Eurographics Symposium on Rendering (2004)
H. W. Jensen, A. Keller (Editors)

An Irradiance Atlas for Global Illumination in Complex Production Scenes

Per H. Christensen and Dana Batali

Pixar Animation Studios

Abstract

We introduce a tiled 3D MIP map representation of global illumination data. The representation is an adaptive, sparse octree with a "brick" at each octree node; each brick consists of 8^3 voxels with sparse irradiance values. The representation is designed to enable efficient caching. Combined with photon tracing and recent advances in distribution ray tracing of very complex scenes, the result is a method for efficient and flexible computation of global illumination in very complex scenes. The method can handle scenes with many more textures, geometry, and photons than could fit in memory. We show an example of a CG movie scene that has been retrofitted with global illumination shading using our method.

1. Introduction

Our goal is to make global illumination a practical, efficient, and flexible tool for CG movie production. This requires a rendering method that can handle scenes with hundreds of light sources, thousands of texture and displacement maps, and geometry consisting of hundreds of millions of polygons when the objects are fully tessellated. We want the benefits of full global illumination without limiting scene complexity or shader generality.

We introduce a tiled 3D MIP map representation for volume and surface data. The representation is an adaptive, sparse octree with a brick at each octree node. A brick is a 3D generalization of a tile; our bricks contain 8^3 voxels each. We call our tiled 3D MIP map representation a *brick map*. The brick map is designed to enable efficient caching.

We demonstrate the utility of the brick map representation by using it to store irradiance data for an efficient and flexible global illumination method able to deal with very complex scenes. Our method is an extension of the photon map method and consists of three steps. The first step is photon tracing. The photons are stored in a collection of photon maps that together cover the entire scene. We call this collection of photon maps a *photon atlas*. In the second step, the irradiance is estimated at each photon position, and for each photon map a brick map representation of the irradiance is constructed. We call this collection of irradiance brick maps an *irradiance atlas*. The last step is rendering using final gathering and irradiance interpolation, with the irradiance atlas providing a rough estimate of the global illumination.

In this paper we focus on multi-bounce soft global illumination on surfaces, but the same tiled 3D MIP map representation can be used for single-bounce global illumination, caustics, participating media, etc.

2. Related Work

Our proposed global illumination method extends the photon map method to handle more complex illumination and geometry. Our method is inspired by 3D MIP maps used for volume rendering and interactive paint programs, and by recent advances in distribution ray tracing of complex scenes.

2.1. Texture MIP Maps

Peachey [Pea90] introduced a multiresolution texture caching scheme that caches texture tiles from 2D MIP maps [Wil83]. Each texture is represented at multiple resolutions, and the texture at each resolution is tiled into 32^2 pixel tiles. Peachey's texture tile cache is highly efficient for scanline rendering, ray tracing, and distribution ray tracing: a cache size of 1% of the total texture size is usually sufficient [CLF*03]. Our 3D tiling and caching approach, introduced in the following, is a natural generalization of Peachey's.

© The Eurographics Association 2004.

2D MIP maps are easily generalized to 3D. One difference is that the 3D data are often sparse, so the data are stored in a sparse octree. Levoy and Whitaker [LW90] and Laur and Hanrahan [LH91] used 3D MIP maps for volume rendering of data sets such as magnetic resonance scans. Neyret [Ney98] used 3D MIP maps to model and render foliage, hair, and fur.

Benson and Davis [BD02] and DeBry et al. [DGPR02] used 3D MIP maps to represent textures on surfaces. This avoids forcing a 2D parameterization on surfaces with no natural parameterization — such as implicit surfaces, subdivision surfaces, and dense polygon meshes. They generated the 3D MIP maps in interactive 3D paint programs. The sparse representation easily adapts to changes in detail such as the intricate texture details at a decal. For texture lookups, the texture filter size determines which MIP map level is used. Benson and Davis only briefly mention that they tile the octree, and they do not describe caching at all. In our application, tiling and caching are absolutely essential, so we will describe this aspect in detail.

Jensen and Buhler [JB02] also used an octree to represent irradiance on surfaces (for computation of subsurface scattering), but they did not tile or cache their irradiance data.

