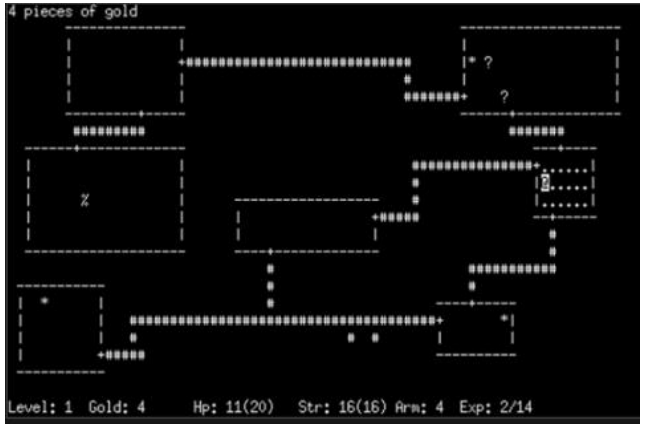
[목차]

1. 프로젝트 개요



- 로그라이크 시초인 '로그'

- 티벳 불교의 교리인 '윤회' : 육도윤회도

'**로그라이크**' 라는 장르의 특성을
티베트 불교의 '**윤회**'라는 교리에 비유하여
제작한 게임 입니다.

프로젝트 개요				
장르	개발 엔진	플랫폼	개발 인원	개발 기간
멀티 액션 로그라이크	UE_5.3	PC	22명	23.01 ~ 23.11

총 22명의 팀원이 8개월동안 협업하였고,
UE5를 사용하여 제작했습니다.
그 중 **클라이언트 프로그래머(파트장)**를 맡았습니다.

2. 역할 및 기여

담당 업무를 두가지 측면으로 구분 하자면 아래와 같습니다.

- 파드마 캐릭터

클라이언트 프로그래머로서,

1. 게임 속의 플레이어블 캐릭터인 '파드마' 구현
2. 보스전 빙결 시스템 구현
3. 프로그래머들의 협업을 위한 주요 코드 구조 결정

프로그래머 파트장으로서,

1. 개발 주요 프로세스 결정
2. 타 파트들과 소통을 하며 일정 조율, 업무 분배 및 지시
3. 노션 페이지를 통해 협업과 관련된 주요 페이지를 직접 작성

3. 개발 환경 구축 – 저장소 결정

본격적인 프로젝트를 시작하기 전, 개발 환경을 구축했습니다.

현 프로젝트 에서는 Azure Devops 저장소를 사용했습니다.
아래 표와 같이 LFS 용량을 고려하여 결정하게 되었습니다.

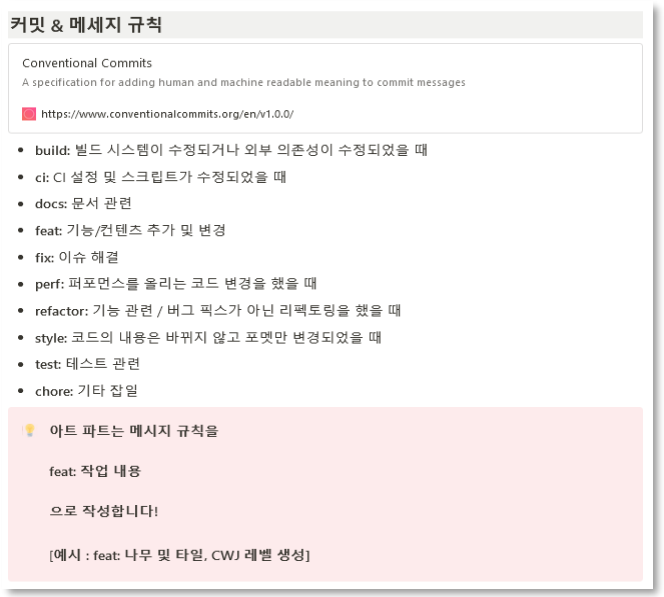
LFS 용량 제한 정리			
저장소	Github	GitLab	**Azure Devops**
제한 용량	1gb	10gb	**무제한**

예외적으로 언리얼 엔진의 레벨 에셋 (.umap) 은 Google Drive 로 관리했습니다.

.umap 파일들은 크기가 커서 Git의 Pull, Push 작업을 느리게 만들었습니다.
또한 저장소 최대 용량도 빨리 소모되게 하는 주 원인이었습니다.

이 문제를 해결하기 위해, 맵 파일은 구글 드라이브로 공유하는 것으로 결정했습니다.

3. 개발 환경 구축 – 버전 관리



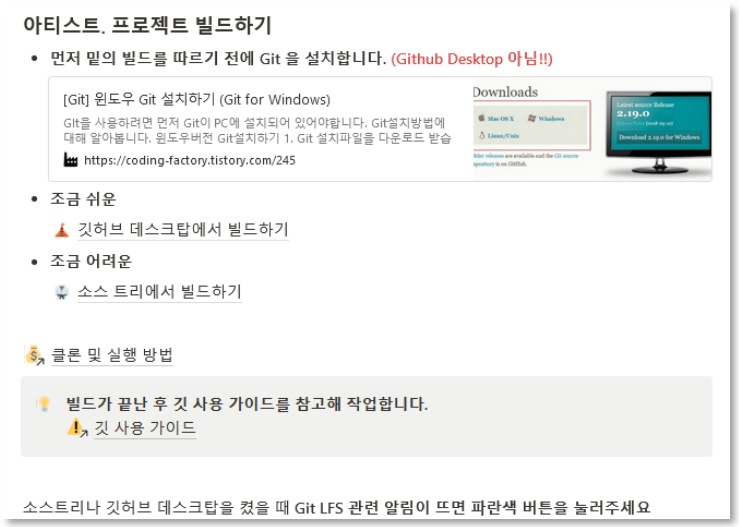
프로젝트 버전관리의 측면에서 신경 쓴 것은 아래와 같습니다.

1. 팀 전원, main branch 에서 작업을 했습니다.

Git 에 미숙한 팀원들이 branch 에서 충돌내는 것을 방지하고, merge 로 인한 이슈 해결에 개발 시간을 소모하지 않기 위하여 결정하게 되었습니다.

