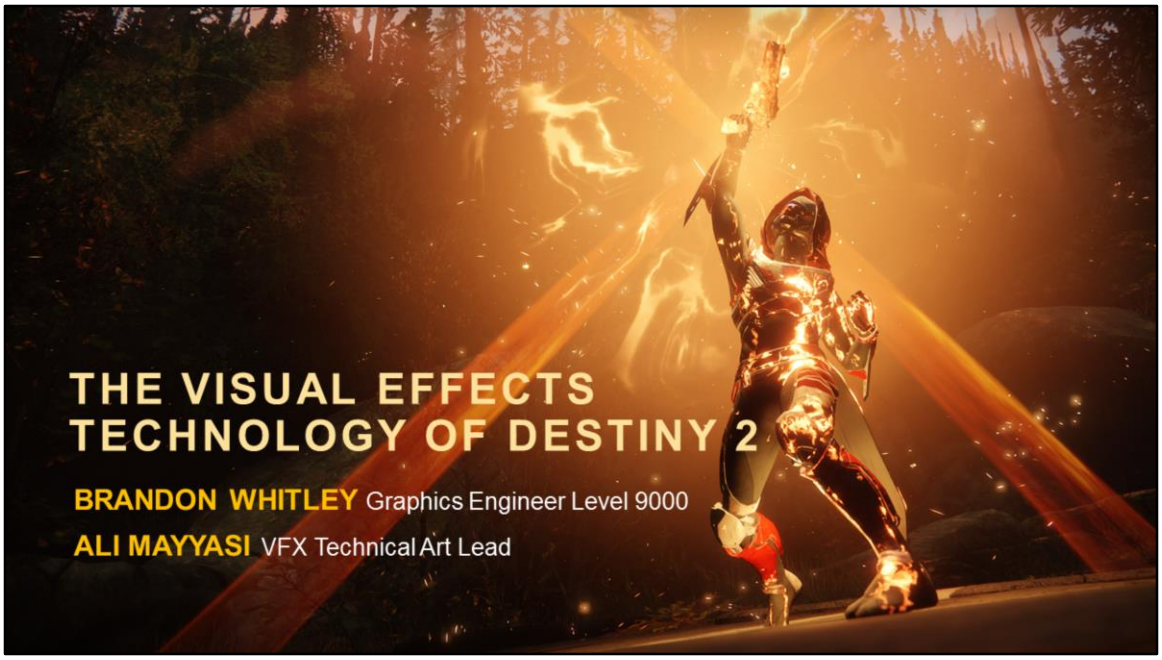


In today's talk we will cover the visual effects technology and tools of the Destiny franchise.
Destiny is a shared-world first-person shooter,
in a mythic science fiction setting.

It is a world filled with "Space Magic"

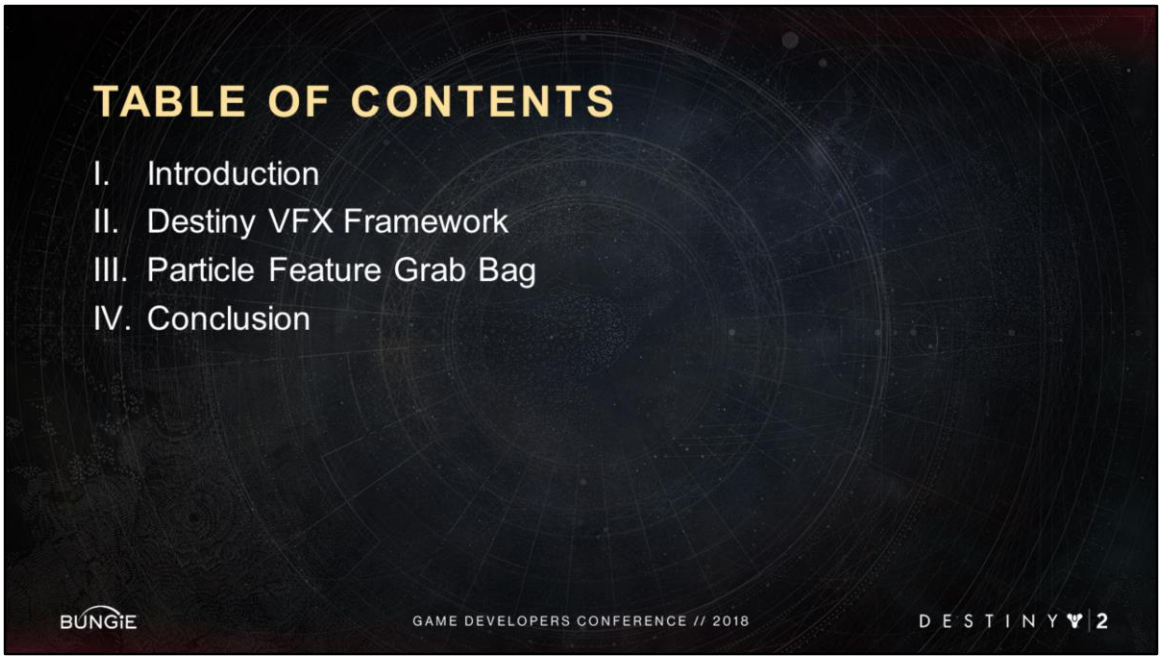
Space magic basically means a lot of FX. Critical to our game. Composed of many pieces.

Most important piece is the particles (although we will talk about a lot more than just particles today).
We want lots of interesting particle motion. Natural, random motion, yet also controllable and directable in a way that's intuitive for artists.

Connecting our fx to the game world.

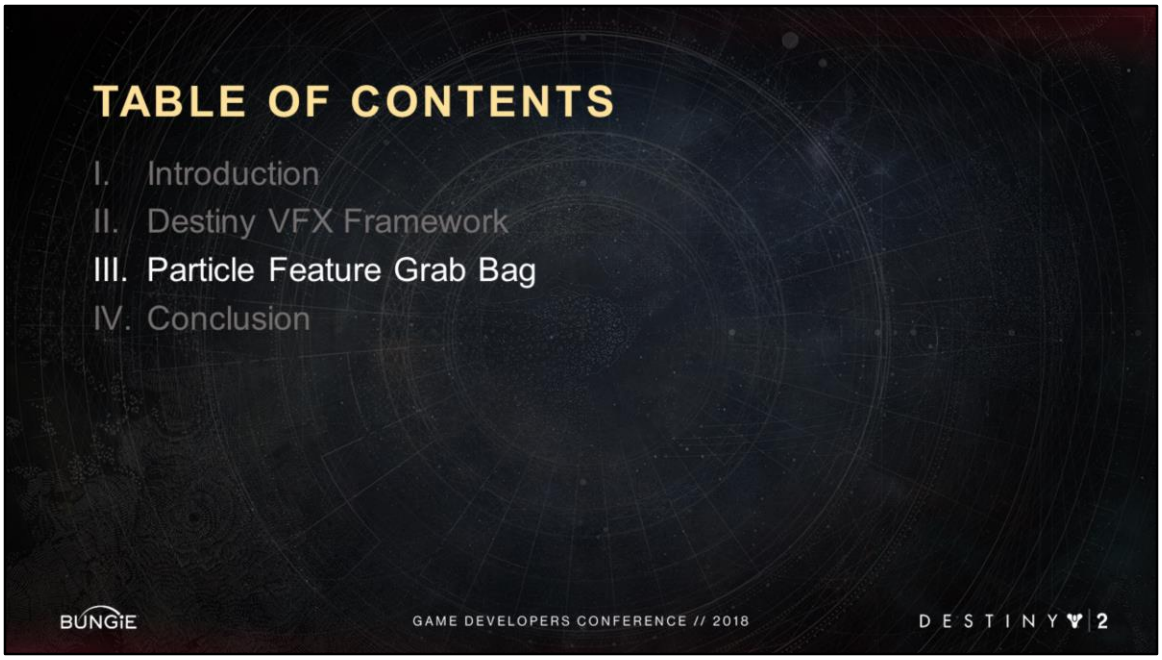
Complex timing of many distinct FX pieces, like lens flares, lights, particles and animations. Again, all of which could be intricately connected to game world state, like enemy health.

And potentially a large number of these FX happening at once, so performance is quite the challenge as well.

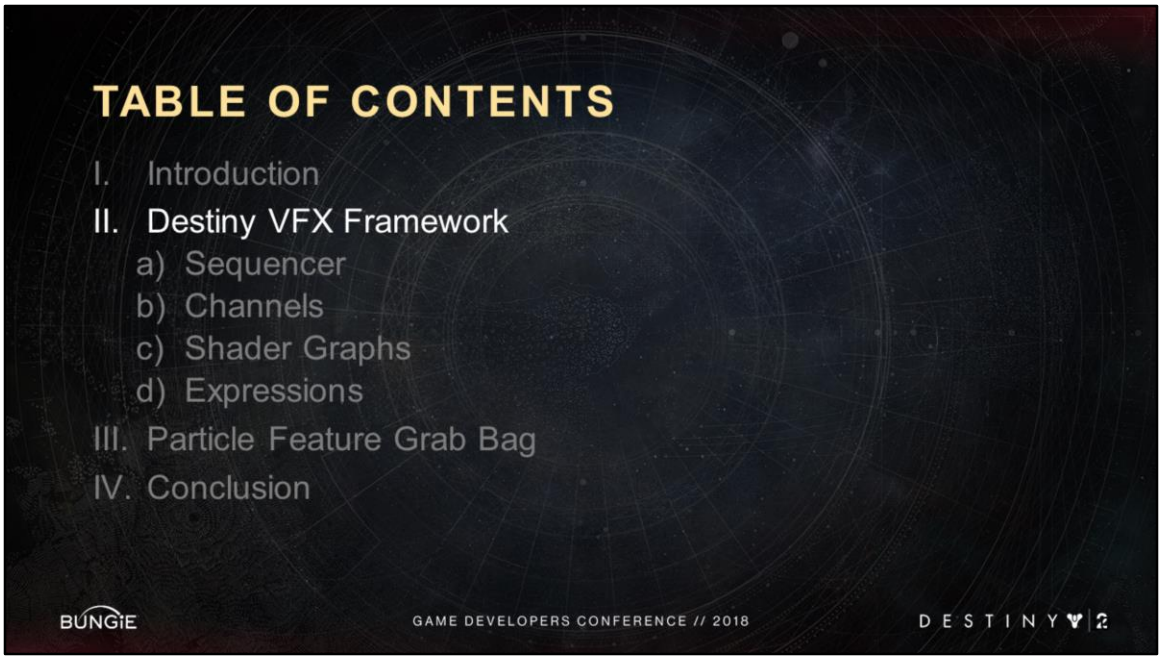


We decided we wanted to share two related yet distinct bodies of work.

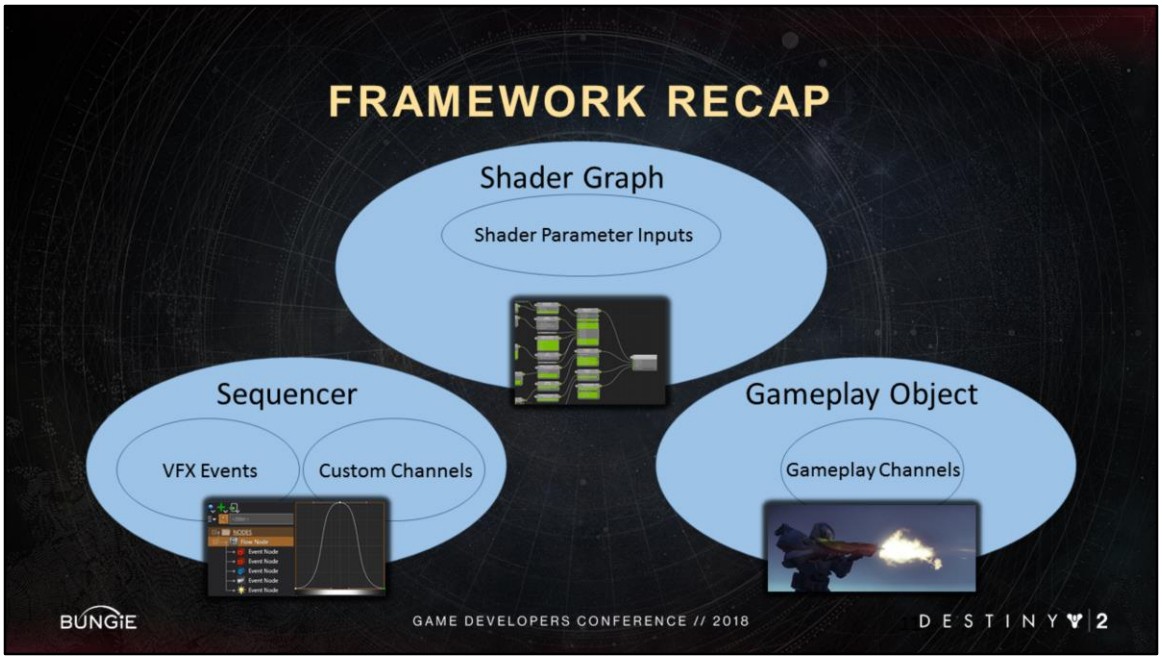
First is what we call our VFX framework. Here you'll learn what it's like to create an FX system in the Bungie engine (and how it's all sequenced, connected to gameplay, and how we give a lot of expressive power to our artists).



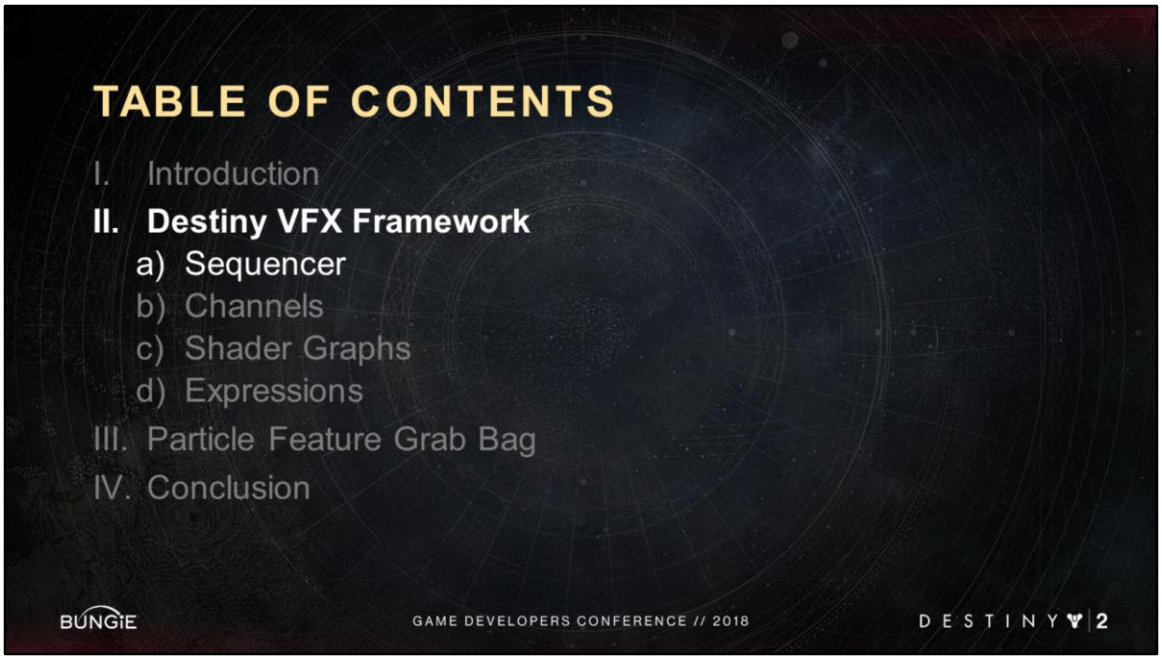
Second is a miscellaneous list of particle specific features that we found really useful. We will call out when these features benefitted from our framework, but they do not require our framework nor do they require each other.



Hello, I am Ali, Visual Effects Tech Art Lead.
Today I am going to walk you through our Visual Effects framework.



This is a high level conceptual overview of our framework.
I will walk you through it piece by piece



The first major tool of our framework is the Sequencer

This is the thermal flare grenade
Its visual effects are built entirely as a “Sequence”.
<Advance Slide>
This is what the sequence for it looks like in our editor.
It’s basically a hierarchy of nodes, called events, sequenced over time.

This is the thermal flare in-game, it demonstrates many of our vfx features.
Next I will reconstruct it one event at a time.

I start with an empty sequence.

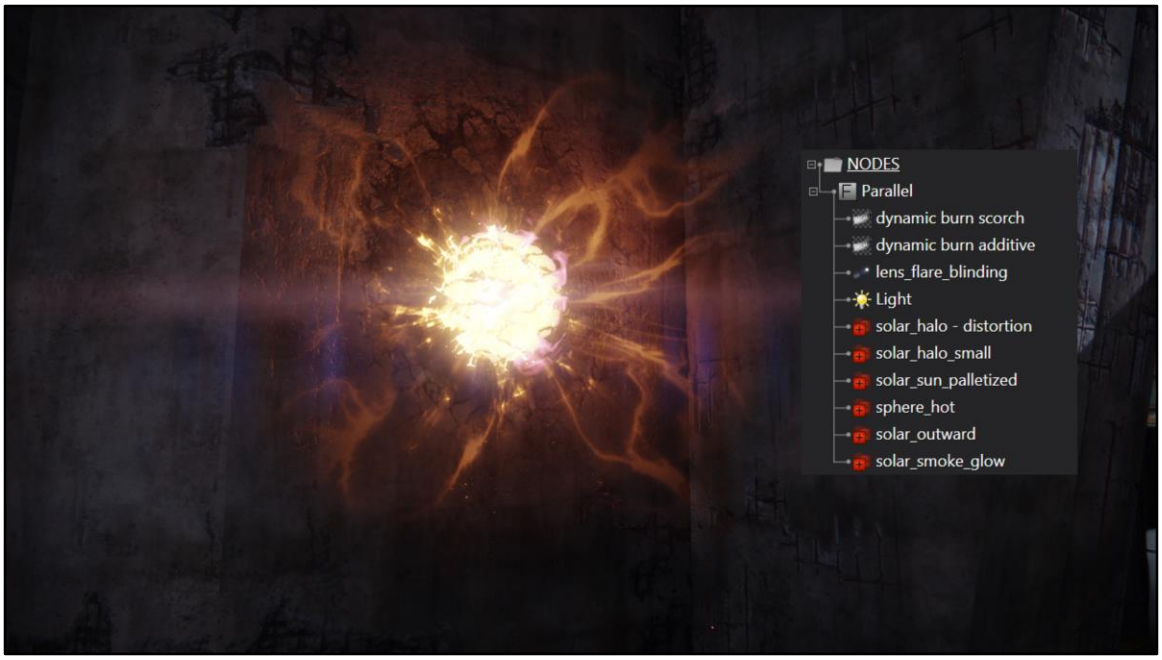
I add a scorch decal

I add an additive decal

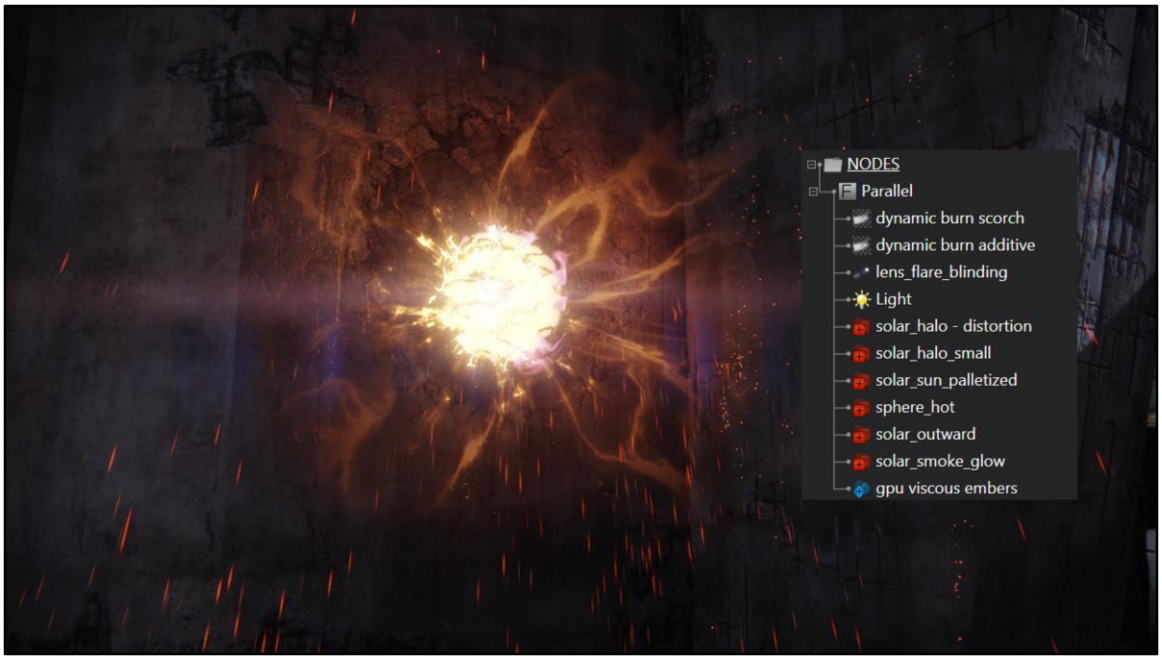
I add a lens flare

I add a point light

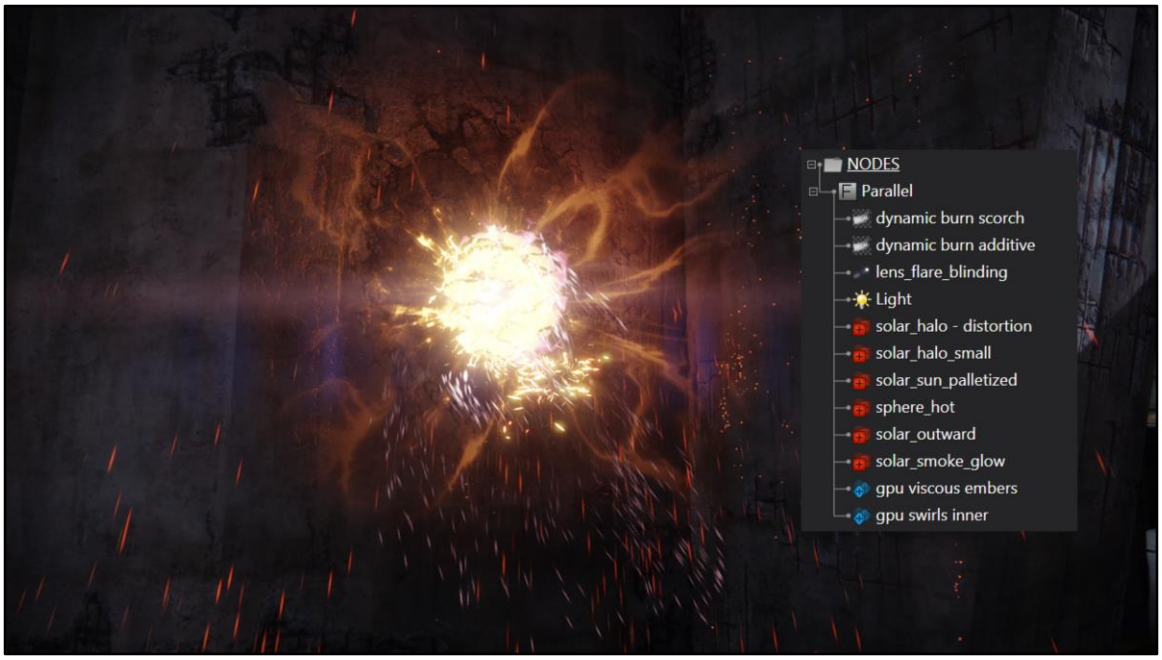
I add some core particles



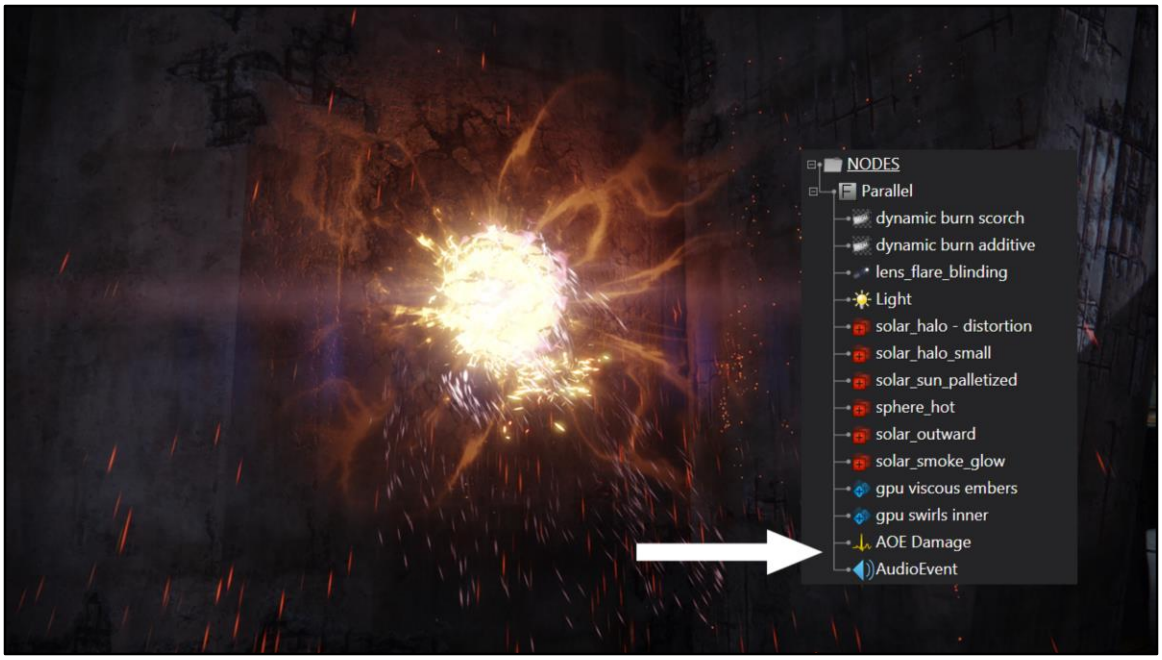
Add some radiating particles



I add some hot embers



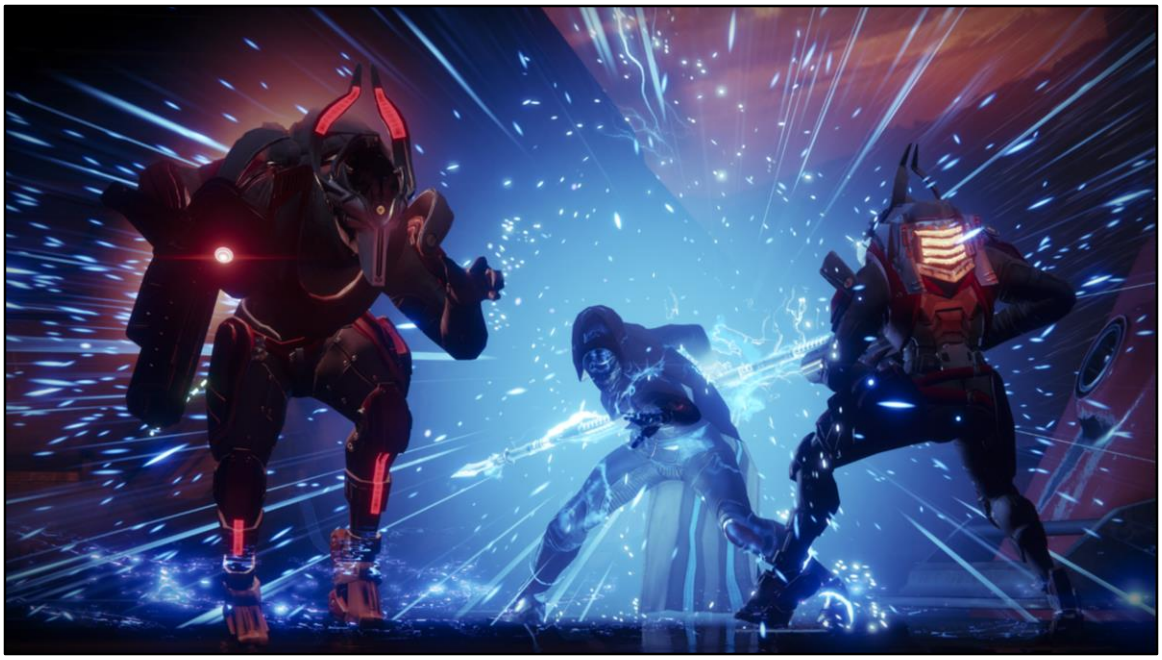
I add some swirly solar flare embers



It doesn’t have to stop there
It can also have audio event
and a damage event
In fact

It can have a whole lot of events.
The sequencer is not just a tool for vfx. It is a core tool across many departments: VFX, Design, Audio, Animation, UI…
They can all collaborate to author a wide range of performances
Like <Advance Slide>

the fire fx in this scene

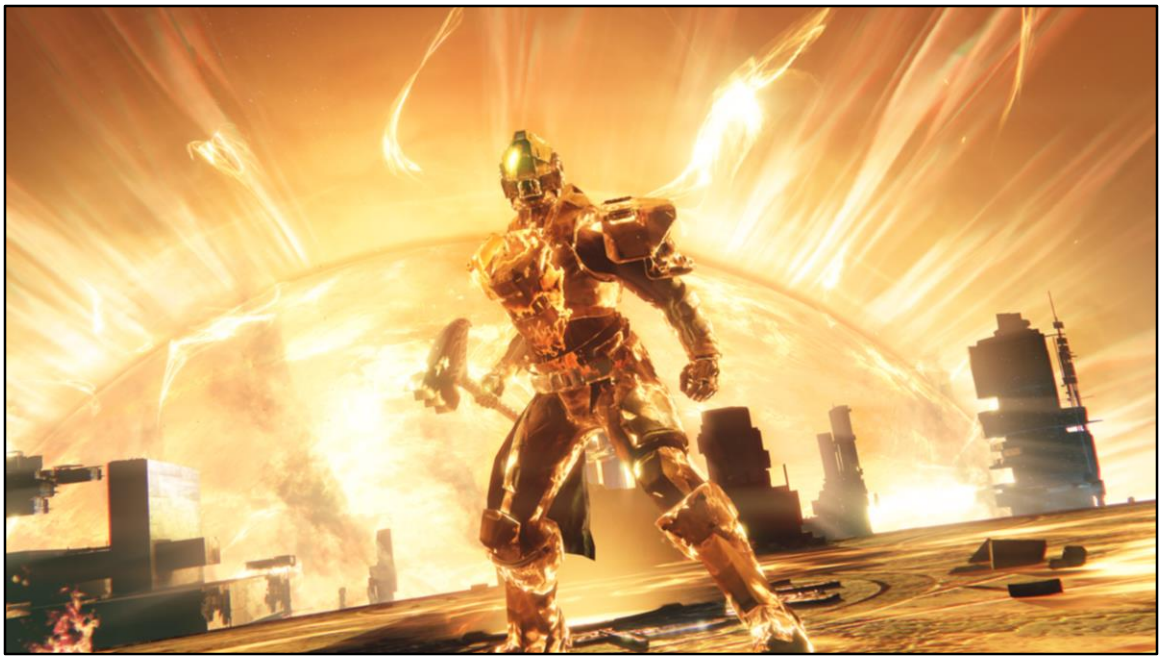


Or sand-boxy combat events

Including our player supers

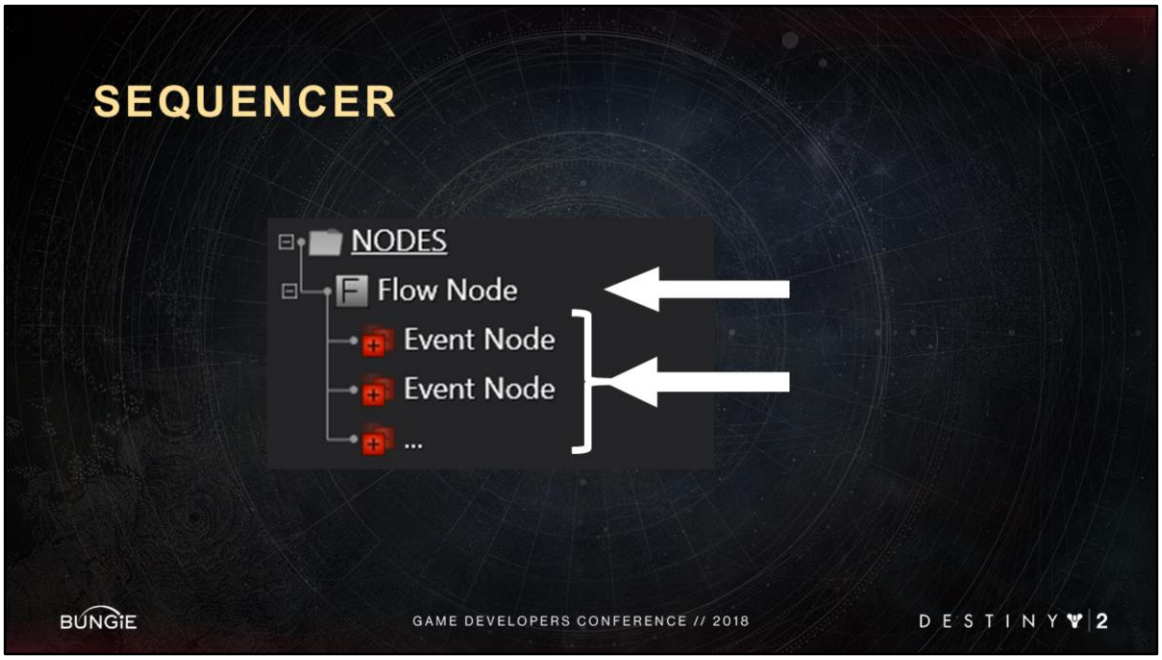
Or map load screens

Or the crazy scripted raid spectacles

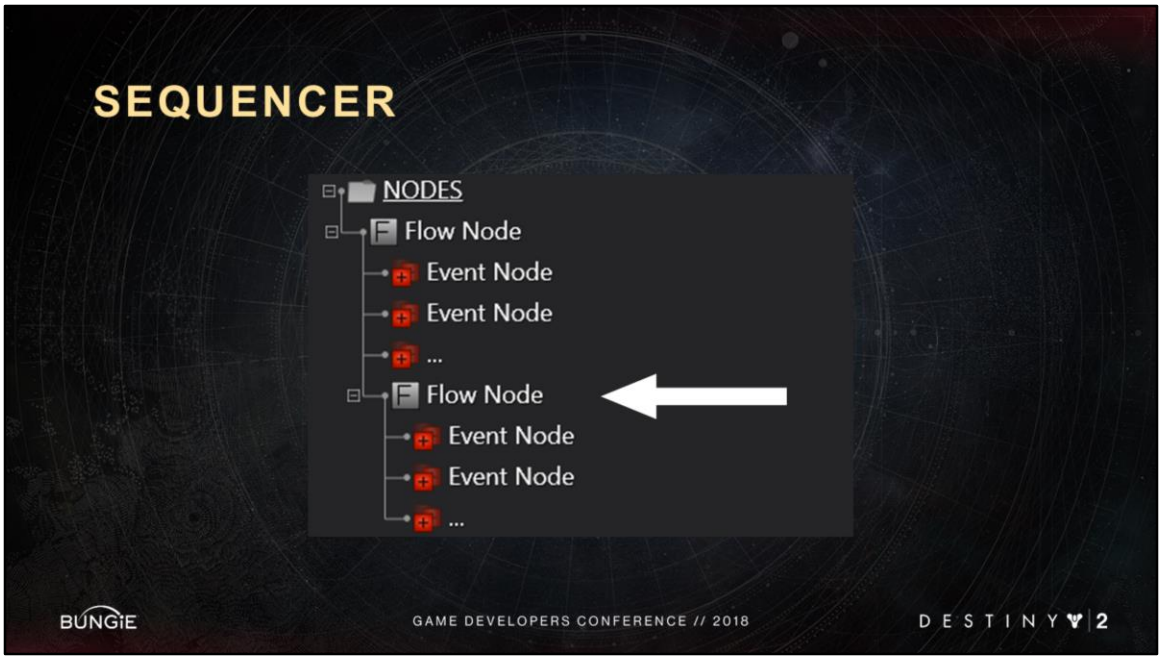


Or in-engine cinematics

Or even our dynamic skyboxes
These are all are built as sequences



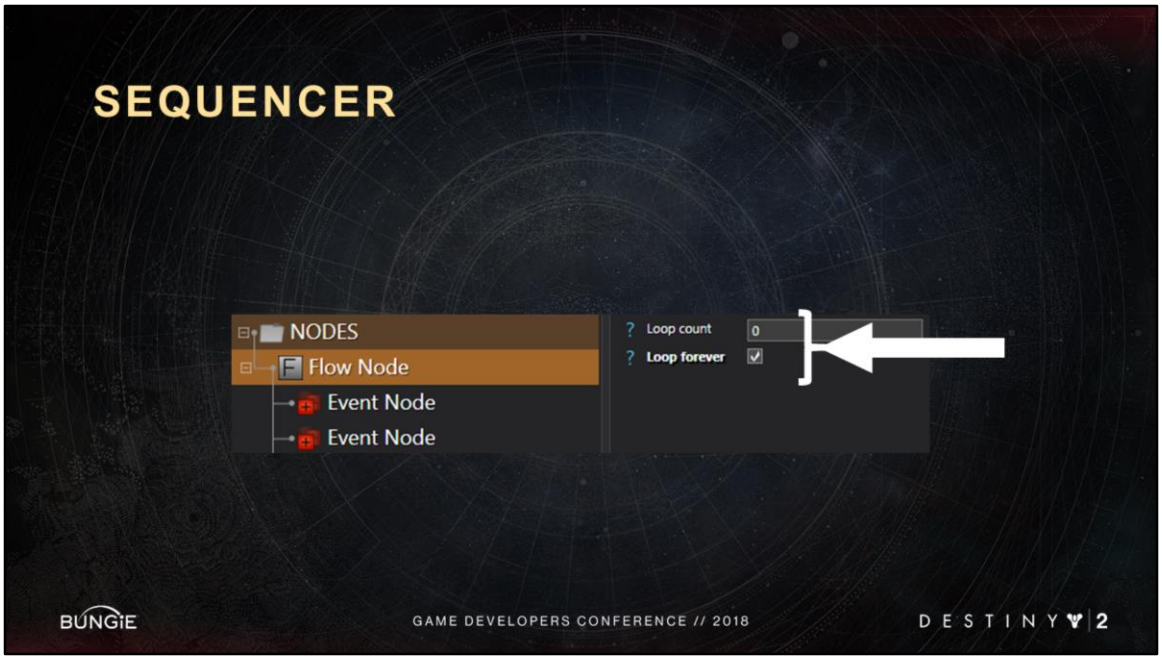
A sequence has two types of nodes
<Advance Slide>
Flow nodes
<Advance Slide>
Event nodes.
Flow nodes basically control how all their event nodes flow over time.
Flow nodes can contain many event nodes.



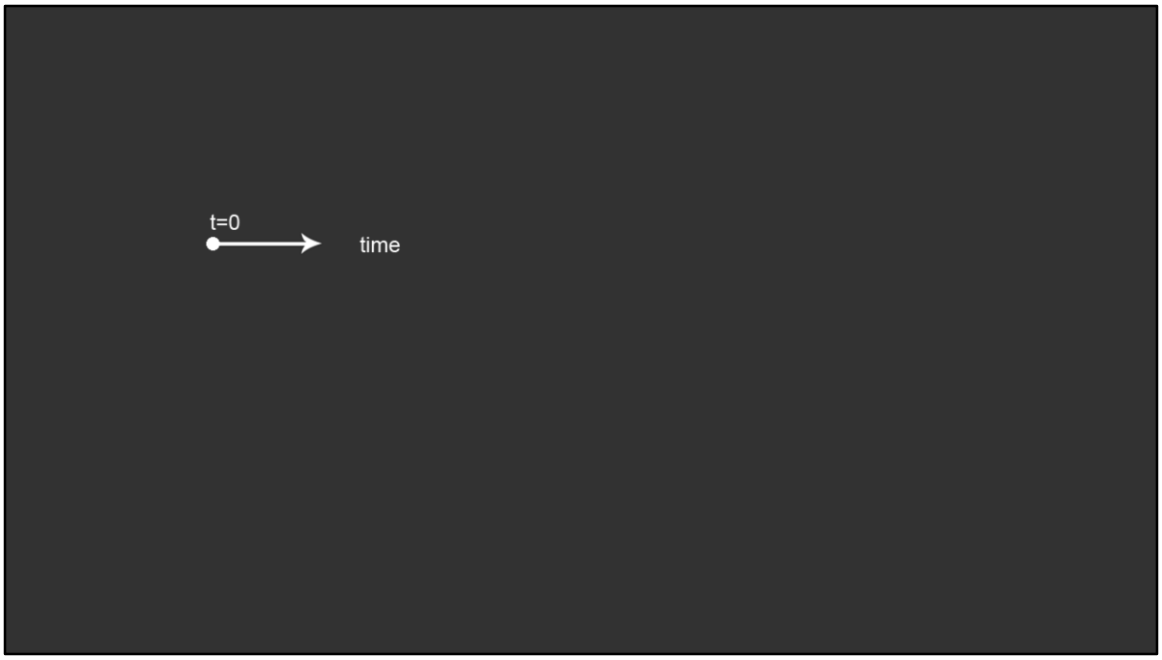
They can also contain more flow nodes
<Advance Slide>
for more complicated sequences.

We have different flavors of flow nodes

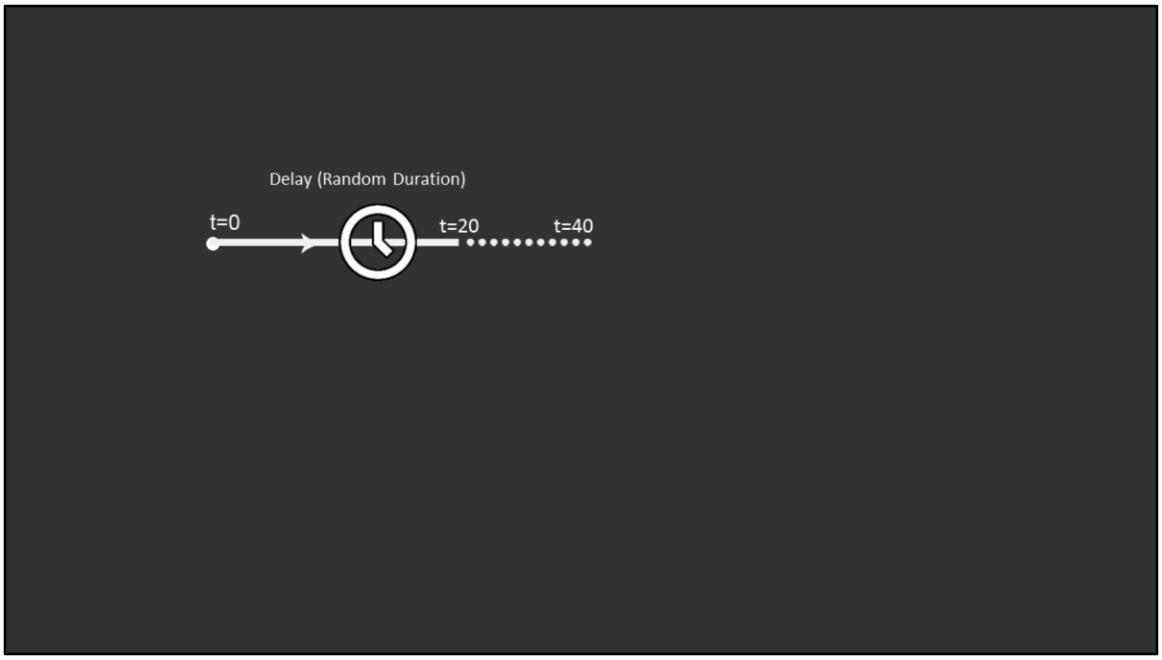
Parallel means all the events play at the same time.
<Advance Slide>
Serial means the events will play one after the other as each one completes.
<Advance Slide>
Random means only one of the event will be randomly selected to play.



