# Emerson: Accessible Scripting for Applications in an Extensible Virtual World

Behram F.T. Mistree, Bhupesh Chandra,
Ewen Cheslack-Postava, Philip Levis
Stanford University
{bmistree, bhupc, ewencp}@stanford.edu,
pal@cs.stanford.edu

David Gay
dgay@acm.org

## Abstract

This paper presents Emerson, a new programming system for scripting objects in user-extensible virtual worlds such as Second Life, Active Worlds, Open Wonderland, etc. Emerson's primary goal is to make it easy for novice programmers to write and deploy interesting applications. Scripting applications for these worlds is difficult due to two characteristics: the worlds must scale to millions of users and are therefore distributed, and there is no central authority or design so interaction is mostly between mutually untrusting applications.

To simplify scripting for novices, Emerson employs two abstractions: multi-presencing and execution sandboxes. Multi-presencing allows a single program to centrally control what seem to be many distributed geometric objects. Execution sandboxes allow safely running application code provided by another object, borrowing the execution and deployment model of modern web applications.

Emerson itself is implemented as a scripting plugin for the Sirikata open source virtual world platform. We evaluate the benefits of its design by describing several application examples. Through these examples, we explore the interactions between sandboxing and multi-presencing as well as their implications and discuss potential future authentication mechanisms that would make secure in-world application development more accessible.

***Categories and Subject Descriptors*** D.2.6 [*Programming Environments*]: Interactive Environments, Graphical Environments; D.3.2 [*Language Classifications*]: Specialized application languages; I.6.8 [*Types of Simulation*]: Distributed, Gaming

***Keywords*** Virtual Worlds, Scripting, Sandboxing, Actors, Metaverses, Distributed Programming, Novice Programming

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*Onward! 2011,* October 22–27, 2011, Portland, Oregon, USA.
Copyright © 2011 ACM 978-1-4503-0941-7/11/10. . . $10.00

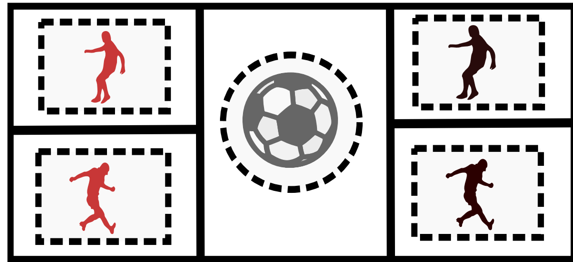
## 1. Introduction

Virtual worlds are shared 3D spaces. Users log into these spaces as "avatars," which are in-world representations of themselves. Through their avatars, users interact with each other and other objects populating the world to play games [29], create art [31, 33], socialize [13, 41], learn [20], and even engage in therapy [23].

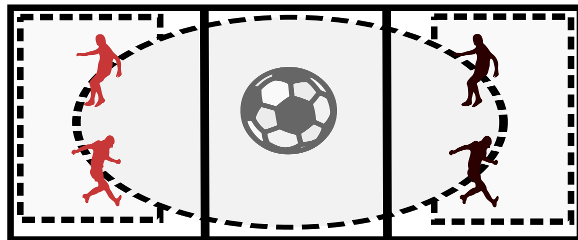
Virtual world objects are controlled by scripts, code which defines their behaviors. For example, a virtual car has code for accelerating when an avatar presses its gas pedal or to blare an alarm when a stranger tries to open its door. Scripts transform pretty but lifeless worlds into dynamic, evolving, and interesting ones.

Many of the most successful virtual worlds today – World of Warcraft [42], EVE Online [11], and The Sims [37] – have centrally authored code and applications. These worlds resemble ISPs of the early 1990s, such as Prodigy and America Online, which centrally controlled and curated content. Large teams of professionals carefully create and manicure the behaviors of all objects in the world. For example, at one point in 2009 World of Warcraft had at least 37 designers creating player classes, professions, levels, and events, and had generated 70,000 spells and 40,000 non-player characters since 2001 [35]. This approach is costly and can limit both a world's scope and its character.

Our goal is to build virtual worlds more like the Web of today, called *metaverses*, where anyone, even novices, can add new applications or services. Applications in virtual worlds are collections of objects that are scripted to act in concert, providing new and interesting experiences for other inhabitants of the world. For example, a virtual art gallery may alter its exhibit dynamically in response to avatars' viewing it, or teams of monsters may coordinates attacks



(a) Standard approach



(b) Emerson approach

**Figure 1.** Emerson changes trust and locality boundaries for scripts. With these more flexible boundaries, one script can control an entire team and the code for the soccer game performs less messaging, making the entire application simpler.

on a virtual citadel. While a few of today's virtual worlds, such as Second Life, make it easy to script *individual* objects, building *applications* remains a significant challenge for even experienced developers.

Just as anyone can create a dynamic web page, any novice should be able to build simple applications in a metaverse. Two characteristics of metaverses make application development hard:

**Size**: To scale to billions of virtual objects, object state and code are distributed across many servers. Because virtual worlds are interactive, applications must remain responsive in the face of *latency*. Distributed application state may lead to *consistency* challenges. Even without state replication, applications must handle *message reordering* and *loss*. An otherwise simple application must handle all the challenges and edge cases of a distributed system.

**No Central Authority**: Objects scripted by different users require *protocol agreement* to interact, but without central control it is impossible to agree on all protocols in advance. How can an application developer ensure his/her application objects will be able to interact with any user? Further, mutually *untrusting* objects must interact securely, for instance without leaking sensitive state stored within a script.

Emerson is a new programming system for metaverses that aims to make it easy for users to write interactive applications in virtual worlds. Built on top of the open-source Sirikata virtual world platform, it is currently undergoing controlled testing with a group of eleven undergraduate students. Earlier work on Emerson focused on its use of prototypic inheritance for virtual world object creation and its, since deprecated, messaging syntax [4]. This paper focuses instead on mechanisms for Emerson that provide better control over where code is executed and support execution of untrusted code. These mechanisms make more code operate locally and make applications written in Emerson simpler than their equivalent versions in other systems.

Metaverses today take the straightforward approach towards trust and locality shown in Figure 1(a). The solid lines show script, and therefore locality, boundaries: interaction between scripts requires messaging, even for objects that are scripted for the same application. Modern systems manage locality boundaries in different ways. For instance, as described in Section 4, Second Life provides an abstraction for "linking" objects, but constrains it in such a way that makes it unusable for many applications. Emerson simplifies code involving multiple virtual world objects with *multi-presencing*, which allows one script to control multiple virtual world objects, or presences. As shown in Figure 1(b), a single Emerson script can control multiple virtual world objects. In a broader context, multi-presencing decouples locality boundaries from separately addressable interfaces.

