

White Paper
by Orion Granatir &
Ryan Shrout

Performance Scaling with Cores: Introducing the SMOKE Framework and Demo

After processor technology improvements slowed with the gigahertz race in 2004, a new era in consumer computing emerged with the first multi-core processor to reach mainstream markets. Along with the doubling of processing cores came the promise of doubling the performance, but now, years later, we all know that's still not the case for some of our favorite applications. Utilizing the full capacity of multi-core processors requires multi-threaded applications, and developing properly threaded software is a different and more difficult task than traditional coding.

Game development is one area that has lagged behind in the progression of multi-threaded software. With a typical coding structure that involves basically one enormous while loop, games have epitomized single-threaded programs that fed off the pure speed of the processors involved in the gigahertz race. Now that we are moving in the direction of “more cores” rather than “more clocks,” game developers need to adjust their programming models and mindsets to take advantage of the hardware available to the PC and even console gaming communities.

As any modern coder knows threading can be difficult, but although initially constructing a many-threaded game engine requires a significant effort, the reward will no doubt be worthwhile and could span years into the future. (For more details, refer to “Designing a Parallel Game Engine” by Jeff Andrews on <http://whatif.intel.com>.)

As a manufacturer of multi-core processors, Intel obviously has an interest in promoting the proper techniques that will utilize the full horsepower of its processors. With this intent in mind, Intel developed Smoke, a dual-purpose framework and tech demo that emphasizes the advantages of multi-threaded gaming that is correctly implemented. It also shows game developers how thread management, resource sharing, and workloads can be balanced to create a highly modularized and flexible gaming engine for both today’s and tomorrow’s CPUs.

What is Smoke?

Smoke is a model framework that maximizes the performance of the processor in a purely gaming environment. Built to take advantage of all available threads, it works equally efficiently on standard dual-core Intel® Celeron processors as well as on new Intel® Core™ i7 Processors with Intel® Hyper-Threading Technology. The Smoke video demonstration, shown at many trade shows and technology events, uses modern game-development technologies, including Havok for physics processing, FMOD for audio playback, Ogre* 3D and DirectX* 9 for graphics rendering, and more. As you would expect for an internally developed demo, the code shows the Smoke framework as a well-partitioned and configurable product.

Intel developed Smoke mainly as a teaching tool to demonstrate the ability to create a framework that can scale to any number of threads. Developers are encouraged to explore the technology by examining new threading techniques and learning about the interactions between various game engine systems that typically hold back a game’s potential threadability. Intel’s goal through this and other efforts is to help prove that multi-threaded gaming can be done effectively and is an investment in time that is worth taking.

The Smoke Framework

The simplicity of Smoke’s framework design (see Figure 1) is obvious once it is broken down into its various pieces, and doing so clearly shows how the software enables hardware with N-threads to function efficiently.

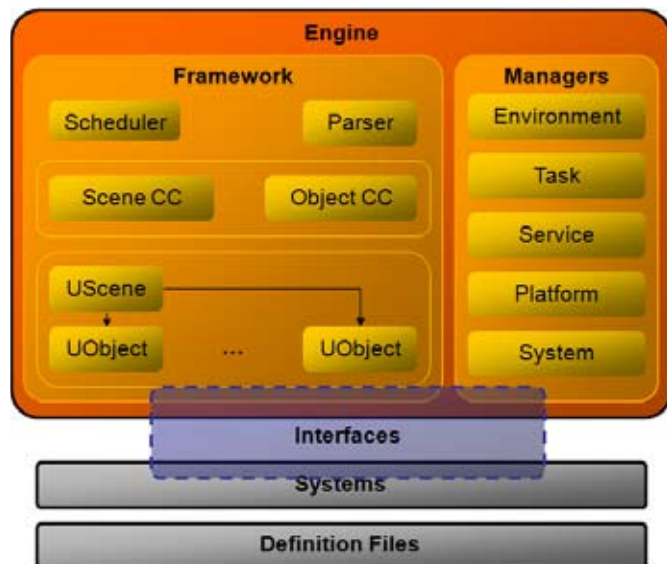


Figure 1: Smoke framework.