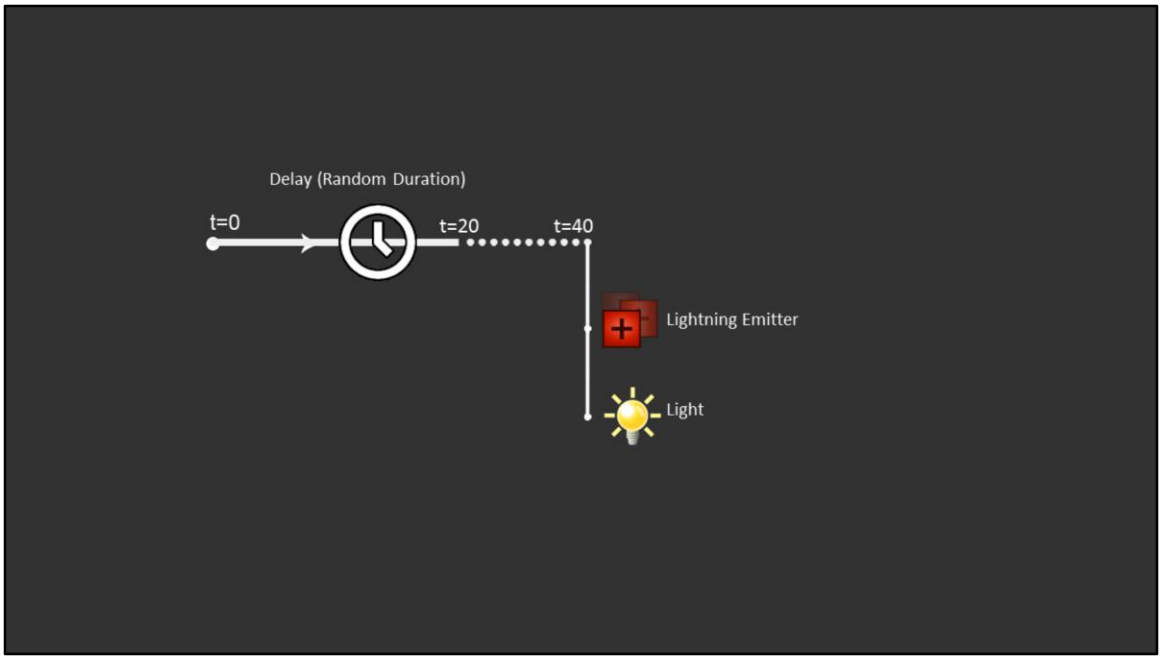
Flow nodes control looping; they can loop forever or have a finite loop count



To make it more clear, I will build a conceptual sequence from scratch.
For Ex: a lightning strike in the skybox,
I would need some sequence of events, like this

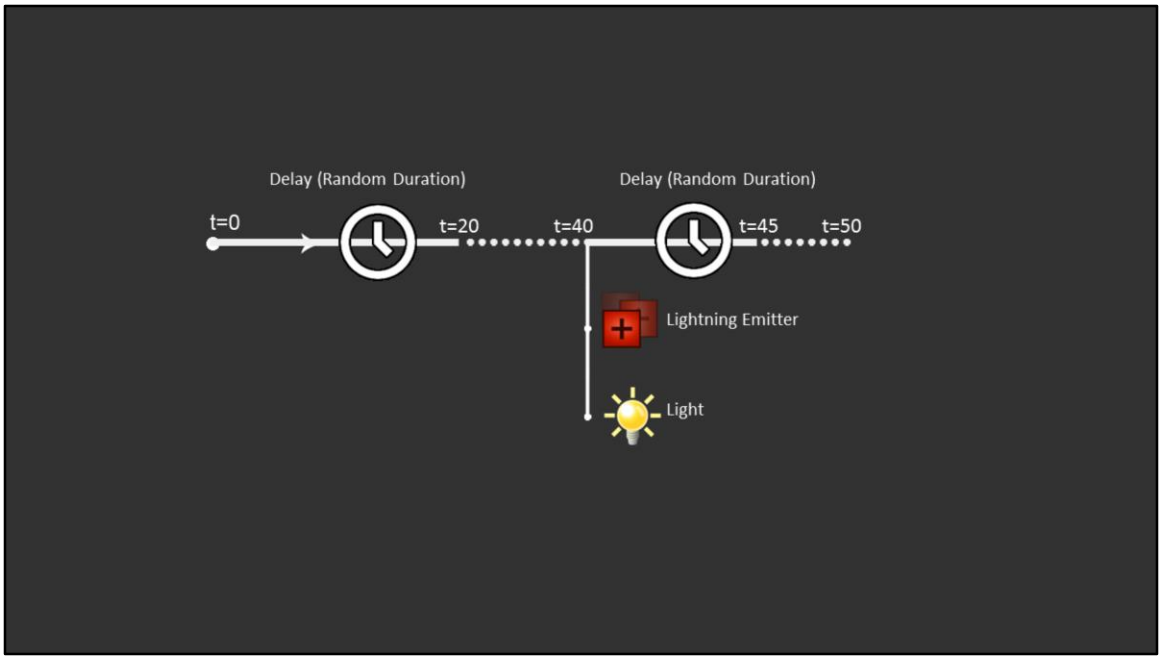


At time 0 I wait for a random duration between 20 and 40 sec
Horizontal sequencing means a serial flow of events

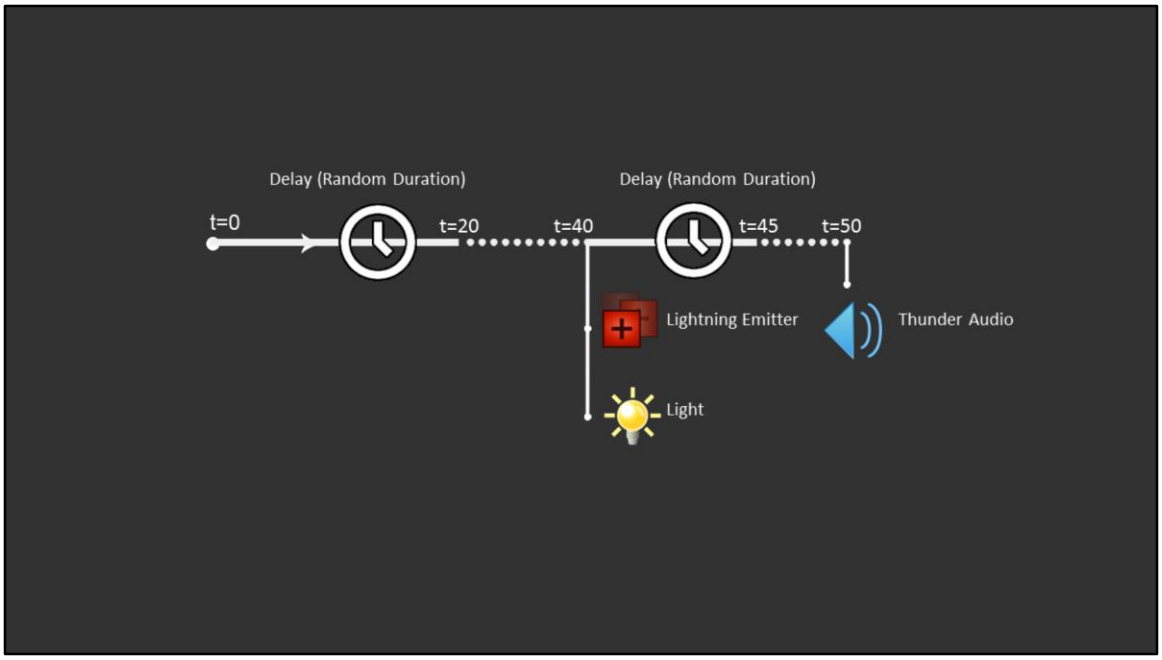


After that, I play in parallel a particle system and a light
These represent the lightning vfx

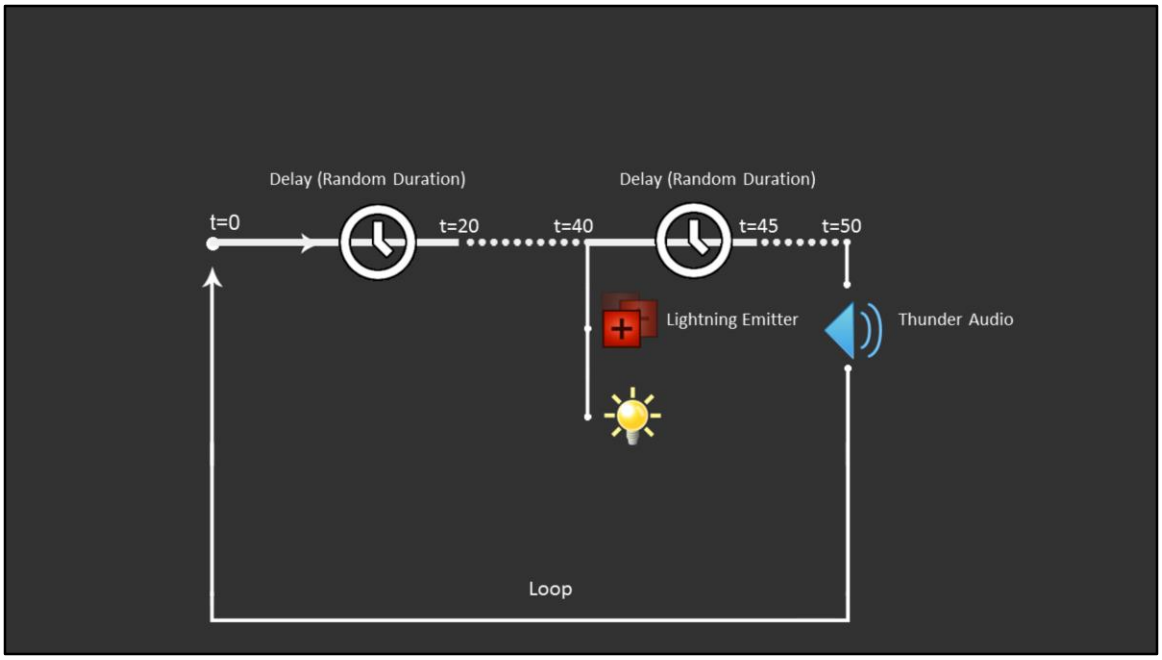
Vertical sequencing means parallel flow of events



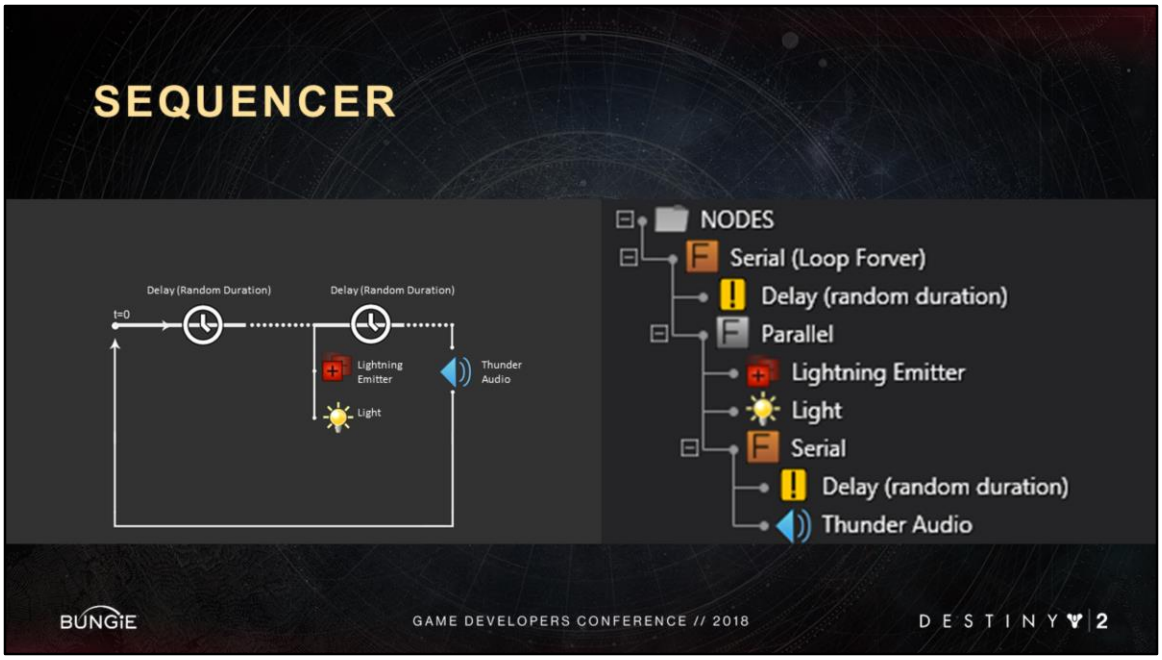
I serially wait another random duration between 5 to 10 sec



After that I spawn an audio for the thunder sfx
The delay gives the impression of travel time.

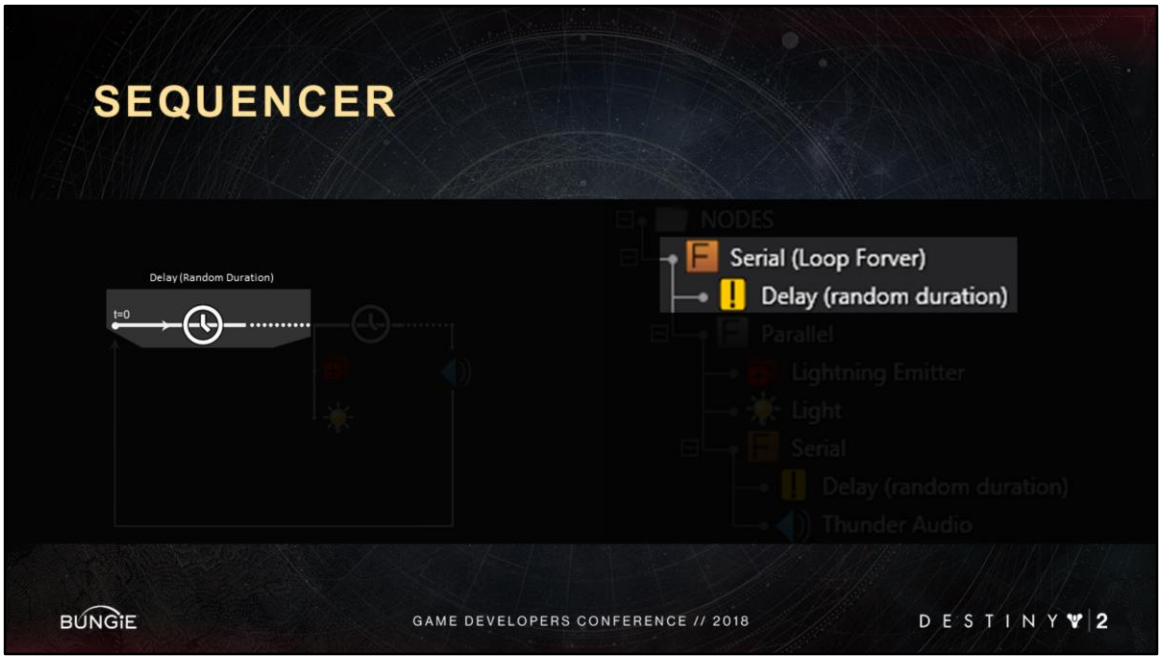


And after all that I loop

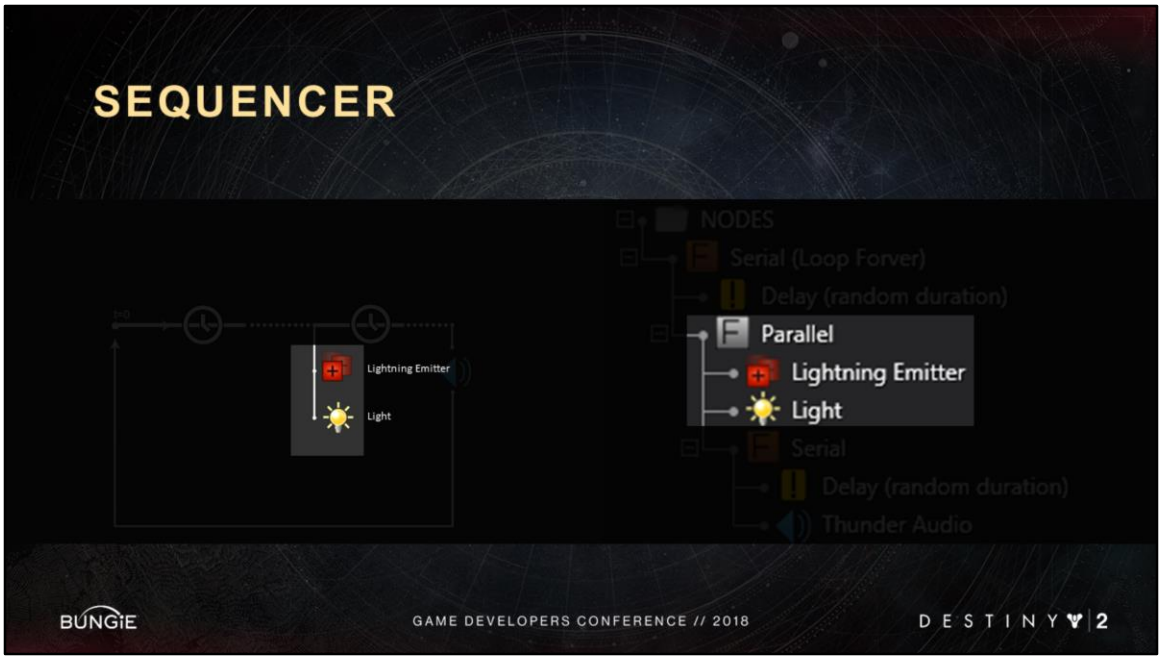


The sequence for this performance would look like this in the tree view.

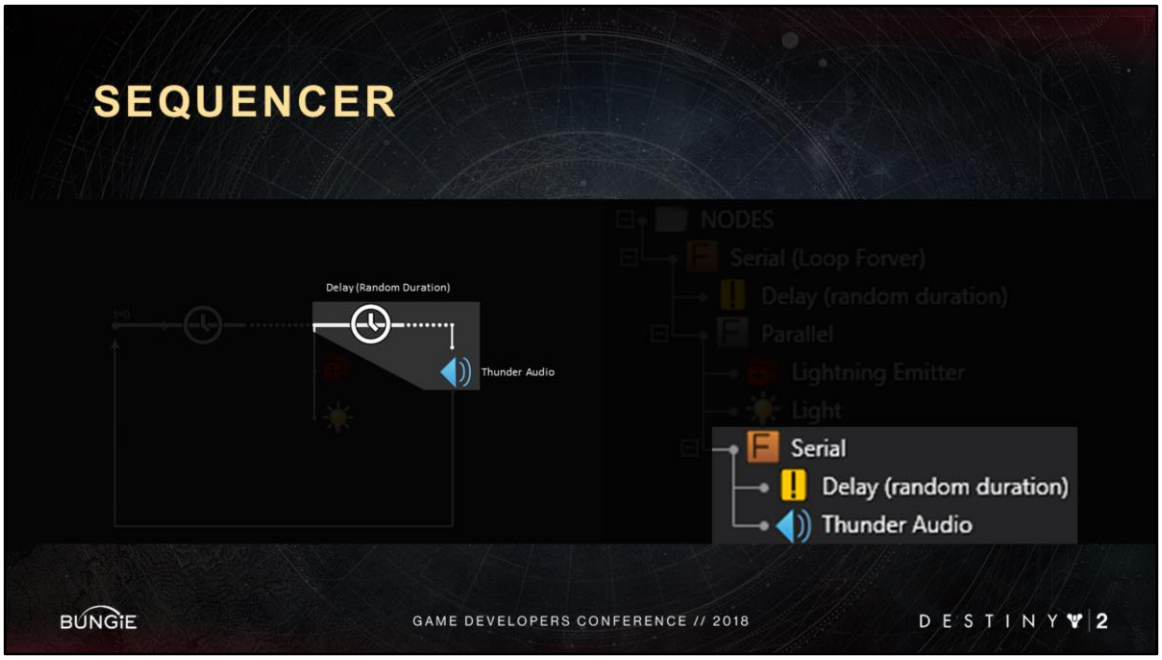
I'll compare the elements.



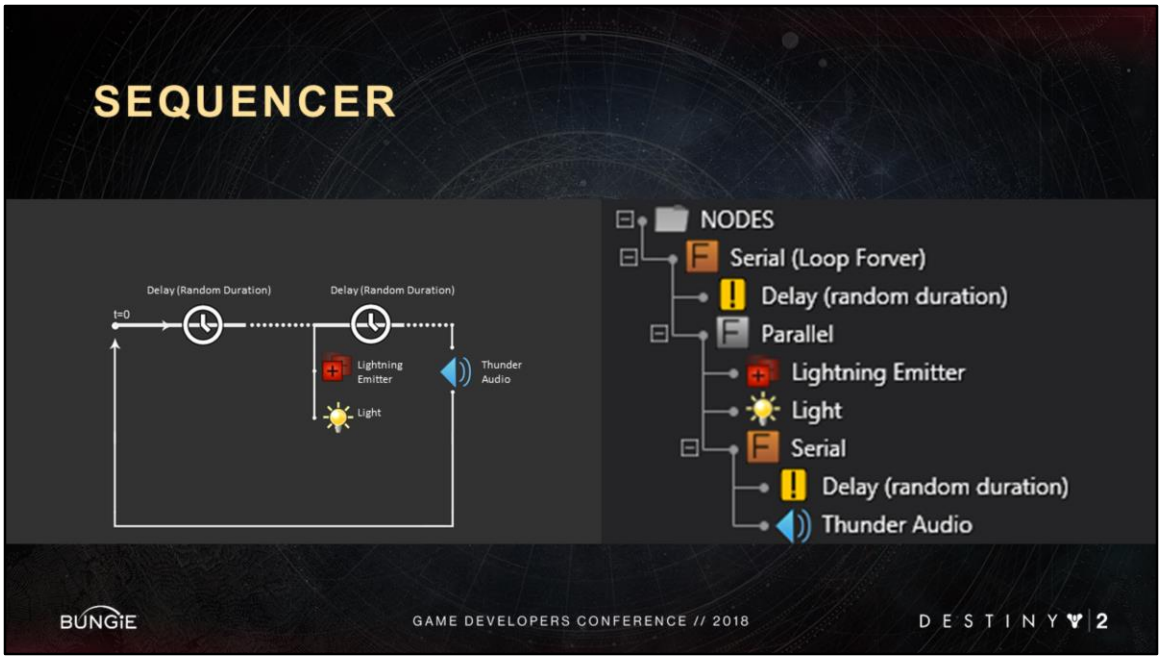
Here is the delay



Here is the lightning elements playing in parallel

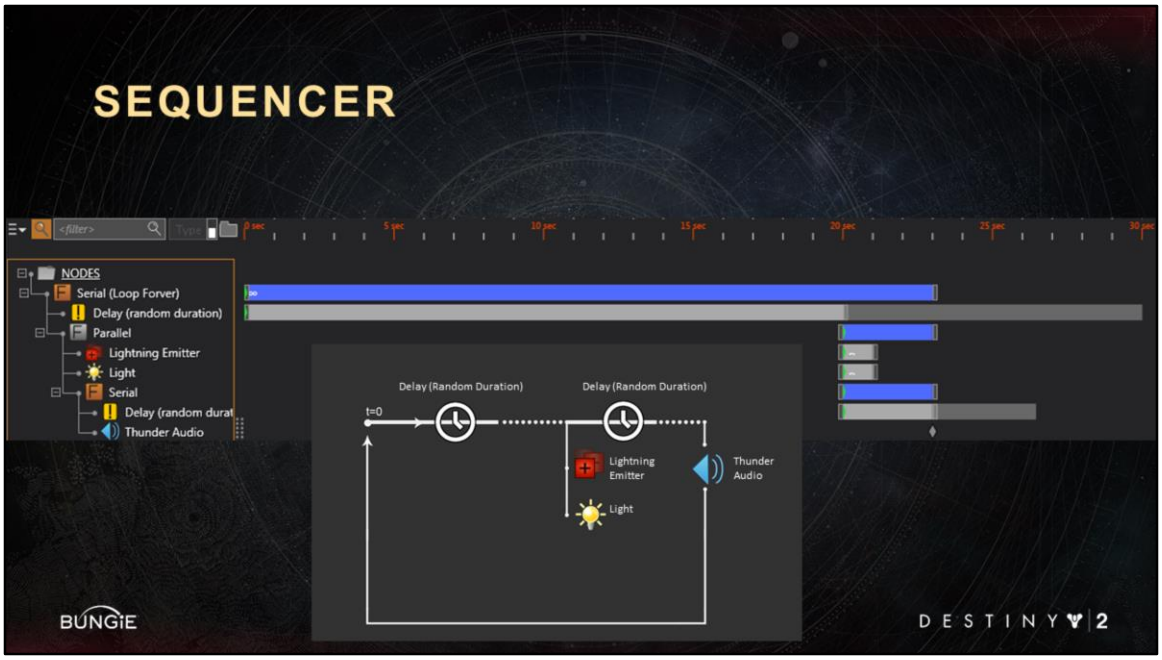


Here is the delay and audio playing serially



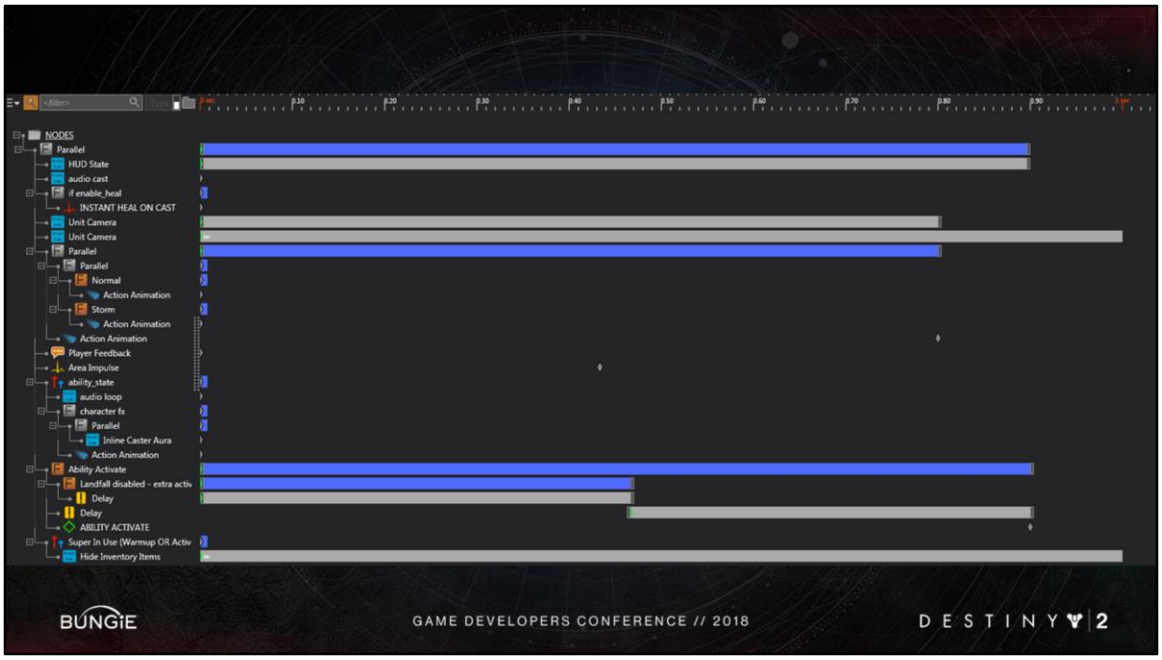
It might take a little getting used to, but we find that the tree view is a very nice and compact way of representing sequences.

They would otherwise need to be authored in something like a node graph UI.



We do however have an alternative UI : the timeline
With it you can more easily see relative timing between events.
<Advance Slide>
Here is the conceptual chart for reference.

If you are wondering how complex these sequences can get, lets check out a player super ability cast



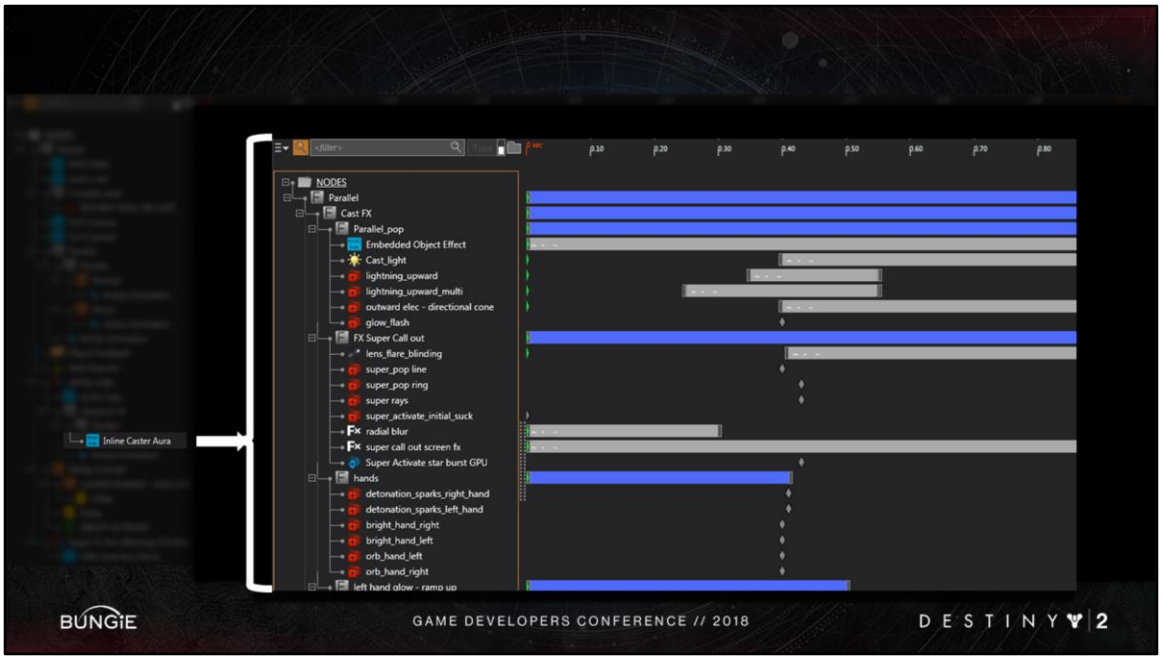
This is a shipping sequence, the stormcaller super ability cast.
In this one sequence, multiple departments have timed their events for a cohesive performance.

Here is a UI transition

a camera transition

a character animation

And this one little event node over here
<Advance Slide>
is actually a whole other nested sequence…
which looks like this



Inlining, or nesting, sequences is great for two reasons:
1) it allows different teams to concurrently iterate on the same performance.
2) it allows us to build modular content that we can plug and play in many other sequences.

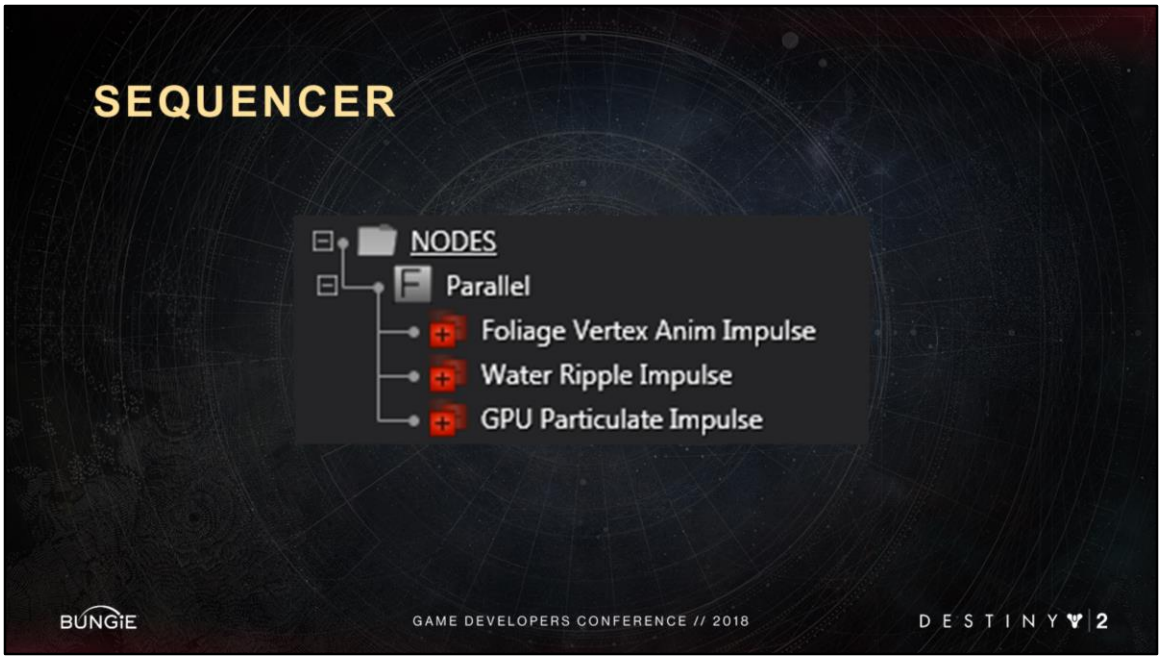
This is critical for us to rapidly evolve our content. By updating just a few sequences, we can propagate changes to the entire game.

For example, on destiny 1 I made these impulse particles that could influence water and foliage

I packaged these impulse particles up as 1 sequence
And I **inlined** it into every explosion in the game.

In destiny 2 Brandon added support for gpu particles, like this snow effect here
We soon realized it looked strange that the snow didnt react to explosions

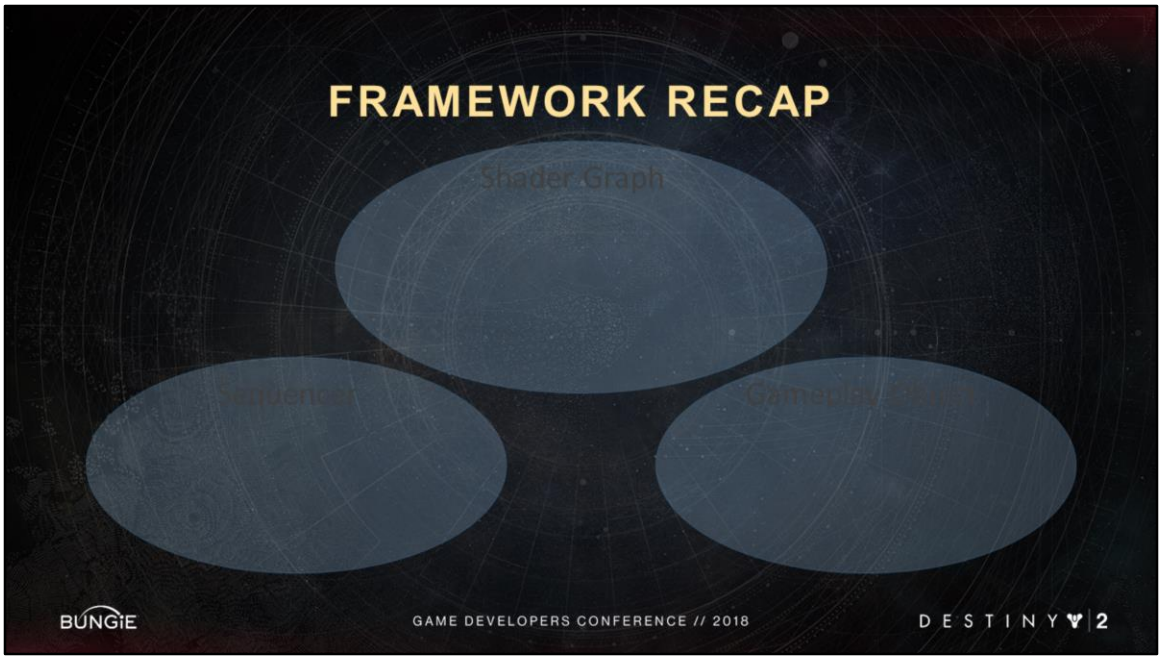
So Brandon added some new tech to solve this: Impulse particles that can push gpu particles from other systems.
Now this feature came in late in the project.



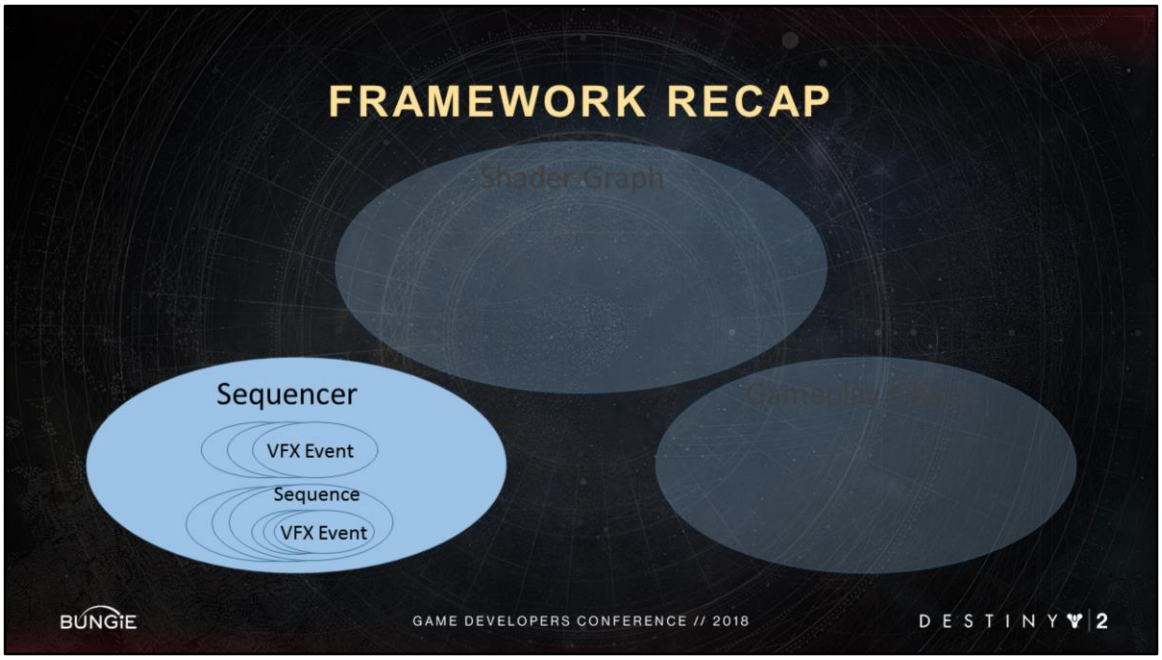
By simply changing this one sequence
From this <Advance Slide> Into this
I added this feature to every explosion in the game.

Without inlining we wouldn't have been able to propagate this new feature that late in the project.

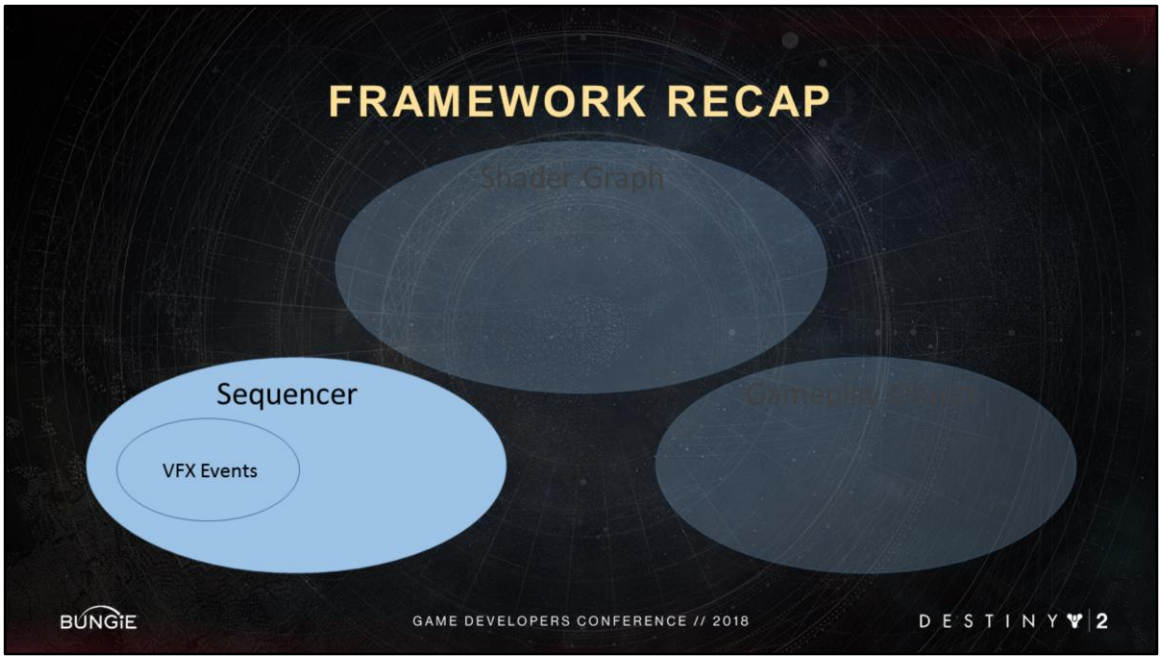
This is the power of inlining and modularity.



So to recap, here is the conceptual chart of our framework.



We have the sequencer which can contain many events, including vfx events.
<Advance Slide>
And the sequencer can inline many other sequences
<Advance Slide>

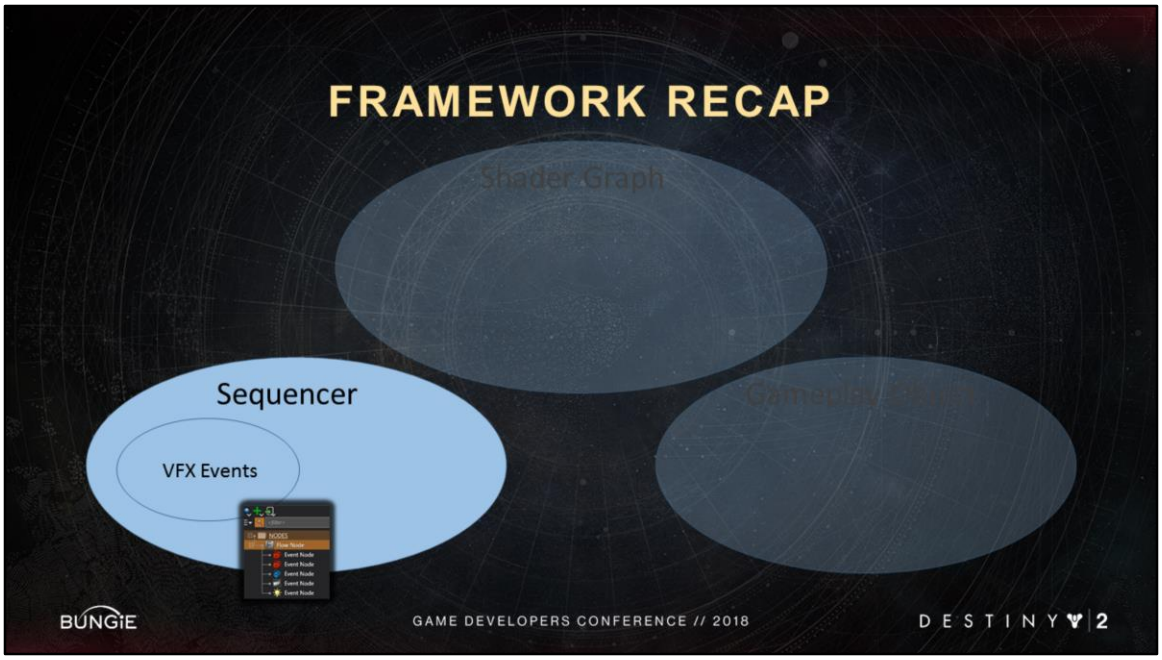


Or this for simplicity

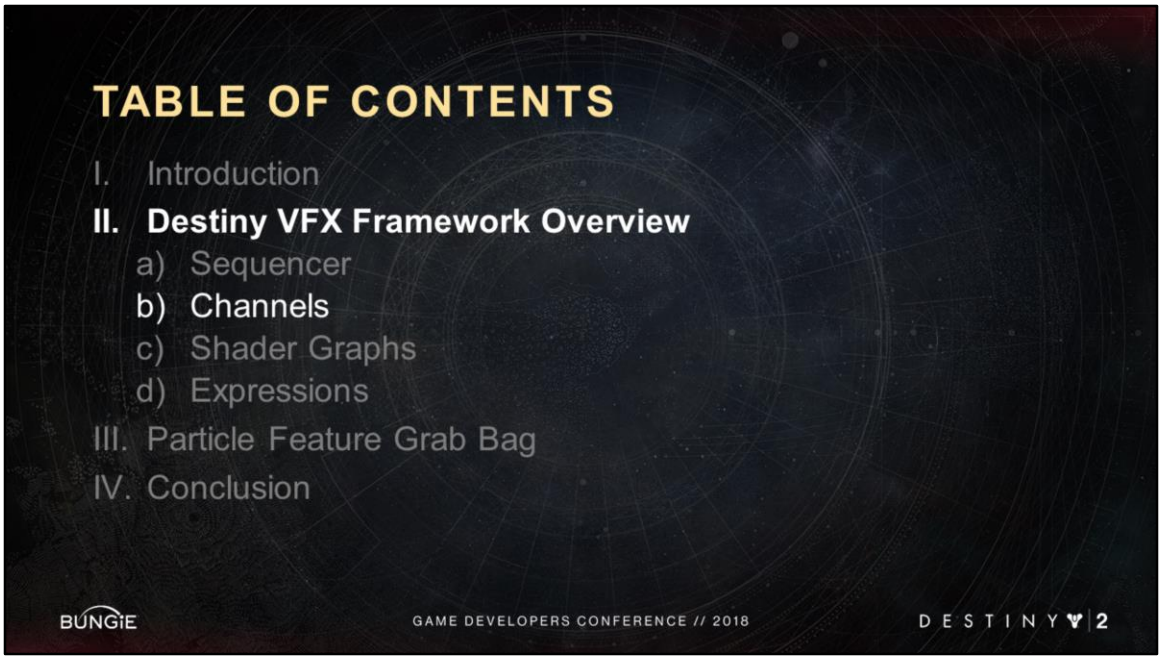
The Sequencer allows multiple teams to collaborate on the same performances

Teams can sequence their events over time in very complex ways.

Inlining allows us to build a modular library of vfx that we can reuse & evolve as new features come online.



And with that we've unlocked the first part of our overall framework.



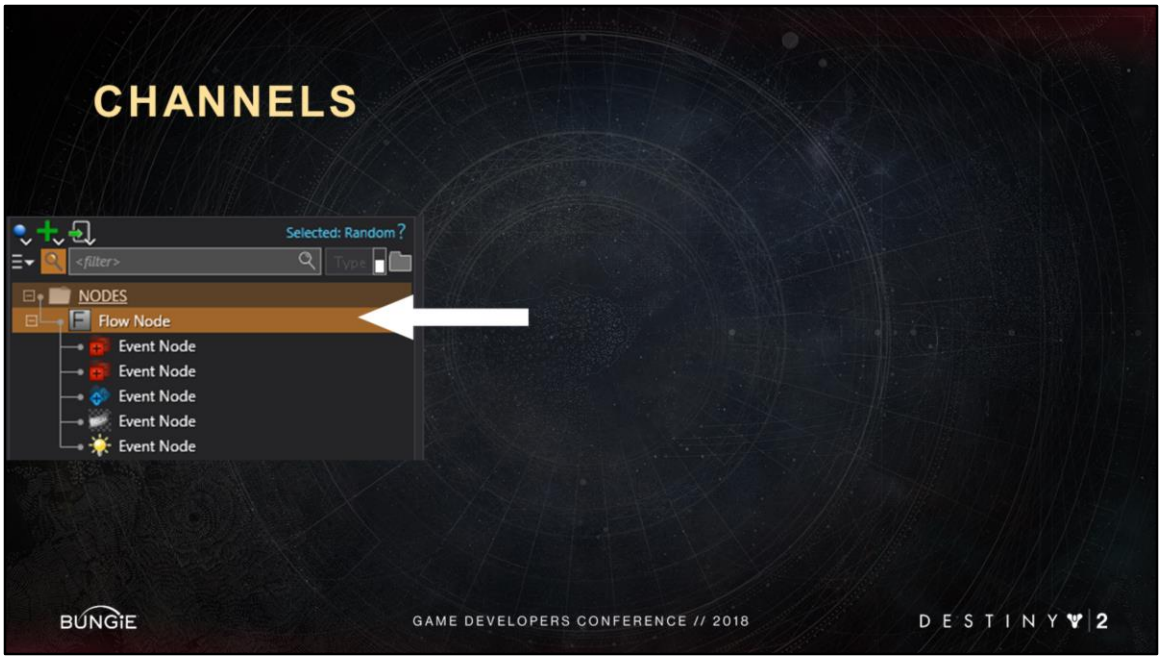
The next major part of our vfx framework is **channels**.

A channel is a property on a gameplay object that can be inspected at runtime.

Example channels on a player would be:
- health / aim_vector (which I am showing here)

Channels on a weapon would be:
- damage type, ammo count

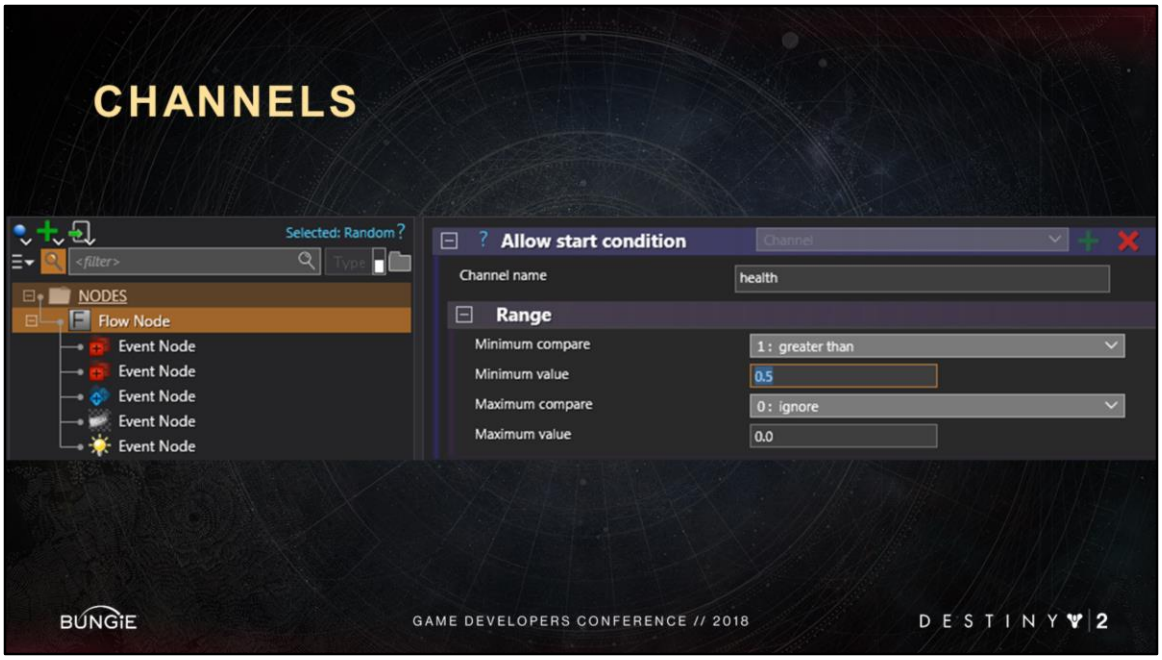
Any gameplay channel can be used by the sequencer



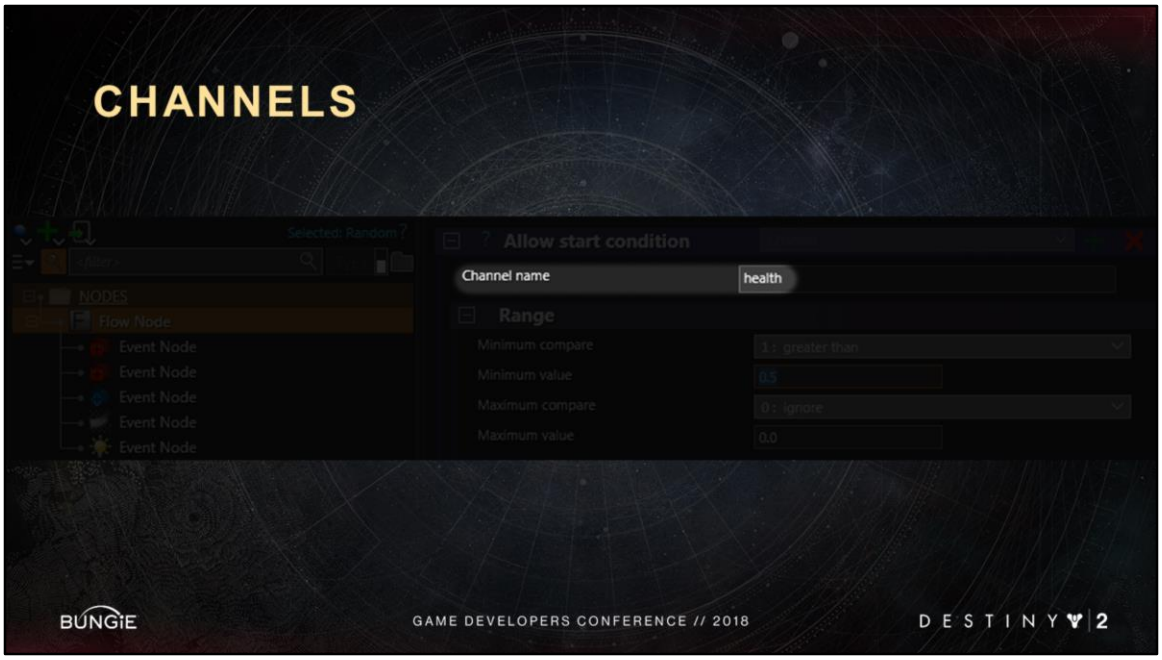
In the previous section we saw that the sequencer uses flow nodes to control **HOW** events activate over time.

