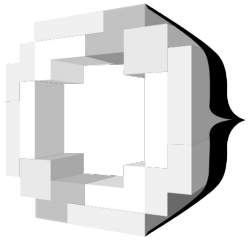
2023 유니티 중급반 4주차

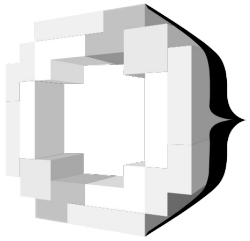
오파츠 12기 여정인, 유태환



목차

몬스터 추가하기

- 몬스터 구현하기
- 오브젝트 풀링으로 몬스터 생성하기



점수 계산하기

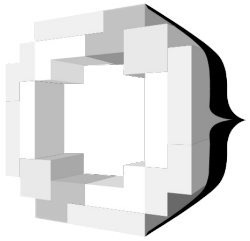
점수는 좀 특이하게 체력을 통해 계산합니다. 초기 체력에서 체력을 깎습니다.

체력 = 초기 체력 - (이동 거리 + 시간 * 5000)

최종 점수는 (체력 * 1.5 / 초기 체력) * 100 (%) 로 계산합니다. (Like 정확도)

따라서 최대 점수는 150%이고, 시간과 이동거리가 늘수록 점수가 줄어들겠죠.

그리고, 승리하거나 패배했을 때 체력 계산을 멈춥니다.

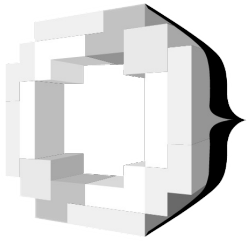


점수 계산하기

캐릭터의 이동거리를 사용하기 때문에,
playerController에서 계산을 합니다.

Init()함수에서 체력을 초기치로 초기화하고,

Update()함수에서 체력을 감소시킵니다.
그리고, 승리하거나 패배한 걸 감지해서 게임이 끝났다면 더 이상 체력을 감소시키지 않습니다.

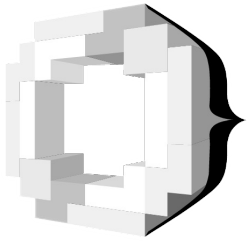


승리, 패배 판정

캐릭터가 깃발에 도착하면 승리, 캐릭터가 장애물에 부딪히거나 체력이 0으로 떨어지면 패배합니다.

캐릭터가 승리하면 승리 메시지와 최종 점수를 화면에 표시하고 게임을 끝냅니다.
캐릭터가 패배하면 패배 메시지를 화면에 표시하고 게임을 끝냅니다.

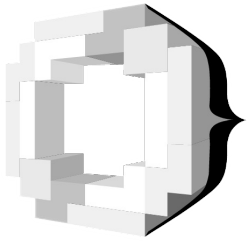
게임이 끝나면 캐릭터가 그만 움직이고, 체력 계산을 멈춥니다.



승리, 패배 판정

승리, 패배 판정은 캐릭터뿐만 아니라 게임 자체에 영향을 주므로 PlayManager에서 관리를 합니다.

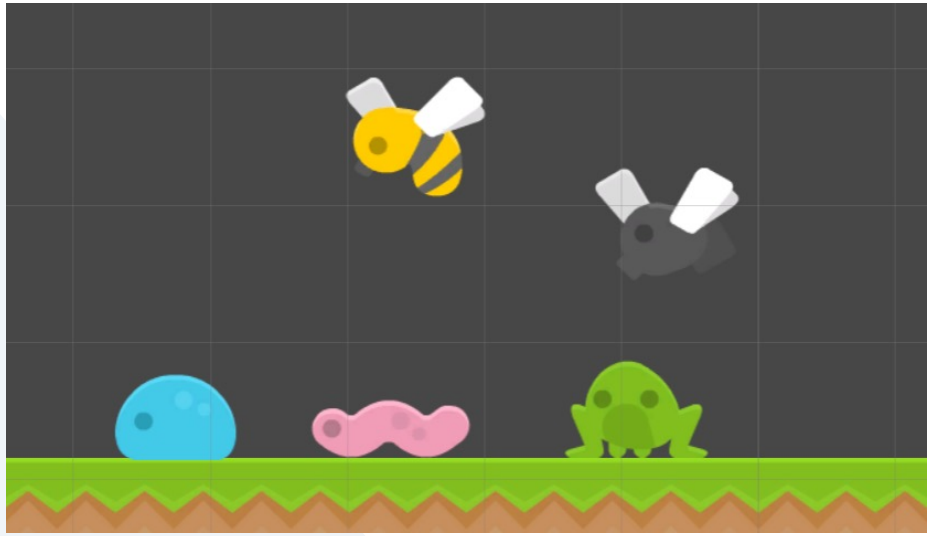
PlayManager은 승리, 패배여부를 저장하고 게임을 새로 시작할 때마다 해당 값들을 초기화합니다.

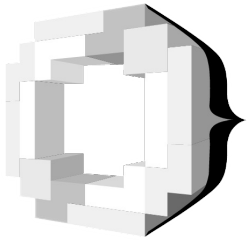


몬스터 구현하기

게임이 좀 밋밋한 듯 하니 몬스터를 추가해봅시다.

저는 슬라임으로 정했지만, 다른 몬스터로 구현해봐도 좋습니다. 몬스터에 따라서 이동 방식과 애니메이션을 약간 다르게 구현해야 하기 때문에, 다양한 몬스터를 만들면 연습하기 좋을 겁니다.





몬스터 구현하기

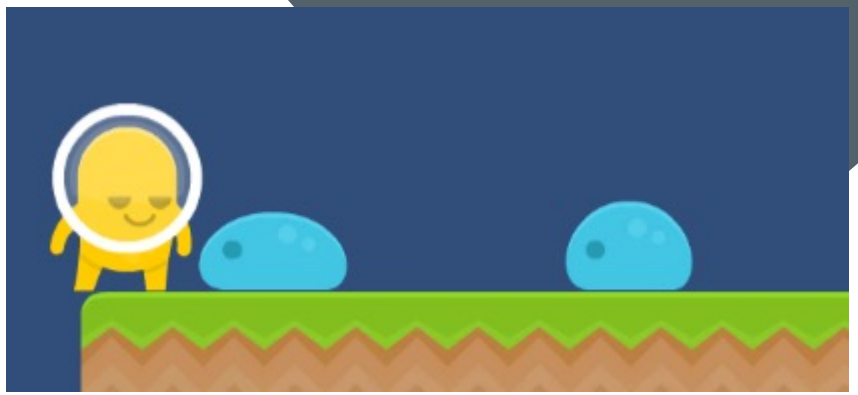
슬라임 구현

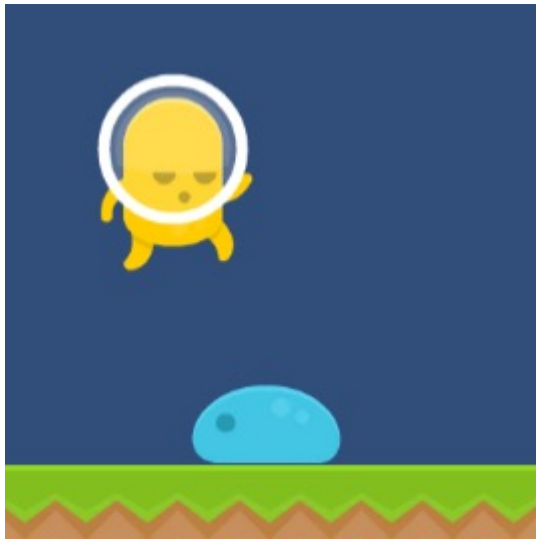
이동 방식
0.6초를 주기로 플레이어를 향해 이동합니다.

