Distributed Ray Tracing
by
Mikhail Kalinin and Alexander Yermolovich

Distributed Computer Systems, CS230/Winter 2008

March 10, 2008

Abstract

Ray tracing for rendering images has been used for some time, however, it is also known to be very computationally intensive task. In this project we focused on finding a practical way to parallelize Ray Tracer algorithm and to find a way to distribute the workload among several computers. The major drawback of traditional ray tracers developed for a single computer is their performance. The rendering of large realistic images and complex scenes can take significant amount of time and often cannot be done fast due to sequential nature of modern processors. Today computer networks are widely available and many modern CPUs have several cores allowing the distribution of parallelizable algorithms such as ray tracer to multiple cores and computers on the network. Distributed approach to ray tracing brings us closer to the goal of rendering large complicated realistic looking scenes in real time.

1. Introduction

Ray tracing is a technique used in computer graphics to render realistic looking images. In the real world "rays" of light originate from the light source. Some objects for example a mirror reflect light while others such as glass refract light. Unlike the real world where the "rays" are emitted by the light source, ray tracing uses different approach. in ray tracing the rays are emitted from the point of view, sometimes referred to as camera. This approach is more efficient, since only rays that reach the eye (camera) are used in computations. It is much more efficient to trace rays from the camera to the light source then it is to trace all rays from the light source to the eyes, because some of those rays may not even hit the camera. Although this is clearly a simplification rays still need to take in to account reflection, refraction, and absorption. When a ray hits the surface of an object it can be reflected in one or several directions at full or partial intensity, which in turn depends on how much light intensity the surface has absorbed. The ray can also be refracted while passing trough the surface, if the surface has translucent or transparent properties. This might also cause the color of the surface to change. The algorithm that handles local lightning model must take into account all of these things. In general, the better the approximation the more resource intensive computation becomes. To make images look more realistic a global illumination model may need to be applied. This model takes in to account the interaction of all objects and light sources in the scene or image. To understand how distributed computing can help this problem, lets look at a basic Ray Tracing algorithm.[2]

2. Ray Tracing Algorithm

First, a few basic terms have to be defined. All the objects that are rendered are part of the

"scene" or "world", these objects are rendered by the ray tracer from a specific view point. This view point is called the "eye" or the "camera". All of the objects are seen through the object called "view window" or screen. Each pixel on the view window is created by a ray that passes through that pixel towards the camera. In reality there is a ray that originates from the eye and passes through each pixel of the view window. A view window can be viewed as a mesh where each cell is a square of certain color. For example, "if you want to create an image at the resolution of 640x400, you would break up the view window into a grid of 640 squares across and 400 square down. The real problem, then, is assigning a color to each square. This is what ray tracing does. "[3]

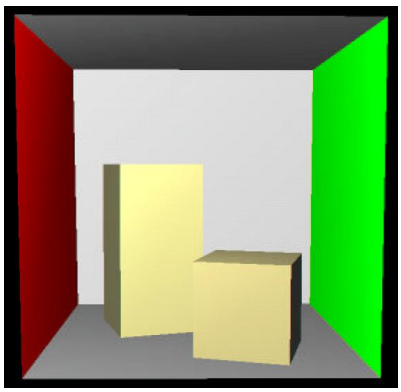

Figure 1.[5]

The basic ray tracing technique is as follows: "for each pixel on the view window, we define a ray that extends from the eye to that point and beyond."[3] From there on, the ray is extended further. If it doesn't hit an object it is assigned default ambient background color. If the ray does hit an object, then local and global illumination models apply. Of the two, the local illumination is the least computationally intensive, but images may look unrealistic (see Figure 1). What gives images the realistic look is the global illumination model, Figure 2.

