Delivering Interactive Experiences with glTF

September 27 / VIRTUAL
Today’s Presenters

Brent Scannell, Autodesk
Dwight Rodgers, Adobe
Ben Houston, Threetik

© The Khronos Group Inc. 2022 - Page 2

This work is licensed under a Creative Commons Attribution 4.0 International License.
Interactivity

Brent Scannell, Autodesk + Dwight Rodgers, Adobe + Ben Houston, threekit
Interactivity

= 3D Model

Image used under license, stock.adobe.com
Interactivity

= 3D Model + Website
Interactivity

Image used under license, stock.adobe.com

= 3D Model
+ Website
+ JavaScript
Have a nice day!
What if you’re targeting a 3rd party page?

3D content aggregators
What if you’re not targeting a browser?

Mobile and HMD Virtual Worlds
What if you want your content to work everywhere?

Web Viewers  Mobile Viewers  HMD Viewers
Interactivity in the glTF ecosystem

Content + Interactivity
for composing and 3rd party viewers

glTF + glXF
Interactivity in the glTF ecosystem

Content + Interactivity
for composing and 3rd party viewers

- Security
- Performance
- Control
- Time to market

Expressive power
At the Top End

Parity with Top Engines
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. Using 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 let you 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 your project.

- **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.
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.

**Variables** - Set, store, read 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.)
Expressiveness from Node Composability

Behavior graphs use simple atomic nodes which are composed together to create complex behavior.

This requires both (1) flow links and (2) immediate parameter links.

Compound nodes that encode multiple actions/conditions/events/logic into one entity are not desirable, as they are fixed functionality, rather than compose-oriented. Compound nodes limit reusability and flexibility. The only exception would be for critical performance reasons.

This is evident in Unreal Engine’s Blueprints and Unity’s Visual Scripting. There is a push towards a lot of atomic and simple nodes rather than a few compound nodes.
A Layered Standardization Approach

Standarding everything available in Unreal Engine Blueprints or Unity Visual Script would be too much to handle. Instead we propose splitting it into a core specification and then a series of additional augmentations, which do not have to be done in order or that can be combined together:

**Core functionality**
- Basic flow control: if/else, sequence, flip flop.
- Events: lifecycle, custom events.
- Logic: number math, boolean logic, comparisons, simple string manipulation.
- Actions: trigger events, logging.
- Variables: get, set, listen for changes.

**SceneGraph functionality**
- Events: scene node changed, render tick
- Logic: vector3 math, quaternion math, matrix4 math.
- Actions: set scene graph visible, translation, scale, rotation.

**Animation functionality**
- Events: animation completed/cancelled.
- Interpolator completed/cancelled.
- Logic:
- Variables:
- Actions: play animation, cancel animation, mix animation, interpolator.

**Input Method functionality**
- Events: on scene nodes mouse hover, click, drag, touch, finger drag, keyboard press.

**Others?**
- Augmented Reality.
- Audio.
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.
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.
The 0th Layer
At the 0th Layer

**Predictability**
- Data Processing
- Dynamic Control Flow

**Security**
- Allocation
- Iteration
- Data access
- Web access
At all levels?

- Common framework for describing the graph

```
Node

- named input 1
- named input 2
  ...

- named output 1
- named output 2
  ...
```

connections
No Data Processing

Trigger and action nodes with only “exec” outputs
+ Constants

No interaction without user action
Small number of actions to keep it:

- Easy to implement
- Easy to secure
- Easy to optimize
- Easy to review
- Quick to Approve and Adopt
What can be made with the 0th layer?

Demo
More than just an Onion

A sprouted potato?
Calls to Action

• Call for use cases: please comment on these github issue (TBD?):
  - https://github.com/KhronosGroup/glXF
  - https://github.com/KhronosGroup/glXF/blob/main/specification/2.0/README.md

• Open source reference implementation for behavior graph:
  - https://github.com/bhouston/behave-graph

• Slack group for discussing interactivity:
  - https://gltfworkspace.slack.com/archives/C044GF9MUKB

• Call for concerns on security model
• Feedback on existing node-graph systems?
Ask the Experts

Use the Zoom Q&A feature to ask your questions
More Information

- glTF: https://www.khronos.org/gltf/
- glTF Experience Format (glXF): https://github.com/KhronosGroup/glXF
- glTF 2.0 Final specification: https://github.com/KhronosGroup/glTF
- glTF Registry: https://registry.khronos.org/gltf/

A recording + slides from today can be found at:
https://www.khronos.org/events/delivering-interactive-experiences-with-gltf
Get Involved!

- Slack  https://khr.io/gltfslack
- Twitter  @glTF3D
- Reddit  https://www.reddit.com/r/gltf
- Github  https://github.com/khronosgroup [search for gltf]
- Stackoverflow  https://stackoverflow.com/questions/tagged/gltf
- Forums  https://community.khronos.org
Delivering Interactive Experiences with glTF

Thank you!