Parallel Radiosity Using the Message Passing Interface

Joshua D. Nasman
nasmaj@cs.rpi.edu

Jon Zolla
zollaj@rpi.edu

Figure 1: *Sample rendering of an 8.5k face mesh*

Abstract

A method is described to parallelize the computation of global illumination using radiosity. Radiosity is a lighting technique which uses the diffuse properties of materials as well as emitting patches to determine the lighting in a closed space. By distributing light throughout a scene each iteration based on the properties of each surface, light is distributed until all light within a given threshold has been approximated. Radiosity is especially effective at rendering effects such as color-bleeding[Goral et al. 1984]. The technique described in this paper parallelizes the initial setup of this method, calculating form factors, using the Message Passing Interface (MPI). We describe a method to distribute the work across processors as well as a number of optimizations to make our algorithm more efficient. Also described herein are the technical challenges faced as this algorithm was developed.

Keywords: radiosity, global illumination, mpi, parallel computing

1 Background

1.1 Radiosity

Radiosity is a view independent global illumination algorithm based on methods from thermal engineering that models the interaction of light between diffuse surfaces[Goral et al. 1984]. Radiosity calculation is an iterative method, where the patch with the most undistributed energy shoots light to every other patch in the scene during each iteration. The amount of light distributed from any patch i to another patch j is dependent on the form factor value F_{ij}. The iterations cease when the total undistrbuted light in a scene comes within ϵ of zero.

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{cos\Theta_i cos\Theta_j}{\pi r^2} V_{ij} \, dA_j \, dA_i \qquad (1)$$

The value F_{ij} is computed using equation (1), where the value V_{ij} is the visibility between patch i and patch j. Θ_i is the angle between the normal and the vector from i to j; Θ_j is the opposite; r is the distance between the faces. If all of these values are known, then F_{ij} can be computed in constant time, meaning it is possible to compute all n^2 form factors in $O(n^2)$ time. However, calculating the visibility parameter is commonly done by casting rays between patches i and j, which can take $O(n)$ time with a simple ray casting implementation, which pushes the overall computation of form factors towards $O(n^3)$.

1.2 Foundations of Parallel Computing

Parallel computing describes any one of a number of ways to divide a task and distribute work across multiple processors[Grama et al. 2003]. This allows more than one processor to work on obtaining a solution to a problem at the same time, and ideally this will lead to a shorter overall time to the result.

There are several metrics to determine whether or not a parallel program has significantly improved execution speed. The first is known as *speedup*, and is calculated using equation (2).

$$S_P = \frac{T_1}{T_P} \qquad (2)$$

Where P is the the number of processors used, and T indicates the execution time for a given number of processors. If T_1 is the execution time of the best known sequential algorithm, then this will calculate absolute speedup. If T_1 is the execution time of the parallel algorithm on a single processor, then this formula will calculate relative speedup.

The ideal speedup would be $S_P = P$. This would mean that the algorithm executes P times faster using P processors.

Efficiency is calculated by equation (3), which the percentage of the ideal speedup that has been achieved. The ideal effiency for a parallel algorithm is 100%.

$$E_P = \frac{S_P}{P} \qquad (3)$$

1.2.1 Types of Parallelism

There are two main ways that sequential problems can be divided to take advantage of multiple processors. The first is task parallelism, where all of the processors operate on the same data set, but perform a different task. A common example of this is dividing a three dimensional space amongst processors, and having each processor be responsible for performing a task on particles as they move through that processor's assigned space.

The second type of parallelism, which we utilized in this paper, is data parallelism. In this case, every process performs the same task on a subset of the entire data set.

1.3 MPI Overview

In this paper we used the Message Passing Interface (MPI) to achieve parallelism. In this paradigm, processors communicate solely by passing messages amongst each other via a set of function calls. These messages can be passed both synchronously and asynchronously. Many types of communication are possible, including one to one, one to many, many to one, and many to many.

Each of the n processors is assigned a rank from 0 to $n-1$. When messages are passed they have both a source and destination rank associated with them, along with a tag value. The reason for this tag value will become apparent later in the paper, since we take advantage of it in our implementation.

