더 효과적인 코루틴이란?

MEC(More Effective Coroutines)는 Unity 에셋 스토어의 무료 에셋입니다. 아직 가지고 있지 않다면 여기에서 얻을 수 있습니다.

보다 효과적인 코루틴은 Unity 코루틴보다 약 2배 빠르게 실행되고 프레임당 메모리 할당이 0인 향상된 코루틴 구현입니다. 성능을 최대화하고 앱에서 코루틴을 위한 견고한 플랫폼을 만들기 위해 광범위하게 테스트되고 개선되었습니다.

특징	Unity의 기본 코루틴	MEC 무료	MEC 프로
수익률을 사용합니다 구조	✓	✓	✓
100,000개의 빈 코루틴을 실행하는 데 걸리는 시간	~ 110.53	~ 9.62	~ 9.64
코루틴의 싱글톤 인스턴스를 실행할 수 있습니다.			✓
프로세스 중간에 코루틴의 타이밍을 전환할 수 있습니다.	✓		✓
통화 지연 주기적으로 그리고 CallContinously 함수		✓	✓
게임 개체로 코루틴을 중지하거나 일시 중지할지 여부를 선택할 수 있습니다.		✓	✓
MEC와 호환 가능 다중 스레드		✓	✓
코루틴 실행 일시 중지 및 계속하기		✓	✓

세그먼트	Unity의 기본 코루틴	MEC 무료	MEC 프로
업데이트	✓	✓	✓
고정 업데이트	✓	✓	✓
늦은 업데이트		✓	✓
느린 업데이트		✓	✓
실시간 업데이트	✓		✓
에디터 업데이트			✓
에디터 느린 업데이트			✓
프레임 끝	✓		✓
수동 기간			✓
핸들에서 직접 코루틴을 제어할 수 있습니다.			✓
코루틴을 연결하여 함께 중지하거나 일시 중지할 수 있습니다.			✓
코루틴은 함수가 true를 반환할 때까지 실행을 보류할 수 있습니다.			✓

변경 방법

더 효과적인 코루틴은 Unity의 코루틴과 약간 다르게 호출됩니다. 구조는 정확히 동일하므로 대부분의 경우 코드 내에서 찾고 바꾸기만 하면 됩니다.

먼저 MEC에서 사용하는 두 개의 네임스페이스를 포함해야 합니다. MEC 코루틴은 MEC 네임스페이스에 정의되어 있으며

Unity 코루틴이 사용하는 System.Collections 기능. System.Collections는 Unity의 코루틴을 제외하고는 거의 사용되지 않으므로 전환하는 쉬운 방법은 "찾기 및 바꾸기"를 수행한 다음 오류가 있는 줄만 변경하는 것입니다. 따라서 이 두 가지 using 문이 MEC 코루틴을 사용하는 모든 C# 스크립트의 맨 위에 있는지 확인하세요.

```
System.Collections.Generic 사용;
MEC 사용;
```

다음으로 StartCoroutine의 모든 인스턴스를 교체하십시오.

```
시작 코루틴(_CheckForWin());
```

..는 RunCoroutine으로 대체됩니다. (프로세스를 정의할 때 실행 루프를 선택해야 하며 기본값은 "Segment.Update"입니다.)

```
// 업데이트 세그먼트에서 실행하려면:
Timing.RunCoroutine(_CheckForWin()); // FixedUpdate 세그먼
트에서 실행하려면: Timing.RunCoroutine(_CheckForWin(),
Segment.FixedUpdate); // LateUpdate 세그먼트에서 실행하려면: Timing.RunCoroutine(_CheckForWin(),
Segment.LateUpdate); // SlowUpdate 세그먼트에서 실행하려면: Timing.RunCoroutine(_CheckForWin(),
Segment.SlowUpdate);
```

참고: 게임 오브젝트를 이동하거나 변경하는 모든 코루틴에 대해 RunCoroutine 호출과 함께 CancelWith(6페이지 참조)를 사용해야 합니다.

