Aura – Top Down RPG

Unreal Engine 5 Gameplay Ability System Tutorial by Stephen Ulibarri – documentation

* C++ Classes
  + **UObject** based
    - UAuraWidgetController – A base Widget Controller class
      * *UOverlayWidgetController* – Overlay controller, UCLASS(BlueprintType, Blueprintable)
      * *UAttributeMenuWidgetController – Attribute menu controller*
  + **AActor** based
    - *AAuraEffectActor* – A generic class applying Gameplay Effect
  + **ACharacter** based
    - *AAuraCharacterBase* – A base class for all characters, there will be no instances of this class (UCLASS(Abstract))
      * *AAuraCharacter* – A class for player character
      * *AAuraEnemy* – A class for enemy characters
  + **APlayerController** based
    - *AAuraPlayerController* – A class used to control player character
  + **AGameModeBase** based
    - *AAuraGameModeBase* – A class used to tie together GameMode related classes
  + **APlayerState** based
    - *AAuraPlayerState* – A class used to create default player state, it is associated with main character pawn and contains Ability System Component and Attribute Set
  + **UAbilitySystemComponent** based
    - *UAuraAbilitySystemComponent* – A class used as base Ability System Component
  + **UAttributeSet** based
    - *UAuraAttributeSet* – A class used as Attribute Set for all characters
  + **UUserWidget** based
    - *UAuraUserWidget* – A generic User Widget class
  + **AHUD** based
    - *AAuraHUD* – A class used to create a default HUD
* C++ Interfaces
  + IEnemyInterface – An Interface used for some Enemy interactions like highlighting
* C++ Structs
  + FAuraGameplayEffect - Holds needed information for each Gameplay Effect used by Effect Actor
  + FEffectProperties – Holds information about entities involved in given gameplay effect, data is filled in PostGameplayEffectExecute
  + FUIWidgetRow (FTableRowBase) – Row structure containing pop-up information data displayed to user
* C++ Enums
  + EEffectApplicationPolicy - Determines Application Policy of given Gameplay Effect
  + EEffectRemovalPolicy - Determines Removal Policy of given Infinite Gameplay Effect
* Blueprints
  + ***BP\_AuraCharacter*** (AAuraCharacter) – Player Character
    - Contains Camera on Spring Arm
  + ***BP\_EnemyBase*** (AAuraEnemy) – base class for all enemies
  + ***BP\_Goblin\_Spear*** (BP\_EnemyBase) – melee enemy
  + ***BP\_Goblin\_Slingshot*** (BP\_EnemyBase) – ranged enemy
  + ***BP\_AuraPlayerController*** (AAuraPlayerController) – controller blueprint
    - Contains Input related information (Input Mapping Context, Input Action(s))
  + ***BP\_AuraGameMode*** (AAuraGameModeBase) – game mode blueprint
    - Contains game mode information (Default Pawn, HUD, Player Controller and State Classes)
  + ***BP\_AuraPlayerState*** (AAuraPlayerState) – player state
  + ***BP\_HealthPotion*** (AAuraEffectActor) – applies gameplay effect (restore health)
  + ***BP\_ManaPotion*** (AAuraEffectActor) – applies gameplay effect (restore mana)
  + ***BP\_HealthCrystal*** (AAuraEffectActor) – applies gameplay effect (restore health over time)
  + ***BP\_ManaCrystal*** (AAuraEffectActor) – applies gameplay effect (restore mana over time)
  + ***BP\_FireArea*** (AAuraEffectActor) – applies gameplay effect (deal damage over time)
  + ***BP\_TestActor*** (AAuraEffectActor) – used to test gameplay effects
  + ***BP\_OverlayWidgetController*** (UOverlayWidgetController) – Overlay controller blueprint
* Attributes
  + Primary
    - Strength – Increases Physical Damage
    - Intelligence – Increases Magical Damage
    - Resilience – Increases Armor and Armor Penetration
    - Vigor – Increases Health
  + Secondary
