Beyond the Binding: Virtual Reality REPL and Application Development with VR JuggLua

Ryan A. Pavlik, Student Member, IEEE and Judy M. Vance, Member, IEEE

Abstract—VR JuggLua is a software library, virtual reality framework, and virtual environment development system originating as Lua bindings of VR Juggler and OpenSceneGraph. This combines a high-level, high-performance interpreted language for application development with the extensive configurability and virtual reality hardware support of VR Juggler, as well as the established interactive graphics and rendering power of OpenSceneGraph. It has been extended from this base by the development of an interactive code execution GUI that permits REPL-like interactivity while the frame loop of the virtual reality system continues to execute, even executing new code in parallel and maintaining consistent state across a cluster. An embedded domain-specific language has been built in Lua to provide for more declarative scene graph construction without introducing new file formats or syntax rules. Additionally, to ease development and maintenance of virtual environments, coroutines have been employed to provide the programmer with the ability to write functions that appear to run linearly and in parallel, containing a draw-like call and controlling the timing of drawing, while transparently maintaining state between frames and hiding the repeated update callback idioms. The full system is open source and freely available online, and has been applied in a number of capacities.

Index Terms—distributed applications, interactive programming environments, domain-specific languages, human-centered computing, artificial, augmented, and virtual realities, distributed/network graphics, virtual reality

1 INTRODUCTION

Virtual reality software frameworks span a wide range of capabilities and areas of focus. Each virtual reality (VR) 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 development. The VR Juggler open source virtual reality software platform ([1]) supports a broad range of VR systems, including large multi-node clusters such as the 49-node C6 facility. VR Juggler also supports Windows, Mac, and Linux. The dynamically-typed Lua programming language ([2], [3]) was selected for integration both as a scripting language to extend C++ applications and as a fully-capable development language for building complete immersive applications. Lua’s clear and minimal syntax, ease-of-use for end-user programmers, and ease of interoperability with C++ supported this choice.

The resulting framework, dubbed VR JuggLua, supports the same wide range of VR systems as its VR Juggler core. It uses the OpenSceneGraph (OSG) graphics library to provide scene organization, model loading, and rendering support. Using VR JuggLua, VR applications can be written entirely in Lua, or use Lua in a lesser capacity in an application written using the VR JuggLua C++ API. On top of the basic binding layer, functionality for interactive programming during the execution of a virtual reality application was added. A domain-specific embedded language built within Lua was developed to provide a more intuitive method of scene graph creation while maintaining compatibility with the full functionality of the raw binding of OpenSceneGraph. The programming language construct of coroutines was applied to allow programmers to perceive a linear control flow despite the frame loop of the underlying software, as well as to run effectively parallel tasks without simultaneous access concerns.

The paper proceeds as follows. Relevant literature in three distinct areas is discussed in Section 2. The basic design of the framework and binding of VR Juggler and OSG functionality to Lua, including design challenges overcome, is detailed in Section 3. Developments beyond the binding of software libraries to Lua that provide valuable functionality for virtual reality application development are presented in Section 4. A few instructive examples demonstrating the value and capability of VR JuggLua follow in Section 5.

2 BACKGROUND

2.1 Virtual reality frameworks

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 ([4]) 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 ([1], [5]). Later development extended its use from high-end graphics systems to commodity computer clusters ([6], [5]). 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 ([7], [8]). The Syzygy system presents multiple frameworks for VR application development, and was developed with an explicit focus on clustered execution ([9]).

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 ([10]). Colosseum3D generates bindings of its C++ classes using the tolua++ utility. The commercial VR authoring environment Virtools integrates a custom scripting language, VSL, for content creation. AVANGO/NG applies a generic field and field container programming interface to a scene graph based on OpenSceneGraph, with Python scripting support ([11]).

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 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 ([12]). 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 ([13]).

