Parallel Baking of Colors in Semi-Reflective Scenes

Precomputing a Ray-Trace Map to Permit Real-Time Visualization

Ally Robinson
Computer Science
Rensselaer Polytechnic Institute
Troy, NY United States
ally@purplish.blue

Gabriel Jacoby-Cooper
Computer Science
Rensselaer Polytechnic Institute
Troy, NY United States
rensselaer@gabrieljc.me

ABSTRACT

In this paper we show a method of baking colors onto a mesh for later real time rendering. The baking process can be run in parallel in order to reduce the time spent rendering, which can be rather substantial. The system allows both variable baked color resolution, and variable refinement depth on the baking process. Choosing values for these parameters and then running the baking process then generates a mesh that has per-vertex color information from evenly sampled angles around the vertex. This allows the real time renderer to interpolate between these different viewpoints angles to get a semi-accurate color from any perspective.

While our current implementation has some limitations, it does provide some unique benefits such as linear runtime with respect to bounce depth, high level of parallelizability, real time viewing of baked results, and the potential to include many other ray-tracing techniques.

KEYWORDS

Parallel, Ray Tracing, Mesh Baking, CUDA, MPI, Precomputation

1 Introduction

Modern graphics hardware continues to improve, but high quality real time ray tracing still remains far out of reach. For many aspects of rendering, this can be worked around or is fine to sacrifice for performance.

One such aspect is reflections. While reflections can be cheated by using environment maps or by baking lighting into a mesh such that it appears to have indirect reflected lighting, these approaches still sacrifice quality. Environment maps can not do self reflections without careful tuning, and it isn't practical for some meshes to encode environment maps on all the objects as the memory use would be too significant. Baking lighting onto a mesh also has problems, while very useful and providing high quality results for diffuse scenes, specular reflections are not incorporated and reflected lighting would require a more complicated approach such as photon mapping.

To address these problems, we provide our parallel mesh baking approach. This allows both reflected indirect lighting, self reflections, and while not small, less memory usage than more detailed environment maps. While our approach does not work well where sharp reflections are needed, such as a scene including a mirror, for those that are mostly glossy reflections, it has the potential for very good results.

2 Review of Related Work

Pichler, *et al.* introduce a method for accelerating ray tracing with a precomputation stage that rejects certain ray-triangle intersections**[4]**. Chen, *et al.* interpolate between viewpoints at the pixel level at runtime**[5]**. Purcell, *et al.* parallelize a ray caster on programmable graphics hardware**[6]**, though their technique is fit into the standard rasterization pipeline instead of the general-purpose parallel capabilities that we utilize via CUDA.

3 Implementation

Implementation consists of a few different components. Since the baking of the meshes to put the color information into them and the rendering are removed, and can be done independently, they are written as two separate programs. Additionally, we had to create a new file format to hold the baked data for it to be rendered later by the renderer. All together this gives us three mostly separate components.

3.1 Bake Shop File Format

The Bake Shop file format is the format that we came up with to hold our new baked mesh data. We chose the extension '.bs'. This format follows a very similar format to that of the '.obj' files that are often used in graphics applications. It consists of three sections; vertexes, materials, and then quads. Each section should be separated from the next with a blank newline.

The first section, vertexes, contains the vertices and the color data that is baked into them (assuming it exists). This is encoded by putting the x, y, and z coordinates on a line separated by spaces.

bakeshop_example.bs

```
 1.0  0.0  1.0
-1.0  0.0  1.0
-1.0  0.0 -1.0
 1.0  0.0 -1.0

: 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0
: 1.0 0.0 0.0 0.0 0.0 0.0 0.8 0.0 0.0

0 1 2 3 0
0 3 2 1 1
```

Figure 1: Example Bakeshop File

If we want to give any baked colors for the vertex, we add a viewpoint entry. A viewpoint entry consists of putting a bar character, '|', on the next line followed by the x, y, and z for a possible incoming normal for the vertex and then the r, g, and b component of the color that one should perceive the vertex as being from this angle. As many viewpoint entries as desired can be added to the vertex by repeating the format described with a new angle.

After the vertex section, we get the materials section. This consists of all the materials used in the mesh. Each material should be put on its own line in the following format; a colon followed by the rgb values for the diffuse color, then the rgb values for its reflective color, and finally the rgb components of its emissive strength for each of those components. One can add as many materials as they'd like, including none.