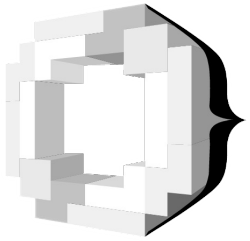
데미지
플레이어에게 닿으면 5000의 데미지를 주고 넉백시킵니다.
플레이어는 0.4초 동안 움직이지 못합니다.

기믹
플레이어가 머리를 밟으면 플레이어를 위로 점프 시킵니다.

체력
체력은 8입니다. 1초마다 1의 체력이 감소하며, 플레이어가 머리를 밟으면 5의 체력이 감소합니다. 장애물에 닿거나 맵 밖으로 떨어지면 즉사합니다.







몬스터 구현하기

이동 방식: 0.6초를 주기로 플레이어를 향해 이동합니다.

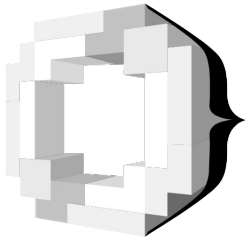
주기 별 동작 또는 시간에 따른 동작을 구현하는 방법

1. Update()와 Time.deltaTime을 사용해서 cooltime 방식으로 구현하기.

2. IEnumerator의 yield return을 통해 Coroutine 사용하기.

```
currentCooldown += Time.deltaTime;
if (currentCooldown > coolTime)
{
    //do something;
    currentCooldown = 0;
}
```

```
void Start()
{
    StartCoroutine("moving");
}

참조 0개
IEnumerator moving()
{
    while (true)
    {
        //do something

        yield return new WaitForSeconds(coolTime);
    }
}
```

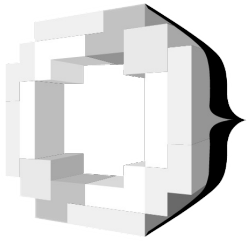


몬스터 구현하기

1. Update()와 Time.deltaTime을 사용해서 cooltime 방식으로 구현하기.

```
currentTime = 0; // 시간 기록

void Update()
{
    currentTime += Time.deltaTime; // 프레임마다 시간 더해주기

    if (currentTime > coolTime)
    {
        // 주기마다 할 것
        currentTime = 0; // 시간 기록 초기화
    }
}
```

```
currentCooldown += Time.deltaTime;
if (currentCooldown > coolTime)
{
    //do something;
    currentCooldown = 0;
}
```



몬스터 구현하기

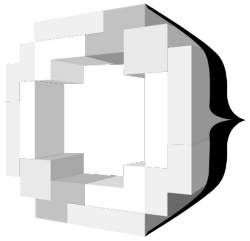
```
void Start()
{
    StartCoroutine("moving");
}

참조 0개
IEnumerator moving()
{
    while (true)
    {
        //do something

        yield return new WaitForSeconds(coolTime);
    }
}
```

2. IEnumerator의 yield return을 통해 Coroutine 사용하기.

```
void Start()
{
    StartCoroutine(“moving”)
}

Ienumerator moving()
{
    while (true)
    {
        // 주기마다 할 것
        yield return new WaitForSeconds(coolTime);
    }
}
```

조건

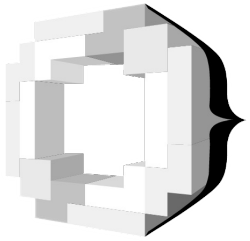
조건 충족 시 여기로 되돌아감



몬스터 구현하기

여기서는 일단 1번을 사용합시다.

```
currentCooldown += Time.deltaTime;
if (currentCooldown > coolTime)
{
    //do something;
    currentCooldown = 0;
}
```



몬스터 구현하기

이동 방식: 0.6초를 주기로 플레이어를 향해 이동합니다.

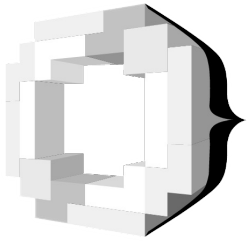
플레이어 gameObject를 태그를 통해 찾습니다.

```
GameObject.FindGameObjectWithTag("태그명");
```

(플레이어 위치) - (몬스터 위치) 를 하면 몬스터가 어느 방향으로 가야하는 지 알 수 있습니다.

Ex) 플레이어가 (3, 0)에 있고, 몬스터가 (1, 2)에 있다면
몬스터는 (2, -2)의 방향으로 가야 플레이어에 도달합니다.

Math.Sign()함수로 방향만 구합시다. (양수면 1, 음수면 -1)
여기다가 이동 속도를 곱하면 원하는 속도를 정할 수 있겠죠?



몬스터 구현하기

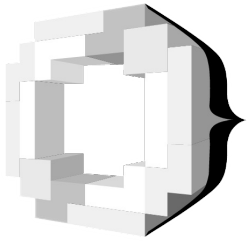
이동 방식: 0.6초를 주기로 플레이어를 향해 이동합니다.

이동은 플레이어에서 이미 구현했지만, 다른 방식도 있습니다.

1. transform.location을 일일이 수정하기
    - 실제 속도는 0으로 계산되며, collision도 안됩니다. = 안씀!

2. rigidbody.AddForce(Vector); 로 힘을 가하기

3. rigidbody.Velocity = Vector; 로 속도를 직접 정하기



몬스터 구현하기

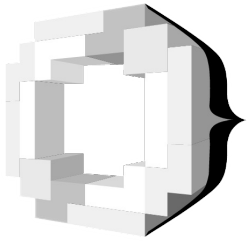
2. rigidbody.AddForce(Vector); 로 힘을 가하기

2번은 주어진 힘에다가 rigidbody의 계수들(무게, 수직마찰)을 계산하여 속도가 정해집니다. 기존의 속도를 고려합니다.

3. rigidbody.Velocity = Vector; 로 속도를 직접 정하기

3번은 최종적인 속도를 직접 정합니다. 기존의 속도를 덧씌웁니다.

구현은 2번이 더 쉽지만, 3번이 이동 속도를 직접 정하기 때문에 직관적입니다.



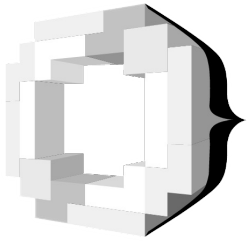
몬스터 구현하기

데미지: 플레이어에게 닿으면 5000의 데미지를 주고 넉백시킵니다. 플레이어는 0.4초 동안 움직이지 못합니다.

몬스터에게 collider을 달아주고, OnCollisionEnter을 통해 플레이어와 닿는지 체크하면 되겠죠?

monsterController에서 데미지를 직접 감소시키는 것 보다, playerController에서 함수를 통해 데미지를 받게 만드는 게 자연스럽습니다. (결국 player 체력은 playerController에 있으니까요.)

playerController에다가 hit(float hitDmg) 함수를 만들고, hitDmg만큼 체력을 감소시킵니다.



