Development and Analysis of a Physics Server for Large Virtual Worlds

Shehan Sirigampola
University of Central Florida
Orlando, Florida
ssirigam@ist.ucf.edu

Sean Mondesire
Army Research Laboratory
Orlando, Florida
mondesire@gmail.com

Glenn A. Martin
University of Central Florida
Orlando, Florida
martin@ist.ucf.edu

Jonathan Stevens
JSARES
Orlando, Florida
jej.stevens@gmail.com

ABSTRACT

Through the use of virtual simulation, trainees can practice procedures and make mental connections necessary for improved task performance on the real task. However, they do have limitations. One is the size of the environment (measured by number of entities). The Military OpenSimulator Enterprise Strategy (MOSES) project from the U.S. Army Research Laboratory is working to improve the next generation of simulation's training effectiveness and is exploring methods to increase the number of simultaneous soldiers within a 3-D virtual world.

The presented work investigates expanding capacity of a simulator by off-loading work onto a remote server (with potentially powerful and/or special hardware). This may increase entity count supported and support the use of less-powerful clients (conversely, boost performance when using that hardware). Initial focus is on off-loading physics calculations. A remote server was built upon the Nvidia PhysX engine, which is optimized for multi-threaded and GPU-enabled calculations, and a plug-in within OpenSimulator developed to communicate with that server. In addition to the architecture, results of an analysis comparing this remote physics capability to three integrated physics capabilities (Open Dynamics Engine, Bullet, and PhysX) is presented.

ABOUT THE AUTHORS

Mr. Shehan Sirigampola is an Assistant in Simulation at UCF's Institute for Simulation and Training. He has worked on numerous projects, including augmented reality, intelligent training tools, and real-time virtual environments. He has a B.S. in Computer Science from the University of Central Florida.

Dr. Sean C. Mondesire is a Postdoctoral Research Fellow at the U.S. Army Research Lab (ARL) and holds a doctoral degree in computer science from the University of Central Florida (UCF). His research focus is on the expansion of virtual world technologies to be used in tactical, military training.

Dr. Glenn A. Martin is a Research Assistant Professor at UCF's Institute for Simulation and Training. He earned a Ph.D. in Modeling & Simulation in 2012 from the University of Central Florida. He pursues research in adaptive and intelligent training, game-based learning, multi-modal simulation, and interactive high performance computing.

Dr. Jonathan Stevens is a Research Scientist, specializing in training simulation research. Dr. Stevens is a retired Lieutenant Colonel of the United States Army with over 22 years of military experience as both an Infantry and Acquisition Corps officer. He received his Ph.D. in Modeling and Simulation from UCF**.**

Development and Analysis of a Physics Server for Large Virtual Worlds

Shehan Sirigampola
University of Central Florida
Orlando, Florida
ssirigam@ist.ucf.edu

Sean Mondesire
Army Research Laboratory
Orlando, Florida
mondesire@gmail.com

Glenn A. Martin
University of Central Florida
Orlando, Florida
martin@ist.ucf.edu

Jonathan Stevens
JSARES
Orlando, Florida
jej.stevens@gmail.com

INTRODUCTION

Persistent virtual worlds can provide an excellent environment for simulation-based training (SBT). They provide ease of access to trainees, as well as robust environments for many different types of training. The effectiveness of virtual worlds in training has been exhibited in exercises such as room clearing (Lackey, Salcedo and Maxwell, 2014). However, these environments do not scale well with the number of dynamic entities (i.e.: avatars and objects that are non-stationary). One of the significant components of a virtual world is the physics engine, which is responsible for resolving the effects of various forces, such as gravitational force, on the rigid and soft bodies in the simulator. It is also responsible for resolving collisions and simulating particle effects. The simulator updates the state of the environment at rates upwards of ten times a second in order to maintain the users' interactivity with the environment. During each of these updates the physics engine must simulate all physical interactions between entities and the environment. The computational cost involved with the physics engine's responsibilities can be excessive, thus it can serve as a significant factor in its inability to scale (Mondesire, Stevens and Maxwell, 2016).

