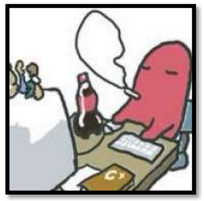
<Reactive Programming In Unity>

UniRx
시작하기



초중급 게임 개발자 스터디 데브루키
박민근(알콜코더)
@agebreak / agebreak@gmail.com
2016.4.9

들어가기에 앞서

이 발표 자료는 아래의 PT를 기반으로 번역, 제작 되었습니다

http://www.slideshare.net/torisoup/uni-rx

이 PT의 내용

UniRx를 어떻게 사용하면 좋을까를 소개하는 내용

타겟층

UniRx 초보자 ~ 중급자 대상

(사실상 중급자 대상)

목차

1. UniRx의 사용예
    - Update()를 없애기
    - 컴포넌트를 스트림으로 연결하기
        - Hot과 Cold에 관해서
    - uGUI와 조합해서 사용하기
    - 코루틴과 조합하기
2. 마지막으로

목차

1. UniRx의 사용예
   - Update()를 없애기
   - 컴포넌트를 스트림으로 연결하기
     - Hot과 Cold에 관해서
   - uGUI와 조합해서 사용하기
   - 코루틴과 조합하기
2. 마지막으로

Update를 없애기

Update()를 Observable로 변환해서

Awke()/Start()에 모아서 작성하기

Obervable화 하지 않은 구현

```
private void Update()
{

    //移動可能状態なら移動とジャンプ処理ができる
    if (canPlayerMove)
    {
        //移動処理
        var inputVector =
            (new Vector3(Input.GetAxis("Horizontal"), 0, Input.GetAxis("Vertical")));

        //スティックがしきい値を超えるだけ倒されていたら移動する
        if (inputVector.magnitude > 0.5f)
        {
            Move(inputVector.normalized);
        }

        //接地中にジャンプボタンが押されたらジャンプする
        if (isOnGrounded &&  Input.GetButtonDown("Jump"))
        {
            Jump();
        }
    }

    //残弾があるなら攻撃できる
    if (ammoCount > 0)
    {
        if (Input.GetButtonDown("Attack"))
        {
            Attack();
        }
    }
}
```

하고 싶은 것을 순차적으로 작성해야 해서, 흐름이 따라가기 힘들다

비교 Obervable

```
private void Start()
{
    //이동 가능할때에 이동키가 일정 이상 입력 받으면 이동
    this.UpdateAsObservable()
        .Where(_=>canPlayerMove)
        .Select(_ => new Vector3(Input.GetAxis("Horizontal"), 0, Input.GetAxis("Vertical")))
        .Where(input=>input.magnitude > 0.5f)
        .Subscribe(Move);

    //이동 가능하고, 지면에 있을때에 점프 버튼이 눌리면 점프
    this.UpdateAsObservable()
        .Where(_ =>canPlayerMove && isOnGrounded && Input.GetButtonDown("Jump"))
        .Subscribe(_ => Jump());

    // 총알이 있는 경우 공격 버튼이 눌리면 공격
    this.UpdateAsObservable()
        .Where(_ => ammoCount > 0 && Input.GetButtonDown("Attack"))
        .Subscribe(_ => Attack());
}
```

작업을 병렬로 작성할 수 있어서 읽기가 쉽다

Update를 없애기의 메리트

작업별로 병렬로 늘어서 작성하는것이 가능

- 작업별로 스코프가 명시적이게 된다
- 기능 추가, 제거, 변경이 용이해지게 된다
- 코드가 선언적이 되어서, 처리의 의도가 알기 쉽게 된다

Rx의 오퍼레이터가 로직 구현에 사용 가능

- 복잡한 로직을 오퍼레이터의 조합으로 구현 가능하다

Observable로 변경하는 3가지 방법

UpdateAsObaservable

- 지정한 gameObject에 연동된 Observable가 만들어진다.
- gameObject의 Destroy때에 OnCompleted가 발행된다

Observable.EveryUpdate

- gameObject로부터 독립된 Observable이 만들어 진다
- MonoBehaviour에 관계 없는 곳에서도 사용 가능

ObserveEveryValueChanged

- Observable.EveryUpdate의 파생 버전
- 값의 변화를 매프레임 감시하는 Observable이 생성된다

Observable.EveryUpdate의 주의점

Destroy때에 OnCompleted가 발생되지 않는다

- UpdateAsObaservable과 같은 느낌으로 사용하면 함정에 빠진다

```
[SerializeField] private Text text;
private void Start()
{
    //座標をUIに描画
    Observable.EveryUpdate()
        .Select(_ => transform.position)
        .SubscribeToText(text);
}
```

이 gameObject가 파괴되면

Null이 되어 예외가 발생한다

수명 관리의 간단한 방법

AddTo

- 특정 gameObject가 파괴되면 자동 Dispose 되게 해준다
- OnCompleted가 발행되는것은 아니다

```
[SerializeField] private Text text;
private void Start()
{
    //座標をUIに描画
    Observable.EveryUpdate()
        .Select(_ => transform.position)
        .SubscribeToText(text)
        .AddTo(this.gameObject);
}
```

AddTo로 넘겨진 gameObject가

Destroy되면 같이 Dispose 된다

목차

1. UniRx의 사용예
   - Update()를 없애기
   - **컴포넌트를 스트림으로 연결하기**
     - Hot과 Cold에 관해서
   - uGUI와 조합해서 사용하기
   - 코루틴과 조합하기
2. 마지막으로

스트림으로 연결

컴포넌트를 스트림으로 연결하는 것으로,

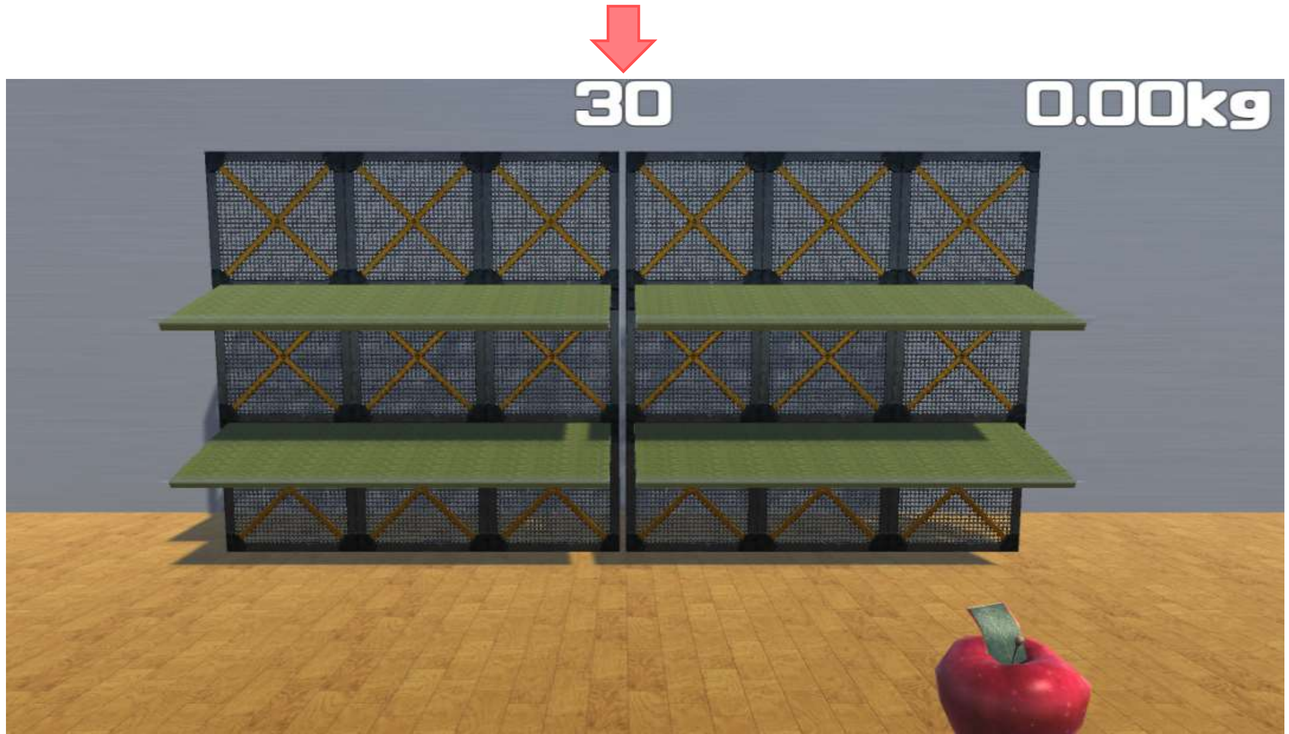
Observer패턴한 설계로 만들 수 있다

- 전체가 이벤트 기반이 되게 할 수 있다
- 더불어 Rx는 Observer패턴 그 자체

타이머 예제

타이머의 카운트를 화면에 표시 한다

- UniRx를 사용하지 않고 구현
- UniRx의 스트림으로 구현



타이머 예제

타이머의 카운트를 화면에 표시 한다

- UniRx를 사용하지 않고 구현
- UniRx의 스트림으로 구현

UniRxを使用しないで実装

```
public class TimerDisplayComponent : MonoBehaviour
{
    [SerializeField]
    private TimerComponent _timerComponent;
    private Text _timerText;

    void Start()
    {
        _timerText = GetComponent<Text>();
    }

    void Update()
    {
        //現在のカウントを取得
        var currentTimeText = _timerComponent.CurrentTime.ToString();

        //カウントが更新されていたら描画を書き換える
        if (currentTimeText!= _timerText.text)
        {
            _timerText.text = currentTimeText;
        }
    }
}
```

UniRxを使用しない場合

```
public class TimerDisplayComponent : MonoBehaviour
{
    [SerializeField]
    private TimerComponent _timerComponent;
    private Text _timerText;

    void Start()
    {
        _timerText = GetComponent<Text>();
    }

    void Update()
    {
        //現在のカウントを取得
        var currentTimeText = _timerComponent.CurrentTime.ToString();

        //カウントが更新されていたら描画を書き換える
        if (currentTimeText!= _timerText.text)
        {
            _timerText.text = currentTimeText;
        }
    }
}
```

매 프레임,

값이 변경되었는지를 확인 한다

(매 프레임 값을 체크해야만 한다)

