Unreal Engine 5 - Lesson 1 - Introduction to Unreal Engine

Nicolas Serf - Gameplay Programmer - Wolcen Studio
serf.nicolas@gmail.com

Summary

- Engine's strengths
- Interface
- Everything is asset
- File hierarchy
- Levels
- Materials
- Data structures

Engine’s strengths - Window system

- It may seems **trivial**, but window system of Unreal is a huge **benefit** for **productivity**
- It seems to lead other **engine** to go that way like **Unity**
- Indeed, each asset type can be open in a **different window** / **tab**. It means that **Material**, **Animation**, **UI Widget** will all open in different window and not in a **“main working tab”**
- When you open a **Material** while another is **already opened**, it will replace the **opened one**, but there is an option to open in another window

- Window system like all engine also offer **dynamic docking**
- It allows to **re-arrange** all the layout of the engine as you like

Engine's strengths - Blueprint, a scripting visual editor

- Probably one of the first choice when choosing Unreal at a first thought for some people : **Blueprint**
- Even for a **professional C++ team**, Blueprint, as we'll be able to see in the next lesson, is a **true strength** from the engine
  - Blueprint has been **developed** and **integrated** for many years in the engine
  - It became one of the **greatest strength** of the engine either for **R&D**, **fast iteration**, **accessibility**, etc...
  - It allows all member from the team to works on production at different level

- Blueprint is not only use in **Gameplay side**, it is the tool used for every **scripting editing**, and the node based **graph system** for every complex system like **Animation Blueprint**, **UMG**, **Material**, etc...

- In my opinion, Blueprint is **equally important to C++** in complex project. Because blueprint **doesn't require** to **recompile** the source code, it is **visual**, it is **accessible** and therefore, allows **designer** to works **alongside** to developer, making the game more **maintainable** and allows dev to focus on **bug fixing** and **complex system**.

Engine’s strengths - C++

- Even if **Blueprint** is important, **C++** is obviously one of the other **strength** and **reason** to choose the engine.
- Being an engine **built in C++**, it allows to understand it from **developers stand point** and modify it if needed.
- C++ language is one of the **most performant language** and also allows a lot of **paradigme** while developing
- **Unreal C++** wrapper for **macros**, **object**, **reflection system** is really heavy and even if it **complexify** a bit the learning, it makes a big difference in the long run and make **integration really smooth**
- It ease the process of switching from **various engines** as **C++** is quite **standard** for a lot of game engine.
  - Lumberyard
  - Cry Engine
  - In-house engine
  - Etc...

Engine’s strengths - Accessible sources and fees

- When you growth as a company and in **complexity** in project, having the engine source is really important
- Unreal Engine can be **built from source**, it means exactly what is means. It is possible to **modify** the **engine source**, to **correct bugs until Epic correct them**, **add new functionalities**, **tweaks it to your needs,** etc…
- Take care, **source access** doesn’t means **open source**, because when you are using the engine to make a game, you are in a **contract** with unreal which is quite **advantaging** for developers and also another strength
    - The engine is **free of charge** until you reach **1M** of earning
    - It is then **5% royalties**

Engine's strengths - Rendering

- We are going to talk about **new innovative** and **game changing** technologies mostly develop and release into **Unreal Engine 5**.

- **Nanite**
    - It is a **virtualized geometry system**, uses a new **internal mesh format** and **rendering technology**.
    - It is driving to **works only** on detail that can be **perceived**
    - Data format is **highly compressed** and support **fine-grained streaming** with **automatic LOD**
    - Frame budget no longer constrained by polycount
    - You can check more about Nanite here
- **Lumen & Global Illumination**
    - **Fully dynamic global illumination** and **reflection system**
    - **Infinite bounces** and **indirect specular reflection** in large and scalable environments
    - Enabled by **default** in UE5.
    - Integrate **well** with **Nanite**

