Project 0x01: Forward Rays
Due Friday, Sept. 19, 2008 at 6:00 pm

1 Specification

Use the photon mapping algorithm to simulate realistic light transport within a scene consisting of Lambertian reflectors and provide a real-time preview for camera placement using hardware-accelerated rendering.
Your program must provide all of the features from project 0x00, plus:

1. Rectangle geometry
2. Per-entity emission function, which can be NULL or a UniformEmission
3. Per-surface (triangle or Entity) BSDF
   a. LambertianBSDF supporting arbitrary reflectance
4. Real-time preview of the scene when the “Move” button is pressed
   a. W, A, S, and D keys to translate the camera
   b. Mouse to rotate the camera
5. Display of rendering time for:
   a. Forward trace (colloquially called “photon mapping”)
   b. Backward trace (colloquially called “ray tracing”)
6. A GUI including:
   a. A GuiNumberBox for changing the number of photons to be simulated
   b. A GuiNumberBox for changing the number of forward bounces
7. Photon mapping implementation comprising:
   a. Forward tracer that emits photons and traces them using Russian roulette
   b. PointAABSPTree photon map
   c. Backward tracer that estimates radiance using the photon map
8. Triangle mesh geometry
9. Scene must have expected $O(\log n)$ ray intersection time (i.e., using a BSP or Oct-tree)
10. Ability to render any one of four or more scenes, chosen at runtime by a command-line argument:
    a. cornell: A Cornell box with a sphere, using rectangle walls, not planes.
    b. sponza: The Sponza atrium (pictured above)
    c. At least two other scenes of your choice. Creativity and presentation style will be recognized here.

Your documentation must provide overview information on a @mainpage that explains your software design and links to the relevant classes. It must also contain a “results” page or section that discusses the performance (including anything clever you did to improve it) and uses images and text to demonstrate correctness.

### 1.1 Restrictions

Your group submits a single solution through subversion. I encourage you to discuss the project with others outside your group and to help each other debug, derive algorithms, and understand the documentation. The code that you submit should be written solely by your group, except for trivial changes made while someone is helping you debug or code that you obtained externally in accordance with the guidelines from the syllabus. Any code not written by your group must be clearly noted in comments and documentation.

You may use any feature of G3D and C++, except you may not invoke or look at the source code for:

1. All G3D::CollisionDetection methods
2. G3D::Ray::intersection
3. G3D::Ray::intersectionTime
1.2 Evaluation

Your work will be evaluated for the following properties, in order of significance:

1. Satisfaction of the specification
2. Readability, including documentation
3. Mathematical correctness
4. Presentation of results
5. Design (including performance)
6. Programmatic correctness (compiling, memory management, safety)

In the event that you have known bugs, add a documentation section explaining what they are, how you attempted to debug them, and what you think the problem is. Use images as necessary. In general I will not deduct many points for bugs that you are aware of and investigated.

2 Discussion

What are you learning in this project, and why?

In terms of theory, you’re learning the Russian roulette algorithm for importance sampling, which is applicable to any sampling problem, not just rendering. You’re learning to use BSP-trees, a fundamental data structure for computer graphics and machine learning. The process of turning the physics model of light transport into an efficient algorithm for evaluating that model is exactly the kind of work that computer scientists in applied domains (i.e., almost everything except pure theory, like networking, graphics, and AI) do every day. You’re learning both advanced mathematics and the techniques computer scientists use to reduce the computational burden of solving those problems using those mathematics, like numerical integration, approximation, and sampling.

Between theory and practice: by reading Kajiya and Jensen’s original rendering papers, you’re becoming familiar with the structure of a computer graphics research paper and how to implement the algorithm described in one. Research papers are focused on the long-term contribution, not implementation details. Both learning how to reading a paper and learning how to fill in the gaps for implementation are important skills. The “advice” sections of these project handouts are largely necessary because even given the paper and a textbook, there’s a lot of
conventional computer science and graphics know-how needed to bring that theory into practice. The projects are helping you to acquire that. You’ll need less advice in time.