타이머 예제

타이머의 카운트를 화면에 표시 한다

- UniRx를 사용하지 않고 구현
- UniRx의 스트림으로 구현

우선 타이머측을 스트림으로 변경

```
public class TimerComponent : MonoBehaviour
{

    private readonly ReactiveProperty<int> _timerReactiveProperty = new IntReactiveProperty(30);

    /// <summary>
    /// 現在のカウント
    /// </summary>
    public ReadOnlyReactiveProperty<int> CurrentTime
    {
        get { return _timerReactiveProperty.ToReadOnlyReactiveProperty(); }
    }

    private void Start()
    {
        //1초마다 timerReactiveProperty 값을 마이너스 한다
        Observable.Timer(TimeSpan.FromSeconds(1))
            .Subscribe(_ => _timerReactiveProperty.Value--)
            .AddTo(gameObject); // 게임오브젝트 파괴시에 자동 정지 시킨다
    }
}
```

우선 타이머측을 스트림으로 변경

```
public class TimerComponent : MonoBehaviour
{

    private readonly ReactiveProperty<int> _timerReactiveProperty = new IntReactiveProperty(30);

    /// <summary>
    /// 現在のカウント
    /// </summary>
    public ReadOnlyReactiveProperty<int> CurrentTime
    {
        get { return _timerReactiveProperty.ToReadOnlyReactiveProperty(); }
    }

    private void Start()
    {
        //1초마다 timerReactiveProperty 값을 마이너스 한다
        Observable.Timer(TimeSpan.FromSeconds(1))
            .Subscribe(_ => _timerReactiveProperty.Value--)
            .AddTo(gameObject); // 게임오브젝트 파괴시에 자동 정지 시킨다
    }
}
```

CurrentTime을 ReactiveProperty로 공개한다

(값이 변경되면, OnNext로 그 값이 통지 된다)

타이머를 사용하는 측의 구현

```
public class TimerDisplayComponent : MonoBehaviour
{
    [SerializeField]
    private TimerComponent _timerComponent;
    private Text _timerText;

    void Start()
    {
        _timerComponent.CurrentTime
            .SubscribeToText(_timerText);
    }
}
```

타이머를 사용하는 측의 구현

```
public class TimerDisplayComponent : MonoBehaviour
{
    [SerializeField]
    private TimerComponent _timerComponent;
    private Text _timerText;

    void Start()
    {
        _timerComponent.CurrentTime
            .SubscribeToText(_timerText);
    }
}
```

타이머에서 값 갱신 통지가 오면,

그 타이밍에 화면을 변경하는것 뿐

스트림으로 연결하는 메리트

Observer 패턴이 간단히 구현 가능하다

- 변화를 폴링(Polling)하는 구현이 사라진다
- 필요한 타이밍에 필요한 처리를 하는 방식으로 작성하는 것이 좋다

기존의 이벤트 통지구조보다 간단

- C#의 Event는 준비단계가 귀찮아서 쓰고 싶지 않다
- Unity의 SendMessage는 쓰고 싶지 않다
- Rx라면 Observable를 준비하면 OK! 간단!

목차

1. UniRx의 사용예
   - Update()를 없애기
   - 컴포넌트를 스트림으로 연결하기
     - Hot과 Cold에 관해서
   - uGUI와 조합해서 사용하기
   - 코루틴과 조합하기
2. 마지막으로

Hot / Cold 한 성질

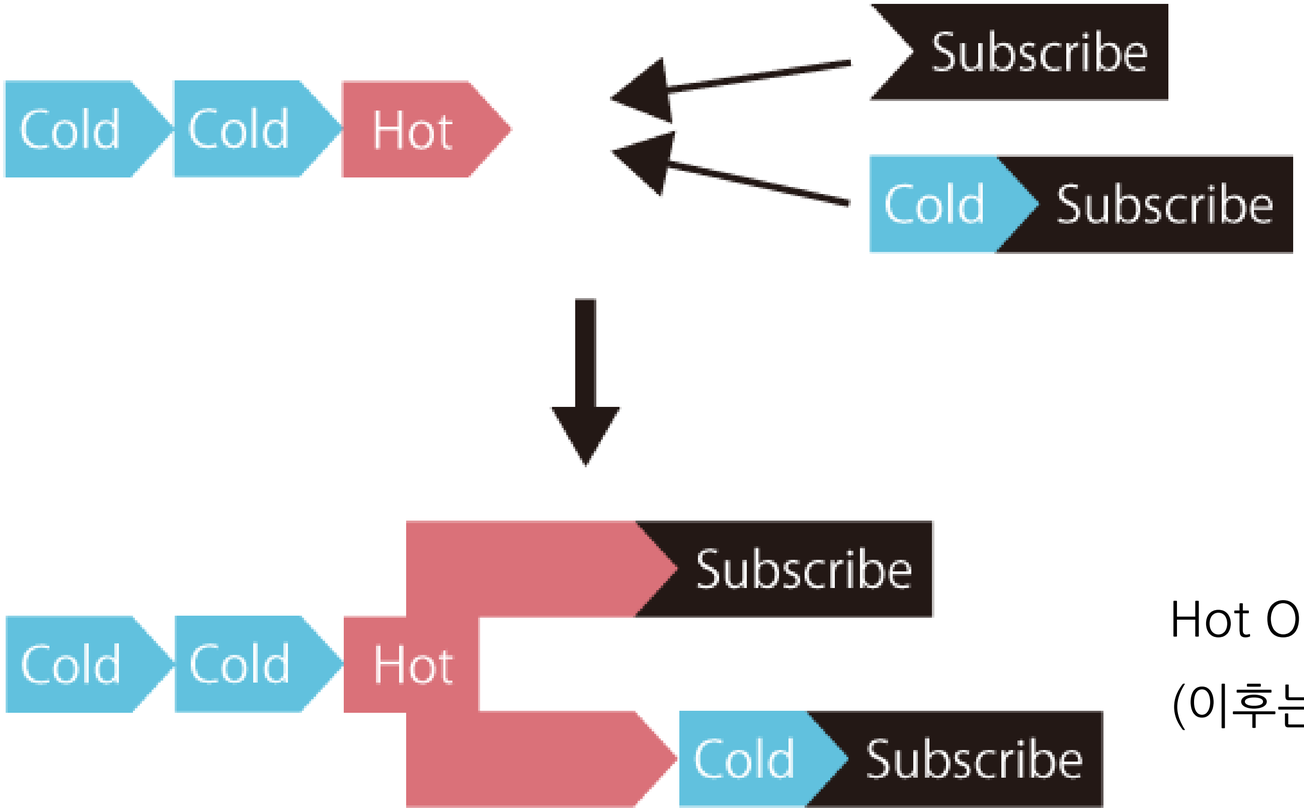
Observable은 성질에 따라 2종류로 구별 된다

- Hot한 성질인 것
- Cold한 성질인 것

Hot한 성질의 Observable

Observer가 없어지더라도 동작 한다

스트림을 분기 시키거나, 메시지를 분배하는것이 가능 하다



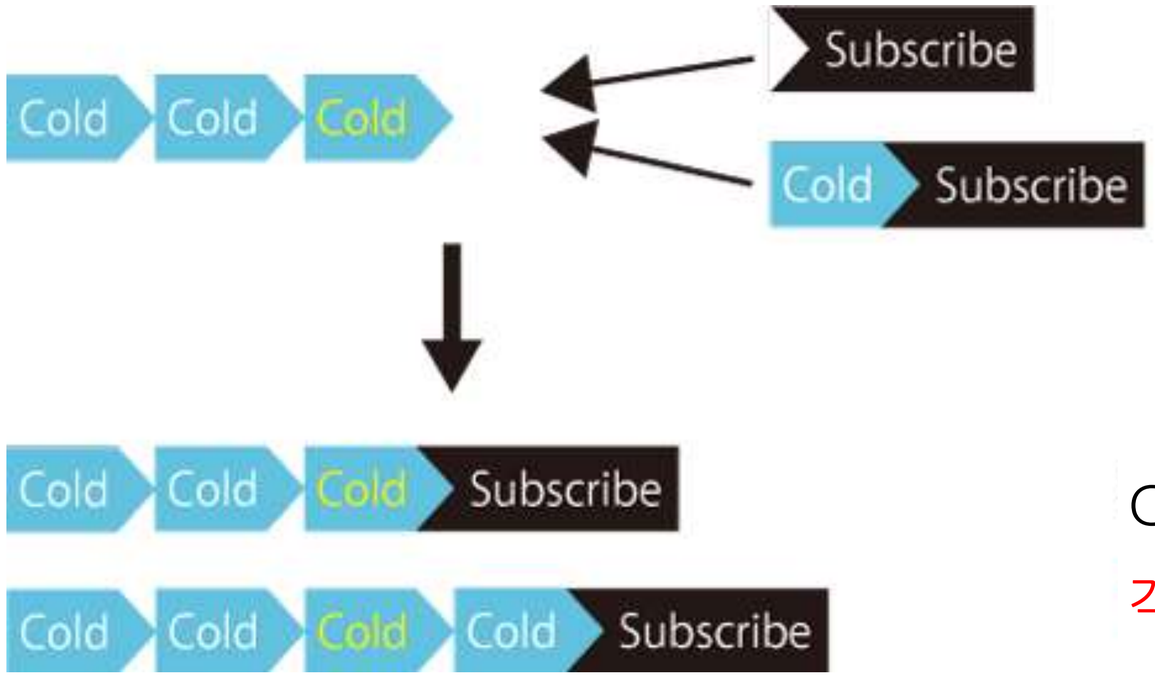
Hot Observable은 스트림의 분기점이 된다

(이후는 동일한 값을 전달한다)

Cold한 성질의 Observable

Observer가 없어지면 동작하지 않는다

Subscribe 될때마다 새롭게 생성 된다 (분기 되지 않음)



