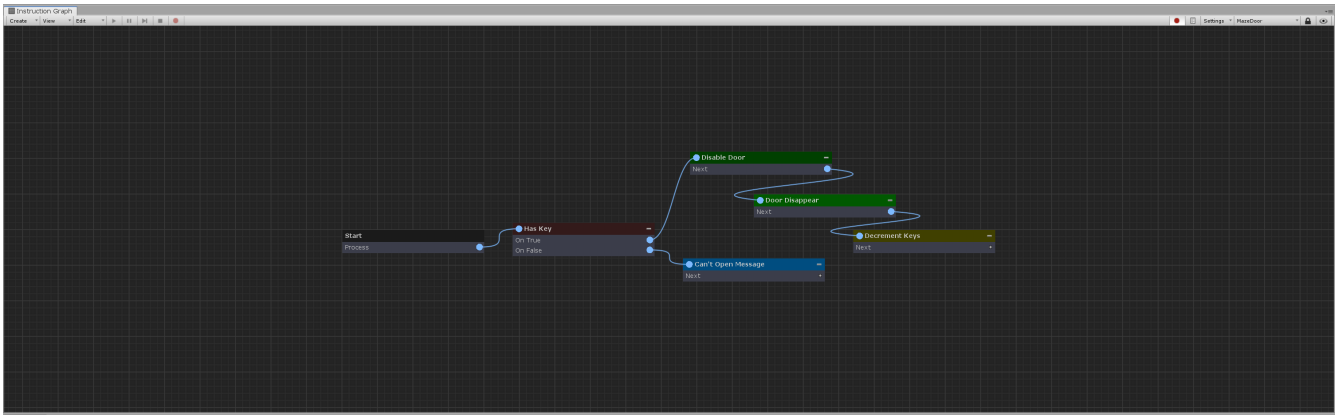
Unity Composition Graphs

PiRho Soft

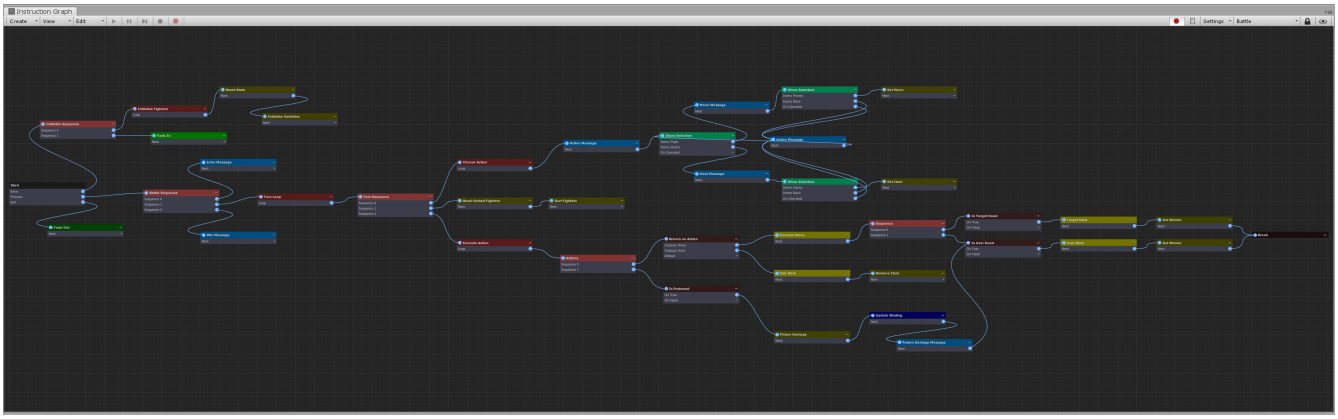
Overview . . . 1
Workflow . . . 1
Topics . . . 2
Nodes . . . 3
Control Flow . . . 5
Branching . . . 5
Sequences . . . 5
Loops . . . 6
Instruction Store . . . 7
Context/Inputs/Outputs . . . 7
Debugging . . . 9
Running Graphs From Script . . . 11
Creating Custom Graphs . . . 12
Creating Custom Nodes . . . 13