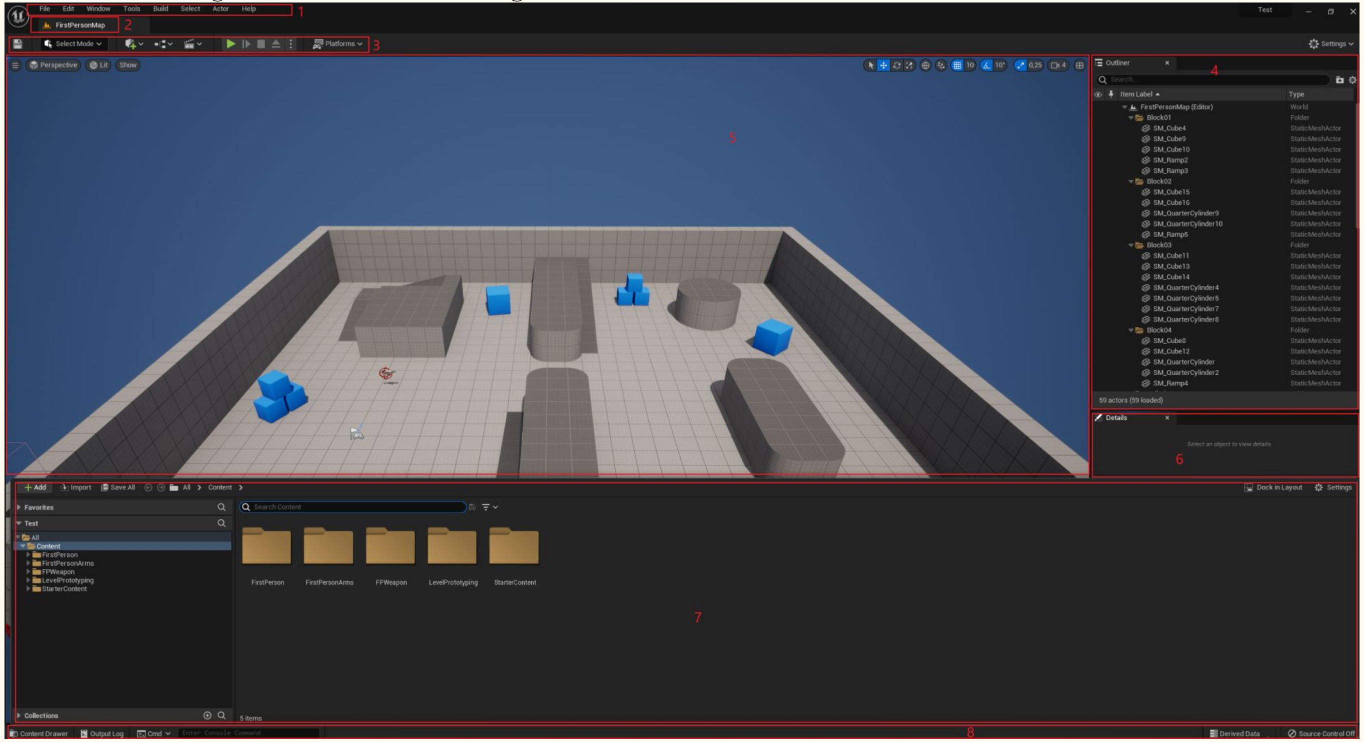
Engine's strengths - Battle tested & Plugins

- Strength list could continue but let's talk about a last point which is important
- Epic is **not only** an **engine developer**, it is also a **game developer** and one of the game developed by **Epic** is **Fortnite** which is one of the **most played game** in the world
  - It ensure that the engine is **battle tested**
  - The engine continues to **growth** in **functionalities**
  - **Bug fixing** will always be an important point
  - Forced to developed **finished product** to be used

- Plugins coming from Epic are also a **lot of libraries** of **high-professional quality**, there is a tons of them but here are some which are amazing and needs to be checked
  - Mass Entity
  - Gameplay Ability System (GAS)
  - Enhanced Input
  - Etc…
- It is important to note that some of this plugin will always **remains** as **plugins** because there are not needed for **every project** like GAS.
- But some others as been **developed** as **plugin** because there was not **production ready** but then, they get **integrated** in the **engine**.

Interface and navigation

- We’ll not go through all **possible interfaces** as it would be way to long and we’ll treat that **separately** when we’ll come to **some topics**
- For now, let's describe the general interface of the engine and the workflow



Interface and navigation

- 1 : **Engine Toolbar**
  - File : Create new level, load level, save, project manipulation
  - Edit : Editor settings, project settings, plugins
  - Window : Open various window for specific system, layout template, etc…
  - Tools : New C++ class, debugger, audit, merge, statistics, etc…
  - Build : All build options and settings
  - Select : Various selection option, geometry, additive, subtractive
  - Actor : Contextual menu when selecting an actor
  - Help : Documentation and external links
- 2 : **Tabs**
  - All tabs opened will appears there in line
- 3 : **Viewport Toolbar**
  - Save
  - Control mode in the viewport between select, painting, modeling, etc…
  - Level blueprint, place actor, level sequencer
  - Play / pause / stop play mode
  - Platform specification when playing, single or multiplayer, etc…
- 4 : **Outliner**
  - It is the representation of the world in an hierarchical view
  - You can organize elements as you want in it, by folders, changing the visibility, change attachment, etc…

