An Introduction to Vuo

Vuo 0.3

Contents

1 Vuo and You

Vuo is an environment for creating interactive multimedia software. It's designed for multimedia artists and other creative people who want to combine graphics, audio, special effects, and interactivity into unique compositions.

Making a Vuo composition is easy. You drag and drop building blocks ("nodes") onto the canvas. You draw lines ("cables") to connect them. Then you hit the Run button. See, hear, and interact with your composition.

Learning Vuo is easy. Just master one simple concept: information flowing through cables. (In case you were wondering, "Vuo" is the Finnish word for "flow".) The Vuo website provides tutorials, examples, and forums to help you learn.

With Vuo, you can. . .

- Modify your composition while it's running – perfect for experimentation and live improvisation.
- Build apps for iOS, Mac, Windows, and Linux.
- Make your own reusable, shareable nodes.
- Embed your composition inside other software.

Welcome to Vuo. Soon you'll be able to create interactive art and music, animations, visualizations, games, special effects, and more.

2 Quick start

2.1 Install Vuo

TODO (build, website): how to download and install Vuo

2.1.1 The easy way: Getting the Vuo package

2.1.2 The hard way: Building Vuo from source code

2.2 Create a composition

TODO (editor): screenshots

Let's make a simple composition. It will just pop up a window with the text "Hello World!"

1. Open the Vuo Editor application in your Applications folder.
2. Go to `File > New Composition`. This opens a blank canvas.
3. Go to `Window > Show Node Library`. This shows all the nodes that are available to you.
4. In the Node Library, find the **Start** node. Drag it onto the canvas.
5. In the Node Library, find the **Write to Console** node. Drag it onto the canvas.
6. Draw a cable from the **started** port of the **Start** node to the receptor port of the **Write to Console** node.
7. Double-click on the **string** port of the **Write to Console** node. This pops up a text box.
8. Type "Hello world!" in the text box and hit `Return`. This closes the text box.

2.3 Run the composition

TODO (editor): screenshots

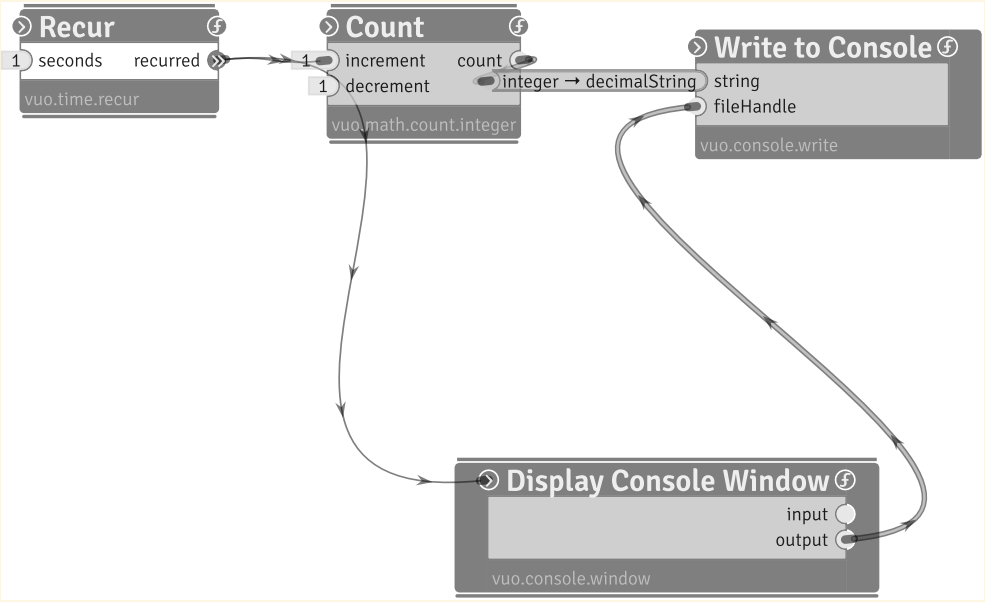
Now let's run your composition.

1. Click the Run button (or go to `Run > Run`).
2. When you're finished admiring the "Hello world!" text, click the Stop button (or go to `Run > Stop`).

2.4 Export the composition to a standalone executable

TODO (export): how to export a composition

3 Making compositions



This is a **composition** that writes the numbers 1, 2, 3, 4, ... to the console, one number every second. Let's look at the composition one piece at a time.

Recur, **Count**, **Convert Integer to Decimal String**, and **Write to Console** are **nodes**. Each node does some work and sends the result to the next node through **cables** connected to the nodes' **ports**.

The **Recur** node generates an **event** every 1 second. It sends that event out its **recurred** port, then along the cable to the **Count** node's **increment** port.

When the event from the **Recur** node hits the **Count** node, it's **Count**'s turn to execute. The **Count** node keeps track of a count. When an event hits its **increment** port, the node adds 1 to its count. The count starts out at 0, becomes 1 after the first event, 2 after the second event, and so on. The **Count** node sends the new count *and*

Note for Quartz Composer users

A **node** in Vuo is analogous to a **patch** in Quartz Composer. Unlike Quartz Composer, which typically executes each patch once per video frame, nodes in Vuo can be executed whenever they receive an event — whether it's 60 per second, 44,100 per second, or 1 per year.