Overview

Intruction Graphs are the main feature of the composition system. Graphs work similarly to flow chart. When run, they traverse a tree of nodes which each execute their own logic. Graphs can branch and loop based on variables, expressions, input, or any other custom logic. The power of graphs comes in the way they allowing timing and sequencing of game logic to occur. Whereas waiting on actions and events in script can be difficult to manage, graphs provide a simple to use visual interface for interacting with your game world to create things like, cutscenes, dialog trees, menus, scene transitions and even entire game systems, without the need to write lengthy scripts. The benefit of graphs is that they can work dynamically based on variables in your game world (see Variables), with each graph able to perform different actions based on the object that is executing the graph and the inputs provided to it. Robust debugging is possible with breakpoints, runtime variable editing, and extensive error logging. In addition to supporting most of Unity's built in systems, graphs are also easily extensible to your own game logic by simply creating custom nodes which can then provide any desired functionality.



An example of a simple graph that opens a door with a key



An example of a more complex graph that runs an entire turn based battle system

Workflow

Create a new graph in the Project window (**Create › PiRho Soft › Graphs**) Open the Instruction Graph Window by double clicking on a graph in the Project window or through the **Window › PiRho Soft › Instruction Graph** menu. Every graph starts with a single *"Start"* node where execution of the graph begins. Selecting the start node will show the graph itself in the inspector

where its properties can be edited (see Context/Inputs/Outputs). Create new nodes from the drop down in the top left of the window, by right clicking anywhere on the graph, or by pressing space. When a node is selected its properties can be edited in the Inspector. In the graph view, each node will show its list of outputs that it can branch to. Create connections from these branches by clicking on the output knob and then the node you would like it to connect to. Many different connections can connect to the same node however, outputs can only connect to a single new node. When a graph reaches an output to a branch that has no connection the graph will end. Keep in mind, flow off a graph is unidirectional and can is visualized with inputs on the left and outputs on the right.

Topics

1. Nodes
2. Control Flow
3. Instruction Store
4. Debugging
5. Running Graphs from Script
6. Creating Custom Graphs
7. Creating Custom Nodes

Nodes

Nodes make up the bulk of the functionality of a graph. They may control both the flow of the graph and/or the behaviour that should be performed. For example, a Conditional Node branches the graph based on the `true` or `false` value of an expression and has no effect on the game world, while a Message Node physically displays a message in the game. The following is a list of built-in nodes by category:

Animation	
Play Animation	Play an animation on a game object
Play Animation State	Play an animation state on an Animator
Set Animation Parameter	Set an animation parameter on an Animator
Play Effect	Create and play an effect Prefab
Play Sound	Play an AudioClip
Play Timeline	Run a timeline

Composition	
Expression	Run an expression
Instruction	Run an instruction
Shuffle	Shuffle a variable list
Sort	Sort a variable list
Reset Tag	Reset variables by tag
Reset Variables	Reset variables by name

Control Flow	
Conditional	Fork the graph based on a condition
Branch	Run a single branch based on a string value
Sequence	Run a list of nodes in order
Loop	Loop until a condition is met
Iterate	Iterate the items in a variable list
Break	Break out of a loop
Yield	Yield for a single frame
Yield	Break out of all loops

Debug	
Comment	Keep notes in the graph
Log	Log a message in the console
Mockup	For visual prototyping

Interface	
Show Message	Show a message
Show Selection	Show and wait for a selection
Input	Wait until a button is pressed
Show Control	Activate an interface control
Hide Control	Hide an interface control
Set Binding	Set the binding value for a binding root
Update Binding	Tell a binding root to update its bindings

Object Manipulation	
Create Game Object	Create a game object from a prefab
Create Scriptable Object	Create a scriptable object
Destroy Object	Destroy an object
Enable Object	
Disable Object	

Sequencing	
Play Transition	Play a transition
Stop Transition	Stop the current transition
Load Scene	Play a transition
Unload Scene	Play a transition
Transform Object	Move or animate the transform of a game object
Time Scale	Set the time scale value
Wait	Wait for an amount of time