Finally, this leaves the quad section. This section contains all of the quadrilaterals in the mesh (quadrilaterals are the only supported polygons in our current implementation). Each quad is put on its own line and consists of 4 integers indicating the four vertices that make it up followed by one more integer indicating the material. A *n* represents the *(n+1)*th vertex listed in the first section. Similarly for the material number. If one wishes to give a quad no material, they can put the number -1.

At the end of the file, directly after the quad section, put a single bang '!' on a new line.

3.2 Baking Program

The Chef is a parallelized program that takes a "raw" Bake Shop file as input and produces a "baked" Bake Shop file as output. The raw input file consists of vertices, materials, and faces that represent the geometry of a 3D scene. Any viewpoints that are encoded in the input file are ignored because the expectation is that those viewpoints will be generated by the Chef itself. The baked output file also consists of vertices and faces, but it by default lacks material data because the shading is now fully specified by the viewpoint data.

For each vertex in the input file, the Chef generates a user-specified number of viewpoint angles around that vertex on a "virtual sphere". Each viewpoint angle is represented as an inverse normal vector that points "into" the virtual sphere (equivalent to a normalized version of a camera ray). The ray trace begins in the first iteration at each viewpoint for each vertex by reflecting that inverse normal vector off of the mesh at the relevant vertex using a "vertex normal" that's averaged from the adjacent face normals. A color value is generated using material properties that are also averaged from the adjacent faces.

Additionally, factors such as the surface's reflectivity and the view of other emissive faces are taken into account. In all, we are performing an incomplete implementation of Whitted Ray Tracing**[3]**.

Further ray-trace iterations are discussed later in this section, but it's important at this point to explain how the Chef is parallelized. Multiple Chefs can work together in parallel in a Kitchen, which assigns a batch of vertices to each Chef. Using the Message-Passing Interface (MPI), each Chef is assigned its own MPI rank. A Kitchen, in this sense, is just a set of parallel Chefs that can communicate with each other over MPI to bake a single ray-trace map together. Each Chef can be further parallelized by

dispatching sub-batches of its assigned vertex batch to an NVIDIA GPU in parallel via CUDA.

A crucial component of our baking approach involves complex I/O interplay between the different Chefs in a Kitchen. Because the vertices are split across different Chefs, color calculations that one Chef performs naïvely on vertices in the scene that aren't part of its vertex batch using just the original material properties would produce incorrect results. To remedy this, each Chef must be able to incorporate color calculations that other Chefs have previously performed. After the first ray-trace iteration, in which only a single bounce per viewpoint per vertex is performed, the Chefs synchronize with each other using an MPI all-gather operation, which combines a base gather operation with a broadcast operation to collect the computed color data from each Chef and to propagate those data to all of the Chefs in advance of the second ray-trace iteration. Each Chef packs the computed color values for its vertex batch into a large, one-dimensional array of doubles that is then sent into the MPI collective to be broadcast to all of the other Chefs. This process is repeated for each subsequent ray-trace iteration until the user-specified limit is reached. One benefit of this iterative approach is that it could easily support intermediate checkpointing as a future extension.

The last step in the baking process involves writing the baked data out to a new Bake Shop file. In the vertex section of the file, each Chef is assigned a chunk in which it can write data. The length of each chunk, which is constant across all Chefs in a single Kitchen, is dynamically determined based on the number of vertices and viewpoints. Each Chef uses parallel MPI I/O operations to write out the viewpoint data for its own vertex batch within its assigned chunk of the output file. Once all of the vertices and viewpoints have been written in parallel, the Chef with MPI rank 0 writes out the face data sequentially. Because the MPI framework adds null bytes as padding between file chunks, a post-processing stage removes all null bytes from the output file.

While the Chef program is optimized to be executed in parallel, it can be effectively serialized in the degenerate case of a single MPI rank. Additionally, to support serial execution on computer systems that do not support MPI, the codebase can use dependency injection during compilation to emulate a single MPI rank using UNIX system calls and the C standard library.

3.3 Visualization Program

For the final rendering code, we used mostly code provided by Dr. Barbara Cutler. This consisted of a cross platform OpenGL based application to view meshes. With some modifications to allow the reading of the bakeshop file format, along with an addition of a new module that packed the triangles, we could view our resulting rendered meshes.

Implementing the bakeshop file loading consisted of using the existing library created for chef and including along with some externs to allow cross language linking. Implementing the new module for our custom mesh packing took a little more work.