2.2 Domain-specific languages

The work presented here builds a domain-specific language within Lua. Domain-specific languages, or DSLs, ([14]) became widely known and used as the many “little languages” developed in early computing and expanded by UNIX utilities ([15]). These little languages allow programmability of software in terms of the problem domain, rather than the general-use programming language they interact with, by abstracting the implementation. These particular languages feature their own parser and interpreter, often generated with tools such as Lex and Yacc. A common reason for using a domain-specific language is to solve a problem in a given application domain with concise, clear code. The DSL might be developed and maintained by a software engineer working with knowledge of the domain, with programs in the DSL written or verified by domain experts themselves. Such code can be easier to maintain and trace to domain requirements. In many cases, the creation of a DSL parallels the process of developing a software product line family, where programs written in the DSL specify members of the family ([16], [17], [18]).

A specific kind of DSL is a domain-specific embedded language, or DSEL ([19], [20]), referred to by some as an internal domain-specific language to avoid confusion with embedded device programming ([21]). A DSEL differs from the “little languages” by building upon and within an existing programming language, rather than starting from scratch. A clear advantage of DSELS over external DSLs is that the developer is spared the overhead of developing a syntax, parser, and lexer entirely from scratch, reducing the start-up cost of the DSL approach. Additionally, tooling for the existing language can be re-used. This lowers the start-up cost of using a domain-specific language as compared with developing applications for a problem domain in a general purpose language ([20]).

Some languages are particularly well-suited to hosting DSELS. Common LISP idioms have been considered DSELS, and the functional language Haskell is also considered well-suited to DSEL development ([19]). Notably, Lua was developed to replace an earlier data description language (a limited type of DSL) called SOL, and
features minimal base functionality with syntactic sugar targeted toward development of DSELs that resemble the bibliographic data file format BibTEx ([3]).

2.3 Read-Eval-Print loop (REPL)

Many commonly-used modern interpreted languages, including Python, Ruby, and Lua, provide a REPL-style interactive shell as a development tool. The concept of a “read-eval-print loop” provides an interactive programming environment by reading user input, evaluating the input as an expression, printing results, then looping to allow further execution in the same context. The history of REPLs traces back to the origins of the programming system LISP (LISt Processor) ([22]). In LISP, all data, including programs themselves, were represented as nested structures based on lists. Printing such structures was an early first step in the development of LISP, and for the sake of data persistence beyond a single session reading list structures soon followed. As even programs were represented in this way, development of the \texttt{eval} function produced the first interpreter, which could be called in an infinite loop ([23]). An interactive interpreter could then be implemented with just the functions \texttt{read, eval, print}, and \texttt{loop}, which led to the term commonly used today. It is difficult to trace when exactly these function names became combined into a name for all such similar systems, but Sandewall does use the phrase in a publication from 1978 when describing the “incrementality” requirement of an interactive programming system ([24]).

The concept of incrementality and interactive computing (what would now be referred to as interactive programming) dates at least back to 1964. Lombardi and Raphael conceive of an “interactive computer,” whose focus is on executing expressions, as distinct from executing stored code ([25]). The implementation of these ideas as REPL environments for both programmer support and pedagogical purposes were well-established by the DrScheme environment ([26]). Scheme, a LISP dialect, was used as a teaching programming language in a number of computer science curricula, often based on or inspired by the well-known “Structure and Interpretation of Computer Programs” (SICP) text developed at MIT ([27]). By providing enhanced features on top of a bare REPL, DrScheme provided a useful environment to experiment and incrementally develop programs in Scheme. Based on the merits of a REPL environment for teaching programming, DrJava was developed providing a similar experience built on a Java interpreter instead of Scheme ([28]). While not all interpreter interactive shells rise to the level of these two teaching-focused programming environments, they do provide a useful tool for both beginning and experienced developers. They may not meet the theoretical requirements of a LISP REPL as strictly constructed, but common usage refers to them as such, and as the example of DrJava demonstrates, they provide a number of the benefits of interactive computing.

3 System Design

This section describes 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. Section 3.2 addresses integrating these systems and presents a coherent, logical interface for application development.

