

GDC



BRYAN KEIREN
PRINCIPAL GAME TECH PROGRAMMER

NODES AND NATIVE CODE: DECIMA'S VISUAL PROGRAMMING FOR EVERY DISCIPLINE



DECIMA™



Est. 1998-ish



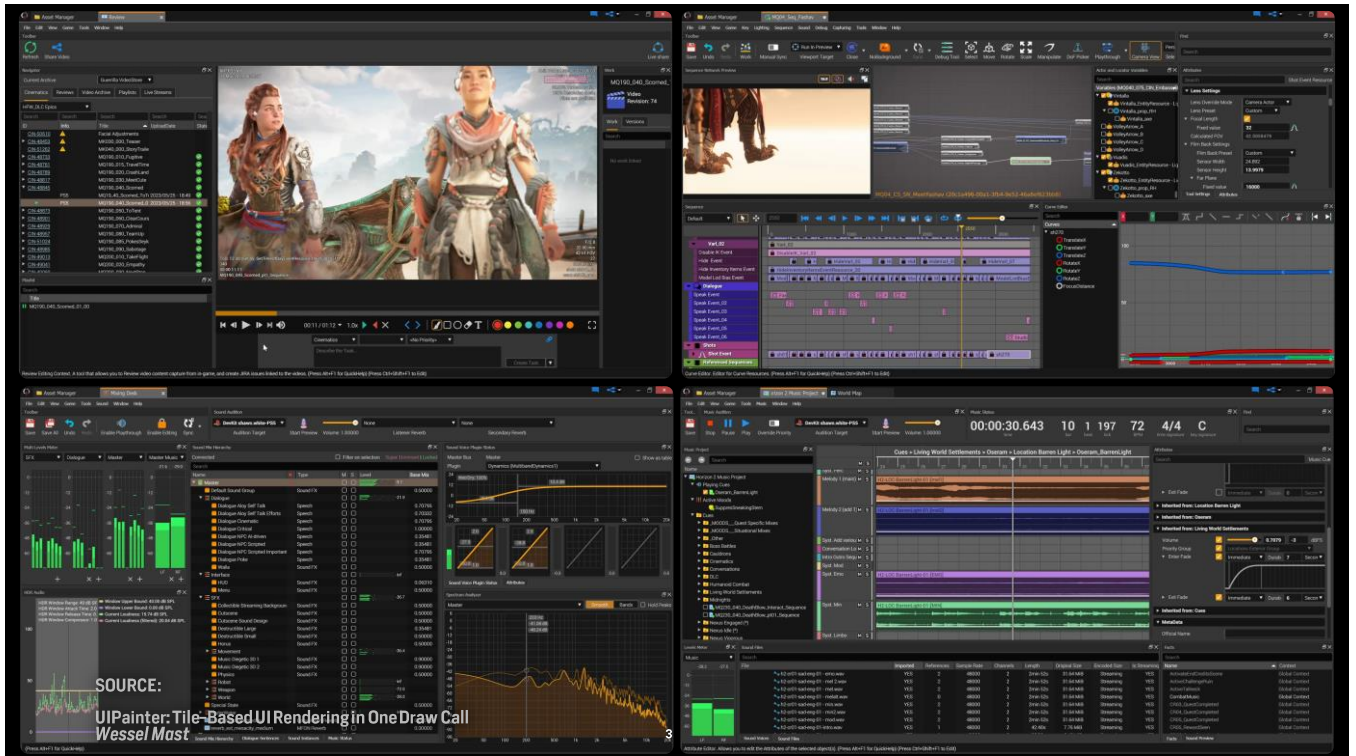
GUERRILLA 2

DECIMA engine developed and used by Guerrilla and external studios

Origins somewhere around 1998.

Since used to build:

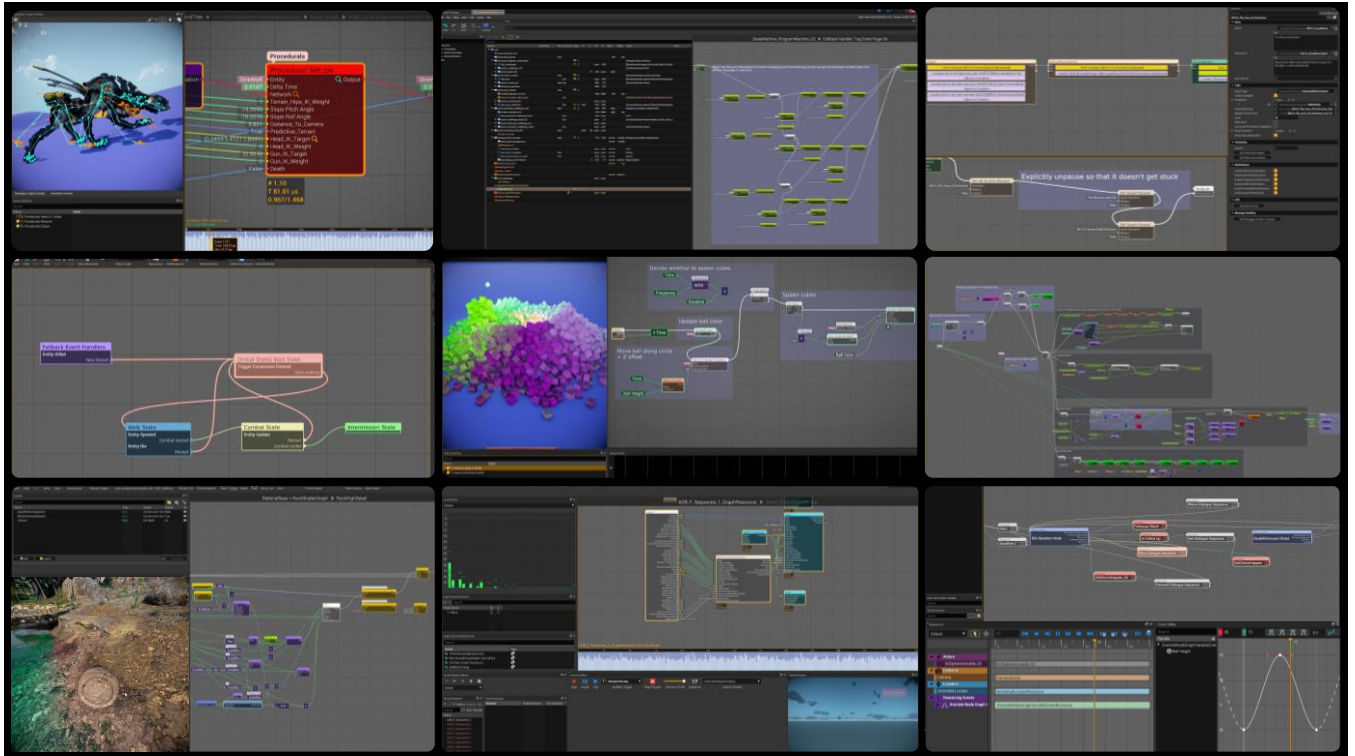
- Killzone series
- Horizon Zero Dawn
- Horizon Forbidden West
- and Death Stranding (by Kojima Productions)



Decima has impressive suite of tools, but tools are not a goal in itself

Want to enable making cool video games quickly and efficiently, independent from programmers

One of the tools that enables us to do so is our visual programming system



Visual programming system has been expanded over span of a decade to topics of:

Animation, Gameplay, Audio, Shaders, Cinematics, Effects, UI, and more...



BRYAN KEIREN



GUERRILLA

Bryan Keiren; Principal Game Tech Programmer at Guerrilla in Amsterdam.

Past 11 years worked on visual programming framework

History
Graphs & State Machines
Asset to Runtime
Debugging
Usability
Animation & Shader Graphs

Topics

ABOUT A DECADE AGO...



**KILLZONE: SHADOW FALL
(2013)**



**HORIZON ZERO DAWN
(2017)**

In 2013, Guerrilla had a challenge ahead

From corridor shooter Killzone: Shadow Fall to open world (Horizon Zero Dawn)

In need of way to fill with interactive content

To be able to do that we knew needed better tools to script gameplay



GUERRILLA 8

```
114
115
116 -----
117 -- GameModeSpawnPlayer
118 -- Called from RespawnPlayers.
119 -----
120 function GameModeSpawnPlayer(inPlayerIndex, inSpawnPoint)
121
122     local player_team = GetPlayerTeam(inPlayerIndex)
123     local member = GetTeamMember(player_team, inPlayerIndex)
124
125     -- Spawn the player.
126     SpawnPlayer(inPlayerIndex, inSpawnPoint)
127
128     -- Configure AI player.
129     if (gGamePlayInfo:IsAIControlled(inPlayerIndex) and member ~= nil) then
130
131         -- A proper role will be assigned later.
132         AssignRole(player_team, member, "unassigned")
133
134         -- Set roaming markers.
135         local actor = gGamePlayInfo:GetPlayerActor(inPlayerIndex)
136         actor:GetAI():SetWanderMarkers(player_team.mRoamMarkers)
137
138     end
139
140 end
141
```

KILLZONE (2004)

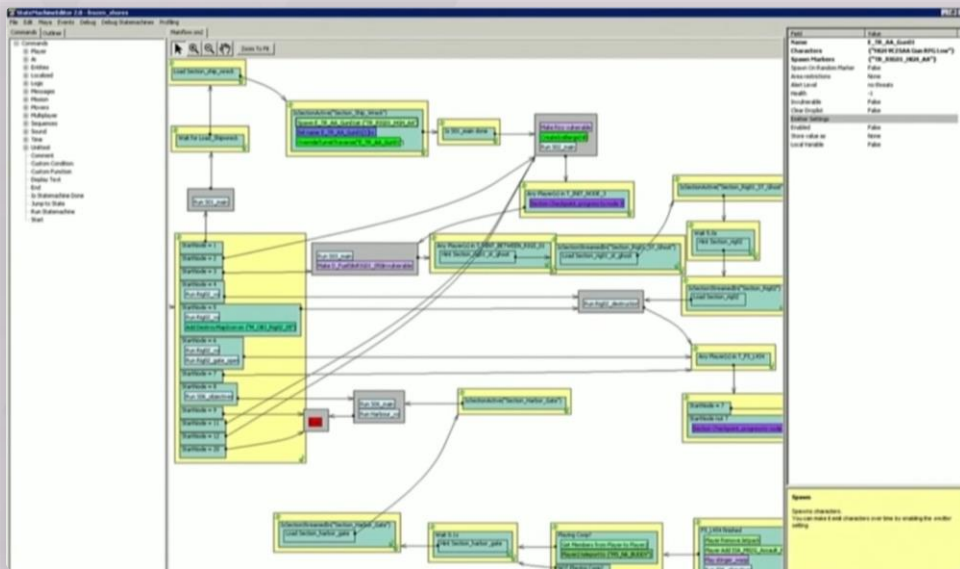
The story of visual programming in DECIMA starts with Lua scripts in Killzone 1 (2004).

Only used for SP encounter scripts, menu, and player spawning logic.

On the right side is an example of some Lua script code used in KZ1.

Integrated by Jorrit Rouwé in like an afternoon and then stuck around for 15 years as these things go.

Somewhere along the way Lua scripts start to become tied together in more complex set-ups using the “State Machine Editor”



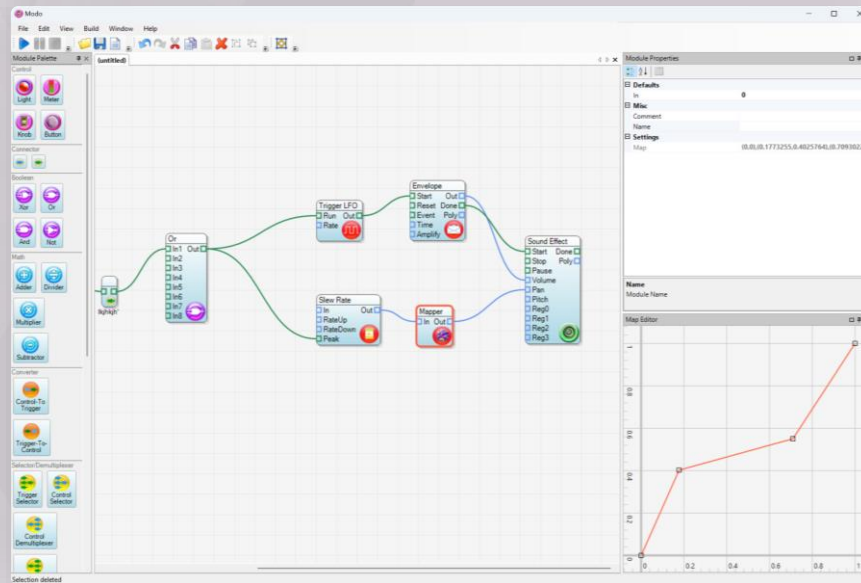
LUA STATE MACHINES

Lua scripts in State Machine

State Machine Editor (SMED)

Only gameplay encounter scripting...

...but other things happening in other departments...



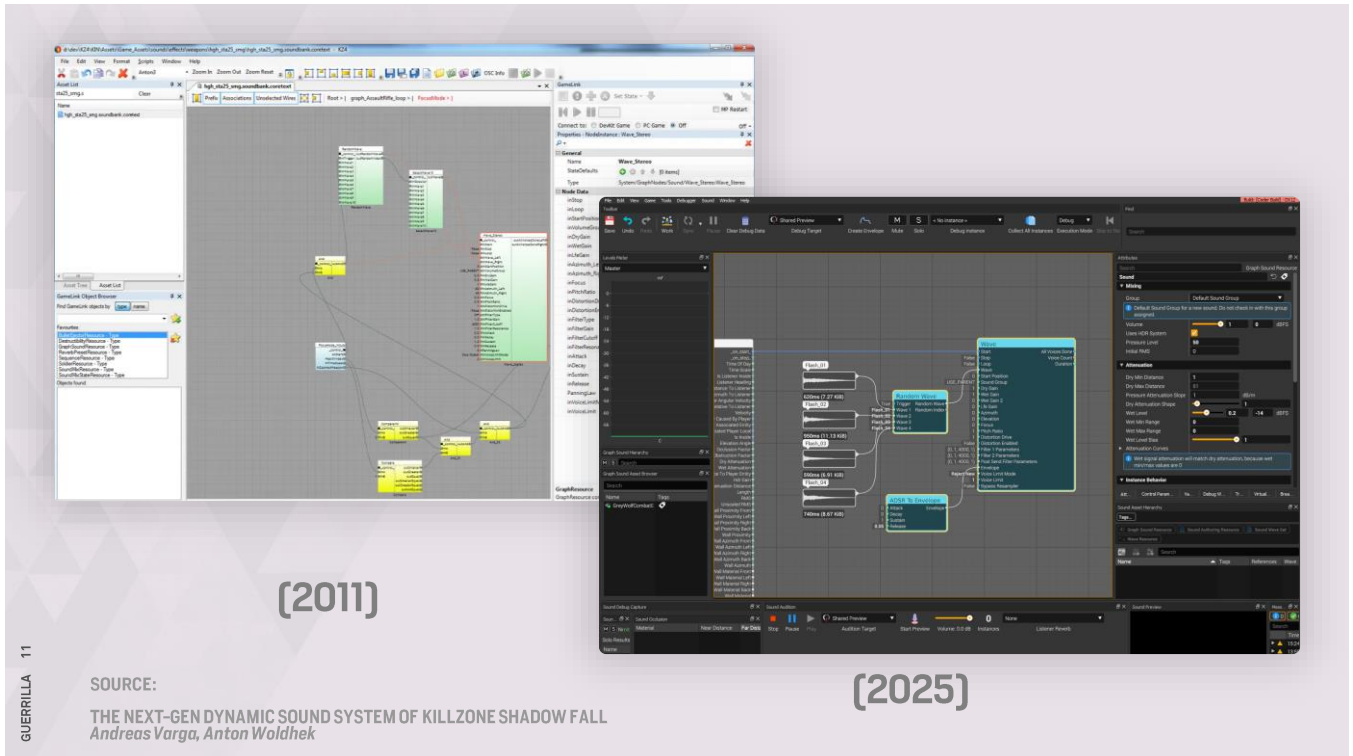
AUDIO SCRIPTING

Audio designers were using tool called MODO.

Node-graph based workflow

Only to author audio; developed outside of Guerrilla studio.

Source of inspiration for....



In-house audio scripting tool.

Introduced ~2011, development of Killzone Shadow Fall

Pure data dependency ****node graphs**** with math nodes, DSP nodes, and some nodes to affect or query game state.