The trust boundary, shown with dotted lines in Figure 1(a), in today's systems is tightly coupled to the locality boundary: all code within an object's execution environment is fully trusted and all code outside is untrusted. Emerson provides *sandboxes* for executing untrusted code locally, much as a browser allows a web site to execute JavaScript locally. Section 3 shows how this simplifies common interactions between applications and their users. For example, instead of sending all input events, such as a user clicking and moving a chess piece, back to the chess game as messages, input handling becomes much simpler and more efficient when performed locally on the avatar's application. Figure 1(b) shows how the trust boundary for an Emerson script can extend into another script and across the locality boundary.

These modifications make code which would otherwise be distributed and challenging to write become simple, single-threaded, and accessible to novices. While the challenges of developing code for execution in a large-scale system and without central authority cannot be completely avoided, they can be greatly mitigated, making application writing in virtual worlds easier and more accessible for novices.

The remainder of this paper describes Emerson in greater detail, focusing on the features which make application development simpler and more accessible to novices. Sections 2 and 3 give background on the virtual world platform on which Emerson is built and a motivating example, respectively. Section 4 gives a detailed description of multi-presencing. Section 5 discusses Emerson sandboxing and proposes a mechanism for authenticated messages in the Emerson programming system. Section 6 describes Emer-

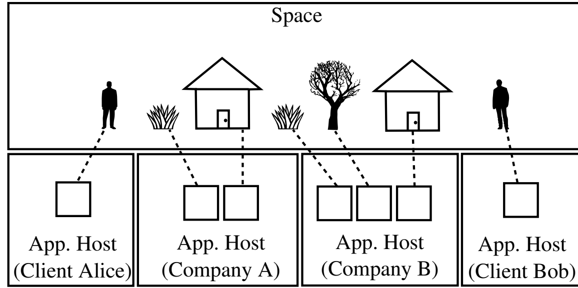



**Figure 2.** An example of a Sirikata world. Multiple application hosts, each in a different administrative domain, connects to the shared space.

son's programming model and runtime in more detail. Section 7 describes related work and Section 8 concludes.

## 2. System Background

Emerson is a programming system built as a plugin to Sirikata [32], a platform for large-scale, federated, seamless virtual worlds. Like virtual worlds in general, at their core, worlds based on Sirikata are collections of distributed, message-passing objects. Emerson drives the simulated behavior of these objects.

Figure 2 shows the components of Sirikata relevant to Emerson. Each *application* is an executing script, and when initialized has no connections to the world. Applications connect to a *space* and obtain a *presence*. Presences give applications a physical manifestation within the space – for example, a building, bicycle, or avatar.

Each presence has properties representing its state in the world, such as position, orientation, mesh, queries for other objects, and a unique identifier for messaging. The space owns and controls these properties, and an application's script must explicitly request the space to change them. Two applications, even those running on the same application host, cannot communicate directly. Instead, they learn about each others' presences through geometric queries to the space. Using the identifiers returned by these queries, applications communicate by exchanging messages between presences in the same space, much in the same way that vats in E send messages to objects in other vats [21].

Emerson does not assume a one-to-one mapping between applications and presences. This enables *multi-presencing*, where an application acquires more than one presence in a space. Section 4 motivates and discusses this feature in further depth.

As Figure 2 shows, applications are hosted separately from the space, allowing multiple domains to connect to the same world: Sirikata virtual worlds are federated. Any presence encountered in a space is potentially running under a different administrative domain.

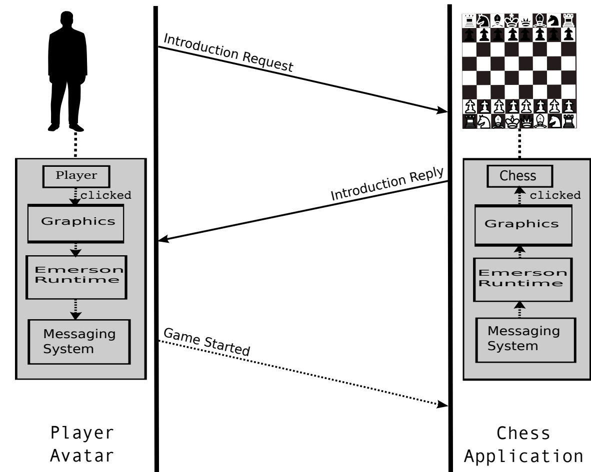



**Figure 3.** A simple introduction protocol combined with sandboxing allows processing of input to be performed within the player's script, simplifying the entire chess application. The solid arrows show messages from the original scripts while dashed arrow shows message from the sandbox.

## 3. Motivating Example

To demonstrate the challenges of writing a virtual world application and how Emerson addresses them, this section describes a simple example: a chess game. Avatars approach the chessboard and can join the game. When a player makes a move, his/her avatar animates to move the piece on the board. The game enforces valid moves, tests for the end of the game, and resets the pieces for the next game.

A scripter faces many challenges when building even this seemingly simple game. First, how do the chess application and avatar, which are developed independently, begin communicating in order to start a game?

Second, once communication is initiated, how does the chess game coordinate the movement of the many presences – the board and all the pieces – involved in the game? Finally, how can the chess game handle interaction with the player, converting basic interaction such as clicks into meaningful actions like moving pieces?

### 3.1 Bootstrapping an Application

The core Sirikata system only implements a small number of protocols, which applications use to perform the most primitive tasks: moving about the world, querying for other presences, and sending and receiving messages. Using these primitives, how can an avatar request to join the chess game? Agreeing on a chess protocol ahead of time is too inflexible, making it impossible to add truly novel applications to the world.

Instead, Emerson provides a simple *introduction protocol* to bootstrap interactions. An introduction message indicates to the receiving application that a presence is interested in further interaction, putting the burden of requesting interac-

tion on the "user" of an application. For avatars, this protocol is initiated by clicking on an object. For the chess example, Figure 3 shows how this translates into an introduction message which is sent to the chessboard.

### 3.2 Coordinating Many Presences

The chess game application contains many presences, all of which must move independently. How does the developer script and coordinate these presences? It must control their movement, reset them when a game completes, and destroy them when the game is "packed up."

Most systems use a straightforward, and perhaps the most intuitive, approach in which locality boundaries are inflexible and tied to exactly one presence, as shown by the solid lines in Figure 1(a). Messaging must be used to control the pieces and send move requests back to the board to be checked by the game logic. In essence, the chess game is a small distributed system, which comes with all the same challenges. What would be simple, single-threaded, synchronous code in a desktop chess application now requires a much more complicated distributed implementation.

Emerson decouples the presence and application abstractions, allowing *multi-presencing*, in which one application with a single script can control multiple presences. Although presences appear independent, the same script controls all of them and, in the chess example, the game state can be centralized. The game logic and event handling become much simpler since all actions can be taken directly. For example, to reset the game, the script simply iterates over all pieces and resets their positions, a much simpler approach than all the messaging and acknowledgements that would otherwise be required. Multi-presencing and its implications on scripting are described in detail in Section 4.