그러면 프로세스의 헤더도 변경해야 합니다. 다음과 같이 바뀝니다.

```
IEnumerator _CheckForWin()
{
...
}
```

이에:

```
IEnumerator<float> _CheckForWin()
{
...
}
```

모든 코루틴 함수 앞에 항상 밑줄을 두는 습관을 갖는 것은 매우 좋은 생각입니다. 그 이유는 코루틴(Unity와 MEC 모두)이

RunCoroutine을 사용하지 않고 함수를 실행하려고 하면 올바르게 실행되는 척하지만 실제로는 전혀 실행되지 않는 경향이 있습니다. 함수 앞의 "_"는 항상 "Timing.RunCoroutine(_CheckForWin());"을 사용하는 것을 기억하는 데 도움이 됩니다. 일반 함수 "_CheckForWin();"을 호출하는 것처럼 호출하는 대신.

다음 프레임을 기다리고 싶을 때마다 "yield return Timing.WaitForOneFrame;"을 사용할 수 있습니다. 또는 "yield return 0;". 이 두 문장은 모두 동일한 것으로 평가되므로 가장 좋아하는 것을 선택할 수 있지만 WaitForOneFrame을 사용하면 코루틴 및/또는 Unity에 익숙하지 않은 사람도 코드를 더 쉽게 이해할 수 있으므로 WaitForOneFrame을 사용하는 것이 좋습니다. 그래서 이거,

```
IEnumerator _CheckForWin()
{
        동안 (_cubesHit < TotalCubes)
        {
                WinText.text = "아직 당첨되지 않았습니다.";

                수익률 반환 null;
        }

        WinText.text = "당신이 이겼습니다!";
}
```

다음과 같이 바뀔 것입니다.

```
IEnumerator<float> _CheckForWin()
{
        동안 (_cubesHit < TotalCubes)
        {
                WinText.text = "아직 당첨되지 않았습니다.";

                항복 반환 Timing.WaitForOneFrame;
        }

        WinText.text = "당신이 이겼습니다!";
}
```

한 프레임이 아닌 몇 초 동안 일시 중지하려면 Timing.WaitForSeconds를 사용할 수 있습니다. 그래서 이거,

```
IEnumerator _CheckForWin()
{
        동안 (_cubesHit < TotalCubes)
```

```
            {
                WinText.text = "아직 당첨되지 않았습니다.";

                yield return new WaitForSeconds(0.1f);
            }
            WinText.text = "당신이 이겼습니다!";
        }
```

다음과 같이 바뀝니다.

```
    IEnumerator<float> _CheckForWin() {

        동안 (_cubesHit < TotalCubes) {

            WinText.text = "아직 당첨되지 않았습니다."; 수익률 반환
            Timing.WaitForSeconds(0.1f);
        }

        WinText.text = "당신이 이겼습니다!";
    }
```

Unity의 기본 코루틴을 사용하면 yield return을 약식 구문으로 사용하여 하나의 코루틴을 만들고 계속하기 전에 완료될 때까지 기다릴 수 있습니다.

```
    IEnumerator _EnableHappyFace() {

        반환 반환 _CheckForWin();
        HappyFaceObject.SetActive(참);
    }
```

위의 함수는 _CheckForWin 코루틴 함수를 시작하고 _EnableHappyFace 코루틴 함수는 _CheckForWin이 실제로 완료될 때까지 일시 중지된 상태로 유지된 다음 HappyFaceObject를 활성화합니다. 위의 yield return 문은 "yield return StartCoroutine(_CheckForWin());"과 동일하지만 Unity는 무대 뒤에서 StartCoroutine을 호출합니다.