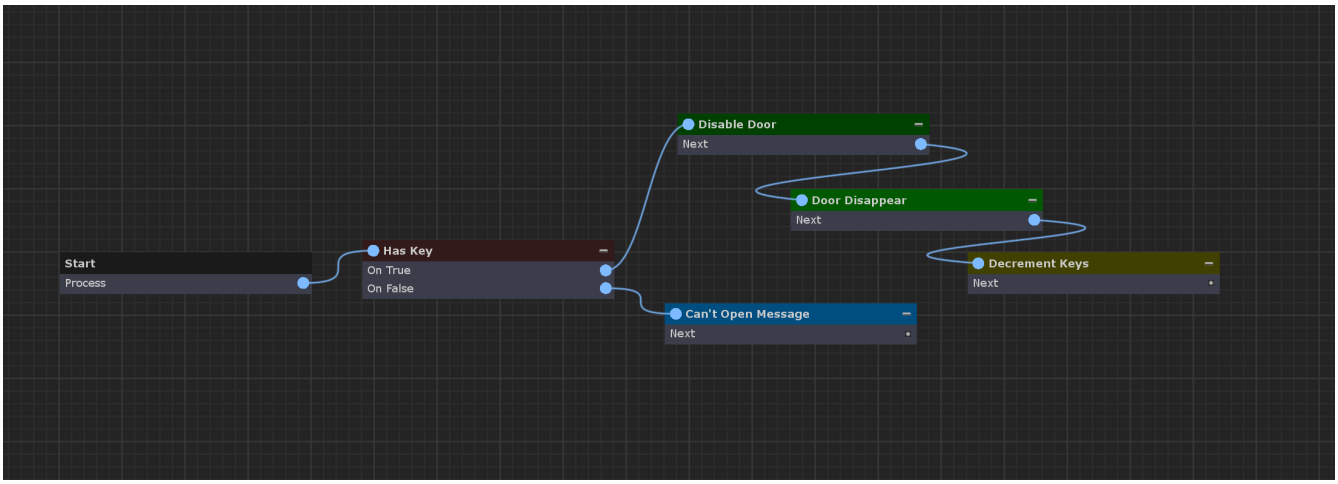
In addition, custom nodes can be created (see Creating Custom Nodes).

Control Flow

While most nodes simply perform an action and then move on to another node, some nodes alter the control flow of the graph, branching to different outputs, looping through its child nodes multiple times, or running a list of nodes in order.

Branching

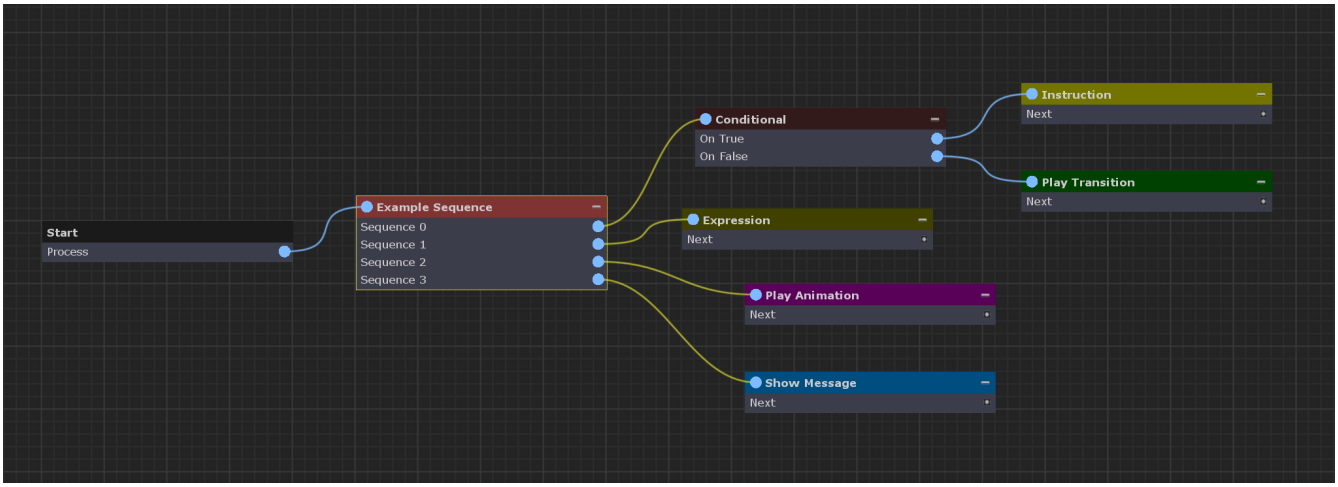
Nodes that branch can have multiple output connections but will only traverse through a single connection. Through branching, graphs can have dynamic behaviour in different executions of a graph.



