**유니티 6장**

**6장에서는** 유한상태머신을 이용해 적 캐릭터의 인공지능을 구현하고 유연한 애니메이션을 위한 메카님 애니메이션 시스템에 대해 실습, 내비게이션 시스템을 통해 이동 및 추적 로직을 구현

적 캐릭터는 NPC(Non-Player Character)의 일종으로 게임에 참여한 플레이어가 조작할 수 없는 캐릭터를 말함. 자신만의 인공지능(AI)를 가지고 있어 게임 속 플레이어와 유기적인 플레이를 할 수 있게 제작해야 현실감 있는 게임이 되겠져

**!유한 상태 머신의 정의**

NPC의 인공지능(AI)을 구현하는 방식에는 여러 가지가 있다.

그 중 하나가 유한 상태 머신(FSM; Finite State Machine)으로 적 캐릭터가 스스로 알아서 주변 환경에 적응하거나 들어오는 반응에 적절하게 반작용하도록 구현한 것을 일컫는다. 적 캐릭터가 생성(Spawn)되고, 일정 범위를 순찰하다가 주인공을 추적하고 사정거리 이내에 근접하면 공격하는 것과 같은 구조를 상태 머신이라고 한다. 또한 피격을 당해 일정 데미지가 누적되면 사망하고 소멸하는 구조이기에 상태가 유한해서 “유한 상태 머신”이라고 한다.

이 챕터에서 구현할 적 캐릭터는 생성되면 주변을 순찰하고, 플레이어를 만나면 추적을 시작하며, 사정거리 내에 근접하면 플레이어를 공격한다. 또한 데미지를 입고 생명이 다하면 사망하는 FSM을 적용해본다.

307p 참고

FSM의 단점이라면 상태가 많아질수록 상태와 상태 간의 연결(Transition)이 복잡해지고 코드의 확장과 유지 보수가 어려워지는 것. 이러한 점을 개선하기 위해 상태를 모듈화하고 계층적으로 분류하는 계층적 유한 상태 머신(HFSM; Hierachical Finite State Machine) 방식을 도입할 수 있다. 또한, 최근에는 행동 트리(BT; Behaviour Tree) 방식도 개발에 많이 적용한다.

**메카님**

FSM의 상태에 따라 적절한 애니메이션 동작을 취할 수 있게 메카님(Mecanim)을 사용해 애니메이션을 제어해보자. 메카님은 유니티 4.0부터 보인 기능으로, 애니메이션 미들웨어 엔진의 일종이다. 기존의 레거시 애니메이션에서는 처리하기 어려웠던 유연한 애니메이션을 처리할 수 있게 해줌

메카님은 리타게팅 기능을 이용해 인체형(Humanoid) 모델의 필수 본(Bone, 뼈대) 구조와 일치하면 다른 모델의 애니메이션이나 모션 캡처 애니메이션을 바로 적용할 수 있다. 따라서 타 모델에 적용한 애니메이션을 재사용할 수 있고, 모델에 고품질의 모션 캡처 동작을 적용 가능해 애니메이션 퀄리티를 높일 수 있다는 장점이 있다.

- 노드 기반의 Visual Editor

- 세밀한 애니메이션 처리

- 리타게팅을 통한 애니메이션 적용

메카님은 시각적인 설계 방식(WYSIWYG)의 에디터를 제공한다. 따라서 애니메이션 클립 간의 관계 설정을 쉽게 할 수 있고, 복잡한 애니메이션 설계가 가능하다. 또한 애니메이터 뷰 자체가 FSM의 설계도이기도 하다.

자.. 이제 만들어보자. 309~

메카님 애니메이션 전환? Project 창에서 Monster 원본 클릭, Rig 선택하면

Animation type이 보임

|  |  |
| --- | --- |
| **애니메이션 타입** | **특성** |
| 레거시 | 유니티 4.0 이전에서 사용하던 애니메이션 시스템 |
| 제너릭 | 메카님 애니메이션(비인간형 모델), 리타게팅 X |
| 휴머노이드 | 메카님 애니메이션(인간형 모델, 2족 보행 모델), 리타게팅 O |

적 모델은 2족 보행 모델이니까 휴머노이드 하면 댐. 그 후 Apply 버튼 클릭.

이 때 해당 모델의 본 구조를 분석해 자동으로 매핑하는 작업을 진행하며 매핑 작업이 완료되면 Configure 버튼이 활성화된다. 정상적으로 매핑이 완료되면 configure 앞에 체크표시 뜸.