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 (Fig. 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 ([1]). It consists of several components that together allow virtual reality applications to be written in C++ and executed using various hardware configurations. 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 presented by VR Juggler. Specializations of the base application object are included that support using scene graph libraries, including OpenSceneGraph and OpenSG ([29]). 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 ([2], [3]). The Lua language must always be tied to a host application. A minimal host application that presents a basic Read-Eval-Print loop (REPL) as well as script execution is included with the standard Lua implementation. The canonical Lua source code is included in the VR JuggLua source tree and built as a library during the software build.

On top of Lua, the Luabind 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 and compile directly to a binding in a single step. VR JuggLua applies 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, reasonably up-to-date bindings for OpenSceneGraph to Lua were available, in a package called osgLua. 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. 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 on newer versions with only minor updates.

Developing the VR JuggLua software framework also resulted in developing a large number of improvements to osgLua and osgIntrospection, to both fix errors and extend functionality. Among the improvements include direct access to object properties without using set/get functions, providing a more natural and Lua-like interface. The introspection-based binding was also augmented with generated code to specifically recognize the vector, matrix, and quaternion data-types, selectively defining additional Lua metatable methods for these values to allow the direct use of the normal math and comparison operators in Lua code.

### 3.1.3 Connecting osgLua and Luabind

The distinct representations of Luabind-wrapped objects and osgLua-managed objects in Lua state presented a challenge during 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. This converter interface was applied to allow osgLua-managed objects to be passed to and returned from methods bound with Luabind.

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. The Boost Type Traits library was used to selectively enable a specialized default converter when a given type inherits from osg::Object. This approach results in the need for only a few-line class to specify the type name for each OSG value type is involved in the Luabind-wrapped method. C preprocessor macros were employed to reduce this to a single line per OpenSceneGraph value 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 Low-level API

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 ([30]) 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 device interface classes in Gadgeteer, the input device management library of VR Juggler. These classes were

---

5. [http://svn.pplux.com/lab/osgLua/](http://svn.pplux.com/lab/osgLua/)
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 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, 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 data types of Sonix, the VR Juggler sound library. 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.  

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. The proxy class derives was created, using a synthesis of the proxy and delegation design patterns ([30]). The proxy class derives from the VR Juggler osgApp class. Lua code can instantiate this application proxy and pass a Lua table data-structure to it, which serves as an application object delegate. If this table has function elements whose names match the methods in 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” as popularized by the Python programming language ([32]). If a Lua table has one or more methods that an application object would have, it can be considered an application object, without requiring the introduction of a specialized type.

Lua does permit binding of classes with virtual methods and the sub-classing 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 sub-classing 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.

4 Extensions beyond the binding

To this point, a fairly direct binding of VR Juggler to Lua has been described, with some novel work to connect distinct Lua binding systems and a relaxing of strict type requirements for application objects as posed by C++. This work forms the foundation for the development of a number of advanced capabilities above and beyond simple creation of VR Juggler applications in Lua code resembling the equivalent C++ code. Taken together, the following extensions beyond the binding provide useful, novel capabilities that both lower the barrier of entry for creating virtual experiences and improve the developer experience across the spectrum of expertise.

4.1 A “Run Buffer” for REPL in event loops

Despite the interpreted nature of Lua and its stock host REPL environment, the basic binding described loses this development interactivity and essentially imposes an “edit-compile-test” cycle with an application restart substituted for the compile step. This is due to the “don’t call us, we’ll call you” design of VR Juggler ([33]), which is common to many GUI event loop architectures. Effectively, once an application completes a few start-up steps, it turns over control to an external event loop which calls it back to handle events. Once the control

transfer statement starting the event loop is executed, it may not return until the application exits. This removes the interactivity produced by a REPL such as the basic Lua interactive interpreter. In the specific case of VR Juggler, the function to start the kernel event loop (known in this case as the frame loop) returns after spawning a thread. However, in practical usage, once the kernel is started by Lua, the Lua thread must block until kernel exit, since any attempt to interpret additional new code from outside the kernel thread would result in concurrent threading problems as the kernel thread and the initial thread both attempt to interact with a single Lua interpreter state simultaneously.

