# Vuo 1.2.8

# Contents

# 1 Introduction

Vuo is a realtime visual programming environment designed to be both incredibly fast and easy to use. With Vuo, artists and creative people can create audio, visual, and mixed multimedia effects with ease.

This manual will guide you through the process of installing Vuo and creating your first composition. Then it will show you the details of all the pieces of a composition and how to put them together. It will explain how to create and run compositions with the Vuo Editor and, as an alternative, the command-line tools. It will show you how to make Vuo do even more by adding nodes to your Node Library.

Many of the compositions in this manual are available from within the Vuo Editor's File > Open Example menu. It may help to open these example compositions in the Vuo Editor and run them as you work through the manual.

For more resources to help you learn Vuo, see our support page on vuo.org.

For more information on what you can do with Vuo, see our features page on vuo.org.

If you have any feedback about this manual, please let us know. We want to make sure you find the manual helpful.

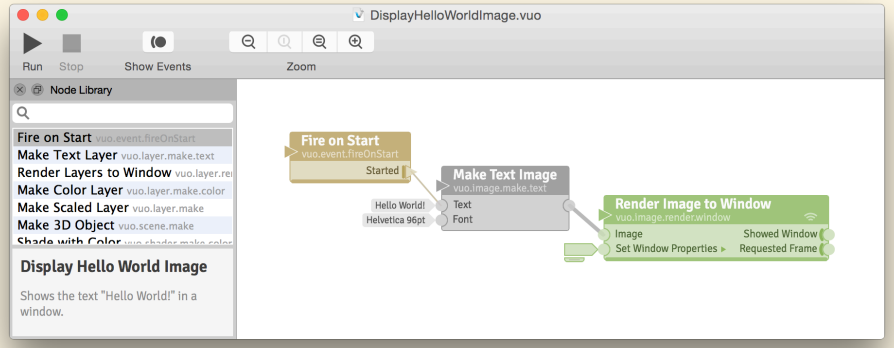
Welcome to Vuo. Get ready to create!

# 2 Quick start

## 2.1 Install Vuo

- Go to https://vuo.org/download.
- Click the “Download Vuo” button.
- Uncompress the ZIP file (double-click on it in Finder).
- Move the Vuo Editor application to your 🗀 `Applications` folder.
- Open the Vuo Editor application from the 🗀 `Applications` folder and the Vuo Editor will appear. The first time you open it, you’ll see a dialog box prompting you to activate your software.
- Click the “Activate Vuo” button to launch a web browser and complete the activation process. Depending on your browser, the activation might happen automatically, or you might need to follow the instructions on the web page.

## 2.2 Create a composition




Let’s make a simple composition. It will just pop up a window with the text “Hello World!”

1. By default, the Vuo Editor displays a “Welcome to Vuo” window when it is first opened. Either click the “Create a new composition” button in that window or go to File ▸ New Composition.
2. A composition window will appear. The Node Library will be on the left, and a canvas with a **Fire on Start** node will be on the right. The **Fire on Start** node is already on the canvas. Many compositions use a **Fire on Start** node, and it helps to remind you that Vuo uses events (such as the event that is fired out of the **Started** port) to make things happen in a Vuo composition.

3. In the Node Library, find the **Make Text Image** node. You can do this by typing part of the node title (such as “text”) into the search bar at the top of the Node Library. Drag the node onto the canvas. You can see that “Hello World!” is written in next to the **Text** input port. The font and size are written next to the **Font** input port.
4. Double-click on the port (circle) next to the **Font** input port. This pops up an input editor. Change the font size to 96 by either typing it in, or using the font size slider. Click on “OK” when you’re done.
5. In the Node Library, now find the **Render Image to Window** node. As before, you can do this by typing part of the node title (such as “render image”) into the search bar at the top of the Node Library. Drag the node onto the canvas.
6. Draw a cable from the **Started** port of the **Fire on Start** node to the **Text** port of the **Make Text Image** node. You can do this by pressing the mouse button while over the **Started** port, dragging the cable that appears, and releasing the mouse button while over the **Text** port.
7. Now draw a cable from the output port on the right side of the **Make Text Image** to the **Image** port of the **Render Image to Window**.

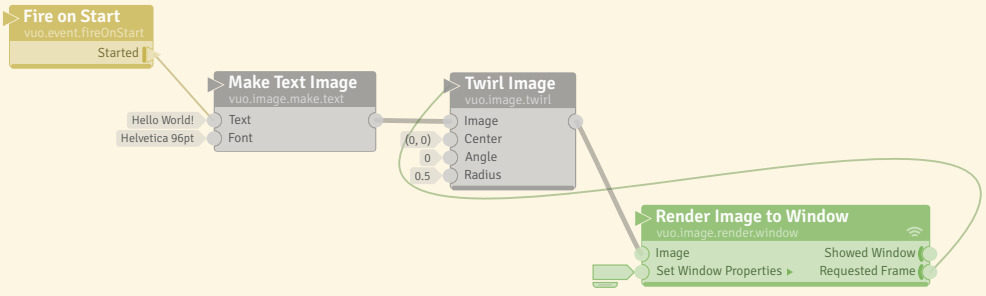
## 2.3 Run the composition

Now let’s run your composition.

1. Click the Run button (or go to Run > Run).

## 2.4 Modify the composition

You can modify your composition while it is running. This is called **live coding**. You can change data, rearrange cables, and even add nodes while a composition is running.




1. In the Node Library, search for the **Twirl Image** node, and drag it onto the canvas.

2. Put your mouse cursor on top of the cable that is going from **Make Text Image** to **Render Image to Window**, and click on it. It will change color to show you have selected it. Press the ⌫, or use the Editor's menu Edit 〉 Cut or Edit 〉 Delete to delete the cable.
3. Draw a cable to connect up the output port of the **Make Text Image** node to the **Image** port on the **Twirl Image** node.
4. Draw a cable from the output port of the **Twirl Image** node to the **Image** port on the **Render Image to Window** node.
5. Notice that nothing has happened yet in your composition. This is because Vuo uses events to drive changes in the composition. Draw a cable from the **Requested Frame** output port on the **Render Image to Window** node to the refresh port (the triangle on the top left of the **Twirl Image** node).
6. Since Vuo is event-driven, you'll need an event to push the image into the new **Twirl Image** node. Right click **Fire on Start**'s **Started** trigger port. You'll see a menu that includes the option, Fire Event. (Don't see the Fire Event option? Make sure your composition is still running.) Click on this option to fire an event to carry the image through the cable to the **Twirl Image** node.
7. Now you can see that your text is distorted.
8. Double-click on the **Angle** input port of **Twirl Image**. By moving the slider you can see the twirl distortion increase or decrease. This is because the **Requested Frame** node's output port is sending a stream of events to the **Twirl Image** node and these events carry the changes to the **Render Image to Window** node.
9. Click on the Show Events button in the Vuo Editor toolbar and you'll see a representation of the events flowing out of the **Render Image to Window** to the **Twirl Image** node.

## 2.5 Close your composition

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

## 2.6 How to learn more

There are different ways to learn about Vuo, depending on how you learn best. You can read more of the manual, watch tutorials, or open example compositions in the Vuo Editor File 〉 Open Example.

### 2.6.1 Example compositions

Examples are arranged by the node set they help explain, such as vuo.image or vuo.math. From File 〉 Open Example in the Vuo Editor, you can hover over a node set to see relevant examples. Hovering

over vuo.color, for example, will display the example composition, Explore Color Schemes. Clicking on an example composition will open it in the Vuo Editor. Alternately, if you know the title of an example composition, you can use the Help > Search box to search for words within the composition title.

To learn more about an example composition, look at the Editor's Node Documentation Panel. This displays the composition's description, which includes instructions on how you can interact with it.

To see a list of descriptions for all example compositions for a node set, look in the node set documentation. In the Editor's Node Library (Window > Show Node Library), find a node that belongs to that node set. For example, to learn more about the vuo.math example compositions, use the keyword "math" in the Node Library search window to find the nodes in that node set, or click on a node in the library that contains "math" in its class name. You can see class names by using View > Node Library > Display by class. Clicking on any node in the node set will show that node's description in the Node Documentation Panel. The description will contain a link to the node set documentation. Using that link will display some general information about that node set, as well as descriptions of the associated example compositions.

More examples are available in the Vuo composition gallery. You can filter compositions in the gallery by using the tabs. The "Helpful" tab contains compositions other users have voted helpful.

Many of the compositions illustrated in this manual are provided as example compositions in the Editor. These are indicated with a menu path to the example composition, such as File > Open Example > vuo.motion > Wave Circle.

# 3 The basics

The previous section walked you through the steps of creating a simple composition. By now, you may know a bit about the process of composing with Vuo, but you may not understand exactly how compositions work or how make your own from scratch. This section introduces the major concepts you need to understand when working with Vuo. The sections after this — How events and data travel through a composition, How compositions process data, and How nodes can be used as building blocks — go into more detail about these concepts. By the end of the section that follows — Putting it all together in a composition — you should be amply prepared to create your own compositions.

## 3.1 A composition is what you create with Vuo

When musicians create a piece of music, they call it a composition. When you create something in Vuo, that's also called a **composition**.

In the Quick Start section, you saw how to create a composition that displays some text in a window and twirls the text around. That's one type of Vuo composition — an animation that displays in a window. Vuo can be used to create much more complex and interesting animations. It can also be used to create many other types of compositions. A composition could be a game. It could be an art installation. It could be a controller for stage lighting. It could be digital signage. It could be a plug-in for other software. Those are just some examples of what a composition could be.

One thing that all compositions have in common is the process of creating them in the Vuo Editor. Just like in the Quick Start section, you start with either a blank canvas or an existing composition, and you pick out building blocks and connect them to make many smaller pieces work together as a larger whole.

Another thing that all compositions have in common is the way that they run. When you click the Run button in the Vuo Editor, all of those building blocks and connections that you laid out as a blueprint get turned into a running application.

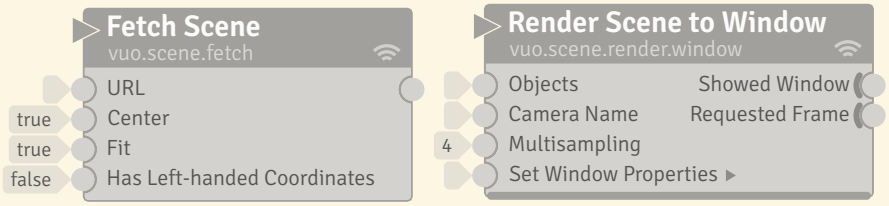
Note for text programmers

A composition is a program whose source code is a visual representation of the program's data flow. It's compiled and linked to create an application or library.

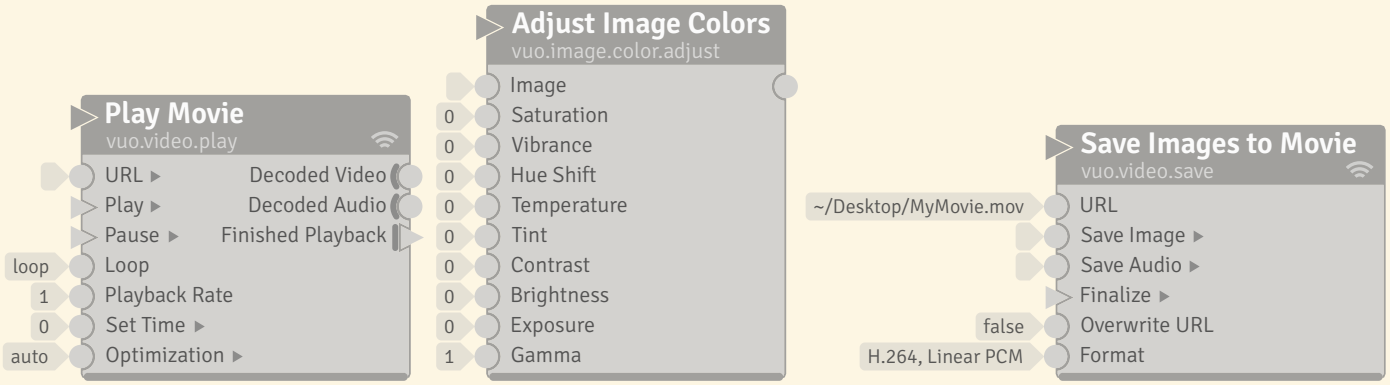
## 3.2 Nodes are your building blocks

Each composition does something unique, and the way that you build up that something is by putting together **nodes**. These are your building blocks.

Let's say you're creating a composition that displays a 3D model. You might use the **Fetch Scene** node to load the 3D model from a file and the **Render Scene to Window** node to render the model in a window.



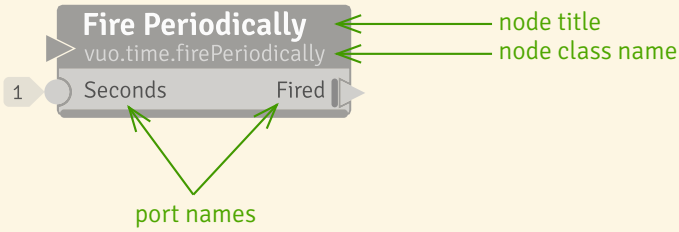

Or suppose you're creating a composition that applies a color effect to a movie. You might use the **Play Movie** node to bring the movie into the composition, the **Adjust Image Colors** node to change the movie's color, and the **Save Images to Movie** node to save the color-changed movie to a file.




Part of the process of creating a composition is taking your idea of what it should do and breaking that down into smaller tasks, where each task is carried out by a node. Each node in Vuo has a specific job that it does. Some nodes do simple jobs, like adding numbers or checking if two pieces of text are the same. Other nodes do something complex, like receiving a stream of video from a camera, finding a barcode in an image, or turning a 3D object into a wiggly blob. You can browse through a list of all the nodes available in the Node Library (the panel along the left side of the Vuo Editor window) or the online node documentation.

When you start making a composition, often the first thing you'll do is pick a node from the Node Library. You can search the Node Library for what you want to do (for example, a search for "movie" brings up a list of nodes for playing, inspecting, and saving movies) and then drag the nodes you want onto the composition canvas.

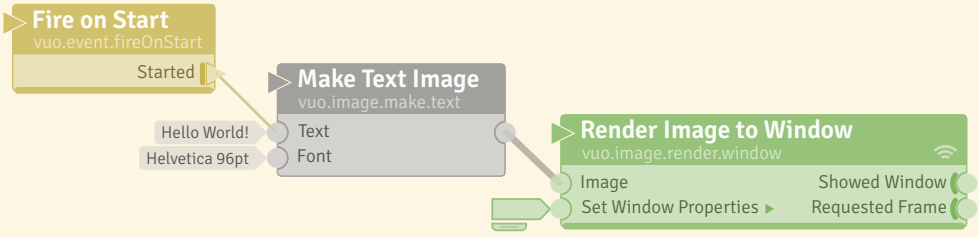
You can learn about a node by looking at its title, node class name, and port names, which are pointed out in the illustration below. For a detailed description of how the node works, you can look at the node's documentation, which appears in the Node Documentation Panel in the lower part of the Node Library. Many nodes come with example compositions (listed in the node's documentation) that demonstrate the node in action.




## 3.3 Events are what cause nodes to execute

Let's think again about creating a composition that applies a color effect to a movie. Your first step might be to drop a **Play Movie** node, an **Adjust Image Colors** node, and a **Append to Movie** node onto the canvas. Then what? How do you tell the composition that, first, you want **Play Movie** to bring the movie into the composition, second, you want **Adjust Image Colors** to apply the effect, and third, you want **Append to Movie** to save the movie to a file? The way that you control *when* nodes do their job and *how* information flows between them is with **events**.

In the Quick Start section, you walked through the process of creating a composition that displays some text:




How do events come into play in this composition? This composition involves a single event that causes the text to render as soon as the composition starts running. The event is **fired** (originates) from the **trigger port** called **Started** on the **Fire on Start** node. (A trigger port is a special kind of port, which you can recognize by the thick line along its left side.) The event travels to the **Make Text Image** node, causing that node to **execute** (do its job). The event then travels onward to the **Render Image to Window** node, causing it to execute as well. From the **Make Text Image** node to the **Render Image to Window** node, the event carries with it the image that was created by **Make Text Image** and will be rendered by **Render Image to Window**.

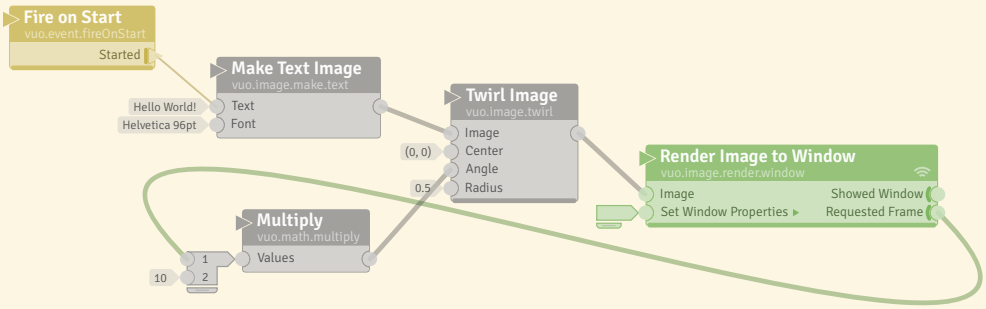
**Note for Quartz Composer users**

If you're familiar with the way that patches execute in a Quartz Composition, then Vuo takes some getting used to. In Quartz Composer, when patches execute is largely based on the display refresh rate, and influenced by patch execution modes and interaction ports. Data is usually "pulled" through the composition by rendering patches. In Vuo, data is "pushed" through the composition by events fired from trigger ports.

**Note for text programmers**

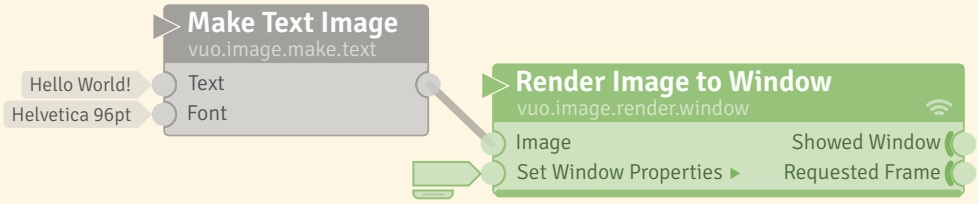
Vuo is event-driven. The events are generated by trigger ports, and the event handlers are implemented by the nodes executed as the event travels through the composition.

Here's a variation on that composition that involves multiple events:




This composition displays an animation of the text becoming more and more twirled as time passes. It still has the event fired from the **Fire on Start** node's **Started** port when the composition starts. It also has events being fired from another trigger port: the **Render Image to Window** node's **Requested Frame** port. Unlike the **Started** port, which fires only once, the **Requested Frame** port fires 60 times per second (or whatever your computer display's refresh rate is). Unlike the event from **Started** port, which is useful for doing something once, the events from the **Requested Frame** port are useful for doing something continuously, such as displaying an animation that changes smoothly over time. In the composition above, each of those 60 times per second that the **Requested Frame** node fires an event, that event (along with a piece of information that says how long the composition has been running) travels to the **Multiply** node. The event (along with the result of multiplying numbers) travels to the **Twirl Image** node. Finally, the event (along with the twirled image) travels to the **Render Image to Window** node. As the event travels along its path, it causes each node to execute in turn, and carries information with it from one node to the next.

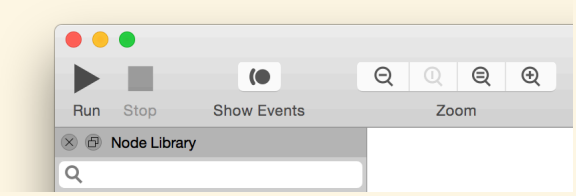
Here's a composition that *doesn't* display text in the window (can you guess why?):




This composition doesn't have any events going into the **Make Text Image** node. Without any incoming events, the **Make Text Image** node never executes and never passes an image along to the **Render Image to Window** node. So no text is displayed. If you want a node to execute, make sure you feed it some events!

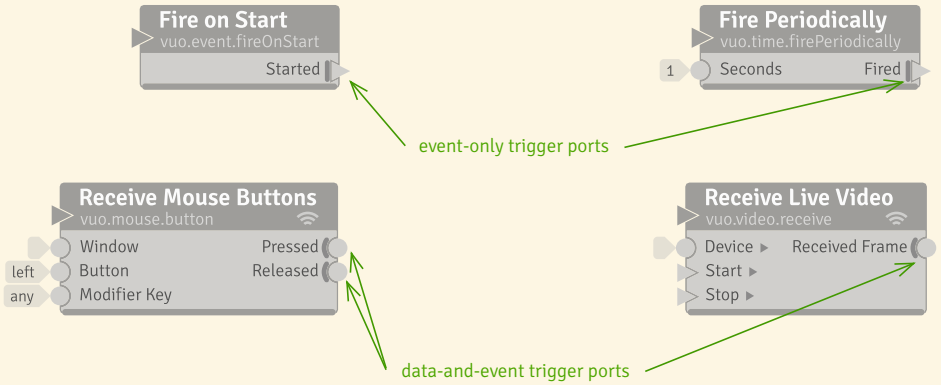
If you'd like to watch the events moving through a composition, you can do that in the Vuo Editor by clicking the Show Events button in the toolbar. As the composition runs, you can see the events being

fired from trigger ports, and you can trace the path of the event by watching each node change color as it executes.




## 3.4 Trigger ports fire events and sometimes data

As you just saw, events are fired from trigger ports, which are special ports that some nodes have. Here are some examples of trigger ports:

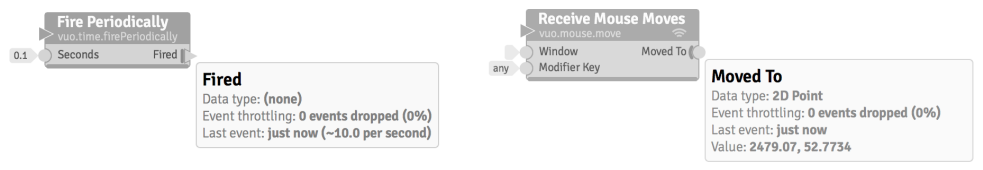



The **Started** trigger port on the **Fire on Start** node fires a single event when the composition starts running. The **Fired** trigger port on the **Fire Periodically** node fires events at a rate determined by the node's **Seconds** port. The **Pressed** trigger port on the **Receive Mouse Buttons** node fires an event each time the mouse button is pressed, and the **Released** trigger port fires an event each time the mouse button is released. The **Received Frame** trigger port on the **Receive Live Video** node fires events as it receives a stream of images from a camera.

Some trigger ports, like **Started** and **Fired**, fire just events. Other trigger ports, like **Pressed**, **Released**, and **Received Frame**, fire **data** (a piece of information) along with each event. The **Pressed** and **Released** ports fire the coordinates of the point where the mouse was pressed or released. The **Received Frame** port fires the video frame received from the camera. This data travels along with the event to the next node. When that node executes, it can use the data to do its job (such as drawing a shape at the given coordinates, or extracting an image from the given video frame).

Nodes with trigger ports are often responsible for bringing information into the composition from the outside world, such as video, audio, device input, and network messages. These nodes can be a good starting point when creating a composition. You can see a list of all nodes with trigger ports by searching the Node Library for “trigger” or “fire”.

As just mentioned, one way to watch what trigger ports are doing in a composition is to run the composition with Show Events enabled. Another way is to click on the trigger port, which opens a view called the Port Popover. As the composition runs, the Port Popover shows how recently the trigger port fired an event and what data (if any) came with the event.

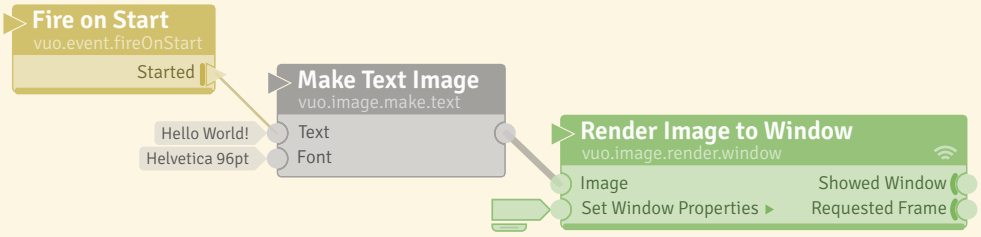



**+ Tip**

If you click on the Port Popover, it becomes a small window that you can leave open as you continue using the Vuo Editor and perhaps open other Port Popovers.

## 3.5 Events and data travel through cables

Let’s take yet another look at this composition that displays text in a window:




The lines connecting the nodes are called **cables**. Cables are the conduits that data and events travel through.

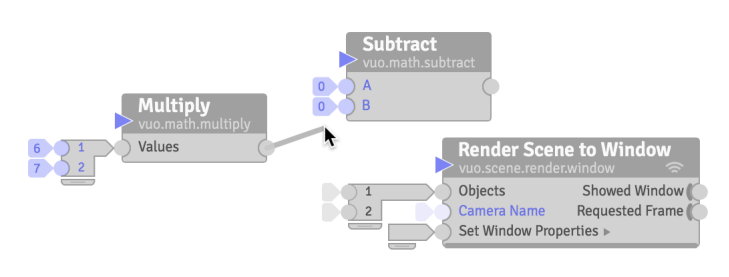
In the composition above, an event travels along the cable from the **Started** trigger port of the **Fire on Start** node to the **Text** port of the **Make Text Image** node. An event and data travel along the cable from the **Make Text Image** node to the **Render Image to Window** node’s **Image** port. Notice the difference between the two cables: the first cable is thinner since it only carries events, while the second cable is thicker since it carries both events and data.

Often it helps to think of cables as pipes that data and events flow through. Like water flowing through a pipe, events and data flow through the cable from one end to the other, always in the same direction. Extending the water analogy, you can think of trigger ports as being **upstream** and the nodes that their events flow to as being **downstream**.

But, unlike water flowing through a pipe, events and data travel as discrete packets instead of a continuous flow. Another way to think of a cable is as a one-way, one-lane road on which each event is a car. On some roads (data-and-event cables), each car carries a piece of data.

Data can't travel along a cable by itself. It always needs an event.

