The Scalability

The coexistence of high-end systems and value PCs can make life hell for game developers.

DEAN MACRI, INTEL

Back in the mid-1990s, I worked for a company that developed multimedia kiosk demos. Our biggest client was Intel, and we often created demos that appeared in new PCs on the end-caps of major computer retailers such as CompUSA. At that time, performance was in demand for all application classes from business to consumer. We created demos that showed, for example, how much faster a spreadsheet would recalculate (you had to do that manually back then) on a new processor as compared with the previous year's processor. The differences were immediately noticeable to even a casual observer—and it mattered. Having to wait only 10 seconds for something that previously took 20 or more was a major improvement and led many consumers and businesses to upgrade their PCs.

Things have changed considerably since then, aside from talking about processor speeds in gigahertz rather than megahertz. Not every stand-alone application requires the computing power that a top-of-the-line processor presents today. As a result, the PC market has diverged into a wide range of market segments. From $400 "budget" PCs to $4,000 "hotrod" models, there's something for everyone and one size certainly doesn't fit all.

For game developers, what was once a relatively easy game (pardon the pun) of "writing for the top-end and your game will sell" has become a daunting task of

Problem

The Scalability Problem

creating games with scalability. A game must be able to run on a low-end system usually dictated by the publisher's minimum system requirements, yet innovate in some way to garner the praises of reviewers and early buyers to spark sales. One way of innovating is to take advantage of new technologies and performance that enhance visuals and enable previously out-of-reach capabilities to create a better game-play experience for end users.

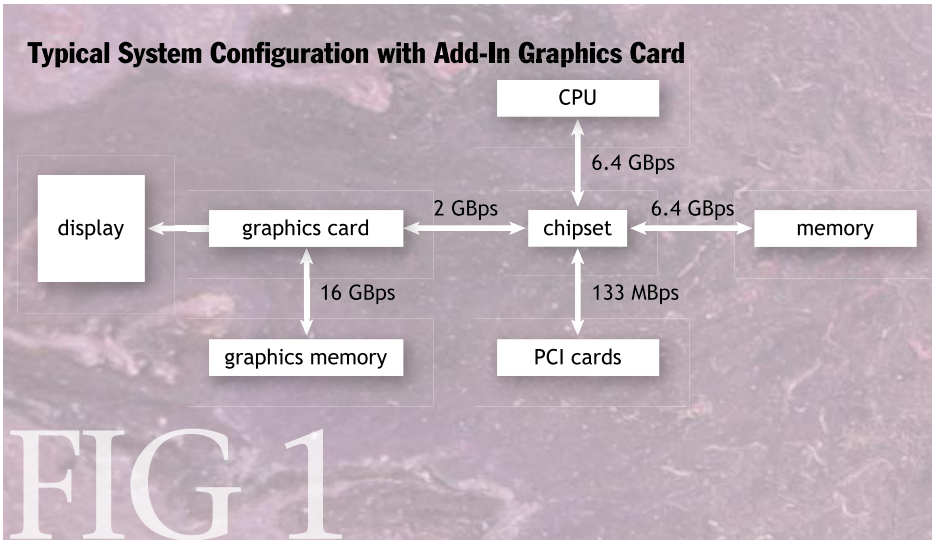
In this article I address a few aspects of this challenge facing game developers. I start out by defining scalability more clearly, take a look at the components having the most influence on the problem, and then examine ways to address scalability in a game. I also discuss a few processor trends and investigate how they can be applied to the scalability challenge to improve PC games in the future. Since we're a long way from photorealistic games, there's plenty of opportunity for scalability, and we'll need to take some intermediate steps to get there. Let's take a look at what some of those steps might be.

WHAT IS SCALABILITY?

The term *scalability* is defined by dictionary.com as: "How well the solution to some problem will work when the size of the problem increases."[1] This definition fits game developers when applied to multiplayer networked games where the number of concurrent users indicates the *size of the problem*. In more common usage among game developers, however, scalability refers to the challenge of making a game that runs acceptably across system configurations that may vary in features, performance, or both. The challenge isn't restricted to just the processor—or even a single computer in the case of multiplayer games. Whereas each component in the system—chipset, memory, hard-drive, networking, sound card, and so forth—plays a role, the two pieces of the scalability challenge that are often the most significant are the processor and the graphics subsystem. These two pieces of the puzzle are typically interconnected to a high degree.