    - Armor – Reduces Damage taken, increases block chance
      * Based on RESilience
      * (RES+2) \* 0.25 + 6
    - Armor Penetration – Ignores percentage of enemy Armor
      * Based on RESilience
      * (RES+1) \* 0.15 + 3
    - Block Chance – Chance to cut incoming damage in half
      * Based on ARMor
      * ARM \* 0.25 + 4
    - Critical Hit Chance – Chance to double Damage plus Critical Hit bonus
      * Based on Armor PENetration
      * APEN \* 0.25 + 2
    - Critical Hit Damage – Bonus Damage added when Critical Hit is scored
      * Based on Armor PENetration
      * APEN \* 1.5 + 5
    - Critical Hit Resistance – Reduces Critical Hit Chance of attacking enemies
      * Based on ARMor
      * ARM \* 0.25 + 10
    - Health Regeneration – Amount of health regenerated every second
      * Based on VIGor
      * VIG \* 0.1 + 1
    - Mana Regeneration – Amount of mana regenerated every second
      * Based on INTelligence
      * INT \* 0.1 + 1
    - Max Health – Maximum amount of Health obtainable
      * Based on VIGor
      * VIG \* 2.5 + 80
    - Max Mana – Maximum amount of Mana obtainable
      * Based on INTelligence
      * INT \* 2 + 50
  + Vital
    - Health – Damage a character can take before dying
    - Mana – Resource used to cast spells
* Gameplay Effects
  + Health Potion – Restore health
    - Instant
    - Tags
      * Asset Tags
        + Message.HealthPotion
  + Mana Potion – Restore Mana
    - Instant
    - Tags
      * Asset Tags
        + Message.ManaPotion
  + Crystal Heal – Restore Health
    - Duration (Periodic)
    - Tags
      * Asset Tags
        + Message.HealthCrystal
  + Crystal Mana – Restore Mana
    - Duration (Periodic)
    - Tags
      * Asset Tags
        + Message.ManaCrystal
  + Fire Area – Deals damage
    - Infinite (Periodic)
  + Attributes – instant, override
    - GE\_AuraPrimaryAttributes – initializes primary attributes
    - GE\_AuraSecondaryAttributes – initializes secondary attributes
    - GE\_AuraVitalAttributes – initializes vital attributes
  + Test
    - GE\_TestAttributeBased – used to test how Attribute based effects work
* Important plugins
  + Enhanced Input
  + Gameplay Abilities
* Enhanced Input
  + **Mapping Contexts**
    - *IMC\_AuraContext*
      * Move to the right – No modifiers
      * Move to the left – Modifiers - Negate
      * Move forward – Modifiers – Swizzle YXZ
      * Move backward – Modifiers – Negate, Swizzle YXZ
  + **Input Actions**
    - *IA\_Move* – Axis2D (Vector2D)
* [Gameplay Ability System](https://docs.unrealengine.com/5.3/en-US/gameplay-ability-system-for-unreal-engine/)
  + Main parts:
    - Ability System Component – Can be added to Actor and handles most important things like granting abilities and activating them, handling notifications when abilities are activated or effects are applied.
    - Attribute Set – has a number of capabilities that allow to associate attributes with various parts of GAS system.
    - Gameplay Ability – Class that encapsulates some sort of functionality of thing that a character or object can do in a game (attacking, casting spells). They run asynchronous tasks.
    - Ability Task – Allow to execute code that runs asynchronous.
    - Gameplay Effect – Used to change the values of attributes. Directly, over time, periodically. Can associate attribute changes with various calculations with other parameters and attributes
    - Gameplay Cue – Cosmetic effects like particle systems and sounds and replicate them.
    - Gameplay Tag – Identifying numerous things and their hierarchical nature. More versatile than simple variables such as Enums, Booleans and strings.
  + Ways of adding Ability System Component:

|  |  |
| --- | --- |
| Directly on Pawn | On other class associated with Pawn |
|  |  |
| ASC and AS exist on Pawn, so when it is destroyed so are ASC and AS. If you want to reuse them with previous values you have to save it somewhere else and reload, otherwise they are recreated with default values. May be used on simple pawns like enemies which don’t need Player State. | When Pawn associated with Player State is destroyed ASC and AS stay on Player State because it persists, so you can spawn a new pan and associate it with existing Player State with previous ASC and AS values. Other reason is when you don’t want to clutter character with ASC and AS when you may wish to swap character but keep ASC and AS and you don’t want them to be unique to one character. |

* + Aura approach:
    - Enemies have ASC and AS directly on their pawn
    - Player Controller Character has ASC and AS on Player State associated with their pawn
  + GAS has its own static library – UAbilitySystemBlueprintLibrary it may be used both in code and blueprint event graph
  + Lots of GAS elements have the concept of level – abilities, gameplay effects, etc. We can scale the magnitude by some value based on level. That solution helps to implement scaling and difficulty.
    - Curve Table – Excel like spreadsheet, where we can associate 2 values (level, magnitude), there is also Graph View which shows curve based on added values/pairs. Curve Table may have multiple curves. When we use Curve Table as a magnitude, flat value becomes another multiplier for the final value of the magnitude
* Ability System Component
  + Always know the actor info such as who own it. It also understand the concept that it might be owned by some Actor, Pawn or another type of entity like Player State – that’s why it has two variables:
    - Owner – what class actually owns the ASC
    - Avatar – representation in the world associated with this ability system
    - Note: Owner and Avatar might be the same entity (enemy case), or different (player case) Obraz zawierający tekst, zrzut ekranu, diagram, Czcionka

      Opis wygenerowany automatycznie
    - We decide what is Owner and Avatar – *InitAbilityActorInfo(Owner, Avatar)*

*Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Opis wygenerowany automatycznie*

* + - * Must be done after possesion (the controller has been set for the Pawn)
      * Player controlled character:
        + ASC directly on Pawn:

Server – PossessedBy()

Client – AcknowledgePossession()

* + - * + ASC on PlayerState:

Server – PossessedBy()

OnRep\_PlayerState() – replication happens after PlayerState has been set on the server. As soon as PlayerState is set on the server it is replicated down to clients.

