Hey everyone. Thank you for being here.

This session is called Walking, Talking, and Projectiles: Storytelling Tools in Destiny 2.

So you might guess that this talk is about our recent experiences
making storytelling tools that help control movement, dialog and explosions.

You’re in the right place.

Now when I first starting to think about applying to speak here at GDC, I actually had a simpler talk in mind.

Standing Still and Talking.

No projectiles. No walking.

The reason, is because standing still, by itself, is some seriously under-appreciated tech. There's a ton of amazing games that do this very well – I'm sure you can think of a few – role-playing games, big sandbox games, open world games, massively multiplayer games.

It's also pretty darn common in real life. Look at me:

I'm standing still right here. I'm even talking!

It's pretty sweet be here.

So given all that, why is standing-still tech so hard?

For one, it’s hard because standing still is about movement.

Weight shifts, head and facial motion,
breathing; we move our eyes and head to watch who's speaking, we jolt away from loud sounds. It's all instinctual, and without these movements everything looks strange. Unreal.

Some game systems are created with this in mind. Player characters are built with continuously moving idles to keep stillness from becoming “too still.” AI combatants have a similar set of idles and stationary behaviors.

And yet — standing still is hard for another reason, and that's the main subject of this talk and the reason for the tool I'm about to introduce.

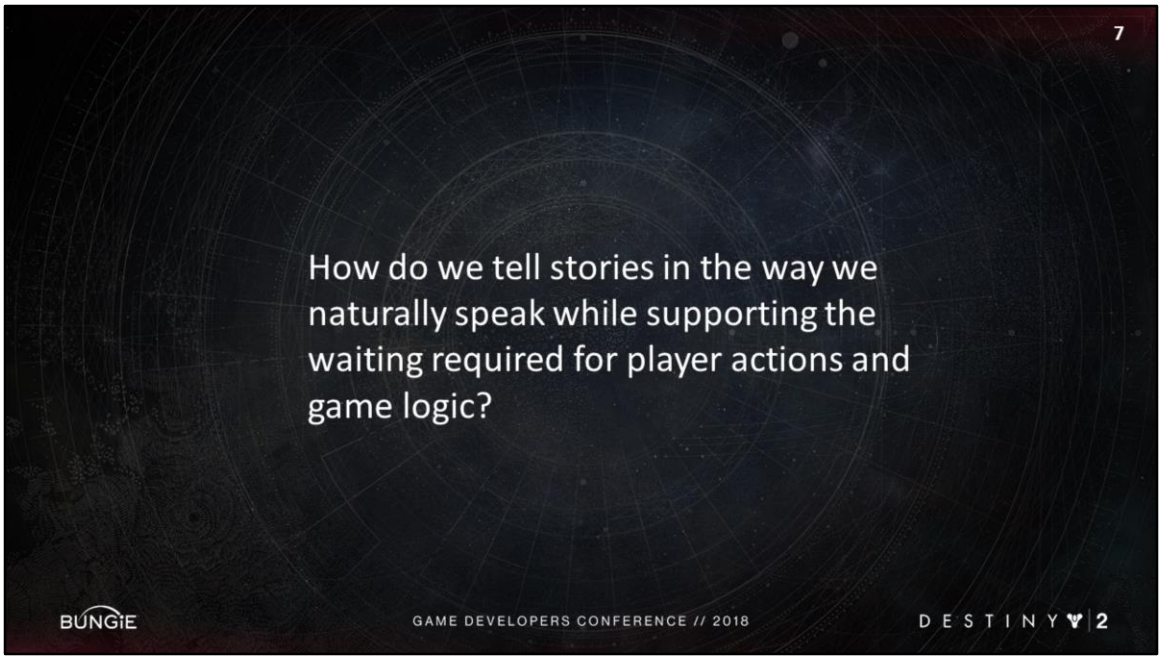
Standing still is especially hard because standing still requires…

Waiting. You see, the NPC character can't move and talk on and on. This is a game! You have to wait for the player, and react to the choices they make!

And waiting is really hard in the moments just before a story begins, because not only is there movement during the waiting, but there's also game logic that's running and running right alongside. The game logic is querying, calculating, and deciding what to do next and when to do it.

The challenge is that the folks who make the story are often not the same people as those who design the missions that serve as the stories' stage. For a game like Destiny 2 both story and gameplay are needed.

And this leads to our fundamental question…



How do we tell stories in the way we naturally speak while supporting the waiting required for player actions and game logic?

This is the question we're going to answer.

Now for Destiny 2, we didn't know all of that yet. Instead we, the story engineering te am, started out with a few goals of our own.

And, like with most goals, they are easier said than done.

Usability Goal: First and foremost, we wanted a usable tool. It’s an easy thing to say, and hard thing to specify. We wanted to tell simple stories faster and without a ton of coding experience required. We also wanted to keep players in the action by experimenting with stories that didn't require taking camera control.

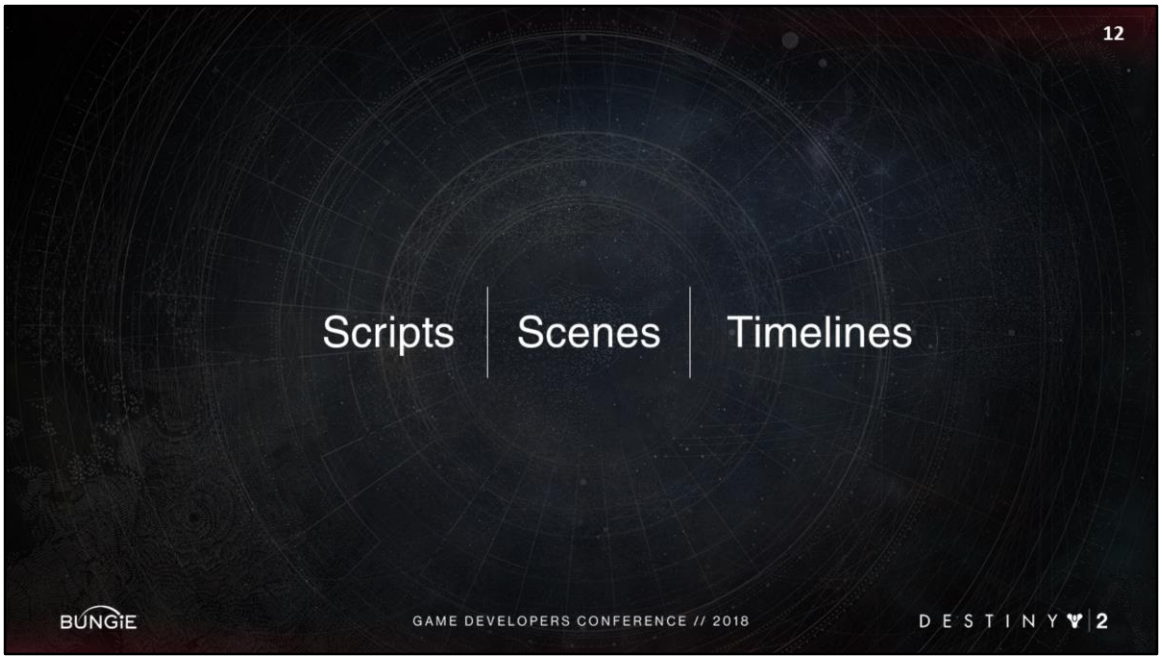
AI Control Goal:
We also wanted to tell stories that would dovetail with combat. So any narrative tool would need some high level control over AI and non-player
character movement, targeting, shooting, temperaments and other behavior.

Lastly, **Triggering Goal**. We needed a unified way to trigger several different engine sub-systems. The Destiny 2 networking model has a bunch of advantages such as map streaming and matchmaking, but it makes triggering certain client side subsystems harder.

If we could hit this goal, we'd make sure we weren't leaving any game systems behind , and narrative designers can be free to choose the right tool for the job.

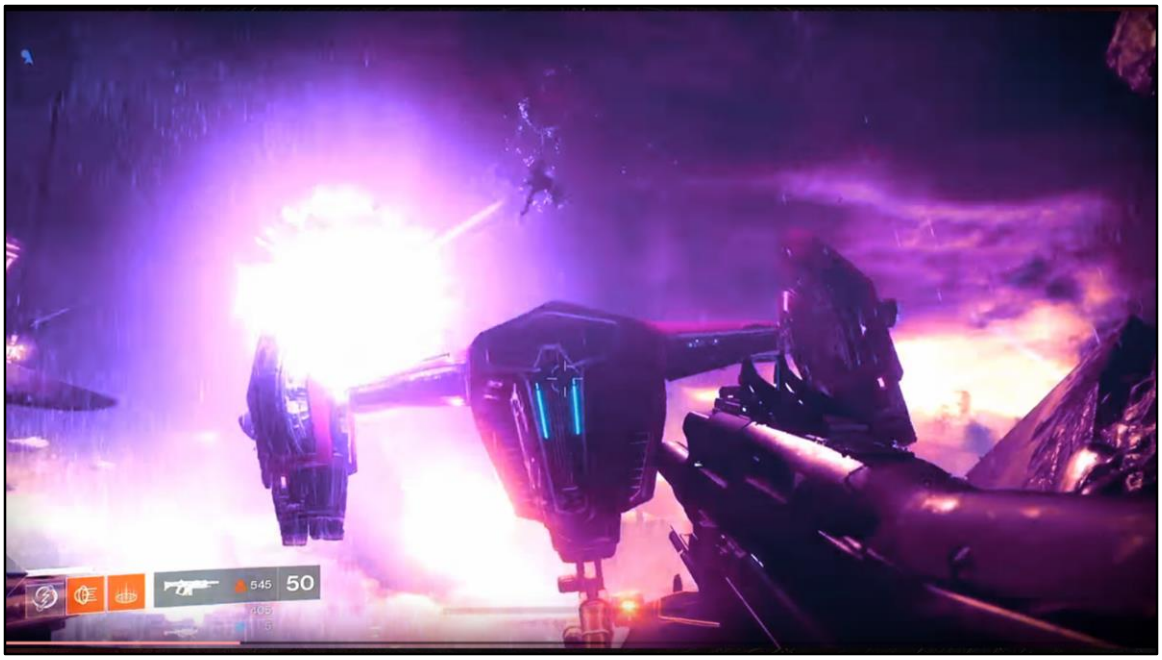
After quite a bit of discussion we decided to go with visual scripting to help tell storie s.

The tool was called the Scene Editor, and it would most cleverly edit…



Scenes. The intuition was that these visual scripting scenes would fit in-between script and timelines. Where **scripts** are programming code that controls high level mission progression such as counting the deaths enemies to complete the mission, and **timelines** control precise low level, frame accurate animation, sound, and special effects. There's no branching or waiting in a timeline. So once they start, they run to completion.

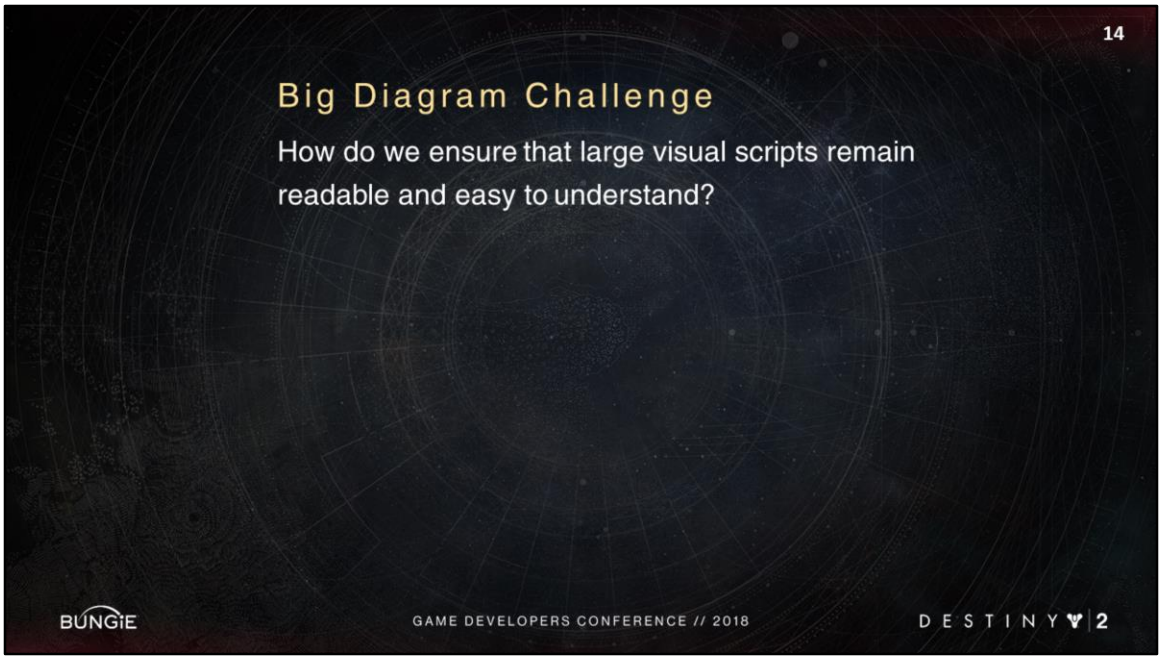
The power of timelines is that they ensure you can be confident to the millisecond when an explosion's shockwave will reach the vehicle. If you script the logic, you can be confident that the explosion won't start until the seventh enemy has been killed. And when you connect both together in a scene, with an easily scannable visual scripting style, game designers get their logic, storytellers get their story flow, animators get their precision.



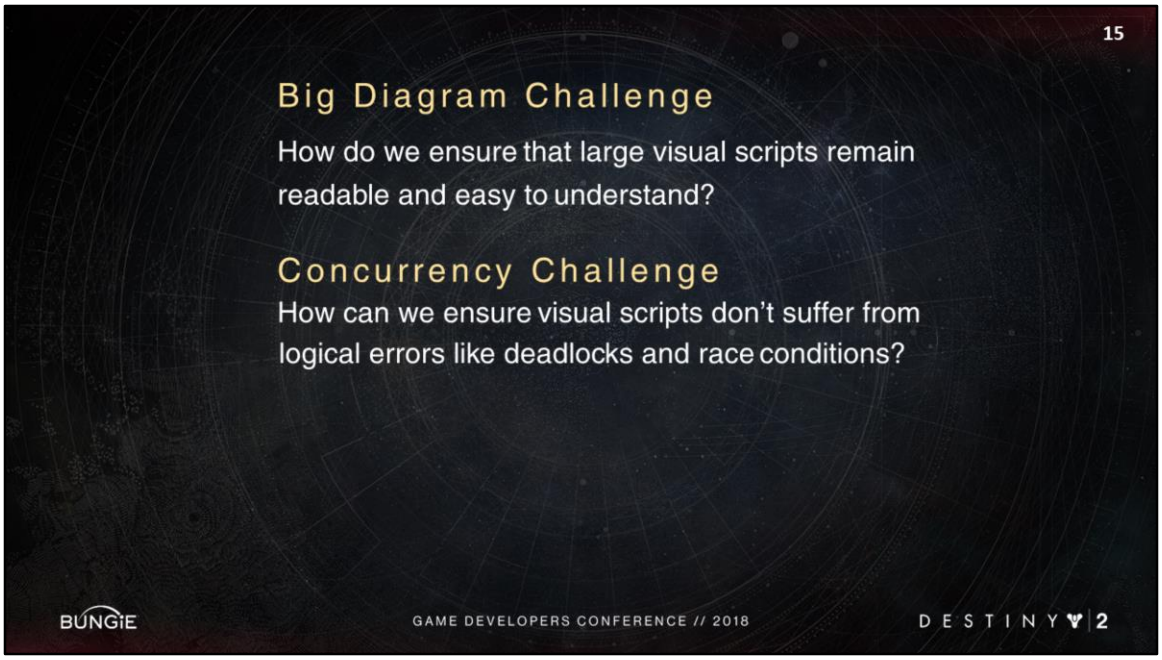
BOOM! WE DID IT!

Visual scripting solved all our problems, and everything was easy… no… that's not true… visual scripts are awesome,
but they still have big challenges that we as an industry are still grappling with.

So let's talk about those.



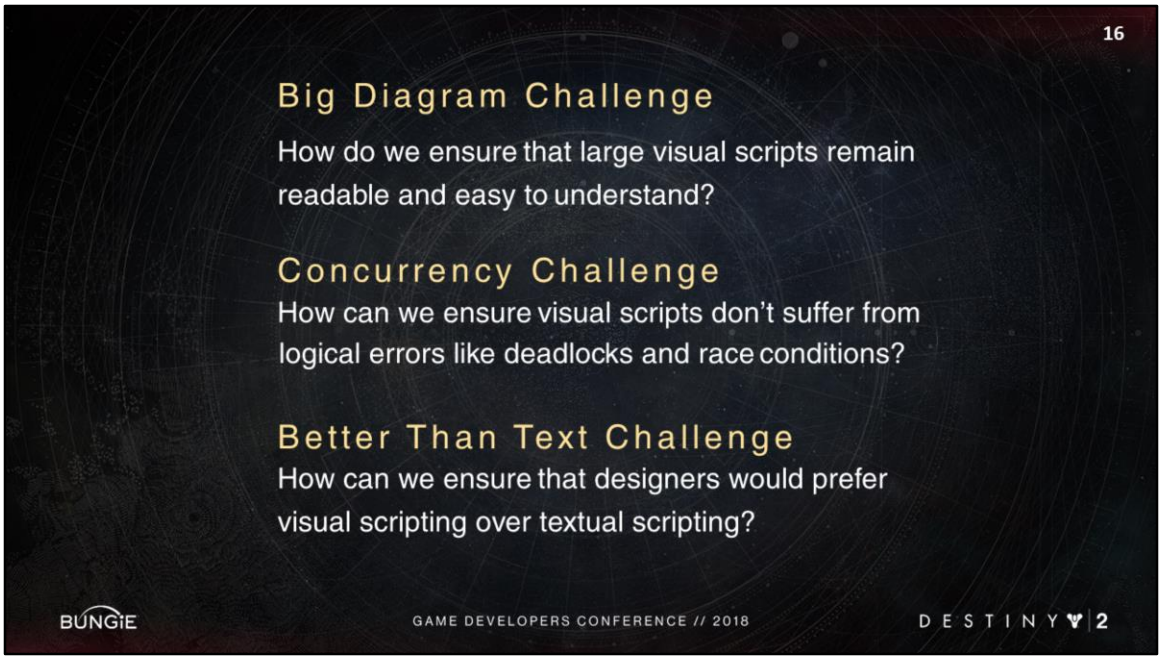
The first challenge is what I call the **Big Diagram Challenge**. It turns out that as visual scripts become bigger, they become hard to understand with so much scrolling and connecting lines all over the place. What can we do from a design perspective to address this?



Next is the **Concurrency Challenge**. AI and NPCs must be controlled in parallel, and an y system that has concurrency is quite susceptible to deadlocks and race conditions. Yes, it turns out characters both walk and talk. It's like patting your head and rubbing your stomach. The hard part is doing it at the same time!

Here's an example: one squad waits for the doors to blast open before entering. A second squad waits for the first squad to enter and then follows them in. Now who opens the door? If we accidentally pick a character on the second squad to open the door, the battle will never start, because that character won't start moving until the door is open. They'll all stand around looking foolish. No grand entrance. That's what I mean by deadlock.

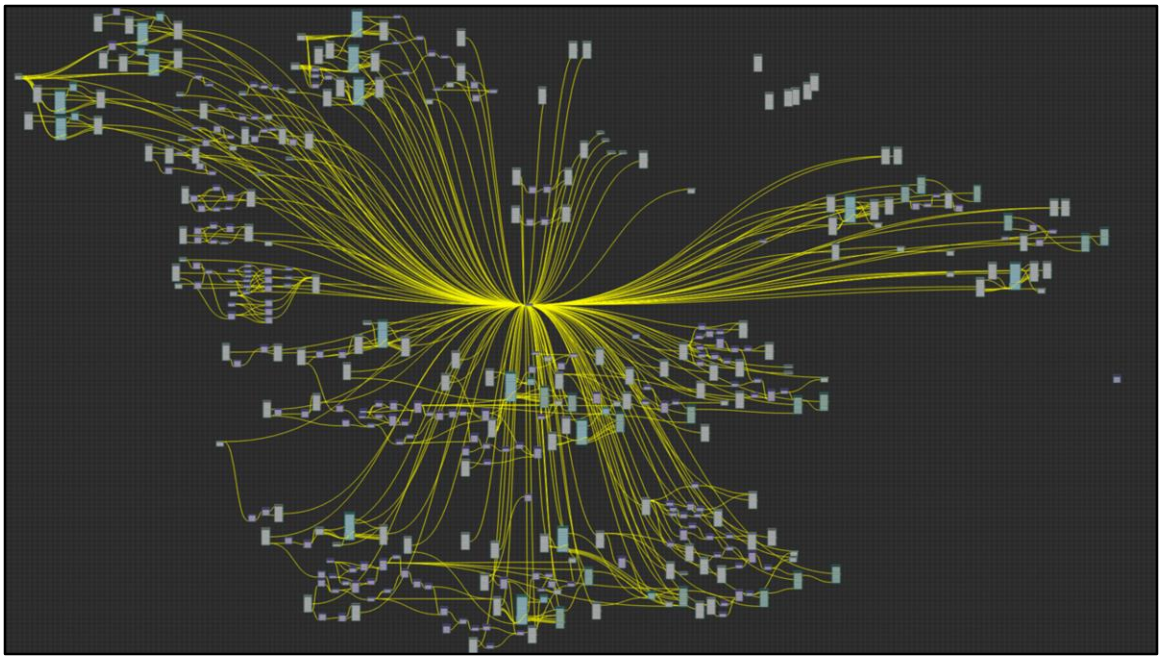
This sort of concurrency problem is challenging for programmers, and we think about this stuff all the time. Because of our usability goal our target audience is everyone who's wants to tell stories. So what can we do from a tool design perspective to prevent concurrency problems?