### 3.3 Controlling an Application

After introduction, how does the rest of the interaction play out? At first glance it seems that simple touch events would work by triggering messages to the application: moves could be indicated by clicking on a piece and then on a space on the board. But this approach is actually quite challenging because it requires coordination between the player, the pieces, and the board. The chess script must handle many edge cases involving message loss, reordering, and latency.

The problem is that only the most primitive events from mouse input are sent from the avatar. If the avatar's script could understand the intention of the sequence of click events, it could convert them into a single message to the chessboard requesting a move. However, the avatar was not aware of the game when it was scripted, so it cannot generate such a message.

Emerson addresses this problem with *sandboxes*. Sandboxes allow a hosting application to safely execute an Emerson script received from another application. In the chess example, the chess board sends a message containing a sandbox request, including an application description, capabilities requested for the sandbox, and code to execute within the new sandbox. When the user accepts, the code element of the sandbox request is executed within a newly created sandbox, and begins listening for user interface events, a capability included as part of the request. The code processes these events and only sends messages back to the board about moves, rather than individual clicks. All input is handled locally and in order, so no ambiguity or errors due to distribution are possible.

By limiting the local state and operations on presences available to code within the sandbox, an application can control the abilities of sandboxed code. In this way, an application can protect sensitive data in its own script, restrict sandboxed code from performing malicious actions with a presence, and avoid accidental interference with the original script. Sandboxes are described in more detail in Section 5.

Sandboxes do not completely remove the need for writing distributed code, but they do make it easier in two important ways. First, all application code is developed in a single application, but some may be executed on other applications. Second, the developer can choose what events and actions operate between applications and require messages. Importantly, delegating this decision to the scripter allows the application developer to choose the smallest possible set of messages for minimal implementation complexity or the set resulting in the least network traffic for optimal performance.

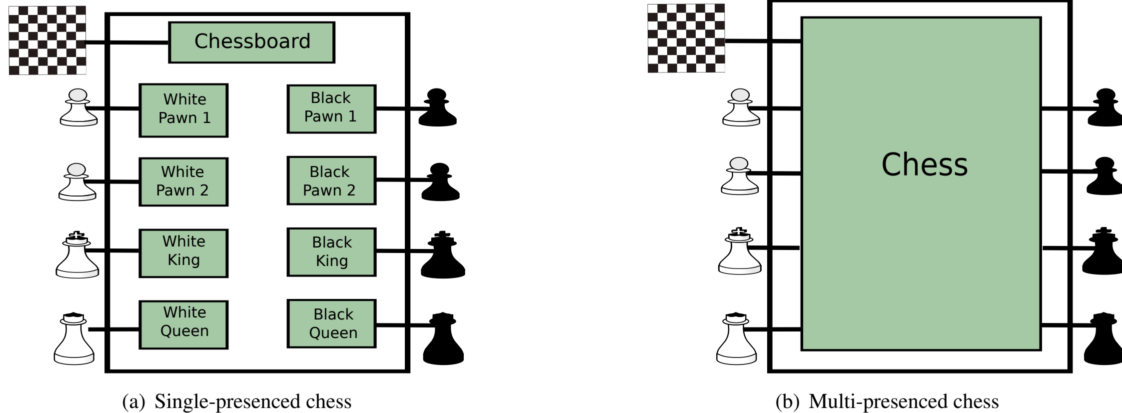
### 3.4 Summary

The chess example shows how complicated even a simple application can become in current systems. Emerson proposes three mechanisms to reduce the difficulty of developing application: a global introduction protocol, sandboxing, and multi-presencing. The remaining sections give a more complete description of Emerson and explore these features and their interactions in more detail.

## 4. Multi-presencing

All but the simplest applications involve multiple presences interacting with each other and the user. A chess game requires many pieces and the chess board. A bank provides many ATM locations throughout the world. An art vendor has multiple galleries with many pieces of art.

Different systems deal with coordination of many presences differently. UnrealScript is designed to run in a single, trusted process and so treats the entire world as one application with many presences. An UnrealScript developer makes direct function calls to control any presence in the world [36]. The ordering, latency, and failure challenges described in Section 3.2 do not apply because the world is contained in a single process. This constraint limits the scale of such worlds and disallows untrusted scripters from participating in programming the world.




(a) Single-presenced chess (b) Multi-presenced chess

**Figure 4.** Multi-presencing greatly simplifies code: a single script drives many presences and the application does not require any messaging to control and coordinate its own presences.

Scripts in Second Life are associated with a single "primitive," but unlike Emerson each primitive can be controlled by multiple scripts. A script can also reach beyond its primitive through a "link set," a collection primitives which have agreed to act in coordination [30]. Link sets have a number of restrictions. First, no more than 256 primitives can be grouped and they all must be contained within a bounding sphere of diameter of 56 meters [17]. A chess game fits within these constraints, but a game of Go can use up to 361 pieces. An ATM application is not useful if it cannot spread ATMs more than 50m apart.

Emerson does not limit the number nor geographic distribution of presences connected to a single application. Each application contains a single script defining its behavior and controlling its presences. The application connects to a world (or worlds) to gain a physical representation, or presence. *Multi-presencing* means that each application directly controls multiple presences.

Figures 4(a) and 4(b) show single- and multi-presenced versions of the chess application, respectively. Instead of an individual script per piece, a single chess script drives the entire application, creating and controlling presences for each piece and the board. All the messaging internal to the application, *i.e.*, between presences it creates, become function calls within the same script, greatly simplifying the code. Players are still separate – they are avatars and are not scripted as part of the game – so not all messaging and distribution-related challenges are removed.

### 4.1 Multi-presencing Details

Multi-presencing introduces one additional global function:

```
system.createPresence(space, cb, ...)
```

`createPresence` requests a new presence in `space`, and invokes `cb` with the new presence, or null for failure, upon completion. Additional parameters set initial values for properties of the presence, such as position or mesh.

Compared to other systems, methods which take an action in the world require an additional parameter: the presence object created by `createPresence` to operate on. Unlike in other systems, all operations involving the space – setting physical properties like position and mesh, querying for other objects, messaging, etc. – are not global and must operate on a presence object. Additionally, presence objects can be destroyed explicitly: instead of coupling the lifetime of the presence and application, destroying a presence only removes that physical representation from the world and the script continues.

### 4.2 Self

Although multi-presencing greatly simplifies code using multiple virtual objects, it seems to have a drawback: a script using only one presence still must track this presence and explicitly specify it in all operations. To a novice this is an additional barrier to entry, one more thing they must understand to write even a simple script.

Even with multi-presencing, in most code there is usually a default presence to operate on. For example, when a presence receives a message, the response should almost always be sent from the same presence. Timers are frequently set in response to an event, and that event is usually associated with some presence. Actions taken in the timer callback will usually use the presence associated with the event that triggered the timer registration.

To avoid unnecessary verbosity and keep simple scripts truly simple, the Emerson runtime tracks the default presence for each event and provides it to the script via the global name `self`. `self` allows the user to easily access the current presence and, in some cases, such as sending messages, elide the presence part of the statement.