MEC Pro에는 RunCoroutine을 자동으로 호출하는 단축 버전도 있지만 MEC에서는
무료 장수 버전을 사용해야 합니다.

```
    IEnumerator<float> _EnableHappyFace() {

        // 이 라인은 Unity의 기본 코루틴을 대체할 수 있습니다.
    MEC Free 및 MEC Pro 모두
        수익률
    Timing.WaitUntilDone(Timing.RunCoroutine(_CheckForWin());
```

```
        // 이 단축 버전은 MEC Pro에서만 작동합니다.
        항복 반환 Timing.WaitUntilDone(_CheckForWin());

        HappyFaceObject.SetActive(참);
    }
```

취소

많은 경우 시간이 지남에 따라 A 지점에서 B 지점으로 버튼을 이동하는 것과 같이 장면의 게임 개체에 영향을 주는 코루틴을 만들 수 있습니다. MEC 코루틴은 생성된 GameObject가 Unity의 코루틴처럼 파괴되거나 비활성화될 때 실행을 자동으로 중지하지 않습니다. 이 동작이 기본적으로 MEC에 포함되지 않는 이유는 모든 코루틴에 항상 의미가 있는 것은 아니기 때문입니다. 코루틴이 UI 요소에서 작동하지 않는다면 이 검사는 기껏해야 처리 낭비이며 최악의 경우 원치 않는 동작으로 이어질 수 있습니다.

UI 요소로 작업할 때 화면을 변경할 때 오류가 발생하지 않도록 이 검사를 켜는 것이 좋습니다. MEC에서는 다음과 같이 CancelWith 확장을 사용하여 이를 수행합니다.

```
타이밍.RunCoroutine(_moveMyButton().CancelWith(gameObject));
```

TLDR: UI 요소에 영향을 주는 모든 코루틴에서 .CancelWith를 사용해야 합니다.

CancelWith 는 모든 코루틴이 생성하는 피할 수 없는 GC 할당의 크기를 약 20바이트만큼 늘립니다. 20바이트는 크지 않지만 가능한 모든 GC 할당을 피하려면 함수를 사용하지 않고 CancelWith가 수행하는 작업을 비교적 쉽게 수행할 수 있습니다. 코루틴 내부의 모든 yield return 문 후에 다음을 확인하십시오.

```
if(gameObject != null && gameObject.activeInHierarchy)
```

코루틴 일시 중지 및 재개

MEC 코루틴을 일시 중지했다가 나중에 다시 시작할 수 있습니다. Unity의 코루틴은 이를 수행하지 않지만 개념은 매우 간단합니다.

```
Timing.PauseCoroutines(handleToACoroutine);
// 코드의 다른 곳...
Timing.ResumeCoroutines(handleToACoroutine);
```

코루틴이 이미 해당 상태에 있을 때 코루틴을 일시 중지하거나 재개하는 것은 괜찮습니다. 예를 들어 카메라를 이리저리 움직이지만 OnDrag 이벤트가 발생하는 모든 프레임에서 카메라를 일시 중지한 코루틴이 있을 수 있습니다. 그런 다음 드래그가 해제되면 resume을 한 번 호출합니다.

코루틴이 WaitForSeconds 호출로 넘어가면 해당 코루틴이 일시 중지되는 동안 초는 카운트다운되지 않습니다.

WaitUntilDone

Unity의 기본 코루틴에는 일부 변수를 반환할 수 있는 몇 가지 경우가 있습니다. 예를 들어 "yield return asyncOperation;"을 할 수 있습니다. MEC에도 이를 수행하는 기능이 있으며 이 기능을 WaitUntilDone이라고 합니다.

```
항복 반환 Timing.WaitUntilDone(wwwObject);
수익 반환 Timing.WaitUntilDone(asyncOperation);
수익률 반환 Timing.WaitUntilDone(customYieldInstruction);

// MEC Pro를 사용하면 WaitUntilDone으로 더 많은 작업을 수행할 수 있습니다.
수익률 반환 Timing.WaitUntilDone(newCoroutine);
  // 위의 코드는 자동으로 새로운 코루틴을 시작하고 현재 코루틴을 유지합니다.


항복 반환
Timing.WaitUntilTrue(functionDelegateThatReturnsBool);
항복 반환
Timing.WaitUntilFalse(functionDelegateThatReturnsBool);
```

느린 업데이트

Unity의 코루틴에는 느린 업데이트 루프의 개념이 없지만 MEC 코루틴에는 있습니다.

