FOCUS Game Development

Game Development Harder Than You Think

JONATHAN BLOW
GAME DEVELOPMENT CONSULTANT

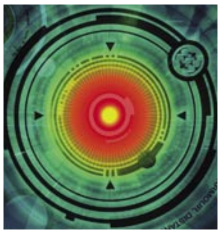
The hardest part of making a game has always been the engineering. In times past, game engineering was mainly about low-level optimization—writing code that would run quickly on the target computer, leveraging clever little tricks whenever possible.

But in the past ten years, games have ballooned in complexity. Now the primary technical challenge is simply getting the code to work to produce an end result that bears some semblance to the desired functionality. To the extent that we optimize, we are usually concerned with high-level algorithmic choices. There's such a wide variety of algorithms to know about, so much experience required to implement them in a useful way, and so much work overall that just needs to be done, that we have a perpetual shortage of qualified people in the industry.

Ten or twenty years ago it was all fun and games. Now it's blood, sweat, and code.

Making a game today is a very different experience than it was even in 1994. Certainly, it's more difficult. In order to talk about specifics, I've classified the difficulties into two categories: problems due to overall project size and complexity and problems due to highly domain-specific requirements. Though this will help me introduce the situation in stages, the distinction between the two

Game Development Harder Than You Think



categories is a bit artificial; we will come full-circle at the end, seeing that there are fundamental domain-specific reasons (problems due to highly domain-specific requirements) why we should expect that games are among the most complicated kinds of software we should expect to see (problems due to overall project size), and why we should not expect this to change for the foreseeable future.

PROJECT SIZE AND COMPLEXITY

To illustrate the growth of games over the past decade, I've chosen four examples of games and drawn graphs of them. Each node in a graph represents a major area of functionality, and the arcs represent knowledge couplings between modules. Two nodes with an arc between them need to communicate heavily, so design decisions made in one node will propagate through its neighbors.

Figure 1 depicts a 2D game from the early 1990s, perhaps a side-scrolling action game for a home console, like Super Metroid. Other genres of game would have slightly different diagrams, for example, a turn-based strategy game like Civilization would gain a node for computer-opponent AI (artificial intelligence), but would lose the node for fast graphics. Certainly Super Metroid itself also has computer opponents, but their behavior is simple enough that it doesn't warrant an extra node; instead the enemy control code is lumped in with "main/misc."

By 1996, 3D games had become a large portion of the game industry's output. Figure 2 shows an early 3D game, for example, Mechwarrior 2. Contrast this with figure 3, a modern single-player game.

The largest endeavor we currently attempt is the 3D massively multiplayer game (MMG), illustrated in figure 4. Everquest is the canonical first example of a 3D MMG, though a more up-to-date example would be The Matrix Online (expected release in 2004).

Contrasting figure 4 to figure 1 should give you a general sense of how the situation has changed. The arcs in these figures assume that code has been ideally factored, but since this is never the case, real-life situations will be more tangled. Keep in mind that each node in these graphs is itself a complex system of many algorithms working together, and that each of these nodes represents somewhere between six thousand and 40 thousand lines of source code.

There's another category of game, the non-massively multiplayer client/server game, which tends to house a smaller number of players at once (perhaps 50) and does not maintain a persistent world. The diagram for one of those would be somewhere between figure 3 and figure 4.

A 2D Game Circa 1994

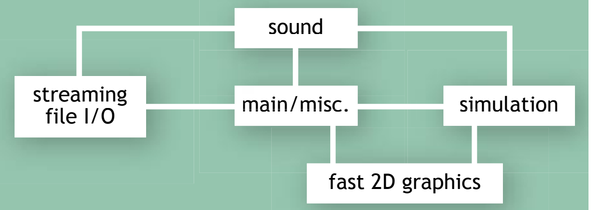


FIG 1

A 3D Game Circa 1996

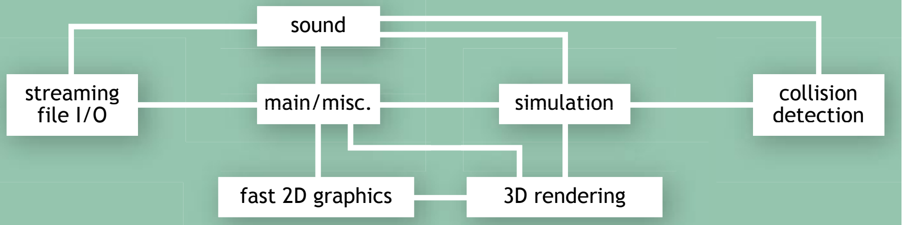


FIG 2

Tools. To tackle such complexity, it helps to have excellent development tools. Sadly, we do not have excellent development tools.