Cold Observable을 여러번 Subscribe 하면,

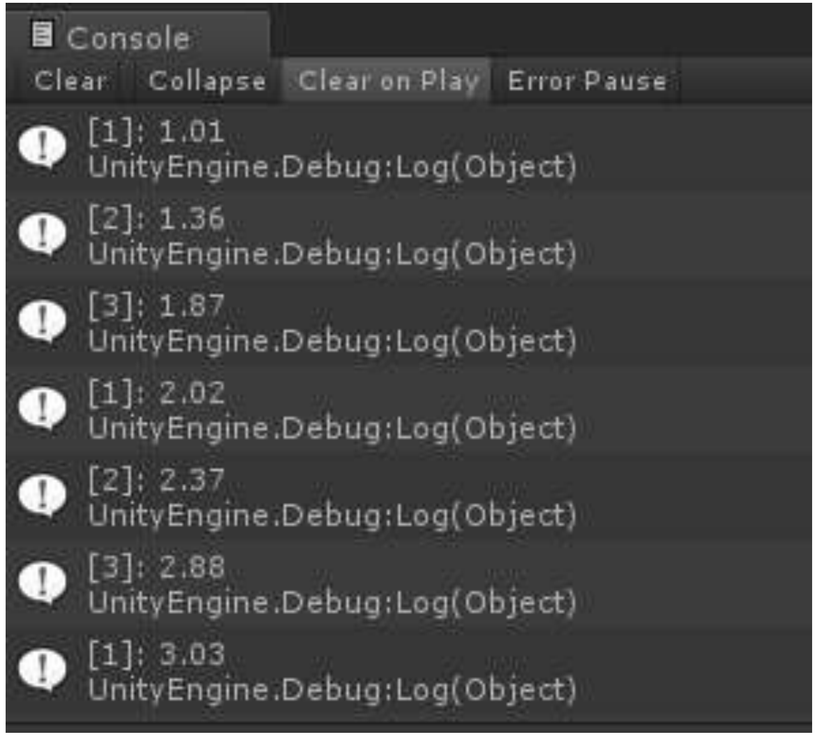
각각 별도의 스트림이 생성 된다

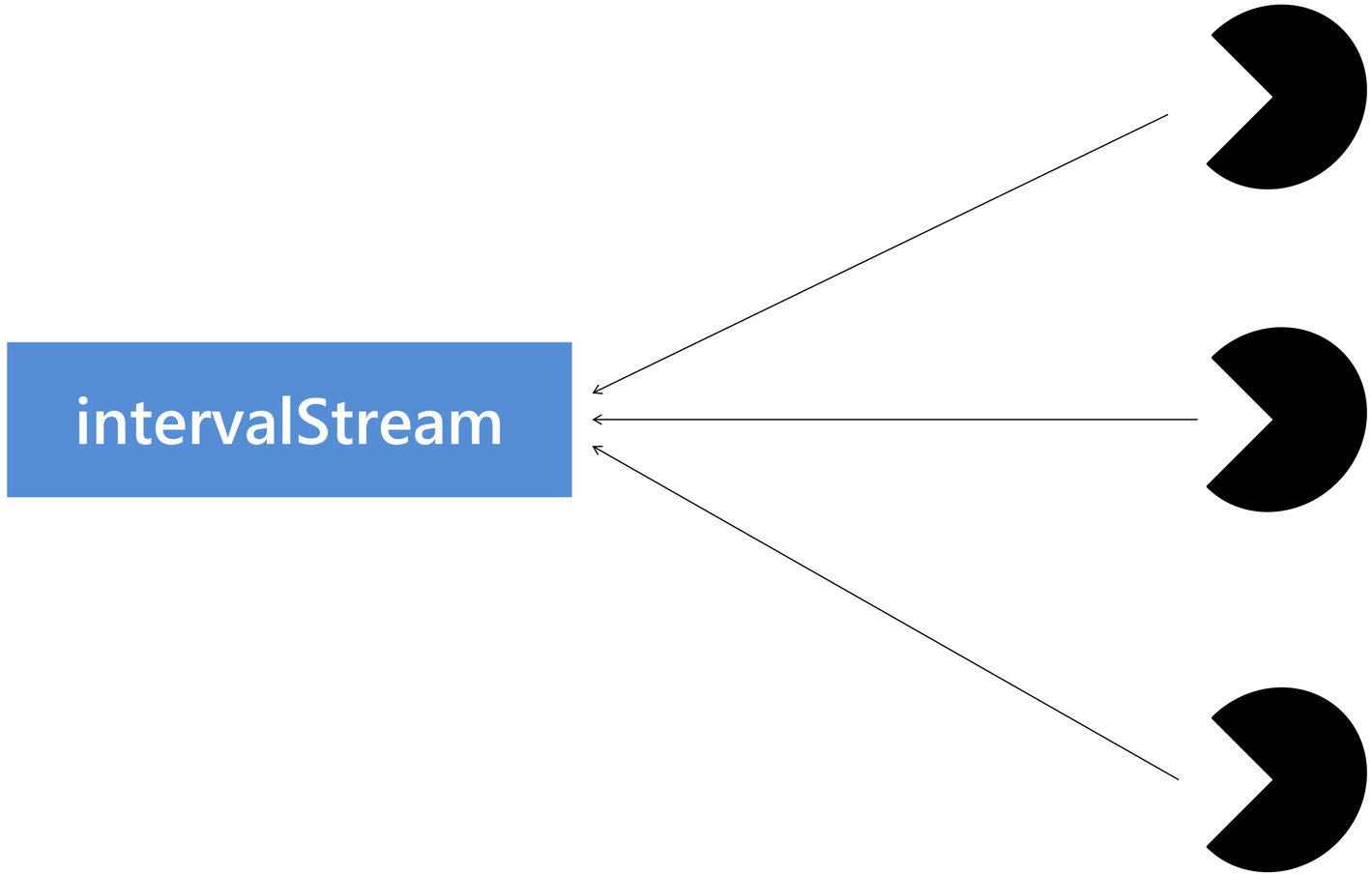
예) Cold Observable의 여러 번 Subscribe

IntervalStream을 시간을 다르게 해서 3회 Subscribe

- 같은 스트림이지만 OnNext의 타이밍이 각자 다르다
- 각각 별도의 스트림이 생성되게 하기 위해

```
var intervalStream = Observable.Interval(TimeSpan.FromSeconds(1));

intervalStream
    .Subscribe(x =>Debug.Log("[1]: " + Time.time.ToString("F2")));

yield return new WaitForSeconds(0.1f);

intervalStream
    .Subscribe(x => Debug.Log("[2]: " + Time.time.ToString("F2")));

yield return new WaitForSeconds(0.5f);

intervalStream
    .Subscribe(x => Debug.Log("[3]: " + Time.time.ToString("F2")));
```



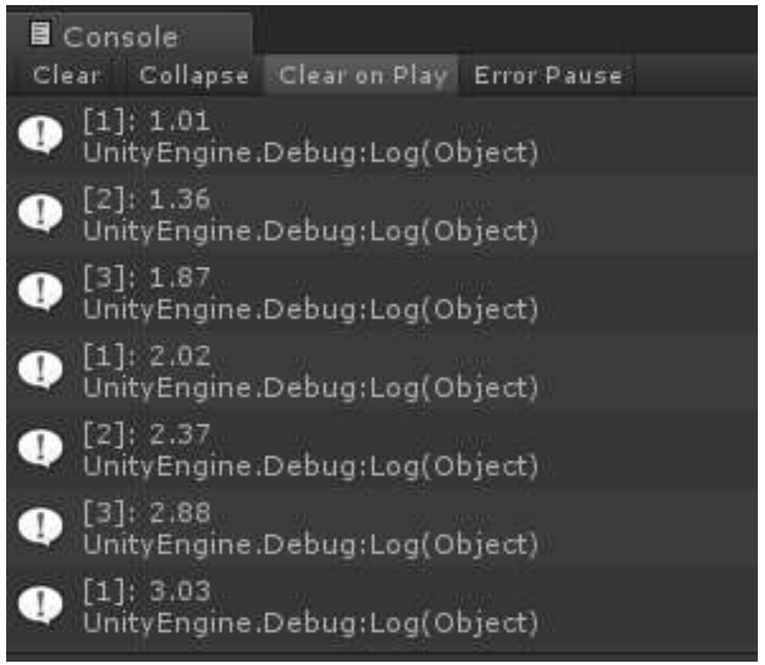


Cold한 성질의 InservalStream을 여러 번 Subscribe

intervalStream

intervalStream

intervalStream



3개다 별개의 스트림이 생성되어 버린다

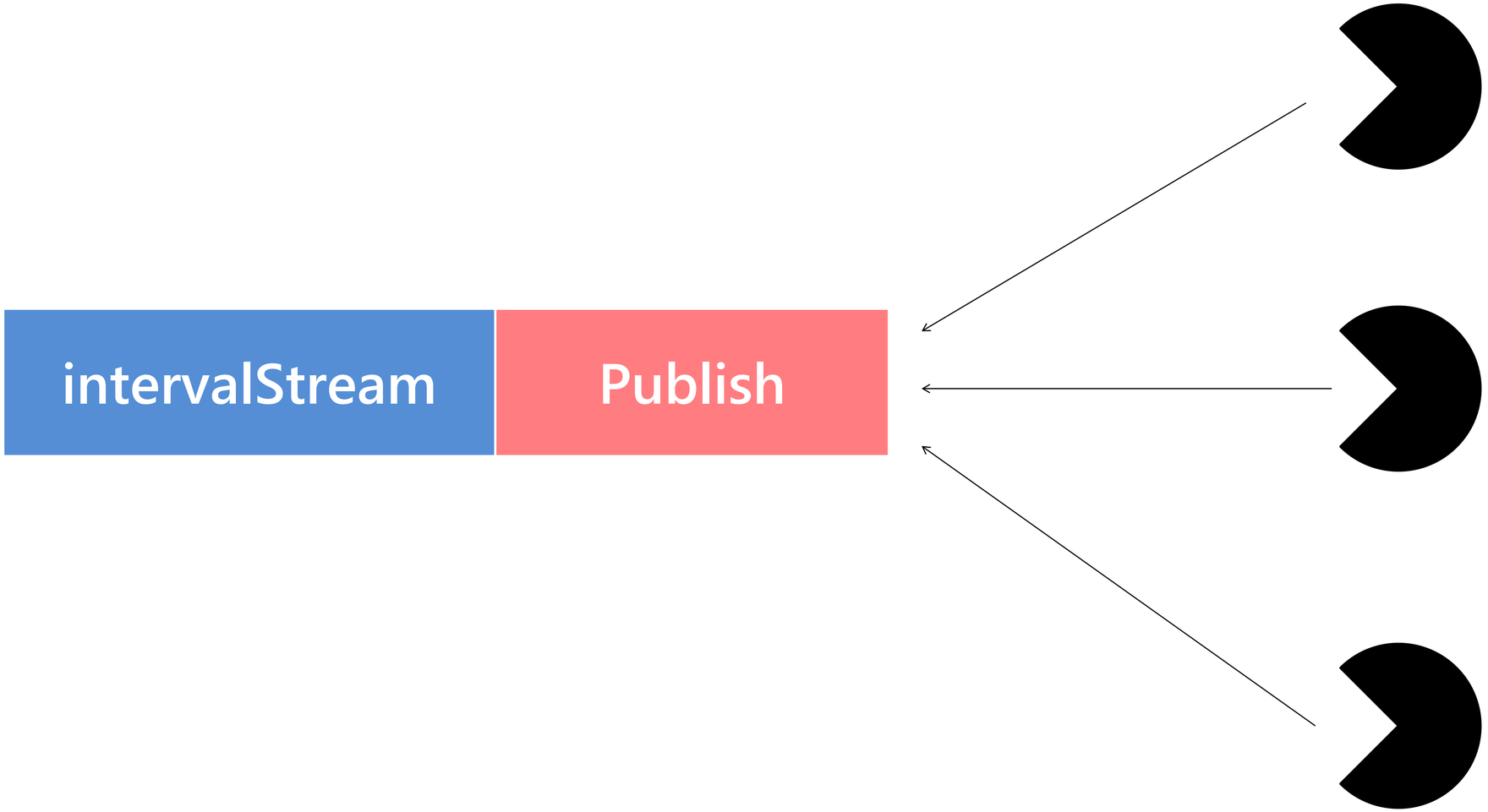
(각각 별도의 타이머를 가지게 있기 때문에 시간이 다 다르다)

