Game-Engine-Assisted Research framework for Scientific computing (GEARS) in Virtual Reality

Brandon K Horton,a,d Erick Moen,b Anders Hafreager,c Michael Qian,d Tao Wei,e and Aiichiro Nakanoa,d

|  |
| --- |
| 1. Department of Computer Science, University of Southern California, Los Angeles, CA 90089-0484, USA 2. Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90089-0241, USA 3. Faculty of Mathematics and Natural Sciences, University of Oslo, 0316 Oslo, Norway 4. Department of Physics & Astronomy, University of Southern California, Los Angeles, CA 90089-0484, USA 5. Department of Chemical Engineering, Lamar University, Beaumont, TX 77710, USA |

Abstract

The recent widespread availability of commodity head-mounted displays (HMDs) has fueled a virtual reality (VR) renaissance in several entertainment and media industries. This resurgence has produced a number of enabling technologies, both hardware and software, that reduce the barrier to entry for new applications of VR. In an effort to leverage these advantages for scientific computing, in this report, we provide a workflow for use with game engines to adapt visualization and simulation techniques for VR. This framework accommodates multiple programming languages and game engines. We also present a hardware-agnostic platform, based on LAMMPS, that allows researchers to directly port existing codebases to VR using familiar programming languages. Additionally, we present a number of software tools that take advantage of VR’s enhanced functionality for scientific computing. To demonstrate the effectiveness of these systems, we visualize the results of simulations carried out on high-performance computing clusters (HPCCs) as well as run scientific computing code from within the visualization. Building off our previous work on the subject, we employ the workflow to implement a real-time VR simulation of biological electron transfer (ET) in cytochrome proteins. We also outline the inclusion and optimization of LAMMPS for use in VR. Lastly, we develop new tools to enhance these simulations, including a shader that can be used as a virtual confocal microscope. We apply this virtual scope to exam a molecular model of a desalination membrane.

PACS: 02.70.-c, 05.10.Ln, 07.05.Rm  
*Keywords:* Data Visualization; Virtual Reality; Real-time Simulation, LAMMPS

# Introduction

Data visualization plays a key role in scientific discovery. Though quantitative analysis is indispensable, researchers are often forced to apply metrics blindly. Unfortunately, these statistics are not only limited in their ability to describe the system under test but can often be misleading. An elegant illustration of this principle is Anscombe’s Quartet. Though this particular illustration applies to summary statistics, it would be trivial to extend this concept to other metrics employed to describe and understand more complicated systems. It is no wonder, then, that researchers may find it difficult to decipher previously unidentified trends, while nuances can be missed entirely. Often, this forces researchers to make assumptions about data sets they intend to analyze, creating a source of confirmation bias, as they search for trends they believe exist. If this is true in even modestly-sized data sets, such as Anscombe’s Quartet, these issues are exacerbated many times over as data sets grow in volume, variety, velocity, and veracity. This is an especially critical consideration for programs and simulations utilizing high-performance computing clusters (HPCCs). These simulations are often accompanied by monumental data sets that can be unwieldy and difficult to parse [3–6]. Compounding these issues, because of its varied nature, data visualization for scientific computing is a fragmented field generally requiring additional software or programming skills.

Data visualization programs like ParaView and Visual Molecular Dynamics (VMD) do an excellent job of providing a straight-forward interface with which to create three-dimensional (3D) images and observe patterns in the output of simulations utilizing HPCC [7, 8]. Large-scale, 3D visualization rooms like the CAVE2 (developed by Electronic Visualization Laboratory at the University of Illinois at Chicago), meanwhile, provide unique spaces to collaborate and review data representations on a more manageable scale [9, 10]. Both of these solutions, however, require specialized knowledge of the respective systems and clear expectations of the simulations outcome at the start. These implementations also lack a visceral interface that could enable users to draw new insights and identify patterns not discernable from available metrics. Furthermore, these systems are expensive, making them cost prohibitive for many researchers [11].

A more feasible solution may be an easily-accessible, agnostic software platform designed for the increasingly available head-mounted displays (HMDs) developed for virtual reality (VR). VR is a powerful visualization tool that has grown steadily in popularity over the past two decades. This growth has increased exponentially with the development of more affordable, commodity head-mounted displays like the Oculus Rift and HTC Vive as well as the introduction of mobile VR solutions like Google Cardboard. These commodity HMDs hold the promise of a truly immersive, fully customizable experience. Despite its widespread adoption for media applications, as well as its potential as the most immersive and intuitive method for viewing data, VR has yet to see widespread use in scientific computing and simulation.

The tools for immersive data visualization in VR are entirely accessible and inexpensive, but under-utilized by the scientific community. The resurgence of VR hardware has, in turn, spurred on a number of video game engines to include native support for a host of equipment. The Unity game engine, for example, has become a key programming platform for developing VR experiences and has been included in workflow for visual effects in films. This versatility, combined with the ability to run scripts in either Java or C#, has the potential to bring immersive data visualization to the desktop of any scientist.

