Emerson: Scripting for Federated Virtual Worlds

Bhupesh Chandra*, Ewen Cheslack-Postava*, Behram F.T. Mistree*, Philip Levis*, and David Gay†
*Stanford University
Email: {bhupc, ewencp, bmistree}@stanford.edu, pal@cs.stanford.edu
†Intel Labs
Email: dgay@acm.org

***Keywords*—Virtual Worlds, Scripting Languages, Scalability, Federation, Exploratory Programming**

***Abstract*—We introduce *Emerson*, a scripting language for virtual worlds that are seamless, scalable, and federated. These worlds present a number of unique challenges. Most importantly, scripts that specify the behavior of the world are distributed across many hosts and users may generate and host scripts. These constraints imply features not common in other systems, such as frequent use of asynchronous message passing for basic interaction between entities, a lack of trust between entities inhabiting the world, and live-editing of entities in the world. Emerson addresses these challenges with three core design concepts: entity-based isolation and concurrency, an event driven model with concise and expressive pattern matching to find handlers for messages, and strong support for example-based programming within the live virtual environment. Our prototype implementation of Emerson, based on the V8 JavaScript engine, demonstrates that a variety of applications can be easily written in Emerson in a live system.**

I. Introduction

"Build, therefore, your own world."
- Ralph Waldo Emerson
Nature

Many users of today's virtual worlds yearn for an open, user-extensible platform. Although some make do with the features of closed, proprietary games (Machinima Inc. 2004; World of Warcraft Movies 2010), users of more open worlds have developed a variety of novel applications, ranging from virtual art, theatre, and music performance (SLAN), to collaborative visualization (Intel Corporation 2009), mixed reality art (Leeson 2008), and distance learning (Magid 2007). This trend towards open, extensible platforms mirrors the way the web replaced closed content providers such as Prodigy and CompuServe in the mid 90s. While closed, proprietary platforms will always have their place, this trend suggests that users desire open, extensible, federated worlds that serve as application platforms.

We believe virtual worlds, like the web, must be seamless, scalable, and federated to address this demand. **Seamless** virtual worlds provide a single, continuous space in which all entities interact instead of sharding entities into disjoint, non-interacting sets. A **scalable** world supports millions or billions of objects. In a **federated** world anyone can host content and add it to the world, just as anyone can run a web server. Some virtual worlds have made great strides toward the application platform model, however none support all these features. Second Life, for instance, provides a seamless and scalable virtual world, but does not support federation.

While existing work explores system design challenges for such a platform, such as scalable architectures (Horn et al. 2009) and entity fault tolerance (Vaz Salles et al. 2009), no existing work addresses the language design necessary to support such a platform model. This paper introduces ***Emerson***, a new scripting language named after Ralph Waldo Emerson, a 19th century Transcendental philosopher who emphasized the spiritual importance of all objects in the universe. To him, carrots, rocks, and people were all imbued with the same spiritual facilities and powers.

The Emerson scripting language tackles the challenges imposed by a seamless, scalable, federated virtual world for creating behaviors for entities – physical items, avatars, buildings, and creatures. Entities connect to a world and interact with it via several core services, such as location, movement, and communication. Just as existing system architectures do not translate to this model, we believe that existing scripting languages are ill-suited for the challenges of a virtual world application platform.

Three challenges make this problem different from scripting for existing virtual worlds. First, both novice and expert users will develop entities incrementally in the live system. Thus the scripting language must support iterative, exploratory, and interactive development. Second, because the world is federated, interacting entities may not trust each other. They cannot operate on each other's state directly. Finally, scalability and federation require distributed simulation. Therefore, interacting entities may be on different hosts, introducing lossiness and latency.

While each of these points has been addressed individually by other systems, the combination is unique to this application. Emerson addresses these challenges by building upon the following core concepts:

- example-based programming in a live environment
- entity isolation and concurrency
- event-driven model with event pattern matching

With these concepts, users can create new entities using another as a template; add features to that entity to make it responsive to dynamic events; and easily match events to the desired response behavior.

