



THE DEFINITIVE GUIDE TO CREATING

ADVANCED VISUAL EFFECTS IN UNITY

2021 LTS EDITION

Contents

Introduction	5
Contributors	6
Getting started with real-time VFX	7
Visual workflow	9
Core graphics packages.....	10
Additional sample content	10
The Built-in Particle System.....	11
Introduction to the VFX Graph	14
The VFX Graph Asset and component	15
The VFX Graph window	17
Graph logic	19
Systems, Contexts, and Blocks	20
Properties and Operators	24
The Blackboard	25
Group Nodes and Sticky Notes	26
Subgraphs	27
Attributes	29
Events.....	30
Event Attributes	32
Exploring VFX sample content.....	33
More resources	38
Additional references.....	38
Shader Graph integration	39
Visual effects by example	41

The VFX Graph Samples (HDRP)	42
GooBall	43
Physics-based effects	48
The Ribbon Pack	50
Meteorite sample	53
Mesh sampling effects	58
Skinned Mesh sampling	59
More examples	61
The Spaceship Demo	61
Corridor Outliner	62
Augmented reality user interface	64
Holographic table	68
Core effect	70
Interactivity	72
Event Binders	73
OnPlay and OnStop Events	73
Mouse Event Binder	75
Rigidbody Collision Event Binders	76
Trigger Event Binder	77
Visibility Event Binder	78
Timeline	79
Event Attributes	81
Property Binder	81
Output Events	84
Pipeline tools	88
Point Caches	89
Point Cache Bake Tool	90
Using Point Caches	91

Signed Distance Fields	92
Using SDFs	92
SDF Bake Tool	93
Vector Fields	94
VFXToolbox	95
Image Sequencer	96
TFlow (Asset Store)	96
Digital Content Creation tools	96
SideFX Houdini	97
Autodesk Maya	97
Blender	97
Adobe Photoshop	97
Optimization	98
The Unity Profiler and Frame Debugger	99
Debug modes	101
Bounds	103
Mesh LOD	105
Mesh Count	107
Particle rendering	107
Case study: Spaceship optimization	108
New and future developments	111
Universal Render Pipeline	112
Lit output	112
2D Renderer support	112
Graphics Buffer support	113
Additional resources	116
Video tutorials	117
VFX projects on GitHub	118
Professional Training for Unity creators	119

INTRODUCTION



Galaxy VFX Graph from the Spaceship Demo

Whether you plan on shooting fireballs from your fingertips or traveling through a wormhole, visual effects (VFX) in a game make the impossible, well, possible. Not only do they enhance the atmosphere and help tell the story of your game, visual effects bring imagined worlds to life with details that can truly captivate your players.

Unity is pushing the boundaries of real-time graphics with tools such as the VFX Graph. This node-based editor enables technical and VFX artists to design dynamic visual effects – from simple common particle behaviors to complex simulations involving particles, lines, ribbons, trails, meshes, and more.

Our comprehensive guide is intended for artists and creators looking to incorporate the VFX Graph into their game applications. It provides specific instructions on how to use the VFX Graph and its related tools to build real-time visual effects in Unity.

Taking into account experiences of solo developers and those on hundred-person teams, this guide is filled with many examples to make our AAA-quality tools more accessible. This way, everyone can find themselves at the fun part of game design.

Contributors

Wilmer Lin is a 3D and visual effects artist with over 15 years of industry experience in film and television, now working as an independent game developer and educator. Wilmer's feature credits include *X-Men: Days of Future Past*, *The Incredible Hulk*, and *The Chronicles of Narnia: The Lion, the Witch, and the Wardrobe*.

Significant contributions were also made by Unity experts Marie Guffroy, technical artist, Mathieu Muller, senior product manager for high-end graphics, and Vlad Neykov, senior manager, Foundation Quality, alongside Thomas Iché, VFX artist and specialist.

The background image shows a majestic mountain range under a vast, colorful sky. The mountains are rugged and partially obscured by mist or low-hanging clouds. The lighting suggests either a sunrise or sunset, with warm, golden hues on the peaks and a darker blue-grey in the shadows.

GETTING STARTED WITH REAL-TIME VFX



VFX Graph examples

Today's gamers crave deeply immersive experiences. As hardware advancements push the limits of what mobile and console platforms can do, what used to be available only for creating Hollywood blockbusters can now be attained in real-time.

Visual effects in games continue to have their moment as both interest and investment in advanced graphics trend upward. After all, gameplay effects transport your players into the action.

It's difficult to imagine a fantasy role-playing game (RPG) without characters casting magic, or a hack-and-slash brawler without glowing weapon contrails. When race cars plow the asphalt, we expect that they kick up a cloud of dust in their wake.

Not even your environments are the same without visual effects. If you're telling a film noir detective story, you'll likely cloak your cityscape in rain and fog. But if your characters go on a quest through the wilderness, you might make your foliage and vegetation sway in the wind, reacting to their every move.



V Rising by Stunlock Studios is a game made with the High Definition Render Pipeline and VFX Graph.

Visual effects uniquely enhance the gaming experience. However, creating them requires you to don the mantle of a multidisciplinary artist who can manipulate shape, color, and timing.

That's where the VFX Graph comes in. This sophisticated tool is equipped with workflows that reflect those of motion picture VFX – except working at 30, 60, or more frames per second (fps).



Image from a project in development by [Sakura Rabbit](#), made with Unity's VFX tools

Visual workflow

For complex, AAA-level visual effects on high-end platforms, use the VFX Graph to create GPU-accelerated particles in an intuitive, node-based interface.

Leverage the [VFX Graph](#) to:

- Create one or multiple particle systems
- Add static meshes and control shader properties
- Create events via C# or [Timeline](#) to turn parts of your effects on and off
- Extend the library of features by creating subgraphs of commonly used node combinations

- Use a VFX Graph inside of another VFX Graph (e.g., smaller explosions as part of another, larger effect)
- Preview changes at various rates and/or perform step-by-step simulation

The VFX Graph works with the [Universal Render Pipeline \(URP\)*](#) and the [High Definition Render Pipeline \(HDRP\)](#). It also adds support for the **Lit** outputs and **2D Renderer** available with URP. Check the VFX Graph feature comparison for all render pipelines [here](#).

The VFX Graph requires compute shader support to maintain compatibility with your device. Supported devices include:

- macOS and iOS platforms using [Metal graphics API](#)
- Linux and Windows platforms with [Vulkan](#) or [GLES3](#) APIs
- Android for a subset of high-end compute capable devices (only with URP)

* For Unity 2021 LTS onward, Camera Buffer access is only available with HDRP. Read more about the VFX Graph's compatibility in the [documentation](#).

Core graphics packages

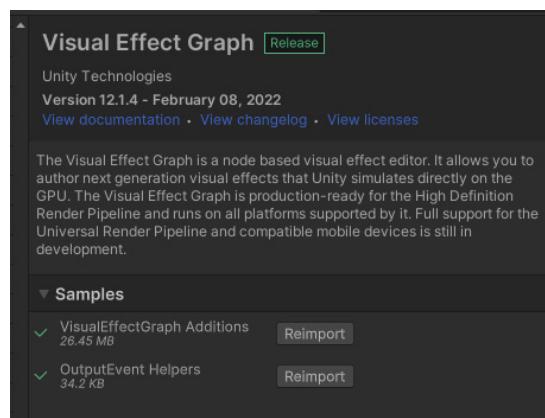
As of Unity 2021 LTS, you can efficiently install VFX Graph in your project. Core graphics packages are now embedded within the main Unity installer to ensure that your project is always running on the latest, verified graphics code.

When you install the latest release of Unity, the most recent packages for URP, HDRP, Shader Graph, VFX Graph, and more, are included in the installation.

For details, see [Getting started with VFX Graph](#) or read [this blog post](#).

Additional sample content

If you're new to the VFX Graph, consider installing the **Visual Effect Graph Additions**, available under **Samples** in the **Package Manager**. These additions contain some basic examples for you to explore.



Extra content in the Visual Effect Graph Additions

You'll find Prefabs for fire, smoke, sparks, and electricity. Each sample shows a stripped down effect to illustrate fundamental graph logic and construction. Just drag and drop one of the sample **Prefabs** into the **Hierarchy** to see them in action.



Sample Prefabs included with the Visual Effect Graph Additions

The Built-in Particle System

If your target platform does not meet the [minimum system requirements](#) for the VFX Graph, the [Built-in Particle System](#) is another option for creating real-time effects in Unity. Access the Built-in Particle System by selecting **GameObject > Effects > Particle System**.

Like the VFX Graph, the Built-in Particle System allows you to create a [variety of effects](#) such as fire, explosions, smoke, and magic spells. It remains a valuable tool for real-time effects, even though it renders fewer particles than the VFX Graph.

The primary distinction between the VFX Graph and the Built-in Particle System lies in their hardware. The Built-in Particle System is simulated on the CPU, whereas the VFX Graph moves many of the calculations to compute shaders, which run on the GPU.

The VFX Graph has the advantage of simulating millions of particles, but there's a caveat; being simulated on the GPU means that it's computationally nontrivial to send data back and forth to the CPU.

If you're using a mobile platform, you'll need to verify that it supports compute shaders in order to use the VFX Graph. Otherwise, you might need to use the Built-in Particle System for CPU-based effects.

Here's a side-by-side comparison:

CPU		GPU
Particle System		
Particle count	Thousands	Millions
Simulation	Simple	Complex
Physics	Underlying physics system	Scene representation <ul style="list-style-type: none">•Primitives (spheres, boxes, etc.)•Signed Distance Fields Depth buffer
Gameplay readback	Yes	No*
Other	NA	Can read frame buffers

Comparing the Built-in Particle System with the VFX Graph

The Built-in Particle System can use the underlying physics system and interact with gameplay more directly, but its particle count is limited and its simulations must stay relatively straightforward.

Download the [Particle Pack](#) from the Unity Asset Store to get a set of examples with the Built-in Particle System. This sample asset demonstrates a variety of in-game effects (fire, explosions, ice, and dissolves, among others). You can also check out this [Dev Takeover](#) for more information on using Shader Graph with the Built-in Particle System.

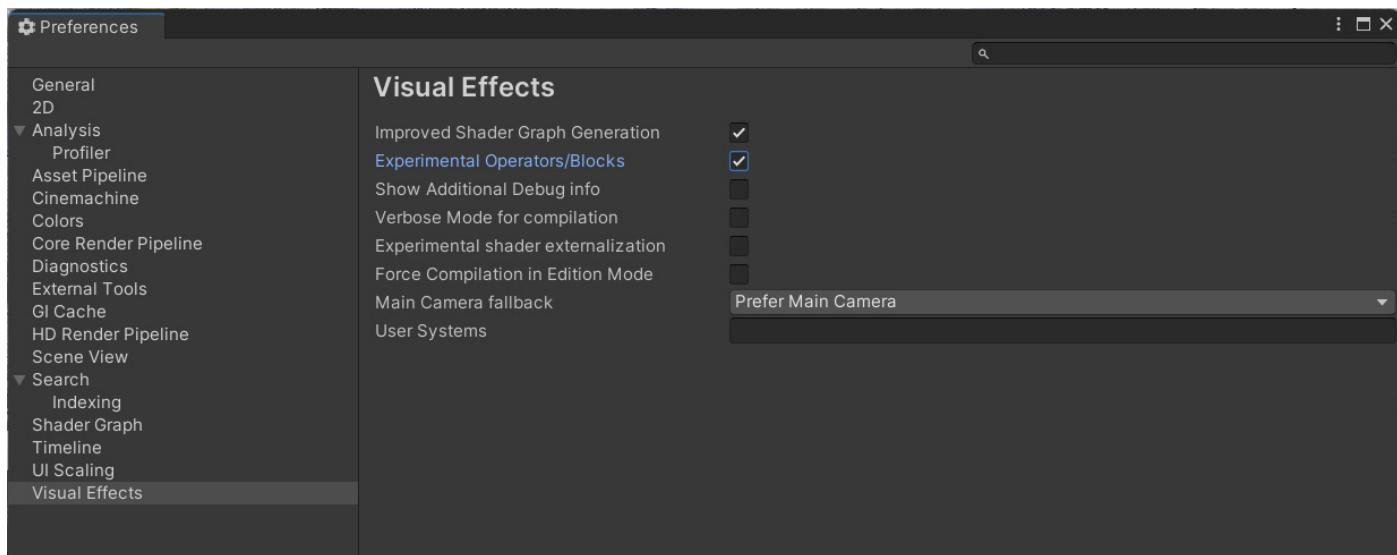
URP offers standard shaders ([Lit](#), [Unlit](#), [Simple Lit](#)) for the Built-in Particle System, whereas HDRP provides Shader Graph-based shader samples from the [HDRP package sample](#). You can review the particle system feature comparison for render pipelines [here](#).



The Particle Pack available on the Unity Asset Store

Note: Experimental features are not fully validated and are thus subject to change. The LTS version of Unity is recommended for production work.

You can enable experimental features from this guide via **Preferences > Visual Effects > Experimental Operators/Blocks**, as shown below.



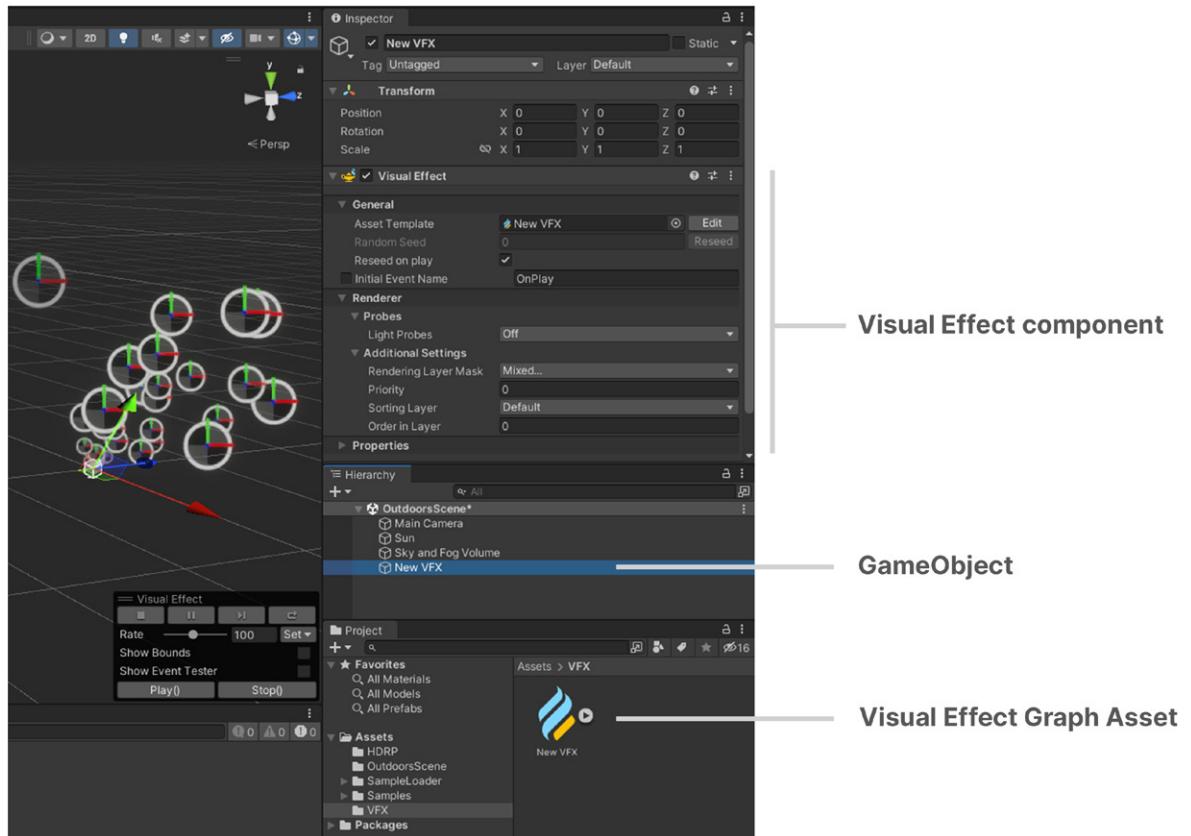


INTRODUCTION TO THE VFX GRAPH

Any visual effect in the VFX Graph is made up of these two parts:

- [Visual Effect \(VFX\) component](#) attached to a [GameObject](#) in the scene
- [Visual Effect \(VFX\) Graph Asset](#) that lives at the project level

As Unity stores each VFX Graph in the Assets folder, you must connect each asset to a Visual Effect component in your scene. Keep in mind that different [GameObjects](#) can refer to the same graph at runtime.



The VFX Graph Asset and Visual Effect component

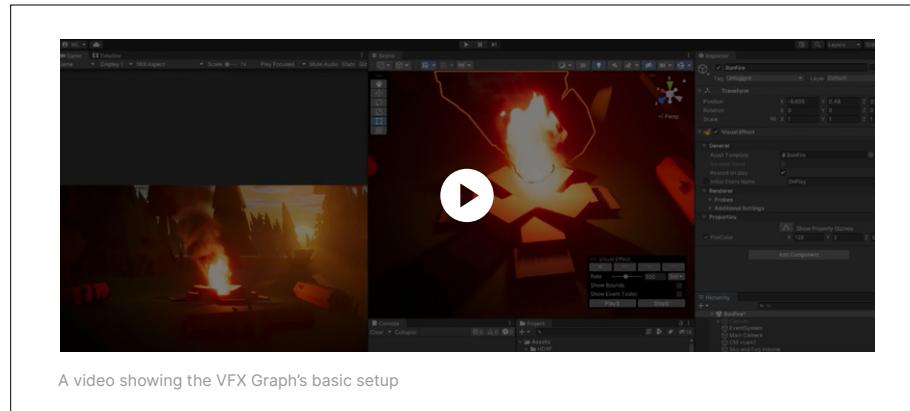
The VFX Graph Asset and component

To create a new visual effect, right-click in the **Project** window and navigate to **Create > Visual Effects > Visual Effects Graph**. This results in the VFX Graph Asset.

Unity provides a few ways to connect the **VFX Graph Asset** to a **GameObject** with a [Visual Effect component](#):

- Drag the resulting asset into the **Scene** view or **Hierarchy**. A new default [GameObject](#) will appear in the **Hierarchy** window.
- Assign the asset to an existing **Visual Effect component** in the **Inspector**. Create a **GameObject** separately and use the **Add Component** menu.

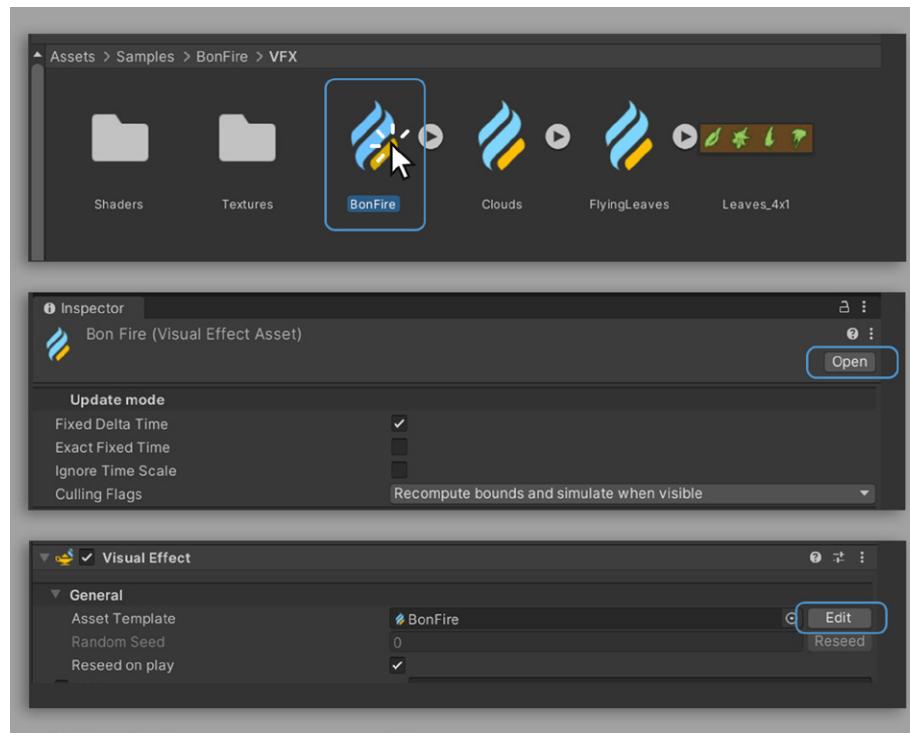
- With a **GameObject** selected, drag and drop the asset into the **Inspector** window. This creates the Visual Effect component and assigns the asset in one quick action.



The VFX Graph Asset contains all the logic. Select one of the following ways to edit its behavior:

- Double-click the **VFX Graph Asset** in the **Project** window.
- Select the **VFX Graph Asset** in the **Project** window and click the **Open** button in the header.
- Click the **Edit** button next to the **Asset Template** property in the **Visual Effect component**.

The asset opens in the [VFX Graph window](#), available under **Window > Visual Effects > Visual Effect Graph**.

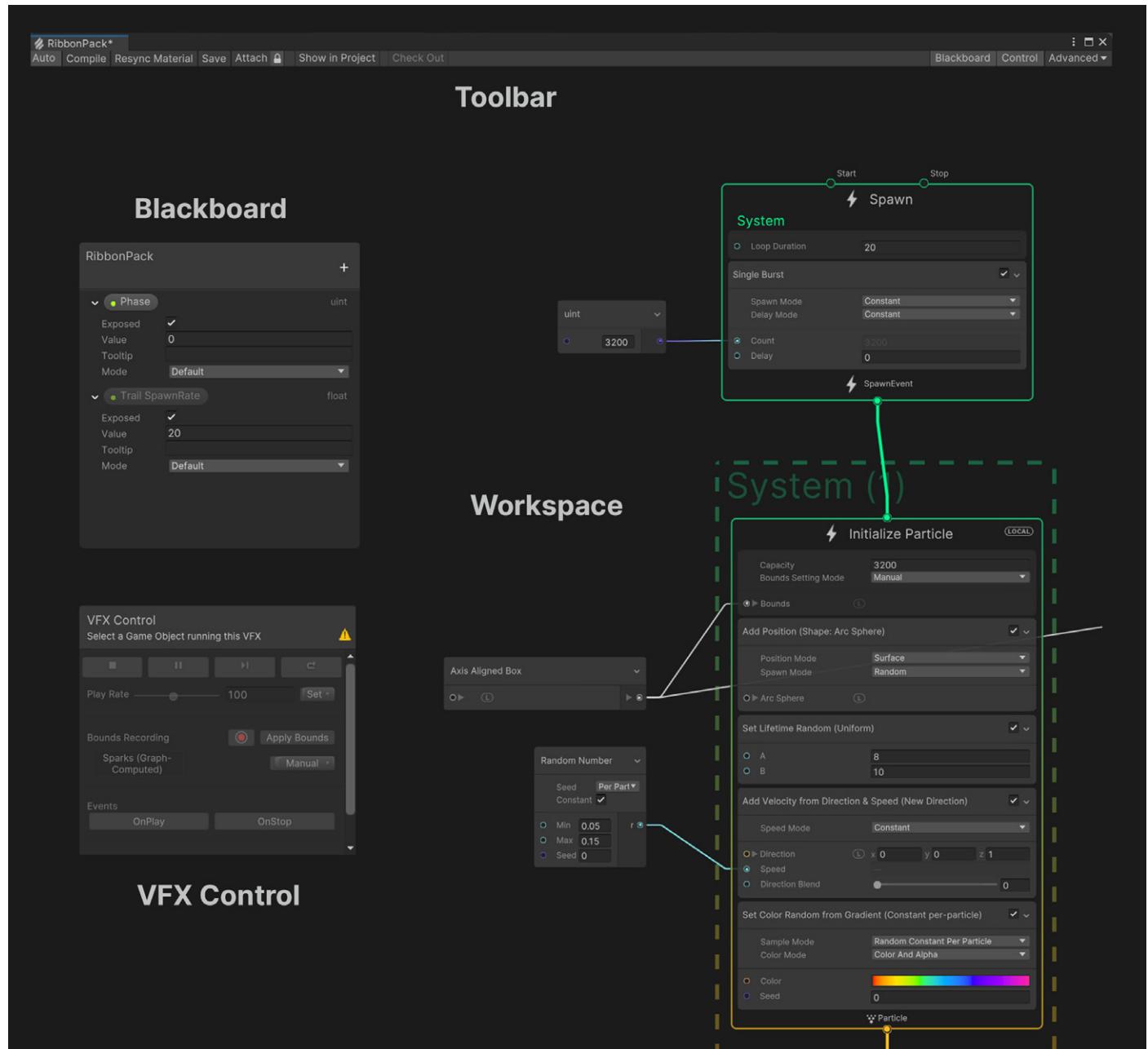


Three ways to open a VFX Graph

The VFX Graph window

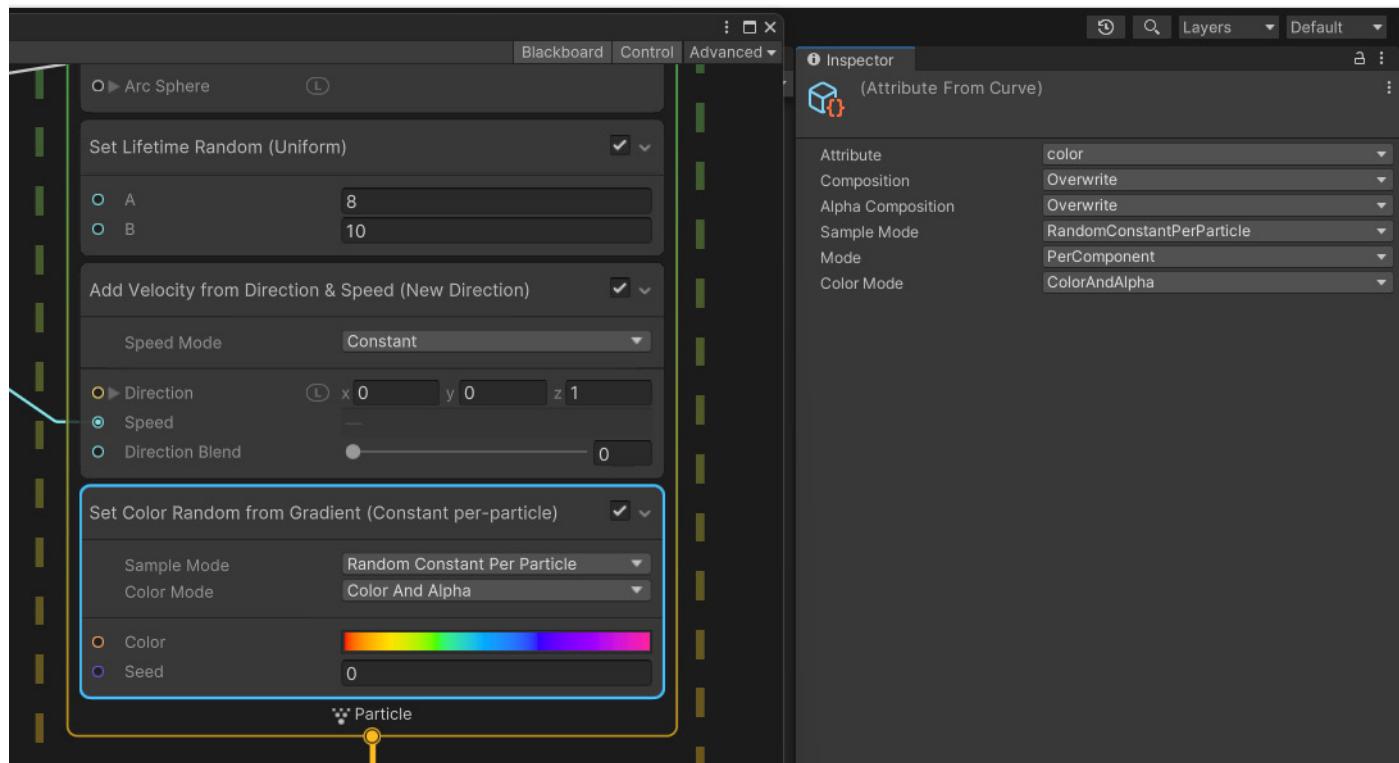
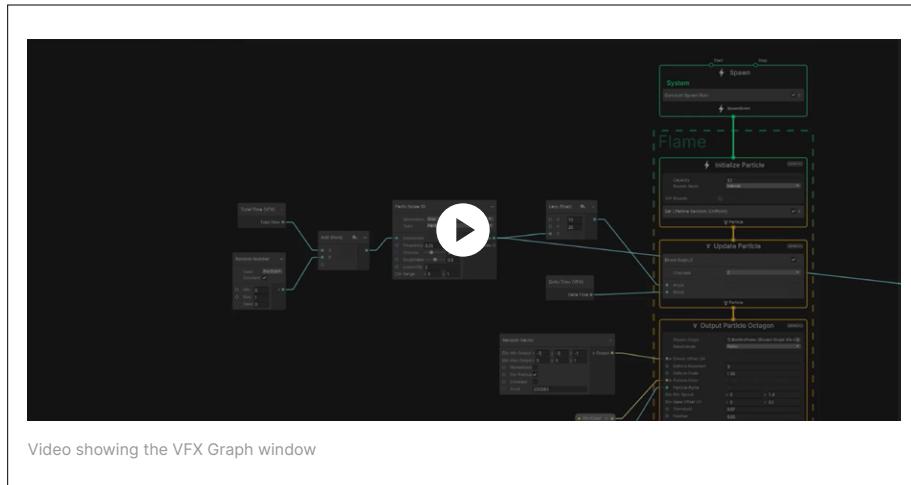
Familiarize yourself with this [window's layout](#), including its:

- **Toolbar:** To access Global settings, as well as toggles for several panels
- **Node workspace:** To compose and edit the VFX Graph
- **Blackboard:** To manage properties and Global variables that are reusable throughout the graph
- **VFX Control panel:** To modify playback on the attached GameObject



The VFX Graph window

Make sure you leave some space in the Editor layout for the Inspector. Selecting part of the graph can expose certain parameters, such as partition options and render states.

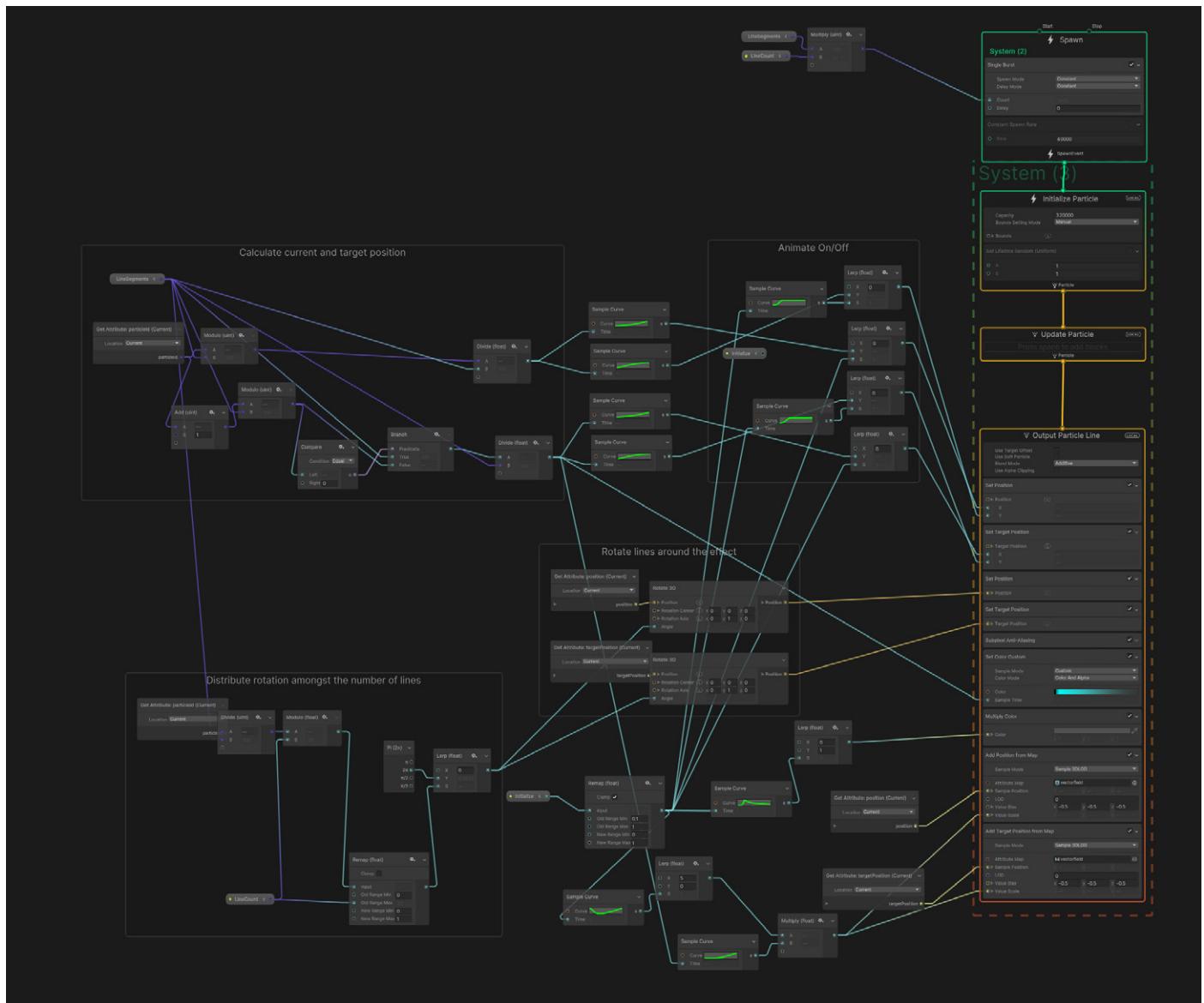


Use the Inspector to change certain parameters.

Graph logic

You must build your visual effect from a network of nodes inside the window's workspace. The VFX Graph uses an interface similar to other [node-based tools](#), such as **Shader Graph**.

Press the spacebar or right-click to create a new graph element. With the mouse over the empty workspace, select **Create Node** to create a graph's **Context**, **Operator**, or **Property**. If you hover the mouse above an existing Context, use **Create Block**.



A VFX Graph can consist of a complex network. [Download this example](#).

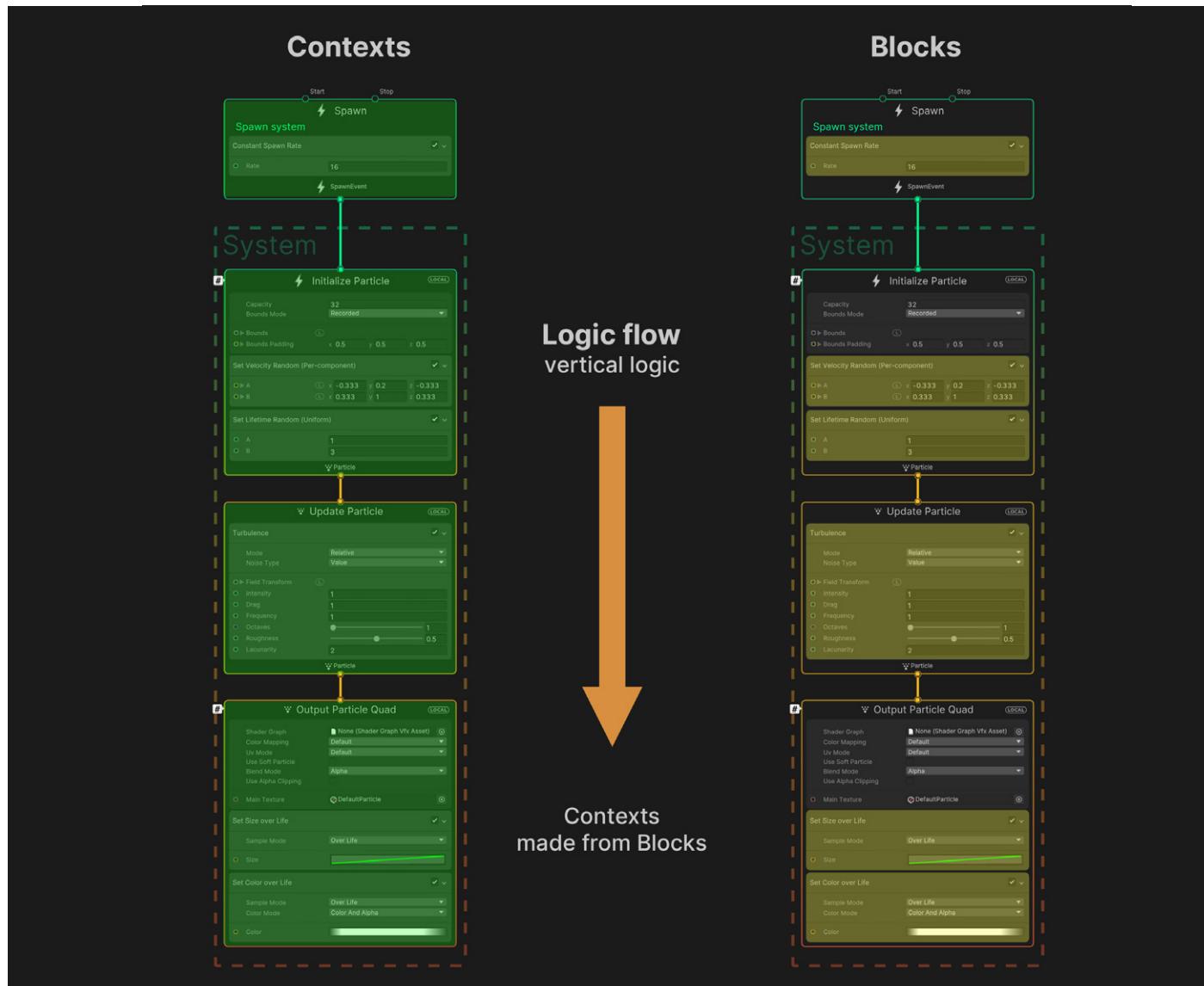
Opening up a complex VFX Graph can be daunting at first. Fortunately though, while a production-level graph can include hundreds of nodes, every graph follows the same set of rules – no matter its size.

Let's examine each part of the VFX Graph to learn how they work together.

Systems, Contexts, and Blocks

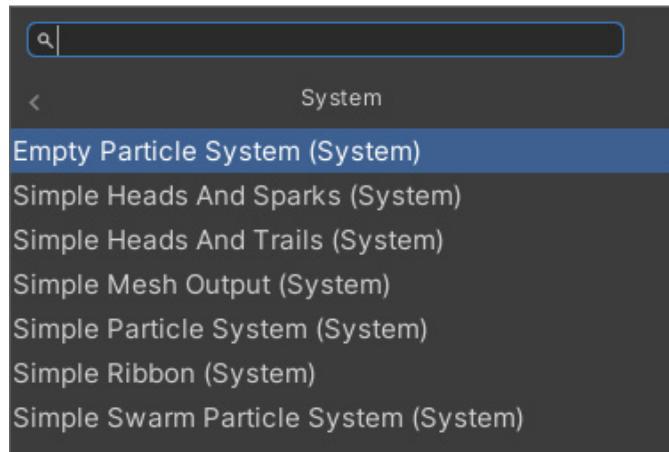
A VFX Graph includes one or more vertical stacks called [Systems](#). Systems define standalone portions of the graph and encompass several [Contexts](#). A System is denoted by the dotted line that frames the Contexts it consists of.

Each Context is composed of individual [Blocks](#), which can set [Attributes](#) (size, color, velocity, etc.) for its particles and meshes. Multiple Systems can work together within one graph to create the final visual effect.



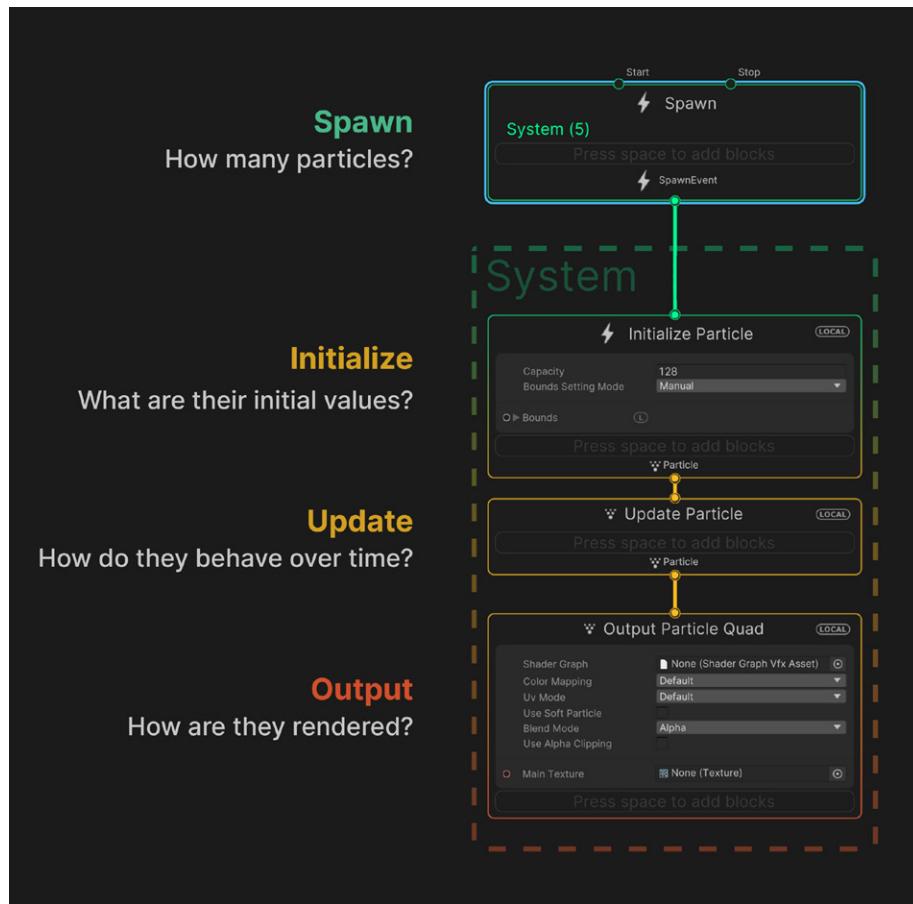
The vertical logic in a graph flows downward.

