send its work. For this, there are several approaches: The user could input the list manually or it could be build ”automatically” using either multicast, broadcast or having central server which keeps track of the servers available.

Entering the list of servers manually, would quickly become quite annoying and tedious; especially when there are a lot of servers. Therefore an automated approach was greatly preferred. The central server was dispelled since it would make the server list distribution depend on a single host. Broadcast could only be used on LANs, therefore our initial choice fell on multicast, since it could be used both on LANs and on the Internet.

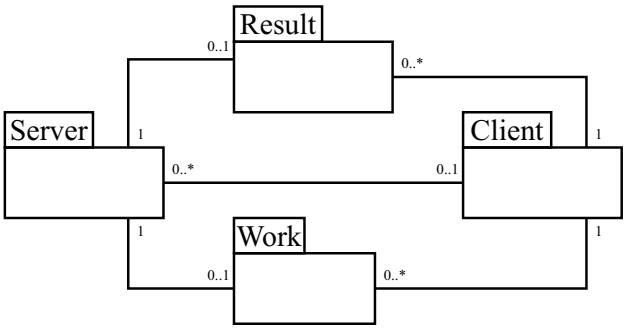

Figure 2.2: Object structure.

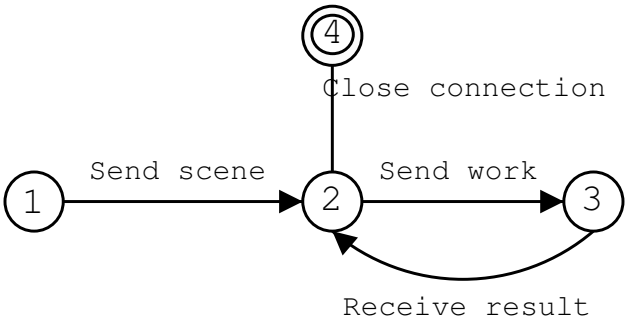

Figure 2.3: State chart diagram for client.

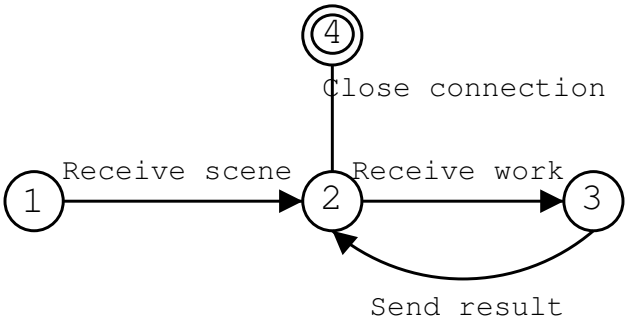

Figure 2.4: State chart diagram for server.

2.2.1 Multicast

If one have some data that are to be distributed on the Internet to an unknown group of hosts, it would be nice to have some sort of way to broadcast this information. However the Internet today consists of several million hosts, which mean that simply broadcasting the data to all hosts on the Internet is not a viable option.

Multicast omits this problem by creating a group of routers, where to data can be "broadcasted". This works by cooperation between the routers connecting intranets to the Internet and the hosts behind these routers. In these two layers most of the complexity is in the routers. Seeing it from the hosts side, multicast is very simple. The host simply asks its (multicast enabled) router, to add it to a multicast group. The router then adds it to the group and the host will start receiving whatever data is sent to the group, and is able to send to the group as well.

Viewing it from the routers side, multicast is somewhat more complicated. When the router receives a "subscribe" from a host on its intranet, the router checks whether such a group currently exists. If this is not the case the router creates this group.

If the group already exists the router must add it self to this group of routers. This group of routers is usually connected by a tree structure, meaning that no cycle can exist between them. This means that a router can simply send the multicast data that it received to all multicast connections except the one from which it received that data. By using this approach all data will be distributed to all routers, which will distribute it to their subscribed hosts. This scheme is depicted on figure 2.5 on page 26.

There exists several protocols for multicast, but in principle they work by the scheme described above. Describing them and their differences is out of this reports scope, and as it will be revealed later, we choose not to go with multicast.

Our plan with multicast

To build the server list we initially wanted to use multicast, since it has some nice properties. It would not make the distribution of the server list dependent on any single machine (this is of course only true if more than one router is used). Also it would make the implementation of server list distribution quite simple, since a lot of the complexity is located in the router. These two things added together would make the server list simple, while still

being quite robust.

To distribute the server list, all the Raja servers would have to join a certain multicast group. When a client wishes to receive a server list, it asks its router to join the specific group and sends a message to the group saying that it wants a server list. Each server responds by sending out a packet to the multicast group containing its IP number and port. From this information the client can build a list of the available servers, which enables it to connect to the Raja servers. The client would send this request, at a certain time interval to get any new servers. Another approach would be to have the Raja servers send out a packet containing its IP and port number at some time interval, and have the client building its list from this information. We created a working implementation from the last idea. However we discovered some problems with multicast. These will be described in the next section.

Troubles with multicast

At first multicast seems like a very nice idea. It is relatively simple to understand and Java has a class *MulticastSocket*[2] which should make the implementation straight forward.

First we thought that to use multicast, a multicast enabled routers was needed. This was not a problem since such a router existed at our institute. We were worried that it could not be used on LANs without a multicast enabled router. However we discovered that a switch which receives a packet that it does not know where to send (which is the case with a multicast packet), simply broadcasts the packet, meaning all hosts would receive the package. This would mean that on a LAN, multicast works almost like broadcast, meaning that all hosts would receive the multicast packets send. Unfortunately this is not the way things works in practice.

When a switch receives a multicast packet it broadcasts it. However it is not received by anyone, since Ethernet cards immediately throws away packages that is not for its own MAC address. And thus when packages is sent to the multicast group no one would receive it. Actually you can enable the Ethernet cards to accept all packages that it receives, by setting it into ”promiscuous mode”. However this does not work, since the Java *MulticastSocket* does not receive the packages for some reason. But even if it did, setting the Ethernet card to ”promiscuous mode” requires root/administrator access to the computer, which we cannot assume the user of Fraja will have.

[2]http://java.sun.com/j2se/1.4.1/docs/api/java/net/MulticastSocket.html

This means that to be able to use multicast one does indeed need a multicast enabled router. This router can then receive a packet from a host which wishes to join a multicast group. When the router receives data for the multicast group, it sends data out on the interfaces to which hosts who want to receive multicast are connected. This data has to go through some switches to reach the destination. Whether these have enabled multicast is a matter of finance and security. On small low cost switches the packages would just be broadcasted, but on expensive switching hardware (e.g. Cisco or 3COM) you have the option to configure the switch to accept multicast and broadcast.

Due to the above reasons we chose not to use multicast. The only place we know of, where multicast equipment is available and configured properly is at our department: Institute for Computer Science at Aalborg University.

2.2.2 Master server

The process to build the server list by using multicast or broadcast turned out to be almost impossible in practice. Therefore we choose to use a central server, i.e. named a master server. A master server has the responsibility of building the server list and providing it for the clients. If a server crashes then the master server must ensure that this server is removed from the server list.

Server architecture

The network architecture model for the master server is similar to the ones used in many modern computer games, like the popular game ”Counter Strike”.

The system is based on a master server, which only job is to maintain a list of active servers. This master server has to be known to the regular servers and the clients when they are started. The servers will then periodically send packets to the master server which will then build a list of active servers. A master server will remove a server from the list if it fails to report in after a given amount of time. Now that the master server has this list, all a client needs to do, is to regularly send a request for the server list to the master server. For an overview of the client, server and master server see figure 2.6 on page 26.

This is a solution that scales quite well, both on the Internet and on a LAN since adding an extra server or client to the network will only cause a linearly increase in the amount of packets being send.

The weakness of this choice is that the creation of the server list depends on the master server. If the system only has one master server, it is necessary to determine who should take over if it crashes.

To avoid the above weaknesses a server architecture with multiple master servers can be used. The strength of multiple master servers is the ability to maintain a shared master server list and coping with up to $n - 1$ master crashes, where n being the number of master servers.

Ways to implement multiple master servers can be by electing a master server between potential master servers or by having several master servers running concurrent. These two different ways are discussed next.

Election

Election is used for choosing a unique process to play a particular role. This unique process corresponds to the master server. The election can be done by using a ring based election algorithm or the bully algorithm. As the first algorithm is not capable of coping with crashes, thus focus will be on the latter algorithm.

The bully algorithm has three assumptions:

1. The system can detect timeouts.
2. Each process knows which process have higher identifier.
3. The process can communicate with other processes with higher identifiers.

The bully algorithm selects the process with the highest identifier. A process begins an election when it notices a timeout. The election is done by sending three different types of messages:

Election message: This message is sent to announce an election. When a process receives an election message it will in turn start its own election by sending election messages to processes with higher identifiers than itself. If a process has already started an election, it will not start a new now.

Answer message: On receipt of election messages the process returns an answer message to the sender.

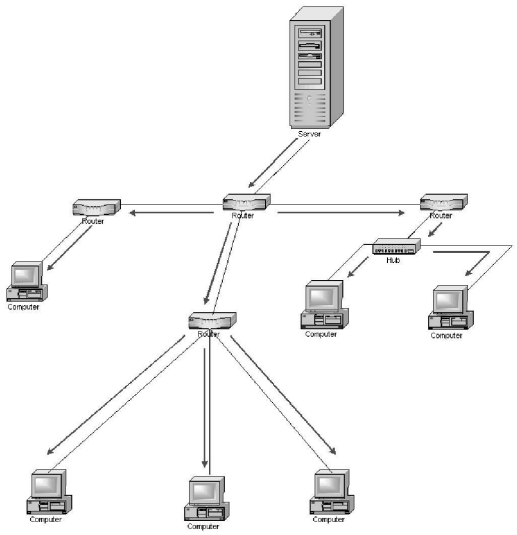

Figure 2.5: Multicast.

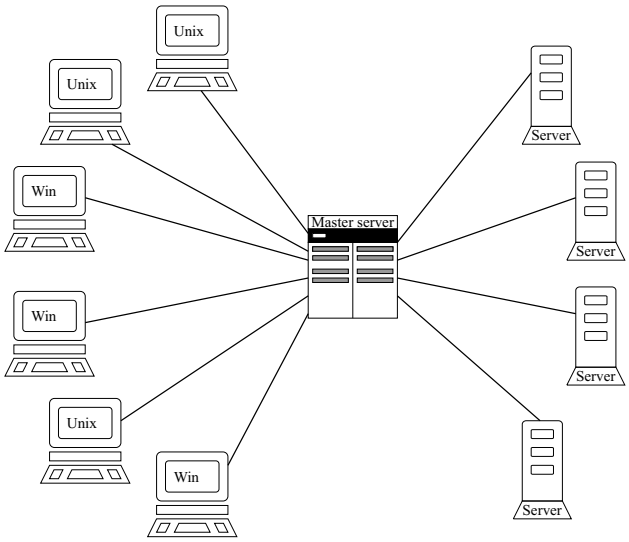

Figure 2.6: The server architecture with one master server.

