VR JuggLua: A Framework for VR Applications Combining Lua, OpenSceneGraph, and VR Juggler

Ryan A. Pavlik*
Human-Computer Interaction
Graduate Program
Iowa State University

Judy M. Vance†
Virtual Reality Applications Center
Iowa State University

ABSTRACT

A gap exists between virtual reality (VR) software platforms designed for optimum hardware abstraction and cluster support, and those designed for efficient content authoring and exploration of interaction techniques through prototyping. This paper describes VR JuggLua, a high-level virtual reality application framework based on combining Lua, a dynamic, interpreted language designed for embedding and extension, with VR Juggler and OpenSceneGraph. This work allows fully-featured immersive applications to be written entirely in Lua, and also supports the embedding of the Lua engine in C++ applications. Like native C++ VR Juggler applications, VR JuggLua-based applications run successfully on systems ranging from a single desktop machine to a 49-node cluster. The osgLua introspection-based bindings facilitate scene-graph manipulation from Lua code, while bindings created using the Luabind template meta-programming library connect VR Juggler functionality. A thread-safe run buffer allows new Lua code to be passed to the interpreter during run time, supporting interactive creation of scene-graph structures. It has been successfully used in an immersive application implementing two different navigation techniques entirely in Lua and a physically-based virtual assembly simulation where C++ code handles physics computations and Lua code handles all display and configuration.

Keywords: Virtual reality, software tools, human-computer interaction, C++, Lua, interactivity.

Index Terms: H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems—Artificial, augmented and virtual realities; C.2.4 [Computer-Communication Networks]: Distributed Systems—Distributed applications

1 INTRODUCTION

*e-mail: rpavlik@iastate.edu
†e-mail: jmvance@iastate.edu

R.A. Pavlik and J.M. Vance, "VR JuggLua: A Framework for VR Applications Combining Lua, OpenSceneGraph, and VR Juggler," **Workshop on Software Engineering and Architectures for Real-time Interactive Systems (SEARIS) in IEEE Virtual Reality**, Singapore: 2011. doi:10.1109/SEARIS.2012.6231166

©2012 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other users, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works for resale or redistribution to servers or lists, or reuse of any copyrighted components of this work in other works.

A variety of software frameworks for creating interactive virtual reality (VR) applications exist. Each framework provides some subset of the following features: operating system portability layer, input device abstraction, display view-port configuration, VR system simulation, cluster support, three-dimensional (3D) scene data structures, event system, and scripting. Frameworks that emphasize the systems level provide little or no higher-level content authoring support. Conversely, frameworks that explore the experience of content creation generally fall short in system independence and compatibility with complex or high-end virtual reality systems. This gap limits the ability of experience designers and researchers to both develop real-time interactive environments using high-level constructs and run their environments on a broad range of VR computing systems.

This research builds upon existing mature software to produce a framework supporting rapid development and iteration of virtual environments (VEs) with the potential to run on the broadest possible range of VR systems. VR Juggler was selected as the basis for this research. The VR Juggler open source virtual reality software platform [8] supports a broad range of VR systems, including a 49-node cluster that poses a difficult challenge for other systems, and supports Windows, Mac, and Linux. The dynamically-typed Lua programming language [17, 16] was selected for integration both as a scripting language supporting C++ applications and as a fully-capable language for building standalone immersive applications. Lua's clear and minimal syntax, ease-of-use for end-user programmers, and ease of interoperability with C++ supported this selection.

The VR JuggLua framework supports the same range of VR systems as its VR Juggler core. It uses the OpenSceneGraph[1] graphics library to provide scene organization, model loading, and rendering support. Using VR JuggLua, VR applications can be written entirely in Lua, building from the base level of the VR Juggler kernel frame loop. Paradigms for interactive application design can be rapidly implemented in Lua and used to provide a higher level application base, which is a current area of research. Also, applications can be written using the VR JuggLua C++ API, applying the Lua scripting engine in any capacity from simple configuration to management of all audio-visual output.

The rest of this paper is structured as follows: Section 2 discusses background and related work. Section 3 discusses the specifics of the VR JuggLua design and implementation, including use and extension of existing software, as well as the potential of this system to streamline VE implementation and support research into paradigms for immersive interaction. Section 4 describes some applications built on VR JuggLua that showcase different aspects of its feature set, and Section 5 presents conclusions and future work.

