Towards a Reactive Game Engine

João Paulo O. Marum
Department of Computer and Information Science
University of Mississippi
University, MS, USA
jomarum@olemiss.edu

J. Adam Jones
Department of Computer and Information Science
University of Mississippi
University,MS, USA
jadmj@acm.org

H. Conrad Cunningham
Department of Computer and Information Science
University of Mississippi
University, MS, USA
hcc@cs.olemiss.edu

***Abstract*—Developing new ways to improve game engines and enabling the creation of more accurate experiences is an important issue, especially for real-time applications such as Virtual Reality. As this technology has evolved, the gap between rendered frames has become smaller and environments have become more complex. As a result, mistakes can arise with regard to order-dependent events. Often these mistakes result in a temporarily unstable state in the environment, and may not be visible because of the time lapse between rendered frames. These errors are compounded by events which are not directly driven by the game engine, such as user movement and input. There is a continuous interaction between the user and the environment and the precise actions that user may take often cannot be predicted a priori.**

Reactive programming is a still evolving paradigm that has been employed in many different applications such as robotics and user interfaces. Our research presents a reactive, dependency-based framework for game engines using associations between components to establish the order in which they update. Automated tests were developed to evaluate the framework and compare its performance to Unity3D's default update structure as well as an existing reactive programming framework.

***Index Terms*—reactive programming, game engines, dependency graph, script**

I. Introduction

Our research presents a reactive, dependency-based framework for game engines. The reactiveness is defined by detecting the dependencies between components and encoding them in a dependency graph. These dependencies are then flattened into a priority queue, setting the order in which the components' update functions should be called by the reactive component.

This project's practical objective is to guarantee the execution order of the components' update functions on script-based game engines without degrading performance while preserving the correctness and increasing the predictability of order-sensitive operations.

978-1-7281-0137-8/19/$31.00 ©2019 IEEE

The correctness and predictability come from the ability to accurately determine the chain of reactions that occurs after a given action is taken by the user or by a non-human actor in the system. This is guaranteed because the chain of reactions completes within a single update cycle with no side effects. The final state of the tree is deterministic allowing predicted results to be compared with the system's performance.

We test our framework using Unity3D [1], a commonly used game engine. It does not currently allow programmers to specify the update order for the components within a game at runtime. As a result, programmers must take appropriate countermeasures to maintain accurate game states. For example, they can ignore updates that occur in the wrong order or that use outdated inputs since these would result in incorrect game states.

The predictability and determinism of the internal interactions between components in game engines are especially important for graphical, order-sensitive systems that may require high accuracy, such as simulations of mathematical, physical, and chemical interactions. It is necessary for interactions in the simulated world to occur in the same order as they would in the real world.

In this research, we built a framework as a component to be embedded as a script at the root of the game tree. To evaluate the effectiveness of this approach, we measure how well the proposed solution guarantees the accuracy and predictability of simulations, mathematical calculations, and logical evaluations.

Using a small platform as a proof of concept, we have tested our approach with a wide variety of scene complexities. These include mathematical calculations and a small shooting game. We compare the results achieved for these scenarios to the results achieved by similar applications built on UniRx [2], the reactive extension for Unity3D.

II. Background

A. Reactive Programming

Bertoluzzo [3] defines reactive programming as a programming paradigm that focuses on how the software should react

Imperative	Reactive
A = 1 B = A + 1 C = B + 1 A = 10 Result: A = 10 B = **2** C = **3**	A = 1 B = A + 1 C = B + 1 A = 10 Result: A = 10 B = **11** C = **12**

Fig. 1. Difference between imperative and reactive programming [4].

to changes in the system's state. A change may be caused by either an internal mechanism—such as some calculation setting the value of a variable—or by a user-initiated external event.

Westberg [4] describes this approach as follows: "When the value of a specific cell is changed, all the dependent cells to that cell are instantly updated as a function of that change. This type of model makes it possible to have sequences of dependent values and events. If object C is dependent on B, while B is dependent on A, then, if A changes, B is reevaluated, followed by C. Changing object A creates a chain of reaction." A change in the value of A causes B to be recomputed based on A's new value. A change in the value of B, in turn, causes C to be recomputed because of the changes in the value of B. The changes thus ripple through the entire graph of dependencies. This is the familiar computing paradigm used in spreadsheet software. It gives a push-based or event-driven model of computation, where the environment rather than the program determines the speed at which the program interacts with the environment. Figure 1 illustrates this behavior.

