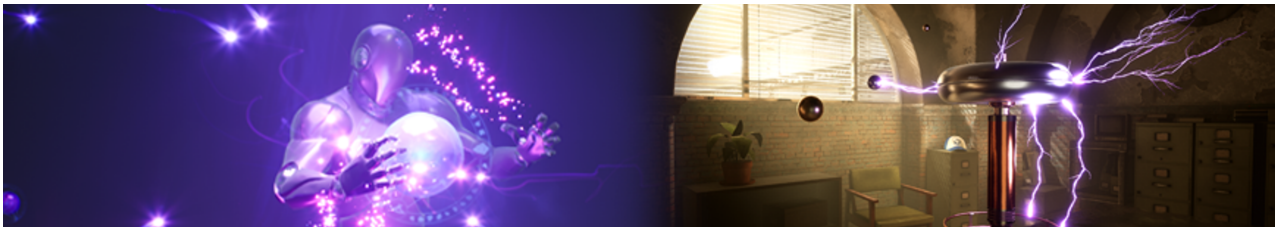
Niagara Overview

dev.epicgames.com/documentation/en-us/unreal-engine/niagara-overview



Both Cascade and Niagara can be used to make visual effects (VFX) inside of Unreal Engine (UE4), but the way you use Niagara to create and adjust VFX is very different from how you use Cascade.

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. We have made the system as adaptable and flexible as possible, while making it easy to use and understand.

Core Niagara Components

In the Niagara VFX system, there are four core components:

- Systems
- Emitters
- Modules
- Parameters

Systems

Niagara systems are containers for multiple emitters, all combined into one effect. For example if you are making a firework effect, you might want multiple bursts in your firework. To do so you would create multiple emitters, and place them all into a Niagara system called Firework. In the Niagara System Editor, you can modify or overwrite anything in the emitters or modules that are in the system. The **Timeline** panel in the System Editor shows which emitters are contained in the system, and can be used to manage those emitters. The Emitter Editor and System Editor are mostly the same. For more information on the System Editor interface, see the Niagara System and Emitter Editor Reference. UI elements that act differently in the System Editor are indicated in each section.

Emitters

Niagara emitters are containers for modules. They are single purpose, but they are also re-usable. One unique thing about Niagara emitters is that you can create a simulation using the module stack, and then render that simulation multiple ways in the same

emitter. Continuing our firework effect example, you could create one emitter that had a sprite renderer for the for the spark, and a ribbon renderer for the stream of light following the spark. For more information on the Niagara Emitter Editor interface, see the Niagara System and Emitter Editor Reference.

Modules

Niagara modules are the base level of Niagara VFX. Modules are the equivalent of Cascade's behaviors. Modules speak to common data, encapsulate behaviors, stack with other modules, and write functions. 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. For more information on the default modules available in a Niagara emitter, see the Niagara System and Emitter Module Reference.

Parameters and Parameter Types

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 UE4, 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. For more information on these and other emitter modules, see the Niagara System and Emitter Module Reference.

Niagara Stack Model and Stack Groups

Particle simulation in Niagara operates as a *stack*—simulation flows from the top of the stack to the bottom, executing programmable code blocks called *modules* in order. Crucially, every module is assigned to a *group* that describes when the module is executed. For more information, see the Niagara Key Concepts page and the Niagara System and Emitter Module Reference.

