Proceedings of the 2011 Winter Simulation Conference
S. Jain, R. R. Creasey, J. Himmelspach, K. P. White, and M. Fu, eds.

A METHODOLOGY FOR MANAGING DISTRIBUTED VIRTUAL ENVIRONMENT SCALABILITY

H. Lally Singh
Denis Gračanin

Virginia Tech
2202 Kraft Drive
Blacksburg, VA 24060, USA

ABSTRACT

Distributed Virtual Environments (DVEs) are a large class of real-time simulation systems. We present an implementation-independent methodology for measuring, analyzing, and comparing DVE systems performance. The methodology comprises of a process of requirements elicitation and their conversion into measurable objectives. The process for determining quality requirements for a DVE is discussed, with a focus on interaction-based scenario analysis. An example is given with a simple game — Asteroids — that has been modified to support two players (users).

1 INTRODUCTION

Any virtual reality system with more than one computer involved may be considered a form of a Distributed Virtual Environment (DVE). This includes any distributed version of flight simulators, multiplayer game, collaborative virtual environment, or multi-user training system. In general, DVE systems are difficult to design.

DVE systems comprise of a real-time physical simulator, synchronization protocol, and graphics system. Each has complex requirements. The physical simulation has to tolerate synchronization errors, handle large numbers of objects, and be accurate enough to both achieve the objective of the DVE — be it training, collaboration, or entertainment — and suspend disbelief for every one of its users. Additionally, the interactions of these systems with each other and non-software components (its *assets*) — such as physical models of objects in the environment — complicates analysis. The assets contain the graphical and physical models of objects in the simulation: avatars, buildings, projectiles, vegetation, vehicles, etc. During simulation, they may be ignored, treated as immovable barriers, pushed, bounced, walked-on, or thrown. In all but the first situation, their specific attributes affect the performance of the simulation.

Traditionally, performance management was done on an ad-hoc basis. As these systems continue to become more sophisticated, a methodology becomes more useful.

For DVEs, scalability has two major factors: the ability for many users to log-in to the system simultaneously, and for them to receive acceptable system performance once logged-in. The former can be tested through the simple act of logging in many users. In the methodology presented, a load simulator is built and many instances are run simultaneously, testing this first factor. The latter performance factor is complex; it involves the performance and interacting behaviors of the major software components and assets of the DVE, and some of the behavior of the users themselves. Once the required number of users may log in, we abbreviate scalability to the derivative of performance, with respect to logged-in users.

Quantified performance is an important topic of interest in DVE systems. With increasing demands for better performance, better scalability, and additional functionality, quantification becomes a natural component of a system improvement process. Key concerns within measurable performance objectives include the number of sustainable users within a given resource envelope, synchronization facilities to support specific simulation accuracies, and high-demand collaborative interactions. The relationships between quality factors, system load, and the computer resources required to support them are key concerns in understanding and managing the scalability of a DVE system.

We have developed a process, Software Scalability Engineering (SSE), designed to measure, model, predict, and control the relationship between a DVE's assets, software components, resource requirements, and performance. This process is incremental in nature, and allows its own process overhead to be tuned to the project as more or less accuracy is desired in understanding those relationships. While our initial research has focused exclusively on DVEs, the process can be used for many kinds of simulations. The unique inclusion of user behavior and the assets of the system enable better analysis than other methods.

In this paper, we will discuss the process and illustrate it by analyzing a very small multiuser game.

1.1 Types of DVEs

Several types of DVE systems exist today: (1) simple client-server systems with one server; (2) cellular client-server systems with one server per region; (3) shared client-server systems with hot spare servers; and (4) a ever-growing body of novel peer to peer systems in research and industry.

All require that some node do real-time physical simulation and a node-node synchronization mechanism. DVE systems types 2–4 require a mechanism to split up the work of maintaining the DVE. Type 3 requires a replication mechanism. Type 4's peer to peer engines may require any of a large set of additional mechanisms, including shared state canonicalization, consistency management, dynamic partitioning, identification protocols, and super-node selection.

With all the problems DVE system designers have to solve, a common subset is present throughout: synchronization and simulation (Singh, Gračanin, and Matković 2008, Singh and Gračanin 2009). These activities are often significant components of a DVE engine's workload — driving significant parts of the resource requirements for operating the system. While we analyze entire DVE systems, we focus our analysis for these two areas specifically in the hope that the models and instrumentation points we have built can be reused with minimal effort.

