God of War: Ragnarök's Visual Scripting Solution

by Sam Sternklar

Santa Monica Studio

Hello everyone, My name is Sam Sternklar

Santa Monica Studio™

I am a senior programmer at Santa Monica Studio

GOD OF WAR™

RAGNARÖK

And I worked on God of War (2018) and it's sequel, God of War: Ragnarok.

What is this talk?

- Built a new Visual Scripting language for Ragnarök

1. Why?
2. Guidelines and Details
3. Tools and Technology
4. The Results
5. Summary and Conclusion

Santa Monica Studio

So I want to tell you a story about the development of God of War: Ragnarok, about the development of our new visual scripting system. In order to tell that story I've split this talk into five key parts:
The reasons we decided to build a new scripting system,
the guidelines and details comprising our system,
some of the tools and technologies we used for our system,
the results we got from shipping the system,
and finally a summary of our findings and conclusion.
Before we get fully underway, please note that this talk has minor spoilers for both God of War (2018) and God of War: Ragnarök.

Part 1: Why?

God of War (2018)'s Lua Scripting Solution

So to start our story, we have to get some background info, and that means talking about what came before Visual Scripting at Santa Monica Studio

The date was April 20th, 2018...

- 🎉God of War (2018) shipped!🎉
- We took a close look at how we built it
- Including taking a close look at our scripting systems

We start our story right after God of War (2018) shipped, when we held a postmortem of our various technologies. The process took a lot of time, but helped us understand where we were and where we wanted to go. Among the technologies that was most commonly called out by the team as problematic was our scripting system

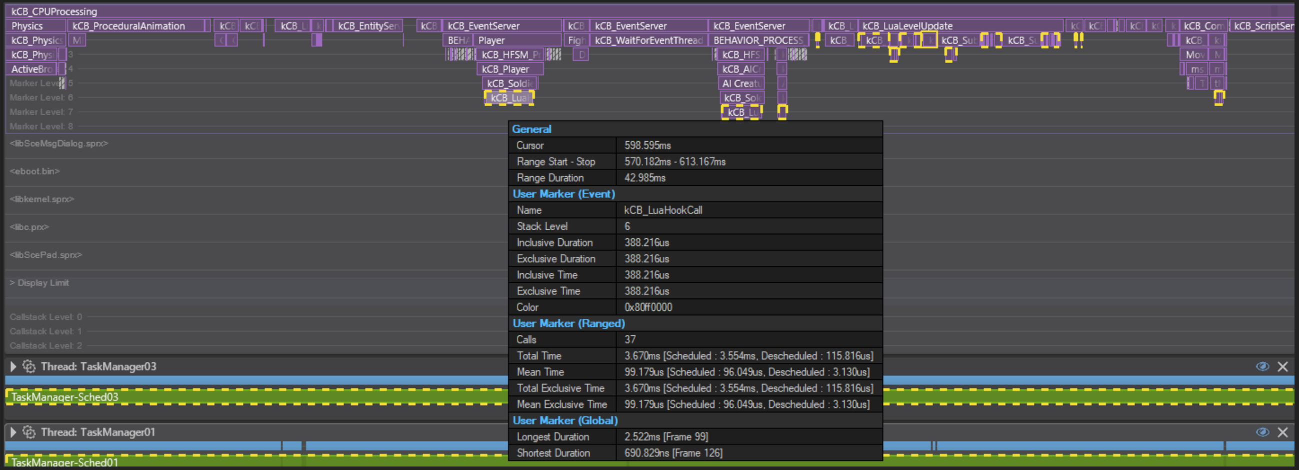
God of War (2018)'s Lua Scripting Solution

- Newly implemented lua 5.2
- Modified slightly to fit our specific needs
- Greatly improved designer freedom
- Critical to our success
- Had severe technical complications

We had recently implemented Lua 5.2 into our technology stack, though our version had some small extensions added to work with our object model better. Prior to it's inclusion, the scripting tools available to designers weren't particularly strong, and the addition of Lua led to a vast increase in freedom to our designers. Being able to shift work directly to the designers vastly improved iteration times and contributed directly to the success of God of War (2018). However, it was not without it's share of technical complications

The Complications

- It wasn't fast enough
- 3.5 ms...



General	
Cursor	598.595ms
Range Start - Stop	570.182ms - 613.167ms
Range Duration	42.985ms
User Marker (Event)	
Name	kCB_LuaHookCall
Stack Level	6
Inclusive Duration	388.216us
Exclusive Duration	388.216us
Inclusive Time	388.216us
Exclusive Time	388.216us
Color	0x80ff0000
User Marker (Ranged)	
Calls	37
Total Time	3.670ms [Scheduled : 3.554ms, Descheduled : 115.816us]
Mean Time	99.179us [Scheduled : 96.049us, Descheduled : 3.130us]
Total Exclusive Time	3.670ms [Scheduled : 3.554ms, Descheduled : 115.816us]
Mean Exclusive Time	99.179us [Scheduled : 96.049us, Descheduled : 3.130us]
User Marker (Global)	
Longest Duration	2.522ms [Frame 99]
Shortest Duration	690.829ns [Frame 126]

Santa Monica Studio

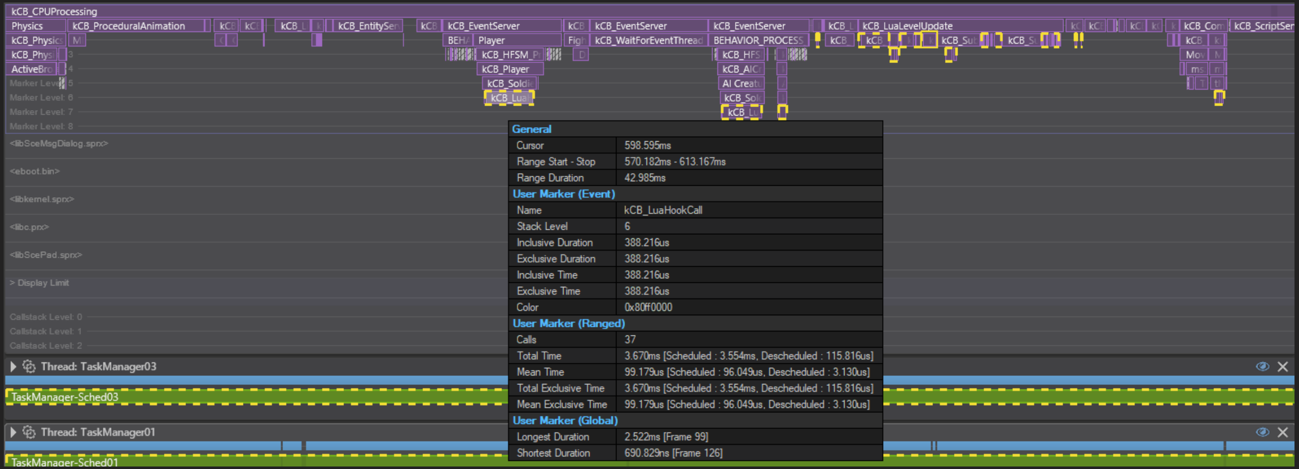
The first problem is that it took a lot of CPU time. The razor capture you see here shows a lua scripting frame at just over 3.5ms. Our target frame time for both God of War (2018) and God of War: Ragnarok was 33ms, or 30fps, on a standard PS4. With such a high-octane action game, we want to fit as much awesome stuff on screen at once. Let's take a look at what the player was doing while this capture was taken:

The Complications

☺ High octane ☺

The Complications

- It wasn't fast enough
- 3.5 ms standing still, Up to 5ms in combat
- Frequent Tick usage



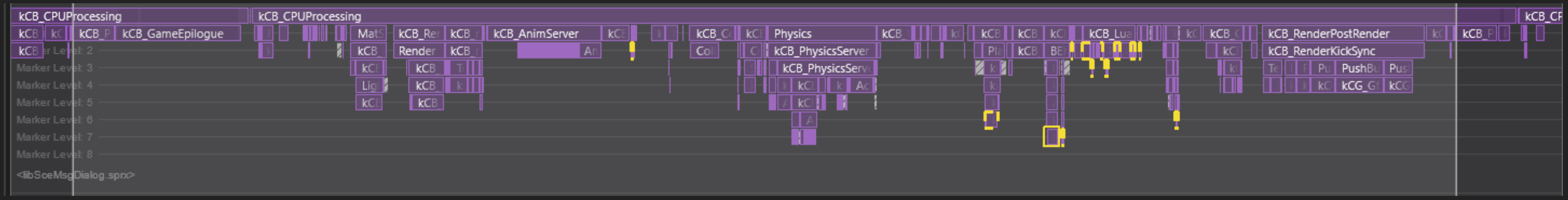
General	
Cursor	598.595ms
Range Start - Stop	570.182ms - 613.167ms
Range Duration	42.985ms
User Marker (Event)	
Name	kCB_LuaHookCall
Stack Level	6
Inclusive Duration	388.216us
Exclusive Duration	388.216us
Inclusive Time	388.216us
Exclusive Time	388.216us
Color	0x80ff0000
User Marker (Ranged)	
Calls	37
Total Time	3.670ms [Scheduled : 3.554ms, Descheduled : 115.816us]
Mean Time	99.179us [Scheduled : 96.049us, Descheduled : 3.130us]
Total Exclusive Time	3.670ms [Scheduled : 3.554ms, Descheduled : 115.816us]
Mean Exclusive Time	99.179us [Scheduled : 96.049us, Descheduled : 3.130us]
User Marker (Global)	
Longest Duration	2.522ms [Frame 99]
Shortest Duration	690.829ns [Frame 126]

Santa Monica Studio

So yeah, hands completely off the sticks, no actions occurring, with only Kratos and Atreus spawned in the center of the Lake of Nine, the capture shows we're spending 3.5ms just updating various lua scripts. Granted, the Lake of Nine is a main hub for the game, and there's a number of level modules present in the area, but 3.5ms is still excessive. In combat, this number gets as high as 5ms. Most of this is due to the scripting paradigm used – designers heavily leveraged ticks, and by the time we tried to final the game there was not enough time to re-architect the vast amount of lua scripting. We might have been able to add more events to prevent excessive polling, but that would have led to making our next problem even worse…

The Complications

- It was inflexible (for engineers)
- It could be called from anywhere, and had access to everything
- Code systems couldn't be moved to other threads because of scripting referencing it



Santa Monica Studio

The second problem is that it was inflexible for engineering. This razor capture is the same frame as the previous image, now showing the entire frame. Look at the yellow highlighted segments. During development, lua could be called from anywhere in the main game frame, including from the sections labeled Physics and Animation. This gave designers a ton of flexibility where engine events were provided, as they could do anything in lua. The problem is, if they can do anything, that necessarily means they must have write access to everything. And if they have access to everything, other threads can't. This locked not only scripting to the main thread for large portions of the frame, but prevented other systems from being moved to other threads because either they called into scripting, which changed state on the fly, or because another system called into scripting and referenced the to-be-moved system's data. Even on the same thread, within the same system, we had some problems – scripts could do anything, including modify the state of currently running systems, so moving where script executed relative to a system sometimes created incredibly subtle timing bugs. For an example, imagine you have two overlapping trigger volumes, where entering one will turn off the other and perform some operation. The evaluation of the trigger volumes themselves would change the output of the system if you were to change the timing of the scripting of the trigger volumes!

The Complications

- It was fragile
- Our designers are not engineers
- Various assumptions about valid (non-null values)
- Stale handles referenced frequently
- Single mistakes took down entire level VMs

The third problem is that it was fragile. We expected all our designers to be able to script things. Our designers, while amazing designers, are not disciplined engineers, nor (in my opinion) should they need to be. But with a general programming language like lua, it's very easy to miss a check for nil somewhere, and when that happens it can leave entire levels in a bad state that require us to stop running any scripting to preserve a stable dev environment. Even though the dev environment would still be running to provide information on what went wrong, once you shut down an entire level, it's likely that the developer is stuck anyways, and can't simply walk around the problem. In the final game package, any lua error will crash the game, making this situation untenable.

The Complications

- It relied on dynamic allocation and GC
- Our engine relies on a static memory model
- Frequently had to over-size memory pools for spikes
- GC leads to unpredictability frame-to-frame
- We don't like spending time on GC

Santa Monica Studio

The fourth problem is that lua is a garbage collected language. Our engine heavily pre-allocates memory, leaving very little slack around for growable heaps. We had very little ability to control the usage of our lua VMs, meaning we over-sized the lua memory pools for spikes – often a level which took a few hundred kilobytes on an average frame were sized for multiple megabytes due to unpredictable allocation on other frames. Furthermore, it's unpredictable and nondeterministic when GC will need to run, and when it does run it spikes. While the existence of GC allows lua to have extremely powerful data manipulation constructs, it's unpredictability frame-to-frame with such tight budgets means that it becomes more of a liability than an asset for us as we seek more and more control of our frame.

The Complications

- It wasn't fast enough
- It was inflexible (for engineers)
- It was fragile
- It relied on dynamic allocation and GC
- We can only kind of fix it without compromising lua

And among all of these problems, the most frustrating one was that there was nothing we could do. Sure, we could mitigate these problems – shortcuts throughout our engine to keep speed up, more careful systems design, better designer training practices, etc – but we couldn't FIX these problems without compromising what made lua so powerful.

So where does that leave us?

- We knew the sequel was going to be bigger
- Scaling with these problems would be painful
- Decided to experiment with a new language
- Specific to SMS, not a general purpose language

Santa Monica Studio

We knew from the start that the sequel had to be bigger, and that these problems don't scale. We wanted to use less memory, use less frame time, with more stable allocations and less error-prone designer flows, all while powering more content at once. To accomplish this, we decided to build our own scripting system specific to us and our needs.

Part 2: Guidelines and Details

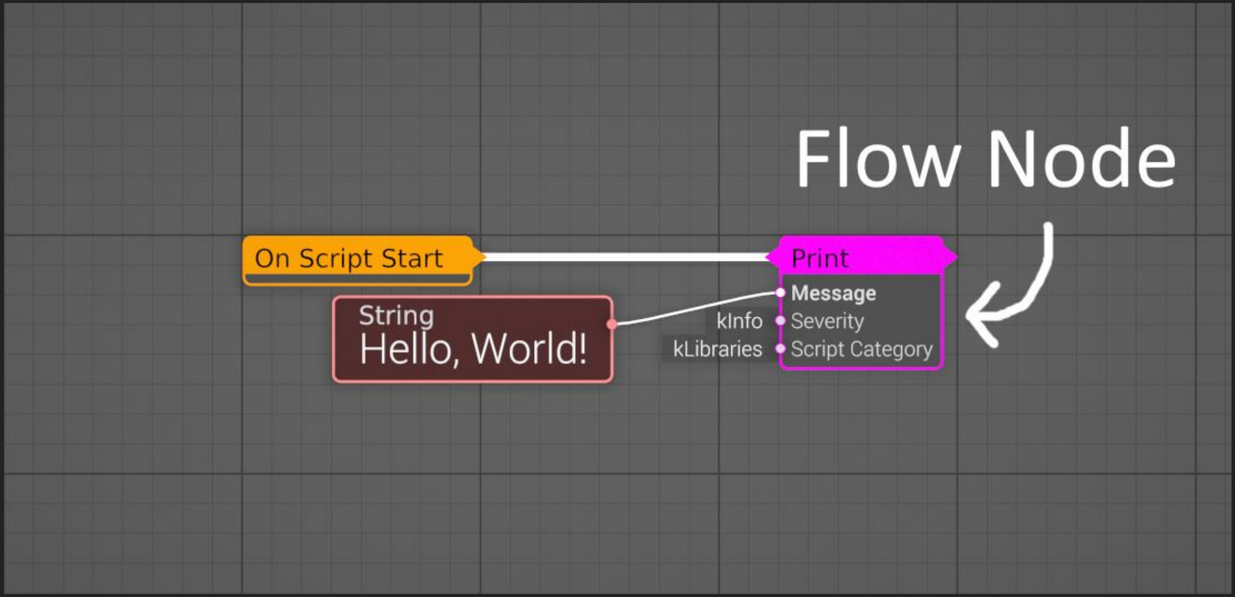
How we addressed some of those complications

Santa Monica Studio

When we went to get started with our system, we first had to design it. We came up with this set of core guidelines to fulfill the needs of our design groups alongside a set of basic constructs to power our scripting system. I'll give an overview of our basic constructs before diving into the guidelines and how we fulfilled them.

Basic Constructs – Flow Nodes

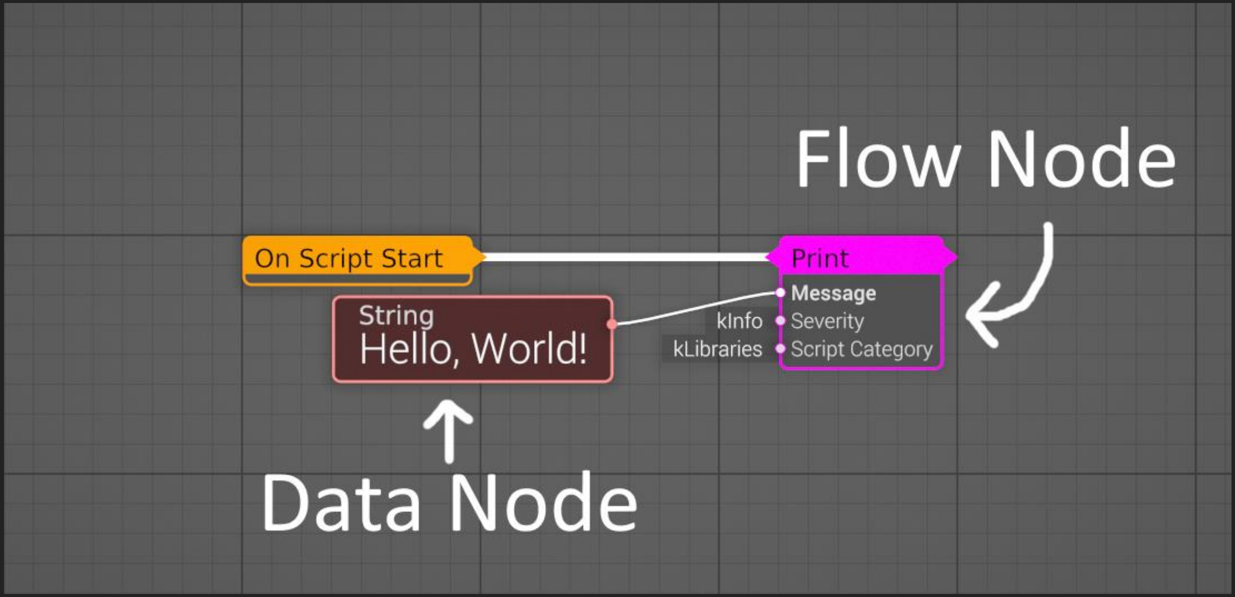
- Flow Nodes can Read and Modify state
- Must be connected by Flow



To start with, flow nodes are your standard function call. They can read and modify game state, but must be connected to other nodes by flow – the thick white line representing control flow through the script.

Basic Constructs – Data Nodes

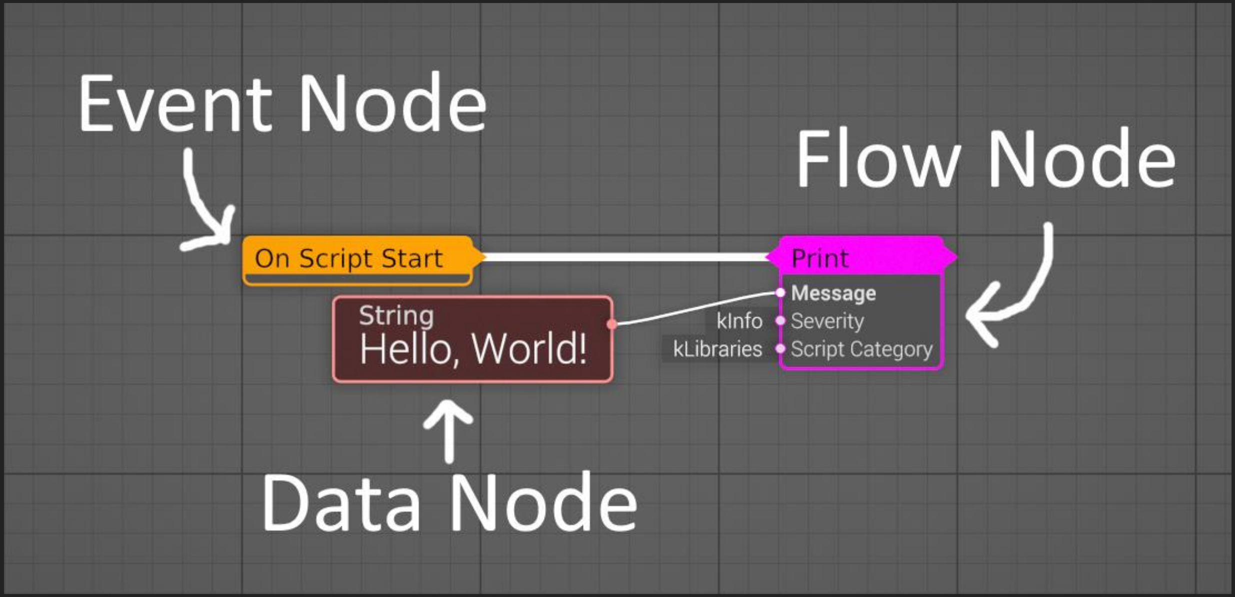
- Data nodes can only read state
- Do not need flow connection



Data nodes are basically const functions, they cannot modify state, but do not need to be connected to flow. They merely pass along data, represented by the thinner line connecting the string to the "Message" field on the print node.