Vasiv [5] characterizes the general class of programming problems that are solvable by reactive programming as problems where there is a continuous interaction between the program and the environment. In this class of problems, any meaningful event or interaction depends on the user's choices; the event thus cannot be predicted beforehand.

One way to achieve reactive programming is to manage the data dependencies. In this work, we represent the data dependencies by encoding them in a directed acyclic graph (DAG), where the nodes represent individual states (i.e. variables) and edges represent direct dependencies between states.

At the start of the application, the framework creates the dependency graph. It then traverses this graph whenever a change occurs to any data item. It continues traversing the graph as long as changes are propagated from one node to others. When it stops, the graph remains stable until the next chain of reactions.

The benefit of reactive programming becomes more evident in applications that are highly interactive with a multitude of user-interface events that can change data values. Applications that can be improved by the reactive approach include robotics, autonomous vehicles or drones, virtual reality applications, and web and mobile applications.

B. Game Engines and Game Development

Sherrod [6] describes the game engine as "a framework comprised of a collection of different tools, utilities, and interfaces that hide the low-level details of the various tasks that make up a video game".

Cowan [9] and Lavalle [8] summarize the main features provided by several game development frameworks. The features include:

- Scripting—controlling the game objects and events
- Rendering—generating the 3D scene with sufficient speed and accuracy
- Animation—moving and deforming objects (such as characters)
- Artificial Intelligence—carrying out behaviors normally requiring "intelligence" (e.g. path finding)
- Physics—depicting realistic physical behaviors in the scene (e.g. when objects collide or when force is applied to an object)
- Audio—rendering sounds to have an appropriate spatial location withing the environment
- Networking— provide online interaction with other human actors inside the environment by sharing information through a network

The code that provides each functionality is used along with the others to produce the finished game. A graphical user interface (GUI) is commonly provided by the game engines which ties together several editors. Cowan [9] and Lewis and Jacobson [7] lists the tools commonly included within the game engines:

- Scene or Level Editor—enables creation and modification of virtual 2D or 3D "worlds"
- Script Editor—supports customization of object behaviors by attaching scripts to the objects
- Material Editor—enables creation and modification of object surfaces and visual effects by combining Shader code and images
- Sound Editor—supports combining sound settings with filter and other general effects provided by the sound engine

A commonly used game engine is Unity3D [1], which is described by Hocking [10] as "a powerful graphics platform and game engine that became widely used in the field because it provides physics simulation, normal maps, screen space ambient occlusion, dynamic shadows and many other professional graphics resources." Many other game engines boast such features, but Unity3D has three main advantages over other contemporary game development tools, as described by Smith and Queiroz [11]: a productive visual workflow, support for cross-platform development, and a modular component system used to construct game objects.

Unity3D uses component-oriented programming. It attaches scripts to objects as components. Thus one object can be

many different things (i.e. has many inheritances), giving programmers the ability to mix and match functionality on every game object. Every script provides programmable aspects for the application, modelling any desired behaviour that could be attached to one or more game objects. Every script adds functionality to the game object. Cube objects have a Cube component, Sphere objects have a Sphere component, and so on.

Games in Unity3D are composed of multiple Game Objects that contain meshes, scripts, sounds, or other graphical elements such as Lights. Unity3D event system execute every script by calling special functions that every script contains, even implicitly. Once a function has finished executing, execution is passed back to Unity3D. Unity3D uses a naming scheme to identify which function to call for a particular event during gameplay.

Unity3D does not define an explicit order of execution. It executes components in reverse order of their creation or modification. It executes the most recently created or modified objects first, because that is how the objects are placed in the self-generated metadata used by the compiler to determine contextual information about the scripts. If the game requires a specific update order, the programmer can use the Inspector to order the script execution. But the order must be known beforehand. It must be changed manually before execution begins; it cannot be changed during execution. That is an error-prone and time-consuming method that seems inappropriate for a complex animation with many objects. Unity3D lacks the capability to programmatically change the component update order at runtime.

In its only reference to this issue, the Unity3D manual [1] states: "By default, the `Awake`, `OnEnable` and `Update` functions of different scripts are called in the order the scripts are loaded (which is arbitrary). However, it is possible to modify this order using the Script Execution Order settings." However, what it does not say is in which order a number of `Update` functions are executed. This can be important as each update can change things that then affects the next update.