Note for text programmers

A **node** in Vuo is analogous to a **class instance method** that takes a list of inputs and returns a list of outputs.

the event out its **count** port, along the cable, to the **Convert Integer to Decimal String** node's **integer** port.

The **Convert Integer to Decimal String** node takes the value in its **integer** port (a number) and converts it to a string (text). That's because the next node, **Write to Console**, only works with text, not numbers.

The final node, **Write to Console**, prints the count to the console.

In summary, the **Recur** node generates an event every 1 second, which travels along cables and triggers **Count**, then **Convert Integer to Decimal String**, then **Write to Console**.

3.1 Nodes

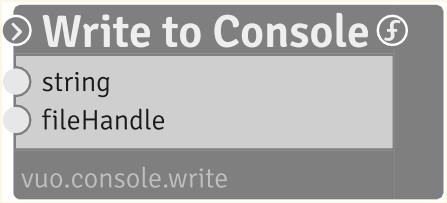
Each **node** in a composition does a small task, like counting numbers or writing to the console or periodically generating events. Nodes are building blocks. You can put them together in all sorts of ways.

3.1.1 Interacting with the environment

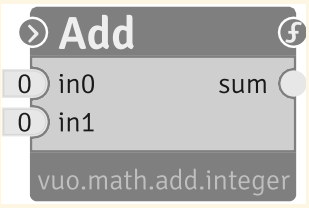
Some nodes interact with the world outside the composition – files, networks, and devices.

Nodes that bring information into the composition from the outside world are called **providers**. An example of a provider is the **Read from Console** node, which reads words typed in the console.

Nodes that affect the outside world are called **consumers**. An example of a consumer is the **Write to Console** node, which writes a message to the console.



A node can be a provider, a consumer, both, or neither. If it's neither, then it's called a **processor**. Examples of processors are the **Add** node, which sums its inputs, and the **Recur** node, which generates events whenever a timer goes off.

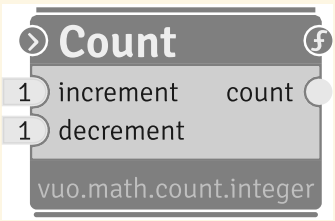


Note for Quartz Composer users

Vuo's interaction modes are distinct from Quartz Composer's execution modes with the same names. In Quartz Composer, a patch's status as a provider, processor, or consumer not only indicates how it interacts with the outside world, but also controls when it executes and whether it can be embedded in macro patches.

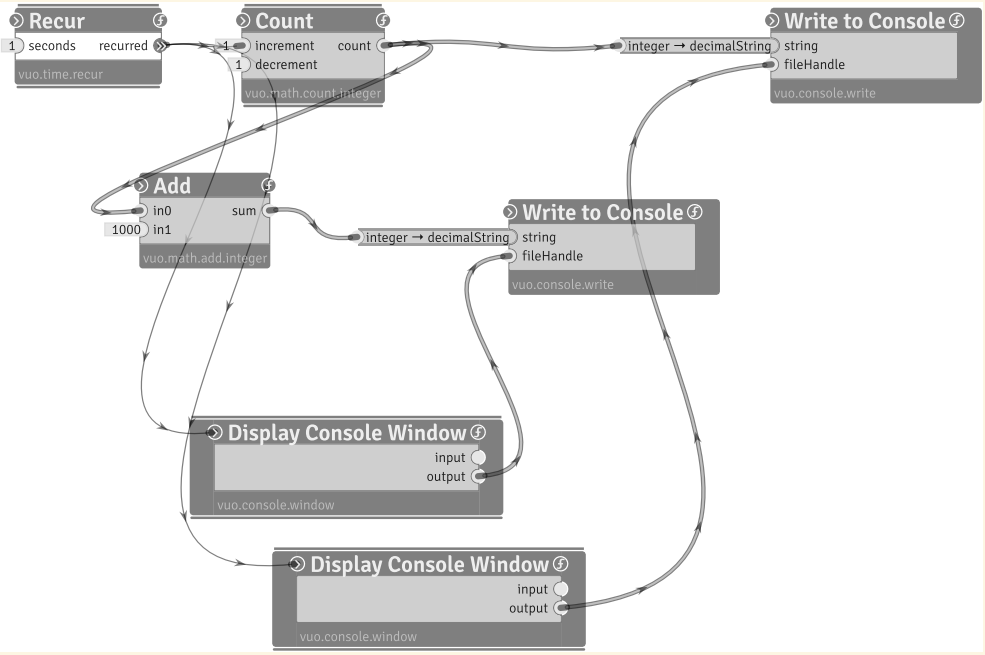
3.1.2 Storing information

Some nodes have **state**. They remember information from previous times they were executed. An example of a **stateful** node is **Count**. If a **Count** node is told to increment, its state (the count) changes.