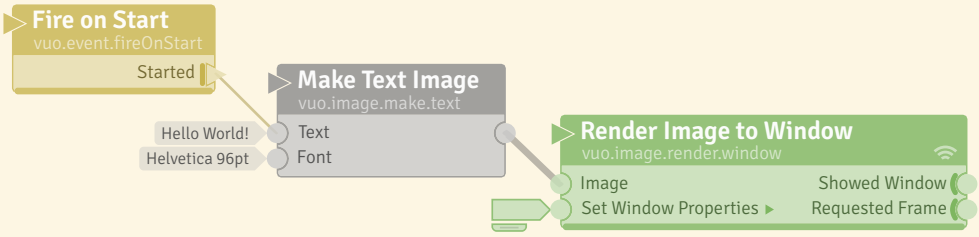
In the Vuo Editor, you can create a cable by dragging the mouse from one port to another. While you're dragging, the ports that you're allowed to connect the cable to are highlighted. If you're not allowed to connect a cable from one port to another, it's because the two ports have different, incompatible types of data. For example, you can't connect a port whose data is a number to a port whose data is a 3D model.




## 3.6 Events and data enter and exit a node through ports

When an event (and possibly data) is fired from a trigger port and travels along a cable, what happens when it reaches the port on the other end of the cable?

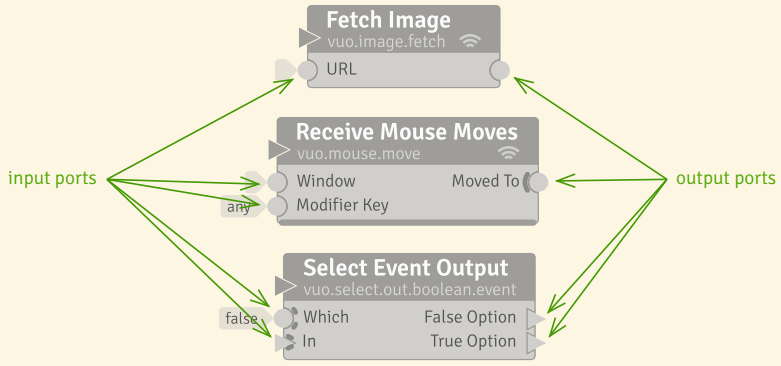
That port on the other end is called an **input port**. You can think of it as a portal that inputs (receives) information into the node.




In the above composition, the **Text** input port of the **Make Text Image** node inputs an event, which causes the node to execute. The **Image** input port of the **Render Image to Window** node inputs the event and an image. When the node executes, it uses that image to do its job of rendering an image to a window.

You may have noticed that, in the above composition, some input ports have data that's attached to the port rather than coming in through a cable. The **Text** input port has the data "Hello World!", and the **Font** input port has as its data a description of a Helvetica font. These are called **constant values** because they don't vary the way that data coming through a cable can. Like data coming in through cables, constant values are also used by the node when it executes. If a port has a constant value, you can edit it by double-clicking on it.

After a node executes, it outputs (sends) information through its **output ports**. The information outputted — events and possibly data — can then travel along cables from the output ports to other input ports.



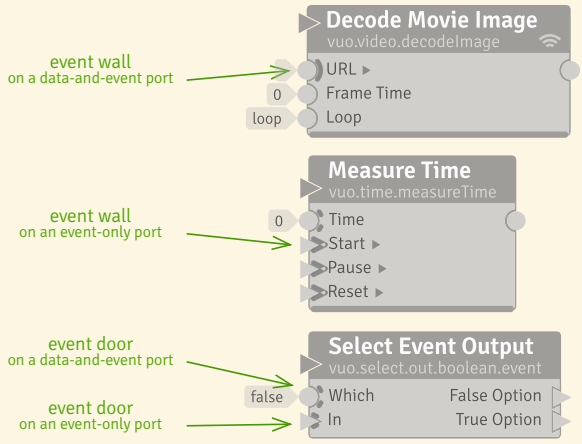

On most nodes, every event that comes in through one or more inputs ports goes out of all of the output ports. But there are a couple of exceptions.

One exception is trigger ports. Although trigger ports are output ports, events that come in through input ports are never outputted through them. Trigger ports can only fire new events, not transmit existing events.

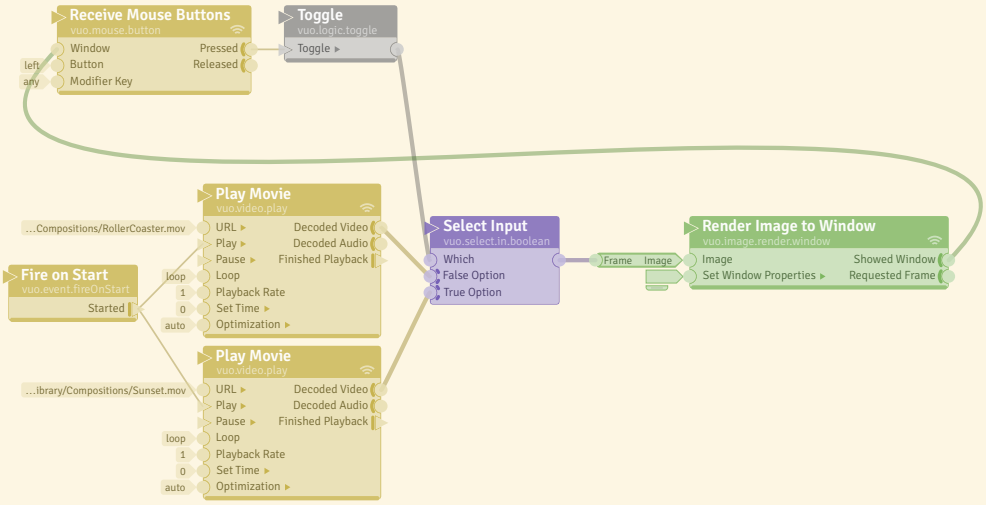
The other exception is for nodes whose input ports have thick lines along their right side, which are called **event walls** and **event doors**. If an event comes into a node only through an input port with a wall, then the event won't go out any of the node's output ports. If an event comes in only through an input port with a door, then the event may or may not go out of some or all of the node's output ports — the exact behavior depends on the node, and is explained in the node's documentation.

+ Tip

Both for trigger ports and for walls and doors, the thick line is a hint to remind you that events may be blocked.




The composition below, Select Movie (File > Open Example > vuo.select), demonstrates one way that event doors can be useful. This composition displays one of two movies at a time, switching between them each time the mouse is pressed. The doors on the **Select Input** node's **False Option** and **True Option** input ports allow the node to let the stream of events and video frames from one movie through while blocking the stream from the other movie.

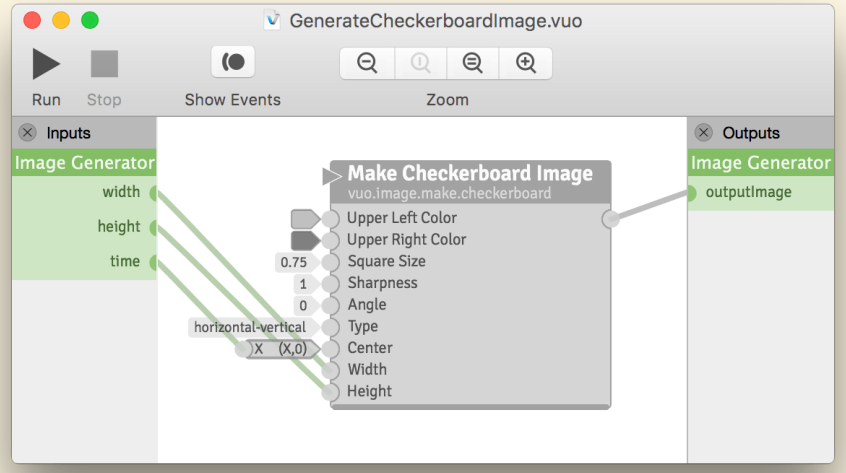



If you're not sure if a node is letting events through or blocking them, you can enable Show Events or look at Port Popovers to see where events are flowing.

## 3.7 Events and data enter and exit a composition through published ports

Earlier, you learned that a composition is made up of nodes, each of which is a building block that has a specific job to perform. If you think about it, the composition as a whole also has a specific job to perform. It's like a node, but on a larger scale. A composition can even be used as a building block within another application.

Just as a node can input and output information through its ports, a composition can input and output information through **published ports**. If a composition has published ports, the Vuo Editor shows them in sidebars along the left and right sides of the composition canvas.



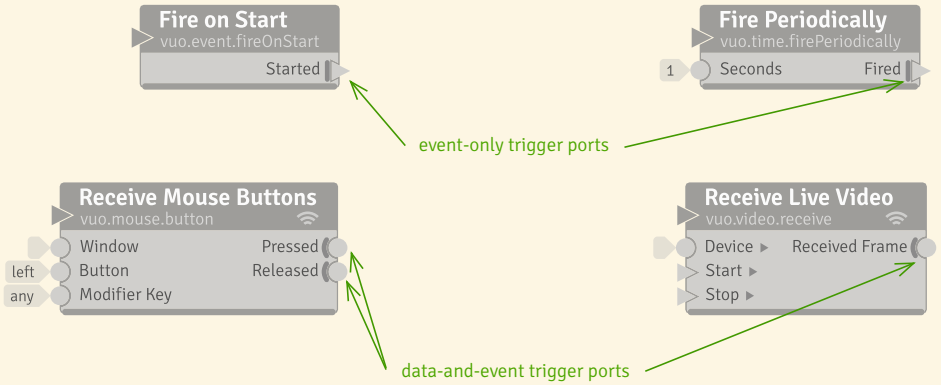

Above is an example of a composition with published ports: Generate Checkerboard Image (File > Open Example > Image Generators). It inputs events and data through published input ports called **width**, **height**, and **time**. It outputs events and data through a published output port called **outputImage**. If you run the composition in the Vuo Editor, it displays an animated checkerboard image. Because this composition has published ports, it's also possible to run it inside of other applications, besides the Vuo Editor, that accept Vuo compositions. For example, you could use this composition within a VJ application to generate a stream of images that you'd mix with other video streams.

# 4 How events and data travel through a composition

Events are what make things happen in a composition. As you get to know Vuo, you'll be able to look at a composition and imagine how an event comes out of a trigger port, flows through a cable into a node's input port, and either gets blocked or flows through the node's output ports and into the next cables. The previous section gave an overview of how that works. This section describes the process in detail.

## 4.1 Where events come from

Each event is fired from a **trigger port**, a special kind of output port on a node.




Some trigger ports fire events in response to things happening in the world outside your composition. For example, the **Receive Mouse Moves** node's trigger port fires an event each time the mouse is moved. The **Play Movie** node's trigger port fires a rapid series of events (for example, 30 per second), so that you can display images in rapid sequence as a movie. Similarly, the **Render Scene to Window** node's **Requested Frame** trigger port fires a rapid series of events, so that you can use these events to display scenes in rapid sequence as an animation.

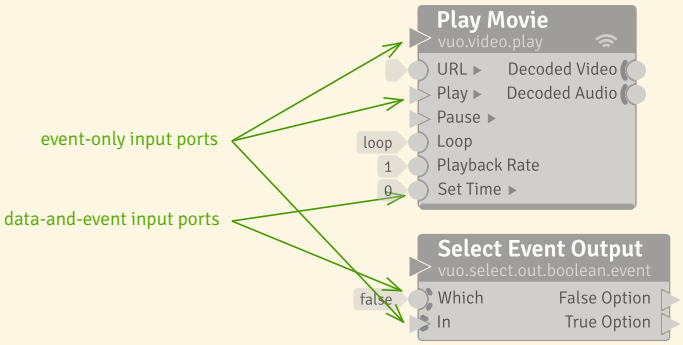
Other trigger ports fire events in response to things happening within the composition. For example, the **Fire on Start** node's trigger port fires an event when the composition starts. The **Fire Periodically** node's trigger port fires events at a steady rate while the composition is running. A node's trigger port can even fire in response to an event received by the node, as happens with the **Spin Off Event** node. (However, this is a *different* event than the one that was received by the node. For more information, see the section Run slow parts of the composition in the background.)

Some nodes block events until a certain condition is met. The node **Became True**, for example, only lets an event through when the condition changes from false to true. These nodes are not trigger nodes, since they don't create events, but they control when events are output.

## 4.2 How events travel through a node

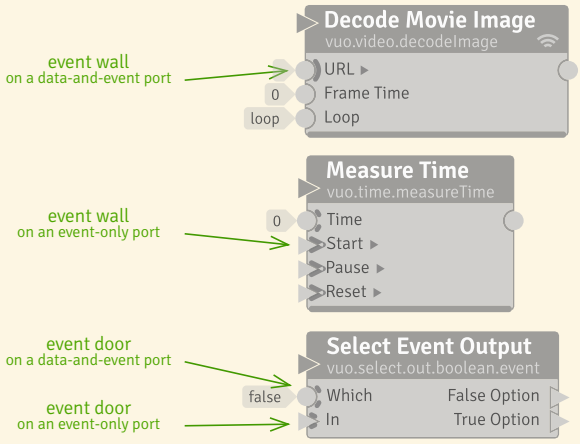
An event can come into a node through cables connected to one or more of its input ports. When an event reaches the node's input ports, the node executes, and it may or may not send the event through its output ports.

### 4.2.1 Input ports




An **input port** is the location on the left side of the node where you can enter data directly, connect a data-and-event cable, or connect an event-only cable. When an event arrives at an input port, it causes the node to execute and perform its function based on the data present at the node's input ports.

**4.2.1.1 Event walls and doors** Some nodes, like the ones shown below, have input ports that block an event. This means the node will execute, but the event associated with that data won't travel through any output ports. Event blocking is useful when you want part of your composition to execute in response to events from one trigger port but not events from another trigger port, or when you're creating a feedback loop.



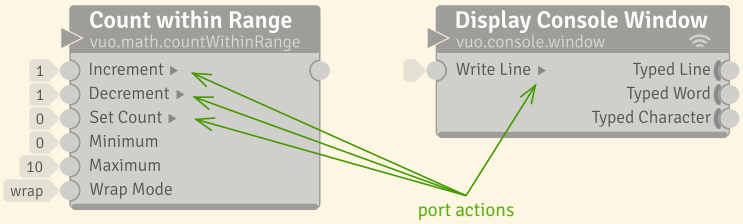

Ports that always block events have a solid semi-circle (like the **URL** port above) or a solid chevron (like the **Start** port above). This is called an **event wall**. The node must receive an event from another port without an event wall for the results of the node's execution to be available to other nodes.

Ports that sometimes block events have a broken semi-circle (like the **Which** port above) or a broken chevron (like the **Time** port above). This is called an **event door**. Event doors are useful when you want to take events from a trigger port and filter some of them out or route them to different parts of the composition. For example, in the **Select Output** node, the value at the **Which** port will determine whether the data-and-event coming into the **In** port will be transmitted to the **Option 1** port or the **Option 2** port.

The manual section How events travel through a composition has more information on how events move through a composition.

> **Tip**
>
> The event wall is visually placed inside the node to indicate that the event gets blocked inside the node (as it executes) — rather than getting blocked before it reaches the node.

**4.2.1.2 Port actions** Some input ports cause the node to do something special when they receive an event. In the **Count within Range** node shown below, the **Increment**, **Decrement**, and **Set Count** ports each uniquely affect the count stored by the node — upon receiving an event, they increment the count, decrement the count, or change the count to a specific number. Likewise, in the **Display Console Window** node, the **Write Line** input port does something special when it receives an event — it writes a line of text to the console window. Each of these ports has a *port action*.



If an input port has a **port action**, then the node does something different when that input port receives an event than it does when any other input port receives an event. What counts as "something different"? Either the node outputs different data (immediately or later) or the node affects the world outside the composition differently.

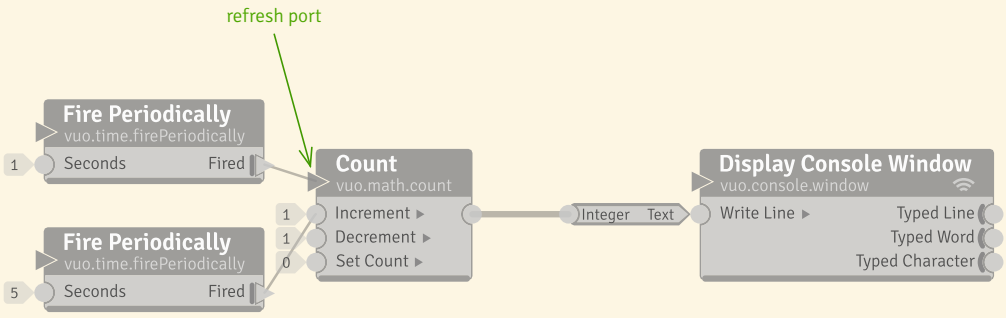
Looking again at the **Count within Range** node, you can see that the node has some input ports with port actions and some without. For the ports without port actions — **Minimum**, **Maximum**, and **Wrap Mode** — the node will output the same number regardless of whether the event causing the node to execute has hit one of these ports. The node uses the data from these ports and doesn't care if they received an event. For each of the ports with port actions, however, it makes a difference whether the event has hit the port. The **Increment** port, for example, only affects the count if the event came in through that input port.

Rather than affecting the node's output data, as in the **Count within Range** node, the **Display Console Window** node's port action affects the world outside the composition. When the **Write Line** input port receives an event, it doesn't affect the data coming out of the node's output ports. Rather, it affects what you see in the console window.

You can recognize an input port with a port action by the little triangle to the right of the port name. In Vuo, the triangle shape symbolizes events. The little triangle for the port action reminds you that this port does something unique when it receives an event.

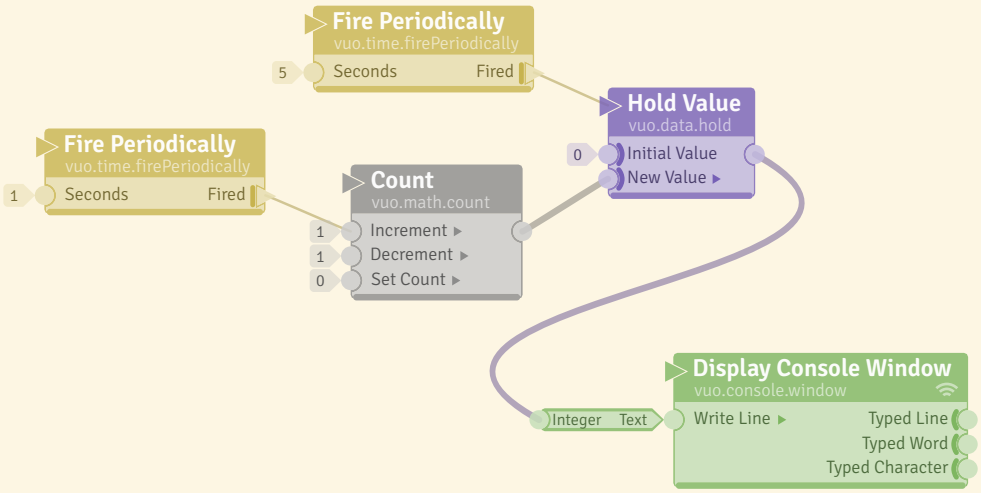
#### 4.2.1.3 Refresh ports

Every node has a built-in event-only input port called the **refresh port**. The purpose of this port is to execute the node without performing any port actions.




For example, the composition above shows how you can use the refresh port to get the **Count** node's current count without incrementing or decrementing it. When the lower **Fire Periodically** node fires an event, **Display Console Window** writes the incremented count. When the upper **Fire Periodically** node — which is connected to the **Count** node's refresh port — fires an event, the count stays the same. **Display Console Window** writes the same count as before.

On other nodes, like **Hold Value**, the refresh port is the only input port that transmits events to any output ports. The **Hold Value** node lets you store a value during one event and use it during later events. The composition below shows how you can use a **Hold Value** node to update a value every 1 second and write it every 5 seconds.




When the left **Fire Periodically** node executes, the count, as a data-and-event, is transmitted to the **Hold Value** node. The **Display Console Window** node doesn't execute because, when the event arrives at the **Hold Value** node, it is blocked.

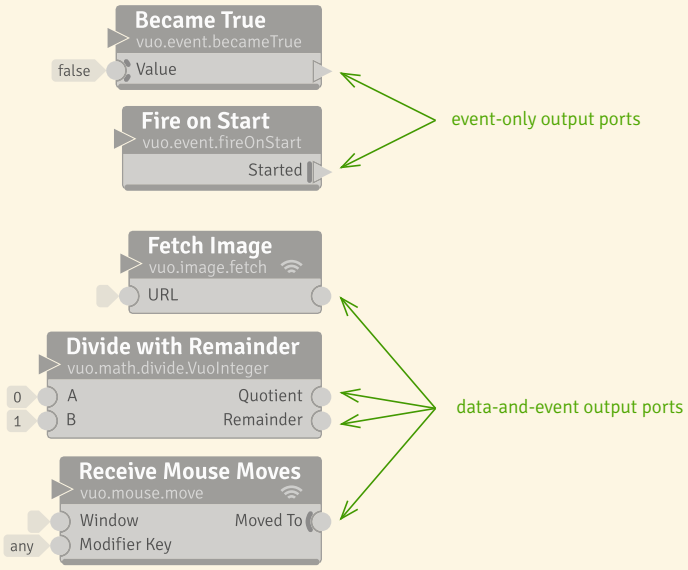
When the upper **Fire Periodically** node executes, the count stored in the **Hold Value** node travels to the **Display Console Window** node and gets written.

**+ Tip**

Refresh ports are useful for allowing event flow out of nodes that have event walls. The event entering the **Refresh port** can travel to downstream nodes, while an event encountering an event wall cannot.

### 4.2.2 Output ports

When an event executes a node, the event can travel to downstream nodes using the **output ports**. Like input ports, output ports can be data-and-event or event-only.




**4.2.2.1 Trigger ports** Although trigger ports can *create* events, they never *transmit* events that came into the node through an input port (hence the thick line to the left of each trigger port — an event wall), nor do they cause any other output ports to emit events.

## 4.3 How events travel through a composition

Now that you've seen how events travel through individual nodes, let's look at the bigger picture: how they travel through a composition.

### 4.3.1 The rules of events

Each event travels through a composition following a simple set of rules:

1. **An event travels forward through cables and nodes.** Along each cable, it travels from the output port to the input port. Within each node, it travels from the input ports to the output ports (unless it's blocked). An event never travels backward or skips around.
2. **One event can't overtake another.** If multiple events are traveling through the same cables and nodes, they stay in order.
3. **An event can split.** If there are multiple cables coming out of a trigger port or other output ports, then the event travels through each cable simultaneously.
4. **An event can rejoin.** If the event has previously split and gone down multiple paths of nodes and cables, and those paths meet with multiple cables going into one node, then the split event rejoins at that node. The node waits for all pieces of the split event to arrive before it executes.

Note for Quartz Composer users

Quartz Composer provides less control than Vuo does over when patches execute. Patches typically execute in sync with a framerate, not in response to events. Patches typically execute one at a time, unless a patch has been specially programmed to do extra work in the background.

Note for text programmers

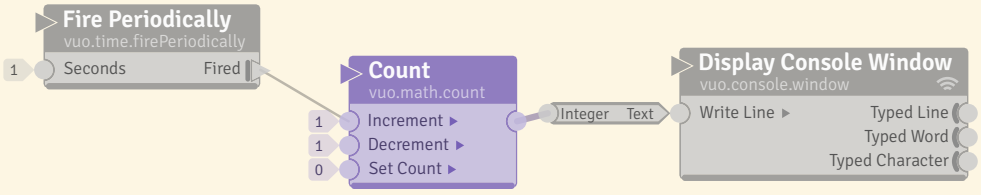
This section is about Vuo's mechanisms for control flow and concurrency.

5. **An event can be blocked.** If the event hits an event wall or door on an input port, then although it will cause the node to execute, it may not transmit through the node.
6. **An event can travel through each cable at most once.** If a composition could allow an event to travel through the same cable more than once, then the composition is not allowed to run. It has an infinite feedback loop error.

Let's look at how those those rules apply to some actual compositions.

### 4.3.2 Straight lines

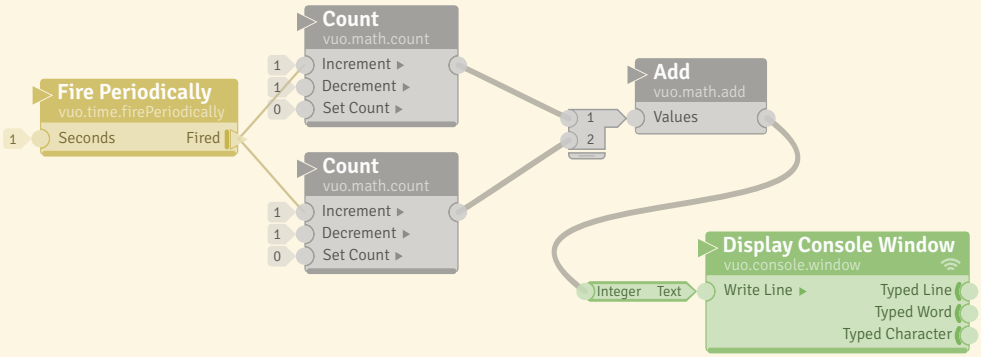
The simplest event flow in a composition is through a straight line of nodes, like the composition below.




In this composition, the **Fired** trigger port fires an event every 1 second. Each event travels along cables and through the **Count** node, then the integer-to-text type converter node, then **Display Console Window** node. The event is never split or blocked.

### 4.3.3 Splits and joins

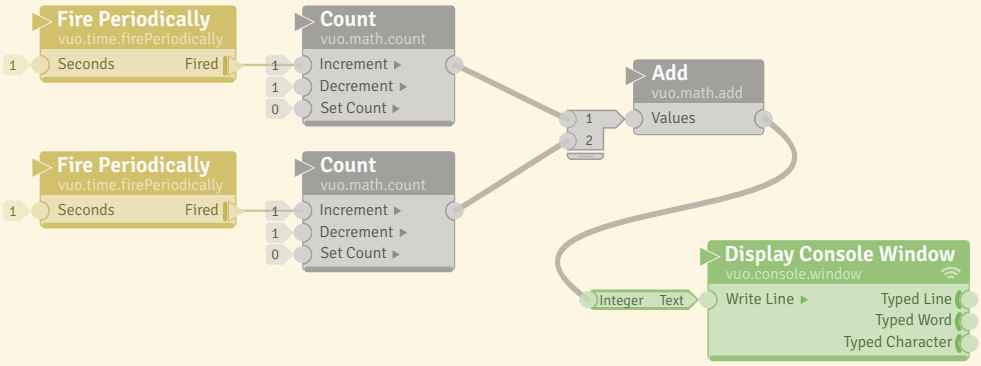
When you run a composition in Vuo, multiple nodes can execute at the same time. This takes advantage of your multicore processor to make your composition run faster.



