Dependency Graph-based Reactivity for Virtual Environments

João Paulo Oliveira Marum* University of Mississippi
J. Adam Jones† University of Mississippi
H. Conrad Cunningham‡ University of Mississippi

ABSTRACT

In Virtual Environments (VEs), the system must quickly respond to user actions and accurately display the result. Current solutions on the Unity3D game engine often respond too slowly and display temporarily inaccurate or misleading states, resulting in low user satisfaction. To alleviate this problem, we develop a reactive programming approach that encodes the complex relationships among Unity3D game components in a dependency graph and then uses the graph to order the updates of the components without violating the dependency constraints. This enables more timely updates and more accurate visualizations, potentially providing users with a more satisfying experience. We evaluate our approach by comparing its performance with native Unity3D and with UniRx, the Reactive Extensions library for the Unity3D platform.

Index Terms: Software and its engineering—Software notations and tools—Development frameworks and environments—Object oriented frameworks; Human-centered computing—Human computer interaction (HCI)—Interaction paradigms—Virtual reality

1 INTRODUCTION

Virtual reality (VR) and augmented reality (AR) applications, collectively referred to as virtual environments (VEs), are *reactive* in nature. That is, they engage in ongoing interactions with their environments [17]. They respond to events, which may correspond to an interaction with the outside world (e.g. a user's movement) or with other components of the application (e.g. changes in the values of important data attributes).

In a VE, multiple objects interact with one another. When an event affects an object, there is often a "ripple effect" where all objects around the initial object are affected, and then all objects around those objects, and so on until the effects propagate through the entire system. It may take several update cycles for the states of all components to be updated and the system to reach a stable state. This is often called *transitional turbulence*. Transitional turbulence can result in inconsistencies in the visible state of the system. Sometimes an external observer cannot perceive this inconsistent state because the multiple update cycles are completed before a new frame is rendered. Sometimes an external observer can perceive this inconsistency when a new frame is rendered before stability is reached. In this case, the observer (at least temporarily) perceives the simulation as unreliable and inaccurate.

In this paper, we examine this problem by seeking to remove the instability corresponding to the transitional turbulence. We do this by reordering the execution of events so that an entire "ripple effect" can often be completed within one update cycle. We focus our attention on applications running on the Unity3D game engine, which is a popular platform for low-cost VE applications. During our tests on Unity3D, we found that Unity3D does not provide a way to control the order of execution. Thus, if two components `A` and `B` are executed in this order, if `B` affects `A` and `A` updates before `B`, then the modification made in `A` for `B` will not be made by `A` until the next update cycle. The ability to change the execution order of components and consequently enable in each execution the correct ordering of the components' executions in a scene graph is essential for high accuracy systems. In such systems, simulated interactions must occur in the same order as the interactions would in a corresponding real-world situation. If they do not, then the simulation fails to be realistic. An example would be a chain of dominoes falling, when the first falls, the second falls only when the weight of the first domino causes it to fall, and then the third falls similarly, until the last one falls. If one of the dominoes does not perform its fall correctly, the entire chain is compromised.

We are far from the first to propose creating reactive programming frameworks for virtual environments. There have been many solutions to this problem in the past, but most of these utilize specialized or purpose-built development environments [3, 9, 11, 14, 22, 27, 28]. We propose an alternative approach that can be applied to modern, widely used development environments, such as Unity3D. Furthermore, we propose designing this approach in such a way as to be intuitive for developers who are already familiar with these environments. The approach that we chose utilizes the development environment's native object hierarchy to implicitly define reactive relationships between objects in a virtual environment.

In this research, we apply a reactive programming approach that exploits the existing object hierarchy of modern game engines such as Unity3D. As described in Section 3, our approach analyzes the relationships among the components in the virtual scene and constructs a dependency graph. In this graph, there is an edge from component `A` to component `B` if the execution of `A` somehow affects the execution of `B` (e.g. `A` changes the value of a variable used by `B`). Our approach then sorts the graph topologically to determine an execution order for the components that satisfies the identified dependencies. This process is repeated whenever there is a change to the object hierarchy, so the system can respond dynamically to changes in the scene structure at runtime. Our approach does not use locks or otherwise modify Unity3D's event system. Instead, it defines large-grained events, each of which causes the execution of the reactive code of all the components involved in a single ripple of effects resulting from a user action. If `B` depends on `A`, then `A` executes before `B`. All the reactions in a chain can be completed in a single update cycle. This approach reduces the behavioral and computational errors without compromising the performance. It does so by using a non-locking, event-mimicking dependency-based approach that is easy to incorporate into the development of regular VE applications with the purpose to create a controlled event system inside the original Unity3D event system and reorder reactive version of Unity3D events using dependencies in order to achieve better accuracy in the final result.

In Section 4, we evaluate to what extent our approach improves the accuracy and predictability of simulations with mathematical and logical constraints. To do so, we develop an automated testing platform that includes game object hierarchies emulating the behavior of mathematical expressions. In Section 5, we compare our solution's performance to the results achieved by the same solutions built using the default Unity3D event system and also using UniRx [11], an existing, reactive library for Unity3D. Section 6 summarizes the results of this work. But first, in Section 2.1, we

*e-mail: jpmarum@ieee.org
†e-mail: jadamj@acm.org
‡e-mail: hcc@cs.olemiss.edu

explore the concepts of reactive programming and virtual environments.