A **stateless** node doesn't remember anything about previous times it was executed. If you give it the same inputs, it'll always give you the same outputs.

3.2 Ports and cables



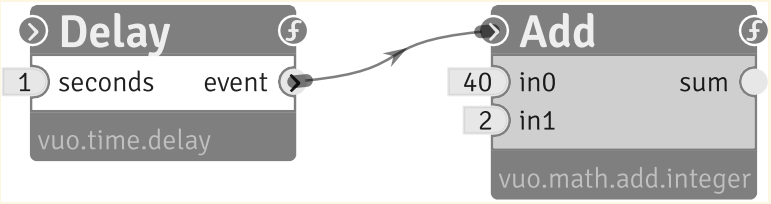
Nodes talk to each other by sending data and events through **cables** plugged into **ports**. Data and events flow from the output port of one node, through a cable, to the input port of another (or the same) node. For example, data and events flow from the **count** output port of the **Count** node to the **integer** input port of the **Convert Integer to Decimal String** node and the **in0** port of the **Add** node.

More than one cable can be connected to an output port, but only one cable can be connected to an input port.

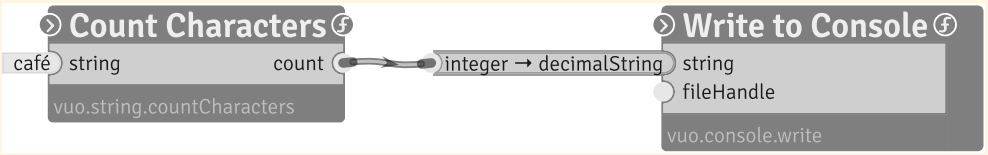
3.2.1 Generating events

Some output ports generate new events. They're called **generator ports**. One example is the **Recur** node's **recurred** port. Another is the **Start** node's **started** port.

3.2.2 Passing data and events between nodes



All cables carry events. An example of a cable that carries an event (and nothing else) is the cable coming out of the **Delay** node's **event** port.

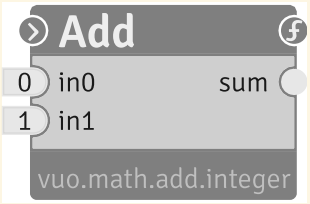


Some cables also carry data. For example, the cable coming out of the **CountCharacters** node's **count** port carries an event plus some data: the character count.

When you connect two ports with a cable, the types of the ports have to match. You can connect an integer port to an integer port, or a string port to a string port, but you can't connect an integer port to a string port. (If you want to convert an integer to a string, use the **Convert Integer to Decimal String** node.) You can connect a data-and-event output port to a data-and-event input port, but you can't connect a data-and-event output port to an event-only input port. (If you want to convert a data-and-event cable to an event-only cable, use a **Discard Data from Event** node.)

(You *can* connect an event-only output port to a data-and-event input port. That's because, as explained below, the port has data in it even if no cable is connected.)

3.2.3 Setting a constant value for a port



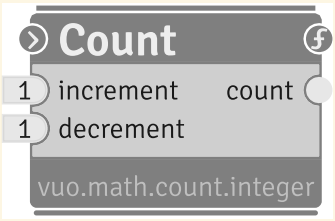
Instead of passing data through a cable, you can give a data-and-event input port a **constant value**.

A constant value is "constant" because, unlike data coming in from a cable, which can change from time to time, a constant value remains the same unless you edit it.

If you change a constant value for a node's port, the node will use the new port value the next time it executes. Setting a port value won't cause the node to execute.

3.2.4 Using the default value of a port

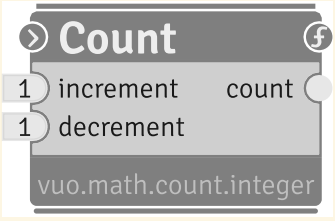
If a data-and-event input port isn't connected to a cable and doesn't have a constant value, then it reverts to its **default value**. The default value for an input port is the same for all nodes of a given type. For example, the **increment** port of all **Count** nodes defaults to 1.



3.3 Conducting events

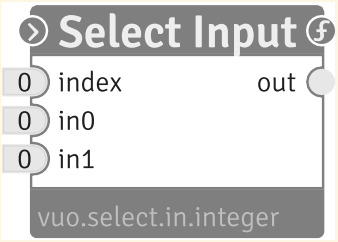
When an event flows along a cable and reaches a node, it triggers the node to **execute**. In all of the examples so far, the node would then transmit the event through its output ports to downstream nodes. But sometimes a node blocks the event so it doesn't flow to downstream nodes.

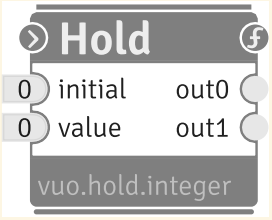
Nodes that are **always conductive** will always transmit an event from any input port to all output ports. Examples of always-conductive nodes are the **Count** node and the **Delay** node.



TODO: vuo.time.delay isn't always-conductive — its 'seconds' port shouldn't conduct, right?

