Behavior Graphs for glTF
Who am I?
Ben Houston.

Coder since the 1990s. Started in the games industry, then Hollywood Visual Effects and then Web 3D.

Of most relevance:
- I designed and contributed the Three.js Animation system, which I based on the best practices in Unity. This is similar to this project in many ways.
- Threikit is a company focused on ecommerce product visualization. We’ve lived with a Trigger-Action List and it has been incredibly limiting.
What are Behavior Graphs?

**Behavior Graphs** are an *Emerging Standard for Representing Behavior in Assets, Experiences and Levels within High-End Gaming & Metaverse Engines.*

They are a flexible and extensible execution model for node-based behavior systems.
The **Blueprint Visual Scripting** system in Unreal Engine is a complete gameplay scripting system based on the concept of using a node-based interface to create gameplay elements from within Unreal Editor. As with many common scripting languages, it is used to define object-oriented (OO) classes or objects in the engine. As you use UE4, you’ll often find that objects defined using Blueprint are colloquially referred to as just “Blueprints.”

Blueprint Classes are ideal for making interactive assets such as doors, switches, collectible items, and destructible scenery. In the image above, the button and the set of doors are each separate Blueprints that contain the necessary script to respond to player overlap events, make them animate, play sound effects, and change their materials (the button lights up when pressed, for example).
Unity Visual Scripting

Visual scripting in Unity empowers creators to develop gameplay mechanics or interaction logic using a visual, graph-based system instead of writing lines of traditional code.

Features

- Script Graphs
  Script Graphs are the main tool for creating interactions in your projects. Use node-based actions and values, these graphs let you execute logic in any order you specify, be it at every frame or when an event occurs.

- State Graphs
  State Graphs allow you to create self-contained behaviors that tell objects what actions to execute when they are in a particular state. They are suited for high-level logic such as AI behaviors, scene or level structure, or any aspect of a scene that requires behaviors that transition between states.

- Live editing
  Make changes to graphs in Play mode to see visual scripting updates in real-time. This approach offers a much faster way to iterate and test ideas without needing to recompile project changes.

- Debugging and analysis
  Visual scripting is designed to show values throughout the graph during Play mode, and it will highlight which nodes are being executed as it happens. If an error occurs at runtime, the source will be identified and highlighted within the graph.

- Designed for ease of use
  Increase accessibility for less-technical creators with user-friendly naming conventions, commenting and grouping features that keep graphs simple to read, and the Fuzzy Finder menu, which makes it easy to search for what you need.

- Codebase compatibility
  Use a method, field, property, or event from Unity, any third-party plugin, or even custom scripts within your graphs. Visual scripting directly accesses your codebase via reflection and is always up to date.

Create scripting logic visually

Visual scripting in Unity helps team members create scripting logic with visual, drag-and-drop graphs instead of writing code from scratch. It also enables more seamless collaboration between programmers, artists, and designers for faster prototyping and iteration.
What is OmniGraph?

OmniGraph is the compute engine of Omniverse. It allows the worlds in Omniverse to come alive with behavior. OmniGraph addresses all sorts of computations, from deformers to particles to event based graphs and more. OmniGraph is not a single type of graph, but a composition of many different types of graph systems under a single framework. In this first release, we are focusing on action graphs, which allow event driven behaviors. Also released is particle graphs, which allow the creation of particle systems. More graph types, such as deformer and other graphs, will be released in time.
Behavior Graphs are a Super Set

Behavior Graphs are a super set of Trigger-Action Lists.

- This means that any Trigger-Action List-based system, like Apple Realty Composer or Adobe Aero, can export to Behavior Graph-based systems.

- But Behavior Graphs-based systems, like Unity or Unreal Engine or Omniverse, will have trouble exporting efficiently to a Trigger-Action List, if they can be exported at all.

Because Behavior-Graphs also allow for low level nodes, because of their flexible execution model, they do not require as many built-in nodes to be defined, while offering more expressiveness.
Terminology

**Behavior Graph** - a set of interlinked Behavior Nodes, forms a directed, acyclic graph (DAG.)

**Behavior Node** - a single node in the Behavior Graph

**Socket** - either an input or an output of a Behavior Node

**Socket Value Type** - the typing of the Socket, either number, boolean, execution, etc.

**Link** - the linkage between an output Socket and an input Socket.
Categories Nodes

**Events** - Entry point into a behavior graph. Entity lifecycle, user interactions, timeline events.

**Actions** - Cause changes to occur in the scene. Trigger animation, set scene graph properties, play a sound.

**State** - Read and write asset state.

**Queries** - Request live data from scene graph, user, player or environment.

**Logic** - Basic data manipulation. Constants, math functions, boolean logic, etc.

**Flow** - Controls the flow of execution. If statements, loops, sequences, delays.
Flow Sockets and Links

The execution model is driven by Flow-type sockets and links. These are the white arrows on the flow nodes and the white lines connecting them. This is the path down which execution flows.