This report outlines the development of a software platform that facilitates the adoption of VR technologies and allows researchers to take advantage of the unique analytical advantages that the medium offers. The platform utilizes game engines and is thereby designed for use with common programming languages as well as existing algorithms and codebases. To demonstrate the utility of this platform, a number of VR-specific tools to facilitate and improve data analysis are also developed. Additionally, we outline a number of interface technologies that can be incorporated for increased user comfort and control.

# Methods

## Hardware Options

Before we progress to our platform and its development, we begin with a brief review of some of the available hardware and software options for desktop VR systems. This section is not meant as an exhaustive review of these options, but, rather, to draw attention to these options through a few examples. In previous work, we utilized the Oculus Rift Development Kit 2 (DK2). This system consists of a HMD and a sensor to detect head-movement. Though the DK2 was one of the first widely-available commodity VR headsets, it was a developer release and has since been replaced with the consumer version (CV1). It has also been joined in the marketplace by a number of competing headsets.

Chief among these competitors is the Vive, developed by HTC and Valve. While the Vive is more expensive than the Rift ($300 more at the time of publication), the system includes two wireless controllers, a built-in camera on the headset, and two base stations that track movement (with submillimeter accuracy) in an area of 4.6 m2 [46, 47]. These capabilities provide unique opportunities for data visualization.

In addition to these “high-end,” desktop systems, mobile VR solutions provide an alternative for immersive visualization at a much lower cost. While the desktop systems are significantly less expensive than previous 3D hardware like the CAVE2, these solutions still represent a significant investment for academia with installed costs on the order of $2,000 [11–13]. Alternatively, mobile VR headsets like the Gear VR or the Google Daydream are available for less than $100 [14]. Furthermore, options like Google Cardboard are available for $15 [48]. These options rely on existing hardware (i.e. the user’s cell phone) but represent an attractive solution for academic settings because of their minimal cost and installation overhead. Additionally, mobile VR could represent an excellent conduit for collaboration among researchers, delivering a comparable dynamic visualization to those without access to the higher-performance systems.

Regardless which HMD is selected, a suitable PC will be needed for development and, in the case of desktop systems, operation. Our system for the following experiments will be a PC with an Intel Core i5 quad-core central processing unit (CPU) and NVIDIA GeForce GTX 750 graphic processing unit (GPU). This PC is adequate for rendering the high polygon-count and memory intensive data associated with scientific computing. More information on recommended system specifications can be found on manufacturers’ websites.

## Software Options

Our platform makes use of commodity game engines to simplify access to VR headsets. As with the previous section, this brief review is not exhaustive but highlights options we have found particularly useful in our endeavors. In previous work, Unity was used as our preferred engine because of its native integration with the Rift, its robust asset import procedure, and the availability of prefabs within its store. These options make it ideal for the rapid prototyping common to data visualization. Unity’s free personal license also make it an attractive option from a cost perspective. Additionally, simulations developed in Unity can be easily ported from the desktop HMDs to lower-cost mobile VR headsets for dissemination.

While Unity was our initial choice, a number of alternative game engines are well-suited to VR development. One such alternative is the Unreal Engine by Epic Games, which, along with Unity, is one of the most widely used development platforms and among the first to provide native VR support. Unreal is freely available for Academic purposes (though commercial products are subject to a 5 percent royalty). Additionally, the engine supports scripting in C++, making it an attractive option for scientists looking to reuse existing code and quickly incorporate VR-based data visualization into their research endeavors [45].

LAMMPS, first developed by Steve Plimpton, is an invaluable, well-supported library capable of reducing the complicated code-writing necessary for MD down to several line scripts. Although one conventionally sets up a LAMMPS simulation then runs a large amount of iterations in one call, we wish to run single time-steps and immediately visualize them via Unreal. To do this, we chose to compile LAMMPS into a dynamically linked library (DLL), which can then be accessed via Unreal, thus opening the door to real-time simulation visualization and steering. This, however, proposed several challenges. LAMMPS compilation relies on many POSIX commands and therefore must be done on a Linux system. As far as official documentation permits, Unreal only provides a library for importing Windows DLLs into the game engine. It may be possible to do this on the Unreal Build for Linux, but there does not appear to exist nearly as much documentation or support for the engine on that operating system ([how to dll in unreal](https://wiki.unrealengine.com/Linking_Dlls)). Furthermore, VR is best supported on Windows (For both SteamVR and Oculus). Luckily, we were able to cross compile LAMMPS for Windows using the MINGW64 compiler. And conveniently, Microsoft recently released *Bash on Ubuntu on Windows* for Windows 10, allowing us to do the compilation and importing to Unreal without ever leaving the Windows 10 OS. We’ll include our step-by-step method for creating the LAMMPS DLL via *Bash on Ubuntu on Windows* in Appendix # as well as a copy of the MINGW64 Makefile we used in the supplementary materials.