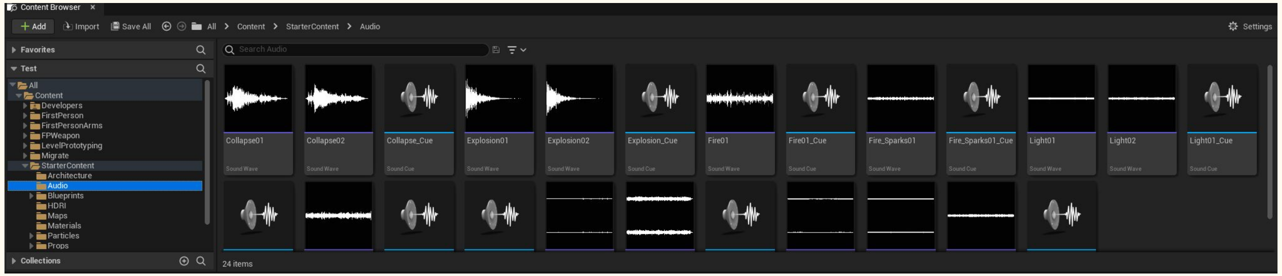
Interface and navigation

- 5 : **Viewport**
  - Viewport is the main place to design your levels, place actors, select elements, etc...
  - On the top left, you can configure how you want the viewport to be displayed
    - This settings are only used while in editor, when you click play, it is not taken into consideration
    - It can be used to show navmesh, lighting, etc...
  - On the right, important tool when manipulating actor
    - Move, rotate, scale
    - Possibility to scale the manipulation tool, offering a fine-grained when you want to precisely place some elements
- 6 : **Details panel**
  - It is a contextual menu and will change based on what is selected
  - It is an important window that will be present in kind of all window system (Material, UMG, etc...)
  - It allows to change properties of the element selected
- 7 : **Content Drawer**
  - Content drawer is your project file system
  - You can drag & drop element in it to import
  - Create favorite
  - Filter elements which is really important
- 8 : **Additional Toolbar**
  - Content Drawer : Open the content drawer
  - Output Log : Open log window
  - Cmd : Modified CMD type
  - Console command : Enter command that affect editor
  - Derived Data : Caches and statistics
  - Source Control : Allow to connect an Source Code Management (SCM) to the project

Everything is asset

- When you browse into **content drawer**, you'll see a lot of **elements** that are **contains** into folder
- Each **element** present in the content drawer is a **.usset** so an **asset** regarding Unreal.
- Assets are **binaries files** which are **not mergeable**
- This binaries assets when cooking for a build will be **translated** to a **comprehensible format** based on the **platform**, build settings, etc…

- You'll see that it is **not possible** to **import** an element for example a texture from a **project** to another by having **both project open** and **drag & dropping** from one content folder to another
  - It is not possible because Unreal **cannot import a .uasset** file.
  - In order to do so, you'll need to use the **migrate** button by right **clicking > Asset Actions > Migrate**.
  - It will query **which files** you want to **migrate** and a folder **where the migration** should be done.
  - **Content** is the main root of your project and a **migration** needs to be **done in it.**



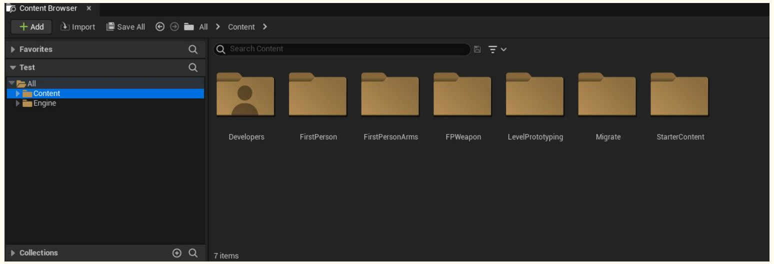
File Hierarchy

- Let's briefly talk more in details about **file hierarchy** in unreal
- Every asset are **located** in the **Content Folder.** It is the root for everything related to asset from material to Blueprint

- C++ files and modules are located in the **Source Folder.** Inside it is divided into 2 folders
  - **Engine** : All sources related to the engine
  - **Game** : **One directory per module** will be present contains sources of that module with **private (.cpp)** and **public (.h)**
    - In the **Content drawer** you'll not see that separation, but only the **gameplay class** header files

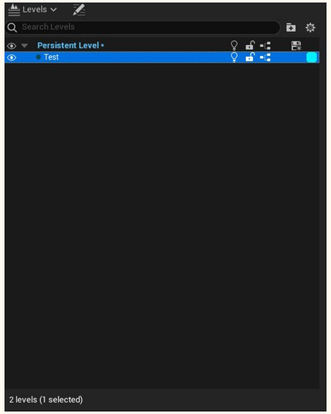
- **Developer Folder** is quite useful when you want to have some **test** blueprint, asset etc... that'll not be **shared** as developer folder is **unique** for **each person** working on the project and linked to the **local computer**

- Finally, let's talk about **redirectors**, when you move a file from a folder to another, Unreal creates a **redirector** in order to **ensure** that all **reference** to the asset that just get moved are **still correct.** It is a good practice to sometimes fix-up the redirectors by **right clicking on the folder**, and **Fix Up Redirectors in folder**



Levels

- **Level** is a type of **asset** that is used to represent a **world.**
- The **viewport** is **displaying** a **level**, and you can open a **level** to be **modified** in the **viewport** by simply **double clicking** it
- There is various denomination for level which is the standard unreal way to name then
  - Dojo : Testing purpose
  - Map : General naming convention
  - Scene : Unity convention
  - Etc...
- It is possible to works with additive / sublevel
  - It is more a work for **level designer**, but **thinking carefully** the **subleveling** of your game can allow **multiple people** to **works** on **several part** of the map like **audio**, **collision**, **geometry**, etc... instead of **locking** a **file** for a **single person**
  - To do so, open **Window>Levels** and navigate to add a new level. You can then just **double click** on a level to make it **"current"** modified one