Left side old editor; right side current

History

Graphs & State Machines

Asset to Runtime

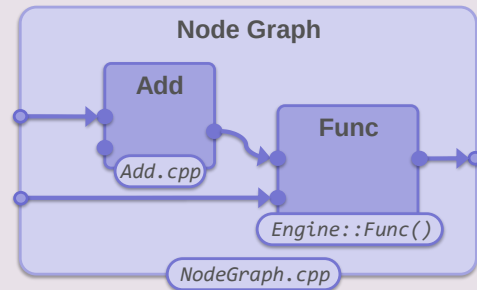
Debugging

Usability

Animation & Shader Graphs

NODE GRAPH

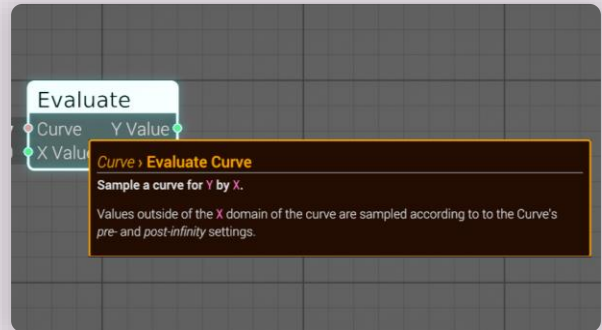
- Nodes
 - Sub-graph
 - Code snippet
 - Exposed engine functions
- Wires
 - Carry **values** (*int/float/Vec3/Entity/etc...*)
 - Are dependencies between nodes
- Generates **native C++ code** for performance



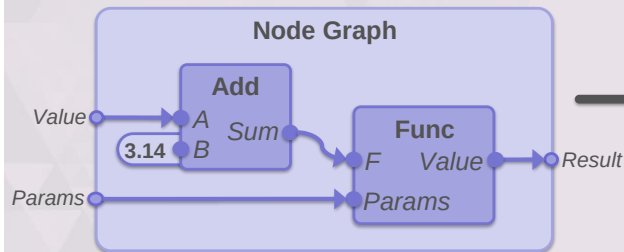
Anatomy of Node Graph

NODES

- Made by **both** programmers and content-creators
- **Categorized** and **documented**
- **Strictly-typed** or **generic** pins



NODE GRAPH



```
float Add(float inA, float inB)
{
    return inA + inB;
}

namespace Engine
{
    float Func(float f, Params* params);
}

void NodeGraph(Inputs* inputs, Outputs* outputs)
{
    float add_sum = Add(inputs->mValue, 3.14);
    float func_value = Engine::Func(add_sum, inputs->mParams);
    outputs->mResult = func_value;
}
```

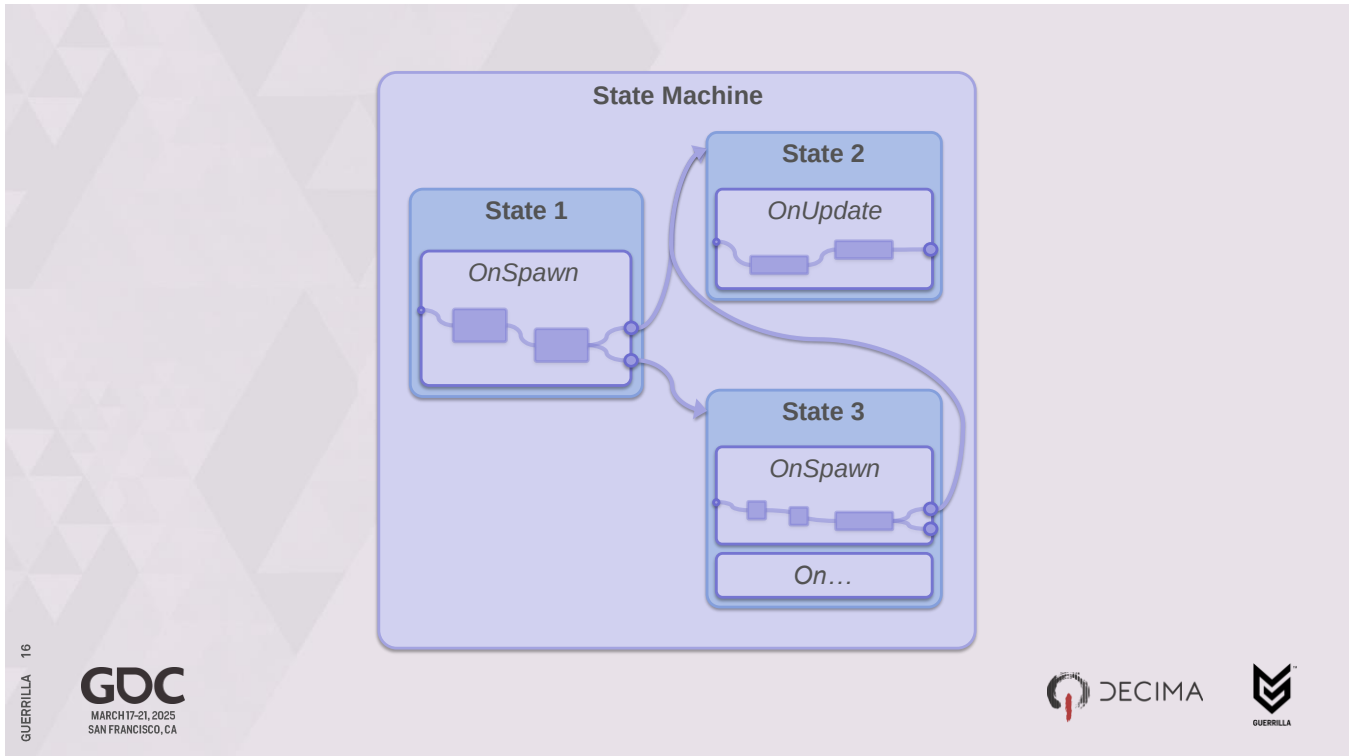
Look at how pseudo generated code looks:

1. Entire graph is a function with inputs/outputs
2. Nodes can be snippet with function like Add node
3. Or exposed engine function like DSP node
4. Node outputs passed as variables or written to graph output

But this is audio scripting...

ALSO wanted this tool for game logic and encounter scripting...

And found LUA State Machine workflow very useful for game logic scripting....



So, wanted to unify State Machine concept with Node Graphs.

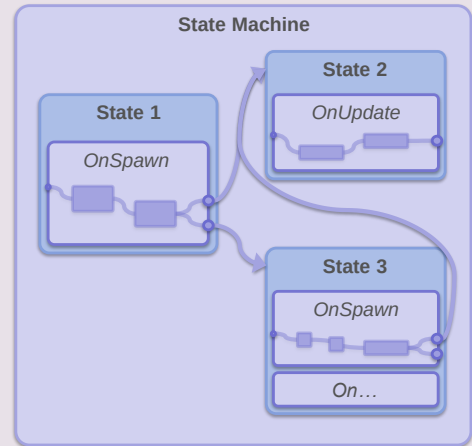
Wanted to build State Machine as part of Node Graphs

Lets look at anatomy of our vision:

1. State machine has 1 or more states
2. Each state has zero or more Events it can listen to
3. Each event can trigger a transition to another state
4. Each event handler is actually just a Node Graph with some special outputs that control whether the state machine transitions to another state

EVENT DRIVEN STATE MACHINE

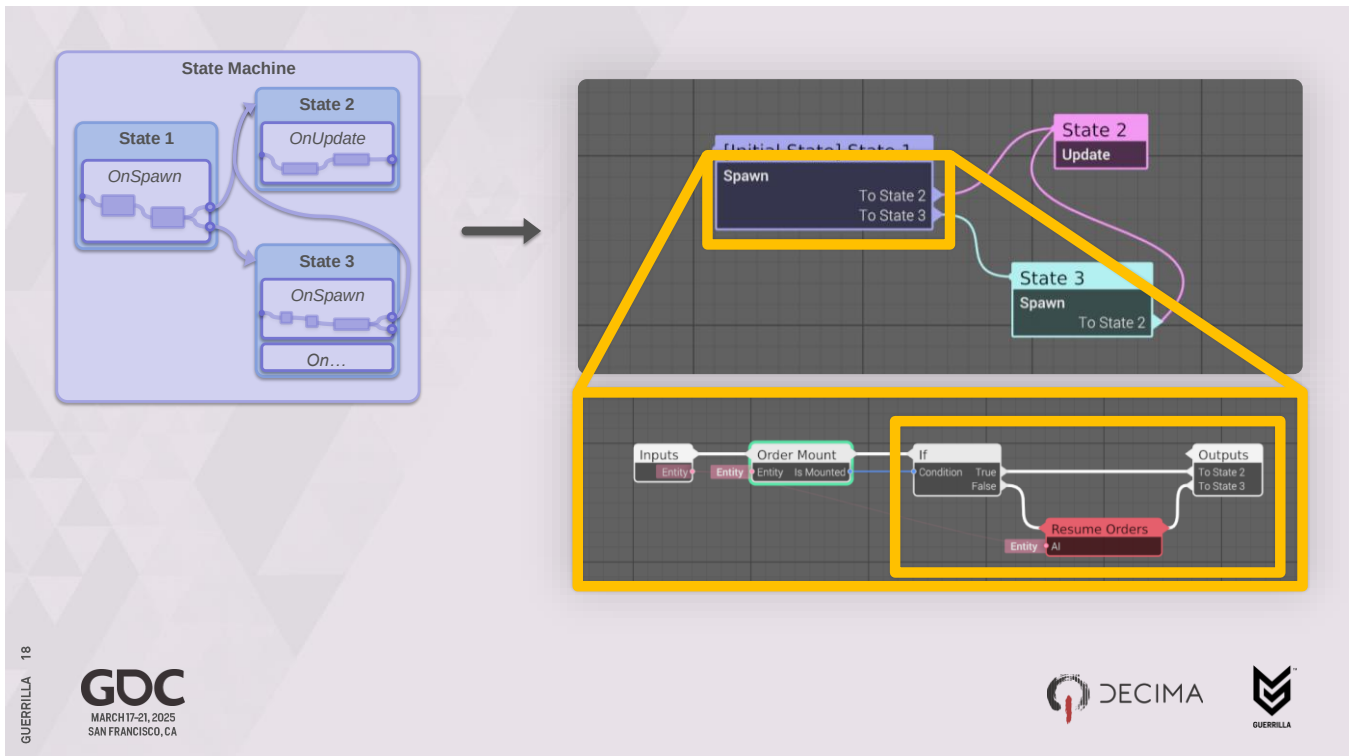
- Scripts with “*state machine*” *execution model*
- **Generalize** by introducing:
 - Events as **script entry points**
 - **Control Flow**
 - **Persistent variables**
- **Bespoke** UI and code generation



Anatomy of a State Machine

BONUS: Prepared us for other execution models in the future. Specifically, animation graphs as state machines.

Any engine system can use Node Graph/State Machine



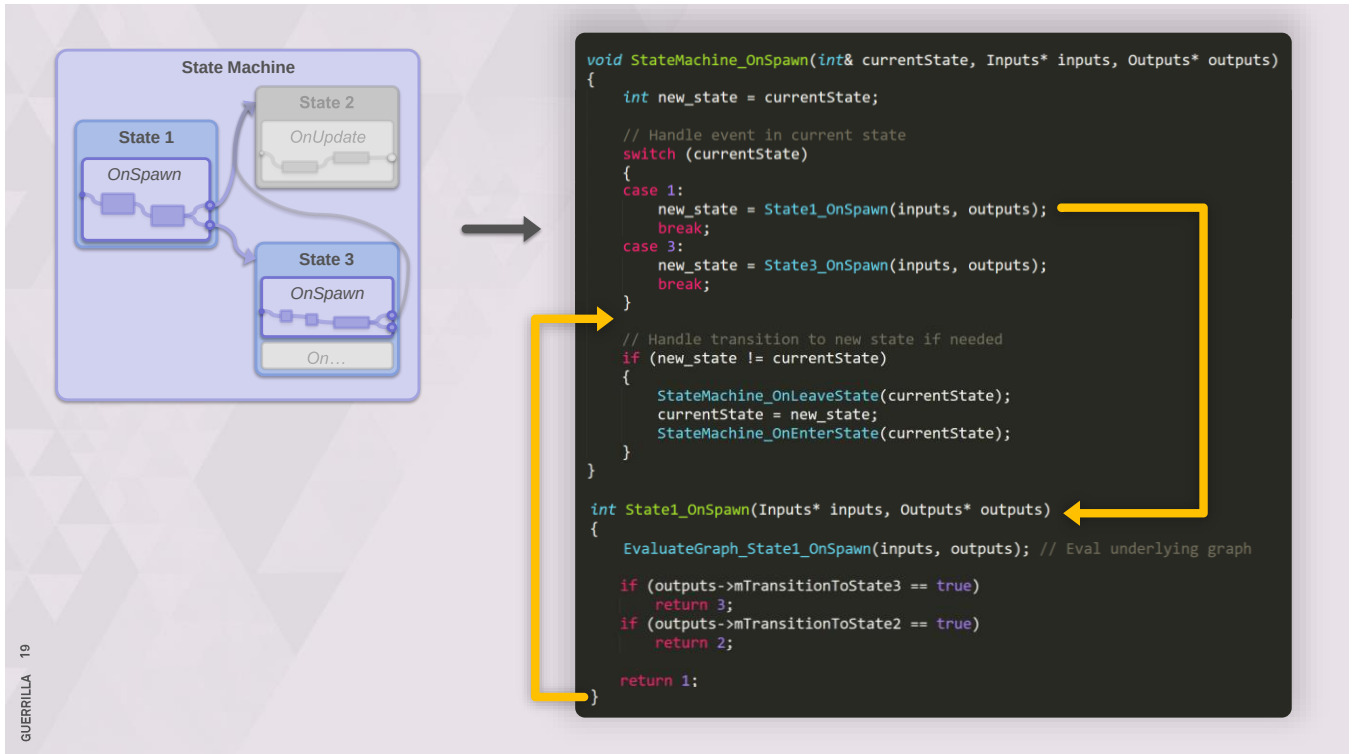
Eventually all of that gave us state machines in node graph framework.

Looked like this in editor

And OnSpawn handler is a Node Graph for State 1.

Control Flow can be seen in node graph here:

1. If-statement, skips node in False path.
2. TRANSITION of state machine.



Pseudo generated code for State Machine

1. Entry point function for “Spawn” event
2. Switch statement forwards to dedicated function for current state.
3. That function evaluates underlying handler graph...
4. And then returns what should be the next state
5. After the switch we check if we want to be in different state
6. If we should transition, invoke “Leave Current State” event, then “Enter State” event.

Lot of technical details so far, so lets see a pretty picture...



First user of this was “Thunderjaw” robot prototype

Although... back in 2011 it probably looked more like... this

History

Graphs & State Machines

Asset to Runtime

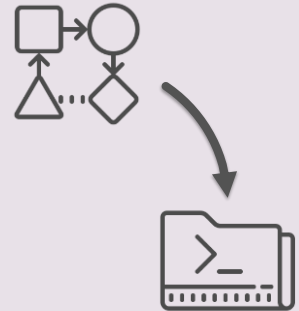
Debugging

Usability

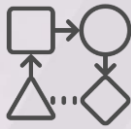
Animation & Shader Graphs

CONVERSION

- DECIMA assets are **text** format
 - **Converted** to **binary** for runtime
- Node Graphs' binary format depends on *type of asset*
 - *Shader Graphs* > **Compiled shader binaries** to run on GPU
 - *Other Graphs* > **Compiled C++ dynamic library** to run on CPU
- Node Graph code generation happens during conversion process
 - **Mostly the same between CPU and GPU graphs**



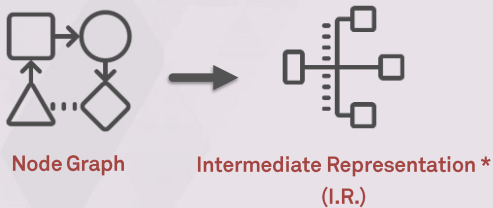
Lets look at how code generation works



Node Graph

We start with the Node Graph asset in text format

Semantic Analysis



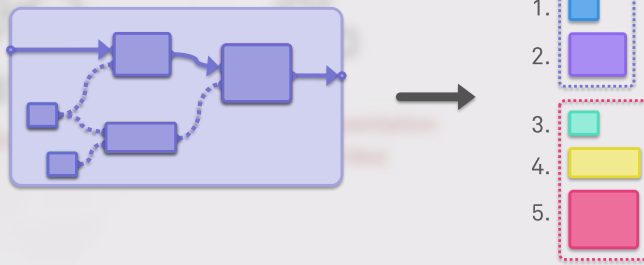
The Node Graph asset undergoes semantic analysis and turned a language-agnostic intermediate representation.

It is a custom format, similar to an ABSTRACT SEMANTIC GRAPH (ASG), in case you are familiar with the concept, but a little higher level.

This representation can be used to perform optimizations and remove unused variables, logic, etc.

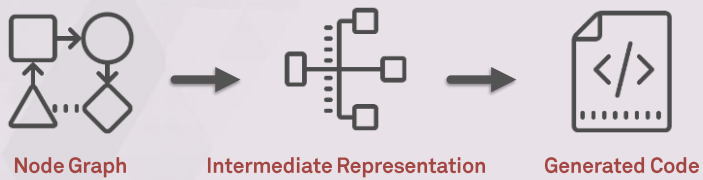
Semantic Analysis

Determine a **grouping** and **ordering** of **actions**
that represent the meaning of the graph/state machine.

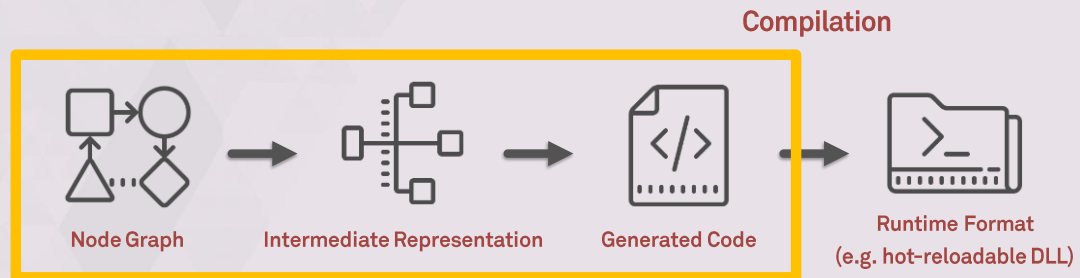


Semantic Analysis explanation

Code Generation



The intermediate representation is then forwarded to a language-specific backend to generate code (C++, shader code, etc.).



Then, generated code is compiled.

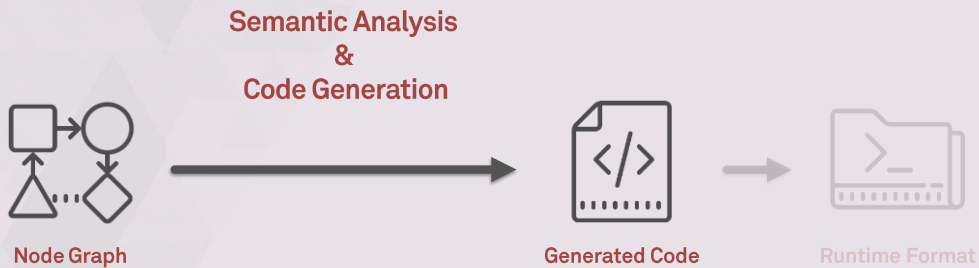
In case of C++ gets linked into a dynamic library that can be hot-reloaded.