The last challenge is the **Better Than Text Challenge**.

When we asked around about
people's experience with visual scripting they mentioned that folks might start with visual scripting, but after a while they end up rewriting everything into textual scriptin g because it feels better in some way to
them. Is there anything we can do from a tool perspective to ensure the benefits to n arrative designers outweigh the perceived costs of leaving text scripting behind?

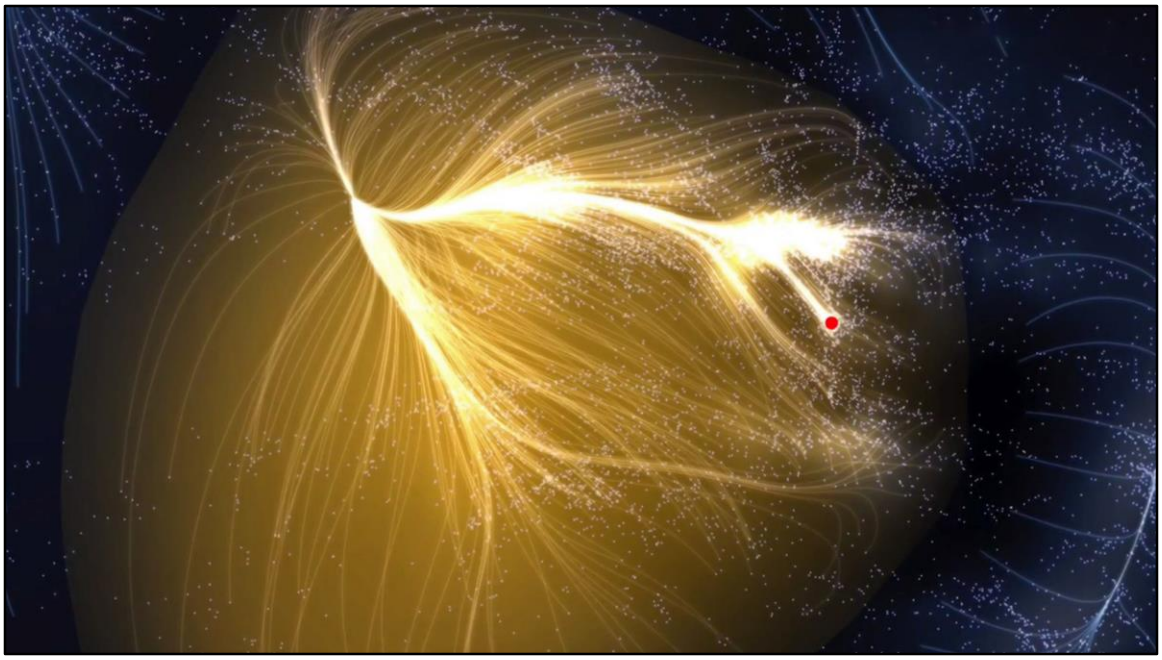
First up, let's focus on the **Usability Goal** while keeping in mind the **Big Diagram Challenge**. How can we ensure visual scripts remain readable and easy to understand?



This problem is hard.

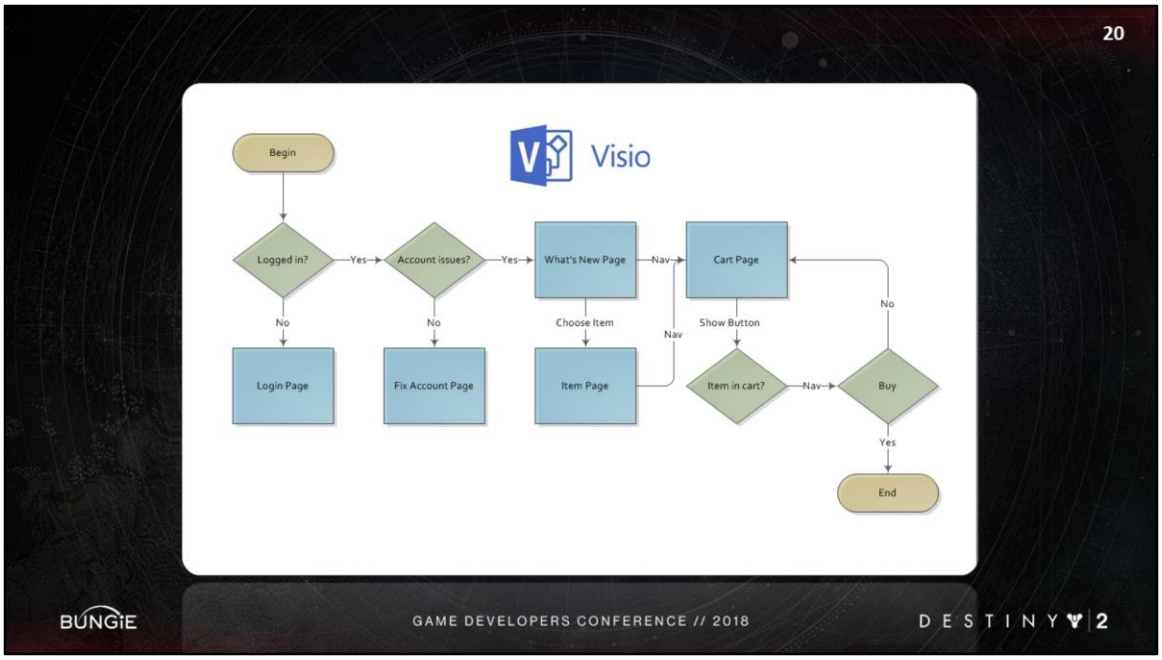
Now this is not a Bungie script. But for those of you who are a bit skeptical – I know you out there – this is probably what you were picturing when I mentioned visual scripting.

As you can see visual scripts lend themselves to branching and rejoining in a powerful ways, but the size of the drawing can get…



…overwhelming

Now, you wouldn’t think it’s possible to have a lot of expertise in giant flowcharts, but it turns out I actually do.

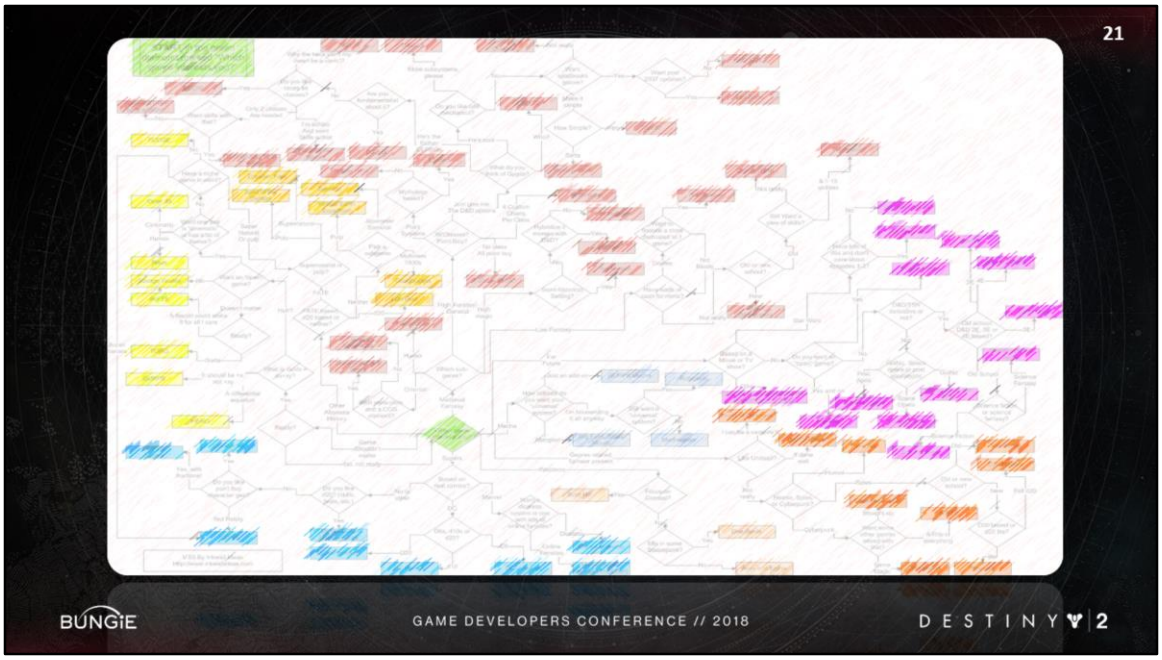


In a previous life, back before I founded a small indie game studio and before being here at Bungie, I actually was a part of Microsoft on the Visio team.

I was a software engineer working on core Office Visio features like integrating the Ribbon, theming, and programmable shape authoring for flowcharts, timelines, org charts, and more.

Needless to say I saw a ton of connected diagrams.

And I can say from experience that drawings get…

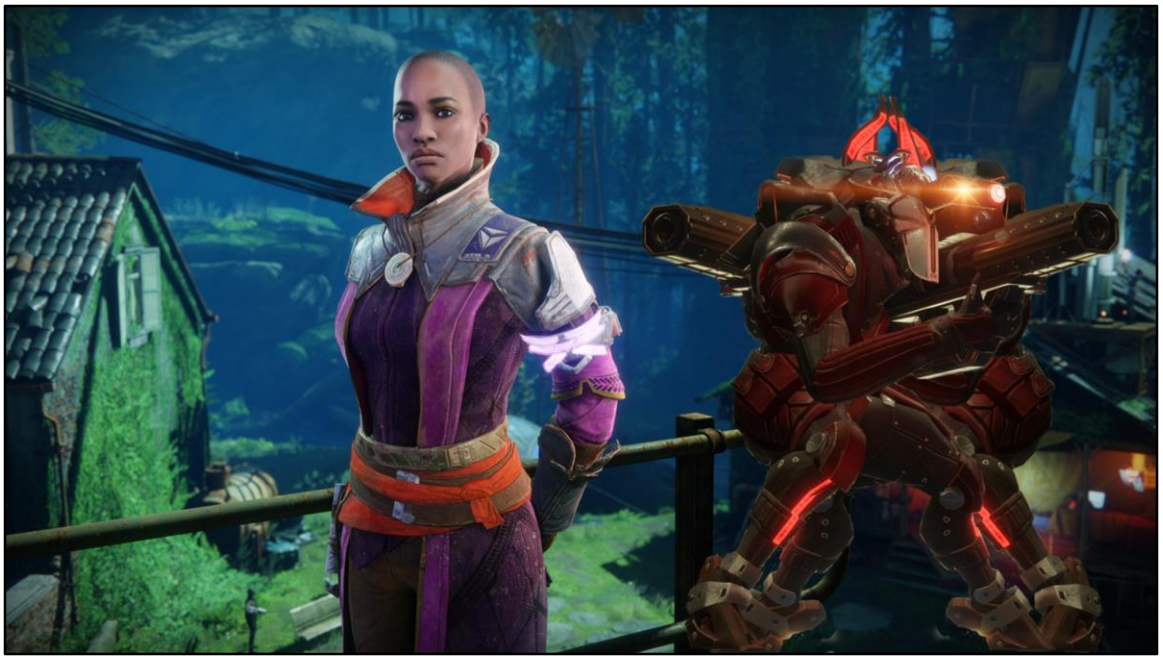


Messy. This is not unusual!
The bigger the diagrams or the more connections, the harder they are to follow.

I think you've all felt this. You open up a big diagram and your brain doesn't even kno w where to start. You don’t know where to look. You’re eyes will scan around looking for something to latch on to.

Ok, enough of business diagrams. Let’s tell stories.

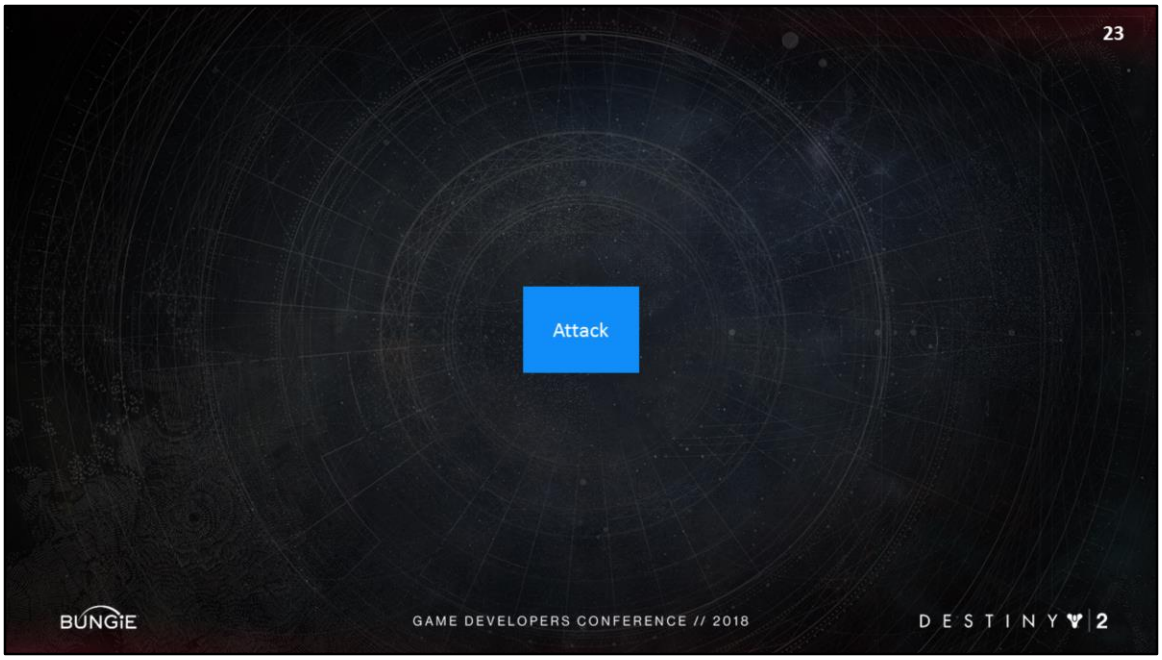
This story is about…



Ikora Ray. A Warlock and elite member of the Vanguard that leads us guardians in defending this solar system. Ikora is out investigating a disturbance in the European Dead Zone, when she comes across a Psion. Ikora stares into the eyes of the Psion, intimidating it, and the Psion immediately runs away to raise the alert, but Ikora shoots it before it can get away.

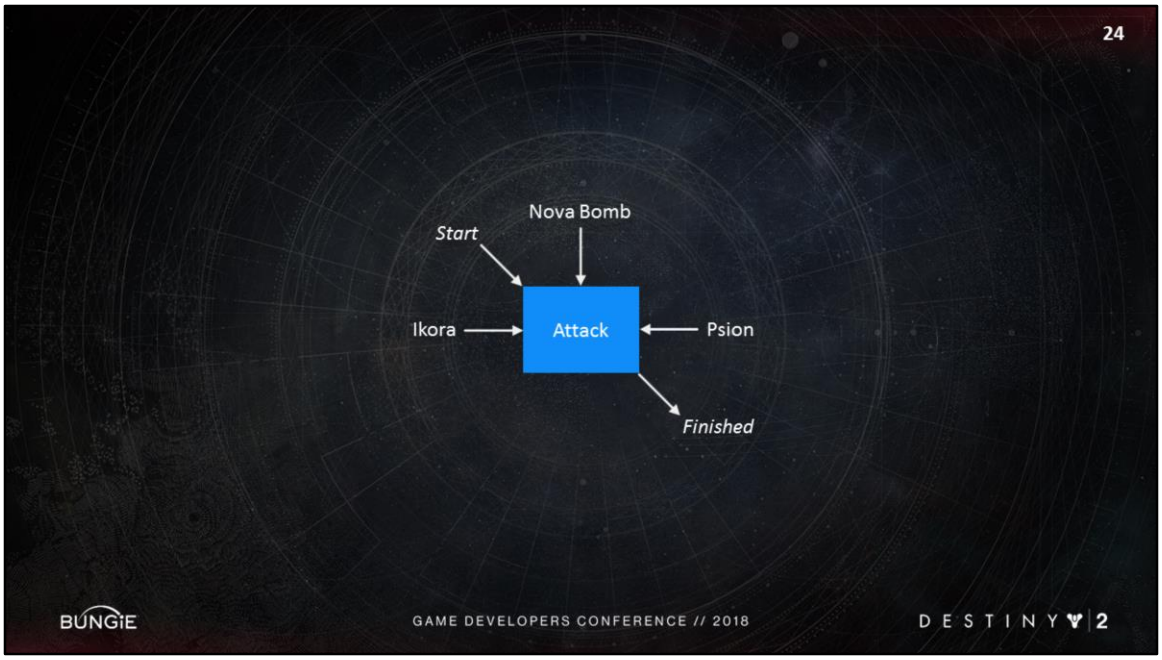
How do we tell this story? It, of course, can be told in script because code can do anything. It's Turing complete. This doesn't make it easier to tell, mind you, but it's important to acknowledge to remember what's already possible. This story can't be told in timelines, however. This is because we wanted to use AI pathfinding and shooting and pathfinding and rockets don't have a fixed timeframe. Some paths last longer than others, some rockets travel slower or faster, so the nodes need to wait for the action to finish.

Let's try visual scripting.



Ok, we made a box and we're attacking! So simple. I knew we could do it! Ah, but who's attacking? With what are they attacking?

This is the challenge with flowcharts and connected diagrams,
this information is coming from somewhere else other than the attacking node.



The weapon used is also an input, as well as the attacker's target.

When does the attack start? When did it finish?

This is a hint at a design goal we pursued, which was to put information together into one place. Like with like. The attacking node must also include who the attacker is and who the attacker is targeting, in this case the Psion.

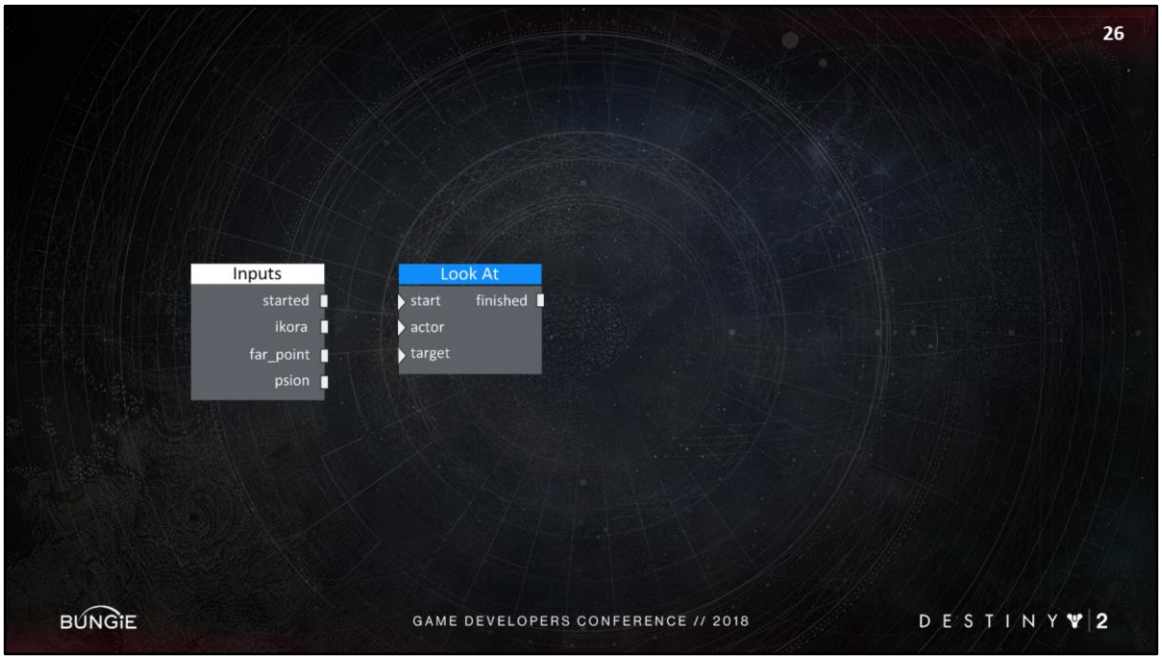
Now let's move away from a business flowchart approach into more traditional visual scripting approach. We'll use that as a jumping off point, and then walk through some design iterations until we reach the scene editors real user interface.