느린 업데이트 루프는 기본적으로 초당 7번 실행됩니다. 절대 시간 척도를 사용하므로 Unity의 시간 척도를 늦춰도 SlowUpdate는 느려지지 않습니다. SlowUpdate는 사용자에게 텍스트를 표시하는 것과 같은 작업에 적합합니다. 그보다 더 빠르게 값을 업데이트하면 사용자는 이러한 빠른 변경 사항을 실제로 볼 수 없기 때문입니다.

SlowUpdate를 사용하는 것과 항상 "yield return Timing.WaitForSeconds(1f/7f);"로 항복하는 것 사이에는 두 가지 주요 차이점이 있습니다. 첫 번째는 절대 시간 척도이고 두 번째는 모든 SlowUpdate 틱이 동시에 발생한다는 것입니다. 텍스트 상자를 1초에 7번 변경하는 것은 모든 텍스트 상자가 동일한 프레임 동안 변경되는 경우에만 보기에 좋아 보이기 때문에 중요합니다.

```
Timing.RunCoroutine(_UpdateTime(), Segment.SlowUpdate);

개인 IEnumerator<float> _UpdateTime()
{
        동안(사실)
        {
                시계 = Timing.LocalTime;
```

```
            수익률 반환 0f;
        }
    }
```

SlowUpdate는 임시 디버깅 변수를 확인하는 데도 잘 작동합니다. 예를 들어 프로젝트를 다시 빌드하는 데 오랜 시간이 걸리는 경우 스크립트에서 해당 스크립트의 값을 재설정하는 공개 bool을 설정할 수 있습니다. 해당 bool의 값이 주기적으로 true로 설정되었는지 확인해야 하며 해당 확인을 수행하기에 완벽한 기간은 SlowUpdate입니다. 사용자가 확인란을 선택하면 즉시 응답하는 것처럼 느껴지지만 초당 30 – 100번(프레임 속도에 따라 다름)보다 1/7초마다 확인하는 앱에서 훨씬 적은 처리를 사용합니다.

참고: Unity의 Time 클래스는 이 세그먼트에 대해 아무것도 모르기 때문에 Unity의 Time.deltaTime 변수는 SlowUpdate에서 올바른 값을 반환하지 않습니다. 다행히도 대신 Timing.DeltaTime을 사용할 수 있습니다. Timing.DeltaTime 및 Timing.LocalTime은 실행 중인 타이밍 세그먼트에 관계없이 MEC 코루틴 내에서 항상 좋은 시간 값을 갖습니다.

원하는 경우 SlowUpdate가 실행되는 속도를 변경할 수도 있습니다.

```
Timing.Instance.TimeBetweenSlowUpdateCalls = 3f;
```

위의 줄은 SlowUpdate가 3초마다 한 번만 실행되도록 합니다.

태그

코루틴을 시작할 때 태그를 제공할 수 있는 옵션이 있습니다. 태그는 해당 코루틴을 식별하는 문자열입니다. 코루틴 또는 코루틴 그룹에 태그를 지정하면 나중에 KillCoroutine(tag) 또는 KillAllCoroutines(tag)를 사용하여 해당 코루틴 또는 해당 그룹을 종료할 수 있습니다.

```
무효 시작()
{
        Timing.RunCoroutine(_shout(1, "안녕하세요"), "소리쳐");
        Timing.RunCoroutine(_shout(2, "세계!"), "외쳐");

        Timing.RunCoroutine(_shout(3, "I"), "shout2");
        Timing.RunCoroutine(_shout(4, "좋아요"), "shout2");
        Timing.RunCoroutine(_shout(5, "케이크!"), "shout2");

        Timing.RunCoroutine(_shout(6, "굽기"), "shout3");
        Timing.RunCoroutine(_shout(7, "나"), "shout3");
        Timing.RunCoroutine(_shout(8, "케이크!"), "shout3");

        Debug.Log("죽었습니다"              + Timing.KillAllCoroutines("shout2"));
}
```

```
IEnumerator<float> _shout(부동 시간, 문자열 텍스트)
{
        수익률 반환 Timing.WaitForSeconds(시간);

        Debug.Log(텍스트);
}


// 출력:
// 죽임 3
// 안녕하십니까
// 세계!
// 빵 굽기
// 나
// 케이크!
```