The sequencer can also control **IF** events activate at all; using conditions checks on channels.

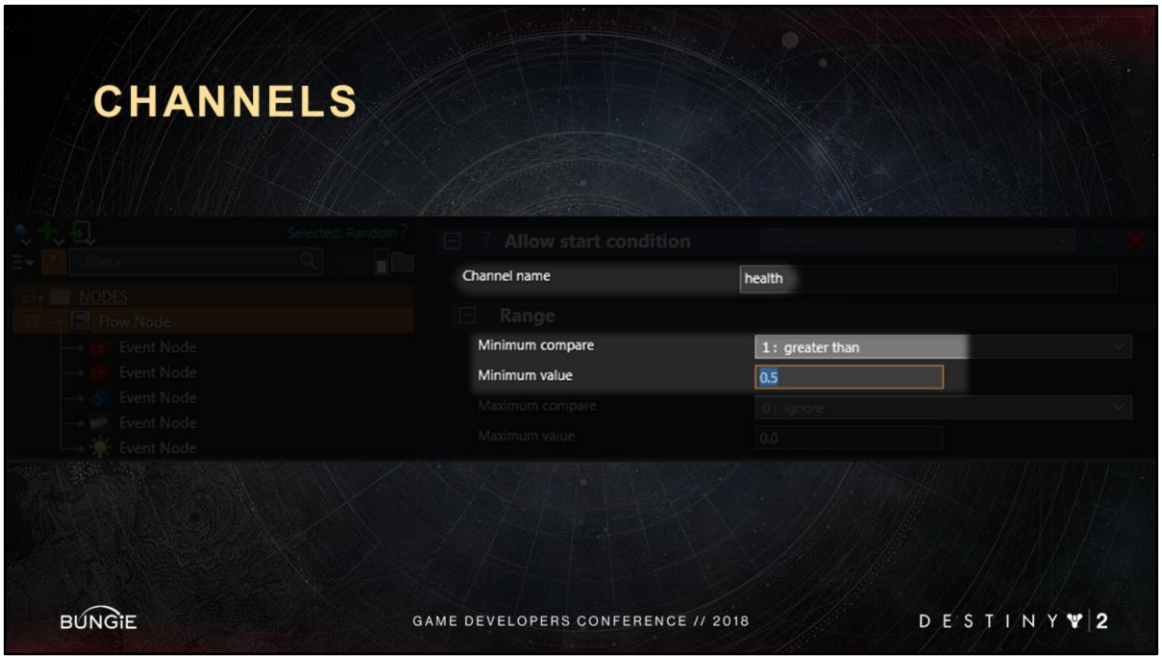
If you click on this flow node here…
<Advance Slide>



You can see its property panel on the right
And in there you can set a condition check.

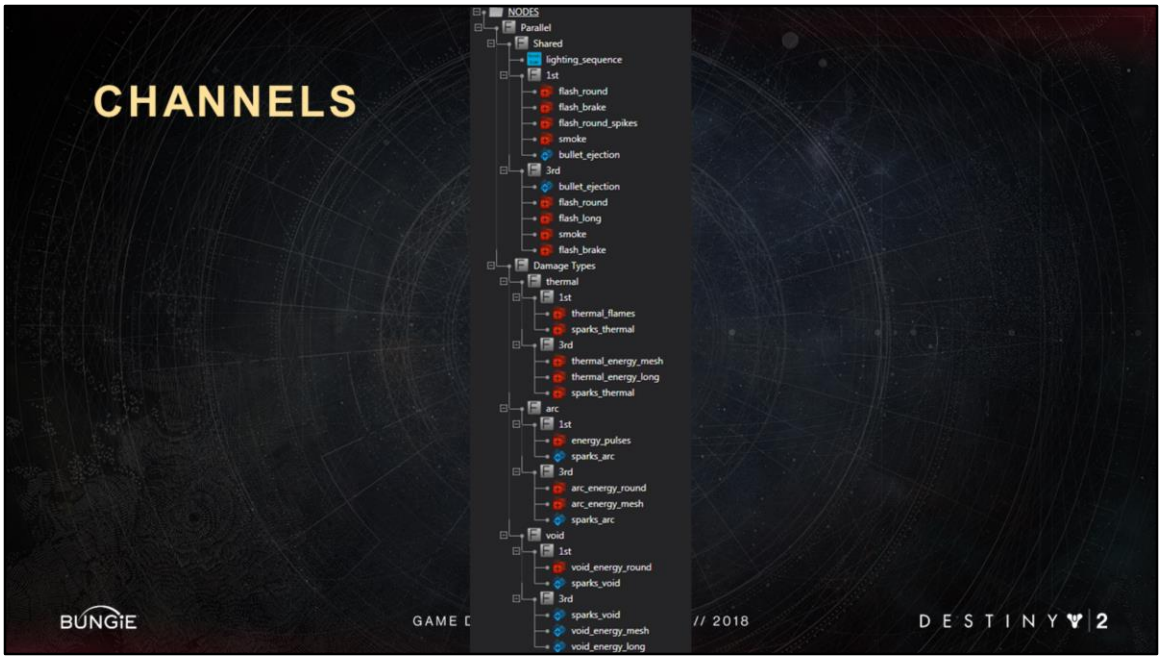


In this example, the channel I am looking at is health

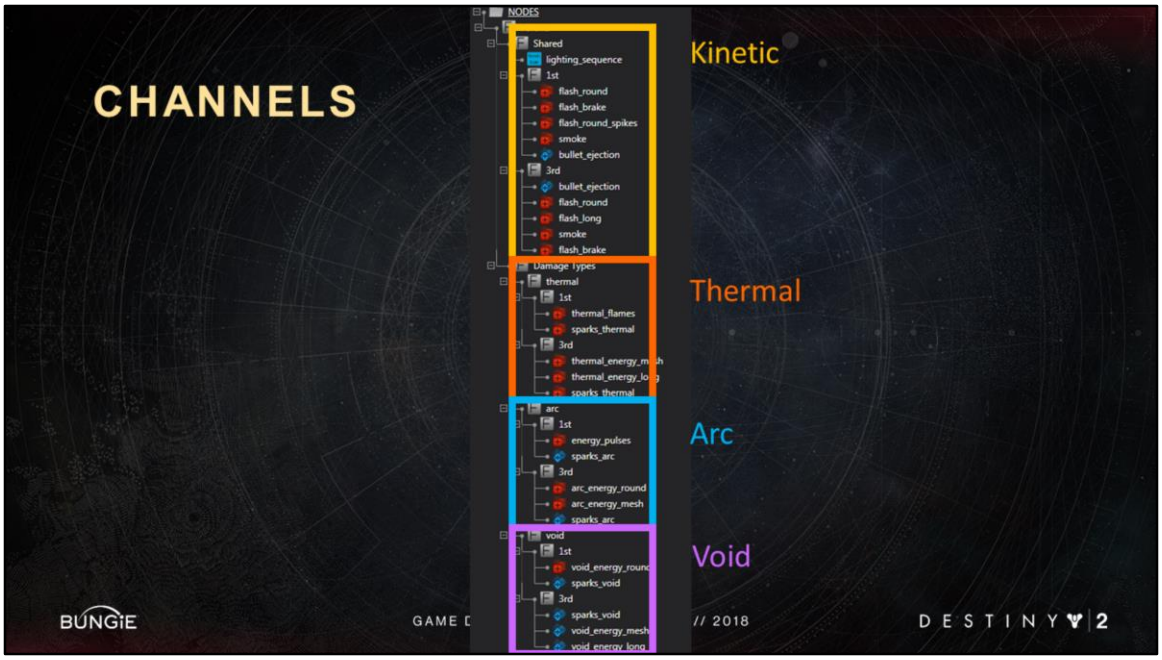


And the condition says: if my health channel is greater than 0.5 then the flow node can activate.

In Destiny, we have several different damage types.
As you can tell by the muzzle flashes here each has its own color and visual language.
And if you look at a muzzle flash sequence…



It would look like this.
You might be wondering why there are so many events for just a muzzle flash.
Well, not all of them activate at the same time.
We use channel conditions to decide which ones activate.



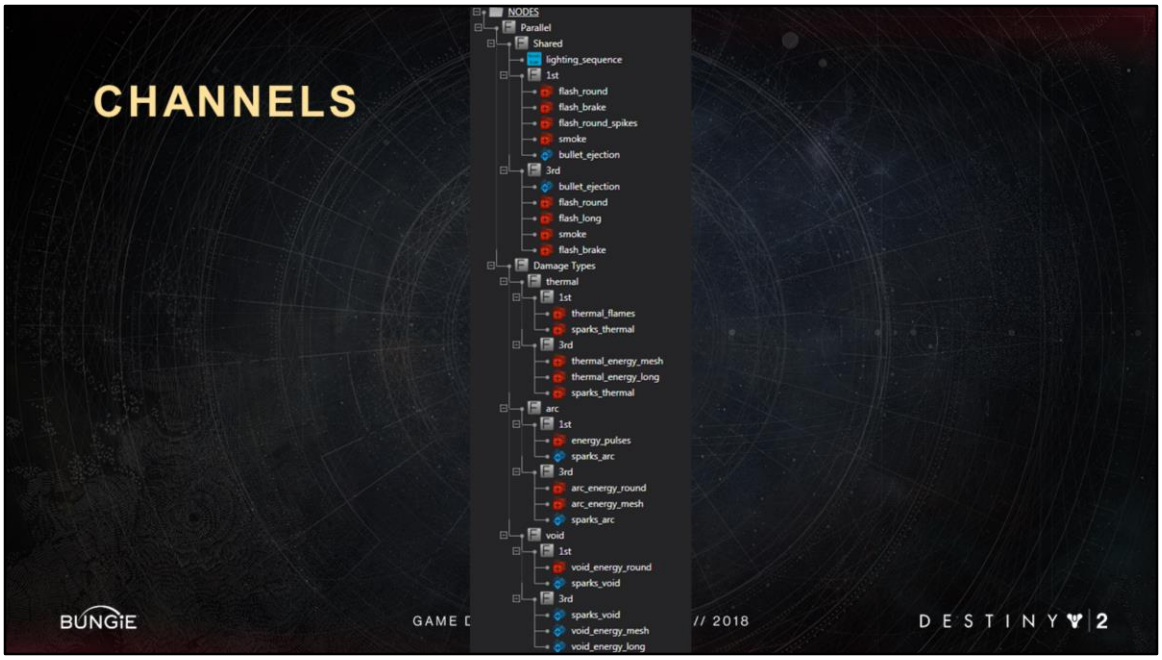
First, the sequence checks for the damage type channel of the weapon, and only activate the nodes that match
<Advance Slide> these nodes are the kinetic and shared muzzle flash (we'll come back to the shared ones in a bit)
<Advance Slide> these are the thermal
<Advance Slide>…

Second, we check for whether the weapon is in FP or TP

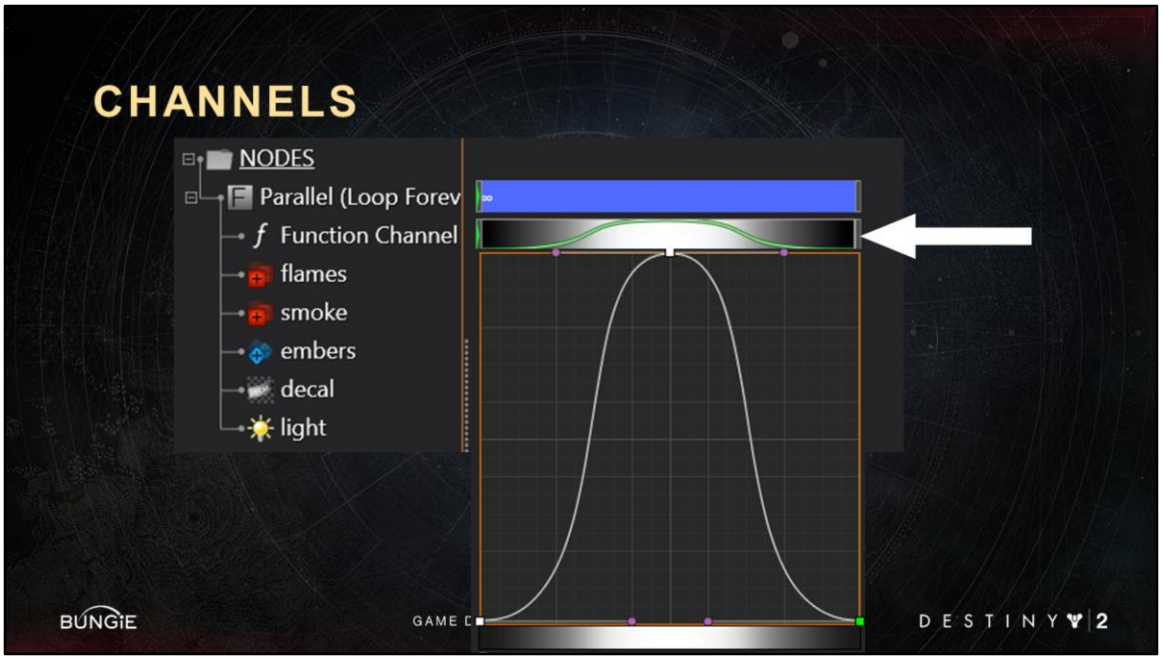
These nodes <Advance Slide>x2 only play if the weapon is in FP

Why do we not use the same vfx? Because we want to hand craft vfx that have been tuned and optimized for each camera type.

And these two conditions compound so that only one these small boxes ever activates.



This is just one shipping example of how we use gameplay channels in the sequencer.



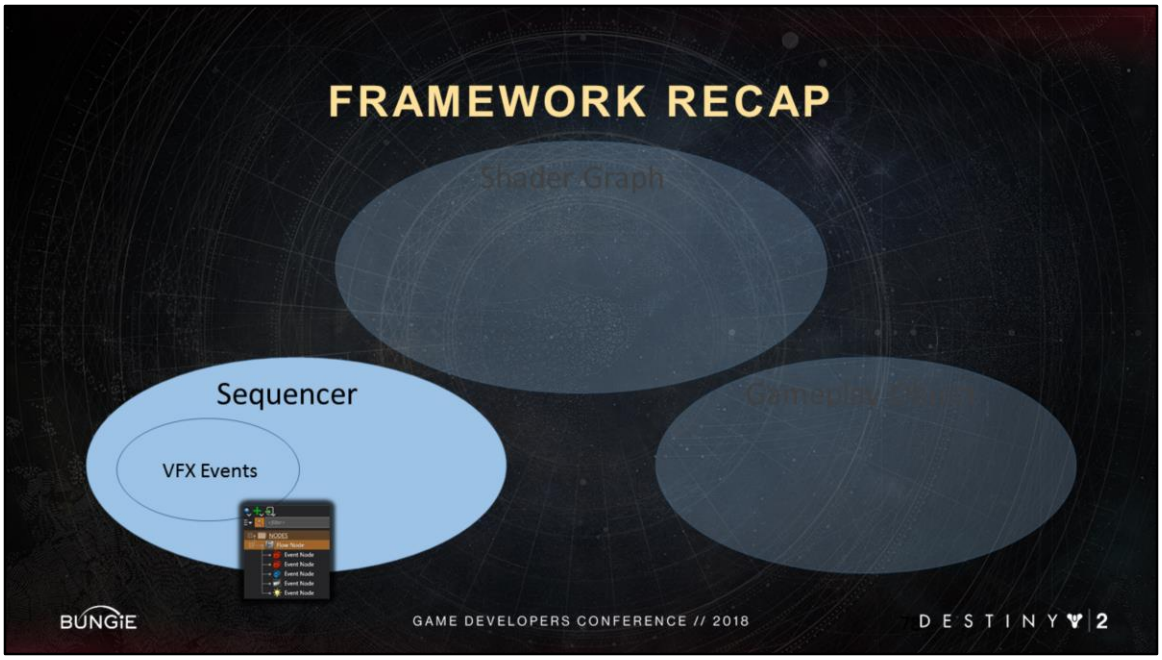
A second really powerful aspect of channels is that we can define our own custom channel right in the sequencer.
You can see one here
<Advance Slide>

What this gives us is an arbitrary keyframed curve we can consume is ALL vfx events in the sequence.

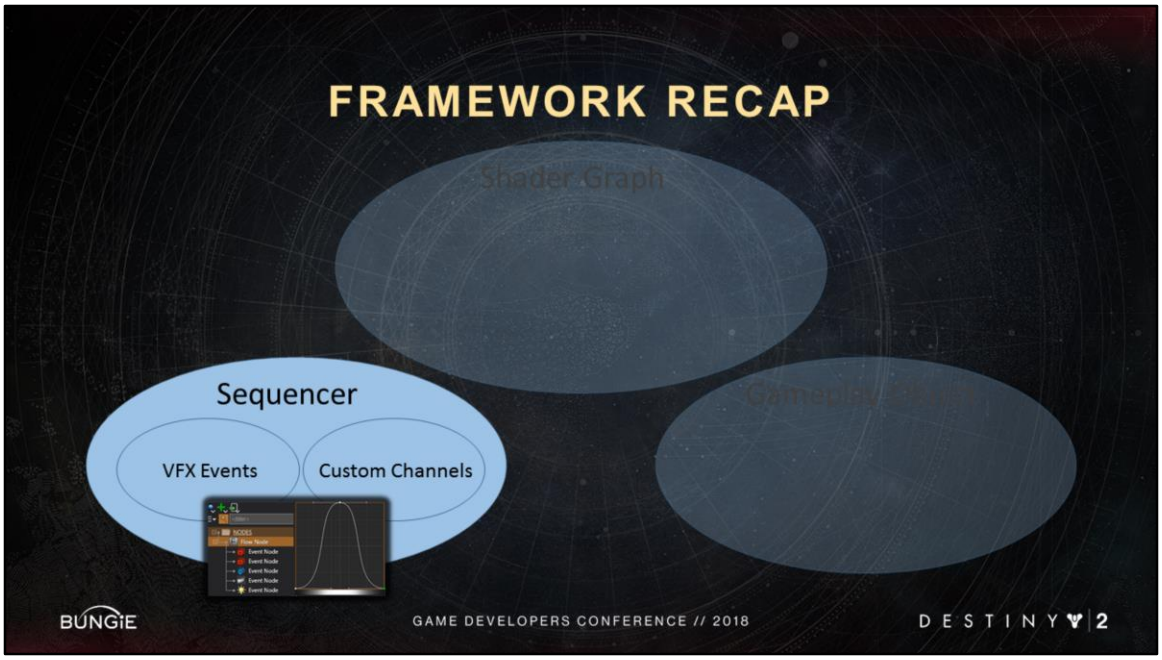
Here is an example video of a vex boss Argos in a test map.
All the VFX are being synchronized with a single custom curve called “charge_intensity”

In fact the custom channel here is also driving the animation blend.

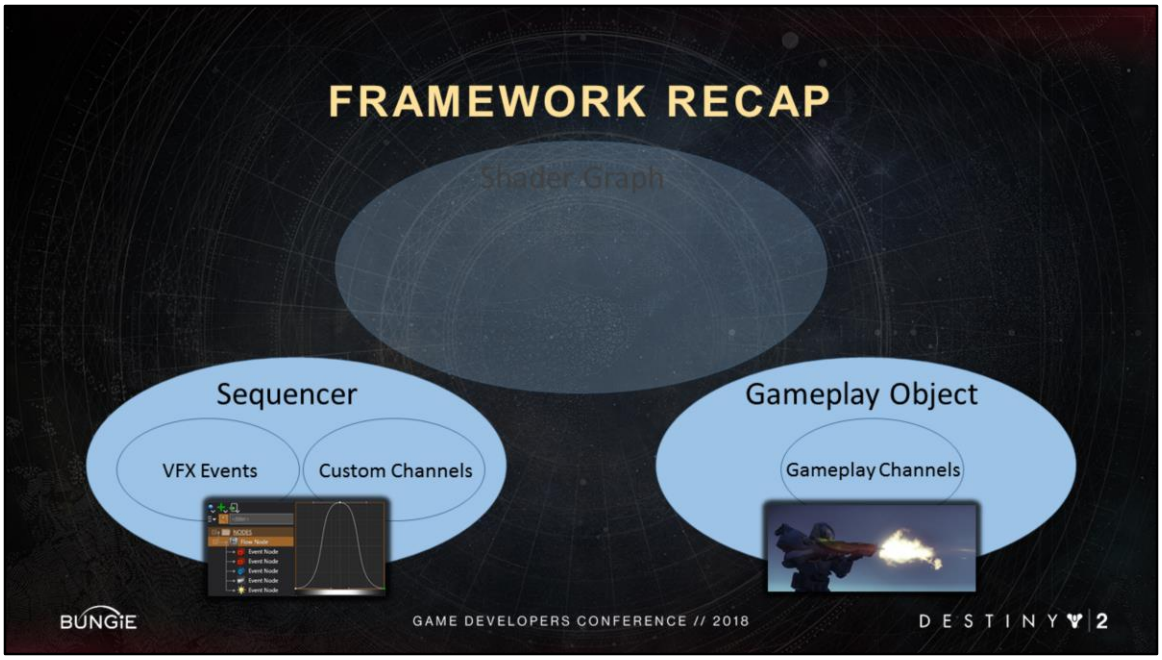
The best part here is that designers can change around the duration of the custom channel and everything just works.



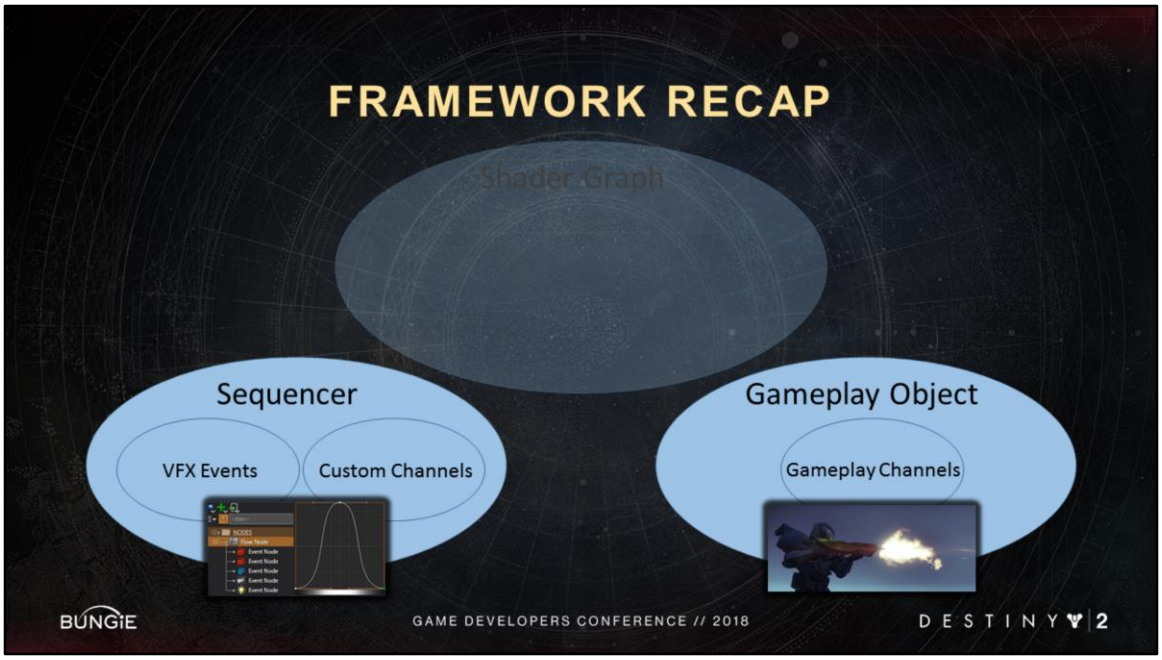
Going back to our conceptual chart. This is where we left things off



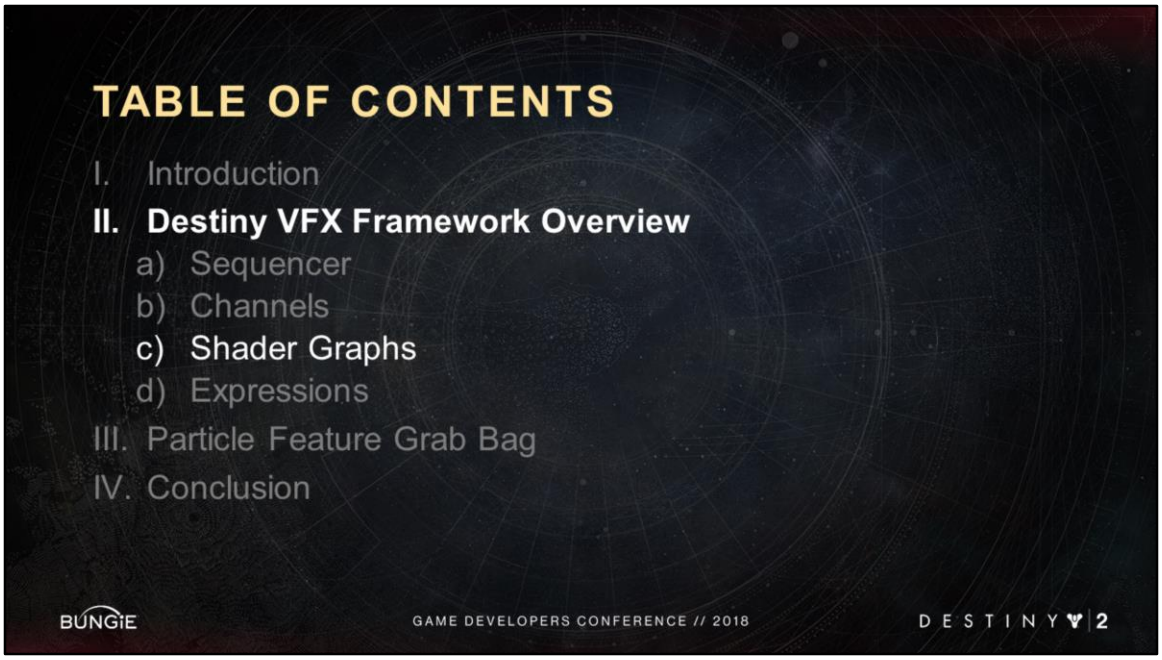
We saw how sequences can define their own custom channels
Like with the vex boss



We saw how gameplay channels
<Advance Slide>
Like damage type and camera type can be used by the sequencer

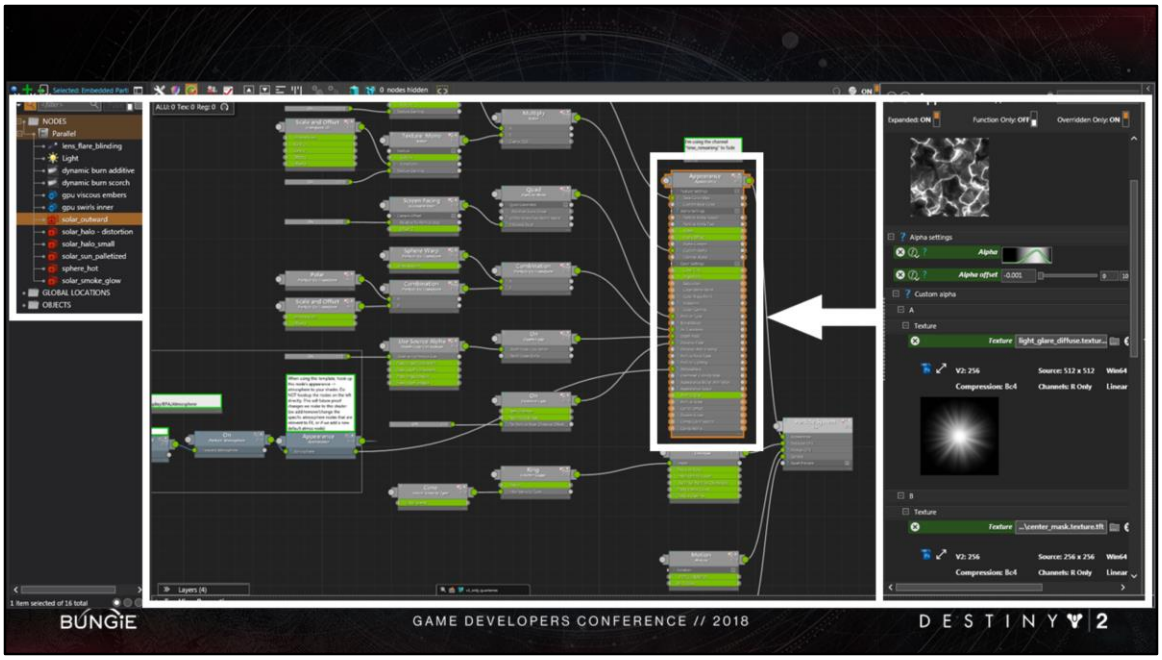


And with that we've unlocked the next part of our overall framework.

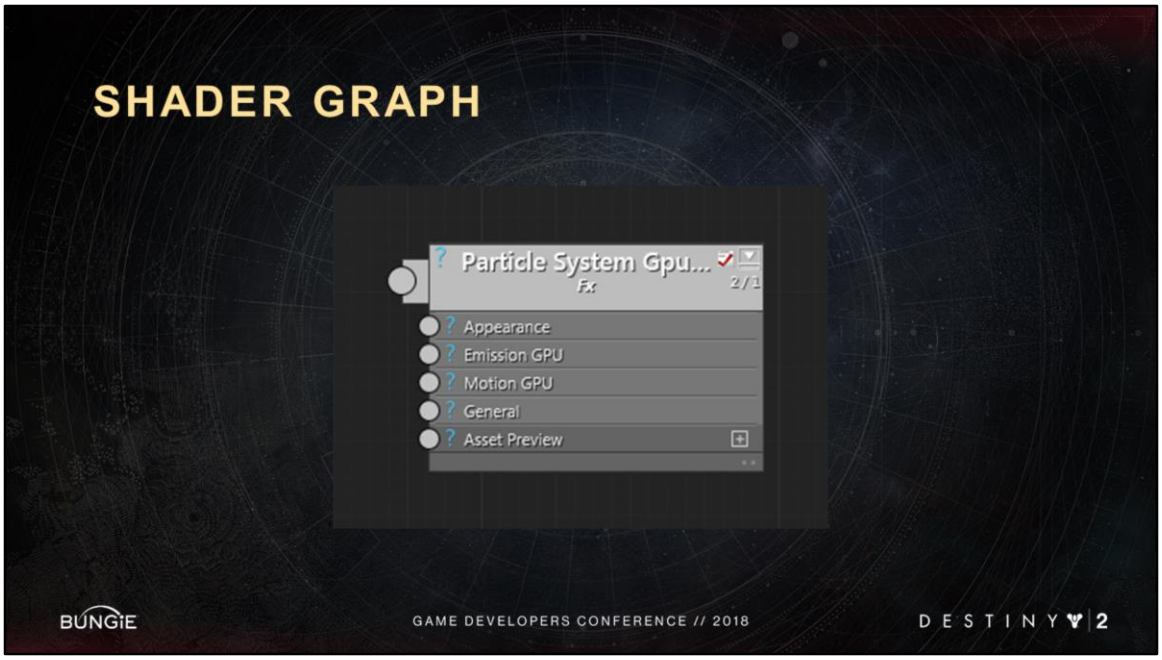


- The next major tool I want to talk about is the shader graph.

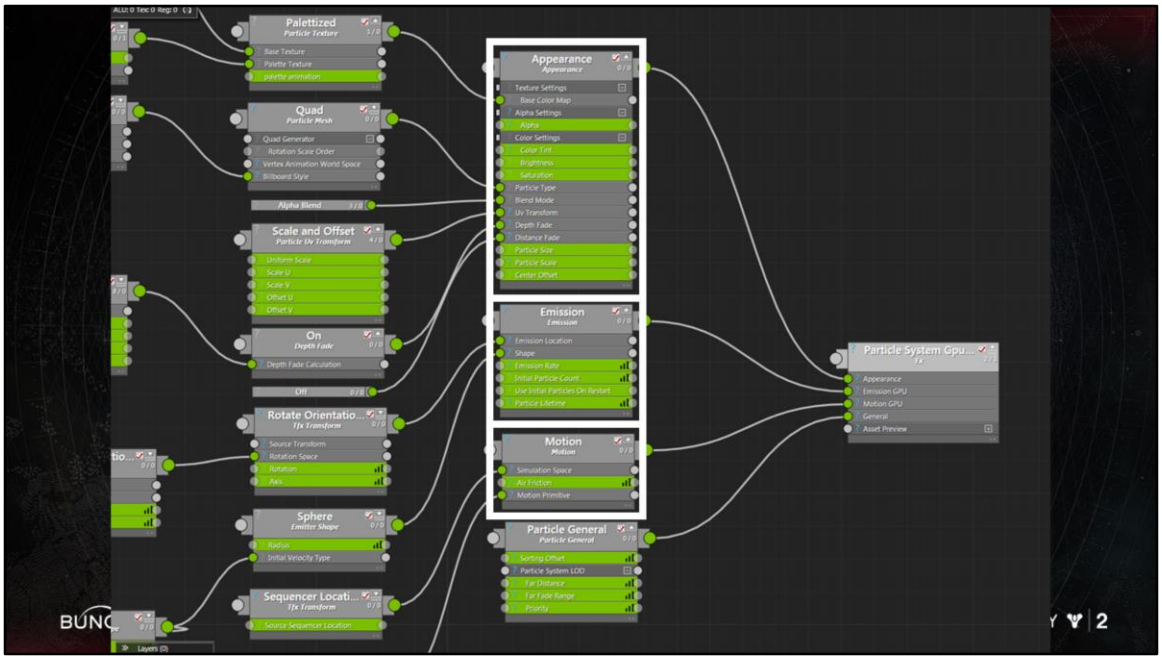
Going back to our the thermal flare explosion sequence
<Advance Slide>
We've talked a lot about vfx events, but what's inside them?
If I click one of these events, you would see:
<Advance Slide>



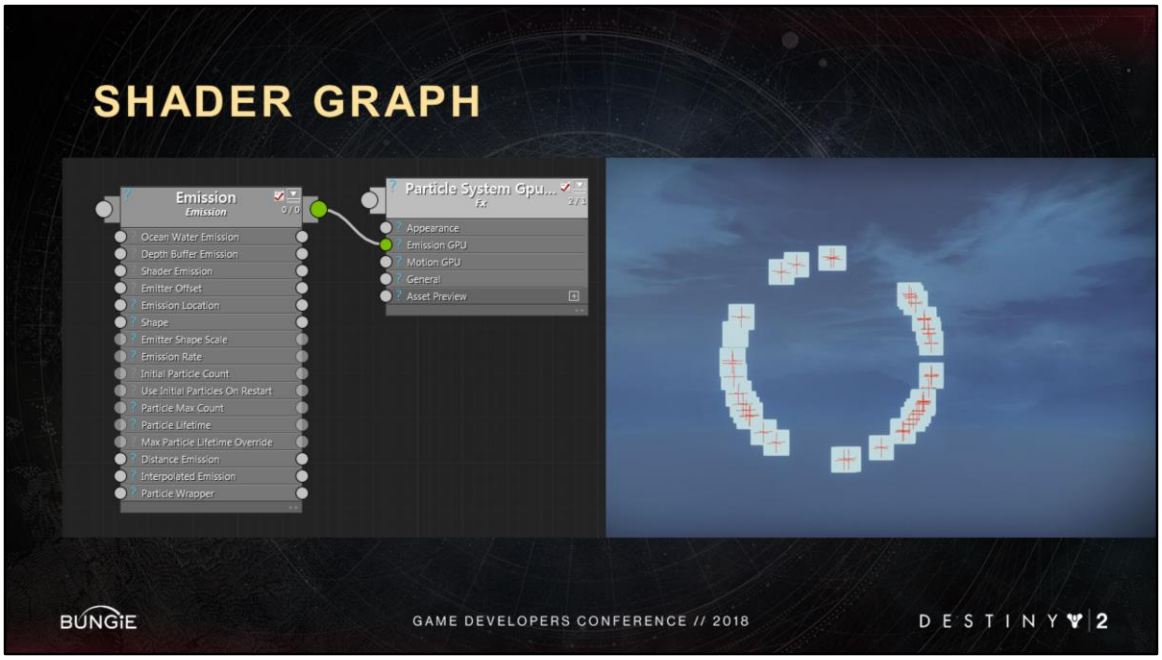
A shader graph embedded directly in the sequencer.
<Advance Slide>
On the upper left you can see the tree view of the sequencer with an particle system event selected.
<Advance Slide>
In the center you see the shader graph of the particle system.
<Advance Slide>
On the right you see the shader property panel of my <Advance Slide> selected shader graph node



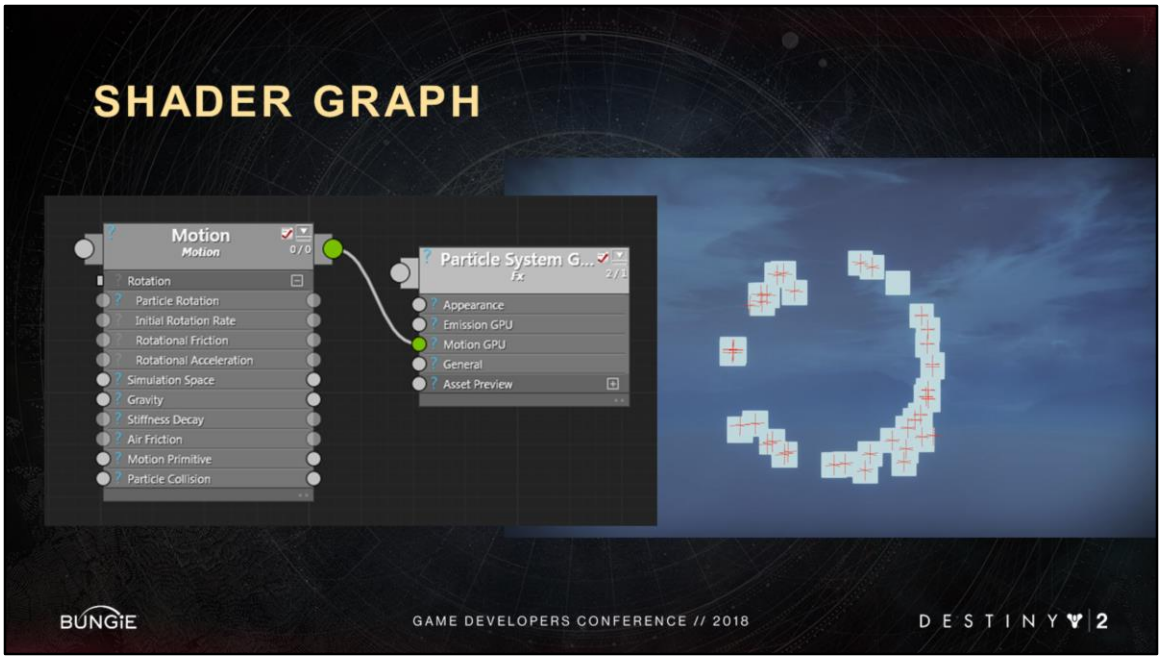
This is the particle system shader root node with its primary components.



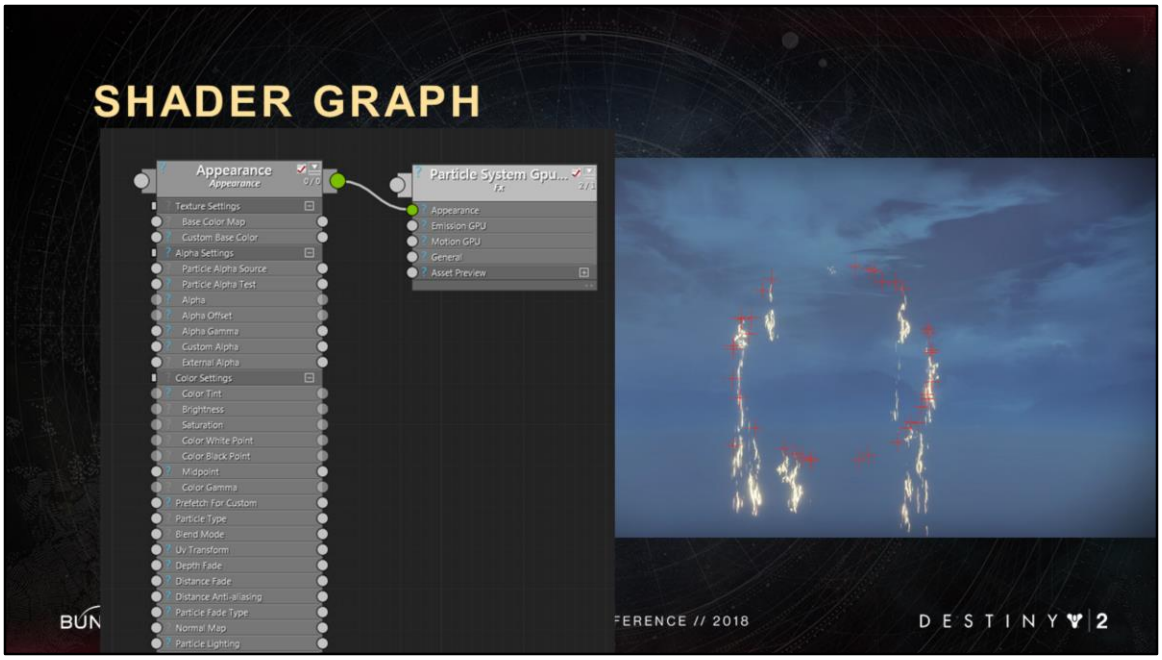
And this is what a basic particle **system** node looks like.
This shader graph not only defines the appearance of a particle,
<Advance Slide> but also defines emission
<Advance Slide> and motion
<Advance Slide> this node graph controls **ALL** aspect of the particle system.



Emission defines the initial state of particles as they spawn.
where we create them
what initial velocity they get
In this example we have a ring emitter, the particle are set to pick a random position along a circle

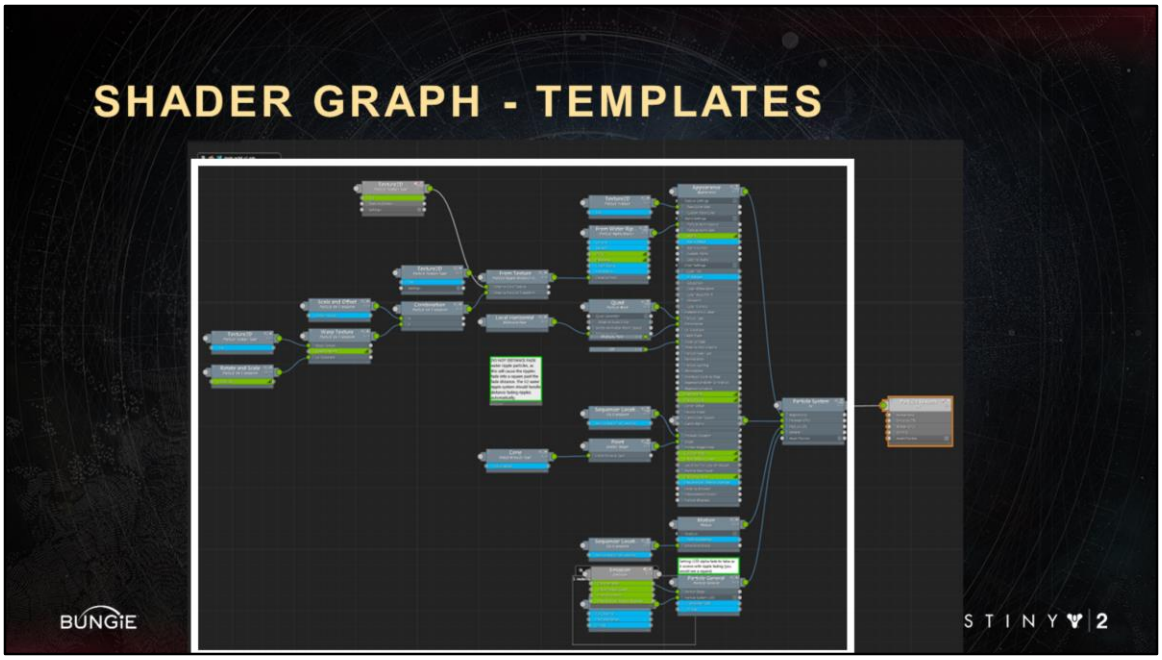


Motion defines how particles move over time.
It controls things like Rotation, Gravity, Acceleration, Air Friction
In the example we added some randomized gravity

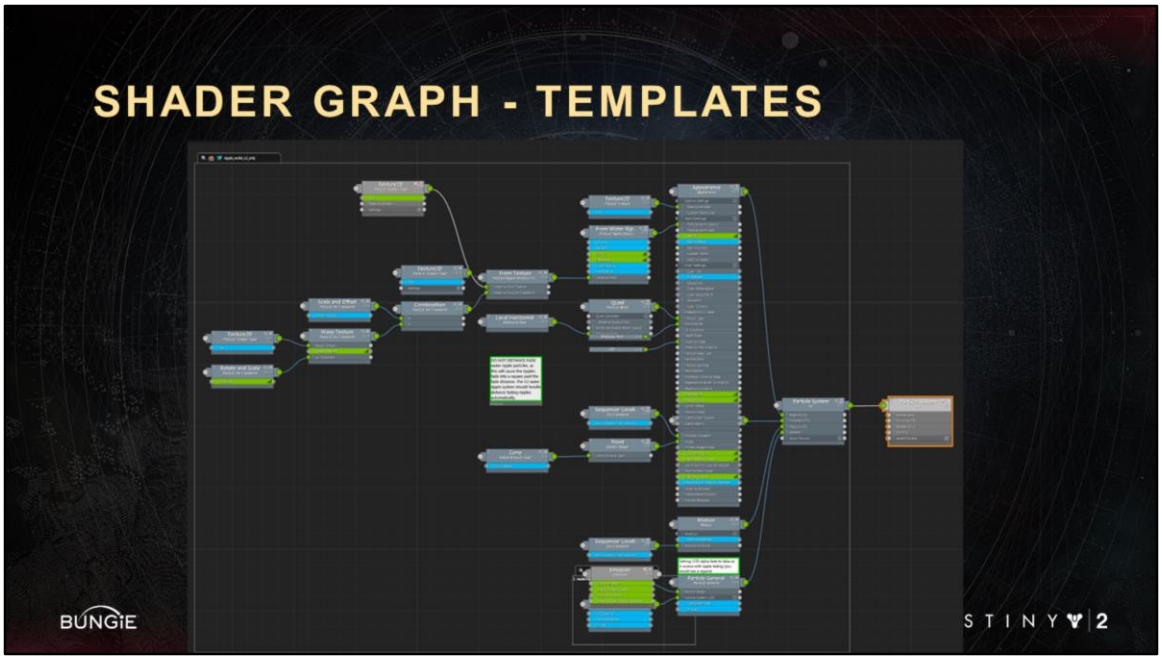


Appearance defines how we render each particle.