몬스터 구현하기

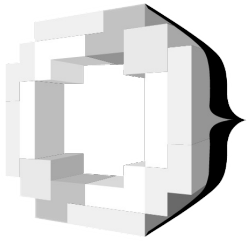
데미지: 플레이어에게 닿으면 5000의 데미지를 주고 넉백시킵니다. 플레이어는 0.4초 동안 움직이지 못합니다.

이때는 AddForce를 통해 플레이어를 날리는 것이 좋겠죠?
하지만 플레이어가 왼쪽, 오른쪽 어디로 날아가야 하는지 알려면 방향이 필요합니다.

몬스터가 플레이어를 바라보는 방향 = - (플레이어가 몬스터를 바라보는 방향)

이기 때문에, 이동에서 구한 (플레이어 위치) - (몬스터 위치)를 그대로 사용합시다

hit(int direction, float hitDmg) 로 매개변수를 바꾸고, direction 방향으로 플레이어를 날립니다. 위쪽 방향으로도 약간 힘을 주면 더 수월하게 날라갑니다.

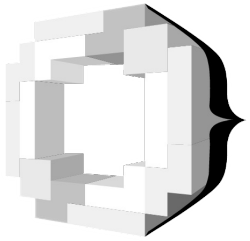


몬스터 구현하기

데미지: 플레이어에게 닿으면 5000의 데미지를 주고 넉백시킵니다. 플레이어는 0.4초 동안 움직이지 못합니다.

이번에는 Coroutine을 사용해봅시다.

우선 Coroutine이 뭔지 자세히 살펴봅시다.

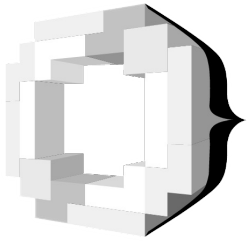


Coroutine

오른쪽의 코드는 흔히 실수하는 잘못된 코드입니다.

왜일까요?

```
void Update()
{
    if(isHit)
    {
        float time = 0;
        while(time < 3f)
        {
            isHit = true;
            time += Time.deltaTime;
        }
        isHit = false;
    }
}
```



Coroutine

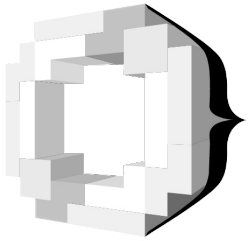
Update 함수는 기본적으로 while(true)와 같습니다. 매 프레임마다 같은 동작(루틴)을 반복합니다.

Update 함수에 저렇게 while을 쓰면 Update 함수에 추가된 다른 동작(루틴)들은 일제히 정지됩니다. While문에서 빠져나올 때까지 해당 Update 함수는 “실행중..”이 됩니다.

따라서 Update 함수에 들어가는 다른 이동, 스킬, 애니메이션 처리 등의 동작이 멈춰버립니다.

그렇기 때문에 Update 함수에는 절대! 시간에 영향을 받는 while 문을 사용하면 안됩니다.

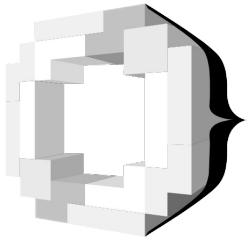
```
void Update()
{
    if(isHit)
    {
        float time = 0;
        while(time < 3f)
        {
            isHit = true;
            time += Time.deltaTime;
        }
        isHit = false;
    }
}
```



Coroutine

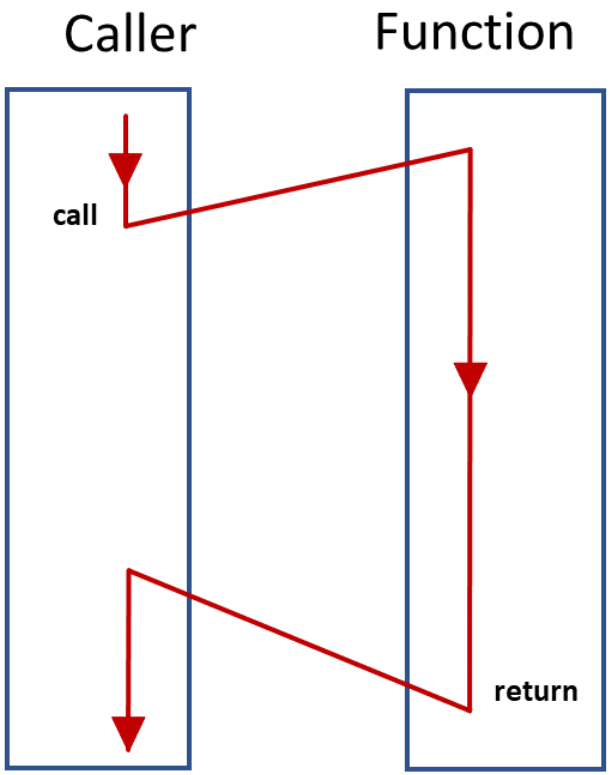
Update 함수에서 벗어나서, 3초를 따로 기다려주는 별도의 동작(루틴)이 필요합니다.

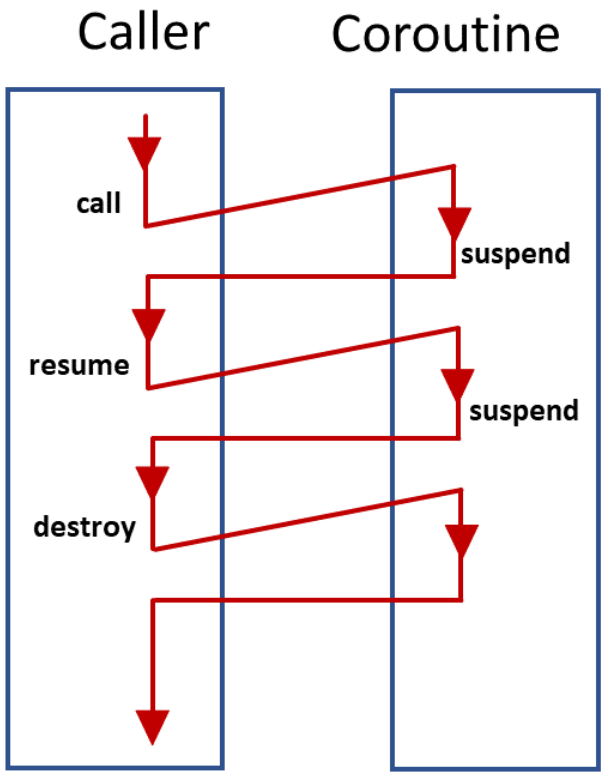
이때 사용하는 것이 Coroutine(코루틴)입니다.

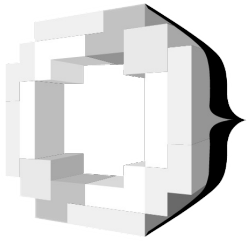


Coroutine

Coroutine이란 순차적으로 계속 진행되는 기존 함수와 달리, 중간에 실행을 멈추고 다시 원래 함수의 흐름으로 돌아갈 수 있는 특별한 함수입니다.