For programming on PCs, we use a compiler development environment like Microsoft Visual Studio, which is basically a wrapper around their C++ compiler; most games now are written primarily in C++. Clearly, we are not the target market Microsoft has in mind. Visual Studio seems to be aimed heavily at developers of Visual Basic and C# applications, and to the extent it caters to C++, it's meant for applications that make heavy use of COM objects and create many windows with variegated UI elements. We do very little of that stuff in modern games. We would much rather have that manpower spent to make the system compile programs quickly, or generate efficient code, or produce reasonable error messages for code that uses C++ templates. Even so, Visual C++ is the best compiler we have on PCs—with no competitive alternatives—so we're just sort of along for the ride.

On consoles, the console maker as well as one or two third-party companies will provide some development tools (compiler, debugger, profiler, etc.). Console life cycles, however, are about five years long, and there isn't much motivation for the tool-maker to improve their products toward the end of that cycle. Typically, a console developer will be using an environment with only one to four years of maturity—not an enviable situation.

To build game content like 3D meshes and animations, we use programs like Maya or 3D Studio MAX. However, these programs were originally created for people who make non-realtime animations (like the graphics rendering for feature films), so they present a poor fit. Lately, as games have become a bigger business, the makers of these tools have begun to pay more attention to us, to the point that they put "games" at the top of the list of their products' relevance. But these tools are so deeply rooted in the "wrong area," and so big and slow to change, that they still represent something very different from what we really need. For example, most game studios would benefit from the ability to build large continuous 3D world meshes, with multiple artists working on the same mesh at once—or methods of editing triangular meshes to ensure that cracks and holes do not appear. This would be much more interesting to us than much of the functionality these vendors develop and tout, such as sophisticated cloth simulation (useful to us only for pre-rendered cinematics, which are becoming increasingly rare in games).

Thus we need to augment these content packages with our own plugins and post-processing tools, which will in general be poorly integrated and feature-starved, and may present robustness problems. Sometimes, for building the geometry of the world, we just write our own domain-specific editors from scratch (Worldcraft and UnrealEd are examples of this).

Historically, the situation with regard to asset management tools has also been poor. A modern game studio needs a fast and robust system for networked revision control of source code, 3D models, animations, sound effects, and all the other various data files involved in a game. Lately, some companies have risen to provide asset control specifically for game projects. These tools are still far from ideal, but we have reason to hope that they will improve.

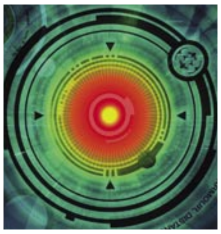
Workflow. We also have a lot of workflow problems that are not so directly tied to specific tool software. On the programming side, our compile/edit/debug cycles are usually far too long. Many games take half an hour or longer to compile when starting from scratch, or when a major C++ header file is changed. Even smaller changes, causing a minimal amount of recompilation and relinking, can take as long as two minutes. In general, C++ seems to encourage long build times. Once the build time has grown too long, a team may end up putting a significant amount of work into refactoring their source code to make it build more quickly. Often this happens too late, as the spaghetti of file dependencies has become so severe that fully refactoring it would be akin to restructuring the project from scratch. In fact, the best way to avoid long build times is to architect the entire code base to minimize dependencies (sometimes giving up runtime efficiency in the process!). This does not happen too often because many studios do not take these workflow issues as seriously as they ought to as the effect of the problem is somewhat intangible, and there are always so many clear and present issues to deal with—or they don't have sufficient discipline to deal with such a subtle issue over periods of time measured in years.

Another way to attack the build problem is to use a third-party tool to distribute compiles across many machines (one such product is Incredibuild). These tools can help significantly but they are not cure-all solutions.

Once the game is compiled, we must run it and test our changes. However, startup times can be very long, since games often need to load large amounts of data. Startup time can typically be three minutes for a debug build with large data files for which load-time optimization has not been done. Add this to the compile-and-link time, and you can easily have a five-minute delay between making the smallest possible code change and seeing the new version of the game running. Testing the actual change will take longer as the programmer needs to set up the proper conditions within the game world to exercise that code path.