Nodes that are **semi-conductive** may or may not transmit an event to a given output port. It depends on the type of node, the node's input data and events, and the node's state. Examples of semi-conductive nodes are the **Select Input** node and the **Hold** node.

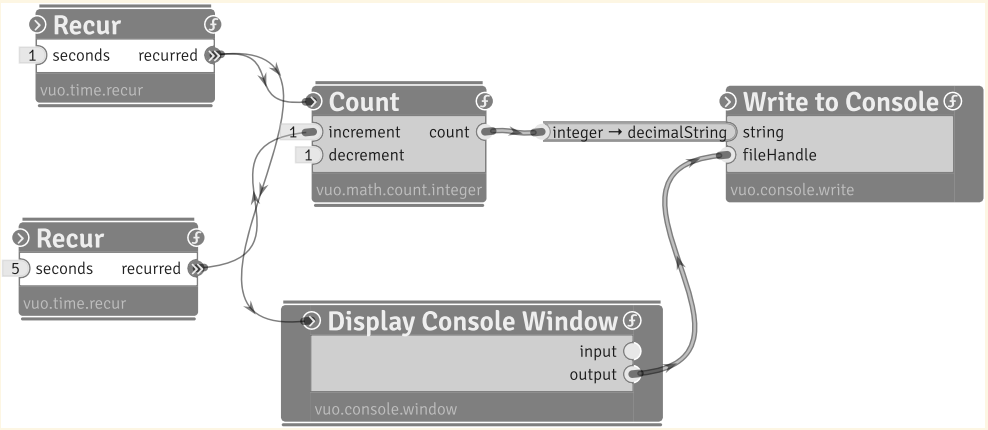




Nodes that have only generator output ports, or no output ports at all, will, of course, never conduct events.

3.3.1 The receptor port

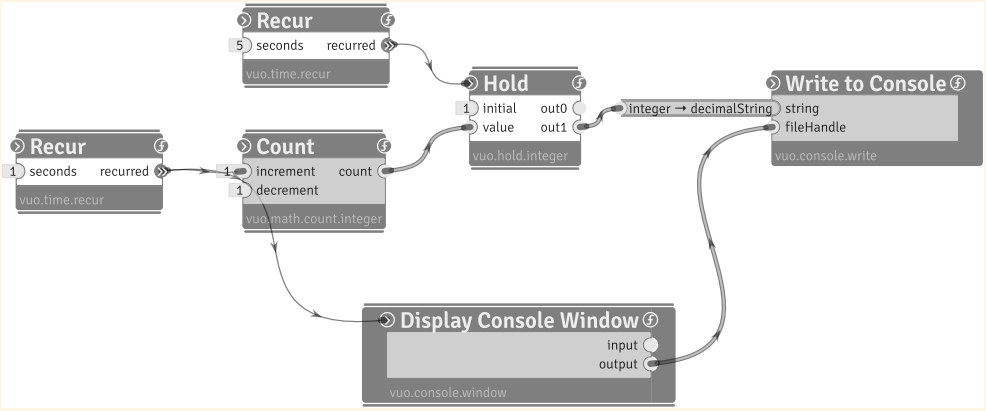
Every node has a built-in event-only input port called the **receptor port**. The behavior of this port depends on the type of node.



On some nodes, like **Count**, the receptor port lets you bypass the work typically done by the node and just conduct the event. For example, this graph shows how you can use the receptor port to get the **Count** node's current count without incrementing or decrementing it.

When the **Recur** node connected to the **Count** node's **increment** port generates an event, **Write to Console** prints the incremented count. When the other **Recur**

node, which is connected to the **Count** node's receptor port, generates an event, the count stays the same. **Write to Console** prints the same count as before.



On other nodes, like **Hold**, the receptor port is the only input port that conducts events. The **Hold** node lets you store a value during one event and use it during later events. This graph shows how you can use a **Hold** node to update a value every 1 second and print it every 5 seconds.

When the **Recur** node connected to **Count** executes, the count is transmitted to the **Hold** node — and stops there. The **Write to Console** node doesn't execute.

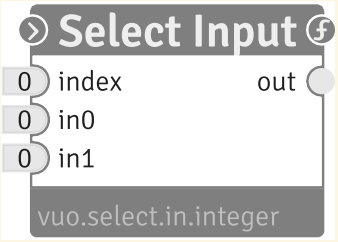
When the **Recur** node connected to **Hold** executes, the count stored in the **Hold** node travels to the **Write to Console** node and gets printed.

3.4 Controlling when nodes execute

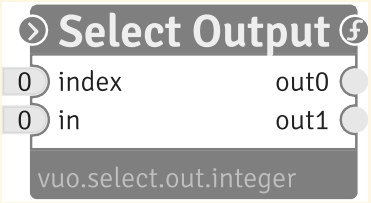
A composition can be as simple as a straight line of nodes, one executing after the other. But it doesn't have to be. A composition can make decisions. It can have feedback loops. It can execute more than one node at a time. You can create compositions that use control flow and concurrency.

