Controlling the Scalability of Distributed Virtual Environments

Hermanpreet Singh

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science and Applications

Denis Gračanin, Chair

Shawn A. Bohner

Wu-Chun Feng

Roger W. Ehrich

Ivica I. Bukvic

April 3, 2013

Blacksburg, Virginia

Keywords: Virtual Reality, Performance, Scalability, Distributed Systems

Controlling the Scalability of Distributed Virtual Environments

Hermanpreet Singh

(ABSTRACT)

A Distributed Virtual Environment (DVE) system provides a shared virtual environment where physically separated users can interact and collaborate over a computer network. More simultaneous DVE users could result in intolerable system performance degradation. We address the three major challenges to improve DVE scalability: effective DVE system performance measurement, understanding the controlling factors of system performance/quality and determining the consequences of DVE system changes.

We propose a DVE Scalability Engineering (DSE) process that addresses these three major challenges for DVE design. DSE allow us to identify, evaluate, and leverage trade-offs among DVE resources, the DVE software, and the virtual environment. DSE has three stages. First, we show how to simulate different numbers and types of users on DVE resources. Collected user study data is used to identify representative user types. Second, we describe a modeling method to discover the major trade-offs between quality of service and DVE resource usage. The method makes use of a new instrumentation tool called `ppt`. `ppt` collects atomic blocks of developer-selected instrumentation at high rates and saves it for offline analysis. Finally, we integrate our load simulation and modeling method into a single process to explore the effects of changes in DVE resources.

We use the simple Asteroids DVE as a minimal case study to describe the DSE process. The larger and commercial Torque and Quake III DVE systems provide realistic case studies and demonstrate DSE usage. The Torque case study shows the impact of many users on a DVE system. We apply the DSE process to significantly enhance the Quality of Experience given the available DVE resources. The Quake III case study shows how to identify the DVE network needs and evaluate network characteristics when using a mobile phone platform. We analyze the trade-offs between power consumption and quality of service.

The case studies demonstrate the applicability of DSE for discovering and leveraging trade-offs between Quality of Experience and DVE resource usage. Each of the three stages can be used individually to improve DVE performance. The DSE process enables fast and effective DVE performance improvement.

Acknowledgments

I'd like to thank my adviser, Denis Gračanin. His patience, wisdom, patience, guidance, and patience has been invaluable through this long endeavor. He's been a mentor and a friend, and I can't put my gratitude into words.

I'd also like to thank my Ph.D. committee, Shawn Bohner, Roger Ehrich, Ivica Bukvic, and Wu Feng. They have all been with me through this, and some for many years before. I've been at VT a long time, and my time here will always be a part of who I am, and a lot of that comes from them.

I would not have finished without the love and support of my wife and family. I love you. Thank you. The support of my colleagues and friends at Google helped me keep perspective and sanity. Special thanks to Dino Oliva and Greg Russell.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Distributed Virtual Environment (DVE) systems provide a shared virtual environment (virtual world) where physically separated users can interact and collaborate over a computer network. A DVE system consists of a physics engine, a virtual environment, a network-synchronization, and the user interface components (we describe DVE systems in detail in Section 2.1). Each user has an *avatar*, the user's representation in the virtual environment. The avatar often looks, moves, and acts like a person. The users interact with each other and the virtual environment through their avatars, typically using a dedicated computer. The computer allows the user to control the avatar and displays a user interface, a graphical rendering of the virtual environment from the avatar's point of view.

Current performance engineering techniques are not well suited for DVE systems. The primary performance factors in DVE systems are not well understood and vary from system to system. The user load is unusually complex since it is determined by the user behavior inside the virtual environment and the overall demands the users' behaviors puts on the

DVE system and resources. When the user load placed on the system exceeds regular usage patterns, the system performance degrades and error conditions occur. The only accurate DVE user load test method available is to observe large groups of users using the DVE system. It is often too expensive to realistically test user load on the DVE system. Without adequate testing methods, it is difficult to determine how DVE system components govern DVE system performance. Consequently, it is hard to make good DVE system design choices.

The DVE system performance requirements depend on the attributes of the provided virtual environment. We need to understand interactions between user behavior, the virtual environment, and system performance in order to determine the performance requirements and performance consequences of DVE design decisions for the virtual environment and the supporting DVE system components. Scalable DVE systems should continue to function well and meet a user need when the number of users or the user load increase. Very often a DVE system has a sub-optimal balance between the quality of the virtual environment it provides, the number of users it supports, and the computer resources it uses. More users in a virtual environment usually improve quality of experience, in part due to the increased opportunities for interaction and cooperation. However, more users in a virtual environment usually increase the load on the DVE system leading to reduced system performance and reduced quality of experience. In order to improve DVE system scalability, we need to address the three major challenges: effective DVE system performance measurement, understanding the controlling factors that determine system performance/quality and determining the consequences of DVE system changes.

1.1 Motivation

DVE systems provide a basis for a wide range of multi-user applications, from everyday entertainment/consumer-oriented activities and massively multiplayer online games to training, education and digital libraries, as well as medical, robotics, and other purposes. A well-known example is Second Life, a DVE system that provides an easy to use, persistent user-generated virtual environment with massive and dynamic content.

Unfortunately, DVE systems have traditionally been hard to scale. As more users participate, a DVE system could deplete some resources (e.g., memory, network bandwidth) resulting in the reduced quality of the user experience. DVE systems can make some quality trade-offs, but eventually the quality is too poor for users to tolerate.

Very few DVE systems support many users without substantial restriction in realism or performance. The DVE systems supporting many users typically partition the virtual environment into small segments and restrict the number of users in each partition. However, subtle design decisions can keep users from noticing the partitioning. The maximum size of each partition is limited in the same way as the DVE system as a whole, thus limiting the number of users in a partition.

The current scalability limits in DVE systems reduce their utility and their quality of user experience. The DVE systems need to have partitions to scale the number of users. The attempts to hide the existence of these partitions make DVE systems harder to use due to their requisite user interface complexity and artificial in-environment limitations. For example, users may have to face "realm selection" interfaces to choose which partition to

log into (e.g., an island in Second Life). Partitioning limits interactivity between users in different realms wishing to just text chat or use voice communications. A partition can reach limits or overflow, causing the partition to restart (temporarily forcing all users out of the virtual environment) or block new users from joining existing groups.

User movement between partitions needs special system considerations. A specialized partition can be made for the transportation "vehicle," such as a single rail car or aircraft with few simultaneous users. The vehicle won't travel in the same virtual space as non-traveling users, so that those on virtual train tracks won't be able to look through a window in a rail car and see a real user's avatar. Other systems may require direct selection of another realm through a menu or map.

Scalability limits can prevent representation of some real-life situations, due to the number of users those situations would require. Large audiences for an event, such as a concert or sporting match, can't be supported in a DVE system due to the high volume of user interactions. For example, the users that are members of a large audience in a virtual environment can't interact to form a "wave" through virtual bleachers, form a large virtual political protest, or run a virtual marathon with hundreds or thousands of other users.

The subtle limitations of spatial user density, regardless how the partitioning is done, and overt limitations in the number of users within a partition, hurt the DVE system's quality of experience.

Furthermore, DVE systems rarely optimally balance trade-offs between the virtual environment, the types of interactions its users can have, and resources. As we discuss later, the DVE system resource usage can grow super-linearly. Complementing that usage trend is a

wealth of options for changing the virtual environment and user interactions. Partitioning the virtual environment is a common, coarse technique for controlling user load growth. More subtle approaches, such as discouraging system-taxing activity in the most-loaded areas, or encouraging activities that the system is well-optimized for, are rare.

Unfortunately, the time, resource, and management costs of user load tests prohibit active use of these tests during development. Instead, user load tests are run at the end of primary development in a large testing phase. These few and late-run user load tests prevent use of the test data to make substantial improvements to the DVE system design.

The DVE systems under heavy user load (a large number of users) have both poor quality and poor utilization of computer resources. With a better understanding of the major factors that influence resource usage and the available options for controlling these factors, we can design DVE systems that provide better quality, support more simultaneous users, and better utilize the available resources.

1.2 Research Problem

DVE systems are resource-intensive systems with many complex performance requirements, complex user load, and many ways to change performance and resource use. Figure 1.1 shows the overall dependency, a loop, between qualitative (user experience) and quantitative aspects of DVE systems [1]. This dependency, called the "Causal Chain," is a linking among Quality of Service (QoS) attributes in an environment (user interface usability, responsiveness, and navigation options), the user's perceptions of the system, and the user's

behavior when using the system. Quality of Experience (QoE) attributes include the subjective elements of the user's overall experience. An overview of the Causal Chain and related frameworks is available in Section 2.2.

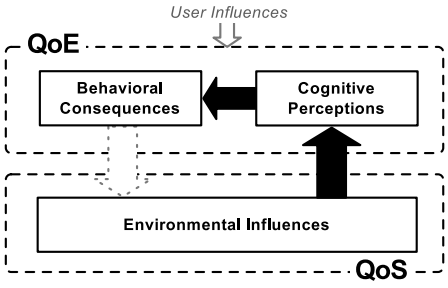


Figure 1.1: The overall model of dependency (interaction) between qualitative (user experience) and quantitative aspects of DVE systems (Causal Chain), from Wu et al. [1]

In the case of DVE systems, the "environment" shown in Figure 1.1 is the virtual environment. The QoS attributes are the DVE system's performance and quality. The dotted arrow shows the primary area of challenge for scaling DVE systems: the connection between load (user behavior) and the system's QoS. The filled arrows show connections studied through research by Wu et al. [1]. We discuss the Causal Chain in detail in Section 2.2.

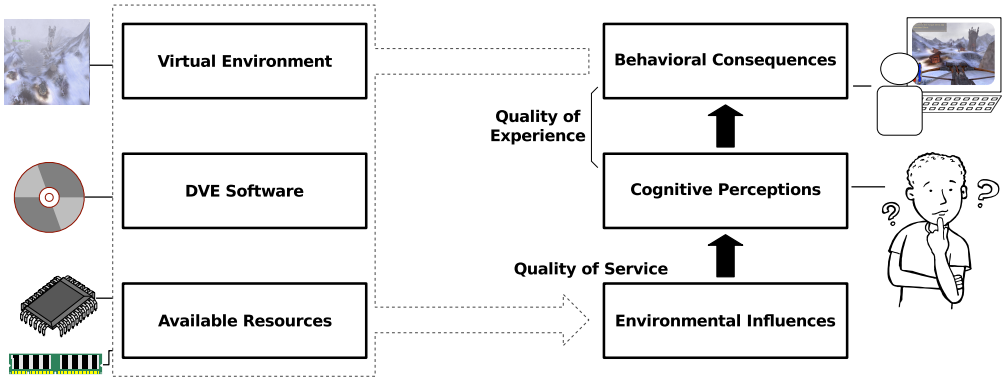


Figure 1.2: A DVE system in the Causal Chain

We rotate and elaborate the diagram shown in Figure 1.1 to provide a diagram for DVE

systems (Figure 1.2). The same white-filled, dotted arrow in Figure 1.1 between Behavioral Consequences and Environmental Influences now contains the components of a DVE system in Figure 1.2: the virtual environment, DVE system software, and other computer resources.

Elaborating Figure 1.2 further, we can provide a deeper description of the Causal Chain. We include and map QoS values to the DVE system components. Figure 1.3 shows this mapping. We have filled in connections between the components of the Causal Chain, and the components of the DVE system. Unfortunately, we don't have any information on what or how the DVE system components interact with one-another.

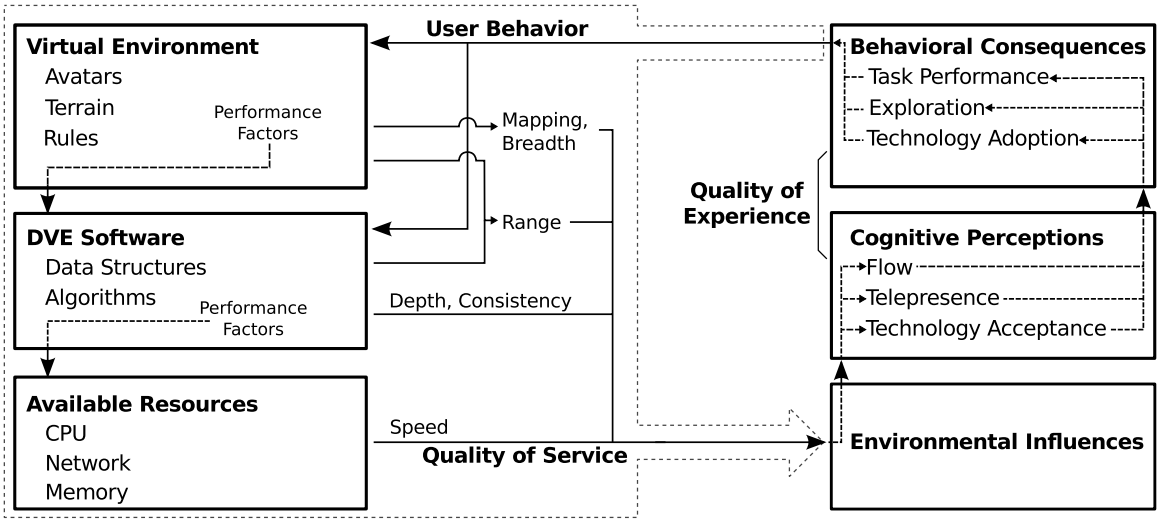


Figure 1.3: Detailed relationships between the DVE and the Causal Chain

The left side of Figure 1.3 includes all the elements of a DVE system. At the top is the virtual environment: the rules governing what users can do in the environment, the virtual space, and the media assets. Some combination of these elements, coupled with the way users act because of them, may be major factors in the performance and quality of the system (abbreviated to "Performance Factors" in the diagram in Figure 1.3). The specific combinations of the virtual environment's elements and the users' behaviors that become

major performance factors is unknown.

The second block is the DVE's software implementation. This includes all the data structures, algorithms, utility and runtime components, and the system design and architecture. In this block may also be major factors that determine the performance and quality of a DVE system.

The final block on the left side of Figure 1.3 is the set of hardware resources available to the DVE system. The available CPU cycles, memory, and network bandwidth are given as examples, but any available resource may constrain the DVE system's scalability. We discuss these resources in Section 3.1.

Figure 1.3 shows that the DVE system's Quality of Service is determined by a set of measurable attributes that include user interface choices, input devices, output media, and performance characteristics of the DVE system. The user experiences them all as a combination, and changes in one attribute's metrics can affect the desired values of other attributes' metrics. The user's response to the experience, in terms of actions within the virtual environment and specific inputs given to the DVE software, affects the amount of work the system must execute to maintain the virtual environment.

The cycle complicates the three major parts of performance engineering normally used to scale systems: determining the current level of performance, understanding the controlling factors of its performance, and changing performance.

1.2.1 Measuring the Scale of a DVE

The most widely used but still expensive and time-consuming method to accurately user load test a DVE system is conducting a user study with many real users. In March 2012, Weilbacher [2] noted that the Gears of War 3 project needed over 100 participants to perform a large user study test. Larger beta tests can involve substantially more participants, but require that the DVE system be well-developed and the side effects — setting user expectations for the DVE system — be acceptable. These tests also require substantial coordination and human effort.

Due to the dependency between DVE system performance and human behavior and super-linear growth in some resource requirements, it can be hard to develop small-scale tests that provide representative data of full-scale performance and resource use.

Programs acting in place of human participants hold promise in this area. Traditionally, the programs would replay traces of observed human behavior. If sufficiently phase-shifted, many instances of these programs could run simultaneously, appearing similarly to humans. However, this trace-playback approach does not account for changes in behavior due to changes in the number of users. This failure is important because users may act differently in some parts of a virtual environment, depending upon the number of other users they see there.

1.2.2 Understanding Factors of a DVE's Scale

Unlike the connected sub-components of the blocks on the right side of Figure 1.3, there is little understanding of how parts of a virtual environment or the related DVE software components affect performance.

The software for one DVE system may be wholly inappropriate for handling the load of another type of a DVE system. For example, the software for a virtual flight simulator may safely assume that each user moves in a virtual environment at a high speed and sees other users relatively infrequently. The related DVE software design focuses on accurate aerodynamics, physics and accurate collision detection. The same software would make a poor choice for a virtual soccer game. The users playing soccer move much slower than an aircraft, running-velocity physics are substantially simpler than complex aerodynamics, and collision detection has to be more accurate.

The amount of variability between virtual environments and the supporting software makes it very difficult to develop a general model of performance factors. Some less general model could be feasible for related DVE systems (same application domain), but a systematic method for developing such a model is yet unknown.

1.2.3 Changing the Scale of a DVE

If we were able to user load test a DVE system and knew the primary factors of the DVE system resource use, we would only then need to adjust those factors to improve the scale of the DVE system. We would have to address the ramifications of those adjustments.

One primary ramification can be a change in the user's behavior, and this change needs both testing and accommodation. If we simply adjust each factor in the virtual environment serially, from largest to smallest, we may end up with a "whack a mole" situation where adjusting one factor shifts load to another factor. The resulting process can consume substantial engineering resources and not converge to a better-performing system.

A process is needed to guide changes in a DVE system. Changes in the virtual environment and software may invalidate any code or data supporting a user load test, model data about load factors, or instrumentation. The process should properly require that any data, code, models, or instrumentation logically invalidated by changed properties of the virtual environment be rebuilt. For example, a change in the virtual environment could change how users behave, and that would require new user behavior models for the user load simulator. Also, that same change in user behavior can change the user load on the DVE system, and thus affect performance models.

The process should indicate these invalidations, direct development of necessary artifacts, and directly guide reasonable and convergent performance engineering.

1.3 Research Tasks and Methodology

We researched the requirements for all three stages of scaling DVEs: user load testing, data interpretation, and modification. The three major challenges are effective DVE system performance measurement, understanding the controlling factors that determine system performance/quality and determining the consequences of DVE system changes. We developed

a DVE Scalability Engineering (DSE) process that addresses these three major challenges for DVE design.

Our strategy was to develop an easy-to-run simulated user load experiment with deep instrumentation. We developed a customized performance-engineering process around the experiment. We also took an existing model basis for DVE system performance and extended it for use within the process. The DSE process allows us to identify, evaluate, and leverage trade-offs among the available DVE system resources, the DVE system components, DVE system software, and the virtual environment. The DSE process substantially extends Software Performance Engineering (SPE) [3] by integrating new user load testing and modeling methodologies. As a consequence, it is possible to explore and analyze the effects of changes in the DVE system design on the DVE system performance and quality of experience. The findings help us understand the primary scalability factors and help us make informed DVE system design decisions.

1.3.1 Develop a Reliable Simulated-Load Experiment

It's important to keep all the DVE system components in our study as realistic as possible. The environment, users, software, and runtime environment should be realistic. Consequently, we start with a real DVE system and attempt to build a realistic load simulation system for it.

To develop the experiment, we start with the most complex part: human behavior. We run user studies, characterize the observed behavior and then modify a copy of the DVE software

to emulate the observed behaviors.

With a mechanism for accurate user load testing, we next move into instrumentation for the DVE system to complete our platform for DVE system analysis. Using the data from the instrumentation, our modeling basis, and modeling method, we build a calibrated model of how the DVE system uses its resources. By doing this, the part of the DVE system software using the most resources is identified, and additional instrumentation and simulated-load experiments help to identify the controlling factors.

1.3.2 Develop a Modeling Basis and Method for Factor Discovery

Singhal and Zyda [4] defined the Networked Virtual Environment (NVE-IP) Information Principle relating key characteristics of a DVE system's structure and workload to the DVE system resource needs. We apply NVE-IP to each of three primary resources: CPU time, memory, and network bandwidth. The equations developed for each primary resource have variables/components that are directly measurable in a DVE system. These equations provide the modeling basis.

Using this modeling basis, we have defined an iterative modeling process that traces resource use from top-level DVE system software structures down to individual software components. The process traces the resource it finds most constrained. For example, the process follows CPU usage down the function call stack of each thread. Similarly, memory usage follows the instance-graph of allocations. The tracing process follows resource use up the relevant hierarchy until it identifies primary factors.

The process works thanks to a new instrumentation tool `ppt`. The `ppt` tool is designed for high-rate, low-overhead instrumentation of systems with a focus on enabling outlier and correlation analysis in and between developer-selected groups of variables. Combined with offline analysis, the process and tool can quickly find dominating factors in resource use.

1.3.3 Develop a Process for DVE Scalability Engineering

Once we identify performance factors, we can experiment with changing the virtual environment or DVE software. The factors may directly reduce resource use or change user behavior to indirectly reduce use.

We started with an existing engineering process, Software Performance Engineering (SPE) [3] and adjusted it as needed. SPE is an iterative process based on determining the performance feasibility of a design. We replaced the initial analysis with an accurate simulated user load experiment, and specialized its iterations into several "cycles" designed to more accurately build the model and experiment with changes in both the DVE software and virtual environment.

1.4 Results

Our research methodology has yielded good results. We now have the ability to measure, understand, and modify a DVE system's scalability. We formed this ability from a facility to run load simulation experiments, a custom measurement tool `ppt`, a modeling methodology for finding the primary factors controlling a DVE's scalability, and a performance engineering

process for measurement, modeling, and modification of DVE scalability.

1.4.1 Load Simulation Experiments with Measurement

Our first result was a methodology for constructing load simulators capable of representing a model of user behavior. The method takes user studies and converts characterizations of observed behavior into executable finite automata. Those automata are then implemented in a modified version of the DVE software.

Surprisingly, the existing instrumentation tools were insufficient for our needs, so a new tool was developed. `ppt` is lightweight enough to avoid interfering with measured performance and resource-use values in DVEs. It preserves raw collected data for sophisticated offline analysis and allows ad-hoc, arbitrary collection of any desired data.

1.4.2 Modeling Basis and Methodology

We extended Singhal and Zyda's [4] Networked Virtual Environment Information Principle and used it to develop CPU, network bandwidth, and memory models.

Once we have a QoS value to track, we map it back to a constrained resource. The appropriate projection of the NVE-IP is selected, and instrumentation is added to the DVE software to measure the resource against the projection. Successive iterations of the methodology add instrumentation deeper into the DVE system software, following the software component which uses most of the resource. Using this process, we can find the controlling factors.

To illustrate, we took a tiny (five-thousand lines of code) game and made it into a model of

a DVE system. The rendering and space were two-dimensional and quite primitive, but it did have all the basic parts of a DVE: a simulation system, a synchronization system, and a user interface.

1.4.3 Performance Engineering Process

We altered Software Performance Engineering into a new process, DVE Scalability Engineering (DSE). DSE has cycles for modeling, software-change experiments, and environment-change experiments. The Modeling Cycle formalizes our modeling methodology into a process, with proper dependency management for user study and environment changes.

1.4.4 Demonstrated New Capabilities

With the three parts of our solution codified into our process, we took two of the most popular DVE systems (Torque and Quake III) with source code and significant real-world use, and applied the DSE process.

We built a numerical model of Torque's CPU usage. The primary DVE software component using the CPU was collision detection, the running time was exponential to the number of collisions. We were able to reduce the collision-related spikes in CPU usage by reducing the number of collisions. For example, using a density map (a two-dimensional histogram), we were able to identify one place in the virtual environment where users collided heavily. We placed a building there and users spread out thus reducing the number of collisions.

We also modeled Quake III's network usage. Using power consumption models of the network

communication links found on semi-modern mobile phones, we determined that 802.11b (Wi-Fi) network communication link was the best option. We demonstrated how the DSE process can be used to model and experiment with a DVE system to meet network latency or power consumption requirements.

Our contributions, especially the DSE process, provide the means to understand what performance factors affected user behavior and how to quantify these effects through correlation analysis. In other words, the DSE process facilitates an understanding of how user behavior affects DVE system performance for a particular DVE system.

1.5 Document Layout

After the introduction, a small primer on the related work is given in Chapter 2. That chapter gives a brief overview of the high-level structures of DVE systems.

We define the research problem in Chapter 3. We provide our definition of scalability and then describe the primary elements that dominate scalability within a DVE system. These elements include resources, quality of experience, and system-level limits. Chapter 3 also discusses trade-off mechanisms between quality and resource use.

Our research approach is described in detail in Chapter 4. The description covers issues related to integrating Quality of Experience and Quality of Service concerns into our performance measurement framework for the DVE Scalability Engineering (DSE) process. Then, we discuss our simulated user load experiment and how to use the experiment results to enhance the scalability of a DVE system.

In Chapter 5, we demonstrate the use of the DSE process for two different DVEs systems. For the first DVE system, Torque, we substantially enhance the system latency using only changes to the virtual environment. Additionally, the substantially higher costs of software-only changes for similar latency improvements are discussed. The second DVE system, Quake III, is modeled for its bandwidth requirements, and several types of network communication links are evaluated for mobile phone platform deployment.

The results are discussed and analyzed in Chapter 6. We provide our conclusions and identify possible future work in Chapter 7.

Appendix A provides additional details on DVE system operations, mechanisms, performance, and the system performance elements that DVE systems depend on. Appendix B provides a comparison of DVE systems' unique consistency and quality requirements against other distributed systems.

Chapter 2

Related Work

There are many different DVE systems, with differing styles, topologies, designs, data structures, network systems, event systems, and performance characteristics. Despite many inherent similarities, there is an extensive body of research addressing DVE system design and performance.

First, we open with a discussion of the basic interaction style in Section 2.1.1 followed by a brief discussion of the rendering systems in modern DVE systems. Rendering is clearly a significant DVE subsystem, and rendering requirements percolate throughout the rest of a DVE system's architecture.

Next, we will focus on the core problem of our research, scalability. We consider some early strategy, followed by some topical research. We also discuss why scalability must be considered with some context of performance.

2.1 Distributed Virtual Environments

Distributed Virtual Environment (DVE) systems are multi-user, multi-computer systems that create, maintain, and present a virtual world to users. Each user sees the world through the viewpoint of an *avatar*: a representation of the user within the virtual environment. DVE systems are used for therapy, training, collaboration, and play. For example, Baños et al. [5] discuss using virtual environment to recreate the traumatic experience to help people recover from Post-Traumatic Stress Disorder (PTSD). Another example is World of Warcraft, a Massively Multiplayer Online Role-Playing Game (MMORPG) with ten million users. Pittman's five-week study [6] found a few players online for over eight hours, but most user sessions lasted 200 minutes or less.

DVE systems are combinations of three software components: a high-rate three-dimensional graphical renderer, a physical simulation, and a network synchronization module. Each user's computer (host machine) displays the virtual environment from the avatar's point of view, communicates with other users' computers to synchronize, and collaborates with them to simulate the virtual environment. DVE system *architectures* primarily differ in the physical laws simulated, rules enforced, communication protocol and topology, and distribution of simulation workload.

In each participating computer, the DVE systems can be deployed in part or in whole, depending on the architecture. For simplicity, we will discuss the components together as the *DVE system*, as if it was a single executable. Our process doesn't depend on specific attributes of the DVE system architecture, so that simplification should suffice.

DVE systems are unique in their overall requirements and capabilities. One common element is a dynamic trade-off mechanism between resource requirements and performance. The networking protocol may accept packet loss for reduced latency. The renderer may sacrifice detail for frame rate. The physical simulator may trade accuracy for its simulation rate.

Quality requirements for the DVE system's performance vary from one situation to another, and DVE systems both suffer and exploit that variation. For example, a DVE system may shut down areas of a virtual environment when no users occupy them, not simulating physical motion, running artificial intelligence code for non-player characters, or synchronizing its current state to different computers.

The dynamic trade-offs between resources and quality a DVE system makes and the variability in quality requirements make DVE system performance a complex topic. For comparison, consider three other distributed networked systems: a multi-user video chat application, a database server, and a live collaborative word processor application. Within the trade-off and requirement axes, each takes a different point.

2.1.1 Architectures

It is important to look first at the various kinds of DVE systems currently available. While we look at what's common today, it is also important to identify incoming trends.

The inventors of the CRACK! [7] system suggest that more emphasis be made on the level, media, and methods users use to interact with each other. Instead of simple text messaging and game-level interactions, they suggest multiple media be provided to enable new forms

of communication as they are appropriate; such as video conferencing or traditional real-life communication systems like the phone or fax. Similarly, multiple integrated methods of user awareness should be available, such as integrated web sites and VR systems. Finally, they suggest that the systems be accessible from many different interface technologies, such as the web, cellular phones, and VR systems.

Frame-Rate First Person Shooters

Many DVE systems follow the pioneering work done by DIS [8] and SIMNET: a shared virtual environment with system updates at essentially frame rate. While some systems may require that each user awaits their turn to act, the experience in these FPS DVE systems is fully interactive, with time moving continuously.

The list of exemplar First Person Shooters (FPSs) is large, some examples are Doom, Quake, and America's Army. The typical setup is a first or third-person point of view of a three dimensional virtual world, a virtual environment; where walking and running, usually along with view vector control, is available. Direct manipulation of nearby objects via pushing, picking up, or special "action" buttons is usually involved in this type of DVE system. In all of our examples, and a sizable percentage of the FPS DVE systems available, some type of combat simulation is provided. Some weapons, terrain, and buildings provide cover and space for individuals or groups to fight.

As movement is apparently at frame-rate, updates between players has to be sent at similar levels. As described below, the underlying simulation is often running at a different rate, independent of the rendering system. Movement data is sent as needed, dependent on

this simulation rate. We will discuss the full complexity of this update problem in detail throughout this work.

As the continuously updated, fully-three dimensional virtual environment is fairly representative of the real world, users tend to expect certain parts of reality to be realistically modeled as well. First and foremost is collision detection. The DVE system should provide some mechanism to prevent objects from interpenetrating. Either they should be stopped or some sort of believable deformation of one or both objects should occur. Similarly, some level of physics simulation is expected. For DVE systems with earth-like terrain, on solid ground, users expect gravity to be part of the environment. Projectile trajectories should follow the traditional parabolic curve, and actions like jumping or driving off ramps should have a similar fate.

At a continuous rate, the rendering speed has to be kept fast. Users expect updates to occur as in the real world. Significant drops or jitters will be quite noticeable. Below, we will discuss the requirements and constraints of a FPS's rendering system.

Similarly, the data structures of the FPS have to be built to support the rendering and update-rates necessary to keep the DVE system usable. Because these two requirements are quite demanding and important, the data structures are often built for these first, with everything else built atop. We will discuss Again, these are discussed in greater detail below.

Turn by Turn

In comparison to the Frame-Rate FPS, other DVE systems move in fully-synchronized rotating turns resulting in significantly different requirements and constraints. Each user interacts with other users during their turn, usually with a fixed amount of reaction time allowed. The other users sit and wait until their turn comes around again.

Examples include networked chess and Microsoft's Age of Empire games. For the chess game the system needs only to validate moves and communicate them.

However, the Age of Empire game has each participant's computer simulate the result of the user action in parallel. This game also tends to make significant use of a scripting language atop the DVE system software. Unlike the simulation system of a FPS, this simulation is often rule based and uses symbolic computation instead of continuous equations for physics.

For all types of Turn by Turn DVE systems, the network latency requirements are quite relaxed. Unlike the "twitch" style motion in a FPS, the communication load is directly related to the computation of the user. Latencies as high as several hundred milliseconds are perfectly acceptable, and the bandwidth requirements are often quite low. As the majority of the DVE systems focuses on the results of the simulation, the virtual environment remains immersive despite the limited interactive activity between turns.

Rendering is similarly simplified. The simulation results can be animated during the computation, but much of that can be pre-rendered. As the rest of the virtual environment doesn't change quickly, the display doesn't need to be updated too often. For example, a chess board isn't changing anytime soon, so why re-render the same set of pixels?