Since the physics engine can have a significant impact on the performance of a persistent virtual world simulator, the *Army Research Laboratory's* (ARL) *Military OpenSimulator Enterprise Strategy* (MOSES) project has been evaluating the effectiveness of several approaches to improve this behavior. MOSES uses a derivative of OpenSimulator (OpenSim), which is an open source server used to host three-dimensional virtual worlds. The project aims to be able to field at least a company of training soldiers along with a robust environment inside OpenSim. It is also important to increase scalability of the simulation while still maintaining real-time framerates so as not to degrade the users' experience. In order to achieve these goals, alternatives to OpenSim's current physics engine options were implemented. The presented work explores one of these alternatives, a distributed approach to physical simulation. An analysis of this approach is presented in this work, which compares the distributed approach to a more traditional integrated one.

BACKGROUND

Before working on physics engine alternative, OpenSim's current physics simulation infrastructure was first analyzed. OpenSim uses a plug-in architecture for its physics simulation component. Thus, if a contributor wanted to introduce a new physics engine to OpenSim, he/she simply has to fulfill the requirements of the physics interface defined by OpenSim. By default, OpenSim has two physics engines at its disposal: the *Open Dynamics Engine*™ (ODE) and the *Bullet Physics Library* (Bullet). Several tests were performed in order to evaluate each engine's performance. These tests revealed that Bullet outperformed ODE in high dynamic entity situations (Mondesire, Maxwell, Stevens, Zielinski & Martin, 2016).

Once Bullet was established as the more optimal engine, further analysis of its performance was conducted. OpenSim's Bullet plug-in, which serves as an interface between OpenSim and Bullet, has a setting which determines whether the plug-in runs on its own thread or runs on its owner's thread. These two settings were compared to each other. The results showed that the metrics for measuring the frame time of the separate thread Bullet plug-in were ineffective and misleading. Furthermore, running Bullet in a separate thread created more overhead, which caused a decrease in that configuration's performance (Mondesire, Maxwell, Stevens, Zielinski & Martin, 2016). This led us to seek out a more robust multi-threaded implementation for OpenSim's physics.

Since Bullet does not have built-in multi-threading support (other than forking onto a single, separate thread), other engines were explored. As a result of this search, we chose to focus on NVIDIA®'s PhysX® engine. Among the reasons for choosing this engine was that PhysX has built-in multithreading. The developer needs to only specify the number of threads, and PhysX's CPU dispatcher will distribute tasks among the worker threads automatically. Another reason for using PhysX is that it has been highly optimized for real-time physics simulation and has had many applications in simulation and entertainment (Micikevicius, 2009). Also we hope to add GPU accelerated rigid body physics simulation and PhysX's built-in GPU acceleration in the future. Currently, rigid body simulation is not performed on the GPU because traditional rigid body simulation is not very parallelizable, and so does not perform well on the GPU (Micikevicius, 2009). We built a new plug-in for OpenSim using PhysX that has similar functionality to the existing Bullet plug-in. Several tests were performed that compared the new plug-in to its counterparts, and the results showed that the PhysX plug-in was a marked improvement over the Bullet and ODE plug-ins in high load situations (Mondesire, Maxwell, Stevens, Zielinski & Martin, 2016).

In addition to creating the PhysX plug-in, we also started working on a distributed OpenSim physics plug-in (remote physics plug-in). Instead of performing the physics computations on the local machine, this plug-in would communicate with a remote server, which would act as a physics simulation service. Multiple OpenSim plug-ins can connect to and run many simulations on a single remote server. This remote manager is called MOSES Remote Manager (MRM). In order to relay the state of the simulation between the remote physics plug-in and MRM, we created a simple protocol used to describe object state, object geometry, joints, and general simulation state. These messages are in binary form so as to cut down on packet size and serialization costs. Communications between the remote physics plug-in and MRM are facilitated through two different transport protocols: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). TCP is a reliable protocol that is used to relay important control messages, such as creating objects and advancing the simulation time, since losing these messages would result in major loss of fidelity in the virtual world. High frequency messages are sent through UDP, because a loss of one of these messages would only affect a single frame of simulation. UDP provides lower latency than TCP for these messages. Mainly object state updates are sent through UDP since these are sent out during each simulation frame in which they are active. The remote physics plug-in contains multiple threads in which data can be processed concurrently. This is so that network data transmission can occur uninterrupted while data is being serialized or deserialized (See Figure 1). MRM uses a plug-in architecture, which also allows it to perform simulation tasks concurrently with network data processing. This approach could offload a significant amount of computation to a remote machine, but would also add networking latency. The rest of this work explores the benefits and costs of a distributed approach to physics simulation and compares that approach to a more traditional integrated one.

