

Second Life and the New Generation of Virtual Worlds

Sanjeev Kumar, Jatin Chhugani, Changkyu Kim, Daehyun Kim, Anthony Nguyen, and Pradeep Dubey, Intel Corp.

Christian Bienia, Princeton University

Youngmin Kim, University of Maryland, College Park

Unlike online games, metaverses present a single seamless, persistent world where users can transparently roam around without predefined objectives. An analysis of *Second Life* illustrates the demands such applications place on clients, servers, and the network and suggests possible optimizations.

A virtual world is a computer-based simulated environment that users can inhabit and in which they interact with others either as themselves or through software agents (bots) or graphical representations called avatars. A virtual world can have anything one might encounter in the real world as well as objects and phenomena with no real-life counterparts.

Due to the limited computational power and network resources available on today's machines, virtual worlds are often implemented using 2D graphics with simple physics and AI engines. This is especially true for virtual worlds that primarily target mobile devices. However, as computers become more powerful, virtual-world designers are using 3D graphics and more sophisticated algorithms. The "Key Features of a Virtual World" sidebar describes some of the functions that a realistic virtual world must support.

TYPES OF VIRTUAL WORLDS

Three-dimensional virtual worlds can be broadly classified into online games and metaverses.

Online games

Online games have been around for over a decade.¹ Some—for example, first-person shooters like *Quake*—are designed to let a small group (up to tens

of users) play together. Massively multiplayer online games (MMOGs) are designed to scale from hundreds to thousands of players simultaneously.

With users geographically spread across the Internet, most online games rely on the client-server model, in which each user runs a client program that connects to one or more machines that run the server program. The server is responsible for managing interaction between the multiple avatars and objects in the virtual world, as well as communication with the clients. Consequently, the server's computational requirement is proportional to the size of the world and the number of clients concurrently connected to it.

In the case of MMOGs capable of supporting thousands of players at the same time, such as *EVE Online* (www.eve-online.com) and *World of Warcraft* (www.worldofwarcraft.com), a cluster of machines is necessary to implement a server that scales with the number of users. The cluster is divided into *shards*, each of which hosts a particular instance of the virtual world. A player selects a shard at the start of each game session and only interacts with players in that shard for the session's duration.

Metaverses

Meta-universes, or metaverses, are fully immersive virtual spaces that significantly differ from online games in several ways.

Key Features of a Virtual World

To provide a realistic experience, a virtual world must support several key features.

Multimodal input. A diverse set of interfaces is needed to effectively support a broad set of user activities in a virtual world. These include standard inputs for a keyboard and mouse as well as—depending on the application’s requirements—voice, gesture, and haptic devices.

Heterogeneous clients. A virtual world must support a broad range of client machines, from high-end gaming desktops to ultramobile PCs. The standard way to address this issue is to provide different levels of user experience depending on client capability. However, to ensure a base level of user experience, the server must be able to execute a core set of features. In addition, some functions must be executed on the server whether clients are connected or not—for example, the script for a clock must run continuously.

Server scalability. To ensure a persistent world, the server must continually send relevant updates of objects and avatars to clients through the network. A virtual world runs at a minimum rate of 30 frames per second, which limits the time to render high-quality graphics for each client to 1/30th of a second. Given these large computational demands, highly scalable machines are needed to implement the functionality on the server.

Network constraints. A virtual world’s servers are connected to high-bandwidth links, while clients are connected to lower-bandwidth links. In addition, clients typically have much higher download than upload bandwidths. Finally, network latency to clients around the world can be significant. Any virtual-world implementation must take into account these

network constraints. For example, to reduce network traffic, the client must maintain a cache of recently visited objects.

Object encoding. A virtual world must accurately represent users, in the form of avatars, as well as objects such as buildings, terrain, and trees. Objects can be modeled either explicitly as a triangular mesh of vertices or implicitly as a parametric surface. Parametric surfaces compactly encode objects using geometric shapes like cubes, pyramids, and cylinders that enable efficient implementation of physical simulations. However, some shapes like human faces are difficult to encode using parametric surfaces and must be modeled as a triangular mesh of vertices. Virtual-world implementations should support both types of object encoding to ensure realism.