2 Background

2.1 Reactive Programming

Manna [17] defines a reactive program as software that engages in an ongoing interaction with its environment. Operating systems and embedded systems are examples of reactive programs that do not terminate. Reactive programming (RP) focuses on how software reacts to changes in a system's state. These changes can be caused internally, by interactions between the components of the software, or externally, by an external actor such as the user or even another software system. The developer must determine the chain of events caused by a particular change. Throughout the execution of the application, the program reacts to the changes as defined by the developer.

A simple example of reactive behavior is the commonly used spreadsheet. The model underlying most spreadsheets stores the connections between the cells, so when one cell changes, the other cells that rely on the first cell's value to calculate their own results also update, thus propagating reactions throughout the sheet until all values are updated. This, of course, is not to say that all RP is as simple as a spreadsheet. For instance, RP has been applied to a wide range of theoretical and applied areas including robotics [8, 12], graphical user-interfaces [4, 7, 13, 25], autonomous drones and vehicles [1, 5], biological simulations [6], telecommunications [23], and even arts [19, 20].

Vasiv et al. [26] generalizes programming problems solvable by RP as problems where the interaction between the program and the environment can be described as continuous and unpredictable. In such situations, the user chooses how and when to interact with the program; the system can only react upon the events afterwards. An evident example, as shown in Van den Vonder et al. [25], is a highly interactive application with an assortment of ways that the user can interact with the software. That is, there are many possible events that can change data values. This is not dissimilar to how virtual and augmented reality systems function.

One common way to enable reactiveness is to capture the data dependencies between the program's components and execute the chain of functions necessary to propagate the changes based upon these dependencies [7, 15]. A similar prototype for Unity3D was described by Marum et al. [18]. We adopt and extend this approach by introducing a more robust and comprehensively tested system with automated tests and a random expression tree builder.

There are many examples of RP frameworks ranging from those designed to add reactivity to an existing and purpose-built languages. For instance, Czaplicki and Chong [4], and Blackheath and Jones [2] focus on the development process of reactive libraries (Elm and Sodium) from scratch. Blom and Beckhaus [3] examine a Haskell library for functional reactive game development. Their approach is based on a RP layer that acts like a middle-man between the reactive user interface and the non-reactive VE manager.

2.2 Virtual Environments and Game Engines

Though reactive programming for VEs is not a new concept, with prototypes dating back over a decade, these solutions are often purpose-built or have a limited scope of reactivity [3, 11, 27]. The solution proposed here is different from these in that its pattern can be applied to any game engine, or other development environments, where components are natively organized into a hierarchy. This enables the programmer to apply reactive behavior to a large array of inputs, streams, and game objects. Since this method is also based on the environment's native hierarchy, the structure of the reactive relationships can change at runtime without loss of performance. As VEs become more shareable between users, this feature becomes particularly important as users and their associated virtual objects may enter and leave shared environments arbitrarily, making having a set of pre-established relationships among objects less predictable.

A commonly used game engine is Unity3D. Seligmann [21] describes it as a video game engine for the development of two- and three-dimensional games, including support for virtual and augmented reality. Unity3D supports development for a wide variety of game platforms and provides a C#-based scripting system. Unity3D uses a hierarchical, component-oriented programming approach to organize these scripts into an application. Each script forms a component that can be attached to one or more objects. Each object can have several capabilities, giving programmers the ability to easily attach many functionalities to a particular game object.

Unity3D [24], like many other game engines, seeks to support cross-platform development and thus restricts itself to a generic set of features common to most game platforms. This has a number of consequences, but, for our purposes, it means that there is no defined order of execution for objects within a game's hierarchy.

The correct ordering of the internal interactions between components in a scene graph is essential for high accuracy systems. In such systems, simulated interactions must occur in the same order as the interactions would in a corresponding real-world situation. If they do not, then the simulation fails to be realistic. For example, in the real world, a human hand does not penetrate a solid object that it touches. In a simulated world, if the hand first penetrates a solid object and then is subsequently repelled, then the simulation is not realistic. But such a situation can result from an arbitrary order of execution of the simulated actions.

The only comment that the Unity3D manual [24] has on this issue is: "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."

The Unity3D manual does not describe in what order the `Update()` functions are executed in a given group of scripts. This is an important issue as each component may change the value of other components and affect the overall reactivity of an environment. We performed experiments to assess the execution order of the updates in Unity3D. Each experiment involved a system with several objects, each of which inserted its unique number into a list each time it updated. The list was static, so the application contains only one copy of the list accessed by all objects. The update test involves programming the update function of every object to add a number (a unique index for every object in the scene) into the global list. Since the order in which objects and components are added seems to be the order used to arrange the updates, we searched for what type of modification to the game tree causes a modification in the update order. From one test scenario to another, the only thing that changed was the order in which the game objects and their components were inserted or modified in the tree.

We performed the following sequence of tests:

1. Create the game objects level by level from the game tree, inserting them from the root toward the leaves.

2. Alter the creation order from the initial setup by exchanging the positions of objects from different levels of the tree.

3. Alter the creation order from the initial setup by exchanging the positions of objects within the same level of the tree.

4. Rearrange the creation order for the whole tree used as the initial setup but keep components assigned to the same game objects.

5. Detach some of the components from their original game objects in the initial setup and reattach them to other game objects in different levels of the tree.

