An Approach to Tuning Distributed Virtual Environment Performance by Modifying Terrain

H. Lally Singh[*]
Google and Virginia Tech
Blacksburg, VA, USA

Denis Gračanin[†]
Virginia Tech
Blacksburg, VA, USA

Krešimir Matković[‡]
VRVis Research Center
Vienna, Austria

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. These behavioral elements are important for evaluating virtual terrains. While two terrains may be similar in terms of user experience, task efficiency, immersion, and even aesthetics, they may exhibit substantially different performance out of the DVE 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 substantially improve the CPU requirements of the Torque engine. With a key factor determined, we evaluate several modifications to the original terrain. Finally, a modification is selected for its ability to stabilize the simulation time.

Categories and Subject Descriptors

H.5.1 [**Information Interfaces and Presentation (e.g., HCI)**]: Multimedia Information Systems—*Artificial, augmented, and virtual realities*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Design, Performance

1. INTRODUCTION

Distributed Virtual Environments (DVEs) systems themselves are performance-sensitive, but remain elusive to analyze due to their complex interactions between software, geometric assets, and user behavior. However, DVE system performance is an important metric. It indicates how long a user waits between supplying input to the system and seeing the world update in response. Looking at the simplest DVE configuration: client/server, we may break down the server's workload into input, simulation, and output phases, where the I/O denotes communication with clients.

The resulting DVE performance characteristics 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. The changes in human behavior can be very hard to predict, and result in substantial changes to the resulting load applied to the system. The resulting level of reality perceived by the users is sensitive to the performance of the system. 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. Modern software practice provides tools that can help. We present such a methodology designed for DVE systems, Software Scalability Engineering (SSE), followed by an in-depth example of it being used for substantial analysis, change, and positive impact on the user-visible performance of a Torque [2] based DVE. We will use SSE to determine the key performance factors in Torque, and use that information to alter the virtual terrain to enhance performance.

2. RELATED WORK

Performance has often been a key constraint in Distributed Virtual Reality systems. Zhang et al. [6] describe a system using a combination of simulation and measurement to determine the values of performance metrics. Quax et al. [3] provide an analysis of the effects of latency and jitter on user performance, as measured by the in-game score in Unreal Tournament 2003. Watson et al. [5] studied the time-to-feedback from user actions and its effect on user performance. We present a methodology derived from Software Performance Engineering (SPE), an iterative, model-based software engineering process. Smith and Williams describe it in detail in [4]. 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

[*]lally@vt.edu
[†]gracanin@vt.edu
[‡]Matkovic@VRVis.at

static assets (e.g. 3D models) of the system being built.

3. THE SSE PROCESS

Our methodology is an strongly derived from Software Performance Engineering [4], 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. 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 have to process user logins well beyond the point where it can provide reasonable service.

When beginning the scalability engineering process, or the engineering project as a whole, some initial *preflight* work up-front is needed. The desired scalability, allowable effort, and critical components of the software are identified. The *analysis* phase is the head of SSE's iteration loop. Initially, it is entered after preflight, and re-entered after changes to the virtual environment or substantial changes to the DVE software. We gather human usage data to determine how users commonly end up using the system. We then modify or re-implement the client software to behave representatively like the observations.

For *modeling*, we use an incremental analyze-and-simulate method to scale the effort, and focus on the most quantitatively significant parts of the system. We place instrumentation into the engine to validate and calibrate the models. Then we run the load simulator against the engine and record data. We then look at the instrumentation data, to 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. The model, engine, or virtual environment assets are modified in the next iteration to make the models show confidence in desirable performance in the system.