Basic Constructs – Event Nodes

- Event nodes act as the entry point to a given script
- Can marshal state to script for evaluation



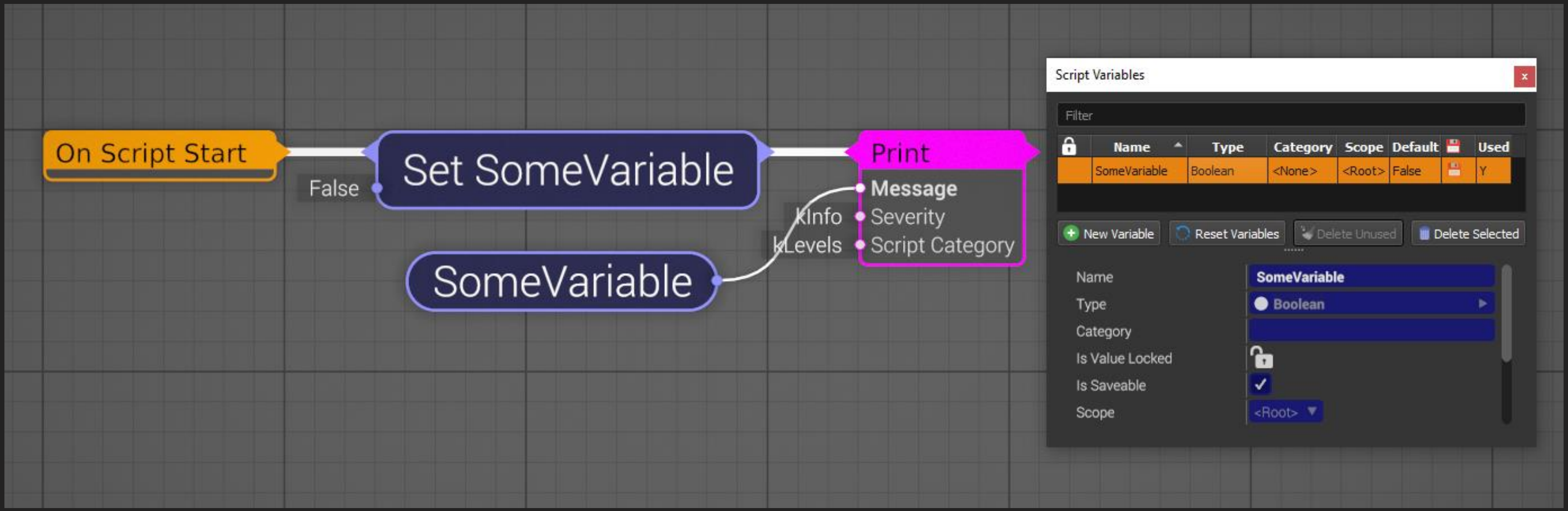
Santa Monica Studio

Event nodes are the entry point to any script, representing an event from the engine. They can marshal state into the script to assist the evaluation, but otherwise don't do much on their own.

In case you haven't noticed, the example I've used for the last three slides is "Hello, World!"

Basic Constructs – Variables

- Variables store script data in a safe form
- Can be accessed and modified by special nodes



Name	Type	Category	Scope	Default		Used
SomeVariable	Boolean	<None>	<Root>	False		Y

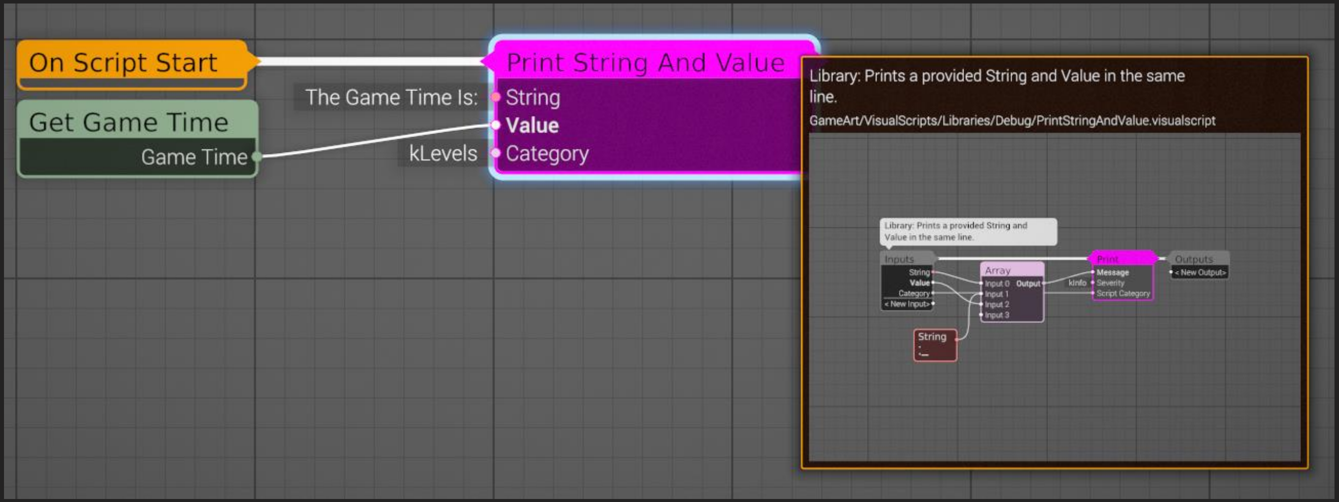
Name	SomeVariable
Type	Boolean
Category	
Is Value Locked	
Is Saveable	✓
Scope	<Root>

Santa Monica Studio

Variables are used to store script state between invocations, and are accessed and modified by special script nodes. You can see an example of the set and get nodes for a simple Boolean variable here.

Basic Constructs – Embedded Scripts

- Embedded Scripts are macros, expanded during compilation
- Placed in other scripts to encapsulate functionality



Library: Prints a provided String and Value in the same line.
GameArt/VisualScripts/Libraries/Debug/PrintStringAndValue.visualscript

And finally, we have Embedded Scripts, which are macro-scripts expanded during script compilation. Designers can author these embedded scripts to encapsulate complicated functionality, providing some basic organization to the script. You can see a quick view of the embedded script in the tooltip, but it's not always the easiest thing to read in such a small space.

Core Guidelines

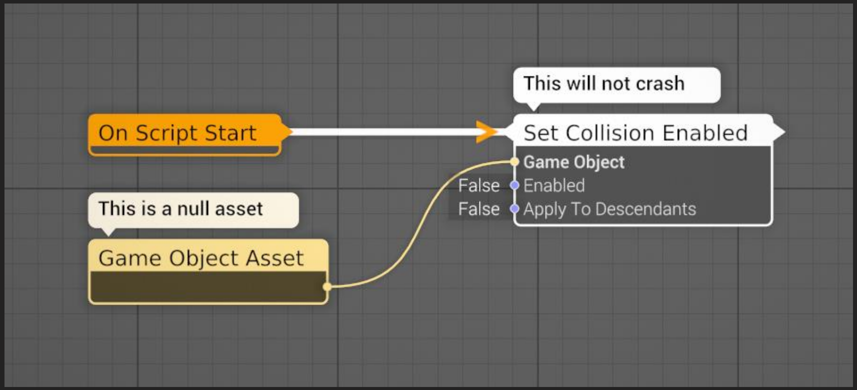
- All scripts run to completion, no formal halting/coroutines
- Execute with consistent, static memory
- All scripting possible from non-“OnTick” engine events
- Script Runtime should spend as little time in the interpreter as possible
- Scripts should be composable from other scripts
- Everything should be referenceable as an asset
- Complex Editor, Simple Runtime

Now that we’ve covered our base concepts, we can discuss our core guidelines. Here they are laid out. They are, in the order that I’ll cover them:
All scripting must always run to completion, with no formal halting or coroutines
All scripts should execute with a consistent and static memory footprint
All scripting should be possible from engine-originated events without the usage of OnTick
The scripting runtime should spend as little time in the interpreter as possible
Scripts should be composable from other scripts, representing functions or modules
Everything should be referenceable as an asset
We should aspire to have a simple runtime, pushing as much complexity to the editor where possible

Let’s go through each of these in depth:

Run to Completion – Errors

- Most scripts just grabbed an object, queried some state, then called a function on it
- All code-side nodes gracefully fail rather than crash
- Allowed better telemetry for specific errors

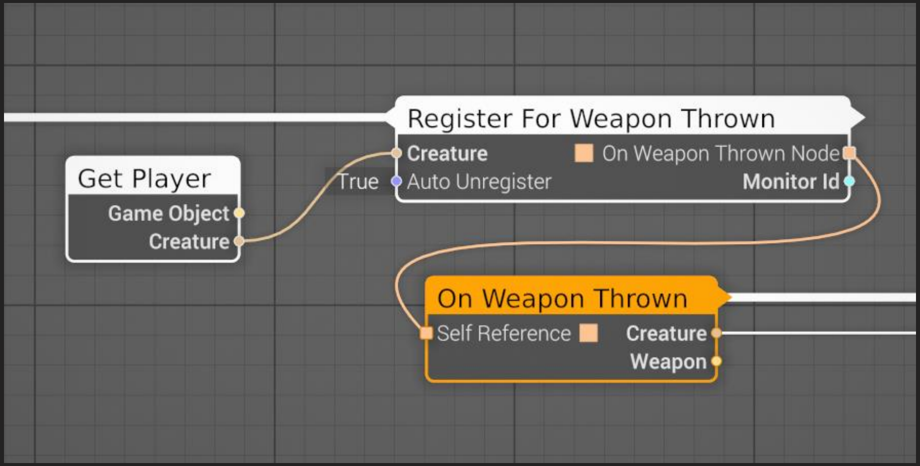


Santa Monica Studio

Our first guideline is that all scripts must run to completion. This means that we can't halt on a null object or missing reference. Instead of crashing, we would emit errors and allow the script to continue. We moved much of the error checking from design's responsibility to engineering's responsibility. The simple missing nil check was responsible for a very large portion of God of War (2018)'s scripting crashes, so anything we could do to mitigate this issue paid dividends. Because of this, we ended up with a much more stable executable, and with a lot more telemetry for what was going wrong in the game at any point.

Run to Completion – Closures/ Coroutines

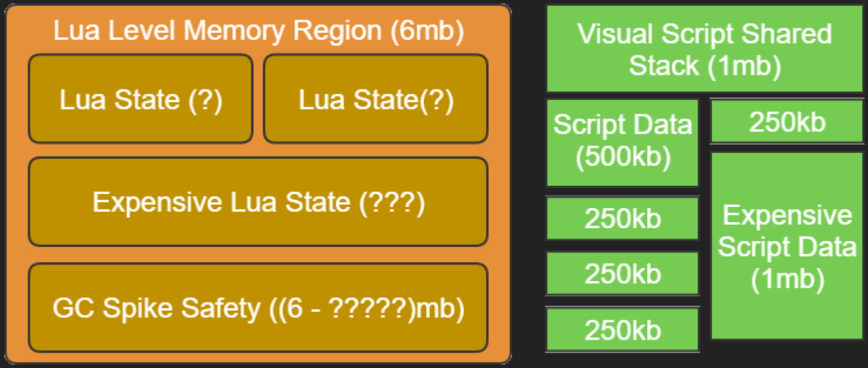
- None of our scripts needed full stack retained
- Instead we registered events with specific outputs
- Vastly simplified runtime requirements



In order to ensure scripts run to completion, we also avoided the use of formal coroutines and closures. We didn't ever want to worry about stack handle safety or memory used from a large number of suspended stacks. Instead, we allowed scripts to register for specific callbacks with specific outputs relevant to the event, and re-querying everything else. This vastly simplified our runtime and prevented the need for extensive handle safety checks. This also helped ensure that we were never working with stale values caught within closures, with the performance hit for re-fetching some values being an acceptable, even preferable cost.

Static Memory Usage

- On load, allocate for all variables
- Set aside reuseable 1mb stack buffer
- Scripts always run to completion, no need for closure space, coroutine stacks, or multiple concurrent stacks



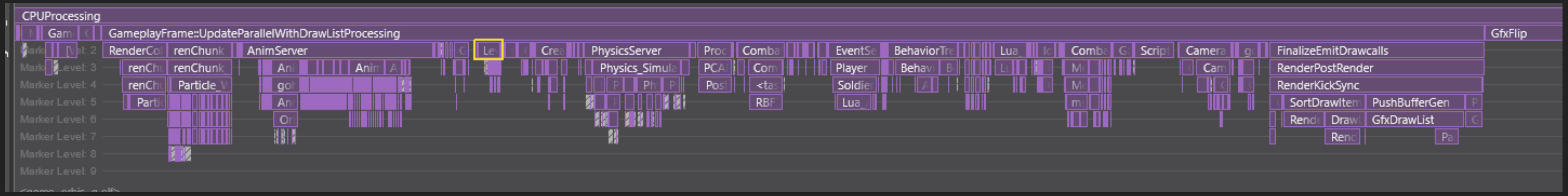
Lua Level Memory Region (6mb)
Lua State (?)
Lua State(?)
Expensive Lua State (???)
GC Spike Safety ((6 - ?????)mb)
Visual Script Shared Stack (1mb)
Script Data (500kb)
250kb
250kb
250kb
250kb
Expensive Script Data (1mb)

Santa Monica Studio

Our second guideline is that scripts must use a static amount of memory. As mentioned, this matches our engine model, where we pre-allocate pools of objects on init and leave very little up to general memory allocation. On loading the script, we pre-allocate space for all of the script's variables. In order to save on stack memory, we set aside a 1mb shared buffer that all scripts reuse. Where in Lua we had very little control over our allocation and were at the mercy of the VM and garbage collection, in Visual Script we knew where every single byte went. This had a secondary benefit of simplifying our script state management: If we had to destroy and recreate some pooled game object, we knew exactly what needed to be reset for it's script state rather than abandoning some set of tables within the general level VM and running GC, hoping that nothing is improperly holding a reference to the object's data.

Events and Eventization

- All calls into scripting are packaged and processed for later use, rather than occurring immediately
- All events execute at the same point in the frame
- Safe changes to system internals without affecting callbacks or scripting internals

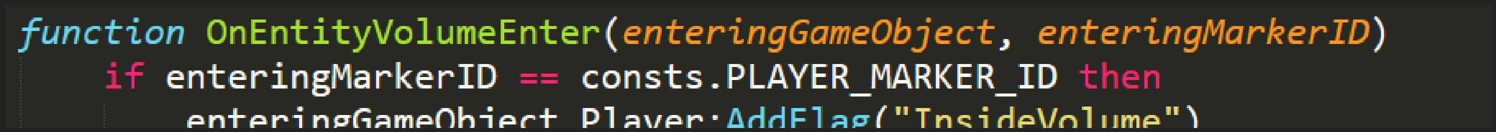


Our third guideline is that all access in and out of scripting is gated by events, especially avoiding OnTick. No matter where a call into scripting comes from in the frame, we package the call into an event and process it at a specific point in the frame. The razor capture below shows all of the visual scripting execution in the frame. This allowed us to change system internals at will without compromising the timing of callbacks. On rare occasion, this even meant we could safely put entire systems on other threads that were previously stuck to the main thread due to those scripting calls.

I promise by the end of the talk I'll tell you how much time that little segment takes, but you'll have to wait a moment for performance numbers ☺

Events and Minimizing Interpreter Time

- On evaluation, we realized the first thing that many of our events were doing was:



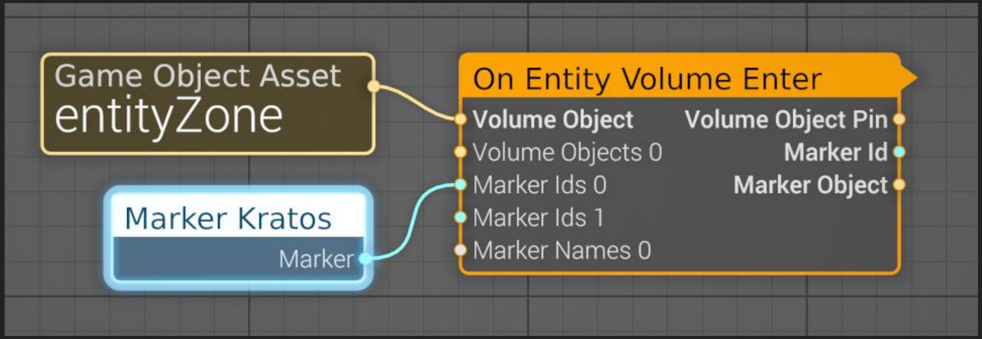
```
function OnEntityVolumeEnter(enteringGameObject, enteringMarkerID)
    if enteringMarkerID == consts.PLAYER_MARKER_ID then
        enteringGameObject.Player:AddFlag("InsideVolume")
```

- Paying VM startup and exec cost just to early out
- We identified common early-return conditions and added event filtering for them

As we scanned our existing lua to inform the design of our system, we noticed that the first thing that almost all of our existing events did was early-out on some condition relevant to the event. While it's good to restrict work done, we still had to pay the full cost of setting up the VM every time we called that function. We introduced event filtering for common early-return conditions to deal with this.

Events and Minimizing Interpreter Time

- Designers set up filters directly on the nodes



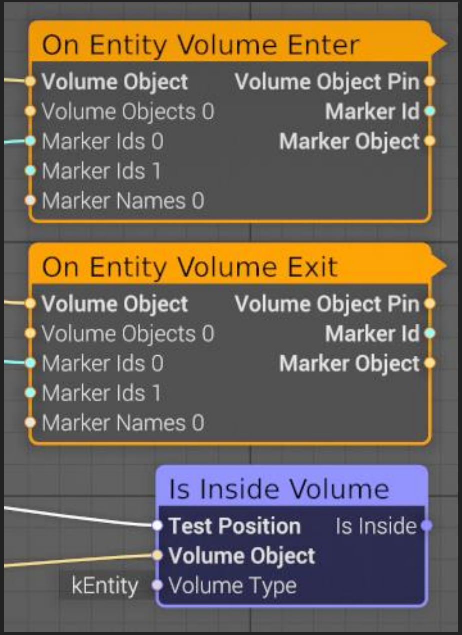
- Filters are evaluated at compile time, nodes accelerated by common filters
- Can be effectively batched for instances of script

Santa Monica Studio

Rather than checking the data once the event was being processed, we instead put filters directly on our event nodes. These are compile-time values, used when constructing acceleration structures for each node type. Before starting the VM, we check all of the accelerated pre-filters, and early out if any of them fail. What this does is move one of the most expensive parts of our scripting directly into C++, skipping much of the script VM execution we'd otherwise have to do! After all, if most of your scripts just early out, why not just move the early out into C++ and leave the rest to the designers? If the filter is completely independent of any context of the script, you can even batch them together, allowing you to evaluate hundreds of nodes' worth of behaviors at once without even setting up a full interpreter environment. This assists with our fourth guideline, that we should spend as little time in the interpreter as possible. This was so effective that we were able to treat all of our events as broadcasts by default, which also allowed designers to freely script without needing to create scaffolding between different scripts that would have received more targeted events!

Encouraging Event-Based Scripting

- This only works if we have the events
- Any time we add a query for a state, add an event for that state changing
- Educated designers on best practices
- Worked closely with designers to build any required functionality



One of the challenges with this event-based scripting approach is that it only works if we have a lot of events and don't need to leverage un-filterable events like OnTick. We made a rule for ourselves that any time we added a query for some engine state, we also add an event for that state changing. Between that and educating designers on best practices, we made the path of least resistance to any scripting in the game be one of the most common paradigms for improving script performance. We also regularly worked with designers to build any missing functionality as quickly as we could to ensure that they would not need to deviate from this golden path.

Composable Scripts

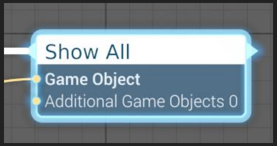
- Embedded Scripts naturally encapsulate state
- Designers can build their own complex behavior with embedded scripts
- Engineers have seamless drop-in replacement

Santa Monica Studio

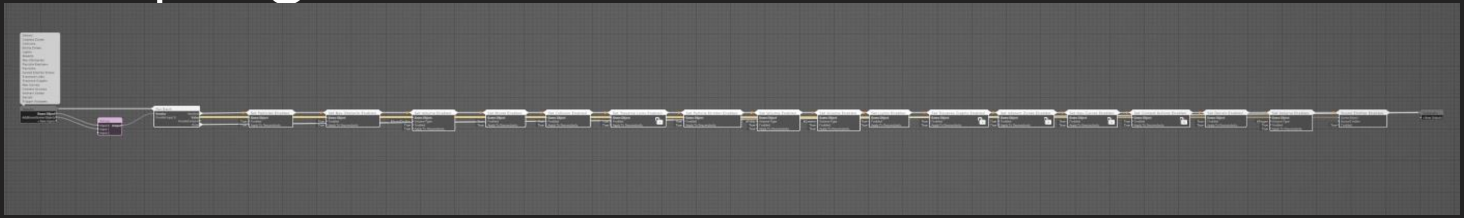
Our fifth guideline is that our scripts must be composeable from other scripts. We already mentioned embedded scripts, which solve this problem for us. What might not be obvious is that with such well-encapsulated state, programmers have an easy drop-in replacement point for common behaviors. This was so important that our editor had a dedicated button for packaging up sections of logic into their own miniature sub-scripts.