In the example above, imagine we run a graph for when player is attempting to open a locked door. If the player has the key we can branch to nodes that run the open animation, load the next scene, etc. If the player doesn't have the key we can branch to a different set of nodes that displays a message that says the door is locked.

Sequences

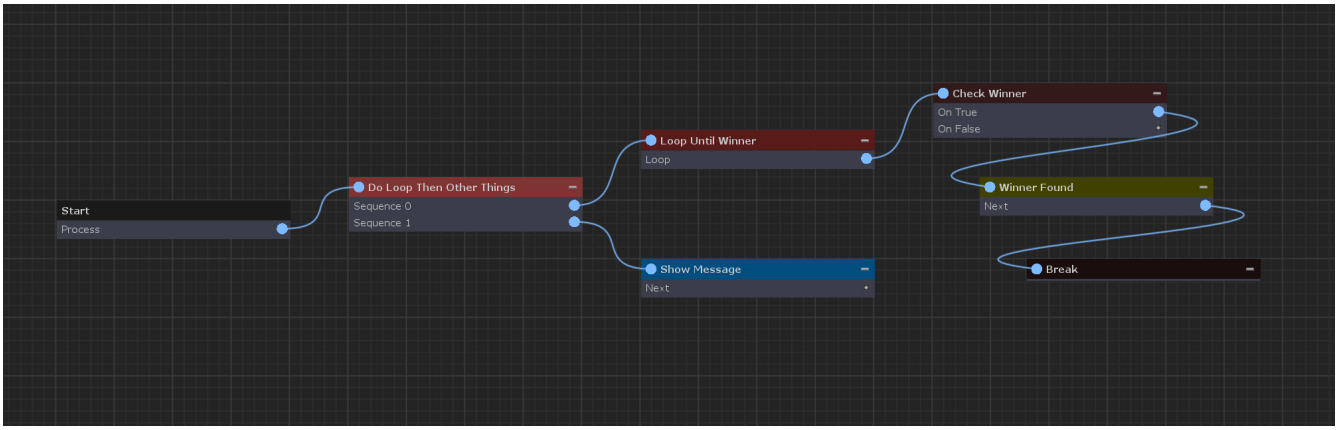
Nodes that implement the ISequenceNode interface act as sequences. A sequence will cause the graph to continually return to this node until is does not have another node to run. For example, the Sequence Node will branch to each one of it outputs in order. That is, when the first branch reaches the end of its execution (when it has no connections) the graph will revert back to the sequence and then run the next branch until all of branches have been run.



For example in the example above each of the four branches of the sequence will run in order (from top to bottom) before the graph ends

Loops

Nodes that loop implement the ILoopNode interface. Looping causes the graph to continually return to this node until it is says that it is done.



In this example, the Loop Node will continue to run through each of its child nodes until a condition is met. To preemptively break out of a loop, use a Break Node. This will tell the graph to stop running the most recently executed ILoopNode. In many cases once a loop has finished, it may be desired to continue executing other nodes. Because loop nodes don't have a concept of where to continue after they are finished executing, it is common to place a sequence node immediately prior to the loop (as shown in the example above) to continue execution. Additionally an Exit Node may be used to break out of all loops and sequences.