4. TORQUE PERFORMANCE ANALYSIS

The Torque engine comes with a sample level ("Original"), a small village with a lake, three large towers, and a small dock. Figure 1 shows the village and one of the towers. The other levels are part of an experiment discussed later. The only avatar available is a large Orc, with a single weapon, an explosive crossbow. We iteratively build a model of the user behavior at load based on the conducted user study, while iteratively building a model of the DVE. 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, the users are all equals and fight each other. Group dynamics, if present, seem to be lost in the normal variation found during user study. As the game was very simple — one weapon, small town, and only kills counted for points — the number of tactics people used proved small. The players' locations were recorded in 1 Hz intervals. This *density map* (Figure 3 has 4 examples) will help estimate the amount of work done during collision detection in simulation.

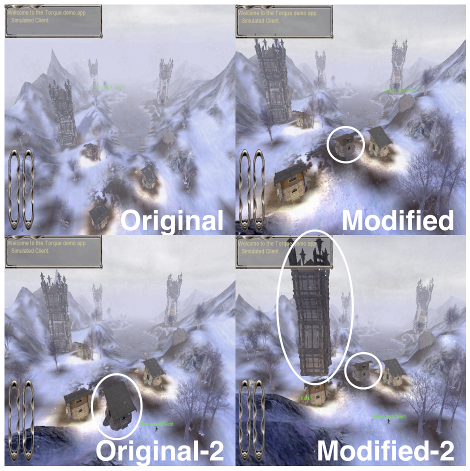

Figure 1: Torque Levels Evaluated

4.1 Load Simulation, Metrics, Geometry

An AI-driven synthetic player is created. It implements the behaviors observed at the frequency they were observed. They have enough logic to move, select targets, aim, chase, and fire. A table of way–points and direct linear motion towards them was used for navigation. 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.

We run a quick load simulation to observe the system running. The instrumentation that the operating system provides — bandwidth counters and `top(1)` — provide sufficient instrumentation for our needs. Even at high load, Torque does not use a fraction of the memory or network bandwidth available on the server — improvements in their availability since Torque was developed have been substantial. However, substantial CPU usage is shown.

In our current Torque setting, we only have one geometric model for the player and one level. The effects of the player's model are indistinguishable from other components of the performance. When the detailed behavior of that routine is of more interest do the models' attributes or distribution come into interest. The data was taken from instrumentation of the user simulations only — projectiles, power-ups, and buildings are not represented. The `simulate` frame from the `ppt` instrumentation — discussed later — was the source of this data. The relevance of this data source selection will become clearer in the case study.

4.2 Data Collection

We developed the `ppt` tool for data collection. It uses *frames* of data, each atomic, to assemble blocks of related data together. A stream of frames is collected from the program and converted to a text table for statistical analysis. The first frame is `coreloop`, which represents a cycle of the top-level loop in Torque. Individual event times can be summed as contributions to the `coreloop`-spent time. The `coreloop` frame also includes the number of logged-in users.

The `simulate` frame represents a full simulation of the virtual world. It increments over all objects and simulates them a discrete time step forward. it contains the bulk of

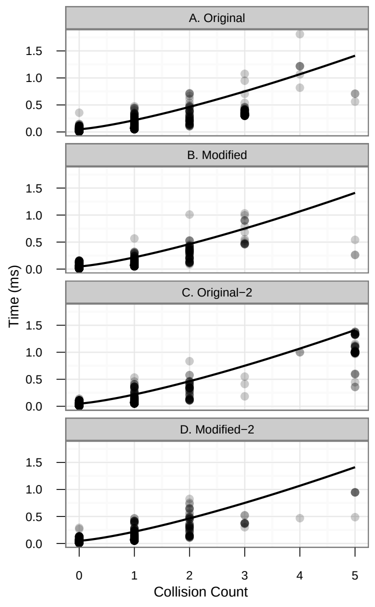

Figure 2: Collision Detection Times vs Collision Counts at N=30, with our Model Overlaid as a Line

our data and the focus of our analysis: (1) `object`: A unique identifier Torque gives to every object instance in the virtual world; (2) `type`: An integer representing the runtime type of the object; (3) `x,y,z`: The current location of the object; (4) `start,end`: The start and end times of the simulation work done for that object in the current time step, in `hrtime_t` format.