[1] `http://www.openscenegraph.org/`

2 BACKGROUND

A wide variety of software frameworks for building virtual reality applications have been developed. The CAVE Library initially developed for use with the CAVE Automated Virtual Environment [9] is an example of early work in the systems category of virtual reality frameworks. It has evolved into a commercial solution integrating clustering support and focusing on multi-screen application development. VR Juggler introduced a highly modular architecture for VR applications to provide a "virtual platform" for development and execution on diverse systems [8, 7]. Later development extended its use from high-end graphics systems to commodity computer clusters [1, 7]. The FlowVR platform was developed based on experience in using VR Juggler in a clustered environment, and emphasizes a data-flow model for distributed real-time interactive computation with high modularity [3, 2]. The Syzygy system presents multiple frameworks for VR application development, and was developed with an explicit focus on clustered execution [21].

Other frameworks focus more on the content authoring experience, often integrating an interpreted scripting language for rapid development. Colosseum3D integrates OpenSceneGraph, physics capabilities, and audio rendering, and combines the use of C++, a custom object-description format, and Lua scripting [5]. Colosseum3D generates bindings of its C++ classes using the tolua++ utility. The commercial VR authoring environment Virtools[2] integrates a custom scripting language, VSL, for content creation. AVANGO/NG applies a generic field and field container programming interface to a scenegraph based on OpenSceneGraph, with Python scripting support [18].

A programming model more closely linked to the use of an interpreted language has also found success in creating several varieties of immersive interactive experiences. WorldViz Vizard[3] is a commercial application framework, using the Python language with a custom integrated development environment (IDE) to create experiences rendered using OpenSceneGraph. However, it has limited clustering support when compared to some of the systems-focused frameworks designed explicitly for distributed execution. TINT is an augmented reality (AR) and mixed reality (MR) framework designed to present a pure Python programming interface, with optional interaction with C++ modules compiled for improved performance [10]. By delegating computationally-intensive tasks to compiled code, the bulk of applications can be written using Python for development efficiency and still achieve interactive performance. The HECTOR platform takes a similar approach integrating compiled code and interpreted Python, with an event-driven architecture for virtual reality applications [25].

3 SYSTEM DESIGN

This section explores the implementation of the VR JuggLua framework starting from the foundation of existing software and extending and continuing toward higher levels of the platform. Section 3.1 discusses the base levels of existing software used in this framework, while Section 3.2 addresses integrating these systems and presents a coherent, logical interface for application development. Section 3.3 presents the potential for developing increasingly clear and interactive virtual experiences using VR JuggLua.

[2]http://www.virtools.com/

[3]http://www.worldviz.com/products/vizard/

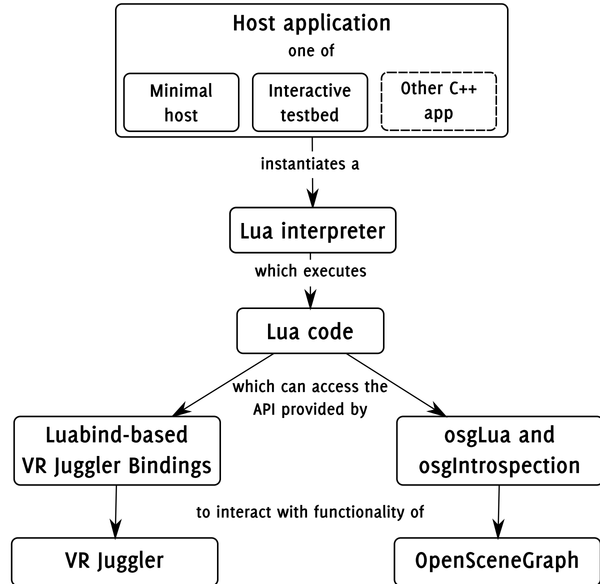

Figure 1: System diagram

As a full framework, VR JuggLua encompasses its foundational software, bindings for this software to Lua, the Lua interpreter library itself, and basic host applications. A typical application will have only one Lua interpreter state with access to all bindings included in VR JuggLua. A VR JuggLua application uses both the osgLua module and the VR Juggler bindings included in the VR JuggLua framework to access a complete set of virtual reality functionality from Lua (Figure 1).

3.1 Foundational Software

The VR Juggler software framework is a "virtual platform" for development of VR software that can be used on a wide variety of VR computing systems [8]. It consists of several components that together allow virtual reality applications to be written in C++ and executed using various hardware configurations. VR Juggler Portable Runtime (VPR) is the cross-platform portability library supporting access to operating system functionality like threading, networking, and serial input/output: capabilities primarily used by the other VR Juggler components rather than directly by framework client applications. Configuration of all components is handled by the Juggler Configuration and Control Library (JCCL), which provides a structured method of processing XML configuration files. The Gadgeteer component provides input device abstraction and dynamically-loaded hardware support, as well as input sharing and application-specific data sharing in cluster environments. Its uniform interfaces for device access are directly utilized by any interactive application built on VR Juggler. Sonix, an optional component, provides an abstraction layer over basic immersive audio capabilities. Its interface is simplistic, and can be configured along with the rest of the VR Juggler suite, so it is a suitable choice for providing entry-level audio playback in a virtual environment based on the VR Juggler system. Finally, the VR Juggler library provides display management and transfers control during specific periods of the frame loop to application objects. Application objects are the highest-level of content authoring interface pre-

sented by VR Juggler. Specializations of the base application object are included that support using scene-graph libraries, including OpenSceneGraph and OpenSG [20]. The VR Juggler kernel, however, is intentionally independent of any particular scene system, and can even support DirectX graphic rendering in addition to OpenGL and OpenGL-based scene-graphs.

