CHAPTER EIGHTEEN

18 RETROSPECTIVE AND THE FUTURE

pbrt represents one single point in the space of rendering system designs. The basic decisions we made early on—that ray tracing would be the geometric visibility algorithm used, that physical correctness would be a cornerstone of the system, and that Monte Carlo would be the main approach used for numerical integration—all had pervasive implications for the system's design. An entirely different set of trade-offs would have been made if pbrt were a renderer designed instead for real-time performance or for maximum flexibility for artistic expression. This chapter looks back at some of the details of the complete system, discusses some design alternatives, and also discusses some potential major extensions to the system that are more complex than have been described in the exercises.

18.1 DESIGN RETROSPECTIVE

One of the basic assumptions in pbrt's design was that the most interesting types of images to render are images with complex geometry and lighting. We also assumed that rendering these images well—with good sampling patterns, ray differentials, and anti-aliased textures—is worth the computational expense. One result of these assumptions is that pbrt is relatively inefficient at rendering simple images.

For example, on a 2.5-GHz dual-core CPU, pbrt takes 0.4 seconds to render a 512×512 image with no geometry or lights in the scene—a black image! 25% of this time is generating pseudo-random numbers and low-discrepancy sample points. Simple scenes with a few dozen geometric primitives and a small number of lights do not take much longer than this, indicating that the fixed per-ray cost for these parts of the computation dwarfs the expense of computing intersections and shading for trivial scenes. All of the effort used to compute samples and ray differentials for camera rays and to add the

filtered contribution of the rays to the image is clearly a substantial fraction of the time spent rendering simple scenes.

For a more complex scene, the time spent finding ray intersections, evaluating textures, and applying Monte Carlo integration algorithms dominates running time, and the relative time spent on sample generation and image sample filtering becomes less important. (For a rendering of the TT car scene, for example, pseudo-random number and sample point generation consumed 7% of run time.) Because we believe that complex scenes are the most important (and interesting) types of scenes to render, the fact that some design decisions in the system hurt performance for simple scenes was never considered a pressing problem to address.

Another performance implication of our design approach is that finding the BSDF at a ray intersection is more computationally intensive than it is in renderers that don't expend as much effort filtering textures and computing ray differentials. We believe that this effort pays off overall by reducing the need to trace more camera rays to address texture aliasing, although, again, for simple scenes, texture aliasing is often not a problem. `pbrt` generally performs better as more samples are taken to compute exitant radiance at a given intersection point. For example, increasing the number of shadow rays traced to an area light source amortizes the antialiasing work done in computing the BSDF's textures, while tracing more camera rays to reduce noise in shadows is less efficient, since a BSDF needs to be computed at each of their intersection points.

The simplicity of some of the interfaces in the system can lead to unnecessary work being done. For example, the `Sampler` always computes lens and time samples, even if they aren't needed by the `Camera`; there's no way for the `Camera` to communicate its sampling needs. Similarly, if an `Integrator` doesn't use all of the samples requested in its `GetSamples()` method for some ray, the `Sampler`'s work for generating those samples is wasted. This case can occur, for example, if the ray doesn't intersect any geometry.

Another example of potentially wasted computation is that `Shapes` always compute the partial derivatives of their normal $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$, even though these may not be needed. Currently, they are only used if the `BSDF` has specular components and ray differentials are being computed for the reflected or refracted rays. There's no way for the intersection routine to know if they will be needed at intersection time, however, so their values are always computed. Indeed, there's currently no way to know if the BSDF will have specular components at ray–shape intersection time. The BSDF must be created by the `Material` for this information to be known, and the `Material` needs the differential geometry at the intersection point to compute the BSDF!

One way to address this shortcoming would be to allow the material to conservatively describe all of the fields in the `DifferentialGeometry` that it needs—for example, by setting flags. These flags could be passed to the `Shape` intersection routines, which could then skip setting the member variables that aren't needed. This approach could further save execution time by allowing shapes to skip computing (u, v) parametric coordinates if they weren't needed, and so on.