Find prebuilt templates under the **Create Node > Systems** menu to view some examples of graph logic.



The Systems menu

Notice the four Contexts present in the empty particle system graph:



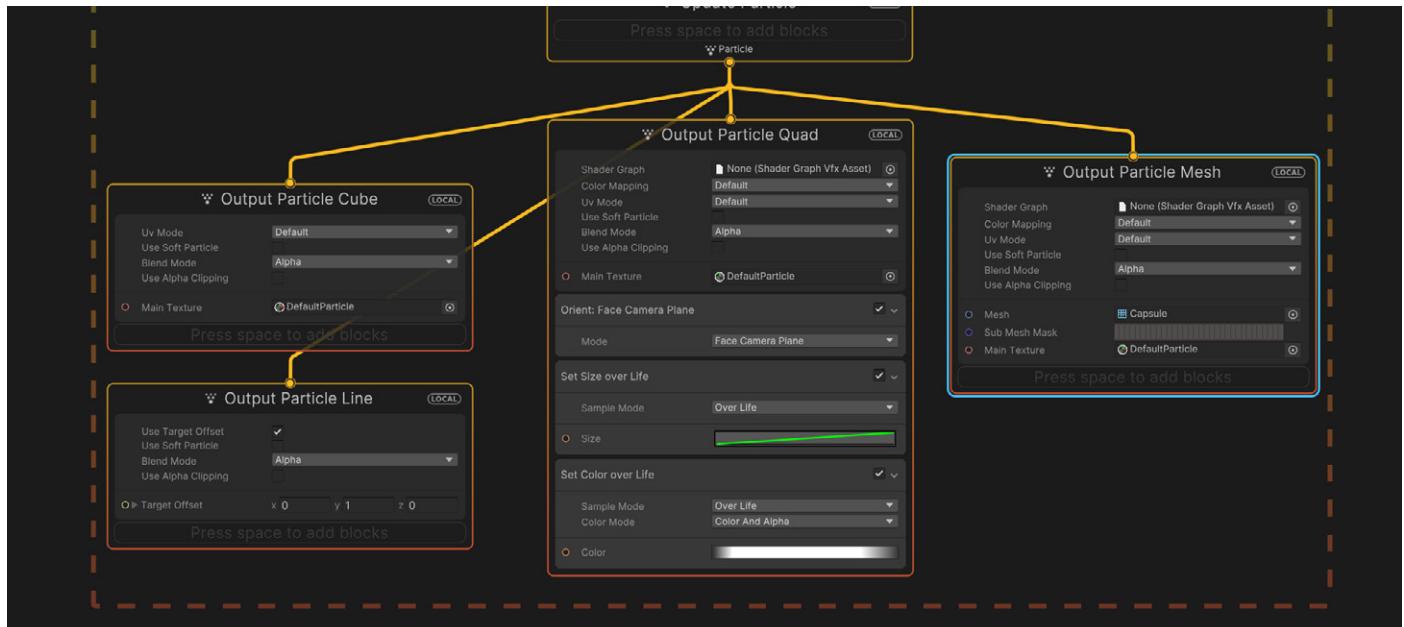
An empty particle system

The flow between the Contexts determines how particles spawn and simulate. Each Context defines one stage of computation:

- **Spawn:** Determines how many particles you should create and when to spawn them (e.g., in one burst, looping, with a delay, etc.)
- **Initialize:** Determines the starting Attributes for the particles, as well as the Capacity (maximum particle count) and Bounds (volume where the effect renders)
- **Update:** Changes the particle properties each frame; here you can apply Forces, add animation, create Collisions, or set up some interaction, such as with Signed Distance Fields (SDF)
- **Output:** Renders the particles and determines their final look (color, texture, orientation); each System can have multiple outputs for maximum flexibility

Systems and Contexts form the backbone of the graph's "vertical logic," or [processing workflow](#). Data in a System flows downward, from top to bottom, and each Context encountered along the way modifies the data according to the simulation.

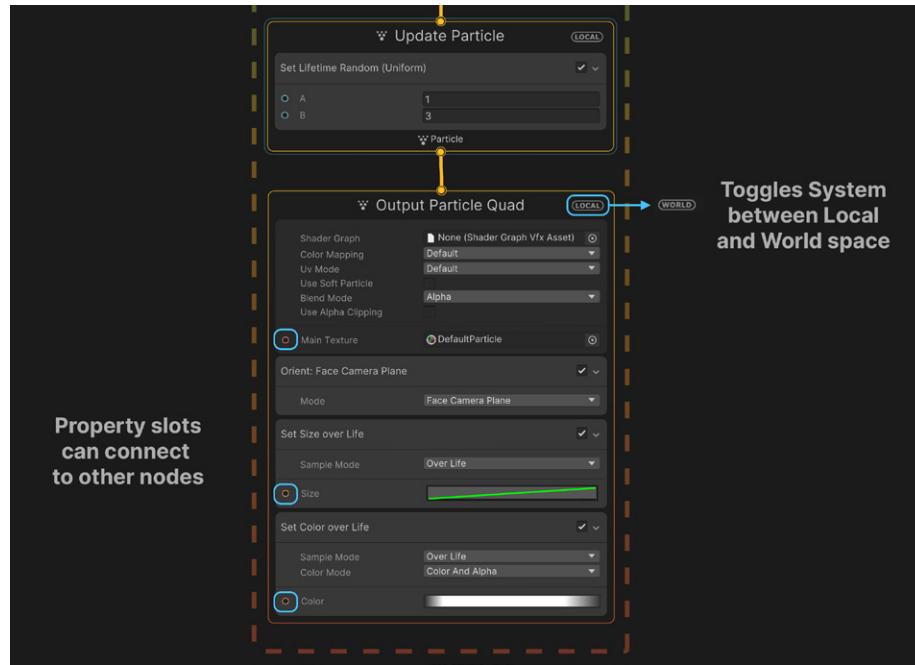
Systems are flexible, so you can omit a Context as needed or link multiple outputs together. This example shows more than one **Output Context** rendering within the same System.



More than one Output Context within the same System

Contexts themselves behave differently depending on their individual Blocks, which similarly calculate data from top to bottom. You can add and manipulate more Blocks to process that data.

Click the button at the top-right corner of a Context to toggle the System's simulation space between **Local** and **World**.



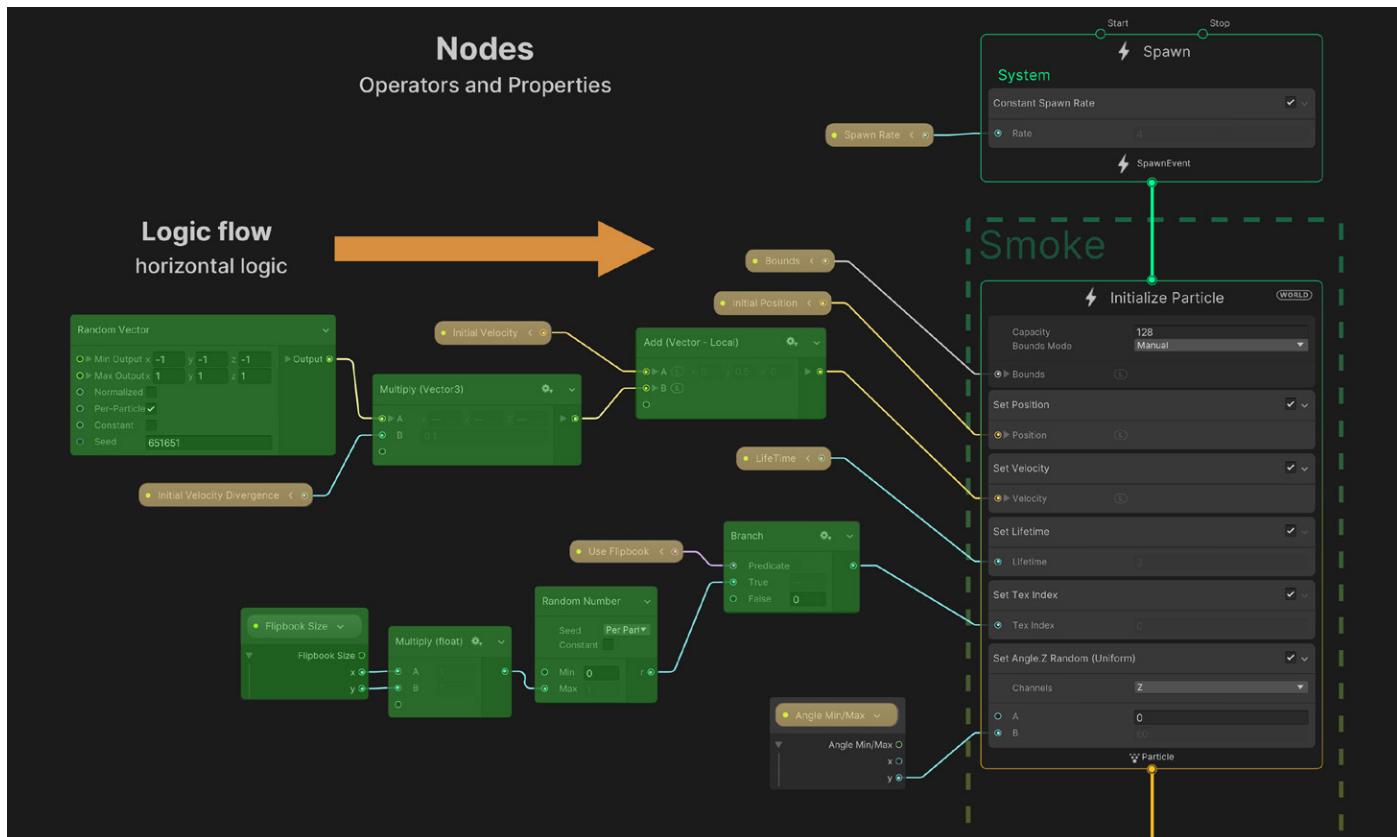
Examples of different Blocks

Blocks can do just about anything, from simple value storage for **Color**, to complex operations such as **Noises**, **Forces**, and **Collisions**. They often have slots on the left, where they can receive input from Operators and Properties.

See the [Node Library](#) for a complete list of Contexts and Blocks.

Properties and Operators

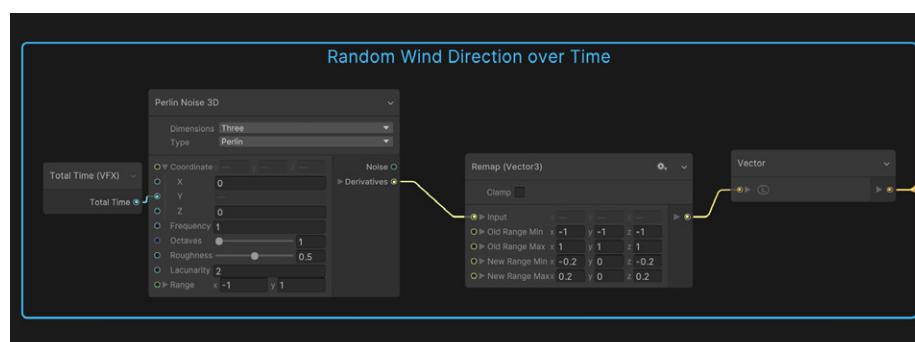
Just as Systems form much of the graph's vertical logic, Operators make up the "horizontal logic" of its [property workflow](#). They can help you pass custom expressions or values into your Blocks.



Horizontal logic

Operators flow left to right, akin to Shader Graph nodes. You can use them for handling values or performing a range of calculations.

Use the **Create Node** menu (right-click or press the spacebar) to create **Operator Nodes**. These Operators from the Bonfire sample, for instance, compute a random wind direction.

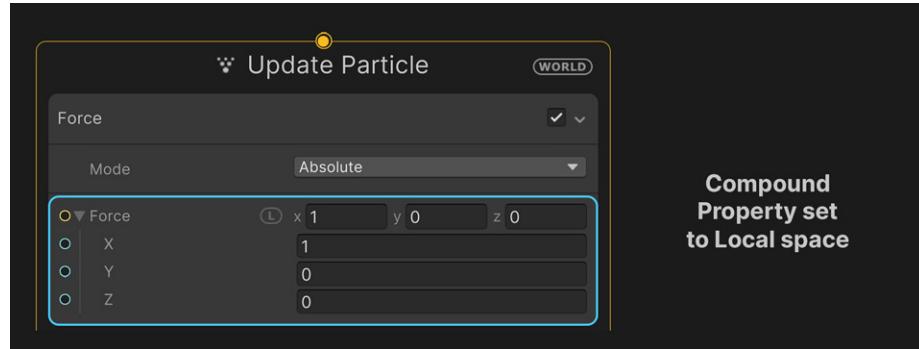


How wind direction is determined in the Bonfire sample

Properties are editable fields that connect to graph elements using the [property workflow](#). Properties can be:

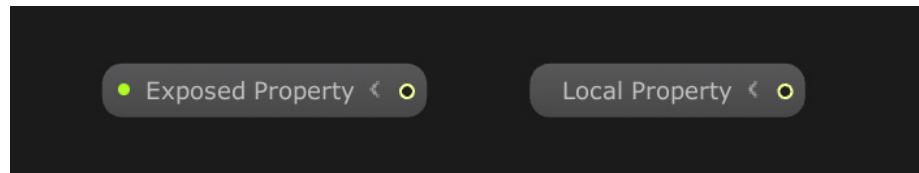
- Any **Type**, including [integers, floats, and booleans](#)
- Made from **Compound** components, such as [Vectors and Colors](#)
- [Cast and converted](#) (e.g., an integer to a float)
- [Local or World space](#); click the L or W to switch between them

Properties change value according to their actual value in the graph. You can connect the input ports (to the left of the Property) to other Graph nodes.



A Force Property in a Block

Property Nodes are Operators that allow you to [reuse the same value](#) at various points in the graph. They have corresponding Global properties that appear in the Blackboard.



Property Nodes

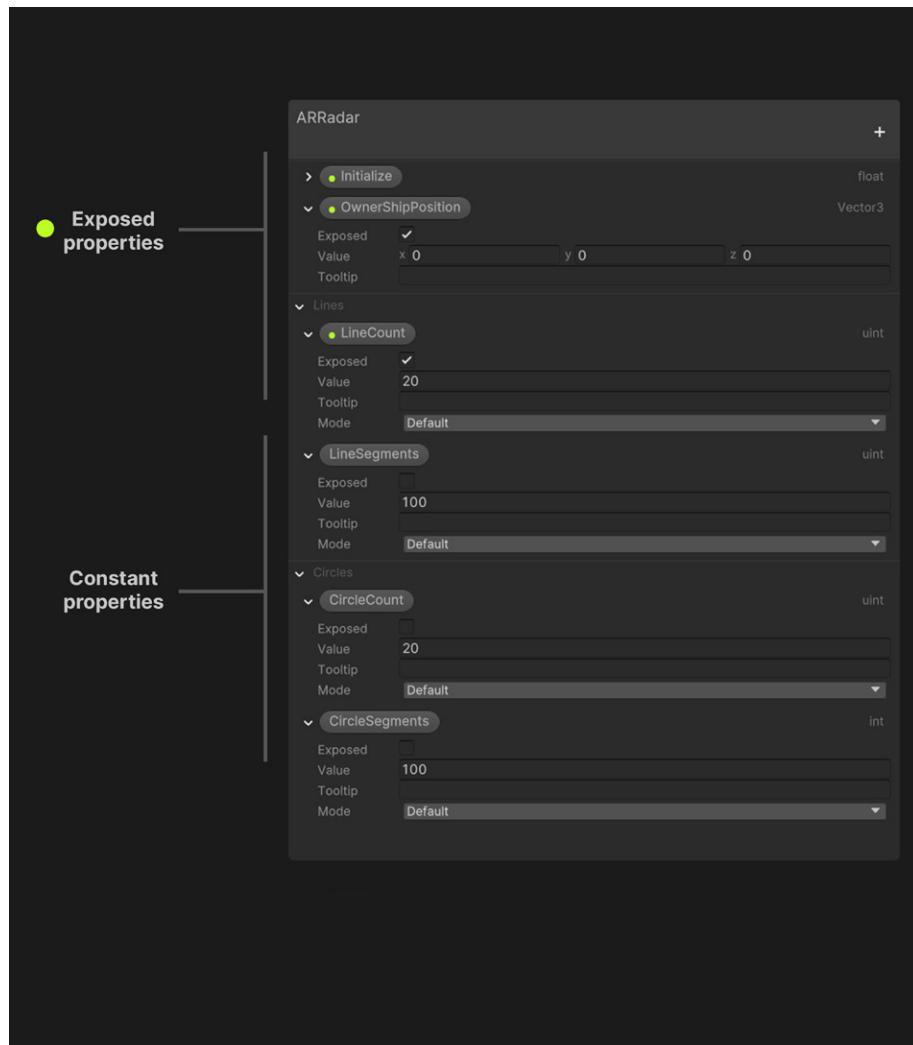
The Blackboard

A utility panel called the Blackboard manages Global properties, which can appear multiple times throughout the graph as Property Nodes.

Properties in the Blackboard are either:

- **Exposed:** The green dot to the left of any Exposed Property indicates that you can see and edit it outside of the graph. Access an Exposed Property in the Inspector via script using the [Exposed Property class](#).
- **Constant:** A Blackboard property without a green dot is a Constant. It is reusable within the graph but does not appear in the Inspector.

New properties are Exposed by default, and as such, appear in the Inspector. You must uncheck the **Exposed** option if you want to hide your property outside of the graph, and create **Categories** to keep your properties organized.



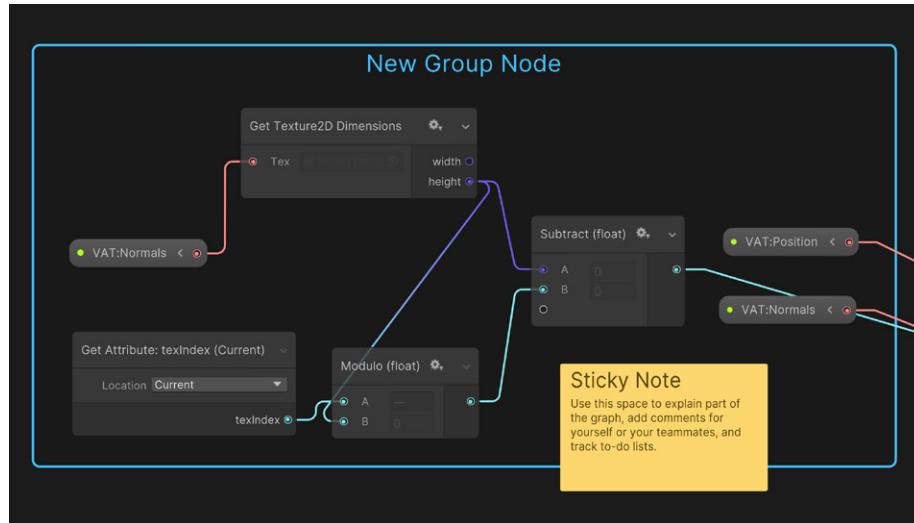
The Blackboard and its available properties

Group Nodes and Sticky Notes

As your graph logic grows, use Group Nodes and [Sticky Notes](#) to cut down on clutter. With Group Nodes, you can label a group of nodes and move them as one. On the other hand, Sticky Notes operate like code comments.

To create Group Nodes, select a group of nodes, right-click over them, then choose **Group Selection** from the **Context** menu. You can also drag and drop a node into an existing Group Node by holding the Shift key to drag it out. By deleting a Group Node, either with the Delete key or from the Context menu, you do not delete its included nodes.

Meanwhile, you can use Sticky Notes to describe how a section of the graph works, plus leave comments for yourself or your teammates. Add as many Sticky Notes as you need and freely move or resize them.

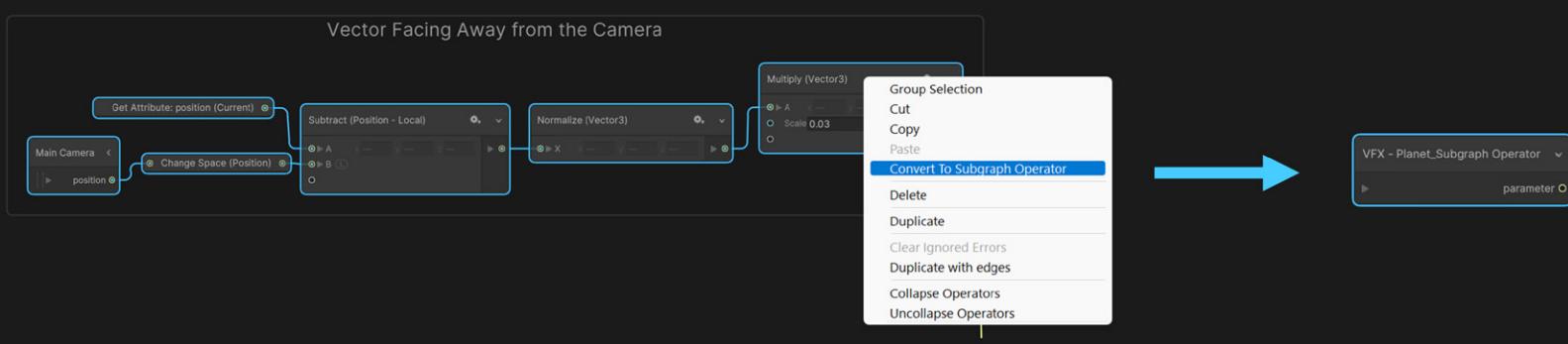


Work with Group Nodes and add Sticky Notes.

Subgraphs

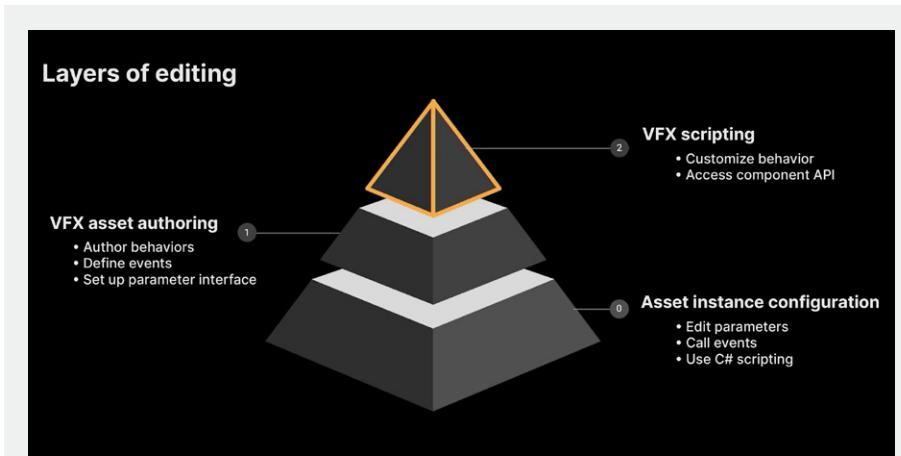
A Subgraph appears as a single node, which can help declutter your graph logic. Use it to save part of your VFX Graph as a separate asset that you can drop into another VFX Graph for reorganization and reuse.

To create a Subgraph, select a set of nodes and then pick **Convert To Subgraph Operator** from the right mouse menu. Save the asset to disk and convert the nodes into a single **Subgraph Node**. You can package Systems, Blocks, and Operators into different types of Subgraphs.



How to create a Subgraph

Creating a Subgraph is analogous to refactoring code. Just as you would organize logic into reusable methods or functions, a Subgraph makes elements of your VFX Graph more modular.



Layers of editing with the VFX Graph

The VFX Graph supports three different levels of editing:

- **Asset instance configuration:** Use this to modify any existing VFX Graph. Designers and programmers alike can adjust exposed parameters in the Inspector to tweak an effect's look, timing, or setup. Artists can also use external scripting or events to change preauthored content. At this level, you're treating each graph as a black box.
- **VFX asset authoring:** This is where your creativity can truly take charge. Build a network of Operator Nodes to start making your own VFX Graph, and set up custom behaviors and parameters to create custom simulations. Whether you're riffing off existing samples or starting from scratch, you can take ownership of a specific effect.
- **VFX scripting:** This supports more experienced technical artists or graphics programmers using the [component API](#) to customize the VFX Graph's behavior. With VFX scripting, your team can enjoy a more efficient pipeline for managing specific effects, and access advanced features like the Graphics Buffers.

Regardless of your experience level, you can start creating effects with the VFX Graph. Begin with a premade effect to get familiar with the workflow, and then gradually assemble your own graphs.

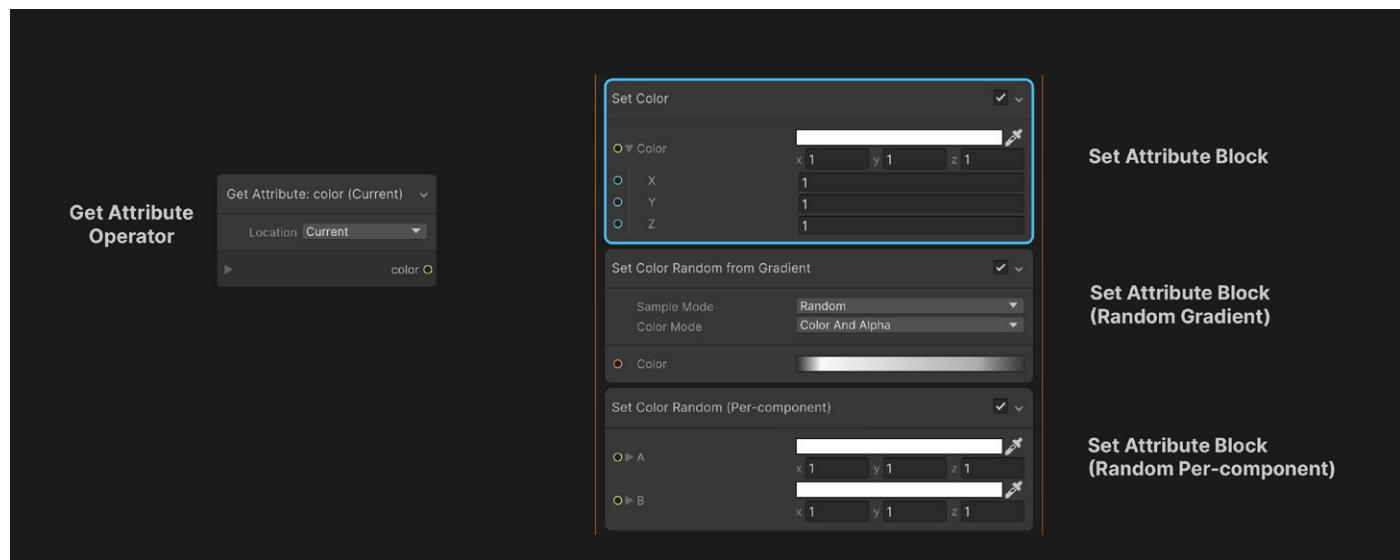
Attributes

An Attribute is a piece of data you might use within a System, such as the color of a particle, its position, or how many of them you should spawn.

Use nodes to read from or write to Attributes. In particular, use the:

- **Get Attribute Operator** to read from Attributes in the **Particle** or **ParticleStrip System**
- **Experimental Spawner Callbacks** to read from Attributes in **Spawn systems**
- **Set Attribute Block** to write values to an Attribute; either set the value of the Attribute directly or use a random mode (for example, set a **Color Attribute** with a **Random Gradient** or **Random Per-component Block**)

See the [documentation](#) for a complete list of Attributes.

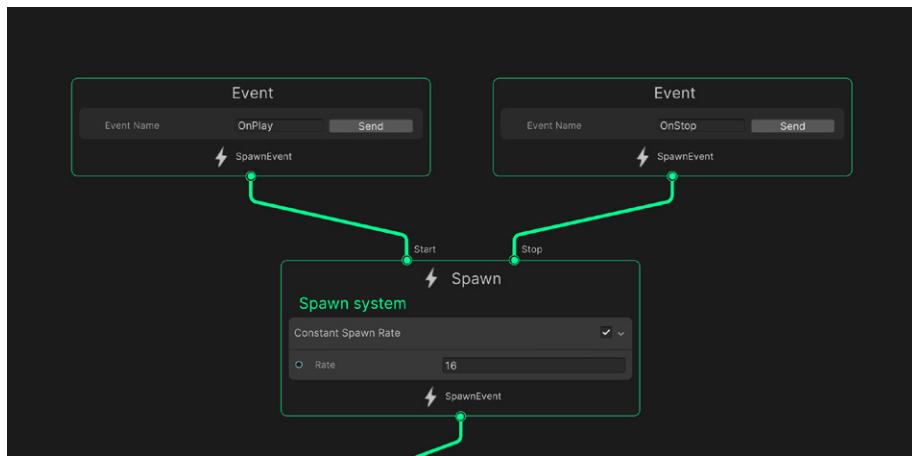


Get the Attribute with an Operator and set the Attribute with a Block.

Note: A System only stores Attributes when it needs them. In order to save memory, it does not store any unnecessary data. If you read that the VFX Graph has not stored the simulation data from an Attribute, the Attribute passes its default constant value.

Events

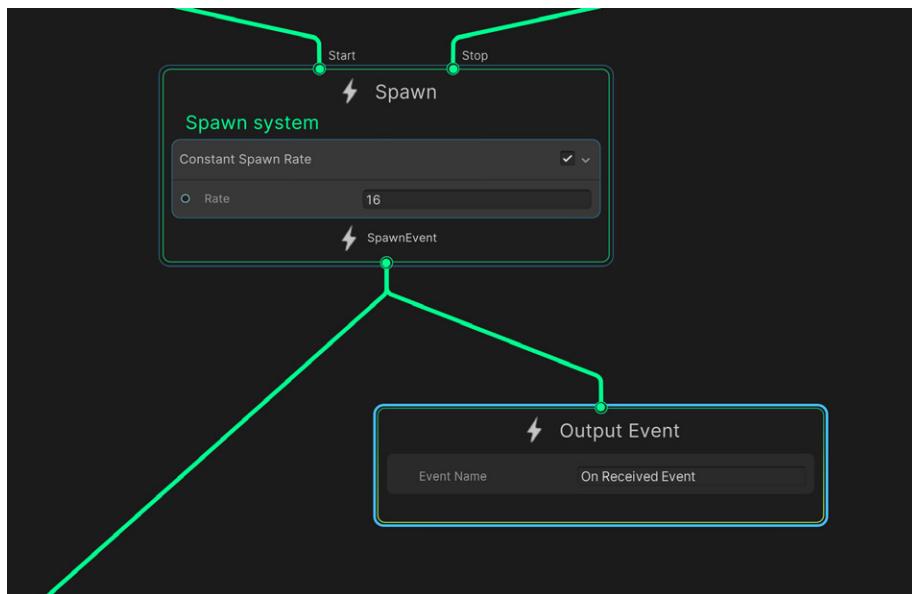
The various parts of a VFX Graph communicate with each other (and the rest of your scene) through **Events**. For example, each **Spawn Context** contains **Start** and **Stop** flow ports, which receive Events to control particle spawning.



Events control particle spawning.

When something needs to happen, external GameObjects can notify parts of your graph with the **SendEvent** method of the [C# API](#). Visual Effect components will then pass the Event as a string name or property ID.

An **Event Context** identifies an Event by its [Event string name or ID inside a graph](#). In the above example, external objects in your scene can raise an **OnPlay Event** to start a Spawn system or an **OnStop Event** to stop it.

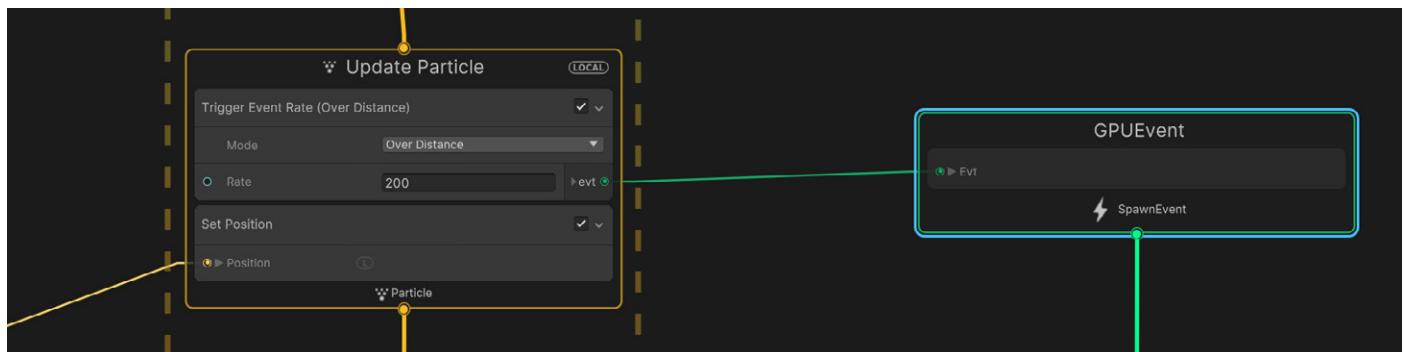


An Output Event can send messages to the scene.

You can combine an **Output Event** with an **Output Event Handler**. Output Events are useful if the initial spawning of the particles needs to drive something else in your scene. This is common for synchronizing lighting or gameplay with your visual effects.

The above example sends an **OnReceivedEvent** to a GameObject component outside of the graph. The C# script will then react accordingly to intensify a light or flame, activate a spark, etc. See the [Interactivity section](#) for more information on Output Events.

At the same time, you can use **GPU Events** to spawn particles based on other particle behavior. This way, when a particle dies in one system, you can notify another system, which creates a useful chain reaction of effects, such as a projectile particle that spawns a dust effect upon death.



A GPU Event Context receives an Event from the Trigger Event Rate Block.

These **Update Blocks** can send **GPU Event** data in the following way:

- **Trigger Event On Die:** Spawns particles on another system when a particle dies
- **Trigger Event Rate:** Spawns particles per second (or based on their velocity)
- **Trigger Event Always:** Spawns particles every frame

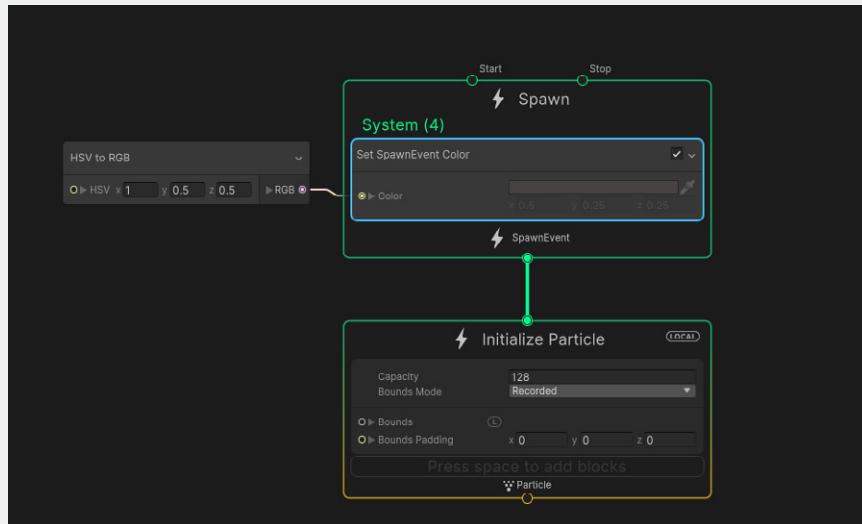
The Blocks' outputs connect to a **GPU Event Context**, which can then notify an **Initialize Context** of a dependent system. Chaining different systems together in this fashion helps you create richly detailed and complex particle effects.

The Initialize Context of the GPU Event system can also inherit Attributes available in the parent system prior to the Trigger Event. So, for instance, by inheriting its position, a new particle will appear in the same place as the original particle that spawned it.

Event Attributes

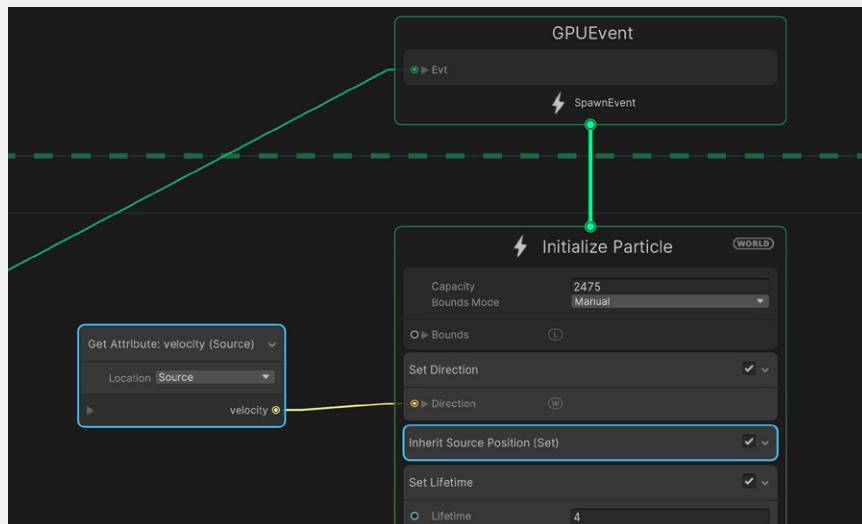
Use **Event Attribute Payloads** to pass data like 3D position or color along with the Event. These Payloads carry Attributes that implicitly travel through the graph where you can “catch” the data in an Operator or Block.

You can also read Attributes passed with **Spawn Events** or **Timeline Events**. The **Set SpawnEvent Attribute Block** modifies the Event Attribute in a Spawn Context.



Reading an Event Attribute in a Spawn Context

To catch a Payload in an Initialize Context, use **Get Source Attribute Operators** or **Inherit Attribute Blocks**.

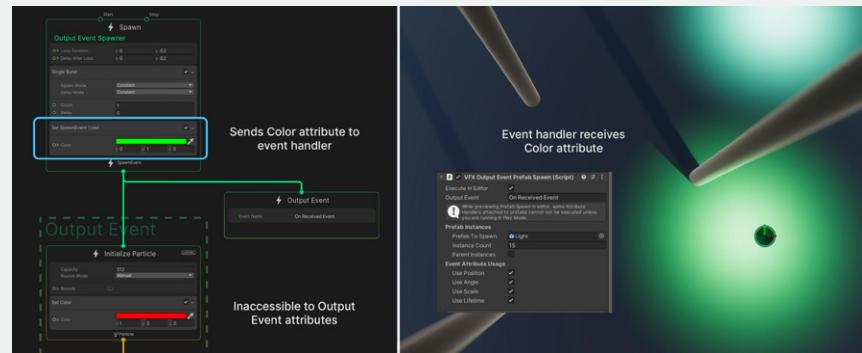


Catching an Event Payload in Initialize

However, it's important to keep these caveats in mind when using Event Attributes:

- **Regular Event Attributes** can only be read in the Initialize Context. You cannot inherit them in Update or Output. To use the Attribute in a later Context, you must inherit and set it in Initialize.
- **Output Event Attributes** only carry the initial values set in the Spawn Context. They do not catch any changes that occur later in the graph.

See [Sending Events](#) in the Visual Effect component API for more details.

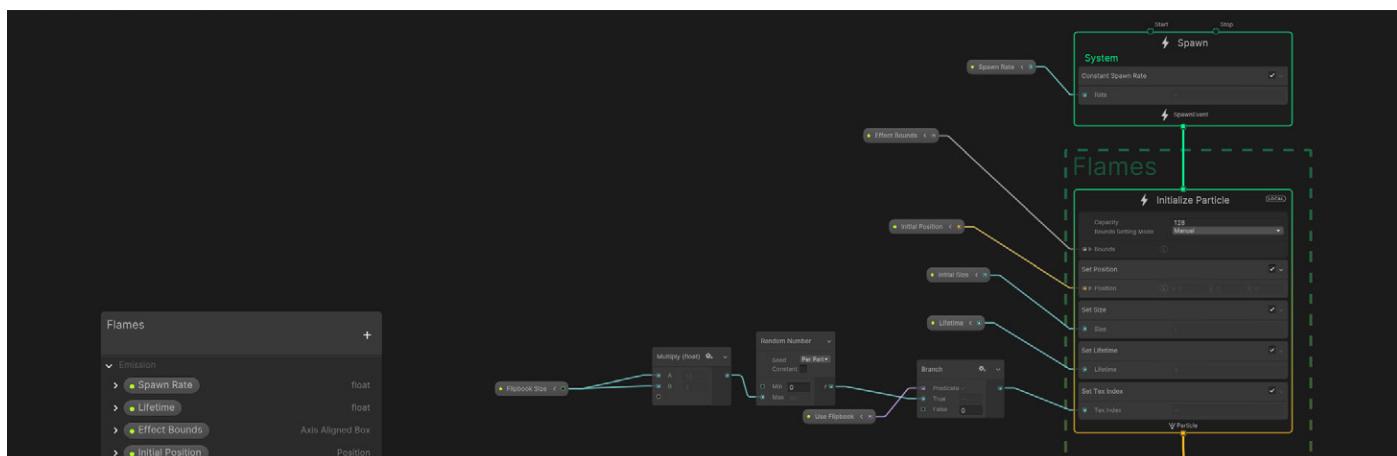


Output Event Attributes carry values from the Spawn Context.