In the remainder of this paper, Section II gives a detailed description of the features of Emerson, Section III describes our current prototype, Section IV discusses related work, and Section V concludes.

II. Language Design

The syntax of Emerson is based on JavaScript, an easily embeddable, prototype-based, object-oriented and functional language. JavaScript has proven accessible to novice developers like web designers, and its syntax and design patterns should be familiar to experienced developers. Each entity contains a single Emerson context, which stores the current script state and can be used to execute Emerson commands. Scripts are dynamic, so users can edit entities interactively by executing additional Emerson commands.

Emerson is event driven. Its runtime invokes event handlers in response to events such as timers or message receptions. The language explicitly distinguishes message passing and asynchronous events from local, synchronous operations made with function calls. Emerson entities are single-threaded, executing only one event handler at a time. The benefits (such as easy horizontal scalability and the relative simplicity of single-threaded programming) and limitations (such as inconsistency between entities) of this approach are well studied, both theoretically and practically (Abdallah et al. 2005; Agha 1986).

The rest of this section describes the unique features of Emerson. Code snippets from an art gallery script demonstrate these features. The gallery loans virtual art to users in the world. Figure 1 shows the components of this entity as well as how it connects to multiple worlds. The examples will refer back to Figure 1 throughout.

A. Programming By Example

Emerson is designed for virtual worlds where all users can be content authors. Many games, such as World of Warcraft (Gamasutra 2009), hire dozens of professional developers to provide in-world content. In contrast, both professional developers and end users who want to extend and improve the world write Emerson entities. As a result, a non-user-friendly scripting language may result in unfeatured blankness rather than a dynamic and engaging virtual world.

To address the needs of novice users and encourage innovation and experimentation by experienced users, Emerson is strongly oriented towards exploratory programming. Specifically, Emerson's entity prototyping and dynamic language features support this style of development.

Entity Prototyping A growing corpus of literature suggests that a common way a programmer both learns and programs is through a "copy-and-paste" model (Kim et al. 2004; Brandt et al. 2009): a programmer finds extant snippets of code, copies them into her development environment, and tailors them to suit her applications.

While any programming language trivially supports textual copy-and-paste programming, Emerson more deeply incorporates the spirit of copy-and-paste coding through entity prototyping. Users create new entities by cloning an existing entity as a starting point. In contrast to other languages which use delegation for prototyping, such as Smalltalk and Self, Emerson must copy the entire entity to preserve efficiency. This is necessary because entities in Emerson are distributed, so the clone may not be on the same host as the original. In our example, the artist might begin building her gallery by first finding a building suitable for her purpose – a museum, an apartment, or a home. In Figure 1, the majority of the contents of the entity, including objects and handlers, would have been copied directly from the original. She then customizes the clone to suit her needs, adding, for instance the `openGallery` function and registering it as an event handler. Although the artist's gallery defaults to being copyable, she can change it to non-copyable to protect sensitive information or her intellectual property.

Because objects, as described in Section II-B, can be serialized and passed in messages, prototyping occurs at the object level as well. This encourages code reuse and permits sharing of difficult-to-program objects. For instance, instead of writing her own account management scheme, our artist could copy a constructor for a complicated account management object into her gallery entity, instantiate a copy of it, and add an event handler which uses the account manager to handle transactions. In Figure 1, this might include copying the `handleLoan` method.

Dynamic Language Features Rapid prototyping and iterative design play important roles in exploratory programming and real system development (Gordon and Bieman 1993). Emerson provides a number of dynamic language features which enable this process while working in the live system.

Most importantly, the `eval` keyword in Emerson supports dynamically adding and modifying code. Using just `eval` and messages, users can extend scripts with code updates passed in messages. For instance, an entity, such as our artist's gallery, will accept script messages and evaluate them in its context by default. The artist can build the behavior of her gallery incrementally by sending commands to it. In practice, a GUI hides this process, providing a simple command prompt for the entity. The artist first adds basic data structure objects and instantiates them, possibly copying them from other entities. Then she sets several variables, such as item descriptions, and registers event handlers that handle interaction with customers, as discussed in Section II-C. Later, when she needs to apply a bug fix, she simply sends additional commands to update the gallery's methods and state.