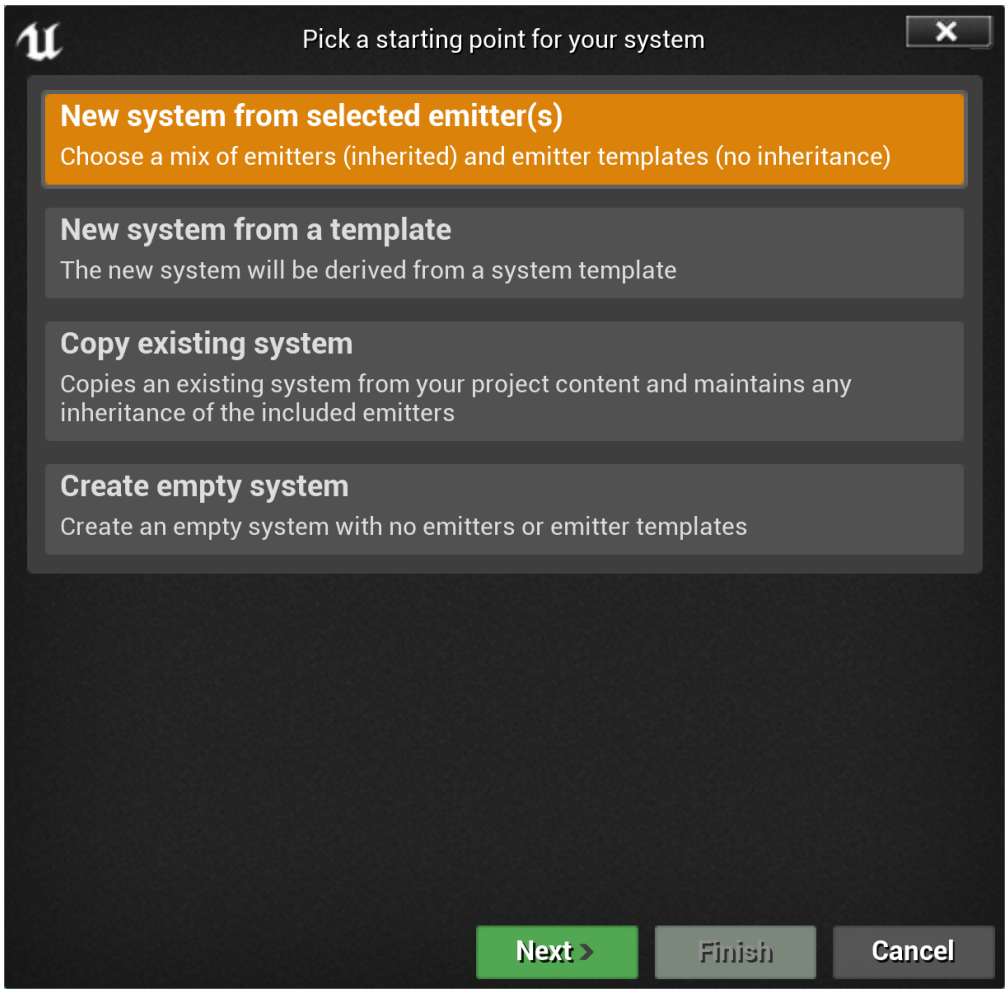
Templates and Wizards

When you first create a Niagara emitter or Niagara system, a dialog displays offering several options for what kind of emitter or system you want to create. One of these options is to choose a template. These templates are based on some common base effects, and include various modules already. You can change any of the parameters in the template. You can add, modify or delete any of the modules. In a system template, you can also add, modify or delete any of the emitters. Templates are just there to jumpstart your creativity and give you something that you can work with immediately. for more information on the Niagara Editor UI, see the Niagara Editor UI Reference.

System Wizard

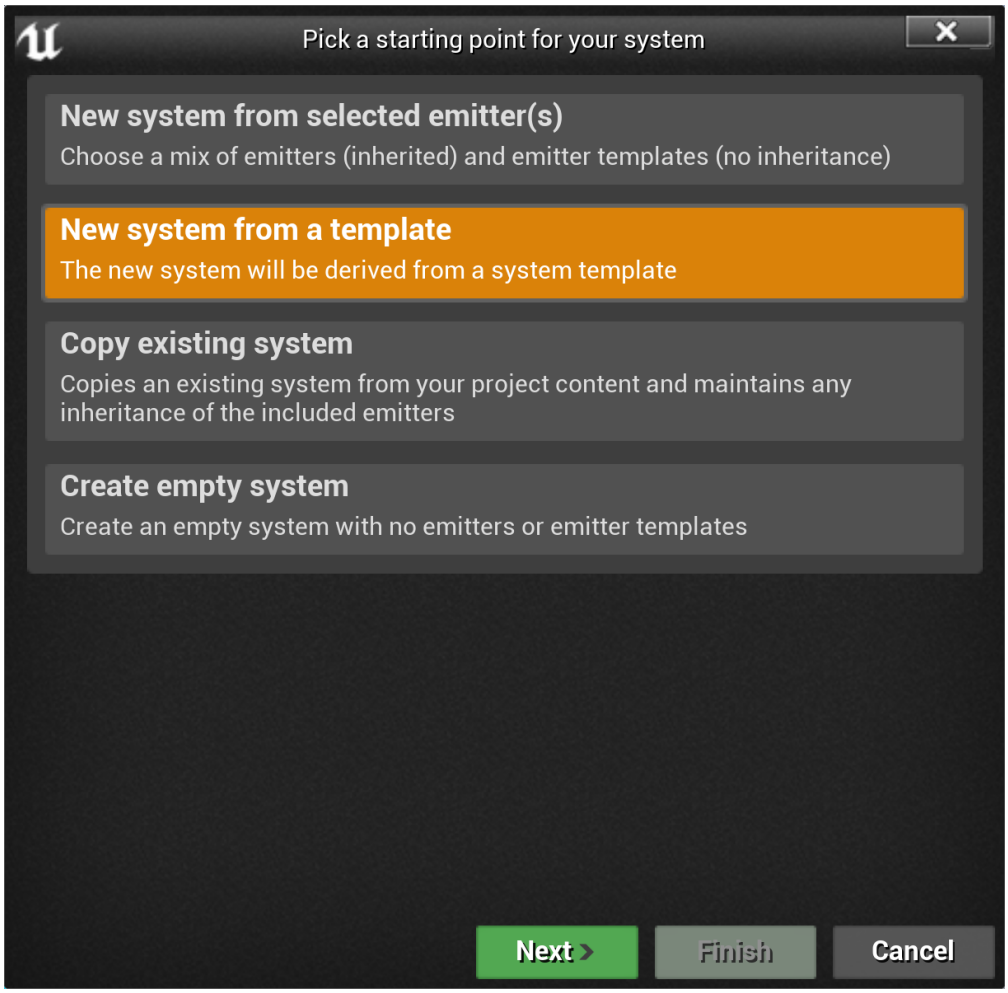
The System Wizard offers the following options for creating a new system:

