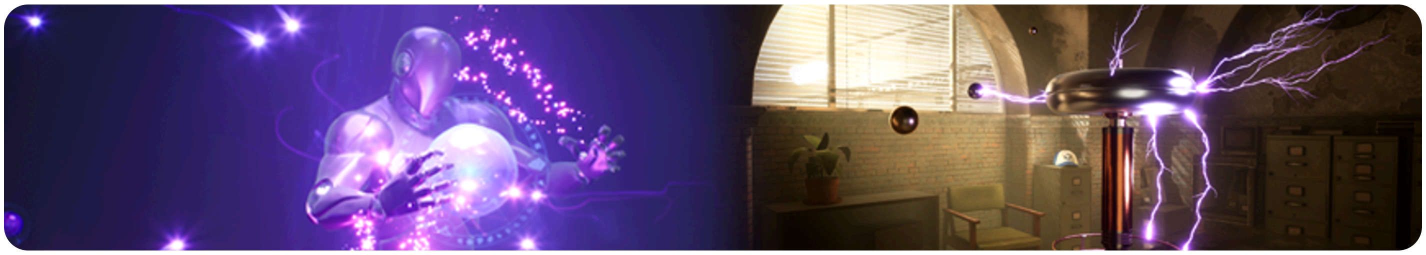
Niagara Key Concepts

This page explains the key concepts and design philosophy behind Niagara.



Niagara Design Philosophy

Why Reinvent Visual Effects for Unreal Engine?

Unreal Engine is expanding its user base, and is now used in many industries outside of the game development space. Examples are:

- Architectural visualization
- Industrial design, such as automotive designs
- Virtual TV and film production

Our users are more diverse than ever before—from design students, to small indie developers, to large professional studio teams, to individuals and companies outside the game industry. As we move forward, Epic developers will not know everything about every industry using Unreal Engine. We wanted to create a visual effects (VFX) system that would work for all our users, across industries.

Our Goals for a New VFX System

We wanted to create a new system that would give all users the flexibility to create the effects they need. Our goals for our new VFX system were:

- It puts full control in the hands of the artists.
- It is programmable and customizable on every axis.
- It provides better tools for debugging, visualization, and performance.
- It supports data from other parts of Unreal Engine, or from external sources.
- It gets out of the way of the user.

How Niagara Achieves Our Goals

Data Sharing

Total user control starts with access to data. We want the user to be able to use any data from any part of Unreal Engine, as well as use data from other applications. So we decided to expose everything to the user.

Particle Payload

In order to expose all this data to the user, we have to establish how someone can use that data. Namespaces provide containers for hierarchical data. For example, Emitter.Age contains data for an emitter; Particle.Position contains data for a particle. Our parameter map is the particle payload that carries all of the particle's attributes. As a result of this, everything becomes optional.

Many Types of Data Can Be Added

Any type of data can be added as a particle parameter. You can add complex structs, transform matrices, or Boolean flags. You can add these or any other data type, and use that in your effects simulation.

Combining the Graph Paradigm and the Stack Paradigm

There are advantages to both the stack paradigm (such as that used in Cascade) and to the graph paradigm (such as is used in Blueprints). Stacks provide users with modular behavior and readability. Graphs give users more control over behavior. Our new effects system combines the advantages of both of these paradigms.

Hierarchy for Niagara's Hybrid Structure

Modules

Modules work in a graph paradigm—you can create modules with HLSL in the Script Editor using a visual node graph. Modules speak to common data, encapsulate behaviors, and stack together.

Emitters

Emitters work in a stack paradigm—they serve as containers for modules, and can stack together to create various effects. An emitter is single-purpose, but it is also reusable. Parameters transfer up to the emitter level from modules, but you can modify modules and parameters in the emitter.

Systems

Like emitters, systems work in a stack paradigm, and also work with a Sequencer timeline — which you can use to control how the emitters in the system behave. A system is a container for emitters. The system combines these emitters into one effect. When editing a system in the Niagara Editor, you can modify and override any parameter, module or emitter that is in the system.

Niagara Selection Stack and Stack Groups

Particle simulation in Niagara conceptually operates as a stack — simulation flows from the top of the stack to the bottom, and executes modules in order. Crucially, every module is assigned to a group that describes when the module is executed. For example, modules that initialize particles or that act when a particle spawns are in the **Particle Spawn** group.

Within each group, there may be multiple *stages*, which are called at particular points in a system's life cycle. Emitters, systems, and particles all have **Spawn** and **Update** stages by default. Spawn stages are invoked in the first frame where that group exists. For example, systems invoke their Spawn stage when the system is first instantiated in the level and activated. Particles invoke their Spawn stage whenever the emitter emits a particle, and Spawn instructions will be executed for each new particle that is created. Update stages are invoked in every frame where the system, emitter or particle is active.

There are also advanced stages, such as **Events** and **Simulation Stages**, that can be added onto the Spawn and Update flow. **Events** are invoked whenever a particle generates a new event and an emitter is set to handle that event. Where possible, the event handler stages occur in the same frame, but after the originating event. **Simulation Stages** are an advanced GPU feature. This feature enables multiple Spawn and Update stages to happen in sequence, and is useful for constructing complex structures like fluid simulations.

A module is an item, but an item is not a module. *Modules* are editable assets a user can create. *Items* refer to parts of a system or emitter that the user cannot create. Examples of items are system properties, emitter properties, and renderers.

Stages, Groups, Namespaces and Data Encapsulation

By adding each module to a *stage* (Update, Spawn, Event, or Simulation) in a *group* (System, Emitter, or Particle), you can control when a module executes as well as what data a module operates on. Stack groups are associated with **Namespaces** that define what data the modules in that group can read or write to.

For example, modules that execute in the **System** Group can read and write to parameters in the System Namespace, but can only read from parameters that belong to the Engine or User Namespaces. As execution goes down the stack from the System Group to the Emitter Group, modules executing in the **Emitter** Group can read and write to parameters in the Emitter Namespace, but can only read from parameters in the System, Engine and User Namespaces. Modules in the **Particle** Group can only read from parameters in the System and Emitter Namespaces.

Because modules in the Emitter groups may read from parameters in the System Namespace, simulation that is relevant for all emitters can be performed once by modules in the System group, and the results of that (stored in the System Namespace) can be read by the Emitter group modules in each individual emitter. This continues with Particle group modules which can read from parameters in the System and Emitter Namespaces. Refer to the table below for a more concise representation of these relationships.

Module Group	Read From Namespaces	Write To Namespaces
System	System, Engine, User	System
Emitter	System, Emitter, Engine, User	Emitter
Particle	System, Emitter, Particle, Engine, User	Particle