In this composition, the two **Count** nodes are independent of each other, so it's OK for them to execute at the same time. When the **Fire Periodically** node fires 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 **Fire Periodically** fires an event. The event branches off to the **Count** nodes and joins up again at **Add**.

### 4.3.4 Multiple triggers



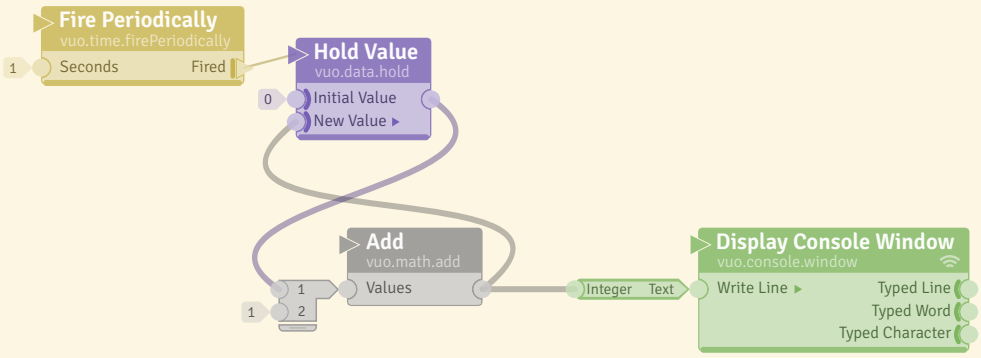

In this composition, the **Add** node executes each time either **Fire Periodically** node fires an event. If one of the **Add** node's inputs receives an event, it doesn't wait for the other input. It goes ahead and executes.

If the two **Fire Periodically** nodes fire 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 **Display Console Window** before the first event has a chance to reach **Display Console Window**.) The second event can't execute **Add** or **Display Console Window** until the first event is finished.

Compare this composition to the one above it. Since in this composition the **Fire Periodically** nodes can execute in either order, or at the same time, the results are unpredictable. When you want to ensure events are executed by separate nodes at the same time, use the *same* event.

### 4.3.5 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.



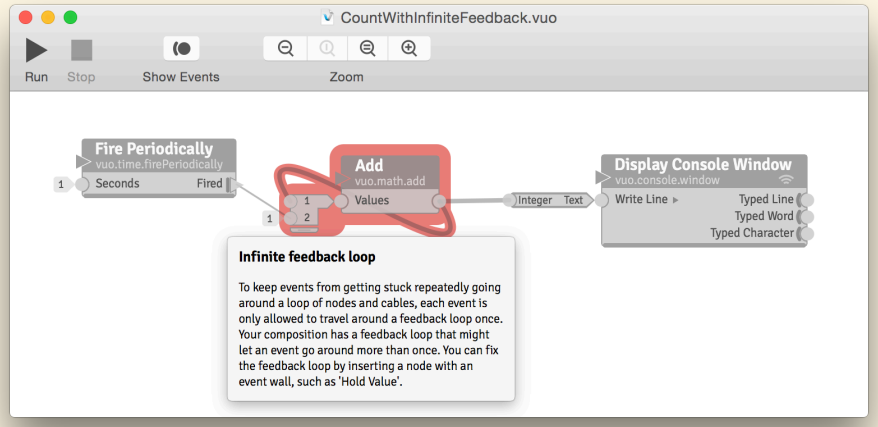

This composition uses a feedback loop to keep track of a count, which it prints upon a console window: 1, 2, 3, 4, . . .

The first time the **Fire Periodically** node fires an event, the inputs of **Add** are 0 and 1, and the output is 1. The sum, as a data-and-event, travels along the cable to the **Hold Value** node. The new value is held at the **New Value** port, and the event is blocked, as you can see from its event wall; **Hold Value** doesn't transmit events from its **New Value** port to any output ports.

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

#### 4.3.5.1 Infinite Feedback Loops

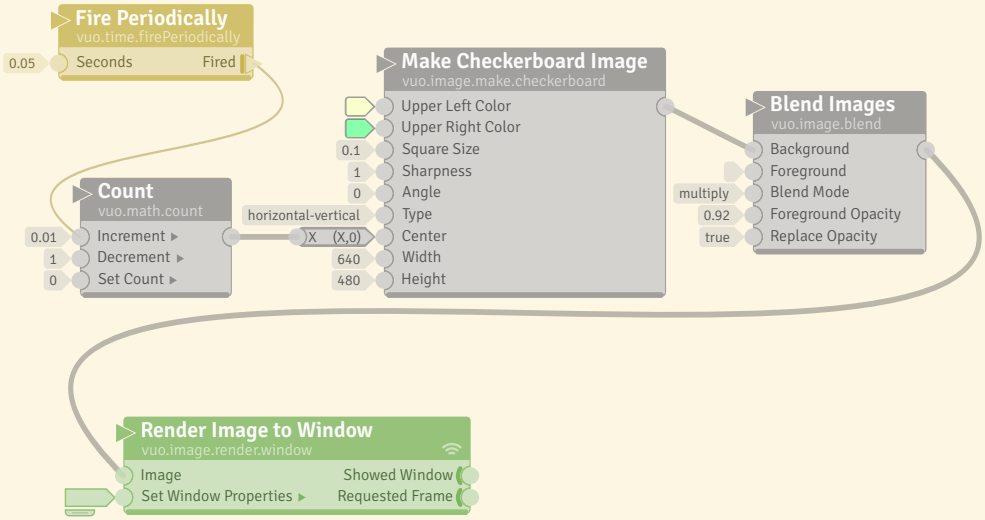
Each event is only allowed to travel once through a feedback loop. When it comes back to the first node of the feedback loop, it needs to be blocked by a walled input port. If your composition isn't set up like this, then Vuo will tell you there's an **infinite feedback loop** and won't allow your composition to run.




The above composition is an example of an infinite feedback loop. Any event from the **Fire Periodically** node would get stuck forever traveling in the feedback loop from the **Add** node's **Sum** port back to the **Item 1** port. Because there's no event wall in the feedback loop, there's nothing to stop the event. Every feedback loop needs a node like **Hold Value** to block events from looping infinitely.

If your composition has an infinite feedback loop, there are several ways you can fix it. Let's walk through a composition where you encounter a feedback loop, and look at ways to solve it.

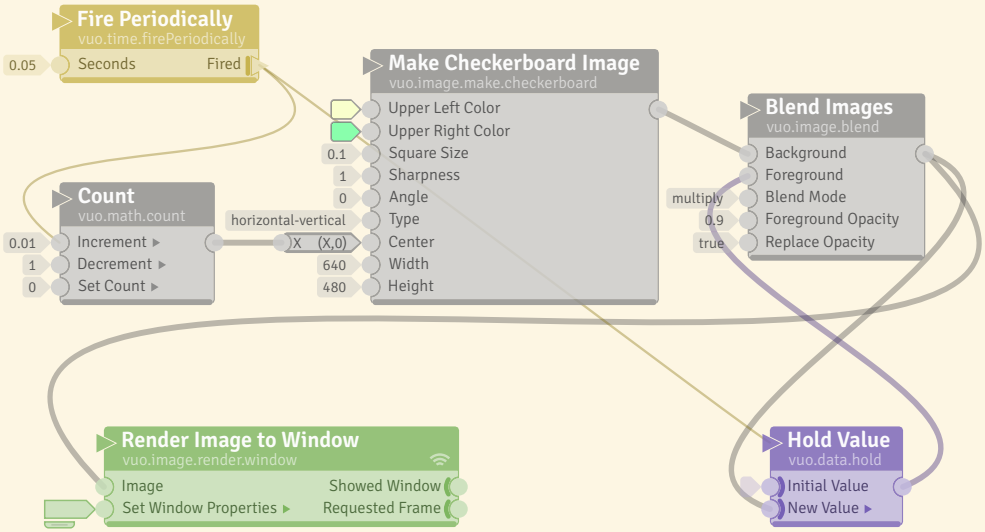
You can use Vuo's **Blend Images** node to blend two images together. This first composition creates one checkerboard image, and the image moves across the window because events from the **Fired** trigger port are incrementing the count that becomes the X value of the 2D point that is the center of the checkerboard.



But you want to see the effect of taking the produced image from the **Blend Images** node and feeding it back into the **Foreground** input port of the node. If you try that, you see that you create an infinite feedback loop, because there is no event wall on the **Foreground** input port to stop an event from entering the node a second time.

So, let's introduce a **Hold Value** and connect the output from the **Blend Images** to the **New Value** input port of the **Hold Value** node and the output from the **Hold Value** node to the **Foreground** input port of the **Blend Images** node. Now, what event source can we use? The composition has two nodes with trigger ports, the **Fire Periodically** and the **Render Image to Window**.

One strategy is to see if you can use the same trigger port that is already providing events to the node that will be part of the feedback loop. So, you can connect up the **Fired** trigger port to the refresh port of the **Hold Value** node.

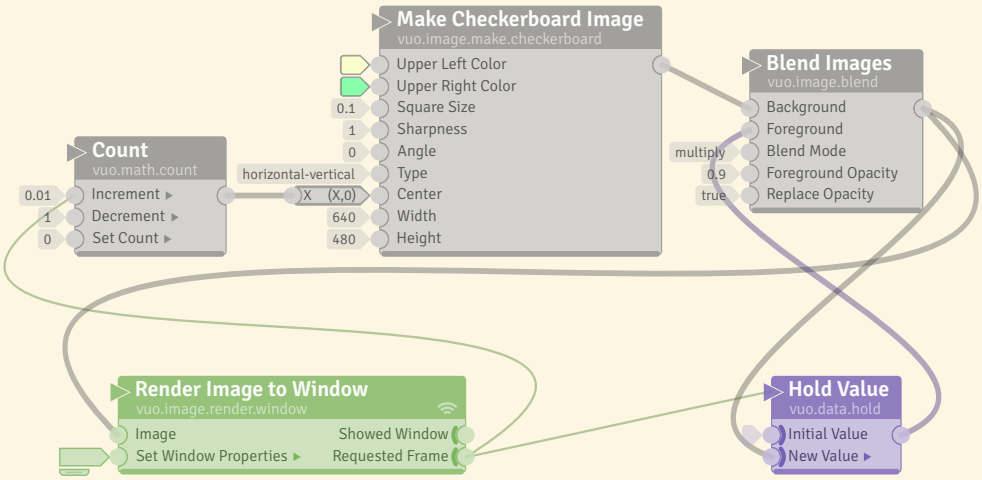



This works. An event from the **Fire Periodically** node travels through the **Count** node to the **Make Checkerboard Image** node, to the **Blend Images Node**. The same event travels through a separate cable to the **Hold Value** node, and then to the **Blend Images** node. There the two events are joined.

Notice that the first event through the **Hold Value** node will output the value present at the **Initial Value** port (which has no image present), and the second event will output the value present at the **New Value** port. It's not until the second event with its data reaches the **Blended Images** node that two images are blended. This is not important in this composition because the time between the first and second events reaching the node is small, but properly initializing your **Hold Value** nodes and understanding when the **New Value** port's data will arrive may be important in other compositions you create.

What about using the trigger port from the **Render Image to Window** node? When you look at your composition, using an event from a different trigger port may work better. So, a second strategy is to

see if you can use the trigger port of another node in your composition to provide events to your **Hold Value** node's refresh port. This composition uses the events that are generated by the **Render Image to Window** node's **Requested Frame** trigger port.

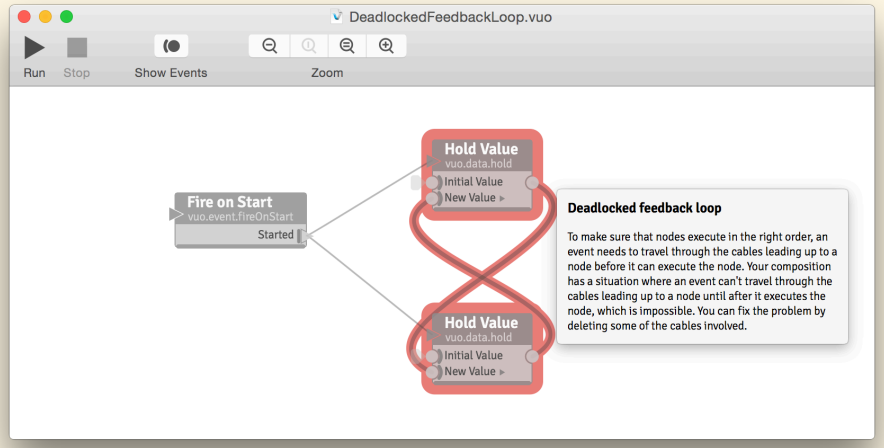



In this case, you can use the same stream of events into the **Count**'s **Increment** port, simplifying your composition. Using events from a **Rendered Frame** trigger port is usually the best way to provide events for any type of animation in your composition.

A third approach is to insert a **Spin Off Event** node into your composition to generate a separate event. This is covered in the section Run slow parts of the composition in the background.

#### 4.3.5.2 Deadlocked feedback loops

In most cases, an event needs to travel through all of the cables leading up to a node before it can reach the node itself. (The one exception is the node that starts and ends a feedback loop, since it has some cables leading into the feedback loop and some coming back around the loop.) A problem can arise if the nodes and cables in a composition are connected in a way that makes it impossible for an event to travel through all the cables leading up to a node before reaching the node itself. This problem is called a **deadlocked feedback loop**. If your composition has one, Vuo will tell you so and won't allow your composition to run.




This composition is an example of a deadlocked feedback loop. Because the top **Hold Value** node could receive an event from the **Fire on Start** node through the cable from the bottom **Hold Value** node, the top **Hold Value** node needs to execute after the bottom one. But because the bottom **Hold Value** node could receive an event from the **Fire on Start** node through the cable from the top **Hold Value** node, the bottom **Hold Value** node needs to execute after the top one. Since each **Hold Value** node needs to execute after the other one, it's impossible for an event to travel through the composition. To fix a deadlocked feedback loop, you need to remove some of the nodes or cables involved.

### 4.3.6 Summary

You can control how your composition executes by controlling the flow of events. The way that you connect nodes with cables — whether in a straight line, a feedback loop, or branching off in different directions — controls the order in which nodes execute. The way that you fire and block events — with trigger ports and with event walls and doors — controls when different parts of your composition will execute.

Each event that's fired from a trigger port has its own unique identity. The event can branch off along different paths, and those paths can join up again at a node downstream. When the *same* event joins up, the joining node will wait for the event to travel along all incoming paths and then execute just once. But if two *different* events come into a node, the node will execute twice. So if you want to make sure that different parts of your composition are exactly in sync, make sure they're using the same event.

## 4.4 Controlling the buildup of events

What if a trigger port is firing events faster than the downstream nodes can process them? Will the events get queued up and wait until the downstream nodes are ready (causing the composition to lag), or will the composition skip some events so that it can keep up? That depends on the trigger port's **event throttling** setting.

Each trigger port has two options for event throttling: **enqueue events** or **drop events**. If enqueuing events, the trigger port will keep firing events regardless of whether the downstream nodes can keep up. If dropping events, the trigger port won't fire an event if the event would have to wait for the downstream nodes to finish processing a previous event (from this or another trigger port).

Each of these options is useful in different situations. For example, suppose you have a composition in which a **Play Movie** node fires events with image data and then applies a series of image filters. If you want the composition to display the resulting images in real-time, then you'd probably want the **Play Movie** node's trigger port to drop events to ensure that the movie plays at its original speed. On the other hand, if you're using the composition to apply a video post-processing effect and save the resulting images to file, then you'd probably want the trigger port to enqueue events.

When you add a node to a composition, each of its trigger ports may default to either enqueuing or dropping events. For example, the **Play Movie** node's trigger port defaults to dropping events, while each of the **Receive Mouse Clicks** node's trigger ports defaults to enqueuing events.

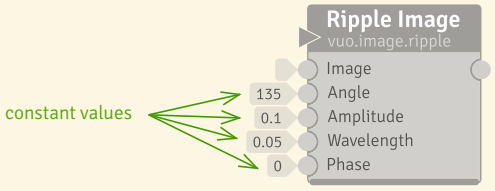
In the Vuo Editor, you can right-click on a trigger port and go to the `Set Event Throttling` menu to view or change whether the port enqueues or drops events.

# 5 How compositions process data

## 5.1 Where data comes from

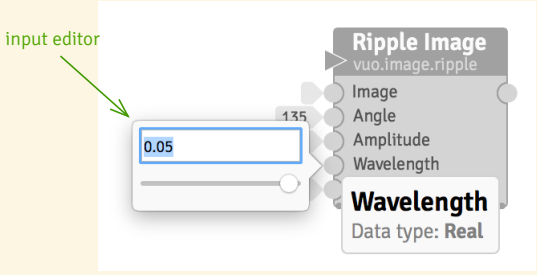
Over time, data in your composition can remain the same (be constant), or change as data flows through **data-and-event** cables. In some cases, data can come from outside your composition using **interface** nodes.

### 5.1.1 Constant input port values




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.

For many types of data (such as integers and text), you can edit a constant value by double-clicking on the constant value attached to an input port. This will pop up an **input editor** that lets you change the constant value. (If nothing happens when you double-click on the constant value, then the Vuo Editor doesn't have an input editor for that data type. To change the data for that input port, you have to connect a cable.)



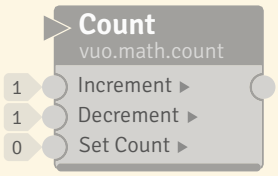

When using an input editor to edit text, you can enter multiple lines by using ⌥↵ (instead of ↵) to insert a line break.

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

### 5.1.2 Default input port values

When you add a node to a composition, each input port has a preset constant value called 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. The port stays at its default value until it receives an event.

If you disconnect a data-and-event cable to a port that previously had a constant value, then the port retains the last value it received from the cable. If you did not set a constant value, it goes back to its default value.



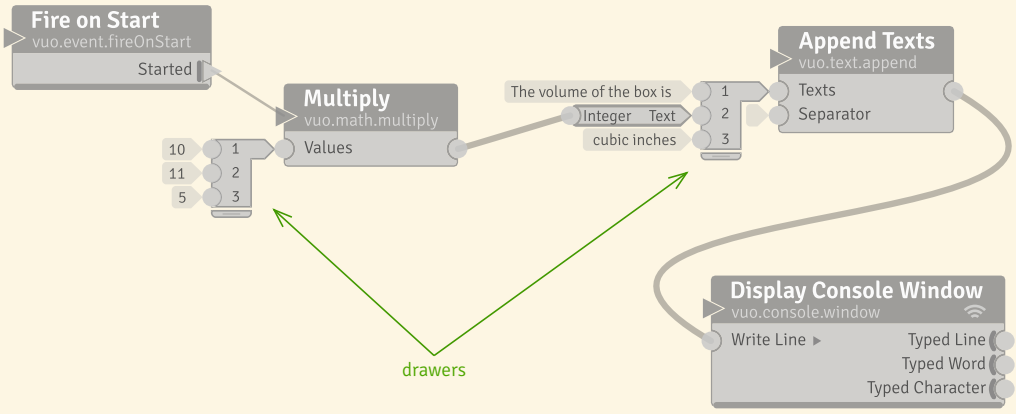

### 5.1.3 Drawers

Some input ports accept a list of values rather than a single value. If no cable is connected, a list input port has a special attachment called a **drawer** that lets you input each item of the list separately. For each input port within the drawer, you can either connect a cable to it or set its constant value.

In most cases, you can change the number of items in the drawer. (The one exception is the drawer on the **Calculate** node's **Values** input port, which changes automatically when you edit the **Expression** input port's constant value.) To change the number of items in a drawer, you can either drag on its handle (the bar along the bottom of the drawer) or right-click on the drawer and select `Add Input Port` or `Remove Input Port`.

In the following composition, which calculates the volume of a box, the dr **Multiply** node and the **Append Texts** node have drawers.

Below is an example of a composition that uses drawers. It calculates the volume of a 10 x 11 x 5 box. The height, length, and width of the box are listed in the drawer on the **Multiply** node. The pieces of text that are combined to form the console output go into the drawer of the **Append Texts** node.

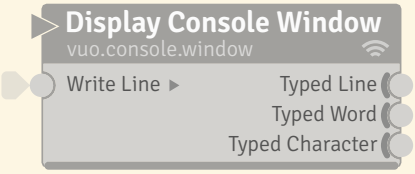



When you edit a drawer's constant values or change the number of drawer items, this affects the attached list input port in the same way as if you had edited a constant value on the list input port itself. For example, in the composition above, the **Multiply** node's **Values** input port contains the list *10, 11, 5*. When the **Multiply** node receives an event from **Fire on Start**, it calculates the product of those numbers, *550*, and sends that through its output cable.

### 5.1.4 Interface nodes

Some nodes interact with the world outside the composition — such as windows, files, networks, and devices.

Nodes that bring information into the composition from the outside world and/or affect the outside world are called **interface** nodes. An example of an interface node is the **Fetch Image** node, which can download an image from the Internet. Another example is the **Display Console Window** node, which reads text typed in a console window and writes text to that window.


Interface nodes have three radio arcs in the top-right corner, symbolizing that they send or receive data with the world outside your composition.

### 5.1.5 Dragging files onto the canvas to create an interface node

In Vuo, dragging an audio, video, image, scene, or projection mesh file onto the canvas will create the appropriate interface node with that file as input. For example, when you drag an image onto the canvas, the Editor will create a **Fetch Image** node with the file path entered in the **URL** input port. The default will be the file's relative path. To enter an absolute path, hold down the option key when dragging the file on the canvas.

Note for Quartz Composer users

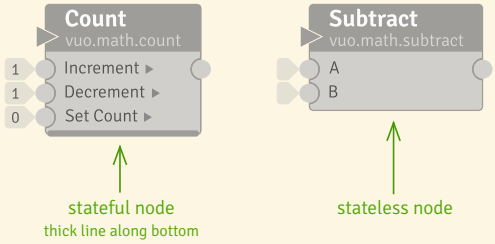
Instead of Vuo's interface and non-interface nodes, Quartz Composer has an execution mode for each patch: provider, consumer, or processor. A patch's execution mode not only indicates how it interacts with the outside world, but also controls when it executes and whether it can be embedded in macro patches.

## 5.2 How data travels through a composition

Data can't travel to another node by itself. Data travels along with events through **data-and-event** cables. If multiple data-and-event cables are connected to an output port, the data and events travel along all the attached cables.

## 5.3 How data is stored within a node

Data stays in a node input port until it is replaced by other data. If you detach a cable while the composition is running, the input port will keep the latest data that came through that cable.




Some nodes have **state**. A **stateful node** retains information from the last time it executed. An example of a stateful node is **Count**. If a **Count** node is told to increment, its state (the retained count) changes.

Stateful nodes have a thick bar along the bottom that resembles a **data-and-event cable**. This symbolizes that the node's state data is kept after the node executes, and used next time it executes.

A **stateless node** doesn't retain any information about previous times it executed. If you give it the same inputs, it'll always give you the same outputs. Stateless nodes have a thin bottom border.

## 5.4 Port data types

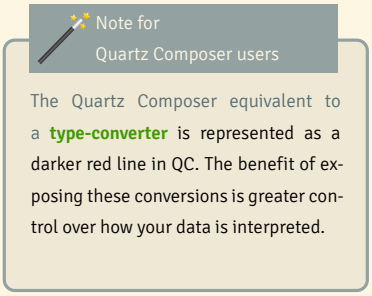
Nodes are sensitive to **data type**. Vuo works with various types of data, such as integer (whole) numbers, real (decimal) numbers, Boolean (logical) values, text, 3D points, images, and more.

For example, the **Count Characters** node's **Text** input port has data type text, and its **Character Count** output port has data type integer. If you see a **Count Characters** node that has a cable connected to its **Character Count** port, then you can know that the port at the other end of the cable must also have type integer.

### 5.4.1 Type-converter nodes

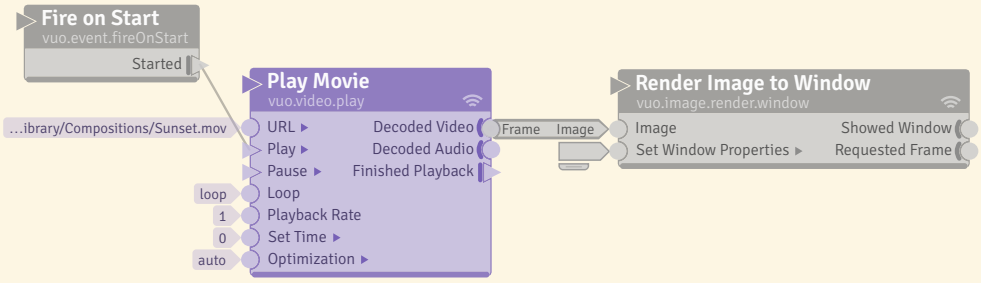
What if you want to connect the **Count Character** node's integer output port to a text input port? If you start to drag a cable from the **Character Count** port, any ports in the composition that the cable can connect to are highlighted. This includes not just ports of type integer, but also ports of type text. If you drag the cable to a text input port and release the mouse, then a type converter node will be automatically added to convert the integer data to text data.

A **type converter node** is just a node that happens to have a single input port of one data type and a single output port of another data type. Whenever you want to connect a port of one data type to a port of another data type, you can try drawing the cable to find out if any type converters can be inserted automatically. Or you can search the Node Library using the term "convert" for a node that has an input port of the type that you want to convert from to an output port of the type that you want to convert to. When you drag a convert node onto the canvas and connect it up, it will collapse to only show the original data type name and the data type the data is converted to.




For some types, there's more than one way to do the conversion. For example, if you want to convert from a real number to an integer, you can round the real number to the nearest integer, round it down to the integer below, or round it up to the integer above. If you try to connect a cable from a real-number output port to an integer input port, a menu will pop up allowing you to pick the conversion you want to use.

The composition below, Play Movie (File > Open Example > vuo.video), shows a type converter from a movie frame to an image.