2. Commit 메시지 규칙을 정했습니다.

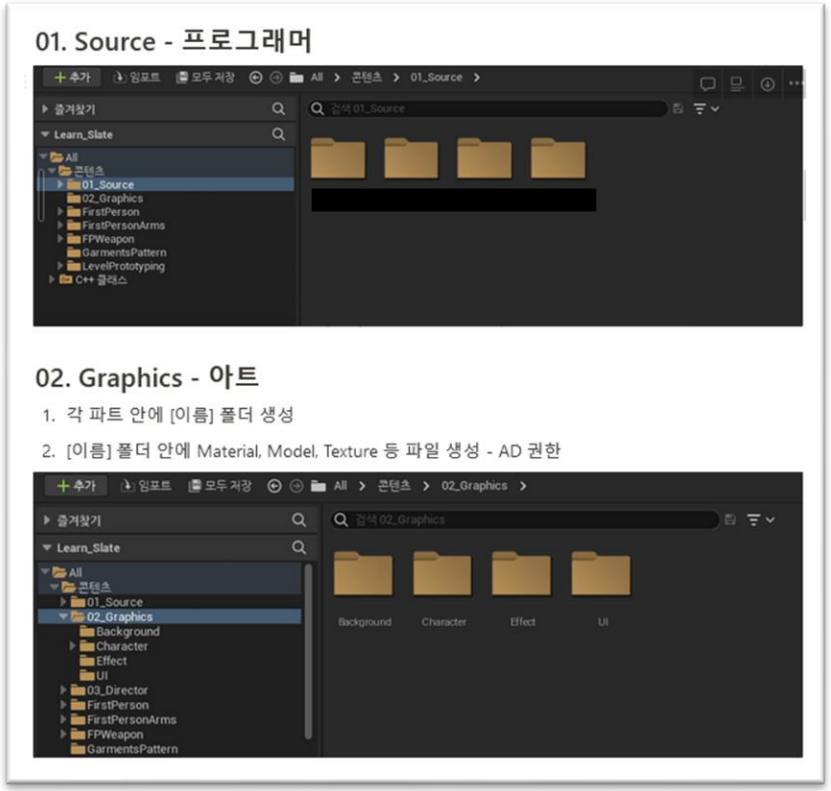
팀원들의 활동을 파악할 수 있고,
문제 원인도 빠르게 파악할 수 있다고 판단하여 규칙을 정하게 되었습니다.



3. Git 가이드를 작성했습니다.

Git 에 익숙하지 않은 타 파트를 위해,
팀원들이 사용하는 Git 관리 툴에 맞춰 작성했습니다.

3. 개발 환경 구축 – 협업



4. Unreal Engine 에서 각 파트 별 폴더 구조 규칙을 정했습니다.

버전 관리에서 서로 충돌이 일어나지 않게 하기 위해서,
폴더를 구분하고 [이름] 폴더를 만들어 진행했습니다.
팀원 모두 main branch 에서만 작업하는 만큼 충돌이 나지 않도록
주의했습니다.

5. Unreal Engine 개발 가이드를 작성하여 공유했습니다.

플러그인 적용 방법, 5.3 버전 업데이트 방법 등,
팀원들이 어려움 없이 프로젝트를 진행할 수 있도록
노션 페이지로 정리하여 공유했습니다.

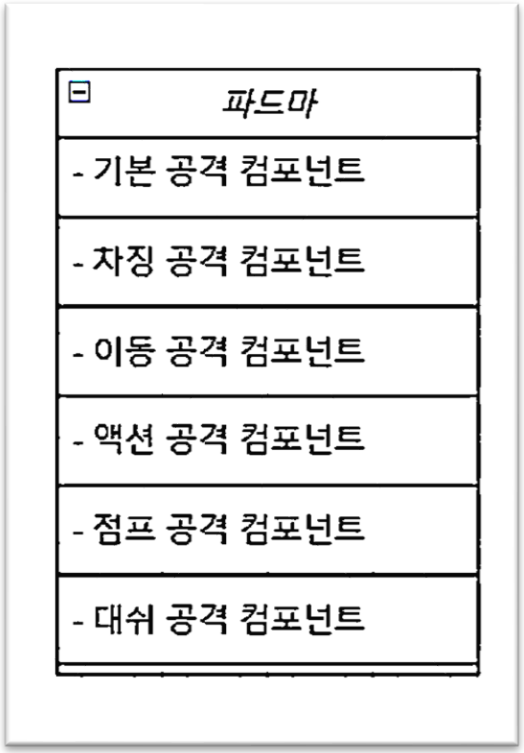
4. 주요 기능 및 구현 내용 - 개발 과정 개요

주요 기능을 구현한 내용들을 정리한 로드맵입니다.
시행착오를 겪으며 점진적으로 개선했습니다.

- 1.Component 구조로 시작
- 2. Action System 기반의 구조로 개선
  - 2-1. 협업을 위한 Action Blueprint
- 3. AttackAction 구조로 확장
  - 3-1. AttackAction 의 문제
- 4. 데이터 관리 및 협업을 위한 Google Sheets
- 5. Google Sheets 데이터 파싱
  - 5-1. Unreal Editor Module 을 이용한 확장 및 세부 구현
- 6. AttackActionData 분리
  - 6-1. AttackActionData 의 기능
  - 6-2. AttackActionData 와 DataAsset

1. Component 구조로 시작

- 초기의 캐릭터 구조



초기에는 각 스킬을 컴포넌트로 분리해 캐릭터에게 붙이는 구조로 시작했습니다.