First, it helps to write down our **inputs** so we know all the data we have at the start of the story.

In the case of this Ikora story there are four inputs (X):

- A **trigger** that indicates when the mission has **started**
- **Ikora** Ray, our protagonist.
- **Far point**. The place the Psion is trying to run to.
- And the **Psion**, is the enemy that's getting away.

Next, let's show our actions…



The first action is **Look At** action.

All actions have a **start** input trigger indicating when it starts, and a **finished** action indicating when it has completed is behavior.

The **Look At** action also has two more inputs, the **actor** who is doing the looking, and the **target** that the actor is looking at.

The second action is **Run To**. It has **start** and **finished**, and also has two more inputs: the **actor** that's running, and the **location** the actor is running to.

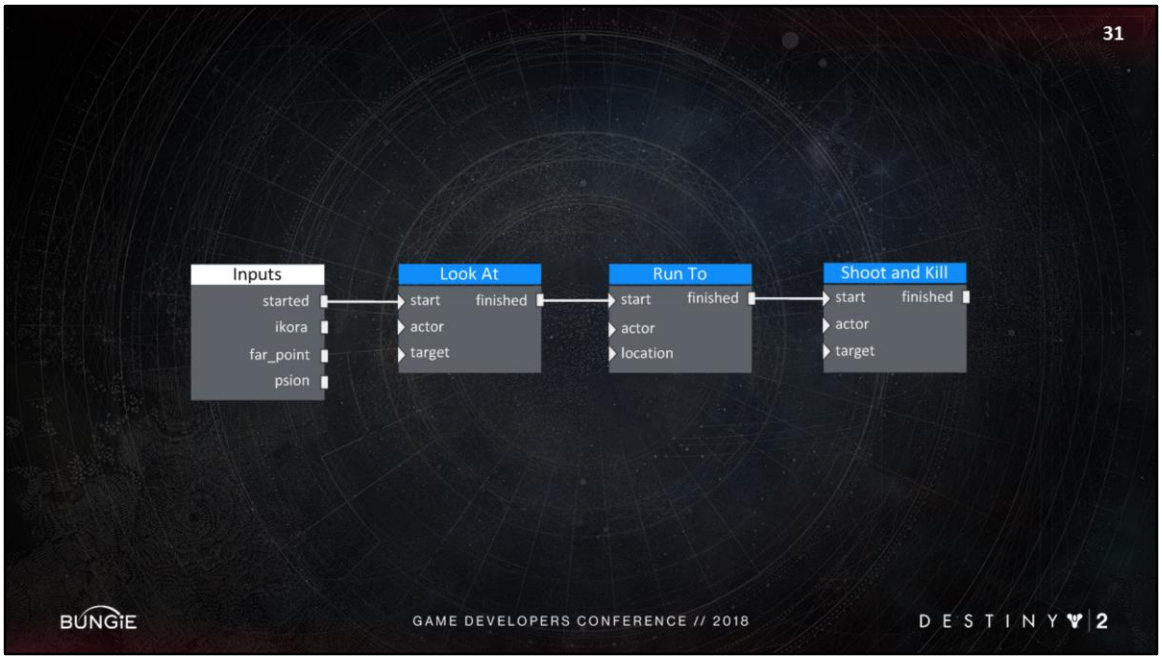
The third and final action is **Shoot and Kill**. It has start and finished, and also has **actor** that's doing shooting, and the **target** that's is being shot and killed.

One quick comment on this node. Notice that **it isn't** the **Shoot and Do A Bunch of Damage** node. No health calculation are involved. It's so easy to have written this in terms of how much damage Ikora can do, and just hope it kills the Psion. Then, later, someone will change the health of the Psion and it will stop working. This action is more semantic and tries to use words to show more clearly the intension of the story.

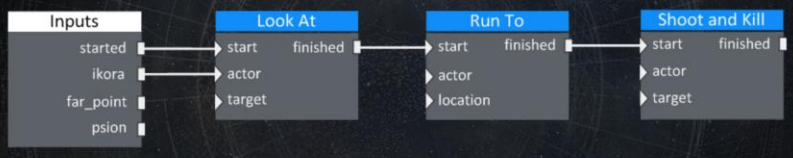
Now let's link our start and finished triggers to ensure these actions happen one after another.

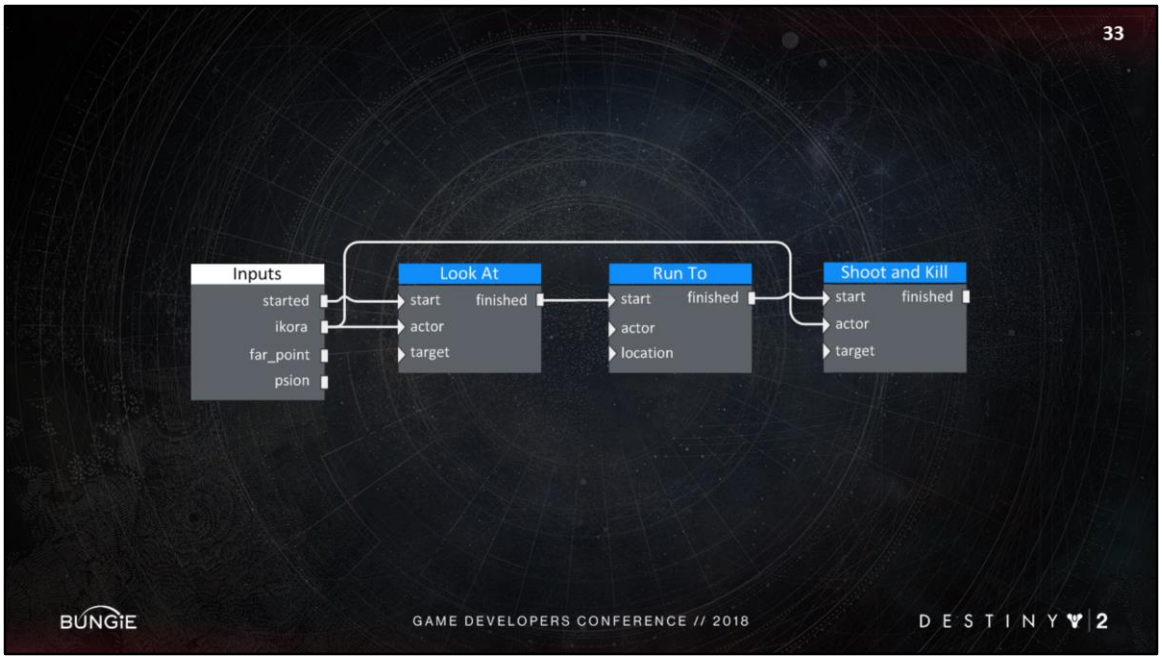

Inputs
started
ikora
far_point
psion
Look At
start
finished
actor
target
Run To
start
finished
actor
location
Shoot and Kill
start
finished
actor
target


Inputs
started
ikora
far_point
psion
Look At
start
finished
actor
target
Run To
start
finished
actor
location
Shoot and Kill
start
finished
actor
target

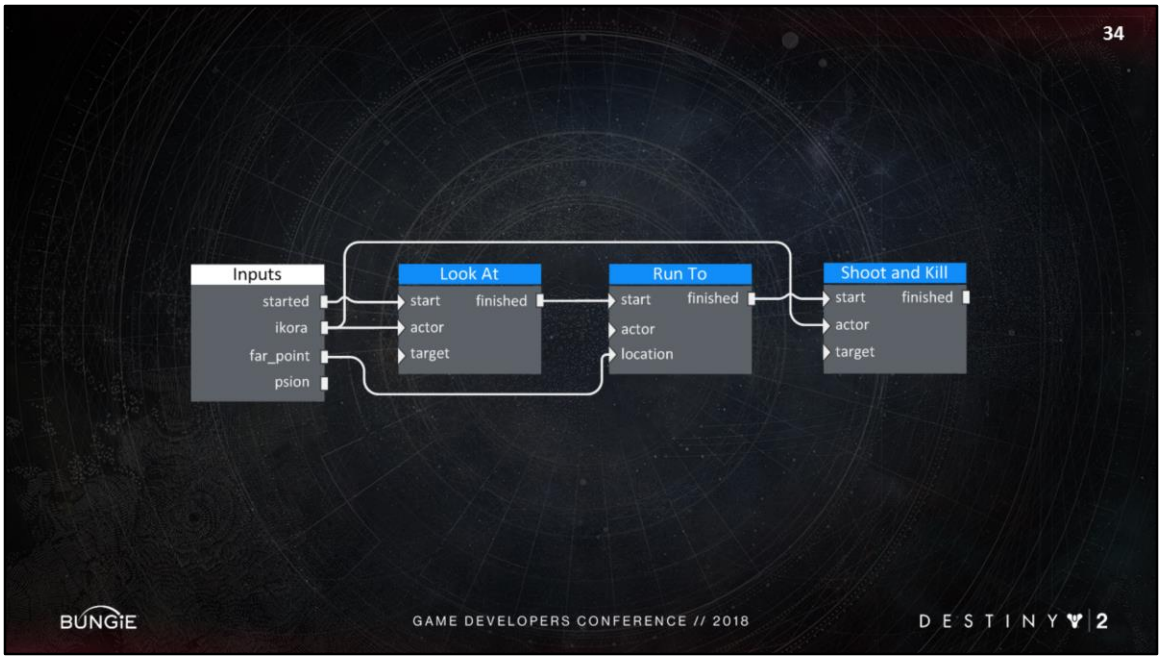


Ok, great. Next let’s link Ikora to our actions.

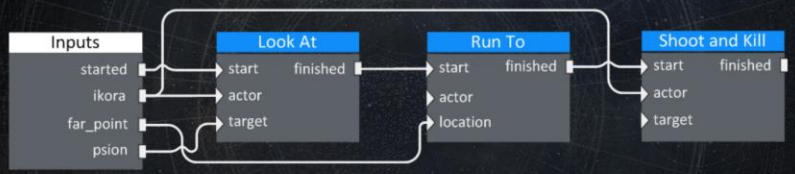

Inputs
started
ikora
far_point
psion
Look At
start
finished
actor
target
Run To
start
finished
actor
location
Shoot and Kill
start
finished
actor
target

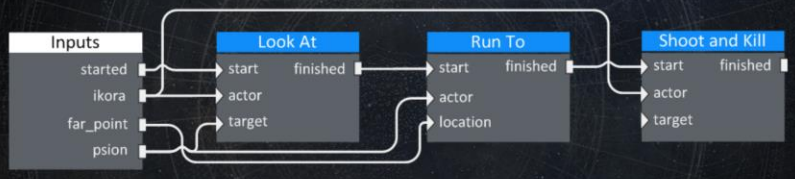


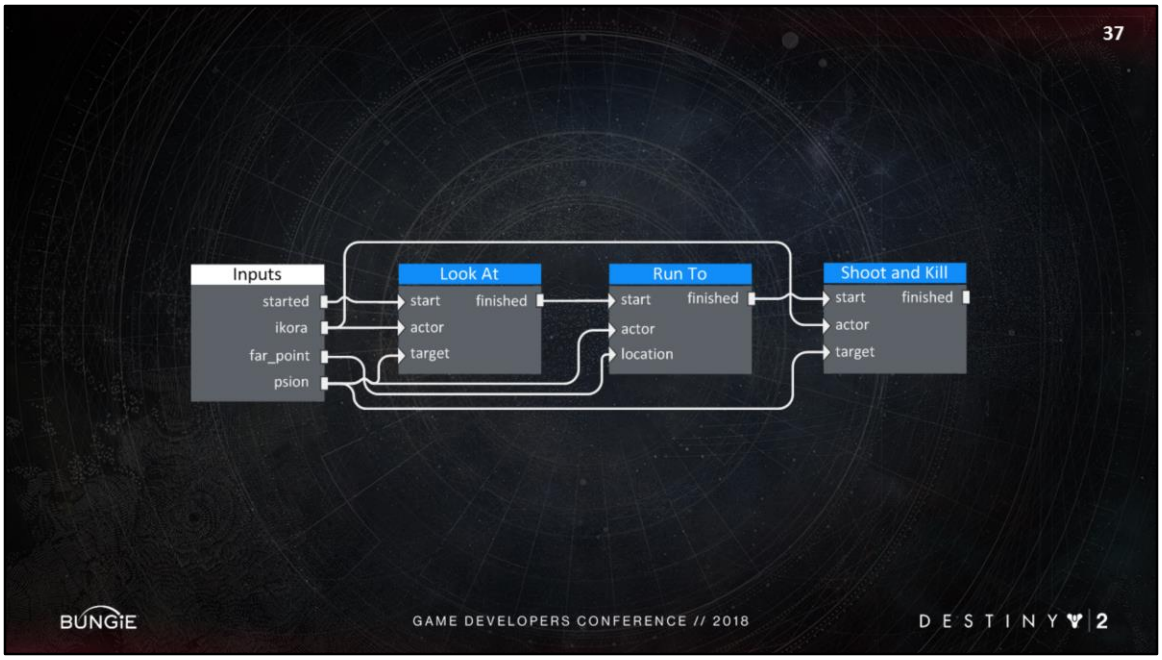
Now, let's link **far point** to the **location** where the **Psion** is running to.



Finally, the Psion is involved with all three actions.


Inputs
started
ikora
far_point
psion
Look At
start
finished
actor
target
Run To
start
finished
actor
location
Shoot and Kill
start
finished
actor
target


Inputs
started
ikora
far_point
psion
Look At
start
finished
actor
target
Run To
start
finished
actor
location
Shoot and Kill
start
finished
actor
target

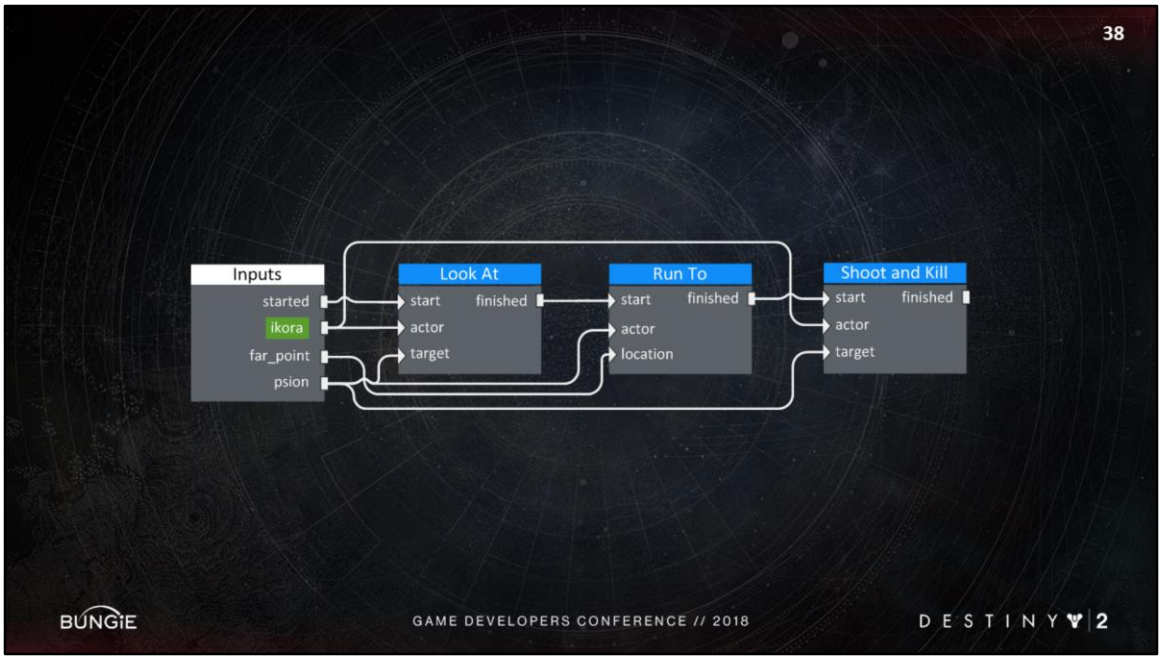


I think you can already see this is starting to become a lot of connection lines!

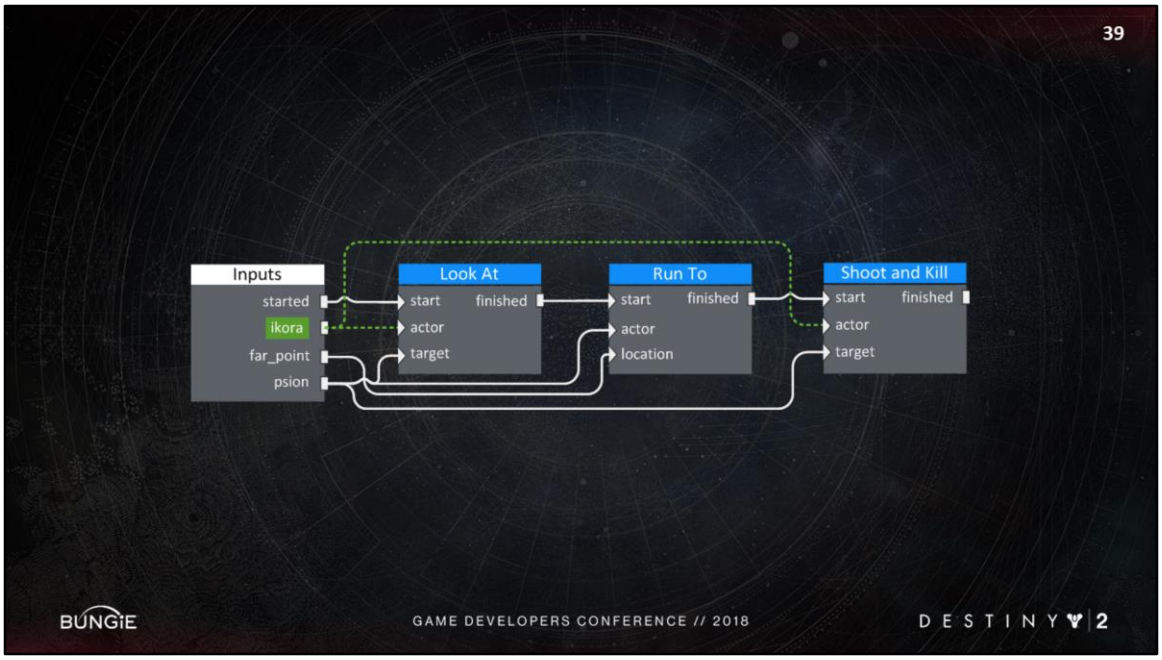
Since we are talking this through step by step the visual script is still clear to you. Often the writer of the visual script can keep everything in mind pretty well. But what if you saw this for the first time. Is it clear? Pretty clear.

Can it be better? Let's try. The first approach we
took to simplify the visual script is naming. We humans are really at language and names.

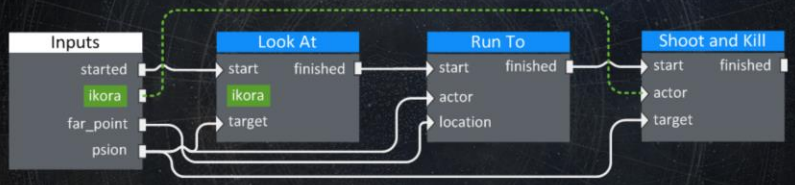
To demonstrate this, let's start by replacing connectors with names for Ikora

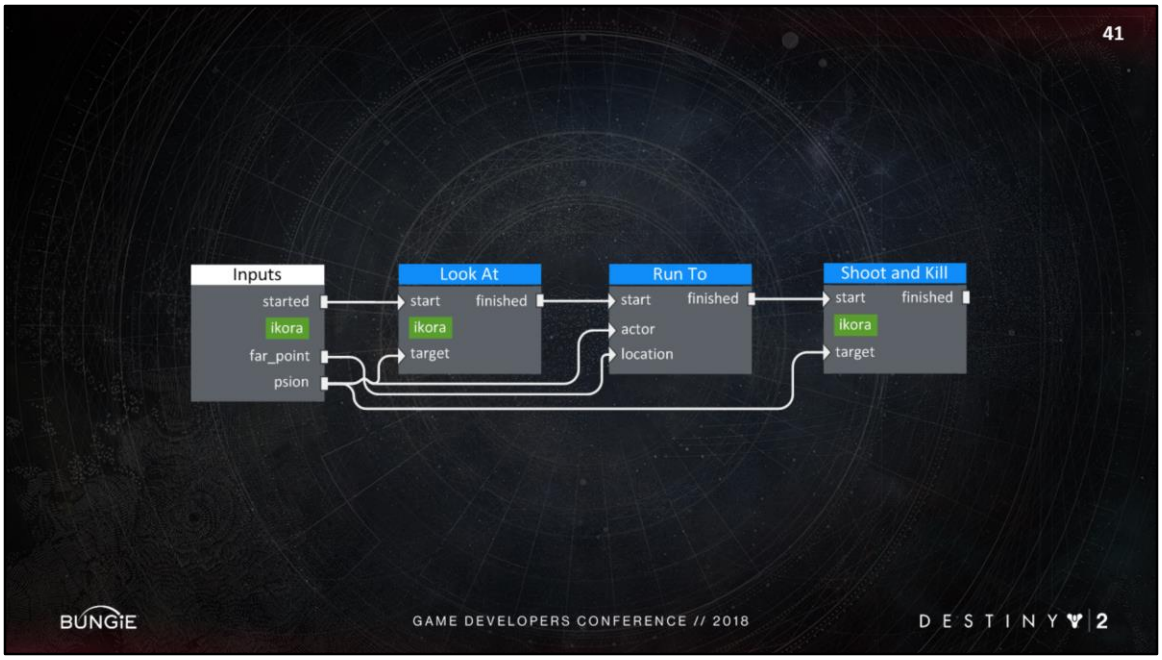


All right, let’s follow the Ikora connection lines and turn them dotted green.