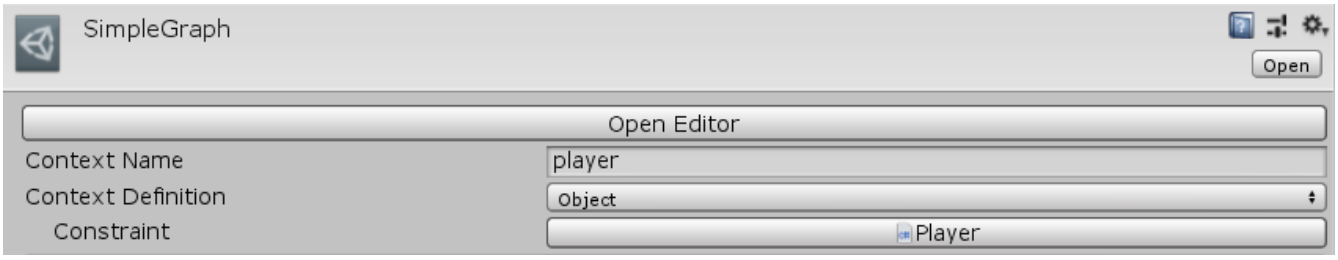
Instruction Store

Every graph is created with an Instruction Store. The instruction store provides access to all the variables that nodes in the graph can access through variable references and expressions (see Accessing Variables for more info). The graph's instruction store provides access to the following variables by string name:

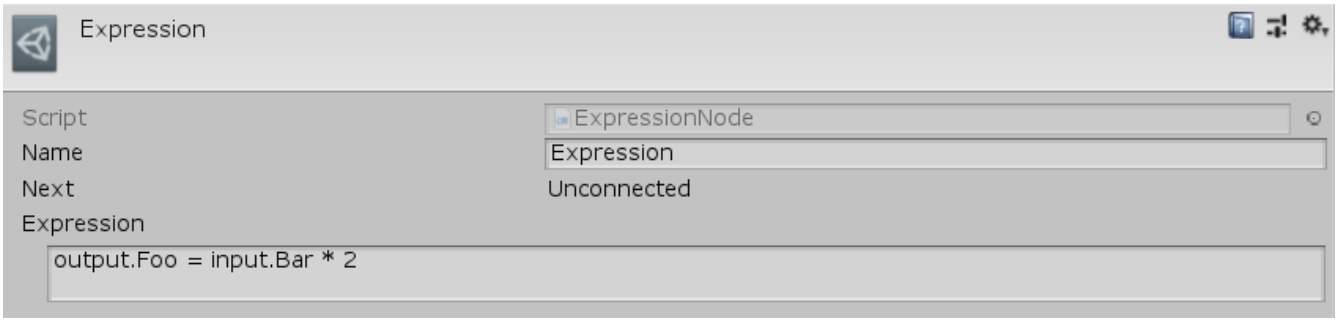
Name	Description
`global`	The global variables stored on the Composition Manager (detailed in Accessing Variables)
`local`	A pool of temporary values created and accessible only by nodes in this graph
`scene`	GameObjects by name in the loaded scenes (detailed in Accessing Variables)
`input`	Values passed into this graph by the instruction caller (detailed below)
`output`	Values returned by this graph to the instruction caller (detailed below)