The DVE types have different synchronization systems, but they have a common structure that we will exploit in the analysis therein. While some systems may have different classes of synchronization connections — peers, clients, identification servers, etc., we believe that they can be analyzed with the process described in this paper. Focusing on the most reusable subset of a large family of systems, we have chosen to focus on client-server DVE systems.

2 RELATED WORK

While DVE engines simulate physics in discrete time, other events simultaneously occurring lend them to use discrete-event simulations internally. These other events include network and user interface I/O, timeouts, etc. Using discrete-event simulation, network and input events may be handled together within a simple framework (Eberly 2005). Simulations have accuracy requirements unique to their context, those appropriate for DVEs will be discussed later in this paper.

First some higher-order concerns for simulation are considered. Their applicability to DVEs depends on the objectives of the DVE in question.

Robinson (Robinson 2002) provides a high-level methodology for analyzing the quality of a simulation, in terms of software engineering and the process of social change:

1. *Quality of the Content:* How well does the simulation development process fits with the original requirements?
2. *Quality of the Process:* How was the simulation work performed?
3. *Quality of the Outcome:* How useful the work is within the scope it was done for?

Watson et al. evaluate the effects of frame rate on task performance in (Watson, Spaulding, Walker, and Ribarsky 1997). The research evaluated both frame rates and variations in frame rate as dynamic factors affecting user task performance. The research showed that user susceptibility to changes in frame rate is higher at lower frame rates (10 Hz) than at higher ones (20 Hz). At the higher frame rates, variations "have little or no effect on user performance" for the types of tasks they used in the experiment.

For traditional First Person Shooter (FPS) games, Quax et al. (Quax, Monsieurs, Lamotte, Vleeschauwer, and Degrande 2004) provide an analysis of the effects of latency and jitter on user performance, as measured by their in-game-score in Unreal Tournament 2003.Using a router that simulated specific latencies and jitters, they found that users had noticeable impairment when the round-trip-time (RTT) surpassed 60 ms. Bhatti and Henderson (Henderson and Bhatti 2003) found that the number of times the player kills per minute (KPM) dropped from 1.456 KPM to 0.6233 KPM when lag was artificially introduced. Similarly, the number of times the player was killed per minute jumped from 0.6042 KPM to 1.430 KPM.

Corwin and Braddock (Corwin and Braddock 1992) investigate the use of metrics in distributed systems, from development through operations. They examine the values of development metrics and models to quantify assumptions, and operational metrics to assist in adjusting system policies and planning upgrades.

At the network level, the Internet's best effort service is only one of many defined. RFC 2211 (Wroclawski 1997), allows for controlled load service, acting as a best effort link over a lightly loaded link. This reduces loss and jitter significantly, leading to stochastically stabler performance for DVE synchronization. The DVE will provide more predictable performance due to the additional stability in its synchronization system. RFC 2212 (Shenker, Partridge, and Guerin 1997) also specifies guaranteed service covering hard bounds on delay, in turn giving guarantees on bandwidth. Hard guaranteed systems may have to sacrifice average throughput for stable throughput, giving less performance than what would be possible on a controlled load flow over the same link. A quality specification system is already in place for describing Internet QoS (Braden, Zhang, Berson, Herzog, and Jamin 1997). It can describe soft or hard guarantees for latency, bandwidth, and jitter.

3 APPROACH

We present a process methodology, derived from Software Performance Engineering (Smith and Williams 2003) to construct a scalable DVE system. This methodology has several properties that work well with the constraints seen during the development of distributed virtual environments. First, it is incremental, with a minimal first round. Second, its code analysis technique directly connects the constructed models and predictions with the software components, enabling modification of specific portions of the system in a controlled manner. Finally, the work artifacts — the models and load simulator — are useful beyond the design and engineering process. They are also useful for observing the system during operations, evaluating future changes, and — if so desired — enabling dynamic adaptation to load.

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.
3. Large components of the system load are determined by both the human behavior in its players and the system-specific artistic assets created to populate the scene graph.

The process is shown in Figure 1.

We use three resource estimators as the system performance model to manage through the engineering process: Network bandwidth, CPU capacity, and memory (real available memory) requirement.

Developers may create additional models for additional resources which may bottleneck during runtime, such as dedicated hardware (e.g. server-side CUDA GPUs) or router memory. We believe that the methods used for determining the three presented are easily transportable to other resources as needed.

