Radiosity on Graphics Hardware

Greg Coombe Mark J. Harris Anselmo Lastra

Department of Computer Science, University of North Carolina, Chapel Hill, North Carolina, USA
{coombe, harrism, lastra}@cs.unc.edu

Abstract

Radiosity is a widely used technique for global illumination. The computation is typically done offline, but the result is a model suitable for real-time display. We present a technique for computing radiosity, including an adaptive subdivision of the model, using graphics hardware. Since our goal is to make the solution run at interactive rates, we exploit the computational power and programmable functionality of recent graphics hardware. Using our system on current hardware, we have been able to compute a full radiosity solution for a small scene in less than one second.

Keywords: **Graphics Hardware, Global Illumination.**

1 Introduction

Progressive refinement radiosity [Cohen et al. 1988] is a common technique for computing a global illumination solution of a diffuse environment. We have selected this technique for a graphics-hardware implementation for the same reasons that it is popular on the CPU: it requires no explicit storage of the radiosity matrix and the model can be displayed as the solution progresses. We first present an implementation equivalent to a uniform subdivision of the model, and then describe an extension that uses adaptive subdivision.

Almost all of the computation is done on the graphics hardware; there is minimal readback (only one pixel per shooting pass). The host CPU only handles bookkeeping and data structure management. There are several reasons for performing all of the computation on the GPU (Graphics Processing Unit). One reason is that the GPU can compute the radiosity values directly in textures and use them immediately for fast rendering. This is useful in many real-time systems because it frees up the CPU for other computationally intensive tasks such as collision detection or physics. Another reason is that the performance of graphics hardware is increasing faster than that of the main CPU, so our goal of radiosity at interactive rates is easier to achieve.

Previous generations of GPUs are not suitable for radiosity computations because radiosity requires high precision computation. As we show in Section 3, computations on the GPUs using integers result in visual artifacts. The floating-point capability of newer GPUs provides the high dynamic range computation required by radiosity.

After reviewing previous work, we describe the basic progressive refinement implementation. For ease of explanation, we first describe the uniform subdivision method. Since uniform subdivision has several drawbacks, we have also implemented an adaptive subdivision method. We then present results, conclusions and future work.

Figure 1. Radiosity solution computed on graphics hardware.

2 Previous Work

The most relevant previous work is that on the hemicube method [Cohen and Greenberg 1985] and the progressive refinement approach for solving the radiosity equations [Cohen et al. 1988]. It would be difficult to cover all of the relevant areas of radiosity in this short paper; a lucid explanation of many aspects of radiosity can be found in [Cohen and Wallace 1993].

Several researchers, including [Baum and Winget 1990; Varshney and Prins 1992; Funkhouser 1996], have made use of parallel processing to speed up radiosity computation. The visibility portion of the hemicube approach maps well to graphics hardware [Cohen and Greenberg 1985]. [Nielsen and Christensen 2002] use textures to uniformly subdivide polygons and compute form factors. Other authors, such as [Keller 1997], have made use of older (non-programmable) graphics

hardware to accelerate portions of the radiosity computation.

Programmability has opened up a new area of research in hardware-accelerated global illumination algorithms. [Purcell et al. 2002] demonstrated a ray-tracer running entirely on graphics hardware. This technique bypassed the traditional graphics pipeline in favor of the powerful programmable fragment processor. [Carr et al. 2002] used the hardware as a ray-triangle intersector and used the CPU to generate rays.

3 Progressive Radiosity on Graphics Hardware

In this section we describe a hardware radiosity implementation that is equivalent to conventional patch-oriented radiosity on a uniformly subdivided domain. Our system stores the radiosity and the residual as textures on the modeling polygons. Each texel represents an element of the radiosity solution and is also used for rendering as described by [Bastos et al. 1997]. The algorithm is as follows.

```
foreach(i)
   B_i = r_i = E_i;
   while (not converged) {
      // Choose highest energy element as shooter
      i = element with max(r_i * area_i);
      // Build item buffer on hemicube
      RenderHemicube(i);
      // traverse all other elements in scene
      foreach(texel j ∉ i) {
         if (j is visible from i) {
            // Update radiosity(j) and residual(j)
            ΔB = r_i * ρ_j * F_ji;          (1)
            r_j += ΔB;
            B_j += ΔB;
         }
      }
      // zero shooter's residual
      r_i = 0;
   }
```

Here, E is the initial energy of emitters, B is radiosity, r is the residual, and ρ is the reflectance of the material.