Note that doesn't require the game executable to be re-compiled.

Important for iteration; content creators can't be expected to recompile game just because of changes to some content.

Now, want to focus on this part little more; because it is important to our process.

OLD APPROACH

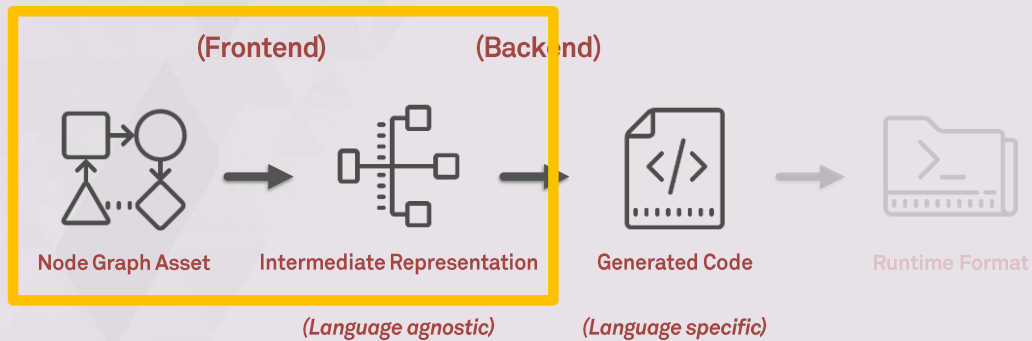


Used to have combined analysis and code gen logic.

No intermediate representation.

- Difficult to maintain
- Most often only want to change analysis logic, not code-gen
- But all changes involve code string formatting
- Hard to share analysis logic between languages

CURRENT APPROACH



Looked at compiler frameworks like LLVM.

Adopted separation of semantic analysis / code gen.:

- Analysis yields language-agnostic Intermediate Representation (ASG-like).
- I.R. consumed by language-specific “backend” for code-gen.

With this approach:

- Most changes only IR, not code-gen
- Can share analysis and IR between languages
- ...Which makes it easier to add languages or new functionality to multiple languages
- Even share code-gen for C-like syntax languages too!

I.R. GRAPH

- **Value**
 - *Variable*
 - *Binary expression*
 - ...
- **Block**
 - *Function Call*
 - *Conditional*
 - *Assignment*
 - ...
- **Function**
- **Module**
- **Type**

Intermediate Representation
(Semantic Graph)

So, what does IR look like?

Lowest level: IRValue. Represents typed values. Examples:

- Variable / Literal / Binary/Unary expression

Then, IRBlock. Represents more high level statements. Examples:

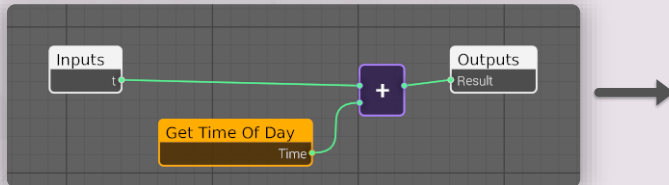
- Function calls / Conditionals (If/Switch)
- For-loops / Value assignments

Highest level, to glue it all together:

- FunctionContext (i.e. function definition)
- Module (collection of contexts)
- Type / Class

GENERATING I.R.

Function
"Graph(in t, out Result)"



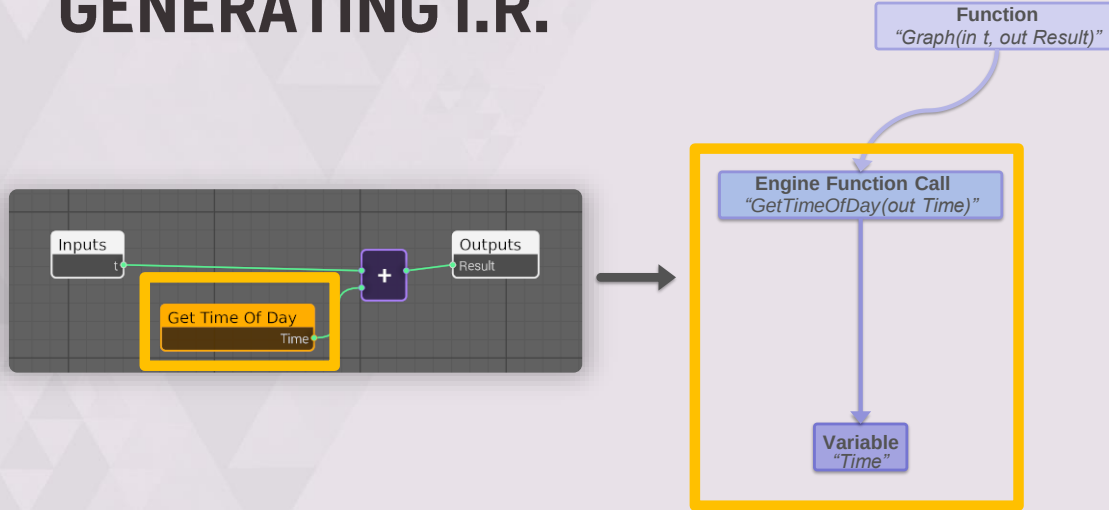
To generate IR, we have a “generator” per node-type

Generator does semantic analysis and emits corresponding IR

In example: Graph as a whole generates a Function Context.

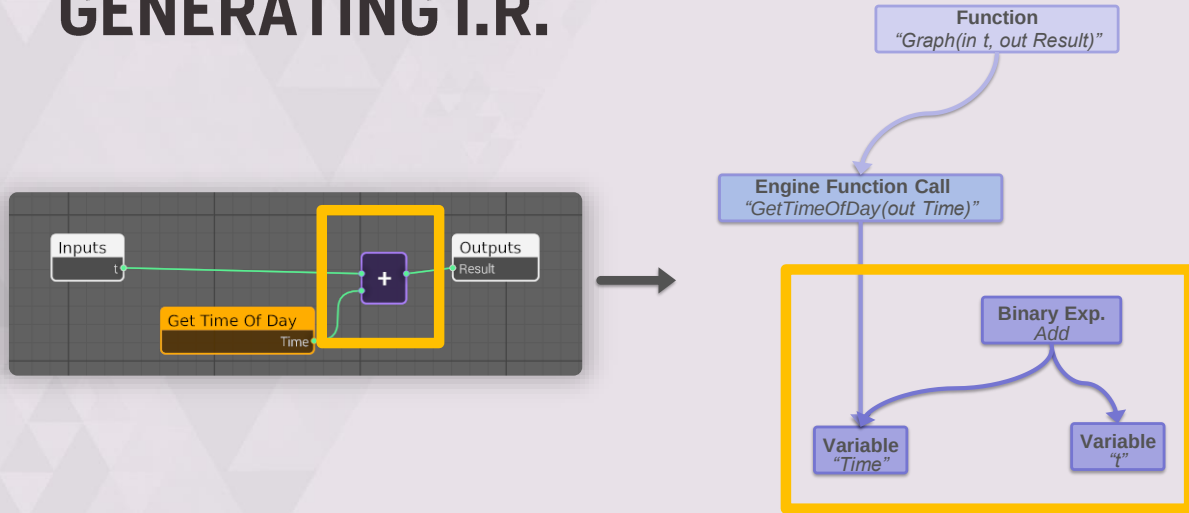
Then defers to dedicated generator for each node...

GENERATING I.R.



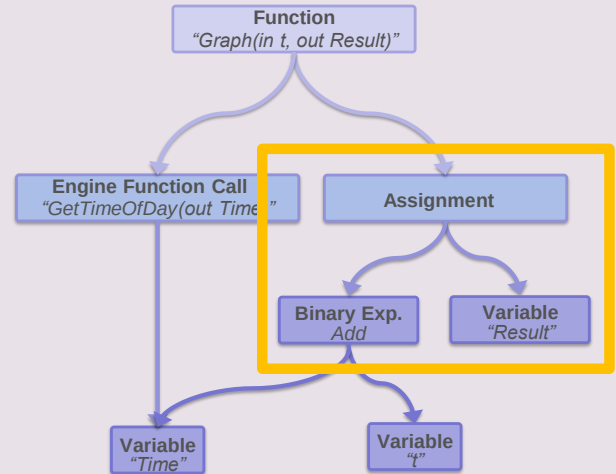
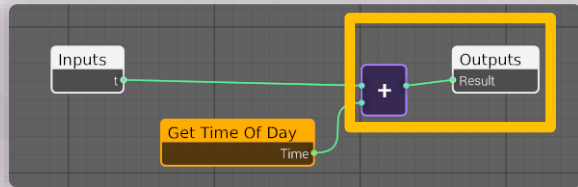
This TimeOfDay node becomes a Function Call block that references the "Time" variable

GENERATING I.R.



The “Add” node becomes a Binary Add Expression that references its two inputs: The “Time” and “input T” variables

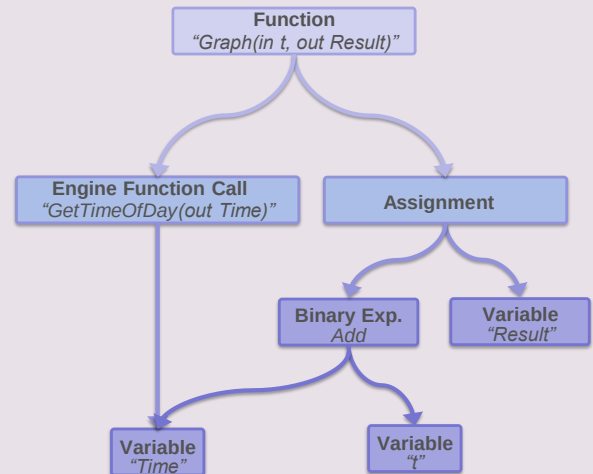
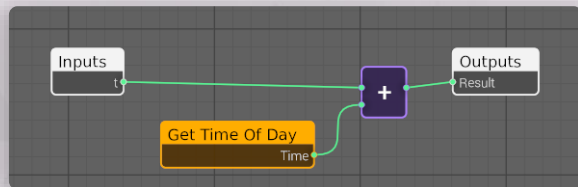
GENERATING I.R.



Finally, the Graph generator ties it all together:

It assigns the Binary expression value to the “Result” output variable.

GENERATING I.R.



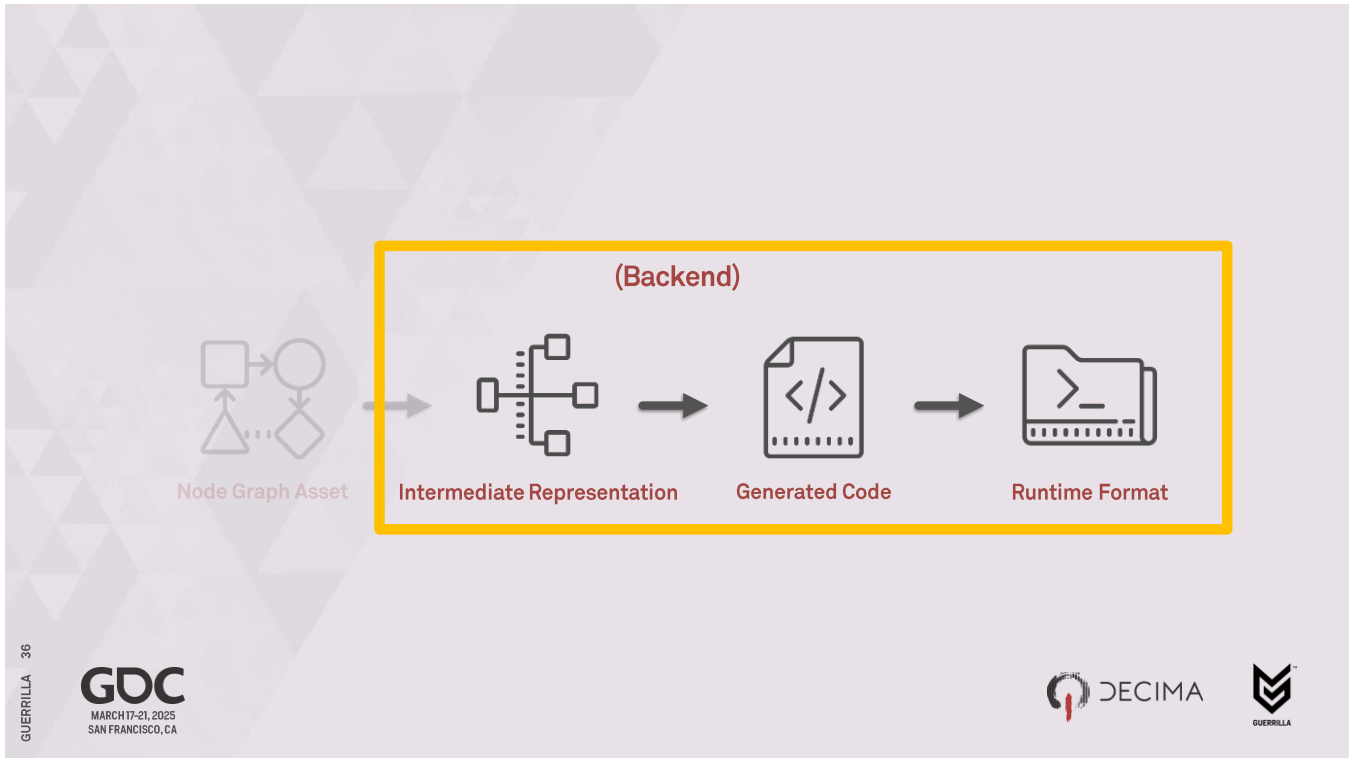
Have generators for Graphs, State Machines, Functions...

and for specialized nodes that want bespoke IR generation.

****Most**** nodes don't need specialized generator.

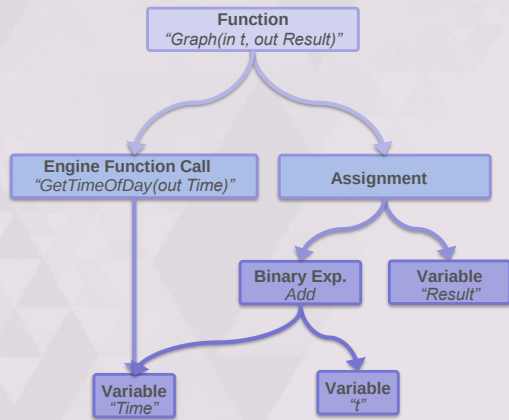
Most nodes are just a function snippet, so they use default generator for that.

Post processing after this phase (Remove unused variables, etc.)



Lets focus on this part now.

I.R. BACKEND



```
void Graph(float i_t, float* o_Result)
{
    /* [IRExportedFunctionCall] */
    float time = 0.0f;
    Game::GetTimeOfDay(&time);

    /* [IRAssignment] */
    *o_Result = i_t + time;
}
```

IR consumed by backend

Traverse IR graph and, concatenate strings

Produces code string

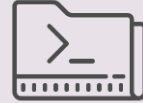
Backend cares IR, not original graph

Similar languages share code

Output of backend is sent to compiler, to turn into runtime format.

RUNTIME DLL

- **Build farm** makes monolithic **fullgame.dll** with **all scripts** *(except for local content changes)*
- Monolithic DLL keeps number of DLLs in check
- Game executable **does not link** against these DLLs



Custom symbol im-/export mechanism

DLL SYMBOLS

- In *Horizon Forbidden West* + DLC...
 - 7,300 types/functions exposed from engine
 - 82,000 graphs
 - ≥ 1 exported entry point each
- Linker supported $2^{16}=65.536$ symbols...
- Bespoke export/import mechanism

```
LibraryExport gLibraryExportsTable[] =
{
    { 0x8dcbbe8d6e9e15882c3fb152969daae, (void*)&FooGraph_UpdateEvent_EntryPoint },
    { 0x94b8200057d8a30b596bf1f9ff13b34, (void*)&FooGraph_SpawnEvent_EntryPoint },
    ...
};

// Define library imports as function pointers
namespace Game
{
    float (*GetTimeOfDay)(); // float Game::GetTimeOfDay()
}

using Exporter = bool (*)(LibraryExport* inTable, uint32 inCount);
using Importer = void* (*)(uint32 inSymbolHash);

// Primary DLL entry point (only user of dllexport).
// Importer/Exporter defined in- and provided by engine.
_declspec(dllexport) extern "C" void DllEntryPoint(Exporter inExporter, Importer inImporter)
{
    // Import symbols from the engine
    Game::GetTimeOfDay = inImporter(0x109010e8);
    // Export functions to the engine
    inExporter(gLibraryExportsTable, gElemCount(gLibraryExportsTable));
}
```

Note on DLL symbols

Fullgame.dll in HFW has 82.000 graphs.

Each graph at least one entry point function to export from DLL.

Linker only allowed 65.536 exported symbols

So, bespoke mechanism.

