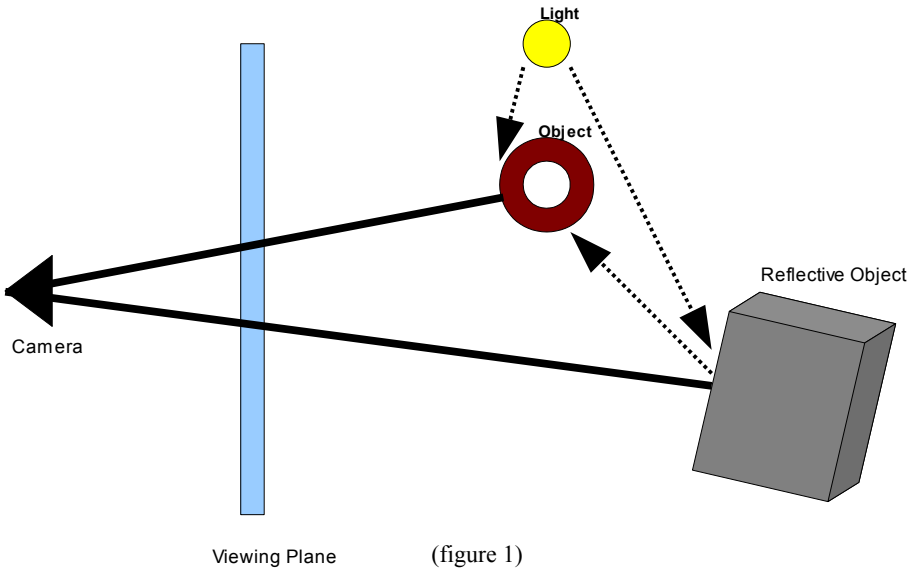
A Simple Distributed Raytracer

Jason Rickwald
CSC 569-01, Fall 2005
Final Project

1. Project Description

Raytracing is a technique used to render a three-dimensional scene on a computer. When implemented well, a raytracer can often render scenes with near photorealistic quality. However, raytracers are inherently slow because of how they work [4]. For each pixel in a rendered scene, a ray must be sent out into the world. If the ray hits an object, a lighting calculation is made using the color attributes, texture attributes, reflection attributes and refraction attributes for that object. Both reflections and refractions involve sending out another ray from the point that was hit on the object. Again, if this ray hits an object, then another lighting calculation is made. This continues until no object is hit, in which case a background color is used. A sketch of this process is shown in figure 1.



(figure 1)

The process may be very time consuming if there are many pixels to be rendered. The good news is that pixels are not dependent on each other. Therefore, the work can be split up safely among different raytracers. The objective of this project is to distribute the work of raytracing across multiple computers in order to shorten the time it takes to render a scene.

2. Requirements

From the start, I knew that I would use the raytracer code that I wrote when taking CSC-473 in the Fall of 2004 [4]. With the exception of the modifications necessary for adapting the code to a distributed application, I had no intention of adding new features to that raytracer. Therefore, this project only sets out to satisfy a few basic requirements in regards to the raytracer:

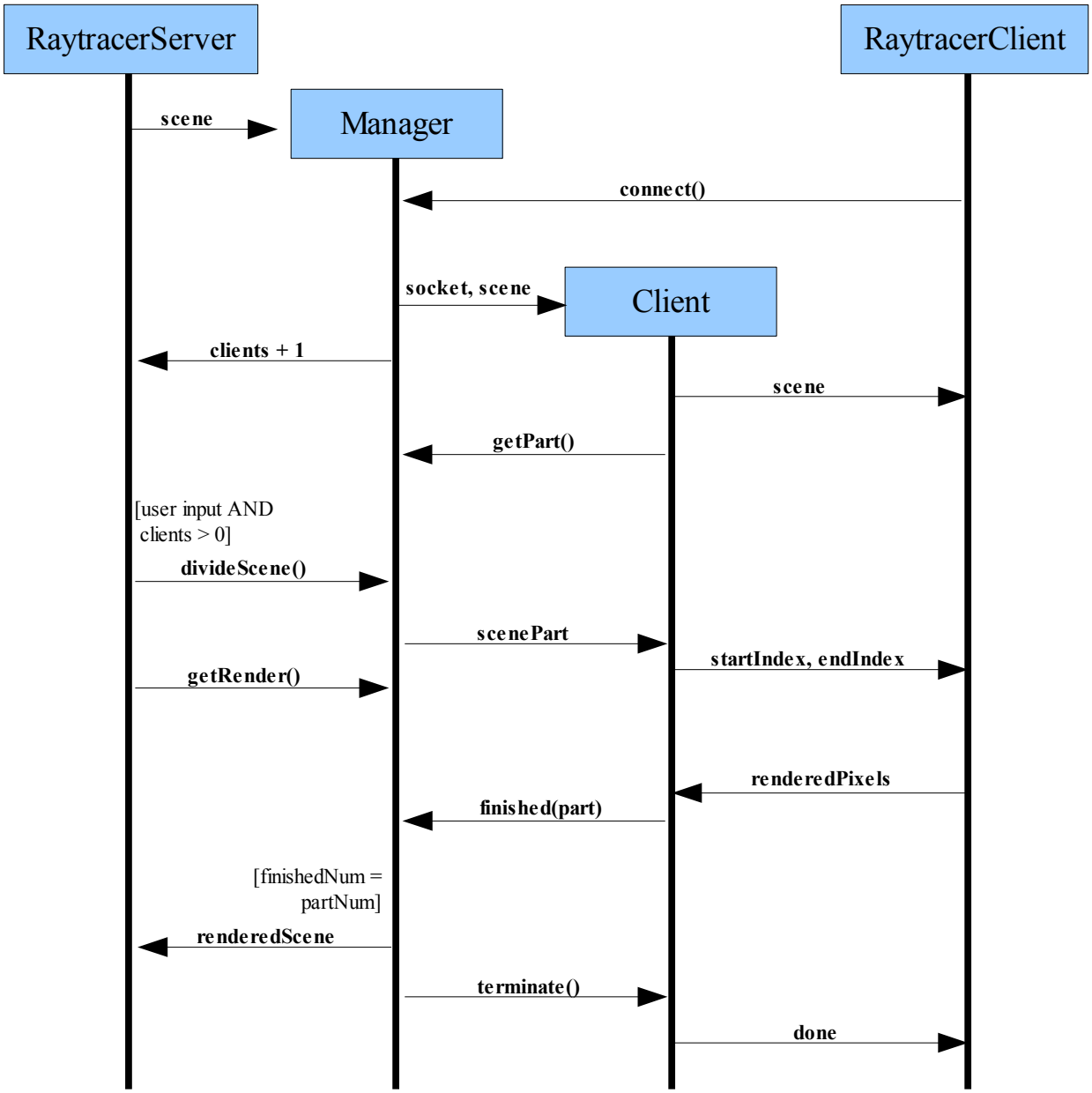
1. The raytracer will render only static scenes.
2. The raytracer will render primitive solids such as spheres, cubes, cones, and toruses.
3. The raytracer will render simple polygonal meshes (defined as a series of vertex triples).
4. The raytracer will render boolean combinations of solids.
5. The raytracer will render reflections.
6. The raytracer will render refractions.
7. The raytracer will allow the artist to assign textures to solids.
8. The raytracer will allow the artist to assign normal maps to solids.
9. The raytracer will render soft shadows.
10. The raytracer will use multisampling for antialiasing of the rendered scene.
11. The raytracer will allow the artist to specify the width and height of the render.

The requirements related to the distribution of the raytracer are:

12. The artist will be able to add any number of computers to the system to expedite the render process.
13. The artist will still be able to render a scene using just one computer.
14. Any rendering machine (client) may fail without affecting the rendering job as long as:
    a) The machine that goes down isn't also the machine that is hosting this render job (the server).
    b) At least one client remains connected and can render.
15. The server will only send a minimal amount of data to the clients to avoid wasting network bandwidth and time.
16. The server will attempt to give all clients an equal share of rendering work. This doesn't mean giving all clients the same number of pixels to render. Some parts of the scene may contain many reflections, refractions, and other complex computations that result in more work per pixel for the client rendering that part. The server will try to detect this and adjust workloads.
17. The user should notice a decrease in the rendering time of a scene as compared to the single-machine single-processor version of the raytracer when rendering the same scene. This decrease will obviously not exist when there is only one rendering client, as the overhead added in the distributed version will actually slow rendering. However, the user should see a decrease in rendering time with three or more clients on the same local area network.