`self` is sometimes undefined. For example, when initialized, a script has no presences so it cannot have a default. Besides initialization, there are two types of events. Callbacks generated by events from the space (*e.g.*, messages, proximity events) set `self` to the presence associated with

the event (*e.g.*, message recipient, presence receiving proximity event). Callbacks generated by runtime requests, such as timers, inherit `self` from the callback that triggered the request.

`self` was intentionally named to evoke self-referencing pointers and references in object-oriented languages: whereas in languages such as Ruby and Smalltalk, `self` refers to the current software object, in Emerson it refers to the "current" presence. JavaScript's `this` keyword retains its traditional semantics in Emerson. In our limited experience thus far, `self`'s naming has not caused confusion, and `self` itself has been fully-adopted by our users.

### 4.3 Limitations

Multi-presencing enables application developers to write code that would necessarily be distributed in other systems as simple, single-threaded code. However, multi-presencing is not always applicable. The following are some application characteristics that are incompatible with multi-presencing.

**Multiple Domains of Trust** Applications may require coordination between presences controlled by different users. While most presences in the chess game can be unified under a single presence, players are independent avatars with separate scripts. The next section explains how Emerson simplifies this type of interaction.

**Scalability** Some applications will require more resources (memory, CPU, bandwidth) than a single host can provide. For example, a bank providing thousands of ATMs likely cannot run on a single host.

**Fault Tolerance** If an application must maintain high availability, a centralized multi-presenced application is not an appropriate solution. These applications might still take advantage of multi-presencing, but cannot rely on a single multi-presenced application to provide their service.

We believe that the vast majority of applications do not fall into these categories. Aside from user interaction, most applications exist under a single domain of trust. Most applications aim to exploit the benefits of locality that a virtual space provides, so scalability is usually achieved by independent instances if necessary. And most applications are not mission critical: a short period without service is simply an annoyance, rather than a catastrophe.

Application developers cannot be fully protected from distributed programming. At a minimum, any useful script will need to interact with some other objects, such as avatars. However, Emerson reduces exposure to distributed programming by allowing as much code as possible to be written as a single script with shared state and where all operations are synchronous.

## 5. Sandboxing

Multi-presencing makes scripting interactions between presences connected to the same application easier. However, it does not address the challenges of scripting interactions between presences on *different* applications. Such interactions are common in a virtual world. For instance, Second Life has a multi-million USD in-world economy [9]. As part of that economy, avatars can interact with objects in the world by transferring "Linden Dollars" to them to buy and sell virtual goods.

In the chess example, the chess application sends the player code to run locally. There is a tradeoff in supporting this interaction. Allowing the chess application to execute code within the avatar simplifies the application. First, it avoids the requirement of protocol agreement by allowing the chess application to specify how to translate mouse events into messages it can decode. Second, it avoids message reordering because mouse events are guaranteed to be processed in order on the player's host and moves can be requested over a single, ordered stream.

However, executing this untrusted code locally aggravates the trust challenge and requires the avatar to understand the details of sandboxes to ensure security. This tradeoff motivates the general principle behind the design of Emerson sandboxes: *Emerson's sandboxes eschew complicated, edge-case functionality to make sandboxes as safe and usable as possible for novice scripters.*

### 5.1 Sandbox Interface

All code in an Emerson application is executed within a sandbox that provides data isolation: each sandbox has its own environment and cannot directly access data in other sandboxes. Each application starts with a single root sandbox with full capabilities. Except for the root sandbox, each sandbox has a single parent and can have arbitrarily many child sandboxes.

Figure 5 shows a more complicated example than chess, which requires a tree of sandboxes. The root sandbox for an avatar has initialized multiple child sandboxes for different applications it has encountered. One is for a marketplace service, which finds nearby vendors using a proximity query on the avatar's presence. When the avatar approaches a vendor, the marketplace sandbox creates its own child sandbox to execute code from the vendor, allowing it to open a display for purchasing items. This results in the tree structure shown in the figure and safely isolates code originating from different applications.

It is easy for a scripter to create a sandbox and execute code within it. Returning to the chess example:

```
var caps = new Capabilities(Messaging, Gui);          (A)
var newSandbox =
    system.createSandbox(avatar, chessboard, caps); (B)
newSandbox.execute(chessCode);                        (C)
```

Line B shows the creation of the sandbox. The first two parameters associate the sandbox with one presence owned by this application (`avatar`) and one other presence in the

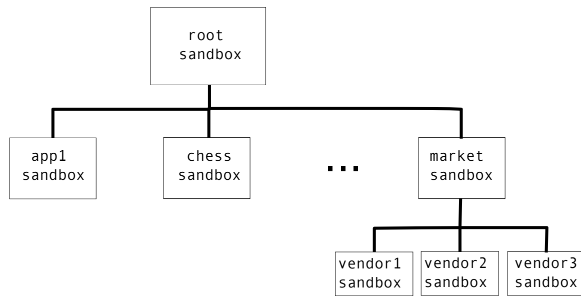



**Figure 5.** Each application has a tree of sandboxes, and each sandbox can only communicate directly with its parent and children.

space, *e.g.*, the chessboard's presence that sent the request for the sandbox (`chessboard`).

Each sandbox is created with a presence because all interactions with the world require a presence: a sandbox which can have no external effects would be useless. However, the sandbox is not restricted to this single presence: given the right capability, described below, the sandbox may create more presences.

The second parameter, another presence in the space, provides the *origin* of the sandbox. This allows the script to provide blame for a poorly behaving sandbox. But more importantly, as sandboxes are frequently used to filter events and relay information back to the creator, storing this owner presence allows the system to provide a shorthand for reporting messages back to the owner:

```
system.sendHome(moveMsg);
```

which sends `moveMsg` to `chessboard` from `player`.

### 5.1.1 Capabilities

Each sandbox has a set of capabilities, allowing the creator to restrict what actions it takes in the world (lines A and B above). Capabilities both restrict actions that can be taken on presences, such as registering queries, sending messages, and moving the presence, as well as other system actions, such as creating new presences, creating child sandboxes, and presenting a GUI to the user. All capabilities are off by default: although this requires explicit agreement by the scripter (or indirectly via the user interface, the user), it is worth the improved security. An error in specifying capabilities means a script failure rather than the ability to take over an application.

Capabilities are specified when the sandbox is created. After this point, they are immutable. We could only envision a narrow range of uses where mutability was strictly required, and felt that handling dynamic capabilities would be too challenging for novices. With dynamic capabilities, sandboxed code is difficult to write because its abilities may change at any moment. Additionally, the sandbox host may forget to revoke capabilities after enabling them temporarily.

Children sandboxes have a subset of the capabilities of their parents: code cannot increase its capabilities by creating additional sandboxes and executing from within them.

### 5.1.2 Sandbox Messaging

Emerson sandboxes are isolated so they cannot directly interact via function calls. Instead, Emerson allows child and parent sandboxes to exchange messages. This feature allows scripters to use messaging to extend the interface between sandboxes beyond their built-in capabilities.