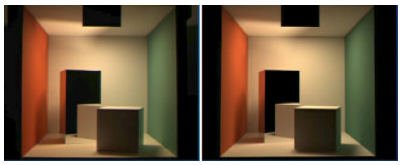

Figure 2.[5]

3. Local, Phong, Illumination Model

Local illumination is a very simplistic and non-physical model. The color of the point is computed independently from all other points through direct illumination by the light sources. It does not take in to account other objects in the scene. The shading of the pixel only depends on the material of an object, location and properties of the light sources, location from where the ray originated, orientation in space and local geometry of the surface. Most often when people talk about local illumination model they mean Phong's model. The color of the point is determined from the 3 components in the equation: $I = ambient + diffuse + specular$. The ambient component of the equation is the general brightness of the light and does not depend

on surface orientation or the location of the light. The diffuse component of the equation is the light that is diffusely reflected off the surface. Finally the specular component of the equation is the light that is specularly reflected from the surface. One more component that can be added to the equation is the *globalambient*. It represents general brightness of the scene and doesn't not depend on any light sources.[4]

4. Global Illumination model

Global illumination model takes in to account all other objects in the scene and their interaction with each other. It also takes into account multiple rays from their origins such as other objects in the scene, and uses the results of those rays in combination with the local illumination model to assign the final color to the pixel. This approach produces much more realistic looking images, nevertheless there is still room for improvement. More advanced technique like distribution ray tracing can be used, to increase realism even further, but generally at the expense of ray tracing performance. This technique will be discussed later in this paper.

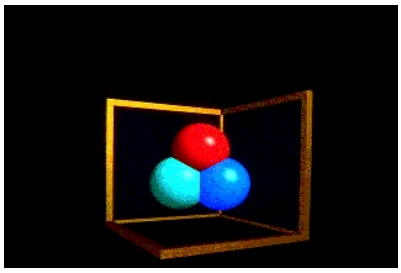

Figure 3.[1]

The first type of ray used in the global illumination model is a "shadow ray". This ray is needed in order to determine if the point which an incoming ray hits is in the shadow. To do this, a shadow ray is generated from the intersection point of the original ray and an object towards all the light sources. If the shadow ray hits another object on the way to the light source it means that for that particular light source the origin of the shadow ray is in the shadow. In that case lightning model only applies ambient color for that light source.

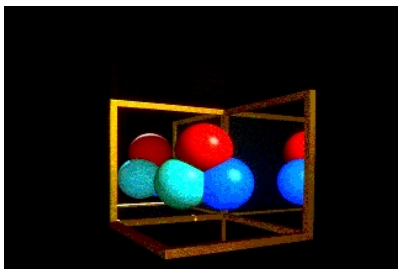

Figure 4.[1]

The second type of ray called a "reflection ray" is generated when a primary ray hits an object and reflects of its surface. If a "reflection ray" hits an object then local illumination model is applied at the intersection point and propagated back to location where the ray originated

from. Otherwise the default background or the ambient color is propagated back. Reflection rays can also generate shadow rays, new reflection rays, and transmition rays. In essence ray propagation is a recursive algorithm. Clearly such process can go on forever, hence the need for artificial limiters. This is especially important when objects are located in a room or between several plane objects. Of course such limit affects the quality of an image. As can be seen in Figure 3 if no reflection rays are generated, nothing is reflected from the mirrors. With one recursion depth there is some reflection in the mirrors, Figure 4. With two levels of recursion even more details can be seen.

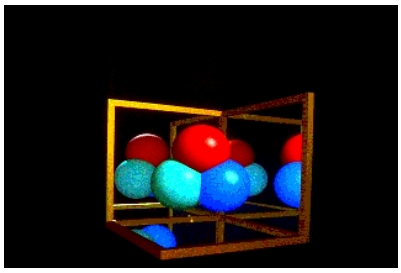

Figure 5.[1]

The third type of ray that is generated is "refracted [or transmition] ray". This ray is generated when an object hit by a ray is transparent or semitransparent. "If the materials on either side of the surface have different indices of refraction, such as air on one side and water on the other, then the refracted ray will be bent, like the image of a straw in a glass of water. If the same medium exists on both sides of the surface then the refracted ray travels in the same direction as the original ray and is not bent."[3] If refracted ray hits an object, then, just like with the reflection rays, local illumination model is applied and the information is propagated back to the origin of the ray. Also, just like reflected rays, refracted rays can generate new reflected, shadow and refracted rays.

