UniRx and Unity 5
Working with C# and object-oriented reactive programming

Johannes Westberg

Faculty of Arts
Department of Game Design

Bachelor’s Thesis in Game Design, 15 Credits
Programme: Game Design and Programming

Supervisors: Mikael Fridenfalk, Hans Svensson
Examiner: Masaki Hayashi

June 2017
Abstract

Gameplay programming is vital for video game development and benefits from good tools and techniques. However, techniques are still used in the industry that involves describing how the computer operates. Reactive programming is a way to describe eventful and stateful computer programs declaratively, focusing on what the program should accomplish. This thesis uses the reactive library UniRx with the game engine Unity 5 to create an FPS with reactive techniques, and discusses the advantages and disadvantages of these. Object-oriented reactive programming is used to combine reactive objects with Unity’s component-based framework without using wrappers for non-reactive objects. The results include static methods for observable user input, patterns when defining game components with reactive content, and communication between game objects through interface components. It can be concluded that reactive programming for game programming enables the programmer to describe and understand game logic in a declarative manner. However, combining UniRx and Unity 5 led to complex patterns. Suggestion for future work is using reactive programming with game engines that allows to design game objects that are fully reactive.

**Keywords:** reactive programming, reactive-functional programming, game programming, gameplay programming, object-oriented reactive programming, programming patterns, function composition, UniRx, Unity 5.
Abstrakt

# Table of contents

1 Introduction.................................................................................................................................................. 1  
2 Purpose and limitations.................................................................................................................................. 2  
3 Background.................................................................................................................................................. 3  
   3.1 Reactive programming.......................................................................................................................... 4  
   3.2 Reactive programming with UniRx ......................................................................................................... 5  
   3.3 Previous work ......................................................................................................................................... 7  
4 Materials and methods .................................................................................................................................. 9  
   4.1 Materials ................................................................................................................................................ 9  
   4.2 Methods.................................................................................................................................................. 10  
      4.2.1 Observable user input ....................................................................................................................... 10  
      4.2.2 Object-oriented reactive programming ............................................................................................. 10  
      4.2.3 Component granularity ..................................................................................................................... 12  
      4.2.4 Shared and non-shared state ............................................................................................................. 12  
      4.2.5 Mutation on Unity’s non-reactive objects ......................................................................................... 13  
      4.2.6 Preventing memory leakage ............................................................................................................ 13  
5 Results .......................................................................................................................................................... 14  
   5.1 Observable user input ............................................................................................................................ 14  
      5.1.1 Keyboard......................................................................................................................................... 14  
      5.1.2 Mouse............................................................................................................................................. 15  
      5.1.3 Joystick ........................................................................................................................................... 16  
   5.2 First-person controller ............................................................................................................................. 17  
   5.3 Player character components ................................................................................................................. 20  
6 Discussion .................................................................................................................................................... 23  
7 Conclusion ...................................................................................................................................................... 25  

References  
Glossary of terms  
Appendix
1 Introduction

Gameplay programming is necessary for successful and fun video games. Without it, there would be no game. The gameplay is fundamental for every game that exist in this world. It involves feedback for players, interesting and engaging game rules, and overall polish. The gameplay programmer plays an indispensable role in realizing these experiences. In theory, the gameplay programmer focuses just on the logic in games, and not on how the game engine works. This is unfortunately not true in the game industry today. Techniques are still used that involve more than logic, thus requiring the gameplay programmer to deal with more than just logical problems and solutions. The Achilles’ heel lies in control: how the computer operates.

When programming was new, we had to think about programs in terms of zeros and ones. This has evolved since people created techniques for translating human letters for the computer to interpret. Letters have evolved into full words, and now we have programming languages that enable us to create and understand programs in a way we would not have if we had continued with binary coding. Among the most popular languages, like Java, C, and Python, we still need to puzzle together the order of operations in which the computer executes. It gives great control for advanced programmers, but in many cases, it is arguably redundant, particularly in gameplay programming.

Reactive programming is a way to model and understand the logic in programs. It involves composing relationships between objects and always keeps the system up-to-date when any changes occur. Programs become compositions, where the programmer becomes the composer of data and state. By letting the tools for reactive programming take care of how the program operates, the programmer can focus only on what the program is. This liberates the programmer from a lot of bug prone complexity. In the case for gameplay programming, this would be a very powerful tool.

Unity 5 is one of the most popular game engines today. Both amateurs and professional studios use the engine for creating high-end video games. Unity provides C# as a high-level scripting language, but it is heavily dependent on how the programmer describes the sequential order of executions. This thesis is an attempt to create gameplay in Unity by describing the game declaratively with the reactive library UniRx.
2 Purpose and limitations

This thesis presents an example of how reactive programming can be utilized in Unity 5 with UniRx. A local multiplayer first-person shooter is implemented to demonstrate how reactive methods can be used for one of the most popular game genres today. The purpose is not to provide for a comprehensive guide of how UniRx can be used with Unity 5. The purpose is neither to give an exclusive direction on how the tool should be used by realization of the wide field of applications that the UniRx offers. The intention of this thesis is to strive towards efficiency, ability of expression, and flexibility in gameplay programming. Among the questions to be answered, the following are included:

- Are there any advantages or disadvantages of using UniRx with Unity 5?
- Are any of the advantages/disadvantages caused by using reactive programming for game development or are they caused by using reactive programming in Unity 5’s framework? And if so, why?

In this work, the data classes in Unity’s API are left intact and are not adapted to be reactive. This could be performed by the creation of data types that wrap data types that are adapted. This requires extra effort. It also results in a library of new data types that need to be understood. These topics are avoided to save time and to maintain a model that is as easy to understand as possible when reading the methods and results in this thesis. This thesis pays extra attention to readers that are already familiar with Unity 5 and its API. Glossary of terms is included at the end of this document.
3 Background

Gary Dahl (2016) argues in his article about reactive game architectures that most advances in game development tools, game engine architectures, and game programming paradigms have been in favour of describing what happens in games rather than describing when these things happen. These advancements have not happened just in game programming but in programming in general. Still, there are many programming languages like C and Java that let the programmer describe how a program operates, which is close to the nature of computers in how they handle tasks in sequences. These languages are considered as imperative and have advantages by offering a low-level control to the programmer.