The server and load simulators were set up on Amazon Elastic Compute Cloud (EC2) virtual machine instances. One machine instance was dedicated to the server, and fifteen client simulators were on each of four other machine instances. `ppt` data, after conversion, creates text files that we analyze in GNU R.

4.3 Performance Analysis

Looking at the breakdown of processor time, the engine spent 72% of all CPU time in simulation of the users, even at low load ($N = 1$ through 12). The rest of time in simulation, another 1.2%, was spent simulating arrows. The remaining 16.2% of CPU time was spent in synchronization with client machines. Additional iterations of the analysis cycle gave more detailed information.

The simulator works by simulating the entire virtual world in small time–steps. Each object is moved forward along its current trajectory, with forces applied, for the length of the time step. Then, collisions are checked, and any responses (such as health adjusted for hits, or position adjustment to avoid going through walls) are processed.

For Torque, the time step is 32 ms, but the length of the time–step may be increased if the software cannot simulate the time step in 32 ms. Then, each object is moved further before collisions are checked. As the time spent simulating an object is independent of its time–step duration, this technique is effective in making the engine auto–balance its simulation accuracy with latency.

The work for a simulating a single step is executed through a call from `ProcessList::advanceObjects()`. This method iterates through all objects in the virtual world and invokes a `processTick()` method on each. As players are the primary user of CPU time in the simulator, we will focus on the relevant method: `Player::processTick()`.

`updateMove()` and `updatePos()` were determined to have nontrivial execution times at 250 μs and 1.6ms. These correspond to the movement and collision-detection phases, respectively. The others totaled to roughly 7.2 μs. The method `updateMove()`, while nontrivial in execution time, had a very low variance: 6.99e-4 ms^2. The method `updatePos()`, however, had a variance of 20.45 ms^2. Through algorithm analysis of the code in the method, combined with instrumentation, the collision count was found to be the largest factor. The time spent in `Player::updatePos()` is shown on the vertical axis in Figure 2. We have a simple model for `Player::updatePos()`, a polynomial based on the number of collisions (c):

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

Equation 1 is based on a simple curve-fit of the data, initially assuming that the function was in the family $y - y_0 = Ax^B$. The model leaves out many factors affecting the runtime of `updatePos`: cache misses, context switches, and variation in search time through the scene–graph data structures for candidates to collision–check. The time given is for a specific computer, and will vary with other machines; however, the difference will be linear to this function, and the exponent dominates.

The key metric we intend to construct is the distribution of th erunning time of a player's physical simulation, e.g. a single call to `Player::processTick()`. The metric indicates when the map is too expensive to simulate, due to too much work required for collision processing. Equation 1 models the time spent in collision detection, as shown in Figure 2. The values of this factor (c) drive the amount of CPU time that Torque needs to support a DVE. So, an experimental hypothesis is formed: can the number of collisions be controlled, to control the CPU requirement of the system? More specifically, can we modify the virtual terrain to reduce the number of collisions that need processing at any moment in time?

5. LEVEL ALTERATIONS

Figures 1 and 3 show four variations of the DVE. We first did the analysis on the "Original" level (A). There is one particularly dense spot at approximately ($x = 250, y = 200$), the dark square in Figure 3 (A). Players are often there in number. We put a new building there to diffuse player positions. That attempt is shown as the "Modified" version (B). The building itself was a duplicate of one of the four identical buildings already in the same area: a small ten–by–ten meter single room with walls on four sides, windows, and a door. We assume that collisions correlate with spatial density.