2.2. Photon Maps

The photon map method for computation of global illumination was introduced by Jensen [Jen96, Jen01]. It is a three-pass method. First photons are emitted from the light sources, traced through the scene, and stored in a photon map at every diffuse surface they hit; then the unorganized collection of stored photons is sorted into a kd-tree; and finally the scene is rendered using final gathering (a single level of distribution ray tracing). The irradiance at final gather ray hit points is estimated from the density and power of the nearest photons. Irradiance interpolation [WH92] is used to reduce the number of final gathers.

The final gathering can be sped up by a factor of 5 to 7 by augmenting the second pass to precompute the irradiance at the photon positions [Chr99]. During rendering, the precomputed irradiance of the nearest photon is looked up at final gather ray hit points, so no density estimates are necessary during rendering. In this paper, we propose a further enhancement of the second pass to be able to handle huge numbers of photons (and also improve the filtering of the irradiance estimates).

In order to compute sufficiently accurate indirect illumination in large scenes with intricate geometric detail, a lot of photons are necessary. Photons can be traced in very complex scenes, and if the photons are streamed to disk the photon map size can exceed the memory size of the computer. However, if the photon map is too large, it cannot be read back in! For example, on a computer with 1 GB memory, we typically cannot expect to have more than 300 MB available for photons during rendering (since we also need to store the scene description, etc.); this corresponds to only 10 million photons.

Final gather rays hit the scene in very incoherent order. Hence the memory containing a standard photon map is accessed very incoherently during final gathering. If the photon map information is not carefully organized, a limited-size cache is of no help. What we need is a tiled, hierarchical representation of the photon map information.

Some variations of the photon map method use importance to reduce the number of photons stored in a scene for a view-dependent global illumination solution [PP98, SW00, KW00]. But in our applications, we prefer a view-independent collection of photon maps that can be used for fly-through-like animations, so we cannot reduce the number of photons this way.

Other methods store the photons on surfaces instead of in a kd-tree [Mys97, TWFP97, WHSG97]. However, those approaches cannot handle very complex scenes since each (potentially tiny) surface has a separate irradiance representation. In contrast, a single brick can cover many small surfaces if the illumination is smooth.

2.3. Rendering Complex Scenes

Pharr et al. [PKGH97] used path tracing to compute global illumination in complex scenes. They reordered the ray intersection tests to increase the coherency of geometry accesses. This reordering made it possible to render scenes that were 10 times larger than the geometry cache size, but unfortunately introduced shader limitations.

In Christensen et al. [CLF*03], we presented a method to perform distribution ray tracing in very complex scenes. We used a MIP map representation of tessellated geometry and a multiresolution geometry cache that is very similar to Peacheys 2D texture tile cache. Ray differentials [Ige99, SW01] were used to determine the required tessellation accuracy. An insight about cache coherency properties enabled efficient scanline rendering, ray tracing, and distribution ray tracing of very complex geometry. We demonstrated distribution ray tracing in scenes with full tessellations over 100 times larger than the geometry cache size.

Our distribution ray tracing method could be extended to handle full multi-bounce global illumination by using an irradiance cache as in the Radiance system [WRC88, WS98]. However, here we limit the distribution ray tracing to a single level, and instead use photon tracing and an irradiance atlas to more efficiently capture the effect of multiple bounces.

3. The Brick Map

In this section, we introduce the brick map, a general, tiled 3D MIP map representation for volume and surface data.

© The Eurographics Association 2004.

The brick map is a 3D generalization of Peachey's tiled 2D texture MIP maps — very similar in spirit to the sparse, adaptive octrees used by Benson and Davis and DeBry et al., but designed with more emphasis on tiling and cacheability.

3.1. Brick Map Data Structures

The data are organized in a sparse, adaptive octree. Each node in the octree has eight pointers to its children and a pointer to a brick. A brick has N^3 voxels (we have chosen 8^3 in our implementation). Each voxel can be empty or contain texture data such as diffuse and specular color, specularity, irradiance, etc. Each voxel also contains a weight ("coverage" or "alpha") indicating how much data has been inserted in that voxel. For surface data, we also store the average normal for each voxel, and a flag indicating whether the normals are incoherent. For the finest data (at octree leaves), we store multiple voxels in a linked list at the same voxel position if the data in that position have incoherent normals. The voxel, brick, and octree data structures are:

```
#define N 8

struct Voxel {
  float *data;
  float weight;
  struct Vector averagenormal;
  bool incoherentnormals;
  struct Voxel *next;
};

struct Brick {
  struct Voxel voxel[N*N*N];
};

struct OctreeNode {
  struct OctreeNode *child[8];
  struct Brick *brick;
};
```

Figure 1 shows the top three levels of a brick map for surface data. The brick at the root of the octree contains a very coarse approximation of the 3D texture, while bricks at leaf nodes contain the most accurate representation. The brick map representation automatically adapts to data density: the octree is only deep in regions with many data points.

