Graphics Hardware (2003)
M. Doggett, W. Heidrich, W. Mark, A. Schilling (Editors)

GPU Algorithms for Radiosity and Subsurface Scattering

Nathan A. Carr, Jesse D. Hall and John C. Hart

Dept. of Computer Science, University of Illinois, Urbana-Champaign

Abstract

We capitalize on recent advances in modern programmable graphics hardware, originally designed to support advanced local illumination models for shading, to instead perform two different kinds of global illumination models for light transport. We first use the new floating-point texture map formats to find matrix radiosity solutions for light transport in a diffuse environment, and use this example to investigate the differences between GPU and CPU performance on matrix operations. We then examine multiple-scattering subsurface light transport, which can be modeled to resemble a single radiosity gathering step. We use a multiresolution meshed atlas to organize a hierarchy of precomputed subsurface links, and devise a three-pass GPU algorithm to render in real time the subsurface-scattered illumination of an object, with dynamic lighting and viewing.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Subsurface Scattering

1. Introduction

The programmable shading units of graphics processors designed for z-buffered textured triangle rasterization have transformed the GPU into general purpose streaming processors, capable of performing a wider variety of graphics, scientific and general purpose processing.

A key challenge in the generalization of the GPU to non-rasterization applications is the mapping of well-known algorithms to the streaming execution model and limited resources of the GPU. Some popular graphics and scientific algorithms and data structures, such as ray tracing [19, 3] as well as preconditioned conjugate gradient and multigrid solvers [2], have nevertheless been implemented on, or at least accelerated by, the GPU. This paper expands the horizon of photorealistic rendering algorithms that the GPU can accelerate to include matrix radiosity and subsurface scattering, and describes how the techniques could eventually lead to a GPU implementation of hierarchical radiosity.

Matrix radiosity is a classic technique for simulating light transport in diffuse scenes [7]. It is capable of synthesizing and depicting the lighting, soft shadowing and color bleeding found in scenes of diffuse materials. Our mapping of radiosity to the GPU uses the floating-point texture format to hold the radiosity matrix, but also pays attention to cache coherence and the order of computation to efficiently perform a Jacobi iteration which gathers radiosities as it iterates toward a solution. Given precomputed form factors, we are thus able to both compute and display a radiosity solution entirely on the GPU. While the geometry is fixed, the emittance is not, and our GPU algorithm can support dynamic relighting as well as dynamic alteration of patch reflectances.

Lensch *et al.*[15] shows how the BSSRDF formulation of subsurface (multiple diffuse) scattering[12] resembles a single radiosity gathering step. Light transport in the object interior is computed by gathering, for each patch, the diffused image of the Fresnel transmitted irradiances from the other patches. The BSSRDF can be integrated to form a *throughput factor*[15] that re-

sembles a form factor. We use this similarity to bridge our GPU implementation of matrix radiosity into a GPU implementation of subsurface scattering.

Subsurface scattering can be computed more efficiently through a multiresolution approach. Whereas others have used an octree representation[12], we have opted for a MIP-mappable texture atlas representation[4] of surface radiosities. This representation allows the BSSRDF to be integrated over the surface of an object by applying a fragment program to the appropriate level of the MIP-mapped atlas. This results in a GPU implementation of multiresolution subsurface multiple diffuse scattering that runs in real time (61Hz) on a mesh with 7K faces.

2. Previous Work

2.1. Radiosity

Early radiosity techniques beginning with Goral *et al.*[7] required computationally intensive solutions, which led to numerous acceleration strategies including shooting[5] and overrelaxation. Though form factor computation is generally considered the bottleneck of radiosity techniques, hardware accelerated methods such as the hemicube[6] have been available for a long time, though recent improvements exist[17].

Graphics researchers have looked to hardware acceleration of radiosity long before commodity graphics processors became programmable[1]. For example, Keller[14] used hardware-accelerated OpenGL to accelerate radiosity computations. He performed a quasi-Monte Carlo particle simulation for light transport on the CPU. He then placed OpenGL point light sources at the particle positions, setting their power to the radiosity represented by the particle's power along the random walk path. He then used OpenGL to render the direct illumination due to these particles, integrating their contribution with an accumulation buffer.

Martin *et al.*[16] computed a coarse-level hierarchical radiosity solution on the CPU, and used graphics hardware to refine the solution by texture mapping the residual.

In each of these cases, graphics hardware is used to accelerate elements of the radiosity solution, but the bulk of the processing occurs on the CPU. Our goal is to port the bulk of the radiosity solution process to the GPU, using the CPU for preprocessing.

2.2. Subsurface Scattering

Hanrahan and Krueger[9] showed subsurface scattering to be an important phenomena when rendering translucent surfaces, and used a path tracing simulation to render skin and leaves. Pharr *et al.*[18] extended these techniques into a general Monte-Carlo ray tracing framework. Jensen and Buhler[13] used a dipole and diffusion to approximate multiple scattering. These methods made clear the importance of subsurface scattering to the graphics community, and led some to consider additional approximations and accelerations.