### 5.4.2 Generic port types

Some nodes can work with many different types of data. For example, a **Hold Value** node could hold an integer, an image, a 3D point or really, any type of data. Rather than cluttering up the Node Library with a separate **Hold Value** node for each data type, Vuo lists a single **Hold Value** node that can be made to work with any type.

When you drag a **Hold Value** node from the Node Library onto the canvas, each of the node's ports has a **generic data type**. This means that the data type for these ports hasn't been decided yet. Right now, each of the **Hold Value** node's ports has the potential to connect to any type of port in the composition. You can see that a port is generic by hovering over it with the mouse and reading the port popover that pops up.

When you connect one of the **Hold Value** node's ports to a non-generic port, then all of the **Hold Value** node's generic ports change to non-generic ports with the same type as the connected port. For example, if you connect the **Hold Value** node's **Held Value** port to a **Count Character** node's **Text** port, then all of the **Hold Value** node's generic ports change to text ports, matching the **Text** port.

Another way to change a generic port to a non-generic port is to right-click on the port, go to the `Set Data Type` menu that pops up, and choose a data type.

If you want to change the port back to generic, you can right-click on the port and select the `Revert to Generic Data Type` menu item. (This will delete any cables between non-generic ports and ports changed back to generic.)

Some generic nodes can work with several different types of data, but not all types. For example, the **Add** node can work with with integers, real numbers, 2D points, 3D points, and 4D points, but not text or images. Some generic nodes are automatically turned into non-generic nodes when first created. For example, when you drag an **Add** node from the Node Library onto the canvas, its ports are automatically changed from generic to real numbers, because real numbers are usually a suitable choice for the **Add** node. But if you want to work with integers instead, you can use the `Revert to Generic Data Type` command and then the `Set Data Type` menu to change the ports to integers.

When you drag an **Measure Distance between Points** node from the Node Library onto the canvas, its **A** port is generic, but it can only connect to a port of type 2D point, 3D point, or 4D point. You can see the list of compatible types for a generic port by looking at the port popover.

You can connect a generic port to another generic port, as long as they work with compatible types of data. For example, you can connect a **Hold Value** node to an **Add** node, since the **Hold Value** node is compatible with any type. However, you can't connect the output of a **Find Maximum** node to a **Measure Distance between Points** node, since the **Find Maximum** node is only compatible with integers and reals while the **Measure Distance between Points** node is only compatible with 2D, 3D, and 4D points.

A generic node can have ports that each work with a different generic type. For example, the **Get Message Values** node in the **vuo.osc** node set has several output ports with independent generic types. You can change one output port's generic type to integer and another output port's generic type to text, for example. You can see if a node's generic ports use the same generic type or different generic types by looking at the port popover. The port popover displays a name for the port's generic type, such as "generic #1" or "generic #2". If two generic ports have the same name for their type (including if they're on separate nodes connected by cables), then changing the generic type of one port will also change the other port. If two generic ports have different names for their type, then changing one will not affect the other.

## 5.5 Making a composition input and output specific data using protocols

A **protocol** is a predetermined set of published ports with certain names and data types. Vuo supports the following protocols:

- **Image Filter** — Alters an image.
  - Published input ports:
    * **image** (Image) — The original image.
    * **time** (Real) — A number that changes over time, used to control animations or other changing effects. When previewing the composition in the Vuo Editor, this number is the time, in seconds, since the composition started running. In other applications, this number should be described in their instructions for creating Vuo compositions.
  - Published output ports:
    * **outputImage** — The altered image.
- **Image Generator** — Creates an image.
  - Published input ports:
    * **width** (Integer) — The requested width of the image, in pixels.
    * **height** (Integer) — The requested height of the image, in pixels.
    * **time** (Real) — A number that changes over time, used to control animations or other changing effects. When previewing the composition in the Vuo Editor, this number is the time, in seconds, since the composition started running. In other applications, this number should be described in their instructions for creating Vuo compositions.
    * **offlineRender** (Boolean) — Optional. In the Vuo Editor and the `vuo-export` command-line tool, this port's value is *true* if the composition is being exported to a movie and *false* otherwise.
    * **offlineFramerate** (Real) — Optional. In the Vuo Editor and the `vuo-export` command-line tool, this port's value is the movie framerate, in frames per second, if the composition is being exported to a movie and 0 otherwise.
    * **offlineMotionBlur** (Integer) — Optional. In the Vuo Editor and the `vuo-export` command-line tool, this port's value is the number of frames rendered per output frame. 1 means motion blur is disabled; 2, 4, 8, 16, 32, or 64 means motion blur is enabled.
  - Published output ports:
    * **outputImage** — The created image. Its width and height should match the **width** and **height** published input ports.

To create a composition that conforms to a protocol, go to File > New Composition from Template. If you've already started working on a composition and want to make it conform to a protocol, go to Edit > Protocols and choose a protocol.

Optional published ports (**offlineRender** and **offlineFramerate**) don't automatically appear in the Vuo Editor, but you can add them just as you would other published ports. Be sure to type the names exactly as they appear in this documentation (case-sensitive).

The Vuo Editor provides some examples of protocol-compliant compositions under File > Open Example > Image Generators.

When you run a protocol-compliant composition in the Vuo Editor, events and data are automatically fed into its input ports, and the resulting image is displayed in a window. This makes it easy to preview how your composition will look when run inside some other application.

When you run an Image Filter composition in the Vuo Editor, you can change the image being filtered by dropping an image file onto the running composition's window.

# 6 How nodes can be used as building blocks

Nodes are the building blocks of Vuo, which you can assemble in any way you can think of to create compositions. When you download Vuo, it comes with a large set of nodes that support 2D and 3D graphics, video, audio, networking, user interaction, and more. If you've purchased Vuo Pro, then you have some bonus "pro nodes" available to you. Whether you're using Vuo or Vuo Pro, you can also download nodes by third-party developers to add to your collection.

## 6.1 Finding out what nodes are available

The Vuo Editor has a list of all nodes called the Node Library. (If you don't see it, go to View > Show Node Library.) You can skim through the Node Library to see what's available, or you can search by node title, port name, or keyword. For example, if you're wondering if Vuo has nodes for working with hues, search for "hue" and you'll find several nodes related to color.

For a complete list of built-in nodes, you can go to the online node documentation.

If you're interested in third-party nodes, developers share them among the Vuo community.

## 6.2 Learning how to use a node

Each Vuo node has documentation, or in other words, a description of how it works. You can view this description in the Node Documentation Panel (lower panel of the Node Library) after clicking on the node in the Node Library or on the composition canvas.

Besides the documentation for individual nodes, there's also documentation for node sets. At the top of the Node Documentation Panel, most nodes have a link to their node set's documentation. For example, the **Make 3D Object** (vuo.scene.make) node has a link for vuo.scene, which provides documentation that applies to nodes throughout the vuo.scene node set.

Documentation both for nodes and for node sets is available in the online node documentation.

Besides documentation, many nodes also come with example compositions, which demonstrate use of the node within a composition. For nodes that have them, the example compositions are listed near the bottom of the Node Documentation Panel.

## 6.3 Pro vs. non-pro nodes

**Pro nodes** are only available in Vuo Pro, not in the regular Vuo or the free trial. If a node is pro, then the Node Documentation Panel says so at the bottom. If you try to open a composition containing pro nodes using non-Pro Vuo, then you'll be warned that you won't be able to run the composition.

If you plan to share a composition that contains pro nodes, keep in mind that Vuo users without Vuo Pro can't run the composition. If you want others to be able to use your composition even if they don't have Vuo Pro, consider exporting it to an app.

Pro nodes can be used when running compositions in another application, as long as Vuo Pro has been activated on the computer running the application.

## 6.4 The built-in nodes

This section gives an overview of some of Vuo's built-in nodes. The purpose is to give you a sense of what you can accomplish with the built-in nodes and where to start. For more details, see the node and node set documentation.

### 6.4.1 Graphics/video

Vuo comes with many different nodes for working with graphics. These can be roughly divided into 2D and 3D graphics (along with some nodes to convert between them).

For **2D designs and animations**, the `vuo.image` and `vuo.layer` node sets are your starting point. These let you arrange and manipulate shapes and images, and render them in a window or composite image.

For **3D models and meshes**, the `vuo.scene` node set is your starting point. It lets you load or build 3D objects, warp them, and arrange them within a scene, which you can render in a window or image. When building 3D objects, two additional node sets are helpful: `vuo.transform` for positioning, rotating, and scaling an object, and `vuo.shader` for painting a pattern or material on an object.

When working with 2D or 3D animations, the `vuo.motion` node set lets you control the **path and speed of a moving object**.

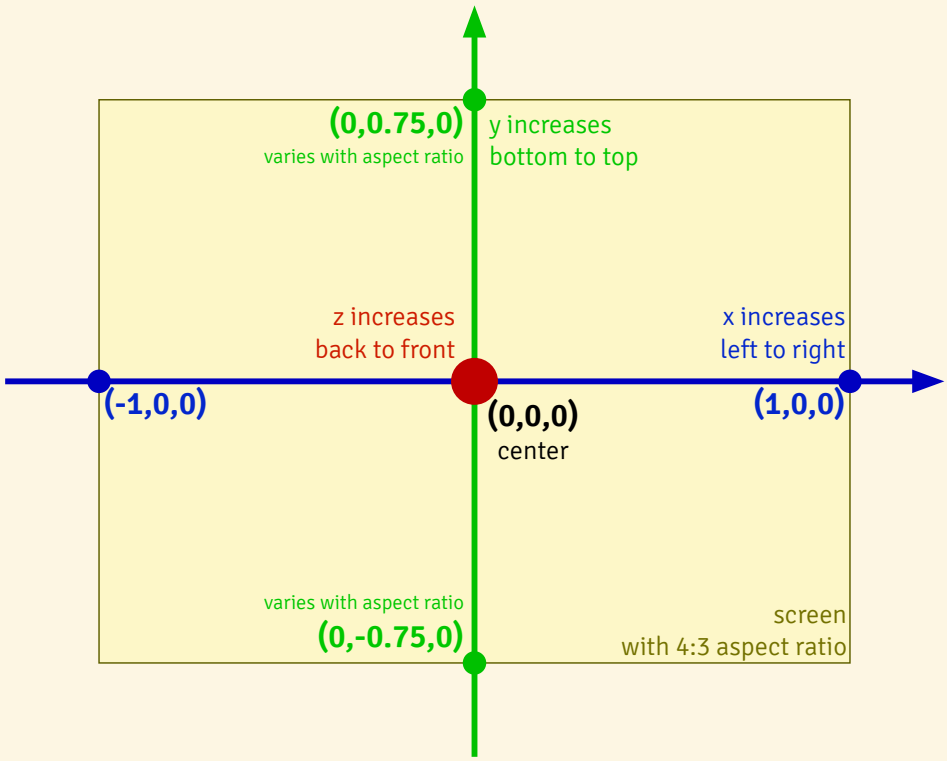
For **video**, the `vuo.video` node set handles playing movies and receiving video from cameras. When working specifically with the cameras on a Kinect, you can use the `vuo.kinect` node set. If you want to send and receive video between Vuo compositions and other applications, there's the `vuo.syphon` node set.

**Make Quad Layer** and related nodes in the `vuo.layer` node set support **projection mapping**.

#### 6.4.1.1 Vuo Coordinates

When drawing graphics to a window or image, you need to understand the **coordinate system** of the area you're drawing to. For example, when you use the **Render Scene to Window** node to display a 3D scene in a window, typically the point in your 3D scene with coordinates (0,0,0) will be drawn at the center of the window. (If you're not familiar with the concept of 2D and 3D coordinates, see http://simple.wikipedia.org/wiki/Cartesian_coordinate_system and other references to learn more.)

All of the built-in nodes that work with graphics use **Vuo Coordinates**:




Typically, as illustrated above, the position (0,0) for 2D graphics or (0,0,0) for 3D graphics is at the center of the rendering area. The X-coordinate -1 is along the left edge of the rendering area, and the X-coordinate 1 is along the right edge. The rendering area's height depends on the aspect ratio of the graphics being rendered, with the Y-coordinate increasing from bottom to top. In 3D graphics, the Z-coordinate increases from back to front.

When working with 3D graphics, you can change the center and bounds of the rendering area by using a **Make Perspective Camera** or **Make Orthogonal Camera** node. For example, you can use a camera to zoom out, so that the rendering area shows a larger range of X- and Y-coordinates.

### 6.4.2 Sound/audio

The `vuo.audio` node set lets you work with audio input and output. You can use audio input to create music visualizations or control a composition with sound. You can use audio output to synthesize

sounds. Together, audio input and output can be used to receive a live audio feed, process the audio, and play it aloud.

### 6.4.3 User input devices

There are many built-in nodes you can use to make your compositions interactive, including:

- `vuo.mouse` for getting input from a mouse or trackpad
- `vuo.keyboard` for getting input from keys typed or pressed
- `vuo.hid` for getting input from a USB Human Interface Device (HID)
- `vuo.leap` for controlling a composition with hand and finger movements from a Leap Motion device
- `vuo.osc` for remotely controlling a composition via a TouchOSC interface on a phone or tablet
- the **Filter Skeleton** node for getting input from Delicode NI mate 2

### 6.4.4 Music and stage equipment

Your compositions can control and be controlled by music and stage equipment — such as keyboards, synthesizers, sequencers, and lighting — using several common protocols:

- `vuo.osc` for receiving OSC messages
- `vuo.midi` for sending and receiving MIDI events
- `vuo.artnet` (pro) for sending and receiving Art-Net messages

The `vuo.bcf2000` nodes interface with the Behringer BCF2000 MIDI controller.

### 6.4.5 Applications

Applications that send or receive messages via the OSC, MIDI, or Art-Net protocol can communicate with your composition if you use the `vuo.osc`, `vuo.midi`, or `vuo.artnet` nodes.

Your composition can send video to and receive video from other applications via Syphon using the `vuo.syphon` node set.

With the `vuo.app` node set, your composition can launch other apps and open documents in them.

### 6.4.6 Sensors, LEDs, and motors

The `vuo.serial` nodes allow your composition to connect to serial devices, including programmable microcontrollers like Arduino. Via the Arduino, your composition can receive data from sensors, and send data to control LEDs and motors.

### 6.4.7 Displays

Two node sets let you fine-tune how a composition's windows are displayed on the available screens. The `vuo.screen` node set provides information about the available screens. The `vuo.window` node set controls how each window is displayed, including its aspect ratio and whether it's fullscreen.

### 6.4.8 Files

Your composition can open files on your computer's filesystem or download them from the internet using "fetch" nodes, such as **Fetch Image**, **Fetch Data**, and **Fetch XML Tree**.

Your composition can save files to your computer's filesystem using "save" nodes, such as **Save Image**, **Save Data**, and **Save Images to Movie**.

For opening, manipulating, and saving XML and JSON files, there's the `vuo.tree` node set. And for CSV and TSV files, there's the `vuo.table` node set.

The `vuo.file` nodes enable your composition to interact with your computer's filesystem.

### 6.4.9 Internet

With the `vuo.rss` nodes, your composition can download RSS feeds.

To retrieve data from an XML or JSON web service, you can use the `vuo.tree` nodes.

## 6.5 Adding nodes to your Node Library

You can expand the things that Vuo can do by adding nodes to your Node Library.

### 6.5.1 Creating a subcomposition

One way to add a new node to your Node Library is by building a composition out of existing nodes and using File > Save Composition to Node Library or File > Move Composition to Node Library. For an explanation of subcompositions, see Using subcompositions inside of other compositions.

### 6.5.2 Creating a text node

Another option is to implement a new node with text programming, using Vuo's API. See Developing Node Classes and Types for Vuo.

### 6.5.3 Installing a node

Other people in the Vuo community have created nodes that you can download and install. Once you've downloaded a subcomposition file (`.vuo`) or node file (`.vuonode`), you can install it with the following steps.

First, place the file in one of Vuo's Modules folders. Subcomposition files can only be installed in the User Modules folder, while node files can be installed in either the User Modules or the System Modules folder. You can get to these folders via menu items in the Editor:

- Tools > Open User Modules Folder
  - Or, if you'd rather find the folder yourself: In your home folder, go to 🗀 `Library` ▸ `Application Support` ▸ `Vuo` ▸ `Modules`. (On macOS 10.7 and above, the 🗀 `Library` folder is hidden by default. To find it, go to Finder, then hold down the Option key and select the Go > Library menu option.)
- Tools > Open System Modules Folder
  - Or, if you'd rather find the folder yourself: In the top-level folder on your hard drive, go to 🗀 `Library` ▸ `Application Support` ▸ `Vuo` ▸ `Modules`.

For node files, you'll typically want the User Modules folder, since yours will be the only user account on your computer that needs access to the node. Use the second option only if you have administrative access and you want all users on the computer to have access to the node.

Second, restart the Vuo Editor. The node should now show up in your Node Library.

# 7 Putting it all together in a composition

## 7.1 Designing a composition

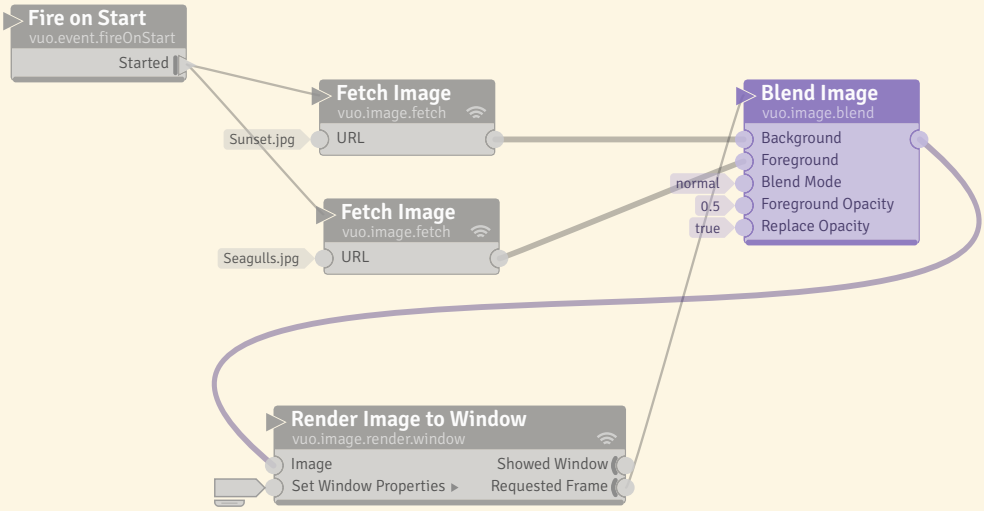
### 7.1.1 Starting new

When you create a composition, there is usually a problem you want to solve. Let's use an example to demonstrate this approach. How would I create a composite image made up of two images blended together?

1. First, you know you'll need the two images, so drag the two files onto the canvas. Vuo will create two **Fetch Image** nodes, with the image locations filled in.
2. Search the Node Library for "composite image," and drag **Blend Images** onto canvas. Alternately, you could search the Node Library for "image" and see a list of all the nodes that manipulate images, and pick from that list.
3. Look at the Node Description Panel for the **Blend Images** node, and see that it expects the two images, one for foreground and one for background.
4. Drag a cable from one **Fetch Image** node's output to the **Foreground** input port and the other to the **Background** input port.
5. Confirm by looking at the port popover for **Blend Images**'s output port (by left-clicking on the output port) to see that its type is Image.
6. Search the Node Library for "display image," or, if you remember the quick start example, search for "window" or "render." Drag the **Render Image to Window** onto the canvas.
7. Start dragging a cable from **Blend Images**'s output port and drop the cable onto the **Image** port of the **Render Image to Window** node.
8. Run the composition — nothing shows up on screen. Why?
9. At this point you could check the port popover for the **Blend Images**'s output port. It says "Last event: (none observed)." That is the same message when tracing back to the **Fetch Image** input ports.
10. Think about what trigger ports you can use to fire an event. **Fire on Start** fires once when the composition starts, which should be good for loading images. Connect up the **Fire on Start** node's **Started** port to the refresh port (top left triangle) on the **Fetch Image** nodes.
11. Stop the composition and run it again. Now the images show up, blended.
12. Yet, if you change **Foreground Opacity**, the change doesn't affect the composition. You can either check the port popovers again to see that the most recent event was before you changed the port value, or you could remember that Vuo needs events to carry changes downstream.

13. Look again at trigger ports available in composition and the corresponding node descriptions. The **Requested Frame** output port of **Render Image to Window** fires around 60 times per second (or whatever your display refresh rate is set to), so it will update the blended image in real time as you experiment with opacities and blending modes.
14. Drag a cable from the **Requested Frame** output port to **Blend Images**'s (highlighted) refresh port.
15. Use the input editors on **Foreground Opacity** and **Blend Mode** to control the blending.
16. Now you have a blended image.

Your final composition should look like this:

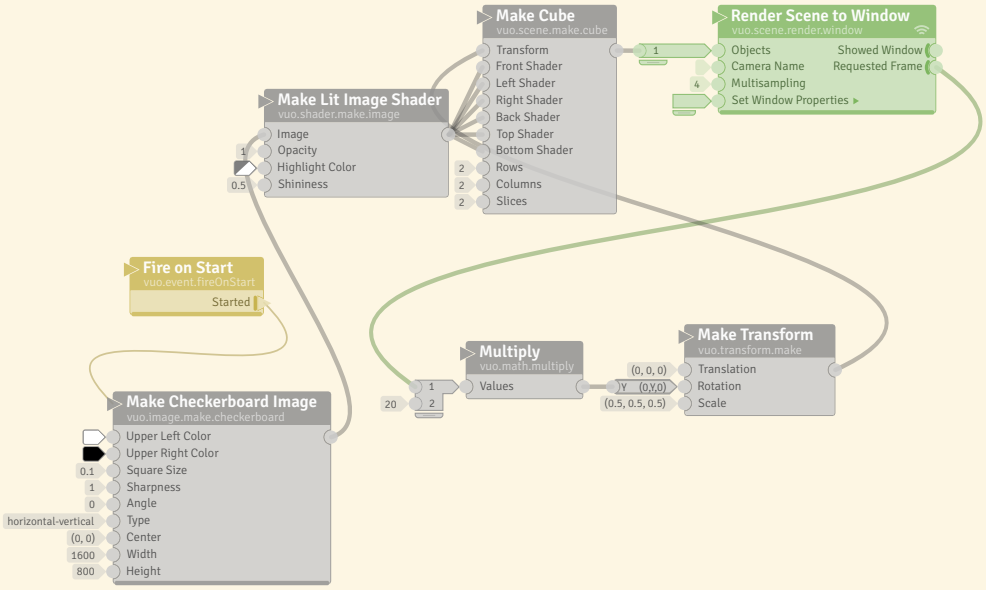



### 7.1.2 Modifying an existing composition

Again, let's start with a problem you want to solve, say to display a rotating cube.

1. You have read from the example composition section that each node set has several example compositions. You know from experience that vuo.scene likely covers 3D objects and scenes.
2. The example composition, Spin Sphere, sounds pretty close to what you want to do, so open that.
3. Run the example and see that it spins a sphere.
4. Search for "cube" in the node library and find **Make Cube**. Add it to the scene.
5. Now as you inspect the example composition you see that it has a **Make 3D Object**. This might be where to insert the **Make Cube** node. But, **Make 3D object** has three cables going into it. You don't think you'll need **Make Sphere Mesh**, but what about the **Make Lit Image Shader**?

6. Start by replacing the **Make 3D Object** node with the **Make Cube** node and connect it up to the **Render Scene to Window** node. Leave the **Make Lit Image Shader** node not connected. You see that there is a **Transform** input port on the **Make Cube**, like there was on the **Make 3D Object**. Connect up the output port of the **Make Transform** node to the **Transform** port.
7. Run the composition.
8. Now you see a large rotating cube, but it doesn't have any color on its sides, just a ghostly checkerboard. Let's connect up the **Make Lit Image Shader** to all sides of the cube, and run the composition again.
9. Now the cube has a black and white checkerboard on all sides, but you'd like it to be smaller. Adjust the **Scale** on the **Make Transform** node to 0.5, 0.5, 0.5.
10. and now you have a rotating cube.




## 7.2 Common patterns - "How do I"

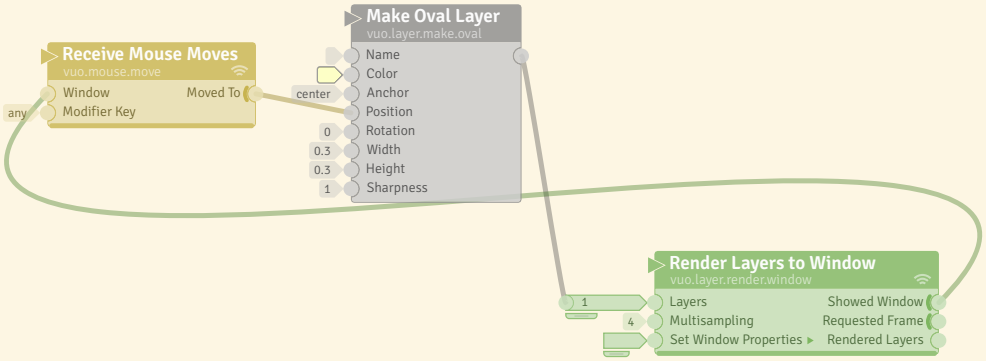
If you're trying to figure out how to accomplish something in Vuo, one good starting point is the Node Library search bar. For example, if you want to make a random list of things, search the Node Library for "random" to find relevant nodes like **Make Random List** and **Shuffle List**. Another good starting point is the example compositions for each node set, found under File > Open Example.

Some problems you might want to solve with Vuo aren't specific to one node or node set. Certain patterns come up again and again, whether you're making compositions to display graphics, play audio, or anything else. This section covers these general patterns. Reviewing these patterns can help you create compositions more quickly and easily.

### 7.2.1 Do something in response to user input

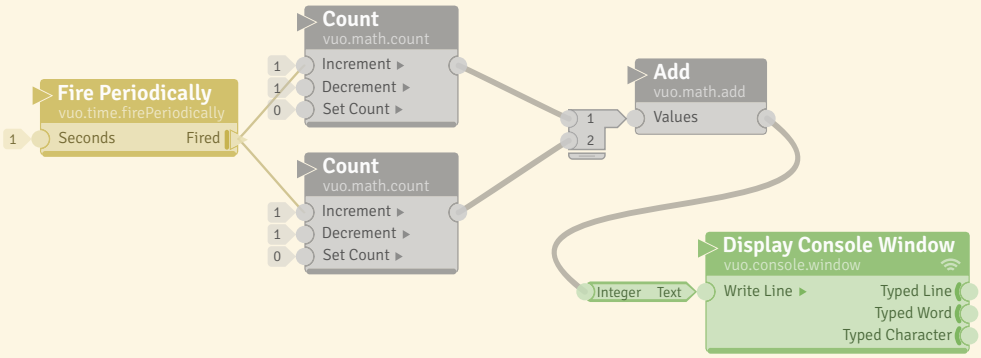
Since Vuo is event-driven, this is easy. Most nodes that get user input have a trigger port that fires an event each time new input comes in. To make something happen in response to that event, just connect a cable from the trigger port to the nodes that make it happen.

