Live Programming and Text Editor Integration in the Croquet Microverse 3D Collaborative Construction System

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

Aran Lunzer
Croquet Corporation
Los Angeles, CA, USA
aran@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 a web-based virtual 3D multiuser construction environment that supports a practical live-programming mechanism usable by professional programmers.

Developing a 3D collaborative application is time consuming. On each change of the application code, you need to load a new build onto all participants' machines, re-establish a situation that you were trying to affect, then perform an action to see whether the change has had the intended impact.

We have designed and created a full-stack software architecture to enable live programming in a 3D multiuser application called Croquet Microverse. The most notable feature of Microverse is the ability to integrate a regular text editor of the developer's choice into the real-time and collaborative live programming workflow, implemented on a general purpose network infrastructure.

In this paper we explain the Croquet architecture and the Microverse application and how its live programming feature works.

The standard version of Microverse is available at https://croquet.io/microverse, and its source code is available on https://github.com/croquet/microverse.

KEYWORDS

3D Application, Shared Experiences, Collaboration Application Framework, Live Programming

1 INTRODUCTION

Building a 3D collaborative application is difficult and time consuming. Even when collaboration is not involved, making a 3D application often requires arranging 3D models by modifying numbers in text, and then going through a laborious build process before getting to see the results on screen. When writing and modifying code for objects to respond to events, from the user or elsewhere, your workflow involves getting the objects into a state where they are ready for the events in question. Only then can you see if your code change has had the desired effect.

It is even harder to make a collaborative application. If the action you are writing requires multiple users to be in a certain state, you need to install the new build to all participating computers. If application code on the server is involved, you need to make sure that that code is in sync with the clients' code.

On the other hand, imagine how simple it would be if a group of colleagues could build an application in a "live" manner. Ideally developers and designers can work together in the same shared development session with a tight feedback loop. Each user would have their own input devices and displays. 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 created is code as well as visual appearances of objects.

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, having a live programming facility is even more important for a multi-user collaborative environment than a single-user one.

We have been developing a collaborative 3D construction environment called Croquet Microverse, addressing the needs of a broad range of users. Professional programmers can write application code in a live manner and share the changes in real time. End-users, who might not be programmers, can load models, and can use our object-extension mechanism called "behaviors" to customize the object by attaching and detaching self-contained code packages. In this environment, you can seamlessly transition between solo development and group real-time collaborative development.

The defining feature of Croquet Microverse is its text editor integration with its deterministic and real-time propagation of code changes to all participating peers. A developer can run a small server on the local machine and have the server watch code updates in a directory. When a code file change is detected, the server injects the change into the collaborative session. This feature is implemented on top of a general purpose collaboration framework.

Croquet Microverse is built on top of the Croquet OS, which is based on the replicated computation model [16] 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. It is carefully engineered to support real-world applications. For an overview of the current Croquet platform, see [2, 9].

Croquet Microverse uses the Croquet Worldcore 3D application framework, and Three.js as the rendering engine. Its code is available as open source at github.com/croquet/microverse, and

a reference installation is available at croquet.io/microverse. The code base is small (≈ 15,000 lines of commented code) so that a developer can understand, modify, and extend it.

The rest of the paper is organized as follows. Section 2 describes the foundational Croquet architecture. Section 3 introduces the Worldcore application framework. Section 4 describes the Microverse application. Section 5 explains the concept of the object extension mechanism we use, and Section 6 displays our collaborative live programming experiment within Microverse. Section 7 discusses related work.

2 CROQUET

Croquet is a platform for creating rich multi-user applications. As its library and associated back-end infrastructure perform the critical operating-system role of managing computational space and time on behalf of applications, we refer to the platform as the Croquet OS. 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 all peers receive the same sequence of external events, Croquet ensures that the 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, for transcendental functions).

Application code is executed purely on the peer machines. 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. 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.

When any peer needs to inject a state change into the session, typically due to an input action by its user, it informs the other 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.

3 WORLDCORE

Worldcore is an entity-management system that sits on top of Croquet OS.