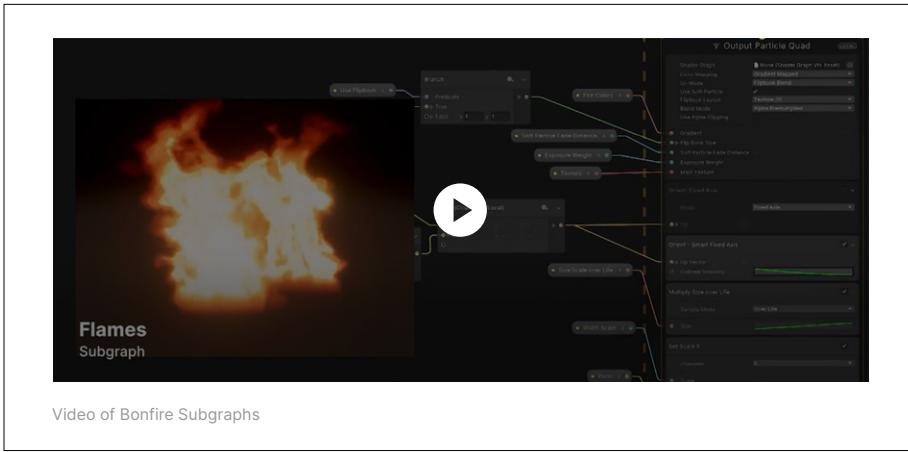
Exploring VFX sample content

A VFX Graph is more than the sum of its parts. It requires a solid understanding of how to apply each Node and Operator, along with the ways they can work together.

The VFX Graph [Additions](#) in the Package Manager demonstrate several simple graphs, making them a great starting point for learning how to manage particles. In the example below, you can see how the Smoke, Flames, and Sparks build up to form the Bonfire effect:

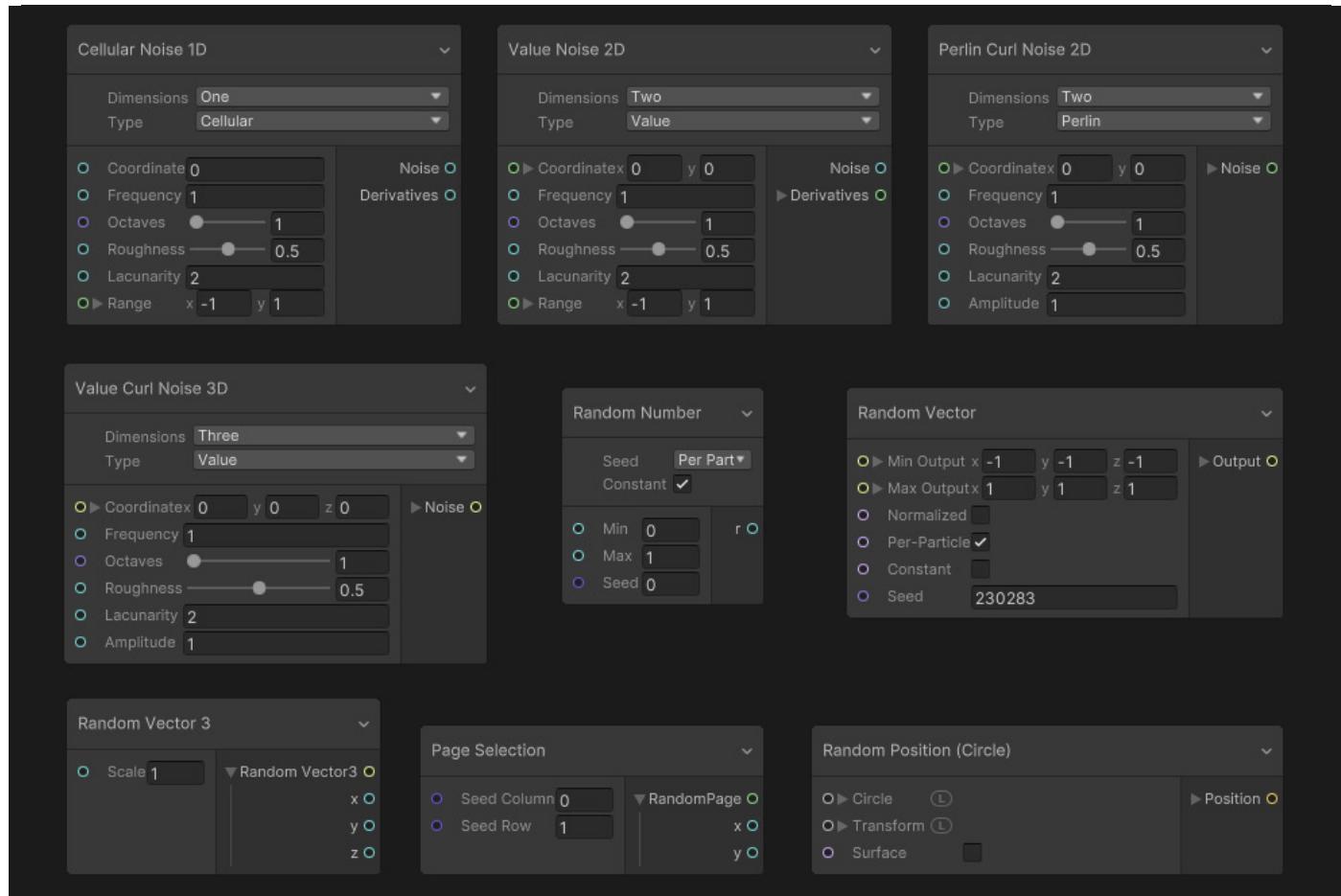


The Flames effect shows a basic graph. [Download this example](#).



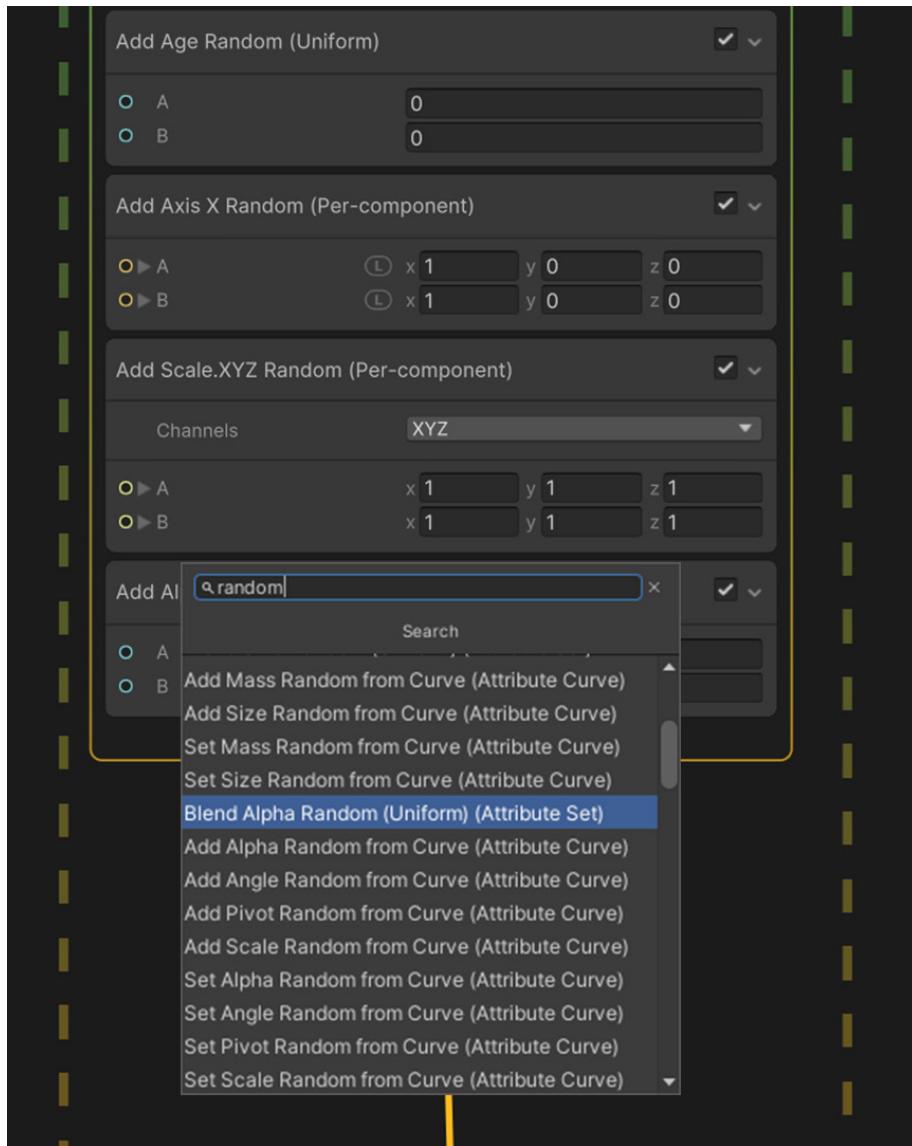
You'll encounter some common Blocks and Operators as you explore the samples provided:

- **Noise and Random Operators:** Procedural Noise helps reduce the “machine-like” look of your rendered imagery. The VFX Graph provides several Operators that you can use for one-, two-, and three-dimensional [Noise](#) and [Randomness](#).



Noise and Random Operators

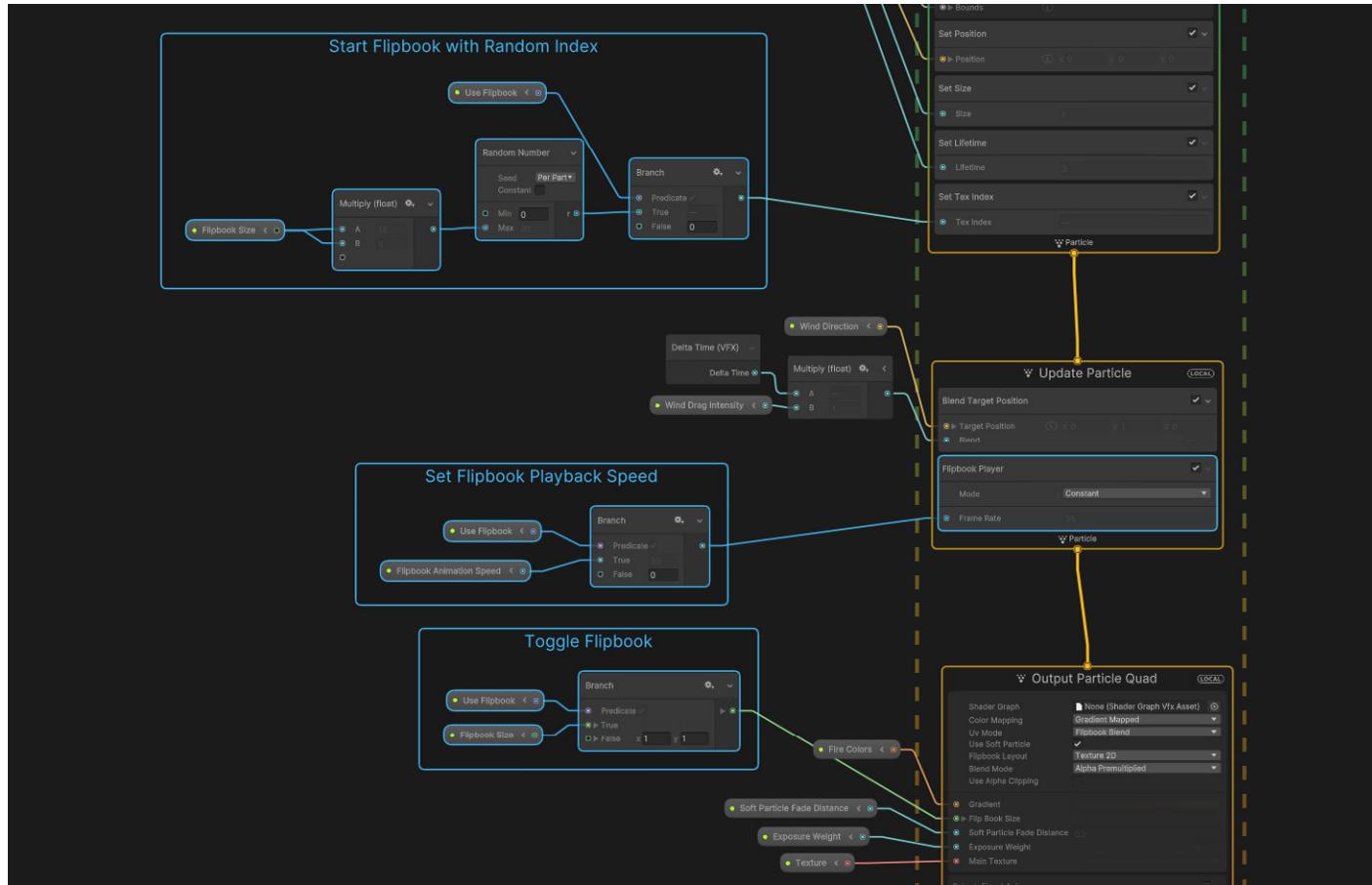
- **Attribute Blocks:** These similarly include the option of applying Randomness in various modes. They can vary slightly per Attribute, so experiment with them to familiarize yourself with their behavior.



Randomness Blocks

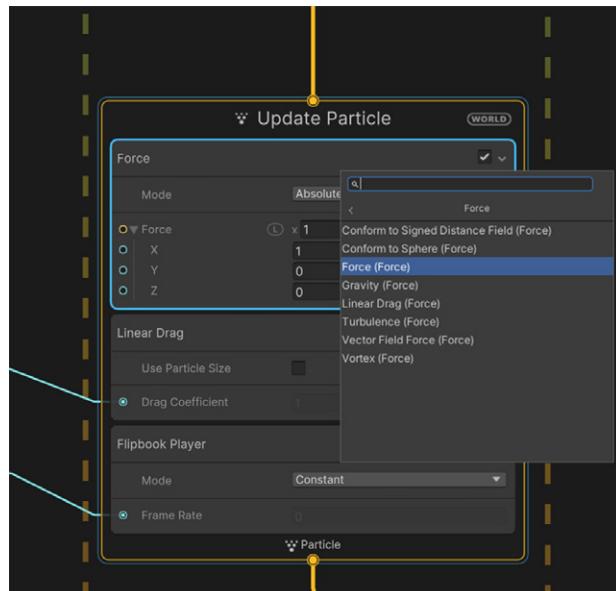
- **Flipbooks:** An animated texture can do wonders to make your effects believable, as you can see in the Smoke and Flames samples. Generate these from an external **Digital Content Creation** (DCC) tool or from within Unity. Use Operators to manage the [Flipbook Block](#).

For more information on creating your own flipbooks within Unity, check out the [Image Sequencer](#) in the [VFXToolbox section](#).



Flipbook Nodes. [Download this example.](#)

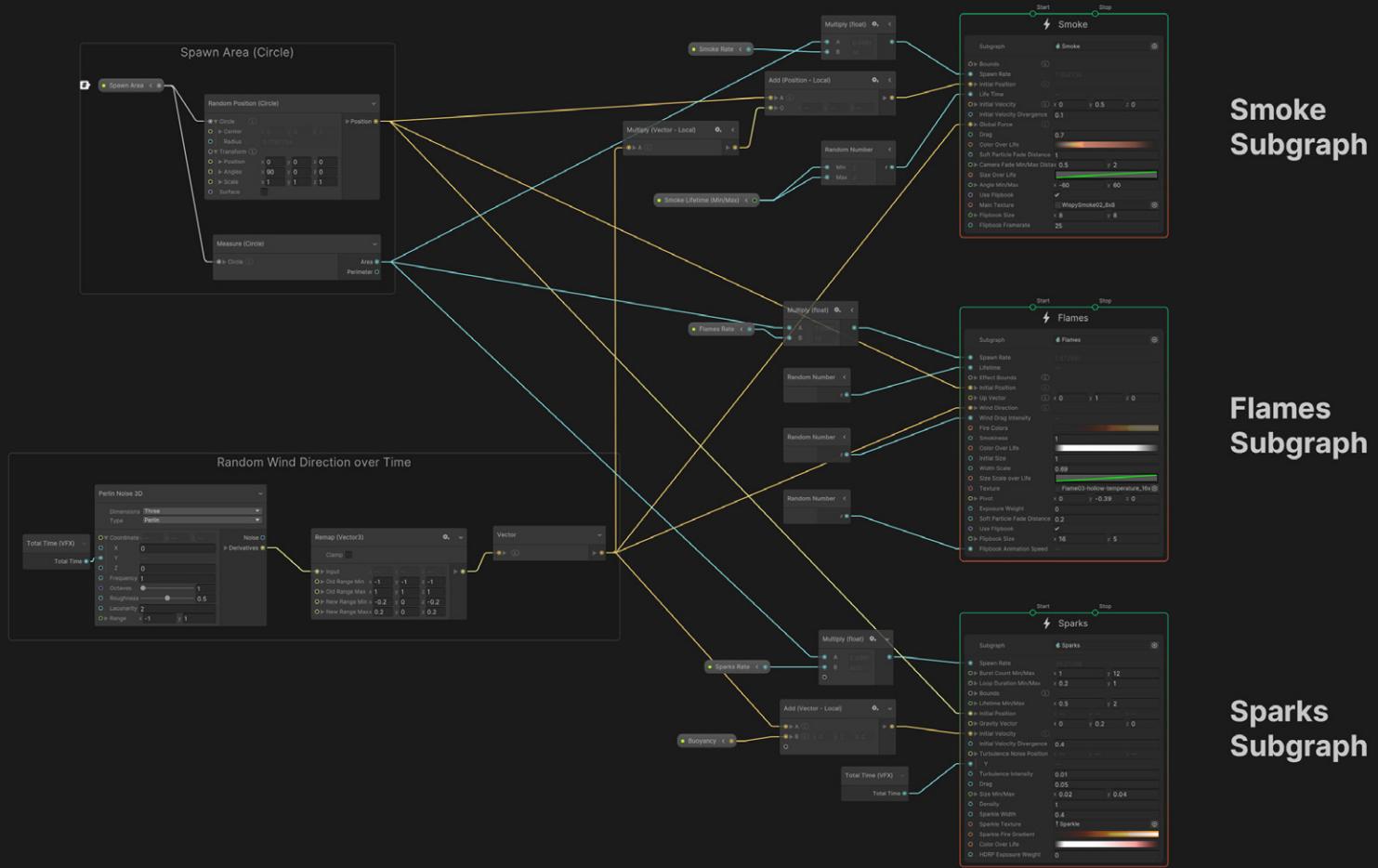
- **Physics: Forces, Collisions, and Drag** are essential to making particles simulate natural phenomena. But don't be afraid to push the boundaries of what's real. As the artist, you get to decide what looks just right.



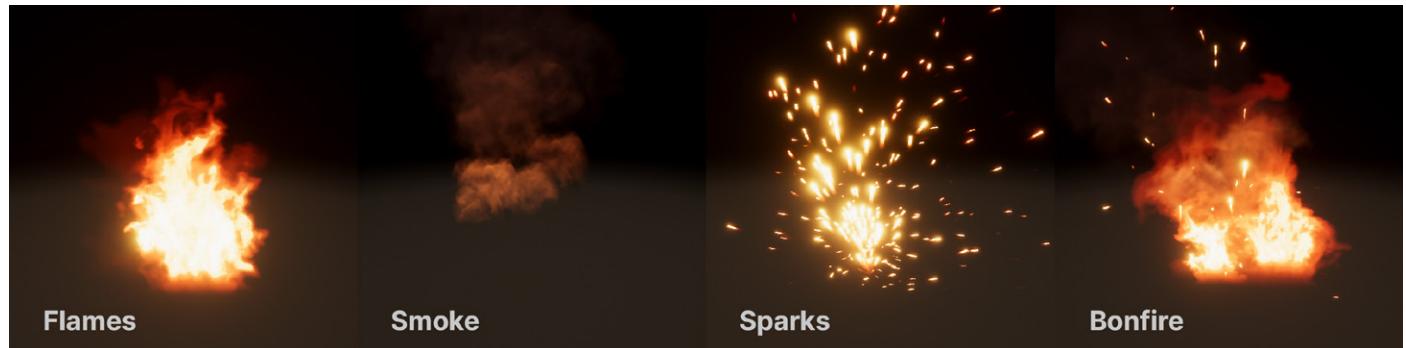
Physics Blocks

- **Subgraphs:** The Smoke, Flames, and Sparks are [Subgraphs](#). They are the parts of a VFX Graph that can be saved as an asset for later reuse.

Splitting the main elements into smaller parts makes the Bonfire graph more readable. So if you need to make a new explosion effect somewhere else in your application, for example, you can now deploy it by dragging and dropping it into another graph. This works because the Subgraph is an asset.



The Bonfire graph uses three Subgraphs



Individual Subgraph elements combine to form the Bonfire.

For a breakdown of how to construct the Bonfire graph, among other effects, watch [these community videos](#) from Thomas Iché, a senior VFX and technical artist involved in creating the samples and the Unity Spaceship Demo.

More resources

Once you're familiar with the basic workings of a VFX Graph, try building a few effects from scratch. Start with a simple system for [falling snow](#), then play around with [fire, smoke, and mist](#).

The following videos offer an introduction to several effects:

- [Making snow with the VFX Graph](#)
- [Creating fire, smoke, and mist effects with the VFX Graph in Unity](#)
- [Rendering particles with the VFX Graph in Unity](#)
- [Multilayered effects with the VFX Graph in Unity](#)
- [Hardspace: Shipbreaker Tech Talk: Explosions with the VFX Graph](#)
- [Real-time VFX workflows in *The Heretic*](#)

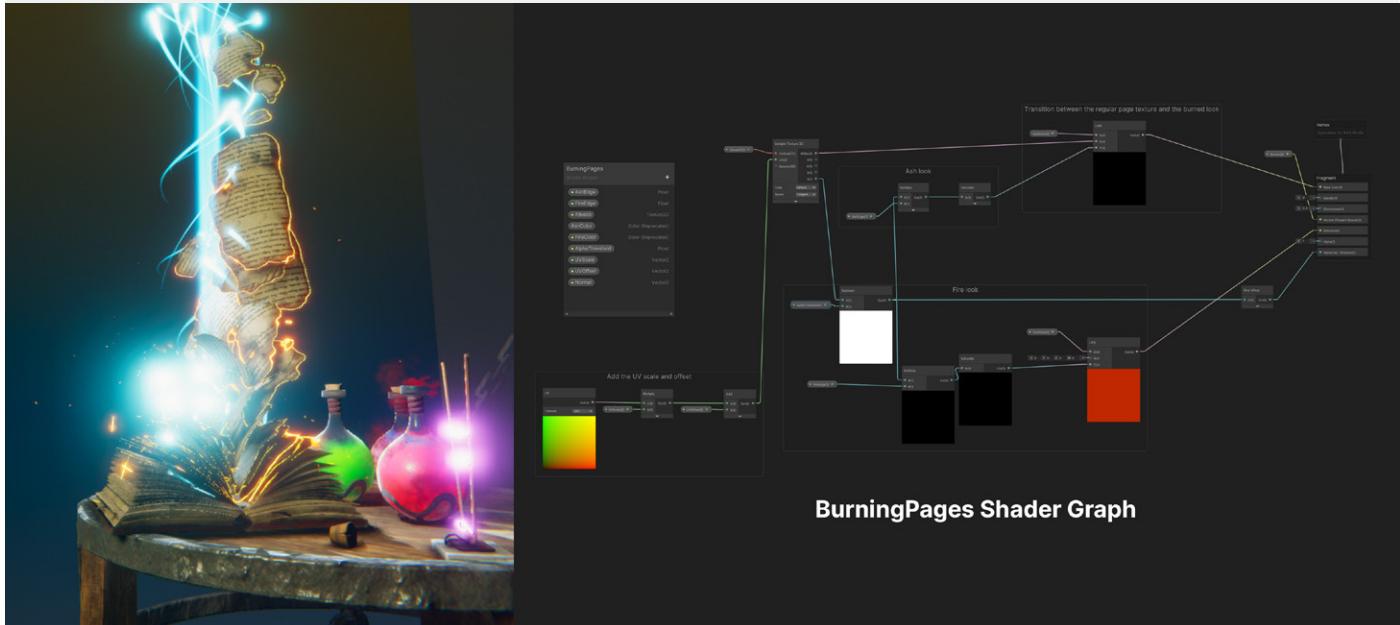
Additional references

As you get more comfortable with the VFX Graph, you can dive deeper to discover its nuances. Keep these pages handy when you need to reference specific Node or Operator functionalities:

- The [Node Library](#) describes every Context, Block, and Operator in the VFX Graph.
- The [Standard Attribute Reference](#) offers a comprehensive list of all common Attributes.
- The [VFX Type Reference](#) lists Data types used in the VFX Graph.

Shader Graph integration

[Shader Graph](#) enables technical artists to build custom shading with a graph network. Although it's separate from the VFX Graph, writing shaders is often intertwined with creating effects.



Specialized shaders complement your visual effects.

In the Magic Book sample, for instance, you can write a specialized shader to make the flying pages magically dissolve into embers.

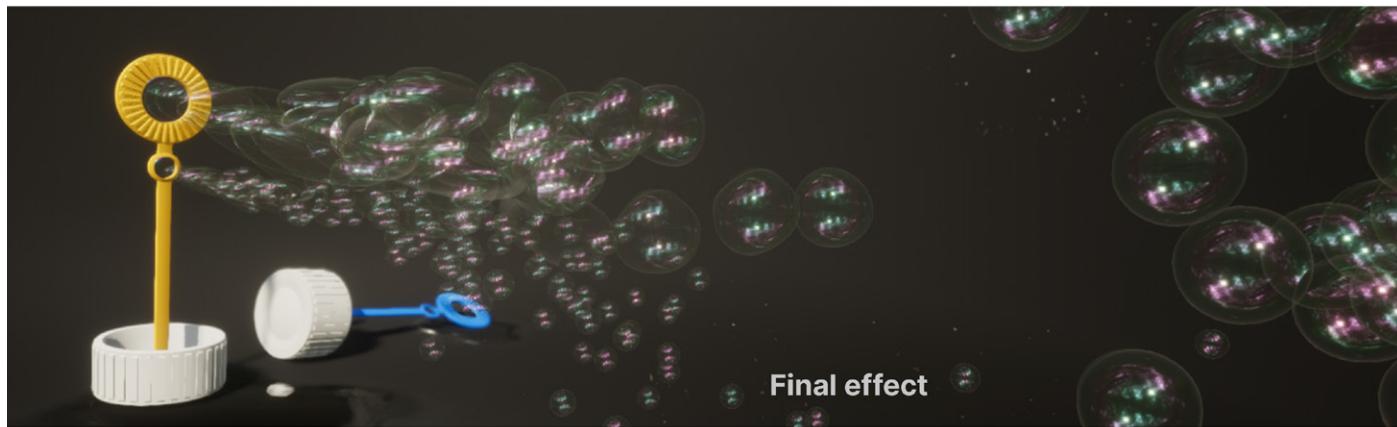
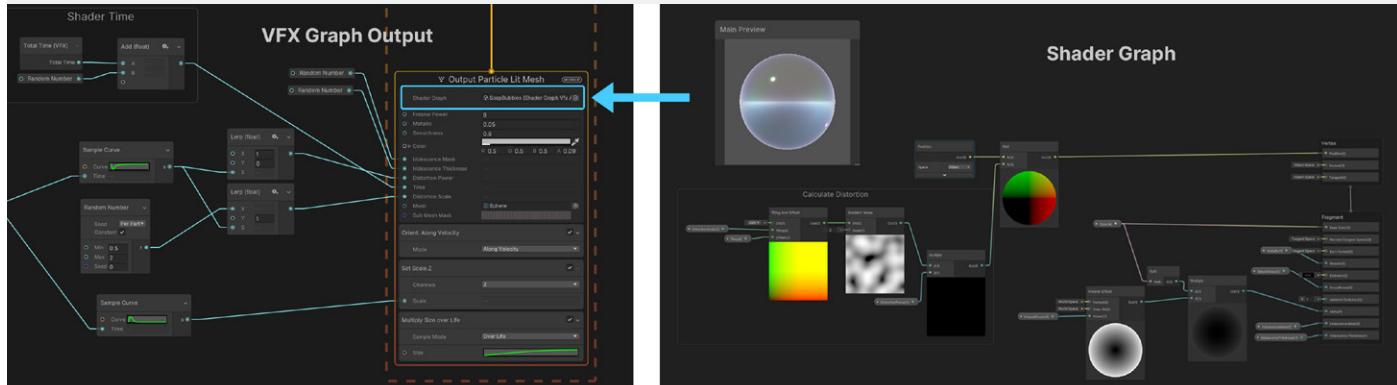
With Shader Graph, you can:

- Warp and animate UVs
 - Alter surface appearance procedurally
 - Add fullscreen image filters
 - Change an object's surface based on information like world location, normals, distance from the Camera, and more

To use Shader Graph, you don't need to learn a specialized language like HLSL. The visual interface of Shader Graph gives instant feedback, helping creators iterate more quickly.

The VFX Graph now includes smoother integration with Shader Graph in HDRP. When creating a new Shader Graph, check the **Support VFX Graph** option and then assign it to the corresponding field in the VFX Graph's Output. This lets you keep your shading and VFX parameters together, further facilitating your visual adjustments.

Even more, you can work with the full range of HDRP materials. From fur to fabric, refine your visuals directly within the VFX Graph.



Shader Graph integration with HDRP

VISUAL EFFECTS BY EXAMPLE





One of the VFX Graph Samples available in the repository

Once you understand the fundamentals of the VFX Graph, challenge yourself to craft more complex graphs. A number of example projects are available to help you better prepare for problems you might encounter during production.

These samples run the gamut of what's possible for your visuals. From ambient smoke and fire to fully scripted, AAA cinematic gameplay, take your time to explore.

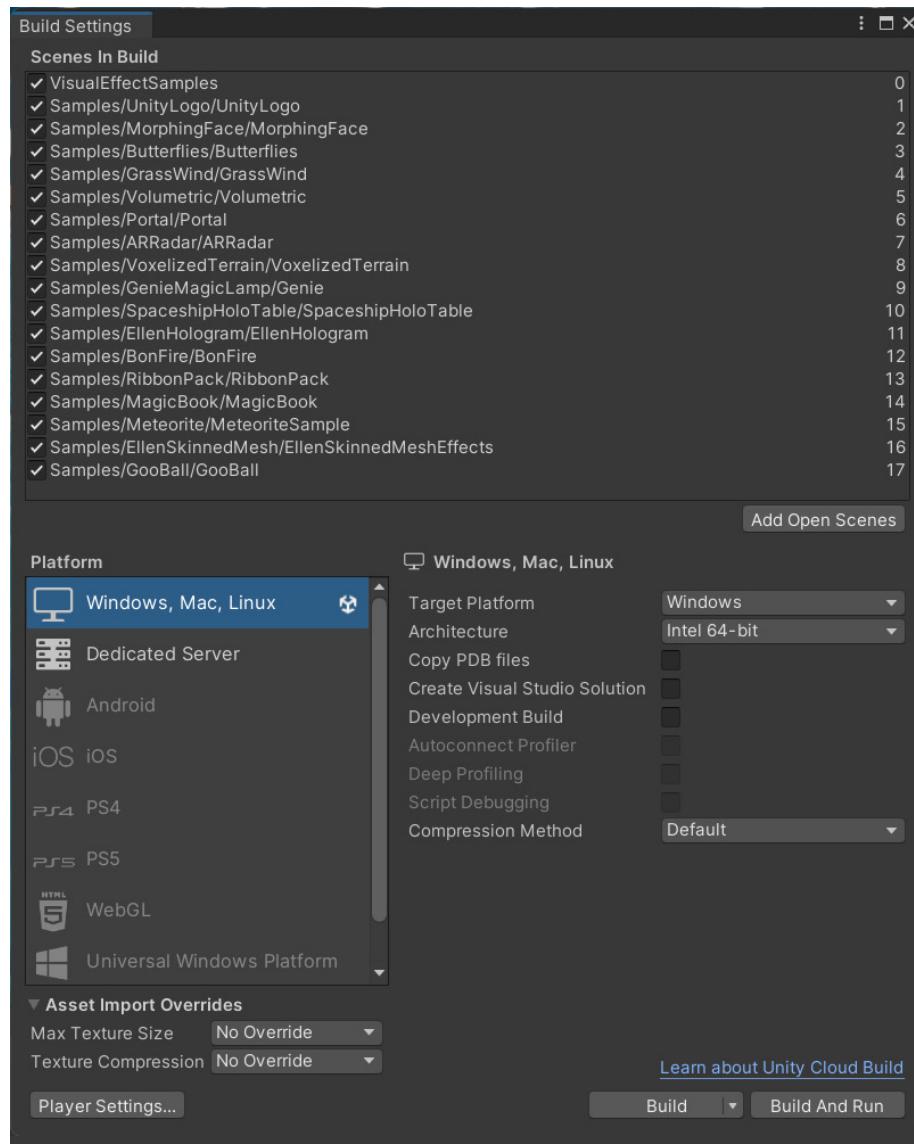
The VFX Graph Samples (HDRP)

The [Visual Effect Graph Samples](#) highlight different scenarios that involve the VFX Graph. You can view some of the example graphs as case studies, so it's helpful to download and keep the project as a working reference.

Note: The samples only support HDRP and are therefore incompatible with URP.

Go to the [Release](#) tab to find snapshots of these samples, as well as links to prebuilt binaries. Alternatively, you can clone the entire repository.

Each sample appears in a subdirectory within the project's **Assets/Samples** folder. The main **VisualEffectsSample** scene lives at the root.



The Build Settings

If you need to build a player, ensure this main scene is set to index zero within the **Build Settings**. Then add all other scenes you plan to cycle afterward.

Each scene part of the VFX Graph Samples showcases a unique effect. Let's take a closer look at some of them.

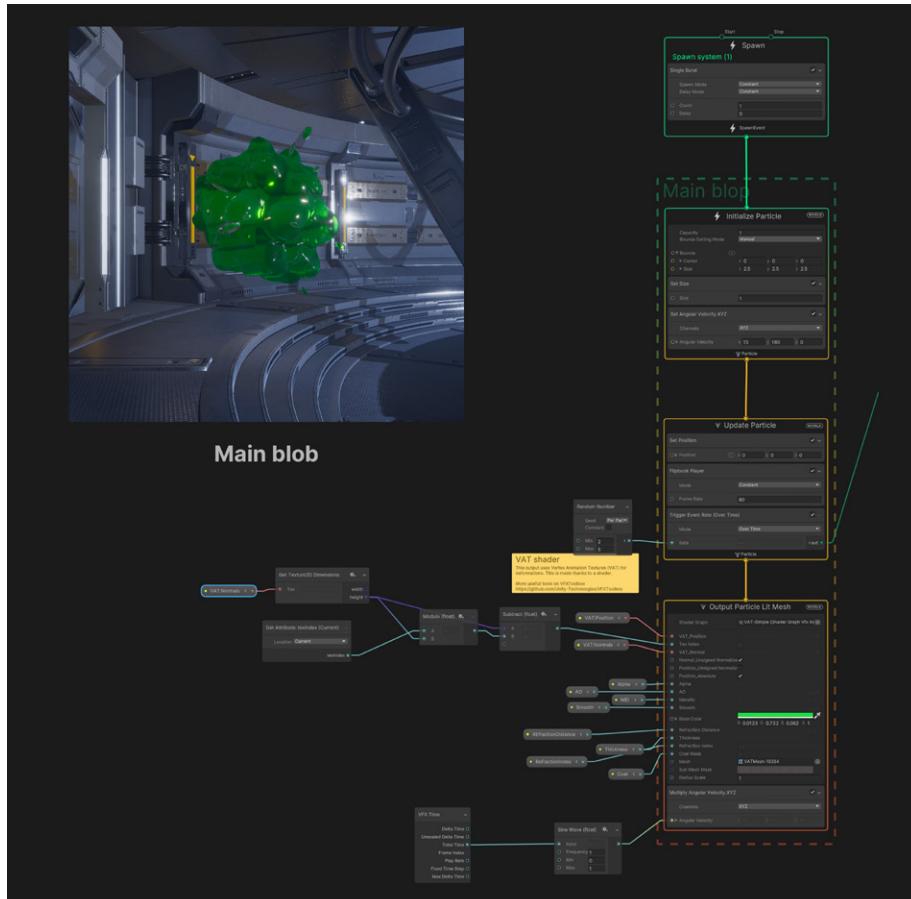
GooBall

Have you ever wanted to splatter paint in Unity? This playful sci-fi demo incorporates decals that simulate gooey substances interacting with their surrounding environment. Like other production examples, this multilayered effect leverages several Systems to achieve its final look.



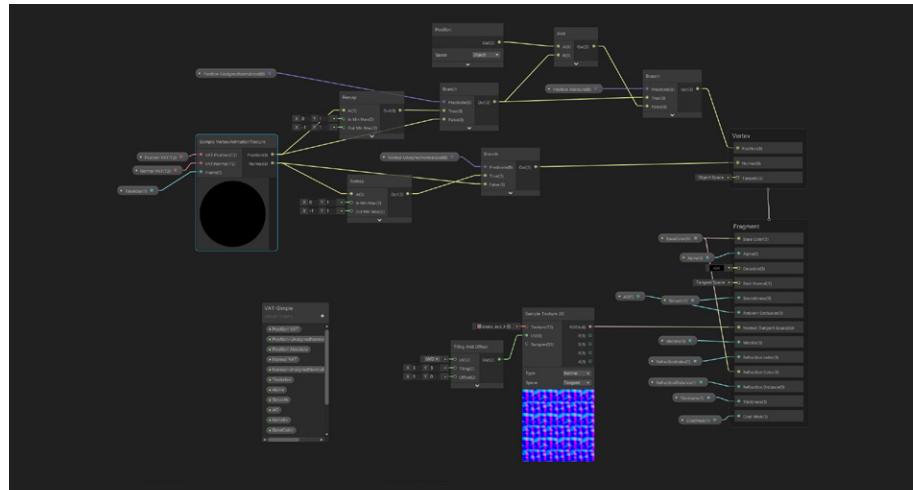
The GooBall effect

The center blob starts with a **Vertex Animation Texture** (VAT) Shader Graph that creates the impression of fluid movement. A scrolling texture sheet in the shader constantly undulates the mesh's 3D points, which is ideal for sci-fi goo.



The main blob

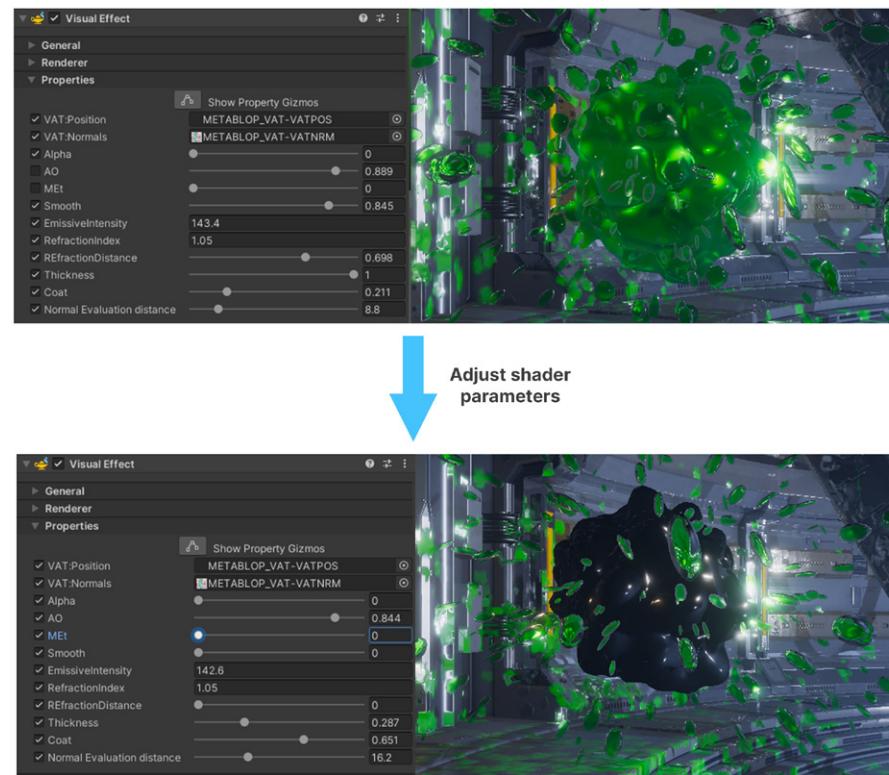
Because the Shader Graph connects directly to the [Output Particle Lit Mesh Context](#), there are some input ports that influence the shader. Here's the VAT Shader Graph at a glance:



The VAT Shader Graph. [Download this example](#).

In addition to the vertex motion, the Shader Graph also creates the appearance of a transparent, green glass-like material. This makes up the goo's look.

