[ Crash Course ]

5. Important to know – Crash Course

6. Install Unity

7. Windows and Tools

8. Components in Unity

Component

: 유니티의 컴포넌트는 오브젝트에 기능을 부여할 수 있는 동작 또는 프로퍼티 세트와 비슷

9. Basics of C#

10. Unity’s execution order

Monobehavior

: 스크립트가 게임 오브젝트에서 컴포넌트로 작동할 수 있도록 하는 베이스 클래스

: 상속받지 않으면 유니티에서 사용할 수 없음

: 유니티의 라이프 사이클 관리

11. C# - Functions

12. Input & Movement

13. Decision making & Jump

14. Animator

15. Animation transitions

16. Character flip logic

17. Collision detection

18. Jump, Fall & Blend tree

19. Attack animation

20. Attack logic & Enemy detection

21. Timers & Cooldowns

22. OOP – Inheritance

23. OOP – Polymorphism

24. OOP – Encapsulation

25. Conclusion & Project

[ FSM & Project Setup ]

26. Create project & Get assets

27. What is Finite State Machine

State Machine : 객체가 특정 시점에 하나의 상태(State)만 가질 수 있고, 특정 조건에 따라 다른 상태로 전환(Transition) 되는 구조. 각 상태는 고유한 행동과 로직을 가지며, 상태 간의 전환은 미리 정의된 규칙에 따라 이루어짐

<구현 방법>

1. Switch문 기반 : 간단한 상태 관리에 적합
2. 클래스 기반 : 각 상태를 별도 클래스로 구현
3. Unity Animator : 비주얼 상태 머신으로 애니메이션과 연동
4. 서드파티 라이브러리 : Playmaker, Bolt 등

상태머신을 사용하면 복잡한 게임 로직을 명확하게 구조화 가능.

디버깅이 쉬워짐.

새로운 상태 추가나 기존 상태 수정이 용이해짐.(캐릭터 컨트롤러나 게임 플로우 관리에서 매우 유용)

28. Create Finite State Machine

29. New States & Structure

30. Setup New Input System

31. Clean up & Project

<3개의 핵심 컴포넌트>

1. EntityState : 모든 상태의 기본 클래스(추상 클래스)
2. StateMachine : 상태를 관리하고 전환하는 관리자
3. 구체적인 상태들 : Player\_IdleState, Player\_MoveState

<각 클래스 상세 분석>

1. EnityState : 추상 기본 클래스

* Enter() : 상태에 진입할 때 호출(초기화)
* Update() : 매 프레임마다 호출(상태별 로직 실행)
* Exit() : 상태를 나갈 때 호출(정리)

1. StateMachine : 상태 관리자

* Initialize() : 최초 상태 설정
* ChangeState() : 상태 전환(Exit -> Enter 순서로 호출)
* UpdateActiveState() : 현재 상태의 Update() 실행

1. 구체적인 상태들 : 대기상태, 이동상태 등

<동작 흐름>

1. 초기화 과정

* Awake() : 상태 머신과 각 상태 객체들 생성
* Start() : Idle 상태로 초기화
* stateMachine.Initialize( idleState ) -> IdleState의 Enter() 호출

1. 런타임 동작

* 매프레임 : Update() -> stateMachine.UpdateActiveState() -> 현재 상태의 Update() 실행
* 입력 감지 : InputSystem이 이동 입력을 moveInput에 저장
* 상태 전환 : 각 상태의 Update()에서 조건 확인 후 상태 변경

<이 구조의 장점>

1. 확장성 : 새로운 상태(점프, 공격 등)를 쉽게 추가 가능
2. 명확성 : 각 상태의 로직이 분리되어 이해하기 쉬움
3. 안전성 : 한번에 하나의 상태만 활성화
4. 재사용성 : 다른 캐릭터에도 동일한 구조 적용 가능

<궁금했던 부분>

1. 왜 State가 StateMachine을 참조해야 하는가?

* 상태 전환의 주체 문제
* 누가 상태를 바꿔야 할까? => State가 직접 요청
* 각 상태는 자신의 조건을 가장 잘 알고 있음
* 상태 전환 조건이 복잡할 수 있음(여러 변수 체크, 타이머 등)
* 상태 머신은 ‘어떻게 전환할지’만 관리, ‘언제 전환할지’는 각 상태가 판단

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

* 대안적 설계 방식 : 만약 StateMachine이 모든 전환을 관리한다면?