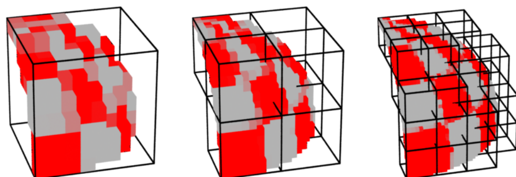

Figure 1: *Sparse brick map for surface data.*

For the global illumination application we are interested in here, we are only concerned with data that come from points on surfaces. Hence many voxels will be empty. In other applications (for example solid textures [Pea85, Per85] and volume photon maps [JC98]), data are present in the entire volume. This results in a full octree where all voxels contain data. Figure 2 shows the top three levels of a brick map for volume data.

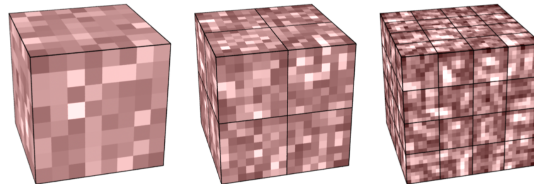

Figure 2: *Dense brick map for volume data.*

3.2. Creating a Brick Map

Given a set of points (a "point cloud") with associated normals, radii, and data, we want to create a brick map representation of the data. We do this in three steps.

The first step is to create a sparse octree structure based on point density and radii. (The radius associated with each point can for example be determined from the local point density.) First the bounding volume of the data set is found. Then the volume is divided evenly into eight subvolumes, and the data points are divided according to which subvolume they are contained in. Then each subvolume is recursively divided again, etc. This recursive division stops when the subvolume contains no points with smaller radius than half a voxel diagonal (or no points at all).

The second step is to create a linked list of the points that overlap each leaf node. For each data point p, we first compute a small volume V_p based on its position and radius. Then, starting at the root node, we determine which child nodes overlap V_p, and recursively go to those nodes. If the node is a leaf, we insert point p in a linked list at that node. Note that each point can be inserted in more than one linked list. The advantage of this strategy is that the information needed to create each brick is now completely local.

In the third step, we insert the point data. For each octree node, starting at the leaves, the data that overlap the node are inserted in its brick and the bricks of its parent nodes. When a point's data are inserted into a brick, we determine which voxel(s) the point volume overlaps, and add the data values $\bar{d}_p$ to the data $\bar{d}_v$ in those voxels. When added, the data are multiplied by a weight that is determined by how large a fraction of the voxel volume V_v is overlapping the point volume V_p. We also increase the voxel weight, and update the average normal for each covered voxel:

$$\begin{aligned} \bar{d}_v &\mathrel{+}= w_{vp}\,\bar{d}_p \\ w_v &\mathrel{+}= w_{vp} \\ \vec{n}_v &\mathrel{+}= w_{vp}\,\vec{n}_p\,, \end{aligned}$$

© The Eurographics Association 2004.

with

$$w_{vp} = \frac{V_p \cap V_v}{V_v}.$$

If a point is inserted into a voxel of an octree leaf node, and the point normal is very different (for example more than 45 degrees) from the average normal of the voxel, we allocate a new voxel, insert the point data in the new voxel, and point the previous voxel's next pointer to the new voxel. At internal nodes (non-leaves), we do not allocate new voxels, but add the data, weight, and normal as described above and set the voxel's "incoherentnormals" flag. There are relatively few leaf node voxels with incoherent normals, so this strategy does not increase the memory use significantly.

When all the node's data have been inserted, we divide each data value in a voxel by the weight of the voxel. We then determine the maximum data (and normal) variation of all $2 \times 2 \times 2$ voxel groups of the brick. If the variation is smaller than a user-specified maximum error, we eliminate the brick. If the variation is larger, we write the brick to disk. Empty voxels are not written; this saves a lot of disk space for sparse brick maps. We don't write the weights either, since all data have already been divided by their weights.

3.3. Looking Up in a Brick Map

Given a position, normal, and filter size (radius), we want to find the interpolated value of the data at that point — smoothed as appropriate for the given filter size.

