Programowanie Gier Komputerowych

Wykład 7: Gameplay Ability System (GAS)

Mgr Inż. Staniszewski Hubert

Gameplay Ability System

Stworzony przez Epic Games z myślą o dużych projektach. Początkowo wykorzystywany wewnętrznie w grze „Paragon”, później udostępniony społeczności. Powstał jako odpowiedź na potrzebę elastycznego systemu łączącego zdolności, atrybuty i efekty. Integruje się z replikacją i multiplayerem w Unreal Engine co ułatwia pracę nad grami wieloosobowymi. Obecnie stanowi jeden ze standardów przy tworzeniu rozbudowanych mechanik postaci.

Gameplay Ability System

Gameplay Ability System – Wymagania

- Dobra znajomość architektury Unreal Engine (Blueprinty i C++)
- Rozumienie komponentów, dziedziczenia i klas aktorów
- Podstawy systemu tagów i replikacji w multiplayerze*
- Umiejętność pracy z systemem atrybutów (AttributeSet)

Budowa GAS – Główne komponenty

- **Ability System Component (ASC)** – centralny komponent zarządzający zdolnościami i efektami
- **Gameplay Ability** – definicja konkretnej umiejętności (np. atak, skok, leczenie)
- **Gameplay Effect** – opis efektów (obrażenia, leczenie, buffy, debuffy)
- **Attribute Set** – zestaw statystyk postaci (HP, mana, siła itd.)
- **Gameplay Tags** – system etykiet do warunków, ograniczeń i logiki działania abilities

Ability System Component

- Przechowuje i zarządza zdolnościami oraz efektami postaci
- Odpowiada za aktywację, anulowanie i cooldowny umiejętności
- Komunikuje się z Attribute Set, by zmieniać statystyki (np. HP)
- Obsługuje replikację zdolności w multiplayerze
- Wymaga przypięcia do aktora (np. postaci), aby działał poprawnie

AttributeSet – zarządzanie statystykami postaci

- Przechowuje dane takie jak zdrowie, mana, siła, wytrzymałość
- Każda statystyka to zmienna z możliwością replikacji
- Zmiany wartości są wywoływane przez **Gameplay Effecty**
- Umożliwia łatwe śledzenie i modyfikowanie atrybutów w czasie gry
- Współpracuje z ASC do synchronizacji w trybie multiplayer

Gameplay Tags – kontrola logiki zdolności

- Służą do oznaczania i filtrowania zdolności, efektów i stanów postaci
- Pozwalają definiować warunki aktywacji i blokady umiejętności
- Ułatwiają organizację i zarządzanie złożoną logiką rozgrywki
- Działają lokalnie i sieciowo, wspierają multiplayer

Przykład: tag "*State.Stunned*" może zablokować wszystkie akcje gracza

Gameplay Effects – buffy, debuffy i modyfikatory

- Służą do zmiany atrybutów postaci (np. HP, siła, szybkość)
- Mogą działać natychmiastowo lub przez określony czas
- Obsługują efekty pozytywne (buffy) i negatywne (debuffy)
- Integrują się z replikacją i systemem tagów

Przykład: efekt zadający 20 obrażeń co 1 sekundę przez 5 sekund

•Zalety i wady użycia GAS w projekcie

Zalety	Wady
Skalowalny i elastyczny system do rozbudowanych mechanik	Wysoka krzywa uczenia się – szczególnie w C++
Gotowy do pracy w multiplayerze	Złożona struktura – nadmiar kodu dla prostych projektów
Ułatwia zarządzanie statystykami, cooldownami i efektami	Trudniejsza integracja z gotowymi blueprintami początkujących
Modułowa architektura wspierająca wielokrotne użycie kodu	Trudności z przepisywaniem kodu
	Brak Dokumentacji*

Alternatywy dla GAS

- **Custom Ability System** - Lepsze dla mniejszych projektów, gdzie potrzeba tylko podstawowej logiki.
- **Unity Animator i Event System** - Używanie **animacji i triggerów** do aktywacji umiejętności.
- **Modularne skrypty i klasy w C++** - Własne skrypty w C++ pozwalają na pełną kontrolę nad mechaniką gry i umożliwiają implementację prostszych systemów, jeśli GAS jest zbyt rozbudowany.
- **ScriptableObjects w Unity** - W Unity, **ScriptableObjects** pozwalają na przechowywanie danych związanych z umiejętnościami i efektami, które można łatwo modyfikować i przypisać do różnych obiektów w grze.

Alternatywy dla GAS – Kiedy go nie używać?