Here's an example that makes a circle follow the mouse cursor as the user moves the mouse around.




### 7.2.2 Do something after something else is done

This is often quite easy, too, because of Vuo's rules for event flow. If you want one node to execute before another, you can just draw a cable from the first node to the second node. In the composition below, for each event from **Fire Periodically**, the two **Count** nodes always finish executing before the **Add** node begins executing.



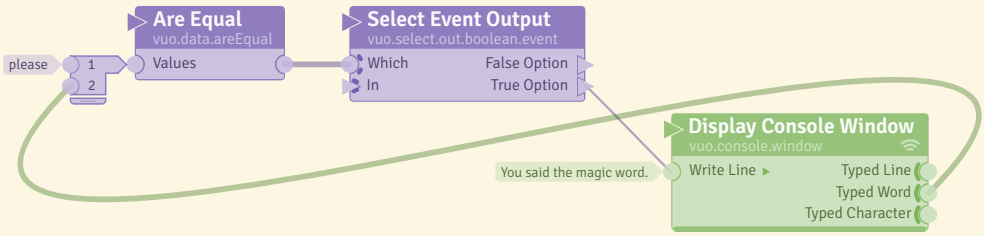

Sometimes you might need to enforce a "do something after something else is done" rule that's more complicated than putting nodes in a sequence, as above. For example, you might want a composition to do something only after the user has typed a certain word. The next section explains how to check for conditions like that and do something when they're fulfilled.

### 7.2.3 Do something if one or more conditions are met

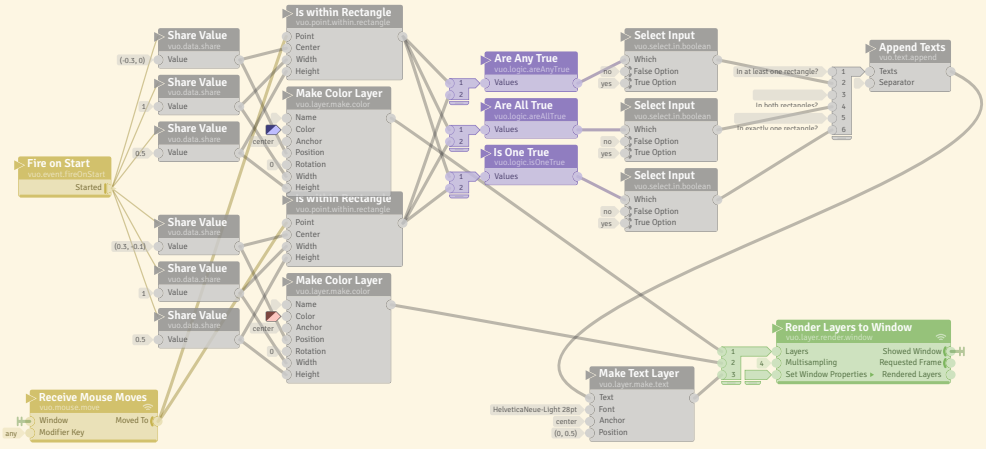
Vuo has a data type that represents whether a condition is met: the Boolean data type. If a node has a Boolean port, that port's value can be one of two things: *true* or *false*. *True* means "yes, the condition is met". *False* means "no, the condition is not met".

When checking if conditions are met, you'll often be working with nodes that have a Boolean output port. Many such nodes have a title that starts with "Is" or "Are", like **Is Greater than** and **Are Equal**.

Here's an example that writes a message on the console window when the user types the word "please".

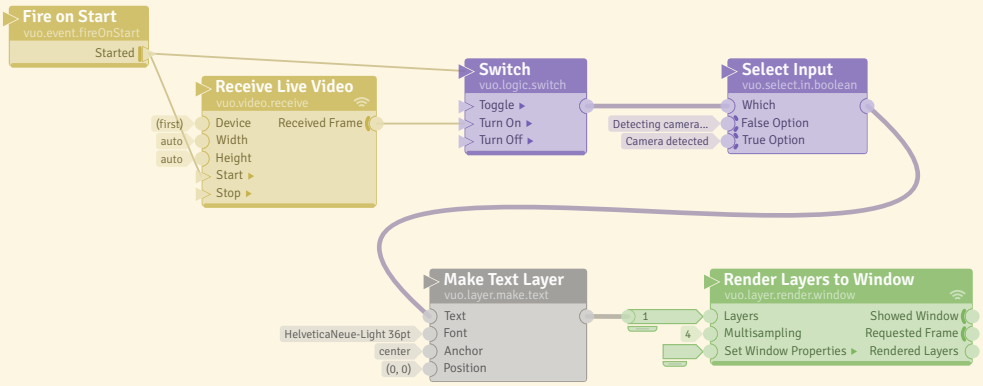



Below is an example (File > Open Example > vuo.logic > Is Mouse Within Intersecting Rectangles) that checks two conditions: is the mouse cursor within the blue rectangle? is it within the red rectangle? The **Are Any True** node says yes (*true*) if the mouse is within at least one of the rectangles. The **Are All True** node says yes if the mouse is within both rectangles. The **Is One True** node says yes if the mouse is within one rectangle and not the other.




Here's one more example. It demonstrates how conditions can be used to coordinate between nodes downstream of different triggers. The composition displays the message "Camera detected" once it starts receiving input from the user's video camera, that is, once the **Receive Live Video** node's trigger

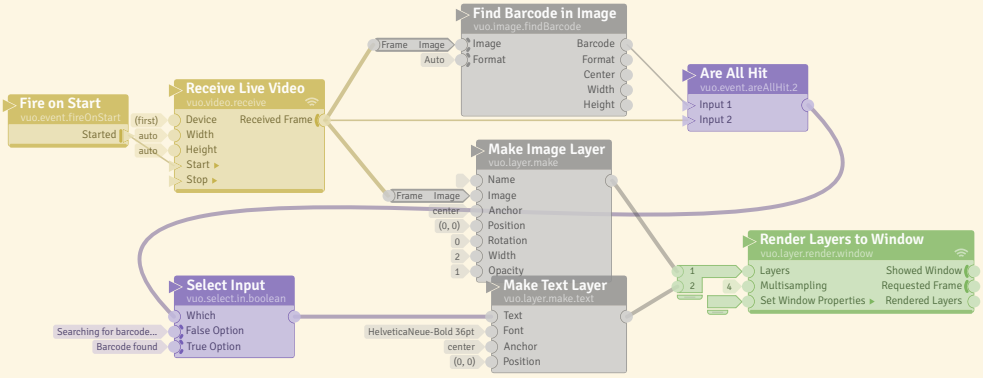
port starts firing events. The events from that trigger port change the **Switch** node's output to *true*, indicating to the rest of the composition that "Camera detected" should be displayed.




### 7.2.4 Do something if an event is blocked

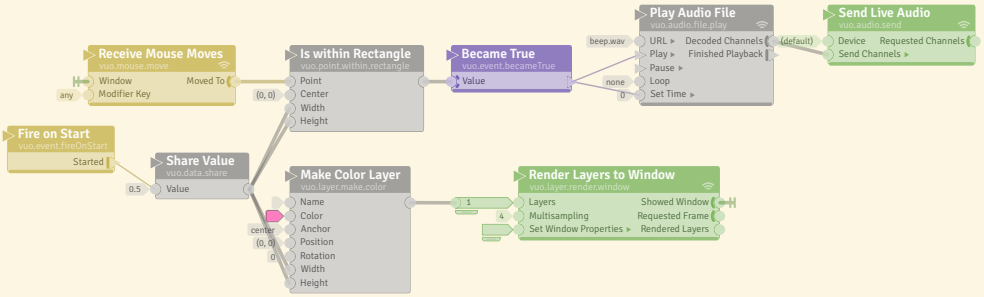
Nodes that have an event door on an input port can let some events through and block others. If you want to do something different depending on whether the event was let through or blocked, you can use an **Are All Hit** node.

Below is an example that checks if a barcode was found in an image. Since the **Find Barcode in Image** node blocks events when no barcode is found, the **Are All Hit** node is used to check whether the event was blocked. **Are All Hit** outputs *false* if **Find Barcode in Image** blocks the event and *true* otherwise.

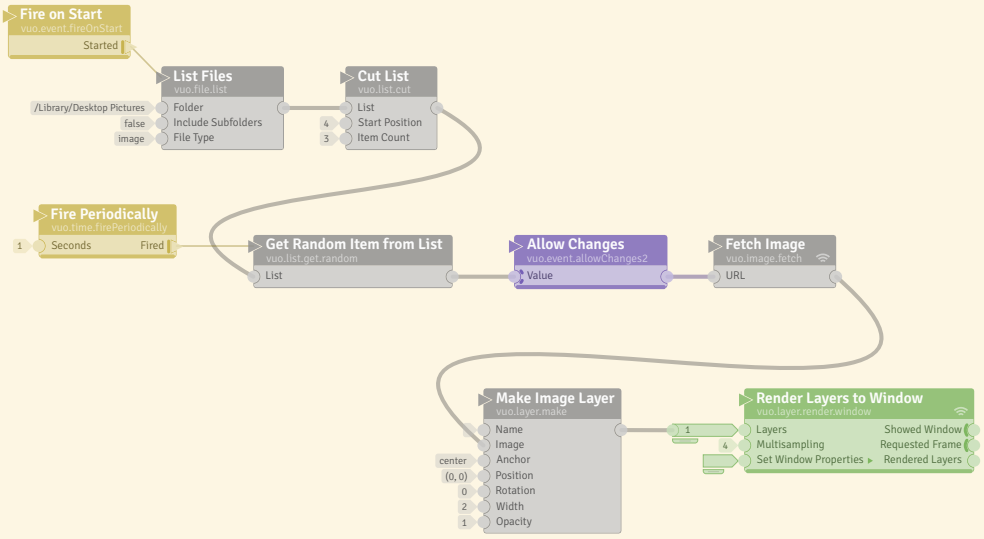



### 7.2.5 Do something if data has changed

Several nodes check if data has changed in a certain way and only let an event through if it has: **Changed**, **Increased**, **Decreased**, **Became True**, and **Became False**. In the composition below, the **Became True** node outputs an event each time the output of **Is Within Rectangle** changes from *false* to *true*, emitting a sound effect each time the mouse cursor enters the square.




Like **Became False** and the other nodes just described, the **Allow Changes** node only lets an event through if the data has changed. But **Allow Changes** is different because it passes the data through along with the event. This can be useful when your composition does something time-consuming or processor-intensive with the data, and only needs to do that work when the data changes. For example, this composition periodically picks a large image file to load, but avoids reloading the same image file if it's picked twice in a row.

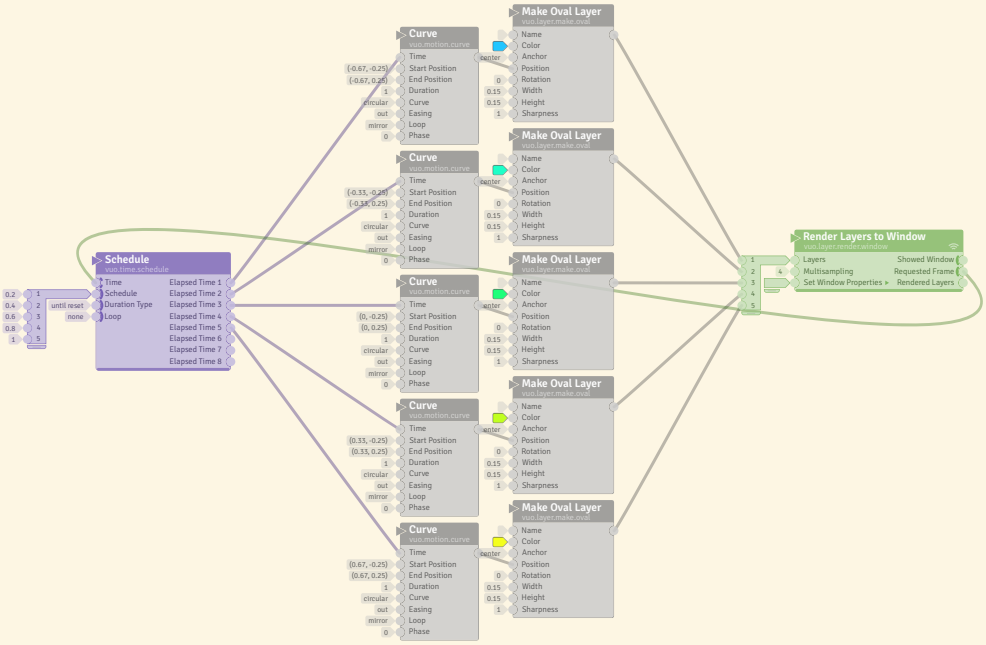



### 7.2.6 Do something after an amount of time has elapsed

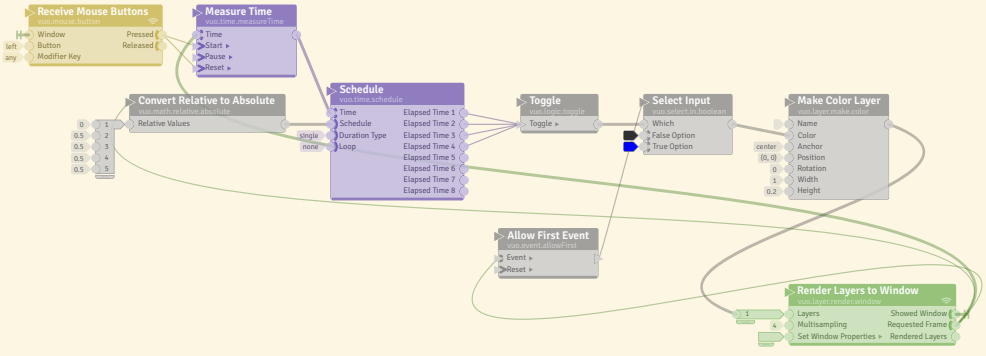
Sometimes, you may want a composition to do something immediately in response to an event. Other times, you may want it to wait until a certain amount of time has passed to do something — for example, launch an animation, start a video, or display a message.

This composition (File › Open Example › vuo.time › Animate On Schedule) launches a series of animations. At 0.2, 0.4, 0.6, 0.8, and 1 second after the composition starts, it sets in motion the next in a series of circles. The bouncing movements of the circles are staggered because each **Elapsed Time**

port of the **Schedule** node outputs a time that's 0.2 seconds after the previous **Elapsed Time** port's value.



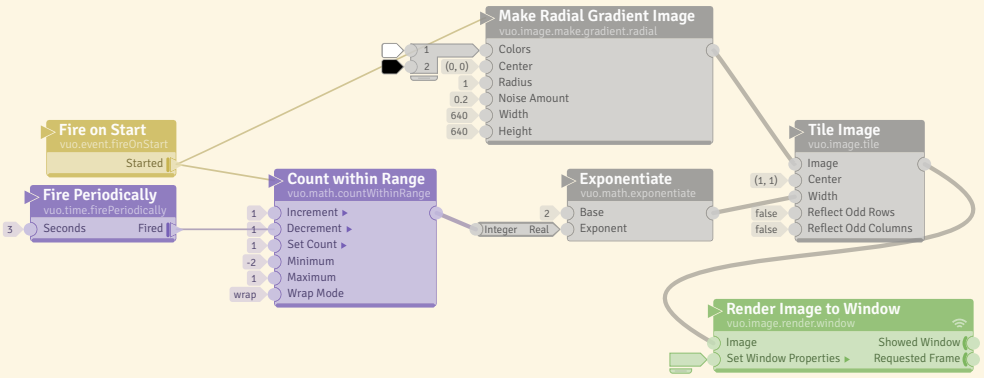

Instead of scheduling things relative to the start of the composition, the composition below (File > Open Example > vuo.time > Flash On Mouse Press) schedules things relative to the most recent mouse press. When the mouse is pressed, the rectangle's color changes to blue, then gray, then blue, then gray. Why does the **Schedule** node in this composition schedule things relative to the most recent mouse press, instead of relative to when the composition started, as in the previous example? Because the **Schedule** node's **Time** input port gets its data from the **Measure Time** node, which outputs the time elapsed since the mouse press.



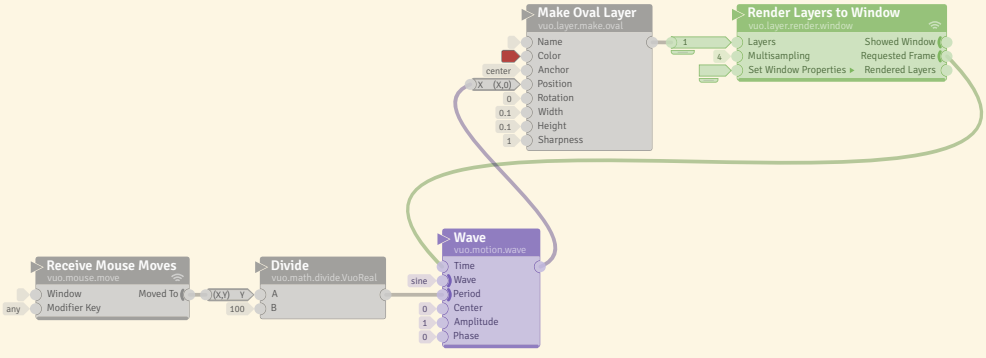
### 7.2.7 Do something repeatedly over time

If you want a composition to do something every N seconds, there are several nodes that fire events at a steady rate. The **Requested Frame** trigger port on nodes such as **Render Scene to Window** and **Render Layers to Window** fires every time the computer display refreshes, which is usually about 60 times per second. For a faster or slower rate, you can use the **Fire Periodically** node.

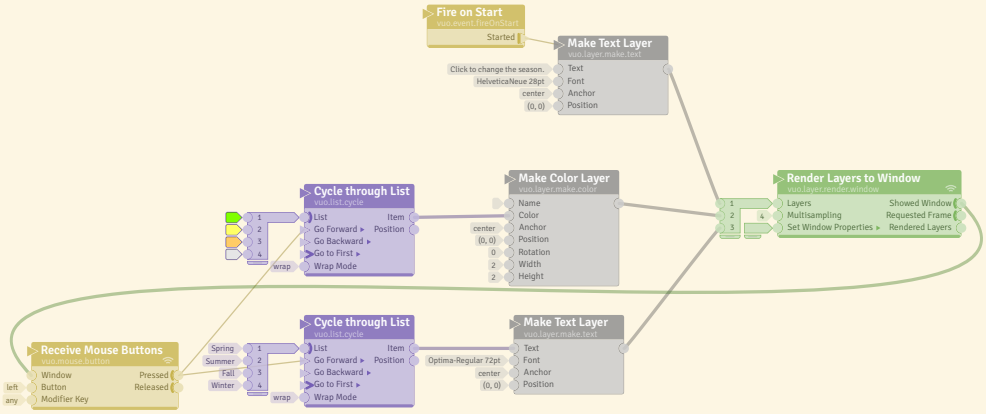
The composition below uses a **Fire Periodically** node to change the width and number of tiled copies of an image every 3 seconds. This composition actually has two kinds of repetition over time. One is the change in tile width that occurs every 3 seconds because of the **Fire Periodically** node. The other is that the tile width repeats itself every 12 seconds. It goes from 2, to 1, to 0.5, to 0.25, and then back to 2. This wrapping-around of the tile width is done by the **Count within Range** node.




**Count within Range** is one of many ways to cycle through a series of numbers. Another is the **Curve** node when its **Loop** port is set to *Loop* or *Mirror*. And another is the **Wave** node. The composition below (File > Open Example > vuo.motion > Wave Circle) uses the **Wave** node to make a circle move back and forth.



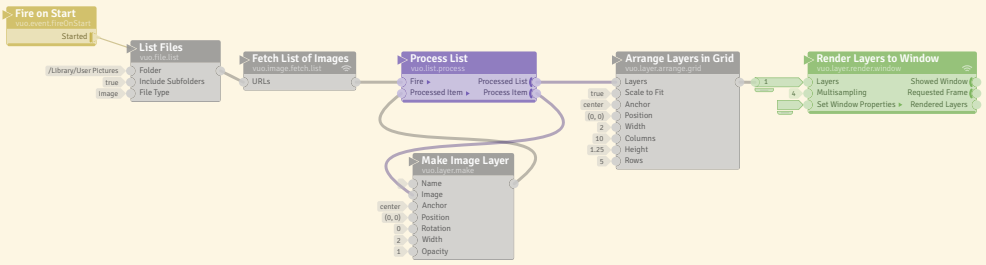

If you want to cycle through a series of things other than numbers, you can use **Cycle through List**. Here's an example (File > Open Example > vuo.list > Cycle Seasons) that uses **Cycle through List** nodes to cycle through colors and texts, displaying the next one each time the mouse is pressed.




## 7.2.8 Do something to each item in a list

The previous section showed how to do something with each list item in turn, using a **Cycle through List** node. For each event the **Cycle through List** node receives, it outputs one list item. If instead you want an event to do something to all list items, you can use the **Process List** node.

Here's an example (File > Open Example > vuo.list > Display Grid Of Images) that turns a list of images into a list of layers using **Process List**. When **Process List** gets an event and list of images into its **Fire** port, it rapidly fires a series of events through its **Process Item** port, one event for each image in the list. The image and event go through the **Make Image Layer** node, and the created layer and event go into the **Process List** node's **Processed Item** port. Once that port has received as many events as **Process Item** fired, the **Processed List** port fires an event with the accumulated list of created layers.




## 7.2.9 Create a list of things

If you don't have a list to start with, one way to create one is with the **Build List** node. **Build List** looks a lot like **Process List**. The difference is that the **Build List** node's **Fire** port inputs an integer (the number of list items to create) instead of a list, and the **Build Item** port rapidly fires a series of integers (from 1 to the number of list items) instead of input list items. Here's an example (File

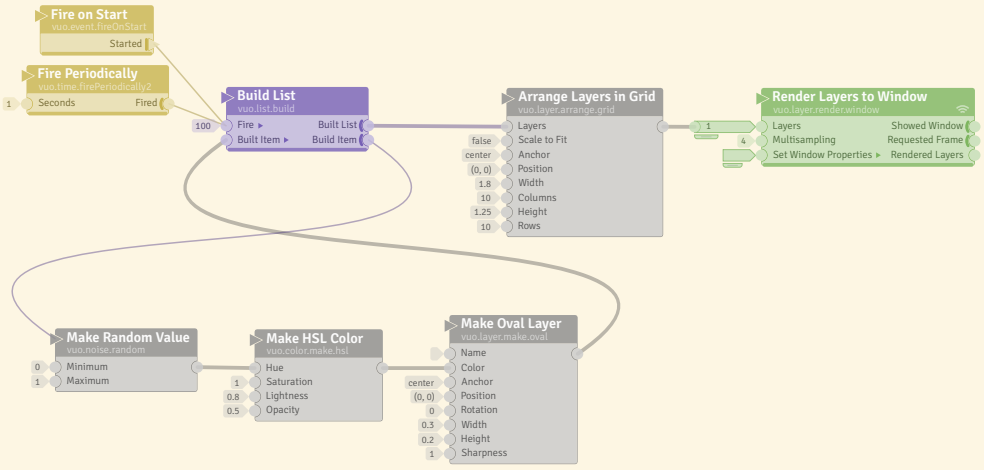
**Note for Quartz Composer users**

Where you would use an **Iterator** patch in Quartz Composer, you might use a **Process List** or **Build List** node in Vuo.

**Note for text programmers**

**Process List** and **Build List** are Vuo's general-purpose nodes for iteration. They're similar to text programming constructs such as loop control structures and foreach and apply functions.

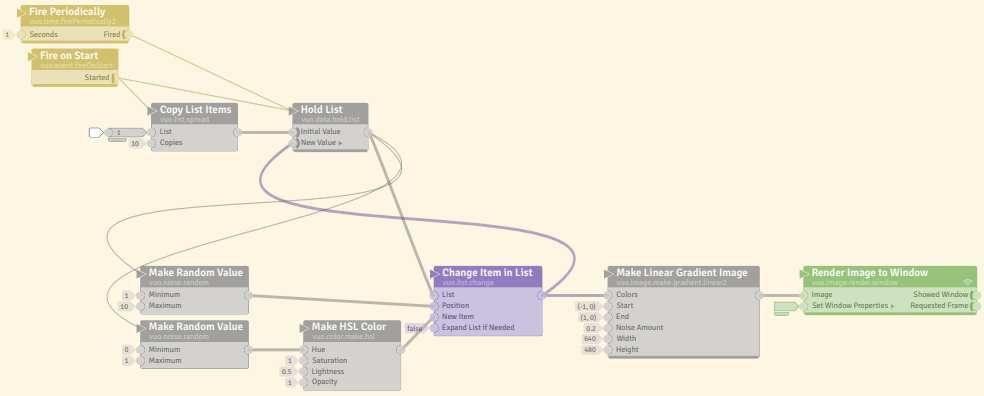
Open Example > vuo.list > Display Rainbow Ovals) that uses the **Build List** node to display a grid of 100 different-colored ovals.




**Build List** and **Process List** are general-purpose tools. Vuo also provides some simpler, more specialized ways to create certain types of lists. These include **Make Random List** to make a list of random numbers or points, **Copy Layer** and **Copy Scene** to duplicate a 2D or 3D object, and **Enqueue**, which is explained in the next section.