There is no defined order to `Update` functions. If a specific order is needed, then programmers must implement it themselves. For example, the program could create an array of objects and, from a single `Update` function, call functions on those objects in the array in order or could create a single "game manager" object that updates the other game objects in the scene in the specific order.

III. Related Work

In the previous section, we identified the need to control the order that updates are performed if we wish to create an accurate simulation of the real world. Many other problems have similar characteristics. They must perform tasks in a particular order known beforehand (such as according to a dependency graph) or they must modify the order dynamically according to the stimuli received or generated. Researchers are tackling these problems using using reactive programming in areas such as robotics [12] [13] [14], chatting AI bots [15], autonomous drones [16] or vehicles [17], graphical user interfaces (web or desktop based) [18] [19] [20] [21] [22], biology-emulating behavior systems [23], telecommunications [24] arts [25]), and education [26].

Czaplicki [27] and Blackheath [28] set the background on reactive programming and examined the development of reactive libraries (Elm and Sodium) from scratch. Furness [29] examines the theoretical aspect of the development and design of virtual environments.

Some researchers have also looked at how to use reactive programming to improve the graphics platform with interesting results. Blom and Beckhaus [30] examined the development of a system called Functional Reactive Virtual Reality. This system implements a Haskell library that inserts a reactive layer between the user interface and the virtual environment manager. However, this work is no longer contemporary and does not address the level of complexity seen in modern game engines.

The general characteristics of the research examined in this section are that the interactions between the user and the system are unpredictable, and accurate results require that various internal state changes occur in a correct order. Every decision must be taken with the latest available information from the inputs.

IV. Implementation

Our research presents a reactive, dependency-based framework for game engines using associations between components to establish the order in which they update. This section describes our implementation. Figure 2 shows the overall operation of the framework.

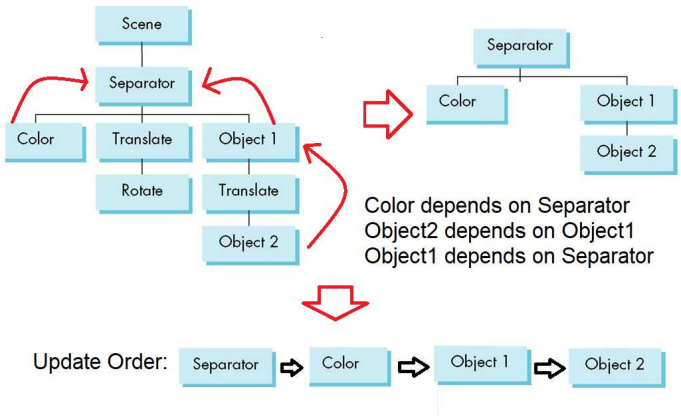

Fig. 2. Functionality of the framework on the game tree.

The first step in Algorithm 1 is to perform a Depth-First Search (DFS) from top to bottom on the game scene. The algorithm determines the dependencies for each component in each game object. A dependency is a relationship between two node. If component A attached to a node has an attribute that references component B, then component A depends on component B.

For every game object, when a new node is inserted, all the components attached to the node are inserted into the dependency graph. If component Y depends on component X, then a direct edge from Y to X is inserted into the graph. The is only done once when the game starts up.

For every new edge created, the algorithm ensures that no circular dependency exists. A circular dependency would result in an infinite loop in the Depth-First Search and, hence, in the update process.

Algorithm 1 Build a Dependency Graph

```
Q = empty queue;
Tree = hierarchy of game objects;
Root = root of the game tree;
Enqueue Root on Q;
while Q is empty do
    Obj = Dequeue the next object in Q;
    Enqueue in Q all the child objects of Obj;
    Populate list L with all the components of obj in Q;
    while There is an unchecked Component C in List L do
        if c is not in Unity3D Framework or Net Framework
        then
            Insert c as a Node in the graph;
            while There is an unchecked Field or Property P
            in the Component C do
                if P is a component of a programmer type and
                value of P is not null then
                    Insert P as a Node in the graph;
                    if edge between c and P do not cause a
                    cycle then
                        Create a edge in the graph between
                        node c and node P;
                    end
                end
            end
        end
    end
end
```

The result is a graph consisting of a list of nodes, where each node contains the component object, its game object information, and its type information. The edges are composed of a destination node and a source node.