휴머노이드 타입의 본 구조가 아니면 X가 뜸. 경우에 따라 수동으로 본을 연결해야 될지도

Per-Muscle Settings에서 각 관절이 회전할 수 있는 최소 각도와 최대 각도의 범위를 설정하는 기능을 제공한다. 부채꼴로 표시되는 각도의 범위는 RGB 색상으로 해당 관절의 회전축을 의미한다. 조회 및 설정을 완료했으면 인스펙터 뷰 아래에 있는 Done 버튼을 누르면 댐

**애니메이션 클립 속성**

[Animation] 탭을 클릭하면 해당 모델이 가진 애니메이션 목록이 나열돼 있고, 각 애니메이션 클립의 속성을 수정할 수 있다. Loop Time 속성은 애니메이션 클립을 무한 반복하는 옵션이고 Loop Pose는 애니메이션을 반복할 때 끈김 없이 부드럽게 해주는 옵션이다. 각 애니메이션 클립은 시작 프레임과 종료 프레임을 설정할 수 있으며, 내가 쓸 몬스터 모델은 이미 설정되어 있다.

**애니메이션 리타게팅 기능**

리타게팅이 대체 뭐냐? 다른 애니메이션 동작을 가져와 재사용하는 기능!을 말함. 적 캐릭터 역시 Humanoid 타입의 모델이므로 다른 모션 캡처 애니메이션을 가져와 적용할 수 있다.

<http://www.mixamo.com> << 다양한 애니메이션과 리깅을 직접 할 수 있는 서비스 제공하는 사이트 ㄷㄷ 쩐당. 애니메이션 파일 확장자는 name.fbx로 끝나나보다. 316p 참고

Animator 컴포넌트

기본적인 메카님속성은 설정했으니께 프로젝트 뷰의 Monster 모델을 씬 뷰 또는 하이러키 뷰로 드래그해 씬에 배치하자. 몬스터 모델은 기본적으로 Animator 컴포넌트를 포함하고 있다.

(복습! Animation 컴포넌트는? 레거시. Animator 컴포넌트는? 메카님이다~)

Animator 컴포넌트의 각 속성은 다음과 같다 :

|  |  |
| --- | --- |
| **속성** | **설명** |
| **Controller** | 애니메이션 클립 간의 연결 정보를 저장한 Animator Controller를 연결 |
| **Avatar** | 해당 모델의 본 매핑 정보를 저장한 Avatar 에셋을 연결 |
| **Apply Root Motion** | 애니메이션의 갱신과 Time Scale의 적용 여부에 대한 설정  \* Normal : Update 주기와 동기되며 애니메이션의 속도는 Time Scale의 영향을 받는다.  \* Animate Physcis : FixedUpdate 주기와 동기되며 Rigidbody와 상호작용을 할 때 사용한다.  \* Unscaled Time : Update 주기와 동기되며 애니메이션 속도는 Time Scale에 영향을 받지 않는다. |
| **Culling Mode** | 애니메이션 클립의 동작 정지에 대한 설정  \* Always Animation : 항상 애니메이션 실행하는 옵션, 렌더링을 하지 않아도(카메라 영역 밖이어도) 동작한다.  \* Cull Update Transforms : 렌더링을 하지 않을 때 Transform, IK, Retarget 기능을 정지한다.  \* Cull Completely : 렌더링을 하지 않으면 애니메이션의 모든 기능을 정지한다. |

**애니메이터 컨트롤러**

적 캐릭터에 애니메이션 기능을 추가해보자. 메카님으로 변환한 모델에 애니메이션 기능을 부여하려면 애니메이터 컨트롤러(Animator Controller)가 필요하다. 애니메이터 컨트롤러는 메카님을 적용한 모델에 애니메이션을 적용하고 특정 조건에 따라 다른 애니메이션(State)로 전이(Transition)하는 규칙(Role)을 설계할 수 있는 인터페이스를 제공한다.

이 애니메이션 컨트롤러는… 매우 익숙하다. 그 설계도. 같다. 블루프린트같다.

기본적으로, 알트 키를 눌러야 맵 전체를 둘러볼 수 있다. 아니면 휠키. A를 누르면 전체 스테이터스를 한번에 볼 수 있다.

|  |  |
| --- | --- |
| **스테이트 종류** | **역할** |
| **Entry** | 시작 스테이트로 최초의 진입점이다. |
| **Exit** | 종료 스테이트로 모든 스테이트가 종료되는 마지막 스테이트다. |
| **Any State** | 현재 어느 스테이트를 실행하고 있더라도 조건에 만족하면 다른 스테이트로 분기시킬 수 있다. |

애니메이터 위에 애니메이션 올리는 건 그냥 드래그하면 된다. 처음에 추가한 애니메이션 클립은 주황색인데, 이게 해당 모델에서 제일 처음 실행되는 기본 애니메이션 클립이라는 뜻이며, 바꾸려면 애니메이션 클립을 오른클릭해서 Set as Layer Default State 를 클릭하면 댐.