* + - * AI controlled character:
        + For both client and server – BeginPlay(), ASC and AS are set on Pawn
  + It is easy to bind to AttributeValueChange delegate - AbilitySystemComponent->GetGameplayAttributeValueChangeDelegate, it’s not dynamic delegate
* Attribute Set
  + When created alongside the ASC inside the owner class it is automatically registered with it. ASC has access to it and any other attribute sets that are registered with it
  + There may be multiple Attribute Sets but they must be different classes
  + If attributes depend on each other it is simpler to have them in one class
  + Attribute Set stores Attributes, attributes are floats and exist within a structure called FGameplayAttributeData
  + Attribute Set supervises attributes and responds to changes with any functionality that we like
  + Attribute values can be set directly in code, but preferred way to change them is by applying Gameplay Effects – mainly because they allow us to predict (more in notes) changes to attributes
  + Attributes consist of two values:
    - Base value – permanent value of an attribute
    - Current value – base value + temporary modifications caused by Gameplay Effects (Buffs and Debuffs)
  + Adding an Attribute to Attribute Set
    - Create variable FGameplayAttributeData SomeAtt
    - UPROPERTY(ReplicatedUsing = OnRep\_SomeAtt) – UE convention OnRep\_Name
    - Create UFUNCTION() OnRep\_SomeAtt(const FGameplayAttributeData& OldSomeAtt)
      * When function is called SomeAtt will be new value, but we still know what the old value was thanks to passed in OldSomeAtt
    - Add macro GAMEPLAYATTRIBUTE\_REPNOTIFY(ASClassName, AttributeName, OldValue) in OnRep function. It allows server to keep track of changes and roll back if prediction was wrong.
    - Override GetLifetimeReplicatedProps
    - In GetLifetimeReplicatedProps add macro DOREPLIFETIME\_CONDITIONNOTIFY(ASClassName, AttributeName, COND\_None, REPNOTIFY\_Alway)
  + Attribute accessor – ATTRIBUTE\_ACCESSORS(ASClassName, AttributeName), it creates:
    - 2 getters
      * FGameplayAttribute - struct
      * Float – current value
    - Set – get base value
    - Init – set base and current value
  + UAttributeSet functions
    - PreAttributeChange
      * Changes to Current Value, before the change happens
      * Triggered by changes to Attributes by Attribute Accessors or Gameplay Effects
      * Doesn’t permanently change the modifier, just the value returned from querying the modifier
      * Later operations recalculate the Current Value from all modifiers, we need to clamp again
    - PostGameplayEffectExecute
      * Parameter FGameplayEffectModCallbackData contains information about the effect that was applied
        + Data contain information about Effect Spec, Evaluated Data and Target’s Ability System Component
        + We have access to every entity involved in given gameplay effect
        + Here we can clamp values properly
  + To initialize Attributes we can:
    - set AbilitySystemComponent to VisibleAnywhere in AuraPlayerState and in Blueprint editor we can set Default Starting Data – DataTable with correct row structure (AttributeMetaData) – AttributeMetaData is still work in progress and it is good only to initialize values, it doesn’t implement any additional functionality yet
    - Apply Gameplay Effect
      * We must create proper Gameplay effect and apply it to the character
      * We can only initialize attribute values on client side because they are all replicated, if we initialize them both on server and client side (it is also correct) we just won’t have to wait for the server to replicate data back down
      * Modifier operation used in this situation is override
* Gameplay Effect
  + It’s an object of type UGameplayEffect. Used to change Attributes and Gameplay Tags. Gameplay effect is Data only, they have no logic. We create blueprints based on the new gameplay effect class. We don’t subclass the gameplay effect into child classes. Gameplay effects change attributes through modifiers and executions.
    - Modifiers – various possibilities, including complex custom calculation for when attribute changes. They specify a type of operation to be performed on the attribute in question. They take a value called magnitude and use magnitude to change the attribute in a way that depends on the modifier operation
      * Add – can be negative to subtract a value
      * Multiply
      * Divide
      * Override – replaces the attribute value with the value provided for the magnitude
    - The Magnitude used in these operations is produced from the magnitude calculation
      * Scalable Float – simplest type, you specify hardcoded value for the magnitude directly or by using table which scales magnitude based on gameplay effect level
      * Attribute based – the magnitude is calculated using attribute value(s) on which other calculations may be applied like division, multiplication or addition
      * Custom Calculation Class (MMC) – Class designed to capture other values like attributes, experience level or any other value
      * Set by Caller – key-value pair, assigning a magnitude associated with name or a gameplay tag. Useful when setting the modifiers magnitude based on code logic at the time that we crate the gameplay effect or apply it.