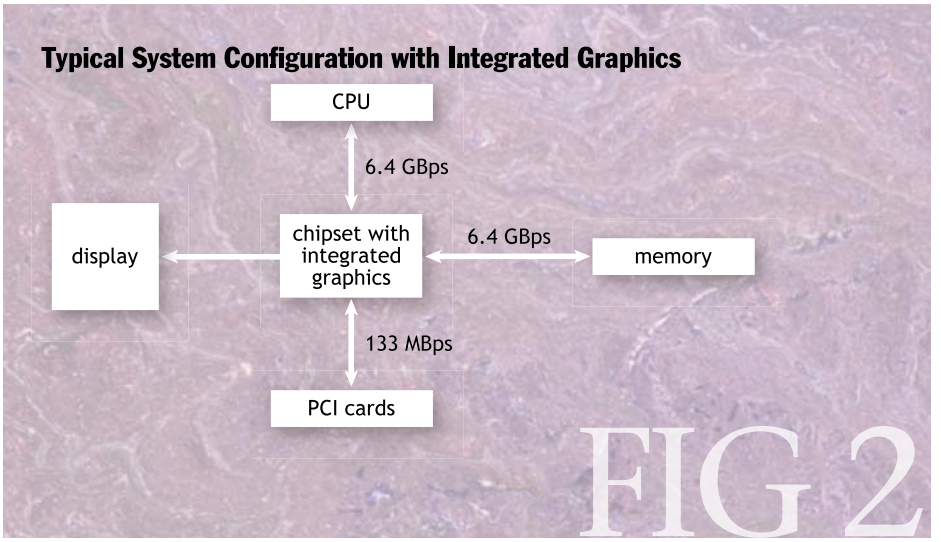
Figures 1 and 2 show the two most common configurations with some typical bandwidths for the various components in the system. Notice that in the configuration with the add-in graphics card (figure 1), the card has its own memory for storing data necessary for rendering. In the integrated graphics configuration (figure 2), the main system memory is shared by applications running on the processor, as well as by the graphics rendering engine.

To take advantage of features introduced with state-of-the-art graphics hardware, a high-performance processor needs to be coupled with it to feed the data used to generate complex visual scenes. A high-end processor may be able to simulate complex physical systems at interactive frame rates, but not being able to render them with sufficient visual fidelity quickly makes the graphics subsystem the weakest link in the chain. In addition, other combinations of processors and graphics hardware are quite common, and game developers have to devise solutions that can give each end user the best experience possible for that user's system configuration. Trade-offs must be made. Choosing the right trade-offs without alienating large classes of gamers is the heart of the scalability challenge. Let's take a look at how these trade-offs typically happen.



APPLYING SCALABILITY

The development cycle for a PC game can range anywhere from several months to four years or more, depending on the scope of the project, the



intended audience, and the budget. For four-year projects, the high-end system at the start of development will probably be the low-end system when the game ships, so developers can often begin the project with that system as their target. Of course, as new technologies surface, the developers must take advantage of some of them or be faced with a game that's lackluster compared with one that was on a two-year development timeline and could take advantage of the leading-edge technologies.

Regardless of the development time frame, publishers usually impose minimum system requirements that encompass systems that were high-end anywhere from three to five years prior. Taking into account just the two key components—the processor and the graphics hardware—game developers must consider a number of configurations. On one end of the spectrum is the bare minimum: an old processor and old graphics hardware. In that case, everything is probably scaled back to the basics necessary for the game to be functional, but there likely won't be any bells or whistles over what last year's titles could achieve. On the opposite end of the spectrum, those who just bought brand new PCs with really fast processors and high-end graphics hardware will want to show their friends how awesome their games run on the premium configuration.

Developers' options for addressing scalability across the gamut of configurations can be narrowed to three techniques:

1. Create just one version of the game that runs with mediocre performance on a system that meets the minimum system requirements. Someone with a higher-performance system will get better performance, but only up to a certain level. At some point, the incremental increase in performance will not be measurable (e.g., beyond a certain frame rate). The feature set will be the same regardless of the system on which the game is played.