3.4.1 Switches

A **Select Input** node lets you pick one of several input values and route it to the output. The input ports are numbered 0, 1, 2, ... You pick an input port by passing its number to the **index** input port.



A **Select Output** node lets you route an input value to one of several outputs. Similar to the **Select Input**, you pick an output port using the **index** input port.



Note for Quartz Composer users

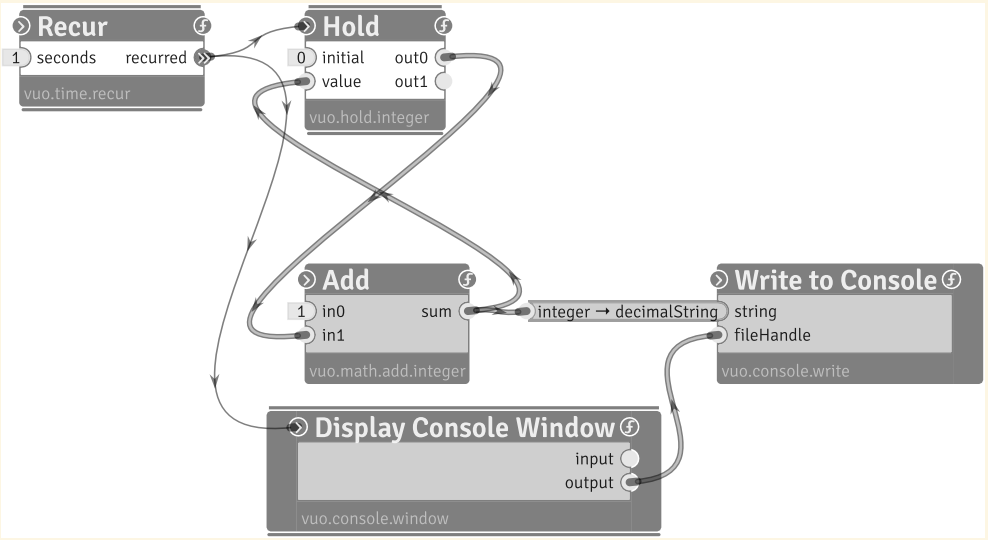
Vuo's **Select Input** node is similar to Quartz Composer's Multiplexer patch. Vuo's **Select Output** node is similar to Quartz Composer's Demultiplexer patch.

Note for text programmers

Vuo's **Select Input** and **Select Output** are similar to `if/else` or `switch/case` statements.

3.4.2 Feedback loops

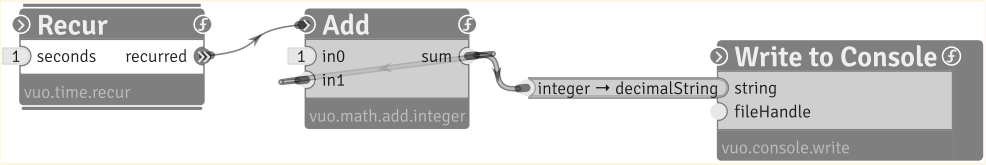
You can use a **feedback loop** to do something repeatedly or iteratively. An iteration happens each time a new event travels around the feedback loop.



This graph prints a count: 1, 2, 3, 4, . . .

The first time the **Recur** node generates an event, the inputs of **Add** are 0 and 1, and the output is 1. The sum travels along the cable to the **Hold** node – and stops there, because **Hold** doesn't conduct events through its **value** port.

The second time the **Recur** node generates an event, the inputs of **Add** are 1 (from the **Hold** node) and 1. The third time, the inputs are 2 and 1. And so on.

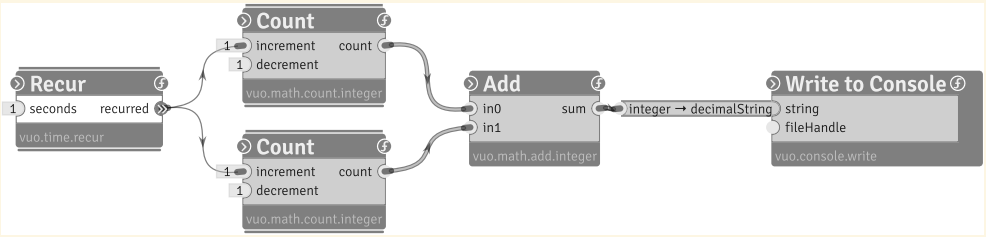


TODO: mark as counter-example

This graph is invalid. That's because any event from the **Recur** node will get stuck forever traveling in the feedback loop from the **Add** node's **sum** port to its **in0** port. Every feedback loop needs a semi-conductive or never-conductive node like **Hold** to block events from looping infinitely.

3.4.3 Executing nodes in parallel

Vuo is a **parallelizing compiler**. It figures out which parts of your composition are safe to execute at the same time. It executes those parts concurrently to make your composition run faster.