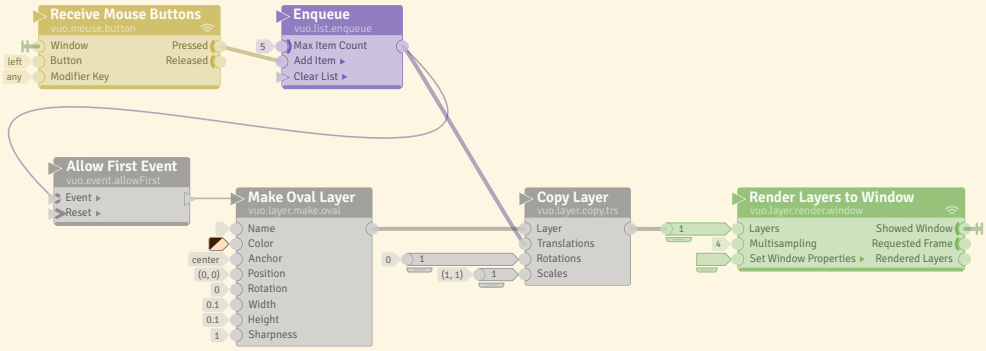
### 7.2.10 Maintain a list of things

Sometimes you may want not only to create a list, but also to hold onto it and make changes to it over time. One way to do that is with a feedback loop, as in the example composition below (File > Open Example > vuo.list > Replace Colors In Gradient). It maintains a list of colors, randomly changing one of them every 1 second.



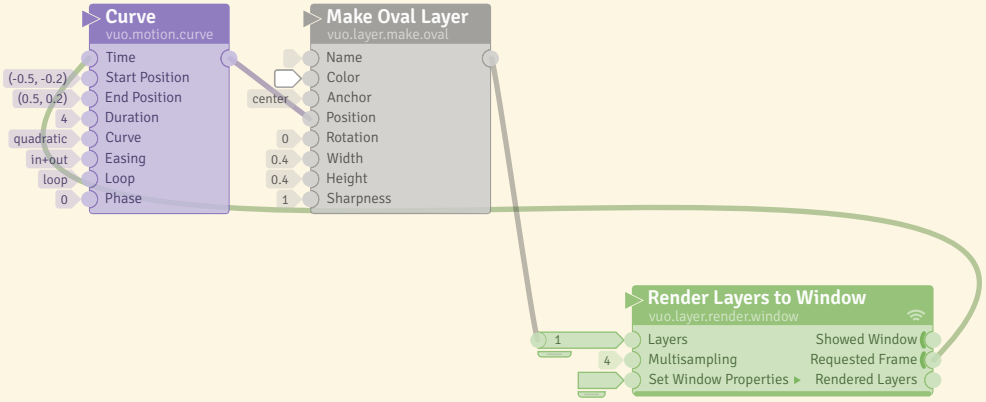

Another way you might want to maintain a list is to accumulate a queue of items over time, using the **Enqueue** node. A queue in this node is like a queue of people waiting in line. It's first-in-first-out,

meaning that new items get added to the end of the line, and the item that's been waiting in line the longest is the next one that can leave the queue. Here's an example that uses **Enqueue** to remember the positions of the 5 most recent mouse presses.

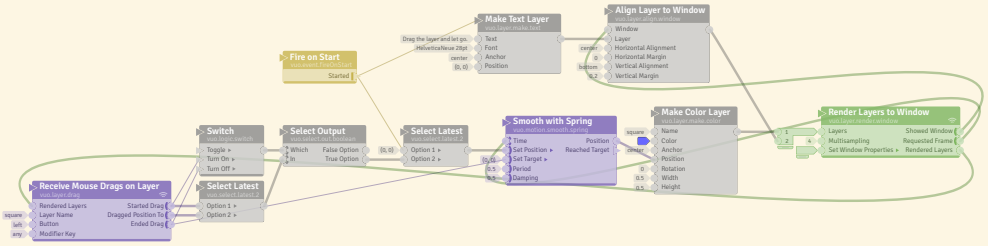



## 7.2.11 Gradually change from one number/point to another

Earlier, under "Do something repeatedly over time", the **Curve** and **Wave** nodes were mentioned as ways to cycle through a series of numbers or points. You can also think of these nodes as ways to gradually change from one number or point to another. Here's an example that uses a **Curve** node to gradually move a circle from one point to another. Since the **Curve** port is set to *Quadratic* and the **Easing** port is set to *In + Out*, the circle starts moving slowly, picks up speed, and then slows down as it reaches its destination.




Another way to gradually change from one number or point to another is with the "Smooth" nodes — **Smooth with Duration**, **Smooth with Inertia**, **Smooth with Rate**, and **Smooth with Spring**. Here's an example (File > Open Example > vuo.motion > Spring Back) that makes a square spring back to the center of the window when the user drags and releases it.

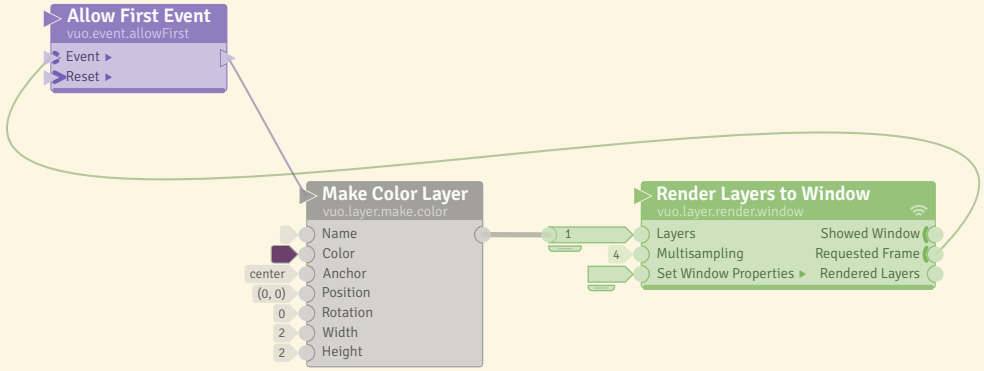



## 7.2.12 Set up a port's data when the composition starts

When a composition starts running, its data-and-event input ports start out with some initial data — either the port's constant value, if you've used the input editor to set one for the port, or the port's default value. An input port with an incoming data-and-event cable stays at its default value until the first data-and-event comes in through the cable. Sometimes you may want to send certain data with that first event so that the port will start off with the right value.

A simple way to do that is with a **Fire on Start** node. In the **Smooth with Spring** example in the previous section, the **Fire on Start** node fires an event that sets up the data for two input ports. One is the **Align Layer to Window** node's **Layer** input port, which gets the layer created by **Make Text Layer**. The other is the **Smooth with Spring** node's **Set Position** input port, which gets initialized to (0,0). The **Select Latest** node helps out here by sending (0,0) to the **Set Position** port for the **Fire on Start** event and, after that, the current mouse position each time the **Receive Mouse Drags on Layer** fires an event.

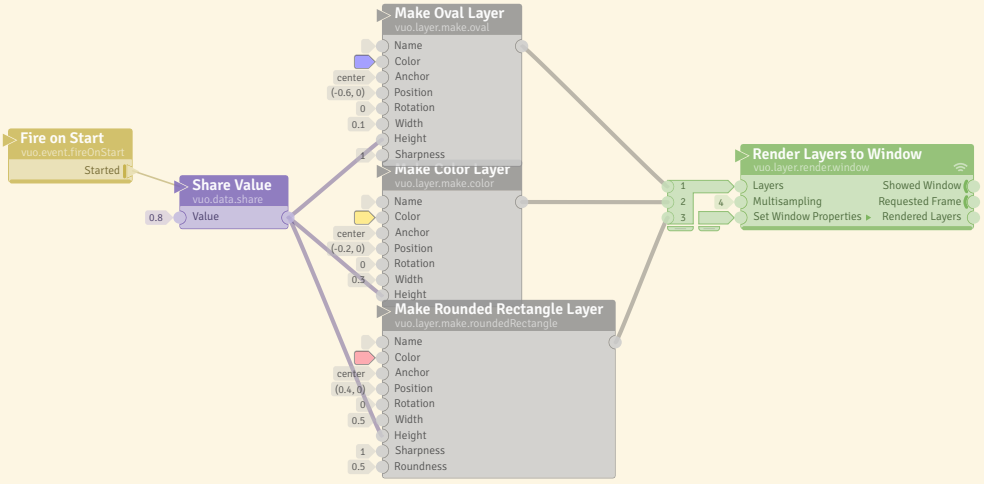
Using **Fire on Start** to set up data is pretty simple, but it has one weakness: the **Fire on Start** node's trigger isn't coordinated with other triggers in the composition. If you're trying to use **Fire on Start** together with the **Requested Frame** port of **Render Layers to Window** or **Render Scene to Window**, you might see a momentary flicker or adjustment in graphics as the composition starts. That's because the **Fire on Start** event and the first **Requested Frame** event are setting up different parts of the graphics at slightly different times. So how can you avoid the flicker? Instead of **Fire on Start**, which fires its own event, use **Allow First Event**, which can borrow the event fired from **Requested Frame**. Here's an example.



### 7.2.13 Send the same data to multiple input ports

If you have several input ports in your composition that all need to stay in sync with the same data, then it's usually a good idea to feed cables to all of them from a single output port. But what if the data isn't coming from an output port — what if it's a constant value? In that case, you can use a **Share Value** node to set the constant value in one place and propagate it from the **Share Value** node's output port to all connected input ports.

Here's an example that draws several shapes, all of the same height. You could accomplish the same thing without the **Share Value** node by using input editors to individually set the **Height** input ports to 0.8. The advantage of using **Share Value** is that, if you change your mind and decide the height should be 1.0 instead, you only have to edit it on the **Share Value** node's input port instead of on all connected input ports.



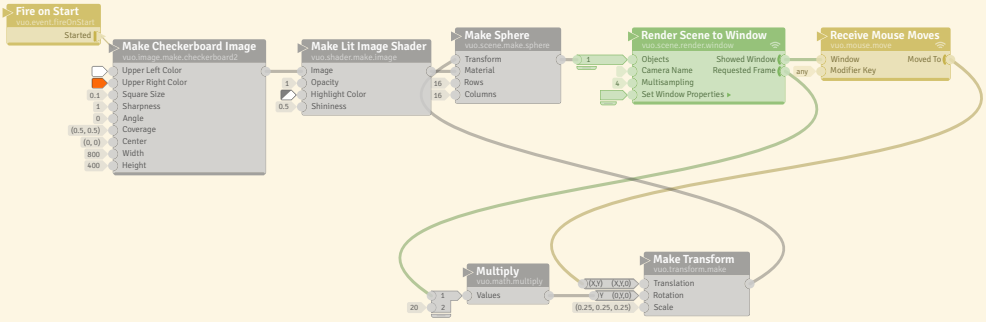

### 7.2.14 Strip out data, leaving just an event

In the previous section's example, you could think of the **Share Value** node as adding data to an event — an event goes into the node, and an event plus data comes out. What if you want to do the opposite — start with an event plus data, and end up with just an event? You don't need a node to do this. Instead, hold down ⇧ (Shift) while dragging a data-and-event cable to change it to an event-only cable.

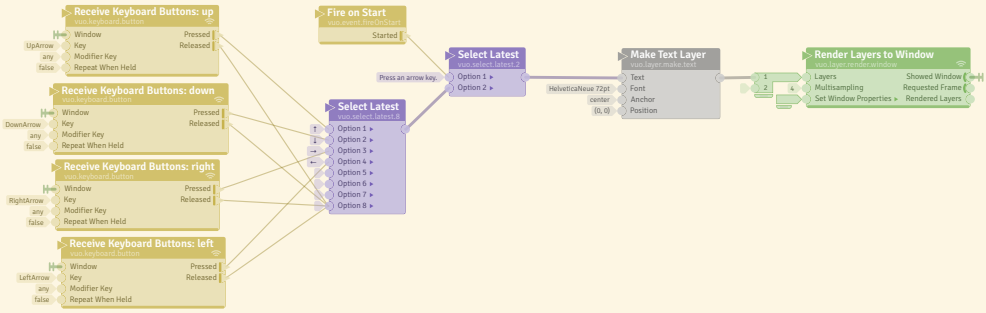
### 7.2.15 Merge data/events from multiple triggers

When you have streams of events from multiple triggers flowing through your composition, usually those streams of events have to merge somewhere in the composition.

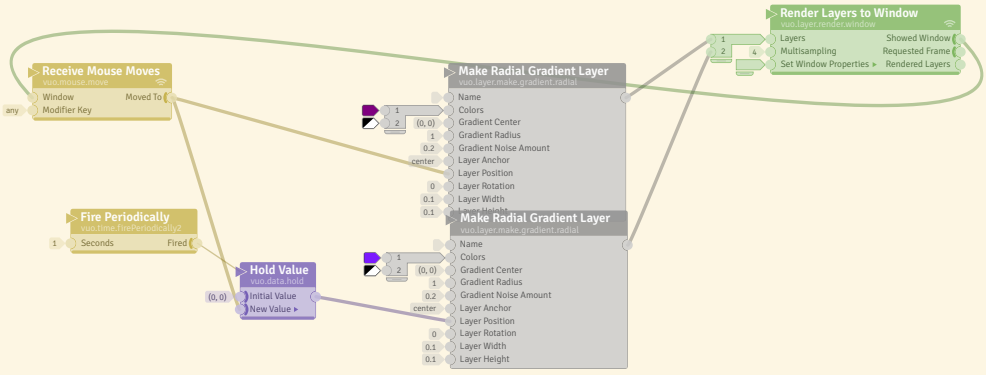
Sometimes the streams of events just naturally overlap, as in the example below (File > Open Example > vuo.scene > Move Spinning Sphere). The events fired from the **Requested Frame** port on **Render Scene to Window** and the events fired from the **Moved To** port on **Receive Mouse Moves** both travel through the **Make Transform** and **Make Sphere** nodes to the **Render Scene to Window Node**.




Other times, you may want to merge the event streams more intentionally. Here's an example (File > Open Example > vuo.select > Show Arrow Presses) that takes input from key presses on different arrow keys, and displays a message for each one. The **Select Latest** node lets the events from each arrow key through.



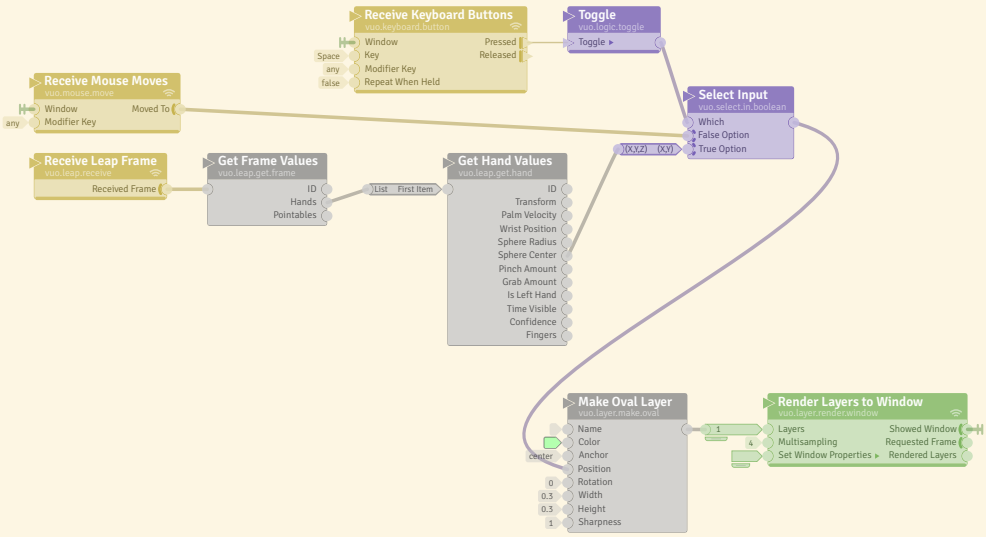

Here's an example that shows a different way of merging two event streams. This composition (File > Open Example > vuo.data > Store Mouse Position) draws two gradients that each follow the mouse cursor a bit differently. The purple (upper) gradient stays with the mouse all the time. The violet (lower) gradient only updates every 1 second. For the lower gradient, the event streams from **Receive Mouse Moves** and **Fire Periodically** merge at the **Hold Value** node. Unlike the composition in the previous example, which let both event streams through, this composition lets one event stream through and blocks the other. However, the data left by the blocked event stream (from **Receive Mouse Moves**) gets picked up and carried along downstream by the other event stream (from **Fire Periodically**).




## 7.2.16 Route data/events through the composition

In the last example in the previous section, events from the **Receive Mouse Moves** node's trigger were always blocked at the **Hold Value** node, and events from the **Fire Periodically** node's trigger were always allowed through. Instead of always blocking one trigger's events and always letting another trigger's events through, what if you want to switch between the event streams?

Here's an example with a keyboard control that switches the data-and-event stream that controls a circle's position. When the user presses the space bar, setting the **Select Input** node's **Which** port to *true*, the circle is controlled by the Leap Motion device. When the user presses the space bar again, setting the **Which** port to *false*, the circle is controlled by the mouse. Whichever data-and-event stream is *not* controlling the circle at a given time is blocked at the **Select Input** node.




Instead of taking multiple event streams and picking one to let through, as in the previous example, what if you have a single event stream and want to pick one of several downstream paths to route it to?

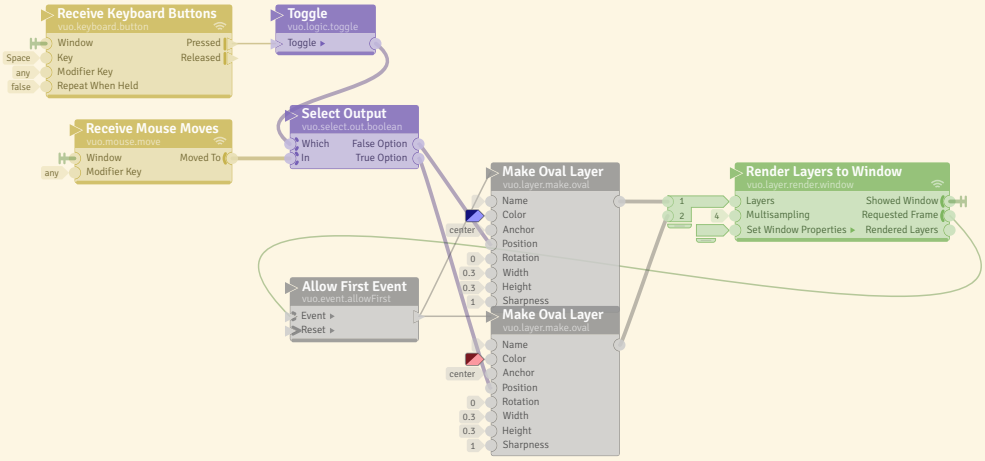
**Note for Quartz Composer users**

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.

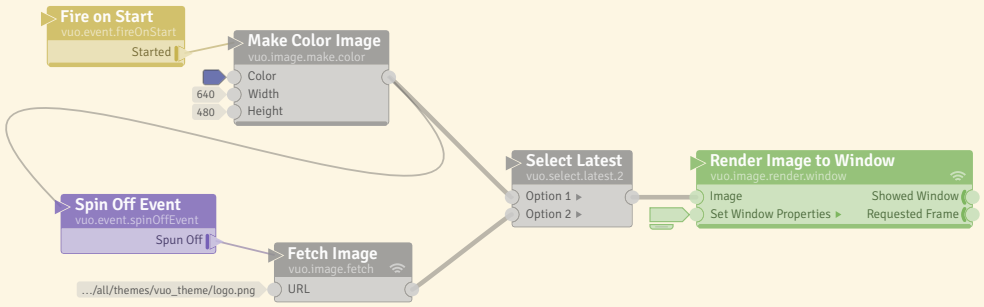
Below is an example of that. The space bar toggles between two circles. Whichever circle is chosen at a given time is controlled by the mouse. This works because the **Select Output** node routes the data-and-event stream from **Receive Mouse Moves** through just one of its output ports at a time.




## 7.2.17 Run slow parts of the composition in the background

Different parts of the composition can be executing simultaneously. If you have multiple triggers firing events through the composition, events from both triggers can be traveling through the composition at the same time. This fact comes in handy if you want a composition to start working on a slow task and do something quicker in the meantime.

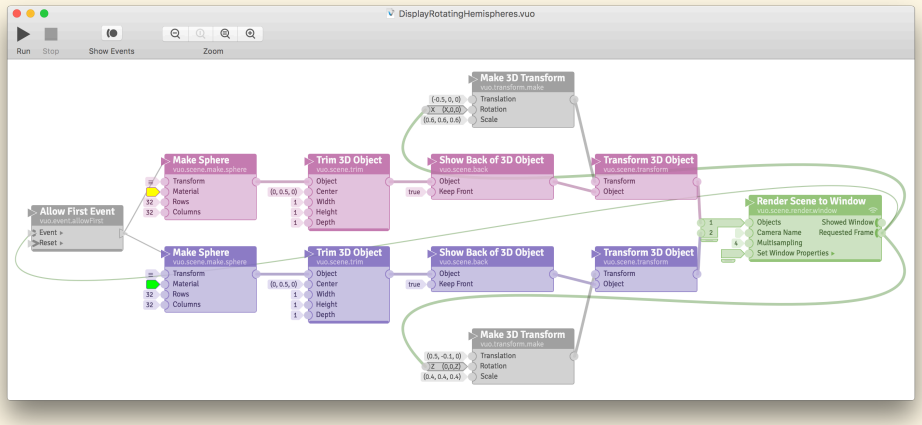
Here's an example (File > Open Example > vuo.event > Load Image Asynchronously). The slow task, in this case, is to download an image from the internet. Immediately after this composition starts running, it starts downloading the image and, in the meantime, fills the window with a solid color. The **Spin Off Event** node is what allows the download to happen in the background. If **Spin Off Event** weren't there, then the **Select Latest** node would wait for both **Make Color Image** and **Fetch Image** to complete before it executed. But, thanks to **Spin Off Event**, the **Fetch Image** node is now executed by a different event than the **Make Color Image** node, so **Select Latest** can go ahead and execute as soon as **Make Color Image** is complete.



# 8 Using subcompositions inside of other compositions

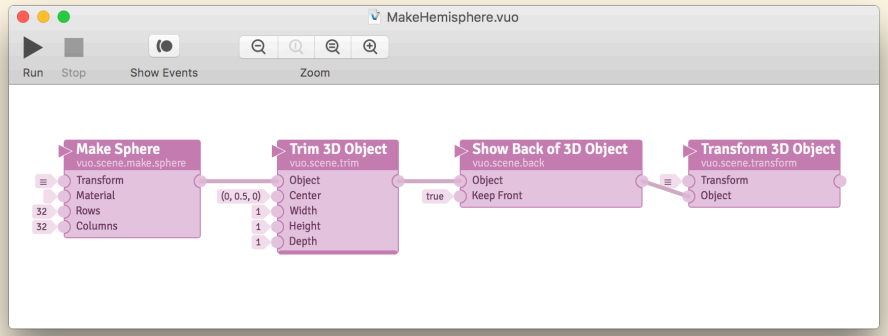
If you find yourself copying and pasting the same group of nodes and cables into many compositions, you may want to turn those nodes and cables into a **subcomposition**. A subcomposition is a composition that can be used as a node inside of other compositions. A subcomposition saves you the effort of having to recreate the same nodes and cables over and over. They're packaged neatly inside a node, which you can drag from the Node Library onto your canvas just like any other node.

Let's walk through an example. Suppose you often draw hemispheres (half spheres) in your 3D compositions, and it would be convenient to have a **Make Hemisphere** node in your Node Library. The first step is to identify the nodes and cables that you want to package into a subcomposition.



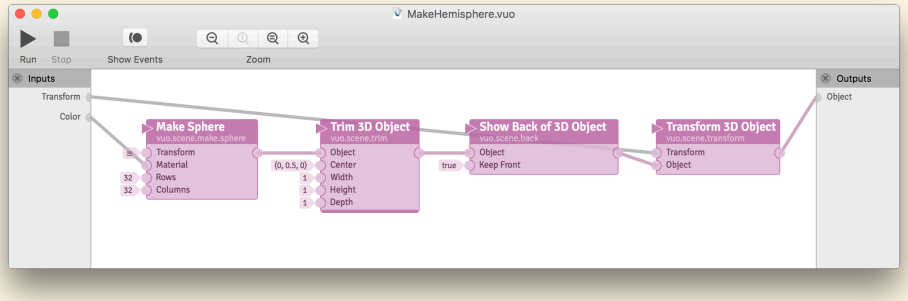

The composition above renders two rotating hemispheres to a window. (The **Trim 3D Object** node cuts off half of the sphere. The **Show Back of 3D Object** node makes the inside of the sphere visible.) In other compositions, you might want to create any number of hemispheres. The hemispheres could have different rotations, positions, sizes, and colors. They could be rendered to a window or an image. For the subcomposition, let's choose a piece of the composition that's flexible enough to be used in all of these cases: the nodes and cables tinted magenta.

The next step is to create the subcomposition, which starts as a regular composition (File › New Composition). You can copy and paste a piece of an existing composition into the new composition.




The subcomposition above contains the nodes that were repeated in the previous composition to make two hemispheres. In the previous composition, the **Make Sphere** node's **Material** input port and the **Transform 3D Object** node's **Transform** input port controlled the color, position, rotation, and size of the hemisphere. The **Transform 3D Object** node outputted a Scene Object that was sent to the **Render Scene to Window** node.

But in this subcomposition, since there aren't any cables going into the first node in the line (**Make Sphere**), none of the nodes will execute. And since there aren't any cables going out of the last node in the line (**Transform 3D Object**), the Scene Object won't be rendered. In short, this composition won't do anything. The next step is to add some cables.

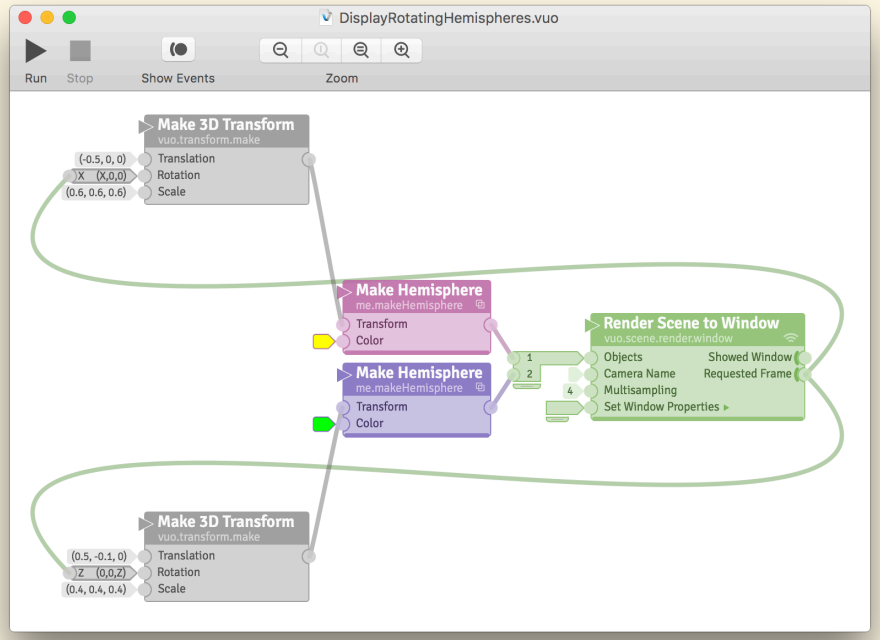



The subcomposition above now has published ports and cables. Published ports are the subcomposition's way of inputting/outputting events and data from/to the composition that contains it. When you turn the subcomposition into a node (in the next step), its published input and output ports will become the input and output ports on the node.

