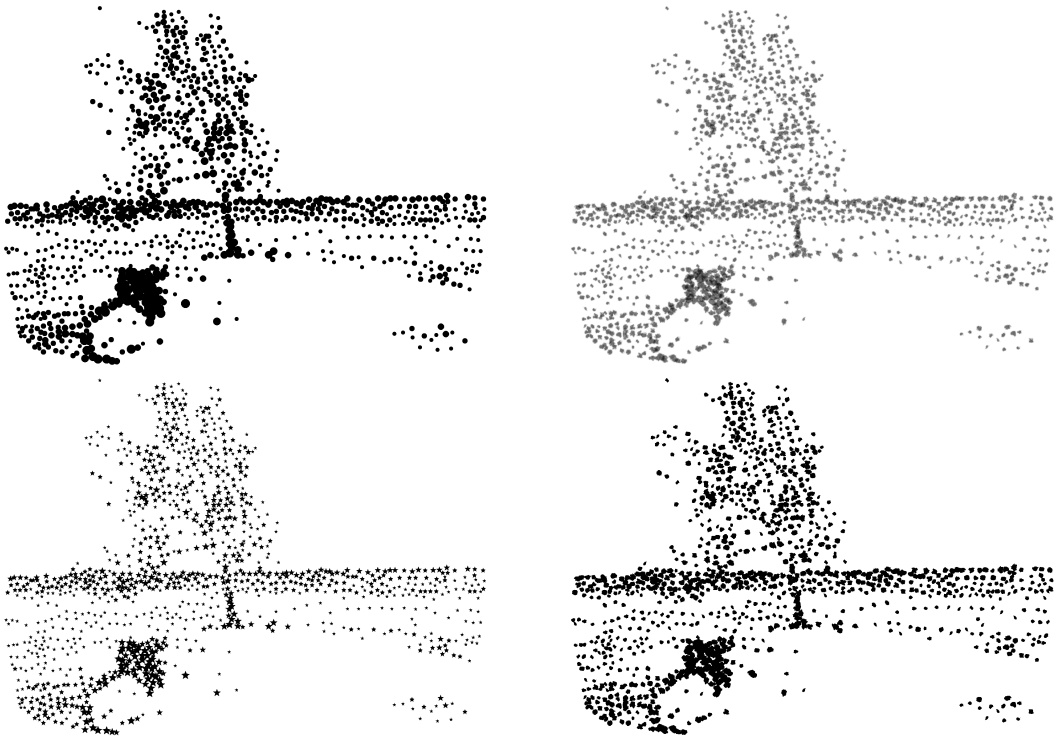


StippleShop

Domingo Martín

version: 0.2
1/2019

Contents

General information

1.1 Copyright

This software has been programmed by Domingo Martín (dmartin@ugr.es). Most parts are original and others are based on original or modified versions of public software. In some cases, the used software is GPL and has been included directly. In other cases, for security, due to the unknown license type, we have used the public software to implement the filters but they have been removed from the source that is published in Github with a GPL license, but it can be downloaded from my web page (`http://calipso.ugr.es/dmartin/Research/Software`). The binary versions of the full programs for Linux and Windows are also in my web page.

StippleShop is a demo program for research purpose only. It contains errors, it may be difficult to use, and there are a many things to improve and/or to include. So, please be patient. Anyway don't hesitate to send us your questions, doubts, improvements or comments.

We want to remark that some of the algorithms that we have implemented are based on the information obtained directly from the articles not from any other software. For this reason it is possible that the results of the new code do not match exactly the results of the original articles. Even for some cases, the algorithms we have developed may differ due to some code optimization made to improve the performance of the underlying algorithms. This can produce slightly different results that those shown in the original articles.

If you want to correct, change or add something, please don't hesitate to contact me at dmartin@ugr.es.

1.2 Compiling

First of all, download the source code from:

`https://github.com/dmperandres/StippleShop`.

Once the source code is downloaded, the first step is to edit the project file (.pro). This project is to be used with the Qt IDE, QtCreator. The important changes are to select the O.S., Linux (default) or Windows, and then to modify the paths (they are marked with XXXXX) to the include and libs folders of the used libraries, GLEW, OpenCV, and of course, OpenGL. After that changes, it will be possible to compile the program.

It is important to note that the code in Github is not complete: some of the filter are removed

due to license issues. In case that you want to include those filters, please, go to `http://calipso.ugr.es/dmartin/Research/Software` and download the file with the code. Unzip it in the folder with the rest of code and uncomment the corresponding lines in the project file (those that begin with DEFINES+ and CONFIG+). Recompile and you will get the full program.

1.3 Introduction

StippleShop is a free software created for designing and rendering stippling images. We created a program with a fixed functionality that allows to test different algorithms, so the main goal of this software is to perform comparisons between different methods. Secondly, this software allows to test new approaches by connecting different techniques. So, we started to develop a program that could dynamically change the results interactively. In order to implement this idea, we considered each method as a black box taking one or more inputs and resulting in only one output. Finally, this boxes can be chained to get a final result. As you may have been realized, we have maintained the Unix philosophy in mind: minimalist, modular software development. For this reason, our software emphasizes building simple, short, clear, modular, and extensible algorithms that can be easily maintained.

The first step in this software development was to define its functionality. We wanted to produce stippling with some of the methods that were published, creating the programs by ourselves in some of the cases, and using the sources lent by authors in other cases. By that moment, we started to realize the need of implementing some auxiliary methods that could help us to improve the input images as well, as for example, to increase the contrast before applying some stippling method. This idea implied that the output of one method should be connected to the input of other method. Applying some functionality to that result in order to produce another output image, then we started to call them **filters**.

It is easy to see that in some cases, the output of one filter is the input of another filter. So that led us to the idea of having a set of images and a set of filters. The combination of several filters was called an **effect**. The order of execution is done recursively, so the software is kept simple.

Given the need of applying several methods for image processing, we chose OpenCV to commit that work. And in order to create a multi-platform user interface we decided to use Qt.

This software is entirely programmed in C++.

1.4 Functionality

StippleShop has a simple architecture underlying that allows us to extend its functionality in a easy way as filters: every filter is derived from a basic class named `_filter`, being necessary to overload its `update()` method. We use a two level approach for user interaction: a middle class that connects the class containing the basic functionality with an especial class that manages the widgets and their callbacks. This class has been adapted to Qt, so we can configure the UI library in an easy way.

There are two types of filters: filters with one input and one output, and filters with two inputs and one output.

Generally, all filters support color and grayscale images indistinctly, but grayscale images seems to be faster. Only in a few cases, the input must be a color image to produce the correct output[1].

So far we have implemented the following filters:

- Placement and Stippling
  - Capacity-Constrained Voronoi Tessellation by Balzer et al. [3]
  - Example-Based Grayscale stippling by Martín et al. [6, 7]
  - Recursive Wang Tiling by Kopf et al. [2] (Not included)
  - Stippling by Example (based on the original source code) by Kim et al. [5] (Not included)
  - Structure-Aware Stippling by Li and Mould [4]
  - Weigthed Centroidal Voronoi Diagrams by Secord [1]
- Halftoning
  - Adaptive Clustering and Selective Precipitation by Wong and chi Hsu [11] (Not included)
  - Contrast-aware halftoning (without priority list) by Li and Mould [9]
  - Error diffusion by Ostromoukhov [8] (Not included)
  - Space Filling Curve based on Velho and Gomes [10] and Wong and chi Hsu [11] (Not included)
- Dot output control
  - Example-Base Grayscale by Martín et al. [6]
  - SVG

[1] It is more precise to talk about 3 channels (color), and 1 channel (gray-scale) images. It is easy to see that 1 channel filters are faster than 3 channels ones.