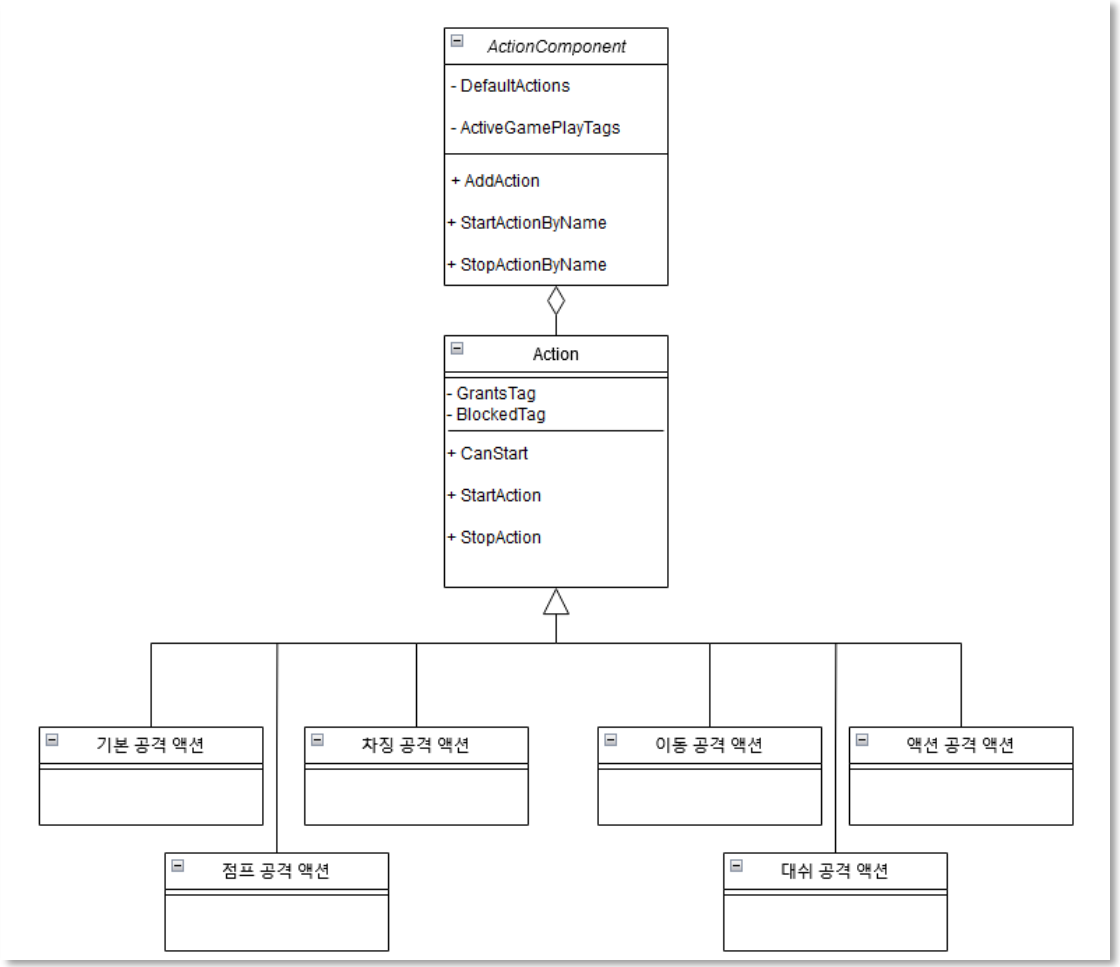
하지만, 아래와 같은 문제점들이 발생했습니다.

1. 스킬의 개수가 많아질수록 캐릭터 클래스가 복잡해짐.
2. 관리해야 하는 데이터와 기능이 많아짐.
3. 타 파트들과 같은 캐릭터 블루프린트를 수정하기 어려웠음.
4. 협업 시 Git 충돌 문제가 잦게 발생함.

위의 문제를 해결하기 위해, 구조를 다시 변경하게 되었습니다.

2. Action System 구조로 개선 - 개요

- 중반의 캐릭터 구조

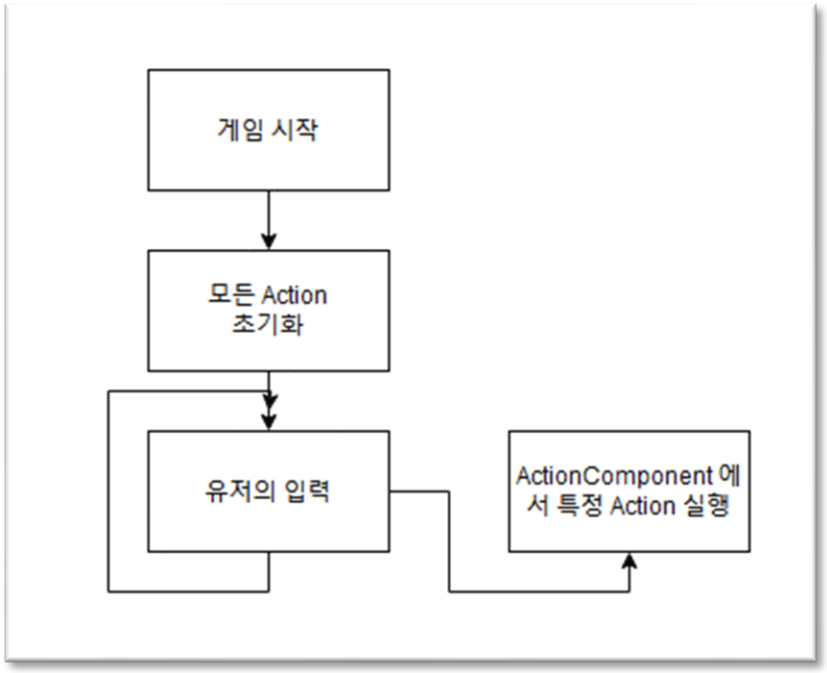


Unreal Engine 의 Game Ability System을 참고하여
Action System 기반의 구조로 캐릭터 시스템을 변경했습니다.

Action System 은 크게 2가지로 구성했습니다.

1. ActionComponent: Action들을 관리하고, 실행하는 역할
2. Action: 실제로 기능을 수행하고, GamePlayTags 로 관리됨

2. Action System 구조로 개선 – 세부1



Action 들은 캐릭터가 수행하는 하나의 스킬을 담당합니다.

초기화 -> 실행 -> 중지 순으로 진행됩니다.

Action 초기화 :

게임 시작 시 모든 Action 들은 ActionComponent 의 DefaultActions 배열에 추가되며, 초기화 작업을 진행합니다.

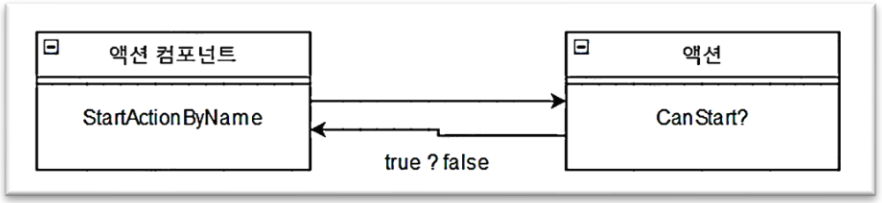
Action 실행 :

ActionComponent는 StartActionByName 을 통해 DefaultActions 에서 이름으로 특정 액션을 찾아 실행합니다.

Action 중지 :

캐릭터의 애니메이션이 종료되거나, 다음 Action 을 실행할 때 자동으로 기존 Action 을 중지하도록 구현했습니다.

2. Action System 구조로 개선 – 세부2



Action 이 시작될 때,
Action은 자신이 실행 가능한지를 판단합니다.

```
bool UAction::CanStart(AActor* Instigator) const
{
    const UActionComponent* Comp = GetOwningComponent();

    if(Comp->ActiveGamePlayTags.HasAny(BlockedTags))
    {
        return false;
    }
```

판단하기 위하여 ActionComponent 가 가진,
ActiveGamePlayTags Container에 Blocked Tag의 여부를 체크합니다.

Blocked Tag 가 없을 경우 아래의 과정을 통해 Action이 실행됩니다.