Composable Scripts – Behaviors

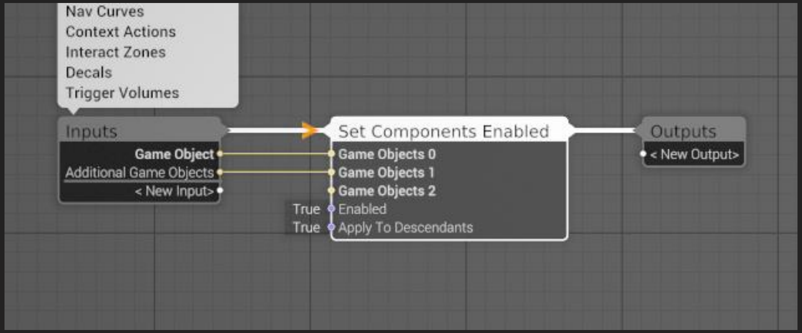
- This is our "Show All" script:



- Before programmer intervention:



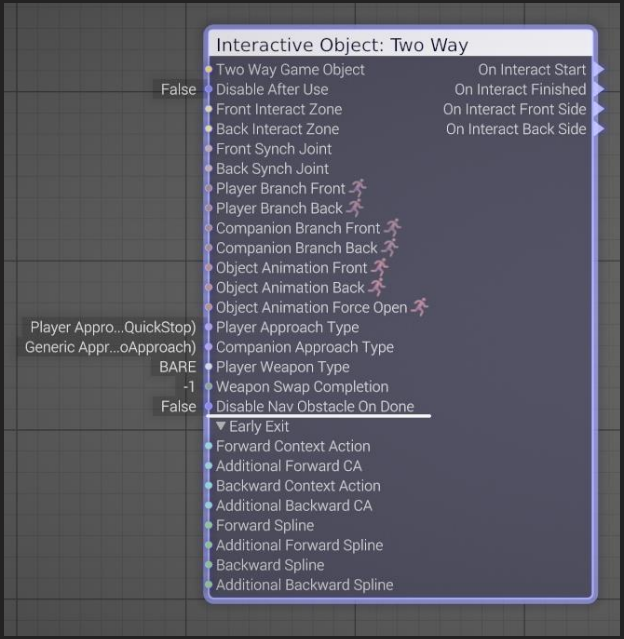
- After:



Santa Monica Studio

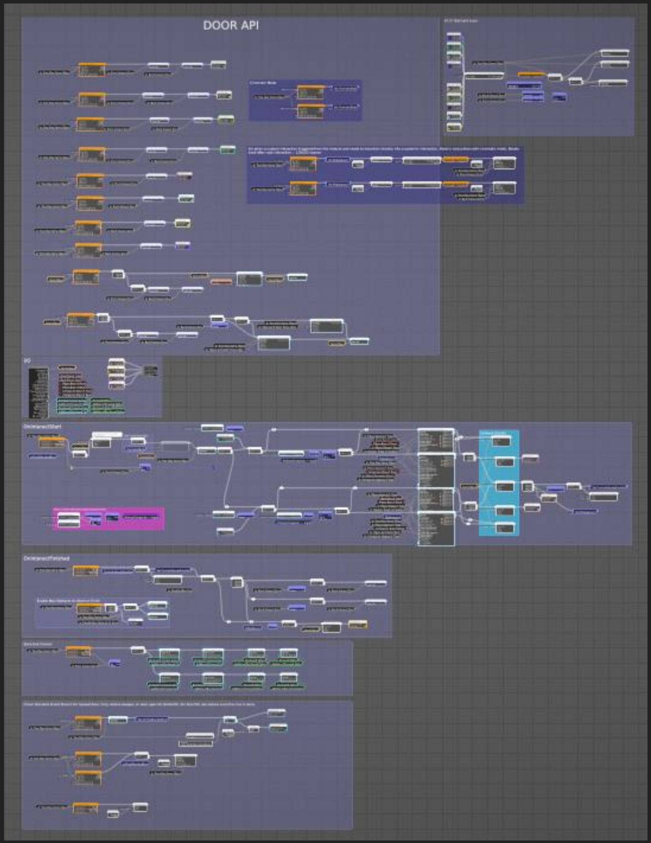
An example of this is our "Show All" script. During general iteration, a designer created the first iteration of the script, which just goes through every possible component on a game object and enables them. This is inefficient when used extensively, since so much of the behavior occurs directly in scripting, so a programmer made the second node below, which wraps up all of that logic in nice C++. We'll revisit this example later in Part 4.

Composable Scripts – Modules



Interactive Object: Two Way

	Inputs	Outputs
	Two Way Game Object	On Interact Start
False	Disable After Use	On Interact Finished
	Front Interact Zone	On Interact Front Side
	Back Interact Zone	On Interact Back Side
	Front Synch Joint	
	Back Synch Joint	
	Player Branch Front	
	Player Branch Back	
	Companion Branch Front	
	Companion Branch Back	
	Object Animation Front	
	Object Animation Back	
	Object Animation Force Open	
Player Appro...QuickStop)	Player Approach Type	
Generic Appr...oApproach)	Companion Approach Type	
BARE	Player Weapon Type	
-1	Weapon Swap Completion	
False	Disable Nav Obstacle On Done	
	▼ Early Exit	
	Forward Context Action	
	Additional Forward CA	
	Backward Context Action	
	Additional Backward CA	
	Forward Spline	
	Additional Forward Spline	
	Backward Spline	
	Additional Backward Spline	

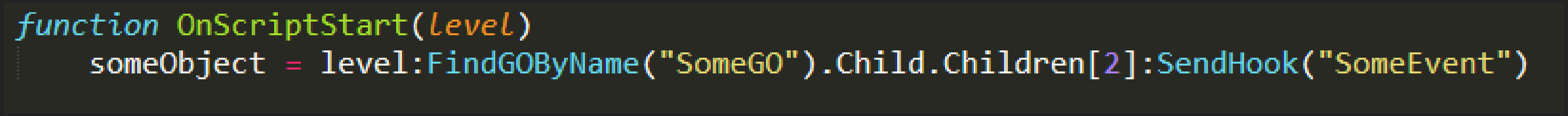


One of our other usages of scripting is to wrap up large swaths of scripting into modules. These modules can be placed directly into any script to inherit large pieces of functionality. The "Interactive Object: Two Way" node you see on screen now is used in all of our door and gate scripts that separate spaces from one another, without needing to manage either script inheritance or component interoperability.

It's notable that we accomplished this level of abstraction by treating all embedded scripts as true macros, expanded during compile time, rather than attempting to re-use the same set of node data for multiple instantiations of a single embedded script. As we just mentioned, we put a lot of static data on our event nodes to let pre-interpreter filtering work. Without treating embedded scripts as expandable macros, there would be no clean way to map static data exposed via an embedded script back to the various events that would be later evaluated. The added complexity would have drowned us in work, and the nodes themselves were not that big, so we didn't worry too much about it. There were a few cases we had to optimize late in the project, but they were isolated occurrances.

Assetization

- An unfortunately frequent pattern:



```
function OnScriptStart(level)
    someObject = level:FindGOByName("SomeGO").Child.Children[2]:SendHook("SomeEvent")
```

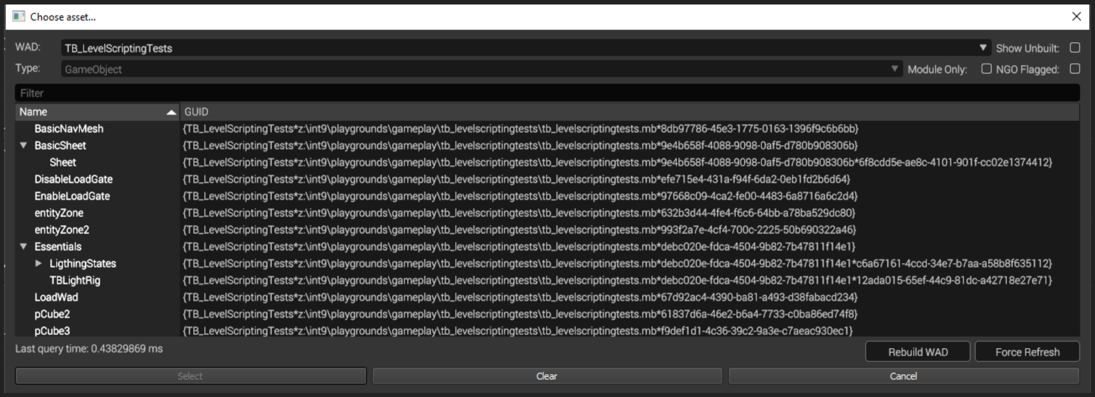
- Did not have strong tools to reference objects from game data
- Could only look up objects by name

Santa Monica Studio

Our sixth guideline is that almost anything that could be treated as an asset should be treated as an asset. An unfortunately common lua pattern was to grab some root game object in a level, then traverse it's children until you reach some other arbitrary child object. We could only find objects by name, and if it's an instanced object there's no clean way to grab some child without traversing the entire game object tree for it. We had few tools to build strong references to any game data.

Assetization – The Asset Database

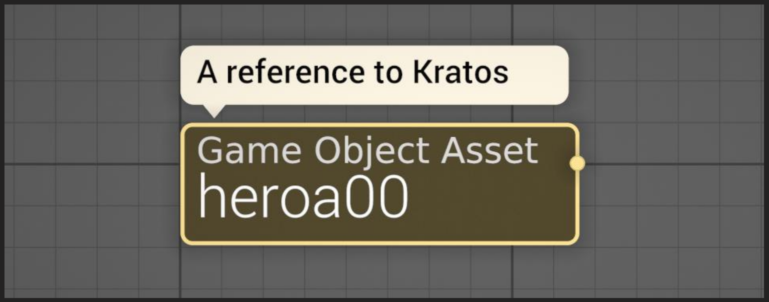
- Built a database from build outputs of object paths
- Runtime lookup of GUID->Object handle



WAD: TB_LevelScriptingTests
Type: GameObject

Name	GUID
BasicNavMesh	{TB_LevelScriptingTests*z:\int9\playgrounds\gameplay\tb_levelscriptingtests\tb_levelscriptingtests.mb*8db97786-45e3-1775-0163-1396f9c6b6bb}
▼ BasicSheet	{TB_LevelScriptingTests*z:\int9\playgrounds\gameplay\tb_levelscriptingtests\tb_levelscriptingtests.mb*9e4b658f-4088-9098-0af5-d780b908306b}
Sheet	{TB_LevelScriptingTests*z:\int9\playgrounds\gameplay\tb_levelscriptingtests\tb_levelscriptingtests.mb*9e4b658f-4088-9098-0af5-d780b908306b*6f8cdd5e-ae8c-4101-901f-cc02e1374412}
DisableLoadGate	{TB_LevelScriptingTests*z:\int9\playgrounds\gameplay\tb_levelscriptingtests\tb_levelscriptingtests.mb*efe715e4-431a-f94f-6da2-0eb1fd2b6d64}
EnableLoadGate	{TB_LevelScriptingTests*z:\int9\playgrounds\gameplay\tb_levelscriptingtests\tb_levelscriptingtests.mb*97668c09-4ca2-fe00-4483-6a8716a6c2d4}
entityZone	{TB_LevelScriptingTests*z:\int9\playgrounds\gameplay\tb_levelscriptingtests\tb_levelscriptingtests.mb*632b3d44-4fe4-f6c6-64bb-a78ba529dc80}
entityZone2	{TB_LevelScriptingTests*z:\int9\playgrounds\gameplay\tb_levelscriptingtests\tb_levelscriptingtests.mb*993f2a7e-4cf4-700c-2225-50b690322a46}
▼ Essentials	{TB_LevelScriptingTests*z:\int9\playgrounds\gameplay\tb_levelscriptingtests\tb_levelscriptingtests.mb*debc020e-fdca-4504-9b82-7b47811f14e1}
▶ LigthingStates	{TB_LevelScriptingTests*z:\int9\playgrounds\gameplay\tb_levelscriptingtests\tb_levelscriptingtests.mb*debc020e-fdca-4504-9b82-7b47811f14e1*c6a67161-4ccd-34e7-b7aa-a58b8f635112}
TBLightRig	{TB_LevelScriptingTests*z:\int9\playgrounds\gameplay\tb_levelscriptingtests\tb_levelscriptingtests.mb*debc020e-fdca-4504-9b82-7b47811f14e1*12ada015-65ef-44c9-81dc-a42718e27e71}
LoadWad	{TB_LevelScriptingTests*z:\int9\playgrounds\gameplay\tb_levelscriptingtests\tb_levelscriptingtests.mb*67d92ac4-4390-ba81-a493-d38fabacd234}
pCube2	{TB_LevelScriptingTests*z:\int9\playgrounds\gameplay\tb_levelscriptingtests\tb_levelscriptingtests.mb*61837d6a-46e2-b6a4-7733-c0ba86ed74f8}
pCube3	{TB_LevelScriptingTests*z:\int9\playgrounds\gameplay\tb_levelscriptingtests\tb_levelscriptingtests.mb*f9def1d1-4c36-39c2-9a3e-c7aeac930ec1}

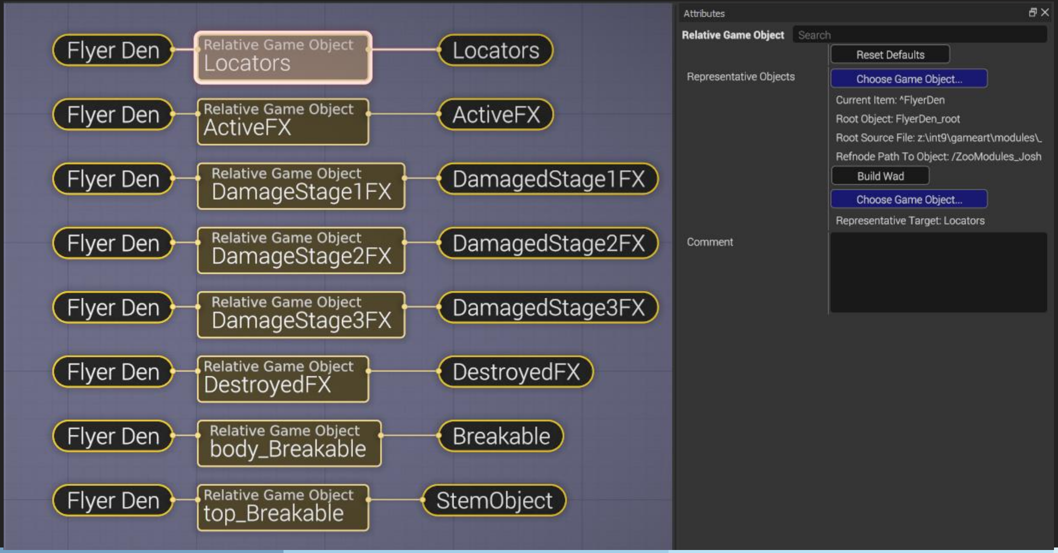
Last query time: 0.43829869 ms



To resolve this, we built an asset database that tracked the GUIDs of each object and leveraged an existing runtime lookup of GUID path through the scene to game object to marshal data into the scripting engine. What you see on a left is the editor window that shows the object, with the node on the right being a direct reference to a game object. The asset database allowed us to track every scripting reference to an object, and helped us identify when any asset reference was potentially erroneous. Without that, the simple act of debugging a missing reference wouldn't have been possible.

Assetization – Relative Assets

- Given a pair of representative objects, rebase path from input to some other object in the scene below it
- Indistinguishable from direct reference at runtime



Relative Game Object

Representative Objects

Current Item: ^FlyerDen
Root Object: FlyerDen_root
Root Source File: z:\int9\gameart\modules_
Refnode Path To Object: /ZooModules_Josh

Representative Target: Locators

Comment

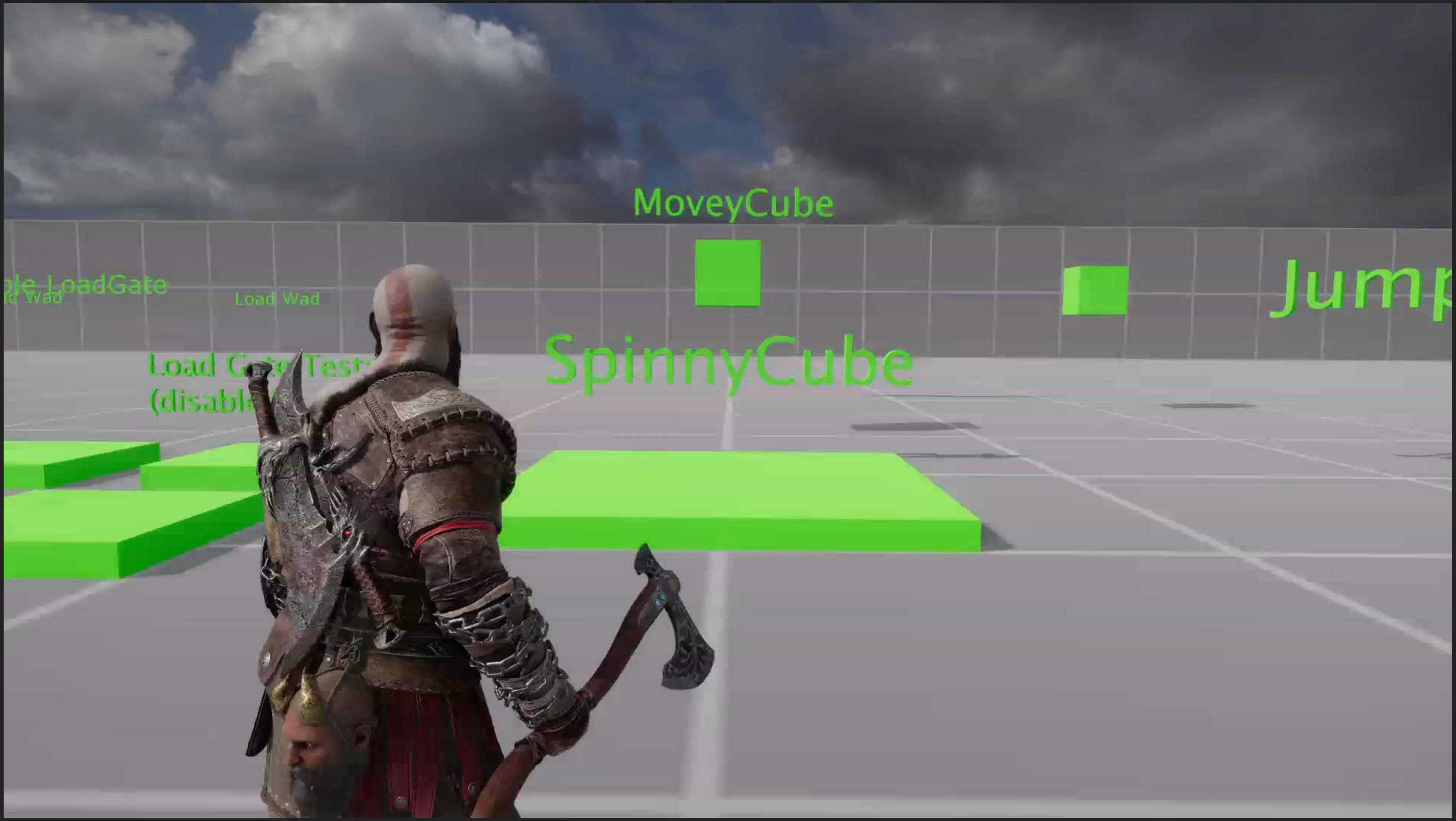
Some of our modules rely on input references. Since those input references may be to that same root object we had in the lua example, and we needed some way to safely fetch child objects within the scene, we also created a relative asset lookup. Because we stored the whole path of each object reference, given a representative base and target object, we can find any arbitrary child object by appending the relative path from the base object to the target to whatever runtime object we are searching for. Standard mapping issues apply when a designer deletes and re-creates an object, but in general this allowed us to create a robust and far less error-prone representation of any game object in our scripting system.