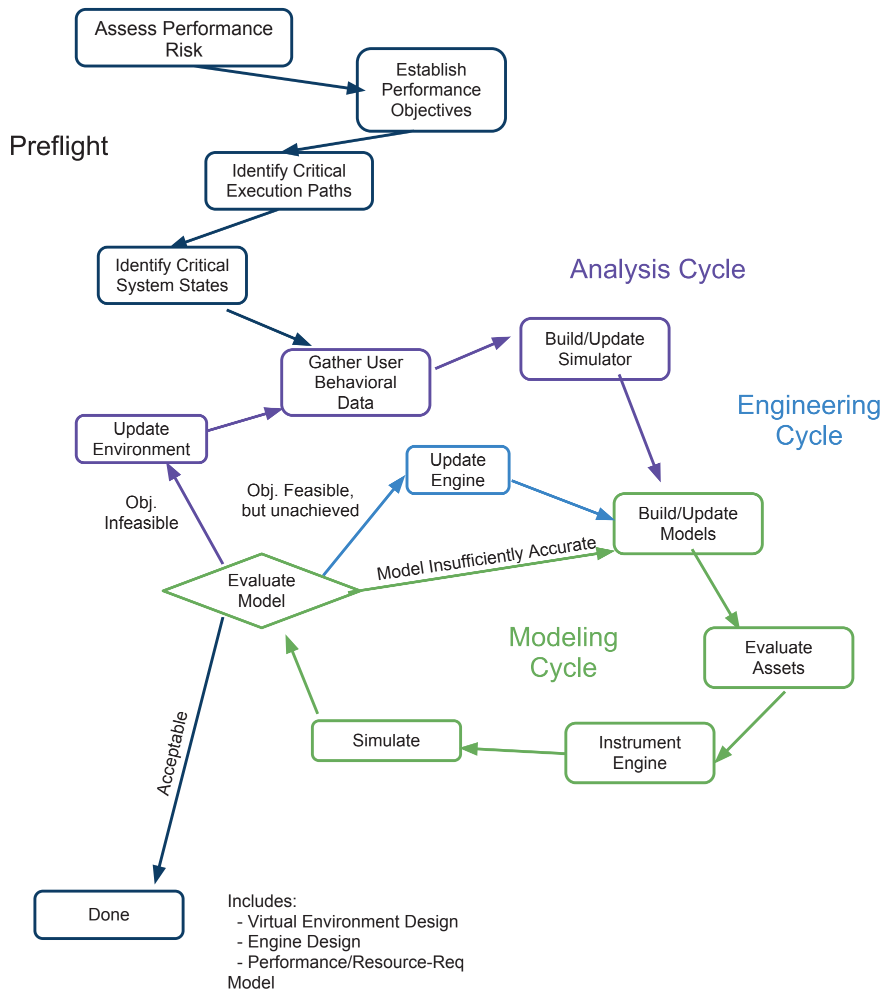

Figure 1: Analysis and modeling procedure.

We define a DVE's scalability as the set of relationships between the number of logged-in users and the system's resources. These relationships are expressed as functions upon the number of logged-in users. We also define three categories of models:

1. *Load Model* that includes the actual values of variables and the distributions of events coming into the system at different levels of load.
2. *Resource Model* (RM) that translates the load model's values into actual resource requirements (CPU, memory, network bandwidth), the RR-envelope for short.
3. *Performance Model* that describes the system behavior requirements for acceptable performance. This model often includes: the minimum update rate to a client for their own state, the minimum rate for a client to receive updates about objects fitting various criteria, and the numerical accuracy of the physical simulation.

The Load Model has separate estimates for each software component involved in critical scenarios, and how they apply to critical resources: typically including CPU, memory, and network resources.

The Resource Model estimates the resource usage ramifications of those components at those loads, and totals them into a top-level estimate for the system load. The Performance Model relates the resource model values to the performance objectives of the project.

The steps are listed below. We prefix each with a letter denoting which cycle they belong to: (P)reflight, (A)nalysis, (E)ngineering, and (M)odeling. The phrase "Build/Update" denotes the construction of an artifact the first time, and a modification to it on subsequent passes.

(P) When beginning the scalability engineering process, or the engineering project as a whole, some initial work up-front is needed. *Assess Performance Risk:* Determine what levels of scalability are desired, and how much engineering effort it is worth. *Identify/Establish Critical Paths:* Determine which software components compose the performance and scale-critical paths through the software. *Identify Critical State:* Determine what data structures are used for the critical paths. *Establish Scalability Objectives:* Determine 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.

