

Dataflow Programming

- [Dataflow Programming](#)

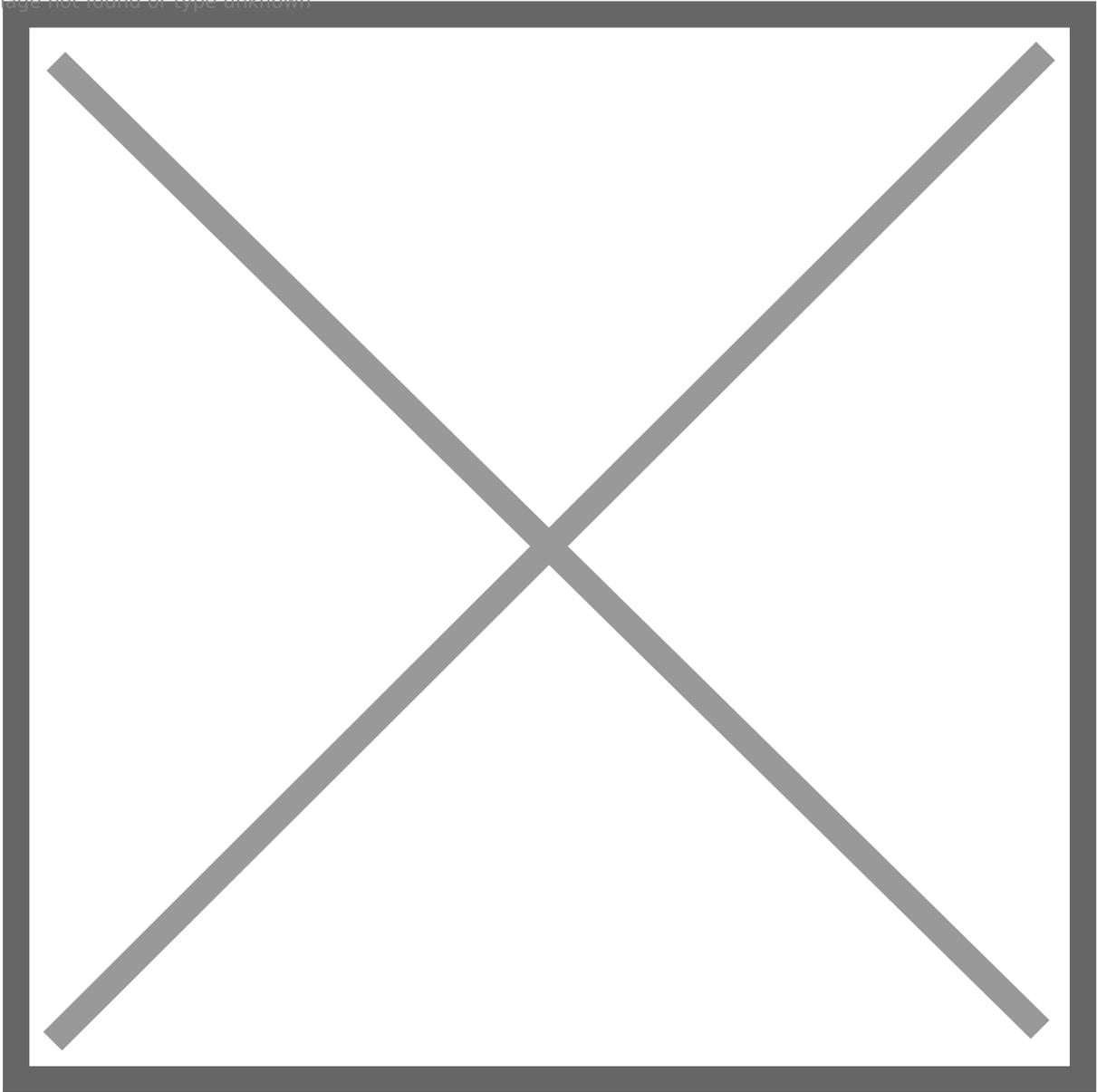
Dataflow Programming

<https://medium.com/@olegalexander/how-to-write-large-programs-628c90a70615>

Dataflow Programming

While modular programming can help you build a single large program, [dataflow programming](#) can help you build a large pipeline of many interconnected programs. As a former technical artist, dataflow programming is very near and dear to my heart. Over half of all the code I've ever written fits under this category. Dataflow programming is very common in the VFX, 3D Animation, and Video Game industries. In these industries, you can't throw a rock (and I mean literally) without hitting a 3D artist working in some kind of dataflow program. Popular dataflow programs are Maya, Nuke, Substance Designer, and Houdini. These programs are often called "node-based", "non-destructive", or "procedural".