A sandbox cannot send a message directly to any sandbox that is not its parent or its child. For instance, a sandbox cannot send directly to a grandparent, sibling, grandchild, or cousin, and instead must rely on parents and children to forward the message on. This design ensures linear data flow: for any two sandboxes to interact, there is only one shortest path that messages can take between them. This makes sandbox messaging easier to reason about and makes it easy for scripters to filter messages that might lead to security vulnerabilities.

Messages are not shared state: the receiver is passed a copy of the original. We experimented with other techniques but concluded they were too challenging for novices. For instance, we initially allowed parents to pass additional object references (not copies) to the child as parameters to `createSandbox`. We discarded this approach for two reasons. First, scripters had trouble keeping track of everything a single variable referred to, and therefore the collection of items being exposed to the child sandbox. For example, an object might, by chance, have a reference back to a parent's `system` object, allowing the child sandbox to access nearly all state available to the parent. Second, this encouraged unsafe laziness, such as passing the entire `system` object to child sandboxes.

### 5.1.3 Sandbox Control

Sandboxes run untrusted code, so they cannot always be expected to behave well. In these cases, parent sandboxes can forcefully control the execution of children with three methods: `suspend`, `resume`, and `clear`. `clear` destroys the sandbox and may be used for instance if the sandbox uses excessive CPU, memory, or is no longer necessary. Once cleared, code within a sandbox cannot be resumed, and all the objects within it are marked for garbage collection. Although this decision may be revisited in the future, to provide sandbox scripters some guarantees about message ordering, sandbox execution prevents execution of other sandboxes. To avoid denial-of-service attacks, such as code within a sandbox executing an infinite loop, system-enforced resource limits automatically clear sandboxes that have continuously executed too long.

The `suspend` and `resume` methods allow temporarily pausing execution of the sandbox. Suspend and resume are useful for code associated with activities that are transient but repeated, and for which maintaining state is useful. For

example, a player in a game may run a sandbox which enables participation in the game and which also maintains statistics about its participation. The sandbox should only be enabled during play but should not be cleared because it stores valuable state.

All events from presences, such as messages and proximity updates, are dropped while suspended. Other events, such as timers, are deferred until `resume` is called. The Emerson runtime synthesizes additional events for updates where dropped messages lead to inconsistent or impossible states, for example the streaming updates provided for proximity queries.

### 5.2 Sandbox Authentication

An implication of our sandboxing approach is that applications cannot rely solely on the sender of a message to authenticate sensitive transactions. For example, consider a bank application that uses only the presence identifier of a message for sender authentication. Avatar A executes code from both the bank and code from another application – a chess game – in separate sandboxes, both of which have been given the capability to send and receive messages from Avatar A. Although the bank trusts the avatar itself, it does not trust any other sandboxed code running on A. If the bank receives a message from A to transfer money to another avatar in the world, the bank cannot be sure if this message was truly initiated by its sandbox executing on A or whether the chess application's sandbox deceptively constructed a transfer request message from its sandbox on A: both messages would have the same sender identifier. Similarly, with the ability to receive messages, the chess sandbox can snoop on traffic intended for A, so bootstrapping secure communication with simple in-band approaches such as a secret token is not possible. Thus, sandboxes as described make it difficult to ensure secure communication.

Web applications are a familiar example of a system which faces similar challenges. Like in Emerson's sandboxes, JavaScript code can send messages on behalf of the user. To reduce the risks this ability creates, browsers institute a same-origin policy: although any sandboxed script can make a *request*, it cannot read a reply unless its origin matches the protocol, domain, and port for the site being messaged.

We believe the same-origin policy is too restrictive for virtual worlds, and emerging web standards indicate it is too restrictive for Web applications as well [39]. For instance, a banking application in which an avatar can interact with many ATM presences controlled by the application would require a separate sandbox per ATM under this policy: a sandbox for an ATM could only communicate with that ATM.

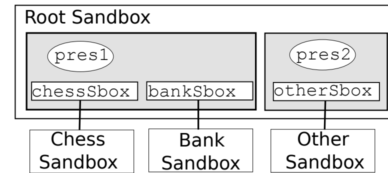
To address the message authentication problem in Emerson, we are considering modifying the messaging rules described in Section 5.1.2 to include *SMessages*. SMessages add source and destination sandbox fields to regular messages. Emerson would restrict delivery to the specified destination sandbox, much like TCP and UDP ports restrict the receiver of packets. SMessages provide the sender finer control over the destination of messages and address the problem of snooping sandboxes. In the bank example, messages from the bank can always be sent to the avatar's root sandbox, ensuring only the original, trusted script interacts with the bank during sandbox creation. Similarly, messages sent to the bank can be verified as coming from the avatar's root sandbox.

```
var caps = new Capabilities(Messaging);

var chessSbox  = system.createSandbox(caps,pres1,chess);
var bankSbox   = system.createSandbox(caps,pres1,bank);
var otherSbox  = system.createSandbox(caps,pres2,other);

chessSbox.execute(chessCode);
bankSbox.execute(bankCode);
otherSbox.execute(otherCode);
```

(a) Sandbox creation code




(b) Application state

**Figure 6.** Without SMessages, code within the bank sandbox and chess sandbox can read any message sent to the application's first presence.

Root sandboxes always have a sandbox identifier of zero so that the original script can easily be addressed. Although scripters can explicitly set a bit to deliver to all sandboxes with the capability of receiving messages on a presence, without a specified sandbox, SMessages deliver solely to the root sandbox.

As a concrete example of how SMessages may work, consider Figures 6(a) and 6(b). Because `chessSbox` and `bankSbox` both have their capabilities set so that they can listen for messages on `pres1`, without SMessages *any* message sent to `pres1` can be read by the code executing in the chess sandbox and the bank sandbox. With SMessages, a sender can specify that his/her message should only be read by a particular target sandbox, for instance, the root sandbox or bank sandbox, and avoid eavesdroppers.

We believe that SMessages are both secure and flexible. Unlike an approach that would adopt a same-origin policy similar to browsers', using SMessages, sandboxed code (given the capability) can still message any other presence in the world. The remaining challenge lies in making this mechanism as accessible to scripters as possible. Building systems and abstractions that allow applications that make it painless to send to appropriate destination sandboxes and that discourage common security bugs will be absolutely essential. The prevalence and severity similar bugs in related

systems, such as cross-site request forgery vulnerabilities in many Web applications, suggests how important and difficult it may be to make in-world security usable for novice scripters.

## 6. Emerson Details

The discussion to this point has focused on the high-level semantics of Emerson. This section provides additional details to give the flavor of developing a complete Emerson application. For brevity and diversity, we depart from the chess example and examine an *indexer* application. The indexer is multi-presenced and catalogs some basic information – identifier and position – for presences in the world. It uses timers to make its presences wander through the world, proximity queries to find other presences, and messaging to introduce itself to them. When one of the indexer's presences receives an introduction reply, it catalogs the sender's address and position in the world for later use.