(A) *Gather Behavior Data:* 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. *Build/Update a Load Simulator:* Modify or re-implement the client software to behave representatively like the observations.

(M) 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. *Build/Update Models:* We construct or update the load, resource requirement, and performance models. See Section 3.1 for details. *Evaluate Assets:* Analyze the structure of the artistic assets, to determine their effects on load.

(A,E,M) *Instrument Engine:* Place instrumentation into the engine to validate and calibrate the models. *Simulate:* Run the load simulator against the engine, enable instrumentation, and record data. *Evaluate Model:* 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. If the engine does not perform sufficiently well, go to the "Update Engine" phase, which will carry on to "Build/Update Models," completing the Engineering Cycle. 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.

(Update back-arcs) (A,E) *Update Environment:* 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 it next iteration. *Update Engine:* With behavioral and instrumentation data, some changes may be desired to make the system perform. When complete, the engineering cycle restarts at the "Build/Update Model" stage.

DONE: The engine has been measured to perform as desired within the expected resource constraints. Simplify the instrumentation to reduce overhead, but leave enough to allow observation of the system in production.

Our incremental method starts with an analysis of the system. We map the critical behaviors to specific paths through the code of specific software components. We'll denote these paths as critical paths. Then, we break the code down into blocks, put together a rough $O(N)$ model for it, with N being the number of logged-in users — and add instrumentation to confirm (and calibrate) or refute that model. Simulation will fill the model in with data from the instrumentation.

For DVEs, we start with the bulk-sum resource usage: the total per-top-loop time for the engine, the total memory usage, and the total bandwidth used by the process. Iterations through the modeling cycle will indicate which model components need finer accuracy to determine whether a constraint or requirement is met.

Confidence interval analysis should indicate which components of the model are acceptable and which ones need additional modeling work. Additionally, simulating at high levels can identify software components that will need additional work — be it modeling, re-engineering, or DVE concept alteration — to become acceptable. For additional details (Jain 1991) is recommended.

3.1 Code Analysis

While many of the components we have discussed so far have derived from traditional methods, such as Software Performance Engineering (Smith and Williams 2003), we have a novel three-stage process for direct program code (software) analysis. Our term *Code Analysis* denotes a calibrated algorithm analysis technique, applied across critical code paths. We convert our system load parameters into values, rates, and distributions of input on the system. We then propagate the load into states of specific variables, data structures, and flows of execution that lead to estimates for the resources required for the system to operate.

Load Reification: First, the we convert the load values from our load-performance envelope to real expected event values and rates within the source code. For our load parameter N, we expect that each client will transmit updates to us at 10 Hz. That is $10N$ input packets per second, and roughly $10N$ state changes. Additionally, that means that our scene graph has N avatars, plus a number $proj(N)$ of projectiles in-flight. Most importantly, we instrument the values we put in our load model to verify and calibrate it.

Value Propagation: The reified load values are applied to the code within a software component. When that component uses another, we use the relationships between the load values and the parameters of the invoked component to build a load model of the called component. Through this mechanism, we propagate the reified load values into a full load model for the the transitive closure of the call graph of the critical paths of the system.

Resource Model Derivation: With the component load model complete, we convert the load into resource requirements. Normal algorithm analytic techniques can derive a basic resource model for the CPU: we simply add instrumentation to calibrate the constants. Network I/O is a transmitted rate expectation, multiplied by a transmitted size expectation.

Memory size is more complex. We have to build an expected state of data structures within the system under load. This includes the scene graph and any network protocol data. In the latter category we have any data needed for reliable transmission of events, protocol data, and network socket information.

4 CASE STUDY: ASTEROIDS ENGINE

We'll use a small example engine, **Asteroids**, to illustrate. Seen in Figure 2. Asteroids (Brett) is a simple one-player version of the 1970s classic, that we have modified to be two-player. Each player is represented by an isosceles triangle, and they may fly around by a single thruster. They can also rotate arbitrarily and fire bullets at the simulation rate — that is, each simulation pass can create a new bullet for each player. Their bullets inherit their current motion vector, plus a fixed vector in the direction fired.

As an added twist, the edges wrap around at an offset. The top and bottom edges wrap, as do the left and right. The offset has the effect that going straight up causes the player to show up at the bottom of the screen about halfway over to the right from where they came in. Likewise, going to the far right causes the player to reappear on the left side, only shifted halfway up the screen, modulo the screen height.