Physics engine. A virtual world uses various mechanisms to determine how objects and avatars move or evolve over time. Physical simulations model natural behavior like gravity and collisions, but detailed simulations of interactions can be computationally expensive, necessitating various animation and scripting routines. Virtual worlds also require AI techniques like path planning and crowd simulation to manage non-playing characters, avatars not controlled by humans, to perform routine tasks.

Security, privacy, and fairness. A virtual world must control user access to the various areas within it to address the various security, privacy, and fairness needs of resident individuals, groups, and corporations. Relying on clients to enforce such rules has resulted in numerous well-documented “cheats” that circumvent protection mechanisms. Implementing these rules on the server makes them more fool-proof.

Seamless persistent world. Metaverses present a single seamless, persistent world where users can transparently roam around different regions without predefined objectives. This requires sophisticated communication protocols, highly scalable storage capacity, and automatic load balancing on the servers that simulate the world. Although some MMOGs also provide a persistent world, their scale and complexity are orders of magnitude lower than that of metaverses.

User-generated content. In online games, experts generate all content. While this limits the range of user activity, it also guarantees that content looks good and is rendered at interactive rates. In metaverses, however, users generate and retain ownership of almost all content. Because many users are novices who know little

about their choices’ performance implications, metaverses must automate optimization techniques employed by expert content designers.

Massive and dynamic content. A metaverse is considerably larger than any single online game. It often features a growing economy in which users buy and sell virtual content, including land, as well as real-world goods. Daily transactions in some metaverses can total millions of real dollars.

Metaverse content is also dynamic, with users continually creating and modifying objects and places. In contrast, online games rely on a static scheme that presents the virtual world’s entire content on the client machine. The server need only communicate relatively small amounts of data—primarily user inputs and avatar location updates—during game play.



Figure 1. Scenes from *Second Life*: (a) *Miramare*, (b) *Swedish Embassy*, (c) *Edelman Island*.

SECOND LIFE

Metaverses are becoming immensely popular, with millions of active users worldwide. The growing collection of metaverses used for social networking, workplace collaboration, retail sales, virtual tourism, marketing, distance learning, disaster recovery training, and the arts is often called the 3D Internet.

Some virtual worlds grant access only to specific groups of users. For example, Forterra Systems (www.forterrainc.com) creates or hosts private virtual worlds based on their On-Line Interactive Virtual Environment platform. These metaverses focus on staff training or collaborative decision-making and typically replicate real-world locations.

Most metaverses, however, are open to all users and serve as vehicles for social networking, commerce, education, or entertainment with some fantasy-based elements. The first metaverse, *CitySpace*, was launched at the 1993 Siggraph conference and was active until 1996 (<http://en.wikipedia.org/wiki/Cityspace>). Since then, numerous metaverses have emerged including *Active Worlds* (www.activeworlds.com) and *There* (www.there.com).

The most popular metaverse today is Linden Lab's *Second Life* (<http://secondlife.com>), in which many corporations, universities, cities, embassies, artists, and individuals have created a virtual presence. Figure 1 provides screenshots of three regions in *Second Life*: the futuristic-themed *Miramare*, a virtual Swedish Embassy created by the Swedish Institute, and an island created by global public-relations firm Edelman.

Metaverses' computation and communication requirements significantly exceed those of online games. For example, while MMOGs can handle several thousands of clients per server, *Second Life* can accommodate only about 40 clients per server. To better understand this emerging type of application, we performed a detailed analysis of *Second Life*—its software architecture as well as its demands on clients, servers, and the network.

SOFTWARE ARCHITECTURE

Second Life users run a client program that connects to a central server, which employs three main clusters

of machines. The data servers manage a central database, a logging database, an inventory database, and a search database.² Another collection of servers simulates the virtual world as it evolves over time. Finally, a small number of machines perform utility functions like user authentication during login and instant messaging during sessions.