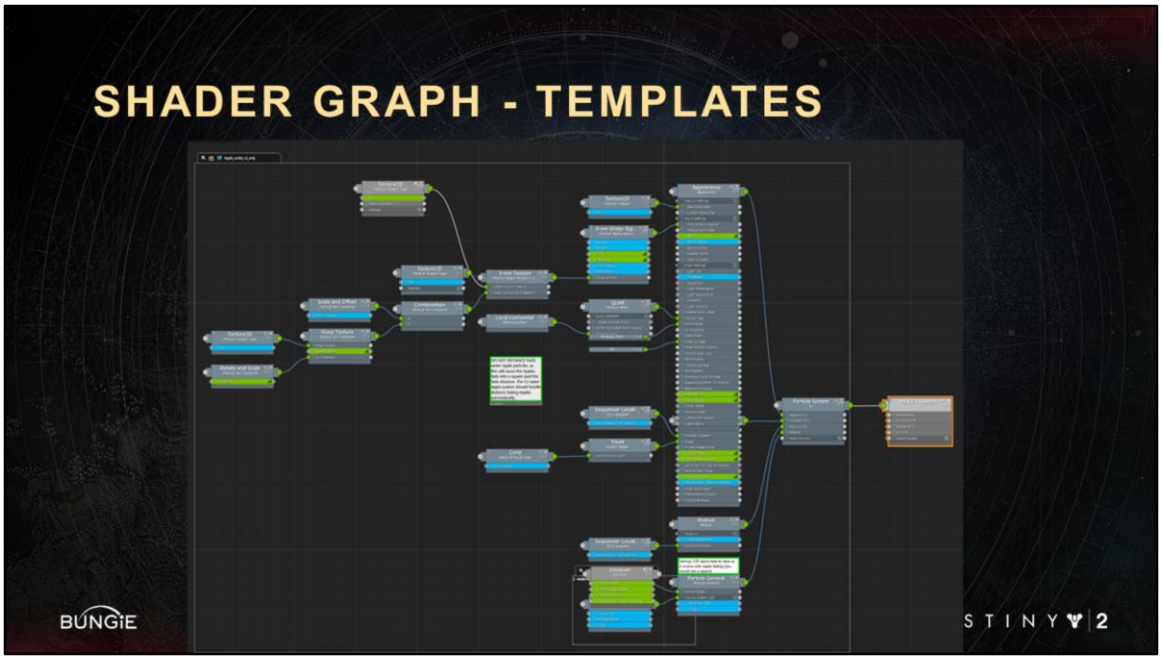
Appearance controls the color, alpha, blend mode, lighting, size, etc…



Here is a zoomed-out view of our simple particle system node graph.
What is this box?
That is what we call **a template**



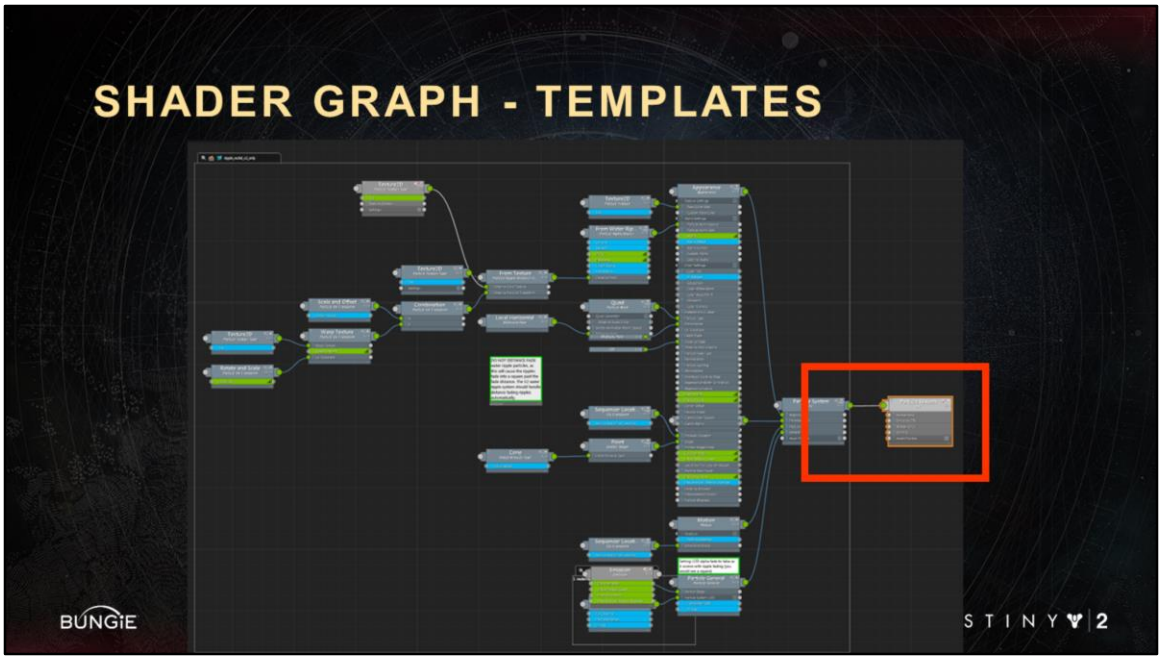
Artists can drag shader graphs into any other graphs, to inherit from them.
Nodes in a box all belong to one shader that is being templated.



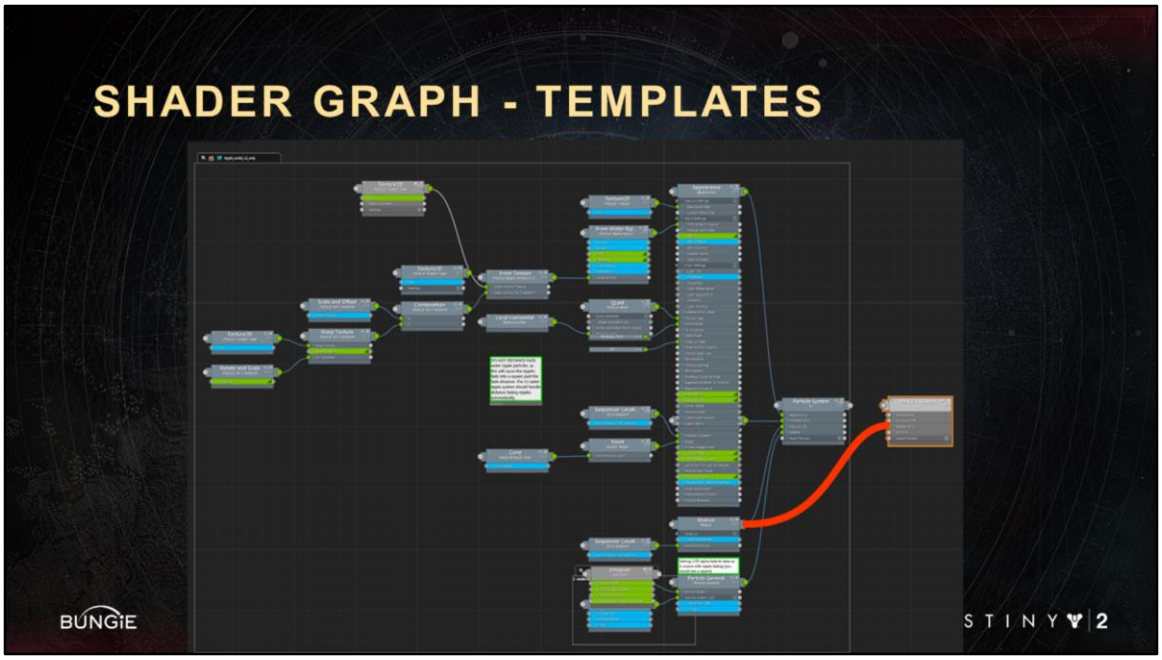
Parameters that have been set in the original shader being templated here appear in blue, to indicate the are being inherited.

Parameters that have been set directly in this shader graph appear green.

I can override any parameter in the template and it will turn green.



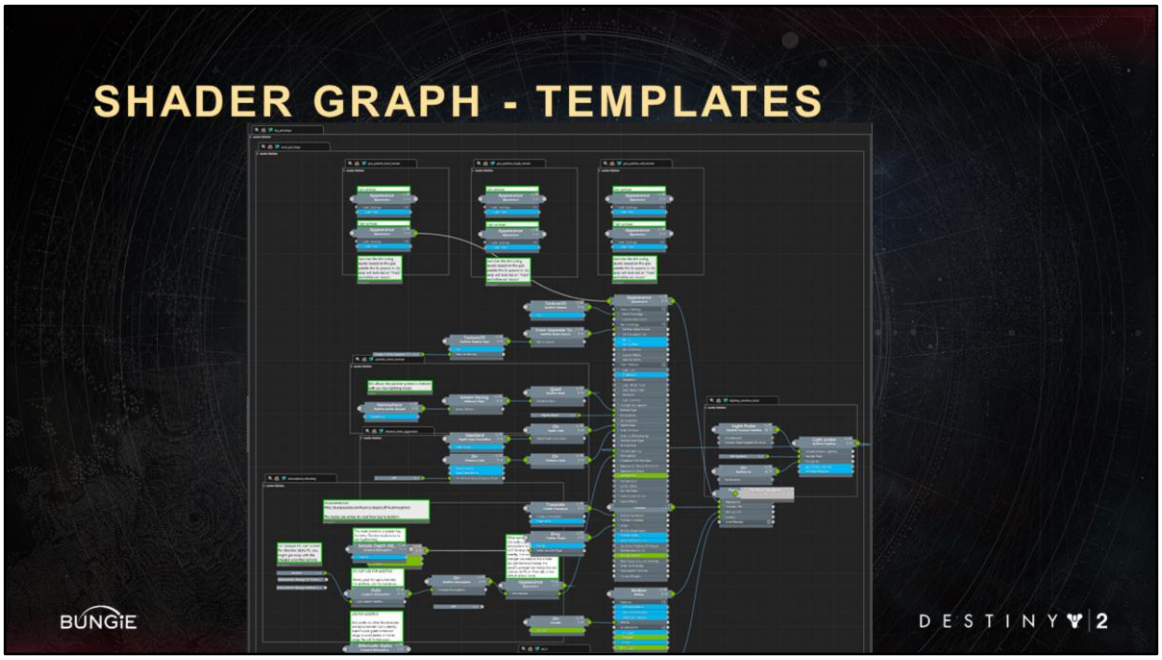
Artist can inherit all of a template like I am doing here
<Advance Slide>
With this one wire connection



Or they can selectively inherit specific nodes within a template, like the motion node here

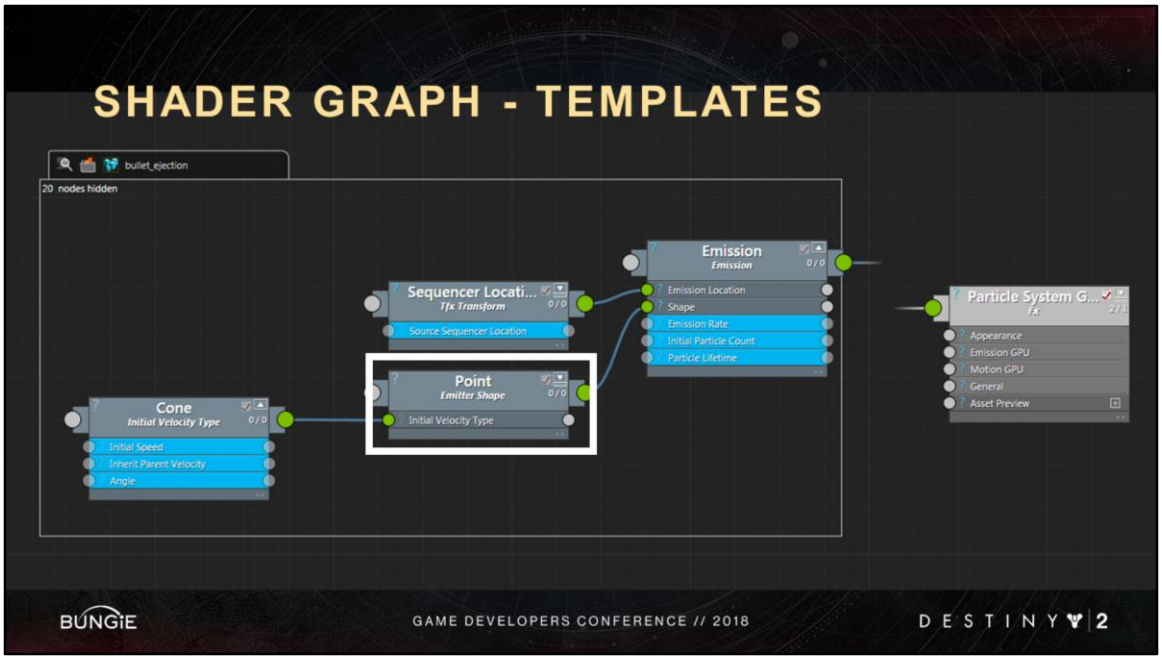
<Advance Slide>

Selective inheritance.



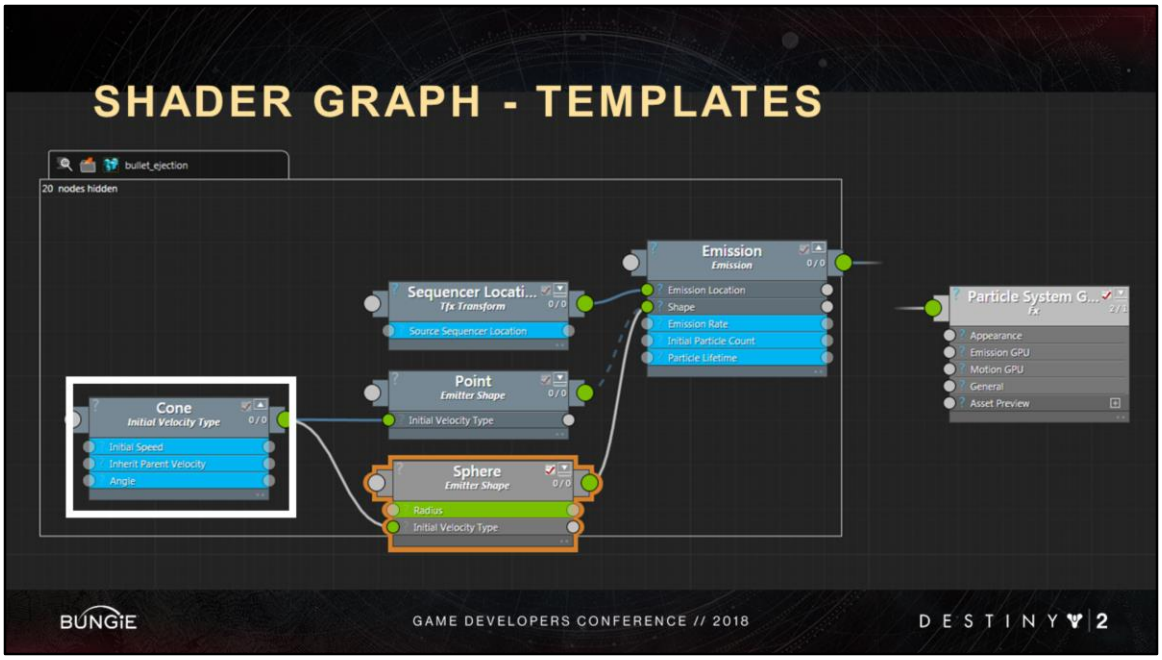
You can drag in as many templates in as you need.
This one shader here is built from a ton of other templates.

This might seem messy but with a few key strokes, artist can hide anything they don't want to see
They can take a messy graph like this and reduce it to this

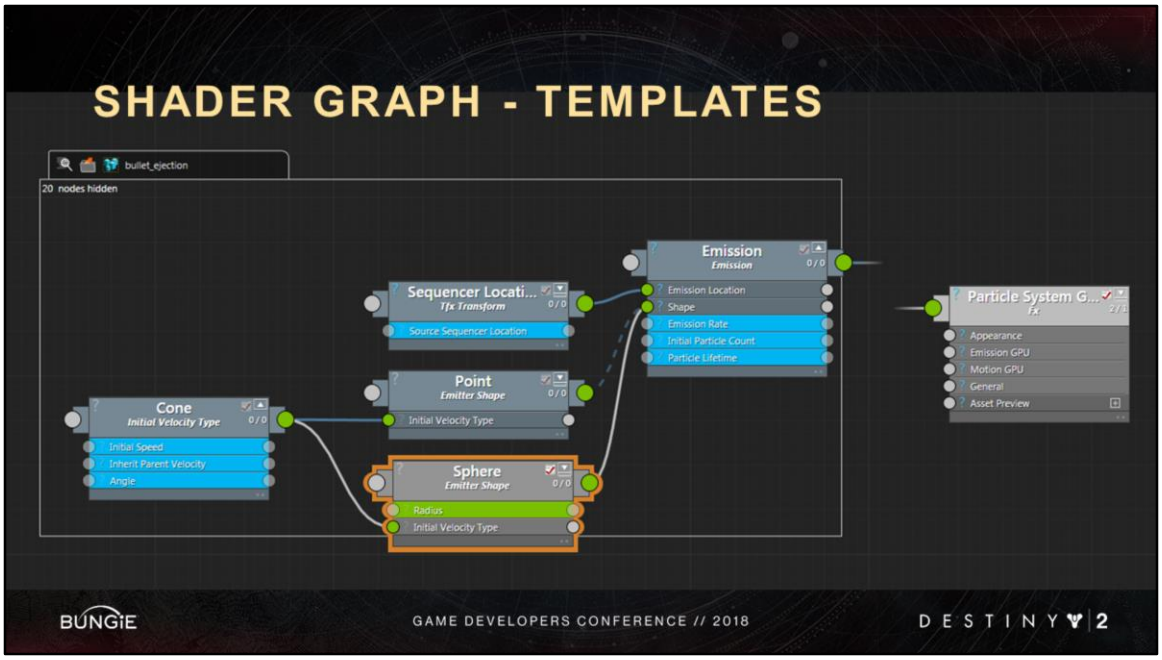


But **most** importantly, Artists can entirely bypass nodes as needed

For example, in this emission template we can bypass the point emitter
<Advance Slide>



And change it to a sphere emitter
While still inheriting the velocity <Advance Slide> from the original template
You can’t do that with blackboxing



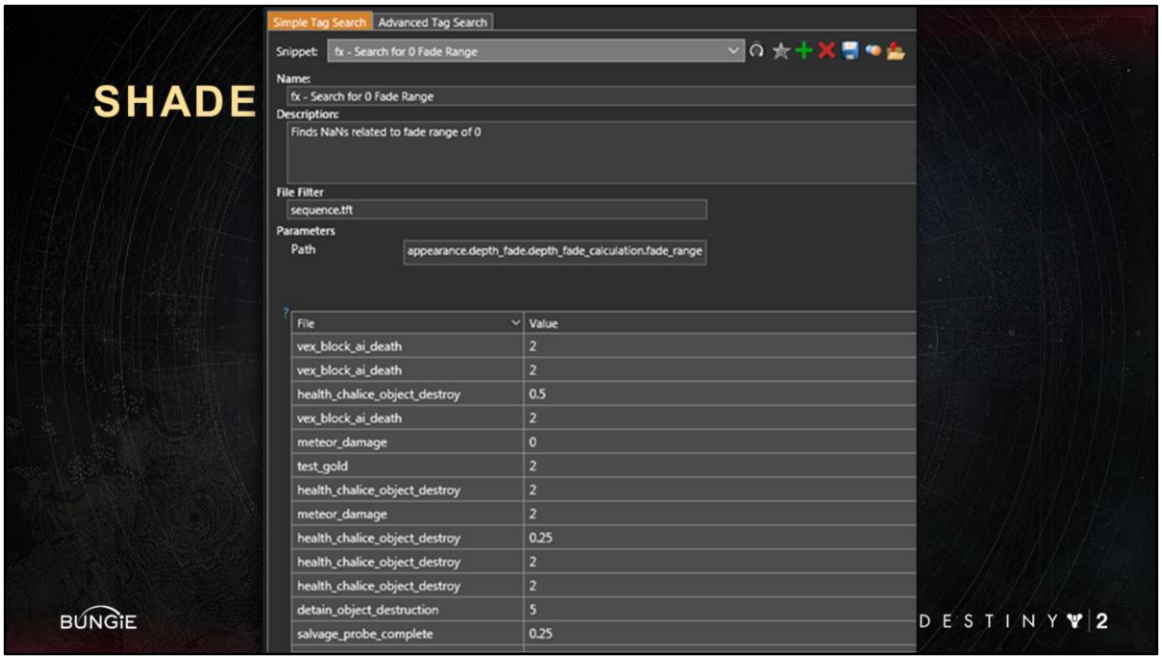
Templating allows us to build shaders is a very modular and resuable way across all our shader types.

Templating goes beyond blackboxing because artists can bypass templated nodes and appending new ones as needed.

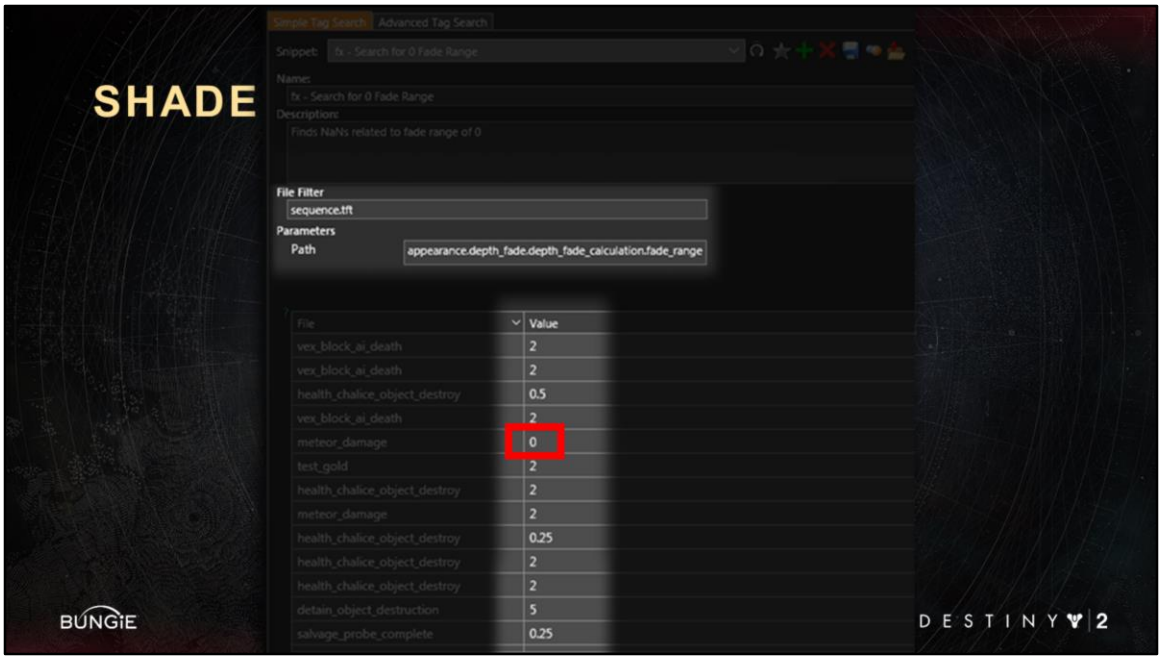
Artists can inherit all or some nodes.

As new features come online, we just modify a few templates to propagate changes across the entire game.

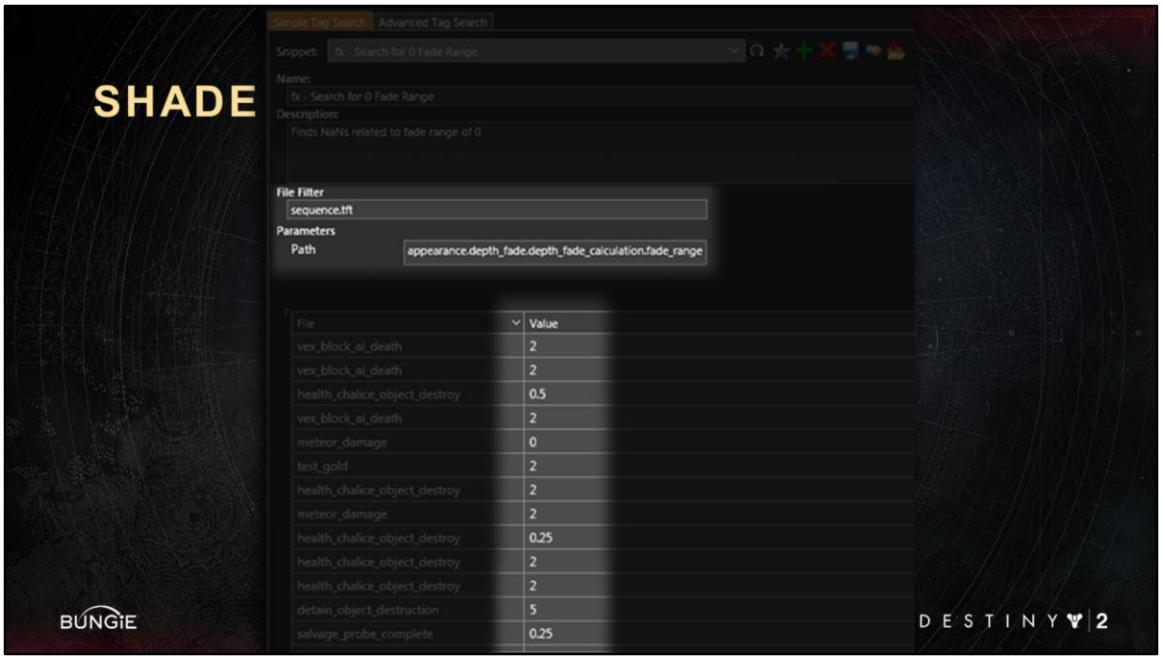
Tons of content management power here.
Brandon will also show more examples of templates.



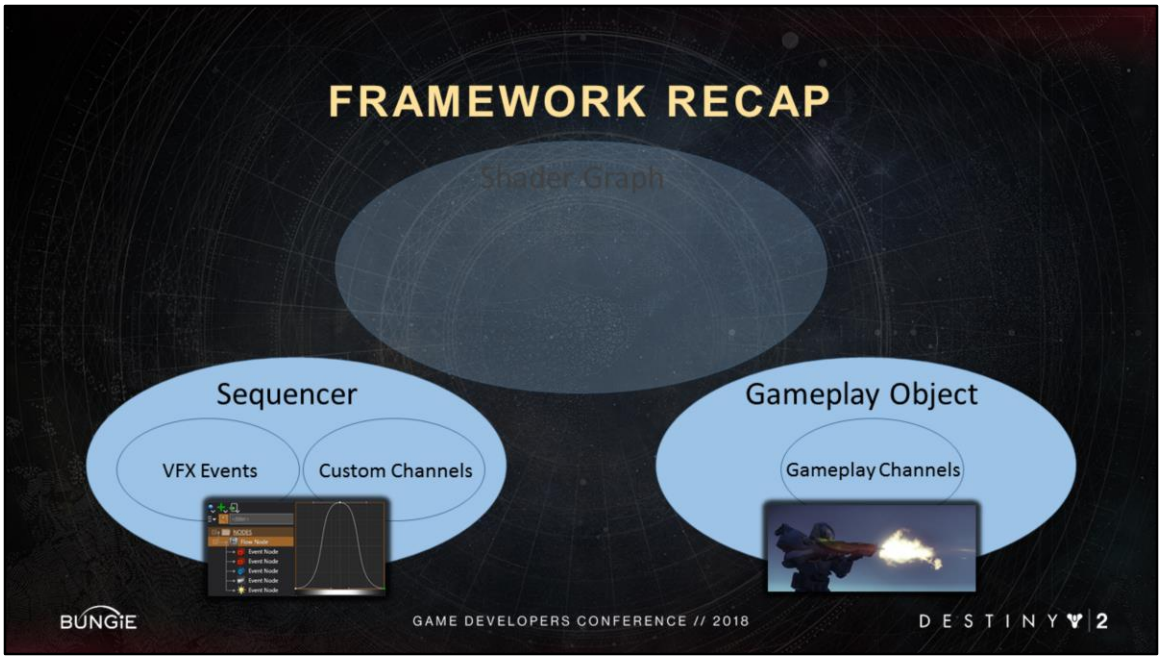
One last aspect of our node graph that I want to mention, is our ability to run audits.
We can search for shader parameters, and show their values
Across the entire game



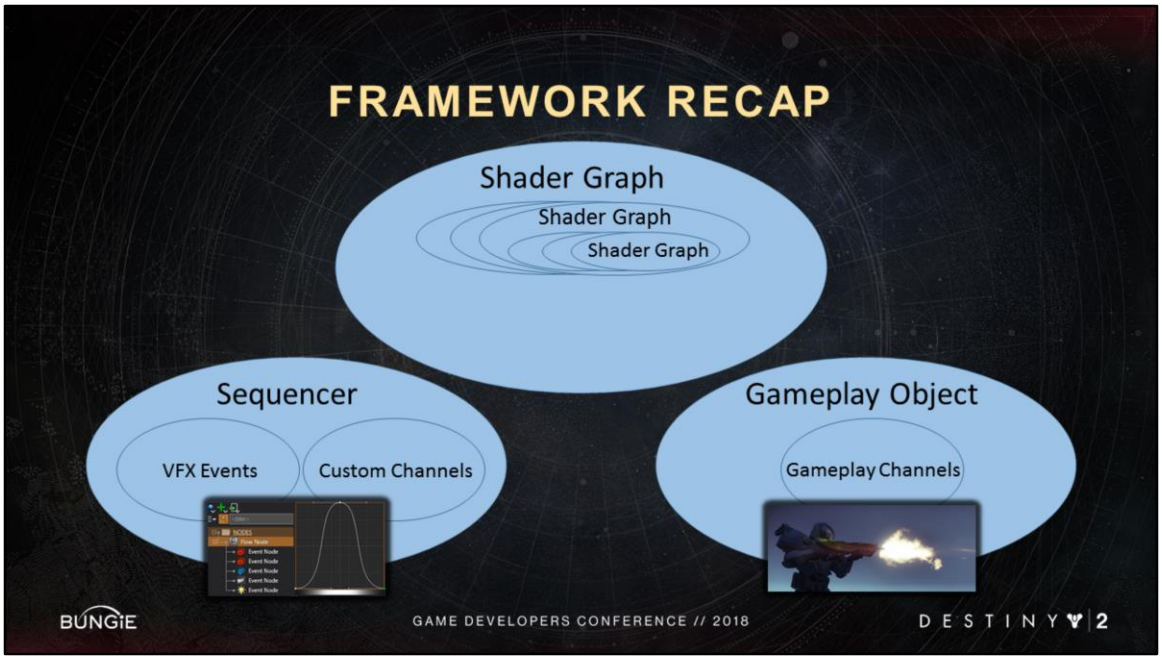
here I am looking at the “fade range“ parameter
And this one value here is bad, it will cause a divide by 0 in the shader, which causes graphical corruption artifacts on the consoles
Auditing our shaders really lets us proactively find outliers
1) sanity check values
2) find bugs
3) Optimize



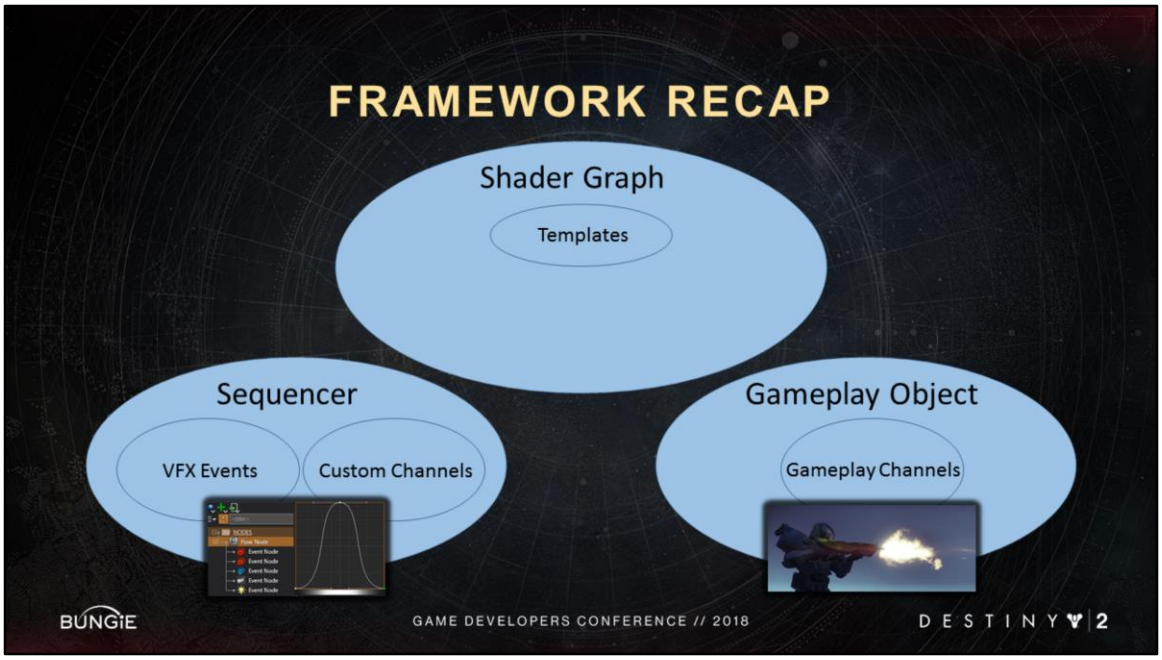
If needed we could also update the script to fix up the values it finds.
As a TechArtist, I am constantly evolving our shader graph nodes in response to feature development, feedback, optimizations.
It is super valuable that we can fixup our shader values across the entire game if and when we change the underlying source code of any node.



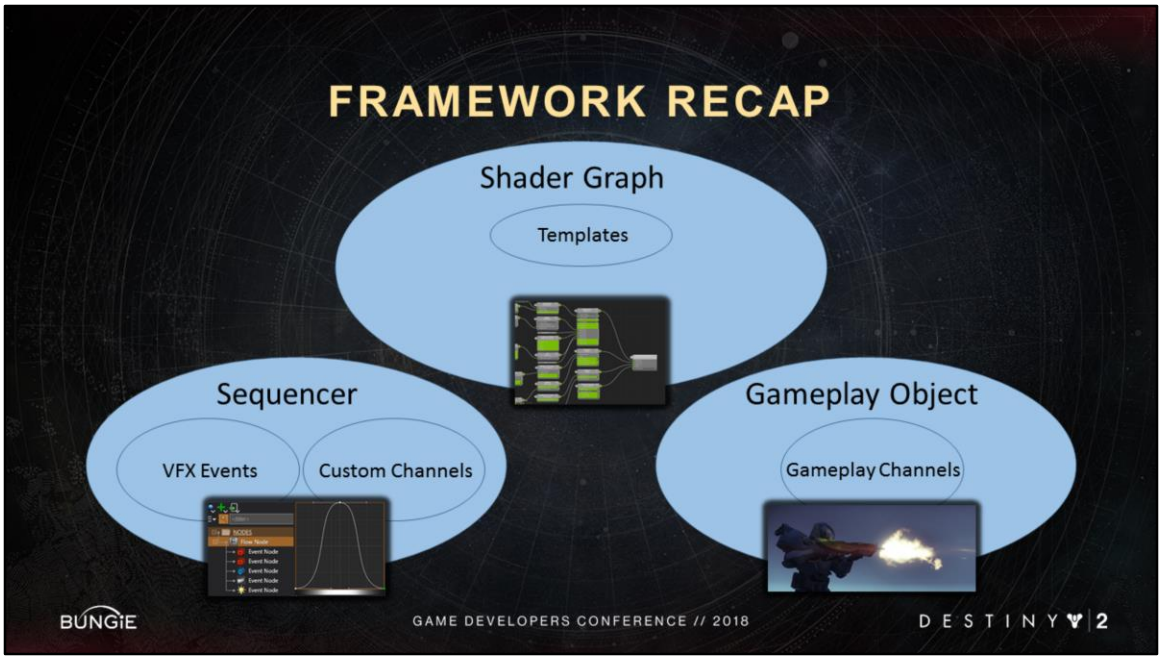
Back to our chart here. This is where we left things off.
In this section we talked about



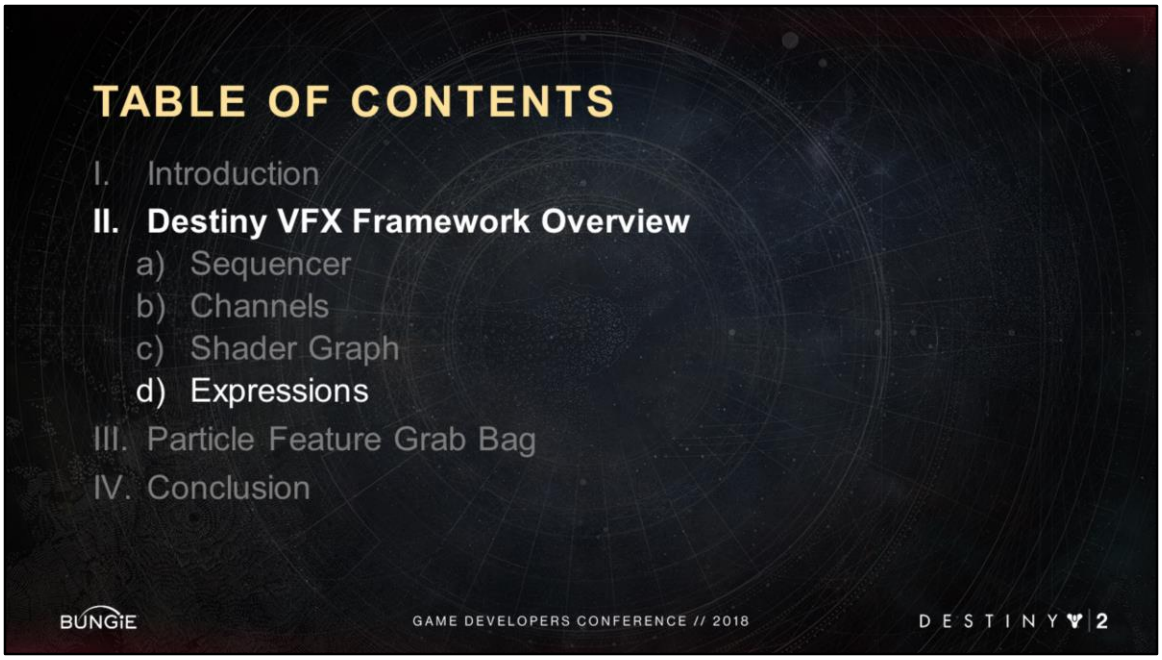
The shader graph and how it can be built from more shader graphs



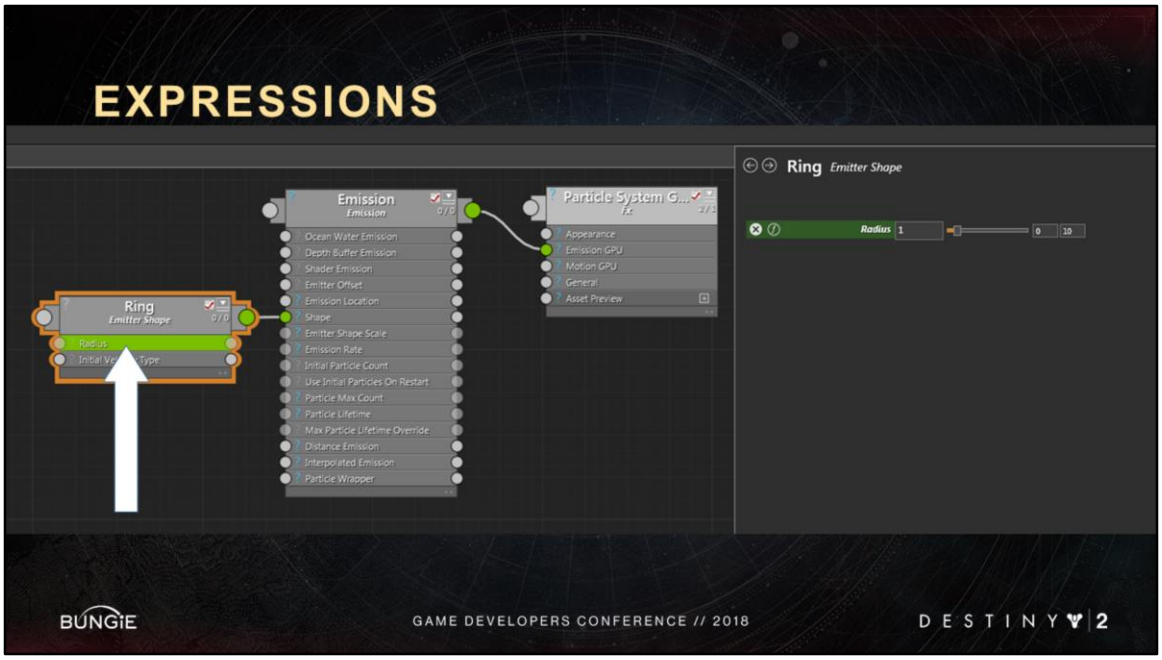
Or this for simplicity



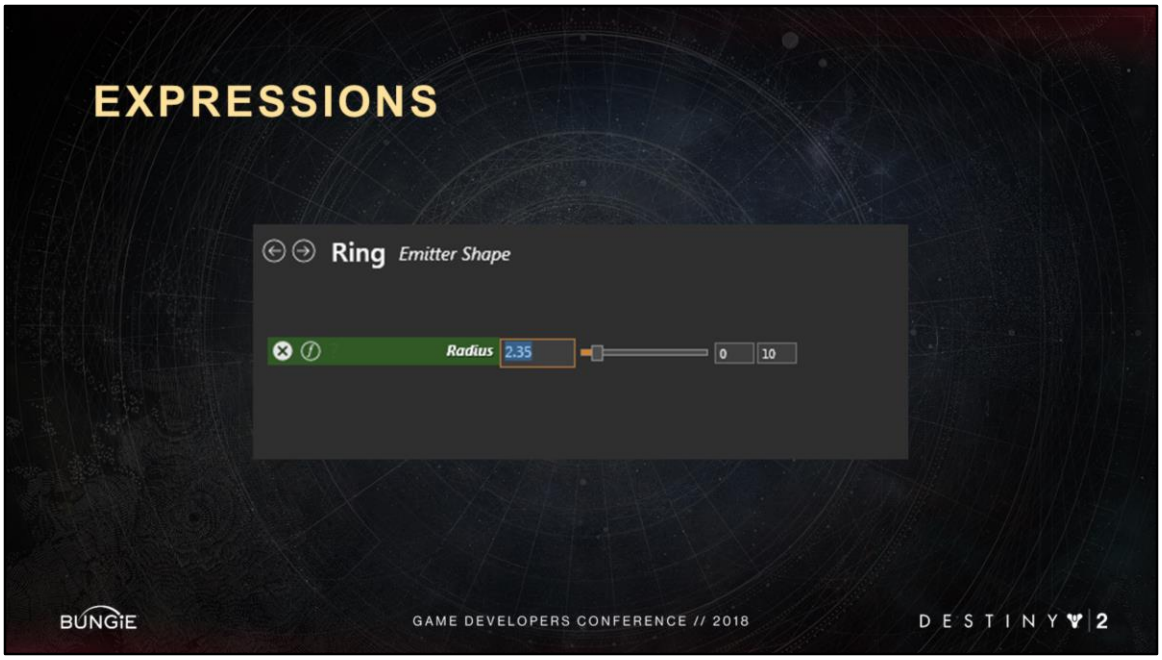
And with that we've unlocked the next part of the framework



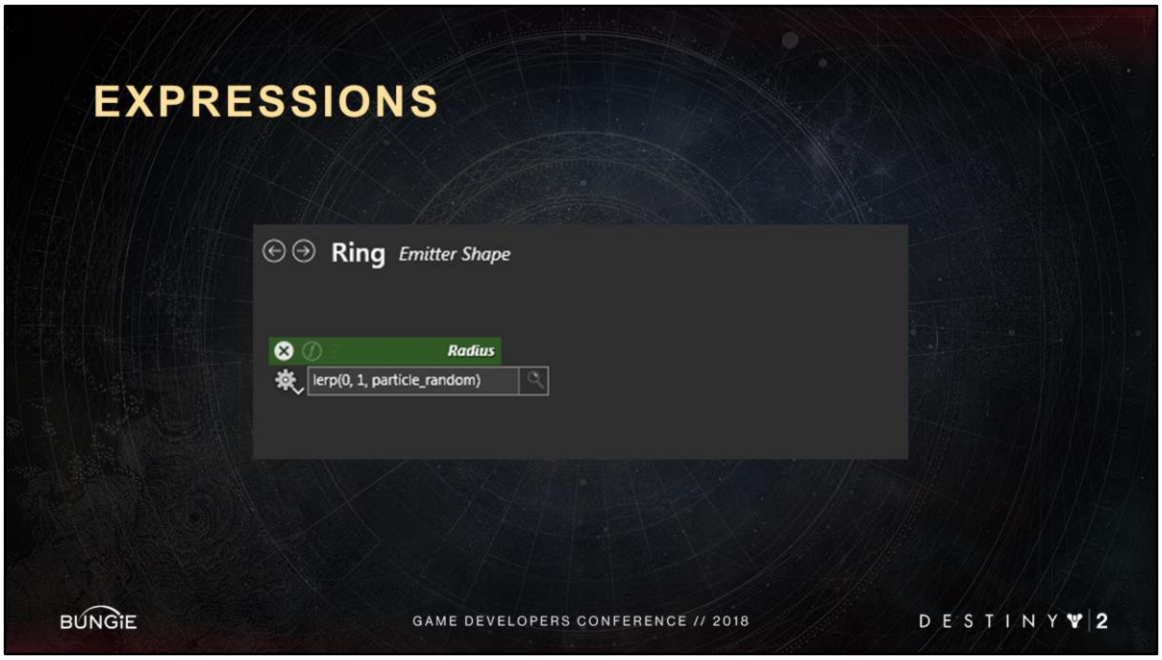
In this **last** section I will talk about shader expressions



In this example, I am looking at the ring emitter shape of my particle system
I select that node and see the properties on the right

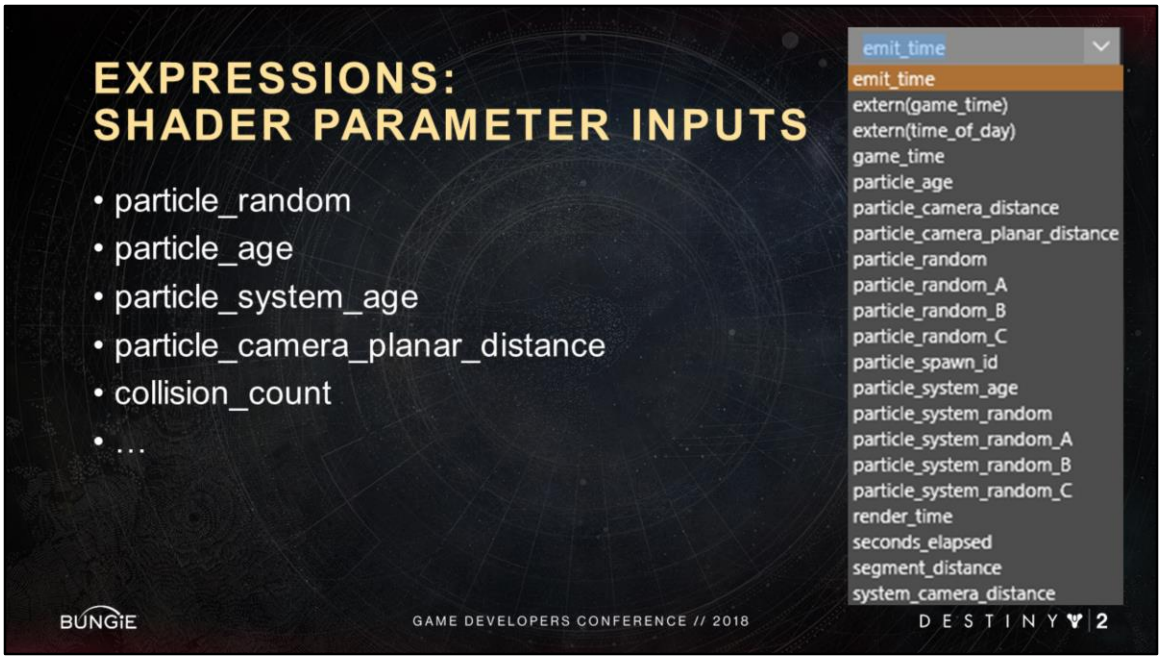


I can set the radius to a constant value of course but I can also

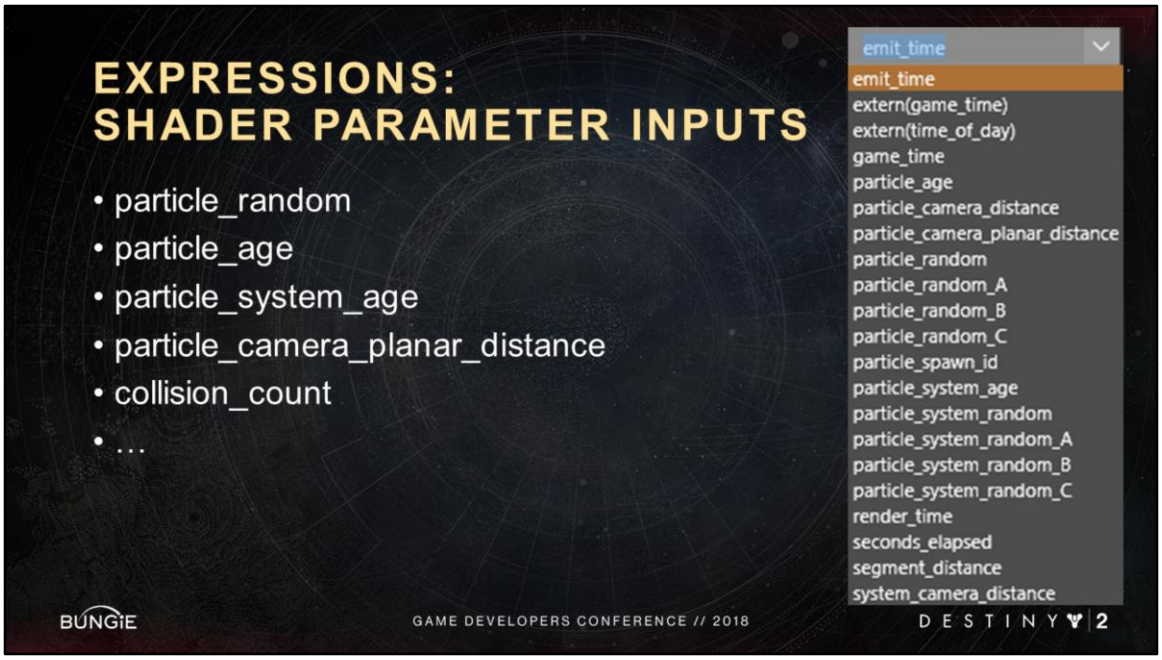


convert it into an expression. In this expression I am lerp'ing between 1 and 5 based on a per particle random.
This would effectively turn the ring emitter into a disc emitter.

