The Continuous World of Dungeon Siege

Scott Bilas
Gas Powered Games

Introduction

Chris Taylor's Dungeon Siege is a fantasy action game by Gas Powered Games that shipped in spring 2002 after four years of development. One of the key requirements of the Dungeon Siege engine is that once the game starts, the player never sees a loading screen or even a hitch in gameplay from beginning to end. Chris called this feature the "continuous world". This paper is the story of its development and how it works. First we'll talk about what continuous world means to Dungeon Siege, then we'll cover implementation of the various components of the system, along with the hard lessons the development team learned along the way.

Continuous World?

Many games have worlds that are continuous, for example Crazy Taxi, Jak and Daxter, Spec Ops, and most flight simulators. So what does that term specifically mean to Dungeon Siege? Originally, it simply meant "no loading screens", where the gameplay never stops or hitches for any reason during normal play. All map-specific game content, such as geometry, scripts, and textures, is loaded on demand in the background as the game plays out, and not preloaded all at once at the start of the map. As the engine evolved, though, more requirements sprung up as we realized what was possible with the technology:

- World content is streamed in on a fine enough scale to avoid forcing the players through choke-point transition areas. Vast open spaces can be (and were) created, allowing a large variety of gameplay styles, such as the linear single player campaign and the hub-style multiplayer map.

- There are no arbitrary constraints on the size or shape of the world in any direction. A level designer can start to attach new nodes to any point in the existing world and head off in a different direction altogether. The engine is designed to integrate this new addition in a seamless, continuous fashion.

- Designers can create rich and dramatic worlds with a high density and variety of placed objects, huge vertical distances, spiral staircases, lots of fat textures packed tight. The density of textures and objects is a critical part of the "look" of Dungeon Siege.

- Multiple players can simultaneously move through any part of the world, playing together, in bands, or solo, joining and leaving the game on the fly, in a client-server setup. This requirement nearly killed us.

- It must be possible to teleport from one part of the map to another in a “continuous” way, i.e. without stopping gameplay. This was an unexpected and scary (but critical) feature that we hacked in at the last minute.

By the end of the project, continuous world had spread its tentacles into nearly every part of our collective consciousness.

Concepts

The continuous world for Dungeon Siege consists of three main software components: the terrain system which is the world itself, the game object system which is the interactive logic (the “action”), and the world streamer which does the continuous loading. Concepts from these are all very closely tied together so let’s quickly cover those first.

Siege Node

This is the basic 3D terrain element of Dungeon Siege. The terrain system drives the entire game, and so Siege Nodes (or simply “nodes”) are its fundamental building blocks. Many other games, especially the 2D variety, have a similar building block concept, and may call them “tiles” or “static meshes”.

Siege Nodes are 3D meshes created and textured in 3D Studio by artists, exported using a custom tool, and then checked in to source control for the game to use. There are no constraints on the size or shape of nodes, but typically they are built on a grid and are of regular sizes such as 4x4 meters or 4x8 meters. This is done for ease of world building, and is similar to how the Lego system works (although the nodes look nothing like Lego pieces, and are usually highly detailed). As required, the artist will mark node polygons as floor, water, etc., for the game engine to use for its interactive content. Floor polygons are special and are heavily preprocessed for pathfinding optimizations when the map is saved from within Siege Editor, our world building tool.

Siege Door

Nodes are connected together using “doors”, and may have as many of these as they like. A 3D Studio tool lets the artist choose a set of vertices and say “that’s a door”. The term “door” is a relic from a time when the doors really were wooden doors and such. It simply refers to a connection point on a node.

Game Object

Game objects (or “Go’s”) are used to represent all non-terrain and interactive logic content. Almost all other games have something similar, though they are usually called by other names such as Entities, Objects, Actors, or Pawns. In Dungeon Siege, every tree, bush, monster, sword, and pile of gold is a Go. Invisible objects that control gameplay are also Go’s, such as mood altering triggers, light flickering controllers, and path waypoints for AI commands. Every Go is based in a

Siege Node, which acts as a sort of spatial owner. On a higher level, nodes act as buckets for spatial sorting and relative query purposes (i.e. “who’s near me?”).

World Frustum

The world frustum refers to the set of nodes and Go’s that are active and kept in memory. It is a large superset of the view frustum, and is represented as a box of configurable dimensions whose center is located at the position of the Go that owns it. The world frustum dimensions are set by default for the map but can also be adjusted on the fly as needed to meet performance demands for a particular area. The world frustum can be visualized as a bubble moving through the world node graph, where its contents are the game world, and anything outside of it *does not exist*. If we were to fix the frustum’s position then travel to its border, we would see the “edge of the world” (i.e. nothingness).

It’s worth mentioning that, technically, the term “world frustum” is completely inaccurate and misleading, and thereby follows the grand tradition of game developers cheerfully butchering terms from academia. A *real* frustum is a cone or pyramid cut by two parallel planes, and so a more proper term for the world frustum in Dungeon Siege would be something like “world working set plus eviction algorithm”.