There are several implementations of this API, including MPICH, OpenMPI, and vender specific distributions. There are subtle differences between these implementations which we ended up encountering as we implemented our algorithm.

1.4 Blue Gene/L

One of the target systems for this project was the IBM Blue Gene/L computer located at Rensselaer Polytechnic Institute's Computational Center for Nanotechnology and Innovation. This computer contains 32,768 processors and is built specifically for quick communication using its 3 networks. The Blue Gene's 3 networks are the point to point network, the collective network, and the barrier network. The point to point network is for simple sends and receives between processors. The collective network is useful for operations such as scatter, gather and all reduce. `All_Reduce` allows an operation over a common variable across all processors on the machine.

The BlueGene also has an extremely limited amount of memory. It is set up such that it has 16,384 nodes of 2 processors a piece. Some nodes have 512 MB of ram while others have 1 GB of ram. Because of this, once large jobs are done each processor will only have 256 MB of ram. This means that as this project develops it will be necessary to be extremely careful of memory issues.

2 Goals

Before undertaking this project, four distinct goals were set forth. They were:

- Speed up radiosity using parallel computing
- Obtain at least a factor of two speedup from one to four processors
- Implement this algorithm using the Message Passing Interface (MPI)
- Provide a good foundation for further research

Our first goal was motivated by the availability of an IBM Blue Gene Computer to the RPI campus. The Graphics research group of the Computer Science department uses radiosity for much of their research and an efficient parallel version of the code could ultimately be very useful.

As this project had approximately a month of time available to it, the group wanted to show scalability on a small scale in this time before moving to a larger platform such as the Blue Gene/L. For this reason a factor of two speedup on a four processor run was considered a reasonable goal.

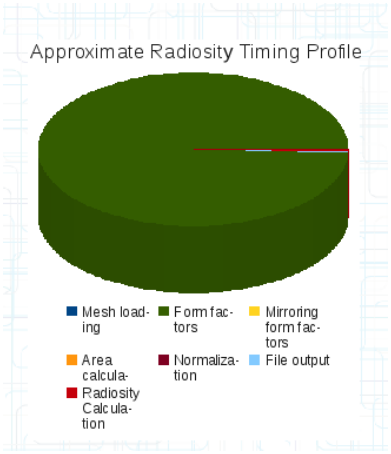

Figure 2: *Distribution of work in radiosity*

After running our original code on some sample meshes we determined that calculating for factors accounted for more than 99% of the computation time on larger meshes. You can see the approximate ratios in Figure 2. Parallelizing this portion of the code seemed an appropriate goal for a semester project.

The Message Passing Interface was decided on as the parallel library of choice because of its prevalence in the distributed computing field. The architecture of the Blue Gene and other high performance computers are designed to take advantage of MPI communication . The three networks of the Blue Gene correspond to the different types of MPI messages..

This project will be continued by Joshua Nasman after the goals of this paper are reached, so it was also important that a good foundation was established so future research could proceed.

3 Implementation

3.1 Separation of Rendering and Form Factor Computation

The initial code were were using for our implementation was a combined radiosity solver, ray tracer, and renderer. This meant that in order to parallelize the form factor compuation using MPI, we had to work MPI into the OpenGL renderer. This proved to be difficult for several reasons. First, we were unable to ever call `MPI_Finalize`, used to clean up after MPI programs, since the GLUT library provides no callback to execute code when the window is closed. Secondly, interactive OpenGL rendering is not available on systems such as the Blue Gene.

To get around these problems, we separated the form factor computation into a standalone executable which would ouptut a binary file containing the computed form factor matrix. This file could then be loaded by the renderer. We found this method to work much better, depsite the annoyance of maintaining some repeated code in the project.

3.2 Form Factor Computation

Calculating the form factors is a naturally parallel task. Initially, form factors were calculated by naively dividing the rows of the

form factor matrix and having each processor send its results to processor 0, whereupon processor 0 assembled the matrix. This was done my using `MPI_Send` from all of the worker nodes (node 1 through $n-1$) and using `MPI_Recv` from processor 0.

