Unreal Engine 5 - Lesson 4 - Materials & VFX

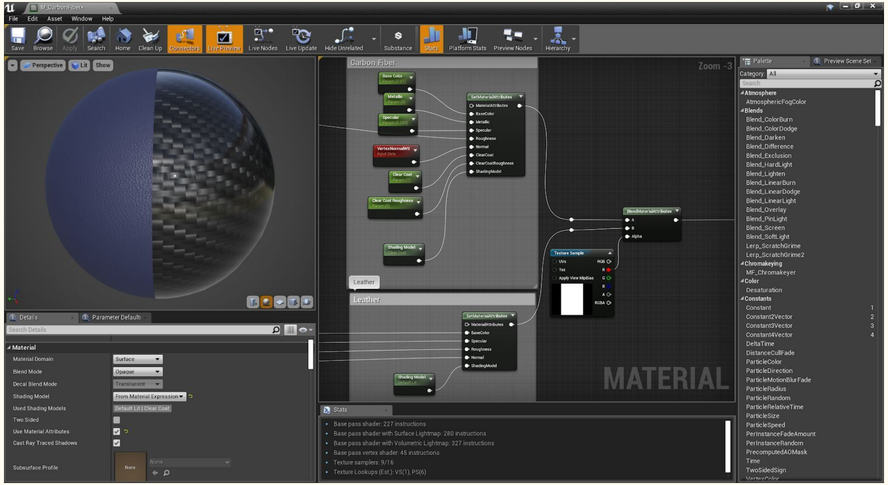
Nicolas Serf - Gameplay Programmer - Wolcen Studio
serf.nicolas@gmail.com

Summary

- Foreword
- Material
  - Material Parameter
  - Material Function
  - Physically Based Material
  - Master Material & Material Instance
  - Material Instance Dynamic
  - Custom Primitive Data
  - Layered Material
  - Material Parameter Collection
- Niagara
  - Niagara Systems
  - Niagara Emitters
  - Niagara Modules
  - Niagara Parameters
  - Spawning through BP
  - Spawning through Anim Notify
  - Modifying Parameters
  - Callback
    - Register as a callback - Actor Receiver
    - Register as a callback - Configuring NS

Foreword

- In this lesson, we’ll **dive** more in **details** about what is possible to do with **material** and **niagara system**
- To make it **clear** once again, we’ll not go through the process of **writing** actual **Material** or **Niagara System**
  - It will most likely **not be your job**, **writing complex material** or **niagara system** involves an **artist** point of view, an **knowledges** in writing good **shaders**
  - It is a full time job such as **VFX artist**, **technical artist**, etc…
  - It would require entire **lessons** in order to highlight **important nodes**, **shader** way of thinking, etc…



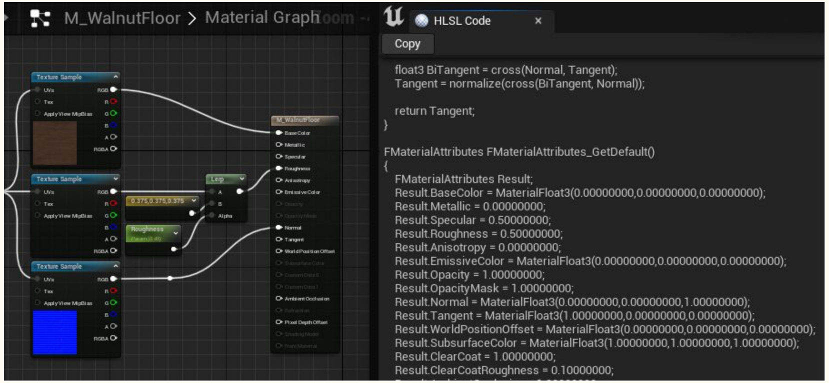
Material

Material Domain: Surface
Blend Mode: Opaque
Decal Blend Mode: Translucent
Shading Model: From Material Expression
Used Shading Models: Default Lit | Clear Coat
Two Sided: □
Use Material Attributes: ☑
Cast Ray Traced Shadows: ☑
Subsurface Profile: None

- Base pass shader: 227 instructions
- Base pass shader with Surface Lightmap: 280 instructions
- Base pass shader with Volumetric Lightmap: 327 instructions
- Base pass vertex shader: 45 instructions
- Texture samplers: 9/16
- Texture Lookups (Est.): VS(1), PS(6)

Material

- A Material is an **asset** in Unreal Engine
- It can applied to a **mesh** to control its **visual look**
- From an **high level** point of view, we can think of Material as **paint** that is **applied** to an object
- From a lower level, it is misleading as a Material **literally** defines the **type** of **surface** from which your material **appears** to be made. You defines
  - **Color**
  - How **shiny** it is
  - How **transparent** it is
  - Etc...
- In more **technical** terms, when **light** from the **scene hits** the surface, a Material is used to **calculate** how that **light interacts** with that surface. These calculations are done using **incoming data** that is input to the Material from a **variety** of images (**textures**) and **math** expressions, as well as from **various** property settings **inherent** to the **Material** itself.
- Under the hood of **Material node Editor**, the node graph are translated into **HLSL shader code**
- It is possible to see shader code from **Window > Shader Code > HLSL Code**



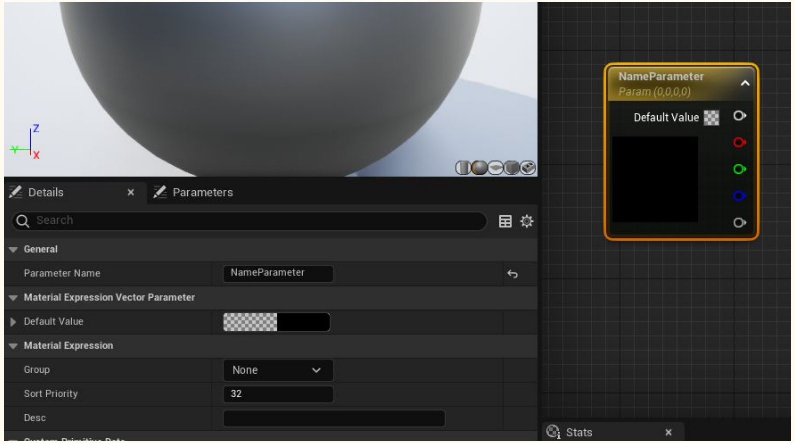
```
   float3 BiTangent = cross(Normal, Tangent);
   Tangent = normalize(cross(BiTangent, Normal));

   return Tangent;
}

FMaterialAttributes FMaterialAttributes_GetDefault()
{
   FMaterialAttributes Result;
   Result.BaseColor = MaterialFloat3(0.00000000,0.00000000,0.00000000);
   Result.Metallic = 0.00000000;
   Result.Specular = 0.50000000;
   Result.Roughness = 0.50000000;
   Result.Anisotropy = 0.00000000;
   Result.EmissiveColor = MaterialFloat3(0.00000000,0.00000000,0.00000000);
   Result.Opacity = 1.00000000;
   Result.OpacityMask = 1.00000000;
   Result.Normal = MaterialFloat3(0.00000000,0.00000000,1.00000000);
   Result.Tangent = MaterialFloat3(1.00000000,0.00000000,0.00000000);
   Result.WorldPositionOffset = MaterialFloat3(0.00000000,0.00000000,0.00000000);
   Result.SubsurfaceColor = MaterialFloat3(1.00000000,1.00000000,1.00000000);
   Result.ClearCoat = 1.00000000;
   Result.ClearCoatRoughness = 0.10000000;
```