New system from selected emitters: If you select this option and click **Next**, a list of available emitters displays. This list includes both existing emitters in your project and template emitters. Select the emitters you want to include in the new system, and click the green **Plus sign** icon (**+**) to add them. Then click **Finish** to create the system. If you choose an existing emitter, the system will inherit from those emitters. If you choose a template emitter, the system will have no inheritances. Also, the template emitter is an instance that can either be strictly local to that system, or you can save it as a separate emitter asset.



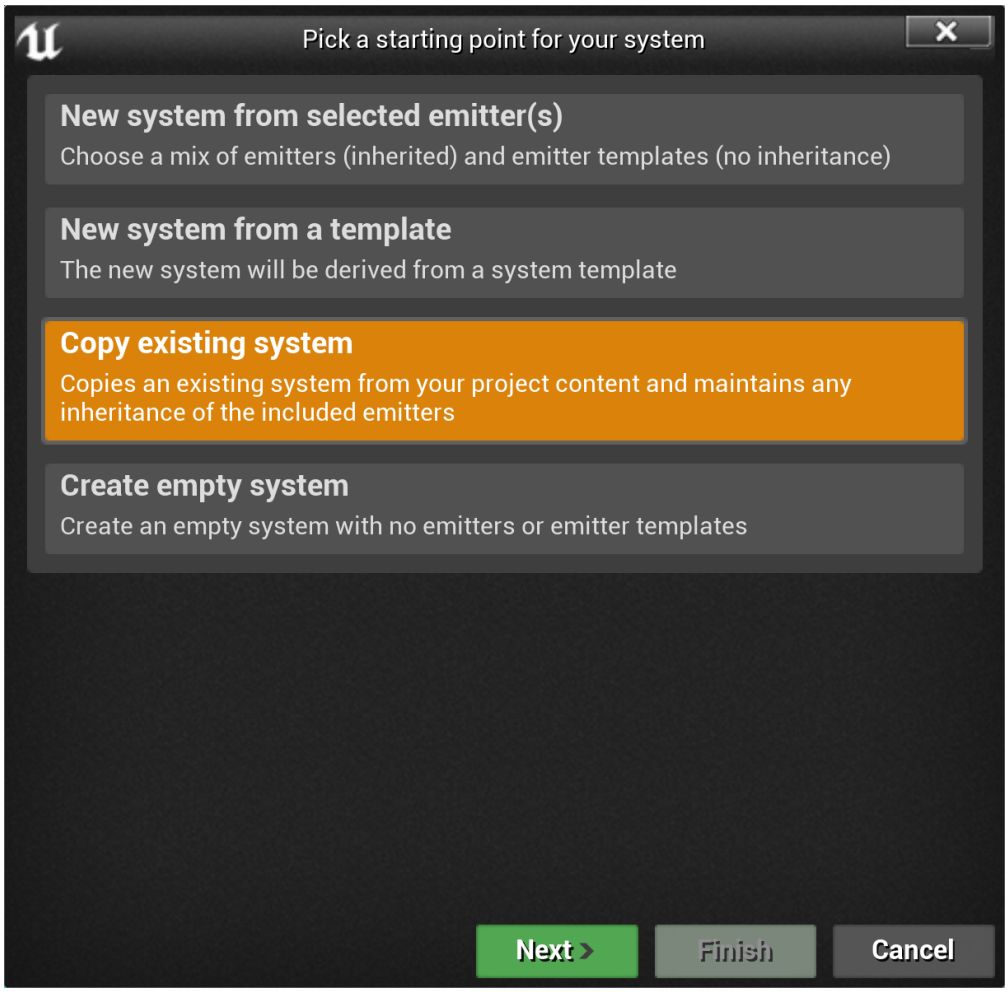
Click image for full size.