The framework’s design is built around a Scheduler that manages system jobs and a Change Control Manager (CCM) that is responsible for minimizing the traditional drawback of threaded games: thread synchronization. The data structures share data between the various jobs and threads in the Smoke framework and are designed to support independent processing and system modularity with easy-to-develop interfaces.

The Scheduler is the logical component that manages the various system jobs that are determined by each individual system. Once each system implements the proper interface, that system is able to interact with the framework easily and efficiently.

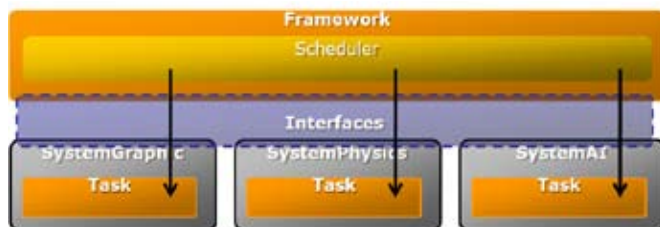


Figure 2: The CCM is responsible for inter-system communications.

The CCM (see Figure 2) reduces the framework’s dependence on heavy thread synchronization operations. For example, if the physics API system changes an object that is also registered with other systems, instead of sending that change to them directly, the other

systems are notified of the change at the end of the current frame, allowing them to continue processing the current frame independently.

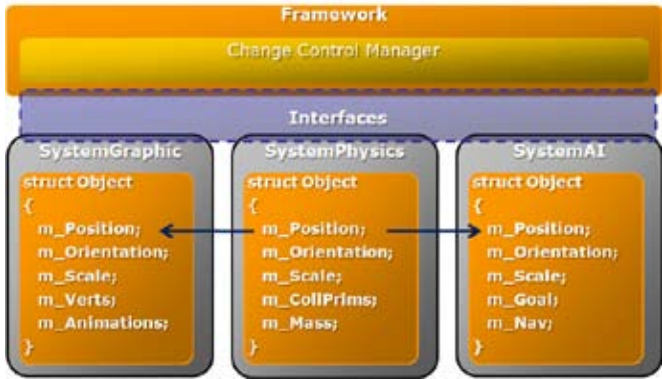


Figure 3: SystemAI and SystemGraphic are subscribing to position changes from SystemPhysics.

The registration of objects between systems occurs during initialization of the framework using the interfaces developed in Smoke. When a system modifies its data it tells the CCM, and that information is passed on during the next frame refresh (see Figure 3). The CCM does not pass the data through to each subscribed system, but instead leaves it to the system itself to copy the data only if necessary for its present operation.

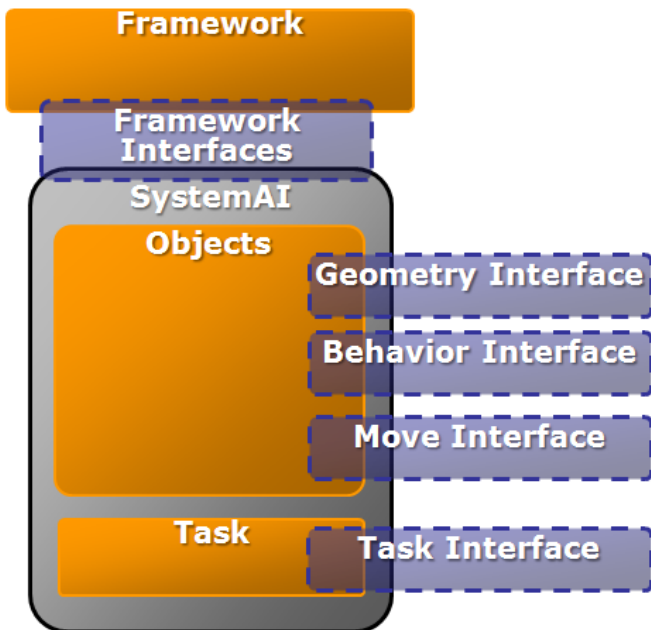


Figure 4: Interfaces make communication between systems easy.