Meanwhile, there are languages and tools that provide a high level of abstraction by automating the control, leaving the programmer to only describe the logic in a program. This in contrast to imperative programming is considered as declarative programming. This term focuses on what the program should accomplish without specifying how it should be accomplished (Torgersson 1996). Kowalski (1979) describes algorithms as algorithm = logic + control, and argues that computer programs would be more correct, improvable, and modifiable if control and logic were identified and separated in code. The programmer can write code that expresses the intent, not the mechanism. If Dahl is right about advances in game development tools, then this concept should be noticed as a possible advancement in game programming.

Unity 5 is a cross-platform game engine that is built with a component based object-oriented programming framework. The engine is originally implemented in C++, but provides C# for users as a layer above the underlying implementation. C# is an object-oriented language and involves control like imperative programming. Object-oriented programming is typical of classes which encapsulates state and can change its state through invoked procedures, usually known as methods. In Unity 5, the programmer can define methods that are invoked at certain events by the game engine, for example when the game updates, when two game objects are colliding, or when game objects are destroyed. The program enters the top of the method in an imperative style and computes everything inside sequentially until leaving the method at the bottom or by a return-statement. Methods like these, especially methods that return nothing, for example with the type void, enables to check or change the temporary state in the program.

Callbacks methods and imperative operations tend to become problematic if not handled carefully. Race condition is a term that describes the problem when an output of some sort is dependent on a specific sequence or timing of uncontrollable events. Bugs can appear this way when the order of events does not occur the way the programmer intended. Ideally, if every operation occurs in the intended order, this type of bug is technically avoidable. A synchronous program is just one sequence of tasks that is completed one after another, excluding bugs from race condition if controlled and structured correctly. However, when synchronous programs get large, and without a good structure and model, bugs may appear like race condition. An example in Unity 5, multiple game objects may start in a scene at the
same time. The programmer can define a method named `Awake` in game component scripts and is invoked by the engine upon the creation of the attached game object. If multiple scripts reference each other in the `Awake` method, the output may vary depending on which order the engine creates each game object. This can lead to unintentional results. Because video games are an interactive medium, and many games engines is modelled with game objects in scenes, the games can appear like asynchronous programs, like multiple tasks is handled simultaneously. Without any good model for these behaviours, the programmer must have good awareness of the order of computations.

Another issue lies in how dependencies are expressed. For example, in Unity 5 the programmer cannot re-define an UI image’s position by what dependencies it has. Instead, the programmer can only keep it up to date and to simulate dependency by changing the state explicitly. This is inevitable in imperative programming. The programmer must manually keep the system up-to-date when changes are needed.

### 3.1 Reactive programming

Reactive programming, or rather reactive-functional programming, enables describing stateful programs declaratively. A stateful program refers in this case to a program that changes the value on data multiple times while running. Unlike pure functional programming, reactive programming notices the impure and unpredictable nature of some programs. It involves modelling of relationships between objects through function composition. Functions are evaluated whenever any changes or events occur. Ideally, the programmer writes the logic in a program and lets the computer deal with the mechanism behind it. An object in a reactive program is a sequence of pure functions. In contrast to object-oriented programming, a reactive object is defined by what functions it is composed of, what dependencies it has, and what output it creates. As an effect, reactive objects can be declared and defined as variables in single handed expressions. A reactive object does not have to be defined with a unique data class.

Reactive programming is a general term in programming that focuses on programs that react to changes. Changes in a reactive system can be specific data values or events that occur during the program. The programmer determines which or what these values or events are and can then use these changes to compute text, i.e., the program “reacts” to that change. An example is how cells work in a spreadsheet. 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. As seen in Figure 1, if object C is dependent on B, while B is dependent on A, then, if A changes, B is revaluated, followed by C. Changing object A creates a chain of reaction. This behaviour makes it possible to have multiple objects being dependent on a common object. A single object could also be dependent on multiple other objects.
Event-driven programming is a type of reactive programming. Hence in this thesis, the term reactive programming refers to the concept of reactive-functional programming, which involves function composition with reactive objects. Event-driven programming is often used with imperative programming where methods in objects are registered as callback methods on certain events. These methods may alter the program’s state or do I/O operations. Unless the programmer instructs the program to invoke new events to notify when changes happen, other parts of the program will not know what had happened until they have the control to check the current state of the program. The order of events can also be troublesome without a good model and may cause race condition bugs.

3.2 Reactive programming with UniRx

UniRx is a library that adapts the Rx.NET for Unity. Rx.NET is a library for reactive-functional programming written in C#. Ben Christensen describes RxJava, which is similar to Rx.NET, but written in Java, as following:

RxJava is a specific implementation of reactive programming for Java and Android that is influenced by functional programming. It favors function composition, avoidance of global state and side effects, and thinking in streams to compose asynchronous and event-based programs. It begins with the observer pattern of producer/consumer callbacks and extends it with dozens of operators that allow composing, transforming, scheduling, throttling, error handling, and lifecycle management (Christensen 2016)

To be even more clear, reactive-functional programming and functional reactive programming (FRP) have similarities but are not entirely the same. While there is agreement that the concepts are different, there are various opinions of what the differences exactly are. Christensen states that functional reactive programming differs by involving continuous time, whereas reactive-functional programming with Rx.NET only deals with discrete events. Heinrich Apfelmus (2016) states “it is all about whether your operations on values have a
well-defined denotational semantics or not”. He means that the key difference is not because continuous time. In the library Reactive Banana in the programming language Haskell, there are specific data types where time is included. The result of Rx.NET of not having the same defined semantics is that it is possible that events happen in incorrect order when multiple events occur nearly simultaneously. This would be avoidable by checking the order in time. However, this problem did not show to cause any specific problems in this work. Nevertheless, it may be good to be aware of such issues. Only discrete-time events were used in this work.

The most interesting datatype in UniRx is `IObservable<T>`. The type is an interface and holds any type of value that can be “observed” for new values. An `IObservable<T>` can be extended into new observable variables by applying operations like filtering and transformation. A filter-operation can be used to decide what values are important and a transform-operation can be used to transform the value, like changing its data type. The `IObservable<T>` can be viewed as a stream of values and it can also be viewed as a single variable that changes its state. It is helpful to think like it is a stream of values to understand how some of the operations work, which are inspired by function composition on data lists with the library LINQ.

A reactive object can encapsulate state by having recursive functions, which means it can remember previous values that alter the new value, i.e., by accumulation. As reactive objects do not have side effects, they cannot by themselves produce any sort of output that may be interesting for the user, for example audio and graphics. Instead, they can be a part of the program that observe these values and make side effects like rendering and printing text on a display.