Image not found or type unknown

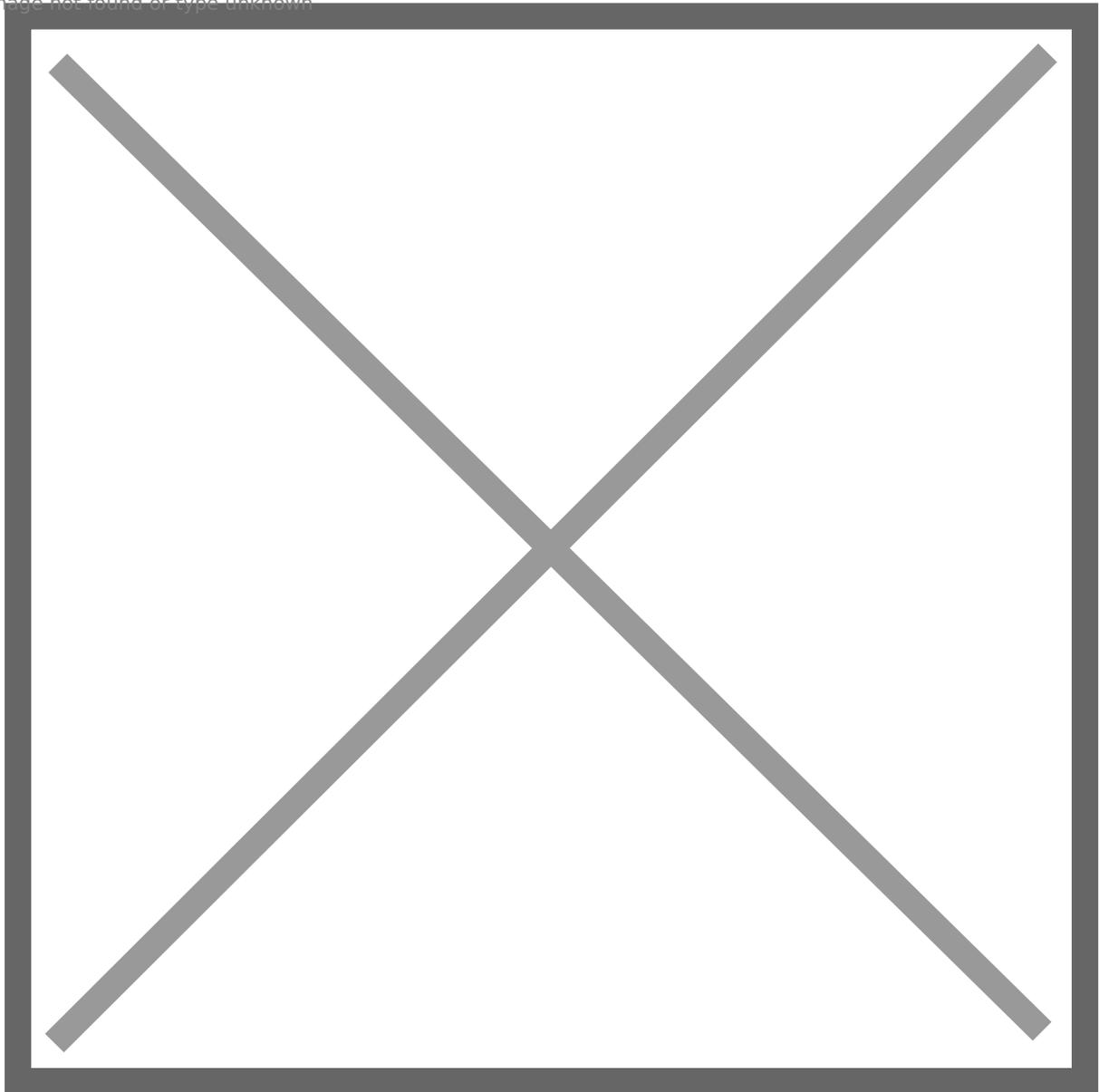


A typical dataflow pipeline

Let's define exactly what dataflow programming is:

- In dataflow programming, the program is decomposed into black box processes called *nodes*. These nodes have a set of inputs and a set of outputs. Nodes transform inputs into outputs in some way. For example, in Nuke, you could load an image using a Read node, then resize that image to quarter resolution using a Reformat node, then save out the smaller image using a Write node. The original input image is never overwritten, and this is why dataflow programming is called non-destructive editing.