3.1.1 Binding to Lua

The Lua language is a high-performance language designed for embedding and extension [17, 16]. The Lua language must always be tied to a host application. A minimal host application that presents a basic Read-Eval-Print loop (REPL) [12, 4] as well as script execution is sufficient, and one such application is included with the standard Lua implementation. Lua is coded in the platform-independent subset of C that is also valid C++, and thus presents a C-centric application programming interface (API). The latest stable release of Lua, version 5.1.4, is included in the VR JuggLua source tree and built into a static library during the software build. It is compiled as C++ to support exceptions at all levels of the software stack.

On top of Lua, the Luabind[4] library provides an intuitive method of wrapping C++ classes, methods, and functions for access from Lua. It uses template meta-programming techniques to generate appropriate Lua C API calls for binding at compile-time, which allows it to automatically deduce function signatures in most cases. This distinguishes Luabind from some other wrapper generator systems, such as SWIG [6] and tolua++[5], which require a separate step before compilation to process either unmodified header files or interface files resembling headers. VR JuggLua uses Luabind to create bindings to VR Juggler components. These bindings function like any other Lua module, extending the functionality of any interpreter state in which they are loaded.

3.1.2 OpenSceneGraph and osgLua

OpenSceneGraph (OSG) was selected as the graphics subsystem of VR JuggLua. It is a mature scene-graph, supported in VR Juggler, with good interoperability across platforms and import plug-ins for a wide variety of image and model file formats. Importantly, there exists reasonably up-to-date bindings for OpenSceneGraph to Lua, in a package called osgLua[6]. Rather than manually creating bindings for all of OpenSceneGraph, or preprocessing the OSG headers, osgLua uses the osgIntrospection library to provide access to nearly all OSG classes. As a part of OSG 2.8.x, osgIntrospection loads wrapper dynamic libraries generated automatically from the source, and allows reflection, type instantiation, property access, and method calling generically with arbitrary OSG data structures. By dealing only with osgIntrospection types, values, and methods, rather than statically binding to specific types and methods, osgLua is able to avoid falling behind upstream OSG development and offer nearly complete coverage of the library's capabilities. Though public development of osgLua seems to have stalled in late 2007, this introspection-based approach allows it to work fully on the latest stable OpenSceneGraph 2.8.3 with only minor updates.

Improvements to osgLua were made while developing the VR JuggLua software framework. The use of NodeVisitors and `enum` data-types from Lua was fixed, improving the amount of OSG functionality usable from Lua. Direct access to object properties without using set/get functions has also been added, providing a more natural and Lua-like interface. Work on VR JuggLua also extended the introspection-based binding with methods that recognize the vector and matrix data-types, selectively defining Lua metatable methods for these values to allow the direct use of the normal math and comparison operators in Lua code.

[4] `http://www.rasterbar.com/products/luabind.html`

[5] `http://www.codenix.com/~tolua/`

[6] `http://svn.pplux.com/lab/osgLua/`

3.1.3 Connecting osgLua and Luabind

The distinct representations of Luabind-wrapped objects and the osgLua objects presented a challenge for VR JuggLua implementation. A key insight is that once osgLua is loaded, OSG types can effectively be considered "native types" in Lua. Luabind provides a template-based system allowing seamless conversion between C++ string types and Lua strings, C integer and floating point types and Lua numbers, and so on. Luabind has a public `native_converter_base` interface to allow developers to provide similar converters for their own specialized classes wrapping these basic data-types.

OpenSceneGraph types can be divided into two groups: reference types, which are always allocated on the heap and passed by pointer, and value types, which may be allocated on the stack. Templated subclasses of the Luabind native converter template base class were made to handle these two categories of datatypes. This approach results in the need for only a few-line class to specify the type name for each OSG type that is involved in the Luabind-wrapped method. C preprocessor macros were employed to reduce this to a single line per OpenSceneGraph type. When VR JuggLua is compiled, it invokes the macros for the common OSG types that it uses. If a client application written in C++ wishes to bind functions to Lua and requires support for additional OSG types in the binding, the header can be included and the preprocessor macros can be invoked for any available type. This solution allows OSG types to be passed seamlessly between Lua and Luabind-bound C++ code.