New system from template: If you select this option and click **Next**, you can choose from a list of templates that represent several commonly used effect systems. As with emitter templates, this list can be curated by art leads or creative directors. If you are new to UE4, this option will give you an example of how FX systems are built in Niagara.



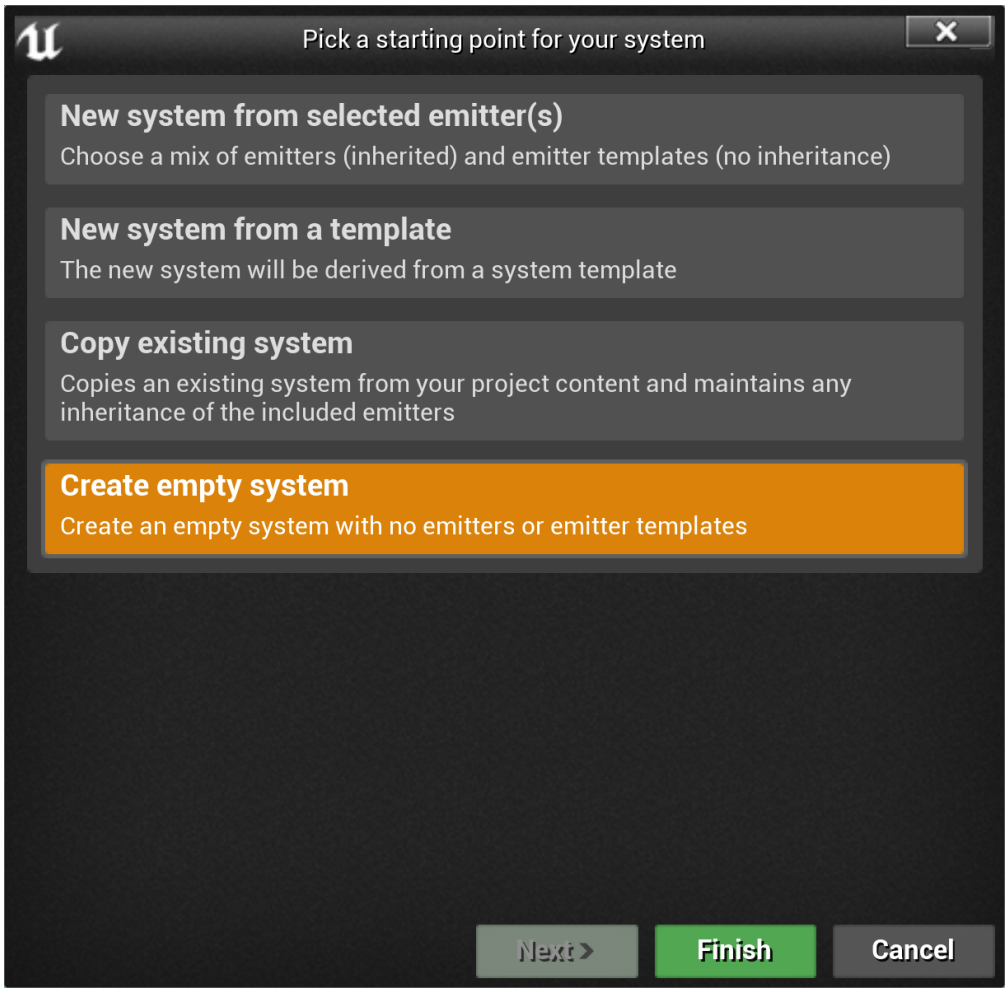
Click image for full size.

