Visual programming of ImageJ using ImageFlow

Kai Uwe Barthel and Daniel Senff

HTW Berlin, Wilhelminenhofstraße 75A, 12459 Berlin, Germany

ABSTRACT

ImageFlow is a plugin that allows the generation and execution of macros in ImageJ based on the visual programming paradigm. Image processing steps are described by connected nodes on a workspace. Specifying image-processing algorithms corresponds to visually editing block diagrams (the workflow of successive processing steps). This abstraction makes it very flexible and easy to save, restore and to change processing workflows afterwards. ImageFlow is a tool, which gives the user a lot of flexibility to create workflows and extend its original functionality by creating new processing nodes to include individual plugins or to encompass more macro-functionality within the nodes. The workflow execution is based on ImageJ's macro interface and thereby is an alternative and more visual user interface to build macro scripts.

In the first half of this workshop we will show the potential of ImageFlow by creating several example workflows and displaying various functions the application has to offer. The second half will be about customizing ImageFlow and will introduce ways to create individual processing-nodes and how to make your plugin fit for ImageFlow.

Keywords: ImageJ, Macro generation, Visual programming

1. INTRODUCTION

ImageFlow abstracts the actions that are performed on an image. The center of attention is not the image itself anymore, but the number and order of processing steps performed on this image. These steps are described in a graph, the workflow. Instead of focusing on the image, the focus is in building the workflow that is performed on the image. Every processing step is described by a *Node* element. Every node can have an individually specified
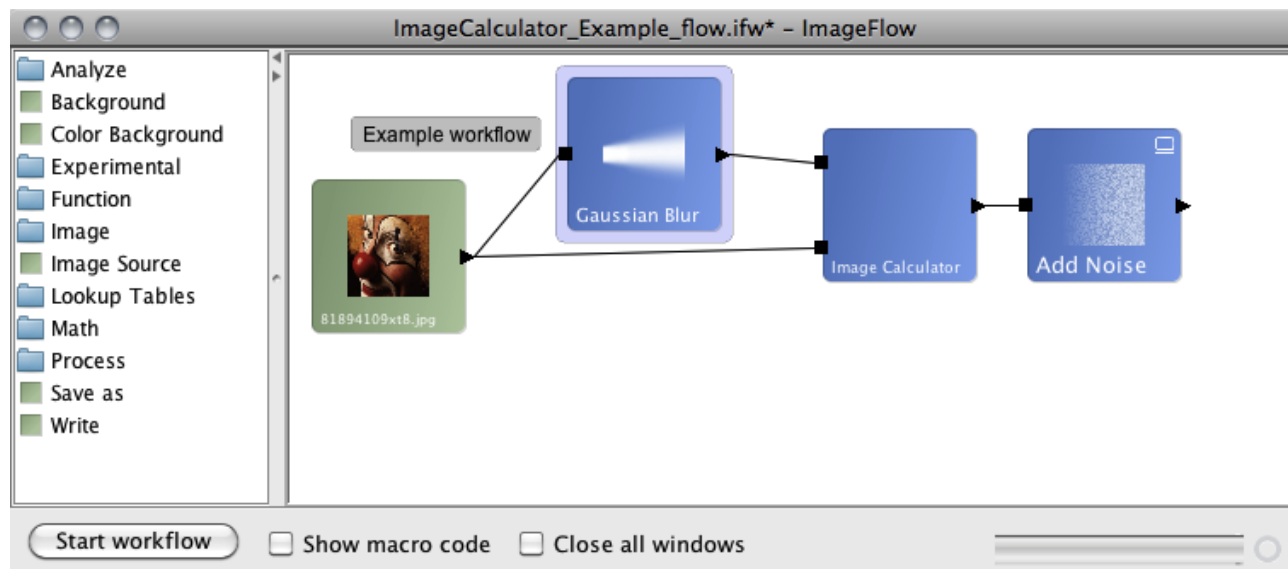