BSDF 478
Camera 302
DifferentialGeometry 102
Integrator 740
Material 483
Sampler 340
Shape 108

18.1.1 ABSTRACTION VERSUS EFFICIENCY

One of the primary tensions when designing interfaces for software systems is making a reasonable trade-off between abstraction and efficiency. For example, many programmers religiously make all data in all classes `private` and provide methods to obtain or modify the values of the data items. For simple classes (e.g., `Vector`), we believe that approach needlessly hides a basic property of the implementation—that the class holds three floating-point coordinates—that we can reasonably expect to never change. Of course, using no information hiding and exposing all details of all classes' internals leads to a code maintenance nightmare. Yet, we believe that there is nothing wrong with judiciously exposing basic design decisions throughout the system. For example, the fact that a `Ray` is represented with a point, a vector, and values that give its extent, time, and recursion depth is a decision that doesn't need to be hidden behind a layer of abstraction. Code elsewhere is shorter and easier to understand when details like these are exposed.

An important thing to keep in mind when writing a software system and making these sorts of trade-offs is the expected final size of the system. The core of `pbrt` (excluding the implementations of specific shapes, lights, and so forth), where all of the basic interfaces, abstractions, and policy decisions are defined, is under 10,000 lines of code. Adding additional functionality to the system will generally only increase the amount of code in the implementations of the various abstract base classes. `pbrt` is never going to grow to be a million lines of code; this fact can and should be reflected in the amount of information hiding used in the system. It would be a waste of programmer time (and likely a source of run-time inefficiency) to design the interfaces to accommodate a system of that level of complexity.

18.1.2 DESIGN ALTERNATIVE: TRIANGLES ONLY

Another instance where the chosen abstractions impact the overall system efficiency is the range of geometric primitives that the renderer supports. While ray tracing's ability to handle a wide variety of shapes is elegant, this property is not as useful in practice as one might initially expect. Most real-world scenes are either modeled directly with polygons or with smooth surfaces like spline patches and subdivision surfaces that either have difficult-to-implement or relatively inefficient ray–shape intersection algorithms. As such, they are usually tessellated into triangles for ray intersection tests in practice. Not many shapes that are commonly encountered in real-world scenes can be described well with spheres and cones!

There are some advantages to designing a ray tracer around a single low-level shape representation like triangles and only operating on this representation throughout much of the pipeline. Such a renderer could still support a variety of primitives in the scene description but would always tessellate them at some point before performing intersection tests. Advantages of this design include the following:

- The renderer can depend on the fact that the triangle vertices can be transformed into world space in advance, so no transformations of rays into object space are ever necessary.

Ray 66
Vector 57

- The acceleration structures can be specialized so that their nodes directly store the triangles that overlap them. This improves the locality of the geometry in memory and enables ray–primitive intersection tests to be performed directly in the traversal routine, without needing to pass through two levels of virtual function calls to do so, as `pbrt` does now.
- Displacement mapping, where geometry is subdivided into small triangles, which can then have their vertices perturbed procedurally or with texture maps, can be more easily implemented if all primitives are able to tessellate themselves.

These advantages are substantial, for both increased performance and the complexity that they remove from many parts of the system. For a production renderer, rather than one with pedagogical goals like `pbrt`, this alternative is probably worth pursuing.

18.1.3 INCREASED SCENE COMPLEXITY

Given well-built acceleration structures, a strength of of ray tracing is that the time spent on ray–primitive intersections grows slowly with added scene complexity. As such, the maximum complexity that a ray tracer can handle may be limited more by memory than by computation. Because rays may pass through many different regions of the scene during a short period of time, virtual memory often performs poorly when ray tracing complex scenes due to the incoherent memory access patterns.

One way to increase the potential complexity that a renderer is capable of handling is to reduce the memory used to store the scene. For example, `pbrt` currently uses approximately 600 MB of memory for the one million triangles in the ecosystem scene in Figure 4.1. This works out to 600 bytes per triangle, if all memory use (acceleration structures, geometry, textures, and materials, etc.) is amortized over the geometric complexity. We have previously written ray tracers that managed an average of 40 bytes per triangle for scenes like these, including all memory overhead. Clearly, much reduction is possible.

