An Experiment in Live Collaborative Programming on the Croquet Shared Experience Platform

Yoshiki Ohshima
Croquet Corporation
Los Angeles, CA, USA
yoshiki@croquet.io

Aran Lunzer
Croquet Corporation
Los Angeles, CA, USA
aran@croquet.io

Jenn Evans
Croquet Corporation
Vancouver, BC, Canada
jenn@croquet.io

Vanessa Freudenberg
Croquet Corporation
Los Angeles, CA, USA
vanessa@croquet.io

Brian Upton
Croquet Corporation
Los Angeles, CA, USA
brian@croquet.io

David A. Smith
Croquet Corporation
Cary, NC, USA
david@croquet.io

ABSTRACT

This paper describes our experiences in building a live collaborative programming environment on top of the JavaScript version of the Croquet shared experience platform.

Croquet provides a clean substrate for building real-time collaborative applications. We created an application framework that supports live programming, and used that framework to build the Greenlight collaborative application, then in turn, modified it to do live programming experiments. The environment allows multiple users to modify the running application from within, with changes taking effect immediately.

The experiment was inspired by earlier work including Douglas Engelbart's oN-Line System (NLS) and the Kansas system in Self. Analogically, the system is like the Smalltalk environment made collaborative.

In this paper we explain the Croquet architecture, its library and framework, and the Greenlight application used to make the live programming environment.

The standard version of Greenlight is available at https://croquet.io/greenlight, and the modified demo system is available at https://croquet.io/scripting.

CCS CONCEPTS

• **Software and its engineering** → **Programming teams**; • **Human-centered computing** → **Synchronous editors**.

KEYWORDS

Shared Experiences, Collaboration Application Framework, Live Programming

ACM Reference Format:
Yoshiki Ohshima, Aran Lunzer, Jenn Evans, Vanessa Freudenberg, Brian Upton, and David A. Smith. 2022. An Experiment in Live Collaborative Programming on the Croquet Shared Experience Platform. In *Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming (<Programming> '22 Companion), March 21–25, 2022, Porto, Portugal*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3532512.3535224

<Programming> '22 Companion, March 21–25, 2022, Porto, Portugal

ACM ISBN 978-1-4503-9656-1/22/03…$15.00
https://doi.org/10.1145/3532512.3535224

1 INTRODUCTION

Imagine that a group of colleagues are building an application. The tasks include developing not just the visual design but also the software components themselves. Ideally developers and designers can work together in the same shared development session with a tight feedback loop. Of course each user would have their own input devices and display. And of course a user should be able to join the session from anywhere on Earth, at any time. The environment would be similar to popular collaboration applications such as Figma and Google Docs; the difference is that the material being developed is code.

It is vital for such an environment to be "live": one should not have to reload the whole application to see the effects of a small code change, nor should one user's changes interrupt the workflow of their collaborating developers and designers. In this sense, the live programming facility is even more important for a multi-user collaborative environment than a single-user one.

We decided to build an experimental collaborative live programming environment on top of the Croquet real-time shared experience platform. Croquet is based on the replicated computation model [24] for writing real-time shared web applications. The system design draws upon earlier systems with the same name, but was fully re-implemented in JavaScript. (For an overview of the current Croquet platform, see [4, 12]).

It is important for us that such a programming environment be usable for collaborative editing and improvement of the environment itself, as well as new applications. This idea was inspired by highly productive self-sustaining systems like Smalltalk and Lisp, and also by the bootstrapping idea from Engelbart's oN-Line System (NLS).

The rest of the paper is organized as follows. Section 2 describes the foundational Croquet architecture, with Section 3 showing a simple example application written in Croquet. Section 4 introduces an application framework called Croquet Virtual DOM Framework that supports live programming, Section 5 describes the Greenlight application that was built on top of that framework, and Section 6

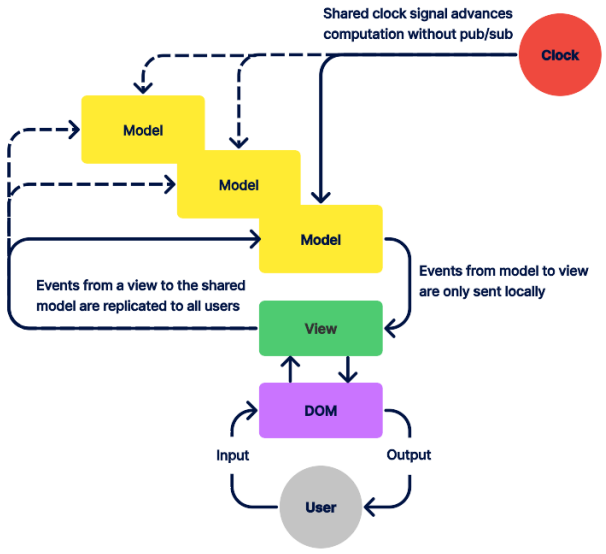

Figure 1: A diagram of the Croquet architecture.

displays our collaborative live programming experiment within Greenlight. Section 7 discusses related work.