Coordinator message: This message is sent to announce the identity of the elected process. The process sending the coordinator message knows that it has the highest identifier, thus it knows it is the new coordinator to be elected.

The bully algorithm ensures that a process is elected even though one of the processes crashes. This is depicted in figure 2.7 on the following page which sketches a situation with four processes.

Stage 1: Process 1 detects that process 4 has crashed, thus it starts an election by sending an election message to process 2 and 3.

Stage 2: If process 1 does not receive any answer messages within a certain time bound, it elects itself as the new coordinator. In this example we assume that process 1 receives answer messages from 2 and 3. Now, process 1 waits for a coordinator message to be received within yet another time bound. If such a message does not occur, it will start a new election.

Stage 3: Process 2 and process 3 start an election independent of each other.

Stage 4: Process 3 does not receive an answer from process 4 within a time bound as it has crashed, thus process 3 elects itself as the new coordinator. This is announced by process 3 by sending a coordinator message to process 1 and 2.

In the following the run time of the bully algorithm will be analyzed with focus on worst case and best case. Worst case and best case are used to compute the running time of an algorithm purely as a function of the length of the input. In worst case we consider the longest running time of all inputs and in best case we consider the shortest running time. Average case could also be considered but it is too complex to compute the average of all the running times of inputs of a particular length.

The performance of the bully algorithm is $O(n^2)$ in worst case and $O(n)$ in best case.

Worst case: Occurs when the process with least identifier detects that the coordinator has crashed. Upon this detection the process starts an election by sending election messages to processes with higher identifier than itself, which in this case is all the process except the coordinator.

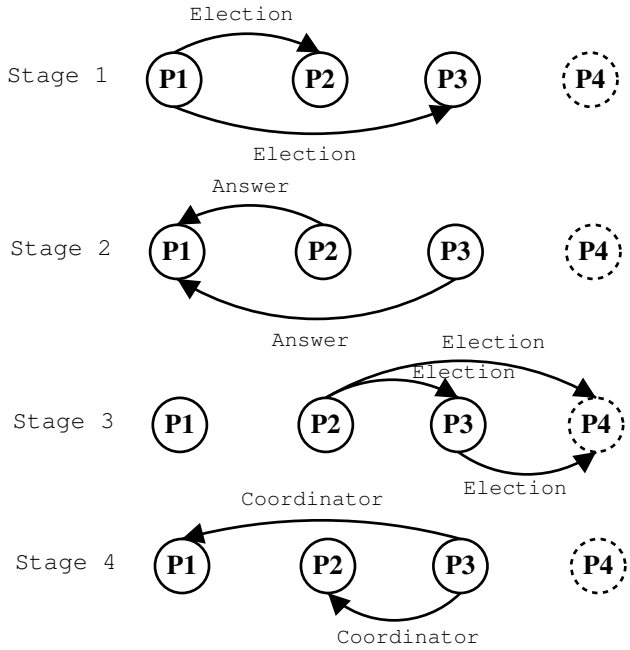

Figure 2.7: Process 4 crashes and a new process is elected based on the bully algorithm.

Every processes receiving such a message will itself start an election. The process with least identifier will send $n-1$ election messages. The second least process will send $n-2$ election messages. Continuing like this the process with second highest identifier will send one election message to the process with the highest identifier. Mathematically this looks like this:

$$(n-1)+(n-2)+\ldots+2+1+0$$

which has an upper bound corresponding to $O(n^2)$.

Best case: Occurs when the process with highest identifier detects that the coordinator has crashed. This implies that this process elects itself as the new coordinator. The process only has to send (n - 2) messages to the other processes. These messages are the coordinator messages.

In this system the server and client could call an election when a master server timeout is detected. A new master server is then elected between the collection of potential master servers.

However two problems arise when using this election algorithm to cope with failure of master server.

1. Relying only on one master server to maintain a server list can cause bottle neck when too many clients and servers connects to a single master server.

2. Two processes can elect themselves as the new coordinator under certain circumstances. When a coordinator crashes and is replaced by a new process taking the crashed coordinator's identifier. During this replacement an election is running. A process with a lower identifier might announce itself as the new coordinator unaware of the process replacement has taken place. Concurrently, the newly replaced process announces itself as the new coordinator, as well. As these two coordinator messages can arrive in different order at the other processes, they can draw different conclusions on which is the new coordinator.

We have investigated further options in order to avoid the two problems mentioned above.

Replication

Instead of having just one master server and choosing it by election the system could consist of several master servers sharing the same server list. Replication can provide high availability and fault tolerance by propagating server list updates between the master servers. The new master server architecture is depicted in figure 2.8 on the next page.

With the introduction of the new master server architecture the clients and servers can use any master server of their choice. If a master server crashes then any of the other master servers can be used instead. As depicted in figure 2.8 there is a collection of master servers for the clients and servers to choose among. In order for this to be possible a list of master server needs to be maintained at the clients, the servers and among the master servers themselves.

Master server list The master server list needs only to be updated when a new master server is added or an existing master server has crashed.

The following describes the situation when a new master server is added (see figure 2.9 on the facing page):

1. When a new master server is to be added, the new master server sends an add message to one of the currently running master servers; on startup every master server has a default list of master servers.
2. The master server receiving the add message returns a master server list and a server list, as these are the most updated ones.
3. The master server informs the other master servers on the received master server list that it has started. The corresponding master servers adds the new master server to their master server list.
4. Next time when the client requests for a new master server list and the master server has a new master server list the master server replies with the master server list.
5. The master server informs the local servers, i.e. the servers connected to it, of updates by sending the new master server list.

The following describes the situation when an existing master server crashes (see figure 2.10 on the next page):

1. The server or client detects that its master server has stopped responding.

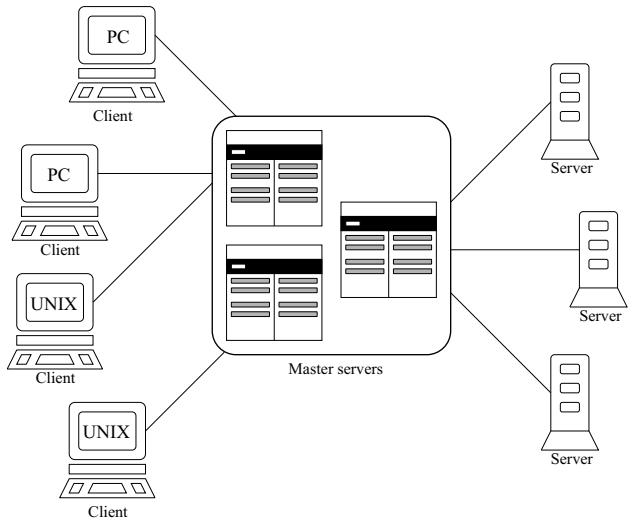

Figure 2.8: The new master server architecture.

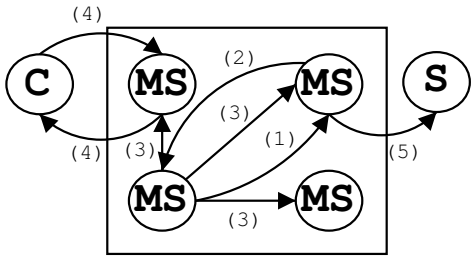

Figure 2.9: How a master server is added. MS is a master server, C is a client and S is a server.

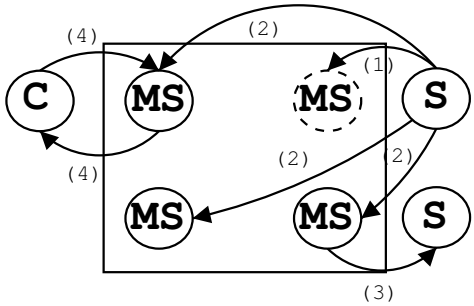

Figure 2.10: How a master server is removed. MS is a master server, C is a client and S is a server. The dotted lines illustrates that the master server has crashed.

2. The server or client sends a remove message to all the running master servers.
3. Each master server will inform their local servers of the master server removal by sending the new master server list to the server.
4. The client will receive the new master server list next time it requests the master server list from a master server.

Server list The server list needs only to be updated when a new server is added or an existing server stops providing rendering service.

The following describes the situation when a new server is added (see figure 2.11 on the facing page):

1. The first message telling the master server that the server has started is sent from the server to one of the master server.
2. The receiving master server returns a master server list.
3. The server sends an add server message to all the master servers in the received master server list.
4. The client receives the updated server list upon the next request.

The following describes the situation when an existing server stops providing the rendering service (see figure 2.12 on the next page):

1. The master server stops receiving messages from the server.
2. The master sends an update telling the other master servers to remove the server in question.
3. The client receives the updated server list upon the next request.

This master server architecture provides robustness into the process of continuously maintaining an updated list of servers for the client. For this to happen a list of master servers needs to be maintained as well, and distributed to the client and servers. The robustness is expressed through the situation where a client looses connectivity to its master server. Based on its list of master servers it can connect to the next master server and resume work as nothing has happened. This is also the case for the servers.

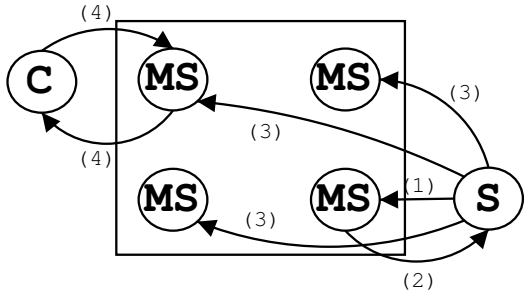

Figure 2.11: How a server is added. MS is a master server, C is a client and S is a server.

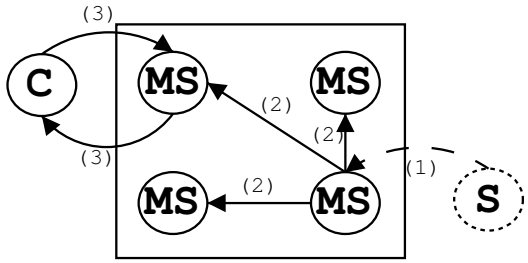

Figure 2.12: How a server is removed. MS is a master server, C is a client and S is a server. The dotted lines illustrates that the server has crashed.

2.3 Connectivity

The method for connectivity is one of the important decisions for distributing Raja. We have considered two solutions, static and on demand connections.

Having a static connection for each server makes administration of connections and renders on the servers side easier, since they have a one to one relationship. When using static connections a large number of sockets will be used on the client i.e. one for each server, but sending data through the connections, can be done without doing TCP handshake first.

When using on demand connections the server must determine from which client the data comes in order to render in the right scene. On demand connections requires opening and closing of the connections, when sending data. This implies an overhead for TCP handshake.

For the main program we chose to implement the on demand approach, but for benchmarking purposes we have implemented a simple version using static connectivity.

2.4 Distributed ray-tracing

As described earlier, rendering large images or movies can take quite a lot of time. A way to reduce this time, without comprising quality of the picture, is to spread out the calculations on several computers. In this section it will be described how to parallelize ray-tracing, so that the calculations can take place on more than one CPU. Furthermore it will be explained how to split up the work, e.g. the picture and how to balance the work load across several machines, so that none of them will stand idle, while the others are working.

