Towards Liveness in Game Development

Andrew R Martin[1] and Simon Colton[1,2]
[1]*Game AI Research Group, Queen Mary University of London, UK*
[2] *SensiLab, Faculty of IT, Monash University, Australia*
andrew.m@qmul.ac.uk

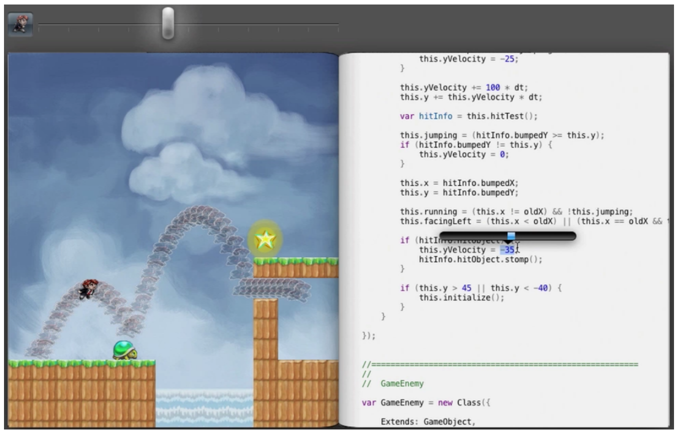
***Abstract*—In general, videogame development is a difficult and specialist activity. We believe that providing an immediate feedback cycle (liveness) in the software used to develop games may enable greater productivity, creativity and enjoyment for both professional and amateur creators.**

There are many different methods for achieving interactivity and immediate feedback in software development, including read-eval-print loops, edit-and-continue debuggers and dataflow programming environments. These approaches have each found success in domains they are well-suited to, but games are particularly challenging due to their interactivity and strict performance requirements. We discuss here the applicability of some of these ideas to game development, and then outline a proposal for a live programming model suited to the unique technical challenges of game development. Our approach seeks to provide an extensible way to automate the process of obtaining feedback, through the use of a reactive programming model and dataflow-style UI. We describe our progress in implementing this design, with reference to a simple example game.

***Index Terms*—Game Development, Live Programming, Reactive Programming**

I. Introduction

Chris Hancock conveyed the dramatic effect that continuous feedback can have in a programming environment with an archery vs. water hose analogy [1] as follows. A traditional programming feedback cycle can be compared to an archer trying to hit a target. They fire one arrow at a time, able to make adjustments only when they see where their arrow lands, which could go on indefinitely. In contrast, it would be trivial to hit the same target using a water hose, simply by moving the hose until the water hits it. In 2012, Bret Victor gave a presentation entitled *Inventing on Principle* [2], in which he advocated for tools which provide an immediate connection between creators and the things they create. This presentation included a highly influential demonstration of a game development tool, portrayed in figure 1. The presentation inspired many research projects in academia and industry, such as Apple's Swift Playgrounds (apple.com/uk/swift/playgrounds) and Chris Granger's Light Table (chris-granger.com/lighttable), which crowdfunded investment from more than 7,000 backers.



```
            this.yVelocity = -25;
        }

        this.yVelocity += 100 * dt;
        this.y += this.yVelocity * dt;

        var hitInfo = this.hitTest();

        this.jumping = (hitInfo.bumpedY >= this.y);
        if (hitInfo.bumpedY != this.y) {
            this.yVelocity = 0;
        }

        this.x = hitInfo.bumpedX;
        this.y = hitInfo.bumpedY;

        this.running = (this.x != oldX) && !this.jumping;
        this.facingLeft = (this.x < oldX) || (this.x == oldX &&

        if (hitI
            this.yVelocity = -35;
            hitInfo.hitObject.stomp();
        }

        if (this.y > 45 || this.y < -40) {
            this.initialize();
        }
    }

});


//=========================================================
//
//  GameEnemy

var GameEnemy = new Class({

    Extends: GameObject,
```

Fig. 1. Bret Victor's time-travel demo. The top slider controls the timeline. The slider in the code changes a variable in real-time, affecting the in-game character.