2. Create two or more *bins* of performance that are either selected dynamically by profiling the performance of the system or are configurable menu items that end users can choose. Typically, the installation program determines the level of performance and configures in-game options accordingly. Gamers can then choose to alter those choices at the expense of performance or quality. Enhanced features that don't affect game-play are usually enabled for the top bins and disabled for the bottom bins. Far and away, this is the most commonly used mechanism for introducing scalability into a game.

3. Use techniques that are more scalable, such as dynamic elimination of triangles from meshes, or infinitely scalable such as NURBS (nonuniform rational B-splines) or implicit surfaces. The challenges associated with making dynamic choices that affect performance on a per-frame basis are so great that developers rarely choose to do it.

Because of the indeterminism associated with technique 3 and the lack of high-end differentiation caused by technique 1, most developers choose to create scalability by creating bins of performance as described in technique 2. Often, the bins are a factor of two elements: the clock speed of the main processor and the API (application programming interface) support of the graphics subsystem. Unfortunately, these two factors don't encompass all possible system configurations equally well and tend to create situations where a system capable of higher performance, such as one with a high-end processor but integrated graphics, may be placed in a lower-performance bin. Conversely, a low-end processor coupled with high-end graphics hardware may have features enabled that the graphics hardware is capable of rendering, but to which data can't be supplied from the processor quickly enough for acceptable game-play.

Going on the assumption that a game will use the *binning* technique for scalability, let's now take a look at

The Scalability Problem

some processor trends that can help developers apply the technique.

PROCESSOR TRENDS

Each successive generation of processors introduced to the market adds enhancements that developers can leverage to improve their games. One recent example is the introduction of Hyper-Threading (HT) technology, which enables a single physical processor to appear to the operating system and applications as two processors. Pipeline stalls limit how much instruction-level parallelism can be extracted from a single instruction stream. The stalls occur whenever a cache miss happens or a branch misprediction takes place. HT technology enables two threads of execution to make better use of the processor's physical execution resources. For example, each of two threads can execute when the other is experiencing a pipeline stall, or one thread can use integer execution units while the other is using floating-point units.

HT technology is a stepping stone along the path toward processors with multiple physical cores. Introduced to desktop PCs in 2002, each successive generation of processors will increase the performance speed-up possible for two or more concurrent threads. For games to continue to push the envelope, they'll have to have multiple threads of execution working on various stages of the game loop.

Another way that processors are extended to enable higher performance in applications is through new instruction sets. Examples include MMX (Multimedia Extensions) technology, SSE (Streaming SIMD Extensions), and SSE2. These particular instructions provide SIMD (single-instruction, multiple data) operations that can work on integer and floating-point data types of various sizes. Using such instructions and arranging data appropriately to work with the instructions, games can do more with each clock cycle. Some of the scalability techniques I'll discuss here can be enabled more readily by taking advantage of new instruction sets. Future processors will continue to introduce instruction set extensions that can be used to enhance the performance of certain algorithms. For example, the Intel processor code-named Prescott, which will be introduced shortly, has a handful of SIMD instructions that can help game developers optimize techniques such as quaternion calculations.

A third trend that has recently emerged is the drive toward processors and accompanying components that consume less power to enhance mobile platforms. Although I won't investigate this any further in this article, developers working on multiplayer games will need to consider the impact that low-power-consumption platforms will have on their games. In particular, there are code optimizations that will reduce the power consumption of an application. It's not something that game developers currently think about, but it may be in the not-so-distant future.

With these trends in mind, let's look at a typical game loop and then examine some areas for applying scalability in games today.