3.2 Programming Interface

The approach taken to binding the VR Juggler components to Lua was to keep the interface simple and allow the most common use cases to be written entirely in Lua. From the application's point of view, interaction with the VR Juggler kernel is limited to specifying the jconf configuration files, and starting and stopping the application thread. In the C++ API, all kernel interactions take place with the singleton [13] instance of the kernel. In the Lua binding, then, the singleton kernel instance is implied, and small free functions were bound that look up the singleton pointer and call the method.

Access to device input takes place through a variety of Gadgeteer device interface classes. These classes were bound one-to-one, but with slight modifications. The need to separately call an `init` function with the name of the device alias, mandated in C++ by the smart-pointer pattern implemented by these device interfaces, was eliminated in favor of a parameter to the constructor. Getter methods are used in the C++ interface, while in Lua, the input device data can be easily presented as directly-accessible properties. VR Juggler uses the GMTL matrix and vector math template library[7] that, while suiting the purposes of VR Juggler applications without a scene-graph system, does not directly inter-operate with the equivalent types in OpenSceneGraph. The Lua binding offers the opportunity to standardize on the OpenSceneGraph types,

[7] `http://ggt.sourceforge.net/`

so positions and transforms are accessible as OSG vector and matrix types, using meters as the units.

To provide a fully-featured VR software framework, VR JuggLua also includes Lua bindings to the main Sonix data types. As with the Gadgeteer device interfaces, the Lua binding exposes read-only or read-write properties instead of getter/setter methods where feasible. Sounds can be configured either externally in a jconf configuration file, or at run-time in Lua code, and triggered by Lua code when applicable. The ability to keep sound triggering code in Lua improves the clarity of C++ simulation code by separation of concerns [15].

3.2.1 Creating Application Objects in Lua

To complete the binding of VR Juggler to Lua, a method for creating application objects, the basic unit of the VR application, was needed. Application objects implement a C++ interface specifying action to take during initialization and each of the steps in the kernel frame loop: `preFrame`, `latePreFrame`, `draw`, `intraFrame`, and `postFrame`. In applications based on VR Juggler and OpenSceneGraph, the `osgApp` specialization of the application object interface contains an implementation of the draw method to render the scene-graph. Most application logic is called during the `preFrame` or `latePreFrame` stages, which can update the scene-graph based on newly-received input device data.

To allow an application to be written entirely in Lua, an implementation of the `osgApp` interface was needed. To allow kernel calls to application object methods to invoke Lua functions, an application object proxy class was created, using a synthesis of the proxy and delegation design patterns [13]. The proxy class derives from the `osgApp` class. Lua code can instantiate this application proxy and pass it a Lua table datastructure, which serves as an application object delegate If this table has function elements named matching the application object interface, the application proxy will call those functions during the appropriate phase of the kernel frame loop. Defining an application object this way is an application of latent or "duck typing" [8] as popularized by the Python programming language [11]. If a Lua table has methods that an application object would have, it can be considered an application object, without requiring a particular type.

Luabind does permit binding of classes with virtual methods and the subclassing of those classes entirely in Lua, so a strict typing approach to creating Lua application objects is possible. However, the application proxy object approach taken in VR JuggLua has several advantages over direct subclassing in Lua. For instance, the application proxy object can perform some error checking. If an application delegate has not been set by the time the kernel requests application object and scene initialization, a useful error message can be produced and execution can be stopped. Similarly, if a delegate has been set, but no forwarded calls have succeeded in an entire frame loop, the application proxy can assume that a logic error has occurred and stop execution. The application proxy layer also allows simplifying standards to be implemented. For example, despite display configuration taking place in meters, the default projection with VR Juggler produces a foot unit-based scaling. As VR JuggLua standardizes on meters for positional device data, the application proxy creates a root scaling transform node to produce an apparently meter-based display setup for VR JuggLua applications.

[8]Originating in a quotation attributed to nineteenth-century poet James Whitcomb Riley: "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

3.2.2 Lua Run Buffer

The elements already discussed are sufficient to create a virtual reality application entirely in Lua, with rapid development iteration due to Lua's interpreted nature. However, the "don't call us, we'll call you" design of VR Juggler [14] effectively removes the interactivity produced by the basic Lua interactive interpreter. Once the kernel is started, any attempt to interpret additional new code will likely result in concurrent threading problems as the kernel thread and the initial thread both attempt to interact with the same Lua interpreter state simultaneously.