Because the rendering and network loads are so light, easy simulation or programmability become the focus of the data structures. A simple object database is often sufficient, making the manipulation of the various objects used in groups easy to manage. Additionally, the speed of simulation rarely becomes an issue, leaving the scripting environment free to optimize for easy maintenance or simple programming.

2.2 Quality of Experience in DVEs

Quality of Experience (QoE) focuses on the subjective measurement of overall end-user experience. The quality of the user's experience in a DVE system is an important factor that can affect how well users can achieve their goals and whether they choose to participate.

Several classifications in this area exist, each separating qualitative elements as QoE, and quantitative elements as Quality of Service (QoS).

One particularly comprehensive attempt at defining QoE is presented by Laghari and Connelly [9]. It incorporates four domains: human, technological, contextual, and business. The human domain includes roles, demographics, objective and subjective QoE factors. Human QoE factors include psychological and physiological factors, such as intentions and behavior [10], memory, attention, task performance, and human response time.

Wu et al. [1] present a QoE Quality Framework with a focus on quantitative analysis and prediction. It is shown in Figure 2.1. Their methodology includes measurable system performance values (latency, jitter, etc.), measurable subjective user values (sense of presence, perceived system usefulness, etc.), and measurable objective user behavioral consequences.

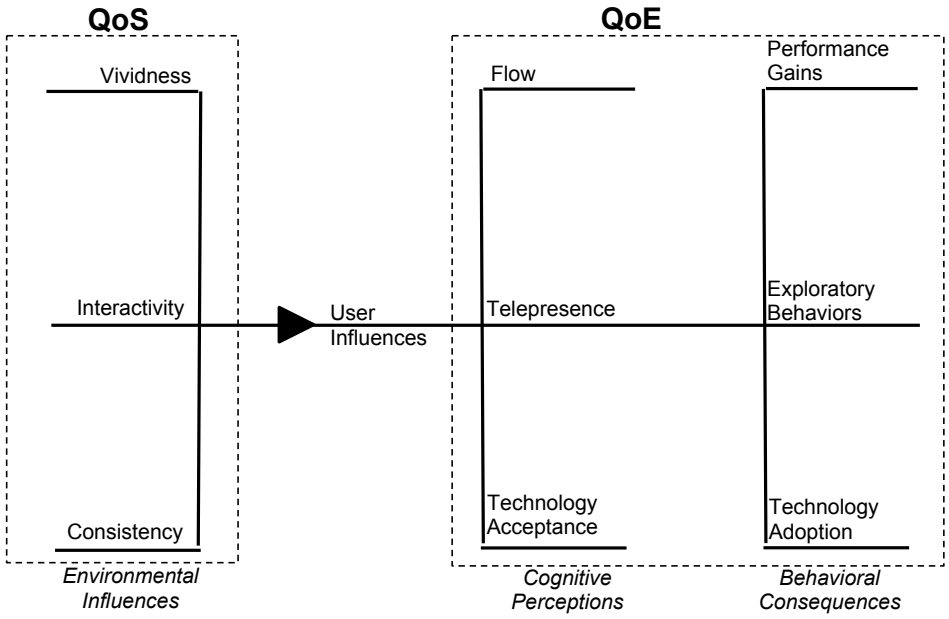


Figure 2.1: Wu et al. QOE Quality Framework [1]

The latter two are gathered via surveys.

Wu et al. correlate survey results with measured QoS variables as users perform tasks. In their study, significant correlations between factors were found. These are summarized in Table 2.1.

		QoS		
QoE Component	**Interactivity**	**Vividness**	**Consistency (Obj)**	**Consistency (Subj)**
Concentration	0.09	0.39	-0.13	0.41
Enjoyment		-0.16		
Telepresence		-0.02		
Perceived Usefulness	0.12	0.58	-0.06	0.14
Perceived Ease of Use		0.53		

Table 2.1: Significant ($p < 0.005$) Correlations found by Wu et al.

They use a "causal chain" starting with the environment, then focusing on cognition, and lastly, user behavior. Technology acceptance is a cognitive factor, with adoption as a behavioral consequence. Additional behaviors include task performance and exploration within

the DVE system.

2.2.1 User Behavior and Participation

There are three primary behaviors: (task) performance, exploration, and adoption of technology. They can be predicted from the user's cognitive perceptions, which are covered below.

Task performance is a metric of how well users complete tasks in a virtual environment. Task performance can vary with system performance parameters. Watson et al. [11] conducted three experiments to study task performance under variations in system performance. In each experiment, users had to grab a moving object in a virtual environment, and accurately place the object. The frame rate varied across these three experiments. The first experiment varied the frame rate and the second experiment varied the variance on the rate. The third experiment also varied the frame rate, but with much higher frame rates. Measuring the number of attempts needed to grab the object, time to grab, and time to place the object, they found some significant effects. Grasp time was significantly affected by both frame rate and latency for the first two experiments at lower frame rates. The number of grasps was also significantly affected by latency. For all levels tested, the time to place the object was significantly affected by frame rate and variance.

More generally, DVE system designers will have to determine the important tasks they want users to accomplish in the DVE system, and the desired metrics for them. In a training simulator, this may be based on how well the users execute the tasks in real life. Some DVE

systems make some tasks particularly difficult, so that successful executions rarely occur (e.g., a three-point shot in a basketball game).

Exploratory behaviors can be spatial, they can include experimentation with tools, or attempts to determine how stringently rules are enforced.

The basis for predicting user participation comes from the Technology Acceptance Model (TAM) [12] (Chutter [13] provides a review of its evolution and validation since its creation). This model treats the user's intention to use the system equivalently to actual usage of the DVE system. The user intent prediction comes from two factors: the user-perceived ease of use of the system, and its user-perceived usefulness. These two factors are part of the cognitive perceptions in the causal chain.

2.2.2 User Perceptions and Technology Acceptance

A user's cognitive perceptions, includes their feeling of flow, telepresence, and their perceptions of using the system. We will cover that last element below, as part of its enclosing TAM model.

Flow is a combination of concentration and enjoyment, resulting in "the holistic sensation that people feel when they act with total involvement" [14]. Empirical findings in research [15, 16] find three metrics to be significantly relevant for systems like DVE systems: concentration, intrinsic enjoyment, and sense of control.

Telepresence, also sometimes just called presence, is the user's feeling of actually being in the virtual environment. The psychological elements of the feelings experienced by a person

essentially moving from reality to a virtual environment have been studied for use in both training and treatment. Schuemie et al. provide a good survey [17]. There are many theories about presence. Some elements of a comprehensive feeling of presence include social richness, realism, and autonomy. Surprisingly, studies summarized by Schuemie et al. [17] show little correlation between (tele)presence and task performance. A shorter study by Wu et al. [1] shows an inverse non-correlation between presence and any QoS factors.

The Technology Acceptance Model (TAM) [12, 18] is a framework for predicting user intent, specifically on use of a computer system. TAM uses surveys with questions on Likert scales. For example, "I find it easy to get the electronic mail system to do what I want to do." [18], with a user responding with a number from one to seven, where one indicates the user strongly agrees with the statement, and seven indicating that the user strongly disagrees. Using these surveys, users' perceptions of system usefulness and system ease-of-use are measured. These studies have validated [13] the theory that accurate measurement of these two variables can predict how much users will use the system.

Wu et al.'s framework [1] correlates TAM factors to QoS values. These measurable system attributes include system performance.

2.2.3 DVE Systems and System Performance

DVE systems form the "environment influences" part of Wu et al.'s framework. There are three categories for environmental influences: vividness, interactivity, and consistency. Vividness indicates the number of sensory channels open to the user (breadth), and their

quality (depth). Interactivity includes latencies and jitters in system response to input, the range of available input, and the usability of the input system. We will discuss consistency below.

The level of visual and sound realism presented by the system, its frame rate (and variance), and the accuracy of the physical simulation contribute towards *immersion* — the quality of the user sensory experience [17].

Surprisingly, the QoS parameters of the framework include usability metrics. This links to TAM [12, 18, 13] and its Perceived Ease of Use parameter. No metrics were given by Wu et al., and common usability metrics tend to be aliases for user task performance. However, some usability-relevant parameters may include the number of clicks to achieve each task or lengths of lists that the user has to search to select objects.

Consistency indicates both the similarity of each user's view of the virtual environment and how well the rules are enforced. Two different users may simultaneously see an object in different states. The users may find out when they both try to act on the object. For example, both may try to pick it up. One user gets it. The other sees the object disappear from their hands. Alternatively, a DVE system may weaken its enforcement of some rules when overloaded. Some collision detection mechanisms may use simpler-but-faster algorithms at times. The user may observe objects that didn't touch or bounce off of each other, or a normally vigilant virtual referee stops enforcing some game rules when the system becomes loaded.

DVE system performance values indicate speed (latency) and regularity (jitter) of a service. DVE system performance issues include how quickly a user gets synchronized to other users,

how quickly they see the DVE system respond to their input, and the accuracy of the results.

Controlling this last category of issues is fundamentally different than QoE. As we mentioned initially, the system performance depends on user behavior. Wu et al. [1] only covers the inverse. Additionally, the domain of this problem is different, focusing more deeply on the software, protocols, and representations of the virtual world instead of the users within it.

2.2.4 Position Distribution Metrics

Existing performance analyses of DVE systems tend to use system metrics for their evaluation, and user clustering models for load simulation. Lui and Chan [19] used three user-position distributions for modeling: *uniform*, where (x, y) pairs were evenly spaced across two dimensions; *skewed*, where most avatars are within a single quadrant of the space; and *clustered*, where k clusters are randomly generated with (x, y) centers, having uniform distribution in the space. Avatars are assigned evenly across the k clusters and assigned a uniformly-distributed (dx, dy) distance from the cluster center. They used these distributions to evaluate partitioning users across multiple servers. Morillo et al. [20] used the same position distributions to evaluate another partitioning algorithm.

These position distributions were static, with no effects of changes in distribution considered. However, Morillo et al. did consider variations in server load, as part of a larger partitioning algorithm used for load balancing. Server load was determined by the speeds and Area of Interest (AoI)-visibility of avatars occupying them. Lui and Chan showed the partitioning problem to be NP-Hard. For analyzing Quake, Abdelkhalek et al. [21] used recordings of

players and played them back directly, using a modified client for playback. When played back at the same number of original players, the load simulation provided realistic human behavior. However, there was no attempt to adjust or abstract the behavior for additional players.

The DINE framework from Ta et al. [22] assumes dynamic latency conditions, and collects latencies to help evaluate latency-reduction methods.

2.2.5 Task Performance

Watson et al. [23] evaluate the effects of frame rate on task performance. The research evaluated both frame rates and variations in frame rate as dynamic factors affecting user task performance. Their 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 FPS games, Quax et al. [24] provide an analysis of the effects of latency and jitter on user performance, as measured by their in game-score in Unreal Tournament 2003. They found that users had noticeable impairment when the round-trip-time (RTT) surpassed 60 ms. Bhatti and Henderson [25] 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 [26] investigated the use of metrics in distributed systems, from devel-

opment through operations. They examined 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 types of service defined. RFC 2211 [27], allows for controlled load service, acting as a best effort link over a lightly loaded link. This reduces loss and jitter significantly, leading to a more stochastically stable performance for DVE system synchronization. The DVE system will provide more predictable performance due to the additional stability in its synchronization system. RFC 2212 [28] also specifies guaranteed service covering hard bounds on delay, in turn giving guarantees on bandwidth. Systems providing "hard" guarantees may have to sacrifice average throughput to ensure that they make their commitments, 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 [29] to describe soft or hard guarantees for latency, bandwidth, and jitter.

2.3 Performance Measurement Tools

For measurement, common tools include `oprofile` [30] and `dtrace` [31]. `oprofile` takes regular samples of the program counter (e.g. `%rip`) and determines which functions' code the counter was in each time. Its predecessor `gprof` [32] used a modified executable with code at function entry and exit. The call graph `oprofile` reports is an estimate, and may require some user interpretation to determine the real graph. It also does not require

instrumentation of the code — just an unstripped executable.

Both of these profilers are useful for determining a singular bottleneck in the system. For example, when the running time listed for a function is taking longer than desired to complete. However, this tells relatively little about the overall performance behavior of the system in its various dynamic states, because that information is lost in the data aggregation. The overhead incurred by the instrumentation is relatively static, and there is little a developer can do to adjust it.

The information provided uses a function-based performance model. The application state and input are not connected to the work they incur, leaving the developer to determine what happened when two reports of the system differ.

Some ability to see the application state and statefully observe the system can be beneficial. `dtrace` [31] uses `awk(1)`-like syntax to process events. `dtrace` takes a pattern- and internal-variable- based event specifications to trigger the execution of processing actions. This functionality enables developers to specify which probes — such as invocations of specific functions with specific values or properties of their arguments — they want to observe, and to collect and aggregate the data as they see fit. Probe points can also include function invocations in the kernel caused by an application, such as page faults. `dtrace` lets the developer designate their own probe points. Using an interface declaration-style syntax, they can specify which events they want to add to the set available. `dtrace` generates a header and object file for programs generating the events. The header includes C macros to invoke with arguments. Each macro invocation can fire an event (if listened to: the macro expands to a conditional predicated on there being a listener) and includes its arguments.

`dtrace` has three primary limitations. First, no floating point values are supported: the event handlers aren't capable of floating-point arithmetic. Second, the data analysis and aggregation has to occur within the script. Finally, the scripts cannot save their output to a disk file for later analysis.

When an event is being listened to, the overhead involved does involve a half-context switch: the process goes into kernel mode to save the data, and then returns back. The overhead of switching to a new process is saved. Factors controlling the amount of overhead involved, such as when the context switch occurs relative to the event, and whether the relevant memory addresses are already cached, are not controllable.

2.4 Performance Management Processes

One of our tools is a process derived from Software Performance Engineering (SPE): an iterative, model-based software engineering process. Smith and Williams describe it in detail in [3]. For a process more closely tied to the UML metamodel, the ROPES process by Douglass [33] also uses an iterative, model-based engineering process, with a stronger focus on automatic code generation.

2.4.1 Software Performance Engineering (SPE)

Software Performance Engineering (SPE) [3] is a iterative process for evaluating design alternatives and performance objectives. SPE is UML based, and it uses UML notation and terminology for the process, models, and analysis.

Figure 2.2 shows the SPE modeling process.

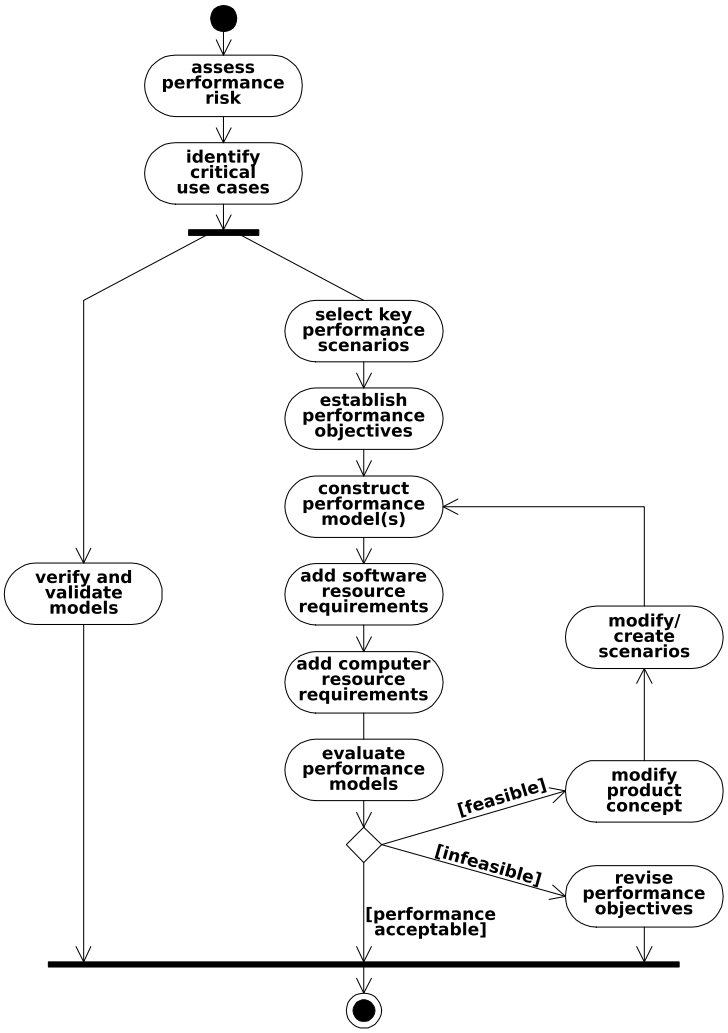


Figure 2.2: SPE modeling process

SPE has nine primary steps:

- *Assess Performance Risk:* Determine the potential impacts of undesirable performance, and the likelihood of their occurrence. Use them to determine an appropriate amount of effort to expend on SPE activities. For example, a novel system with a business critical function may require substantial SPE work, if the business function has real-time throughput requirements. In contrast, a system similar to those developed

by this team, with similar requirements as before, may need only light SPE work to validate.

- *Identify Critical Use Cases:* Identify externally-observable use cases that determine performance acceptability. A use case includes one or more *actors:* people or other systems, requirements for the system, a context for it, and a set of related scenarios.

- *Select Key Performance Scenarios:* From the identified use cases, select specific and measurable scenarios that will serve as performance metrics.

- *Establish Performance Objectives:* For each selected scenario, select at least one quantitative objective. Specify the required conditions for each objective. For example, a throughput objective should include a workload description.

- *Construct Performance Models:* Construct *execution graphs* from the scenarios' sequence-diagram representations. Their format is described in detail below. These graphs contain quantitative relationships between workload and resources taken.

- *Add Software Resource Requirements:* Determine resource requirements in the software domain. Examples include messages, queries, or events. This forms a decomposition of the system's work into smaller, more homogeneous units of effort.

- *Add Computer Resource Requirements:* Map the software-domain resources into computer requirements. Messages may get mapped into units of bandwidth, queries into disk seeks, and events into CPU time.

- *Evaluate Performance Models:* Solve the execution graph for the desired contexts and

compare the computed estimates against the objectives. The process is complete if estimates meet objectives.

  - *Modify Product Concept:* Look for feasible, cost-effective alternatives to satisfy the missed objectives.
  - *Revise Performance Objectives:* If no feasible and cost-effective alternatives apply, re-evaluate the objectives. Stakeholders should be consulted.

- *Verify and Validate Models:* In parallel with the primary process, check the models in two aspects. First, verify that the models are being built correctly. The estimates and model structure should all be reasonable. Second, validate that the proper models are being built. Specifically, verify that all relevant parts of the system are included in the model.

  This step requires measurement of a prototype or system simulator. The measurements are compared against the models' estimates and assumptions.

Execution Graphs

To model the system, SPE uses *execution graphs:* flow-charts of major steps of the program's execution, each annotated with best-case resource usage. An execution graph can be generated by tracing over the vertical axis of a UML sequence diagram. Then, some hierarchization is applied to provide the appropriate level of abstraction.

Figure 2.3 shows an execution graph of an Automated Teller Machine (ATM) scenario. On the left is the top-level scenario, with a user inputting their ATM card and PIN, executing

n transactions, and then completing them. Items in the scenario can be broken down into smaller sub-scenarios, such as "Process Transaction" (Figure 2.3, right).

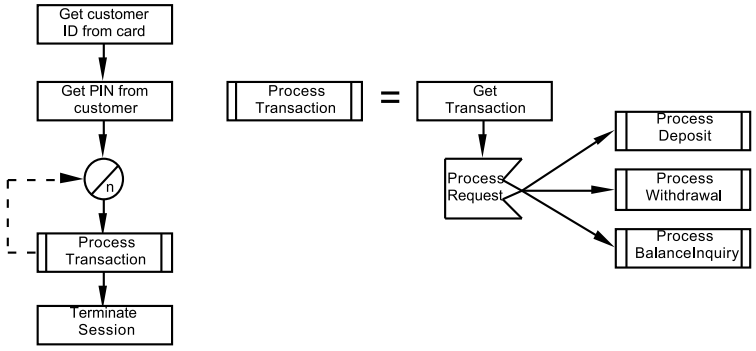


Figure 2.3: Example SPE execution graph [3]

Each element in the execution graph is then assigned a software-resource vector. As an example, the vector could have the format: (messages, device I/Os, CPU time). Then, the elements of Figure 2.3 could have values $Card = (0, 1, 0)$, $PIN = (0, 1, 1)$, $Process = (2, 1, 5)$, and $Terminate = (0, 1, 1)$. Then, one can calculate software resource requirements for different traces through this scenario with simple vector math. For example, a three-transaction scenario would simply be: $Card + PIN + 3 * Process + Terminate$.

Each software resource is converted (projected or applied) into computer resources. A unit of CPU usage would be converted into a number of CPU-seconds of time. A message unit may have networking and CPU components. A device I/O may include an amount of time running an embedded card-feed motor. In the vector space we mentioned before, a transformation matrix can be used to convert from software to computer resources.

2.4.2 Rapid Object-oriented Process for Embedded Systems (ROPES)

The Rapid Object-oriented Process for Embedded Systems (ROPES) process [33] uses standard UML meta-model notation for its semantic framework and notation, and has four (serious) phases, as shown in Figure 2.4. ROPES is a general, iterative process that can be used for primary software development. It includes performance and safety as requirement domains alongside traditional functional ones. ROPES is intended for use in embedded and real-time contexts.

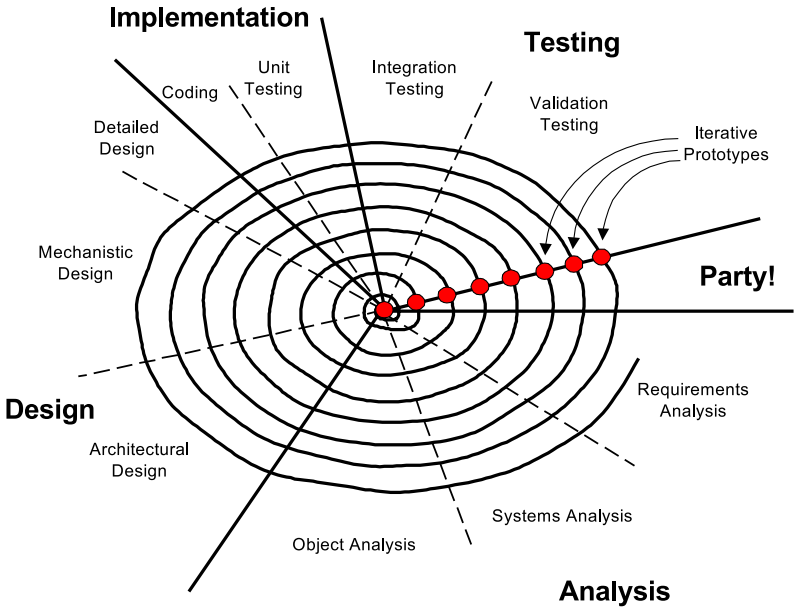


Figure 2.4: ROPES process diagram [33]

The four primary phases form a single iteration. Earlier iterations work with small prototypes of the system. The prototypes evolve over later iterations into the completed system.

Analysis Phase

The analysis phase determines what to build. It starts with requirements analysis, determining what the customer wants from the system. This includes both use cases and constraints on the system.

Systems analysis follows, determining key algorithms, identifying the large-scale components of the system, and partitioning them into electronic, mechanical, and software categories.

Finally, object analysis identifies key objects and classes with their important properties. Only those objects and classes "essential to all possibly correct solutions" [33] are identified in this sub-phase. This includes both critical behavioral (algorithms) and structural (data structures) elements.

Design Phase

The design phase is next. The model of the system-being-built is constructed and one of the design alternatives selected. Traditional concerns for design include: concurrency, task scheduling, inter-process communication, resource ownership rules and error handling policies. This phase will refine the results of the analysis phase's object analysis to a final design (at least for this iteration).

First, an architecture for the system is selected. This includes patterns for safety, fault tolerance, and error handling. Traditional "architectural" concerns, such as the number of tiers and each ones' responsibilities, are also selected.

Next, a mechanistic design operates over smaller units of the system. Individual groups of

components: six to twelve objects, have their collaborations decided. Specific design patterns are selected.

Finally, detailed design completes the phase. The individual structure and behavior of classes is selected. The results of this sub-phase include the object model, state charts, activity diagrams, and pseudocode.

Translation Phase

ROPES strongly suggests automated model-to-code translation. Even when available and used, some manual intervention may be necessary. The detailed design completed in the last phase is converted, by hand or algorithm, to a set of runnable software components. Unit tests are also created.

Testing Phase

The generated/written code has integration and validation testing executed in this phase. Validation tests are completely "black-box." Safety testing is not "black-box" since something typically has to be intentionally broken to verify proper responses.

Ad-Hoc Engineering

Many practitioners do not use a formal process for performance engineering. Instead, a combination of four kinds of performance tests are often used.

- *Performance Regression Testing:* To track performance changes due to changes be-

tween releases.

- *Performance Optimization and Tuning:* To support ongoing optimization and tuning activities.
- *Performance Benchmarking:* To verify performance on like-deployed hardware with realistic workloads.
- *Scalability Testing:* To test the system performance and behavior under heavy load.

Similar to the process, the models chosen for the activity may also be ad-hoc. We will cover major methods of modeling in the following section.

2.5 Performance Models

There are many representation and reasoning forms for a performance model. Regular mathematics and statistics can and often do work, as well as execution graphs. However, sometimes more is required.

2.5.1 Queuing Networks

Queuing networks (QNs) are graphs representing the processing of a task. Queued centers represent system components that do work on *jobs*, such as a CPU, database, or disk. Edges connect queued centers to each other, and to the other types of items in the graph: sources, sinks, terminals, and delay centers. *Open* QNs have jobs coming from sources and eventually ending in sinks. *Closed* QNs have a fixed amount of job-sources (e.g. users) that

continuously circulates. The users interact with terminals that serve as user I/O for both the job specification and result output. Delay centers reflect delay on a job.

Queued centers, sources, users, edges, and delay centers have parameters that define their behavior in the QN. Sources have arrival processes (e.g. Poisson with a fixed value for λ), user types have classes and distributions, and edges have routing probabilities when plural. Queued centers have service-time distributions. Delay centers have similar delay-time distributions.

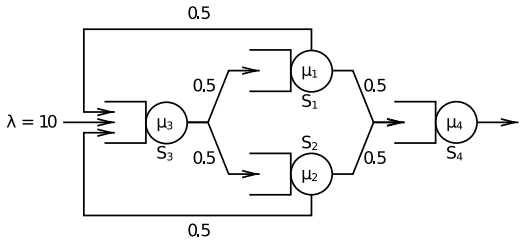


Figure 2.5: An example Queuing Network by Balsamo [34], labeled with proportions of work on each arc

Cortellessa et al. [35] give a detailed overview in the context of software performance analysis. Balsamo et al. [34] provide a transformation algorithm to convert annotated UML models to Queuing Network (QN) models.

2.5.2 Stochastic Petri Nets

Petri nets are directed bipartite graphs with *tokens*, *places*, *transitions*, and *edges*. Places and transitions are nodes in the graph. A place contains tokens. When a transition fires, a token moves between the connected places. Tokens can represent work items, places can be states of these items, and transitions can be events that cause state changes. The example Petri net in Figure 2.6 has three tokens represented as black dots, three places ($p1$,\$p2\$,$p3$),

and two transitions ($t1$, $t2$). A token may move along the edges from $p1$ to $t1$, $p2$, $t2$, and finally $p3$.

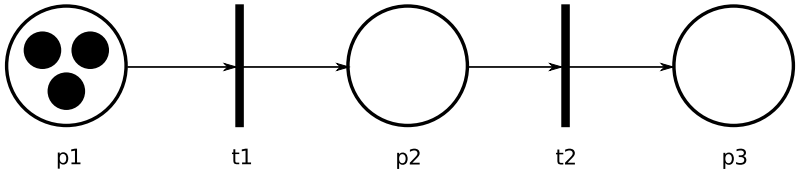


Figure 2.6: An example Petri Net by Cortellessa [35]

Petri net transitions are instantaneous. A Stochastic Petri Net (SPN) is a Petri net with transitions that take random time to complete. The times are described as random variables with their own distributions. Transitions in Stochastic Timed Petri Nets (STPNs) are all like this. Generalized Stochastic Petri Nets (GSPNs) have a mixture of these timed transitions and instantaneous transitions. Kartson et al. [36] provide a unified theory of GSPNs and examples from many applications.

2.6 Human Player Simulation

Simulated human players are often called AIs or NPCs (Non-Player Characters). They generally do not aim to convincingly replace human beings in the system, but act in places where humans are desired and unavailable. These include acting as virtual shop keepers, opponents, bystanders, or teammates. Some interactions can include menu-based chat. Jie et al. [37] give a short overview in some of the techniques used. Buckland [38] covers these topics in detail.

2.6.1 State Management and Reasoning

Virtual actors tend to have complex internal states, and move through these states by sensing the outside world, and acting upon it. A primary means of representing the top-level behavioral logic of a NPC is through a finite state machine. Sometimes, a progress-towards-goal model is more appropriate.

Finite State Machines

Finite State Machines (FSMs) form good representations for many types of NPCs. We use them in both our Torque (Section 5.1.2) and Quake (Section 5.2.1) simulators. For Torque, we randomly select a behavior (acting here as a non-deterministic finite automata) and execute a deterministic finite automata that implements the behavior.

State transitions are typically driven by achievement of particular objectives, such as reaching a position, an opponent's demise, or acquiring a specific item.

Goal Driven Behavior

Goal driven behavior starts with a selection of the best *strategy* to execute given the current sensed state of the world. A strategy is a hierarchy of goals, where peer goals are idempotent, and leaves in the hierarchy are directly executable tasks.

Figure 2.7 shows a top-level goal to acquire a tool. Once this top-level goal is selected, the two children are considered. The "Buy Tool" sub-goal cannot activate until "Get Gold" is complete. Similarly, the "Mine Gold" leaf goal cannot activate until "Go to Mine" is

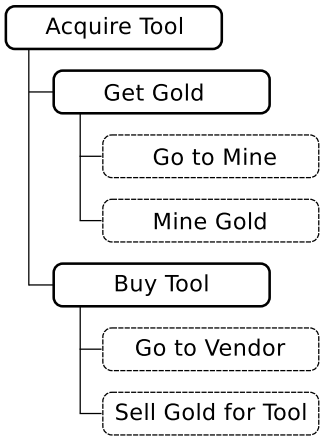


Figure 2.7: A single goal's hierarchy

complete. The first leaf goal activated is then "Go to Mine," followed by "Mine Gold," then "Go to Vendor" (after having completed "Get Gold" and moving to "Buy Tool"), and finally "Sell Gold for Tool."

2.6.2 Movement

Once we have determined the top-level reasoning of the NPC, we must implement the actual activities in the DVE system (such as moving around). Realistic movement can be non-trivial, and poor implementations can drastically affect the realism of the NPC.

Chasing

The simplest means for an NPC to move realistically is to follow a human. This way, the movement vector is simply the difference between the NPC and the human's positions, clamped to an appropriate velocity maximum. Once an NPC selects a human to follow, it may alter its speed in the pursuit.

An NPC can get stuck. The shortest path to the human may not be the same path the human took, and may instead have an obstacle. There can be walls, holes, or trees. The Torque simulator has a small re-route facility to go around buildings in town, and uses a simpler strategy for the surrounding space. Whenever the Torque simulator has chosen a target and a chase strategy, it begins firing at a randomly-selected interval. When out of town, the only obstacles are large, and the random-firing will cause backwash that will eventually move the NPC out of the stuck-state (or kill it, causing a re-spawn in a new location).

For complex virtual environments, the chasing strategy may be to select a target and path-find their way there.