Copy existing system: If you select this option and click **Next**, a list of existing systems displays. Choose one of them to copy, then click **Finish**.



Click image for full size.

Create empty system: If you select this option, your system contains no emitters or emitter templates. This option is useful if you want to create a system that is totally different from your other systems.

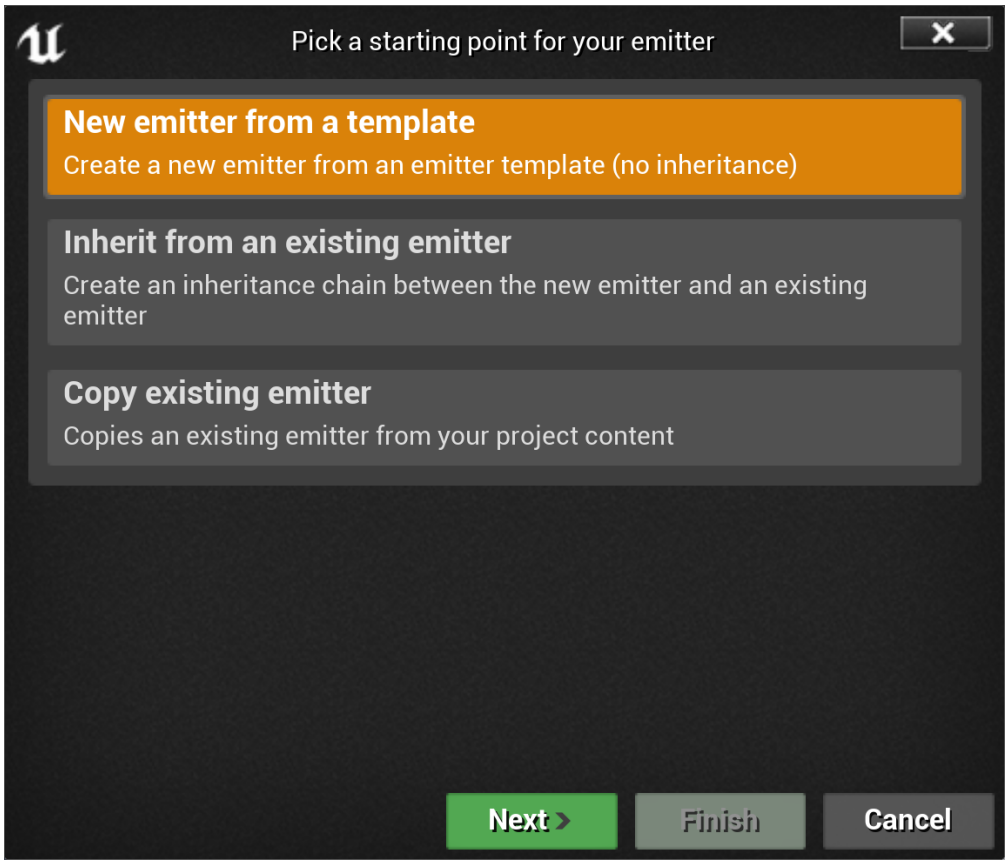


Click image for full size.