LocalTime 및 DeltaTime

MEC는 각 세그먼트 내부의 현지 시간을 추적하고 LocalTime 및 DeltaTime 변수를 업데이트된 상태로 유지합니다. Unity의 기본 Time 클래스는 대부분 잘 작동하지만 SlowUpdate에서는 제대로 작동하지 않으며 EditorUpdate 및 EditorSlowUpdate 세그먼트에서는 완전히 사용할 수 없습니다. 그렇기 때문에 MEC 코루틴 내에서 Time.time보다 Timing.LocalTime을 사용하는 것이 거의 항상 더 좋습니다.

추가 기능

Timing 객체에는 CallDelayed, CallContinously 및 CallPeriodically의 세 가지 도우미 함수도 포함되어 있습니다.

- CallDelayed는 몇 초 후에 지정된 작업을 호출합니다.
- CallContinously는 몇 초 동안 매 프레임마다 작업을 호출합니다.
- CallPeriodically는 몇 초 동안 "x"초마다 작업을 호출합니다.

이 세 가지 모두 코루틴을 사용하여 쉽게 만들 수 있지만 이 기본 기능은 너무 자주 사용되어 기본 모듈에 포함시켰습니다.

```
// 지금부터 2초 후 _RunFor5Seconds가 시작됩니다.
Timing.CallDelayed(2f, 대리자 {
    타이밍.RunCoroutine(_RunFor5Seconds(핸들)); });

// 이것은 동일한 작업을 수행하지만 클로저를 생성하지 않습니다.
// (클로저는 GC 할당을 생성합니다.)
// "handle"은 CallDelayed에 전달되고 CallDelayed는 // 변수 "x"로 RunCoroutine에 다시 전달합니다.
```

```
Timing.CallDelayed<IEnumerator<float>>(핸들, 2f, x => {
타이밍.RunCoroutine(_RunFor5Seconds(x)); });
```

```
private void PushOnGameObject(Vector3 금액)
{
        transform.position += 금액 * Time.deltaTime;
}

// 이것은 이 객체를 4초 동안 초당 하나의 세계 단위 앞으로 밀어냅니다.

Timing.CallContinuously(4f, delegate
{ PushOnGameObject(Vector3.forward); }, Segment.FixedUpdate);

// CallContinously에는 클로저가 아닌 버전도 있습니다. CallContinously에서 클로저를 만들지 않으려는 시도는 //
추가로 중요합니다. // 그러면 모든 프레임에서 GC 할당이 발생하기 때문입니다.

Timing.CallContinuously<Vector3>(Vector3.forward, 4f,
    벡터 => PushOnGameObject(벡터), Segment.FixedUpdate);
```

유체 아키텍처

어떤 사람들은 더 읽기 쉽고 현대적으로 보일 수 있기 때문에 코드를 작성할 때 유동적인 구문을 선호합니다. 이 구문은 MEC를 사용하는 완전히 선택적인 대체 방법입니다.

```
// 코루틴을 실행하는 일반적인 방법
타이밍.RunCoroutine(_Foo().CancelWith(gameObject));

// 코루틴을 실행하는 유동적인 방법
_Foo().CancelWith(gameObject).RunCoroutine();
```

그래서 여기에 "코루틴 함수"."modifier"."modifier"."RunCoroutine("코루틴 함수"."modifier");"가 아니라 "RunCoroutine"이 있습니다. 논리가 출발점에서 결론으로 흐르기 때문에 유체라고 합니다.