Terrain Implementation

The terrain system is one of the most successful components of Dungeon Siege, partly because of its carefully thought-out design and partly because James Loe, the owning engineer, was so meticulous about keeping it speedy and flawless. It went through several iterations and had a lot of critical features like elevators and fading added to it before it came to rest, but its basic functionality stabilized early on and formed the foundation of much of the game.

The Precision Issue

From the beginning, the engineering team knew that the continuous world was going to significantly affect the engine and content design, and the core issue was numerical stability. Imagine two characters walking in formation two meters apart heading east away from the origin. At some point, the distance from each other is overwhelmed by the distance from the origin, and the characters will appear to be “at the same location”.

With floating point, the further you get from the origin, the more precision you lose, which can cause all manner of nasty problems. Things don’t sort right, cracks appear between adjacent meshes, space starts to quantize, and cats and dogs start living together. Dungeon Siege uses the FPU in single precision mode for the obvious performance benefits, and to match the native precision of the video hardware. However, even if we increased the precision, it ultimately could never solve the problem because the world was planned to be, and ended up, incredibly large.

The precision problem meant it would not be possible to have a unified world coordinate space like most other games. Instead, the solution was to segment the continuous world into a set of independent coordinate spaces, and switch among them periodically to reset the precision. A

variety of ideas were tried out within these constraints, and we eventually settled on a variation of a standard portal system.

Our solution consists of a relational node-based coordinate system, in which each chunk of geometry (Siege Node) has its own coordinate space, and is linked spatially to neighboring geometry via the doors it shares in common with those neighbors. The arrangement of nodes connected by doors forms a continuous graph that represents the entire world map. This node system evolved over time from its original goal of maintaining FPU precision to become the primary method of efficiently subdividing space, and the root of countless optimizations.

In order to encapsulate the concept of a 3D position *relative to a specific node*, the traditional (x,y,z) vector had to be augmented with a node ID (x,y,z,node) and represent an offset from the origin of a specific node instead. This 4-tuple is encapsulated as a Siege Node Position, or SiegePos. Later on, we added a SiegeRot (quaternion,node) in order to handle comparisons between orientations across nodes.

The phrase “there is no world space” became a mantra to the team, although it literally took years for everyone to fully grok what it meant.

From Martha to Siege: Engine Mechanics

Originally the Siege Engine was called the MSL (“Martha Stewart Library”) because it was all about stitching terrain sections together along their seams, borrowing vertices from one mesh to use in another, resulting in a large patchwork quilt. Each patch in the quilt could be constructed using an independent method (static mesh, height field, procedurally generated, etc.) as long as the seams between the patches remained aligned. After some discussion and experimentation (and we ran out of Jolt) we dumped all but the static mesh terrain construction technique. We then determined that the added complexity of stitching was unnecessary and instead chose to just render the nodes right on top of each other. With precise math, the seams would not be visible. In practice, this worked out quite well (especially considering the name change in the library).

The terrain portion of a map in Dungeon Siege is a simple graph of instanced nodes, each with an ID unique within the map, hooked together through their doors. The 3D Studio tool that lets an artist mark doors on a node converts each door into the transform that would be required to hook another node up to that door with the proper alignment. This transform is the key to constructing terrain, in that it tells the system exactly how to place one node next to another so that the world appears seamless. Getting from the connected node back to the original is a simple 180 degree rotation of the transform.

Although the Dungeon Siege world is “true 3D” with arbitrary 3D meshes representing terrain, it is very much aided by the fact that up is always “up”, anywhere in the world. The game design does not require the up vector to ever change, and although great vertical distances are required by the game for mood and other aesthetic purposes, most parts of the interactive game experience can be engineered as if it were all in an old-fashioned 2D tile engine, conveniently ignoring the vertical dimension. Nodes always “stand straight up”, and are rotated only in flat space to connect to each other (again, this is similar to the Lego system). This vastly simplified many parts of the engine and permitted significant optimizations. For example, the Dungeon Siege pathfinder is blazing fast, because it can work by “projecting” the terrain onto 2D (sort of).

This is part of the reason we don't have true flying creatures in Dungeon Siege, but instead have hovering creatures. The only difference between them and ordinary walking creatures is that their animations place them above the ground instead of on it. This is one of those cases where we saw the limitation and changed the design to compensate, e.g. we don't have a whole lot of flying monsters. It's still silly to see some of the flying creatures carefully pathfind over to a bridge, "fly" across it, then go to where the players are to attack. Better monster placement could have solved this, but in general it was not a problem worth worrying about.

Constructing Worlds

The level designers use Siege Editor to instance Siege Nodes into maps by choosing a door on an existing node, choosing a new node to place near it, and then flipping through doors and orientations on the new node until it "fits". Repeat this tedious process hundreds of thousands of times, and you end up with what Dungeon Siege shipped with – several thousand unique nodes instanced into two gigantic maps with 30 or 40 hours of play each.

The design of the actual nodes changed drastically over the course of development, as the artists and designers experimented with a variety of styles. The nearly constraint-free node and door design also guarantees that experimentation will continue long into the future. However, because the shape of a node and how it is placed in the world are closely connected, getting a good process in place here for the art pipeline is critical.

