Gama Network Presents:

Gamasutra.com

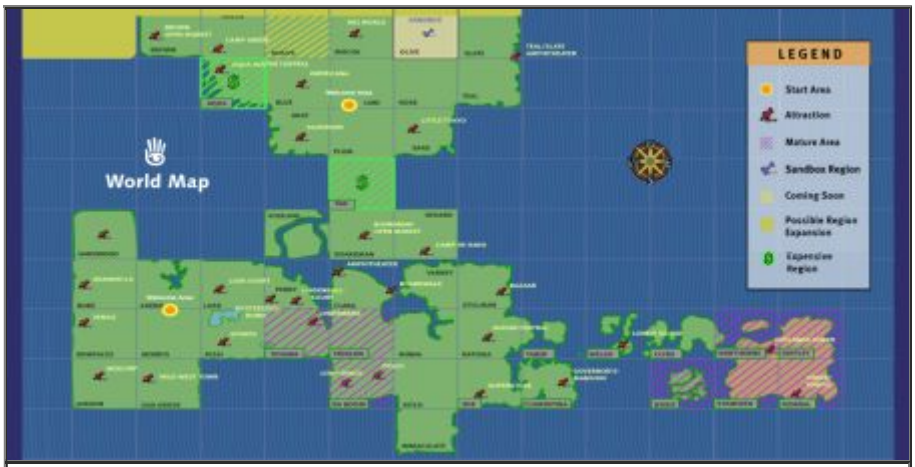
Enabling Player-Created Online Worlds with Grid Computing and Streaming

By Philip Rosedale and Cory Ondrejka
Gamasutra
September 18, 2003

URL: http://www.gamasutra.com/resource_guide/20030916/rosedale_01.shtml

"Within minutes, players have an uncanny ability to discover the size of the possibility space." - Will Wright, speaking at PCForum, 2003

The hours and intensity with which players engage in games makes for very fast consumption of content. Players rapidly feel out the design space of games, requiring creators to adopt strategies to provide tremendous amounts of content for the user to experience (*Final Fantasy*); make the content reusable enough to provide extended replay value (*GTA3*, *Gran Turismo*); or make the experience dependent upon direct interaction with other human players (*Quake*, *Ultima Online*). Persistent-world online games are similarly differentiated by their ability to put tremendous amounts of frequently updated interactive content in front of users. Games which offer 100 hours of content exploration can easily be exhausted by the player in the first month or even weeks of play - making it a real challenge to extract a monthly subscription fee to stay in the environment.



Map of the *Second Life* world. It's also a diagram of our scalable server grid. Each square on the map represents 16 virtual acres of land in *Second Life*. Each square also represents one "sim" - an off-the-shelf Linux server edge-connected to up to four neighboring servers. More land is added by adding more "sims".

Player customization and modification of game features is a strategy that already extends the lifetime of many FPS games, but extending this approach to the MMOG space raises challenging issues. Whereas for single-player games, custom content can be as simple as documenting the various file formats for the players (or in some cases letting them figure it out for themselves!), in MMOG environments a number of weighty technical problems are added. The content (graphics, sound, geometry, animation, and behavior) associated with player-created content must be delivered to other players, potentially in real time. The various permissions associated with ownership and exchange of content must be handled. If player-created objects can exhibit behaviors and via scripting, the resources consumed by those objects must be appropriately limited and balanced.

For our debut project, *Second Life* (http://secondlife.com/), Linden Lab implemented a distributed grid for computing and streaming the game world which supports a large, scaleable world with an unlimited amount of user-created and real-time editable content. In addition, we feel it exhibits a level of performance and content complexity that rivals the best existing 'static' environments. In this article we'll describe how our grid works, and how it benefited the design of the game.