In extending the REPL to this type of architecture, consider that in event loop/frame loop programming, idle operation consists of a continuously executing 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 new code input, however, must still take place within the event loop. The “print” portion of such a REPL consists both of optional text print output (logically directed back to the UI) as well as a change in the continually-running environment’s state. In VR Juggler, the 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. A delay of at most a frame-length between code entry and evaluation does occur due to the asynchronized nature of the GUI console and the kernel frame loop, though this is imperceptible.

Based on these concepts, VR JuggLua includes a thread-safe run buffer system supporting interactive code execution during application runtime, illustrated in Fig. 2. Code can be added to this circular buffer at any time from C++ (including an interactive console) or Lua. 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 latePreFrame state. An interface for an interactive GUI console, with text-based stub, FLTK8, and Qt9 implementations, exposes this code-entry functionality to a user. Any VR JuggLua-based application can use this GUI console as a drop-in component, supporting not only REPL-style code entry and display of print output from Lua, but also additional functionality built into the GUI such as script file loading/saving and drag and drop handling.

Applying the run buffer and GUI console, an interactive virtual reality REPL capability was created. An application initially designed to serve as a testbed for scene creation and manipulation, referred to as “NavTestbed,” is the immersive parallel of the minimal Lua host REPL. 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 start-up, and user code is executed interactively and apparently immediately. This interactive console allows learning of syntax to proceed more rapidly than the edit-compile-run cycle of C++ or even the edit-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. While this application was initially intended to be a minimal testing host, when combined with the control structures discussed in 4.3, it has become the host for nearly all VR JuggLua-based development and operation.

Figure 3 shows this “NavTestbed” application running on Windows 7, on a desktop system in simulator mode. The window in the background is the simulator view, showing a representation of the room coordinate sys-

tem as well as the simulated head and wand position. 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.

The solution described to this point provides an effective REPL for the VR Juggler event loop system in VR JuggLua when run on a single node. However, many target environments for VR Juggler include a cluster of render nodes in addition to an interactive head node. VR Juggler provides a system for synchronize user data across a cluster along with the input device data every frame. By registering the run buffer with this user data synchronization mechanism, code entered interactively on the head node is synchronized and run simultaneously on all nodes of the cluster. By only running the code that had accumulated in the buffer by the time of the pre-frame data synchronization during that frame, consistent execution state down to a per-frame resolution can be guaranteed. Essentially, the code entered by the user/developer between the start of consecutive pre-frame states is held and run on all nodes only once it has been synchronized. When the GUI console is launched in a cluster environment, render nodes can either replace the GUI console with a stub text console, or allow the GUI console to automatically detect its cluster state and disable code input on that node.

4.2 Embedded domain-specific language for scene graph description

Lua allows rapid development of experience authoring techniques, primarily due to its concise “constructor” syntax and table data structures that allow for development of data description and domain-specific languages. 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 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.

```
t = osgLua.loadObjectFile("teapot.osg")
xform = osg.PositionAttitudeTransform()
xform:setPosition(osg.Vec3(1, 0, 0))
xform:addChild(t)
root:addChild(xform)
```

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},
        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 DSEL 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. In a sense, the code resembles the data structure it represents. In this way, an embedded domain-specific language was developed for the most commonly-used scene graph components, including `Transform`, `MatrixTransform`, `Group`, `Switch`, and `Geode` (a geometry node). By building this language within Lua instead of externally such as in Colosseum3D ([Backman2005]), it could be
developed incrementally, and it avoids introducing an additional set of syntax rules and a new file format. Since each DSL function returns the standard osgLua datatype, interoperability is maintained between the DSL and C++-style osgLua code. By incrementally creating wrapping functions for the existing API, the DSEL in VR JuggLua can be characterized as applying the API-to-DSL notation decision pattern ([34]).