During development, it became obvious that a lot of the nodes were identical except for their texture mappings. This was common with floor tiles, ramps, hills, rivers, etc. So we invented "generic" nodes that can have texture sets applied to them, for themes like snow, grass, and forest, letting the designer switch among these on the fly in Siege Editor. Our team has generally settled on Lego-style, regularly shaped, reusable rectangular pieces because of its flexibility in world building, with particular areas having many "custom" nodes that are intended for one purpose (such as the dragon's lair).

Regardless of node design, the tile model has a significant advantage over traditional levels in that the testing is far simpler. Because the nodes are instanced, any bugs in texture mapping or polygon flags will be repeated and easier to find. And once fixed, it is fixed for all instances of that node in all maps that use it. Unfortunately this is purely a node-level thing, and Go's, because they are individually placed and set up, do not necessarily enjoy these advantages to the same degree.

The Siege Editor does not use a continuous world, and instead keeps all the nodes and Go's in memory at once. This is mainly done for programming and user interface convenience. Managing a continuous world in the editor would be difficult. But with an essentially infinite sized world, keeping everything in memory at once eats up memory and CPU quickly, making editing difficult, so the maps are subdivided into chunks called "regions" that fit into memory better for editing convenience. This also solves the problem of multiple designers wanting to work on a map simultaneously.

Regions are stitched together at their edges using a tool in Siege Editor that builds a small stitch database for use by the game. The game does not need to pay attention to regions, except for debugging and content lookup purposes, so the stitching is merged into the main node map inside

of the loader only, and the rest of the game doesn't know the difference. During normal gameplay it considers the entire world to be a continuous stream of nodes and Go's.

There were a few problems that regions caused, unfortunately. Lighting across them was difficult to line up, and could only be seen in-game, and not in the editor, which made it tedious to fix. And any scripting that was intended to occur across a region boundary had to be rearranged or carefully constructed to work.

The Space Walk

Of course, there *has* to be the concept of world space at some point, otherwise it would not be possible to do basic operations such as calculating difference vectors and rendering triangles. Dungeon Siege does indeed have world space, however it is not guaranteed to be valid for more than one frame at a time, and should not *ever* be stored. Only a SiegePos has enough information in it to stay valid, so there needs to be a way to convert it to or from world space.

The Siege Engine tracks what it calls the "target node". Its node space *defines* world space until the target node is changed, which happens periodically. The target node is chosen by using the SiegePos of the currently active world frustum. When the frustum's center crosses a node boundary, the target node changes. At that time, the coordinate system for the game changes and space must be rebuilt for all the nodes in the active working set. This means that each node must recalculate how its space relates to the target node's origin. This is done by "walking" outward from the target node, visiting each neighboring node through its connected doors, and accumulating transforms. This is similar in concept to how a skeletal animation system transforms bones into world space.

When the space walk is completed, each node in the world frustum will have cached the transforms required to convert a SiegePos to or from world space. Those transforms are valid until the next target node change, which could occur at any time. During that frame of simulation, converting any SiegePos rooted in any node of the working set to or from world space is a simple transform operation.

The question arises: how often should we switch to a new target node and force a recalculation of the transforms for each node? It should be reset as infrequently as possible to avoid bogging down the CPU with transform calculations, while still enough to keep floating point precision issues in check. Another issue arises – say it's possible to only reset it every five minutes of play. This raises major testing problems. Code would inadvertently come to rely on the fact that the coordinate system is stable for a while and only at those five minute markers would we get a chance to see that code fail. So to avoid this insane boundary condition, we assume that *every frame* it changes, and counter-intuitively change our coordinate system as *often* as we possibly can without hitting the CPU too much.

Space Usage

In practice, we don't do the expensive type of space transformations all that much in the game. While all this constant rebuilding of space certainly sounds scary, most of the game operates in a pseudo-2D mode, and does only small amounts of basic math, costing very little CPU overall. However, certain parts of the engine, such as the path collider and path follower, do spend a lot of

time doing spatial calculations, and going through the transform for every SiegePos comparison would quickly overwhelm the CPU. So instead, these systems will establish and work within temporary local coordinate systems that remain consistent regardless of any change in the target node, and this bypasses the performance problems. This trick does run into problems when nodes start moving around on the fly (we call those types of nodes “elevators”), but that topic is well beyond the scope of this paper.

Another option is to work completely in node local space. The particle effects engine can get away with this frequently when the effect is small and immobile, such as the fire in a torch. Unfortunately, this engine, along with the physics engine (it manages arrows and flying gibs and such), have an entirely different set of problems to solve regarding space, such as:

- A lightning bolt or an arrow path will usually straddle several nodes. This has complications as one or more of those nodes enter and exit the world, especially when the owner of the arrow or spell effect is in yet another separate node.

- Nodes are used for coarse culling of the rendered scene. This is convenient, but when we are staring at the center of a long effect whose endpoints are outside of the view frustum, special code is required to make it render despite the culling.