An important concept about reactive programming is that it is event-driven, which means that there has to exist an original event source. Without any event, or activity, there cannot be any reactivity. Therefore, the first object in a sequence of reactive objects has to be independent and only create events. In Rx.NET, the type `Subject<T>` is provided for the programmer to manually invoke events for event-driven programming. `Subject<T>` inherits from `IObservable<T>` which makes it possible to observe it like any other reactive object. It makes it possible to adapt frameworks that are not reactive to Rx.NET. For example, the framework in Unity 5 is not based on reactive programming but it can be adapted to it by using a `Subject<T>`. In this thesis, the notation of a method in the form `Method()` refers to a method with an arbitrary list of arguments. The method `OnNext()` sends a new value to all objects that subscribe, or “listen” to it. Below, the update loop in Unity 5 is adapted by using a `Subject<T> OnUpdate` that emits the last delta time:
As UniRx is a library that adapts Rx.NET to Unity 5, the example above is not essential, since a similar method already exists, namely Observable.EveryUpdate(). To give a brief example of how UniRx can be used in Unity 5, mouse motion can be represented as a reactive value with the type Vector2. The value is declared with the type IObservable<Vector2>:

```csharp
var mouseMotion = Observable.EveryUpdate()
    .Select(_ =>
        new Vector2(
            Input.GetAxis("Mouse X"),
            Input.GetAxis("Mouse Y")))
    .Where(motion => motion != Vector2.zero);
```

In this case, the code can be read as “every update, select a new vector with mouse motion x and y where the motion is not zero”. The variable mouseMotion can then be extended by creating a new observable value that is dependent on mouseMotion. The variable can also be observed through a subscription to create output actions. For example, mouseMotion together with a sensitivity variable can be used to move an UI image, like a mouse pointer:

```csharp
mouseMotion.WithLatestFrom(sensitivity, (m, s) => m * s)
    .Subscribe(v => mousePointer.transform.position += v);
```

If sensitivity is changed, the whole behaviour changes. Note that in the lambda function inside the parentheses of Subscribe(), the variable position is mutated by adding vector v. This mutation is a form of output that is not applied by the reactive object. The reactive object that is returned from the operationWithLatestFrom() applied on mouseMotion is subscribed to create side effects, like mutation. The return type of Subscribe() is IDisposable and is used to stop the subscription to not be invoked anymore. Unlike IObservable<T>, IDisposable cannot be observed and extended like a reactive object.

### 3.3 Previous work

Examples on how Rx.NET can be used in Unity 5 with UniRx have been presented in various blog posts and articles on the internet. Cole (2016) demonstrates an implementation of a first-person controller by using observable types from UniRx together with Unity’s component system. It includes how user input can be adapted to observable variables, demonstrating different techniques that can be used with UniRx, creating data types that are used within reactive sequences, and separating observable variables in different game components. The results are a first-person controller that can look around, walk, run, and jump, controlled by
user input. A camera animation is controlled in a separate game component that makes the camera move up and down while the first-person controller is moving. However, there are few examples on how observable variables can be composed together. Cole uses UniRx in a way that is similar to typical event-based programming, which involves a lot of imperative programming.
4 Materials and methods

The materials used in this work consist of different software and applications. The methods describe how the work was achieved by use of programming patterns, techniques, and structures. This chapter aims to give the reader a clear understanding of these software systems, and to enable the reader to apply the same methods in their own work.

4.1 Materials

Unity version 5.6

Unity 5.6.0f3 is the game engine that was used in this work, released on March 31, 2017. In comparison with its previous version, the new version includes improved 2D features, lighting, particle systems, and more.

Unity Standard Assets

The 3D models used in this work is the Standard Assets package that is included when downloading Unity 5.6 from Unity’s official website. The package includes a folder named Prototyping that has 3D models, materials and prefabs intended for prototyping 3D levels.

UniRx version 5.5.0

Version 5.5.0 of UniRx, released on October 3, 2016 on Unity Asset Store, is used. According to Kawai, the author of the library, the library is supported on Unity versions 4.6 and 5. In this thesis, the version 5.6 was used for Unity and is therefore not verified to be supported by UniRx. However, the library worked without any noted problems in this work.

Visual Studio 2015

Microsoft Visual Studio Community 2015 is a software used for the development of computer programs in different programming languages and was used for the programming in this work. Visual Studio Tools for Unity was used to integrate the IDE with Unity’s editor, for example to receive debug information from Visual Studio when working in the Unity editor.

C#

C# is an object-oriented programming language and has a similar syntax as the languages C, C++, and Java. Scripts in Unity 5 can be written in C# and the library UniRx is written in C#. C# is also a common language used in the industry for game development. It is therefore the choice of the programming language in this work.
4.2 Methods

4.2.1 Observable user input

Unity’s methods for accessing user input is adapted to be observable in a reactive manner. UniRx does not provide that in its current version. As Unity’s input methods were static, thus globally accessible throughout the program, the observable input is made static as well. The observable input was placed in static classes where each class was categorised by the type of input device it managed, like keyboard, mouse, and joystick. This was to make it clear and apparent where different types of user input were accessed.

In Unity 5, the standard way to acquire input from keyboard, mouse, and joystick is through each game loop. The state of input devices can be checked for each update by using methods from the static class `Input` in Unity’s API. This was utilized with the static method `Observable.EveryUpdate()`, provided by UniRx, to make the input as observables. The operation `DistinctUntilChanged()` was used to get input values only at change, preventing the generation of redundant input values at every update. The operation `Skip(1)` was used to skip the first emitted value from all inputs. If not performed, a value is issued when the program starts without the detection of any user input.

The static methods for different input actions are designed to include observable values as arguments. A method that returns key presses from a specific key would be based on the observable key that is passed to the method. If the key name later changes, the observable value that was previously returned, is changed to emit key presses from the new key.

4.2.2 Object-oriented reactive programming

Due to the framework of Unity 5, and due to the limitations of this work, the style used for programming can be considered as object-oriented reactive programming. The definition of this term is described by Elisa Gonzalez Boix et al. (2013) as objects that contain reactive variables. The objects themselves are not reactive. It is their contents that are built of reactive variables composed together. As any other object-oriented style, these individual values can be mutated or accessed through a set of methods defined by the programmer. In contrary to functional reactive programming, where a reactive object has only one input and one output, the object-oriented object with reactive values has multiple inputs and multiple outputs. The object-oriented object is created in the memory and functions as a container of state that allows itself to be affected by any outside sources.