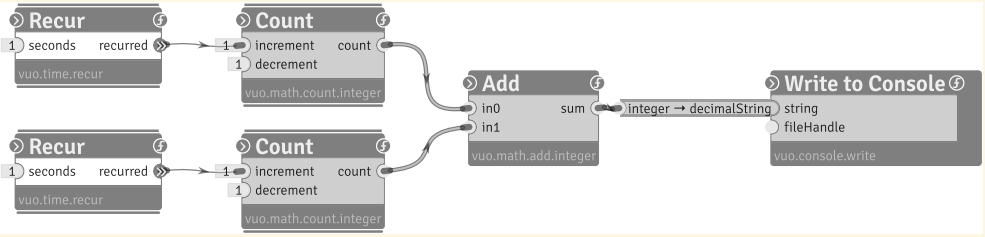


In this graph, the two **Count** nodes are independent of each other, so it's safe for them to execute at the same time. When the **Recur** node generates an event, the upper **Count** node might execute before the lower one, or the lower one might execute before the upper one, or they might execute at the same time. It doesn't matter! What matters is that the **Add** node waits for input from both of the **Count** nodes before it executes.

The **Add** node executes just once each time **Recur** generates an event. The event branches off to the **Count** nodes and joins up again at **Add**.

If the **Recur** node generates an event, then generates a second event before the first has made it through the graph, then the second event waits. Only when the **Write**

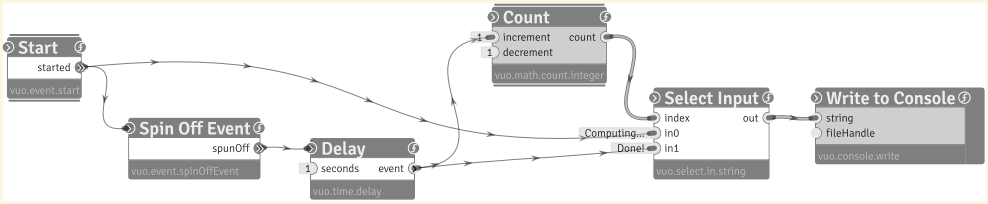
to Console node has finished executing for the first event do the **Count** nodes begin executing for the second event.



In this graph, the **Add** node executes each time either **Recur** node generates an event. If one of the **Add** node's inputs is pushed, it doesn't wait for the other input. It goes ahead and executes.

If the two **Recur** nodes generate an event at nearly the same time, then the **Count** nodes can execute in either order or at the same time. But once the first event reaches the **Add** node, the second event is not allowed to overtake it. (Otherwise, the second event could overwrite the data on the cable from **Add** to **Write to Console** before the first event has a chance to reach **Write to Console**.) The second event can't execute **Add** or **Write to Console** until the first event is finished.

3.4.4 Executing nodes in the background



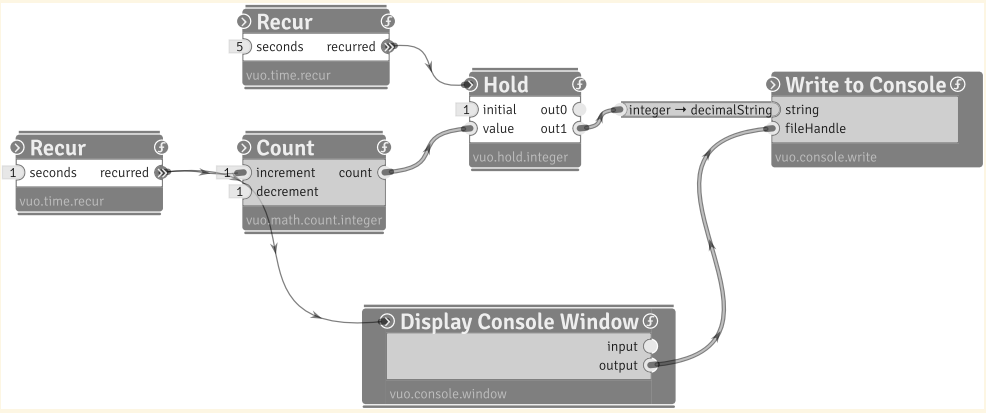
This graph shows how you can do some work in the background and be alerted when it's done. It prints "Loading..." when the work begins and "Done!" when it finishes.

The work is simulated with a **Delay** node, which just waits for 3 seconds before conducting the event to its output port.

The **Spin Off Event** node is what makes the work run in the background. When an event reaches the **Spin Off Event** node, the **Spin Off Event** node doesn't conduct the old event, but it generates a new event.

When the **Start** node generates an event, one branch of the event ends at the **Spin Off Event** node and the other reaches the **in0** port of the **Select Input** (which doesn't wait for input on its **in1** port, since that branch of the event is ended) and goes on to the **Write to Console** node. Meanwhile, the **Spin Off Event** node generates a new event and the **Delay** node begins executing. When it finishes, the new event travels on to **Count**, **Select Input**, and **Write to Console**.

3.4.5 Executing nodes at a steady rate



This graph writes a count to console every 5 seconds. The count updates every second.

The **Hold** node prevents the count from being printed each time it's updated by the 1-second **Recur** node.

