A Load Simulation and Metrics Framework for Distributed Virtual Reality

H. Lally Singh* — Virginia Tech, USA
Denis Gračanin† — Virginia Tech, USA
Krešimir Matković‡ — VRVis, Austria

ABSTRACT

We describe a simple load-measure-model method for analyzing the scalability of Distributed Virtual Environments (DVEs). We use a load simulator and three metrics to measure a DVE's engine with varying numbers of simulated users. Our load simulator logs in as a remote client and plays according to how users played during the conducted user study. Two quality of virtuality metrics, fidelity and consistency, describe the user's experience in the DVE. One engine performance metric provides the cycle time of the engine's primary loop. Simulation results (up to 420 users) are discussed.

1 INTRODUCTION

Our overall research focuses on building scalability models of Distributed Virtual Environments (DVEs). Specifically, we're studying the per-user growth rates of memory, CPU, and network bandwidth requirements. The process is straightforward: we provide different levels of load to the system and measure the resources the engine uses. That data will be used to construct a scalability model of the engine, providing memory, cpu, and network utilization estimates per logged-in user.

While we add load, we have to make sure that the DVE stays usable. Even if several hundred clients can log in, it doesn't matter if the DVE is no longer realistic. To keep the simulation accurate, we provide some metrics measuring the quality of the virtual reality presented. We call it *Quality of Virtuality* (QoV).

With a minimal QoV requirement, we ensure the simulations and the resulting models are relevant and accurate. When the system fails to provide this minimal QoV, we can stop modeling it's scalability —- we expect users to start logging off.

Here we discuss our load simulator, our QoV metrics, and a very simple engine-level metric for CPU usage.

2 LOAD SIMULATION

A simple method to add load is to add on many users and some loss and jitter to the network. Adding jitter and wild user movement help, but they should be kept within a representative amount.

Getting many people to come together & play is difficult to do repeatedly, and requires many client computers. Instead, a software simulator can be constructed. It doesn't have to be nearly as sophisticated as a human player — we have some useful limitations that will reduce the complexity of our software.

We have elected to build a simulator based directly on observations of humans. This is plausible because the game our DVE executes is incredibly simple.

The game we use as our DVE basis is known as a simple "twitch-action" First–Person Shooter, known for working better with good reflexes than complex strategy. In our game there is no strategic advantage to any player's state beyond position. We modified the original one-weapon game to have infinite ammunition and immediate respawn.

*e-mail: lally@vt.edu
†e-mail: gracanin@vt.edu
‡e-mail: Matkovic@VRVis.at

IEEE Virtual Reality 2008
8-12 March, Reno, Nevada, USA
978-1-4244-1971-5/08/$25.00 ©2008 IEEE

2.1 User Study

The data came from a pair of user studies, where users played a Torque [2] game against each other. On two occasions, we placed users in a computer lab and had them connect to a single server. Each computer recorded every message with the server. The first time, we had ten students, the second, five.

We took recordings of each player and analyzed them separately. The simulator executes 16 different behaviors, with the observed frequencies of each. Figure 1 lists them. We later found that sniping at less than 50m was functionally equivalent to standing and attacking. Hence, the simulator merges those two behaviors.

We built a simulator based on this list of behaviors. It executes each of the tactics according to that distribution — the selection is random based on the set of under-executed tactics. The duration of the tactic is randomly selected, as is the firing rate.

Behavior	Total %	Behavior	Total %
Circle-Strafe	15.9%	Hide-Snipe	0.9%
Scatter	15.7%	Snipe 50m	4.0%
Snipe 100m	6.6%	Snipe 150m	2.7%
Snipe 200m	1.7%	Snipe 300m	0.9%
In-Building Snipe	4.6%	Chase-Kill	15.9%
Reverse-Attack	3.5%	Wander	10.2%
Shoot into Building	1.5%	Inactive	7.7%
Parallel Strafe	1.5%	Sinusoid	2.1%
Stand & Attack	4.6%		

Figure 1: Behaviors from the User Study and their distribution

2.2 Simulator Behavior

The simulator contains a set of scripted actions that takes over a client at login, and executes each of the 16 actions randomly. When one action has had more than it's appropriate share of runs, it's taken out of the candidate list for selection. Different actions depend on varying levels of knowledge about the environment — things that a user would normally understand by sight.

In practice, the simulator works reasonably well. Observing it in action, it traverses in and out of buildings, tracks down, chases, and kills other players, and snipes them from various distances. Using the knowledge base of the level's features, it routes around buildings and can use them as cover when attacking.

3 METRICS

We have three metrics, grouped into two categories: Quality of Virtuality and Engine Performance. In the former category, we present two metrics that measure the fidelity and consistency of the system. In the latter, we present a simple metric for CPU usage.

3.1 Quality of Virtuality

The first two metrics cover the fidelity and consistency of the DVE. We define these experiences as those that validate from a single second of the experience (such as random jumping of players) for the former, to those that take much longer to figure out (such as failures in the physics or collision systems) for the latter.

Authorized licensed use limited to: IEEE Xplore. Downloaded on November 18, 2008 at 09:55 from IEEE Xplore. Restrictions apply.

Fidelity: The Circle Metric — DVEs often communicate by sending dead–reckoning (DR) vectors to one another [1], indicating the current position, direction, and velocity of the object. We take advantage of this fact by running an avatar in a curved path, and measuring how much error we get on the remote side.