Figure 1. ImageFlow workspace

Further author information: (Send correspondence to Daniel Senff) E-mail: kontakt@danielsenff.de
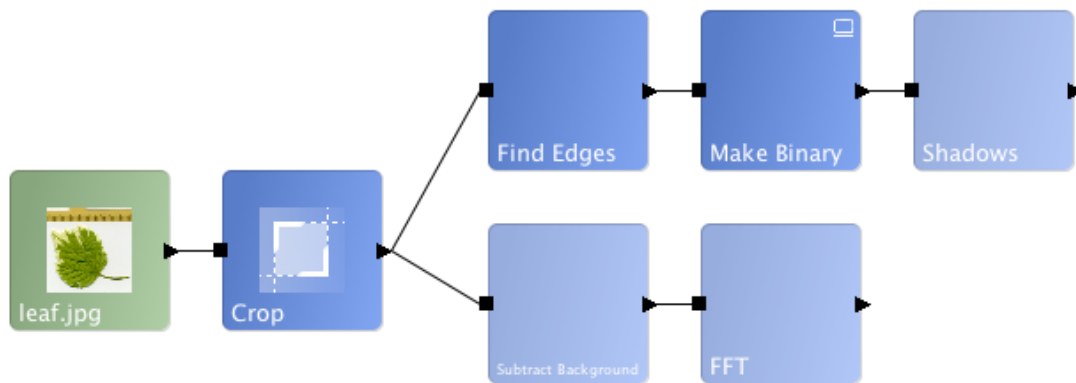
Figure 2. Partial graphs (shown in lighter color) that do not need to be executed

number of *Inputs*, *Outputs* and *Parameters*. Inputs describe the data that is passed into the processing node, while outputs describes the result of the process. Parameters are settings and variables that can influence the performed action. Inputs can be mandatory or optional. Optional Inputs can be coupled with parameters.

ImageFlow workflows support a variety of data types that can be passed along the graph, thereby allowing to build complex structures including mathematical operations, stack processing, data conversion and regular image processing tasks.

Adding nodes to the workspace and connecting them constructs a workflow. There are four kinds of nodes: sources, processing-nodes, sinks and comments. Sources create data objects such as images or integers. Sources do not need any input data. Processing-nodes take various input data and result in new output data. Sinks do not have any output; an example is the "Save-Unit" that saves your image to hard disc. Comments are simply used for making notes on the workspace, they are not written to the macro.

2. IMAGEFLOW PRINCIPLE

2.1 Creating workflow

ImageFlow starts with an empty workspace. Nodes can be added from the list of available processing *Units*. ImageFlow comes with a large variety of units, which are based on ImageJ's core functions. Their naming and categorization is analog to their names within ImageJ. All units related to image manipulation are sorted under *"Process"*, while measurements and non-evasive actions are contained within *"Analysis"*. The functionality represented by the units is kept as atomic as possible, however it is also possible to have more complex macro-code embedded in a processing-node.

Double-clicking will add the unit to the workspace. The squares on the left of the unit represent the *Inputs*, while the triangles on the right represent the *Outputs*. Double-clicking the unit-element will open a settings-window, which displays the possible *Parameters*, as well as a more detailed albeit technical view of the inputs and outputs and their *Connections*. Unit-elements can be renamed to make their purpose more clear, especially once several units of the same type are within a workflow. When several units have been added to the workspace, they can be connected to each other. This is established by dragging a connection from one output to one or more inputs. ImageFlow will try to help you avoid mistakes that can happen by connecting non-supported units. Every input and every output has a specified data type. This data type describes what data it can handle and to which other data types it is compatible. Possible data types are *Strings*, *Integer* or *Double* numbers and images. *Images* also have the distinction between different image types regarding their bit depth. When two units are connected, their corresponding data types are checked and marked accordingly, if an incompatibility is discovered. Another problem that can happen by accident is the creation of loop connections. ImageFlow will also catch this.

A valid workflow with connections between units can now be executed. Every processing node has a visibility switch, that defines, whether the results from this node shall be displayed after execution or not. In general a unit only produces temporary results, and only the result of the final unit is displayed.