- **Proste gry** – Jeśli projekt jest mały, z prostymi mechanikami (np. gry 2D, casualowe), użycie GAS może być nadmiarowe.
- **Brak multiplayera** – Dla gier jednoosobowych, które nie potrzebują skomplikowanej synchronizacji w sieci, proste skrypty i klasy mogą wystarczyć.
- **Niskie wymagania techniczne** – Jeśli projekt nie wymaga zaawansowanego zarządzania zdolnościami, efekty i atrybuty mogą być obsługiwane bez GAS, używając prostych zmiennych i logiki.
- **Mała złożoność umiejętności** – Gdy mechaniki są bardzo proste, np. kilka umiejętności czy efektów, systemy takie jak GAS mogą być przesadą.
- **Brak Umiejętności** – Dla programistów nie potrafiących płynnie posługiwać się GAS praca z tym systemem będzie męczarnią.

GAS w Fortnite – zastosowanie w praktyce

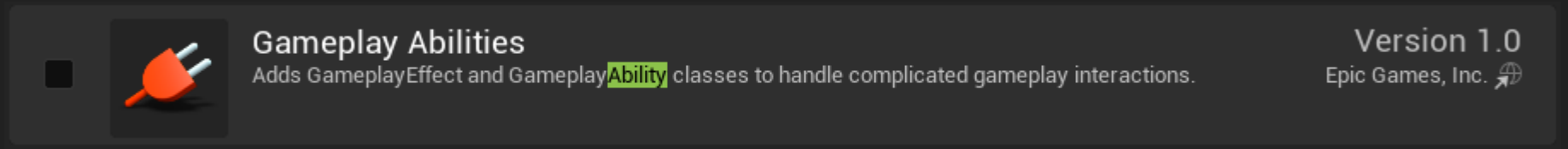
- **Zarządzanie umiejętnościami postaci** - umożliwia tworzenie i zarządzanie złożonymi zdolnościami, takimi jak budowanie, sprintowanie, specjalne ataki czy interakcje z otoczeniem.
- **Gameplay Effects** – Przykładem jest system zdrowia (HP), tarczy, obrażeń, leczenia oraz innych statystyk, które mogą być modyfikowane przez **Gameplay Effects**.
- **Dynamiczna replikacja w multiplayerze** - efekty i umiejętności graczy są automatycznie replikowane na wszystkich klientach, zapewniając synchronizację w trybie online.
- **Gameplay Tags do kontroli efektów i stanów** - Fortnite używa **Gameplay Tags** do kontrolowania stanów postaci, takich jak bycie w powietrzu, w ruchu czy w stanie powalenia.
- **Wielostopniowe systemy efektów** - efekty takie jak leczenie, zmniejszanie obrażeń, zwiększanie prędkości ruchu czy boosty są zarządzane przez GAS, pozwalając na skomplikowaną interakcję między umiejętnościami.

Porównanie GAS z prostymi systemami zdolności

Cecha	Gameplay Ability System (GAS)	Proste systemy zdolności
Skala i złożoność	Zaawansowany, do skomplikowanych systemów i gier multiplayerowych.	Prosty, idealny do małych projektów i podstawowych mechanik.
Elastyczność	Bardzo elastyczny, pozwala na tworzenie zaawansowanych mechanik, jak np. modyfikatory atrybutów, efekty w czasie rzeczywistym, cooldowny, etc.	Mniej elastyczny, ograniczony do prostych efektów i interakcji.
Wsparcie dla multiplayera	W pełni wspiera replikację i synchronizację między klientami w trybie multiplayer.	Często wymaga dodatkowego kodowania, aby wspierać multiplayer.
Zarządzanie statystykami	Umożliwia dokładne zarządzanie atrybutami postaci (np. zdrowie, mana) za pomocą **Attribute Set**.	Brak zaawansowanego systemu atrybutów – musisz to obsługiwać ręcznie.
System efektów	Zaawansowany system **Gameplay Effects**, który pozwala na dynamiczne modyfikowanie statystyk postaci, takich jak obrażenia, leczenie czy różne buffy/debuffy.	Proste efekty, często hardcodowane w kodzie, bez elastyczności.
Replikacja i synchronizacja	Automatyczna synchronizacja zdolności i efektów w trybie online dzięki wbudowanej replikacji.	Replikacja i synchronizacja wymaga samodzielnego kodowania.
Złożoność implementacji	Wymaga zaawansowanej wiedzy o Unreal Engine, C++ oraz systemach replikacji.	Łatwiejszy do implementacji, szczególnie w małych projektach.
Modularność	Bardzo modularny – różne systemy (zdolności, efekty, statystyki) mogą być tworzone i modyfikowane niezależnie.	Zazwyczaj nie jest tak modularny, trudniejszy do rozbudowy.
Zastosowanie	Idealny do gier RPG, MOBA, battle royale i innych, gdzie zdolności postaci są rozbudowane i wymagają dużej elastyczności.	Lepiej sprawdza się w grach o prostych mechanikach, jak platformówki czy gry 2D.
Czas rozwoju	Długi czas wdrożenia z uwagi na złożoność i konieczność dopasowania do gry.	Krótszy czas rozwoju, idealny do prototypów i małych projektów.