Material Parameter

- We'll talk about **Material Parameter** because they are kind of the **main interaction** you'll have with Material if your team is big enough and you **don't need** to write material
- **Material Parameter** are **special node** that you'll define in the **node graph** and you'll give a **parameter name** to them
- There is a lot of type for **parameter**, the ones you are going to use the **most** are obviously **float** and **vector**
- You can decides how they are **ordered** and **grouped** thanks to **detail panel**
- There is 2 reasons that values are created as parameters, one implying that you communicate with technical artist that will create the material
    - **Artist using material instance to vary material application** : You'll most likely **don't need** to tell artist as they **know** that they want to use the **master material** on several **different looking meshes**
    - **Design requiring gameplay visual feedback** : This is the second case if **communicate** is not **pushed enough**. It is most likely that as **gameplay programmer**, you'll be reading the **document** and **dictating technical** aspect of a **feature**. In this situation, it may be your job to ask artist to **provide** a **set** of **parameter** in the **main material**.
- If you use a **static parameter**, it is a parameter that **can** be **modified** in a **Material Instance** but not at **runtime** with a **dynamic**



Parameter Name: NameParameter

Default Value

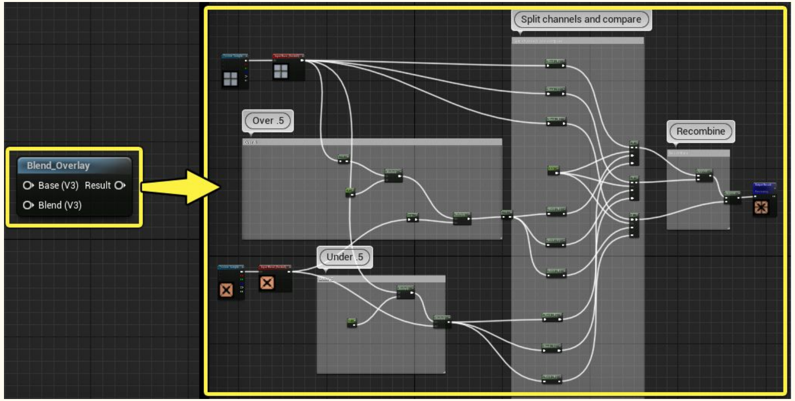
Group: None

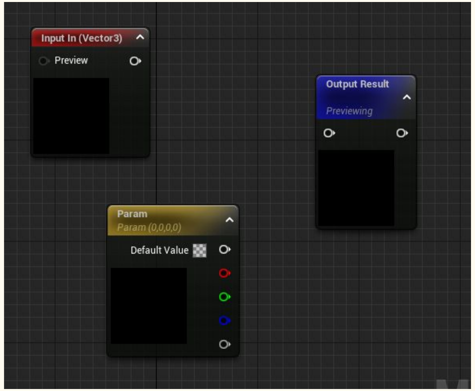
Sort Priority: 32

Desc

Material Function

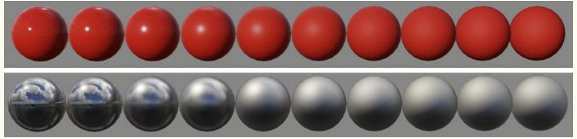
- Even if you may not write material graph, it is important to **highlight** that **material function** exists
- Like the name suggest, they allow to **encapsulate** part of a material graph into a **reusable piece** simplified in a **single node** in the calling graphs
- Unlike **main material**, material function **doesn't** have a **Main Material node** but an **output** node and **potential inputs** like traditional functions
    - You can have **multiples output nodes**
    - You can have **multiples input nodes** on which you can **parametrize** the input type
- If you want your **material function** to be **exposed**, you need to tick the option **Expose To Library** in detail panel
- Like **main material**, you can use **parameter** which will be **directly propagate** and modifiable from the **Material Instance**



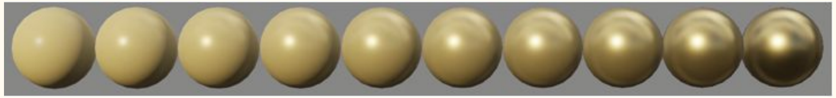


Physically Based Material

- We'll not go into details for material but let just make a **quick review** of **Physically based material (PBR)** which is what you'll most **likely** have for project **except** if you works on **creative environment** setup
- PBR try to approximate the way **light behaves** in the real world. PBR material tends to be more **accurate** and more **natural looking**
- PBR consist of 4 main output
  - **Base Color** : Vector3 (RGB) which defines the **overall color** of the Material. It is the color **from real world** which is taken by a **polarizing filter**. Polarizing filter removes the **specular** of **nonmetal** when aligned
  - **Roughness** : How **smooth** a **Material's surface** is. In the **Material** this manifests as **how sharp or blurry reflections** appear on the Material
    - A Roughness of 0 (smooth) results in a **mirror reflection.**
    - A Roughness of 1 (rough) results in a **diffuse** or **matte surface.**
    - It is most likely that artist will be using a **grayscale texture** to represent **roughness** as it is most likely **complex** on a mesh



  - **Metallic** : Defines whether your **Material** behaves as a **metal** or **nonmetal.** Metallic in most cases, is treated a **binary** property, either **0** or **1**.
    - **Black and white mask** passed into the Metallic input are **often** used in order to have more **complex metallic map**

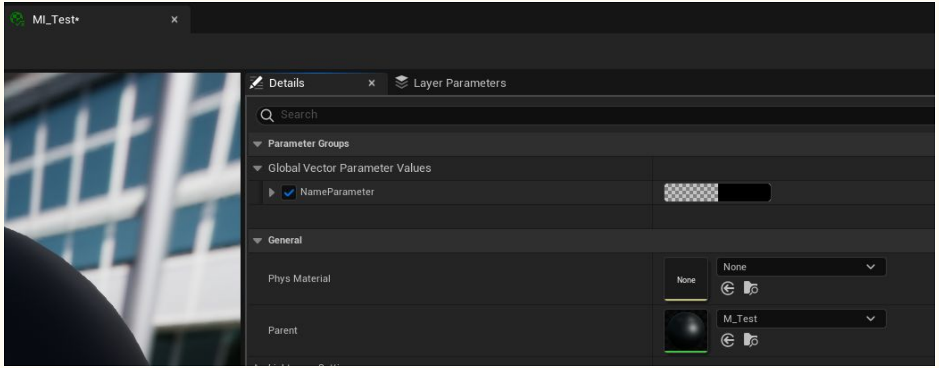


  - **Specular** : Controls how much specular light the surface reflects.
    - A Specular value of 0 is fully non-reflective.
    - A Specular value of 1 is fully reflective.
    - It is defaulted to 0.5 which is accurate for a vast majority of materials

Master Material & Material Instance

- What we talking about until now are only **master material**.
- They are commonly refer as asset named **M_MaterialName**
- A **master material** is acting as a **template** which is **not modifiable without** modifying the **master material values** and **recompiling**

- Will it is completely **possible** to **apply** a **master material** into a mesh, it is most likely that you, or artist will create a **Material Instance**
- They are commonly refer as asset named **MI_MaterialInstanceName**
- They'll present a **different interface** that the main material, mainly composed of the **parameters** you exposed in the material
- Material Instance are creating by **right clicking** on a master material and clicking **Create Material instance**.
- Just like master material, you can then **apply** a Material Instance by **drag & dropping**
- Material Instance **hide complexity**
- Material Instance **don't require to recompile** master material