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,

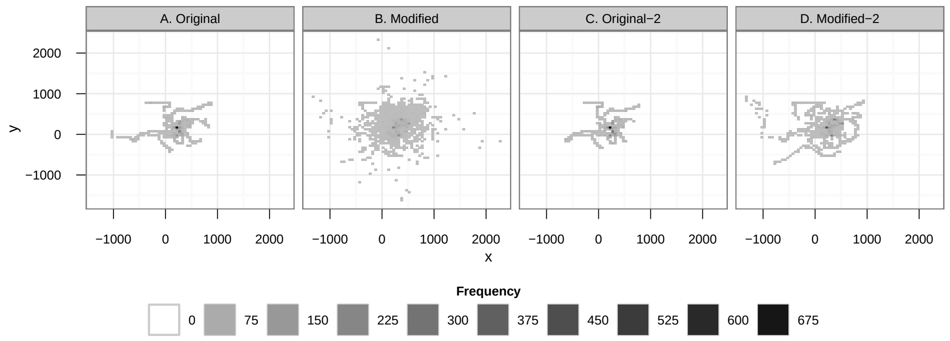

Figure 3: Original (A), Intentionally Modified (B), and Randomly-Modified (C,D) Level Densities

blocking a key snipe–point, a hill to the left of the pictured area, from working. In Figure 1, it occludes an existing tower (changes are shown with white ellipses).

Looking at Figure 3, 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. Figure 3 shows data acquired to show movements over the instrumented time interval. Four simulations were run, each with 60 users. All sub-plots show five-thousand randomly–selected invocations of `Player::processTick()`.

The level modification was simple: place another building in the middle of the densest region — the middle of the virtual village. The effect was better than expected. The building density in the village increased enough that the load simulators moved out to a nearby clearing – on the right side of the area visible in Figure 1. 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.

The original (A) average time through the main loop was 10.66 ms and the modified (B) level 10.36 ms — a three-percent improvement. More importantly, the variance in simulation time — the difference between players being in the least and most densely-populated regions — was reduced from 43.42ms^2 to 20.65ms^2. The occasional peak in simulation time indicated by the high variance caused dissonance between the client and server-side simulation. It manifests in objects "jumping" from the position simulated by the client to that received from the server. The other two levels had a worse mean runtime, 10.91 ms for "Modified-2" and 10.99 ms for "Original-2". The variances were 86.05ms^2 and 87.32ms^2, respectively.

6. SUMMARY AND FUTURE WORK

Through roughly a half-dozen iterations through the cycle, we have identified the system bottleneck, modeled how it was being used, identified its largest factor, and completed a successful experiment in relieving pressure on that bottleneck. A simple, general method for relieving bottleneck pressure was found in the process. While Software Scalability Engineering (SSE) can and has been used for altering system software, it can just as easily be used to determine which parts of the user interface can be adjusted for better performance. In the case od DVEs, SSE's user behavior modeling and simulation facilities have demonstrated promise in being able to guide enhancements to the exposed virtual reality.

Determining the largest factor in system performance — simultaneous collisions detected for a single player for a single time step — enabled us to understand the terrain's affect on performance. Specifically, we looked at density maps (spatial histograms) of player positions and used the peaks as areas of concern in terrain design. We hypothesized that reducing the peak density in a terrain can have substantive, measurable effects on system performance. In the future, applications of SSE to even more parts of the DVE system stack — synchronization, quality of experience management — are planned.

7. REFERENCES

[1] T. de Senna Carneiro and J. Cotrim Arabe. Load balancing for distributed virtual reality systems. In *Proc. of Int. Symp. on Computer Graphics, Image Processing, and Vision*, pages 158 –165, Oct. 1998.

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

[3] 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 *Proc. of ACM SIGCOMM 2004 workshops on NetGames '04*, pages 152–156, New York, 2004. ACM.

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

[5] 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, Sep. 1998.

[6] M. Zhang, H. Xie, and A. Boukerche. A design aid and real-time measurement framework for virtual collaborative simulation environment. In *Proc. of IEEE Int. Symp. on Parallel Distributed Processing, Workshops and PhD Forum*, pages 1–6, Apr. 2010.