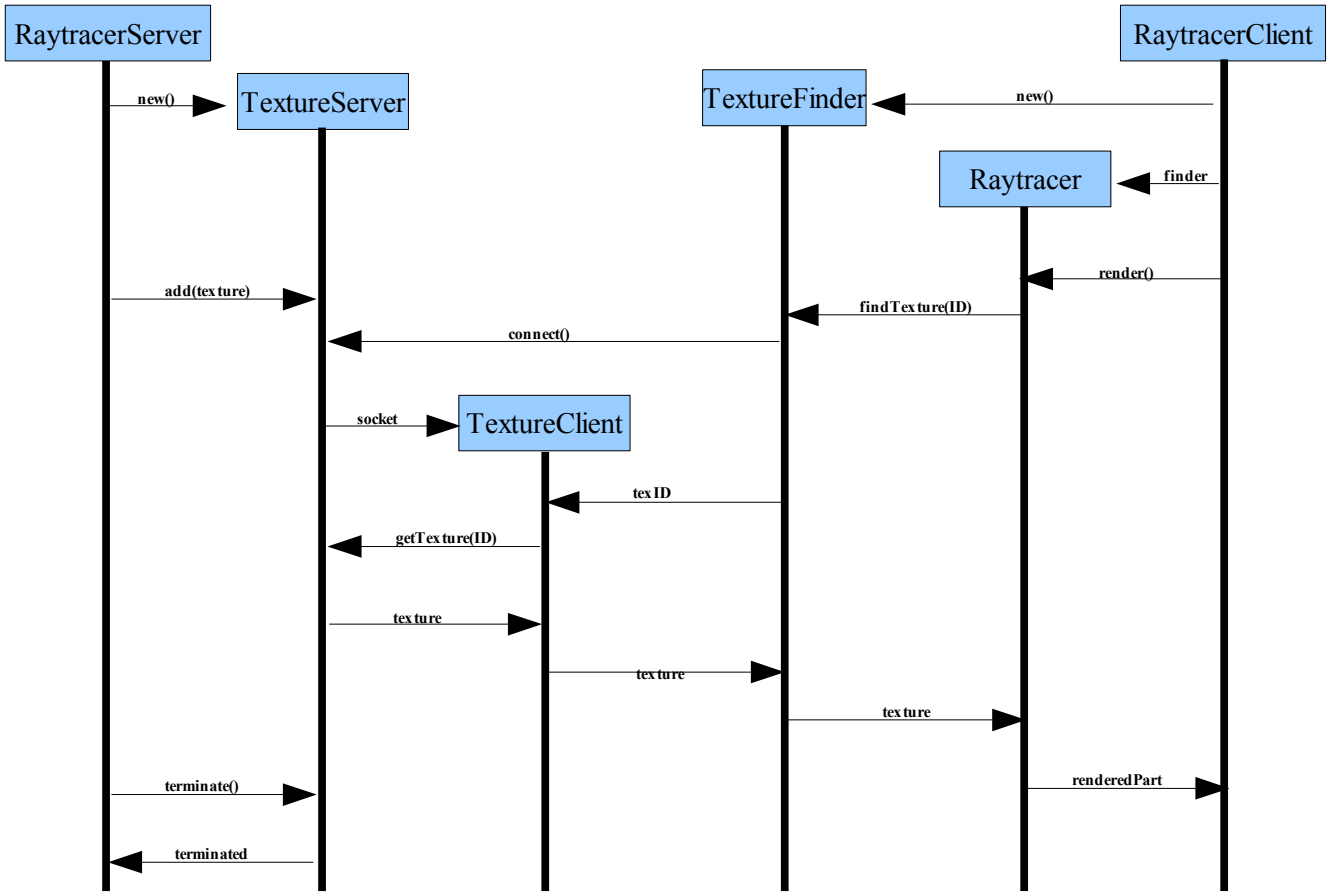
3. Design

At its most basic, the distributed raytracer functions much like the gender-detecting distributed word counter used for CSC-569 programming assignments [1] and described in CSC-569 lecture [2]. A client-server design pattern is used [2]. The server hosts the work to be done – a scene to render, along with the width and height of the render. Clients connect to the server to make requests for work to do and return with the raytraced pixels. The server uses a master-slave design pattern [2]. Upon client connection it creates a slave thread to handle communications with that client. A sequence diagram for this basic design is shown in figure 2. This sequence diagram will be described in more detail in the implementation section.