- Contrast
  - Contrast and brightness
  - Retinex filter by Land [12]
- Edge detectors
  - Canny's edge detection [13]
  - DoG: difference of Gaussians [14]
  - Kang et al.'s lines [15]
- Blur
  - Gaussian blur
- Other effects
  - Bilateral
  - Dilation
  - Distance field
  - Erotion
  - Inversion
- Combination
  - Combination of two inputs, pixel to pixel, with an associated operation (AND, OR, SUM, SUBSTRACTION, MULTIPLICATION, DIFFERENCE)
- Measuring
  - SSIM [16] and PSNR

1.5 Software included

These software have been used (with or without modification) for the development of *StippleShop* :

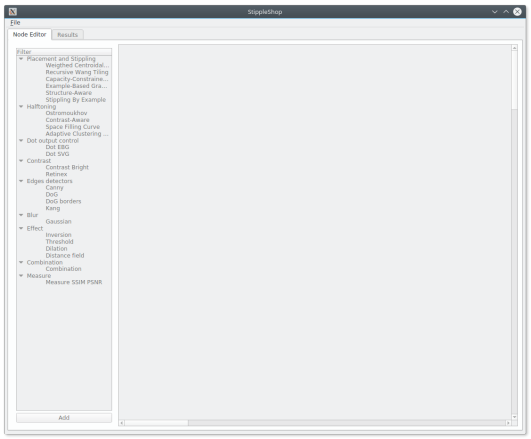
- Error diffusion halftoning by Ostromoukhov [8]
  https://liris.cnrs.fr/victor.ostromoukhov/publications/publications_abstracts.html#SIGGRAPH01_VarcoeffED
- Stippling by Example by Kim et al. [5]
- Capacity-constrained Voronoi tessellation by Balzer et al. [3]
  https://code.google.com/archive/p/ccvt/

- Recursive Wang tiling by Kopf et al. [2]
  http://johanneskopf.de/publications/blue_noise/source_code/index.html
- Halftoning with selective precipitation and adaptive clustering by Wong and chi Hsu [11]
  http://www.cse.cuhk.edu.hk/ ttwong/papers/halftone/sfc.html
- Retinex filter (Land [12]) based on GIMP code.
- Node editor by Stanislaw Adaszewski
  http://algoholic.eu/qnodeseditor-qt-nodesports-based-data-processing-flow-editor/

Using Stippleshop

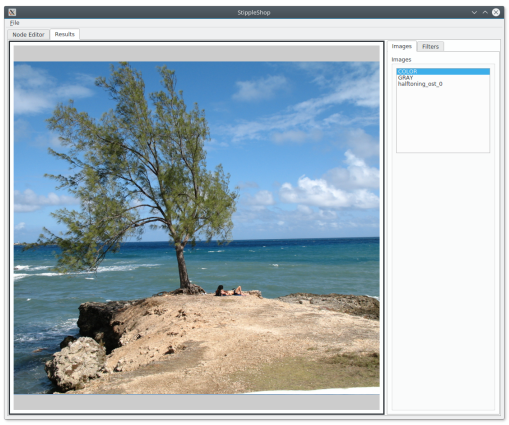
2.1 Working pipeline

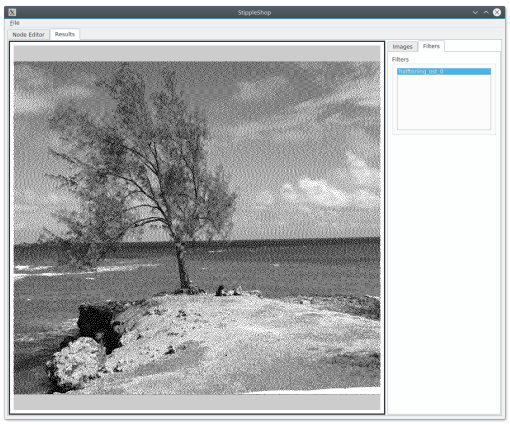
The program is divided in two main processes: edition and visualization. These processes are related with the two main tabs: the *Nodes editor* tab and the *Results* tab.



The edition process is simple: the user creates a new effect or loads a previously defined effect that can be edited until the desired result is obtained. The *Nodes editor* shows the filters and their connections. It allows us to add new filters to design some effect or to delete one or more filters from some existing effect. Filters can be connected by dragging the output port and dropping the link at the input port of the target box.

The visualization process applies the effect to an input image and produces the intermediate and the final results. To produce these images the user must click at the *Results* tab. Before obtaining any result it is oblied to load an image, otherwise the final result will not be computed. The loaded image becomes two boxes, representing two different versions of the same image: one in color and one in grayscale. These boxes should be the input of any of the next filters in the effect. If the loaded image is grayscale, both color and grayscale version will be the same.





It is important to take in mind that every filter produces an output result named like the filter, which may be the input of another filter in turn. Each output image can be selected by the user to show some intermediate result. This is done by selecting the output at the list provided in the Images tab. Each image is correlated to the output of a filter, so the name will be the same as the filter.

For each filter, the user can modify its parameters to adjust the result. In order to set those parameters, the *Filters tab* must be clicked, and then a filter must be selected. Changes are propagated in the chain, so some change in one input filter affects to all the connected boxes until the final result.

This uncoupling between images and filters provides an extraordinary flexibility, allowing to explore different values for a filter and to show the effects in other filters as well.

Once the final result of an effect is obtained, it is possible to click again at the *Nodes editor* tab. To edit the result it is possible to go back and to add or remove filters, or even to modify the connections between filters by deleting or making new connections. Again it is possible to show the result by clicking at the *Results* tab. This process can be repeated until the desired result is obtained.

For saving an effect, it is necessary to create or to load some effect. Take in mind that to load an image is a requirement for the software because it is the way our software has to check that the effect is right. If that is the case, the current parameters will be saved. In the same way, for saving images it is necessary to create an effect and to load some input image too.

2.2 Limitations

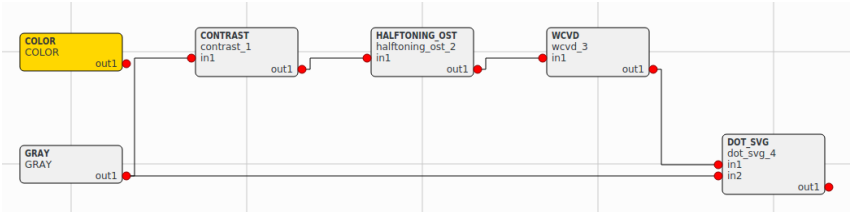
StippleShop has many limitations due to its very nature as demo tool, not only in its user interface but also in its functionality. We will try to fix and improve the program as soon as we can but in the mean time here is some of these limitations:

It is possible to connect one output to several inputs but the user should avoid to create loops in the connections. The current version does not check for loops, so be careful with this condition or just expect the worse.