**스테이트 전이와 파라미터**

기본 애니메이션인 idle에서 walk로 전이(Transition)가 발생하려면 두 스테이트 사이를 연결해야 한다. Idle을 마우스 오른클릭한 뒤 팝업된 메뉴에서 Make Transition을 선택하면 다른 스테이트로 연결할 수 있는 흰색 연결선이 나온다. 이 선을 walk 스테이트 위로 가져가 클릭하면 두 스테이트가 연결될 것이다.

자, 이제 walk 스테이트로 가기 위한 조건을 만들자. 변수를 만들어야지. Parameter 바를 누르고 필요한 변수 타입과 어떻게 조건을 걸 것인지 잘 생각해보고 만들자. ㄱㄱㄱ

isTrace가 true면 walk 스테이트로 전이되고 false면 walk로 돌아오게 해보자.

화살표(이름은 transition이다)를 클릭하면 인스펙터에서 조건 설정 창이 뜰 것이다.

Has Exit Time을 체크하면 isTrace 변수가 true가 되어도 현재 실행하는 Idle 애니메이션을 다 수행하고 나서 walk 스테이트로 전이된다.

따라서? **조건을 만족했을 때 바로 전이해야 하는 상황에선 Has Exit Time을 언체크**한다.

**네비게이션 – 적 캐릭터의 순찰 및 추적**

장애물이 없는 곳에서 특정 목적지까지 가는 것은 그리 어렵지 않다. 목적지를 향해 회전하고 직진하는 것으로 그만이다. But 장애물이 있으면 달라지겠져. 이러한 로직을 구현하기 위해 추적 또는 길 찾기 알고리즘이 필요하다.

길 찾기 알고리즘은 여러 방식이 있지만, 그 중 가장 많이 알려진 알고리즘은

A\* PathFinding이다. 이 알고리즘은 유니티에서 네비게이션 기능을 제공하기 이전에 많이 사용하던 알고리즘 중 하나였다. 에셋 스토어에 있다. 100달러다. ㅈㄴ비싸네

유니티는 3d 모델을 분석해 추적할 수 있는 네비게이션 기능을 제공한다. 이 기능을 이용해 추적하는 로직을 구현해보자. 애니메이션을 walk 애니메이션으로 변경하고 플레이어에 근접했을 때는 attack 애니메이션으로 변경되도록 메카님을 구성해본다.

유니티에서 제공하는 네비게이션의 구현 방식은 스테이지를 구성하고 있는 3d 메시를 분석해 내비메시 데이터를 미리 생성하는 방식이다. 즉, 추적할 수 있는 영역(Walkable)과 장애물로 판단해 지나갈 수 없는 영역(Non Walkable Area)의 데이터를 메시로 미리 만드는 것이다. 또한, 높은 곳에서 낮은 곳으로 연결하는 오프 메시 링크(Off Mesh Link) 기능을 이용해 뛰어내리는 동작을 연출할 수 있다. 이렇게 유니티 에디터 모드에서 미리 빌드(베이크라고도 함)해서 내비메시를 생성한 다음 런타임에서 그 정보에 따라 최단 거리를 계산해 추적 또는 이동할 수 있게 한다.

**네비게이션 설정 – Navigation Static Flag**

네비게이션을 베이크(Bake)하려면 그 대상이 무엇인지 지정해야 한다. 우선 바닥으로 사용되는 Floor의 Static 옵션을 설정한다. 그면 전부 스태틱 처리된다. 유니티에서는 기능별로 Static 옵션을 설정할 수 있다! 이름 옆에 보면 Static 칸이 있다. 드럼통도 마찬가지.

내비메시를 베이크할 때 필요한 정보는 해당 게임오브젝트의 메시 정보다. Barrel의 Mesh Filter 컴포넌트는 하위에 있는 Barrel에 추가돼 있기 때문에 Navigation Static 플래그를 하위의 객체까지 모두 적용한 것이다. 따라서 Mesh Filter 컴포넌트가 상위 객체에 있다면 No, this object only 를 선택한다.