Path-Finding

The route to a destination may be complex, and the task of finding this path can be involved. Path-finding takes a representation of the virtual environment and uses a simple search algorithm to find a route to the destination. The representation can be a directional graph of points, where straight-line movement is possible between points. For this mechanism to work properly, straight-line paths between any place in the virtual world and some point in the graph must be possible.

The current location and destination points are matched to the closest points in the graph, and a heuristic search, such as A^* can be used to determine the route through the graph. An A^* search recursively chooses the candidate closest to the destination point and back-tracks if the route is untenable. Hayes-Jones [39] gives a detailed description.

Swarm Movement

A swarm system may be appropriate for many NPCs working together. Swarm systems make sure that these groups do not interfere with each other by constant collisions or interference. It also provides basic coordination, and amortizes part of their computational load across many NPCs.

Finney [40] provides an introduction, starting with three rules:

- The members of the swarm should avoid bumping into each other (collision avoidance).
- Each member of the swarm should attempt to maintain position with respect to the center of the swarm (swarm centering).
- Each member of the swarm should attempt to match its velocity with the average velocity of the other members (velocity matching).

A leader is assigned to all members of the swarm. This leader determines the primary activities of the swarm, using the mechanisms described above. Each member then follows that leader at a fixed distance, and avoids other members of the swarm.

2.7 Discussion

We have completed a short overview of work in DVE systems, considerations of quality within DVE systems, means to measure computer systems performance, processes for managing computer performance, models for computer performance, and methods for simulating

human users. This overview should suffice for our immediate needs. However, additional information is provided in appendices. Appendix A provides substantially more information on the internals of DVE systems, and the performance of the components DVE systems depend on. The unique properties of DVE systems, in terms of their quality and performance trade-offs, is discussed in terms of comparison with other distributed systems in Appendix B.

Chapter 3

Problem Definition

Scalability is the relationship between the number of users able and willing to simultaneously access a DVE system, the quality of the virtual environment, and the computer resources required. The performance of a DVE system and a user's behavior within the virtual environment have a symbiotic relationship, which directly affects scalability. The DVE system's performance workload — and its conjugate, performance — depend on the users' behaviors. The more users congregate, interact, and view the virtual environment, the more the DVE system has to simulate physics, detect collisions, and synchronize the environment's state to them.

As a corollary, the user's behavior depends on the performance of the virtual environment. As Watson et al. [11] state, users become less proficient at performing tasks as system performance degrades. Users will adapt by changing their behavior or not participating.

The symbiosis between the DVE system performance and user behavior prevents traditional

performance engineering approaches, like Software Performance Engineering [3], from creating an optimal balance between available computer resources, DVE system performance, features of the virtual environment, and user capacity.

That road-block caused by the dependency between the DVE system performance and user behavior creates challenges in identifying, understanding, or changing the scalability of a DVE system. We will first cover three primary sources of limitations in scalability, and then discuss each of these three sources/challenges more in depth.

3.1 Limits in Scalability

Scaling a DVE system is confined by three limitations: hard limits, resource limits, and quality limits. When they apply, limits can restrict DVE systems in two ways: a DVE system may become unavailable when a limit is hit or performance can degrade and make users less willing to join.

Memory provides an excellent example of all three limits. We will trace the memory use for all three types of limitations.

3.1.1 Hard Limits

Hard limits are design-imposed restrictions in an applicable standard, hardware architecture, the DVE system software, or the operating system. Hard limits can be sufficiently difficult to change that avoidance is the only mitigation strategy.

Examples include the sixteen-bit limit in IP port numbers, width of the address bus, size of object IDs in the DVE's network protocol, or maximum allowed simultaneously open file descriptors.

For memory, the address space size of a machine can be a hard limit. A thirty-two bit process can normally only address four gigabytes of memory, with some significant portion reserved for the operating system. Attempts to allocate more memory will fail, and very few systems handle out-of-memory situations well. Unless virtual memory is enabled, risking other limits (which we will discuss shortly) threatens system availability.

Hard limits are generally well understood, and traditional software engineering or operations processes can manage them. We will briefly review them here for completeness. Round-robin use of multiple IP addresses can reduce IP port pressure, scene-graph segmentation across processes can reduce address space pressure, and archive-file formats can reduce file-handle pressure. Occasional process restarts can reduce pressure when the software can't directly do so.

The other two restrictions, resources and quality, have a more complex and fluid impact on DVE scalability, and form the primary concerns.

3.1.2 Resource Limits

Constraints in computer resources can restrict the DVE system's scale in terms of availability or quality. Resource limits can constrain DVE systems to nonnegotiable availability limitations like hard limits. However, the resource limits tend to "trade" quality for resource

availability.

For example, memory has tiers, starting at the processor caches and ending at the swap partition on disk. As the DVE needs more memory, it expands down the hierarchy from the fastest mechanisms to the slowest. The later tiers take substantially more time to read data, adding to the latencies of the system.

Most other primary resources are rate based. Two primary resource rates are processor clock rate and network I/O rate (bandwidth). Some systems may also need vector or general-purpose graphics processing (GPGPU) units. These specialized units have similar bases in operation-execution rates. Some DVEs may work in power-sensitive environments, such as on mobile phones, and the power consumption rate can be just as important as CPU or network rates.

Some quality metrics will depend on parameters of system resources. Specifically, rate parameters of those resources. Most capacity-based resources have some virtualization that exchanges capacity for latency.

For example, a DVE's simulation rate depends on the rate of the processor. Higher simulation rates result in more accurate physical simulation, quicker collision detection, and better task performance rates [11].

A DVE system runs the simulation in discrete steps of virtual time. The simulation run-time depends greatly on the state of the virtual environment and substantially less on the length of the virtual-time interval simulated. The DVE system matches the virtual-time interval to the run-time of the simulation routine. This matching maximizes the simulation rate and

keeps the simulation from falling behind.

The virtual-time matching is a primary quality-to-resource trade-off in DVE systems. Unfortunately, the matching complicates performance analysis because it binds quality and resource together. If we want to compare algorithms or design decisions in a DVE for their performance impact, we have to make sure they deliver the same quality. Alternatively, we have to control performance when comparing quality.

To separate actual utilization from quality compromises, we introduce a new term. The resource usage of a system at fixed, maximum quality is the resource *pressure*. This term acknowledges when resource usage may be "relieved" by reducing quality. Other means of resource-pressure relief, such as faster algorithms, are often preferable to reducing quality.

3.1.3 Quality Limits

Quality limits can reduce a user's desire to participate. Poor quality can frustrate the user. Quality limits are measured by Quality of Service (QoS) and Quality of Experience (QoE) metrics. QoS metrics focus on physical and physiological elements of the user's interactions with the DVE system. QoE metrics focus on the emotional, social, and cognitive elements of a user's interactions with the DVE system.

Quality of Service metrics include the usability of the system's controls, the ranges of those controls, the temporal and logical consistency of the environment, and the system's latencies [1]. Direct measurement of the control devices or DVE system state can measure QoS. For example, unresponsive controls or inconsistencies in the DVE system can be QoS quality

limits. QoS can worsen as additional users participate, when the system has more work and may sacrifice QoS.

QoE metrics can include the user's role, demographic, business, and contextual motivations for participating [9]. QoE metrics can also include subjective and objective elements of the user experience, such as feelings of flow, telepresence, perceived usefulness, and ease of use [1]. User and stakeholder surveys measure QoE. QoE quality limits include user's feelings that the system is hard to use, that they don't have motivations for using the system, or that they aren't able to "get into" the flow of the DVE system. One of these limits may be more likely to apply when more people are in the system, making them scale-related. For example, if there are key areas of the virtual space that get filled up with users, other users may feel that their participation would be ineffective and stop using the DVE.

A principal theory behind a user's willingness to participate is the Technology Acceptance Model (TAM). Chutter [13] provides an overview of its origins and development. TAM considers the factors and the cognitive processes that lead to the use of a system. Technology adoption is a predicted result of key interest to work in scalability, because: it can help predict how many users are willing to use the DVE. Figure 3.1 shows a refined, final version of TAM.

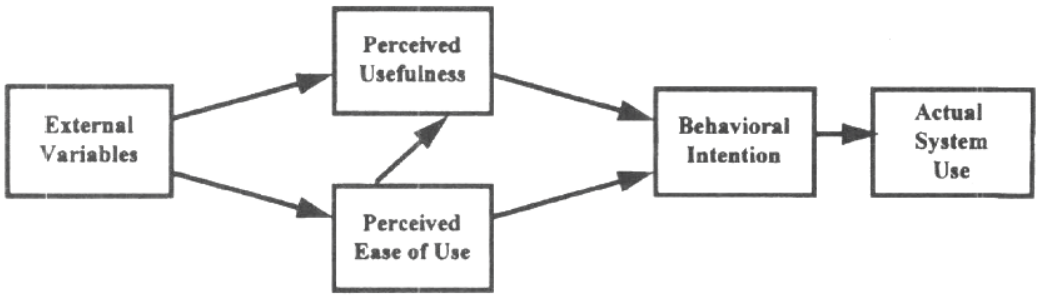


Figure 3.1: Final version of TAM from Chutter [13]

In one of the many studies related to TAM, Davis, Bagozzi, and Warshaw [41] studied 107 users twice, fourteen weeks apart. They found that the users' intention to use the system (an IBM graphics package) had a high correlation with the users' perceptions of both the system's usefulness and ease of use. Adams, Nelson, and Todd [42] replicated the experiment on MBA students using WordPerfect, Lotus 123, Harvard Graphics, and other applications. TAM maintained its consistency in predicting and explaining system adoption.

The users' activities within the virtual environment are often important to the DVE system's goals. For example, a training simulator DVE system succeeds only if its users' activities correspond to their training. If users continue to log in after a certain quality threshold, but stop acting or interacting in the system, we may consider the DVE system outside its acceptable quality. Wu et al. [1] provide a theory, called the "Causal Chain," (Chapter 1, Figure 1.1) of QoE that includes technology acceptance and users' behaviors within the virtual environment. Wu et al. used correlation analysis for connecting together different factors in the Causal Chain. Section 2.2 discusses the theory.

3.2 Determining the Scalability of a System

Current methods for testing a DVE system's scalability are either inaccurate or expensive. All involve measuring the DVE while under some sort of user load. User load simulation methods include simple protocol simulators, scripted users, and groups of user study participants. All three methods have issues limiting their accuracy or applicability.

3.2.1 Simulating Load

Protocol simulators are simple, nearly stateless programs that can generate messages in the DVE system's internal format. The simulators provide randomized user load input to the DVE system. They only maintain enough state to randomly move in uniform, skewed, and clustered distributions. Morillo et al. [20] and Lui and Chan [19] both used this method to evaluate partitioning algorithms for DVEs. These simulators provide some basic load for evaluating single algorithms, but provide insufficient realism, compared to the humans they proxy, for evaluating an entire DVE system.

Scripted users are program scripts that emulate users. Typically, these scripts are separate instances of the DVE system acting in a user-simulation mode. The name comes from the scripting languages commonly implemented in DVE systems for controlling high-level user behavior. Scripted users implement simple behaviors: path-finding, finite-state machines, and goal-seeking. They provide substantially more accurate user load simulation than protocol simulators thanks to their internal state management.

Scripted users are traditionally used as either complementary characters in the virtual environment, or as moving "unit tests" for the DVE system. They have not, as of yet, been made to accurately replicate expected user behavior.

For accuracy, some DVE systems test with a large pool of users. Weilbacher [2] discusses the test process used for Gears of War 3, using over 100 professional testers. This type of test generates very good data on all three kinds of limits (hard, quality, and resource), with relatively minor concerns on how testers will use the DVE during the trials differently than

regular users. Since the tests are shorter than a user's normal experience in the DVE, the load tests risk having testers use more short-sighted activities than regular users. Regular users judge their actions by long-term consequences, which may have a longer time span than the load test.

Apart from the accuracy concerns, using a large population of testers requires coordination and time. As a result, substantial preparation and planning is necessary to run these types of tests, making them longer and more difficult than the other two types of user load tests.

3.2.2 Measuring Performance

Once we have load, we can start evaluating the DVE system's performance. The DVE system typically measures its own performance. Torque [43] has a built-in profiling system. Weilbacher [2] describes measurement through process and database instrumentation. The instrumentation includes latency for a single simulation run, database connection timeout rates, processor utilization, and transaction rates.

These metrics do not measure quality, but rather measure process health and load applied to support systems, such as the task scheduler or database. Quality metrics described by Wu et al. [1] have not traditionally been measured in load tests. At least part of the reason is a poor understanding of how quality metrics are relevant for user experience.

Even with metrics able to measure quality, if a system's quality proves insufficient, it is not always clear how to improve it.

3.3 Understanding Current Scalability

If the challenges in user load simulation are properly addressed, DVE system designers will be able to measure their current scalability. However, that is not enough. The scalability is a result of many factors connecting the contents of the virtual environment, design and implementation of the DVE system software, and computer resources. Without understanding these factors, DVE systems have no ability to respond if the current scalability is insufficient and would not be able to take advantage of any unexpected additional resource.

As users enter the virtual environment, the DVE system should "understand" what effects the users' actions will have on the DVE system components. If users congregate into small clusters will they load the collision detection component? If users smash virtual objects into many debris, will the scene graph grow substantially and tax the synchronization system? Will there be enough processor power for the collision detection, memory for the scene graph, and network bandwidth for synchronization?

The *performance structure* of the DVE system answers these questions. It is the set of factors that determines the DVE system's performance. These factors can be in any part of the system and cross between parts, from the virtual environment to the DVE system software, to the underlying computer resources.

Different factors can affect which parts of the DVE system software run, how frequently they run, and the sizes of data structures.

The performance structure can explain the performance consequences of decisions in the design of the virtual environment or DVE system software. It can also explain the inverse,

the utilization of a resource by the software component using it, and the parts of the virtual environment invoking the software component.

Unfortunately, the performance structure for any particular DVE system is not well known, and little research has provided results for DVE systems in general. Singhal and Zyda [4] describe the Networked Virtual Environment Information Principle that relates resource usage and the DVE system software. The Principle relates them in a vague magnitude of overall resource cost, but doesn't provide any per-resource breakdown means of understanding the effects of optimizations. We will cover this in detail in Section 4.2.

In contrast, extensive research exists in the general field of software performance analysis, and the research includes per-resource and optimization modeling. Algorithm analysis can help estimate the resource utilization of individual software components. Instrumentation can generate a broader view of the DVE system's performance.

3.3.1 Algorithm Analysis

Traditional algorithm analysis converts an algorithm into a function of its input to an operation count. With a little calibration, that function can typically predict the resource costs of execution.

Traditional algorithm analysis doesn't provide much help for DVE system performance analysis. There are many algorithms in a DVE system software. Each algorithm may have specializations that are not described in documentation. With so many specialized algorithms, analyzing all of them is prohibitively expensive. We also don't know how to analyze the DVE

system's composition of these algorithms. If we follow the function-call stack and assemble a composite algorithm representing the entire system, the result would be unusable. It would have hundreds of terms, take a prohibitively large amount of effort to analyze, and take substantial effort to calibrate for resource estimation.

Scaling a DVE system does not need complete accuracy, just for the major elements of the performance structure. We can make trade-off decisions with a handful algorithms affecting the those major elements.

Even though only a few algorithms will be part of the primary factors, we can't reduce our analytical work very much. We don't know which algorithms to analyze and which ones to ignore. Even if we did, we have the problem of combining them, and integrating them with user behavior. Instead of analyzing the program code, perhaps measurement is a better approach.

3.3.2 Program Instrumentation

Program instrumentation is typically either by function-invocation or by programmer-selected sections of the program. The former is popular because the instrumentation process can be automated. The profiling tools like `gprof` [32] and `oprofile` [30] automatically detect and instrument function entry-points and add instrumentation code. Unfortunately, these tools add overhead and give data biased towards the function-call structure of the program.

Automatic program instrumentation indicates how often each function is called over the

measurement period, how much of the processor's time is spent in that function, and how much is spent in functions it calls. The overhead is at the function-call boundary. That boundary is an implementation artifact, and has little connection to the performance structure of the DVE system. The profiling tools automatically instrument the entire program, which is enough instrumentation overhead to severely distort the readings.

We mentioned in Section 3.1.2 that the DVE system software will adapt to limited resources. For DVE systems, the distortion of function instrumentation is particularly harmful. The overhead can cause a quality-reduction response from the DVE system software. The instrumentation data then collects data on the DVE system while it runs at a quality level reduced to compensate for the instrumentation overhead. The collected data poorly represents the DVE system's normal performance.

Dynamic instrumentation tools like `dtrace` [31] let the developer select which functions to instrument. The developer is unlikely to select all functions for instrumentation. Even if they want to instrument many, they can choose a few at a time to keep the overhead low, and avoid a quality-reduction response from the DVE system. The data from instrumenting a developer-selected set of functions will likely be more representative of a DVE system's normal performance.

Customized instrumentation, by hand-written code or `dtrace` configuration, will tend to give more accurate data than tool-generated instrumentation. These traditional instrumentation techniques, by `dtrace` or hand-written code, tend to focus on statistics of the invocation and execution of the instrumented code: min, mean, max, invocation counts, and possibly variance. While useful for many applications, that data gives little insight into the

DVE's performance structure.

Significant trends in user or software behavior could help identify parts of the performance structure. However, statistics over a measurement period can easily conflate many different significant trends inside the virtual environment into an opaque aggregate. If we only have statistical moments of the underlying data, we will have to run many experiments to connect trends in the virtual environment to observations in data. For example, to determine what user activities in the virtual environment load the DVE system, we'd have to test each one in turn. Most of those tests would only tell us which activities don't load the DVE system.

Like the algorithm analysis problem above, analysis with traditional instrumentation would require a substantial number of tests, with most of them producing little information.

3.3.3 The Networked Virtual Environment Information Principle

Singhal and Zyda [4] introduced the *Networked Virtual Environment Information Principle* to help understand how major design elements in a DVE system structure its resource usage:

> The resource utilization of a Net-VE is directly related to the amount of information that must be sent and received by each host and how quickly that information must be delivered by the network.

The principle is quantified in Equation 3.1.

$$\text{Resources} = M \times H \times B \times T \times P \tag{3.1}$$

M = Number of messages transmitted.

H = Average number of destination hosts for each message.

B = Average amount of network bandwidth required for a message to each destination.

T = Timeliness with which the network must deliver packets to each destination (that is, large values of T imply that packets must be delivered with minimal delay, while small values of T imply that packets may be delivered with longer delays).

P = Number of processor cycles required to receive and process each message.

This model generally defines a relationship between resources utilization, message counts, bandwidth, size of the network, latency constraints, and available processor power.

The model, however, is insufficient for scaling DVE systems because it does not help in interpreting a DVE's use of any given resource. Also, the model does not provide guidance on the elements unique to a DVE that could be changed to improve resource usage, such as a data structure, space in the virtual environment, or tools available to users. It also fails to provide adequate information on the performance structure of the DVE.

3.4 Changing Scalability

DVE system scalability measurement is immediately useful provided it indicates good scalability. When the results are undesirable, a modeling technique could identify necessary changes to the DVE system. The changes then need execution and testing. Provided that the models remain valid and efficiency is preserved, these necessary changes: adjustments to the virtual world, software, and computer resources with reasonable effort can improve scalability.

We need a process for making controlled changes to the DVE system. The process should help us determine the DVE system's scalability and performance structure. It should also help us make changes in the DVE system. It should also keep our understanding of the performance structure current as we make those changes.

The specific needs of DVE system scalability engineering fall outside traditional performance engineering processes, such as ROPES or SPE, for three principal reasons.

First, the load of a DVE system adapts to the DVE's performance. The DVE system itself may adjust quality. The users may adapt to both quality and other QoS attributes by adjusting their behaviors. The adjustments can include shifting between high and low accuracy tools, recruiting more team members to help in complicated tasks or logging off due to frustration.

Second, the processes do not give guidance on the use of the large non-software components of DVE systems: the virtual terrain, the rules, interactions, activities, geometries, and artwork that comprise the virtual environment itself. These elements can provide a large parametric

space for adapting a DVE system's scalability without software changes.

Finally, the traditional performance engineering processes work with well-specified models that start in early iterations of the system design. They are intended to help guide the system design from the start.

The primary design focus of a DVE system is not the software, but the virtual environment. The software is available "off the shelf" as complete DVE system software (Torque, Quake, Unreal, or Unity) or as assembled DVE system software components from open-source and commercial components and middleware. A DVE system's full software is often available in the first month of development. The virtual environment contains all the desired features of the DVE system, as well as many of the primary risk factors. A DVE system development project is better suited to a performance *tuning* approach, adapting an existing system to the desired levels, than an approach that attempts to engineer the right system software from the start.

As an example, Figure 3.2 shows a common approach to performance engineering in modern DVE systems. In this approach, developers match an engine with a virtual environment. They fill in new activities, tools, user abilities, and elements in the virtual environment. The developers then decide on acceptable engineering costs for additional scalability (*assess risk*), and set goals (*establish objectives*). Developers add basic performance instrumentation as they develop the system (*instrument*). When the DVE system is nearly complete, they start high-volume testing. Currently, that testing has a large set of paid users (*user load test with human participants*). After *evaluation*, changes are made to *fix problems* found in the test.

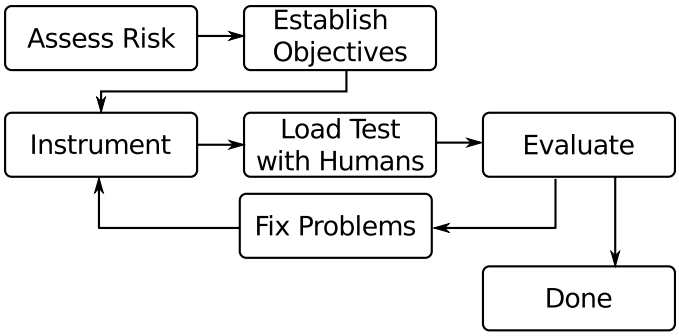


Figure 3.2: The current and common DVE scaling process

The process gives poor results: the high costs of load testing result in few iterations of the cycle, and the instrumentation doesn't provide systematic guidance on necessary changes. Instead, a "whack a mole" strategy is almost necessary, meaning the part of the DVE system software using the most resource is adjusted. Whatever little contextual data the instrumentation gives provides the only guidance for the adjustment. As a result, a larger, systematic approach to re-shaping the virtual environment, user behavior, and the software becomes impossible.

Not surprisingly, the process in Figure 3.2 results in small, low-risk, unintegrated changes to the system at a high cost.

3.5 Summary

The complexities of human behavior, compounded with the complexities of the underlying software system, make DVEs difficult and expensive to scale. Testing the scalability of a DVE typically involves having many people use the system at once. The costs of repeated tests in this manner, especially with concerns of people using strategies only successfully in

the test environment, make it difficult to characterize how users will tax the system in real use.

The DVE system software is complex and it is difficult to separate its parts for individual analysis. The software is too large to fully analyze, and it is difficult to determine the relevant subset of the system without several tests, each with different instrumentation.

Processes like the one in Figure 3.2 are common and almost categorically lead to poor results. Partially, they don't have the advantages of an inexpensive load-test process or modeling methodology. However, they also take a retroactive repair approach that only detects scalability issues when they are big enough to be prohibitive. Such a process does not enable any sort of experimentation or discovery that can either help in developing the DVE further or add to a knowledge base of general principles.

Additionally, DVE system development projects can often start with a pre-developed software stack ready to go and need an approach closer to performance tuning than engineering. A basic physical simulator (physics engine) and network synchronization system are easily acquired in open source projects, commercial toolkits, or CDs attached to textbooks. While some of the engineering work will involve adding unique new capabilities and effects to the DVE system, a substantial effort will come from adjusting the software to perform well for the intended virtual environment and users.

Chapter 4

Approach

We propose a new instrumented user load test experiment, modeling methodology, and engineering process for DVE systems. The user load testing simulates users logging into an instrumented version of the DVE system. The modeling methodology identifies the primary factors that control the DVE system's resource usage. The process incorporates the user load testing and modeling methodology and uses them as a basis for experimenting with changes in the DVE system.

The user load testing experiments simulate the users (i.e., real users' behavior in the virtual environment), and are based on user study observations. The user load testing experiment can run autonomously at arbitrary scale, limited only by available computer resources. The experiments can also be relatively quick to run, with a batch of tests runnable in a few hours. The instrumentation gives data for the modeling methodology, and runs with minimal overhead.

Over multiple user load testing experiments, the modeling methodology measures the parts of the DVE system that use the most resources. The instrumentation tool collects ad-hoc data at high rates (multiple kHz) and lets us run sophisticated analysis after the user load test. Over each cycle, the modeling methodology uses the instrumentation to follow resource usage from the top-most of the DVE system software to the individual software components that use the most resources. The correlation analysis is used to identify factors that control those software components' resource usage. The factors can be in software, or in interactions between user behavior and the DVE system software.

We call the developed process the DVE Scalability Engineering (DSE) process (Figure 4.1). The DSE process also includes invalidation rules for user behavioral data. The users may act differently when the virtual environment or DVE system quality changes, invalidating the collected behavioral data, and the DSE process' invalidation rules require re-testing when needed.

The DSE process is strongly based on Software Performance Engineering (SPE). Smith and Williams [3] describe SPE, and we provide an overview in Section 2.4.1. We remove the original SPE modeling methodology and other parts that don't apply to the DVE systems, and insert our modeling and user load testing. We show the common user load test and the process using SPE terminology in Figure 3.2. We will continue to use SPE terminology to derive the DSE process. This way, we can build on top of the SPE and immediately focus on our contributions.

We will first cover the user load testing experiment. Next, we will cover our modeling methodology. We will finish with a derivation of the DSE process, and apply it to a small

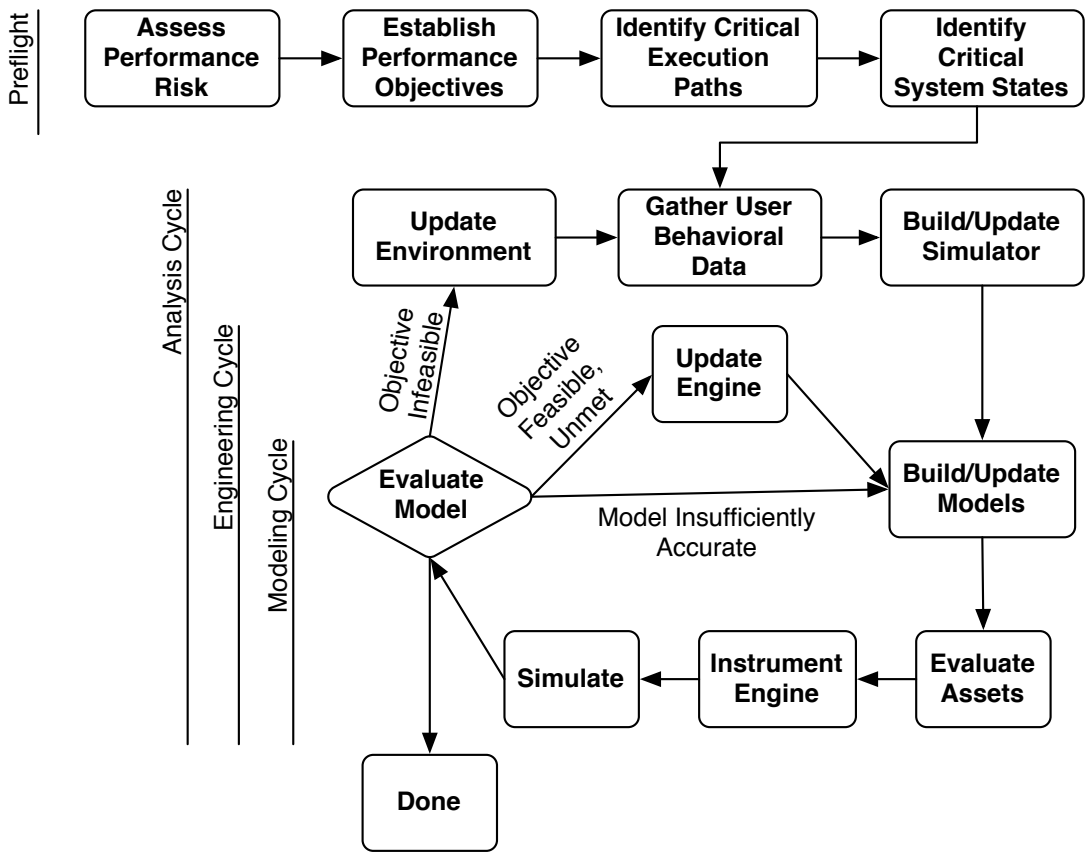


Figure 4.1: Completed DVE Scalability Engineering (DSE) process

(five thousand lines of code) example DVE system (Asteroids).

4.1 The User Load Testing Experiment

The DSE process user load testing experiment has two parts: the user load simulator and the DVE system software instrumentation. The user load simulator acts as our dynamic model of the DVE system users's behavior, the experiment acts as the test of the model, and the instrumentation acts as our view of the results.

For an accurate simulation, we should include a sufficient number of users and their individual and group tactics and strategies. This approach is unusual because user load simulators for

DVE systems often have randomized user activity or pre-recorded sessions played back (at user counts different from the recorded session). Our behavioral-simulation approach lets us directly experiment with behaviors when running user load testing experiments.

As we outlined in Section 3.3.2, good instrumentation will add very little overhead and give us raw data. We don't want the DVE system to distort quality due to measurement overhead. Each user will vary over time the load they apply to the DVE system. The variations in the user load correspond to the different user activities in the virtual environment. We want to identify those user activities separately from the raw data in off-line analysis.

Figure 4.2 shows a process that replaces the human experiment (user study) from Figure 3.2 with the user load testing experiment. The "Fix Problems" stage may change how users behave in the system, so we provide two options: one that re-acquires user behavioral data, and one that doesn't.

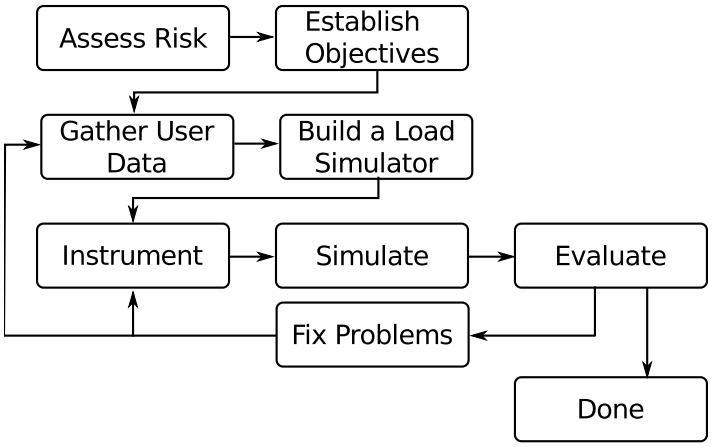


Figure 4.2: A DVE scaling process with user load simulation experiments

We will cover the user load simulator and then the instrumentation tool.

4.1.1 Load Simulation

We partially replicate human behavior for a realistic DVE system user load. User behavior is purposeful, imperfect, tactical, strategic, and social. Our approach attempts match it. We will discuss the user studies we use to gather behavioral data and behavioral reproduction techniques. We finish with a small discussion on the simulator's accuracy and some secondary benefits.

Simulator construction starts with a user study. This is labeled "Gather User Behavioral Data" in Figure 4.1. The user study has users acting as normal as possible within the DVE system. Their actions are recorded by the DVE system's "demo recording" mode, normally used for demonstration and debugging purposes. This mode records all data in and out of a user's session in the DVE system, and can play them back later.

Next, a DVE system designer/developer plays each demo back. The developer categorizes the observed user activity into behaviors and records the time breakdown of each behavior in the recorded session (for all the recorded sessions). The resulting data forms a behavioral distribution usable for constructing the user load simulator.