We first construct a lookup volume V_ℓ from the lookup point position and filter size. Then we recursively traverse the octree, starting at the root and visiting all children that the lookup volume overlaps (usually just one child, but can be up to eight children). This recursive traversal continues until the node contains voxels of approximately the same size as the lookup volume or a leaf node has been reached. If the voxels that overlap V_ℓ have the "incoherentnormals" flag set, we reduce the size of the lookup volume and look deeper in the octree to resolve the normals. This resolves potential color leaking problems along edges and through thin objects [DGPR02].

When we have reached the appropriate level in the octree, we determine which voxels overlap the lookup volume and have normals similar to the lookup normal (for example within 45 degrees). If we are at a leaf node, we may have to follow the linked list of voxels sharing the same voxel position in the brick. We increment the lookup result $\bar{d}_\ell$ by the voxel data multiplied by the fractional overlap of V_v and V_ℓ:

$$\bar{d}_\ell \mathrel{+}= w_{v\ell}\, \bar{d}_v$$

with

$$w_{v\ell} = \frac{V_\ell \cap V_v}{V_v}.$$

(Empty voxels and voxels with inappropriate average normals do not contribute to the lookup results.)

If the lookup volume overlaps a neighbor octree branch that does not have as much detail as the branch that contains point p itself, we use the data at the available resolution. The weights of the data are still determined by the ratio of the overlap volume and the (fine or coarse) voxel volume. More information about lookups that overlap different levels of detail can be found in Benson and Davis [BD02].

Generally the lookup volume size will fall between two levels in the octree. In this case, we can choose to look up in each of the two levels and do a linear interpolation of the resulting values; this ensures smooth transitions between different resolutions.

3.4. Brick Map Caching

The entire octree of a brick map is read from disk the first time the brick map is accessed, and then kept in memory. This is acceptable because each octree node consists of only nine pointers (eight pointers to child nodes and one pointer to a brick), and typically the octree nodes make up less than half a percent of the total size of a brick map. Even for a collection of brick maps with 1 million bricks, the octrees only use 1 million * 9 pointers * 4 bytes = 36 MB.

Bricks are read from disk on demand and cached in memory. If the fine lookups are coherent (as they are in our global illumination method), the cache has a high hit rate and caching is very efficient. Note that even though the individual bricks can be sparse, the cache slots need to have space for all 8^3 potential voxels in a brick since the same cache slot may be filled later with a dense brick. (We dynamically allocate and free the "extra" voxels at leaf nodes with inconsistent normals. To avoid fragmentation, we use a pool of pre-allocated voxels for this.) Our cache uses a least-recently-used (LRU) replacement strategy. In our implementation, the brick map cache size can be selected by the user. The default size is 10 MB, corresponding to a capacity of 1280 bricks.

We have found that in typical applications, less than 50% of the voxels in the brick cache have data in them. In other words, more than half of the voxels are empty. This means that it would be possible to compress the 10MB brick map cache to around 5MB. However, the cost would be many free's and malloc's of voxels every time a brick is read into the cache.

4. Global Illumination using Brick Maps

Before the global illumination computation begins, the user must select groups of objects that should share a photon map file. (Alternatively, photon maps could be automatically assigned based on the object hierarchy and sizes, or divided along creases between major objects [LC03].) The groups should be chosen such that no single photon map ends up with more photons than can fit in memory at one time. We have found this grouping easy and intuitive in practice.

© The Eurographics Association 2004.

In the first global illumination step, photons are traced as in the standard photon map method. When a photon hits a diffuse object, it is written to that object's photon map file. Then the photons in each photon map file are sorted into a kd-tree and the irradiance and local area is estimated at each photon position. (The radius associated with each photon follows directly from its area estimate.) The result is a collection of oriented, colored disks reminiscent of the surfels used for surface representation [PZvBG00]. Then a brick map is constructed from the irradiance disks for each photon map file.

During rendering, we need to determine the radiance at final gather ray hit points. The irradiance at the point is looked up in the irradiance map of the hit object, and multiplied by the local diffuse color at the hit point. The ray differential is used to determine the filter size both for the irradiance map lookup and the diffuse texture lookup.

Due to the coherency properties of scanline rendering, ray tracing, and distribution ray tracing [CLF*03], the fine brick accesses are coherent. This means that it is sufficient to have a brick cache with a capacity much smaller than the total number of bricks in the irradiance atlas.

It is interesting to note that an irradiance map lookup is usually faster than computing the illumination (even just the direct illumination) if there are several light sources each requiring shadow ray tracing or a shadow map lookup.