2.4.1 Parallelizing ray-tracing

To parallelize ray-tracing the rendering task needs to be split into parts which can be calculated individually. The obvious choice is to have a pixel represent a single work unit, since each pixel is independent of each other. However a pixel can be split into smaller parts, i.e. the rays that is casted when the color of pixel is calculated. Unfortunately these rays are dependent on the previous ray, meaning that a pixel has a highly recursive nature, making it quite hard to parallelize since ray calculation is dependent on previous data. Taking into consideration that a single pixel does not take a lot time

to render, parallelizing a pixel calculation does not make a lot of sense.

In some extreme cases, where the scene is too big to fit into a single computer, e.g. a lot of textures, splitting up pixels into rays might be necessary since a single computer might not hold all the textures needed to calculate the color of a pixel. Since Raja does not support textures, the scene description is relatively small. Furthermore the scene is immutable meaning that it will not change over time. These two things combined gives that the complete scene can easily be distributed to all servers.

After the scene has been distributed, the client can start sending out work, i.e. pixel coordinates. The server can then calculate the color of the pixel and return it to the client. Using this scheme on several servers simultaneously, the rendering process has now been parallelize. This process can be optimized, since there is an substantial overhead when just sending a single coordinate over network. The rest of this section will deal with how to optimize this process, so that the picture can be rendered in the fastest possible way.

2.4.2 Models for distributed ray-tracing[3]

In this section we will discuss the different models for work splitting. The main goal for the models is to minimize the time spend on a computation, the models should therefore avoid processor idle time. The ideal solution is that all processors have 100% load under the computation and that they all finish in the exact same time. The last part is important to avoid that all processors except one have finishes their computation and therefore have to wait on the last processor to finish. Load balancing is therefore a very important aspect of the models. If all the processors in the system have the same processing power, the load balancing would be perfect if all processors have a job of the same size, but if the processors have different speeds and starts the computations at different times it become more complicated.

The models for work splitting can be categorized in two groups. Domain decomposition and algorithmic decomposition. Domain decomposition can be used, when the work can be split into domains, which can be solved independently of each other. Algorithmic decomposition can be used, when the algorithm contains routines that can be executed in parallel. In this section we describe two domain decomposition methods, data driven and demand driven. We also describe two algorithmic decomposition methods,

[3]This section is based on [1].

fork and join and temporal parallelism.

Data driven

The data driven model copy the rendering algorithm to each processor in the system, but only a part of the problem domain is computed on each of the processors. The problem domain is split into a number of pieces of equal sizes, where the number of pieces equals the number of processor in the system. Then the entire data required for the computation is transferred to each processor. The data driven model is a static model, because the entire problem domain is divided and distributed to each of the processors in the starting phase of the computation.

If there are differences in the processors processing power, it is often advantageous to use an unbalanced version of the data driven model.

The problem is that the fastest processor will computes its work faster than the other processors in the system, so instead of splitting the problem domain into equal sizes, the problem domain should instead be split into pieces whose sizes stand in relation to the processors. The ideal solution is that processor A with the double amount of processing power is seen in relation to processor B, should also have a piece of the problem that is twice as big as processor B, so they each finish at the same time. Unfortunate this model requires that both the weight of each pieces and each of the processors processing power should be known before the computation. The normal approach is to approximate the weight of the pieces with some initial computation, this will of course add some overhead to the algorithm. The processors power can also be estimate with some computation.

It is straight forward to use the data driven model to implement distributed ray-tracing, but in order to reach an acceptable load balancing it is very important that the picture is analyzed before the distribution of the workload. If this is not the case, it is very likely that one processor end up with a more complicated part of the picture, it could e.g. be a part with a lot reflective surfaces. The advantage with the data driven model is that there is not a lot of communication under the computation. After the workload has been distributed, only the results of the computations are communicated.

Task stealing A way to optimize the data driven distribution model is to add task stealing. Task stealing assumes that a each server has a list of all the other servers rendering. When a server has completed its work batch it sends the result back to client. Hereafter it asks a random server to hand

over half of its unrendered work batch. This way servers ”steal” work from each other, making sure that no servers stand idle, during the rendering. Even though this scheme gives no guaranty that a single server will not be standing left and rendering, while all the other servers are asking each other for work, task stealing works quite well.

Demand driven

The demand driven model allocates the work dynamically to the processors. The problem domain is split into a lot of work pieces and when a processor becomes idle, it requests a new piece of work. The idea is that the client have a pool of available work pieces, which it can feed to the servers, when they request a new piece. When the work pool becomes empty and the computation is not completed, the client may begin to assign work pieces that already have been assigned or the client can choose to ignore the request and simply wait for the other processors.

When a processor finishes a work piece, it will be idle until it receives a new work piece from the client, this can easily be avoided with a buffer. The buffer should at least have a size that allow the processor to work at 100% load and still not finish before the new work pieces is received. With a large buffer, a processor could end up having a buffer filled with work pieces, when the rest of processor have finishes their computations, the buffer should therefore not be extremely large.

The demand driven model is a bit more complicated to implement than the data driven model, but the data driven model require analysis of the problem domain and the processors attaches the system to reach an acceptable load balancing. The demand driven approach solves the load balancing problem completely, it is also more robust than the data driven model. If a server suddenly loses processing power when another task is started on it, it is still able to adopt to the situation. A work piece could consist of a number of pixels in the image, and it should therefore not be difficult to split problem domain. The buffers should be analyzed to find the best size for them.

Fork and join

When having a substantial amount of servers, the client can become so busy communicating with the servers, that it cannot keep up sending work to the servers, meaning that the servers will start having idle time. A way to overcome this problem is to use the fork and join scheme. Fork and join

works by creating a tree structure with the client as the root and the servers below. For simplicity we will assume that the tree is a binary tree. The method works by the client splitting up the picture in three parts. One for it self to render, and one for each of its children. The servers continue to hand out work down through the tree. When the servers are done with their work batch they send the result to their parents, which sends their work batch and its children results to their parent. This way the client will receive a part of the completed picture from each of its children.

This approach has some of the same problems as data driven, because some servers will be idling while others are still rendering. To overcome this problem the task stealing method can be applied. Fork and join is usually used when the client is too overloaded to keep the servers busy. The base model is quite sensitive to server failure, since it can be hard for the client to recover in reasonable time if one of its children dies. Death of a child would result in losing part of the picture. Various methods for overcoming this problem can be thought of, but we will not go into them here.

Temporal Parallelism

Parallelizing ray-tracing by splitting up the picture into several parts, is usually know as functional parallelism. However another form of parallelism exists. When rendering movies the time taken to render a single frame (i.e. a picture) is not so important. Therefore a single machine can be set to render one frame, and another machine to render the next frame. This form of parallelism is known as temporal parallelism. It can of course only be used when rendering movies. When using temporal parallelism the communication overhead is quite low since the client only have to send out the scene, and a time coordinate, and then wait for the server to finish the frame. Temporal parallelism is only useful when you have the time to wait for the frames.

3 Design & implementation

3.1 Requirements

Based on the system definition we are able to list a number of design criteria. Figure 3.1 marks the importance of each criteria. The reasons for the choices will be elaborated right after.

Efficient: *Optimal usage of the technical platform.*
The idea of distributing the rendering task is to speed up the rendering process.

Correct: *Fulfilling formulated conditions.*
Constraining the distribution of data between client and servers by

Criterion	Very important	Important	Less important
Efficient	√		
Correct		√	
Reliable	√		
Flexible	√		
Comprehensible		√	
Reusable			√
Interoperable		√	

Figure 3.1: Checklist for prioritizing design criteria.

means of a protocol in order to ensure the correctness of the rendering task.

Reliable: *Fulfilling demanded precision in execution of functions.*
The system should be failure tolerant. If one or more servers or master servers crashes, this should not have any impact on the entire system.

Flexible: *The cost of modifying the deployed system.*
The system should highly modularized. The modules such as client and server should be exchangeable as long as the interface is kept unmodified.

Comprehensible: *The effort needed to obtain a coherent understanding of the system.*
Little effort should be needed in order to understand the system as the code will be well commented and the class structure is modularized.

Reusable: *The potential for using system parts in other related systems.*
Fraja has been developed as an extension to The Raja Project and the code is not intended for use with any other projects.

Interoperable: *The cost of coupling the system to other systems.*
The spirit of the system is to integrate it into Raja, thus interoperability is considered important.

3.2 Protocol

The purpose of the protocol is to lay out rules for communication between client and servers. Furthermore the protocol is to ensure that only valid objects and commands are exchanged by the client and servers. In addition, timeouts occur in order to detect possible client and server crashes.

3.2.1 Protocol design

There has been designed a protocol for client and server respectively. The client protocol ensures the correct order of sending a scene and a number of work objects. In addition, it also ensures the correct receival of processed work objects. However, timeout can occur while waiting for the result of a previously sent work object. The client protocol is depicted in figure 3.2 on page 42.

The server protocol ensures that the scene and work objects are received in a correct order. The sending of results are carried out in accordance to the previously received work objects. Timeout can occur in two cases: When waiting for a scene and work-objects. The server protocol is depicted in figure 3.3 on the following page.

3.2.2 Protocol implementation

Section 3.2.1 on the preceding page describes the protocol as we intended to implement it. It turned out that this protocol is too simple. In the following the actual implementation is described, with all its special cases and extensions.

Scene and work transfer

When sending the scene and work to the servers, a thread for each task is started. We do not control the sequence of thread executions, thus the work may be transferred to some servers before the scene. When a server receives work without having a scene, it saves it in a buffer until the scene is received. The server will never receive more than two chunks of work, as the client do not sends out additional work to a server, before it has received a result from that server.

Timeouts

Every time a client or a server tries to send an object to each other the transmission might timeout. This happens if the sending part cannot make a connection within 90 seconds. The two different systems, the client and the server, behaves differently when a timeout occurs. The client will inform its dispatcher, which in turn will tell the scheduler to set the state of the pixels in question to ”ready”, see 3.3.2 on page 44. The server on the other hand will simply discard the work and nothing else as it assumes that the client is dead. Note that if the client is not dead the server will never receive work from this client again, as the client will interpret the lack of response from the server as a server crash.

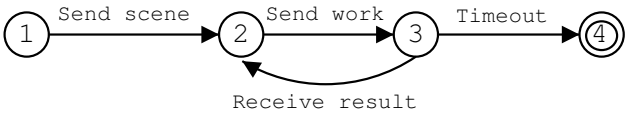

Figure 3.2: Client protocol.
(1) Init: Start state before sending the scene.
(2) Scene sent: Sent a scene or received a result and ready to send work.
(3) Work sent: After sending work and waiting for the result.
(4) End state: After the occurrence of timeout.

