Software Scalability Engineering for Distributed Virtual Environments

H. Lally Singh[*] Google and Virginia Tech, *Member, IEEE*

Denis Gračanin[†] Virginia Tech, *Senior Member, IEEE*

Krešimir Matković[‡] VRVis Austria, *Member, IEEE CS*

ABSTRACT

Distributed Virtual Environments (DVEs) must continue to perform well as users are added. However, DVE performance can become sensitive to user behavior in many ways: their actions, their positions, and even the direction that they look. Two virtual words may be similar in terms of user experience, task efficiency, immersion, and even aesthetics. However they may exhibit substantially different performance when many users are logged in. We discuss an approach — Software Scalability Engineering (SSE) — that uses load simulation and iterative modeling to locate causes of undesirable performance, experiment with changes, and verify improvements to DVE systems. Presented here is a case study of using the approach to determine the primary bottleneck of the Torque engine. Once that bottleneck is identified, we continue to use SSE to determine the primary cause of the bottleneck, and the primary controlling factor for that user. SSE allows us to look at changing both the engine software and the virtual world to improve performance.

1 INTRODUCTION AND RELATED WORK

Distributed Virtual Environments (DVEs) are a complex amalgamation of many systems with a critical sensitivity to their performance. A primary challenge of DVEs is scalability — supporting many simultaneous users. The system's scalability is sensitive to performance — as performance degrades, users will stop using the system. Unlike most systems scalability problems, users will stop accepting the system before the system will stop accepting users.

Consequently, performance management is important during a DVE's development. With the complexity of the systems involved: the physical simulation, the network synchronization, the virtual terrain, tools, avatars, vehicles, and even the maximum velocities of each can become performance factors. As some of these elements can be difficult to change late in the development cycle, starting performance management early in the development cycle can reduce the total work.

In this paper, we will cover the SSE process, and show its application to a large commercial game engine. We will cover its application in identifying the bottleneck resource, the software component taking up the majority of that bottleneck, the controlling factor of that component's performance, and two different ways to enhance it.

Like many systems currently deployed, this will be a client/server system. We will be studying the server and thusly some rather interesting parts of DVE systems (e.g. rendering) won't be analyzed. SSE does not require, or even prefer, a client/server system — it will work just as well with any topology selected.

Distributed Virtual Environments come in many forms. Oliveira et. al. [9] provide a broad domain model for analyzing them. Three major categories can be formed from the network topology. A common architecture has a single server performing all synchronization work, with all clients connected to it. Systems like this include Torque [5] and Quake [17] (releases I-IV). On the other end, there are a variety of Peer to Peer (P2P) system architectures, where more than one computer executes some part of the synchronization work. As an example, the Hydra [4] system has each user's computer act as both a server to some part of the virtual world, and a client for the user. A single "global tracker" is run to map regions of virtual worlds to host IP and port numbers. They mention it being alternatively implementable as a Distributed Hash Table (DHT). In these distributed environments, additional concerns such as load balancing [11] occur. In that work, the spatial region to host mapping is determined by cluster analysis of user positions.

The MASSIVE [6] system was a mostly peer to peer system with a few centralized *aura* — region of interest — managers for each part of the virtual environment. For hybrid P2P systems with a set of servers and a set of clients, Morillo et al. [8] studied a "quality function" for assigning clients to servers. They find little correlation between that function and average system response. Instead, they propose a mechanism to prevent CPU saturation, as that tended to be the largest factor in system performance.

Independent of the network topology, many DVE systems are using multiple threads of execution. Mönkkönen [7] provides an introduction to how the work can be split up across multiple processors on a single system. Three models are presented. First, physics can be run in a separate thread and synchronized per cycle. Second, logic, physics, and rendering are run in independent loops, in separate threads. Finally, the logic and physical work for each object is assigned a separate thread. Abdelkhalek and Bilas [1] modify Quake to use multiple threads. At eight threads, they were able to raise the player capacity by 25%, but were spending 70% of their CPU time in lock contention and waiting. Zyulkyarov et al. [19] use Software Transactional Memory to reduce contention. They peak at less than 26% transaction aborts, but the prototype compiler caused too many problems to give results useful for comparison.

System performance affects user task performance. For a traditional first-person shooter game, Quax et al. [10] provide an analysis of the effects of latency and jitter on user performance, as measured by the in-game score in Unreal Tournament 2003. Using a router that simulated specific latencies and jitters, they found that users had noticeable impairment when round-trip time (RTT) latency surpassed 60 ms.

Watson et al. [18] studied *system responsiveness* and its effect on user performance. System responsiveness is the time-to-feedback from any user action and the response being perceivable by the user in the virtual world. Originally done as a study based primarily on local systems with single simulators, it studied the effects of display lag on users executing tasks. Among the conclusions were that sensitivity to system responsiveness was task-dependent. Unfortunately, no similar study was done for the distributed case, where local simulation continues providing the user fast feedback on their actions, but may have their results changed (or undone) upon reception of new data from remote sources.

We present a methodology derived from Software Performance Engineering (SPE), an iterative, model-based software engineering process. Smith and Williams [16] describe it in detail. Our own methodology, Software Scalability Engineering (SSE) is a derivative of it. SPE is a general-purpose process applicable to many systems, but does not directly address the effects of complex human behavior or static assets (e.g. 3D models) of the system being built.

[*]lally@vt.edu
[†]gracanin@vt.edu
[‡]matkovic@vrvis.at

Abdelkhalek et al. [2] analyzed the performance of Quake using an analysis similar to benchmarking Online Transaction Processing (OLTP) systems. Using CPU-level events, and built-in system instrumentation, they replayed individual user-session recordings for load simulation.