The Trouble with Shards

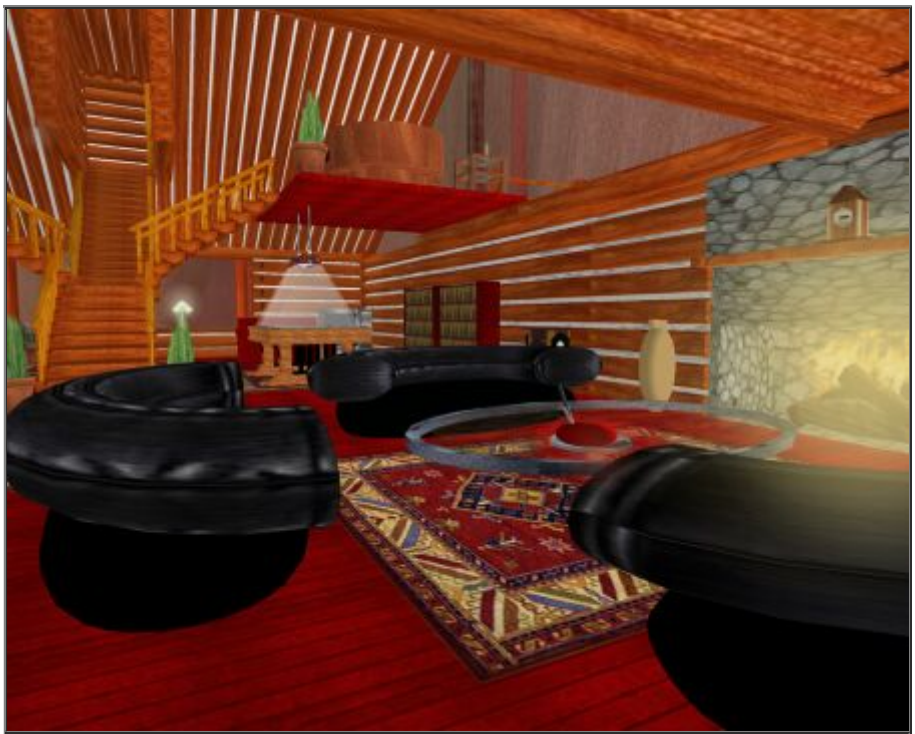
Shards (an expression originally used by *Ultima Online* in its server deployment) split an online game into a number of parallel worlds through clustered servers which have identical sets of world content but different sets of users. The use of shards in a shard cluster has benefits, but shards aren't suitable for deeply user-driven or highly scaleable worlds.

The advantage of shard clustering allows creators to multiply their content development efforts by reusing content across multiple sets of users. But the drawback is that it separates users into separate, non-interacting worlds. Friends are unable to collaborate unless both players are logged into the same server. Additionally, shards are not dynamic load-balancers -- everyone may choose to play on a certain shard, leaving the others empty and losing money. Since some user-developed content (bank accounts, inventory contents, and dropped objects) exists on even the simplest MMOG games, there is no straightforward way to balance or unify the load by moving players between shards. Less populated shards become useless in games which depend heavily on social interactions and are removed, resulting in displaced and unhappy players.

The shard model relies on a fundamental separation between static and dynamic content at the level of the environment (mostly static) and the user (mostly dynamic). The static content can be replicated and the dynamic stuff moves around. So what if you want the whole world to be dynamic? Then either you have to have too many shards, or the whole world needs to somehow become one big one.

The Scale of the Problem

An online world that is comprised of one big contiguous space populated by players who are heavily engaged in creating, editing, and moving objects is potentially a major database problem. In just two months since launch, the number of in-world objects in our game has reached into the hundreds of thousands -- and that isn't counting the things that are in players' inventories. Each of these objects, potentially, has unique textures, physical properties, shapes, permissions, prices, and behaviors. Shipping the entirety of *Second Life*'s current contents to users in a box would require at least 100 CD-ROMs, and attempting to "patch" users with new in-world content every day when they logged in would require downloads of tens of megabytes per day (and this would grow linearly with new users).



Thousands of in-world objects allow users to decorate their houses inside...

If all of this was handled by a single database, the transaction rate during peak hours would be tens of thousands of database reads/writes per second. This is the challenge faced by many upcoming games as well - the amount of content that needs to be manipulated by successful online games in the near future is vastly larger than what we see today.

Enter the Distributed Grid

Like many similar problems in scaling, the massive object explosion mentioned in the last section can only be handled by some sort of solution that distributes these objects across a large system in a way that doesn't create any critical bottlenecks that will fail as the user base and content set scales.