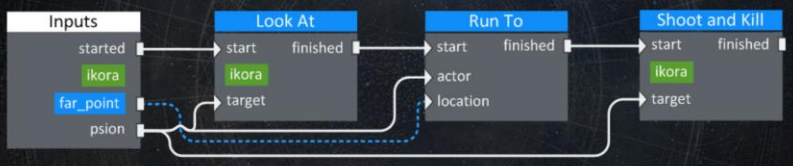


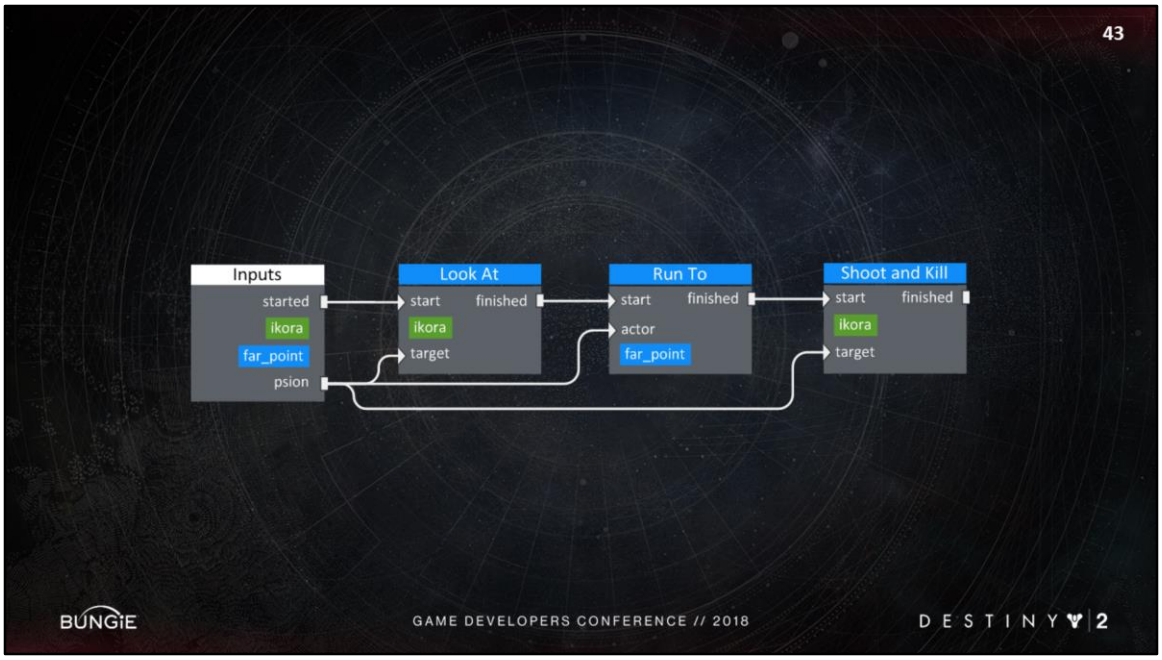
Let’s now use Ikora’s name to reduce the number of connection lines


Inputs
started
ikora
far_point
psion
Look At
start
finished
ikora
target
Run To
start
finished
actor
location
Shoot and Kill
start
finished
actor
target

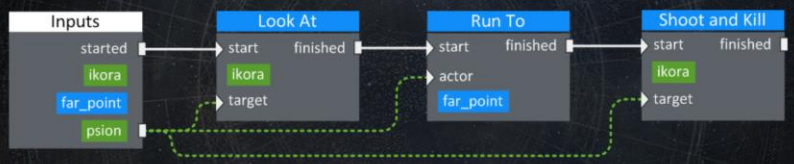


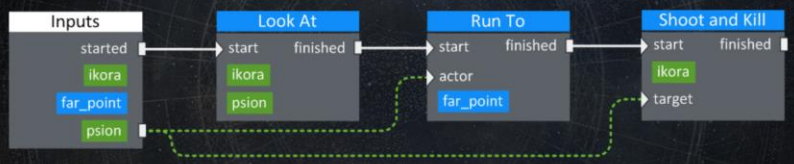
Now, let's do the same for **far_point**.

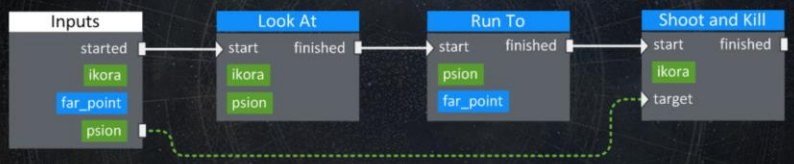

Inputs
started
ikora
far_point
psion
Look At
start
finished
ikora
target
Run To
start
finished
actor
location
Shoot and Kill
start
finished
ikora
target



And now for **Psion**.

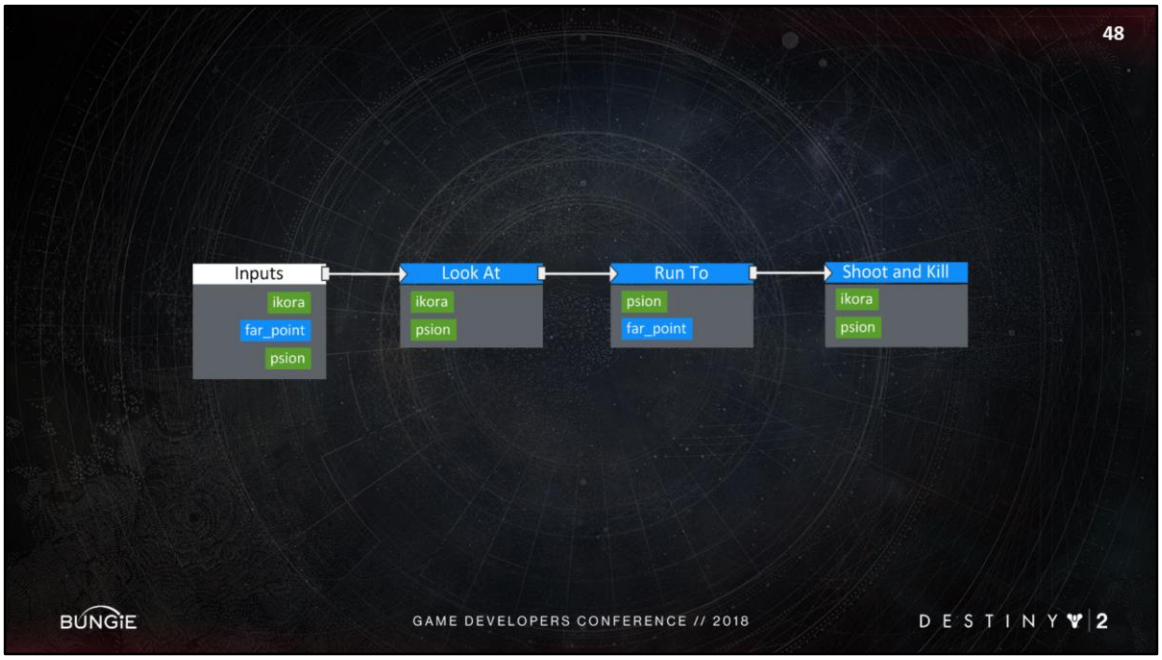

Inputs
started
ikora
far_point
psion
Look At
start
finished
ikora
target
Run To
start
finished
actor
far_point
Shoot and Kill
start
finished
ikora
target


Inputs
started
ikora
far_point
psion
Look At
start
finished
ikora
psion
Run To
start
finished
actor
far_point
Shoot and Kill
start
finished
ikora
target


Inputs
started
ikora
far_point
psion
Look At
start
finished
ikora
psion
Run To
start
finished
psion
far_point
Shoot and Kill
start
finished
ikora
target

Notice how the names are now within the actions they are used. This reduces the number of connection lines significantly.

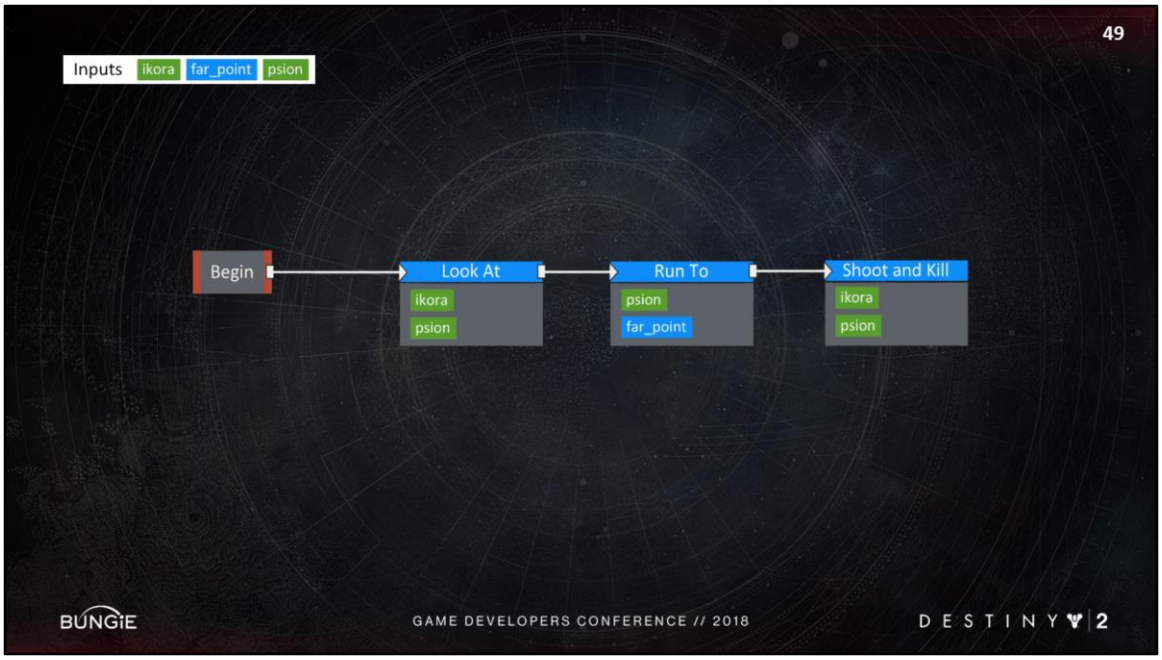
Also, notice the only connectors left are about triggering when nodes **start** and **finish**. This repetition is true for all actions, so because of this similarity we don't need the textual hints anymore.



You may be wondering how to tell the difference between the actor and target, who is launching the projectile and who is… “catching it”. We’ll clear this up in the next section.

(BEAT)

Now Inputs, is a bit different from the other nodes. Let’s pull that out and begin the scene explicitly.

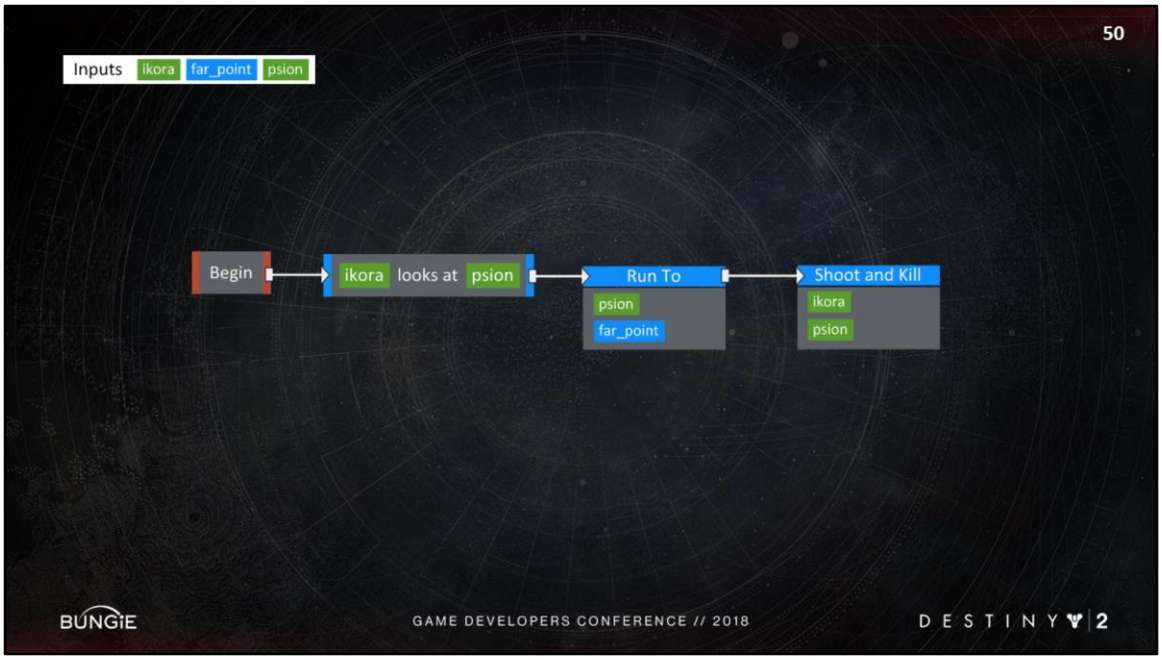


Here you can see the inputs in the top left corner of the screen. These inputs are available for the whole scene, and they use names so that no connection lines are needed between them and action nodes.

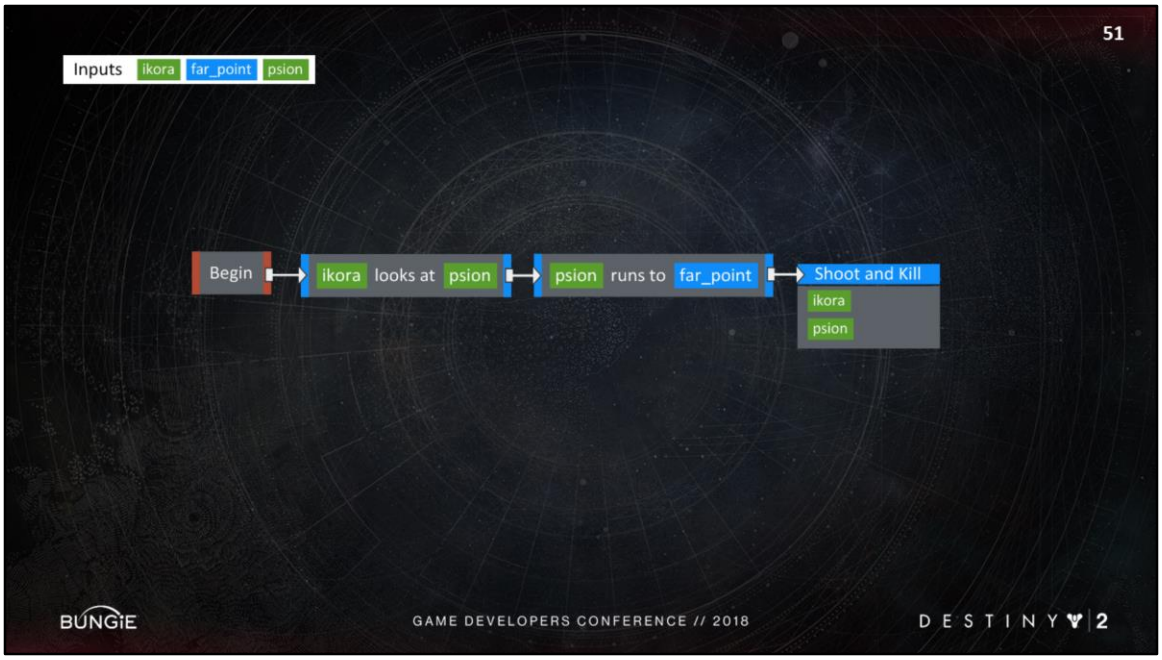
Maybe, it's just me, but as a programmer separating out inputs from their the actions makes me very happy ☺

Now, the begin node is a special action that starts the story. It's red to indicate it's an input trigger instead of a blue action. What is an input trigger from? In all the examples today it's from script, but it could be from another scene. The script is telling the scene to begin.

The next step is to make the actions even more readable. This was done by making them read like sentences.



The story begins and then **Ikora looks at the Psion**.



The **Psion runs to the Far Point**

Ikora shoots and kills the Psion.

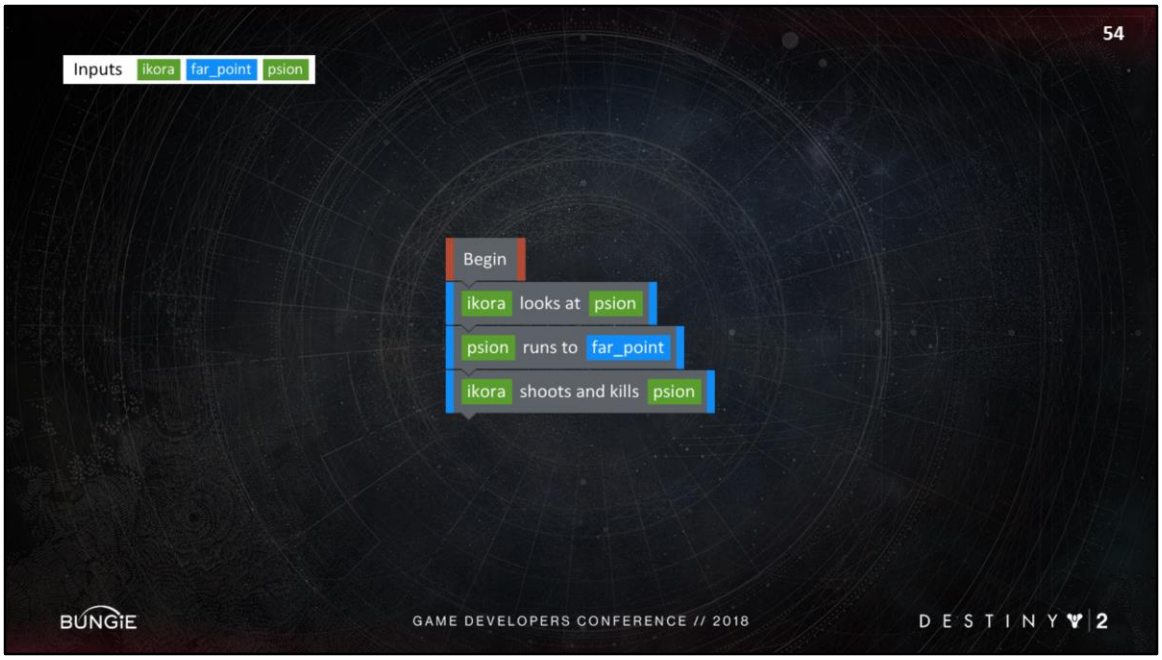
It’s starting to feel like a story!

Ok, now for English speakers we start our words on the left and we flow to the right, so the temptation is to keep adding segments right on and on.

This is totally valid, but in writing usually the screen starts wrapping the text. With this intuition, we took the approach of flowing downwards.

As you can see, by moving downwards it makes the sentences fit better on the screen .
This is because “sentence shaped” nodes are much wider than they are tall, so flowin g downwards will make more of them fit on the screen at once. This also helps make scrolling easier with a mouse
wheel, since it’s much easier to scroll downwards than it is side-to-side.

The next change to improve readability was to add a feature I call “snapping”.