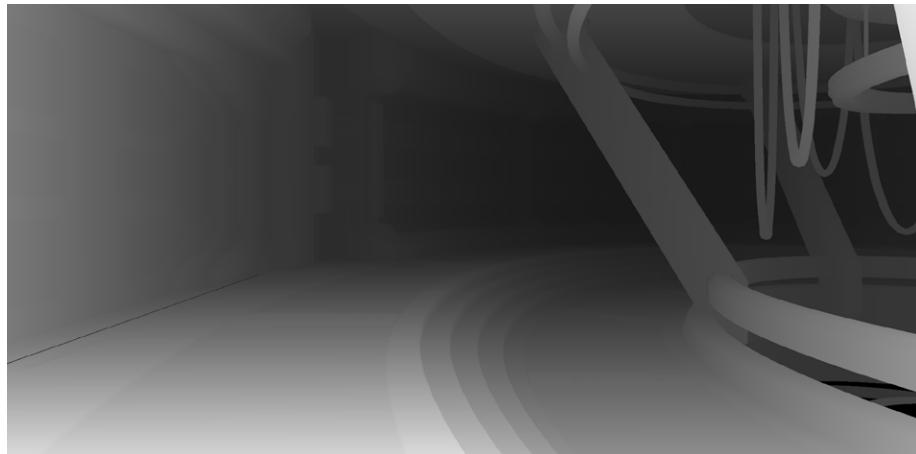
Shader Graph is [closely integrated](#) with the VFX Graph. In this example, you can edit some of the GooBall's shading parameters from the VFX Graph's Blackboard. Go to the **Inspector** to access a unified set of sliders and fields for tweaking the goo's appearance.



Adjust the Blackboard properties to modify the goo's appearance

The main blob triggers a GPU Event to spawn some particles on the surface of a sphere. Downward force is applied with the proper maps and some droplets of goo drip periodically.

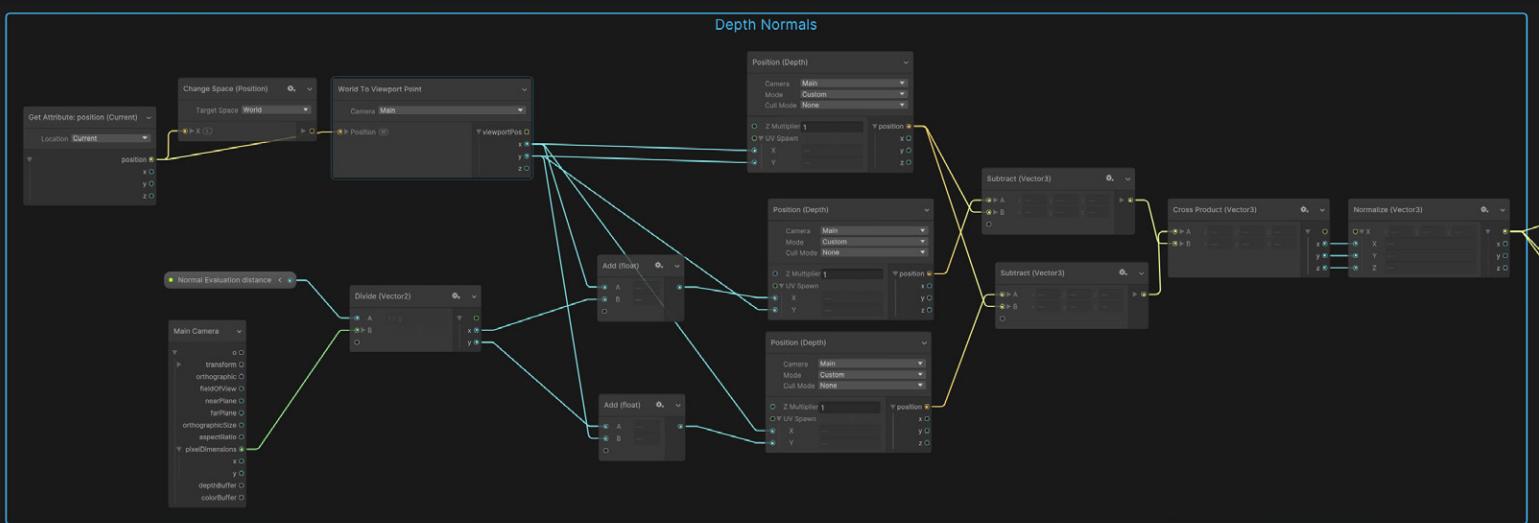
Even though the VFX Graph doesn't directly interact with a Collider on the floor, you can approximate the environment with the Camera's **depth buffer**. This works if precision isn't a concern.



Visualizing the Camera's depth buffer

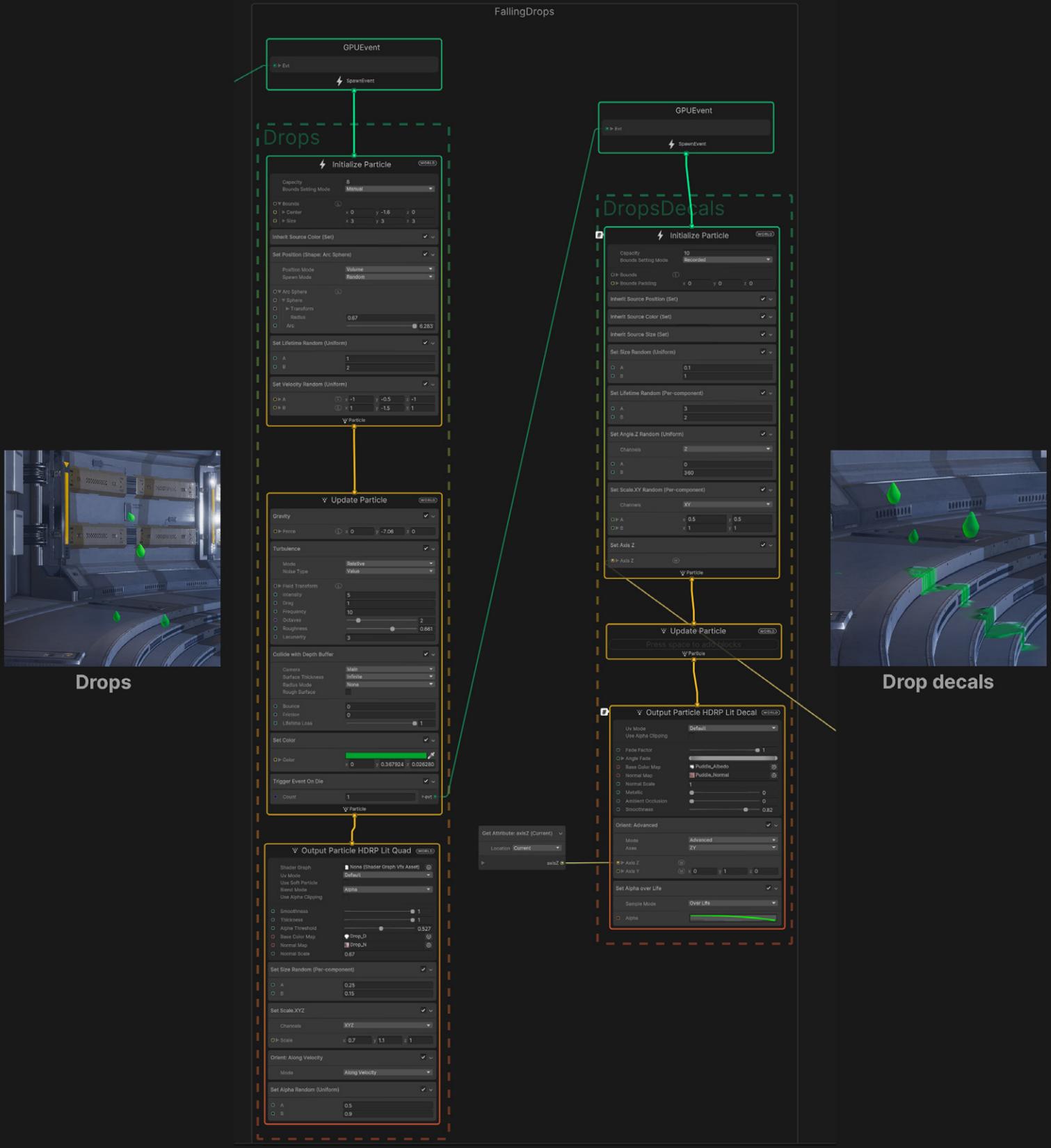
As drops hit the depth buffer, they trigger a GPU Event. This Event passes the position, color, and size Attributes to a separate System for handling the decals.

To orient the splats correctly against the geometry, the buffer's depth normals are calculated using something like this:



Depth normals. [Download this example.](#)

They then pass into the Z axis Attribute so that all the decals face the right way. With the new puddle texture created, each decal splats convincingly across the uneven surface of the floor.



Drops fall off the surface and leave decals.

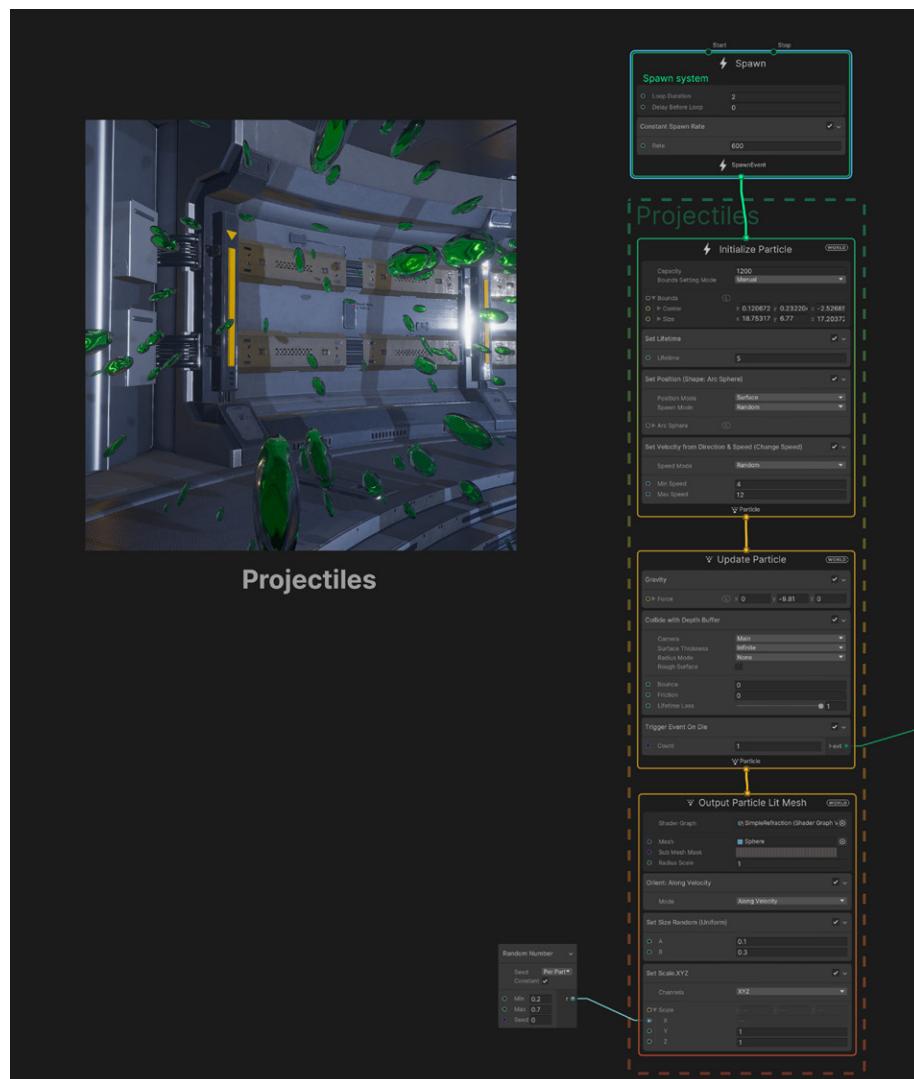
The scene gets more chaotic with extra projectiles that fire off randomly from the spherical surface. This reiterates the power of VFX.

Physics-based effects

The VFX Graph can compute complex simulations and read frame buffers. However, it does not support bringing particle data into C# or connecting to the underlying physics system.

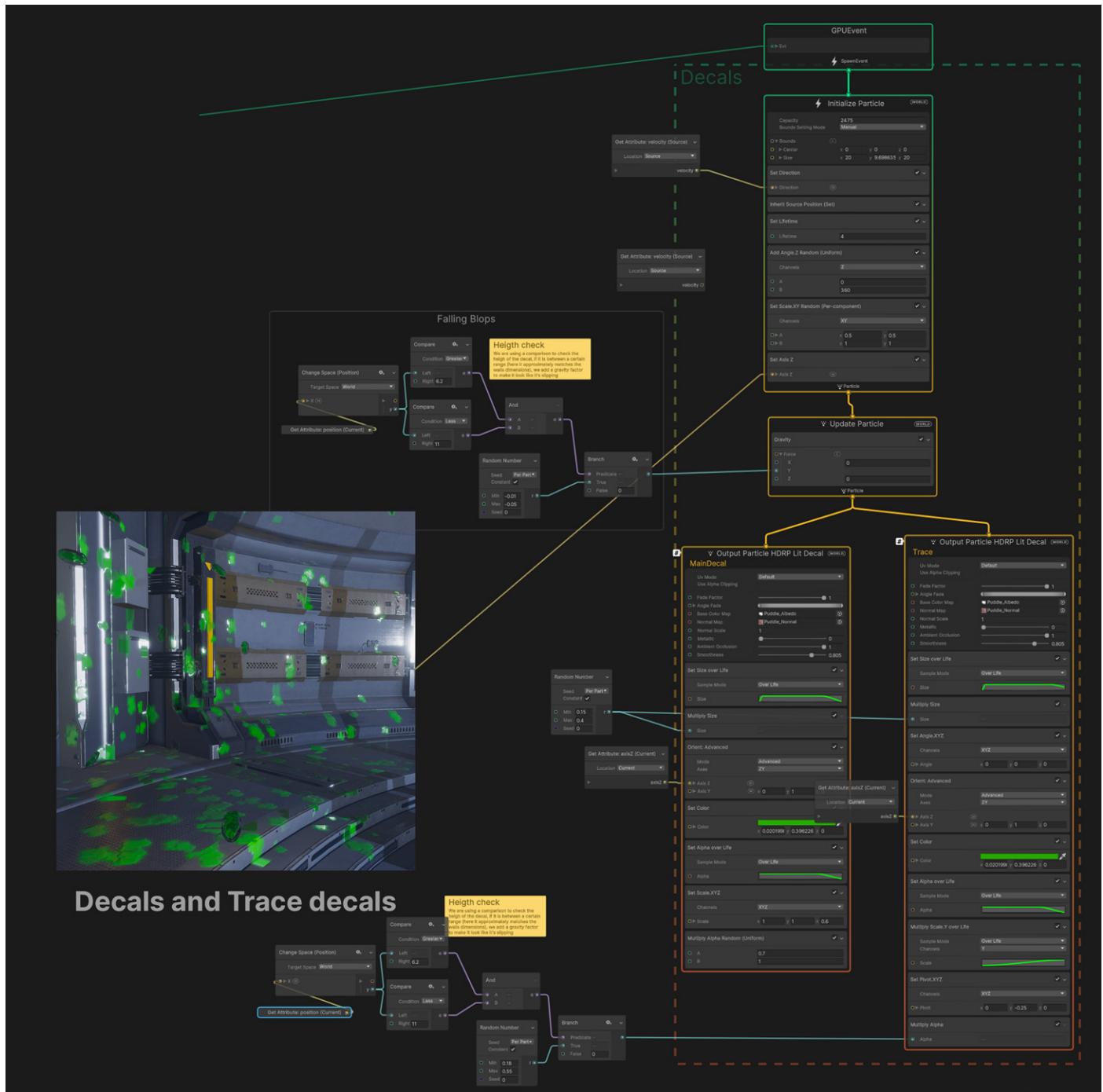
That's why you'll need to use some workarounds to create physics-based effects, such as:

- The depth buffer
- Primitive representations (sphere, box, torus, etc.)
- 3D textures like **Signed Distance Fields**, **Point Caches**, or **Vector Fields**



Projectiles firing chaotically from the GooBall

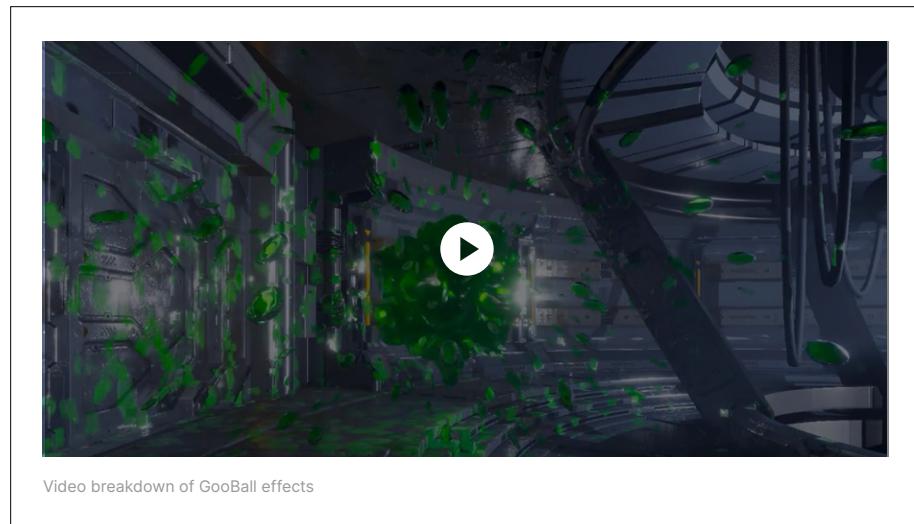
As the projectile collides with the room geometry, a similar technique calculates the splats. This time, though, you should also check if a second Trace decal falls within the height range of the walls. If it does, the Y scale will animate slightly, completing the illusion of the slime slipping down the smooth, metallic surfaces.



Splat and Trace decals complete the effect.

Even if you're not aiming to reproduce this exact effect, the GooBall scene shows you how to:

- Incorporate Shader Graph into your Output Context
- Use GPU Events to trigger other Systems in the same graph
- Apply decals over your environment using depth normals



Video breakdown of GooBall effects

The Ribbon Pack

This abstract effect demonstrates the use of **Particle Strips** – that is, chains of particles rendered as lines or strips of quads.

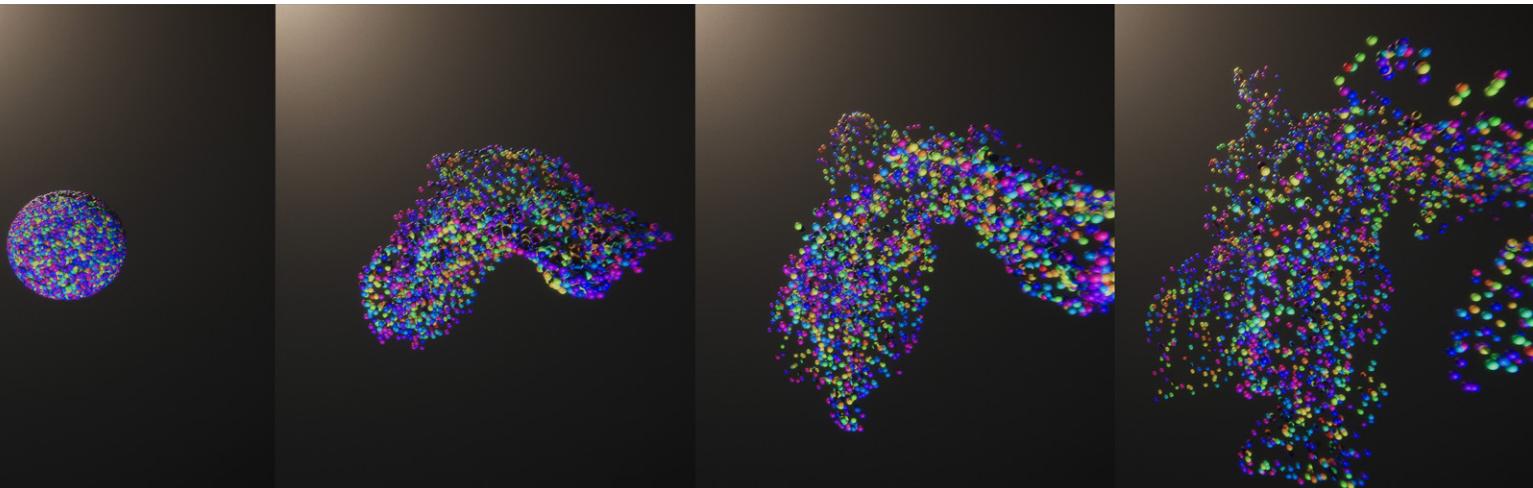
Often, they can simulate animated trails. Guide them with other particles using GPU Events, and notice how every point of a trail evolves independently, allowing you to apply wind, force, and turbulence.



The Ribbon Pack features Particle Strips.

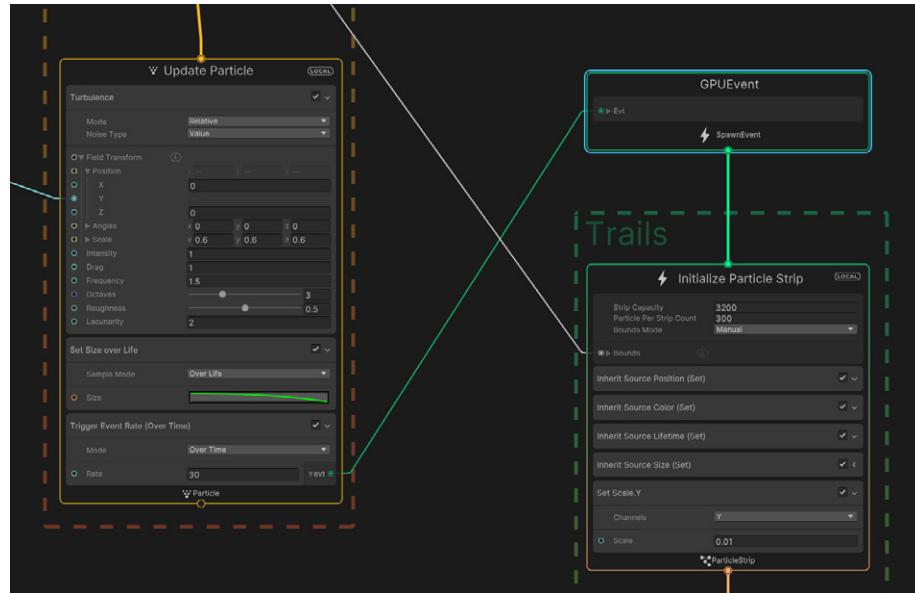
The Ribbon Pack graph spawns multicolored particles on the surface of a spherical arc. Use **Noise** to add some organic motion, or modify the available **Blocks** to customize each Particle Strip's texture mapping, spawning, and orientation.

If you were to render this effect conventionally with mesh particles, it would resemble something like this:



Rendering the Ribbon Pack without Particle Strips

Instead, the first System is hidden and doesn't appear onscreen. A **Trigger Event Rate Block** is used to invoke a GPU Event.



The Trigger Event Rate Block invokes a GPU Event.

This sends a message to a series of Contexts that initialize, update, and render the Particle Strips. The result is a prismatic tangle of cables or fibers.



The many ribbons

Particle Strips have numerous applications; think of magical streaks, weapon trails, and wires, to name a few. The Magic Book sample uses Particle Strips for the trails swirling around each beam.



Particle Strip trails in the Magic Book sample

They also stand in for blades of grass in the Meteorite sample, discussed in the section below.

Meteorite sample

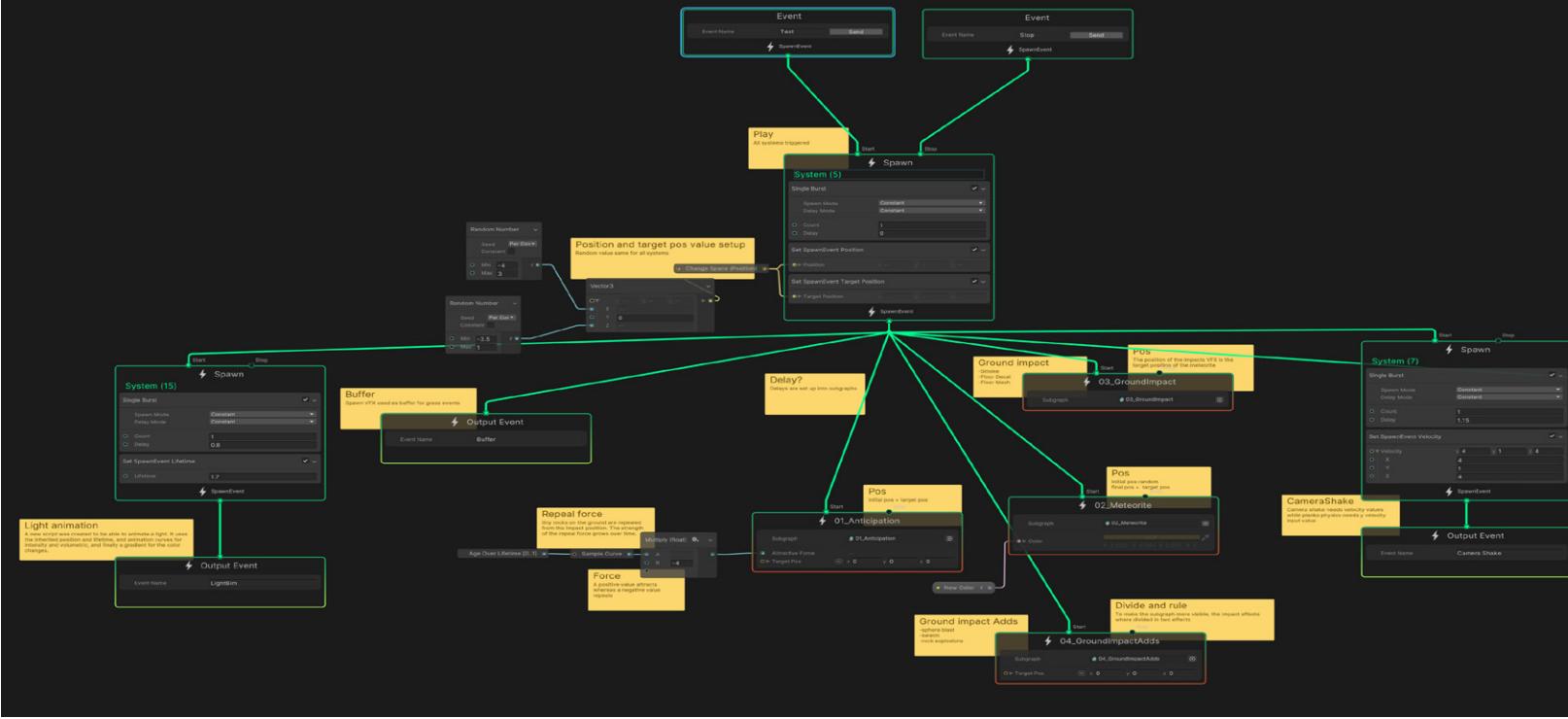
The Meteorite sample combines several effects to accentuate the impact of a meteor crashing into the earth. A **MeteoriteControl** script on the **Timeline** object listens for your keypresses or mouse clicks and then activates the meteorite effect.

In this scene, the ground and surrounding trees react to the blast. But don't worry, no VFX critters were harmed in the making of this effect.



The Meteorite sample

Here, a single graph called **MeteoriteMain** drives several others. The graph itself is neatly organized:



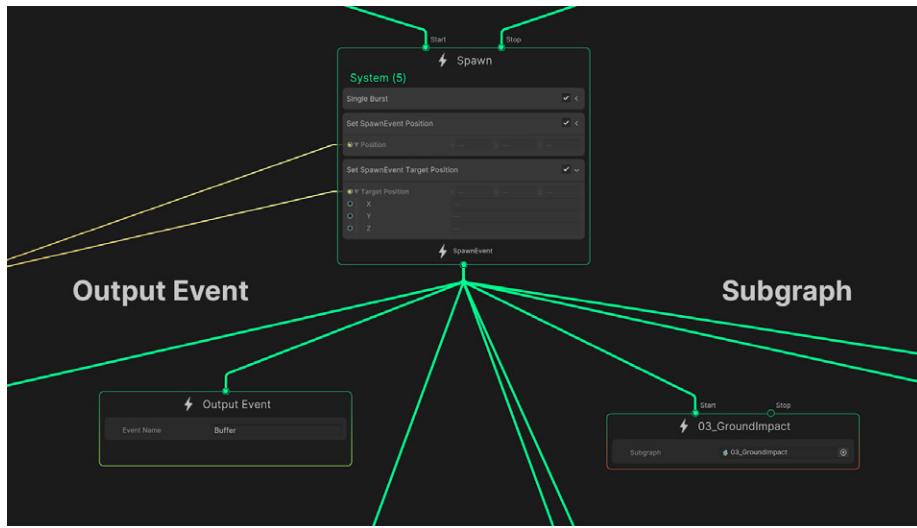
The MeteoriteMain graph

A chain reaction of effects plays every time a meteor drops from the sky. It consists of a central Spawn Context that triggers many other effects:

- Vibrating rocks and a light effect anticipating the meteor
- The crashing meteor, along with smoke trails
- A burst of animated light to jolt your viewers on impact
- Rigidbody “planks” and debris that interact with the ground
- Camera shake to further evoke the force of impact

Note: The MeteoriteMain graph looks relatively clean because much of it is broken into Subgraphs. The Spawn Event plugs directly into many of their Start ports. Drill down into each individual Subgraph to see its specific implementation.

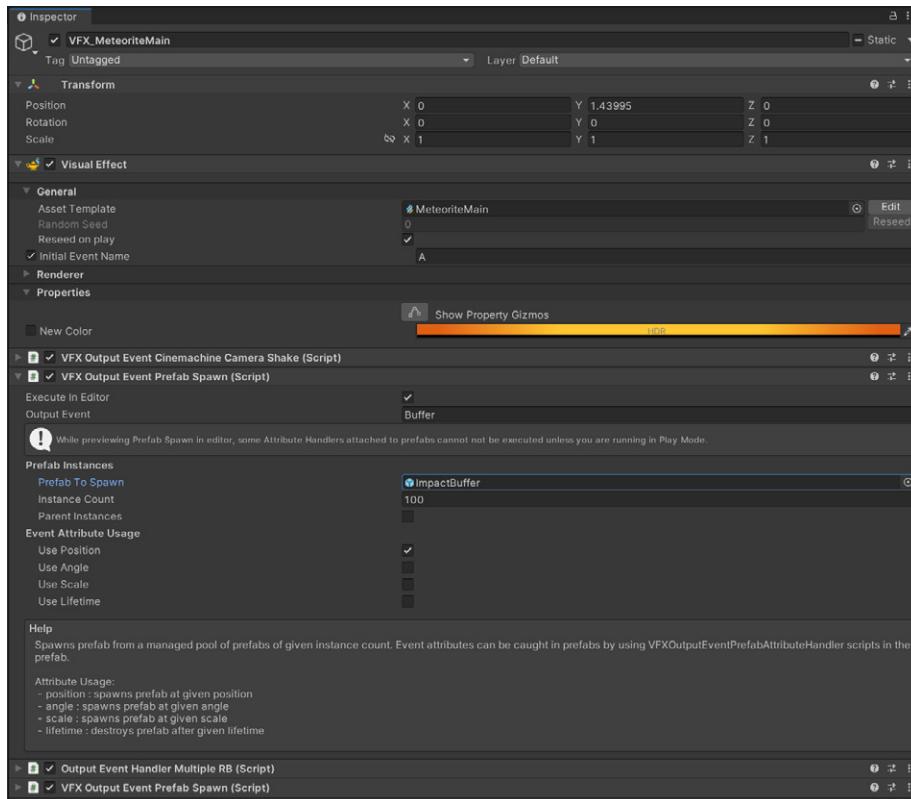
This structure relegates a number of the details into smaller, more manageable parts. It makes the graphs easier to navigate, so you won’t have to wade through a confusing web of nodes.



The Spawn Event plugs into the Subgraphs and Output Events.

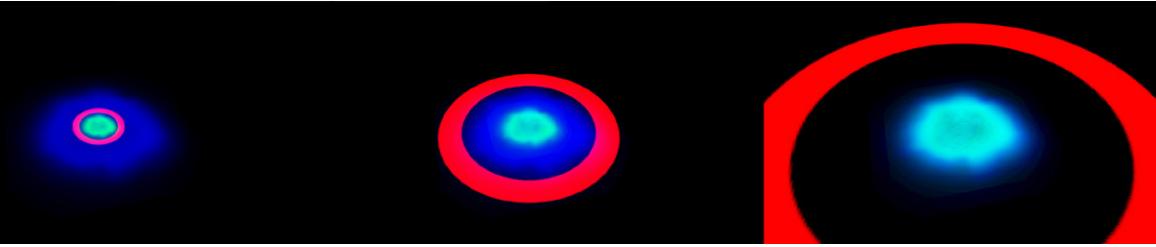
Output Events are used to communicate with other components outside of the graph. In particular, the light animation, camera shake, and plank debris have their own Output Events.

If you select the GameObject called **VFX_MeteoriteMain**, you'll see various Output Event Handler scripts that receive these events and respond accordingly.



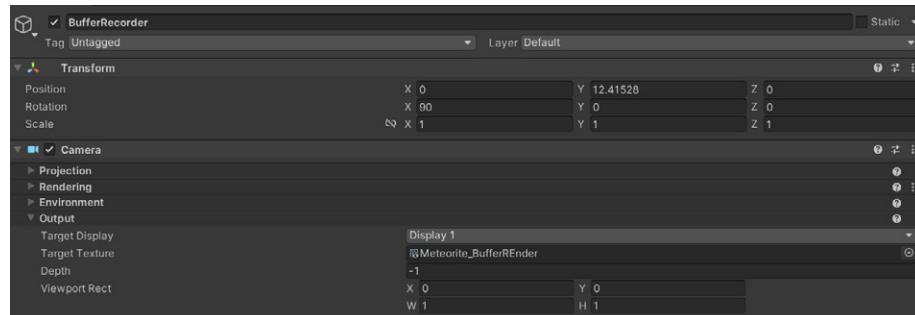
Output Event Handlers on the GameObject

Meanwhile, the graph's **Buffer Output Event** spawns a Prefab called **ImpactBuffer**, which has its own VFX Graph. It animates ground decals, separated into red, blue, and green color channels. The ImpactBuffer effect looks something like this:



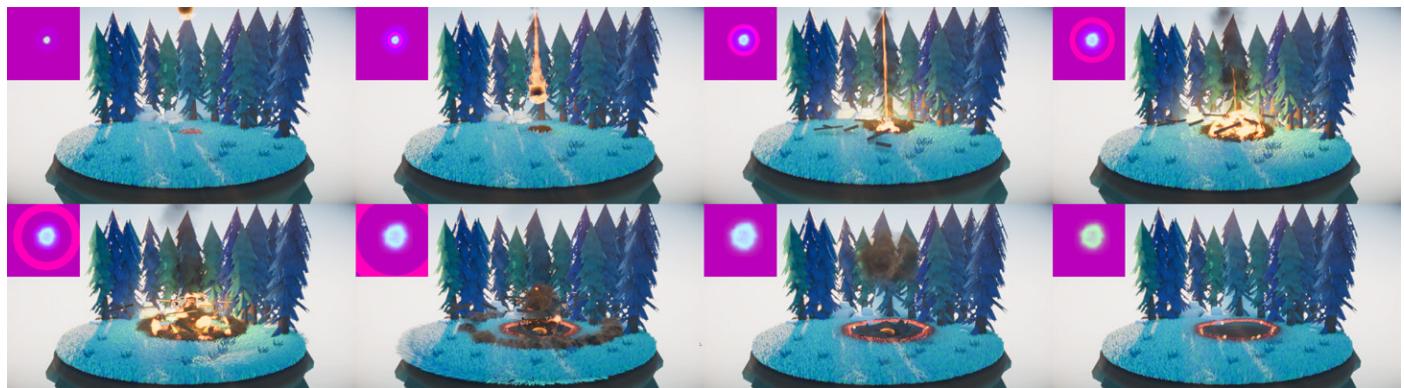
The ImpactBuffer Prefab in red, blue, and green

You only need a 2D recording of it. A separate camera called **Buffer Recorder** looks straight down and generates a render texture called **Meteorite_BufferRender**. This texture buffer then sends data to a separate graph called **GrassStrip**.



The BufferRecorder

The texture plugs into the GrassStrip graph, driving the movement and rendering of the Particle Strips. Here you can see how the texture buffer drives the effect:

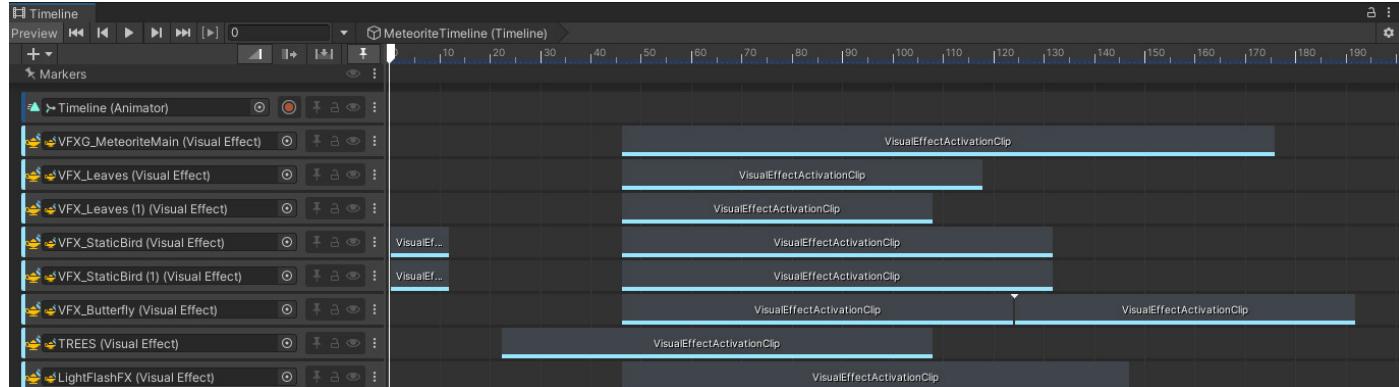


A texture buffer controls the grass effects.

Beyond the main meteorite, there are some secondary effects, including:

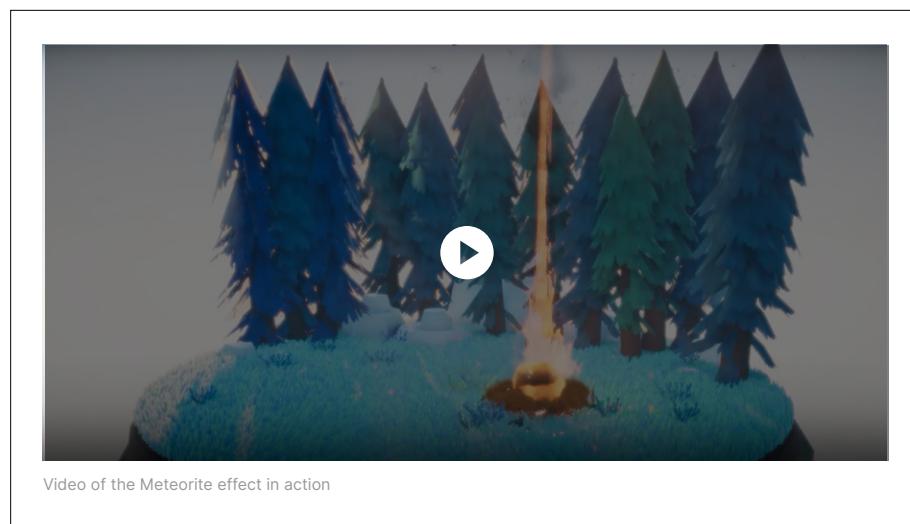
- Leaves dropping from the trees
- Birds scattering in the foreground and background
- Butterflies disappearing and reappearing

These secondary effects rely on the **Meteorite Timeline**, which contains several **Visual Effect Activation Tracks**. When the MeteoriteControl script kicks off the **Playable Director**, each secondary effect plays back according to its prescribed timing.



The Meteorite Timeline

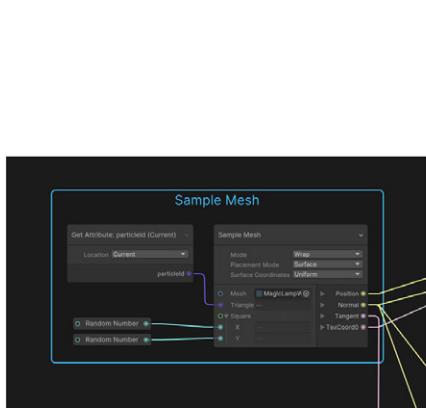
See the [Interactivity chapter](#) for more information on how to set up Timeline and Output Events.



