University College London

OpenMPD Unity Node Tool: Tutorial

Multi-Sensory Devices lab

Robert Cobden
22-Mar-23

Change Log

Date	Change
22/03/2023	Initial version
03/05/2023	Review, minor corrections, and release (Diego Martinez Plasencia)

Table of Contents

1 About this document:

We have developed a graphical tool for describing OpenMPD content within Unity. This document will introduce you to the tool and walk you through specific examples. These examples are designed to provide you with a good foundation of knowledge with which you can start creating your own content. This tool is still a work-in-progress and some features are still being developed. Currently this tool can describe primitive positions, however tools for amplitudes and colours are still in development.

1.1 Nomenclature

The content of this document assumes that you have an understanding of the terminology used within the OpenMPD ecosystem. Namely the following terms:

- Primitive

  Within the OpenMPD framework the term **primitive** is used to describe a 'point' in 3D space which has an associated position, amplitude, and colour at any given point in time. The most common use of this concept with particle-based displays (PBDs) is to create an acoustic trap which can carry polystyrene beads. Varying the position of this trap with time allows for animated 3D content. Varying the amplitude of the primitive with time allows for the emission of audible sound waves. Varying the colour with time allows for colouring different parts of a volumetric/PoV path.
- Descriptor

  A **descriptor** refers to the encapsulation of a sequence of either: positions, amplitudes, or colours. Primitive properties are driven by their attached descriptors and the display of descriptor content can be synchronised, such as combining light projection and primitive / levitated bead positions. A simple example would be to have the bead travel on a circle path which has three different projected colours, each for a third of the path length.

  Primitives always have a descriptor attached to them even if none were specified, e.g. the default position descriptor contains one position: (0, 0, 0).
- Trap

  A point of consistent low-pressure created through the combination of sound waves from several transducers, referred to as a **trap** due to its ability to capture lightweight objects and hold them in a fixed position.

The following abbreviations are used in this document:

- ASG – Animation Scene Graph
- GOR – GameObject Reference
- PS – Primitive State node
- SC – State Collection node

2 Installation / Setup:

The latest version of the code for the node design tool can be found on the "node-tool" branch of the OpenMPD GitHub repository. After cloning this branch, follow the typical setup procedure for OpenMPD with Unity.

If you want to use the node design tool with a newer version of the OpenMPD Unity integration, firstly try copying the node design folder (OpenMPD Client/Assets/Node design) from the above git branch into the latest version of the OpenMPD Client from the main branch. If you encounter integration problems, please raise an issue on GitHub or contact one of the developers.

3 Node-Design Overview

This tool provides a graphical interface for describing OpenMPD content. The graphical interface consists of a graph of **nodes** (which represent encapsulated information or behaviours), and **connections** between those nodes which describe the flow of information. An example of the editor can be seen in Figure 1Figure 1. This figure shows three nodes, two nodes with only outputs (Float node) connected to a node with both inputs and outputs (Vector 3 Node). It can be seen from the figure that one of the inputs of the Vector3 node is not connected to the output of any other node, if possible, the interface displays an input box so this value can still be set.

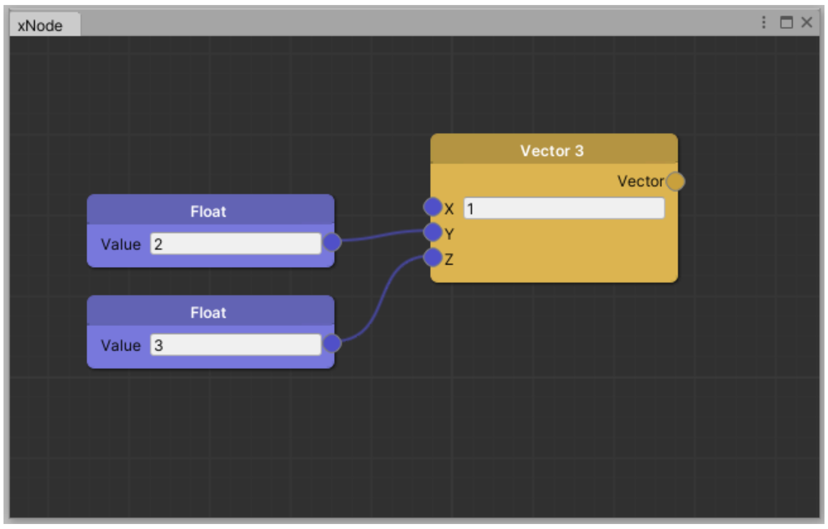
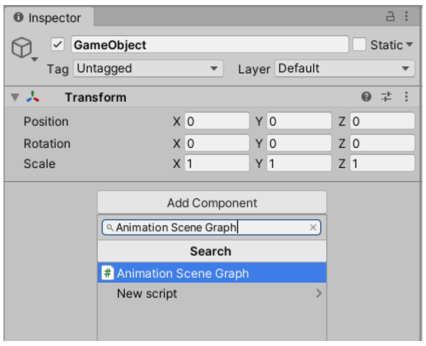


Figure 1: A screen capture of the node design interface showing two float nodes providing values to the Y and Z inputs of a Vector3 node.

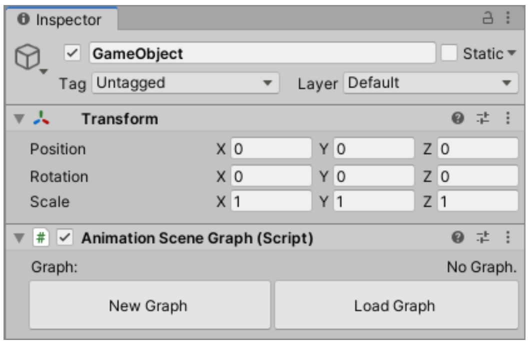
Outputs can be connected to inputs via clicking and dragging from the output circle to an input circle of the same type. Right clicking while dragging will add a graphical pathing node for the current connection. The inputs and outputs are coloured according to the type of the underlying data, Figure 1 shows the float inputs and outputs are the same colour, and different to the vector output.

