Scalability and Best Practices

Learn about Niagara's scalabilty settings

Overview

The main goals of the strategies outlined in this document are to reduce the amount of work done by your particle systems on an individual and aggregate level. The guidance in the first tutorial should point you to where the extra work is being done, and this tutorial should give you the tools and options to make changes in those areas. When reducing the amount of work being done in general we can either avoid unnecessary work, or choose solutions that have less overhead.

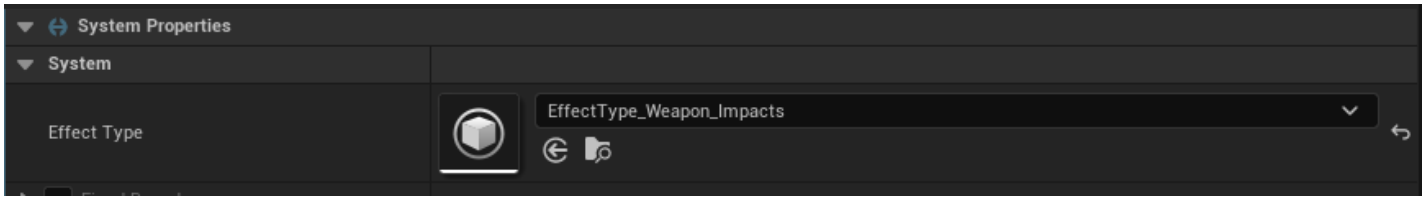
Instance Counts: Reducing the Number of Systems in the Level

Even optimized systems can impact performance if there are enough of them in a level. By default we tick instances of the same system in batches with the Niagara World Manager, but the more instances there are of the system, the more work needs to be done on the game thread. In general, simulating the same number of particles across more instances will be less performant in terms of work being done, but having more instances lets Niagara cull particles

on a more granular level, so there is a tradeoff between a small number of large scale instances and a large number of small scale ones.

Instance Count reduction Strategies

Overall you can reduce the number of instances by controlling the number of Systems you spawn in your level, or by more aggressively culling them after they have been spawned. Both approaches have their places, and we offer tools for both, but it's usually best to start with setting up some core Effect Types for your project, and making sure all your Niagara systems have one assigned.



Scalability and Effect Types

We offer many scalability options in Niagara including reusable assets that can be applied to any system with our Effect Type asset, and overrides at the System, Emitter and Renderer level directly in Niagara assets. They allow for controls based on scalability levels e.g. Medium, High and Epic and per-platform, so you can differentiate what each level means for each platform your project targets. Many of these settings will reduce the number of Niagara System Instance in your levels by culling them, but can also be used to scale down the number of particles being spawned, to disable expensive renderers, or disable emitters and systems entirely.

System as a Service

Another approach to reducing instance counts is to get the most out of your systems that do exist. For particle effects like projectiles and impacts it’s common for several instances to be spawned in short succession in a concentrated area. Pooling can be leveraged here to some extent to prevent re-allocating the components for these instances, but there is still an overhead to re-activating them, and relies on previous instances completing and being returned to the pool. A systems as a service approach seeks to add new particles to an

existing system rather than adding new systems to a level. 5.4 is introducing new features to streamline this workflow, but in older versions user parameters can be used instead. This strategy will be covered more in depth in the next tutorial.

Emitter Counts: Reducing Overhead at the Asset Level

When building Niagara systems, especially if you are switching from Cascade's paradigm, it can be natural to add a new emitter for every unique visual element to your system or to have slight variations in behavior. This can negatively impact performance, however, since there is an overhead associated with every emitter.

Niagara uses a virtual machine for running its scripts on the CPU. We do this for two main reasons, extensibility and parallelization. With a VM that mimics the SIMD nature of the GPU, we can have all our Niagara scripts work for both CPU and GPU by default. Building them as scripts from the start lowers the barrier to entry for users wishing to create their own, improving extensibility, and designing them for SIMD allows Niagara to parallelize their execution more readily on the CPU.

The trade-off for this VM is the overhead associated with every emitter. Even when targeting your particles for GPU simulation, we still run the System and Emitter scripts on the CPU so they can interact with the game thread. This means the overhead is fixed, and the only way to avoid it is to reduce the number of emitters in a system or the number of instances in your level.

For 5.4 we have experimental Lightweight Emitters, that forgo extensibility, and other features in favor of performance. In future versions they will be a viable alternative in many use cases to reducing emitter counts, as these emitters will not have a VM overhead.

Mitigation Strategies

Below are some strategies that can be used to reduce the number of emitters in a system. The general idea is to consolidate them where the simulation's behavior is more or less the same, and the only difference is the visual elements (renderers) or initialization data like starting position.

Spawn Index

The spawn index can be used to create groupings of particles in your emitter that have common behavior or starting data. Particles will save their spawn index as a parameter, which can be used to dynamically select different data in Niagara scripts. A common example of this is to create groups for sampling different sections of a mesh, which would otherwise have identical behavior. For example contrails in Fortnite that sample from the character's hands and feet. In Cascade one emitter was created for each trail, but in Niagara one emitter is created for the whole effect with four spawn groups, each set to sample a different bone or socket on the character's skeleton.

Multiple Renderers

With Niagara a single emitter can have multiple renderers, so multiple visual elements can be created from each particle the emitter simulates. A common use case is to have a light renderer combined with a sprite or mesh renderer so the particles can emit light and still have associated geometry. This can also be used to vary the geometry by randomly selecting between different meshes or sprite representations. This can also be used to transition from meshes to sprites based on distance. You can bind different parameters to different renderers to allow for slight variations within a simulation. For example you could have a different position binding for each renderer to give an offset to one, but still have it follow the other.