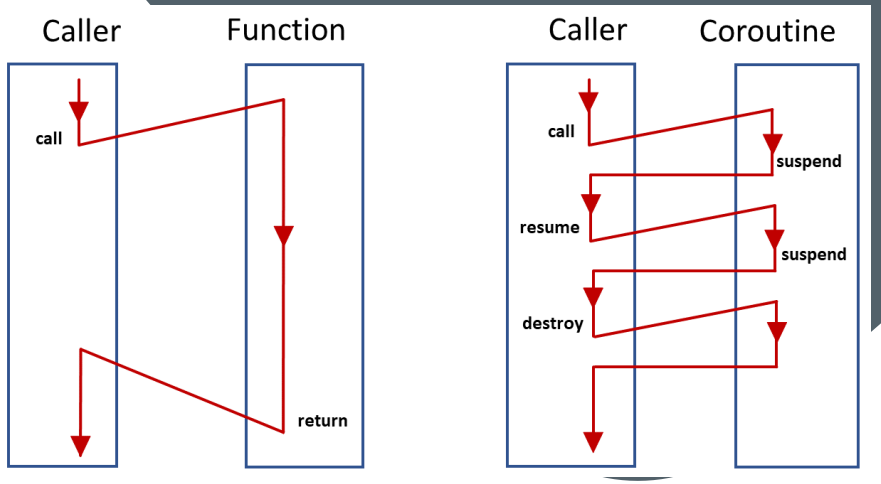


Coroutine

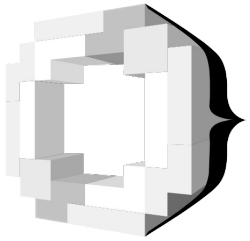
Coroutine 함수는 좀 특이하게 작성합니다.

IEnumerator이라는 반환형을 사용하며, 그냥 return이 아닌 yield return을 사용합니다.
(제어권을 양보!)

완전히 종료하려면 yield break를 사용합니다.
(코루틴의 진짜 종료)




```
IEnumerator test1()
{
    Debug.Log("testing1");
    yield return null;
    Debug.Log("testing2");
    yield return new WaitForSeconds(1f);
    Debug.Log("testing3");
    yield break;
}
```



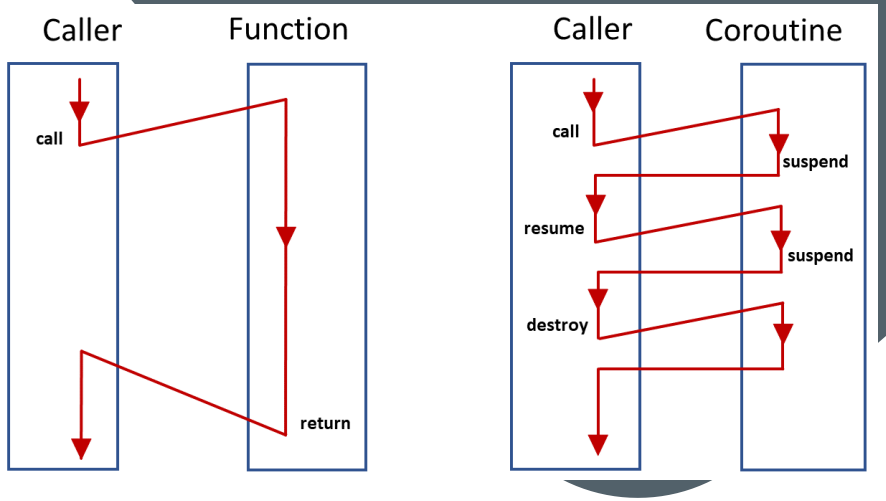
Coroutine

yiedl return <조건>;

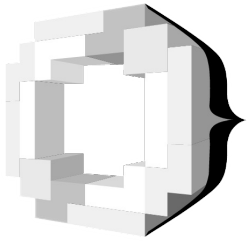
yield return null;
은 Update 함수처럼 바로 다음 프레임에 바로 다시 실행됩니다.

yield return new WaitForSeconds(float);
해당 float 값 만큼의 초를 기다린 후 다시 실행됩니다.

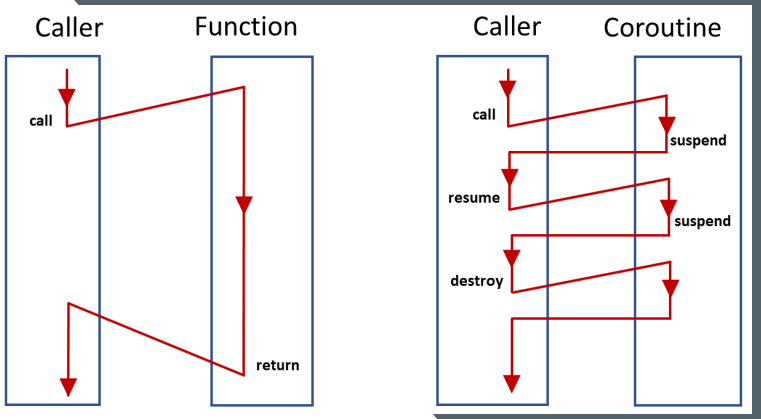
오른쪽의 예제는 testing1이 바로 표시되고, 다음 프레임에 testing2가, 1초 뒤에 testing3이 표시됩니다.




```
IEnumerator test1()
{
    Debug.Log("testing1");
    yield return null;
    Debug.Log("testing2");
    yield return new WaitForSeconds(1f);
    Debug.Log("testing3");
    yield break;
}
```



Coroutine



코루틴은 호출 방법도 특이합니다.

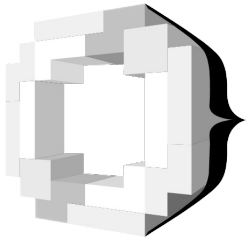
일반 함수처럼 호출하면 안되고, StartCoroutine()이라는 별도의 함수를 사용합니다.

함수의 인자로 함수명 string이나 함수 자체를 넘겨줍니다.

그리고 StopCoroutine()을 통해 해당 코루틴을 조기 종료할 수 있습니다. yield break를 바로 실행하는 것과 같은 효과입니다.

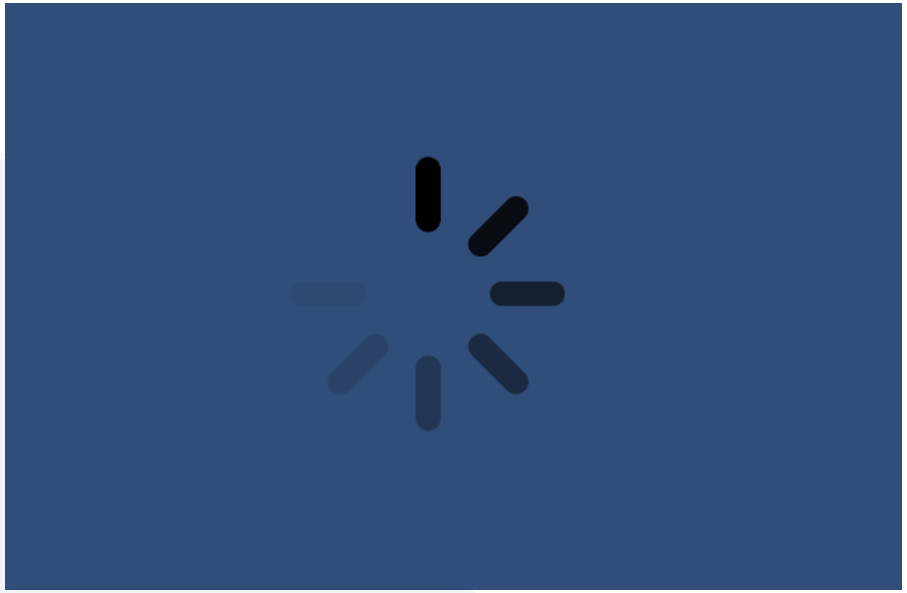
StopAllCoroutine()은 실행 중인 모든 코루틴을 정지합니다.