2 CROQUET

Croquet is a platform for creating rich multi-user applications. Instead of having to write client/server and networking code, developers write code to be executed in a shared virtual machine (VM) running on each peer[1] in a session, which is automatically synchronized by Croquet. This gives the appearance of each peer having direct access to a *single shared computer*, which in our experience is a much simpler mental model for writing multi-user applications than designing a networked client/server application.

Croquet relies on absolutely bit-identical deterministic behavior of code in that VM, so that the illusion of a single shared computer is preserved. By controlling the progress of time and ensuring that the sequence of external events is identical, all peers stay in sync.

Our current system is implemented on top of JavaScript and can run in a web browser or on Node.js. The ECMAScript standard ensures a high degree of conformity in the execution semantics across different platforms. Where the standard allows differences, we provide solutions that ensure the same outcome on every platform (in particular, transcendental functions behaviors).

All application code is only executed on the peer machines, and can be deployed in a static webpage. Croquet applications consist of two parts: a shared part running in the synchronized VM, and a non-shared part for each user handling input and output. We call these parts *models* and *views*, respectively (see Figure 1).

Application computation and resulting changes of state happen only on the peers. When a peer needs to inject a state change into the session, typically due to an input action by its user, it informs its peers by transmitting an event to the session's synchronization server. This server puts a timestamp on the event and "reflects" it to all the peers, without needing to examine the payload at all; this is why we internally refer to the servers as *reflectors*. All transmission of events is encrypted end to end.

Application code uses the Croquet publish/subscribe API to send events. Events published by a view affecting a model are sent to the reflector and reflected back to all peers, so they stay in sync. This is typical for user input. All other events are delivered only locally. While Croquet enforces a strong *separation between models and views*, it allows views to read data directly from models. This allows very efficient rendering of complex shared data.

Models must be implemented in an object-oriented style. The way a new peer joins an existing session is by loading a snapshot of the models' state that Croquet previously captured automatically; for this to work, no model can contain any value that is not serializable, such as a JavaScript closure. The model code is also forbidden from using any platform features that would make it non-deterministic, such as local system properties or wall-clock time.

The above restrictions do not apply to view code. The only requirement is that a view never modify model properties directly, because that would make the local model different from the other peers, breaking synchronization. Views are free to adapt to local system features, including screen size and input-device availability.

This design allows a user to join and leave a Croquet application at any time. When a peer joins a session, the last snapshot as well as all messages after the snapshot are sent to the peer. When the peer finishes replaying the messages starting from the snapshot, it is guaranteed to have identical model state to all other peers.

The reflectors are extremely efficient, since they do not have to perform any computation on application data. We operate many reflectors around the globe. The system chooses a close-by reflector for each session. This allows a typical session to have only tens of milliseconds of latency, which is determined by the round-trip time to a nearby data center. With more Edge reflector deployments, there will be even lower latency without additional effort on the part of the application developer.

We have been running the Croquet system for a few years now, with many different applications (written by us and other developers ranging from CS students to commercial developers), and we have found that they work very reliably. Croquet reduces complexity not only by eliminating networking and server code from applications, but also by not having to resolve conflicts or perform speculative execution or rollbacks at the framework level.

Note, however, that Croquet apps still require careful design of the program and interface to provide a smooth shared experience. For example, consider two users working in a collaborative text editor, and simultaneously hitting the delete key with cursors in two different points in the text. The two delete requests are sent to the server, which sequences them and sends them to peers in deterministic order. If each deletion were naively specified in terms of text index, the state after the first executed deletion may cause the second to occur in an unintended place. A collaborative text editor written on top of Croquet must avoid such errors, for example by appropriate transformation of the operations, though

[1] We use the term "peer" to refer to the client-side Croquet software system, and "user" for a human participant who is using a Croquet application.

```
1  class MyModel extends Croquet.Model {
2    init() {
3      this.count = 0;
4      this.subscribe("counter", "reset",
5                     this.resetCounter);
6      this.future(1000).tick();
7    }
8    resetCounter() {
9      this.count = 0;
10     this.publish("counter", "changed");
11   }
12   tick() {
13     this.count++;
14     this.publish("counter", "changed");
15     this.future(1000).tick();
16   }
17 }
```

```
18 class MyView extends Croquet.View {
19   constructor(model) {
20     super(model);
21     this.model = model;
22     countDisplay.onclick = event=>this.counterReset();
23     this.subscribe("counter", "changed",
24                    this.counterChanged);
25     this.counterChanged();
26   }
27   counterReset() {
28     this.publish("counter", "reset");
29   }
30   counterChanged() {
31     countDisplay.textContent = this.model.count;
32   }
33 }
```

Figure 2: The model and view view code of a simple automatic counter example.

the deterministic ordering of events does simplify the necessary algorithm (see an actual implementation in [6]).