In this atmosphere of excitement, we might have expected the ideas from this presentation to make their way into game development tools in the 7 years that followed, but this hasn't happened. Part of the reason is that the demos were only intended to convey ideas rather than robust technical solutions, as discussed in [3], and generalising the ideas to general-purpose development environments has proven challenging. One area where huge strides forward have been taken is in front-end web development. In particular, the release of the *Elm* programming language [4] and the *React* web framework [5] provided the foundations for robust, generalised programming tools, such as hot reloading with state preservation and time-travel debugging [6], [7].

To apply the techniques used in Elm and React/Redux frameworks to game development, one must overcome performance limitations. In particular, these frameworks rely on persistent data structures, as described in [8]. It can be challenging to build a high-performance game simulation with data structures like these. Furthermore, as Victor argued in his presentation, liveness of code on its own is often insufficient to create a powerful connection between a creator and their work. To really unlock the value of this programming model, one approach would be to build a tool that users can reshape on the fly and extend to suit their purposes. In particular, the node-based dataflow programming paradigm [9] has a natural affinity for liveness and extensibility: by connecting live nodes together, users build their own interface. This is demonstrated in environments such as Dynamo [10] and Jonas Gebhardt's visual programming demo [11], where any React component may be used as a node in the graph.

We are currently building a generalised programming environment in which live feedback is given, and using this to experiment with ways in which continuous feedback can benefit game developers. We also aim to provide a blueprint

978-1-7281-1884-0/19/$31.00 ©2019 Crown

that existing game development tools can use to begin improving their own feedback cycles. To describe this work in progress, in section II, we survey four approaches to interactivity in game development and highlight the pros and cons of each. In section III, we describe the design for our new game development environment, which combines textual programming with a visual node graph interface, and we provide details of our implementation progress. We conclude with a discussion of future work.

II. Interactivity in Game Development

Interactivity in software development has a rich history, and elements of interactivity have long been included in game development toolkits. The four approaches described below are not meant to be exhaustive, but do give a flavour of the kinds of ways in which liveness can be employed in software engineering.

A. REPLs and Notebooks

Read-eval-print loops, or REPLs, are a very successful UI paradigm in software development. Popularised by the Lisp family of languages, they are now the basis of most scientific programming environments. The rapid feedback cycle they enable allowed programming languages to become a tool for thought, rather than just a way to specify behaviour. REPLs are often embedded in game loops for querying and debugging game state, and injecting commands for experimentation.

Any modifications made using a REPL are usually temporary. This makes them excellent for exploration and introspection, but poorly suited as a primary method for authoring code. This problem is usually overcome by exposing REPL functionality through the notebook UI paradigm, in which a user edits a script and is able to evaluate different parts of it at will, visualising the results between blocks of code. However, it is not clear how much benefit a notebook would offer beyond a traditional code editor for creating a game. The user would likely need to run the whole contents of the notebook to launch or update the game, just like running a normal program. Notebooks might be well-suited to experimenting with game asset pipelines, but do not offer an obvious way for the user to interact with live game behaviour.

B. The Smalltalk model

Smalltalk pioneered a programming model, runtime technology and visual UI paradigm that allowed software to be edited as a live artefact [12]. A user can inspect a Smalltalk program via a class browser and make arbitrary changes to it while it is running. These changes are retained as a permanent part of the program, making this viable as a primary mode of development. Many popular game development tools feature editors which could be compared to the Smalltalk class browser. They allow users to browse and modify entities, components, scripts and assets, all while a game is running. However, changes to dynamic state cannot be saved because they have no clear meaning; for instance, what does it mean to pause a game of tetris, move a falling block slightly to the left, and then hit save?

Changes to static data or code might not provide any feedback, as they may have already affected the state of the game and won't necessarily ever affect it again. For example, changing a variable which controls the size of enemies in a game may not affect the enemies that have already spawned, depending on how the game is programmed. In fact, the game may now contain enemies of two different sizes, which is not representative of the code and can never happen again. These live editing tools are powerful, but if used without planning or structure, they are likely to produce incoherent results or no visible results at all.

C. Dataflow and Reactive Programming

Dataflow and Reactive programming are broad fields with much in common [13]. Programs are typically expressed as transformations over some set of inputs, which may be discrete event streams, continuous signals or both, depending on the semantics of the system in question. The state and output of the program can be calculated deterministically from these inputs. Calculations are generally modelled as a directed, acyclic graph of cascading changes, starting from the inputs. It is this dependency tracking that enables a reliable feedback cycle. Spreadsheets, for example, are one of the most successful examples of a reactive system. Changes made to the data or formula in one cell immediately trigger the recalculation of all dependent cells. One of the strengths of these tools is to provide granular, incremental feedback. Intermediate results can trivially be visualised, and changes trigger minimal recalculations.