B. Entities, Presences, Objects

Federation and scalability, and the distribution implied by both, introduce two challenges not commonly faced in most other systems. First, in a federated world, two interacting entities may not be controlled by the same administrator. This means that they do not trust each other and it would not be safe to allow one to directly inspect and modify the state of the other. Second, multiple hosts execute entities, both for scalability and because users may host their own content. As a result, entities must handle network latencies and message delivery failures.

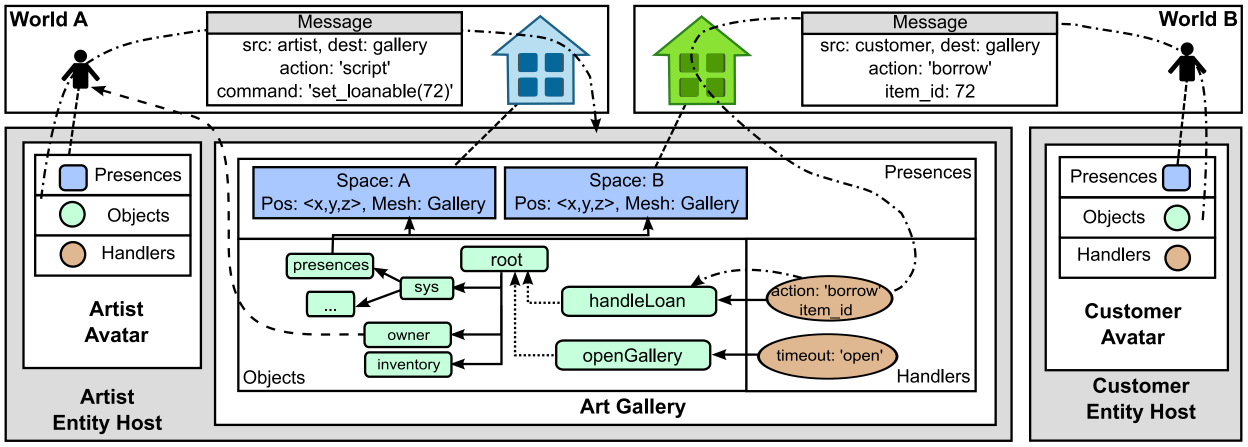

Fig. 1. An example of Emerson's components and how they interact in multiple worlds. The figure shows three entities (shown bottom left to right, an artist avatar, her gallery, and a customer) with presences in two spaces (top, the artist in World A, the gallery in both, and the customer in World B). The gallery has different meshes and locations in the two worlds. The artist entity host runs the artist avatar and gallery; another runs the customer avatar. Only the gallery's internal details are shown. The gallery has two presences, two event patterns and handlers, and internal object state, anchored by a root object. In World A, on the left, the vendor sends a message to her gallery to set an item to be loan-able to customers. This change is reflected in both worlds. In World B, on the right, the customer sends a message to borrow an item from the gallery. This shows the complete path of the message: from the customer entity host, routed by World B from the customer's presence to the gallery's, matched to an event pattern, and finally dispatched to the event handler, handleLoan.

To address these challenges, Emerson distinguishes between *objects*, used for local, trusted, synchronous operations and *presences*, used for remote, untrusted, asynchronous, and geometric operations. Each entity runs an Emerson script, which contains objects and, through the entity, connects to worlds to obtain presences.

Figure 1 shows how these components, with event handlers (discussed in the next section), make up an entity and how they interact with the world. In the figure, both the artist and her gallery are entities: they do not share state and any messaging between them is asynchronous – for instance, if the the artist wants to send a message to her gallery to update the gallery's inventory, she does so asynchronously. As seen in Figure 1, the artist has a presence in World A, while the gallery has a presence in both Worlds A and B.