So what is this particle_random?
Perhaps **the** most powerful aspect of our fx shader graphs is that they have special inputs relevant to the shader type
For example, in a particle shader, we have:

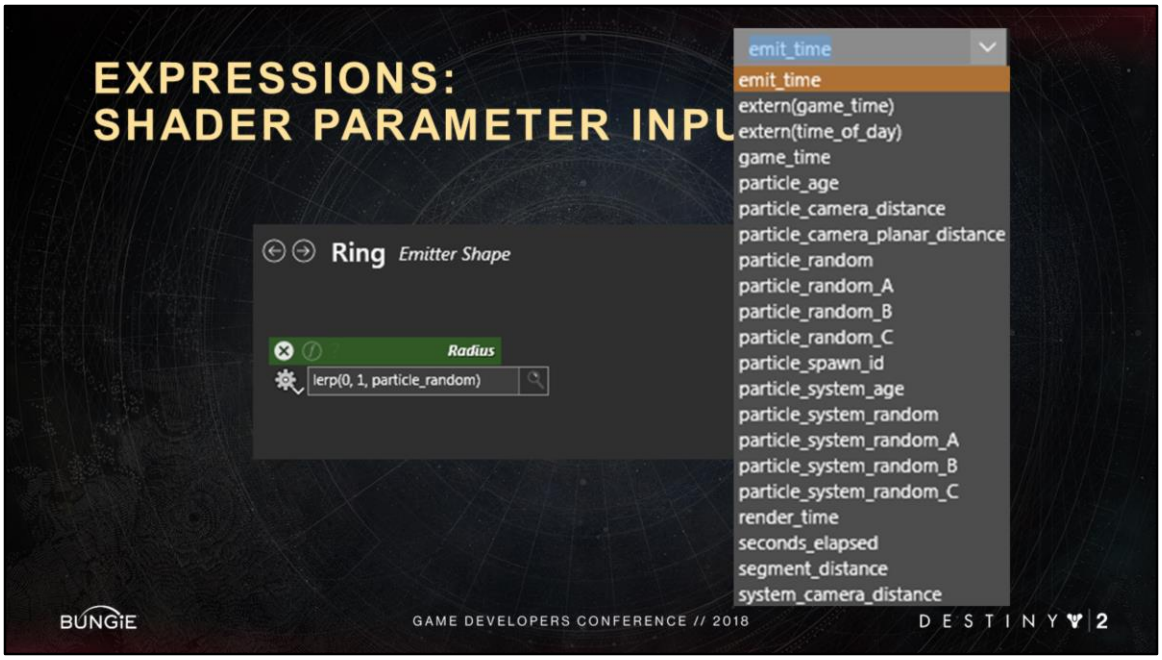


Particle_random: is a random value between 0 and 1.
Particle_age: is a 0 to 1 value, normalized over the lifetime of the particle.
Particle_system_age: is a 0 to 1 value, normalized over the lifetime of the system.
particle_camera_distance, collision_count… And so many more!

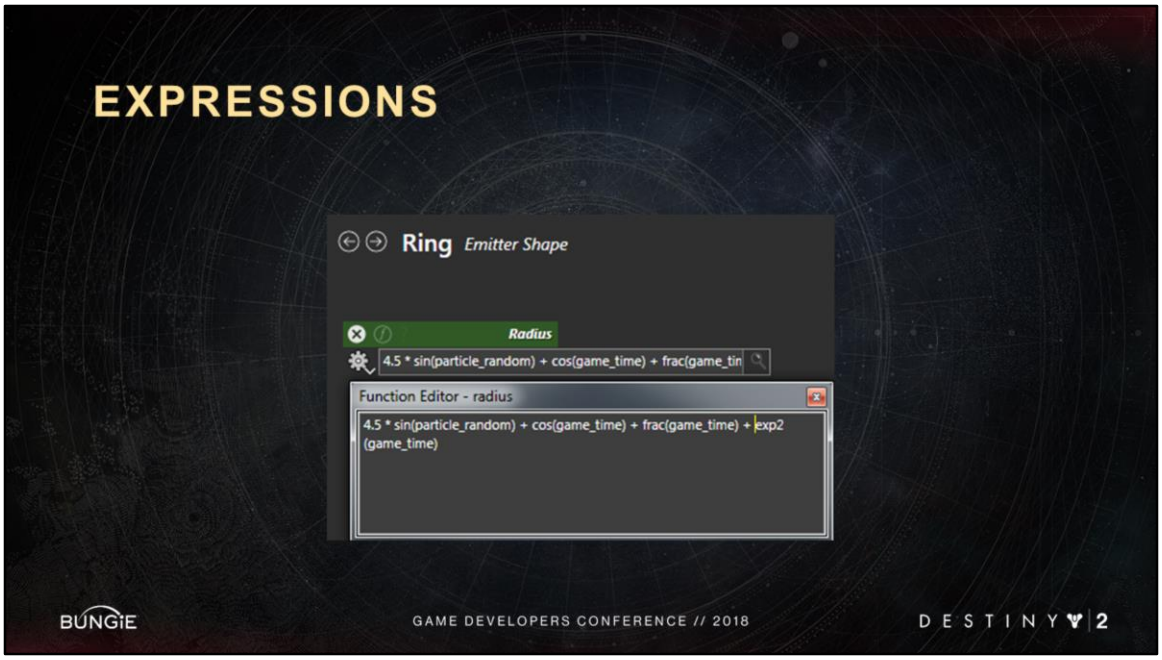


These are just a few of the inputs we can use in our particle shaders to create expressions.

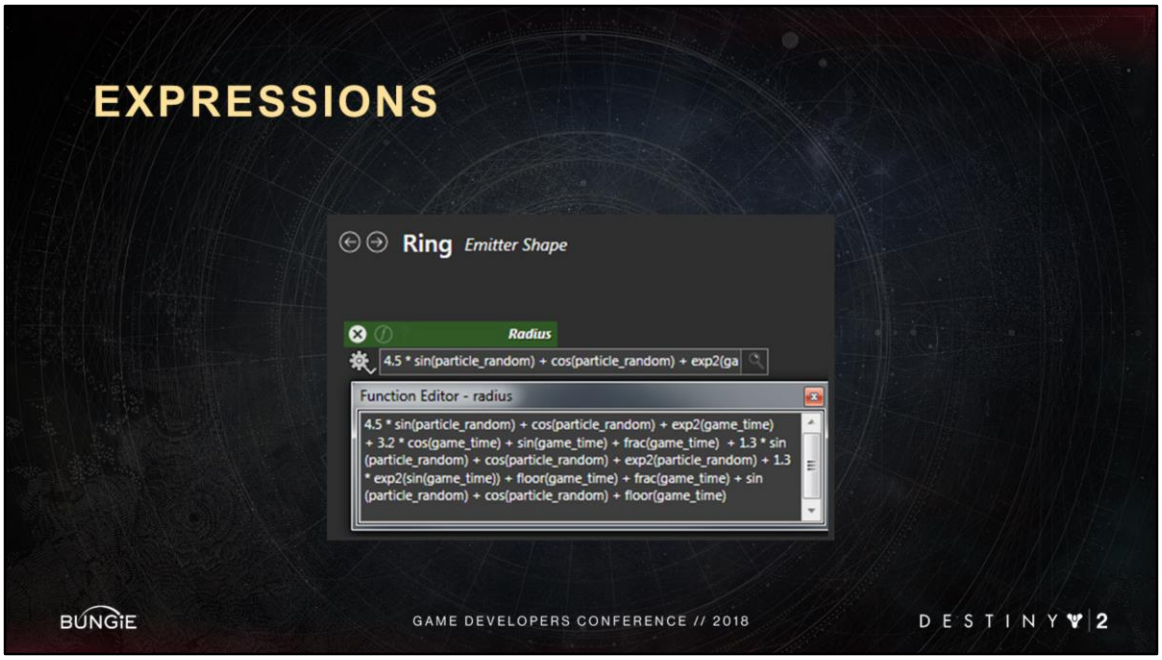
The exact list changes per parameter and shader and is context sensitive.
The main idea here is that **any** parameter in **any** shader has special inputs like these that are automatically available.
For ex: lights we have a light_age shader param input, similar to particle_age.



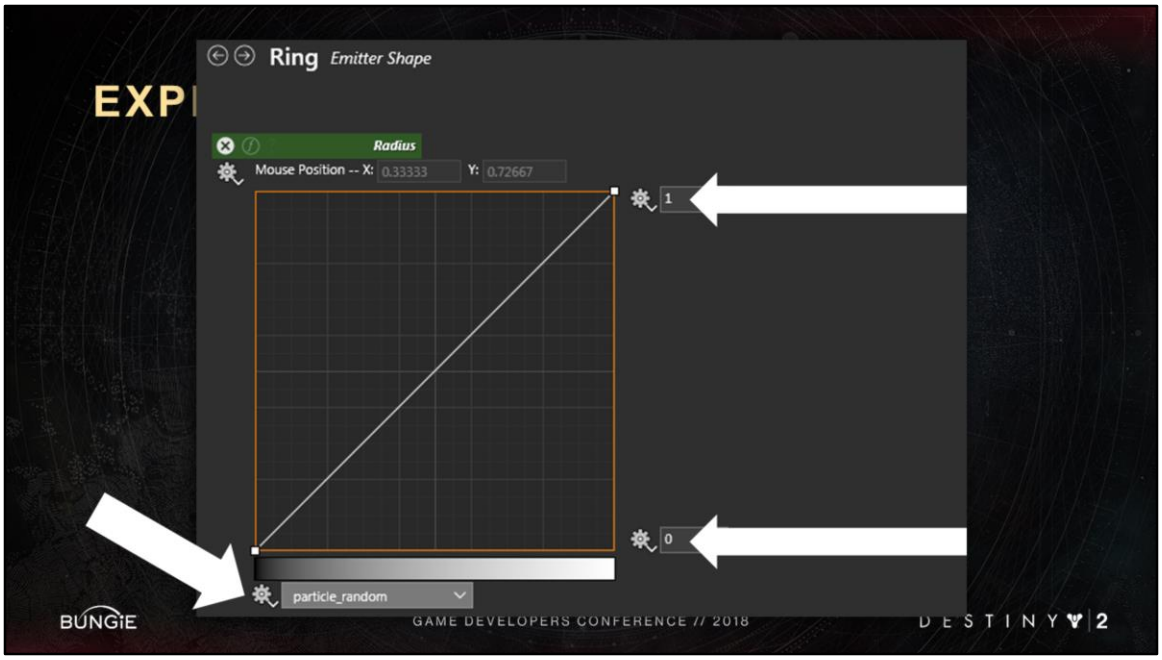
So on the fly, I could type any of these parameters here in this expression and it will update in realtime.



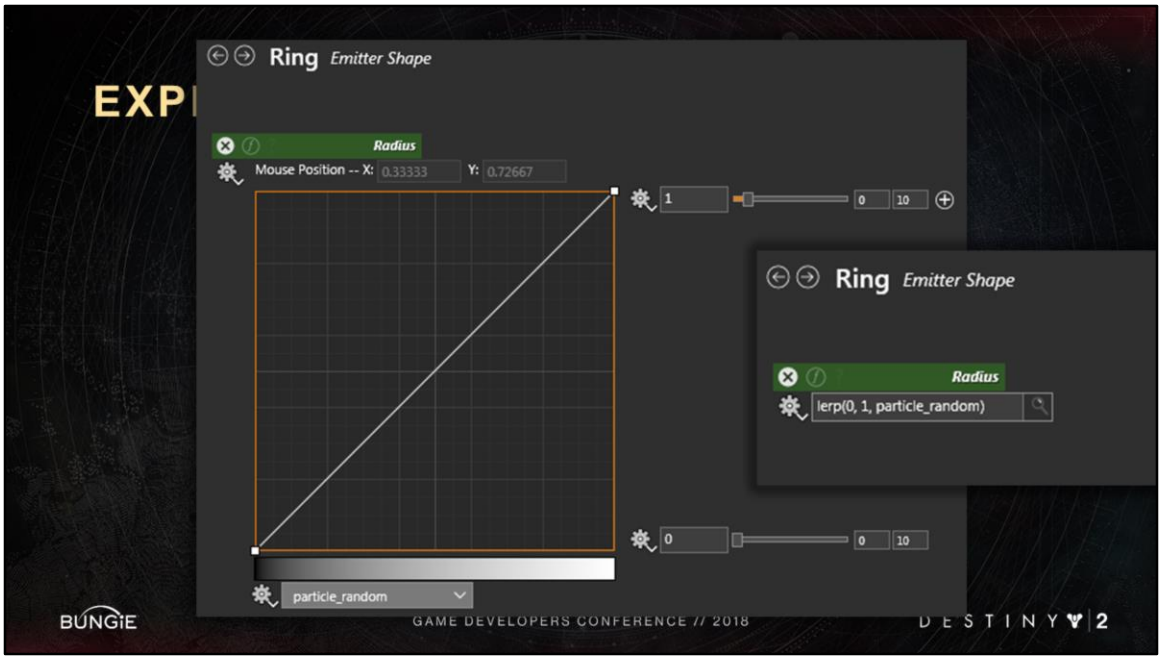
More technical artists could go a little further and type more complex expressions.



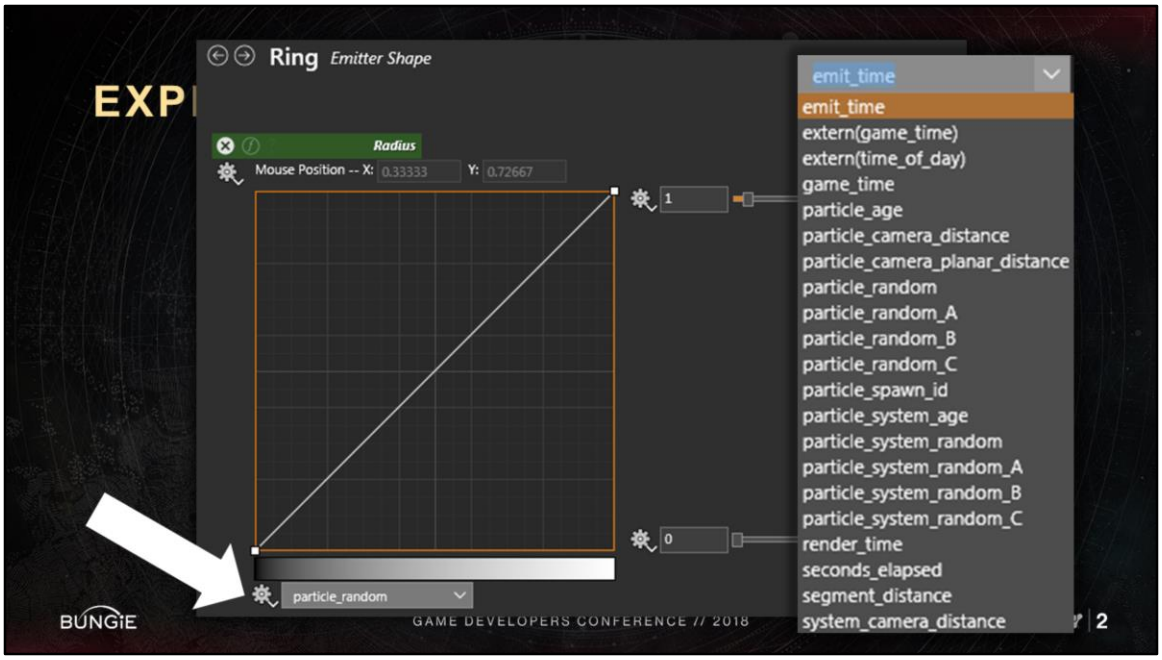
Or they could go a lot further.
Expressions can be as complicated as you want.
And the best part is that they are not evaluated per pixel. They are evaluated on the cpu per frame or on the gpu per particle, so they are cheaper.



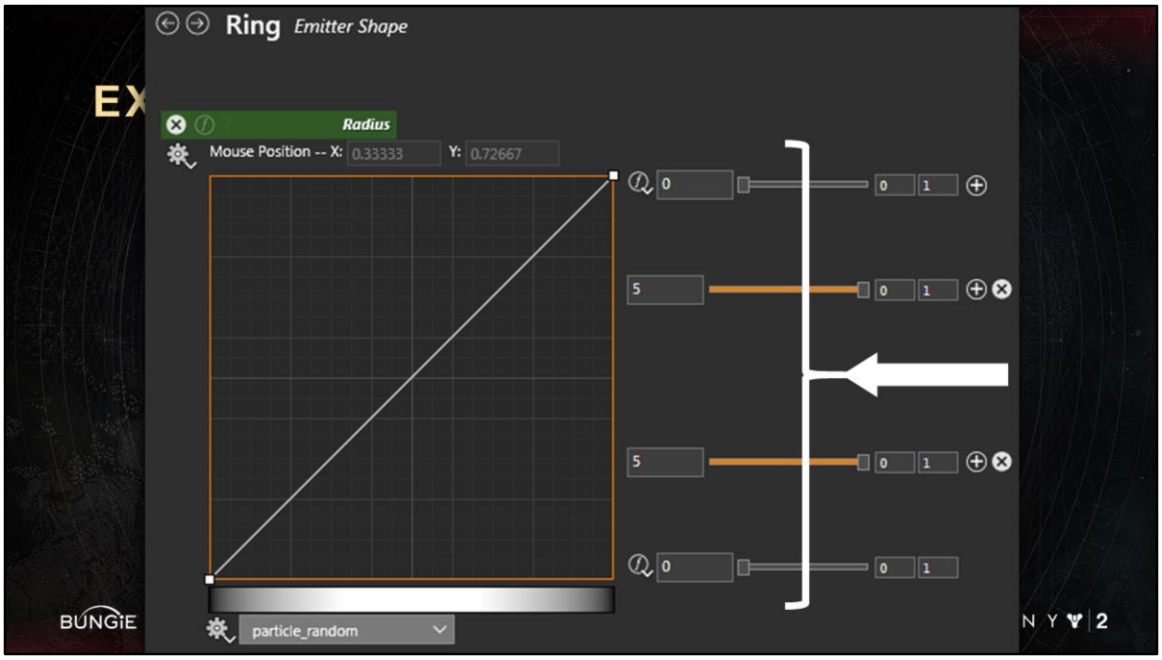
We have alternative UI for creating expressions: our spline editor.
Here is my min
Here is my max
And this is the domain I am blending over



These two expressions are identical by the way
They are both picking a random number between 0 and 1 per particle



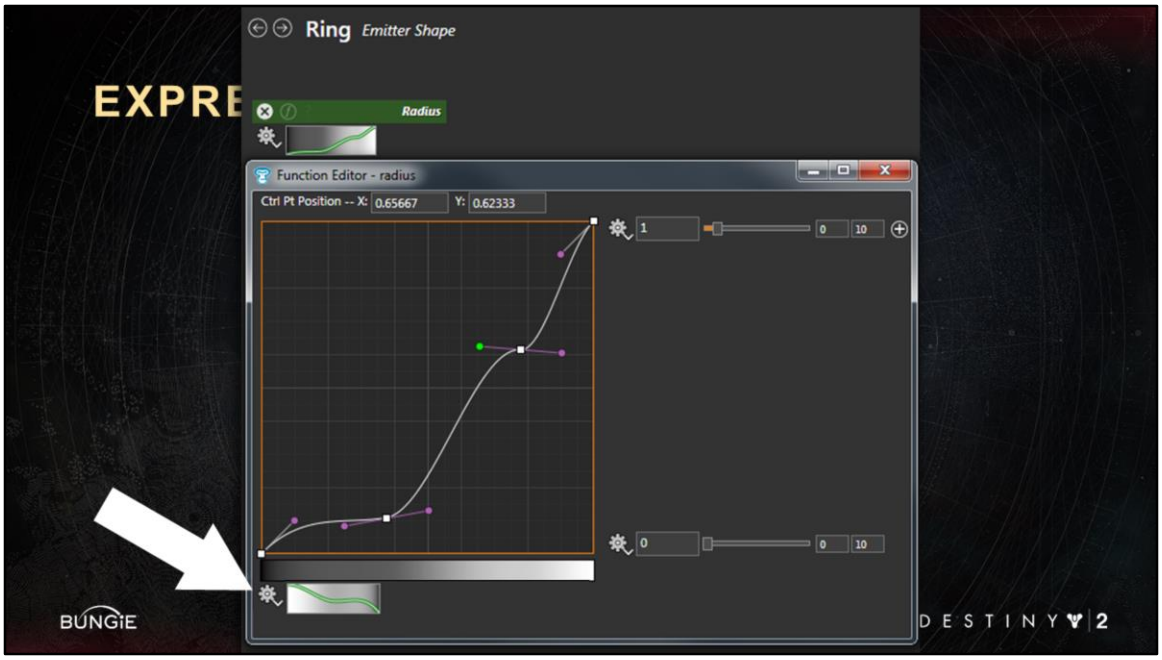
Similar to the typed expressions, artists can expand this domain dropdown list here
<Advance Slide>
And switch up the inputs **at a whim** to experiment with different results



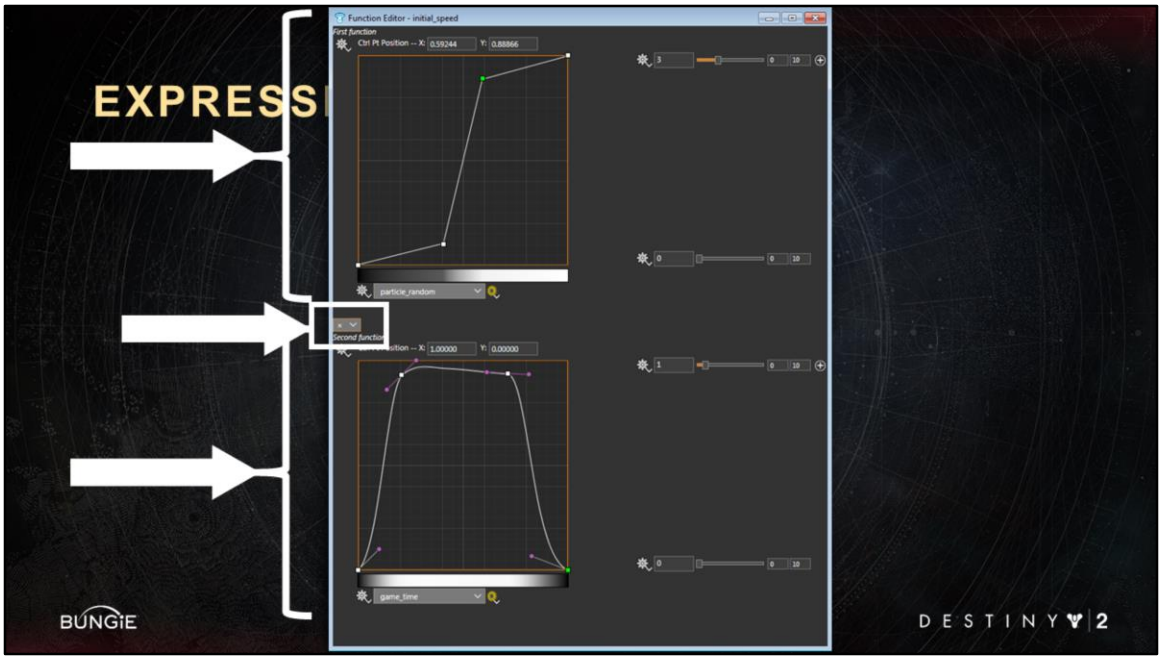
they can add different stops along a curve <Advance Slide> effectively doing a remap from my x axis to my y axis here.

they can add keyframes of course.

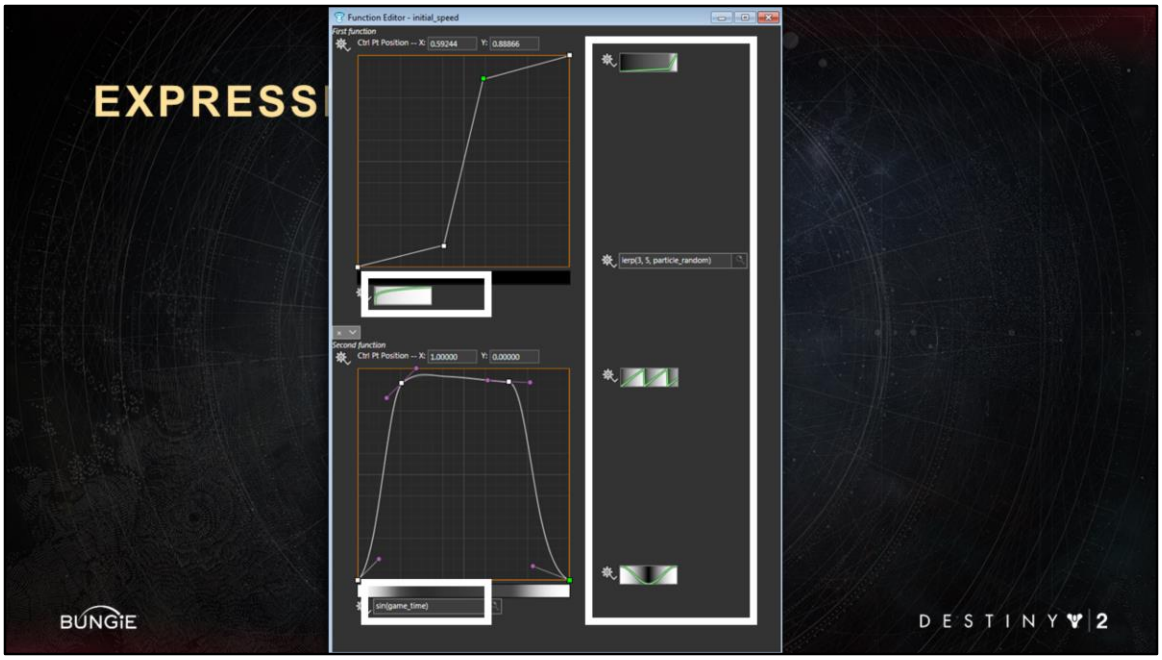
And change the min and max to be expressions as well



Change the domain to be an expression

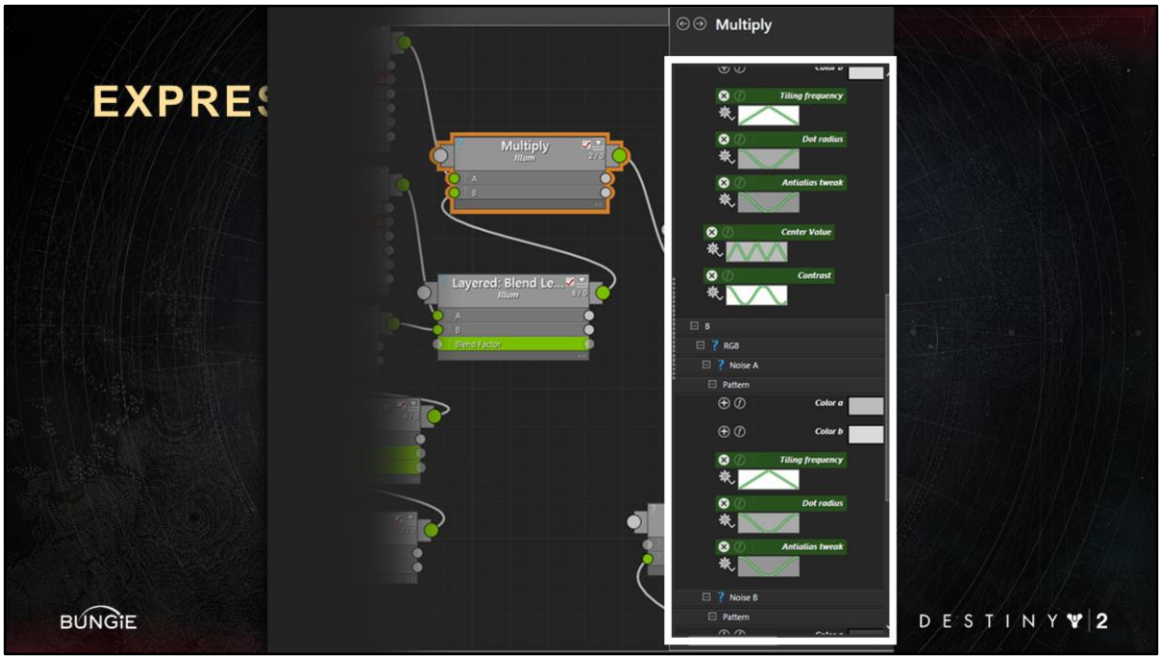


They can take any expression <Advance Slide>, and multiply it <Advance Slide> with a second expression <Advance Slide>.
Each expression can use different inputs.
They can keep on chaining expressions like this indefinitely, using add, multiply, subtract and divide operations.



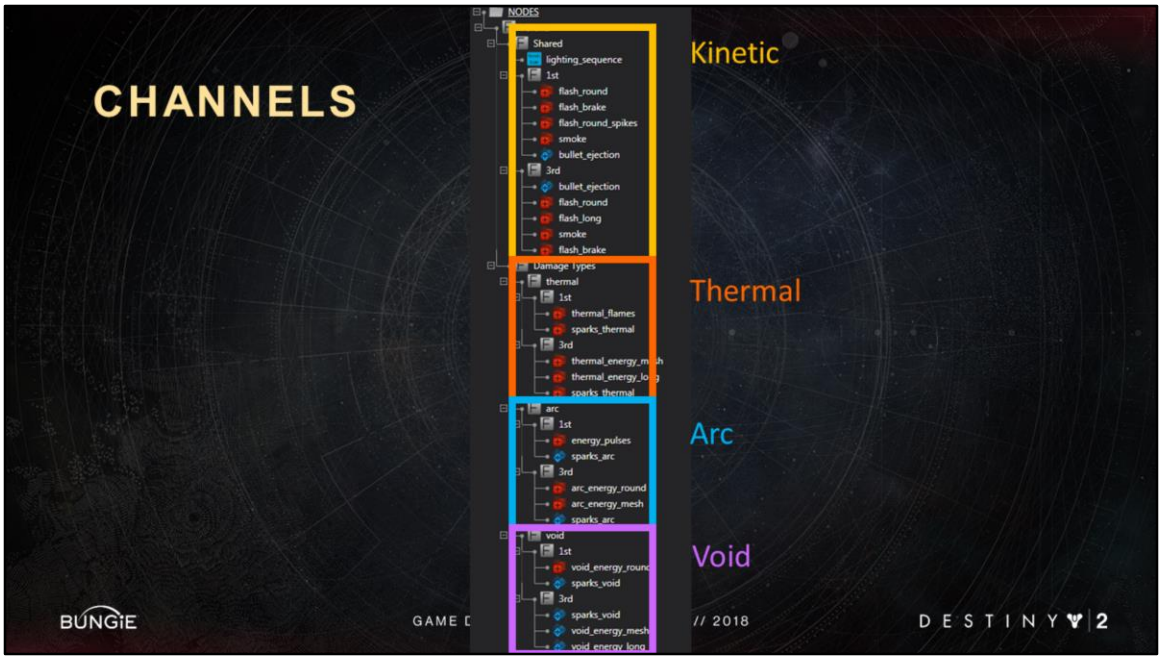
And I can still convert all the <Advance Slide> mins/maxes/domains into more expressions!
Each can have with different or many shader inputs like particle_age or particle_random.

Now keep in mind that this expression is just on a single parameter.

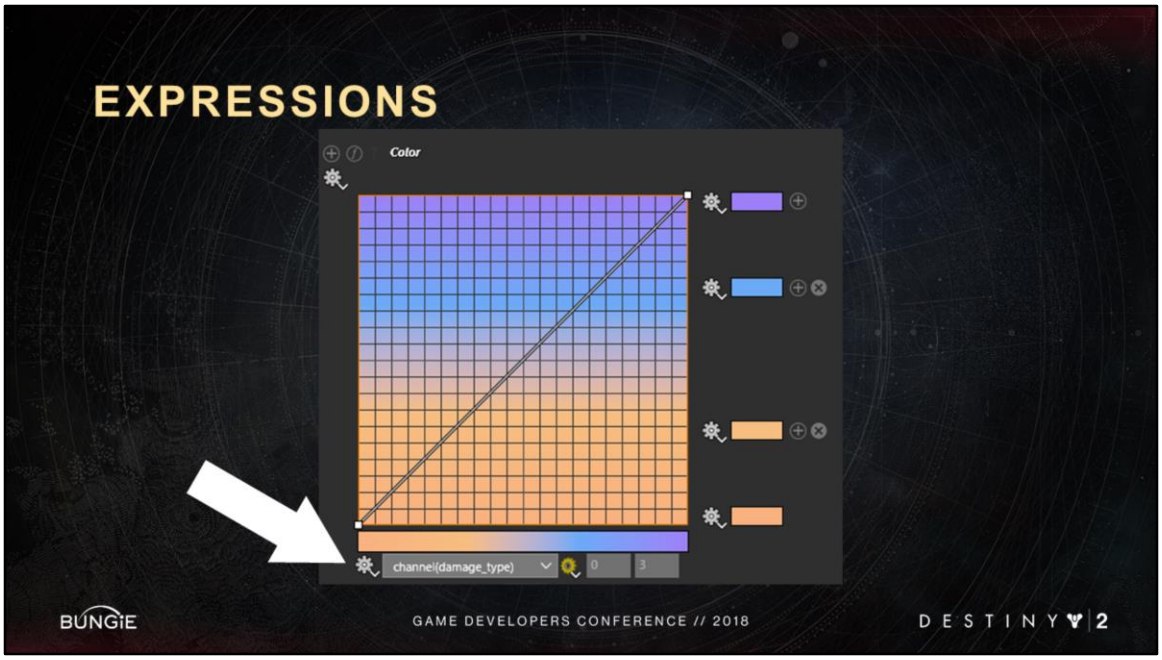


I can put these expressions <Advance Slide> on many or every shader node parameter. Across all shader types. At a whim.
And this gives us all the flexibility and expressive power.

This is an simple example. This is the arc flux grenade.
For explosion sparks we typically use a cone emitter, but that usually gives a distribution that is too uniform.
With a single expression we were able to randomize the emitter to give it a more natural chunky look.



Remember the damage type channel we used in muzzle flashes?
We used it in the sequencer as a condition.
Well we can use it inside the shaders as well!



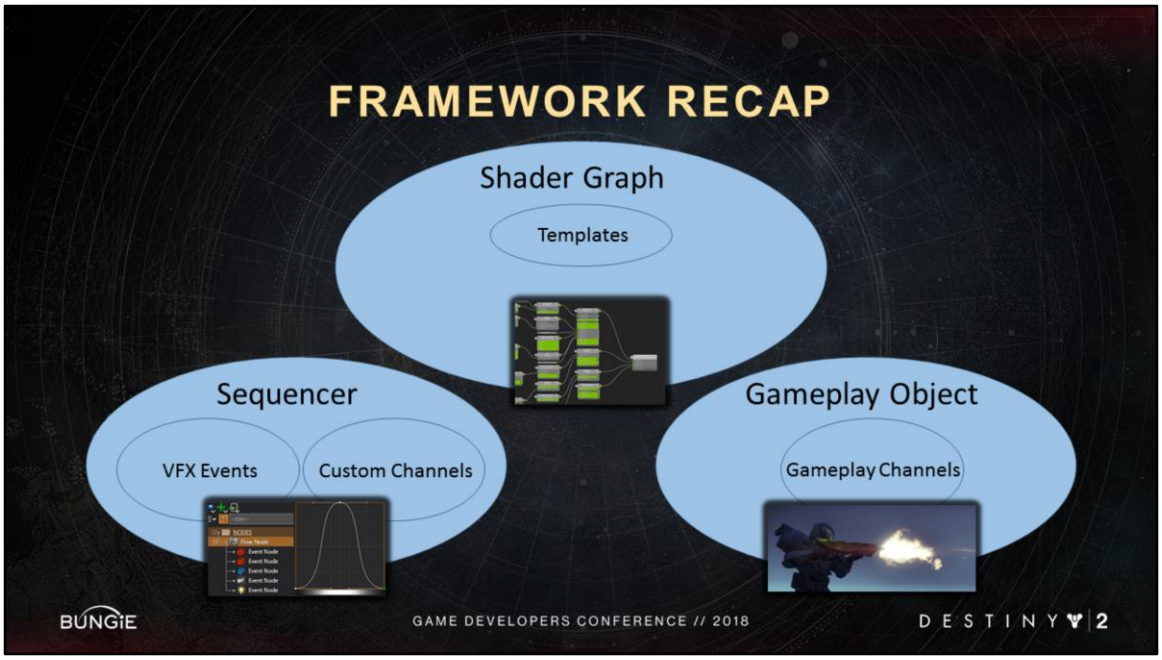
In the muzzle flash light shader I can use the dmg type channel <Advance Slide> to tint the light to match the damage color.

You can see the lighting bouncing off the characters here change color with the damage type.
So just like artists can type in any shader parameter input into their expression, they can also type any gameplay as well.

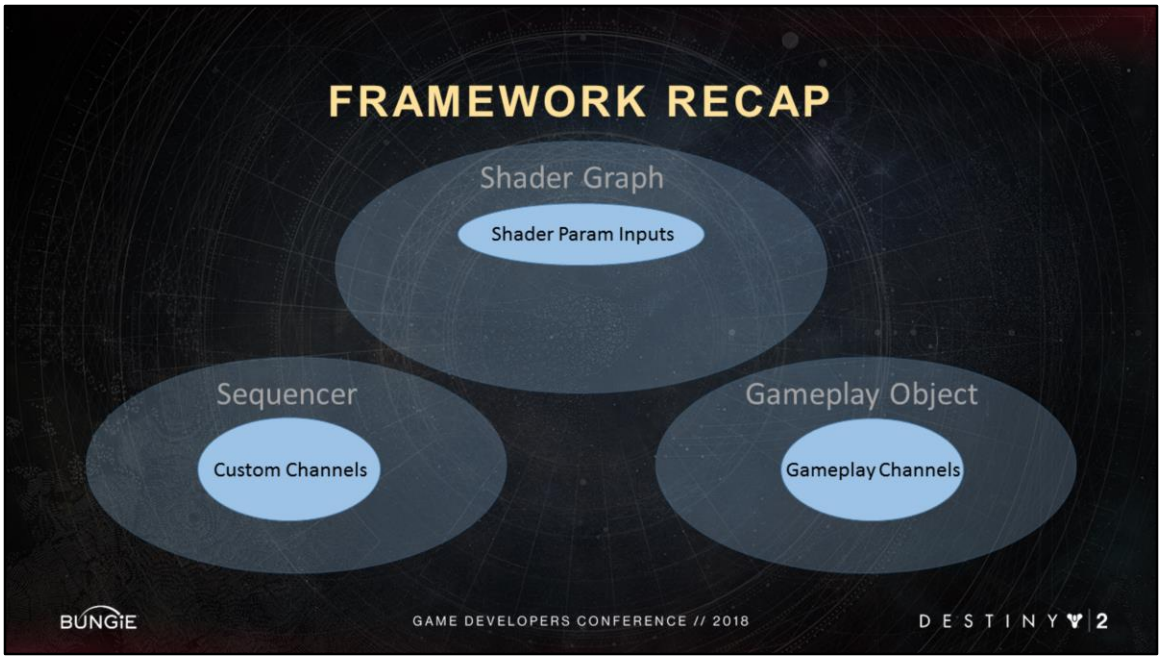
Not only that, but artists can just as easily type in any custom channels into an expression as well!
Like the vex boss charge channel.

Lots of expressive power here.

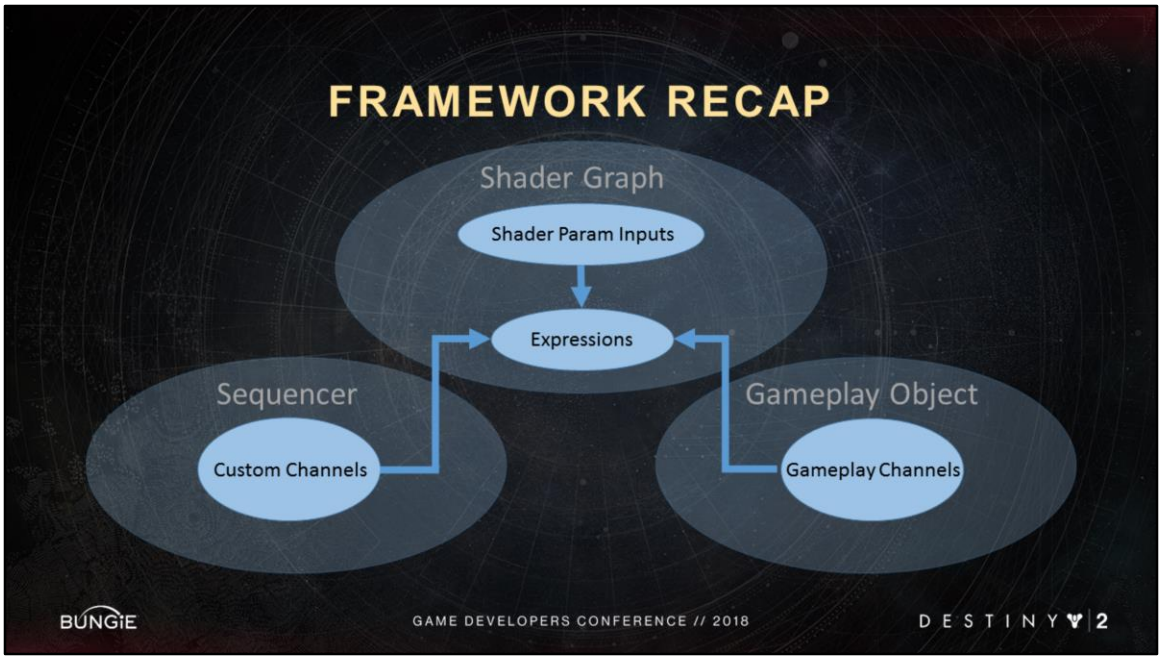
Brandon will also show more examples of expressions in general.



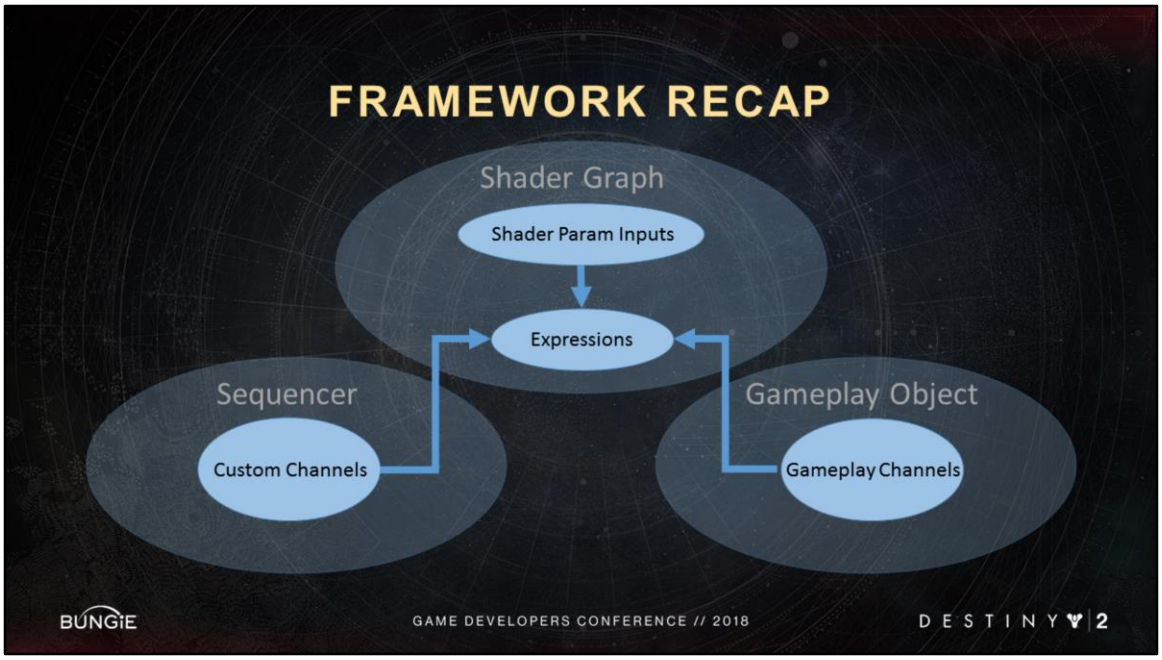
And with that we've unlocked the last part of the framework.



<Advance Slide> Sequences can have custom channels, like the vex boss charge intensity
<Advance Slide> Gameplay object have gameplay channels like damage type
<Advance Slide> Shader graph have shader parameter inputs like particle_age



And all these things can be used by expressions.
This right here is the heart of our shader and vfx tech.



Our ability to create complex expressions
at a whim
on any parameter
across particle appearance, emission and motion;
as well as **any** of our shader types

That, I think, is the secret sauce of our vfx.