(복습! Static이란? 유니티 엔진에 고정된 게임오브젝트임을 알려주는 옵션! 드럼통의 Static 옵션을 모두 선택하면 움직이지 않고 고정되어 폭발 시 날아올라가지 않는다. 따라서 필요한 Static 옵션만 선택해야 한다)

**네비메시 베이크**

메뉴 Window 🡪 AI 🡪 Navigation 을 선택하면 네비게이션 뷰가 나타남. Bake -> bake 버튼을 누르면 네비메시가 베이크됨.

**NavMeshAgent 컴포넌트**

NavMEshAgent 컴포넌트는 네비메시 데이터를 기반으로 목적지까지 최단 거리를 계산해 이동하는 역할을 하며, 장애물과 다른 NPC 간의 충돌을 회피하는 기능도 제공한다. 또한 NavMeshAgent 컴포넌트는 실시간으로 최단 거리를 계산할 때 A\* PathFinding 알고리즘을 사용한다.

몬스터에 NavMeshAgent를 추가해야 함. NavMeshAgent 속성도 알아보자!

|  |  |
| --- | --- |
| **속성** | **설명** |
| **Agent Type** | 장애물을 회피하기 위한 회전 반경 및 넘어갈 수 있는 계단의 높이, 경사로의 등판 각도의 정보를 정의한 것을 에이전트라고 한다. 기본값은 Humanoid로 지정돼 있으며 개발자가 추가로 정의해 지정할 수 있다. |
| **Base Offset** | 실린더 형태로 표현된 NavMeshAgent는 베이크된 내비메시 표면에 붙어서 이동한다. 이때 캐릭터가 바닥에서 뜬 상태로 이동하는 것처럼 보이는 것을 수정하기 위해 NavMeshAgent의 높낮이를 조절하는 기능이다. |
| **Speed** | 최대 이동속도(초당 월드 단위) |
| **Angular Speed** | 최대 회전속도(초당 각도) |
| **Acceleration** | 최대 가속(m/s2) |
| **Stopping Distance** | 목표 지점에 가까워졌을 떄 정지하는 근접 거리 |
| **Auto Braking** | 목표 지점에 가까워졌을 때 속도를 줄이는 기능으로 순찰과 같이 여러 포인트를 부드럽게 이동하기 위해서는 이 기능을 비활성화해야 한다. |
| **Radius** | 에이전트의 반경으로 장애물을 크게 돌아갈 것인지, 아니면 짧게 돌아갈 것인지 결정한다. 또한, 다른 에이전트 간의 충돌을 계산하기 위해 사용한다. |
| **Height** | 에이전트가 머리 위에 있는 장애물을 지나갈 때의 높이를 의미한다. 또한, 이 수치가 클수록 넘어갈 수 있는 계단의 스텝(Step)도 커진다. |
| **Quality** | 장애물을 회패할 때의 품질을 설정하는 것으로 많은 에이전트가 있으면 품질을 낮춰서 CPU 부하를 줄일 수 있다. |
| **Priority** | 에이전트 간의 회피 우선순위를 결정하는 것으로 자신보다 낮은 에이전트는 회피대상에서 제외한다. 0~99 사이의 값을 가지며 낮은 수가 높은 우선순위다. |
| **Auto Traverse Off Mesh Link** | 분리된 메시 간에 자동으로 링크를 생성하는 옵션이다. Off Mesh Link 컴포넌트를 이용해 직접 링크를 만들 때 이 기능은 비활성화해야 한다. |
| **Auto Repath** | 이 속성을 활성화하면 이동할 경로가 유효하지 않을 때 다시 경로를 재탐색한다. |
| **Area Mask** | Area Mask를 지정한 후 내비메시를 베이크하면 특정 영역별로 이동을 제한할 수 있는 기능으로 특히 런타임에서 비트 마스크를 통해 조작할 수 있다. |

Nav Mesh Agent 컴포넌트의 Stopping Distance 속성은 추적목표를 추적하다가 정지하는 근접거리를 말한다. 이 속성을 0으로 설정하면 목표물과 겹쳐 버린다. 따라서 추적하다가 일정 거리만큼 근접한 이후에는 멈춰야 하므로 2 정도 설정 ㄱㄱㄱ

몬스터가 스테이지에 생성됨과 동시에 추적 대상인 Player의 위치 정보를 가져오기 위해 태그를 생성해 지정한다. 새로운 태그 Player를 생성하고 하이러키 뷰의 Player를 선택한 다음 PLAYER 태그로 지정 rr