Objects do not have a geometric representation in the space and constitute the internal state of an entity. In the example, the gallery's `inventory` is an object. Objects exist only in the context of an entity and are not addressable from outside the entity. Therefore operations on them are always local and use synchronous method calls. In the example, the inventory is only accessible within the gallery entity.

Figure 1 shows how entities and objects differ. If the artist wants to begin lending a piece to patrons that she previously kept private, the gallery entity's inventory object must be updated. Because the inventory is an object contained by the gallery entity, the artist entity may not directly change it. Instead, she sends an asynchronous message to the gallery entity. The gallery decides how to process the request, either honoring it and altering the state, or ignoring it. Access control is an important aspect of the system, but we do not currently incorporate it into the language.

The following code, based on Figure 1 modifies the gallery's inventory list as described. The inventory is a map of presences for available art entities.

```
set_loanable = function(item_id) {
  piece = inventory.find(item_id);
  if (piece != undefined)
    piece.loanable = true;
}
```

This code shows the use of objects by creating a `set_loanable` function. Emerson uses dot-notation for synchronous method calls and field accesses, as shown by the call to `inventory.find`. A *root object* serves as the last possible location for field lookup and assignment. Here, `set_loanable` is added to the root object because it was executed in the global context. Also, the `inventory` refers to the field in root object because it is not prefixed by a specific object. As shown in Figure 1, the Emerson runtime also provides a `sys` field in the root object to expose entity host functionality such as timers, access to world content, and `http` requests.

Presences give an entity geometric properties, such as location, velocity, and mesh. Entities own their presences while holding references to the presences of other entities. Scripts use both kinds of presences, which contain the presence's address in the world, to send and filter messages, as well as lookup geometric properties of the associated entity. Entities also use presences to update their own geometric properties. Entities may have multiple presences in order to bridge worlds, with unique representations in each world. The example gallery has two presences, which it uses to service both worlds from the same inventory. As shown in the figure, these two presences have different meshes.

Operations on presences always require communication over the network. Therefore, these operations are asynchronous and use distinct syntax from the object dot-notation used for synchronous operations. We draw inspiration from Erlang, which similarly distinguishes between operations within processes and sending messages between processes. In Emerson, the operators `<-` and `->` indicate incoming and outgoing operations on presences, respectively. The former is discussed in Section II-C. An example of an outgoing operation is message passing:

```
  Message -> Presence [ -> ErrorHandler ]
```

where `Message` is a serializable object. Unlike method calls on objects, sending a message could fail long after initiated (e.g. a timeout). Therefore, the above statement has an optional error handler callback, which uses the same asynchronous syntax. Emerson uses a reliable, in-order protocol by default, but the underlying unreliable protocol can be accessed by advanced scripters.

The gallery, upon receiving a loan message, would reply with a message confirming the customer's loan:

```
receipt = new Receipt(status='OK',
                      item=item_id);
receipt -> current_customer_id ->
                 confirmation_failed
```

Emerson can use any serializable object as a message, as `receipt` is here. This gives the user flexibility in how arguments in messages are specified (an array for a traditional method interface or an object using named fields as named parameters).

C. Events and Event Handling

Emerson provides a unified event model and an event matching syntax to make handling events efficient, simple and flexible. Each event in Emerson is represented as a single object. Special syntax creates event handlers. Handlers are described declaratively: a pattern is used to match events and map them to an associated handler. Handler rules can be added or removed individually at any time, and events may cause multiple event handlers to be invoked. Besides being easier to use, this approach allows the runtime to optimize message dispatch, similar to the optimization possible in SQL query processors.

In our example, the gallery must handle a variety of events: open in the morning, attract customers when they come nearby, handle loans, and close up shop in the evening. The artist may build this functionality incrementally, first declaring handlers for `look` and `borrow` events, then adding code to handle opening and closing the gallery. The `borrow` event only needs to be handled if it contains an item identifier and comes from a nearby object. The artist simply adds event handler rules as she writes the handler functions for them.