5. Advanced Ray Tracing techniques: Distribution Ray Tracing

Using both the local and the global illumination models produces quality images, but they still can be improved. To be more precise, the image would appear too "clean". This is due to the fact that with conventional techniques there are no imperfections so everything looks too crisp; "the shadows are perfectly sharp [Fig 6.], the reflections have no fuzziness, and everything is perfectly in focus." To make images even more realistic soft shadows, fuzziness in reflections and variable degrees of focus can be added. This is where distribution ray tracing techniques come in. [5]

Since antialiasing, soft shadows, and reflections affect performance the most, we describe them in some details. The basic premise for antialiasing is to take an average of the area of a pixel instead of just at the center point. To achieve this ray tracing algorithm fires multiple rays from the eye to each pixel. Just like with single ray models global and local illumination is used to calculate the color value for each ray. When color values are computed for all rays shot through the pixel, the values are averaged to determine the final color. This adds to the realism of the pixel at the expense of performance or computational speed, because, just like with a single ray model, reflections, refractions and shadow rays are produced for a larger number of rays.

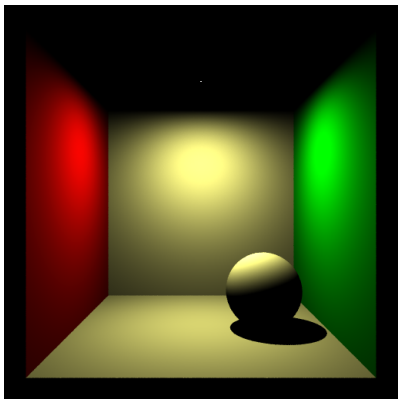

Figure 6.[1]

Similar principle applies to produce "soft" shadows in the image. In "Hard" shadows a ray tracer treats light as infinitesimal points of certain intensity. So the shadow ray either hits that point or it doesn't. Which creates umbra, but not penumbra. To solve this problem the algorithm can be modified n such a way so that instead of light being treated as points, light is represented as an area. The intensity of the light is reduced proportionally to the area. Such algorithm will generate multiple shadow rays that intersect the area in various locations. For points in penumbra this will cause some rays to reach the light while others will be blocked by objects. The average is then calculated for all shadow rays. This would result in penumbra points being lighter then the points in umbra, hence producing soft shadow effect. This effect can be observed in Figure 7.

To achieve "fuzziness" effect with reflections, the same trick as with shadow rays can be used. Instead of generating just one reflected ray, multiple reflection rays are generated for (towards) some small area. To achieve the desired effect and to avoid artifacts that may originate in the process, the location in the region can be altered slightly at random. As with regular reflection rays described previously, each ray like this can potentially generate many more other rays in a recursive way.

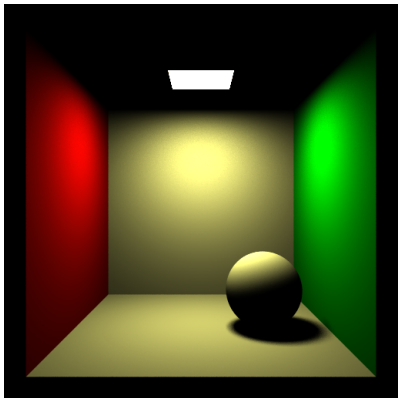

Figure 7.[7]

6. Distributed Ray Tracer

Ray tracing is well known parallelizable application. Many techniques have been proposed and developed over the years [8,9,10,15,16,17]. There are two primary ways to parallelize the raytracer [16,8]