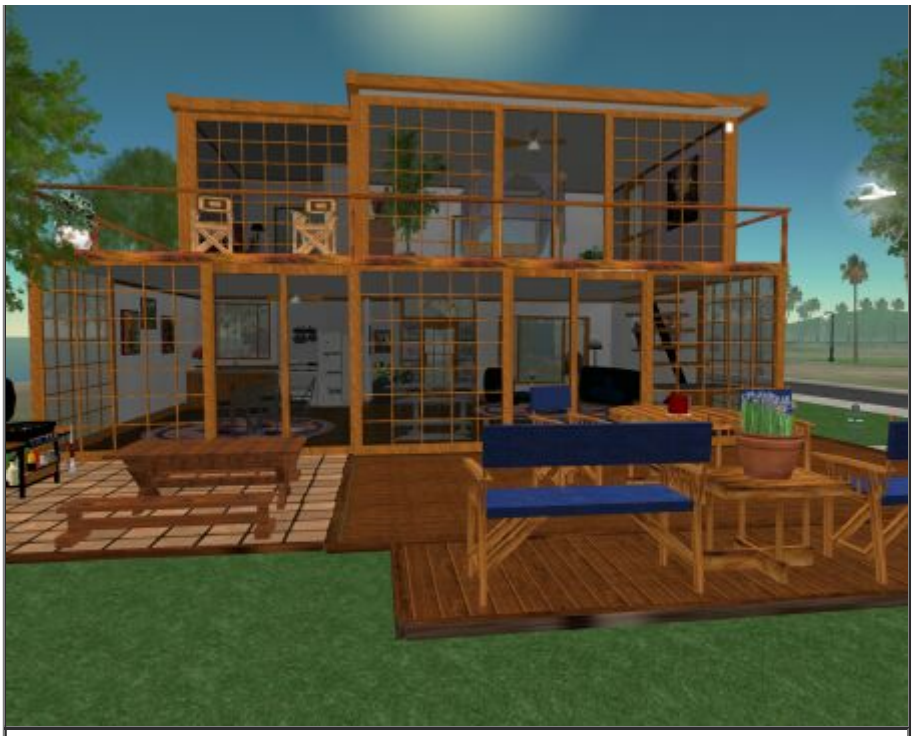
Whether they use a shard cluster or a grid, all MMOGs use multiple machines to distribute the game processing load. To solve the problem of player-created content in *Second Life*, we built a topologically tiled grid of "nearest neighbor" connected machines, each of which simulates the physics, manages all the objects, runs scripts (behaviors) and manages the terrain for everything within a fixed square region of space (about 16 acres per machine). These "simulator" machines talk only to the four nearest neighbors, so there isn't a transactional scaling problem as the world becomes really large. Typical bandwidth between adjacent machines is in the hundreds of Kbps, so the servers can be at different co-location centers, and don't need to be connected via gigabit ethernet or in the same physical cluster.

As objects move around the world, their representation is transferred (along with scripts, objects inside them, textures, and so on) from simulator to simulator as they cross over the edges. Through the use of high-order prediction of the client, these transitions are not visible to the users, resulting in the world feeling seamless. The game's design required the ability for players to move through the world smoothly without loading delays, and while that problem required significant engineering time and effort, solving it correctly proved to be a critical success factor in building a physically simulated world across multiple machines.

Physical simulation (which is a real scaling problem because collision computations for large numbers of objects scales worse than linearly) is also passed between simulators -- each simulator handles all the objects in the column-space above its "land". This supports full rigid-body dynamics and accurate, polygon-level collision detection, as well as cellular-automaton-based simulation, such as our winds and clouds that time-evolve across the entire world resulting in a level of complexity not possible on a single machine.

As you move around inside the game world, you maintain a streaming connection only to the simulator machines you are near. The simulators compute the objects and information that you can currently see, and transmit information appropriate to only those objects that are either new to you or that have changed. This approach offers the advantage of being implemented with inexpensive Linux-based server hardware, and allows a very thin client to be the only thing a player needs to download and install.