Hot으로 변환해 보자

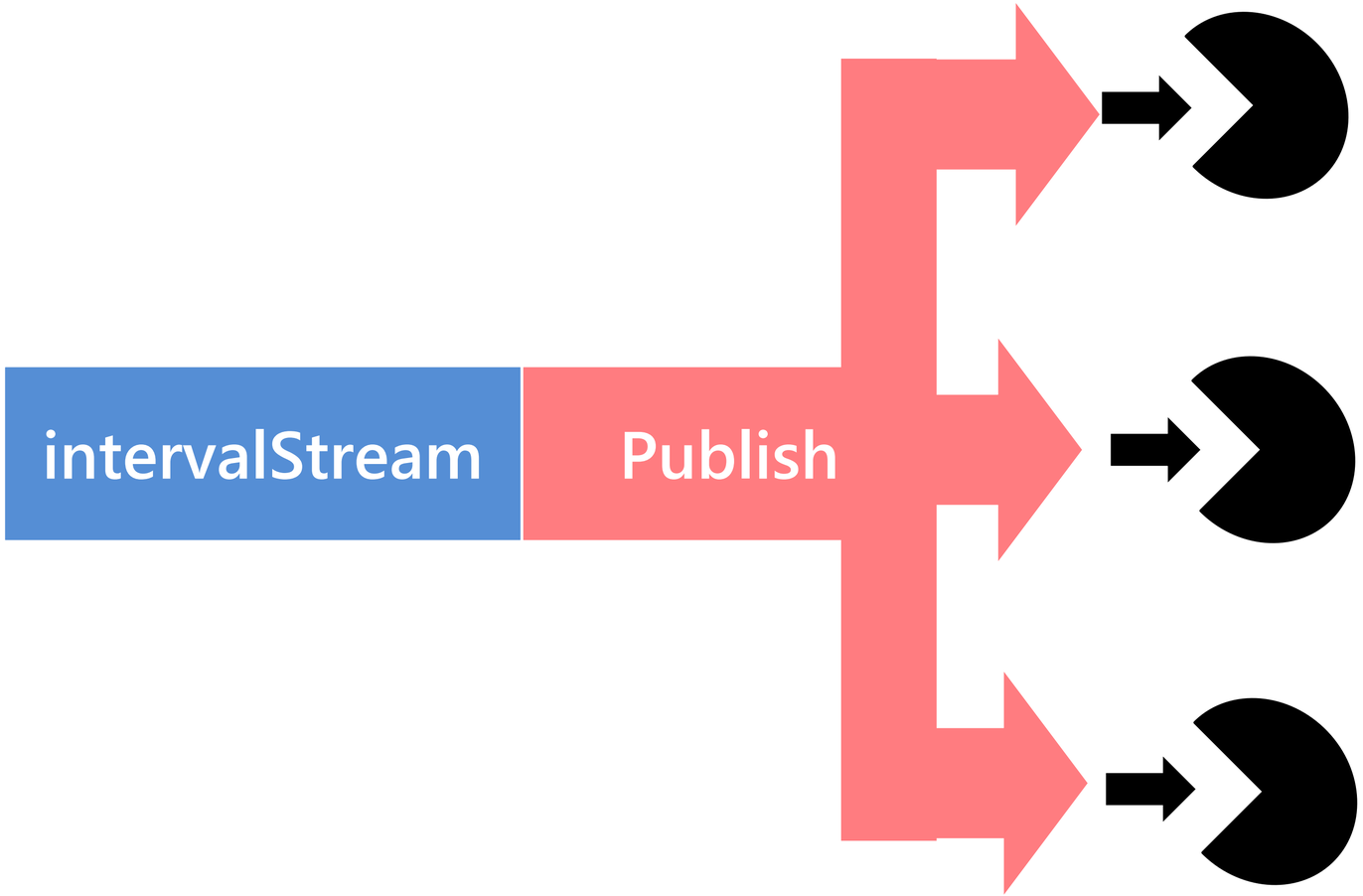
Obervable은 Hot한 성질로 변경이 가능하다

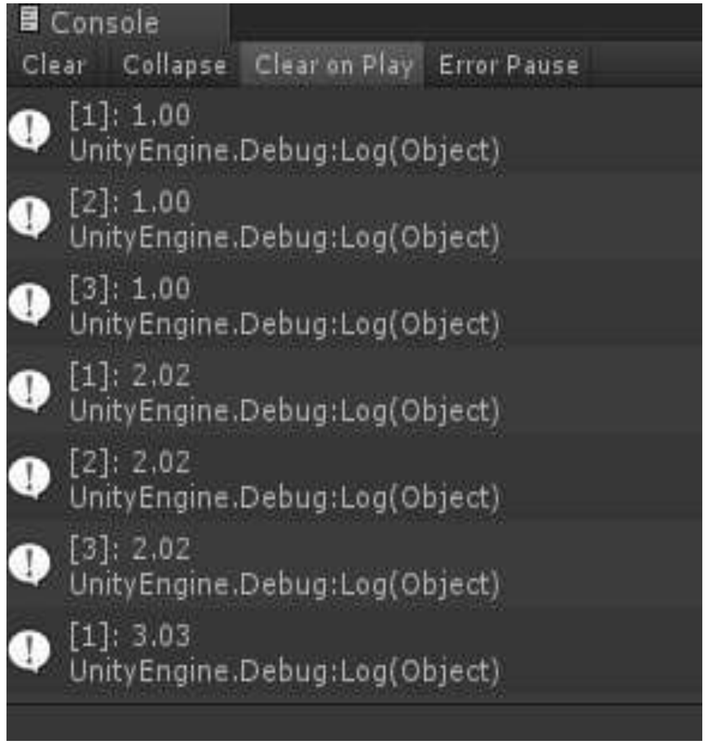
- Hot 변경 오퍼레이터를 추가하는 것으로 Hot으로 하는 것이 가능하다
- 앞의 IntervalStream을 Hot으로 변경해 보자

```
var intervalStream =
    Observable.Interval(TimeSpan.FromSeconds(1)).Publish().RefCount();

intervalStream
    .Subscribe(x => Debug.Log("[1]: " + Time.time.ToString("F2")));

yield return new WaitForSeconds(0.1f);

intervalStream
    .Subscribe(x => Debug.Log("[2]: " + Time.time.ToString("F2")));

yield return new WaitForSeconds(0.5f);

intervalStream
    .Subscribe(x => Debug.Log("[3]: " + Time.time.ToString("F2")));
```



Hot변경 오퍼레이터를 끝에 배치해서 Subscribe





OnNext의 타이밍이 모두 일치

Hot변경한 부분이 분기 된다

(1개의 타이머를 3개의 Observer가 공유하는 것이 가능하다)

Hot / Cold 정리

Rx의 빠지기 쉬운 함정 하나

- 형태로부터 구별할 수 없기 때문에 상당히 번거롭다
- 스트림이 Hot인가 Cold인가를 신경쓸 필요가 있다
- 스트림을 외부에 공개할 때는 Hot변경을 끼워 넣으면 안전

대부분의 스트림은 Cold이다

- 오퍼레이터는 거의 전부가 Cold한 성질
- Subject를 내부에 가진것이 유일한 Hot 성질이다
  * Subject, ReactiveProperty, Hot 변경용 오퍼레이터등

좀더 자세한 내용은…

아래 링크의 내용들을 참조 (일본어)

http://qiita.com/toRisouP/items/f6088963037bfda658d3
http://qiita.com/toRisouP/items/c955e36610134c05c860

목차

1. UniRx의 사용예
   - Update()를 없애기
   - 컴포넌트를 스트림으로 연결하기
     - Hot과 Cold에 관해서
   - **uGUI와 조합해서 사용하기**
   - 코루틴과 조합하기
2. 마지막으로

UniRx와 uGUI를 조합하기

uGUI에서 사용할 수 있는 Model-View-00 패턴

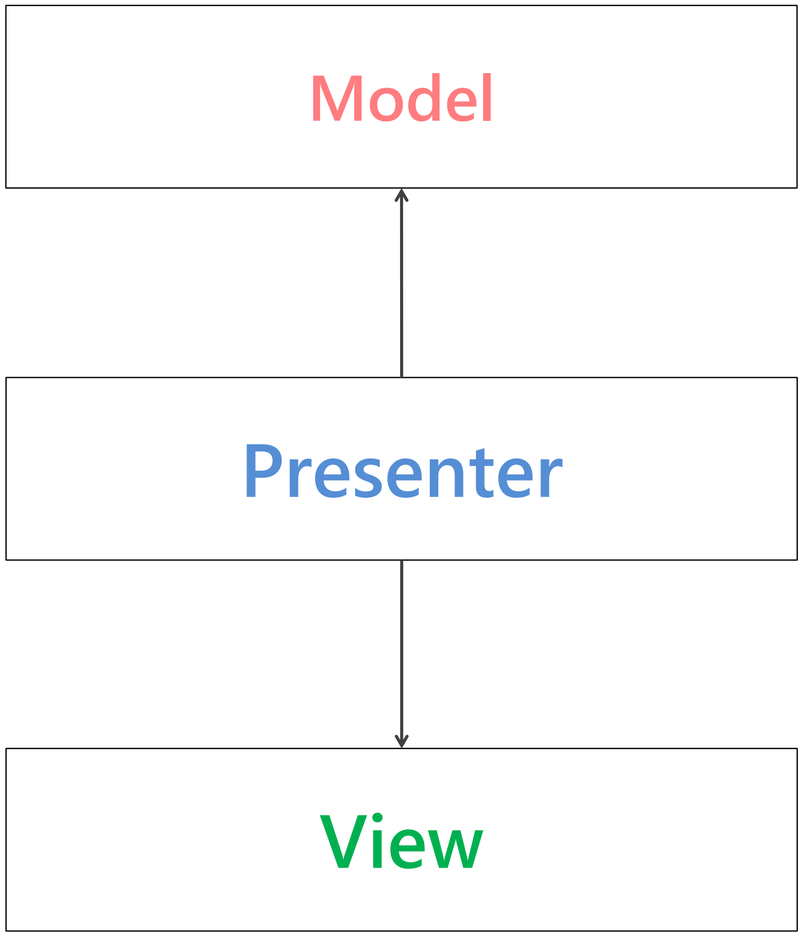
- uGUI에 유용한 MVO패턴이 지금까지 존재하지 않았다
  * MVC 패턴은 원래 사람에 따라서 개념이 각각 너무 다름
  * MVVM은 데이터바인딩이 없기 때문에 사용할 수 없다
- Obervable과 ReactiveProperty를 조합하게 되면, uGUI 관련을 깔끔하게 작성가능