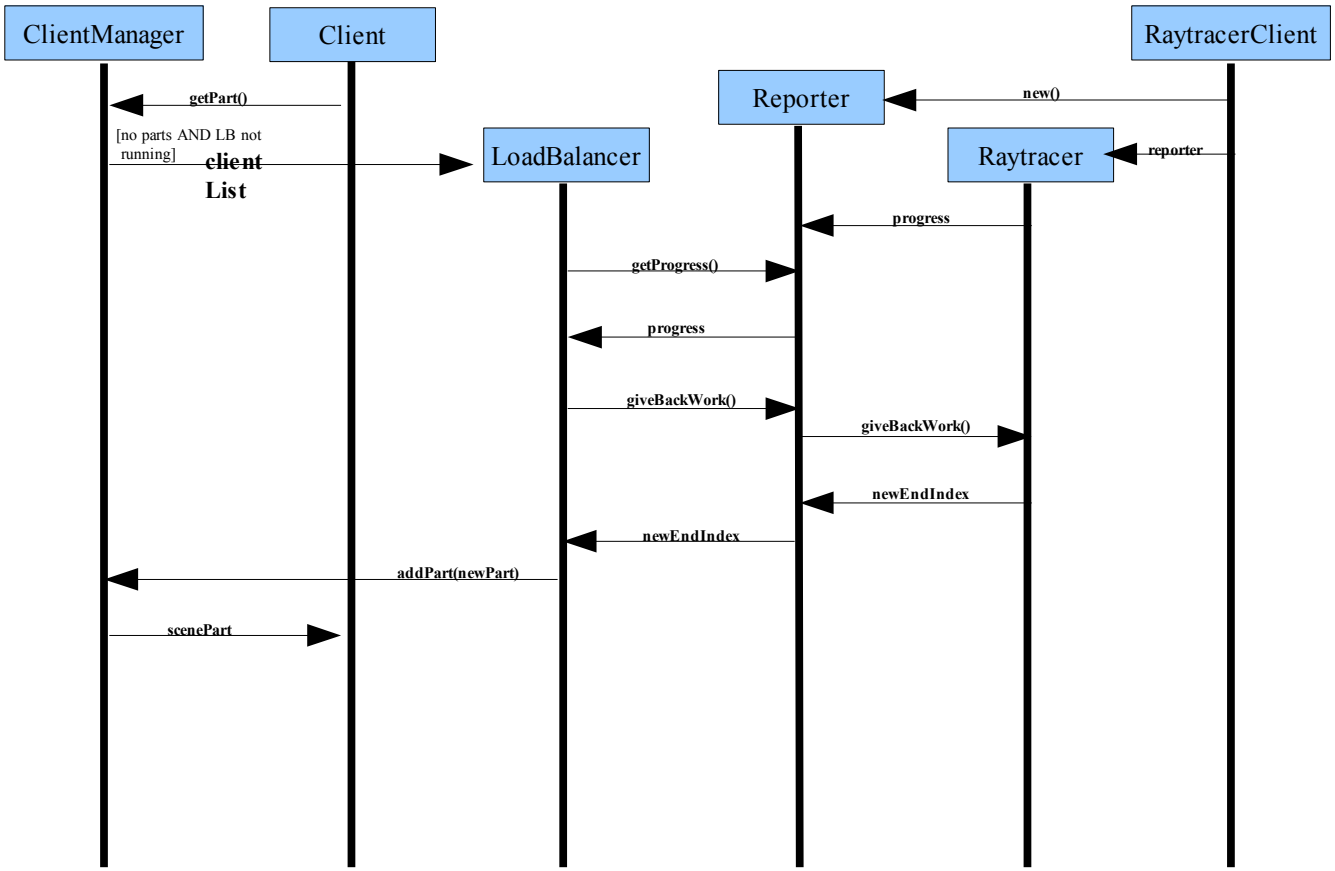
(figure 2)

Next, we can get away with sending less scene data over the network – an enhancement that will satisfy requirement 15. This is done by not sending the whole scene to every client that connects. Instead, only the structural data is sent, and all textures and normal maps are withheld. Then, when a client finds that it needs one of the missing textures or normal maps, it requests that texture or normal map from the server. This works to reduce scene data sent because every client needs the whole scene structure, but clients don't need every texture and normal map. They only need the textures and normal maps that are required for rendering their portion of the scene. This is another utilization of the client-server design pattern and the master-slave design pattern [2]. Usage of these patterns is the same as previously described. A sequence diagram of the enhancement is shown in figure 3. This diagram only shows additions, and it will be discussed in depth in the implementation section.



(figure 3)

Another enhancement, one that satisfies requirement 16, is balancing rendering loads between clients. As mentioned in the description of that requirement, not every client gets an equal amount of rendering work, as rendering work isn't based solely on number of pixels to render. For example, one client might get a chunk of pixels to render that consists mostly of background, and another may get a chunk that contains many solids with complex refractions. In this case, the first client will finish long before the second, and most of the render time will consist of waiting for the slow client. This is dealt with in the following way. When a client needs work (either it is just starting or it has just finished), it requests that work from the server. If the server already has a chunk of render work for it to do, then it simply gives that work to the client. However, if the server doesn't have any work for the client to do, then it attempts to find a rendering client that is in the most need of help (how this is determined is described in the implementation section). If the server finds such a client, then it requests that the client put some of its remaining work back. Once the server gets this work back, it gives it to a client that is waiting for work to do. A sequence diagram for this is shown in figure 4. This sequence diagram only shows additions to the previous two, and will be described in the implementation section.



(figure 4)

A further enhancement to the distributed raytracer application would add a proxy [2]. The proxy would act as a third party between the server and the client. In this way, the clients would not have to know about a particular server, but instead would know the proxy. Many servers could then connect to the proxy with work to do. Clients requesting work would go to the proxy, and the proxy would pass that request on to any of the servers. This technique would allow any server with work to do to get that work done without the need to set up clients specific to that server. Also, it could allow for a further degree of load balancing in that the proxy could take steps to give clients to the server that is in most need of clients (it has the most work to do). Unfortunately, this enhancement was not implemented for this project.

4. Implementation

As mentioned before, the raytracing code was already available from a CSC-473 project. I tried to modify that code as little as possible, but some changes were necessary. First, the original raytracer would only render an entire scene. It was modified to take start and end indexes and render only those pixels between. Also, all classes used in representing a scene had to be made serializable, so that the scene could be sent to render clients. Finally, all of the raytracing code was moved into a "raytracer" Java package, and the distributed application code was put into an "application" package. Now the basic implementation of the distributed raytracer, shown in figure 2, could be written. Both `RaytracerClient` and `RaytracerServer` are Java applications. Both applications use plain Java stream sockets [3] to do their communication. The `RaytracerServer` takes the dimensions of

the image to render from the command line arguments. Ideally, it would also take the name of the scene file to render. However, I did not have time to implement scene files, so the scene is currently hard-coded into the `RaytracerServer`'s `main()` method. The `RaytracerClient` takes the host name/address of the machine running the server.