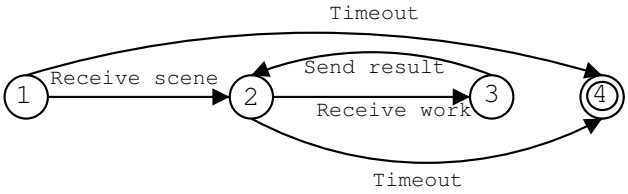

Figure 3.3: Server protocol.
(1) Init: Start state before receiving a scene.
(2) Ready to work: After receiving a scene or sent a result and ready to receive work.
(3) Work processed: After receiving work and before sending the result.
(4) End state: After the occurrence of timeout.

3.3 Overall Picture

The following three sections will give a short explanation of the class diagrams. Please note that while the name of the actual classes, i.e. the Java files, have been prefixed with the name of the part of the system to which they belong e.g. Client or Server, these prefixes have been exclude from the diagrams to increase readability.

3.3.1 MasterServer

The *MasterServer* class is the core of the master server program, it is responsible for maintaining a list of servers that are believed to be active. When launching the *Startup* class handles the issue of connecting to a running master server to get an up-to-date master server and server list. The list of servers and master servers is maintained by using a list of *ServerElement*s and *MasterServerElement*s. The list of *MasterServerElement*s is located in the *MasterServers* object, which also is the object that is sent from the master servers to the servers and clients, when transferring the master server list. Each *ServerElement* contains an IP number and a timestamp that tells when the server was last heard from. Each *MasterServerElement* contains the IP address and the ports of a master server. To maintain the lists the *MasterServer* starts a *ListenerUdp* thread that listens for UDP[1] packets. The master server checks a number in the package that indicates whether it is from a server stating that it is available or from a client requesting a list of servers or master servers. In the latter case it will inform the *MasterServer* which in turn will create an *ObjectSender* to handle the actual transmission. The *MasterServer* also starts a *ListenerTcp* which is responsible for receiving master server and server lists from other master servers.

To determine if a server has stopped sending ”ImAlive” messages stating that it is available, the *MasterServer* initially starts a *Cleaner* that at a given interval tells the *MasterServer* to check its list for dead servers. This is done by comparing their time stamp to the current time.

The choice of UDP as ”ImAlive” packages from the servers to the *MasterServer* is based upon the fact that package loss is acceptable and there exist no need for a connection.

The above classes and their relations are all depicted in figure 3.4 on page 45.

[1] UDP: User Datagram Protocol.

3.3.2 Client

The *DistributedRenderer* is responsible for instantiating a *Client* which is the core of the client program. At first the *StartUp* class finds an active master server from the default master server list. The *Client* uses the *ServerListRequester* and *MasterServerListRequester* to periodically contacts the master server asking for an updated server list and master server list, respectively. The *ListReceiver* maintains a list of active servers and master servers by receiving server lists and *MasterServers* objects from the master server. Furthermore the *Client* instantiates a *Dispatcher* that handles the distribution of the work to the servers. This is done by asking the *Scheduler* for pixel collections and then starting an *ObjectSender* to take care of the actual transmission of either a server list or master server list.

The above mentioned *Scheduler* provides functionality to determine which pixels should be sent next. This is done by searching a *PixelMatrix*, which in principal functions as a two dimensionally array containing *PixelMatrixElement*, that either have been computed (done), is being computed (pending), or neither (ready). The *Client* have a *Receiver* that listens for completed work being sent by servers.

To detect whether a master server has crashed the *MasterServerChecker* periodically checks a timestamp for the currently used master server.

The above classes and their relations are all depicted in figure 3.5 on the facing page. The *MasterServers* object and the associated *MasterServer*-*MasterServerElement*s have been excluded for a better overview.

3.3.3 Server

The *Server*'s only task is to start a *KeepAlive*, a *ListenerUdp* and a *ListenerTcp* thread. The *Server* also has a *StartUp* class that connects to an available master server. The *KeepAlive* will periodically send a message to the master server, stating that the server is still active. The *ListenerTcp* listens for work being sent by a client which it passes on to a *CalculaterWrapper* that will compute the pixels and then start an *ObjectSender* to send the complete work back to the client. The *ListenerTcp* also listens for *MasterServers* which can be sent from the master server. The *ListenerUdp* class listens for packages from the master server. The *MasterServerChecker* class checks the timestamp of the current master server and makes sure to change master server at time out. To handle different clients the *ListenerTcp* maintains a list of *ClientInfo* objects, which holds all the information needed to compute

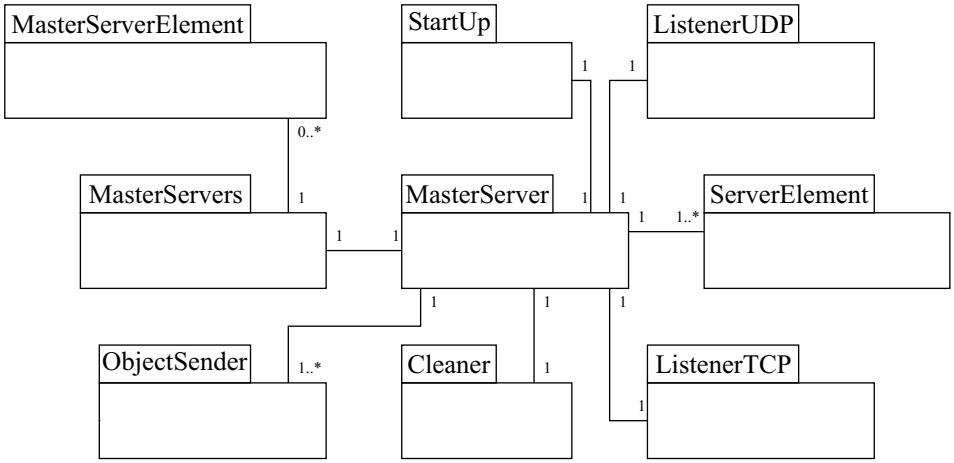

Figure 3.4: Class diagram of the master server.

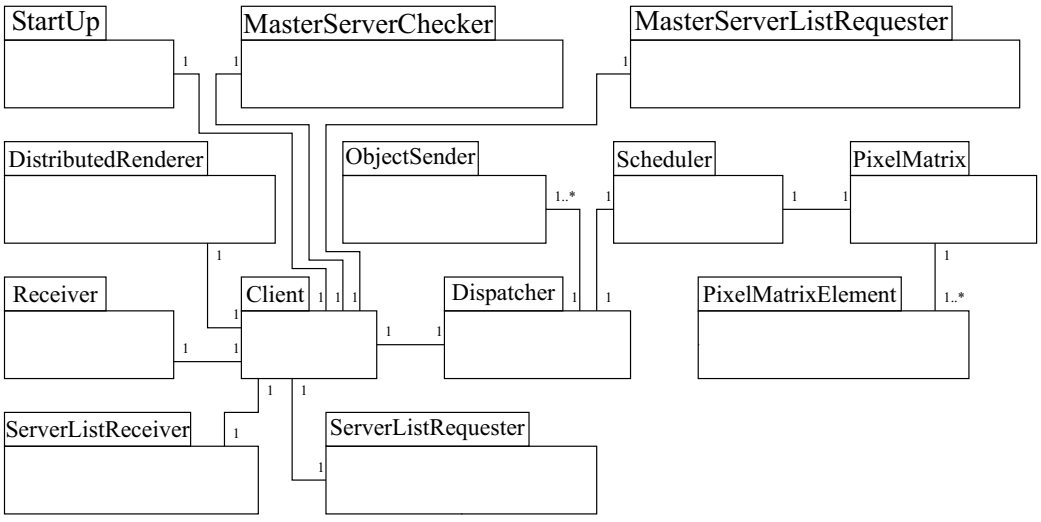

Figure 3.5: Class diagram of the client.

work from a specific client. The *ClientInfo* objects also hold a buffer to store extra work from the client, so that the server does not need to idle while waiting for new work after it has finished a pixel collection.

The above classes and their relations are all depicted in figure 3.6 on the next page. The *Server* has the same association to *MasterServers* as the *MasterServer*, so *MasterServers* has been excluded from the diagram.

3.4 Network

Fraja uses two ports for the communication between servers, master servers and clients. These are port 50057 and 50058. The actual connections are depicted on figure 3.7 on the facing page.

The client listen on port 50057 and 50058 for data transferred using TCP. Port 50057 is for the reception of calculated pixel collections and port 50058 is for server and master server lists.

The servers listens on port 50057 for *RenderWrapper*s, pixel collections, server and master server lists using TCP. Port 50058 is used for receiving acknowledgments for previously sent "ImAlive" packets using UDP.

The master server listens on port 50057 for updates of server and master server list from other master servers. Port 50058 is used for UDP packets from both client and servers. The client sends to this port when requesting for server and master server lists. The servers sends "ImAlive" packets to this port.

3.5 Picture splitting

3.5.1 Basic splitting

The *BasicScheduler* is implemented using a demand driven approach to picture splitting, meaning that every time a server need some new chunk of pixels it receives the next available one. This is done without regard to the capacity of the servers or the complexity of the chunks.

As mentioned earlier the scheduler has a matrix that it runs through to find a pixel collection to distribute. To avoid running through the matrix every time a new collection is to be found, the scheduler remembers the position of the last found collection and starts from there, when asked to find another.

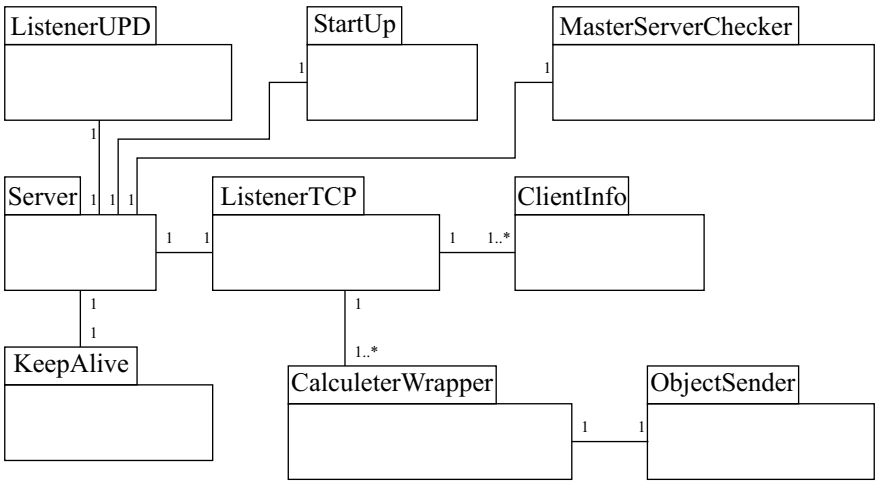

Figure 3.6: Class diagram of the server.