Similarly, while Croquet itself does not use speculative execution, an app designer may want to incorporate it so that a user gets immediate feedback instead of having to wait for an event's round trip to the reflector. For example, the position of an object being dragged by one user could be updated in that user's view immediately, at the same time as sending events to all peers via the server. On receipt of the event, each peer's model—including that of the initiating user—will calculate the object's definitive position, taking into account any constraints imposed by the model such as collisions with other objects that are being dragged. The initiating user's view would then, if necessary, correct its prediction with the actual result.

3 CROQUET APPLICATION DEVELOPMENT IN A NUTSHELL

The Croquet JavaScript Library offers essential features for writing Croquet-powered applications. The main task when developing an application is to define the shared behavior as the "model", and how it interacts with the outside world through each peer's "view".

In practice, a developer creates model objects as a subclass of the base `Croquet.Model` class, and customizes the subclass to store shared data and implement shared behavior. A customized `Croquet.View` subclass provides input/output for a user by interacting with the browser's Document Object Model (DOM) elements [8].

Croquet offers a simple event publish/subscribe mechanism for describing model/view interactions. Note that unlike traditional pub/sub, peers do not directly interact with each other, but only with the shared model.

The left hand side of Figure 2 shows the model code of a Croquet application example (a full description is available as the Hello World tutorial within the Croquet documentation: https://croquet.io/docs/croquet/). The `count` property is the only application state (line 3). Line 4 with `subscribe` declares that when an event called `reset` in the `counter` scope arrives, it invokes `resetCounter()`, which resets the `counter` property to zero and publishes an event named `changed` (lines 8-10). The `tick` method is called from `init()` with the *future* mechanism, which specifies to invoke that method 1000 milliseconds in the future. `Tick` calls itself again with `future`, so that `tick()` is executed at 1000 millisecond intervals (lines 12-16).

A simple way to write view code is to manipulate the DOM elements defined in HTML. The right hand side of Figure 2 shows the corresponding view code written in that way.

A DOM element accessible from the code as `countDisplay` provides the user interface for this counter. The element's content is updated in response to a change in the count by way of the view's subscription to the `changed` event in the `counter` scope (line 23), which causes each such event to invoke the `counterChanged` method (line 30). The element also supports a trivial form of input, in that its `onclick` DOM event handler (line 22) invokes the view's `counterReset` method, which publishes a `reset` event (line 28). Because the Croquet VM can see that there is a subscription to that event in the model, it automatically routes the event via the server so that it will be received by all the peers. Every peer's model will receive the reflected event at effectively the same time, and call its `resetCounter` method (line 8) immediately, so the application state remains synchronized. On the other hand, the Croquet VM can see that the `changed` event only travels from the model to the view, which is a local effect that does not have to go through the network; for this event, the Croquet VM triggers the subscription handler on the view and all peers' DOM elements are updated accordingly.

Just like Croquet takes care of the event routing, it takes care of snapshotting the model state. When a peer joins a session that does not yet have a snapshot, it runs the model's `init` method to initialize the model state. Snapshots of the state are taken periodically as the session runs, so that if a new peer joins a mature session it can use the snapshot to jump forward to whatever state the model has. Then, these later-joining peers do not run `init` at all.

Note that the `count` value is *not* needed as an argument to the `changed` event. Instead, the `counterChanged` method can directly access the shared model state for rendering. In more complex applications this is a significant advantage over other technologies, where the shared state resides on some remote server and can only be accessed via the network.

Direct model access works because of the bit-identical deterministic execution behavior of Croquet models. To ensure this, Croquet uses customized versions of certain library functions that model

code might use. For example, if a model invokes `Math.random()` it executes a custom random-number generator that yields the exact same sequence of numbers on all peers. This means that even a complex real-time simulation that relies on getting thousands of random numbers each second will stay in sync across all peers without them having to communicate at all.

4 THE CROQUET VIRTUAL DOM FRAMEWORK

While the Croquet Library described in the previous section is effective for writing simple applications, complex applications call for more powerful frameworks. The Croquet Virtual DOM Framework [7] is designed to support easier manipulation of DOM elements to help writing DOM-based 2D shared applications. Another goal for the framework was strong support for dynamic update of application code.

We took inspiration from many existing live programming systems and application frameworks. The first consideration in our design was how to show visual elements on screen: specifically, whether to use the canvas element exclusively, or to embrace a richer set of DOM features.

4.1 Canvas vs. DOM

There are two major directions that one can take for displaying visual elements in a modern browser window.

The native "widget set" of the modern web browser is Document Object Model (DOM), which provides various types of visual elements, and the ability to customize their appearance and behavior through with CSS (Cascading Style Sheets). One of the element types is "canvas": a surface onto which images can be drawn using two-dimensional (or three-dimensional) graphical operations.