Reducing memory overhead requires careful attention to memory use throughout the system. For example, in the aforementioned system, we provided three different `Triangle` implementations, one using 8-bit `uint8_t`s to store vertex indices, one using 16-bit `uint16_t`s, and one using 32-bit `uint32_t`s. The smallest index size that was sufficient for the range of vertex indices in the mesh was chosen at run time. Deering's paper on geometry compression (Deering 1995) and Ward's packed color format (Ward 1991b) are both good inspirations for thinking along these lines. See the "Further Reading" section in Chapter 4 for information about more memory-efficient acceleration structure representations.

A more complex approach is geometry caching (Pharr and Hanrahan 1996), where the renderer holds a fixed amount of geometry in memory and discards geometry that hasn't been accessed recently. This approach is useful for scenes with a lot of tessellated geometry, where a compact higher-level shape representation like a subdivision surface can explode into a large number of triangles. When available memory is low, it can be worthwhile to discard some of this geometry and regenerate it later if needed.

Triangle 139

The performance of such a cache can be substantially improved by reordering the rays that are traced in order to improve their spatial and thus memory coherence (Pharr et

al. 1997). An easier-to-implement and more effective approach to improving the cache's behavior was described by Christensen et al. (2003), who wrote a ray tracer that uses simplified representations of the scene geometry in a geometry cache. More recently, Yoon et al. (2006) and Budge et al. (2009) have developed improved approaches to this problem. See also Rushmeier, Patterson, and Veerasamy (1993) for an example of how to use simplified scene representations to compute indirect illumination.

18.2 THROUGHPUT PROCESSORS

After many years of the CPU being the heart of the computer system, recent innovations in processor design have led to a new type of high-performance processor being available in many computers. Starting with data-parallel graphics processors (GPUs) at the start of the 21st century, these new processors have made significantly more bandwidth and raw computational power available to applications than CPUs have. As of the time of writing, the hardware architectures and programming models that they support are rapidly changing. As such, this version of `pbrt` is not written to run on these new architectures. However, we will briefly discuss the development of these new processors and some topics related to implementing rendering systems like `pbrt` on them.

CPUs have long been designed to run a single thread of computation as efficiently as possible; these processors can be considered to be latency focused, in that the goal is to finish a single computation as quickly as possible. (Only since 2005 or so has this focus started to slowly change in CPU design, as multicore CPUs have provided a small number of indepdendent latency-focused processors on a single chip.) Starting with the advent of programmable graphics processors around the year 2003, *throughput processors* have increasingly become the computational heart of many computer systems. These processors focus not on single-thread performance but instead on efficiently running hundreds or thousands of computations in parallel with high aggregate computational throughput, without trying to minimize time for any of the individual computations. Major hardware vendors developing such throughput processors in 2010 include AMD, Intel, and NVIDIA.

By not focusing on single-thread performance, throughput processors are able to devote much less space on the chip for caches, branch prediction hardware, out-of-order execution units, and other features that have been invented to improve single-thread performance on CPUs. Thus, given a fixed amount of chip area, these processors are able to provide many more arithmetic logic units (ALUs) than a CPU. For the types of computation that can provide a substantial amount of independent parallel work, throughput processors can keep these ALUs busy and very efficiently execute the computation. As of the time of writing, graphics processors offer approximately ten times as many peak FLOPS as high-end CPUs; this makes them highly attractive for many processing-intensive tasks (including ray tracing).[1]

1 However, graphics processors typically consume more power and are physically larger chips than CPUs; some of their improved performance comes purely from using more power and more chip area. A fair comparison would be to consider performance per watt or per square millimeter of silicon.

Single instruction, multiple data (SIMD) processing, where processing units execute a single instruction across multiple data elements, is the key mechanism that throughput processors use to efficiently deliver computation; both today's CPUs and today's throughput processors have SIMD vector units in their processing cores. Today's multicore CPUs provide a handful of processing cores and SIMD instruction sets like SSE and AltiVec to provide four-element vector SIMD instructions on each core, while current throughput processors have tens of processing cores,[2] each with SIMD vector units between 8 and 32 elements wide. It is likely that the number of processing cores on both types of processors will go up over time, as hardware designers make use of additional transistors made possible by Moore's law. The width of the vector units is likely to change more slowly; Intel has announced an eight-element SIMD instruction set to be launched in 2010, but this is the first increase in CPU SIMD vector width on x86 processors since 1999.