Breaking the mesh of quads into triangles for OpenGL was fairly simple given we were using quads as our representation. Getting the colors for each triangle used a custom KD-Tree**[1]** that mapped every precomputed angle on a vertex to its color for a very fast look up. Additionally, in order to speed up lookups and reduce representation size, the KD-Tree was flattened into array in a heap-like structure in which a node at array children were at index $2n + 1$ and $2n + 2$, see figure 1. Both of these things allowed very efficient sampling of expected colors from any angle, and allowed the real time viewing of data-dense meshes that resulted from the baking process.

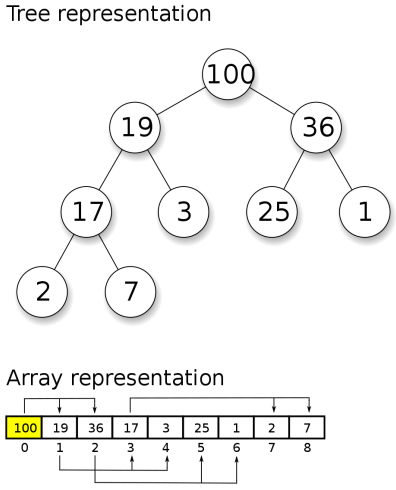

Figure 2: Heap Tree Representation[1]

[1] By Kelott - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=99968794

3.4 Choice of Sample Points

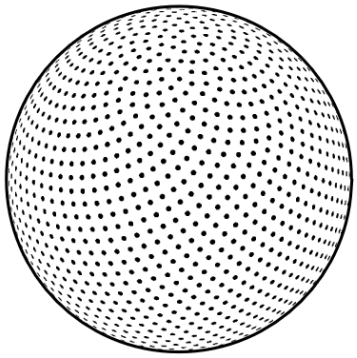

Figure 3: Fibonacci Point Distribution of 1000 Points [2]

So far, we have talked about catching the color of vertices from different angles, but we haven't yet talked about how those angles were chosen. We wanted to choose angles on the points that would be easy to generate, consistently spaced around the sphere as to avoid any obvious visual artifacts, and to allow easy sampling. To these ends we settled on the Fibonacci point distribution**[2]**.

As you can see in figure 3, the points are qualitatively very evenly distributed. Though additionally, we found that quantitatively, by using equation (1) to make our sample radius always gave us about 8 points, plus or minus 2. In this equation, r_s is the sample radius and N is the number of Fibonacci points on the sphere.

$$r_s = \sqrt{32/N} \qquad (1)$$

One can rederive this formula by taking the surface area of a sphere of radius 1 and the area of a circle with radius r and solving for r where the area of the circle is 8/N that of the sphere's surface area.

4 Results

After much fiddling with CUDA and MPI configurations, the program works as expected. The Chef can take Bake Shop files and produce a rendered final version with the correct material properties. These files are then able to be passed to the visualizer to be viewed in real time.

The resulting renders can be seen in figure 4, note that they can be moved around and viewed in real time. The rendering times will be discussed in depth, but briefly, the Cornell Box-inspired mesh shown in a-f took about 20 minutes to render using 4 cores on a standard desktop computer while the bunny took about 10 minutes. Both ray-trace maps used 1000 samples per vertex.

4.f shows the initial (incorrect) result that we had seen; the ray tracing code was not interpolating the color for triangles correctly leading to this bumpy result. After eventually finding that we were using the same color multiple times were we shouldn't be, we were able to get the much better results in the rest of the table.

4.b shows a visual artifact as a result of our approach: the fact that the colors are stored with the vertices means that colors are calculated on a per-vertex basis, so when a vertex lies on a corner, rays are bounced off of it as if it were a rounded edge as the vertex calculates its normal as the average of the surrounding faces.

Figure 4.d shows the per-vertex coloring from the perspective of figure 4.c. This is just to show once again how the actual mesh is being colored using cached colors and the camera's position and no ray tracing is done at the visualization stage.

Figure 4.e and 4.f show another artifact of our approach that is much more difficult to overcome robustly: due to the fact that the color data are stored with the vertices and that the faces interpolate between those vertex colors, the vertices are sometimes easy to see. The grid that's shown in the figure is to help the viewer locate the vertices, which are more brightly green than the surrounding gray mesh.

Finally, figures 4.g and 4.h show a very reflective gray bunny in a room with a red light at the top and a blue light at the bottom. This shows that while big lighting shifts don't get very muddied, the polygons start to become more visible with a more complicated mesh. Due to the nature of any one edge having the same colors on its vertices for both faces it is attached to, we did not expect any edges to be visible, because the linear interpolation of the color should be identical on both sides of the edge. As such, this points to a bug in our rendering code or something similar. Investigation on our part is required.