This visibility allows executing partial workflows by having branches of the workflow-tree without displaying results. These unused branches are recognized and discarded from execution (see figure 2). On execution the workflow tree is converted into a linear macro-script by determining the dependencies of every unit element. The generated macro code is passed to an instance of ImageJ, which performs the actual image processing. [1]

One general problem of node-based user interfaces is, that the workspace can easily get crowded and a clear display of the modules becomes an issue. ImageFlow has two measures to help this issue. Every node has two possible icon sizes. Instead of the regular square size, a unit can be collapsed to a smaller element that only displays the name and the visibility. The second means to reduce the complexity of a workflow is by grouping related units. Having unit-groups the workflow is still executed as expected, however the units are included in one node. This contains their own miniature workflow with connection between the units within the group and external connections connecting to the units outside. Groups can be ungrouped to restore the original workflow.

Of course ImageFlow allows saving and opening workflows for later use.

2.2 Creating a node-element

Every processing node is described in a special XML-specification. This XML specifies how a node-element will look like on the workspace, what abilities it has and which data types it supports. It also contains the macro commands that are used in the actual macro execution.

The specification is divided into four sections. First is general information about the unit. Among them are the name, description, color, icon and of course the macro code template, that is later used in the macro generation.

The next section is a list of all parameters that can be set for this unit-element. Each Parameter has a defined name, description and data type. At the moment only data types that are available as form-templates in ImageJ's GenericDialog are supported. This constitutes for all primitive types: *Integer*, *Double*, *Boolean*, *String* and *StringArray* (Comma-separated list). Parameters have a defined default value.

The last two sections of the XML describe the available inputs and outputs. Both definitions are very similar and contain - analog to the above definition – the name, the description and the data type. Additional to the primitive data types above is the *"Image"-Object* that can be passed from output to input. While primitive data types are compatible to each other, to determine the compatibility between images another information is required regarding the image type. The image type is analog to the compatibility constants defined in ImageJ's PlugInFilter-class.

There might be situations where the value of a parameter should not be predefined, but rather be calculated during the execution of the workflow. To accommodate this, it is possible to link a parameter with an optional input of the same data type.

The parameter works as a fallback, in case nothing is connected to this input. In case it is connected, the value of the parameter is overridden with the value that is passed through the input from the workflow during the execution. More detailed information about the *XML-Specification* is available in the documentation. [2]

The unit-XML-definitions are loaded on startup of ImageFlow and generate a tree of delegate objects that serve as a factory for building unit-elements with the properties described in the XML definition. These XML are located in 2 predefined locations. The default set of unit-XML definitions is located within the jar file. Additional units can be read from the file system by creating an "xml_units" folder within the folder where the "Imageflow.jar" is located.

The next section shows the XML-description of the gaussian blur unit.

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Gaussian Blur XML-Unit-Specification -->