Model-View-(Reactive)Presenter

MV(R)P 패턴

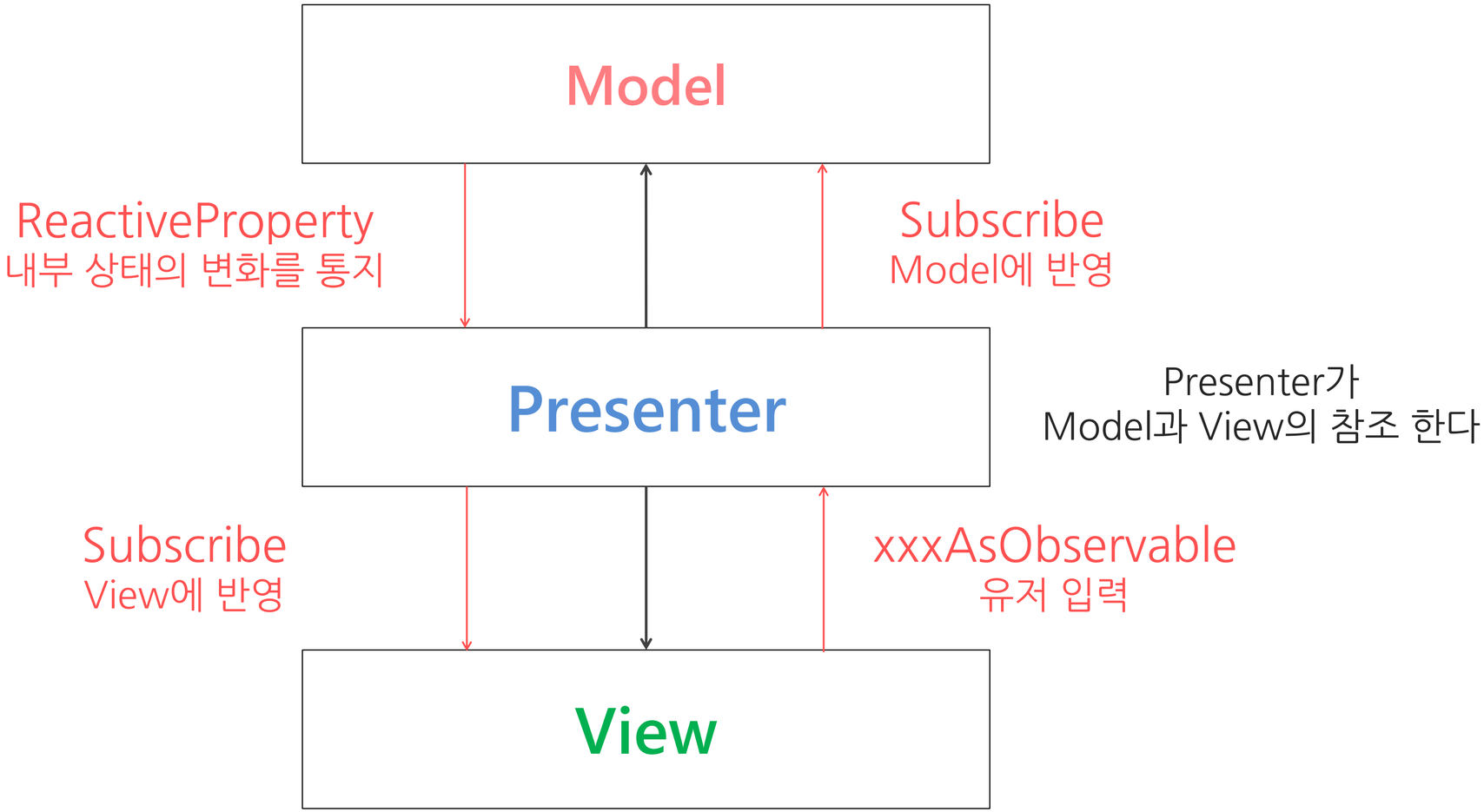
- Model - View - Presenter 패턴 + UniRx
- 3개의 레이어를 Observable로 심리스하게 연결

MV(R)P의 구조



Presenter가
Model과 View를 참조한다

MV(R)P의 구조


Model
ReactiveProperty
내부 상태의 변화를 통지
Subscribe
Model에 반영
Presenter
Presenter가
Model과 View의 참조 한다
Subscribe
View에 반영
xxxAsObservable
유저 입력
View

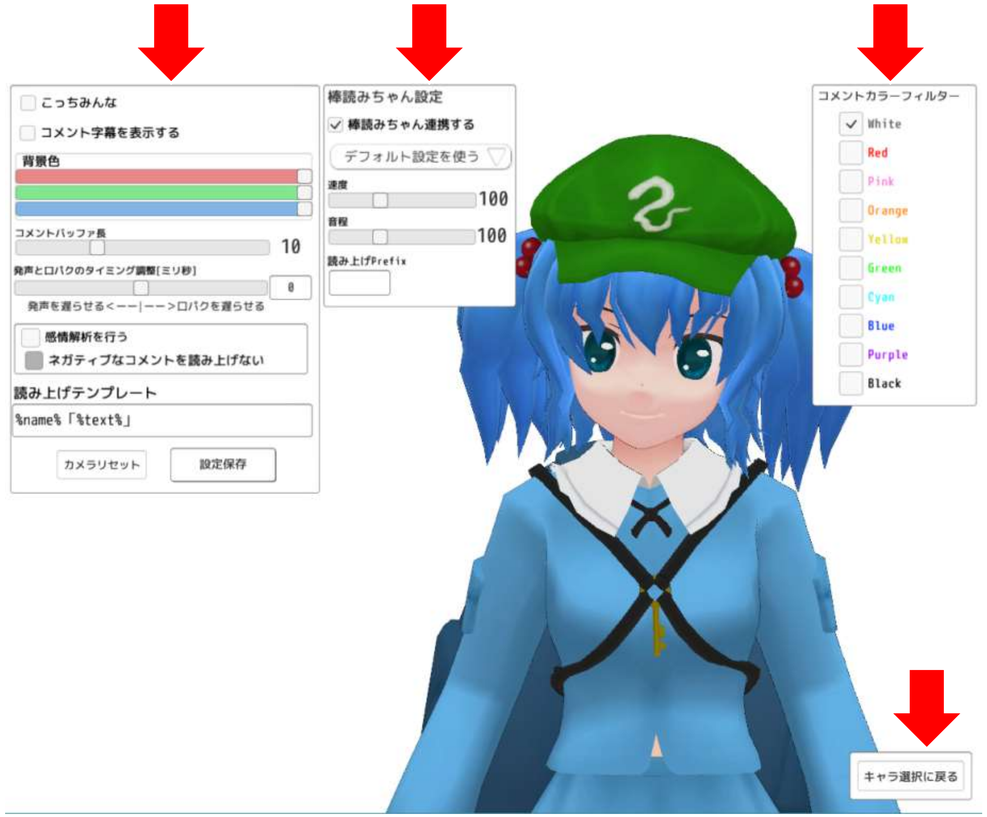
MV(R)P패턴 만드는 법

1. Model에 ReactiveProperty를 가지게 한다
2. Presenter을 만든다
3. Presenter에 Model과 View를 등록 한다
4. Presenter내에서 View의 Observable과 Model의 ReactiveProperty를
   각각 Subscribe해서 연결한다

사용예) 미쿠미쿠 마우스

MMD 모델에 니코생방송의 코멘트를 읽게 하는 툴

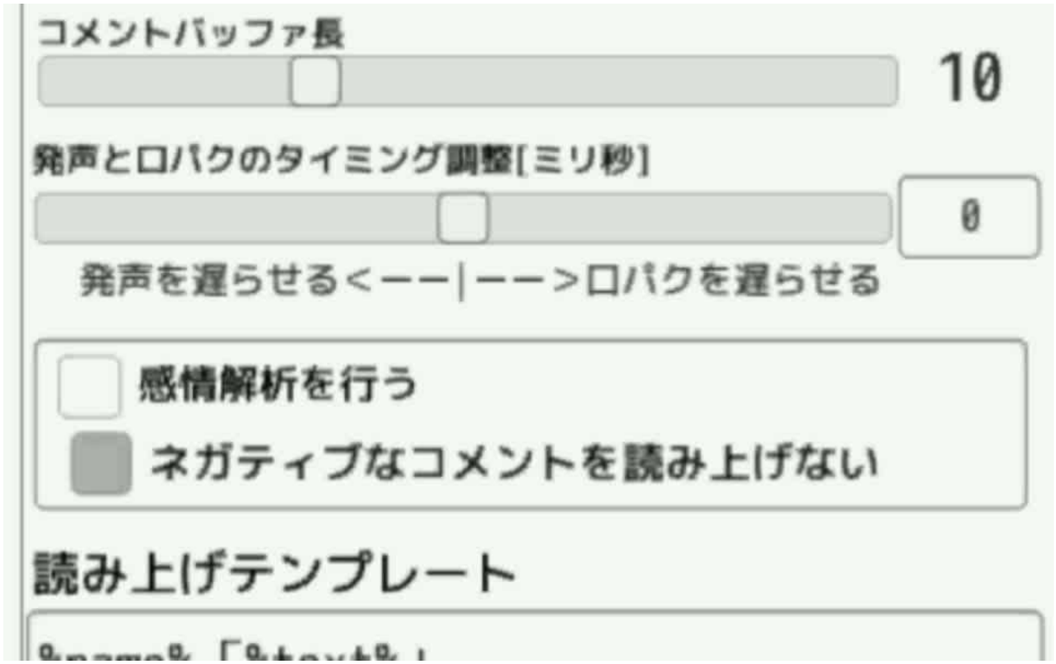
UI를 MV(R)P 패턴으로 구현하였다



MV(R)P 구현예제 소개

코멘트를 읽어들이는 타이밍의 조절 슬라이더

- Slider와 InputField가 연동한다
- 입력된 값은 ConfigComponent가 유지 한다



컴포넌트간의 관계도

Model

SpeechTimingReactiveProperty
(읽어들이는 타이밍 조절 시간의 실제값)