One approach to creating a 2D application framework is to instantiate a single canvas element that occupies the entire window area, and to show the application's graphical elements by drawing them painstakingly with canvas drawing commands. This approach leads to a clean and simple abstraction, and the ability to take full control of the display model. However, manipulation of text or of graphical elements is laborious to implement, and scrolling or zooming of the viewport requires re-execution of all drawing commands for the newly visible screen area. In addition, a canvas-based application cannot embed independent JavaScript context in the special DOM element type called `iframe`.

A second approach is to embrace DOM for showing the visual elements. It leads to a system that is closely integrated with the browser, and can use many powerful features such as highly optimized CSS-tuned layout of text and widgets, and viewport transformations. The embedding of web pages can be achieved straightforwardly by using the `iframe` DOM element. One downside is that it requires intimate coupling with CSS, and having to be aware of the numerous incompatibilities between different browser implementations.

We considered both options. Because the main requirement for the primary application of this framework (described in the next section) was to be able to embed and manage many iframes, we decided to create an application framework that embraces DOM.

4.2 Virtualizing the DOM Specification

Having decided on using DOM, we had to find a way to work within the Croquet rule that a model cannot include any objects that are not serializable. That precludes all DOM elements. Inspired by Dan Amelang's work on implementing the DOM spec in JavaScript for the Mico framework [1], we decided to "virtualize" the DOM elements into pure JavaScript descriptions that can be stored in a model. We define a plain JavaScript object that has properties such as `parentNode` and `childNodes`, `style` (which in turn has methods such as `setProperty()` and `getPropertyValue()`), `classList`, and methods such as `addEventListener()`. When a property of a virtualized DOM element changes in the model, the framework notifies the view to update the actual DOM elements on screen. The application framework calculates the difference between the last state and the current state, and applies it to the view's actual DOM elements.

In short, an application in the Croquet Virtual DOM Framework consists of a set of virtual DOM elements in the model as well as corresponding virtual DOM views in the view, with the view managing the actual DOM elements on-screen.

In this scheme, the application programmer can write a program against the familiar set of functions that the DOM interface provides. Any event that causes change on the virtual DOM properties is sent to all peers, and since all peers execute the same code, their virtual DOMs remain identical. The visual appearance on each user's screen is updated accordingly. It is fine for the users to have different browser window sizes or different input devices; CSS and DOM absorb the difference, while keeping the virtualized model identical.

4.3 Full DOM or Subset of DOM

The HTML specification defines over 130 element types. Some of those are essential for our purposes, while others can be emulated by a `div` element with some CSS attached. We therefore chose to limit the types of DOM elements supported: as of writing, we have `div`, `iframe`, `canvas`, `img`, `video`, and `textarea` elements. Although the `div` virtual element allows us to emulate many element types, the others in the list each have special features that such emulation would not support. For example, having a separate `canvas` virtual element is necessary as its `width` and `height` attributes have special meaning. `Iframe` has many attributes for controlling an embedded page, including `src`. Our `textarea` is quite different from the true DOM version, as ours supports collaborative editing by multiple users. The `img` element could have been partly emulated by a `div` with the `background` CSS property, but for some applications we wanted to take advantage of browser-native `img` features such as image drag and drop and the "save as" command.

In addition to supporting their subset of the DOM API, the virtual DOM elements are Croquet `Model` and `View` objects so that they have `publish` and `subscribe` methods.

4.4 Behavior Representation: Instance-based Expanders

Now we know how to handle DOM elements in a Croquet model: we represent an element as a serializable plain JavaScript object in

the model, create a corresponding actual DOM element in the view, and have a Croquet view object manage that.

What we are interested in is a programming system where users can collaboratively update application code. The model-view separation requirement means that changeable data in the model—which in this case includes code—must be serializable, and that when a change occurs, that change needs to be applied on all peers identically and cleanly. For example, there must be a safe way to remove a piece of code without breaking the running application.

Based on the rich literature around PIE extensible objects [14], Mixins [2], Traits [22], Expanders [26], as well as the way an application like Etoys allows attachable/detachable behavior in object-oriented programming [13], we have devised a scheme that we call instance-based expanders.

An instance-based expander is a set of methods wrapped in the JavaScript class syntax. Those methods are required to have no external references or free variables so that they can be recreated from stringified data. The system evaluates the string representation of an expander to create a JavaScript class (function) and stores it in a dictionary in the object being expanded, which is either the model- or view-side manifestation of a virtual DOM element.

The expanders are called "instance-based" because they are installed in a specific virtual-DOM object instance. In our experience, a common path for development was to customize one instance at a time in an exploratory manner. Some optimizations are implemented so that identical code is not stored multiple times in different instances, and further optimizations can be applied when the developer decides to deploy a version of the application that does not have to support live updating of code.

As often discussed in the literature mentioned above on object customization, the object identity of an expanded object (i.e., what the "this" pseudo-variable refers to) is an important design trade-off. In the case of instance-based expanders, "this" refers to the base object. Upon invoking an expander method, a JavaScript Proxy is created on the base object, and property read and write access are passed through. This means that the property names used in separately developed expanders that are applied simultaneously to a single base object might collide, but from our experience the benefit of being able to communicate between the multiple expanders outweighs the burden of taking care of potential clashes.