Flow sockets and links is what makes the execution model unified and flexible.

(Not to be confused with Wikipedia’s “Flow-based programming” article.)
Security Model

Behavior graphs, while they oriented as a means of visual scripting, also exhibit an excellent security model.

**Constrained Sandbox** - All behavior that a behavior-graph exhibits is through client supported node types. You can restrict any action you want by not supporting those node types.

**At-Will Execution** - Because execution steps are controlled by the host, e.g. how many nodes to process in a time slice, it is not possible to DOS a client which enforces limited time slices, even if an asset somehow gets into an infinite loop.

**Load-Time Validation** - In order to deal with a) corrupt, b) improperly structured, or c) unsupported node types, one can employ efficient load time validation to determine if one should even execute a given behavior-graph.
Future Proof / Deeply Extensible

There are a ton of straightforward extensions (available on other platforms) that can enhance capabilities, mostly by just adding additional node definitions:

**Control Flow** (Unity, Unreal Engine, Omniverse.) Loops, sequences.

**State** (Unity, Unreal Engine.) Allows for abstract state. Can be “simulated” using non-visible scene state as a workaround.

**Custom Events** (Unity, Unreal Engine.) Allow for custom triggers of behavior. Can be used internally to reduce code duplication and organize graphs.

**Public Interfaces** (Unity, Unreal Engine.) Enables introspection and composability of assets (e.g. bring together a few to create an AR experience or game level.)
Common Arguments Against Behavior-Graphs

1. Turing Completeness
Pretty much all behavior systems, even Trigger-Action Lists, are Turing complete because they can emulate variables (e.g. store and retrieve data in unseen parts of the scene graph) and they can loop (e.g. behavior a “raise” an event that then re-executes the behavior.)

Instead of trying to prevent Turing Completeness, it is more realistic and useful to design a system that can deal with misbehaving behaviors by ensuring a sandbox and the ability to limit execution resources.

2. Insecure
Both Trigger-Action Lists and Behavior-Graphs have identical security models if the same node types are supported. This discussion should be able the node types actually supported, rather than the difference between execution models.
Common Arguments Against Behavior-Graphs

3. Complex Implementation

Behavior-Graphs, because they have a simple and unified execution model, are actually fairly easy to implement. To help dispel the idea that Behavior-Graphs are complex, we’ve written and open sourced a Behavior-Graph execution engine:

https://github.com/bhouston/behavior-graph

4. I Want a Trigger-Action List Creation UX

The underlying representation of Behavior-Graphs does not require you to use Behavior Graphs in your user creation experience. You can convert Trigger Action Lists to Behavior Graphs as part of a straightforward distillation process. The converse isn’t true.
Why Behavior Graphs?

- **Simple** - They have a simple model for execution: There is a directed acyclic graph composed of immediate nodes and flow nodes.
- **Orthogonal and Expressive** - They can represent Trigger-Action lists and have simpler non-compound node definitions. Less code.
- **Safe** - Are executed within a constrained sandboxed environment defined by which nodes are supported.
- **Performant** - Because it is node-based, rather than scripted, execution can easily be limited to explicit time slice allocations.
- **Extensible** - It is easy to add additional features with just new node types.
Headless Behavior Graph Library

A bunch of us are working on a series of open source initiatives for an open behavior graph system.

The first component is an open source headless, extensible behavior graph engine:

https://github.com/bhouston/behave-graph

```
Hello World

Print out the text "Hello World!" as soon as the graph starts up!

[  
  {  
    "type": "event/start"  
  },  
  {  
    "type": "action/log",  
    "inputs": {  
      "flow": {  
        "links": [ { "node": 0, "socket": "flow" } ] },  
      "text": { "value": "Hello World!" }  
    }  
  }  
]

Console output:

> npm run exec -- ./examples/basics/HelloWorld.json
Hello World!
```
Web-Based Behavior Graph Editor

Build upon React and the behavior-graph library:

https://github.com/beeglebug/behave-flow

Demo here:

https://behave-ui.netlify.app/
Draft glTF Extension

This is being lead by Norbert Nopper of UX3D:

KHR_behavior

https://github.com/KhronosGroup/glTF/pull/2191

The challenge is really which nodes (including events & actions) to support in the first Minimum-Viable-Spec.

Behave-Graph Discord

We have a community growing here:

https://discord.gg/kWSWtVg6