One important design decision was that a deterministic algorithm was used to assign rows to each processor, which could be used by rank 0 to determine the rows assigned to every processor. Because of this, it was possible to simply have each processor send its form factors to processor 0 without prefacing the message to indicate which rows were going to be sent.

A problem appeared in the first implementation, where the messages being sent were larger than the MPICH permitted on the test system. In order to alleviate this problem, a temporary hack was inserted where each message contained a maximum of 100 form factors. After moving to more stable MPI implementations, it was discovered that this was not only uneccessary, but also dramatically reduced scalability. Because of this, we removed the message size limitation.

In order to more efficiently calculate form factors, the fact that F_{ij} only varies from F_{ji} by a factor of the areas of the second face was exploited. Instead of calculating every form factor, only the upper right half of the matrix was calculated in parallel. This half of the matrix was then sent to rank 0. This processor then copies the upper right half of the matrix across the diagonal to the lower left. Finally each entry is divided by A_j.

3.3 Workload balancing

Once we began calculating only half of the form factor matrix, our initial assignment algorithm ran into issues. Because the rows were being assigned in increasing order, and the amount of work associated with each row varied, each processor was assigned a very uneven amount of work. Initially we were also not utilizing rank 0 to do any form factor computation, it was only collecting form factors.

Processor 0 was utilized more effectively by assigning it work as well. The one difference in it's implementation is that it does not send its form factors since it already has them.

In order to balance the workloads on the half matrix, different methods of dividing work were discussed. One idea was to divide up the geometry of the triangle such that the work done for processors $\frac{n}{2}$ to $n-1$ (which formed a triangle) was fit into the empty triangle from processors 0 to $\frac{n}{2}-1$. Another method discussed was to simply divide the form factors needed by the number of processors and no longer divide by rows at all.

Ultimately, a simple solution was found. The method was to initially delegate only half of the rows to processors. The second half was then delegated in reverse order. This evens out the workload since when a processor is assigned a row involving more computation from the top of the matrix, this is balanced by assigning a row with less work from the bottom of the matrix. This method adequately divides work, except when the number of processors starts approaching half of the size of the mesh. When this occurs, the difference in work assigned becomes noticeable, since a very small number of rows are being assigned to each processor.

One of the important qualities of our load balancing algorithm is that each processor has full knowledge of the assignment algorithm. This means that no communication between processors is necessary to initiate form factor computation. However, this also has the downside of providing no dynamic load balancing capabilities – the quality balance is determined completely by the static assignment function.

3.4 Ray Casting

One of the most expensive operations in the form factor computation is the ray casting used to determine visibility between patches. The current implementation uses no type of spatial or hierarchal data structure to store geometry, so every face in the mesh must be examined for every ray. This increases the cost of form factor computation from approximately $O(n^2)$ to $O(n^3)$.

To get around this increased cost, we gave the option to use a separate mesh for ray casting. This mesh would have the same physical geometry as the normal sized mesh, but would have significantly fewer faces. This meant our cost to cast a single ray would not increase with the size of the mesh being used for form factor calculations.

3.5 Usage of MPI

Our implementation utilizes MPI communication to send computed form factors from ranks 1 to $n-1$ to rank 0. This is accomplished by using synchronous `MPI_Send` and `MPI_Recv` calls.

Once each processor finishes computing its assigned rows in the form factor matrix, it initiates one call to `MPI_Send` for the first row it computed, which blocks until rank 0 receives it. The message utilizes the tag field in order to communicate which row is contained in the message. Once rank 0 receives this message, another is sent until all rows have been exhausted.

Rank 0 does not specify a source rank or tag in the `MPI_Recv`, so the order messages are received is abitrary. Once a message is received, the tag value is read to determine where to place the received message. `MPI_Get_count` is then used to determine the number of form factors contained in that row, so the precise placement in the form factor matrix can be calculated. Since rank 0 can calulate the total number of rows and knows how many it computed itself, it can calculate how many messages it should expect to receive.