Jensen *et al.*[12] accelerates subsurface scattering using a hierarchical approach consisting of several passes. Their first pass finds surface irradiances from external light whereas the second pass transfers these irradiances to the other surface patches. Their hierarchical approach uses an octree to volumetrically represent the scattered irradiances in the interior of the translucent substance.

Hao *et al.*[11] approximated subsurface scattering using only local illumination, by bleeding illumination from neighboring vertices to approximate local back scattering. They precompute a scattering term for source lighting expressed in a piecewise linear basis. They reconstructed scattering per-vertex from directional light by linearly interpolating these terms computed from the nearest surrounding samples. Their technique was implemented on the CPU but achieved real-time rendering speeds.

Sloan *et al.*[21] uses a similar precomputation strategy, though using a spherical harmonic basis for source lighting. They precomputed per-vertex a transfer matrix of spherical harmonic coefficients from environment-mapped incoming radiance to per-vertex exit radiance that includes the effects of intra-object shadowing and interreflection in addition to subsurface scattering. They were able to compress these large transfer matrices in order to implement the evaluation of precomputed radiance transfer entirely on the GPU to achieve real-time frame rates.

Lensch *et al.*[15] approximated back scattering by filtering the incident illumination stored in a texture atlas. The shape of these kernels is surface dependent and precomputed before lighting is applied. They also approximated forward scattering by precomputing a vertex-to-vertex throughput factor, which resembles a form factor. Forward scattering is rendered by performing a step similar to radiosity gathering, by collecting for a given vertex the irradiance from the other vertices scaled by the throughput factor.

While Lensch *et al.*[15] benefited from some hardware acceleration, for example using a vertex shader to accumulate external irradiances, the application of the vertex-to-vertex throughput factors to estimate forward scattering, and the atlas kernel filtering to estimate backward scattering is performed on the CPU

(actually a pair of CPUs). They expressed a desire to explore full GPU implementations of these techniques, and our GPU implementation of subsurface scattering is an extension of their technique. But as we have discovered and documented in this paper, their technique requires novel extensions to be efficiently implemented given the limited resources of a modern GPU.

2.3. Texture Atlas Generation

A texture atlas is a one-to-one mapping from an object surface into a 2-D texture map image. Texture atlases arise from a parameterization of an object surface, and provide a mechanism for navigating complex surfaces by navigating its charts in a flat texture space. The texture atlas also provides method for applying the pixel shader processors of a GPU to a sampling of an entire object surface (instead of just the visible portions of the surface).

Following Carr and Hart[4], we use a multiresolution meshed atlas (MMA) to distribute available texture samples evenly across the entire object surface. Unlike more common atlases, the MMA is discrete, packing each triangle individually into the texture map, independently from its neighbors. Seam artifacts that can occur when a triangle's sample is inadvertently drawn from a neighboring texel in the texture map are avoided by carefully matching the rules of rasterization with the rules of texture sampling. A half-pixel gutter surrounds each texture map triangle to support bilinear texture interpolation to avoid magnification aliasing.

The MMA is based on a binary face clustering where each node in a binary tree represents a simply connected cluster of mesh triangles. The two children of a node represent a disjoint partitioning of the parent cluster into two simply connected (neighboring) subsets. Leaf nodes correspond to individual surface mesh triangles. Hence the binary tree contains exactly $2T-1$ nodes, where T is the number of triangles in our mesh.

The MMA creates a correspondence between each triangle cluster corresponding to a node in the binary tree with a rectangular (either 2:1 or square) region of the texture map. Hence the face cluster hierarchy is packed as a quadtree hierarchy in the texture map. This organization allows the texture map to be MIP-mapped such that a each texel value (color) in a lower-resolution level of the MIP-map corresponds to an average cluster value (color) in the mesh. Other MIP-mappable atlas methods exist[20], but the MMA more completely utilizes available texture samples. This MIP-map allows the MMA to reduce minification aliasing, as well as providing a data structure to store and quickly recall precomputed sums or averages over mesh regions.