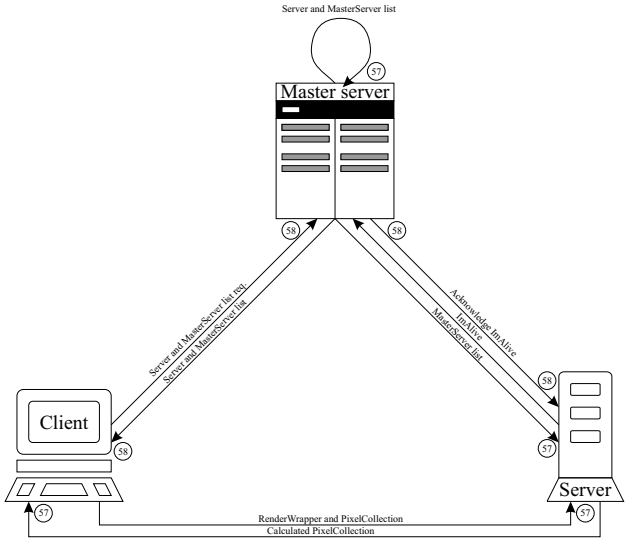

Figure 3.7: Communication ports.

Should the scheduler fail to find any pixel that has not already been sent out to a server it will start to "overbook" collection, that has not yet been return, starting with the one it has waited for the longest time.

3.5.2 Advanced splitting

A more advanced method is picture analysis. This would require the system to try to estimate the complexity of a chunk, so that it can be sent to a server with a matching capacity. This requires the following two things.

First the system must have some method to determine the capacity of a server. One way to do this is to send the server a dummy chunk and have it returned the time it took to compute it.

Secondly the system needs to be able to compute the complexity of a chunk. As this is very time consuming, the system instead estimates the complexity. This is done by tracing a ray through the center pixel of the chunk without doing any computation of the color, and only counting the number of times the ray reflects. Since the pixels surrounding another pixel most likely will hit the same object as the pixel in question, the estimate for the center pixel is a good approximation to the complexity of the entire chunk. Should a more precise estimate be needed, the system can trace more rays through the chunk. Although this is not as time consuming as calculating the color, the overhead of analyzing must not exceed the time gained be doing it.

Advanced splitting is clearly more suited for a data driven approach.

3.6 Straja

This section describes how Straja works and especially how it differs from Fraja. It will not be described as detailed as Fraja, as the two versions are much alike.

Since Straja was developed to make benchmarks, a lot of functionality could be omitted yielding a simple design. In the following it will be explained how Straja's client and server works.

3.6.1 Client

The class diagram for the Straja client is depicted on figure 3.8 on page 50. The three most important classes are *DistributedRenderer*, *Client* and *Client-*

Connection. Beside these classes, the *Queue* class also appears on the diagram.

Straja's *DistributedRenderer* does almost the same as Raja's *BasicRenderer* and creates a *Queue* and *Client* object. The queue is used for pixels returned by the servers. The *Client* is responsible for creating connections to each server. When a connection has been made, an instance of *ClientConnection* will be made to handle traffic that is sent between the client and server. After a connection has been established the thread in *Client* will start sending work out.

3.6.2 Server

The class diagram of the server part of Straja is depicted on figure 3.9 on the following page. Whereas the Straja client is integrated into Raja itself, the server part is more independent. It uses Raja's classes to render the color of pixels given to it by the client. When a server is started it creates a *ServerListener* which listens for new connections on a given port. When a listener has established a new connection, it creates a new *ServerConnection*. The *ServerConnection* class works much like the class *ClientConnection*. However there are a small differences.

When a *ServerConnection* receives a scene from a client, it creates a *ServerRenderer* which creates a buffer. This buffer is used to hold *ImageLocation* objects, which represents coordinates to be calculated. Whenever the *ServerConnection* receives a pixel it puts it in this buffer. The *ServerRenderer* can then grab work from this buffer. The *ServerRenderer* works like the rendering part in Raja, except that it gets its *ImageLocation*s from a buffer and instead of writing to a picture it sends the resulting pixel back to the client. Note that a connection has a renderer if it has received a scene, and that a *ServerRenderer* always have a connection attached to it. This one to one relation between a connection and a renderer yields a simpler design than Fraja, since it does not have to determine where the received objects comes from in order to handle them accordingly.

3.7 Fraja versus Straja

Even though Fraja and Straja have many similarities, there are still noticeable differences. The most obvious is that Straja lacks many of its counterpart's functionalities. It has no master server or automatic discovery of

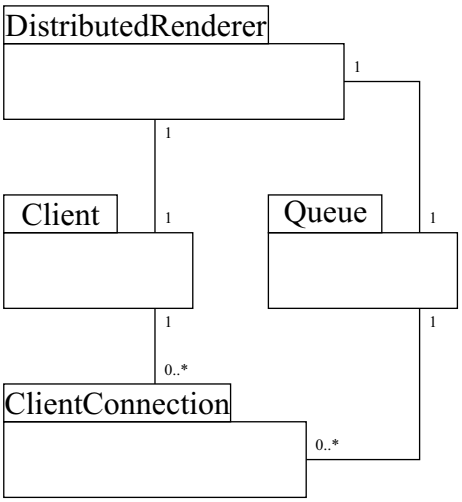

Figure 3.8: Class diagram of Straja client.

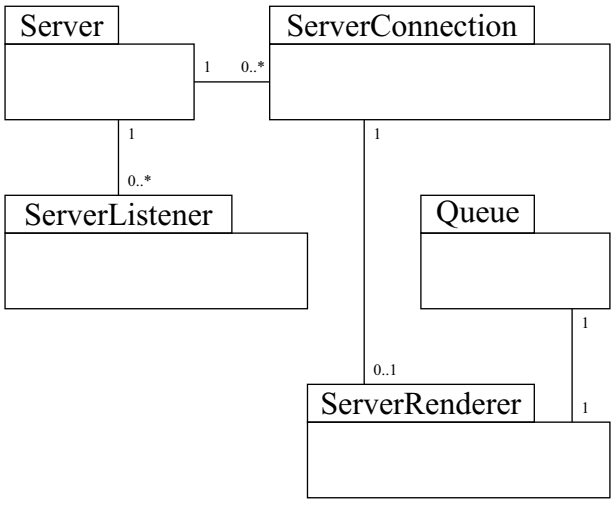

Figure 3.9: Class diagram of Straja server.

servers of any kind, which makes Straja inflexible. Furthermore Straja lacks the functionality of recovering from a server crash.

This difference in the implementations origins from the two different approaches for connectivity. Since Straja uses static connections its server knows which renderer an incoming object belongs to. On the other hand Fraja must search through its list of clients every time.

Finally Straja differs in the way it schedules pixels, as it uses Raja's schedulers instead of implementing its own, optimized for distribution, as Fraja does.

4 Performance analysis

4.1 Benchmark

This chapter consists of four parts. The first part of this section focuses on the optimizations of the pixel collection size, which is the only variable parameter in the code. In the second part we identify the bottlenecks of the system. Next the program is evaluated for scalability and finally we benchmark the connectivity Fraja versus Straja.

4.1.1 Benchmarking environment

During the benchmark as many parameters as possible are kept constant, e.g. the host on which the servers, master server and client runs. All servers are run on the university's blade machines since they are identical. Configurations for the default servers, the client and master servers can be seen in table 4.1 on the next page. The blade machines are publicly available, thus we cannot prevent CPU load caused by external sources e.g. other users launching applications on the machine.

Pictures used for rendering during the benchmark are "Cone" and "OverlappingSpheres" see figure 4.1. "Cone" is a picture with very low complexity evenly distributed in the picture, whereas "OverlappingSpheres" has a very high complexity unevenly distributed. The high complexity is due to the fact that "OverlappingSpheres" consists of several overlapping glass spheres,

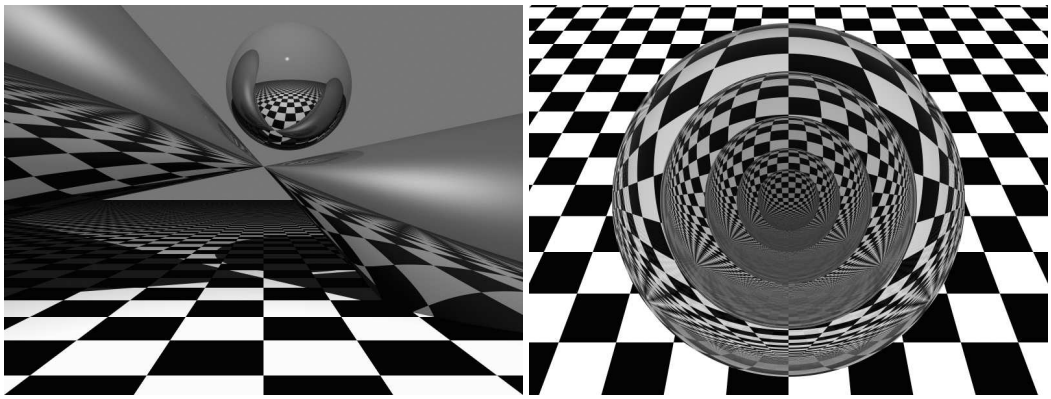

Figure 4.1: "Cone" to the left and "OverlappingSpheres" to the right.

	Client	**Server(s)**	**MasterServer**
CPU	Mobile Celeron 1066 MHz	Ultra Sparc 2 500 MHz	Ultra Sparc 2 2×300 MHz
RAM	256 MB SDRAM	512 MB SDRAM	512 MB SDRAM
NIC	100 Mbit	100 Mbit	100 Mbit
OS	Red Hat Linux 7.3	SUN Solaris 2.8	SUN Solaris 2.8
JAVA	SUN Java 2SE 1.4	SUN Java 2SE 1.4	SUN Java 2SE 1.4

Table 4.1: Hardware and software used for benchmark.

yielding a massive amount reflections for each ray. Furthermore this section will use the word benchmark to denote a single serie of results e.g. table 4.2 and figure 4.2 on the facing page is a benchmark.

4.1.2 Optimizing pixel collection size

In this section we will analyze the influence of pixel collection size on the computation time. The test for the optimal pixel collection size uses the simple cone picture, with no antialiasing, as this represents our "worst case" scenario. Antialiasing in Raja is done by tracing more rays through the same pixel and calculating the average color from these. This means that whenever the client sends a pixel to a server, the amount of antialiasing will have a direct effect on how much time pases before the pixels is sent back, thus high antialiasing will spread the network load over a greater amount of time. As expected the computation time achieved with a collection size of 1 is very poor indeed. This is due to the fact that the server can compute a pixel before it receives an extra pixel for its buffer. In short the servers have much

Client:	Default	Picture:	Cone
# Server:	1	Size:	640×480
Master:	Default	Antialiasing:	1 (none)
Collection size:	Varied	Depth[1]:	500 (full)

Collection size	1	4	8	16	20	24	32	64
Time	2910	280	147	110	105	104	103	104

Table 4.2: The configuration and results of the pixel collection benchmark.