In an implementation based on the VR Juggler kernel loop, idle operation consists of a continuous event loop, rather than a blocking input call as found in a command-line REPL-type application. As such, a separate user interface (UI) thread is the simplest way to implement the "read" portion of a REPL independent from the idle loop. The UI can effectively block waiting for the user's input of code to evaluate. The evaluation of a completed code subsection, however, must take place in the idle event loop for simplicity of design and use of a single interpreter state. The `preFrame` or `latePreFrame` step of the kernel loop is the most logical place to evaluate new code, since the application state at that point corresponds neatly to a mental model of interactive execution: accessing device interfaces will return the most recent data, and changes to the graphical state are possible and will be immediately reflected in the display in the subsequent `draw` step.

Based on these concepts, VR JuggLua includes a thread-safe run buffer system supporting interactive code execution during application runtime, illustrated in Figure 2. Code can be added to this circular buffer at any time, and a single method call on the buffer runs all contents, in order. This run buffer method call is bound to Lua and can be placed in the application delegate function corresponding to the `preFrame` or `latePreFrame` states. An interface for an interactive GUI console, with text-based stub, FLTK[9], and Qt[10] implementations, exposes this functionality to a user. Any VR JuggLua-based application can use this GUI console as a drop-in component, supporting code entry, display of print output from Lua, and loading and saving of script files.

3.3 High-Level Potential

Binding of VR Juggler and associated subsystems to allow full VR application development has benefits beyond allowing applications to be written in Lua equivalent to comparable C++ code. Exploration of the possibilities produces a range of future research topics. In this section, two of the unique possibilities afforded by the combination of Lua and VR Juggler will be discussed. The Lua language supports syntactic sugar designed for intuitive data description that can be used to provide a higher-level interface to C++ functionality. Furthermore, the Lua run buffer and GUI console components are building blocks for a fully-interactive virtual reality REPL-type code execution environment.

3.3.1 Lua Syntactic Sugar for Application Description

Lua allows rapid development of experience authoring techniques, primarily due to its concise "constructor" syntax and table data structures. The osgLua library provides a fairly direct translation of the C++ API of OpenSceneGraph to Lua. While this approach allows access to the full potential of the

[9]`http://www.fltk.org/`

[10]`http://qt.nokia.com/`

library, it can make common tasks repetitive and unclear. For instance, using pure osgLua syntax, the following code would be used to load a model, attach it to a transform, and attach this transform to a root scene-graph node.

```
teapot = osgLua.loadObjectFile("teapot.osg")
transform = osg.PositionAttitudeTransform()
transform:setPosition(osg.Vec3(1.0, 0.0, 0.0))
transform:addChild(teapot)
root:addChild(transform)
```

Lua allows tables, which are a data structure like associative arrays, to be created in-line with `{}`, and function calls passing a single table argument may be made simpler by abbreviating `functionCall({data, data})` as `functionCall{data, data}`, which is known as the constructor syntax. Clearly-named functions designed for constructor syntax can replace the scene-graph creation code listed above with this simpler, yet equivalent alternative:

```
root:addChild(
    Transform{
        position = {1.0, 0.0, 0.0},
        Model("teapot.osg")
    }
)
```

Here, `Transform` is a function taking some named arguments specifying property values for a `PositionAttitudeTransform`, as well as any number of unnamed arguments corresponding to OSG nodes to add as children. It is being called with a position argument, as well as the results of a call to `Model`, a simple wrapper around `osgLua.loadObjectFile` to load a given file and report an error if the load is not successful. The `PositionAttitudeTransform` node created and returned by `Transform` is passed directly to the original C++-style `addChild` call to connect it to the scene-graph root. This alternate syntax more clearly indicates the values assigned to node properties, and also directly conveys the nesting of the model node within the transform node, an important aspect of the scene's organization that might be missed in the more procedural original code.

3.3.2 Interactive Testbed Application

Applying the run buffer and GUI console, an interactive virtual reality REPL was created. To serve as a testbed for scene creation and manipulation, all details of setting up a VR Juggler application are handled behind the scenes. A minimal Lua application object provides navigation capability and runs the code accumulated in the run buffer. An empty scene and console are presented on startup, and user code is executed interactively and apparently immediately. (A delay of at most a frame-length does occur due to the asynchronized nature of the GUI console and the kernel frame loop, though this is imperceptible.) This interactive console allows learning of syntax to proceed more rapidly than the save-compile-run cycle of C++ or even the save-run cycle of a bare Lua VR JuggLua application. Lua errors are presented immediately in the GUI console, and by default do not halt the execution of the application. The user is thus encouraged to try the code again, with modifications as errors would point out. In anecdotal experience, the console serves well to localize errors in longer, more complex virtual environments: if the full script does not produce the desired results, users quickly learn to try pasting code incrementally. In effect, the debugging behavior of stepping through problem code arises spontaneously as a user works with the environment. The testbed application does not impose any specific structure on application code developed interactively. Executed code, interspersed with text output formatted as Lua comments, is logged and available for saving to a script file or copying and pasting into a text editor.