Because its primary representation is a string, the application can update it, recreate the class object and replace it cleanly.

Figure 3 shows the "counter" example written in the Virtual DOM Framework. The base element is created as a virtual `textarea` element, and has been inserted into the top-level virtual DOM element by code elsewhere. Note that in principle we would like to use a keyword such as `expander` rather than `class` (line 1), but sticking to the standard syntax for now lets us take advantage of existing tools such as `eslint`.

The code is similar to the Croquet Library version presented earlier, with a few notable differences. One is that we only need to write code for the model but not for the view. The `init()` initializer calls `addEventListener()` on the virtual DOM element (line 3), which sets up some event handlers and Croquet publish/subscribe calls so that a DOM event on the corresponding actual DOM element will be sent to this virtual DOM element. Because the virtual DOM element for `textarea` has a property called `value`, just as a real

```
1  class Counter {
2    init() {
3      this.addEventListener("click", "reset");
4      this.count = 0;
5      this.future(1000).call("Counter", "next");
6    }
7    next() {
8      this.count++;
9      this.value = "" + c;
10     this.future(1000).call("Counter", "next");
11   }
12   reset() {
13     this.count = 0;
14     this.value = "" + c;
15   }
16 }
```

Figure 3: The counter example in the Virtual DOM Framework

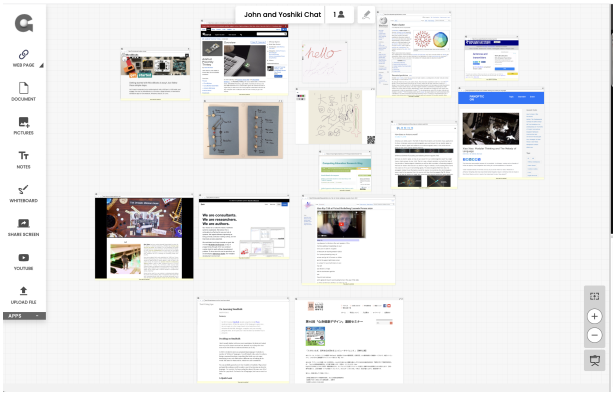

Figure 4: A screenshot of a typical Greenlight session, where the user has zoomed out to see various objects in the room. If a Croquet app is loaded as one of the objects, all users in the room become participants in that embedded app.

`textarea` element would, the code can assign a new value into it. The framework will notice the change in the value property and apply the difference to the actual DOM element.

With the design and implementation described in this section, we now have a powerful application framework that also is a good basis for providing live programming capabilities. In the next two sections we will describe an application that is built on top of the framework, and how we can unleash the power of live programming using it.

5 GREENLIGHT

Greenlight [5] is an application built on top of the Croquet Virtual DOM Framework. It is inspired by Self's Kansas system [25]. As in Kansas, it is a large and flat rectangular area where you can open collaborative text editors, images, and `iframe` elements that in turn embed other Croquet applications. The screenshot in Figure 4 shows a typical session in Greenlight. Each user has their own mouse cursor, and their positions change in real time as other users interact with the application.

One can think of the large area as like a tabletop where various papers, images and dynamic applications are laid out (although

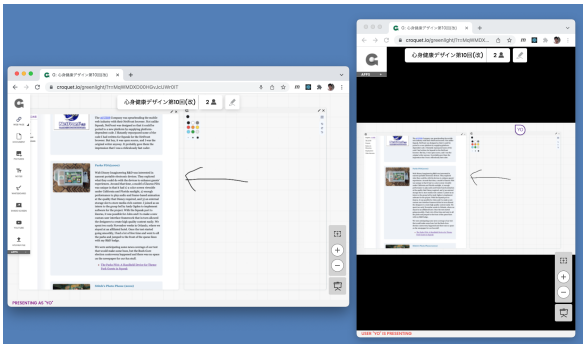

Figure 5: A screenshot showing two user tabs open on a session with presentation mode enabled. The tab on the left is acting as presenter, and the one on the right is following. Because the presenter's tab is wider, the presentation in the follower's tab is letterboxed so that exactly the same region of the room appears.

since we refer to a session as a "room", perhaps one should rather think of them as being laid out on the floor). Each user by default has an independent viewport in the room, and can zoom and pan to see different parts of it. By contrast, whenever a user moves or resizes an embedded object, all users see these changes as they happen. Optionally, a session may have a video-chat app, so that users can talk to each other while working together.

As well as collaborating at the level of the presence and layout of objects in the room, users can collaborate within the applications that have been loaded into the room. For example, a group can open an iframe containing a Google Doc page, and create and edit a document in that iframe without leaving the Greenlight browser tab.