Visual C++ provides an "edit and continue" feature wherein one may splice code changes into a running

Game Development
Harder Than You Think



program and avoid these delays. However, this feature doesn't work reliably enough to eliminate the problem (though when it does work, it is very welcome). This feature is not usually present in the compiler environments for console systems. Another way to avoid this turnaround time is to write a significant amount of your code in a higher-level extension language that can be dynamically reloaded by the game engine without restarting. (For more on this, see Andrew M. Phelps and David M. Parks' "Fun and Games with Multi-Language Development" on page 46 of this issue.)

There's an analogous issue for the content development parts of the team with regard to how long it takes them to see the effect of changing a texture or model. Fortunately this problem is easier to solve; as loading these assets is handled entirely by our game engines, we are empowered to fix the situation. Currently, some game engines written by experienced developers provide automatic reload of content resources at runtime, which is becoming a more widespread trend.

Jamie Fristrom[1,2] has recently written some columns for Gamasutra[3] describing these workflow issues from a manager's point of view.

Multiplatform Development. Many games are developed to run on multiple systems. During development we often have to build the game for all build types (Debug, Release) for all target platforms (PC, Playstation 2, Xbox) before committing our changes to source control. Whenever this is not done, Murphy's Law nearly guarantees that small differences in header files or system behavior will cause a compile-time or runtime error, disrupting the work of the rest of the programming team—a bad situation. So before a programmer can check in a batch of changes, they may need to perform between two and five full recompiles (which, as we mentioned earlier, sometimes take half an hour each!). The programmer can easily be waiting for hours, so there's a strong motivation to check in code changes as infrequently as possible. But they can't wait too long, or the code will drift too far out of sync from the official version, causing headaches when it comes time to merge.

As in large business projects, bigger game teams tend to have a "build master," a person whose job is to watch over the build, ensuring that disruptions are remedied as quickly as possible. Sometimes pleasing the build master can be a difficult task. Yet despite the presence of a build master, builds still seem to be broken too often.

The result of all this is that, too often, a game programmer can't just sit down and get work done; there are significant barriers to push through.

Third-Party Components. There are many nodes in figures 3 and 4 (see my discussion of highly domain-specific requirements in this article below). We ought to be able to leverage third-party products for some of those boxes in order to reduce our workload. Licensable third-party modules exist for some of those nodes. Depending on the nature of the task, however, some of these products have been more successful than others at meeting industry needs. Available products cover these areas: *audio, low-level* (products have been very successful); *rendering, low-level* (very successful); *rendering, scene management* (mixed success); *collision detection and physics* (only somewhat successful, but it's very hard to write these systems on your own, so there's a significant win for third-party tools here); *networking, low-level* (slightly successful, could be better but nobody has come to market with the right products); *skeletal animation and morph targets* (very successful); *persistent object storage* (mixed success); and *scripting languages* (mixed success). Most notably, no useful products for AI functionality exist, though there have been a few misguided attempts.

Because games are complicated and require deep technical knowledge (again, see my discussion of highly domain-specific requirements below.), it can be difficult just to use these third-party components; often the programmer must have a lot of experience in the problem domain in order to understand how to interface with the product successfully. Even if this is the case, the programmer still may face great difficulties in integrating the third-party module with the rest of the game.