      * Custom executions – Gameplay Effect Execution Calculation – Exec Calc, these can change more than one attribute and they can do anything else that we choose to implement, it is the most powerful way to modify attributes
  + Gameplay effects have Duration Policy:
    - Instant – modifier is applied instantly in a one off action, changes base value
    - Duration – modify attribute for a set period of time after which the modification is undone, changes current value
    - Infinite – modify attribute and doesn’t end until we manually remove it, changes current value
    - Periodic – Type of Duration and Infinite effects, it apply change periodically and the change isn’t after gameplay effect is removed, changes base value
      * To make gameplay effect periodic we set Period > Period to not 0 value. This value represents in what period the effect will be applied (1 – every 1 second, 0.1 every 0.1 second, etc.).
      * Period > Execute Periodic Effect on Application – decide if effect is applied instantly or after first period - if our effect is “add 2 health with duration 2, period 1 and set true to Execute Periodic Effect on Application” the effect will add 6 health overall (2 on application and 2 every second for 2 seconds)
      * Inhibition policy -
  + Gameplay effects may stack with their own stacking policy, add gameplay tags, gran abilities
  + Gameplay effects can be applied directly but usually we create more lightweight version of them called Gameplay Effect Spec. Concept of Spec is common in GAS and is a form of optimization
    - Spec contains thee bare bones information needed to perform the modification and the only actual instance of the gameplay effect class that typically gets instantiated is the class default object
    - Carries information about gameplay tags that the effect has
    - Has an effect context- additional class that can store more information about the effects
    - References to causer and target of the effect which can be used to determine what happens to the overall magnitude calculation and trigger various gameplay mechanics to any actors involved in this effect
  + Gameplay effects are often applied by Gameplay Abilities but it is not necessary – An actor may have a gameplay effect which is applied to other actor which overlaps with it
  + Applying Gameplay Effect in code
    - Get Target ASC
    - Create FGameplayEffectContextHandle – ASC->MakeEffectContext()
    - Add source to EffectContextHandle - .AddSourceObject(EffectActor)
    - Create FGameplayEffectSpecHandle – ASC->MakeOutgoingSpec(GE class, Level, EffectContextHandle)
    - Create FActiveGameplayEffectHandle – ASC->ApplyGameplayEffectSpecToSelf(\*EffectSpecHandle.Data.Get())
  + Stacking
    - Stacking type
      * None – Every effect is applied without any care for stacking
      * Aggregate by… We set Stack limit which describes how many stacks of the effect we can get
        + Source – stacks are tracked by source of the effect, so if stack limit is 2 and the source already applied 2 stack the 3 one is not applied, if there is another source of the effect we may have 4 total stacks of effect – 2 per every source
        + Target – stacks are tracked by target of the effect, so if stack limit is 2 and the target already have 2 stack the 3 one is not applied, no matter how many sources there are target can only have 2 stacks of the effect
    - Refresh Duration Policy – how effect duration should be refreshed while stacking
      * Refresh on successful application – duration is refreshed by any successful application
      * Never refresh – the duration doesn’t refresh so if effect duration is 1 seconds and we apply it and after 0.5 second reapply the same effect – the 1st one will last whole second but the second one will end as soon as 1st effect ends so It will only last for 0.5 second.
    - Refresh Period Policy – how effect period should be refresh while stacking
      * Reset on successful application – period is reset it means that after application we must wait for another full period to apply effect. If applied effect has period of 1 and we reapply the effect half time before next period we must wait 1 whole second to get that periodic application – we wait a total of 1.5 second for that one periodic application
      * Never reset – period isn’t reset, so taking previous example after second application we only wait half time of period to get next periodic application
    - Expiration Policy
      * Clear entire stack – if effect expires whole stack is cleared
      * Remove single stack and refresh duration – when effect expires we remove 1 stack and refresh duration. If we have 2 stacks healing 15 each, we get healed by 15, 1 stack is removed, and after another 15 another 1 stack is removed
      * Refresh duration – works like infinite – after 1 stack expires we refresh duration but we don’t remove that stack. It can be used to handle stack decrements using OnStackCountChange callback
  + Modifier order of operations
    - Add – add to calculated value
    - Multiply – multiply calculated value
    - Divide – divide calculated value

