Michael Lewis and Jeffrey Jacobson

GAME ENGINES IN SCIENTIFIC RESEARCH

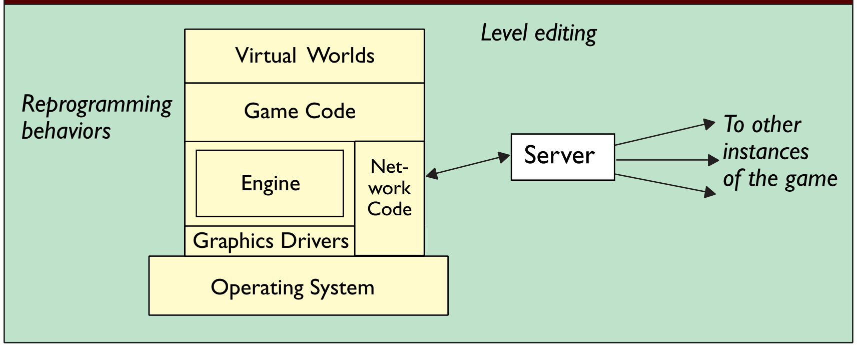
SERIOUS COMPUTATIONAL RESULTS ARE DERIVED FROM COMPUTER-BASED GAMES.

Six-figure workstations and custom software are not producing the best graphics or virtual reality simulations anymore. Today, the only way to have the fastest, most realistic simulations and sophisticated graphics is to trade down from the expensive gear to standard PCs running game software. Virtual reality (VR), augmented reality (AR), and high-fidelity physical simulation have long posed too high a barrier to entry for any but the most generously funded researchers. Significant advances in computer graphics in these areas have traditionally depended on expensive, specialized graphics hardware running on scientific workstations. High-fidelity simulation/graphics software has also remained an expensive niche market largely the province of the military and high-end VR labs seeking to drive costly peripherals such as Caves, datagloves, and head-mounted displays (HMDs). In the past two years, however, the situation has changed remarkably. Now the mass market for computer games, grown even larger than the movie industry, has expropriated the best in computer hardware and software for itself. The most sophisticated rendering pipelines are now found not on specialized scientific machines but on PC video cards costing less than $500. The most sophisticated, responsive interactive simulations are now found in the engines built to power games.

Despite the stigma of violence and gore associated with first-person games, there has been a long history of unpublicized cooperation between computer scientists and the game industry [1, 2]. Games have provided the first and sometimes the only market for advanced graphics techniques, demonstrating the extent to which realism could be conjured up even from relatively weak graphics environments using artistry and texture maps.

The cost of developing ever more realistic simulations has grown so huge that even game developers can no longer rely on recouping their entire investment from a single game. This has led to the emergence of game engines—modular simulation code—written for a specific game but general enough to be used for a family of similar games. This separability of function from content is what now allows game code to be repurposed for scientific research.

Modular game engine structure.



Early computer games consisted of little more than the event-loop, state tables, and graphic routines needed by simple 2D games such as Space Invaders, Galaxians, or Raptor. The 1993 release of Doom by id Software ushered in a new era of game design and play. Although not the first game to offer an immersive first-person perspective, Doom was the first to do so successfully. It achieved a sense of realism even on the 80486-based PCs of the day by assiduous use of texture maps, 2-1/2 D/4 DF animation, and other programming tricks. This tradition of coaxing more realism than possible by brute force continues to distinguish gaming simulations from their scientific counterparts. Two other features significant to scientific users introduced by Doom were multiplayer play over a network and user/third-party programmability.

Because Doom and its descendants have been designed from inception for network play, their architectures have more in common with large-scale distributed military simulations such as SimNet or ModSAF than VR systems such as World Toolkit or languages such as VRML. By elevating the problems of updating and synchronization to a primary concern, game engines provide superior platforms for rendering multiple views and coordinating real and simulated scenes as well as supporting multiuser interaction. Despite its initial success, Doom's peer-to-peer architecture with parallel games updating in lockstep proved cumbersome and has been replaced by client/server schemes in later games. The Doom engine offered only the minimal degree of programmability by providing a generic game executable to which the user could add data in a prescribed format. The new levels had changed layouts but continued to look and play just like Doom. Later games have significantly expanded programmability.