- **Data driven approach** In demand driven ray tracer the data is replicated to all computational notes and the screen is subdivided into a number of distinct regions. From there on each processor performs rendering on the regions. Once all subparts are completed, the results are combined to form the whole image. [8,9,10,16] The advantages of this approach is that communication is kept to a minimum, up until the point where results of subtasks have to be accumulated on one of the nodes in order to form an image. Perhaps the main disadvantage of this approach is that different subparts may take longer then others depending on how many object and their complexity the subpart has. There are several ways to solve this problem. One way is to divide the scene into many subparts such that there are many more subparts then processors. A centralized server processor then can distribute the tasks to its clients. Once a given subpart is completed on a particular node, server can allocate the next task to the available node. This way one computational node can work on one complicated subtasks while another node can work on several small subtasks. Another way to solve this problem is to estimate ahead of time approximate difficulty of subtasks. With this approach a larger but less difficult subpart can be given to one node, while a smaller but more complicated subtask can be given to another node.

- **Data Parallel approach**
  Whenever a scene is very large and cannot fit into a single computational nodes memory a Data Parallel approach can be involved. With this approach data is distributed among the processors. Each processor has a particular space allocated to it and whenever a ray passes through that subspace the processor performs tracing of rays on objects in its subspace. If a ray ventures outside processors designated region it notifies processor responsible for that region and passes that ray and its properties that that node [8,9,10]. The main problem with this approach is a network. Rays can traverse many different scene subparts before ever hitting any object and when they do a reflected ray and or shadow rays have to be taken into account. The problem can be intensified due to the fact that some subparts may get fewer rays passing through them then the others. Hence some nodes will be performing more computations then others. One way to solve this problem is to perform sample tracing, whereby shooting perhaps one ray per region (for example of size 10x10) and count the number of rays passing through its subpart. This will reduce computation by a factor of 100 and yet, will give an estimate as to how many rays will pass through each region. It is possible then to partition space such that the number of rays entering each regions is approximately equal.

- **Problems in the real world**
  While each particular way to parallelize ray tracer can have a set of unique problems and drawbacks, there are problems that are common to all of them. Consider a real world where ray tracer computations are done on multiple nodes. Each node may have different hardware and different average (ambient) load. Some computing nodes can be much more powerful then others. In the ideal case all nodes would have an equal load. Without prior diagnostic or testing one very slow node can slow down the entire process. To address this problem, it is possible to send a sample task to each node to benchmark their computational performance ahead of time and to determine its average load. Network system

is also important. Slow network can cause a slowdown. Access to some computational nodes can be much faster then other nodes. One way to overcome this problem is to estimate the data transfer speed. Nodes on a slow connection can work on a smaller, but more computationally intensive subpart. Working on a smaller subpart ensures that once the computation has completed, the smaller output dataset will be transfered. Nodes that perform computation on larger parts, but perhaps less computationally intensive can afford to communicate more often to the server and send larger output back to the server upon completion without significant network delay.

7. **Our approach**

A ray tracer application that was developed as part of this project employs data driven approach. A screen of desired size X by Y is divided into a number of sub screens, depending on the number of computational nodes. For example, with screen size of X by Y and 2 computational nodes the screen will be subdivided into two screens of size X/2 by Y. Figure 8 shows example of the ray tracer results where the screen is divided in to 5 subscreens each being processed on one of the 5 computers. Coordinates of all objects and camera position are kept constant. Rays from the camera are shot through each pixel of a screen just like they would be in a traditional ray tracer.