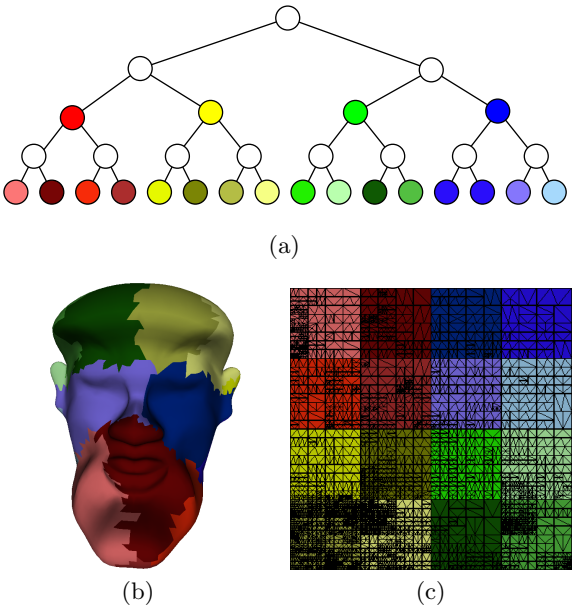


Figure 1: *A multiresolution meshed atlas of a cow. Each node in the binary tree (a) corresponds to a cluster of triangles in the model (b) and a rectangular region in the texture domain (c). Each cluster/region is the union of its corresponding node's children's cluster/region.*

3. Matrix Radiosity

The recent support for floating point texture formats provides an obvious application to matrix problems. Matrix radiosity[7] expresses the illumination of a closed diffuse environment as a linear system

$$B_i = E_i + \rho_i \sum_{j=1}^{N} F_{ij} B_j \qquad (1)$$

where B is a column vector of N radiosities, E are the emittances, ρ are the diffuse reflectivities and F_{ij} is the form factor. This system is solved in $Ax = b$ form as

$$MB = E \qquad (2)$$

where M is a matrix formed by the reflectivities and form factors.

A variety of techniques have been used to solve (2) in computer graphics, including standard numerical techniques such as Jacobi and Gauss-Seidel iteration. Gauss-Seidel is typically preferred because it requires less space and converges about twice as fast as Jacobi. However, Gauss-Seidel is inherently serial, whereas Jacobi iteration offers significant parallelism, and hence takes better advantage of the streaming na-

ture of GPU architectures. The typical Jacobi iteration formula is

$$B_i^{(k+1)} = E_i - \sum_{j \neq i} M_{ij} \frac{B_j^{(k)}}{M_{ii}} . \quad (3)$$

To avoid the conditional execution caused by the $j \neq i$ condition (which cannot be done efficiently with current fragment processing architectures), we do the full matrix-vector product:

$$B^{(k+1)} = B^{(k)} + E - \mathrm{diag}(M)^{-1} M B^{(k)} . \quad (4)$$

The matrix $\mathrm{diag}(M)^{-1}$ is the inverse of the matrix containing only the diagonal elements of M. (For constant elements, as are typically used, $\mathrm{diag}(M) = I$, and the denominator also drops out of (3).)

We used one texel per matrix element. The radiosity system is computed per wavelength of light, so an RGB texture can simultaneously represent a red, green, and blue matrix. When solving a single-channel luminance (as opposed to RGB) matrix system, one can pack the vectors and matrices into all four color channels, such that four elements of a vector (or four columns of one row of a row-major matrix) can be accessed in a single texture lookup. We have found this organization works best for matrix-vector products, whereas a block-channel packing (where e.g. the upper-left submatrix is stored in a texture's red channel) works better for single-channel matrix-matrix products[8].

Each pass operates on a block of the matrix-vector product, where the block size is dictated by the number of pixel shader instructions available per pass. For example, our GeForce FX implementation uses a block size of 254 elements, since each four-element texel requires four instructions (two texture fetches, a multiply-add, and an update to the texture coordinates), and the GeForce FX pixel shader supports a maximum of 1024 instructions (there are a few instructions of overhead for each pass). Thus for an N-patch radiosity solution, each Jacobi iteration takes $\lceil N/254 \rceil$ passes.

3.1. Results

We tested the solver on a simple Cornell box without any occluders, since form factor complexity should not significantly affect solution time. Figure 2 demonstrates the solution, which was solved and displayed (without interpolation) entirely on the graphics card, using precomputed form factors. The vector E and the matrix M were formed by a CPU preprocess, and loaded as a 1-D and 2-D texture, respectively. A pixel shader performed the Jacobi iteration to solve for the unknown radiosities B which were also stored as a 1-D texture. The iterations can be observed by displaying the scene using texture coordinates based on the 1-D radiosity texture, though a 1-D texture prevents bilinear interpolation of the displayed radiosities.