Quake [1] was a lessons-learned follow-on from id Software. It was truly 3D with environment and actors made from texture-mapped polygons, although a clever split of rendering pipelines between entities and environment was needed to attain sufficient speed. In place of peer-to-peer networking it provided a client/server architecture with the game state maintained on the server while the compute-intensive 3D engine ran on the client. Quake also extended its programmability to provide the first game-independent game engine providing both a level editor for changing layouts and QuakeC, a byte-compiled scripting language, for changing behavior in the simulation.

Quake II extended the performance and fidelity of Quake, adding support for hardware-accelerated graphics that were just beginning to appear on PC video cards. With the release of Quake III Arena and Epic Games Unreal Tournament in 1999, game engines reached their present mature state. Both games are strictly multiplayer with single-user play simulated through play against a synthetic opponent. They each provide extensive support for hardware acceleration, and support user modifications through level editors and scripting languages. As with Quake II, Quake III Arena makes the C language game source code available, keeping only the graphics engine code proprietary. Unreal Tournament is set up in much the same way, except that its open source game code is written in UnrealScript, a bytecode-compiled scripting language similar to Java. Although there are more than 600 commercial game engines including the high-concept graphics of Everquest's LithTech engine, Quake III Arena and Unreal Tournament provide the most developed, flexible, and usable engines for research purposes.

What are Game Engines?

In today's modularly constructed games, the game's engine refers to that collection of modules of simulation code that do not directly specify the game's behavior (game logic) or game's environment (level data). The engine includes modules handling input, output (3D rendering, 2D drawing, sound), and generic physics/dynamics for game worlds. The figure here shows this overall structure.

At the top level are the virtual worlds or scenarios with which the players interact. They come in a wide range of appearances and rules of interaction, even different types of simulated physics. Many are authored by fans of the game, who primarily use an advanced program development environment that comes free with the game itself. Often they include components of existing worlds or are built using knowledge the authors freely exchange via the Web.

The level below is the game code, which handles most of the basic mechanics of game itself, like simple physics, display parameters, networking, and the base- or atomic-level actions for animations such as autonomous agents or behaviors in the player's own avatar. Altering the game at this level requires a more detailed knowledge of its internals and is accomplished using a game-specific scripting language.

The rendering engine is the crown jewel of the game. It incorporates all of the complicated code needed to efficiently identify and render the player's view from a complex 3D model of the environment. The rendering engine is a proprietary black box and not open to any kind of user modification.

The networking code supports a robust, built-in networking protocol similar to DIS/HLA, which allows several users in remote locations to explore and interact in the same virtual environment (VE). One computer acts as the server or host for the environment, while the others support the individual users. Networking code is so efficient and well-developed in Unreal Tournament and Quake III that players on a low-latency local-area network can expect to get synchronization that exceeds frame rate of their displays.

The graphics drivers translate generic requests from the rendering engine to the underlying graphics library, using APIs such as DirectX, OpenGL, and others. Because the drivers are open source they can easily be modified to accommodate new display types such as Caves or HMDs.

Finally, the server is a separate process, usually on a different machine, which maintains information on whichever virtual world it is supporting at the time. It communicates with the game clients used by the players to maintain global information about shared environments, player interactions (such as damage) and synchronization information. For example, each player has a different window into their shared VE. The scenes presented to the players must be consistent so if one player changes the state of some object, for example, that object must exist for all the other players and be in the same state.

COAXING MORE REALISM THAN POSSIBLE BY BRUTE FORCE CONTINUES TO DISTINGUISH GAMING SIMULATIONS FROM THEIR SCIENTIFIC COUNTERPARTS.

Scientific Use of Game Engines

While control is the key to science, game engines are by design opaque. Speed-up strategies such as swapping texture maps as distance increases produce the illusion of reality but can leave the experimenter without cleanly manipulatable parameters. Precise behavior of control systems, as found in Microsoft Flight Simulator, or response of materials to stress are not usually part of the worlds simulated by game engines. If, on the other hand, ecological validity is crucial to the research, photorealistic game engine graphics and animations will work far better than well-controlled but more primitive scenes constructed in a principled way. If research requires only faithful maintenance of player, object, and terrain locations then game engine fidelity is completely adequate.

While precise multimodal VE simulations such as those required for remote surgery will continue to demand custom solutions, for most VE applications a game engine can add value by performing 3D bookkeeping, providing networking and synchronization, or driving high-fidelity hardware-accelerated graphics.

The research projects described in the short articles in this section illustrate the variety of ways in which game engine capabilities can be repurposed for research. Gal Kaminka's GameBots, Bylund and Espinoza's QuakeSim, and John Laird's artificial intelligence systems all de-emphasize game graphics in favor of game engines' capabilities as general-purpose 3D simulations for keeping track of players, locations, and objects in complex 3D worlds.