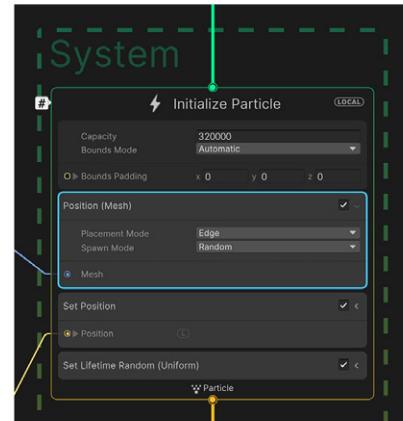
Mesh sampling effects

Mesh sampling is an experimental technique that lets you fetch data from a mesh and use the result in the graph. Sample a mesh with either the:

- [Position \(Mesh\) Block](#)
- [Sample Mesh Operator](#)



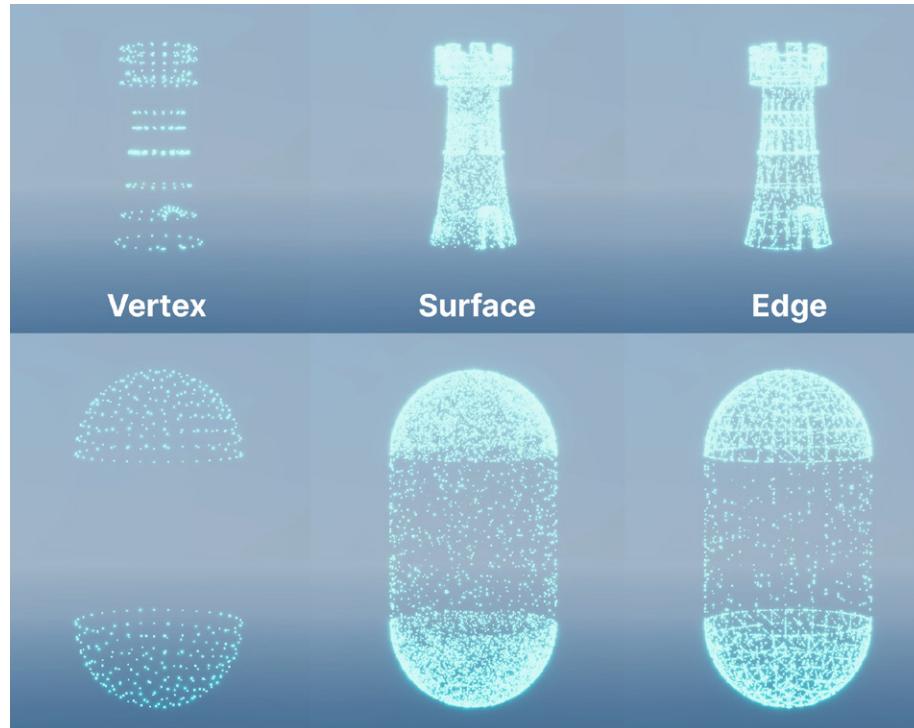
Sample Mesh Operator



Position (Mesh) Block

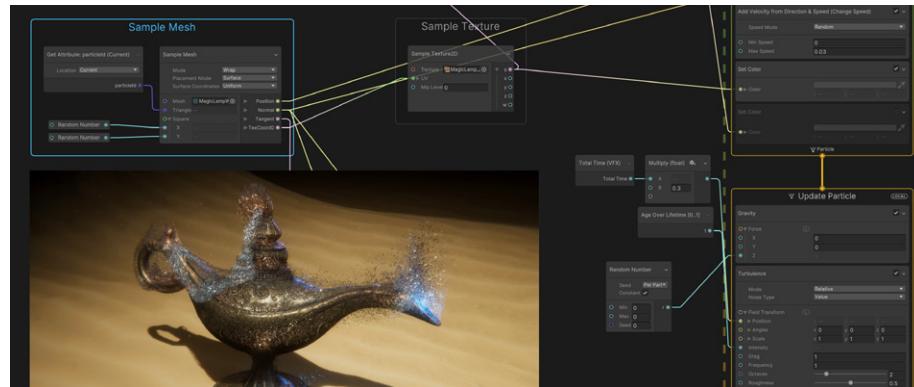
Sample Mesh Operator and Position (Mesh) Block

The **Placement Mode** can be set to **Vertex**, **Edge**, or **Surface**. Here's the **Position (Mesh) Block** at work on some simple meshes:



Vertex, Surface, or Edge sampling

This variation modifies the original Magic Lamp scene from the samples. With some additional work, you can use the **Sample Mesh Operator** to initialize particles on the lamp's surface and change their orientation as they float away. Grab the Magic Lamp's **Sample Texture** to smoothly integrate the particle colors with the mesh surface.



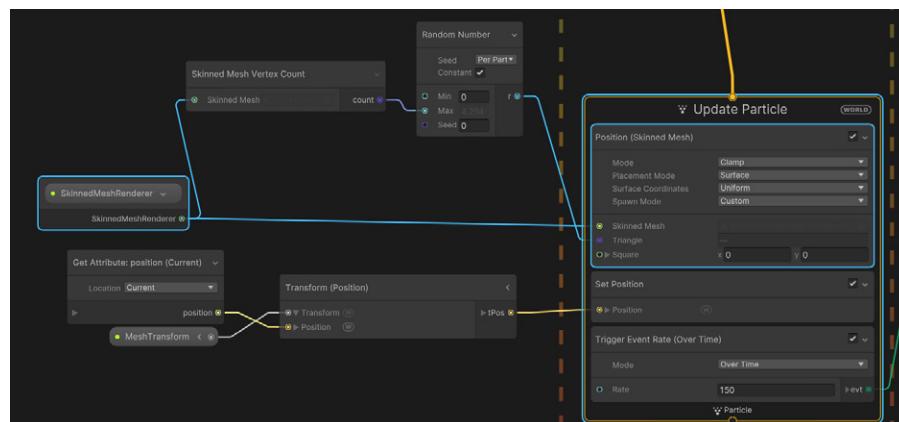
Magic Lamp sample scene

Skinned Mesh sampling

Applying effects to your characters in a game can add fun and eye-catching elements to the gameplay. Do you need a vampire to disintegrate after being staked through the heart? Or to burst into flames? By sampling a **Skinned Mesh Renderer**, either option is possible.

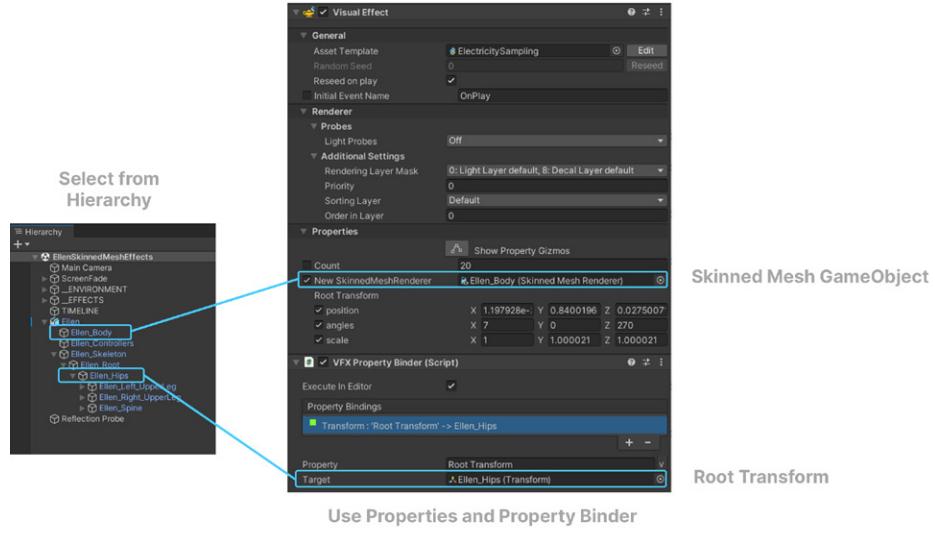
The sample effect called **EllenSkinnedMeshEffects** illustrates a variety of effects involving its title heroine. However, the general process for sampling skinned mesh data is similar in each case:

- Add a **Position (Skinned Mesh) Block** to a **Context (Initialize, Update, or Output)** depending on your intended effect).
- Expose a **Skinned Mesh Renderer** property on the **Blackboard**, and connect it to the Skinned Mesh flow port.
- Use a **Skinned Mesh Operator** if you need additional access to **Surface**, **Edge**, or **Vertex** data (see the Hologram and Disintegration graphs for example usage).



Setting up the Position (Skinned Mesh) in the graph

- In the **Inspector**, set the **Property** with a **Skinned Mesh Renderer** from your scene.
- Use a **Property Binder** (see [Interactivity chapter](#)) to transform the effect's position and orientation in your character's skeleton. Set this **Transform** and its corresponding **Set Position Block** to **World** space for optimal results.



You'll also need to add a **Mesh Transform Property Binder** that accounts for the **Base Transform** of the skeleton* itself. This varies from character to character, but in this case, it connects the character's hip joint. Now the particles can follow the Skinned Mesh Renderer.



The EllenSkinnedMeshEffects example

*Unity 2021 LTS is shown here. This step is not necessary in Unity 2022.2 or newer.

Leverage Skinned Mesh sampling's versatility when creating effects for characters and objects alike. In the samples, the Ellen character is shown:

- As a sci-fi holographic projection
- On fire, with smoke and sparks
- Wet, with water droplets trailing from her skin
- Dashing at super speed, with colored streaks left behind
- Jolted by electricity
- Turning into ashes

Each of these is a separate graph that samples a Skinned Mesh Renderer. You can examine their implementation details to see how the particles interact with the character mesh.

More examples

There are many more possibilities you can explore with the VFX Graph Samples. Be sure to check out the other scenes in the project; each one demonstrates a different set of techniques for creating a specific visual effect.

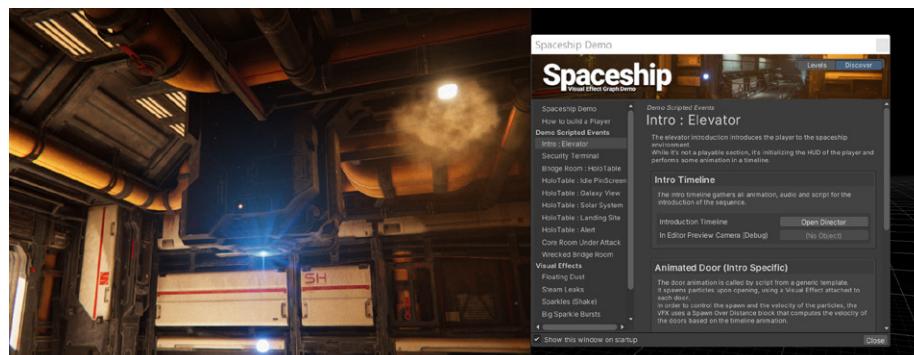
Read [this blog post](#) for more information or look out for [#unitytips](#) on VFX and shaders.

The Spaceship Demo

The Unity [Spaceship Demo](#), available on [GitHub](#) and [Steam](#), is a real-world project with all of its visual effects authored in the VFX Graph. The project serves to simulate actual game production conditions.

This demo showcases many common visual effects in game development, while still maintaining strong performance. It is a vertical slice of a AAA-quality cinematic, targeting 60+ fps on PC.

Note: The Spaceship Demo is verified for Unity 2021 LTS (version 2021.2), but will not be maintained for future releases.



The Discover window

Go to the **Discover** window via **Help > Discover > Spaceship Demo**. It will guide you through some key aspects of this large project. Then select the specific sequence on the left, and click on the right to choose the associated VFX Graph, Timeline, and/or Visual Effect GameObject.

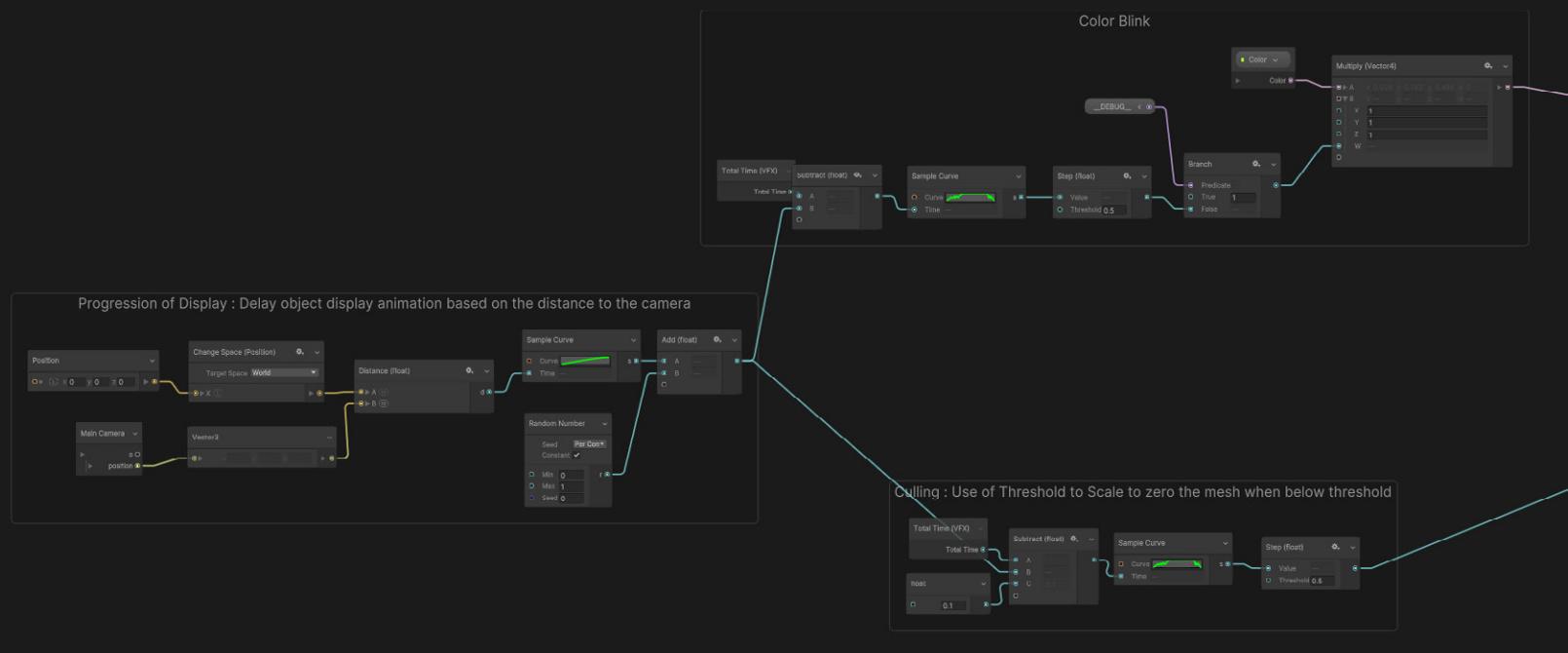
Let's look at a few of the notable graphs in the project.

Corridor Outliner

The cinematic opens with a holographic effect that introduces the viewer to the spaceship environment. This Outliner graph is an example of a VFX Graph that doesn't need particles at all; it doesn't require a Spawn, Initialize, or Update Context.

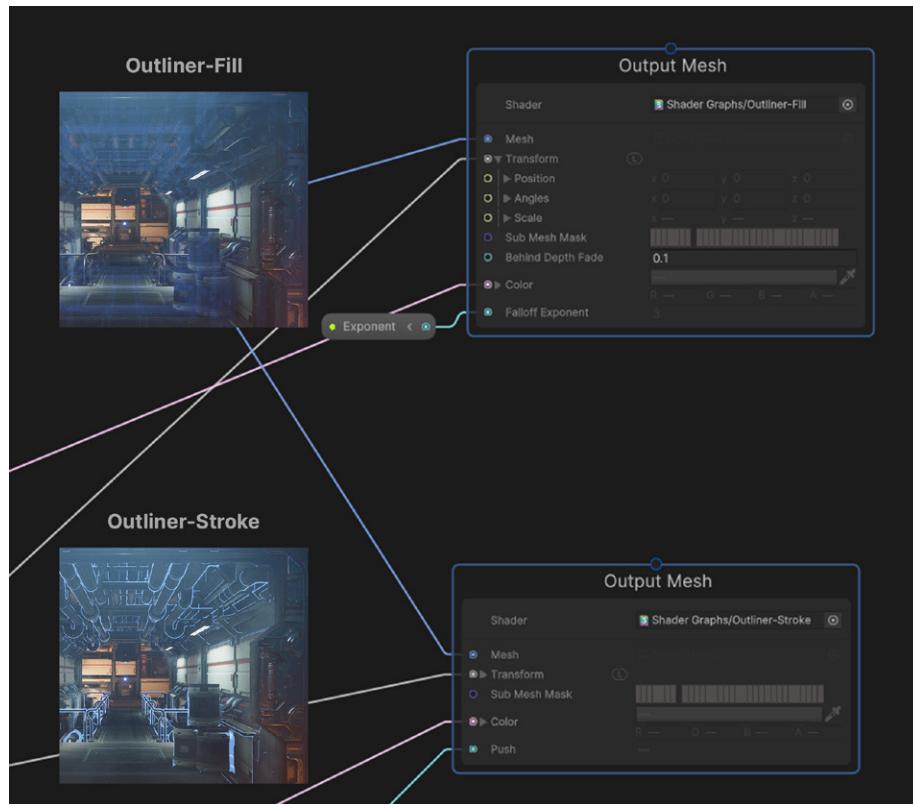
Instead, the graph has two **Output Mesh Contexts** with the effect relying on:

- Operator logic for the glitchy animation of the hologram activating
- Shader Graphs for the stylized, sci-fi rendering of the corridor mesh



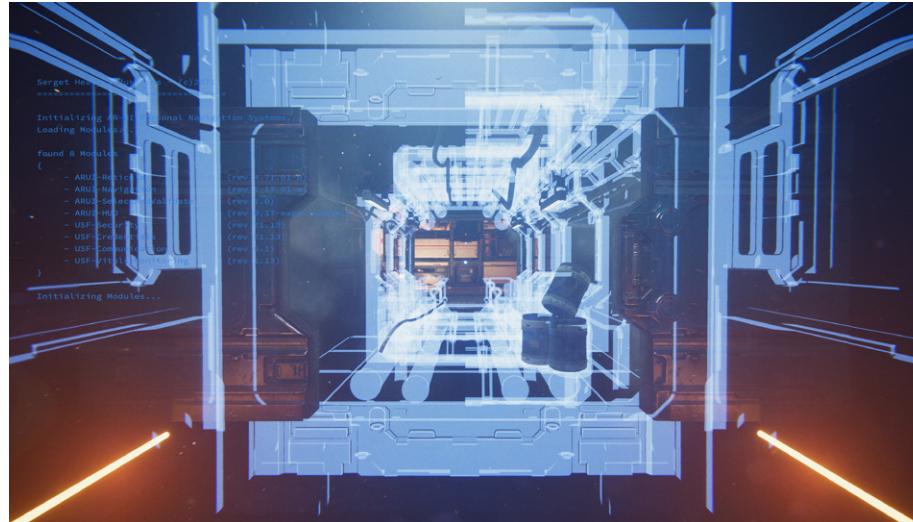
The Operators not only control how the walls blink on, they highlight each section to draw us out of the elevator. A custom Shader Graph integrated with the Output Context drives the effect.

The top Output Mesh uses a shader called **Outliner-Fill** to render the walls with a screenlike pattern. The bottom Output Mesh uses an **Outliner-Stroke** shader to draw the outlines.



Output Mesh Contexts

By itself, the effect resembles a translucent rendition of the mesh with some heavy outlines.



The Outliner effect rendered without the corridor mesh

Combining this effect with the original geometry of the **Start_Corridor** creates the impression of a hologram superimposed on the spaceship.



The Outliner effect rendered with the mesh geometry

Every part of the corridor activates in succession, leading the viewer forward. Though relatively subtle, this sets the stage for the rest of the augmented reality (AR) effects that follow.

Augmented reality user interface

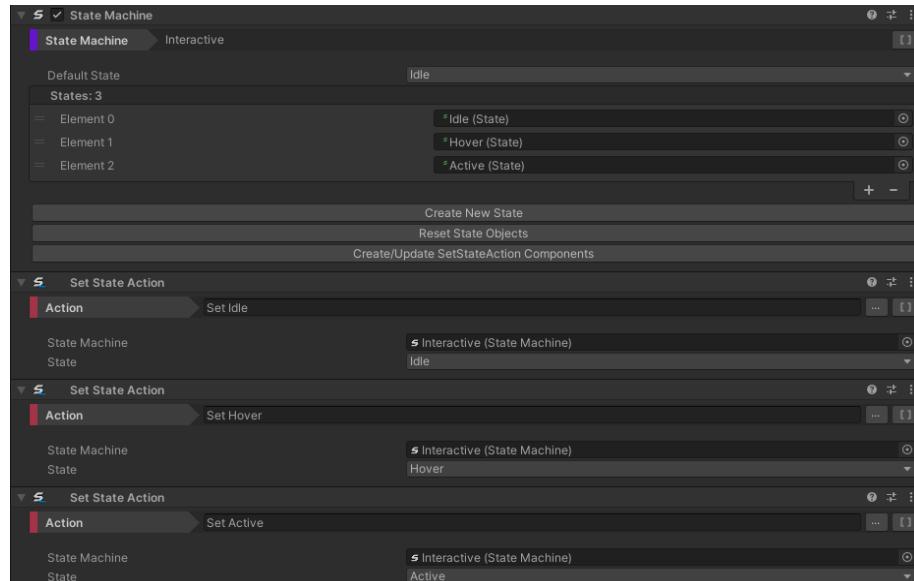
The spaceship's security room depicts a particular augmented reality effect. This AR effect comprises a graph simulating the login screen to a computer terminal.



The Security Terminal effect

The effect begins with a glowing dot; just one particle with an endless lifespan. As the viewer approaches, a **State Machine** script transitions between **Idle** and **Hover** states.

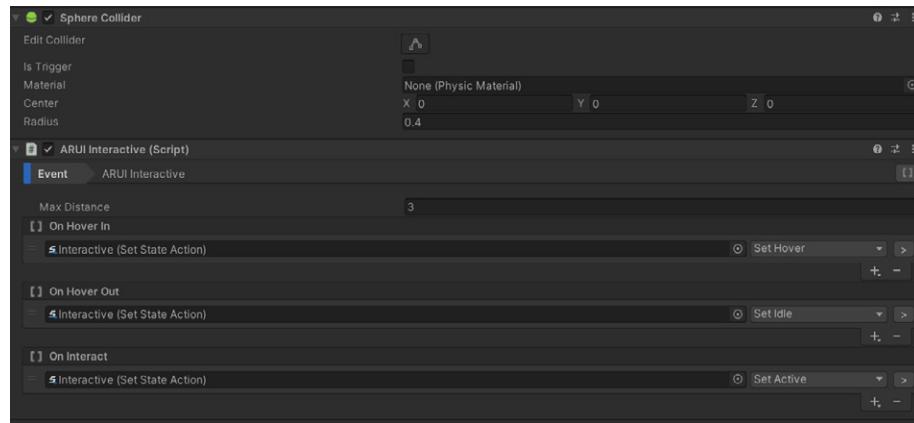
The **State Machine** derives from the [Gameplay Ingredients](#) package created by Thomas Iché. It is an open-source repository of scripts that speed up simple tasks while making games and prototypes. Instructions for installing the [Gameplay Ingredients](#) via the Unity Package Manager can be found [here](#).



A custom State Machine

A **Sphere Collider** with a custom **ARUI Interactive** script uses a **Physics**.

Raycast to verify whether the mouse cursor is over the intended target of the display. The graph called **InteractiveButton** loops the animation until you click the mouse button.

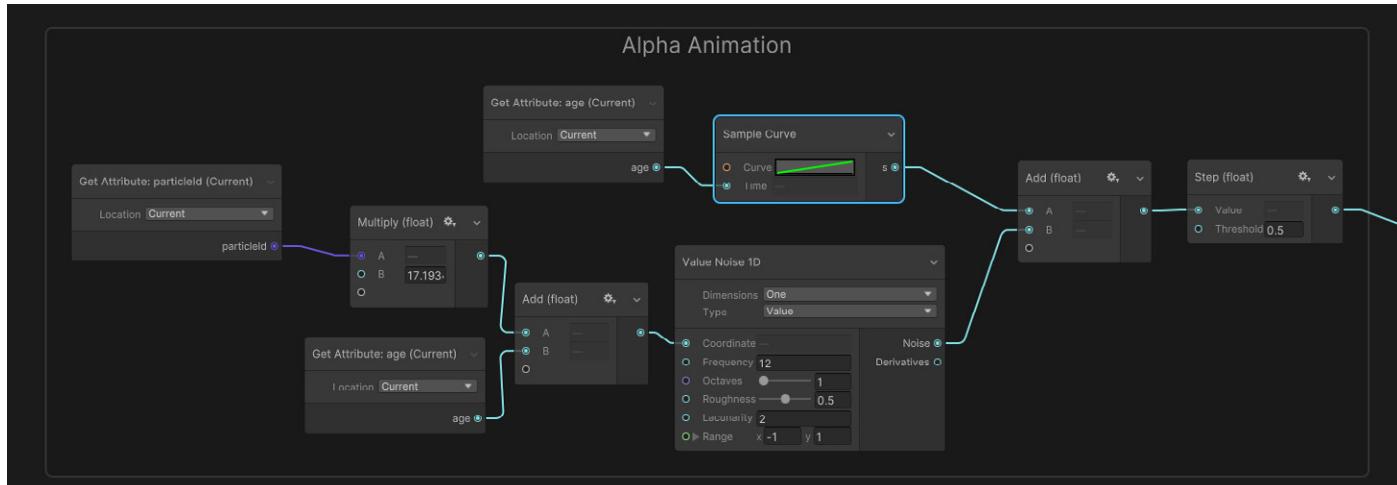


The ARUI Interactive component

By clicking, you switch the ARUI Interactive to the **Active** state, setting off the terminal's motion graphics and animated text. Several VFX Graphs play together in concert to achieve the final effect.

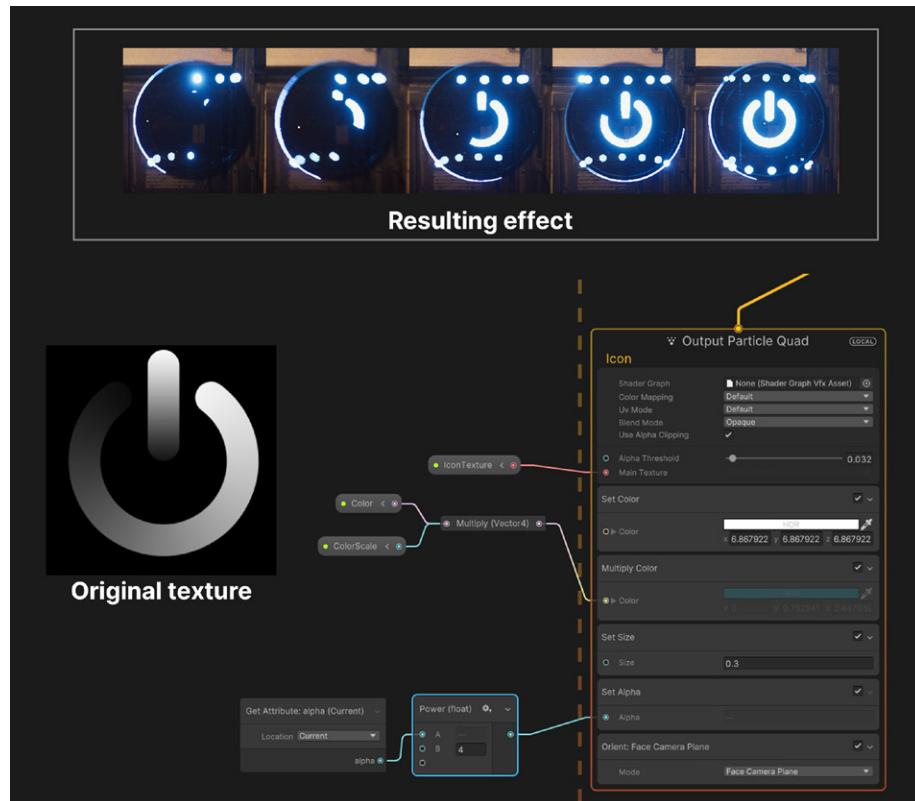
Take a closer look at the individual graphs within the **TerminalRoom** folder to see some of these techniques at work:

- **Use Sample Curves to add quick animation:** Pass the particle's **Age** from the **Get Attribute Operator** into the Sample Curve's **Time** input. Define your own custom **Animation Curve** and then manipulate each particle's size, opacity, and speed based on its duration or lifetime.
- **Add Noise to make the motion more organic:** If your motion is too perfect or mechanical, a **Noise Node (Perlin or otherwise)** can break up its uniformity.



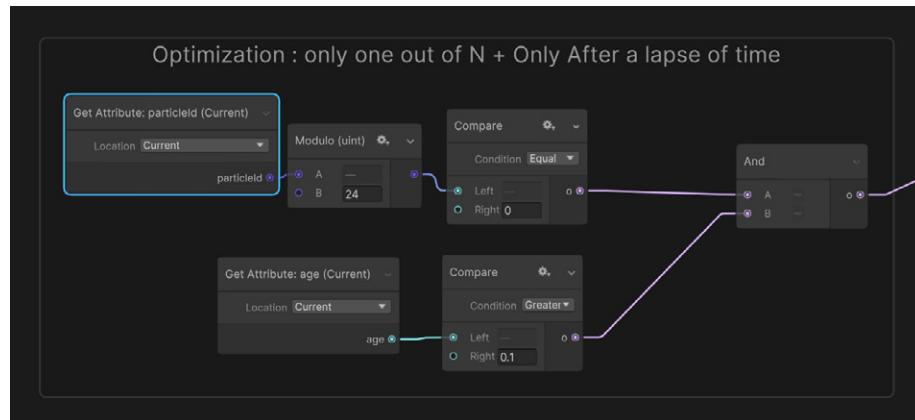
A Sample Curve with added Noise

- **Pack Texture alpha channels to animate with alpha erosion:** With the right Operators, you can use alpha erosion to create a sense of motion. Here, the Power icon appears to dissolve in and out, even though it's just a static image with gradients over parts of the alpha channel.



The Power icon "animates" via alpha erosion

- **Use each particle ID to your advantage:** Every particle has a unique ID number, which can be handy. To sample only every nth particle, you can use the **Particle Attribute** and pass it into a **Modulo Operator**. This example keeps only one out of 24 particles, enabling them after a short delay.



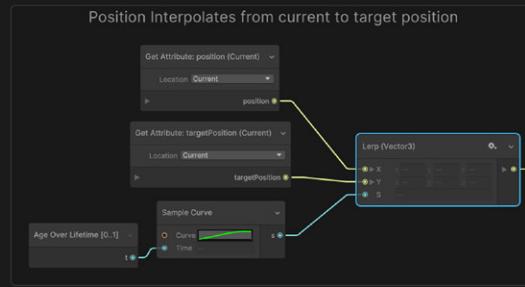
Use the particle ID

- **Lerp for direct control:** Particle motion doesn't always need to be driven by physics. You can interpolate to keep particles on a specific path.

In the **Matrix Burst** effect part of the **ARUI-Bg**, a bunch of particles are scattered over a surface. Set their **Target Position**. By resetting the points back to origin, you can lerp between the start and end positions.



Move particles with the Lerp operator



The Matrix Burst effect uses lerp movement.

Holographic table

The holographic table reuses the ARUI Interactive component in a more complex interface. Much like the Security Terminal effect, the player can select items from the onscreen map.

Although the HoloTable looks more sophisticated, its core functionality is the same. Hover your mouse over any part of the visual effect, and click for the w component to make the planetary map react to your pointer.

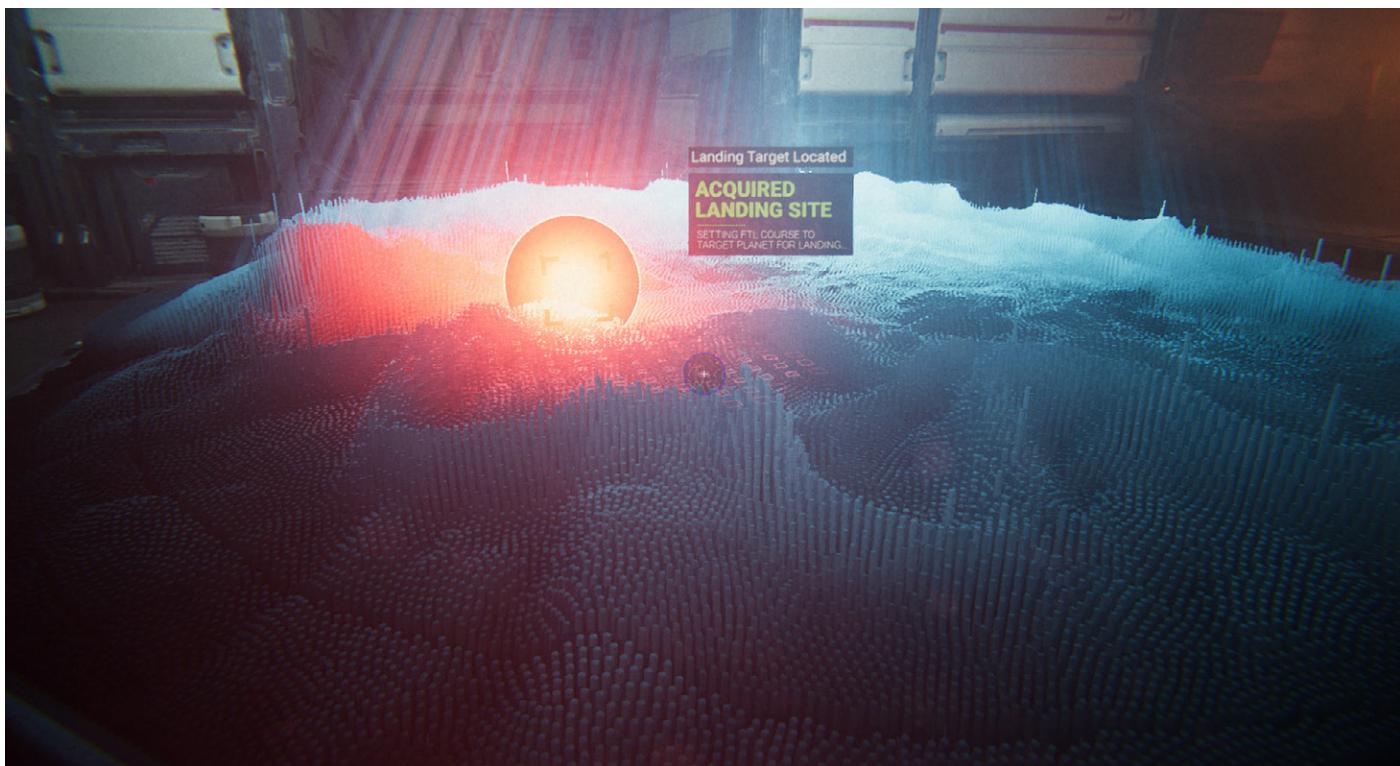


The HoloTable reacts to the mouse pointer.

The player selects a target system and a landing zone before the rest of the sequence plays out. While this particular cinematic has a predetermined ending, there are interactive point-and-click elements that help keep the players engaged.



The ARUI Interactive

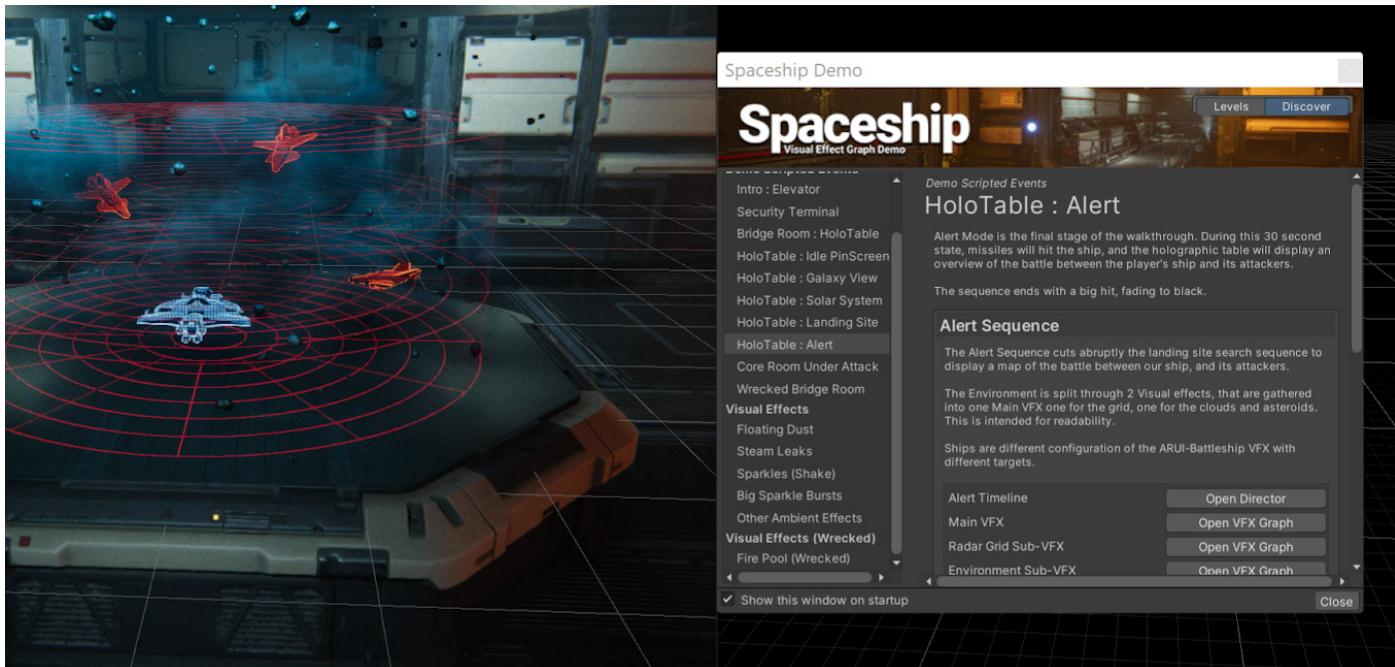


The PinScreen effect

Note: While the ARUI Interactive script uses a custom solution, you can alternatively use **Event Binders** (see [Interactivity](#)) to achieve the same transition between states in the State Machine.

Core effect

Once you select a landing zone from the HoloTable, enemy ships appear to attack the spaceship onscreen. Every missile hit depletes the spaceship's core energy. After a final hit, the core collapses and flames engulf the environment.

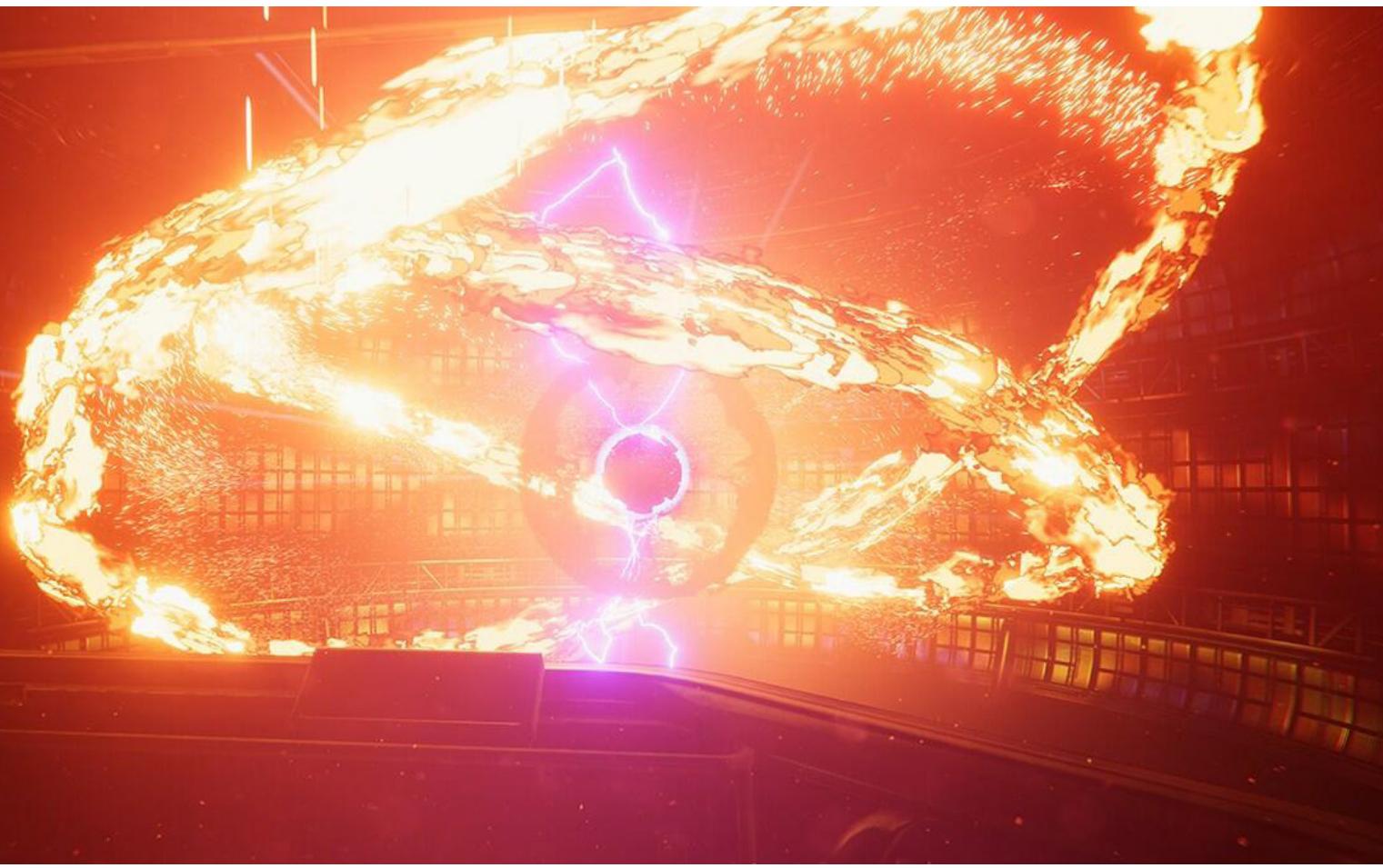


The Alert sequence

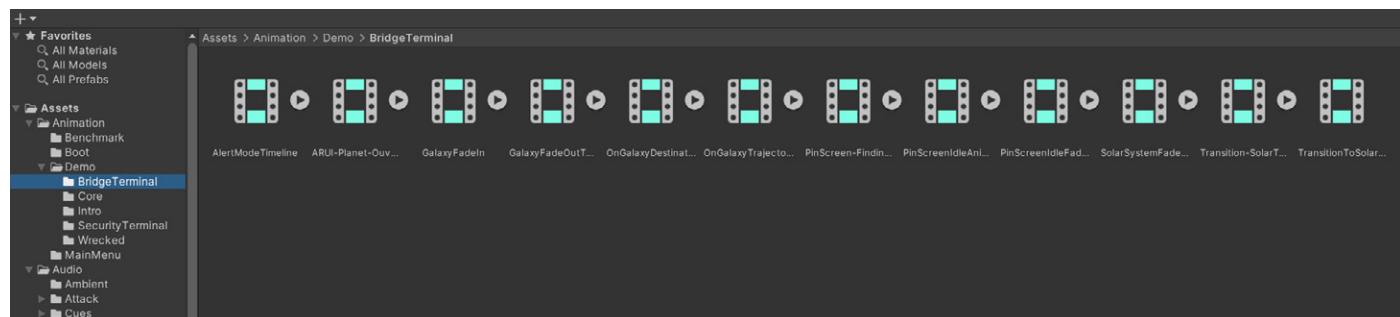
The project uses carefully orchestrated Timelines to sequence the various effects:

- **The AlertModeTimeline** determines how the missiles strike the spaceship as the action plays out on the HoloTable. This acts as an overarching Timeline that can control other Timelines.
- **The CoreHitTimeline** plays back moments where the camera shakes or sparks fly for each missile strike on the spaceship.
- **The CoreAttackTimeline** controls the core effect and how its energy depletes.