The order in which the components are updated is defined by performing a DFS that will result in the definition of the component queue and its update process in every frame.

Once for every frame, the algorithm rechecks the graph using a DFS as in the start-up routine. However, this time it compares every component in the scene graph with the previous state stored in the dependency graph.

If a new object is in the scene, the algorithm adds a new node to the graph. It computes the new component's dependencies and, if some other component depends on the new component, the addition causes a recomputation of that component.

Algorithm 2 Update a Dependency Graph

```
Q = empty queue;
Tree = hierarchy of game objects;
Root = root of the game tree;
Enqueue Root on Q;
while Q is empty do
    C = Dequeue the next object in Q C1 = object that is equal
    to C in the Dependency Graph, null if none is found if C1
    is null then
        Insert C1 as a Node in the graph;
        while There is an unchecked Field or Property P in
        the Component C do
            if P is a component of a programmer type and
            value of P is not null then
                Insert P as a Node in the graph;
                if edge between c and P do not cause a cycle
                then
                    Create a edge in the graph between node
                    c and node P;
                end
            end
        end
    end
    else
        while There is an unchecked Field or Property P in
        the Component C1 do
            if P is a component of a programmer type and
            value of P is not null then
                P1 = object that is equal to C in the Depen-
                dency Graph, null if none is found if P1 is null
                then
                    Insert P as a Node in the graph;
                    if edge between C1 and P do not cause a
                    cycle then
                        Create a edge in the graph between
                        node c and node P;
                    end
                end
                else
                    if value of P1 null or is different from P
                    then
                        Update the value of P as a Node in
                        the graph;
                    end
                end
            end
        end
    end
end
```

In case of a change in the object in the scene (like items on the floor being picked by the player or items being released by the player), the algorithm updates the dependency graph and recomputes all dependencies, including adding the new dependencies.

For every node that changes, only the nodes depending on it are subsequently changed. After this step is done, a new update order is recomputed using a DFS on the dependency graph and then all the components are called in order.

If one component has several other components depending upon it, it is called as soon as possible after the updates of all components upon which it depends.

A component's updated value remains available to be used by all other components that depend on it. When a component is called to be updated, the algorithm determines whether all its dependencies are solved and then the component updates using the latest values.

The algorithm that creates and updates the graph requires that any script inserted must implement the interface `IUpdatable`. This interface specifies that any script that implements it must have a `FakeUpdate` function. This function is called instead of the default `Update` function. All functionality to be handled in a reactive manner must be included in this function.

Thus the Unity3D internal classes, the .Net framework classes, and other scripts that the developer does not want to treat reactively are not in the queue. They neither are triggered by changes in other scripts nor trigger changes in other scripts. This avoids unnecessary and time-consuming calculations. In the root of the game tree, we attach the dependency graph manager. It creates the dependency graph, analyzes changes, determines the best order for the updates to occur, and triggers the `FakeUpdate` functions.

The components that we want to monitor implement the interface `IUpdatable`. Using `IUpdatable`, the manager can track the changes in components. When a change occurs in any component, the manager can propagate the change to all that component's dependencies, which occur later in the queue. These components will then be updated subsequently in the update cycle.

Theoretically speaking, every frame represents an instant in an infinite time stream. For every instant, each reactive object has an observable state.

V. Testing Setup

For the development and testing of our framework, we chose the following software tools:

- *Unity3D version 2018.2.2*. We based our framework on the October 10, 2018 release of the Unity3D game engine.
- *Unity3D Standard Assets*. We selected 3D models from the Standard Assets package included with the Unity3D installation. The package includes a Prototyping folder that holds 3D models, materials, and prefabs to aid in prototyping 3D levels.
- *Visual Studio Community 2017*. We developed the software using the Microsoft Visual Studio Community 2017 edition. It is an Integrated Development Environment (IDE) that supports several different programming languages. We integrated the use the Visual Studio Tools for Unity3D with Unity3D's editor. This enabled us, for example, to receive debugging information from Visual Studio when working in the Unity3D editor.
- *C#*. We wrote the Unity3D "scripts" (i.e. programs) in C#, the primary programming language supported by Unity3D. C# is a multi-paradigm programming language with a syntax similar to C++, and Java. It is a common language used for game development in industry.