Our SSE process was introduced in [14]. It was elaborated in [13], with a very small — only 5,000 lines of code — two player system. In that work, we determined the cost, in simulation time, of adding additional objects to the virtual environment.

2 Challenges in Scaling Distributed Virtual Environments

Distributed Virtual Environment (DVE) systems are large and complex systems. The standard that they must meet — the quality of experience for each user — is also complex. When trying to make DVEs able to support a large number of users, that standard must be kept as users are added. Unfortunately its elements: the responsiveness of the system, the accuracy of the environment, and its consistency of events, each have many contributing factors. Additionally, the contributing factors can often depend on some unique properties of individual DVE components.

The state space of a DVE system includes the positions of all objects, players, buildings, and terrain. Their individual state spaces can include their geometry, position, orientation, health, sounds emitted, transient graphical states (such as burned or glowing), blast marks, or task-related modes. In aggregate, the synchronization state space includes the last-known state of every object in every host's memory. Depending on the synchronization mechanisms used, inter-object visibility can be a major factor in the state-update frequency between hosts. Users occlude one another, and can constrain the amount of data they each have to receive in order to maintain a live view of the environment.

Additionally, the performance characteristics of a DVE system can be quite sensitive to changes in that state space. A cluster of objects in one area can cause one part of a scene-graph's data structure to contain substantially more objects than average, causing disproportionate simulation times for that region. Changes to small numbers of otherwise–insignificant objects (e.g. foliage) can substantially alter the synchronization load on the system if the inter–visibility of other objects is altered. Atop of all this, changes in human behavior can be very hard to predict, and result in substantial changes to the resulting load applied to the system.

Finally, the resulting level of reality — the Quality of Experience (QoE) perceived by the users is sensitive to the performance of the system. First, the longer the interval between the user's host sending an update and receiving a world-state that contains the results of that update directly results in visible reaction lag from the DVE. Second, the frequency between updates directly affects error from client-side prediction, such as dead–reckoning [3]. In systems with parallel computation, such as multi–threaded DVE servers on multi–core machines, different rates of simulation across the virtual environment can have substantial fairness repercussions. Finally, the simulation rate on the server — and the resulting receive rate of world-state updates by a user's host — can directly affect their performance on specific tasks [12].

The individual problems above: the size of the state space, the sensitivity of performance to relatively small details in the system, and the complexity of managing the Quality of Experience, are inherent elements of a DVE's complexity. Unfortunately, the elements also compose into further complexity. The interdependencies between the geometric assets of the virtual world: terrain, buildings, objects, foliage, etc., can affect user behavior. The elements can affect the system state space, resulting in changes to the resulting performance. During the development of a DVE system, these changes can be frequent. Unfortunately, the resulting changes in system performance — and the ability of the system to satisfy its performance requirements — can be just as variable.

To address the relationships, we will have to understand changes in one component and invalidate existing understanding of others. For example, a change in the virtual environment can change how often a software component is run or instantiated.

If these components were small, this problem may be directly manageable. Sadly, they're quite large. For the Torque engine we discuss later, the total source code — headers, source, and parser specifications — total nearly 350 thousand lines of source code. The virtual environment specification — for a small environment with a handful of tiny buildings — is a half-gigabyte of data. The scripted data and behaviors in that level, supporting a crossbow, a player that moves in a loop, and the basic user interface — is another 25 thousand lines of source code. Common data and behaviors, in a separate library used by this level, are another twelve thousand lines of source code. To address size, we have to be able to quickly classify the system components' impacts on the relevant elements of QoE.

The element of user behavior is the final part of the challenge we can address. User behavior is sensitive to both the virtual environment and QoE, and is a large new factor all by itself. The places the users go in the virtual environment, how those movements cluster, and all their actions can all affect the load on the underlying DVE system. Additionally, these behaviors can be complex and aggregate — users may act together, against each other, and involve complex strategies that can result in modal load[1]. To address user behavior, we have to sample it from live experiments. The process we envision uses dependencies and invalidation, quickly classifies component impact on QoE, and includes user behavioral samples from live experiments.

3 Software Scalability Engineering

Our methodology is strongly derived from Software Performance Engineering [16], tuned for an instrumentation–driven cycle with a focus on the construction of DVE systems. Denoted Software Scalability Engineering (SSE), it attempts to assimilate the effects of human behavior and static assets into the simulate–analyze–evaluate iteration.

Specific focus is given to the distinguishing traits of DVE systems: (1) when the load is steady, the system runs in a steady-state, continuous form; (2) performance of the system is expected to vary with available resources, gracefully adjusting as they change; and (3) large components of the system load are determined by both the human behavior in its players and the system-specific geometric assets created to populate the scene graph.

The resulting process is shown in Fig. 1.

In the process, we typically identify key resources that are areas of concern in the effective performance of the system. The first includes the available CPU capacity, as a rate consumed. Network bandwidth and memory may also be concerns. Developers may choose to focus on other resources that become scarce at runtime, such as dedicated hardware (e.g. general–purpose GPU units). If they have an accounting method established, the majority of resources should be modeled with this method.

We define a DVE's scalability as the relationship between the system's resources and the DVE's effective capacity. This latter value is the maximum number of users that (1) can fit into the system and have it run without crashing and (2) continue to find the experience sufficient to meet their goals. This latter clause requires that system behavior and performance continue to be acceptable to users, as a DVE may well be able to process user logins well beyond the point where it can provide reasonable service.

[1]That is, the load on the system software is in different phases — perhaps one each for a group approaching a position, attacking it, and then moving on.