☑ NameParameter

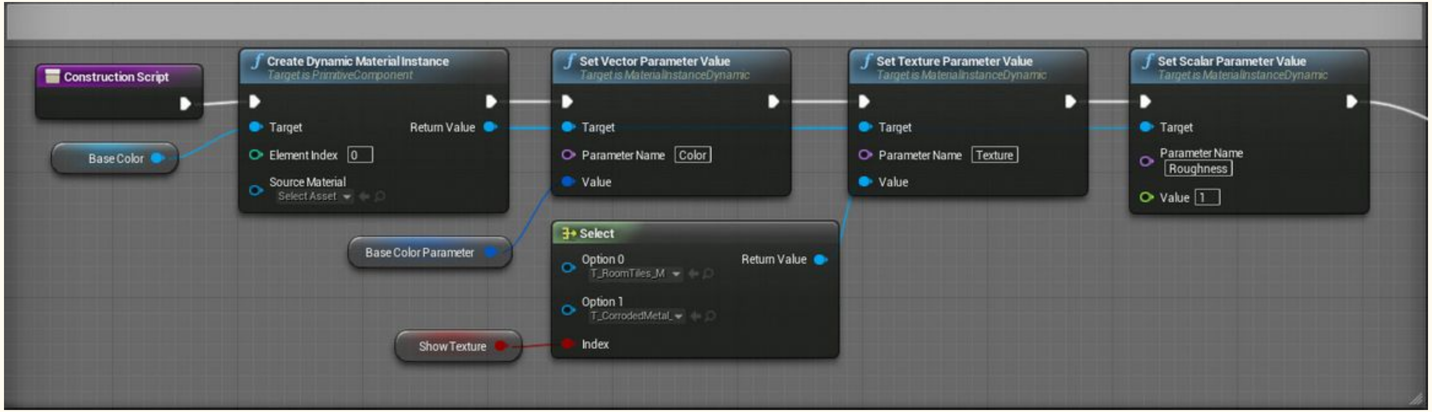
Phys Material: None

Parent: M_Test

Material Instance Dynamic

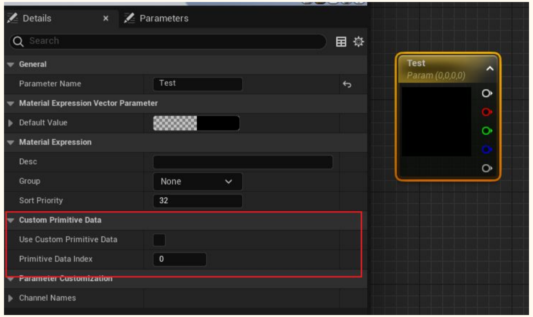
- **Material Instance** we just talked about are also known as **Material Instance Constant**
- They obviously has the **advantage** of **not require recompilation** and **less resource consumption.**

- **Material Instance Dynamic (MID)** in the other hand can be calculated during gameplay
- You'll be able to **use code**, either from **c++** or **blueprint** to change **parameter** in an **instance dynamic**
- You'll most likely use **Create Dynamic Material Instance** node in order to create a **dynamic material instance** which will create a **copy** of the **instance** on which you'll be able to **modify parameters**



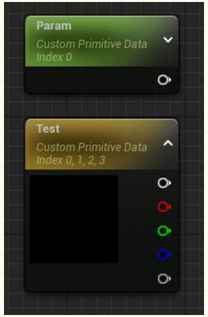
Custom Primitive Data

- Unreal Material system provides a way to **store custom data** in an **index array** which is **accessible** through **blueprint** and **c++** through the **Custom Primitive Data (CPD)**
- You can think about CPD the same way as **Material Instance Dynamic** which provides a way to control some aspect of material thanks to **material parameters.**
  - The **difference** being that CPD **stores data** on the **primitive** itself rather than the **material instance**
  - It **lowers** the number of **draw calls** for **similar** placed **geometry** in levels
- In order to configure it, you must go in the **material**, click on the **parameter node** and you'll have an option to set this parameter to **Use Custom Primitive Data.**
  - You must define a **unique index** for each data you want to use
  - It is important to understand that when **setting** a **scalar**, the **index** is directly **written**
  - When you define a **vector**, there is **4 scalar representing R, G, B, A**, which mean that if you decide index of **0**, it will be using **0, 1, 2, 3** for each **RGBA output**



General
Parameter Name: Test
Material Expression Vector Parameter
Default Value
Material Expression
Desc
Group: None
Sort Priority: 32
Custom Primitive Data
Use Custom Primitive Data: □
Primitive Data Index: 0
Parameter Customization
Channel Names

Test
Param (0,0,0,0)

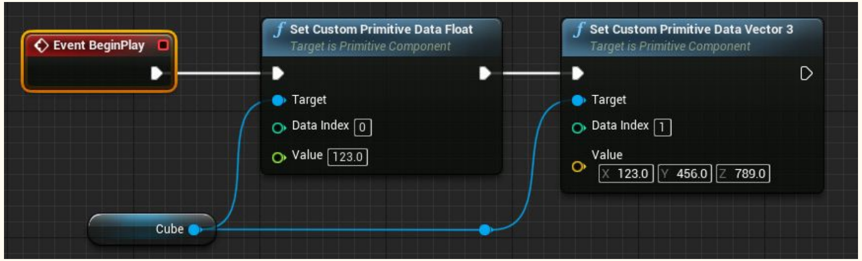

Param
Custom Primitive Data
Index 0

Test
Custom Primitive Data
Index 0, 1, 2, 3

Custom Primitive Data - Setting parameter

- In order to access scene primitives that use **Custom Primitive Data** through **Blueprint** you can use the following nodes
  - **Set Custom Primitive Data Float**
  - **Set Custom Primitive Data Vector**
- Unlike the **traditional** way of setting parameter through **name**, you'll use **index** this time and the value that you want to set

- The **CPD workflow** has the advantage of significantly **reducing draw** calls for similar geometry in your Level when it is using a material set up with custom data. Draw calls are reduced using the **Mesh Drawing** refactor that automatically dynamically instances scene primitives.
- To check how well **dynamic instancing is working** for your Level, open the console (`) and enter the command '***stat scenerendering***'. This command shows general rendering statistics for your current scene view. It is a good starting point to **finding** general areas of **slow performance** in the **rendering** process, along with **counters** for the number of **mesh draw calls** and **lights** in the scene.

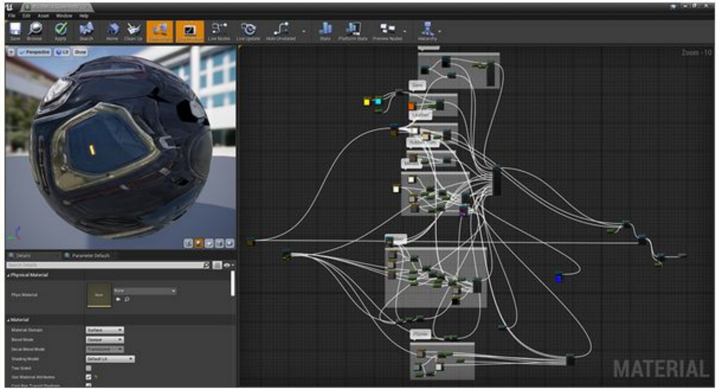

Event BeginPlay
Set Custom Primitive Data Float
Target is Primitive Component
Target
Set Custom Primitive Data Vector 3
Target is Primitive Component
Target
Cube

Set Custom Primitive Data Float: Data Index 0, Value 123.0

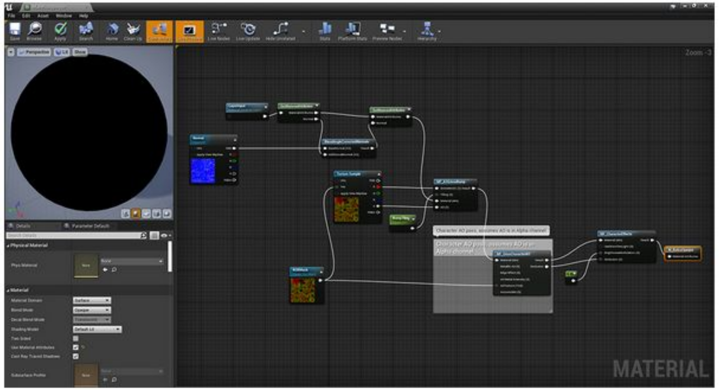
Set Custom Primitive Data Vector 3: Data Index 1, Value X 123.0 Y 456.0 Z 789.0