Image not found or type unknown



A simple Nuke pipeline to resize an image

- Nodes are arranged into a “pipes and filters” pipeline, similar to a manufacturing assembly line, where the pipes carry data and the filters are process nodes. A dataflow pipeline always forms a directed acyclic graph (DAG).
- Nodes are executed in topological sort order from upstream to downstream. Changing any of the inputs in an upstream node automatically recomputes all downstream nodes. In this way, we say that the data is *flowing through* the nodes.
- While dataflow programming and functional programming are similar, there are a few important differences. First, in dataflow programming, the structure of the DAG is specified externally in some runtime environment. Nodes aren’t aware of each other, whereas in functional programming functions can call other functions. Second, dataflow programming doesn’t allow recursion. Third, dataflow programming is typically set up for parallel execution, whereas in functional programming parallel execution is not a given.
- Nodes usually have their own parameters. These parameters are often stored externally together with the DAG. Sometimes the parameters are computed using other nodes or expressions.

- Nodes are coupled *only* by data, which makes them endlessly reconfigurable. At any moment, the artist can inspect the outputs of a specific node. This leads to a deep understanding of every step of the process. Skilled artists can build node networks of incredible complexity without ever writing a single line of code. This ability for artists to do “visual programming” is probably why dataflow programming is so attractive to them.

Dataflow programming extends beyond a desktop application runtime like Maya. Bigger runtimes, called Render Farm Management Software, exist to orchestrate massive distributed pipelines of renderers and other command line tools. (If you’re more comfortable with Web tech than VFX tech, check out [Apache Airflow](#).) And these are the kinds of pipelines that I’m often tasked with designing and writing.

But how do you write a command line tool which can be plugged into a dataflow pipeline? What are the rules for writing such a tool well? Here’s a list of rules I follow:

1. **The tool should not have an interactive prompt.** The tool cannot be interactive because the Render Farm Management Software, which will run the tool, is an automated process. Therefore, the tool can only accept command line arguments.
2. **The tool must be like a pure function.** The only difference is that the data exists on disk instead of in memory. For example, if you were to write a command line tool to composite image A over image B, it could have the following specification: `over <pathToImageA> <pathToImageB> <unpremultiply> <pathToOutputImage>`.
3. **The tool must fail gracefully.** Error handling can be done in two ways: exit codes and logging. The exit code is programmatic, while the logging is meant for humans. Exit code 0 always means success. The meaning of other exit codes should be documented. In the image compositing example above, the exit codes might be: `0: OK, 1: IMAGE_A_HAS_NO_ALPHA_CHANNEL, 2: INCOMPATIBLE_IMAGE_DIMENSIONS, 3: INVALID_IMAGE_FORMAT, 4: IMAGE_DOES_NOT_EXIST, 5: CANNOT_WRITE_OUTPUT, 6: CANNOT_OVERWRITE_INPUTS, etc.`
4. **The tool must be idempotent.** Running the tool more than once with the same inputs should always produce the same outputs. You should assume the tool *will* be run more than once due to retries. The tool can *never* overwrite the inputs! And it should *always* overwrite the outputs! None of this `--overwrite` flag BS.
5. **The tool should be as dumb as possible.** It should never try to massage invalid inputs to make them work.
6. **The tool must always exit.** I once worked with a third-party tool which said “Press any key to exit” at the end. Please don’t do that.
7. **The tool should not be aware of any other running processes or frameworks.** It’s up to the Render Farm Management Software to manage dependencies between processes. Wrapping a third-party command line tool (or several) into a single process is okay. Spawning threads is also okay.
8. **Avoid data collisions.** The tools are arranged into DAG pipelines in the Render Farm Management Software. The DAGs themselves are usually parameterized so that different instances of the same DAG can run in parallel. It’s important that data in one DAG instance is completely isolated from data in another DAG instance. All you have to do is put the data for each DAG instance into a separate folder. Also, you should prevent a DAG instance with the same parameters from being created twice.

9. **Avoid data corruption.** What if you have a batch process which somehow touches data in all of the DAG instance folders? In that case, you must **stop** all running DAG instances, run the batch process, and then restart the DAG instances again. Think of it as a crosswalk. The DAG instances are the cars and the batch process is the pedestrian wishing to cross the street. Bad things will happen if the pedestrian doesn't wait for the cars to stop.
10. **Updating an upstream node must automatically update all downstream nodes.** Never rerun a single node upstream (with different parameters) without also rerunning all downstream nodes in topological sort order. If your Render Farm Management Software doesn't come with a "Requeue Downstream Jobs" feature, make sure to write this feature yourself.

That's all there is to it! I can tell you from hard-earned experience that breaking any of these rules will lead to data corruption. But with practice, these rules will become second nature to you.

Dataflow programming is my go-to decomposition technique whenever there is a stream of data flowing from one process to the next. The first thing I do when designing a pipeline is draw a [Data Flow Diagram](#). Once I'm confident that I understand both the data and the processes involved, test data can be gathered and implementation of the processes can begin. If the data is clearly defined, then multiple developers (using potentially different languages) can work in parallel.