For purposes of comparison, the tests were also conducted on two other arrangements besides our framework:

- Unity3D alone
- Unity3D plus UniRx

UniRx is an initiative that seeks to implement reactive programming in game engines and graphics platforms. As described on its website [2]: "UniRx (Reactive Extensions for Unity3D) is a re-implementation of the .NET Reactive Extensions". Reactive extensions are libraries developed for several languages and technologies with the goal of implementing the principles laid out in the Reactive Manifesto. The Reactive Manifesto [31] is a document that states the theoretical background, benefits, and practical characteristics that any reactive system implementation should exhibit.

The UniRx library consists of reactive asynchronous and event-based programmable artifacts using observable collections and LINQ-style query operators. The library extends the Unity3D object set, representing events such as sensor data or game loops as reactive sequences. UniRx was motivated by the desire to resolve web connection issues. UniRx thus has a limited scope and is tightly integrated with the .Net framework.

We developed a set of automated tests using both graphical and non-graphical applications. The tests include operations to delete, modify, and add components and game objects throughout the execution.

For the first test scenario, we randomly build trees that represent mathematical expressions. The task of the computation is to determine the current value of the expression. A leaf of the tree represents a numerical constant. An internal node represents a binary operator that combines the values of its two children. The diagram shown in Figure 3 illustrates the way the game tree is constructed to emulate a calculation.

The result of node `A` depends on the availability of the values of its two child nodes `A1` and A2. `A1` also depends on the values of its two children `A11` and `A12`. So an order in which the updates are done correctly is `A11` $\rightarrow$ `A12` $\rightarrow$ `A1` $\rightarrow$ `A2` $\rightarrow$ `A`. Any flow that executes `A` before `A1` or `A2` would cause the calculation to be done with an outdated or nonexistent value, which would raise an error.

In our tests, after every update cycle we compare the value of each node with its expected value, which can be computed beforehand.

Throughout the execution of the tests, we simulate possible real-world changes to the game scene by inserting, deleting, or modifying nodes. After the next update cycle following a modification, the test compares the result of evaluating the expression with the expression's expected value. We repeat the test several times between modifications to ensure that, with no modifications, the result remains stable.

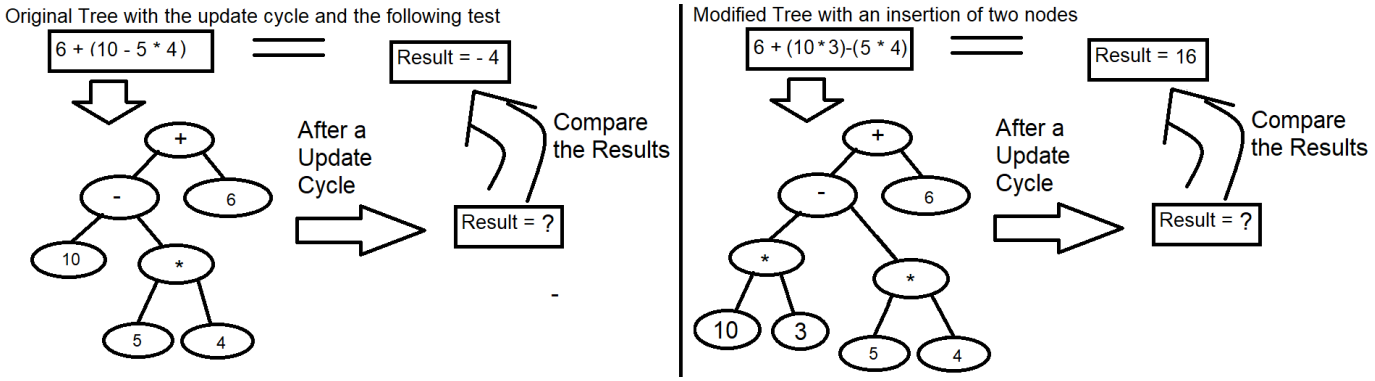

Fig. 3. Diagram of the test performed.

We also created another test scenario—a scene in which a player can walk around, collide with walls, look for targets, and shoot a target. We adapted this scenario from an example in the Standard Assets and included fully functional scripts.

We applied the same methodology to this test scenario that we used in the first scenario above. In this scenario, we change components and game objects throughout the execution. For example, at some point in the test, a wall that did not previously react to shots now receives the component that causes it to start reacting to shots, and vice versa.