<UnitDescription>
      <General>
             <UnitName>Gaussian Blur</UnitName>
             <HelpString>Performs a gaussian blur</HelpString>
             <PathToIcon>GaussianBlur_Unit.png</PathToIcon>
             <Color>0x6e91de</Color>
             <ImageJSyntax>run("Gaussian Blur...", "sigma=PARA_DOUBLE_1 STACK");</ImageJSyntax>
      </General>
      <Parameters>
             <Parameter>
                    <Name>Radius</Name>
                    <HelpString>Radius of the gaussian kernel</HelpString>
                    <DataType>double</DataType>
                    <Value>4</Value>
             </Parameter>
      </Parameters>
      <Inputs>
             <Input>
                    <Name>Input</Name>
                    <ShortName>I</ShortName>
                    <DataType>Image</DataType>
                    <ImageType>63</ImageType>
                    <NeedToCopyInput>true</NeedToCopyInput>
             </Input>
      </Inputs>
      <Outputs>
             <Output>
                    <Name>Output</Name>
                    <ShortName>O</ShortName>
                    <DataType>Image</DataType>
                    <ImageType>-1</ImageType>
                    <DoDisplay>false</DoDisplay>
             </Output>
      </Outputs>
</UnitDescription>
```

2.3 Making a node for your plugin

As every plugin for ImageJ can be executed from macro code, in theory every plugin could be turned into a processing-node in ImageFlow. However there are some obstacles that may make these translations hard and may neglect the way ImageFlow expects to handle Macros.

An important factor is how well a plugin was conceived for automatic execution in a macro. For example, no user interaction is expected during the execution of the macro. This means, the user should not be asked for parameters when the plugin is called. The plugin should make use of arguments given via the setup()-Method.
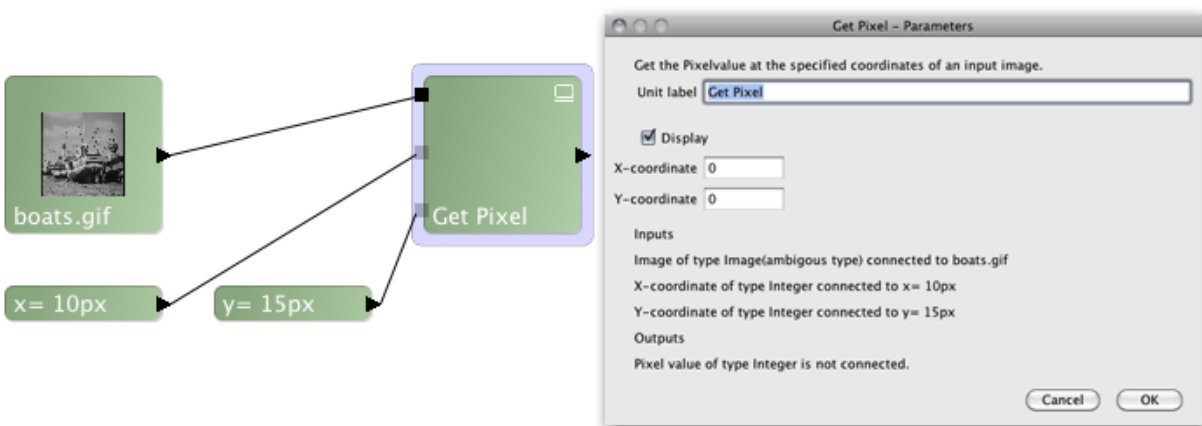

Figure 3. Parameters may be set from other units

Popup dialogs will be annoying, as they popup every time the workflow is executed and defy the purpose of predefining the executed process.[5]

Right now another obstacle is the number of possible outputs. Having a plugin that creates multiple output images is problematic, as every Image has to be identified to the corresponding output. As plugins have different naming schemes to identify generated images (for example "Color Histogram" appends " (red)", " (green)", " (blue)" after the original image's title), reconnecting the resulting images with the expected outputs needs to be done manually in the macro code syntax template of the XML-definition.

3. CONCLUSIONS

ImageFlow started out as the idea to implement a new user interface for the ImageJ macro-interface [6]. Following the visual programming paradigm it has become a toolkit for visually constructing image processing and analysis workflows. It is an alternative interface for newcomers to ImageJ, who have no experiences with the macro interface and serves as an alternative approach towards the regular document/image-focused user interface applied in most applications today. ImageFlow is not without limits and certainly has not yet exhausted it's potential in many regards.

ImageFlow is Open Source and licensed under the GNU General Public License V2. Its source is available at github.[4]

4. REFERENCES

[1] http://wiki.github.com/Dahie/imageflow/macro-generation

[2] http://wiki.github.com/Dahie/imageflow/unit-xml-specification

[3] http://imageflow.danielsenff.de

[4] http://www.github.com/Dahie/imageflow/

[5] http://albert.rierol.net/imagej_programming_tutorials.html#How to automate an ImageJ dialog

[6] Kai Uwe Barthel and Daniel Senff: "Visual programming of image processing algorithms using ImageJ", ImageJ Conference, November 2008