### 6.1 Programming Model

Emerson is a dialect of JavaScript. Although we add some simplifying syntax, it mostly preserves JavaScript's syntax and semantics. JavaScript's prototypical inheritance and very dynamic object system fit the application well, but we also chose JavaScript for a more pragmatic reason: we expect more novices to have encountered JavaScript through simple Web programming than other languages. Emerson users with some familiarity of JavaScript should be able to quickly enter the world and create scripts. We can also leverage existing tools and tutorials to get new developers started.

JavaScript also trivially supports interactive development. Many existing worlds such as Second Life and Unreal [36] require scripters to "reboot" objects to reflect changes in their associated scripts. We believe this makes experimentation and debugging more difficult for novice users and increases the cost of each iteration of development. We believe a better model is one in which users customize virtual world objects in an interactive process, for instance adding a `wagTail` method to a virtual dog so that it appears excited when petted. JavaScript naturally supports this behavior without requiring off-putting recompilation steps.

Emerson scripts are single-threaded and event-driven. Examples of events in Emerson are timer expiration, a presence's connection to and disconnection from the world, and message reception. Users can provide callbacks to Emerson, which execute when each of these events are triggered. We chose to avoid multi-threading because it introduces challenging concurrency problems such as serialization, races, and deadlocks, all of which are too difficult for novices to deal with. Because Emerson's target audience is novices writing relatively simple scripts, we believe this is the right tradeoff.

### 6.2 Events

An Emerson script is driven by a simple event loop. This section discusses the subset of events that the Emerson programming system exposes relevant to the simple indexer example.

#### 6.2.1 Timeout events

Timeout events are the simplest events in Emerson. A scripter registers for a timeout event through a call to the `system.timeout` function:

```
system.timeout(3.5, userCallback);
```

This triggers the `userCallback` function with no arguments after 3.5 seconds.

#### 6.2.2 Proximity events

Applications discover other applications' presences by registering geometric proximity queries:

```
self.setProxQuery(.5, userProxAddedCallback,
                      userProxRemovedCallback);
```

Such queries are standing: they remain registered until explicitly canceled and after returning an initial set of results they continue to stream updates as presences enter or exit the result set. The system manages a proximity result set for each presence. Each presence entering or exiting a presence's result set generates a proximity event on the querying application. If specified in the `setProxQuery` call, callbacks will be invoked when this set changes. Emerson wraps the properties provided by the system, such as address, position, and mesh, into an argument to the callback, called a *Visible*. When a presence is removed from the set its corresponding Visible is marked as invalid. The first parameter adjusts the proximity query's range.

#### 6.2.3 Message Events

Writing distributed applications is conceptually difficult for the reasons mentioned in Section 3.2. Although multi-presencing and sandboxing reduce the amount of distributed programming, they do not remove the problem entirely, and presences must still communicate over the network. In the indexer example, for instance, the indexer's presences must send introduction messages to presences on other applications that it encounters.

At its most basic level, Emerson provides a `sendMessage` function that allows scripters to send messages to presences in the world. Emerson messages are simply objects that are automatically serialized. Section 5.1.2 described extensions to this basic message sending structure, which would make sandboxes within an application addressable. This section describes Emerson-specific syntax that emphasizes the use of simple stop-and-wait protocols to constrain complexity.

During development, we observed that rather than simply sending one-off messages, many applications used request-reply patterns, assigning resources or preparing behavior

in anticipation responses to messages they sent. The Introduction Protocol mentioned in Section refsec:probStatebeginGame is an example of such behavior: the sender of the introduction message listens for a response from the receiver.

Emerson provides special "angle-angle" syntax to accommodate this design pattern. To send a message, `introductionMsgObj`, from `presToSendFrom` to some presence in the world, `somePresence`, a scripter writes

```
presToSendFrom # introductionMsgObj >>
                 somePresence       >>
                 [onRespFunc,timeToWait,onNoRespFunc];
```

The message object sent is automatically tagged with a stream identifier and sequence number. When `presToSendFrom` receives a reply to `introductionMsgObj` (from `somePresence`, with the same stream identifier, and with an incremented sequence number), the reply message is dispatched to `onRespFunc`. If no reply is received after `timeToWait` seconds, the Emerson system executes `onNoRespFunc`, and subsequent messages will not be passed to `onRespFunc`.

To simplify creating replies to messages, the runtime of the receiver writes a `makeReply` function into each received message. Calling `makeReply` with an object as its argument constructs an object with the sender, receiver, sequence number, and stream identifier correctly specified for a reply.

For instance, if a scripter wants to echo all replies to `introductionMsgObj` back to the responder, the scripter's `onRespFunc` would be written as follows:

```
function onRespFunc(replyMsg,replier)
{
   replyMsg.makeReply(replyMsg) >> [];
}
```

The angle-angle message sending syntax encourages applications to interact with other applications through basic stop-and-wait protocols. Although this approach may reduce throughput between message senders, it constrains complexity: there is a unique of messages, reducing the number of states a script must recover from on error.

The Emerson system also provides special syntax for receiving messages. To listen for introduction messages, an indexer's presences register as follows:

```
introMsgHandler << introMsgPattern
```

where `introMsgPattern` is a special *pattern* object, which filters incoming messages, or an array of pattern objects. Patterns have the form:

```
{ field[.subfield]: [value] : [prototype] }
```

which filter individual fields of the message, checking for existence, their value, and their prototype object. Fields in square brackets are optional, so a scripter can, for example, only check the existence, but not value, of a field. As a slightly more complete example, the statement below dispatches the function `introReqCB` on receiving an introduction request message (a message with field `msg` that has value `introRequest`):

```
introReqCB  << {"msg":"introRequest"};

function introReqCB (msg, sender) {
  system.print("Intro received.");
}
```

### 6.3 Putting it together

Figure 7 shows how to compose the simple events described in Section 6.2 to create a full indexer application. The code in the figure assumes a single user-specified array containing basic presence initialization information. Based on this array, the script creates several new presences (line A). When each presence connects to the world, it issues a proximity query (line B). When one of indexer's presences discovers an in-world presence through its proximity query, it sends an introduction request message to the discovered presence and sets a handler that dispatches to `introRepCallback` on replies (line C). Finally, upon receiving an introduction reply message, indexer catalogs the in-world presence that sent it (line D).

## 7. Related Work

The majority of research in scripting languages for worlds is geared towards games or small virtual worlds that are centrally controlled, execute on a single node, and usually developed by professionals. In this section we focus mainly on systems designed for novice programmers and which ease distributed systems programming.

### 7.1 Programming for Novices

Kelleher and Pausch [15] survey and taxonomize programming languages targeted at novice users. The first level of categorization is between *Teaching Systems*, which aim to teach programming for its own sake, and *Empowering Systems*, which aim to make programming accessible in service of other goals, such as entertainment or education in other domains. Emerson falls squarely in the second category: an Emerson user's goal is to build an interesting new experience, not to learn how to program.