There have been many attempts at writing games using reactive programming or dataflow systems, but no mainstream tools are built around either approach. Some game engines use node-graphs to express behaviour, but they usually rely on traditional control-flow evaluation semantics, rather than dataflow semantics. Most game development tools struggle to embrace dataflow programming because they depend heavily on the use of mutable data structures and shared-pointers, both for performance and ease-of-use. This makes it very easy to introduce accidental coupling in ways that would break a dataflow or reactive programming model. However, dataflow programming is still used in some parts of game development. For example, node-based shader editors and VFX tools like Houdini both incorporate concepts and UI metaphors from dataflow programming.

D. Reactive GUI frameworks

Reactive GUI frameworks like Elm and React are usually modelled as a tree of components which may hold state, such that each implements the same simple interface: they can initialise their state, update their state in response to an event, and render themselves. There are many variations on this model, including versions which handle state updates using reducers and immutable data structures. This variation is the one most commonly used to build tools like time-travelling debuggers, as it is trivial to cache previous component states by holding immutable references to them, allowing them to be re-rendered on command.

Each update to a reactive component causes the entire tree of components to re-render itself. Although this is the model presented to the programmer, behind-the-scenes optimisations may mitigate much of the cost of this. The simple component interface described above is very similar to a basic game loop. Game engines also typically re-render the whole screen according to their internal state, although this may happen on a timer instead of in response to changes.

III. Designing a new system

We are building a reactive, node-based game development tool, wherein a node graph processes streams of discrete events and uses a combination of both push and pull semantics, as discussed in [13]. Nodes in our tool can hold mutable state, but by replaying input event streams, we can visualise, explore and transform the history of this state. This functionality will be used to build tools like time-travelling debuggers. Nodes are implemented using a simple, text-based imperative programming language. Nodes may also be populated with custom GUI elements, such as data views or interactive editors. These custom nodes are implemented using the tool itself. Game logic and libraries are expressed primarily using the imperative programming language. The node graph's purpose is to provide a flexible, user-friendly mechanism for obtaining useful feedback.

As discussed in section II-B, existing game development tools often can't provide immediate feedback in response to live changes. This forces the user to perform a series of additional manual interactions every time they need feedback, such as manually playing through part of their game. We intend to show that many interactions like these can be concisely expressed using reactive nodes instead, such that feedback becomes immediate and automatic.

A reactive node graph is well-suited to expressing live asset pipelines and other compile-time processing steps. Thanks to the reactive nature of the graph, we believe it can also be used to recreate and expand upon the ideas discussed in section II-D, which aim to provide feedback for interactive processes in time (such as GUI applications).

A. *Reactivity without immutability*

Reactive programming systems are often built using components (or cells) that can update each other in response to changes. These updates are performed deterministically and efficiently by maintaining a graph of all dependencies between components. If two components become accidentally coupled in a way that is not represented in the dependency graph, the program will no longer function correctly. In most programming languages this can happen very easily, through the ubiquitous use of shared mutable pointers. Building systems entirely from immutable data structures is one way of avoiding this, which is why it is the foundation of many reactive systems.

It is challenging to build a high performance game simulation with only immutable data structures. Instead of relying on immutability, we can avoid accidental coupling by preventing mutable state from being shared amongst more than one component. Dependent nodes may borrow and even re-export immutable references to these data structures, as long as they do not store them. This can be achieved by preventing mutable references from passing between nodes, using the semantics and type system of the programming language.

B. *The programming language*

Ease-of-use is very important for an interactive programming environment. However, performance is also very important for games. In recent years, languages like Swift (swift.org) and Julia [14] have demonstrated that careful language design can offer performance close to systems languages (such as C and C++), while retaining the memory safety and convenience features of productivity languages. Similar performance is now being achieved in C#, thanks to the *High-Performance C#* subset and the Burst compiler developed at Unity [15]. Once we have stabilised our design, it is possible that languages such as these will be suitable alternatives, bringing the system to a wider audience.