Emitter Wizard

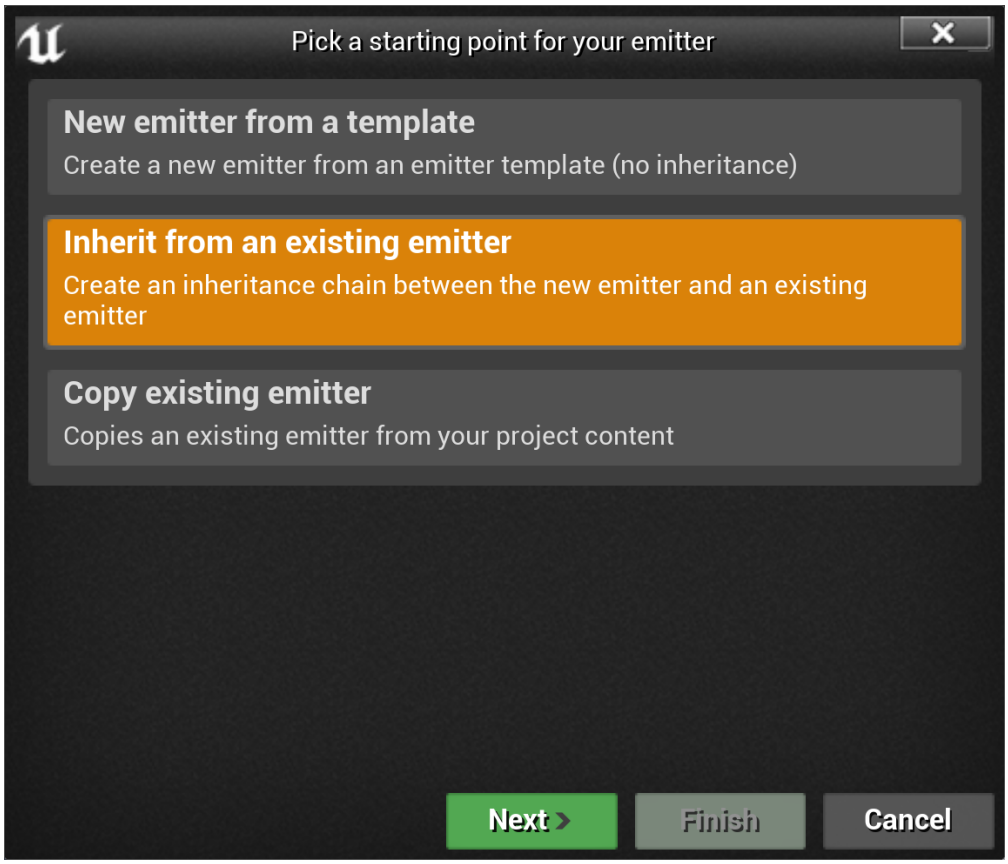
The Emitter Wizard offers the following options for creating a new emitter:

> **New emitter from a template**: If you select this option, you can choose from a list of templates that present several types of commonly used effects. In a large development studio, art leads or creative directors can curate the list of templates, ensuring that the company's best practices are baked into the templates. These templates also offer a great starting place if you are new to UE4.



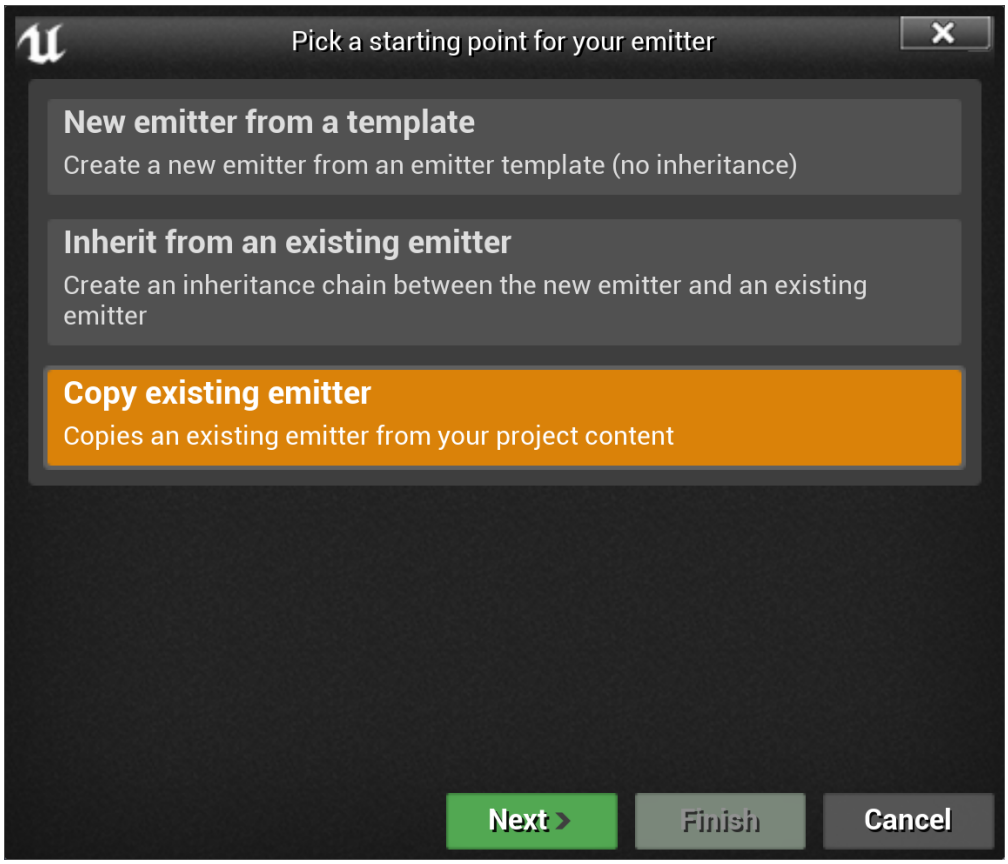
Click image for full size.