DLL SYMBOLS

- DLL initialization:
- Export from DLL:
Hash table of function pointers
- Import from engine:
Copy pointers of functions/variables into global DLL variables
(registered through RTTI)

```
LibraryExport gLibraryExportsTable[] =
{
    { 0x8dcbbe8d6e9e15882c3fb152969daae, (void*)&FooGraph_UpdateEvent_EntryPoint },
    { 0x94b8200057d0a30b596bf1f9ff13b34, (void*)&FooGraph_SpawnEvent_EntryPoint },
    ...
};

// Define library imports as function pointers
namespace Game
{
    float (*GetTimeOfDay()); // float Game::GetTimeOfDay()
}

using Exporter = bool (*)(LibraryExport* inTable, uint32 inCount);
using Importer = void* (*)(uint32 inSymbolHash);

// Primary DLL entry point (only user of dllexport).
// Importer/Exporter defined in- and provided by engine.
_declspec(dllexport) extern "C" void DllEntryPoint(Exporter inExporter, Importer inImporter)
{
    // Import symbols from the engine
    Game::GetTimeOfDay = inImporter(0x109010e8);

    // Export functions to the engine
    inExporter(gLibraryExportsTable, gElemCount(gLibraryExportsTable));
}
```

On DLL initialization

1. Export from DLL with **hash table** of pointers
2. Import from engine by copying pointers into global DLL variables

RUNTIME NUMBERS



GUERRILLA 41

GDC
MARCH 17-21, 2025
SAN FRANCISCO, CA



Runtime Case study; Chainscape pub

RUNTIME NUMBERS

- Case study: **Horizon Forbidden West** (PS5)
- **High-load** scenario
 - **~13.700** node graphs in memory
 - **~900** executions **per frame**
 - Incl. animation, quests, sequences, audio, entities, spawn- and streaming conditions.
 - Distributed over **13 threads**
- **~9 μ s** (average) / **~2 μ s** (median) per execution

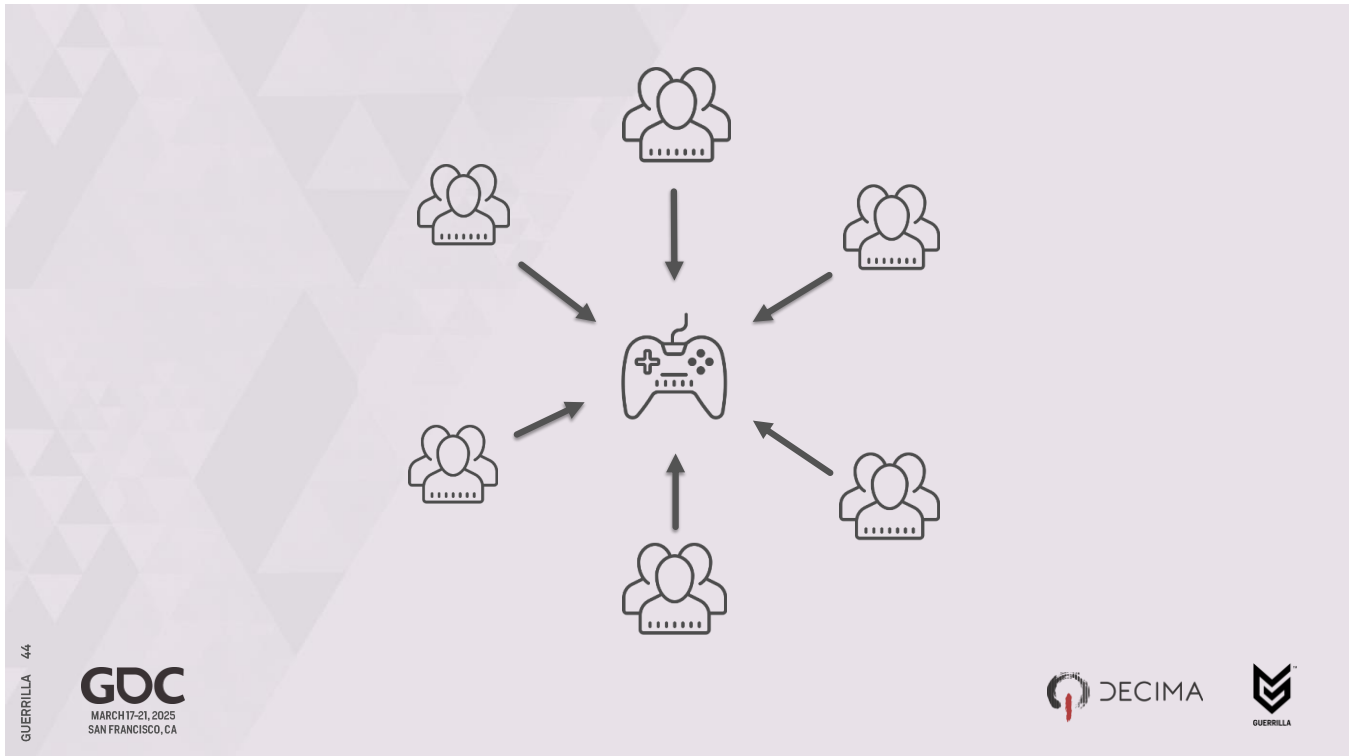


Runtime Case study; Measured using our in-game CPU profiler.

Node Graphs do not know about multi-threading even though instances can execute concurrently. Engine supports concurrency at a different level through job scheduler with dependencies. Systems that incorporate node graphs are responsible for ensuring dependencies are indicated *before* executing a node graph. This way, node graph doesn't need to know about it.

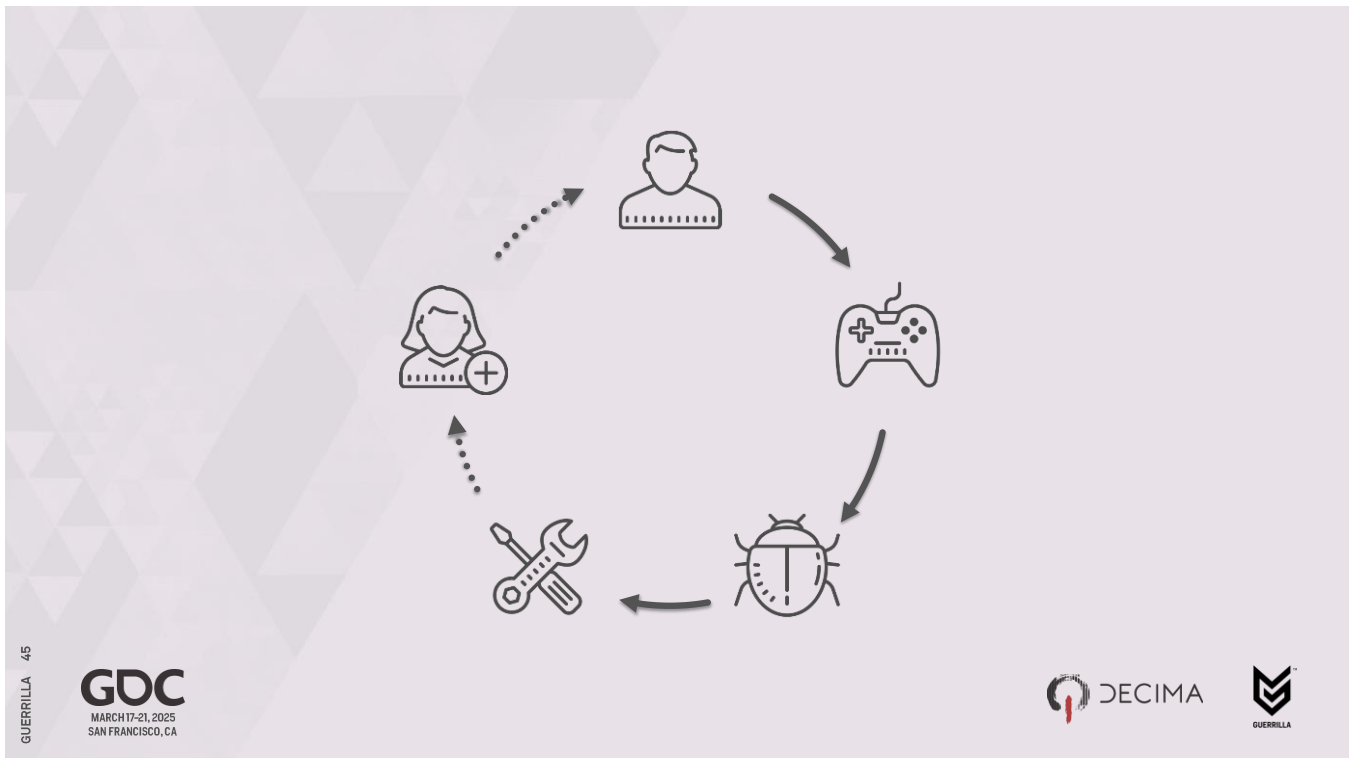
History
Graphs & State Machines
Asset to Runtime
Debugging
Usability
Animation & Shader Graphs

So, now that we know how to get to the runtime, time to talk about our approach to debugging at runtime!



So, we had a system that was being used in production work, and we had a lot of people actively using it to produce Horizon Zero Dawn content.

But, if you're producing a lot of content you're probably producing a lot of bugs as well.



Node Graph execution used to be a black box for users.

Every time someone found a bug in Node Graph-related content it needed to be debugged by the content creator. After all, they know their content best.

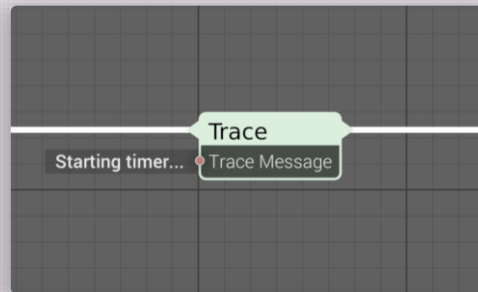
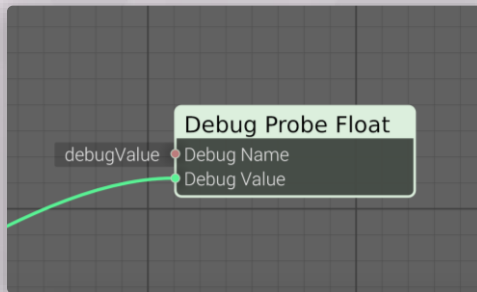
Debugging cycle looks like this:

- 1: Create content, then test content
- 2: Find a bug
- 3: Try solving it with one of debug tools...
- 4: Often had to fall back to including a programmer for help.

Content Creator and programmer then work together to debug (usually Visual Studio debugger).

Costs time to get up to speed with content's structure.

Costs Content Creator time initially, and both CC and programmer time afterwards too

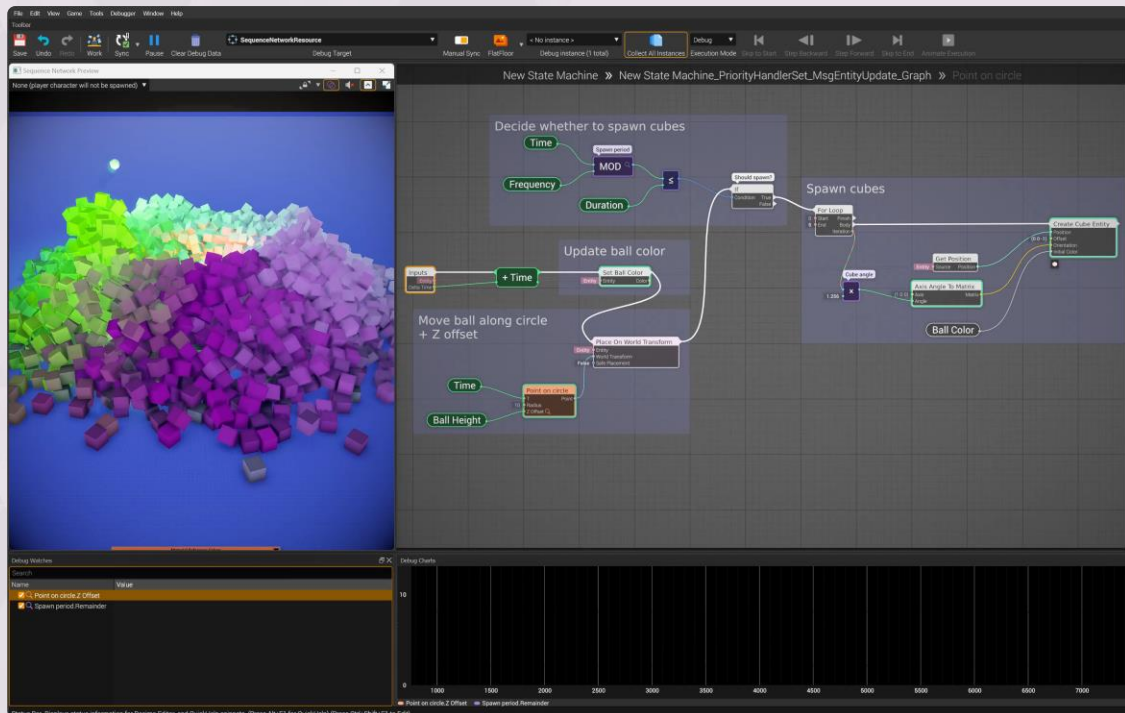


NodeInstance	NodeResource	Connection	Type	Connector	Value
graph_Inputs	owl_engine_gr...	GraphInput	SoundProgramContext	inContext	(nil)
graph_Inputs	owl_engine_gr...	GraphInput	bool	inInside	True
graph_Inputs	owl_engine_gr...	GraphInput	float	inDistanceToListener	3.9023235
graph_Inputs	owl_engine_gr...	GraphInput	float	inAzimuthToListener	357.06683
graph_Inputs	owl_engine_gr...	GraphInput	bool	on_start	False
graph_Inputs	owl_engine_gr...	GraphInput	bool	on_stop	False
graph_Inputs	owl_engine_gr...	GraphInput	float	inForwardSpeed	0
graph_Inputs	owl_engine_gr...	GraphInput	float	inStrafeSpeed	0
graph_Inputs	owl_engine_gr...	GraphInput	float	inPanSpeed	0
graph_Inputs	owl_engine_gr...	GraphInput	float	inTiltSpeed	0
graph_Inputs	owl_engine_gr...	GraphInput	float	inHealth	1
graph_Inputs	owl_engine_gr...	GraphInput	float	inScanningPanSpeed	6.9155671e-05
graph_Inputs	owl_engine_gr...	GraphInput	float	inScanningTiltSpeed	3.5573905e-08
graph_Inputs	owl_engine_gr...	GraphInput	float	inAimAlignedToTarget	0
graph_Inputs	owl_engine_gr...	GraphInput	bool	inHasTarget	False

Log, Manual (probe & trace)

Clean up probe/trace

Also: In-game, Automatic (table)

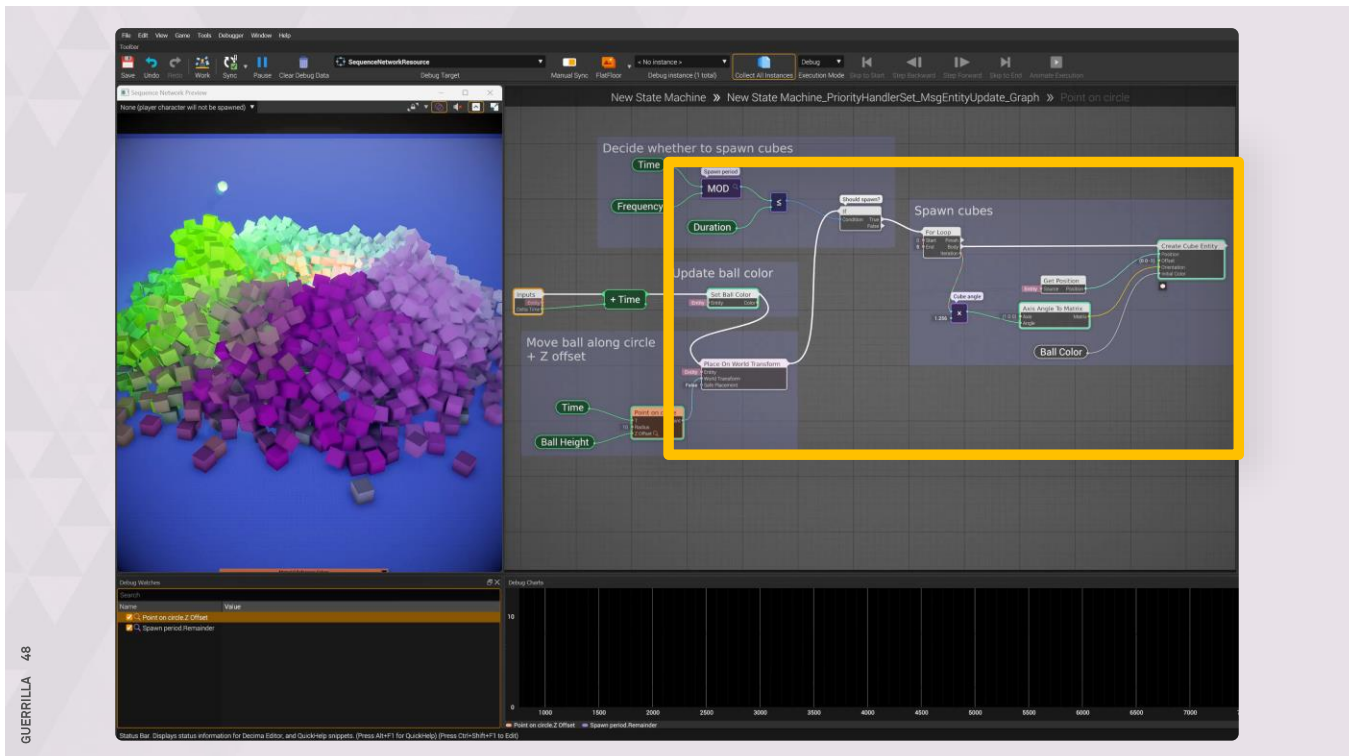


Decided to build debugging tools into editor that users are accustomed to.

Already have a system that is collecting debug values.

Let's combine them!

And show them in the editor. Let's break down some of the features of this debugger.