We expect that user load testing is already part of a DVE system's development process. The user interface, virtual environment, interactions, and active virtual objects need testing inside the DVE system for validation. Once the data is collected, we move to the "Build/Update Simulator" stage in Figure 4.1. We take the data and turn it into a user load simulation component.

Our simulation approach is to reapply the existing work. Substantial work in realistic Non-

Player Characters (NPCs, also referred to as AI characters) already exists. That work includes techniques for chasing other users, path-finding, swarm movement, and goal selection, and are discussed in Section 2.6.

We implement each observed behavior in a modified version of the DVE system software that serves as a user load simulator. The user load simulator emulates user input and reads the DVE system's local view of the virtual environment to execute the user behaviors. It selects these behaviors according to the tabulated distribution. To reduce the computing resource costs of the user load simulation, the user load simulator can also have its graphical rendering and user input and sound systems disabled.

We run one instance of the simulator for each user that we want to simulate. With availability of commodity virtual machines (cloud computing), we can run simultaneous user load simulations with simulated users at relatively little cost.

The user load simulator needs only be accurate enough to match the instrumentation data we would get from the equivalent (in terms of the number of users) user study. For small numbers of users, the user load simulator is less likely to significantly represent the actions of users. For simulations with more simulated users, we expect that the underlying population sample used for the behavior model to become more representative of the expected behavior. Alternatively, the user load simulators can be artificially tuned to cause some excessive user load in a different mode, to test the DVE system's ability to handle exceptional user situations.

4.1.2 Instrumentation

We have developed a custom low-overhead performance instrumentation tool called `ppt`. Traditional instrumentation tools are not intended to help the explorative correlation analysis used by DSE process' modeling cycle. In that cycle, almost arbitrary parameters of the DVE system are measured or recorded together for the sake of correlation or causal discovery.

Function profilers such as `gprof` [32] or `oprofile` [30] can add substantial run-time overhead (altering quality, discussed in Section 3.1.3), provide only function-level statistics, and little data on the underlying distribution. `dtrace` [31] reduces the overhead, and enables applications to export their own custom data. `dtrace` only provides simple statistics of data collected.

The user load in DVE systems is complex. That complexity can manifest in the distribution of user positions, movements, and activities. These variations in user load can cause large variances in the resources required for the DVE system to process.

For example, some sports simulations, like soccer, can have many users follow a ball simultaneously. These "herd" behaviors can cause substantial and complex load on a DVE system. Avatars may be moving between collision-detection zones, constantly invalidating cached meta-data (e.g., nearest-visible opponent), and causing events (e.g., footstep sound effects). An untuned DVE system could expend a substantial amount of effort executing these bookkeeping activities individually. It would be preferable to identify these correlations in activity and behavior, and batch-optimize the DVE system's bookkeeping for herd behavior.

When measuring such a DVE system, one would need to identify when herd behaviors occur and identify the corresponding DVE system software components that consume more resources in response. Aggregate statistics may not be able to identify these bursts of bookkeeping effort, and their magnitude may be lost when averaged with non-burst activity. Higher-level moment analysis can sometimes help, but complexity in the analysis can make factor identification difficult.

Additionally, there may be other major factors in the DVE system performance. There may be other user behaviors that cause bursts of high load. Or a behavior-independent DVE system activity may occasionally have spikes, such as a garbage collector. A DVE system developer needs to connect load data and DVE system state to properly identify major DVE system performance factors.

We created `ppt` for measuring systems like DVE systems. It provides very low overhead instrumentation of application-specific data, with access to all data collected for offline analysis. As DVE systems are rarely memory-bound, `ppt` uses large buffers to keep most overhead to streaming writes to shared memory, and large-block writes to disk.

Data is gathered into "frames," each containing individual data points of related variables in the DVE system and time-stamps of individual steps in processing. For example, consider a soccer simulation. If we wanted to determine the effects of herd behavior, the frame for simulation of an individual object in a sports game would include:

- The type of the object,
- A unique identifier for the object,

- The position of the object,
- The time simulation started,
- The distance the object was moved in this time-step,
- The time the simulation finished,
- The time collision detection started,
- The number of collision tests executed,
- The time collision detection finished.

After the data has been collected, an analysis can show correlations between simulation time and distance moved for avatars. As that actual movement operation is distance independent (being a single matrix multiply), it makes for an anomaly that needs an investigation. An analysis of avatar movements against the ball, will show correlation between the ball's distance-traveled and the avatars' distance-travelled. Once the correlation is shown, the number of uniquely identified avatars with slow simulation at the same time will provide solid clues to the underlying cause. Additional data collection work, with more data in the frame, can help identify which DVE system components are consuming the time.

`ppt` converts these frame declarations into C `struct` declarations and defines macros to populate and publish these frames to shared memory. As the macros just assign members to a `struct` and write it to an array (with memory-barrier operations for consistency), the overall CPU cost per frame-write is dominated by access to the relevant cache line. In

experiments at 3.5KHz (data sampling frequency), the measured overhead was less than a millisecond of CPU time per second.

4.2 Modeling Methodology

Our modeling methodology uses a "divide and conquer" strategy to identify the DVE system software component that uses the most resources. The strategy minimizes analysis of components that don't use many resources. Once identified, the methodology uses correlation analysis to find factors that control the component's usage.

Singhal and Zyda's NVE Information Principle (NVE-IP) [4] represents the relationships between resources and system requirements, but its format isn't directly usable for a particular resource. With differing costs and availability of each resource, and different consequences on other factors (e.g., power consumption), a more specific breakdown may be desirable. Below, we take the Principle and apply it onto three resource axes: CPU time, memory, and network bandwidth. The list isn't exhaustive and additional resources may be relevant for particular deployments. For example, GPU resources used for simulation may a need similar consideration.

Our modeling methodology focuses on one resource at a time. The focus resource can often be identified with a system-level tool, such as `top(1)` during an initial load test. Once that resource is identified, we can use one of our NVE-IP based resource estimates as an initial model. The methodology instruments the DVE system's primary software loops to identify the primary source of resource pressure. From the primary software loop, it identifies the

part of the software loop that produces the most pressure. It then instruments code that runs in that part of the loop and runs another user load test. Further iterations focus into smaller parts of the code that produce resource pressure. After the methodology identifies the component taking the most resources, it uses collected data to find any controlling factors of resource usage. Correlation analysis of collected `ppt` data can help find them.

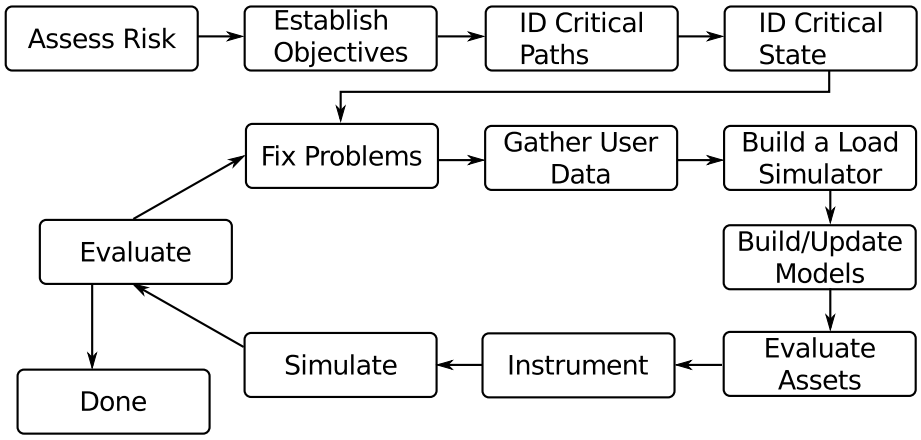


Figure 4.3: A DSE process with user load simulation and modeling

Figure 4.3 shows the process from Figure 4.2 integrated with this methodology. We describe the new stages below.

Identify Critical Paths Identify the primary loops or other top-level control structures in the DVE system. This can also include thread pools or event loops. These critical paths are the starting points for the methodology.

Identify Critical State Identify the primary elements of the program state. For DVE systems, this will certainly include the scene graph and any synchronization state kept for any server, client, or peer.

Build/Update Models Take instrumentation data from the last user load test and integrate it into the model. The model starts with one of our resource-projections of the NVE-IP, and then changes to fit to the DVE system through iterations of this step.

Evaluate Assets Look at any assets in the system: graphical models, terrain descriptions, collision geometries, procedural generation parameters, etc., and determine any parameters needed for the model. For example, the number of points in a collision geometry can indicate the number of loop iterations a collision detection component runs each time it encounters an object with that geometry. Thus, the number of points can be a useful model parameter for predicting the collision detection component's CPU usage.

4.2.1 CPU Requirements

Our first use of the NVE-IP is for CPU usage. A DVE system's CPU requirements are more complex than Singhal and Zyda's message-processing model which includes work for synchronization (P and M of the NVE-IP) and simulation (S terms below).

We will not consider the presentation elements of the DVE system: graphical rendering, input, and sound. First, they have been well studied Luebke et al. [44], and second, they don't necessarily affect scalability with the same magnitude as simulation and synchronization.

$$\text{CPU Load} = R_M \times P_{RM} + X_M \times P_{XM} + R_S \times P_S + \epsilon \tag{4.1}$$

R_M = Incoming message rate (Hz).

P_{RM} = Average number of processor cycles required to receive and process each message.

X_M = Transmitted message rate (Hz).

P_{XM} = Average number of processor cycles required to construct and transmit a message.

R_S = Rate of simulation — e.g. how often the simulator is invoked (Hz).

P_S = Average number of processor cycles required to run the simulator.

ϵ = Everything else.

In all three applications of the NVE-IP, we include an ϵ term to include resources devoted to overhead or novel parts of the system.

While CPU performance has increased substantially, the synchronization work isn't much larger than it was ten years ago. For instance, the update frequency used in modern DVE systems is usually the same 10Hz (e.g., Massive II), and limits in the DVE system scalability have kept the size of the scene graph relatively stable.

In simulation, the work breakdown of P_S depends on the number of objects to simulate, their complexity in terms of applicable physical rules, and the desired accuracy of the results. Synchronization work terms P_{XM} and P_{RM} depend on the number of objects and events to synchronize; the (de-)serialization work required; any prioritization, filtering, or lag-compensation mechanisms involved; and compression or encryption on the data stream.

Measurements of the top-level summation can determine which part, simulation ($R_S \times P_S$) or synchronization (everything else) dominates. As P_S is often dominated by the sum simulation time of each object in the scene graph, we can look at factors controlling their simulation times to understand the total. For example, some users tax the DVE system more with their behaviors than others, or close groups of users force the use of slower but more accurate physics.

4.2.2 Memory

The memory requirement for a DVE system is the size of the scene graph plus the synchronization states of all connected hosts.

$$\text{Memory} = H \times S_{sync} + \left[\sum_{t \in \text{types}} S_t \times C_t \right] + \epsilon \tag{4.2}$$

$H =$ Number of connected hosts.

$S_{sync} =$ Size of synchronization state per host (bytes).

types $=$ Set of object-types in the environment.

$S_t =$ Average size of an instance of type t.

$C_t =$ Count of instances of type t.

$\epsilon =$ Everything else.

The number of objects in the scene graph can dominate this equation. The synchronization state of each host is determined by which objects it must synchronize, and how close its view of the object is to the host. Large numbers of objects will affect S_{sync} and C_t. Factors

controlling the number of objects can include choices in e.g., whether to simulate collapsing walls with virtual bricks or use a pre-recorded animation, the firing rates of weapons, the amount of simulated debris from vehicle crashes, or whether a virtual paintball can ricochet.

In our research, we have yet to find memory being a major bottleneck in DVE system scalability. The per-user memory footprint in a server was typically a few dozen megabytes for the single-user case, with substantially less than a megabyte added for each additional user.

Instead of fitting the main memory capacity, a more interesting concern may be fitting the primary elements of simulation work within L3 cache, or a GPGPU's VRAM. Partitioning the required data into higher levels of the memory hierarchy for each task may lead to substantial improvements in DVE system performance. For example, a NUMA-configured two socket motherboard could dedicate one node (CPU and its half of memory) to simulation and the other to synchronization.

4.2.3 Bandwidth

The bandwidth required for a DVE system can be expressed as:

$$\text{Bandwidth} = (X_M + R_M) \times B \times H + \epsilon \tag{4.3}$$

R_M = Incoming message rate (Hz).

X_M = Transmitted message rate (Hz).

B = Average amount of network bandwidth required for a message to each destination.

H = Average number of destination hosts for each message.

ϵ = Everything else.

An analysis of B can be revealing: it's the size of the serialized portion of the scene graph synchronized between this machine and a host plus protocol overhead (e.g., packet headers, acknowledgments, etc.). The selections of which parts of the scene graph to synchronize, and their serialized forms, are both substantial factors.

Some DVE systems may try to fill a specific goal size for B, choosing as many objects as will fit. B is itself a design decision to balance bandwidth against the consistency, latency, and completeness needs of synchronization. In this approach, other factor is the prioritization criteria for selecting objects for serialization.

Other DVE systems may send over the entire region's set of virtual objects in delta-form, with an occasional full copy of each object.

In any synchronization mechanism, the serialized format of each virtual object is a major factor in the total bandwidth used by the DVE system. That format often includes a partitioning of the object's properties into blocks. DVE systems can then send only the changed blocks regularly, substantially reducing the amount of data sent per object. Effective partitioning depends on accurately understanding the change rates of each object's properties.

For example, a virtual tank's z-coordinate may be irrelevant most of the time, as it's too heavy to go in the air. Similarly, a projectile may be better represented by its launch time and vector, than with a series of individual new positions. Decisions on these matters can substantially affect a DVE system's bandwidth requirements.

We have considered every variable from Singhal and Zyda's NVE Information Principle, save one. The T parameter (a constraint, defined as inverse tolerable latency) doesn't directly cause resources to be expended in many DVE systems — they simply result in changes in quality. Instead, T more directly affects consistency.

With spare bandwidth available, data can be sent more redundantly to reduce the costs of packet loss, helping to maintain consistency.

Now that we have applied the NVE-IP into resource spaces, a modeling methodology to adapt them to specific DVE systems, and a load simulation experiment to gather data for the methodology, we can look at effectively using them together to make DVE systems scale better.

4.3 DVE Scalability Engineering

In the DVE Scalability Engineering (DSE) process the user load testing experiment provides data for the modeling methodology. The user load testing is used as many times as necessary to find critical factors. Once found, it experiments with changes to manipulate the factors to make the DVE system scale better. The DSE process includes three primary cycles. We will start with the Modeling Cycle, which works to discover factors. Depending on the nature of

the factor — be it a design or implementation choice in the DVE system software, or a design decision of the virtual environment, one of the other two cycles can help. The Engineering Cycle experiments with changes for factors controlling the DVE system software's resource usage. The Analysis Cycle experiments with changes to the virtual environment.

Figure 4.3 shows several stages with multiple exit arcs, with no guidance on which one to take at any time. Also, the "Fix Problems" stage doesn't use the model constructed, doesn't have any guidance on how to fix anything, or indicate what needs to be fixed.

Meta-Structure The DSE process can only help Quality of Service (QoS) parameters that depend on resource usage. Resource independent QoS parameters include the user interface and input-device mappings, as per the taxonomy by [1].

If the underperforming QoS metrics are not constrained by resources, then other human interaction concerns dominate and should be addressed through usability engineering processes.

With a QoS metric that depends on resource pressure, the DSE process can be effective.

For each QoS metric and resource pair, the developer starts iterating Modeling Cycle to determine the primary factors. Depending on the nature of the factor, iterations of either the Engineering or Analysis cycles manipulate the factor to improve QoS.

Asteroids We'll use an example DVE system to help illustrate the purpose and effects of the preflight and modeling parts of the process. *Asteroids* [45] is a simple one-player arcade game that we've already modified to be a two-player game. A sample screen is in Figure 4.4.

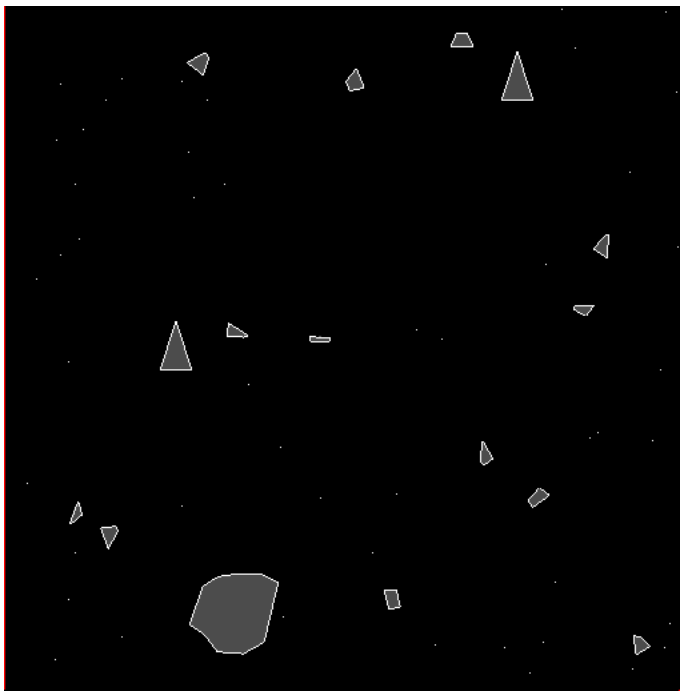


Figure 4.4: Asteroids

Asteroids was originally a single-player game and a single-threaded DVE system. The modifications create a network link to the peer and sends player state and new-bullet information over the link at 10 Hz. A second thread was added to process data received from the peer.

4.3.1 Preflight

The "Preflight" phase runs once at the beginning of the DSE process. It determines the scope and objectives of the work, and initializes data for our modeling methodology. The preflight phase is based off the initial non-loop block of SPE.

Assess Performance Risk First, determine the value of the QoS (and resulting scalability enhancements) in terms of its available engineering effort. During the DSE process, the QoS and scalability return on the investment of DSE effort must be understood to keep realistic

goals and provide acceptability criteria before attempting major changes in the system.

Establish Performance Objectives The Causal Chain [1] can help determine the QoS metrics that, if improved, can improve scale. Users may unpredictably react to changes in QoS and change behavior. Substantial changes in one QoS value can thus alter the behavior balance supporting the QoE in the DVE system. The final values for desired QoS are only determinable by additional user testing.

Identify Critical Execution Paths and System States The Modeling Cycle depends on finding sources of resource pressure, and those start from the top-level functions on all threads, and the primary DVE system state. As we mentioned earlier, it uses a "divide and conquer" strategy for finding the software components that consume the most resources. The strategy needs the top-level functions and system state for a starting point.

Asteroids Preflight The DSE objective for Asteroids is to illustrate the DSE process. We use latency as the system metric to drive both better consistency (by maintaining a 10 Hz update cycle) and speed. Latency follows the execution path of the primary thread from reception of input from the user and their view of the resulting DVE system state.

The primary execution path in the Asteroids source is a loop within `main()`. The primary system state is the set of virtual objects in the virtual environment and the queued bullet and avatar states to transmit to the peer. The virtual environment has virtual objects for a player (an avatar in a form of a space ship), an asteroid, or a bullet.

4.3.2 The Modeling Cycle

The Modeling Cycle adds minimal instrumentation to all the state or functions, depending on the bottleneck resource, identified in Preflight. After a user load testing experiment, it identifies the subset of the DVE system that used the most resources, at the granularity of the instrumentation added. Further iterations add more instrumentation to the identified subset, and "drill down" to an individual DVE system software component.

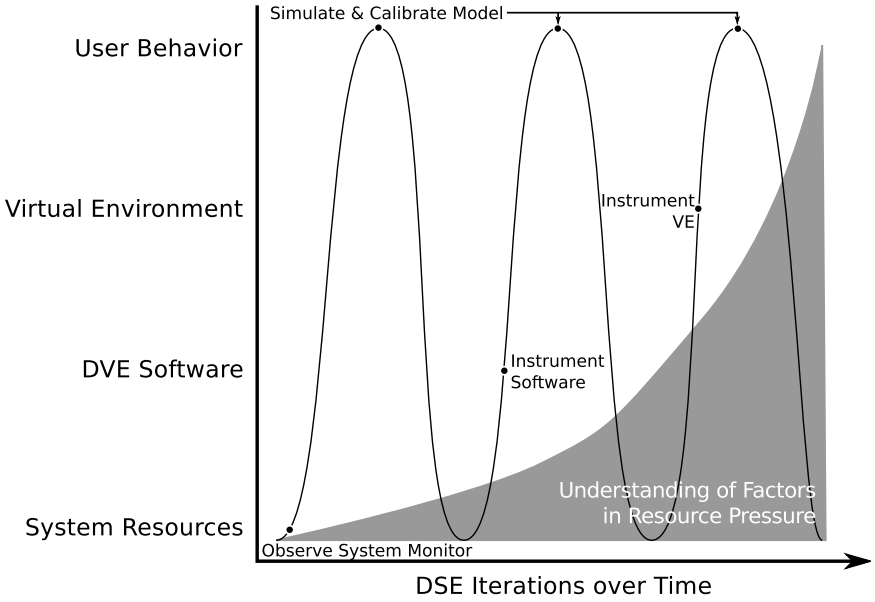


Figure 4.5: Iterations of the Modeling Cycle, with growing understanding of the DVE system performance

Figure 4.5 shows the iterations as oscillations between phases of the DSE process. The vertical axis shows major parts of the system, and the oscillations follow the Modeling Cycle of the DSE process (Figure 4.1). Each cycle connects data from the load simulation down to measurements in the DVE system resources. Instrumentation on states of the DVE system software or virtual objects in the virtual environment connect the parts in between. Over several iterations of the Modeling Cycle, the system becomes better understood.

Asteroids: First Iteration We determine the initial model for Asteroids by looking at the software loop in `main()` method. The loop has three parts: collision detection, drawing, and state transmission to the peer. The loop runs as fast as possible.

We will determine the largest factor in local response latency. The latency between the time the user manipulates an input device and when the user sees an on-screen response.

The primary thread controls that latency. The second network-listener thread won't be counted because it doesn't affect latency. It primarily contributes to consistency. We assume that the two threads won't significantly contend for the same CPU core.

We start with the *CPU Load* equation we provided earlier and derive a *Local Latency* equation from it. This Local Latency will be the basis of our iterations. First, we add *draw* time to this model, as it is part of Local Latency in Asteroids. Second, we separate the elements that contribute to local latency from those that don't. We always transmit, simulate, and draw the main loop. The rates (X_M, R_S) are all equal to the main loop frequency, and do not apply within a single iteration.

$$\text{CPU Load} = \overbrace{X_M \times P_{XM} + R_S \times P_S + \text{draw} + \epsilon}^{\text{Local Latency}} + \overbrace{R_M \times P_{RM}}^{\text{Other thread}} \tag{4.4}$$

$$\overbrace{\text{Local Latency}}^{t_{\texttt{main}}} = \overbrace{P_{XM}}^{t_{\texttt{send}}} + \overbrace{P_S}^{t_{\texttt{collide}}} + \overbrace{\text{draw}}^{t_{\texttt{draw}}} \tag{4.5}$$

This completes our "Build/Update Models" stage of the DSE process. We don't yet have any assets to consider: later on, we may want to characterize the vertex-count of the asteroid

fragments in case we find that asteroid fragment collision is a concern. The `ppt` frame for this iteration will store timestamps for the entire loop iteration, and the demarcations between these phases. Then, we will insert `ppt` macro calls into the loop's source code and run a simulation. This is our Instrumentation stage of the DSE process.

We run the game live in lieu of a simulation. We had two windows open and play the game. The windows are placed adjacent to one another, and a quick flick of the mouse moves the keyboard focus between them. The machine was a Linux laptop, with an Intel i7-2820QM @ 2.3 GHz CPU and 20GB of RAM. One of the peers has `ppt` attached to it for the duration of play. This gives us the following data on these components. All units are in milliseconds.

	Min	**1st Quartile**	**Median**	**Mean**	**3st Quartile**	**Max**
$t_{\texttt{main}}$	0.1938	16.6300	16.7100	16.7100	16.7900	32.9400
$t_{\texttt{collide}}$	0.0010	0.0054	0.0092	0.0383	0.0151	1.4910
$t_{\texttt{draw}}$	0.1487	16.5300	16.6300	16.6000	16.7100	32.8800
$t_{\texttt{send}}$	0.0197	0.0371	0.0540	0.0679	0.0745	0.3233

Table 4.1: First iteration of Asteroids data

$t_{\texttt{main}}$ mostly runs around 16.6ms, dominated by the draw time, which typically runs around 16.5ms. The other two elements of work are in the tens of microseconds.

Asteroids: Second Iteration We would like to know more about why drawing takes so long for such a simple scene. We found out that the model is insufficiently accurate so we returned to the "Build/Update Models" stage of the DSE process. We did so considering that the collision detection and network synchronization seems so much more complex than drawing a dozen polygons' outlines. Looking back at the main loop, there are lines of source in the drawing phase: (1) `world->draw()`, (2) `gui->draw()`, and (3) `Display->update()`.

We insert two timers between these elements, and update our model:

$$t_{\texttt{main}} = t_{\texttt{collide}} + \left[t_{\texttt{draw\_world}} + t_{\texttt{draw\_gui}} + t_{\texttt{draw\_update}}\right] + t_{\texttt{send}} \quad (4.6)$$

	Min	**1st Quartile**	**Median**	**Mean**	**3st Quartile**	**Max**
$t_{\texttt{draw\_world}}$	0.0218	0.0389	0.0543	0.0745	0.1216	0.4357
$t_{\texttt{draw\_gui}}$	0.0003	0.0005	0.0008	0.0089	0.0010	0.2621
$t_{\texttt{draw\_update}}$	0.0993	16.4400	16.5400	16.5200	16.6400	32.8000

Table 4.2: Second iteration of Asteroids data

Where $t_{\texttt{draw}} = t_{\texttt{draw\_world}} + t_{\texttt{draw\_gui}} + t_{\texttt{draw\_update}}$. Table 4.2 shows the results of a second simulation. Clearly the last, the call to `Display->update()`, consumes the CPU the most. Looking at that procedure, it has exactly two lines: one to swap the graphics buffers, and another to clear the current one not being used for drawing. With the `update()` procedure being two lines long, and both routines provided by the graphics library, and the fact that these lines are idiomatic for many computer graphics systems, we consider the graphics library the culprit. There is little reason to stick with this library for such little work.

4.3.3 The Engineering Cycle

The Engineering Cycle is a simple experiment with changes to software. It hypothesizes that traditional software performance engineering and optimization techniques can help improve QoS. Data structure or algorithm selection, tuning, or implementation optimizations can be

quite effective, especially with the specific guidance obtainable from the Modeling Cycle.

The scope of an iteration of the Engineering Cycle's is a single software-change experiment. Any further instrumentation or model development is considered Modeling Cycle work.

In this cycle, a change is made to the software and the simulation re-run. The existing instrumentation indicates the change in QoS-relevant values. If the desired result is not directly obtained, the modeling cycle can be used to correct and clarify issues in the model.

In the running Asteroids example, an Engineering Cycle is an excellent next step in improving latency. Replacing the implementation of `Display->update()` with a faster implementation, such as moving to drawing directly in X buffers and swapping them on the X server-side (instead of the current `libSDL` shared-memory implementation) may help reduce the substantial latency imposed by drawing. We will not try an experiment on Asteroids, but save that effort for a more substantial engine.

4.3.4 The Analysis Cycle

The Analysis Cycle is an experimental change in the virtual environment. This cycle gets its name from the requirements analysis needed to decide on possible environmental changes, in exchange for the desired QoS.

The cycle begins with a change to the virtual environment. We consider the change likely to cause changes in user behavior. So, a user study and an update to the simulator is required. The rest of the cycle is an evaluation of the experimental results.

As with the Engineering Cycle, the Analysis Cycle is a fixed-scope environment-change ex-

periment. Changes to the virtual environment, rules, interactions, activities, or major media assets to the system are within scope. Any further instrumentation or model development is considered Modeling Cycle work.

4.4 Summary

We started by developing a repeatable and instrumented DVE user load test that doesn't require conducting user studies each time that the test is run. We used the user load testing as a building block for a modeling methodology that finds major factors in the DVE system resource usage. We combine the user load test and modeling methodology into a DVE Scalability Engineering (DSE) process. The DSE process allow us to manipulate resource-use factors and study their effect in order to enhance the DVE system performance. We described the DSE process using Asteroids, a small DVE system, as an illustrative example and a small case study.

In Chapter 5 we provide two case studies using commercial DVE systems, Torque, and Quake III. We will cover the entire DSE process in the Torque case study (Section 5.1) and then illustrate making design-level decisions in Quake III case study (Section 5.2).

Chapter 5

Case Studies

There are several commercial open source DVE systems that are real-time (versus turn-based) such as include Torque [43], Quake III, and Quake IV. Torque was used as the basis of the Tribes II game on desktop computers, and is now used as the basis of several downloadable XBox 360 games. Quakes III and IV were both released directly as games. We used Torque and a maintained version of Quake III called `ioquake` [46].

We avoid Quake IV for two primary reasons. One, since it was only recently released (Nov 2011), there is little community support for its use or modification. Two, the primary changes between Quake III and IV are in rendering and the scripting language. For scalability, there is little to be gained from the analysis of the renderer. Level of Detail (LoD) [47] optimizations have made renderers able to handle many users, and rendering systems do not directly interact between hosts. The scripting language is too low-level an implementation detail to affect scalability.

Instead, the simulation and networking components are of primary concern. Those components are similar between Quake III and Quake IV.

Our DVE process was used for the case studies to establish a simulated user load testing experiment, iteratively model the DVE system to identify primary performance factors, and experiment with changes to the DVE system. The DSE process diagram is shown again in Figure 5.1.

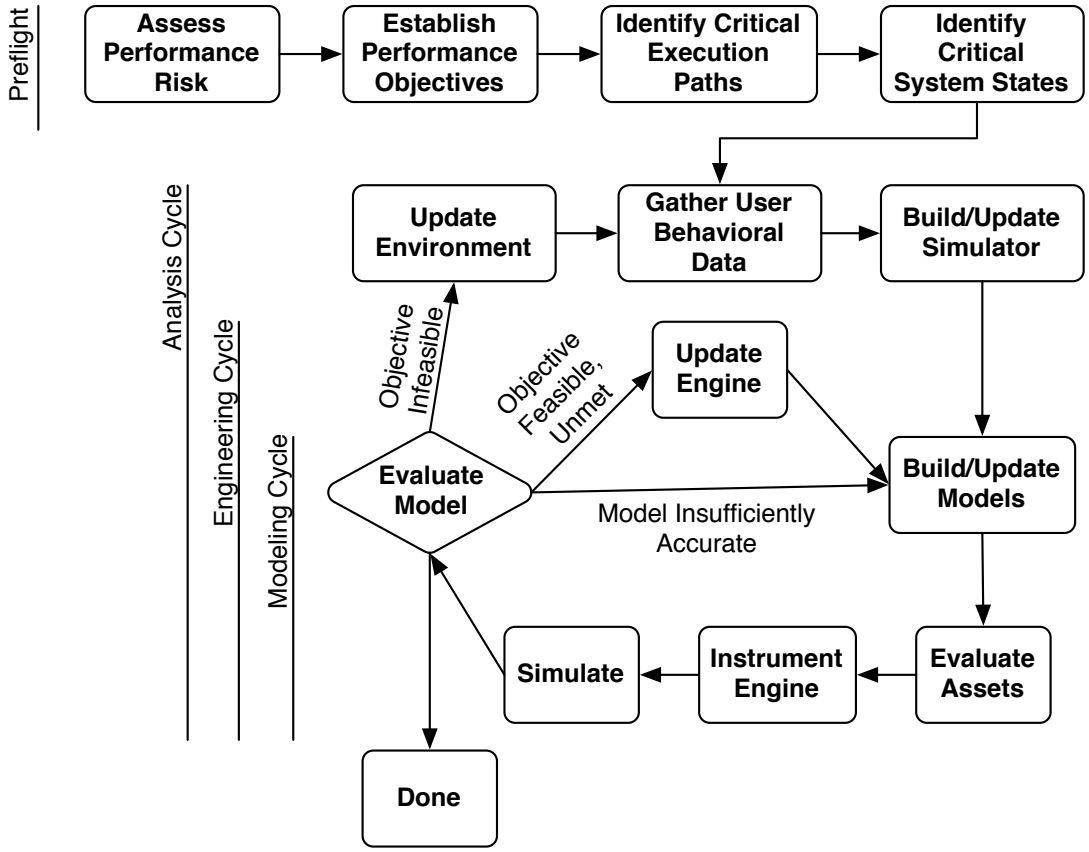


Figure 5.1: DVE Scalability Engineering (DSE) process

5.1 Torque Case Study: DVE System Capacity

Our first case study is Torque [43]. Our hypothetical scenario is a deployment of many instances ("shards") of a Torque-based DVE system. We use the example "starter" virtual environment definition that comes with Torque's development kit. We initially aim for supporting instances of sixty users each. The "starter" virtual environment is a tiny virtual village, which is crowded with sixty people. We can imagine it being one of many parts of a larger virtual environment, connected by a realm selection mechanism similar to what we discussed in Section 1.1.