Though it is possible to change from the edition process to the visualization process and vice versa in any moment, the implementation is not optimal. If there is no edition in the *Nodes editor* tab, switching to the visualization tab implies no computing. Otherwise, if the effect is edited, switching to the *Results* tab implies that the effect is constructed again from the very beginning. This is due to the complexity of checking all the possibilities in the pipeline.

2.3 A step by step example

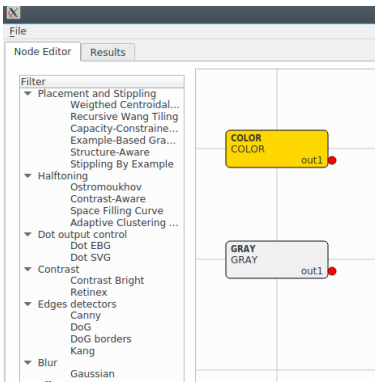
Now this section will show you how to create an effect in a step by step example. Let's start with an easy example that shows all the actions that are necessary to produce a final result. For this example, we want to produce an effect that takes an input image, applies a contrast filter to eliminate information, then applies a halftoning method (Stromoukhov), a WCVD method for dot placement, and finally, it produces a result that uses SVG dots and modulates the size depending on the tone of the original image. The idea is to reproduce Secord's WCVD technique using the possibilities of *StippleShop* . Here is the final structure we want to obtain.



Once the program is running, we must first decide between creating a new effect by selecting *Menu* → *New effect* or by loading an effect in *Menu* → *Open effect file*. For this example, we select the creation of a new effect.

Nodes editor is initialized by *COLOR* and *GRAY* blocks.

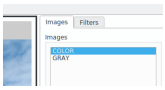
These blocks represent the color and the grayscale versions of the image that must be loaded to apply effects. Also, the list of filters is enabled (it should appear in dark tone). If we load an effect from a file, a diagram with all the filters and connections will be displayed.



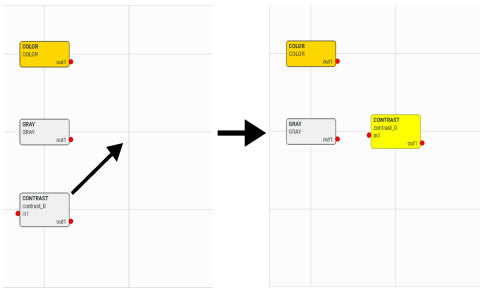
Now it is possible to see the result of applying the effect (though in this case we can only see the loaded image). So, in order to display the effect, we click on the *Results* tab. The initial result is a gray image because we have not loaded any image yet.

The next step is to load the image by means of the menu *File* and *Open input image*, and then selecting our image. The program loads the image and produces a complementary version: if the loaded image has 3 channels of color (RGB) the program will create a grayscale version too; if the original image is grayscaled, an RGB image will be created. The RGB image is named as *COLOR* and the grayscale image is named as *GRAY*. It is possible that the program asks about adjusting the size of the input image. This may happen because our software needs an image to be multiple of 4. Notice that filters that need and/or produce an RGB image are

drawn in orange, whereas filters that need and/or produce a grayscale image are drawn in gray.

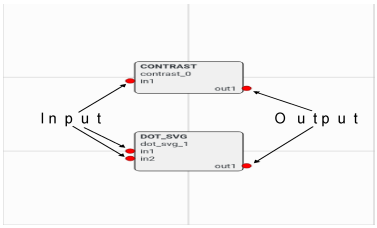


Now it is possible to see the effect of each filter by selecting *Images* tab. The box named *COLOR* - the RGB version of the input image - is selected by default.



We can now create the new effect returning to the *Node Editor* tab. In order to add our filter named *Contrast-Bright*, we search for it at the list, then we select the filter by clicking on its name, turning the selected filter into a bright blue color. The filter is added by pressing the *Add* button at the bottom of the list or by right-clicking and selecting the *Add filter* option. In any case, the new filter will appear at the same position under the *GRAY* filter. This new filter contains the uppercased name of the type of the filter and the lowercased name of the filter joined to a number. It is possible to change its name by double clicking on the block. A limitation of our software is that the names of blocks must be unique (*Warning: the current version does not check there is no names repetition*).

It is necessary to move the filter from this initial position in order to use it and to allow that other filters can be added: if the initial position is not empty, the program will warn us about it.



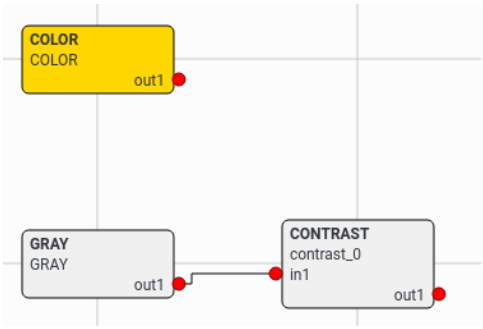
The filter is moved by a drag-and-drop operation: When we click at the filter it become selected, changing its color to yellow (it can be unselected by clicking anywhere at the background of the application). We can move the filter by dragging and dropping it in any desired position.

The next operation consists in connecting the output of one filter with the input of other filter. In this case, we want to connect the output of the GRAY filter to the input of the *Contrast-Bright* filter (its gray color implies that this filter accepts a grayscale image as input; it is also possible to connect an RGB output to a grayscale input and vice versa but this implies a conversion that requires some extra computation time). The input and output ports of a filter are drawn as red circles. Given that the process runs from left to right, the input ports are on the left of the filter and the output port is always on the right of the filter. They are also marked with the text “in” and “out” and a number.

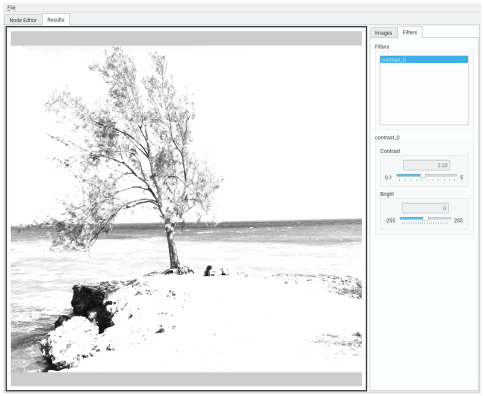
The process for connecting two boxes is easy: click on the red circle of the output port for *GRAY* filter, drag the arrow and drop it at the red circle that represents the input port of *Contrast-Bright* filter.

When two boxes are linked, a path is drawn connecting theirs ports. This implies that the

output of the *GRAY* filter is the input of the *Contrast-Bright* filter. Also, this implies that a new image with the same name as the *Contrast-Bright* filter will be created.



To check this effect we must switch to the *Results* tab, as we previously did. It is possible to see the effect of *Contrast-Bright* filter applied to the grayscale input image by clicking on the name of the new filter that has been included in the *Images* tab. Due we have not set the parameters of our filter, we get as result of applying the *Contrast-Bright* filter, the same images as the original grayscale input. We can adjust the parameters of the filter to see some change, so we switch to *Filters* tab. This displays the list of filters that have been defined in the effect. There should only be one filter: *Contrast-Bright*. We select the filter from the list by clicking on it (in this case, as there is only one, it is selected by default). Now all the parameters of the filter should appear below the list. The parameters of *Contrast-Bright* filter are a multiplication factor for the contrast (which values are in between 0.1 and 5) and an addition factor for the bright (which values are in between -255 and 255).