An interesting problem we ran into was making arrows work. For an arrow to damage and stick in a Go, it must be able to detect a collision with it, which requires knowing what to collide with, which means knowing what node the arrow is currently in. This is more difficult than it may seem, given the arbitrary 3D connections between nodes, and the fact that projectiles can be shot at any angle. As an arrow moves it constantly ray traces straight down to ask where it is (this is a heavily optimized function), and recurses through the graph a bit to find the “best” node, which is then used for collision testing. If the depth is too far, it will get the wrong node, and not think it hit anything. So for example, shooting arrows over a chasm will not work – it will simply pass straight through the Go it was aimed at – unless there are actually nodes connecting the two sides. This was an interesting source of obscure bugs with arrows not hitting targets.

Actually, arrows and other projectiles in Dungeon Siege are just completely wacky in general – they would break in some fashion nearly every couple weeks of development, whether it was fading improperly, popping to be 10x their normal size for one frame before deletion, or sticking into things at the wrong angle. We definitely plan to improve this system in the future.

Evicting Nodes

The world frustum is effectively a cache management system. It determines what is kept active in memory, and what is thrown away (evicted). The system chooses what nodes are contained in a world frustum by walking outward from its center node and intersecting a box whose dimensions are defined by the world frustum with each node it finds in the map’s node graph. Any nodes inside or cut by the box are considered active and loaded, and any outside are unloaded and purged from memory.

Originally it was fairly simple – there was only a single world frustum, and it was controlled by the location of the camera. However, with the requirement that the party may split up, and the need for people to play in independent locations in the world in multiplayer, the single frustum idea wouldn’t

work. While the party splitting feature ended up being dropped as an important requirement for design reasons (although it is still supported), multiplayer remained a key requirement. To solve this, we added support for multiple simultaneous frustums, with one assigned to each Go that can be directly controlled by the player. Because any party member can be moved wherever the player wishes, each gets its own frustum. Summoned monsters can't be controlled directly, and so they don't get frustums.

Necessary as it may be, having multiple world frustums complicates the game considerably, especially considering many horrifying pathological cases we ran into during final testing. So there are several independent coordinate systems running at once, each of which may change at any moment! What happens when they overlap? Who "wins" and owns the space for each node? The solution was to invent a "glomming" technology that lets frustums merge and separate, with a dominant frustum being the owner of the intersection of nodes among the glommed frustums. Each world frustum has an ID assigned from a bit field (so there may be up to 32 frustums total) and node and Go membership is the OR'd result of the ID's of the frustums that it intersects.

A world frustum's position is defined by the Go that owns it. As the Go moves, the frustum moves with it to keep the Go at its center. Party members are always at the center of the activity in Dungeon Siege, and so they get to determine what needs to be loaded and made active. In this way, the player should never see the edge of the world. The coordinate system only changes when a party member crosses a node boundary, changing the target node, which causes space to be rebuilt. Given player character movement speed, and average node size, the coordinate frame will usually change every few seconds.

In single player, only one frustum is considered "active", and only the Go's contained in it are updated. This keeps the action where the camera is currently viewing it, and avoids confusion from some far-off party member being attacked while the player can't see it. It also helps out CPU load considerably. In multiplayer, all frustums are active and updating simultaneously, which is why an 8-player game requires a pretty hefty server to run it if everyone decides to play in totally separate areas of the world (hint: don't do this).

Great World Detail

The game world was sparsely populated until the artists started experimenting with building high density forests, just to see if it would work. The results were so good that highly detailed, lush environments became the new art style for the game, eventually becoming one of the defining features that we believe set us apart from competing games. With this shift, the engine started getting used at and beyond its full capacity, and engineering constantly struggled to keep up with optimization support.

We were able to get away with this art style because our world frustum is very tiny compared to the size of a level in other games. This is a significant advantage, and it meant we were able to crank the detail sky-high on texture and object density, simply because we didn't have to worry about fitting an entire level into memory at once. A typical frustum worth of the world in Dungeon Siege is well over 1000 Go's in memory at once (several hundred are usually visible depending on the camera angle), with forty megs of textures and a few hundred Siege Nodes loaded. In half a minute of walking, the player could come to a new area with a completely new set of textures and monsters, if the designer wanted (and they actually did this in a few places for dramatic effect).

As the design became more concrete, our designers had to come to grips with the impact of the type and quantity of objects and triggers they were plopping into the world. Even the physical layout of the files on the disk became an issue in keeping the streaming running smoothly without hitches in frame rate. Automatic cache management systems can only do so much, and so we were forced to continually research and add new types of optimizations to keep the game speedy. It was a lot of work, but turned out to be well worth the effort.

War story: one thing we nearly missed during our final profiling was that we had unwittingly become dependent on smaller nodes. These types of nodes were just the "right size" for many of our algorithms, such as the spherical traits-based occupants queries the AI used, or the calculations in the static and dynamic lighting systems. Because these nodes comprised the majority of the game, our profiling and optimizing efforts were focused on them. As a result, Dungeon Siege simply works best with small, chunky nodes (i.e. not long and thin). One region of the game, the Castle Ehb, which had a reputation for being slow, had some very large and oddly shaped nodes in it. Luckily it got added to our profiler tests at the last minute, which showed that these nodes were pathological to many of our algorithms. It highlighted a few important optimizations we could make to lighting and collection management, and we were able to double or triple performance in many areas because of it.