Other Timelines in the Animation/Demos control the Security Terminal, Intro, and Wrecked Bridge Room sequences. Edit each [Playable Director](#) component to send messages to the VFX objects in the scene.



The Core room



Various Timelines in the Spaceship Demo project

Download the Spaceship project from [GitHub](#) and watch [this video](#) for a breakdown of its unique effects. Then continue to the next chapter to find out more on integrating [Timeline](#) with the VFX Graph.

INTERACTIVITY

Visual effects often involve many moving pieces. Connecting them to the correct points in your application is essential to integrating them at runtime.

Whether you need a projectile to explode on contact or bolts of electricity to jump from the mouse pointer, one of these available tools can be used to play back the effect:

- **Event Binders:** These listen for several different things that happen in your scene and react to specific actions at runtime.
- **Timeline:** You can sequence visual effects with Activation Tracks to send events to your graph at select moments. Gain precise control with pre-scripted timing (e.g., playing effects during a cutscene).
- **Property Binders:** These link scene or gameplay values to the [Exposed properties](#) on your Blackboard so that your effects react to changes in the scene, in real-time.

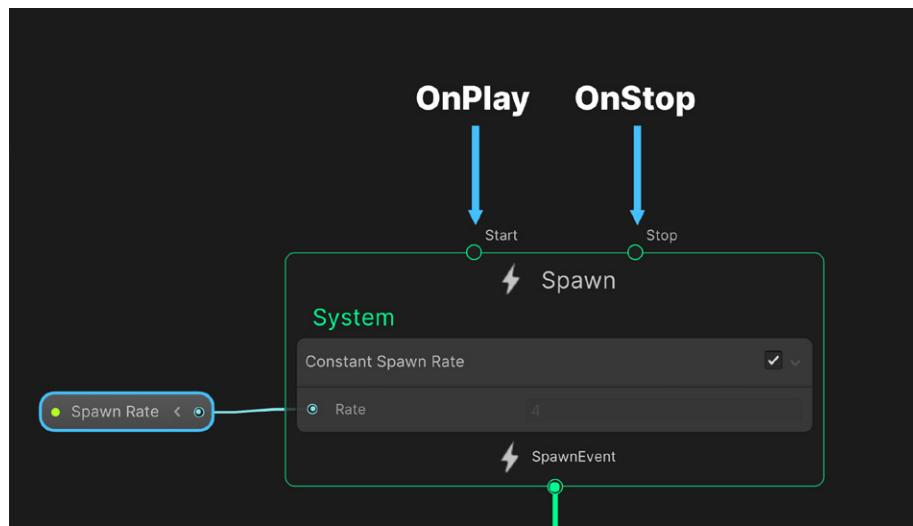
Let's explore each of these tools in more detail as they are crucial techniques for bridging your GameObjects with VFX Graphs.

Event Binders

Event Binders are **MonoBehaviour** scripts that can invoke [Events](#) from within the VFX Graph. They ensure that your effects react to mouse actions, collisions, triggers, and visibility events in the scene.

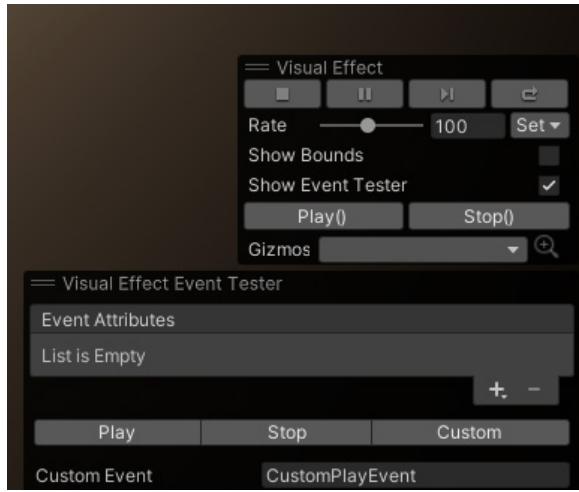
OnPlay and OnStop Events

By default, a Spawn Context in the VFX Graph includes an OnPlay or OnStop Event. The Start and Stop flow slots in each Spawn Context receive these implicitly if you don't plug in a specific Event.



OnPlay and OnStop implicitly flow into the Spawn Context.

If you're familiar with the GameObject playback controls in the Scene view, the **Play()** and **Stop()** buttons at the bottom send the OnPlay and OnStop Events, respectively.



Use the Play and Stop buttons to send the OnPlay and OnStop Events.

Events facilitate the process of sending messages between objects. In the VFX Graph, Events pass as strings. Pressing OnPlay or OnStop doesn't change the effect immediately, especially compared to using the playback icons at the top. They simply provide signals to the Spawn system.

If you open the dialog window **Visual Effect Event Tester**, you can use the Play and Stop buttons for the same effect. Take advantage of this flexibility to specify a **Custom Event** and invoke it with the Custom button.

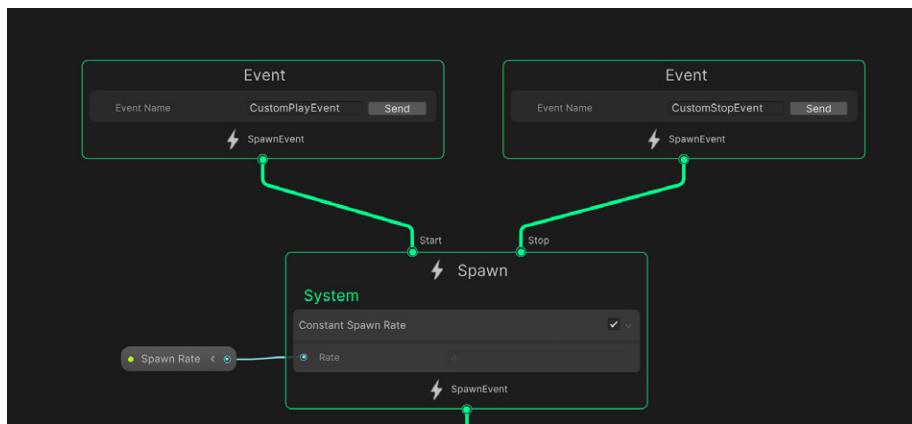


The Visual Effect Event Tester

Add your own Event to the graph using **Node > Context > Event**. Press the **Send** button to raise the Event manually when testing.

Creating Custom Events is a matter of changing the **Event Name** string and invoking the Event with an **Event Binder** or **Timeline Activation Clip** at runtime.

In this example, two Events called **CustomPlay** and **CustomStop** have been added.

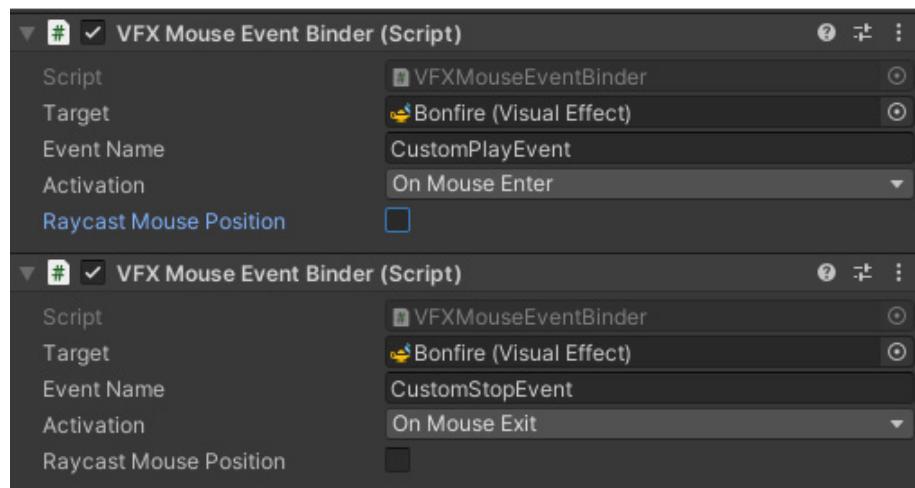


How to set up Custom Events in your graph

Mouse Event Binder

Clicking, hovering, or dragging a mouse pointer can send messages to your graph using a VFX Mouse Event Binder. This only requires a GameObject with a Collider.

In the example, two Event Binders connect the **CustomPlay Event** and **CustomStop Event** to the Bonfire effect.



Activating an effect with a Mouse Event Binder

If the mouse pointer enters the Collider onscreen, send the CustomPlay Event to the graph. This Event begins spawning the flames, smoke, and sparks. If the mouse pointer exits, the CustomStop Event notifies the Spawn Context to stop.

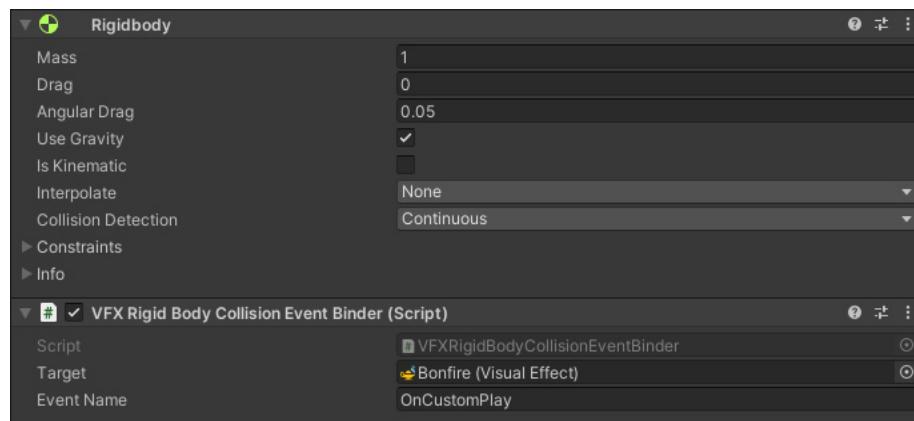


The mouse pointer raises Events when entering and exiting the Collider.

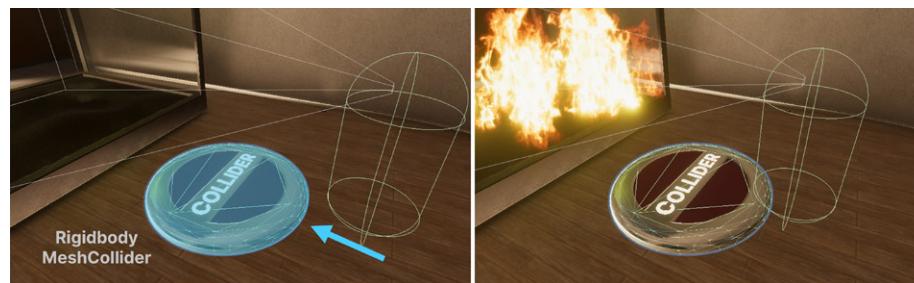
You can bind Events to any standard mouse actions (**Up**, **Down**, **Enter**, **Exit**, **Over**, or **Drag**). The **Raycast Mouse Position** option passes the pointer's 3D location as an [Event Attribute](#).

Rigidbody Collision Event Binders

A Rigidbody Collision Event Binder enables a physics object to alert the graph when a collision occurs. Attach it to any suitable GameObject with a **Rigidbody** and **Collider**.



A Collision sends an Event to the VFX Graph.



Rigidbody Collision

Any Collider making contact raises an Event (with the specified Event Name) and sends it as a message to the **Visual Effect Target**.

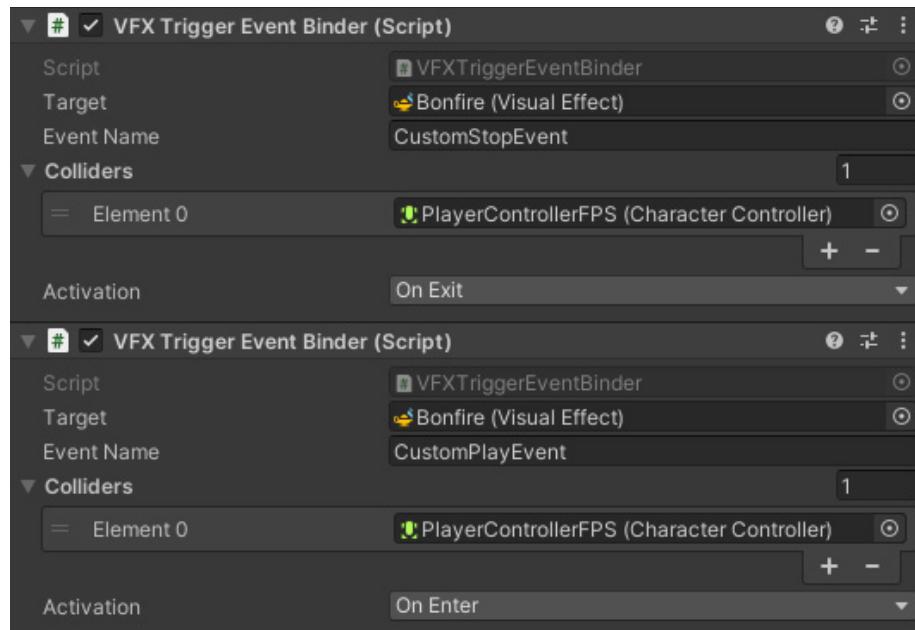
In this example, the **Mesh Collider** with a Rigidbody acts as a button. The effect only begins spawning once the player makes contact.

Seeing as this Event Binder responds to Collisions with a Rigidbody, you can create different forms of interactivity with it. Imagine a sphere that emits particles every time it bounces, or a force field that becomes distorted when hit with a projectile.

Trigger Event Binder

You can similarly add a VFX Trigger Event Binder to a GameObject with a Collider set as a trigger. This component can register certain **Activation Events** like **OnEnter**, **OnExit**, or **OnStay**. It sends an Event with the Event Name to the **Target** graph.

Use this to trigger an effect when the player, or some other GameObject, reaches a particular part of the level.



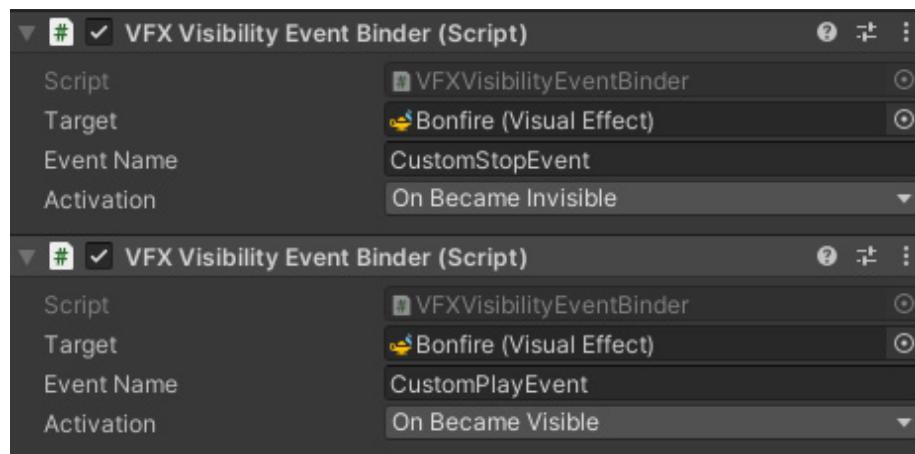
Trigger Event Binders



A Trigger Event Binder detects specific Colliders.

Visibility Event Binder

A Visibility Event Binder allows you to raise an Event depending on when an object becomes visible or invisible. Here, we attach it to a GameObject with some type of Renderer.

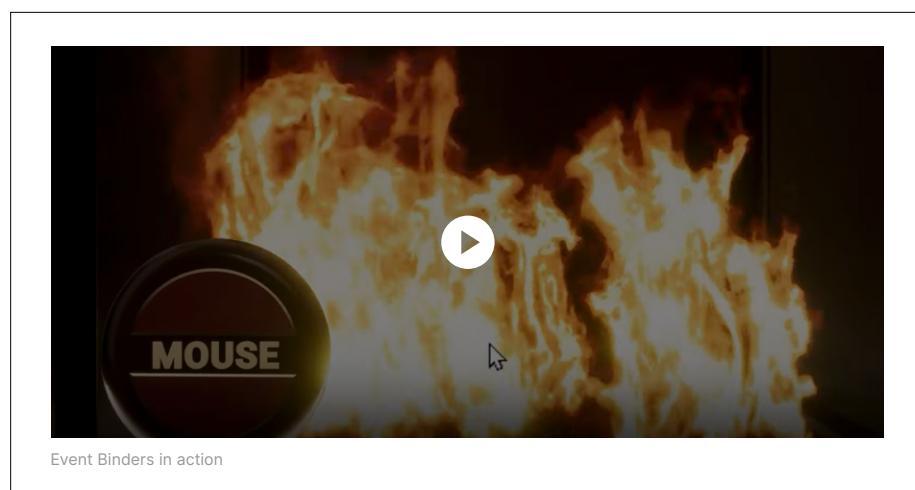


Visibility Event Binder

Pass the specific Event Name to the Target graph based on its **Activation**, either **OnBecameVisible** or **OnBecameInvisible**. This notifies the graph when the Renderer enters or leaves the Camera frustum, or toggles its Renderer on and off.

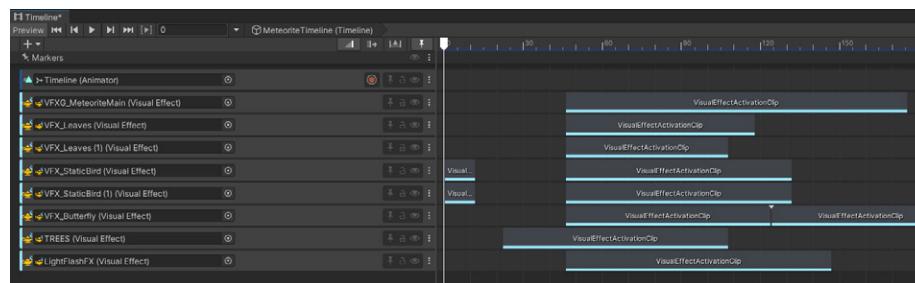


The visual effect only activates once the target becomes visible.

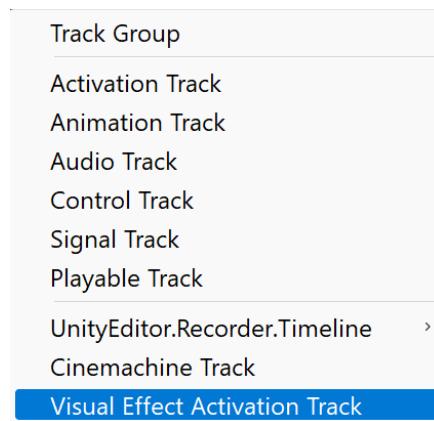


Timeline

Timeline offers another way to communicate with your graphs, should you need to turn your visual effects on and off with precise timing. You can coordinate multiple layers of effects with Visual Effect Activation Tracks.

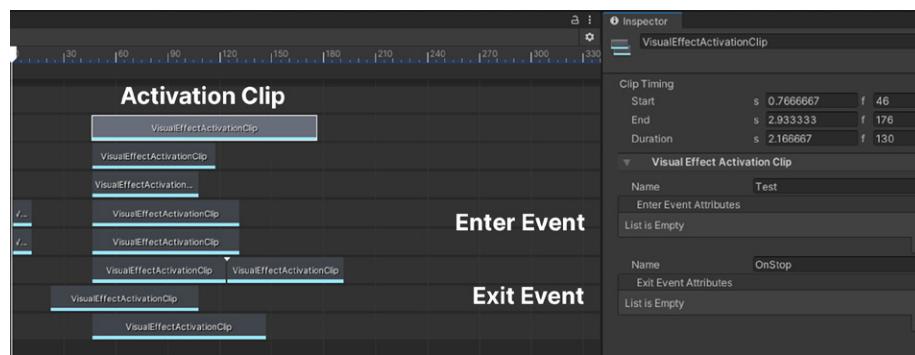


Visual Effect Activation Tracks



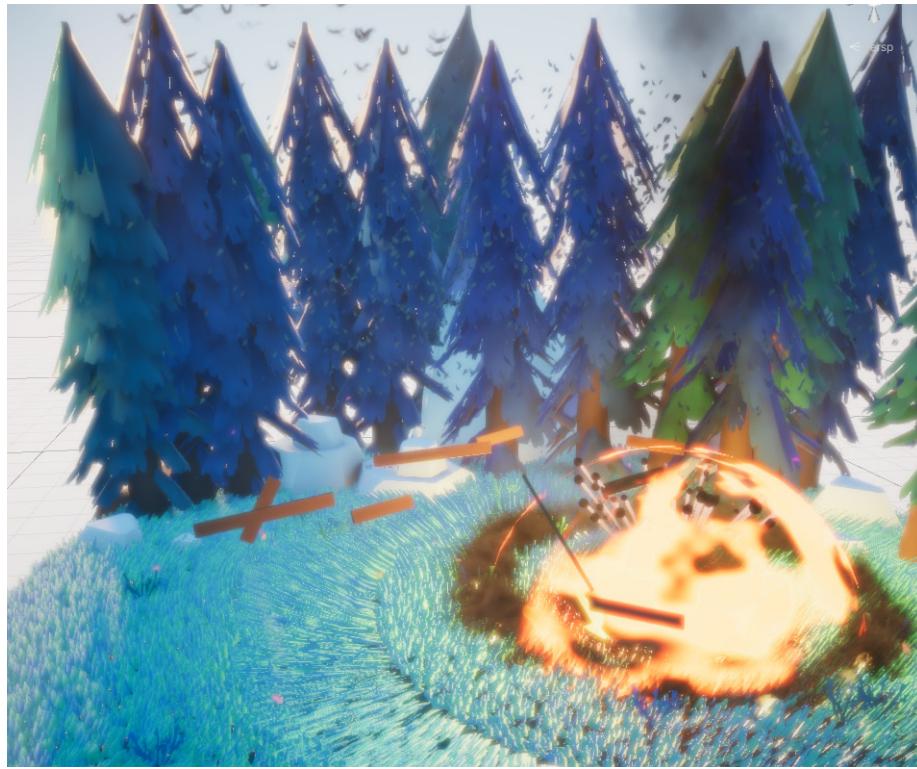
Right-click in the Timeline to create a new Activation Track.

Assign an effect to an **Activation Track** and then create one or more **Activation Clips** in the Timeline track. Each clip sends two Events; one at the beginning and one at the end.



Each Activation Clip sends two Events to its VFX Graph.

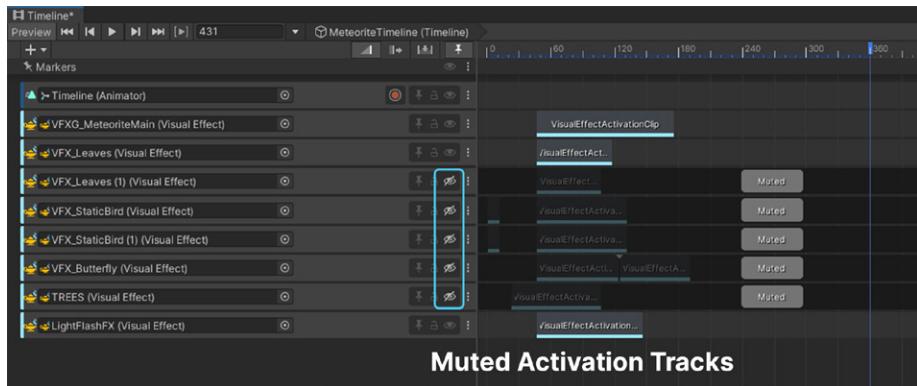
The Meteorite sample demonstrates how you can use Timeline to control multiple effects that play back in concert.



The Meteorite sample from the VFX Graph Samples project

Timeline helps organize the pieces that collectively create the sum total of the effect. Here, a separate Activation Track is used for each Subgraph and passes in Events through the Activation Clips.

By sliding the clips within Timeline, you can adjust the timing interactively. The custom MeteoriteControl script then invokes the Playable Director component.



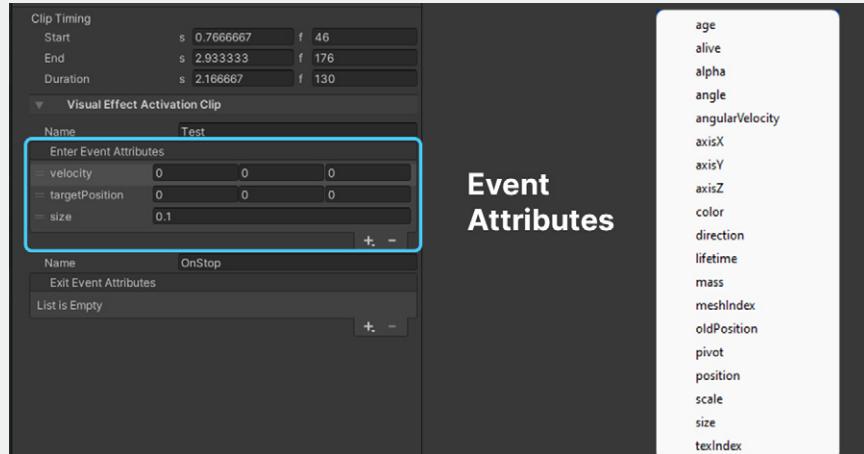
Muting the Activation Tracks

You can also use Timeline to mute specific Visual Effect Activation Tracks, which temporarily stops them from receiving Events. This can be useful when troubleshooting.

Event Attributes

Both Event Binders and Timeline Activation Clips can attach [Event Attribute Payloads](#) to Events. In doing so, they pass along extra information with an Event when it's invoked.

For instance, you might create an Event Attribute with an exposed **Vector3** property that notifies the graph where to instantiate the effect.

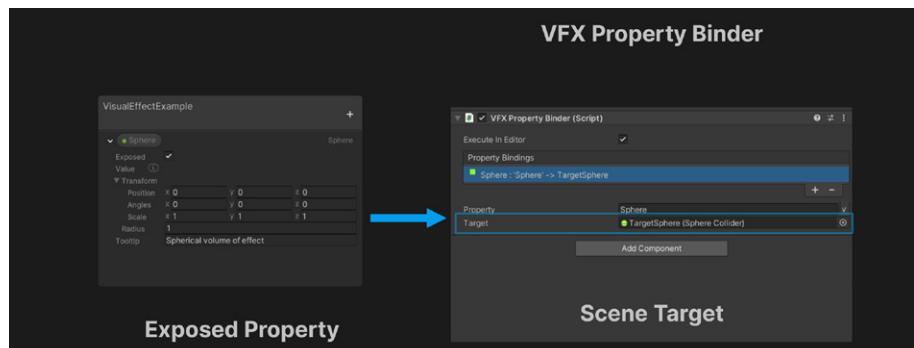


To set these Attributes in a VFX Graph, use the **Set Attribute Blocks** in the **Spawn Contexts**. You can also attach them to Events sent from C# scripts. See the Visual Effect [component API](#) for more information.

Property Binder

Property Binders are C# behaviors that enable you to connect scene or gameplay values to the [Exposed properties](#) of the VFX Graph. You can add Property Binders through a common MonoBehaviour called the **VFX Property Binder**.

For example, a **Sphere Collider Binder** can automatically set the position and the radius of a **Sphere Exposed Property** using the values of a Sphere Collider in the scene.

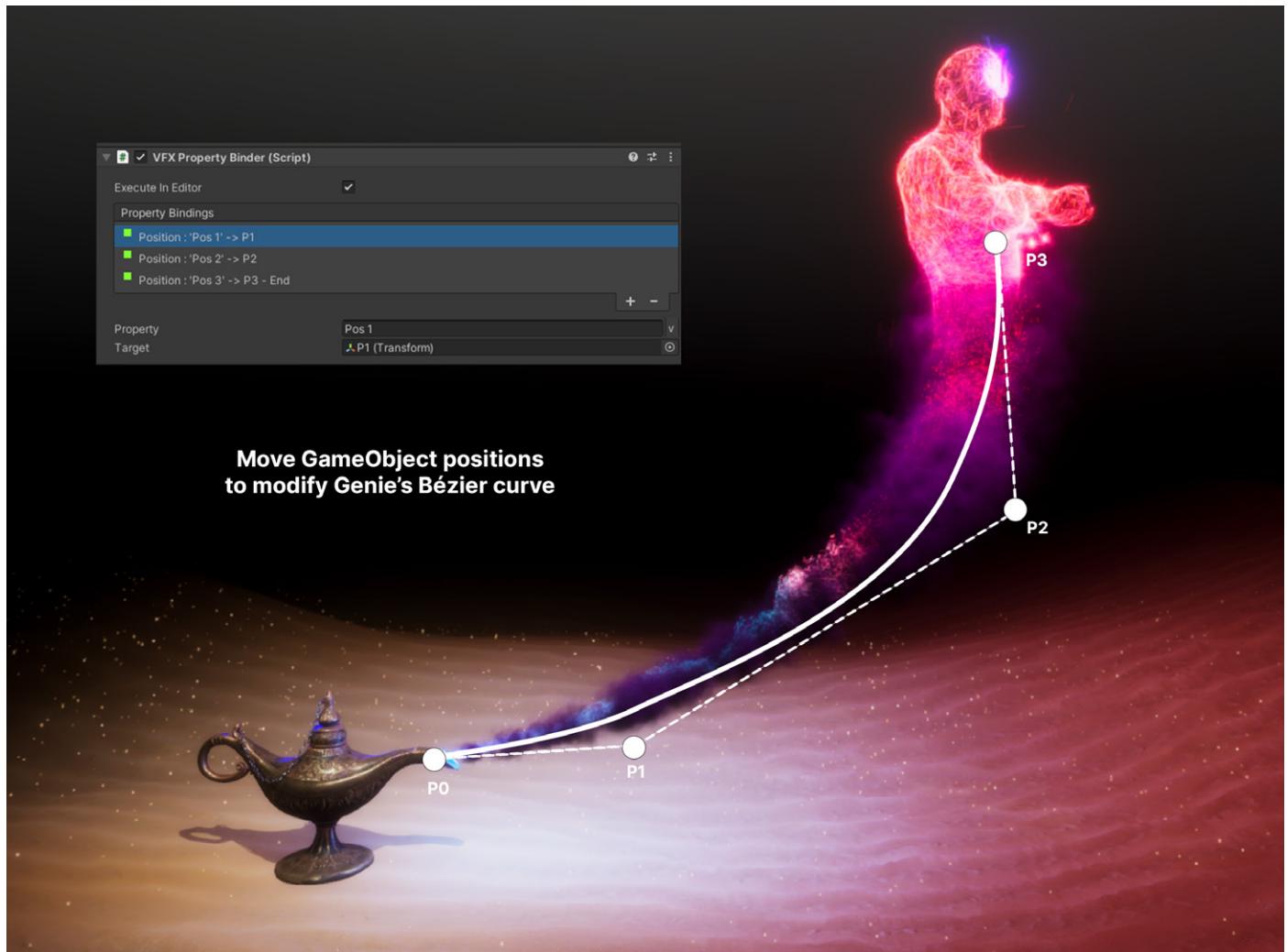


The Property Binder connects a scene value to an Exposed Property.

Do you need a light or camera in the scene to influence your effect at runtime? Does the effect follow a Transform or a Vector3? A Property Binder can sync a number of Exposed Property types with values in your scene. Go to **Add Component > VFX > Property Binders** for the complete selection of what's available.

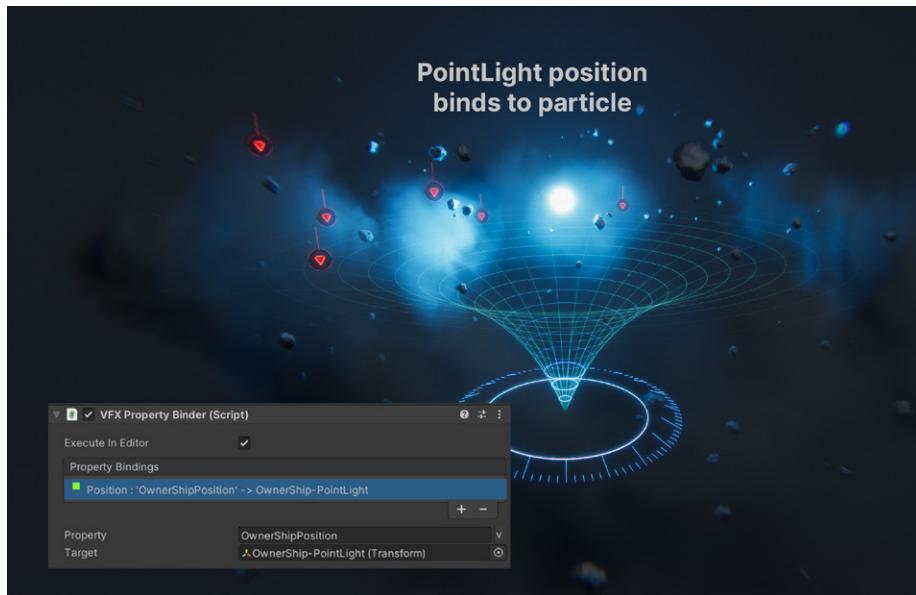
There are instances in the Visual Effect Samples that show how Property Binders can connect the Scene Hierarchy to the graph:

- In the Magic Lamp sample, the Property Binder ties the position of several Scene objects (P1, P2, and P3) to the graph's Blackboard properties (Pos1, Pos2, and Pos3). These **Vector3 Nodes** then form a **Bézier spline** defining the genie's overall shape. Move the P1, P2, and P3 **Transforms** in your scene, and the genie's smoke trail will respond in real-time.



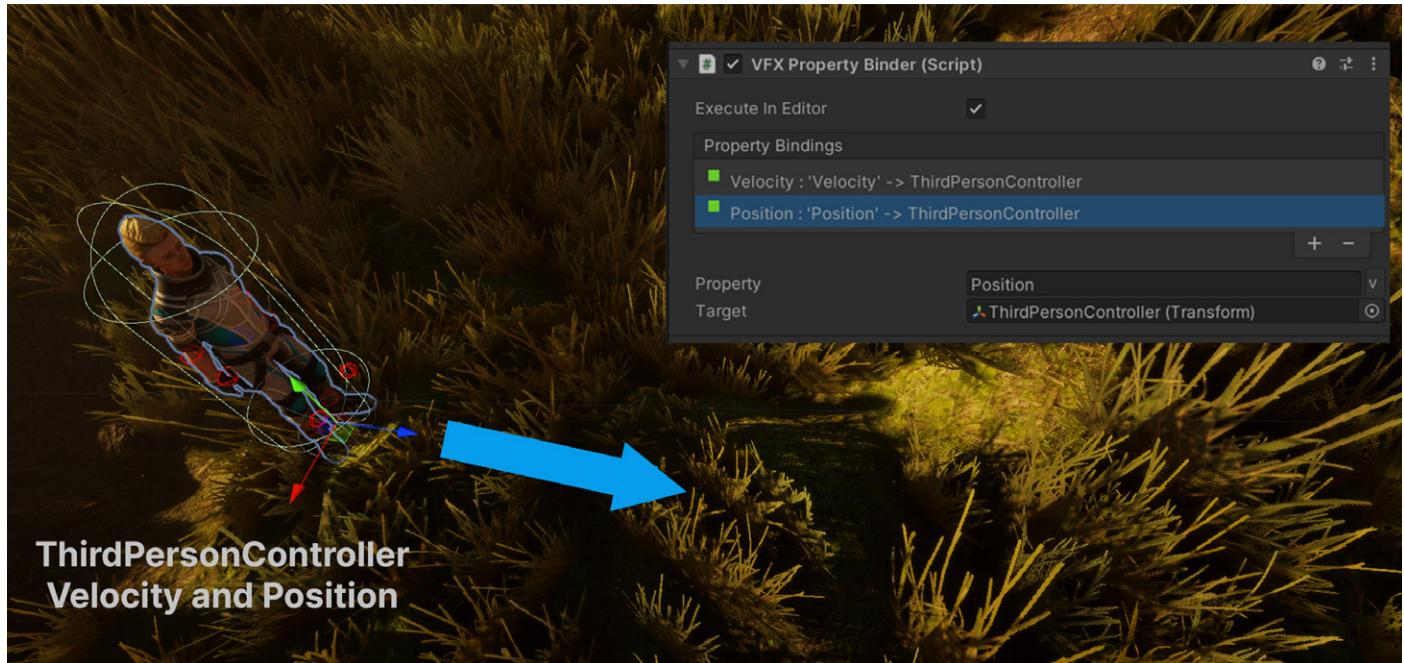
The resulting Bézier curve drives the genie's shape.

- In AR Radar, a **PointLight Transform** determines where the player's ship appears on the 3D radar screen. It syncs the glowing blip with real-time light.



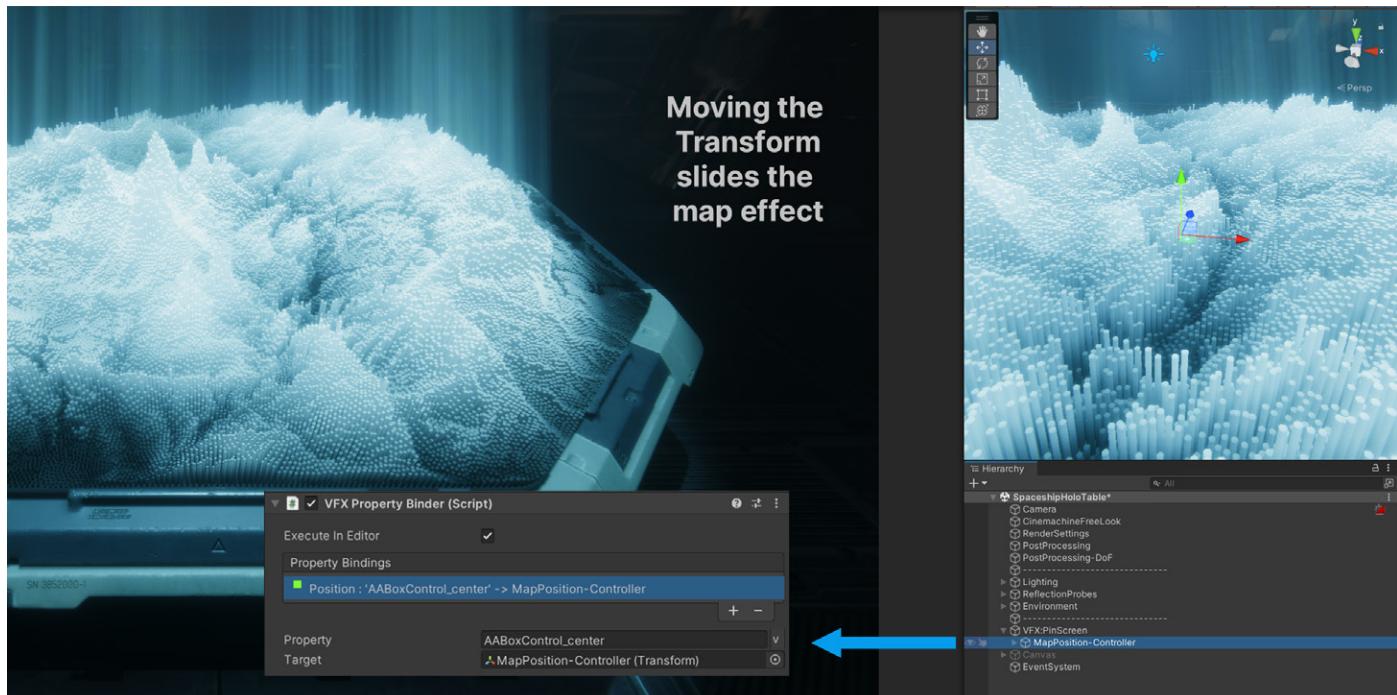
The AR Radar connects a PointLight to the ship icon.

- In the Grass Wind sample, Property Binders capture the Position and Velocity of the Transform called **ThirdPersonController** to push the grass.



The Grass Wind sample

- The SpaceshipHoloTable uses a Transform called **MapPosition-Controller** to drive the Position of the PinScreen effect on the tabletop. Let the **Animator** move this through a predetermined motion or drag it around at runtime to watch the hologram come to life.