Complex Editor, Simple Interpreter

- Our compilation and build system are deterministic
- Our full game simulation is not
- Easier to debug deterministic data
- Safer and easier to test for changes in compiler output than complex runtime changes

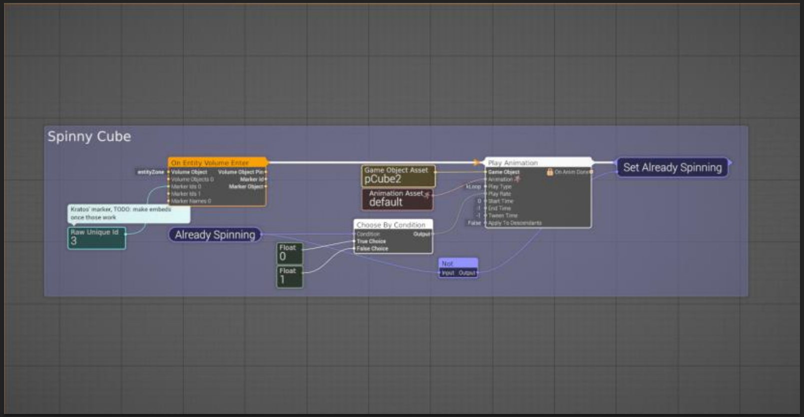
Our seventh and final guideline is that we wanted our interpreter to be as simple as possible (as noted on other slides), and that we were to push as much complexity to the editor as possible. We're not language designers, and debugging realtime applications is difficult. We much prefer debugging deterministic compiler output than we do a complex runtime. Being honest with ourselves, we knew we were going to make mistakes and that catching them would be difficult, and so we had to choose the safest route. With safety comes security, and with security comes trust. And a designer must trust their compiler, or else every issue a designer ever encountered would suddenly become the compiler's fault (whether or not it actually was)

Examples – Spinning Cube



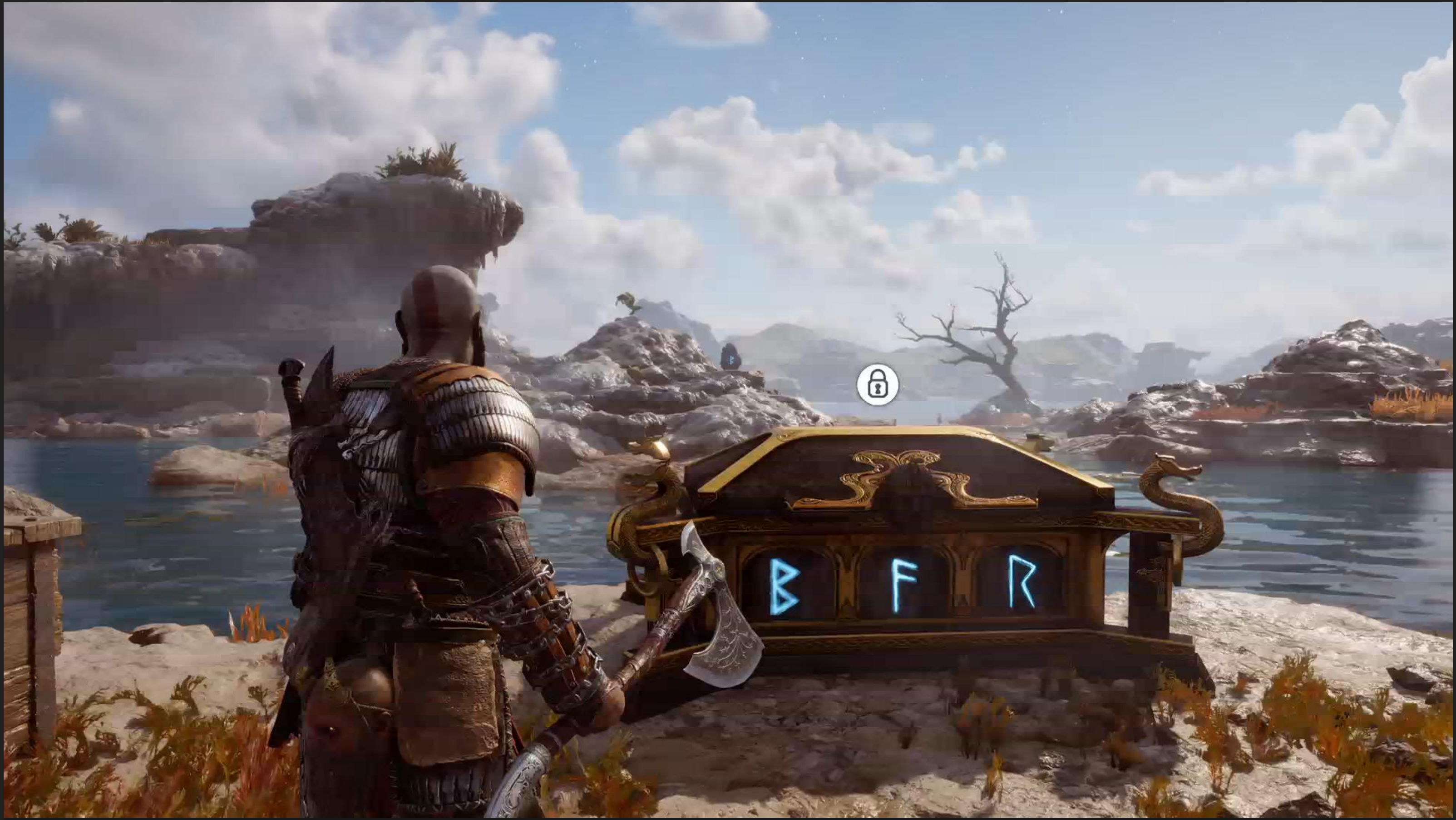
Let's talk examples. One of our earliest test cases was this simple spinning cube you see now. When you step on the pad, it either starts or stops spinning based on its previous state.

Examples – Spinning Cube



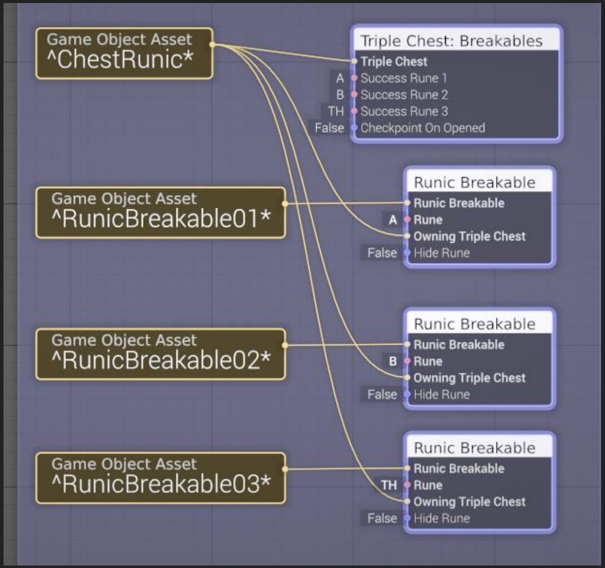
And this is the entire script used to make this work. When Kratos enters volume "entityZone", we play or pause an animation based on whether or not the object was already spinning or not. As this was one of our first tests, you can see artefacts like a raw Unique ID that would nowadays be part of an embedded script.

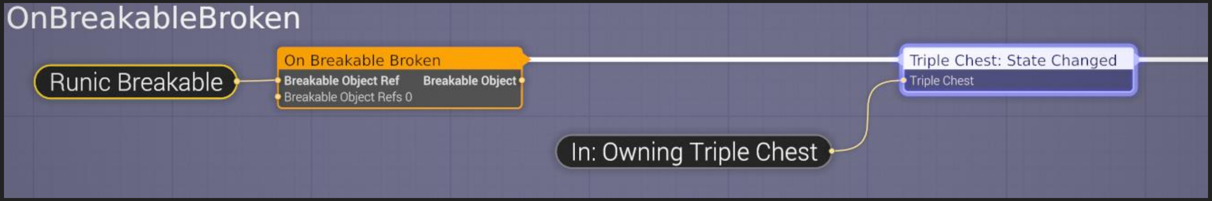
Examples – Nornir Chest

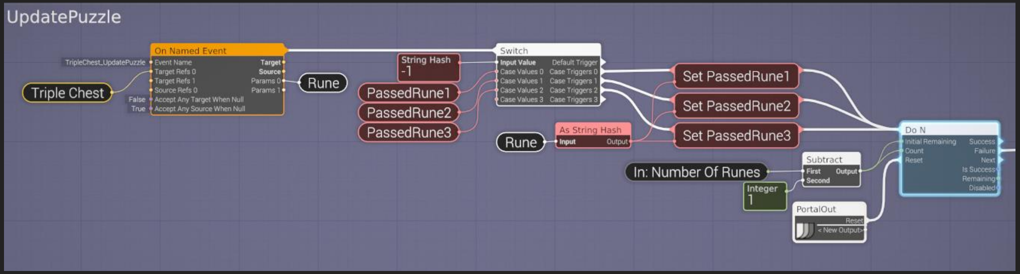


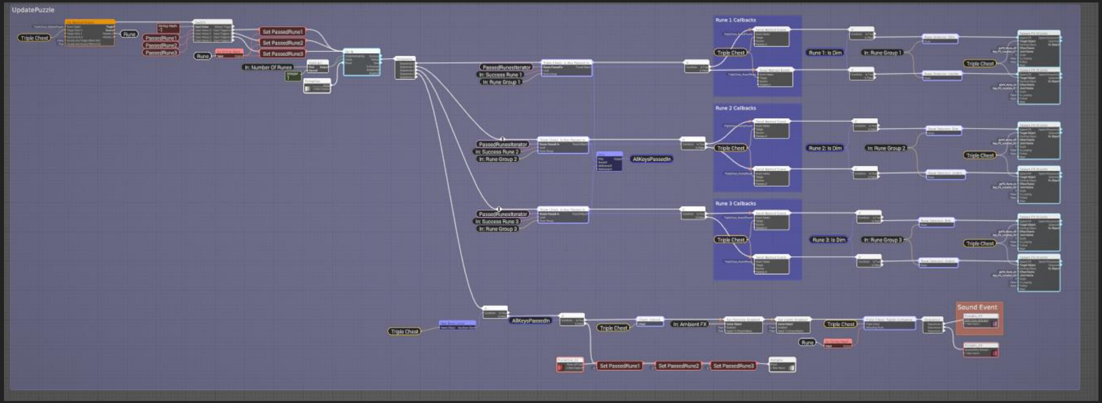
A slightly more complex and shipping example is this Nornir Chest, or as it's called internally, a Triple Chest. The puzzle is relatively straightforward – break three gnomes, or otherwise solve three mini puzzles matching the runes on the chest to open the chest and claim the spoils. These chests are often paired with other mechanics that may obscure individual pieces, like the geyser in this video.

Examples – Nornir Chest









What you can see here are excerpts from a number of different scripts. In the top left, you see what is placed in each level script – a module script representing the chest and it's three breakables gnomes. In the top right is some scripting from the breakable statue script, showing that when the object breaks it will send an event to the triple chest script. The event is caught in the images below and to the right, which perform some arbitration depending on which rune was broken before triggering state changes like lights and FX on the chest, and eventually opening the chest itself.

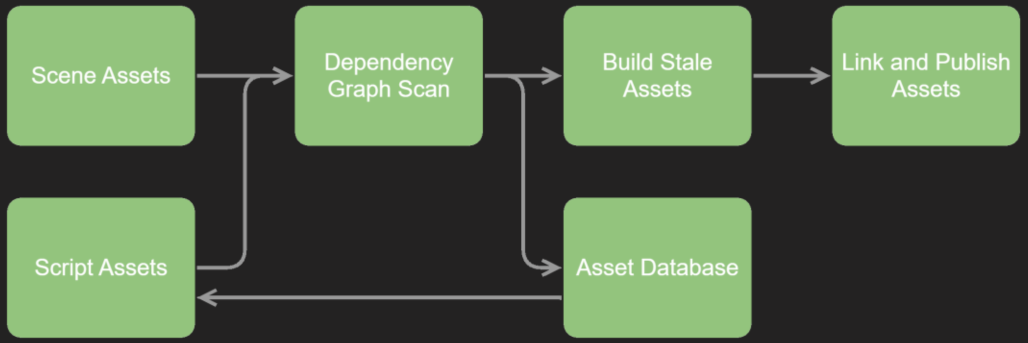
Part 3: Tools and Tech

What tools did we need to fulfill those guidelines?

So, now that we've covered our guidelines, we have to talk about the tools we built to make this all work and to make it easy to iterate on. All of these tools are in some way related to our editor tools.

Missing Pieces – Asset Database

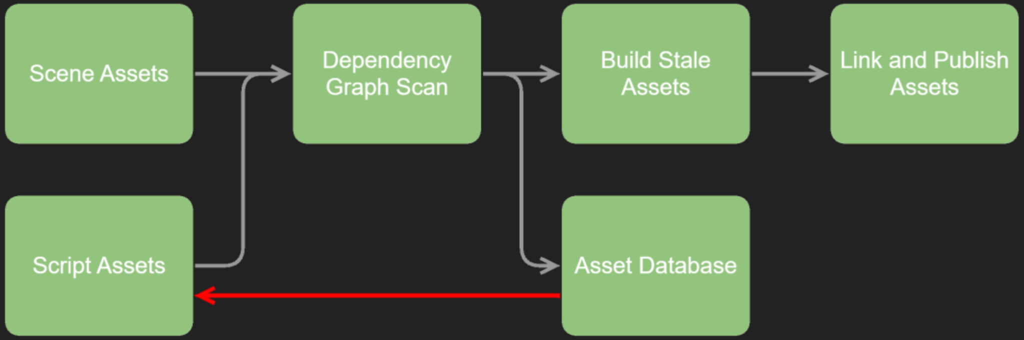
- Core plan requires an asset database
- We did not have an asset database
- We appended a MySQL DB to our build pipeline to capture build outputs



To start with, our core plan required an asset database in order to support assetization and editor lookups, which we didn't have at the start of the initiative. We appended a MySQL DB and asset scraping service to our build pipeline as a quick way of getting this stood up.

Missing Pieces – Asset Database

- Requires frequent scene scans just to populate DB
- Scales poorly closer to completion
- Had to make custom build modes to reduce overhead



However, because the database feeds into the scene, and the scene feeds into the database, this required frequent scene rebuilds just to populate the database. This dependency is in part because of how our scene is structured, but that's a topic for another talk. While the database itself held up well, this workflow scaled poorly as more assets were added to the game and required an asset-database-specific mode to our build system to reduce unnecessary overhead during database refresh builds. Given the state of our scene structure at the time that we were figuring out the workflow, we didn't really have a better option, so we did the best we could with what we had.

Missing Pieces – Editor

- We did not have an editor
- Our sister studio, Guerrilla, did for their engine
- They gave us a copy to retrofit to our tech

Any visual scripting system needs an editor to manipulate script data. We did not have an editor. Thankfully, our sister studio Guerilla did for their engine, and were happy to let us take a copy of their codebase. We gutted and retrofitted their technology to work with our own, and while that took a lot of time upfront it also gave us a truly priceless starting point that we leveraged into better functionality.

Missing Pieces – Editor

- Editor took longer than everything else put together
- 5 human-years of requests generated in 1 month

That is not to say that the editor was a small endeavor. Designers requested almost 5 human-years of features in just the first month of usage alone! Across the whole project, the editor got by far the most work. However, we did somewhat expect this, what with us wanting a complex editor and simple interpreter.

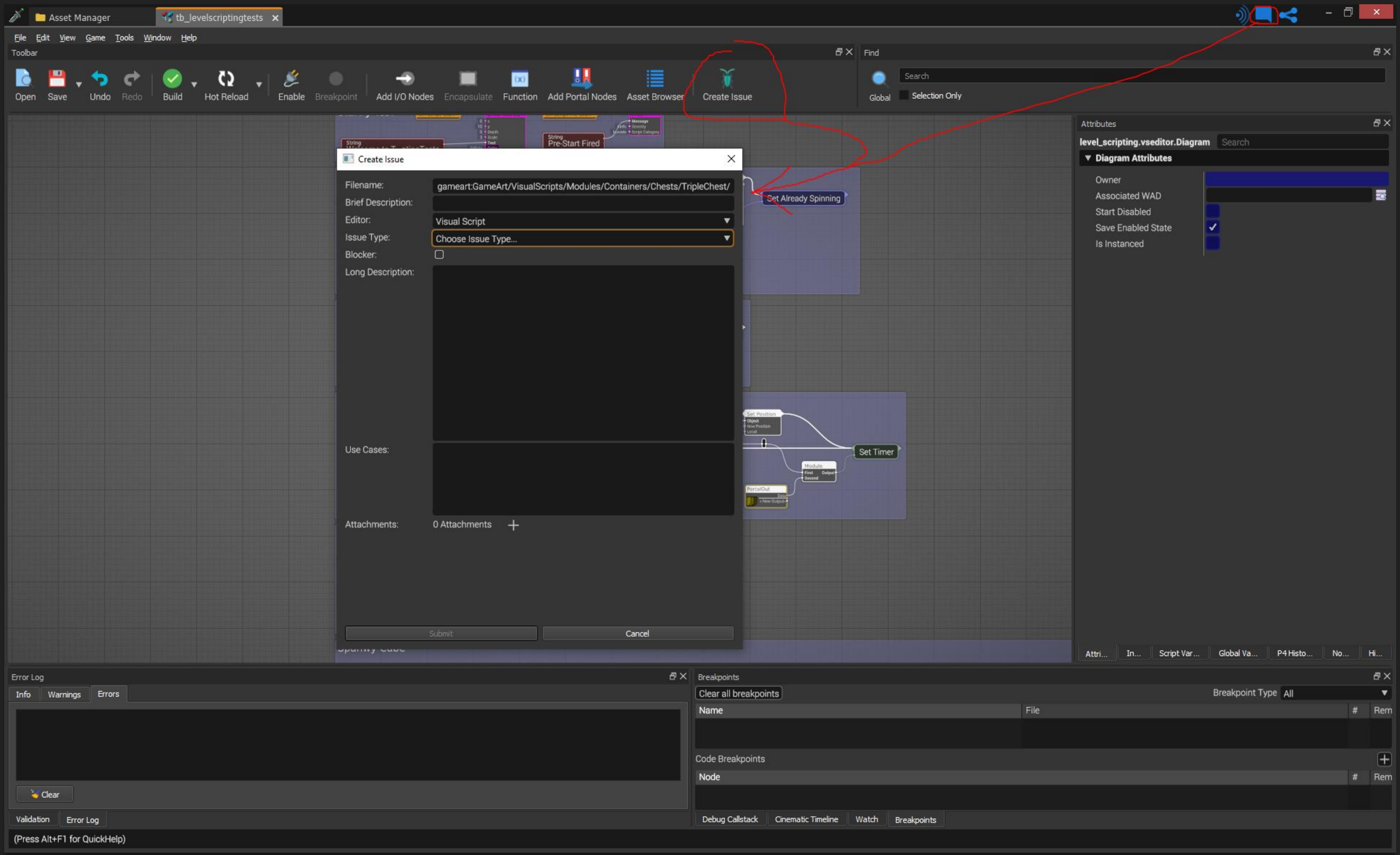
Managing Editor Requests

- Users provided feedback through team ambassadors, voting on common pain points
- Weekly triages of common issues with Design and Engineering both present
- Easily accessible bug reporting within the tool itself with encouragement to report anything seen

Santa Monica Studio

In order to manage the work, we had users provide workflow feedback through team ambassadors, who would organize votes on common pain points. Weekly triages of common issues were set up to ensure we were getting the most bang for our buck that we could. We also added an easily accessible bug reporting tool within the tool itself, which both tripled the number of bugs coming in and increased the quality of feedback that team ambassadors were getting – once the average user felt empowered to provide feedback at all, they were much freer with their thoughts

Managing Editor Requests – Create Issue Buttons



Create Issue

Filename:	gameart:GameArt/VisualScripts/Modules/Containers/Chests/TripleChest/
Brief Description:	
Editor:	Visual Script
Issue Type:	Choose Issue Type...
Blocker:	☐
Long Description:	
Use Cases:	
Attachments:	0 Attachments

level_scripting.vseditor.Diagram

Diagram Attributes

Owner	
Associated WAD	
Start Disabled	
Save Enabled State	✓
Is Instanced	

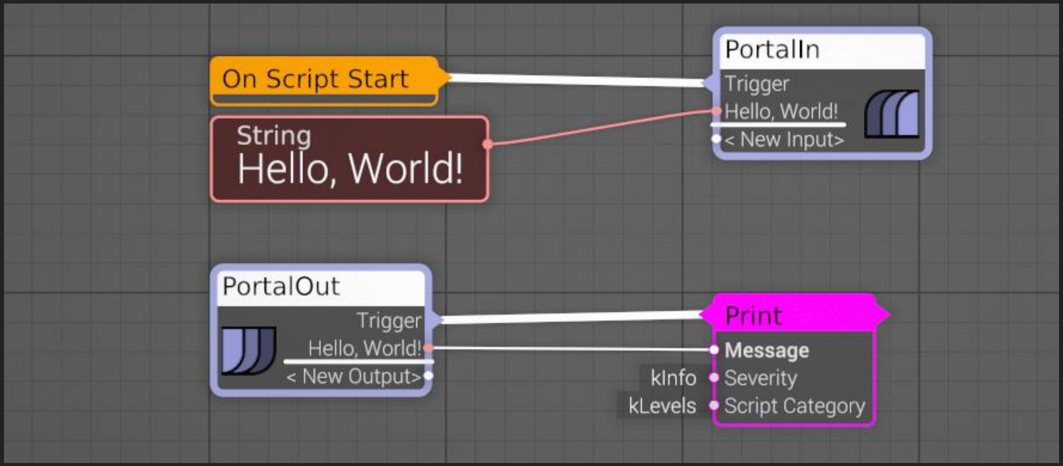
Breakpoint Type: All

When I said that the tool was easily accessible, I meant it – we had buttons within all common editor views, as well as a tool-wide report issue button. I spend time on this only because of how critical getting good feedback was to our general success.

Also, this is what our editor looks like, in case you were curious. These editor requests brought us a number of powerful tools features, which I want to show off a bit.

Editor Features – Portals

- Node graphs can become unruly, wires crossing can be hard to read, looks like literal spaghetti code
- Portals redirect logic for cleaner-looking scripts

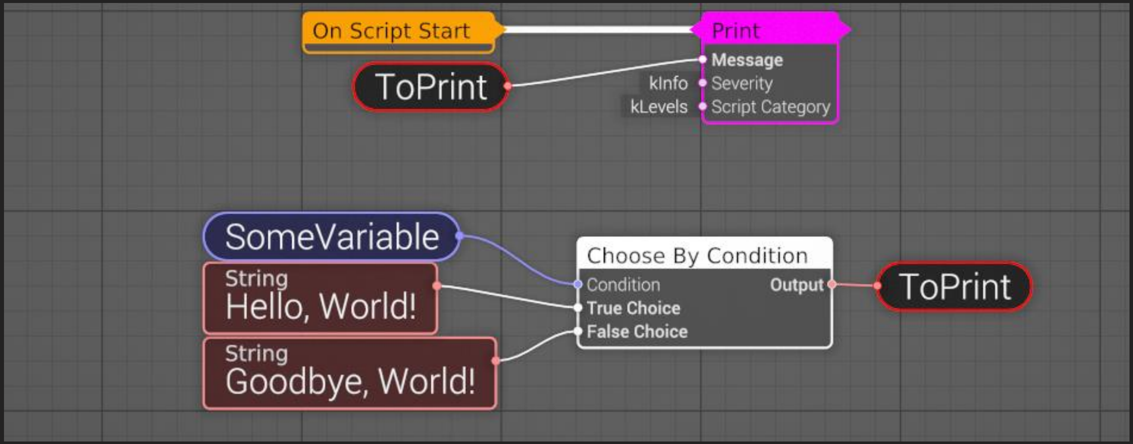


Santa Monica Studio

As anyone who regularly uses node-based scripting should know, they become a mess very quickly as people try to coordinate logic. To help us manage this, we created Portal nodes, which redirect logic from one part of the graph to another. They are purely syntactic sugar, completely removed at build time, but serve an important role at helping designers keep their scripts clean.