The engine was originally a single-player, single-threaded system, but it was minimally modified to support a second player. The modifications were:

1. *Store New Bullets* — As new bullets were created, they were added to a list. This list is drained when the new bullets' data are transmitted to the peer.
2. *Setup Networking* — A command-line flag indicates whether the program will wait and listen for a peer as a server (`-s`) or connect as a client (`-c <ip_addr>`).
3. *Transmit State* — The new-bullet list and the current player state are transmitted at 10 Hz.
4. *Receive State* — A new thread awaits and processes updates from the peer.

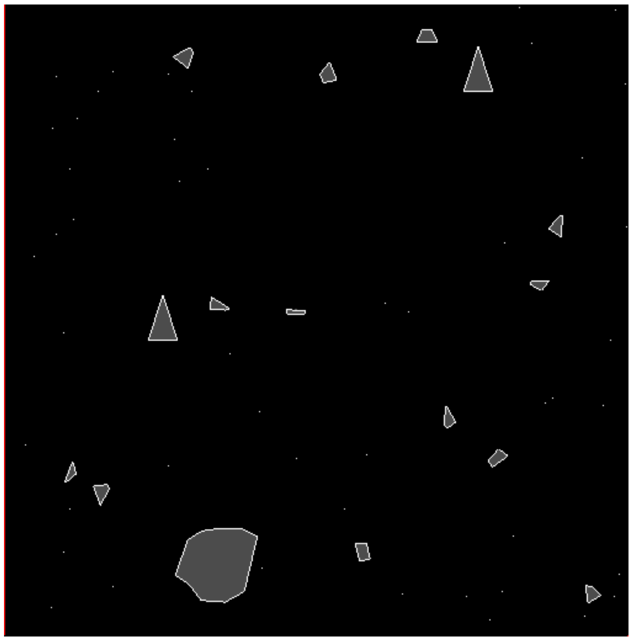

Figure 2: Asteroids.

5. *Use Recursive Mutexes* — Recursive mutexes were added to the scene-graph and player state objects to allow direct update from the receiver thread.
6. *Change Initial Position* — The flag used for determining whether the program is in “client” or “server” mode is also used to determine the initial starting point for their space ship. Server players are moved halfway left, clients halfway right. Otherwise, a cycle of new-life, explode, new-life prevented the game from being playable.

Asteroids themselves are represented as simple convex objects of varying shape, size, and position. They move in straight lines until they bounce. Asteroids destroy players. Bullets destroy asteroids and players.

4.1 Preflight: Performance Risk, Critical Path, State, Objectives

With the Asteroids project, we begin by determining what we want to achieve and how much system design work the goals are worth. First, we should determine an objective for the analysis: to show a small walk-through the process, and to get a basic understanding of how Asteroids works. Different projects will clearly have their own work metrics, such as project iterations or programmer-hours.

From inspection of the source code, the critical path of this game is a simple two-part loop on the server:

1. *Simulation*: the virtual environment is simulated for a time step.
2. *Update Transmission*: the new state of the virtual world is transmitted to to the peer at 10 Hz.

As we mentioned earlier, the processing of input data from the other player is handled in a separate thread. While the two threads may compete over the processor, we really don’t expect either of them to require enough computational power to really be an issue. We treat them as if they’re competing for the same processor.

The clients transmit a simple update packet over UDP: a player state (a position, velocity vector, and a rotation angle), and a variable number of bullets (each with a position and velocity vector). While incredibly simple to describe, implement, and analyze, there are a few caveats:

1. The two simulations may diverge due to synchronization lag from the 10Hz update rate, network latency, or dropped packets.
2. There is no synchronization for game-level events, such as a level completing.

Generally, having one server simulate for both (and send the entire state down to the client) will solve the divergence issue, and a second TCP stream for game-level events will maintain synchronization. However, such additions don't particularly help our analysis-elucidation goals and instead add substantial complexity.

Each client acts independently, with its own simulation system and scene-graph. The peer's projectiles (bullets) and player state are transferred over as a mirroring effect, but are themselves then subject to the local simulation.

The critical state for Asteroids is the scene-graph and the received update packets.

4.2 Analysis Cycle: Behavior Data, Load Simulator

From preflight, we enter the analysis cycle. This cycle starts with gathering user behavior data for the DVE. Normally, we would put two players (users) together, and record their movements. We would start with individual games and then move to a two-player game. However, for a game this simple our normal load simulation cycle is unnecessary — we can get two users to play easily.

The users can exhibit various behaviors.