We later discovered a different problem of too *many* nodes. Our multiplayer map had a very large castle of its own with an extreme density of nodes, sometimes 2000 loaded at once, which was about eight times the normal number. This caused the regular space walks that the Siege Engine took to overwhelm the profile, and as a result, that whole area isn't kind to the frame rate. We weren't able to slip in optimizations for this problem in time, and so that remains an area of improvement for the future.

Game Object Implementation

In stark contrast to the terrain engine, the game object system is a complicated, scary, huge piece of heavily mutated code that is currently plotting to take over the world. The dynamic portion of the Go system is composed of two main pieces: the Go's themselves and the Go database (GoDb) that manages them. The Go's are chunks of logic that may update on a simulation tick and process world messages. The GoDb started out as a simple database that created and destroyed Go's, but ended up very large, mostly due to the load-time dependencies of the Go's and our choice of threading model.

About Game Objects and ID's

The required support for multiplayer and continuous world radically affected this game object system. We knew that the Go allocation scheme would need careful attention if it was going to work at all. While the system it was carefully planned, the evolving optimization requirements pushed it to its limits.

Go's may be created in Dungeon Siege one of two ways: either automatically via the world loading process, or directly via a call to create one by game code (e.g. "ready an arrow"). There are three main types of Go's: globals, locals, and clone sources. Globals are created and managed by the server and share a common address space across all machines, locals are created and managed

locally per machine, and clone sources (looking back, a better name for these would have been “prototypes”) are special cache Go’s for making new Go’s, also managed locally. Clone sources also hang around after last touched for an extra 30 seconds to boost the cache hit rate, and can be preloaded to smooth out generators, which spawn Go’s on the main thread. Each of these types is stored in its own database within the GoDb.

Later on, we added new types of Go’s to handle the evolving needs of the game. For example, the Go managing a spell effect would stop working after it passed out of the world (this was mainly an issue with long duration effects), and so we created “omni” Go’s, which are not part of the world, and always active. They are used as manager objects in-game for timers and gameplay tracking systems, as well as the special 3D GUI and character chooser in the front end. Other new types of Go’s we had to create, such as server-only, all-clients, and lodfi, are discussed later in this paper.

Every Go has two types of ID’s for addressing purposes – a Game Object Identifier or Goid, and a Static Content Identifier or Scid. The Goid is assigned at creation time to all Go’s, and is guaranteed unique within a machine. Each type of Go (global, local, clone source) has its own independent range of Goids to use. Because only global Goids use a common address space across all systems, there is special safety code to detect and error when some incorrectly written code tries to send non-global Goids across the network. Global Goids are also structured to permit recursive creation calls to optimize network bandwidth, so that a monster may create its inventory items while the monster itself is being created, without the server needing it to tell it to create each item individually.

The Scid is a map-wide unique ID assigned at Go creation time by Siege Editor for instanced Go’s placed by a designer. A definition block for an instanced Go will be the template name (the type of Go to create), its Scid, its location, and any fields that the designer has overridden from the template, such as scale or attack range. Go’s that are generated at runtime in the game get assigned a special constant “spawned” Scid. Examples of these types of Go’s may be anything from arrows shot at monsters to effects managers to creatures spawned from monster generators. Various functions are available to quickly resolve a Go to or from its Scid or Goid address.

Evicting Go’s

With 60,000 Go’s placed per map, and many times that number generated during gameplay, we have a problem of deciding whether or not a Go should be kept in memory. A non-critical Go such as a tree or an arrow should obviously be deleted when it passes outside of the world frustum and is no longer needed (or more accurately, when the world frustum moves away from the tree or arrow). And an absolutely critical Go such as a player character, or any of the items in their current inventory should obviously be kept in memory at all times. What about other objects, such as doors, dead monsters on the ground, chests, elevator movers, light togglers, and quest completion triggers?

If we keep too many Go’s in memory, then memory usage will spin out of control and, by the end of the game, we’ll spend all our time paging to disk. Save game will become unbearably long and the file size will be enormous. On the other hand, if a certain chest holds the Axe of Kill Kill Carnage as a quest reward, we definitely want to remember that it was already opened and taken, otherwise the player could easily exploit this by going back again and again until the whole party was outfitted,

throwing off the game balance. And it's easy to imagine a number of gameplay blocking issues if the state of locks on doors and such are not maintained.

There is no single easy solution to this problem, so we implemented a variety of optimizations to tackle it. At first, we had nothing; we simply kept everything in memory. A quick calculation of the number of objects we'd have in memory by the end of the game showed us this was a bad idea. Our next idea was to "cache out" objects, keeping them all in memory but reducing each to a small save file. With our particular implementation of the GoDb, only 65000 (2^{16}) simultaneous root Go's could be kept in memory at once, so that idea wouldn't work either, plus even with a small save header per Go we'd still have an enormous memory overhead. We calculated that, no matter what, we'd just have to start deleting things. Not just the obvious things like trees and rocks, but monsters and even magic items, as well. Our final solution consisted of several parts: fluff removal, expiration of non-critical Go's, Scid retirement, and data reduction to 32 bits.