5. Results

We implemented this global illumination method in Pixar's RenderMan, a widely used commercial renderer [AG00]. All our tests were done on a Linux PC with a 3.4 GHz Pentium 4 processor. The tests used less than 600 MB of memory. The images were rendered at resolution 1024×553.

5.1. Test Scene

We tested the method on a scene from the CG movie "Monsters, Inc." [DP01]. The scene was not modeled with global illumination in mind, but we retrofitted it with global illumination shaders for these tests. Figure 3 shows the scene: a city block with many individually modeled buildings, trees, cars, etc. The scene consists of 36,000 high-level primitives, mostly NURBS patches and subdivision surfaces. In this image, the scene is only illuminated with direct light (from a directional light source similar to the sun) with ray traced shadows. Large parts of the scene are completely black since no direct light reaches them. The render time was 6 minutes.

5.2. Photon Tracing

The objects were manually grouped into 41 groups. Emitting 300 million photons took 29 minutes and resulted in 52 million photons being stored. (Due to a rather loose scene bounding box, the majority of the emitted photons did not hit any objects.) The maximum photon reflection depth was set to 10 to ensure that the photon maps capture enough bounces of indirect illumination. The photons were stored in 41 photon map files with a total size of 2.2 GB. Each street, building, and car has a separate photon map file. Figure 4 shows two of the photon maps. The photon map for the car contains 76,000 photons (file size 3.2 MB) while the photon map for the building contains 3.4 million photons (144 MB).

Figure 3: *Monstropolis city block with only direct illumination. (© Disney/Pixar)*

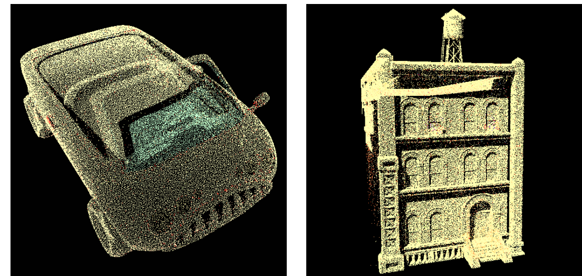

Figure 4: *Photon map for car and building.*

For an overview of the illumination in the scene, figure 5 shows a coarse photon atlas of the entire scene. This figure only shows 0.1% of the photons in the full photon atlas.

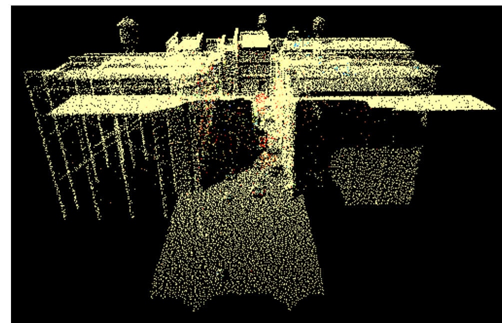

Figure 5: *Coarse photon atlas for entire scene.*

© The Eurographics Association 2004.

5.3. Generating the Irradiance Atlas

The irradiance and area (radius) was estimated at each photon position; this took 18 minutes for all photon maps. Each irradiance value and area was estimated using the nearest 50 photons.

Next, irradiance brick maps were computed from the irradiance point clouds; this computation took 25 minutes. For brick elimination, we set the maximum error to 0.03 and the normal deviation threshold to 45 degrees.

Figure 6 shows the top four levels of the irradiance brick maps for the car and building. The car's irradiance map has 959 bricks (file size 69 MB); the irradiance map of the building has 31,700 bricks (190 MB). The total size of all the irradiance brick map files is 2.4 GB. The irradiance atlas has 253,000 bricks in total — nearly 200 times the default brick cache capacity.