Our intention is to reduce the amount of information, so we increase the contrast up to 2 (the user can change it dynamically until the desired result is obtained).

It is possible that you do not see any change but the color image. This is due to that the image that is shown in *Images* tab is unlocked from the image that is selected in *Filters* tab. This is done intentionally to improve the flexibility: it allows us to modify the parameters of one filter and to see the effect of another filter.

In order to see the resulting effect of the adjusted parameters of the *Contrast-Bright* filter, this must be selected in the *Images* tab.

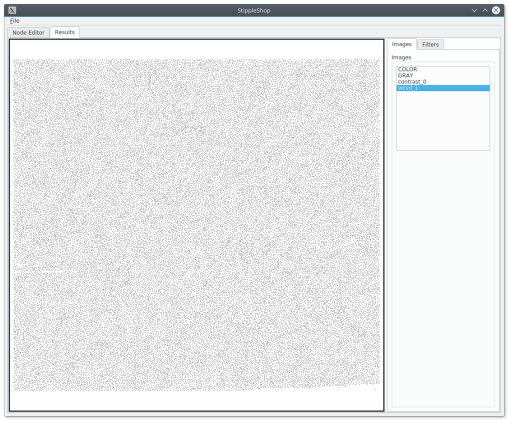
The next filter we want to add is the *WCVD* filter. So, we must return to *Node Editor* tab. The process for including the *WCVD* is basically the same. Once the filter is added and moved, we place it near the *Contrast-Bright* filter. We proceed to connect the output of the *Contrast-Bright* filter with the input of the *WCVD* filter. The result is shown by clicking on the *Results* tab and then selecting the *WCVD* filter in the *Images* tab. The result is a set of pixels that approximates the input image.

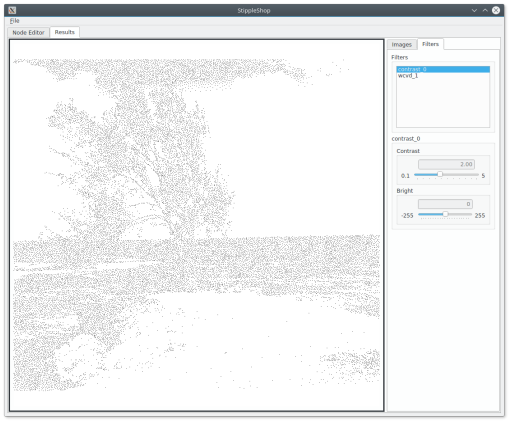
At this moment the result may seem confusing for the user but take into consideration that each pixel represents the position of each dot without any aesthetic characteristic. Another important thing to take into account is that the WCVD implementation randomly creates a

set of seeds for the CVD using the darker pixels (values lesser than 255) of the input image.

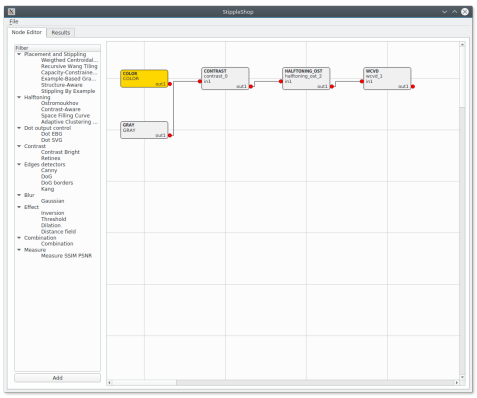
We can play with contrast to see how much information is removed. For example, when we set the contrast to 1 (the default value), there are many dots and it is very difficult to distinguish any shape. Although when we set the contrast to 2, the most important zones are visible and the the less important areas have been removed.

But if we want to follow Secord's approach we are not done yet. We must also use a halftoning input image, in our case we are going to use a Ostromoukhov's halftoning.





We need to switch again to *Node Editor* tab. As we need the *Ostromoukhov* filter between the *Contrast-Bright* filter and the *WCVD* filter, it is necessary to disconnect the *Contrast-Bright* filter from the *WCVD* filter. This done by right clicking on the path that connects both filters and selecting the option *remove*. Now we can include the *Ostromoukhov* filter following the same steps as before. Though the filters can be placed anywhere, it is convenient to organize them in a graphically coherent way for the sake of clarity.



The last step is to change the appearance of each dot from a pixel to a more complex and aesthetic representation. In this example we want to produce a vectorial result. So, we select the *Dot SVG* filter. There is no enough space to include the new filter so we can turn the mouse wheel until we have adjusted the image to our needs. Once this is done, the new filter can be added. We repeat the same steps: first by adding a new *Dot SVG* filter, later moving it to another position and linking it. This filter has two inputs: input 1 represents the position of the dots, input 2 is used to modify the size of the dots depending on the tone.

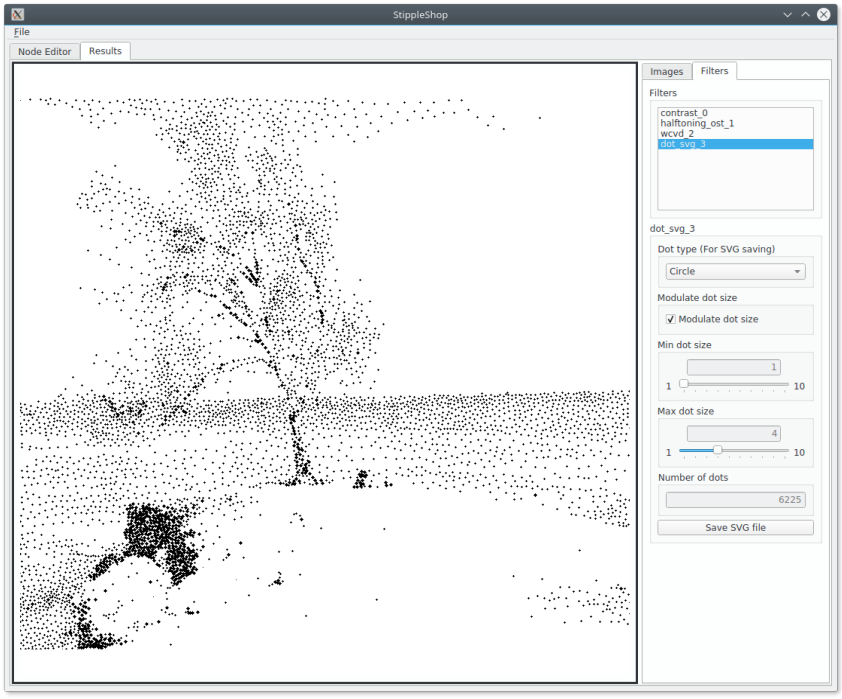
Finally, to get the final result, we must return to the *Results* tab and select the *Dot SVG*

filter in the *Images* tab. In order to set the parameters, we select the *Filters* tab and select the *Dot SVG* filter. It is important to note that the displayed image is not vectorial but an approximation. The vectorial result is obtained by pressing the *Save SVG file* button at the bottom of the widget. It is possible to select between different shapes. The default option replaces each dot by a black circle.