(A side note: Every 5 seconds, when the two **Recur** nodes fire at nearly the same time, whether the count will be printed before or after it's incremented is unpredictable.)

3.5 Creating and running compositions with the Vuo Editor

The easiest way to make a composition is in the Vuo Editor. Once you've installed Vuo (see the Quick Start section – TODO: link), you can find the Vuo Editor in your Applications folder.

3.5.1 Drawing nodes and cables

TODO (editor): screenshot of Node Library with one node class selected

When you create a composition, your starting point is always the **Node Library** (`Window > Show Node Library`). The Node Library shows all the nodes that are available to you. In the Node Library, you can search for a node by name or keyword. You can see details about a node, including its documentation and version number.

TODO (build): directory(ies) from which nodes are loaded; what to do if you don't see a node – link to section "Installing a node class"

You can add a node to your composition by dragging it from the Node Library to the canvas. (Or you can double-click on the node in the Node Library.)

You can delete a node by selecting it and hitting `Delete`.

TODO (editor): screenshot of node with data editor

You can change the constant value for an input port by double-clicking the port, then entering the new value into the **Data Editor** that pops up. (Or you can open the Data Editor by hovering the cursor over the port and hitting `Return`.) When the Data Editor is open, press `Return` to accept the new value or `Escape` to cancel.

TODO (editor): screenshot of drawing a cable

You can draw a cable by dragging from an output port to a matching input port. (Or you can drag backwards from an event-only *input* port to a matching *output* port.) When you drag a cable, the ports you can connect it to are highlighted.

You can delete a cable by dragging the end away from the input port and releasing it. (Or you can select the cable and hit `Delete`.)

3.5.2 Running and stopping a composition

TODO (editor): screenshot of Run and Stop buttons

You can run a composition by clicking the Run button. (Or go to `Run > Run`.)

You can stop a composition by clicking the Stop button. (Or go to `Run > Stop`.)

3.5.3 Exporting compositions

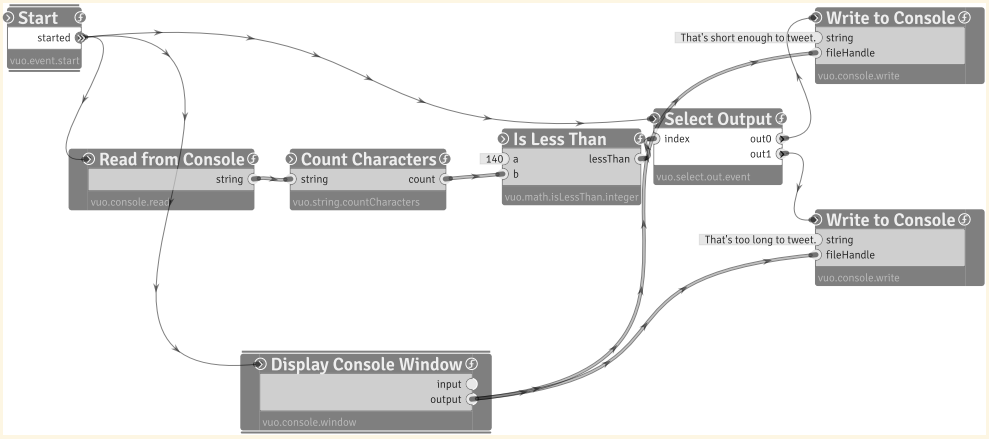
TODO (export): how to export a composition

3.6 Creating compositions with a text editor and running them with a shell

Sometimes you may want to work with compositions outside of the Vuo Editor. Maybe you want to write a script that compiles and runs compositions. Or maybe you want to edit a composition over SSH. Or maybe you just want to peek under the hood and learn what the Vuo Editor is doing. You can.

3.6.1 Writing a composition in a text editor

Composition files are in Graphviz DOT format, an open format for describing nodes and edges (cables).



Listing 1: TweetLength.vuo

```
1  */
2
3  digraph G
4  {
5  Start1 [type="vuo.event.start" label="Start|<receptor>receptor\l|<started>started\r"
       pos="0,0"];
6  ReadfromConsole1 [type="vuo.console.read" label="Read from Console|<receptor>receptor\l
       |<string>string\r" pos="80,140"];
```

```
CountCharacters1 [type="vuo.string.countCharacters" label="Count Characters|<receptor>receptor\l|<string>string\l|<count>count\r" pos="280,140"];
IsLessThan1 [type="vuo.math.isLessThan.integer" label="Is Less Than|<receptor>receptor\l|<a>a\l|<b>b\l|<lessThan>lessThan\r" pos="490,120" _a="140"];
SelectOutput1 [type="vuo.select.out.event" label="Select Output|<receptor>receptor\l|<index>index\l|<out0>out0\r|<out1>out1\r" pos="640,100"];
WritetoConsole1 [type="vuo.console.write" label="Write to Console|<receptor>receptor\l|<string>string\l|<fileHandle>fileHandle\l" pos="800,0" _string="That's short enough to tweet."];
WritetoConsole2 [type="vuo.console.write" label="Write to Console|<receptor>receptor\l|<string>string\l|<fileHandle>fileHandle\l" pos="800,200" _string="That's too long to tweet."];
DisplayConsoleWindow [type="vuo.console.window" label="Display Console Window|<receptor>receptor\l|<input>input\r|<output>output\r" pos="250,343"];

Start1:started -> DisplayConsoleWindow:receptor;
Start1:started -> ReadfromConsole1:receptor;
Start1:started -> SelectOutput1:receptor;
ReadfromConsole1:string -> CountCharacters1:string;
CountCharacters1:count -> IsLessThan1:b;
IsLessThan1:lessThan -> SelectOutput1:index;
SelectOutput1:out0 -> WritetoConsole1:receptor;
SelectOutput1:out1 -> WritetoConsole2:receptor;
DisplayConsoleWindow:output -> WritetoConsole1:fileHandle;
DisplayConsoleWindow:output -> WritetoConsole2:fileHandle;
}
```