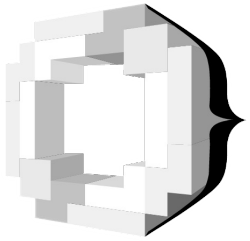
```
void test()
{
    test1(); //Wrong!!

    StartCoroutine("test1");
    StartCoroutine(test1());

    StopCoroutine("test1");
    StopCoroutine(test1());

    StopAllCoroutines();
}
```



Coroutine

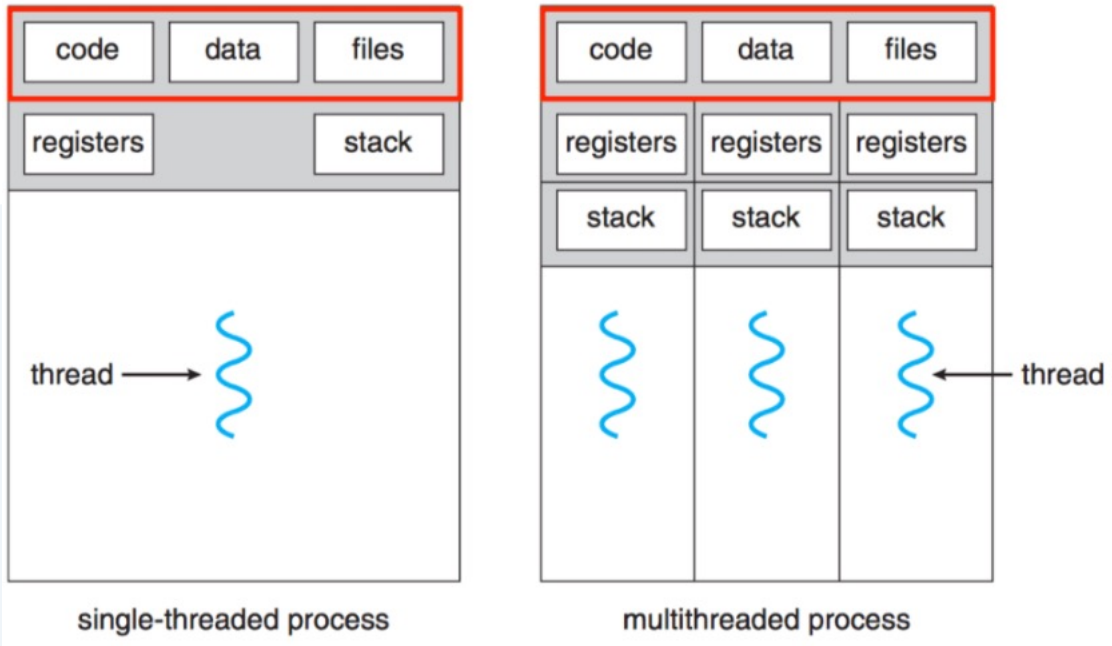
로딩 화면을 예시로 보고 옵시다



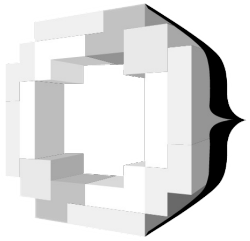


Coroutine? 쓰레드?

코루틴은 멀티 쓰레드와는 조금 다릅니다



결국 따지고 보면 함수 중간에 빠르게 다른 함수로 갔다 올 수 있다 뿐이지 프로그램 단위로 보면 순차적으로 한 줄 한 줄 실행되는 건 똑같습니다.



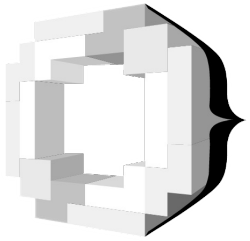
Coroutine? 쓰레드?

유니티에서 멀티 쓰레드는 많이 사용되지 않습니다.
(심지어 공식 문서에서 비추하고 있음)

코루틴을 사용하면 마치 동시에 처리되는 것처럼 보이거든요!

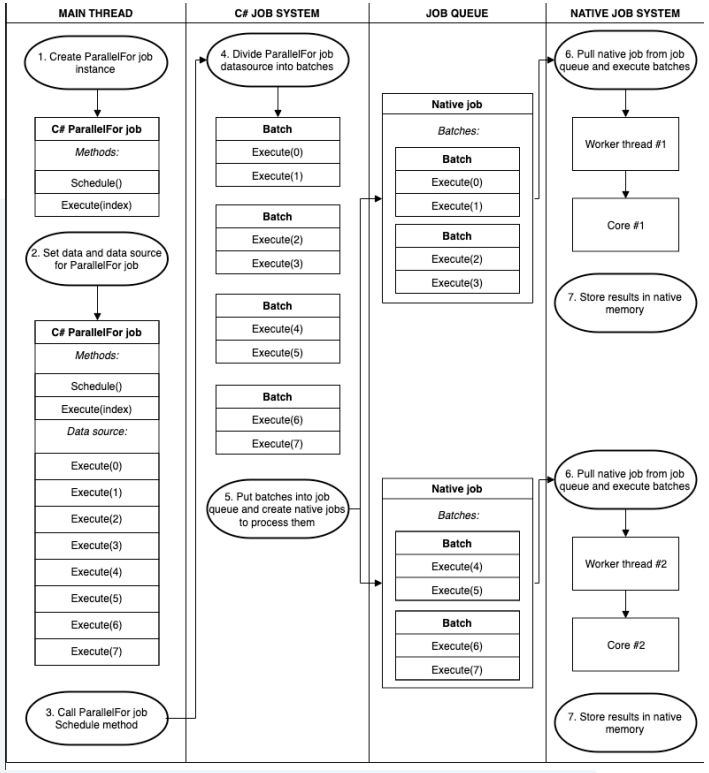
```
IEnumerator VeryVeryHeavyCoroutine()
{
    for (int i = 0; i < 10000; i++)
    {
        VeryHeavyWork();
        yield return null; // 순서 양보하기
    }
    isWorking = false;
}
```

하지만 코루틴 내에 또 무거운 작업이 있다면?

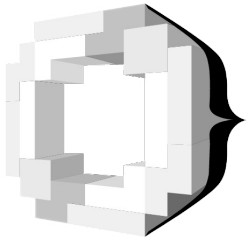


Coroutine? 비동기?