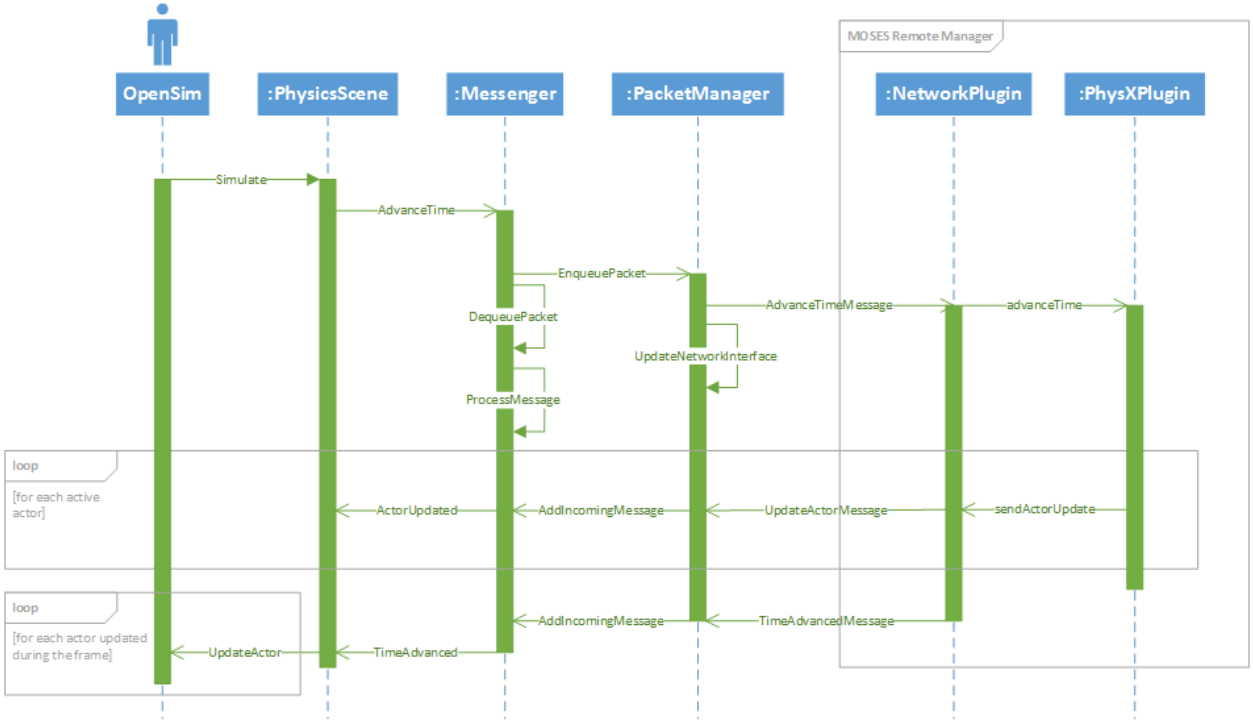

Figure 1. Sequence Diagram of AdvanceTime

METHODOLOGY

This paper compared the performance of distributed physics processing against a traditional, integrated physics engine. In this experiment, *distributed physics* processing was realized through the execution of two separate program instances: one module that processed all of the simulation's physics calculations, and one for all of the remaining simulator's functionality. Distributed physics was compared with an i*ntegrated mode*; the default physics architecture and mode of execution where all of the simulator's components (including the physics engine) are tightly coupled into a single program instance. The presented experiment quantifiably compared how both physics engine modes affected simulator performance and physics engine load scalability.

For performance comparison, each configuration was examined on how well both the simulator and physics engine reacted to increased physical object load. To generate load, a testing scenario was designed to repeatedly stress test each engine configuration. The ball pit scenario in (Mondesire, Stevens & Maxwell, 2016; Mondesire, Maxwell, Stevens, Zielinski & Martin, 2016; Mondesire & Maxwell, 2016) was employed in this experiment. In the scenario, the simulated environment contains a series of 10 concentric-circled, 5-meter tall walls. Each circled-wall resided 10-meters from its two nearest neighbor. Each wall was non-penetrable, static, and fixed in place.