Following the model-view separation requirement of the Croquet OS, but also borrowing terminology from the Unreal Engine [8], a model object is typically called an actor, and a view object is typically called a pawn. In the context of 3D programming, an actor holds information such as the object's translation, rotation, scale and parent-child relationships, as well as optional values such as the location of 3D-model or texture assets. The pawn takes those values and creates Three.js objects to render them.

Worldcore offers vector and matrix calculation features. It also offers useful features that fit well for Croquet. For example, motion of an object caused by a program on the model side can be "smoothed" (i.e., interpolated) on the view side on each animation frame. This relieves the model of responsibility for the full animation of the view's position, reducing program complexity and the number of Croquet messages needed between model and view.

While Worldcore provides reference implementations of event handling, spatial audio, a custom-made WebGL renderer and other features, we built Microverse on top of the "kernel" of Worldcore, that does the basic handling of actors and pawns.

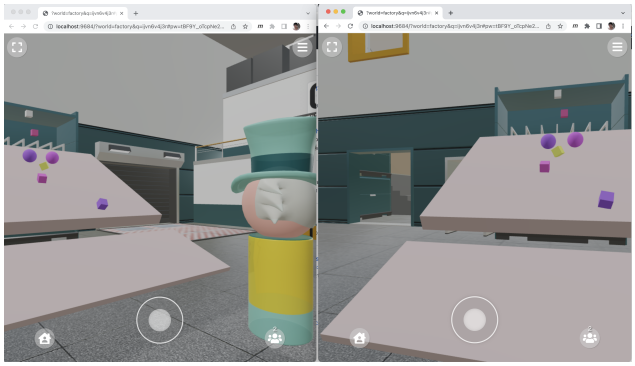

Figure 1: Two users' views of a Microverse space with a physics simulation.

4 MICROVERSE

Croquet Microverse is a web-based 3D collaborative construction environment. A user can enter a session by opening a URL in a browser. Additional users can join the same session at any time, and the state of the Microverse world will be perfectly synchronized among them all. Any user can perform actions such as uploading a new 3D model, or adding and removing "behaviors" for an object (see Section 5) to customize that object's actions.

To facilitate interactive update of object properties, there are some basic manipulation features for object repositioning, and an interface that we refer to as a property sheet [5] that supports viewing and editing of property values.

Furthermore, a user can connect a running Microverse session to what we call a watch server, that runs locally on the user's computer and watches for changes in code files. When the watch server detects a change, it immediately injects the new code into the running world, updating the behavior of the related objects. Like any other event, this form of update is replicated to all users in a bit-identical manner, so that their world state continues to be fully synchronized.