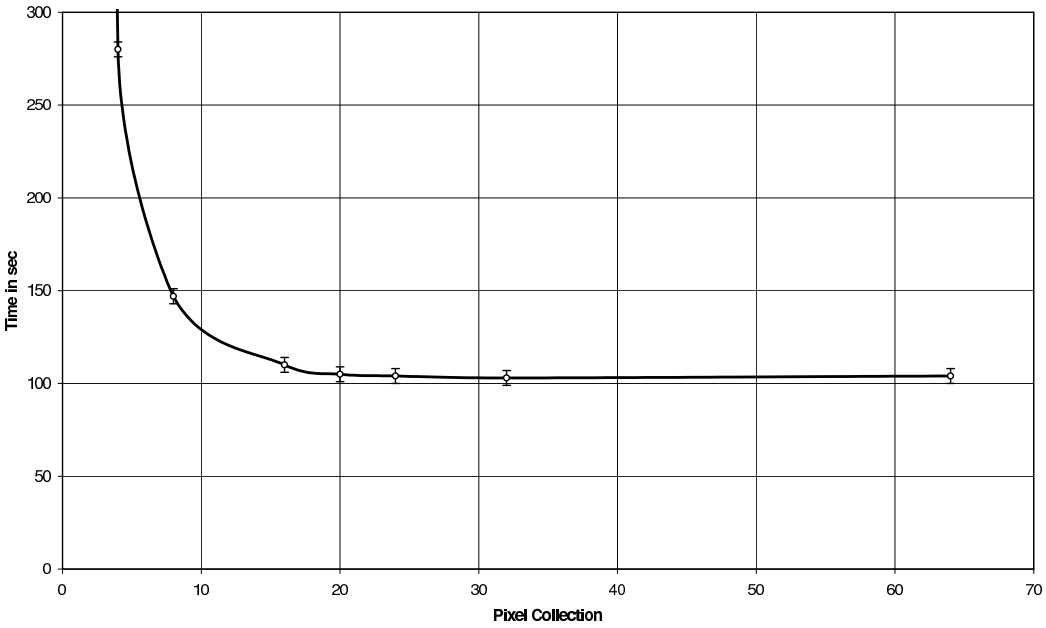

Figure 4.2: The relation between the pixel collection size and the computation time with a picture.

idle time.

The first benchmark is performed with the parameters seen in table 4.2. As can be seen from figure 4.2 the computation time do not improve beyond a pixel collection size of approximately 24. This denotes the point where a server cannot complete the computation of its first pixel collection before it receives a new one, i.e. the server has no idle CPU time and is therefore used optimally.

Two additional benchmarks have been performed to certify that the collection size also is optimal for an increasing number of servers and that increasing the collection size (in this case to 64) does not make it scale better. The reason for

Client:	Default	Picture:	Cone
# Server:	Varied	Size:	640×480
Master:	Default	Antialiasing:	1 (none)
Collection size:	24	Depth:	500 (full)

Servers	1	2	4	8	12	16
Time	104	54	29	17	16	17

Table 4.3: The configuration and results of the pixel collection benchmark.

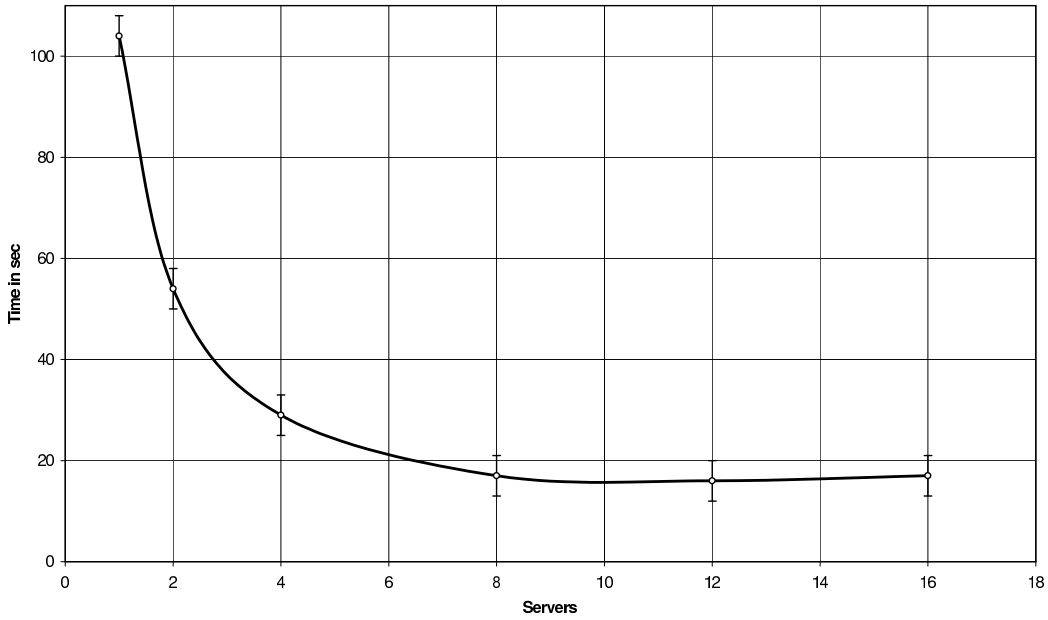

Figure 4.3: The relation between the number of servers and the computation time of a picture using a pixel collection size of 24.

increasing the collection size in the latter test is, that an increased collection size will cause fewer connection to be made during the computation of the picture and could give a conceivably better scalability. The configurations and results can be found in table 4.3 and 4.4 on the next page. If figures 4.3 and 4.4 are compared, it becomes clear that neither of the configurations scale beyond the 8th server. Furthermore they use the same amount of time to render the picture (within a small margin of seconds). These facts lead to the conclusion that the pixel size is optimal.

Client:	Default	Picture:	Cone
# Server:	Varied	Size:	640×480
Master:	Default	antialiasing:	1 (none)
Collection size:	64	Depth:	500 (full)

Servers	1	2	4	8	12	16
Time	102	54	29	18	17	17

Table 4.4: The configuration and results of the pixel collection benchmark.

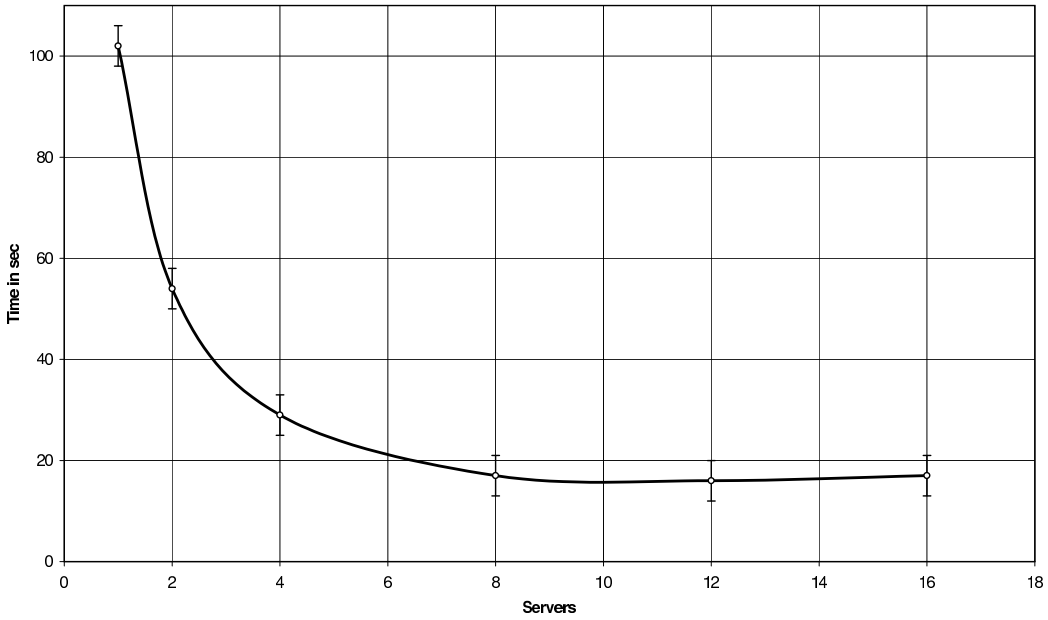

Figure 4.4: The relation between the number of servers and the computation time of a picture using a pixel collection size of 64.

4.1.3 Bottlenecks

In this section we will try to identify the bottlenecks in the system. On figure 4.3 on the facing page the time stops decreasing around 8 servers, since this is the place where the bottleneck sets the limitation, it is a good configuration to perform the measurements on.

The first benchmark measures the amount of RAM used on the client: Fraja uses 27 MB RAM when it is started and when the rendering begins it increases to 41 MB. A client with more than 41 MB free memory will be a bottleneck.

The second benchmark measures the network traffic on the client. On fig-

ure 4.5 on the next page the network traffic on the client is illustrated. The traffic hovers around 1 MB/s, which should not be a problem on a 100 Mbit switched LAN and it is therefore not a bottleneck with our configuration.

The third and last benchmark measures the CPU utilization on the client. On figure 4.6 on the facing page the clients CPU utilization in percent is illustrated. The CPU utilization peaks at 100% after 3 seconds and then slips back to 50% for a couple of seconds, then it again climbs to around 100% until the test finishes. The reason for the drop in CPU utilization three seconds from the start, is that after the client has sent a pixel collection to all the servers, it takes a bit time before the results begin to return from the servers. This is clearly the bottleneck in CPU utilization configuration.

The conclusion is that with the current client machine, it can scale to around 8 servers. With a faster CPU it should be possible to scale to a larger number. It should be noted that eight servers is the worst case, since there is no antialiasing and the "Cone" picture is a simple picture. With an antialiasing of 16, each pixel would be 256 times more complex to calculate. This means that in theory the client machine would be able to scale to $8 \times 256 = 1024$ servers and no other bottlenecks kicks in.

4.1.4 Scalability

In this section we will analyze how Fraja scales with variable number of servers. We have performed benchmarks with antialiasing level 1, 4 and 8, but not 16. Even though the antialiasing 16 configuration would be the one with the best scaling we have not performed the benchmark as it would require a large amount of time.

The configuration and results for the tests can be seen in table 4.5 and figure 4.6 on page 61. It is interesting to see that both antialiasing 1 and antialiasing 4 reach minimum score around 17 seconds. Antialiasing 8 will probably also reach 17 second with 128 blades. Unfortunately we only have 48 blades at our disposal. The reason that they all can reach 17 seconds is that the client is the bottleneck in the system. The tests are performed with the same resolution, so the client has to send the same amount of pixel collections in the test, it just requires a larger amount of servers to perform the calculations.