ConfigComponent

Presenter

SpeechTimingSlider
Presenter

SpeechTimingInputField
Presenter

View

OnValueChangedAsObservable()
슬라이더가 변화한 순간에 그 값을
통지하는 Observable

OnEndEditAsObservable()
InputField의 입력이 완료된 순간에
그 값을 통지하는 Observable

Modelの구현

```
/// <summary>
/// 設定を管理するコンポーネント
/// </summary>
public class ConfigComponent : SingletonMonoBehaviour<ConfigComponent>
{
    /// <summary>
    //읽어들이는 타이밍(ms)
    /// </summary>
    public ReactiveProperty<int> SpeechTimingReactiveProperty
        = new IntReactiveProperty(0);

    //以下他のReactivePropertyの定義等が続く
```

ReactiveProperty를 외부에 공개

Presenter의 구현 (Slider 측)

```
using UnityEngine;
using UniRx;
using UnityEngine.UI;

public class SpeechTimingSliderPresenter : MonoBehaviour
{
    void Start()
    {
        var slider = GetComponent<Slider>();
        var config = ConfigComponent.Instance;

        //Model->Slider 반영
        config.SpeechTimingReactiveProperty
            .Subscribe(x => slider.value = x / 10);

        //Slider->Model 반영
        slider
            .OnValueChangedAsObservable()
            .DistinctUntilChanged()
            .Subscribe(x => config.SpeechTimingReactiveProperty.Value = (int)(x * 10));
    }
}
```

Model → View

View → Model

Presenter의 구현 (Input 측)

```
using System;
using UnityEngine;
using UniRx;
using UnityEngine.UI;

public class SpeechTimingInputField : MonoBehaviour
{
    public void Start()
    {
        var config = ConfigComponent.Instance;
        var inputField = GetComponent<InputField>();

        //View → Model
        inputField.OnEndEditAsObservable()
            .Select(x => Int32.Parse(x))
            .Select(x => Mathf.Clamp(x, -1500, 1500))
            .Subscribe(x => config.SpeechTiming = x);

        //Model→View
        config
            .SpeechTimingObservable
            .Select(x => x.ToString())
            .Subscribe(x => inputField.text = x);
    }
}
```

View → Model

Model → View

uGUI와 조합하기 정리

MV(R)P 패턴으로 uGUI 관련 설계가 편해진다

- uGUI를 사용할 경우에 아마도 현시점에서의 최적 해결법
- 프로그래머에게는 다루기 쉽지만,
  비프로그래머에게는 다루기 어려워질 가능성이 있으니 주의

Presenter의 베스트한 작성 방법에 대해서는 아직 모색중

- Presenter에 어떻게 Model과 View를 등록할 것인가
- Presenter을 하나로 모을 것인가, 분할해서 만들 것인가
- 동적으로 Presenter를 생성하는 경우에는 어떻게 할 것인가

목차

1. UniRx의 사용예
   - Update()를 없애기
   - 컴포넌트를 스트림으로 연결하기
     - Hot과 Cold에 관해서
   - uGUI와 조합해서 사용하기
   - **코루틴과 조합하기**
2. 마지막으로

코루틴과 조합하기

Unity의 코루틴과 Obervable은 상호교환가능

- 코루틴과 조합함으로서 Rx의 약점을 보완하는 것이 가능하다

코루틴 <-> Observable

코루틴 -> Observable

- Observable.FromCoroutine

Observable -> 코루틴

- StartAsCoroutine

코루틴 <-> Observable

코루틴 -> Observable

- Observable.FromCoroutine

Observable -> 코루틴

- StartAsCoroutine

코루틴->Observable의 메리트

복잡한 스트림을 간단하게 작성 가능

- 팩토리 메소드나 오퍼레이터 체인 만으로는 작성할 수 없는 복잡한 로직의 스트림을 작성하는 것이 가능하다

연속적인 처리를 캡슐화 가능하다

- 복잡한 처리를 코루틴에 숨겨서 외부에서는 Observable로서 선언적으로 취급하는 것이 가능하다

예제) 코루틴으로부터 Observable을 만들기

플레이어의 생사에 연동된 타이머

- 플레이어가 살아있을 때만 카운트다운
- 플레이어가 사망했을 때는 타이머를 정지한다
- 팩토리메소드나 오퍼레이터 체인으로 만드는것은 복잡하다

타이머의 정의

```
private void Start()
{
    //타이머의 Subscribe
    Observable.FromCoroutine<int>(observer => CountDownCoroutine(observer, 30, player))
        .Subscribe(count => Debug.Log(count));
}

/// <summary>
//플레이어가 살아있을때만 카운트다운하는 타이머
// 플레이어가 죽어있는 때는 카운트는 정지한다
IEnumerator CountDownCoroutine(IObserver<int> observer,int startTime,Player player)
{
    var currentTime = startTime;
    while (currentTime > 0)
    {
        if (player.IsAlive)
        {
            observer.OnNext(currentTime--);
        }
        yield return new WaitForSeconds(1);
    }
    observer.OnNext(0);
    observer.OnCompleted();
}
```

코루틴에서 변환 한다

```
private void Start()
{
    //타이머의 Subscribe
    Observable.FromCoroutine<int>(observer => CountDownCoroutine(observer, 30, player))
        .Subscribe(count => Debug.Log(count));
}

/// <summary>
//플레이어가 살아있을때만 카운트다운하는 타이머
// 플레이어가 죽어있는 때는 카운트는 정지한다
IEnumerator CountDownCoroutine(IObserver<int> observer,int startTime,Player player)
{
    var currentTime = startTime;
    while (currentTime > 0)
    {
        if (player.IsAlive)
        {
            observer.OnNext(currentTime--);
        }
        yield return new WaitForSeconds(1);
    }
    observer.OnNext(0);
    observer.OnCompleted();
}
```

코루틴안에서 OnNext

```
private void Start()
{
    //타이머의 Subscribe
    Observable.FromCoroutine<int>(observer => CountDownCoroutine(observer, 30, player))
        .Subscribe(count => Debug.Log(count));
}

/// <summary>
//플레이어가 살아있을때만 카운트다운하는 타이머
// 플레이어가 죽어있는 때는 카운트는 정지한다
IEnumerator CountDownCoroutine(IObserver<int> observer,int startTime,Player player)
{
    var currentTime = startTime;
    while (currentTime > 0)
    {
        if (player.IsAlive)
        {
            observer.OnNext(currentTime--);
        }
        yield return new WaitForSeconds(1);
    }
    observer.OnNext(0);
    observer.OnCompleted();
}
```

코루틴 <-> Observable

코루틴 -> Observable

- Observable.FromCoroutine

Observable -> 코루틴

- StartAsCoroutine

Observable을 코루틴으로 변경한다

StartAsCoroutine

- OnCompleted가 발생되기 전에 yield return null을 계속 한다
- 완료시에 최후의 OnNext값을 하나 출력 한다

```
var v = default(int);
yield return Observable.Range(1, 10).StartAsCoroutine(x => v = x);
Debug.Log(v); //10
```

비동기 처리를 동기처리처럼 쓸 수 있게 된다

- Task의 await와 비슷한 일을 Unity 코루틴으로 실현 가능

구현예

텍스트를 Web으로부터 다운로드

버튼으로 텍스트를 넘기면서 표시한다

구현한 코루틴

```
private void Start()
{
    StartCoroutine(ShowTextCoroutine());
}

private IEnumerator ShowTextCoroutine()
{
    _text.text = "ダウンロード開始...\n";

    IEnumerator<string> textIterator = null;

    //텍스트를 다운로드해서 변환한다
    yield return ObservableWWW.Get(url)
        .Select(result => TextParser(url))
        .StartAsCoroutine(iterator => textIterator = iterator);

    _text.text += "ダウンロード完了...\n\n";

    while (textIterator.MoveNext())
    {
        _text.text += textIterator.Current;

        //버튼 클릭이 올때까지 대기 한다
        yield return _nextButton.OnClickAsObservable().FirstOrDefault().StartAsCoroutine();
    }
}
```

텍스트의 다운로드를 기다림

```
private void Start()
{
    StartCoroutine(ShowTextCoroutine());
}

private IEnumerator ShowTextCoroutine()
{
    _text.text = "ダウンロード開始...\n";

    IEnumerator<string> textIterator = null;

    //텍스트를 다운로드해서 변환한다
    yield return ObservableWWW.Get(url)
        .Select(result => TextParser(url))
        .StartAsCoroutine(iterator => textIterator = iterator);

    _text.text += "ダウンロード完了...\n\n";

    while (textIterator.MoveNext())
    {
        _text.text += textIterator.Current;

        //버튼 클릭이 올때까지 대기 한다
        yield return _nextButton.OnClickAsObservable().FirstOrDefault().StartAsCoroutine();
    }
}
```

버튼이 눌릴때까지 기다림

```
private void Start()
{
    StartCoroutine(ShowTextCoroutine());
}

private IEnumerator ShowTextCoroutine()
{
    _text.text = "ダウンロード開始...\n";

    IEnumerator<string> textIterator = null;

    //텍스트를 다운로드해서 변환한다
    yield return ObservableWWW.Get(url)
        .Select(result => TextParser(url))
        .StartAsCoroutine(iterator => textIterator = iterator);

    _text.text += "ダウンロード完了...\n\n";

    while (textIterator.MoveNext())
    {
        _text.text += textIterator.Current;

      //버튼 클릭이 올때까지 대기 한다
        yield return _nextButton.OnClickAsObservable().FirstOrDefault().StartAsCoroutine();
    }
}
```

코루틴과 조합하기 정리

Unity의 코루틴과 Obervable은 상호교환가능

- 복잡한 스트림을 코루틴을 사용해서 구현 가능
- 코루틴 안에서 이벤트를 기다리는 것이 가능
- **비동기처리를 동기처리처럼 작성하는 것이 가능하게 된다**

목차

1. UniRx의 사용예
   - Update()를 없애기
   - 컴포넌트를 스트림으로 연결하기
     - Hot과 Cold에 관해서
   - uGUI와 조합해서 사용하기
   - 코루틴과 조합하기
2. 마지막으로

UniRx를 쓸때의 마음가짐

[모든것은 스트림] 이라는 세계관을 가지자

- 스트림화 가능한 곳을 찾아서 스트림으로 바꾸면 편리해진다
- 다만 과용하는 것은 금물

Rx는 만능이 아니라는 것을 알자

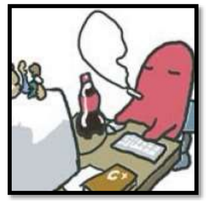
- 하고 싶은 것이 선언적으로 잘되지 않는 부분도 당연히 있다
- 그런 경우에는 코루틴과 조합해서 사용하면 유용하다

설계를 의식하자

- 무조건 쓰고보자라는 설계로 UniRx를 사용하는것은 상당히 위험
- Observer패턴의 장단점을 활용하자

진짜로 끝!