[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  Constants.DefaultCards = [
2    {
3      card: {
4        name:"world model",
5        type: "3d",
6        dataLocation: "./assets/3D/artgallery.glb",
7        modelType: 'glb',
8        singleSided: true,
9        shadow: true,
10       layers: ["walk"],
11       translation:[0, -1.7, 0],
12     }
13   },
14   {
15     card: {
16       name: "light",
17       layers: ["light"],
18       type: "lighting",
19       behaviorModules: ["Light"],
20       dataLocation: "./assets/sky/sky.exr",
21       dataType: "exr",
22     }
23   },
24   {
25     card: {
26       name: "image card",
27       translation: [12, 0.6, 10.77],
28       rotation: [0, -Math.PI / 2, 0],
29       type: "2d",
30       textureType: "image",
31       textureLocation: "./assets/images/Logo.jpg",
32       behaviorModules: ["Spin"],
33     }
34   },
35 ]
```

Figure 2: A part of a template file for Microverse

The Microverse system is built on top of Croquet OS and the kernel of Worldcore. We use Three.js as the rendering engine, providing state-of-the-art graphics.

The initial state of a Microverse world is described in a file that mostly consists of a JSON-like declaration.

A Microverse consists of a set of objects that we call cards. A card is an object with properties like 3D coordinates and specification of its visual appearance, along with a set of behavior modules and behavior-specific property values.

The 3D model for the terrain for an avatar to walk on is also a card (line 3). When the layers property (line 10) has `"walk"` as an entry, the code for all avatars treats this card as a walkable terrain, determining where the avatars can roam.

The lighting specification is also manifested as a card. A 3D scene typically needs elaborate lighting settings, which here have been defined in a behavior module called `Light` (line 19). It instantiates Three.js lights and puts them in the scene.

The third card is a `2d` type (line 29) card with the texture of `Logo.jpg`, with a translation and rotation also specified in the card spec.

Microverse comes with bindings to a deterministic 3D physics engine called Rapier [6]. The Croquet execution model allows a complex physics simulation to be synchronized perfectly across all users without any need to send states of the simulation over the network. Figure 1 shows a microverse session with a physics simulation. Two users are viewing the same simulation; the user whose view is on the left is also seeing the avatar of the other user.

You can have many different worlds in a single Microverse installation. A search param "`world`" in the URL specifies which world file to use. The URL also specifies a unique session name, which is how all users navigating to that URL arrive in the same session. If a world is loaded with no session name specified, Microverse automatically generates one and adds it to the URL, ready for sharing.

By default, you can move your avatar around in the Microverse world by the "joystick" at the bottom center of the screen. You can go to the spawn point by pressing the "home" button. The "gather" button starts the presentation mode: when a user hits this they become the presenter, and all other users' avatars follow the presenter through the space as if on a guided tour.

There is an event routing algorithm that supports dynamically modifiable rich interaction [4]. This event routing algorithm delivers a user event to the closest card that has a listener for the event, and that is on the ray cast from the avatar to the mouse pointer location. You can also specify a card to be a "first responder" to override the normal routing.

The user's avatar itself is a card and can have behaviors and receive events. The avatar often serves as the "last responder" of an unhandled event to provide additional user interaction.

5 BEHAVIORS

The single most important factor for enabling live collaborative programming is whether or not *code is also data*. All design aspects need to fall into a coherent system around this choice.

We designed an object-extension mechanism for Croquet, called behaviors. In the context of an earlier 2D collaboration system [14] we referred to this mechanism as instance-based expanders, but we now use the more generic term.

The behavior mechanism is inspired by past work such as Mixins [1], Traits [15], Expanders [17], as well as the way an application like Etoys allows attachable/detachable behavior in object-oriented programming [10], but its closest resemblance would be to the PIE [11] mechanism.

A behavior 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 a string. The system evaluates the string representation of a behavior to create a JavaScript class (a function object) so that it can be executed. Because its primary representation is a string, the application can update it, recreate the class object and replace it cleanly.

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 behaviors, "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 to that object. This means that the property names used in separately developed behaviors 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 behaviors outweighs the burden of taking care of potential clashes.

A behavior can invoke a method that the base objects implements, and it can invoke a method that is defined at the same behavior in the conventional method call syntax:

```
1  class TurnActor {
2    setup() {
3      this.addEventListener("pointerTap", "toggle");
4    }
5    toggle() {
6      this.turning = !this.turning;
7      if (this.turning) {
8        this.turn();
9      }
10   }
11   turn() {
12     if (!this.turning) {return;}
13     this.rotateBy([0, 0.1, 0]);
14     this.future(100).turn();
15   }
16 }
17
18 export default [{name: "Turn",
19                   actorBehaviors: ["TurnActor"]}];
```

Figure 3: An example of a behavior module

```
this.foo(10, 20, 30);
```

A behavior can also invoke a method defined within another behavior that is installed on the same base object, with a syntax that explicitly specifies the behavior (like the `#as:` message in PIE):