3.1 Adding a Graph to a Scene

To add a node design graph to a scene, firstly attach an **Animation Scene Graph (ASG)** script to an object in the scene as shown in Figure 2A.



A

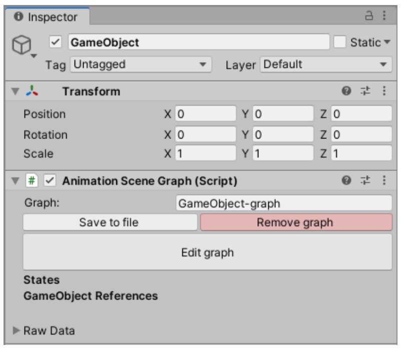


B

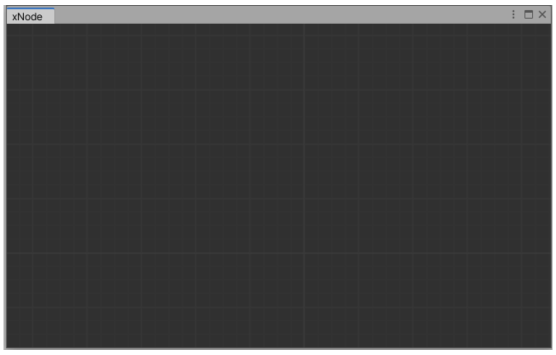
Figure 2: A) Adding an Animation Scene Graph to a Unity GameObject. B) The Animation Scene Graph is initialised without a graph.

Once the script attached to a game object, a graph can be added by either creating a new graph or loading a graph as shown in Figure 2B. Clicking **New Graph** will intialise a blank graph within the ASG.

A blank graph contains no nodes, the editor showing this can be seen in Figure 3B. In the editor nodes can be added from the available list via the right-click context menu.



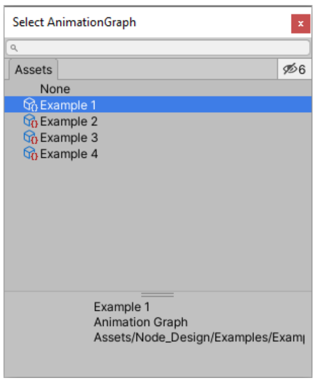
A



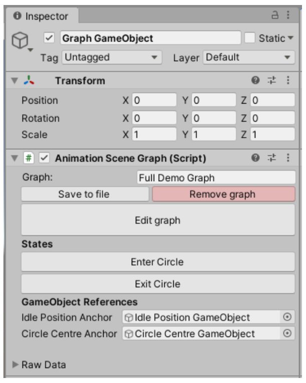
B

Figure 3: A) The Animation Scene Graph after the ***New Graph*** *button is pressed. B) The editor window for an empty graph.*

Alternatively a graph can be loaded from a Unity asset file, the load graph menu and the resulting loaded graph are shown in Figure 4A and Figure 4B respectively.



A



B

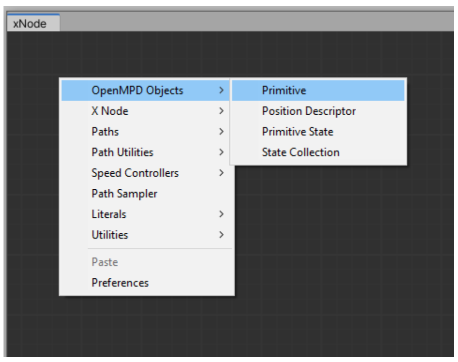
Figure 4: A) The load graph menu, showing all saved AnimationGraph assets, double clicking an option will load that graph. B) The ASG once a graph with nodes has been loaded, this graph contains states and game object references.

4 Examples

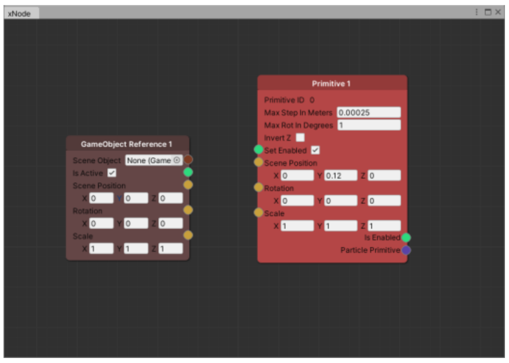
4.1 Example 1: Primitives, Transform Manipulation, and Game Objects

This first example will introduce you to the notion of a primitive, teach you how to manipulate that primitive, and how to tie the primitive to a game object. This example assumes you have loaded the "Example 1" scene from the Node_Design/Examples folder.

The graph we will use in this example is attached to the game object titled "Animation Graph Object" and is currently empty. We will begin by adding a Primitive node and a Game Object Reference to this graph, do this by right clicking and going to *OpenMPD Objects -> Primitive* and then to *Utilities -> Game Object Reference*. This is shown in Figure 5A. You can also load the Example 1 graph asset, the graph should look like Figure 5B.



A



B

Figure 5: A) The context menu accessible through right clicking in the graph editor. B) A graph showing a GameObject Reference node and a Primitive node, with no connections between them. This is the graph at the start of Example 1.

Looking at the primitive node, we can see it has an enable input, and inputs that can be used to drive its transformation matrix. Set the Y value for scene position to 0.12 (the centre of the levitation area).