Layered Materials

- We’ll **not** go into **details** about **Layered Materials** as it is more an **artist production issue** but you need to be **aware** that it exists
- In older version of the engine, and even some production still use the **Layered Materials** with **Material Functions.**
- However, you can now use **Material Layer asset** type in your material graph to **combine** and **blend textures**
- This alternative workflow **simplifies** the **process** of **layered** and **blended materials** by taking advantage of **Material Instancing**
- **Performance-wise**, it is important to understand that **each layer** in a Layered Material is a separate **draw call**
- Even if Layered Material with Material Function are possible, they still are a mess to maintains
    - There is **multiple reason** for that but the main one being that **any parameters** created in a **Material Function cannot** be **referenced** in an **instanced material** without first being **duplicated** in the **base Material**
    - In contrast, you can **instance any parameters** set up in a Material Layer Asset



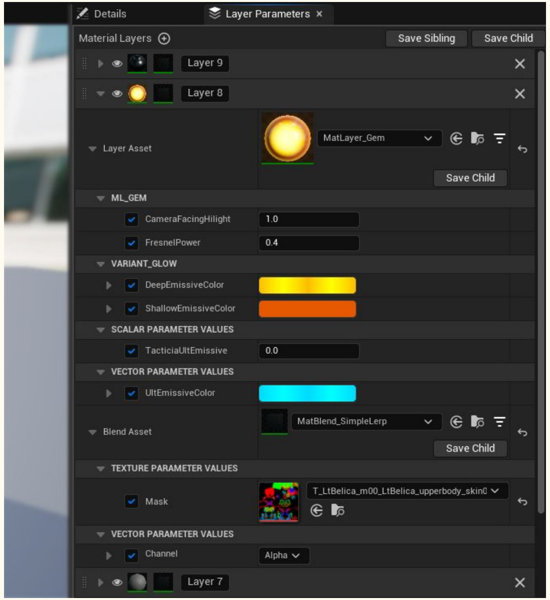
Base Material using Material Function workflow



Base Material using Material Layers workflow

Layered Materials

- If you want **more detail** about how to create a complete **Layered Material**, you can check unreal documentation
- We'll simply **highlight** that there is **multiple layers** as you can see on the image
- A Layer has a **name**, and is **composed** of a **Material Layer** which itself contains **parameters**
- Our job as programmer will be to **allow designers** to modify this parameters
- In order to do so, we'll **not use the standard method** to modify parameter like **SetVectorParameterValue**
- There is a blueprint node and the corresponding c++ which is **SetVectorParameterValueByInfo**
  - Info is a struct which allow to **configure** the parameter you want to set
  - **Name** : **Name** of the **parameter**
  - **Association** : It is a **global parameter**, a **blend layer parameter** or a **layer parameter**
  - **Index** : **-1** if **global parameter**, or the **index** on which the blend **layer** / **layer** is located



Layer 9

Layer 8

Layer Asset: MatLayer_Gem

ML_GEM

☑ CameraFacingHilight 1.0

☑ FresnelPower 0.4

VARIANT_GLOW

☑ DeepEmissiveColor 🟨

☑ ShallowEmissiveColor 🟧

SCALAR PARAMETER VALUES

☑ TacticiaUltEmissive 0.0

VECTOR PARAMETER VALUES

☑ UltEmissiveColor 🟦

Blend Asset: MatBlend_SimpleLerp

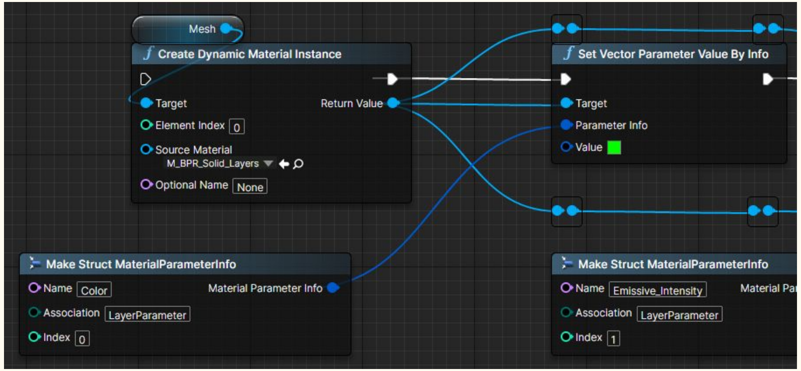
TEXTURE PARAMETER VALUES

☑ Mask T_LtBelica_m00_LtBelica_upperbody_skin0

VECTOR PARAMETER VALUES

☑ Channel Alpha

Layer 7


Mesh
Create Dynamic Material Instance
Target
Return Value
Set Vector Parameter Value By Info
Target
Parameter Info
Make Struct MaterialParameterInfo
Material Parameter Info
Make Struct MaterialParameterInfo
Material Pa

Element Index 0

Source Material M_BPR_Solid_Layers

Optional Name None

Value 🟩

Name Color

Association LayerParameter

Index 0

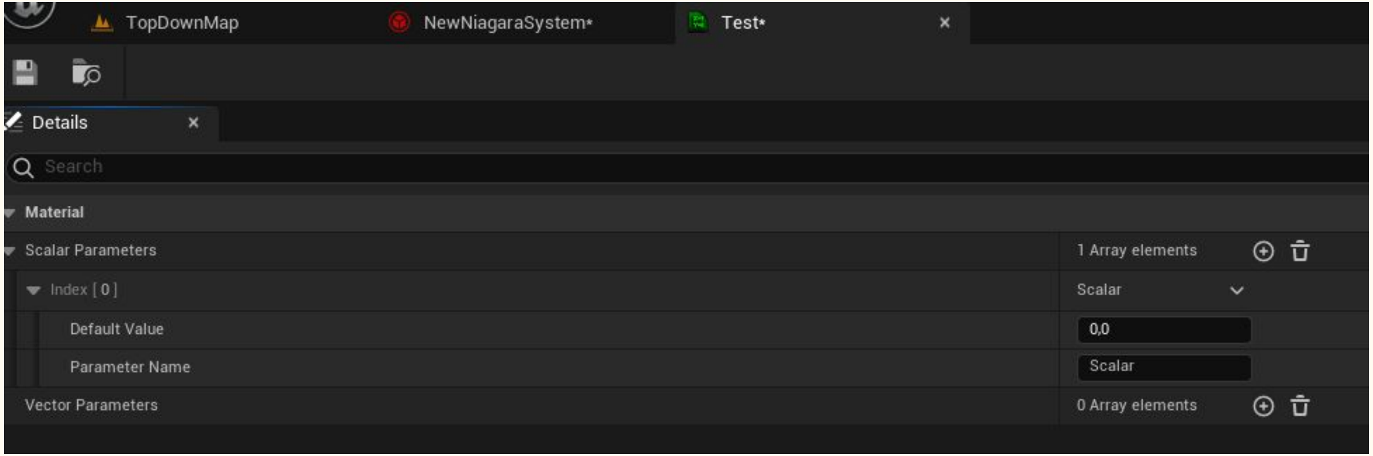
Name Emissive_Intensity

Association LayerParameter

Index 1

Material Parameter Collection

- A **Material Parameter Collection** is an **asset** that you can **create** from the **content drawer.**
- It act has a **container** for various **parameter** that can be used in **multiple materials**
- You will then be able to **modify** that **centralized parameter** from **code** and have the modification **affecting** every **material** that is **using** that variable in the **collection**
- Again, **performance-wise** it is **better** because we are **not forced** to create a **material instance**, as we are **modifying** a **parameter** which is directly affecting the **main material**
- In the **material graph**, you can just get a **reference** to a **parameter collection**, given you a **detail panel** where you select the **collection** you want to use and the **parameter name**