감사합니다.



박민근(알콜코더)
@agebreak / agebreak@gmail.com

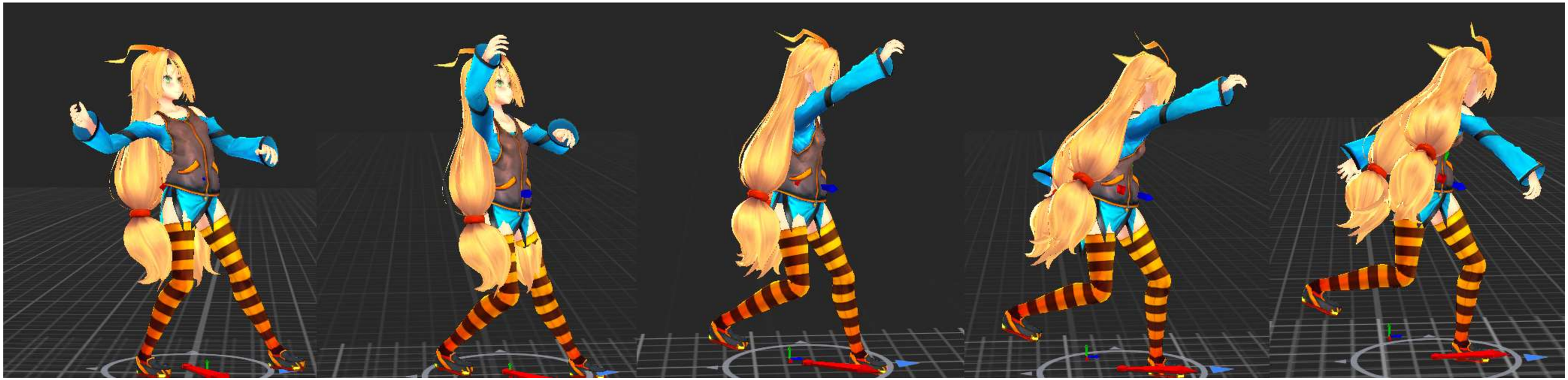
보너스

예제) Animation 동기화를 깔끔하게 만들기

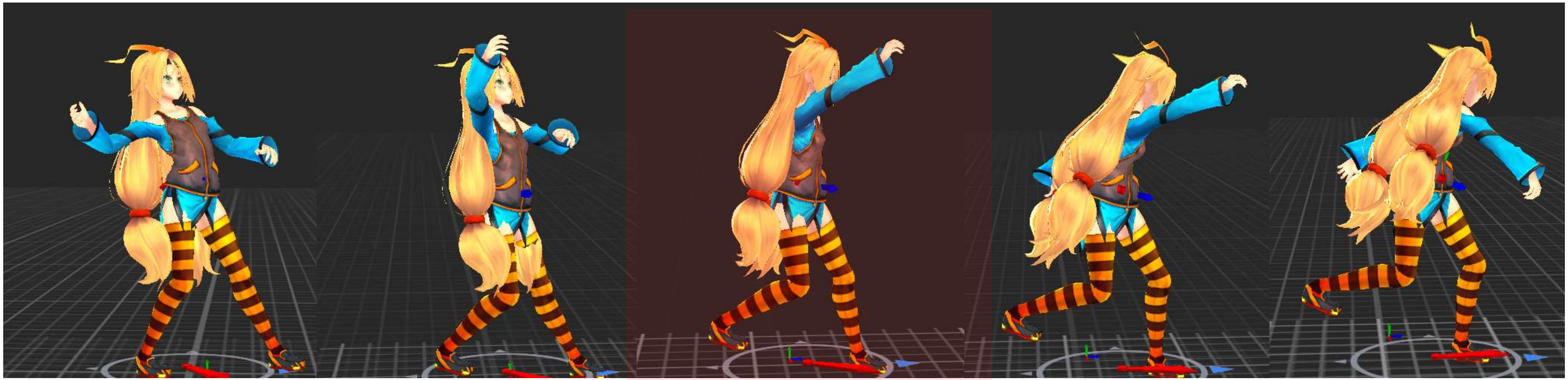
유니티짱이 공을 던진다

- 애니메이션에 동기화 시켜서 던진다
- 던지는 공의 파라미터도 지정 가능하다

공을 던지는 애니메이션



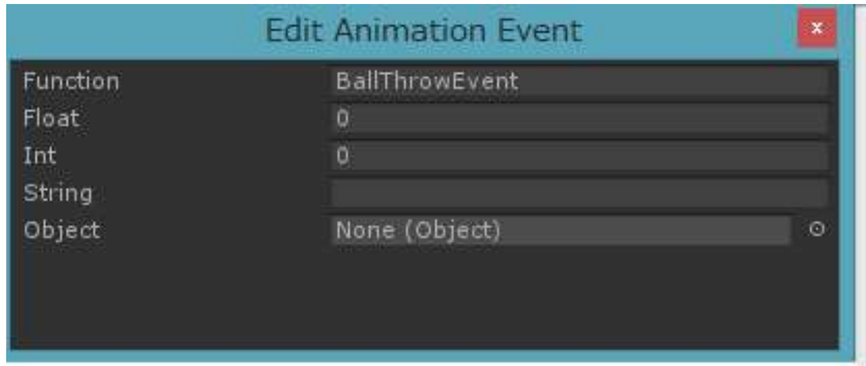
공을 던지는 애니메이션

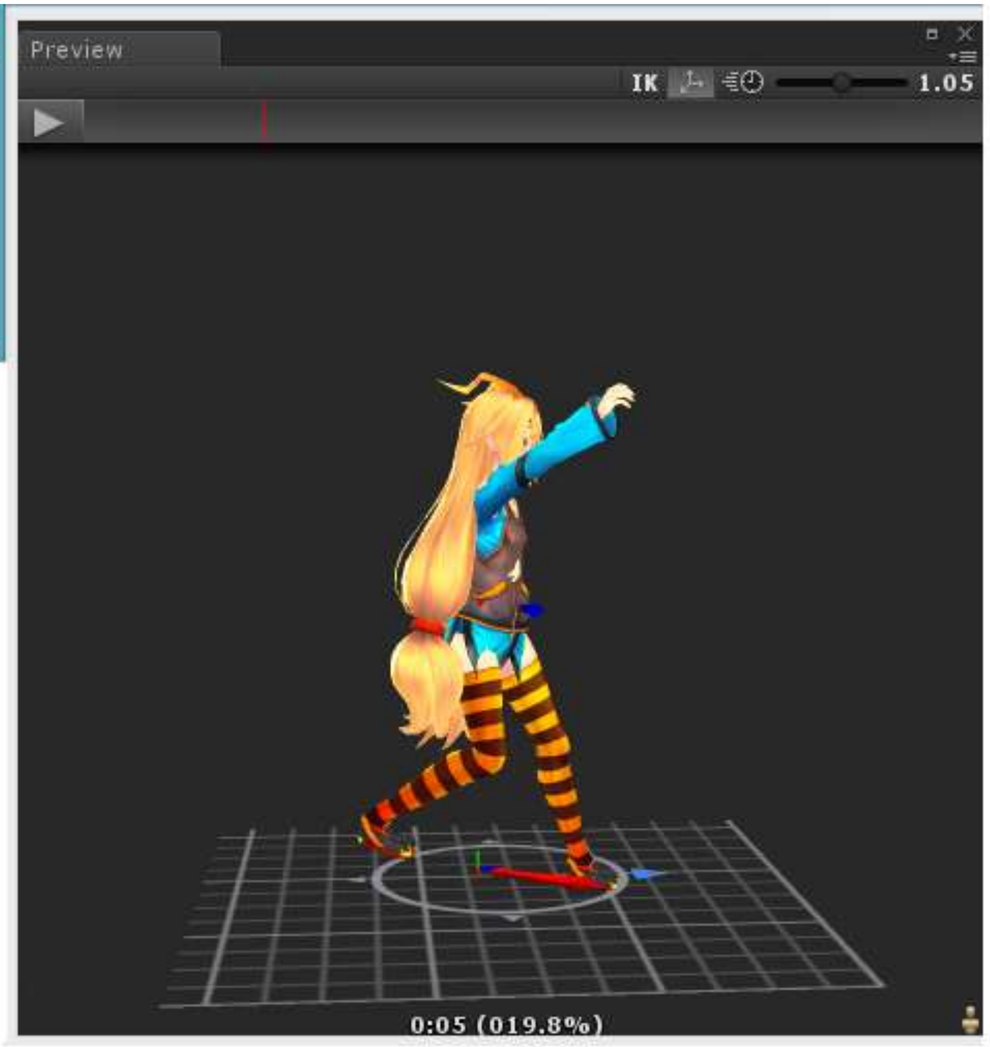


이 타이밍에 공을 생성해서 발사한다

AnimationEvent

이 타이밍에 [BallThrowEvent]가 정의되어 있다





UniRx를 사용하지 않고 작성하면

```
//速度の一時保存変数
private int _ballSpeed;
//向きの一時保存変数
private Vector3 _ballDirection;

/// <summary>
//공을 던지는 처리를 시작한다
/// </summary>
/// <param name="speed"></param>
/// <param name="direction"></param>
private void StartThrowBall(int speed, Vector3 direction)
{
    _ballSpeed = speed;
    _ballDirection = direction;
    _animator.SetTrigger("TriggerBallThrow");
}

/// <summary>
// AnimationEvent의 콜백
/// </summary>
private void BallThrowEvent()
{
    // 공을 생성해서 던진다
    var ball = CreateBallAndChangeVelocity(_ballSpeed, _ballDirection);
    // 5초후 소멸한다
    Destroy(ball,5.0f);
}
```