Exercise: Press play to display the current content, the rendered scene should show a white dot in the centre of the levitation area, and if a device is connected then a trap should be created there. Try adjusting the values for scene position within the graph, the rendered primitive and trap should move towards the newly specified position. The primitive can also be enabled/disabled by modifying the Set Enabled input.

In this scene we have not specified any descriptors for the created primitive and so it is using the default descriptors for each category, i.e. position of (0,0,0) and constant maximum amplitude etc. Every primitive has a transformation matrix which defines the mapping from scene origin to primitive origin, and that is what we are adjusting in this exercise. This causes the primitive's (0,0,0) position to move relative to the scene, and it moves smoothly with interpolation due to the primitive's Max Step in Metres parameter.

It can be very useful (e.g. for combining unity content with levitated particles or dynamic content description) to refer to the current position of a Unity game object. This is what the **Game Object Reference (GOR)** node is for. If you set the Scene Object of the GOR to an object in the scene, the real-time position rotation and scale of the object can be accessed within the graph environment. This can be done through the node, via drag and drop into the graph editor, or through the ASG fields (see Figure 4B).
Attach the GOR to the object in the scene named "Anchor Object" and then connect the 'is active', position, rotation, and scale outputs to the primitive. The graph should look like Figure 6. This setup is equivalent to the ParticlePrimitive prefabs from Example 1 of the regular OpenMPD Unity tutorial.

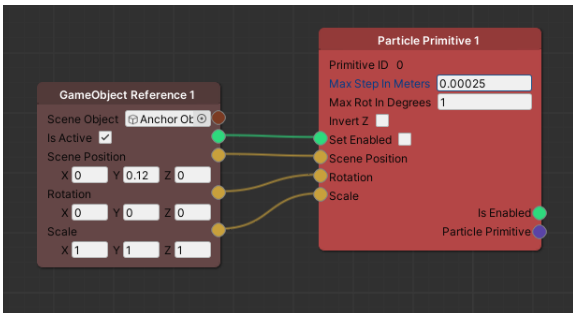


Figure 6: A Particle Primitive node, with its Scene Position, Rotation, and Scale derived from a GameObject in the scene named "Anchor Object". These values are updated as part of the unity update cycle.

Exercise: Press play to display the current content, move the game object within the scene and the rendered primitive / trap should somewhat follow the movements of the game object. There is a simple script attached to this game object to allow you to move it in 3 dimensions by pressing WASD or the arrow keys in combination with space and ctrl.

Additional Tasks: In play mode, try:

- Enable the sphere child attached to the anchor object and observe how the rendered particle moves slower than the anchor object while moving (this is to smooth trap movement and prevent dropping the trapped bead). Adjust the primitive's value for Max Step in Meters and observe the change in behaviour.
- Changing the value of the primitive's, rotation, or set enabled input – this can also be done by connecting it to the anchor node state through the GOR node. Changing the scale should only affect the content displayed in later examples.
- Create a second primitive, control them independently, create and attach the second primitive to a new game object in the scene. Give both anchors a shared parent and move them simultaneously.

4.2 Example 2: States, Descriptors, and State Collections

This example introduces the concept of primitive states, descriptor nodes and state collections. Please load the scene Example 2 from *Node Design/Examples*. This example is designed to convey three important concepts about primitive states:

1) The latest state of a primitive will loop endlessly unless a new state is applied.
2) The content of a looped state should be cyclical for particle-based content.
3) If multiple states are enqueued at once they will each run once until the final state.