The first thing that the `RaytracerServer` does is to create and start the `ClientManager` thread. The client manager listens on port 4815 for incoming client connections. It maintains synchronized lists of connected clients, a list of `SceneParts` to be given to clients (pixels to render), and a list of `SceneParts` that were rendered. All lists are initially empty. The `ClientManager` is also given a copy of the scene when created so that the scene can be sent to clients after their connection. The `ClientManager`'s `run()` method is shown in figure 5.

```
public void run ()
{
    while (getAcceptConnections())
    {
        try
        {
            Socket clientSocket = serverSocket.accept();
            Client client = new Client(clientSocket, this, scene);
            client.start();
            System.out.println("Client ready for render: " + clientSocket.getInetAddress().toString());
            synchronized (this)
            {
                clients.add(client);
            }
        }
        catch (SocketException e)
        {
            // server socket forced closed
            setAcceptConnections(false);
        }
        catch (Exception e)
        {
            System.err.println("Error while trying to accept client connections. " + e.toString());
            e.printStackTrace();
            setAcceptConnections(false);
        }
    }

    for (Client client : clients)
    {
        while (client.isAlive())
        {
            client.terminate();
            try
            {
                client.join(5000);
            }
            catch (InterruptedException e)
            {
                // wait some more
            }
        }
        client.close();
    }

    try
    {
        if (!serverSocket.isClosed())
        {
            serverSocket.close();
        }
    }
    catch (IOException e) {}
}
```

(figure 5)

The `ClientManager` will continue to loop waiting for connections until the synchronized boolean flag `acceptConnections` is set to false. This is done when the `RaytracerServer` tells the `ClientManager` to terminate. Also, calling `terminate()` on the `ClientManager` forces a `close()` on the `serverSocket`. This way, if the `ClientManger` thread is blocked on `serverSocket.accept()`, it will throw a `SocketException` [3], which is then caught. Whenever the `ClientManager` gets a new client connection, it creates a new `Client` thread for

handling communications with that client. The final steps of `run()` are to tell all connected clients to terminate as well. This is done in three steps (to account for all clients). First, when the `ClientManager` is told to terminate, there is nothing in the work queue, as the scene has been rendered. The `notifyAll()` method is called on the work queue so that all clients waiting on work are no longer blocked. When a Client gets back `null` work from the `getScenePart()` method, it knows to terminate (let its `run()` method complete). By now, all clients should be done trying to do work; but just in case, the `ClientManager` calls the `terminate()` method on all clients. It then waits up to five seconds for each `Client` thread to complete. Lastly, the `close()` method is called on all `Clients`. This method is shown in figure 6.

```
public void close ()
{
    try
    {
        ObjectOutputStream output = new ObjectOutputStream(socketOutputStream);
        output.writeObject(-1);
        output.flush();
    }
    catch (IOException e)
    {
        System.err.println("Exception while trying to send terminate message to client " + e.getMessage());
        e.printStackTrace();
    }
    finally
    {
        try { socketInputStream.close(); } catch (IOException e) {}
        try { socketOutputStream.close(); } catch (IOException e) {}
        try { socket.close(); } catch (IOException e) {}
    }
}
```

(figure 6)

The `close()` method for a client sends a message to the remote `RaytracerClient` for that client. This message, the `Integer -1`, is interpreted by the `RaytracerClient` to mean that it should exit. Finally, the streams and socket are closed.

After starting the `ClientManager`, the `RaytracerServer` creates the `JFrame` that will display the rendered scene. The `windowClosed()` event for the `JFrame` is tied to terminating the `ClientManager`. The `RaytracerServer` then waits for user input. When the user is satisfied with the number of connected clients, she presses the enter key and the `RaytracerServer` stops blocking. It then calls the `populateWorkload()` method on the `ClientManager`. This method divides the total number of pixels that must be rendered by the number of clients connected. A `ScenePart` with that appropriate number of pixels is then created for each client and put into the work queue. For each `ScenePart` that is added to the work queue, a counter is incremented to keep track of the number of `SceneParts`, and the `notifyAll()` method is called on the work queue so that a `Client` can get that new work. Finally, the `RaytracerServer` waits for the render to complete. This is done by calling the `ClientManager`'s `getClientsRender()` method. This method is shown in figure 7. It utilizes the lock for the `finishedParts` queue to wait until the number of finished parts equals the total number of scene parts. The `notifyAll()` method is called on `finishedParts` every time a completed `ScenePart` is added to it. When all parts have been rendered, this method assembles the pixels into one complete image and returns it. The `RaytracerServer` then displays this image in its `JFrame`.

```
public int[] getClientsRender ()
{
    int[] pixels = new int[scene.W * scene.H];

    synchronized (finishedParts)
    {
        while (finishedParts.size() != getScenePartNum())
        {
            try
            {
                finishedParts.wait();
            }
            catch (InterruptedException e)
            {
                // keep looping
            }
        }
    }

    for (ScenePart part : finishedParts)
    {
        for (int i = part.startIndex; i <= part.endIndex; i++)
        {
            pixels[i] = part.pixels[i - part.startIndex];
        }
    }

    return pixels;
}
```