The syntax for event handler registration is

```
   Handler <- [Pattern]  [ <- Presence]
```

where elements in brackets are optional. Handler is a method which is invoked if the event matches Pattern. Because matching events with particular source entities is expected to be common, it has dedicated syntax.

Patterns allow matching fields of events (and fields of fields, etc.) on three properties: name, prototype, and value. Each rule takes the form:

```
 [proto] field[.subfield[...]] [: value]
```

where the patterns are sets of rules:

```
  (rule1, rule2, ...)
```

The `proto` component in the rule compares the structure of the object to the given prototype. This is a simple way to quickly match common structures. For instance, specifying `Vector3` as a prototype is equivalent to checking for `x`, `y`, and `z` fields. Our experience thus far suggests that these rules cover the vast majority of cases.

In our example, the gallery registers the following events:

```
handleLoan <-      (action: 'borrow',
                    item_id) <- customer;
openGallery    <- (timeout: 'open');
closeGallery   <- (timeout: 'close');
```

The first handles a loan request message, sent by a customer. It invokes `handleLoan` when a message from `customer` has: a) a field with name `action` and value `'borrow'`; and b) a field with name `item_id`. The last two are handlers for timeouts, generated by the system, and handle opening and closing the gallery. As described above, these lines may actually occur far from each other in the script.

III. Prototype

We have implemented a prototype of the Emerson language in Sirikata (Sirikata), a platform for seamless, scalable, and federated virtual worlds. The prototype is an entity host scripting plugin built on the V8 JavaScript engine (Google 2008). A single V8 instance runs all Emerson scripts on an entity host, assigning a separate context to each entity. Contexts are light-weight, so using one per entity is efficient and automatically provides isolation of Emerson scripts.

The prototype exposes entity host basics such as world connection management and timers; basic presence functionality such as setting the mesh, position, velocity; and simple messaging as described in Section II. A GUI allows users to interactively work with entities, exploring their state and modifying their behavior. Our prototype enables this ability by default. Any commands a user issues modifies running entities without rebooting them.

We have implemented the example described in this paper and others to test our prototype. Although incomplete, the prototype can be used to implement a variety of applications. Besides filling in gaps, we are investigating additional language features and building a standard library for this Emerson prototype.

IV. Related Work

Although targeting different problems and features, White et al. (White et al. 2009) details the effects of scripting language design on game development, and establishes the importance of an efficient, usable scripting language. Prototype-based object-oriented models have been used successfully in context of games and virtual worlds (Ierusalimschy et al. 2007; Emmerich 2009).

Work such as Miryung et al. (Kim et al. 2004) and Brandt et al. (Brandt et al. 2009) inspire Emerson's emphasis on copy-and-paste support. Popular languages such as the Linden Scripting Language and UnrealScript require recompilation and entity reboot for code updates. Alternate Reality Kit (Smith 1987), designed in Smalltalk (Goldberg and Robson 1983), influenced Emerson's provisions for live code modification.

Emerson's concept of entities is similar to the cells in Project Red Dwarf (formerly Darkstar) (RDS). However, our system stores geometric properties in the space, supports federation of entities besides avatars, and does not assume centralized persistent storage for entities. Emerson's event matching is similar in spirit to ActorSpaces (Agha and Callsen 1993), but the latter focuses on filtering entities whereas Emerson focuses on filtering events. Erlang (Virding et al. 1996) 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.

V. Conclusion

This paper presents Emerson, a scripting language that addresses the challenges of describing entity behavior in seamless, scalable, and federated virtual worlds. Emerson supports a wide spectrum of developer aptitudes; event driven programming as the norm; and a distinction between local, trusted objects and remote, untrusted, geometric entities. Language features such as entity and object cloning, entities and presences, and event pattern matching directly address these challenges. Examples such as the art gallery described throughout, built in our prototype, suggest Emerson is powerful and expressive enough for many applications.

Beyond completing the prototype, we believe there are further challenges presented by a virtual world application platform that could be addressed by the scripting language. Persistent storage for entity state has been addressed in previous work (Vaz Salles et al. 2009), but isolation of entities suggests this issue should be revisited. Security is another challenge, ranging from how to secure an entity from other avatars, to more complex scenarios such as virtual transactions in untrusted, federated systems. We are actively pursuing these extensions.