이럴 때 사용하는 방법 중 하나가 멀티 쓰레드입니다. 쓰레드마다 서로 다른 CPU 코어에서 돌아가면 **진짜로 동시에** 작업이 이루어집니다.



그냥 넘어갑시다…



Coroutine

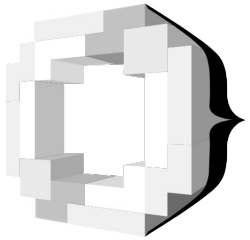
조건이 복잡하거나, 어떤 동작에 이어서 일어나는 동작이라면, coroutine을 해당 조건에 맞춰서 실행하는 것이 좋습니다.

Update에 구현한다면 조건을 매 프레임마다 비교하기 때문에 불필요한 연산이 많아집니다.

피격 시, 스킬 사용 시 등 특별한 조건이 있다면 해당 동작 이후에 코루틴을 호출하는 방법으로 구현하는 것이 더 좋습니다.

```
currentCooldown += Time.deltaTime;
if (currentCooldown > coolTime)
{
    //do something;
    currentCooldown = 0;
}
```

```
void skill()
{
    //do something
    StartCoroutine(cooltimer(4));
}


참조 1개
IEnumerator cooltimer(float coolTime)
{
    Debug.Log("skill cooldown!");
    yield return new WaitForSeconds(4f);
    Debug.Log("skill ready!");
}
```



몬스터 구현하기

데미지: 플레이어에게 닿으면 5000의 데미지를 주고 넉백시킵니다. 플레이어는 0.4초 동안 움직이지 못합니다.

코루틴을 배웠으니 이제 사용해봅시다.

플레이어는 피격 당한 후 0.4초동안 움직이지 못해야 하니,

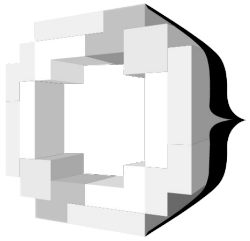
피격을 당했을 때, isHit = true; 로 피격 상태를 설정해주고, 0.4초 후에 isHit = false로 피격 상태를 풀어줍니다.

해당 코루틴을 hit 함수에서 호출합니다.

그리고 Update()의 이동 조건에 !isHit도 추가합시다.

피격상태 중에 피격이 또 되면 코루틴이 꼬이게 되니, 이전 코루틴을 정지해줍시다.

```
public void hit(int direction, float dmg)
{
    life -= dmg;
    rgbody.AddForce(new Vector2(direction * 400, 400));
    if (isHit)
        StopCoroutine("hitting");
    StartCoroutine("hitting");
}

참조 0개
IEnumerator hitting()
{
    isHit = true;
    yield return new WaitForSeconds(0.4f);
    isHit = false;
}
```



몬스터 구현하기

기믹: 플레이어가 머리를 밟으면 플레이어를 위로 점프 시킵니다.
체력: 플레이어가 머리를 밟으면 5의 체력이 감소합니다.

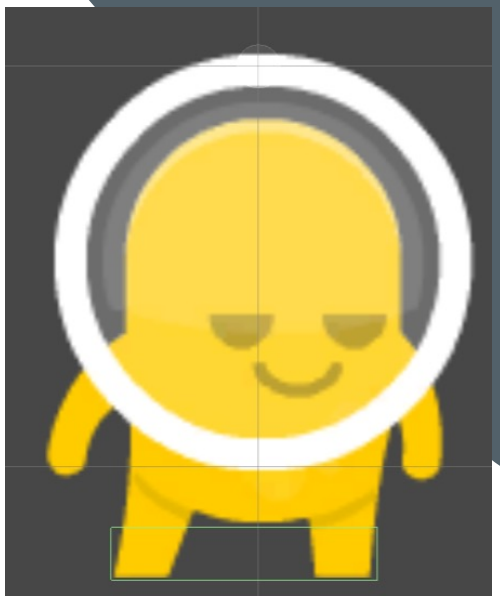
플레이어의 발이 몬스터의 머리를 밟을 때를 감지해야 합니다.

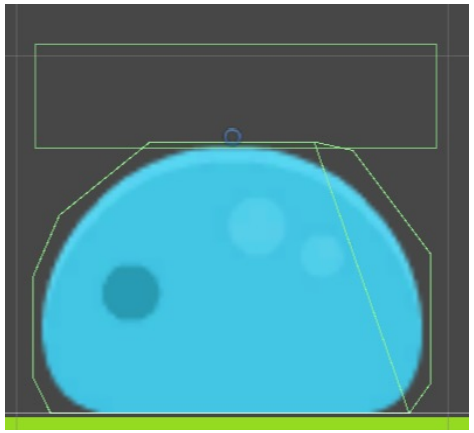
한 오브젝트에 콜라이더를 여러 개 추가하면 단순히 충돌 범위가 늘어날 뿐입니다.

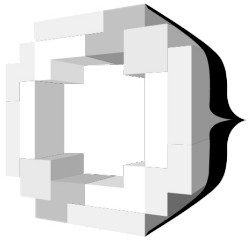
콜라이더에 따라 다른 작동을 원한다면?

-> Empty child를 추가해서 플레이어의 발과 몬스터의 머리에 각각 collider을 trigger로 추가해줍시다.

Tag를 새로 추가해 충돌을 제대로 감지할 수 있게 만들어줍니다.







몬스터 구현하기

기믹: 플레이어가 머리를 밟으면 플레이어를 위로 점프 시킵니다.
체력: 플레이어가 머리를 밟으면 5의 체력이 감소합니다.

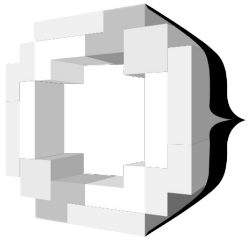
몬스터의 체력 감소와 플레이어의 발사가 필요하기 때문에 스크립트를 어디에 넣든 상관 없습니다. 플레이어 발과 몬스터 머리 양측 다 넣어도 가능합니다.

OnTriggerEnter2D() 함수에 tag를 검사해서 충돌을 확인하고
monsterController에서는 몬스터의 체력을 감소시키는 함수를,
playerController에서는 플레이어를 위로 발사하는 함수를 작성해서 추가합니다. 이때, 서로 충돌한 collision은 monster이나 player이 아닌 monster의 child, player의 child임을 유의합시다.