The above subcomposition has published input ports for the hemisphere's **Transform** and **Color**. These published input ports provide enough flexibility to change the hemisphere's position, rotation, size,

and color. They don't allow changing the **Trim 3D Object** node's inputs, since that would make the subcomposition output a shape other than a hemisphere.

Now that the subcomposition contains the nodes needed to create a hemisphere and the published ports and cables needed to input and output data and events, you can turn the subcomposition into a node. Go to File > Move Composition to Node Library (or File > Save Composition to Node Library if you haven't yet saved the composition). This makes the subcomposition appear as a node in your Node Library. You can now use the node inside of other compositions, as demonstrated below.




## 8.1 Reasons to use subcompositions

The **Make Hemisphere** subcomposition illustrated one motivation for using subcompositions: to avoid recreating the same composition pieces over and over again. A subcomposition enables you to assemble a composition piece once and reuse it many times. If you notice a problem with the subcomposition or want to improve it, you only have to make the change in one place to have it apply everywhere the subcomposition is used.

Another reason you may want to use subcompositions is to better organize large compositions to make them more readable. You can replace a complex network of nodes and cables with a subcomposition that has a descriptive title and a clearly defined set of inputs and outputs.

A third reason for using subcompositions is to share your work with others in a modular format. When you create a composition piece that other people might like to use inside of their compositions, you can package it as a subcomposition that others can install in their Node Libraries.

## 8.2 Saving a subcomposition

When you turn a composition into a node with File 〉 Move Composition to Node Library, Vuo automatically moves the composition file to your User Modules folder.

If you haven't yet saved the composition file, the menu item is File 〉 Save Composition to Node Library, and Vuo saves the composition file to your User Modules folder.

## 8.3 Naming a subcomposition

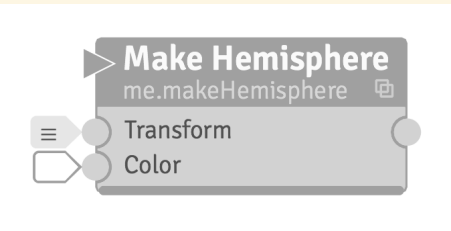
When you turn an already-saved composition into a node, the node's title derives from the composition's file name. A composition file called `Scribble.vuo` or `scribble.vuo` would be turned into a node titled **Scribble**. A composition file called `Solve Anagram.vuo` or `SolveAnagram.vuo` would be turned into a node titled **Solve Anagram**.

If you haven't yet saved the composition file, Vuo prompts you to enter a node title.

The node's class name is your Vuo user name followed by a period followed by a lower-camel-case version of the node title — for example, `me.scribble` or `me.solveAnagram`.

After turning a subcomposition into a node, if you want to change the node's title and class name, you can rename the installed subcomposition file. Do this by right-clicking on the subcomposition node in the Node Library and choosing the menu item Open Enclosing Folder, finding your installed subcomposition in that folder (for example, `me.scribble.vuo`), renaming the file, and restarting Vuo. Be careful renaming a subcomposition, because any compositions that use the subcomposition by its old name will have an error until you substitute in the new version of the subcomposition.

## 8.4 Editing a subcomposition



You can recognize a subcomposition node by the icon in its top-right corner. The icon consists of two overlapping squares, symbolizing that the subcomposition node has a composition inside of it that you can view and edit.

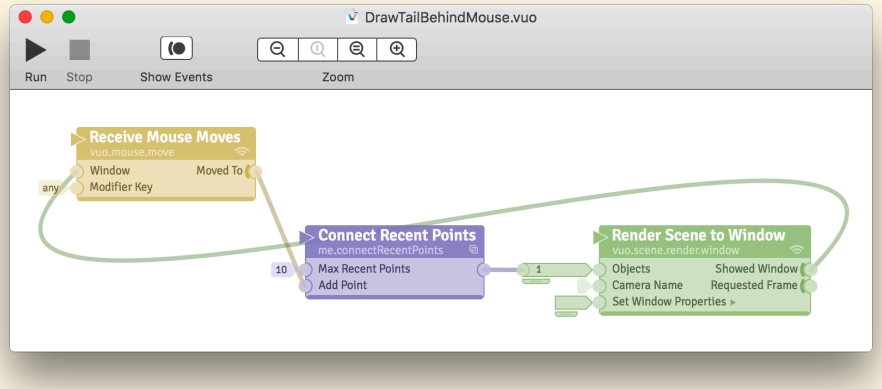
You can open that composition by double-clicking on the subcomposition node on the canvas, or by right-clicking on the subcomposition node on the canvas or in the Node Library and choosing Edit Composition... . When you save changes made to the subcomposition, the changes apply everywhere the subcomposition is used — every instance of the subcomposition node in the composition you have open, and all other compositions that contain the subcomposition.

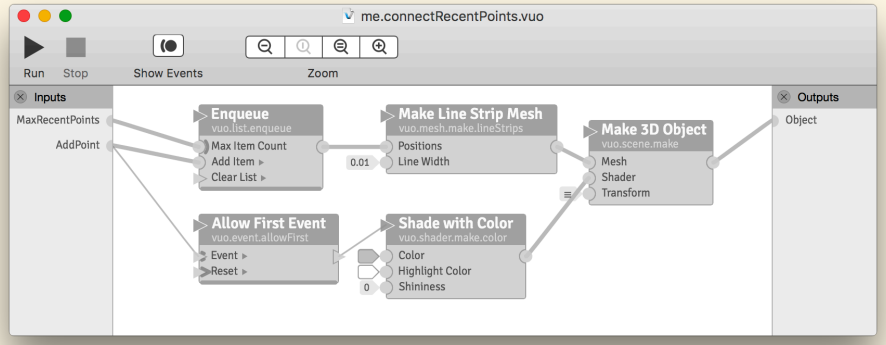
## 8.5 How events travel through a subcomposition

### 8.5.1 Events into a subcomposition

When an event hits any input port of a subcomposition node, the event travels into the subcomposition through *all* of its published input ports.

To illustrate, here's a composition that uses a subcomposition node called **Connect Recent Points** to draw a series of connected line segments behind the mouse cursor as it moves. Below that is the **Connect Recent Points** subcomposition.

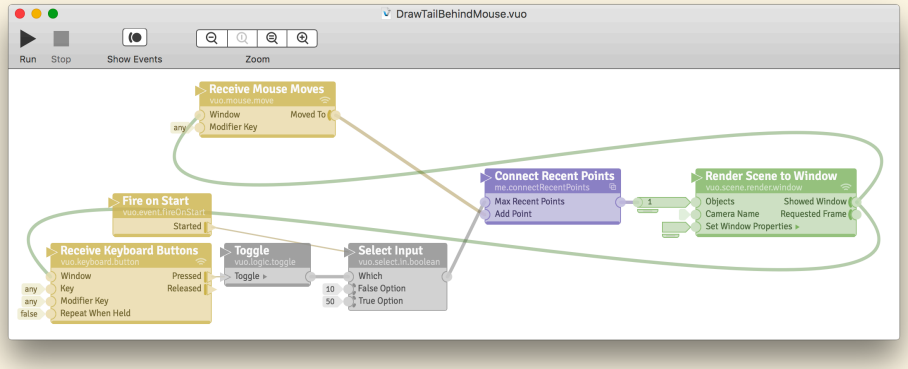





When an event hits the **Add Point** input port of the **Connect Recent Points** node (in the first composition above), it travels into the subcomposition (second composition above) through both of the subcomposition's published input ports, **Add Point** and **Max Recent Points**.

If you were to edit the **Connect Recent Points** node's **Max Recent Points** port's constant value, the **Enqueue** node's **Max Item Count** port's value would immediately change, but the **Enqueue** node wouldn't execute. The next event into the **Connect Recent Points** node's **Add Point** port would cause the **Enqueue** node to execute and, thus, the new **Max Item Count** port value to take effect.

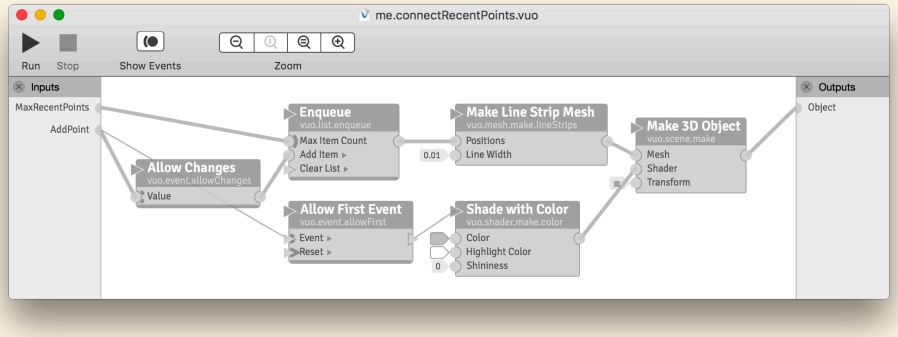
Here's a modification of the first composition above that allows the user to toggle between a short tail and a long tail by pressing any key.




If the user stops moving the mouse and repeatedly presses a key to change the number of recent points, something unintended happens: each time the user toggles back to the short tail, the tail gets even shorter than it was before. After several keypresses, the tail disappears completely. Why?

When the event fired from the **Receive Keyboard Buttons** node hits the **Connect Recent Points** node's **Max Recent Points** port, it goes into the subcomposition through *both* of the subcomposition's published input ports. Inside the subcomposition, the event hits both the **Max Item Count** and **Add Item** port of the **Enqueue** node. This means that not only is the maximum number of points altered, but the most recently added point is added to the queue again — bumping the least recent point out of the queue. Since changing the maximum number of points has the unintended side effect of deleting the oldest point in the tail, the tail gets shorter and shorter.

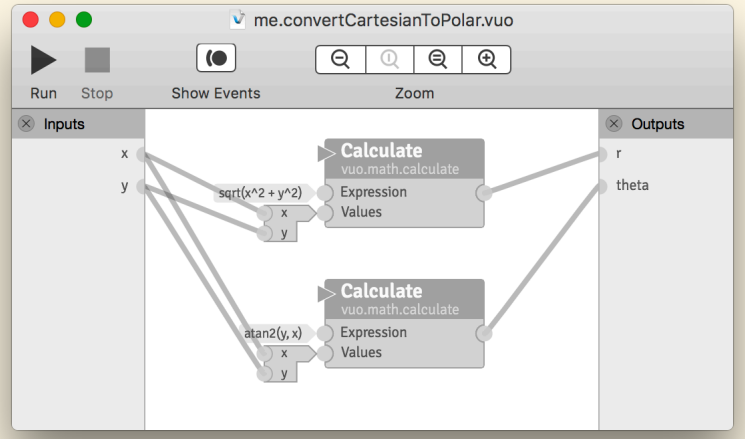
To avoid this problem, you can modify the subcomposition to block the event into the **Enqueue** node's **Add Item** port when it carries the same point twice in a row. The modified subcomposition below accomplishes this with an **Allow Changes** node.




### 8.5.2 Events out of a subcomposition

If an event reaches the published output port of a subcomposition, it travels out of the corresponding output port of the subcomposition node.

If an event into a subcomposition node reaches multiple published output ports of the subcomposition, it travels out of all of the subcomposition node's output ports simultaneously. For example, in the subcomposition below, even though the **Calculate** nodes can execute concurrently and may not output their values at exactly the same time, the **Convert Cartesian To Polar** subcomposition node always outputs the event from its **R** and **Theta** ports simultaneously.

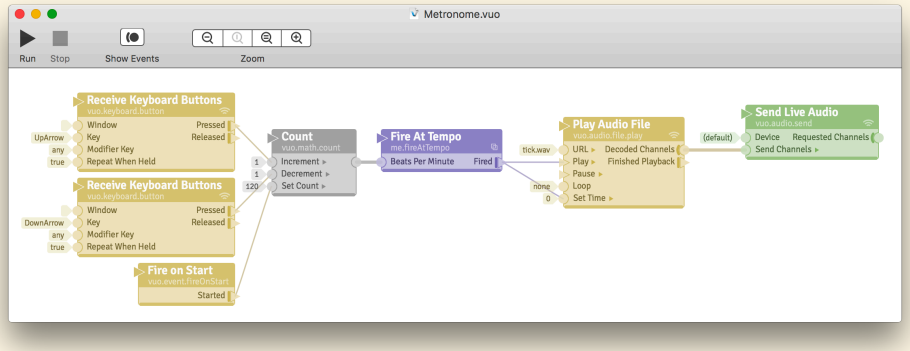


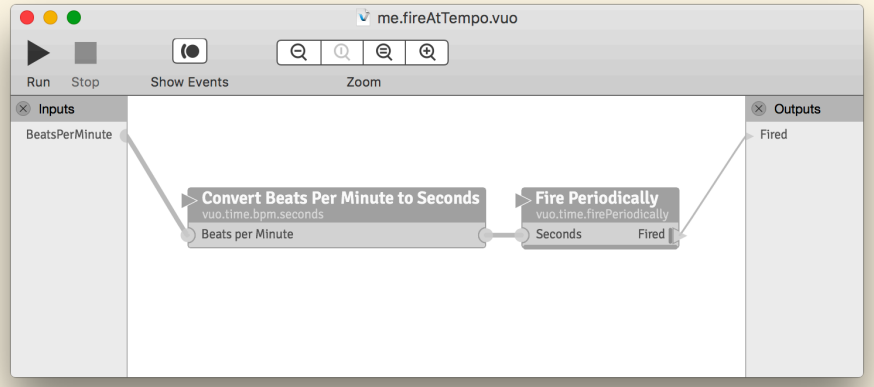

If an event that comes in through a subcomposition's published input ports reaches at least one of the subcomposition's published output ports, then the event comes out of *all* of the subcomposition node's output ports.

If an event that comes in through a subcomposition's published input ports doesn't reach any of the subcomposition's published output ports (because of wall or door ports within the subcomposition), then the event doesn't come of out any of the subcomposition node's output ports. The subcomposition node blocks the event.

The refresh port is an exception. When an event hits the subcomposition node's refresh port, the event always comes out of all of the subcomposition node's output ports.

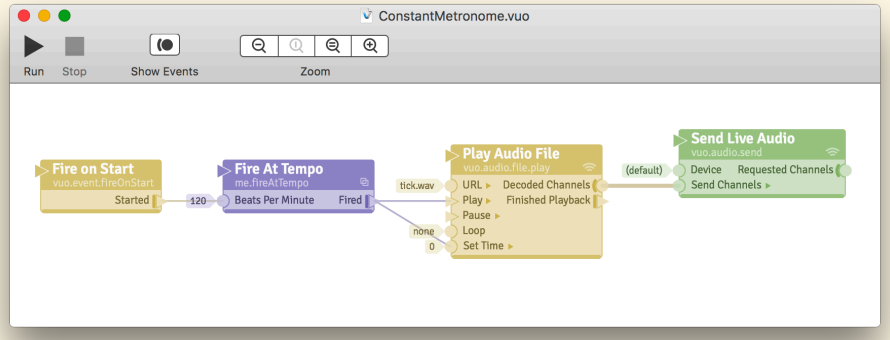
A subcomposition can fire events, as demonstrated below. The **Fire At Tempo** subcomposition node is set to fire at a rate of 120 beats per minute.






### 8.5.3 Constant input port values

A variation on the previous example demonstrates an important point about how events flow into and through a subcomposition.




In the composition above, when the event from **Fire on Start** hits the **Fire At Tempo** subcomposition node, the event travels through the **Convert Beats Per Minute to Seconds** node and on to the **Fire Periodically** node, changing that node's firing rate to 120 times per minute. But if you edit the constant value of the **Fire At Tempo** node's **Beats Per Minute** input port, the firing rate doesn't change. Why?

When you edit the **Fire At Tempo** node's **Beats Per Minute** port's value, the **Convert Beats Per Minute to Seconds** node's **Beats per Minute** port's value immediately changes. But there's no event to make the **Convert Beats Per Minute to Seconds** node execute and carry its output data to the **Fire Periodically** node. For the new tempo to take effect after you've edited the input port value, you have to manually fire an event from **Fire on Start**.

# 9 Using compositions in other applications

Some applications can load Vuo compositions as plugins — making it possible for you to customize or add to the application's behavior. For example, CoGe and VDMX, VJ applications that do media mixing and compositing, let you mix in visual effects made with Vuo. In order for a Vuo composition to communicate with CoGe, VDMX, or another application, you need to create the composition in a special way using published ports and protocols.

If you're going to use a Vuo composition in an application, usually the application expects the composition to have certain published input and output ports. For example, an application that supports Vuo compositions for video effects would expect the composition to have a published input port that receives an image and a published output port that sends an altered image. To guarantee that your composition's published ports match what the application is expecting, you can use a protocol. The composition shown above adheres to a Vuo protocol.

**Note for Quartz Composer users**

In Quartz Composer, there's a single, default rendering output. VJ apps typically capture that output and mix it into the VJ app's other feeds, so there is no need for a protocol for composition playback. In Vuo, use the Image Filter or Image Generator protocol to produce output.

# 10 Exporting compositions

When you create something really interesting with Vuo, you might want to share it with the world. Sometimes you may want to share the composition (.vuo) file so that other people with Vuo can see how your composition works and tweak it themselves. But other times you may want to show your work to people who don't necessarily have Vuo. The best way to do that is to export your composition to a format that other people can easily work with — a movie, an image, or an application.

## 10.1 Exporting a movie

Vuo offers several ways to create a movie from a composition:

- For an easy way to record the graphics displayed in a window, in the composition's menu go to File 〉 Start Recording.
- For the highest-quality rendering, make your composition use the Image Generator protocol, and in the Vuo Editor go to File 〉 Export 〉 Movie...
- To control the movie export from within your composition, use the **Save Images to Movie** node or the **Save Frames to Movie** node. (See each node's description for details.)
- To control the movie export from the command line, use the `vuo-export` command-line tool. (See Exporting a composition on the command line for details.)

### 10.1.1 Recording the graphics in a window

To record a movie:

- Run a composition that shows at least one window.
- If your composition has more than one window, click on the one you want to record to make it the active (frontmost) window.
- Go to File 〉 Start Recording. This immediately starts recording the movie.
- Let the composition run for as long as you want to record the movie. You can interact with the composition while it's recording.
- Go to File 〉 Stop Recording. This immediately stops recording the movie and presents a save dialog.
- In the save dialog, choose the file where you want to save your movie.

When you start recording, the graphics showing in the window at that moment are added as a frame in the movie. After that, each time the window being recorded renders some graphics — in other words, each time the **Render Image to Window**, **Render Layers to Window**, or **Render Scene to Window** node receives an event — a frame is added to the movie. If your composition is rendering about 60 frames per second, then your movie will play back at about 60 frames per second. If your composition renders once, then waits 10 seconds, then renders again, your movie will do the same — show the first frame for 10 seconds, then show the second frame.

The dimensions of the rendered movie match the dimensions of the window's graphics area at the moment when you start recording. If you resize the window while the recording is in progress, then the recorded images will be scaled to the movie's dimensions.

If your composition has multiple windows, then the active (frontmost) window at the time when you went to File 〉 Start Recording will be the one recorded. Only the content displayed within the window's graphics area — not the window's title bar, not the cursor, and not any audio — will be recorded in the movie.

Although recording from a composition window is an easy way to create a movie, and allows you to interact with the composition while the recording is being made, it does limit the quality of the movie. Recording a movie in real time means that your computer has to do extra processing, beyond just running the composition. Depending on how powerful your computer is, this may slow the composition down or make it render choppily, and do the same to the recorded movie.

The most reliable way to avoid slowness or choppiness is to export a movie from an Image Generator composition, as described in the next section. But if you do want to record from a composition window, here are some ways to improve the quality of your recording:

- Avoid doing other processor-intensive things on your computer (such as running other compositions) while the recording is in progress.
- Limit the size of the window that you record. (Larger windows require more processing power.)
- Avoid resizing the window during a recording. (Scaling the movie frames after the window has been resized requires more processing power.)

### 10.1.2 Exporting a movie from an Image Generator composition

Another way to create a movie from a composition is with File 〉 Export 〉 Movie... in the Vuo Editor. Instead of recording a composition in real time, this option runs the composition invisibly and takes as long (or short) as needed to render each movie frame. The resulting movie has a precise frame rate and no dropped frames. You can choose the start and end time, frame rate, and dimensions. Optionally, you can add antialiasing and motion blur (if you have Vuo Pro).

To export a movie:

- Make your composition conform to the Image Generator protocol.
- Go to File ⟩ Export ⟩ Movie....
- In the dialog that appears, choose the movie file to output to and the other settings for your movie.
- Click the Export button.

When exporting the movie, Vuo sends a series of events through the composition's published input ports. The **time** published input port value increases with each event, by an amount determined by the frame rate you chose in the dialog. Each event comes in through all published input ports. Vuo waits for the event to reach the **outputImage** published output port, and adds that image to the movie.

When setting up your composition to export a movie, make sure that, for each event that comes in through the published input ports, exactly one event goes out through the published output ports. If an event into the published input ports never reaches the published output ports, then the export will hang because Vuo will be stuck waiting for that event. If extra events go out through the published output ports (fired from triggers within the composition), then the exported movie may contain unexpected images.

## 10.2 Exporting an image

If you want to capture an image of a composition, you can either take a screenshot (open the Preview app and go to File ⟩ Take Screen Shot) or use the **Save Image** node (see the node's description for details).

## 10.3 Exporting an application

Using the File ⟩ Export ⟩ Mac App... menu item, you can turn your composition into an macOS application (*.app* file).

When exporting a composition that refers to files on your computer (such as images, scenes, or movies), typically the Vuo Editor will know to copy those into the exported app. If you've added these files to your composition by dragging them onto the canvas (without holding down ⌥) — creating a node such as **Fetch Image** or **Play Movie** — then the files will automatically be copied into the exported app. In fact, the Vuo Editor will automatically copy files and folders for all relative paths found in ports named **URL**, **URLs**, or **Folder** on nodes that read files.

If you've held down ⌥ while dragging a file onto the canvas, or if you've typed an absolute path into the input editor for a URL, then the Vuo Editor won't copy the file into the exported app. This is useful

if you want to refer to a file that you know will be in a certain location on every computer that runs the app, such as an image that comes with the operating system.

In some cases, you may want a file to be copied into the app, but the Vuo Editor may not be able to figure this out. This may happen, for example, if your composition uses an **Append Text** node to construct relative file paths out of smaller pieces. If the Vuo Editor doesn't copy your files into the exported app automatically, then you can copy them yourself. For example, if your composition uses a file called `image.png`:

- Place `image.png` in the same folder as your composition (.vuo file).
- In the Vuo Editor, go to File > Export > Mac App... and create `MyApp.app`.
- Right-click on `MyApp.app` and choose Show Package Contents.
- In the package contents, go to the `Contents▸Resources` folder. Copy `image.png` into that folder.

# 11 The Vuo Editor

## 11.1 The Node Library

When you create a composition, your starting point is always the **Node Library** (Window > Show Node Library). The node library is a tool that will assist you in exploring and making use of the collection of Vuo building blocks ("nodes") available to you as you create your artistic compositions.

Because you'll be working extensively with the node library throughout your composition process, we have put a great deal of effort into maximizing its utility, flexibility, and ease of use. It has been designed to jump-start your Vuo experience — so that you may sit down and immediately begin exploring and composing, without having to take time out to study reams of documentation.

When you open a new composition, the Node Library is on the left. 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.

**Note for Quartz Composer users**

Many of the same shortcuts from Quartz Composer also work in Vuo. As an example, ⌘↵ opens the Node Library, and ⌘R begins playback of your composition.

### 11.1.1 Docking and visibility

By default, the node library is docked within each open composition window. The node library may be undocked by dragging or double-clicking its title bar. While undocked, only a single node library will be displayed no matter how many composition windows are open.

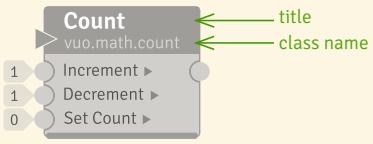
The node library may be re-docked by double-clicking its title bar.

The node library may be hidden by clicking the X button within its title bar. Once hidden, it may be redisplayed by selecting Window > Show Node Library or using ⌘↵. The same command or shortcut, ⌘↵, will put your cursor in the node library's search window.

Whether you have left your library docked or undocked, visible or hidden, your preference will be remembered the next time you launch the Vuo Editor.

### 11.1.2 Node names and node display

Each node has two names: a title and a class name. The **title** is a quick description of a node's function; it's the most prominent name written on a node. The **class name** is a categorical name that reveals specific information about a node; it appears directly below the node's title.




Let's use the **Count** node as an example. "Count" is the node's title, which reveals that the node performs the function of counting. The class name is "vuo.math.count". The class name reveals the following: Team Vuo created it, "math" is the category, and "count" is the specific function (and title name).

Depending on your level of familiarity with Vuo's node sets and your personal preference, you might wish to browse nodes by their fully qualified family ("class") name (e.g., "vuo.math.add") or by their more natural human-readable names ("Add").

You may select whichever display mode you prefer, and switch between the modes at your convenience; the editor will remember your preference between sessions. You can toggle between node titles and node class names using the menu items View > Node Library > Display by class or Display by name.

The Modifying and rearranging nodes and cables section explains how to change node titles.

### 11.1.3 Node Documentation Panel

The node library makes the complete set of Vuo core nodes available for you to browse as you compose. By clicking on a node in the library, a description of the node will appear in the **Node Documentation Panel** below the node library. It describes the general purpose of the node as well as details that will help you make use of it. In addition to the Vuo core nodes, if you have access to pro nodes, you'll see those displayed.

