Avatars and their embodiments - MIDI devices, their past and future

Colin Clark
OCAD University
cclark@ocadu.ca

Antranig Basman
Raising the Floor
amb26@ponder.org.uk

Abstract

We will situate the concept of an **avatar** *(a working simulacrum of part of a system separated from it in space or time) with respect to traditional concepts of programming language and systems design. Whilst much theory and practice argues in favour of* **insulation** *(the creation of architectural boundaries prohibiting the leakage of information) we will find that many successful systems take a diametrically opposed approach. We name this family of systems as those based on* **externalised state transfer**. *Rather than hiding implementation details behind APIs, object interfaces or similar, these systems actively advertise their internal structure and its coordinates via data and metadata. Examples of these systems include RESTful web applications, MIDI devices, and the DWARF debugging file format. We discuss such systems and how we can purposefully design new systems embodying such virtues in a more distilled form.*

1. Avatars

An avatar is a part of a system of which a working simulacrum can be embodied elsewhere. The most common appearance of this pattern is in supporting a remote UI, whereby a single system exposes the potential for control to different locations, separated by a network of some kind. To the extent that the remote system might be agnostic as to whether the controlled system is local or remote, it could be said to be working against an avatar, or functioning local replica, of the controlled system. However, the avatar pattern isn't the only possibility for implementing such a remote UI — it is just one of the most convenient. It would be perfectly possible to imagine the remote UI being bound to the controlled system by a message bus along which passed essentially arbitrary messages, a remote procedure call (RPC) idiom that simply stood in for whatever function calls the two parts of the architecture found it necessary to make to each other from time to time.

However, it's not accidental that by far the most prolific technology for supporting remote UIs, the web, is also the birthplace of a software idiom, REST (Fielding & Taylor, 2000), which is tightly bound up with the avatar model. In REST, "representations of state are transferred" — that is, the remote system does not simply answer arbitrary message, but instead guarantees to transfer complete (exhaustive) representations of part of its application state to the UI client. The expectation is that the client then operates on this state in the form of a local avatar of some greater or lesser degree of faithfulness, as a functioning replica of some aspect of the remote system.

However, there are numerous other productive uses of the avatar model, which need not be limited to scenarios where the separation is in space. The avatar model could be used to support advanced live programming scenarios (Basman, Church, Klokmose, & Clark, 2016), where a system which is being authored can be asked, explicitly or implicitly, to speculate about states it might reach in the future as a result of different authorial gestures. This supports the highest level of liveness (L6) in (Church, Söderberg, Bracha, & Tanimoto, 2016)'s classification of live programming paradigms. In this case, part of the current system becomes an avatar for part of its state in the future — it is a working simulacrum allowing the user to see some or all of its behaviour, but which is integral in itself and may be immediately discarded without any side effects on the behaviour of the current system.

Almost all current programming languages, their philosophies (whether procedural, functional, or object-oriented), toolchains and implementation technologies (compilers, virtual machines) work to ob-

struct the efficient creation of avatars. We assert the typical bent of deliberate thought by software engineers and computer scientists is to only cater for cases where a system is operated on from inside itself, and give no thought to its participation in wider authorial networks where it is embedded in wider contexts of use, by clients separated in space and time, using different languages, idioms and technologies.

In practice, whenever artefacts are embedded in a particular context of use where avatar-style working is strongly required, particular technological patterns have emerged, often informally, which support this style of working. These patterns are either overlooked or if they are observed, are usually decried by traditional software technologists as being antithetical to their values ("The web is broken!", as seen in (Tiselice, 2015) and others). These patterns support the avatar model by purposing the messages which are transported between systems as "state transfer" in the REST sense. The fact that these messages are supported have implications for how the remote architecture is seen by others, and usually also on how it sees itself — rather than lying in opaque "black boxes" beloved of the proponents of information hiding (Parnas, 1972), object orientation, or the API model, the architecture is instead coordinatised (Basman et al., 2016), with its elements laid out in a spatial grid or tree, each element of which has well-known coordinates and whose state is in theory available for inspection and modification at any time.