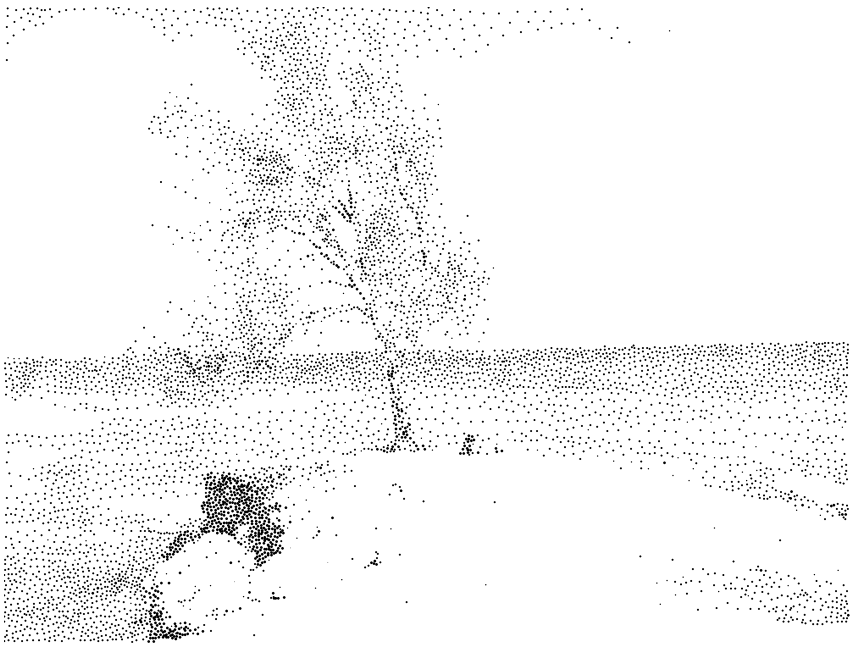
The filter computes the size of each dot by means of a random value in between two defined numbers (a minimum and a maximum). These values can be manually set by the user. Other possibility is to define the size of each dot depending on the tone of the corresponding pixels of the second input image, and using the same minimum and maximum values as boundaries.

Now the result is displayed as the final image and its corresponding vectorial result.

Now, if we are happy with the result we can save the effect. We switch to the *Node Editor* tab, and then we select the *Save effect file* option in the menu.



Domingo Martín©



2.4 Results

Now we show some results that can be obtained by *StippleShop* .

2.4.1 Example 1

This effect improves the contrast of the input image, then a halftoning based on Otromoukhov's method is obtained, and finally, a final version is produced by means of scanned dots.

```
{
  "effect":
  [
  {
    "type_filter": "CONTRAST"
    "input_image_0": "GRAY",
    "output_image_0": "contrast_1",
    "input_image_1": "NULL",
    "contrast": " 2,20",
    "bright": "0"
  },
  {
    "type_filter": "HALFTONING_OST"
    "input_image_0": "contrast_1",
    "output_image_0": "halftoning_ost_1",
    "input_image_1": "NULL"
  },
  {
    "type_filter": "DOT_EBG"
    "input_image_0": "halftoning_ost_1",
    "output_image_0": "dot_ebg_1",
    "input_image_1": "GRAY",
    "pixel_density": "300PPI",
    "modulate_dot_size": "false",
    "black_dots": "false",
    "black_threshold": "220"
 }
 ]
}
```

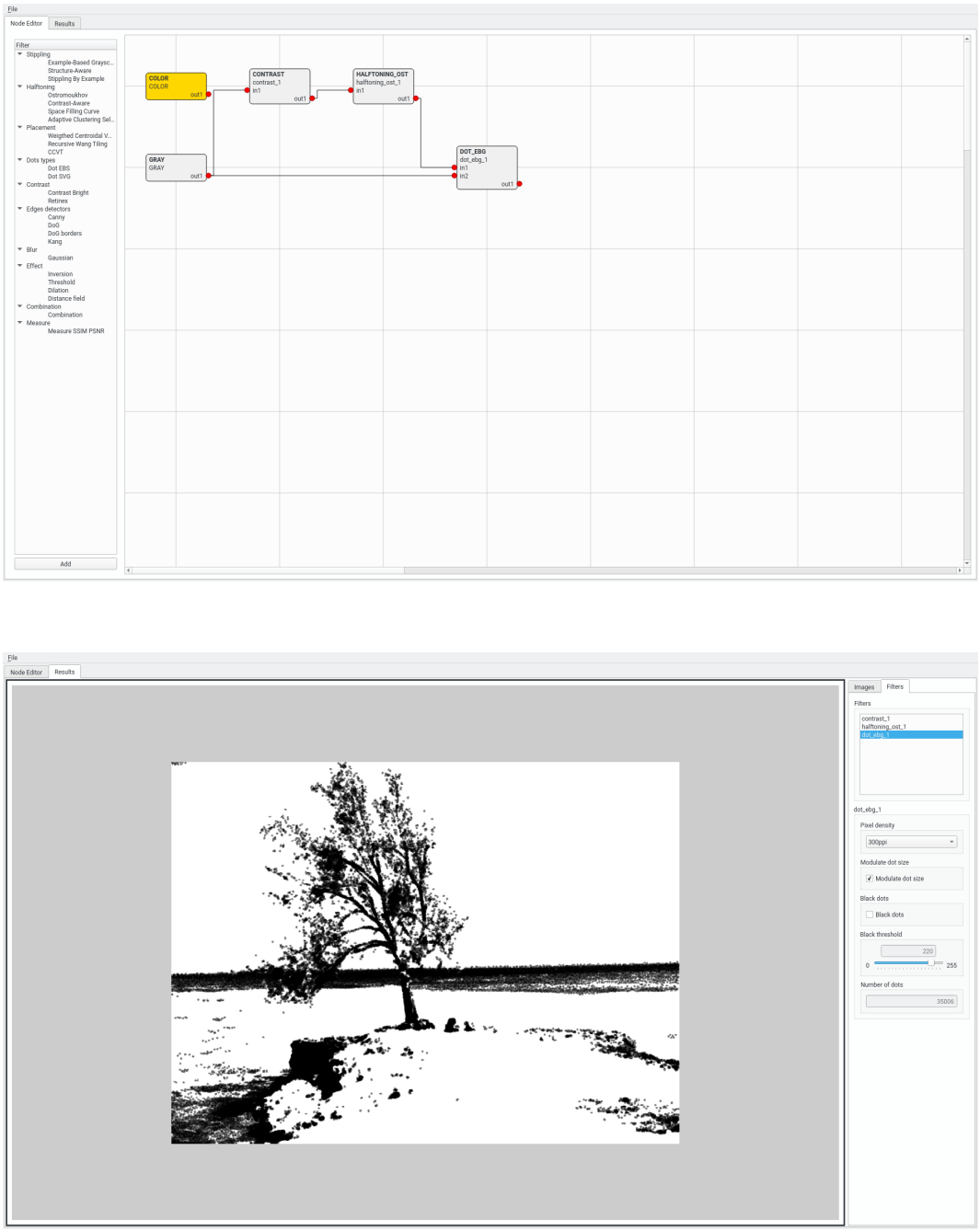
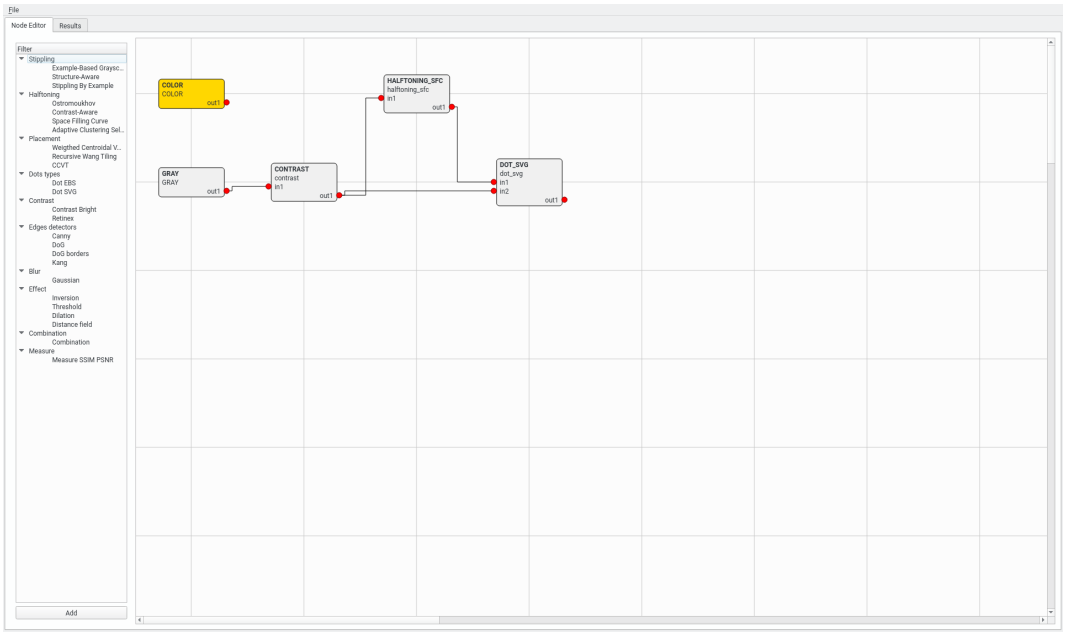