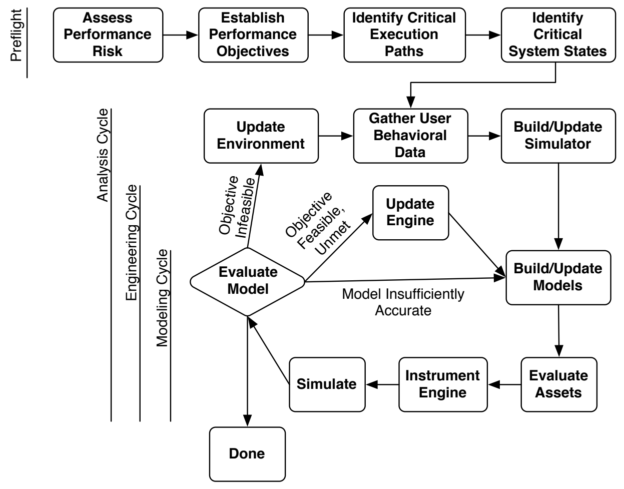

Figure 1: Analysis and modeling procedure.

The phases of the SSE process are summarized below and summarized in Figure 1. Interested readers are directed to more thorough descriptions in our prior discussions of it [13, 14]. Space in this paper is reserved for an in–depth case study with Torque.

3.1 Preflight

When beginning the scalability engineering process, or the engineering project as a whole, some initial work up-front is needed. Determine: what levels of performance and scalability are desired, and how much engineering effort they are worth; which software components and data structures compose the performance and scale-critical paths through the software; the critical variables that determine load, and what performance envelope we want the system to exhibit under that load.

The overall process is intended to be incremental, with initial phases using very coarse models, instrumentation, and simulators at the start. We heave four stages in preflight. In *Assess Performance Risk,* we determine what levels of scalabilty are desired, and how much engineering effort they are worth. In *Establish Performance Objectives,* with the engineering budget known, we determine what levels of performance are desired. In *Identify Critical Execution Paths,* we identify the critical paths in code that drive performance. Finally, in *Identify Critical System States,* we identify the critical variables and data structures involved in the execution paths we identified earlier.

3.2 The Outermost Cycle: Analysis

The analysis phase is the head of the iteration loop. Initially, it is entered after preflight, and re–entered after changes to the virtual environment or substantial changes to the DVE software. First, gather human usage data to determine how users commonly end up using the system. A two-phase analysis is done to analyze the gathered data. First, each recording is played back individually. At a sample interval relevant for the DVE, categorize the user current behavior. Next, combine all the frequencies of all the categories. Finally, modify or re-implement the client software to behave representatively like the observations.

Upon re–entry of the Analysis cycle, a step is prepended. We adjust the virtual environment to allow more desirable performance. After the environment has been updated, new behavioral data is required, and the analysis cycle begins its next iteration. During normal engineering, an occasional execution of the Analysis cycle, even if no single change seems to require it, is recommended. Small changes can sometimes be surprising in their affects on user behavior.

3.3 The Middle: Engineering

If the engine does not perform sufficiently well, update the engine and restart. Possibly modify the DVE system software as seen fit. The user behavior shouldn't normally change for changes to the engine, avoiding the need to update the user behavior executed by the simulator. DVE system engineers may choose to do so if they believe that the changes are sufficiently substantial. When complete, the engineering cycle restarts at the "Build/Update Model" stage.

If the DVE itself is infeasible with current technology, then modify it in the "Update Engine" phase and go back to "Gather Behavior Data" for the new DVE. This closes the Analysis Cycle. If the models are sufficiently accurate and show a sufficiently–performing DVE, then no additional cycle is needed and one may consider the process done. If there is any doubt, a new iteration through the Analysis cycle (without modifying the level) may be used to verify that assumptions on user behavior still hold. Simplify the instrumentation to reduce overhead, but leave enough to allow observation of the system in production.

3.4 The Innermost Cycle: Modeling

We use an incremental analyze–and–simulate method to both let us scale the effort, and focus it on the most quantitatively significant parts of the system. When we find unacceptable resource usage, we construct a more detailed model. We construct or update the load, resource requirement, and performance models. Then we analyze the structure of the geometric assets, to determine their effects on load.

First, place instrumentation into the engine to validate and calibrate the models. Then, run the load simulator against the engine, enable instrumentation, and record data. Finally, looking at the instrumentation data, determine if the model gives enough accuracy, if the engine can provide the performance required, or if it is even objectively feasible to build such a DVE with the performance requirements. If the model is insufficiently accurate, go back to the "Build/Update Models" phase. This back-arc completes the Modeling Cycle.

3.5 Load Simulation

The load simulator is based on observed human behavior collected through a controlled human experiment. Users are placed into a facility with computers running the DVE. Typically this will be a computer lab. Each computer records every update sent between each client machine and any intermediaries. Additionally, the computers that execute system–wide services (e.g. a server running the canonical simulation) will have instrumentation attached for their resource utilization. With a representative group of users, in a representative setting, the data collected should be suitable for a load simulator. More than one phase of the study should be executed, with differing numbers of users.

The collected data is then analyzed to determine behaviors, tactics, important locations in the virtual world, and strategies. The observed behaviors are necessarily terrain-specific, and attempting to extrapolate more general behavioral rules would be unsuitable.

Each behavior, tactic, and strategy is converted into a set of software routines. Once written, the software routines are integrated into a copy of the DVE software that runs on a user's machine. It synthesizes avatar movements and actions using the local scene graph as its knowledge base for the current virtual environment state. For example, it may identify an opponent in the scene graph, notice that it has a clear shot, and fire at the opponent.

Balancing the differences between the capabilities of AI systems and human participants is a key concern in ensuring realistic load simulation. The problem has been a concern in game development for some time. In our own work, we use general assumptions about the virtual environment to guide strategy — e.g. get near the center of town to find opponents to attack — and simple scene analysis to guide tactics. In terms of aiming, we balance the human's ability to predict their target's movement with the AI's ability to exactly aim at the target's current direction.