In terms of engineering practice, you’re working with code from a previous project that maybe wasn’t even written by you. Using that code will help you become accustomed to working with existing software; more importantly, the frustrations you encounter will teach you a life-long lesson about how to write your own code so that it is easy for you and others to extend in the future. (We’ll do this again next week, just in case the lesson isn’t clear enough!) From now on, you’ll find that it isn’t about me giving you a grade for design and readability so much as your project partner appreciating the level of effort you put into those aspects.

You’re also working with a program that has a large number of classes, runs on multiple processors with different architectures, and possibly even uses multiple threads on those processors. Designing and debugging this kind of program is an engineering challenge in its own right, and the kind of challenge that you’ll encounter in either graduate school or industrial software development.

3 Advice

As always, these are tips about how I implemented the specification. You are free to solve the problem any way that you want as long as it meets the specification and need not follow my advice.

I suggest that you approach the specification in the order the elements are listed, since the earlier pieces will help you debug the later ones.

At some point in this project, you’re going to find that you are spending a lot of time waiting for your algorithm to run. Don’t just sit there reading e-mail! One of the challenges of working with large data sets and algorithms with long run times is making them fast enough that your debugging cycle is reasonable. There are several techniques that you can employ to bring down the running time for faster debugging:

1. Use a smaller imager. Backtracing time is linear in the area of the imager. In the extreme, a 1x1 Film works very well for debugging the basic ray cast and photon gather.
2. Run an optimized build. icompile --opt --run will produce a binary that is 2x to 10x faster. It also will crash very hard if anything goes wrong.
3. Forward trace fewer photons. Forward trace time (except for the tree balance) is linear in the number of photons. You have to increase your neighborhood radius when doing this to get useful results.
4. Forward trace fewer bounces. Forward tracing is linear in the number of bounces.
5. Multi-thread your program.

Multithreading is very easy under G3D if you are careful to avoid synchronization problems. I made BackTracer and ForwardTracer subclasses of G3D::GThread. This allows you to create multiple instances of them, throw those instances into a G3D::ThreadGroup, and then run the group until all threads terminate. For BackTracer, have each instance render a different portion of the screen (e.g., assign thread \( i \) out of a group of \( k \) threads to render lines on the imager where \( y = kj + i \)). For ForwardTracer, have each thread trace \( N/k \) of the \( N \) total photons. All of the queries against the BSP trees don’t modify the trees, so they are inherently threadsafe. Writing to the Film on multiple threads is safe as long as you don’t touch the same pixel (which you shouldn’t be doing). The catch is that you can’t insert into the photon map on multiple threads at the same time.
time. So when you multithread, you need to make the forward tracers accumulate their photons into one array per forward tracer, and then when all forward tracers have completed dump those arrays into the photon map on a single thread.

Note that if you have more than two threads, your program likely won’t get much faster. There are only two processor cores on our machines. It is a good idea to set your program up so that you can turn off multithreading as well—that way you can quickly trouble-shoot problems based on the threads themselves.

Ray-rectangle intersection is an extension of ray-plane intersection. RTR3 16.9 has a more general discussion, but the easiest way to do it is to make a test similar to the ray-triangle one in RTR 16.8 and then figure out how to test against a rectangle instead of a triangle, given \( u \) and \( v \). When implementing ray-rectangle and ray-triangle intersections, I recommend storing the \((u, v)\) values for the closest intersection, say in your Hit class. Those values will be handy next week for a feature called “texture mapping”.

The “real time preview” requirement has one part that is trivial and one part that is straightforward. The trivial part is that if your RayTrace class inherits a defaultController member from GApp. If you enable that controller, it will automatically track W, A, S, D and mouse (when the right mouse button is held down) to “fly” the defaultCamera around the scene. So that part is basically done for you.

The straightforward part is that you need to use OpenGL to draw an approximation of the scene while you’re flying, so that you can see where you are flying. The way to do this is to replace the code that draws the Film with one that draws in 3D in onGraphics:

1. Push the current state of the RenderDevice.
2. Set the RenderDevice viewport to have the same dimensions as the Film.
3. Set the RenderDevice’s camera and projection matrices based on the camera that you’re using.
4. Invoke a Scene::drawPreview method.
5. Pop the old state of the RenderDevice.