Material

Scalar Parameters — 1 Array elements

Index [0] — Scalar

Default Value — 0,0

Parameter Name — Scalar

Vector Parameters — 0 Array elements

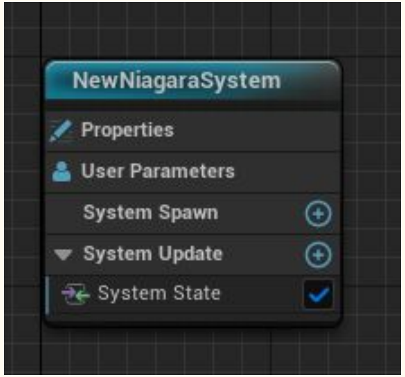
Niagara

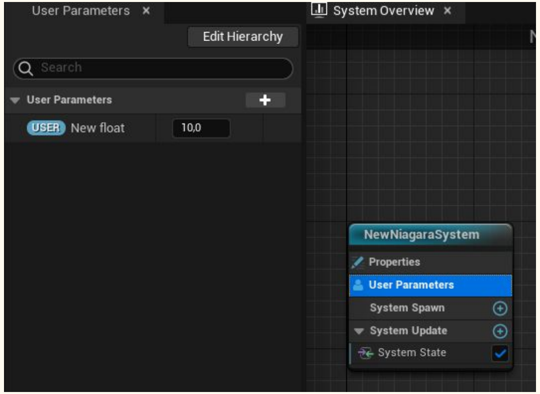
- Niagara is **Unreal Engine's** next-generation **VFX system**. With **Niagara**, the technical artist has the ability to create **additional functionality** on their own, without the assistance of a **programmer**. The system is **adaptable** and **flexible**.
- Just like **Material**, we'll **not go through into detail** on how to create a niagara system because it is not o**ur job as programmer** to create **coherent** and **beautiful visual** for **VFX**
- However, it is important to **understand** how it works **generally** speaking, and how you'll be able to **interact** with a **Niagara System** from the outside
- Keep in mind that you may **actually** need to **develop** some **complex system** which may involve some Niagara system

- A Niagara System is composed of four core components:
  - Systems
  - Emitters
  - Modules
  - Parameters

Niagara - Systems

- A **Niagara system** is a **container** for everything you will need to build that **effect.** Inside that system, you may have **different building blocks** that stack up to help you produce the **overall effect.**
- You can modify some **system-level behavior**s that will then apply to everything in that **effect.**
- In this node, there is one **particular section** that you should be aware of : **User Parameters**
- Do **not worry** too much about **how** this **variable** will be used later on by **VFX artist**. Actually, it is even **artist** that will most **likely create** the **parameter** based on **GDD** or on what **you expect** them to **provide** as entry point
- Just to **highlight** how this **parameter** can be used, let’s switch to a **direct demonstration** in the engine

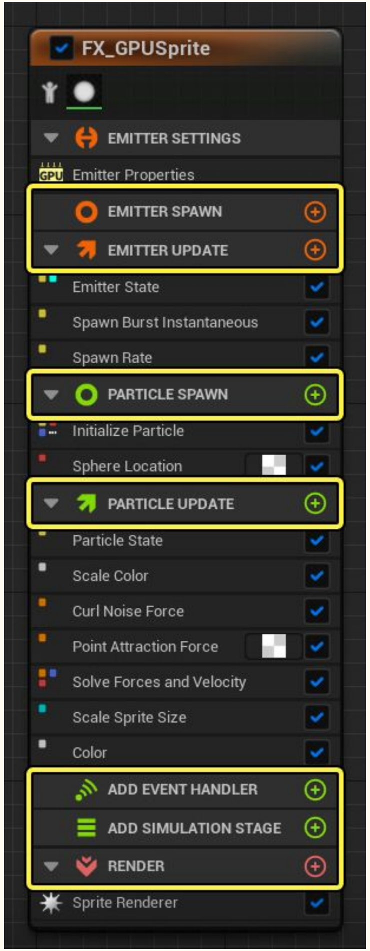




USER New float 10,0

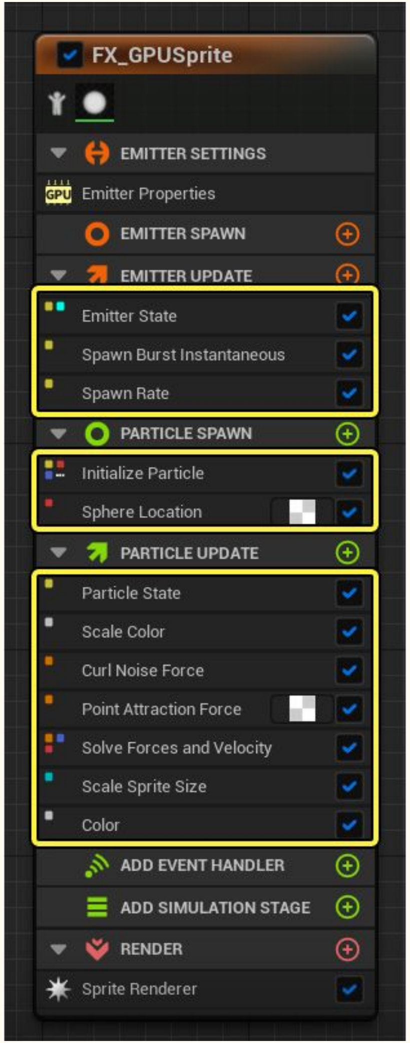
Niagara - Emitters

- Emitters are where **particles** are **generated** in a **Niagara system**. An emitter controls **how particles are born**, what **happens** to that **particles** as they **age**, and **how the particles look and behave.**
- The emitter is organized in a **stack**. Inside that stack is **several groups**, inside which you can put **modules** that accomplish **individual tasks**. The groups are as follows in the **image**
- As you can see, there is multiples section in an emitter
  - **Emitter Spawn** : Defines what **happens** when an emitter is **first created** on the CPU. Use this group to **define initial setups** and defaults.
  - **Emitter Update** : Defines **emitter-level modules** that occur **every frame** on the CPU. Use this group to **define spawning** of **particles** when you want them to **continue spawning** on every frame.
  - **Particle Spawn** : Called **once per particle**, when that particle is **first born**. This is where you will want to **define** the **initialization details** of the particles, such as the **location** where they are **born**, what **color** they are, their **size**, and more.
  - **Particle Update** : Called **per particle** on **each frame**. You will want to define here anything that needs to **change frame-by-frame** as the particles age.
  - **Event Handler** : In the **Event Handler** group, you can create **Generate events** in **one** or **more emitters** that define certain data. Then you can **create Listening** events in other emitters which trigger a **behavior** in **reaction** to that generated event.
  - **Render** : This is where you **define** the **display** of the **particle** and set up **one** or **more renderers** for your particles. You may want to use a **Mesh renderer** if you want to define a **3D** model as the basis of your particles, upon which you could **apply** a **material**. Or, you may want to use a **sprite renderer** and define your particles as **2D sprites**. There are many different renderers to choose from and experiment with.
- You'll most likely be interested in the **Event Handler** section, but we'll come back to it later on