In all of the tests, Unity3D consistently updated in the same fashion. This order is independent of the hierarchical position of the game object as the first test appeared to demonstrate, updating in an order starting from the leaves of the tree and finishing with the root. The updates are done by going from the newest objects inserted or modified to the oldest. Dragging objects around within the hierarchy has no effect on this order. The only test that produced a different execution order is the fifth test, where we remove a component from the root of the tree and insert it beneath one of the other components (one of the branch nodes). This change may occur as a result of a change to the project's generated metadata. Because of this change, both the root game object (where the component is taken from) and the branch game object (where the component is inserted) move to the beginning of the update order. Also, it is important to note that such modifications are done when the code is not running. Any modification during runtime (rearranging objects, attaching, reattaching or detaching components) has no effect on the metadata and thus has no effect on the Unity3D execution order.

Besides our approach, another possible way to handle this would be make changes directly to the metadata or somehow change the way Unity3D works. We did not consider these solutions because they involve deep knowledge of the internal mechanisms of Unity3D, and both of them may cause undesired, erratic behavior by the other game objects. Making changes directly to the metadata or restricting the execution order inside of the Unity3D would require a deep level of knowledge of its internal operations that appears to be beyond the scope of the publicly available documentation.

2.3 UniRx

Reactive Extension for Unity3D (UniRx) is a library for developing asynchronous and event-based programs using observable collections and LINQ-style query operators to implement reactive programming in Unity3D. Kawai describes UniRx as "a reimplementation of the .NET Reactive Extensions" [11].

Malawski [16] argues that UniRx alleviates the side-effects for asynchronous execution in Unity3D. UniRx represents any data sequence from Unity3D as an observable sequence. An application can subscribe to these observable sequences to receive asynchronous notifications as new data arrives. UniRx was motivated by the desire to improve web connection support for games and minimize errors and thus has a somewhat narrow scope. It relies strongly on the capabilities of the .Net framework. It is important to note that dependencies in UniRx must be explicitly specified by the programmer.

3 Implementation

This section describes the implementation of our reactive, dependency graph-based component for game engines. This operates by capturing the associations between components to establish their execution order. This implementation was based upon a proof-of-concept implementation of a dependency graph reactive component described by Marum et al. [18].

We dynamically reorder the execution of the object hierarchy to correspond to the current data dependencies among the components. We seek to schedule the execution of each component so that up-to-date values of all its data are available and the execution occurs soon thereafter. We create a dependency graph by analyzing the system, looking for a relationship between components that would fit the established criteria of dependency. An update order is then generated in such a way that it will meet the dependency graph's constraints. All future update cycles then start by the checking for any changes in the hierarchy. The update order is then recomputed accordingly. Finally, these components are called in the order their hierarchical dependencies.

Dependency-based reactivity uses a dependency graph to rearrange those components in a new structure where they are connected to each other. The system uses a non-locking approach that works around the Unity3D's default event system by calling the execution of the reactive event in each component in an order that satisfies the dependency graph's constraints. Our approach has the following steps:

- Analyze the scene graph to determine the underlying connections between the components.
- When the application starts, traverse the object hierarchy to build the dependency graph. Every object in the scene is copied to the dependency graph, and whenever a connection exists between two components, an edge is created between the same components in the dependency graph.
- After the entire dependency graph has been built, topologically sort the graph.
- On every update cycle, recheck the program structure to determine whether there are any changes in the dependencies between components. Update the dependency graph accordingly.
- Ensure that the order of execution satisfies the constraints imposed by the dependency graph. Execute them accordingly whenever a change occurs to a reactive item. This continues the chain of reactions while the changes are propagated from one node to others until the scene graphs achieves a stable state where no more changes are propagated.

Algorithm 1 Evaluating the scene and building the dependency graph.

```
Q = empty queue;
Tree = All the game objects;
Root = root of the game tree;
Enqueue the root in Q;
while Q is not empty do
    comp = Dequeue the next object in Q;
    Enqueue in Q each child object of comp;
    while for each Component C attached to game object comp do
        if C is not in Unity3D or .Net type then
            Insert C as a Node in the Graph;
            while for each Field or Property P in the Component C do
                if value of P is a component that implements IUpdatable and P is not null then
                    Insert P as a Node in the Graph;
                    if Edge between C and P does not exist and do not cause a cycle then
                        Create a Edge in the Graph between source C and destination P;
                    end
                end
            end
        end
    end
    Breadth-First Search to produce a valid update queue
end
```

The implementation encodes the data dependencies as a directed acyclic graph (DAG), where the nodes represent the component's state and edges between nodes represent direct dependency relations between components. The DAG is built by analyzing the game

engine's object hierarchy and extracting the dependencies as it is shown in Algorithm 1. Each node contains the component object copied from the scene graph, its parent game object information, and the component's type. Each connection is encoded as a directed edge. The edges are directed, which means they contain a destination node and a source node. The source is the component that contains the value used by the other component, and the destination is the component that depends upon the other—the component that uses the value from the source node.

We define dependency as a relation where a component `A` has the value of itself or the value one of its properties or fields fully or partially defined by the value of another component `B`, the value of one of `B`'s property or one of `B`'s fields. Also, a dependency exists if the update function of component `B` alters the value of component `A` itself, one of the `A`'s properties or one of the `A`'s fields. In these cases, the system records an edge going from `A` to `B`. We have established, as a condition for the system to work, that any component or value to be changed by an event of the component analyzed must be explicitly defined as a property or field of this component. This condition can be relaxed in the future by adding code to verify which components and values are being modified by an event.