The `MonoBehaviour` class is used for creating components that can be attached to game objects in a game scene. A single game object can have a multitude of different components, provided by the component based architecture in Unity 5. While they are called components, they are technically objects like in object-oriented programming. Each component can contain a state that is mutated during the program. A component can have multiple inputs and outputs with mutator methods and accessor methods that are invoked
from outside. In this work, components that are made of classes derived from
MonoBehaviour will contain observable variables that are encapsulated. The accessibility on
the variables from outside sources is controlled through class methods to ensure, in this
specific context, a bug free system.

Subject<T> and BehaviorSubject<T> from UniRx is used for observable variables that are
encapsulated within MonoBehaviour classes. This is due to how Unity 5 initiates components
through callback methods in an implicit order. With these types, each variable can be
instantiated when they are declared as member variables in a class. An alternative way is to
instantiate the member variables in the callback methods Awake() or Start(). These
methods are invoked when the component is created when the program is running. Unity 5
provide these because the programmer is not allowed to define constructors in classes that
derive from MonoBehaviour.

Errors may occur when a component accesses a variable that is not instantiated from another
component. If instantiation of member variables is done in Awake() or Start(), the
invocation order for different components becomes a problem. A race condition occurs.

IObservable<T> is not used for encapsulated member variables due to the issue of race
condition. The type cannot be instantiated when declared and later be defined with a new
value which observers adapts to. There is a risk that observers observe a variable that has not
yet been defined with the intended value. Instead, a Subject<T> or a BehaviorSubject<T>
can be used as both an observer and observable simultaneously. The pattern used in game
components is to have encapsulated member variables with either the type Subject<T> or
BehaviorSubject<T>, depending on the requirement. Each variable is instantiated in their
declaration with new Subject<T>() or new BehaviorSubject<T>(...). The variables are
composed together with UniRx operations inside the callback method Awake. The results
are applied to member variables by subscribing them to the resulting compositions. The
subscription is done by passing variables in the parentheses of the operation
Subscribe(IObserver<T> observer). This method works because Subject<T> and
BehaviorSubject<T> can both observe other variables, and be observed.

The difference of Subject<T> and BehaviorSubject<T> is that the latter always stores the
latest value and can therefore be accessed as a stateful variable. In contrast, when
Subject<T> emit values, the values are not stored and could potentially be lost forever.
Another difference between the two types is that BehaviourSubject<T> emits the latest
value to new observers at subscription and then continues like Subject<T>. This became
useful for solving the problem of race condition. If a component was dependent on the initial
state, or value, from a variable in another component that was instantiated before, the
component was ensured to receive the initial value from the other component.
4.2.3 Component granularity

Components for game objects are kept as coarse-grained as possible. This means the system of game components is broken down to as few components as possible. Individual components contain large sets of tasks, and the reason is to retain a system that is as understandable as possible. The opposite of a coarse-grained system is to divide it into many more small components, in other words fine-grained system, but this approach often leads to more complex systems. Fine-grained systems are often more flexible than coarse-grained systems, but, in favour of the purpose, a more coarse-grained approach is used. For example, a single component may control the behaviour of a player character, containing movement, health, score, actions, etc. Another component may control the behaviour of a game session and its rules, containing references to all player characters, keeping track of the score, and controlling when the game pauses or continues. Fine-grained systems generally also decrease the execution speed of applications by the addition of overhead operations.

A simplified version of the model-view-controller pattern is used. Each type of component is divided in two. One is a controller component that controls state and data that are important for the game mechanics. The other is a UI component that handles user interfaces, for example user input, animations, audio, and particle effects. The different user interfaces are kept in the same UI component, in favour of coarse-granularity.

4.2.4 Shared and non-shared state

Data that are shared or not shared between game objects are divided through different methods. Local state for individual game objects are stored within MonoBehaviour scripts. The state includes variables and values that change and are unique to a specific game object. An example is the number of lives left in a game for a player character. The player may start with a maximum number of lives of three. When the character loses one life, two lives are left. This value is unique for that specific player character while other characters have different number of lives left.

Shared state between game objects are stored within ScriptableObject scripts. Classes that derive from ScriptableObject can be created through Unity’s editor as objects that exist as datafiles in the project. The object can contain a set of variables and methods and can be referenced by game objects in a scene. In this work, the ScriptableObject is used to store raw public data and not to contain any class methods, or use any techniques like encapsulation. The data within the object is kept as close to primitive types as possible by using basic data types like float, int, bool, and string. Therefore, observable values with UniRx are not used with ScriptableObject. An example of using the method is storing the maximum number of lives a player character can have in a game. Player characters that exist as game objects in a scene can have a reference to the same scriptable object to base their maximum number of lives on.

Scriptable objects are used to store data that exists between multiple scenes and when the program is stopped. The values of these objects are stored in the computer memory and thus
continue to exist when the computer is shut down. In this work, game settings for number of players is stored with scriptable objects to communicate between multiple game scenes. Controller settings are stored in scriptable objects to save the values between multiple game sessions and when the computer is shut down and restarted.

4.2.5 Mutation on Unity’s non-reactive objects

Data classes from Unity’s API were left intact and are not adapted to be reactive due to the limitations. *Mutation* on objects of these types is done imperatively through callback methods and in anonymous methods subscribed to observable values. For example, the velocity of an object of type Rigidbody was modified by subscribing to an observable variable named jumpVelocity and assigning the values directly to the velocity field on Rigidbody:

```csharp
jumpVelocity.Subscribe(v => Rigidbody.velocity = v)
    .AddTo(this);
```

Parameters that are subscribed and used for mutating states are expected to contain the most recent updated value. Any calculations and operations are not performed in the subscribing methods. These methods are only intended to assign values to non-reactive objects.