|  |
| --- |
| **Figure 1**: a. Workflow for importing precomputed geometry data into GEARS. b. Workflow for setting up real-time manipulable simulations in GEARS. c. Running simulation, previewed via the development environment for UnrealGEARS |

## Control Scheme

A carefully designed control scheme is essential to facilitate interaction between the user and various data representations within the immersive simulation, such as object manipulation, spatial translations, and temporal scaling. This interaction, especially as it relates to user motion within the environment, should mimic natural motion as much as possible to limit user discomfort [49–51]. Game engines can accept input from a variety of sources, including the keyboard and mouse, an Xbox Kinect, and touchscreens, among many other modalities [33, 45]. This section describes a simple, yet versatile, control scheme that includes the LeapMotion, Oculus Touch, and HTV Vive motion controllers, among others. Each of these tools offers intuitive gesture control of objects and navigation within the simulation [52].

LeapMotion already provides the software necessary to have the user’s arms and hands visible in game. For Unity, LeapMotion provides a prefab, “LMHeadMountedRig”, which you can drag into your scene (just be sure to remove the default camera), and the hands will already be in the scene. If you want added control, LeapMotion also provides Detection Utilities that will detect pinches, hand positions, and palm direction, among other features. Using custom C# scripts that inherit from these detectors, we added the ability to grab and rotate our simulation data. We also used these detection utilities to toggle the visibility of different parts of our simulation geometry data, which led to the development of our Virtual Confocal Microscopy system, detailed in section #.#.#. Although user specific controls can be implemented fairly easily with custom C# scripts, LeapMotion also provides a sample script, called *RTSLeap.cs*, for rotating, translating, and scaling in-game objects. If situation-specific controls are not necessary, this is perhaps the easiest method for adding basic LeapMotion controls for a simple data-viewing instance of UnityGEARS. A step-by-step procedure to use the LeapMotion and *RTSLeap* script will be provided in Appendix ## and in the supplementary materials.

Although the LeapMotion provides immersive virtual models of the users hands, and consequently a more immediately learnable control scheme, the device is limited by its lack of standard buttons to execute commands. All operations are initiated based on gesture controls, but some users may wish to have dedicated buttons, triggers, and axes that they can use to control and fine-tune their simulation data. This is highly useful in simulation steering when the user may need to pause or speedup the simulations, or simulation viewing when they need to move the camera position across large distances.

Our HMDs of choice, the Oculus Rift and HTC Vive, both have dedicated motion controllers: Oculus Touch and Vive Controller. Programming for these is similar to that of the keyboard and mouse. Unity and Unreal both have Motion Controller input libraries, which can tell us when any of the buttons on these devices is pressed or released. Using these, we development the movement and playback system in UnrealGEARS. The motion detection is done automatically via the Oculus sensors and Vive Base Stations; No additional coding is necessary. We will include blueprints and scripts for each of these control schemes in the supplementary materials as well as a video tutorial on how to integrate them into GEARS

It should be noted that in lieu of access to these specific controller devices, the keyboard and mouse are still easily integrable options into GEARS. Both have readily accessible functions for detecting when and where key presses and mouse clicks occur. For instance, Unity has the “Input.GetButton()” scheme for the keyboard keys, and Unreal’s blueprint editor has event nodes that fire for each key when pressed/released. However, once an HMD is worn by the user, the keyboard and mouse are not automatically visible in-game, making them difficult options for integration into VR-based simulation viewers. It would be possible to project a virtual keyboard into the viewer’s virtual space, but this would be considerably more effort than using the dedicated motion controllers for the Oculus Rift and HTC Vive, or even the LeapMotion.

## Performance Optimization

To identify bottlenecks and improve performance optimization, it helps to categorize the simulation and rendering calculations into 3 stages: game-related computation, draw calls, and rendering via the GPU. Because simulations often involve many particles and heavy time step calculations, several measures were taken to ensure that our computations were alleviated from or offloaded to the optimal stages from the 3 listed above. Our main methods to increase our particle count and simulation complexity were a combination multi-threading, static mesh instancing, and billboarding. The easily accessible source code and libraries provided by Unreal Engine 4 to accomplish these 3 tasks served as a quintessential motivation to develop GEARS for LAMMPS in Unreal. However, it should be noted that these optimizations are technically feasible in Unity 5 as well.