In your Scene::drawPreview method you need to draw each of the Entities. You can do this in two ways. The first way is to use the G3D::Draw class, which knows how to draw most primitives. Make a BSDF::previewColor method so that you can chose the right color for each primitive.

The second way to draw the Entities is to issue individual vertices using RenderDevice::sendVertex inside a begin/end primitive section. This is essentially the same as how you drew the gradient background last week, except that this week you’re drawing 3D points instead of 2D ones (and maybe you want to draw TRIANGLES instead of QUADS for some of the Entities). If you want to get really fancy, you can draw outlines over the surfaces by switching to RenderDevice::RENDER_WIREFRAME mode and drawing the surfaces in black before you draw the surfaces with color.

When using OpenGL, you do not need to worry about the order in which the Entities are drawn. There is a data structure used by the OpenGL renderer that resolves order and only shows the first intersection with an eye ray on the screen. This data structure is called a z-buffer and will be discussed in class in about four weeks when we cover GPU rendering.
To accomplish the \(O(\log n)\) goals in this project you’re going to use a kind of 3D tree called a Binary Space Partition Tree and a related tree called a scene graph. You can read about these in RTR3 14.0, 14.1, 17.1 and 17.2. We’ll discuss them in lecture during the week. Since G3D implements one for you in AABSPTree, you can get started and use the BSP tree before you really understand how it works, though. The right mental model for this data structure is that it is a set of Entities that proves \(O(\log n)\) ray, box, and sphere intersection by using an internal tree. That tree takes a long time \(O(n^2 \log n)\) in the worst case!) to create, so you’ll create it once when the scene is loaded and then just perform queries at runtime.

Since MeshEntity has a lot of triangles inside it, you’ll want one BSP tree for the scene itself that works off the Entity’s bounding box and each MeshEntity will also have another BSP tree inside itself for managing its triangles.

The photon mapping algorithm is described in the reading and will be presented in class. In brief, the forward tracing process is:

Emit \(N\) random photons from light sources, sampling uniformly with respect to intensity. For each photon:
1. Bump the photon a small distance along its propagation direction
2. Normalize the photon’s radiance
3. Find the first intersection of the photon’s ray of transport with the scene.
4. Store the photon in the photon map
5. Scatter the photon according to the BSDF at the surface. Possible outcomes are:
   a. Scattered:
      i. Change the propagation direction to the one selected by the BSDF
      ii. Modulate the photon radiance by the BSDF’s reflectivity (i.e., shift the colors)
      iii. Return to step 1
   b. Absorbed: stop simulating this photon

It is a good idea to implement “return to step 1” as a while loop around the whole per-photon simulation. If you instead used recursion, you’d find that your program ran a lot slower because it takes time to create the stack frame for each method invocation. The algorithm above may run for an arbitrary (but finite, assuming reflectivities are less than 1) amount of time. One way to rein this in is to count the number of times a photon has bounced and simply stop simulating it after a fixed number of bounces, like six. Don’t forget to detect the case where the photon didn’t hit anything in the scene—in that case, you should definitely stop simulating it.

After forward tracing has completed, balance the photon map and then run the backwards tracer. Modify your backwards tracer so that when it encounters a surface at location \(x\), it estimates the radiance at that location as follows:

1. Find \(P\), the set of all photons within a neighborhood (small sphere) of \(x\). These photons represent measurements of incident radiance at \(x\). The exitant radiance is the sum of the BSDF \(f\) applied to each:
2. Let \(sum = 0\)
3. For each photon \(p\) in \(P\):
   a. \(sum += f(x, \omega_{in}, \omega_{out}) \cdot p.radiance\)
4. The net radiance is \(sum / (N \cdot \text{cross-section area of neighborhood})\)
Note that in the last step, you are dividing by the number of photons emitted, not size of $P$ or the number of photons in the photon map!

This is slightly different from the method that Jensen describes in two ways. This approach is faster but produces some sampling artifacts. Your surfaces will look scuffed up. Jensen corrects this in two ways (both of which are optional for you), which trade quality for time. The first way is that he performs a “final gather” step that instead of estimating the radiance at $x$ directly instead shoots a large number of rays outward, measures the incoming radiance due to those points (using the algorithm above), and then applies the BSDF to those incident radiance values. This effectively performs one extra backwards bounce, and that smooths out the result.