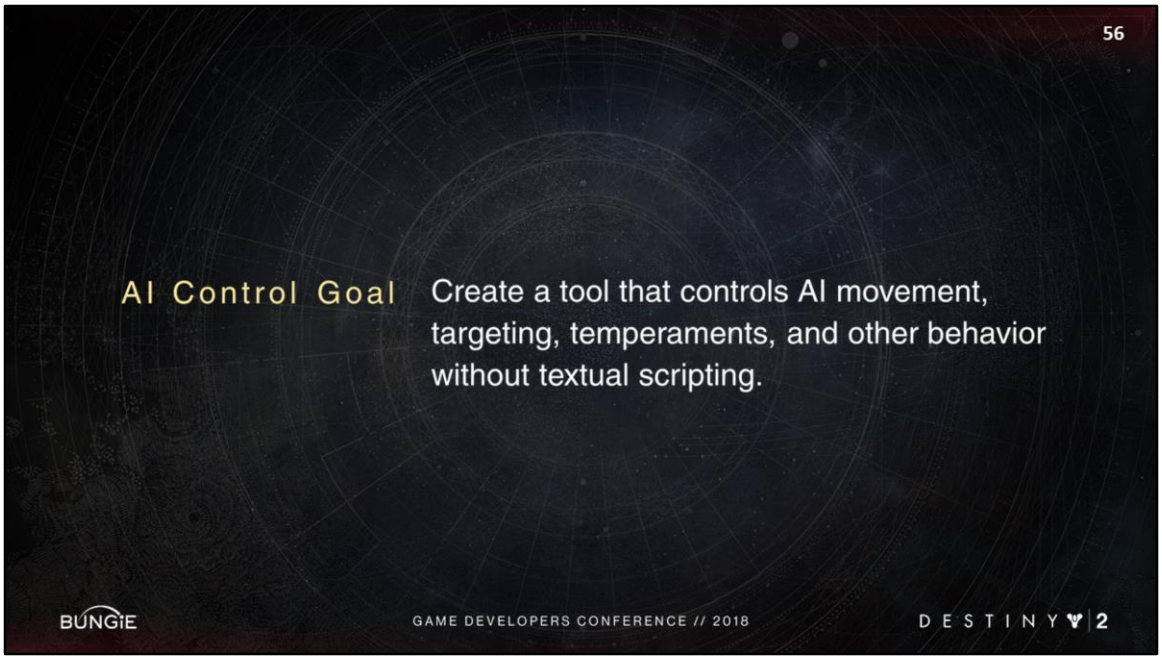
Snapping means that when nodes touch each other they snap together. When snapped, the lower node waits for the higher one to finish automatically. This replaces the start and finished triggers we saw before.

This is an optional feature for people who prefer tighter diagrams.

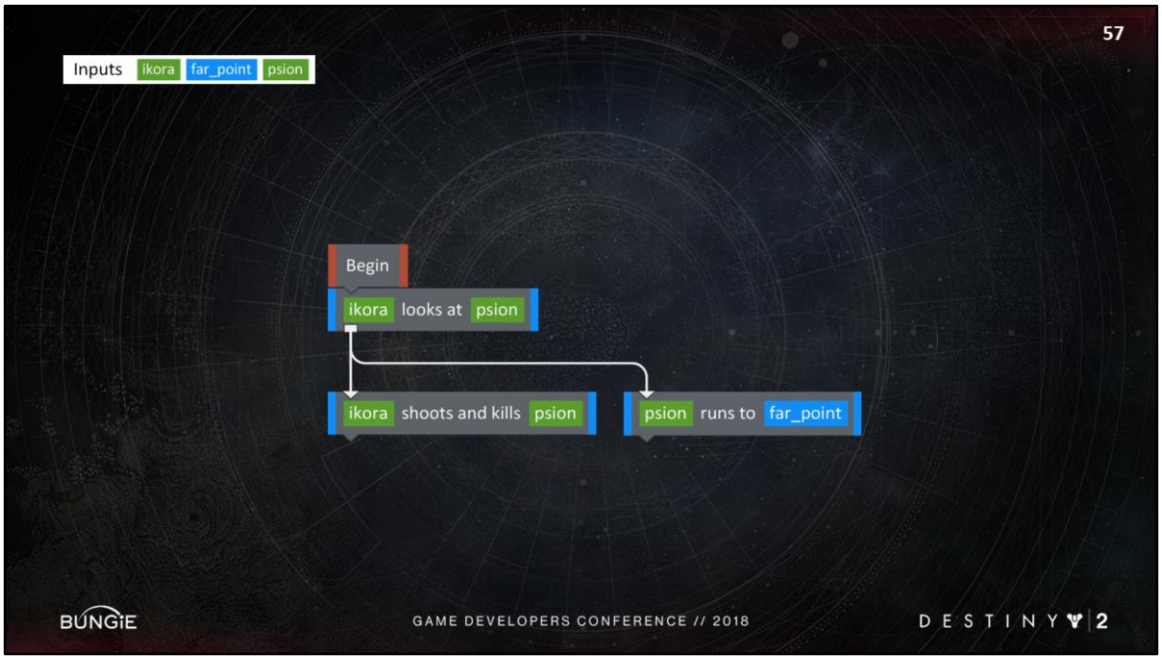
Also, if you drag the top node this helps you drag the entire group.

Those were our approaches to tool design to help make visual scripting more useable .

The next goal up to support high-level AI control.

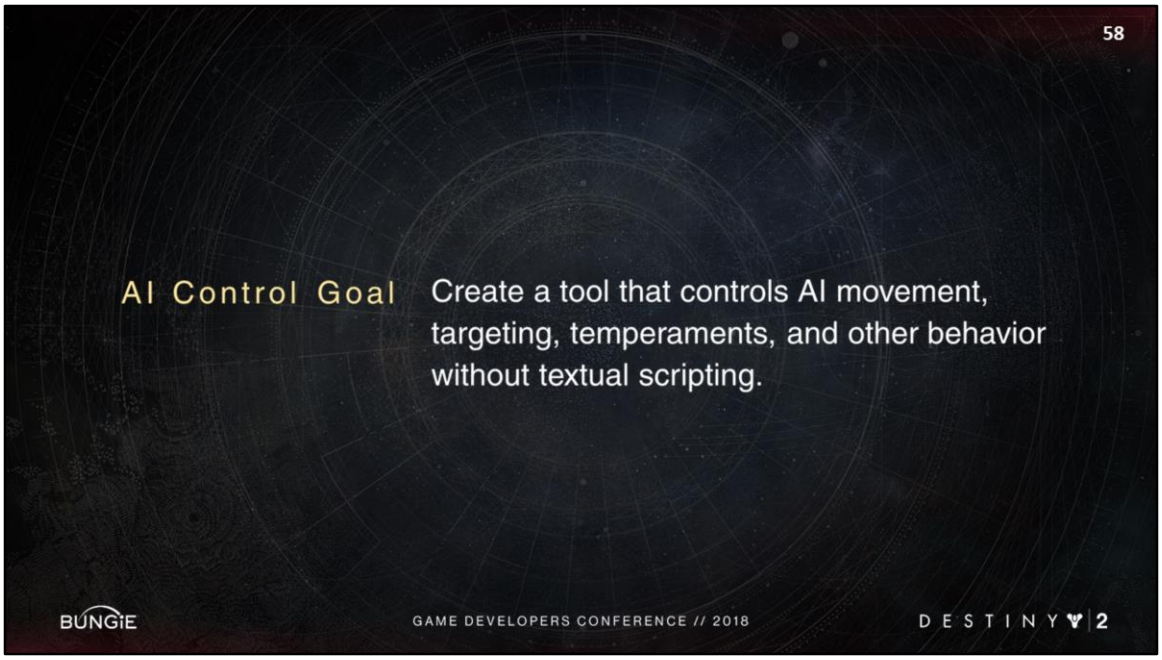


Well, we've been making really good progress at removing connection lines. It turns out connection lines are actually really useful for one thing: visualizing concurrency. NPCs and AI all move independently. Sometimes a good story requires coordinating action by a whole group. If one character misses their cue for their battle cry because they're hidden behind a pillar out of sight, then it's a little anticlimactic.
We need a way to control ALL the characters at once.



In scene visual scripts, lines mean the connecting nodes happen at the same time. Let's make the running and shooting happen at the same time.

So in this case, the Psion starts running towards the `far_point` at the same time as Ikora shoots and kills the Psion in one shot.



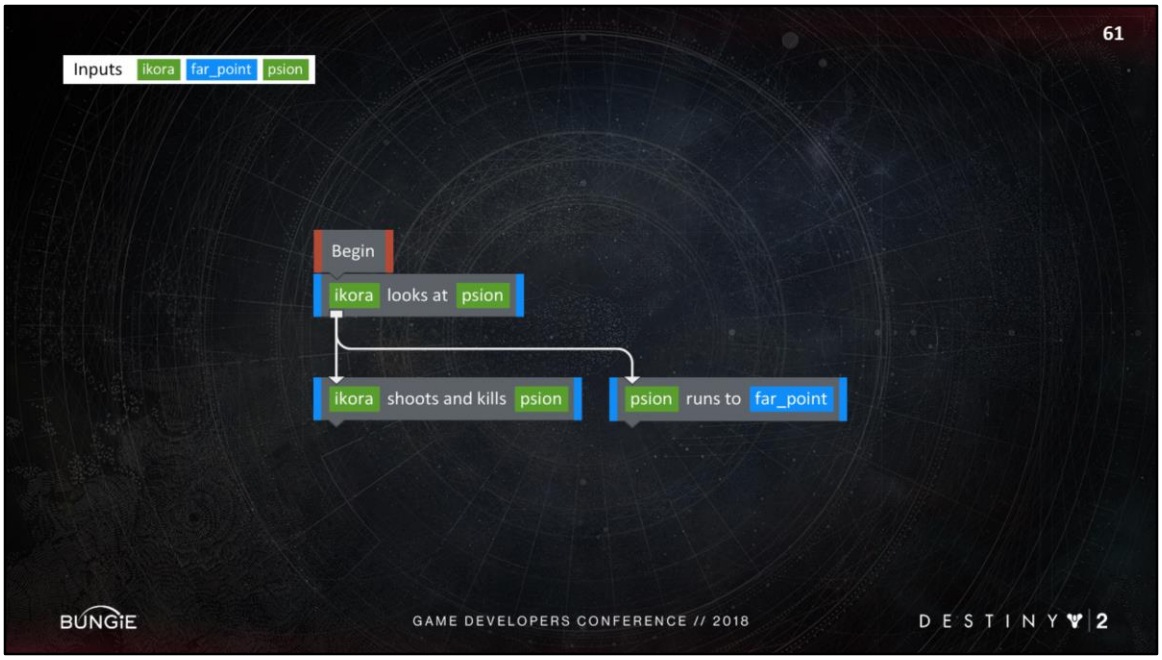
That’s the beginning of how we support AI control with concurrency primitives, but it brings to mind the concurrency challenge again.

Concurrency issues are often quite subtle. When I was first prototyping these sorts of branching scenes, I would show people what the scene might look like to tell different possible stories. Often someone would come up afterwards and say – Hey Evan, did you notice that you kill all the enemies quickly you can both win and lose the mission at the same time? Nope, missed that one. Thank you.

Concurrency is hard for everyone. This is something we really need to address to make tool easy to use.

Next up, let's review our triggering goal.

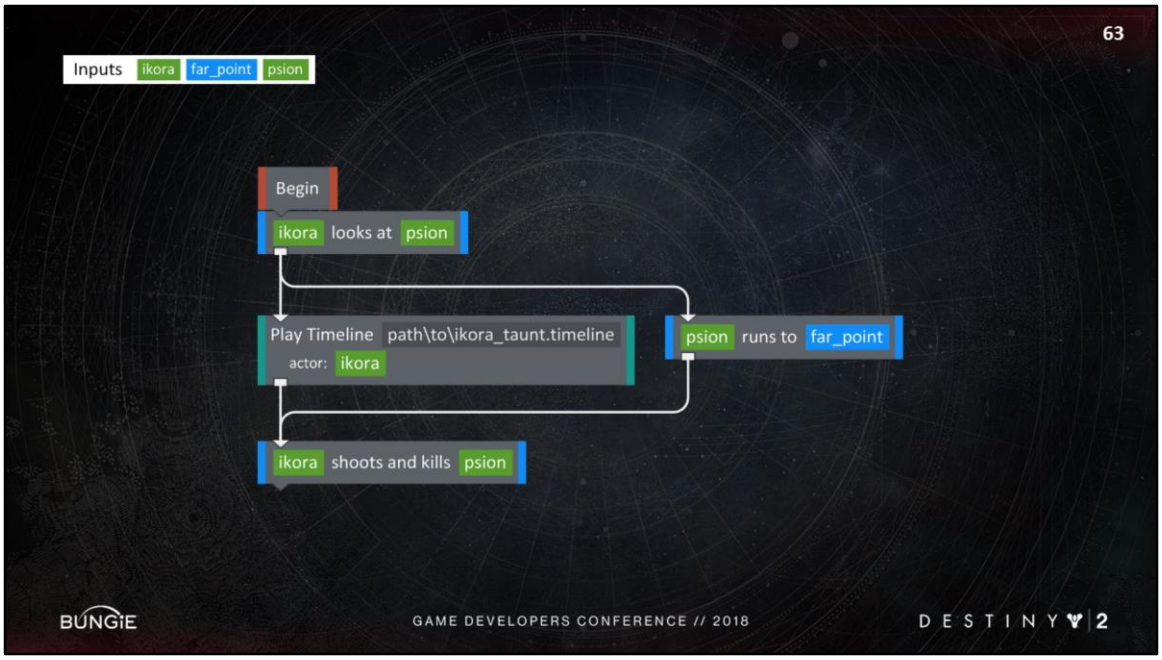
We wanted to trigger different subsystems like dialog and timelines. Let’s see how scenes can do that



And add a node to trigger dialog.

With these connections, Ikora will look at the Psion, and then shoot the Psion when either she finishes saying "I've got my eye on you", or if the Psion reaches the Far Point. This is because connecting lines happen in parallel, and if two connecting lines point to the same node, they'll start whichever node finishes *first*.

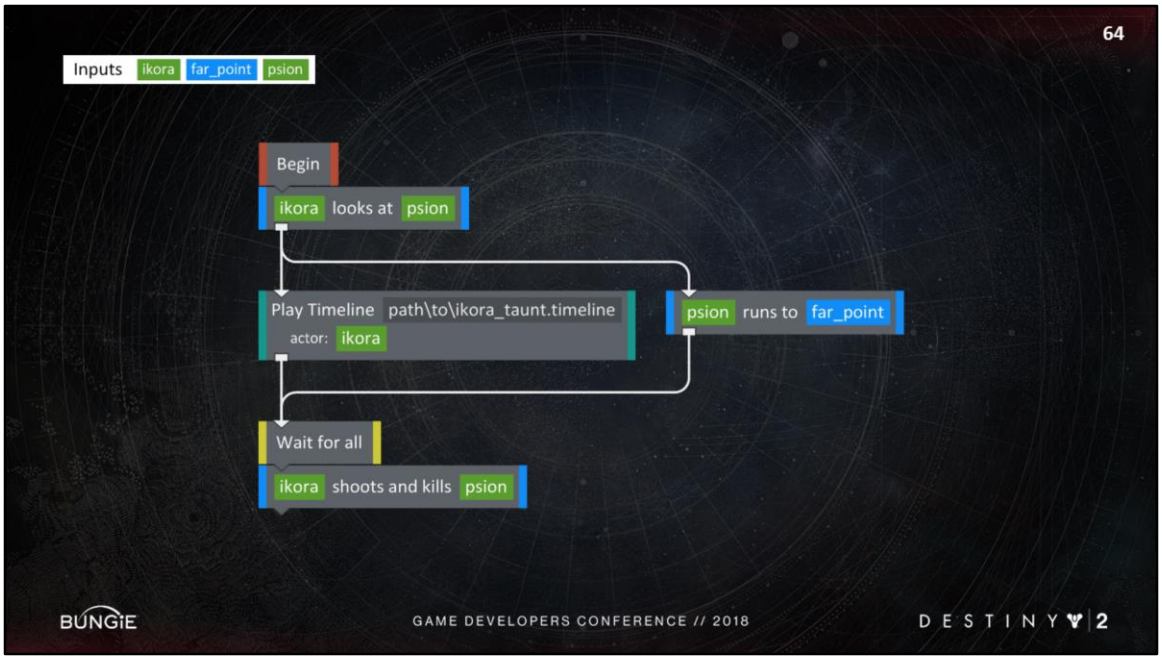
Now one common thing is for storytellers to want to do custom animation during a dialog performance. To do this, we can upgrade dialog into a timeline.



This is the play timeline node. Timelines are triggered just like scenes in that they have named input and output parameters. Since timelines have more inputs then can comfortably fit into a readable sentence, these inputs are displayed as name-value lists.

Notice here that Ikora is passed into the `actor` input of the timeline.

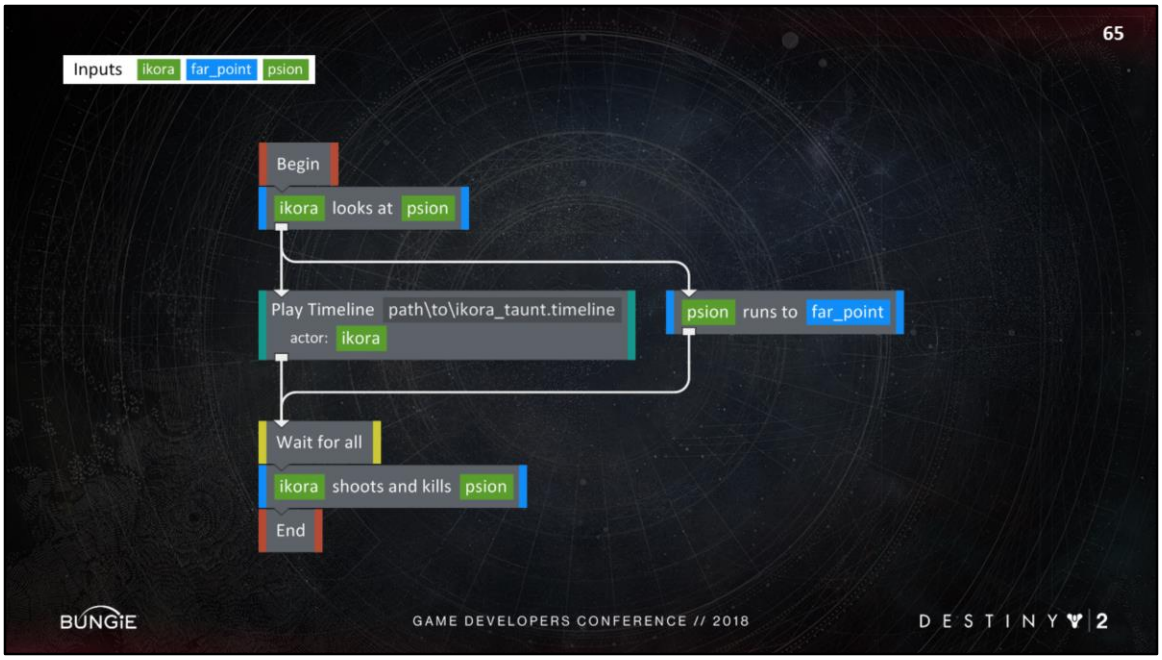
Going back to controlling concurrency. The timeline and the running action both trigger the shoot and kill node. What if we wanted to wait for both of these nodes to complete before the shot is made?



This adds the **Wait for All** node to solve this.
It's yellow because it adjusts the control flow of scene and changes what the graph means. In this
example, Ikora shoots at the Psion when she has finished speaking AND the Psion has reached the Far Point. So she waits for both to finish before she shoots.

So by default connecting likes trigger with OR logic, and with Wait for All nodes that have AND logic.

Other examples of control flow nodes are if nodes for logical choice, random nodes fo r choosing paths based on weighted chance, and delay nodes that wait for a certain time to elapse.



Lastly **End** nodes help the scene clean up similar to C++'s function scoping behavior.

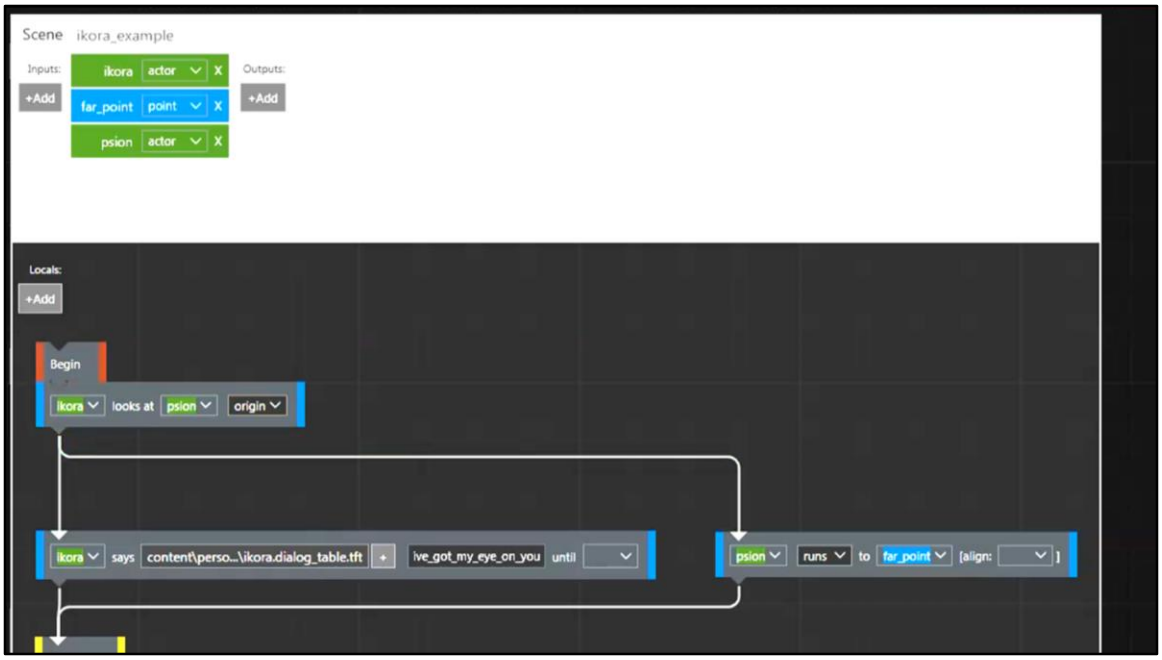
There's always a lot of state to cleanup and we don't want the AI to be permanently looking at a wall or permanently invulnerable. The end node helps us clean this up automatically.