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

1. 왜 Player가 모든 것을 들고 있는가?

* 현재 구조의 이유

텍스트, 폰트, 스크린샷이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

* 상태간 참조 필요 : 각 상태가 다른 상태로 전환할 때 참조가 필요
* 생명 주기 관리 : Player가 생성될 때 모든 상태도 함께 생성
* 단일 책임 : 각 상태는 자신의 로직만, Player는 전체 조율만
* 다른 설계 방식과 비교
* 방식 1 : 상태 내부에서 생성

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

* + 성능 문제 : 매번 새 객체 생성
  + 메모리 낭비 : GC 압박
* 방식 2 : Factory 패턴 사용

텍스트, 폰트, 스크린샷, 친필이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

* 더 나은 설계 예시

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

1. 결론

* 학습용으로는 완벽 : 구조가 명확하고 이해하기 쉬움
* 소규모 프로젝트에 적합 : 상태가 많지 않을 때 효율적
* 확장 시 고려 필요 : 상태가 10개 이상 되면 다른 패턴 고려
* 상태 머신 패턴에는 정답이 없음

<추가 질문 사항>

StateMachine이 여러 개가 존재하는지? => 하나만 존재함!

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

참조는 메모리를 거의 사용하지 않지만(C#에서는 보통 8바이트), 상태가 수백 개 이상이 되어 참조 조차 부담스러울 경우

1. Static 참조

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

1. 이벤트 시스템

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

1. Context 패턴

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

<실제 게임 개발에서는?>

대부분의 경우 현재 구조로 충분

모바일 게임 : 상태 10 ~ 20개 정도면 전혀 문제없음

PC 게임 : 상태 50 ~ 100개여도 참조 오버헤드는 무시할 수준

MMO급 게임 : 이때는 아예 다른 아키텍처 사용(ECS 등)

<ECS(Entity Context System)란>

전통적인 방식(상속 기반)

텍스트, 폰트, 스크린샷, 친필이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

ECS 방식(조합 기반)

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

핵심 구성 요소

Entity(개체) : 게임 오브젝트의 고유 식별자

Component(컴포넌트) : 순수 데이터(위치, 체력, 속도 등)

System(시스템) : 특정 컴포넌트 조합을 가진 Entity들을 처리하는 로직

장점

성능 : 같은 타입의 데이터가 메모리에 연속적으로 배치되어 캐시 효율성 극대화

유연성 : 런타임에 컴포넌트 추가/제거로 객체 행동 변경 가능

확장성 : 수만 개의 Entity도 효율적으로 처리

언제 사용하나?

적합한 경우 : 대규모 시뮬레이션, RTS 게임, 수천 개의 유닛이 동시에 움직이는 게임

부적합한 경우 : 소규모 인디 게임, 복잡한 UI 중심 게임, 프로토타입 단계

Unity에서는 DOTS(Data-Oriented Technology Stack)라는 이름으로 ECS 제공 중. 그러나 학습 곡선이 가파르고 기존 Unity 시스템과 호환성 문제가 있어서 정말 성능이 중요한 경우가 아니라면 전통적인 Monobehaviour 방식 추천

예시

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

<람다식(Lambda Expression)>

간단한 함수를 한 줄로 표현하는 방법. Unity에서 이벤트 처리할 때 매우 자주 사용

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

텍스트, 스크린샷, 폰트, 친필이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

주의 사항

1. 성능 고려

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

1. 메모리 누수 방지

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

* 이벤트 구독이 해제되지 않을 때, 메모리 누수가 발생함
* 올바른 해결 방법

명시적 구독 해제/OnDisable() 등

안전한 콜백 체크

WeakReference 사용(고급) \*\*

* 메모리 누수 감지 방법

Unity Profiler 사용, 디버깅 코드 추가

* 핵심 포인트

이벤트 구독하면 반드시 해제하기

람다식이 참조하는 객체의 생명주기 고려하기

OnDestroy에서 정리 작업 수행하기

Unity Profiler로 메모리 사용량 모니터링 하기

장점

간결성 : 코드가 짧고 읽기 쉬움

가독성 : 관련 로직이 한 곳에 모임

유연성 : 즉석에서 간단한 함수 정의 가능

<인라인 함수 vs 람다식>

<LINQ>