Figure 3 shows the testbed application running on Windows 7, on a desktop system in simulator mode. VR Juggler simulator mode allows keyboard and mouse inputs to be translated into immersive device inputs, such as head and wand position tracking and wand button presses. Though simulator mode loads by default, configuration files for an immersive VR system can also be loaded, allowing experimentation with virtual environment design to take place in the actual hardware system used for running completed applications. The GUI console can either float above the windows rendering the immersive display or be moved to an additional non-immersive display.

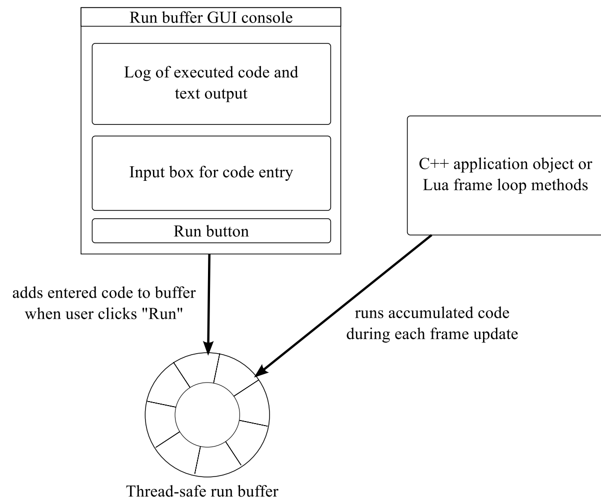

Figure 2: Thread-safe run buffer for interactive execution

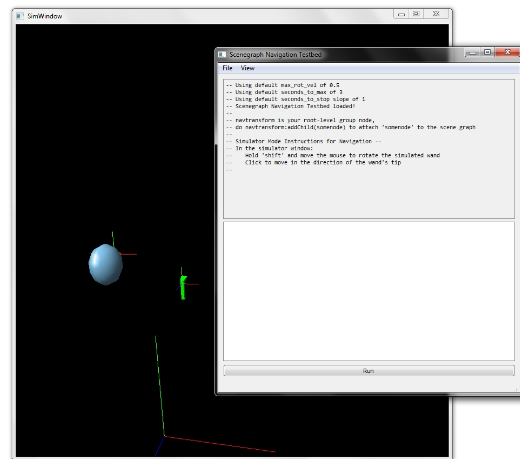

Figure 3: Interactive testbed application

4 EXAMPLES

Immersive applications have been successfully written using the VR JuggLua system, both in pure Lua and in a combination of C++ and Lua. This section will highlight a few samples of the results achieved using VR JuggLua and the aspects of the framework's design that they illustrate.

4.1 Learning Virtual Reality Interactively

The interactive testbed application was applied in an unstructured undergraduate learning environment which focused on concepts of scene-graphs and 3D virtual reality. A sample task of scene design was assigned, with the goal of prototyping a more sophisticated application. A reasonably-complex scene was built from multiple models, sourced internally as well as from the Google 3D Warehouse[11]. An iterative process on typical laptop and desktop computers was observed, with rapid iterations of the application script tested interactively using the testbed application. The script constructed in this way was then launched in a single-machine two-walled immersive environment for more thorough testing. It was ultimately demonstrated in the C6, a six-wall high-resolution CAVE-like system powered by a 49-node cluster. The application performed smoothly and as designed.

4.2 Testing Navigation Techniques

In the course of a summer program for undergraduates, a scenario was developed for testing navigation in a user study in the C6 environment. An application was written, entirely in Lua, by undergraduate and graduate students. The application loaded sophisticated models, and supported comparison two navigation techniques based on device input. The necessary transforms and manipulations to display the externally-sourced models were developed on desktop machines using the interactive testbed. The navigation techniques interpreted analog and positional data from sensors on an instrumented real object, to provide an on-screen registered virtual version of the object and to allow movement in a natural way. Logging of navigation data was implemented, and a successful user study was completed, in a limited time frame. High performance of the application was observed, despite the use of an interpreted language and very complex graphical model. This is made possible due to the delegation of graphics rendering to the C++-based OpenSceneGraph. Lua code traversed and modified models at load time and updated transforms during run time, but the actual rendering code in a VR JuggLua application remains part of OpenSceneGraph.

4.3 Integrating with C++ Simulations

Research into applications of virtual reality technology to manufacturing engineering led to the development of virtual assembly simulations with haptic force-feedback capability. The *Scriptable Platform for Advanced Research and Teaching in Assembly* (SPARTA) is the successor to the *System for Haptic Assembly and Realistic Prototyping* (SHARP) as developed by Seth *et al.*[22, 23, 24]. SPARTA is an application built on VR JuggLua in which C++ code performs physically-based simulation of interactions between part models, rendering corresponding haptic force feedback to haptic devices such as the PHANTOM Omni® by Sensable™ and the Virtuose™ 6D35-45 by Haption at a rate of 1000 Hz.