3.6 Iteration Strategy

Preflight is critical for determining both the goals of the work and the space in the program, in terms of code and data, that the work will cover. It will have the most substantial impact on the rest of the SSE process, as it will form both the basis of the models and the criteria used to evaluate a simulation's results. For most cases with client/server systems, we expect preflight to identify the server-side code alone as origin of the critical code path and data.

After executing preflight and having constructed the load simulation, the first iterations of SSE should be small and exploratory. These early iterations should be identifying the major components of system performance. The instrumentation should quickly show where the resources are being expended, at least in a course-grained fashion. Further iterations construct a more detailed understanding of those major components, and explore their controlling factors. Final iterations experiment with altering the system at a software or virtual-environment level to achieve the goals determined in preflight.

In the Torque case study below, we start with `top(1)` to identify the CPU as the primary component of system performance — as the bottleneck, additional availability of other resources would have little effect. Through additional iterations, the primary user of the CPU is found to be player simulation, then the collision detection part of that simulation. The player position, in relation to other players, is the primary controlling factor over collision detection time, with a polynomial relationship. We consider moving player simulation to other threads, but decide to alter the virtual environment to cause fewer simultaneous collisions in the same area.

4 SSE WITH THE TORQUE ENGINE

We will trace through the entire SSE process with the Torque engine. Torque is a single–threaded engine with input processing, simulation, and output processing all branching off of the same primary loop.

However, the player's clustering in the level is quite relevant. Figure 4 shows the distribution of players across the virtual environment. The original level that ships with Torque is listed as "A. Original." As discussed later, this clustering correlates to the collisions that will occur between users and anything they fire at one another.

4.1 Preflight

For Torque, we don't have a risk model or specific scalability objectives. Instead, we have the general desire to understand the engine's performance and what its contributory factors are. The engine was licensed and now it's time to determine its performance. When we understand the current performance and the factors that control it, we can make determinations for how we want to realize the DVE we want to build.

We'll start with a basic performance-factor analysis. With DVEs, performance is a critical factor that we shouldn't allow to surprise us later in the development process when substantial resources have already been invested.

The critical paths for our example are the core loop of the engine on the server.Input from the network is processed, as is a 31.25 Hz simulation cycle, and a 10 Hz send-updates cycle. Our critical state is the scene–graph. If the network I/O components were under consideration, the per-connection data structures would also be considered critical.

4.2 Initial Cycle

The initial cycle is a pass through the analysis, engineering, and modeling cycles. We start with building a load simulator, and a very initial model. A small trick for quickly determining bottleneck resources is discussed.

4.2.1 User Behavior

Modeling the general behavior of human participants is clearly infeasible in the general case, but it *can* be possible within the limited scope of how the behavior affects load factors on the system. For Torque, we have a fairly simple version of the problem. The players are all equals, and fight each other. Group dynamics, if present, seem to be lost in the normal variation found during user study. The data came from a pair of user studies. Each study was a one–hour session of college students playing the game against each other.

With it, we took recordings of each user and analyzed them separately. Each recording was played back individually, and the user's behavior noted in sub-minute intervals. As the game was very simple — one weapon, small town, and only kills counted for points — the number of tactics people used proved small. Recording each tactic used, and how often, we built a characterization of their activity.

4.2.2 Load Simulation

To convert the model into load, an AI-driven synthetic player is created. It uses an internal map of the level, and the current location of it's own avatar and other players nearby as its knowledge base for action. We intend for each instance of this modified client to simulate the load of a single human player. It executes the behaviors in the distribution observed, using just enough logic to move, select targets, aim, chase, and fire. The virtual world used is almost completely open, smooth hills.A table of way–points and direct linear motion towards them was used for navigation. Using this list of tactics, and distribution of their usage, we build a simulator. More detail on load simulation construction is available in [13] and [14].

To provide equivalent load of some number N users, we run N copies of the load simulator simultaneously. We run them on one or more separate machines from the Torque server to prevent CPU starvation on the server.

4.2.3 Model Construction

We mentioned a little trick earlier: we run a quick load simulation to observe the system running, with no instrumentation or assets analyzed. In terms of SSE, this is a run through the modeling cycle (with some phases skipped) to determine the current bottleneck(s). The instrumentation that the operating system provides — bandwidth counters and `top(1)` — provide sufficient instrumentation for a first-run modeling pass.

Even at high load, Torque does not use a fraction of the memory or network bandwidth available on the server — the available technology and reduced costs have significantly improved their situation since its initial release in the late 1990s. However `top(1)` does reveal substantial CPU usage. Single–core performance has not kept up with Torque's needs as well as memory and network availability. It only uses one CPU, but that single core does get substantial use. Concerns for growing scaling the DVE rest on single–core performance.

So, we decide to focus on CPU usage on the critical code paths in the system. This includes all items dispatched from the primary loop in Section 4.1: input, simulation, output. We'll start by instrumenting the total time spent in each, and the total time spent in each core loop iteration.

4.2.4 Asset Analysis

Currently, our model is rather simple: the CPU is dominant, that's all we know. We don't have any use for any metrics to collect from the graphical assets. In our current Torque setting, we only have one geometric model for the player, one for the projectile, and one level. While we may find some other discriminating features later on, we don't have anything to measure right now.

4.2.5 Engine Instrumentation

As mentioned earlier, we study the primary loop in the software, and the three top–level elements it invokes: the input processing, simulation, and output routines. Initially, high–resolution timestamps for the start and end of the loop and all three elements are recorded. We use our own `ppt` [15] tool, designed exactly for this type of work.