- Idle behavior: the user may be away from the computer, talking to someone, thinking, or have any other reason for not pressing a key.
- Asteroid hunting behavior: firing directly at asteroids.
- Anti-asteroid behavior: moving to avoid asteroids.
- Local attack behavior: the user fires upon the other user, without moving towards them.
- Chase attack behavior: the users fire and move towards their target simultaneously.
- Run behavior: the user moves away from bullets coming from the other user.

The users execute these behaviors as they like. Normally, a modified version of the software would execute these behaviors using a distribution observed during a user study, but human players are more accurate than a simulator and thusly are preferable when feasible.

4.3 Modeling Cycle: Modeling, Instrumentation, Simulation, Evaluation

We start with a simple set of models for our system: a load model, a resource requirement model, and a performance model. Each assists in the construction of the next. For each model, we go through the critical path of the system: input, simulation, and transmission. We define a few variables, for values that we're best determining by instrumentation: A is the number of active asteroids. B is the number of active bullets.

4.3.1 Load Model

- **State:** The scene graph has A asteroids, two users, and B bullets. We instrument the engine to determine these values.
- **Input:** Each client transmits its changes to the scene state at 10Hz. Input processing will execute at the input rate, which also will be (at most, after possible dropped packets) 10Hz. The total input processing time is a linear function of ΔB.

- **Simulation:** The simulation work starts. The keyboard event queue is drained, and applied to the current input-state map. The objects of the world are simulated forward in time, then a collision check is performed.

Every moving object — asteroid, bullet, and ship, are simulated in a discrete time step forward until now (we denote it `increment` phase, as we move one step forward as a time). Then, every object that's died is deleted and any items created during simulation are added to the scene (`cleanup`). Then, the traditional Asteroids wrap-around behavior is implemented: any object touching any edge of the screen is also partially drawn at the opposing edge of the screen (`wrap`). The effect makes the left and right edges map next to each other, and the same for the top and bottom. Finally, a collision detection pass (`collide`) marks objects for destruction. The simulation workload is the sum of the phases' work. They are:

- `increment`: $O(A+B+2)$ — Move two users, A bullets, and B bullets forward in time one step.
- `cleanup`: $O(\Delta A+\Delta B)$ — Add or remove objects that were created or destroyed during the time step.
- `wrap`: $O(A+B+2)$ — Check each object against the edges of the screen.
- `collide`: with $k=A+B+2$, $O(\frac{k(k+1)}{2})$ — Check each object against every other object.

The resulting complexity is:

$$O(\frac{A^2+2AB+6A+6B+B^2}{2}+\Delta A+2\cdot\Delta B)$$

Denoting the uncalibrated work for a single time step, which should equal the total work through the entire main loop, not counting the transmission. The transmission is $O(\Delta B)$ at 10 Hz.

Transmission: The state transmission rate and work is identical to the state reception work, possibly less after packet loss. Each client transmits its changes to the scene state at 10Hz. Input processing will execute at the input rate, which also will be (at most, after possible dropped packets) 10Hz. The total input processing time is a linear function of ΔB.

4.3.2 Resource Model

The scene graph has A asteroids, two users and B bullets. The memory is allocated and freed off of the heap, using reference-counting smart-pointers (`boost::shared_ptr<T>` (Group)) manage lifetime. A bullet (`class shell`) was 64 bytes, a user (`class ship`) 192 bytes, and an asteroid (`class rock`) 128 bytes. The shared pointer itself is 16 bytes. Everything but the dynamic portion of the resources — the numbers of bullets and asteroids — are counted as the *static load* of the system. This load is constant. The memory for maintaining the scene graph is the sum of two users, A asteroids, B bullets, and the shared pointers for each reference to those objects. The scene graph itself only keeps one reference to each object.

$$A\cdot(128+16)+B\cdot(64+16)+2\cdot(192+16)$$

Additionally, the input processing phase has a queue of pointers to new bullets that have not been transmitted over:

$$\Delta B\cdot 16$$

The memory resource model is simply the sum of these two equations.