Fluff Removal. This was a relatively easy optimization. We already had a system in place called "lodfi", which stood for "level of detail for items" and was intended for an object detail slider that would cause non-critical game elements (fluff – wheelbarrows, trees, bushes, etc.) to not get loaded at all. This was an optimization for low end systems to trade off prettiness for performance. Every object in the game had a lodfi range that would determine whether or not it was a candidate for skipping during the load process, and we did a few passes through the templates in the content database to set lodfi ranges on decorations and such. Conveniently, this lodfi setting was also an easy way to say "this is non-interactive game content" and lodfi-tagged Go's could be optimized out of all sorts of things. They didn't need to go in the save game, they would never cross the network and so could be constructed as locals, the AI queries can completely ignore them, and most importantly, they could be deleted almost immediately after leaving the world frustum. The majority of Go's with Scids in the game are lodfi objects, and so this solved perhaps 80% of the runaway memory problem for us.

Expiration. As the party moves through the world, killing and looting everything in sight, a lot of things get dropped on the ground. This includes swords, armor, potions, and other things that the player often chooses to leave behind because it's not as phat as their current gear. We added the ability for a Go to have an "expiration class", which says how long it is permitted to stay around after it leaves the world. After a Go exits the world frustum, a countdown timer starts on it (usually ten to fifteen minutes in duration). If the timer fires and the Go is still outside of the world, it is simply deleted. We used a timer because the player may be near enough to a town to want to run back and pick up some more items to sell off, so those items needed to be maintained longer than half a frustum's width of running. There is also the option to set an expiration class to "never", which is useful for quest items and other objects that should never be deleted.

Scid Retirement. Monsters were the first things to get tested out in the expiration system, and they were also the first things to fail. After deleting a Go, the next time you go through the area where it was placed in the world in the editor, it would get constructed again fresh, which would throw off game balance. So Scid retirement was a way of marking Scids as "never construct this again", and was used to keep monsters from respawning. Later on we made as many one-shot things self-destruct as possible to take advantage of this.

Data Reduction. Even after all of this work, there were still thousands of extra Go's being dragged around in memory by the very end of the game. We still wanted to track whether or not a chest had been opened, or a door was unlocked, but we needed a way to track this information without actually keeping the object in memory to do it. Last minute optimization ScidBits™® was born. This is just a trivial database of 32-bit words that are mapped to Scids as a tight save game method. An object could use bits from its ScidBits to track on/off states. For example, a chest could detect the leaving-world message, then set bit 0 of its ScidBits to say whether or not it had been opened, and then self-destruct without retirement. Later, if that chest had been reloaded, the first thing it would do is check that bit to see if it should animate to the open position and delete its contents. This was the last piece of the puzzle. Eric Tams, our content engineer, applied this simple technique to almost all the interactive content in the game that could handle it. After this, the game's memory usage profile was consistent and smooth from beginning to end.

Once the memory usage profile was consistent, we had to do something about its footprint. The expiration time for Go's was long enough that the animations, textures, and bones of their models would start to pile up in memory fairly quickly. To cut down on this, we added a multi-level purging ability to our models that would do a staged unloading of the resources and temporary storage that are required for rendering. The resources would be brought back into memory on demand.

Continuous Logic

It's a huge conceptual and implementation problem to build an action/adventure game without having self-contained, reasonably sized levels. This also took the team years to fully grok. Any attempt to cover all of the issues we ran into in this small paper would be futile, so instead let's cover the million dollar one.

The core problem that we had so much trouble with is that, with our smoothly continuous world, there are no fixed places in the world to periodically destroy everything in memory and start fresh. There are no standard entry/exit points from which to hang scripted events to initialize or shut down various logical systems for plot advancement, flag checking, etc. There is no single load screen to fill memory with all the objects required for the entire map, or save off the previous map's state for reload on a later visit to that area. In short, not only is the world continuous, but the *logic* must be continuous as well!

About 0.4% of the game's content is in memory at any time, and it is constantly being recycled as the player moves through the world. In addition, Go's are constantly *leaving* the world, which means that if one Go is dependent upon another, it needs to handle the possibility that its target will simply get deleted *at any time*. This can throw carefully scripted trigger sequences off if they are not activated correctly. Say you have a baton-passing scene where, when activated by a player character, non-player character Bob picks up an item and runs to Jane to hand it off. If you were to trigger the sequence, then walk away from it, and Jane were to pass out of the world before Bob gets there, what happens? Suddenly Bob can no longer find a path to Jane, and in a few seconds, Bob himself will exit the world. What happens then?

Having objects constantly entering and leaving the world is most often a problem at frustum boundaries. On a frame tick, objects may not only change their coordinate systems, but they may completely cease to exist! Building a general solution to scenarios like these nearly drove us mad. We ended up having to modify our systems to become extremely tolerant of problems and

unexpected conditions. The game now contains a number of “self-healing” features. For example, the Go that modifies a light to make it flicker checks to make sure that the light it is modifying exists before it starts up, and will wait if necessary if it’s not. And an AI job such as “attack that Krug” will intelligently abort and choose a new target if its destination suddenly disappears. Many systems in the game had no choice but to be tolerant of having the rug pulled out from under them. And then, of course, adding the tolerance required for multiplayer made all previous efforts appear to have been a cakewalk in comparison.