… 코드

// 네비게이션 기능을 사용하기 위해 추가해야 하는 네임스페이스

using UnityEngine.AI;

public class MonsterCtrl : MonoBehaviour

{

    private Transform monsterTr;

    private Transform playerTr;

    private NavMeshAgent agent;

    void Start() {

        // 몬스터의 Transform 할당

        monsterTr = GetComponent<Transform>();

        // 추적 대상인 Player의 Transform 할당

        playerTr = GameObject.FindWithTag("PLAYER").GetComponent<Transform>();

        // NavMeshAgent 컴포넌트 할당

        agent = GetComponent<NavMeshAgent>();

        // 추적 대상의 위치를 설정하면 바로 추적 시작

        agent.destination = playerTr.position;

    }

}

일단 네비게이션 기능을 이용하기 위해 UnityEngine.AI 패키지(네임스페이스)를 불러와야 한다.

추적 대상인 주인공 캐릭터의 Transform 컴포넌트는 다른 게임오브젝트의 컴포넌트다. 따라서 먼저 해당 게임오브젝트를 검색해서 찾는다. GameObject.FindWithTag(“태그명”) 함수는 하이러키 뷰에 존재하는 게임오브젝트 중 해당 태그로 지정된 게임오브젝트를 찾는다. 태그명의 대소문자가 다르거나 오탈자가 있을 때나 Player 게임오브젝트에 “PLAYER” 태그를 지정하지 않았을 때 “Null Refernece …” 오류가 발생할 수 있다.

! FindGameObjectWithTag, FindGameObjectsWithTag와 같이 Find~ 계열의 함수는 처리 속도가 느리기 때문에 Update 함수에선 사용하지 않는다. 따라서 Awake, Start 함수에서 변수에 할당한 후 사용하길 권장한다.

Agnet 변수는 NavMeshAgent 컴포넌트를 저장하며 destination 속성을 사용해 이동할 좌푯값을 지정한다. Destination 속성 외에도 SetDestination 함수도 사용할 수 있다.

agent.SetDestination(playerTr.position);

MonsterCtrl 스크립트를 몬스터에 추가해서 잘 쫓아오나 보자.

**유한 상태 머신 구현**

FSM은 자신의 상태 값을 갖고 있어야 하며 현재 어떤 상태인지 갱신하고 해당 상태에 맞는 행동을 취해야 한다. 적 캐릭터의 상태를 순찰, 추적, 공격, 사망으로 정의해보자.

적 캐릭터의 행동 시나리오는 처음 생겼을 때 불규칙적으로 순찰하다가 주인공 캐릭터와 근접하면 추적을 시작, 공격 사정거리 이내로 들어오면 주인공을 향해 공격한다. 따라서 적 캐릭터와 주인공 캐릭터의 거리를 측정해 적 캐릭터의 상태를 주기적으로 업데이트하고, 해당 상태에 맞는 동작을 취하게 해야 한다. 이러한 로직을 Update 함수에서 처리할 수도 있지만 \매 프레임 실행하는 것은 성능상 오버헤드를 줄 수 있으므로 0.2초 또는 0.3초 간격으로 적 캐릭터의 상태를 조사해도 이상 없는 로직이라면 코루틴(Coroutine) 함수를 적극활용하자.

**적 캐릭터의 상태 체크**

적 캐릭터의 상태는 0.3초 간격으로 체크해 상태를 갱신한다. 주인공 캐릭터와 자신의 거리를 측정해 추적 사정거리와 공격 사정거리 이내에 들어왔는지 판단한 다음 상태를 변경한다.

이제 행동구현 하고… 애니메이션까지 동기화하는 코드들 전부 작성중.

|  |  |
| --- | --- |
| **Set 함수** | **Get 함수** |
| **Animator.SetBool** | Animator.GetBool |
| **Animator.setFloat** | Animator.GetFloat |
| **Animator.SetInteger** | Animator.getInteger |
| **Animator.setTrigger** | Animator.getTrigger |

애니메이터 뷰에서 생성한 파라미터는 다음 함수를 이용해 접근할 수 있다.

공격루틴까지 만들었다.

자 이때 애니메이터 뷰에 정의한 파라미터에 접근하는 방식을 String에서 hash로 변경했다**(StringtoHash)**. 왜냐면 글자로 검색하는 건 속도면에서 불리하고 애초에 애니메이터가 해시 테이블에서 뽑아오기 때문에 경제&속도 측면에서 좋다.

**몬스터 피격 리액션**

피격 리액션을 구현해보자. 와.. 오늘 좀 빡세다. 오늘은 23년 2월 14일이다. ㅅㅂ. 내 발렌타인데이.