Classes in SPARTA representing the physics simulation, physical bodies, and manipulator devices are bound for Lua access by Luabind. Lua code executed using the GUI console and run buffer is used to configure interaction devices and techniques, load parts to interact with, and start the physics simulation. Lua scripts performing these tasks are used in place of configuration files, offering extended functionality for complex configurations and eliminating the task of writing a configuration file parser. Scripts are either loaded from the command line, or interactively using the GUI console, which has been included as a "drop-in" component and allows incremental development of SPARTA configurations akin to the incremental development of VR JuggLua applications enabled by the interactive testbed application. Figure 4 shows SPARTA running in simulator mode with the GUI console visible. Of course, like all VR JuggLua applications, SPARTA can be run in a fully-immersive mode without any code modifications.

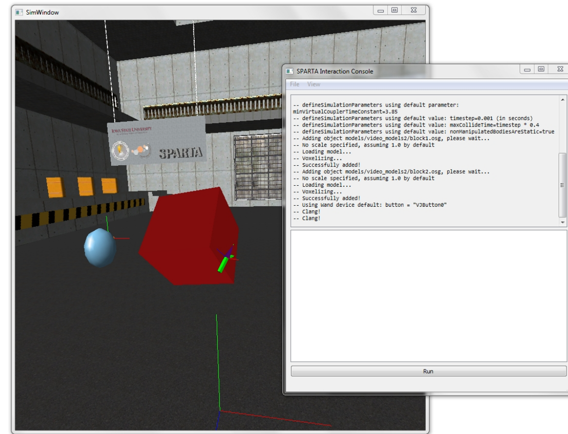

Figure 4: SPARTA in simulator mode showing Lua console

Furthermore, while C++ code performs the physics computations and high-rate simulation in SPARTA, the visual and audio feedback is written entirely in Lua. A simple application object delegate handles updating the positions and orientations of models in the scene-graph based on the current simulation state, and collision statistics are monitored by separate Lua code to trigger appropriate sounds using the Sonix binding in VR JuggLua.

The case of SPARTA illustrates a high-end application of VR JuggLua: it is a sophisticated application taking advantage of the VR JuggLua C++ API and binding its own internal objects to Lua. It uses Lua code to configure the C++ core, translate simulation state into visual and audio displays, and provide an extension point for investigating interaction techniques.

5 CONCLUSIONS AND FUTURE WORK

Through enabling applications to be written in Lua and run on the versatile VR Juggler framework, VR JuggLua provides a basis for exploring high-level design of virtual experiences that are rapidly developed, yet capable of running on even the most high-end virtual reality systems. The selection of Lua provides opportunities for clarifying syntactic improvements over C++, and the implementation of a thread-safe run buffer with an interactive GUI console enables interactive REPL-like development of virtual reality applications. Experience in using the framework has demonstrated its suitability for use in introductory virtual reality exploration, as well as in sophisti-

[11] `http://sketchup.google.com/3dwarehouse/`

cated physically-simulated interactive research applications.

Future work includes exploring the possibilities of using Lua to create interactive experiences. This work includes the use of Lua features such as co-routines [19] to support increasingly complex environments based on procedural, rather than frame-oriented, mental models of end-user programmers. Research into applying Lua's support of first-class functions (functions as values) for interaction and action selection is anticipated. Extending the run buffer across a clustered rendering environment to permit synchronized run-time code evaluation on high-end immersive systems will also be investigated.

ACKNOWLEDGEMENTS

Thanks to Dhanraj Selvaraj, Patrick Carlson, Meisha Rosenberg, Livien Yin, Terrence Scott-Cooper, Troy Lambert, and Carl Kirpes for using early versions of VR JuggLua in real-world applications and providing valuable feedback. The comments of the anonymous reviewers on an early version of this paper were also valuable and appreciated. This work was performed at the Virtual Reality Applications Center at Iowa State University, and was supported by NSF award CMMI-0928774.

REFERENCES

[1] J. Allard, V. Gouranton, L. Lecointre, E. Melin, and B. Raffin. Net Juggler: Running VR Juggler with Multiple Displays on a Commodity Component Cluster. In *IEEE Virtual Reality Conference (VR 2002)*, volume 2002, pages 273–274, Los Alamitos, CA, USA, 2002. IEEE Comput. Soc.

[2] J. Allard, J.-D. Lesage, and B. Raffin. Modularity for Large Virtual Reality Applications. *Presence: Teleoperators & Virtual Environments*, 19(2):142–161, Apr. 2010.