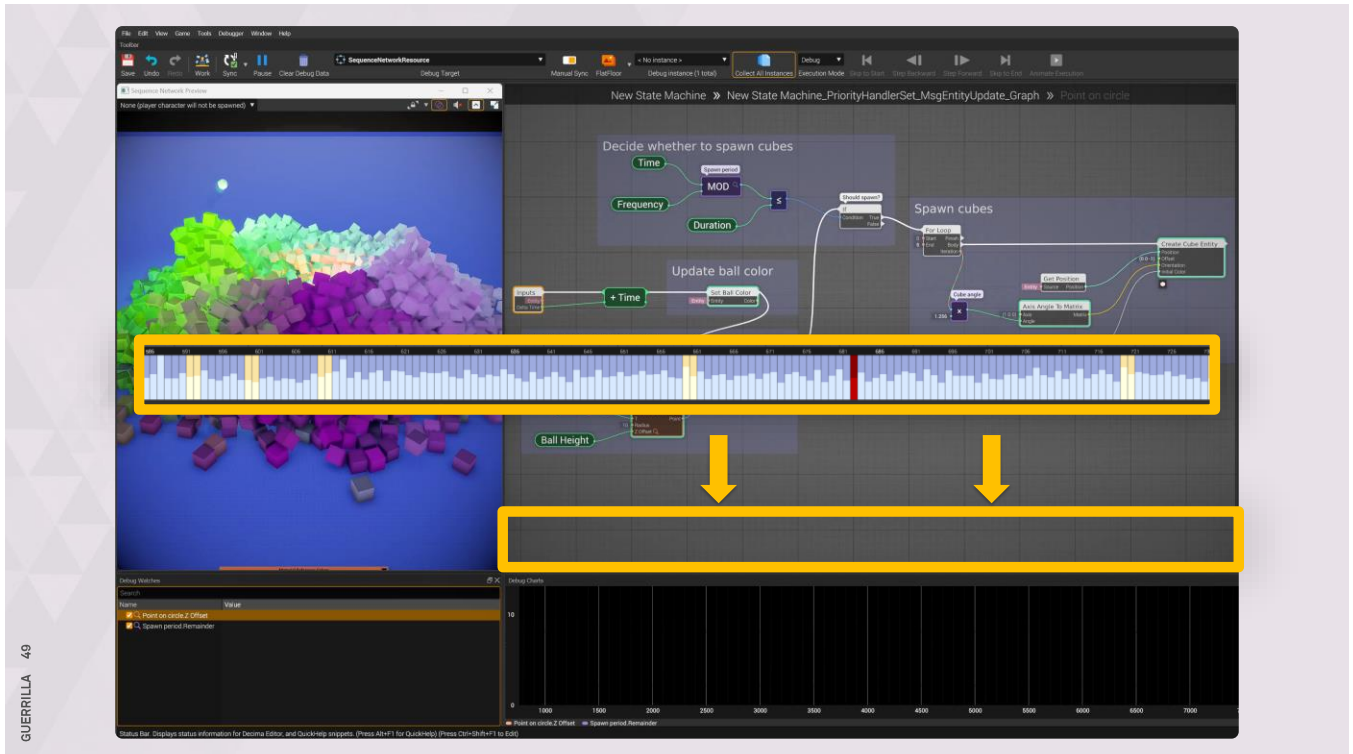
Canvas view highlights current control flow path.

Uses DECIMA's high-perf profiling protocol for real-time visualization

Dims nodes that are not active.

Highlights pin values on nodes when changed.

Under each node shows an execution index and time taken (with debug overhead)



Timeline bar at bottom of the canvas.

Shows every execution of the graph.

Can be zoomed and panned

BACK IN TIME: Selected execution shows in canvas

Two shades of blue; height of light blue indicates time spent relative to other bars:

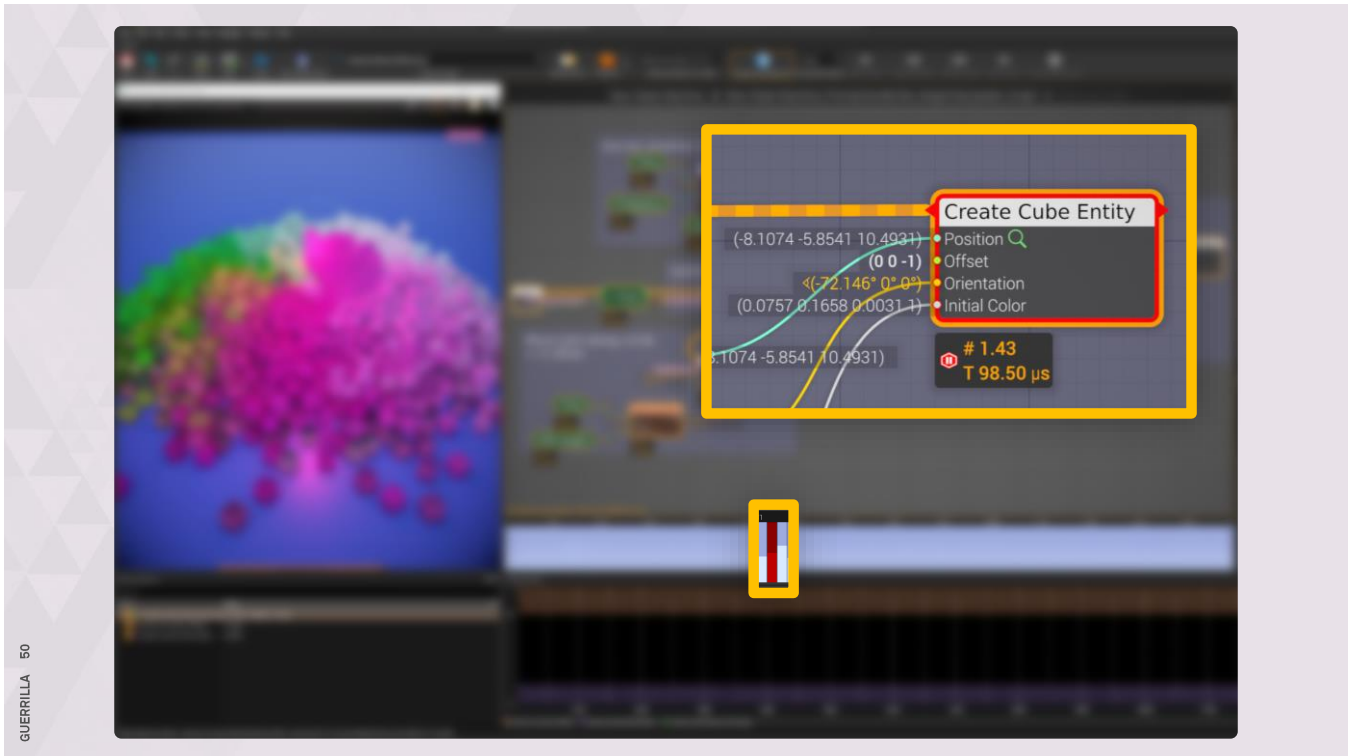
Helps to find expensive outlier frames.

Zooming in shows frame time spent.

Yellow bars shows “change” :

- Helps to find state machine transitions, or
- When control flow path changes

Red bars show breakpoint hit



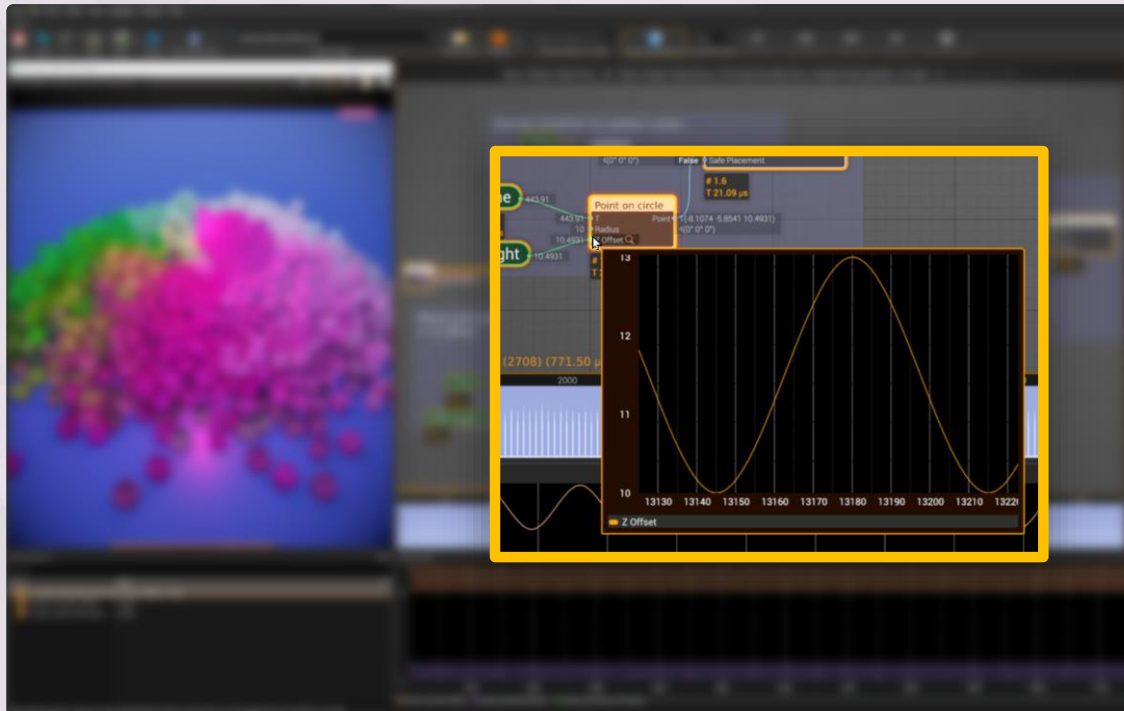
Breakpoints are useful for:

- pausing game when specific node executes (e.g. inspect game state)
- finding node execution without browsing timeline first

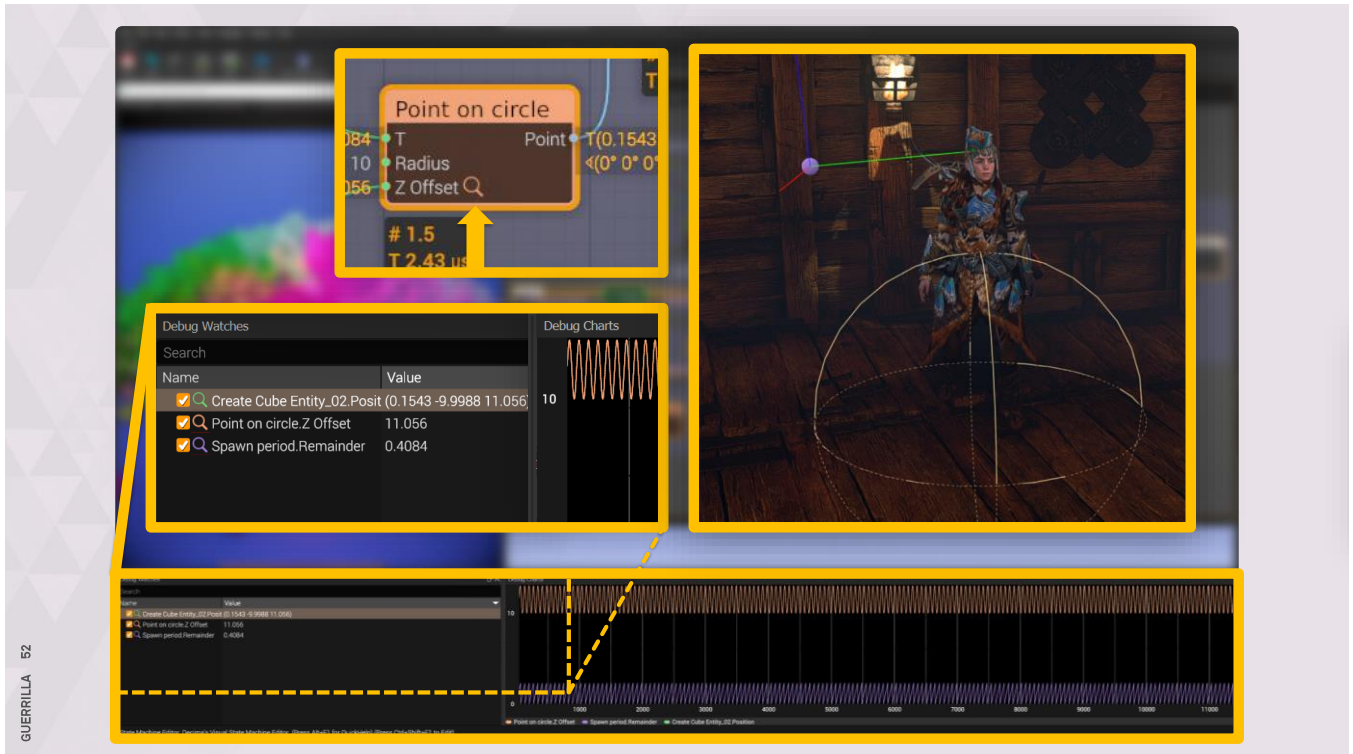
When hit:

- debugger window focusses on node
- node shows red outline
- red frame in timeline.

(Extra: Show a red indicator under node.)



Can inspect values on nodes on the canvas and hover over pins for graph over time



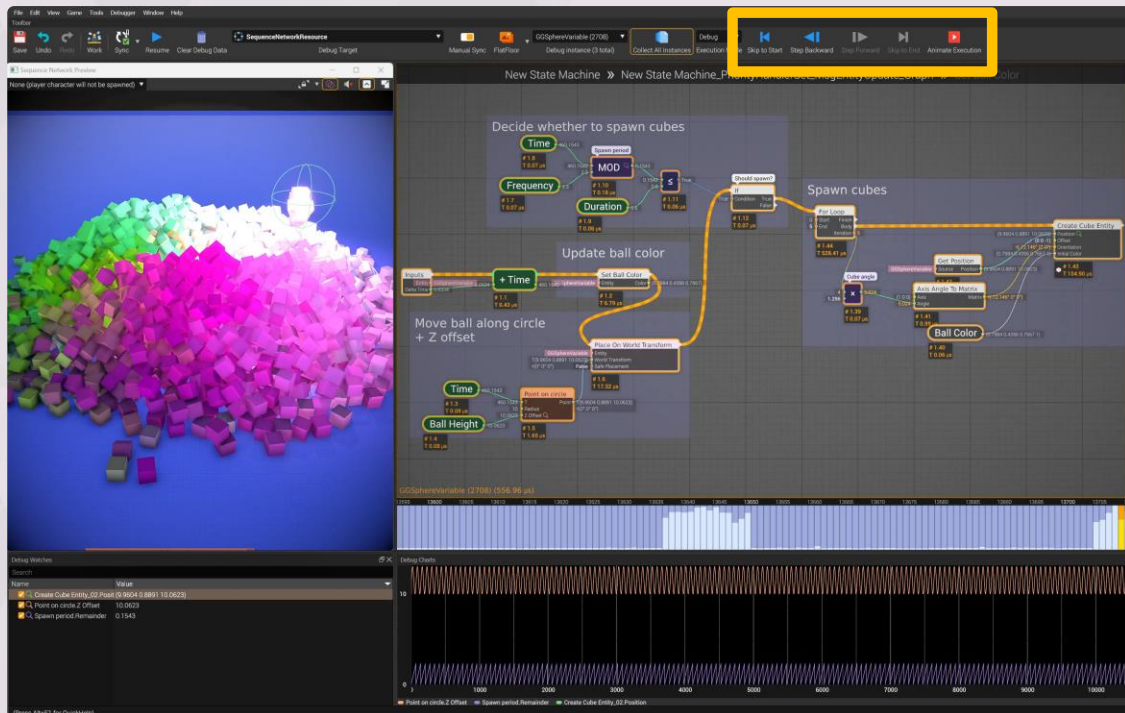
Sometimes inspecting single values on canvas is not enough.

Want to compare values over time, or maybe even in-game.

Set per pin on nodes. Shown with a magnifying glass icon on pin.

Selected pins also end up in a list view as shown here.

Values shown in graph view, or in 3D game view if possible.



Have ability to step through execution frame node-by-node.

Can right click nodes to skip to that node's execution.

Useful for:

- inspecting order of node execution or loop iterations.
- Investigate if order is as expected or not

DATA COLLECTION

- Does **not** rely on existing debug format (e.g. DWARF)
- **Debug-only macros** sample values and mark events (*node start/end, ...*)
- **Serialize** data to socket-based binary stream
 - Handle **breakpoints**
 - Sent to **editor**

```
void Graph(float i_t, float* o_Result)
{
    // 'GetTimeOfDay' node starts here
    _STORE_BEGIN_NODE(1 /* id */)
    /* [IRExportedFunctionCall] */
    float time = 0.0f;
    Game::GetTimeOfDay(&time);
    _STORE_OUTPUT(time)
    _STORE_END_NODE()

    // 'Add' node starts here
    _STORE_BEGIN_NODE(2 /* id */)
    _STORE_INPUT(i_t)
    _STORE_INPUT(time)
    /* [IRAssignment] */
    *o_Result = i_t + time;
    _STORE_OUTPUT(o_Result)
    _STORE_END_NODE()
}
```

Implemented by annotating generated code with debug-only macros.

Compiled out in release.

Macro examples shown on right side:

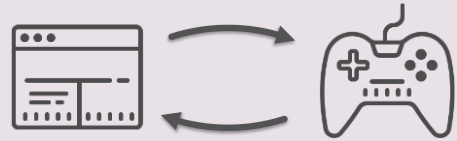
- Surround each node
- Store node UIDs, begin/end markers, input/output values
- Save to some buffer
- Also for state machine transitions

After each node:

- Flush buffer to engine
- Handles breakpoints
- Serializes to socket-based binary steam

EDITOR COMMUNICATION

- **Editor stores** binary stream as traversable callstack-like data structure
- **Interpret data** as needed for visualizations
- Editor **sends debug requests** to game:
 - Graphs to **debug**
 - Nodes with **breakpoints**
 - Watches to **draw in-game**



Editor receives EventTrace data > stores in traversable data structure.

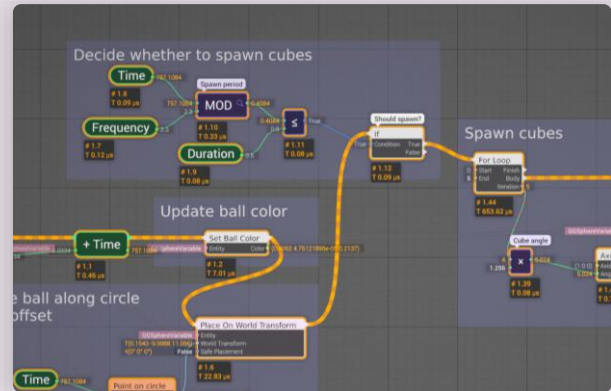
Reads and parses data as needed to visualize data.

Editor sends debug requests to game:

- Selected graph instance to collect debug data from
- Breakpoints to break on
- Watches to visualize in-game in 3D space

DEBUGGER

- Real-time *and* back-in-time debugger
- Visualizes **values** *and* **control flow**
- Attaches to **any DECIMA target**
(Editor, Standalone game, DCC, etc.)