충돌 감지를 위해 Collider가 필요하다. OnCollisionEnter로 구현한다. 전에 드럼통 구현할 때처럼.

**Resources 폴더**

지금까지는 프리팹을 사용하기 위해 변수를 선언하고, 해당 프리팹을 유니티 에디트 모드에서 미리 연결해 사용했다. 이번에는 기존에 사용했던 방식과 달리 사용하려는 프리팹을 런 모드에서 스크립트를 통해 로드하는 방법을 사용해보자. 스크립트에서 프리팹 또는 에셋을 직접 로드하려면 해당 프리팹이나 에셋이 Resources 폴더 하위에 있어야 한다. 프로젝트 뷰에서 새로운 폴더를 생성하고 이름을 “Resources”로 지정한다. 이 폴더에는 유니티에 예약된 폴더로서 대소문자를 정.확.히 지켜야 한다.

Resources 폴더를 생성한 후 BloodSprayEffect 프리팹을 Resources 폴더로 드래그해 옮긴다. 새로운 프리팹으로 만들겠냐는 팝업창이 나오면 Original Prefab 버튼 클릭.

**적 캐릭터의 공격 능력**

앞서 몬스터가 플레이어와 일정 거리 이내로 좁혀지면 공격 애니메이션을 수행하는 로직을 구현했다. 이제 실제로 주인공 캐릭터가 공격 데미지를 입어 피딿는 로직을 구현해보자.

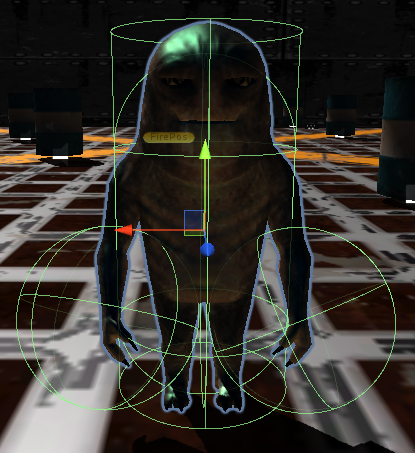
몬스터가 주인공 캐릭터에 데미지를 입히는 방법은

1. 크게 물리적인 충돌을 통해 직접 데미지를 전달하는 방식

2. 일정한 간격으로 공격 데미지를 수치상으로 주는 방법

이 있다. 내가 쓴 몬스터는 공격 상태가 되면 양손으로 타격하는 애니메이션이 있으니 전자 방식으로 구현해보자.

물리적인 충돌 발생을 위해 몬스터의 양쪽 손에 Collider와 Rigidbody 컴포넌트를 추가한다. 하이러키 뷰의 Monster를 선택, 본 구조에서 손목 찾고 양손에 Sphere Collider와 Rigidbody를 하나씩 추가하자.



콜라이더 모양이 아주 요상하다. ㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋ 양손 모두 콜라이더랑 Rigidbody 추가하고, is trigger랑 is kinematic 체크해라. 그다음에 Tag 바꿔라. PUNCH로

플레이어도 충돌을 인식해야 하니까 콜라이더랑 rigidbody 추가하고.

PlayerCtrl에서 콜라이더 충돌 시 isTriggerEnter에서 구현하자.

**! C#의 문자열 보간**

      // 일반적인 문자열 조합. 익숙하다.

      Debug.Log("Player Hp : " + currHp + "/" + initHp + "=" + currHp/initHp);

      // LogFormat을 사용한 표현. 배열 느낌

      Debug.LogFormat("Player Hp : {0}/{1}={2}", currHp, initHp, currHp/initHp);

      // 문자열 보간($)을 사용한 표현

      Debug.Log($"Player Hp : {currHp}/{initHp} = {currHp/initHp}");

**특정 레이어 간의 충돌 감지**

지금까지 만든 몬스터의 공격 로직은 겉으로 보기에는 별 이상이 없어 보인다. 하지만 몬스터 몸체에 추가된 Capsule Collider와 양손에 추가한 Sphere Collider는 몬스터가 움직일 때마다 지속적으로 충돌을 감지하고 있다. 양손에 추가한 Sphere Collider가 Rigidbody 컴포넌트를 포함하고 있기 때문에 몬스터 자신의 Capsule Collider와 충돌을 발생시킨다.

이와 같은 물리 엔진의 불필요한 부하는 특정 Collider 사이에 충돌이 감지되지 않게 설정함으로써 줄일 수 있다. 이러한 물리엔진 설정은 레이어로 판단하기 때문에 각 Collider를 서로 다른 레이어로 지정해야 한다. 포토샵 그느낌 맞긴 함.