(figure 7)

When a `Client` thread is first created and started by the `ClientManager`, it is passed the scene that is being rendered. The first thing that the `Client` does, then, is send that scene to its remote `RaytracerClient`. Next, it calls the `getScenePart()` method on the `ClientManager` to get rendering work. If there is no work in the queue, it waits there until work is added (waiting on the work queue's lock). Eventually, it will either get work to do, or it will get `null`. The `null` value, as described above, causes the `Client` thread to terminate. If it gets a `ScenePart`, it sends the start and end indexes to the remote `RaytracerClient` via the socket. It then waits on the socket for the `RaytracerClient` to return the rendered pixel data. When this data is received, it is put into the `ScenePart`, and that part is put into the `finishedParts` queue. If there is an error during either the send or receive, the `Client` thread puts the work back in the work queue and terminates.

The `RaytracerClient` is very simple. It creates a `Raytracer` object that it will use to do the actual work of rendering pixels, and it creates a socket connection to the host that was passed to it on the command line. It then waits on an `ObjectInputStream` read from the socket. When the read occurs, it checks to see if the object is the `Integer -1`, or if it is a `Scene`. If it is `-1`, it terminates gracefully, closing the input streams, closing the socket, and exiting. If it is a `Scene`, it moves into a loop. In this loop, it waits on `ObjectInputStream` reads for the start and end indexes. Again, if it gets a `-1` for either of these indexes, it terminates gracefully. If the indexes are not `-1`, it uses the `Raytracer` to render those pixels, and then sends the rendered data back over the socket.

Next, the enhancement shown in figure 3 was implemented. Again, this enhancement allows less scene data to be sent over the network by only sending clients textures or normal maps when they need it (request it). The `RaytracerServer` does this by first going through the `Scene` that is to be rendered and replacing all instances of `ImageTexture` with `ImageTexturePlaceholder` and all instances of `NormalMap` with `NormalMapPlaceholder`. Each placeholder is given a unique integer ID. The `RaytracerServer` then creates a `WithheldPartManager` thread, passing it a `Hashtable` of IDs to textures and normal maps. This thread creates a server socket on port 4223. When it receives a connection on that socket, it creates a slave thread, the `PartClient` thread, to handle that connection. The `PartClient` doesn't live long. It simply reads the ID number from the

socket, looks up the texture or normal map in the `Hashtable`, and sends the image data for that texture or normal map back over the socket. Once the `WithheldPartManager` is created and running, the `RaytracerServer` can continue as previously described – creating and starting a `ClientManager`, waiting for user input, and waiting for the render.

For this enhancement to work correctly on the client side, the raytracing code had to be further modified. Now it may encounter placeholders while rendering. This is dealt with by having the `Raytracer` recognize when it has encountered a placeholder. If the placeholder does not contain the image data that it needs, then it must request that data from an outside source. This outside source is called a `PartFinder`. A class that uses the `Raytracer` for rendering pixels must first give it an object that implements the `PartFinder` interface. The `PartFinder` interface is shown in figure 8.

```
package raytracer;

/**
 * Finds a withheld scene part.
 * @author Jason Rickwald
 * @version Nov 4, 2005
 */
public interface PartFinder
{
    public Object getPart ( int partId );
}
```

(figure 8)

The `RaytracerClient` passes the `Raytracer` an instance of `SocketPartFinder`. The `SocketPartFinder` implements the `PartFinder` interface, and retrieves withheld scene data by making a socket connection to the `WithheldPartManager`'s port on the server machine.

The last and most important feature implemented was the balancing of workloads among clients. A basic description of how this would be done, along with a simple sequence diagram (figure 4), was given at the end of the design section. I will now describe the implementation decisions made to make load balancing work.

Load balancing only needs to occur when a client requests work from the server, but there is no work in the queue. When this happens, the `ClientManager` checks to see if there is a `LoadBalancer` running. If there is, it does nothing, and the `LoadBalancer` will hopefully add work to the queue shortly. If there is no `LoadBalancer` thread running, the `ClientManager` creates and starts a new one, passing it the list of connected clients. The code for this is shown in figure 9. In this way, the `LoadBalancer` is a form of singleton [2].