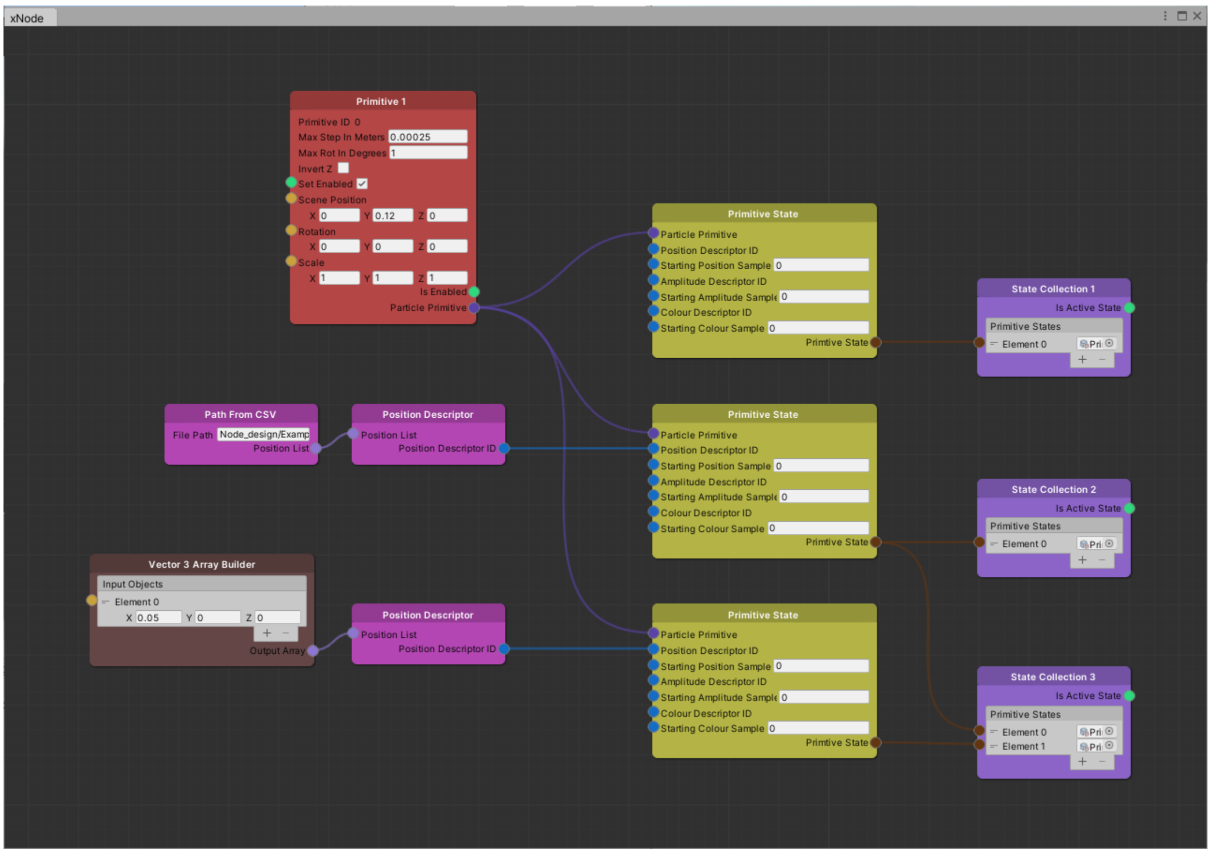


Figure 7: The graph for the second example, showing three different primitive states, and how state collections can be used to transition between or to combine sequences of states.

It was previously described that primitives always have at least a default set of descriptors applied to them, the combination of a set of descriptors with a primitive is referred to as a **Primitive State (PS)**. In the graph for this example (see Figure 7) we can see one primitive node, three numbered primitive state nodes, three numbered state collection nodes, two position descriptor nodes, and two Vector 3 Array Builder nodes.

- The node Primitive State 1 describes a state in which no position, amplitude, or colour information is provided. In other words, this is the default state for a primitive.
- The node Primitive State 2 describes a state in which a sequence of positions are specified, this is done through the use of a Path From CSV node (to load a list of positions), and a position descriptor node (to register those positions within OpenMPD and encapsulate them with a descriptor ID).
- Using a Vector3 Array builder, the node Primitive State 3 describes a state in which only a single position is provided.

State Collection (SC) nodes are used to manage the transitions between primitive states. **When a primitive enters a state, it remains in that state until a new state is provided**. A newly created primitive stays in a state using the default descriptors until instructed otherwise.

- The State Collection 1 node contains only the Primitive State 1 node and so can be used to return the particle to its default state.
- State Collection 2 node contains only the Primitive State 2 node. When this state is entered, the primitive will iterate through the described positions from first to last, and then since there are no additional states, the content will repeat. **While a primitive is in a state, the content loops repeatedly.**
- State Collection 3 node contains both the Primitive State 2 node, and the Primitive State 3 node.

EXERCISE: Run the scene to display the content. Select the Game Object with the graph attached and from the inspector click the buttons to transition between the state collections. Observe that for:

- State Collection 1 returns the primitive to the centre and stays there.
- State Collection 2 the sequence of points repeats endlessly.
- State Collection 3 the sequence of points occurs once, and then the primitive stays in the final position.

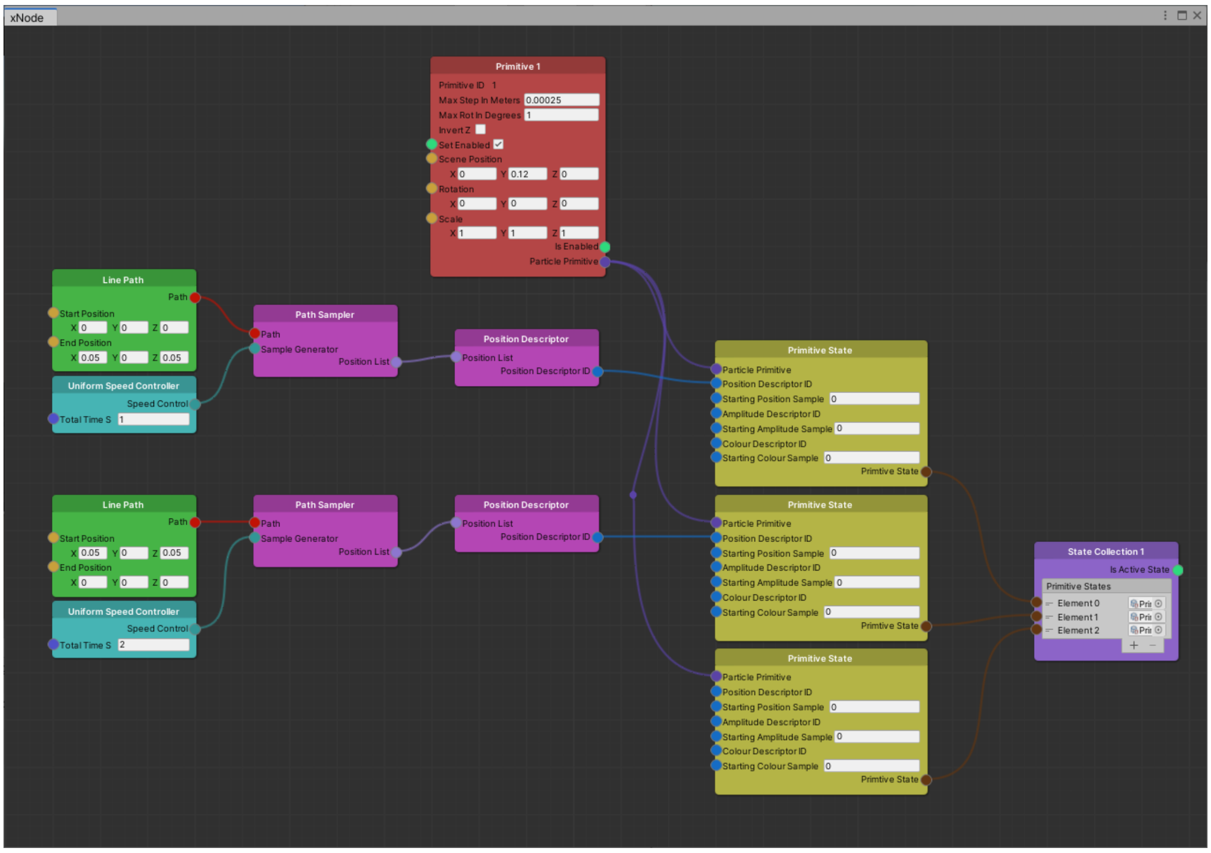
When using a primitive to levitate a particle, it is important for the resulting trap to take a continuous path without sudden changes in location. In this example, State Collection 2 would result in the particle being dropped because the content of the state is looping, instantaneously moving the trap location far away from the particle. **Content should be designed in a way that the final Primitive State of each State Collection is a cyclically continuous one** (e.g. a circle, or stationary point). State Collection 3 provides an example of this.
In a scenario where multiple primitives are used simultaneously; this should be done on a per primitive basis for each modified primitive.