For each new edge, the algorithm makes sure that it does not create a circular dependency. A circular dependency is a situation where starting from a particular node the search can reach the same node by following some chain of dependencies. This would result in an undesirable infinite loop. In the case where a cycle is detected, the dependency is omitted by the system but the execution will still happen as non-reactive.The components' update order is defined using a topological sorting through a Breadth-First Search (BFS). Figure 1 shows the process of building the dependency graph and the update order from the game tree.

If a component is dependent on several other components, it will be called as soon as all of the other components are updated. A component's updated value will be available to any component that may need it. When it is the turn for a component to be updated, the framework checks all its dependencies that must be already updated so the component updates using the latest values. Another important characteristic is that the current state of the system is self-contained, which means there are neither dependencies across states nor across frames. That is a crucial characteristic of the system since accuracy and predictability on each update cycle is the core issue of this work.

The reordering of the execution of each component is done using a non-locking approach that mimics most of the properties of Unity3D's original event system, but works only with the reactive event present on the reactive components created in the application. So all the execution of the reactive components works inside the event of a single component, an atomic execution inside the original event system. This way we define our system locking the execution order without causing disturbance on the original execution of the other components.

Once for every update cycle, the algorithm reanalyzes the graph using a DFS. This process is described in Algorithm 2. During this cycle, the framework determines whether any component in the scene graph has been modified, deleted, or inserted relative to the current state of the dependency graph. The algorithm reacts differently in three distinct cases:

New component added to the scene: The algorithm adds the new component to the graph and determines which components that it depends upon. The algorithm then recomputes the dependencies for all components that became dependent upon the new component.

A component modified in the scene: If some of the component's properties are changed, the algorithm must update the dependency graph around that component appropriately. The modified component may now be dependent upon different components and different components may now be dependent upon the modified component.

A component deleted from the scene: All the edges coming from or going to the deleted component must be deleted from the dependency graph. The dependencies for all components that depended upon the deleted component must be recomputed.

Algorithm 2 Reanalyze the scene graph in order to perform any needed updates to the dependency graph.

```
Q = empty queue;
Tree = All the game objects;
Root = root of the game tree;
Enqueue Root on Q;
while Q is not empty do
    Comp = Dequeue the first object in the queue;
    C1 = same instance of Comp previously stored in the Graph,
    null if none is found;
    if C1 is null then
        Insert Comp as a Node in the Dependency Graph;
        while for each unchecked Field or Property P in Comp do
            if value of P is a component that implements IUpdatable and P is not null then
                Insert P as a Node in the Dependency Graph;
                if Edge between C1 and P do not cause a cycle then
                    Create a Edge in the Dependency Graph
                    between Node Comp and Node P;
                end
            end
        end
    end
    else
        Update the value of C1 in the Graph;
        while for each unchecked Field or Property P in C1 do
            if value of P is a component that implements IUpdatable and P is not null then
                P1 = object that is equal to P in the Dependency
                Graph, null if none is found;
                if P1 is null then
                    Insert P as a Node in the Dependency Graph;
                    if Edge between C1 and P does not exist or do
                    not cause a cycle then
                        Create a Edge in the Dependency Graph
                        between Node C1 and Node P;
                    end
                end
                else
                    if value of P is null or different from P1 then
                        Update the value of P in the Graph;
                    end
                end
            end
        end
    end
    Breadth-First Search to produce a valid Update Queue;
    while for each Component c in the Update Queue do
        execute Update Function;
    end
end
```

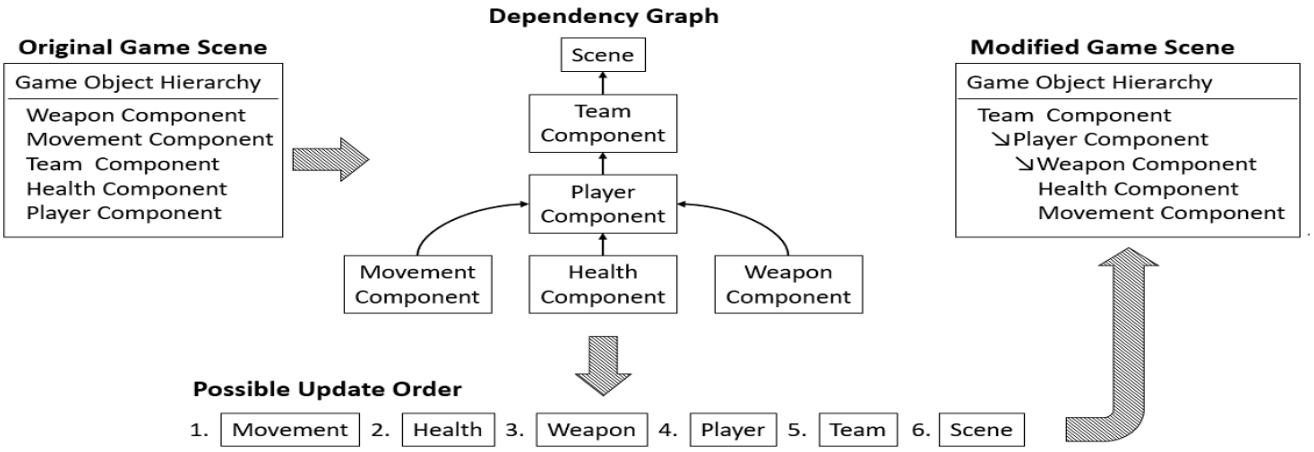

Figure 1: Building the dependency graph and the execution queue from the scene graph.