To maintain Smoke’s highly modular design, interfaces were developed between the various systems and the framework itself—a key feature. The framework’s job is to enable communication between the various systems (see Figure 4). For example, we might have a geometry inter-

face for changing the positions of models or a behavior interface for changing AI states of specific objects. The Scheduler uses the task interface to schedule and invoke work with each system.

Running through a typical frame process in Smoke helps clarify this activity. Imagine a single frame in the engine being rendered, where the process starts with the framework’s systems subdividing the tasks for processing. The Scheduler invokes each system per frame allowing the system to naturally divide its own work into granular pieces that can be broken up into various jobs. Competent middleware is able to accomplish this quite easily, which minimizes the Scheduler’s necessary work.

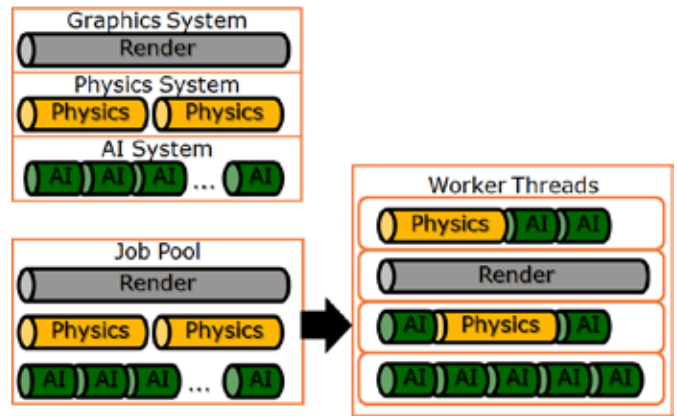


Figure 5: Worker threads are assigned jobs from the pool of available processing work.

Once all of these tasks have been created, they are collected into a single pool that all available worker threads can access. A properly designed framework allows for one worker thread per core, and this is where the power and scalability of the design really starts to play out. Each job in the pool is then assigned to a thread (see Figure 5) based on the load of each particular thread and the framework’s estimation on how processor-intense each job is. The one area that the Smoke developers admit is problematic is in this step: Properly balancing the load of jobs across each thread requires some kind of pre-determined knowledge about the work in each job. Obviously in a very lightweight framework, such as the one we are describing, that information is difficult to come by.

Cache coherency on the processor is another important issue to keep in mind when subdividing tasks. In the most optimal setup threads should work on blocks of data rather than random or interleaved data, allowing the processing core to access cache and memory resources in a more linear, and thus quicker, fashion.

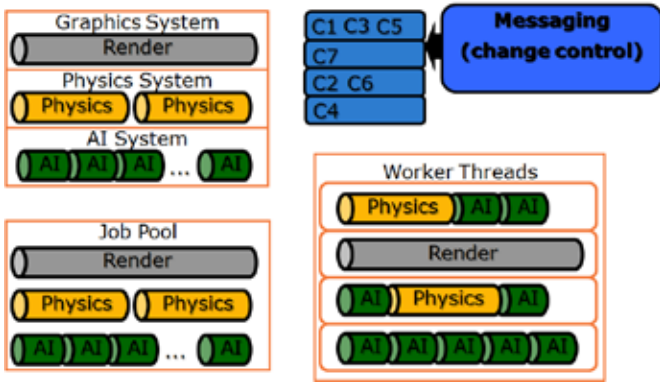


Figure 6: Each of the worker threads has a change queue that is accessed by the CCM.