ADDITIONAL TASKS: In play mode, try:

- Modifying the Vector3 position used in Primitive State 3, and then entering State Collection 3
- Change the scene position, scale, or rotation properties of the primitive and see how they interact with the content of each state
- Try connecting the On Enter outputs from the individual State Collections to the primitive to enable or disable the primitive during certain states.

4.3 Example 3: Generating Paths

The previous examples have introduced concepts that were also available within the standalone Unity integration of OpenMPD. The benefits of this node-based tool become apparent when describing animated content with increased complexity. In this example we will explore different ways of describing lists of positions (or paths) for a primitive to follow.


xNode

Primitive 1
Primitive ID 1
Max Step In Meters 0.00025
Max Rot In Degrees 1
Invert Z ☐
Set Enabled ✓
Scene Position X 0 Y 0.12 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Is Enabled
Particle Primitive

Line Path
Path
Start Position X 0 Y 0 Z 0
End Position X 0.05 Y 0 Z 0.05

Uniform Speed Controller
Speed Control
Total Time S 1

Path Sampler
Path
Sample Generator
Position List

Position Descriptor
Position List
Position Descriptor ID

Line Path
Path
Start Position X 0.05 Y 0 Z 0.05
End Position X 0 Y 0 Z 0

Uniform Speed Controller
Speed Control
Total Time S 2

Path Sampler
Path
Sample Generator
Position List

Position Descriptor
Position List
Position Descriptor ID

Primitive State
Particle Primitive
Position Descriptor ID
Starting Position Sample 0
Amplitude Descriptor ID
Starting Amplitude Sample 0
Colour Descriptor ID
Starting Colour Sample 0
Primtive State

Primitive State
Particle Primitive
Position Descriptor ID
Starting Position Sample 0
Amplitude Descriptor ID
Starting Amplitude Sample 0
Colour Descriptor ID
Starting Colour Sample 0
Primtive State

Primitive State
Particle Primitive
Position Descriptor ID
Starting Position Sample 0
Amplitude Descriptor ID
Starting Amplitude Sample 0
Colour Descriptor ID
Starting Colour Sample 0
Primtive State

State Collection 1
Is Active State
Primitive States
Element 0 Pri
Element 1 Pri
Element 2 Pri

Figure 8: A graph showing how paths and speed controllers can be used to describe animated content.

When we provide OpenMPD with a primitive state that contains a position descriptor, OpenMPD will iterate through each of the positions within that descriptor, moving the primitive / trap from position 1 to position 2 to position N. We have seen an example of this in example 2 where we manually specified the positions in a Vector3 Array Builder, or by loading a sequence of positions from a csv file.

Smooth movement is achieved by having sequential positions be close enough to each other that the particle moves but also remains trapped. The size of the spaces between positions and the rate at which positions are presented determine the speed of the primitive. OpenMPD updates the state of primitives at a much faster rate than unity's update loop – typically at a rate of 10kHz or **10,000 positions per second** (this also applies to amplitudes). This rate is adjustable and is defined by the value chosen for: OpenMPD Presentation Manager -> Pipeline Control Parameters -> Divider. The rest of this document assumes a 10kHz update rate is chosen.

With this knowledge, if we specify 10,000 positions between point A and point B we know that it will take the primitive 1 second to complete that journey. That is a lot of positions to describe and far too many to be done manually, therefore this tool provides a solution: a method of describing and sampling paths. This process is done using three types of nodes:

1) **A Path node -** A path is used to describe a continuum of positions in 3D space between a start point (at 0%) and an end point (at 100%), these are generally based off of mathematical functions (e.g. line, circle, ellipse, etc.) and are separated from the concept of time.
2) **A Speed Controller node -** A speed controller determines how a path is converted into discrete positions, it is used to control the spacing between sampled positions and the total number of positions. This controls the speed (and acceleration) of the primitive across the length of the path and thus the total duration taken to complete the path.
3) **A Path Sampler node -** This node combines the speed controller with the path node and outputs a list of positions to be registered as part of an OpenMPD descriptor.

Describing paths in a parametric way allows for separation between the shape of the path and the resulting primitive positions, making content easier to design and modify. This is somewhat equivalent to storing images in a vector format, the images/paths can be viewed/sampled at any desired resolution using the same description data.

Figure 9 shows an example of this, a Line Path node is being used to generate points between (0,0,0) and (0.05, 0, 0.05), and this path is being sampled with uniform spacing between samples using a Uniform Speed Controller. The total number of positions generated (10000) will result in the travel taking exactly 1 second.