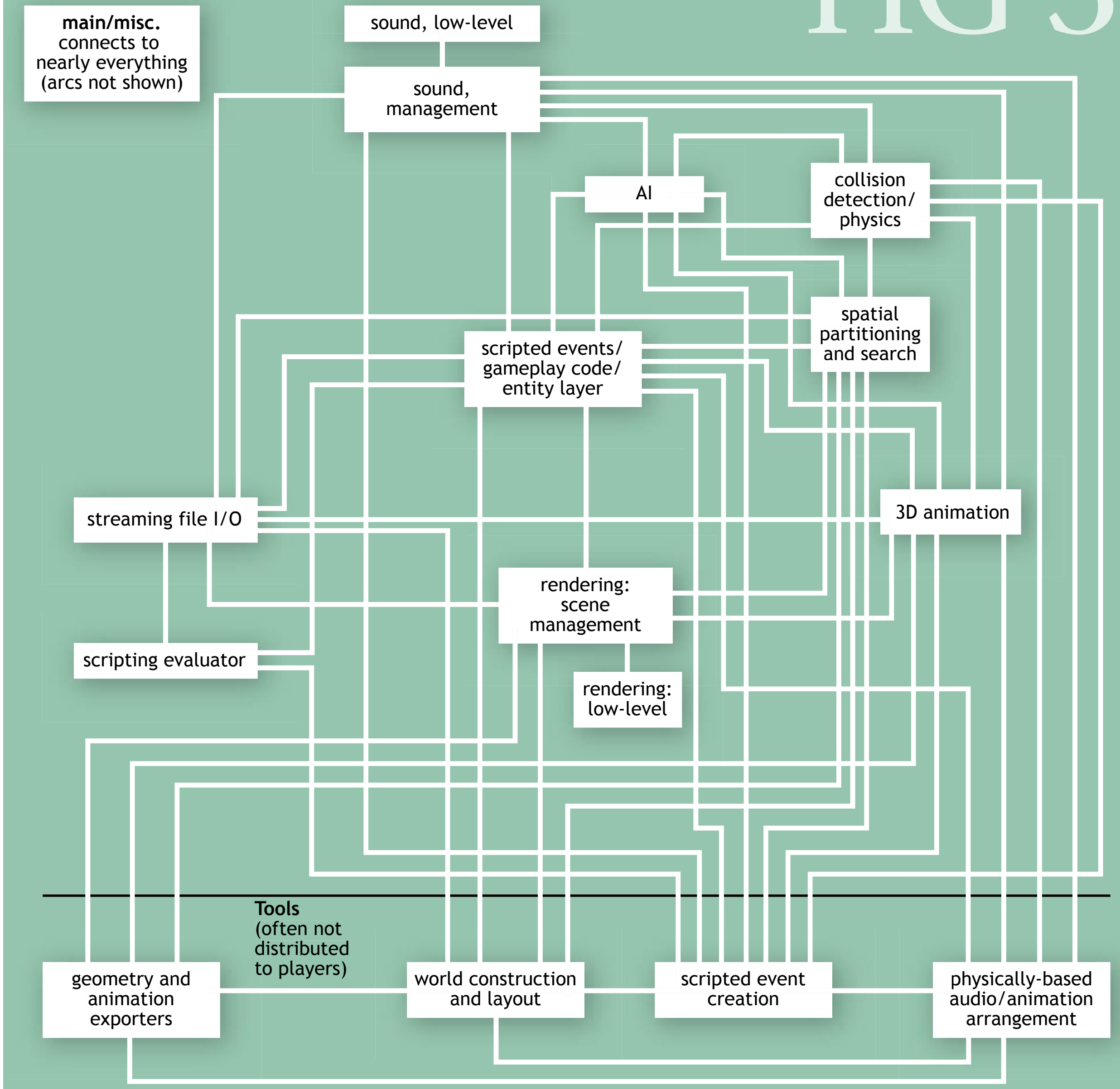
Most of these modules were themselves technically challenging to create, so they tend to be less than perfect. Often the API (application program interface) is difficult to deal with because it embodies some conceptual model that is a poor fit for the way your game needs to work. Thick glue layers are usually necessary between the main game code and the third-party API. Application program interfaces for rendering or physics often want data organized in very specific ways, a situation that propagates through the rest of the program and imposes difficult constraints (because a lot of data needs to be passed back and forth, we can't just convert the data between formats at function call time as that would be too slow). And since games are so CPU-intensive, it will often happen that the third-party component presents a significant performance bottleneck for some input scenarios—and the programmer must fix these situations or work around them.

Often when third-party code fails, it's because the problem it solves is insufficiently large; for the amount of work the development team spends to make the code succeed, they might as well have written the module from scratch—something you certainly don't want to find out *after* failing with the licensed code. The decision to license third-party code should always be preceded by a careful cost/benefit analysis as there's no guarantee that the product will actually hasten your development.

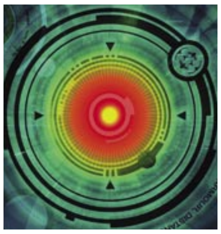
Full-Figure Option. Instead of licensing components, we can license an entire game engine from a company that has successfully built a solid one (see my discussion of highly domain-specific requirements in this article).