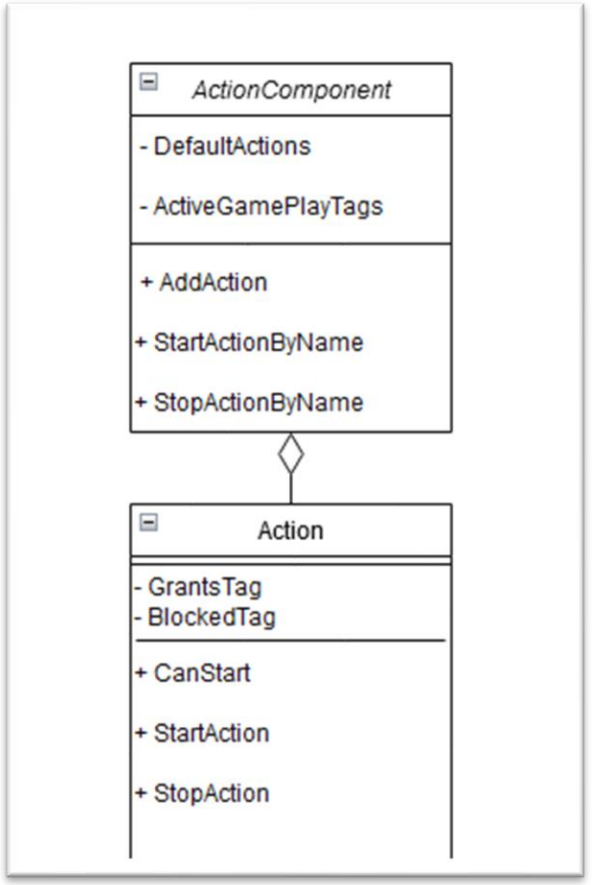
1. Action의 StartAction 함수를 호출
2. ActiveGamePlayTags Container에 GrantsTag 를 추가
3. Action 기능 수행

- 실제 구현 코드

```
void UAction::StartAction(AActor* Instigator)
{
    UActionComponent* Comp = GetOwningComponent();
    Comp->ActiveGamePlayTags.AppendTags(GrantsTag);
    Comp->OnActionStarted.Broadcast(GetOwningComponent(), Action: this);
    bIsRunning = true;
```

Blocked Tag 가 있을 경우 Action이 실행되지 않습니다.

2. Action System 구조로 개선 – 세부3



- Action 과 Action Component

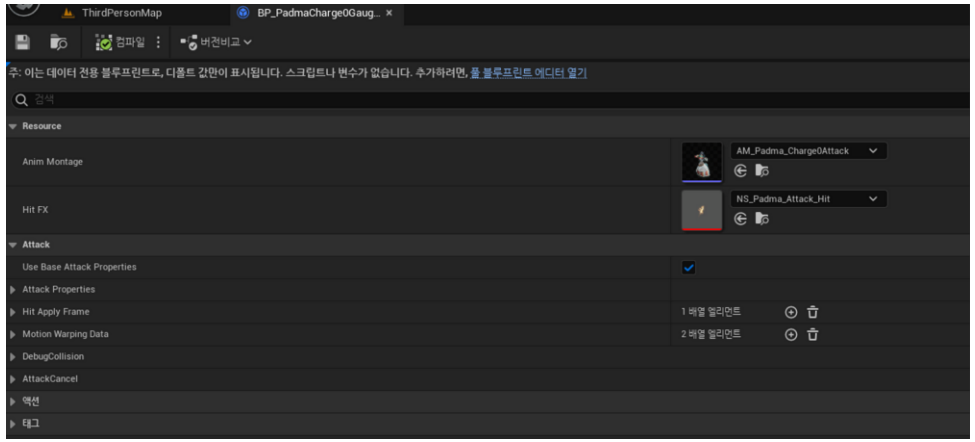
Action System 구조로 얻은 성과

1. Action 클래스의 밑의 함수만 구현하면 스킬들을 만들 수 있음.
   Initialize : Action 의 초기화
   StartAction : 캐릭터의 동작을 수행
   StopAction : 캐릭터 동작 중단

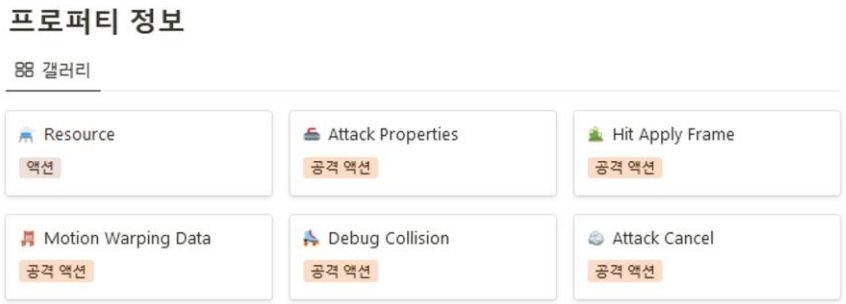
2. ActionComponent 로 Action 들을 쉽게 관리할 수 있음.
3. 캐릭터의 동작 단위가 Action 으로 구분 되어 직관적이며, 디버깅 편리.

2-1. 협업을 위한 Action Blueprint

- Action 클래스 블루프린트



- 블루프린트의 프로퍼티들



- 노션 페이지

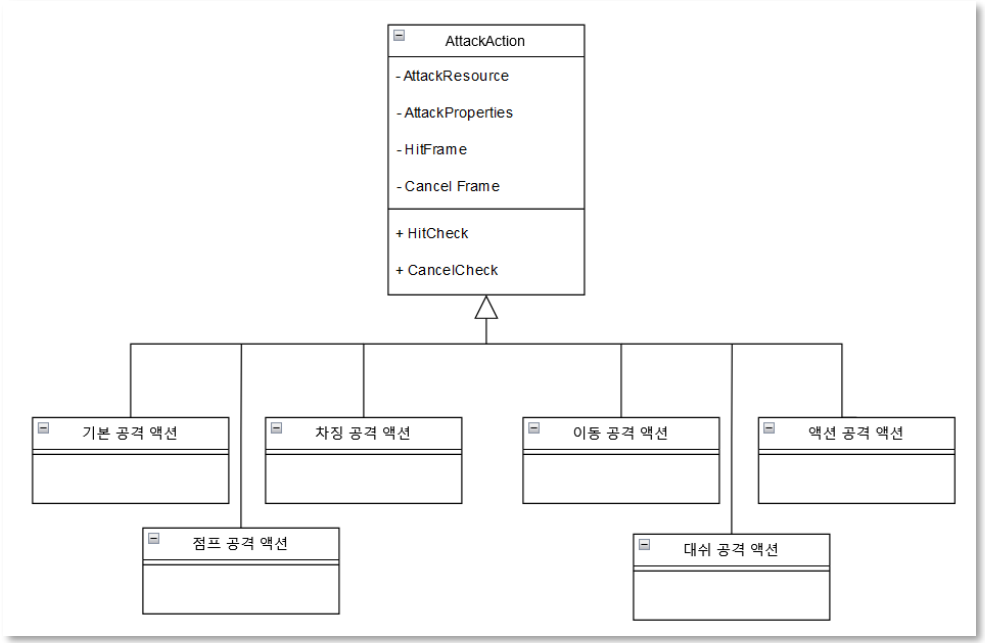
C++ 의 Action 클래스들을 언리얼 에디터에서 각각의 Blueprint 로 생성해서 관리했습니다.