Computationally, we've already determined how much work is necessary through the critical path; it simply needs calibration. The compute-resource model (which we may abbreviate as the CPU model) is a linear combination of the work required for the input, processing, and transmission phases.

$$\begin{array}{rcl} & C_1 & \\ + & C_2 \cdot 2 \cdot (A+B+2) & \texttt{increment + wrap} \\ + & C_3 \cdot (\Delta A + \Delta B) & \texttt{cleanup} \\ + & C_4 \cdot \frac{(A+B)^2+(A+B)}{2} & \texttt{collide} \\ + & C_5 \cdot 2 \cdot (\Delta B) & \texttt{input + transmit} \end{array}$$

Which we've simplified (using another set of constants and the reduction above) to:

$$C_1 + C_2 \cdot (A^2 + 2AB + 6A + 6B + B^2) + C_3 \cdot (\Delta A + 2 \cdot \Delta B)$$

The network bandwidth required is $10Hz \cdot (20 + \Delta B \cdot 16)$. The numbers 20 and 16 are the sizes of the users and bullet states in the update packet, respectively. The bullet count is implicit in the packet size.

4.3.3 Performance Model

To determine the system performance, we first determine the rates of work required versus the rates of requisite resources being available. For the CPU, we convert the computational part of the resource requirement model to a time unit, calibrated to the deployment processor. However, with such a simple game, it is prohibitively difficult to collect timings on these components — accessing the system timer would be more computationally expensive than a good part of the code we are investigating.

Instead, we use a simpler method: instrument for values of A, B, ΔA, ΔB, and the timings of the main loop and input thread. Calibration would require additional numbers of users to determine $C_1 \ldots C_3$, so we have them stand as unknown in this iteration. When we find a particular constants interesting, we can look at them later on.

4.3.4 Simulation

To get the values for A, B, ΔA, ΔB, and the loop times, we use our `ppt` (Singh) tool. We use six timers: the beginning and end of the main loop, the beginning and end of the drawing phase, and the start and end of the receiving thread. The drawing phase is completely within the main loop. We store values of A, B, ΔA, ΔB, and counters for how many bullets are sent and received.

Each simulation run is rather simple: two people play each other and we store the instrumented values. The mean measured values were: A: 3, B: 2.286, and ΔA: 0 — this was level one, with few asteroids, and the users seemed to aim for each other more than anything else. ΔB has an interesting oscillating pattern between zero and five — our users seemed bursty in their firing rate. We'll count it as 2.5 for now.

The primary loop averaged 181.5ms, surprisingly high. However, the drawing phase itself was 175.5ms on average. While there is some A and B-dependent work in the drawing phase, we count it as static as it is dominated by buffer I/O. The remaining 6ms is the result of our simulation and networking work.

With some raw data, we gathered these zero-values (Table 1).

Table 1: Zero values: $A = 0$, $B = 0$, and $A = 0$ & $B = 0$.

Condition	Average Main Loop	Average Draw	Average Remainder
$A = 0$	56.0ms	55.3ms	0.6ms
$B = 0$	241.0ms	238.4ms	2.5ms
$A = 0$ and $B = 0$	160.4ms	159.8ms	0.5ms

Note that we only had one point for $A = 0$, which makes it pretty suspicious.

With this data, ΔB averaged 2.5 steadily, and ΔA stayed at zero. Clearly it may change with other users, levels, or moods. With these essentially constant, we can fold C_3 into C_1, solving them to 100 microseconds. As a quick (if not terribly robust) method, we used means for $A = 0, 1, 2$ at $B = 0$, to also solve the quadratic around C_2 as 374.83 microseconds.

4.3.5 Results

Our simulation results tell us quite a bit about both the engine and what to explore next for further analysis. We determined that our drawing time is nearly thirty times longer than simulation time. If there's anything we can optimize in drawing, it should be our first effort. This may require a more detailed analysis of the drawing engine, as we hadn't instrumented it at all, focusing more on the simulation work.

Still, considering the graphical simplicity of the game, we expect that the drawing is I/O bound on buffers between memory and the X server (we were using a remote X server connected to the game inside of a virtual machine).

On a scalability perspective, the engine takes quadratic simulation time for either bullets, asteroids, or (with a low ceiling) users. With a low maximum for the number of users, we are mostly concerned about how well the system performs at high levels of intensity: high asteroid and/or bullet counts. The constant C_2 tells us how expensive additional bullets or asteroids are: a jump from three to four asteroids will cost us another 4.8 milliseconds in simulation time. These add up quickly, and the engine really doesn't "scale" well for new objects.

As an obvious conclusion, we should not add ships that fire more than one bullet at a time, the cost to simulation time would quickly be prohibitive — both in bullets simulated and the small shards of asteroid created as a result. If the drawing is found to be I/O bound, we should consider additional work in the game in terms of textures or sound effects.