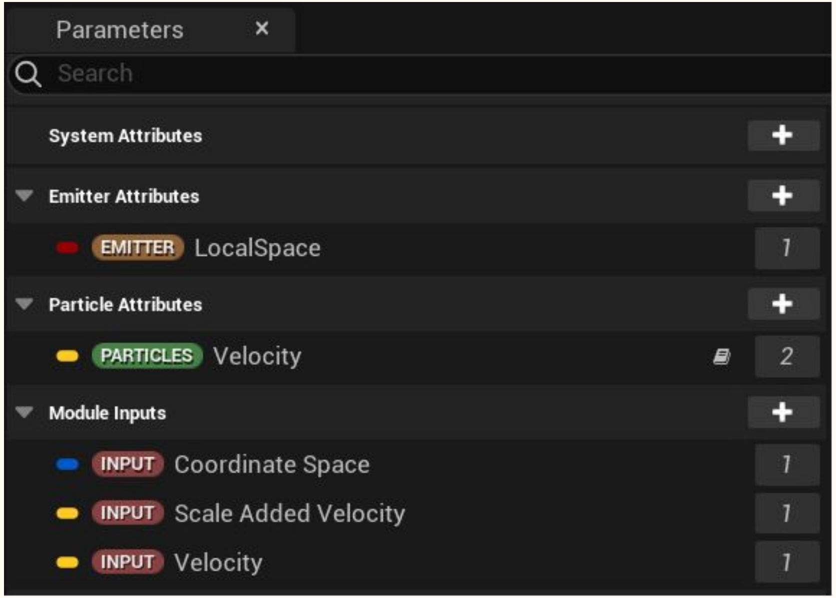
Niagara - Modules

- Modules are the **basic building blocks** of effects in **Niagara**. You add **modules** to groups to make a **stack**. Modules are **processed sequentially** from **top** to **bottom**.
- You can think of a module as a **container** for doing some **math**. You pass some **data** into the **module**, then inside the module you do some **math** on that data, and then you write that data back out at the **end** of the **module**.
- Modules are built using **High-Level Shading Language (HLSL)**, but can be built visually in a **Graph** using **nodes**. You can **create functions**, **include inputs**, or write to a **value or parameter map**. You can even write **HLSL code inline**, using the CustomHLSL node in the Graph.
- You can **double-click** any module from an emitter in Niagara to take a **look** at the **math** that's happening inside. You can even copy and create your own modules.
- The script starts by **retrieving inputs** - the **velocity input** and the coordinate space. It then gets the **current velocity** of the particles, as well as an **inputted** scaling factor. Then, the input velocity is **scaled**, transformed in the correct **coordinate space**, and added to the current velocity of the particles. Once that work is complete, the new particle velocity is written back out so that any modules that need velocity information further on down the stack can retrieve it.
- All **modules** are built with that **basic methodology**, though for some the **internal math** may be more **complex**.



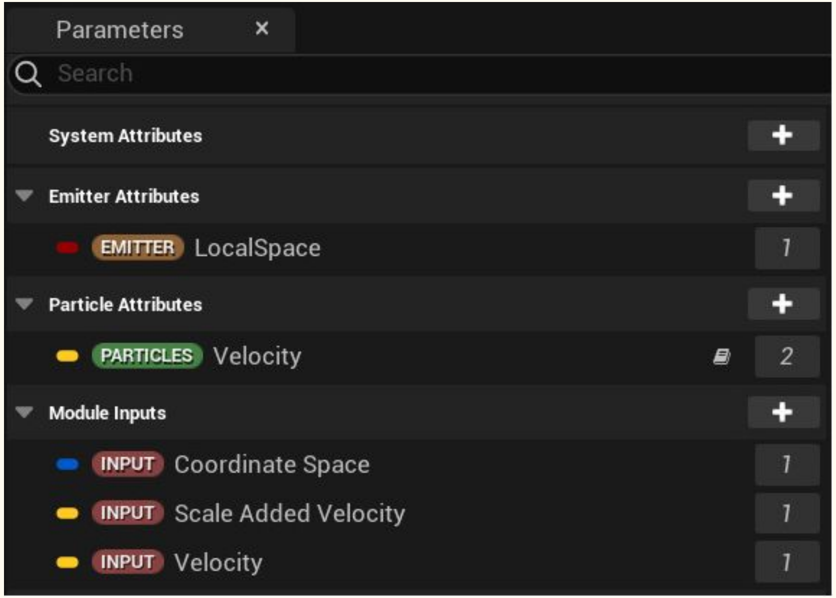
Niagara - Parameters

- Parameters are an **abstraction** of data in a **Niagara simulation**. **Parameter types** are assigned to a **parameter** to define the data that parameter represents. There are **four** types of parameters:
    - **Primitive**: This type of parameter defines **numeric data** of varying precision and channel widths.
    - **Enum**: This type of parameter defines a **fixed set** of **named values**, and assumes one of the named values.
    - **Struct**: This type of parameter defines a **combined** set of **Primitive** and **Enum types**.
    - **Data Interfaces**: This type of parameter defines functions that provide data from **external data sources**. This can be data from other parts of UE5, or data from an outside application.
- You can add a custom parameter module to an emitter by clicking the Plus sign icon (+) and selecting Set new or existing parameter directly. This adds a Set Parameter module to the stack. Click the Plus sign icon (+) on the Set Parameter module and select Add Parameter to set an existing parameter, or Create New Parameter to set a new parameter.
- Live demonstration about **parameters**



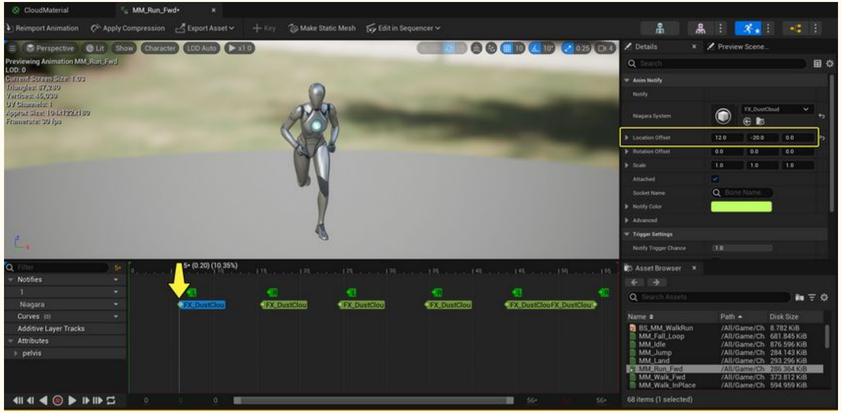
Niagara - Spawning through BP

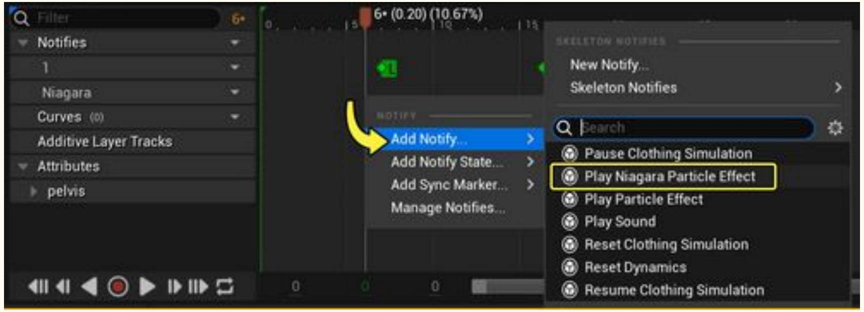
- Parameters are an **abstraction** of data in a **Niagara simulation**. **Parameter types** are assigned to a **parameter** to define the data that parameter represents. There are **four** types of parameters:
  - **Primitive**: This type of parameter defines **numeric data** of varying precision and channel widths.
  - **Enum**: This type of parameter defines a **fixed set** of **named values**, and assumes one of the named values.
  - **Struct**: This type of parameter defines a **combined** set of **Primitive** and **Enum types**.
  - **Data Interfaces**: This type of parameter defines functions that provide data from **external data sources**. This can be data from other parts of UE5, or data from an outside application.
- You can add a custom parameter module to an emitter by clicking the Plus sign icon (+) and selecting Set new or existing parameter directly. This adds a Set Parameter module to the stack. Click the Plus sign icon (+) on the Set Parameter module and select Add Parameter to set an existing parameter, or Create New Parameter to set a new parameter.
- Live demonstration about **parameters**