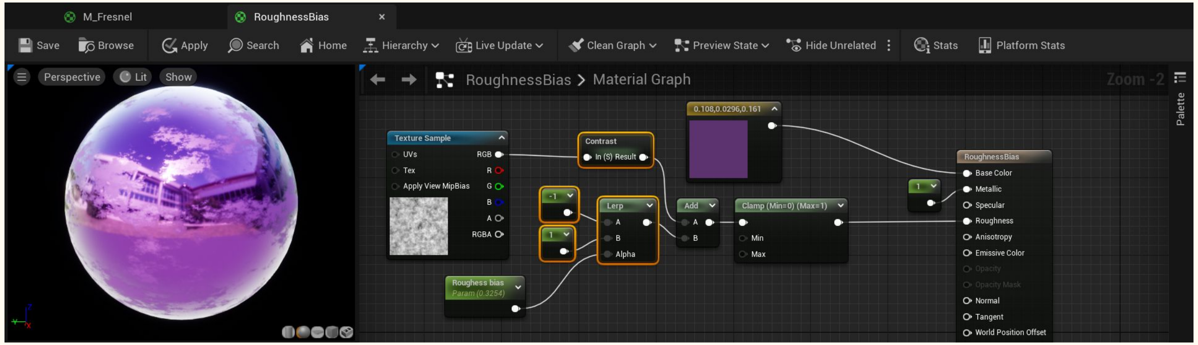


2 levels (1 selected)

Materials

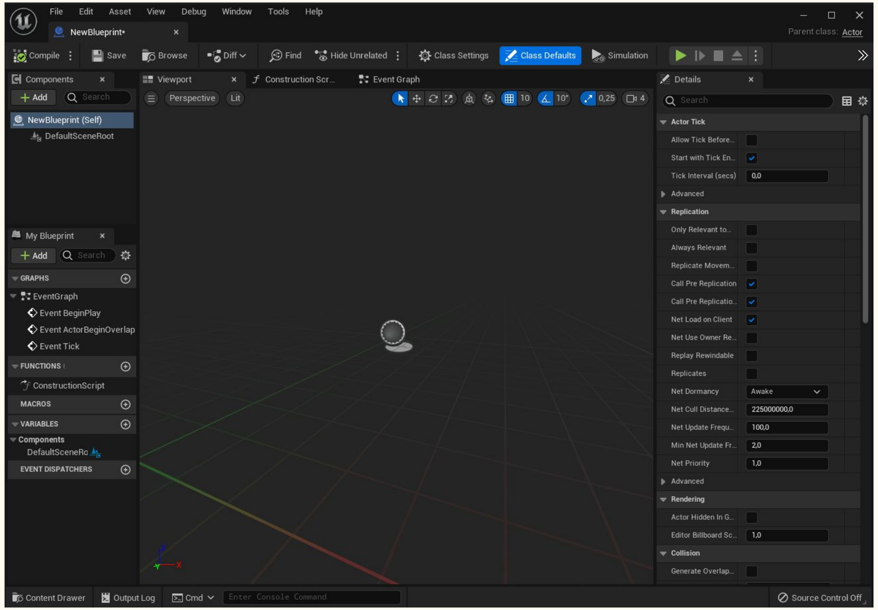
- **Material** is a really **complex system** and may require multiple **big lesson** in order to get an idea of all the **possibilities** with it
- Because the purpose of this first lesson is to get a **vague idea** of big **concept** in the engine, we'll simply show the **existence** of **materials**
- In a future lesson, we'll try to go into **more details** of what it is possible to do
- Keep in mind that Material are usually not made by **developer** but by **artists** or **technical artist**, so in the dedicated lesson we'll simply get the basics without going into **details** and not explaining how complex **shader programming** even with node graph can be

- As every game engine, **material** is a **central part** of the process. It allows to **apply** a **texture** into a **mesh**, and give additional **information** about this **mesh**, or part of the **mesh** when using **light map** on how it should behave regarding **lighting** and **environment**
- **Material** are a type of asset which can be created from the **Content Drawer** and applied to meshes