There are many subtle issues around designing user interface features that work in the multi-user setting. For example, Greenlight has a "presentation mode" in which one user takes control of the viewport seen by all users, to make it easy to direct everyone's attention to the same embedded objects; a design issue was how to control exactly what a "follower" user sees in this mode when everyone's screens (i.e., their browser tabs) can have different sizes and aspect ratios. As shown in Figure 5, our solution was to use letterboxing on follower tabs to ensure that everyone sees exactly the same viewport.

We have released Greenlight as a locked-down application without telling people about a secret feature that lurks within. This feature—support for collaborative live programming—is discussed in the next section.

6 LIVE PROGRAMMING IN GREENLIGHT

As described in Section 4, the code of an application written on top of the Virtual DOM Framework is stored as text data. This means that if we provide a collaborative editor for changing that text, and a way for changed text to be adopted as the application's new code, we have ourselves a live collaborative programming environment.

A trivial demonstration of such live code editing is a workspace that evaluates a snippet of JavaScript code (Figure 6) and shows the

Figure 6: The expression 3 + 4 has been entered into a collaborative text editor and evaluated.

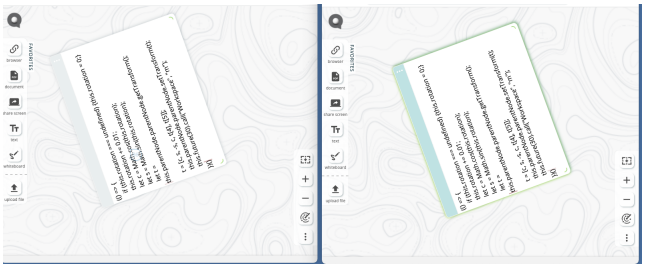

Figure 7: The workspace itself can be made to rotate by changing the behavior of the "this" object.

result. All users can see and edit the code snippet, and whenever anyone hits Cmd-S, the current content is evaluated and the result appears. Note that the appearance of the result in every user's workspace is not another shared edit to the workspace content, but happens because each user is locally evaluating the snippet and displaying the result.

The "this" variable in this workspace refers to the `textarea` object that is showing the code. This makes it easy to perform some tricks on the environment itself. For example, one can write a method to access and modify the CSS transformation of the `textarea`'s grandparent node (a `div` representing the window frame) so that the window is given, say, a new rotation angle. Repeatedly evaluating this code with the help of a Croquet future() call will mean that on every user's screen the window will start to rotate steadily (Figure 7).

Note that, thanks to the power of the browser and CSS, the collaborative text editor itself is fully functional even as it rotates; pointer and keyboard events are automatically transformed, allowing users to continue to edit the code and save new versions to change the window's behavior on the fly.

Beyond such simple visual tricks, there are of course more interesting capabilities to live programming. A real application consists of multiple objects, and the job of the application code is to make

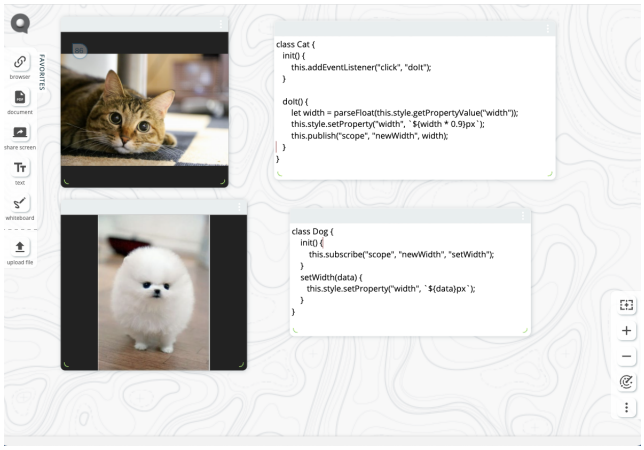

Figure 8: Two objects that each have their own expanders, and communicate with each other via the publish/subscribe mechanism.

those objects work together. One building block is adding code that causes an event on one object to affect other objects. The following is a simple demonstration of loading two images and adding such event handling on the fly.

In Figure 8, two virtual DOM elements are loaded, and two separate script editors opened to edit their instance-based expanders. As explained in Section 4, the elements are equipped to publish and subscribe to Croquet events. In this example, the cat object listens for a DOM `click` event, and when it occurs the cat finds its CSS `width` property and replaces it with the existing value multiplied by 0.9. The cat then publishes a Croquet `newWidth` event with the new `width` as payload.

Independent from the cat's behavior, the dog listens for the `newWidth` Croquet event; when an event arrives, the dog sets its own `width` based on the event's `data` payload.

With those two scripts, whenever any user clicks on the cat, both the cat's and dog's widths change for all users. Anybody can edit code while others are experimenting, and the new definition takes effect as soon as Cmd-S is hit (and the content compiles).