We want to fit as many instances we can onto our hypothetical deployment hardware. After we fit all of our desired users, we can try to alter the DVE system to fit more users in each shard. We know from Bhatti and Henderson [25] that we must keep our total round-trip-time (RTT) between a client and server below 60ms, and that includes the time spent between the server's reception of client data, processing, and transmission of data back.

Torque uses a client-server architecture, and we focus on the server. The clients have less work, except for rendering.

5.1.1 Early DSE and the Load Experiment

We first go through the Preflight "checklist."

Preflight

Assess Risk: The DVE system was built by a team of experienced developers, and our biggest risk is finding no way to improve the system.

Establish Performance Objectives: We want to keep the same performance in the DVE system. So, we aim on reducing the resource requirements of a sixty-user instance of the DVE system. For the same set of computer resources, that will give us a higher instance capacity.

Identify Critical Execution Paths: Torque is a single-threaded game engine with a single primary loop. Each iteration handles input processing, simulation, and output processing, as needed.

Identify Critical Execution State: A Torque server's state is the virtual environment as represented in the DVE system, and the synchronization states of network links to each client. The starting virtual environment is the one shipped with the Torque software (and literally named "starter"). The synchronization states are instances of `class NetConnection`.

Analysis Cycle: Setting up the Load Experiment

We run user studies and build a program capable of roughly simulating the behavior we observe.

Gather User Behavioral Data: For the Analysis Cycle, we run a user study to get data for a user load simulator. We actually ran two of these cycles. In each, we put study participants in a computer lab and asked them to play the game for an hour. Torque's

built-in mechanisms recorded every message sent or received.

We played back each recording individually, and noted the user's behavior in sub-minute intervals. We recorded each tactic used with its frequency. Table 5.1 shows the tactics observed, and their frequencies. The "starter" world is very simple: one weapon, a small village, and only "kills" counted for points. There were small health packs scattered across the area. We modified the "starter" to ensure that every player had unlimited ammunition.

Note that the larger ("10G") user study group used more defense-aware strategies. "Stand and Attack" was reduced by almost half when there were more opponents. Randomly moving about while attacking indeterminately, the "Scatter" strategy, is less popular (by ten percent).

Strategy	**Total %**	**10G %**	**5G %**
Circle-Strafe	16%	18%	11%
Hide-Snipe	1%	1%	2%
Scatter	16%	15%	18%
Snipe 50m	4%	4%	3%
Snipe 100m	7%	6%	10%
Snipe 150m	3%	3%	2%
Snipe 200m	2%	2%	1%
Snipe 300m	1%	0%	2%
Chase-Kill	16%	17%	13%
Wander	10%	11%	8%
Reverse-Attack	4%	4%	0%
In-Building Snipe	5%	5%	4%
Shoot into building	2%	2%	1%
Inactive	8%	7%	11%
Parallel Strafe	2%	2%	0%
Sinusoid	2%	0%	8%
Stand & Attack	5%	4%	7%

Table 5.1: Relative strategies from the user study, for the entire group, the 10-Person group (10G), and the 5-person group (5G)

This characterization gives both the expected user behaviors and the kinds of activities that we need to support. If we were looking for specific QoS levels to maintain, we could look into task performance metrics for each of these activities, and the QoS levels they need.

For example, the first activity is Circle-Strafe, an arc-shaped movement created by moving laterally and forward at the same time, while maintaining aim on a target. This activity's requirements for performance are primarily in consistency for nearby players. The target should move smoothly, and any sudden jumps that can be caused by dead-reckoning errors [48] will become visible. During this strategy, distance is more important than the angle between the chaser and its target.

At different ranges from their targets, the Snipe family of activities requires high-quality synchronization of distant players in a small cone projecting from the user's viewpoint. Players at similar distances — but away from this cone — will be substantially less relevant. The simulation user load for Circle-Strafing player, with rapid movement and a high rate of fire, is substantial. A stationary sniper with a low rate is substantially less taxing.

Build/Update Simulator: The next step of the Analysis Cycle is to build a load simulator. We take a copy of the client program and modify it to synthesize the keyboard events use by the players for movement and firing according to the behaviors listed in Table 5.1. The client program uses an internal map of the level, and the current location of its own avatar and other players nearby as its knowledge base for action.

We implement each behavior using a state machine. The client selects each to run with the same distribution as observed: the *"Total %"* of Table 5.1. The client matches the distribution by randomly selecting a behavior out of the set of those under-represented in prior rounds.

It doesn't do any path-finding. The terrain is flat and regular enough for straight-line paths. In the center of the virtual environment is a small village of four buildings. Their

locations and orientations are represented in the load simulator's internal map. We hard-code knowledge of the scene: places with good cover, sniping points, etc. For behaviors that attack other players, they choose the closest other player. Unless they were sniping, then they chose a player at an appropriate distance from their snipe point.

Sixty instances of these simulators should give us roughly the same user load as sixty users. We had some instrumentation running during the user study, and our load simulator matched it. However, we did all of our user studies in the beginning when we didn't have much instrumentation in place.

5.1.2 The Modeling Cycle

Our initial model is empty and we did not know which resource is a bottleneck. It could be either CPU, memory, or network bandwidth. In order to identify the bottleneck resource, we ran a user load simulation. An Intel Core2-Duo based 17" Apple Macbook Pro acts as the server, and a Sun Ultra 40-M2 hosts the load simulators. The MacBook Pro has some monitoring abilities built-in for all three resources.

The CPU utilization (of a single 2.2 GHz core) hits capacity, reaching 100% while the other two resources: a gigabit Ethernet link and 4 GB of memory, are unperturbed at less than 100KB/sec and 1 GB, respectively. Now that we know that the CPU is the capacity bottleneck, we will focus on that.

We time each run of Torque's primary software loop's four parts: the core loop, input processing, physical simulation of the virtual environment, and output. Mapping these

elements to Equation 4.1, we have:

$$\text{CPU Load} = \overbrace{R_M \times P_{RM}}^{\texttt{input}} + \underbrace{\overbrace{X_M \times P_{XM}}^{\texttt{serialize}} + \overbrace{R_S \times P_S}^{\texttt{simulate}}}_{\texttt{coreloop}} + \epsilon \tag{5.1}$$

Their `ppt` frames are `coreloop`, `input`, `simulate`, and `output`, respectively. The `coreloop` represents a cycle of the top-level loop in Torque. Each frame has a `start` and `end` time-stamp, whose difference indicates the time spent in that part of the work cycle. The `coreloop` `start` and `end` times will fully contain the `input`, `simulate`, and `serialize` times. Individual event times can be summed as contributions to the `coreloop`-spent time. Additionally, the `coreloop` stores the number of logged-in users (`N`) and the `ticks` scalar mentioned above.

Another simulation with this instrumentation reveals that the time spent in `simulate` dominates the other two. Table 5.2 shows the results.

Component		**Measured CPU Time**
$R_M \times$	P_{RM}	16.2%
$X_M \times$	P_{XM}	
$R_S \times$	P_S^{Avatar}	72%
	$P_S^{\text{Projectile}}$	1.2%
	$P_S^{\text{all other objects}}$	-
	ϵ	-

Table 5.2: Measured Torque data and model variables

The input and output synchronization processing together represent only 16.2% of CPU time. The rest is taken up by simulation, loop overhead, and debugging systems that were turned off. We expect everything outside of the simulation to be trivially small, so we bundle it all into one ϵ parameter.

Looking at the simulation, each type of object that needs it implements the `SimObject` interface. It has a `processTick()` method that does the relevant physical simulation work for that object. Instrumenting `processTick()` should give us a per-type simulation time breakdown.

We instrument the two types of virtual objects we think may take up a lot of simulation time: user's avatars and the projectiles they fire. The former has complex movement and collision detection. There are many instances of the latter that may add up.

Another simulation gives us clear results. 72% of all CPU time in the system is spent simulating users' avatars (`Player::processTick()`), and 1.2% is spent on the projectiles (`Projectile::processTick()`) they fire. Between these two types of virtual objects, and the simple synchronization measurement, we have almost ninety-percent of Torque's CPU usage covered.

We also know the biggest part of our work: avatar simulation. However, that doesn't give us anything actionable.

Focusing on Avatar Simulation

We want to find out what makes `Player::processTick()`, P_S^{Avatar}, in our model, take so long. Hopefully the result will be due to something we can control.

It has a primary loop that attempts, up to three times, to find an object to collide with. It can also collide with a rise in the floor height, requiring an upward step. Normal gravity simulation covers the opposite case. The loop can find another object and a rise in the floor simultaneously. We updated the `simulate` frame to specifically model this step. It took us three instrument/simulate iterations to have enough instrumentation.

By then, we had a lot of instrumentation. Our `simulate` frames had over fifty components. Within the loop, we had counters for any branches taken for surfaces touched, timers within major sections of the work, including movement, collision detection, animation, etc. Outside the loop, we used flags for marking if branches were taken. We also had time-stamps for each major part of work.

The data from that final set of instrumentation gave us a clear view into what `Player::processTick()` was doing. Large parts of the method took very little time to run. The time-consuming parts were actually in other methods it calls. We have a simple version of it below, with the four major parts broken out. On the right, we have the average run-time in comments. The times are all from a sixty-user simulation.

```
Player::processTick() {
   // Small, constant Factors            //  .006  ms
   updateMove();                         //  .250  ms
   // More small, constant factors       //  .0012 ms
```

```
   updatePos();                          // 1.6    ms
}
```

After that iteration, we found `updateMove()` (movement) and `updatePos()` (collision-detection) to have nontrivial execution times. The others totaled to roughly 7.2 μs. `updateMove()` and `updatePos()` are both methods of `class Player`, and do not occur in any other classes.

`updateMove()`, while nontrivial in execution time, had a very low variance: 6.99e-4 ms^2. `updatePos()`, however, had a variance of 20.45 ms^2. We will focus on this during the last iteration.

We added more instrumentation, simulated, and determined that the collision count was the largest factor in the runtime of `updatePos()`. The time spent in `updatePos()` is shown on the vertical axis in Figure 5.3b. The values are spread horizontally according to the number of collisions processed in that data point.

We have a simple model for `updatePos()`. It gives an estimated runtime in milliseconds, based on the number of collisions (c):

$$t_{\texttt{updatePos}} = 0.05 + 0.165c^{1.3}(\text{milliseconds}) \tag{5.2}$$

The equation is based on a simple curve-fit of the data, initially assuming that the function was in the family $y - y_0 = Ax^B$. Different host computers should only require recalibration of the constant and linear scalars of Equation (5.2).

We compare it with N in Figure 5.2. The model leaves out many factors in the runtime of `updatePos()`, such as variation in search time through the scene-graph data structures for candidates to collision-check. These all contribute to the c-dependent diffusion found along the vertical axis in Figure 5.3b.

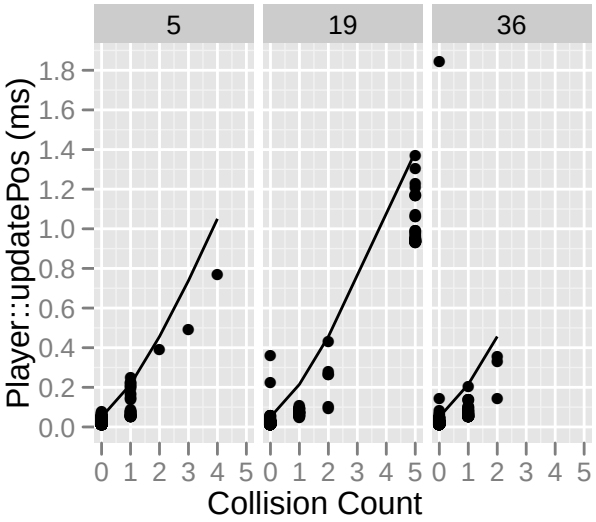


Figure 5.2: Runtimes for `updatePos()` vs Collisions and Model for $N = 5, 19, 36$

Now that we know the primary factor in `Player` simulation time, we can try to control it, or reduce our sensitivity to it.

5.1.3 Change Experimentation

We consider two options for changing Torque. First, we can run on multiple threads. This lets us run implementations of `SimObject::processTick()`, such as `Player::processTick()` and `Projectile::processTick()`, on a pool of worker threads. This way, our latency isn't bound by the sum of all their run-times. Second, we can try to reduce the simulation times by using our newly discovered factor: collisions. We consider both options.

Engineering: Evaluating Multi-threaded Conversion

We can also try to parallelize the engine. We don't even have to parallelize the entire thing, just `Player::processTick()`. It takes 72% of the CPU all by itself. We'll consider what's needed for the most straightforward way to do it, such as running simulation in a worker pool of threads and making the method thread-safe.

Unfortunately, that method tests for collisions against the rest of the scene graph. To make it safe for multi-threading, we need to add some mutual exclusion. We would have to serialize access to the objects individually, or to the scene graph as a whole.

Serializing access to the graph as a whole would serialize the longest-running part of `Player::processTick()`. The contention on that single lock would only allow one thread's `Player::processTick()` to run at a time. So, we have to look at adding locks to each object in the scene graph.

Unfortunately, that would affect a lot of code. There are many kinds of objects that implement the interfaces used by collision detection. If the amount of work was linearly related to the amount of code, we could still consider it feasible. However, there isn't sufficient documentation on Torque, nor does it encapsulate behavior well enough, to guide us where to add the locks.

We can't just add locks on every method body. The current APIs don't presume the need for atomicity when used, and the current code could easily leave objects in invalid states between calls to their methods. In the single-threaded case, there are no observers to those invalid states. If we add additional threads, one of those threads could observe an invalid

state and cause intractably hidden synchronization bugs that we aren't equipped to find.

People do this kind of retroactive development often. However, it's a lot of work. The benefits can (roughly) triple the current capacity: if there are sufficient additional cores for the avatar simulation threads. Additionally, the primary thread must only do the $\tilde{2}8\%$ remaining non-avatar work.

Can we get a better return on investment? We know that collisions are the reason that our avatars take so long to simulate. Can we get an easier capacity improvement by reducing collisions?

Analysis: Environment Alteration

We want to try modifying the "starter" terrain to cause fewer collisions. If we do that, we can make `Player::processTick()` take less CPU time, and we'll have more CPU time available for more shard instances.

We'll start by looking at the virtual space. The most recent `simulate` frame included sufficient information to construct a spatial histogram for the terrain: `x`, `y`, and `z` members. We ignore the z-dimension, as the virtual environment has very little overlapping spaces large enough to hold avatars. We call the two-dimensional spatial histogram a *density map*.

We can use it to estimate the frequency of avatar-avatar collisions in different regions of the virtual environment. The more people in an area, the more likely that they will collide.

We try putting something big and obtrusive wherever the users currently congregate. Hopefully, this will spread them out and reduce collisions.

Figure 5.3c shows four density maps. The grid size used for all four density maps is large enough to hold exactly one avatar at a time. Each density map is taken from a random sample of 5,000 `simulate` frames for avatars. The first ("A. Original") is the "starter" terrain we have now. We'll discuss the rest momentarily.

In the density map for our current ("Original") terrain, we can see a single cell substantially darker than the rest, at approximately $(250, 200)$. That is near the center of the virtual village. As an experiment, we put a small building there. We call the resulting new virtual environment "Modified."

As controls, we created two additional variants. The first variant is "Original-2," with a new building at a different part of the village. The second variant is a second variation upon "Modified," denoted "Modified-2." It keeps the new building from "Modified" and adds another tower to the periphery of the village. Figure 5.3a shows the variants.

Since we have modified the virtual environment, we also have to update the user load simulator.

Analysis: Simulator Updates and Execution

Whether we need a new user study now is an open question. The behaviors we saw before in the prior studies, in Table 5.1, necessitate much work with buildings. They are used primarily as hiding places, and we've added more of those. Additionally it is also more difficult find targets with more occlusions.

Subjectively, the terrain is almost all empty space. The users congregate in the center of

the virtual village to either pick up the health packs there, or to attack players going for the health packs. That part of the play hasn't changed.

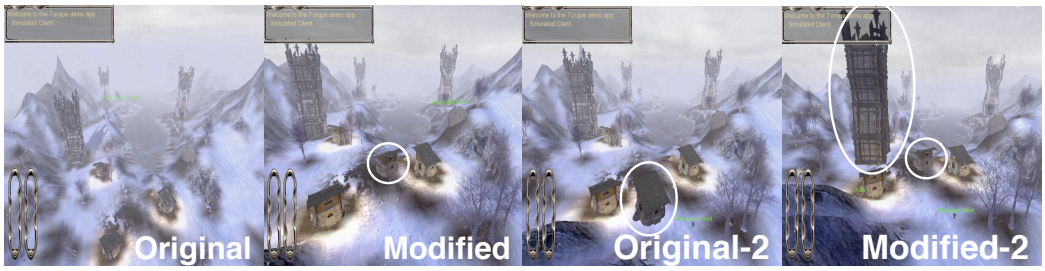
We don't believe that the additional buildings will shift user behavior significantly enough to impact our tests. We may be wrong, but wrong or right, we won't be running another user study for the change to the virtual environment.

In normal usage, we expect semi-regular user studies as a matter of course, similar to re-running unit tests regularly. Like unit tests, they are small tests with few users, but would cover a large proportion of the virtual environment. They would flag unexpected changes in user behavior, similarly to a new unit test failure indicating that some code was broken. A larger user study would have to be run on the area, to re-calibrate the load simulator.

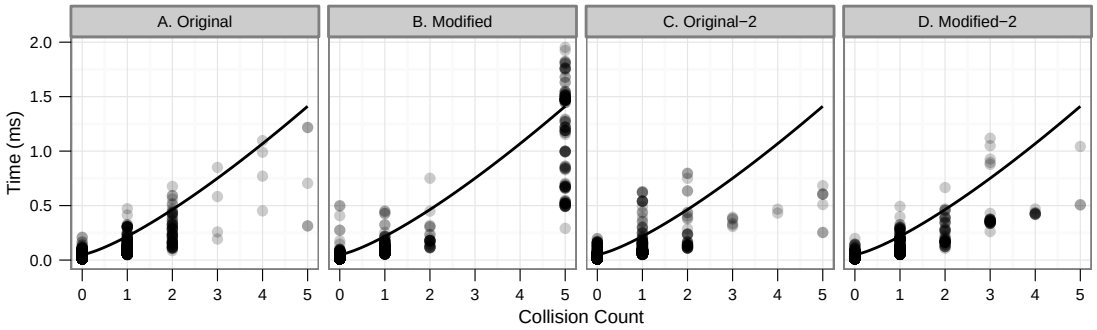
Without new user studies, we still have to update the load simulators' internal maps of the virtual environment. The maps form the basis of simple path-finding inside the virtual village, and act as proxy for a user's ability to see the virtual environment. We believe that the simulators' movements after the map updates will reasonably estimate how users would react to the new buildings.

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

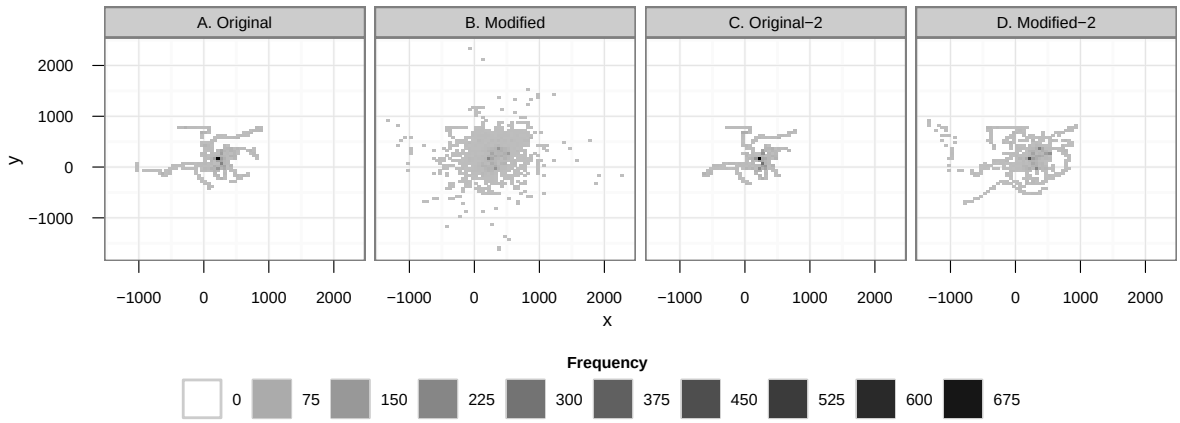
All the plots show the data of 5,000 randomly selected avatar simulation steps. Each is from



(a) Level Screenshots



(b) Collision Detection vs Collisions, with Model Overlay



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

Figure 5.3: Level variants, collisions, and densities

a different simulation, with sixty users.

Figure 5.3b shows the collision detection times in all four variants, with the model overlaid. Note that there were fewer times when there were more than three simultaneous collisions in the "Modified" and "Modified-2" variants, than in the "Original."

Figure 5.3c shows the density maps for each variation. It shows substantial variation. The darkest point in the original map is substantially lighter in the "Modified" version. It is still present in the first random variant, "Original-2." The combined intentional and random variant, "Modified-2," has a reduced, but still substantial, dark spot there. Apparently, the reduced space caused by the taller building coming from "Original-2" caused enough space pressure to push users back around the area of contention, despite the lack of clearance around the building first added in "Modified."

In terms of reducing areas of high density, the intentional modification alone seems the most effective. In terms of overall computational load, the original average time through the main software loop was 10.66ms, and the modified variant 10.36ms — a tiny three-percent improvement.

Both give maximum simulation quality. Torque, unsurprisingly, has a dynamic adaptation mechanism for quality versus latency. Its simulation system moves each virtual object forward by a discrete amount of time, called a "tick." Each tick is thirty-two milliseconds. If the total time through the primary loop takes more than one tick, it makes the tick longer. The tick gets longer in increments of thirty-two milliseconds. With mean simulation times far less than a single tick, our mean simulation-step is the smallest it can be. That gives us the best simulation accuracy.

The run-time variance for `Player::processTick()` for the original was 43.42ms^2. The standard deviation (6.58ms) was more than sixty-percent of the mean. This causes two problems. First, one must substantially over-allocate CPU resource to run the DVE system for the peak simulation times. The ninetieth percentile is 30.42ms, for a single user's simulation

step. A single user hitting a peak can take the entire simulation past a single tick's time.

That causes the second problem: jitter in simulation quality. As we add more users, the likelihood that one of them causes a jump in the tick-increment increases. We could see the time-interval for the simulation move frequently between higher and lower values. For small user counts it's worse, as a small change in tick-increment is much larger, percentage-wise. Larger user counts should stabilize into plateaus of lower simulation quality, with the occasional many-collision event causing further temporary drops.

On that area, the modification has more impact. The variance in simulation time was reduced to 20.65ms^2. The ninetieth percentile is substantially lower at 23.99ms, a 21.73% improvement. The other two variants had worse mean runtimes, 10.91ms for "Modified-2" and 10.99ms for "Original-2." The variances were 86.05ms^2 and 87.32ms^2, respectively. The additional buildings, placed poorly, seemed only to increase collision pressure in the region.

In comparison to multi-threading a substantial part of Torque's source code, we've done pretty well. The multi-threading work would only have had higher variance from the added synchronization. It did give us options for CPUs with higher core counts and slower single-core performance. That may fit better with CPUs that modulate core frequencies by the number of active cores.

We can continue to tune the virtual environment further for a flatter collision density. The changes only take a few hours to do, and don't risk introducing latent bugs. If we find that we don't get enough additional capacity from these efforts, we can reconsider multi-threading the DVE system.

Summary

We used the DSE process on Torque, to find and verify ways to increase the number of DVE system instances we can run at once. Each instance supports sixty users in their own copy of a small virtual village.

We ran about a dozen user load simulations. The first told us that we were CPU-bound. Three more told us that the avatar's physical simulation was taking up 72% of the CPU. A few more found its primary factor: the number of collisions. We created three variations of the virtual village, and simulated all of them.

After the simulations, we had a reasonable variant of the village that took twenty-two percent less CPU time to run at the ninetieth percentile. If we want better confidence, and thus lower quality jitter, the variant will fare even better in comparison.

5.2 Networking and Power in the Quake Engine

We used `ioquake3` [46] for our analysis, with the original Quake III Arena terrain set. We used the `q3dm17` level, made solely of small floating platforms. Falling off of a platform killed the player.

Quake III has two hard-set limits on how many users it will hold. Globally, a Quake III server can have a configurable maximum that defaults to twelve. Each level also has their own immutable maximum set for in-game balance and available virtual environment.

We have already demonstrated the DSE process for CPU-based analysis with Torque in

Section 5.1. CPU analysis will probably be a popular use for the DSE process. However, the DSE process gives enough information to make higher level trade-off decisions than the design of the virtual environment.

We will evaluate Quake III for deployment on mobile devices, such as smartphones or tablets. Specifically, we'll look at the appropriate network technology to use. We have a few concerns. First, which network technologies on these devices support enough bandwidth for a client, or server? Second, how much power do they consume? A device with enough energy for a day of use in one configuration can be drained in an hour in another configuration.

The DSE process will give us our network-usage model, and we'll use it to evaluate networking options.

5.2.1 Early DSE and the Load Experiment

Preflight

With the existing body of work on the Quake series, we looked for modern concerns on an older engine. Mobile devices are an interesting new deployment option for multi user DVE systems. We aim to understand Quake's networking characteristics, for applicability to the different wireless technologies available on a modern mobile phone.

Analysis Cycle: Load Experiment

We ran user studies as a regular weekly game with Google employees using their own computers. Users were constantly running and shooting at one another on the ground, mid-jump,

or mid-fall. Most of them had played the game before, a few of them were experts.

We found a fast-moving, tactics-driven playing style across all users. Each one varied greatly in aim, speed, and coverage of the map. When not tracking one user and observing the entire sequence executing in parallel, a simpler structure emerged. A fast cycle between four primary behaviors was found: chasing another user, moving to a new place, getting a health kit, or a new weapon or ammunition. We found that the relative percentages for these were 40%, 30%, 10% and 20%. Through observation, we estimate the mean behavior duration as roughly a second and a half.

The Load Simulator

Behavior	Percentage	INIT	MOVE	POST
CHASE	40	Target	*none*	Shoot and approach
PLATFORM	30	Path	Along path	Shoot and approach
HEALTH	10	Health-pack and path	Along path	*none*
POWER-UP	20	Power-up and path	Along path	*none*

Table 5.3: Quake users behaviors

Table 5.3 shows the top-level decision table for the load simulator. The first column identifies the behavior. The second column identifies the desired frequency.

A state machine implements each behavior. Each has three sub-states: `INIT`, `MOVE`, and `POST`. The first (`INIT`) sub-state selects behavior-specific state, the second (`MOVE`) executes any necessary movements, and the final (`POST`) completes any final activity after having moved to the destination.

The values in the `INIT` column in Table 5.3 identify the selection. The `MOVE` column identifies any movement across a selected route. The `POST` column identifies the activity

executed after INIT and MOVE.

The simulator selects a behavior and runs it for one and a half seconds. Next, it increments a behavior-specific *count*, and a global *behavior counter*. After the increment, it determines the *relative representation* of each behavior as the ratio of its count versus the global behavior counter. Finally, the simulator selects a new behavior with a representation lower than in the second column of Table 5.3.

The simulator selects the closest other user or object, as needed. It represents the platforms as sets of points. A bit-mask identifies which points were directly reachable from one another. Points between platforms were also marked as co-reachable, and the path-finding algorithm selected points to reach its destination. Even though there were only forty-five points, a full traversal of the graph for an optimal path wasn't feasible in real-time.

The simulator uses an A* path-finding algorithm, executing a depth-first search for the destination platform from a starting point. Each iteration of the algorithm selects the step moving it closest to its destination. The algorithm back-tracks when stuck. It selects the first path constructed this way as its route. Each platform is split into convex regions, and the simulator directly moves inside each region. It uses co-reachable points to move across regions, even if the regions are on different platforms.

The only weapon on this level was a rocket launcher. It slow to fire, but was able to hit targets at distance with a damage radius of about two meters. When in `PLATFORM` or `CHASE` behaviors, the mouse-state was modified to have the "fire" button held down. This is roughly identical to what participants did in the user study.

Instrumentation

We set up a `ppt` frame that stored the time of the packet (sent or received), a flag indicating if the packet was going in or out, and the size of the packet. The low-level calls to send and receive packets were instrumented. The relevant macro calls are inserted at each `send(2)` and `recv(2)` call.

We use a small variation of Equation 4.3, as Quake's client sends a very different update from the one it receives. The former is a "command" indicating which direction the avatar should move, and which, if any, type of projectile it should fire. The latter is a subset of the virtual world state.

Our version is:

$$\text{Bandwidth} = X_M \times B_X + R_M \times B_R + \epsilon \tag{5.3}$$

Where the B term of Equation 4.3 is broken into B_X for the average amount of bandwidth needed for transmission, and B_R for received data. The instrumentation measured the total bandwidth used.

5.2.2 Modeling: Results

We ran a 15-user simulation and collected a little over 2.5 million values over a short (9.3-minute) interval. We instrumented the server only. Two percent (51,120 I/O records) was lost in the process.

Table 5.4 shows basic statistics of the collected data. While constantly sending with only an average $54\mu s$ between packet transmissions, there was a fairly bursty receive interval in clusters centered over 8.043ms.