직접 코드를 수정하지 않고도, 프로퍼티들의 에디터에서 쉽게 수정할 수 있게 되었습니다.

협업의 측면에서

Blueprint 안의 프로퍼티 값 변경 및 사용 방법을 공유해 기획자들 과의 협업도 원활하게 진행했습니다.

3. AttackAction 구조로 확장




```cpp
void UAttackAction::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    if(false == bIsRunning || !IsValid(PlayerCharacter)) return;

    const float CurrentSectionPos = Montage_GetPosition(AttackData->Resource.AnimMontage);
    if(CurrentSectionPos == 0.f) return;

    ApplyAttackCancel(CurrentSectionPos);
    ApplyHitCheck(CurrentSectionPos);
}
```

- 위처럼 AttackAction은 실행 중에
AttackCancel 과 HitCheck 처리를 수행합니다.

근접 전투 캐릭터인 “파드마”의 Action 들은
공통적인 데이터와 동작이 많았습니다.

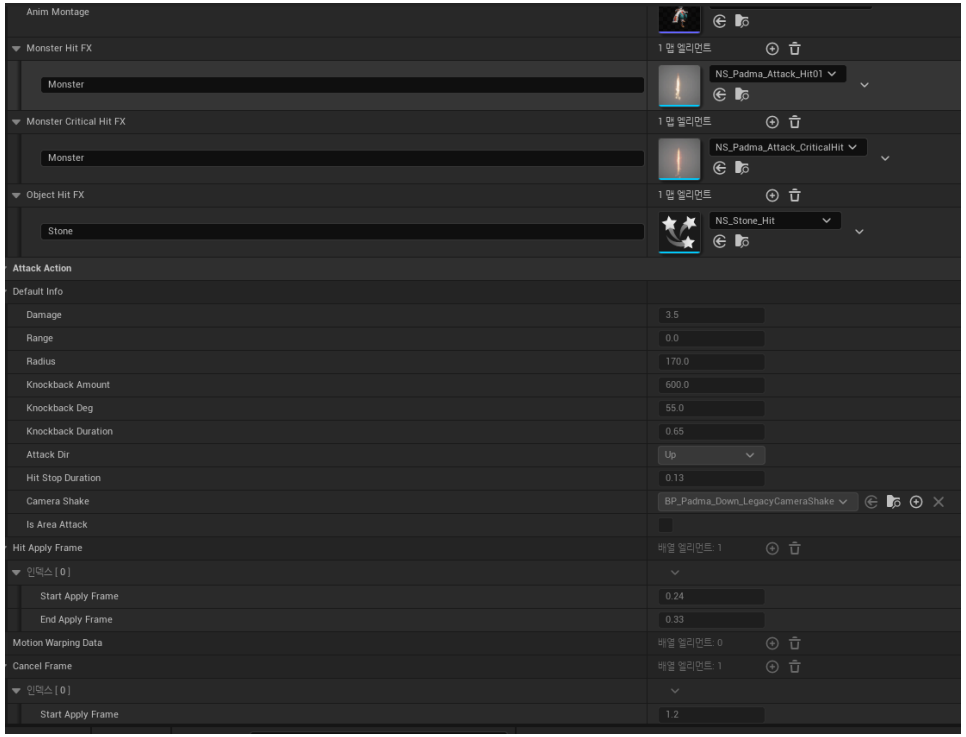
1. Resource 데이터:
   공격의 애니메이션 몽타주, 히트 이펙트

2. HitCheck 기능:
   무기를 휘두를 때 특정 애니메이션 프레임에만 실제 공격 판정 적용
   공통적으로 Damage, Radius, Range, Hit Frame 와 같은 데이터들이 필요.

3. AttackCancel 기능:
   공격 후 딜레이를 줄이기 위해서 공격을 캔슬하고
   다른 공격을 수행하거나, 캐릭터가 이동할 수 있게 만듦.

따라서 Action 클래스를 상속 받아 확장했습니다.
위와 같은 필요한 속성들을 가지고,
기능을 수행하는 AttackAction 을 구현했습니다.

3-1. AttackAction 의 문제



- 많은 블루프린트들을 일일이 수정하기 불편하다.

하지만 불편한 점이 있었습니다.
AttackAction이 기능을 수행하기 위한 데이터들이 많았습니다.

Action 클래스 블루프린트의 개수가 많아서,
하나하나 클릭해 수정하는 것도 번거로웠습니다.

이를 해결하기 위해 Google Sheets 를 사용했습니다.

4. Google Sheets 로 데이터 관리

기본 공격 정보	Damage	범위 공격인지	Range	Radius	KnockbackAmount	KnockbackDeg	KnockbackDuration	AttackDir	HitStopDuration
기본공격1	2	0	150	100	1100	0	0.35	Left	0.032
기본공격2	3.5	0	150	100	1000	0	0.5	Down	0.06
이동공격	1	0	0	170	50	0	1	Down	0.09
차징공격-심판	전용 시트	1	300	500	600	30	0.85	Up	0.15
차징공격-천둥치기	3	1	300	430	0	0	0.5	Down	0
대쉬공격	3.5	0	0	170	600	55	0.65	Up	0.13
액션공격-소용돌이	2.5	0	150	100	100	0	1	Left	0.013
유틸스킬	2	1	400	500	0	0	0	Left	0
점프공격	3.5	0	0	170	1200	0	0.5	Down	0.13

- AttackProperties 시트

히트 프레임	시작	끝	시작	끝	시작	끝
기본공격1	0					
기본공격2	0.16	0.35	3.4	3.57		
이동공격	0.7	0.75				
차징공격-심판	범위	공격은	노티파이로	변경		
차징공격-천둥치기	범위	공격은	노티파이로	변경		
대쉬공격	0.24	0.33				
액션공격-소용돌이	0.53	0.9				
유틸스킬	범위	공격은	노티파이로	변경		
점프공격	0.64	0.68	6.67	6.69		

- HitFrame 시트

----------	충전속도	충전 유지가능시간	충전 단계간격	재사용대기시간	차징 단계별 캔슬 프레임 간격
충전	1	8	1	10	0.4

/

----------	x단계_피해량계수	x단계_피해거리	x단계_피해범위반지름
충전 1단계	5	200	250
충전 2단계	6.5	300	350
충전 3단계	8	400	450
충전 4단계	10	500	550

/

----------	퍼짐	최대 넉백 각도
넉백	1	60

- 차징 공격 – 심판 시트

AttackAction 에서 필요한 데이터들을 모두 시트로 분리 후 틀을 만들어 기획자들에게 공유했습니다.