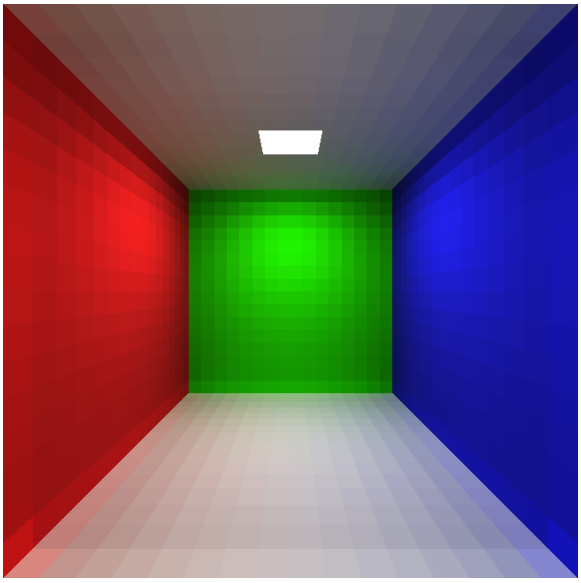


Figure 2: *A simple radiosity scene solved and displayed completely on the GPU.*

In order to smoothly interpolate the displayed radiosities, the 1-D radiosity texture B would need to be resampled into a 2-D displayed radiosity texture. An easy way to perform this resampling is to create an atlas of the scene, such as the meshed atlas described in Section 2.3, and render the texture image of the atlas using 1-D texture coordinates corresponding to each vertex's index in the radiosity vector.

As Figure 3 shows, our GPU Jacobi radiosity implementation takes about twice as many iterations to converge as does a Gauss-Seidel solution on the CPU. Moreover, each Jacobi iteration of our GPU solver (29.7 iterations/sec. on a NVIDIA GeForce FX 5900 Ultra) takes longer to run than a Gauss-Seidel iteration on the CPU (40 iterations/sec. on an AMD Athlon 2800+). This corresponds to 141 MFLOPS/s and 190 MFLOPS/s for the GPU and CPU, respectively (the floating point operations for indexing on the GPU are not included in this number).

We found, however, that the CPU implementation is limited by memory bandwidth, while the GPU implementation is only using a fraction (perhaps as little as 10%) of its available bandwidth. This difference can be observed by comparing the super-linear CPU curve to the linear GPU curve in Figure 4. We believe

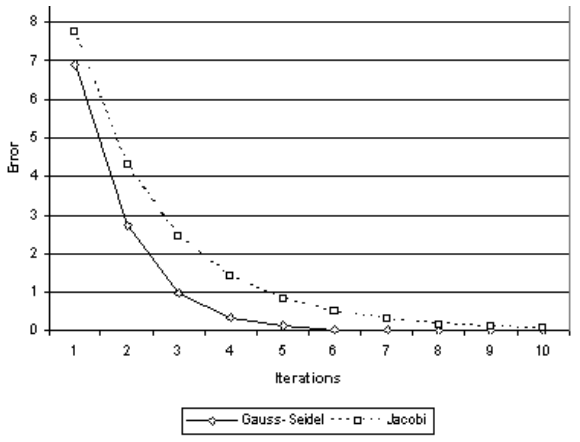


Figure 3: *Gauss-Seidel converges faster than Jacobi iteration. Error is measured as the mean squared error of the residual $MB - E$.*

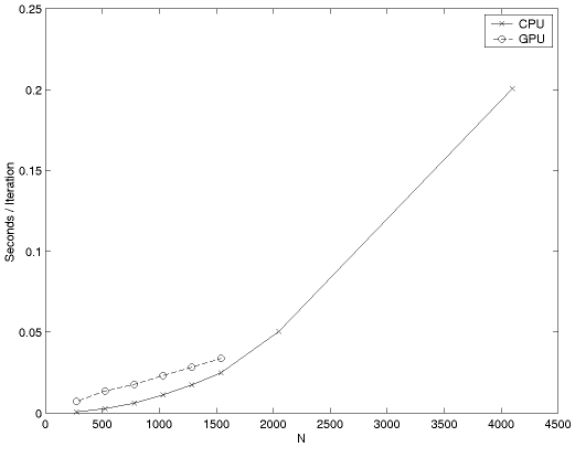


Figure 4: *Though the CPU currently outperforms the GPU on this matrix radiosity example, the CPU is memory bandwidth bound and its performance degrades on larger matrices, whereas the GPU is compute bound and its performance scales linearly.*

the GPU curve overtakes the CPU curve for matrices larger than 2000 elements, but we were limited by a maximum single texture resolution of 2048. As computation speeds have historically increased faster than memory bandwidth, we also expect the GPU will readily outperform the CPU in the near future.

4. Subsurface Scattering

We base our subsurface scattering scheme on the method derived by Jensen *et al.*[13]. The standard rendering equation using a BRDF approximation has been used for many years to model light transport and the reflectance of surfaces. However, the BRDF formulation assumes that light entering a material at a point also leaves the material at the same point. For many real-world surface this approximation is sufficient, but for numerous translucent materials (skin, milk, marble, etc..), much of their appearance comes from internal scattering of light. To account for subsurface scattering, the BRDF is replaced with a more general model known as the BSSRDF (bidirectional surface scattering reflectance distribution function). The amount of radiance leaving a point x_o due to subsurface scattering can be expressed as

$$L_o(x_o, \vec{\omega}_o) = \int_A \int_\Omega S(x_i, \vec{\omega}_i; x_o, \vec{w}_o) L_i(x_i, \vec{\omega}_i)(\vec{n}_i \cdot \vec{\omega}_i) d\vec{\omega}_i dA(x_i). \quad (5)$$

Integration is performed over the surface A at points x_i and all incident light directions. The term S is the BSSRDF, which relates the amount of outgoing radiance at point x_o in direction $\vec{\omega}_o$, given that there is incoming radiance at some other surface point x_i in direction $\vec{\omega}_o$.