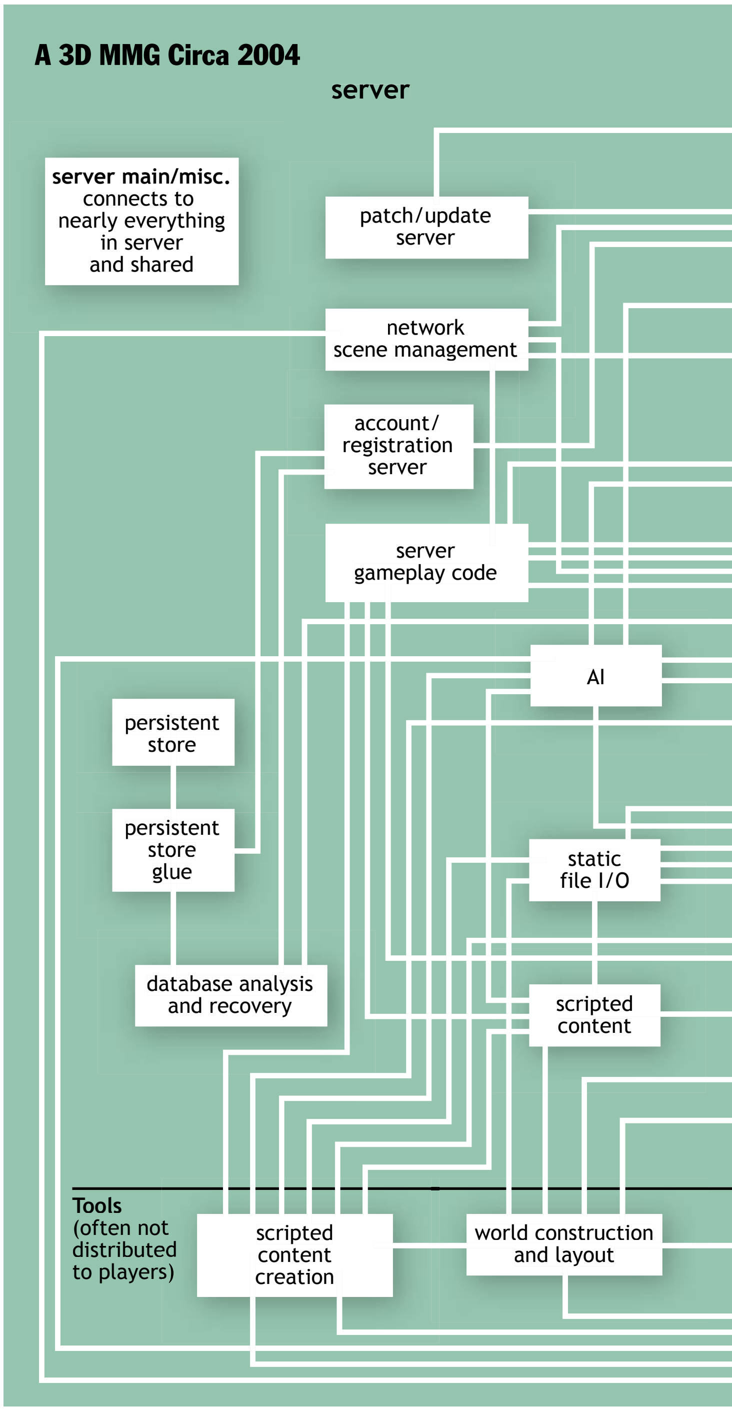
FIG 3

A 3D Single-Player Game Circa 2004


main/misc. connects to nearly everything (arcs not shown)
sound, low-level
sound, management
AI
collision detection/ physics
spatial partitioning and search
scripted events/ gameplay code/ entity layer
streaming file I/O
3D animation
rendering: scene management
rendering: low-level
scripting evaluator
Tools (often not distributed to players)
geometry and animation exporters
world construction and layout
scripted event creation
physically-based audio/animation arrangement

Game Development Harder Than You Think




A 3D MMG Circa 2004
server
server main/misc. connects to nearly everything in server and shared
patch/update server
network scene management
account/ registration server
server gameplay code
AI
persistent store
persistent store glue
static file I/O
database analysis and recovery
scripted content
Tools (often not distributed to players)
scripted content creation
world construction and layout

It's more difficult to build a licensable engine than it is just to make a game, so there are not many of these to reasonably choose from. Some recent examples are the Quake 3 engine and the Unreal engine. The cost of such a license tends to be high, perhaps $300 thousand to $600 thousand per retail SKU (stock keeping unit). If you're trying to make a game that is not doing anything new technologically, such a license can be a safe decision. But if you're trying to be technologically expansive, you will probably run into the poor-fit problems mentioned earlier, but on a larger scale this time—you might find yourself spending $500 thousand for code that you end up largely rewriting, disabling, or working around. (Even so, it's possible for this to be money well spent because having the engine gives you a kick-start that's sometimes better than starting with nothing.)

Both of the aforementioned engines come from the genre of *first-person shooters* (FPSs), which is the area where the finest-honed game technology has flourished. For games that are very different from an FPS, you may have a difficult time finding a serviceable engine. There are no market-proven engines for MMGs.