The `LoadBalancer`'s `run()` method gets the progress for each client by calling the `Client`'s `queryForProgress()` method. Each client's progress is returned as a double between `0.0` and `1.0`. Naturally, not all clients are doing work all the time. The `queryForProgress()` method returns `-1.0` in that case or when an error occurs. If one or more clients returns a number other than `-1.0`, then the `LoadBalancer` picks the one that has made the least amount of progress. It then calls the `giveWorkBack()` method on that `Client`. This code is shown in figure 10.

```
public ScenePart getScenePart ()
{
    ScenePart returnPart = null;
    synchronized (sceneParts)
    {
        while (sceneParts.isEmpty() && acceptConnections)
        {
            if (loadBalancer == null || !loadBalancer.isAlive())
            {
                loadBalancer = new LoadBalancer(clients, this);
                loadBalancer.start();
            }

            try
            {
                sceneParts.wait();
            }
            catch (InterruptedException e)
            {
                // don't care, keep looping
            }
        }

        if (!sceneParts.isEmpty())
        {
            returnPart = sceneParts.removeFirst();
        }
    }

    return returnPart;
}
```

(figure 9)

```
public void run ()
{
    LinkedList<Client> checkClients;
    synchronized (clientsLock)
    {
        checkClients = new LinkedList<Client>(clients);
    }

    for (Client client : checkClients)
    {
        client.queryForProgress();
    }

    Client slowestClient = null;
    double slowestProgress = 1.5;

    System.out.println("\tLoad balance checking.");

    for (Client client : checkClients)
    {
        double progress = client.getProgress();
        if (progress != -1.0 && progress < slowestProgress)
        {
            slowestClient = client;
            slowestProgress = progress;
        }
    }

    if (slowestClient != null)
    {
        System.out.println("\tFound. Doing balance.");
        slowestClient.giveWorkBack();
    }
    else
    {
        System.out.println("\tNone found.");
    }
}
```

(figure 10)

Before creating the `Raytracer`, the `RaytracerClient` creates and starts a `ProgressReporter` thread. The `ProgressReporter` starts a server socket on port 4248. It then loops waiting for connections on that socket. The master-slave design pattern [2] is not used, because the `ProgressReporter` should only ever receive one connection at a time on that socket – the connection from the `Client` thread on the server. The `RaytracerClient` then passes the `ProgressReporter` to the `Raytracer`. The raytracing code had to be modified to take a

ProgressReporter, and update it with the current progress of the render (based on the current start and end indexes). Also, whenever the Raytracer is not rendering, the progress is set to -1.0. When the LoadBalancer thread calls each Client's queryForProgress() method, the Client thread creates a socket connection to the ProgressReporter's port on the client computer. The ProgressReporter is used both for getting the render progress from the render client and requesting work back from the render client. To specify which one it wants, the Client thread sends the Integer -1 over the socket. This tells the ProgressReporter to send back the render progress.

When the LoadBalancer thread picks a slowest Client, it calls the giveWorkBack() method on that Client. That Client thread then communicates to the Raytracer, through the ProgressReporter, in order to get some render work back from that render client. An important note is that the render client continues to render between the time that the progress is reported to the time that it actually sends work back. This means that it is possible for that client to finish the job it was working on or even start on a new job during the load balancing process. To keep the server's Client thread and the remote rendering client on the same page, the Asynchronous Completion Token design pattern is used [2]. When the Client requests some work back from the remote client, it sends the start index of the ScenePart that it thinks the remote client is working on. This value is sent via a socket connection to that remote client's ProgressReporter. When the ProgressReporter receives the start index, and not a -1, it knows to call the Raytracer's cutCurrentWork() method. This method takes that expected start index, and returns the new start and end indexes (the smaller chunk of pixels that it plans to render). The code for cutCurrentWork() is shown in figure 11.

```
synchronized public int[] cutCurrentWork ( int startIndex )
{
    if (startIndex == startPixel && startPixel != endPixel && index != endPixel)
    {
        int newEnd = (index + endPixel) / 2;
        if (newEnd < (endPixel - 8))
        {
            endPixel = newEnd;
            System.out.println("\tChanging work to render from pixel " + startPixel + " to pixel " + endPixel);
        }
    }
    return new int[] { startPixel, endPixel };
}
```

(figure 11)

If the expected start index doesn't match the actual start index that it is using to render, then it doesn't change it's end index. However, if they agree, then it will typically change the end index so that the Raytracer only does half of what's left. The exception to this is when the new end index is less than eight pixels away from the old end index. This exception is to keep load balancing from uselessly occurring when the Raytracer is nearly done already. The ProgressReporter then sends back the start and end indexes returned by the Raytracer. The Client (in the giveWorkBack() method) receives those indexes and compares them to the indexes in its ScenePart. If the start indexes match and the new end index is less than the old end index, then it modifies the end index of the ScenePart for that client, creates a new ScenePart from newEndIndex+1 to the old end index, and adds that new ScenePart to the work queue. It also increments the scenePartNumber counter so that the ClientManager knows when the render job is done.