TYPICAL GAME LOOP

Figure 3 shows some key elements of a simplified, typical

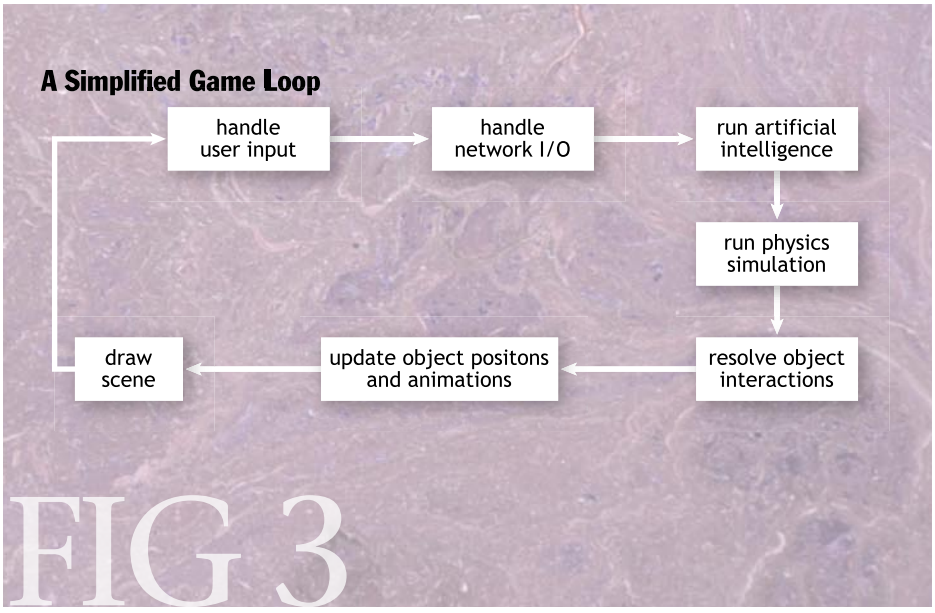


FIG 3

game loop. This example shows the operations happening in sequence, but some of the tasks could be performed in parallel. Some aspects of a game loop, such as scoring or handling sound, aren't shown here.

As games continue to evolve and improve in realism, all stages of the game loop shown here must be enhanced collectively. The weakest stage is always the one that stands out to a gamer. If the visuals are exceptional but the AI (artificial intelligence) doesn't have much "I," then the game won't really be fun to play. Or if the physics simulation is leaps above what's been done previously, but it's nearly impossible to manipulate with the keyboard and mouse, then gamers will quickly tire of it and move on to something more fun.

Applying scalability to the different stages of the game loop requires different techniques. For the graphics subsystem, which deals almost exclusively with the *DrawScene* stage, scalability can be addressed by enabling different rendering techniques based on the graphics features available. For the physics simulation, different techniques can be applied for solving different types of problems. Let's look at a few of these problems and how to address them with scalability.

TREES

The first 3D games were based almost exclusively in indoor settings because the amount of 3D geometry required to display a hallway is considerably less than what's required to display a forest scene, for example. As 3D accelerators improved, outdoor scenes appeared, and, now, massively multiplayer games are almost all based outdoors. Unfortunately, because the geometry for a tree is so complex (and large), games typically recycle a few of each variety. If you're running through a forest, you will often see the same exact tree over and over again in different locations. Additionally, trees in a game typically have several levels-of-detail (LOD)—one for trees in the distance, one for trees at an intermediate range, and one for close-up trees. A technique to increase the variety of trees will have to apply appropriately to the different LODs.

Generating a complete tree is possible procedurally using some parameters to indicate the type of tree to make. The algorithm described by Weber and Penn[2] has enough flexibility to create forests of trees. But the trees generated contain a lot of geometry and cannot currently be generated quickly enough to do so at runtime. What could be done, though, is to generate pieces of trees (e.g., trunks and branches) and then use a new algorithm to assemble them in different ways at runtime. The routine for doing the assembly could be put in another thread and its output could specify which trunk and branch pieces to use and transformation matrices to put them together.

Scalability for this technique could be introduced in several ways. First, the number of triangles used to create the trunks and branches could vary based on the available processor and graphics hardware performance. Second, the tree assembly routine could be placed in a low-priority thread. When the main thread needs a tree, it would pull one off the top of a queue that's getting filled by the low-priority thread. If there isn't one on the queue, then the main thread would just use the previous one again. In this way, a system with cycles to spare could create more tree variations.