Editor Features – Data Portals

- Data, being one-to-many, gets special named portals
- Served as effective named references
- Quickly proliferated everywhere



In addition to general portal nodes, we also created specialized "data portals" which connect one source of data to as many outputs as the designers want. Again, they are syntactic sugar, but proved invaluable in maintaining script readability. They served as both miniature functions as well as a kind of helper variable, and ended up being one of the most used editor features.

A quick note about portal nodes is that they are not a silver bullet for messy scripts. Clean scripting will get cleaner with them, but poor scripting will be just as difficult to read, just a little more spread out. That said, they really pull a lot of weight towards improved readability.

Editor Features – Hot Reload

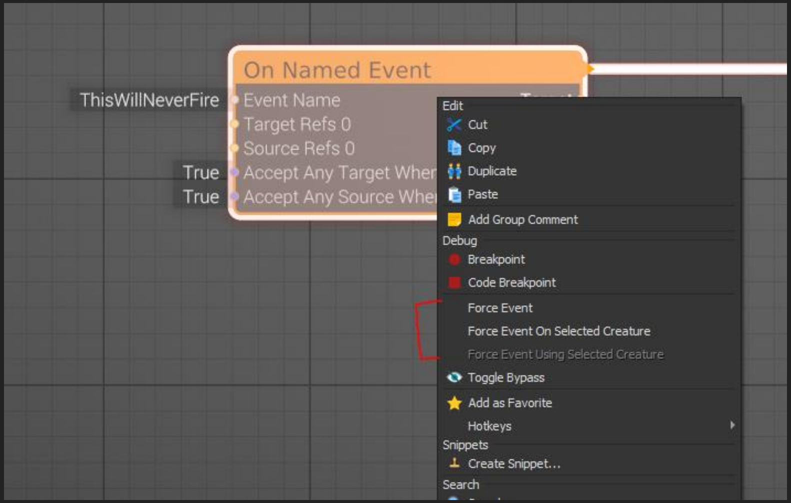
- Needed a quick way to test changes from editor
- Simply replaces any currently loaded version of the script with the new version
- Scripts only run at one point in the frame, and there's no coroutines/closures, so we can easily safely replace scripts

One of our biggest concerns was making sure our scripts could be effectively iterated on, as reloading the game every time you make a change is time-consuming, so we implemented hot-reload functionality that replaces the running copy of a script with an updated version of it.

Since scripting only runs at one point in the frame and always runs to completion, we just cache any hot-reload requests until right before we run all scripts for the frame, with no concerns for any "running" processes.

Editor Features – Force Event

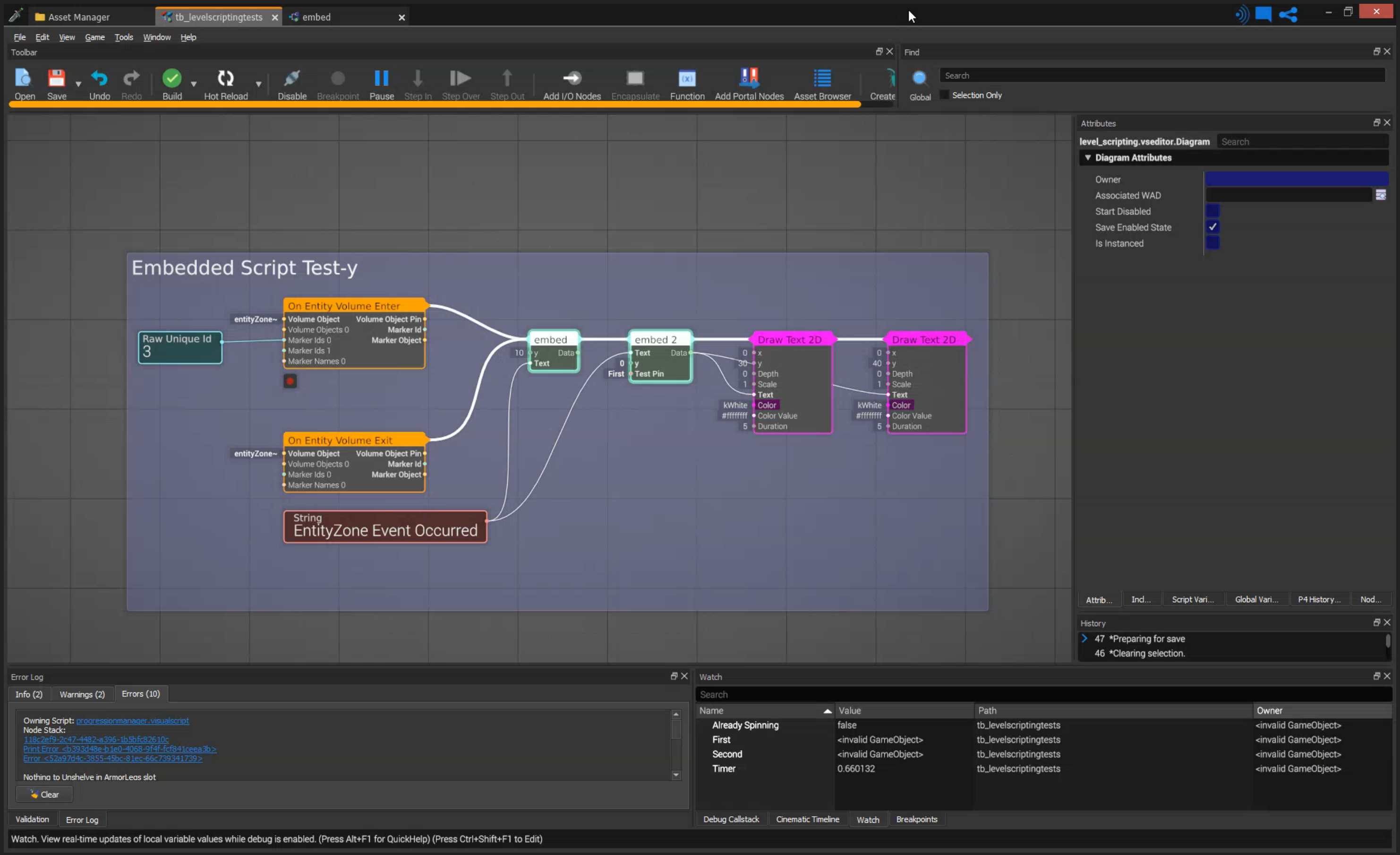
- Quick way to test behavior triggered by script
- Simply set up a test case, and force it to occur directly from the editor



Santa Monica Studio

Following the needs of Hot Reload, sometimes you want to iterate on a specific segment of scripting without performing all of the steps to get there like a real player. We created a way to directly force an event to run to aid that iteration. As we went to close the project, we found this feature invaluable.

Editor Features – Debugger



History
47 *Preparing for save
46 *Clearing selection.

Error Log

Info (2) Warnings (2) Errors (10)

```
Owning Script: progressionmanager.visualscript
Node Stack:
118c2ef9-2c47-4482-a396-1b5bfc82610c
Print Error <b393d48e-b1e0-4068-9f4f-fcf841ceea3b>
Error <52a97d4c-3855-45bc-81ec-66c739341739>

Nothing to Unshelve in ArmorLegs slot
```

Watch

Name	Value	Path	Owner
Already Spinning	false	tb_levelscriptingtests	<invalid GameObject>
First	<invalid GameObject>	tb_levelscriptingtests	<invalid GameObject>
Second	<invalid GameObject>	tb_levelscriptingtests	<invalid GameObject>
Timer	0.660132	tb_levelscriptingtests	<invalid GameObject>

Debug Callstack · Cinematic Timeline · Watch · Breakpoints

Watch. View real-time updates of local variable values while debug is enabled. (Press Alt+F1 for QuickHelp) (Press Ctrl+Shift+F1 to Edit)

Sometimes things don't work how you expect them to, and a debugger is a very powerful way to figure out why. We support a full debugger suite. We can put a breakpoint on any node or embedded script. Once hit, values show up for every pin in scope of the execution. We support stepping in to, out of, and over the currently running node, with a full debug callstack helping the user track exactly where they are and how they got there.

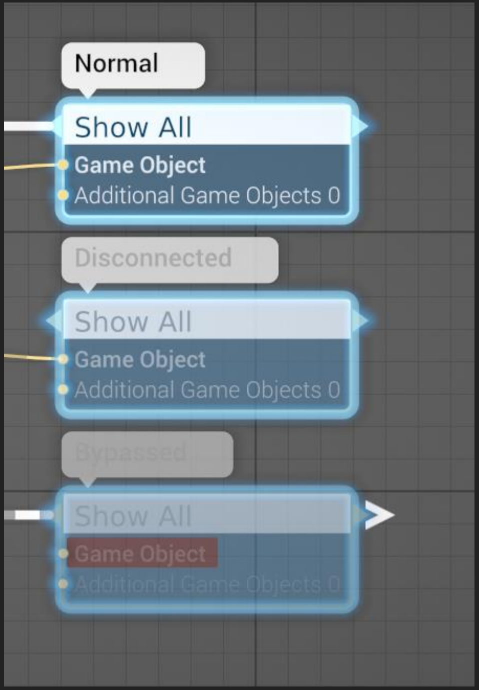
Some Notes About Debugging…

- Everyone does it
- Any time spent improving tools will yield dividends
- Stay in contact with power users for ideas
  - Breakpoint on Selected Object Only
  - "Code" Breakpoints

An important note about debugging is that everyone who touches scripting does it, designers and programmers alike. Your debugging capabilities are often limited by your debugging tools, so any time spent on providing stable and reliable introspection tools will pay off in the long run as you try to understand what state your game got into. As with everything else, we also found it incredibly useful to understand our power user's debugging workflows, where we could provide more esoteric logic like ony breakpointing when a node is hit relevant to the last selected object, or triggering a scripting breakpoint whenever a node of a certain type was hit in order to figure out where an errant usage of a node was coming from.

Editor Features – Bypass

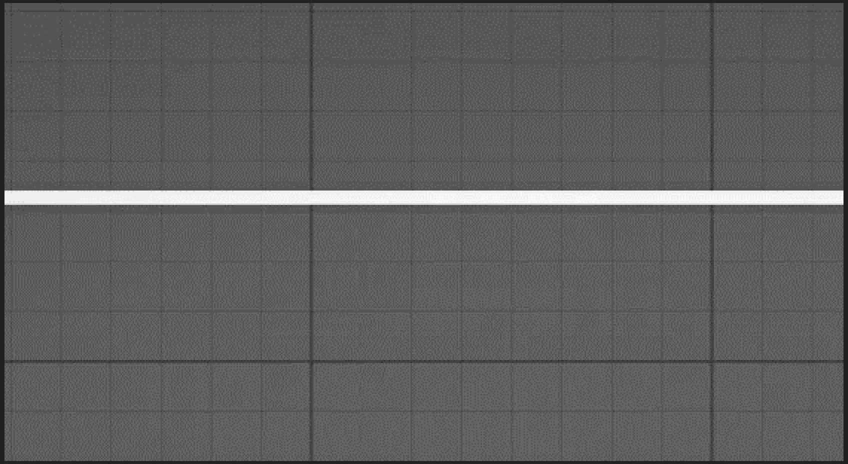
- "Commenting out" behavior takes a lot of clicks for a common operation
- Less user inputs for common operations is better
- Compiler just skips that node, treats as a no-op



As part of the creation of the editor we also created a lot of quality-of-life features One that we found particularly useful was our Node Bypass feature. "Commenting out" a node takes a lot of clicks for such a common operation what with disconnecting wires and connecting them elsewhere, and the less user inputs for common operations the better, so we just created a shortcut to do it. If a node is bypassed, it gets faded out and the compiler just skips that node entirely as if it was a no-op. The node also gets some markup on the flow itself to help signify it's bypassed state.

Editor Features – Node Insertion

- Adding behavior to the middle of a flow takes a lot of clicks for a common operation
- Less user inputs for common operations is better

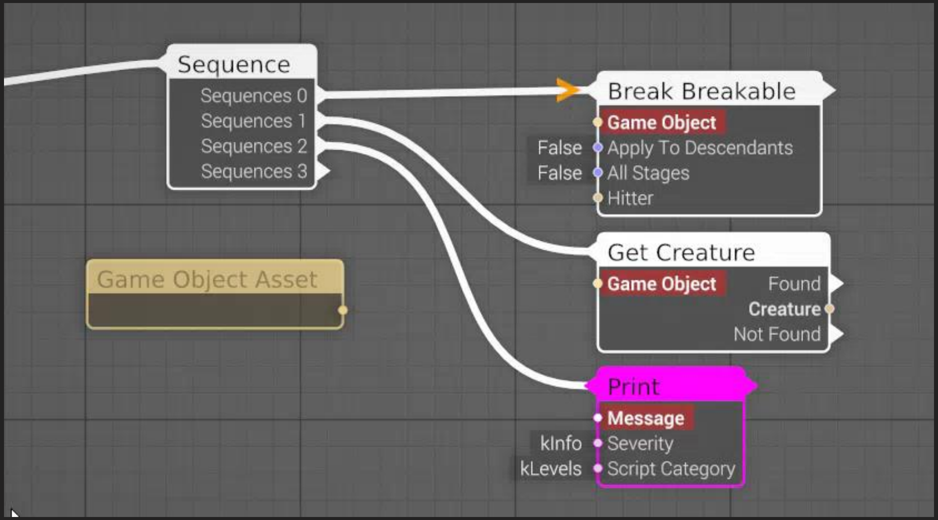


Santa Monica Studio

Another quality-of-life feature we added was a node insertion feature. Adding a node in the middle of an existing flow takes a lot of clicks, and as we said before the less user inputs required for a common operation, the better. By just holding control while dragging a node onto an existing wire, the node will be automatically inserted into the flow.

Editor Features – Multi-Connect

- Connecting one pin to a collection of nodes is time consuming and requires a lot of clicks
- Less user inputs for common operations is better

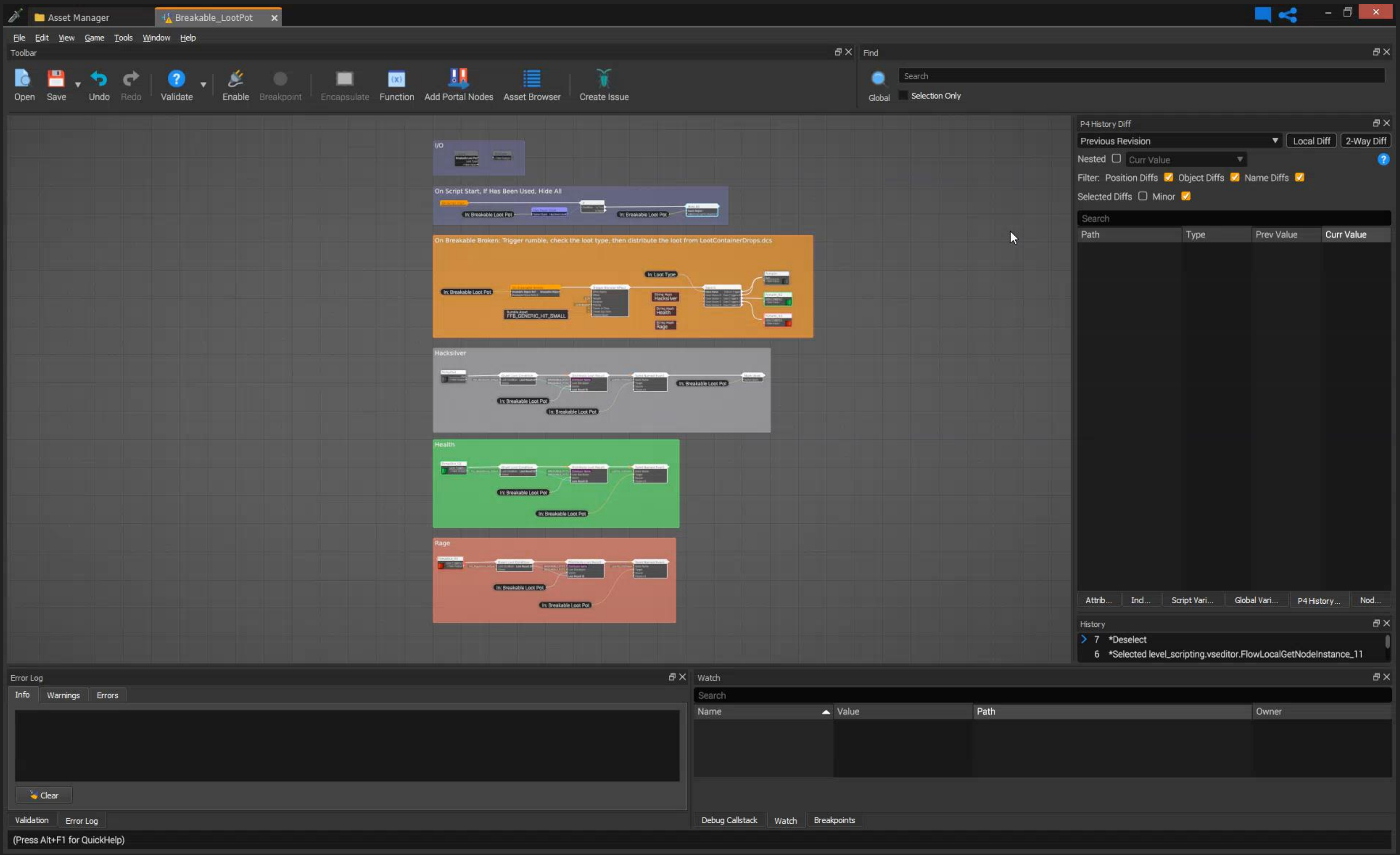


Santa Monica Studio

Yet another quality of life feature was our multi-connect feature. If you have to introduce some new connection or overwrite a bunch of existing connections, that's a lot of clicks. In a lot of places, we just used data portals to avoid this issue, but the desire for something better when we needed it still remained. At the risk of sounding like a broken record, the less user inputs required for common operations, the better. Being able to set up many connections at once helped take care of larger refactors when a module's core layout changed.

We could have chosen not to do any of these quality-of-life features, but it was important to us that we provided designers with the tools they asked for, even if we thought that it would be used by a small subset of power users. If the designers didn't believe they'd be able to get new features, how could they trust that we'd be able to provide critical support when they really needed it? You would also be surprised just how widely used many of these features were – just because you intuit that a feature is just for power users doesn't mean that you're right, it can be hard to predict what features become workflow staples.

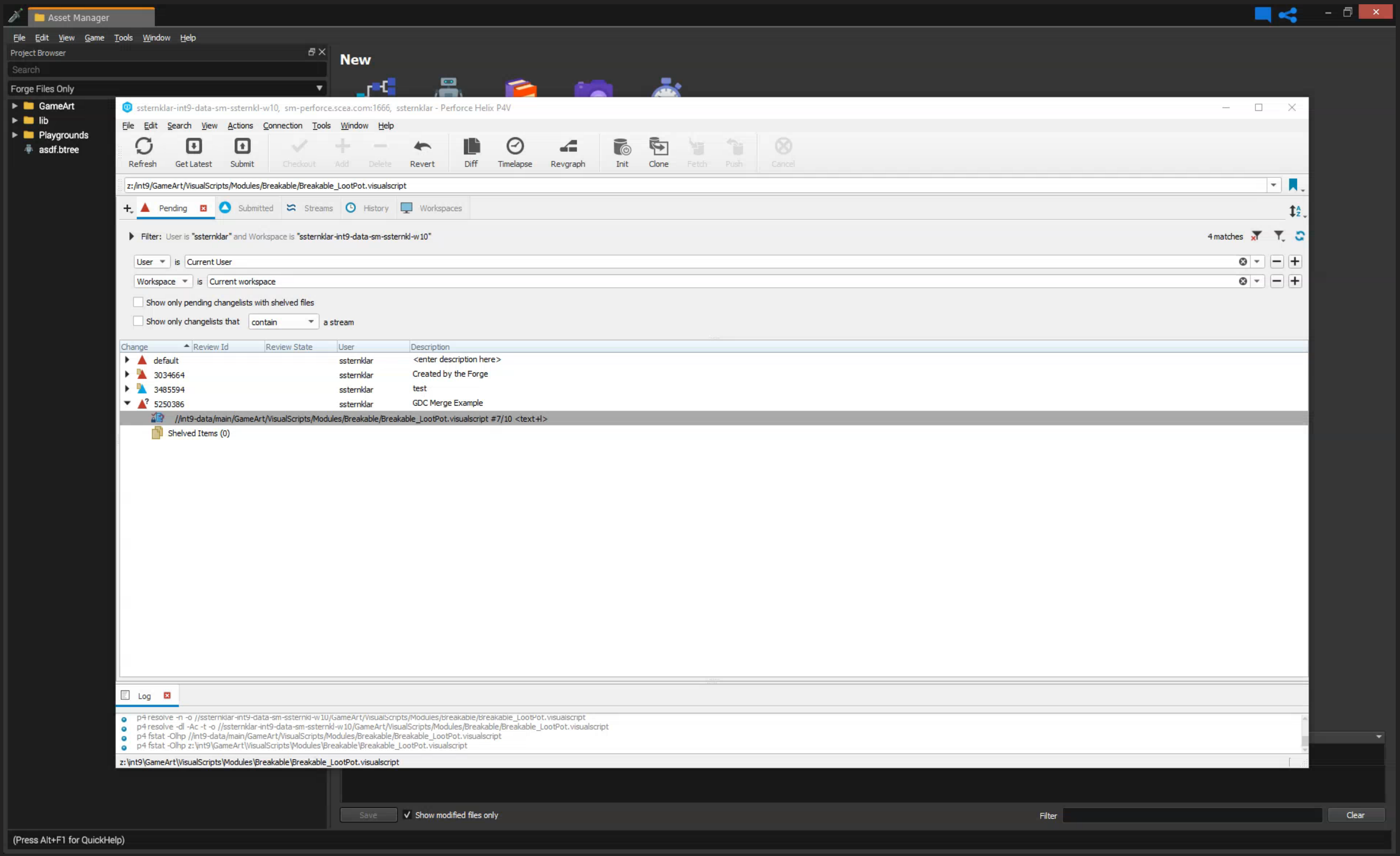
Editor Features – Diff



Asset Manager
Breakable_LootPot
File Edit View Game Tools Window Help
Toolbar
Open Save Undo Redo Validate Enable Breakpoint Encapsulate Function Add Portal Nodes Asset Browser Create Issue
Find
Search
Global
Selection Only
P4 History Diff
Previous Revision
Local Diff
2-Way Diff
Nested
Curr Value
Filter: Position Diffs Object Diffs Name Diffs
Selected Diffs Minor
Search
Path Type Prev Value Curr Value
Attrib... Ind... Script Vari... Global Vari... P4 History... Nod...
History
7 *Deselect
6 *Selected level_scripting.vseditor.FlowLocalGetNodeInstance_11
I/O
On Script Start, If Has Been Used, Hide All
On Breakable Broken: Trigger rumble, check the loot type, then distribute the loot from LootContainerDrops.dcs
Hacksilver
Health
Rage
Error Log
Info Warnings Errors
Clear
Validation Error Log
Watch
Search
Name Value Path Owner
Debug Callstack Watch Breakpoints
(Press Alt+F1 for QuickHelp)

As development proceeded it became obvious that we needed tools that let us track how a script evolved over time. With code we use diff tools, so we created a diff tool for our visual scripting system. At any point, a user can select what revisions they want to compare against in a special diff window, and get a printout with highlights of all changes from a prior version of the script. If they so desire, they can open a more traditional two-window diff tool that will directly show the older version of the script as well. Both sides of the diff mirror any changes, whether that's changes in the diagram's viewport, what auxiliary window is shown, or what nodes are selected.