4.2.6 Preventing memory leakage

UniRx uses the delegate type from C# to handle events for observable variables. The version of C# used in this work does not handle garbage collection automatically for the delegate type. Therefore, the programmer has to control how garbage collection is handled from subscriptions on observable variables. The operator Subscribe() from IObserver<T> returns the type IDisposable, which can be used to manually stop a subscription, thus preventing memory leaks. UniRx provides the operator AddTo() from IDisposable to automatically stop a subscription when a game component is destroyed during the program. In this work, AddTo(this) is used after every subscription operation, where this is the game component that contains the subscribers.
5 Results

5.1 Observable user input

5.1.1 Keyboard

The method `Key` in the static class `Keyboard` returns an observable value of type bool from a desired key. The return value is true when the key is pressed and returns a false value when released:

```csharp
return Observable.EveryUpdate()
    .WithLatestFrom(key, (_, k) => Input.GetKey(k))
    .DistinctUntilChanged()
    .Skip(1);
```

The method takes an observable key code as an argument. Making the argument an observable value enables the new observable value to correspond if the key code is changed.

Inside the method, a new observable returned and is created by first combining `Observable.EveryUpdate()` and the observable key code with the operation `WithLatestFrom()`. This operation extends every update by taking the latest value from another observable value. In this case, it is the key code that is passed on. The process of the combination is defined with a function that takes the values from the original observable and the other observable and returns a new value of any type. A lambda expression is used in this case for combining every update and the latest key code. The first parameter from the update value is ignored and thus clarified by naming it with just an underscore, which is a valid name for variables in C#. The second parameter is passed to `Input.GetKey()`, which is a Unity method that returns true or false whether a key is currently pressed or not.

Without extending the observable any further, the result is an observable value that emits Boolean values at every update. This may be desired in some cases but not for this specific method. The operation `DistinctUntilChanged()` passes through values that are different from the previous ones. The stream of values become a sequence of values where each value always differs from the previous one. In this case, the new observable value emits Boolean values just at the first update where the key press is changed.

`KeyDown()` and `KeyUp()` are static methods that extend `Key()`. `KeyDown()` filters only values that are true, in other words, when a specific key is pressed. `KeyUp()` does the same thing but only with values that are false, or in other words, when the key is released. `KeyDown()` is shown below:
public static IObservable<Unit> KeyDown(IObservable<KeyCode> key)
{
    return Key(key).Where(b => b == true).AsUnitObservable();
}

KeyDown() takes an observable key code as argument and passes the value to the method Key(). Where() is an operation that filters values by a function that returns true or false. The argument to the function must be the same data type as the values that are received from the previous observable, which in this case are Boolean. When a value is received, the operation passes that value to the function and checks the result. If the result is true, the value passes through. If the result is false, the value does not pass through and nothing happens. Observers would not be notified if a value is not passed through. AsUnitObservable() transforms the observable to only emit values of type Unit. Unit represents void and in other words represents nothing. This is due to that every value that passes through the previous filter operation can only be true. It is therefore redundant information to emit values that have the type Boolean in this case. The event itself of a pressed key is only interesting. Figure 2 shows a simplified view on how KeyDown extends Key.

Fig 2. A simplified representation of how the observable keyboard input is modelled

5.1.2 Mouse

Button() is a static method in the static class Mouse that takes an observable button index of type int and returns true or false when a mouse button is pressed or released. The method is like Keyboard.Key() but replaces Input.GetKey() with Input.GetMouseButton(), which takes an index as an argument instead of a key code. Button() is extended with ButtonDown() and ButtonUp() which are equivalent to the methods Keyboard.KeyDown() and Keyboard.KeyUp().

Input from mouse movement is made on the same way as the code example in page 13. The value is stored as a public static readonly variable named Motion in the Mouse class. The
keyword `readonly` is used to resemble an immutable constant variable. The keyword `const` could not be used because of how UniRx was implemented.

5.1.3 Joystick

The static class `Joystick` contains methods for obtaining input from joystick devices. Input from joysticks in Unity includes buttons and axes. Joystick buttons, like keyboard keys, are either pressed or released, true or false. Joystick axes generate decimal values, unlike binary buttons. An axis value ranges between -1.0 and 1.0.

In Unity 5, button input from joystick devices is either acquired by defining buttons in the input settings in the Unity editor, or by passing key codes to the methods `Input.GetKeyDown()`, `Input.GetKeyDown()`, or `Input.GetKeyUp()` from Unity’s API. For example, the enumerated type `KeyCode` contains the values `JoystickButton0` and `Joystick3Button4`. The first value refers to the first indexed button on any connected joystick device. The second value refers to the fourth indexed button on the third out of multiple connected joystick devices.

The static method `Button()` in `Joystick` translates to specific key codes for joystick buttons to index values. The method takes an observable joystick index and an observable button index as arguments and is combined and translated to a corresponding key code. The key code is then used in the same way as keyboard keys are accessed with `Keyboard.Key()`. `Button()` returns an observable value that emits true or false when a joystick button is pressed or released. The method is extended with the separate methods `ButtonDown()` and `ButtonUp()`.

The static method `Axis()` returns an observable value that emits decimal values when a joystick axis position is changed from an observable joystick index and an observable axis index. The values passed in are combined to a string and then used with Unity’s method `Input.GetAxisRaw()` for every update. `DistinctUntilChanged()` is used to make sure that only new axis values are emitted in comparison of their previous emitted value. `Skip(1)` is used to ignore the first emitted value as it is not triggered by the user.

The format of the string that is used in `GetAxisRaw()` is “Joystick [j] Axis [a]”, where `j` is the joystick index and `a` is the axis index. This type of format requires that Unity’s input settings have defined axis names using the same labels. In this work, the input settings where modified to have defined joystick axes from “Joystick 0 Axis 0” to “Joystick 4 Axis 9”.

In the game, the user can use a joystick axis to trigger a laser gun. The method `Joystick.GetAxisAsButton()` takes a joystick index and button index in addition with a decimal value describing the dead zone. The dead zone describes a range of values where nothing happens. When the axis value is above or under the dead zone, depending in which axis direction is set, the return value is true, or else false:
5.2 First-person controller

The class FirstPersonController was designed to simulate a character that can run, jump, and look around from a first-person camera view. FirstPersonController is implemented as a class that is derived from MonoBehaviour and can be added as a component to game objects. The component is made to be a stand-alone component that serves as a component that suits any first-person game. For this work, it is used as the base for a functioning player character.