Research in robotic coordination requires common environments for simulating sensor data and motoric responses. The popular Robocup [3] soccer simulation does not provide the degree of complexity or challenge needed to prepare teams of robots for real-life tasks. By defining socket-based interfaces to connect simulated robots to an Unreal Tournament server, Kaminka opened up the entire library of Unreal Tournament worlds to robotic play, test, and evaluation. QuakeSim takes advantage of this same access to simulated sensor data but uses it to test software concepts involving context-aware services. John Laird's game-based research is closest in spirit to the games themselves. Working in both Quake and Unreal Tournament, he and his team at the University of Michigan are seeking to use the

Soar learning architecture to develop and test human-level artificial intelligence through controlling synthetic characters within the games.

The other two research projects exploit both the modeling and graphics capabilities of the game engines. In the augmented reality game ARQuake, game levels are written to match actual physical locations and players wearing see-through HMDs encounter and do battle with monsters superimposed over real scenery. CaveUT takes advantage of both the graphical and networking capabilities of the Unreal engine by using multiple players' viewpoints to construct a panoramic Cave-like display. There are probably as many potential applications for game engines as there are research problems requiring medium-fidelity 3D simulation or high-fidelity interactive graphics. Our hope is to raise awareness of the high-power/low-cost alternative game engines can offer.

Choosing an Engine

A researcher's choice between the Quake and Unreal engines is similar to that between Coke and Pepsi. Both are more than adequate for their intended purpose yet each has its avid fans and detractors. An examination of contributors here indicates our researchers are evenly divided with Kaminka and Jacobson lining up with the Unreal engine, Bylund and Espinoza choosing Quake III, and John Laird and colleagues working with both. For research purposes it is hard to imagine an application for which one would be suited and the other not. However, because a choice must be made there are some differences that should be considered.

As the older game, the Quake III engine has developed a large collection of customized levels and modifications to play. Quake III also has a large cadre of users and user sites to use as resources for programming questions and assistance. It is not unlikely that someone in your organization may have Quake programming experience. The Quake engine has been optimized to the point where it is the fastest-running game engine and has what many believe the best overall polygonal architecture.

As the newer engine, Unreal benefits from current trends in programming and technology—see the sidebar "The New Cards." Unlike Quake, which has descended from a succession of C language implementations, the Unreal engine is strictly object-oriented. This pays big dividends through mutators and other programming constructs that allow a programmer to make robust changes in game behavior without requiring detailed knowledge of the involved code. The Java-like scripting language, Unreal Script, is easy to use and well documented as is the well-designed UnrealEd development environment. Although the Quake engine offers true curves and direct access to graphics functions such as deformation shading, the Unreal

The New Cards

Michael Lewis

The most significant recent development in gaming and graphics-intensive computing has been the availability of supercomputer-level graphics on commodity-priced graphics processing units (GPUs). Leapfrogging earlier limits, Nvidia's Quadro2 Pro GPU delivers 31 million polygons per second at a fill rate of over one billion texture-mapped pixels per second, which makes it the world's fastest GPU by some measures. This speed comes despite a raft of new hardware operations including multiple shadings per pixel, multitexturing, texture and lighting, bump mapping and other compute-intensive operations not found until recently in animated PC graphics. The new Nvidia chips are so effective they have found outside applications as the graphical heart of Microsoft's new Xbox game player and at the core of the multifunction avionics display for the F-22 fighter aircraft.

The really big graphics applications like Caves, Power Walls (high-resolution wall-size displays), and open-canopy flight simulators, however, rely not on a single GPU but on systems built from many graphics pipelines working together in parallel. The top-of-the-line Silicon Graphics Onyx 3800, for example, manages up to 16 simultaneous graphics pipelines for an upper limit of 210 million polygons per second and a fill rate of 12 billion pixels per second. Achieving coherent graphics on this scale requires elaborate coordination and management to rebalance loads among pipelines at the crucial transition in processing between object parallelism and image parallelism. The SGI InfiniteReality architecture designs graphics pipelines to accept broadcast primitives at just this point. General-purpose GPUs designed for PC graphics and embedded applications do not have the luxury of this "sort-middle architecture" because they are designed to work as self-contained standalone units.