3.1 Shooting

In conventional progressive radiosity, the element with the highest residual energy is chosen as the next shooter. We choose the scene polygon with the highest total residual energy. The process for selecting this polygon on hardware is described later in this section. All of the elements (texels) on the polygon shoot their energy in turn. As Heckbert [Heckbert 1990] suggested, shooting at a lower resolution can increase performance. We do so by creating a mipmap of the residual texture and using the desired level. For example, by using the first mipmap level above the base texture we can combine 4 texels into a single shot. In our system we typically shoot from two levels above the base resolution.

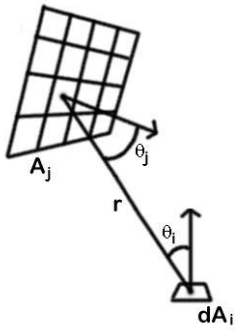

Figure 2. The differential area-to-area form factor computation.

3.2 Visibility

We use the hemicube method to sample visibility from the current shooter. GPUs support reads from arbitrary texture memory locations, but allow only very restricted writes. This means that it is not feasible to implement a feed-forward radiosity solution that traverses the pixels of the hemicube and directly updates the radiosity textures. Feed-backward techniques are more amenable to hardware implementation, so we invert the shooting computation and iterate over the receiving elements.

In the first step (the visibility pass), the scene is rendered with polygon ID as color onto the faces of a hemicube and stored as five textures, which are used as item buffers. For the second step (the reconstruction pass), each polygon that is potentially visible (not back-facing) from the shooter is rendered to a frame buffer at the resolution of the radiosity texture. This places the texels of the radiosity texture and the fragments of the frame buffer in a one-to-one correspondence, enabling us to test the visibility of each radiosity texel using a fragment program.

The visibility of the texel from the shooter is tested by back-projecting the texel onto the shooter's hemicube faces. The point of projection onto the hemicube is used to index the hemicube item buffers. If the ID found in the item buffer matches the ID of this texel's polygon, then this texel is declared visible from the shooter. This process closely resembles shadow mapping.

3.3 Form factor

If the element is visible from the shooter, then the next step is to compute the intensity that it receives. We use the disc approximation to a differential area-to-area form factor equation [Wallace et al. 1989]

$$F_{dAi \to Aj} = A_j \sum_{i=1}^{m} \frac{\cos\theta_j \cos\theta_i}{\pi r^2 + A_j / m}, \qquad (2)$$

between two elements i and j (shown in Figure 2). This equation divides the shooter into m subsections and approximates their areas with oriented discs.

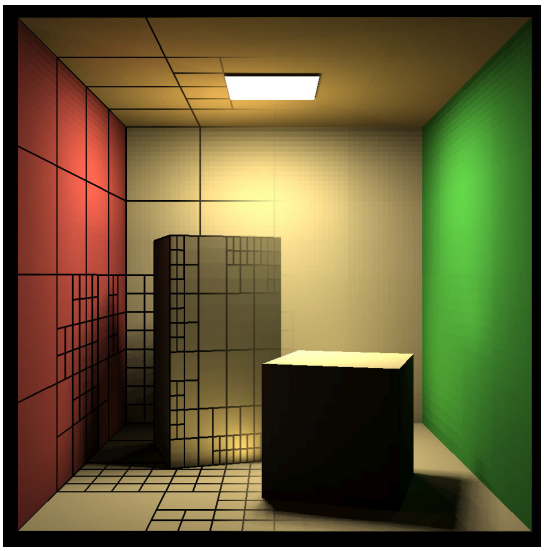

Figure 3. Radiosity solution using adaptive subdivision. The subdivided texture quadtrees are shown on the left.

An assumption implicit in this equation is that the area of the differential element is small compared to r^2. In a system using uniform subdivision, this must be enforced during the selection of the static texture sizes, which can be difficult to ensure. Violating this assumption results in artifacts, particularly at places such as the corners of rooms. Using the disc approximation instead of the standard differential area-to-area form factor greatly reduces these artifacts.

Equation (2) is computed at every texel in a fragment program, which also computes the delta radiosity term using equation (1). The energy, E, of the emitter and its reflectance, ρ, are parameters to the program.