4.2.6 Simulation

Our simulation setup is quite simple. We use Virtual Machine (VM) instances for clients, and one more for the server. The VMs are hosted on Amazon's Elastic Compute Cloud (EC2), each running "small" instances with a single 1 GHz CPU (1 EC2 Compute Unit) and 1.7GB of RAM. While we have not been able to ever get a guarantee from Amazon about how to get the server instance on a separate physical machine than the clients, we also run a lot of simulations with stable results between them— it's unlikely that all the simulations ran with the clients (often just 3-6 instances) and server together.

Each instance is a copy of a pre-configured system with the engine and level already installed. We run the server first. It indicates, at roughly six–second intervals, how many users are logged in. Clients start up and launch load simulator instances that connect to the server.

When the server indicates that it has reached a few (e.g. three to five) clients, we have `ppt` "attach" (that is, begin listening) to the server and save the records to disk.

In early iterations, a simple function profiler such as `gprof` is perfectly acceptable to identify major users of the CPU. However, profilers are discouraged for use as a primary source of instrumentation over the run of the SSE process. They require a substantial understanding of the code base to be able to interpret their output. For detailed instrumentation, the collection overhead is substantial, and can take substantial effort to analyze.

Additionally, they have a tendency to encourage "whack-a-mole" optimization, where the first entry in the list of largest CPU users is optimized until it's no longer the first. The cycle repeats with the new topmost entry, until performance is improved. The result is often substantial optimization work with little understanding of the system's overall performance structure.

Hand-instrumentation with a tool like `ppt`, while certainly tedious up–front, provides the opportunity to only pay the runtime instrumentation costs of relevant data, have it saved in easily-analyzable form, and have the instrumentation available to the program itself, for possible runtime adaptation. A handful of simple timestamps quickly identify the largest users of the time, in terms of the system's overall structure and are easily annotated with parameters (e.g. the number of elements checked in a collision search) to identify why the most CPU intensive components are invoked as many times as they are, and why they are the most intensive.

4.3 Modeling Cycles

The overall modeling process is simple: start with an overall metric of system performance, then add measurements in between different stages of work. Measure with different numbers of synthetic players attached. Many stages of work will likely act in either constant or linear time, but some won't. For those that do, their models are constants or linear functions of the number of logged-in users.

For those stages that aren't constant or linear to the number of logged-in users, "drill down" with additional instrumentation and simulation runs. Stop when a satisfactory model is determined. As we are studying the bottleneck identified: single–core CPU time, we track down which code consumes the CPU's time in a non-linear and non-constant fashion. The result is a model in $O(f(N))$ format, where N is the number of logged in users, and $f(N)$ is the amount of time required to execute the code we identify.

4.3.1 Data Collection

As mentioned before, we use the `ppt` [15] tool for data collection. It uses *frames* of data, each atomic, to assemble blocks of related data together. The `ppt` tool makes each frame is made a component of a discriminated union. When `ppt` *attaches* to the program, the program attaches to a `ppt`–created shared memory buffer. The program writes frames into the buffer, and `ppt` reads them out, saving them to disk.

As mentioned before, we have four primary points of instrumentation: the core loop, input processing, physical simulation of the virtual environment, and output. Their frames are `coreloop`, `input`, `simulate`, and `serialize`, respectively.

The `coreloop` represents a cycle of the top-level loop in Torque. Torque server is single-threaded, and works off of a standard event loop.

`input` and `serialize` record the start time, end time, and the client involved. The last is `simulate`, representing the bulk of our data and the focus of our analysis. We create a `simulate` for each simulation step of each object in the virtual world. We start with an object identifier, an integer representing the runtime type of the object, it's position, and the start and end times of the simulation step. This set of data for `simulate` was enough to start with, but proved insufficient when more detailed performance models were needed. Additional iterations added more data to `simulate`.

4.3.2 Iterating the Modeling Cycle

We ran the modeling cycle quite a few times. The `simulate` frame consistently took the vast majority of CPU time during simulations, even at low (12 users) load.

Every object that needed simulation had a single method `processTick(ticks)` that was invoked once per simulation cycle. The `ticks` parameter of `coreloop` indicated how much time to simulate, as a multiple of 32 milliseconds. This value was constant for the entire loop iteration, including all simulation steps executed. `ticks` was adjusted in every core loop iteration to make the interval of time being simulated match the observed interval between calls to the simulation system.

Each individual implementation of `processTick(ticks)` looked at the current state of the scene graph, treating equally the objects who have been simulated for the current cycle, and those not yet. Each moving object would be moved along its velocity vector for the time step and then checked for collisions. Lower values of `ticks` would result in a more precise physical simulation, as the entire scene-graph would have been fresher and less virtual distanced would be traveled between collision checks.

Our instrumentation identified players' avatars as taking the most simulation time, by far. Players themselves took 72% of all CPU time in the system. The rest of time in simulation, another 1.2%, was mostly spent simulating arrows. The remaining 16.2% of CPU time was spent in synchronization with client machines.

4.3.3 Modeling Player Simulation

The player's simulation step has a primary loop that attempted, up to three times, to find an object to collide with, and/or a collision with a rise in floor height that required an upward step. Both could be found simultaneously.

For the first iteration, we split `processTick()` into four parts: a header, a call to the parent types' `processTick()`, a "physics section", and a tail. The header and contents of the "physics section" are described below. The remainder of the method took a small and stable amount of time to run. In the second iteration then split up that "physics section" into parts. The original version of that section was essentially a set of calls to six methods.

A simplified model of the code for `Player::processTick()` is shown below. These values are based off of an $N = 60$ simulation under the original level.

```
Player::processTick() {                 // (Abbreviated Form)
   // Small, Constant Factors                      ~.006  ms
   updateMove()                        //      ~.250  ms
   // Other small constant factors,                ~.0012 ms
   updatePos()                         //     ~1.6 ms
}
```

After that iteration, we found `updateMove()` and `updatePos()` to have nontrivial execution times. They also roughly correspond to the movement and collision-detection phases, respectively. The others totaled to roughly $7.2\mu s$. The `updateMove()`, while nontrivial in execution time, had a very low variance: $0.007ms^2$. The `updatePos()`, however, had a variance of $20.45ms^2$. It will be the focus of the last iteration.