Figure 2 illustrates the client-server execution sequence during a single time step. Each server interacts with multiple clients and vice versa. The system uses visibility computation to determine the relevant subset of data to send to each client. Servers store all objects and perform the key actions to evolve the world while maintaining reasonable consistency. Clients perform other actions to improve realism, such as cloth simulation, and cache objects to reduce network communication. Cyclic redundancy checks validate cached objects.

To scale processing, *Second Life* divides the world into 256×256 meter tiles. Each tile is statically bound to a particular simulation server that executes on a single CPU core. When a user is in the middle of a tile, the client only needs to communicate with the server responsible for that tile. However, when a user approaches a tile's boundary or crosses into other tiles, the client must communicate with the servers managing those tiles.

To minimize communication, *Second Life* partitions objects in the virtual world among the servers. At any given time, only one server maintains an object's state, and when an object moves across a tile boundary, it transfers the object's state to the destination server. Because time can progress at different rates in different tiles depending on the servers' load, this static partitioning approach can result in visual artifacts. For example, a person walking on a plank across a tile boundary could fall through the plank as if it didn't exist.

Ideally, a virtual-world implementation would present a seamless world to users by marshaling cluster resources based on the load at any particular moment—the system would allocate more servers to busy regions of the world than to inactive ones. However, implementing such a scheme poses numerous challenges and is an open research problem.

CLIENT EVALUATION

To evaluate *Second Life* client performance, we used a high-end desktop with the Windows XP operating system, a 2.93-GHz Intel Core 2 Quad (Kentsfield) CPU, 4 gigabytes of dynamic RAM, and an NVIDIA GeForce 8800 GTX graphics card.

CPU performance

We used Intel VTune software to analyze CPU performance in two basic scenarios. In the *cached* scenario, the user had recently visited the same location and thus a significant portion of the region's data was available to the client in its disk cache. In the *uncached* scenario, the user had not recently visited the location and the client had to fetch all data—descriptions of terrain, buildings, avatars, and so on—from the server.

On a four-core machine, 100 percent CPU utilization indicates that all four cores are executing active tasks. We found that CPU utilization on the *Second Life* client ranges from 16 to 28 percent. This low utilization rate is mainly due to the client spending most of its time in a single thread, despite spawning as many as 24 threads. The other threads are used for offloading high-latency operations, like network and disk input/output, that are computationally inexpensive.

The *Second Life* client supports both opaque and translucent objects to enrich the user's visual experience. Much of the CPU processing is dedicated to sorting translucent objects and correctly rendering them in back-to-front order. The CPU also spends considerable time decompressing textures stored as JPEGs, which require 10 to 20 times less network bandwidth than uncompressed files. Since modern graphics processing units (GPUs) do not support the JPEG format, the client must decompress them before using them for rendering.

The current *Second Life* client does not expend much CPU processing time in character animation or client-side physical simulation. By parallelizing clients to fully exploit multiple processor cores,³ the application achieves better visual effects and generates more detailed geometry for enhanced rendering.

Other client CPU tasks include view-frustum culling and level-of-detail (LOD) computation to reduce the bandwidth for transferring objects to the graph-