Jensen et. al. noted that single scattering for many common materials only accounts for a small percentage of the outgoing radiance of a material[12]. Also, light tends to become heavily scattered when it enters materials, removing the relationship between incident and exitant light directions. This simplifies the BSSRDF to a four dimensional function R_d known as the diffuse BSSRDF. Reformulating the subsurface scattering equation (5) to only account for subsurface scattering we have

$$L_o(x_o, \vec{\omega}_o) = \frac{1}{\pi} F_t(\eta, \vec{\omega}_o) B(x_o) \quad (6)$$

$$B(x_o) = \int_{x_i \in S} E(x_i) R_d(x_i, x_o) dA(x_i) \quad (7)$$

$$E(x_i) = \int_\Omega L_i(x_i, \vec{\omega}_i) F_t(\eta, \vec{\omega}_i) |\vec{n}_i \cdot \vec{\omega}_i| d\vec{\omega}_i \quad (8)$$

where $R_d(x_i, x_o)$ is the diffuse subsurface scattering reflectance. It encodes geometric information and also the volumetric material properties anywhere in the object dealing with light transport from x_i to x_o. For R_d we use the dipole approximation model detailed by Jensen et. al. [12], and also used in Lensch [15]. Also found in Jensen et. al. are the scattering and absorption coefficients: σ'_s, σ_a for common materials that are used in this model. For brevity we have omitted the details here.

Lensch et. al. noted this strong similarity between the radiosity formula and the formula in (8). This equation may be solved by discretizing the scene into patches.

4.1. Real-Time Subsurface Approximation Algorithm

We start by discretizing our object into a collection of N patches where P_i and P_j are patches on the surface S. We can reformulate (8) into its discretized form as follows:

$$B_i = \sum_{j=1}^{N} F_{ij} E_j \quad (9)$$

which resembles a single transport step of radiosity transport (1). The multiple diffuse scattering throughput factor F_{ij} is expressed as:

$$F_{ij} = \frac{1}{A_i} \int_{x_i \in P_i} \int_{x_j \in P_j} R_d(x_j, x_i) dx_j dx_i. \quad (10)$$

For a static model, we can precompute the F_{ij} factors between all pairs of patches. Using (9), the radiosity due to diffuse multiple scattering now reduces to a simple inner product for each patch resulting in $O(N^2)$ operations to compute the incident scattered irradiance for all patches.

A simple way to reduce the number of interactions is to turn to a clustering strategy like that used to solve the N-body problem and hierarchical radiosity[10]. This is particularly applicable to subsurface scattering since the amount of scattered radiance drops rapidly as it travels further through the media. This implies that patches that are far away from the patch whose scattered radiosity we are computing may be clustered together and be replaced by a single term to approximate the interaction.

4.2. A Three Pass GPU Method

We now formalize a solution the diffuse subsurface scattering equation (9) as a three pass GPU scheme (as shown in Fig. 5) preceded by a pre-computation phase. By assuming that our geometry is not deforming we are able to precompute all of our throughput factors F_{ij} between interacting surfaces.

Our first pass to the GPU computes the amount of light incident on every patch of our model, and scales this by the Fresnel term, storing the radiosity that each patch emits internal to the object. This map forms our *radiosity map*.

Our second pass acts as a gathering phase evaluating equation (9). For every patch/texel the transmitted radiosity is gathered and scaled by the precomputed throughput factors and stored into a *scattered irradiance map*.

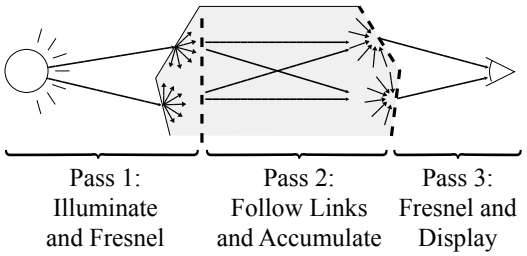


Figure 5: *Three passes for rendering subsurface scattering on the GPU.*

In our third and final pass we render our geometry to the screen using the standard OpenGL lighting model. The contribution from subsurface scattered light is added in by applying the scattered irradiance texture map to the surface of the object scaled by the Fresnel term.