VI. Results and Analysis

The test configuration used in this work is a three-way comparison between our framework on top of the Unity3D, Unity3D by itself, and UniRx on the top of the Unity3D.

Performance-wise, the results achieved do not show any improvement or perceptible increase in the time spent on the update cycle taking in consideration our framework against UniRx or against Unity3D. All of them performed with no considerable difference in the average time for an update. The use of our framework does not implicate a severe loss of performance.

On the results related to reactiveness, which is the capacity to create a stable experience that obtains the correct and predicted answer even on unpredictable situations, our framework performed much better than UniRx and Unity3D with default functionality.

In the UniRx implementation, we observed episodes of miscalculation in only 20% of the executions of the original scenario. However, the miscalculation episodes increased to 80% when changes were made during the execution. These episodes involved use of outdated or nonexistent values and always occurred after a modification of the game tree. During the period of instability, UniRx performed erratically, producing two possible situations:

- the system crashed with a null exception or a type-related exception. (The system was expecting a value of a certain type and found a value of another type or found nothing.)
- the system totally ignored the new/modified node.

Though UniRx was designed to reactively handle streams of input, it is not reactive to changes in objects themselves. This is a significant limitation when developing virtual environments or games where objects, users, or other components may arbitrarily enter or leave an environment. Consequently, UniRx would only be able to handle such events if they were known to occur ahead of execution.

In the default Unity3D implementations, we observed episodes of miscalculation in 90% of the cases in the original scenario. It required several updates (one update cycle for each dependency issue) until all the values were up-to-date.

The only situation where Unity3D carried out all the calculations correctly is when a test adds nodes to the tree in the precise order they need to be calculated. The Unity3D Event System organizes events in an arbitrary order. As a result, when a component updates there is no guarantee that the values it uses are up-to-date (as described in the section on game engines). It then takes several update cycles for a single chain of changes to spread fully through the game tree.

We also observed such behavior when changes were made to the game tree. Unity3D took several seconds to stabilize and start using the updated game tree. It inserted the new/modified node at the end of the update queue. When changes were inserted, we observed that the system produced miscalculations in 95% of the cases. The calculation only worked correctly when a change was purposely made to the root of the tree by modifying or replacing it.

In the implementation using our framework, we observed episodes of miscalculation in only 15% of the cases. These were mostly in the initial update cycles and update cycles occurring while a change was being made. Relative to the other implementations, our framework embraced the changes as they occurred, introduced them into the dependency graph, responded to the changes quickly, and then reached stability faster than the others.

Why was there such a difference between UniRx and our framework? UniRx was initially developed to handle network communication and asynchronous operations. It was not designed for environments that can change structurally from one frame to another. When a user sends a request to a server, UniRx expects the response to come back to the same game structure that sent the request. When this behavior is disrupted, UniRx considers it an error, or at least a misbehavior, of the

system. It either triggers an exception or computes an incorrect answer.

We developed our framework to take this issue into account. Our aim is to tackle this issue in particular and not deal with general asynchronous operations. Thus our framework does not currently satisfy all the expectations of the Reactive Manifesto. It should be enhanced and expanded to do so in the future.

VII. Conclusion

In a complex interaction among multiple objects, transitional turbulence often occurs. Multiple cycles may be necessary to update all the components and reach the desired final result. In many situations, this transitional turbulence causes a temporary inconsistency in games, simulations, and animation. Fortunately, these inconsistencies are often hidden from the users because of the difference between logical update cycles and rendering cycles.

This research sought to determine what effect it would have to reorder the updates according to the dependency relation among the components, that is, to carry out the update of a component only when all components that affect it have been fully updated. In particular, this research sought to improve the update process by using a reactive programming approach. We hypothesized that the increased control of the update cycle would improve responsiveness to changes in the game tree and ensure determinism of the result. To accomplish this, we developed a reactive framework built around the dependency relationships among the nodes of the game tree. We developed algorithms for constructing and maintaining the graph, flattening it to a priority queue, and using the priority queue to schedule the updates of components so that they completed in one cycle.

The tests we performed show that our framework is effective. The framework mitigates much of the transitional turbulence and produces a smoother experience when interactions occur among multiple objects. It fares well when compared with the existing Unity3D event system and UniRx, an existing reactive framework for Unity3D. Because our system is tailored to this problem, it effectively propagates changes through the entire game tree thus increasing the system's adaptability to changes among objects within the tree.