4.3 Coroutines for parallel linear control flow

Coroutines are a control abstraction that allow control flow to re-enter at the point where it exited, essentially suspending and resuming an independent thread ([35], [36]). Note that here, thread does not necessarily imply concurrent execution, or the synchronization and locking concerns that come with it. Rather, Lua coroutines provide multiple threads of control, with full lexical scoping and independent call stacks, in a single operating system thread. The application of coroutines to virtual environment creation was reinforced by anecdotal experience working with a novice developer. When considering how one might make an element of a scene continuously rotate, the first reaction of the new developer was to volunteer the idea of an infinite loop. More experienced developers familiar with event loops will quickly recognize that this is not feasible in the traditional way, as the update callback must return for the system to begin drawing. One implementation might transform this concept by preserving state between update callbacks in a member variable or similar structure that persists between calls. However, coroutines transparently preserve execution state between suspend and resume. This provides the basics to effectively inverting the control structure, allowing a user to write code that appears to control drawing and execution.

This is implemented in VR JuggLua as follows. A collection of coroutines resumed each frame, called “frame actions,” is managed by the application delegate, which resumes each one in turn during every latePreFrame state. A developer can create a “frame action” procedure that appears to run with linear control flow, in parallel with other such procedures, by passing a function to a function called Actions.createFrameAction(), which creates a coroutine for that function and adds it to the collection of frame actions. Frame actions may freely call other functions, interact with state, and create additional frame actions. VR JuggLua provides a number of wrappers around the coroutine.yield() Lua function for use within frame actions, the simplest being called Actions.waitForRedraw(). When each still-active frame action is resumed by the application delegate, the duration between the previous frame and the current frame is passed. The result is that waitForRedraw() and its peers appear to pause the calling procedure to draw, then return the amount of time that they paused. It is important that this duration be returned, so that frame-rate-independent animation or simulation can be performed in the frame action.

With this system in place, the combination of the run buffer and the frame action system in the application delegate provides a base for application development without the need for developing additional application delegates. Pure Lua VR JuggLua applications are typically constructed with some immediately-executed code (often setting up the scene graph) and typically one or more frame actions, handling interactions, navigation, and other behaviors. The run buffer and REPL allows additional code to be added during runtime, whether it executes immediately, creates a new frame action, or both. No expressive power is lost by avoiding the use of custom update functions in an application delegate: the idiom of repeated calls to an update function can be emulated by a frame action containing a loop with just an update call and a wait for redraw.

5 Examples

Immersive applications have been successfully written using the VR JuggLua system, both in pure Lua and in a combination of C++ and Lua. Students in a graduate-level introduction to virtual reality course have used it across several semesters to produce final projects. VR JuggLua has been the platform for research in virtual reality and related fields leading to a number of publications ([37], [38], [39], [40], [41], [42]). This section will highlight a few samples of the results achieved using VR JuggLua notable for illustrating particular aspects of the framework’s design.

5.1 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 [10]. 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.

5.2 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 of 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, eventually leading to publication ([38]). 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.

5.3 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. ([43]). 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 Geomagic® Touch™ 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 along with the simulator window. Of course, like all VR JuggLua applications, SPARTA can be run in a fully-immersive mode without any changes beyond config files.

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. Collision statistics are monitored by separate Lua code to trigger appropriate sounds using the Sonix binding in VR JuggLua. Frame actions are used for additional functionality, as well as automation of user study procedures. In effectively providing a super-set of VR JuggLua’s capabilities, most Lua-only functionality developed in VR JuggLua can be used directly in SPARTA.

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 and translate simulation state into visual and audio displays. The Lua interface provides an extension point for investigating interaction techniques.