For narrow SIMD widths (like four-element SSE), reasonable performance gains can be attained by opportunistically using the SIMD unit. For example, one might modify `pbrt` to use SSE instructions for the operations defined in the `Spectrum` class, thus generally being able to do three floating-point operations per instruction (for RGB spectra) rather than just one if the SIMD unit was not used. This approach would achieve 75% utilization of the SSE unit for those instructions, but doesn't help with performance in the rest of the system. In some cases, optimizing compilers can identify multiple computations in scalar code that can be executed together on SIMD units. However, achieving *excellent* utilization of SIMD vector units, especially for the SIMD vector units on throughput processors, generally requires that the entire computation be expressed in a *data parallel* manner, where the same computation is performed on many data elements simultaneously.

A natural way to extract data parallelism in a ray tracer is to have each processing core responsible for tracing n rays at a time, where n is at least the size of the SIMD width, if not larger; as such, each SIMD "lane" is responsible for just a single ray, and only a single scalar computation is performed for each of n rays at once with each instruction. Thus, high SIMD utilization comes naturally, except for the cases where some rays require different computations than others.

This approach has seen success with high-performance CPU ray tracers (where it is generally called "packet tracing"). Wald et al. (2001a) introduced this approach, which has since seen wide adoption. In a packet tracer, the camera generates "packets" of n rays which are then processed as a unit. Acceleration structure traversal algorithms are modified so that they visit a node if *any* of the rays in the packet pass through it, primitives in the leaves are tested for intersection with all of the rays simultaneously, and so forth. Packet tracing has been shown to lead to substantial speedups, although it becomes increasingly less effective as the rays to be traced become less coherent; it works well for eye rays and shadow rays to localized light sources, since the packets of rays will pass through similar regions of the scene, but efficiency generally falls off with multi-

2 The definition of a "core" on a throughput processor is notoriously tricky, with different hardware vendors promoting different definitions. Here, we are following the relatively vendor-neutral terminology proposed by Fatahalian (2008).

Spectrum 263

bounce light transport algorithms. Finding ways to retain good efficiency with packet tracing is an active area of research.

Packet tracing on CPUs is usually implemented with the SIMD vectorization made explicit: intersection functions are written to explicitly take some number of rays as a parameter rather than just a single ray, and so forth. In contrast, the vectorization in programs written for throughput processors is generally implicit: code is written as if it just operates on a single ray at a time, but the underlying compiler and hardware actually execute one instance of the program in each SIMD lane. For processors that directly expose their SIMD nature in their instruction sets (like CPUs or Intel's planned Larrabee throughput processor), the designer of the programming model is able to choose whether to provide an implicit or an explicit model to the user. See, for example, Parker et al.'s (2007) ray tracing shading language for an example of compiling an implicitly data-parallel language to a SIMD instruction set on CPUs. See also Georgiev and Slusallek's (2008) approach, where a generic programming approach is taken in C++ to allow implementing a high-performance ray tracer with details like packets well hidden.

Both the explicit and the implicit vectorization models have their advantages and disadvantages; the implicit model is generally easier for the programmer, though the explicit model allows wider visibility into the instances of computation being performed in the lanes of the vector unit, which can make more run-time optimizations possible. In either case, the programmer who cares about high performance generally must consider the effects of SIMD execution when designing algorithms and tuning performance.

18.2.1 THE FUTURE

Innovation in high-performance architectures for graphics seems likely to continue in coming years. As CPUs are gradually increasing their SIMD width and adding more processing cores, becoming more similar to throughput processors, throughput processors are adding support for task parallelism and improving their performance on more irregular workloads than purely data-parallel ones. Whether the computer system of the future is a heterogeneous collection of both types of processing cores or whether there is a middle ground with a single type of processor architecture that works well for a range of applications remains an open question.

The role of specialized fixed-function graphics hardware in future systems remains undecided as well; while recent hardware architectures have increased programmability and generality while reducing the role of fixed-function hardware, fixed-function hardware can be substantially more efficient than programmable hardware. As the critical computational kernels of future graphics systems become clear, fixed-function implementations of them may be worthwhile.

18.2.2 FURTHER RESOURCES

Purcell et al. (2002, 2003) and Carr, Hall, and Hart (2002) were the first to map general-purpose ray tracers to throughput graphics processors. While these systems do not provide many of the features available in `pbrt`, they demonstrate the feasibility of using GPUs for fast ray tracing.