One of the advantages of texel-based computation is that it allows us to determine the intensity at the resolution of the receiving textures rather than at the resolution of the hemicube. This eliminates blocky hemicube aliasing artifacts [Wallace et al. 1989], but there are still other hemicube aliasing problems such as missing small features.

3.4 Next shooter selection

Progressive refinement techniques achieve fast convergence by always shooting from the scene element with the highest residual power. To find the next shooter, we use a simple z-buffer maximum algorithm. First, we set up a one-pixel frame buffer. A fragment program reads the top (1x1) mipmap level of the residual texture, which contains the average of all texels. The sum of the radiosity is this average radiosity multiplied by the base texture resolution. Since progressive refinement uses the total residual power, the total radiosity is multiplied by the area of the polygon. The reciprocal of this value is rendered as depth into the frame buffer and the polygon ID is rendered as color. In this way (with the z-buffer set to "nearest") we can select the polygon with the highest residual power. By reading back the single pixel frame buffer we can get the ID of the next shooter. In addition, convergence can be tested by setting the far clipping plane distance to the reciprocal of the convergence threshold. If no fragments pass, then the solution has converged.

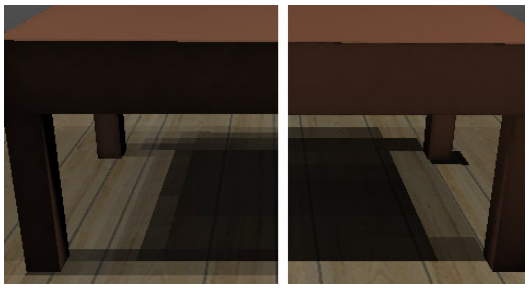

Figure 4. Detail of shadow resolution with an adaptive subdivision (*left*) and a uniform subdivision (*right*).

4 Adaptive Subdivision

As previous researchers have demonstrated, uniform subdivision is not the best approach for radiosity. Too many elements are used on areas of the model with little or smooth variation, while areas with high detail are undersampled. In this section we extend our system to use adaptive subdivision. We use the same hemicube and form factor computations as the uniform approach described in Section 3 .

4.1 Texture quadtrees

In place of the uniform textures described in the previous section, this method subdivides the scene geometry hierarchically into a quadtree with the radiosity data stored at the leaf nodes. Each of the leaf nodes stores two textures, one for residual energy and one for accumulated energy. These leaf textures are small (16x16) textures.

The radiosity is computed on the GPU at the texels of the leaf texture as described in Section 3. Using a fragment program, the gradient of this radiosity texture is computed and evaluated for smoothness. [Vedel and Puech 1992] suggest using the gradient (instead of the value) because it avoids over-subdividing in areas where linear interpolation will adequately represent the function. Evaluating the subdivision criteria on the GPU avoids the penalty of readback.

Fragments with gradient discontinuities are discarded, and a hardware occlusion query[1] is used to count discarded fragments. If any fragments were discarded, we subdivide the node. This process is repeated recursively. The recursion terminates when either the radiosity is found to be smooth, or a user-specified maximum depth is reached.

[1] Occlusion queries simply return a count of pixels rendered. They are supported on most current GPUs.

To reconstruct radiosity from subsequent shooters, we do not rebuild the quadtree from scratch. Instead, we traverse the tree until we reach an existing leaf node. At this node, we compute the radiosity and evaluate the subdivision criteria as before. If the radiosity solution is smooth, the node is not subdivided. If it is not smooth, we subdivide and recur. In this situation, we must "push" the existing radiosity at the former leaf node down to the four children.

4.2 Next shooter selection

Adaptive subdivision with texture quadtrees allows us to easily choose shooters at a higher granularity than the polygons used with uniform textures. We use the z-buffer maximum algorithm described in section 3 , but instead of using the power of the entire polygon, we use the power of each node.

Since each node consists of two small textures, we can gain efficiency by packing node textures into larger *node set* textures. If both the leaf node resolution, R, and the resolution of the node set are a power of two, we can compute mipmaps for every node in the set by computing the first $\log_2(R)$ mipmap levels of the node set.

4.3 Display

To render a texture quadtree, we traverse it recursively, subdividing the geometry at each level. When a leaf node is reached, a quadrilateral is rendered with the node's radiosity texture. By rendering all leaf nodes, we display all scene polygons textured with the adaptive radiosity solution, as shown in figure 3.