For multithreading, Unreal Engine 4 provides the FRunnable class, which can run a member function on a separate thread, and C# (powering Unity) has its own Thread library. We use a separate thread to handle the calculation of the next time step in our KMC and LAMMPS simulations, discussed in section 3.#.# and 3.#.#. When the time step is finished, the spawned thread signals to the main thread that it is safe to update heme occupation states or particle positions in the running game. By having the simulation time step offloaded onto a separate thread from the engine’s main thread, the user is then able to move and look around at the visualized particles without the worry that the engine will stall due to a particularly long time step calculation. Such stalls could cause the VR HMD to stutter or skip frames, spoiling the experience for the scientist viewing the simulation. Although this alleviates the actual simulation calculation from the main game thread, it would still need to make draw calls to the GPU for the update positions of each particle – updating game objects from separate threads is highly discouraged and led to many crashes in our system. Depending on the system size, this can be an enormous bottleneck for the GEARS system. To alleviate this strain on the engine and GPU, we utilized instancing of our particle static meshes.

Instancing is the technique used to render many objects that all share the same mesh and material, among other similar properties ([potential paper to cite](http://ieeexplore.ieee.org/abstract/document/945338/?reload=true)). Since all particles in our simulation will be represented via colored spheres, we can use this technique to cut down on the draw calls to our GPU. Unreal Engine 4 provides an instanced static mesh class for this exact purpose called UInstancedStaticMeshComponent. Using this class we were able to keep of one instantiated object per particle type (e.g. water, oxygen, sulfur, etc.). Positions, orientations, and render materials for all instances of a given type were tracked via their corresponding instantiated class. This reduces the draw calls to the GPU to once per particle type, as opposed to once per particle, significantly reducing the strain on both the main game thread and its communication with the GPU. Lastly, as a measure of performance optimization for rendering these real-time simulations, we cut down GPU computations by using billboards to represent our particles as opposed to actual spheres.

Billboarding is a technique used in computer graphics to represent axially and radially symmetric objects with considerably low rendering costs ([potential reference](http://www.cs.virginia.edu/~gfx/courses/2002/BigData/papers/Interfaces%20and%20Software%20Systems/IRIS%20Performer.pdf)). It involves projecting a 3D manifold or geometric structure as texture on a 2D surface that always orients itself to face the camera/eye of the user. If used with appropriate shading, this can give the illusion that a 2D image is actually a 3D manifold in space. This is a useful tool in virtual reality applications as it allows us to represent a large amount of objects using as few simplicial complexes as possible. In our case, we wish to represent the particles in a simulation as mathematically perfect spheres such that lighting during our simulations permit proper depth perception and shading. Rather than use the provided Unreal Engine 4.16 static sphere mesh, which takes ### (i think 400) triangles to represent, we can use a 2D billboard, which only requires 2 triangles to represent. This is done through the Unreal Engine 4 Material Blueprint interface, a high level visual programming system for shading. We will provide our billboard material blueprint in the supplementary materials. To add semi-realistic reflections and shading, we apply a normal map to the billboard. The added benefit of doing this allows for proper depth perception of particles, making data acquisition more intuitive for human users.

The above three methods of performance optimization were used to facilitate efficient rendering in the engine. However, we also used OpenMP to speed up the running LAMMPS simulation itself. By downloading the OpenMP package for LAMMPS before compilation, we were able to build the library into our DLL, allowing us to optimize LAMMPS for the computer they it ran on. Once compiled into the DLL, we only needed to add the lines:

|  |
| --- |
| package omp 40  suffix omp |

to the beginning of our LAMMPS input scripts. Note that “40” is the number of effective cores in our system, therefore, this will vary for different users’ systems.

# Results

Building on the options presented in the methods section, each of the following sub-sections reviews a specific aspect of GEARS. We begin with viewing pre-computed results in VR before advancing to a discussion of dynamic simulations. The last sections discuss new tools developed to take advantage of the unique perspective offered by VR as well as programming enhancements to improve run time and frame rate.

## Interactive Viewing

The most straightforward application of VR to data visualization is interactive viewing of pre-computed results. To demonstrate this aspect of GEARS, we will use the results from a ##### atom MD simulation. The simulation investigates the anisotropic frictional response (and resulting heat dissipation) of RDX crystal along various crystallographic planes. The overall simulation time is , with a time step of , resulting in a total step count of . The position of every molecule is calculated at each step. We isolate one step, or frame, of this simulation and import this snapshot to be rendered and manipulated by UnityGEARS. We do so by following the iBET workflow, which is described in a previous manuscript [ref]. In brief, VMD and Blender can be used to create an object that can be imported directly into game engines. Once imported, the .obj file can be added to the scene and the appropriate script (for example, RTSLeap for the Leap Motion controller) can be added as described in the methods section. The result is shown in Figure #.

**Fig. 2.** Snapshot of the friction simulation. The Leap Motion controller allows the user to naturally manipulate the results of the simulation and the HMD provides a detailed means of observing it.

This aspect of GEARS is a quick, straightforward outlet for immediate visualization of snapshots of data from HPC simulations or structure files like those found on the RCS Protein Data Bank. Though we only use one “frame” of data for illustrative purposes, this process could be expanded by creating multiple scenes, each containing the results from a different time step. Then the simulation could be replayed dynamically frame by frame.