Lets summarize:

The debugger supports real-time and back-in-time debugging.

Has support for per-node breakpoints and stepping.

Visualizes values and control flow both in-editor and in-game.

Enables coarse-grained performance profiling of Node Graphs.

Attaches to any DECIMA target:

- Editor, Standalone game, DCC with DECIMA integration (e.g. Maya).

RESULTS

- Node Graph execution **no longer a black box**
 - **Autonomized** users for debugging
(*though still got semantic questions about flow, etc.*)
- Saved a lot of **users' time**
- Saved a lot of **programmers' time**
- Success!



The debugger was a smash hit! For users, gained insight in what was happening.

Node graph execution was no longer a black box. Users could independently debug their content now.

As a result, practically overnight -> Programmers no longer needed for help with debugging

So, saved a lot of user time. Saved a lot of programmer time. Both sides love it!

Overall, great success!

History

Graphs & State Machines

Asset to Runtime

Debugging

Usability

Animation & Shader Graphs

USABILITY

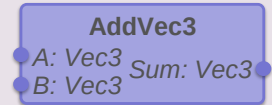
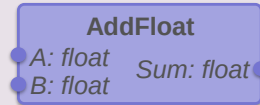
- **Maintaining** node variants
- **Exposing variables** to systems
- **Organizing** data with **user-defined types**



Too many usability features to discuss, so have to focus on some important ones

NODE VARIANTS

- Had to implement many **type variants of nodes**
 - *AddInt, AddFloat, AddVec3, etc.*
- **Commit** to variant **when spawning** node
- Takes **effort** to **change** type in graph



```
static inline void AddFloat(const float inA, const float inB, float* outSum)
{
    *outSum = inA + inB;
}
```

```
static inline void AddVec3(const Vec3& inA, const Vec3& inB, Vec3* outVec)
{
    *outVec = inA + inB;
}
```

More on reusability.

Many nodes + many types = Many type variants of similar nodes (AddInt, AddFloat, AddVec3)

Effort to implement multiple variants

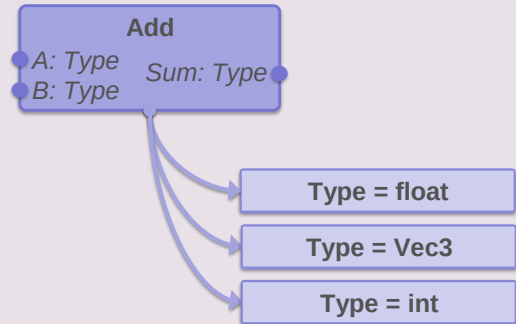
User commits to type when spawning the node, but then takes effort to change afterwards.

Any time user had chain of math nodes for example, and data type needed to be changed. Needed to change out all nodes manually. Replacing nodes, reconnecting wires, etc.

Wanted better solution that saved users time, and if possible also author of nodes.

TEMPLATE NODES

- Merge type variants into **single definition**
 - `Add <int/float/Vec3>`
- Pin types become **type constraints**
 - `<Type>`, `Number`, `Enum`, `DerivedFrom`, ...
 - Fewer **permutations**
- Code uses templates (C++) or overloads



```
template <typename taType>
static inline void Add(const taType& inA, const taType& inB, taType* outResult)
{
    *outResult = inA + inB;
}
```

Solution merges type variants into single definition.

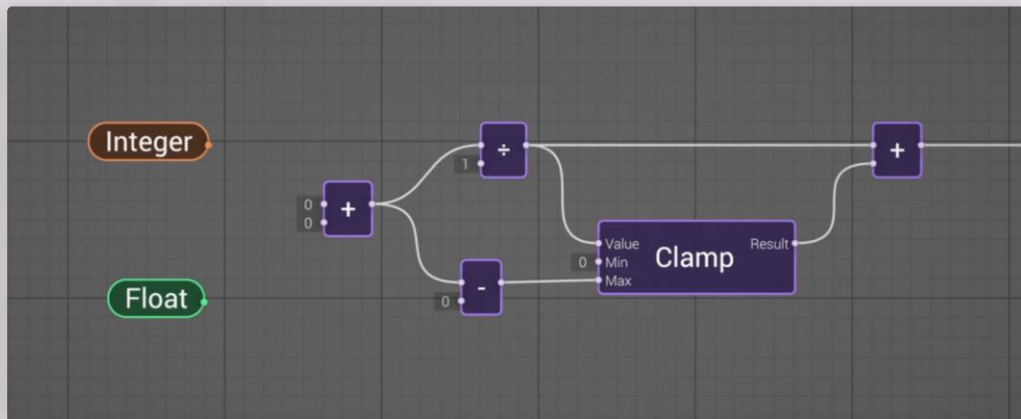
Example: A single Add node that can take either int, float, or Vec3

Took inspiration: C++20 “constraints and concepts”.

Replace pin types with type constraints

- “Number”
- “Enum”
- “DerivedFrom”

Implementation can use overloads / templates / IR generator can generate exactly typed IR.

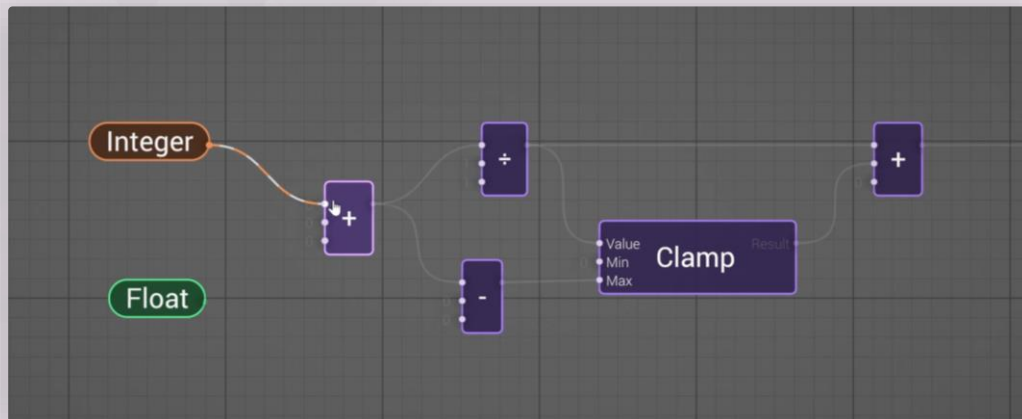


DYNAMIC TYPE RESOLVING

We dynamically resolve types in the editor and during conversion, and don't store the results on disk

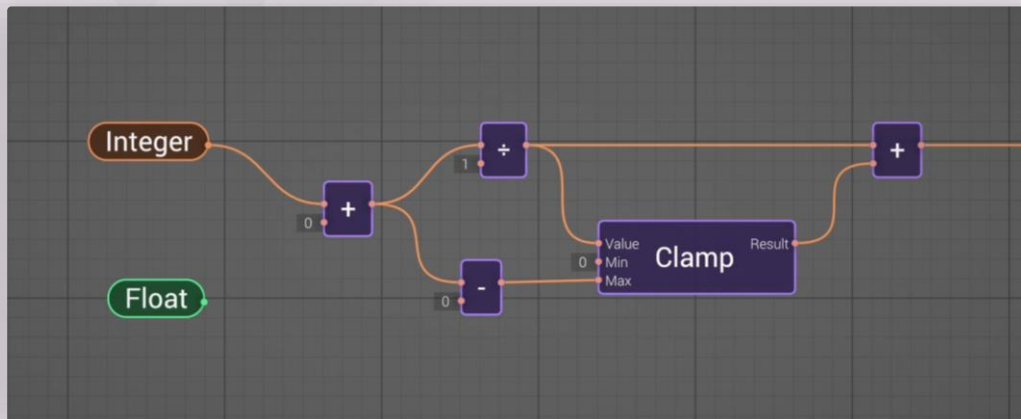
Lets look at editor workflow. In this situation we have a network of math nodes that do not know yet what data type they operate on.

- GRAY wires = Unknown type
- ORANGE = Integer
- GREEN = Float
- WIRE multi color



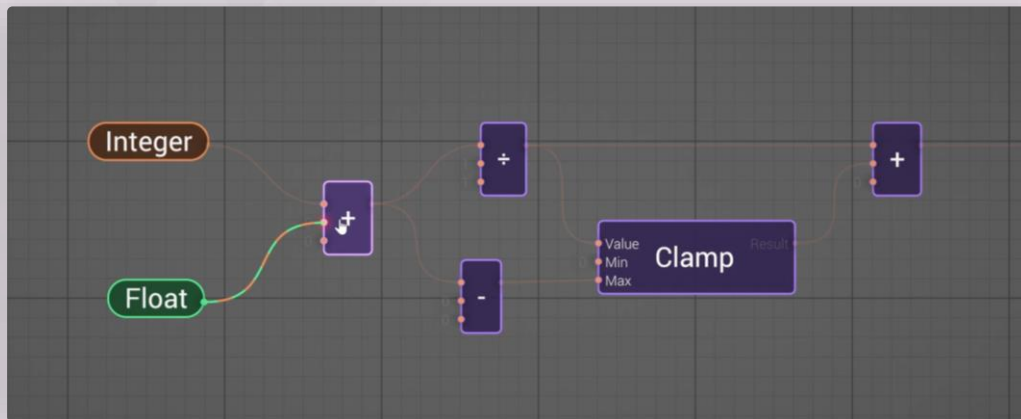
DYNAMIC TYPE RESOLVING

Start by connecting the integer to the math nodes



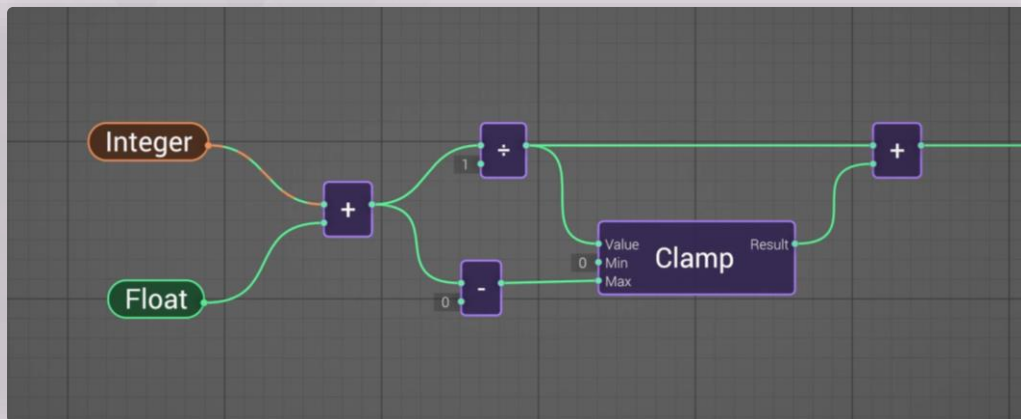
DYNAMIC TYPE RESOLVING

When we connect the integer, the math resolves to integers



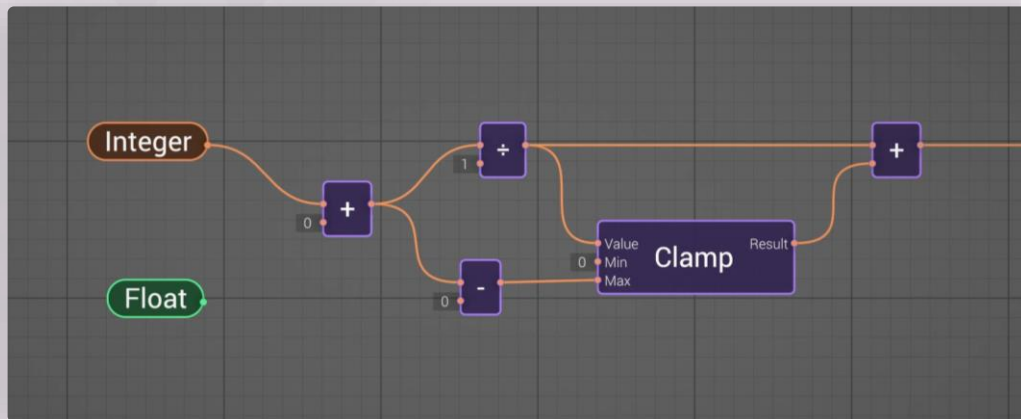
DYNAMIC TYPE RESOLVING

Now we also connect a float.



DYNAMIC TYPE RESOLVING

Now all the math nodes resolve to float because we have type precedence and prefer the type with more precision.



DYNAMIC TYPE RESOLVING

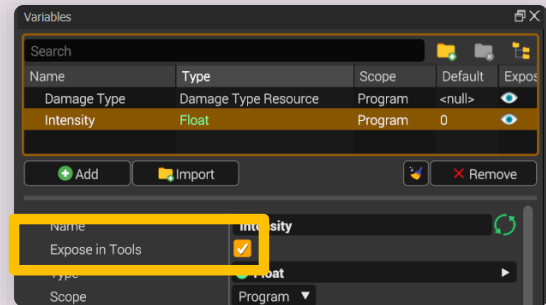
Disconnecting the float resolves to integer again.

Disconnecting the integer then resolves to unknown again.

This happens continuously in the editor.

EXPOSED VARIABLES

- **Expose** variables to **other systems**
 - *E.g.: Cinematic Sequencer, UI Editor, Maya, etc.*
- Other system chooses how to **drive variables**
- *E.g.:*
 - *Curve-over-time*
 - *Sequencer event progress*
 - *Maya Locator position*



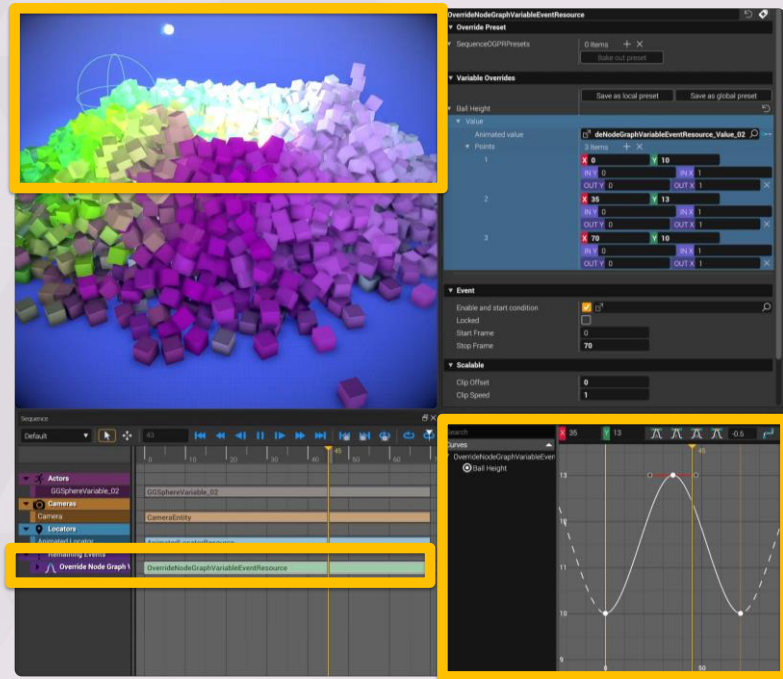
Another way approached usability was to allow variables to be exposed to external systems, such as:

- Cinematic sequencer
- UI editor
- Maya

Generic UI for constant values/resource links; Extensible for domain-specific sources, like:

- Curve-over-time
- Sequencer event duration
- Entity position

Code can also manually read/write exposed variables as needed



This is Cinematic Sequencer where variables tend to be heavily used.
Same example Node Graph as before.

Ball height is exposed and controlled by sequence editor here.

Sequence controls ball height with curve.

Can see changing curve from the sequence affects the behavior of the ball.

Node Graph is unaware of sequence, but sequence can drive behavior through exposed variable.



Example use-case for this was the “lighting rig” system. This system is node graph-based and drives the position, intensity, and color, of lights around characters’ heads in cinematics. The colored balls seen here are lights, being driven by a node graph. The node graph in turn is being driven by the cinematic sequence through exposed variables. This way, the lighting rig logic was user-defined in a node graph once, and then each cinematic can use the same node graph. This gives a unified lighting rig set-up where the node graph variables define a user-friendly interface that is exposed to the cinematic sequence.

USER-DEFINED TYPES

- Engine types are defined in C++
- Custom DECIMA Run-Time Type Information (RTTI) system
 - **Serialization**
 - **Reflection**
 - **Attribute Editors** in editor
- **Exposing** engine types to Node Graphs requires **programmer support**

```
RTTI_START_COMPOUND(SpotLightResource)
{
    RTTI_START_ATTRS(SpotLightResource)
    {
        RTTI_ATTR(SpotLightResource, "ConeAngle", RTTI_OF(float), &mConeAngle)
        RTTI_ATTR(SpotLightResource, "HotspotAngle", RTTI_OF(float), &mHotSpotAngle)
        RTTI_ATTR(SpotLightResource, "NearRange", RTTI_OF(float), &mNearRange)
        RTTI_ATTR(SpotLightResource, "FarRange", RTTI_OF(float), &mFarRange)
    }
    RTTI_END_ATTRS(SpotLightResource)
    RTTI_START_EXPORTED_SYMBOLS(SpotLightResource)
    RTTI_END_EXPORTED_SYMBOLS(SpotLightResource)
}
RTTI_END_COMPOUND(SpotLightResource)
```

First, some background about Runtime Type Information in DECIMA.

Engine types are defined in C++ code.

Includes Classes, Enums, containers, pointers, etc.

DECIMA has a custom RTTI system that provides:

- Serialization
- Reflection
- Attribute Editors in editor
- and more...