3.6 Testing

By outputting a form factor file, we could simply compare this file with previous results as we scaled to ensure that our calculations weren't changing with respect to a serial algorithm. Although this testing accounts for any errors that would be introduced during parallelization, it would fail to catch any issues with the original form factor computation itself. We considered this acceptable since the project focused mainly on the parallel scaling of the algorithm.

4 Results

The first results we obtained measured how performance changed across processor counts with a fixed size mesh. We chose a mesh of size 5.5 thousand faces because of memory restrictions and current limitations in the program. Figure 3 shows how the results scaled close to linearly for smaller runs. Figure 4 is provided to show a different perspectice on scaling. It shows how closed to perfect scaling was obtained. It can be seen that up to 256 processors the scaling was nearly perfect. We believe the dropoff in performance which followed was largely a result of the limited sized mesh. The mesh size was 5.5K and rows of the matrix were divided in groups of 2 (as described above). As the processor count approached 1/2 the number of faces our program understandably has poorer load balancing. On the 2048 processor case shown in the graph some processors were given 2 rows while other were given 4. For this reason many of the processors wasted nearly half of the computation time on form factors and scaling was only around 64%.

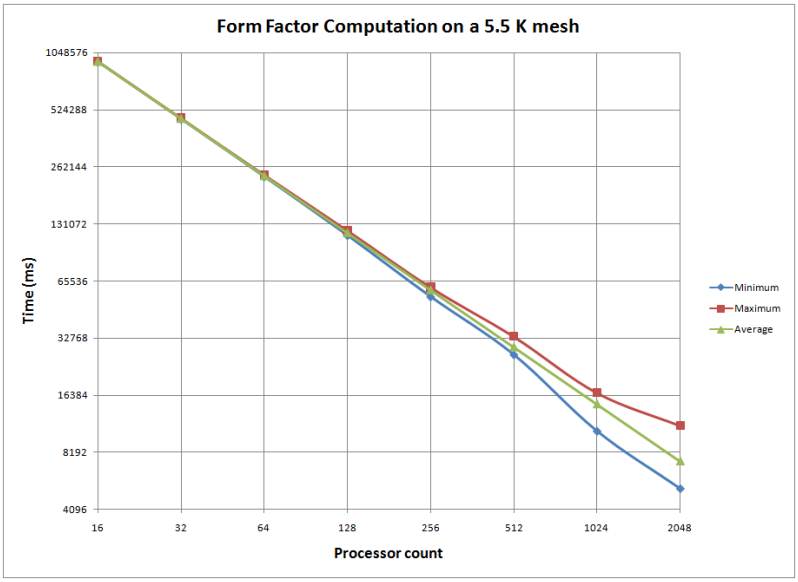

Figure 3: *Effect of processor count on execution time*

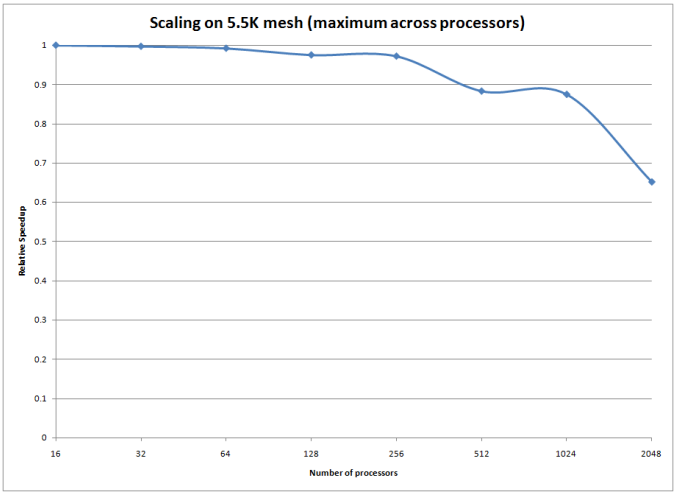

Figure 4: *Scaling as processor count increases*

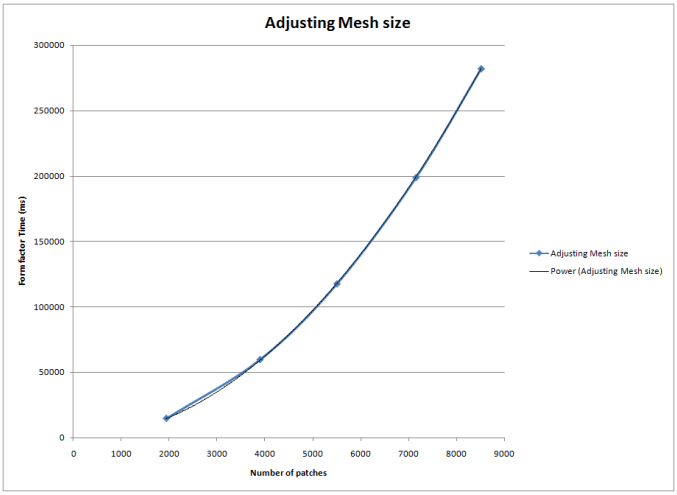

Figure 5: *Performance as mesh size increases*

For our second test we varied the number of mesh faces across a set processor count (128). The results in Figure 6 showed that the time increased quadratically as the mesh size was increased. Our initial algorithm was n^3 but our raycasting technique reduced it to quadratic. It is important to notice that the time isn't exactly a function of n^2 because a small mesh is used in the base case for ray casting and this is used throughout the test. This factor was on the order of 700 faces for most of our test (which is significantly greater than even log n). So our form factor algorithm is slightly greater than $n^2 logn$ overall.