6 Conclusions and Future Work

VR JuggLua began as a binding of VR Juggler and OpenSceneGraph to Lua. Subsequent research and development have extended its potential in three major ways. Interactive code execution in the manner of a REPL was re-introduced using a cluster-capable, thread-safe run buffer. An embedded domain-specific language for scene graph description in Lua was built. Finally, coroutines were applied to invert apparent control flow to permit more linear execution and formulation of virtual environment behavior. The software is developed under an open-source license and both source and Windows binaries are provided online.

The interactive GUI prompt adds code to a run buffer asynchronously from the frame loop. The run buffer concept allows interactive code execution simulating the behavior of a blocking REPL while the frame loop
continues. Just syncing the buffer when device data is synchronized is not sufficient to extend this to a cluster because it’s possible to enter code after synchronization but before the contents of the buffer are run. This problem was solved by only executing those run buffer entries that have been synchronized, and delaying code entered between sync and run until the next frame. This allows interactive code execution, in a running frame loop, across a cluster maintaining consistent state on all nodes.

Building bindings for VR Juggler and providing a base application shell eliminates the boilerplate required for a simple application. Between new bindings written for VR Juggler and the enhanced introspection-based OpenSceneGraph bindings, a large proportion of the underlying functionality is available. However, this functionality was primarily available through imperative-style code strongly resembling its C++ equivalent. By building an embedded DSL for scene graph description in Lua, common scene graph construction tasks can now be written in a way that reflects the structure in a more declarative style. By employing the API-to-DSL notation decision pattern, the DSL is seamlessly compatible with the fullness of functionality accessible through the imperative C++-style API. The DSL simplifies common code in a way that is both useful to novice VR programmers learning about scene graphs and helpful to experienced programmers working on development and maintenance of a virtual environment.

Typical control flow in VR software involves relinquishing control to an event loop that will periodically call a particular function for application-specific logic and updates. In the context of developing a virtual environment, it can be more natural to think of program logic proceeding somewhat linearly, rather than as repeated calls to an update function. Coroutines provide a model and method of multiple parallel, non-concurrent threads of control that can pause and resume execution. By implementing a collection of coroutines resumed each frame during the update function, and wrapping a “yield” (return and suspend) call as a “wait for redraw,” a system of “frame actions” was developed for providing more linearly-structured execution with state transparently preserved across multiple frames. The frame action coroutine system can trivially emulate the per-frame “update callback” idiom, so it loses no expressive power.

Future work includes collaborative work to broaden usage of VR JuggLua outside of Iowa State University courses and research. Some technical changes could be made to more easily permit use of third-party Lua modules for orthogonal functionality. While the package runs on Windows, Linux, and Mac OS X, some additional work is needed to provide an easily-distributable application bundle on Mac. Formal evaluation of the system’s usability with novice programmers is also being considered.

ACKNOWLEDGMENT

Thanks to Leif Berg, Patrick Carlson, Meisha Rosenberg, Adam Carlson, Alicia Fleeger, Andrew Moore, and Jane Peters for using VR JuggLua in diverse real-world applications and providing valuable feedback. This paper builds upon work initially presented at the Fourth Workshop on Software Engineering and Architectures for Realtime Interactive Systems at IEEE VR 2011. This work was performed at the Virtual Reality Applications Center at Iowa State University. This material is based upon work supported by the National Science Foundation under grant #CMMI-0928774.

REFERENCES


Ryan A. Pavlik received a BS in computer science and math from St. Norbert College. He is a PhD candidate in human-computer interaction and computer science at Iowa State University. His research interests include usability, virtual reality, virtual assembly, and haptics. He is a member of ACM, ACM SIGCHI, and IEEE.

Dr. Judy M. Vance received her BS, MS and PhD degrees in Mechanical Engineering from Iowa State University. She is the Joseph C. and Elizabeth A. Anderlik Professor of Engineering at Iowa State University. Her research interests include virtual reality applications in mechanical design, especially with respect to decision making in early stage product design. Vance holds an honorary Doctorate of Engineering degree from Heriot-Watt University. She is a fellow of ASME, a life member of SWE, and a member of IEEE, AIAA and ASEE.