	Min.	**1st Qu.**	**Median**	**Mean**	**3rd Qu.**	**Max.**
Send Interval ($\frac{1}{X_M}$, ms)	0.029	0.039	0.048	0.054	0.063	0.288
Receive Interval ($\frac{1}{R_M}$, ms)	0.011	3.130	10.630	8.043	11.000	22.140
Sent Bytes (B_X)	15	21	27	28.59	32	338
Received Bytes (B_R)	21	23	24	27.24	25	234

Table 5.4: Quake I/O statistics

	Variance	**Standard Deviation**	**90^{th} Percentile**
Send Interval ($\frac{1}{X_M}$)	0.0005	0.0224	0.12
Receive Interval ($\frac{1}{R_M}$)	19.8384	4.4540	21.41
Sent Bytes (B_X)	241.8887	15.5528	75.25
Received Bytes (B_R)	652.5949	25.5459	103.88

Table 5.5: Quake dispersion statistics

We calculated the variance, standard deviation, and one–sided 90^{th} percentile value for both the intervals and byte sizes in Table 5.5. The percentile was calculated using a variation of Cantelli's inequality: $P(X - \mu \geq k\sigma) \leq \frac{1}{1+k^2}$, with $k = 3$. Using the data in Tables 5.4 and 5.5, we are able to determine that a 15-player Quake 3 session would require a mean 2.60 $kBps$ send transfer rate, but a 619.09 $kBps$ receive rate. Balani [49] gives transmission power formulas for three networks: Bluetooth 1.1, 802.11, and GSM/EDGE. While not listed in the report, we presume from the relevant time period that it discusses 802.11b.

We only have transmission-power data, but will assume that the reception requirements are similar. Table 5.6 lists the power equations and the application of them to the mean and ninetieth percentile transmission rates.

The receive rate is too high for Bluetooth (1.1, 1.2, or 2.0) and GSM/EDGE. Neither of

these networks could support the server's data requirements. However, we can (roughly) take a fifteenth of our send and receive amounts, assume that they'll get transferred at most as much as the server is doing, and get our per-client rate numbers. They should be in-range for all networks. While GSM/EDGE lets us connect to a central network that can host the server, Bluetooth uses pairwise links. Bluetooth could handle the bandwidth for each client but it has no way of relaying that data to a server.

	BT 1.1[1]	**802.11b**	**GSM/EDGE**
Equation	$0.0022b + 14.8$	$0.0067b + 316.67$	$0.0905b + 20$
Server Send μ (mW)	N/A	334.08	N/A
Server Send 90^{th} (mW)	N/A	345.28	N/A
Client Send μ (mW)	14.80	317.78	34.94
Client Send 90^{th} (mW)	15.83	319.79	62.20

[1] While the transfer rates are feasible for clients on Bluetooth, they are not for any server.

Table 5.6: Continuous power draw for transmission, by technology and bytes/sec (b)

If we want to run a server on the same wireless network as the clients, the clients will pay a substantial power price, as the most power-consuming Wi-Fi system is the only one capable of supporting the required bit-rate. Single-hop latencies for 802.11b are in the range of 14ms as found by Bicket [50] — the best of the three technologies. The convenience of running a server on one of the mobile devices, paired with the low latency, may be desirable for many.

If the trade-off for power is too expensive, users can use about ten-percent of the power consumption with GSM/EDGE. The network latencies are in the range of 150ms [51]. The users also do not have to be in physical proximity in order to play. If there was a way to use Bluetooth links to connect to a server, such as a single dedicated machine with many radios, then Bluetooth would be a much more power-savvy option. Bluetooth uses half the power of GSM/EDGE, and has slightly more than one quarter the network latency. We used a 40ms

transmission interval for Bluetooth: the lowest found in our source [49]. If we were to run Quake 3 on mobile devices, we could simply run an Internet server and let mobile clients connect. Their power usage would be better than with the Wi-Fi case.

5.2.3 Summary

We ran a user study and characterized the load on the system as a set of user behaviors. The behaviors were then implemented using modern techniques for non-player characters: path-finding, chasing, and opportunistic health and power-up selection. We instrumented the DVE system for its network utilization characteristics, analyzed it for the amount and frequency of data transferred, and compared well-established wireless network technologies that one would typically find on a modern mobile phone.

We found two options: one case of using 802.11b for its low network latency, at the cost of a substantial power consumption requirement, or GSM/EDGE at ten to twenty percent of the power consumption, with a cost of over ten times the network latency.

If we had a particular model of a mobile phone in mind, we could go farther and run DVE system on a larger part of the mobile phone's power consumption profile, such as the CPU use. With models of both, we can run the engineering or analysis cycles of DVE system to experiment with changes to reduce power requirements. For example, a more aggressive compression algorithm or a reduced network update rate may be worthwhile. Alternatively, more data may be sent at the lower rate to compensate for the constant factor in the power equation. With a longer battery life, this can result in longer-lasting DVE system sessions,

possibly resulting in more users choosing to participate.

Chapter 6

Discussion

In our research we tried to identify and quantify the relation between user behavior and system performance. The Casual Chain described in Section 1.2 provided a basis for an iterative process for scaling DVE systems. We built on top of the existing techniques for user load testing and developed a way to accurately simulate specific user populations. We combined the user load simulation system with a custom performance measurement tool to develop a rapid, iterative modeling methodology, a DVE Scalability Engineering (DSE) process. The DSE process allowed us to find the major performance factors in a DVE system while minimizing the effort related to testing, modeling, and changing the DVE system to enhance its performance.

We describe all of our primary contributions next.

6.1 Measuring Scale

The first problem we addressed is how to test the DVE system's scalability. The scalability testing has two parts: providing a realistic user load to a DVE system and measuring the DVE system performance.

The user load testing by conducting a user study with a large number of participants can be slow, expensive, and difficult to coordinate. We proposed conducting a single (or a small number) case study with small number of participants — large enough to get a representative sample of users' behavior, but not large enough to provide information about heavy user load — and capture the observed user behaviors in the DVE system software. We created a new, low-overhead instrumentation technique that avoids quality degradation.

We verified our load simulators through a minimal instrumentation developed before each user study. Further verification and tuning is possible through additional user studies and additional instrumentation. The instrumentation introduced no measurable increase in the DVE system overhead.

6.1.1 Load Simulation

Traditional user load testing methods through user studies are expensive, time consuming, and had repeatability problems. It takes substantial time and effort to organize participants and configure computer hardware and network. Additionally, the participants' behavior may be affected by the testing setup. Some of the participants' strategies for the DVE system use may differ from their strategies when using the DVE system in real world situations.

The user study participants may consider long term strategies that are not relevant to the conducted study, even though that attitude inadvertently gives less representative user load.

Our user load test method lets us build repeatable tests. We can take the results from controlled user studies and implement a model of user behaviors. We can compensate for things we can't directly test, such as an under-representation of long-term strategies in user study data. We test for shifts in user behavior and for shifts in response to virtual environment changes.

We believe the user load simulation approach can be an effective solution for the DVE user load testing. First, we only have to match the same total user load profile for the user population. We do not have to match the user behavior of any individual, real or idealized. In observations of load simulators for both Quake III and Torque, the only (subjectively) distinguishing characteristic between the simulated and human participants, was a larger behavioral variation between the human participants. The workload for the DVE system in either case was similar.

Second, the user load-simulator behavior was specified in the virtual environment's domain. The simulator's user behavioral distribution replicates the users' behavioral distribution. The act of constructing the simulator, and adjusting it through further user studies, is a part of the DSE process. This activity helps with reasoning about the user load on the DVE system, and completes the conversion from Causal Chain (Figure 1.3) elements to the DVE.

There are few ways to measure how well a user load simulator works. When measured exactly under the same conditions as human participants, we can measure and compare the resource usage and QoS. We used `top(1)` to measure CPU usage to verify similar performance.

When scaling up the number of simulated users, the user load equivalence (compared to human participants) becomes difficult to measure. Our research has focused on DVE systems that keep this problem simple: users work individually, the environment is stateless, and the general state of the virtual environment is largely constant. No high-level strategy was required (or observed).

There are DVE systems where the user load equivalence measurement problem is not as simple. Those DVE systems will require better metrics for determining load equivalence between the simulated users and human participants. One expensive method is to execute more user studies every time there is new instrumentation, to verify that the user load simulator still matches the human participants behavior.

6.1.2 Instrumentation

The user load test is affected by the dependencies between user behavior and DVE system performance. We need to measure the DVE system performance to identify these dependencies. Traditional measurement tools like `oprofile` [30] or `gprof` [32] add a substantial overhead and provide information that gives little insight into the reasons for current performance. We have developed a new tool, called `ppt`, that collects programmer-specified system data elements, groups these data elements and transfers them, via a buffer, to an external disk for offline analysis. The use of the buffer and a helper process helps keep the instrumentation from interfering with the DVE system software's performance.

We've combined the `ppt` tool and user load testing to develop a modeling methodology that

finds primary factors of resource pressure. We separate the resources the DVE system needs for full quality versus those the DVE system uses to compensate when quality is reduced. The tool allows us to use a minimum (and controllable) amount of additional computer resources to measure the DVE system. Our modeling methodology uses the tool to trace resource usage from top-level DVE system control or state structures down to their controlling factors. Most of the DVE system software and virtual environment don't substantially affect resource usage, and the methodology lets us ignore them safely. In other words, we can focus only on those DVE system software components that are critical for the DVE system performance.

Existing methodologies tend to use opaque "whack a mole" techniques, where heavyweight instrumentation represented the DVE system as a regular software system. The DVE system is represented as a set of function (or method) bodies, threads, or clients giving requests (or queries) to support systems. The software component initially requesting the resource shows up without context and consequently gets batted down through local optimization or redesign. The DVE system is eventually modified into a workable but not necessarily optimal implementation. The individual optimizations and redesigns didn't have any context or understanding of the connections to the larger performance structure of the DVE system.

Without that context, the performance of the DVE system can still be brittle and hard to change due to the optimizations. Also, many of the optimizations may have been unnecessary. They may have simply been intermediate elements in whatever representation the measurement tool used. For example, a string operation may show up quite high in the list of most CPU-consuming functions, even though the root cause is an inefficient synchronization protocol that re-transmits strings the recipient already has.

`ppt`'s instrumentation technique worked because it removed most of the burden from the DVE system software. `ppt` also allowed data to be lost, in exchange for lower overhead. The in-process instrumentation just copied data to a memory buffer. A second process occasionally scanned the buffer for new data and saved it to disk. After the user load test, we analyzed the data using a standard statistics package. The buffer-write procedure didn't even need atomic operations. Small amounts of additional buffer space was used for sequence-number data, which served as the loosely-coupled synchronization and data-tracking mechanism.

Unfortunately, the data loss isn't uniformly random. During bursts of high activity, the transfer buffer can fill and overwrite older data before it's picked up by the reading process. That property sacrifices the instrumentation data to maintain quality in the DVE. Larger buffers can mitigate the problem, but consume memory and can thrash cache.

6.2 Modeling Scale

Our "follow the resource" tracing technique for factor identification allowed us to understand how and why a DVE system scales in a certain way. The model extensions based on the NVE Information Principle [4] served as an good starting point for each resource study and were easy to ground in the measurements in all three DVE systems (Asteroids, Torque, Quake III).

In Torque, the collision detection mechanism worked well for small numbers of collisions. However, the run-time was exponential in the number of collisions. Traditional instrumen-

tation techniques would have shown a small ten-millisecond average runtime, and would not have explained some of the DVE system performance consequences. The larger runtime causes high jitter in the network updates as many collisions can take more CPU time than the defined inter-transmit interval for network updates. Even with a low percentage in `top(1)`, we were observing the ten-Hertz transmit interval dropping to six. Similarly, the simulation interval rarely went under ten "ticks" — with each being 32 milliseconds of simulated time — even when `top(1)` showed a mostly idle processor.

In Asteroids (Section 4.3), we found a major library-dependency issue that consumes substantial amounts of CPU time. In Quake III (Section 5.2), we quickly determined the networking requirements and were able to predict power requirements for different network communication link options.

We think that continued use of our DSE process will help us identify patterns in the specific DVE system models that can lead to more general DVE system models. Those models can extend the NVE Information Principle with more input variables in terms of design choices, parameters, and user behavioral properties.

6.3 Changing Scale

Our DVE Scalability Engineering (DSE) process provides a workflow for user load testing and modeling. It also allows a DVE system designer to explore the effects of the DVE system design changes.

The DSE process is quite effective and allow us to quickly build accurate, relevant, and

concrete models of the DVE system's performance structure. We focused on ways to improve scalability, through we could also experiment with ways to trade excess scalability for additional DVE system features.

We used Torque to explore scalability issues, in part because Torque doesn't have a fixed upper limit on the number of users. DSE's analysis cycle gave us a very inexpensive, highly-targeted change to the virtual environment that substantially reduced the user density. The reduced user density, in turn, reduced the collision detection processing load — the standard deviation was cut in half, and thus the 90^{th} percentile CPU time was cut by roughly 22%. For a server machine running multiple server instances that means that several CPU cores (or processes) could be freed to run more instances.

We tried to minimize power consumption when running Quake III on a mobile phone platform. We focused on the available network communication links and identified a massive trade-off between power consumption and network latency. The GSM (2G) network link had latencies too high for Quake III (around 150ms) when 60ms is an upper bound for this type of a DVE system (Section 2.2.5). That leaves 802.11b, or a Wi-Fi network link as the only option, requiring 5–10 times the power consumption. Additionally, the GSM network link didn't have the network bandwidth needed to support communication with a Quake III server, so even if GSM had lower network latency, a mobile phone platform couldn't be used due to the network bandwidth limitations.

Chapter 7

Conclusions and Future Work

We have introduced a new DVE Scalability Engineering (DSE) process that addresses these three major challenges for DVE design: effective DVE system performance measurement, understanding the controlling factors that determine system performance/quality and determining the consequences of Distributed Virtual Environment (DVE) system changes. The DSE process provides a user load-test modeling method and allows a DVE system designer to measure, understand, and change the scalability of a DVE system.

We demonstrated the utility of the DSE process for scale-related analysis in two real-word cases studies: Torque and Quake III DVE systems. The Torque case study illustrated the use of DSE process for scalability issues (in terms of the number of users). The Quake case study demonstrated scaling a DVE system for use on a mobile phone platform. We analyzed Quake III to model the network requirements that were used as a basis for network latency/network bandwidth/power consumption trade-off analysis. The DSE process has also been shown useful in a very small DVE system, Asteroids. We believe the DVE process

can be effectively used for very large DVE systems with large number of users.

The DSE process builds on top of the Causal Chain [1] and the Networked Virtual Environment (NVE-IP) Information Principle [4] to provide additional insights and understanding how the behavioral consequences of user interactions in the virtual environment (QoE) affect the DVE system performance (QoS) and three primary DVE system resources: CPU time, memory, and network bandwidth.

The DSE process provides a foundation for exploring new research challenges and opportunities. By using the DSE process we can better understand the dependency between performance (QoS) and scale (QoE). This, in turn, could lead to better understanding of a global optimal balance in QoE, QoS, user tasks/goals in the virtual environment, and required DVE system resources. Applying the DSE process to various DVE systems will help us identify the common trade-offs and DVE system design patterns for scalability.

Appendices

Appendix A

Additional Related Work in DVE Scalability

Substantial research exists in DVE systems, their performance, other distributed systems, and performance of networked systems. We will cover each major DVE system: physical simulation (the function of the "physics engine"), network synchronization, and graphical rendering. We will cover each major system in turn. For each, where available, we will cover major approaches for implementation, current and past research in each, and research in related systems.

We follow our discussion of the three major systems and follow up with some research results in performance.

A.1 Physical Simulation

A large part of a DVE system's software is its physical simulation system. This system moves and mutates objects in the virtual world at high (60+ Hz) rates. Each one of those move/-mutate cycles involves a simulation of a small amount of time (a "time step"). Each time step executes through every object, moving them and emitting collision events as needed. These events are then routed through the normal event system, ending at the proper responder points, in code or in the scripting engine.

DVE engines vary significantly in the level of complexity they provide in this area. Some provide basic linear movement and simple box-bounded collision, while others provide constrained motion, geometry-accurate collision, etc. The Torque engine provides only the simplest of physics, often even failing to maintain numerical stability between frames. Other DVE systems can provide significantly better physics simulation, or it can be acquired through middleware like Havok [52] or the Open Dynamics Engine [53]. They provide basic rigid body simulation, joints, friction, springs, and their own collisions. Also within the field of physics simulation are things often too complex to simulate in real time for DVEs, such as suspensions, hydraulics, or drive-trains. However, in DVEs where the specifics of some of these matter, the individual parts are modeled, such as Grand Turismo IV [54]. It realistically models vehicle dynamics, aerodynamics, and small variations in terrain's effects on a race car's behavior. Other DVEs also model deformations of vehicle crashes, although accurate modeling is still rare.

Eberly [55] describe the mathematics required for that simulation, including a basic differ-

ential equation solver to properly move objects in even the most basic motions. Schneider and Eberly [56] and Ericson [57] describe useful techniques for collision detection.

Ageia [58] describe a dedicated co-processor for physics simulation. Sony Computer Entertainment [59] describe the Cell processor, which combines a PowerPC primary CPU core with eight vector processors. It is used on DVE systems running on the PlayStation 3 platform.

A.2 Network Synchronization

The networking engine in a modern DVE is typically sophisticated. Ferguson [60] and Frohnmayer and Gift [61] describe two such engines. DVE systems often have to transfer a mix of reliable data, such as text chat messages between users, and unreliable data, like the position of a moving object. Postel [62] describes TCP's retransmit–everything behavior can cause undesirable delays, and Postel [63] describes UDP's lack of prioritization or retransmission.

Ravindran and Lin [64] analyze the trade-offs involved in using higher-level networking protocols (such as TCP) over their simpler counterparts (such as UDP). Features such as message ordering, atomicity, and level of delivery are considered in terms of their costs. The primary concerns are protocol complexity, in terms of different states the connection can take, and the network costs of additional messaging required to support the features.

Valve software's Yahn W. Bernier [65] covers methods for Half-life and a bit of Quake III.

In a DVE system, each object's movement is represented as a change in a tuple representing its position, orientation, and velocity. The tuple is often called a Dead Reckoning (DR) vector. The physical simulation systems in DVE systems move objects along their DR vectors,

and the network synchronization systems receive new DR vectors for each object. This way, a DVE system node doesn't need to send updates to other nodes when the extrapolated DR vector state is the same (within some ϵ) as what would have been in the update. The Dead Reckoning system also allows some laxity in the clients' transmissions of the same; if the user keeps moving at the same rate in the same direction, then the DR vector is unchanged and doesn't need transmission.

A.2.1 Protocols

DVE systems are "soft" real–time processes, and they have multiple streams to send to the same host with dynamically varying constraints. Additionally, the communications should resist tampering be either the user or an intermediate network node. Neither TCP nor UDP does satisfies its requirements. DVE systems typically place a new protocol atop of UDP, containing the components of a specialized system for the DVE's particular needs.

DVE networking systems work primarily for three purposes:

1. *Authentication* — Allowing users into the system.

2. *Data Download* — Getting levels, textures, character positions, etc. to the user for the relevant parts of the DVE.

3. *Event Updates* — Letting users know about changes to the virtual environment, after the initial data download. These updates can include Dead Reckoning vectors for moving (or moved) objects.

The Tribes [61, 43] system was well-optimized for high–latency, low–speed technologies such as modems and early ISDN systems. It can serve as an illustrating and representative example.

Example: The Networking Engine of the Tribes System

The Tribes system included three layers of traffic within each packet: user movements, events, and object state replication. The first provided a compact representation of the last position, heading, and speed of the client. The next provided reliable (e.g. retransmitted as needed) transport of application–specific events that weren't represented in the normal states of an avatar or other virtual object. The final layer provided a mechanism for object states to propagate, automatically, over the network stream. Object states are prioritized, compressed, and sent over the packet stream as needed to ensure reasonable synchronization even in the cases of latency, jitter, and loss.

The object–state propagation system was called "ghosting," for its ability to insert foreign objects into a local machine's scene graph. Each object would encode its state changes in a minimized form for network transmission. The ghosting engine would prioritize and retransmit the data as necessary to keep the DVE running smoothly.

The ghosting system, which could leak useful information about objects otherwise unknown to the user, uses a dynamic ID mechanism to hide the identities of objects from anyone but the intended recipient. By requiring extensive state management between both sides of the network connection, the Tribes engine makes state extraction difficult from the stream.

Together, these functions provide a useful transport mechanism for DVE data between a client and server. They would likely work well for a peer to peer mechanism, but that hasn't been tried yet. By prioritization and compression, it handles low bandwidth situations. By retransmission, it handles noisy or lossy links. By using dynamic IDs, it makes tampering harder.

Optimization: Booster Boxes

Despite the efficiency of Dead Reckoning and the potential optimizations available with a customized network protocol, DVE systems often still have much data to process. Bauer, Rooney, and Scotton [66] suggest an overlay network of application–specific routers they call "booster boxes," operating at ISPs on the edges of the Internet. These boxes would perform low–level event routing at high speeds, with ASIC support as necessary. The goal would be to reduce the amount of traffic hitting the servers by providing a more direct route for packets that should be sent to other clients.

The booster box idea allows the parallel transmission of the client data to the server and other clients. This prevents the additional latency involved in hopping all the way to the server and then to the other clients. Of course, there are some open questions about security in this matter — only the server can really know if the other clients should have known about this data. After all, the other clients may be running cheats that make the walls semitransparent.

A.2.2 The Client / Server Topology

Often a single "server" DVE system software process acts as a "hub" for a DVE system. Each user operates a "client" process that connects to the server over a network link. The Client/Server topology is well understood. Institute [67] provides a historical overview.

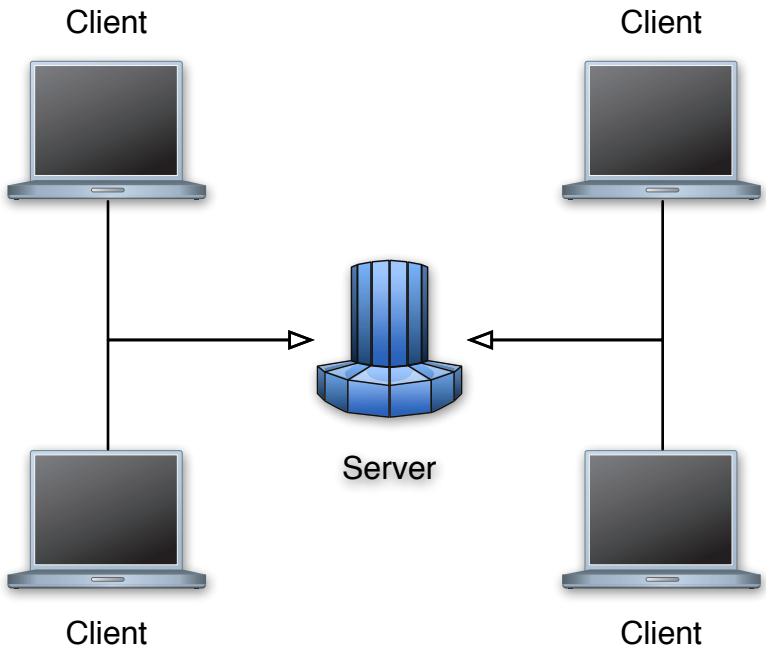


Figure A.1: Client/Server Network Topology

A DVE system's server process acts as the controller for the virtual world. Clients log into the host and send their movements and actions to the server, which then distributes the updates to the other clients, as necessary.

For First Person Shooter (FPS) DVE systems, the client-server architecture is very common. It has some distinct advantages:

1. *Authoritative Ownership of Objects* — The server has full control over what clients can do to themselves, each other, and other objects in the DVE. This eliminates problems

with shared ownership of data. The server's view of the world is singular and canonical. The server keeps the world consistent, preventing disputes.

2. *Event Filtering* — The server knows everything about every user and can filter what each user knows about the virtual world. The filtering reduces the data link and computational requirements for the system, as well as preserving secrecy for parts of the virtual environment that the user should not know about (e.g. if someone is behind the wall).

3. *Isolation* — The server process may run on one of the clients' machines, and the entire set of participating computers involved in the DVE system is isolated to that server and the set of clients.

The Torque [43], Tribes [61], and Quake [60] systems are examples of client–server topologies. Entertainment [68] is the only FPS-style DVE system that does not use a client/server topology. The *authoritative* and *event filtering* advantages help stabilize consistency in the DVE state and prevent data from "leaking" to users who should not have it, and from malicious users corrupting the virtual environment state to their own advantage.

The *isolation* advantage avoids global network bottlenecks. Users wishing to participate in a copy of the virtual environment can use low-bandwidth, latency-insensitive, and computationally-cheap rendezvous systems to find and publish running servers.

The limitations of client/server systems are better understood than Peer to Peer systems. The server can be a bottleneck and is a single point of failure.

A.2.3 Peer to Peer Systems

Androutsellis-Theotokis and Spinellis [69] provides an overview of Peer to Peer (P2P) technologies. P2P technologies offer potential for very large virtual environments. Using these technologies, a DVE system may not need to have a central bottleneck or point of failure.

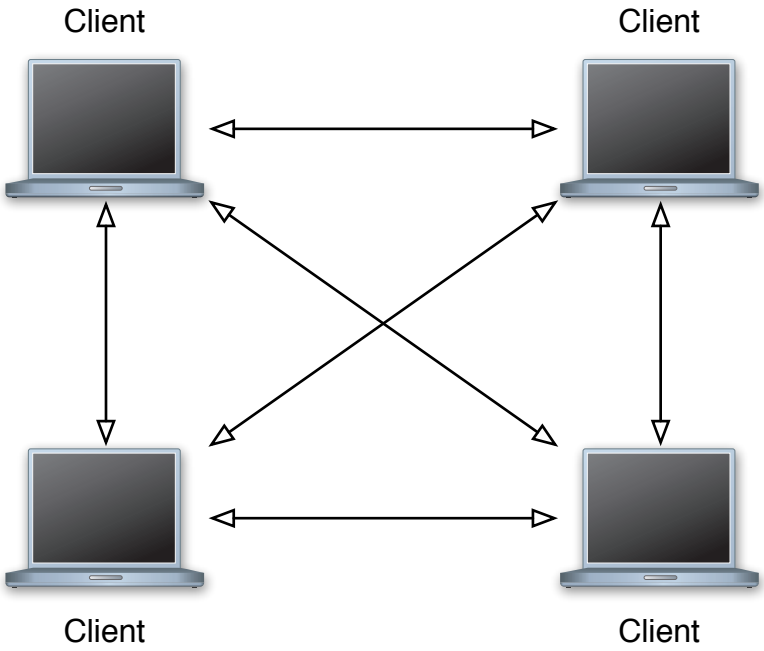


Figure A.2: A Simple Peer to Peer System

One common application of P2P technologies has been in file-sharing programs designed to let people share files on each others' disks. These programs connect to an indexing service, which enables anyone on the network to query and download files from anyone else. Once a desired file is found, the programs can transfer files between each other. Some indices are implemented as a Distributed Hash Table (DHT) that maps content to nodes in the set of P2P nodes. The DHT itself is split around the P2P network, and P2P algorithms can find, read, and write data into this table. As a traditional hash table, it easily handles large amounts of content with low sensitivity to the number of hash buckets (i.e. hosts in the P2P

network).

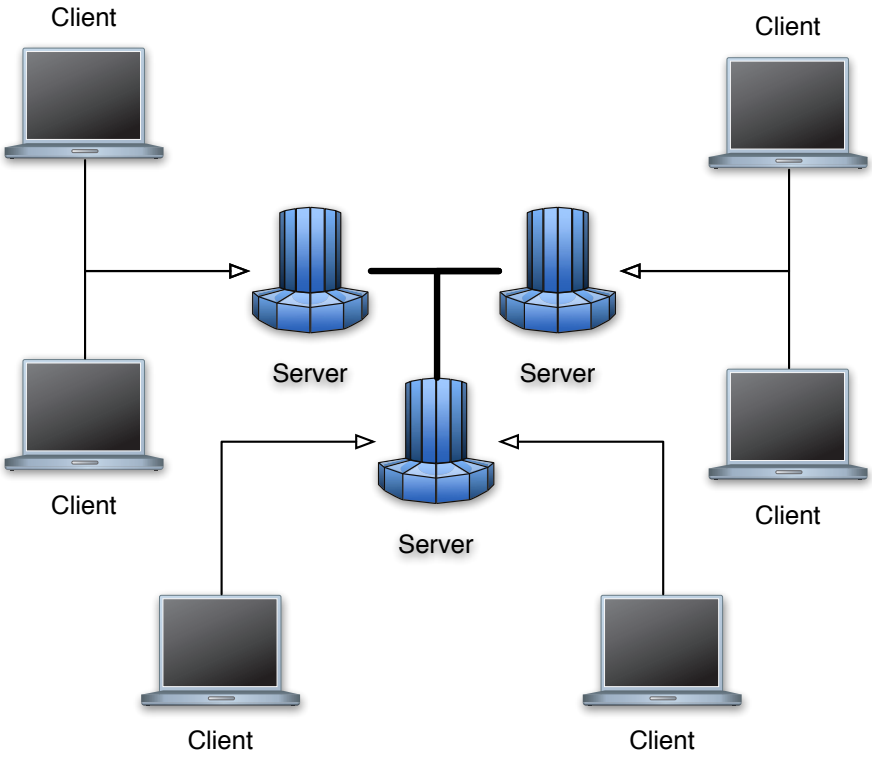


Figure A.3: A Hybrid Peer to Peer and Client/Server System

Rolia, Vetland, and Hills [70] define an analysis method for distributed programs, on axes of composition and distribution. Peer to Peer systems can fall into many places on that coordinate plane, depending on the structure of their index, and the roles of different processes and nodes in the system. Some P2P systems have no servers, instead passing data back and forth to each other as needed with no central arbiter. One DVE by Limura, Hazeyama, and Kadobayashi [71] splits the virtual environment into a set of zones, and then uses a DHT to determine which machine is responsible for which zone. The DHT holds both an index of nodes responsible for each zone and an indicator of which roles (client, zone server, or both) a given node holds. The allocation and roles of nodes change over time, and the DHT is responsible for them; however they are also slow-changing enough to keep consistent in

the DHT. Data about what occurs within each zone, such as collisions, player state changes (alive → dead, etc.), and the like are maintained at each zone's server in a traditional client/server manner.

P2P systems provide some strong advantages:

1. *Scalability* — There are very few bottlenecks in a P2P system. They can grow without being limited to a single (or small group) of systems' capabilities. Current limits only seem to relate to the ratio of "super-nodes" to normal nodes. Super-nodes act as nexus points within the P2P network, doing much of the aggregation and indexing work. However, the assignment of super-node status is dynamic and can adapt to network needs.

2. *Fault Tolerance* — Going along with the fact that there is no central bottleneck in the network, there is also no single point of failure. Nodes can disconnect randomly and the overall network is generally unaffected. In fact, many P2P networks have average node connect times of a single hour, making the entire network quite dynamic.

   Similarly, faults related to scale are handled well. For example, a congested network link is unlikely to affect much more than a few peers, versus the Client / Server case where it may block the central link to the server. Similarly load-based denial of service problems are much harder to form in a P2P network.

3. *Global View* — Typically there is only one global P2P network of each type. These networks connect everyone to a single view of the system, which provides a distinctly different experience from hopping between servers.

P2P systems may thusly be useful for very large–scale, long–running systems with potentially high node turnover. For example, a DVE where an average user session is an hour.

However, the costs are definitely worth mention: there isn't a single, authoritative host, and change propagation can be slow. Without an authoritative host, conflict resolution can be difficult: a real issue with resolving inconsistencies in a DVE simulation. A single authoritative host quickly becomes a bottleneck and single point of failure for the system. Similarly, attempts to dynamically select authoritative hosts can have legitimacy and trust issues.

Similarly, changes in state can be difficult. To avoid long latencies in actions that change states, a P2P system must make a trade-off between acceptable latencies with caching or longer latencies with fewer cache-induced consistency problems. Due to the authority issues discussed above, it's also hard to determine which value of a piece of data is the correct one. Even if the authority of a data change is well–established, the delay in getting it to the entire network can be significant. Global cache invalidation isn't in any of the systems we've covered.

Consistency is difficult in P2P systems: there's no central time-base, no singular view of the system, and no latency-inexpensive mechanism to resolve disputes. The shared view of the virtual environment's state can lead to fairness and correctness problems. Similarly, integrity can be problematic, as some peers may simulate a region inappropriately for the benefit of a subset of users.