Because the leaf nodes in the tree represent different levels of detail, the quadtree subdivision introduces "t-vertices." This causes linear interpolation artifacts between neighboring quadtree nodes, as shown in Figure 5. In our system, we can post-process the quadtree by collapsing it to a flat texture for rendering. This allows us to filter the texture, but consequently increases memory usage. In addition, bilinear filtering is not able to eliminate all of the artifacts, and can introduce Mach bands. [Bastos et al. 1997] present a better solution that used hardware bicubic filtering (available on SGI's) to render their adaptive radiosity solution.

4.4 Implementation

Like most mesh-based subdivision schemes, there are several user-specified variables in the quadtree implementation. In our implementation, we have a gradient subdivision threshold, a maximum quadtree depth, and a residual convergence condition. We typically set the maximum quadtree depth to 4, which would be equivalent to the resolution of a 256x256 texture (assuming a 16x16 node texture resolution). By adjusting this parameter, quality vs. speed can be balanced.

5 Results

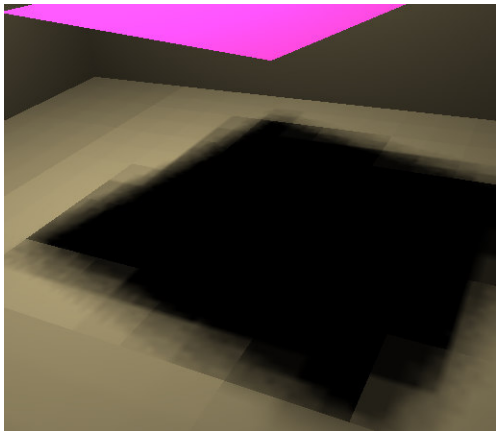

Figure 5. Blocky artifacts in the quadtree are caused by t-vertices.

We have implemented both the uniform and adaptive radiosity algorithms described in this paper. Using texture quadtrees for adaptive radiosity substantially reduces the number of texels processed, while preserving the quality of the solution. Our system allows the use of both uniform textures and texture quadtrees together in a scene. Typically, uniform textures are used for the light sources and small scene polygons, and texture quadtrees are used for the rest of the scene.

Our system was implemented using Cg[cite] on an NVIDIA GeForce FX GPU. Performance numbers for several scenes are shown in Table 1. Somewhat unexpectedly, the adaptive solution is slower than the uniform solution for all but the smallest scenes. There are several possible reasons for this. Copying the node textures may require a context switch, which occurs multiple times per quadtree evaluation (instead of just once for the uniform case). Also, since the subdivision has to test the current node before it can calculate the children nodes, we may be causing pipeline stalls.

In the uniform case, the limiting factor on performance is the number of texels processed. Reducing the number of texels will increase performance. In this paper, we use conservative back face culling, and consequently much of our time is spent processing texels that are not visible. There are a number of visibility pre-processing techniques that we could use.

All computation in our system is done using the floating-point vertex and fragment processors of the GPU. The radiosity values, however, can be stored in either integer or floating-point textures. There is a performance/quality tradeoff between the two. Fixed-point values are cheaper to store and faster to process, but cannot represent the high dynamic range of radiosity solutions. As a result of truncation of very small intensities, fixed-point solutions

	Texels		Secs per shot	
	Uniform	Adaptive	Uniform	Adaptive
Cornell	31K	18K	0.25	0.17
Museum	172K	25K	1.1	1.41
Grotto	1.6M	225K	6.45	10.27

Table 1. Performance of several test scenes.

tend to lose energy and converge too quickly. This causes unrealistically dark images. We use floating-point textures for all of the images in this paper.

6 Conclusions and Future Work

We have presented a progressive refinement radiosity algorithm running on graphics hardware. The adaptive approach is especially promising because it is well suited to dynamic environments, subdividing appropriately as the objects move.

Our next steps are performance optimizations. As we learn more about the capabilities and limitations of the newest generation of graphics hardware, we will find ways to achieve higher raw performance. Another optimization we have explored is the use of vertex programs to perform visibility computations in a single pass by using a true hemispherical projection rather than a hemicube, which requires five rendering passes. The difficulty with this technique is that while straight edges project to curves on a hemisphere, rasterization produces straight edges on screen regardless of how the vertices are projected. One way to reduce this rasterization error is to highly tessellate the surfaces. It may be possible to perform this compensating tessellation on the fly using a GPU that supports curved PN triangles [Vlachos et al. 2001].