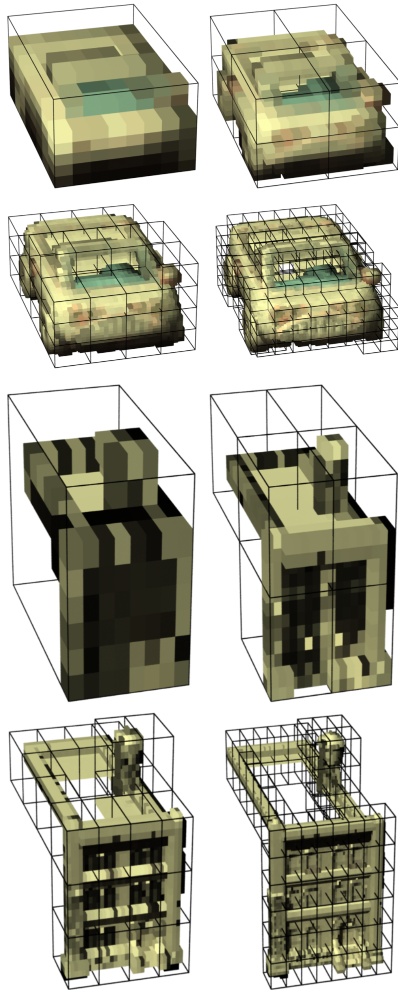

Figure 6: *Irradiance brick maps for the car and building.*

Figure 7 shows the car and building shaded with irradiance from their respective irradiance maps.

Figure 7: *Car and building shaded with irradiance from their irradiance brick maps.*

Figure 8 shows the entire scene rendered with irradiance from the irradiance atlas. Note the color bleeding from the red car onto the street and between the leaves on the trees.

Figure 8: *Entire scene shaded with irradiance from the irradiance atlas. (© Disney/Pixar)*

Figure 9 shows the scene with the objects shaded by the irradiance times the local diffuse color. This image clearly illustrates why the irradiance maps should not be rendered directly — they are simply too noisy and blurry.

Figure 9: *Irradiance times local diffuse color. (© Disney/Pixar)*

© The Eurographics Association 2004.

Figure 10: *Global illumination in a production scene with 237 million (unique, non-instanced) triangles and 52 million photons. Photon tracing took 29 minutes, computing an irradiance atlas took 43 minutes, and rendering took 3.8 hours on a 3.4 GHz CPU. Without the irradiance atlas representation, the 52 million photons would not fit in memory and it would be impossible to render the global illumination efficiently. (© Disney/Pixar)*

5.4. Rendering

Figure 10 shows a final gather rendering of the entire scene. Note the high quality of the indirect illumination in the shaded areas, for example the houses on the left side of the street. There is also a very subtle color bleeding from the red car onto the street. Rendering this image took 3.8 hours and required 73 million final gather rays and 4 million shadow rays. During rendering, the scene was divided into 463,000 surface patches, which corresponds to 237 million triangles at maximum tessellation rate. The 2D texture and tessellation cache sizes were set to 10 MB and 30 MB, respectively. There were 215 million brick cache lookups. Table 1 shows cache statistics for different brick cache sizes. As it can be seen, our default cache size of 10 MB is a reasonable compromise between size and speed.

cache size	misses	hit rate	render time
0	215M	0%	6.7h
1MB	21M	90.3%	4.1h
10MB	2.3M	99.0%	3.8h
100MB	0.9M	99.6%	3.7h

Table 1: *Brick cache statistics.*

6. Discussion and Future Work

6.1. Brick Map Creation

Our algorithm for generating a brick map minimizes the number of bricks that are kept in memory at one time — only $O(\log n)$ bricks are needed, where n is the number of octree nodes. This is usually only 10–20 bricks. In our first implementation, we used a simpler algorithm that stored all bricks in memory during brick map generation, and only eliminated and wrote bricks in the end. However, storing all the bricks in memory turned out to be a significant memory bottleneck.

With our current workflow, each photon map is sorted into a kd-tree to estimate the irradiances, and the irradiances are then converted to a brick map. It may be possible (and more efficient) to combine these two steps, i.e. computing the irradiances directly in the brick map once the brick map structure has been determined from the photon positions.

Another possible optimization is to compute fewer photon irradiance estimates in regions with fairly uniform photon density. Currently such uniform irradiance estimates are deleted in step 3 of the brick map creation, but it would be more efficient to not compute them at all.

© The Eurographics Association 2004.

6.2. Brick Map Lookups

Compared to the original photon map method, we often get the benefit of filtered (less noisy) irradiance estimates. While the original photon map method always uses a fixed number of photons to estimate the irradiance at final gather ray hit points, we use the ray differential to determine the lookup level in the brick map. This means that we get the average of several irradiance estimates when appropriate.

Our brick map lookups currently use quadrilinear interpolation (trilinear interpolation between neighbor voxels; linear interpolation between two levels) and a single filter size (radius) to determine the MIP map level. Interesting future improvements would be higher-order interpolation and anisotropic filtering.