Actors

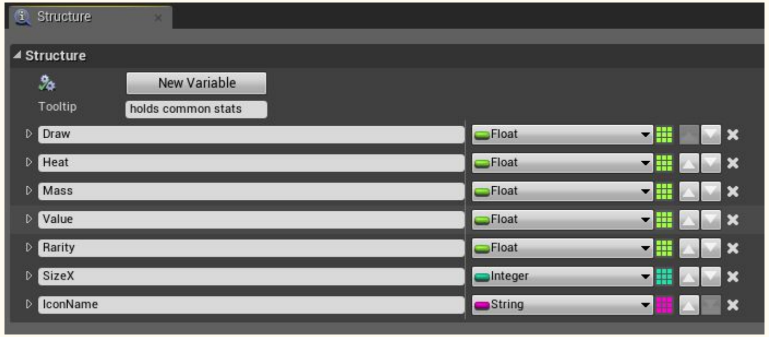
- **Actors** is the main type when you want to **create** something that'll be **placed** later on in your **level**
- You **cannot** place something that do not **inherit** from an **actor**
- In order to **create** an actor, **right click** in the content drawer, click **Blueprint Class** and from the **dropdown** select **Actor** or a **subclass** from it
- You can then simply **drag & drop** from the **content drawer** to the **viewport** an **actor blueprint class** and it'll be **spawned** in the level
- When **double clicking** on the **class**, it will open a **new window** which is the blueprint editor with 5 main tab
  - **Viewport** : It is the main place to modify visually your actor
  - **Construction Script** : It contains your initialization code for that actor
  - **Event Graph** : It contains your gameplay code and scripting
  - **Components** : A component is a piece that **compose** an actor, that can be **mesh**, a **system**, a **VFX**, a **sound player**, etc...
  - **Blueprint** : It contains all coding element you may needs from functions to variables
  - **Details** : It serves the same **purpose** as the **details** in main window, but this time it is **displaying** the **selection** on the **left** (**Component**, **variables**, **functions**, etc...)



Data structures

- **Data structures** is a key factor when developing a game.
- Either as a **small** or a **big** team, having a **proper** and **independent data structure** for **gameplay data** is a **priority** in order to allow the game to be **evolutive** and **maintainable**
- **Generally speaking**, as a **developer** data structure can be seen as how are your **member variables** organize into a **class**.
- You'll need to have a more **generic vision** over the **whole process** of game development

- Let's take for example **enemy definition** consisting of **Health**, **Attack Speed** and **Movement speed**
  - You may think only at **developer level**, saying that this variable should be **populated** directly from an **Entity class**
  - Even if this is **correct**, and obviously the **Entity** need those **values**, it needs to be **fed from somewhere else**
  - It needs to be **fed** from an **external sources** which gives the **entity** its **datas**
- This is where **Data Asset** or **Data Table** are useful
- What about having an asset that has the **single responsibility** of **holding** this data
  - It allows to **centralize data**
  - It allows for **designer** to not have to **worry** about what is happening in **source code**

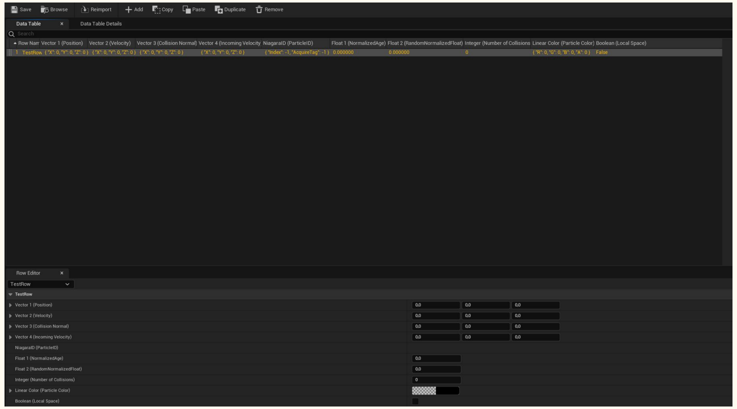


Tooltip: holds common stats

Draw	Float
Heat	Float
Mass	Float
Value	Float
Rarity	Float
SizeX	Integer
IconName	String

Data structures - Data tables

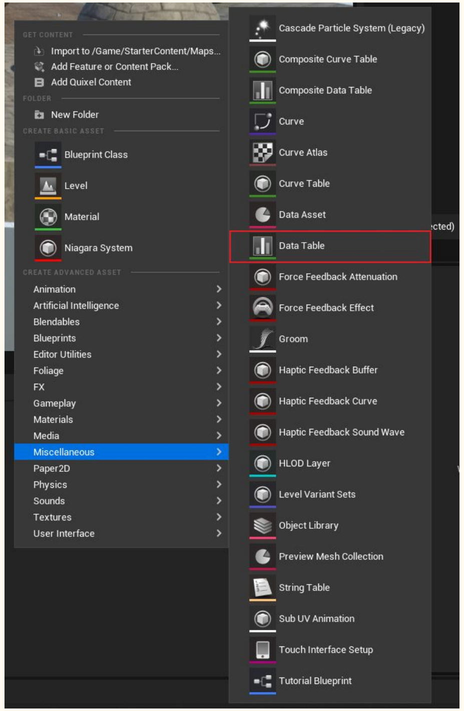
- **Data tables** is an asset type which is a collection of tuple in a table. It is a **UDataTable** in the framework
- Each entry being a **data collection** representing a given **struct** when you **create** the data table
- You can create a data table from **right clicking** in the **Content Drawer**.
- You can use any **struct blueprintable** as the data for tuples of the table
- Each row has a **row name** which can be **modified** and **represent** the **identifier** of that line in the table
- We'll not cover it in this lesson but it is also **possible** to **import** an **excel file** directly into a **Data Table.**
  - It may be needed if you are working with a **company** that has a lot of **process** in **Excel**
  - Team wants the possibility to **change game** values **without opening** the **engine**
  - Possibility to **export** this **data** into a **readable format** for an external agency (**translation**, etc...)
- In C++, your data structure must inherit from **FTableRowBase**
- To reference other Data Table rows, use **FDataTableRowHandle**