4.1 Performance Scaling

# Ranks	# Viewpoints	Time
1	20	0.412030
2	40	0.444345

4	80	1.377331
8	160	2.618694

Table 1: Baking times for the "cube" example mesh with 8 vertices and 1 ray-trace iteration on the AiMOS supercomputer.

# Ranks	# Viewpoints	Time
1	20	258.522557
2	40	261.497459
4	80	265.060030

Table 2: Baking times for the "cornell_big_raw" example mesh with 1924 vertices and 4 ray-trace iterations on the AiMOS supercomputer.

# Ranks	Time
1	1293.262907
2	652.941984
4	330.310601

Table 3: Baking times for the "cornell_big_raw" example mesh with 1924 vertices, 100 viewpoints, and 4 ray-trace iterations on the AiMOS supercomputer.

4.2 Analysis of Performance

Our implementation exhibits nearly linear strong scaling, with the rendering time cut roughly in half for each further vertex-batch subdivision. Weak scaling is less effective, with additional I/O overhead from synchronizing between ranks and from writing the viewpoint data to disk in parallel eventually overwhelming the raw compute performance gains, especially at low vertex counts. Note that the current implementation requires that the number of vertices be a multiple of the number of parallel MPI ranks. However, the number vertices in a particular rank's assigned vertex batch need not be a multiple of the number of parallel GPU threads for that MPI rank. In general, dispatching rays to the GPU worsens performance significantly, often taking twice as long as the same computation just on the CPU. We suspect that this is because the KD-tree search is heavily recursive, which can quickly overwhelm the GPU's call stack. Indeed, we must quadruple the default configuration of CUDA's call-stack limit to accommodate the recursive invocations.

4.3 Limitations

Our precomputed rendering technique comes with several limitations. The most significant is that it only works for static scenes. Because the technique shades at the polygon level, very high polygon density is necessary to achieve acceptable visual fidelity. Polygonal artifacts—*i.e.*, discontinuous shading at the boundaries of polygons—are sometimes visible, especially at low polygon or viewpoint density. The discrete nature of the ray-trace map can result in small but unexpected color artifacts when the camera directly "locks onto" a particular predefined viewpoint at runtime. Memory usage during the baking process is high, and the intermediate Bake Shop files can grow to be quite large, often on the order of a few megabytes for a simple scene with a few discrete meshes. This also means that initially loading an intermediate Bake Shop file into the runtime visualizer can cause noticeable lag, though camera manipulation remains smooth with imperceptible latency once that loading process completes. Additionally, the Chef does not currently support more advanced features like texture sampling or distributed ray tracing**[7]**, though we are not aware of any fundamental hurdles to implementing those features in the future.

5 Future Work

Future work could include expanding the capabilities of the Chef, including the addition of support for distributed ray tracing**[7]** and texture sampling. Distributed ray tracing could help reduce polygonal artifacts by averaging the shading contributions from locally distributed rays. This would also play into the vague reflections that our results seem to show this technique favors. Texture sampling could permit sub-polygon shading variance beyond just viewpoint interpolation, but the finer level of detail that that would afford might make polygonal artifacts more pronounced.

STATEMENT OF WORK

Ally Robinson implemented the real-time visualizer, parts of the core ray tracer, and the CUDA parallelization. Gabriel Jacoby-Cooper implemented the Bake Shop data structures and I/O library, parts of the core ray tracer, and the MPI parallelization.

ACKNOWLEDGMENTS

We would like to acknowledge Drs. Barb Cutler and Christopher Carothers at Rensselaer Polytechnic Institute in Troy, NY.

REFERENCES

[1] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. Communications of the ACM 18, 9 (September 1975), 509–517. https://doi.org/10.1145/361002.361007

[2] Álvaro González. 2009. Measurement of Areas on a Sphere Using Fibonacci and Latitude–Longitude Lattices. Math Geosci 42, 1 (November 2009), 49–64. https://doi.org/10.1007/s11004-009-9257-x

[3] Turner Whitted. 1980. An improved illumination model for shaded display. Commun. ACM 23, 6 (June 1980), 343–349. https://doi.org/10.1145/358876.358882

[4] Thomas Alois Pichler, Andrej Ferko, Michal Ferko, Peter Kán, and Hannes Kaufmann. 2022. Precomputed fast rejection ray-triangle intersection. Graphics and Visual Computing 6, (June 2022), 200047. https://doi.org/10.1016/j.gvc.2022.200047