Editor Features – Merge



z:/int9/GameArt/VisualScripts/Modules/Breakable/Breakable_LootPot.visualscript

Filter: User is "ssternklar" and Workspace is "ssternklar-int9-data-sm-ssternkl-w10"

User is Current User

Workspace is Current workspace

☐ Show only pending changelists with shelved files

☐ Show only changelists that contain a stream

Change	Review Id	Review State	User	Description
default			ssternklar	<enter description here>
3034664			ssternklar	Created by the Forge
3485594			ssternklar	test
5250386			ssternklar	GDC Merge Example
//int9-data/main/GameArt/VisualScripts/Modules/Breakable/Breakable_LootPot.visualscript #7/10 <text+l>				
Shelved Items (0)				

```
p4 resolve -n -o //ssternklar-int9-data-sm-ssternkl-w10/GameArt/VisualScripts/Modules/Breakable/Breakable_LootPot.visualscript
p4 resolve -dl -Ac -t -o //ssternklar-int9-data-sm-ssternkl-w10/GameArt/VisualScripts/Modules/Breakable/Breakable_LootPot.visualscript
p4 fstat -Olhp //int9-data/main/GameArt/VisualScripts/Modules/Breakable/Breakable_LootPot.visualscript
p4 fstat -Olhp z:\int9\GameArt\VisualScripts\Modules\Breakable\Breakable_LootPot.visualscript
```

z:\int9\GameArt\VisualScripts\Modules\Breakable\Breakable_LootPot.visualscript

✓ Show modified files only

It turns out that once you have a suitable diff suite, merging is not too far behind. A user can start a merge from perforce like they would a text merge, using a wrapper to forward commands to the currently running instance of the editor. Editing of the to-be-merged script is locked while the merge is underway. We go through and create diffs of the base-to-target, base-to-source, and target-to-source versions of the file. If something was added in the target, but does not conflict with anything in the source, it can be safely added to the merged document, and vice versa. If however, something has changed in both target and source, and they are not identical, that is a conflict. If conflicts arise, they are displayed to the user to resolve. A user can then make further edits to the merged script to ensure everything is correct before it is saved out, and the user can return to perforce to complete the merge.

Some Notes About Diff/Merge…

- Merging data is hard and dangerous
- Have strong editor abstractions before doing it
- Time and labor intensive
- Worth it

Santa Monica Studio

There are many complexities to the merge process, including what occurs when multiple interdependent files have conflicts, but thankfully our editor abstractions can gracefully handle missing data most of the time. If you ever determine that it can't be gracefully handled, you have a maximum-priority issue that requires an immediate fix or else you risk breaking user data. I say this because we ran into one very late in development, and it jeopardized trust in the merge tool as a whole. One thing that we learned is that if your editor model and runtime model diverge at all, do the merge with the editor model's logic. Your merge tool is operating on editor data, and must understand the rules surrounding your editor, even if it is conceptually simpler to do it as your runtime expects. This was one of the features with the longest running total development time, clocking in at almost three months just to get it stood up with even more time dedicated to testing and bug fixing, but it was truly priceless to development to get away from needing to exclusive-checkout our visual scripts. It is hard to overstate how much more effective we were after this tool's creation, especially once we began making dedicated branches for playtests and release candidates.

The Last Editor Slide

- This was a ton of work!
- Required a dedicated engineer and several other rotating contributors
- Our situation was unique
- Do not underestimate how much time to budget!

Santa Monica Studio

As stated earlier, this was a ton of work. In addition to a rotating group of talented contributors, it required a dedicated engineer for the entire project, to the point where even when they were low on work our leads and directors refused to move them off due to how much work could be added at any point. Our situation and ability to provide for our designers feels somewhat unique, given that we didn't have to build the entire editor framework ourselves, so even though it was a lot of work to build these features it wasn't nearly as much as if we needed to start from scratch.

Part 4: The Results

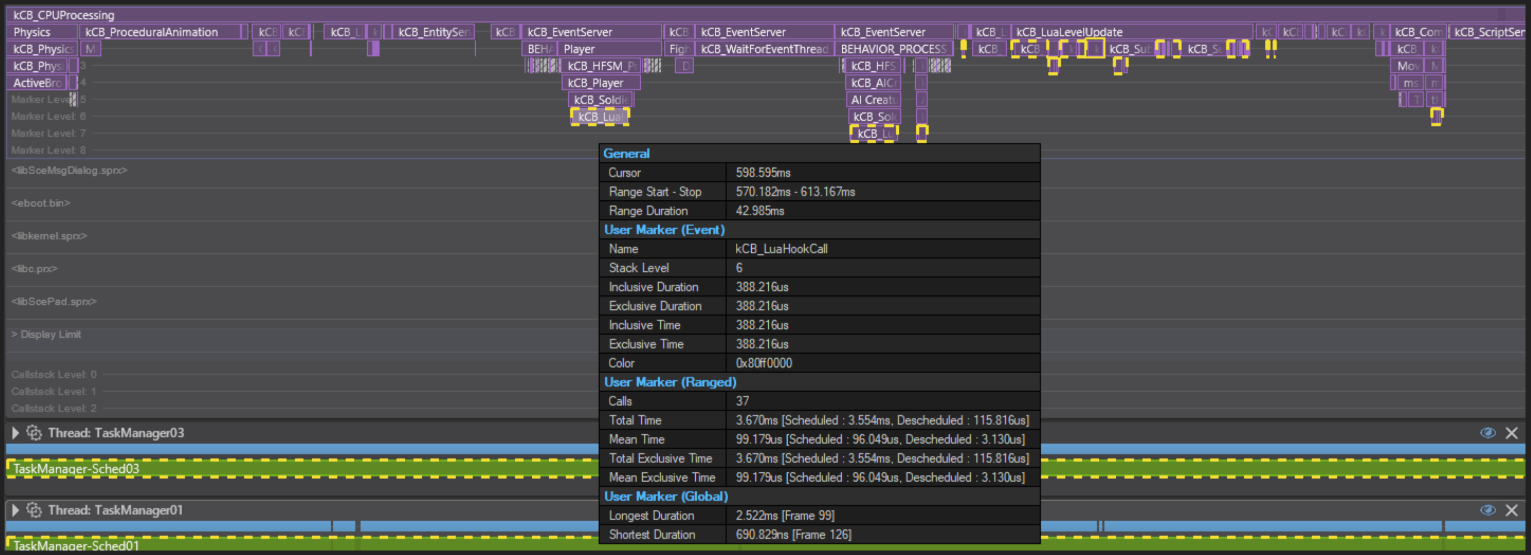
The part with performance details

Santa Monica Studio

So with all this talk of principles and editor features done, you may be wondering if we accomplished what we set out to do. We already spoke at length about the workflow improvements, but I haven't touched much on performance numbers. I apologize, I had to build suspense somehow!

So did it work?

- Previous Script Budget: Std Avg 3.5ms, Max Avg 5ms

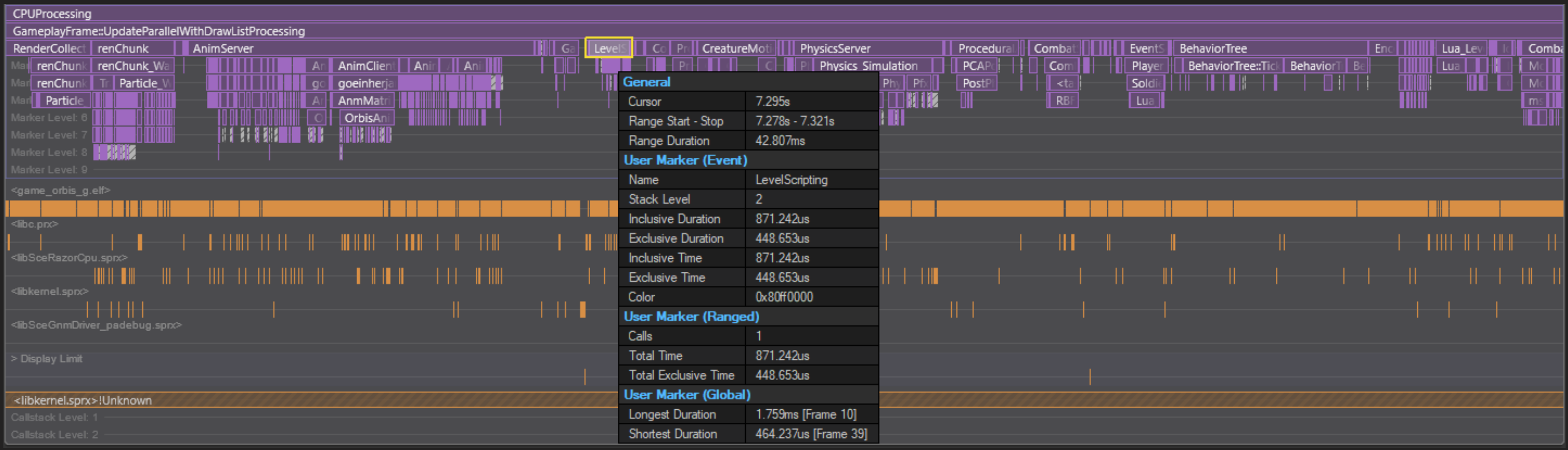


General	
Cursor	598.595ms
Range Start - Stop	570.182ms - 613.167ms
Range Duration	42.985ms
User Marker (Event)	
Name	kCB_LuaHookCall
Stack Level	6
Inclusive Duration	388.216us
Exclusive Duration	388.216us
Inclusive Time	388.216us
Exclusive Time	388.216us
Color	0x80ff0000
User Marker (Ranged)	
Calls	37
Total Time	3.670ms [Scheduled : 3.554ms, Descheduled : 115.816us]
Mean Time	99.179us [Scheduled : 96.049us, Descheduled : 3.130us]
Total Exclusive Time	3.670ms [Scheduled : 3.554ms, Descheduled : 115.816us]
Mean Exclusive Time	99.179us [Scheduled : 96.049us, Descheduled : 3.130us]
User Marker (Global)	
Longest Duration	2.522ms [Frame 99]
Shortest Duration	690.829ns [Frame 126]

As a reminder from the start of this talk, we started from a place where we spent an average of 3.5ms per frame on lua scripting just standing still, with an average combat maximum of 5ms per frame.

So did it work?

- Prev Script Budget: Std Avg 3.5ms, Cmbt. Avg 5ms
- Current Script Budget: Std Avg 1ms, Cmbt. Spike 2ms
  - *plus another ~1ms of remaining lua



General	
Cursor	7.295s
Range Start - Stop	7.278s - 7.321s
Range Duration	42.807ms
User Marker (Event)	
Name	LevelScripting
Stack Level	2
Inclusive Duration	871.242us
Exclusive Duration	448.653us
Inclusive Time	871.242us
Exclusive Time	448.653us
Color	0x80ff0000
User Marker (Ranged)	
Calls	1
Total Time	871.242us
Total Exclusive Time	448.653us
User Marker (Global)	
Longest Duration	1.759ms [Frame 10]
Shortest Duration	464.237us [Frame 39]

Santa Monica Studio

At the time of shipping God of War: Ragnarok, we have an average standing-still scripting budget of 1ms, which we rarely use all of, with combat spikes to 2ms. It is worth noting that we did not eliminate all lua, which I will cover momentarily, but we still see an almost 45% drop in processor time for the lua we did replace. You can see in the capture that for the selected frame, from a capture in combat, the selected frame took 0.8ms, with a maximum spike over the course of the capture of 1.76ms.

How about memory?

- Previous Average Lua Heap Size: ~3.1mb
- Current Average Visual Script Heap Size: ~0.7mb

- Previous Worst Case Lua Heap Size at ~20mb
- Current Worst Case Visual Script Heap Size at ~6mb

Santa Monica Studio

Okay, so performance is great, how about memory budget? Our previous average lua heap was around 3.1mb per level, with our current average VS heap at around 700kb. Our previous worst case lua heap not including the UI was almost 20mb, with our worst case visual script heap is at around 6mb. This doesn't come as much of a surprise due to how much we had to allocate to deal with the spiky heap allocation of lua, but it's nice to know we did well! We also cheat this number quite a bit with our shared visual script stack buffer, but even accounting for the extra megabyte we are still far better than any of our standard lua cases.

It appears to have worked!

- 🎉 The game shipped successfully! 🎉
- It's fast enough!
- Late-stage optimization wasn't too dangerous!
- We hit a smaller budget most of the time!
- We had noticeably less bugs come from script instability!
- Our designers don't totally hate us!

Overall, this initiative seems to be a massive success. Not only did our game with More Stuff ship successfully, our visual scripting performed faster than our previous lua implementation and with less memory. We were able to optimize systems surrounding scripting in a way we weren't able to last time. Furthermore, we had far less bugs from script instability, all while our designers didn't totally hate us for doing it! All must be well, right?

Caveat Emptor

- 🎉 The game shipped successfully! 🎉
- It's fast enough!
- Late-stage optimization wasn't too dangerous!
- We hit a smaller budget most of the time!
- We had noticeably less bugs come from script instability!
- Our designers don't totally hate us!

Santa Monica Studio

Oh, how I wish that were true!

Is it fast?

- Cutting our frame utilization by that much implies our interpreter is fast

Santa Monica Studio

The first thing to address is the question of how fast our interpreter actually is. With such a large reduction you'd think that some of it comes from smart decisions in the interpreter making it somewhat fast at executing script

Is it fast?

- Cutting our frame utilization by that much implies our interpreter is fast
- It is not

Perf/Language on PS4	Hello World (x1000)	Prime Sieve (N=100000)	Mandelbrot (800x800)
C++/x86	2.807ms	0.871ms	498.7ms
Lua 5.2 (modified)	3.258ms	43.33ms	11676ms (11 seconds)
SMS Visual Scripting	3.565ms	171ms	347017103ms (347 seconds)

Santa Monica Studio

I can say it is not. The numbers in the table you see here are taken from a shipping version of the game running on a standard PS4 devkit, with the only modifications being what I needed to set up and run the benchmarks. You can see that in every case our visual scripting is orders of magnitude slower than C++ and lua alike, the difference increasing with the scale of computation. Each benchmark implementation was as close to identical as possible, as I didn't want to just show off how fast SIMD instructions were in C++. Suffice to say, our visual scripting is slow at general computation.

So where does the speed come from?

- Design's better event-based practices
- Pre-interpreter filtering efficiently skipping work
- Aggressive pre-interpreter filter acceleration
- No Garbage Collection

- **General Compute != Your Scripting Workload**

Santa Monica Studio

So where does the speed come from? As mentioned before, we set up our entire system to help design teams script better. Design effectively leveraged event-based practices, meaning we ran less code overall on any given frame. Of the code that did run each frame, aggressive pre-interpreter filtering and acceleration thereof prevented the vast majority of "waste" work from occurring. And we didn't spend one nanosecond of visual script time on garbage collection, meaning both our average frame is lower AND we know deterministically what's happening on any particular frame.

We planned for and expected all of this from the very start, as laid out in our core principles. It just goes to show that while games do billions of complex calculations a second, general compute is not always a great analogy for your game's scripting workload, and you need to consider what you can do with your scripting environment as a whole alongside your general interpreter semantics.

So where does the speed come from?

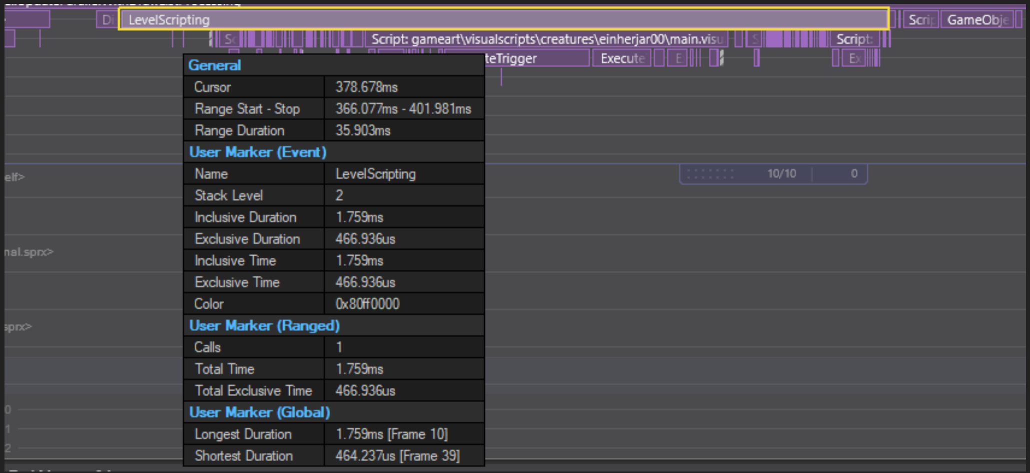
- Eventization allowed us to optimize other systems
  - Scripting being deferred allowed us to move things around without potential script timing issues
  - Events could be sent from other threads safely
- Remaining 1ms of time is either system overhead or Ticks that we couldn't get rid of in time

Santa Monica Studio

It's also worth pointing out that while it's not reflected in the numbers I've shown you directly, eventization allowed us to optimize other systems entirely. We were able to restructure entire systems safely with respect to scripting in a way that we couldn't before, and could even move entire systems to other threads. A great example of this is our trigger volume system (which we refer to as Entity Volumes), which was previously locked to the main thread but due to the eventization was able to be moved entirely to a worker thread without disrupting scripting at all! Most of the remaining 1ms of our average frame is a combination of static overhead to manage the acceleration structures, and ticks that we just couldn't remove.

Did we hit budget?

- Yes, mostly
- Sometimes heavy workloads were unavoidable
- Combat could cause ~1ms spikes


LevelScripting
Script: gameart\visualscripts\creatures\einherjar00\main.visu
teTrigger
Execute
GameObje
10/10

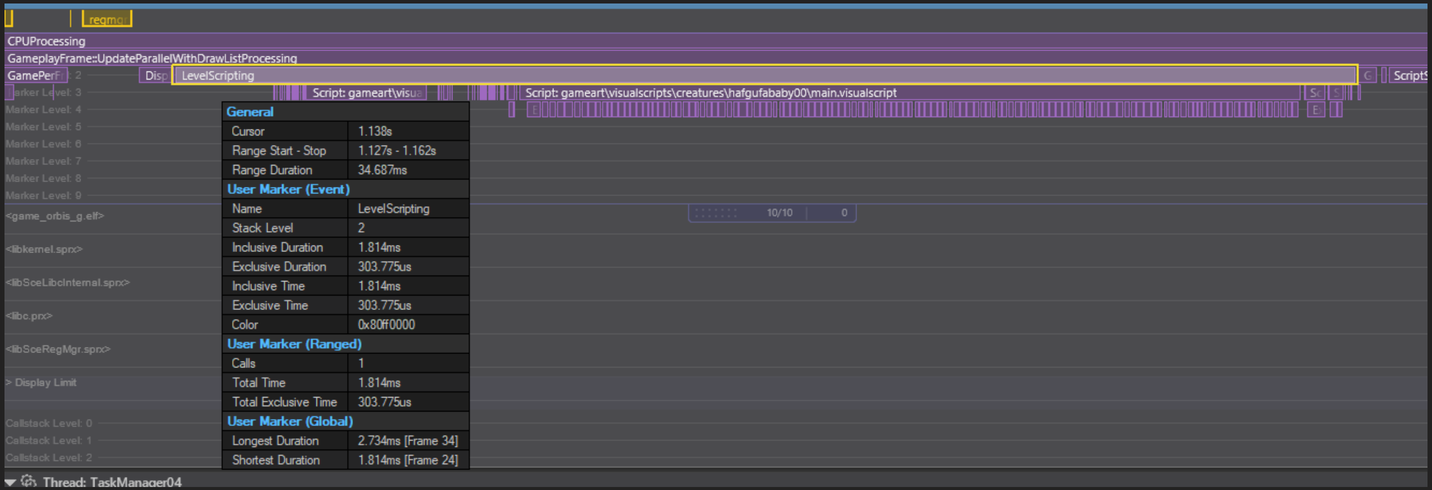
General	
Cursor	378.678ms
Range Start - Stop	366.077ms - 401.981ms
Range Duration	35.903ms
User Marker (Event)	
Name	LevelScripting
Stack Level	2
Inclusive Duration	1.759ms
Exclusive Duration	466.936us
Inclusive Time	1.759ms
Exclusive Time	466.936us
Color	0x80ff0000
User Marker (Ranged)	
Calls	1
Total Time	1.759ms
Total Exclusive Time	466.936us
User Marker (Global)	
Longest Duration	1.759ms [Frame 10]
Shortest Duration	464.237us [Frame 39]

Santa Monica Studio

In general, we were able to hit our budget. In certain segments we just had to pay the cost of extra work. Combat, for example, caused some frequent 1ms spikes while resolving events. You can see an example of such a frame in the razor capture here, where an einherjar is resolving some expensive damage event against multiple hit targets. For the most part, the 1ms spikes were eaten by general slack we had in the frame to compensate for such one-off workloads.