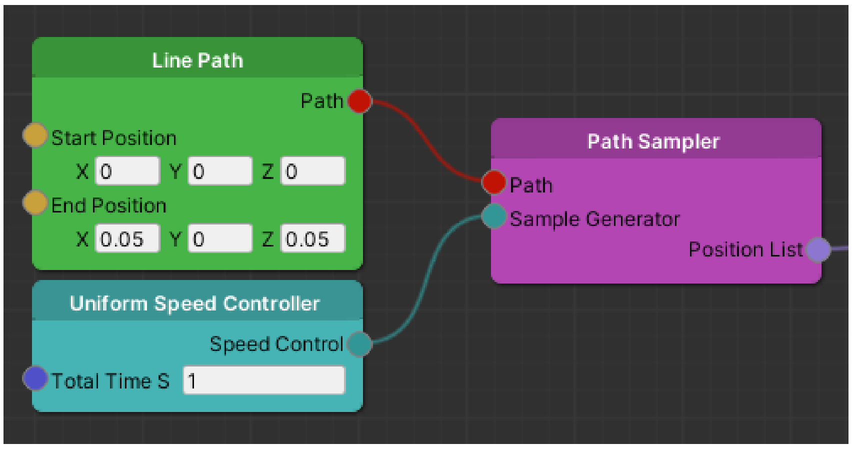


Figure 9: The nodes required to generate a list of positions which move a primitive along a straight line. The speed will be uniform across the length of the path and due to the number of positions generated the total time taken will be 1 second.

EXERCISE: Run the scene to display the content, trap a bead and enter the state. The bead should move to the specified point in 1 second and take 2 seconds to return. Try modifying the time values.

Moving a particle at high speed can take advantage of the human eye's persistence of vision (POV) to create what looks like a solid line or shape rather than a fast-moving bead. To facilitate the fastest cyclical shapes it can be necessary to first accelerate the particle from stationary. With this tool it is incredibly easy to do so through the use of an easing speed controller. We previously established that the speed of a particle is described by the distance between sequential positions, an easing speed controller can be used to describe non-uniform spacing, and thus changes in speed along a path. The resulting speed is determined by the chosen easing curve. Some of the templates within this tool describe, acceleration, deceleration, accelerating and then decelerating (shown in Figure 10) or the curve can be modified to describe an entirely custom user-defined curve.

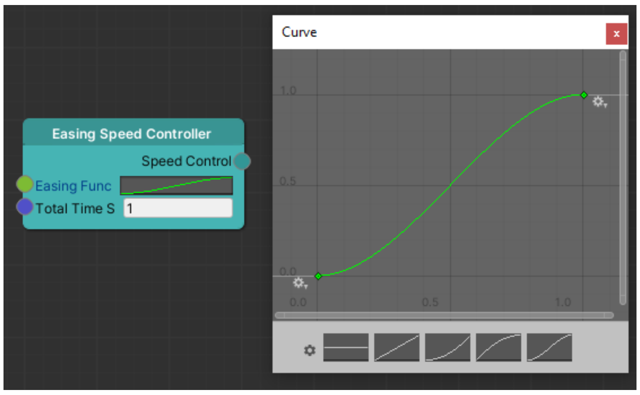


Figure 10: An easing speed controller can be used to vary the speed of the primitive while travelling along a path. Clicking on the small graph in the node next to "Easing Func" opens a dialogue window where either various templates can be chosen, or the graph can be modified directly to create custom variations in speed. The speed of the primitive is determined by the length of the path divided by the total time, and then multiplied by the gradient of the easing function. The templates (from left to right) are: 1) No speed 2) Constant speed 3) Accelerate in 4) Decelerate out 5) Accelerate in and Decelerate out.

EXERCISE: Swap the uniform speed controller(s) for an easing speed controller. Run the scene and explore the effect of the different easing curves (click on the small graph within the node to modify). Use the curve editor to create a custom curve of your own (double click on the green line to make a custom control point). What's the shortest time needed to travel the path without dropping the bead when using the template curves for:

- No acceleration or deceleration
- Acceleration only
- Acceleration and deceleration

CHALLENGE: Can you create the forward and backward motion of the original example using only a single path node, a single easing speed controller, and one less primitive state? Try modifying the Y coordinate of the final point of the easing curve.

The example explored above made use of a straight-line path between two points. Multiple paths can be combined in sequence to create complex paths such as shapes (e.g. for POV content). This tool provides the following fundamental path elements:

- Fixed Point
- Line Path
- Circle Path
- Oval Path

This is not a definitive list and other path nodes may be added in future development. Paths can either be combined through sequential animations or through the use of the **Path Merger** utility node as shown in Figure 11. This node either A) scales and combines two full paths so that the path transition occurs at the defined percentage value, or B) crops and splices together two paths, transitioning from the first to the second at the defined percentage point.

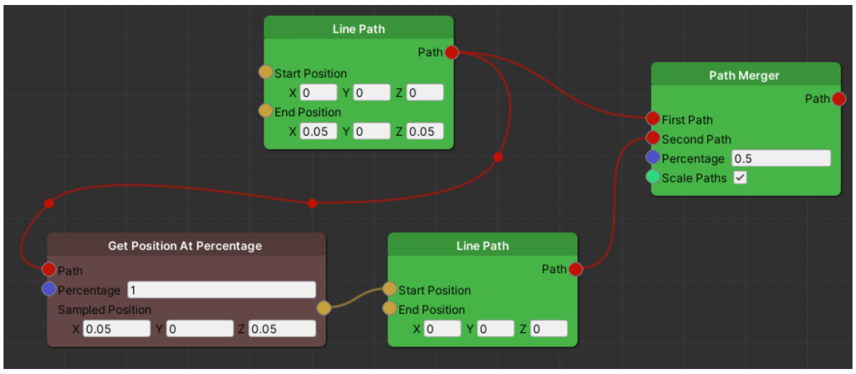


Figure 11: Properties of a path can be used as parameters for other paths, and paths can be combined. Here the first line path is being sampled to provide the start position of the second line path. Later the two paths are combined and scaled in equal proportion.

