DOI:10.1145/2447976.2447996

Technical Perspective
The Ray-Tracing Engine that Could

By Matt Pharr

AN EFFECTIVE APPROACH to building flexible software systems has been to make them extensible through embedded scripting languages. Languages like TCL, Python, and Lua have allowed programmers to orchestrate and customize the behavior of many software systems—examples include games, which are mostly written in C++ but often have AI for characters and other gameplay mechanics implemented in Lua, and high-performance computing applications written with Python code coordinating the execution of C++ or Fortran library code.

Most embedded scripting languages are interpreted, and thus are not suitable for the implementation of performance-critical inner loops. Furthermore, the runtime overhead of passing through the interface between the core system and the embedded language is too much to accept in many performance-focused domains, especially when frequent transitions are needed. Thus, most performance-oriented systems have not offered the option of programmability in the heart of their performance-critical parts.

However, in graphics, a programmable high-performance rasterization pipeline has been at the heart of interactive rendering for the last decade. Programmers of modern graphics processing units (GPUs) provide code for the pipeline's inner loops, writing "shaders" in C-based languages like HLSL or GLSL. The shader programming model is data-parallel; this model provides abundant parallelism, which maps well to the underlying SIMD hardware architecture. Although programmable GPUs have now been used in many domains beyond graphics for high-performance computation, it has been an open question whether it is possible to build GPU-targeted high-performance software systems that are themselves programmable.

The following paper by Parker et al. shows how to achieve both programmability and high performance in such a system. The domain of their system, OptiX, is interactive ray tracing for image synthesis. Ray tracing is a very flexible approach to rendering, and can simulate many important lighting effects more effectively than rasterization, but its use in interactive graphics has until recently been limited—prior to OptiX, users had the choice of high-performance ray-tracing systems that were insufficiently flexible, or highly flexible ray-tracing systems that had insufficient performance.

The authors have developed an elegant expression of the classic ray-tracing algorithm as a programmable data-flow graph assembled from user-supplied code at each stage. OptiX supplies highly optimized implementations of core geometric and parallel work scheduling algorithms that run between stages. Just like the GPU rasterization pipeline, the programmer has full control of the system's behavior at key points of programmability without needing to worry about the gritty details of high-performance GPU programming. OptiX uses the CUDA language for the user-supplied kernels; CUDA provides a data-parallel programming model that runs with high efficiency on GPUs.

OptiX achieves programmability without sacrificing performance by eliminating the barrier between the core OptiX system code and the code provided by the user. It applies a specializing JIT compiler to both collections of code, allowing for not just inlining across the boundary between the two parts of the system but also for constant propagation and dead code elimination, thus generating a specialized version of the system. The core OptiX system can thus provide functionality that may not be needed in the end; the code for such functionality is removed when the system is compiled together.

This system implementation approach allows users of OptiX to implement, for example, custom representations of the 3D scene geometry as well as custom algorithms to simulate lighting and reflection—both key areas for customization in ray tracing. An indication of the authors' success in designing the right decomposition of the problem is the wide variety of applications of ray tracing—spanning not just rendering, but even audio simulation and collision detection—that have been implemented with OptiX. The resulting systems are close enough to peak efficiency that OptiX has quickly become the standard foundation for most GPU ray tracing.

One of the unexpected successes of the introduction of the programmable rasterization pipeline on GPUs has been the creativity programmers have shown in using the GPU rasterization pipeline in ways never imagined by its original designers. By putting both flexible and high-performance ray tracing in the toolbox of many more developers than before, it seems quite likely that OptiX will spark innovation in ways that are impossible to predict today.

This paper is a must-read for anyone who cares about writing extensible software systems that are also high-performance software systems. Although the target hardware architecture for this work is GPUs, the underlying ideas are equally applicable to high-performance software systems on CPUs. Today, with the availability of high-quality compiler toolkits like LLVM, the barrier to entry for implementing all sorts of systems in this manner is now quite low, while the potential advantages are significant. C

Matt Pharr (matt.pharr@gmail.com) is a software engineer in the Google [x] group at Google, Inc., Mountain View, CA.