FirstPersonController contains references to a rigid body component, a camera component, and a game object that is used for checking walkable ground. A reference to an object of the type FirstPersonControllerSettings, derived from ScriptableObject, was used in the same way where shared data is stored, for example jump height and movement speed. Public variables were used to be able to get values by dragging and dropping game components in the Unity editor. Otherwise, the variables remain null:

```csharp
public class FirstPersonController : MonoBehaviour
{
    public FirstPersonControllerSettings settings = null;
    public new Camera camera = null;
    public new Rigidbody rigidbody = null;
    public GameObject groundTrigger = null;
    //...
}
```

The types Subject<T> and BehaviorSubject<T> were used for encapsulated member variables. The variable isGrounded used BehaviorSubject<bool> and represented a Boolean value for the first-person controller standing on the ground or not. Subject<T> was not used in this context because the variable represented a value that needed an initial value, in this case the value false. Classes outside of FirstPersonController received the last value from isGrounded upon subscription on its accessor method.

The accessibility to the variables for outside sources were controlled through C# Accessors. Every private member variable of types Subject<T> or BehaviorSubject<T> had their own accessor where the accessibility varied. Every accessor returned the type IObservable<T>. Some could be set publicly, and some could only be set privately, but all could be accessed from outside classes through a get-method. The variable isGrounded had the accessor IsGrounded, with an uppercase, where the get-method was public, and the set-method was private. The get-method returned isGrounded as an IObservable<bool> without any special features. The set-method subscribed isGrounded to the parameter that is passed in
and used the operation AddTo(this) to automate garbage collection when an object of FirstPersonController is destroyed:

```csharp
private BehaviorSubject<bool> isGrounded = new BehaviorSubject<bool>(false);

public IObservable<bool> IsGrounded {
    get {
        return isGrounded;
    }
    private set {
        value.Subscribe(isGrounded).AddTo(this);
    }
}
```

In the method Awake, walkable ground is checked by using the operators OnTriggerEnterAsObservable() and OnTriggerExitAsObservable() on the referenced game object used for ground checking. The first operator returns an observable value that emits colliders which enters an attached collider that is set as a trigger. The second operator does the same thing but when colliders exits the trigger. These are merged together after a Boolean value has been applied to both, resulting in the value true on trigger enter, and the value false on trigger exit.

The number of colliders that enter and exit the trigger are counted with the operator Scan() to find out how many colliders are inside the trigger simultaneously. When the count is more than one, the value is true, or otherwise false. If this is not done, the value false is emitted when a collider exits the trigger, while there are still other colliders inside the trigger. The resulting observable value is applied to the accessor IsGrounded after DistinctUntilChanged() is used for preventing redundant values:

```csharp
IsGrounded = IsGrounded =
    Observable.Merge(
        groundTrigger.OnTriggerEnterAsObservable()
            .Where(c => !c.isTrigger)
            .Select(_ => true),
        groundTrigger.OnTriggerExitAsObservable()
            .Where(c => !c.isTrigger)
            .Select(_ => false)
    .Scan(0, (n, b) => b == true ? n + 1 : n - 1)
    .Select(n => n > 0)
    .DistinctUntilChanged();
```

IsGrounded was used in the same class to determine the maximum speed and acceleration of the movement, depending on if the first-person controller was touching the ground, or not:
MaxSpeed = IsGrounded.Select(b => b ? settings.groundSpeed : settings.airSpeed);

MaxAcceleration = IsGrounded.Select(b => b ? settings.groundAcceleration : settings.airAcceleration);

Input for moving, jumping, and rotating the FirstPersonController was handled through settable and gettable accessors that were public, thus allowing input from other classes. The variable movementInput of Subject<Vector2> where used for the movement direction, where the x in Vector2 represented left and right movement, and where y represented backward and forward movement, in relation to the game object. The accessor MovementDirection returned the movementInput as an IObserver<Vector2>. The input values to MovementDirection were clamped to a magnitude of 1 in the set-method to prevent too large values, as the direction was used for multiplication with a maximum movement speed:

```csharp
set {
    value.Select(v => v.magnitude > 1 ? v.normalized : v)
        .Subscribe(movementInput)
        .AddTo(this);
}
```

The velocity of the referenced rigid body was controlled through a BehaviorSubject<Vector3> to create a layer of abstraction. The accessor Velocity used the input values to manipulate the velocity directly in the rigid body:

```csharp
private set {
    value.Subscribe(v => rigidbody.velocity = v).AddTo(this);
}
```

The set-method was used in FirstPersonController to define the velocity by merging movementVelocity and jumpVelocity. The variable speedAcc is MaxSpeed and MaxAcceleration combined into a tuple. The variable jumpVelocity is an upward velocity when input is received for jumping while touching ground:
```csharp
var movementVelocity = Observable.EveryFixedUpdate().WithLatestFrom(MovementDirection, (_, v) => v).Select(v => v.y * transform.forward + v.x * transform.right).WithLatestFrom(speedAcc, (v, t) => Vector3.Lerp(rigidbody.velocity, v * t.Item1, Time.fixedDeltaTime * t.Item2)).Select(v => new Vector3(v.x, rigidbody.velocity.y, v.z));

Velocity = Observable.Merge(movementVelocity, jumpVelocity);
```

The same pattern was used in the rest of the class. Camera rotation was done in a similar way as mutating the velocity directly in the referenced rigid body, but instead changing the rotation on the referenced camera component. The complete definition of FirstPersonController is presented in the Appendix.

### 5.3 Player character components

The class PlayerCharacter was designed to contain functionality for a player character in a first-person shooter game. It included player health, number of lives left in a game session, etc. Audio, UI, and some of the animations, were separated from the class PlayerCharacterUI.

**PlayerCharacter**

The same pattern as in FirstPersonController was used in PlayerCharacter. References to non-reactive objects were accomplished using public variables, thus accessible in Unity’s editor view. Encapsulated observable variables and their accessibility were controlled through accessor methods. Definitions for reactive behaviours were made in the method Awake().

Among the references on non-reactive objects were FirstPersonController and LaserGun. The first was used for the character movement, jump, and rotation. The second was used for shooting projectiles towards the camera direction. The LaserGun is briefly described as a container of observable variables, where the same pattern was used as in the other classes. Inputs for shooting projectiles with LaserGun were handled in the accessor method TriggerInput and was utilized in PlayerCharacter. The inputs were received and distributed to LaserGun and FirstPersonController by the accessor Input in PlayerCharacter.