Despite their different motivations, teaching systems provide valuable lessons. For example, Emerson is inspired by LOGO's [24] emphasis on accessible power and interactive development and debugging: Emerson scripts can be written in-world and built on dynamically to modify and debug behavior in real time. Emerson is based on JavaScript, which is itself heavily influenced by Self [38]. Self aims for conceptual economy and the programming model fits very well with metaverse-style virtual worlds: there are only objects (no classes, prototype-based), no distinction between properties and methods, and minimal built-in control structures. Prototype-based object-oriented models have previously been used successfully in context of games and vir-

```
var dictionary = {};

//Creates new presences, resulting in one invocation
//of onConnection for each
for (var i = 0; i < initData.length; ++i) {
  system.createPresence(initData[i].worldID,
                        onConnection,
                        initData[i].pos,
                        initData[i].mesh);          (A)
}

// When onConnection executes, self is now connected
// to world.
function onConnection() {
  // register a callback for proximity add events
  // no action is taken for proximity remove events
  self.setProxQuery (.5, proxAddCallback, null);     (B)

  system.timeout(10, updatePosition);
}

// new_pres contains the address and position
// of a presence in the world that satisfies
// self's proximity query.
function proxAddCallback(new_pres) {
  // create and send the actual introduction message
  var intro = {"msg":"introRequest"};
  intro >> new_pres >> [introRepCallback];           (C)
}

// msg contains message object received from sender.
function introRepCallback(msg, sender) {
  var info = {"name": sender.address,
              "pos":  sender.position,
              "meta": msg.metadata};

  dictionary[sender.address] = info;                 (D)
}

// Moves self and schedules another movement
function updatePosition() {
  self.position = self.position + <50,0,0>;
  system.timeout(10, updatePosition);
}
```

**Figure 7.** A simple indexer that collects names and positions for nearby presences.

tual worlds [10, 14], including the text-based predecessors of today's virtual worlds, called MOOs [7].

Alice is a programming environment designed "to engineer authoring systems for interactive 3D graphics that will allow a broader audience of end-users to create 3D interactive content without specialized 3D graphics training" [6], a goal closely related to Emerson's. While many lessons learned in Alice are applicable to Emerson, such as the use of egocentric coordinate systems and intuitive units such as turns/second rather than radians/second, it is targeted at small, local applications and therefore does not address the challenges of scripting for a large scale distributed world.

The Alternate Reality Kit [34], designed in Smalltalk [12], is a system for creating interactive animated simulations. It notes the importance of both "literalism", adhering strongly to a real world metaphor, and "magic," where certain features break that metaphor in favor of simplicity. Although not magic in the user interface sense, Emerson similarly breaks the physical metaphor for simplicity: many presences can be driven by a single script and behavior "owned" by one application is permitted to run on another in a sandbox.

Scratch [25] is a media-rich programming environment designed to increase technological fluency. During early development of Scratch, researchers realized that multi-object interaction made it difficult to share, exchange, and build upon objects because object scripts were not self-contained. They disabled object interaction and hoped to create more loosely coupled object interaction. Emerson partially addresses this problem through multi-presencing, which allows all code that must be executed together to be contained in a single script. The introduction protocol corresponds to the loosely-coupling and dynamic interaction they aimed for.

Finally, Brandt et al. [3] note that even much professional development is "opportunistic," emphasizing speed and ease of development. They hope to build tools which better emphasize this style. While Emerson has focused on a language features which ease development, improving tools and the programming environment are important areas of future work.

### 7.2 Implementation

Much research has attempted to address the challenges of distributed programming [16, 18, 40]. However, we recognize that distributed programming is still challenging even for expert developers. Therefore, Emerson tries to minimize the amount of distributed programming performed by using multi-presencing and enabling remote code execution in sandboxes to convert previously parallel code into simple, single-threaded code.

As mentioned earlier, Emerson's multi-presencing model is similar to vats in E. Each object in an E vat is separately addressable from external vats and objects within a vat can cooperate directly amongst themselves. Emerson builds on this basic abstraction, automatically managing relevant presence with `self`.

In addition to multi-presencing, Emerson reduces distributed programming is by allowing execution of code from other applications. Emerson accomplishes this by sandboxing, a technique also available in other languages, such as Lua [19]. However, in many cases sandboxes are short-lived: Lua sandboxes only exist for the lifetime of a function call. Further, Emerson aims for simplicity by requiring that capabilities for sandboxes are specified up front and are static, whereas many other sandboxes allow dynamic changes to capabilities.

Sandboxing has also been used in the context of virtual worlds. One example is `llRemoteLoadScriptPin` [28] in Second Life, which allows different applications to exchange scripts. However, this mechanism is extremely limited, essentially requiring a pre-existing relationship be-

tween the source and host of the sandbox. Further, Second Life has an unusual model where multiple, independent scripts can execute on the same object. The existing script is not aware of the new one, and vice versa, so there can be no interaction between them. Finally, once accepted, these scripts receive complete control over the object: there are no capabilities or permissions on sandboxes. This is commonly used to upgrade scripts to new versions, but much less commonly for dynamically exchanging scripts between objects with different owners.

Alternatives to sandboxing include language-based information flow [5, 26] and dynamic taint analysis [22], which can enforce data flow policies to ensure sensitive data does not escape an application or is not accessed by untrusted code. Sandboxing may seem heavy-handed compared to these alternatives, but it is simpler to understand and use, a more important property for Emerson.

Object capabilities have been applied in the context of virtual worlds to provide fine-grained, flexible permissions [27]. Similar ideas could be applied in Emerson to control access to services via messaging. For example, in the chess game a message to perform a move might require a capability token, generated securely by the Emerson runtime. This might be an intuitive way to encourage good security for novice application developers.

Where Emerson is unable to remove distributed programming, it builds on existing models and languages. Applications are essentially actors [1], independent threads of execution which communicate via asynchronous messaging. Emerson's event matching is similar in spirit to ActorSpaces [2], but the latter focuses on filtering applications whereas Emerson focuses on filtering events. Erlang [40] processes use a similar pattern matching strategy to handle messages received from other processes. Erlang supports selective receive, but suffers from large receive pattern matching blocks.

## 8. Conclusion

This paper describes Emerson, a programming system for user-extensible virtual worlds targeting novice users. In particular, metaverses, in which any user can script objects, present a challenging scripting environment: they must be distributed to scale to millions of users and common interactions involve two mutually untrusting applications.

Emerson simplifies programming for this environment in two key ways. First, unlike most systems which tie each script to a single presence in the world, Emerson supports multi-presencing. By controlling multiple presences from a single script, code which is distributed becomes local and single-threaded. This avoids many challenges in writing distributed programs including inconsistent state, message loss, and message reordering. Second, Emerson enables local execution of untrusted code in sandboxes. This approach reduces the amount and complexity of messaging in an application by allowing some processing to occur on the "client" of the application. It also addresses the challenge of protocol agreement because it enables bootstrapping of new interactions with a minimal introduction protocol.