As the world grows, the number of back-end server machines grows with it. Currently, each simulator machine is a commodity 1U Linux box that handles a square 256 meters on a side (16 acres), plus all the airspace above it. Linden Lab creates terrain for a rack of machines at a time (specifically, a short rack of 33U for power density reasons) and deploys new parts of the world in relatively large chunks. This allows our users to engage in land rushes with hundreds of users flying out into the unexplored parts of the world looking for the perfect new home location.



And out.

Populating the World

While a distributed computing grid enables a contiguous online world capable of handling huge amounts of player-created content, the world still needs to be populated with content. There have been several "eureka!" moments as *Second Life* grew out of its original incarnation as research in streaming 3D content. The first was that if you developed a system that allowed the whole world to be delivered in real time to the users, there was no reason why that content couldn't be highly dynamic, fluid, and changing, rather than constrained to one content set on a CD. The second, and much more important realization, was that if the content could be dynamic, then the users should be the ones creating and customizing it.

Historical critiques of user-built environments have often focused on the fact that without a central design aesthetic, or severe zoning, or approval of all content, such worlds explode to include mostly bad content - lots of unfinished experiments. We strongly agreed with these thoughts, and designed an economy for the game to solve the problem.

In short, part of the "play" in the game is amassing in-world money to be able to afford the things you want to put into the world. There are taxes that are levied (like real-world property taxes) that restrict how much a player can have in-world, and how much land the player can own. Additionally, the in-world economy includes systems that reward visiting places and being well rated by other users. The effect of these features is to "evolve" the world organically, based on societal preferences -- things that aren't popular or can't make money tend to go away.

An additional and oft-overlooked issue with allowing users to create and shape the structure of a game world is that the resulting density of interactive objects in the world can become enormously higher than that for a visually comparable static environment. In centrally-designed (that is, by the game developers) content sets, considerable time is spent balancing global and local resource density and allocation. In user-created worlds, these issues are typically handled by fine-grained local rules such as land ownership, economic concerns, and caps of overall objects that can be supported in an area. The resulting worlds that are created are fascinating and vital places, but typically have a higher density of dynamic objects than a comparable world designed by the development team and delivered to the users. This disparity can potentially be very large -- factors of 10x or more -- and therefore it places impossible demands on the system architecture.

Further exacerbating the problem, if the design goal is to provide real-time creation, modification, and delivery of the content, common preprocessing techniques such as potentially visible set determination for geometric complexity or radiosity techniques for lighting are not available due to the complexity and computational requirements of these methods. When we realized this problem, we knew that we had to find a way to solve the problems of delivering geometry, textures, sounds, and animations to the users in real time. It also prompted us to create *Second Life*'s 3D modeling and texturing tools, which are embedded into the client and used by both our residents and in-house art team.

Streaming Reality

Second Life's grid topology allows for a tremendously large, seamless world, with the computational, storage, and transmissions loads spread across many simulator machines. It also enables real time creation of content. However, real time creation of large amounts of content creates a significant delivery problem. Even with a cleverly scaleable system that sends information about only the objects a player needs, there is still the problem of bandwidth. Given that many of the objects are things that have not been seen before, the player needs to receive the geometry, textures, sounds, and animation for things coming into view, and they need these assets very quickly. Load times and constant pauses would prevent smooth motion through the large world enabled by the grid, so a streaming approach is required. Streaming also becomes necessary if you want to enable what might be the "grail" of online gaming - the ability to not only make, customize, buy, or sell interesting things, but the ability to do it with someone else watching, or collaboratively with others. The feeling of immersion and community that comes from helping someone work on their virtual car and then driving off with them in it is something that must be experienced to be believed. For this reason, we focused on building a "streaming" system rather than a "patching"' system for the game -- one that could actually get the pieces of new content to the player in real-time.

The first decision was to require broadband -- something more and more games are likely to do. The average bandwidth to a Second Life player hovers around 100Kbps, although players with larger pipes can allow spikes to several times that if their connection allows it. Even with broadband, a huge amount of compression is needed. Each simulator in *Second Life* can handle around 10,000 objects, and although you can't see all those objects at any time, you can often see a good percentage of them. Existing compression techniques for transmitting generalized meshes could never get that many objects into view at a speed adequate for making the experience feel fluid, so the decision was made to throw out meshes for most of the objects in world and to switch to a constructive solid geometry approach built around geometric primitives. These basic shapes are represented algorithmically and allow a tremendous degree of flexibility via scales, cuts, twists and through combinations of multiple primitives that can be transmitted in an extremely terse form.