Context/Inputs/Outputs

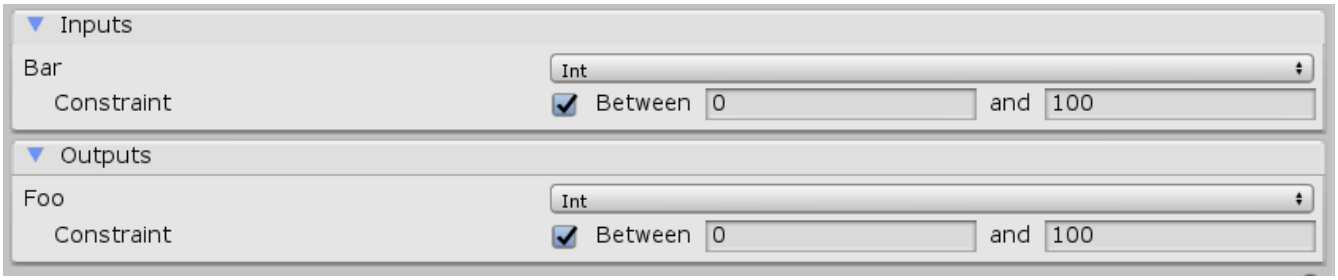
In addition to the values above, each instruction store has a context object that is passed in by the script that ran the instruction. Select a graph in the Project window or click on the graph's start node to select it in the Inpector. Here a graph's context object, inputs, and outputs can be viewed. The context object is accessed by the string enterred as the *ContextName* property. By default the context object can be of any type stored in a variable value. With the *ContextDefinition* property this value can be defined as a certain type and constrained to any desired parameters. When run, the graph will assert that the given context object is of the correct type and constrained correctly. If it is an object type, it will be automatically cast to the defined type.



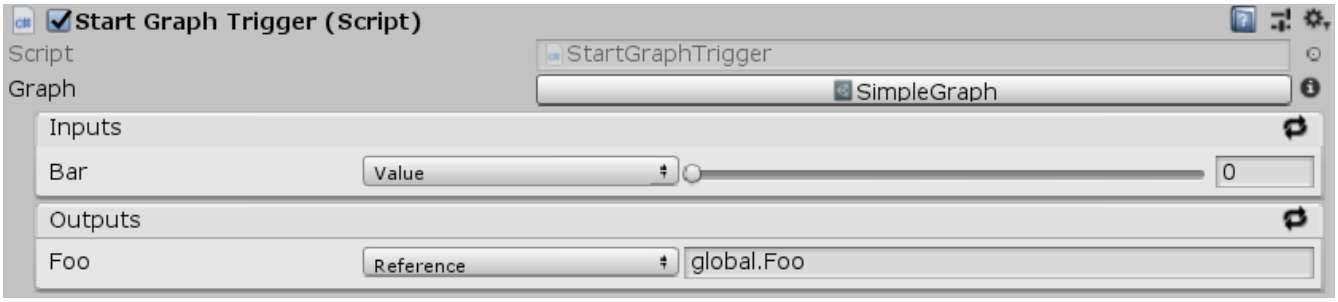
Graphs may have input values that are passed in by the calling object and output values that are subsequently returned to the calling object, similar to parameters and return values of methods in script. Inputs and Outputs are automatically retrieved from each node in the graph that potentially needs them. Consider the following Expression Node:



The variable *Foo* will be automatically added to the graph's output list and the variable *Bar* will be auto added to its inputs. However since an expression contains no type information, like the context, these variables may have types and constraints defined for them on the graph's list of inputs and outputs.