4.3.4 Collision Detection in `updatePos()`

Through algorithm analysis of the code in the method, combined with instrumentation, the collision count was found to be the largest factor of `updatePos()`. The time spent in `Player::updatePos()` is shown on the vertical axis in Figure 3. The values are spread horizontally according to the number of collisions processed in that data point. We have a simple model for `Player::updatePos()`, a polynomial for the number of milliseconds required to run, based on the number of collisions (c):

$$t_{\texttt{updatePos}} = 0.05 + 0.165c^{1.3}(\text{milliseconds}) \qquad (1)$$

The equation is based on a simple curve-fit of the data, initially assuming that the function was in the family $y - y_0 = Ax^B$. We compare it to collected data for varied values of N in Figure 2.

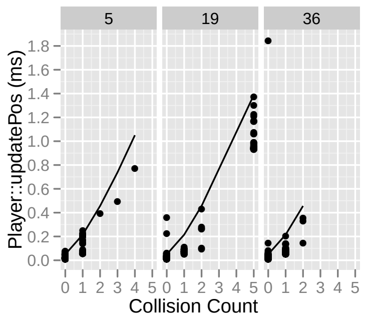

Figure 2: Runtimes for `Player::updatePos()` vs Collision Counts, vs Model of Same (Line), for N=5,16,36

4.3.5 Evaluation

We have a time–exponential step in physical simulation, and it constitutes the majority of where our single core's processing time is going. The key metric we intend to construct is the distribution of runtime of a player's physical simulation, e.g. a single call to `Player::processTick()`.

Equation (1) models the time spent in collision detection, as shown in Figure 3. The values of this factor (c) drive the amount of CPU time that Torque needs to support a virtual environment.

4.4 Engineering: Evaluating Multi-threaded Conversion

To improve the relationship between CPU time availability and our requirement, we can also try to parallelize the engine. With 72% of the simulation time going to one method in one class — `Player::processTick()`, we only need handle this one case. So, how much can we determine from what we've already discovered, for the feasibility, cost, and expected benefits?

Theoretically, we could run player simulation in a secondary thread, and everything else in the primary thread. Primarily, the issue is mutual exclusion. When one thread accesses an object, others must be excluded from modifying them. The amount of work here is doable, but would require a good amount of work. The amount of code that would require modification for adding mutexes is unknown. Additionally, testing for latent synchronization bugs is rather difficult. We may get many of the primary areas of contention, but miss just one or two that can become problematic later. A small change in how the threads are scheduled such as a kernel upgrade or multi-threaded process on the same machine contending for processor time can expose latent synchronization bugs that no amount of load testing in the current configuration would likely find.

With this much work and risk, some consideration is necessary. The work is feasible, but costly and rather risky. The benefits can give us roughly triple the current capacity — assuming we can move almost all of that 72% of player simulation onto other available CPU cores. However, we haven't figured out if we need this much capacity — we haven't designed the DVE yet. As we mentioned in Section 4.1, we haven't determined how we want to realize the DVE yet. Instead of modifying the engine software — a completely new piece of work to resolve a concern we haven't determined is a problem yet — we can build the DVE knowing that collisions cost us dearly.

4.5 Analysis: Experimenting with Level Alterations

The level that we currently have is the one that shipped with Torque, as an example to start with. Instead of creating a new level from scratch, we intend to alter this example level significantly. This way, we always have a workable level, even if it may be schizophrenic mid–development.

While we have a high–level understanding of what we want our artists to make the level into, there is certainly some flexibility in the small– and medium–scale structure of the level.

We also know that the example level does not perform as well as we desire. In user testing, with only fifteen players, the performance is jumpy, and we're hitting the CPU's capacity. Now that we have tabled the idea of enhancing the engine's performance, we should consider altering the level to enhance performance.

Looking at the data we already have, we have positional data in `simulate`. That data is shown in Figure 4 under "A. Original," the primary region in $(x = -100, y = -100)$ to $(x = 750, y = 500)$ is the central inner area of a small village. The data was taken from instrumentation of the player simulations only.

In Figure 4 we have four variations of the virtual environment. There is one particularly dense spot at approximately $(x = 250, y = 200)$, the dark square in Figure 4 (A). Players are apparently often there, and likely more than one at a time.

4.5.1 Update Environment

The level modification was incredibly simple: place another building in the middle of the densest region — the middle of the virtual village. We put a new building there to try and diffuse player positions. Hopefully, users would find other areas more useful, and those areas would have enough space to accommodate them without having users bump into one another. That attempt is shown as the "Modified" version (B).

We are assuming that collisions correlate with spatial density. We can verify that, and may do so if needed. However, it's easier to just try altering the level and seeing if that's enough. SSE has the property that after the load simulation is set up, experiments can be executed for modest effort.

To compare that change versus a random change in the terrain, we make two more altered versions. First, we put another building on the opposite side of the virtual village, opposing the dense spot. This is denoted "Original-2" (C). Next, we take our modified version and put another building next to the one we added in (B). We denote it "Modified-2" (D). A large tower is placed off to the side of the village, blocking a key snipe–point — a hill to the left of the pictured area — from working. It occludes an existing tower. The effect was better than expected. The building density in the village increased enough that the load simulators moved out to a nearby clearing.