코루틴을 실행하는 다른 방법과 마찬가지로 RunCoroutine 사용을 잊어버린 경우 컴파일러는 불평하지 않지만 코루틴을 실행하지 않는다는 것을 기억하십시오. 이 사실은 Run 호출이 줄 끝에 있을 때 기억하기가 조금 더 어려울 수 있습니다. 또한 Unity의 기본 코루틴 API는 해당 구문을 지원하지 않기 때문에 유동적인 구문은 Unity의 기본 코루틴 작업에 익숙한 다른 개발자에게 혼란을 줄 수 있습니다.

최종 결과는 정확히 동일하며 어느 쪽도 다른 쪽보다 성능이 좋지 않습니다(유동 아키텍처의 경우 항상 그런 것은 아니지만 이 경우에는 해당됩니다.). 어떤 구문을 더 좋아하느냐의 문제일 뿐입니다.

자주하는 질문

Q: MEC에 WaitForEndOfFrame에 대한 기능이 있습니까?

MEC Free에서는 구현되지 않지만 MEC Pro에는 이에 대한 세그먼트가 있습니다.

참고: WaitForEndOfFrame이 실제로 수행하는 작업에 대해 약간의 혼동이 있습니다. 다음 프레임까지 양보하고 싶을 때 WaitForEndOfFrame은 이상적인 명령이 아니므로 "yield return null;"을 사용하는 것이 좋습니다. 많은 사람들이 Unity의 코루틴을 사용할 때 WaitForEndOfFrame을 사용합니다. 왜냐하면 Unity의 기본 코루틴에서 WaitForOneFrame과 가장 가깝기 때문에 이것이 미묘한 문제를 일으킬 수 있다는 사실을 깨닫지 못하기 때문입니다. MEC는 상수 Timing.WaitForOneFrame을 정의하므로 MEC를 사용하면 EndOfFrame을 사용하여 발생할 수 있는 시각적 결함 및 성능 저하 가능성을 생성하지 않고 원하는 경우 명시적 변수 이름을 사용할 수 있습니다.

이 페이지 각 프레임의 타이밍을 보여주는 그래프가 있습니다. 그래프에서 볼 수 있듯이 WaitForEndOfFrame은 모든 렌더링이 완료된 후 실행됩니다. Unity 코루틴에서 해당 호출을 사용하여 화면에서 버튼을 이동하는 경우 버튼은 항상 이전 프레임에서 설정한 위치에 그려집니다. 대부분의 경우 프레임 속도는 시각적으로 차이를 알아차리지 못할 만큼 충분히 높지만 이 방법을 사용하면 설명하거나 디버그하기 어려운 미묘한 시각적 결함이 발생할 수 있습니다.

예를 들어, 적 함선과 "적선 폭발" 애니메이션이 있는 경우 적함선 개체에서 Destroy를 호출하고 동시에 폭발 애니메이션을 인스턴스화하는 것이 일반적입니다. 그러나 WaitForEndOfFrame을 호출한 Unity 코루틴에서 이 작업을 수행한 경우 사용자는 배가 폭발하기 전에 잠시 깜박이는 것처럼 보이는 것을 볼 수 있습니다.

또한 캔버스는 일반적으로 업데이트 세그먼트 직후에 다시 계산하므로 Unity의 WaitForEndOfFrame을 잘못 사용하면 캔버스가 프레임당 두 번 다시 계산해야 하기 때문에 성능에 심각한 부정적인 영향을 미칠 수 있습니다.

Q: MEC에 StopCoroutine 기능이 있습니까?

A: 네. Timing.KillCoroutines()라고 합니다. 이전 Timing.RunCoroutine 명령에서 반환된 코루틴에 대한 핸들을 사용하거나 태그를 사용할 수 있습니다.