Interaction between player characters were mostly performed by hitting each other with either projectile shots from laser guns or explosions caused by charging up and releasing projectiles. DamageInfo, derived from MonoBehaviour, was used as an interface for different types of damaging sources, including projectiles and explosions. It contained primitive data types,
without any observable variables, including the amount of damage and a reference for which player who was causing the damage. This was used in Awake in PlayerCharacter to check received damage on trigger enter. The triggering collider is checked if it has a DamageInfo component attached to it. The damage amount is selected, if it is not the same player that is causing the damage:

```csharp
ReceivedDamage = this
   .OnTriggerEnterAsObservable()
   .Select(c => c.FindComponent<DamageInfo>())
   .FilterMaybe()
   .Where(info => info.Player != this)
   .Select(info => info.Damage);
```

The player health starts at a maximum value, and is reduced for each received damage point. Similarly, received health points consist of the incrementation of the health value.

**PlayerCharacterUI**

The class PlayerCharacterUI was used to handle the presentation to the user through graphics and animations that did not impact the core gameplay mechanics. It contained a reference to a PlayerCharacter, used for the different presentations. The amount health was presented with an image on the screen that changed scale horizontally. Number of lives left was presented with a text on the screen. When the player character received damage, a red image, filling the entire screen, was displayed and faded out.

Animation for the first-person camera was defined in PlayerCharacterUI. It consisted of camera shaking when the character received damage and vertical camera bob when the character jumped or landed on the ground. The camera shake was defined by selecting an animated Vector3 whenever player character received damage. The Vector3 was an observable value that constantly emitted a perlin noise for each axis. The shake was faded out by decreasing the shake magnitude over time. The methods MoveTowards01() and PerlinShake(), as shown in the Appendix, was used to achieve the final result for the camera shake:

```csharp
var cameraShake = character.ReceivedDamage
   .Select(_ =>
      Util.PerlinShake(
         Observable.Return(75f),
         Util.MoveTowards01(0.3f)
         .Select(t => (1 - t) * 0.2f))
   .Switch()
   .StartWith(Vector3.zero);
```

The full camera animation, together with camera shake and camera bob, was combined and applied by directly manipulating the camera position. In this case, the pattern for settable
accessors was not used. The original camera position upon Awake was stored to be included in the combination of animations:

```javascript
var camera = playerCharacter.firstPersonController.camera;

Observable.CombineLatest(
    cameraOriginalPosition, cameraShake, cameraBob,
    (a, b, c) => a + b + c)
  .Subscribe(p => camera.transform.localPosition = p)
  .AddTo(this);
```

Animation for the laser gun was made in similarity with the camera animation. The animation involved recoil when fired, up-and-down movement when the character was running while on the ground, and delayed movement that followed the original position while the character was falling. The position on the laser gun was directly manipulated through the reference from PlayerCharacter.
6 Discussion

The advantages of using UniRx with Unity 5 seem to depend on how it is used. The patterns that were used in this work suited well with the combination of encapsulated objects containing reactive properties. Game components were not reactive themselves, but communication and relationships could be described clearly and loosely coupled through interface components, like DamageInfo used in PlayerCharacter. The way Unity 5 enables to get components from game objects on collisions and triggers allowed components and their encapsulated variables to have clear definitions within classes. A similar way for communicating between objects was to prepare with settable observable accessors for allowing input from outside sources, like for movement, jump, and rotation input in FirstPersonController. It arguably leads to a vague definition on some relationships because it is not clear where the input comes from, which is slightly different from declarative programming. However, the input values could be controlled, like clamping the movement direction value in FirstPersonController. Controlling the input was important to receive desired values only.

Composing and declaring observable variables made it possible to create behaviours without need for special data classes. For example, checking ground for the first-person controller was declared and defined as an observable variable in just one expression. In this case, the ground checking was fairly simple and could probably be extended even further with checking the ground type, slope, etc. The same expression could possibly be written as a static function to enable it to be reused in other contexts. In contrast, a common component-based solution would have been to have this feature as a type of component, for example a class with the name GroundChecker. This would lead to the use of more data types in a project, which were avoided by composing observable variables.

When making logical solutions by declaration, the notion of “good versus bad code” depended mostly on how straightforward and concise solutions were expressed. When dealing with just logic, without thinking about control, the logical solution is always right in theory. The way IsGrounded was defined in FirstPersonController is logically correct and can therefore not be wrong, as long as the underlying mechanism works correctly. Because imperative programming involves control, rather than just logic, bad code can become problematic. Bad code may come from programmers with tight schedules and stress. Imperative programming requires understanding control flow, and errors in the implementation can lead to bugs that fails to meet product requirements. Fixing badly written code is time consuming. Badly written code is not encouraged, but eliminating control in imperative programming, and instead focusing on logic with declarative programming, could be beneficial for enduring short term decisions in code during stressful periods of game developing.

This work did not study how UniRx can be used by programming teams, but rather by a single programmer which did not involve the need for communication and cooperation within a group. For reactive programming to get foothold in gameplay programming in the game
industry, the method for team work would need to be proven to help when developing games. As in any other field, there are risks taken when investing in games, and is therefore most safe to use proven methods, including imperative object-oriented programming and component based programming. Often in the form of high-level scripting languages in established game engines like Unity 5 and Unreal Engine. There would be a big risk to use reactive and declarative programming in these cases because imperative programming is common in schools and educations. Most game programmers tend to have skills that are adapted to the industry, which is positive, but a change from imperative to declarative programming for gameplay development will probably depend on the act of an established game company. If reactive programming is going to become a norm in gameplay programming, an increased interest for reactive programming and game development will probably start from independent developers. Independent developers have an advantage in the sense that not being dependent that a team should have specialized knowledge on reactive programming.

In terms of reactive programming, game objects and components could rather have been just observable variables that encapsulated state within UniRx operators. The combination of UniRx and Unity’s object-oriented, component-based, framework required that methods had be compensated for using different programming techniques. In object-oriented programming, state and change is encapsulated in data classes. In reactive programming with UniRx, state and change can be encapsulated within observable variables. Unity’s component based framework and UniRx’s reactive features made the separation of reactivity and non-reactivity difficult to understand, and the combination difficult to implement to a consistent whole. A lot of concepts were combined, like object-oriented game objects and components, encapsulated classes, accessor methods, explicit mutations, and object references, and reactive objects, observables, subjects and behaviour subjects, subscriptions, implicit mutations, operators, and so on. This resulted in complex programming patterns that could be difficult to understand for new programmers, and will probably not become the norm for the future of game programming. An important criterium for reactive game programming to become popular would arguably be in favour for simplicity.