A useful principal when designing node-based content is that if two inputs refer to the same value, make a shared node which feeds its output to both of them. This way, when changes are made to the graph in the future, it is obvious which values must be kept in synchronicity. The graph for this example is a good demonstrator of that – the end position of the first line path, and the start position of the second line path should always be the same, but this is currently hardcoded in the input field. Using a Vector3 node to connect to both these inputs is one solution, but for more complex paths (e.g. an arc or ellipse) another solution is available: the path utility node named "Get Position At Percentage" as shown in Figure 11. This node samples the connected path at the given percentage and returns the sampled position. Here it is used to feed the end position of the first line, into the start position of the second line.

ADDITIONAL TASKS: Using play mode to test your creations:

- Use Circle Path Nodes and Path Merger Nodes to describe a path that follows a figure of 8 at a steady speed. Tip: Break the path down into arc-sections, describe each using the circle path nodes, and then merge them.
- Create a path that forms a closed semicircle (D-shape), configure a speed controller to reduce speed during the transitions from the arc to the straight line and vice versa.

4.4 Example 4: Complex State Machines

In this final example for this tutorial, we discuss the higher-level use of state collections and how content may be designed in a state-machine-like structure. This is not the only organisational method for content and is meant only to provide guidance for a common use-case.

A recent exhibition of our work was presentation of a gimbal-mounted levitation setup at *New Scientist Live* (London, Oct 2022). The levitator was configured to display and transition between various single-bead POV content while a gimbal structure was either rotating or stationary. For this we used a simple state-based approach as shown in Figure 12.

At start-up and between content, the bead was at rest in the centre of the levitated area. For each piece of POV content we had a transitional state to move the bead from the idle position to the start of the POV path. Once this movement was complete (after a set time elapsed) we entered the POV state, which typically consisted of an initial slower/accelerating path, followed by the full speed cyclical path. The bead then loops the cyclical POV path until it receives further content. A third state was used to transition out of the POV with a decelerating path, and then transition to the idle point and wait there. Having a common idle position made it possible to transition reliably between content.

A strong motivator behind this approach was that the OpenMPD Unity integration did not provide a feedback mechanism for when a primitive state was finished. Thus, to synchronise gimbal rotation with POV content they had to be triggered simultaneously, and this was achieved with a timed delay after triggering the transition to POV start position.

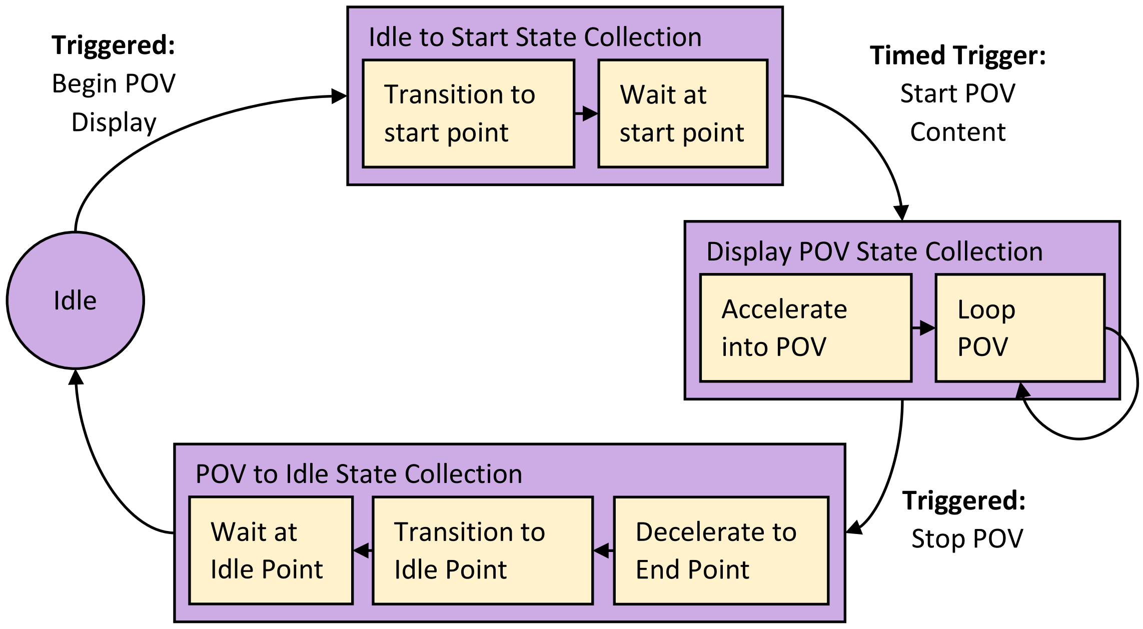


Figure 12 A diagram representing the state collections and primitive states used in a previous POV demonstration.

The node graph for this example (shown in Figure 13) follows a similar approach to the above description but (as there is no gimbal synchronisation) it combines the first two state collections – transitioning to the start point and immediately beginning the POV content.