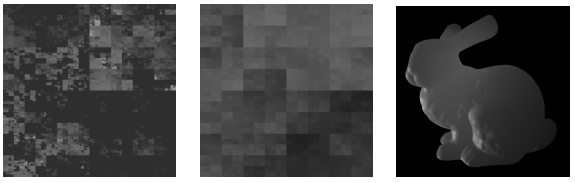


Figure 6: *Pass one plots triangles using their texture coordinates (left), interpolating vertex colors set to their direct illumination scaled by a Fresnel transmission term. Pass two transfers these radiances (center) via precomputed scattering links implemented as dependent texture lookups into a MIP-map of the previous pass result (left). Pass three scales the resulting radiances by a Fresnel transmission factor and texture maps the result onto the original model.*

4.2.1. Pre-computation Phase

We start by forming a hierarchical disjoint face clustering and texture atlas parameterization of our mesh with a method previously developed for real-time procedural solid texturing[4]. Every texel in our texture atlas corresponds to a patch on the surface of our model. This parameterization method is ideal for a GPU solution to the subsurface scattering problem for a number of reasons. First, it provides a natural face cluster hierarchy necessary for a hierarchical approach. Secondly, it works directly with the GPU rasterization rules allowing the GPU to perform surface computation in a seam-free manner. Thirdly, it is MIP-mappable, allowing the GPU to compute fast average and sums

over the surface hierarchy. Lastly, by using a parameterization scheme as a domain to store and compute our surface samples, the number of surface samples is independent of both the tessellation of our geometry, and the resolution of the screen we are rendering to. This marks a difference between earlier interactive subsurface scattering approaches where vertex to vertex interactions were used to discretize the scattering equation.

Full evaluation of equation (9) of patch to patch throughput factors would require P^2 interactions, where P is the number of patches (and also texels in our texture atlas). Each interaction forms a link. Since all of our patches are in texture space, we need only store a u, v location and a throughput factor for each link. By adding an LOD term into the link structure we can access any level in the MIP-map surface hierarchy. Based on our computation and space restrictions we assign some maximum number of links L that we store for each patch P_b in the base level of our texture map.

For a non-adaptive approach we can choose some level l in the hierarchy. We then build all links from patches P_b to P_l where P_b are the patches at the lowest level in the hierarchy, and P_l are patches at level l in the hierarchy.

An adaptive top-down approach for the construction of links may be done similar to that of hierarchical radiosity. For every patch in the lowest level of our hierarch P_b, we start by placing the root patch P_r of our hierarchy onto a priority queue (with highest throughput factor at the top). We then recursively remove the patch at the top of the queue, compute the throughput factors from P_b to its four children, and insert the four children into the priority queue. This process is repeated until the queue size grows to the threshold number of links desired.

Once L adaptive links for each patch/texel in our atlas have been created, we store the result into L textures maps that are $\sqrt{P_b} \times \sqrt{P_b}$ in size. We use an fp16 (16-bit floating point) texture format supported on the GeForceFX to pack the link information: u, v, F_{ij},LOD into the four color channels to be used during pass two of our rendering stage. To reduce storage in favor of more links, we opted to reduce our form factor to a single scalar. Form factors per color channel may be supported at a cost of space and bandwidth.

In the case of a non-adaptive approach, the link locations and LOD are the same for every patch P_b, we therefore store this information in constant registers on the graphics card during the second pass. The throughput factors, however, vary per-link. We store the form factor information into $L/4$ texture maps where each texel holds four throughput factors (corresponding to 4 links) in the rgbα channels.

4.2.2. Pass 1: Radiosity Map

Given our face cluster hierarchy and MIP-mappable parameterization, we must first compute the E_j's for the patches in our scene. To do this, we start by computing a single incident irradiance for every texel in our texture atlas, thereby evaluating lighting incident on all sides of the model. We scale the result of the incident illumination by the Fresnel term, storing the result in the texture atlas. Each texel now holds the amount of irradiance that is transferred through the interface to the inside of the model. This step is similar to the *illumination map* used in Lensch *et al.*[15].

To accomplish this efficiently on the GPU, we use the standard OpenGL lighting model in a vertex program on the GPU. Using the OpenGL render-to-texture facility, we send our geometry down the graphics pipeline. The vertex program computes the lighting on the vertices scaled by the Fresnel term placing it in the color channel and swaps the texture coordinates for the vertices on output. The radiosity stored in the color channel is then linearly interpolated as the triangle gets rendered into the texture atlas. Our method does not prevent the use of more advanced lighting models and techniques for evaluating the incident irradiance on the surface the object.

As an alternative to computing our transmitted radiosity in a vertex program, we could perform the computation entirely in a fragment program per-texel. In addition to higher accuracy, this approach may have improved performance for high triangle count models. A geometry image (e.g. every texel stores surface position) and a normal map may be precomputed for our object and stored as textures on the GPU. Rendering the radiosity map would only entail sending a single quadrilateral down the graphics pipeline texture mapped with the geometry image and normal map. The lighting model and Fresnel computation can take place directly in the fragment shader.