[5] Shenchang Eric Chen and Lance Williams. 1993. View interpolation for image synthesis. Proceedings of the 20th annual conference on Computer graphics and interactive techniques - SIGGRAPH '93 (1993). https://doi.org/10.1145/166117.166153

[6] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. 2002. Ray tracing on programmable graphics hardware. ACM Transactions on Graphics 21, 3 (July 2002), 703–712. https://doi.org/10.1145/566654.566640

[7] Robert L. Cook, Thomas Porter, and Loren Carpenter. 1984. Distributed ray tracing. ACM SIGGRAPH Computer Graphics 18, 3 (January 1984), 137–145. https://doi.org/10.1145/964965.808590

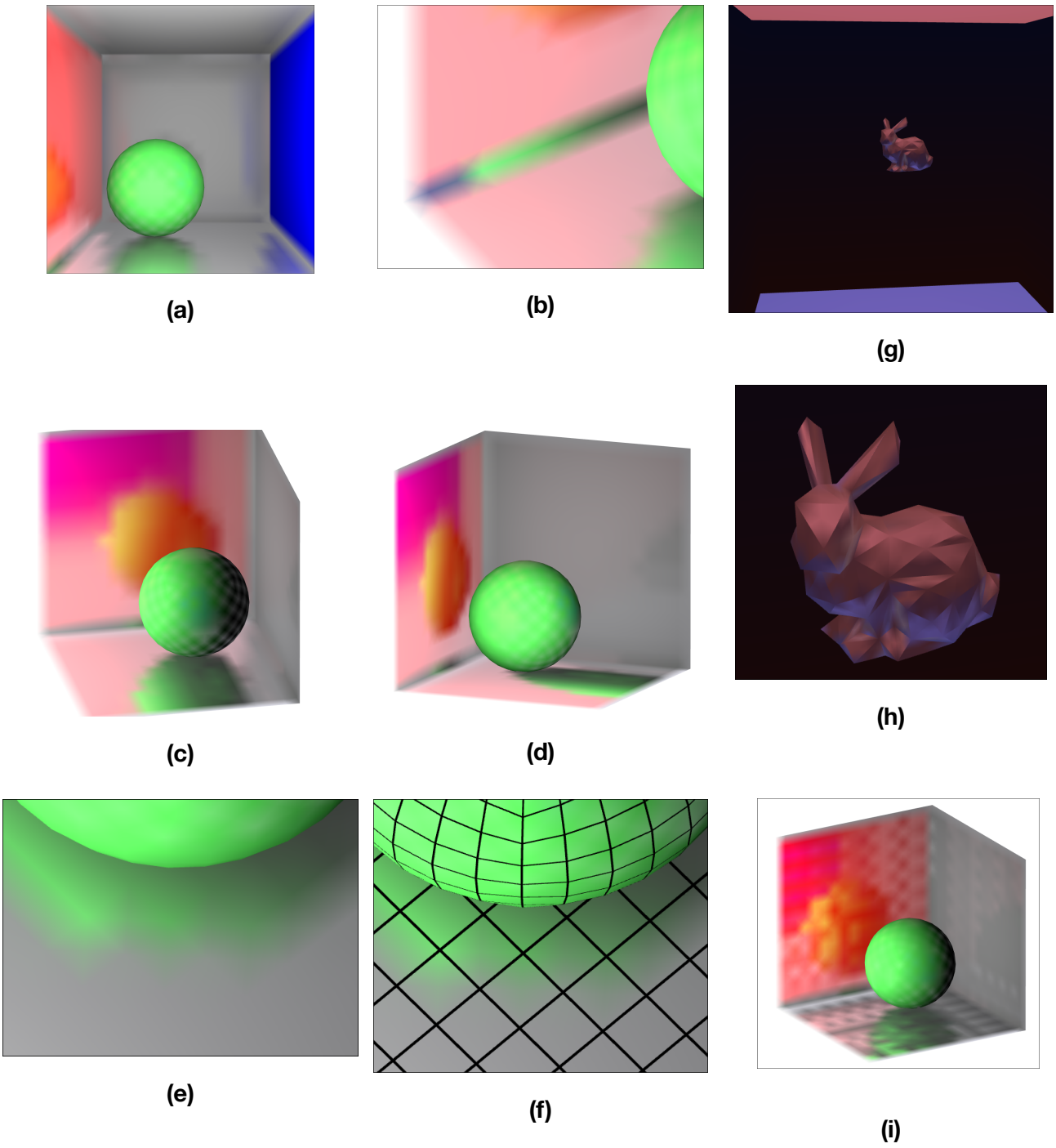

Figure 4: Renderings of Baked Results