Finally, the ProgressReporter thread terminates gracefully in a similar way to how the ClientManager thread is terminated on the server side. When the RaytracerClient is told to

terminate, it calls the `terminate()` method on its `ProgressReporter`. This stops the `ProgressReporter` from looping on the server socket's `accept()` method. It also forces a `close()` on the server socket, which `SocketExceptions` out of the `accept()` method if the `ProgressReporter` is blocked there [3].

5. Testing and Performance Evaluation

Testing was typically done with both the client and server running on the same machine (the development machine). Testing was also done in the labs with up to eleven computers. During testing, the rendering job was allowed to run completely through without any failures. It was also tested by killing clients during a render. If all clients were killed, the server would just sit and wait for a new client connection, at which point it would continue the render. Also, testing was performed where clients were added while the render was running. In all cases, the system ran as expected.

Rendering requirements were defined in terms of the current raytracer code's abilities. Therefore, requirements 1-11, mentioned in the requirements section, were already satisfied.

Requirement 12 is satisfied to a degree. It is definitely possible to add a large number of computers to the task of rendering a scene. However, at some point the overhead of inter-computer communication over the network outweighs the performance gain of multiple render machines. This trade off point is reached when the time it takes an average client machine to render an average chunk of pixels is less than the time that the client spends communicating with the server. In other words, the client is spending more time communicating with the server than it is rendering pixels. The maximum number of clients that can be added to the rendering job while still seeing a worthwhile gain in performance can change from render to render because of this dependence on the number/complexity of the pixels rendered. The proxy design pattern was mentioned in the design section as a way to balance loads between servers. One way that such a proxy could do this would be to calculate the approximate advantage of adding another client to each server, and giving that client to the server that would receive the biggest performance gain.

Requirement 13 is satisfied in that both the server process and the client process can be ran on the same machine. The performance would be below that of a single-processor single-machine version of the raytracer, though, because of the overhead added by making the application distributed.

Requirement 14 is satisfied. The design and implementation sections above described how the server would recover from client failure. The testing also showed that the server could handle many client failures, but continue to render as long as there was at least one running client.

Requirement 15 was satisfied by the feature that only the structural data of the scene was sent to all clients. Clients could then request any other scene data that they required for rendering. Another feature worth consideration would be to perform lossless compression of the rendered data before sending it over the network. This technique may not be worth the extra time it takes to perform the compression, but it could be useful if the machines aren't all on the same network (perhaps they are spread out across the Internet).

Requirement 16 was satisfied. The design and implementation sections above describe the load balancing thread that is run on the server whenever a clients requests work but the work queue is

empty. The metric used was percent complete. However, another may also be worth testing – number of pixels left. There are trade offs to both metrics, so a weighted combination of the two metrics might even be worthwhile. Also, time could be spent to develop other ways to balance workload. For example, the clients could periodically report their progress to the server, and the client that has made the least progress since the last time it reported is the one that should give back some work.

Requirement 17 was satisfied. The original single-processor single-machine version of the raytracer would take around four minutes to render the same scene that was used to test the distributed version at 600 pixels by 600 pixels. Figure 12 shows the rendering times for the distributed raytracer, with load balancing on, rendering a 600 pixel by 600 pixel scene.

(figure 12)

(figure 13)

Rendering with just one client is obviously slower than four minutes. However, as seen in Figure 12, adding just one more client speeds the rendering time up to around three and a half minutes. In this chart we can also see the gradual leveling off of the performance increase as the load balancing and message passing starts to account for the majority of the time. Figure 13 shows the same scene at the same resolution rendered by clients with load balancing turned off. The same gradual decrease can be seen as message passing begins to account for the majority of the time, although it is more subtle. Comparing the numbers we can also see that load balancing wasn't worth the added overhead until there were at least three clients connected. Keep in mind, though, this is the case for this particular scene and resolution, but for other scenes at different resolutions it may only be worthwhile to have load balancing with four or five clients. This isn't something that can be known before rendering begins.

6. References

1. Liu, Mei-Ling L. "CSC 569 Assignments." CSC 569 - Graduate Course In Distributed Computing. California Polytechnic State University. Dec. 2005 <http://www.csc.calpoly.edu/~mliu/csc569/protect/labs/index.html>.

2. Liu, Mei-Ling L. "Mei-Ling L. Liu's CSc569 Lecture Notes." CSC 569 - Graduate Course In Distributed Computing. California Polytechnic State University. Dec. 2005 <http://www.csc.calpoly.edu/~mliu/csc569/protect/lectureNotes/index.html>.

3. "Overview (Java 2 Platform SE 5.0)." Sun Developer Network. Sun Microsystems. Dec. 2005 <http://java.sun.com/j2se/1.5.0/docs/api/>.

4. Pokorny, Cornel K.E. "CSC 473 - Advance Rendering Techniques." California Polytechnic State University. San Luis Obispo. Fall 2004.