Figure 2.1: Example 1.

2.4.2 Example 2

This effect improves the contrast of the input image, then a halftoning based on Space Filling Curve method is obtained, and finally, a final version is produced by using SVG dots.

```
{
  "effect":
  [
  {
    "type_filter":"CONTRAST",
    "input_image_0": "GRAY",
    "output_image_0": "contrast",
    "input_image_1": "NULL",
    "contrast":"default",
    "bright":"default"
  },
  {
    "type_filter": "HALFTONING_SFC",
    "input_image_0": "contrast",
    "output_image_0": "halftoning_sfc",
    "input_image_1": "NULL",
    "cluster_size": "9"
  },
  {
    "type_filter":"DOT_SVG",
    "input_image_0": "halftoning_sfc",
    "output_image_0": "dot_svg",
    "input_image_1": "contrast",
    "dot_type":"default",
    "modulate_dot_size":"false",
    "min_dot_size":"default",
    "max_dot_size":"default"
  }
  ]
}
```



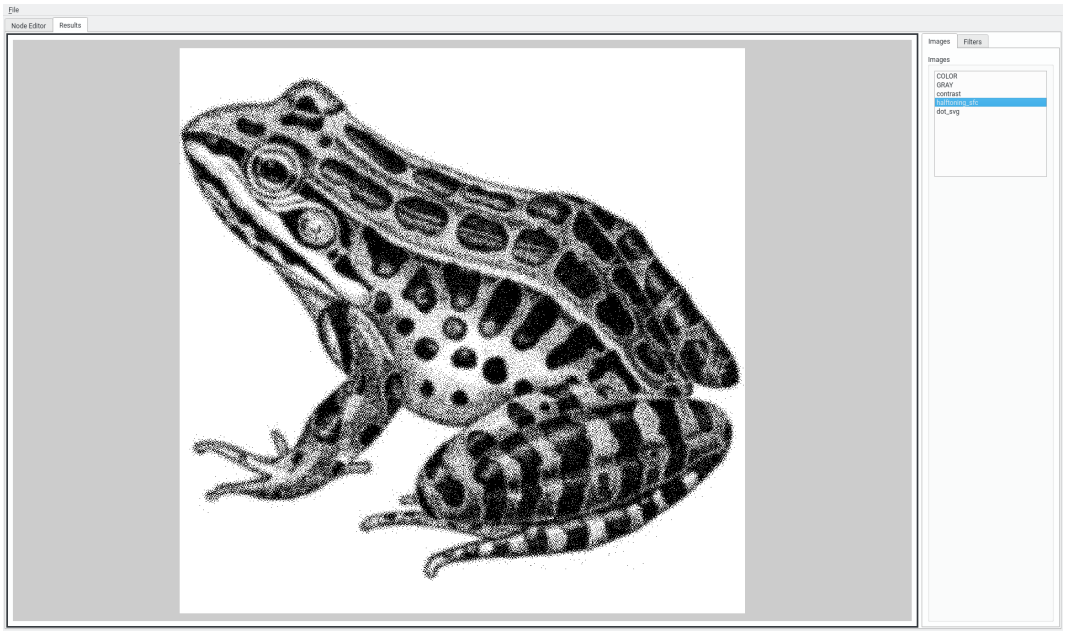


Figure 2.2: Example 2.

2.4.3 Example 3

This effect improves the contrast of the input image, then a halftoning based on Recursive Wang Tiling method is obtained, and finally, a final version is produced by using SVG dots.

```
{
  "effect":
  [
  {
    "type_filter":"CONTRAST",
    "input_image_0": "GRAY",
    "output_image_0": "contrast",
    "input_image_1": "NULL",
    "contrast":"default",
    "bright":"default"
  },
  {
    "type_filter": "RWT",
    "input_image_0": "contrast",
    "output_image_0": "rwt",
    "input_image_1": "NULL"
  },
  {
    "type_filter":"DOT_SVG",
    "input_image_0": "rwt",
    "output_image_0": "dot_svg",
    "input_image_1": "contrast",
    "dot_type":"default",
    "modulate_dot_size":"false",
    "min_dot_size":"default",
    "max_dot_size":"default"
  }
  ]
}
```