Layer Collision Matrix 설정은 흔히 게임에서 아군과 적군이 있을 경우 아군이 발사한 총알의 충돌은 적군에게만 발생하게 하는 로직에 응용할 수 있다. 아군의 레이어와 적군의 레이어를 분리하면 되니까. 이 방법은 물리엔진의 부하도 줄일 수 있고, 로직도 간결하다.

**본 구조의 최적화**

보통의 인체형 모델은 애니메이션을 수행하기 위해 Rig(뼈대)가 설정되어 있다. 적 캐릭터 모델도 이러한 관절이 설정돼 있으며 하이러키 뷰에서 볼 수 있음

하이러키 뷰 상에 표시된 각 관절은 게임오브젝트이며 모두 Transform 컴포넌트를 가지고 있다. 이처럼 하나의 3D 모델이 가진 많은 Transform 컴포넌트의 이동 및 회전 처리는 런타임 시 내부적으로 다양한 연산처리를 하므로 최적화가 필요하다. 따라서 실제 게임에서 사용되는 관절만 남겨두고 다른 것은 노출되지 않게 설정하면 속도 향상에 도움을 줄 수 있으며, 하이러키 뷰를 간결하게 유지해 작업 능률을 높일 수 있다.

몬스터 모델에 필요한 뼈대는 일단 양손 뿐이다. 이 부분만 하이러키 뷰에 노출되게 설정해보자.

몬스터 프리팹 말고 원본 찾아가서 rig 선택하고 Optimize Game Objects를 켠다. 그러면 Extra Transforms to Expose 항목이 뜬다. 그럼 거기서 계층 뒤져서 L\_wrist와 R\_wrist 에 체크하고 Apply.

**몬스터 공격 중지**

주인공 캐릭터 hp가 0 이하일 때는 죽는 애니메이션을 실행하거나 Game Over 화면으로 넘어가게 된다. 이때 몬스터한테 주인공 뒤졌다고 알려줘야 한다.

씬 뷰에 있는 특정 게임오브젝트에 접근하기 위한 여러 방법 중에, 다음 함수와 같이 Tag를 사용해 접근하는 방법이 있다. Tag는 정말 많이 쓰인다. ㄷㄷ

\* GameObject.FindGame(ObjectWithTag(string tag);

\* GameObject.FindGameObjectsWithTag(string tag);

    void PlayerDie()

    {

      Debug.Log("플레이어 죽음!");

      // 게임 오브젝트 중 MONSTER 태그를 가진 애들을 전부 모아서 배열에 담을 것이다.

      GameObject[] monsters = GameObject.FindGameObjectsWithTag("MONSTER");

      // 모든 몬스터의 OnPlayerDie 함수를 순차적으로 호출

      foreach(GameObject monster in monsters)

      {

        monster.SendMessage("OnPlayerDie", SendMessageOptions.DontRequireReceiver);

      } // monster한테 OnPlayerDie 함수가 있으면 실행해라! 없다고? ㄱㅊ 안알려줘도댐 응 알어~

    }

PlayerDie함수에 추가한 내용은 씬 뷰에 있는 모든 몬스터를 태그로 검색한 후 배열에 저장한다. 그 다음 foreach 구문을 사용해 배열의 처음부터 끝까지 순회하면서 OnPlayerDie 함수를 호출한다.

SendMessage 함수는 첫 번째 인자로 전달한 함수명과 동일한 함수가 해당 게임오브젝트의 스크립트에 있다면 실행해라! 라는 뜻이다. 여러 게임오브젝트의 함수를 호출하는 로직에 사용하면 효율적이다. 두 번째 인자를 SendMessageOptions.DontRequireReceiver로 설정하면 호출한 함수가 없더라도 함수가 없다는 메시지를 반환받지 않겠다는 옵션이다. 빠른 실행을 위해 반드시 작성할 것.

    // 플레이어 죽었을때 호출될 함수

    void OnPlayerDie()

    {

        // 몬스터의 상태를 체크하는 코루틴함수를 모두 정지시킴

        StopAllCoroutines();

        // 추적을 정지하고 애니메이션을 수행

        agent.isStopped = true;

        anim.SetTrigger(hashPlayerDie);

    }

이제 이거 해놓고, 플레이어 디지면 몬스터가 강남스타일을 출 것이다. Animator 드가서 강남스타일 추가하고, Any State랑 Exit 연결하자. 파라미터는 PlayerDie 추가하고, AnyState->강남스타일, 강남스타일->Exit 조건 모두 PlayerDie를 추가하자. 391p 참고

**애니메이션의 재생 속도 조절**

몬스터들은 한치의 오차도 없이 같은 동작으로 강남스타일 춤을 춘다. 칼 군무가 목적이 아니라면 조금 어색하다. 동작을 조금씩 다르게 해보자.