Many researchers and developers have investigated the best methods for harnessing the computational power of SIMD instruction sets and throughput processors for rendering. Reshetov et al. (2005) generalized packet tracing, showing that gathering up many rays from a single origin into a frustum and then using the frustum for acceleration structure traversal could lead to very high-performance ray tracing; they refined the frusta into subfrusta and eventually the individual rays as they reached lower levels of the tree. Reshetov (2007) later introduced a technique for efficiently intersecting a collection of rays against a collection of triangles in acceleration structure leaf nodes by generating a frustum around the rays and using it for first-pass culling. See Benthin and Wald (2009) for a recent technique for using ray frusta and packets for efficient shadow rays.

Data-parallel construction of data structures tends to be challenging; see Zhou et al. (2008) and Lauterbach et al. (2009), respectively, for techniques for building kd-trees and BVHs on data-parallel throughput architectures. As throughput processors start to support task parallelism in addition to data parallelism, this task will become somewhat easier for the programmer.

The biggest current challenge in using throughput processors for rendering systems can be finding coherent collections of data-parallel computation. Consider a Monte Carlo path tracer tracing a packet of n rays; after random sampling at the first bounce, each ray will in general intersect completely different objects, likely with completely different surface shaders. At this point, running the surface shaders will likely make poor use of SIMD hardware. This specific problem of efficient shading was investigated by Hoberock et al. (2009), who resorted a large number of intersection points to make coherent collections of work before executing their surface shaders.

A number of researchers have developed methods for reordering small batches of incoherent rays to improve SIMD efficiency; representative work in this area includes papers by Mansson et al. (2007), Boulos et al. (2008), Gribble and Ramani (2008), and Tsakok (2009). Aila and Laine (2009) carefully investigated the performance of SIMD ray tracing on a graphics processor, using their insights to develop a new SIMD-friendly traversal algorithm that was substantially more efficient than the previous best known approach. Their insights are worth careful consideration for implementors of high-performance rendering systems.

An interesting trade-off for renderer developers to consider is exhibited by Hachisuka's path tracer, which uses a rasterizer with parallel projection to trace rays, effectively computing visibility in the same direction for all of the points being shaded (Hachisuka 2005). His insight was that, although this approach doesn't give a particularly good sampling distribution for Monte Carlo path tracing, in that each point isn't able to perform importance sampling to select outgoing directions, the increased efficiency from computing visibility for a very coherent collection of rays paid off overall. In other words, for a fixed amount of computation, so many more samples could be taken using rasterization versus using ray tracing that the much larger number of less well-distributed samples generated a better image than a smaller number of well-chosen samples. We suspect that this general issue of trading off between computing exactly the locally desired result at a single point versus computing what can be computed very efficiently globally for many points will be an important one for developers to consider on the throughput processors of the future.

18.3 CONCLUSION

The idea for pbrt was born in October 1999. Over the next five years, it evolved from a system designed only to support the students taking Stanford's CS348b course to a robust, feature-rich, extensible rendering system. During that time, we have both learned a great deal about what it takes to build a rendering system that doesn't just make pretty pictures, but is one that other people enjoy using and modifying as well. What has been most difficult, however, was designing a large piece of software that others would enjoy *reading*. This has been a far more challenging (and rewarding) task than implementing any of the rendering algorithms themselves.

After 2004, the book enjoyed widespread adoption in advanced graphics courses worldwide, which we have found very gratifying. We were unprepared, however, for the impact that pbrt has had on rendering research. Writing a ray tracer from scratch is a formidable task (as so many students in undergraduate graphics courses can attest to), and creating a robust physically based renderer is much harder still. We are proud that pbrt has lowered the barrier to entry for aspiring researchers in rendering, making it easier for researchers to experiment with and demonstrate the value of new ideas in rendering. Both of us continue to be delighted to see papers in SIGGRAPH, the Eurographics Rendering Symposium, High Performance Graphics, and other graphics research venues that either build on pbrt to achieve their goals, or compare their images to pbrt as "ground truth." We would like to sincerely thank everyone who has built upon this work for their own research, to build a new curriculum, or just to learn more about rendering. We hope that this new edition continues to serve the graphics community in the same way that its predecessor was able to.