Server

Client

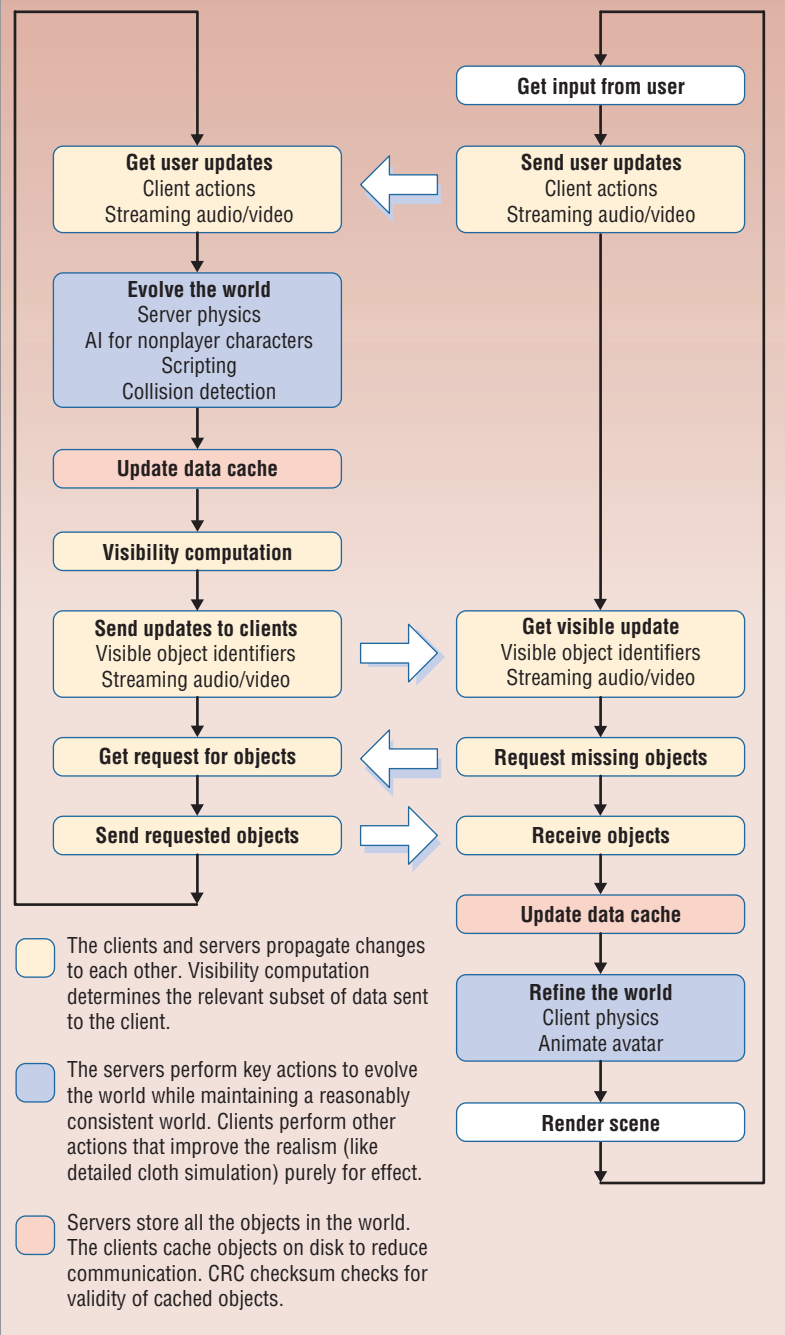


Figure 2. *Second Life* client-server execution sequence during a single time step. Each server interacts with multiple clients and vice versa.

ics card, thereby increasing the rendering rate. View-frustum culling involves discarding objects outside the user's viewing zone. LOD refers to the number of triangles used to represent an object. *Second Life* renders distant objects at a lower LOD—that is, with fewer triangles—than nearby ones. The client maintains a maximum of four LODs for any object, and chooses the appropriate one based on object's position with respect to the user.