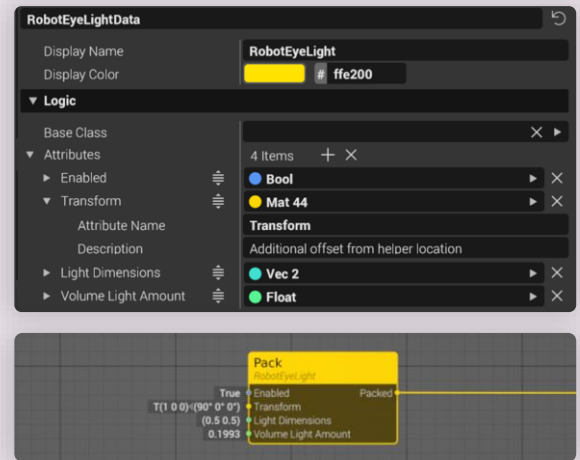
Exposing types to Node Graphs relies on this RTTI system

All of this requires programmer support, but also wanted users to define their

own types

USER-DEFINED TYPES

- Content-driven but functions like any engine type
- Compounds and Enumerations
*E.g.: **Event** messages or **Telemetry** payloads*
- No programmer support needed
 - Extensively used now



User defined types are content driven classes and enums

Act like any other runtime-type-information type

No programmer support needed, but allowed class member types dictated by programmers in engine.

Classes pack data values as single unit

- Supports inheritance for
- entity/engine messages to communicate between content
- even telemetry system uses this for content-defined payloads

Enums safe alternative for String or Integer identifiers. Content refers to descriptive enum labels, so changing order/values/options in enum doesn't require changing relevant content.

Powerful system. Used extensively once it came online.

Again, making users independent from programmers

History

Graphs & State Machines

Asset to Runtime

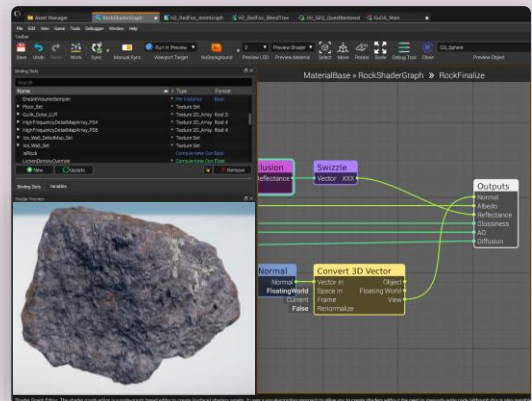
Debugging

Usability

Animation & Shader Graphs

SHADER GRAPHS

- Move visual shader authoring to **DECIMA** editor
- In-editor preview with **real game data** (*terrain ecotone, rain wetness, etc.*)
- Unified “**surface material**” graph



Already had system of Maya shader graphs translating to DECIMA representation.

This approach was okay, but wanted more integration with DECIMA.

Wanted better previewing; in-situ with real game data.

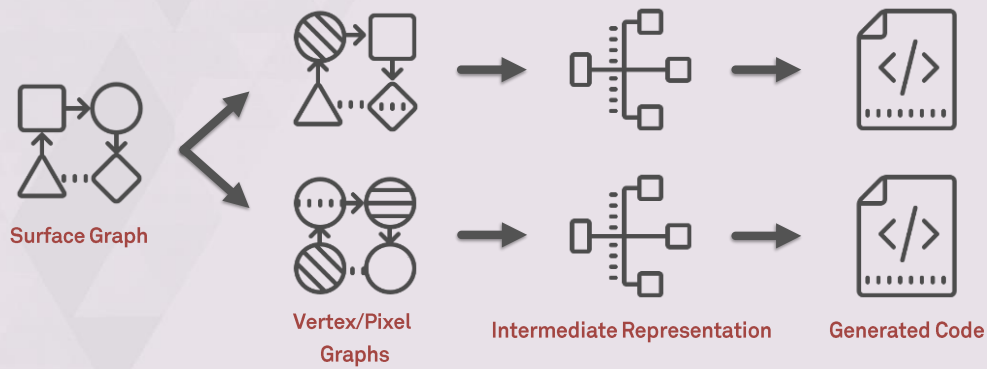
And since we already had the Node Graph framework...

...decided to built Shader Graphs on top of that too.

Didn't want pixel/vertex shading passes exposed to user.

Graph should describe properties of surface (albedo, reflectance, etc.).

VERTEX / PIXEL SEPARATION



Since single graph represents separate passes, so conversion process looks like this.

Surface graph is analyzed; then split into vertex/pixel graphs.

Intelligently puts some logic on either side, depending on GPU interpolator capabilities.

Those are turned into intermediate representation as usual by framework.

That is turned into generated code by shader backend.

REDUCE, REUSE, RECYCLE

- Re-used (and extended):
 - **Asset structure**
 - **IR phase**
 - **Node definitions** (*math/utility/etc.*)
 - With **shader code** implementation
 - **Editor code**
- Bespoke **shader code** backend
 - Extends existing C backend



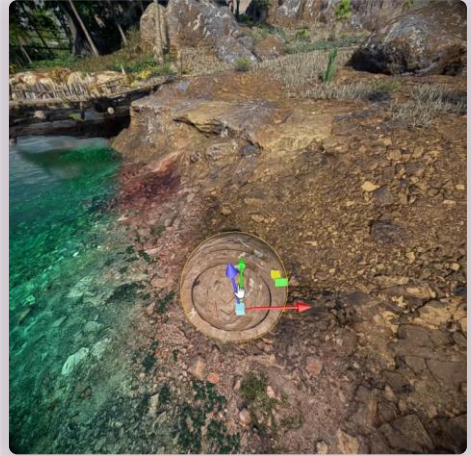
By using this approach we could re-use a lot of existing code with some bespoke extensions:

- Re-used asset structure (with some derived classes)
- Re-used the IR phase
- Re-used most math/utility node definitions (but with shader code implementations)
- Re-used editor code

Bespoke shader code backend but even this builds on existing C backend.

RESULTS

- On HFW release, **all** *character/building block/vegetation* shaders were Shader Graphs
- **Authored** fully in **DECIMA editor**
 - Nodes and features 100% in our control
- **Editor Preview** is also DECIMA itself:
 - Loads **real game data**
 - **Live-tweaking**
 - **Hot-reloading**



On release, **all** character, building block, and vegetation shaders were Shader Graphs.

We could author and preview shaders in DECIMA

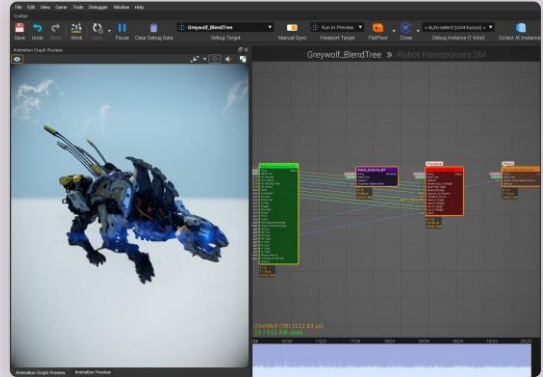
Preview is DECIMA itself instead of third-party renderer, so:

- Loads real game data
- Supports standard DECIMA live-tweaking and hot-reloading of assets

All in all, the Shader Graphs were **also** a success.

ANIMATION GRAPHS

- Runtime **character animations** with State Machines
- **Replace** middleware with similar workflow
- States **evaluate pose** with graphs
 - Event Graph > Animation **Blend Tree**
 - **Multiple active states** for blending
(Stand/Jump/Turn/Walk/etc...)



During development of Horizon Forbidden West, work started on expanding Node Graphs to Animation Graphs.

Planned from the start

Goal: build on Node Graph framework: support driving character animations.

Node Graph framework mature

Some changes were needed:

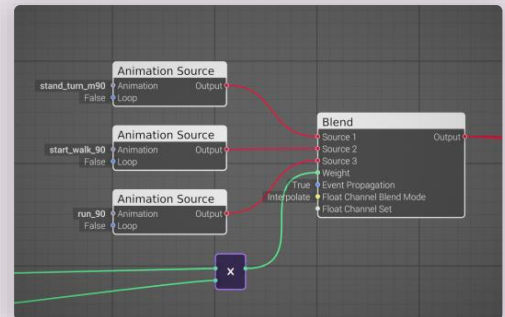
Multiple active states for blending between states

State event graph becomes animation blend tree

This means that states evaluate and produce the animation pose to display with pose evaluation graphs.

BLEND TREE

- Graph without Control Flow, resulting in a pose
- Still skip “inactive” nodes
E.g.: Don't sample animation with blend factor 0
- Different execution model
 - **Phase 1:** Choose active nodes (left-to-right)
 - ...
 - **Phase N:** Evaluate *only* active nodes (right-to-left)



...For these we wanted a different execution model without Control Flow, but still skip “inactive” nodes

(For performance. Example: only want to sample poses to blend if we actually use them)

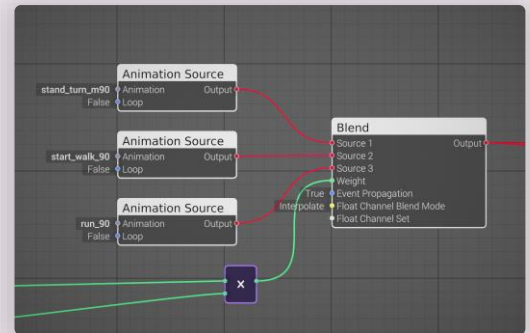
This was different execution model:

- Nodes become multi-phase
- Phase 1: Determine active inputs (left to right)
- Phase 2: Evaluate **only** active nodes (right to left)

Image: A blend node takes three separate sampled animations as input and blends them together. If the blend weight is 0, only the first animation should be sampled; the rest is redundant.

BLEND TREE

- User **shouldn't care** about execution model
- Framework **accommodates!**
- Phases **entirely** in IR and node implementations
 - **Opaque** to the user
 - **Editor UX** like any other node graph



But don't want user to think about this; graph should be just data dependence graph.

Fortunately, framework accommodated!

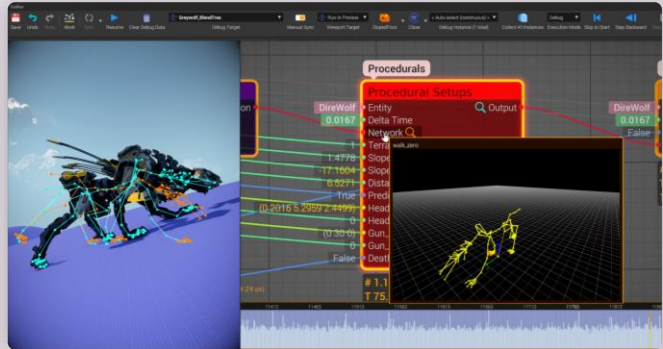
All implemented in Intermediate Representation and node implementations.

Opaque to the user

Graph in editor doesn't show phases, just a data dependency graph like they are used to.

EDITOR IMPROVEMENTS

- Domain-specific debug visualizations
- Control Parameters
- Scrub in-game state with timeline



Ability to preview on characters in editor

Domain-specific debug visualizations

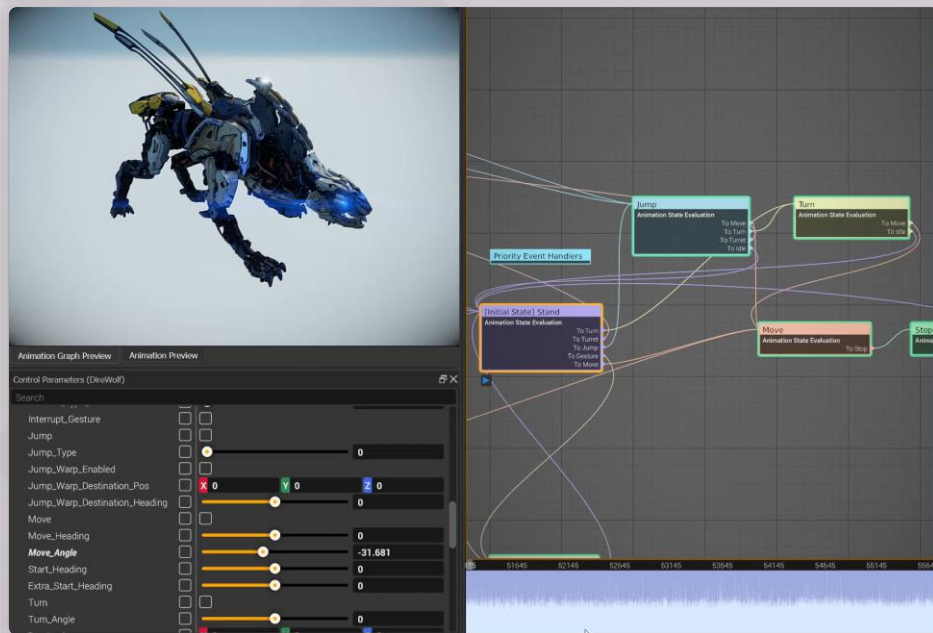
Implemented using debug watch system.

Uses existing data

Added scrubbing game state along with node graph timeline

And added something called Control Parameters.

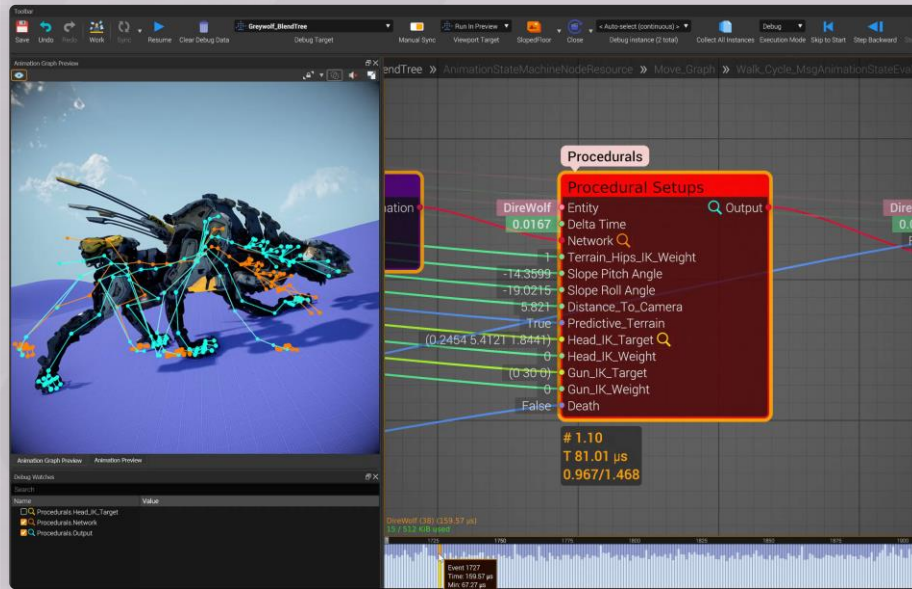
Let's look at last two



CONTROL PARAMETERS

Control Parameters are a way for users to drive node graph inputs without first hooking it up to an engine system, or to a controller setup. This allows users to quickly test-drive their animation graph content.

Control Parameters are automatically exposed as widgets in the UI seen above, but users can also define another Node Graph to procedurally drive them if desired.



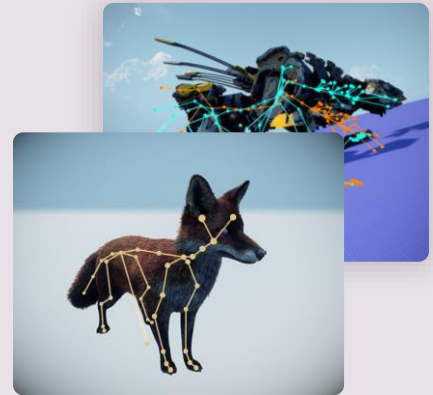
GAME STATE SCRUBBING

In this video:

- Scrubbing timeline; robot entity scrubs with it.
- Two debug watches shows pose in 3D view before and after the red node
- ORANGE pose is ****before**** red node on canvas
- BLUE/CYAN pose is ****after**** red node on canvas has applied procedurally corrections for terrain slope

RESULTS

- **Expected** limited use in HFW (*e.g. small animals/robots*)
- Framework proved **robust yet extensible**
- Ended up with (*almost*) **all animals and robots**
 - **Complete** and **production-proven** engine feature



When work began on Animation Graphs, expected to use it for only some characters in HFW (e.g. small animals and some robots)

But implementation was easier than expected because the framework proved robust yet customizable.

So upon release of HFW, ****all**** animals and robots used the new Animation Graph system!

Now, Animation Graphs are a complete and production-proven engine feature of DECIMA in their own right.

History

Graphs & State Machines

Asset to Runtime

Debugging

Usability

Animation & Shader Graphs

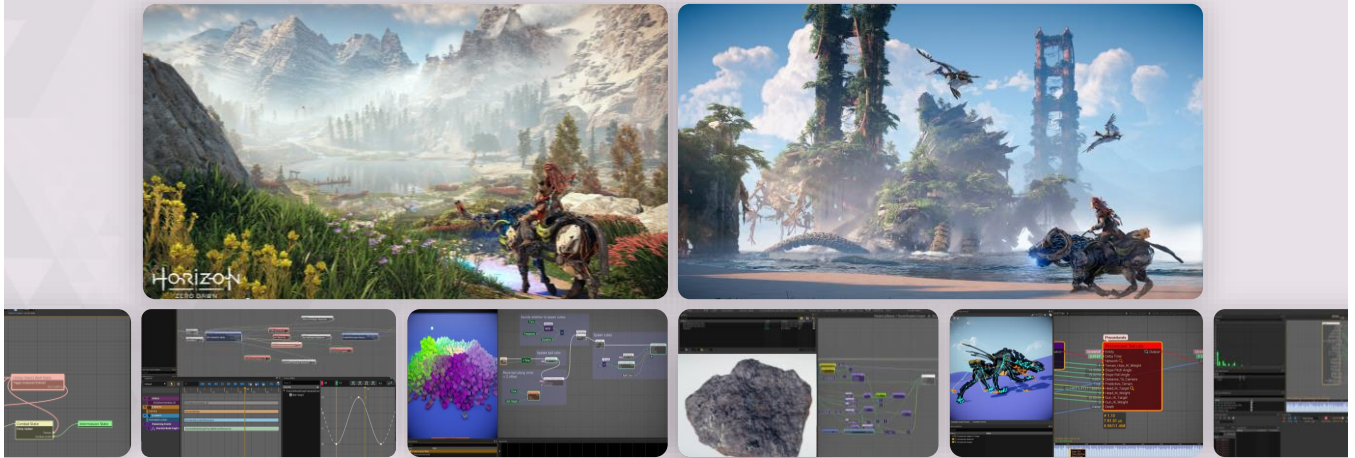
Looking back & ahead

ABOUT A DECADE AGO...