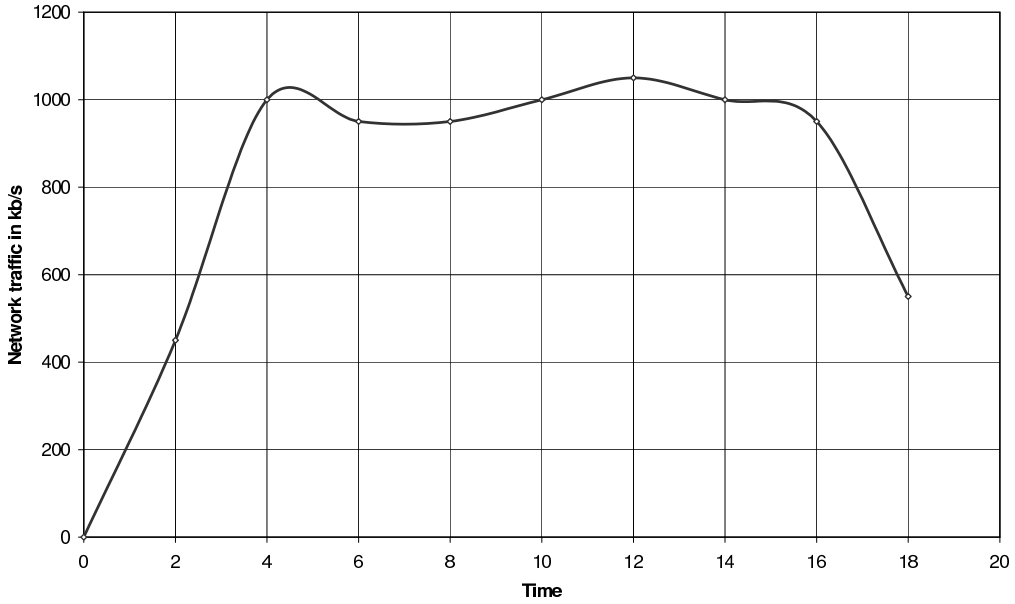

Figure 4.5: The graph over network traffic on the client during the benchmark with a pixel collection size 24 and 8 servers.

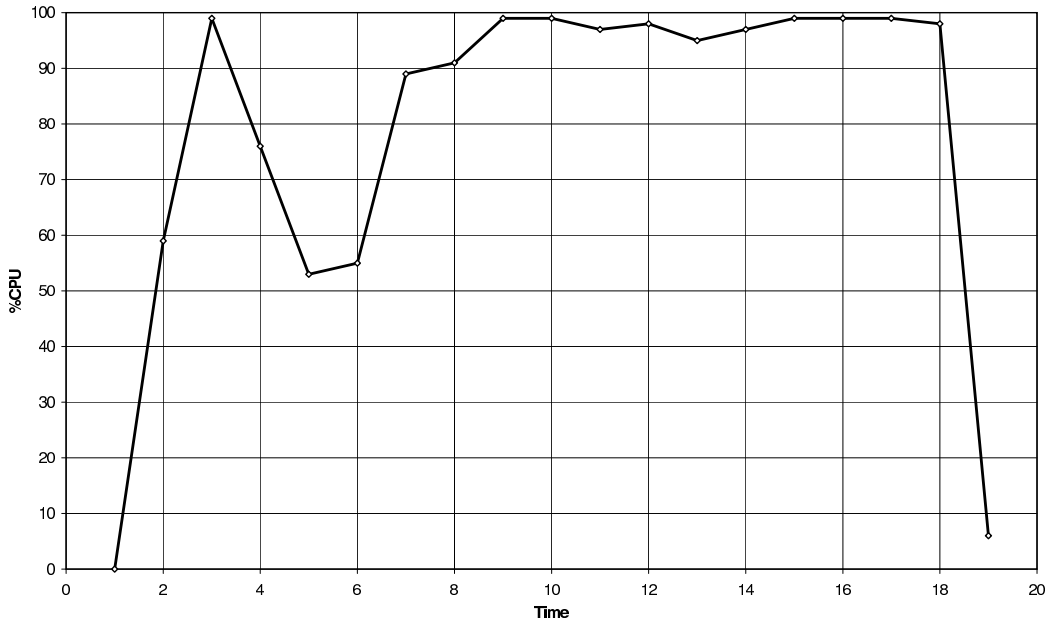

Figure 4.6: The graph over CPU utilization on the client during the benchmark with a pixel collection size of 24 and 8 servers.

Utilization

Table 4.7 on page 62 displays the average utilization of the servers in the test. The utilization for the test with one server is set to 100%. The utilization is calculated with this fraction:

$$\frac{\text{time for the test with one server}}{(\#\text{ servers in this test}) \times (\text{time for this test})} \times 100\%$$

The combined utilization of the servers in a test seen in relation to the base case with one server, can be calculated by removing the (# servers in the test) factor in the denominator from the above fraction. E.g. 8 servers with antialiasing 1: $8 \times 76.5\% = 612\%$, the numbers are from table 4.3 on page 56.

On table 4.8 on page 62 it can be seen that the average utilization drop faster in the tests with a lower antialiasing. The reason for this is that it is easier to feed the servers, when each pixel collection contains a larger workload, but does not increase the workload for the client or increase the network traffic.

Table 4.9 on page 63 shows the utilization for each server in the new batch. The utilization is calculated with this fraction:

$$\frac{(\text{this \# servers} \times \text{this avr. util.}) - (\text{prv. batch's \# servers} \times \text{prv. batch's avr. util.})}{(\text{this \# servers}) - (\text{previous batch's \# servers})}$$

On table 4.10 on page 63 it can be seen that the utilization for the new servers drop faster in the tests with a lower antialiasing. The explanation is the same as with the average utilization, it is just more explicit when focus is on the new batch of servers.

When the new servers utilization drops to 0%, the new servers does not make the computation any faster. The reason why the lines are not straight is due to the measure deviation on 4 seconds in the test.

The conclusion is that the higher antialiasing the better Fraja scale. Therefore Fraja is best suited for rendering larger picture with a high antialiasing, this in fact the pictures most people would want to distribute, because the large amount of time these picture takes to render on a normal computer.

Showoff

As a final test we have rendered "OverlappingSpheres" in 1600×1200 with antialiasing 16 and depth 500 (full). Fraja took approximately 2 hours with

Client:	Default	Picture:	Cone
Master:	Default	Size:	640×480
# Server:	Varied	antialiasing:	Varied
Collection size:	24	Depth:	500 (full)

Servers	1	2	4	8	12	16	24	32	40	48
Antialiasing 1	104	54	29	17	16	17				
Antialiasing 4	360	181	73	49		27	19	17	17	
Antialiasing 8	1140	588	299	150		80		43		31

Table 4.5: The configuration and results of the cone benchmark with different levels of antialiasing

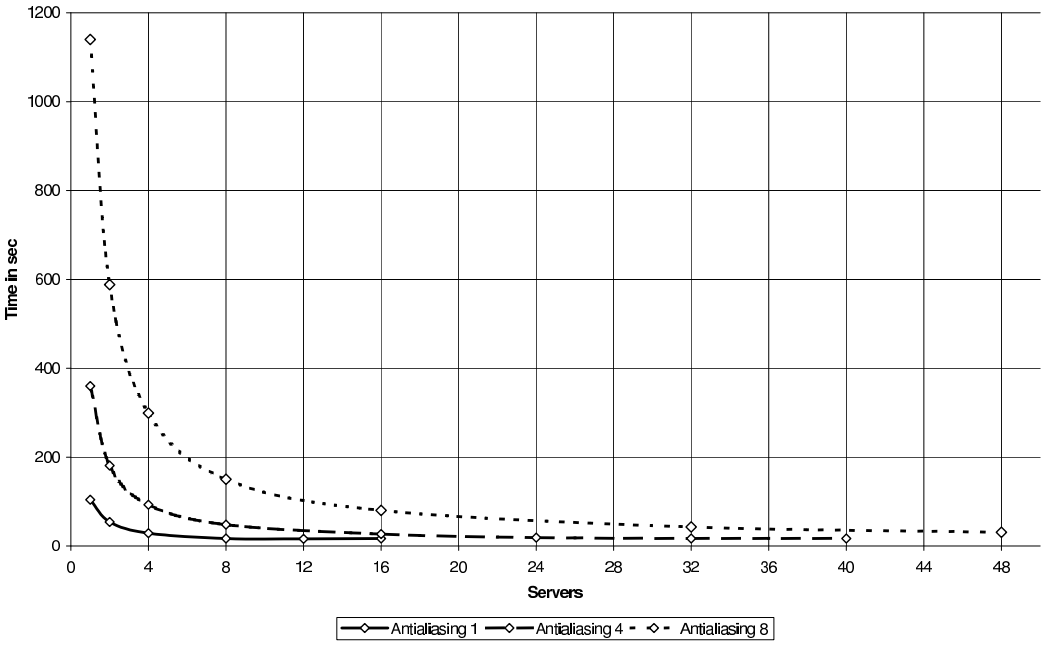

Table 4.6: Graph showing scalability with varying levels of antialiasing.

Servers	1	2	4	8	12	16	24	32	40	48
Antialiasing 1	100	96,3	89,7	76,5	54,2	38,2				
Antialiasing 4	100	99,5	96,8	95,8		83,3	79,0	66,2	52,9	
Antialiasing 8	100	96,9	95,3	95		89,1		82,8		76,6

Table 4.7: The average utilization for the servers.

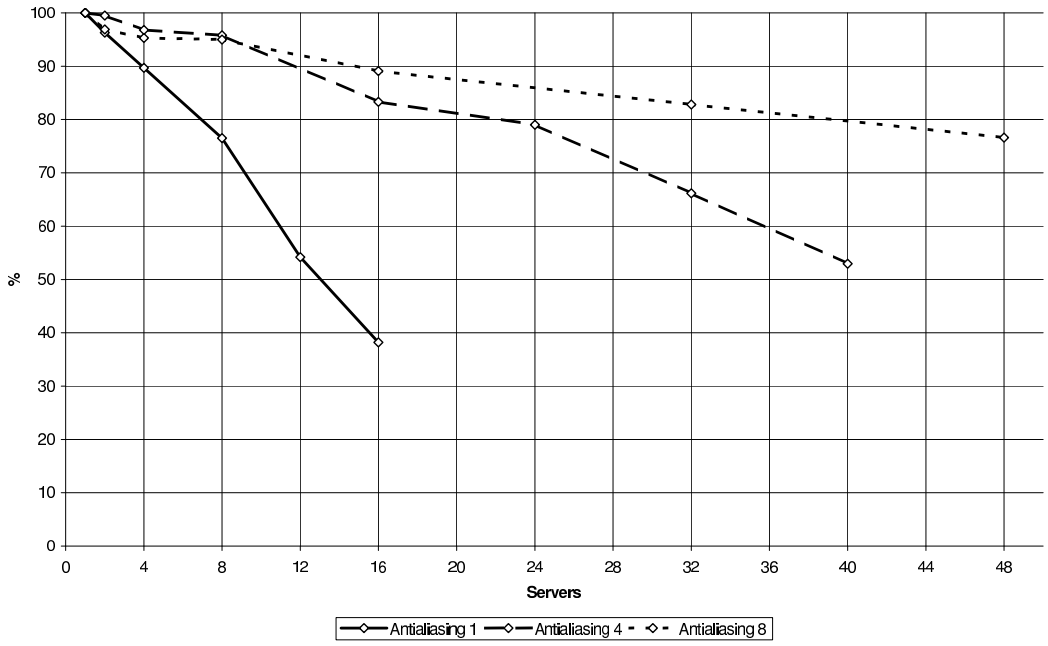

Table 4.8: Graph showing the average utilization for the servers.