GAS - Inicjalizacja

Inicjalizacja w silniku



Inicjalizacja w kodzie rozpoczyna się w pliku Build gdzie niezbędne jest dodanie tagów, tasków i GameplayAbilities.

```
PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "InputCore", "EnhancedInput", "GameplayAbilities"});

PrivateDependencyModuleNames.AddRange(new string[] { "GameplayTags", "GameplayTasks" });
```

GAS – Ability System Component

W zależności od implementacji ASC może być bardzo rozbudowane lub ograniczać się do zaledwie kilku linijek.

```
DECLARE_MULTICAST_DELEGATE_OneParam(FEffectAssetTags, const FGameplayTagContainer& /*AssetTags*/);

/**
 *
 */
UCLASS()
class AURA_API UAuraAbilitySystemComponent : public UAbilitySystemComponent
{
	GENERATED_BODY()

public:
	void AbilityActorInfoSet();

	FEffectAssetTags EffectAssetTags;

protected:
	void EffectApplied(UAbilitySystemComponent* AbilitySystemComponent, const FGameplayEffectSpec& EffectSpec, FActiveGameplayEffectHandle ActiveGameplayEffectHandle);
};
```

GAS – Ability System Component

```cpp
#include "AbilitySystem/AuraAbilitySystemComponent.h"

void UAuraAbilitySystemComponent::AbilityActorInfoSet()
{
	OnGameplayEffectAppliedDelegateToSelf.AddUObject(this, &UAuraAbilitySystemComponent::EffectApplied);
}

void UAuraAbilitySystemComponent::EffectApplied(UAbilitySystemComponent* AbilitySystemComponent, const FGameplayEffectSpec& EffectSpec, FActiveGameplayEffectHandle ActiveGameplayEffectHandle)
{
	FGameplayTagContainer TagContainer;
	EffectSpec.GetAllAssetTags(TagContainer);

	EffectAssetTags.Broadcast(TagContainer);

}
```

GAS - Replikacja

Replication Mode	When to Use	Description
`Full`	Single Player	Every `GameplayEffect` is replicated to every client.
`Mixed`	Multiplayer, player controlled `Actors`	`GameplayEffects` are only replicated to the owning client. Only `GameplayTags` and `GameplayCues` are replicated to everyone.
`Minimal`	Multiplayer, AI controlled `Actors`	`GameplayEffects` are never replicated to anyone. Only `GameplayTags` and `GameplayCues` are replicated to everyone.

GAS – AttributeSet

```
#define ATTRIBUTE_ACCESSORS(ClassName, PropertyName) \
        GAMEPLAYATTRIBUTE_PROPERTY_GETTER(ClassName, PropertyName) \
        GAMEPLAYATTRIBUTE_VALUE_GETTER(PropertyName) \
        GAMEPLAYATTRIBUTE_VALUE_SETTER(PropertyName) \
        GAMEPLAYATTRIBUTE_VALUE_INITTER(PropertyName)
```

```
UPROPERTY(BlueprintReadOnly, Category = "Health", ReplicatedUsing = OnRep_Health)
FGameplayAttributeData Health;
ATTRIBUTE_ACCESSORS(UGDAttributeSetBase, Health)
```

```
UFUNCTION()
virtual void OnRep_Health(const FGameplayAttributeData& OldHealth);
```