Multiplayer

In multiplayer, having every single machine completely in sync with the server is not possible due to network bandwidth requirements (even on a LAN it would hurt), and so each client machine only “knows about” the Go’s in the frustum owned by its player’s hero. The server manages all of the frustums at once, giving it a superset of the entire game, and is responsible for telling each client to create the global Go’s as they move through the world.

The server has to keep track of which client machines have which Go’s constructed, so that it can issue correct create and destroy commands as gameplay proceeds. Every non-Omni Go has a SiegePos saying where it is located, which implies that each belongs to a node. Nodes have membership bits (cached by each Go) saying which frustum they are part of. The GoDb on the server uses these bits to determine which players in a multiplayer game can see the Go, which in turn is used to dispatch network commands to the correct clients as an additional bandwidth optimization.

We ran into a problem right away with this system: what happens when a client runs into a monster or some other item that has been modified (attacked, killed, flipped, spindled, mutilated, etc.) by another player? Leaving it in its default state would not work, because it would look incorrect, have the wrong name, be in the wrong pose, or even crash the game in certain cases.

To solve this, we had the server keep track of when Go’s are “dirty” and if so, construct a delta packet (we call these “packed bits xfers” internally) and send it along with the construction order to the client as it is spawned on demand into the client’s world frustum. The delta packet contains information about how the Go was different since it was first created in its “const” form, and may include things like its animation pose, spells cast on it, current life and mana levels, and if it is a potion, how full it is (in support of our high-tech Potion Sipping Technology™©). Most of the time, the delta packet is very small, because the players are running through pristine wilderness, or the objects they have interacted with have self-destructed as per our memory optimizations. We also carefully bit-pack the packet, use some LRU position caches, and on top of the network compression layer, this doesn’t really hit the bandwidth too much, all things considered.

Multiplayer adds an entirely new dimension to our optimizations, requiring new types of support Go’s to be created. Server-only Go’s are special globals that only need appear on the server, such as triggers driving quests, to avoid eating up the network bandwidth. And “all-clients” Go’s are objects that exist on all client machines in a multiplayer session, to avoid delta syncing problems with the complicated player character Go’s as they pass in and out of each other’s world frustums.

War story: one major problem we discovered towards the end was the expense of stores. A “store” is a Go that owns a lot of inventory for sale, such as weapons, potions, spells, etc. The way we

implemented the system was to have a store construct its entire inventory immediately upon creation (actually, this just leveraged the same code that monsters use to generate their own inventory). This worked well early on, but as the game got tuned, the stores grew in size to have hundreds of items in them. Worse yet, most of them were located in towns, which already had very dense object placement. The dirty Go algorithm completely broke down the instant someone bought anything from the store, because its state had changed, and so after that every item had to be constructed individually for any machine that moved its frustum to contain that store's node. In practice, this was a brutal (but ultimately shippable) drain on the network, hard drive, and CPU, and by the time we noticed its severity, it was too late to fix. Stores are a good example of the engine being pushed almost beyond its limits, and the system is in the crosshairs to be fixed in the future.

World Streamer Implementation

The world streamer (internally called the Siege Loader) is a secondary thread that is constantly pulling in new data in the background, usually from requests made by the primary thread. This system went through several total rewrites before coming to rest at its current implementation. At one point in its development, the loader was doing walks of the frustum on the second thread, and loading nodes directly from there. This was far too complicated to manage because of the complex interaction with the main thread for rendering and space transformation, and so this was simplified to work as it does today. Unfortunately, today's model is still fairly complicated. The terrain side of things is clean, but the Go loader needs a good rewrite.

In Dungeon Siege, node loading is a three-part staged process that may happen immediately or take several seconds to finish, depending on the load on the system. First, the node is "preloaded": its storage is allocated, the physical mesh is loaded, its properties get set, and then the node is hooked into the world's node graph. Next, the node is added to the world frustum, which triggers Go loading for all of the Scids it contains. Finally, the node may potentially be "upgraded", if it is part of a frustum belonging to the screen player, which ends up issuing texture load work orders for the node. The upgrading step is an important multiplayer optimization we added for the server (we call this "slim loading"), and avoids loading textures for nodes that will never render.

The preloading, texture loading, and Go loading portions of this process are expensive, and are usually done on the secondary thread. At any point in this load pipeline, which can take some time to complete, the player may decide to go in the other direction, which ends up deleting the node. This will cancel any outstanding requests and "roll back" the load process. Note that the load process is actually quite a bit more detailed than this brief overview, because of many optimizations and special cases we had to add.