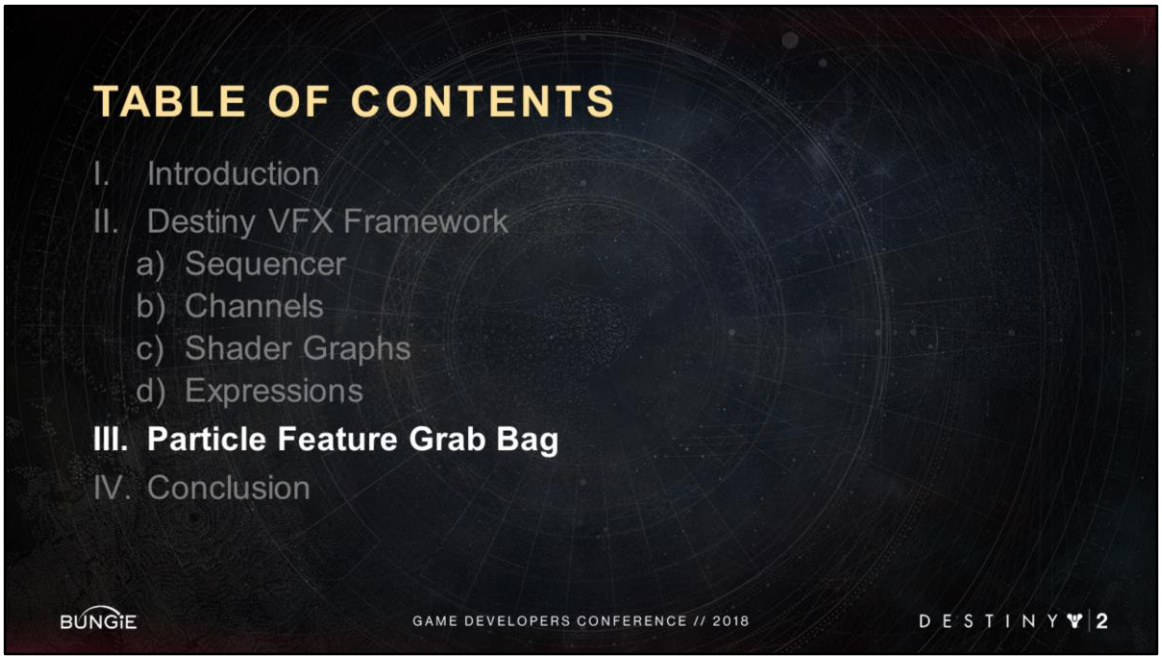
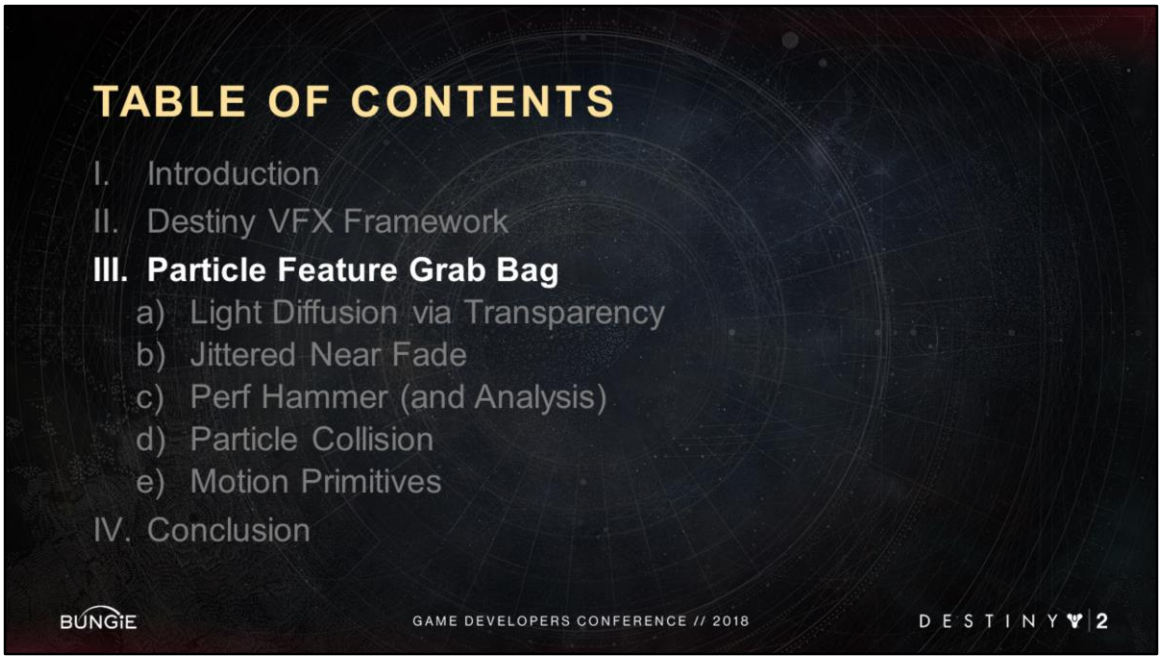


TABLE OF CONTENTS
I. Introduction
II. Destiny VFX Framework
a) Sequencer
b) Channels
c) Shader Graphs
d) Expressions
III. Particle Feature Grab Bag
IV. Conclusion
BUNGIE
GAME DEVELOPERS CONFERENCE // 2018
DESTINY 2


TABLE OF CONTENTS
I. Introduction
II. Destiny VFX Framework
III. Particle Feature Grab Bag
IV. Conclusion
BUNGIE
GAME DEVELOPERS CONFERENCE // 2018
DESTINY 2



Reminder of what this section is all about. Miscellaneous list of particle features. Do not require each other nor do they require the framework. We will call out, however, where they benefit from the framework.

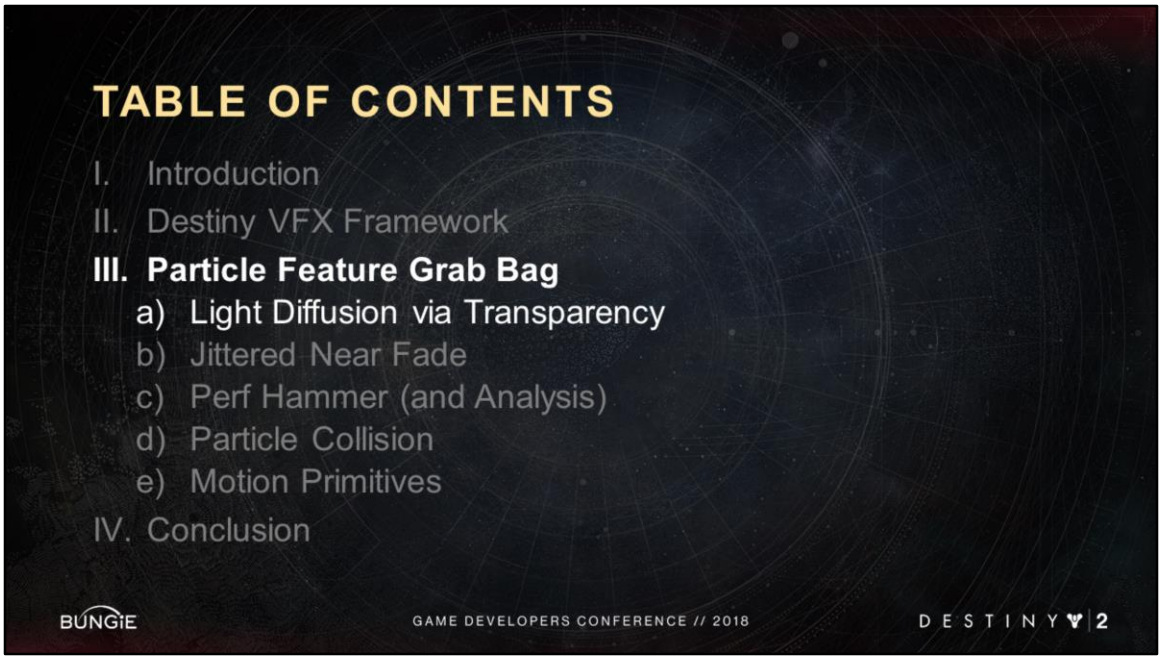
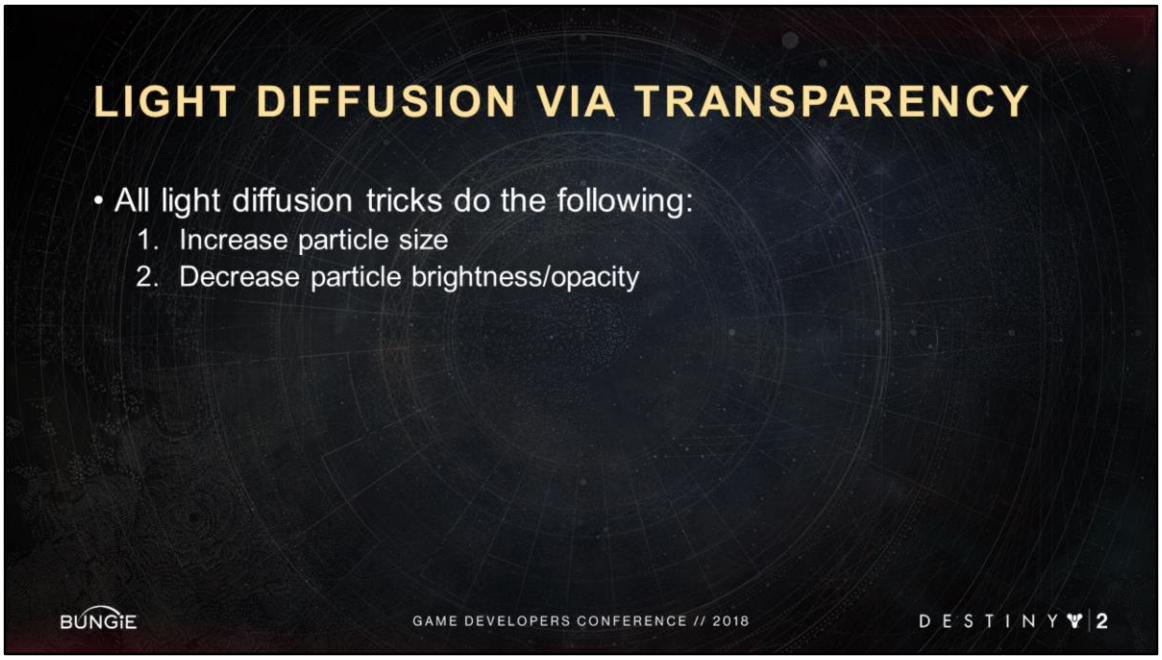


TABLE OF CONTENTS
I. Introduction
II. Destiny VFX Framework
III. Particle Feature Grab Bag
a) Light Diffusion via Transparency
b) Jittered Near Fade
c) Perf Hammer (and Analysis)
d) Particle Collision
e) Motion Primitives
IV. Conclusion
BUNGIE
GAME DEVELOPERS CONFERENCE // 2018
DESTINY 2

Can also call these “partial coverage” technique. Anti-aliasing in particular was critical to Destiny 2.

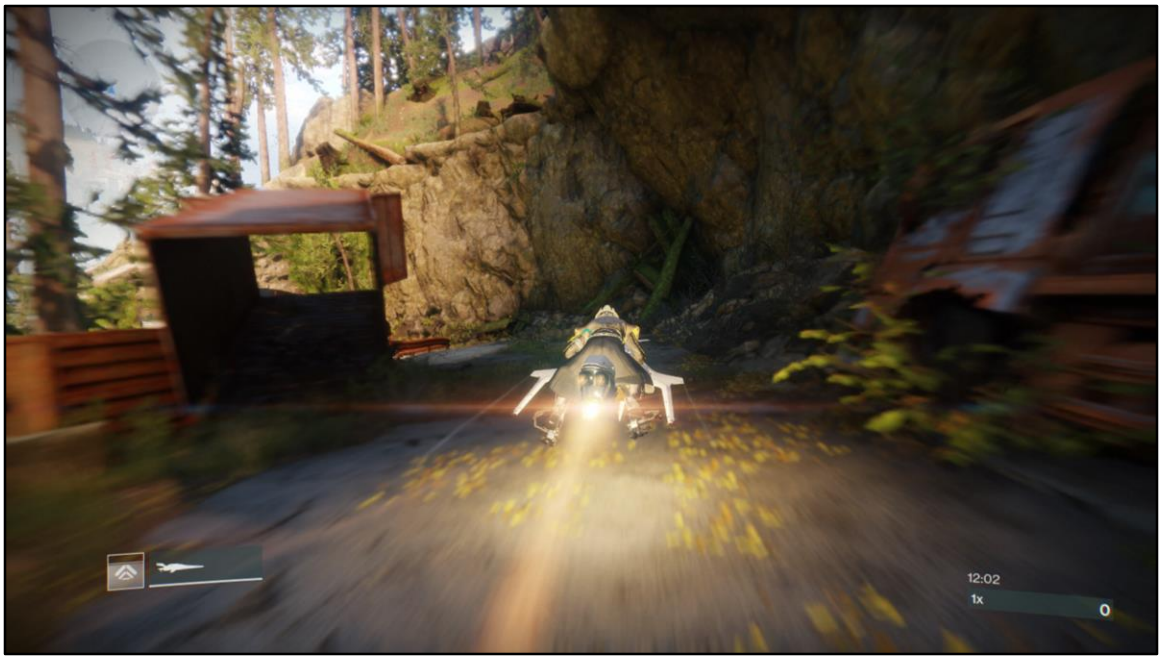


This should become clear as we go through the three examples.

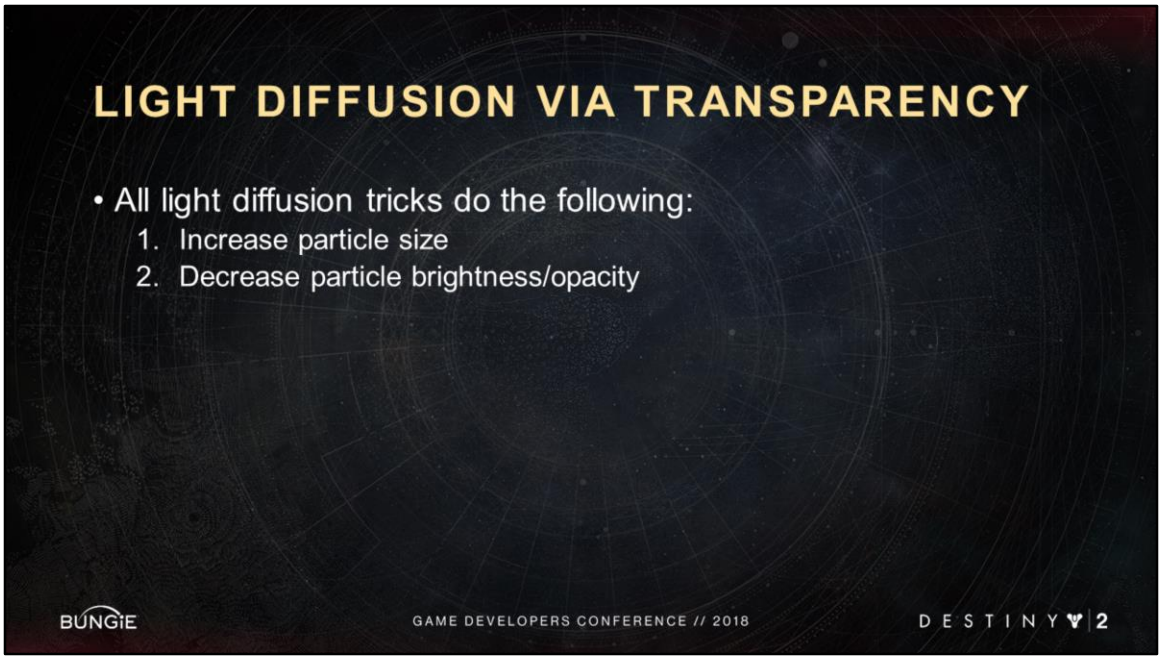
What is motion blur in this context? The diffusion of light over time. Over the length of the exposure of the camera. Light spreads in the direction of motion, taking on appearance larger than it actually is, while decreasing in apparent brightness relative to what it would be while stationary.

By E01 - originally posted to Flickr as London bus, CC BY-SA 2.0, https://commons.wikimedia.org/w/index.php?curid=4575184

Opacity of bullet casing is diminished in some places, because at those pixels the bullet casing was not present for the entirety of the exposure.



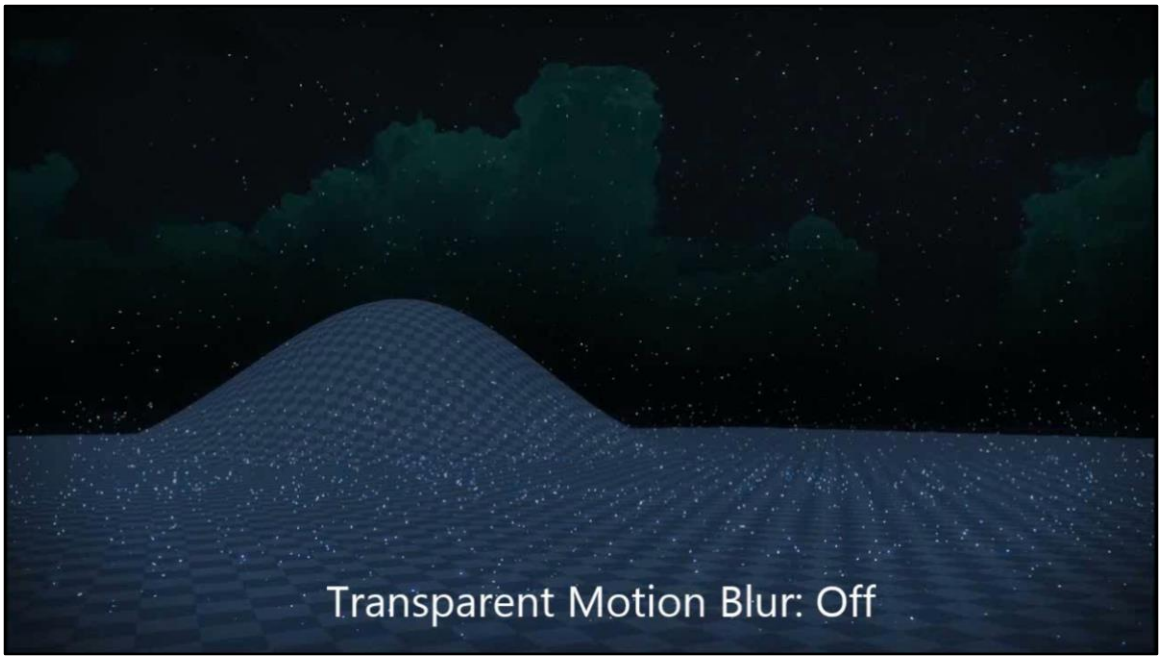
In games, often motion blur applies only to opaque objects. Might have a velocity buffer that stores the velocity at each pixel for the opaques.



Transparencies can stretch along the direction of the motion based on the particle speed, and decrease the brightness a corresponding amount.

This is an approximation, but can work well for small quad shaped particles.

Might call it a partial coverage problem with diffusion approximation.

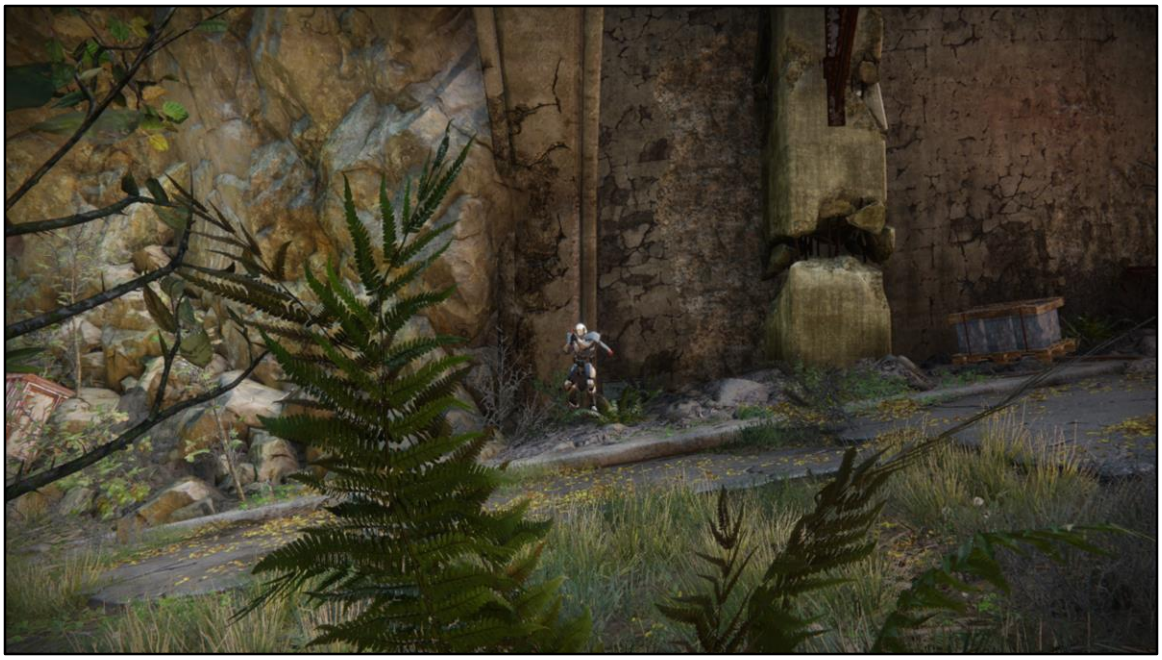


This is the fist of havoc impact effect, or sometimes called the titan smash. We'll be seeing this more throughout this section.

If it wasn’t clear in the last video, this one will hopefully make it more clear, especially when it pauses at the end. Also random dancing guy lol.

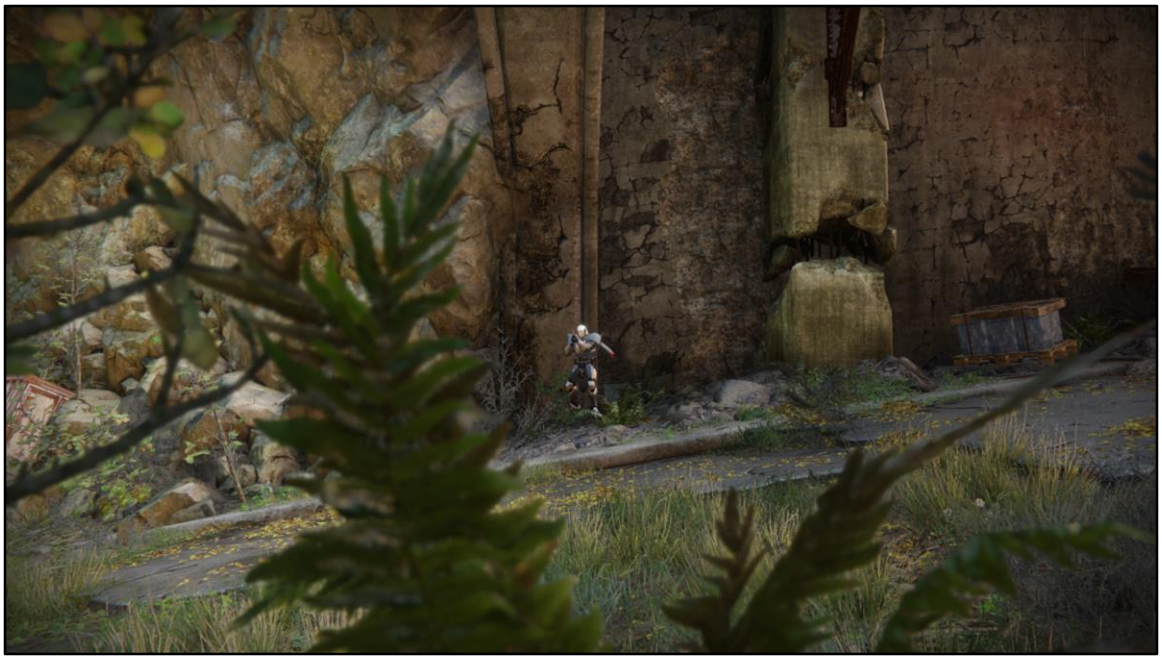
Again, it’s an approximation to real motion blur, but it works well and we used it everywhere.

In this context, depth of field is the diffusion of light through a lens.



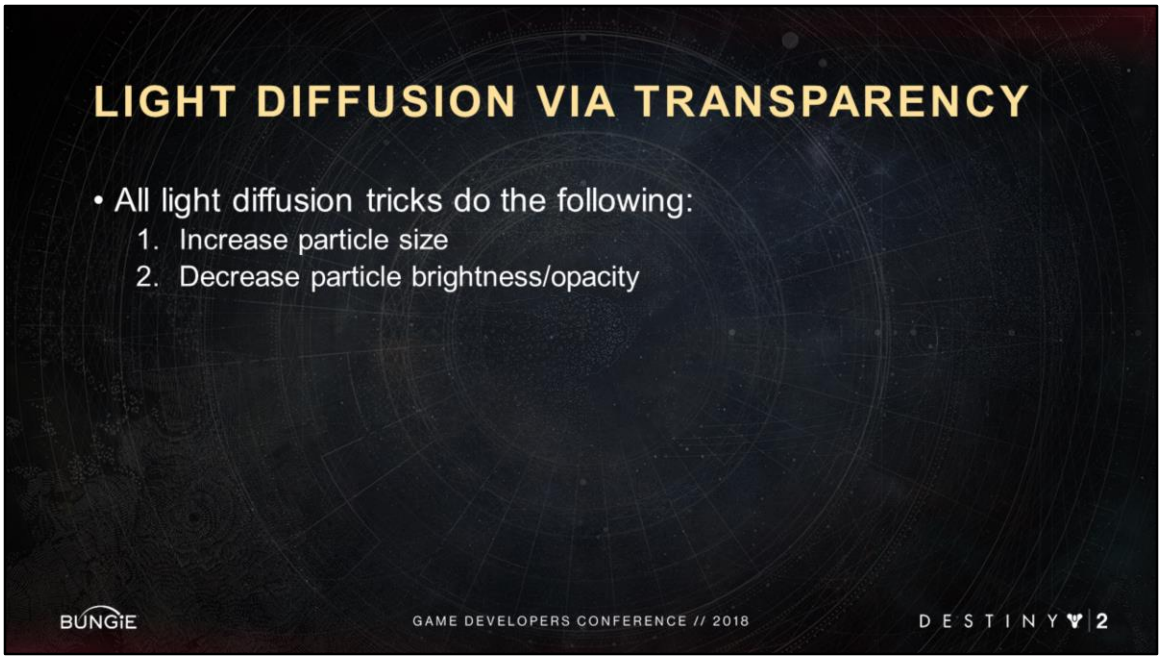
Again, in games this is usually applied to opaque objects. Depth buffer drives it.

For instance, look at this fern in the foreground.



We apply your standard screen space depth of field to it.

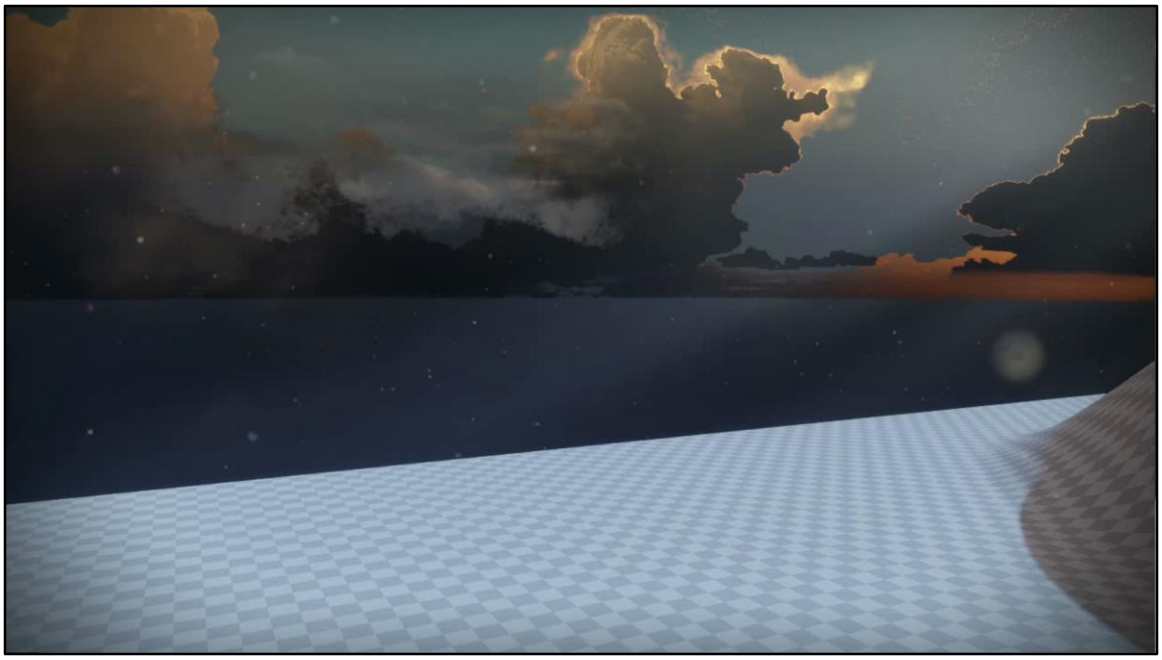
But now a solar grenade goes off in your face. Transparent sparks are sharp despite being at the same depth as the fern. They should be blurry.



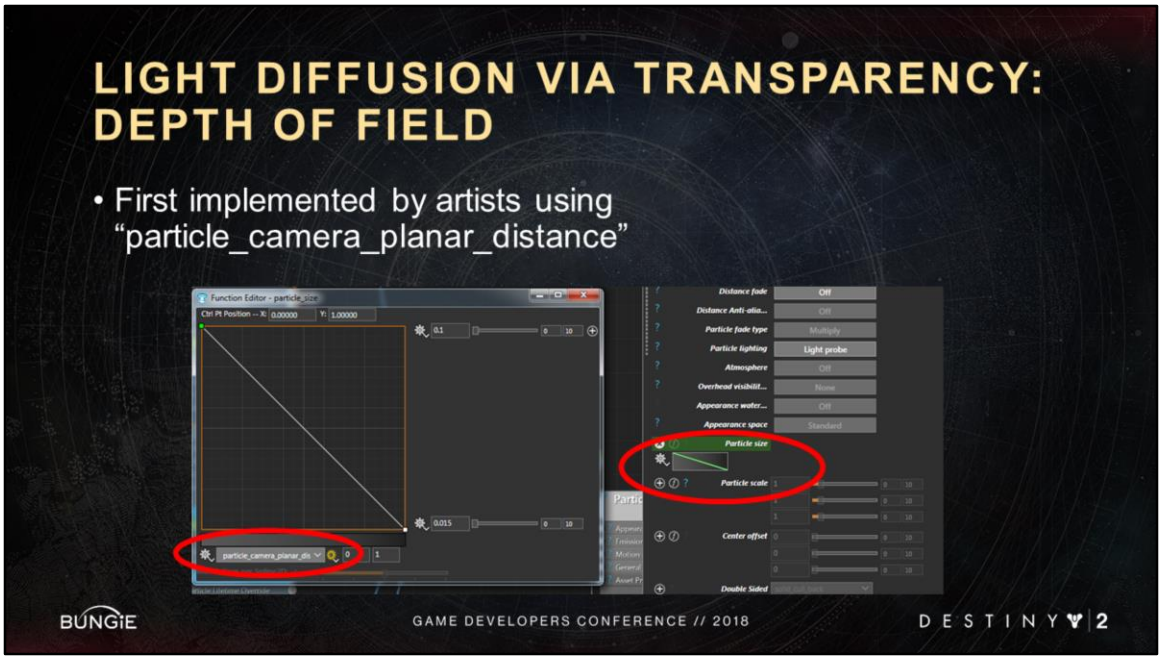
Apply the same idea as before. This time instead of based on the particle speed, it will be based on how in or out of focus the particle is (circle of confusion).

This is again an approximation that works well for simple geometry like a small particle quad.

Can also use a blurrier mip level for the texture, or blend to an entirely different texture, like bokeh style texture.

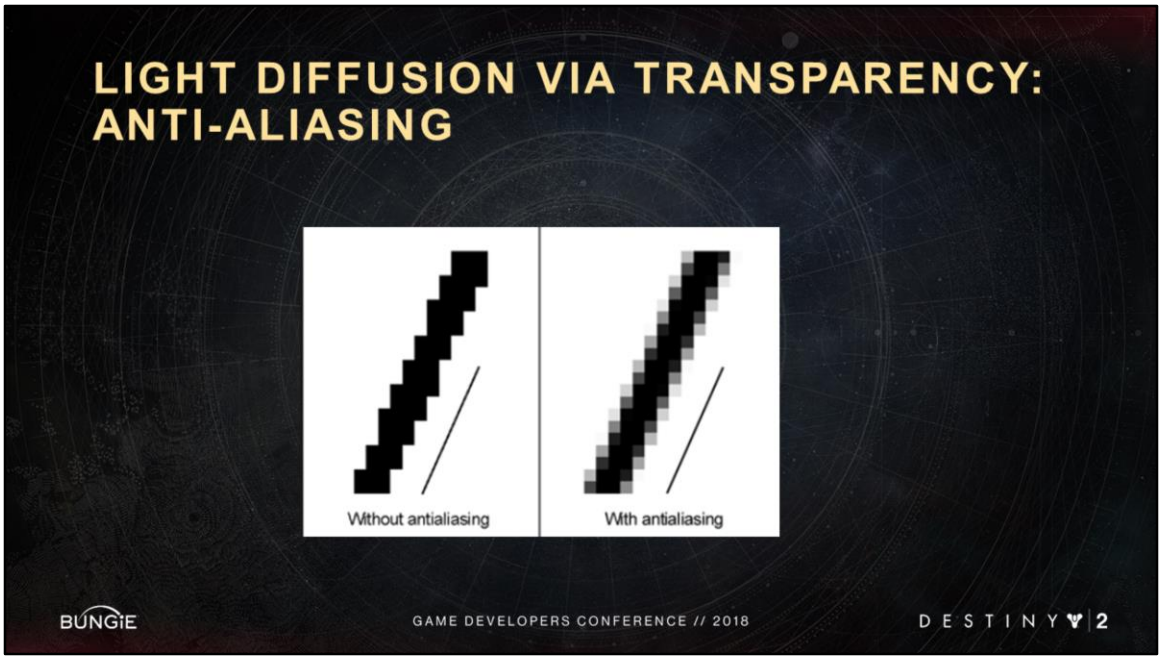


This is used in the Nessus destination in Destiny 2.

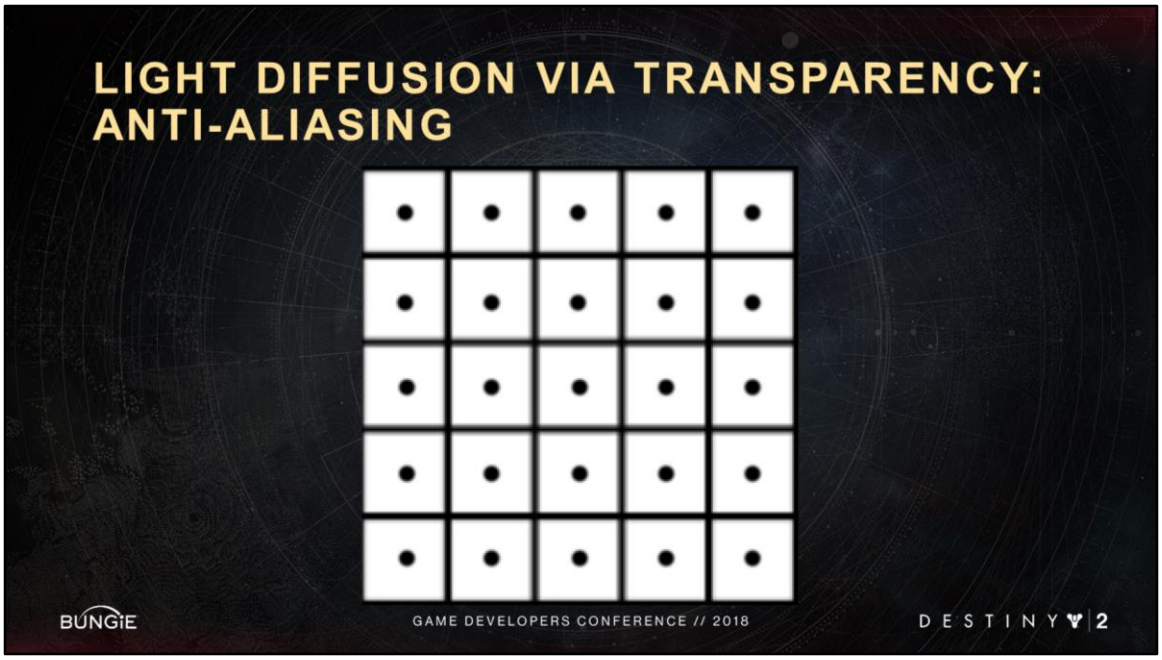


Expression parameter inputs enabled artists to prototype this on their own. Not the final solution we want, but a cool example of our framework empowering artists to experiment.

We intended to actually implement this fake depth of field in code, and give it a nice wrapper node and intuitive interface, but once we saw that an artist was already doing it, it lowered in priority so we never did.

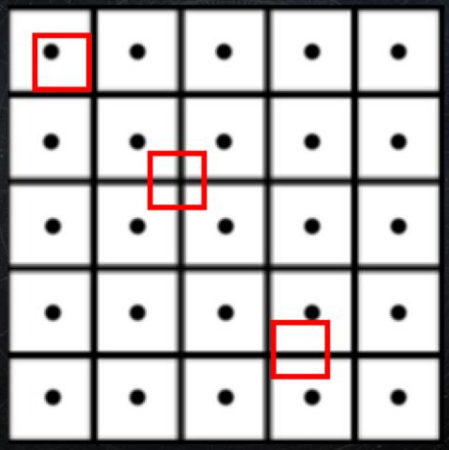


Again, this one ended up being the most critical of the three examples to Destiny 2.



We are interested in a particular case of aliasing, called undersampling. Here we have a quad that is smaller than the size of a pixel, and thus we have no guarantee that it will be covered by a pixel, and thus it seems to phase in and out of existence.

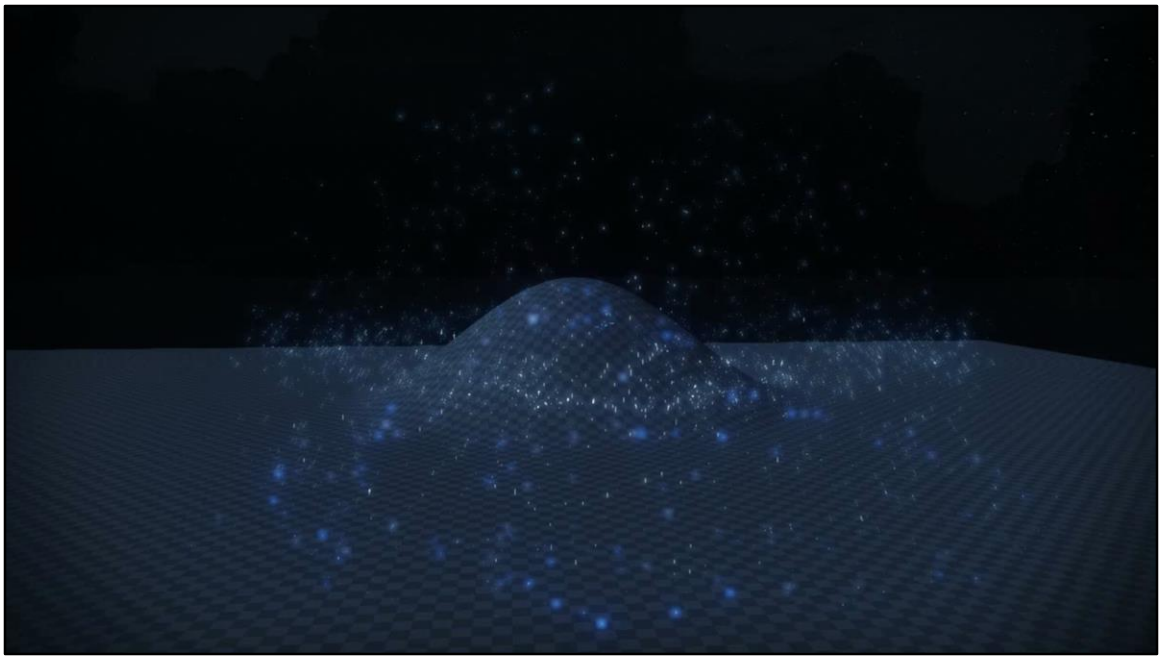
LIGHT DIFFUSION VIA TRANSPARENCY: ANTI-ALIASING



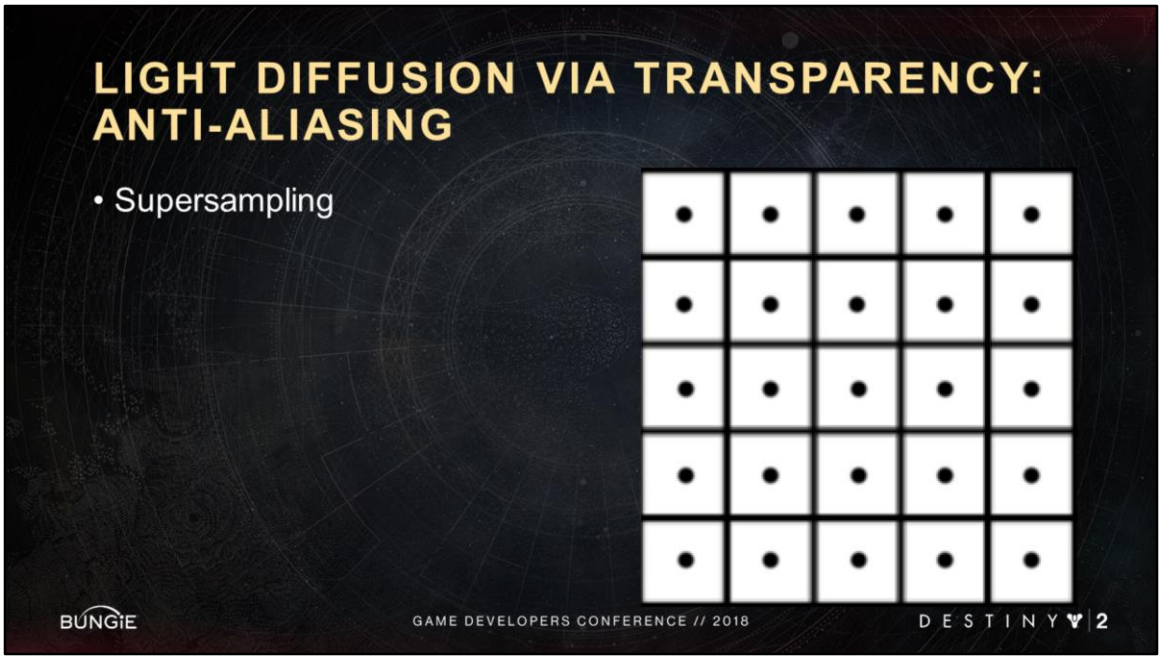
BUNGIE

GAME DEVELOPERS CONFERENCE // 2018

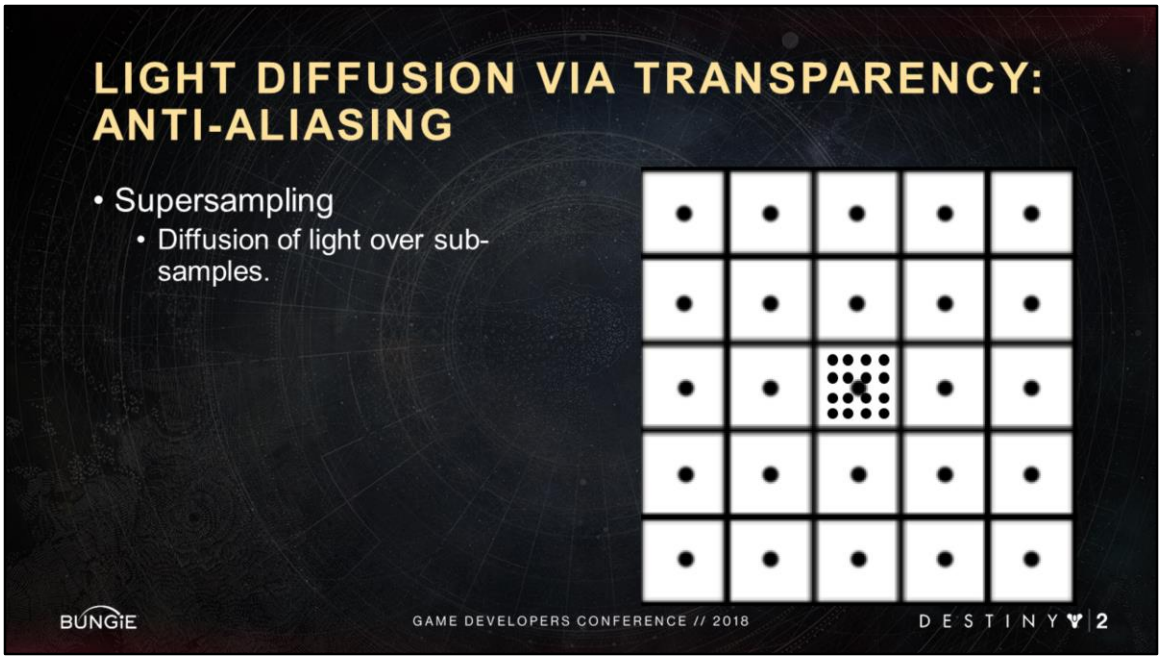
DESTINY 2



Here’s our titan smash impact effect again. If we don’t do anything to combat undersampling, we get this lovely mess.



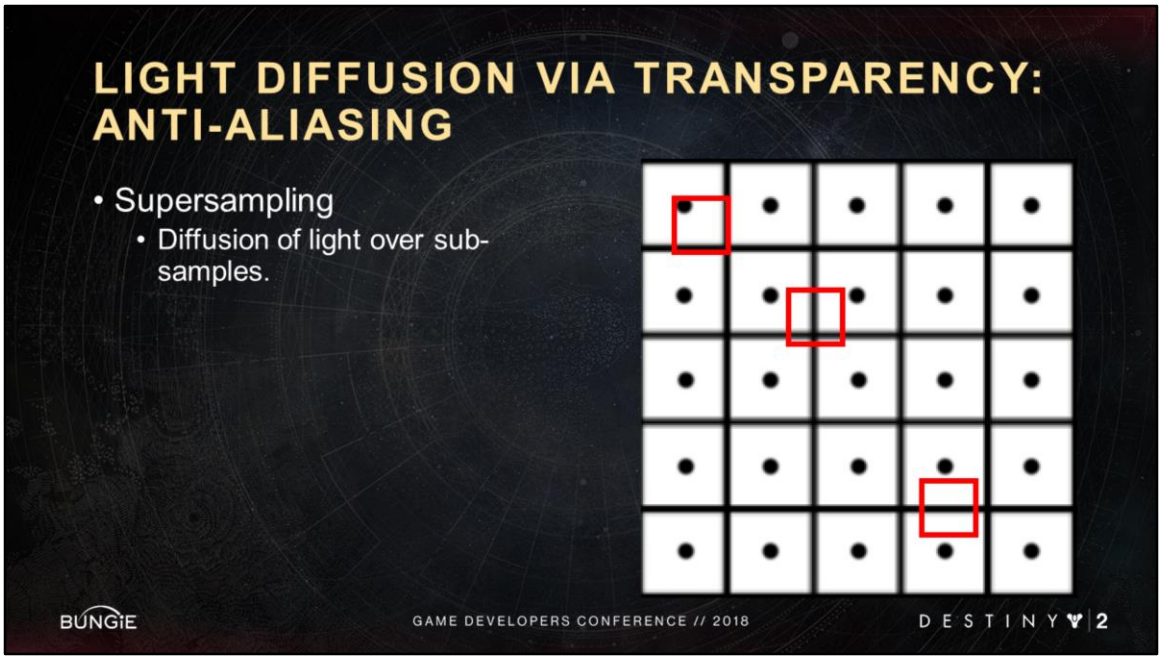
Super sampling is a way to combat this. Simply take many samples for each pixel, then average those samples. But of course that's expensive.