Let's look at TweetLength.vuo. It declares each of the nodes (**Start**, **Read from Console**, etc.) followed by each of the cables (`Start:started -> ReadfromConsole:receptor`, etc.). For each node, it gives the type (e.g. `vuo.event.start`) and a label listing all of the node's ports. The label is hard to read, but don't worry about typing it exactly right — you can just copy and paste the node declaration from the output of this command:

Listing 2: Showing all node classes

```
vuo-compile --list-node-classes=dot
```

For each node, TweetLength.vuo also gives a constant value for each data-and-event input port that isn't connected to a cable. For example, the **a** port of the **Is Less Than** node has a constant value of 140.

For each cable, TweetLength.vuo gives the starting port and the ending port. For example, `Start:started -> ReadfromConsole:receptor` declares a cable extending from the **started** output port of the **Start** node to the **receptor** input port of the **Read from Console** node.

3.6.2 Rendering a composition on the command line

Listing 3: Rendering a composition

```
vuo-render --output-format=pdf --output TweetLength.pdf TweetLength.vuo
```

Since composition files are in DOT format, you can also render them without Vuo styling using Graphviz:

Listing 4: Rendering a Vuo composition using Graphviz

```
dot -Grankdir=LR -Nshape=Mrecord -Tpng -oTweetLength.png TweetLength.vuo
```

3.6.3 Building a composition on the command line

You can turn a .vuo file into an executable in two steps.

First, compile the .vuo file to a .bc file (LLVM bitcode):

Listing 5: Compiling a Vuo composition

```
vuo-compile --output TweetLength.bc TweetLength.vuo
```

Then, turn the .bc file into an executable:

Listing 6: Linking a Vuo composition into an executable

```
vuo-link --output TweetLength TweetLength.bc
```

(These are separate steps because you might use a compiled .vuo file as a subgraph inside another .vuo file, instead of as a standalone executable.)

You can run the following commands to see more options:

Listing 7: Compiler and linker help

```
vuo-compile --help
vuo-link --help
```

3.6.4 Running a composition on the command line

Run the executable you created just like any other executable:

Listing 8: Running a Vuo composition

```
./TweetLength
```

3.6.5 Running a composition inside a C/C++/Objective-C program

TODO: building a graph to .bc, linking your code with that .bc, and calling the graph's main function

3.6.6 Dynamically compiling and running compositions inside a C/C++/Objective-C program

TODO: explain programmatically invoking the compiler and using our wrapper around LLVM's ExecutionEngine->getPointerToFunction()

4 Adding node classes to your Node Library

Node class is the technical term for "type of node". Vuo comes with a built-in set of node classes, all of which are listed in the Node Library in the Vuo Editor. A node class is like a blueprint or cookie cutter for creating nodes. You can create a node by picking a node class from the Node Library and dragging it onto the Canvas.

There are many reasons that you might want to add node classes to your, or someone else's, Node Library:

- You find yourself making the same subgraph over and over in different compositions. You want to make it once and reuse it.
- You made a cool subgraph that you want to share, so other Vuo users can add it to their Node Library.

- You want to create node classes as a front end or wrapper to let Vuo users use an existing library.

First, let's talk about how to install a node class created by someone else. Then we'll talk about creating a node class of your own.

4.1 Installing a node class

A node class is distributed as a .bc (LLVM bitcode) file. To install a node class, just place the .bc file in one of these folders:

- In your home folder, go to Library > Application Support > Vuo > Modules.
  - On Mac OS X 10.7 and above, the Library folder is hidden by default. To find it, go to Finder, then hold down the Option key, go to the Go menu, and pick Library.
- In the top-level folder on your hard drive, go to Library > Application Support > Vuo > Modules.

You'll typically want to use the first option, since yours will be the only user account on your computer that should have access to the node class. Use the second option only if you have administrative access and you want all users on the computer to have access to the node class.

You can organize the files in subfolders inside your nodes folder any way you want. Vuo will search the nodes folder and all of its subfolders for node classes.

4.2 Creating a node class in the Vuo Editor

TODO (subgraphs)