In conjunction with creating varieties of trees, more realism could be incrementally added to games by animating the leaves and branches of the trees. Some games already do this to a degree. Usually, though, the motion is precalculated by an artist. Using the procedurally generated trees just described, developers could animate the parts, again based on available processor cycles. On low-end systems, the trees would be stationary. On mid-level systems, the main trunk could be animated to sway (using the technique described by Weber and Penn[2]). On high-end systems, even the branches and leaves could be animated using the techniques described by Peterson.[3]

Implementing the ideas that were just described should be straightforward, but we need to consider the following:

- First, procedurally creating content in a game is likely to raise the game designers' eyebrows. They want to make sure that their vision for how the game should look is not broken by some code randomly putting geometry together. To ensure that doesn't happen, it may be necessary to put extra constraints on how much variation the procedural generation can introduce, and some parts of the trees might still need to be created by the artists.
- Second, if the main game loop is designed to run as fast as possible, the low-priority thread creating the trees might not get enough CPU cycles to be of use. Providing a way to guarantee significant progress by the tree-creation thread and at the same time maintaining sufficient performance of the primary thread will require some tweaking.
- Finally, maintaining visual consistency between the procedurally generated trees at different LODs will require some experimentation; otherwise, visual "popping" artifacts between LODs could be severe.

The Scalability Problem

CLOTH

Another area of games that stands out as departing from reality is the characters' clothing. A few games, such as Hitman: Code Name 47 (IO Interactive, 2000), have implemented some simple techniques for trench coats and capes. But most characters in games still look as if their clothing is a permanent plastic attachment. Going from what we have today to full-scale simulation of clothing isn't going to happen overnight. It may be possible, however, to use some of the techniques in the research and motion-picture fields to get there incrementally.

Cloth simulation is the subject of lots of recent research. Baraff and Witkin[4] set the stage for using implicit integration to achieve more stable cloth simulation. More recently, Choi and Ko[5] solved an additional part of the instability problem, and Baraff, Witkin, and Kass[6] addressed the problem with cloth-cloth collisions introducing tangles. Applying this research to cloth simulation in games, though, is very difficult.

One solution that could be applied through scalability would be first to replace the clothing on characters in a game with actual geometry that separates the clothes from the underlying character model. A duplicate set of "invisible" geometry would then be used to simulate the movement of the clothing. The duplicate set would be tessellated to different degrees based on the performance of the system on which the game is running, and the actual geometry would be moved according to corresponding movement in the duplicated version. Low-end systems may not be able to achieve any movement of the actual clothing. High-end systems may tessellate to a significant degree and then simulate the cloth, treating certain vertices as "fixed" so that full cloth-cloth and cloth-object collisions wouldn't have to be handled initially. As processor performance increases over time, the simulation could become more detailed. Regardless of the degree of simulation, putting these calculations into a second thread will help on today's processors, as well as tomorrow's.

For nonclothing usages of cloth, processor performance has already reached a point that allows more physical simulation. Sails on ships, store awnings, or flags blowing in the wind are simple examples of *ambient* effects that can increase the realism of games, not affect game-play, and where the number of triangles simulated can be increased or decreased to address scalability.

Implementing cloth simulation is difficult—and none of the research mentioned here is a panacea. Any time physical simulation has control of portions of the game, the possibility exists that a configuration will arise that didn't come up in testing—and something will look horribly wrong. To prevent that, a simulation with cloth should make sure that the time step never exceeds a maximum value determined as certainly as possible through experimentation. The impact of this, however, is that excessive time spent in the cloth simulation could bog down the main game engine. As with any scalable solution, the implementation should try to detect and prevent or correct this situation as quickly as possible.

FLUIDS

Fluid dynamics is a broad field that will most readily be applied to games in the areas of smoke, fire, and water. Because of the complexity involved in solving the Navier-Stokes equations that describe the motion of fluids, most games haven't even attempted realistic fluid simulation. Recent research by Stam,[7,8] however, has introduced the possibility of solving simple fluid problems with visually believable results. The techniques can be applied in a scalable fashion (assuming the results are just ambient effects) by varying the grid size of the simulation. It's a matter of selecting the appropriate grid size based on available processor performance. Of course, some grid sizes may be too small to be useful, so a fallback to a different technique will be required.