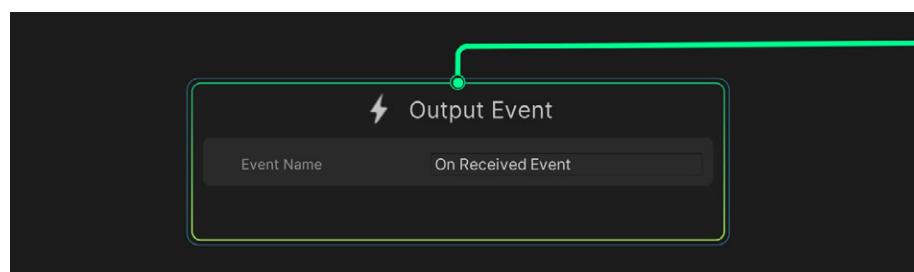
The SpaceshipHoloTable

These are just a few instances where Property Binders can solidify the relationship between a graph and your scene. Find [built-in Property Binders](#) for audio, input, physics, and UI, among other components.

Need something that doesn't exist yet? Use the [UnityEngine.VFX.Utility](#).
VFXBinderBase class to [write your own](#) Property Binder.

Output Events

Just as you can leverage Events to send messages into the VFX Graph, you can similarly use them to send messages out. With [Output Events](#), you have the ability to obtain the Attributes of new particles from a **Spawner Context**. Use them with [Output Event Handlers](#) to notify your C# scripts in the scene.



Output Events

Create any number of behaviors that respond to your effect; shake the Camera, play back a sound, spawn a Prefab, or anything else your gameplay logic dictates.

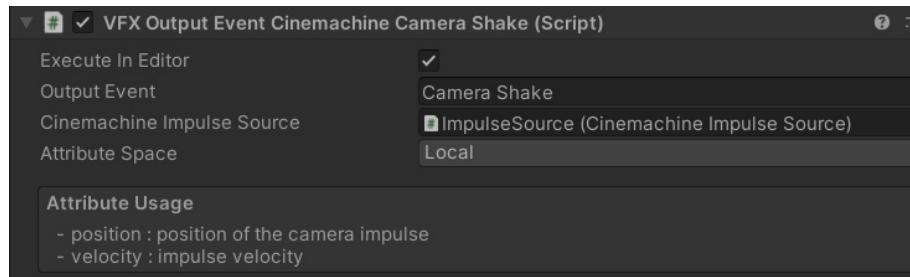
In order to receive Events, [inherit your scripted class](#) from **VFXOutputEventAbstractHandler** in the **UnityEngine.VFX.Utility** namespace. Then, override this method:

```
override void OnVFXOutputEvent(VFXEventAttribute  
eventAttribute)
```

Unity calls `OnVFXOutputEvent` whenever an Event triggers, passing the Event Attributes as parameters. Look for provided implementations in the **Output Event Helpers**, included with the VFX Graph. Install them from the Package Manager to review the example scripts.

You can also revisit the Meteorite sample to see how they work within an actual graph. Attach some of these Output Event Handlers to the **VFX_MeteoriteMain GameObject**:

- **Camera Shake: The VFX Output Event Cinemachine Camera Shake** component rattles the Camera upon the **Camera Shake Event**.



The Camera Shake script

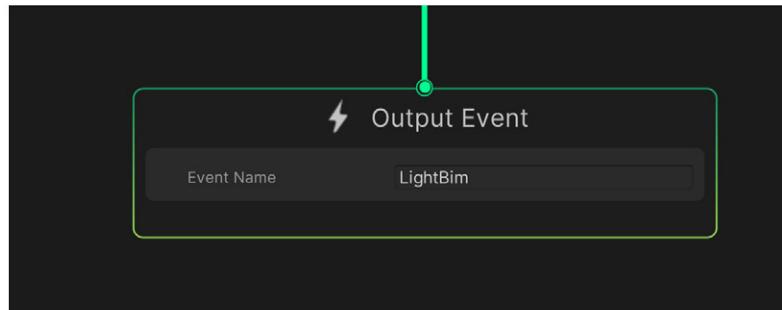
- **Secondary effects:** The **VFX Output Event Prefab Spawn** components raise the **Buffer** and **PlankImpulse Events**. A resulting shock wave passes through the grass and sends wood planks flying, courtesy of Output Event Handlers.



Output Event Handlers spawn secondary effects.

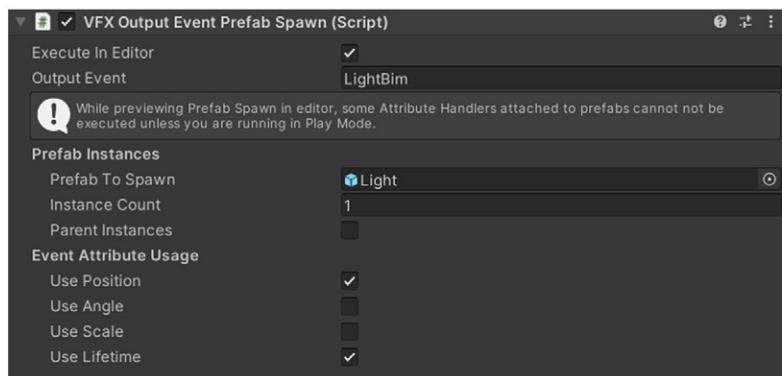
- **Light animation:** Another VFX Output Event Prefab Spawn creates the light animation upon the meteor's impact. A custom **Output Event Handler Light Update** script modifies the volumetric scale, brightness, and color to add more dramatic flair to the collision.

VFX Graph Output Event



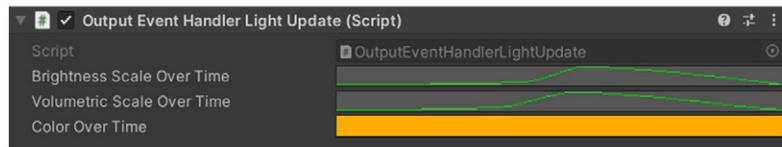
GameObject

Output Event Handler
spawns Prefab



Prefab instance

Light animation
from another
Output Event Handler



The custom Light Update Output Event Handler



A light reacts to the meteor impact effect.

Begin by exploring a few of these ideas. You can use the sample scripts directly without writing any code or treat them as a starting point for your own scripts. Give it some time, and soon you'll be able to roll your own Output Event Handlers for whatever your application requires.

PIPELINE TOOLS



Effects aren't isolated in a vacuum. Often you'll need to supply them with external data to achieve your intended look.

What if you want the genie to emerge from a magic lamp? Or you'd like to integrate a hologram with the sci-fi spaceship? Though you can accomplish much of this with math functions and Operators, you might need the effect to interact with more complex shapes and forms.

For this reason, Unity provides support for a number of Data types:

- **Point Caches:** Store attributes of points in space, such as Transforms, normals, colors, and UVs.
- **Signed Distance Fields:** Attract and collide with particles using a volumetric representation.
- **Vector Fields:** Push particles in 3D space after sampling the particle's position.

Unity also offers some support utilities to facilitate the generation of these file formats.

Point Caches

A [Point Cache](#) is an asset that stores a fixed list of **Particle Attribute** data, including points and their positions, normals, and colors.

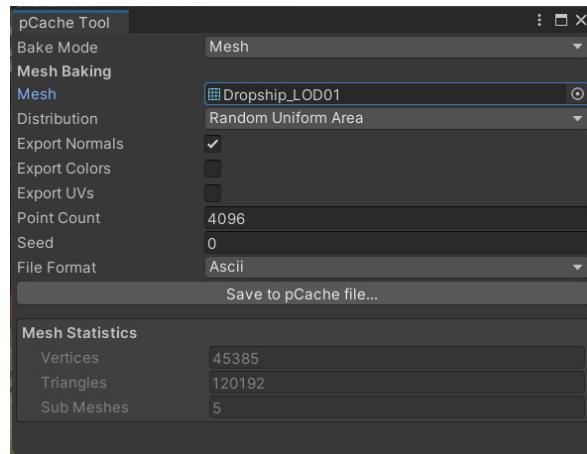
Point Cache assets follow the open-source [Point Cache specification](#) and use the **.pCache** file extension. Internally, Point Caches are nested [ScriptableObjects](#) containing various textures that represent the maps of Particle Attributes. They are less resource intensive than Signed Distance Fields.

To generate a Point Cache for use in a visual effect:

- Use the built-in [Point Cache Bake Tool](#) via **Window > Visual Effects > Utilities > Point Cache Bake Tool**.
- Select the **Houdini pCache Exporter** bundled with [VFXToolbox](#).
- Build your own custom exporter. See the [pCache README](#) for more information on the asset format and specification.

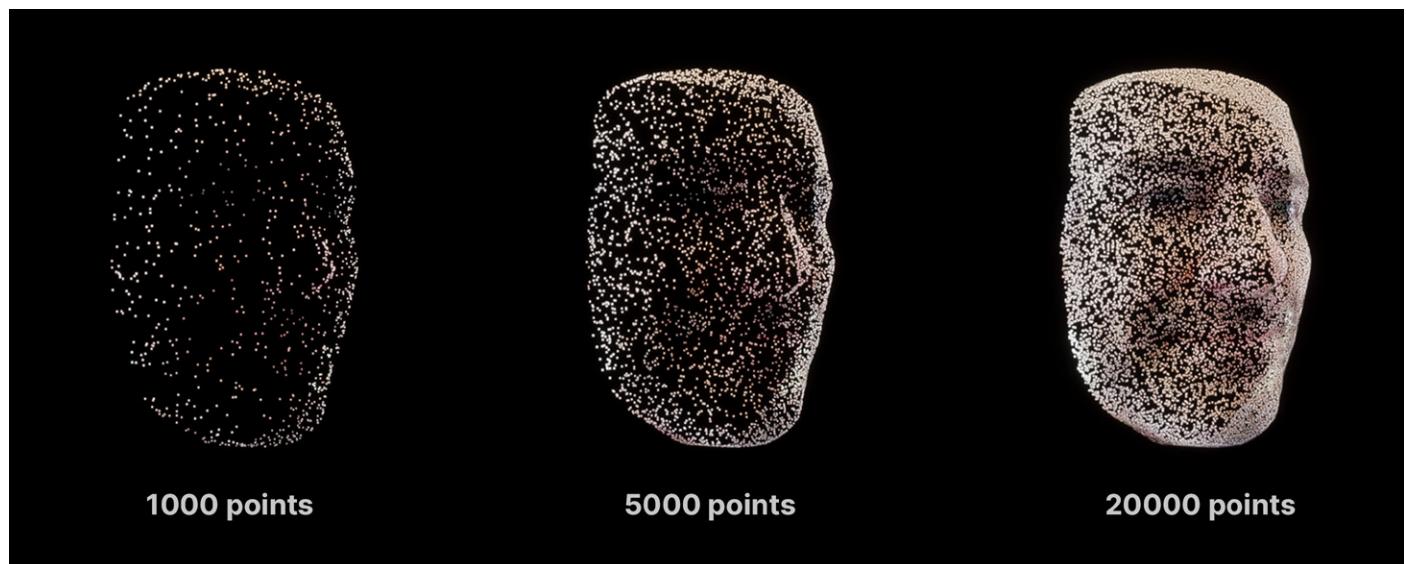
Point Cache Bake Tool

Use the Point Cache Bake Tool to create a Point Cache from an input [Mesh](#) or a [2D Texture](#).



The Point Cache Bake Tool

Point Caches store lists of points generated from 3D meshes or 2D textures – but not their actual geometry. During baking, additional filtering relaxes the points in order to separate them more evenly and reduce the number of overlaps. Choose a **Mesh** or **Texture**, set an adequate **Point Count**, and select **Save to pCache file**.



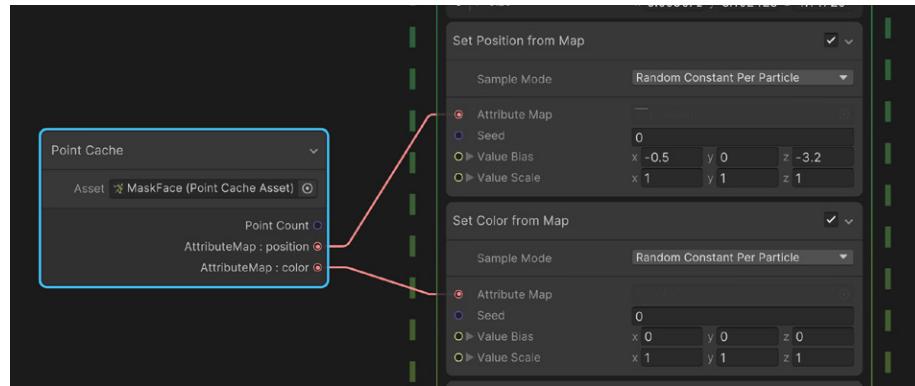
A visualization of a Point Cache with different samples

Point Caches are similar to [Stanford PLY](#) files, but the .pCache file format removes the polygons and adds support for vectors. As such, they are more easily readable and writable in Python or C#.

Using Point Caches

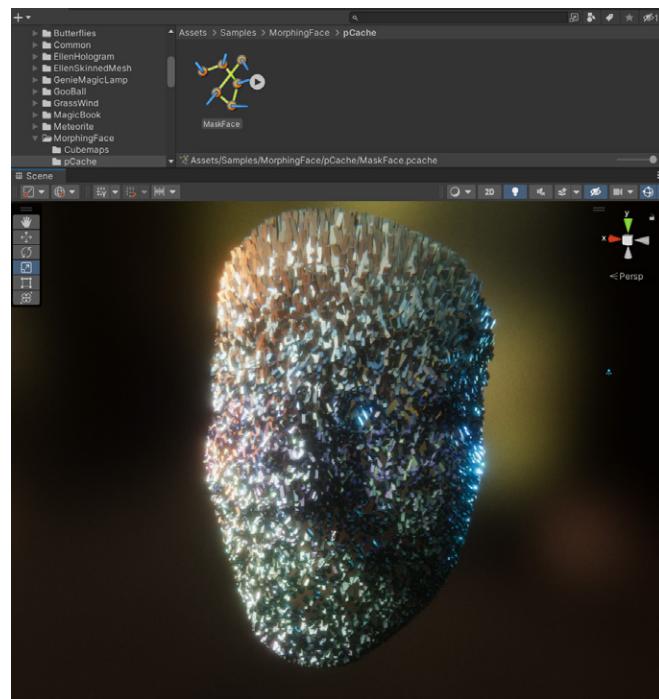
In the VFX Graph, the **Point Cache Operator** extracts the number of particles and their Attributes from the Point Cache asset. It then exposes them as output ports in the Operator.

Looking at the Morphing Face sample, the Operator creates one output slot for the Point Count and separate texture slots for Attribute Maps. You can connect the outputs to other nodes, such as the **Set Attribute from Map Block**.



Using Point Caches in a VFX Graph

The **MaskFace Point Cache** drives the underlying structure of the effect.



The Morphing Face effect applied to the Point Cache asset

For more information, see the [Point Cache documentation](#).

Signed Distance Fields

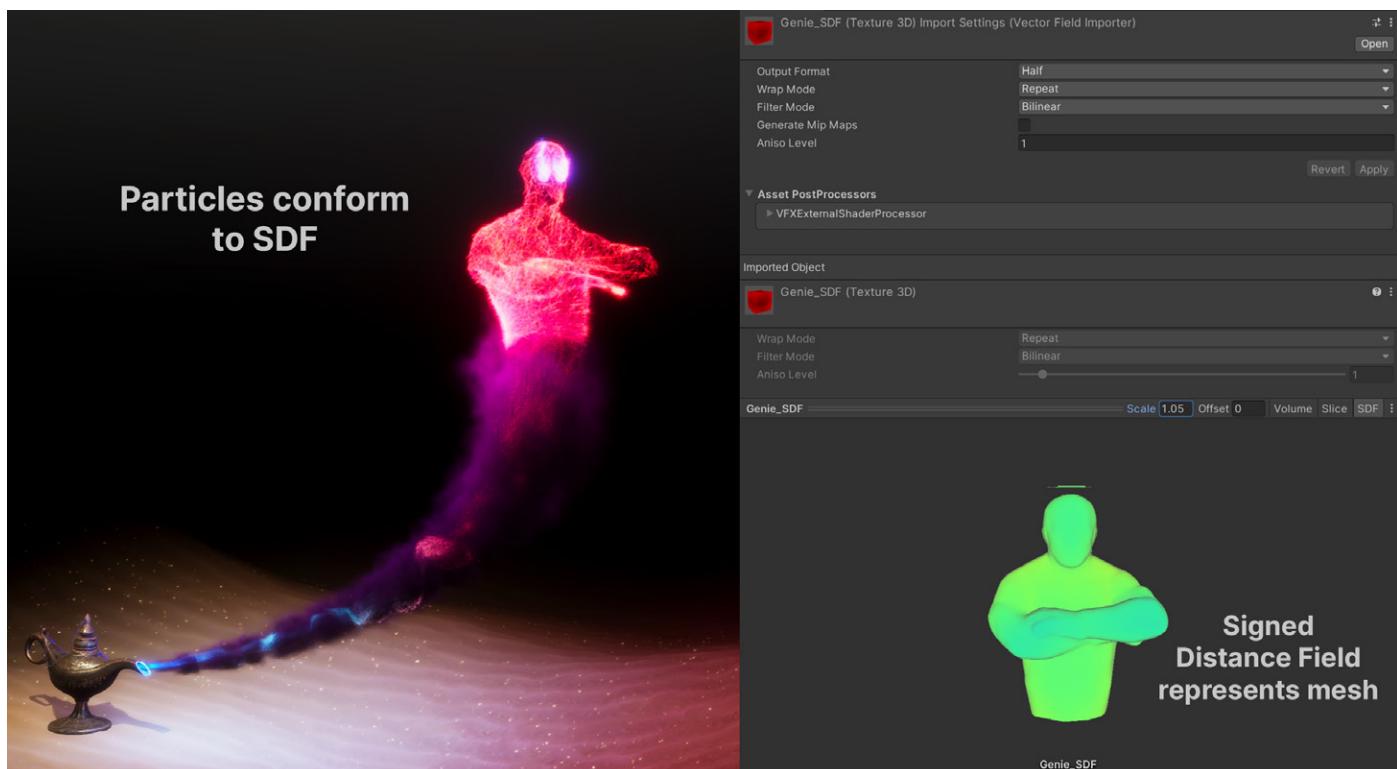
A [Signed Distance Field](#) (SDF) is a 3D texture representation of mesh geometry. Each texel stores the closest distance value to the surface of the mesh.

By convention, this distance is negative inside the mesh and positive outside of it. You can thereby place a particle at any point on the surface, inside the bounds of the geometry, or at any given distance from it.

While it's more resource intensive to calculate SDFs than Point Caches, they can provide additional functionality. Very detailed meshes require a high texture resolution, which typically takes up more memory.

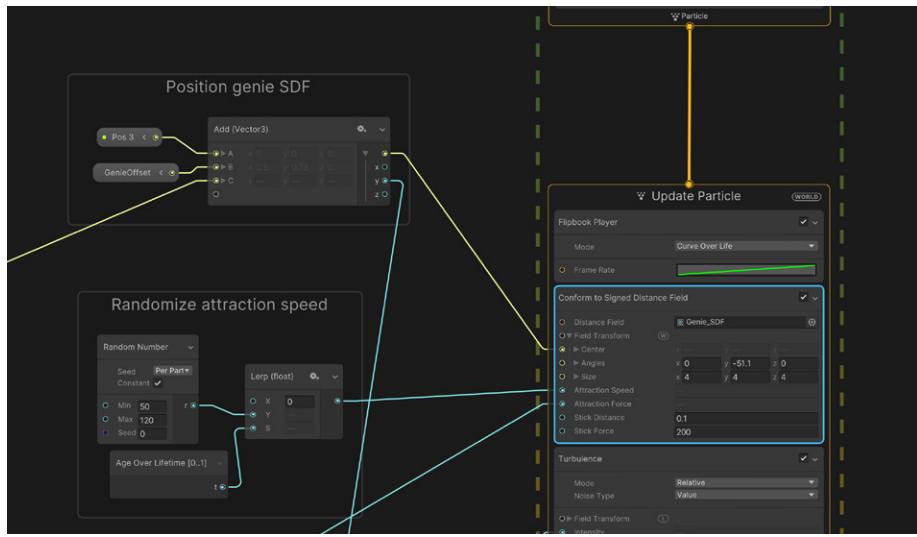
Using SDFs

A visual effect can use SDFs to position particles, conform particles to a shape, or collide with particles. In the Magic Lamp sample, an SDF was used for the genie's body. A preview of the SDF asset looks like this:



A visual representation of a Signed Distance Field

Unity imports the SDF asset as a 3D texture [Volume File](#) (.vf). Compatible VFX Graph Blocks and Operators then make the particle system interact with the sampled points. In this example, the **Conform to Signed Distance Field Block** attracts the particles to the area where the genie's torso appears.

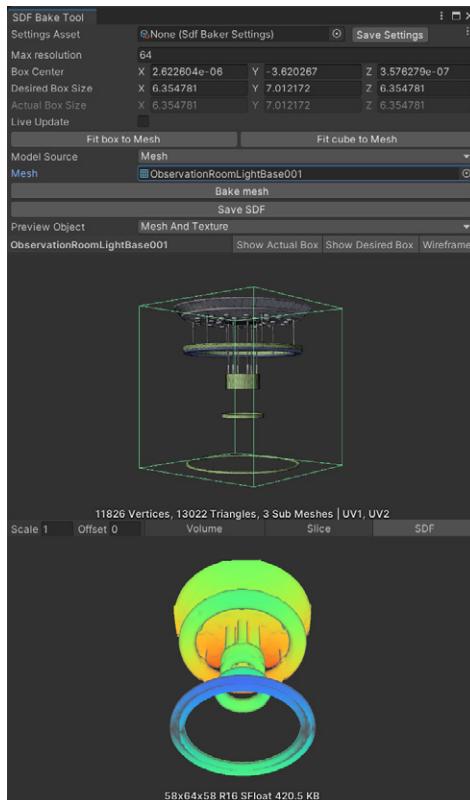


Using an SDF in a VFX Graph

SDF Bake Tool

You can use an external DCC tool, such as **SideFX Houdini**, to create SDFs. The **VFXToolbox** utility can bake Volume Files from the **Houdini Volume Exporter**.

Unity includes a utility to simplify this process: The **SDF Bake Tool** (**Window > Visual Effects > Utilities > SDF Bake Tool**) takes an input **Mesh** (or multiple Meshes in one Prefab) and generates a 3D texture representation of it.



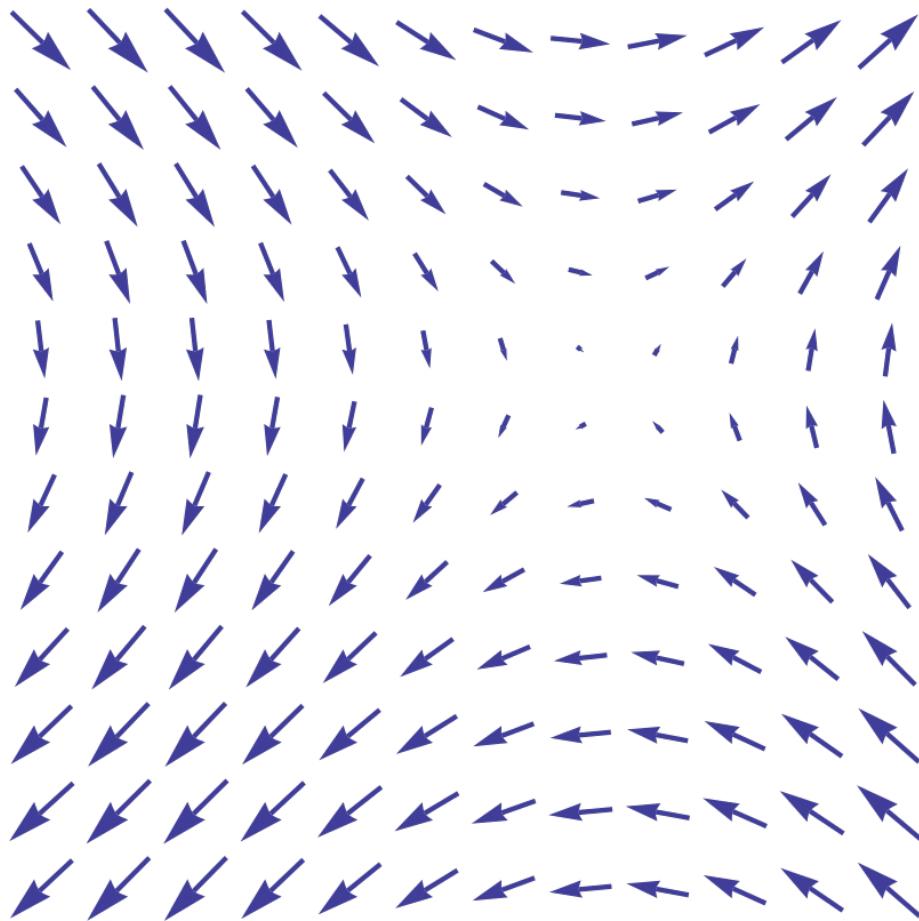
The SDF Bake Tool

You can also bake SDFs at runtime and in the Unity Editor with the [SDF Bake Tool API](#). Just be aware that runtime baking is resource intensive. Best practice is to use a low-resolution SDF and only process every nth frame.

See [Signed Distance Fields in the VFX Graph](#) for more details on how to generate and use SDFs, or find additional samples in [this repository](#).

Vector Fields

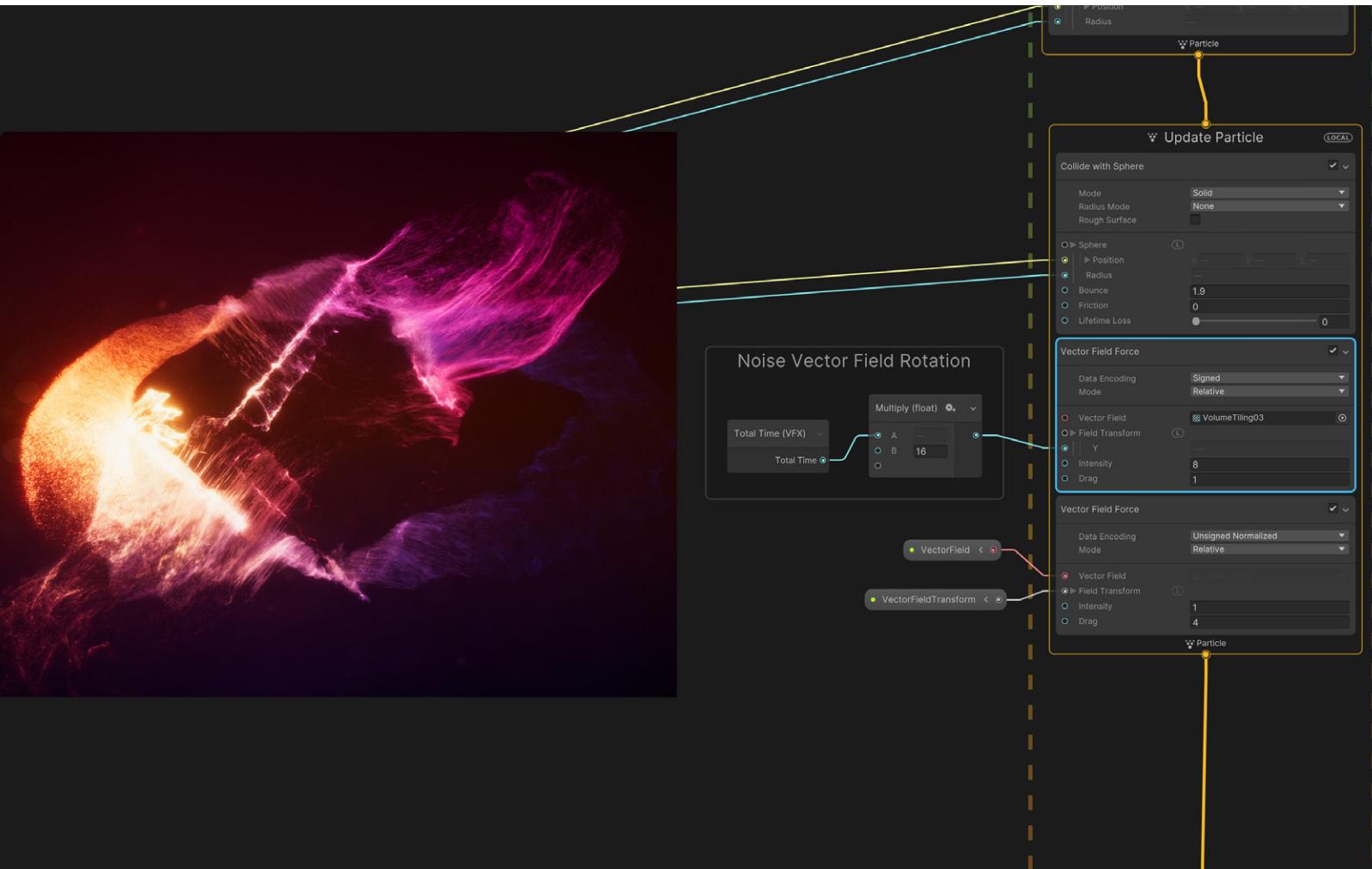
A Vector Field is a uniform grid of vectors that controls the velocity or acceleration of a particle. An arrow represents each vector. The larger the size of the vector, the faster the particles will move through it.



A 2D Vector Field (Source: [Wikipedia](#))

As with Signed Distance Fields, you can represent Vector Fields using the open-source [Volume File \(.vf\)](#) format or generate vector fields from the Houdini VF Exporter bundled with [VFXToolbox](#). You can even write your own VF File Exporter that follows the Volume File specification.

In the UnityLogo scene, the particles flow as if pushed by the unseen currents of the Vector Field.



A Vector Field drives the UnityLogo effect

VFXToolbox

The VFXToolbox features additional tools for Unity visual effects artists. It enables the export of .pCache and .vf files from SideFX's **Houdini Point Cache Exporter** and **Volume Exporter**.

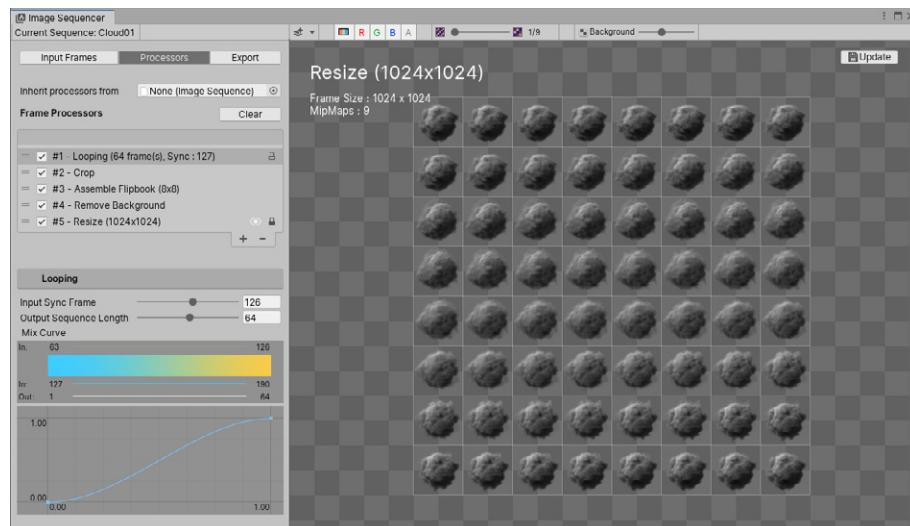
Download [this repository on GitHub](#) and install it with the Package Manager.

Image Sequencer

Use **Flipbook Texture Sheets** to bake animated effects into a sprite. If you don't have the frame budget to simulate effects like smoke, fire, or explosions, saving the images as a Flipbook Texture Sheet can produce a comparable "baked" effect without the high cost.

First, use Unity or another DCC package to render an image sequence of effects into a project folder. Next, convert the individual images into a single texture sheet using the Image Sequencer. Retime and loop the images to your liking before playing them back with the [Flipbook Player Block](#).

Be sure to check out some of the [sample flipbooks](#) created with this tool.



An example of a flipbook texture

TFlow (Asset Store)

TFlow, available on the [Unity Asset Store](#), is a "motion vector and motion blur generator that helps increase the utility and quality of your flipbooks." It can be used with both the [Built-in Particle System](#) and [VFX Graph](#).

Digital Content Creation tools

While Unity is a central tool for game development, it's not your only tool. Creating real-time visual effects can be complex, so you might look into specialized assistance from Digital Content Creation (DCC) software outside of Unity. Here are a few tools that many artists use to complement Unity and the VFX Graph.

SideFX Houdini

Houdini has long been an industry-standard tool for simulation and visual effects. Its procedural workflows and node-based interface facilitate the production of textures, shaders, and particles in comparatively fewer steps. Its Operator-centric structure encourages nonlinear development and covers all the major areas of 3D production.

Autodesk Maya

Maya fortifies the foundation of many game development studios. Its relatively new Bifrost system makes it possible to create physically accurate and incredibly detailed simulations in a visual programming environment.

Blender

Blender is a free and open-source 3D creation suite. Its features cover all aspects of 3D production, from modeling and rigging to animation, simulation, and rendering. Blender continues to receive widespread community support, as it is cross-platform and runs equally well on Linux, Windows, and macOS.

Adobe Photoshop

Along with the other 3D tools discussed, you'll benefit from an image-editing software such as Adobe Photoshop. Use Photoshop to edit and create raster images in multiple layers, and support [masks](#), [alpha composites](#), and several [color models](#). Photoshop uses its own PSD and PSB file formats to uphold these features.

Of course, these are just some of the DCC tools available. As you start building your graphs, they'll help you fill in flipbook textures, meshes, or anything else to achieve your vision.

OPTIMIZATION

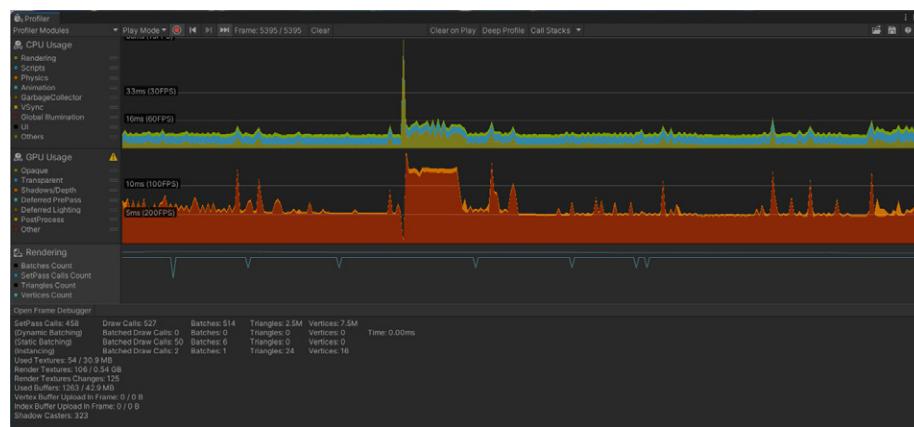


After working closely with VFX Graphs, you'll likely want to reorganize and optimize them, much like how a programmer profiles code and checks its performance. Once the effect looks right, make sure it's not using excess resources before deploying to your final game or application.

The Unity Profiler and Frame Debugger

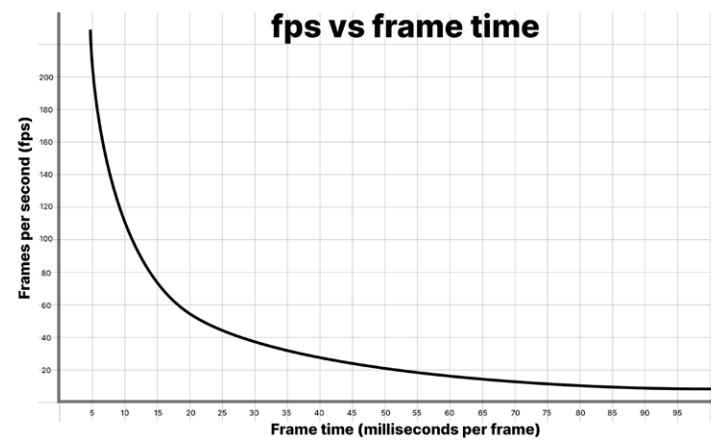
The [Unity Profiler](#) (**Window > Analysis > Profiler**) and [Frame Debugger](#) (**Window > Analysis > Frame Debugger**) can be used to optimize your graphs for stronger performance. However, the Unity Editor can affect your profiling information, leading to inaccurate results.

Use the **Profiler Standalone Process** option or create a separate build when you need to measure real-world performance. Consider the [fundamentals of graphics](#) performance to maintain high frame rates, and in turn, deliver the best possible experience to your players.



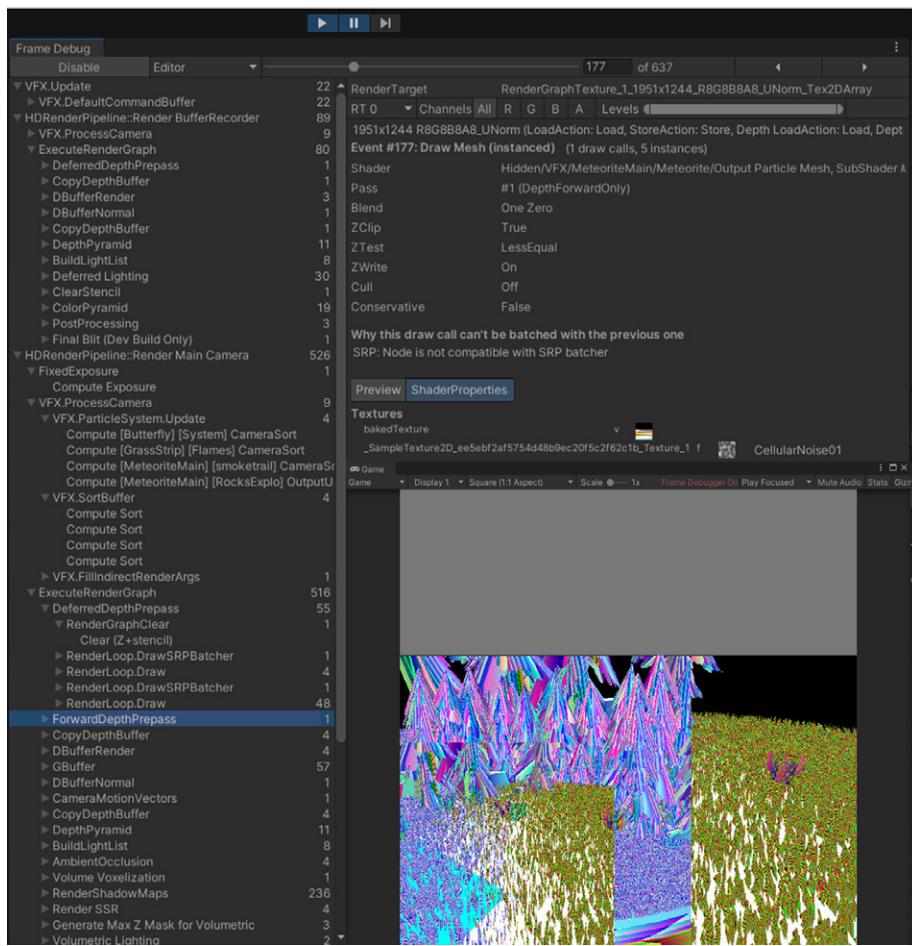
The Unity Profiler

When examining [rendering statistics](#), take note of the time cost per frame rather than frames per second. The fps can be misleading as a benchmark because it's nonlinear (see the graph below). If you're aiming for 60 fps, use 16 ms per frame as your frame budget (or 33 ms per frame for 30 fps).



Use the frame time as a guide for optimization.

The Frame Debugger shows draw call information, so you can control how the frame is constructed.



The Frame Debugger

In the image above, the left panel shows the sequence of draw calls and other rendering events arranged hierarchically. Meanwhile, the right panel displays the details of a selected draw call, including shader passes and textures. This helps you play “frame detective” and find out where Unity is spending its resources.

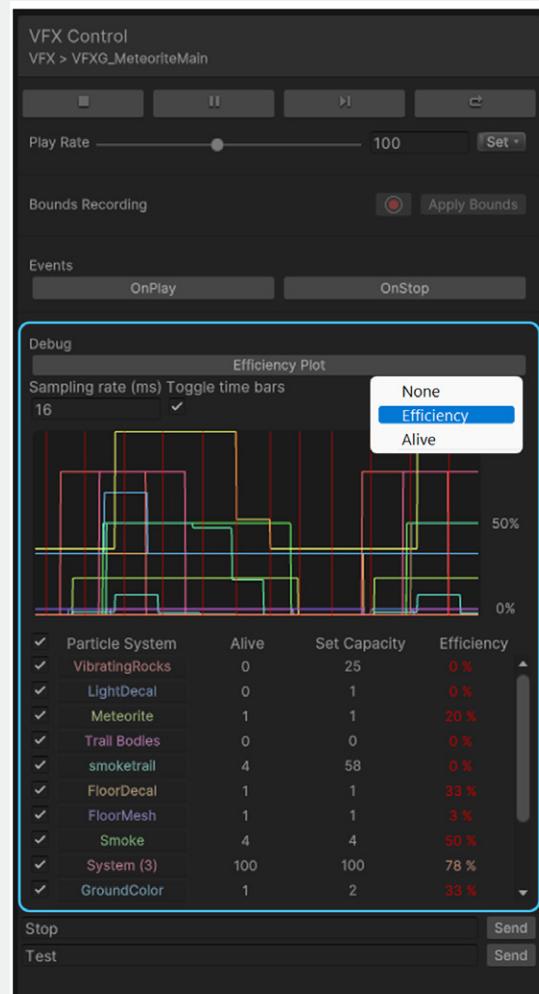
Remember to go through the usual suspects when optimizing your effects:

- **Texture size:** If the asset doesn’t get close to the Camera, reduce its resolution.
- **Capacity:** Fewer particles use less resources. Set the Capacity in the Initialize Block to cap the System’s maximum number of particles.
- **Visibility and lifetime:** In general, if you can’t see something onscreen, turn it off.