GAS – AttributeSet

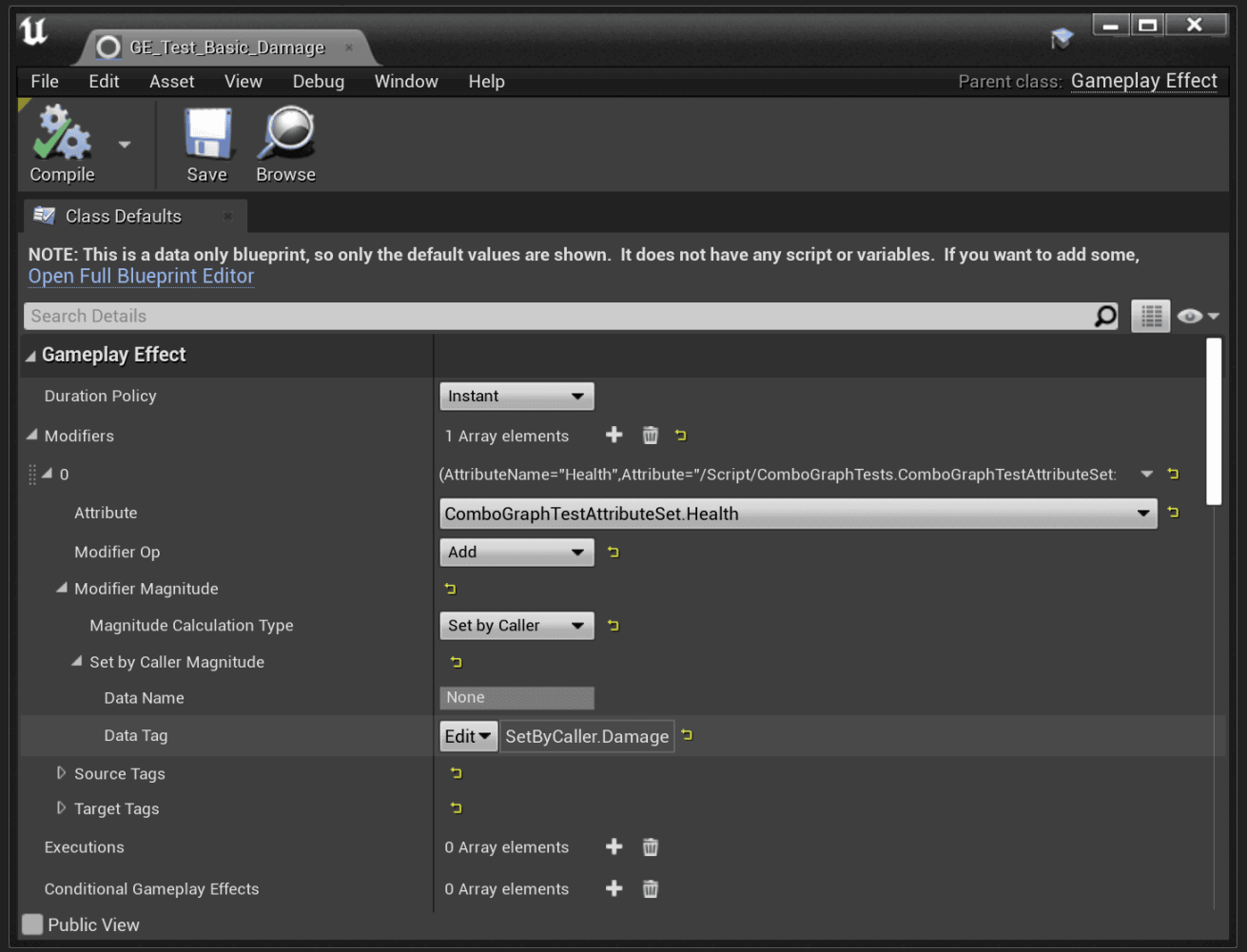
```cpp
void UGDAttributeSetBase::OnRep_Health(const FGameplayAttributeData& OldHealth)
{
	GAMEPLAYATTRIBUTE_REPNOTIFY(UGDAttributeSetBase, Health, OldHealth);
}
```

```cpp
void UGDAttributeSetBase::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const
{
	Super::GetLifetimeReplicatedProps(OutLifetimeProps);

	DOREPLIFETIME_CONDITION_NOTIFY(UGDAttributeSetBase, Health, COND_None, REPNOTIFY_Always);
}
```

GAS – GameplayEffects

Duration Type	GameplayCue Event
`Instant`	Execute
`Duration`	Add & Remove
`Infinite`	Add & Remove



GAS – Dodanie do gracza

```
APAPlayerControllerBase::AcknowledgePossession(APawn* P)

	Super::AcknowledgePossession(P);

	APACharacterBase* CharacterBase = Cast<APACharacterBase>(P);
	if (CharacterBase)
	{
		CharacterBase->GetAbilitySystemComponent()->InitAbilityActorInfo(CharacterBase, CharacterBase)
	}
```

```
void APACharacterBase::PossessedBy(AController * NewController)
{
	Super::PossessedBy(NewController);

	if (AbilitySystemComponent)
	{
		AbilitySystemComponent->InitAbilityActorInfo(this, this);
	}

	// ASC MixedMode replication requires that the ASC Owner's Owner be the Controller.
	SetOwner(NewController);
}
```