For games that have action occurring on a boat, the water of the surrounding lake or ocean typically needs a form of animation to appear realistic. In the simplest form, sine waves are used to move the vertices of the water up and down. Combining several waves of different amplitudes and frequencies can introduce more variation. The repeating patterns are usually still evident, however.

One game in development is using a more advanced statistical method, described by Tessendorf,[9] to simulate realistic-looking ocean water. Low-end systems use the sine wave technique, and high-end systems use the Tessendorf technique. By combining the better simulation with fancier rendering on high-end graphics hardware, the in-game results are quite impressive.

The application of scalability to fluid simulation has its share of challenges as well. Like any physical simulation, if the problem size changes (e.g., different grid sizes) or if the step size between simulation times varies, the end results will be different. So the initial uses of fluid simulation will have to be either devoted entirely to ambient game effects or simple enough to run on the minimum system specs without any scalability to higher systems. Most likely, the introduction of fluid simulation to actual game-play, not just ambient effects, will require the combination of both, so the visual quality may be scalable but the simulation quality will be fixed.

THE SCALABILITY CHALLENGE

Scalability is a challenge facing game developers that they can't just ignore. It's not the only challenge they'll face, but it's one that can significantly impact the quality of their games and the differentiation of one game over the competition. Fortunately, scalability can be addressed by taking advantage of new processor features and leveraging work being done in the research community.

We've examined the evolution of consumer PC processors and seen that threading will be essential to get the best performance possible from future platforms. Several stages of the game loop will need to be executed in their own threads to benefit from the performance available. In addition, other processor extensions such as new instruction sets will provide a means for developers to introduce new techniques in a scalable fashion. Using these processor enhancements, games will continue to mature and come closer to the photorealistic worlds we see in computer graphics-generated motion pictures.

As one step to getting there, we've taken a look at a few areas in games that developers can improve by using scalable techniques based on processor features and extensions. All three of the areas examined—trees, cloth, and fluids—can benefit from both threading and SIMD instructions. These three are only a few aspects of games that can scale using the processor. Other areas can be explored that apply scalability to the graphics hardware in conjunction with the processor. By applying scalability to elements of the game loop, developers can have the freedom to innovate while creating games accessible to the huge installed base of consumer PCs. Q

REFERENCES

1. Dictionary.com: see http://dictionary.reference.com.
2. Weber, J., and Penn, J. Creation and rendering of realistic trees. *Proceedings of the ACM SIGGRAPH* (1995), 119–128.
3. Peterson, S. Animating trees. Silicon Valley ACM SIGGRAPH (2001); http://silicon-valley.siggraph.org/MeetingNotes/shrek/trees.pdf.
4. Baraff, D., and Witkin, A. Large steps in cloth simulation. *Proceedings of the ACM SIGGRAPH* (July 1998), 43–54.
5. Choi, K.-J., and Ko, H.-S. Stable but responsive cloth. *Proceedings of the ACM SIGGRAPH* (July 2002), 604–611.
6. Baraff, D., Witkin, A., and Kass, M. Untangling cloth. *Proceedings of the ACM SIGGRAPH* 22, 3 (July 2003), 862–870.
7. Stam, J. Stable fluids. *Proceedings of the ACM SIGGRAPH* (1999), 121–128.
8. Stam, J. Real-time fluid dynamics for games. *Proceedings of the Game Developers Conference* (March 2003).
9. Tessendorf, J. Simulating ocean water. ACM SIGGRAPH Course Notes (2001).

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

DEAN MACRI is a staff technical marketing engineer in the Software and Solutions Group at Intel. He works primarily with game developers to help them optimize their games for present and future processor architectures and take advantage of the processing power available to enable new features. He has a B.A. in mathematics and computer science with a minor in physics from St. Vincent College, Pennsylvania, and an M.S. in computer science from the University of Pennsylvania. After completing his master's degree in 1992, he spent five years developing highly optimized C, C++, and assembly language routines for a 2D graphics animation company. He joined Intel in 1998 to pursue his interests in 3D computer graphics.