[3] J. Allard, C. Ménier, E. Boyer, and B. Raffin. Running Large VR Applications on a PC Cluster: the FlowVR Experience. In E. Kjems and R. Blach, editors, *Proceedings of the 9th Int. Workshop on Immersive Projection Technology, 11th Eurographics Workshop on Virtual Environments, IPT/EGVE 2005*, pages 59–68, Aalborg, Denmark, 2005. Eurographics Association.

[4] E. Allen, R. Cartwright, and B. Stoler. DrJava: a Lightweight Pedagogic Environment for Java. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education - SIGCSE '02*, volume 34, pages 137—-141, New York, New York, USA, Mar. 2002. ACM Press.

[5] A. Backman. Colosseum3D – Authoring Framework for Virtual Environments. In E. Kjems and R. Blach, editors, *Proceedings of the 9th Int. Workshop on Immersive Projection Technology, 11th Eurographics Workshop on Virtual Environments, IPT/EGVE 2005*, pages 225–226. Eurographics Association, 2005.

[6] D. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996-Volume 4*, page 15. USENIX Association, 1996.

[7] A. Bierbaum, P. Hartling, P. Morillo, and C. Cruz-Neira. Implementing Immersive Clustering with VR Juggler. In O. Gervasi, M. Gavrilova, V. Kumar, A. Laganà, H. Lee, Y. Mun, D. Taniar, and C. Tan, editors, *Computational Science and Its Applications - ICCSA 2005*, volume 3482 of *Lecture Notes in Computer Science*, pages 1119–1128. Springer Berlin / Heidelberg, 2005.

[8] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. VR Juggler: a virtual platform for virtual reality application development. In *IEEE Virtual Reality Conference (VR 2001)*, pages 89–96, Los Alamitos, CA, USA, 2001. IEEE Comput. Soc.

[9] C. Cruz-Neira, D. J. Sandin, and T. A. DeFanti. Surround-screen projection-based virtual reality: the design and implementation of the CAVE. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques - SIGGRAPH '93*, pages 135–142, New York, New York, USA, 1993. ACM Press.

[10] U. Eck and C. Sandor. TINT: Towards a Pure Python Augmented Reality Framework. In *SEARIS Workshop in IEEE Virtual Reality*, 2010.

[11] B. Eckel. Strong Typing vs. Strong Testing. In J. Spolsky, editor, *The Best Software Writing I*, pages 67–77. Apress, 2005.

[12] R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Programming Languages: Implementations, Logics, and Programs*, pages 369–388. Springer, 1997.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, volume 206. Addison-Wesley, Reading, MA, 1995.

[14] P. Hartling. VR Juggler: The Programmer Guide.

[15] W. L. Hürsch and C. V. Lopes. Separation of Concerns. Technical report, College of Computer Science, Northeastern University, 1995.

[16] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The evolution of Lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages - HOPL III*, pages 2–1—-2–26, New York, New York, USA, 2007. ACM Press.

[17] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho. Lua – An Extensible Extension Language. *Software: Practice and Experience*, 26(6):635–652, June 1996.

[18] R. Kuck, J. Wind, K. Riege, and M. Bogen. Improving the AVANGO VR/AR framework – Lessons learned. In *Workshop Virtuelle und Erweiterte Realität*, pages 209–220, Magdeburg, 2008.

[19] A. L. D. Moura, N. Rodriguez, and R. Ierusalimschy. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, 2004.

[20] D. Reiners, G. Voß, and J. Behr. OpenSG: basic concepts. In *Proc. of OpenSG Symposium 2002*, 2002.

[21] B. Schaeffer and C. Goudeseune. Syzygy: native PC cluster VR. In *IEEE Virtual Reality Conference (VR 2003)*, volume 15, pages 15–22. IEEE Comput. Soc, 2003.

[22] A. Seth, H.-J. Su, and J. M. Vance. A Desktop Networked Haptic VR Interface for Mechanical Assembly. In *ASME 2005 International Mechanical Engineering Congress and Exposition (IMECE 2005)*, pages 173–180. ASME, 2005.

[23] A. Seth, H.-J. Su, and J. M. Vance. SHARP: A System for Haptic Assembly and Realistic Prototyping. In *ASME 2006 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference (IDETC/CIE2006)*, volume 2006, pages 905–912. ASME, 2006.

[24] A. Seth, H.-J. Su, and J. M. Vance. Development of a Dual-Handed Haptic Assembly System: SHARP. *Journal of Computing and Information Science in Engineering*, 8(4):044502, 2008.

[25] G. Wetzstein, M. Göllner, S. Beck, F. Weiszig, S. Derkau, J. P. Springer, and B. Fröhlich. HECTOR – Scripting-Based VR System Design. In *ACM SIGGRAPH 2007 posters*, volume 143, San Diego, California, 2007. ACM.