About a decade ago we had this challenge when moving from Killzone to Horizon, and we only had Node Graphs for sound scripting.

...OVER A DECADE LATER



GUERRILLA 87

Over a decade later we have managed to make the huge world of Horizon that we set out to make. We were able to make all the content we needed to fill it with by extending the Node Graph framework to more domains.

Then we did it again for Horizon Forbidden West and extended the Node Graph framework even more.

In the meantime Kojima Productions also made Death Stranding with the same tools.

And now the Nodegraph Framework has many more domains than we started with.

TAKEAWAYS

- We're happy with this direction!
- Visual programming...
 - ...Made our developers **independent**
 - ...**Enabled** us to fill large, vibrant open worlds with content
- Invest in...
 - **Debugging**
 - **Iteration speed**
 - **Future domains** to **multiply** your ROI



Visual programming made our developers more independent from programmers.

Let us build huge, vibrant open worlds filled with content

As such, us Programmers slowly remove ourselves from this equation. This makes it all the more important to offer good tools for debugging **by** users.

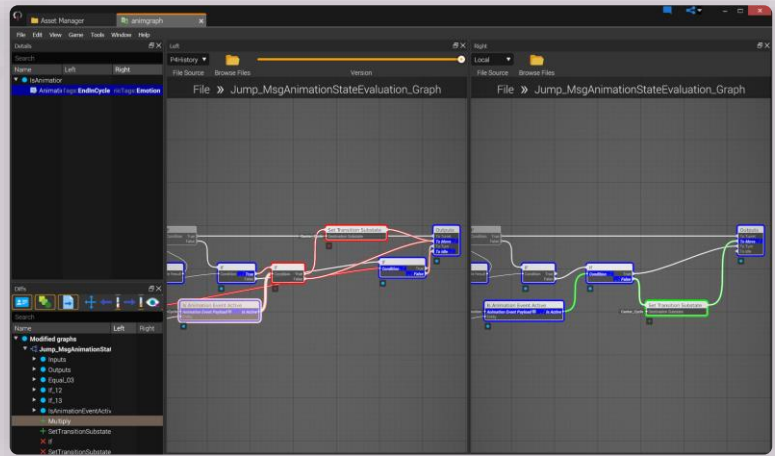
Same goes for maintainability **by** users: Strive for making easy to document, re-use, and re-factor content, etc.

Iteration speed > Debugging, ease-of-use, immediate editor feedback

We invested in extensibility by building a framework with future domains in mind, and reaped benefits from that approach (e.g. shader graphs, animation graphs, etc.). It multiplied our return-on-investment.

WHAT'S NEXT?

- Visual Diffing
- Function Graphs in user types
- More domains
 - Particle Effects
 - Audio DSP
 - Animation Rigs



Next for us:

We have been introducing visual diffing to make it easier to inspect changes to Node Graphs.

We want to extend our user-defined types with function graphs and go for an Object Oriented Programming approach.

We want to extend the framework to even more domains, such as:

- Particle Effect Graphs
- Audio Digital Signal Processing Graphs
- Animation Rig Graphs (i.e. author animations through a graph offline instead of blending existing animations at runtime)

SPECIAL THANKS

David Ramos De La Fuente
Maarten van der Gaag
Guido de Haan
Olaf Jansen
Floris Kappeyne van de Coppello
Michiel van der Leeuw
Jorrit Rouwé
Laurens Simonis
Remco Straatman
Dan Sumaili
Andreas Varga
Shawn White
Anton Woldhek

THANK YOU WE'RE HIRING!

MORE DECIMA TALKS

'Scaling Tools for Millions of Assets for Horizon Forbidden West' (2024) – *David Marcelis*
'Architecting Jolt Physics for Horizon Forbidden West' (2022) – *Jorrit Rouwé*
'Making Tools for Big Games' (2019) – *Michiel van der Leeuw*
'Creating a Tools Pipeline for Horizon Zero Dawn' (2017) – *Dan Sumaili & Sander van der Steen*
'The Next-gen Dynamic Sound System of Killzone Shadow Fall' (2014) – *Andreas Varga & Anton Woldhek*



bryan.keiren@guerrilla-games.com
Bryan Keiren



guerrilla-games.com/join



Special thanks to these people listed here.

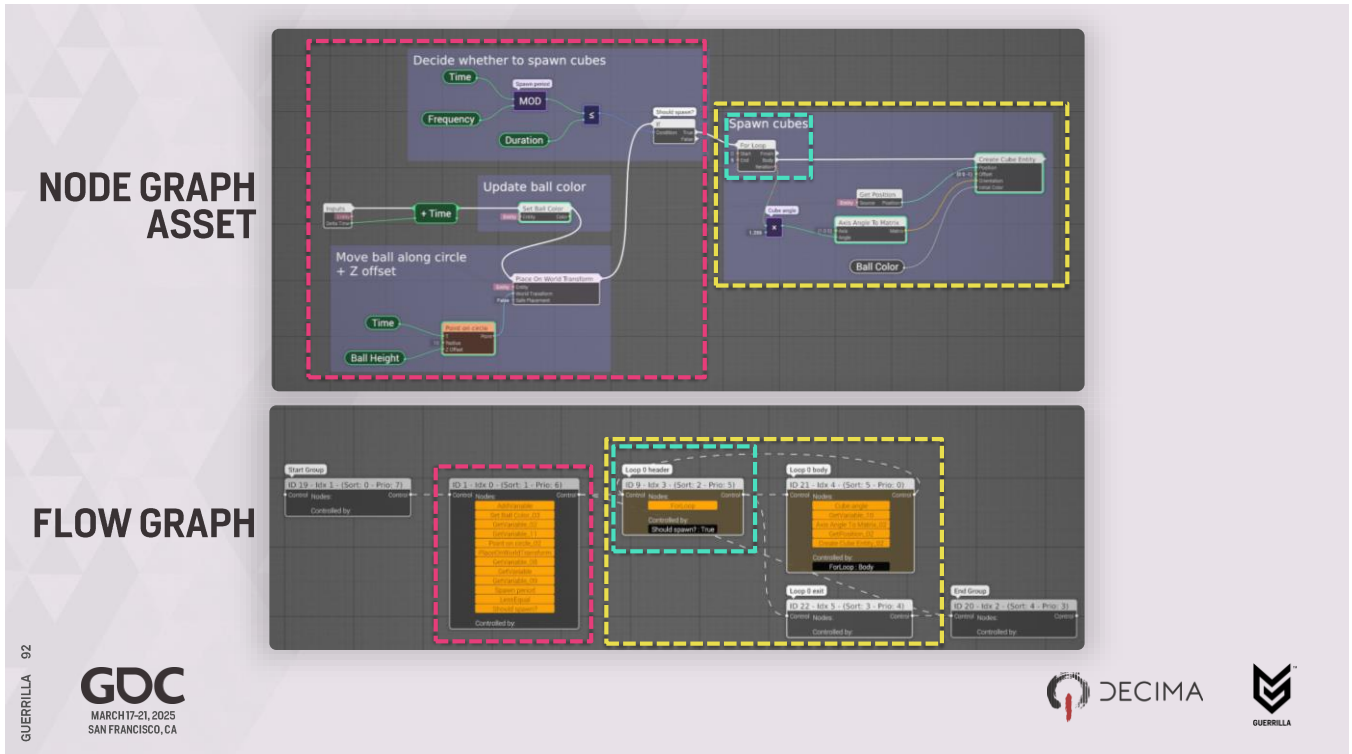
If you're looking for other cool talks related to DECIMA I can suggest these ones.

If you're looking for a really cool place to work, we're hiring!

Flow Graph

BONUS SLIDE: Flow Graph

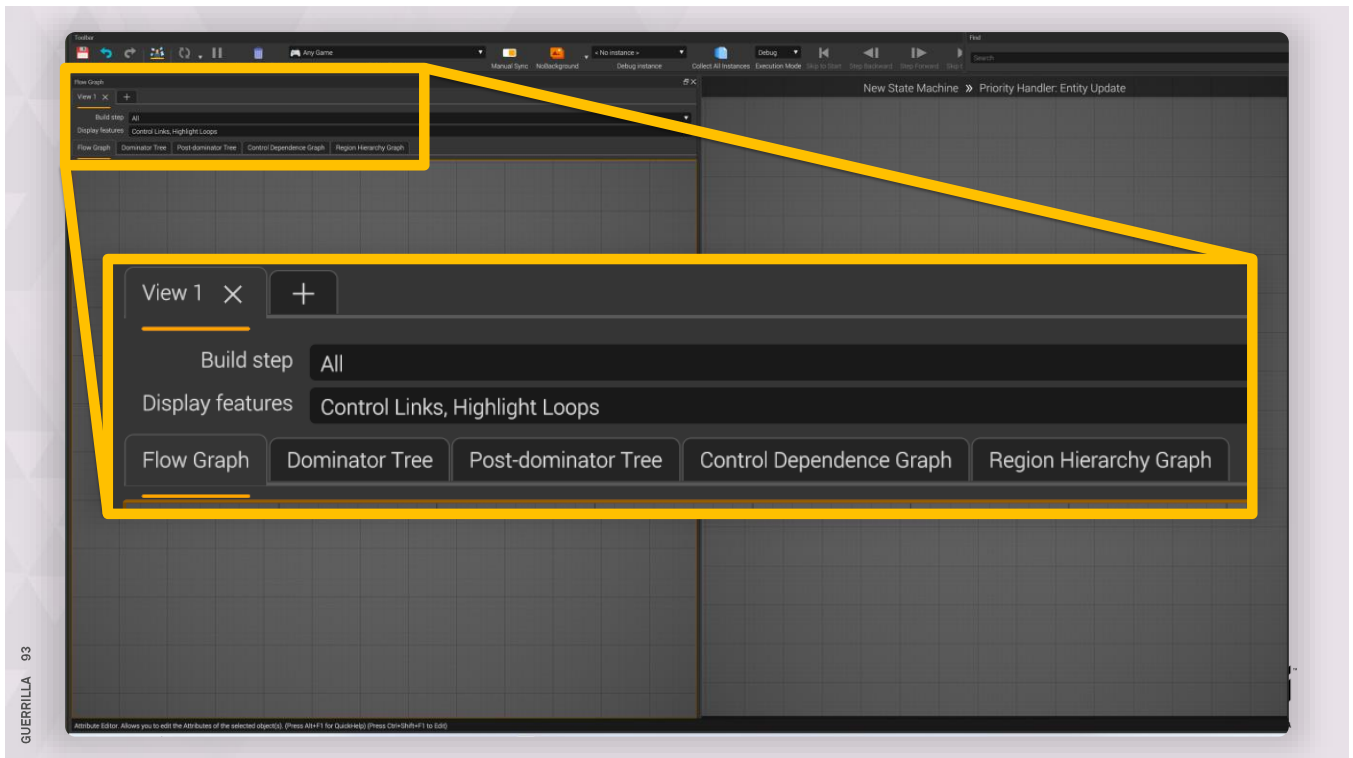
The flow graph is an intermediate data structure that we use to generate the IR graph for a given Node Graph. It helps us to identify the structure of the graph and generate appropriate equivalent code for it.



Top is a node graph assets

Bottom is the associated flow graph, visualized in our editor.

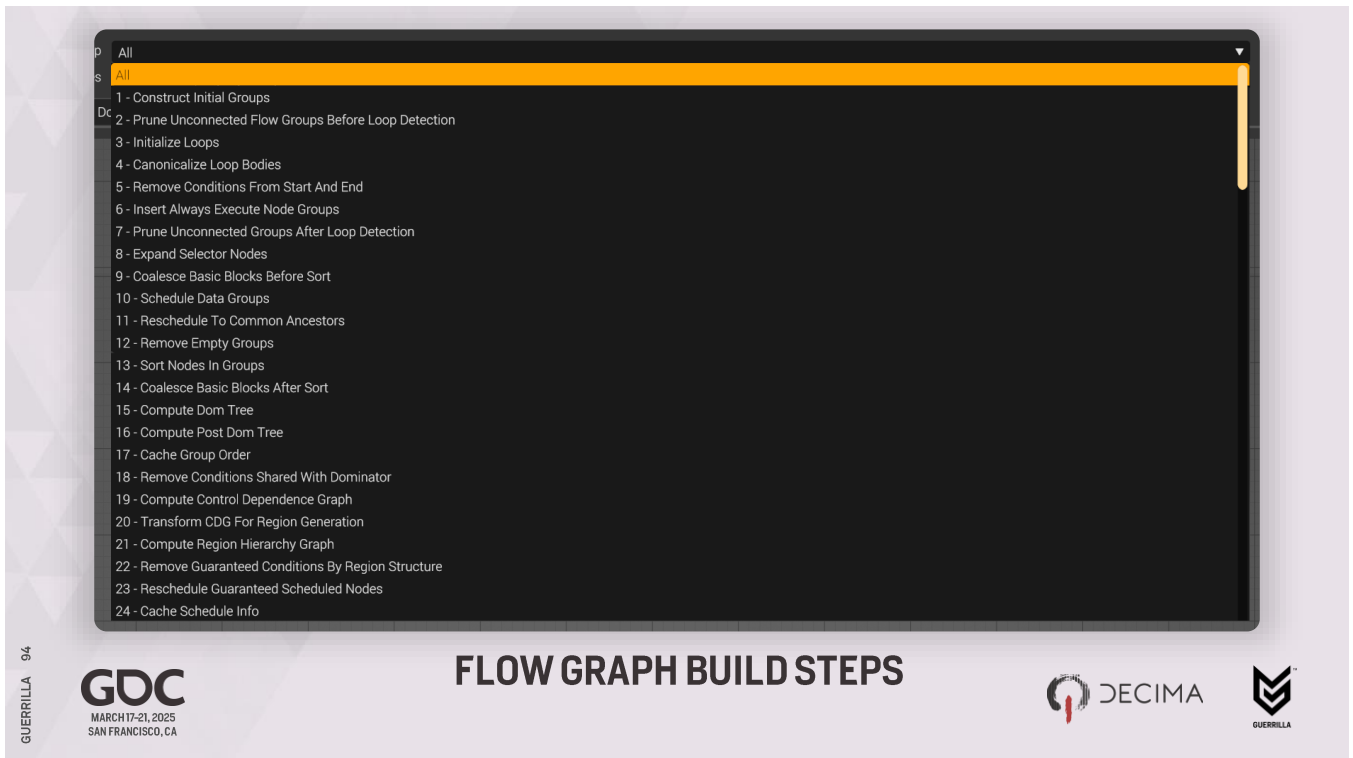
The bottom image shows how the network of nodes inside of the pink box has been grouped together as a single “block” of sequentially-executable nodes. Only at the end of this block do we reach a branching point (because of the “If” node). At that point the flow graph indicates a potential jump to the end of the node graph, or to the “For Loop” node. The “For Loop” node starts its own flow graph block, which then also correctly encodes the jumps to the loop body or loop exit, and the loop body itself shows the latch back to the loop header. By encoding these actual flow paths in the flow graph, we can make inferences about scheduling, grouping, and priorities within the flow graph by using concepts like (post-)dominator trees, and graph regions.



Have a view in the editor to visualize the flow graph for any node graph asset. Flow graph construction involves several other graphs as well

- Dominator Tree
- Post-dominator Tree
- Control Dependence Graph
- Region Hierarchy Graph

All of these are based on the Flow Graph and can help us infer information about the flow graph.



Flow Graph construction consists of these steps in our system.

Refactoring

BONUS SLIDE: Refactoring

REFACTORING

- I.e.:
“Replace/edit content to deprecated or optimize”
- Don't **know** all content
 - **WIP content** not yet in repository
 - **External studios** using DECIMA
- Need **forward compatibility**
- Prefer **automatic** over **manual** solution(s)



By refactoring I mean: Content or systems change over time, so we want ability to replace/edit content because of deprecated functionality or optimizations.

Example, see graphs on right side.

Don't know all content, because:

- WIP content is not yet in the repository when applying refactor
- External studios use DECIMA, and don't have control over that content

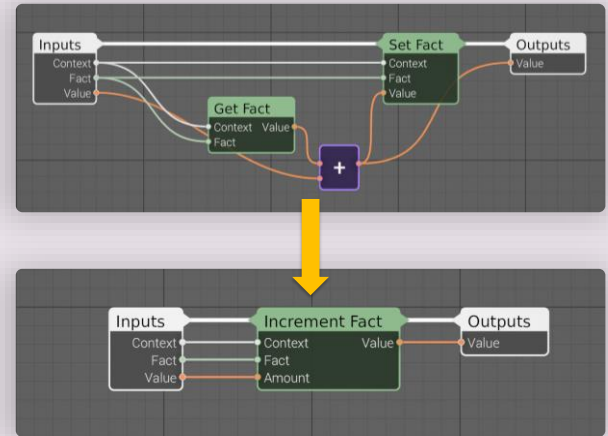
Because of that, we:

- Need forward compatibility
- Prefer automatic over a manual solution

Have some solutions...

REWRITE PATTERN GRAPHS

- **First** Node Graph refactor “tool”
- Define “**pattern**” and “**replacement**” graphs in content
- **Editor** matches and **asks to replace** at edit time
- **Content driven**, but must **maintain old content** as pattern graph



First attempt built long time ago

Define a “pattern” and “replacement” graph in content.

Editor matches at edit time/on open.

Asks to replace matches with replacement.

Has to be maintained, but is content driven.

PROGRAMMABLE REFACTORING

- Python API
 - **General**; usable for **any Decima content** change
 - **No versioning**
 - Script can **break** with time
- RefactorTractor
 - **General**
 - **Built-in** RTTI-based versioning
 - **Automated** part of **continuous integration**
 - Most **powerful** option
 - Only accessible to **coders**