참고: KillCoroutine은 다른 함수에서 코루틴 함수를 중지하기 위한 것입니다. 해당 코루틴의 함수 내부에서 코루틴을 종료하려면 "yield break;"를 사용하는 것이 가장 좋습니다. 이는 "return;"을 호출하는 것과 같습니다. 다른 기능에서. yield break가 더 나은 이유는 KillCoroutine 명령 이 현재 실행 중인 코루틴 함수를 종료할 수 없기 때문에 해당 함수는 다음 yield 명령까지 계속 실행되지만 yield break에는 이 문제가 없기 때문입니다.

Q: MEC에 StopAllCoroutine에 대한 기능이 있습니까?

A: 네. Timing.KillCortines(). 모든 것을 일시적으로 중지하려는 경우 Timing.PauseCortines()
및 Timing.ResumeAllCortines()를 사용할 수도 있습니다.

Q: MEC에는 다른 코루틴이 완료될 때까지 하나의 코루틴을 생성하는 기능이 있습니까?

A: 네. 보유하려는 코루틴 내부에서 "yield return Timing.WaitUntilDone(coroutineHandle);"를
호출합니다. Timing.RunCoroutine을 호출할 때마다 핸들이 반환됩니다.

Q: MEC는 GC 할당을 완전히 제거합니까?

A: 아니요. MEC는 모든 프레임별 GC 할당을 제거합니다. (코루틴 내부의 힙에 메모리를 할당하지 않는 한 MEC는 이를 제어할 수 없습니다.) 코루틴이 처음 생성되면 함수 포인터와 여기에 전달하는 모든 변수가 힙에 놓이고 결국 정리해야 합니다. 가비지 컬렉터에 의해 이 피할 수 없는 할당은 Unity의 코루틴과 MEC 코루틴 모두에서 발생합니다. MEC 코루틴은 Unity 코루틴보다 평균적으로 더 적은 가비지를 할당합니다.

Q: MEC 코루틴은 항상 Unity 코루틴보다 메모리 효율성이 더 높습니까? 아니면 일부 경우에만 해당합니까?

A: MEC 코루틴은 큰 문자열을 할당하고 코루틴의 태그로 할당하는 경우를 제외하고 모든 경우에 Unity 코루틴보다 적은 GC 할당을 생성합니다.

Q: 감소된 GC 할당은 훌륭하지만 Unity의 코루틴에 비해 MEC 코루틴에 다른 이점이 있습니까?

A: MEC 인프라는 Unity의 코루틴 인프라보다 약 2배 빠르게 실행됩니다. 이에 대한 자세한 내용은 이 문서의 끝에 링크된 MEC vs Unity 코루틴의 성능에 대한 비디오를 시청하십시오.

Unity의 코루틴은 코루틴을 시작한 개체에 연결되는 반면 MEC는 중앙 개체를 사용하여 모든 프로세스를 실행합니다.
즉, 다음 세 가지가 사실입니다.

1. 현재 개체가 비활성화된 경우 Unity의 코루틴이 시작되지 않습니다. MEC 코루틴은 상관하지 않습니다.
   (CancelWith 또는 PauseWith를 사용하여 그들에게 주의를 요청하지 않는 한.)
2. GameObject를 비활성화하면 그에 연결된 모든 Unity 코루틴이
   실행을 종료합니다(재활성화할 때 다시 시작하지 않습니다.) MEC 코루틴은 명시적으로 지시하지 않는 한 이 작업을 수행하지 않습니다.
3. Unity 코루틴을 시작한 GameObject가 파괴되면 연결된 모든 코루틴도 종료됩니다. MEC 코루틴도 이 작업을 수행하지 않습니다.

MEC 코루틴을 사용하면 코루틴 그룹을 생성하여 동시에 전체 코루틴 그룹을 일시 중지/재개하거나 파괴
할 수 있습니다. Unity의 코루틴은 외부에서 코루틴을 일시 중지 및 재개하는 것을 허용하지 않으며 Unity의 코루틴
은 항상 코루틴이 시작된 gameObject별로 그룹화됩니다.

MEC 코루틴을 사용하면 원하는 경우 LateUpdate 또는 SlowUpdate 세그먼트에서 코루틴을 실행할 수 있습니다.
MEC Pro에는 더 많은 세그먼트가 있습니다.