A future work on using UniRx in Unity 5 would be to implement wrappers for Unity’s API. Krishnamurthi et al. (2006) proved that an object-oriented GUI toolkit could be adapted to a functional reactive environment, and their methods could possibly be used for Unity 5. However, this would create a layer on Unity’s API which would be unfamiliar for experienced Unity users. The benefit is Unity’s availability, optimizations, editor, and cross-platform features, but alternative game engines and libraries exists for using reactive programming. A suggestion is to design a reactive framework for game programming using Rx.NET with MonoGame. MonoGame provides a framework, based on Microsoft’s XNA, for developing video games, including 2D and 3D graphics, porting cross-platform, etc. MonoGame does not include an editor like Unity 5, but the lower level of features with MonoGame would give more freedom to make a reactive based environment for game programming. For example, game objects can be made as fully reactive objects with observable variables.
7 Conclusion

The work showed that a limited first-person shooter can be made in Unity 5 while using reactive techniques with the library UniRx. The work showed that the results could be achieved by a single person, but the same methods used in team work remain to be tried, in terms of communication and cooperation. For example, the team could consist of two programmers, or more, who are dealing with code that involves linked dependencies.

The purpose of this thesis was to look at the advantages and disadvantages of using UniRx. Advantages were most apparent when describing the game logic. Extending observable variables for new behaviours became second nature, and describing game logic in a reactive and declarative way was helpful for understanding the program as a system with relationships. Disadvantages were most apparent when combining reactive techniques with Unity’s established framework. Unity 5 was based on an object-oriented component based framework which forced programming patterns that adapted multiple ideas and concepts. It led to a complexity which is arguable avoided if one of the concepts is excluded. For reactive programming to gain popularity in game development, simple and understandable patterns need to exist. A suggestion for future work is to use the library Rx.NET with MonoGame to make an environment that embraces the ideas from reactive programming, for example including fully reactive game objects. It would test reactive programming even further if it is a viable technique for gameplay programming.
References


Glossary of terms

**Accessor** – Refers to method in C# that has similar syntax to variables when referred to, thus not having parentheses like normal methods. An accessor can be defined with either a set-method, get-method, or with both.

**API** – An abbreviation for Application Programming Interface which means a set of defined tools and building blocks to make it easier to develop computer programs for programmers.

**Encapsulation** – A technique for preventing direct access to an object’s inner components.

**FPS** – An abbreviation for First-person Shooter which is a video game genre where a player controls a character from first-person perspective and can use guns to shoot targets and opponents.

**High-level scripting language** – A programming language that can be written in and be interpreted by a separate program allowing to add new features and behaviours.

**Lambda function** – An anonymous function in C# that can be assigned to variables and can be used as argument types in methods and functions.

**Local multiplayer** – A feature in video games where multiple players play simultaneously on a single video screen.

**Model-view-controller** – A programming pattern where data, control of data, and presentation are separated.

**Mutation** – In programming, mutation refers to when the value in a variable is changed.

**Mutator method** – A method in a class that causes any type of mutation, i.e., on member variables.

**Perlin noise** – A gradient noise often used for simulating randomness with a smooth transformation.

**Side effect** – A function or expression in a program that mutates or modifies any state outside of its scope.

**Tuple** – A data type that contains a list with elements of varying data types.

**Wrapper** – A general term in programming which involves techniques to hide, adapt, or simplify features in a program or API by programming features that serves as a layer on top of the original features.
Appendix

```csharp
public void Awake()
{

    Yaw = RotationDirection
        .Select(dir => dir.x)
        .Scan(transform.eulerAngles.y, (acc, x) => acc + x);

    Pitch = RotationDirection
        .Select(dir => dir.y)
        .Scan(0f, (acc, y) => Mathf.Clamp(acc - y, -90, 90));

    IsGrounded = Observable.Merge(
        groundTrigger.OnTriggerEnterAsObservable()
            .Where(c => !c.isTrigger)
            .Select(_ => true),
        groundTrigger.OnTriggerExitAsObservable()
            .Where(c => !c.isTrigger)
            .Select(_ => false))
        .Scan(0, (n, b) => b == true ? n + 1 : n - 1)
        .Select(n => n > 0)
        .DistinctUntilChanged();

    MaxSpeed = IsGrounded.Select(b => b ? settings.groundSpeed : settings.airSpeed);

    MaxAcceleration = IsGrounded.Select(b => b ? settings.groundAcceleration : settings.airAcceleration);

    OnJump = Jump.WhereTrue()
        .WithLatestFrom(IsGrounded, (_, b) => b)
        .WhereTrue();

    var jumpVelocity = OnJump.Select(_ =>
        Mathf.Sqrt(
            2 * (-Physics.gravity.y) * settings.jumpHeight))
        .Select(yVel => new Vector3(
            rigidbody.velocity.x,
            yVel,
            rigidbody.velocity.z));

    var speedAcc = MaxSpeed.CombineLatest(MaxAcceleration, Tuple.Create); 

    var movementVelocity = Observable.EveryFixedUpdate()
        .WithLatestFrom(MovementDirection, (_, v) => v)
        .Select(v => v.y * transform.forward + v.x * transform.right)
        .WithLatestFrom(speedAcc,
            (v, t) => Vector3.Lerp( 
                rigidbody.velocity, 
                v * t.Item1, 
                Time.fixedDeltaTime * t.Item2))
        .Select(v => new Vector3(v.x, rigidbody.velocity.y, v.z));

    Velocity = Observable.Merge(movementVelocity, jumpVelocity);
}
```
public static class Util
{
    public static IObservable<float> MoveTowards01(float duration)
    {
        return Time
            .Select(t => t / duration)
            .TakeWhile(t => t < 1)
            .Concat(Observable.Return(1f));
    }

    public static IObservable<Vector3> PerlinShake(
        IObservable<float> frequency,
        IObservable<float> magnitude)
    {
        return Time
            .WithLatestFrom(frequency, (t, f) => t * f)
            .Select(t =>
                new Vector3(
                    Mathf.PerlinNoise(t, 0),
                    Mathf.PerlinNoise(t, t),
                    Mathf.PerlinNoise(0, t))
            .Select(v => v - Vector3.one / 2)
            .WithLatestFrom(magnitude, (v, m) => v * m);
    }

    public static IObservable<float> Time
    {
        get
        {
            return Observable.EveryUpdate()
                .Select(_ => UnityEngine.Time.deltaTime)
                .Scan(0f, (t, dt) => t + dt);
        }
    }
}