During the processing of the current frame, the various worker threads and jobs post messages to the CCM (see Figure 6) indicating updates to the status of any registered objects. Once the frame has been completed, the CCM sends those messages on to the appropriate systems that have subscribed to the updated objects before the next frame's processing begins, allowing the updated data to propagate through the framework. And thus, the cycle repeats, and we have a fully threaded, independent-processing model for a game engine.

Smoke Demo at Work

To demonstrate the framework's power, Intel built a graphical demo that allows a viewer to visualize the framework's performance advantage. The scene depicts a farmhouse under siege by meteor strikes, which are causing raging fires to erupt in the surrounding trees (see Figure 7).



Figure 7: The Smoke demo running with all eight processing threads active.

The Smoke demo allows the viewer to control the number of worker threads that can access jobs created by the framework. Matching the number of threads to the total number of accessible threads in a system (eight threads with the Intel Core i7 Processor) offers the best

performance, but Intel's demo allows for running on multiple iterations of thread counts in order to see the performance gains the framework provides.

Figure 7 shows the demo running with eight total threads with a frame rate of 74 fps on our Intel Core i7 Processor-based test system. You can see that all eight CPUs in the performance box are evenly loaded, and the total CPU utilization is in the upper 80 percent range. You can also see the breakdown in workload for each system in the demo, including procedural fire, trees, AI, physics and so on.



Figure 8: The same area of the demo using four threads of processing.

Here we reduced the thread count in Smoke from eight threads to four and with the demo set at four worker threads as in Figure 8, the system uses approximately 46 percent of the total CPU power and an average frame rate of 49 fps—a 50 percent drop from our eight-thread performance.



Figure 9a: With only two active threads the performance of the demo is affected drastically.



Figure 9b: The same work can be done with just a single worker thread, although performance is affected.

Figures 9a and 9b show the same scene running with two worker threads and a single worker thread, respectively; performance drops more than significantly in each downward step. The demo runs more than 6.7x faster with eight worker threads than with a single thread, which proves that with a properly implemented multi-threaded game engine developers can really push performance on systems with any number of processing cores.

What Smoke Means Today

While we might immediately jump to the conclusion that this multi-threaded framework was built with Intel's new many-core graphics architecture in mind, adopting Smoke-like designs in today's gaming engines has clear benefits. With the Intel Core i7 Processor now on the scene, systems with four hyper-threaded cores (and thus allowing for eight logical threads) are going to be streaming into the gaming scene. Taking advantage of that processing power is definitely a challenge, and a design similar to what we have described in this article will allow game developers to use much more of the available processing horsepower than do gaming titles today.

Intel's software development team built Smoke simply to demonstrate that an efficiently threaded gaming engine for any number of threads is possible. While we do not expect to see game developers pick up on the design directly and port this exact framework to their own games, Intel does present the code freely to all game developers at <http://Whatif.intel.com>, helping to educate the market on what is possible with today's and tomorrow's CPU hardware.

Having software that can completely utilize the hardware in our PCs is not just a neat technology topic, but is important to current and future computing on a higher level as the demands for power-efficiency and green technologies take a front-row seat in the global market. And while we definitely need to see increased efficiency in all areas of processing, gaming is equally important and equally lucrative: After all, what's the use of all the work without a little play? The Intel Smoke framework is a fantastic forward-looking example of how software developers can utilize Intel hardware for the best possible user experience.

About the Authors

Orion Granatir is a senior engineer with Intel's Visual Computing Software Division. He is the Tech Lead on the Smoke project. Prior to joining Intel in 2007, Orion worked on several PlayStation 3* titles as a senior programmer with Insomniac Games. His most recent published titles are Resistance: Fall of Man*, and Ratchet and Clank Future: Tools of Destruction*.

Ryan Shroul started his career with computers at an early age when he created a Web site for hardware enthusiasts while still in high school. Today, more than eight years later, this Web site has grown into one of the largest technology editorial destinations. As the editor-in-chief at PC Perspective (pcper.com), Ryan turns a critical eye on CPU and GPU designs, with a focus on their underlying technologies and the real-world benefits they may offer.