```
this.call("Other", "foo", 10, 20, 30);
```

This "call" syntax may be seen as going against modularity principles, but its use is rarely essential: the publish/subscribe communication mechanism that Croquet provides can be used to trigger invocations between behaviors, whether on the same object or not, obviating most needs for "call".

Figure 3 illustrates an actor-side behavior called `TurnActor`. `TurnActor`'s `turn()` method (line 11) rotates the object around the y-axis by 0.1 radians and then schedules another call to itself 100 milliseconds in the future (line 14).

In this example, the `setup` method, which is called when the behavior is attached to an object, sets up a listener for the `pointerTap` event to invoke the `toggle` method. The `toggle` method flips the `this.turning` flag, and if it became `true`, it starts the "future loop" of `turn()`.

The workings of the actor-side "`addEventListener()`" call merit some explanation, given that `pointerTap` is a view-side event that only a pawn can listen for. What the call does is to set up a listener in the pawn, along with an actor-side subscription to the event that the pawn will send when `pointerTap` happens. In that way, when any user's pawn detects a tap, all users' actors will be notified and will handle the event identically.

When you want to create visual appearances programmatically (by instantiating Three.js objects, for example), you write a pawn-side behavior. An example of a pawn-side behavior is shown in Figure 4.

In Figure 4, the `setup()` method sets up event listeners so that `pointerEnter` and `pointerLeave` trigger `hilite` and `unhilite`, respectively. The methods `makeButton` creates a Three.js Mesh, (after removing left over meshes at line 11). The property `this.shape` is the root of the Three.js object for the card's visual appearance. The `setColor` method finds the sphere mesh in `this.shape` and sets the `color` property.

Why is the line with `removeFromParent` (line 11) needed? It is because the `setup()` method is called when the user changes

```
1  class HilightSphere {
2    setup() {
3      this.addEventListener("pointerMove", "nop");
4      this.addEventListener("pointerEnter", "hilite");
5      this.addEventListener("pointerLeave", "unhilite");
6      this.makeButton();
7    }
8    makeButton() {
9      [...this.shape.children].forEach((c) => {
10       this.shape.remove(c)});
11     let THREE = Microverse.THREE;
12     let geometry = new THREE.SphereGeometry(0.15, 16, 16);
13     let material = new THREE.MeshStandardMaterial(
14       {color: 0xCC0000, metalness: 0.8});
15     let button = new THREE.Mesh(geometry, material);
16     this.shape.add(button);
17   }
18   setColor(color) {
19     if (this.shape.children[0]) {
20       this.shape.children[0].material.color.set(color);
21     }
22   }
23   hilite() {
24     this.setColor(0xFF0000);
25   }
26   unhilite() {
27     this.setColor(0xCC0000);
28   }
29 }
```

Figure 4: An example of a pawn-side behavior

the definition of the behavior dynamically. The base object may have a mesh created by the previous invocation of `makeButton()` (called from `setup()`). When the user updates the definition of the behavior, the state of the base object itself is kept unchanged; this is the state found during the subsequent invocation of `setup()`.

This manual management of life cycle of dynamically created objects is slightly cumbersome for a developer, but the system cannot automatically infer the developer's intent. We leave the developer to decide what to clean up or recreate and what to retain.

Another note is that the color change in this example does not send any event to the model; that means that the sphere's color change is only visible to the user who hovers the mouse pointer over it. A developer can and must make decisions on how an object behaves in a multi-user environment, determining which events are shared with other users and which are not.

As you can see, a behavior is written in unmodified JavaScript. We use the keyword `class` to declare a behavior so that we don't need to modify existing code analysis tools. The only global variable allowed is `Microverse`, which contains system objects like `THREE`, so that a behavior can be recreated from its string representation identically for all participants.

6 LIVE PROGRAMMING IN MICROVERSE

As described in Section 5, the code of a Microverse behavior is stored as text data. This means that if we provide a collaborative editor for changing that text, we can simply do live collaborative programming in the system itself.

In Figure 5, the card with the drone 3D model has a behavior called `CircleActor`, whose `step()` method specifies that when it is invoked, the "turn by a little, forward by a little, and repeat" action is executed, and it then schedules a `future()` message to itself. When any user edits the `step` code and saves the new definition by hitting

Figure 5: The definition of CircleActor can be changed from a collaborative text editor.

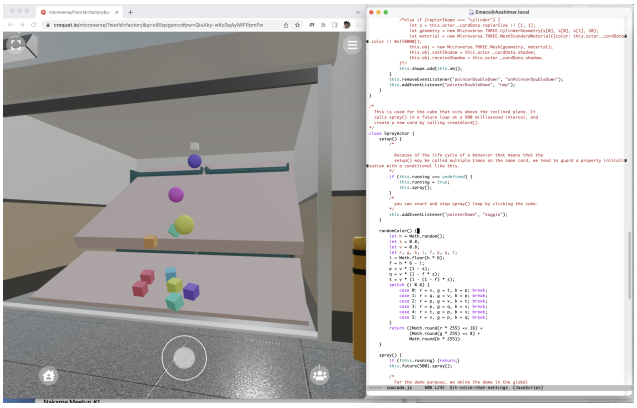

Figure 6: A text editor editing a physics simulation.