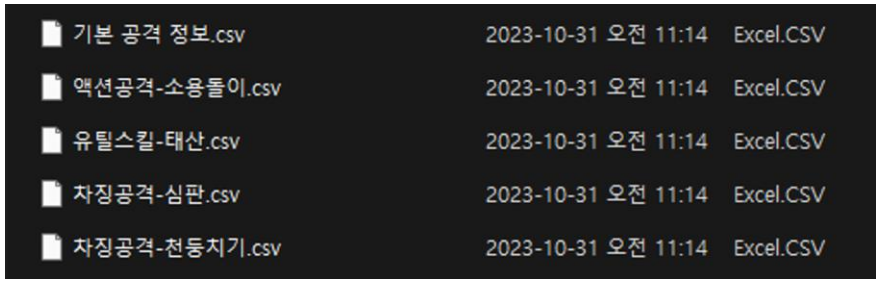
Google Sheets 를 사용 후 성과.

1. 기획자가 쉽게 원하는 데이터 추가 및 수정 가능함.
2. 시트 수정 후 플레이만 하면 되므로 테스트 속도 올라감.
3. 기획자들이 엔진을 사용하면서 생기는 실수가 적어 짐.

공통적인 데이터들 이외에 고유의 데이터가 많은 AttackAction 들은 따로 시트를 추가해서 데이터를 파싱했습니다.

5. Google Sheets 의 데이터 파싱

- .csv 다운로드 실행파일



- 다운로드 받은 .csv 파일

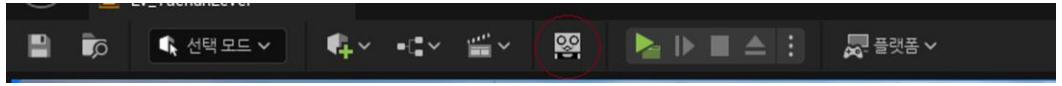
Google Sheets 의 시트들은 download_csv.exe 를 실행하면 .csv 파일로 다운받아 Contents 폴더에 저장됩니다.

그리고 게임 초기화 단계에서 .csv 데이터를 파싱해 저장합니다.

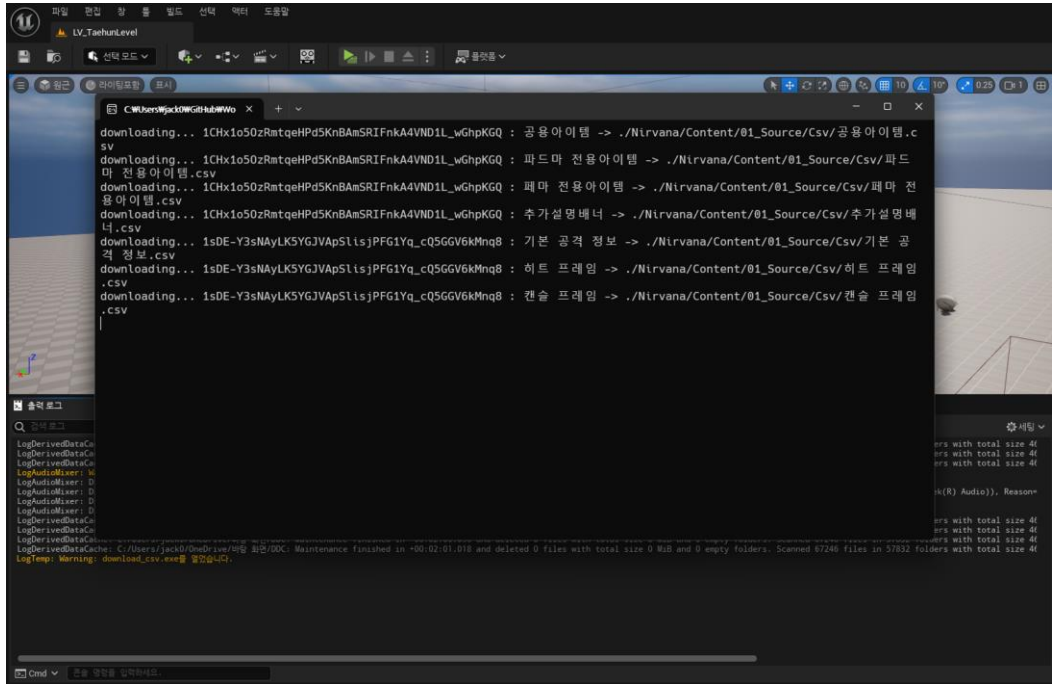
이 과정에서,
매번 파일 탐색기를 열어 더블 클릭해 실행시켜야 하는 것이 불편했습니다.

따라서 에디터를 확장해 버튼을 누르면 실행 파일이 실행되도록 구현했습니다.