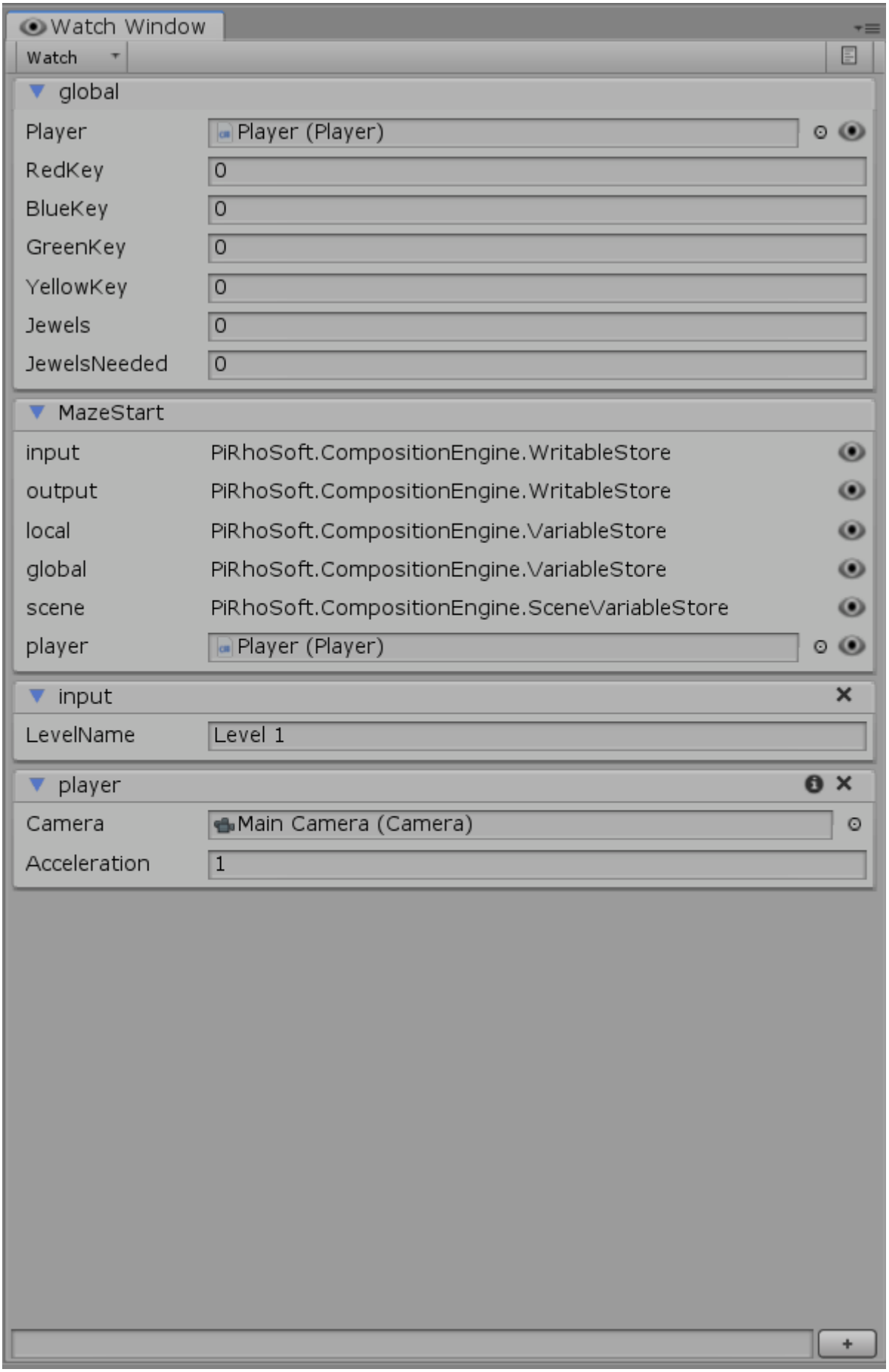
They will also automatically appear in the inspector of an InstructionCaller (see Running Graphs From Script) that will run this graph. The instruction caller is where the actual values of what is passed into the graph are defined.



Debugging

Robust runtime debugging tools are provided to help find errors, profile performance, and inspect values of graphs while they are running. Place breakpoints on individual nodes using the red button on the left side of the Instruction Graph Window's toolbar. Execution will pause when a node with a break point is reached. Playback of the graph can then be controlled using the play, pause, step, and stop buttons  . Use the logging button  to track flow of the graph in the console. Node connections and properties can all be edited at runtime.

Use the Watch Window (**Window › PiRho Soft › Watch Window**) to inspect and edit variables on graph's instruction store while it is running. Use the dropdown at the top and type in the path to a variable store or click on the inspect button to the right of a variable store to add it to the watch list. Use the text box at the bottom of the watch window to run an expression.


Watch Window
Watch
global
Player
Player (Player)
RedKey
0
BlueKey
0
GreenKey
0
YellowKey
0
Jewels
0
JewelsNeeded
0
MazeStart
input
PiRhoSoft.CompositionEngine.WritableStore
output
PiRhoSoft.CompositionEngine.WritableStore
local
PiRhoSoft.CompositionEngine.VariableStore
global
PiRhoSoft.CompositionEngine.VariableStore
scene
PiRhoSoft.CompositionEngine.SceneVariableStore
player
Player (Player)
input
LevelName
Level 1
player
Camera
Main Camera (Camera)
Acceleration
1

Running Graphs From Script

Running graphs from script is as simple as calling `CompositionManager.Instance.RunInstruction()`.