One of our goals is to be able to operate with already established VE systems and technologies such as Unity3D. Thus, to preserve the proper functioning of the original update mechanism of those systems, our framework requires that any reactive script in the scene implement the interface `IUpdatable`. This interface specifies a single function `ReactiveUpdate`. This function is called in a reactive manner instead of the default `Update` from the Unity3D event system. All code in the script that will be handled reactively must be executed in this function.

This is done so that any behavior that must be dealt without reactivity can update in the default way without sacrificing performance or dealing with the internal mechanisms of the Unity3D framework. From the same standpoint, Unity3D internal classes, the .Net prototype classes, and other scripts that will not be reactive are ignored by our framework. They are not triggered by changes and do not trigger changes in other scripts.

In the root of the game scene tree, the developer must attach the dependency graph manager component. This is the component that does the functions described above. All the reactive components implement `IUpdatable`, allowing the manager to detect changes within these components. When a change occurs in any reactive component, the manager propagates the change through the entire dependency chain. The reactive components must be updated as a single block during the subsequent update cycles.

One of the biggest issues encountered in the development phase was that the Unity3D system was unable to determine equality between references to the same object in different data structures. In our approach, comparison of multiple instances of the same object is a key feature needed to trace changes in objects and spread them throughout the dependency graph. This is also important when checking if new objects were added or deleted and if new dependencies were added, changed, or deleted.

When comparing using `object.equals()`, there were no positive answers, even though the comparisons were made between exactly the same objects. The same was observed when comparing tags and references (using `ReferenceEquals()`). The use of `Find()`, `FindByTag()`, and `FindByName()` requires removing many false positives and leads back to the problem of finding a positive using one of the techniques mentioned above. As such, we defined an indirect definition for equality comparisons between objects that focuses on the characteristics of the objects. If two components have the same type and name and they belong to game objects that are equal (which means they also have the same properties), then they must be the same. This approach relies on the fact that no two objects of the same component type and with the same properties (name, tag, and position) can be attached to the same game object. This set of information represents, for our purposes, a unique identification for each component. These are required to allow us to define equality of objects effectively.

4 Test Methodology

For purposes of comparison, we built and tested similar environments using two other system combinations in addition to our framework:

- Unity3D (using its own event system)
- Unity3D with UniRx (Unity3D Reactive Extension)

The algorithm described in the previous section builds a dependency graph and traverses it completely in an amount of time very close to the time performed by Unity3D alone between each frame. We measure the time spent for all the systems tested in all scenarios using the `StopWatch` class from the .Net framework. This class emulates the behavior of a regular stopwatch, giving us the ability to start and stop it as needed. We start it at the beginning of the `start()` and `Update()` and stop it at their end. The property `Elapsed` from the class `StopWatch` gives the amount of time in milliseconds spent from start to stop. Our tests do not reveal any significant performance degradation either in the update time or the start-up time. That happens because only the objects that can trigger reactivity are considered and only when their values change. On average, the time consumed for the system on the updates remains relatively small in comparison with Unity3D alone. In the future, we plan to develop tests to measure more rigorously the above time increase utilizing the test methodology described in Jones et al. [10].

To test the effectiveness of our approach, we built a test scenario that demonstrates how the update order affects common interactions among multiple objects by using the example of an expression tree calculator. This test case passes through a set of automated tests using UniRx, Unity3D, and our reactive framework.

It is generally quite difficult to determine whether a series of objects updates in the expected way in a virtual environment without degrading the system performance by storing state data in memory or writing to a file. Instead, our approach was to design a game environment where objects within the scene behaved as computation components within an expression tree. This allowed us to build test cases where the value computed by the expression tree is known a priori. Any error in execution order would then be detectable in the final state of the system simply by comparing the tree's computed

values with the expected values. This served as a very sensitive method for detecting errors in execution without introducing additional overhead costs. The test scenario included insertion, deletion, and modification of components. Performing the test involved:

- Randomly generating binary trees that represent mathematical expressions.
- Randomly placing integer values in the leaf nodes and binary operators in the internal nodes of each tree. The "current" value of an internal node can be computed by performing its operation on the "current" values of its two children.
- Computing the "current" value of the tree by computing the "current" value of the root. For the "current" value of the tree to be the correct value of the expression, the values of all nodes must be computed in the correct order. That is, the value of both sub-trees of an internal node must be computed before the value of internal node.

The algorithm randomly generates a component that represents an expression tree. The tree is either a leaf or an internal node. If the tree is a leaf, then the algorithm randomly selects some integer value. If the tree is an internal node, then the algorithm randomly selects an arithmetic operator chosen from addition, subtraction, multiplication, division, and exponentiation. To keep the expressions relatively simple, we limit the number of operators that can be selected to five. We also require that the root node be an operator to eliminate trivial cases. This algorithm generates mathematical expressions such as the following:

- $5+(4*9)+3-5^2$
- $32*(7+9)$–12–16
- 50–$(12+16)/8$–$(12-9)$

The diagram in Figure 2 depicts the test process. The tree is first embedded in a game hierarchy, then the "current" value of the tree is calculated, then this value is compared to the expected (i.e., correct) value of the expression. To determine the adaptability, the test is repeated with several variations of the original tree.