5 CONCLUSION

Using a modified version of Software Performance Engineering and detailed code analysis, we can determine the primary factors in the performance of a system. More relevantly, we can model these factors' influences in performance parameters of the system as we alter the number of logged-in users. We can predict how many system resources — CPU, memory, and network capacity — are required to support a given number of users. Using this information, we can evaluate a system design or current implementation, and alter it to better suite requirements. That ability enables direct management of the system scalability during development.

Using the simple Asteroids game as an example, we have shown that a small run through the entire process can be quick and light-weight, yet still illuminate some of the inherent properties of the system performance. While the game presented here could only handle two users, the same techniques can be applied — even incrementally — to the more sophisticated networking system, client/server configuration, and additional simulation load required for a system handling many users.

Finally, the incremental nature of the process enables us to discover attributes of the system that need exploration in one pass (as the drawing phase), and guidance for how to explore them in additional rounds. The presented results provide an example how to made project-specific decisions on how much to analyze, and how much project capital to expend in the analysis. Practitioners can take the same decision process and scope their scalability-management efforts similarly to meet both their development budgets and project requirements.

REFERENCES

R. Braden and L. Zhang and S. Berson and S. Herzog and S. Jamin 1997, September. "Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification". RFC 2205 (Proposed Standard). Updated by RFCs 2750, 3936.

Nick Brett. “Asteroids”. http://www-pnp.physics.ox.ac.uk/~brett/projects/asteroids.html [accessed July, 2011].

Corwin, B. N., and R. L. Braddock. 1992. “Operational Performance Metrics in a Distributed System. Part I.: Strategy”. In *Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing (SAC ’92)*, edited by H. Berghel and G. Hedrick, 867–872. New York, NY, USA: ACM Press.

Eberly, D. H. 2005. *3D Game Engine Architecture*. Morgan Kaufmann.

The Boost Group. “Boost C++ Libraries”. http://www.boost.org/ [accessed July, 2011].

Henderson, T., and S. Bhatti. 2003. “Networked games: a QoS-sensitive application for QoS-insensitive users?”. In *RIPQoS ’03: Proceedings of the ACM SIGCOMM workshop on Revisiting IP QoS*, 141–147. New York, NY, USA: ACM Press.

Jain, R. K. 1991. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley.

Quax, P., P. Monsieurs, W. Lamotte, D. D. Vleeschauwer, and N. Degrande. 2004. “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*, 152–156. New York, NY, USA: ACM Press.

Robinson, S. 2002, April. “General concepts of quality for discrete-event simulation”. *European Journal of Operational Research* 138 (1): 103–117.

S. Shenker and C. Partridge and R. Guerin 1997, September. “Specification of Guaranteed Quality of Service”. RFC 2212 (Proposed Standard).

Singh, H. L., and D. Gračanin. 2009. “Load Characterization for Distributed Virtual Environments”. In *Proceedings of the 1st International Workshop on Concepts of Massive Multi-user Virtual Environments (CoMMVE’09)*. Kassel, Germany.

Singh, H. L., D. Gračanin, and K. Matković. 2008, 8–12 . “A Load Simulation and Metrics Framework for Distributed Virtual Reality”. In *Proceedings of the 2008 IEEE Virtual Reality Conference (VR ’08)*, edited by M. Lin, A. Steed, and C. Cruz-Neira, 287–288.

Lally Singh. “ppt: Portable Performance Tool”. http://github.com/lally/libmet [accessed, July 2011].

Smith, C. U., and L. G. Williams. 2003. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Revised ed. Addison-Wesley Object Technology Series. Addison-Wesley Professional.

Watson, B., V. Spaulding, N. Walker, and W. Ribarsky. 1997, March. “Evaluation of the effects of frame time variation on VR task performance”. *Virtual Reality Annual International Symposium, 1997., IEEE 1997* 0:38–44.

J. Wroclawski 1997, September. “Specification of the Controlled-Load Network Element Service”. RFC 2211 (Proposed Standard).

AUTHOR BIOGRAPHIES

H. LALLY SINGH is a PhD student in the Department of Computer Science at Virginia Tech. His research focuses on the design of distributed virtual environment systems and related performance issues. His email address is lally@vt.edu.

DENIS GRAČANIN is an Associate Professor in the Department of Computer Science at Virginia Tech. His research interests include virtual reality and distributed simulation. He is a senior member of ACM and IEEE and a member of AAAI, APS, ASEE and SIAM. His email address is gracanin@vt.edu.