Since the visibility process resembles shadow-mapping, we could take advantage of existing research in shadow-mapping, particularly percentage-closer filtering. This technique improves shadow boundaries by filtering after the texture lookup instead of before.

For interactive scenes, importance based methods, such as the one presented by Smits *et al.* [Smits et al. 1992] will be useful because they attempt to focus the solution on areas of importance to the viewer. Based on our adaptive subdivision results, we think it will be beneficial to explore hierarchical radiosity [Hanrahan et al. 1991] on graphics hardware.

One of the disadvantages of radiosity is its inability to handle view-dependent effects. We are investigating extensions to our technique to handle glossy surfaces, perhaps using a method such as those of [Sillion et al. 1991] or [Immel et al. 1986], but the computations and storage may be prohibitive for current hardware. We are also exploring future graphics hardware architectures that directly support non-diffuse global illumination.

6.1 Adaptive Subdivision on Graphics Hardware

Adaptive hierarchies appear in many areas of computer graphics, and the ability to build and use them on graphics hardware may prove useful in a number of applications. Possibilities include terrain rendering, hierarchical visibility processing, subdivision surfaces, and level of detail techniques. Any application that computes and stores data in textures can perform adaptive subdivision on the GPU.

6.2 Hardware Extensions

New features on graphics hardware would increase performance. Mipmap filtering of floating-point textures would help with the computation of the maximum energy to find the next shooter. The addition of min, max, and sum operations to the mipmap generation hardware would be even better.

The pipelined hardware occlusion query is very useful, but could be improved by making it more general. This query could be set to perform a particular action, such as comparison or summation. A register would hold the results of this operation and return the value when queried. This would allow us to sum the values of all of the elements in a texture, which we currently have to do in multiple passes using the mipmap hardware. There are several other applications, such as stochastic iterative searches and conjugate gradient computation that need a single value computed over an entire texture. This would also allow accumulation of fragment statistics while still allowing the fragments to be written.

The visibility test, which projects texels into the hemicube, closely resembles shadow mapping. The difference is that instead of comparing depth, we are comparing the polygon ID. If the shadow-mapping hardware included an “equals” comparison, our visibility operation could exploit this dedicated hardware. Currently, this computation is done in a fragment program.

The radiosity computation uses about twice the amount of memory to compute a solution as to display. The fixed amount of memory on most graphics cards is a problem for large scenes. However, 3DLabs already has a card that supports virtual memory[cite], and we expect more manufacturers will follow suit.

7 Acknowledgements

The authors would like to thank Jeff Juliano, Stephen Ehmann, Paul Keller and David Kirk of NVIDIA for providing engineering boards and technical help. We would also like to thank the UNC Global Illumination Group for numerous detailed discussions. This work was supported in part by NVIDIA Corporation, US NIH National Center for Research Resources Grant Number P41 RR 02170, US Office of Naval Research N00014-01-1-0061, US Department of Energy ASCI program, and National Science Foundation grants ACR-9876914 and IIS-0121293.

Bibliography

BASTOS, R., GOSLIN, M. AND ZHANG, H. 1997. Efficient Radiosity Rendering using Textures and Bicubic Reconstruction. *ACM-SIGGRAPH Symposium on Interactive 3D Graphics*.

BAUM, D. R. AND WINGET, J. M. 1990. Real Time Radiosity Through Parallel Processing and Hardware Acceleration. *Proceedings of 1990 Symposium on Interactive 3D Graphics*.

CARR, N. A., HALL, J. D. AND HART, J. C. 2002. The Ray Engine. *Graphics Hardware 2002*, Saarbrucken, Germany.

COHEN, M. F., CHEN, S. E., WALLACE, J. R. AND GREENBERG, D. P. 1988. A Progressive Refinement

Approach to Fast Radiosity Image Generation. *Proceedings of SIGGRAPH 88*.

COHEN, M. F. AND GREENBERG, D. P. 1985. The Hemi-Cube: A Radiosity Solution for Complex Environments. *Proceedings of SIGGRAPH 85*.