Making the transition from competing with desktop workstations to taking on refrigerator-sized, rack-mounted capital investments requires solving the problem of parallel processing using commodity GPUs.

engine encapsulates a broader range of sophisticated graphics including bump maps and other tweaks recently added to graphics hardware. In the balance, the Unreal engine is slightly slower, has more sophisticated graphics, and is probably an easier environment for the inexperienced game programmer.

Getting Started

Now that you have chosen your engine all you need to get started is a copy of the base game. Depending on where you shop the game may cost anywhere from $20–$50. You will need a license for each machine you use so, for instance, a QuakeSim simulation would take a single license, a GameBot installation with two teams would require two licenses, while a five-screen CaveUT installation would require six licenses. A crucial feature of the game engines is that almost all code, except the proprietary graphics engine, is open source. Under its GNU license, this code may be freely distributed, copied, and modified. Several of the systems discussed here are already available online. CaveUT can be downloaded from www2.sis.pitt.edu/~jacobson/ut/-CaveUT.html and GameBot code, documentation, discussion, and even opponents, are located at www.planetunreal.com/gamebots/. While not game-based, the WireGL code for building PC-based Power Walls is available at graphics.stanford.edu/software/wiregl/. The primary Quake site is www.idsoftware.com, while www.averstar.com/~bowditch/QUAKE2/links.html contains links to many secondary Quake sites. The manufacturer's site for Unreal Tournament is www.epicgames. com; a "getting started" FAQ is available at www.unreality.org/Cleaned/FAQ/FaqMain.htm. A quick search will turn up a multitude of sites with information, tools, and programming tips for either of these engines. While neither id Software nor Epic Games is in the business of supporting research, their user communities can provide active sources of help and information for game-using researchers. **C**

References

1. Abrash, M. Quake's game engine: The big picture. *Dr. Dobb's Journal* (Spring, 1997).
2. Bishop, L., Eberly, D., Whitted, T., Finch, M., and Shantz, M. Designing a PC game engine. *IEEE Computer Graphics and Applications* (1998), 46–53.
3. Kitano, H., Tambe, M., Stone, P., Veloso, M., Coradeschi, S., Osawa, E., Matsubara, H., Noda, I., and Asada, M. The RoboCup synthetic agent challenge '97. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, Nagoya, Japan, (1997).

Michael Lewis (ml@sis.pitt.edu) is an associate professor in the Department of Information Science and Telecommunications at the University of Pittsburgh.
Jeffrey Jacobson (jacobson@sis.pitt.edu) is a Ph.D. candidate in the Department of Information Science and Telecommunications at the University of Pittsburgh.

Because the graphics pipelines themselves are inaccessible and closely tied to other parts of the PC architecture, the most convenient approach lies in clustering PCs to achieve "sort-first" machine-wise parallelization. While this architecture is limited by network bandwidth and is less efficient than the "sort-middle" approach of high-end graphics vendors, the huge cost savings and the speed advantage of commodity GPUs makes it hard to resist.

We are aware of two such solutions. WireGL (now Chromium),[1] developed at Stanford University, is a general-purpose system for scalable interactive rendering on a cluster of workstations. WireGL replaces the OpenGL libraries, which allows OpenGL calls to be parallelized and rendered through the cluster. Reported performance of 70 million triangles per second for a 32-PC configuration (16 compute and 16 rendering nodes) compares well with the achievable performance estimate of 29 million polygons per second for a 16-pipeline SGI InfiniteReality configuration. Using PCs and data projectors found in most labs, WireGL can allow researchers to experiment with previously rare and expensive large-scale high-resolution displays.

The same hardware can be used to run CaveUT (described in this issue by Jacobson and Hwang) to create an immersive panoramic Cave-like display. CaveUT follows the same cluster rendering approach as WireGL but takes advantage of the Unreal engine's networking and graphics synchronization capabilities to do the processing. Because game players' views must be computed independently by clients, the opportunity for load balancing is lost. Nevertheless, given lightweight communication protocols, the power of the new commodity GPUs and gigahertz-plus GPUs, display refresh rates have become the limiting factor. **C**

[1]Humphreys, G., Eldridge, M., Buck, I., Stoll, G., Everett, M., and Hanrahan, P. WireGL: A scalable graphics system for clusters. In *Proceedings of the 2001 Conference on Computer Graphics* (Los Angeles, CA, 2001), 129–140.

Michael Lewis (ml@sis.pitt.edu) is an associate professor in the Department of Information Science and Telecommunications at the University of Pittsburgh.