6.3. Variations and Extensions

In this paper, we have focused on the application of brick maps in a multi-bounce global illumination solution. For single-bounce global illumination, brick maps can be used as follows: skip the photon tracing pass but render the scene with direct illumination and store ("bake") the resulting radiance data as 3D point clouds. Then convert the point clouds to brick maps, and do a final gather rendering. The multiresolution caching properties of brick maps are equally useful in this application. More details can be found in [Pix04]. The brick map representation can also be used for caustics and for participating media.

Our approach is very flexible in that the irradiance maps can be manipulated independently after they are generated. If, for example, we want more color bleeding from an object, we could load the brick map into an interactive 3D paint package and increase some or all of the irradiance values.

During final gather rendering, the high-quality 3D global illumination results can be stored as another 3D texture and reused in subsequent renderings of the same scene. This can amortize the computation cost over many images and also gives a lot of flexibility to mix and match global illumination solutions. If accurate global illumination has been computed for the entire scene, rendering the same scene from a different camera angle takes only a few minutes (approximately the same time as to render direct illumination alone). A simplified version could even be rendered at interactive speed.

We believe that our method is particularly well suited for parallel execution. For very complex scenes, the bottleneck for parallel ray tracing and global illumination is usually the geometry and texture accesses, and with our multiresolution caches this bottleneck is eliminated.

7. Conclusion

We have introduced the brick map, a tiled 3D MIP map format for efficient representation and caching of general surface and volume data. We use the brick map format to improve the photon map method with efficient caching of global illumination irradiance data. The resulting method enables efficient and flexible computation of multi-bounce global illumination in very complex scenes. It is our hope that this will lead to more widespread use of global illumination in CG movie production.

Acknowledgments

We are very grateful for the substantial contributions from David Laur, Julian Fong, and other members of the RenderMan Products team. Thanks to Wayne Wooten for digging out the Monstropolis scene data. Thanks to John Anderson, Guido Quaroni, Fabio Pellacini, Eliot Smyrl, Justin Ritter, and others for discussions of 3D data, 3D textures, and the brick map format. The Monstropolis scene is copyright ©Disney Enterprises, Inc. / Pixar Animation Studios.

References

[AG00] APODACA A., GRITZ L.: *Advanced RenderMan — Creating CGI for Motion Pictures*. Morgan Kaufmann, 2000.

[BD02] BENSON D., DAVIS J.: Octree textures. In *ACM Transactions on Graphics, Proc. SIGGRAPH 02* (2002), pp. 785–790.

[Chr99] CHRISTENSEN P.: Faster photon map global illumination. *Journal of Graphics Tools 4*, 3 (1999), 1–10.

[CLF*03] CHRISTENSEN P., LAUR D., FONG J., WOOTEN W., BATALI D.: Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. In *Computer Graphics Forum, Proc. Eurographics 03* (2003), pp. 543–552.

[DGPR02] DEBRY D., GIBBS J., PETTY D., ROBINS N.: Painting and rendering textures on unparameterized models. In *ACM Transactions on Graphics, Proc. SIGGRAPH 02* (2002), pp. 763–768.

[DP01] DISNEY ENTERPRISES, INC., PIXAR ANIMATION STUDIOS: Monsters, Inc., 2001.

[Ige99] IGEHY H.: Tracing ray differentials. In *Computer Graphics, Proc. SIGGRAPH 99* (1999), pp. 179–186.

[JB02] JENSEN H., BUHLER J.: A rapid hierarchical rendering technique for translucent materials. In *ACM Transactions on Graphics, Proc. SIGGRAPH 02* (2002), pp. 576–581.

[JC98] JENSEN H., CHRISTENSEN P.: Efficient simulation of light transport in scenes with participating media using photon maps. In *Computer Graphics, Proc. SIGGRAPH 98* (1998), pp. 311–320.

[Jen96] JENSEN H.: Global illumination using photon maps. In *Rendering Techniques '96, Proc. 7th Eurographics Workshop on Rendering* (1996), Springer-Verlag, pp. 21–30.

© The Eurographics Association 2004.

[Jen01] JENSEN H.: *Realistic Image Synthesis using Photon Mapping*. A. K. Peters, 2001.

[KW00] KELLER A., WALD I.: Efficient importance sampling techniques for the photon map. In *Proc. 5th Fall Workshop on Vision, Modeling, and Visualization* (2000), IEEE, pp. 271–279.