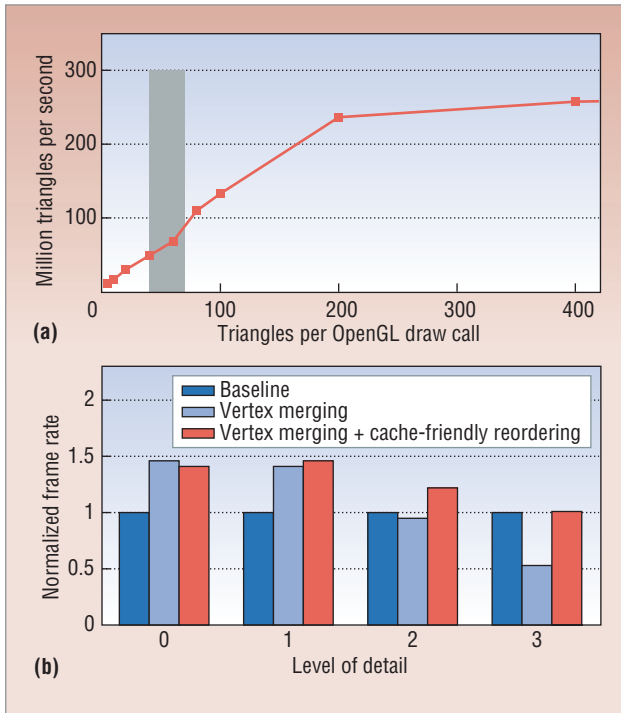


Figure 3. *Second Life* client GPU performance. (a) Triangle throughput for rendering a particular scene. (b) Frame rate optimization.

GPU performance

In general, the *Second Life* client renders triangles belonging to an object in a single OpenGL draw call, with the number of draw calls per frame dependent on the number of visible objects. Figure 3a plots the number of triangles the GPU can render per second depending on the number of triangles per draw call. For a small number of triangles, the GPU spends most of its time interacting with the graphics drivers and thus achieves a small percentage of the maximum triangle-rendering performance. However, as the number of triangles in a draw call increases, performance increases gradually to the maximum possible throughput.

The grey bar indicates the range of triangles per draw call observed in *Second Life*. This number varies between 40 and 70, and the throughput achieved is only 20 to 25 percent of the maximum possible. Generally, virtual worlds are designed to have enough triangles per draw call to minimize the graphics driver overhead. However, the user-generated content in *Second Life* does not consider the overhead per draw call, resulting in the low throughput. By exploiting CPU computation power to merge a few objects into one draw call, it would be possible to achieve three to four times improvement in rendering throughput.

The *Second Life* client triangulates input objects in a manner that duplicates many vertices in the mesh, creating redundant work for the GPU. We implemented algorithms that merge these duplicated vertices and achieved

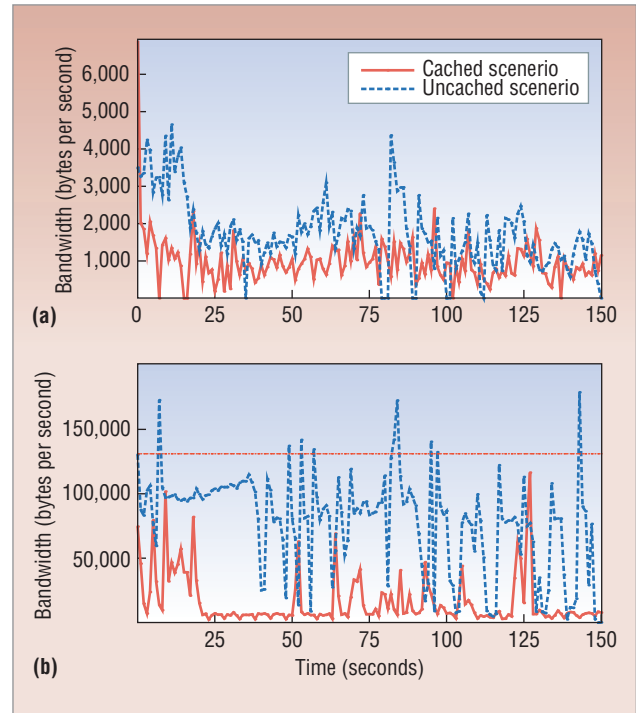


Figure 4. Network traffic during two separate visits—a cached and uncached scenario—to the Swedish Embassy in *Second Life*. (a) Data sent by client. (b) Data received by client.

a speedup of around 50 percent for LOD levels 0 and 1. This reordering, however, produces triangles that are not hardware-cache friendly, and in fact reduces performance for LOD levels 2 and 3. Reordering the triangles to make them cache friendly further improves rendering performance.

These optimizations target specific bottlenecks in the graphics pipeline and are applicable to a wide variety of modern GPUs. Applying them for different LODs of objects in *Second Life* improves client GPU performance by four to six times. We are currently assessing the performance potential of Intel's recently announced discrete GPU chip, Larrabee,⁴ and its more general-purpose compute pipeline for virtual-world simulation-processing needs.