We use the automatic MIP-mapping feature available in recent OpenGL version to compute the average radiosity at all levels of our surface hierarchy. The radiosity map is then used in the next pass of this process.

4.2.3. Pass 2: Scattered Irradiance Map

This pass involves evaluating equation (9) for every texel in our texture atlas. We render directly to the texture atlas, by sending a single quadrilateral to the GPU. In a fragment program, for each texel, we traverse the L links stored during the pre-computation

phase. Texture lookups are performed to get the link information. Using the u,v and LOD link information a dependent texture lookup is performed on the mip-mapped Radiosity Map. The result of this is scaled by the links form factor. All links are traversed for a given texel and the accumulated radiosities form the incident scattered irradiance for the texel.

This pass can be the most costly of the three passes depending on the number of links chosen. For our adaptive approach, $2*L$ texture lookups must be performed in the fragment shader. For our non-adaptive scheme we were able to pack four links into a single texture lookup resulting in $1.25 * L$ lookups.

4.2.4. Pass 3: Final Rendering

Pass three begins with the scattered irradiance map resulting from pass two. This pass multiplies the scattered irradiance texture with a texture of inverted Fresnel terms to get a texture of external radiosities on the outside of the translucent object's surface. This texture product is further modulated by direct illumination resulting from a standard OpenGL lighting pass, evaluated either per-vertex or per-fragment. This results in the final rendering of the translucent object.

4.3. Subsurface Scattering Results

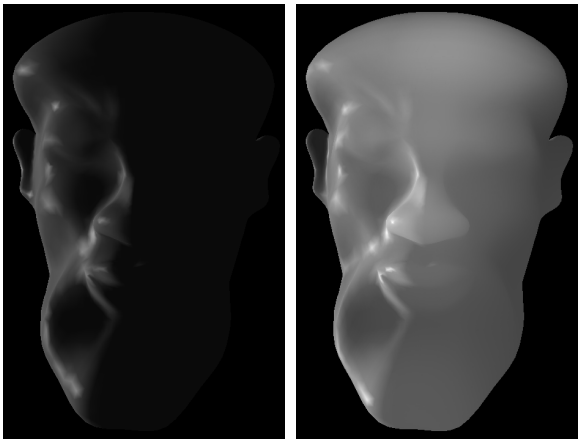
Figure 7 shows the result of our subsurface scattering algorithm running on a GeForce FX card. To achieve real-time performance we are running with either a 512^2 or 1024^2 texture atlas with 16 links per texel. We initially tried to use our adaptive approach for assigning links but found that the adaptivity when using such a course number of links led to visible discontinuities during rendering along the mip-map boundaries. As we increased the number of links, the seams diminished due to the improved approximation.

Precomputation of the links was performed entirely on the CPU and took approximately nine seconds for the 1024^2 resolution.

Model	res.	fps	Pass 1	Pass 2	Pass 3
Head	512^2	61.10	11%	82%	7%
Dolphin	512^2	30.35	43%	43%	14%
Bunny	512^2	30.33	37%	34%	28%
Head	1024^2	15.40	13%	85%	2%
Dolphin	1024^2	15.09	8%	85%	7%
Bunny	1024^2	12.05	18%	68%	14%

Table 1: *Subsurface scattering performance on a GeForce FX 5900.*

We tested the algorithm using an NVidia GeForce FX 5900 on three models. The "head" model with 7,232 faces, a "dolphin" model with 21,952 faces and the "bunny" model with 69,451 faces. The results of the GPU subsurface scattering algorithm is shown in Table 1. At the 1024^2 texture resolution, Pass 2 dominates, and since this pass is a texture-to-texture pass, the use of a texture atlas has effectively decoupled the total shading cost from the tesselation complexity.



(a) (a)

Figure 7: *Without subsurface scattering (a), and with subsurface scattering (b) using* $\sigma_s' = 2.21$, $\sigma_a = 0.0012$, *16 links, 40fps Nvidia GeForceFX 5800.*

5. Conclusion

We have examined the implementation of matrix radiosity on the GPU and used it as an example to examine the performance of the GPU on scientific applications, specifically those involving linear systems.

Our GPU subsurface scattering result is much more successful, yielding full real-time (61 Hz) performance on a present day GPU. Our implementation is novel compared to other real-time subsurface scattering results that are implemented primarily on the CPU. Moreover, our subsurface scattering method is based on a multiresolution meshed atlas, and this multiresolution approach applied to a surface cluster hierarchy is also novel.

The key to our multiresolution surface approach to subsurface multiple diffuse scattering is the assignments of gathering links. These directional links are formed between clusters at different levels in the hierarchy, and a directed link indicates the irradiance over one cluster is gathered from the scattered radiosity of another cluster.