5-1. Unreal Editor 확장



- 툴 바에 버튼



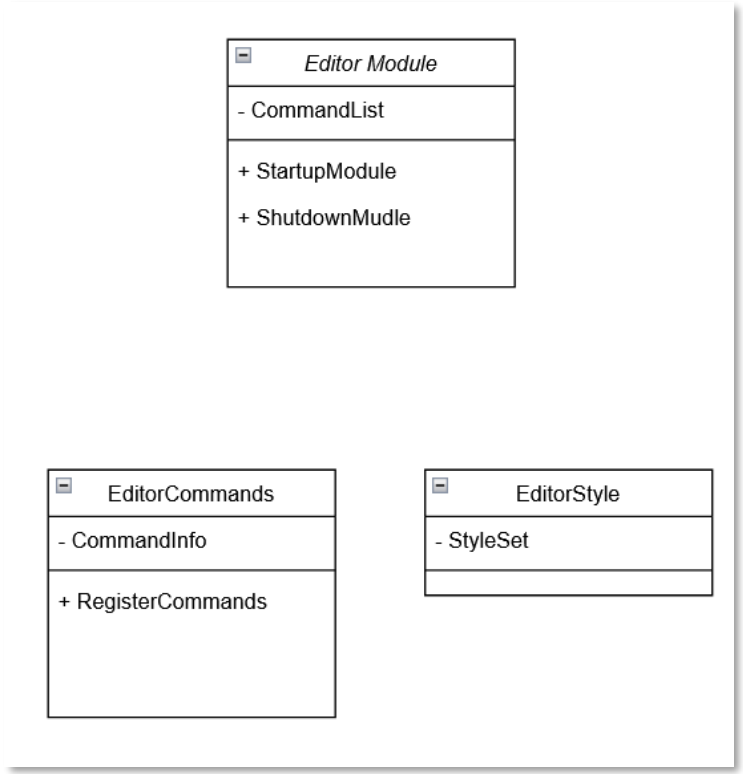
- 버튼을 눌러 실행한 모습

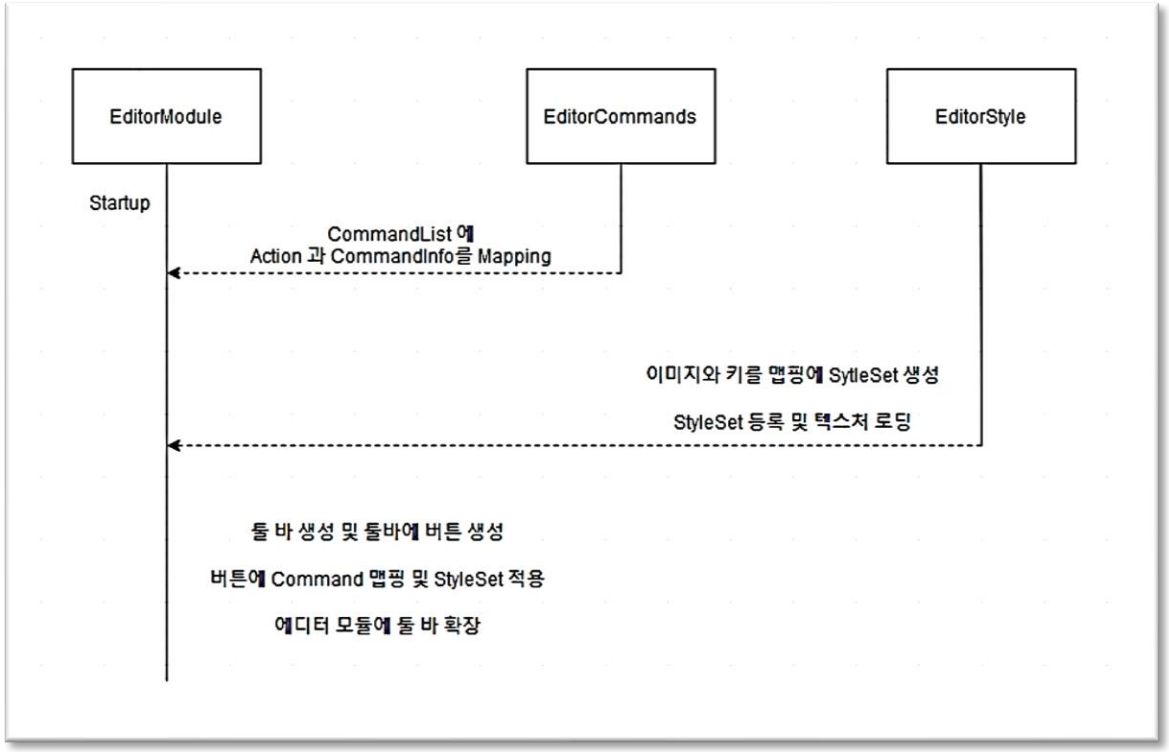
Editor Module 을 구현해 툴 바 버튼을 만든 모습입니다.
툴 바의 버튼을 눌렀을 때 download_csv.exe 를 실행합니다.

에디터 확장 후 성과.