Did we really hit budget?

- Some areas were just naturally more expensive
- Usually negotiated frame budget from other systems



General	
Cursor	1.138s
Range Start - Stop	1.127s - 1.162s
Range Duration	34.687ms
User Marker (Event)	
Name	LevelScripting
Stack Level	2
Inclusive Duration	1.814ms
Exclusive Duration	303.775us
Inclusive Time	1.814ms
Exclusive Time	303.775us
Color	0x80ff0000
User Marker (Ranged)	
Calls	1
Total Time	1.814ms
Total Exclusive Time	303.775us
User Marker (Global)	
Longest Duration	2.734ms [Frame 34]
Shortest Duration	1.814ms [Frame 24]

Santa Monica Studio

There are occasionally, however, some areas that are always that bad. This example, sitting at an average of 2.1ms for the entire scripting frame. was taken standing still in the middle of the a fairly open area after completing a late-game sidequest, where we had 125 instances of a certain entity constantly processing to update their position. Normally, 125 scripts wouldn't be of too much concern with our aggressive acceleration and pre-filtering, but each of them had to tick, meaning we couldn't do anything to avoid the cost of running them. However, this is our absolute worst "average" case, and we were able to negotiate some budget from other systems that weren't using their full budget in the affected area.

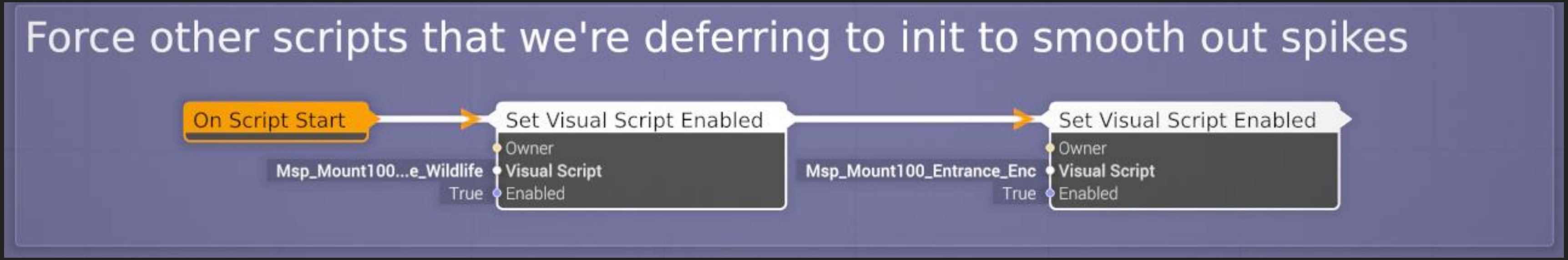
Did we *really* hit budget?

- The worst case of our system is high compute cost that cannot be filtered
- Script initialization after loading is such a cost
  - Reset internal module states
  - Error-check progression state

So like OnTick, any place that we can't do effective filtering means we lose most of the benefits of our system. The whole thing was built assuming that we could elide more work than we performed, after all. While we did a good job generally getting rid of ticks, there's another major contributor of such events: Script initialization. On init, scripts have to handle a lot of state in ways that can't always effectively be moved to code. Whether it's resetting internal states or error-checking the game progression, the work simply must be done.

Did we *really* hit budget?

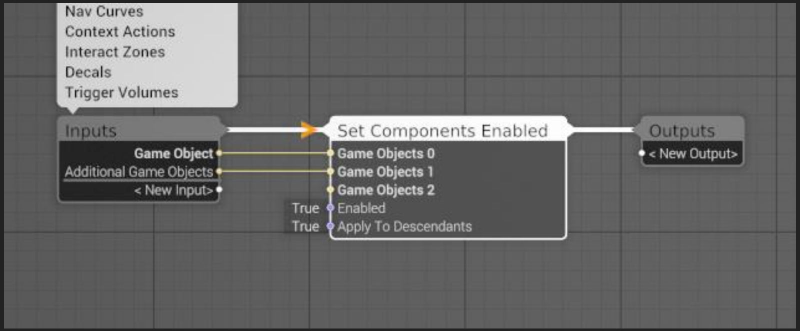
- On Script Start can't be filtered, caused 5-7ms spikes
- **Worst cases approached 60ms spikes**
- "Solved" the problem by breaking scripts up and spreading safe flows over multiple frames


Force other scripts that we're deferring to init to smooth out spikes
On Script Start
Set Visual Script Enabled
Owner
Msp_Mount100...e_Wildlife
Visual Script
True
Enabled
Set Visual Script Enabled
Owner
Msp_Mount100_Entrance_Enc
Visual Script
True
Enabled

On average, this caused spikes of anywhere from 5-7ms, At it's worst, this meant almost 60ms spikes on base PS4 when scripts had to perform a lot of work. There are a lot of reasons for why we got here, from our engine not having great ways to define initial states, to some questionable behaviors in our save state management over the course of development leading to an over-reliance on scripting to perform fixups. At the end of the day, the work had to be done, so we "fixed" it by smearing less-important script initialization across multiple frames to make the spikes less severe, as you can see in the image below. The proper fix to this requires a deep dive into why we have so much scripting running on init, and seeing what we can do to fix it for whatever we do next.

How did we optimize it?

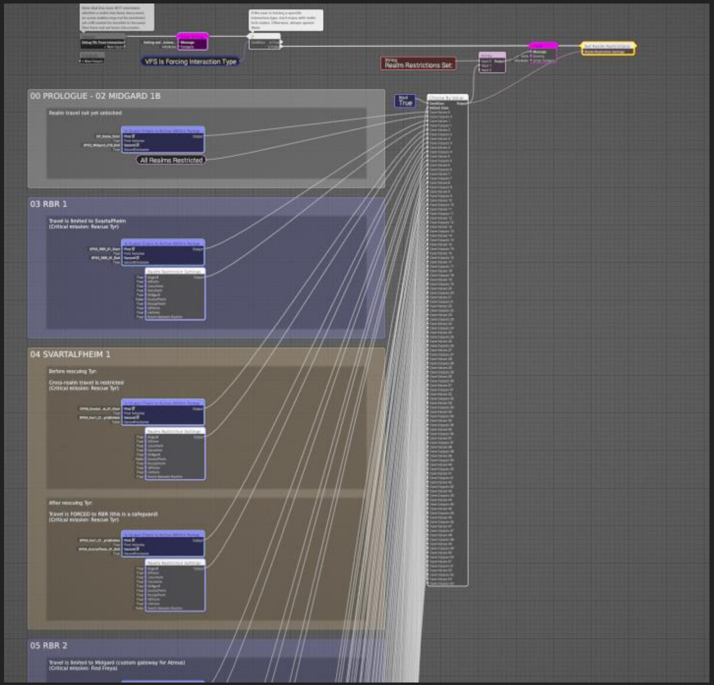
- Expensive embedded scripts replaced with cheaper nodes that batched operations
- "Outside the Interpeter" includes "Inside of a Node"
- N nodes -> 1 node removes N-1 nodes of overhead



The natural next question is to ask how we optimized any of our scripting. One of our principles was that we wanted to spend as little time in the interpreter as possible, as well as have a good encapsulation of common operations. The earlier "Show All" embedded script is a great example of our efforts, because it helps show that "outside the interpreter" is flexible. By moving a set of nodes into one code-driven node, we can avoid hundreds of script operations' worth of overhead, which is less time spent interpreting and more time spent doing important logic.

How did we optimize it?

- Expensive scripts were given special attention and reworking by programmers
- Some expensive scripts moved completely into code



00 PROLOGUE - 02 MIDGARD 1B

Realm travel not yet unlocked

03 RBR 1

Travel is limited to Svartalfheim
(Critical mission: Rescue Tyr)

04 SVARTALFHEIM 1

Before rescuing Tyr:

Cross-realm travel is restricted
(Critical mission: Rescue Tyr)

After rescuing Tyr:

Travel is FORCED to RBR (this is a safeguard)
(Critical mission: Rescue Tyr)

05 RBR 2

Travel is limited to Midgard (custom gateway for Atreus)
(Critical mission: Find Freya)

Santa Monica Studio

As always, this wasn't a silver bullet. Certain expensive scripts got special attention by programmers, as you can see to the right with this special-case node with over 100 inputs for a very specific evaluation. And some scripts got moved completely into code, such as the crank script that drives a number of puzzles and ran into complexity issues, the breakable pots that are peppered throughout the world for causing too much overhead in the scripting system with over 600 running scripts at once, and the water sluice used in specific puzzles because the script was so large that any attempt to optimize it's raw instructions' size would have taken away from other efforts. While we like to minimize the amount that we comandeer ownership of design's modules, we recognize that it's sometimes necessary to get things to ship.

How did we optimize it?

- Where relevant, added new pre-interpreter filters to avoid over-executing flows
  - Refactoring many scripts took a lot of time
  - Had to be careful not to make general cases worse by adding filters for specific situations
- Profiler-guided optimization efforts to the interpeter

Where we could, we also added new pre-interpreter filters. While we have a decent idea about what will be checked when we expose a new event by just checking what data it outputs, we aren't perfect and sometimes we miss things. This sometimes meant refactoring large segments of scripts, and there were cases where we ended up making other areas worse by adding extra filters that we couldn't efficiently accelerate. And lastly, as with any optimization effort, we relied on profiler data to tell us what parts of the interpreter were the slowest, and did our best to address the worst offenders. For the most part, this was the management of the stack, fetching values when we needed them and calling the actual node functions with them. We never attempted common interpreter optimizations like tail call elimination in the core interpreter loop, reverse postordering of interpreter registry entries, or any sort of just-in-time operation elision, entirely out of lack of necessity – the mere act of fetching values itself proved to dominate every other possible optimization.

Was it stable and safe?

- Once it was fully stood up, it was stable
  - Assetization allowed robust trackable errors
  - Negligible crash rate
- But we still had many logical errors
  - Does not crash != Does what you want

Santa Monica Studio

The good news is that for the most part, the interpreter was stable. Compared to lua, where a single missing check brought down the VM, we have a truly negligible crash rate with visual scripting. We were able to gather robust information about which scripts were referencing missing assets or were performing invalid operations on otherwise valid assets, allowing us to stop a ton of bugs before they occurred. The majority of scripting crashes in the game come from the few remaining usages of lua in the engine.

The unfortunate side of this is that we didn't do much to affect the rate of logical errors. There were still a lot of places where blocked progression occurred due to scripting errors, like where someone exited a puzzle in an unexpected manner. The implication here is that we're still missing tools to help our design partners structure their logic in ways that reduce errors, and that future advancements must consider how we encourage our designers to structure logic in the system. At the end of the day, our QA progression percentage over time looked similar between God of War (2018) and God of War: Ragnarok primarily due to these logical errors, which is something we need to take a hard look at. We aren't keen on repeating this a third time.

Designer Perspective – Positives

- Some of our less technical and newer designers did not feel comfortable with text-based scripting
- For the groups that were included in initial discussions, there's positive sentiment
- Increased runtime stability and commitment to constant improvement built trust

Santa Monica Studio

At the end of the day, all of these tools are primarily for designer use, and feelings regarding our scripting system among our design departments are mixed. These mixed feelings come from almost every department, and vary wildly from person to person. On the positive side, visual scripting is more approachable for less technical designers. These days, it's not rare to find designers who may feel uncomfortable being handed a general computing language and being told to get to work, and they greatly appreciated the move. For the design groups involved in initial discussions, we find generally more positive sentiment as their needs were directly considered and thus experienced less friction. The increased runtime stability and our commitment to constant improvement built trust over time that also garnered positive sentiment. These are all the things we planned for, and the things we planned for went well!

Designer Perspective – Negatives

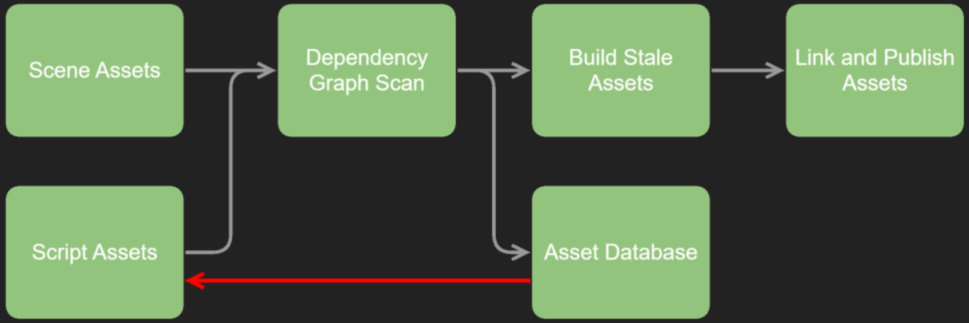
- Build/DB iteration times were frustrating
- Not all design groups were included initially
- Fidelity issues stemming from deferred events
- Some designers don't like visual scripting at all

Santa Monica Studio

Unfortunately, we could not plan for everything. The aforementioned build system and asset database iteration times were a frustration that left many designers wanting for a better solution, and generated negative sentiment towards the tools. Not all design groups that ended up using visual scripting were included in those early discussions, and because of that they had needs that weren't met. On multiple occasions we were left scrambling for lost time trying to help those groups get stood up, and had we known they would be using the tool, we could have planned for them better. And finally, for our combat team in particular, some of our decisions around eventization created unexpected gameplay fidelity issues throughout the project. And some designers just dislike visual scripting as a paradigm. As with how the things we planned for went well, the things we did not plan for went poorly.

We didn't plan for: Our Asset DB Workflow

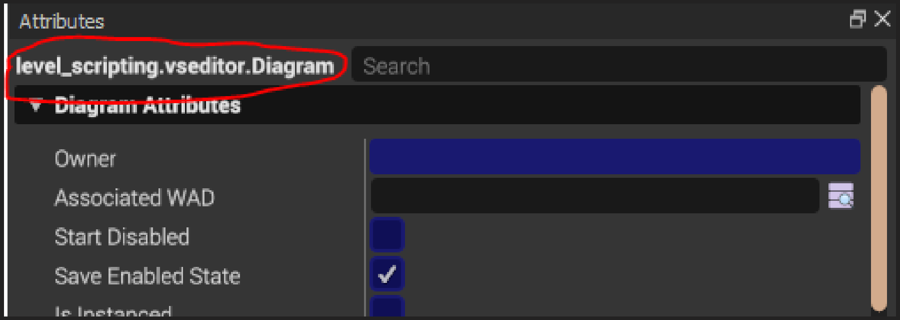
- Our tacked-onto-build DB was not originally shipping
- Were unable to correct with a more general system
- Lesson: Don't bet on something that doesn't exist, plan to ship your interim solution!



An interesting fact about our Asset DB is that it wasn't originally intended to be a shipping solution. We had intended to replace the tacked-onto-build workflow with a system that would dynamically listen for changes in the scene and quickly update themselves, rather than require constant rebuilds. Unfortunately, project needs shifted, and the more general DB solution never materialized. The only reason that we were successful at all shipping our interim solution is that, from a relatively early point in the project, we anticipated that a production-ready backup solution might become necessary, and put the requisite work into it to make it a serious piece of engineering. While the shipping workflow was slow due to dependency on certain aspects of the existing generalized build system to reference final game assets, the fact that we planned for the worst while hoping for the best is one of the main reasons it shipped at all.

We didn't plan for: Widespread Usage

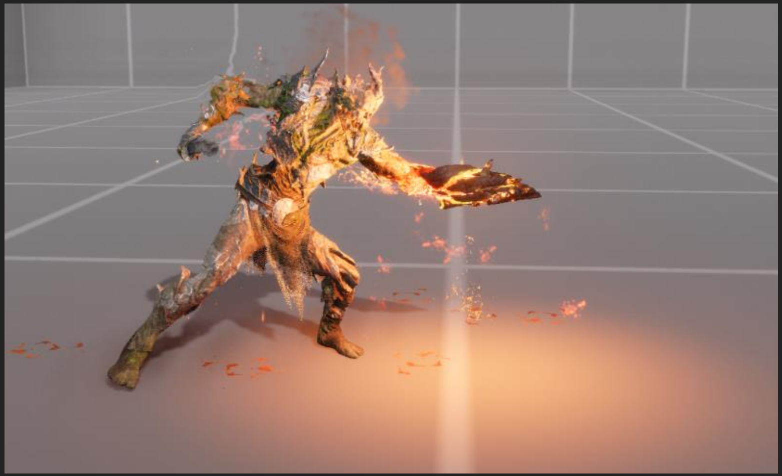
- Initially, this tool was built for level design only
- Had to scramble to rework decisions across the board as more teams leveraged it
- Had to build goodwill with groups initially uninvolved

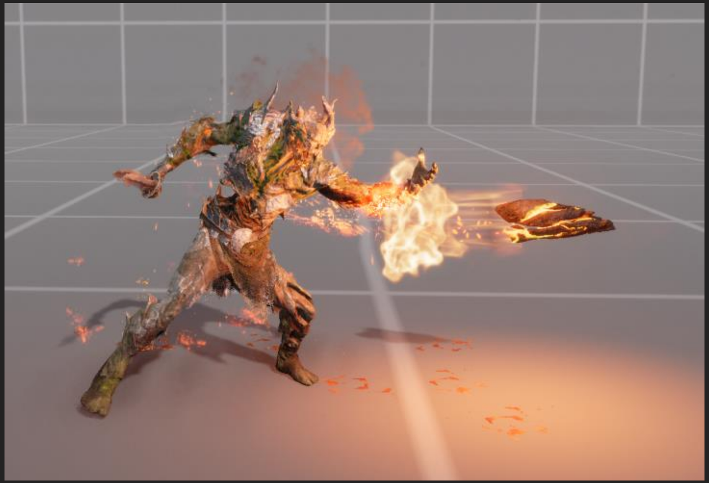

Attributes
level_scripting.vseditor.Diagram
Search
Diagram Attributes
Owner
Associated WAD
Start Disabled
Save Enabled State ✓
Is Instanced

You may have noticed in a number of images in this presentation the phrase "Level Scripting" shows up. This is the name we used internally for the tools for Ragnarok, because it was originally built for our level design team. By the end of the project, it was used for combat, narrative, progression, audio, virtually every design group touched it. Scaling to our initial group of users went well, scaling to the entire design team brought some pain – we had to revisit almost every decision we made to make proper adjustments and had to scramble to build goodwill with the groups that weren't initially involved. The results speak for themselves, but there's still some negative sentiment.

We didn't plan for: Fidelity Issues

- What if another system needs to tell scripts to do something based on some specific frame?
- Deferral to single point makes tuning difficult

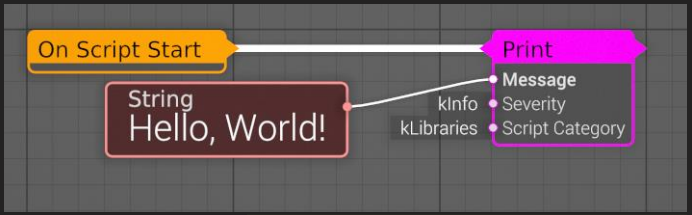




One major unintended consequence of bringing groups on late were fidelity issues. Systems downstream of scripting, like our combat state machine, sometimes wanted to send events back to script to handle complex behavior. If that behavior was visually timed to any element on screen, it became very difficult to tune changes against a variable framerate. Our development environment doesn't always run at target, so changes that look good there may not look as good in a final package. For a high-octane action game with as high a quality bar as God of War: Ragnarok, even a couple frames' difference can be jarring, and deferrals did not make our designer's jobs easier. The two images you see here are one frame apart at 30fps, and help illustrate just how precise we try to get our timing.

Why would anyone prefer text?

- Text-based tools have a lot of support
  - Many different editors
  - Power user tools (like regex searching)
  - Customizeable plugins
- Higher information density
- Less clicks

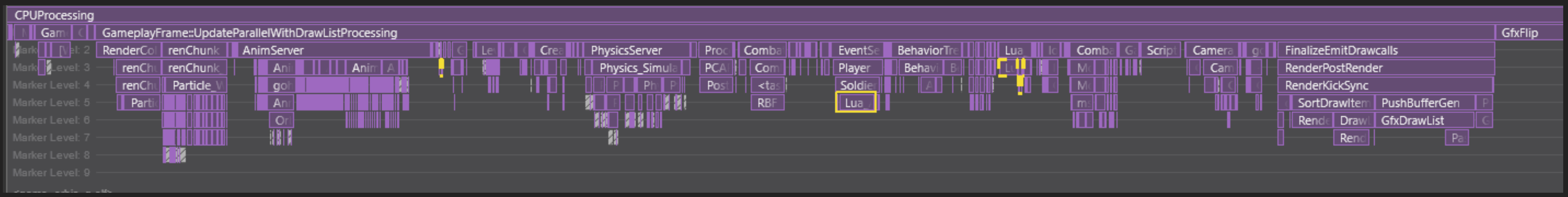


```
print("Hello, World!")
```

Lastly on our major list of complaints, some people simply prefer text. It's not hard to see why, there's no shortage of text editors with sophisticated text manipulation tools, whether you edit with vim, visual studio code, sublime text, or what have you. Some of our designers even wrote their own plugins for their favorite editors. There's a much higher information density to text, no matter how much you try to compress a nodegraph's visuals it's near impossible to beat monospace font. You do not move your hands as much with text, staying entirely on the keyboard instead of moving between the keyboard and mouse. These are hard benefits to beat, and we did our best to mitigate these with all the editor features we provided, but personal preference is personal.

