High-Performance Games: Addressing Performance Bottlenecks with DirectX*, GPUs, and CPUs
by David Conger

The last decade has seen tremendous advances in the field of computer graphics. These advances, often driven by the game industry, are proliferating to users at an amazing rate. Household computers are becoming capable of levels of realism in computer graphics that were completely out of reach just a few years ago.

Graphics software is also undergoing a series of rapid innovations. In particular, a look at the evolution of the Microsoft DirectX* graphics application programming interface (API) shows how advances in DirectX work together with new hardware to overcome many of the bottlenecks common to games.

This article presents an overview of common graphics pipeline bottlenecks and explains how recent advances in DirectX and graphics processing hardware overcome many of them.

A Typical Graphics Pipeline

A bottleneck is anything in the hardware or software that limits the processing throughput of the system. Graphics programs, such as games, generate visual output by following a specific sequence of processing steps, usually referred to as a graphics processing pipeline. Figure 1 shows a simplified view of a typical graphics processing pipeline.

Figure 1. A typical graphics processing pipeline

This view of the graphics pipeline is generic, so it is not a reflection of any particular hardware or software. It’s designed to emphasize the hardware rather than the software that runs the pipeline, because modern graphics hardware often implements the graphics pipeline in hardware.

This illustration indicates which tasks the CPU performs and which tasks the graphics processing unit (GPU) handles. The first thing you may notice is that the GPU does most of the graphics processing tasks, as you might expect. However, this can be a source of problems.
Before moving on, it’s important to note that the term GPU here applies more widely than it has in the past. Specifically, modern computers—especially laptops—often integrate the graphics hardware onto the motherboard. Newer hardware even integrates GPU functionality into the CPU. An example is Intel® HD Graphics, which is built into Intel® Core™ processors.

*Note:* Intel HD Graphics enables CPUs to provide the same basic functionality as a GPU-based video card, but at a much lower cost.

**Bottlenecks in the Graphics Pipeline**

Figure 2 indicates the areas where processing bottlenecks tend to occur.

![Figure 2. Common pipeline bottlenecks](image)

As you can see from this illustration, the pipeline contains some specific types of slowdowns and backups in the flow of data. At the beginning of the hardware pipeline, the CPU fetches geometry data from storage across the computer’s bus. It then sends the data to the GPU’s geometry storage, which is implemented in the graphics hardware’s memory. If this data transfer slows the pipeline, the pipeline is said to be *CPU bound* or *bus bound*—that is, the CPU or the bus is the bottleneck.

The simplest way to find out if your application is CPU bound is to test it with the Intel® Graphics Performance Analyzers (Intel® GPA) and Intel® VTune™ Performance Analyzer tools.

*Note:* Intel GPA is the latest suite of software tools from Intel for conducting platform-level game performance analysis. It runs on both Intel and non-Intel hardware, including components from NVIDIA* and AMD*. When run on Intel hardware, Intel GPA is more accurate and gives access to unique features not available on non-Intel components.

Next, the geometry processor pulls the 3D geometry data from storage and converts it to pixels. This process is called *rasterizing*. If there is a slowdown in this section of the pipeline, then the pipeline is said to be *vertex bound*.

Finally, the hardware pipeline processes the individual pixels by applying textures and executing pixel shaders and other pixel-based operations. Slowdowns at this stage mean that the pipeline is *pixel bound*. If a pipeline is vertex bound or pixel bound, you can also say that it is *GPU bound*.

Of course, this simplified discussion of bounding conditions in the graphics pipelines just scratches the surface of the topic. But it’s important to understand the basic terminology of pipeline optimization so that you can see how the DirectX API addresses these problems.
Graphics Pipeline Bottlenecks and Microsoft DirectX* 9

When it arrived in December 2002, Microsoft DirectX* 9 was a good step forward in generating realistic graphics. It added many new improvements. However, the most important features for the current discussion were advances that addressed bottlenecks in DirectX device drivers. Previous implementations of DirectX caused games to be CPU bound, because each call to the device driver generated so much overhead that it limited the number of objects the CPU could send to the pipeline. Adding special effects for increased realism only resulted in more device driver calls, which made the problem worse.

In addition, DirectX 9 had specific memory constraints. As a result, it limited the number of textures that games could use to improve the appearance of their graphics. To get around this limitation, developers often had to apply textures in multiple passes. Loading extra textures made their games more CPU bound. Making multiple passes through the Texture Filtering stage of the pipeline caused games to be GPU bound.

Graphics Pipeline Bottlenecks and DirectX* 10

Microsoft DirectX* 10 made significant strides in addressing the bottlenecks inherent in previous versions. First, Microsoft redesigned the DirectX device drivers to run partially in kernel mode—a privileged state that the operating system runs in—and partially in user mode—the normal operational state that your programs run in. Moving some of the driver functions out of kernel mode means that DirectX 10–based programs stay in user mode more often, which results in fewer mode switches that tie up the CPU.