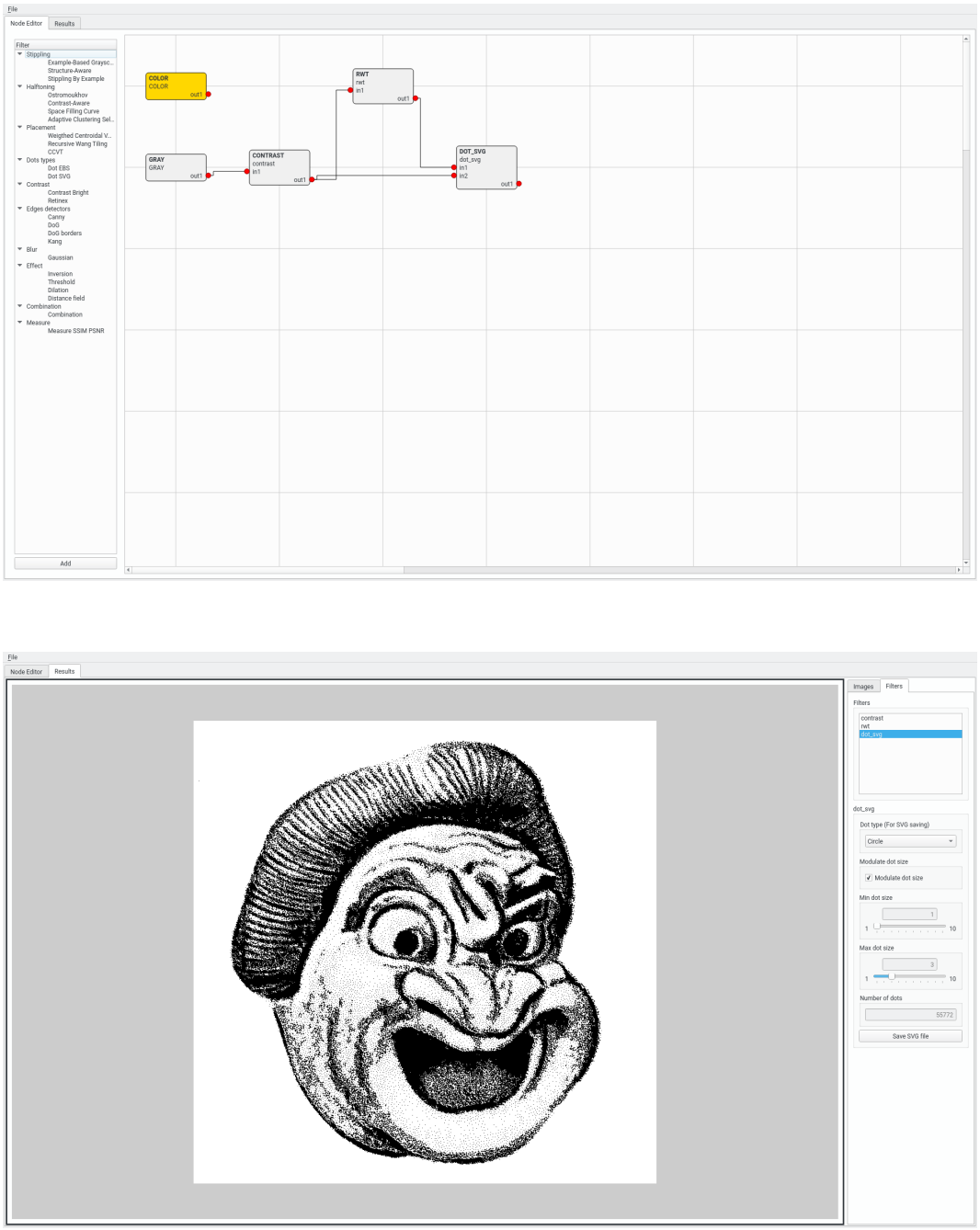


Figure 2.3: Example 3.

2.4.4 Example 4

This effect improves the contrast of the input image, then a halftoning based on Otromoukhov's method is obtained, and finally, a final version is produced by using the Example-Based Grayscale method.

```
{
  "effect":
  [
  {
    "type_filter":"CONTRAST",
    "input_image_0": "GRAY",
    "output_image_0": "contrast",
    "input_image_1": "NULL",
    "contrast":"default",
    "bright":"default"
  },
  {
    "type_filter": "RWT",
    "input_image_0": "contrast",
    "output_image_0": "rwt",
    "input_image_1": "NULL"
  },
  {
    "type_filter":"DOT_SVG",
    "input_image_0": "rwt",
    "output_image_0": "dot_svg",
    "input_image_1": "contrast",
    "dot_type":"default",
    "modulate_dot_size":"false",
    "min_dot_size":"default",
    "max_dot_size":"default"
  }
  ]
}
```

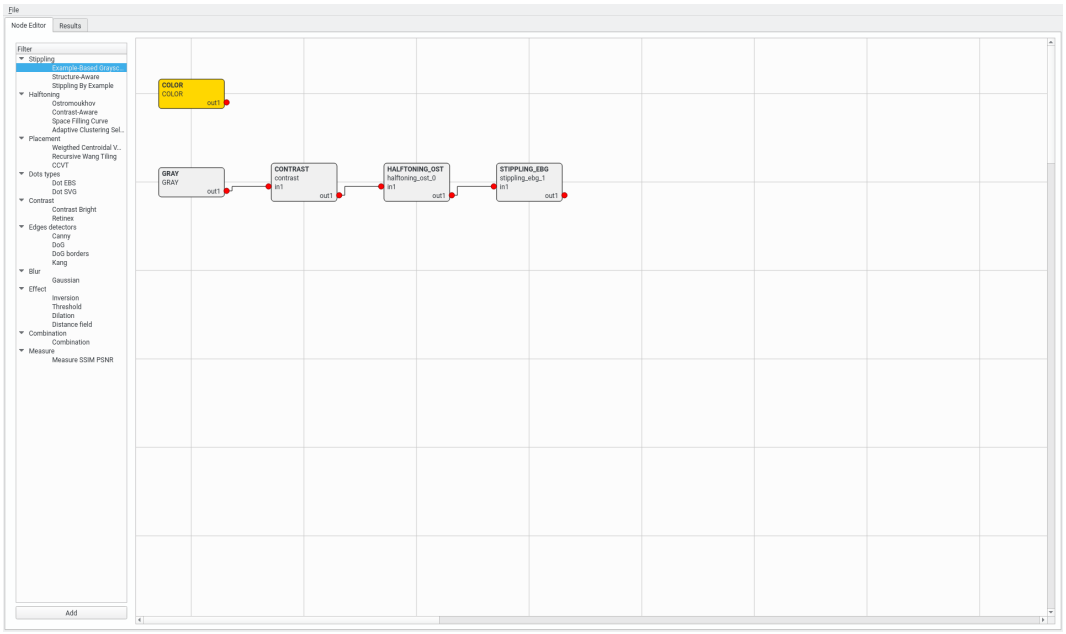


Figure 2.4: Example 4.

Filters

3.1 General definition

Each filter is a blackbox with one or two input images and one output image. Also, it contains a set of parameters. The filter applies an algorithm or an effect to the input images to produce a result, which is placed on the output image.

Notice that both the images and the filters are identified by their names. For the sake of simplicity, the name of the output image is the same name of the filter that generates it.

This way it is easier to find the image in the *Images* tab, or the parameters of the filter in the *Filters* tab.

When an image is loaded, one RGB and one grayscale versions are created. Take into account that if a grayscale image is loaded an RGB version of the same image will be created too (the internal format for the RGB is a 3-channel color image).

The name of the RGB image version is *"COLOR"*. The name of the grayscale version is *"GRAY"*. One of them must be the input of at least one filter.

In order to create an effect it is important to remark that the name of an input image MUST be defined before it is used. This implies that we must follow a temporal order. This order is from left to right.

Taking this idea into account, it is simple to realize that when a filter has changed (this is, one parameter has been modified), not only the result of such filter change but also the following connected filters. In order to improve the performance of the algorithms, the slower filters should be placed at the beginning of the effect if that is possible.