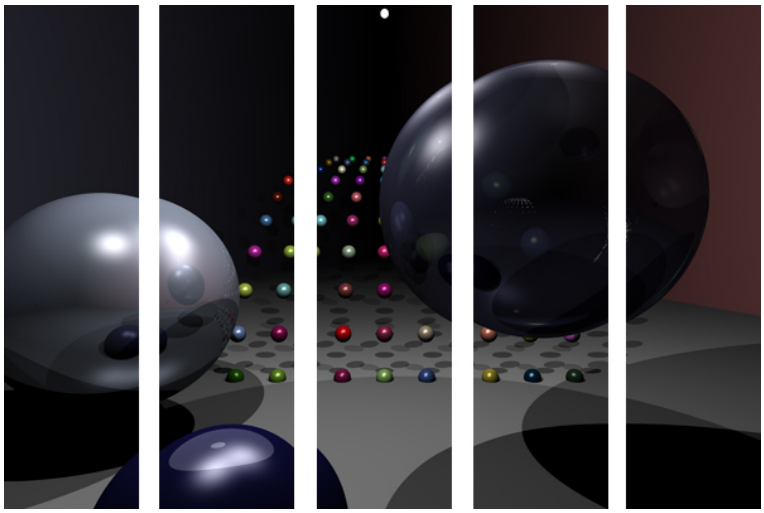

Figure 8

8. **Implementation**

Ray Tracer application

The goal of this application is to show the advantage of performing ray tracing in a distributed fashion, splitting one task into several tasks so that they can be performed in parallel. Hence the current implementation does not include all latest advancements and techniques available, nevertheless some are implemented. Currently a ray tracer has several types of objects: plane, sphere, plane light source, sphere light source. Each object has coordinates that determine its place on the scene. In addition to coordinates spheres also have radius that can be specified. In addition, model for colors and the light intensity as well as reflection, refraction and diffusion

coefficients have been implemented. A simple shading model is also implemented.

TREx
One of the problems with parallelizing any application is the time involved with design and development of communication and task distribution and management system. There were several possibilities: CORBA, RMI, RPC, EJB and many others. For a computational platform we choose TREx system developed by Professor Scherson and his students, because TREx system removed the need to implement a custom system to perform ray tracing in a distributed way. In order to perform parallel ray tracing a few things have to happen: raytracer executable has to be uploaded or manually installed on computational nodes, raytracer configuration files have to be uploaded to computational nodes, all tasks have to be started in an appropriate way, task management has to be performed, results of computations have to be forwarded to a designated server computer. TREx can be used for all of the above.

- **TREx Client**
  TREx consists of client executable and a server executable. Client executable can be installed on several client computers. Upon start client application is running as a background process waiting for server to send executables, configuration files and to receive a start of computation signal upon which the raytracer task is launched.

- **TREx Server**
  Server task manager application can be installed on a node designated as server. Via TREx server executable files and appropriate configuration files can be send to clients. After successfully sending tasks, user can request to start computation on the nodes. Upon completion results are sent back to the server.

Parallel Ray tracer
Currently ray tracer consists of several parts: server ray tracer application and client ray tracer application.

- **Client ray tracer application**
  After the TREx server uploads client application to available computational nodes, ray tracer client application can be started. Upon start it reads configuration files in order to determine which subtask has been designated to it by comparing its (Internet Protocol) IP address to the address in the configuration file. If the IP address matches one of the IP addresses in the configuration file, client application starts. Upon completion a file with the result is saved.

- **Server ray tracer application**
  Server ray tracer application is a multithreaded application that monitors the arrival of data from raytracer clients. It reads the configuration files to determine which client is responsible for which subtask. When the application is started is spawns multiple tread, one for each client. While output from clients is not available it patiently waits for the data arrival. Minimizing the ray tracer server application places it in the background showing only an icon in the windows taskbar (Figure 9).

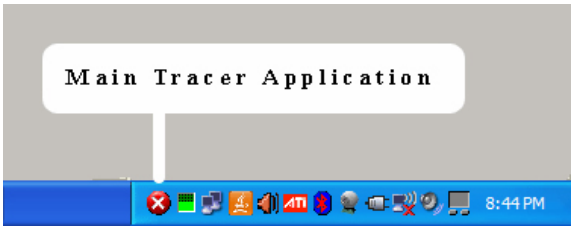

Figure 9

Clicking on the icon resores application state from hidden to normal. Once TREx server received the output from the clients running ray tracer application, appropriate thread responsible for particular output triggers an event that notifies the application that the output from that client is available. Once all outputs are received, ray tracer server application starts assembling outputs it received in to rendered image. Upon completion user can view and save the image (Figure 10)