## Interactive Simulation

To enhance the dynamic aspects of GEARS, in this section we go beyond recreations of previously computed results and 3D movies to generate fully interactive VR simulations. Our first proof-of-concept is a direct adaptation of existing code, written in C, for a molecular dynamics (MD) simulation of particles with Lennard-Jones potentials. We translated this MD code into C# (for Unity) and C++ (for Unreal) and used it to calculate changes in each particle’s position for each time step on each frame update call. The positions of each rendered particle were adjusted on the same update.

An alternative to this approach of directly importing existing code is to reinterpret it such that it takes advantage of some of the unique possibilities within the game engine. To demonstrate this aspect of GEARS, we extended our previous work on biological electron transfer (ET). The original version, immersive biological electron transfer or iBET, was used to visualize the results of a kinetic Monte Carlo (KMC) simulation in VR to understand percolation transitions in ET dynamics among a cytochrome complex. To make this process interactive we internalize the KMC mechanics and utilize a separate thread to populate a queue of *Events* that can be used to animate the simulation. We define an *Event* to be any feasible transition from one occupation state to another for the cytochrome’s heme sites.

As demonstrated in the previous section, the first step of this type of workflow is data acquisition. Depending on the data of interest, a parser can be written or an object exported to a 3D model from VMD into a geometry definition file in the Wavefront OBJ format. Our previous work details the specifics of this process for VMD [ref]. Once the import process is complete, the resulting object can be imported directly into the Unity game engine. IF one wishes to import their object into Unreal Engine, they will need to convert the Wavefront file to the FBX format, via their 3D modeling software of choice (Autodesk, Maya, Blender, etc.). To illustrate this process for the current application, we will use Unity ### on a PC running Windows 10.

We first create a new project, selecting the 3D option. To import the .obj file, we access the ‘Assets’ menu, then select the ‘Import New Asset’ option. Here, asset refers to any digital content usable by Unity and manipulable by the user. Although you have the choice between using JavaScript or C# to run a simulation, we develop in C# as it is a more powerful language, capable of tasks like multi-threading, passing by reference, as well as a host of other options. Furthermore, due to its syntactic similarity with the C and C++ programming language, translating previous simulation code to Unity via C# is much simpler than translating via JavaScript. Once the simulation process of interest has been coded, the update mechanics for each time step must be addressed. We have developed two frameworks on which to run and render a simulation in Unity. The first is to immediately render the results of each time step for the user using the public void Update() function, which Unity calls on each frame. The other is to have the simulation populate an *Event Queue* on each time step. Another class, responsible for rendering, can then pop events off this queue at an appropriate speed, designated by the user.

The first approach, which we denote as Run-and-Render, can be implemented in a few steps using the Start() and Update() methods of the GameObject or class responsible for handling the simulation. We start by creating a member variable, *DeltaTime*, to store the length of a time step in the simulation. Then, in the Start() method (which is placed by default in all Unity classes), all the positions of the objects that represent participants in the simulation are initialized. In our case, we cloned a sphere prefab and stored references to its clones in an array. Next, in the Update() method, we assign the *DeltaTime* variable to the current Unity-based time step (*Time.deltaTime*). This value can then be scaled by an arbitrary tuning factor to account for realistic observation capabilities of a human observer. Otherwise, since Update() gets called once per frame, the simulation may animate much more quickly than the user can beneficially observe. We also choose this assignment of *DeltaTime* so that the simulation executes proportionally to the real time that has passed for the user. Next, we add a call to the single time step method of the simulation inside the function, Update(). After the single time step, we iterate through each participant and update their status.

The second approach, Render-when-Ready, utilizes the multi-threading optimization mentioned in section 2.4 and provides the user more flexibility within the simulation. We begin by defining a new class, called *Event*, to encapsulate all the changes that occur within the system and associate them with each time step. This class can be anything from a collection of coordinates to the occupancy of certain states. Then, we create a member variable queue in either the class responsible for simulating or the one for rendering. The class this queue belongs to does not affect the output as long as both the simulation and rendering class have access to it. Next, in the simulation class, we create a method that will push to the event queue. Subsequently, in the rendering class, we create a method that will pop a single event off the queue. Finally, we spawn a new thread and pass in the simulation class’ queue-pushing method. Because this method is independent of the Update function and stores data independently of the visualization simulation, it is better suited to future applications involving more extreme, calculation intensive time steps.

Using control schemes similar to those outlined in the Methods section, the simulation can be manipulated or varied in real-time. The *RTSLeap* script can handle user calls to move, scale, and rotate the object of interest. As mentioned previously, however, Leap Motion provides a large library of hand-detectors to add more functionality to user interaction with the simulation.