Servers	1	2	4	8	12	16	24	32	40	48
Antialiasing 1	100	92,6	83,1	63,3	9,6	0				
Antialiasing 4	100	98,9	94,0	94,9		70,8	70,2	22,3	0	
Antialiasing 8	100	93,8	93,7	94,7		83,2		76,5		64,2

Table 4.9: The utilization pr. server in the new batch.

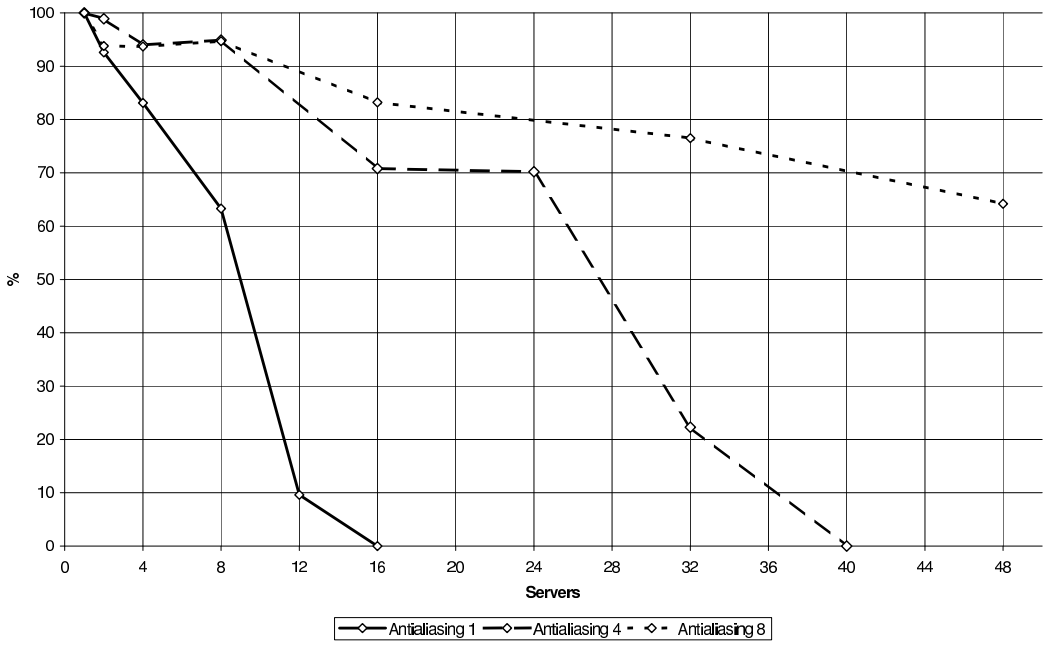

Table 4.10: Graph showing the utilization pr. server in the new batch.

Machine:	Server	Application:	Raja
Picture:	OverlappingSphere	Depth:	500 (full)

Resolution	Antialiasing	Time
400×300	1	103
400×300	2	398
400×300	4	1541
1600×1200	1	1678

Table 4.11: The result for the various benchmark used to estimate the 1600×1200 antialiasing 16.

48 default servers and 44 computers with different configurations. The hardware platforms where Intel, AMD and Sparc and the operating systems used where Solaris 2.8, Red Hat Linux 7.3, Windows 2000 and Windows XP. Besides being a showoff test, this also shows the capability of Fraja running on various platforms, but the different configurations makes this test non scientific.

The time with the normal Raja is only an estimate, as we did not have the time to perform this benchmark. Instead we performed a set of benchmarks used to calculate the time for Raja. The results can be seen in table 4.11. First we calculate the "penalty" for increasing the antialiasing to the double. For 400×300 from antialiasing 1 to antialiasing 2 the penalty is $\frac{398}{103} \times 100\% = 386\%$, for 400×300 from antialiasing 2 to antialiasing 4 the penalty is $\frac{1541}{398} \times 100\% = 387\%$, so 386.5% will be used as penalty. This penalty is multiplied four times with the time for 1600×1200 with antialiasing 1 to find the estimate for 1600×1200 antialiasing 16: $1678 \times 386.5\%^4 = 375598$ seconds. The estimate for Raja is or 4 days and 9 hours versus Fraja's approximately 2 hours.

4.1.5 Connectivity

We will here look at how Fraja and Straja compares to each other when rendering.

To benchmark the two applications we choose to use "Cone" with antialiasing level 1 and 8. To see how the two implementations scale compared to each other, we benchmarked with a different amount of servers. The result of these benchmarks are shown on graph 4.13 on page 66. Even though it does not appear like there are four lines on the graph, they are present,

but they overlay each other. There are, of course small differences, between the two implementations. This is also clear when reading the numbers for the benchmark on table 4.12. The difference between the benchmarks is so small that they have to be considered as deviation, since we do not have exact control over the machines used. There is a small difference between the benchmark with antialiasing level 8 and one server. However since this is just with one server, a job running on this machine could influence this greatly.

The conclusion is that static and on demand connections perform and scale equally well, and that it performance wise does not make a difference, when having our kind of network traffic.

Servers	1	2	4	8	12	16	32	48
Fraja Antialiasing 1	104	54	29	17	16	17		
Straja Antialiasing 1	109	53	29	19	17	18		
Fraja Antialiasing 8	1140	588	299	150		80	43	31
Straja Antialiasing 8	1170	583	293	152		81	45	32

Table 4.12: Benchmarks from Straja.

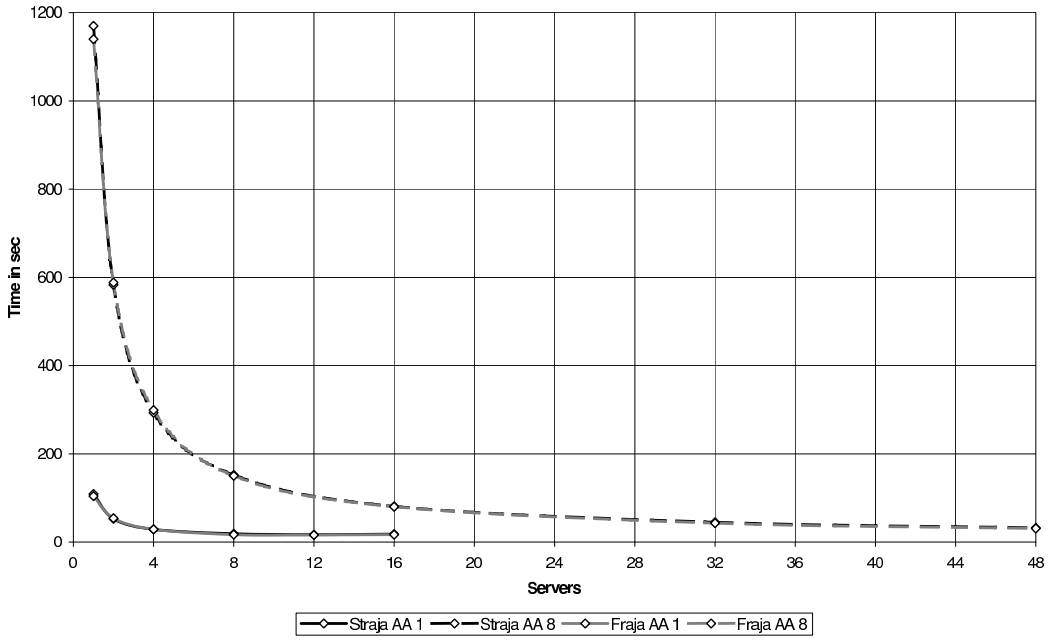

Table 4.13: Graph showing benchmarks from Straja.

5 Conclusion

5.1 Fraja

Throughout the past four months we have managed to develop a robust distributed ray-tracing rendering system that scales well for complicated pictures.

The servers of Fraja support the use of multiple clients on different hosts and the client can survive the crash of arbitrary servers. The part of the code, that handles master server crashes has not been integrated into the final code, because it has not been fully tested.

Furthermore Fraja's scheduler has been optimized to generally run in constant time and worst case linear time. Fraja is implemented in Java and as such runs on multiple operating systems and architectures e.g. the client can run on Linux, while the master server runs on Windows and the servers on Solaris.

5.2 Connectivity

Two kinds of connectivity have been implemented. One that creates them on demand, which is the main implementation, and one that does it static, which has been a side implementation for testing purposes.

An advantage of having two different implementations was that while one team started by making functionality the other focused on the integration in Raja. The knowledge about integration in Raja from the second implementation was to great help during the integration of the primary program.

5.3 Performance analysis

The bottleneck of Fraja is the client, which gets flooded with pixel collections if the servers get pixel collections of too small a size. This makes the client update the picture on the screen and find new work for the server, which has just returned a computed pixel collection. Also the operating system is using more resources due to higher network load.

The conclusion is that the performance of the client decreases if it is flooded with packets. To avoid this, the amount of servers must be a function of the picture complexity. The level of antialiasing is usually the most significant. For example the picture ”OverlappingSpheres” will flood the client in the beginning and end of the calculations because the chess floor is very easy to calculate. When the calculations strike the glass spheres they get so heavy that the client has no problem keeping up. The load on the servers is determined by two things; the work size and the level of antialiasing. If the level of antialiasing is high, say 16, each pixel is $16^2 = 256$ times as hard to calculate the color of than a pixel with no antialiasing.

Also an analysis of Fraja and Straja was made, with the purpose to see which performed the best. However in our test no noticeable differences were found as the two implementations performed very close to identical.

5.4 Further development

Despite the extensive implementation of Fraja, there are still some convenient features that Fraja currently lacks.

First and foremost the implementation of session IDs would allow a client to stop rendering a picture half way and start a new one without the need of restarting the servers. The need for restarting the servers arises from the fact that some servers might still be rendering parts of the picture which they might return to a client rendering a different picture. This would result in a mix of the two pictures in issue.

Secondly Fraja should be modified so that Raja does not need to be restarted after every rendering. This is due to some references and sockets that are not currently killed causing system failures when starting a second rendering.

Also for some pictures it is not necessary to use a lot of servers. It should therefore be possible to either manually set a limit to the number of servers or have the program choosing the appropriate number of servers for you.

Furthermore, the server should be able to split its work into a number of independent threads, thus making it exploit multiple processors.

Lastly the GUI for Raja does not support Fraja at present. This should be relatively easy to implement and would be nice for the total integration in Raja.

A List of figures and bibliography

List of Figures

List of Tables

Bibliography

[1] Alan Chalmers and Erik Reinhard. *Parallel and Distributed Photo-Realistic Rendering.* http://www.cs.bris.ac.uk/Tools/Reports/Ps/1998-chalmers.pdf, 27-11-02.

[2] Lars Mathiassen et. al. *Object Oriented Analysis & Design.* Makro ApS, 2000. ISBN: 87-7751-150-6.

[3] Anthony Steed Mel Slater and Yiorgos Chrysanthou. *Computer Graphics and Virtual Environments from Realism to Real-Time.* Addison-Wesley, 2001. ISBN: 0-201-62420-6.

[4] Paul Rademacher. *Ray Tracing: Graphics for the Masses.* http://info.acm.org/crossroads/xrds3-4/raytracing.html, 25-11-02.