If you're interested in exploring new opportunities, this is an ideal way to casually familiarize yourself with the building blocks available to you in Vuo.

### 11.1.4 Finding nodes

In the top of the Node Library there is a search bar. You can type in part of a node name or a keyword and matching nodes will show up in the Library. Pressing ⌫ while in the search bar will clear out your selection and show the entire library, as will deleting your search term.

Your search terms will match not only against the names of relevant nodes, but also against keywords that have been specifically assigned to each node to help facilitate the transition for any of you who might have previous experience with other multimedia environments or programming languages.

For example, users familiar with multiplexers might type "multiplex" into the Vuo Node Library search field to discover Vuo's "Select Input" family of nodes with the equivalent functionality; users with a background in textual programming might search for the term "string" and discover the Vuo "Text" node family. Users don't have to know the exact node title or port name. To find a node with a trigger port, for example, go to the Node library and type in the keywords "events," "trigger," or "fire."

If you do not see a node, particularly if you have received it from someone else, review the procedures under Installing a node.

## 11.2 Working on the canvas

### 11.2.1 Putting a node on the canvas

The node library isn't just for reading about nodes, but for incorporating them into your compositions. Once you have found a node of interest, you may create your own copy by dragging it straight from the node library onto your canvas, or by double-clicking the node listing within the library.

Not a mouse person? Navigating the library by arrow key and pressing ↵ to copy the node to your canvas works just as well.

You may copy nodes from the library individually, or select any number or combination of nodes from the library and add them all to your canvas simultaneously with a single keypress or mouse drag — whatever best suits your work style.

### 11.2.2 Drawing cables to create a composition

You can create a cable by dragging from a node's output port to a compatible input port or from a node's output port to a compatible input port.

Compatible ports are those that output and accept matching or convertible types of data. Compatible ports are highlighted as you drag your cable, so you know where it's possible to complete the connection.

If you complete your cable connection between two ports whose data types are not identical, but that are convertible using an available type converter (e.g., vuo.math.round for rounding real numbers to integers), that type converter will be automatically inserted when you complete the connection.

Sometimes existing cables may also be re-routed by dragging (or "yanking") them away from the input port to which they are currently connected. It is possible to yank the cable from anywhere within its **yank zone**. You can tell where a cable's yank zone begins by hovering your cursor near the cable. The yank zone is the section of the cable with the extra-bright highlighting. If no yank zone is highlighted, you'll need to delete and add back the cable.

### 11.2.3 Copying and pasting nodes and cables

You can select one or more nodes and copy or cut them using the Edit > Copy and/or Edit > Cut menu options, or their associated keyboard shortcuts. Any cables or type converters connecting the copied nodes will automatically be copied along with them.

You can paste your copied components into the same composition, a different composition, or a text editor, using the Edit > Paste menu option or its keyboard shortcut.

**+ Tip**

Select one or more nodes and drag them while holding down ⌥ to duplicate and drag your selection within the same composition. Press ⎋ during the drag to cancel the duplication.

### 11.2.4 Deleting nodes and cables

Delete one or more nodes and/or cables from your canvas by selecting them and either pressing ⌫ or right-clicking one of your selections and selecting Delete from its context menu.

When you delete a node, any cables connected to that node are also deleted. A cable with a yank zone may also be deleted by yanking it from its connected input port and releasing it.

Any type converters that were helping to bridge non-identical port types are automatically deleted when their incoming cables are deleted.

### 11.2.5 Modifying and rearranging nodes and cables

You can move nodes within your canvas by selecting one or more of them and either dragging them or pressing the arrow keys on your keyboard.

You can change the constant value for an input port by double-clicking the port, then entering the new value into the input editor that pops up. (Or you can open the input editor by hovering the cursor over the port and pressing ↵.) When the input editor is open, press ↵ to accept the new value or ⎋ to cancel.

Input editors take on various forms depending on the data type of the specific input being edited — they may present as a text field, a menu, or a widget (such as color picker wheel), for example.

Some ports take lists as input. These ports have special attached "drawers" containing 0 or more input ports whose values will make up the contents of the list. Drawers contain two input ports by default, but may be resized to include more or fewer ports by dragging the "drag handle."

You can change how a trigger port should behave when it's firing events faster than downstream nodes can process them. Do this by right-clicking on the port, selecting Set Event Throttling from its context menu, and selecting either "Enqueue Events or Drop Events.

You can change a node's title (displayed at the top of the node) by double-clicking or hovering over the title and pressing ↵, then entering the new title in the node title editor that pops up. You may save or dismiss your changes by pressing ↵ or ⎋, respectively, just as you would using a port's input editor. You can also select one or more nodes from your canvas and press ↵ to edit the node titles for each of the selected nodes in sequence. If you delete the title and don't enter a new title, the node will default to its original title.

You can change a node's tint color by right-clicking on the node, selecting Tint from its context menu, and selecting your color of choice. Tint colors can be a useful tool in organizing your composition. For example, they can be used to visually associate nodes working together to perform a particular task.

### 11.2.6 Viewing a composition

If your composition is too large to be displayed within a single viewport, you can use the Zoom buttons within the composition window's menubar, or the View > Actual Size / Zoom In / Zoom Out / Zoom to Fit menu options, to adjust your view. You can use the scrollbars to scroll horizontally or vertically within the composition. Alternatively, if you have no nodes or cables selected, you can scroll by pressing the arrow keys on your keyboard. You can also grab the workspace by holding down the spacebar while dragging.

**+ Tip**

Hold down ⇧ while pressing an arrow key to scroll even faster.

### 11.2.7 Publishing ports

A composition's published ports are displayed in sidebars, which you can show and hide using the menu Window > Show/Hide Published Ports.

You can publish any input or output port in a composition. Do this by right-clicking on the port and selecting Publish Port from the context menu. Alternatively, drag a cable from the port to the Publish well that appears in the sidebar when you start dragging. You can unpublish the port by right-clicking on the port again and selecting Unpublish Port.

In the sidebars, you can rename a published port by double-clicking on the name or by right-clicking on the published port and selecting Rename Published Port. You can reorder published ports (except those that are part of a protocol) by dragging the name of a published port up or down in the sidebar.

For published ports with numerical data types (integers, real numbers, 2D points, 3D points, and 4D points), you can modify the behavior of their input editors by right-clicking on the published port in the sidebar and selecting Edit Details... The Suggested Min and Suggested Max determine the range of values provided by the input editor's slider or spinbox (arrow buttons). The Suggested Step controls the amount by which each click on a spinbox button increments or decrements the value.

+ Tip

If you copy a node with a published port, that port will be published under the same name (if possible) in whatever composition you paste it into. The published port will be created if it does not already exist, merged if an existing published port of the same name and compatible type does exist, or renamed if an identically named published port already exists but has an incompatible type.

### 11.2.8 Using a protocol for published ports

To create a composition with a predetermined set of published ports defined by a protocol, go to the File menu, select New Composition from Template, and select the protocol you want. Typically, a protocol is used when running a Vuo composition inside another application, such as a VJ or video postproduction app. That application should instruct you about the protocol to select.

The published ports in a protocol appear in a tinted area of the published port sidebars, with the protocol name at the top. You can't rename or delete these published ports. However, you can add other published ports to the composition and rename or delete them as usual.

## 11.3 Running a composition

After you've built your composition (or while you're building it), you can run it to see it in action.

### 11.3.1 Starting and stopping a composition

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.)

If you start a composition that was created using New Composition from Template, then extra functionality will be added to the composition to help you preview it. Its protocol published input ports will receive data and events, and its protocol published output ports will send their data and events to a preview window. For example, if you run a composition with the Image Filter protocol, then image and time data will be fed into the composition, and the composition's image output will be rendered to a window.

### 11.3.2 Firing an event manually

As you're editing your running composition, you may want to fire extra events so that your changes become immediately visible, rather than waiting for the next time a trigger port happens to fire.

You can cause a trigger port to fire an event by right-clicking on the trigger port to pop up a menu, then choosing Fire Event. Or you can hold down ⌘ while left-clicking on the trigger port. If the trigger port carries data, it outputs its most recent data along with the event.

You can also fire an event directly into an input port (as if it had an incoming cable from an invisible trigger port). To do this, you can right-click on the input port and choose Fire Event, or you can hold down ⌘ and left-click on the input port.

If you've already manually fired an event, you can fire another event through the same port by going to Run > Re-fire Event. This fires an event through the trigger port or input port that most recently had an event manually fired.

### 11.3.3 Understanding and troubleshooting a running composition

The Vuo Editor has several helpful tools, such as the **Show Events** mode and **Port popovers** to help you understand in more depth what is happening in your composition. These features can help you see exactly what events and data are flowing through your composition. For more information on troubleshooting steps see the section on Troubleshooting.

## 11.4 Working with subcompositions

With a subcomposition, you can use a composition as a node within other compositions. For more on what subcompositions are and why to use them, see Using subcompositions inside of other compositions.

### 11.4.1 Installing a subcomposition

To install a subcomposition, open or create a composition within the Vuo Editor and select the File > Move Composition to Node Library menu item. (If your composition has not yet been saved, the menu item will read Save Composition to Node Library, and you'll be prompted to enter a title for your node.) The subcomposition node will immediately be listed and highlighted within your Node Library for use within other compositions.

### 11.4.2 Editing a subcomposition

There are several ways to edit a subcomposition after it has already been installed:

- Right-click on the subcomposition node, either within the Node Library or on the canvas, and select the Edit Composition... context menu item.
- Double-click on the body of the node on the canvas.
- Select the node on the canvas and press ⌘↓.
- Click the "Edit Composition" link in the node library documentation panel.

### 11.4.3 Uninstalling a subcomposition

To remove an installed subcomposition, you can right-click on the subcomposition node within the Node Library and select the Open Enclosing Folder context menu item, or follow the instructions in the Installing a node section to navigate to your User Modules folder. Locate the `.vuo` file matching the name of your subcomposition and remove it from the folder. The next time you restart the Vuo Editor, the subcomposition will no longer appear in your Node Library.

## 11.5 Keyboard Shortcuts

Vuo has keyboard shortcuts for working with your composition and the Vuo Editor.

In the keyboard shortcuts below, these symbols represent keys in macOS:

| Symbol | Definition |
|---|---|
| ⌘ | Command key |
| ^ | Control key |
| ⌥ | Option key |
| ⇧ | Shift key |
| ⌫ | Delete key |
| ↵ | Return key |
| ⎋ | Escape key |

### 11.5.1 Working with composition files

| Shortcut | Definition |
|---|---|
| ⌘N | New Composition |
| ⌘O | Open Composition |
| ⇧⌘O | Open the most recent composition |
| ⌥⌘O | Open a random example composition |
| ⌘S | Save Composition |
| ⇧⌘S | Save Composition As |
| ⌘W | Close Composition |

### 11.5.2 Controlling the composition canvas

| Shortcut | Definition |
|---|---|

| | |
|---|---|
| ⌘= | Zoom In |
| ⌘- | Zoom Out |
| ⌘9 | Zoom to Fit |
| ⌘0 | Actual Size |
| ⌘↵ | Show Node Library |
| Spacebar Drag | Move the canvas viewport |
| ⌘1 | Set canvas transparency to None |
| ⌘2 | Set canvas transparency to Slightly Transparent |
| ⌘3 | Set canvas transparency to Very Transparent |

### 11.5.3 Creating and editing compositions

| Shortcut | Definition |
|---|---|
| ⌘A | Select all |
| ⇧⌘A | Select none |
| ⌘C | Copy |
| ⌘V | Paste |
| ⌘X | Cut |
| ⌘Z | Undo |
| ⇧⌘Z | Redo |
| ⌘F | Find |
| ⌘G | Find Next |
| ⇧⌘G | Find Previous |
| ⌫ | Delete |
| ⌥ Drag | Duplicate the cable connected to the input port near the mouse cursor, or duplicate the selected nodes and cables. |
| ⇧ Drag | Change the data-and-event cable being dragged to event-only. |
| ↑↓←→ | Move nodes and cables around on the canvas. Hold ⇧ to move further. |
| ↵ | Hover over a node title and press ↵ to edit it. |

| | |
|---|---|
| ↵ | Select one or more nodes and press ↵ to edit their titles. |
| ↵ | Hover the mouse over a constant value and press ↵ to edit it.<br>Press ↵ to accept the new value, or ⎋ to go back to the old value. |
| ⌥↵ | Open a Text input editor and press ⌥↵ to add a linebreak. |

### 11.5.4 Running compositions (when Vuo Editor is active)

| Shortcut | Definition |
|---|---|
| ⌘. | Stop |
| ⌘R | Run |
| ⇧⌘R | Restart |
| ⌘ Click | Do this on an input port or a trigger port to manually fire an event. |
| ⌘T | Re-fire Event |

### 11.5.5 Running compositions (when the composition is active)

| Shortcut | Definition |
|---|---|
| ⌘Q | Stop the composition |
| ⌘F | Toggle between windowed and fullscreen |
| ⌥⌘E | Toggle recording the composition's graphical output to a movie file |

### 11.5.6 Application shortcuts

| Shortcut | Definition |
|---|---|
| ⌘Q | Quit the Vuo Editor |
| ⌘H | Hide the Vuo Editor |

# 12 The command-line tools

As an alternative to using the Vuo Editor, you can use command-line tools to work with Vuo compositions. Although most Vuo users will only need the Vuo Editor, you might want to use the command-line tools if:

- You're using the open source version of Vuo, which doesn't include the Vuo Editor.
- You're writing a program or script that works with Vuo compositions. (Another option is the Vuo API.)
- You're working with Vuo compositions in a text-only environment, such as SSH.

A Vuo composition (.vuo file) is actually a text file based on the Graphviz DOT format. You can go through the complete process of creating, compiling, linking, and running a Vuo composition entirely in a shell.

## 12.1 Installing the Vuo SDK

- Go to https://vuo.org/user and log in to your account
- Click the Download tab
- Under the Vuo SDK section, download the Vuo SDK
- Uncompress the ZIP file (double-click on it in Finder)
- Move the folder wherever you like

Do not separate the command-line binaries (`vuo-compile`, `vuo-debug`, `vuo-link`, `vuo-render`) from the Framework (`Vuo.framework`) — in order for the command-line binaries to work, they must be in the same folder as the Framework.

Next, add the command-line binaries to your PATH so you can easily run them from any folder.

- In Terminal, use `cd` to navigate to the folder containing the Vuo Framework and command-line binaries
- Run this command:

```
echo "export PATH=\$PATH:$(pwd)" >> ~/.bash_profile
```

- Close and re-open the Terminal window

## 12.2 Getting help

To see the command-line options available, you can run each command-line tool with the `--help` flag.

## 12.3 Rendering a composition on the command line

Using the `vuo-render` command, you can render a picture of your composition:

**Listing 1: Rendering a composition**

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

`vuo-render` can output either PNG (raster) or PDF (vector) files. The command `vuo-render --help` provides a complete list of parameters.

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

**Listing 2: Rendering a Vuo composition using Graphviz**

```
dot -Grankdir=LR -Nshape=Mrecord -Nstyle=filled -Tpng -oRenderTextLayer.png RenderTextLayer.vuo
```

## 12.4 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 3: Compiling a Vuo composition**

```
vuo-compile --output RenderTextLayer.bc RenderTextLayer.vuo
```

Then, turn the .bc file into an executable:

**Listing 4: Linking a Vuo composition into an executable**

```
vuo-link --output RenderTextLayer RenderTextLayer.bc
```

If you run into trouble building a composition, you can get more information by running the above commands with the `--verbose` flag.

If you're editing a composition in a text editor, the `--list-node-classes=dot` flag is useful. It outputs all available nodes in a format that you can copy and paste into your composition.

## 12.5 Running a composition on the command line

You can run the executable you created just like any other executable:

**Listing 5: Running a Vuo composition**

```
./RenderTextLayer
```

Using the `vuo-debug` command, you can run the composition and get a printout of node executions and other debugging information:

**Listing 6: Running a Vuo composition**

```
vuo-debug ./RenderTextLayer
```

## 12.6 Exporting a composition on the command line

Using the `vuo-export` command, you can turn a composition into a movie or an application:

**Listing 7: Exporting a Vuo composition to a movie**

```
vuo-export movie --output GenerateCheckerboardImage.mov GenerateCheckerboardImage.vuo
```

**Listing 8: Exporting a Vuo composition to an application**

```
vuo-export macosx --output RenderTextLayer.app RenderTextLayer.vuo
```

If you run into trouble exporting a composition, you can get more information by running `vuo-export` with the `--verbose` flag.

This command is equivalent to the File › Export › Mac App... menu item in Vuo Editor. See the section Exporting an application for more information.

# 13 Troubleshooting

What if you run into problems using Vuo? This section describes some common problems and how you can fix them. If you need help, you can always check out the additional resources on the Vuo Support page.

## 13.1 Helpful tools

The Vuo Editor provides several helpful tools for troubleshooting:

- **Show Events** mode lets you watch the events flow through your composition. You can turn it on and off with the Run > Show Events and Run > Hide Events menu items. In Show Events mode, trigger ports are animated as they fire events. Nodes turn opaque as they're executed and gradually become more transparent as time passes since their most recent execution. Using Show Events mode, you can see if certain parts of your composition are executing.
- **Port popovers** let you inspect the data and events flowing through individual ports. A port popover pops up when you click on a port. If you want to keep the port popover open for a while, for example to look at several port popovers at once, click on the popover. While the composition is running, the port popover shows several pieces of information that can help with debugging:
  - Last event — The time of the most recent event through the port, and the average number of events per second.
  - Value — For data-and-event ports only, the most recent data through the port.
  - Event throttling — For trigger ports only, whether the port enqueues or drops events.
- **Node descriptions** tell you how the node is supposed to work. The node description appears in the lower panel of the Node Library whenever you select the node in the Node Library or on the canvas.

There are nodes to help with troubleshooting, too. One is **Display Console Window**, which shows a text window that your composition can write text on. You can use **Display Console Window** to observe values that are hard to see in port popovers because they're changing too rapidly. To find other nodes that can help with troubleshooting, search the Node Library for "debug" or "troubleshoot".

+ Tip

When watching data with the **Display Console Window** node, you can use the **Allow Changes** node to filter out repeated data.

## 13.2 Common problems

### 13.2.1 My composition isn't working and I don't know why.

The first step is to take a deep breath and relax! OK, now the second step is to understand the problem. Here are some questions to ask yourself (or go through with a friend or collaborator):

- What do you expect the composition to do?
- What is the composition doing instead?
- Where in the composition does the problem begin?

Using the tools provided the Vuo Editor, try to narrow down the problem. Figure out exactly which nodes aren't working as you expect. Then try some of the more specific troubleshooting steps in the rest of this section.

### 13.2.2 Some nodes aren't executing.

If a node doesn't become opaque in Show Events mode, or if its port popover says "Last Event: (none observed)", then the node isn't executing. If a node isn't executing, that means events aren't reaching it. Here are some things to check:

- Is there a trigger port connected to the node? Trace backward through your composition, starting at the node that isn't executing, and looking at the cables and nodes feeding into it. Do you find a trigger port? If not
  - Add a node with a trigger port, such as **Fire on Start**, and connect the trigger port to the node that isn't executing.
- Is the trigger port firing? Check the trigger port's popover (or connect a **Count** node, as described above). If the trigger isn't firing
  - Check the node description for the trigger port's node. Make sure you understand exactly when the trigger is supposed to fire.
  - Check the trigger port's event throttling, which is displayed in the port popover. If it says "drop events", try changing it to "enqueue events". (See the section Controlling the buildup of events.)
- Are events from the trigger port reaching some nodes but not others? Trace forward through your composition, from the trigger port toward the node that isn't executing, and find the last node that's receiving events.

  - Look at the input ports on that last node. Do they have walls or doors? (See the section Event walls and doors.) Check the node's description to help you understand when and why the node blocks events. To send events through the node, you may need to connect a cable to a different input port.
  - Look at the output ports on that last node. Are they trigger ports? Remember that events into input ports never travel out of trigger ports. To send events through the node, you may need to connect a cable to a different output port.

### 13.2.3 Some nodes are executing when I don't want them to.

A node executes every time an event reaches it. If you don't want the node to execute at certain times, then your composition needs to block events from reaching the node. For more information, see the section Common patterns - "How do I".

### 13.2.4 Some nodes are outputting the wrong data.

If your composition is outputting graphics, audio, or other information that's different from what you expected, then you should check the data flowing through your composition. Here are some things to check:

- Where exactly does the data go wrong?
  - Check each port popover along the way to see if it has the data you expected.
  - Add some nodes to the middle of the composition to help you check the data (for example, a **Render Image to Window** node to check the data in an image port).
- Is there a node whose output data is different than you expected, given the input data?
  - Read the node description carefully. The node might work differently than you expected.

### 13.2.5 The composition's output is slow or jerky.

This can happen if events are not flowing through your composition often enough or quickly enough. Here are some things to check:

- Is each trigger port firing events as often as you expected? Check its port popover to see the average number of events per second. If it's firing more slowly than you expected
  - Check the node description for the trigger port's node. Make sure you understand exactly when the trigger is supposed to fire.

  - Check for any nodes downstream of the trigger port that might take a long time to execute, for example a **Fetch Image** node that downloads an image from the internet. Change your composition so those nodes receive fewer events. (See the section Common patterns - "How do I".)
  - Check the trigger port's event throttling, which is displayed in the port popover. If it says "drop events", try changing it to "enqueue events". (See the section Controlling the buildup of events.)
  - Check the event throttling of each other trigger port that can fire events through the same nodes as this trigger port. If the other trigger port's event throttling is "enqueue events", try changing it to "drop events".

- Is each node receiving events as often as you expected? If not
  - Check if there are any event doors that might be blocking events between the trigger and the node. (See the section Event walls and doors.)
- Is the composition using a lot of memory or CPU? You can check this in the Activity Monitor application. If so
  - Check if any parts of the composition are executing more often than necessary, and try not to execute them as often. (See the section Common patterns - "How do I".)
  - Export the composition to an application. When run as an application instead of in Vuo Editor, compositions use less memory and CPU.
  - Quit other applications to make more memory and CPU available.
  - Run the composition on a computer with more memory and CPU.

## 13.3 General tips

Finally, here are a few more tips to help you troubleshoot compositions:

- If you're having trouble with a large and complicated composition, try to simplify the problem. Create a new composition and copy a small piece of your original composition into it. It's much easier to troubleshoot a small composition than a large one.
- If you're having trouble with a composition that has rapidly firing trigger ports, try to slow things down. For example, in place of the **Render Scene to Window** node's **Requested Frame** trigger port, use a **Fire Periodically** node's **Fired** trigger port connected to a **Count** node's **Increment** port.
- If your composition used to work but now it doesn't, figure out exactly what changed. Did you add or remove some cables? Were you using a different version of Vuo? Knowing what changed will help you narrow down the problem.
- Check the Console application (in your Finder application folder, Utilities > Console.app) while running your composition. Some nodes send console messages when they have problems.

- Try rearranging your nodes and cables so you can see the flow of events more clearly. If your nodes and cables are nicely laid out, then it can be easier to spot problems.
- Don't be afraid to experiment (but first save a copy of your composition). If you're not sure if a node is working as you expect, try it with various inputs.
- Don't be afraid to ask questions. For help, go to Vuo Support.

# 14 Contributors

Vuo is built and maintained by Team Vuo and the Vuo Community. Everyone is encouraged to contribute toward improving Vuo.

## 14.1 Contributors

Below is an alphabetical list of the people who have contributed to bringing Vuo to fruition.

- .lov.
- 2bitpunk
- 3lab_tv
- ajm
- akashaslc
- alexmitchellmus
- Anthony
- architek1
- ariam
- atompowered
- automatone
- a_o
- baksej
- balam
- Benedikt
- bLackburst
- bmellen
- Bodysoulspirit
- Bonemap
- botnotbot
- casdekker
- cremaschi
- cwilms-loyalist
- cwright
- cymaspace
- David
- ddelcourt
- destroythings

- Doro
- dumski
- e.duchemin
- eganpc
- ellington
- emervark
- errol
- Eurotrash
- franz
- fRED
- gabe
- gautjac
- George_Toledo
- Gillieron
- iason
- Illuminator
- inadvisable
- inx
- Jérôme Lanon
- jersmi
- jinyaolin
- jmcc
- Jobok31
- joeladria
- johnnykuo
- jokkeheikkila
- jstrecker
- jte2384
- jungbas
- jvolker
- Kewl
- khenkel
- kozistan
- krezrock
- Landscaper
- lhepner
- lipoqil
- Luiz Andre
- manuel_mitasch

- marioepsley
- MartinusMagneson
- mattgolsen
- meno
- miramon9
- mixfilet
- mkegan
- mnstri
- mradcliffe
- mrray
- mutable
- p8guitar
- pbourke
- Pianomatic
- prackvj
- pyramus
- raphael
- rbetin
- richardbyers
- rmercuri
- robaiello
- ryandmonk
- sala28
- Salvo
- savienojums
- sboas
- Scratchpole
- seanradio
- shakinda
- Sigve
- SimHacker
- sinemod
- sinsynplus
- smokris
- Steamboy
- steinboy
- stromqvist
- synnack
- Taco Circus

- teaportation
- tfrank
- timwessman
- tivonrice
- tmoles
- tobyspark
- unfenswinger
- unicode
- useful design
- vjsatoshi
- volkerku
- WARP
- wmackwood
- Xavier dev
- xoanxil
- zwei-p
- zzkj

Thanks be to our contributors!

## 14.2 Software Vuo uses

- Apple Csu
- Apple dyld
- Apple ld64
- BeatDetektor
- Clang
- Conan
- Discount
- DocBook
- Doxygen
- FFmpeg
- FreeImage
- Gamma
- Ghostscript
- Graphviz
- Hap
- JSON-C

- LLVM
- LaTeX
- Leap Motion
- Open Asset Import
- OpenSSL
- Pandoc
- Qt
- RtAudio
- RtMidi
- Snappy
- Squish
- Syphon
- YCoCg-DXT
- ZXing
- csgjs-cpp
- glib
- http-parser
- libcsv
- libcurl
- libffi
- libfreenect
- libintl (gettext)
- libjpeg-turbo
- liblqr
- libusb
- libxml2
- muParser
- nginx
- oscpack
- pngquant
- zlib
- ØMQ

## 14.3 Resources Vuo uses

- Signika