Another key part of streaming is texture and audio data. *Second Life* uses progressive techniques to allow players to put thousands of large textures and a huge number of audio sources into a scene, and then to stream only the levels of detail that are needed. A player can zoom in on a quarter and see the scratches in the metal, or walk into a jungle and be surrounded with sounds. The game also streams physical information about moving objects very compactly, allowing a player to be surrounded by lots of moving things. The MMOG scenes that generate real visceral appeal (like big battles) can really only feasibly be done using server-side simulation and high levels of compression here allow players to see hundreds of moving objects at broadband rates.

Streaming data to viewers in real-time is computationally expensive. It requires passing data related to frustum culling, change detection, motion interpolation and extrapolation, compression, and packet construction and management. Perhaps the biggest challenge of the streaming reality model is bandwidth and packet management. On top of all of the usual network and streaming complexities (managing packet loss, packet ordering problems, surprising routing decisions on the part of the major bandwidth providers, and so on), at any given moment, a client can be receiving object data from multiple simulator machines in the grid, plus textures and audio from the machine that they are in.

This can result in over a dozen distinct streams of data from over nine individual servers. In order to make the system as responsive and controllable as possible, we chose to build our networking code directly on UDP in order to avoid TCP's "slow start" congestion controls. This required the building of our own reliability system, but led to the interesting realization early in development that TCP-style reliability is rarely what was needed in a dynamic environment like ours, since by the time a packet was retransmitted, the data was often out of date. Instead, much of the game's reliability is built around loss detection in order to allow correction of data in the new updated stream.

Another decision was whether multiple servers in the grid should stream data to the client in parallel, or whether one machine should coalesce the data. The original design included the idea of "proxies" that would act in this manner, but we ended up not using them because they weren't needed. One consequence of this decision is that we stream to many different ports on the client machine. That sometimes means that the game gets blocked by some older NATs, as well as firewalls that are set to block or throttle UDP packets. This also makes the option of tunneling our stream in HTTP much more complicated than if we had a single, central stream.

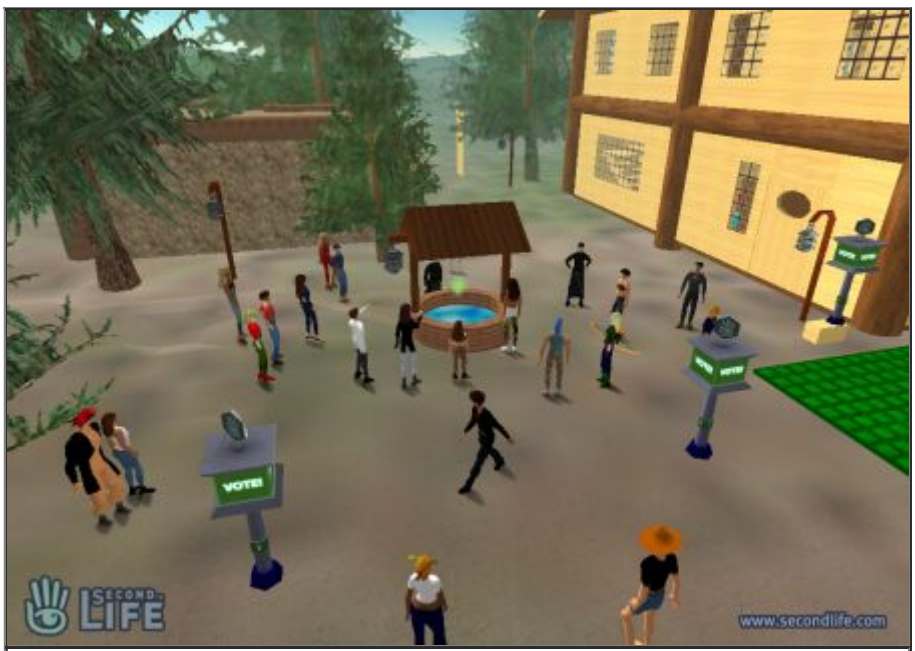
Second Life was designed to be distributed via the web, so a small client download was always a requirement. The client, or "viewer", is currently part of a 10MB download that has broadband users into the world minutes after hitting the web site. By keeping the viewer's functionality to minimum and pushing