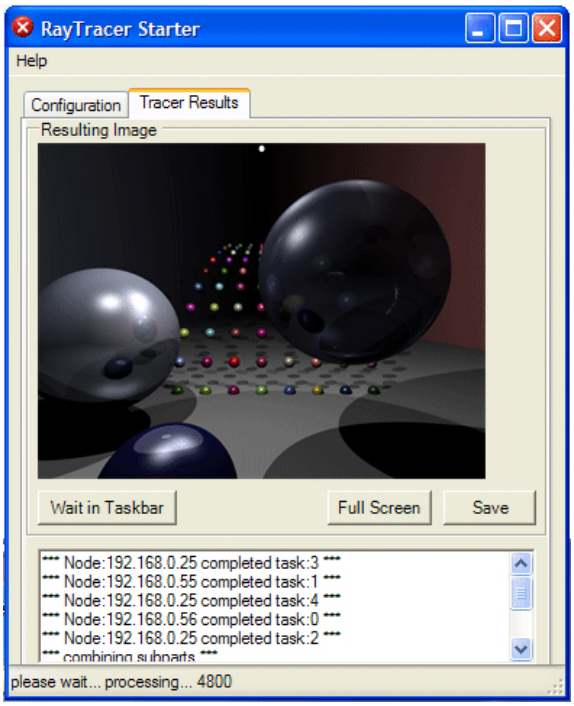

Figure 10

.

9. Results

The results of the ray tracer can be seen in figures 11, 12. Figure 11 shows output image of size 4800x3600 pixels. This image contains 3 planes, 80 small spheres, 3 big spheres and 3 light sources. Note that the left big sphere reflects, while the right sphere refracts most of the light.