Ok, now that you have a sense of the tool's design, let's see what it looks like in it's current form.

This is the actual tool. The white section at the top defines inputs and outputs. The rest defines the behavior of the scene.

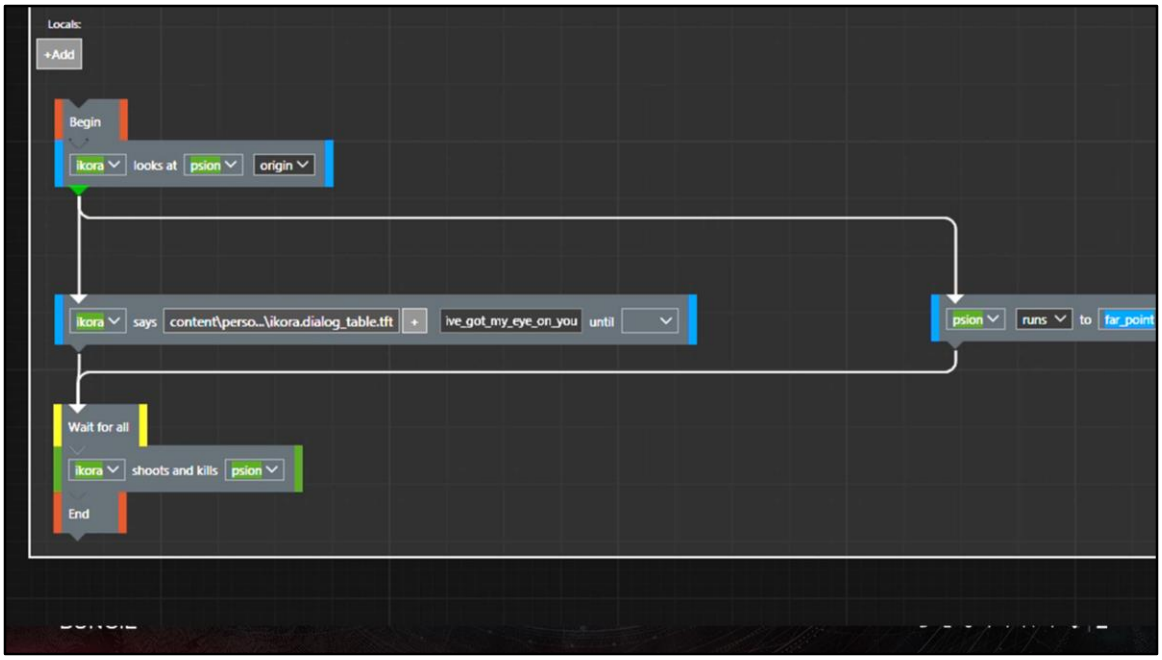
Let's first look at the name inputs to give you a better sense of how they work. What's not obvious is these inputs are all linked together. Let's see what happens as we rename the inputs.

[Video narration]



Next, let's look at how inputs are typesafe. This means that the action nodes only allow inputs of the correct type in their dropdown. Let's see how this is enforced by the pulldowns on each node.

[Video narration]



Let's do an example that involves waiting. Let's wait until the Psion is killed, and then have Ikora **holster her weapon**. This means we need a new type of node, called a When node. Purple when nodes wait until game state is reached, in this case when a combatant is killed

[Video narration]

One cool thing here is that if a player or anyone else kills the psion, Ikora will still holster her weapon.

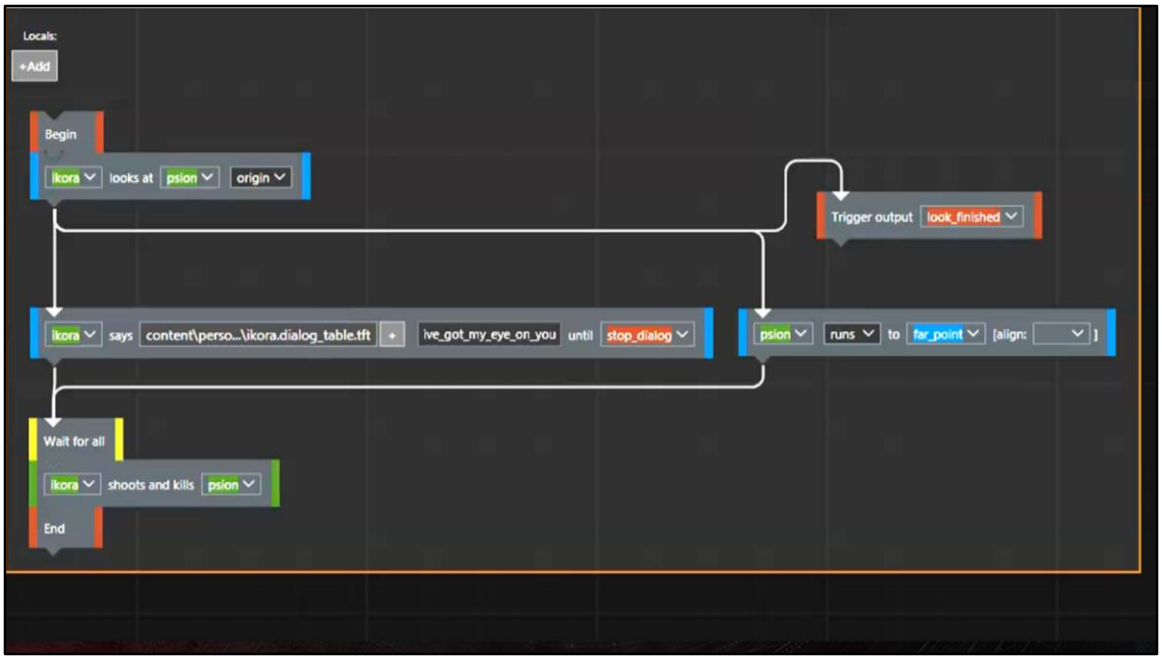
Next, let's talk about the different types of triggers within scenes. This first is an input trigger from script. In this example, script wants to interrupt Ikora's dialog line

[Video narration]

Long running nodes like dialog and looping animation have this **until-style** sentence structure that allows them to be interrupted. This is another way of reducing connecting lines so the end of stories don't have a ton of visual connectors to the beginning of the story.

Let’s continue to explore triggers by introducing an output trigger. Output triggers are used to signal from scenes to text script

[Video narration]



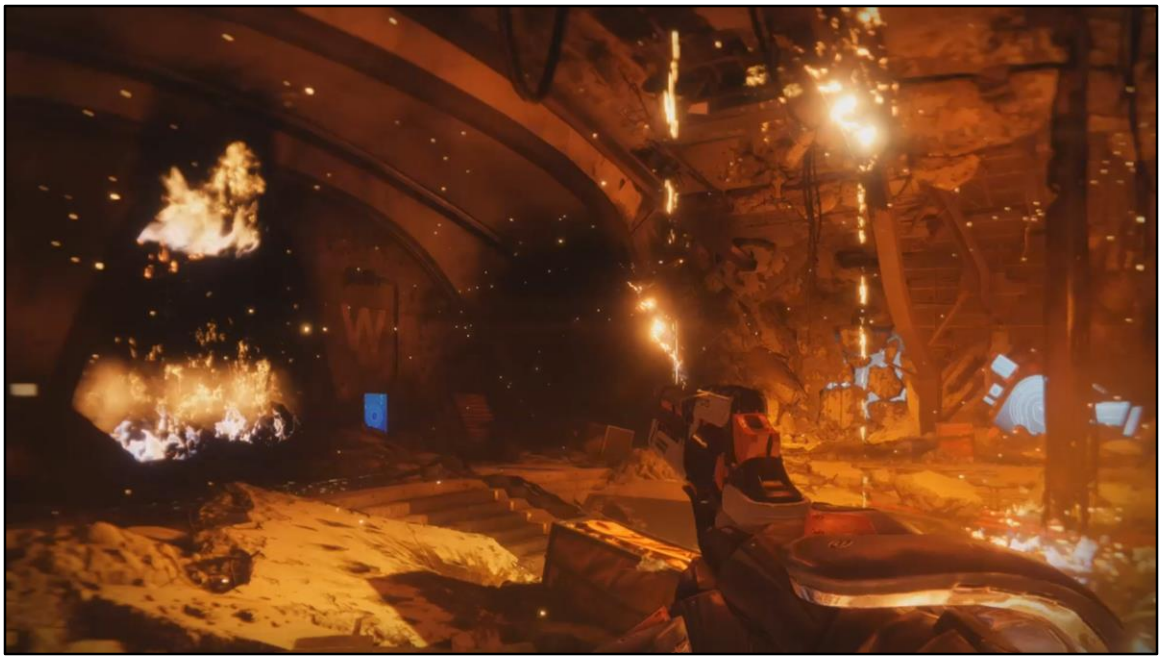
In this final example, let's learn about local triggers. Local triggers allow you to name triggers that are internal to the scene and not shared with script. These are meant to simplify the scene through naming. Let's see it in action.

[Video narration]

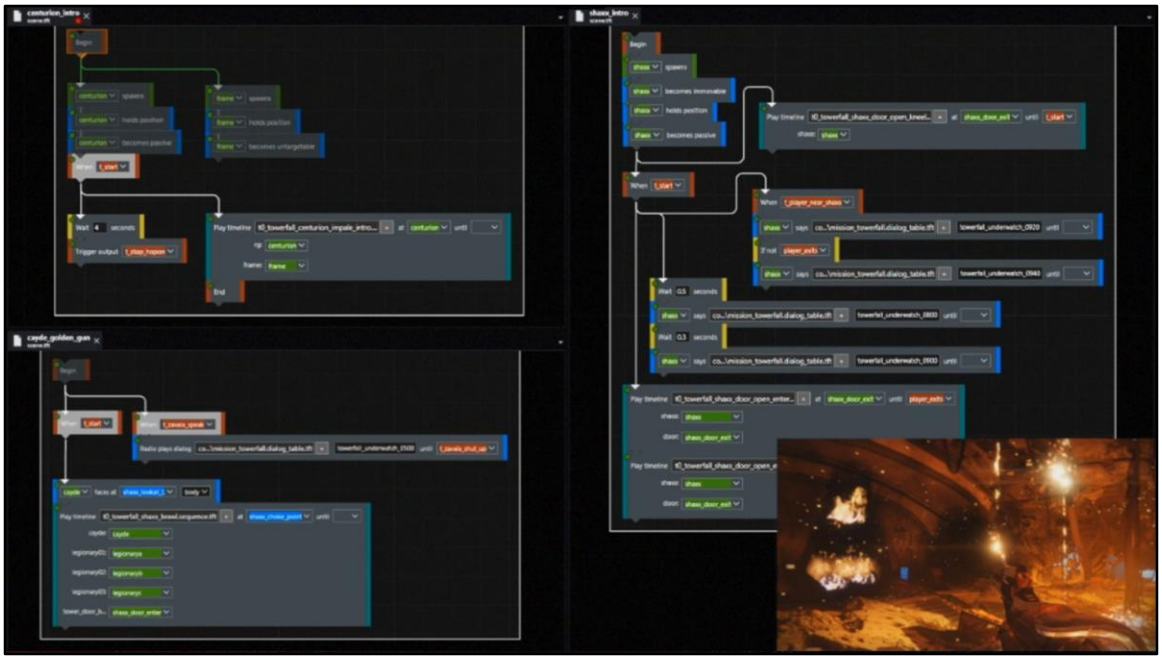
This is used for clarity. Sometimes the connecting lines get too detailed and a naming the concept like **player_arrives** or **dropship_departs**, is easier to understand then traversing through all the connectors.

Ok, enough triggering. Time for some projectiles!

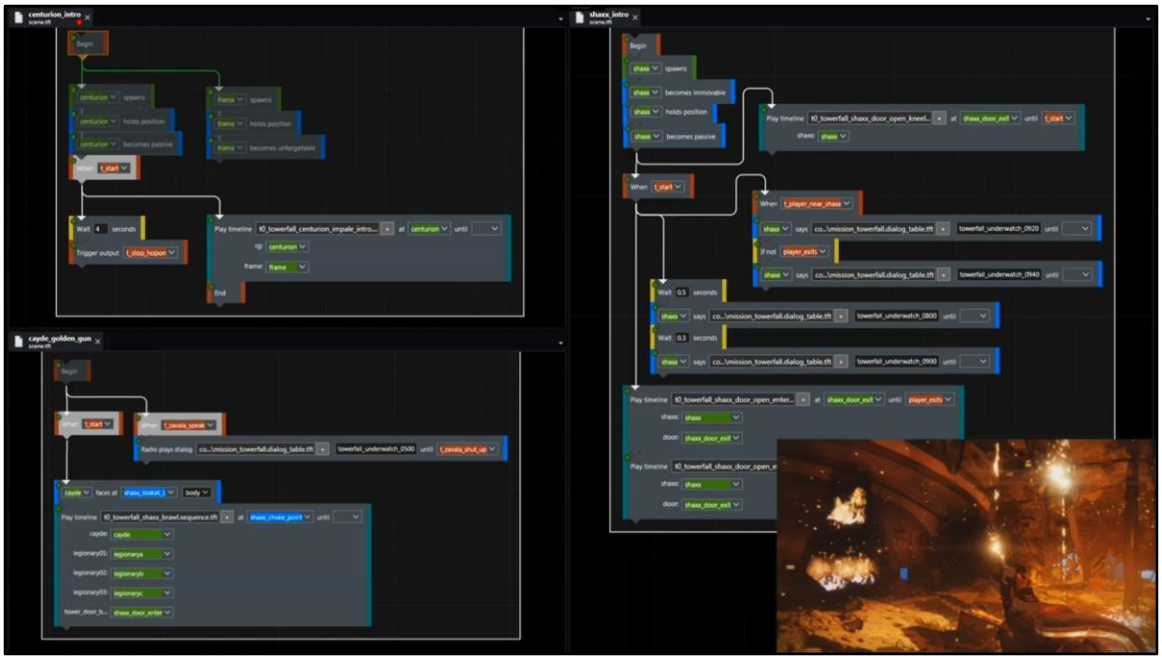
In this game demonstration we'll see three scenes that play out during the Homecoming mission of Destiny 2. These play one after another, so I'll first show the video and then I'll show the video with the scene tool live highlighting as it plays.



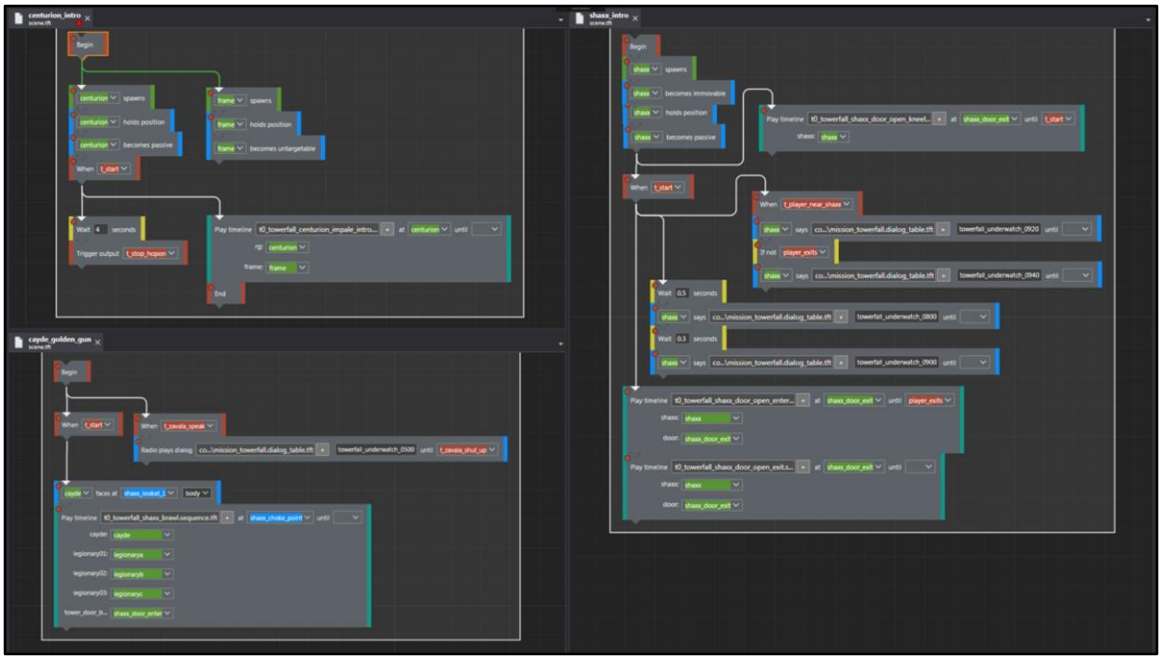
[Video narration]



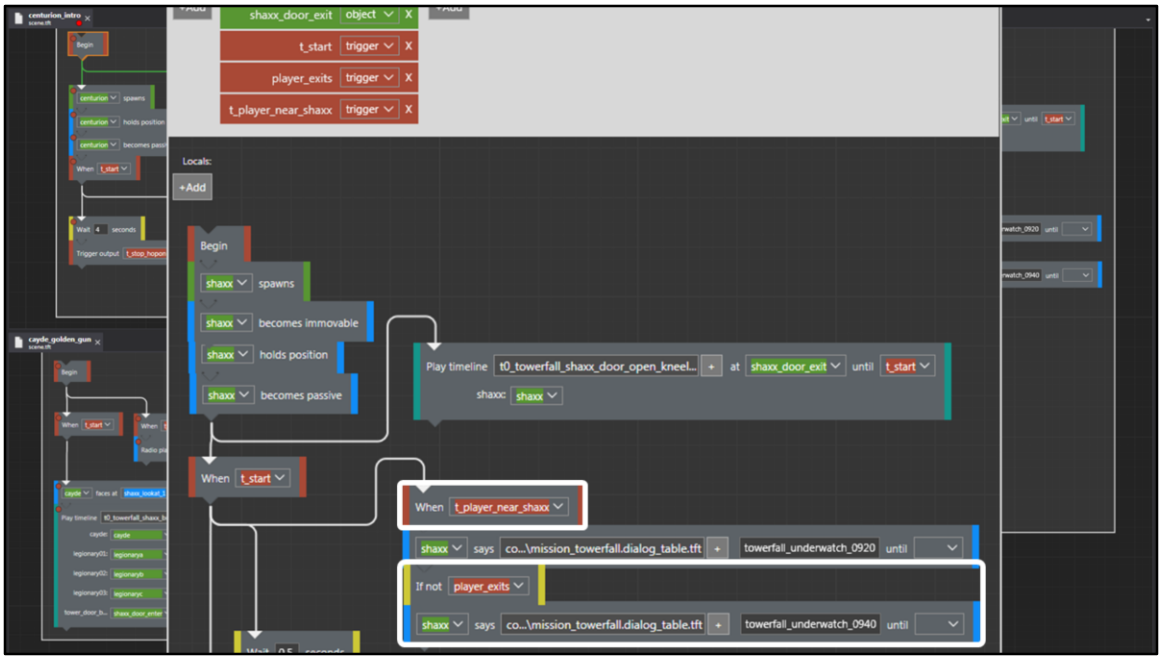
And now let’s see those three scenes with live highlighting turned on. The top-left scene shows the centurion grab the frame. The bottom-left scene shows Cayde shooting. And the right scene shows Shaxx opening the door.