Is there any lua left?

- UI and Camera, and parts of Kratos
- UI and Camera need to do much more complex operations in script than others
- Kratos was left in place due to risk
- ~1ms of the frame



So lastly I mentioned that there's some lua still remaining. The primary users are the UI and Camera, and parts of Kratos. We left these in place because we didn't think the version of visual scripting we had could support them well. Visual script is good when we can do simple sets of operations and skip work with events. Well, our UI has to do some very large transformations on progression data and handle a deep and complex state machine, while our camera needs to change what sets of cameras it's submitting for blending every frame. Kratos was the outlier, where we simply weren't certain we could safely convert the scripts over in time without compromising part of what made the game feel so good – another consequence of the fidelity issues. Overall, it takes an average of 1ms of the frame, which isn't great since that's the sum of all other scripting in the game on an average frame, but we can do a lot better just by continuing to develop the scripting system.

Part 5: Summary + Conclusion

God of War: Ragnarök's Visual Scripting Solution

Santa Monica Studio

With all of that information out in the open, we can now fully evaluate the effects of our core principles and postmortem what went well and what didn't

Principles Summary – Running to Completion

- ~~Coroutines~~ ~~Closures~~ Registered Events!
- Simplicity saved time and eased development
- Design never requested these functionalities
- Programmer error handling provided stability
- Did not manage to reduce logical errors

Santa Monica Studio

From the perspective of running to completion, registered events provided the subset of behaviors we would have needed from coroutines and closures without overcomplicating our runtime. The associated simplicity eased development significantly and helped us adhere to some of our other core tenants. Design never requested those functionalities back. Programmer-driven error handling also proved robust enough, giving us good hints to where things weren't going right while also avoiding many of the problems we had with halting on null objects. However, designers still fell into several logical traps, implying the need to look further at tools to help designers express their logic in more controlled ways. One example may be a general state machine library, which at the time of giving this talk we are actively investigating.

Principles Summary – Static Memory

- We never looked back
- Designers had to manage max size of arrays, otherwise no added complexity
- Memory overages were extremely rare
- NO GC!!!

Santa Monica Studio

As far as static memory goes, we never looked back. It better matched our engine, it had a far smaller footprint, memory overages were extremely rare, and we didn't have any garbage collection whatsoever. The lack of memory overages drastically dropped our remaining scripting error rate. The one downside is that designers had to manage the maximum size of their arrays, but otherwise there was no added complexity. The limits on static memory usage did mean that it was sometimes harder for design to structure their own state machines, but programmers often stepped in to assist.

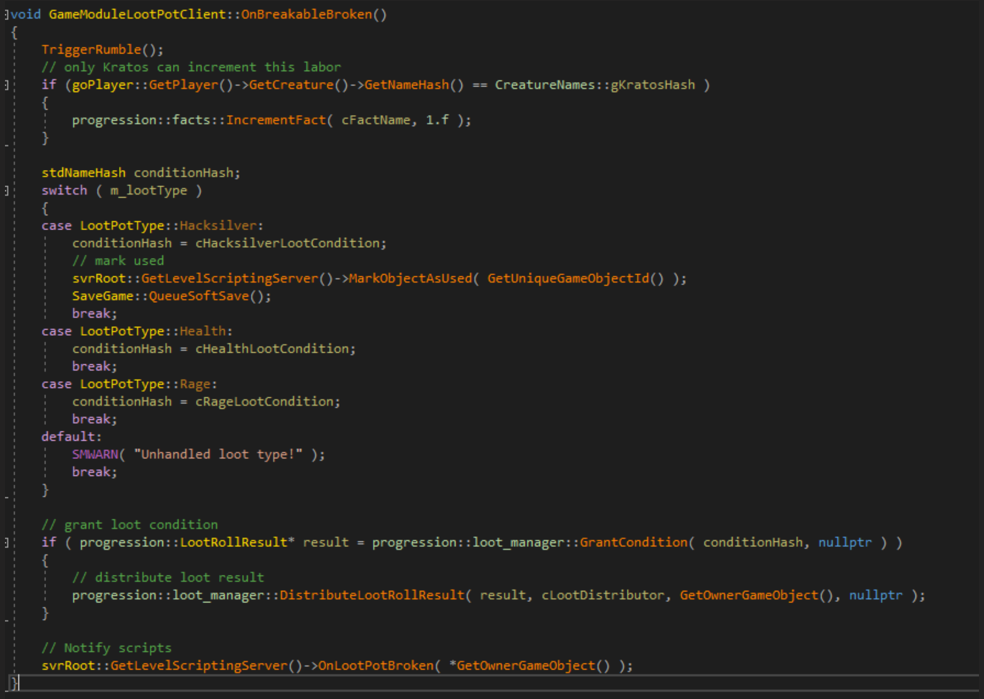
Principles Summary – Eventization

- Pre-interpreter filtering worked out well
- Opportunities for aggressive acceleration
- Gave us the freedom we needed to optimize
- Fidelity with downstream systems is an ongoing issue

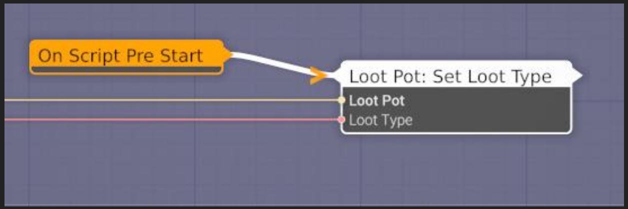
On eventization, things went mostly to plan. Pre-interpreter event filtering and aggressive acceleration likely account for almost all of our performance gains over lua between reducing OnTick polling to near-zero and moving some of the most expensive interpreter operations to after the first few early-returns. It gave us a lot of freedom to optimize other systems too by ensuring that everything runs at one point in the frame. It was so efficient that all but a few special case events are broadcasts, rather than targeted function calls! However, as we extended the tool for use with other design teams, fidelity issues became a concern. A door opening a couple extra frames after Kratos finishes interacting with a crank is one thing, a projectile spawning in an enemy's hand a couple frames late as it hurls a baseball-sized fireball at Kratos is going to look weird. We need to investigate ways to keep the benefits of our limited frame execution while addressing these fidelity issues. The performance gains associated with this approach speak for themselves, but pose an interesting question with the work that could not be elided, such as OnScriptStart logic, as to what logic belongs in scripting at all.

Principles Summary – Composability

- Worked pretty much how we wanted it to



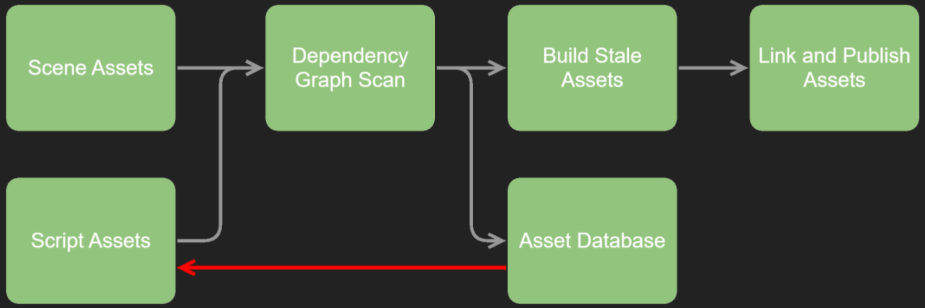
```
void GameModuleLootPotClient::OnBreakableBroken()
{
    TriggerRumble();
    // only Kratos can increment this labor
    if (goPlayer::GetPlayer()->GetCreature()->GetNameHash() == CreatureNames::gKratosHash )
    {
        progression::facts::IncrementFact( cFactName, 1.f );
    }

    stdNameHash conditionHash;
    switch ( m_lootType )
    {
    case LootPotType::Hacksilver:
        conditionHash = cHacksilverLootCondition;
        // mark used
        svrRoot::GetLevelScriptingServer()->MarkObjectAsUsed( GetUniqueGameObjectId() );
        SaveGame::QueueSoftSave();
        break;
    case LootPotType::Health:
        conditionHash = cHealthLootCondition;
        break;
    case LootPotType::Rage:
        conditionHash = cRageLootCondition;
        break;
    default:
        SMWARN( "Unhandled loot type!" );
        break;
    }

    // grant loot condition
    if ( progression::LootRollResult* result = progression::loot_manager::GrantCondition( conditionHash, nullptr ) )
    {
        // distribute loot result
        progression::loot_manager::DistributeLootRollResult( result, cLootDistributor, GetOwnerGameObject(), nullptr );
    }

    // Notify scripts
    svrRoot::GetLevelScriptingServer()->OnLootPotBroken( *GetOwnerGameObject() );
}
```



On composability, this worked almost exactly how we wanted it to. Programmers were able to take entire modules that were previously in script and seamlessly move them into code when necessary, while designers were able to effectively organize their own scripting. The example you see on screen now is a good portion of the loot pot module that was previously just under a hundred nodes that would be instantiated hundreds of times in each visual script, reduced down to a single interface for when we needed to override specific loot pots with behavior. Proper encapsulation of scripting paid huge dividends, even when design's abstractions were imperfect. Treating embedded scripts as macros also didn't cause too many problems with the size of our scripts, even though some sets of nodes were copied hundreds of times – we still had our total allocation far under what existed for lua.

Principles Summary – Assetization

- Safer runtime lookups and error tracking
- Design requested more of this at every opportunity
- Build limitations meant increased DB iteration time
- Time spent on interim solution well worth it!



Santa Monica Studio

Assetization was a mixed bag. On the one hand, we had safer runtime lookups and error tracking, and design seemed eager to request more of this at every opportunity for the tracking and editor functionality benefits. Putting serious engineering effort into ensuring our asset database solution was consistent and robust paid off and provided us the tools necessary to provide all of that tracking and editor functionality. On the other hand, while powerful, certain limitations of our build system itself are inherited by our asset database and create a cyclical build dependency that exacerbated already frustrating build times. Work is already underway to address these limitations in order to provide up-to-date asset info without committing to a full scene scan.

Principles Summary – Simple Interpreter

- Syntactic sugar was easy to manage
- A/B testing complex changes against stable runtime
- Simplicity leads to stability, stability leads to trust, and users MUST trust their scripting language
- We provided a well-featured editor
  - *In large part from the generosity of our sister studios

Santa Monica Studio

On having a complex editor for a simple interpreter, this tradeoff paid off big. A/B testing complex changes against a simple runtime routinely paid off in allowing us to make complex changes to our tools to better serve our designers, including any change where we introduced some new form of syntactic sugar. Whether it was portal nodes, embedded scripts, exotic non-standard default behaviors, or whatever else, having a simple runtime to target helped us validate that everything was as we expected it to be. Simplicity leads to stability, stability leads to trust, and if you ever lose that trust you are never going to get it back. Despite all conventional wisdom to the contrary, we were able to provide a well-featured editor with a single dedicated engineer and a few other contributors, though whether we could have done it or not without Guerrilla' generous donation of code is unknown

The Date was November 9, 2022...

- 🎉God of War: Ragnarök Shipped! 🎉
- We took a close look at how we built it
- Including taking a close look at our scripting systems

We end our story right after God of War: Ragnarok shipped, when we held a postmortem of our various technologies. The process took a lot of time, but helped us understand where we were and where we wanted to go. Given it's prevalence in 2018's postmortem, we wanted to take a hard look at our scripting systems and their complications

The New Complications – Spikes

- It has large spikes… but we can make it faster
  - None of it's principles are interpreter-dependent
  - We can swap the interpreter for an x64 backend
  - We can create better tools to define initial game state and avoid OnScriptStart

Santa Monica Studio

The first problem is that the system has some notable very large spikes. Specifically surrounding OnScriptStart, but in general combat events doing a lot of processing can get quite bad as well. The good news is that we can make it faster. All I've discussed in this presentation regarding performance is event-driven programming and the importance of good acceleration structures, none of that is language dependent. We could swap the current interpreter backend for something a lot faster, like LLVM x64, and we can investigate better tools external to scripting for defining initial states of modules so that we can avoid so much logic running on script start.

The New Complications – Tools Support

- The tools are limited… but we can iterate on them
  - We're not going to stop providing editor support
  - We're fixing the Asset DB iteration times
  - We're talking about better ways to structure logic

The second problem is that our tools are limited. But it's not like we're going to sit on what we have, we're going to keep improving our editor, and we're actively building solutions to our build iteration times which will keep our asset database updating as fast as possible. And we're going to find better ways to structure logic so that we can reduce the number of logical errors our designers have to contend with as well. We won't be satisfied until scripting the game is as easy as speaking one's own first language.

The New Complications – Fidelity

- Delays make tuning hard… but we can fix that
  - Cleaner mechanisms to avoid deferral
  - Potentially running scripting more than once
    - But still at defined safe points!

Santa Monica Studio

The third problem is the fidelity issue, but there are relatively straightforward ways to fix that. We can create cleaner mechanisms to avoid unnecessary deferrals. We're also not above potentially running scripting more than once in the frame, so long as the points in the frame we run it at are safe to do so rather than in the middle of other systems. The solutions to this problem will need to be considered carefully, but they are eminently solveable.

Conclusion

- There are different problems, but we think they're solvable without changing languages again
- The system is measurably more efficient and stable, with a smaller memory footprint
- We believe we made the right choice
- "It's a miracle this system came together as well as it did" - Jon Burke, Gameplay Tech Director, GOW:R

Santa Monica Studio

In conclusion, at the end of the day, we face different problems than we did with lua, but we think we can solve those problems. Compared to lua, where we thought we had hit a dead end, this is an immensely better place to be. In addition, the system is measurably more efficient and stable by any metric that we actually care about. We believe that we made the right choice and did a fine job with it. The tech director overseeing the project once dedared it a miracle that the system came together as well as it did, that major game systems do not typically come online and function this well within a single project. Hopefully this talk gives you enough information to repeat this miracle for your own team, solving your own problems.

A Team Effort

- **Josh Phelan** (Editor owner, runtime interpreter, event dispatching, compiler, codegen, visual script frontend for diff/merge, debugger, optimization, touched almost everything really)
- **Sam Sternklar** (Runtime interpreter, event dispatching, compiler, codegen, engine interface, diff/merge core, editor work, optimization)
- **Darren Rannali** (debugger, profiler, asset database integration)
- **Phil Wilkins** (codegen, eventization scheme, initial interpreter)
- **Fernando Secco** (optimizations and misc. bug fixes)
- **Sam Willis** (Initial editor build-out)
- **Jeff Miller** (diff/merge core)
- **Yanbing Chen** (additional editor support, diff/merge core)
- **Koray Hagen** (asset database integration, build assistance)
- **Enrico Gasperoni** (Initial concept, runtime interpreter)
- **Federico Bianco-Prevot** (codegen and schema framework)
- **Paolo Costabel** (codegen and schema framework)
- **Jon Burke** (Director oversight)
- **Mike Grattan, Fatir Ahmad, Rob Meyer, Adrian Lopez, Adam Oliver, Alex Ortiz, Rene Nones, Henry Lee, Marc Nguyen, Vicki Smith, Zach Bohn** (Design Council)
- **Matty Studivan, Tina Sanchez-O'Hara, Dustin Dobson, Katie Tigue, Bobby Garza** (Production support)
- **Guerrilla** (editor framework)

Santa Monica Studio

Before we end off, it's worth taking a moment to acknowledge everyone who contributed to the visual scripting solution. It would not have been possible without the contributions of every single person on this list, and every single one of them deserves credit for their craftsmanship. I want to give a special acknowledgement to Josh Phelan, who has done so much quality work that he has become synonymous with the visual scripting initiative within the walls of SMS.

Santa Monica Studio

Our journey
Your story

We're hiring for what's next!

We're expanding our family across disciplines and would love to meet you. Please visit sms.playstation.com/careers for all openings or drop us a line at sms.recruiting@sony.com

@ santamonicastudio   @ SonySantaMonica   @ santamonicastudio

And finally, if anything I just spoke about looked interesting or cool to work on, we're hiring!

JOIN US AT GDC 2023

GOD OF WAR RAGNARÖK

BUILD YOUR GOD OF WAR GDC AT:

SCHEDULE.GDCONF.COM

BRUNO VELAZQUEZ ♦ ANIMATION DIRECTOR
DAVID GIBSON ♦ ANIMATION DIRECTOR
ERICA PINTO ♦ LEAD NARRATIVE ANIMATOR
MEHDI YSSEF ♦ LEAD GAMEPLAY ANIMATOR
Animation in 'God of War Ragnarök' ♦ Animation Summit
MONDAY, MARCH 20 ♦ 9:30AM – 10:30AM ♦ ROOM 303, SOUTH HALL

SUE PACETE ♦ SR USER RESEARCHER
Playtesting God of War Ragnarök Accessibility Options ♦ UX Summit
MONDAY, MARCH 20 ♦ 1:20PM – 1:50PM ♦ ROOM 2001, WEST HALL

PAOLO SURRICCHIO ♦ SR STAFF PROGRAMMER
Reinventing the Wheel for Snow Rendering ♦ Advanced Graphics Summit
MONDAY, MARCH 20 ♦ 1:20PM – 2:20PM ♦ ROOM 301, SOUTH HALL

BEN HINES ♦ SR STAFF DEVOPS ENGINEER
Automated Testing at Santa Monica Studio ♦ Tools Summit
MONDAY, MARCH 20 ♦ 4:40PM – 5:10PM ♦ ROOM 3004, WEST HALL

XUANYI ZHOU ♦ PROGRAMMER
Real-time Neural Texture Upsampling in 'God of War Ragnarök' ♦ Machine Learning Summit
TUESDAY, MARCH 21 ♦ 2:10PM – 2:40PM ♦ ROOM 2010, WEST HALL

ETHAN AYER ♦ SR ENVIRONMENT ARTIST
The Art of Making Vistas ♦ Art Summit
TUESDAY, MARCH 21 ♦ 3:00PM – 3:30PM ♦ ROOM 3007, WEST HALL

GÖKSU UĞUR ♦ AI LEAD
Preparing AI Systems For God of War Ragnarök ♦ Programming
WEDNESDAY, MARCH 22 ♦ 9:00AM – 10:00AM ♦ ROOM 303, SOUTH HALL

VICKI SMITH ♦ SR STAFF LEVEL DESIGNER
The Final Battle of 'God of War Ragnarök': Techniques For Delivering High-stakes Sequences ♦ Design
WEDNESDAY, MARCH 22 ♦ 10:30AM – 11:00AM ♦ ROOM 2002, WEST HALL

STEPHEN McAULEY ♦ LEAD RENDERING PROGRAMMER
Rendering 'God of War Ragnarök' ♦ Programming
WEDNESDAY, MARCH 22 ♦ 2:00PM – 3:00PM ♦ ROOM 303, SOUTH HALL

ERIC GOTTESMAN ♦ SR STAFF DEVOPS ENGINEER
Modernizing multiplayer services for "God of War: Ascension" (PS3) ♦ Production & Team Leadership ♦ Presented by Amazon Web Services
WEDNESDAY, MARCH 22 ♦ 2:00PM – 3:00PM ♦ GDC PARTNER STAGE, EXPO FLOOR, NORTH HALL

SAM STERNKLAR ♦ SR PROGRAMMER
'God Of War Ragnarök's' Visual Scripting Solution ♦ Programming
THURSDAY, MARCH 23 ♦ 10:00AM – 11:00AM ♦ ROOM 2006, WEST HALL

ADAM OLIVER ♦ SR COMBAT DESIGNER
Breaking Barriers: Combat Accessibility in 'God of War Ragnarök' ♦ Design
THURSDAY, MARCH 23 ♦ 2:00PM – 2:30PM ♦ ROOM 2002, WEST HALL

GÖKSU UĞUR ♦ AI LEAD
Practical Tools for Transitioning Into Leadership Roles ♦ Leadership
THURSDAY, MARCH 23 ♦ 2:00PM – 2:30PM ♦ ROOM 303, SOUTH HALL

ZACH BOHN ♦ SR STAFF TECHNICAL UI DESIGNER
'God of War Ragnarök': Building The UI For a AAA Title ♦ Design
THURSDAY, MARCH 23 ♦ 4:00PM – 5:00PM ♦ ROOM 303, SOUTH HALL

SALAAR KOHARI ♦ PROGRAMMER
Companion Traversal in 'God of War Ragnarök' ♦ Programming
FRIDAY, MARCH 24 ♦ 10:00AM – 11:00AM ♦ ROOM 2002, WEST HALL

TENGHAO WANG ♦ SR PROGRAMMER
Joint-based Skin Deformation in 'God of War Ragnarök' ♦ Programming
FRIDAY, MARCH 24 ♦ 1:30PM – 2:30PM ♦ ROOM 2006, WEST HALL

HARLEIGH AWNER ♦ TECHNICAL NARRATIVE DESIGNER
How to Build a Home: Designing Narrative For Sindri's House in 'God of War Ragnarök' ♦ Design
FRIDAY, MARCH 24 ♦ 3:00PM – 3:30PM ♦ ROOM 2001, WEST HALL

Santa Monica Studio GDC

Thank You!

God of War: Ragnarök's Visual Scripting Solution
By Sam Sternklar
LinkedIn: ssternklar
Twitter: @Aureon71

That's it. Thank you!