COHEN, M. F. AND WALLACE, J. R. (1993). Radiosity and Realistic Image Synthesis. Cambridge, MA, Academic Press.

FUNKHOUSER, T. A. 1996. Coarse-Grained Parallelism for Hierarchical Radiosity Using Group Iterative Methods. *Proceedings of SIGGRAPH 96*.

HANRAHAN, P., SALZMAN, D. AND AUPPERLE, L. 1991. A Rapid Hierarchical Radiosity Algorithm. *Proceedings of SIGGRAPH 91*.

HECKBERT, P. S. 1990. Adaptive Radiosity Textures for Bidirectional Ray Tracing. *Proceedings of SIGGRAPH 90*.

IMMEL, D., COHEN, M. AND GREENBERG, D. 1986. A Radiosity Method for Non-Diffuse Environments. *Proceedings of SIGGRAPH 86*.

KELLER, A. 1997. Instant Radiosity. *Proceedings of SIGGRAPH 97*.

NIELSEN, K. H. AND CHRISTENSEN, N. J. 2002. Fast Texture Based Form Factor Calculations for Radiosity using Graphics Hardware. *Journal of Graphics Tools* **6**(4).

PURCELL, T. J., BUCK, I., MARK, W. R. AND HANRAHAN, P. 2002. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002)* **21**(3): 703-712.

SILLION, F. X., ARVO, J. R., WESTIN, S. H. AND GREENBERG, D. P. 1991. A Global Illumination Solution for General Reflectance Distributions. *Proceedings of SIGGRAPH 91*.

SMITS, B. E., ARVO, J. R. AND SALESIN, D. H. 1992. An Importance-Driven Radiosity Algorithm. *Proceedings of SIGGRAPH 92*.

VARSHNEY, A. AND PRINS, J. F. 1992. An Environment-Projection Approach to Radiosity for Mesh-Connected Computers. *Proceedings of the Third Eurographics Workshop on Rendering*.

VEDEL, C. AND PUECH, C. 1992. A Testbed for Adaptive Subvidision in Progressive Radiosity. *Second Eurographics Workshop on Rendering*.

VLACHOS, A., PETERS, J., BOYD, C. AND MITCHELL, J. L. 2001. Curved PN Triangles. *2001 ACM Symposium on Interactive 3D Graphics*.

WALLACE, J. R., ELMQUIST, K. A. AND HAINES, E. A. 1989. A Ray Tracing Algorithm for Progressive Radiosity. *Proceedings of SIGGRAPH 89*.

Appendix : Implementation Details

The majority of the computation described in this paper is implemented in fragment programs, which run on the powerful fragment processor. In addition, we can exploit several specialized capabilities of graphics hardware that make it more useful than simply another CPU.

In order to determine the projection of each texel onto each of the five hemicube faces, we need to bind all five hemicube textures and all five projection matrices at once. Instead of computing each of these projections per fragment, we can compute them per vertex and use homogeneous coordinates and hardware interpolation to get the value at each texel. The fragment program only has to compute the perspective division at each texel, which uses the projective texture functionality.

Texture Packing

While the high dynamic range of floating-point textures is necessary for radiosity computation, the high precision is not as necessary. The Cg programming language allows two half-precision (16-bit) textures to be packed into one full-precision (32-bit) floating-point texture. This feature is particularly useful for radiosity, since it allows both the residual and the radiosity textures to be stored (and operated on) together. Our previous implementation created a "delta" texture, which then had to be added to the residual and radiosity textures. This addition required 2 more passes. The trade-off with texture packing is that every step that needs the radiosity or residual values (such as the sort or the mipmap steps) must unpack the values before computation. In architectures that support multiple render targets, this texture packing is not necessary.

Occlusion Queries

One of the ways that we can get better hardware efficiency is by aggregating occlusion queries. This is because the occlusion query must stall the pipeline until the results are ready, so starting a query and then waiting for the result is inefficient. We try to start as many queries as possible, then wait until they are all finished until querying the result of the first one. This complicates the quadtree traversal: The first pass down the tree calculates the new radiosity values, and adds this node to a queue. After we pass over the entire quadtree, we test every node in the queue and determine whether it needs to be subdivided. If a node needs to be subdivided, then we traverse down the tree and subdivide the node. While this sounds inefficient, it is about 20% faster than querying the nodes individually.