Acknowledgments

We thank the developers we surveyed for insightful discussions about the unique challenges of scripting for virtual worlds. In addition, we thank the National Science Foundation, which, through Graduate Research Fellowship #2007050798, partially supported this work.

References

Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors. *Communicating Sequential Processes: The First 25 Years, Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004, Revised Invited Papers*, volume 3525 of *Lecture Notes in Computer Science*, 2005. Springer. ISBN 3-540-25813-2.

Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

Gul Agha and Christian J. Callsen. Actorspace: an open distributed programming paradigm. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 23–32, New York, NY, USA, 1993. ACM. ISBN 0-89791-589-5. doi: http://doi.acm.org/10.1145/155332.155335.

Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Opportunistic programming: Writing code to prototype, ideate, and discover. *IEEE Software*, 26:18–24, 2009. ISSN 0740-7459. doi: http://doi.ieeecomputersociety.org/10.1109/MS.2009.147.

Paul Emmerich. *Beginning Lua with World of Warcraft Add-ons*. Apress, Berkely, CA, USA, 2009. ISBN 1430223715, 9781430223719.

Gamasutra. GDC Austin: An inside look at the universe of warcraft, September 2009. URL http://www.gamasutra.com/php-bin/news_index.php?story=25307.

Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. ISBN 0-201-11371-6.

Google. V8 javascript engine. http://code.google.com/apis/v8/, 2008.

V. Scott Gordon and James M. Bieman. Reported effects of rapid prototyping on industrial software quality. *Software Quality Journal*, 2(2):93–108, 1993. ISSN 0963-9314.

Daniel Horn, Ewen Cheslack-Postava, Tahir Azim, Michael J. Freedman, and Philip Levis. Scaling virtual worlds with a physical metaphor. *IEEE Pervasive Computing*, 8:50–54, 2009. ISSN 1536-1268. doi: http://doi.ieeecomputersociety.org/10.1109/MPRV.2009.54.

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of lua. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1–2–26, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-X. doi: http://doi.acm.org/10.1145/1238844.1238846.

Intel Corporation. ScienceSim: A virtual environment for collaborative visualization and experimentation. White paper, 2009. URL http://techresearch.intel.com/UserFiles/en-us/File/terascale/ScienceSimWP.pdf.

Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in oopl. In *ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 83–92, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2165-7. doi: http://dx.doi.org/10.1109/ISESE.2004.10.

Lynn Hershman Leeson. Life to the power of N. http://www.life-n.net/, 2008.

Machinima Inc. Machinima.com: The entertainment network for the gaming generation. http://www.machinima.com/, 2004.

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

RDS. Red Dwarf Server. http://www.reddwarfserver.org/, 2010.

Sirikata. Sirikata virtual world platform. http://www.sirikata.com/, 2010.

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

Randall B. Smith. Experiences with the alternate reality kit: an example of the tension between literalism and magic. In *CHI '87: Proceedings of the SIGCHI/GI conference on Human factors in computing systems and graphics interface*, pages 61–67, New York, NY, USA, 1987. ACM. ISBN 0-89791-213-6. doi: http://doi.acm.org/10.1145/29933.30861.

Marcos Vaz Salles, Tuan Cao, Benjamin Sowell, Alan Demers, Johannes Gehrke, Christoph Koch, and Walker White. An evaluation of checkpoint recovery for massively multiplayer online games. *Proc. VLDB Endow.*, 2 (1):1258–1269, 2009. ISSN 2150-8097.

Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996. ISBN 0-13-508301-X.

Walker White, Christoph Koch, Johannes Gehrke, and Alan Demers. Better scripts, better games. *Commun. ACM*, 52(3):42–47, 2009. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1467247.1467262.

World of Warcraft Movies. Warcraftmovies.com - world of warcraft movies. http://www.warcraftmovies.com/, 2010.