[Video narration]



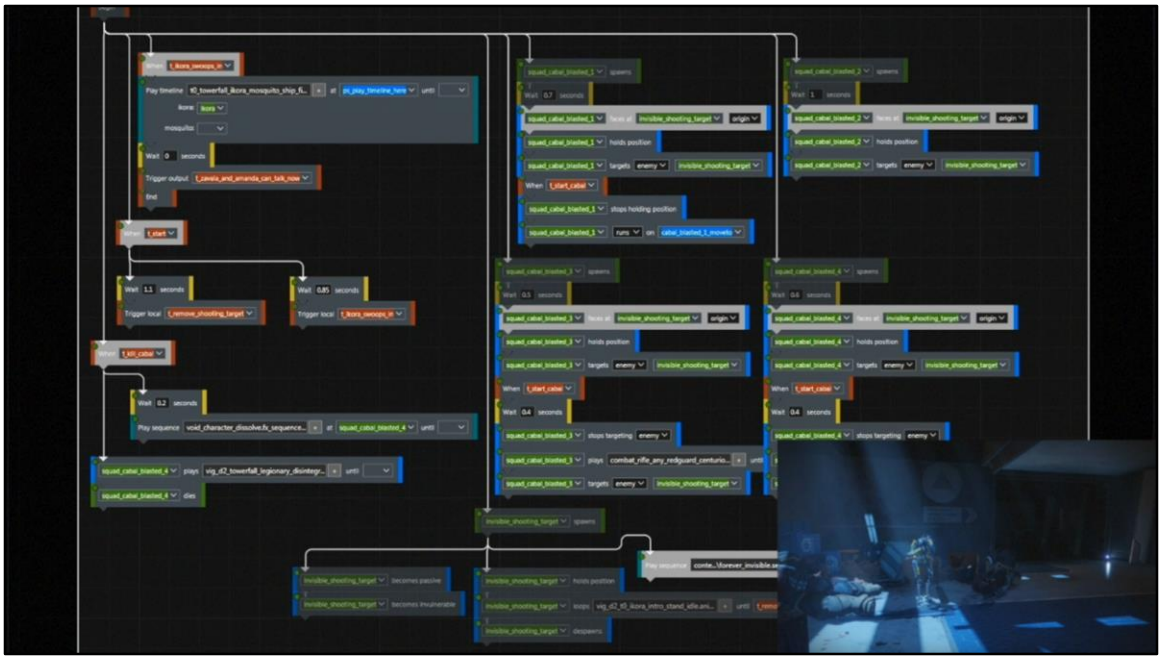
Ok, let’s zoom in and investigate the Shaxx introduction scene on the right.



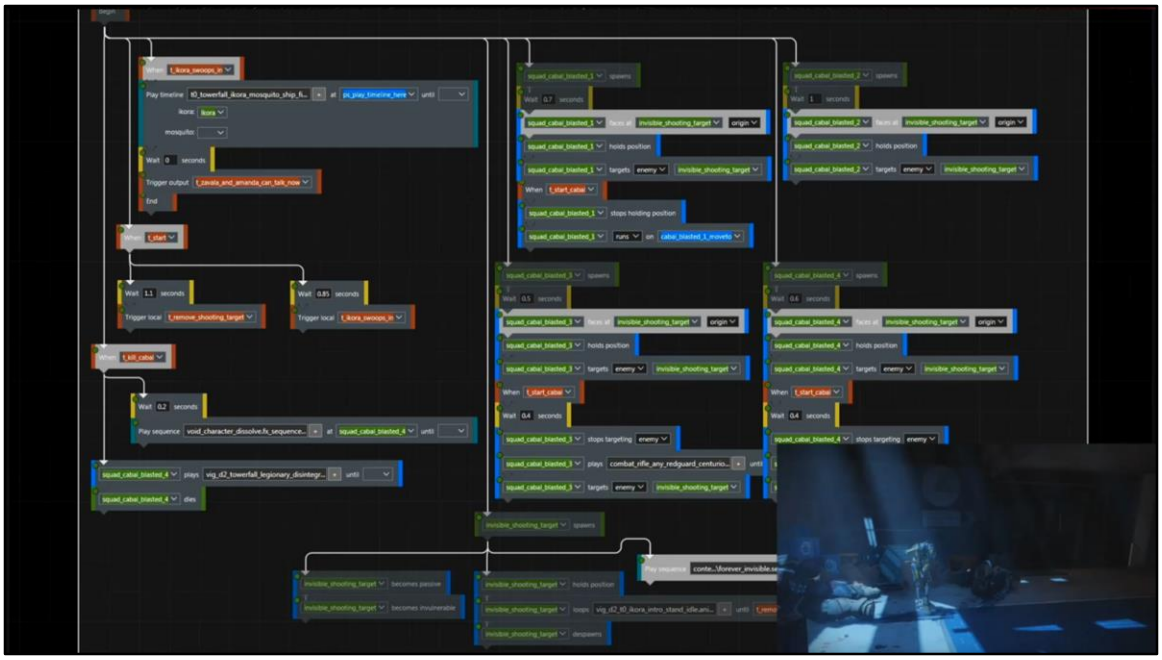
Notice this part which is waiting for script to tell the scene that the player is near Shaxx. It also, optionally plays this dialog line if you stay in the room a long time. This sort of optional performance help us respond to the player's actions and makes the story come alive around you

In this next demonstration we'll see Ikora taking out some centurions with a nova bomb. Again, we'll watch the video first, and then we'll see the scene play out after.

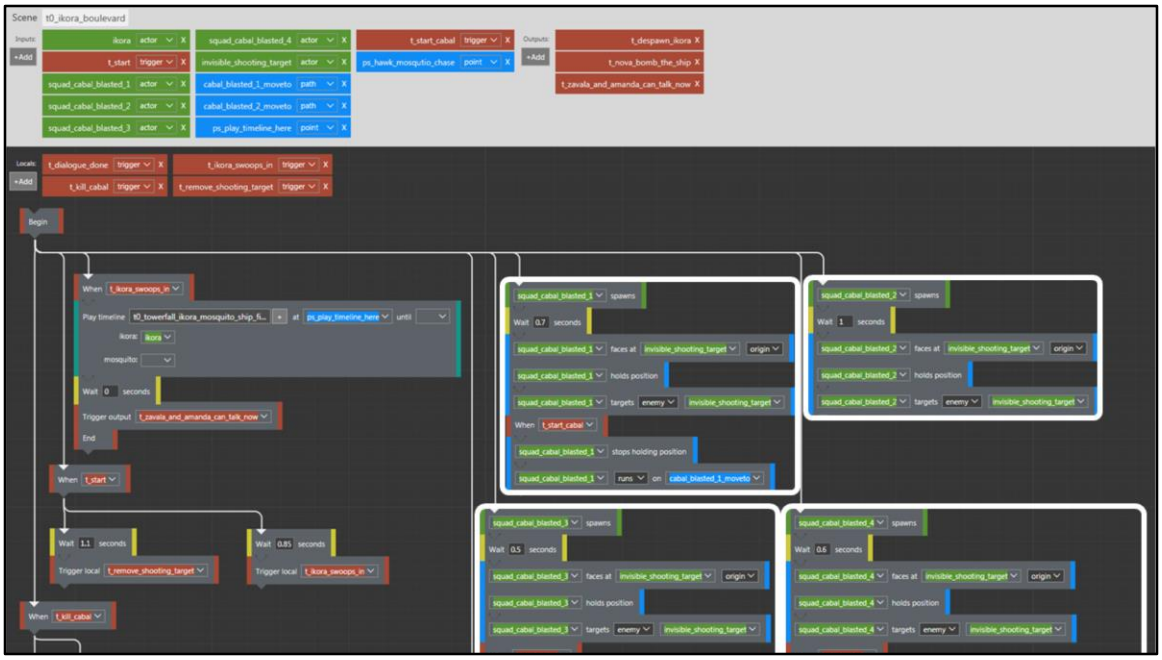
[Video narration]



And now with scene live highlighting.



[Video narration]



Notice these four groupings all start with green spawning nodes. They are responsible for creating the enemies you see as you come around the corner. If you recall in the video these nodes don't stay highlighted long. This is because the enemies are killed while these nodes are running.

This highlights interesting design question. What should happen when a combatant is killed during the playback of a scene? Should the story stop there, or continue on? We tried both, but it became clear that scene needed to continue on. So if a combatant is running and gets killed, the scene pretends they reached their destination and continues on.

This part of the Ikora scene is utilizing this effect. Essentialy it's queueing up for each combatant, but if they are killed by Ikora or the player too soon, those will be skipped over.

The other interesting part about the Ikora introduction scene is it's quite hard to get players to look in the right direction! We could craft the most compelling story moment in the world, but it doesn't matter if the player's checking out the moss on the wall, or looking off into the sunset.

In this scene we use several different tricks to help ensure the player is looking in the right direction. This area uses lighting, 3d positioned audio cues, and a narrow corridor to help focus the player's attention. This way their eyes are drawn to where the action is about to take place.

In this last game example, Zavala helps you defend the Tower plaza by protecting you with his Titan, Ward of Dawn shield.

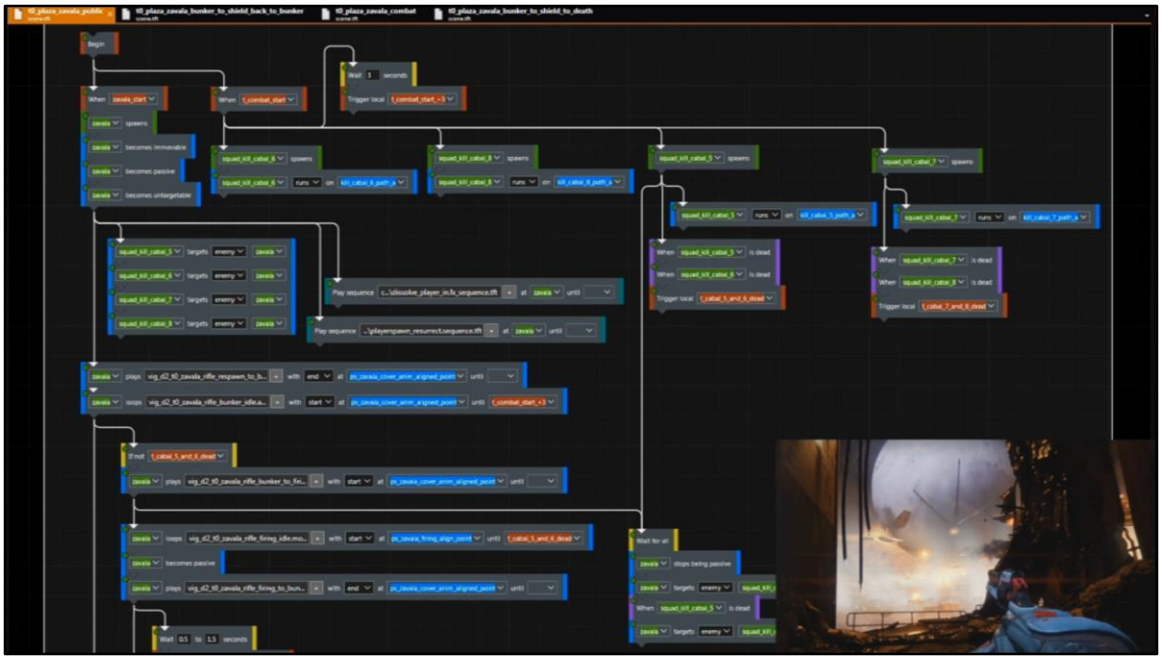
This video is a bit longer, so let’s start it and I’ll narrate as we go.

Video Narration: You're entering a public area where other Guardians may show up and assist. The tower spire is destroyed. Drop pods of unknown origin are falling from the sky. Four cabal are running up the center of the plaza attacking Zavala. Zavala is shooting them. He takes cover. A barrage of missiles comes from a far off enemy ship and Zavala defends you with his Ward of Dawn shield.

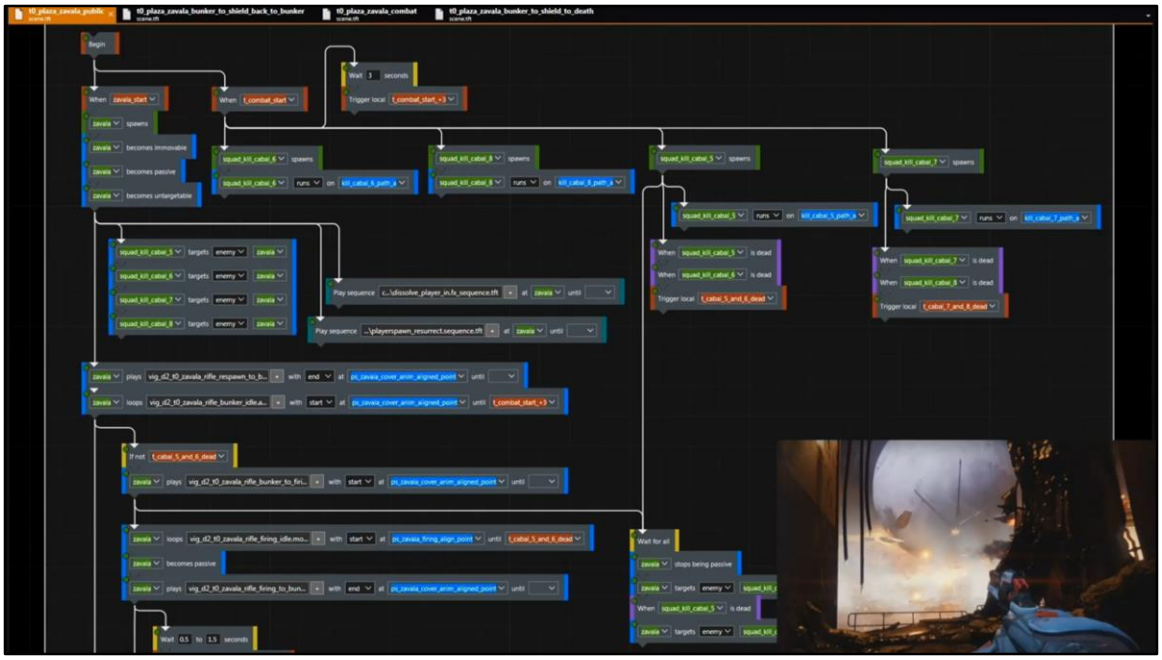
The battle resumes and Zavala takes cover again. Zavala leaves cover and shoots. You take out more enemies. Zavala defends you again with his Ward of Dawn.

(BEAT) This time the last shot of the rocket barrage sneaks past and kills Zavala.

After some time passes, Zavala is respawned by the light of the traveler and the story continues for the next set of guardians coming through.



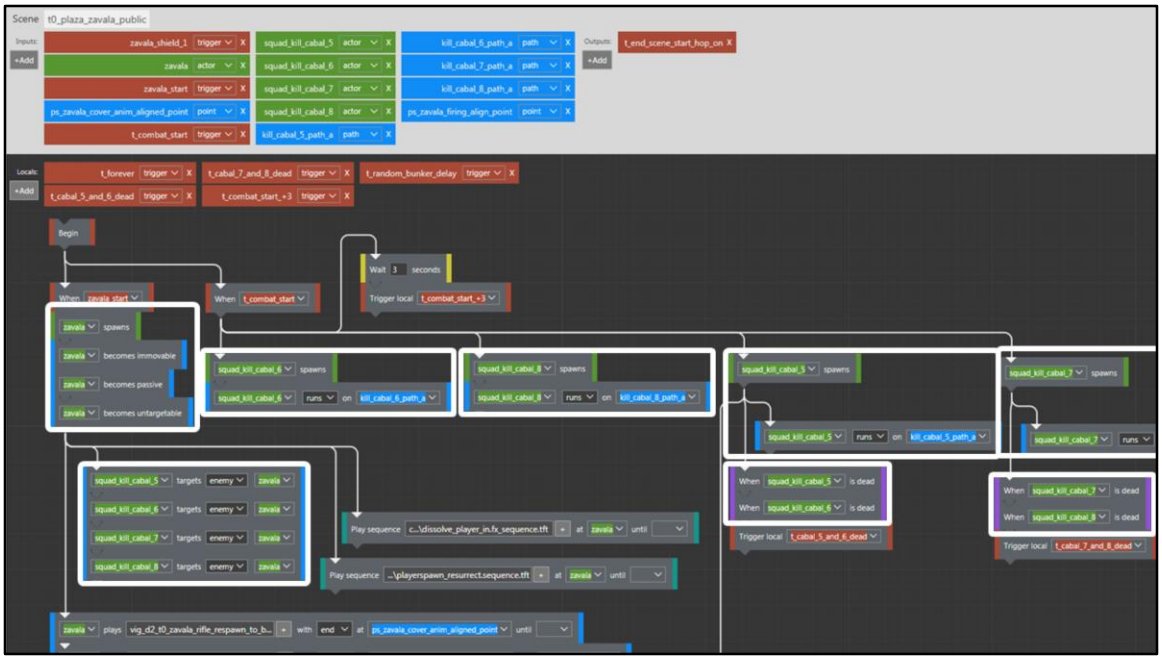
Now let’s see it with scene highlighting as I explain how the scene works.



There isn't just one scene, there are actually four scenes. Each are being turned on and off by script logic. The first scene spawns Zavala and the four cabal enemies. The scene tells the four enemies to run up the center of the plaza towards a Zavala. In the upper right corner the purple when nodes are waiting for each enemy to die. If the player shoots these enemies the scene progresses just the same the if Zavala shoots them. When the cabal are killed Zavala takes cover with a looping animation controlled by the scene.

The next scene starts causing a timeline to be triggered that takes two arguments, Zavala, to his animation and his Ward of Dawn, and the missile spawner, to control the missiles and their detonations.

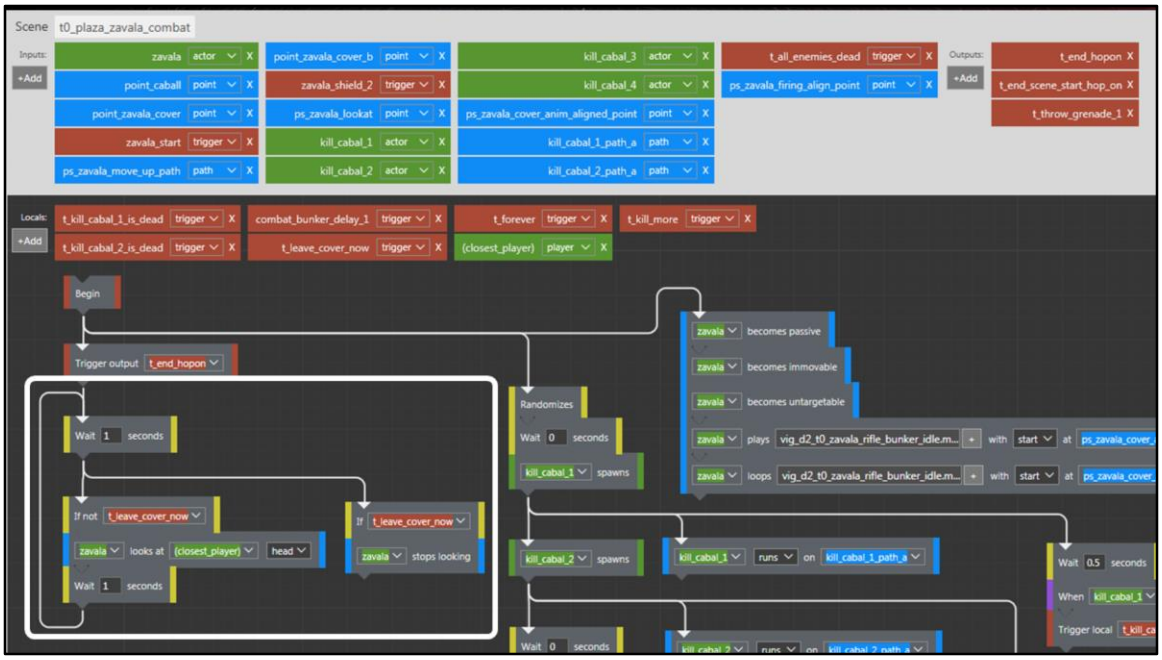
The next scene spawns four more cabal as Zavala resumes combat. Zavala targets and shoots them as before. The script logic is taking complete control progression and will tell the scene with a input trigger when all the enemies are dead. The last scene transitions to Zavala's death. The timeline again controls the Ward of Dawn and missiles. After a short delay the script restarts the first scene that respawns Zavala and begins the story for the next set of players.



This is the first scene we just saw a bit bigger. Zavala spawns here, and he immediately targets the enemies here. The enemies spawn and run towards Zavala here.

Now notice this also uses the purple When node to wait for gameplay state. In this case “When each cabal is dead” the node finishes and continue on.

Lastly, this demonstrates that the scene is meant to be easy to understand at a glance. Your eyes are meant to scan to the red or purple nodes
where the scene is most likely waiting to progress on script triggers or gameplay triggers respectively. Because of this you can think of a scene as a series of waiting queries followed by a grouping of actions that happen in that moment in time.



This is the other complex Zavala scene. Let me highlight one interesting part before I move on. This part of the scene actually loops back on itself. Cycles are allowed within scenes with the limitation that triggers can only be fired once. And that is to help with potential concurrency issues.

In this case the cycle is being used to have Zavala look at the closest player periodically. Then, when red local trigger indicates it's time to leave cover, Zavala stops looking at anyone.

Overall, these Zavala scenes were some of the most complex scenes made in Destiny 2. It was located in an area crowded with guardians, so it tested the limits of scenes ability to handle network synchronization. This scene also had an unlimited duration, it so it needed to restart and loop smoothly. It also was split across multiple scenes which made preserving animation positioning such as being in cover more challenging.