The result of `A` is a calculation between its child nodes `A1` and `A2`, which means that in order to obtain the result of `A` correctly, both `A1` and `A2` must be available and correct. `A1` similarly depends on its two children `A11` and `A12`. So the calculation will obtain the correct final result only if the updates that trigger the calculations respect the following order: `A11` $\rightarrow$ `A12` $\rightarrow$ `A1` $\rightarrow$ `A2` $\rightarrow$ `A`. Any execution order in which `A` executes before its children (`A1` and `A2`) would produce a wrong result since it used an outdated or nonexistent value.

In our tests, after the end of the update cycle, the value of each internal node is compared with the expected value computed beforehand. At some random time during the tests, we introduce changes to the scene graph by inserting new nodes in random locations, deleting random nodes, or modifying the value of a node. In the next update cycle, the test compares the result from the expression with the new expected value. We also keep testing between modifications to ensure that the result remains stable.

The test configuration that we use in this work is a three-way comparison between projects built using Unity3D with our framework, the default Unity3D event system, and UniRx. The computer that we used for testing was a laptop Dell Latitude E5550 with the processor Intel Core i5 – 5300 2.3 GHz, 8 Gb RAM, Intel HD Graphics 5500, running a Windows 10 64-bit operating system, Visual Studio 2017, and Unity3D 2019 3.0.

5 Results and Interpretation

Performance-wise, our framework showed that the time spent in the update cycle is, on average, similar to the performance achieved by UniRx or Unity3D alone. Thus, the use of our framework did not indicate a significant increase in the time spent in the update cycle. Creating the dependency graph and performing the topological sort initially took, on average, 198 ms. When the dependency graph needed to be reconstructed, the update function took up to 100 ms to redo the analysis and sort.

To measure accuracy of each testing platform, we determined whether it reached a correct (i.e. accurate) state at the end of each update cycle, despite having to handle unpredictable situations. In this test, our framework performed better than UniRx and Unity3D with default functionality.

We chose to record three metrics to compare performance between systems. We recorded the measurements and took the average based upon the total number of update cycles, the average results were recorded in Tables 1 and 2 for each platform in each scenario:

- The latency (number of update cycles) needed to get the game into the expected state once set into action.
- The number of errors detected for a sequence of interactions expressed as the average number of update cycles that contained at least one error.
- The average number of errors in a single test where errors were detected. This is how many components' updates were incorrectly ordered in a test where there were errors detected. We count the total number of errors, the total number of runs that reported an error, and finally compute the average.

Table 1 shows the result of the first test, with the expression tree running for 100 user cycles and no modifications inserted.

Platform	Total Cycles	Total Errors	Avg Errors per Cycle	Visible Errors	Latency in Cycles
Unity3D	100	95	5	0	5
UniRX	100	15	2	0	5
Our Tool	100	15	2	0	1

Table 1: Test Results for Scenario #1

Table 2 shows the result of the second test, with the expression tree running for 100 user cycles (which means cycles initiated by a direct user interaction) and modification inserted in the expression tree (game objects modified, added, or deleted).

Platform	Total Cycles	Total Errors	Avg Errors per Cycle	Visible Errors	Latency in Cycles
Unity3D	100	100	5	0	7
UniRX	100	80	5	0	20
Our Tool	100	20	3	0	1

Table 2: Test Results for Scenario #2

When referring to errors, we specifically mean a situation where one of the game components is executed before one or more of its dependencies, thus causing inadvertent use of out-of-date or missing values. This incorrect order causes a failure of the component or a temporarily erroneous state of the component. Latency means the number of cycles (or the time taken) between the beginning of the test and when the expected system state has been reached. Observable inaccuracy is the temporarily incorrect state of one or more