The true potential power of live programming becomes apparent when one considers that the code of the Greenlight application itself is amenable to editing in this way. Greenlight uses thousands of virtual DOM elements, all coded using instance-based expanders and all communicating using Croquet's publish and subscribe. It would be straightforward to change or extend Greenlight on the fly, just as in the demo above, for example adding a script that invokes some Greenlight feature automatically when some condition is fulfilled.

Liveness is considered important in the live programming community, but we think that it is truly essential in the collaborative setting. One can imagine a collaborative session where one user is setting up behavior for some object, while another user tweaks the visual styles of other related objects. Each user's changes are immediately shared with the other, so they both see their combined progress towards the shared goal. Of course there is the potential for clashes if they both try to edit the same object at the same time, but in a cooperative setting they can use side channels such as the video and voice chat to help steer clear of such conflict. The users could also choose to work for extended periods on their own, preparing large sections of code in separate rooms (or copies of the main room) then bringing them into the main room later, either copying and pasting or opening the work in an `iframe`. In other words, the live collaboration on shared objects is not an imposed requirement but an additional available capability.

While the development of Greenlight itself did not fully take advantage of collaborative programming, we believe that it stands as a validation of the following ideas: (1) an application as complex as Greenlight can be written using instance-based expanders to define the behaviors of thousands of cooperating elements; (2) Croquet provides a solid foundation for keeping such a collaborative application in sync; and (3) live editing of such a large-scale application is possible and practical. We feel that with further attention to the editing facilities, collaborative live programming will be a realistic technique for future development of similarly rich applications.

7 RELATED WORK

The replicated computation model can be traced back to TBAG by Elliott et. al [9]. TBAG transmits events in a peer-to-peer manner. The restricted programming model, which was constraint-based, ensured that the resulting application state was replicated. Croquet's computation is more general and allows developers to write interactive programs in various styles, as long as the resulting data is stored in the model objects.

Some multi-user games, such as Age of Empires, use a similar model. The model works well enough for the limited forms of synchronization needed within those games' domains, but Croquet offers a platform for writing a broader range of multi-user applications.

There are many "local first" [18] frameworks based on CRDT [23], including Yjs [10] and automerge [11]. Those frameworks aim to solve similar but still substantially different problems. The primary concern of CRDT and other solutions is what to do when changes cannot be propagated "right away", including the possibility that the network is temporarily disconnected. The merging algorithm can be intelligent so that it preserves the "intent" of changes. A user's intent can be guessed reasonably in a limited domain such as text editing or other well known problems, but merging changes to a generic data structure when radically conflicting changes are made is often hard.

On the other hand, Croquet simply assumes real-time network connectivity for its normal operation, and tries to provide good real-time shared experiences. The server guarantees consistent message order, meaning that eventual consistency between peers is guaranteed, and not something that the developer even has to consider. Croquet also fully takes care of reconnecting a user consistently into the session when recovering from a network disruption. That said, as written in Section 2, a Croquet app still requires careful treatment when users issue events that conflict with each other.

Croquet's session-join mechanisms (based on snapshots and event catch-up) relieve the developer of many challenges that generally arise in shared-application development. This goes beyond

the reconciling of data changes to the storage and recovery of application state; these can be daunting tasks for a developer who is not familiar with the myriad ways in which network disruption can occur. All the features needed are provided as a service, so a developer can create a shared application without setting up a server or network storage.

The concept of self-sustaining live programming environment on the web was influenced by a series of implementation of Lively Web [17], especially the "Lively4" implementation that embraced DOM elements [19].

As for related work for Greenlight, there are many collaboration applications available [15, 16]. Greenlight differs from many of them as it is a meta application to manage other collaborative applications. The entire application is written in about 10,000 lines of code, which means that a single person can fully understand and modify it.

There are also many attempts in the collaborative live programming field [3, 21]. However, many of those focus just on writing code together, without guaranteeing that everyone arrives at identical execution results [20]. The further step of supporting collaborative development of collaborative applications has not been extensively researched.

The collaborative live programming feature in Greenlight is experimental, and is not ready for public consumption. However, the results achieved so far suggest that it is a powerful direction for further pursuit.

8 CONCLUSION

This paper describes our JavaScript-based implementation of Croquet, the base Croquet library, the Virtual DOM application framework on top of the library, the Greenlight application that was built with that framework, and an experimental live programming feature built within Greenlight. We believe that each of these layers is novel, and that what we have shown by way of a possible user experience in collaborative live programming points to a worthwhile direction for future programming environments.

The work was done in a 2D application, but there is nothing that limits this concept to two dimensions. In fact, the liveness and immediate feedback are even more essential in a 3D and immersive environment. We hope that this work inspires future collaborative creation environments of all forms.

ACKNOWLEDGMENTS

The authors wish to acknowledge David Reed, Alan Kay and the late Andreas Raab, who worked with some of us in bringing the original Smalltalk version of Croquet to fruition. We also thank our Croquet Corporation colleagues.

REFERENCES

[1] Dan Amelang. 2008. *Mico.* Viewpoints Research Institute. https://github.com/damelang/mico.