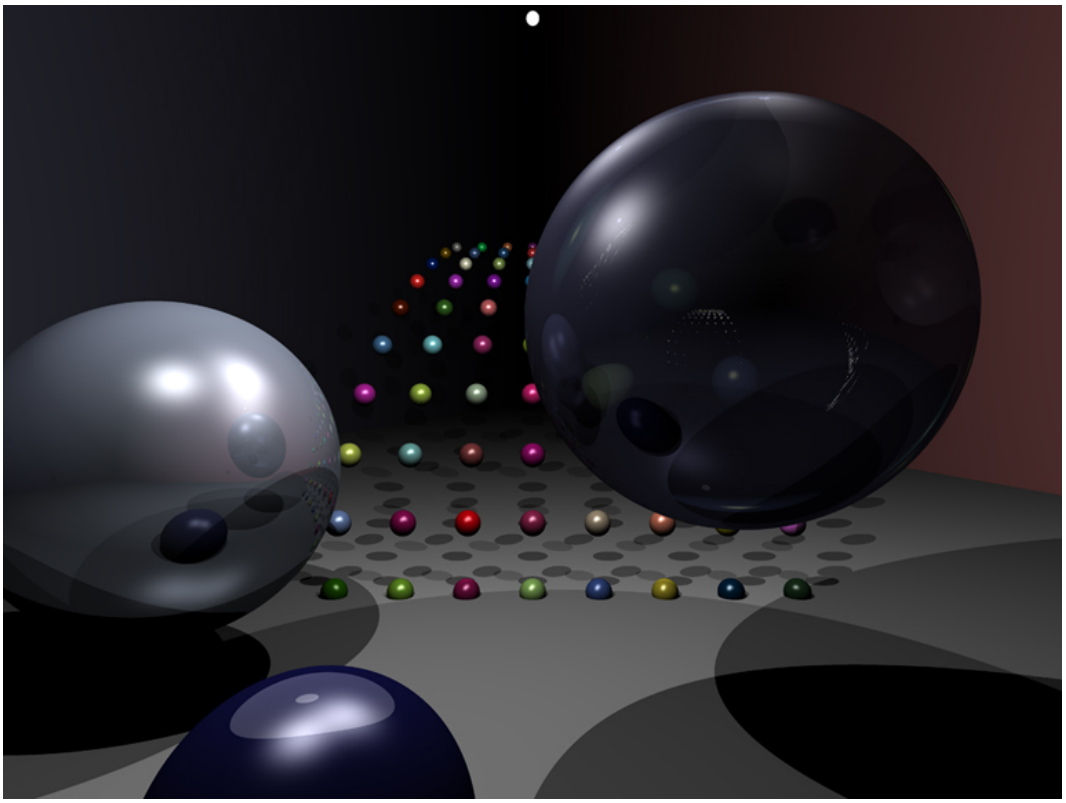

Figure 11

Image presented in figure 11 was computed on 5 computers with AMD Athlon 64, 2.4 Ghz processor (or similar) and 3GB of Ram running Windows operating system. To process the entire image on one computer took 5 minutes 42 seconds. Table 1 shows how long it took to complete each subtask on 5 computers.

Subtask	Duration (min:second.milliseconds)
task 0	0:36.954
task 1	0:39.250
task 2	1:08.578
task 3	2:10.016
task 4	1:08.844

Table 1

Figure 12 shows similar scene, except now the size of the image is 3200x2400 pixels and there are 240 small spheres. This image took 22 minutes 49 seconds on a Intel Core Duo 1.66GHz with 2Gb Ram. Table 2 shows how much time it took to process the same image on 5 computing nodes.

Subtask	Duration (min:second.milliseconds)
task 0	2:32.859
task 1	2:39.468
task 2	4:37.891
task 3	8:37.469
task 4	4:39.344