When storing a graph that will be serialized on an object it is important to define it as a InstructionColler instead of a standard instruction. This will ensure that inputs and outputs on the referenced graph are read and written correctly.

The following example will run a graph when its object is loaded:

```
    public class RunGraph : MonoBehaviour
    {
        public InstructionCaller Graph = new InstructionCaller(); ①

        void Start()
        {
            if (Graph.Instruction && !Graph.IsRunning) ②
                CompositionManager.Instance.RunInstruction(Graph,
CompositionManager.Instance.DefaultStore, VariableValue.Create(this)); ③
        }
    }
```

① The graph to run - notice this is an InstructionCaller

② Make sure to check if the caller's instruction is set and not already running from another process

③ Passes the Composition Manager's default store to use for references to inputs and outputs (second parameter) and `this` as the graph's context object (third parameter)

See Instruction Trigger and its derived classes for other examples of running graphs from script.

Creating Custom Graphs

To create a custom graph, derive from InstructionGraph and implement the abstract method `Run(InstructionStore variables)`. The following is an example of a graph that has three different entry points and runs them sequentially:

```
public class ScopedGraph : InstructionGraph
{
    ①
    public InstructionGraphNode Enter = null;
    public InstructionGraphNode Process = null;
    public InstructionGraphNode Exit = null;

    protected override IEnumerator Run(InstructionStore variables)
    {
        ②
        yield return Run(variables, Enter, nameof(Enter));
        yield return Run(variables, Process, nameof(Process));
        yield return Run(variables, Exit, nameof(Exit));
    }
}
```

① InstructionGraphNode fields on a graph will automatically be added as output options visually for a graph in the Instruction Graph Window.

② Internally, graphs operate as a Coroutine which gives fine control of timing behaviour. Because of this, it is important to `yield` the execution of each entry node that this graph will run. The third parameter is simply a label used in debugging for tracking the flow of execution in the Instruction Graph Window.

Ultimately, graphs are just a Asset so they can store data and implement any functionality desired. Execution will end when the `Run()` function finishes.

Creating Custom Nodes

To create custom nodes, derive from InstructionGraphNode and implement the abstract method `Run(InstructionGraph graph, InstructionStore variables, int iteration)`. The following is an example of a custom node that instantiates a game object from a prefab and stores it in a variable reference.

```
    public class SpawnObjectNode : InstructionGraphNode
    {
        public InstructionGraphNode Next = null; ①
        public GameObjectVariableSource Prefab = new GameObjectVariableSource();
        public VariableReference ObjectVariable = new VariableReference();

        public override IEnumerator Run(InstructionGraph graph, InstructionStore
variables, int iteration) ②
        {
            if (ResolveObject(variables, Prefab, out GameObject prefab)) ③
            {
                var spawned = Instantiate(prefab);

                Assign(variables, ObjectVariable, VariableValue.Create(spawned)); ④
            }

            graph.GoTo(Next, nameof(Next)); ⑤

            yield break;
        }
    }
```

① InstructionGraphNode fields on a node will automatically be added as branch options for that node in Instruction Graph Window.

② The third parameter *iteration* will increment if this node is an ISequenceNode or an ILoopNode every time this node is run.

③ Numerous `Resolve()` helper methods are provided on the base InstructionGraphNode class to simplify retrieving typed objects from variable references and variable sources.

④ The `Assign()` helper method is also provided for setting values to references

⑤ Call `graph.GoTo()` to tell the graph that this node has finished performing its actions. If the passed node is `null`, or if `graph.GoTo()` is not called before the run method finishes, then the graph will finish. The second parameter is simply a label used in debugging for tracking the flow execution in the Instruction Graph Window