Niagara - Spawning through Anim Notify

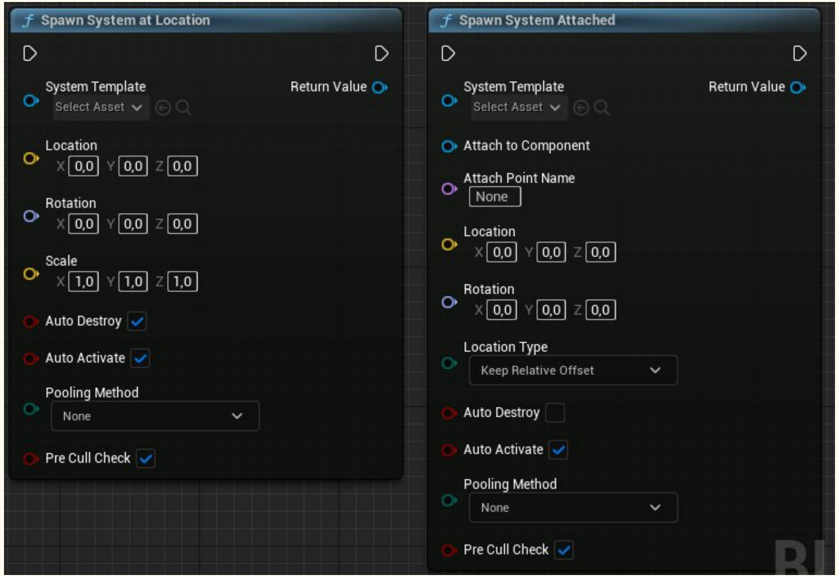
- This is one of the **most** common way to **instantiate** a **niagara system** as you'll often want to get a **visual impact** on some **animation** like a **sword swinging** or a spell casting
- In order to do so, you'll need to open an animation asset and on the timeline, create a **Notify named Play Niagara Particle Effect**
- By selecting the **notify**, you'll have the possibility in the details panel to configure it. Consider that attachment and offsetting position is related to the **pawn** that the **animation** is **playing**





Niagara - Spawning through Blueprint

- Regarding Blueprint, there is two main ways to spawn a Niagara system
- At **runtime** with
  - **Spawn System at Location** : It is the method if you know that your **Niagara system** will only be a **child** of the **pawn**, and you know the **relative transform** that you want to **apply**
  - **Spawn System Attached** : It is the method if you know a **socket** where you want your **VFX system** to be **attached** and **spawned**
- It will give you a reference to the niagara component it spawns, and you'll most likely save that in a variable, register some callback about it etc...
- In **editor** by adding a **Niagara Component**
  - You'll be able to **easily** place the **system** and have a **direct visual feedback** on where it will **appears** at **runtime**
  - You'll be able to **register** the various **callback** directly in the node graph



Spawn System at Location

System Template: Select Asset
Return Value
Location: X 0,0 Y 0,0 Z 0,0
Rotation: X 0,0 Y 0,0 Z 0,0
Scale: X 1,0 Y 1,0 Z 1,0
Auto Destroy ☑
Auto Activate ☑
Pooling Method: None
Pre Cull Check ☑

Spawn System Attached

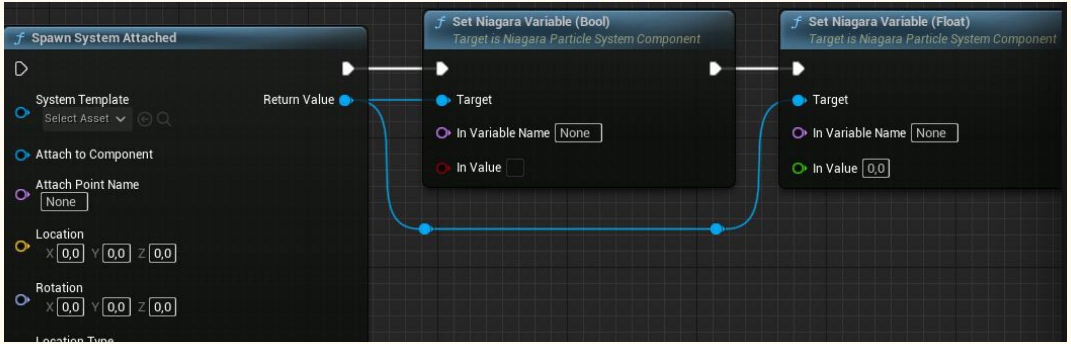
System Template: Select Asset
Return Value
Attach to Component
Attach Point Name: None
Location: X 0,0 Y 0,0 Z 0,0
Rotation: X 0,0 Y 0,0 Z 0,0
Location Type: Keep Relative Offset
Auto Destroy □
Auto Activate ☑
Pooling Method: None
Pre Cull Check ☑

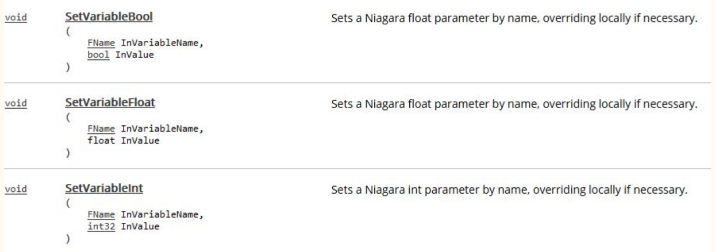
Niagara - Spawning through C++

- As you'll be able to see, **Niagara** is located in a specific **module**, which mean that you need to **add** it to **Module** like we did for **Gameplay Tags**
- You are then able to spawn a Niagara System thanks to this helper function
- As you can see, this function is creating a **NiagaraComponent** which should **not** be **mislead** with a **Niagara System** which is the asset that you are creating thanks to **content drawer**
- As you can see, the function is quite **cumbersome** to write as it **requires location** etc… and n**ot really user friendly** in c++
    - That's why it is **quite common** to have this **component added** from the **blueprint** editor, and **tweak** it directly thanks to the **viewport**

Niagara - Modifying parameters

- Just like material, you’ll have a set of **function** that you can call in order to modify the parameter
  - **SetNiagaraVariable(Bool)**
  - **SetNiagaraVariable(Float)**
  - Etc…
- It requires the **Niagara system component** as reference, and you’ll specify the **parameter name** and the **parameter value**
- You’ll have the **exact same calls in C++**

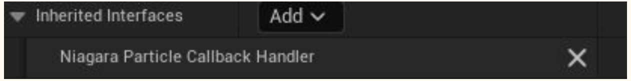




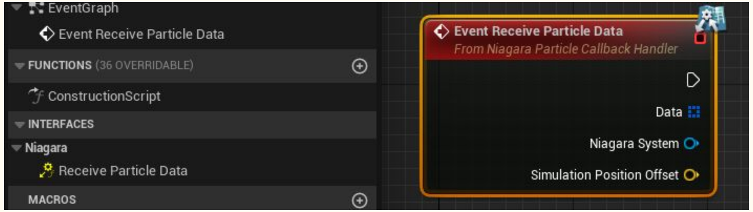
void	SetVariableBool (FName InVariableName, bool InValue)	Sets a Niagara float parameter by name, overriding locally if necessary.
void	SetVariableFloat (FName InVariableName, float InValue)	Sets a Niagara float parameter by name, overriding locally if necessary.
void	SetVariableInt (FName InVariableName, int32 InValue)	Sets a Niagara int parameter by name, overriding locally if necessary.