While consistency, fairness, and integrity are major challenges for P2P DVE systems, there is promising research in the area.

Examples in Research

The MOPAR [72] system uses a DHT, but only for structural zone server information. The rest of the DVE runs on a point to point peer system. The DHT simply takes care of letting every client know who else they're involved with, according to yet another hexagonal spatial cell system.

A full–fledged P2P system, the ASCEND [73] system partitions the virtual environment into a Voronoi diagram. A Voronoi diagram splits the region into connected polygons, each who's interior is the area closest to a given contained point. Each point is a P2P server, and the Voronoi region is the area the individual server covers. Each server only needs to maintain connections with bordering Voronoi regions. This mechanism provides a region–based cellular system that's proximity–optimal for any given server configuration.

GauthierDickey, Zappala, and Lo present [74] a P2P system with fully integrated authentication, communication, storage, and remote computation facilities.

DVEs as Distributed Applications

Some challenges in building and scaling DVE systems may be shared by any distributed application that has support substantial load. We will quickly review some potentially useful research results in the larger space of distributed applications.

Lantz, Nowicki, and Theimer [75] analyzed the factors in the performance of their distributed graphics application. For their system, they found the key factors being (in order):

1. Workstation Speed

2. Remote Host Speed

3. Level of Communications

4. Network Transport Protocol

5. Bandwidth of the Network

Aries et al. [76] provides discussion on designing internal systems designed for tens of thousands of users, focusing on the requirements elicitation, planning, and modeling phases of assembling a large scale distributed system. Vetter [77] provide a mechanism for near–real–time profiling of communications in distributed applications. Bagrodia et al. [78] discuss the COMPASS system, which provides event simulation and measurement for distributed applications, suitable even for testing out unimplemented systems.

Muhlhauser [79] describe the DESIGN environment. DESIGN is a full development system for distributed applications, with it's own programming language, an automatic model builder, and a distributed simulation engine. The language ("DC") extends the "C" language with special constructs for distributed applications and administration. The model builder parses DC code and feeds its output directly into the distributed simulation kernel.

The OSI Management Framework Hong et al. [80] provides software instrumentation, network protocols, and management tools for managing the deployment of large-scale distributed applications.

Middleware is a collection of library code and programs to support a program's primary operations. Okanda and Blair [81] describe OpenPING, which provides middleware and

services for common DVE components, such as replication and interest management. Lee, Lim, and Han [82] describe the ATLAS [82] system, which provides similar functionality.

Key DVE Systems: MASSIVEs v1,2,3

Probably the most educational system we can use to learn about the evolution of DVE communication is MASSIVE, versions 1, 2, and 3. MASSIVE [83] supported multiple users with various types of terminals in multiple locations, and aimed to support tens to hundreds of users working together in normal cooperative activities. It even supported multimedia sharing, like video and audio communication. Unfortunately, Greenhalgh, Purbrick, and Snowdon [84] found that it only took about 20 users before saturating a 10 megabit Ethernet link. It used $O(2^N)$ unicast flows to keep everyone synchronized, which quickly loaded the line. The importance of a good topology became obvious.

MASSIVE-2 [85] used a hierarchy of multicast groups to reduce the load on bottleneck network connections. While it didn't get more than 20 users on the system simultaneously, the limitation was now due to increased CPU use than network load. This was after both network and processing capabilities had been upgraded over the systems supporting the prior version.

MASSIVE-3 [84] went back to a unicast system, making its own logical multicast system atop of standard TCP. It used a fully distributed database system, with each change to the environment becoming a *change event* that was propagated over the logical multicast system. It also added a full hierarchical system for its scene-graph, which should help the system's CPU load. Unfortunately, no measurements were given for how many users they

were able to fit into one VE at the same time.

Key Standard: Distributed Interactive Simulation (DIS)

Fullford [8] describe the IEEE DIS *IEEE standard for distributed interactive simulation - application protocols (IEEE 1278.1-1995)* [86] protocol standard, which synchronizes using Dead-Reckoning vectors. DIS originates with SIMNET, described by Miller and Thorpe [87], and is getting superseded by the High Level Architecture (HLA), described by Dahmann [88]. Cavitt, Overstreet, and Maly [89] describe the PerfMETRICS system that actively reads, reduces, and analyzes DIS network events to help determine system performance. In their use, it helped planners understand the costs associated with given DIS system configurations, resulting in better simulations during DIS–based exercises.

Aggarwal et al. [90] considers how well the DIS mechanism works. It finds that even in low latency networks, dead reckoning can be fairly inaccurate. It suggests we synchronize clocks between client and server to more accurately represent movements in time–space.

Other Systems

In a different path, Chenney and Forsyth [91] use visibility information to reduce physics updates in the simulation, to reduce computational load on the host processor.

The SCORE system [92] uses a spatial cell system to break up the virtual environment, then uses the cells to manage the Area of Interest for each user. The cells form the basis of allocation for multicast addresses.

Instead of a regular shape for cells, Steed and Abou-Haidar [93] explore using quad–trees and k-d trees to dynamically partition a region based on data densities. The system would take regions whose sizes or densities hit above a preset threshold and split them into sets of smaller regions. Clients could then be sure that each region would have a maximal number of objects to it, and would have some dynamic control to which regions they subscribed to.

Network Quality of Service

A QoS (Quality of Service) [29] flow has a soft or hard guarantee about its latency, bandwidth, and jitter.

Through RFC 2211 [27], one may for *controlled load* service, which acts as a "best effort" link over a lightly loaded link. This can reduce packet loss and jitter significantly, leading to a preferable online experience. Through RFC 2212 [28], one may ask for *guaranteed service*, which gives strict bounds on delay and guarantees on bandwidth.

Greenhalgh, Benford, and Reynard [94] use a feedback loop of AoI information to determine the best clusters of data flows. Using these clusters, they continuously (re)allocate QoS streams to get good performance for their DVE. Wu and Chuang [95] used adaptive network QoS monitoring to adjust the coding rate of transferred MPEG media to maintain a given quality level over the network.

The network link is not the only Quality of Service domain of concern. Vogel et al. [96] discuss QoS concerns in streaming media, network protocols, data link technologies (ATM, FDDI, etc), application layer technologies, operating system schedulers, file servers, and

database systems. They continue on to discuss QoS negotiation protocols throughout these domains.

Unfortunately, getting access to QoS services from a network provider is a different issue. Jain, Sadeghi, and Knightly [97] describe mechanisms for building QoS facilities on cellular networks.

Of course, the QoS of a flow depends on the set of flows going over the link — meaning we can adapt our own flow(s) as needed. Campbell and Coulson [98] describe adapting video streams to QoS. Busse et al. [99] model QoS over UMTS (an enhanced version of GPRS, packet data for GSM cellular telephones), relating the bounds of latency to the bandwidth availability on the network.

Krishnamurthy et al. [100] discuss the use of a QoS specification (Quality Objects, or QuO for short) system atop their CORBA Object Request Broker, TAO. Using a standard API, they build a middleware–based distributed application with QoS capabilities. They discuss one system in particular, a distribution and playback system for video captured from an Unmanned Air Vehicle (UAV). Similarly, Duzan et al. describe using a combination of CORBA's QuO and Aspect–Oriented Programming to separate concerns when writing distributed applications. The combination allowed them to build systems that were very adaptable to their surrounding environment, such as dynamically deciding to compress data transfers when there was sufficient processor time, but network conditions were getting tight.

Jehaes et al. [101] studied the common last–mile technologies available. Focusing on modems, ADSL, and cable modems, they Pinged a host across one of these last–mile network links and used Ethereal for accurate measurement. While they found good data suggest-

ing that it's possible to properly segregate game traffic from other on the last–mile link for reasonable QoS, we're mostly interested in their actual measurements. They found that a modem connection has a flow rate of 49 KB/s, latency of 160.2 ms, and jitter at 97 ms. ADSL was better at 145 KB/s, 8.1 ms latency with 2 ms jitter. The cable modem exchanged bandwidth for latency and jitter: 1048 KB/s, 18 ms latency, 123 ms jitter.

A.3 Graphical Rendering

One of the largest responsibilities and defining factors of a DVE is its graphical display. The display is often a two–dimensional projection of the three–dimensional virtual environment. The general procedure for this rendering is a set of mathematical transformations between the geometric objects of the virtual world's objects and a two–dimensional viewport, followed by a evaluation process to determine the appropriate color for every pixel shown for each object.

For rendering the projection, a DVE system needs several properties in its rendering system:

1. *Proper Ordering* — The Z-buffer on a modern graphics card will prevent objects from the wrong front-to-back ordering, but they only maintain a single result for each pixel. Semitransparent objects must be rendered in back-to-front order, so that their pixels may be properly composited with their background.

2. *Speed* — The rendering must occur at a high. It must not starve other parts of the DVE system, such as physical simulation or network synchronization.

3. *Stability* — The time taken for each frame's render should usually be about the same. Watson et al. [11] describe how jumps in render speed are more disturbing to a user than a slower frame rate.

Most notable is the first requirement. For an arbitrary direction, we need a distance-sorted list of objects to draw quickly.

We will discuss two systems for maintaining a quickly-renderable version of the DVE system's scene graph.

A.3.1 BSP Tree

The BSP tree can provide a good solution to the three requirements. The construction of the tree (executed offline, once, before rendering) does most of the polygon-ordering computation. The tree consists of a recursive decomposition of the area into convex subspaces, each partitioned by a polygon. Every object in the scene graph is in a leaf node of this tree, with its spacial orientation to every other object already specified through its relative location in the tree with respect to them.

At runtime, the tree is simply traversed in back to front order, as specified by the relationship of the view vector and the split polygon for the subspace. The ordering requirement is easily satisfied. The speed requirement is satisfied. As long as the tree is balanced, the stability requirement is satisfied.

The BSP tree, being so effective in this area, is a popular strategy for rendering systems. However, it does depend on there being good splitting planes for the world. For indoor

environments, the walls make good candidates for partitions. For outdoor environments, there may be no split planes and the rendering may have to take on different strategies altogether to meet the speed and stability requirements.

A.3.2 Heightmaps and Open Spaces

BSP trees don't have usable partitions for large, open spaces. The objects in the open spaces are often moving, preventing even dynamic split plane generation from being effective. Instead, the ground terrain can be represented as a simple two–dimensional array of heights (the z coordinate), with the array indices representing the x and y coordinates.

If transparency is used conservatively, each object can be depth-sorted by its central point. Level of Detail (LOD) optimizations can help minimize the rendering load. If the terrain's well balanced in complexity, and the dynamic objects are well–dispersed, then the rendering times stay stable.

A.4 Performance

Acceptable DVE performance is a complex topic. While our research focuses on the work of Wu et al. [1], there are many other relevant aspects and viewpoints.

Acceptable system response is a question of measured performance: how well does the system provide, in terms of update latency, frame-rate, etc? If we allow user performance as a basis for quantitative analysis, we can start looking at some existing work for DVEs. We'll spend

a little time in this area, and then quickly jump into the larger bodies of work: measuring the performance of the DVE and analyzing it. From there, we'll cover the last major hurdle: the network that connects the system together.

A.4.1 User Sensitivity to Performance

User performance depends completely on the task at hand, and we're not making any large assumptions about that in our research. But, patterns are likely to show up for similar tasks, and we can likely make inferences from them.

In "Location–Aware Gaming," where users were spread around a physical area, and their locations made relevant in the virtual environment, quality of data was varied to see its effect on users. Mansley et al. [102] showed that different levels of data fidelity, as well as latency, strongly affect the user's experience.

For First Person Shooter games, Quax et. al. [103] provide an analysis of the effects of latency (pure amount and variance) on user performance, as measured by their in-game-score in Unreal Tournament 2003. Using a router that simulated specific latencies (and σ_{latency}), they found that users had noticeable impairment when the network Round-Trip Time (RTT) surpassed 60 ms. Bhatti and Henderson [25] found that the number of kills per minute dropped from 1.456 to 0.6233 when lag was artificially introduced. Similarly, the number of times killed per minute jumped from 0.6042 to 1.430.

However, the sensitivity varies greatly across types of virtual environments. For a Real Time Strategy (RTS) game, several seconds can be acceptable. This type of system has

users controlling software agents that interact with each others' software agents. Sheldon, Girard, Borg, Claypool, and Agu found [104] that high latencies had little effect on game success in Warcraft III. As in many RTSs, the simulation engine does much of the work, with the fairly occasional user command. As the command rate is significantly lower, and the level of automatic action is much higher, the user can afford significant lags without much detriment.

To help with the effects of latency, Gutwin et al. [105] experiment with a visual feedback indicator of network lag. In their tests, users were significantly better at their assigned tasks when they had a dynamic indicator of what kind of lag they were experiencing at any given time.

A.4.2 Performance Measurement

Quax et al. [106] describe the ALVIC (Architecture for Large Scale Virtual Interactive Communities) system, which uses a simulation-based method that uses autonomous avatars to simulate users, and uses that simulation to estimate scalability. Unlike our user-study basis for simulator behavior, Quax et al. [106] used flocking behavior.

Pellegrino and Dovrolis [107] compared the performance of DVEs with three different network topologies: Client–Server, Peer to Peer, and their own slightly hybridized variant, the Peer to Peer with Central Arbiter. This last type is simply a Peer to Peer system with one additional system, the Arbiter, listening in on all state changes, with veto power on those that violate some game or consistency rules.

Outside of DVE systems research, there has been substantial research in measuring the performance of distributed systems.

For middleware–based distributed apps, we have a few performance testing methodologies available. Denaro, Emmerich, and Polini [108] suggest that one can get accurate depictions of a distributed application's performance early on by focusing on the COTS ("Common Off The Shelf") components and middleware being used. Considering that many DVE systems use stock software engines (e.g. Quake, Unreal, Torque), which also take analogous roles to middleware, databases, and other software components from their approach, the approach may be reusable for DVE systems.

Rolia and Vetland [109] discuss both direct measurement and statistical methods for determining the resource requirements of distributed applications. They find direct measurement to be effective, but they also find that getting good measurements can often be very difficult, even more so in heterogeneous environments. When not possible, the discussed statistical methods can be used — assuming that it's possible to get representative input that can drive a good sampling of the system's usage.

Qin et al. [110] hand-instrument code that sends measurements to a *performance data server*, which collects and analyzes the data, building a performance model. In their example, they instrumented and measured a DCE application to build a Layered Queue model.

Litoiu, Khafagy, Qin, Wan, and Rolia [111] use generated profiling information to automatically construct a performance model of an application. The profiling information is inserted by the compiler, which records data during application execution. An analyzer is provided to allow traversal of these records. More interestingly, a model builder is included

which automatically generates the performance model. The developer is, of course and as always, responsible for providing representative inputs to the application to make it behave in ways similar to a full deployment. [111] provides examples using DCE, CGI/HTTP, and Java/CORBA.

Similarly Dumas and Gardarin [112] constructed a workbench tool to help determine the performance of distributed database applications. They built a combination simulation tool, SQL query cost analyzer, and DBMS optimizer to help software engineers design a well–performing distributed database application. The hope is to prevent investment in faulty designs that perform poorly.

Berman et al. [113] describe the AppLeS system, which uses dynamic load monitoring to determine how to deploy a distributed app across a set of systems. Using a subsystem called the Network Weather Service, it can estimate the current costs of network I/O. Combined with the system-level resource load monitoring, it provides a mechanism for application developers to construct their own models and algorithms for determining the optimal deployment configuration of their system.

Vetter and McCracken [114] describe using MPI's built–in profiling layer to get raw performance data. Using the raw numbers, they use rank correlation to filter out the relevant algorithm's normal fluctuations and find the least scaling components of their applications.

Helsinger, Lazarus, Wright, and Zinky [115] discuss a system for monitoring the performance of large distributed agent systems (specifically, COUGAAR). The challenges of gathering, reducing, aggregating, and analyzing the data from such a class of systems is discussed in detail.

A.4.3 Performance Analysis

To help understand the nuances and pitfalls of performance analysis in distributed applications, Rolia and Lin [116] cover common and fundamental mistakes many systems make. They claim one source of common problems is mismatching (or simply ignoring the effects of) *process periods*, the time intervals where raw data is aggregated during large data acquisition runs. Usually when a large, distributed application is getting profiled, the captured data set gets large fast. To compensate, many capture systems will aggregate time periods together into Rolia and Lin's process periods. A *global period* needs determination, which can safely be used for analysis without the consistency issues with process periods. Rolia and Lin discuss the determination method for global periods, and ways to understand and/or reduce the error involved in the measurements.

One popular metric for expressing and comparing the efficiencies of distributed apps is *isoefficiency*. According to Grama, Gupta, and Kumar [117], this metric "relates problem size to the number of processors required to maintain a system's efficiency, and it lets us determine scalability with respect to the number of processors, their speed, and the communication bandwidth of the interconnection network." Fundamentally, it represents how the algorithm speeds up as additional processors are added.

A.4.4 Network Performance

The Cooperative Association for Internet Data Analysis [118] (CAIDA) provides extensive tools, documentation, and data on Internet performance.

Sarvotham, Riedi, and Baraniuk [119] did an analysis in 2001 that puts most of the Internet's traffic as dominated by bursty file transmissions on high–speed links (called α traffic), with the rest (β) fitting in the spots between bursts. Shannon, Moore, and claffy [120] show that the network is quite sensitive to packet size, usually on UDP. Chakraborty et al. [121] show that the overall traffic pattern for the Internet seems to be a fractal.

A.4.5 Flow Models

Färber [122] describe a stochastic model of Counterstrike [122], built from captured network traffic. In their case, the Extreme Value distribution modeled the traffic pretty well, although a shift Lognormal or Weibull distribution would also fit.

Abry, Roughan, and Veitch [123] provided a real–time method for estimating the network flow, using little memory or CPU. It assumes a wavelet–based model. Faerman, Su, Wolski, and Berman [124] presented Adaptive Regression Modeling (AdRM). It used real–time regression analysis of data acquired from the Network Weather Service [113] and its own test runs to estimate the transfer time of data files ranging from 64 KB to 16 MB or more.

A.4.6 Kernel Scheduling Effects

For traditional desktop operating systems such as Linux, Unix, Windows, and Mac OS, the kernel's CPU scheduler tries to make some fair partition of the processor's time for each. However, the user likely wants the DVE system process to get more CPU time than other applications (whose user interfaces are typically occluded by the full–screen DVE). A lot of

that comes from the fact that the kernel can't tell which processes are most important to the user.

Etsion, Feitelson, and Tsafrir [125] study different metrics for detecting what they call *Human Centered* processes — ones that the user's likely using interactively right now. They study text editors (Emacs and OpenOffice), video players (Xine and MPlayer), and a DVE (Quake). By analyzing X server communications, they find that they can determine which processes are likely to have the user's attention. Feeding this data back into the kernel scheduler, they find a significant improvement in the responsiveness of interactive and multimedia applications. Most importantly, they find the DVE gets over 60% more CPU time under heavy load with their system.

Appendix B

Comparing DVEs to Other Systems

Distributed Virtual Environments have a distinct consistency, latency, and quality profile when compared to other common types of distributed systems. We compare DVEs against three common types of distributed systems: a group video chat, a database server, and a collaborative text editor.

B.1 Trade-offs and Quality with Group Video Chat

A multi-user video chat application, such as Skype Group Video Calling, lets groups of people have a simultaneous video chat. Each participant's machine is equipped with a screen, video camera, microphone and speaker. Each user's screen shows video feeds from all users in the chat. Similarly, each user can speak at any time, with their spoken words played back by other participants in real-time. Video-chat systems often choose one speaker at a time, to avoid having to mix sound channels from all users, and avoid possible audio feedback

problems.

Such a chat system has a substantial amount of high-bandwidth, low-latency data to send to each user. Users must transmit their video stream to all others, and similarly has to receive video streams from everyone else. Audio may work similarly. Roughly, the total bandwidth requirements for a chat is quadratic to the number of users — as each user is added to the chat, every other user receives their video (sent unicast over the Internet) and transmits their own video to the new user.

The video and audio streams are transmitted unreliably (e.g. UDP), as new data is continuously sent, because it's better to accept packet drop over the additional latency of waiting for a re-transmit.

Data from each participant can be difficult to predict. If a user stays still, a key-frame can be maintained for a long period of time, with a low rate of changes transmitted. If they are animated during their conversation, the image needs to be retransmitted, through faster key-frames or equivalently-large changes, more rapidly. Also, users not speaking may transmit lower-fidelity video (and audio) streams of themselves, as it's less likely that others are focused on the speaker.

Unfortunately, predicting these high-, low-, or irrelevant-movement states is difficult with humans driving them. This can complicate predicting performance on some deployments: the quality of the users' experience may ultimately depend on how they behaved while chatting, how well the media streams stayed within their resource constraints, and how much users were paying attention to media that had reduced audio and video quality due to those limitations.

DVEs share some similarities with these types of programs. Each user has to be somewhat aware of every other user. The system is very latency sensitive, and implementations often use an unreliable protocol to accept loss for reduced latency. DVE systems also have a shared state that persists, at least for the session, within the virtual world. The world can include buildings, items, avatars, vehicles, and non-player characters.

Unlike video chat, consistency requirements for the virtual environment require that lost changes to the world are eventually recovered and can cause contention between users, requiring some level of arbitration.

The amount of load on the system — user activities that have to be synchronized and simulated — depends greatly on user behavior. Similarly, the required level of quality in simulation and synchronization the DVE provides depends on where users pay attention in the virtual environment. For example, a user needs more up-to-date information about objects nearby and in front of them than objects that are behind them and far away. Similarly, simulation can be less accurate for secondary animations (e.g, smoke particle) than for task-critical elements like the control surfaces of a flown aircraft.

B.2 Trade-offs and Quality with a Database Server

In contrast to a group video chat, a database server does not tend to have quadratic data or low latency requirements. Clients connect over reliable network sockets and execute queries, sometimes modifying the data-set, and wait to receive their results. Each query is executed in a transactional model, with the possibility that they directly affect each other's latency

and results. Queries typically result in data-set sizes unrelated to the number of other clients. Consistency is enforced through the data-sets' schemas, triggers, validation rules, and conventions held by queries.

Like a DVE, there is shared state between participants, and it must be consistent. Furthermore, clients read data from a subset of the entire shared state, and need not receive data outside of their scope (e.g. not within the scope of the query). DVEs have a weaker form of consistency requirements which can depend on the type of object involved and the relevance of changes to user priorities. For example, a DVE race simulator must keep the relative positions of the cars highly consistent with user actions and vehicle dynamics. All users must be presented the same logical view of the race's state. However, the same DVE need not be accurate about the exact positions or actions of the simulated audience in the bleachers around the track. Each user could see different versions of the audience without substantial loss to their experience in the system.

Additionally, the amount of load applied to a database system can be described as the aggregate amount of work required to execute the queries it is given. The set of clients executing those queries are often other software systems that have a limited set of possible queries to send. As the query sources and their types are often predictable, the exact load on a system can often be tractably calculated.

B.3 Trade-offs with Quality with Collaborative Text Editing

Collaborative real-time text editing has started to become available in many products, some living completely within a web browser. The word-processing element of Google Drive provides a word processor as a web-page, with formatted text, comments, images, and tables. Users can share a document with others and give them the ability to read it, add comments, and modify it. Live editing of the document occurs with one cursor per user, each in a different color, directly modifying the text.

Each edit to the document has to be visible to other users, and the final version has to represent the complete set of edits executed by each user. If users are also talking to each other, across a room, over the phone, or over a video chat, they must have timely visibility of each others' edits.

In comparison to a group video chat, this type of user collaboration happens at typing rate, which can peak at 18 Hz (212 words per minute, at five characters per word[1]). The data transferred — changes to the text — are idempotent streams of keystrokes and commands. Users consider them timely if they arrive within a few seconds. Unlike video data, the data must all arrive, in order, and a TCP sliding window is a reasonable means to deliver it. The document must be consistent with all edits applied to it.

That consistency requirement is more like that of a database server. However, a typical document is significantly smaller than a typical database, and global-order consistency of

[1]Guinness Book of World Records, 2005 Barbara Blackburn, Dvorak

edits isn't necessary. While users all must have a copy of the entire document, it may not be completely consistent with the full set, or global order, of all edits executed.

We may interpret each users' incoming edits as a stream of atomic elements, and the document as a combined view of all those edits. A given user may have more recent views of some streams than others. Combined, different users may see slightly different versions of the same document at the same time, with the differences constrained by the timeliness properties of each stream. This stands in contrast to the transactional consistency of a database data-set.

Some data in a DVE has a similar consistency property. The positions of user avatars in the virtual space can often appear slightly off from each others' point of view, and still provide a sufficiently similar user experience. User experience can continue without detriment as long as errors are corrected quickly and unjarringly. Dead-reckoning [86] extrapolates avatar positions at each user's machine from the last-received position and velocity. With a low error rate and magnitude, the errors can be relatively difficult to notice.

Other data in a DVE needs better consistency than user positions. Text messages between users is a common feature in DVEs, and clearly needs to be sent without errors, reordering, or loss. Modal data, such as the state of locks, doors, and assignment of leadership roles, should be consistent within each user's view of the virtual world. Each user should receive the entire scope of the state change (e.g., the prior user losing the leadership role, and the current user gaining it) as a single update. All relevant users should get the update at nearly the same time.

The visibility of the data set is different than a DVE. The entirety of a shared text document should be visible to each user at all times: they may scroll at any time to any other place in

the document and read any part of it to ensure its global consistency when complete.

However, a virtual world can take time to traverse, by foot, or vehicle. Users rarely have a view of the entire virtual world, and the world rarely has a global consistency requirement. The limitation of each user's views creates opportunities for relaxing consistency requirements, because: an object can be in two places at the same time, as long as no user can see both places. The limitations also reduce synchronization requirements: users need only receive updates about objects within their current view. In some cases it is preferable to restrict the updates to that view, to prevent cheating users from seeing through walls or behind them, via hacks to their graphics card driver for texture transparency or field of view.

Appendix C

The Portable Performance Tool

DVE Scalability Engineering (DSE) has instrumentation needs that are unsatisfied by current tools. DSE's factor identification method requires better data granularity than the stochastic moments available from current performance tools. The method also requires the ability to capture secondary data around the measured values. The secondary data includes input values to the measured code, or arbitrary data collected to help analyze code.

Appropriate instrumentation for DSE should also avoid competing with a DVE system's software for computer resources. As instrumentation rarely requires a network link, the computer resources under contention are primarily the CPU, then memory.

We have constructed a new performance tool to capture ad-hoc instrumentation data, to let us analyze all information it captures offline, and to measure without competing with the DVE system software for computer resources. We call the performance tool `ppt`, the Portable Performance Tool.

`ppt` is a tool for custom instrumentation. The instrumentation data is defined, captured, transferred, and stored in blocks called *frames*. A frame is a user-defined group of data members to capture together. Each data member could be a resolution timestamp (nanoseconds since the `time_t` epoch), a native integer, or floating-point value in single or double precision. Members in each frame are automatically reordered to minimize padding overhead, and a definition for C is generated.

Frames are transferred over channels called *buffers*. Multiple types of frames can be transferred over the same buffer.

C.1 Describing Frames and Buffers

The input for `ppt` is a buffer specification. That specification names the buffer and defines the frames that can transfer over it. Listing C.1 shows a small buffer that carries two frames. The buffer is called `buf` and the frames are named `first` and `second`. The buffer specification starts with an `emit` statement, followed by a `buffer` statement. The former has one required argument, whose only current value is "C". The latter (buffer) statement has three arguments: a buffer name, a transfer rate (in integer Hertz), and a buffer size (described in Section C.4).

`ppt` uses the buffer name as an identifier prefix for generated source code. It uses the transfer rate and buffer size as default values in its transfer mechanism, which we will describe later.

The grammar for `frame` definitions are modeled after that of the C `struct`. They are substantially simpler: no unions, no bit fields, and no instance declarations. That last

simplification removes the need for a semicolon after the closing curly brace.

Example Buffer and Frame Description

```
// Options section
emit C // required, and ``C'' is currently the only allowed value

// Name the buffer (``buf''), specify default rate and buffer size
buffer buf default default

// First frame declaration
frame first {
  double total;
  time  start, end;
} // no semicolon needed.

// Second frame declaration
frame second {
  int num_iters;
  time start, end;
}
```

C.2 Generated Code

Given a buffer specification, `ppt` generates several files. `ppt` generates a header (*`buffername`*`.h`) and source (*`buffername`*`.c`) file for inclusion into the program to be measured. The header

includes definitions of C macros to save values to members in frames, and a declaration of a function per type of frame. That function writes a frame to the buffer. The source implements the buffer-write function. We describe its mechanism in Section C.4.

`ppt` also generates *reader* and *converter* programs. The reader participates in the transfer of instrumentation out of the observed process (i.e., the running DVE system software) to a disk file. The converter converts that binary-format disk file to a text format easily that is readable by GNU R, Microsoft Excel, or a hand-written analysis script.

C.3 Data Collection

The `ppt`-generated header and source declare a global instance of each frame, with an obfuscated name to avoid naming collisions with the DVE system source code. The macros defined in the generated header fill a member of that global instance. Every member type has one macro is defined: `WRITE_`*buffername_member*`()`. This macro takes a single argument: the value to save to the respective member of that global instance. A `time` member has a second macro defined: `WRITE_`*buffername_member*`_NOW()`, which takes no arguments. This second macro saves the current time to the member.

At any time, the system-under-observation may invoke the frame-write function. This function copies the global instance of the frame to the buffer. If the same macro is called twice without a frame-write in between, the second value overwrites the first. The first value is never saved to the buffer. The `_NOW()` macros overwrite values saved by their unary-argument colleagues, and visa-versa. Similarly, if there are not calls to any macros for a

frame member between two calls to the frame-write function, then both frame copies on the buffer will have the same value for that member.

DVE system software can write a frame several times to the buffer, with most members held constant. This ability can be advantageous in some situations. For example, a function may have a loop. Some members of a frame can identify the parameters of the function, and some members of the frame can contain data for a specific loop iteration. The function can fill the function-relevant members in the frame once, and fill the iteration-relevant members each time through the loop. The function writes the frame to the buffer on every loop iteration. The data saved to the buffer will have one frame per loop iteration. Each frame will contain both the function-relevant members and iteration-relative members.

C.4 Transfer Mechanism

The buffer is an array of frames in shared memory. Frames in a buffer are padded to have the same size. `ppt` then adds two members for a sequence number to each frame. `ppt` defines one of those sequence numbers as the first member of each frame, and another as the last (after the length-equalizing padding). The generated code for each buffer defines a global sequence number variable. When a frame-write function runs, it increments the sequence number and saves it to the frame as it writes it to the buffer. The array index into the buffer for that frame is the sequence number modulo the array size.

The buffer may not exist when the frame-write function is called. The frame-write function will immediately return without save the frame.

C.4.1 Attaching a Reader

`ppt` program creates the buffer as a shared memory segment as the first step of the *attachment* process. The attachment process is a protocol that `ppt` follows to safely start observing frames from a program that has linked in its generated code. That program would normally be a DVE system's software. That linked-in code participates in the attachment process. `ppt` creates the shared memory buffer and then uses `ptrace(2)`, a debugging system call, to write the shared memory handle to a global variable defined in the linked-in code. The next time a frame-write function defined for that buffer is called, it will check that global variable and attach that shared memory buffer to the address space of the observed process. That running call of the frame-write function, along with all frame-write functions after that, will write to the shared-memory buffer.

Similarly, the *detach* procedure involves `ppt` using `ptrace(2)` to reset the global variable holding shared memory handle back to zero. The next call to a frame-write function will notice the reset and detach the shared-memory buffer from the observed process's address space. Further calls to a frame-write function will notice the lack of a buffer and immediately return without saving their frames.