4.5.2 Update Simulator

For each variant, the load simulators have the coordinates of the new buildings added to their internal tables, and are otherwise untouched from the original version. We believe that their behaviors are sufficiently abstract that they can continue to accurately simulate human behavior with these changes. If we find useful results in this experiment, another user study may be prudent to adjust the load simulator to validate this belief.

4.5.3 Simulate

In our hypothesis modification ("Modified" (B)), the load simulators routed around the new building. A combination of the provided cover and decreased available area in the region pushed the simulated players out to other areas where they could find other players to interact with. Weaker versions of the same effect were observed in the other variants "Modified-2" and "Original-2."

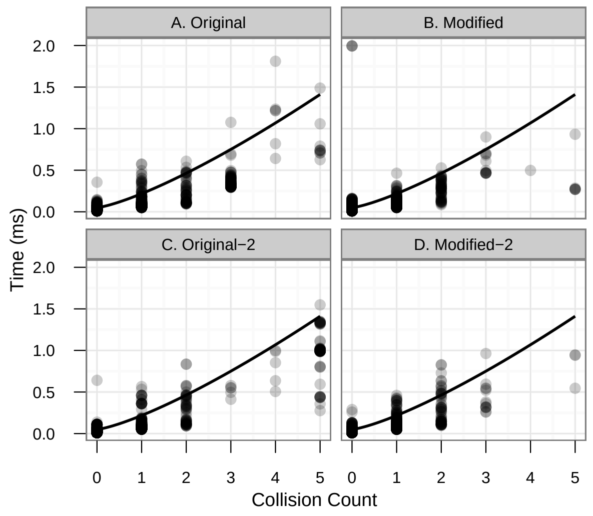

Figure 3: Collision Detection Times vs Collision Counts at N=30, with our Model Overlaid as a Line

Looking at Figure 4, we have quite a variation in user density. The darkest point in the original map is substantially lighter in our "Modified" (B) version. It is still present in our first random variant, "Original-2" (C). The combined intentional and random variant, "Modified-2" (D) has a reduced, but still substantial, dark spot there. In terms of reducing areas of high density, the intentional modification alone seems the most effective. These additional levels give insight into the effectiveness of our experimental technique: putting a building in the densest spot in the level.

Figure 4 shows data acquired from the `ppt` frames to show movements over the instrumented time interval. That way, we can observe the simulation of individual objects in connection with the other performance data we collect. The updated instrumentation was used in a four simulations, each with 60 users. All sub-plots show the data of five–thousand randomly selected player simulation steps. Each step is a single invocation of `Player::processTick()` on an individual object.

If we find this level change beneficial, another user study is worthwhile, to validate the results of the simulator. However, we should first determine if it's worth going that far. Figure 3 shows us how well the model holds up against all four levels.

In terms of overall computational load, we found that the original average time through the main loop was 10.66*ms*, and the modified level 10.36*ms* — a three-percent improvement. Either way, a 90 Hz simulation rate is would be possible, had that average been stable. However, the variance was $43.42ms^2$ — a single standard deviation at over 60% of the mean. While we can handle a reasonable rate most of the time, there are two problems. First, we have to substantially over-allocate CPU power to support the system, to handle transients. Second, the variance is going add jitter to `ticks`. This second factor is rather substantial, as users can easily get surprised with particularly bad simulation when `ticks` is on the high side, and start compensating by distrusting the accuracy of the system simulation. That will detract, substantially, from the user's experience.

In that light, we have been substantially more successful. The variance in simulation time was reduced to $20.65ms^2$. During the longest times through the main loop, peaking 101*ms* in our sample, the engine was falling behind real-time in its update rates to clients, and simulating much longer discrete time steps — higher values of `ticks` — at once.

The other two levels had worse mean runtimes, 10.91*ms* for "Modified-2" and 10.99*ms* for "Original-2." The variances were $86.05ms^2$ and $87.32ms^2$, respectively.

4.5.4 Evaluation

For the amount of work done — less than a day of level editing and a day's verification, the resulting work is encouraging. The change in mean is fairly small, but the original level's large variance substantially affected how often the single–thread's CPU core would saturate. The lower variance both meant that we would stay within a smaller processor–time budget, as well as utilize the CPU better.

In terms of level analysis, our modification strategy looks sensible, if simplistic. A single modification to the level — placing a building in what was the densest part of the level, did substantially improve the system's ability to handle load. Whether the effective capacity of the system, in terms of how many users could fit in the virtual world before feeling too cramped or otherwise uncomfortable, would require further study.

Seemingly small changes can have large effects in our system. For example, a new automatic weapon might change open–field combat to a form of trench warfare, invalidating everything we've studied about user behavior and their spatial density in the virtual world. As changes are made, even if each is small, the analysis cycle should be re-executed. User study results should be compared against studies prior to determine if users have changed behavior. The load simulator should be run to verify that the current set of changes results in expected system performance.

5 Summary and Future Work

We have introduced Software Scalability Engineering (SSE) and shown it through a substantial example. SSE has a multi-layered