Niagara - Register as a callback - Actor receiver

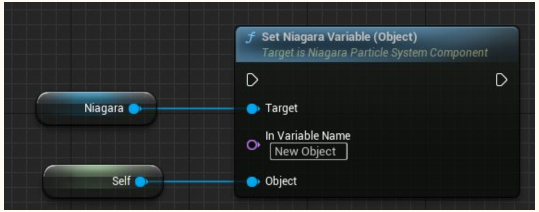
- Finally, we’ll see to **register** a **callback** in blueprint, the obvious first step is to have the **needs** for it. A good example could be that you want to **spawn** a **decal** at the **exact position** of a particle **colliding** with the **floor**
- The very first step in order to do so is to have the **actor** that you want to listen event to have **implementing** a **particular interface** called **Niagara Particle Callback Handler**



- It will gives you a need method to implement in your blueprint which will have all the data you’ll need from it

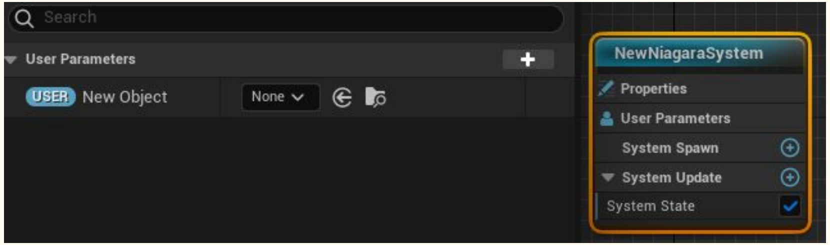

EventGraph
Event Receive Particle Data
FUNCTIONS (36 OVERRIDABLE)
ConstructionScript
INTERFACES
Niagara
Receive Particle Data
MACROS
Event Receive Particle Data
From Niagara Particle Callback Handler
Data
Niagara System
Simulation Position Offset

- At BeginPlay, or any place it makes sense, set the parameter object of the niagara system to the actor, so our actor will be registered as the callback object on which event needs to be sent

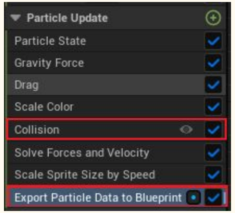

Set Niagara Variable (Object)
Target is Niagara Particle System Component
Niagara
Target
In Variable Name
New Object
Self
Object

Niagara - Register as a callback - Configuring NS

- You must go into your **niagara system** this time, we'll be **setting** a **particular User Parameter** which will be the **actor** which will be **listening** for **event** produce by the **NS**
- Create a **new parameter** with the type **Object** which will allow to **connect** any type of **UObject** which needs to listen for this events.
    - Take care about the **name** you are giving to that **parameter**, because just like any parameter, you'll be setting it through the blueprint and the name obviously needs to be the same

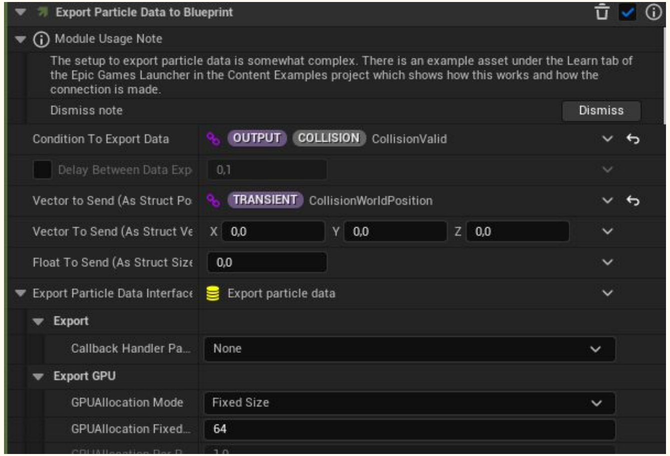


- For the following, we'll consider **sending** an **event collision**, but keep in mind that this **setup** should be **working** for **any type of event** as long as it **makes sense** regarding **the niagara system setup**
- In the emitter, you'll add a module for enabling collision, and another one to export particle data to blueprint


Particle Update
Particle State
Gravity Force
Drag
Scale Color
Collision
Solve Forces and Velocity
Scale Sprite Size by Speed
Export Particle Data to Blueprint

Niagara - Register as a callback - Configuring NS

- Most work to be done is now located on the module Export
- **Condition to Export Data** : You can type **CollisionValid**, and you'll see a corresponding entry which you can select
- **Export GPU system** are like **optimization** and artist may need to **bump** or **reduce** the value if you are **receiving** to **much event** or **not enough based** on the **number** of **particle** it spawns
- You can see that there is then as struct which will be exported to blueprint, if you check the blueprint event callback function, you'll see it contains an array of data, and the type of this data is this struct. You'll see that names are as following
    - **Position :** (1st variable)
    - **Size :** An float (3rd variable)
    - **Velocity** (2nd variable)
- This are **generic name**, but you can obviously **send any type** of value you want
- In the **Export Data Interface section**, you need to setup the **callback handler parameter** to the **parameter** you created **earlier**



Export Particle Data to Blueprint

Module Usage Note

The setup to export particle data is somewhat complex. There is an example asset under the Learn tab of the Epic Games Launcher in the Content Examples project which shows how this works and how the connection is made.

Dismiss note — Dismiss

Condition To Export Data: OUTPUT COLLISION CollisionValid

Delay Between Data Exp: 0,1

Vector to Send (As Struct Po: TRANSIENT CollisionWorldPosition

Vector To Send (As Struct Ve: X 0,0 Y 0,0 Z 0,0

Float To Send (As Struct Size: 0,0

Export Particle Data Interface: Export particle data

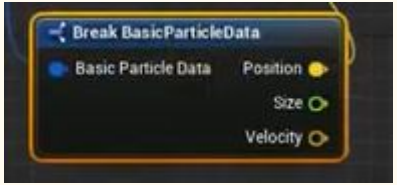
Export

Callback Handler Pa...: None

Export GPU

GPUAllocation Mode: Fixed Size

GPUAllocation Fixed...: 64



Time to.... highlight a concept

Gameplay Ability System (GAS)

Pratice

- **General**
  - Create a material and try the various ways to modify it either from blueprint or c++
    - Through traditional MID
    - Through Custom Primitive Data
    - Through Parameter Collection
  - Create a material layer, and 2 material layer, 1 material layer having a parameter to modify
    - Find a way to modify that parameter
  - Create a Niagara System and have a parameter in it which you modify thanks to blueprint
    - Spawn that Niagara System through an Anim Montage
    - Spawn that Niagara System through Blueprint
- **Follow-through project**
  - On the Pawn for AI, create a new mesh which will be acting as an indicator on the state of the guard with a material on it
    - Green = No alert, Orange = Searching for player which is not visible, Red = Chasing a visible player
  - Add a mesh component on the Pawn for AI which will be representing the Manor coat of arms. Meaning that you need to configure it at runtime. Think larger than the pawn… like having in the future some banner which will be using that value
    - Create the mesh
    - Create the material associated and find a way to have it easily modifiable at runtime
  - Create materials for your manor representing the various elements which you can find in the environment, you can think mainly of 3 materials. Try to make them thanks to PBR technics
    - Floors
    - Walls
    - Gold Ingot
  - Create a Niagara system using an existing emitter, and create a parameter which allow to modify how many particle it spawns
  - This Niagara system will be trigger later on when stepping onto a gold ingot, for now, trigger it at Begin Play of your Player Character