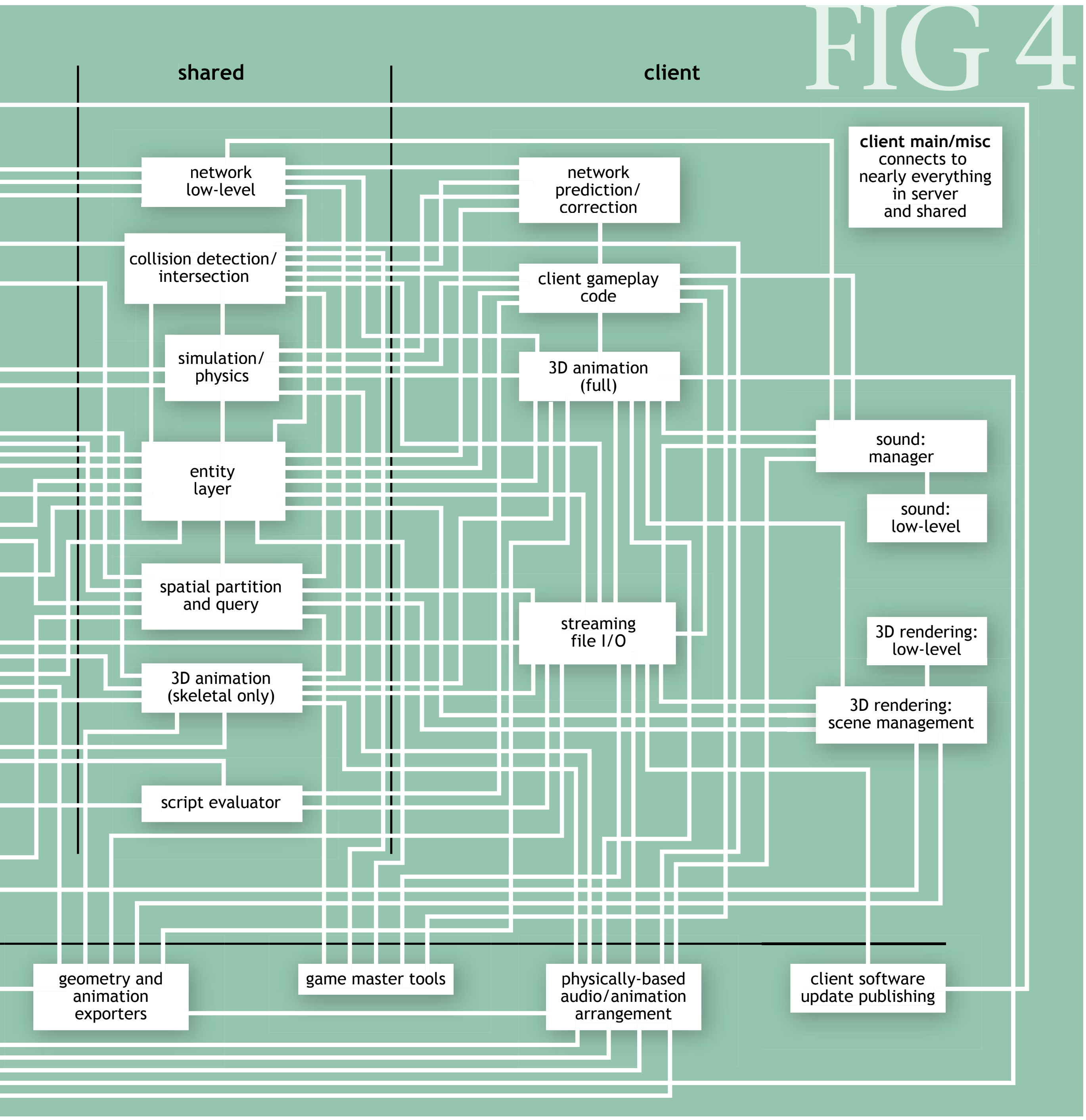
I've discussed a host of tool-related problems that cause difficulty in developing games today. These issues will be slow to change. With better tools and workflow, we will be able to make better games, raising the level of game complexity and functionality that we can handle. However, games will not actually become easier to make because the difficulty of creating a game will always expand until it exceeds our implementation abilities. The next section on the challenges of highly domain-specific requirements will discuss why this is so.

HIGHLY DOMAIN-SPECIFIC REQUIREMENTS

Currently there are three levels of programming in games: script code, gameplay code, and engine code. Script and gameplay code control the overall content, rules, and high-level behavior of the game. For the remainder of this article I will treat them as one concept and just refer to "gameplay code." Sitting below gameplay code is the engine, which provides all the basic mechanisms for simulation and I/O. Engine code is much more difficult to write than gameplay code, first because it requires advanced knowledge, and also because it must be held to more stringent quality and performance standards.