Table 2

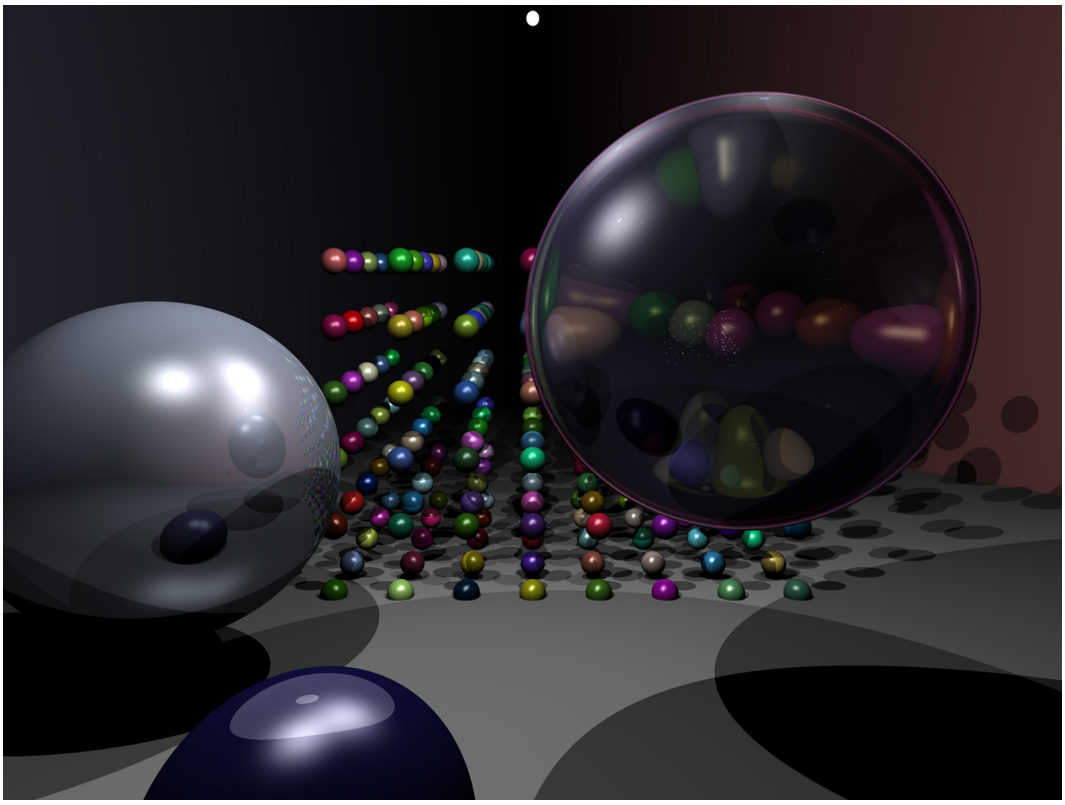

Figure 12

10. **Conclusion and future work** Clearly running a ray tracer application in distributed way is advantageous. The speed-up can make it possible to render very complicated scenes of very large sizes. However, it is also appearant that a better way to separate the scene can be employed. Table 1 shows that node 3 took the longest to finish due to the complexity of its scene, while node 0 finished in just 36 seconds. Results presented in Table 2 indicate similar pattern.

    - One way to improve the performance is to split screen depending on the complexity of the scene. This approach would ensure that the load is more balanced between all computational nodes.
    - Another way, is to split the screen into many subscreens so that the number of subtasks exceeds (perhaps by a factor of 100) the number of computing nodes. In this approach each client will process a subset of screens. Subsets for each process cab be allocated randomly to ensure approximately equal load among them.
    - It is also possible to create an "on-demand" environment, where clients can "ask" a centralized server for a new task once they are on-line, available or when they complete their previous task. Server can have a set of available tasks, such that their number exceeds the number of processors. In this approach some clients may work on one large task, while other clients can work on much greater number of smaller, computationally less expensive tasks.

.

`Executables and other information is available at:`
http : *//sfkd.dyndns.org/mikhail/ray_tr/index.htm*

References .

[1] G. S. Owen, "Ray Tracing," http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtrace0.htm", 1999

[2] G.H. Spencer, M. V. R. K. Murty, "General Ray-Tracing Procedure," Journal of the Optical Society of America, 1962

[3] P. Rademacher, "Ray Tracing: Graphics for the Masses," http://www.cs.unc.edu/r̃ademach/xroads-RT/RTarticle.html

[4] T. Moller, "Introduction to Computer Graphics: Local Illumination," Coursework material. Simon Fraser University, 2004

[5] "Local Illumination Models" Coursework Material. MIT 6.837, http://groups.csail.mit.edu/graphics/classes/6.837/F02/lec 2002

[6] P. Shirley "Fundamentals of Computer Graphics", AK Peters, Ltd. 2004

[7] Image from Z Ray Tracer http://graphics.cs.yale.edu/hongzhi/zrt.html

[8] E. Reinhard, "Scheduling and Data Management for Parallel Ray Tracing," Dissertation, Doctor of Philosophy in the Faculty of Engineering, University of Bristol, 1999

[9] D.E. Orcutt, "Implementation of ray tracing on the hypercube," Third Conference on Hypercube Concurrent Computers and Applications, 1988. vol 2

[10] T. Priol and K. Bouatouch, "Static load balancing for a parallel ray tracing on a MIMD hypercube," The Visual Computer, 1989

[11] E. Weisstein, "World of Physics, Law of Reflection" http://scienceworld.wolfram.com

[12] "3D Geometry Primer. More On Vector Arithmetic" http://www.flipcode.com/geometry

[13] I. Wald, P. Slusallek, "State of the Art in Interactive Ray Tracing," Eurographics 2001

[14] J. Arvo et al, "State of the Art in Monte Carlo Ray Tracing for Realistic Image Synthesis," Siggraph 2001

[15] E. Reinhard and F. W. Jansen, "Rendering Large Scenes Using Using Parallel Ray Tracing," Parallel Computing 1997

[16] I. D. Scherson, E Caspary, "Multiprocessing for ray tracing: a hierarchical self-balancing approach," Visual Computer ,1988

[17] A. Chalmers and E. Reinhard," Parallel and Distributed Photo-Realistic Rendering," ACM Siggraph, 1998