## Integration of Existing Libraries and Simulators

In an effort to make our immersive scientific computing suite accessible to all researchers, ideally, we need to allow users to employ existing programs without requiring additional coding. One means of providing this functionality would be to include widely-used simulators as part of our VR suite. This process can be generalized by defining a procedure for compiling powerful simulators as libraries and providing a VR interface for these libraries. We developed a solution based on these principles for the large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) [39].

LAMMPS, developed by Sandia National Laboratories, is a more sophisticated version of the MD program discussed in section 7.4.1. It is capable of using the Message Passing Interface (MPI) for parallel processing and employs spatial-decomposition techniques to allow for sizable simulations to be conducted on PCs (it is not restricted to HPCC implementations) [39]. To provide native integration for this C++ program, we utilize the Unreal engine.

Once achieved, need to improve Simulation Size

We these simulation sizes running in real-time simulation, post-processing becomes trivial, greatly easing the burden on researchers seeking to understand systems and disseminate results.

Now that we have ability to import existing results and simulations. We should advance these things by providing new tools. An example is Virtual Confocal Microscopy

## Enhanced Visualization Toolbox

The previous sections are intended to embolden researchers to adapt existing scientific programs for use in VR, but, for our purposes, it is not enough to simply provide interoperability. To fully realize the promise of immersive scientific computing, we need to develop new tools to enhance these simulations and visualizations. An example of one of these tools is our ongoing development of a rendering method we call *virtual confocal microscopy*.

Confocal microscopy has become an indispensable biomedical tool, but it has proven useful in many other fields as well [33–35, 36]. It allows for increased resolution, contrast, and optical sectioning while minimizing photodamage and bleaching concerns. The technique uses point illumination and a pinhole to scan the sample and reject out-of-focus light. The resulting image, therefore, only contains information very close to the focal plane, offering greater resolution [35]. An analogous approach in VR could enable more methodical investigations of simulated complex structures.

Recently, investigations by Wei et al. into desalination membranes sought to understand and characterize how local, atomic structures contribute to macroscopic behavior [37]. The relevant dimensions of the polyamide membranes of interest range from a tenth of a nanometer to 100 nm, making it an excellent candidate for computational studies to extract optimization information not easily accessible experimentally. Wei et al. found that water molecules permeated the membrane through benzene rings and that the degree to which these monomers were cross-linked governed the speed of permeation along various paths [37]. However, the construction of these membranes is very complex, making a quick identification of preferential paths difficult, so it would be highly advantageous to be able to section this complicated super-structure to identify prevalent substructures of interest in these simulated materials.

To expedite this optimization process, we developed a virtual confocal microscopy mode for our immersive scientific computing suite that mimics the noninvasive optical sectioning possible in confocal microscopy. Utilizing Unity’s surface shader capabilities, we control how each vertex on the structure is rendered to highlight certain areas or planes of the material in the simulation [38]. Our solution seeks to generate a viewing plane that sits in front of the user’s head, follows their head movement, and always maintains a set distance from the user. This distance, as well as the thickness of the highlighted viewing plane can be specified and changed dynamically by the user. The rest of the simulated membrane will stay mostly transparent (with the opacity also dictated by user input), except for the vertices that intersect with this plane. In this way, the user can scan through the complicated polyamide membrane using just their head movements – a unique experimental advantage only possible in VR. More information on our shader and the associated code is included in the Appendix.

# Discussion

VR has become a powerful new method of engaging with users in a more tactile, visceral way. The resurgence in hardware to support VR has resulted in a rich software environment that allows developers access to these tools with minimal investment. We describe a workflow for rapid prototyping in the Unity game engine for the Oculus Rift. This particular game engine was selected to allow us access to C#, graphic shaders, and native VR support that spans multiple platforms, including mobile VR.

While these devices have a myriad of applications, they could be particularly useful for data visualization. They are orders of magnitude less expensive than traditional VR stations, allow for collaboration through augmented reality simulations, and provide new, unique control sets like head tracking [12–14].

Platform for rapid adoption of VR-based data visualization. Integration of LAMMPS.

Eventually we plan to bring in Script builders to limit need for previous knowledge

Use as a means to HPCC steering

We demonstrate the usefulness of game engines in scientific research such as language (C++, Java, C#) and abilities like multi-threading, shaders, and VR support. Refine integration of LAMMPS.

As the availability of hardware for augmented reality (AR) applications grows, this foundational work will grow in importance.

In future projects, for example, we hope to take advantage of the two controllers and Lighthouse tracking system included with the Vive HMD to allow room-based data exploration [12].