2. MIDI Devices

MIDI-controlled musical devices (synthesizers, drum machines, sequencers, and patch editors) have proliferated over the last 35 years; stretching from the Yamaha DX7 built in 1983, to an iOS-based patch editor called Patch Base used to "program" the DX7, which was updated by its developer only days ago. In this case, the editor's UI acts as an avatar for the ancient synthesizer, controlling it over a stream of MIDI messages whose format has not changed over this timespan. What gives some MIDI devices this longevity, which seems so unlike today's rapidly-obsolescing general-purpose software products? It is common for there to be a variety of available ways to control, inspect, and to "program" a MIDI device from the outside, using tools that are more often than not created independently by third-parties or competing vendors.

These usually take the form of System Exclusive (sysex) editors and libraries, which provide a means for external integration with a MIDI device. MIDI sysex is essentially accidental, and it was not designed expressly as a means for supporting multiple points of control and representation — but it has a key quality that is divergent from most modern software: it provides a fully externalized view of the entire device's state, and this state can be operated on, in whole or in part, externally. This view of state, however, is bizarrely anachronistic and low-level (again, perhaps attributable to the extreme resource constraints of embedded hardware of the era), but it offers a view into a "characteristic of the machine". The MIDI sysex protocol allows for the exchange of essentially arbitrary streams of bytes, and offers the barest formality for framing these byte streams and identifying their sources. One might expect this lack of constraint to result in arbitrary, RPC-like idioms for message encoding and interpretation. Instead, the dominant idiom for these messages is to simply transfer certain sections of the device's memory to and from the host.

This view of state is bizarrely anachronistic and low-level (again, perhaps attributable to the extreme resource constraints of embedded hardware of the era), but it offers a view into a "characteristic of the machine". Contrary to the proponents of reuse who promote insulation and information hiding as the primary routes to this goal, it is devices based on this idiom which has proved the most durable and reusable in historical practice. By contrast, in the world of iOS apps, there is the example of the Roland Sound Canvas app which claims to "emulate perfectly" its popular hardware namesake from the 1990s, yet specifically omits the sysex implementation of the Roland GS specification, preventing it from being externally introspected or configured; here, in the realm of sophisticated and polished user interfaces, there is no longer a means to alternatively control, present, or configure the virtual device. The designer's plan is absolute.

3. UNIX Process Metadata

(Kell, 2015) observes that, through the practical necessity of supporting the crucial authorial activity of debugging, UNIX processes have *de facto* been supplied with sufficient metadata to support substantial introspection into their allocation patterns. This is contrast to the typical computer science design recommendations of providing *services* to support such faculties of reflection. The traditional VM approach to reflection has "the reflecting client consume the services of an in-VM reflection API and/or debug server". (Kell, 2015) notes that such an approach would be substantially limited in function (not supporting the post-mortem case of debugging against a "dead" core dump) and portability (not supporting the use of one vendor's debugger to debug code from another's compiler). By casting its task in terms of externalising access to the state of the system, by mapping its addresses in terms of metadata, the process model promotes the capability of parts of the system to act as avatars for others, even to the extent of bridging the divide between the living and the dead (being able to treat in-memory, running processes on common terms to core dumps in files). Similarly to the ability of state transfer-based MIDI devices to enjoy huge longevity and portability, (Kell, 2015) notes that for a system cast in terms of an API, "it becomes hard to implement reflection features not anticipated in the design of the reflection API or debug server command language. By contrast, metadata is open-ended and naturally decoupling".

4. The Nexus

The GPII Nexus (The GPII Team, 2016) is an attempt to consciously promote and support the values underlying the avatar model, in a concrete embodiment as well as a pattern of morality. The Nexus is currently implemented as a JavaScript application written in node.js, which exports a fully addressed tree of implementation cells, each named a component, over widely supported public web protocols such as HTTP and WebSockets, with payloads encoded as JSON.