Animator 파라미터에 float Speed 추가하자. 이 스피드는 속도를 랜덤하게 설정해서 각각 부여해줄거다.

Parameter에 체크하면 Multiplier 속성이 활성화되면서 자동으로 float 타입의 파라미터인 Speed가 선택된다. 이제 스크립트에서 Speed 파라미터 값을 변경하면 애니메이션의 재생 속도가 변경된다.

**사용자 정의 이벤트 – Delegate**

앞서 주인공이 사망하면 모든 적 캐릭터가 공격을 중지할 수 있게 foreach문을 이요해서 순차적으로 공격 중지 함수를 호출했다. 이때 SendMessage를 사용했는데, 스테이지에 적 캐릭터가 아주 많다는 극단적인 상황을 가정한다면 순차적인 호출 방법은 그리 효율적인 방법은 아닌 듯하다.

순차적인 호출 방식을 이벤트 구동(Event Driven) 방식으로 변경해보자. 이벤트란 특정한 조건을 만족하면 자동으로 알려주는 메시지(Notification)를 의미한다. 지금까지 만든 게임에 적용해보자. 주인공이 디졌을 때 디졌다고 알리는 이벤트를 생성하면 상황이 발생했을 때 시스템에서 이를 통보(Notification)하고, 이 이벤트에 연결된 모든 적 캐릭터에서 정해진 동작을 수행하는 방식이다. 이러한 이벤트 구동 방식은 for, foreach구문을 사용해 순차적으로 호출하는 방식보다 메모리 사용 측면이나 구동 속도 측면에서 효율적이다.

**델리게이트?**

함수(메소드)를 참조하는 변수를 의미한다. 즉, 함수(메소드)를 저장할 수 있는 일종의 변수라고 생각하면 이해하기 쉽다. C++의 함수 포인터와 같은 의미다.

델리게이트를 사용하려면 먼저 선언해야 한다. 어떤 형태의 함수를 저장할지 정의를 내리는 것이다.

\* 접근제한자 delegate 반환\_타입 델리게이트명 (매개변수…);

…?어? 모르겠는데? 일단 진행

**주인공의 사망 이벤트 처리**

PlayerCtrl 스크립트의 선언부에 델리게이트와 이벤트 함수를 선언하고 OnTriggerEvent 함수를 다음 스크립트와 같이 수정한다. PlayerDie 함수에서 현재 생성도니 모든 몬스터를 태그로 찾아와 배열에 담고 그 배열을 순회하면서 SendMessage 함수로 메시지를 전달할 필요가 없다. 간단히 이벤트만 호출하면 된다. PlayerDie 함수 기존 코드는 주석처리해뿌고

  // 델리게이트 선언. 이벤트를 사용하기 전에 꼭 선언해야 함.

  public delegate void PlayerDieHandler();

// 이벤트 선언

public static event PlayerDieHandler OnPlayerDie;

// 이벤트는 언제 호출될지 모르기 때문에 정적 변수, static으로 선언해야 함.

// OnPlayerDie는 이벤트명이라고 하지만 변수의 일종이며 playerDieHandler는 OnPlayerDie

변수의 타입일 뿐이다. 즉 (public static event 변수타입 변수명) 인거다.

void PlayerDie()

    {

      Debug.Log("플레이어 죽음!");

    OnPlayerDie();

      }

    // 스크립트가 활성화될 때마다 호출되는 함수

    void OnEnable()

    {

        // 이벤트 발생 시 수행할 함수 연결

        PlayerCtrl.OnPlayerDie += this.OnPlayerDie;

    }

    void OnDisable()

    {

        PlayerCtrl.OnPlayerDie -= this.OnPlayerDie;

    }

OnEnable()과 OnDisable()은 스크립트가 활서오하되거나 비활성화될 때 수행되는 함수로서 이벤트의 연결과 해지는 반드시 이 함수에서 처리해야 한다!!

\* 이벤트 연결 : (이벤트가 선언된 클래스명).(이벤트명) += (이벤트 발생시 호출할 함수)

\* 이벤트 해지 : (이벤트가 선언된 클래스명).(이벤트명) -= (이벤트 발생시 호출할 함수)

아마 스위치처럼 on, off 인것 같다.