```
class AActor {...}
class APawn {...}

class BActor {...}
class BPawn {...}
class BPawn2 {...}

export default {
  modules: [
    {name: "A",
      actorBehaviors: [AActor],
      pawnBehaviors: [APawn]},
    {name: "B",
      actorBehaviors: [BActor],
      pawnBehaviors: [BPawn, BPawn2]},
```

Figure 7: A behavior module file with five behaviors

Cmd-S, the drone starts using the new definition immediately on all users' computers, in the next `step()` invocation. For example, if one were to change the argument for the `forwardBy` method from 0.03 to -0.03, everyone would see the drone reverse its direction of circling.

We decided to make it explicit that the `setup()` method of a behavior is executed each time the behavior code is udpated. A common case is to the state of an application and run the modified code, but sometimes the developer wishes to reset the state and start over a simulation. Both use cases are naturally supported in the mechanism. We think that it is not a burden for a developer to explicitly initialize a part of application state of her choosing. A program in MIT Scratch, for example, retains the application state, so it is common practice for novice programmers to write code to reset state in initialization.

As we experiment with the live programming feature, we realized that programmers who are accustomed to their own text editor (in some cases showing almost religious attachment) would prefer to continue using that editor to code in our system. To provide the benefit of live programming for those programmers, we built what we call the watch server.

The watch server watches file changes in a directory, and accepts a Websocket connection from a browser tab that is running Croquet Microverse. Crucially, browser rules allow a connection to a Websocket server running on localhost even from a page that has been loaded over the Web; this enables the watch server to be used wherever the world itself has been loaded from. This feature is similar to the hot module reloading feature provided by `webpack-dev-server` and other mechanisms. But instead of using an existing implementation, we wrote a simple server in 150 lines of Node.js code.

Figure 6 shows an example of a live text-editor integration session. A complex physics simulation program can be edited *while it is running*, and code changes are immediately reflected to all participants of the session deterministically.

As a behavior file may contain multiple behavior modules, and each behavior module may contain multiple behaviors, we need to extract individual behavior definitions from a file. For example in Figure 7, the text representation of the five defined behaviors needs to be extracted from one file.

One could imagine using a JavaScript parser and analyzing the results, but we employed a much simpler approach based on a built-in browser feature. When the peer first receives from the watch server the text that constitutes an ES module, the peer creates a DOM `script` element with the received module code (after creating an `objectURL` from it) as its `src`. Once the `script` element has loaded its code from `src`, we can access the exported `Module` object and then extract all actor and pawn behaviors.

The behavior definitions are then converted to strings by calling `toString()`, and they are sent to all peers as a series of replicated events. Upon receiving the string, each user's model evaluates it to create the defined JavaScript functions.

If code is saved with a syntax error, the evaluation fails on the first peer and no harm is done to the system. When code is syntactically correct but contains a logic error in the model side, the error causes the same effects on all peers. If the error happens to be recoverable, the session can be fixed, but if not, the session may be rendered unusable.

The effects of a logic error on the view side can vary widely. In most cases it can be recovered by saving a version of the source code. However, an error that results in `Infinity` or `NaN` in a Three.js data structure could cause an unrecoverable crash, which would typically require you to start a new session. The good news here is that the files you have written are all saved to disk, so you have

not lost any work. This is similar to the role of the `.changes` file in Smalltalk.