Row Nam	Vector 1 (Position)	Vector 2 (Velocity)	Vector 3 (Collision Normal)	Vector 4 (Incoming Velocity	NiagaraID (ParticleID)	Float 1 (NormalizedAge)	Float 2 (RandomNormalizedFloat)	Integer (Number of Collisions	Linear Color (Particle Color)	Boolean (Local Space)
1 TestRow	{ "X": 0, "Y": 0, "Z": 0 }	{ "X": 0, "Y": 0, "Z": 0 }	{ "X": 0, "Y": 0, "Z": 0 }	{ "X": 0, "Y": 0, "Z": 0 }	{ "Index": -1, "AcquireTag": -1 }	0.000000	0.000000	0	{ "R": 0, "G": 0, "B": 0, "A": 0 }	False

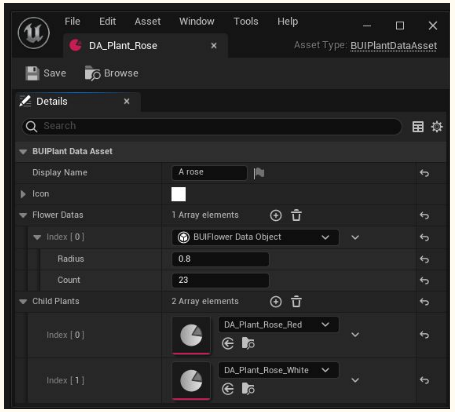
TestRow

TestRow			
Vector 1 (Position)	0,0	0,0	0,0
Vector 2 (Velocity)	0,0	0,0	0,0
Vector 3 (Collision Normal)	0,0	0,0	0,0
Vector 4 (Incoming Velocity)	0,0	0,0	0,0
NiagaraID (ParticleID)			
Float 1 (NormalizedAge)	0,0		
Float 2 (RandomNormalizedFloat)	0,0		
Integer (Number of Collisions)	0		
Linear Color (Particle Color)	▨		
Boolean (Local Space)			



Data structures - Data Asset

- It is possible to create a **Data asset base** from **blueprint**, by inheriting your blueprint from **PrimaryDataAsset.**
- In **C++,** you'll simply inherit from **UDataAsset**
- **Primary** and **secondary asset** is an other subject that will be tackled later
- You'll then **use that class** to **create** your **Data Asset.**
- There is a significant different between creating a **Blueprint subclass** of **UDataAsset** and creating an **Asset Instance** of a **UDataAsset**
- Obviously, **subclassing** is **possible** as the base of the Asset Instance is a class.
- It is good to know that if you want to **modify multiple Data Asset** at the same time, a bit like **Data Table** but with some **limitation** on **types** handled (like asset references), you can use **"Asset Actions > Bulk Edit via Property Matrix"** tool

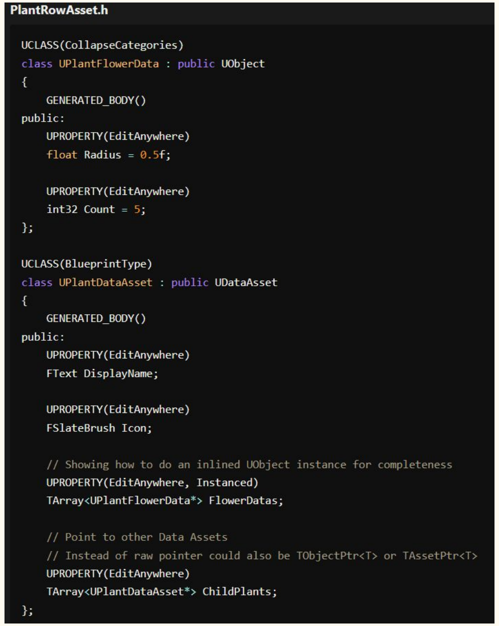


DA_Plant_Rose

Asset Type: BUIPlantDataAsset

BUIPlant Data Asset

Display Name	A rose
Icon	
Flower Datas	1 Array elements
Index [0]	BUIFlower Data Object
Radius	0.8
Count	23
Child Plants	2 Array elements
Index [0]	DA_Plant_Rose_Red
Index [1]	DA_Plant_Rose_White