The second technique that Jensen uses is that he doesn’t use a fixed gather radius for the neighborhood sphere. Instead, he grows his neighborhood until there are a constant number of photons within it, and then he applies the above algorithm. This ensures that the variance is always constant in the sample, which has the net effect of smoothing out the illumination function samples on surfaces. Unfortunately, that growing process means issuing multiple sphere-BSP tree queries, each of which is fairly expensive.

The basic algorithm also has the problem that some light can leak through walls because your neighborhood is a sphere that might poke through the wall. One (optional) way to fix this is to squash the sphere. You can’t query the BSP tree for a sampled sphere, but you can throw away samples that lie outside sliced-up sphere after you’ve sampled. As a hint, the basic algorithm for this is to only include photons for which:

$$| (p.\text{position} - \text{hit}.\text{position}) \cdot \text{hit}.\normal | \leq r$$

where $r$ is the radius of the sliced sphere perpendicular to the surface (i.e., a number much smaller than the radius of your query sphere). If you don’t see why this takes a slice off the top and bottom of the sphere, don’t use it: for enough photons and a small enough photon gather neighborhood, the difference is negligible.

MeshEntities are constructed from ArticulatedModels. The ArticulatedModel structure is defined in the G3D documentation, and you can look at its source code (and post to the G3D user forum) if you find anything in the documentation unclear. This class is a scene graph, which is a tree (or actually, a collection of trees, since it can have multiple roots) of parts. Each part is expressed in a coordinate frame relative to its parent. This is so that, for example, moving the torso of a human model would automatically cause the arm and hand to move in such a way as to stay connected. We’ll explore coordinate frames in two weeks. For now, all that you need to know is that G3D::CoordinateFrame represents one, that multiplying two coordinate frames composes them, and that the “toWorldSpace” methods transform a point, direction, or normal out of the reference frame of that coordinate system.

My MeshEntity class maintains an AABSPTree of MeshEntity::Tri triangles (not G3D::Triangles, which don’t have enough information.) Its constructor walks the tree(s) of an Articulated model and flattens them into a set of these Tris, which it then inserts into the BSP tree. The way that I converted the ArticulatedModel to a MeshEntity was as follows:

1. Iterate over the ArticulatedModel::partArray
2. For each ArticulatedModel::Part part that is a root (has a parent index of -1):
   a. Compute myFrame = parentFrame * part.cframe
   b. For each ArticulatedModel::Part::TriList in part:
i. Compute a BSDF for the trilist
   ii. For every three sequential indices i,j,k in triList.index:
      1. Extract the vertices, normals, and textureCoordinates
         corresponding to index[i], index[j], and index[k]
      2. Transform the vertices by myFrame.pointToWorldSpace
      3. Transform the normals by myFrame.normalToWorldSpace
      4. Create a triangle and insert it into the BSP tree
   c. Recurse into all children of part (which you can find by looking at
      part.subPartArray), passing myFrame as their parent’s frame

Be careful to extract vertexArray[indexArray[i]], not vertexArray[i], and so on for j and k. Don’t forget to update the relevant bounding geometry for your MeshEntity if you have any.

You can find the Sponza model on the course website. It is in a zipfile. You don’t have to unzip it—G3D can automatically load data from inside a zipfile. So just use ArticulatedModel::fromFile(“sponza.zip/sponza.3DS”) to load the data. You’ll have to add some emitters to the scene. I used a giant dim blue rectangle to represent the sky and a small yellow rectangle to represent the sun for the images I rendered.

ArticulatedModel can load any IFS, PLY2, or 3DS file. You can find some of these in the G3D data directory in /usr/local/371. You can find more on the web, for example at the Brown Mesh Set, the Princeton Shape Benchmark, 3Dcafe, and TurboSquid. Many 3DS editors output slightly incorrect files that will confuse ArticulatedModel, and ArticulatedModel also doesn’t support all of the 3DS features. So don’t be too surprised if you load an arbitrary 3DS file and it doesn’t look right. Just try another one. The ability to load arbitrary models gives you the potential to make some really cool scenes…