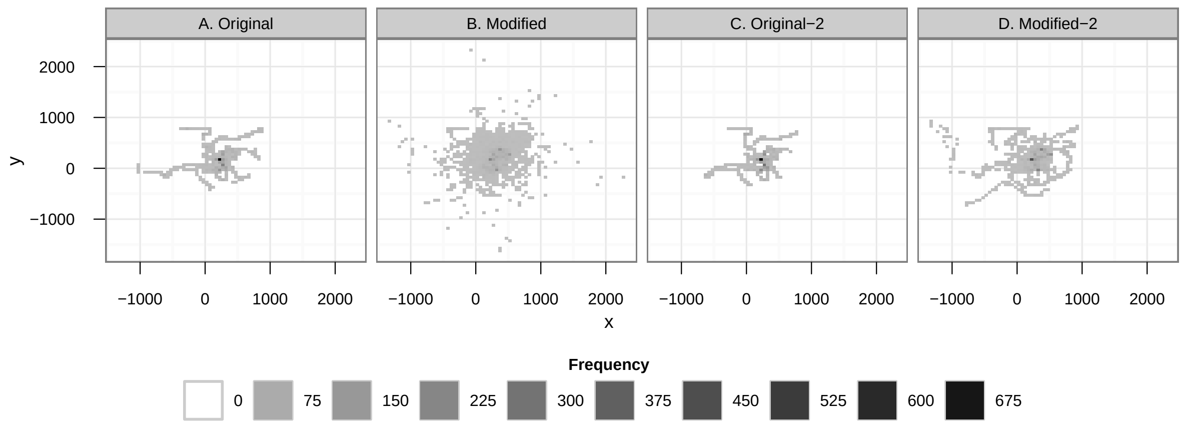

Figure 4: Original (A), Intentionally Modified (B), and Randomly-Modified (C,D) Level Densities, 5000 Randomly–Sampled Points Each

approach for iterative discovery and experimentation with a DVE's performance. Early iterations can determine the current state of performance and the its affectors. Then, we can "drill down" into the details of what drive these performance–controlling elements of the system, to determine their controlling factors. Finally, we can experiment with ways to reduce the load on bottlenecks, through manipulation of those factors.

We started by covering Software Scalability Engineering (SSE) in detail. We then applied SSE to the Torque engine. SSE gave us deep control over the scope and breadth of our work. Starting with a pair of identical user studies to construct a load simulator, we quickly used `top(1)` and a network monitor to isolate the primary bottleneck when scaling up users — the single–core performance of the host. Through roughly a half–dozen iterations through the cycle, we have identified the system bottleneck's main user: player's collision detection; and identified its largest factor: the spatial density of players.

When initially discovering the primary user of the bottleneck resource, we evaluated the work and benefits of multi–threading the engine. Instead of deeply affecting system behavior, and taking the risk of complex multi–threaded bugs, we evaluated changes to the virtual environment itself for improvement in system performance. Here, we completed a reasonably successful experiment in relieving pressure on that bottleneck using a modification in the virtual terrain.

References

[1] A. Abdelkhalek and A. Bilas. Parallelization and performance of interactive multiplayer game servers. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 72, April 2004.

[2] A. Abdelkhalek, A. Bilas, and A. Moshovos. Behavior and performance of interactive multi-player game servers. In *Cluster Computing*, 2001.

[3] S. Aggarwal, H. Banavar, A. Khandelwal, S. Mukherjee, and S. Rangarajan. Accuracy in dead-reckoning based distributed multi-player games. In *SIGCOMM 2004 Workshops: Proceedings of ACM SIGCOMM 2004 workshops on NetGames '04*, pages 161–165, New York, NY, USA, 2004. ACM Press.

[4] L. Chan, J. Yong, J. Bai, B. Leong, and R. Tan. Hydra: a massively-multiplayer peer-to-peer architecture for the game developer. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, NetGames '07, pages 37–42, New York, NY, USA, 2007. ACM.

[5] GarageGames. Torque game engine. `http://www.garagegames.com/products/browse/tge/`.

[6] C. Greenhalgh and S. Benford. Massive: a distributed virtual reality system incorporating spatial trading. In *Distributed Computing Systems, 1995., Proceedings of the 15th International Conference on*, pages 27–34, May 1995.

[7] V. Mönkkönen. Multithreaded game engine architectures. `http://www.gamasutra.com/view/feature/1830/`.

[8] P. Morillo, J. M. Orduña, M. Fernãndez, and J. Duato. Improving the performance of distributed virtual environment systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 16:637–649, July 2005.

[9] M. Oliveira, J. Jordan, J. Pereira, J. Jorge, and A. Steed. Analysis domain model for shared virtual environments. *International Journal of Virtual Reality*, 8(4):1–30, December 2009.

[10] P. Quax, P. Monsieurs, W. Lamotte, D. D. Vleeschauwer, and N. Degrande. Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game. In *SIGCOMM 2004 Workshops: Proceedings of ACM SIGCOMM 2004 workshops on NetGames '04*, pages 152–156, New York, NY, USA, 2004. ACM Press.

[11] S. Rieche, K. Wehrle, M. Fouquet, H. Niedermayer, T. Teifel, and G. Carle. Clustering players for load balancing in virtual worlds. *Int. J. Adv. Media Commun.*, 2:351–363, December 2008.

[12] N. Sheldon, E. Girard, S. Borg, M. Claypool, and E. Agu. The effect of latency on user performance in warcraft iii. In *NETGAMES '03: Proceedings of the 2nd workshop on Network and system support for games*, pages 3–14, New York, NY, USA, 2003. ACM Press.

[13] H. Singh and D. Gracanin. A methodology for managing distributed virtual environment scalability. In *Proceedings of the 2011 Winter Simulation Conference*, November 2011.

[14] H. Singh, D. Gracanin, and K. Matkovic. An approach to quantification and analysis of quality in distributed virtual environments. In *Telecommunications (ConTEL), Proceedings of the 2011 11th International Conference on*, pages 503–510, June 2011.

[15] H. L. Singh. PPT: The portable performance tool. `http://github.com/lally/libmet`.

[16] C. U. Smith and L. G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison Wesley, 2002.

[17] I. Software. Quake 3 arena. `http://www.quake.com/`.

[18] B. Watson, N. Walker, W. Ribarsky, and V. Spaulding. Effects of variations in system responsiveness on user performance in virtual environments. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 40(3):403–414, September 1998.

[19] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero. Atomic quake: using transactional memory in an interactive multiplayer game server. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 25–34, New York, NY, USA, 2009. ACM.