MEC Pro에는 또한 환상적인 새로운 방식으로 코루틴을 실행할 수 있는 수많은 추가 기능이 있습니다 .

공정 수명의 고급 제어

특별한 조치를 취하지 않으면 Timing 개체가 "Movement Effects"라는 새 개체에 추가됩니다. 모든 코루틴 프로세스는 일반적으로 해당 인스턴스에 의해 전달됩니다.

그러나 사물을 더 많이 제어하려면 장면의 게임 개체 중 하나에 타이밍 개체를 직접 연결할 수 있습니다. 다른 객체에 다른 코루틴을 추가하고 원한다면 둘 이상의 Timing 객체를 생성할 수도 있습니다. Timing.RunCoroutine()의 함수는 정적이므로 어디서나 액세스할 수 있지만 Timing 객체의 인스턴스에 대한 핸들이 있는 경우 yourTimingInstance.RunCoroutineOnInstance()를 호출하여 해당 인스턴스에서 코루틴을 실행할 수 있습니다. Timing 개체의 여러 인스턴스를 생성하면 모두 함께 일시 중지되거나 파괴될 수 있는 프로세스 그룹을 효과적으로 생성할 수 있습니다.

OnError 대리자와 TimeBetweenSlowUpdateCalls 변수도 Timing 인스턴스에 연결되어 있으므로 다른 타이밍 개체에 대해 다른 오류 처리를 설정하거나 다른 속도로 실행되도록 SlowUpdate를 설정할 수 있습니다.

예시

다음은 MEC 코루틴을 사용하는 간단한 예입니다.

```
UnityEngine 사용
System.Collections.Generic 사용;
MovementEffects 사용;

공개 클래스 테스팅 : MonoBehaviour
{
        무효 시작()
        {
                CoroutineHandle 핸들 =
Timing.RunCoroutine(_RunFor10Seconds());

                핸들 = Timing.RunCoroutine(_RunFor1Second(핸들));

                타이밍.RunCoroutine(_RunFor5Seconds(핸들));
        }

        개인 IEnumerator<float> _RunFor10Seconds()
        {
                Debug.Log("10초 실행을 시작합니다.");

                수익률 반환 Timing.WaitForSeconds(10f);

                Debug.Log("10초 실행 완료.");
        }
```

```
        개인 IEnumerator<float> _RunFor1Second(CoroutineHandle
    대기 핸들) {

            Debug.Log("1초..");

            수익 반환 Timing.WaitUntilDone(waitHandle);

            Debug.Log("1초 실행을 시작합니다.");

            수익률 반환 Timing.WaitForSeconds(1f);

            Debug.Log("1초 실행 완료.");
        }

        개인 IEnumerator<float> _RunFor5Seconds(CoroutineHandle
    대기 핸들) {

            Debug.Log("5초..");

            수익 반환 Timing.WaitUntilDone(waitHandle);

            Debug.Log("5초 실행을 시작합니다.");

            수익률 반환 Timing.WaitForSeconds(5f);

            Debug.Log("5초 실행 완료.");
        }
    }
```

출력은 다음과 같습니다.

1. 10초 달리기 시작.
2. 1초..
3. 5초..
4. 10초 달리기를 마쳤습니다.
5. 1초 달리기 시작.
6. 1초 달리기를 마쳤습니다.
7. 5초 달리기 시작.
8. 5초 달리기를 마쳤습니다.

MEC와 Unity의 코루틴의 성능에 대해 자세히 알고 싶다면 다음 동영상을 확인하세요. https://www.youtube.com/watch?v=sUYN8XtuUFA

또한 이 비디오: https://www.youtube.com/watch?v=CqHl7sgam7w

MEC Pro는 여기에서 찾을 수 있습니다: https://www.assetstore.unity3d.com/en/#!/content/68480

최신 문서, FAQ 또는 작성자에게 문의하려면 http://trinary.tech/category/mec/ 을 방문하십시오.