Getting the world streamer to fill orders on a second thread is one thing, but balancing it properly is an entirely different problem. The secondary thread has limited resources, in order to maintain a smooth and continuous frame rate – we tried to avoid it going above 20% CPU. We had intended for the thread to spend most of its time blocking on DMA'd disk I/O, but the addition of a zlib-compressed file system dashed those hopes. And in addition to eating up the CPU, the thread can also grab locks to certain key structures as it runs, so that even if CPU usage is low, the main thread may spend time waiting for those locks to be freed up, which causes hitches in the frame rate.

We kept CPU usage balanced by limiting the rate at which it filled work orders. Every work order has a destination time that it must be completed by. This is used as a type of priority system to sort work orders and space them out to smooth the load when possible. We also spent a considerable amount of time trying to keep the contention between the main thread and the streamer thread to a bare minimum, and even had to invent a new type of critical section (the hyper-fast contention-resistant "read-write critical") and many types of thread lock profiling as part of our optimization efforts.

Over time, the thread model slowly became more and more complex. Go creation involves nearly every system in the game, and because this runs on the second thread, most of the game had to be made thread-safe. This reduced performance considerably (5% skimmed right off the top just to serialize the heap) and created a fountain of bugs. We became adept at hunting really bizarre thread contention and protection bugs (*note: this is not something we are particularly proud of*).

Looking to the future, a better model may be to break the Go creation down into more basic stages, and avoid running any game logic at all on the secondary thread, only using it to serve resource load requests. We intend to research this option as a way to reduce complexity in the engine, which is currently one of our biggest challenges. When you get right down to it, Dungeon Siege is really just an exercise in cache design and optimization. Even the caches often have caches. It hurts to think about it sometimes.

Odds and Ends

These are some short but sweet war stories that happened during the development of our continuous world.

Late in the process we realized that we needed some method of near-instantaneous travel from one part of the map to another for multiplayer. Our engine was simply not designed for this type of thing. While teleporting anywhere in the map is a near trivial operation (and a useful and fun feature of our development build) it causes the program to freeze while we force-load the target of the teleport (a load screen!). This is probably fine in single player, but the server in multiplayer cannot afford to do this because one player teleporting would temporarily lock up the simulation on the server for the rest of the players. We hacked around the problem with frightening elevator and frustum size tricks, and the designers made it all part of the game's story (the H.U.B. system).

Another problem with lacking teleportation was what to do when a player got killed in multiplayer. Other games simply respawn the player at some fixed starting point, perhaps making them pay a price of money or weapons for the privilege. Doing this in Dungeon Siege would effectively require teleportation, and this time we couldn't work around it with an elevator trick. Instead, one of our programmers, Chad Queen, invented the penalty box style "ghost" system, that bypasses the problem entirely, and is more fun and more immersive than the old teleport-respawn trick!

Permitting the party to split up had many complications – some geometry broke when characters were placed a certain way because of overlapping issues between regions that would not normally show up with a single character. Also, because multiple party members could be anywhere in the world at the same time, and the player could switch among them instantaneously, we had to treat the mood, music, and weather as attributes of each party member, rather than globals.

One unexpected and severe problem we ran into was FPU precision, but completely unrelated to terrain! Unfortunately, this was only discovered after shipping through some painstaking debugging efforts by Jessica Tams. Time in the game was tracked using floats representing seconds. At high frame rates after around 30 hours of play, a variety of artifacts showed up in the game as time started to quantize, becoming progressively worse. Because the world is continuous, time must be continuous as well, and cannot ever be reset at any point in the game. We solved this in the 1.1 release of the engine by seeking out all the places time was used in the game and converting those sections to temporarily set the FPU to run at full precision during those calculations.

One feature we always wanted was the ability to be able to reach outside of the world frustum, usually to get orientations and distances. For example, it would be convenient, if two party members were separated enough that their world frustums did not touch, to be able to have an arrow onscreen that points from one to the other, so they are easier to find. Unfortunately, because there is *nothing* between the two frustums at that point, and the only way they could be connected is with a completely blind walk (and this would not guarantee to work!), such a question of "where is Sally relative to Jose" is simply impossible to answer. We are researching creative ways around this, by popular demand.

In addition, the node-relative nature of the world meant that those distances and orientations may not make sense anyway because the world doesn't necessary follow traditional Euclidian notions of space! The world can, and often does, intersect itself, and end up bent like a pretzel. Two regions can completely overlap, where one is faded out and the other is visible. The designers took full advantage of this in the multiplayer map, creating an "endless desert", which is literally endless in one of the directions because it is stitched back on itself like a loop. In more practical terms, this ability gave the designers a lot of flexibility in layout out their worlds, because they didn't have to worry about making distances match up, or even make sense. The player would never notice, but it does make building overview maps (say for a strategy guide) a little difficult.

Further Reading

The article "Postmortem: Gas Powered Games' *Dungeon Siege*", by Bartosz Kijanka, redefines the term "crunch mode", and is available on Gamasutra at http://www.gamasutra.com/features/20021218/kijanka_01.htm.

The GDC 2002 talk "A Data-Driven Game Object System", by Scott Bilas, covers the data-driven aspects of the Go system in detail, and is available at http://scottbilas.com.

[Special thanks to Mike Biddlecombe for his many contributions to this paper, and to Chris Taylor and the Gas Powered team for their careful reviews.]