To support result dissemination and collaborative experimentation we seek to expand to Multiple-Users (plugging in both Rift and Vive – leader follower – network progress)

Future work:

* Grand Canonical Ensemble is impossible with our current framework
* LAMMPS animation has to be setup with a specific input script. Anders found a way to rerun any simulation in atomify using LAMMPS fixes. That would be a good thing for the future.
* Figure out how to optimize our multi GPU system better
* More ineteraction and simulation steering
  + Speed up/ slow down
  + On screen menus/charts
  + Dynamically load input scripts, as opposed to recompiling every time

Acknowledgements

The authors are grateful for insightful discussions on multiheme cytochromes with Dr. Liang Shi (Pacific Northwest National Lab) and Dr. Tom Clarke (University of East Anglia). KMC simulations by HSB at the University of Southern California were supported by the U.S. Department of Energy, Office of Science, Office of Basic Energy Sciences, Division of Chemical Sciences, Geosciences, and Biosciences Grant DE-FG02-13ER16415. Additional work on visualization is supported by Air Force Office of Scientific Research Grant FA9550-14-1-0294 to ME-N. The work at Lamar University was supported by the Research Enhancement Grants (REG) fund from Lamar University.

References

Appendix

1. Compiling LAMMPS into a DLL for UnrealGEARS

|  |
| --- |
| 1. Download the Linux Subsystem for Windows (“Bash on Ubuntu on Windows”) 2. Be sure to install g++, mingw-w64, and git, if not already done  * sudo apt-get install g++ * sudo apt-get install mingw-w64 * sudo apt-get install git  1. Clone the lammps repository from GitHub, and enter the source directory  * git clone <https://github.com/lammps/lammps> * cd lammps/src/  1. Copy MPI STUBS into the source directory (these aren't used, but are needed for compilation)  * cp ./STUBS/mpi.h ./mpi.h * cp ./STUBS/mpi.c ./mpi.cpp  1. Edit the Makefile for mingw64:  * lammps/src/MAKE/MACHINES/Makefile.mingw64-cross  1. Remove unnecessary compilation flags   -lmpi\_mingw64 (under MPI\_LIB) -ljpeg -lpng -lz (under JPG\_LIB) -lwsock32 -lquadmath (under LIB)   1. Add these new compilation flags   -static-libgcc -static-libstdc++ -mwindows (under SHLIBFLAGS)   1. To compile, call this from the source directory:  * make mingw64-cross mode=shlib  1. Move the compiled library file to a location accessible by Unreal Engine:  * mv liblammps\_mingw63-cross.so /mnt/c/LammpsDll/lammps.dll  1. To test if LAMMPS actually works on your computer, you can make an executable instead of a linked library, then just run it via the Windows command prompt. Just remove the mode=shlib argument in the compilation step to create the executable. Then when moving it to a Windows 10 accessible location, add the .exe extension to the file. |

1. Getting Started In VR

Having an instance of the game engines output to a VR HMD requires virtually no extra work from preparing simulation data itself. Unity 5 automatically enables VR display tracking once the “Virtual Reality Supported” box is checked in the *Inspector* window under *Edit > Project Settings > Player > Other Settings*. Once that box is checked, there just needs to be a camera object in the scene for the display to be rendered to. For Unreal Engine 4.16, VR enabling can be done straight from the editor by selecting the VR Preview option when playing the game. If you’ve packaged a project into an executable, VR display is enabled via console commands that can be compiled with the game, or with a command line flag (-vr) when the game is executed.

1. Control Schemes

Although it is evidently a useful tool for data visualization in scientific computing, simply viewing data in VR is just one facet of the true utility and power that comes with these game engines. The rich control options already available in the engines yield a plethora of ways in which one can interact with their data. These options include, but are not limited to, Xbox Controllers, LeapMotion, or even the standard Keyboard & Mouse. One particular category that takes heavy advantage of intuitive and immersive gesture controls are motion controllers. For the sake of our demos in Appendix D, we will restrict our control schemes to the LeapMotion controller for UnityGEARS and the Oculus Touch or Vive Controller for UnrealGEARS. The intuitive pinch and grab mechanics of the LeapMotion fit the needs of the data viewers in UnityGEARS, and the dedicated button layout on the Oculus and Vive controllers allows for tighter control over live simulations we demonstrate in UnrealGEARS.

1. LeapMotion in UnityGEARS

As stated earlier, Unity will automatically display the in-game camera’s output to the user’s HMD when VR support is enabled. However, if one wishes to use the LeapMotion, the method we chose was to replace the in-game camera with the LeapMotion library’s provided prefab, *LMHeadMountedRig*. To acquire this prefab, one needs to download and import the *Unity Core Assets 4.2.1* from the LeapMotion developer website. Other versions of the *Unity Core Assets* may be available, but the user may need to address certain compatibility issues with our demos. Additionally, one will need to mount the LeapMotion device onto the front of their VR HMD. LeapMotion provides a mounting kit for this purpose.