저는 monsterHead에서 구현했습니다. ---------->

+ 추가로, LaunchUp() 함수에서 현재 점프 수를 초기화 해서, 슬라임을 밟고 점프를 두 번 할 수 있게 만들었습니다.

```
Unity 메시지 | 참조 0개
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.gameObject.CompareTag("PlayerFeet"))
    {
        monsterController.Hit(5);
        collision.GetComponentInParent<playerController>().LaunchUp();
    }
}
```



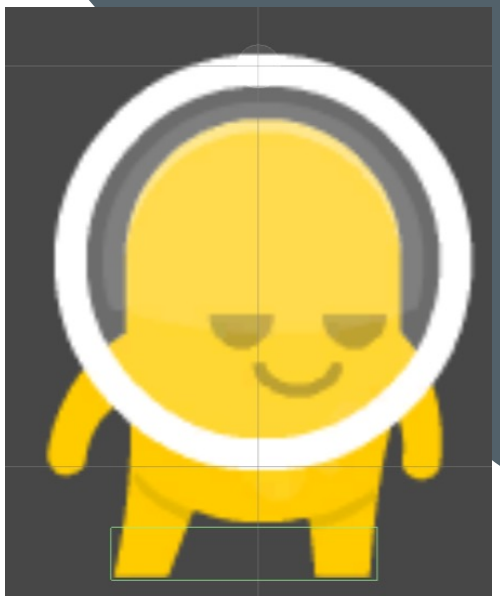
몬스터 구현하기

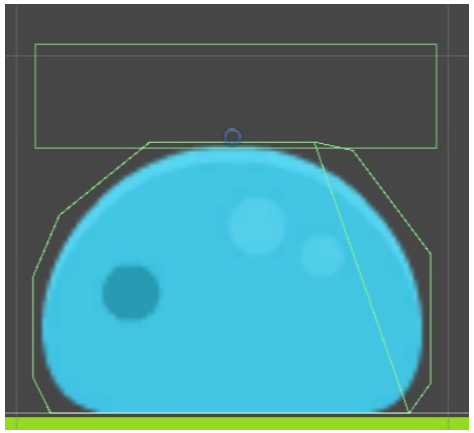
체력: 체력은 8입니다. 1초마다 1의 체력이 감소합니다. 장애물에 닿거나 맵 밖으로 떨어지면 즉사합니다.

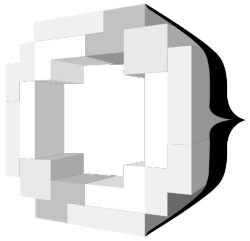
playerController과 같이 maxHp, curHp로 관리하고, Update()함수에서 체력을 시간에 맞춰 감소시킵니다. monsterController에 OntriggerEnter2D()에서 장애물 충돌 처리도 해줍니다.

Update() 함수에서 몬스터 체력을 계속 확인해서 0보다 같거나 작다면 Die() 함수를 호출합니다.

Die() 함수에는 Destroy(gameObject); 로 할까요??





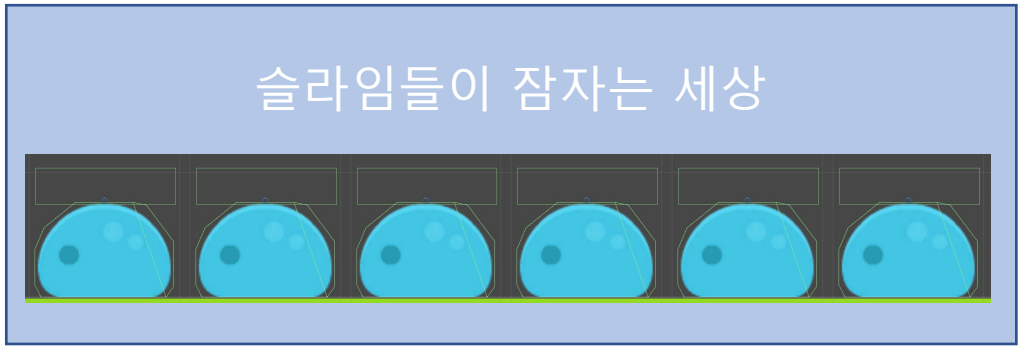


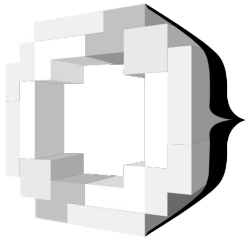
오브젝트 풀링으로 몬스터 생성하기

몬스터를 매번 Instantiate 생성하고 Destroy로 삭제하는 것은 부하가 좀 큽니다.

게다가 방금 사라진 몬스터를 금방 다시 내놓는다면 굳이 몬스터를 삭제할 필요까지는 없죠.

그냥 몬스터를 비활성화 해놓고, 몬스터가 필요할 때 몬스터의 체력만 초기화 해서 활성화 시키면 플레이어 입장에선 똑같지 않을까요?

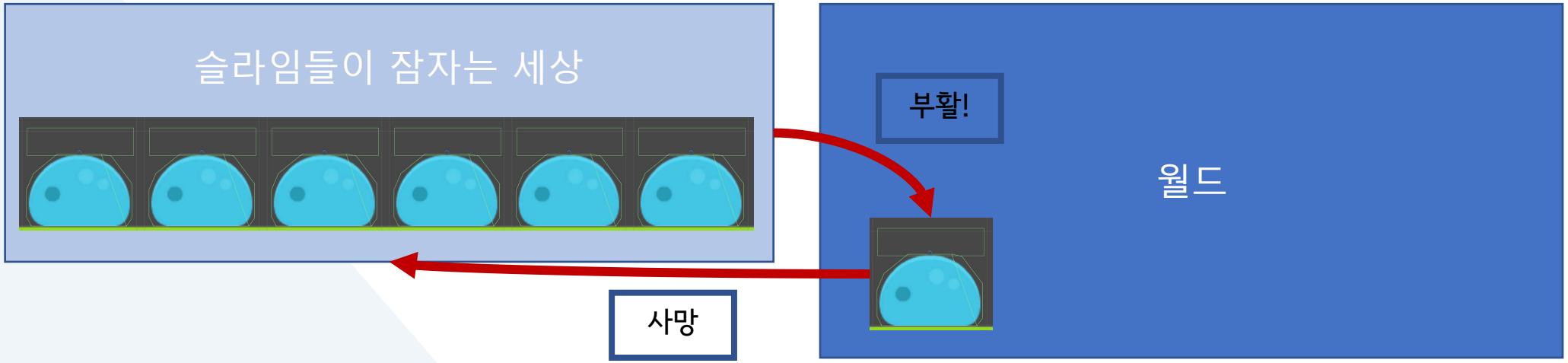


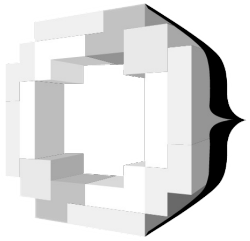


오브젝트 풀링으로 몬스터 생성하기

이 방법을 오브젝트 풀링이라고 합니다.

미리 몬스터를 잔뜩 생성해놓고,
비활성화 -> 초기화 및 활성화(생성) -> 비활성화(사망) -> 초기화 및 활성화(재생성)
를 계속 반복하는 방법입니다.

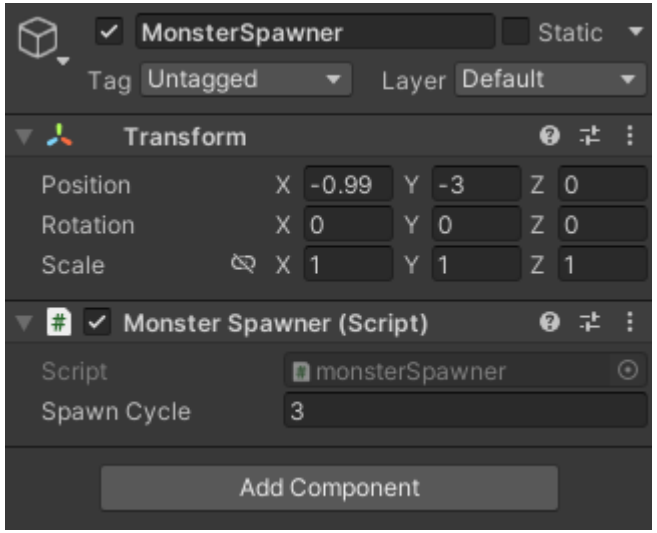


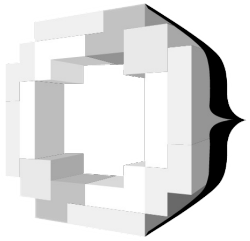


오브젝트 풀링으로 몬스터 생성하기

MonsterSpawner이라는 빈 오브젝트를 생성해서 몬스터를 생성하고 싶은 곳에 갖다놓습니다.

monsterSpawner 스크립트를 만들어서 달아줍시다.