NETWORK TRAFFIC

To analyze network communication in *Second Life*, we captured a trace of raw network packets between our client machine and the servers during two separate visits to the Swedish Embassy, as shown in Figure 4. During the first visit (uncached scenario), the application was fairly unresponsive for a considerable time as the client had to download data for that region. During the second visit (cached scenario), however, the disk cache had data available from the previous visit, and data traffic averaged 80 kilobytes per second.

Second Life clients use at least an order of magnitude more download than upload bandwidth. The

horizontal line in Figure 4b indicates the maximum download bandwidth for individual clients. Online games use significantly less download bandwidth, as game contents such as textures and geometries are preinstalled on the clients. The bulk of download bandwidth usage in online games is for avatar position updates, which require as little as 20 bytes per avatar. Even with 64 clients, online game data traffic is only around 9 Kbps.⁵

Table 1 breaks down client-server traffic in the cached and uncached scenarios of the Swedish Embassy visit by message type. *Second Life* uses 473 different message types broadly classified into the categories indicated. Most of the categories are self-explanatory; “viewer effect” messages communicate physics-simulation results performed on the client to the server. The table shows that texture and object data comprise the bulk of the traffic. Texture data dominates in the uncached scenario, but because textures rarely change and therefore cache more effectively, object data dominates in the cached scenario.

Second Life uses a proprietary communication protocol built on top of the User Datagram Protocol and the Transmission Control Protocol to manage client-server traffic. As reliability is not a requirement for many packets, it relies on UDP for most communication needs and uses TCP only for user log-on and authentication.

To reduce network traffic, *Second Life* employs numerous data-compression mechanisms including JPEG compression for textures and zlib compression (www.zlib.net) for objects. For packets, it uses a simple compression scheme that run-length encodes sequences of zeros while leaving nonzero bytes as they are.

SERVER EVALUATION

We analyzed the *Second Life* server’s performance in terms of both physical simulation and visibility computation.

Physical simulation

Second Life uses the popular Havok Physics engine (www.havok.com/content/view/17/30) to simulate a realistic interactive environment. It primarily supports rigid-body dynamics for objects modeled as spheres, cubes, triangular meshes, and so on along with particle-based simulations. The application does not yet support more complex interactions common in high-end 3D games such as object fracturing and deformation. It performs effects such as cloth fluttering in the breeze, fluid motion in waterfalls and fountains, mist, fire, and smoke entirely on the client side.

Table 1. Network traffic by message type in Swedish Embassy visits.

Message direction	Message type	Cached scenario data (bytes)	Uncached scenario data (bytes)	
Server ⇒ client	Objects	1,696,616	2,343,608	
	Textures	643,095	9,055,926	
	Layer data	180,824	165,502	
	Viewer effect	49,875	90,974	
	Other	85,425	212,209	
	Authentication	65,682	64,142	
Client ⇒ server	User input	66,930	80,451	
	Packet acknowledgments	37,063	74,921	
	Viewer effect	19,692	20,612	
	Object requests	11,502	41,139	
	Texture requests	2,312	30,302	
	Other	16,624	22,745	
	Authentication	15,541	14,625	

Table 2. Computation requirements of physical simulations at 30 FPS.

Simulation type	Data set	Execution time (percent)
Rigid body	1,000 bodies	16.6
	5,000 bodies	72.2
	10,000 bodies	853.7
	20,000 bodies	1,573.3
Fluid	30,000 particles	302.1
	100,000 particles	813.2
	300,000 particles	2,100.4
Cloth	1,000 particles	0.4
	4,000 particles	1.9
	16,000 particles	7.6
	66,000 particles	32.4
	262,000 particles	127.4
	1,000,000 particles	565.5

Table 2 shows the percentage of CPU time required to perform rigid-body, fluid, and cloth simulations at 30 frames per second on a single core of our desktop configuration. A number larger than 100 indicates that the maximum frame rate possible is less than 30 FPS.