C. *Extensibility*

The node graphs in our system are able to define visual, event-driven applications, and so they may also be used to define arbitrary GUI extension nodes. These nodes can be defined in the manner of reactive GUI components, as described in section II-D. In this way, the editor is live and capable of self-modification, in the tradition of systems like Smalltalk. An example featuring a hypothetical pixel editor extension node is illustrated in figure 2.

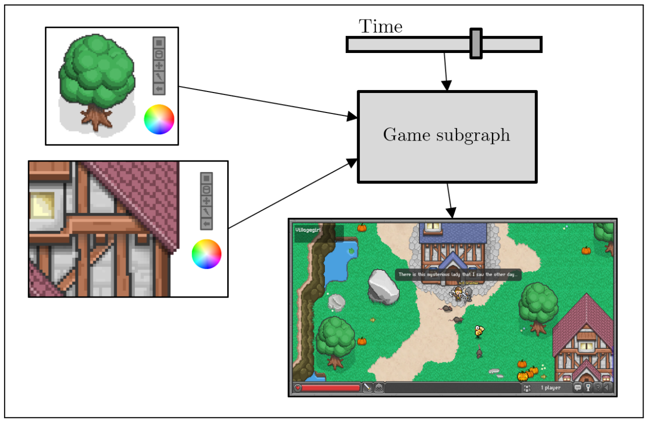
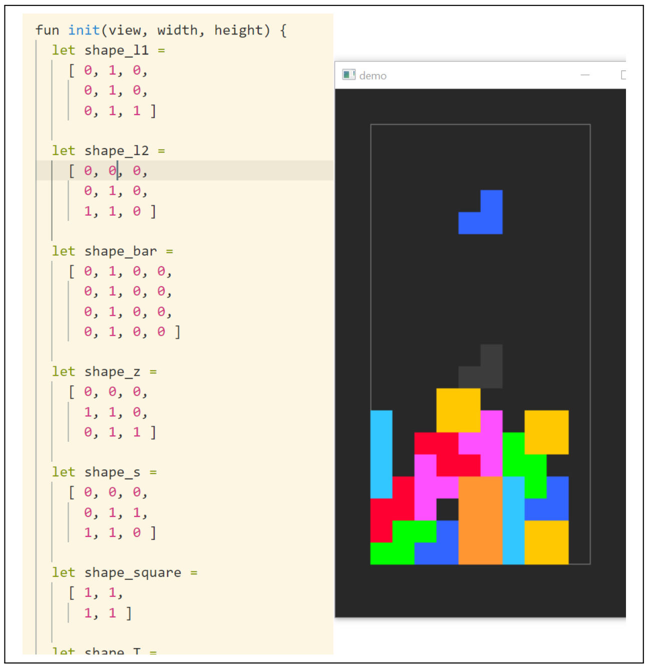


Fig. 2. Hypothetical illustration of pixel art editing nodes feeding into a running game, with another widget for scrolling through time in the game world. Assets borrowed from BrowserQuest (github.com/mozilla/BrowserQuest).

D. *Implementation Progress*

Currently, we have implemented an initial version of the programming language. We have used it to implement a basic text-based prototype of our reactive programming model, which has in turn been used to implement a simple demo of a tetris-like game. This demo supports hot reloading and event-replay, such that changes to game behaviour or the game's initial state cascade through the entire history of the play-through in real-time. It is depicted in figure 3.

The program is divided into two modules: a controller module and a game component module. The game component exposes an initialisation function, update function and rendering function. The controller initialises an instance of the game component and runs an event loop. When it receives events, it logs them and passes them to the game. This includes periodic tick events, which trigger both an update and render step. When the programmer changes the code for the game, the controller discards the game component and initialises a new one. It then runs a tight loop providing all previous logged events to the component. This provides a basic live update loop and could be extended to give the user control over the timeline. However, there are obvious shortcomings. In particular, any change made to the rendering code will cause the entire event log to be needlessly reprocessed. In the full reactive node-based system, these functions would be split across multiple nodes with their dependencies clearly expressed in the graph, averting this problem.



```
fun init(view, width, height) {
  let shape_l1 =
    [ 0, 1, 0,
      0, 1, 0,
      0, 1, 1 ]

  let shape_l2 =
    [ 0, 0, 0,
      0, 1, 0,
      1, 1, 0 ]

  let shape_bar =
    [ 0, 1, 0, 0,
      0, 1, 0, 0,
      0, 1, 0, 0,
      0, 1, 0, 0 ]

  let shape_z =
    [ 0, 0, 0,
      1, 1, 0,
      0, 1, 1 ]

  let shape_s =
    [ 0, 0, 0,
      0, 1, 1,
      1, 1, 0 ]

  let shape_square =
    [ 1, 1,
      1, 1 ]

  let shape_T =
```

Fig. 3. A tetris-like game, depicted after a live modification has been made to the shape of one of the game's block types (the blue L-shape), affecting the whole history of play.