PlantRowAsset.h

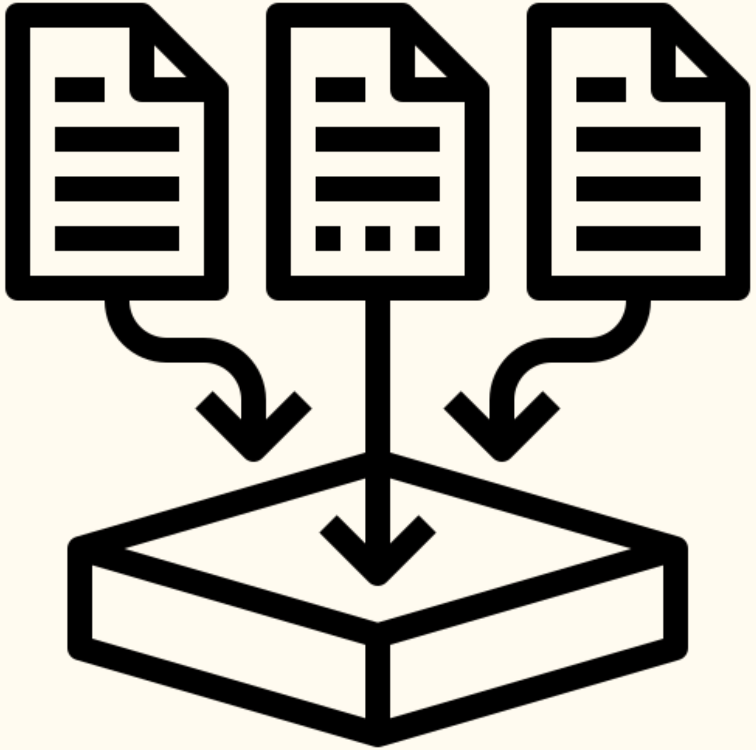
```cpp
UCLASS(CollapseCategories)
class UPlantFlowerData : public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere)
    float Radius = 0.5f;

    UPROPERTY(EditAnywhere)
    int32 Count = 5;
};

UCLASS(BlueprintType)
class UPlantDataAsset : public UDataAsset
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere)
    FText DisplayName;

    UPROPERTY(EditAnywhere)
    FSlateBrush Icon;

    // Showing how to do an inlined UObject instance for completeness
    UPROPERTY(EditAnywhere, Instanced)
    TArray<UPlantFlowerData*> FlowerDatas;

    // Point to other Data Assets
    // Instead of raw pointer could also be TObjectPtr<T> or TAssetPtr<T>
    UPROPERTY(EditAnywhere)
    TArray<UPlantDataAsset*> ChildPlants;
};
```

Data structures - Data Table Vs Data Asset

- Trying to **compare Data Table** vs **Data Asset** is to my opinion **not the way to go**
- It is like trying to compare **apples** and **oranges**
- They do not serve the **same purpose** and have **pros** and **cons** for both of them

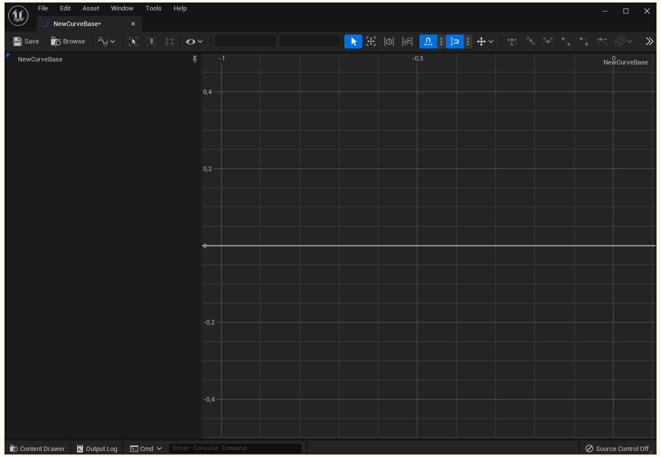
- **Data Table** will be most likely used for
  - **Collection** of data which may be changed by a **single user** (Binary file)
  - **Large amount of data**, as Data Asset can be **cumbersome** to **find** and **maintain** if you have **thousand** of them
  - Necessity to **import** from **excel**

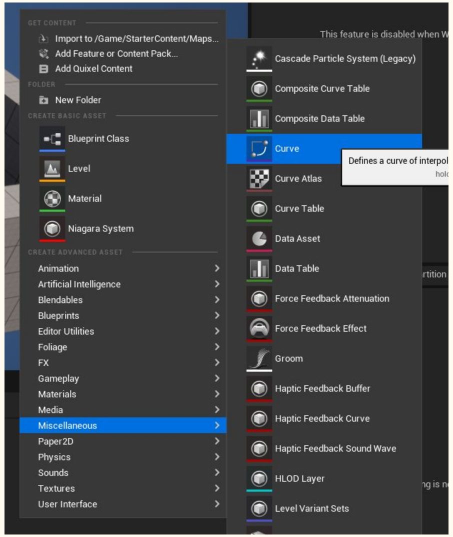
- **Data Asset** will be most likely used for
  - **Easy referencing** as a Data Asset is an **asset** representing a single struct value
  - More **flexible** and **extendable** on the long run
  - Applying more **complex validation check**, etc…
  - Can obtain **UObject** which is a big topic
  - Allow **class hierarchy** and default values