The text-editor integration is the defining feature of Croquet Microverse. It is simple but useful. A classic development process of a collaborative application, especially when you would like to test a new feature with multiple users, is to rebuild the application, and then re-launch it for all participants (not to mention to update the server if it involves a server component), and recreate the situation where your changed code is relevant, and then try the same action with others. This is a time-consuming process. In Croquet Microverse, you edit the code and that is it; the object whose behavior you are editing can be right in front of everyone's eyes, and everyone can try out its new behavior right away.

In practice, a developer might spend a majority of their time writing Microverse code alone, rather than in a shared session. The text editor integration helps just as much in that setting, as a live change takes effect immediately. The developer can open additional browser windows onto the same running session to experiment quickly, and when ready to try with other users can invite them into the same session too.

7 RELATED WORK

(N.B. Please refer to the Related Work Section of our previous paper.)

The replicated computation model can be traced back to TBAG by Elliott et. al [7]. 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.

There are some attempts to make the development of 3D collaborative applications (metaverses) more productive. For example, the Meta company's Horizon platform has a blocks-based programming interface. However, the definitions of the blocks still have to be written in C#, and testing requires a lengthy build and the restarting of the Unity-based application if you need a new block. While Microverse does not offer a blocks-based programming interface yet, we anticipate that our interface will be more uniform, including allowing extension of the blocks editing system itself from within the system.

Mozilla Hubs [3] is a Web/JavaScript based 3D shared environment. Despite some efforts to ease the development effort, Hubs still requires cumbersome building steps and reloading. Also note that the synchronization is based on sending values of properties, which is not sufficient as the basis for a complex shared application.

The concept of a self-sustaining live programming environment on the web was influenced by a series of implementations of Lively Web [12]. Lively Web, in turn, was influenced by Smalltalk, and we borrow many ideas including the live-programming idea itself, and also treating a class as data to implement meta-features.

The reports of experiments by Misback et al. [13] are also of interest to the LIVE community.

8 CONCLUSION

This paper describes the live-programming feature and text-editor integration of the Croquet Microverse 3D collaborative construction application. Microverse is a web-based application built on top of the Croquet OS and the Worldcore 3D application framework.

The text-editor integration is novel and has already shown its value in professional development, helping to unleash the power of live programming.

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] 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.

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

[3] Mozilla Corporation. 2018. *Mozilla Hubs.* Mozilla Corporataion. https://hubs.mozilla.org.

[4] Croquet Corporataion. 2022. *Event Routing of Microverse.* Croquet Corporation. https://croquet.io/blog/june2022/event-routing/.

[5] Croquet Corporataion. 2022. *The Property Sheet of Microverse.* Croquet Corporation. https://croquet.io/docs/microverse/tutorial-PropertySheet.html.

[6] Dimforge. 2020. *Rapier.* Dimforge. https://rapier.rs.

[7] 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

[8] Epic Games. 2022. Unreal Engine Documentation. https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Framework/Pawn/.

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

[10] 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.

[11] 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

[12] 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.

[13] Edward Misback and Steven Tanimoto. 2021. Peer-to-peer Syncing and Live Editing of Shared Virtual 3D Spaces: Challenges and Opportunities. In *The LIVE Workshop '21*. Association for Computing Machinery, New York, NY, USA.

[14] Yoshiki Ohshima, Aran Lunzer, Jenn Evans, Vanessa Frudenberg, Brian Upton, and David A. Smith. 2022. An Experiment in Live Collaborative Programming on the Croquet Shared Experience Platform. In *The Programming Experience Workshop '22*. Association for Computing Machinery, New York, NY, USA, (to appear). also available at https://tinlizzie.org/IADocs/live--programming--greenlight.pdf.

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

[16] 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*. ACM, New York, NY, USA.

[17] 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.