The scenario also contained a ball spawner that was suspended 15 meters off of the ground, centered to the inner-most wall. When invoked, the ball spawner generated 500 balls, where each ball was 1-meter in diameter and was comprised of 1 active primitive (AtvPrm), physical object. Primitive objects, also known as *prims*, are basic shapes inside OpenSim that can have physical, scripted, and persistent properties. With the active primitive set to physical, the object could collide with other objects and was subject to gravitational forces; both properties forced physics engine stimulation. Each generated ball applied additional computational load onto the physics engine. Therefore, when the number of physical object reached certain thresholds, the simulator and physics engine began to degrade in performance. The presented experiment identified when these thresholds were reached with each physics engine configuration and measured performance at certain load intervals.

To identify the load threshold of distributed physics, the ball spawner generated and dropped 500 physical balls repeatedly at 30 second intervals until 10,000 were present in the virtual world. At each interval, performance data was collected. Once all of the balls were generated, the simulator was reset and the scenario was repeated 29 more times for a total of 30 independent trials. The same conditions and process were followed for integrated physics that were captured in previous experimentation (Mondesire & Maxwell, 2016). This work compared the newly collected distributed physics data with the previously captured integrated physics results under the same exact testing conditions.

To facilitate each experiment trial, a modified version of OpenSimulator 0.8.2 was used to generate the simulated world. The modification added NVIDIA's PhysX 3.33 to the simulator as the physics engine. The new engine supported two modes: integrated and distributed physics, the two independent variables of the experiment. Based on the identified optimal thread-allocation from previous experimentation (Mondesire, Stevens & Maxwell, 2016), PhysX was allocated four CPU threads for both integrated and distributed modes. All experiments were executed on the same hardware, supporting an Intel i7 6-core processor with 12 concurrent threads and 32 GB of RAM. The Ubuntu Desktop 14.04 64-bit operating system was used to host the software components of the simulator.

DATA ANALYSIS

The experiment used the physics engine configurations (integrated and distributed) and the number of generated physical, active primitive (*AtvPrm*) objects as independent variables. The dependent, response variables were the *simulation frames per second* (*SimFPS*) and *physics frame time* (*PhysFT*). SimFPS is measured by the number of simulation frames the simulator was able to completely process in a second. A complete frame included one round of updates for every simulation engine component, including the scripting, persistence, and physics engines. OpenSim restricted the frame rate to execute no faster than 11.33 frames per second. SimFPS provided a quantifiable measure of how the overall simulator was performing at any point of the simulation. The second dependent variable was the physics frame time. PhysFT isolated the performance analysis to just the physics engine. PhysFT measured the time required to completely process a physics simulation, which evaluated all physical objects in the world for collisions and updated object positions. PhysFT had no maximum upper-bound and was measured in milliseconds (*ms*).

The first evaluation analyzed the minimum degradation load of each configuration. The *minimum degradation load* (*MDL*) was the least amount of physical objects the simulation was given that produced an average SimFPS below 9 frames per second. Traditionally, 9 SimFPS was the threshold at which the simulator began to perform poorly with delayed interactions between the user and the simulator. Integrated physics produced an MDL of 4,500 physical object, which yielded a 7.79 SimFPS (SD = 0.44). Distribute physics produced an MDL of 500 objects with of 6.7 SimFPS (SD = 2.71). Therefore, integrated physics was able to handle significantly more load than the distributed configuration.

Next, the SimFPS of each configuration was analyzed at different load levels. This analysis determined how the two configurations affected overall simulation performance. With 1,000 physical objects, integrated physics averaged 11.33 (SD = 0.12) SimFPS while distributed physics averaged 3.19 (SD = 0.68). Two-tailed Z-test with an $\alpha = 0.05$, determined that integrated significantly outperformed distributed at this load level ($p < 0.0001$). Integrated also significantly outperformed distributed mode at 5,000 objects with an average of 6.53 SimFPS (SD = 0.49) against distributed's 0.3 SimFPS (SD = 0.16), $p < 0.0001$. At the highest amount of physical object load (10,000 AtvPrm), integrated mode averaged 1.72 SimFPS (SD = 0.3) while distributed averaged 0.3 SimFPS (SD = 0.31), $p < 0.0001$. Again, integrated significantly outperformed distributed physics at every load level. Table 1 displays all of the SimFPS averages and standard deviations. Figure 2 graphs the changes in SimFPS across all of the tested physics load intervals.