Obraz zawierający tekst, zrzut ekranu, Oprogramowanie multimedialne, oprogramowanie

Opis wygenerowany automatycznie

* + Attribute Based Magnitude calculations
    - Attribute based modifiers
      * Coefficient – second operation – multiplies the result of Multiply Value + Attribute Value
      * Pre Multiply Additive Value – first operation – adds a value to attribute value
      * Post Multiply Additive Value – third operation – adds a value to the result of (Multiply Value + Attribute Value) X Coefficient

Obraz zawierający tekst, zrzut ekranu, Oprogramowanie multimedialne, oprogramowanie

Opis wygenerowany automatycznie

* + - Backing Attribute
      * Attribute to Capture – attribute on which our Gameplay Effect is based
      * Attribute Source (Source/Target) – Source from which we capture the attribute
      * Snapshot – when to capture the attribute
        + True
        + False
* Gameplay Tags

Obraz zawierający tekst, zrzut ekranu, Oprogramowanie multimedialne, oprogramowanie

Opis wygenerowany automatycznie

* + Names of type FGameplayTag and they are registered with the Gameplay Tag Manager. They are hierarchical in nature (Parent, Child, Grandchild), each level of the hierarchy is separated by a dot
  + Tags are FNames but thanks to hierarchical nature and implemented features they are more flexible than simple FNames.
  + Tags can be compared for exact equality or partial equality
  + GAS is designed to use gameplay tags in just about every single class. We are giving tags to actors which means we give a gameplay tag to an actor’s ability system component. ASC implements the IGameplayTagAssetInterface, this interface has a number of functions that make it easy to look at all owned tags, see if particular tag is owned, or get all tags that match given tag or check if any of the owned tags match a specified tag
  + We use Gameplay Tag Container to store Gameplay Tags, which has some gameplay tag specific functionality and some added efficiency. They have a concept of tag map count, meaning that you can have more than one instance of single tag in the container. It’s tag map count that tells us how many of a given tag exists in the container and at times it could be zero (e.g. when tag has been added and then removed), in debug mode we may see a tag in gameplay tag container with count of 0.
  + Gameplay Effects may contain tags that they grant to the target’s ASC.
    - Example: We apply duration GE, Now target’s ASC has tag that GE is granting, when GE expires it removes itself and that gameplay tag gets removed
  + Gameplay Tags may also be used to block using a Ability or ASC may have and ability which has a list of required tags which must be owned to use a ability
  + We can use Gameplay Tags for basically anything like inputs, abilities, attributes, damage types, buffs/debuffs, messages, data, etc.
  + Adding Gameplay Tags:
    - In editor Project Settings>Gameplay Tags:
      * Adding just new Gameplay Tag name
      * Using Data Table – choosing gameplay tag table row – Name and Comment. We assign Data Tables in Gameplay Tag Table List
    - Directly in .ini file
  + Setting Gameplay Tags
    - Gameplay Effect – Components (+)
      * Types
        + Combined tags – Inherited + Added - Removed
        + Added
        + Removed
      * Asset Tags – Tags that GE has
      * Granted Tags – Tags applied to actor that GE is applied to
  + To get Gameplay Tag if it exist in our config file we can use function from static library – RequestGameplayTag(FName(“Tag.Name”)), it return FGameplayTag if found
* Multiplayer
  + Types
    - Dedicated Server
      * No human player
      * No rendering to screen
    - Listen Server
      * Is human player(host
      * Host has no lag
  + Server is the Authority
    - *Correct* version of the game
    - *Important* things are made on the server
  + Structure

Obraz zawierający tekst, zrzut ekranu, Czcionka, oprogramowanie

Opis wygenerowany automatycznie

* + - Server
      * Game Mode
    - Clients
      * HUD and Widgets – each client has only their own HUD and widgets
    - Both
      * Player Controllers – The server has information about every Player Controller but each client has only information about their Player Controller
      * Player States and Pawns – All Player States exists on every Client and on the Server
  + Replication modes (EGameplayEffectReplicationMode)

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

* + - Minimal - Only replicate minimal gameplay effect info. Note: this does not work for Owned AbilitySystemComponents (Use Mixed instead).
    - Mixed - Only replicate minimal gameplay effect info to simulated proxies but full info to owners and autonomous proxies

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

* + - Full - Replicate full gameplay info to all
* UI
  + MVC, Model – View – (Widget) Controller , one way dependency