Note: Microsoft built the DirectX 10 driver model on a technology that only ships with Windows Vista* and later Windows* operating systems. That means games written with DirectX 10 do not run in Windows XP*.

On the GPU side of things, DirectX 10 provides virtual memory for the GPU. The GPU can use portions of the CPU's memory space as its own virtual memory, essentially giving it an “unlimited” memory space. Games can now use many more—and much larger—textures than ever before.

Graphics Pipeline Bottlenecks and DirectX* 11

Arriving with Windows* 7, Microsoft DirectX* 11 accommodates advanced threading on multi-core CPUs. With the new threading capabilities, games can attain greater frame rates and increased visual detailing by dividing the rendering process and distributing it across multiple processor cores.

In the past, games used multithreading by creating a rendering thread, a networking thread, an audio thread, and so forth. Nevertheless, the task that most taxed the CPU and the GPU—rendering—still ran on a single thread. DirectX 11 enables you to split rendering onto multiple
threads by using immediate and deferred device contexts. Immediate device contexts execute commands immediately, while deferred device contexts send their output to command lists.

How can you use this behavior to improve performance? One approach is to create a pool of rendering threads. The main thread uses an immediate device context, while all other threads in the pool target deferred device contexts. You divide the rendering tasks among the deferred context threads in the pool. When the deferred threads finish building their command lists, you execute the lists on the main thread that has the immediate device context.

**DirectX* 11 and Simple Economics**

On a more basic level, the support Microsoft DirectX* 11 offers for multi-core CPUs creates a larger, more consistent market for game developers to release products into. To understand why, think a bit about the computers that a typical consumer buys. Although hardcore gamers buy the latest and greatest hardware, the average consumer prefers hardware that usually isn’t on the cutting edge. Therefore, their computers may not have a graphics card with support for all of the features you may want to use in your game. However, as of this writing, new computers being sold into the consumer market do contain at least dual-core CPUs. Many contain quad-core CPUs. It’s actually more reasonable for game developers to rely on a multi-core CPU being present than the latest graphics card. In fact, if the consumer’s computer contains an Intel® Core™ processor, which provides integrated Intel® HD Graphics, a graphics card may not even be present in the computer at all!

Even if a GPU is present, the ability of DirectX 11 to scale easily across multi-core CPUs means a better balance between the activity of the CPU and the GPU. Therefore, your games do not need to face many of the bottlenecks present with previous versions of DirectX.

**DirectX* 11 Right Now**

Of course, to get the full benefit of Microsoft DirectX* 11 and its multithreading features, you must have hardware that is built for DirectX 11. However, it’s possible that users’ hardware may not natively support many of the features of DirectX 11. But that should not prevent you from taking advantage of it: DirectX 11 runs fine on DirectX 10-compatible hardware through the use of feature levels. Feature levels enable backward compatibility with previous hardware. So, even if your user doesn’t have DirectX 11-compatible hardware, you can still use DirectX 11 features such as multithreaded rendering for significant performance improvements.

Be aware that using feature levels to run DirectX 11 software on DirectX 10-compatible hardware does not provide the same performance that running the software on DirectX 11-compatible hardware does. However, it does provide access to multithreaded rendering. Therefore, you should still be able to derive a performance boost even on DirectX 10-compatible hardware.
Looking Beyond DirectX*

Whether or not a module of code depends on DirectX*, there are many techniques for optimizing away bottlenecks. Most of them depend on today's multi-core CPUs.

For instance, a racing game can use multithreading to pre-fetch future scenery, handle network data (for multi-player networked games), and even perform some of the rendering tasks well in advance. If there is a branch in the road ahead, the game can use multithreading to pre-compute the scenery of both paths and discard the one the user doesn’t take. Although it may seem a waste of computing cycles, CPUs with as many as eight processing pipelines make this approach practical.

Another example is dividing the game’s tasks by category and assigning the categories to a thread running on its own processor core. That is, all of the physics can be assigned to its own thread, all of the game’s artificial intelligence can run on another thread, texture loading and decompressing can run on yet another thread, and so on.

In general, game engines should do as much as they can to take advantage of today's massively parallel environments. Tools such as Intel® Parallel Studio, Intel® Threading Building Blocks (Intel® TBB), and Intel® Thread Checker go a long way toward making multithreaded programming as pain-free as possible.

With the adoption of the performance-boosting features of DirectX 11 and with a balanced relationship between the CPU and the GPU that includes highly parallel computing, yesterday's performance bottlenecks can become exactly that: yesterday's performance bottlenecks.

For More Information


About the Author

David Conger is a software engineer, author, and former Professor of Computer Science and Business Computer Programming. During his career, he developed software for parallel-processing real-time
display controllers for military aircraft, has written games, and has written seven books about computers and computer programming as well as two books for children. He has written extensive developer documentation for Microsoft Corporation, mostly dealing with graphics, game programming on Windows* and the Xbox*, embedded software development, and distributed network programming. He currently writes SDK documentation for a variety of clients and develops software for the Apple* iPhone*, iPod*, and iPad*. 