Inherit from an existing emitter: If you select this option, you can create a new emitter that inherits properties from an existing emitter. This option makes the new emitter a child of the existing emitter you selected. If you need many emitters that all have certain properties in common, this is a good option to choose. You can make changes to the parent emitter, and all child emitters will reflect those changes.



Click image for full size.

Copy existing emitter: If you select this option, you can create a new emitter that is a copy of an emitter you already created. This can be useful if you need to create several similar emitters. Click Next after selecting this option, and a list of available emitters displays. You can then choose which one you want to copy.



Click image for full size.

Niagara VFX Workflow

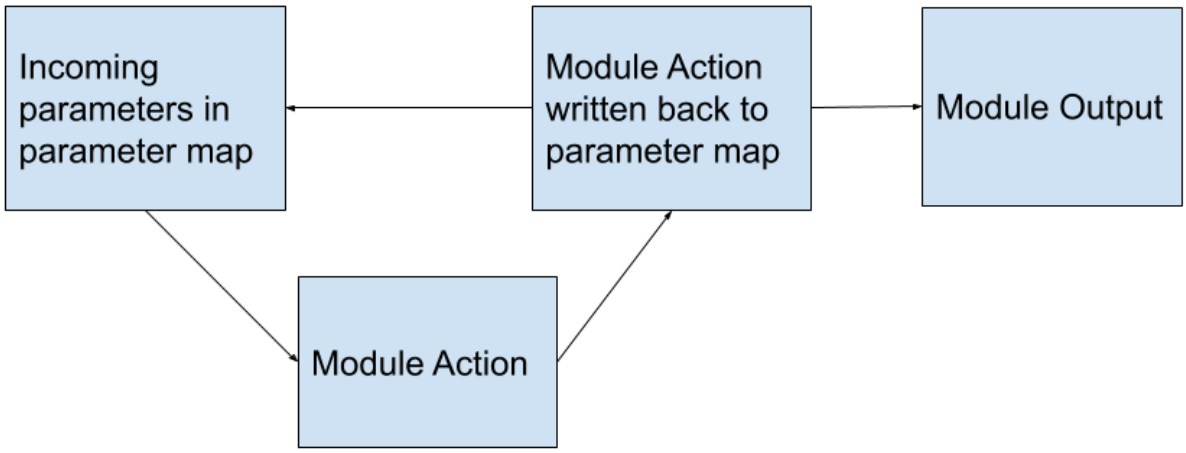
Create Emitters

When you create your emitters, you place modules in the stack that define how that effect will look, what actions it will take, and so on. In the **Emitter Spawn** group, place modules that define what will happen when the emitter first spawns. In the **Emitter Update** group, place modules that affect the emitter over time. In the **Particle Spawn** group, place modules that define what will happen when particles spawn from the emitter. In the **Particle Update** group, place modules that affect particles over time. In the **Event Handlers** 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.