UniRx is still the general solution for dealing with asynchronous operations in Unity3D. It is still the best solution for general cases involving asynchronous calls. The solution proposed in this paper was more finely developed for the case claimed: to coordinate and enforce that operations occur in a given sequence, spread modifications out through the game tree completely by the end of the update cycle, and respond promptly and correctly to any structural modifications of the environment.

The same key concepts applied in this paper can also be applied to other applications, including web and mobile interfaces. An underlying reactive framework, such as the one described in this paper, can enable these applications to respond expeditiously to interactions, adapt quickly to an ever-changing environment, and align the internal mechanisms to construct a production line of components.

Acknowledgment

The first author's work was supported by CAPES, Coordination for Enhancement of Academic Level Individuals—Brazil.

References

[1] Unity - Manual: Unity Manual. [online] Docs.unity3d.com. Available at: https://docs.unity3d.com/2019.1/Documentation/Manual/ Last accessed in 8 February 2019.

[2] Y. Kawai. Reactive extensions for Unity3D. Available at: https://github.com/neuecc/UniRx. 2014. Last Accessed in 8 February 2019.

[3] E. Bertoluzzo. The essence of reactive programming: A theoretical approach : Dissertation, online; last accessed in November 23, 2018.

[4] J. Westberg. Unirx and unity 5: Working with c and object-oriented reactive programming : Dissertation, online; last accessed in November 23, 2018.

[5] M. Vasiv. Functional reactive programming for iOS with Objective-C and Reactive Cocoa: Bachelor's thesis, online; last accessed in November 23, 2018.

[6] A. Sherrod. 2007. Ultimate 3D game engine design & architecture. Charles River Media, Boston, USA.

[7] M. Lewis, and J. Jacobson. 2002. Game engines in scientific research. 45 (01 2002), 27–31.

[8] S. M. LaValle. Virtual Reality, Cambridge University Press, Cambridge, UK. 2017.

[9] B. Cowan, and B. Kapralos. 2014. A Survey of Frameworks and Game Engines for Serious Game Development. In 2014 IEEE 14th International Conference on Advanced Learning Technologies. 662–664. https://doi.org/10.1109/ICALT.2014.194

[10] J. Hocking. Unity in Action, Manning Publications Co., New York, USA. 2015.

[11] M. Smith, and C. Queiroz. Unity 5.x Cookbook, Packt Publishing, Birmingham, UK. 2015.

[12] A. Kirsanov, I. Kirilenko, and K. Melentyev. 2014. Robotics reactive programming with F#/Mono. In Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia (CEE-SECR '14). ACM, New York, NY, USA, , Article 16 , 5 pages. DOI=http://dx.doi.org/10.1145/2687233.2687249

[13] D. Soshnikov, and I. Kirilenko. 2014. Functional reactive programming: from natural user interface to natural robotics behavior. In Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia (CEE-SECR '14). ACM, New York, NY, USA, , Article 9 , 5 pages. DOI=http://dx.doi.org/10.1145/2687233.2687255

[14] C. Helbling, and S. Z. Guyer. 2016. Juniper: a functional reactive programming language for the Arduino. In Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design (FARM 2016). ACM, New York, NY, USA, 8-16. DOI: https://doi-org.umiss.idm.oclc.org/10.1145/2975980.2975982

[15] G. Baudart, M. Hirzel, L. Mandel, A. Shinnar, and J. Siméon. 2018. Reactive chatbot programming. In Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2018). ACM, New York, NY, USA, 21-30. DOI: https://doi-org.umiss.idm.oclc.org/10.1145/3281278.3281282

[16] E. Bregu, N. Casamassima, D. Cantoni, L. Mottola, and K. Whitehouse. 2016. Reactive Control of Autonomous Drones. In Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '16). ACM, New York, NY, USA, 207-219. DOI: https://doi-org.umiss.idm.oclc.org/10.1145/2906388.2906410

[17] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito. 2017. Vehicle Platooning Simulations with Functional Reactive Programming. In Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles (SCAV'17). ACM, New York, NY, USA, 43-47. DOI: https://doi-org.umiss.idm.oclc.org/10.1145/3055378.3055385

[18] N. R. Krishnaswami. 2012. Semantics for graphical user interfaces. In Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation (TLDI '12). ACM, New York, NY, USA, 51-52. DOI=http://dx.doi.org.umiss.idm.oclc.org/10.1145/2103786.2103794