However, these abstractions only provide a starting point: the Emerson programming system is an active research project, and still undergoing active development. A small team of undergraduates with varying programming backgrounds have begun developing applications in Emerson and their experiences have informed and will continue to inform development.

The most immediate area for future work on Emerson is on the language itself. Simple extensions such as cleaner syntax for mathematical operations and a more powerful, intuitive interface for movement are oft-requested features. Additionally, the introduction protocol described in Section 3.1 is particularly useful for avatar-based interactions where an end-user can make decisions about what sandboxed code to run and with what capabilities. Encouraging interaction between non-avatar applications may require a simplified mechanism for registering basic services and categorizing other presences in the world by which services they provide, and we are considering incorporating ideas from AmbientTalk's service discovery protocol to this end [8]. Finally, we are examining ideas from event specification languages to allow scripters to declaratively compose and specify events that lead to callbacks.

We recognize that for complicated applications, it may become difficult to maintain code with so many callbacks. Solutions that obscure callbacks, for instance that provide special wrappers or syntax so that code deferred to be executed on an event appears as straight-line code can reduce code re-use. Although we will examine these schemes going forward, our principle approach to address this maintenance problem will be to build libraries that automate many common sources of callbacks. The request-reply message syntax described in Section 6.2.3 is a good example of our early efforts in this domain.

Additional challenges are emerging in protocol development. The introduction protocol, described in Section 3.1 is primarily designed for interaction involving a human avatar. Enabling intelligent interaction between an arbitrary pair of scripted objects will likely require a narrow set of additional well-known protocols.

Finally, in addition to such new features, additional work remains on improving those already discussed. For instance, although this paper has discussed sandboxes strictly from the perspective of trust, the data isolation they provide may also be useful for code organization and trusted collaborative coding.

## Acknowledgments

We thank the National Science Foundation, which, through Graduate Research Fellowship #2007050798, partially supported this work.

## References

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems*. 1986.

[2] G. Agha and C. J. Callsen. Actorspace: an open distributed programming paradigm. In *Proc. of Principles and Practice of Parallel Programming*. ACM, 1993.

[3] J. Brandt, P. J. Guo, J. Lewenstein, and S. R. Klemmer. Opportunistic programming: how rapid ideation and prototyping occur in practice. In *Proceedings of the Workshop on End-user Software Engineering*. ACM, 2008.

[4] B. Chandra, E. Cheslack-Postava, B. F. T. Mistree, P. A. Levis, and D. Gay. Emerson: Scripting for federated virtual worlds. In *Proceedings of the 15th International Conference on Computer Games: AI, Animation, Mobile, Interactive Multimedia, Educational and Serious Games (CGAMES)*, July 2010.

[5] S. Chong, K. Vikram, and A. C. Myers. SIF: enforcing confidentiality and integrity in web applications. In *USENIX Security*, 2007.

[6] M. Conway, S. Audia, T. Burnette, D. Cosgrove, and K. Christiansen. Alice: lessons learned from building a 3d system for novices. In *Proc. CHI*, 2000.

[7] P. Curtis. *LambdaMOO Programmers Manual*. 1993.

[8] T. V. Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, and W. D. Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *Proceedings of the XXVI International Conference of the Chilean Society of Computer Science*, 2007.

[9] Economic Spotlight: L$ Exchange Rate in Q2 2010. `http://community.secondlife.com/t5/Featured-News/bg-p/blog_feature_news/date/9-5-2010`.

[10] P. Emmerich. *Beginning Lua with World of Warcraft Add-ons*. 2009.

[11] EvE Online. `http://www.eveonline.com/ingameboard.asp?a=topic&threadID=879995`.

[12] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. 1983.

[13] Habbo Hotel. `http://www.habbo.com/`.

[14] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The evolution of lua. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 2007.

[15] C. Kelleher and R. Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37, June 2005.

[16] M. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 2007.

[17] Linkability Rules. `http://wiki.secondlife.com/wiki/Linkability_Rules`.

[18] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proc. Programming Language Design and Implementation*. ACM, 1988.

[19] Lua Wiki. `http://lua-users.org/wiki/SandBoxes`.

[20] L. Magid. Learning architecture in a virtual world, September 2007. URL `http://www.cbsnews.com/stories/2007/09/18/scitech/pcanswer/main3271133.shtml`.

[21] M. S. Miller, E. D. Tribble, J. Shapiro, and H. P. Laboratories. Concurrency among strangers: Programming in e as plan coordination. In *In Trustworthy Global Computing, International Symposium, TGC 2005*. Springer, 2005.

[22] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[23] Online Therapy Institute. `http://www.metaversejournal.com/2009/03/30/interview-deeanna-nagel-and-kate-anthony-online-therapy-institute/`.

[24] S. Papert. *Mindstorms: children, computers, and powerful ideas*. 1993.

[25] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: programming for all. *Commun. ACM*, 2009.

[26] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas In Communications*, 2003.

[27] M. Scheffler, J. P. Springer, and B. Froehlich. Object-capability security in virtual environments. In *VR*, 2008.

[28] Second Life. `http://wiki.secondlife.com/wiki/LlRemoteLoadScriptPin`.

[29] Second Life Games. `http://secondlife.com/destinations/games`.

[30] Second Life Link. `http://wiki.secondlife.com/wiki/Link`.

[31] Second Life Shakespeare. `http://slshakespeare.com/`.

[32] Sirikata. `http://www.sirikata.com/`.

[33] SLAN. Second life art news: News about the arts for second lifers. `http://sl-art-news.blogspot.com/`, 2005.

[34] R. B. Smith. Experiences with the alternate reality kit: an example of the tension between literalism and magic. *SIGCHI Bull.*, 18, May 1986.

[35] G. Staff. GDC Austin: An inside look at the universe of warcraft. `http://www.gamasutra.com/php-bin/news_index.php?story=25307`, September 2009.

[36] T. Sweeney. UnrealScript language reference. `http://udn.epicgames.com/Three/UnrealScriptReference.html`, 2001.

[37] The Sims. `http://thesims.ea.com/`.

[38] D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-*

*oriented programming systems, languages and applications*. ACM, 1987.

[39] A. van Kesteren and World Wide Web Consortium. Cross-origin resource sharing: W3c working draft. `http://www.w3.org/TR/2010/WD-cors-20100727/`, 2010.

[40] C. Wikström. Distributed programming in erlang. In *In PASCO'94 - First International Symposium on Parallel Symbolic Computation*, 1994.

[41] D. Williams, N. Ducheneaut, L. Xiong, Y. Zhang, N. Yee, and Y. Nickell. From tree house to barracks: The social life of guilds in world of warcraft. *Games and Culture*, 2006.

[42] World of Warcraft. `http://us.battle.net/wow/`.