Using the *LMHeadMountedRig* should only cause a set of virtual hands to appear in the engine editor and game. From there, the user is free to program the controls how they see fit using the Detection Utilities found in the *Unity Core Assets 4.2.1* library. For our demo we also use the *Detection Examples 1.0.4* package, provided by the LeapMotion company, to allow for simple interaction with our precomputed simulation data. Specifically, we make use of the LeapRTS.cs script to allow us to grab, rotate, and scale our simulation data. At the time of publication, the *Detection Examples 1.0.*4 package has been deprecated by the company, however, we will still provide the LeapRTS.cs script in our demo files.

To utilize the LeapRTS.cs script, one first needs to add pinch detectors into their scene. This can be done by attaching the PinchDetector.cs scripts to empty child GameObjects of CapsuleHand\_L and CapsuleHand\_R. In the Inspector windows, drag the appropriate CapsuleHand to its corresponding PinchDetector’s “Hand Model” variable. When the simulation data has been converted into an object file and dragged into the scene Hierarchy, attach the LeapRTS.cs script to that object. Assign the appropriate left and right pinch detectors created earlier to LeapRTS.cs’s PinchDetector A&B fields. Refer to Figure ## as a reference.

Once setup, the user should be able to grab and manipulate their simulation data in VR. As noted earlier that the current *Detection Examples 1.0.*4 package for the LeapMotion has now been deprecated at the time of publication. LeapMotion offers a new Interaction library to replace it. This goes to show how quickly these technologies are evolving, encouraging the need for scientists to keep up and use the rich software and hardware available to them.

1. Oculus Touch and Vive Controller in UnrealGEARS

As mentioned in Section 2.4 and 3.3, we moved much of our efforts to Unreal Engine 4 as a means to more simply support larger real-time simulations running via LAMMPS. Because our system sizes in LAMMPS were pushed up to contain up to 500,000 particles, grabbing and manipulating these running simulations with a LeapMotion detected hands was not as practical as with the static snapshots in Section 3.1. Instead of viewing simulation data by manipulating the system’s orientation, we chose to have the main camera move around the simulation via dedicated flight controls. Consequently, we moved from the LeapMotion hand detection controls to more robust motion controllers to allow for easier movement throughout the game environment. Furthermore, the inclusion of dedicated buttons on the motion controllers provide by Oculus and HTC allow for more deliberate steering over the results of a simulation.

Processing input from these motion controllers in Unreal Engine 4.16 is straightforward. The blueprint interface already provides dedicated event nodes that trigger for each button on the motion controllers. Figure ## shows these event nodes as well as the following blueprint logic to allow for game-specific needs. Aside from the dedicated event nodes, users can create their own input mappings via the Input section of the engine Project Settings.

1. How to use UnrealGEARS

Most interfacing with the UnrealGEARS system is done via the blueprint, BP\_LammpsController. To run a simulation, one needs to simply place an Actor of that type into the level. Then in the Details window, the user can adjust the input script name, toggle the animation mode, setup the timesteps of interest (only for animations), and manage particle types. [ask Kenichi how to properly document these next parts]

1. Demos (Code provided in GitHub Repository)
2. Static Viewing (LeapMotion required for interaction)

We first demonstrate the utility and ease-of-use of VR enabled game engines as preprocessed-data viewer. Timestep position data of many types of simulations can be written to various standard coordinate formats – in our case, .xyz. For this demo, we take a precomputed snapshot of a [insert simulation type], convert it to the Wavefront format via VMD, then drag it into our Unity scene. This process and workflow is detailed above in Section 3.1 and Figure 1a.

1. Virtual Confocal Microscopy Shader (LeapMotion required for some interaction)
2. Real-time in-engine Simulation

Here we provide two examples of real-time simulations being computed and rendered by the engine itself on each timestep. The first is a simple molecular dynamics simulation using a Lennard Jones potential on all particles. The code was almost directly copied from the C-coded simulation by Aiichiro Nakano (insert link to code). The only additions made were in initialization, when particles needed to be rendered, and then in the update step when their positions onscreen need to be adjusted. Both the timestep and position updating step are done on each frame (once per Update() call).

Next we demonstrate Unity’s ability to implement our multithreading performance optimization technique described in Section 2.4. This is done via a kinetic Monte Carlo simulation of biological electron transfer in cytochrome proteins. We use C#’s multithreading library to run our timestep calculations. Each transfer event get pushed to a queue, which our main game thread then pop from to update the visualized heme states.

To watch, just load up the scene and press play.

1. LAMMPS Animation via DUMP files (Oculus or Vive Motion Controller for mobility)

An interesting and incredibly useful capability of LAMMPS is that it can read through old coordinate files and essentially rerun a simulation frame by frame. By setting up a LAMMPS system devoid of forces and interaction, we are able to recreate a simulation that Unreal Engine 4.16 can then render. This eliminates the need for us to write our own file parser.

1. Real-time LAMMPS Simulation (Oculus or Vive Motion Controller for mobility)