Table 1. SimFPS Results for All Load Levels

SimFPS				
	Integrated		Distributed	
AtvPrm	AVG	STD	AVG	STD
1,000	11.33	0.12	3.19	0.68
2,000	11.30	0.10	1.32	0.22
3,000	10.97	0.13	0.69	0.12
4,000	9.65	0.33	0.45	0.16
5,000	6.53	0.49	0.30	0.16
6,000	4.18	0.41	0.44	0.20
7,000	1.54	0.38	0.33	0.19
8,000	0.73	0.12	0.36	0.29
9,000	0.68	0.10	0.38	0.40
10,000	1.72	0.30	0.30	0.31

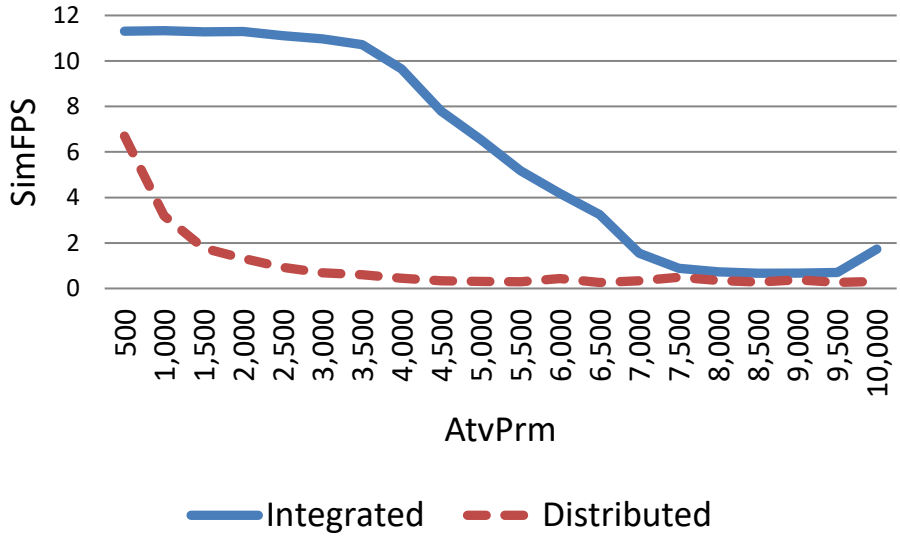

Figure 2. SimFPS for Each Physics Object Load Interval

Finally, the physics frame times for both configurations were analyzed. The analysis identified how much time each physics configuration required to perform a full physics frame's worth of processing. At 1,000 physical objects, integrated averaged a PhysFT of 0.81 ms (SD = 0.05) and distributed averaged 654.51 ms (SD = 265.6). At 5,000 physical objects, integrated physics averaged a frame time of 289.44 ms (SD = 24.59) and distributed averaged 6,141.13 ms (SD = 235.04). Lastly, at the highest load level of 10,000 physical objects, integrated averaged 1,081.37 ms (SD = 84.5) and distributed averaged 6,269.5 ms (SD = 1,011.04). Z-tests confirmed that the frame time differences at 1,000, 5,000, and 10,000 AtvPrms were all significant ($p < 0.0001$); therefore, distributed physics produced significantly larger frame times than integrated at the tested load intervals. Table 2 displays the physics frame times of each configuration while Figure 3 graphs the changes in frame time as physical object load increases.

Table 2. PhysFT Results for All Load Levels

PhysFT

	Integrated		Distributed	
AtvPrm	**AVG**	**STD**	**AVG**	**STD**
1,000	11.33	0.12	3.19	0.68
2,000	11.30	0.10	1.32	0.22
3,000	10.97	0.13	0.69	0.12
4,000	9.65	0.33	0.45	0.16
5,000	6.53	0.49	0.30	0.16
6,000	4.18	0.41	0.44	0.20
7,000	1.54	0.38	0.33	0.19
8,000	0.73	0.12	0.36	0.29
9,000	0.68	0.10	0.38	0.40
10,000	1.72	0.30	0.30	0.31