Mesh renderers allow for an array of meshes to be specified, and have a mesh index binding that controls which mesh from the array to render. This can be used in place of multiple renderers to have a wider variety of mesh shapes, without needing to duplicate renderers, and set-up their enabling and disabling. There are costs associated with both visibility tags and mesh arrays. Overall these should win out over having another emitter, but when deciding between multiple emitters or these strategies there may be different performance characteristics in your project. If your project requires even further optimization you may want to compare the two.

Pooling: Memory Management and Reducing Allocations

The Niagara function library allows for selecting a pooling method when spawning systems through Spawn System at Location and Spawn System Attached. When pooled, Niagara components based on the same asset aren't allocated and garbage collected like normal. Instead a pool is kept of allocated components that are no longer in use, and are re-used when spawning a new component that uses the same asset. This avoids the cost of allocation and garbage collection for components that are frequently used.

Niagara systems have control over their pool sizes and priming in the System properties under performance. Priming the pool can reduce runtime allocations, but can cause hitches if the asset is loaded dynamically.

Simulation Cost: Optimizing Niagara Scripts

GPU vs CPU

As noted previously in the section about emitter count overhead, there is a CPU overhead to all Niagara simulations. Choosing between CPU and GPU simulations only changes the target of the particle simulations, that is, only the scripts in the Particle Spawn, Particle Update, and Simulation Stages are changed based on the Sim Target chosen for an emitter.

Particle scripts have the largest opportunity for parallelization, so they benefit the most from targeting the GPU, and in most cases GPU sims are more performant, and allow for a greater number of particles. For emitters with a small number of particles a CPU sim may be better suited, as GPU resources cannot be divided as granularly. The exact breakdown is hardware dependent, but a simulation of 1 particle can potentially take up the same resources as one with 64 particles.

It's also important to note that in some projects, and on some platforms the GPU is the bottleneck, and CPU sims may be a better fit. This is especially common for platforms that have less GPU memory like mobile. The flipside can be true, and a project that is CPU bound may still benefit from moving simulations, even those with a small number of particles, to the GPU. This is one of the many reasons it's important to validate these performance assumptions when developing your project.

Miscellaneous Best Practices

Profiling your project will give you the most concrete data on where your project's bottlenecks are, and is the best way to find performance issues, but best practices can go a long way to avoid common, or easy mistakes to miss during content authoring. These best practices have been gathered over time from Epic's effects artists working on projects like Fortnite, and represent general advice shared among our team. There are exceptions to this advice and the final call will have to be made on a per-project basis as to where to sacrifice performance for functionality and visuals.

Fixed vs Dynamic Bounds

Generally speaking, fixed bounds will be more performant than dynamic bounds since it requires less work. There are exceptions however, for example, an effect that is relatively small but travels a large distance, like a bullet trace traveling across a large level. Its fixed bounds would be much larger than the actual visuals, but for most of the time the effect would be out of view, however it could still be relevant due to its bounds.

In addition to changing bounds types, we have cvars to reduce Dynamic Transform Update Frequency

- fx.Niagara.EmitterBounds.DynamicSnapValue
- fx.Niagara.EmitterBounds.DynamicExpandMultiplier
- fx.Niagara.EmitterBounds.FixedExpandMultiplier

Warmup

It's best to avoid warmup for particle systems, since each frame in the warmup needs to be evaluated sequentially and can very easily lead to hitches.

Mesh Asset Hygiene

Simplifying the static mesh assets used by Niagara will make generating and rendering those meshes more performant, without changing their geometry. For example:

- Removing Collisions from the static mesh asset.
- Turning off "Cast Shadow" on the LOD used by Niagara if shadows aren't needed. For example, when using a transparent material.
- Turning off the distance field, by setting the LOD's Distance Field Resolution Scale to 0.0
- Making sure the mesh doesn't have extra UV channels.

Large Bursts of Particles

If you need to create a large number of particles, spawning them all in one frame can cause hitches. Spreading the spawn over several frames can spread the work out and avoid these hitches.

Current Frame Data

For particles that don’t rely on up-to-date game data, disabling “Require Current Frame Data” will allow Niagara to start their ticks as early in the frame as possible, spreading their work over a longer period of time. For particle systems that don't move or sample something that moves, this can usually be disabled, but even some moving particles can sample the last frame's data and still look fine. It can be useful to experiment wish disabling this flag and seeing if there is a noticeable change.

Moving Complex Operations up the Stack

Complex work that's the same for each particle or emitter, can often be moved up the stack so the work is not repeated as often. Additionally interactions with Data Interfaces and Niagara Parameter Collections can be more performant with reads in the Emitter and System scripts. Even if the full scope of a module's work would be different for each particle the sampling side of things often can be separated, which could still have performance wins.

Data Interfaces

Similar to complex work, having duplicate data interfaces in emitters and particles can use unnecessary memory. If you can move the data interface higher in the stack, that should be more performant.

Data Interfaces as User Parameters can also use more memory since a new UObject is created for every Niagara component, or every time Set Asset is called on an existing Niagara component.

Avoid Events

Particle reads using the attribute reader data interface are generally more performant than events, and can usually achieve the same behavior.

Sorting

Sorting is disabled by default, but when enabled sort tasks default to running on the GPU, and will add overhead to the GPU, and take up extra memory in the index buffer. This can result in poor draw performance.

CVars can force CPU emitters to sort on the CPU

- Niagara.GPUSorting.CPUToGPUThreshold
- Niagara.GPUCulling.CPUToGPUThreshold