Figure 6 is provided to show how well we performed agains one of our initial goals: to get a scaling factor of at least two on four processors. You will notice that this goal was achieved. On up to the four processors on that case we scaled almost perfectly. The results obtained on the BlueGene exceeded all of our initial goals.

The final test performed used 128 processors, and raycasting was performed using the full mesh (5503 faces) as well as a reduced mesh (712 faces). The results to this experiment are still not fully understood. The reduced mesh had 13% of the faces of the full mesh. Computing the form factors using this mesh took only 11% of the time of the full mesh. Theories of why this may be the case include postulating that it might have been possible for more of the processors to remain in cache when the smaller mesh was used or that it could have been close to perfect scaling but an operating system anomaly could have switched out the case where we were using the larger mesh.

5 Limitations

5.1 Memory Usage

The major limitation of our software that has prevented us from scaling to meshes larger than 8500 patches has been excessive

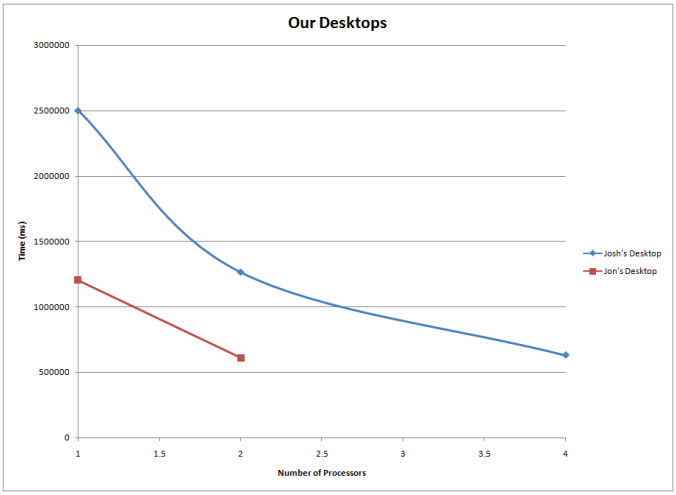

Figure 6: *Performance using desktop hardware*

memory usage. Currently, the entire mesh geometry and form factor matrix is stored on every processor. The form factor matrix alone for 8500 faces is approximately 550MB, so considering the maximum memory the Blue Gene can have available per processor is 1024MB, memory usage is a major issue.

There are several solutions to this problem. First, we can most likely get around storing the entire form factor matrix on any processor at a given time. Also, we could distribute the mesh geometry over all of the processors. This geometry distribution will be discussed in more detail in the future work section.