UniRx를 사용하지 않고 작성하면

```
//速度の一時保存変数
private int _ballSpeed;
//向きの一時保存変数
private Vector3 _ballDirection;

/// <summary>
//공을 던지는 처리를 시작한다
/// </summary>
/// <param name="speed"></param>
/// <param name="direction"></param>
private void StartThrowBall(int speed, Vector3 direction)
{
    _ballSpeed = speed;
    _ballDirection = direction;
    _animator.SetTrigger("TriggerBallThrow");
}

/// <summary>
// AnimationEvent의 콜백
/// </summary>
private void BallThrowEvent()
{
    // 공을 생성해서 던진다
    var ball = CreateBallAndChangeVelocity(_ballSpeed, _ballDirection);
    // 5초후 소멸한다
    Destroy(ball,5.0f);
}
```

여기에서 Animation 재생 시작

UniRx를 사용하지 않고 작성하면

```
//速度の一時保存変数
private int _ballSpeed;
//向きの一時保存変数
private Vector3 _ballDirection;

/// <summary>
//공을 던지는 처리를 시작한다
/// </summary>
/// <param name="speed"></param>
/// <param name="direction"></param>
private void StartThrowBall(int speed, Vector3 direction)
{
    _ballSpeed = speed;
    _ballDirection = direction;
    _animator.SetTrigger("TriggerBallThrow");
}

/// <summary>
// AnimationEvent의 콜백
/// </summary>
private void BallThrowEvent()
{
    // 공을 생성해서 던진다
    var ball = CreateBallAndChangeVelocity(_ballSpeed, _ballDirection);
    // 5초후 소멸한다
    Destroy(ball,5.0f);
}
```

여기에 콜백

UniRx를 사용하지 않고 작성하면

```
//速度の一時保存変数
private int _ballSpeed;
//向きの一時保存変数
private Vector3 _ballDirection;

/// <summary>
//공을 던지는 처리를 시작한다
/// </summary>
/// <param name="speed"></param>
/// <param name="direction"></param>
private void StartThrowBall(int speed, Vector3 direction)
{
    _ballSpeed = speed;
    _ballDirection = direction;
    _animator.SetTrigger("TriggerBallThrow");
}

/// <summary>
// AnimationEvent의 콜백
/// </summary>
private void BallThrowEvent()
{
    // 공을 생성해서 던진다
    var ball = CreateBallAndChangeVelocity(_ballSpeed, _ballDirection);
    // 5초후 소멸한다
    Destroy(ball,5.0f);
}
```

하나의 연결된 처리인데도

분리되어 버린다

UniRx를 사용하지 않고 작성하면

```
//速度の一時保存変数
private int _ballSpeed;
//向きの一時保存変数
private Vector3 _ballDirection;
```

임시 보관용의 변수

```
/// <summary>
//공을 던지는 처리를 시작한다
/// </summary>
/// <param name="speed"></param>
/// <param name="direction"></param>
private void StartThrowBall(int speed, Vector3 direction)
{
    _ballSpeed = speed;
    _ballDirection = direction;
    _animator.SetTrigger("TriggerBallThrow");
}

/// <summary>
// AnimationEvent의 콜백
/// </summary>
private void BallThrowEvent()
{
    // 공을 생성해서 던진다
    var ball = CreateBallAndChangeVelocity(_ballSpeed, _ballDirection);
    // 5초후 소멸한다
    Destroy(ball,5.0f);
}
```

이 것을 StartAsCoroutine으로 작성하면

```
/// <summary>
// AnimationEvent 통지용 Subject
/// </summary>
private Subject<Unit> _animationEventSubject = new Subject<Unit>();

/// <summary>
// AnimationEvent의 콜백
/// </summary>
private void BallThrowEvent()
{
    _animationEventSubject.OnNext(Unit.Default);
}

/// <summary>
// 공을 던지는 일련의 처리를 하는 코루틴
/// </summary>
private IEnumerator ThrowBallCoroutine(int speed, Vector3 direction)
{
    var waitStream = _animationEventSubject.FirstOrDefault().Replay();
    waitStream.Connect();

    // 애니메이션 시작
    _animator.SetTrigger("TriggerBallThrow");

    // 애니메이션 이벤트가 올때까지 대기
    yield return waitStream.StartAsCoroutine();

    var ball = CreateBallAndChangeVelocity(speed, direction);
    Destroy(ball, 5.0f);
}
```

이 것을 StartAsCoroutine으로 작성하면

```
/// <summary>
// AnimationEvent 통지용 Subject
/// </summary>
private Subject<Unit> _animationEventSubject = new Subject<Unit>();

/// <summary>
// AnimationEvent의 콜백
/// </summary>
private void BallThrowEvent()
{
    _animationEventSubject.OnNext(Unit.Default);
}

/// <summary>
// 공을 던지는 일련의 처리를 하는 코루틴
/// </summary>
private IEnumerator ThrowBallCoroutine(int speed, Vector3 direction)
{
    var waitStream = _animationEventSubject.FirstOrDefault().Replay();
    waitStream.Connect();

    // 애니메이션 시작
    _animator.SetTrigger("TriggerBallThrow");

    // 애니메이션 이벤트가 올때까지 대기
    yield return waitStream.StartAsCoroutine();

    var ball = CreateBallAndChangeVelocity(speed, direction);
    Destroy(ball, 5.0f);
}
```

AnimationEvent를 Observable화

이 것을 StartAsCoroutine으로 작성하면

```
/// <summary>
// AnimationEvent 통지용 Subject
/// </summary>
private Subject<Unit> _animationEventSubject = new Subject<Unit>();

/// <summary>
// AnimationEvent의 콜백
/// </summary>
private void BallThrowEvent()
{
    _animationEventSubject.OnNext(Unit.Default);
}

/// <summary>
// 공을 던지는 일련의 처리를 하는 코루틴
/// </summary>
private IEnumerator ThrowBallCoroutine(int speed, Vector3 direction)
{
    var waitStream = _animationEventSubject.FirstOrDefault().Replay();
    waitStream.Connect();

    // 애니메이션 시작
    _animator.SetTrigger("TriggerBallThrow");

    // 애니메이션 이벤트가 올때까지 대기
    yield return waitStream.StartAsCoroutine();

    var ball = CreateBallAndChangeVelocity(speed, direction);
    Destroy(ball, 5.0f);
}
```

코루틴

이 것을 StartAsCoroutine으로 작성하면

```
/// <summary>
// AnimationEvent 통지용 Subject
/// </summary>
private Subject<Unit> _animationEventSubject = new Subject<Unit>();

/// <summary>
// AnimationEvent의 콜백
/// </summary>
private void BallThrowEvent()
{
    _animationEventSubject.OnNext(Unit.Default);
}

/// <summary>
// 공을 던지는 일련의 처리를 하는 코루틴
/// </summary>
private IEnumerator ThrowBallCoroutine(int speed, Vector3 direction)
{
    var waitStream = _animationEventSubject.FirstOrDefault().Replay();
    waitStream.Connect();

    // 애니메이션 시작
    _animator.SetTrigger("TriggerBallThrow");

    // 애니메이션 이벤트가 올때까지 대기
    yield return waitStream.StartAsCoroutine();

    var ball = CreateBallAndChangeVelocity(speed, direction);
    Destroy(ball, 5.0f);
}
```

애니메이션 시작

이 것을 StartAsCoroutine으로 작성하면

```
/// <summary>
// AnimationEvent 통지용 Subject
/// </summary>
private Subject<Unit> _animationEventSubject = new Subject<Unit>();

/// <summary>
// AnimationEvent의 콜백
/// </summary>
private void BallThrowEvent()
{
    _animationEventSubject.OnNext(Unit.Default);
}

/// <summary>
// 공을 던지는 일련의 처리를 하는 코루틴
/// </summary>
private IEnumerator ThrowBallCoroutine(int speed, Vector3 direction)
{
    var waitStream = _animationEventSubject.FirstOrDefault().Replay();
    waitStream.Connect();

    // 애니메이션 시작
    _animator.SetTrigger("TriggerBallThrow");

    // 애니메이션 이벤트가 올때까지 대기
    yield return waitStream.StartAsCoroutine();

    var ball = CreateBallAndChangeVelocity(speed, direction);
    Destroy(ball, 5.0f);
}
```

콜백이 올때까지 yield return

이 것을 StartAsCoroutine으로 작성하면

```
/// <summary>
// AnimationEvent 통지용 Subject
/// </summary>
private Subject<Unit> _animationEventSubject = new Subject<Unit>();

/// <summary>
// AnimationEvent의 콜백
/// </summary>
private void BallThrowEvent()
{
    _animationEventSubject.OnNext(Unit.Default);
}

/// <summary>
// 공을 던지는 일련의 처리를 하는 코루틴
/// </summary>
private IEnumerator ThrowBallCoroutine(int speed, Vector3 direction)
{
    var waitStream = _animationEventSubject.FirstOrDefault().Replay();
    waitStream.Connect();

    // 애니메이션 시작
    _animator.SetTrigger("TriggerBallThrow");

    // 애니메이션 이벤트가 올때까지 대기
    yield return waitStream.StartAsCoroutine();

    var ball = CreateBallAndChangeVelocity(speed, direction);
    Destroy(ball, 5.0f);
}
```

콜백이 들어간 비동기처리를

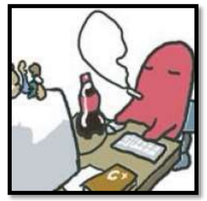
동기처리처럼 쓸수 있다

이 것을 StartAsCoroutine으로 작성하면

```
/// <summary>
// AnimationEvent 통지용 Subject
/// </summary>
private Subject<Unit> _animationEventSubject = new Subject<Unit>();

/// <summary>
// AnimationEvent의 콜백
/// </summary>
private void BallThrowEvent()
{
    _animationEventSubject.OnNext(Unit.Default);
}

/// <summary>
// 공을 던지는 일련의 처리를 하는 코루틴
/// </summary>
private IEnumerator ThrowBallCoroutine(int speed, Vector3 direction)
{
    var waitStream = _animationEventSubject.FirstOrDefault().Replay();
    waitStream.Connect();

    // 애니메이션 시작
    _animator.SetTrigger("TriggerBallThrow");

    // 애니메이션 이벤트가 올때까지 대기
    yield return waitStream.StartAsCoroutine();

    var ball = CreateBallAndChangeVelocity(speed, direction);
    Destroy(ball, 5.0f);
}
```

임시 보관용 변수들도 사라졌다

감사합니다.



박민근(알콜코더)
@agebreak / agebreak@gmail.com