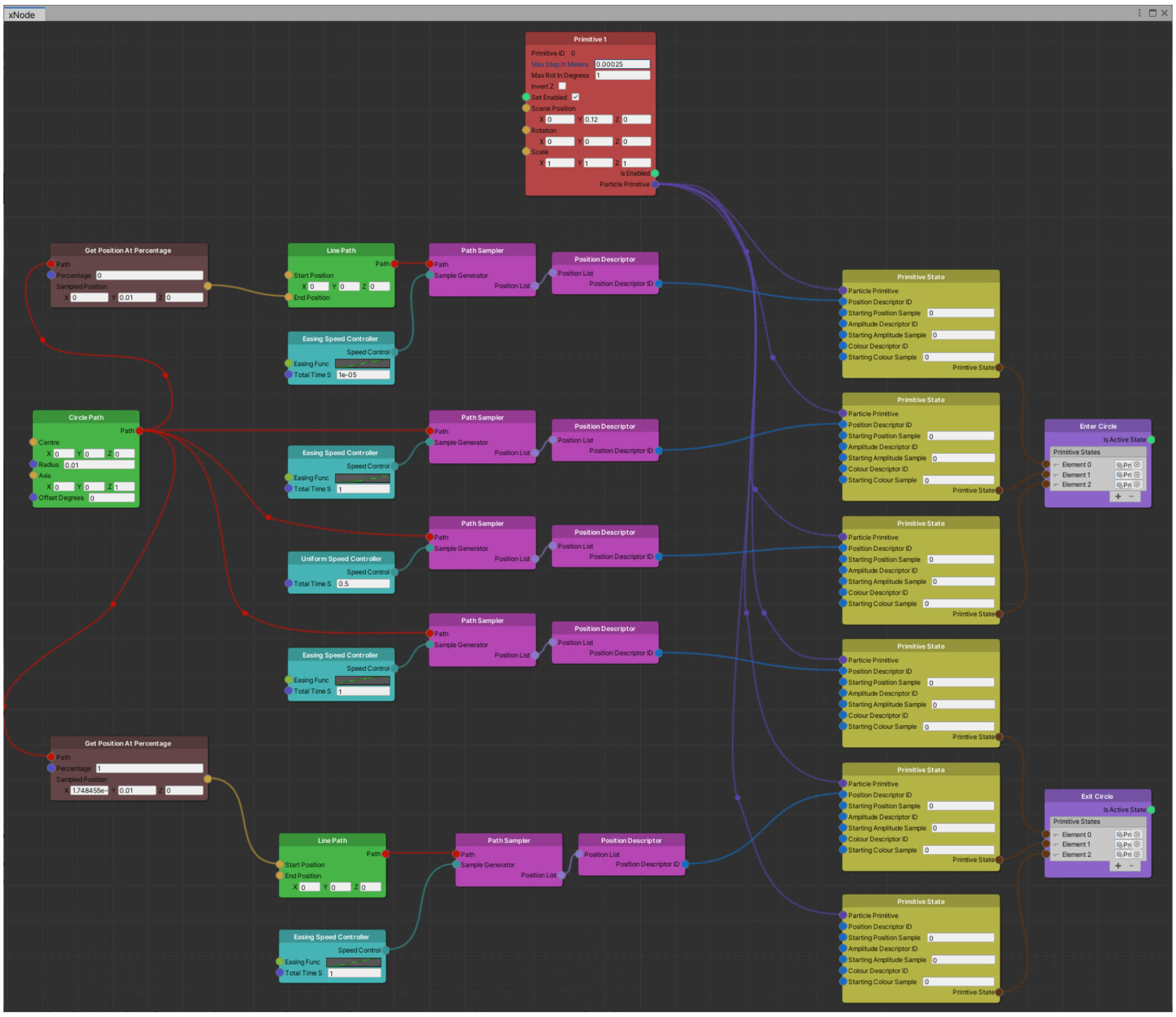


Figure 13 A node graph in which the first state transitions from an idle position to draw a POV circle, and the second state decelerates from the POV circle and then returns to be static at the idle position.

You now have the skills needed to use this tool to its potential: use what you have learnt to create your own content! You can share your creations with us and give us feedback or improvement ideas on the OpenMPD GitHub!

5 Tips and Tricks

5.1 Node Renaming

Right clicking on nodes allows you to set the displayed name of that node. This is particularly useful for setting meaningful names for State Collections and Game Object References.

5.2 Saving Runtime Changes

When changes are made to a unity scene during play mode they are reset when the scene is stopped. If you want to save the graph that you developed during play mode you will need to use the save graph feature to store your changes before stopping the scene. The graph can then be loaded from file when not in play mode.

5.3 Connection Routing Points

When dragging a connection line, right clicking can be used to add a routing point for that connection. This is especially useful for organising long or overlapping connections or in graphs with many nodes and connections. This feature is used extensively in Figure 13.

5.4 Customising Graph Colours

In the preferences panel of the graph editor, many graph settings can be customised, including the colours for each of the input/output connection types.

5.5 Create Your Own Nodes!

If you see a feature that is missing, get your coding hands dirty and create your own nodes! Take a look at the Vector3 Node as a simple example.
Some ideas for potentially useful nodes:

- Math nodes for floats, vectors, and Booleans
    - E.g. Vector3 sum would allow a fixed offset from a GOR node's position
    - Boolean math / logic would allow for primitives to only be enabled during some states.
- Other utility nodes
    - A means to merge vector lists (thus enabling the merging of sampled paths)
    - A node with a button for writing vector3 lists to csv file
- Modify current GOR into Read GameObject node, and then create a write GameObject node
    - Allows for graph content to drive the properties of a gameobject
    - Could create new nodes which have onUpdate behaviour, e.g. iterate through position list, allowing it to use path sampler output.
- Additional Path and Speed modification nodes
    - Split path merger functionality into weighted path adder node and path subsection nodes.
    - Path smoothing node? For each sampled point, take the average position of N samples across ±X%.
    - A means of combining different speed controllers
- Global reference / variable nodes to store and retrieve values
    - Could be used to make graphs tidier or even link between separate graphs
- Nodes for describing amplitude or colour and generating descriptors would allow for synchronised multisensory content.