Create Systems

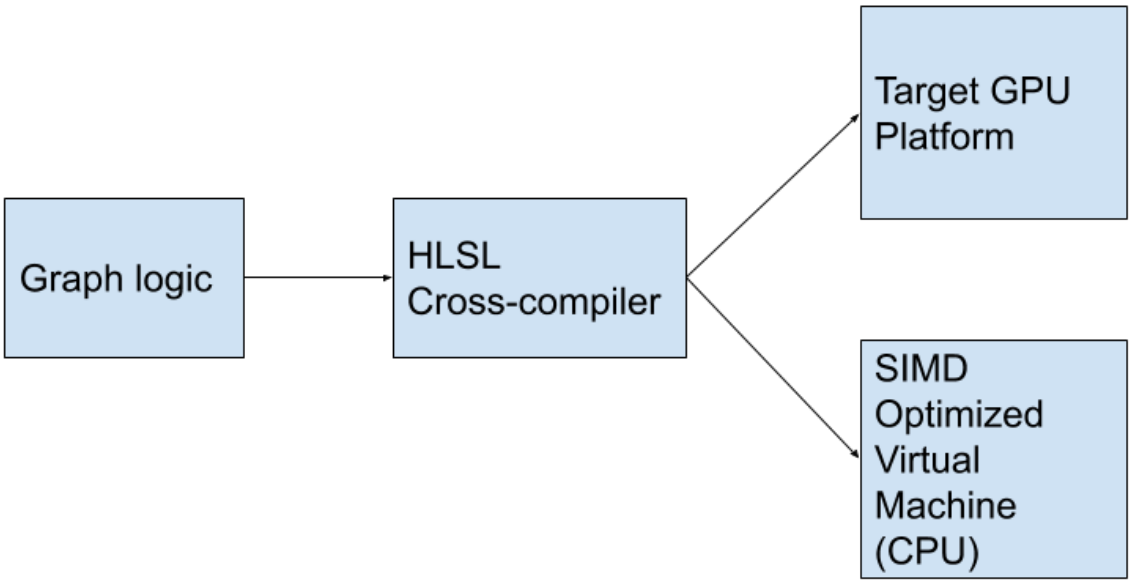
Combine individual emitters into one system, which displays the entire visual effect you want to create. There are modules specific to systems, and some elements of the editor act differently when you are editing a system instead of an emitter. When you are editing a system in the Niagara Editor, you can change or override any modules in any emitter included in the system. You can also manage timing for the included emitters using the Niagara Editor's Timeline panel.

Create a Module



Module function flow

- Modules accumulate to a temporary namespace, then you can stack more modules together. As long as they contribute to the same attribute, the modules will stack and accumulate properly.
- When writing a module, there are many functions available for you to use:
  - Boolean operators
  - Math expressions
  - Trigonometry expressions
  - Customized functions
  - Nodes that make boilerplate functions easier
- Once you create a module, anyone else can use it.
- Modules all use HLSL. The logic flow is as follows:



Remember that each module, emitter and system you create uses resources. To conserve resources and improve performance, look through the modules already included in Niagara to see if you can accomplish your goal without creating a new module. Dynamic Inputs can be used to great effect here.

- With a flat hierarchy, you cannot effectively locate and use the assets you already have in your library, which leads to people recreating those assets. Duplication of effort lowers efficiency and increases costs.
- Hierarchical inheritance increases discoverability and enables effective reuse of existing assets.
- Anything inherited can be overridden for a child emitter in a system.
- Modules can be added, or can be reverted back to the parent value.
- This is also true with emitter-level behaviors such as spawning, lifetime, looping, bursts, and so on.

- Dynamic inputs are built the same way modules are built.
- Dynamic inputs give users infinite extensibility for inheritance.
- Instead of acting on a parameter map, dynamic inputs act on a value type.
- Any value can be driven by Graph logic and user-facing values.
- Dynamic inputs have almost the same power as creating modules, but can be selected and dropped into the stack without actually creating new modules.
- Existing modules can be modified and customized in many ways by using and chaining Dynamic Inputs; this can reduce module bloat and improve performance.

- Any inline value can be converted into an HLSL expression snippet.
- Users can access any variable in the particle, emitter, or system, as well as any HLSL or VM function.
- This works well for small, one-off features that do not need a new module.

- Events are a way to communicate between elements (such as particles, emitters, and systems).
- Events can be any kind of data, packed into a payload (such as a struct) and sent. Then anything else can listen for that event and take action.
- Options you can use:
    - Run the event directly on a particle by using Particle.ID.
    - Run the event on every particle in a System.
    - Set particles to spawn in response to the event, then take some action on those particles.
- Events are a special node in the graph (structs). How to use the Event node:
    - Name the event.
    - Add whatever data you want to it.
    - Add an Event Handler into the Emitter stack.
    - Set the options for the Event Handler.

- There is a separate execution stack for events.
    - You can put elaborate graph logic into Event Handlers.
    - You can have a whole particle system set up, with complex logic, and then have a whole separate set of behaviors that occur when the event triggers.

- There is an extensible system to allow access to arbitrary data.
- Arbitrary data includes mesh data, audio, external DDC information, code objects, and text containers.
- Data interfaces can be written as plugins for greater extensibility moving forward.
- Users can get any data associated with a skeletal mesh by using a skeletal mesh data interface.

- Using Houdini, you can calculate split points, spawn locations, impact positions, impact velocity, normals and so on.
- You can then export that data from Houdini to a common container format (CSV).
- You can import that CSV into Niagara in your UE4 project.