C.4.2 Reading Frames from a Buffer

After attachment, `ppt` invokes the *reader* process that it generated from the buffer specification. The reader attaches the shared memory buffer to its own address space and maintains an internal *index* into the buffer's array. That *index* starts at zero.

The reader begins a scan-sleep loop. During a scan, the reader compares the sequence number of the frame at its *index* in the buffer, against the sequence number it last saw at that *index*. If it has changed, the reader increments the *index* (modulo the buffer's size) forward until either the sequence numbers are no longer new or the index comes back to where it started. After this increment-scan, the range of new elements are given to one or two (depending on whether the end of the buffer was reached) calls to `fwrite(3)` to save to a disk file. Then the reader sleeps.

The reader estimates, from a buffer default or observed rate, the duration needed for the observed process to fill half the buffer with new frames. The reader sleeps for this long. When it wakes up, it scans the buffer again.

This process continues until the user kills the process with `Ctrl-C`. Then the reader will detach and flush the disk file before shutting down.

C.4.3 Performance

The overhead for `ppt`'s generated code has two parts: the part that writes the frames to the buffer, and the part that reads frames from the buffer and saves them to a disk file. The part that writes the frames to the buffer is little more than a few word-copy instructions, an increment (for the sequence number), and a few memory barriers to ensure that the writes of the sequence number bracket the frame data.

The reader attempts to sleep as long as possible. The reader allows a (typically small, $< 2\%$) detectable data loss to minimize performance impact on the observed process. The data loss

occurs when the reader sleeps too long, and the observed process has written more than a buffer's worth of frames. In that scenario, some frames will get overwritten before the reader can save them to disk.

While we would prefer no data loss, our buffer-based transfer mechanism has several advantages. First, if the DVE system's software is busy on the CPU, the reader can avoid competing with it for CPU time until the DVE system's software is blocked or idle. Second, we can tune the likelihood of data loss via control of the buffer's size. The frames are often quite small, on the order of a single cache line, and many frames fit in small buffers. We used a standard half-megabyte buffer size for all of our research and found the worst data loss to be less than two percent.

The transfer mechanism also lets us detect the lost data through the sequence numbers. The sequences will have "holes" where we have lost data. If we find patterns in the holes, we know that either the frame write-rate jumped suddenly (causing the reader's old write-rate estimate to be low enough to have it sleep too long) or that the reader didn't get enough CPU time to copy out the buffer. In either case, the information is potentially useful and we can gather better data just by enlarging the buffer.

C.5 Usage

Looking back at the buffer specification in Listing C.1, we first need to generate the appropriate source files. To generate the code, we have `ppt generate` the files in its internal storage. Then we `retrieve` the generated files back.

```
$ ppt generate example.spec

$ ppt retrieve example.spec

Copying example.h

Copying example.c

Copying example_listen.c

Copying example_listen.lli

Copying example_convert.c
```

Generated File	Description
`example.h`	Macros and Declarations
`example.c`	Support and Transfer Routines
`example_listen.c`	Reader program, in C
`example_listen.lli`	Reader Program, in LLVM Bitcode
`example_convert.c`	File Converter

Figure C.1: `ppt` Generated Sources

The first two generated files here, `example.h` and `example.c`, are for linking directly into the observed DVE system's software. The software should invoke the macros and frame-write functions declared and defined in these files.

The next two files, `example_listen.c` and `example_listen.lli`, are two versions of the *reader* program. The former is a C program, and the latter in LLVM bitcode. Either can be compiled to an executable and used.

Finally, `example_convert.c` converts the binary file saved by the reader program into a set of text files. There is one text file per frame type in the buffer. Each text file has one row per frame, and a tab character between each member. The first line of each file is a header row of member names.

C.6 Inserting Instrumentation

The developer must compile and link the generated source and header into the DVE system's software. The developer should also invoke the generated instrumentation macros and buffer-write functions. A simple example using the instrumentation is in Listing C.1.

Listing C.1: Example Use

```
#include <vector>
#include "example.h" // the generated header

// Returns the average, as a truncated int, of the array 'values'.
int first(double *values, int nvalues) {
  WRITE_BUF_FIRST_START_NOW();
  double ttl = 0;
  int nv = nvalues;
  while(nv--)
    ttl += *values++;
  WRITE_BUF_FIRST_TOTAL(ttl);
  WRITE_BUF_FIRST_END_NOW();
  ppt_write_first_frame();
  return (int) (ttl / nvalues);
}

// Run 'iters' times through a linear congruence random number
// generator, then return the 'iters'-th result.  For the record,
// this is a terrible RNG, but at least it's reentrant.
int second(int iters) {
```

```
  WRITE_BUF_SECOND_START_NOW();
  int nv = iters;  // iters is also the seed.
  WRITE_BUF_SECOND_NUM_ITERS(nv);
  while(iters--)
    nv = 1103515245 * nv + 12345;
  WRITE_BUF_SECOND_END_NOW();
  ppt_write_second_frame();
  return nv;
}

int main() {
  std::vector<double> values;
  values.reserve(64);
  for (int i = 0; i < 512; ++i) {
    double v = rand();
    if (i < 64) {
      values[i] = v;
      printf("%d ", second(first(values.begin(), i)));
    } else {
      values[i % 64] = v;
      printf("%d ", second(first(values.begin(), 64)));
    }
  }
}
```

In the example code, we have an averaging routine called `first`, and a linear congruence random-number-generator called `second`. A `main` routine takes input from the stan-

dard `rand()` routine to feed calls to `second(first([random numbers]))`. The calls `WRITE_BUF_*_START_NOW()` and `WRITE_BUF_*_END_NOW()` for both `*=FIRST` and `*=SECOND` save the amount of time spent in each routine. The calls `WRITE_BUF_FIRST_TOTAL` and `WRITE_BUF_SECOND_NUM_ITERS` save workload factors for both routines.

C.6.1 Collecting Data

To collect data with `ppt`, we *attach* to the listener. Currently `ppt` assumes that the generated listener is compiled and in the `PATH`. Assuming that a copy of our instrumented program is running with pid 5477, the following command attaches and begins reading data:

```
$ ppt attach -p 5477 example.spec
```

By default, the captured data is placed in a binary file named`out.buf`, and capturing continues until the user kills the reader with Control-C. `ppt` always uses the actual instrumentations specification as an identifier. To convert the data, the converter is called:

```
$ example_convert out.buf first-set
```

This will generate two files: `first-set.first.csv` and `first-set.second.csv`, one per frame. The `first-set.first.csv` would begin like this:

```
ppt_seqno   total   start           end
501         511.22  1294621640      1294621640
...
```

Values with invalid or non-matching `ppt_seqno` values are automatically omitted. The file can be directly read by quite a few numeric analysis and statistics tools, as well as the C++ `iostream` library and classic tools like `awk`.

C.7 Conclusions and Future Work

`ppt` provides capture and storage of arbitrary data in raw form. `ppt`'s transfer mechanism has low CPU overhead and the reader actively observes the buffer usage to dynamically minimize CPU overhead further. `ppt` gets its advantages by allowing observable data loss, in lieu of strong synchronization between the observed DVE system software's process and the reading process. Instead of calculating statistical moments online, `ppt` simply captures data for offline analysis. Using it in combination with GNU R, we have found `ppt` to be a useful tool for factor identification and general performance analysis.

In the future we intend to implement a multi-threaded interface for `ppt`. This will allow multiple threads in the observed system's process write to the same buffer safely and simultaneously. Additionally, we would like to make `ppt` emit code for more languages than just C.

References

[1] Wanmin Wu et al. "Quality of experience in distributed interactive multimedia environments: toward a theoretical framework." In: *Proceedings of the 17th ACM international conference on Multimedia*. MM '09. Beijing, China: ACM, 2009, pp. 481–490.

[2] Michael Weilbacher. "Dedicated Servers in Gears of War 3: Scaling to Millions of Players." In: Game Developers Conference 2012, 2012.

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

[4] Sandeep Singhal and Michael Zyda. *Networked virtual environments: design and implementation*. ACM Press/Addison-Wesley Publishing Co., 1999.

[5] R.M. Baños et al. "A virtual reality system for the treatment of stress-related disorders: A preliminary analysis of efficacy compared to a standard cognitive behavioral program." In: *International Journal of Human-Computer Studies* 69.9 (2011), pp. 602–613.

[6] Daniel Pittman and Chris GauthierDickey. "Characterizing Virtual Populations in Massively Multiplayer Online Role-Playing Games." In: *Advances in Multimedia Modeling*. Ed. by Susanne Boll et al. Vol. 5916. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 87–97.

[7] Samuli Pekkola et al. "Collaborative virtual environments in the year of the dragon." In: *CVE '00: Proceedings of the third international conference on Collaborative virtual environments*. San Francisco, California, United States: ACM Press, 2000, pp. 11–18.

[8] Deborah A. Fullford. "Distributed interactive simulation: its past, present, and future." In: *WSC '96: Proceedings of the 28th conference on Winter simulation*. Coronado, California, United States: ACM Press, 1996, pp. 179–185.

[9] K.U.R. Laghari and K. Connelly. "Toward total quality of experience: A QoE model in a communication ecosystem." In: *Communications Magazine, IEEE* 50.4 (Apr. 2012), pp. 58–65.

[10] K. Connelly. "On Developing a Technology Acceptance Model for Pervasive Computing." In:

[11] Benjamin Watson et al. "Effects of Variation in System Responsiveness on User Performance in Virtual Environments." In: *Human Factors: The Journal of the Human Factors and Ergonomics Society* 40.3 (1998), pp. 403–414. eprint: `http://hfs.sagepub.com/content/40/3/403.full.pdf+html`.

[12] Fred D Davis. *A technology acceptance model for empirically testing new end-user information systems : theory and results*. 1986.

[13] M.Y. Chuttur. "Overview of the Technology Acceptance Model: Origins, Developments and Future Directions." In: *Sprouts: Working Papers on Information Systems* 9.37 (2009).

[14] Mihaly Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. First Edition. Harper Perennial, Mar. 13, 1991.

[15] Renata M. Sheppard et al. "Advancing interactive collaborative mediums through tele-immersive dance (TED): a symbiotic creativity and design environment for art and computer science." In: *Proceedings of the 16th ACM international conference on Multimedia*. MM '08. Vancouver, British Columbia, Canada: ACM, 2008, pp. 579–588.

[16] Zhenyu Yang et al. "Collaborative dancing in tele-immersive environment." In: *Proceedings of the 14th annual ACM international conference on Multimedia*. MULTIMEDIA '06. Santa Barbara, CA, USA: ACM, 2006, pp. 723–726.

[17] Martijn J. Schuemie et al. "Research on Presence in Virtual Reality: A Survey." In: *Cyberpsychology and Behavior* 4.2 (2001), pp. 183–201.

[18] Fred D. Davis. "Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology." In: *MIS Quarterly* 13.3 (1989), pp. 319–340.

[19] John C. S. Lui, M. F. Chan, and Student Member. "An Efficient Partitioning Algorithm for Distributed Virtual Environment Systems." In: *IEEE Trans. Parallel and Distributed Systems* 13 (2002), p. 2002.

[20] P. Morillo et al. "Improving the performance of distributed virtual environment systems." In: *Parallel and Distributed Systems, IEEE Transactions on* 16.7 (July 2005), pp. 637–649.

[21] Ahmed Abdelkhalek, Angelos Bilas, and Andreas Moshovos. "Behavior and Performance of Interactive Multi-Player Game Servers." In: *Cluster Computing* 6.4 (Oct. 2003), pp. 355–366.

[22] Duong Ta et al. "A Framework for Performance Evaluation of Large-scale Interactive Distributed Virtual Environments." In: *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*. July 2010, pp. 2744–2751.

[23] B. Watson et al. "Evaluation of the effects of frame time variation on VR task performance." In: *Virtual Reality Annual International Symposium, 1997., IEEE 1997* (Mar. 1997), pp. 38–44.

[24] Peter Quax et al. "Objective and Subjective Evaluation of the Influence of Small Amounts of Delay and Jitter on a Recent First Person Shooter Game." In: *Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games*. Portland, Oregon: ACM, 2004, pp. 152–156.

[25] Tristan Henderson and Saleem Bhatti. "Networked games: a QoS-sensitive application for QoS-insensitive users?" In: *RIPQoS '03: Proceedings of the ACM SIGCOMM workshop on Revisiting IP QoS*. Karlsruhe, Germany: ACM Press, 2003, pp. 141–147.

[26] Bert N. Corwin and Robert L. Braddock. "Operational performance metrics in a distributed system. Part I.: Strategy." In: *SAC '92: Proceedings of the 1992 ACM/SI-*

GAPP symposium on Applied computing. Kansas City, Missouri, United States: ACM Press, 1992, pp. 867–872.

[27] J. Wroclawski. *Specification of the Controlled-Load Network Element Service*. RFC 2211 (Proposed Standard). Sept. 1997.

[28] S. Shenker, C. Partridge, and R. Guerin. *Specification of Guaranteed Quality of Service*. RFC 2212 (Proposed Standard). Sept. 1997.

[29] R. Braden et al. *Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification*. RFC 2205 (Proposed Standard). Updated by RFCs 2750, 3936. Sept. 1997.

[30] Maynard Johnson et al. *OProfile - A System Profiler for Linux*. `http://oprofile.sourceforg` Feb. 2012.

[31] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. "Dynamic instrumentation of production systems." In: *Proceedings of the annual conference on USENIX Annual Technical Conference*. ATEC '04. Boston, MA: USENIX Association, 2004, pp. 2–2.

[32] Free Software Foundation. *GNU Binutils*. `http://www.gnu.org/software/binutils/`. Feb. 2012.

[33] Bruce Powel Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley Professional, 1999.

[34] S. Balsamo, R. Mamprin, and M. Marzolla. "Performance evaluation of software architectures with queuing network models." In: *Proc. ESMc* 4 (2004).

[35] V. Cortellessa, A. Di Marco, and P. Inverardi. *Model-Based Software Performance Analysis*. Springer, 2011.

[36] D. Kartson et al. *Modelling with Generalized Stochastic Petri Nets*. 1st. John Wiley & Sons, Inc., 1994.

[37] Jiang Jie, Kuang Yang, and Shen Haihui. "The Application of AI for the Non Player Character in Computer Games." In: *Computational and Information Sciences (ICCIS), 2011 International Conference on*. Oct. 2011, pp. 1049–1050.

[38] Mat Buckland. "Programming Game AI by Example." In: Wordware, 2005. Chap. 9.

[39] Justin Heyes-Jones. *A* Algorithm Tutorial*. `http://www.heyes-jones.com/astar.html`. Feb. 2012.

[40] K.C. Finney. *Advanced 3D game programming all in one*. Thomson/Course Technology, 2005.

[41] F.D. Davis, R.P. Bagozzi, and P.R. Warshaw. "Extrinsic and intrinsic motivation to use computers in the workplace1." In: *Journal of applied social psychology* 22.14 (1992), pp. 1111–1132.

[42] D.A. Adams, R.R. Nelson, and P.A. Todd. "Perceived usefulness, ease of use, and usage of information technology: a replication." In: *MIS quarterly* (1992), pp. 227–247.

[43] GarageGames. *Torque Game Engine*. `http://www.garagegames.com/products/browse/tge/`.

[44] David P Luebke et al. *Level of detail for 3D graphics*. Morgan Kaufmann Pub, 2003.

[45] Nick Brett. *Asteroids*. `http://www-pnp.physics.ox.ac.uk/~brett/projects/asteroids.html`. July 2011.

[46] Ludwig Nussel et al. *ioquake3*. `http://ioquake3.org`. Mar. 2012.

[47] D. Luebke et al. *Level of Detail for 3D Graphics*. The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Elsevier Science, 2002.

[48] Sudhir Aggarwal et al. "Accuracy in dead-reckoning based distributed multi-player games." In: *SIGCOMM 2004 Workshops: Proceedings of ACM SIGCOMM 2004 workshops on NetGames '04*. Portland, Oregon, USA: ACM Press, 2004, pp. 161–165.

[49] Rahul Balani. *Energy Consumption Analysis for Bluetooth, WiFi and Cellular Networks*. 2007.

[50] John Bicket et al. "Architecture and evaluation of an unplanned 802.11b mesh network." In: *Proceedings of the 11th annual international conference on Mobile computing and networking*. MobiCom '05. Cologne, Germany: ACM, 2005, pp. 31–42.

[51] Ericsson. *The evolution of EDGE*. 2009.

[52] Havok. *Havok Physics*. `http://www.havok.com/content/view/17/30/`.

[53] Russell Smith. *Open Dynamics Engine*. `http://ode.org/`.

[54] Wikimedia Project. *Gran Turismo (video game)*. `http://en.wikipedia.org/wiki/Gran_Turismo_(game)`.

[55] David H. Eberly. *Game Physics*. Ed. by Tim Cox. Morgan Kaufmann, 2004.

[56] Philip Schneider and David H. Eberly. *Geometric Tools for Computer Graphics*. Morgan Kaufmann, 2002.

[57] Christer Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, 2005.

[58] Ageia. *PhysX*. `http://www.ageia.com/products/physx.html`.

[59] Inc. Sony Computer Entertainment. *Cell Broadband Processor*. `http://cell.scei.co.jp/index_e.html`.

[60] Tim Ferguson. *Quake II Network Protocol Specs*. `http://www.csse.monash.edu.au/~timf/bottim/q2net/q2network-0.03.html`.

[61] Mark Frohnmayer and Tim Gift. *The TRIBES Engine Networking Model*. `http://www.garagegames.com/articles/networking1/`.

[62] J. Postel. *Transmission Control Protocol*. RFC 793 (Standard). Updated by RFC 3168. Sept. 1, 1981.

[63] J. Postel. *User Datagram Protocol*. RFC 768 (Standard). Aug. 28, 1980.

[64] K. Ravindran and X. T. Lin. "Structural complexity and execution efficiency of distributed application protocols." In: *SIGCOMM '93: Conference proceedings on Communications architectures, protocols and applications*. San Francisco, California, United States: ACM Press, 1993, pp. 160–169.

[65] Y. W. Bernier. "Latency Compensating Methods in client/server in-game protocol design and optimization." In: *Proceedings of the 15th Game Developers Conference*. Mar. 2001.

[66] Daniel Bauer, Sean Rooney, and Paolo Scotton. "Network infrastructure for massively distributed games." In: *NETGAMES '02: Proceedings of the 1st workshop on Network and system support for games*. Bruanschweig, Germany: ACM Press, 2002, pp. 36–43.

[67] Carnegie Mellon Software Engineering Institute. *Client/Server Software Architectures - An Overview*. `http://www.sei.cmu.edu/str/descriptions/clientserver_body.html`.

[68] Sony Online Entertainment. *PlanetSide*. `http://planetside.station.sony.com/`.

[69] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. "A survey of peer-to-peer content distribution technologies." In: *ACM Comput. Surv.* 36.4 (2004), pp. 335–371.

[70] J. Rolia, V. Vetland, and G. Hills. "Ensuring responsiveness and scalability for distributed applications." In: *CASCON '95: Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research*. Toronto, Ontario, Canada: IBM Press, 1995, p. 55.

[71] Takuji Iimura, Hiroaki Hazeyama, and Youki Kadobayashi. "Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games." In: *SIGCOMM 2004 Workshops: Proceedings of ACM SIGCOMM 2004 workshops on NetGames '04*. Portland, Oregon, USA: ACM Press, 2004, pp. 116–120.

[72] Anthony (Peiqun) Yu and Son T. Vuong. "MOPAR: a mobile peer-to-peer overlay architecture for interest management of massively multiplayer online games." In: *NOSSDAV '05: Proceedings of the international workshop on Network and operating systems support for digital audio and video*. Stevenson, Washington, USA: ACM Press, 2005, pp. 99–104.

[73] Shun-Yun Hu and Guan-Ming Liao. "Scalable peer-to-peer networked virtual environment." In: *SIGCOMM 2004 Workshops: Proceedings of ACM SIGCOMM 2004*

workshops on NetGames '04. Portland, Oregon, USA: ACM Press, 2004, pp. 129–133.

[74] Chris Gauthier Dickey, Daniel Zappala, and Virginia Lo. "A fully distributed architecture for massively multiplayer online games." In: *SIGCOMM 2004 Workshops: Proceedings of ACM SIGCOMM 2004 workshops on NetGames '04*. Portland, Oregon, USA: ACM Press, 2004, pp. 171–171.

[75] Keith A. Lantz, William I. Nowicki, and Marvin M. Theimer. "Factors affecting the performance of distributed applications." In: *SIGCOMM '84: Proceedings of the ACM SIGCOMM symposium on Communications architectures and protocols*. Montreal, Quebec, Canada: ACM Press, 1984, pp. 116–123.

[76] James A. Aries et al. "Capacity and performance analysis of distributed enterprise systems." In: *Commun. ACM* 45.6 (2002), pp. 100–105.

[77] Jeffrey Vetter. "Dynamic statistical profiling of communication activity in distributed applications." In: *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. Marina Del Rey, California: ACM Press, 2002, pp. 240–250.

[78] Rajive Bagrodia et al. "Performance prediction of large parallel applications using parallel simulations." In: *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. Atlanta, Georgia, United States: ACM Press, 1999, pp. 151–162.

[79] Max Muhlhauser. "Using distributed simulation for distributed application development." In: *ANSS '88: Proceedings of the 21st annual symposium on Simulation*. Tampa, Florida, United States: IEEE Computer Society Press, 1988, pp. 189–206.

[80] James W. Hong et al. "Toward distributed applications management using the OSI management framework." In: *CASCON '94: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*. Toronto, Ontario, Canada: IBM Press, 1994, p. 30.

[81] Paul Okanda and Gordon Blair. "OpenPING: a reflective middleware for the construction of adaptive networked game applications." In: *SIGCOMM 2004 Workshops: Proceedings of ACM SIGCOMM 2004 workshops on NetGames '04*. Portland, Oregon, USA: ACM Press, 2004, pp. 111–115.

[82] Dongman Lee, Mingyu Lim, and Seunghyun Han. "ATLAS: a scalable network framework for distributed virtual environments." In: *CVE '02: Proceedings of the 4th international conference on Collaborative virtual environments*. Bonn, Germany: ACM Press, 2002, pp. 47–54.

[83] Chris Greenhalgh and Steven Benford. "MASSIVE: a collaborative virtual environment for teleconferencing." In: *ACM Trans. Comput.-Hum. Interact.* 2.3 (1995), pp. 239–261.

[84] Chris Greenhalgh, Jim Purbrick, and Dave Snowdon. "Inside MASSIVE-3: flexible support for data consistency and world structuring." In: *CVE '00: Proceedings of the third international conference on Collaborative virtual environments*. San Francisco, California, United States: ACM Press, 2000, pp. 119–127.

[85] Chris Greenhalgh. *Dynamic, embedded multicast groups in MASSIVE-2*. Computer Science Technical Report. 1996.

[86] *IEEE standard for distributed interactive simulation - application protocols (IEEE 1278.1-1995)*. 26 Mar 1996.

[87] D.C. Miller and J. A. Thorpe. "SIMNET: the advent of simulator networking." In: *Proceedings of the IEEE*. Vol. 83. 8. IEEE. 1995, pp. 1114–1123.

[88] J.S. Dahmann. "High Level Architecture for simulation." In: *Distributed Interactive Simulation and Real Time Applications, 1997., First International Workshop on*. 1997.

[89] David B. Cavitt, C. Michael Overstreet, and Kurt J. Maly. "A performance monitoring application for distributed interactive simulations (DIS)." In: *WSC '97: Proceedings of the 29th conference on Winter simulation*. Atlanta, Georgia, United States: ACM Press, 1997, pp. 421–428.

[90] Sudhir Aggarwal et al. "Accuracy in dead-reckoning based distributed multi-player games." In: *SIGCOMM 2004 Workshops: Proceedings of ACM SIGCOMM 2004 workshops on NetGames '04*. Portland, Oregon, USA: ACM Press, 2004, pp. 161–165.

[91] Stephen Chenney and David Forsyth. "View-dependent culling of dynamic systems in virtual environments." In: *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*. Providence, Rhode Island, United States: ACM Press, 1997, pp. 55–58.

[92] Emmanuel Léty, Thierry Turletti, and François Baccelli. "SCORE: a scalable communication protocol for large-scale virtual environments." In: *IEEE/ACM Trans. Netw.* 12.2 (2004), pp. 247–260.

[93] Anthony Steed and Roula Abou-Haidar. "Partitioning crowded virtual environments." In: *VRST '03: Proceedings of the ACM symposium on Virtual reality software and technology*. Osaka, Japan: ACM Press, 2003, pp. 7–14.

[94] Chris Greenhalgh, Steve Benford, and Gail Reynard. "A QoS architecture for collaborative virtual environments." In: *MULTIMEDIA '99: Proceedings of the seventh ACM international conference on Multimedia (Part 1)*. Orlando, Florida, United States: ACM Press, 1999, pp. 121–130.

[95] Hsiao-Kuang Wu and Pei-Hung Chuang. "Dynamic QoS allocation for multimedia ad hoc wireless networks." In: *Mob. Netw. Appl.* 6.4 (2001), pp. 377–384.

[96] Andreas Vogel et al. "Distributed multimedia applications and quality of service: a survey." In: *CASCON '94: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*. Toronto, Ontario, Canada: IBM Press, 1994, p. 71.

[97] Rahul Jain, Bahareh Sadeghi, and Edward W. Knightly. "Towards coarse-grained mobile QoS." In: *WOWMOM '99: Proceedings of the 2nd ACM international workshop on Wireless mobile multimedia*. Seattle, Washington, United States: ACM Press, 1999, pp. 109–116.

[98] Andrew Campbell and Geoff Coulson. "A QoS adaptive transport system: design, implementation and experience." In: *MULTIMEDIA '96: Proceedings of the fourth ACM international conference on Multimedia*. Boston, Massachusetts, United States: ACM Press, 1996, pp. 117–127.

[99] Marcel Busse et al. "Lightweight QoS-support for networked mobile gaming." In: *SIGCOMM 2004 Workshops: Proceedings of ACM SIGCOMM 2004 workshops on NetGames '04*. Portland, Oregon, USA: ACM Press, 2004, pp. 85–92.

[100] Yamuna Krishnamurthy et al. "Integration of QoS-Enabled Distributed Object Computing Middleware for Developing Next-Generation Distributed Application." In: *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*. Snow Bird, Utah, United States: ACM Press, 2001, pp. 230–237.

[101] Tom Jehaes et al. "Access network delay in networked games." In: *NETGAMES '03: Proceedings of the 2nd workshop on Network and system support for games*. Redwood City, California: ACM Press, 2003, pp. 63–71.

[102] Kieran Mansley et al. "Feedback, latency, accuracy: exploring tradeoffs in location-aware gaming." In: *SIGCOMM 2004 Workshops: Proceedings of ACM SIGCOMM 2004 workshops on NetGames '04*. Portland, Oregon, USA: ACM Press, 2004, pp. 93–97.

[103] Peter Quax et al. "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*. Portland, Oregon, USA: ACM Press, 2004, pp. 152–156.

[104] Nathan Sheldon et al. "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*. Redwood City, California: ACM Press, 2003, pp. 3–14.

[105] Carl Gutwin et al. "Revealing delay in collaborative environments." In: *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*. Vienna, Austria: ACM Press, 2004, pp. 503–510.

[106] Peter Quax et al. "Using autonomous avatars to simulate a large-scale multi-user networked virtual environment." In: *VRCAI '04: Proceedings of the 2004 ACM SIGGRAPH international conference on Virtual Reality continuum and its applications in industry*. Singapore: ACM Press, 2004, pp. 88–94.

[107] Joseph D. Pellegrino and Constantinos Dovrolis. "Bandwidth requirement and state consistency in three multiplayer game architectures." In: *NETGAMES '03: Proceedings of the 2nd workshop on Network and system support for games*. Redwood City, California: ACM Press, 2003, pp. 52–59.

[108] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. "Early performance testing of distributed software applications." In: *WOSP '04: Proceedings of the 4th international workshop on Software and performance*. Redwood Shores, California: ACM Press, 2004, pp. 94–103.

[109] Jerome Rolia and Vidar Vetland. "Parameter estimation for performance models of distributed application systems." In: *CASCON '95: Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research*. Toronto, Ontario, Canada: IBM Press, 1995, p. 54.

[110] M. Qin et al. "Automatic generation of performance models for distributed application systems." In: *CASCON '96: Proceedings of the 1996 conference of the Centre for*

Advanced Studies on Collaborative research. Toronto, Ontario, Canada: IBM Press, 1996, p. 33.

[111] M. Litoiu et al. "A performance engineering tool and method for distributing applications." In: *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*. Toronto, Ontario, Canada: IBM Press, 1997, p. 14.

[112] Sophie Dumas and George Gardarin. "A workbench for predicting the performances of distributed object architectures." In: *WSC '98: Proceedings of the 30th conference on Winter simulation*. Washington, D.C., United States: IEEE Computer Society Press, 1998, pp. 515–522.

[113] Francine D. Berman et al. "Application-level scheduling on distributed heterogeneous networks." In: *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*. Pittsburgh, Pennsylvania, United States: IEEE Computer Society, 1996, p. 39.

[114] Jeffrey S. Vetter and Michael O. McCracken. "Statistical scalability analysis of communication operations in distributed applications." In: *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*. Snowbird, Utah, United States: ACM Press, 2001, pp. 123–132.

[115] Aaron Helsinger et al. "Tools and techniques for performance measurement of large distributed multiagent systems." In: *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*. Melbourne, Australia: ACM Press, 2003, pp. 843–850.

[116] Jerome Rolia and Bo Lin. "Consistency issues in distributed application performance metrics." In: *CASCON '94: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*. Toronto, Ontario, Canada: IBM Press, 1994, p. 62.

[117] A.Y. Grama, A. Gupta, and V Kumar. "Isoefficiency: Measuring the scalability of parallel algorithms and architectures." In: *Parallel & Distributed Technology: Systems & Applications* 1.3 (1993), pp. 12–21.

[118] The Cooperative Associatioon for Internet Data Analysis. *The CAIDA Web Site: The Cooperative Association for Internet Data Analysis*. `http://www.caida.org`.

[119] Shriram Sarvotham, Rudolf Riedi, and Richard Baraniuk. "Connection-level analysis and modeling of network traffic." In: *IMW '01: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*. San Francisco, California, USA: ACM Press, 2001, pp. 99–103.

[120] Colleen Shannon, David Moore, and k claffy. "Characteristics of fragmented IP traffic on internet links." In: *IMW '01: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*. San Francisco, California, USA: ACM Press, 2001, pp. 83–97.

[121] D. Chakraborty et al. "Self-similar and fractal nature of internet traffic." In: *Int. J. Netw. Manag.* 14.2 (2004), pp. 119–129.

[122] Johannes Färber. "Network game traffic modelling." In: *NETGAMES '02: Proceedings of the 1st workshop on Network and system support for games*. Bruanschweig, Germany: ACM Press, 2002, pp. 53–57.

[123] Matthew Roughan, Darryl Veitch, and Patrice Abry. "Real-time estimation of the parameters of long-range dependence." In: *IEEE/ACM Trans. Netw.* 8.4 (2000), pp. 467–478.

[124] Marcio Faerman et al. "Adaptive performance prediction for distributed data-intensive applications." In: *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*. Portland, Oregon, United States: ACM Press, 1999, p. 36.

[125] Yoav Etsion, Dan Tsafrir, and Dror G. Feitelson. "Desktop scheduling: how can we know what the user wants?" In: *NOSSDAV '04: Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*. Cork, Ireland: ACM Press, 2004, pp. 110–115.