We assume all rigid-body objects to be actively interacting in the world. A scene with only 5,000 objects already requires about 72 percent of the core compute power, while a scene with 20,000 objects requires around 16 cores to maintain an interactive frame rate (assuming perfect scalability). Currently, only a small fraction of objects interact in most *Second Life* tiles, thus the server can support a few thousand objects. However, as users strive to build realistic models, rigid-

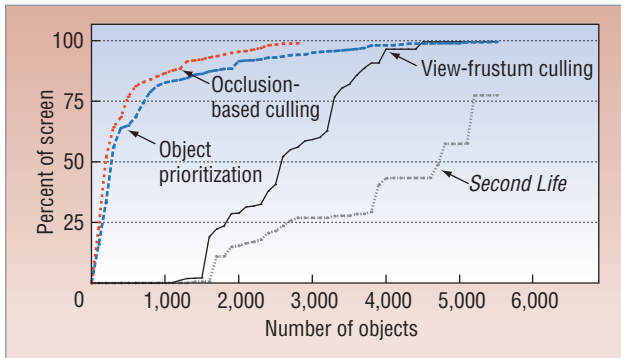


Figure 5. Effectiveness of object-ordering algorithms. *Second Life*'s technique can be improved considerably using view-frustum culling, object prioritization, or occlusion-based culling.

body dynamics computation will likely require several CPU cores.

We simulated fluids using a smoothed particle hydrodynamics algorithm. A simulation depicting water flowing into a bucket requires around 30,000 particles, which translates to about three cores' worth of computation for interactivity. A tub of water requires an order of magnitude more particles and around 21 CPU cores.

We simulated cloth using a mass-spring model that maintains the shape of a particle grid under external forces such as gravity, wind, and drag. Small pieces of cloth and flags can be reasonably modeled using 1,000 to 4,000 particles, requiring only 1 to 2 percent of CPU compute power. However, draping characters in cloth requires 50,000 or more particles, pushing up the compute demand to around 0.3 cores. Extremely large cloth sheets involving millions of particles demand five to six cores.

In the future, we expect the *Second Life* server to execute most of the physical phenomenon to provide a more realistic user experience, thereby increasing the computational and bandwidth requirements by orders of magnitude.⁶

Visibility computation

Along with object and avatar positions, the *Second Life* server computes the set of objects and related textures that the client renders onscreen. The precise algorithm used is not publicly available, but we were able to measure its effectiveness by observing the order in which the server sends objects to the client.

Figure 5 plots the percentage of screen that fills up as the client receives this ordered set from the server. The fewer the number of objects required to fill up the screen, the faster the user can visualize the scene. As the figure shows, *Second Life*'s technique can be improved considerably using one of several algorithms.

View-frustum culling. Each user has a certain viewing frustum that determines the extent of the visible

world. At the very least, the server can compute the list of objects that intersect this frustum and transfer them to the user. This is computationally inexpensive but does not always produce satisfying results. As Figure 5 shows, the first thousand objects transferred to the client occupy less than 1 to 2 percent of the screen, which gradually improves as the user receives potentially all the objects within the viewing frustum.

Object prioritization. A better alternative is to sort the objects within the viewing frustum based on their relative priority. This can be approximated by calculating the projected area of the object's bounding box. At interactive rates, this scheme requires around 15 to 20 percent of CPU compute power per client but, as Figure 5 shows, produces much better results, with 25 percent of the initial objects accounting for nearly 80 percent of the final image. The downside is that it does not take occlusion into account—for example, the server could transfer a large object hidden behind two other objects before either of them.

Occlusion-based culling. The most effective scheme uses object prioritization and also factors in occlusion, with 25 percent of objects filling up around 90 percent of the image. Although this technique is computationally expensive, the server need only send the relevant subset of potentially visible objects to the client, substantially reducing network bandwidth requirements. As Figure 5 shows, the server sends only 45 percent of the total objects because the remaining ones are invisible to the user. This class of algorithms is an active area of research,⁷ with current implementations requiring tens of CPU cores per client for interactive frame rates.