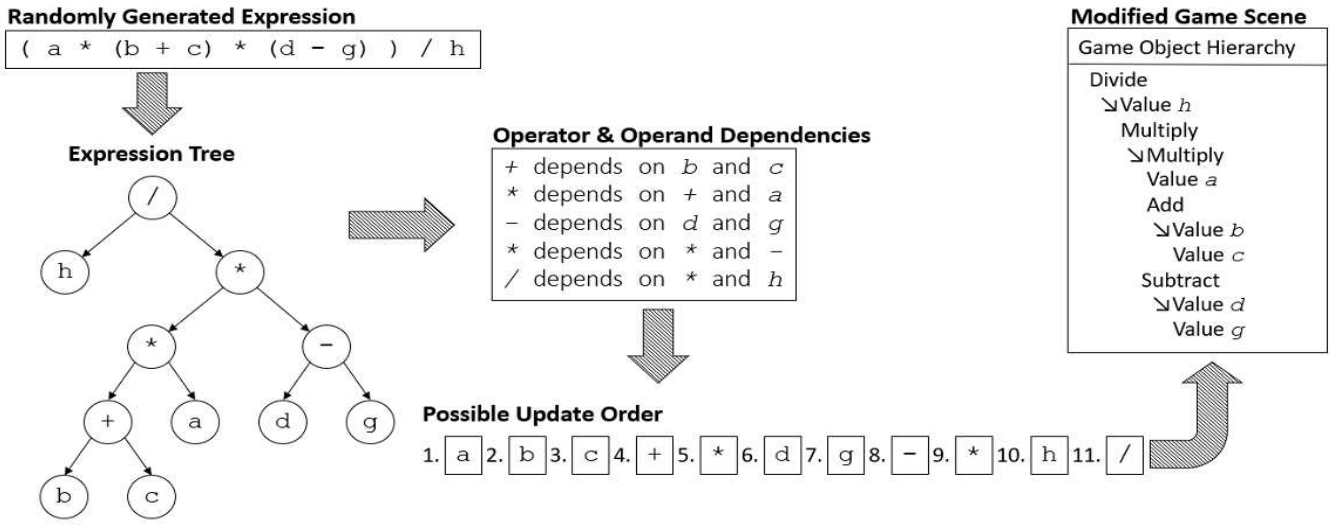

Figure 2: Mathematical Expression tree generation used for testing our framework.

of the components that is visible in the rendered imagery of the game. Since many update cycles can execute per frame of rendered video, we considered an inaccuracy to be observable if it persists for enough cycles to out last a single rendered frame. This does not mean that an observer will necessarily be able to see the inaccuracy in question, but instead that if some visible element of the environment relied on this component, the resulting error could potentially be visible to the observer. As such, this can be thought of as a lower bound or minimal criterion for a visible error to occur in the environment.

As can be seen in Table 1 for the UniRx implementation, we observed inaccuracies in only 15% of the tests using the scenario with no changes. However, Tables 2 show the inaccuracy rate increased to 80% when we introduced changes randomly throughout the test. These episodes of inaccuracy continue to occur for several cycles after a modification of the game tree. During the unstable period, UniRx produced several errors, catalogued in two possible categories:

- The system entered in an error state with a null or a type-related exception. (The system expected an object of a certain type and found an object of another type or found a null pointer.)
- The system ignored the existence of the new or modified node.

In the tests where modifications were introduced at runtime, UniRx took up to 30 cycles to recover from the error and reach a stable state. Additionally, in the tests where there was an error, the update cycle had a high number of incorrectly ordered executions per update cycle. This happens because any interaction with one of the modified/inserted/deleted components was not correctly handled, resulting in an erroneous state.

Though UniRx is designed to handle input streams in a reactive way, it does not react properly to changes in the objects themselves. This significantly limits the dynamics of virtual environments where components may be arbitrarily inserted or removed, such as multi-user experiences. Consequently, UniRx can only handle such situations if they are predictable and properly handled by the programmer.

In the default Unity3D event system, Table 1 demonstrates that episodes of error happened in 95% of the cases in the scenario with no changes to the scene graph. Several update cycles were necessary before the scene graph was up-to-date. This can be explained by the fact that the Unity3D event system uses an arbitrary order for updates [24]. When a component executes, it uses the available values, without knowing if the dependencies were updated accordingly. That is why a single chain of reactions takes time to spread through the scene graph. The number of cycles needed to reach a desirable result in Unity3D is invariably connected to the complexity of the simulations and how many components are involved in that cycle. This behavior can also be observed in the number of errors per update cycle.

The default Unity3D event system behaves similarly when changes are introduced to the game hierarchy. It took several cycles to stabilize the object hierarchy as shown in Table 2. The system also produces errors in 95% of the cases.

For our framework, Table 1 shows that episodes of error occur in only 15% of the cases. Table 2 shows that these results differed very little for situations where changes were introduced to the system. Compared to the other systems, our framework detected the changes and reached a stable state more quickly than the others.

One potential reason for the poor performance of UniRx is that it assumes that the structure of the scene graph will remain unchanged from one frame to the next. When the system makes an asynchronous operation, UniRx expects to find the same game structure that was there when the request was sent. When the structure changes, UniRx considers it to be a situation that it cannot handle and triggers a runtime exception.

6 Conclusion

In this research we have addressed the instability resulting from the transitional turbulence that occurs in virtual environments. We have demonstrated this by exploiting Unity3D's existing object hierarchy. Initially, and whenever the hierarchy changes thereafter, our approach extracts the inter-component dependencies and generates a new event-handling order that satisfies these constraints. Our approach groups a chain of reactions corresponding to some external action into a large-grained reactive event that can be performed in one update cycle, that is, within the execution of a single Unity3D event. This approach seems to have the benefits of locking, non-locking, and wait-free-based approaches without dealing with concurrency issues.

Transitional turbulence can lead to inconsistent and misleading

states within the VE, making the system seem unreliable and unpredictable. By reordering the events based on the dependencies, our approach removes many such inconsistencies without degrading the performance of the system. By dynamically reacting to changes in the object hierarchy, the approach can smoothly handle relatively complex applications. Our tests show that our approach performs better than both an unmodified Unity3D application and a similar application developed using the reactive library UniRx.

A goal of our design is to avoid conflicts with other third-party libraries and assets. Our tests using the Unity Standard Assets indicate that this is possible in principle. We expect the approach to be similarly compatible with a variety of applications, including physics engines (e.g. PhysX and Havok) and libraries for adding other simulation functionality. We also expect that the approach can be readily adapted to Unity3D's new Entity Component System (ECS) with results similar to those reported in Section 5. ECS focuses on how Unity organizes the data. Our approach, instead, focuses on how Unity organizes event execution. So the system behavior should be similar. To gain a better understanding of the approach's capabilities and limitations, we plan to implement the framework on a wider variety of platforms in the future.