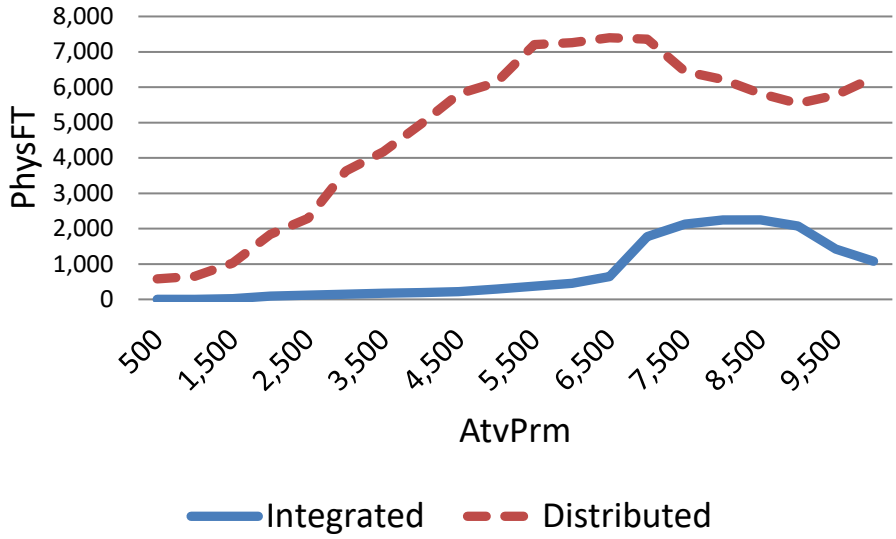

Figure 3. PhysFT for Each Physical Object Load Interval

CONCLUSION

The results presented here show that the integrated physics approach outperformed the remote physics server approach. This is perhaps not surprising at a low number of objects. However, it does raise three questions to be pursued in future work.

Can we improve the distributed physics server as is? Given the current design, there is room to improve the server protocol itself. Additional messages can be added that avoid potential wasted sends. In addition, some updates could potentially be ignored. The PhysX library itself takes object state as input, adjusts the object parameters (e.g. position) accordingly, and returns that state. These updates into, and out of, PhysX could be improved through better dead reckoning and other methods.

At what level will a distributed physics server help? Integrated physics clearly has a limit to its performance. After this number of objects, one could expect to see a remote physics server approach perhaps perform better than an integrated approach. The experiment in this paper tested up to 10,000 objects. Would a higher quantity show any differences (particularly if other improvements to the distributed physics server are made)?

How does additional hardware capability to a remote physics server affect the results? The notion of a distributed physics server is based upon a single server having the necessary hardware to simulate the necessary environment as opposed to each and every client doing so. The server could contain multiple processors and/or Graphics Processing Units (GPUs) to drive performance.

This initial investigation into a remote physics server showed a wider performance gap than anticipated. However, further investigation is warranted. Indeed, such cloud-based environments have many appealing factors and could address many computing needs for virtual environments in the future.

ACKNOWLEDGEMENTS

Acknowledgements to the Dr. Douglas Maxwell and the Military OpenSimulator Enterprise Strategy (MOSES) group at the Army Research Laboratory. This work was performed under Cooperative Agreement W911NF-15-2-0004. All opinions are those of the authors, however.

REFERENCES

A. Maciel, T. Halic, Z. Lu, P. Nedel, S. De (2009). Using PhysX Engine for Physics-based Virtual Surgery with Force Feedback. *The International Journal of Medical Robotics and Computer Assisted Surgery, Volume 5, pages 341-353.*

P. Micikevicius (2009). 3D Finite Difference Computation on GPUs using CUDA. *GPGPU-2 Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units. Washington D.C., USA.*

S. J. Lackey, J. Salcedo & D. B. Maxwell (2014). Virtual World Room Clearing: A Study in Training Effectiveness. *Proceedings of Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC). Orlando, FL.*

S. Mondesire, D. B. Maxwell, J. Stevens, S. Zielinski, G. A. Martin (2016). Physics Engine Benchmarking in Three-Dimensional Virtual World Simulation. *Proceedings of MODSIM World. Virginia Beach, VA.*

S. Mondesire, J. Stevens, D. B. Maxwell (2016). Virtual World Performance Analysis with Vertically Scaled Multi-threaded Physics. *Proceedings of Spring Simulation Multi-Conference. Pasadena, CA.*

S. Mondesire & D. B. Maxwell (2016). Physics Engine Threading Design and Object-scalability in Virtual Simulation. To appear in IEEE Computer Graphics & Applications: Special Issue on Computer Graphics for Defense Applications.