Unlike traditional online games, metaverses like *Second Life* must dynamically provide content—which is mostly user-generated and continually modified—to users depending on their location in the virtual world. Our analysis indicates that this places significant demands on servers, clients, and the network. As virtual worlds evolve to support more users, types of interactions, and realism, these demands will increase by orders of magnitude. Consequently, metaverses will play an important role in the design of future computer systems. ■

References

1. T. Alexander, *Massively Multiplayer Game Development 2*, Charles River Media, 2005.
2. M. Rymaszewski et al., *Second Life: The Official Guide*, John Wiley & Sons, 2006.
3. Y-K. Chen et al., "Convergence of Recognition, Mining, and Synthesis Workloads and Its Implications," *Proc. IEEE*, May 2008, pp. 790-807.

4. L. Seiler et al., "Larrabee: A Many-Core x86 Architecture for Visual Computing," *ACM Trans. Graphics*, Aug. 2008, pp. 1-15.
5. A. Abdelkhalik, A. Bilas, and A. Moshovos, "Behavior and Performance of Interactive Multi-Player Game Servers," *Cluster Computing*, Oct. 2003, pp. 355-366.
6. Y-K. Chen et al., "High-Performance Physical Simulations on Next-Generation Architecture with Many Cores," *Intel Technology J.*, Aug. 2007, pp. 251-261.
7. J. Chhugani et al., "vLOD: High-Fidelity Walkthrough of Large Virtual Environments," *IEEE Trans. Visualization and Computer Graphics*, Jan. 2005, pp. 35-47.

Sanjeev Kumar is a senior staff researcher in Intel's Microprocessor Technology Lab, Santa Clara, Calif. His research interests are parallel architectures, software, and workloads, especially in the context of chip multiprocessors. Kumar received a PhD in computer science from Princeton University. Contact him at sanjeev.kumar@intel.com.

Jatin Chhugani is a staff researcher in Intel's Microprocessor Technology Lab, Santa Clara, Calif. His research interests include developing algorithms for interactive computer graphics, parallel architectures, and image processing. Chhugani received a PhD in computer science from Johns Hopkins University. Contact him at jatin.chhugani@intel.com.

Changkyu Kim is a staff researcher in Intel's Microprocessor Technology Lab, Santa Clara, Calif. His research interests include high-performance microprocessors, memory systems, and parallel processor architectures. Kim received a PhD in computer science from the University of Texas at Austin. Contact him at changkyu.kim@intel.com.

Daehyun Kim is a senior research scientist in Intel's Microprocessor Technology Lab, Santa Clara, Calif. His research interests include parallel computer architectures, intelligent memory systems, and emerging workloads. Kim received a PhD in electrical and computer engineering from Cornell University. Contact him at daehyun.kim@intel.com.

Anthony Nguyen is a senior research scientist in Intel's Microprocessor Technology Lab, Santa Clara, Calif. His research interests include emerging applications for architecture research and next-generation chip-multiprocessor systems. Nguyen received a PhD in computer science from the University of Illinois at Urbana-Champaign. Contact him at anthony.d.nguyen@intel.com.

Pradeep Dubey is a senior principal engineer and manager of innovative platform architecture in Intel's Microprocessor Technology Lab, Santa Clara, Calif. His research

focuses on computer architectures to efficiently handle new application paradigms for the future computing environment. Dubey received a PhD in electrical engineering from Purdue University. He is a Fellow of the IEEE. Contact him at pradeep.dubey@intel.com.

Christian Bienia is a PhD student in the Department of Computer Science at Princeton University. His research interests include computer architecture and emerging applications. Bienia received an MS in computer engineering from the University of Mannheim, Germany. Contact him at cbienia@cs.princeton.edu.

Youngmin Kim is a PhD student in the Department of Computer Science at the University of Maryland, College Park. His research interests include 3D computer graphics, scientific visualization, high-performance computing, and visual perception. He received an MS in computer science from the University of Maryland, College Park. Contact him at [ymkim@cs.umd.edu](mailto:ykim@cs.umd.edu).