Engine Code. Certainly, to write good engine code, you need to have a good grasp of software engineering. But also, there's a lot of domain-specific knowledge required.

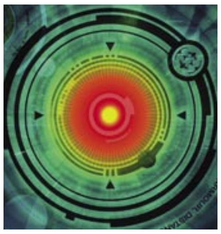
FIG 4


shared
client
client main/misc connects to nearly everything in server and shared
network low-level
network prediction/ correction
collision detection/ intersection
client gameplay code
simulation/ physics
3D animation (full)
sound: manager
entity layer
sound: low-level
spatial partition and query
streaming file I/O
3D rendering: low-level
3D animation (skeletal only)
3D rendering: scene management
script evaluator
geometry and animation exporters
game master tools
physically-based audio/animation arrangement
client software update publishing

This can be roughly broken into two categories, mathematical knowledge and algorithmic knowledge.

Mathematical knowledge. A programmer just isn't going to be competent in a modern game without a decent grasp of basic linear algebra,[4] as well as geometry in 2D and 3D. We often use 4D representations for basic operations (4D homogeneous coordinates for general linear transformations, and the quaternions to represent rotations[5]) so the ability to reason about higher dimensions is extremely useful. Basic calculus is necessary for all kinds of simulation and rendering tasks. For many rendering tasks, signal-processing mathematics is very important—both linear signal processing[6] as well as the murkier study of spherical harmonics.[7] For any kind of sophisticated

Game Development
Harder Than You Think



simulation, you'll want experience with numerical analysis and differential forms. For networking, information theory and the statistics behind compression and cryptography are necessary to build a robust system.

Algorithmic knowledge. A good engine programmer should have working familiarity with a great many algorithms—so many that attempting to list them here would be silly. The most necessary algorithms perform tasks like spatial partitioning, clustering, and intersection and clipping of geometric primitives. Most algorithms will be mainly focused on one task area, like rendering or physics, but these algorithms are often very deep and take a while to master. For years we have been mining academic research to find and modify appropriate algorithms. However, a game engine must meet soft realtime requirements, and most academic work in the relevant subject areas is geared toward batch computation. (Most of the past research in graphics has applied to offline cinematic rendering. Most physics algorithms are unstable and can fail outright, which is solved in a batch setting by tweaking the initial conditions and trying again. These algorithms do not adapt successfully to a soft realtime setting.) As games are now starting to be taken seriously by the academic community, this is beginning to change, but most academic research is still pointed in directions that don't do us much good. So, creating a technically ambitious game engine will often require a substantial amount of original research.

Engine programmers don't necessarily need a deep understanding of all the aforementioned departments of mathematics and algorithms. But because they're working in such a tightly coupled system, even if a concept doesn't arise directly within the module they're working on, it may significantly affect their work by propagating through a neighbor. So engine programmers will need light-to-medium knowledge of most of these subjects in order to get work done, and should be adaptable enough to learn the others as need arises.

Crosscutting Concerns. To successfully build a game engine, it's not enough to understand a lot of math and algorithms. When you put many algorithms together into a tightly coupled system, constraints imposed by the various algorithms will clash. It takes a certain experience and wisdom to choose or discover algorithms that can be combined into a harmonious whole. When game engines fail, it's often because they don't achieve that harmony.

Each of the nodes in figures 3 and 4 represents a complex system full of crosscutting concerns. Also, many of those nodes represent cuts across the majority of the system's conceptual space. Currently we do not have programming paradigms that help us address this fundamental structural problem. (Some new fruits of language research, like *aspect-oriented programming*, are journeying into that area, but none of them are currently practical for production use.)

Depth of Simulation. Game code is inherently about simulating some kind of world. In early games, the simulations were simple and primitive. For a while we focused mainly on graphics, which is a simulation of how light behaves in the game world. But now we are entering a time when the portions of the simulation governing physics and AI can be more important to the end user's quality of experience than the graphics. Since generalized AI is such an unsolved problem, nobody knows what it will look like in the future. Physics, though, we have some grasp of. Working on physics has educated us about some issues that can be generalized as pertaining to all manner of simulated time-evolving complex systems.

Simulating a complex system generally involves integrating quantities over time using numerical methods. At a low level, therefore, quantities must be specified in an integrable way. Functions containing arbitrary discontinuities are very difficult to numerically integrate, but these are also the kinds of functions that computers make by default. (If/then statements create discontinuities unless we make explicit effort that they do otherwise; thus we must be careful with if/then statements when working on low-level simulation!) To help keep things integrable, significant world events, including AI decisions, need to occur at a level higher than the basic integrator; that is, they aren't allowed to just kick in without warning and change the state of the world.