Again, we could call this a partial coverage problem.

LIGHT DIFFUSION VIA TRANSPARENCY

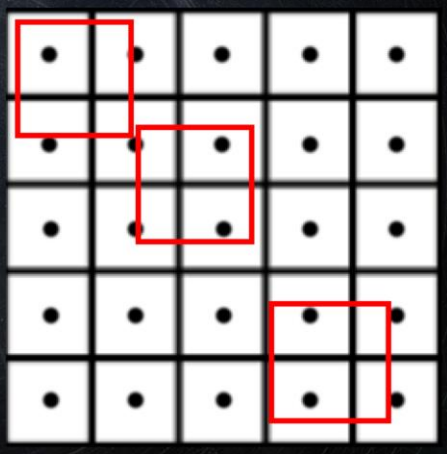
- All light diffusion tricks do the following:
  1. Increase particle size
  2. Decrease particle brightness/opacity

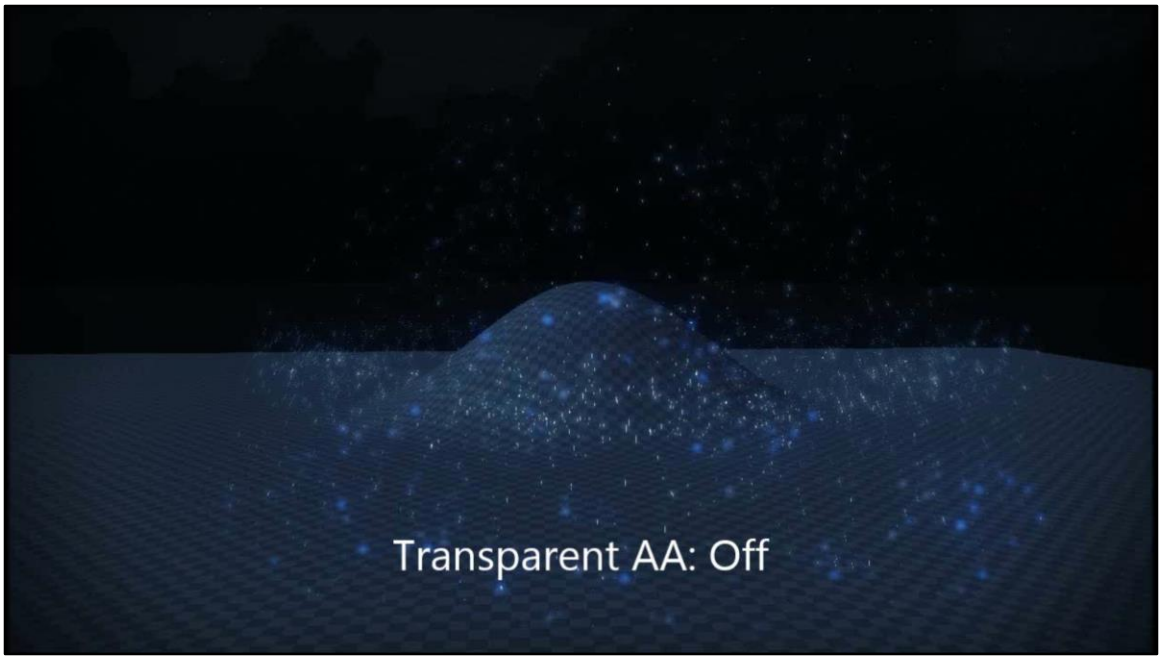


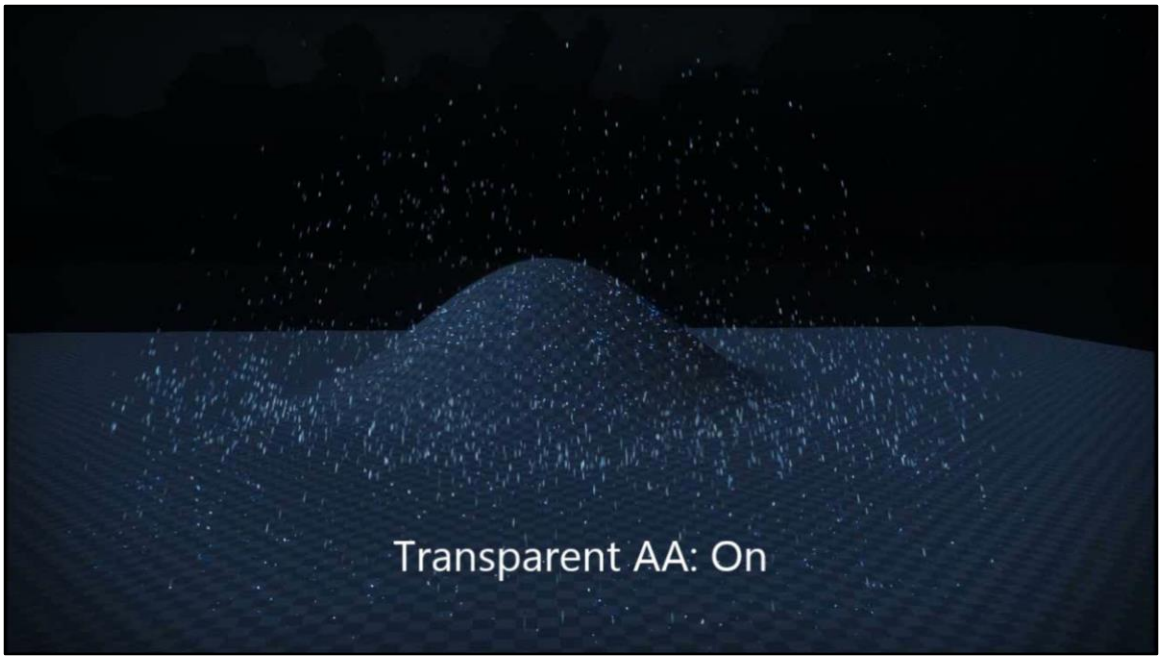
So for our translucent particle quad, we can do the same thing we did for motion blur and depth of field, which is to increase the size of the particle and at the same time decrease opacity, this time based on the screen coverage of the particle, which is calculated from its distance, size, and also the resolution of the frame.

LIGHT DIFFUSION VIA TRANSPARENCY: ANTI-ALIASING

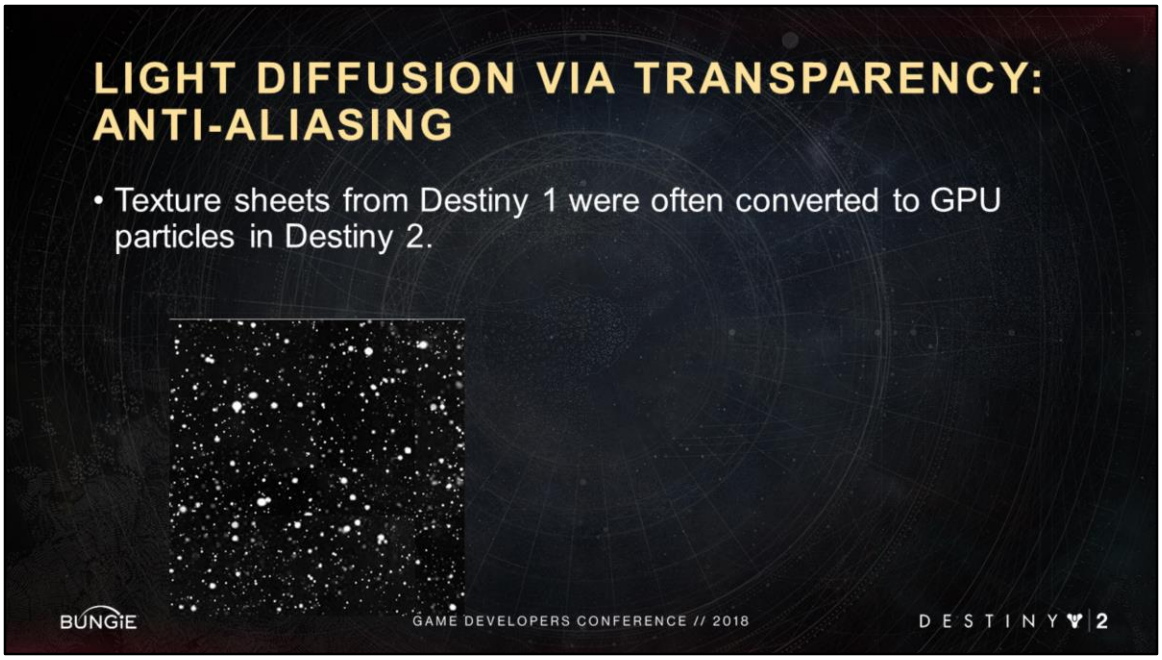
- Supersampling
  - Diffusion of light over sub-samples.




Transparent AA: Off



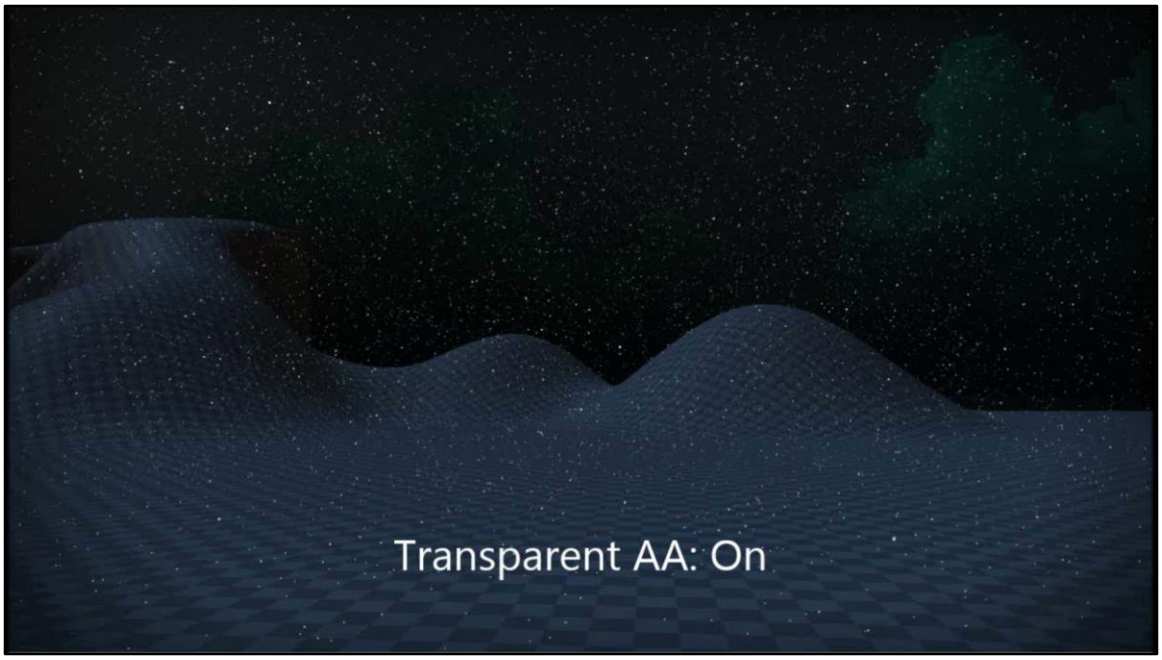
Works at extreme distances as well. As the camera distance increases, we keep increasing the size of the particle so that it covers a single pixel, while further decreasing the particle opacity.



One of the reasons this was super important was because we had a lot of effects that we previously would have done entirely with texture sheets or texture warps that were converted to use GPU particles. For instance this snow texture. But textures have the advantage of mip maps, which counter undersampling. So this anti-aliasing trick ended up being super valuable.

LIGHT DIFFUSION VIA TRANSPARENCY: ANTI-ALIASING

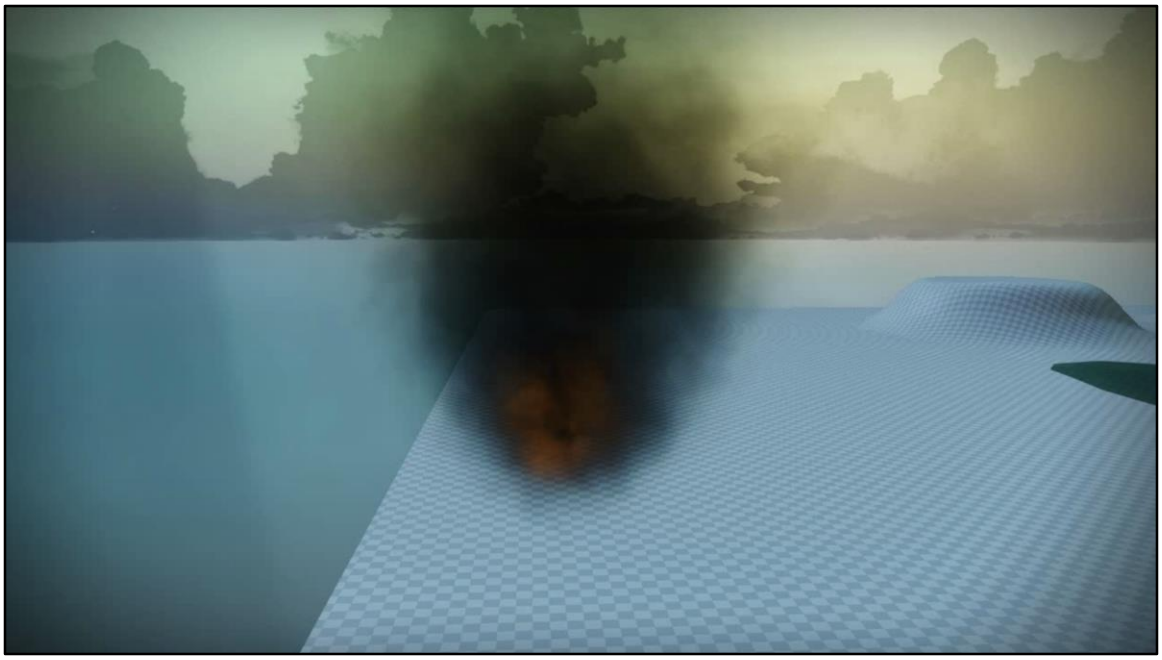
- Texture sheets from Destiny 1 were often converted to GPU particles in Destiny 2.
- Distant transparent anti-aliasing thus became crucial.



Speaking of snow, here’s a snow system with GPU particles.

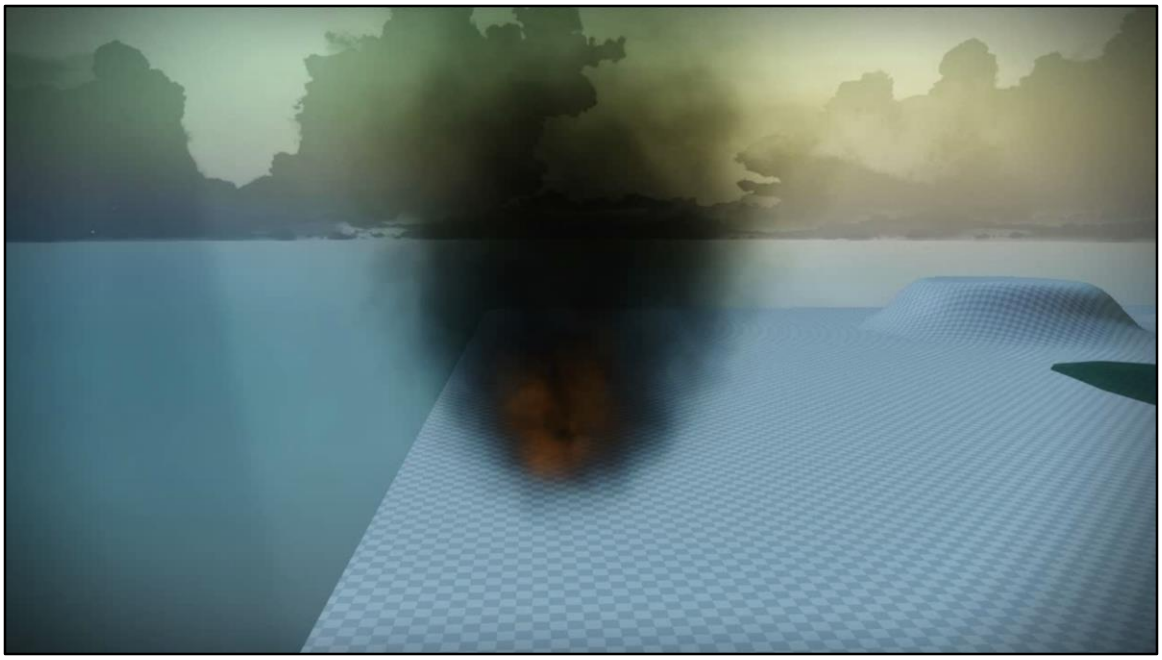
TABLE OF CONTENTS


PARTICLE APPEARANCE: JITTERED NEAR FADE
• Simple extension to the particle near fade.
BUNGIE
GAME DEVELOPERS CONFERENCE // 2018
DESTINY 2



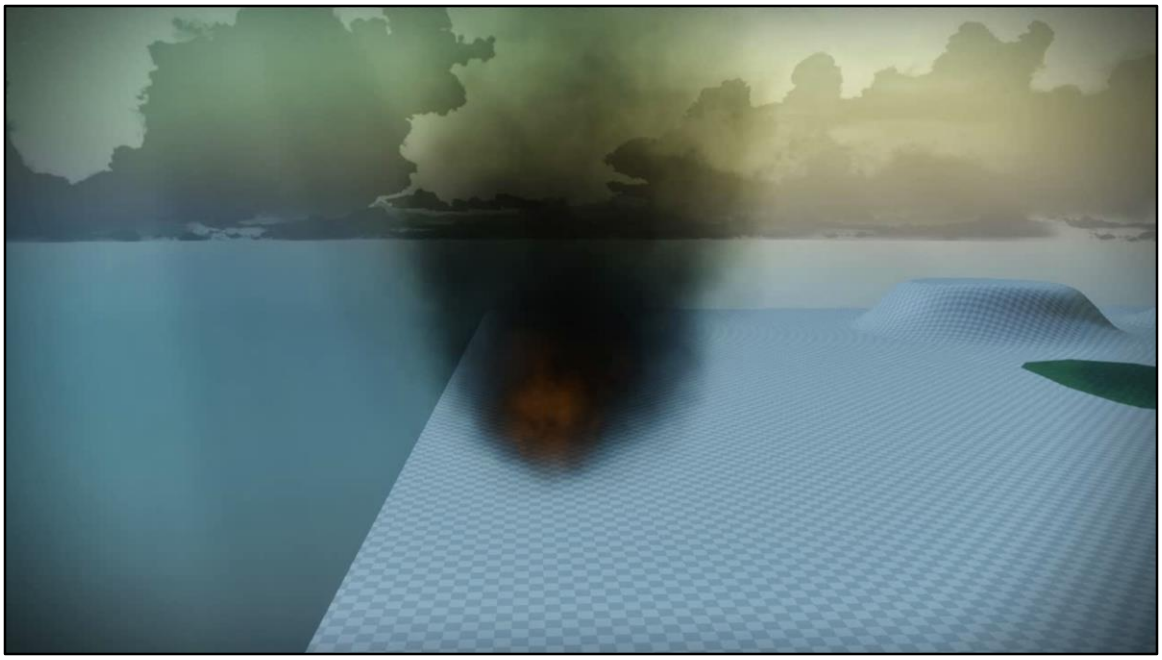
Here’s standard particle near fade. At a certain distance each particle starts to fade out, until it is completely transparent, and then thrown offscreen so we don’t pay any perf for it.

Sometimes we call the visual look the “fade wall”. You get this sense there’s this wall in front of you at all times, and as soon as particles hit the wall they quickly fade out.



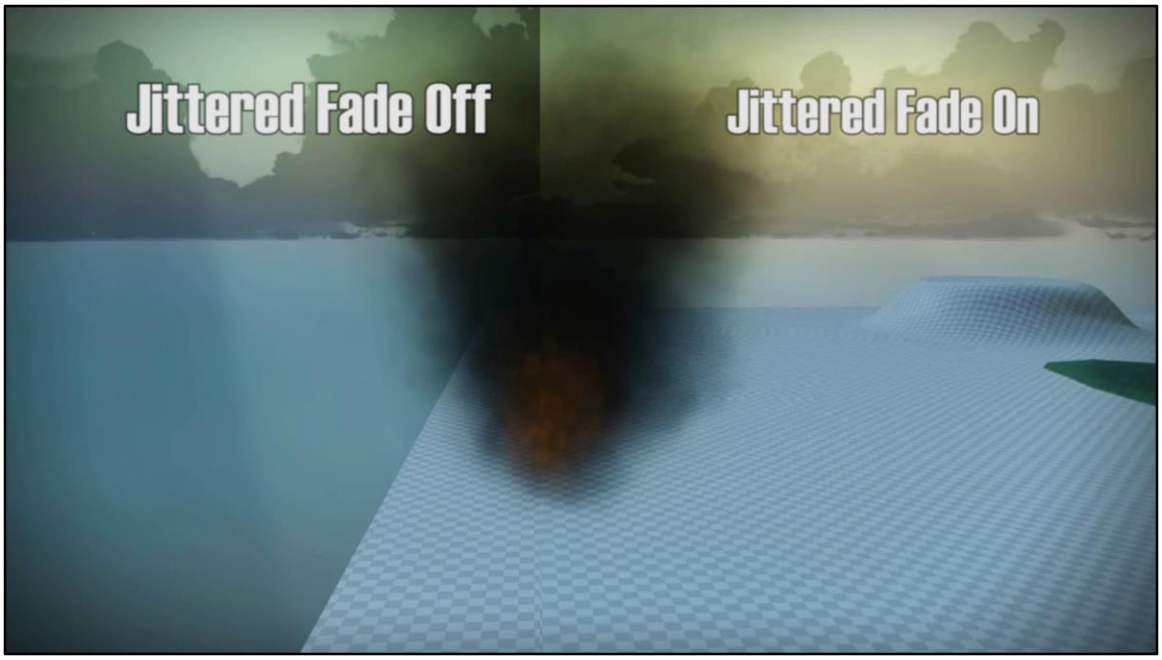
Here’s standard near fade again just to reiterate the inconsistent performance. Perf gets worse as you approach the system, until you the pass fade wall, and then it’s suddenly better. So that’s not great. We want to smooth out that performance.

Randomize the fade distances per particle just a bit. In other words, jitter them. We expect that to fix both our problems.



Here’s the jittered near fade. We’ll show a side-by-side video next to make it more clear, but it provides a more gradual fade of the particle. Sort of like the system is gradually thinning out as you approach it.

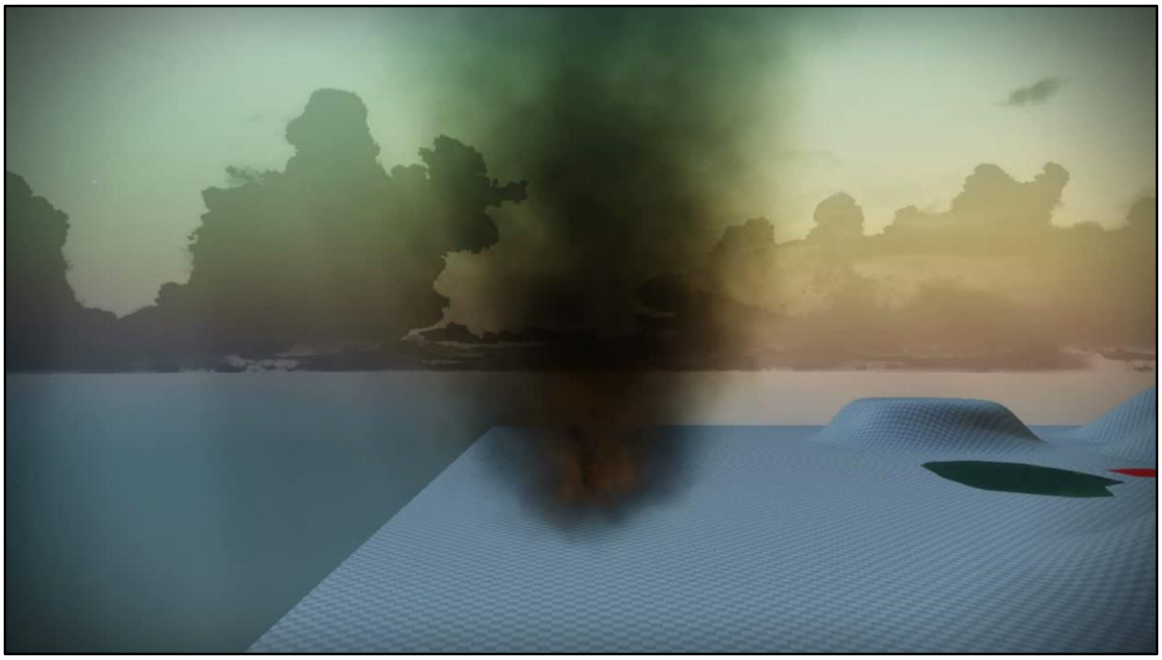
Gets rid of the noticeable “fade wall” issue, and yields more consistent perf because as you approach the system, and the screen coverage increases, we are more gradually compensating by using fewer and fewer particles.


Jittered Fade Off
Jittered Fade On

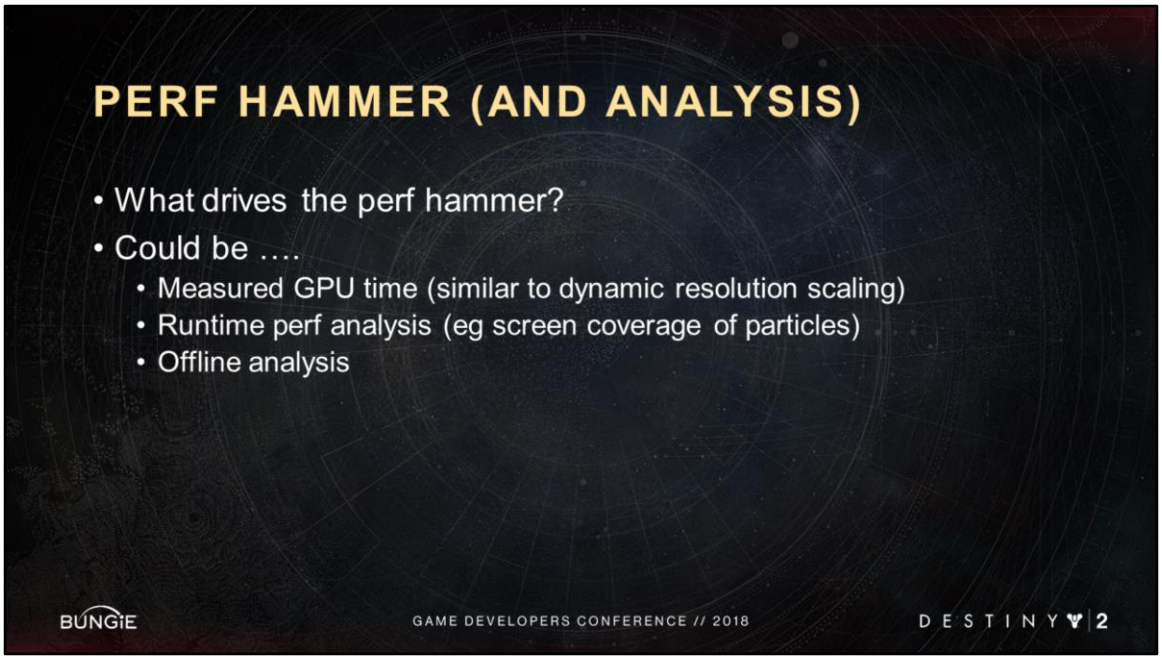
PARTICLE APPEARANCE: JITTERED NEAR FADE

- Implementation: Generate canonical pseudo-random number (0->1) per particle, based on particle ID.
- Easily templated and tuned across all our content.
  - Calibrating a few templates optimized perf across entire game.

TABLE OF CONTENTS



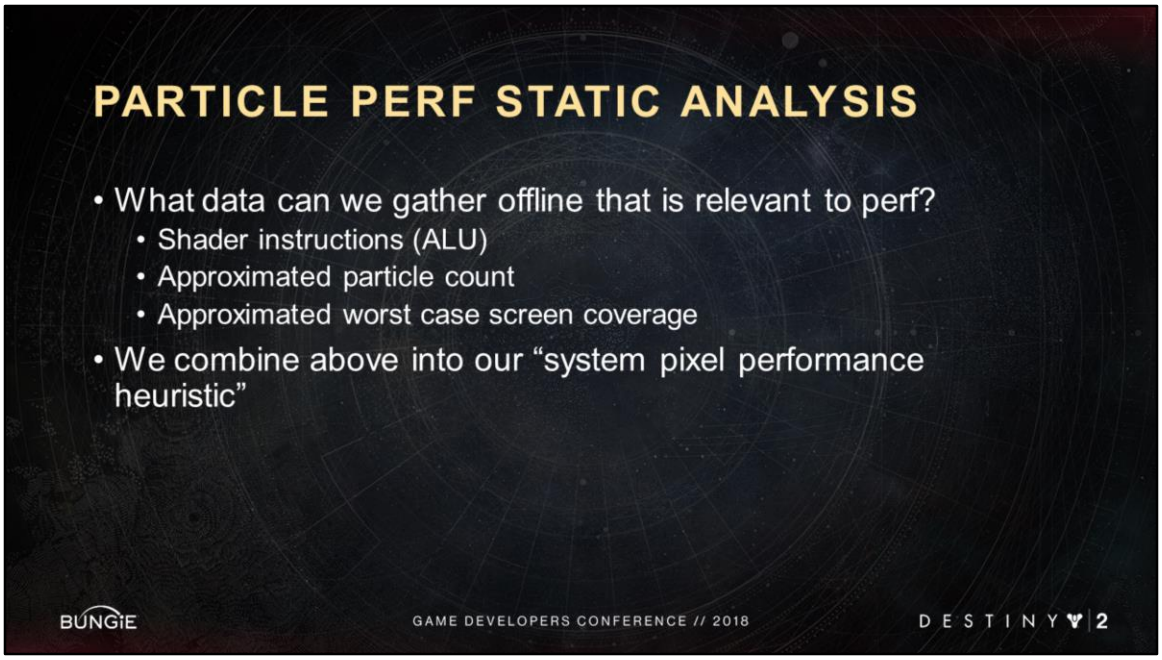
To explain what the perf hammer is, first let's go back to the jittered near fade again. We realized that with jittered near fade we had built the ability to thin out a particle system at will. For instance, we could thin out a particle system not based on distance to camera, but instead based on current GPU performance. And this is what we call our perf hammer. Idea is to thin out particle systems when the perf is bad, and gradually let them fill back in when the perf is acceptable again.



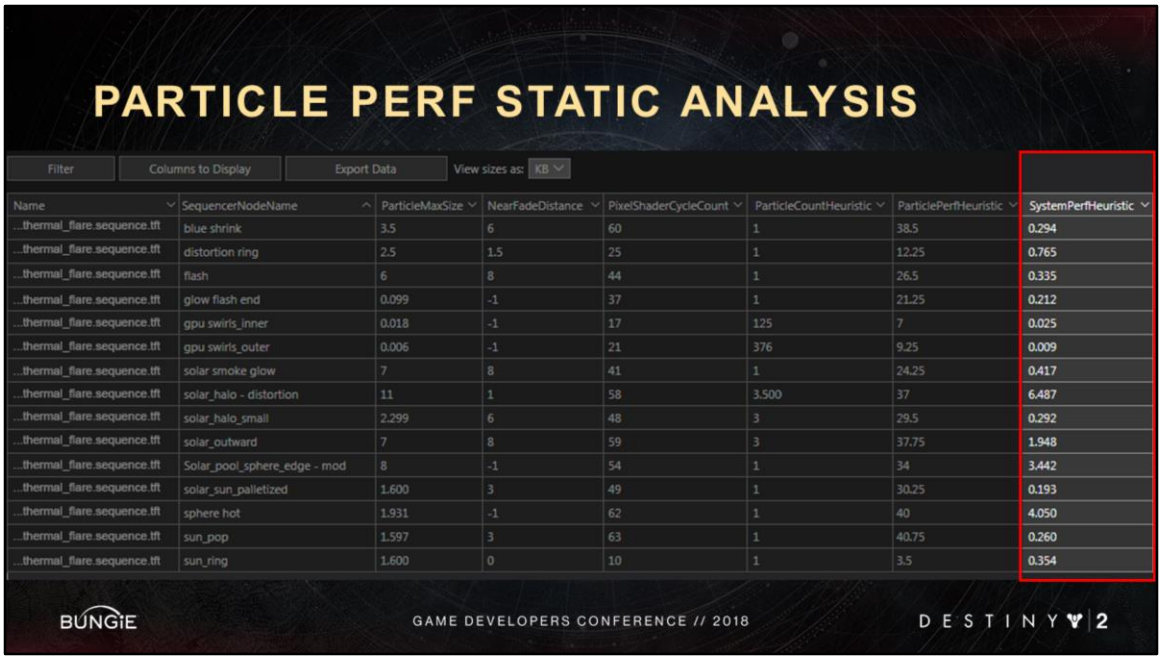
How to drive the perf hammer? Ie how to measure the current GPU performance.

Offline analysis could be used for a number of different things, and I'll talk it about it more on the next slide, but the general idea is let's figure out which particle systems are expected to be the most expensive, and then at runtime we can limit our search to only those systems and do a further runtime analysis of, for instance, screen coverage of the particles.

We end up using a combination of the last two.



We build an offline report with this information.



Name	SequencerNodeName	ParticleMaxSize	NearFadeDistance	PixelShaderCycleCount	ParticleCountHeuristic	ParticlePerfHeuristic	SystemPerfHeuristic
...thermal_flare.sequence.tft	blue shrink	3.5	6	60	1	38.5	0.294
...thermal_flare.sequence.tft	distortion ring	2.5	1.5	25	1	12.25	0.765
...thermal_flare.sequence.tft	flash	6	8	44	1	26.5	0.335
...thermal_flare.sequence.tft	glow flash end	0.099	-1	37	1	21.25	0.212
...thermal_flare.sequence.tft	gpu swirls_inner	0.018	-1	17	125	7	0.025
...thermal_flare.sequence.tft	gpu swirls_outer	0.006	-1	21	376	9.25	0.009
...thermal_flare.sequence.tft	solar smoke glow	7	8	41	1	24.25	0.417
...thermal_flare.sequence.tft	solar_halo - distortion	11	1	58	3.500	37	6.487
...thermal_flare.sequence.tft	solar_halo_small	2.299	6	48	3	29.5	0.292
...thermal_flare.sequence.tft	solar_outward	7	8	59	3	37.75	1.948
...thermal_flare.sequence.tft	Solar_pool_sphere_edge - mod	8	-1	54	1	34	3.442
...thermal_flare.sequence.tft	solar_sun_palletized	1.600	3	49	1	30.25	0.193
...thermal_flare.sequence.tft	sphere hot	1.931	-1	62	1	40	4.050
...thermal_flare.sequence.tft	sun_pop	1.597	3	63	1	40.75	0.260
...thermal_flare.sequence.tft	sun_ring	1.600	0	10	1	3.5	0.354

This number on the right is the heuristic I just mentioned.

PARTICLE PERF STATIC ANALYSIS

Filter | Columns to Display | Export Data | View sizes as: KB

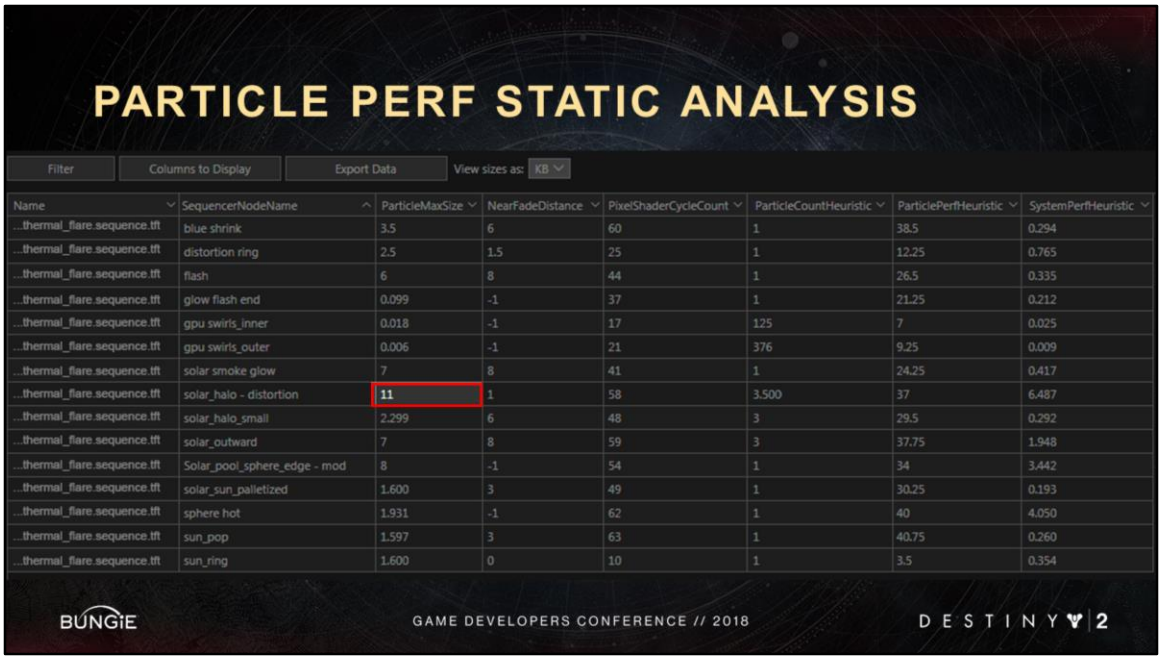
Name	SequencerNodeName	ParticleMaxSize	NearFadeDistance	PixelShaderCycleCount	ParticleCountHeuristic	ParticlePerfHeuristic	SystemPerfHeuristic
...thermal_flare.sequence.tft	blue shrink	3.5	6	60	1	38.5	0.294
...thermal_flare.sequence.tft	distortion ring	2.5	1.5	25	1	12.25	0.765
...thermal_flare.sequence.tft	flash	6	8	44	1	26.5	0.335
...thermal_flare.sequence.tft	glow flash end	0.099	-1	37	1	21.25	0.212
...thermal_flare.sequence.tft	gpu swirls_inner	0.018	-1	17	125	7	0.025
...thermal_flare.sequence.tft	gpu swirls_outer	0.006	-1	21	376	9.25	0.009
...thermal_flare.sequence.tft	solar smoke glow	7	8	41	1	24.25	0.417
...thermal_flare.sequence.tft	solar_halo - distortion	11	1	58	3.500	37	6.487
...thermal_flare.sequence.tft	solar_halo_small	2.299	6	48	3	29.5	0.292
...thermal_flare.sequence.tft	solar_outward	7	8	59	3	37.75	1.948
...thermal_flare.sequence.tft	Solar_pool_sphere_edge - mod	8	-1	54	1	34	3.442
...thermal_flare.sequence.tft	solar_sun_palletized	1.600	3	49	1	30.25	0.193
...thermal_flare.sequence.tft	sphere hot	1.931	-1	62	1	40	4.050
...thermal_flare.sequence.tft	sun_pop	1.597	3	63	1	40.75	0.260
...thermal_flare.sequence.tft	sun_ring	1.600	0	10	1	3.5	0.354

PARTICLE PERF STATIC ANALYSIS

Filter | Columns to Display | Export Data | View sizes as: KB

Name	SequencerNodeName	ParticleMaxSize	NearFadeDistance	PixelShaderCycleCount	ParticleCountHeuristic	ParticlePerfHeuristic	SystemPerfHeuristic
...thermal_flare.sequence.tft	blue shrink	3.5	6	60	1	38.5	0.294
...thermal_flare.sequence.tft	distortion ring	2.5	1.5	25	1	12.25	0.765
...thermal_flare.sequence.tft	flash	6	8	44	1	26.5	0.335
...thermal_flare.sequence.tft	glow flash end	0.099	-1	37	1	21.25	0.212
...thermal_flare.sequence.tft	gpu swirls_inner	0.018	-1	17	125	7	0.025
...thermal_flare.sequence.tft	gpu swirls_outer	0.006	-1	21	376	9.25	0.009
...thermal_flare.sequence.tft	solar smoke glow	7	8	41	1	24.25	0.417
...thermal_flare.sequence.tft	**solar_halo - distortion**	**11**	**1**	**58**	**3.500**	**37**	**6.487**
...thermal_flare.sequence.tft	solar_halo_small	2.299	6	48	3	29.5	0.292
...thermal_flare.sequence.tft	solar_outward	7	8	59	3	37.75	1.948
...thermal_flare.sequence.tft	Solar_pool_sphere_edge - mod	8	-1	54	1	34	3.442
...thermal_flare.sequence.tft	solar_sun_palletized	1.600	3	49	1	30.25	0.193
...thermal_flare.sequence.tft	sphere hot	1.931	-1	62	1	40	4.050
...thermal_flare.sequence.tft	sun_pop	1.597	3	63	1	40.75	0.260
...thermal_flare.sequence.tft	sun_ring	1.600	0	10	1	3.5	0.354

BUNGIE GAME DEVELOPERS CONFERENCE // 2018 DESTINY 2



Name	SequencerNodeName	ParticleMaxSize	NearFadeDistance	PixelShaderCycleCount	ParticleCountHeuristic	ParticlePerfHeuristic	SystemPerfHeuristic
...thermal_flare.sequence.tft	blue shrink	3.5	6	60	1	38.5	0.294
...thermal_flare.sequence.tft	distortion ring	2.5	1.5	25	1	12.25	0.765
...thermal_flare.sequence.tft	flash	6	8	44	1	26.5	0.335
...thermal_flare.sequence.tft	glow flash end	0.099	-1	37	1	21.25	0.212
...thermal_flare.sequence.tft	gpu swirls_inner	0.018	-1	17	125	7	0.025
...thermal_flare.sequence.tft	gpu swirls_outer	0.006	-1	21	376	9.25	0.009
...thermal_flare.sequence.tft	solar smoke glow	7	8	41	1	24.25	0.417
...thermal_flare.sequence.tft	solar_halo - distortion	11	1	58	3.500	37	6.487
...thermal_flare.sequence.tft	solar_halo_small	2.299	6	48	3	29.5	0.292
...thermal_flare.sequence.tft	solar_outward	7	8	59	3	37.75	1.948
...thermal_flare.sequence.tft	Solar_pool_sphere_edge - mod	8	-1	54	1	34	3.442
...thermal_flare.sequence.tft	solar_sun_palletized	1.600	3	49	1	30.25	0.193
...thermal_flare.sequence.tft	sphere hot	1.931	-1	62	1	40	4.050
...thermal_flare.sequence.tft	sun_pop	1.597	3	63	1	40.75	0.260
...thermal_flare.sequence.tft	sun_ring	1.600	0	10	1	3.5	0.354

This is one of our more proactive methods for achieving perf. Whereas the perf hammer is one of our more reactive methods for achieving perf.

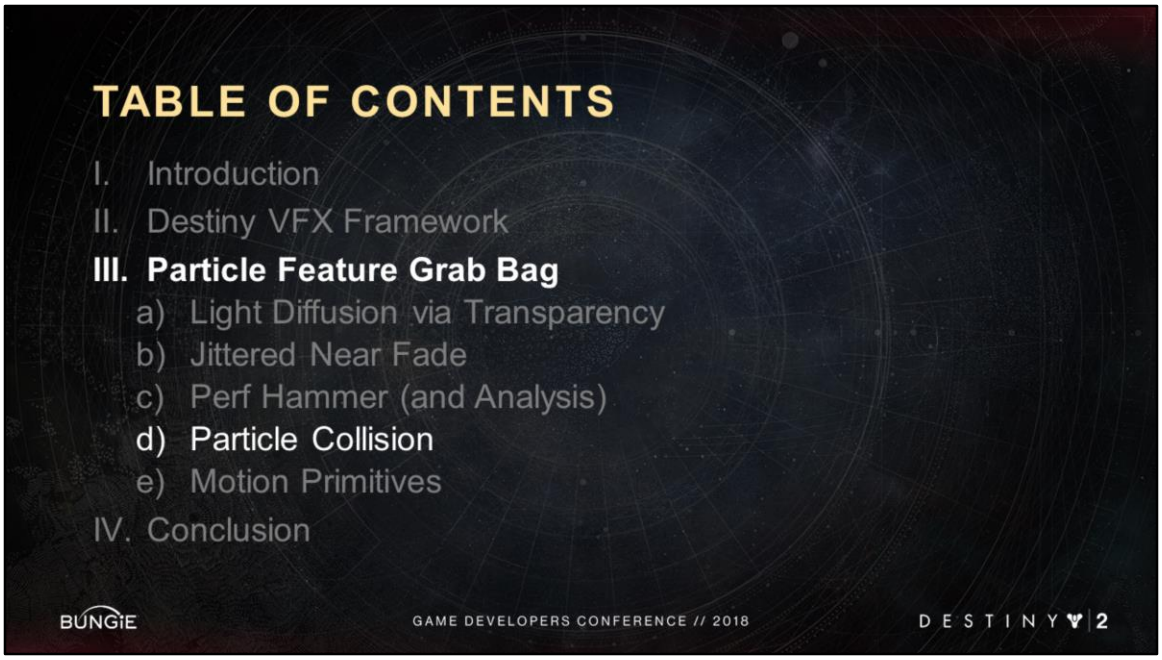


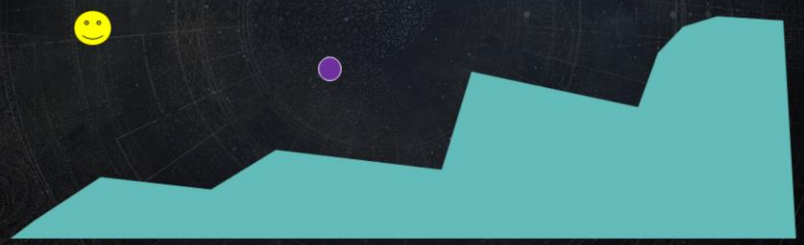
TABLE OF CONTENTS
I. Introduction
II. Destiny VFX Framework
III. Particle Feature Grab Bag
a) Light Diffusion via Transparency
b) Jittered Near Fade
c) Perf Hammer (and Analysis)
d) Particle Collision
e) Motion Primitives
IV. Conclusion
BUNGIE
GAME DEVELOPERS CONFERENCE // 2018
DESTINY 2

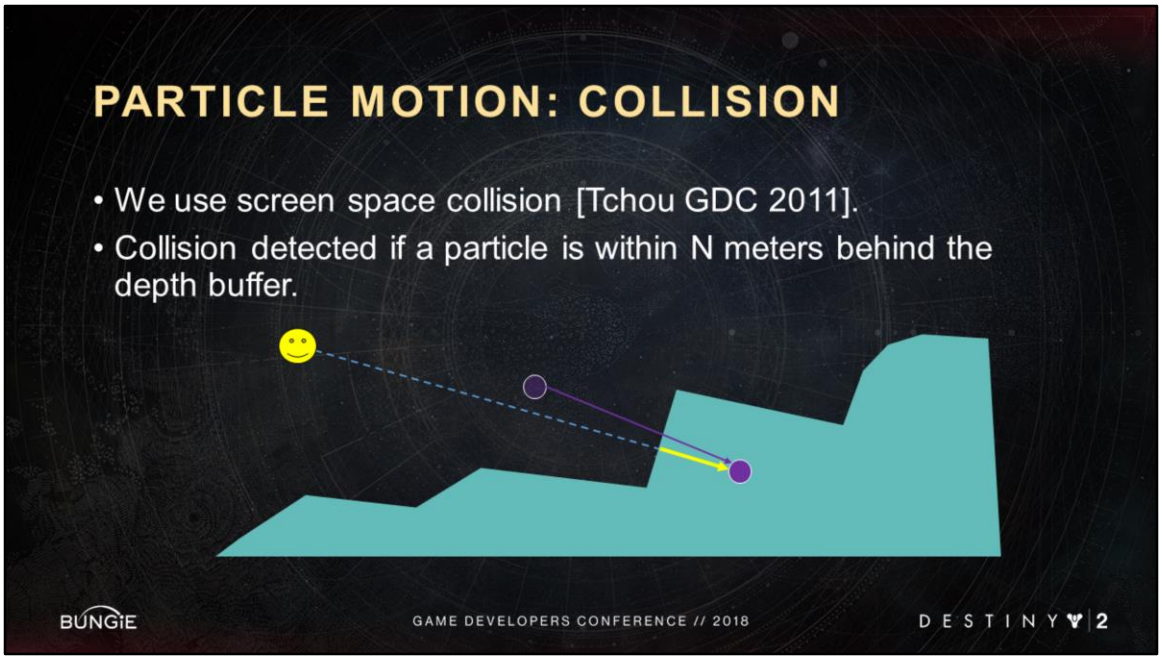
PARTICLE MOTION: COLLISION

- We use screen space collision [Tchou GDC 2011].
- Collision detected if a particle is within N meters behind the depth buffer.