Clients of the Nexus could be easily supported in any language or platform, including within web browsers, low-powered devices, and mobile or desktop platforms. When addressing a particular component in the tree, a segment of the Nexus API corresponds to the well-known "CRUD over REST" protocol whereby resources are managed by HTTP verbs at a particular URL. However, when taken as a whole, the Nexus API supports the transmission of wholesale avatars - querying the contents of a particular section of its component tree over HTTP allows it to be completely replicated at another site, together with any dynamic behaviour (including its capacity to create or destroy further components).

A crucial element of any successful avatar system is transaction demarcation. This typically arises informally in avatar systems to date (necessarily, since almost all of their architecture has arisen informally). However, it is an important goal of the Nexus to formalise such demarcation models as well as providing infrastructure for supporting reference implementations. Transactions are even more essential within avatar systems than they are in traditional database applications storing "dead data" over CRUD - since the moment an avatar is transferred, its state may immediately start to evolve. It's essential that there can be a model of atomicity for these updates so that partially transferred avatars do not begin to corrupt their state. This atomicity is also crucial to support "avatars over time" that we sketched in section 1 — if we begin to speculatively execute some possible future configuration of the system, it's essential that it can be completely backed out without effect if it is not desired in the present.

To support this, the Nexus will support a model inspired by current generations of distributed version management tools such as git, which assign globally stable hashes to the configuration of parts of its tree, to which configuration the tree could always be restored assuming that the storage backing the hash has not been discarded. Every client communication is enlisted in such transactions, in which they are operating on an avatar which isolated from that visible by other clients, until they explicitly commit their transaction. At this point, the updates to the avatar will be resolved - if they cause an error either with respect to the internal avatar contents, or with respect to the rest of the tree, the entire transaction will be backed out and the communication ended without any externally visible side-effects. This process allows safe experimentation with arbitrarily complex updates to the application state, many of which could be speculatively executed against local or remote avatars, supporting the use cases of L6 programming. The

choice of where to execute the update can be driven by the economics of the situation — the relative compute power of the client and its peer, network costs, latency requirements — rather than being a choice forced by the software architecture as is almost always currently the case.

5. Conclusion

We've described the important implementational idiom of the **avatar** and shown its relationship to the externalised state transfer idiom, which is one that arises naturally in communities which need to produce artefacts that are shared as part of an open ecology of function. The effect of the externalised state transfer idiom is to promote and facilitate the implementation of avatars, resulting in harmonious, democratic and open experiences of authorship by integrated communities, rather than proceeding in a hierarchy of command from a technical elite. The pleasant results of this idiom include live programming systems able to assist the user to speculate about the future effects of current authorial decisions, systems which expose a rich variety of interaction idioms and technologies suited to a diverse community of authors and users, and systems which enjoy huge longevity and support through ditching reliance on brittle idioms based on contracts, function calls, and encapsulation boundaries.

6. References

Basman, A., Church, L., Klokmose, C., & Clark, C. (2016). Software and How it Lives On - Embedding Live Programs in the World Around Them. In *Proceedings of the Psychology of Programming Interest Group.*

Church, L., Söderberg, E., Bracha, G., & Tanimoto, S. (2016). Liveness becomes Entelechy - a scheme for L6. In *The second international conference on live coding.*

Fielding, R. T., & Taylor, R. N. (2000). Principled design of the modern web architecture. In *Proceedings of the 22nd international conference on software engineering* (pp. 407–416). New York, NY, USA: ACM.

Kell, S. (2015). Towards a Dynamic Object Model within UNIX Processes. In *Proceedings of the 2015 OOPSLA Companion (Onward)* (pp. 224–239).

Parnas, D. L. (1972). On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, *15*(12), 1053-58.

The GPII Team. (2016). *The GPII Nexus.* Retrieved from `https://wiki.gpii.net/w/The_Nexus`

Tiselice, D. (2015). *Web sucks and here's how we can make it awesome.* Retrieved from `https://www.presslabs.com/blog/web-sucks-how-to-make-it-awesome/`