Obraz zawierający zrzut ekranu, tekst

Opis wygenerowany automatycznie

* + - Model – Numerical values (health, level, experience, unlocked abilities, assigned buttons). Model shouldn’t need to know which controller or widgets are being used to represent its data.
    - View – Display of visual effects of the data (health bars, ability icons, attributes). Depends on the controller.
    - (Widget) Controller – Responsible for calculations, retrieval and broadcasting data to view. Controller shouldn’t need to know which widgets are receiving broadcasted data.
  + Sub Widgets
    - WBP\_GlobeProgressBar – base Widget Blueprint for mana and health progress bars, we assign to Controller’s delegates and respond to proper broadcast on each Globe.
      * WBP\_HealthGlobe – WBP showing current health/max health
      * WBP\_ManaGlobe – WBP showing current mana/max mana
    - WBP\_EffectMessage – widget displaying a message to the screen
  + Widgets
    - WBP\_Overlay – Main widget for the whole screen, containing health and mana globes, abilities, buttons which open menus etc.
  + We assign to delegates in blueprint – cast, assign on DelegateName
  + DT\_MessageWidgetData - Data Table with FUIWidgetRow as a row structure has defined message data
* Notes
  + TObjectPtr<T> - behaves the same as a raw pointer (T\*), but has some addition features in editor:
    - Access tracking – how often the pointer is accessed or dereferenced
    - Lazy loading – asset can’t be loaded until it’s needed or used in the program
  + [FName, FText, FString](https://docs.unrealengine.com/5.3/en-US/string-handling-in-unreal-engine/)
    - FName – lightweight, case-insensitive, immutable, cannot be manipulated, given string is stored only once, even if its reused
    - FText – user-facing text
    - FString – can be searched, modified and compared that’s why FStrings are more expensive
  + Replication – In multiplayer, when and entity changes on the server that change will be replicated (sent down) to all connected clients, lag compensation.
    - When replicated variable is changed on the server, the next update that happens on server sends down server version of that variable to the clients – server value is the *correct* one
    - Replication only happens one way – from server to clients. That means that replication values shouldn’t be changed on client side because server (and other clients) won’t know about it.
    - GAS handles a lot of the multiplayer stuff
    - To make variable replicated we mark it in UPROPERTY()
      * Replicated
      * ReplicatedUsing = RepNotify – e.g. OnRep\_NAME
  + FInputModeGameAndUI – used to configure input behavior like locking mouse to viewport or hiding cursor. ( APlayerController – SetInputMode(IMGaUI)
  + check(Obj) – assert, crashes the game when false. Used when we don’t expect something to be nulltpr
  + checkf(Obj, TEXT(“”)) – check() + printing a formatted string to crash log if check fails
  + CastChecked<T>(O) – cast with check, crashes program if cast fails.
  + Checking nulltpr in if statement – we consider situations in which a pointer might be nulltpr due to e.g. function called too early before pointer is valid.
  + Cursor trace is tracing visibility channel
  + Custom Depth-Stencil Pass (Enabled with Stencil). Mesh > Rendering > Advanced > Render CustomDepth Pass, CustomDepth Stencil Value
  + NetUpdateFrequency – How often the server will try to update clients (number of max tries per second)
  + OnRep\_NAME – Rep notifies – functions called as a result of something being replicated
  + Prediction (Attributes and Gameplay Effects) Obraz zawierający tekst, Czcionka, zrzut ekranu

    Opis wygenerowany automatycznie
  + Be careful with Canvas Panel as it is more expensive than Size Box and Overlay
  + More clear “Cast Getters” – GetController<AAuraPlayerController>()==Cast<AAuraPlayerController>(GetController())
  + Delegates
    - Dynamic – events are assignable in blueprint, when we bind to Dynamic we can use .AddDynamic()
    - Not Dynamic – we can bind with .AddUObject()
    - Multicast – multiple object may bind to this delegate
  + BlueprintType – can be used as a type in event graph (e.g. to cast)
  + Blueprintable – can make a blueprint based on this class
  + GAS uses Handles as wrappers, Handles are lightweight things that contains the pointer to the full object (e.g. FGameplayEffectContextHandle has a pointer to GameplayEffectContext)
  + Remember to remove Static Mesh collisions for actors like potions\
  + Stacking type of GE influence how Tags are stacked, GE stack – Tag count doesn’t go up, GE doesn’t stack – Tag count goes up
  + Instant GE doesn’t really mean for instant effect, but we can use it anyway when GE is processed, so we can for example show something on screen
  + Blueprint – Add custom event
  + Notes

Obraz zawierający tekst, zrzut ekranu, pismo odręczne, Czcionka

Opis wygenerowany automatycznie

* + AA
* To understand better
  + Post process material/volume
  + Replication