IV. Future Work

The next step is to implement the node GUI for our reactive programming model. We will then implement the nodes and UI required to replicate the original Bret Victor game development demo, as a proof of concept for our system. Following this, we will implement several other demos to test the generality of the system. Once we have stabilised the design of our system, we can evaluate the value and functionality it provides in contrast to state-of-the-art game development tools. We will also formalise descriptions of the reactive model and key programming language features required, and propose ways to integrate the functionality demonstrated into existing tools.

We hope to show that liveness in games programming can increase a game designer's productivity and creativity, as well as the enjoyment they gain from the creative process. This is in line with recent thinking around so-called *casual creators* [16], where the fun of making is equal in importance (or more so) than the value of the artefact produced. Casual creators for game design, such as the Wevva iOS app [17], are being used to lower barriers to entry and democratise videogame development. We plan to compare and contrast such casual approaches with liveness-enhanced games programming and possibly to suggest hybrid approaches which might combine the best aspects of choice-based design and live programming.

References

[1] C. M. Hancock, "Real-time programming and the big ideas of computational literacy," Ph.D. dissertation, Cambridge, MA, USA, 2003.

[2] B. Victor. Inventing on principle. [Online]. Available: https://vimeo.com/36579366

[3] G. Bracha. Room 101: Debug mode is the only mode. [Online]. Available: https://gbracha.blogspot.co.uk/2012/11/debug-mode-is-only-mode.html

[4] E. Czaplicki and S. Chong, "Asynchronous functional reactive programming for GUIs," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM Press, Jun. 2013, pp. 411–422.

[5] Facebook. React - a JavaScript library for building user interfaces. [Online]. Available: https://reactjs.org/

[6] D. Abramov. Hot reloading in react. [Online]. Available: https://medium.com/@dan_abramov/hot-reloading-in-react-1140438583bf

[7] M. James. Time travel made easy: Introducing elm reactor. [Online]. Available: http://elm-lang.org/blog/time-travel-made-easy

[8] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," *Journal of Computer and System Sciences*, vol. 38, no. 1, pp. 86 – 124, 1989.

[9] W. Johnston, J. Hanna, and R. Millar, "Advances in dataow programming languages," *ACM Computing Surveys*, vol. 36, pp. 1–34, 2004.

[10] A. Basman, L. Church, C. N. Klokmose, and C. B. Clark, "Software and how it lives on-embedding live programs in the world around them." in *PPIG. 2016*.

[11] ReactEurope, "Jonas Gebhardt - evolving the visual programming environment with React at React-europe 2016." [Online]. Available: https://www.youtube.com/watch?v=WjJdaDXN5Vs

[12] A. Goldberg, *Smalltalk-80: the interactive programming environment*, ser. Addison-Wesley series in computer science. Addison-Wesley, 1984.

[13] E. Bainomugisha, A. L. Carreton, T. Cutsem, S. Mostinckx, and W. Meuter, "A survey on reactive programming," *ACM Comput. Surv.*, vol. 45, no. 4, pp. 52:1–52:34, 2013.

[14] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," *arXiv:1209.5145*, 2012.

[15] On DOTS: C++ & c# unity blog. [Online]. Available: https://blogs.unity3d.com/2019/02/26/on-dots-c-c/

[16] K. Compton and M. Mateas, "Casual creators," in *Proceedings of the International Conference on Computational Creativity*, 2015.

[17] S. Gaudl, M. Nelson, S. Colton, R. Saunders, E. Powley, B. Perez Ferrer, P. Ivey, and M. Cook, "Rapid game jams with fluidic games: A user study and design methodology," *Entertainment Computing*, vol. 27, 2018.