These links are similar to the links used for hierarchical radiosity. Hierarchical radiosity would follow

these links to gather radiosities from other clusters at appropriate levels in the radiosity MIP-map, and radiosities formed at one level would be averaged or subdivided to form radiosities at other MIP-map levels. But hierarchical radiosity gains its greatest speed and accuracy improvement from its ability to dynamically re-allocate these links based on changes in the cluster-to-cluster form factors. The implementation of this dynamic link reassignment on the GPU is of great interest but appears quite challenging, and forms the primary focus of our future work.

Acknowledgments This work was supported in part by NVIDIA and the NSF under the ITR grant #ACI-0113968.

References

1. D. R. Baum and J. M. Winget. Real time radiosity through parallel processing and hardware acceleration. *Computer Graphics*, 24(2):67–75, 1990. (Proc. Interactive 3D Graphics 1990).
2. J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Trans. on Graphics*, 22(3):to appear, July 2003. (Proc. SIGGRAPH 2003).
3. N. A. Carr, J. D. Hall, and J. C. Hart. The ray engine. In *Proc. Graphics Hardware 2002*, pages 37–46, Sep. 2002.
4. N. A. Carr and J. C. Hart. Meshed atlases for real-time procedural solid texturing. *ACM Transactions on Graphics*, 21(2):106–131, 2002.
5. M. F. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg. A progressive refinement approach to fast radiosity image generation. *Computer Graphics*, 22(4):75–84, 1988. (Proc. SIGGRAPH 88).
6. M. F. Cohen and D. P. Greenberg. The hemicube: A radiosity solution for complex environments. *Computer Graphics*, 19(3):31–40, 1985. (Proc. SIGGRAPH 85).
7. C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modelling the interaction of light between diffuse surfaces. *Computer Graphics*, 18(3):213–222, 1984. (Proc. SIGGRAPH 84).
8. J. D. Hall, N. A. Carr, and J. C. Hart. Cache and bandwidth aware matrix multiplication on the GPU. Technical Report UIUCDCS-R-2003-2328, University of Illinois, Apr. 2003. At ftp.cs.uiuc.edu in /pub/dept/tech_reports/2003/ as UIUCDCS-R-2003-2328.ps.gz.
9. P. Hanrahan and W. Krueger. Reflection from layered surfaces due to subsurface scattering. In *Proc. SIGGRAPH 93*, pages 165–174, 1993.
10. P. Hanrahan, D. Salzman, and L. Aupperle. A rapid hierarchical radiosity algorithm. *Computer Graphics*, 25(4):197–206, July 1991. (Proc. SIGGRAPH 91).
11. X. Hao, T. Baby, and A. Varshney. Interactive subsurface scattering for translucent meshes. In *Proc. Interactive 3D Graphics*, pages 75–82, 2003.
12. H. W. Jensen and J. Buhler. A rapid hierarchical rendering technique for translucent materials. *ACM Trans. on Graphics*, 21(3):576–581, 2002. (Proc. SIGGRAPH 2002).
13. H. W. Jensen, S. R. Marschner, M. Levoy, and P. Hanrahan. A practical model for subsurface light transport. In *Proc. SIGGRAPH 2001*, pages 511–518, 2001.
14. A. Keller. Instant radiosity. In *Proc. SIGGRAPH 97*, pages 49–56, 1997.
15. H. P. A. Lensch, M. Goesele, P. Bekaert, J. Kautz, M. A. Magnor, J. Lang, and H.-P. Seidel. Interactive rendering of translucent objects. In *Proc. Pacific Graphics 2002*, pages 214–224, Oct. 2002.
16. I. Martin, X. Pueyo, and D. Tost. A two-pass hardware-based method for hierarchical radiosity. *Computer Graphics Forum*, 17(3):159–164, Sep. 1998.
17. K. H. Nielsen and N. J. Christensen. Fast texture-based form factor calculations for radiosity using graphics hardware. *J. of Graphics Tools*, 6(4):1–12, 2001.
18. M. Pharr and P. Hanrahan. Monte Carlo evaluation of non-linear scattering equations for subsurface reflection. In *Proc. SIGGRAPH 2000*, pages 75–84, 2000.
19. T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. on Graphics*, 21(3):703–712, July 2002. (Proc. SIGGRAPH 2002).
20. P. V. Sander, J. Snyder, S. J. Gortler, and H. Hoppe. Texture mapping progressive meshes. In *Proc. ACM SIGGRAPH 2001*, pages 409–416, 2001.
21. P.-P. Sloan, J. Hall, J. C. Hart, and J. Snyder. Clustered principal components for precomputed radiance transfer. *ACM Trans. on Graphics*, 22(3):to appear, July 2003. (Proc. SIGGRAPH 2003).