Our approach improves responsiveness and performance to changes and results in more accurate mathematical and visual behavior. We hypothesize that the same general solution can be implemented for a variety of programs with built-in object hierarchical structures. For those problems, it should be easy to integrate our approach into the analysis of the structure and understanding of the dependency relationships between controls.

REFERENCES

[1] G. Baudart, M. Hirzel, L. Mandel, A. Shinnar, and J. Siméon. Reactive chatbot programming. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, REBLS 2018, pp. 21–30. ACM, 2018.

[2] S. Blackheath and A. Jones. *Functional Reactive Programming*. Manning, New York, 2016.

[3] K. J. Blom and S. Beckhaus. On the creation of dynamic, interactive virtual environments. In *Proceedings of the IEEE VR 2008 Workshop on Software Engineering and Architectures for Interactive Systems (SEARIS)*, 2008.

[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*, PLDI '13, pp. 411–422. ACM, 2013.

[5] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito. Vehicle platooning simulations with functional reactive programming. In *Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles*, SCAV'17, pp. 43–47. ACM, 2017.

[6] J. Fisher, D. Harel, and T. A. Henzinger. Biology as reactivity. *Commun. ACM*, 54(10):72–82, October 2011.

[7] G. Foust, J. Järvi, and S. Parent. Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems. *SIGPLAN Notices*, 51(3):121–130, October 2015.

[8] C. Helbling and S. Z. Guyer. 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, pp. 8–16. ACM, 2016.

[9] L. Jankovic. Games development in VRML. *Virtual Reality*, 5(4):195–203, December 2000.

[10] J. A. Jones, E. Luckett, T. Key, and N. Newsome. Latency measurement in head-mounted virtual environments. In *Proceedings of IEEE SouthEastCon 2019*. IEEE, April 2019.

[11] Y. Kawai. UniRx: Reactive extensions for Unity, 2014. https://github.com/neuecc/UniRx, last accessed February 9, 2020.

[12] A. Kirsanov, I. Kirilenko, and K. Melentyev. Robotics reactive programming with F# and Mono. In *Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia*, CEE-SECR '14, pp. 16:1–16:5. ACM, New York, 2014.

[13] N. R. Krishnaswami. Semantics for graphical user interfaces. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pp. 51–52. ACM, 2012.

[14] P. Lange, R. Weller, and G. Zachmann. Wait-free hash maps in the entity-component-system pattern for realtime interactive systems. In *2016 IEEE 9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pp. 1–8. IEEE, March 2016.

[15] S. Lehmann, T. Felgentreff, J. Lincke, P. Rein, and R. Hirschfeld. Reactive object queries: Consistent views in object-oriented languages. In *Companion Proceedings of the 15th International Conference on Modularity*, pp. 23–28. ACM, 2016.

[16] K. Malawski. *Why Reactive?* O'Reilly Media, 2016.

[17] Z. Manna and A. Pneulli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[18] J. P. O. Marum, J. A. Jones, and H. C. Cunningham. Towards a reactive game engine. In *Proceedings of the 50th IEEE SouthEastCon*, April 2019.

[19] T. E. Murphy. A livecoding semantics for functional reactive programming. In *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design*, FARM 2016, pp. 48–53. ACM, 2016.

[20] M. C. Negrão. 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, pp. 1–8. ACM, 2018.

[21] R. L. Seligmann. Creating a mobile VR interactive tour guide. Bachelor's thesis, Haaga-Helia University of Applied Sciences, Finland, 2018. https://www.theseus.fi/handle/10024/144759, last accessed February 9, 2020.

[22] L. Stefan, S. Hermon, and M. Faka. Prototyping 3D virtual learning environments with X3D-based content and visualization tools. *BRAIN. Broad Research in Artificial Intelligence and Neuroscience*, 2018.

[23] K. Toczé, M. Vasilevskaya, P. Sandahl, and S. Nadjm-Tehrani. Maintainability of functional reactive programs in a telecom server software. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pp. 2001–2003. ACM, 2016.

[24] Unity Technologies. Unity user manual. https://docs.unity3d.com/2019.1/Documentation/Manual/, last accessed February 9, 2020.

[25] S. Van den Vonder, F. Myter, J. De Koster, and W. De Meuter. Enriching the internet by acting and reacting. In *Companion to the First International Conference on the Art, Science and Engineering of Programming*, Programming '17, pp. 24:1–24:6. ACM, 2017.

[26] M. Vasiv. Functional reactive programming for iOS–with Objective-C and ReactiveCocoa. Bachelor's thesis, Turku University of Applied Sciences, 2018. https://www.theseus.fi/handle/10024/140834, last accessed February 0, 2020.

[27] J. Westberg. UniRx and Unity3D 5: Working with C# and object-oriented reactive programming. Bachelor's Thesis, Uppsala University, 2017. http://uu.diva-portal.org/smash/get/diva2:1107247/FULLTEXT01.pdf, last accessed November 23, 2018.

[28] D. Wiebusch and M. E. Latoschik. A uniform semantic-based access model for realtime interactive systems. In *2014 IEEE 7th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pp. 51–58. IEEE, March 2014.