5.2 Form Factor File Size

We discovered that one major bottleneck for our algorithm was writing out the form factor file. As the mesh size grows, so does the form factor file, which means output takes more time. Since the file output is done by a single processor after all of the computation is complete, this is an unacceptably slow serial portion of the algorithm as a whole.

The best way to avoid this may be to have rank 0 do no form factor computation and instead be entirely responsible for receiving form factor rows and writing them to the file as other processors continue to compute form factors . This would involve changes to the file format so that rows could be written in arbitrary order which may actually increase file size, but could improve the parallelism of the algorithm.

5.3 Brightness Glitch

Near the completion of our project, we noticed that there appears to be a brightness difference between different size meshes. It is unknown whether this is based on the initiation of light energy in the scene or if it is a result of the form factor matrix. Ideally while the smaller mesh should be less accurate, the amount of light in the scene should not be affected.

6 Conclusion

This project accomplished the four goals set forth at the beginning. We were largely successful speeding up radiosity using parallel computing. We ran runs up to 2000 processors while still scaling at greater than 60%. Up to 500 processors the scaling was close to linear.

Our initial goal of scaling by a factor of at least 2 using 4 processors was exceeded when we scaled to nearly 4 times using 4 processors. The implementation was done using the Messages Passing Interface as specified. In the future, we may take advantage of more of the variety of calls in MPI, but we were largely successful. We successfully set a good foundation for future research as will be described in the future work section.

We discovered that scaling to several thousand processors is possible, but that scaling to any more necessitates a more intelligent data structure. The form factor matrix became approximately 550 megabytes on our largest test runs. Because the Blue gene can only have 1 gigabyte of space per processor even in co-processor mode, it will be necessary to change the structure in future work.

Overall, the project was largely successful. All goals were met and future areas of research were discussed and planned.

7 Division of Labor

It is hard to define the division of labor between team members as towards the end of the project much of the project was done in close communication either in the same room or in close communication on-line. Both team members did equal work setting up mpi on their respective machines as well as contributing to the final paper.

Josh did more of the work setting up the initial MPI program, including the initial work in attempting to balance work on processors. He also was responsible for running the jobs on the blue gene, creating the charts, and solving the endianness issues.

Jon's designated work included cleaning up the initial code and providing some additional optimizations. He experimented with the role of processor 0 (ultimately making it compute as well), and adjusted the MPI calls to use MPI pack.

Many other individual bugs were worked out throughout the course of the project which were divided fairly equally through the group.

Overall, both group members spent upwards of 40 hours working on this project.

8 Future Work

As this research is part of a graduate student's research the amount of potential future work is more than there is room for here. This section is a small selection of that.

8.1 Reduce Memory Usage

As the primary current limitation of this code is that the form factor is too large in memory, the most immediate initial work will be to find ways to reduce the size of the form factor matrix. This could be achieved in a number of ways. The simplest one would be to reduce the form factors from double precision to C++ floats. This will enable the form factor to grow to twice the size or have a factor of $\sqrt{2}$ more faces. This will not be sufficient to run the large scale test.

8.2 Distributed Geometry

One major improvement that could be made would be to distribute the mesh geometry over all of the processors, similar to the implementation by Studdard[Studdard et al. 1995]. This would have

several advantages over the current system, where the gometry is replicated on every processor.

First, it would reduce the amount of memory required on each processor and help the software scale to larger meshes. Secondly, if implemented correctly, the geometry can be distributed in a way similar to a spatial data structure, where a single processor contains the faces for certain areas of space. This can reduce the cost of ray-casting, since a query could be done asking each processor if a ray intersects their space, without checking all faces inside. Also, each processor could store their data in a hierarchical data sctructure, such as an octree, to further speed up intersection testing.

8.3 Improved MPI Communication

Another useful topic for future research would be more effectively using MPI. Currently, almost the entire implementation is done using blocking sends and receives. This only utilizes one of the BlueGene's 3 networks. It would be useful to see how performance changes if collective operations such as `MPI_Gather` are used.