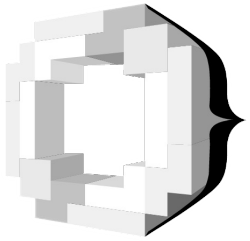
오브젝트 풀링으로 몬스터 생성하기

미리 몬스터를 생성해놓고, 비활성화 -> 초기화 및 활성화(생성) -> 비활성화(사망) -> 초기화 및 활성화(재생성)

우선 몬스터 프리팹을 불러옵시다. Public으로 가져와도 되고, Resources.Load()를 통해 리소스 폴더에서 파일 위치를 명시해 가져와도 됩니다.

생성할 오브젝트(몬스터)를 담아 둘 큐, Queue<GameObject> 타입의 monsterQueue를 생성합시다.

그 다음 Instantiate(prefab, location)로 오브젝트를 생성합니다. 생성한 오브젝트는 monsterQueue에 담아두고, SetActive(false)로 비활성화를 해놓습니다.



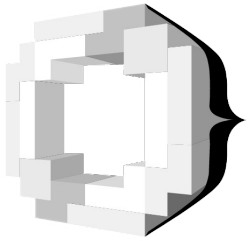
오브젝트 풀링으로 몬스터 생성하기

미리 몬스터를 생성해놓고, 비활성화 -> 초기화 및 활성화(생성) -> 비활성화(사망) -> 초기화 및 활성화(재생성)

몬스터 생성 함수 spawnMonster()을 만듭시다.

우선 monsterQueue.Dequeue()로 오브젝트를 큐에서 빼옵니다.
해당 오브젝트에 .SetActive(true)를 해줍니다.

위치를 좀 랜덤하게 결정하기 위해서 x 좌표에다가 Random.Range(-x, x) 를 더해줍니다.

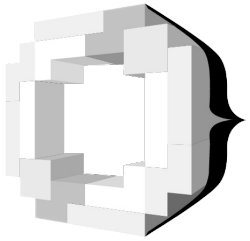


오브젝트 풀링으로 몬스터 생성하기

미리 몬스터를 생성해놓고, 비활성화 -> 초기화 및 활성화(생성) -> 비활성화(사망) -> 초기화 및 활성화(재생성)

초기화는 여기서 해줘도 되지만, monsterController에서 OnEnable() 함수를 사용하면 더 간단합니다. OnEnable()함수는 해당 오브젝트가 활성화 됐을 때 작동하므로, monsterSpawner이 해당 몬스터를 활성화 하면 자동으로 실행됩니다.

OnEnable() 함수에서 몬스터의 현재 체력을 최대 체력으로, 현재 이동 쿨타임을 0초로 초기화 해줍니다.



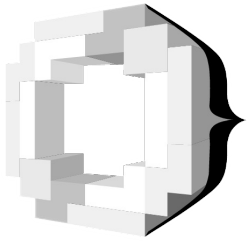
오브젝트 풀링으로 몬스터 생성하기

미리 몬스터를 생성해놓고, 비활성화 -> 초기화 및 활성화(생성) -> 비활성화(사망)
-> 초기화 및 활성화(재생성)

spawnMonster() 함수를 완성했으니, Update() 함수에서 호출만 해주면 됩니다.

monsterQueue.Count는 현재 몬스터큐에 생성할 오브젝트가 얼마나 있는지 나타내는 숫자입니다.
if (monsterQueue.Count > 0) 을 하면 생성할 몬스터가 있을 때마다 작동하겠죠?

만일 시간 차(주기)를 좀 주고 싶다면, 앞서 설명한 Cooltime 방식이나 Coroutine 방식으로 구현할 수 있습니다.



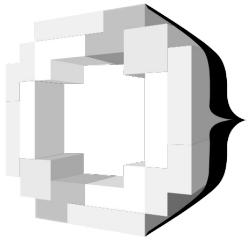
오브젝트 풀링으로 몬스터 생성하기

미리 몬스터를 생성해놓고, 비활성화 -> 초기화 및 활성화(생성) -> 비활성화(사망) -> 초기화 및 활성화(재생성)

몬스터가 사망했을 때 호출 할 함수 monsterDead(GameObject monster) 함수를 만듭니다.

몬스터가 사망했다면 비활성화 후 다시 몬스터 큐에 집어넣어야 되겠죠?

monster.SetActive(false)로 비활성화 한 후,
monsterQueue.Enqueue(monster) 함수로 큐에다 오브젝트(몬스터)를 집어넣습니다.

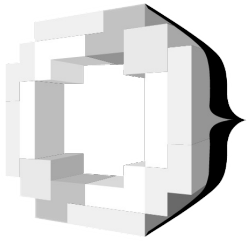


오브젝트 풀링으로 몬스터 생성하기

미리 몬스터를 생성해놓고, 비활성화 -> 초기화 및 활성화(생성) -> 비활성화(사망)
-> 초기화 및 활성화(재생성)

몬스터 사망은 monsterController에서 감지하고 Die() 함수에서 처리합니다.

여기서 monsterSpawner의 monsterDead(gameObject) 함수를 호출하겠습니다.

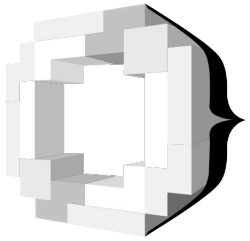


오브젝트 풀링으로 몬스터 생성하기

미리 몬스터를 생성해놓고, 비활성화 -> 초기화 및 활성화(생성) -> 비활성화(사망) -> 초기화 및 활성화(재생성)

Die() 함수에서 해당 몬스터를 monsterSpawner.monsterDead(gameObject)로 넘겨줍니다.
그러면 monsterSpawner에서 해당 몬스터를 비활성화 하고, 다시 생성할 몬스터 큐에 넣습니다.

불안하다면 Die() 함수에서 미리 비활성화를 해줘도 좋습니다.



오브젝트 풀링으로 몬스터 생성하기

미리 몬스터를 생성해놓고, 비활성화 -> 초기화 및 활성화(생성) -> 비활성화(사망) -> 초기화 및 활성화(재생성)

여기까지 왔다면, 재생성은 자동으로 됩니다.
몬스터가 죽으면 다시 비활성화 되어 큐에 들어가고, Update() 함수에서 큐에 생성할 몬스터가 있다면 자동으로 다시 몬스터를 생성합니다.

축하드립니다! 무한 츠쿠요미의 몬스터 스포너를 완성하셨습니다.