PARTICLE MOTION: COLLISION

- We use screen space collision [Tchou GDC 2011].
- Collision detected if a particle is within N meters behind the depth buffer.

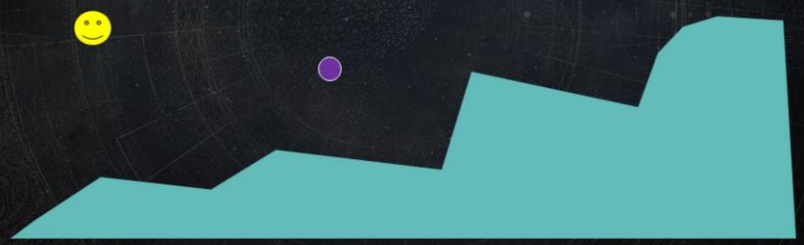


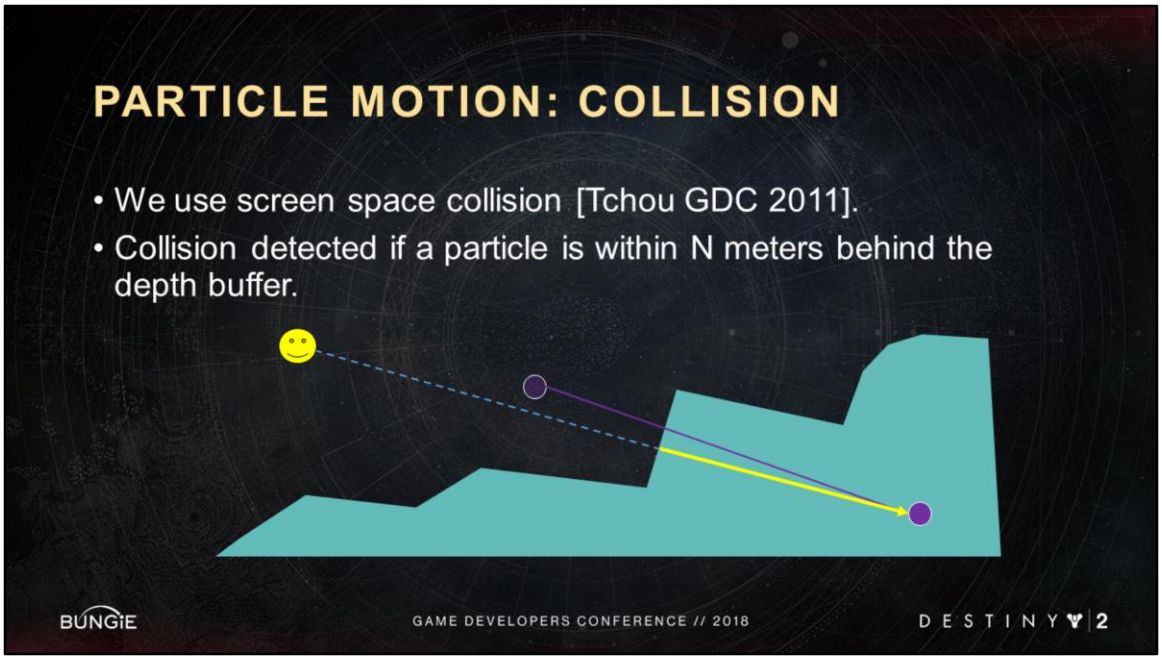


Imagine the length of the yellow arrow is smaller than our threshold, so we call this a collision.

PARTICLE MOTION: COLLISION

- We use screen space collision [Tchou GDC 2011].
- Collision detected if a particle is within N meters behind the depth buffer.





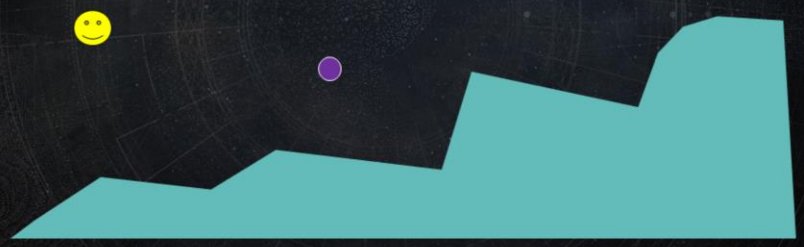
Imagine the length of the arrow is greater than our threshold, so we miss this collision.

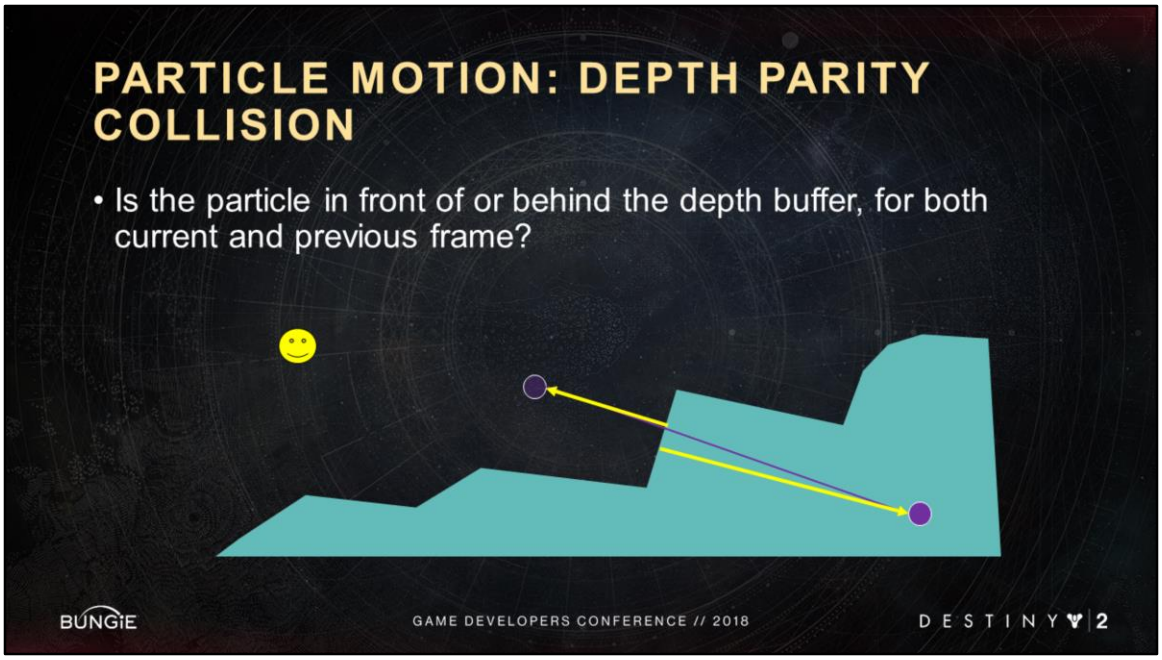
PARTICLE MOTION: COLLISION

- Doesn't work well for high velocity particles.
- We tried a few different approaches and settled on "depth parity".

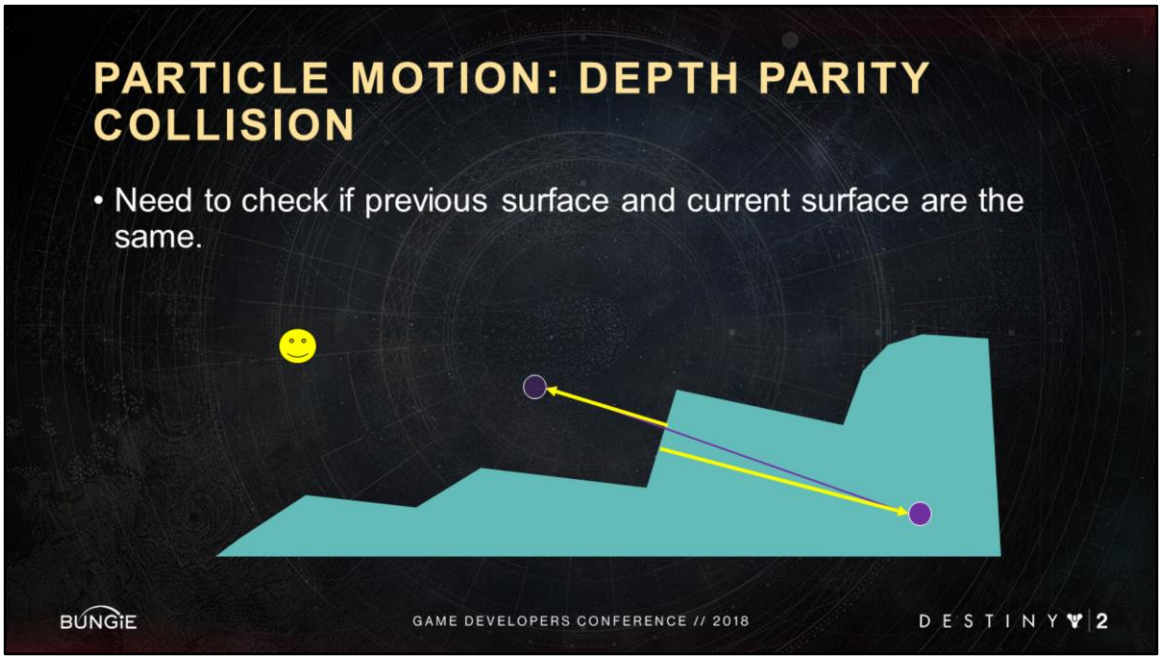
PARTICLE MOTION: DEPTH PARITY COLLISION

- Is the particle in front of or behind the depth buffer, for both current and previous frame?





Arrows point in different directions, so we correctly detect the collision.



Use depth and normal similarity to determine if previous frame depth is from the same surface is the current frame depth.

Titan smash impact effect again. Particles have a very high initial velocity.

PARTICLE MOTION: COLLISION

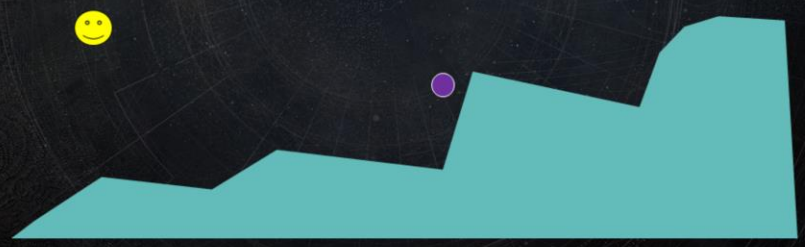
- Also added support for particles sticking and sliding on surfaces.

PARTICLE MOTION: COLLISION

- Also added support for particles sticking and sliding on surfaces.
  - Calculate normal speed and tangent speed.
  - If normal speed below some threshold, start sliding.
  - When sliding, move particle in front of depth buffer.

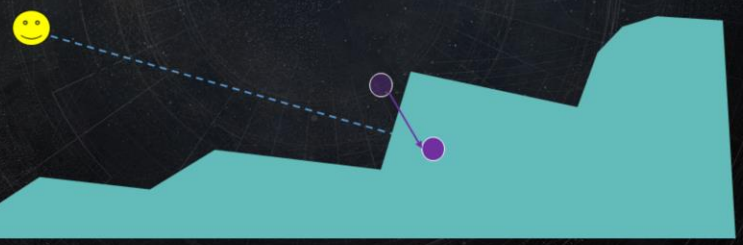
PARTICLE MOTION: COLLISION

- Also added support for particles sticking and sliding on surfaces.
  - Calculate normal speed and tangent speed.
  - If normal speed below some threshold, start sliding.
  - When sliding, move particle in front of depth buffer.



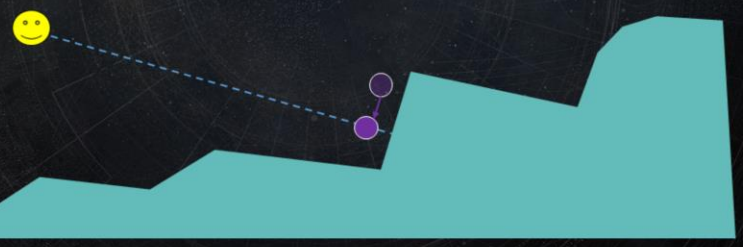
PARTICLE MOTION: COLLISION

- Also added support for particles sticking and sliding on surfaces.
  - Calculate normal speed and tangent speed.
  - If normal speed below some threshold, start sliding.
  - When sliding, move particle in front of depth buffer.



PARTICLE MOTION: COLLISION

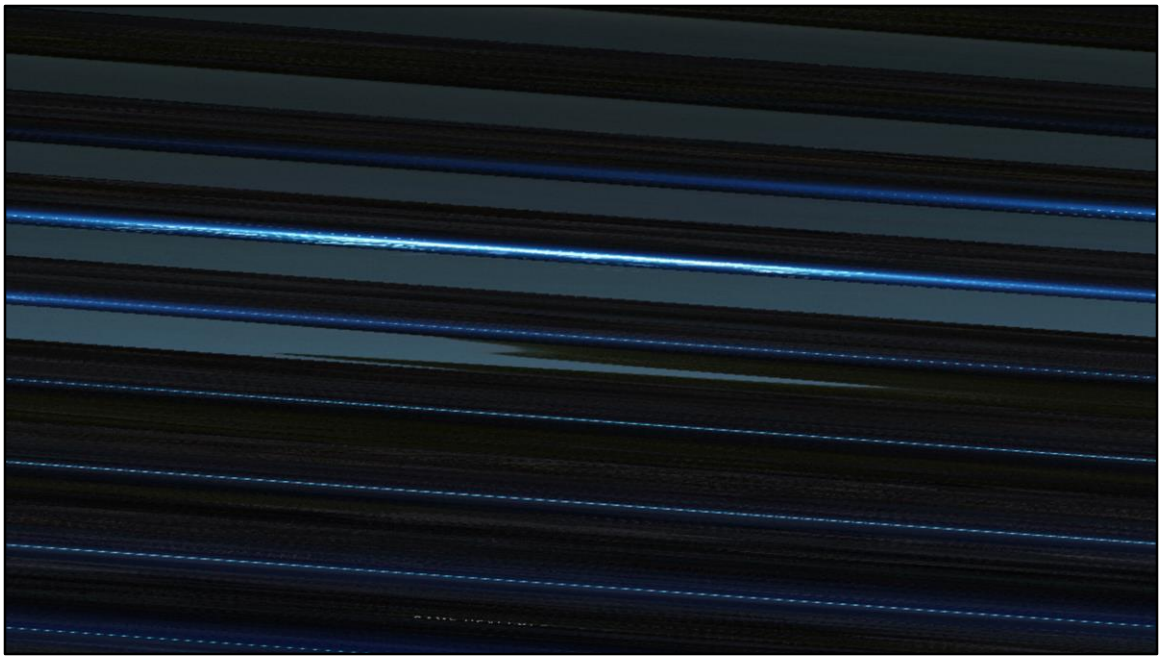
- Also added support for particles sticking and sliding on surfaces.
  - Calculate normal speed and tangent speed.
  - If normal speed below some threshold, start sliding.
  - When sliding, move particle in front of depth buffer.



Ideally the surface would specify friction as well, but we'll take what we can get.

PARTICLE MOTION: COLLISION

- Each particle system specifies some friction coefficient.
- These behaviors were templated as well.



Here’s some particles that change behavior and appearance after the first collision is detected. It’s a good example of what’s possible in the Destiny Engine, because since the collision count is one of our expression parameter inputs, we can change appearance properties on collision, like making the particles brighter, or drastically changing the motion properties after collision.

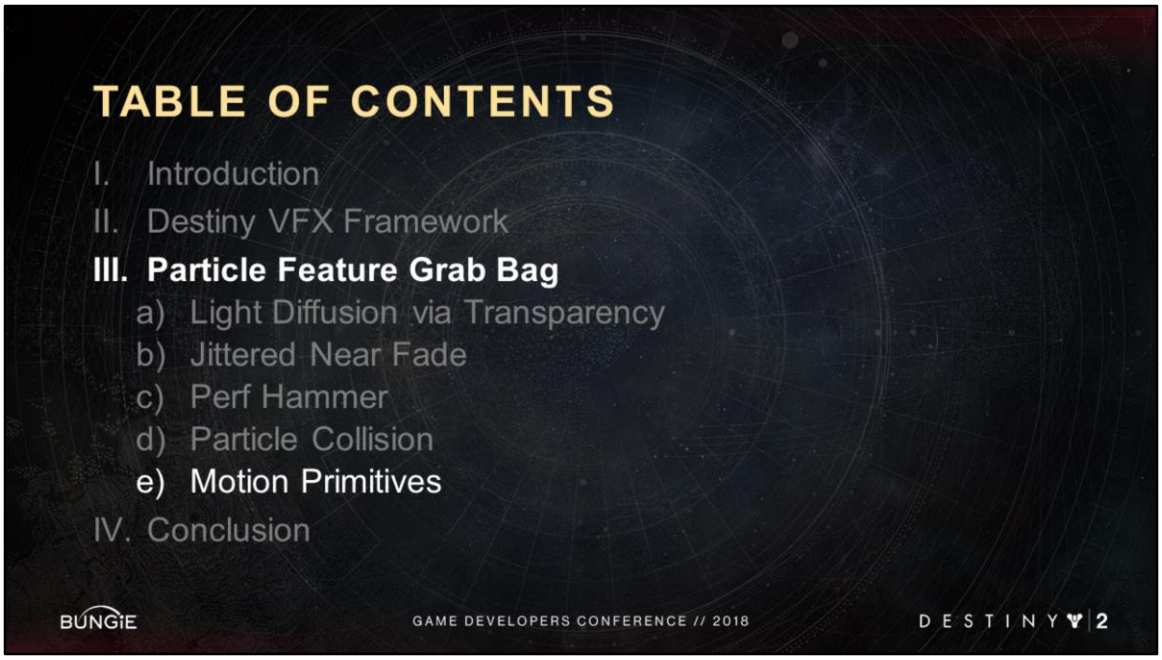
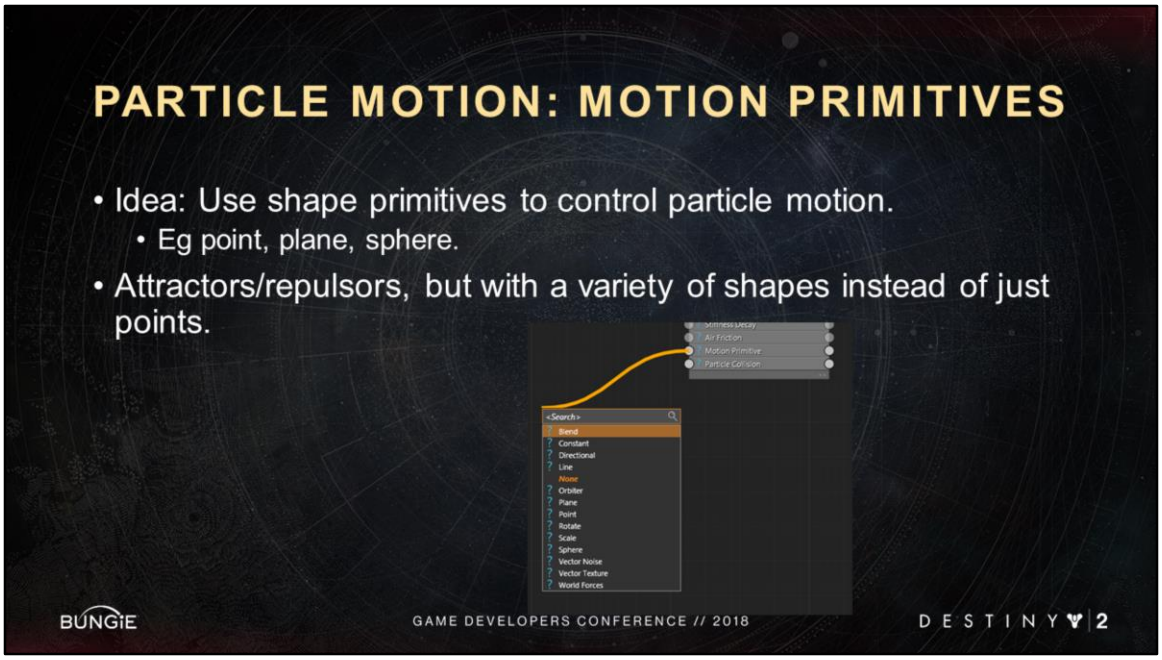
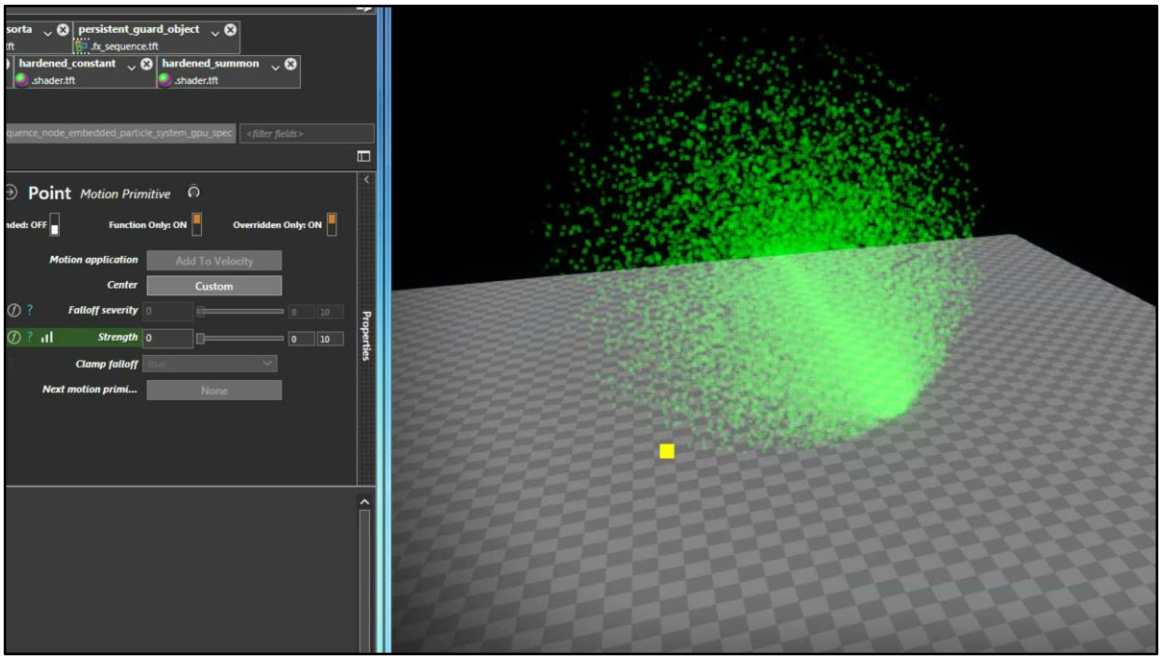


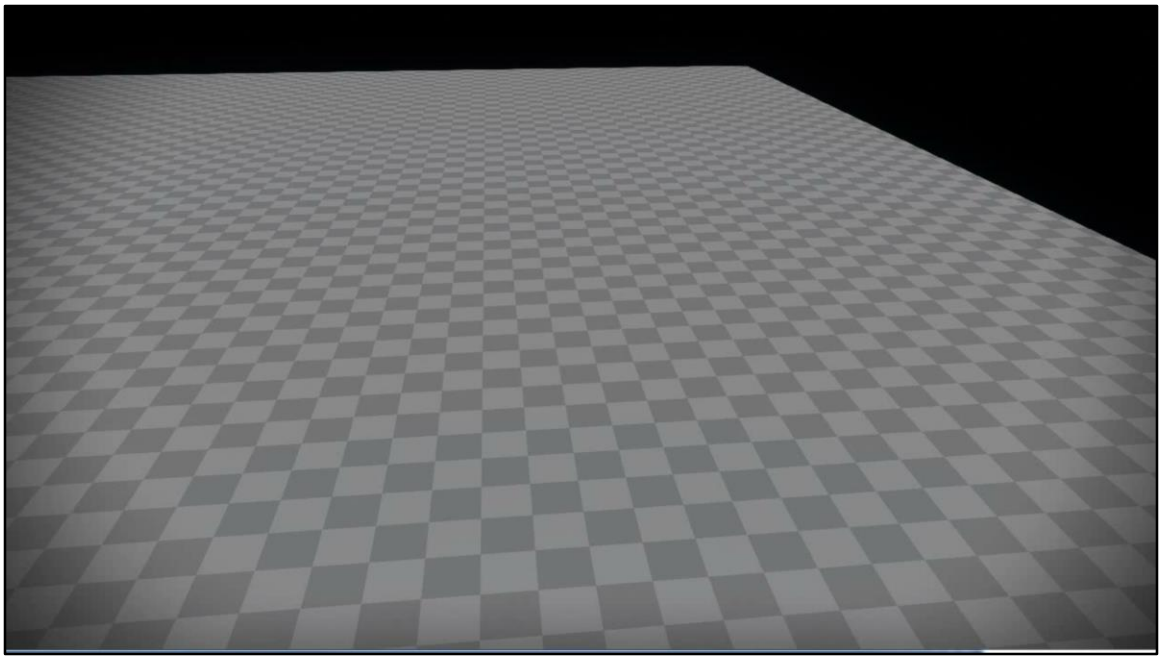
TABLE OF CONTENTS
I. Introduction
II. Destiny VFX Framework
III. Particle Feature Grab Bag
a) Light Diffusion via Transparency
b) Jittered Near Fade
c) Perf Hammer
d) Particle Collision
e) Motion Primitives
IV. Conclusion
BUNGIE
GAME DEVELOPERS CONFERENCE // 2018
DESTINY 2



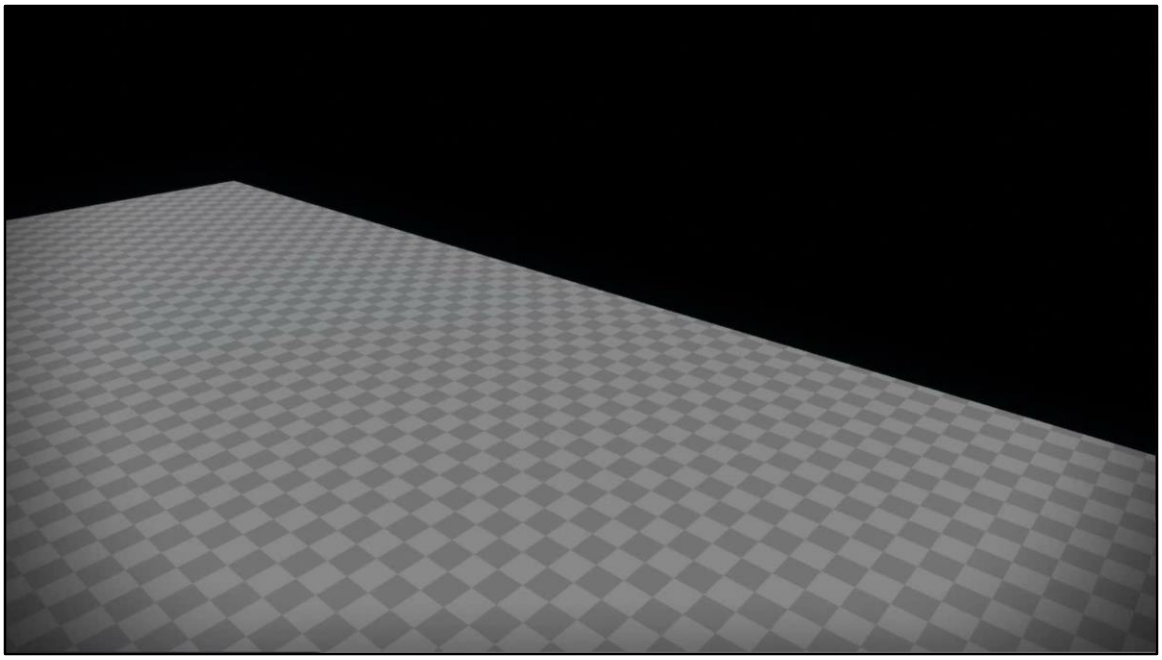
The idea is to use shape primitives to control particle motion like points, planes, and spheres. It's very similar to attractors or repulsors seen elsewhere, but extended to use a variety of shapes instead of just points.



Here's a simple example using a point attractor. The yellow point is debug only. It's our visualization of the motion primitive. In this video I'm modifying the strength of the attractor and seeing that update immediately, and I can also modify the position of the point attractor and see its influence change and the debug visualization change.



Here’s an example of a line motion primitive. Every frame, each particle calculates the point on the line it is closest to, and then accelerates towards that position. Thus the particles propagate forward but create this interesting oscillating shape.



Now let’s go back to the point attractor and see where expressions and parameter inputs can get us.

We can see the particles sticking to this spherical shape, which is accomplished with a sphere motion primitive. Each time the bubble “breaths” that’s accomplished with an expression on the sphere radius, driven over the age of the system instance. A vector noise texture provides these sort of clumpy snake patterns around the sphere.

We also provide methods for more directly influencing the motion of the particle, like forcing particles in an orbital pattern, as opposed to trying to use a standard point attractor to accomplish the same thing which can be difficult to fine tune. In this case it makes it easy for the artist to match the motion of the particles, which are the little orbiting bits here, with the rotating pool of energy underneath, which is an animated texture warp.

Table of Contents

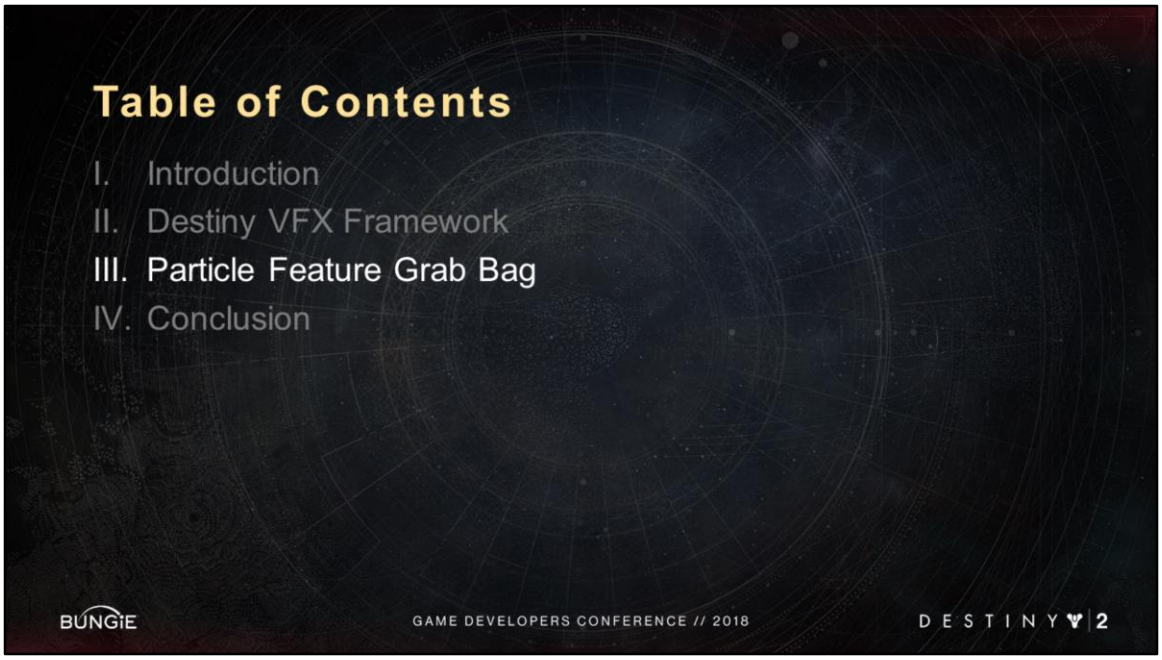
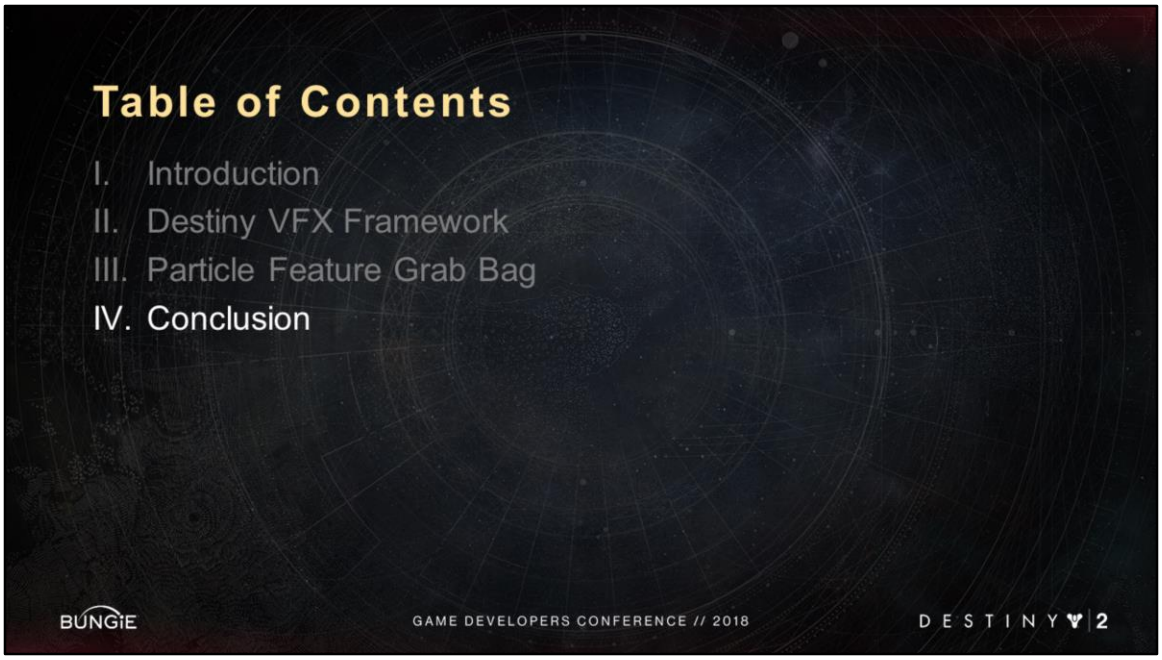
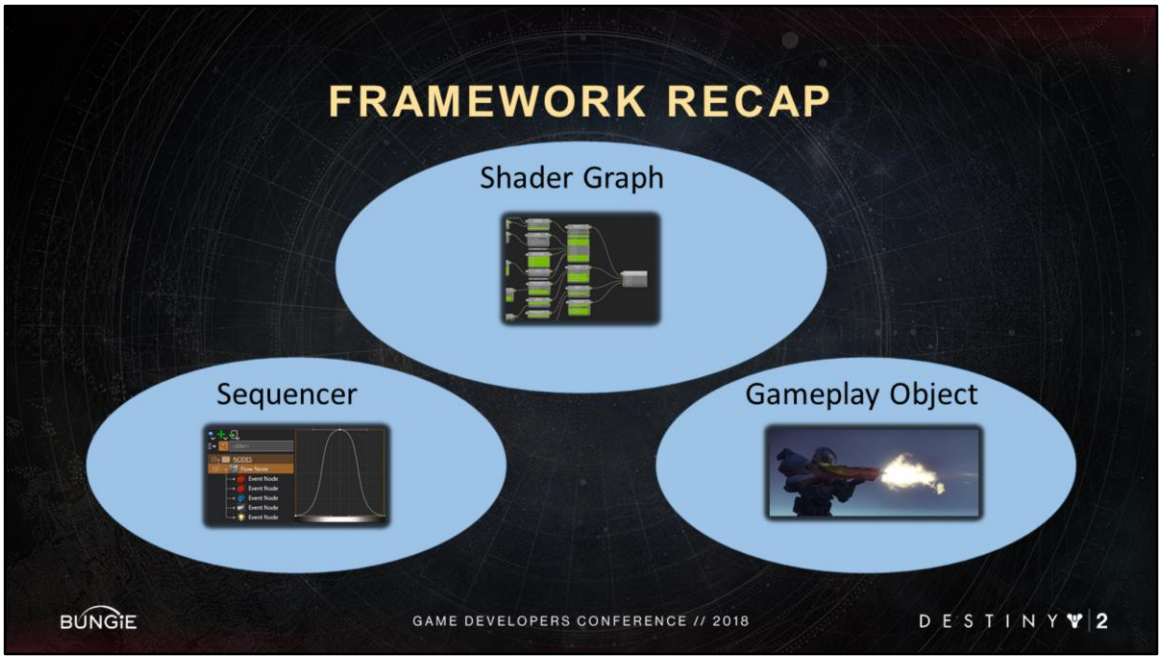


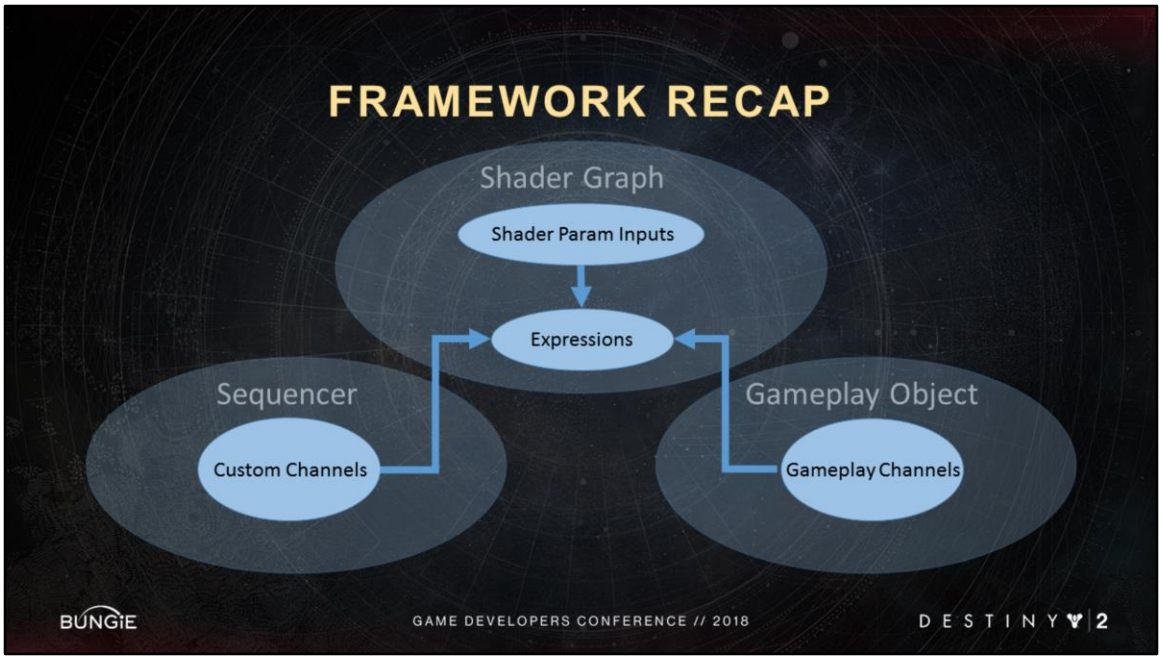
Table of Contents
I. Introduction
II. Destiny VFX Framework
III. Particle Feature Grab Bag
IV. Conclusion
BUNGIE
GAME DEVELOPERS CONFERENCE // 2018
DESTINY 2



In conclusion, I wanted to give a quick recap

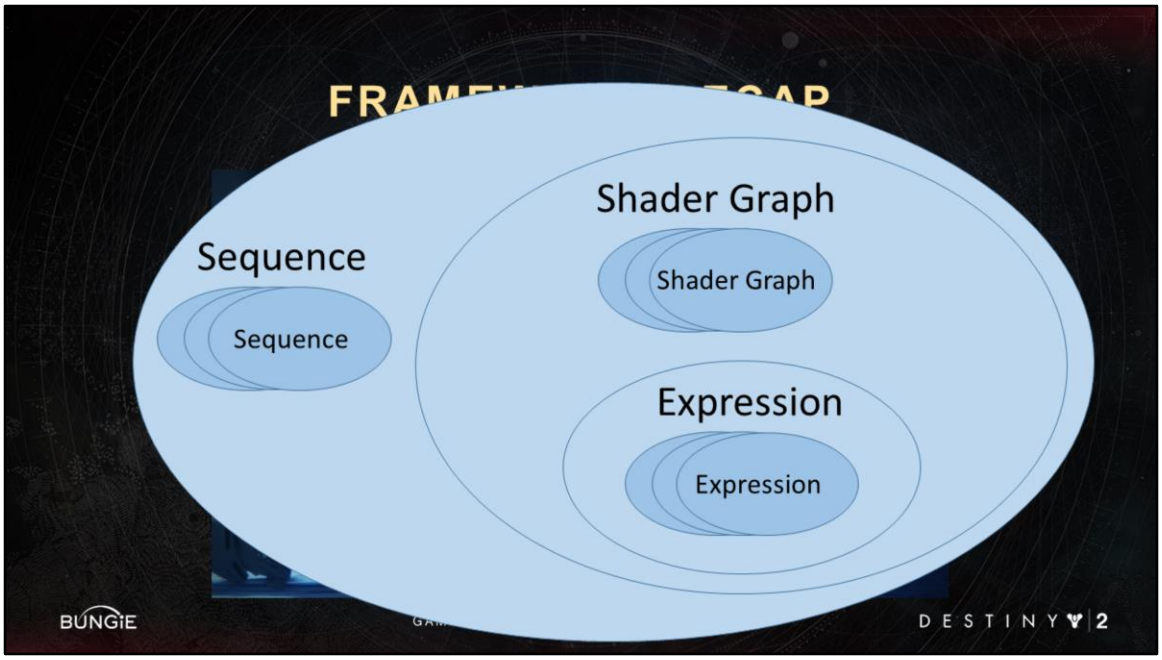


We talked about the major parts of our framework



We saw how each part provides inputs that can be used by expressions
Artist can freely create complex expressions across all params across shader types

It gave them tons of flexibility, expressive power, and creative freedom

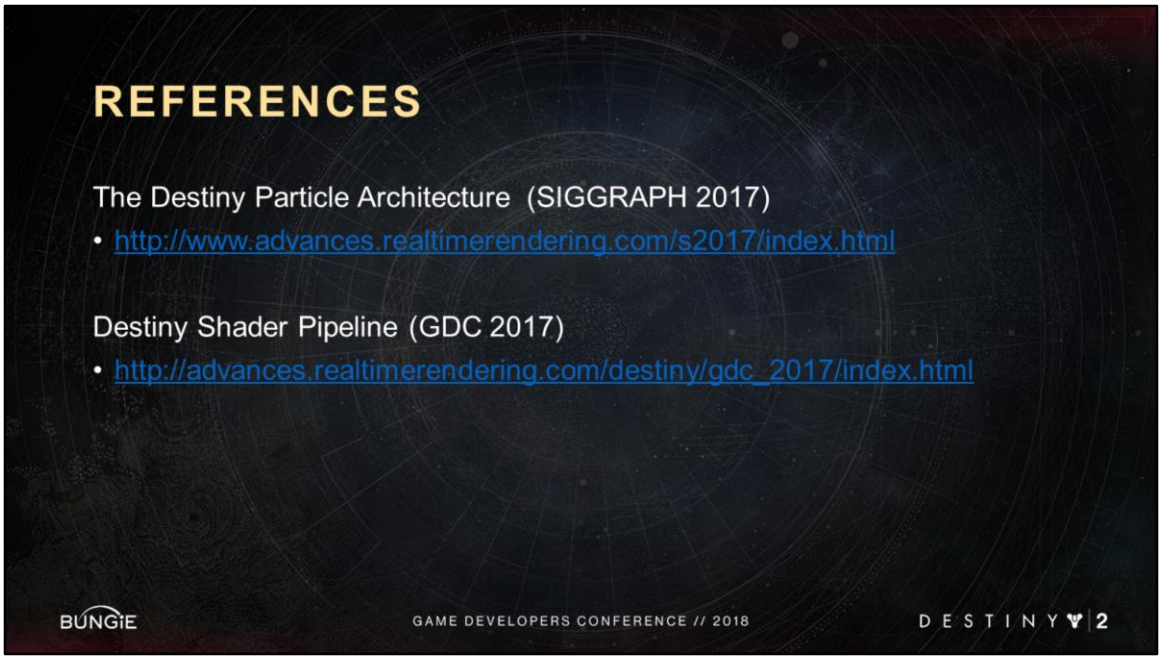


From a house keeping **and** content management perspective
We rely heavily on **modularity**
<Advance Slide>
Sequences can be built from more sequencer (inlining)
Shader graphs can be built from more shader graphs (templating)
Expressions can be built from more expressions

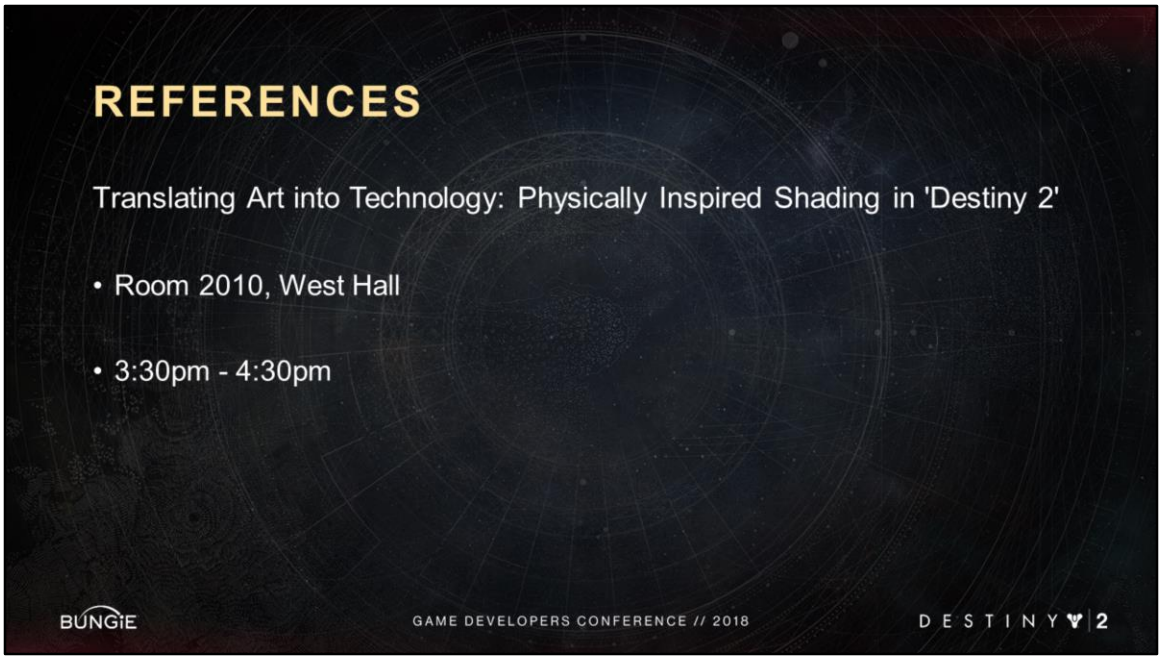
The more modular our content is, the more we can reuse **and** evolve it over time.

We shared with you a grab-bag of features that was really high value for us
Many of these are framework-agnostic, can be applied to any framework
Within our framework, though, these synergized super well with expressions, shader param inputs and templates
and really empowered artists to come up with things we could have never anticipated.

Alright! Thanks you all for your time
And thanks to these teams for their love and support



For a more technical deep-dive, check out our previous talks on our shader and particle technology



If you want to know more about our graphics tech, check out this other Bungie talk at 3:30 today
“Physically Inspired Shading in Destiny 2“

Bungie is hiring! Check out our career page.

Here's our info, get in touch.
And with that, we'll open the floor to questions.