1. 엔진 내에서 편하게 csv 파일을 다운로드 가능
2. 기획자, 프로그래머 모두 불편함을 해소
3. 생산성이 더 올라가는 효과

5-1. Unreal Editor 확장 – 세부 구현





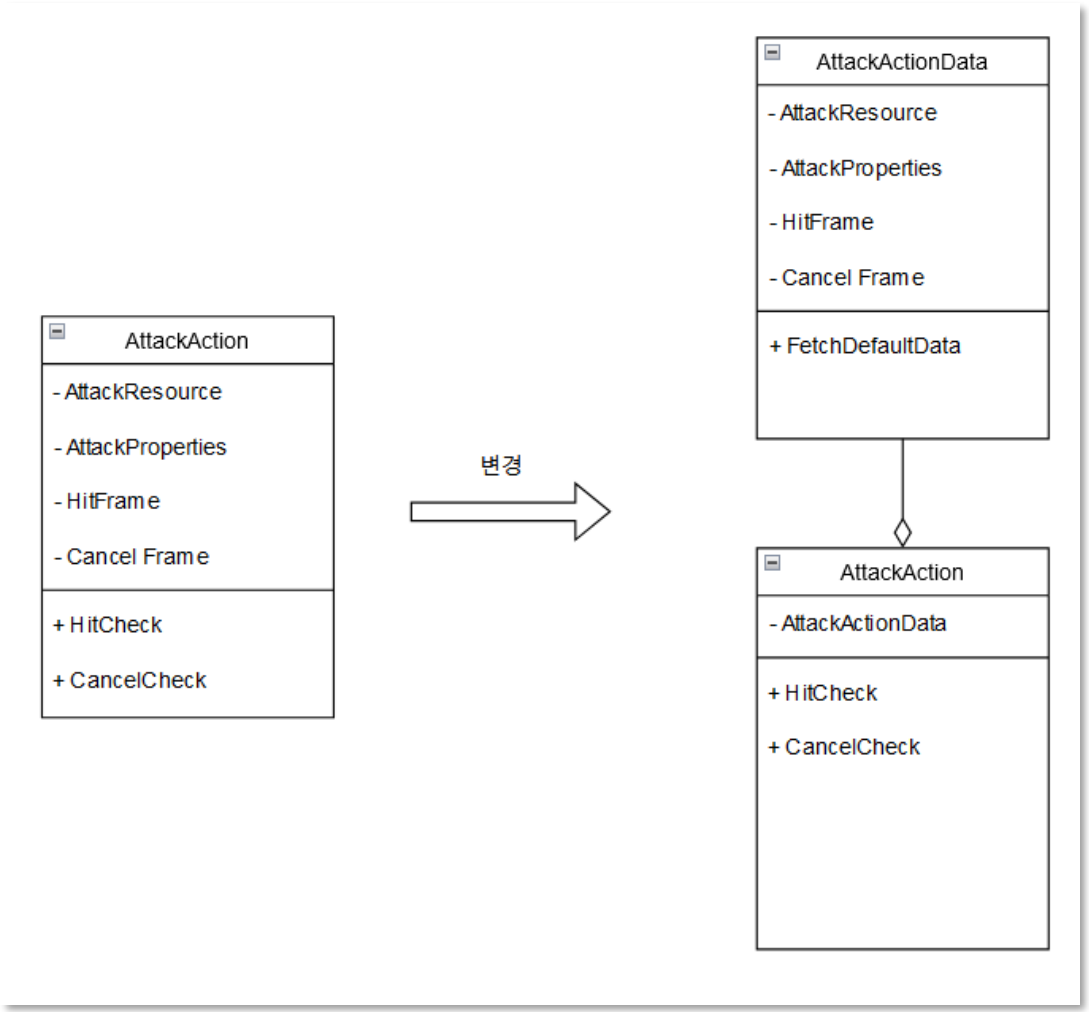
Unreal Editor 를 확장하는 데 필요한 요소

1. Editor Module
2. Commands
3. Editor Style

Editor 확장 플로우

1. Editor Module Startup 시, Commands 와 StyleSet 을 등록.
2. 툴 바를 생성.
3. 생성 시 Command 와 Style 을 버튼에 맵핑.

6. AttackActionData 로 분리



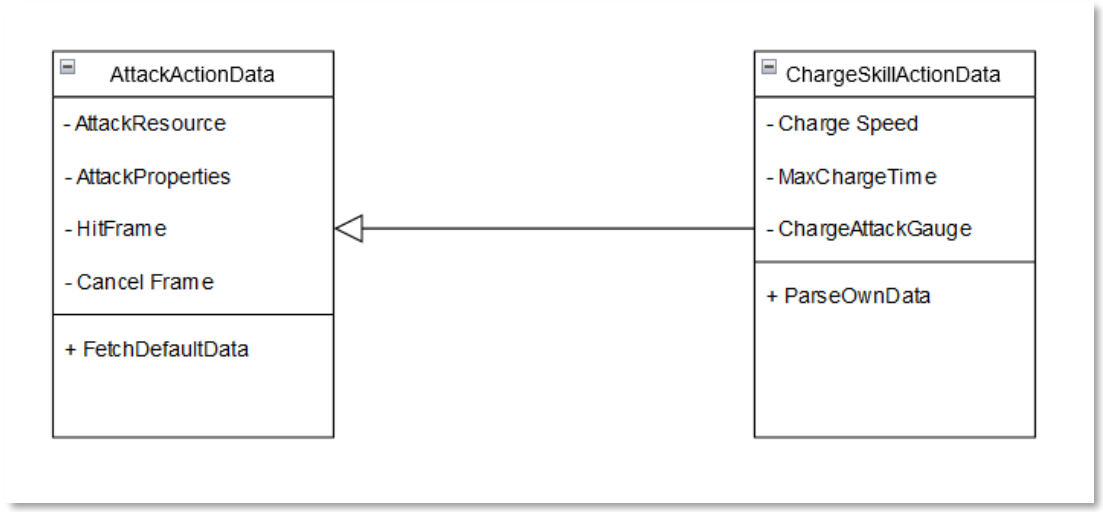
AttackAction 은 로직 뿐만 아니라 Google Sheets 에서 관리되는 모든 데이터들을 파싱 하며, 저장하고 있었습니다.

시간이 지날 수록 AttackAction 이 가진 데이터과 수행하는 기능이 많아졌습니다.

단일 책임 원칙에서 어긋날 뿐만 아니라 코드도 복잡해 진다고 판단했습니다.

따라서 AttackAction 의 데이터들과 파싱 기능을 분리해 AttackActionData 클래스로 구현했습니다.

6-1. AttackActionData 의 기능



AttackActionData 의 구조

1. Resource 와 Attack Properties, Frame 데이터
2. csv 파일을 파싱해 데이터들을 초기화하는 기능

AttackActionData 를 상속받아
공통 데이터들과 고유의 데이터들도 관리할 수 있습니다.

또한 ParseOwnData 기능을 통해 고유의 데이터는 따로 파싱하도록 설계했습니다.

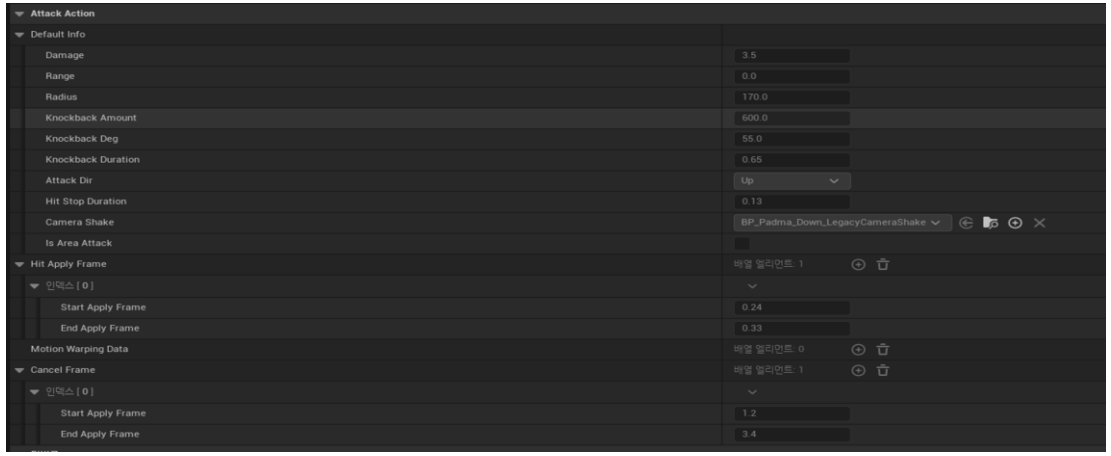
6-2. AttackActionData 와 DataAsset

```cpp
void UAction_PadmaSwirlAttack::Initialize(UActionComponent* NewActionComponent, AActor* Instigator)
{
    Super::Initialize(NewActionComponent, Instigator);

    MyDamageType = EDamagedType::ActionAttack;
    SkillData->FetchDefaultData(AttackName: TEXT("액션공격-소용돌이"), Callback: [] () ->void {});
    SkillData->ParseOwnData(FCsvDataManager::CsvCache[TEXT("액션공격-소용돌이.csv")]);
}
```

- Data 초기화

- DataAsset 들



- DataAsset 내부

이제 게임 시작 시 AttackAction 을 초기화할 때,
AttackActionData 를 같이 초기화합니다.

AttackActionData 는 UDataAsset 클래스를 상속받아

언리얼 기능 중 하나인 DataAsset 형식으로 에디터로
에셋을 열 수 있도록 구현했습니다.

그 이유는 디버깅 편의성입니다.
에디터에서 직접 에셋을 클릭해, 파싱했을 때 데이터가
정상적으로 들어왔는지 확인할 수 있습니다.

또한, Read-Only 데이터로 설정해 실수를 방지하게 했습니다.