[2] Gilad Bracha and William Cook. 1990. Mixin-Based Inheritance. In *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications* (Ottawa, Canada) *(OOPSLA/ECOOP '90)*. Association for Computing Machinery, New York, NY, USA, 303–311.

[3] CodeSandBox. 2021. *CodeSandBox Live.* CodeSandBox. https://codesandbox.io/docs/live.

[4] Croquet Corporation. 2018. *Croquet.* Croquet Corporation. https://croquet.io/docs.

[5] Croquet Corporation. 2019. *Greenlight.* Croquet Corporation. https://croquet.io/greenlight.

[6] Croquet Corporation. 2020. *The Croquet Virtual DOM Framework.* Croquet Corporation. https://github.com/croquet/virtual-dom.

[7] Croquet Corporation. 2020. *The Croquet Virtual DOM Framework.* Croquet Corporation. https://croquet.io/docs/virtual-dom.

[8] Terence Eden, Xiaoqian Wu, Sangwhan Moon, Shwetank Dixit, Scott O'Hara, Patricia Aas, and Bruce Lawson. 2018. *HTML 5.3.* W3C Working Draft. W3C. https://www.w3.org/TR/2018/WD-html53-20180809/.

[9] Conal Elliott, Greg Schechter, Ricky Yeung, and Salim S. Abi-Ezzi. 1994. TBAG: a high level framework for interactive, animated 3D graphics applications. In *Proceedings of the 21th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1994, Orlando, FL, USA, July 24-29, 1994*, Dino Schweitzer, Andrew S. Glassner, and Mike Keeler (Eds.). ACM, New York, NY, USA, 421–434. https://doi.org/10.1145/192161.192276

[10] Kevin Jahns et al. 2014. *Y.js.* Kevin Jahns et al. https://yjs.dev.

[11] Martin Kleppmann et al. 2017. *Automerge.* Martin Kleppmann et al. https://github.com/automerge/automerge.

[12] Vanessa Freudenberg. 2020. Croquet: A Unique Collaboration Architecture. Video is available at: https://www.youtube.com/watch?v=ujOVHVAjXj4.

[13] Vanessa Freudenberg, Yoshiki Ohshima, and Scott Wallace. 2009. Etoys for One Laptop Per Child. In *2009 Seventh International Conference on Creating, Connecting and Collaborating through Computing*. IEEE Computer Society, Los Alamitos, CA, USA, 57–64.

[14] Ira P. Goldstein and Daniel G. Bobrow. 1980. Extending Object Oriented Programming in Smalltalk. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming* (Stanford University, California, USA) *(LFP '80)*. Association for Computing Machinery, New York, NY, USA, 75–81. https://doi.org/10.1145/800087.802792

[15] Google. 2017. *Google Jamboard.* Google. https://jamboard.google.com.

[16] Miro Inc. 2012. *Miro.* Miro Inc. https://miro.com.

[17] Daniel H. H. Ingalls, Krzysztof Palacz, Stephen A. Uhler, Antero Taivalsaari, and Tommi Mikkonen. 2008. The Lively Kernel A Self-supporting System on a Web Page. In *Self-Sustaining Systems Workshop (S3)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 31–50.

[18] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-First Software: You Own Your Data, in Spite of the Cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Athens, Greece) *(Onward! 2019)*. Association for Computing Machinery, New York, NY, USA, 154–178. https://doi.org/10.1145/3359591.3359737

[19] Jens Lincke, Stefan Ramson, Patrick Rein, Robert Hirschfeld, Marcel Taeumel, and Tim Felgentreff. 2017. Designing a Live Development Experience for Web Components. In *Programming Experience 2017.2 (PX/17.2) Workshop*. ACM, New York, NY, USA.

[20] Microsoft. 2018. *Visual Studio Live Share.* Microsoft. https://visualstudio.microsoft.com/services/live-share/.

[21] Replit. 2016. *Replit.* Replit. https://replit.com.

[22] Nathanael Schärli. 2005. *Traits — Composing Classes from Behavioral Building Blocks.* Ph. D. Dissertation. University of Berne.

[23] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems* (Grenoble, France) *(SSS'11)*. Springer-Verlag, Berlin, Heidelberg, 386––400.

[24] David Smith, Alan Kay, Julian Lombardi, Mark McCahill, Rick McGeer, Andreas Raab, and David P. Reed. 2005. Croquet: A platform for Collaboration. In *Working with Vision workshop at OOPSLA 2005, San Diego, CA, USA, October 19.*

[25] Randall B. Smith, Mario Wolczko, and David Ungar. 1997. From Kansas to Oz: Collaborative Debugging when a Shared World Breaks. *Commun. ACM* 40, 4 (April 1997), 72–78. https://doi.org/10.1145/248448.248461

[26] Alessandro Warth, Milan Stanojević, and Todd Millstein. 2006. Statically Scoped Object Adaptation with Expanders. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. ACM, New York, NY, USA, 37–56.