This is the best part of game creation! We make something new, and then we push it to it's limits. Scenes were initially a prototype for walking around and maybe turning on some lights at the same time. It's scenes like Zavala, which push the boundaries and harden the tech for more complex situations going forward.

Doing a postmortem on an ambitious project like this feels like facing off against a Drake tank. So let's stare it down together and revisit some of these goals and challenges. The Usability and AI Control goals went great. AI control had a small number of powerful actions, behaviors and movement, and this high level approach fit the scenes flow-graph quite well.

As for Triggering, what's funny is the moment you start triggering sub systems like timelines and dialog the quicker you realize that those sub systems need improvements too. For example, the dialog node seems simple, but is actually deceptively complex. Playing dialog supports radio or 3d positioning, face animation, subtitling and multi-language support. The complexity of this one node encouraged us to rethink a ton of how we organize and trigger dialog! Ok, that's an understatement. (BEAT) I worked on the dialog tech for a year. It's really important to get it right, because it allows the triggering of it to become more simple!

The Big Diagram Challenge was also fairly successful. Sentence based visual scripts are quite readable, naming minimizes connection lines, and color coding helps draw the eye to where the story starts.

Even with all this many scene visual scripts became gigantic. There are many ideas next on the list to improve this including improved grouping and commenting, and potentially considering automatic node layout.

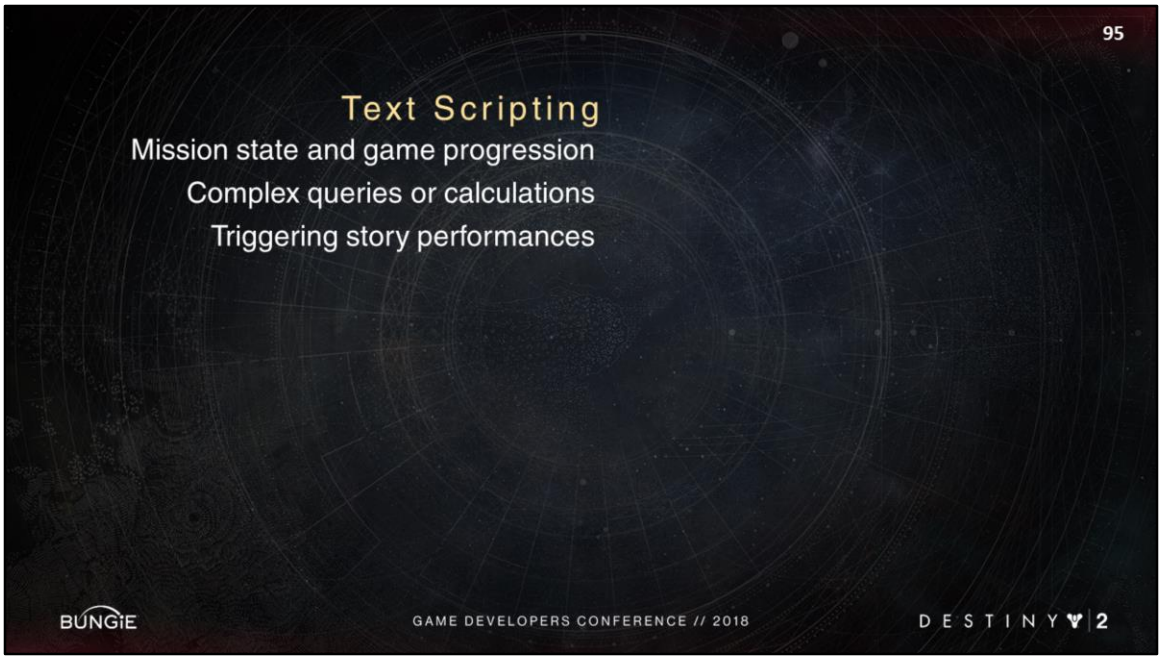
The Concurrency Challenge was solved in a clever way. Perhaps you noticed it? Throughout all the examples I've gone through, you may have realized that scenes don't have variables like counters or timers, and they don't support state change other than triggering – so they effectively have immutable inputs.

This was intentional. Removing direct control over counters and timers means race conditions and deadlocks are impossible. Logically impossible! This isn't just we hope we did it right this time impossible. But actually impossible. We have ideas on how to carefully reintroduce state back, but keeping logic errors out of story performances is critical.

And finally let's go back to the Better Than Text challenge. How did removing variables from visual scripts affect the visual script vs text script debate?

Now, before visual scripts could do everything text scripting could do. This is now intentionally no longer the case. So with this decision we actually chose that visual scripting is not better than text for game logic and complex calculations. Instead we focused in on what scenes were suited for, and that's flow visualization, triggering and waiting.

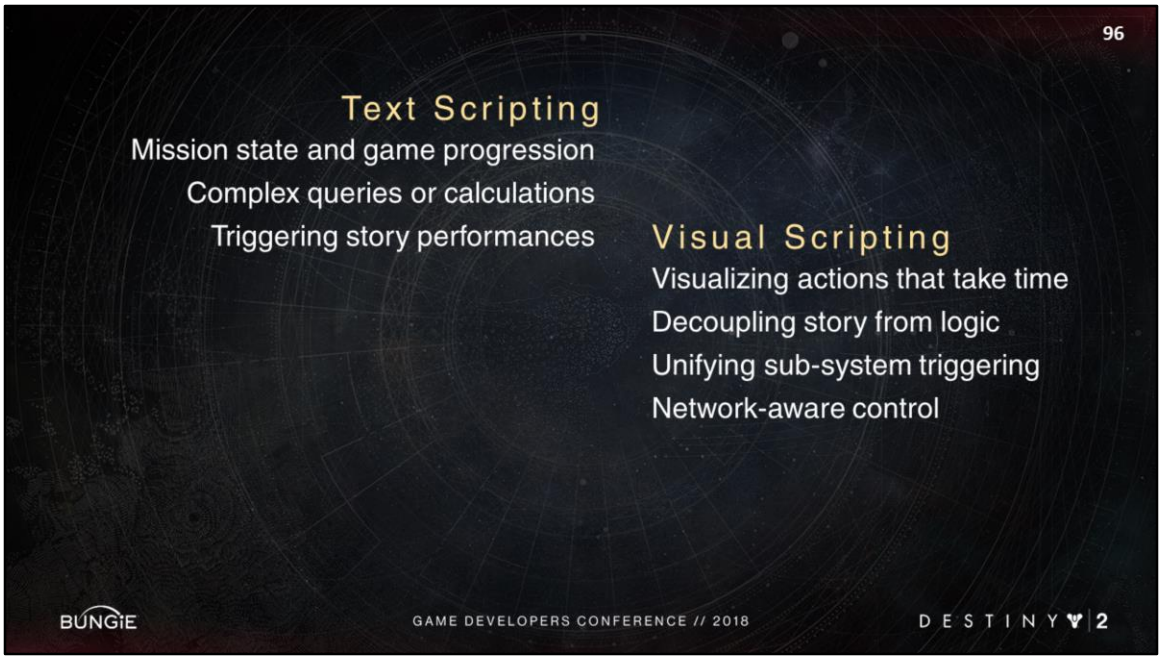
With that in mind, let's look again at how we ended up dividing responsibilities between text scripting, visual scripting, and timelines within the Bungie toolset.



As you saw, since we removed mutable variables from visual scripts, Text scripting now owns variables and state change. It helps you do logic to decide on mission and game progression.

It helps you perform complex calculations or query several data sources to decide what happens next.

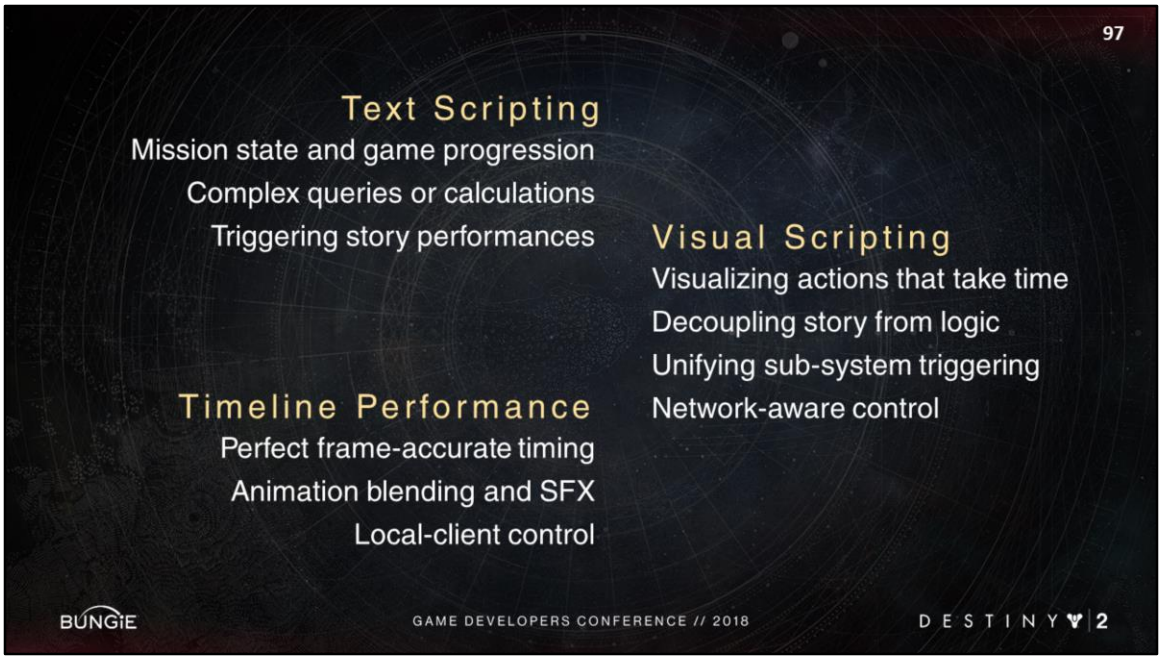
It then can trigger story performances.



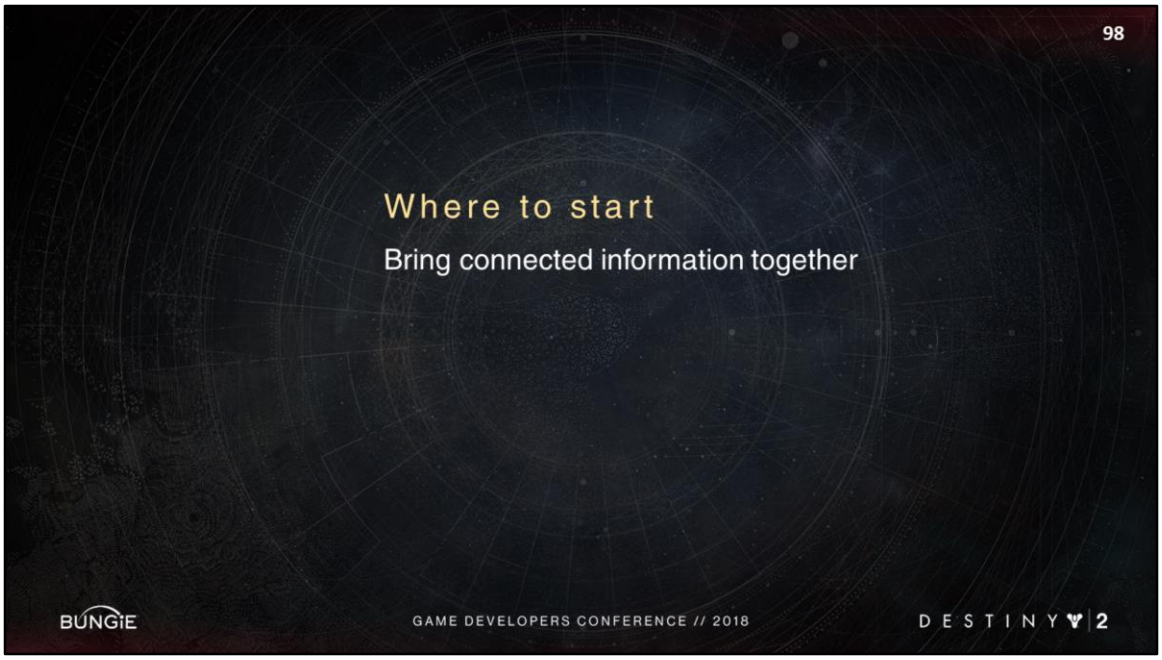
Visual scripting helps you visualize actions that take an unknown amount of time. This includes movement, shooting, animation, and dialog. Pretty much everything isn't a timeline!

By controlling waiting at a high level we decoupled story performance from scripting logic. This let storytellers work separately from game designers. So they could start working on stories, before there was even a map to tell the story in!

Visual scripting helps unify triggering across everything in your game engine. And best of all the statelessness of scenes prevented concurrency logic errors that are especially challenging in a network-aware cooperative game.



Lastly, Timelines provide the perfect frame-accurate timing that many of you already know and use daily. This is an essential tool, and as you saw from the examples we use this extensively timed animation with audio and special effects. It’s the ideal tool for client-side controlled story moments.



Finally, I hope that you all see this and are inspired to make your tools and workflows even better. Maybe you'll incorporate some of these ideas into your visual scripting system or make a visual storytelling tool. Your use cases and tech requirements will probably be a bit different, but here's my best advice on where to start your thinking.

Bring connected information together. For us this was by removing connection lines and emphasizing naming, but the primary goal was to keep all the important information together. This way your users don't have to follow a trail of clues all over the screen to understand what's happening. The goal is to make a tool that uses layout and typography to see the important information in a way your eye can easily follow.

Use sentences to express action. I had hoped this would be a helpful feature, but honestly was even more powerful then that. It turned out not a single person asked, “Hey, what does the shoot-and-kill node do?” “Is there any documentation on that?” Sentences help us use our years of language experience to intuit the meaning of these different actions.

Now ironically, sentences actually became a double edged sword. It turned out for us that sentences were so specific, it was hard to live up to them. The run-to node makes people run, but in a networking environment it was a big challenge to make characters actually reach their destination the same way, every time. Our implementation got better and better, but sentences highlighted any deviations. So my word of advice is, don’t try to pass something off as a run-to action when you’ve really written a run-nearby action!

Define your inputs and outputs. You might think this is about naming, but it's to enable reusing your story performances.

With scenes we used inputs and outputs to help create story prefabs. The prefab specified the inputs of the scene and allow them to be overridden: so the story "Ikora looks at Psion, Psion flees" can quickly become "Zavala looks at centurion, centurion flees".

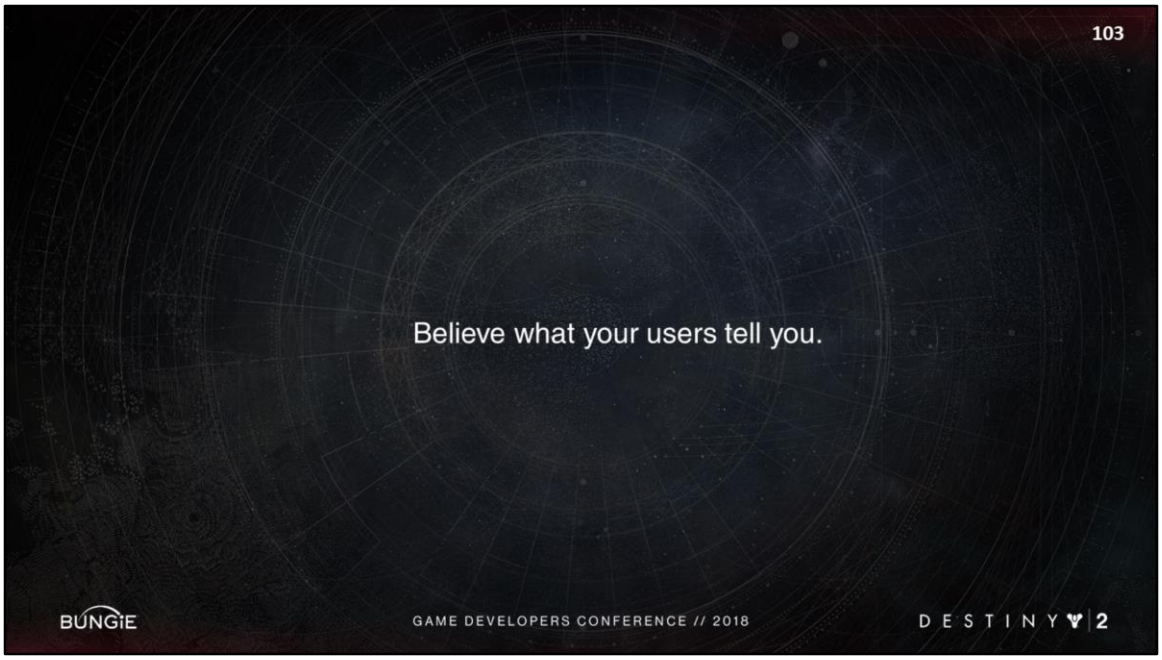
The idea of inputs and outputs can be extended into everything. What are the inputs of a tree? Just any old tree. Maybe the has a height input and a number of branches input, maybe you can specify the season, leaf size and color hue. The more you think about inputs and outputs the more powerful and reusable your prefabs, shaders, and scripts will become.

Finally, and this is a personal takeaway: remember to **stay humble and listen**.

I'm a professional engineer, and I get really excited about type systems, and distributed simulation. Probably too excited ☺

I have pretty clear ideas what *my* ideal visual scripting system would look like – but at the end of the day, this tool wasn't for me. A visual scripting system with my needs in mind will look very different from one for storytellers or designers.

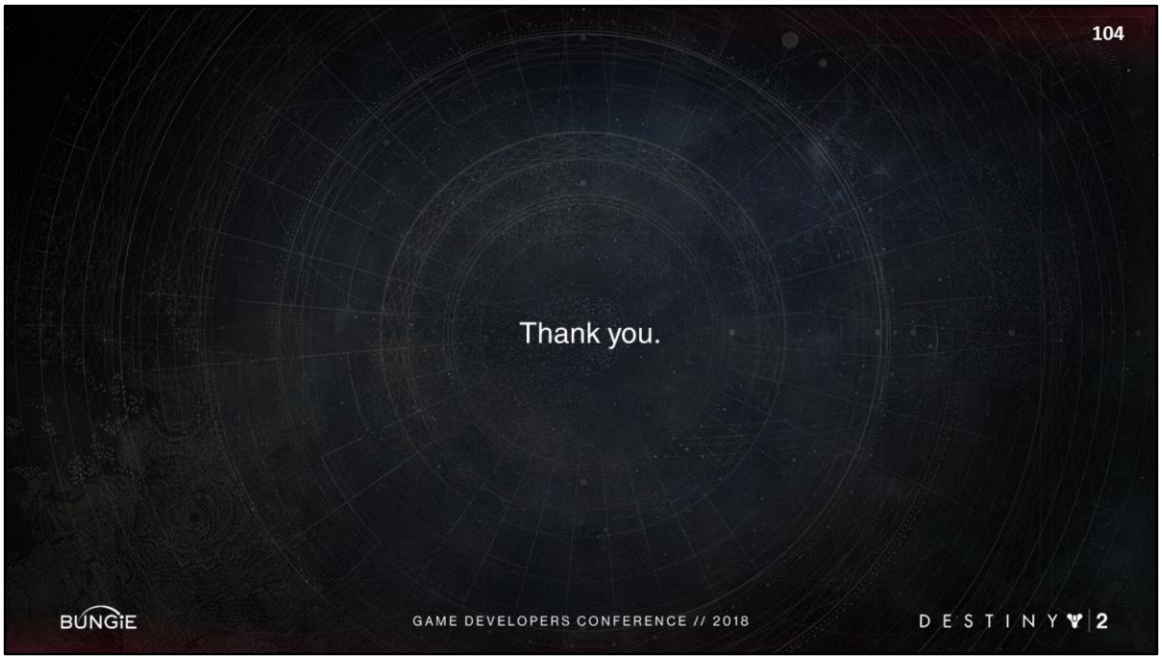
The only way to resolve this is to listen, iterate, and then listen again. But your iterating is only going to work if you <u>believe</u> what your users tell you…



That’s where the humility comes in.

For us, it turned out high level mission calculations and low level animation blending just didn’t fit well in a tool specializing on waiting and flow. So we exercised humility and improved the connections to other tools that could do those jobs better. The whole workflow was better for it.

More can be done, so we’re still iterating, and still listening. So in that spirit I want to thank the designers and artists who I iterated with. These are the creators of the homecoming mission and scene performances that you all saw today.



Thank you Tom Farnsworth, Tom Savile, Steve Bogolub, and Cullen Bradley. You used a *much* earlier version of the tool then everyone here has seen today. You pushed the tool to it's max capability whether it was ready for it not. So thank you.

So that's where to start; Bring information together, use sentences, define inputs and outputs, and listen with humility. If you do all that, you'll be well on your way to make a tool that unleashes your content creator's creativity, and blends story and action in a readable way.

I can't wait to see what you build.

Thanks everyone.