Debug modes

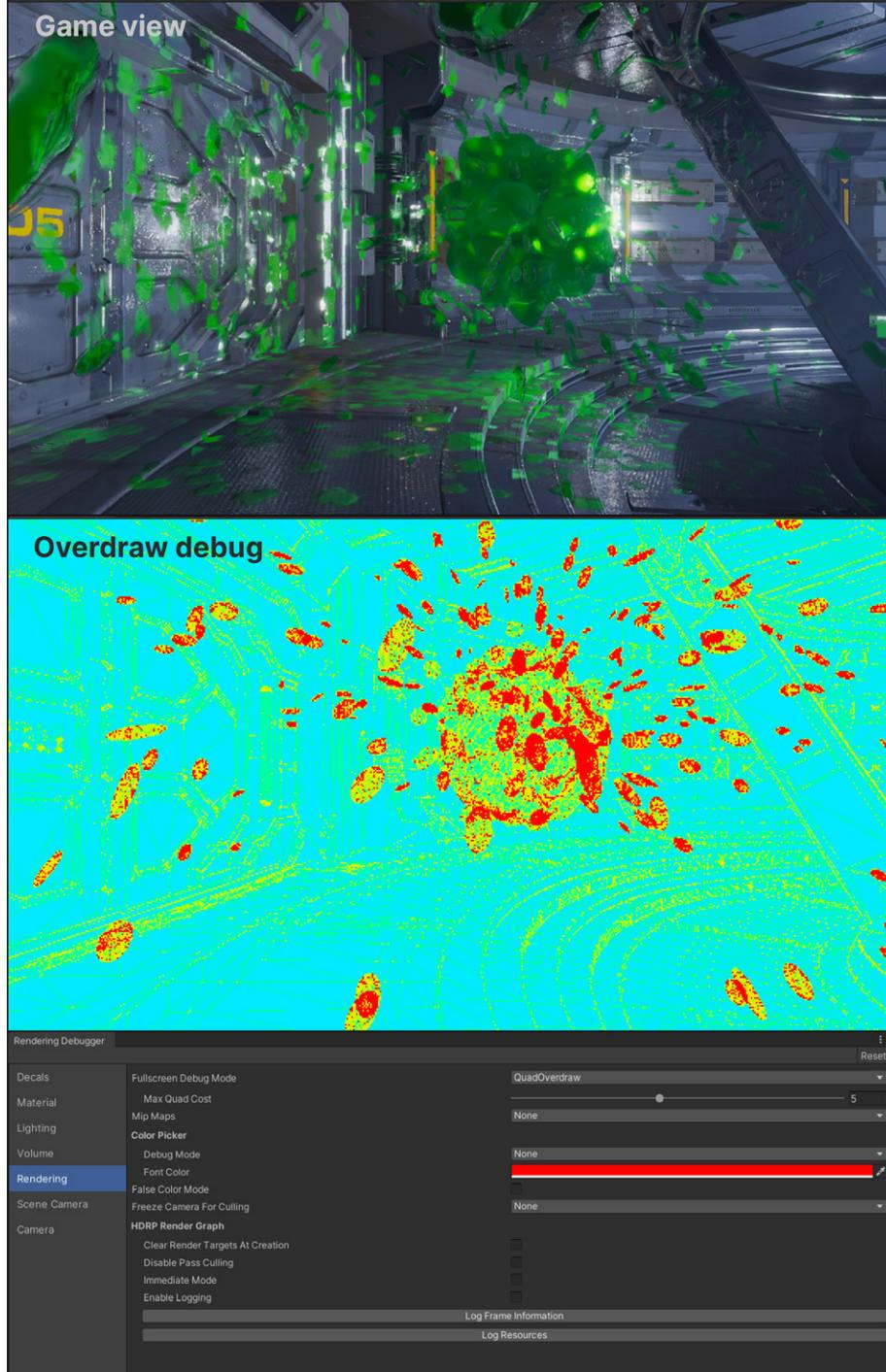
The VFX Control panel includes Debug modes you can use to determine particle lifetime and capacity, which can influence performance and memory usage alike. Edit a Visual Effect instance from the Inspector, then set the Debug option to **Alive** or **Efficiency**.

The resulting plots will show how many particles are alive, or how that count compares to the System's set capacity. Adjust your count and capacity settings to improve your VFX Graph's efficiency.



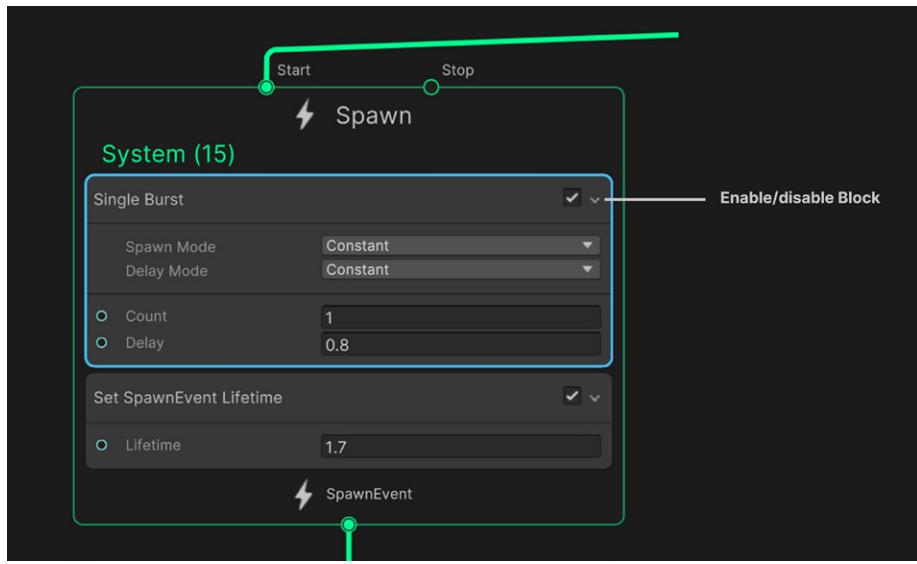
- **Operators and memory:** Simplify unnecessary Operators. If not visibly different, use fewer iterations.
- **Flipbooks:** Rather than simulate every particle, consider pre-rendering certain effects into texture flipbooks. Then, play back the animated texture wherever the full simulation isn't necessary.

- **Mesh size:** If your particles are Output Meshes, be sure to reduce your polygon counts.
- **Excessive overdraw:** If you have a number of transparent surfaces, they will consume your rendering resources. Use the Rendering Debugger (**Window > Analysis > Rendering Debugger**) to check excess overdraw and tweak your graph accordingly. Also, switch to octagon particles when possible.



Minimize overdraw (red) to improve performance

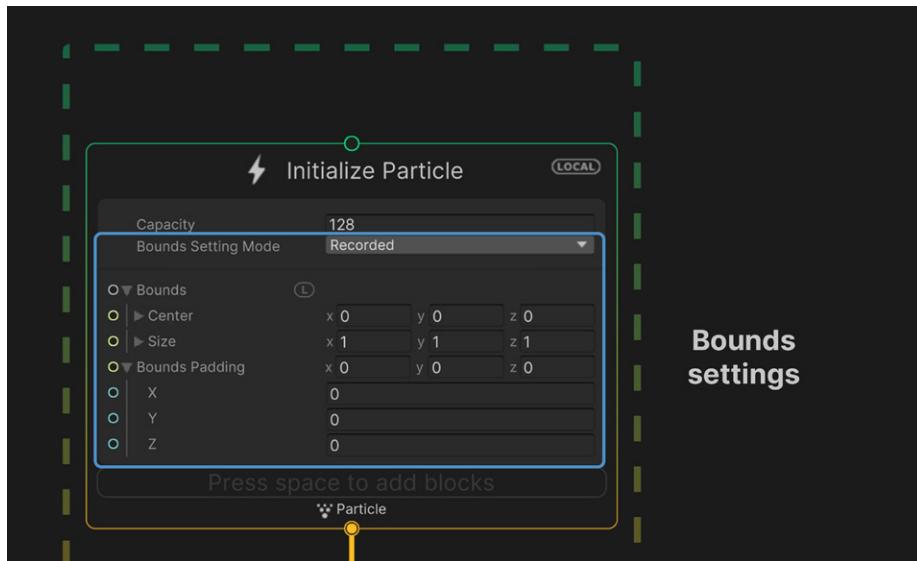
When troubleshooting performance, enable or disable each Block with the checkbox at the top-right corner. This lets you do quick A/B testing to measure performance (before and after) so you can isolate part of your graph. Don't forget to restore your Blocks to their Active state once complete.



Disable/enable a Block for testing.

Bounds

The Bounds of visual effects comprise a built-in optimization based on visibility. You've probably noticed a few settings that appear in every Initialize Context:



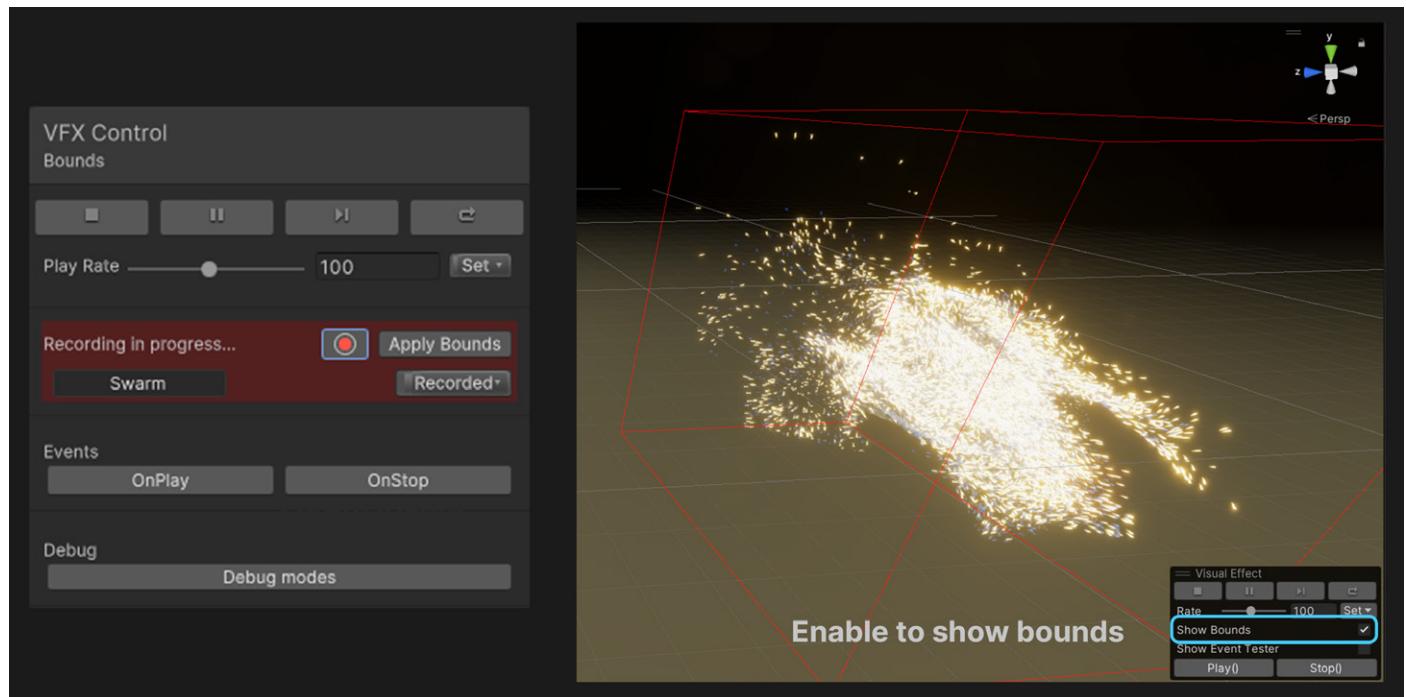
Use the Bounds settings to define where your effect will render.

If the Camera can't see the Bounds, Unity culls the effect, meaning that it doesn't render. Follow these guidelines to set up each System's Bounds:

- If the Bounds are too large, cameras will process the visual effects even if individual particles go offscreen. This wastes resources.
- If the Bounds are too small, Unity might cull the visual effects even if some particles are still onscreen. This can produce visible popping.

By default, Unity calculates the Bounds of each System automatically, but you can change the Bounds Setting Mode to:

- **Automatic:** Unity expands the Bounds to keep the effect visible. If this option is not the most efficient, use one of the other options below to optimize your Bounds.
- **Manual:** Use the Bounds and **Bounds Padding** to define a volume in the Initialize Context. This is simple yet time-consuming to set up for all of your Systems.
- **Recorded:** This option allows you to record the Bounds from the VFX Control panel. The Bounds, shown in red when recording, expand as you play back the effect. Press **Apply Bounds** to save the dimensions.

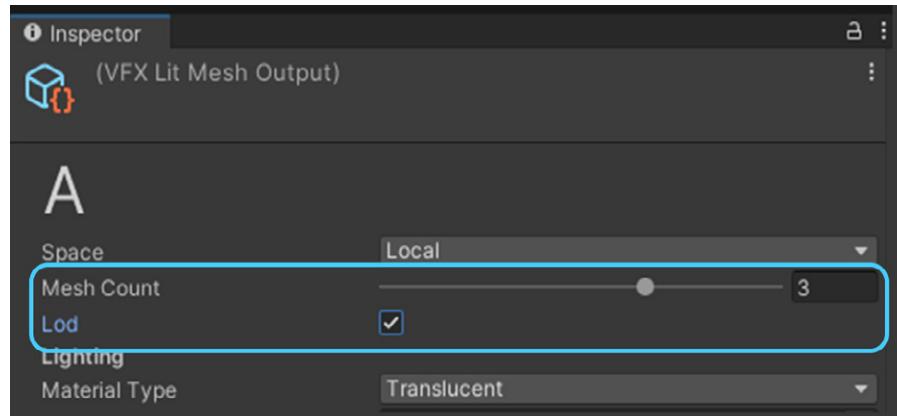


You can use Operators at runtime to calculate the Bounds for each System in **Manual** or **Recorded** mode. The Initialize Context contains a Bounds Padding input; use this Vector3 to enlarge the Bounds' values.

Mesh LOD

Take advantage of **level of detail** (LOD) if your particles are outputting meshes. Here, you can manually specify simpler meshes for distant particles.

Particle Mesh Outputs have a **Mesh Count** parameter visible in the Inspector, which lets you specify up to four meshes per output. When you combine this with the LOD checkbox, you can automatically switch between meshes based on how large they appear onscreen.



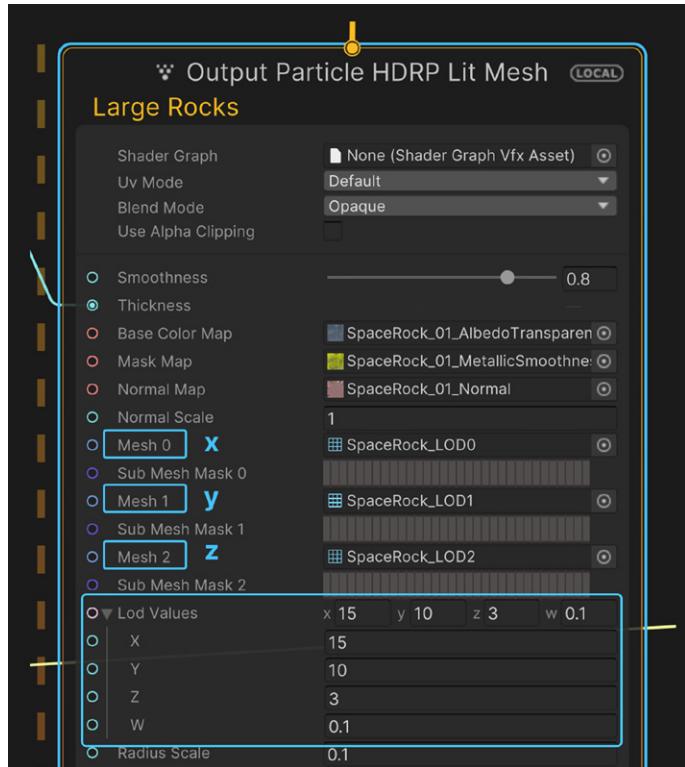
Mesh Count and LOD settings in the Inspector

Higher resolution models can hand off to lower resolution models, depending on the screen space percentage in the LOD values of the Output context.



LOD resolutions

In this example, the SpaceRock_LOD0 model swaps with the smaller SpaceRock_LOD1 model when the mesh occupies less than 15% of the screen.



LOD values

When creating a massive number of mesh particles, you won't need to render millions of polygons. This significantly cuts down the frame time.

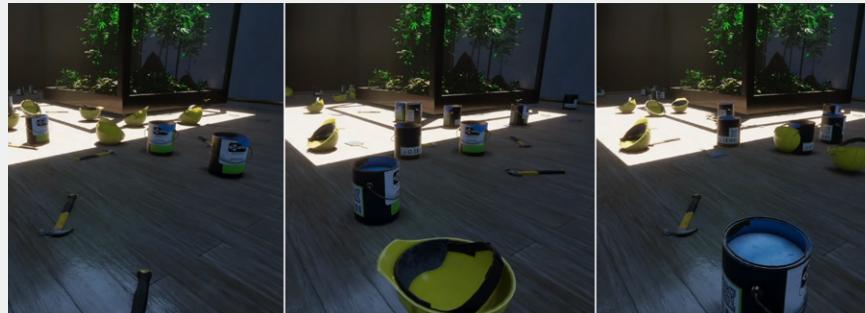


Mesh LODs save rendering resources.

Check out the PlanetaryRing example in [this project](#) to see the Mesh LOD firsthand.

Mesh Count

You can similarly leverage the Mesh Count without LOD. In this case, we use multiple meshes to add randomness: The four different meshes for the Output Particle Lit Mesh create a variety of props scattered on the floor.

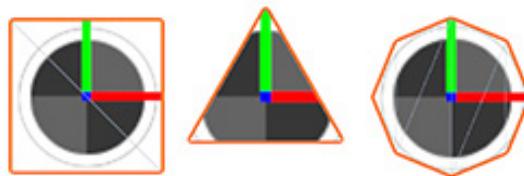


The Mesh Count randomizes mesh particles.

Particle rendering

To hit your target frame rate and frame budget, consider these optimization tips when rendering particles or meshes:

- **Triangle particles:** With half the geometry of quad particles, these are effective for fast-moving effects and rendering large quantities of particles.
- **Simplified lighting:** If you don't need the full [Lit HDRP](#) shader, switch to a less resource-intensive one. Customize outputs in Shader Graph to drop features you don't need for certain effects. For example, the Bonfire sample scene uses a stylized Shader Graph, which greatly simplifies the output.

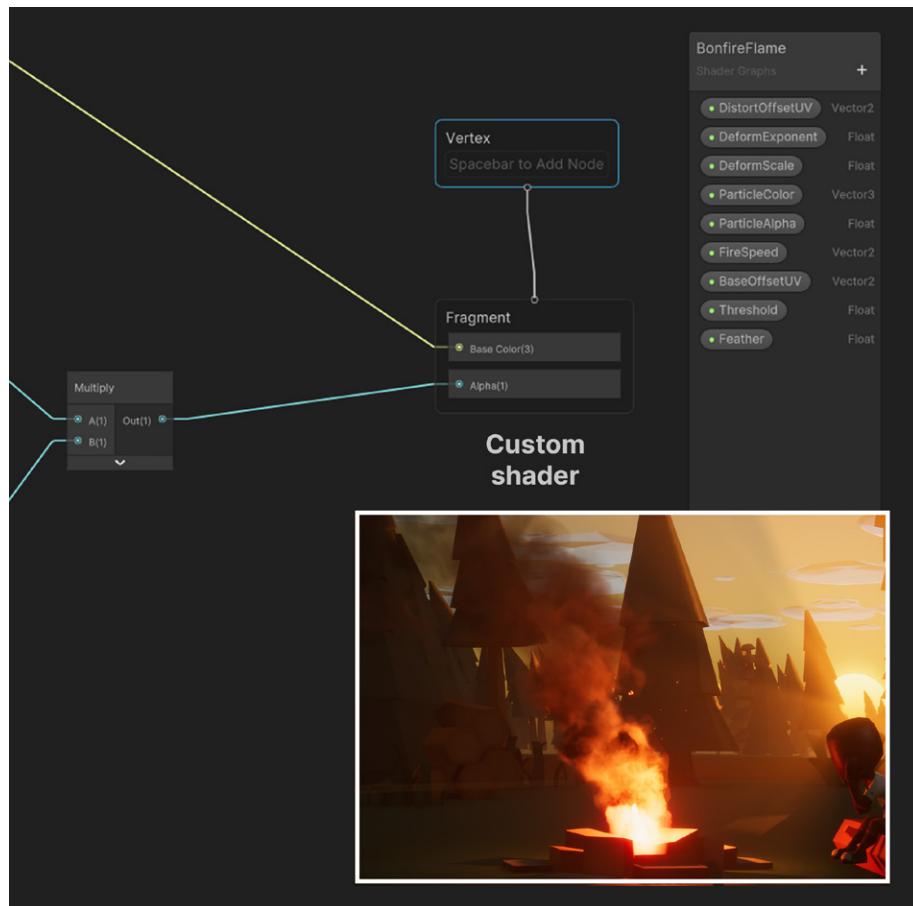


Output Particle Primitives (quad, triangle, octagon)

The following tips apply to HDRP only:

- **Low resolution transparency:** In your [HDRP Rendering](#) properties, enable Low Res Transparency to render your transparent particles at a lower resolution. This will boost performance by a factor of four at the expense of a little blurriness. When used judiciously, it can be nearly indistinguishable from rendering at full resolution.

- **Octagon particles:** Octagon particles crop the corners of quad particles. If your particle textures are transparent in the corners, this technique can reduce or prevent overdraw. Overlapping transparent areas still requires some calculation, so using octagons can save unnecessary work computing where the corners of quads intersect.



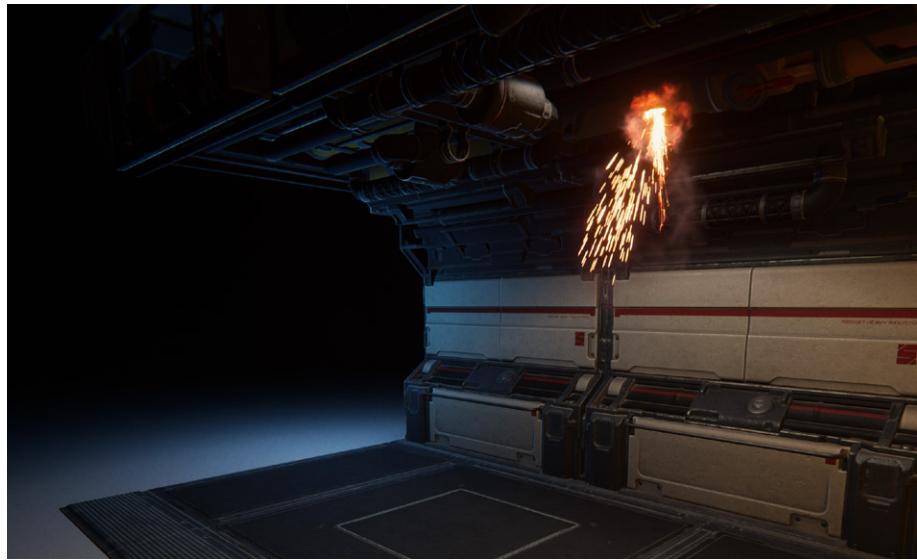
The Bonfire scene from the Visual Effect Samples

Case study: Spaceship optimization

Let's turn back to the Spaceship Demo. In this project, the Sparkle effect from the doomed spaceship illustrates how you can store data within a texture for optimization and reuse.

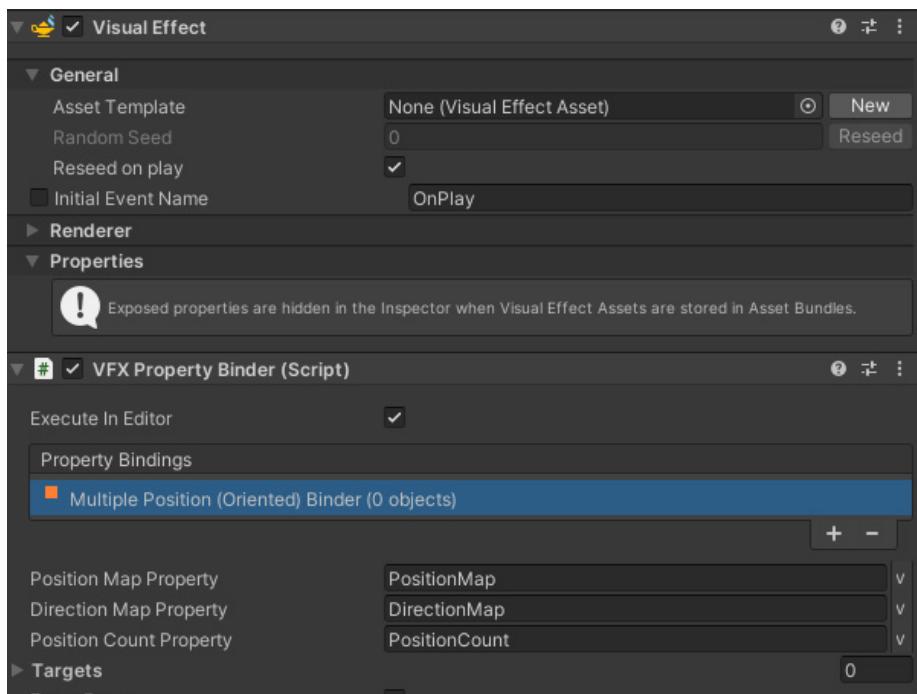
As the ship's integrity begins to fail, the tension is heightened. The camera shakes violently while the sparks fly off the interior bulkhead. The core nears depletion and the sparks appear more frequently.

Remember that even just a few particles for each spark will cost resources. Instantiating a few dozen sparks can generate hundreds of extra draw calls.



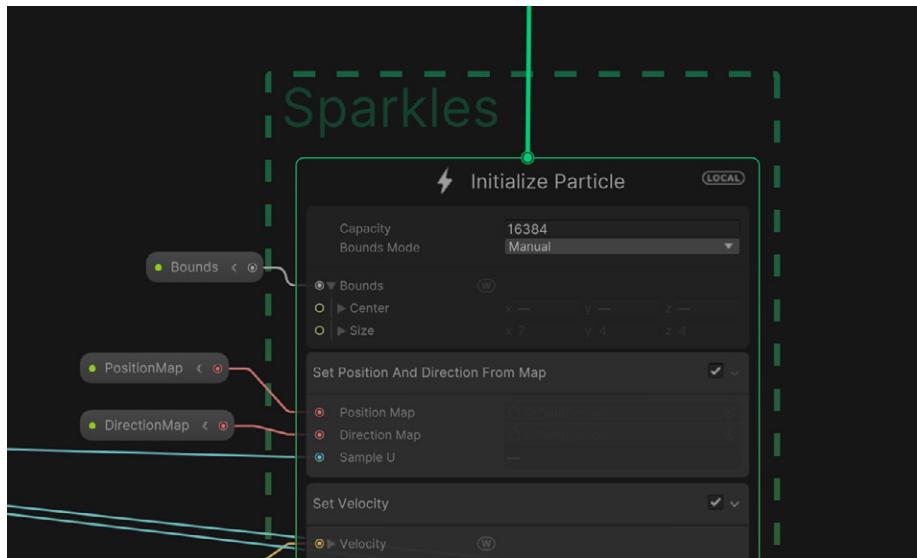
Particles show the spaceship in danger

This situation might be fine while prototyping, but any significant inefficiencies should be rectified in your graphs before your game application ships.



A custom VFX Binder

The solution for the Spaceship project is to use a custom **VFX Binder** – in this example, **MultiPosOrientedParameterBinder**. This custom component converts a large number of Transforms into a **Point Cache Map**. You can then access the data contained in the map.



Accessing the Point Cache Map to optimize VFX in the project

A single simulation can recreate the same Sparkle effect's timing and placement by accessing the **Position Map** and **Direction Map** in the **Initialize Particle**

Context. The result? Hundreds of fewer draw calls and only a handful of VFX instances. Compare the **SparkleBurst_Single** and **SparkleBurst_Shake** graphs in the Spaceship Demo project to see the differences in implementation.

Storing data in textures is a common optimization technique for real-time effects. If you're using Unity 2021 LTS or newer, harness its support for experimental **Graphics Buffers** when moving large amounts of data (see section below).

Watch [VFX Graph: Building visual effects in the Spaceship Demo](#) for more on how to customize a VFX Binder.

NEW AND FUTURE DEVELOPMENTS

SN 3852000-1

The VFX Graph and Unity's other real-time graphics tools continue to evolve alongside the needs of the digital artist community. If you want to experiment with these updates, the majority of them are already available for testing with Unity 2021 LTS.

Note: Please keep in mind that many of these features are still in development.

Universal Render Pipeline

Unity 2021 LTS brings URP support to the VFX Graph. This allows effects to be deployed on a range of compute-capable mobile devices and XR platforms.

Lit output

VFX Graph now supports Lit outputs in URP. Use them to create effects that respond directly to the scene's lighting.



URP support now includes Lit outputs.

2D Renderer support

VFX Graph in Unity 2021 LTS adds basic support for the URP's 2D Renderer. It's now possible to render effects in a 2D project and sort them along with sprites in your scene. The current implementation works with 2D Unlit shaders (with future support for [2D Lit shaders](#) on the roadmap).



VFX Graph Rain effects with the URP 2D Renderer

Note: The VFX Graph requires compute shaders. Compute support on mobile devices varies widely across brands, mobile GPU architecture, and operating systems. Unity's [Built-in Particle System](#) is recommended if your platform does not support compute shaders.

Graphics Buffer support

Unity 2021 LTS adds support for Graphics and Compute Buffers. This makes it easier to handle and transfer large amounts of data from C# to a VFX Graph.

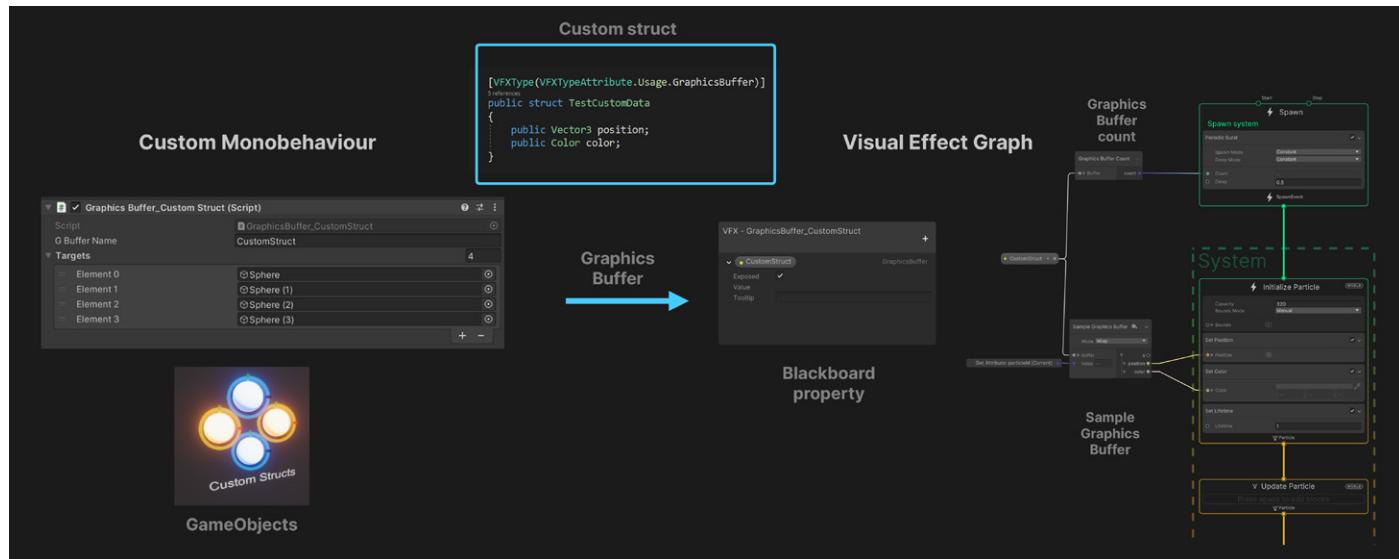
If you are tracking multiple GameObject positions in your graph, you can make that information accessible to your simulation via scripting. Graphics Buffers can similarly replace storing data within a texture, as seen in some of the previous samples.

In this example, we pass the GameObjects' Position and Color data from built-in Types, custom structs, and compute shaders to a VFX Graph.



Graphics Buffers send the GameObject data to the VFX Graph.

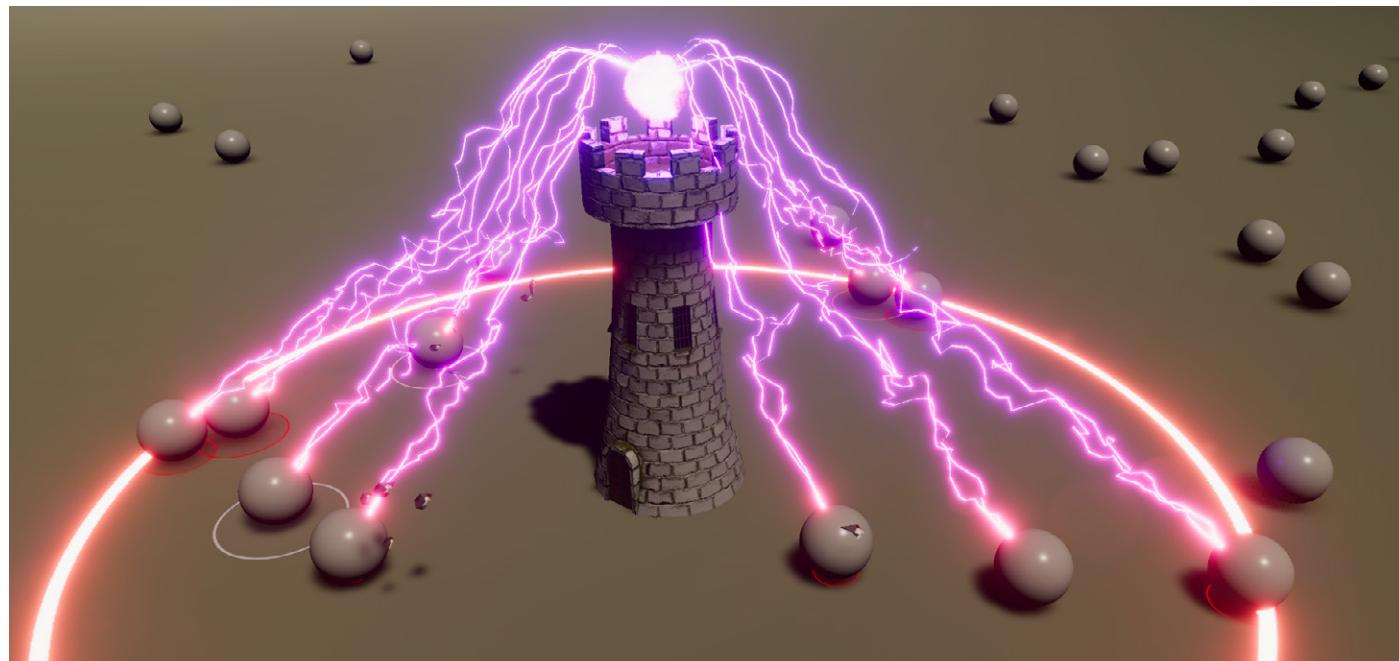
We use a script to define a Graphics Buffer and fill it with data from our GameObjects. We then pass it into the VFX Graph through its Blackboard properties.



Passing data via custom struct

In this more complex example, an electrifying tower can access the approaching sphere positions via Graphics Buffers. With many spheres, it becomes impractical to expose a Property for every GameObject.

The Blackboard has just one custom struct for use within a Graphics Buffer. The tower can potentially hit hundreds of targets, accessing their data with just a few Operators.



Graphics Buffers in action

This demonstrates how your VFX Graphs can interact within your scene; think of complex simulations like boids, fluids, hair simulation, or crowds. While using Graphics Buffers requires knowledge of the C# API, they make trading data with your GameObjects more convenient than ever.

Take a look at [this project](#) for other examples of how to use Graphics Buffers with the VFX Graph in Unity 2021 LTS. For more details on what's to come in future releases, visit the Unity [Graphics product roadmap](#).

Released - 2021.2

URP Support (Compute Capable Devices only):
2D renderer support



URP Support (Compute Capable Devices only):
Lit particles and various features and fixes



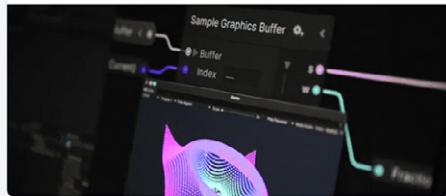
New Shader Graph integration: Modify
particle's Vertex position/normal



Support all HDRP materials with VFXGraph
using ShaderGraph.



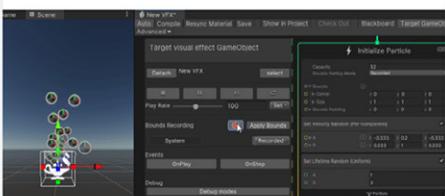
Graphics Buffer Support



Signed Distance Field Baker



Bounds and Capacity Helpers



HDRP Decal Output



Improved node searcher

Improved results order when searching nodes..

The Graphics product roadmap

The background of the image is a scenic landscape featuring a large, dark mountain range on the left, a bright blue lake in the center, and a dense forest of tall evergreen trees on the right. The sky is a clear, pale blue.

ADDITIONAL RESOURCES

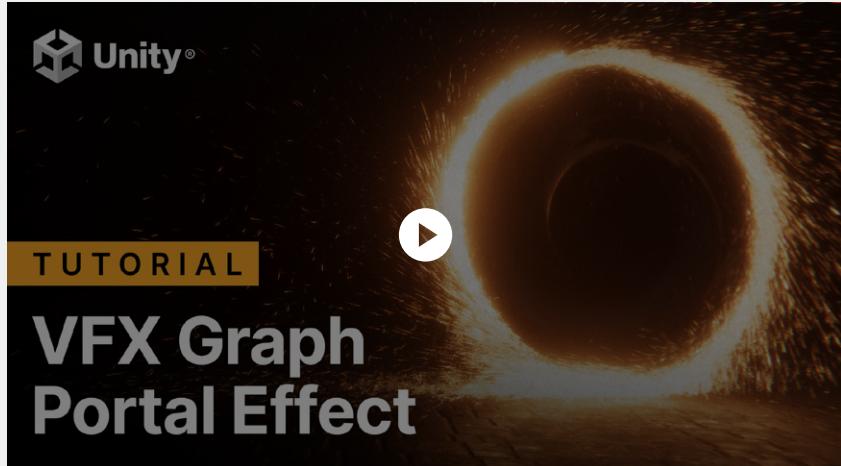
We hope that this guide has inspired you to dive deeper into the VFX Graph and Unity's real-time visual effects toolsets. After all, our mission is to help every creator achieve their artistic vision.

With the VFX Graph, you're fully equipped to captivate your players with hyperrealistic simulations and stunning graphics. We can't wait to see what you create with it.

In the meantime, here is a collection of additional learning resources for taking on the VFX Graph.

Video tutorials

- [Create amazing VFX with the VFX Graph](#): This covers many of the fundamentals for setting up your own VFX Graph.
- [The power for artists to create](#): This video highlights recent updates to VFX Graph features, such as Mesh LODs, Graphics Buffers, and Shader Graph integration.
- [VFX Graph: Building visual elements in the Spaceship Demo](#): This session unpacks a number of techniques used in the Spaceship Demo.
- [Build a portal effect with VFX Graph](#): Generate a portal effect by using the VFX Graph to transform a ring of particles into a more dynamic effect.



VFX projects on GitHub

Explore more possibilities with the VFX Graph, such as audio reactive effects, data visualization, and point cloud or volumetric data playback. Check out these projects created by Keijiro Takahashi, senior creator advocate at Unity:

- Camera tracking + Lidar + VFX Graph: [Here](#) and [here](#)
- Point Cloud (pcx) + VFX Graph
- Azure Kinect + VFX Graph
- Intel RealSense camera + VFX Graph
- DepthKit + VFX Graph
- DepthKit Volumetric video + VFX Graph
- 4DViews Volumetric video + Alembic + VFX Graph
- Alembic + HAP video + VFX Graph
- Audio Reactive + VFX Graph
- VFX Graph + Midi controllers
- Skinned Mesh sampling
- SDF + VFX Graph
- Vertex Animation Texture (VAT) + VFX Graph
- Graphics Buffer test
- Render Geo data using Graphics Buffer + VFX Graph
- Depth of Field particle samples
- Procedural modeling using VFX Graph
- VFX Graph + Procedural ShaderGraph sprite generation shader
- VFX Graph test scenes
- VFX Graph interactive fireworks
- Sushi + VFX Graph

Professional training for Unity creators

Unity Professional Training gives you the skills and knowledge to work more productively and collaborate efficiently in Unity. We offer an extensive training catalog designed for professionals in any industry, at any skill level, in multiple delivery formats.

All materials are created by our experienced Instructional Designers in partnership with our engineers and product teams. This means that you always receive the most up-to-date training on the latest Unity tech.

Learn more about how Unity Professional Training can support you and your team.



unity.com