8.4 Selective Form Factor Sending

To get to even larger matrices without running into scaling isues, a method will be used to only store the more important part of the matrix. This will likely be in the form of only storing the top 10% or 20% of the matrix. If the form factors are still normalized this will enable no light to be lost and should provide a fairly accurate approximation in cases where there are many occlusions (or many rooms).

8.5 Useful Applications

In lighting research it is often useful to compute the amount of light at one particular point in a room. Future research will be done which shows a graph of the amount of light in a room across times in a day in one axis and days of the year in the other.

Acknowledgements

To Barb Cutler for the code base as well as direction in the project.

References

ADIGA, N., ALMASI, G., ET AL. 2002. An overview of the bluegene/l supercomputer. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1–22.

GORAL, C. M., TORRANCE, K. E., GREENBERG, D. P., AND BATTAILE, B. 1984. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph. 18*, 3, 213–222.

GRAMA, A., KARYPIS, G., GUPTA, A., AND KUMAR, V. 2003. *Introduction to Parallel Computing: Design and Analysis of Algorithms*, 2nd ed. Addison-Wesley.

STRZLINGER, W., AND WILD, C. 1994. Parallel visibility computations for parallel radiosity. In *Parallel Processing: CONPAR 94 - VAAP VI (Third Joint International Conference on Vector and Parallel Processing)*, B. Buchberger and J. Volkert, Eds., vol. 854 of *Lecture Notes in Computer Science*, 405–413.

STUDDARD, D., WORRAL, A., PADDON, D., AND WILLIS, C. 1995. A parallel radiosity system for large data sets. In *The Third International Conference in Central Europe on Computer Graphics and Visualization 95*, V. Skala, Ed., vol. 2, University of West Bohemia, 421–429.

STÜRZLINGER, W., SCHAUFLER, G., AND VOLKERT, J. 1995. Load balancing for a parallel radiosity algorithm. In *PRS '95: Proceedings of the IEEE symposium on Parallel rendering*, ACM, New York, NY, USA, 39–45.

Figure 7: *Rendering of 8.5k face mesh, radiance values interpolated across faces*

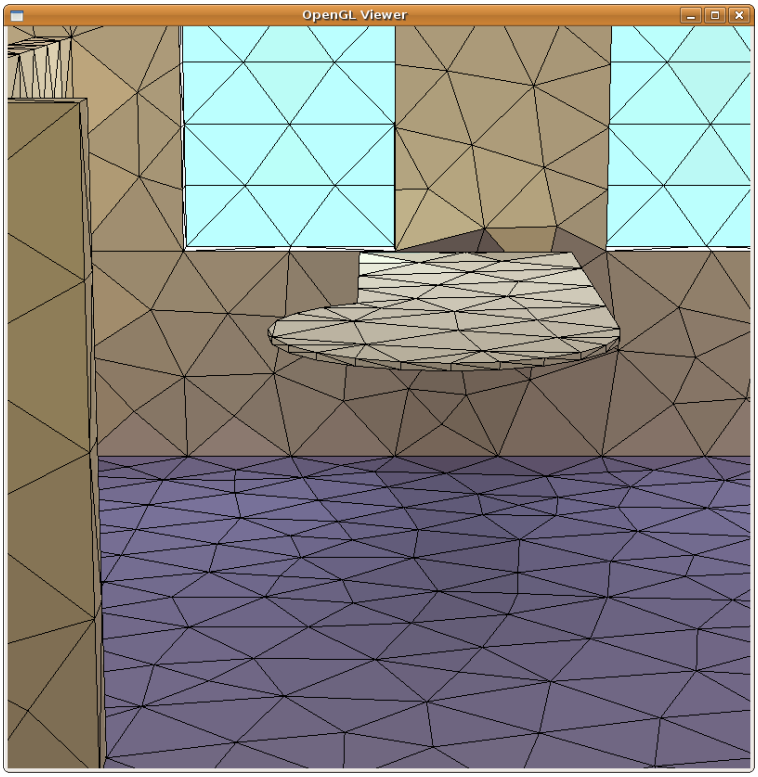

Figure 8: *Rendering of 8.5k face mesh, wireframe view*