[19] Y. Xie, H. Hofmann, X. Cheng, et al.: Reactive programming for interactive graphics.Statistical Science 29, 2 (2014), 201–213. 2

[20] G. Salvaneschi, S. Amann, S. Proksch, and M. Mezini. 2014. An empirical study on program comprehension with reactive programming. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014). ACM, New York, NY, USA, 564-575. DOI: https://doi-org.umiss.idm.oclc.org/10.1145/2635868.2635895

[21] S. Van den Vonder, F. Myter, J. De Koster, and W. De Meuter. 2017. Enriching the Internet By Acting and Reacting. In Companion to the first International Conference on the Art, Science and Engineering of Programming (Programming '17), Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, Article 24, 6 pages. DOI: https://doi-org.umiss.idm.oclc.org/10.1145/3079368.3079407

[22] B. Reynders, D. Devriese, and F. Piessens. 2017. Experience Report: Functional Reactive Programming and the DOM. In Companion to the first International Conference on the Art, Science and Engineering of Programming (Programming '17), Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, Article 23, 6 pages. DOI: https://doi-org.umiss.idm.oclc.org/10.1145/3079368.3079405

[23] J. Fisher, D. Harel, and T. A. Henzinger. 2011. Biology as reactivity. Commun. ACM 54, 10 (October 2011), 72-82. DOI: https://doi-org.umiss.idm.oclc.org/10.1145/2001269.2001289

[24] K. Toczé, M. Vasilevskaya, P. Sandahl, and S. Nadjm-Tehrani. 2016. Maintainability of functional reactive programs in a telecom server software. In Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16). ACM, New York, NY, USA, 2001-2003. DOI: https://doi-org.umiss.idm.oclc.org/10.1145/2851613.2851954

[25] M. C. Negrao. 2018. NNdef: livecoding digital musical instruments in SuperCollider using functional reactive programming. In Proceedings of the 6th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design (FARM 2018). ACM, New York, NY, USA, 1-8. DOI: https://doi-org.umiss.idm.oclc.org/10.1145/3242903.3242905

[26] A. Cleary, L. Vandenbergh, and J. Peterson. 2015. Reactive Game Engine Programming for STEM Outreach. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15). ACM, New York, NY, USA, 628-632. DOI: https://doi-org.umiss.idm.oclc.org/10.1145/2676723.2677312

[27] E. Czaplicki, and S. Chong. Asynchronous Functional Feactive Programming for GUIs. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation(NewYork, NY, USA, 2013), PLDI '13, ACM, pp. 411–422. URL:http://doi.acm.org/10.1145/2491956.2462161, doi:10.1145/2491956.2462161.

[28] S. Blackheath, and A. Jones. 2016. Functional Reactive Programming. Manning Publications Co., New York, USA.

[29] T. A. Furness, and W. Barfield. Virtual Environments and Advanced Interface Design, Oxford Publications, Oxford, UK. 1995.

[30] K. J. Blom, and S. Beckhaus. Supporting the creation of dynamic,interactive virtual environments. InProceedings of the 2007 ACM sym-posium on Virtual reality software and technology(2007), ACM, pp. 51–54.

[31] "ReactiveX - Introduction".[online] ReactiveX.io. Available at: http://reactivex.io/intro.html Last accessed in 12 February 2018.

[32] C. M. Elliott. 2009. Push-pull Functional Reactive Programming. In Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell(Haskell '09). ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/1596638.1596643

[33] G. Foust, J. Jarvi, and S. Parent. Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems, SIGPLAN Not., 51(3), 2015, 121–130,doi:10.1145/2936314.2814207.

[34] D. Kraeutmann, and P. Kindermann. Functional reactive programming and its application in functional game programming.

[35] I. Pembeci, H. Nilsson, and G. Hager. 2002. Functional reactive robotics: an exercise in principled integration of domain-specific languages. In Proceedings of the 4th ACM SIGPLAN International Conference on Principles and practice of Declarative Programming (PPDP '02). ACM, New York, NY, USA, 168-179. DOI=http://dx.doi.org.umiss.idm.oclc.org/10.1145/571157.571174

[36] K. Shibanai, and T. Watanabe. 2018. Distributed functional reactive programming on actor-based runtime. In Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2018). ACM, New York, NY, USA, 13-22. DOI: https://doi-org.umiss.idm.oclc.org/10.1145/3281366.3281370