Data structures - Curves

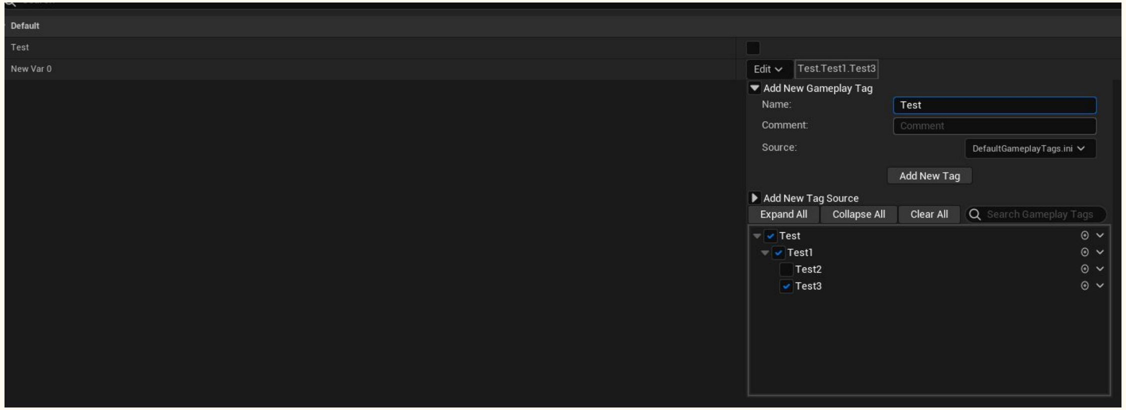
- A bit **different** than **Data Asset** and **Data Table**, **Curves** still provides a **efficiency** way to **hold** data based on **curves representation**
- As always, **Curves** are **asset** of a specific type
- By Default, a Curve can be of 3 types
  - **Float** : Most common one
  - **Vector** : It creates 3 lines on the graph, one for each dimension
  - **Color** : Like Vector but it's comes with a new looking tool to visualize the color over times
- You'll be able to find the curve editor in various places like animation curve,
- **Curves** can be **export** / **import** as **JSON** which is a great **tool** if you want to **externalise** the data from the engine
- In **code**, you have an easy way from a **Curve Asset** to retrieve, based on **axis** of the curves, the values in it
- It can be used in a various ways like
  - Damage fall-off based on distance
  - Color gradient for health bar
  - Health based on level
  - Etc..





Data structures - Gameplay Tag

- Let’s finally end with a **concept** that is in my **opinion** a really important one : **Gameplay Tag**
- **Gameplay Tag** ensure a **uniqueness** and **error-prone** way to have an **identifier** for a lot of things
- A **GameplayTag** is composed of a **hierarchical** descending tree which is mark by **dots** separating hierarchical level
- When you have a type **GameplayTag** in **Blueprint view**, you’ll simply have a dropdown on which you can select the value you want
- **GameplayTag** are associated with the **Project** and can be modified in **project settings**, or in the **.ini** file
- Used **wisely**, **Gameplay Tag** can make a **big difference** in **maintaining** a certains number of **system** based on **identifiers** and **hierarchical concept**
- **Gameplay Tag** should **not be confused** with **tags** on actor which are **different**



Test.Test1.Test3

Name: Test

Comment: Comment

Source: DefaultGameplayTags.ini

Primary vs Secondary asset

- Conceptually, **Asset Management system** in Unreal breaks all **Assets** into two **types**
    - **Primary Asset**
    - **Secondary Asset**

- **Primary Assets** can be **manipulated directly** by the Asset Manager via their **Primary Asset ID**, obtained by calling **GetPrimaryAssetId**. In order to designate **Asset** made from a specific **UObject** class as **Primary Assets**, override **GetPrimaryAssetId** to return a valid **FPrimaryAssetId** structure.
- **Secondary Assets** are **not handled** directly by the Asset Manager but **instead loaded automatically** by the **Engine** in response to being **referenced** or **used** by a **Primary Assets**
- By default, only **UWorld Assets** (Levels) are Primary. All **other assets** are **Secondary**
- A Primary Asset ID has two parts
    - A unique **Primay Asset Type** that identifies a **group** of **assets**
    - A name of that specific **Primary Asset** which default to the **Assets's name** as it appears in the **Content Drawer**.

- This explains why something, you'll find out that in your **build**, some materials are **pink** or **not rendered**. You've **tried** to load that **material** at **runtime** but it was not **reference** by any **Primary Asset** at first and it didn't get **included** by the **cooking process**.

Time to.... highlight a concept

Task management & Importance of push description

Practice

- **General**
  - Familiarize with the engine
    - Play with windows
    - Actor placement, manipulation
  - Create a Data Asset both from blueprint and C++ containing 3 fields
    - Bool
    - Float
    - Gameplay Tag
  - Create a Data Asset Instance from Blueprint Data Asset and from C++
  - Create a Data Table based on a blueprint struct containing 2 fields
    - Gameplay Tag
    - Data Asset C++ you created

- **Follow-through project**
  - Create your C++ project, starting from a top down is a good idea but feel free to pick the template you like the most
  - Create your level for the manor, containing multiple rooms, doors, etc…
  - Create materials to differentiate different elements and assign them to meshes
  - Create your various data structures in order to create as easy as possible a guard and tweaks its gameplay values
  - Create an actor that'll represent a guard