[LC03] LARSEN B., CHRISTENSEN N.: Optimizing photon mapping using multiple photon maps for irradiance estimates. In *Proc. 11th Int. Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)* (2003), University of West Bohemia.

[LH91] LAUR D., HANRAHAN P.: Hierarchical splatting: a progressive refinement algorithm for volume rendering. In *Computer Graphics, Proc. SIGGRAPH 91* (1991), pp. 285–288.

[LW90] LEVOY M., WHITAKER R.: Gaze-directed volume rendering. In *Computer Graphics, Proc. Symposium on Interactive 3D Graphics* (1990), pp. 217–223.

[Mys97] MYSZKOWSKI K.: Lighting reconstruction using fast and adaptive density estimation techniques. In *Rendering Techniques '97, Proc. 8th Eurographics Workshop on Rendering* (1997), Springer-Verlag, pp. 251–262.

[Ney98] NEYRET F.: Modeling, animating, and rendering complex scenes using volumetric textures. *IEEE Transactions on Visualization and Computer Graphics 4*, 1 (1998), 55–70.

[Pea85] PEACHEY D.: Solid texturing of complex surfaces. In *Computer Graphics, Proc. SIGGRAPH 85* (1985), pp. 279–286.

[Pea90] PEACHEY D.: Texture on demand. Pixar technical memo 217 (unpublished manuscript), 1990.

[Per85] PERLIN K.: An image synthesizer. In *Computer Graphics, Proc. SIGGRAPH 85* (1985), pp. 287–296.

[Pix04] PIXAR ANIMATION STUDIOS: Ambient occlusion, image-based illumination, and global illumination. PhotoRealistic RenderMan Application Note #35, 2004.

[PKGH97] PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P.: Rendering complex scenes with memory-coherent ray tracing. In *Computer Graphics, Proc. SIGGRAPH 97* (1997), pp. 101–108.

[PP98] PETER I., PIETREK G.: Importance driven construction of photon maps. In *Rendering Techniques '98, Proc. 9th Eurographics Workshop on Rendering* (1998), Springer-Verlag, pp. 269–280.

[PZvBG00] PFISTER H., ZWICKER M., VAN BAAR J., GROSS M.: Surfels: surface elements as rendering primitives. In *Computer Graphics, Proc. SIGGRAPH 00* (2000), pp. 335–342.

[SW00] SUYKENS F., WILLEMS Y.: Density control for photon maps. In *Rendering Techniques '00, Proc. 11th Eurographics Workshop on Rendering* (2000), Springer-Verlag, pp. 11–22.

[SW01] SUYKENS F., WILLEMS Y.: Path differentials and applications. In *Rendering Techniques '01, Proc. 12th Eurographics Workshop on Rendering* (2001), Springer-Verlag, pp. 257–268.

[TWFP97] TOBLER R., WILKIE A., FEDA M., PURGATHOFER W.: A hierarchical subdivision algorithm for stochastic radiosity methods. In *Rendering Techniques '97, Proc. 8th Eurographics Workshop on Rendering* (1997), Springer-Verlag, pp. 193–204.

[WH92] WARD G., HECKBERT P.: Irradiance gradients. In *Proc. 3rd Eurographics Workshop on Rendering* (1992), pp. 85–98.

[WHSG97] WALTER B., HUBBARD P., SHIRLEY P., GREENBERG D.: Global illumination using local linear density estimation. *ACM Transactions on Graphics 16*, 3 (1997), 217–259.

[Wil83] WILLIAMS L.: Pyramidal parametrics. In *Computer Graphics, Proc. SIGGRAPH 83* (1983), pp. 1–11.

[WRC88] WARD G., RUBINSTEIN F., CLEAR R.: A ray tracing solution for diffuse interreflection. In *Computer Graphics, Proc. SIGGRAPH 88* (1988), pp. 85–92.

[WS98] WARD LARSON G., SHAKESPEARE R.: *Rendering with Radiance*. Morgan Kaufmann, 1998.

© The Eurographics Association 2004.

Figure 3: *Monstropolis city block with only direct illumination. (© Disney/Pixar)*

Figure 8: *Entire scene shaded with irradiance from the irradiance atlas. (© Disney/Pixar)*

Figure 9: *Irradiance times local diffuse color. (© Disney/Pixar)*

Figure 10: *Global illumination in a production scene with 237 million triangles, 52 million photons. (© Disney/Pixar)*

© The Eurographics Association 2004.