Once we have done all this, we need to worry about *stiffness*—the fact that merely by adjusting constants, you can cause the simulation to become unstable. To the best of our current methods, good integration techniques can only provide an area of stability within the simulation space; you must take care not to step outside that area.

We then need to worry about tunneling, which happens when we integrate across a timestep that's too long, causing us to miss a significant world event. The term "tunneling" comes from collision detection, where we move entities essentially by teleporting them small distances through space; if we move an entity too quickly, it may pass through a solid object like a wall, unless we take extra steps to detect that situation. These extra steps comprise an approximation to "what really should have happened," which may result in consistency problems.

Interesting simulations inherently involve subtle interactions between many different entities, an n^2 problem that doesn't really want to be solved in real time. To work around this issue, we need to be good at culling negligible interactions to pare down the size of the problem. But such culling tends to involve black-art heuristics and can go wrong in strange and subtle ways.

Profiling. We're always trying to push the CPU as far as we can, so profiling is very important. Unfortunately, there are no good profilers for games. Games exhibit heavily modal behavior based on dynamic conditions (at one moment, sending triangles to the graphics hardware may be a performance bottleneck; the next moment, detecting collisions between game entities may be the problem.)[8] To improve game performance, we need to identify these individual modes of behavior. Unfortunately, commercial profiling products inherently average the program's activity over time, which melts all these spikes into an indistinct mush, hiding the problems.

Usually, we build our own simple profiling systems into our games. Though useful, it's not like having a mature profiling tool. Vendors of graphics hardware, like ATI and NVIDIA, make some graphics-specific profiling tools, as do the makers of some game consoles. Those tools are also helpful but generally insufficient to get a bird's eye view of the system.

Risk. Computer games have always evolved toward increased technical complexity to give the players things they have never experienced before. As a result, each wave of games is attempting several technical feats that are mysterious and unproven. Thus game developers carry a lot of technical risk (you can't accurately schedule the unknown or predict how it will interact with the rest of the system) as well as game design risk (how will this never-implemented feature feel to the end user? Is it going to be worth all this trouble we are taking to implement it?).

CONCLUSION

Games are hard. This article has tried to present a broad summary of the reasons why; though many relevant factors have been omitted in order to keep the explanations short.

Rather than being discouraging, the challenge involved in making a game is a major part of the reason so many smart people are drawn to the field. The constant development of new methods, in combination with ever-faster computers to run them on, makes this a very interesting time. Q

ACKNOWLEDGMENTS

Thanks to Michael Abrash, Sean Barrett, Atman Binstock, Charles Bloom, Chris Butcher, Doug Church, Chris Green, Chris Hecker, Casey Muratori, and Jay Stelly for their input.

REFERENCES

1. Fristrom, J. Manager in a Strange Land: Turnaround Time. Gamasutra (Nov. 28, 2003); http://www.gamasutra.com/features/20031128/fristrom_01.shtml (free account and password required).
2. Fristrom, J. Manager in a Strange Land: Content Turnaround. Gamasutra (Dec. 5, 2003); http://www.gamasutra.com/features/20031205/fristrom_01.shtml (free account and password required).
3. Gamasutra (Web portal for game developers free account and password required): see http://www.gamasutra.com/.
4. Sheldon, A. *Linear Algebra Done Right, 2nd ed.* Springer Verlag, New York: NY, 1997.
5. Hamming, R.W. *Digital Filters.* Dover, Garden City: NY, 1998.
6. Eberly, D. Quaternion Algebra and Calculus, 1999 (updated 2002); http://www.magic-software.com/Documentation/Quaternions.pdf.
7. Green, R. Spherical Harmonic Lighting: The Gritty Details. Proceedings of the Game Developers Conference (Jan. 16, 2003), 1–47; http://www.research.scea.com/gdc2003/spherical-harmonic-lighting.pdf.
8. Blow, J. Interactive Profiling 1-3. Game Developer Magazine (Dec. 2002-Feb. 2003).

LOVE IT, HATE IT? LET US KNOW
feedback@acmqueue.com or www.acmqueue.com/forums

JONATHAN BLOW is a gaming development consultant who has been working in industry since 1995. Recent projects include Deus Ex 2 and Microsoft Train Simulator 2. Blow also writes a monthly column, "The Inner Product," for *Game Developer* magazine, focusing on cutting-edge technical issues in game development.