the actual processing onto the grid as much as possible, the *Second Life* viewer is able to download, install, and launch very quickly, as well as prevent many common client-side cheats. Upstream messages are largely constrained to keyboard inputs, allowing the simulator machines on the grid to verify both the plausibility of the request and whether the data is coming from the session and location it expects. Users are also protected from each other, since no data is routed directly between client machines. While the viewer is not tasked with simulation or game logic processing, it does have to handle the computationally intense tasks of decompression and rendering simultaneously. Load balancing the viewer (and handling the amazingly broad range of computing horsepower available to our users for both the CPU and GPU) continues to be an area of development and improvement.

Build Not Buy?

Linden Lab built the simulation, streaming, and rendering architectures in *Second Life* from scratch. Although there are products/platforms that did some of the things that were needed, none of them could meet all of our requirements.

There have been some projects on a similar development timeline to *Second Life* (Butterfly, Zona) that use different distributed schemes for managing persistent content and sessions on game servers. We found that these systems weren't designed to handle the huge number of user-created objects and behaviors that our game generates, nor did they include technologies for streaming new objects in a real-time manner, precluding the kind of real-time collaborative creation that has proven so desirable to our players. These capabilities are so intrinsically linked to the server-side object management scheme and physics engine that it didn't make sense to try and add it to an existing platform.



People streaming in and out of views.

Existing rendering engines cannot handle the extremely high rate of change to the scene information that happens in *Second Life* as objects are streamed in and out of views. The usual 3D pipeline is based on the ability to massively accelerate complex, static content -- low numbers of high-polygon objects. *Second Life*, in allowing its residents to build the world, creates huge numbers of relatively low polygon objects, as well as huge numbers of textures and light sources per scene. Different approaches to lighting, shadows, and scene hierarchy are needed in Second Life than we could find in any existing platforms.

Linden Lab used existing products where it made sense. *Second Life*'s graphics and UI are built on OpenGL because they need to run on Macintosh and Linux. For physical simulation and collision detection of rigid bodies, Linden Lab licensed Havok for its performance with the huge number of physically active objects the game might need to simulate. For progressive compression of texture data using the JPEG2000 standard, we chose Kakadu for to its speed and stability. *Second Life*'s audio format is Ogg Vorbis and, to support cross-platform development, a conversion was recently made to the FMOD audio library. Nagios is used to monitor our back server grid.

Pulling It All Together

Second Life brings together a unique set of technology to enable an entirely new user experience -- collaboratively creating and inhabiting a vibrant 3D online world. Through the use of an edge-connected grid topology, typical clustering limitations were eliminated, allowing us to create a large, contiguous

space. Streaming technology allows the world to be created and changed in a fluid, collaborative, and interactive manner by users from all over the world.

So what's the result for players? Imagine a world that stretches to the horizon, where trees sway in a virtual breeze that blows across all the simulators. In the far distance you can see an amazing complex city - dozens of flying avatars arc into and out of it. Lights detail the buildings. Your friend drives up to you and gets out of her hover car. You take a snapshot of her and playfully drop it onto the mirror shades of her sunglasses (she can see it the instant you drop it into the world). She asks for some help with the code that drives the car - you click on the fuselage and pop open the script, edit the thrust levels, save, and have her test it. She gives you an earring she is wearing -- a jewel lit from within that lights up the detailed tattoo on her cheek. You get on next to her as she heads for an Unsafe area kilometers from the city. You pull out a rifle that you bought, confident that the arms merchant who built it knows her stuff. The roar of the bike's engine mixes with the wind in your ears as you cruise down into a dark canyon, dodging a bridge that hadn't been completed the last time you flew this route. This is what you can do today in our game, thanks in large part to the adoption of our server grid.