We use two software-controlled clients that log into the DVE. The first we call the *actor*, which goes to a predermined location and begins strafing (walking sideways) in a bounded circle around a central point. It records its position as it goes. The second we call the *observer*: it finds a viewpoint where it can see the actor all the way around the circle, and records its position. By comparing the differences in the observed versus the actual positions, we have a metric for fidelity lost over the communications line. Some fidelity can be lost in between normal DR updates, and the rest lost when DR updates are lost across the network.

Consistency: The Collide Metric — DVEs often run in a simple input-simulate-render loop. That representation is updated for the current time tick. The simulation includes moving all the objects along their DR vectors, then doing collision checks. Within this loop we have a "catch up" method: all the objects are moved along their (straight) DR vectors for all the time in between the last simulation tick and now. For Torque, that's 16 Hz on the server and 32 Hz on the client. If the engine gets bogged down, then the time through the loop will increase, making the simulation stages further and further apart. Objects in the DVE may get moved so far in a single time through the loop that they may pass completely through each other before a collision check is done. That collision check will register no collision, as the objects have since cleared each other.

Our metric is very simple: we have two software-controlled clients log in and repeatedly move to predetermined locations and run into each other. They record whether they hit each other or got to points past where they should have collided. They then move back to their origins and try again. Using server-relayed messages, they synchronize their runs. We observe the collisions and non-collisions to get a measure of how consistent is the DVE.

3.2 Engine Performance

As a simple control measure, we track how long the engine takes to get through a single iteration of it's topmost loop. This simple metric, *CoreLoop*, serves as our best measurement of the CPU requirements of the DVE engine. The engine doesn't `sleep()`, `select()`, or otherwise block it's own operation. Instead, it runs as fast as it can, using additional CPU for more precise simulations (through a smaller simulation period) and faster framerates. By measuring *CoreLoop*, we have a rough characterization of how much work the engine does.

4 Analysis

We ran a simulation on a cluster of 24 AMD Athlon 3800 (1 GHz) machines, connected with gigabit ethernet. One hosted the server, another hosted our four metrics clients, and the other twenty two ran our simulators. The four simulation clients and the server add a total of five clients to the simulators we add on. We aimed for 500 clients, but could only get up to 420 before clients could no longer log in due to client crashes and server timeouts. All clients were automatically restarted if they crashed or were logged out.

Figure 2 shows how many logged in users we had over the simulation, overlayed with our circle metrics data. Figure 3 shows *CoreLoop* and our collision metric. After the first few clients finally had their levels fully downloaded, *CoreLoop* settled down for some time. It later destabilized as engines started crashing and needed their levels re-downloaded. Non collisions are only reported if both actors got into position and successfully ran towards each other first.

After the downloads finish, the engine's *CoreLoop* drops down to where the collisions register reliably. There are a few collisions detected when the *CoreLoop* is high, but that simply means that the objects happened to advance into colliding positions — not that the simulation reliably worked.

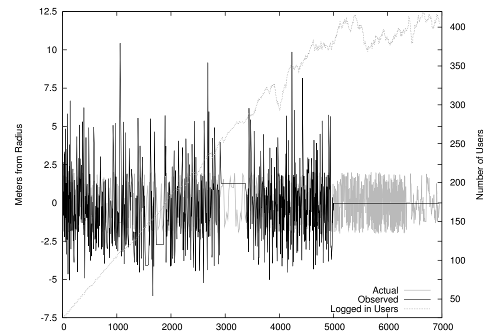

Figure 2: Circle Metric and Logged–in Users

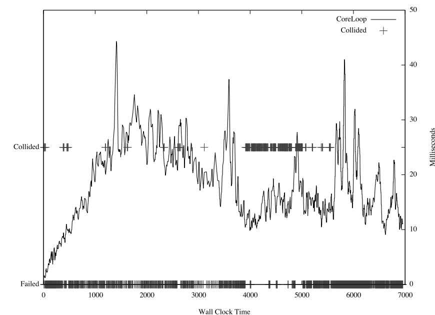

Figure 3: Collision Occurrences

Figure 2 shows the collision actor and observer's data. The actor's value is their distance from the center, minus their orbit radius. The observer's value is the observed distance between the actor and the center. While the actor provides a reliable control, the observer sees the actor quite differently. (The position is reported before going over the wire to the server, so it's not sensitive to it's state.) It was usually about 50-100% off from the actual value, and quit early on us at $t = 5003$. The observer either kept crashing or timing out after that. The lack of correlation with the *CoreLoop* or pure number of logged-in numbers indicates that the engine continues to send reliable DR vectors under significant load.

5 Conclusions

The initial tests show promising results for a heavily loaded server. With these initial results, we have a solid foundation for comparing the simulation results against the actual use of a DVE. With a calibrated load simulator we can then go back and use a fine-grained simulation to find the failure loads of this DVE — the minimum QoV of this DVE.

References

[1] D. A. Fullford. Distributed interactive simulation: its past, present, and future. In *WSC '96: Proceedings of the 28th conference on Winter simulation*, pages 179–185, New York, NY, USA, 1996. ACM Press.

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

Authorized licensed use limited to: IEEE Xplore. Downloaded on November 18, 2008 at 09:55 from IEEE Xplore. Restrictions apply.