Most of the filters are defined to work with a grayscale image, and produce another grayscale image. In the case that the input is an RGB image but the filter needs a grayscale input, the conversion is done automatically. In the same way, the conversion from a color image to a grayscale image is done automatically by the software.

The second input image of those filters that actually only requires one input image must be declared as *"NULL"*;

3.2 JSON format

Filters and their composition can be defined using a JSON text file.

Every effect file must follow the JSON definition. For this program, the filters are included as an array. The format for one filter is as follows:

```
{
  "effect":
  [
   {
   filter data
   }
  ]
}
```

There are one or more filters between the brackets. Each filter is included between braces. Remember that each filter is separated form the next one with a comma: {...},{...},...

The information of the filter is given with the next fields delimited by braces:

```
"type_filter"
"input_image_0"
"output_image_0"
"input_image_1"
```

These fields are mandatory. A colon is used to separate each field and the value, and notice that values are delimited with double-quotation marks. For example:

```
"input_image_0":"COLOR".
```

Each filter can have from 0 to N parameters. They will be placed at the end of the filter definition. In the case that we want to give the default value, it must be used the *"default"* tag.

Note that we usually write a text file following some order for a better comprehension, but JSON format is order free, so the order may change when our program writes the file. Anyway, it should not affect the result though it probably will result in a harder to read file.

3.3 Categories

Our filters are grouped into nine categories:

- Placement and Stippling
- Halftoning
- Dot output control
- Contrast
- Edge detectors
- Blur
- Effect
- Combination
- Measure

3.4 Placement and Stippling

3.4.1 Capacity-Constrained Voronoi Tessellation

Explanation:
This filter uses the Capacity-Constrained Voronoi Tessellation. Each final dot depends on several points, which are used to compute the movement of the centroidals. Given the number of dark dots of the input image, DD, the image will be represented by M, the number of dots, but using N points per dot to compute the solution. *MxN* must be less or equal to DD.

Input_image_0: grayscale image.

Channels: 1 channel $\rightarrow$ 1 channel

Parameters:

- number_of_dots: M. Number of dots to be drawn. Default=1000.
- number_of_points_per_dot": N. Number of points per dot used in the computation of the algorithm. Default=10.

Reference: Balzer et al. [3].

Example:

```
{
"type_filter": "CCVT",
"input_image_0": "contrast",
"output_image_0": "ccvt",
"input_image_1": "NULL",
```

```
"number_of_dots": "default",
"number_of_points_per_dot": "default"
}
```

3.4.2 Example-Based Grayscale

Explanation:
This filter executes an Example-Based Grayscale.

Input_image_0: grayscale image.

Channels: 1 channel → 1 channel

Parameters:

- displacement_ramdomness: 0-1. Amount of dispersion when displacing the dots randomly (percentage of the medium size). Default=0.25.

Reference: Martín et al. [6, 7].

Example:

```
{
"type_filter": "STIPPLING_EBS",
"input_image_0": "halftoning_ost",
"output_image_0": "stippling_ebs",
"input_image_1": "NULL",
"displacement_ramdomness":"default"
}
```

3.4.3 Recursive Wang Tiling

Explanation:
This filter uses the Recursive Wang Tiles method. We have used the given code but only for the generation given the set of data. The main requirement is that input images must be squared. Note that our custom implementation of the modulation of the dot size, as well as the antialias algorithm, differs from the original version.

Input_image_0: grayscale image.

Channels: 1 channel → 1 channel

Parameters:

Reference: Kopf et al. [2].

Example:

```
{
```

```
"type_filter": "RWT",
"input_image_0": "contrast",
"output_image_0": "rwt",
"input_image_1": "NULL"
}
```

3.4.4 Stippling by Example

Explanation:
This filter implements Stipling by example. It is a pretty complex algortihm with different stages. We have taken the given code and used only the rendering part. We had adjusted it to use cv::Mat from OpenCV.

Input_image_0: grayscale image.

Channels: 1 channel $\rightarrow$ 1 channel

Parameters:

- save_intermediate_results: true or false. It saves the intermediate results (default false)
- combine_edges: true or false. It combines the edges image (default false)
- post_processing: true or false. It postprocesses the image to improve it (default true)

Reference: Kim et al. [5].

Example:

```
{
"type_filter": "STIPPLING_SBE",
"input_image_0": "GRAY",
"output_image_0": "stippling_sbe",
"input_image_1": "NULL",
"save_intermediate_results":"default",
"combine_edges":"default",
"post_processing":"default"
}
```

3.4.5 Structure-Aware Stippling

Explanation:
This filter uses Structure-Aware Stippling. It is an approximation of the original method. The priority list has been replaced by a random selection for the sake of the performance of the algorithm. Nevertheless, the result is very similar to the original.

Input_image_0: grayscale image.

Channels: 1 channel → 1 channel

Parameters:

- kernel_size: 3–15. Default=7.
- exponent: 1–5. Default=2.0.
- g+: 1–10. Default=5.
- g-: 1–10. Default=5.
- k: 0–1. Default=0.

Reference: Li and Mould [4].

Example:

```
{
"type_filter":"STIPPLING_CAS",
"input_image_0": "GRAY",
"output_image_0": "stippling_cas",
"input_image_1": "NULL",
"kernel_size":"default",
"exponent":"default",
"g+":"default",
"g-":"default",
"k":"default"
}
```

3.4.6 Weighted Centroidal Voronoi Diagram

Explanation:
This filter creates a stippling image from another halftoning image by means of a Weighted Centroidal Voronoi Diagram following Hoff's method.

Input_image_0: grayscale image.

Channels: 1 channel → 1 channel

Parameters:

- percent_of_dots: 0–100. Percentage of dots to be used in WCVD. The percentage is applied to a subset (35%) of the darker dots in the input image. This parameter has priority over the number of dots. Default=25.
- number_of_dots: N. Number of dots to be used in WCVD. Default=1.
- save_intermediate_images: true or false. This parameter allows to save - as images in the WCVD folder - the intermediate stages when displacing the centroids. Default=false.

Reference: Secord [1].

Example:

```
{
"type_filter":"WCVD",
"input_image_0": "GRAY",
"output_image_0": "wcvd",
"input_image_1": "NULL",
"percent_of_dots":"10",
"number_of_dots":"default",
"save_intermediate_images":"false"
}
```

3.5 Halftoning

3.5.1 Adaptive Clustering and Selective Precipitation Halftoning

Explanation:
This filter produces a halftone from a gray input image using a space filling curve with adaptive clustering and selective precipitation (see Wong and chi Hsu [11]).

Input_image_0: grayscale image.

Channels: 1 channel $\rightarrow$ 1 channel

Parameters:

- cluster_size: 1–256. Default=9.
- threshold: 0–600. Default=100.
- adaptive_clustring: true or false. Default=true.
- selective_precipitation: true or false. Default=true.

Reference: Wong and chi Hsu [11].

Example:

```
{
"type_filter": "HALFTONING_ACSP",
"input_image_0": "GRAY",
"output_image_0": "halftoning_ascp",
"input_image_1": "NULL",
"cluster_size":"default"
"threshold": "100",
```

```
"adaptive_clustering": "true",
"selective_precipitation": "true
}
```

3.5.2 Contrast-Aware Halftoning

Explanation:
This filter produces the halftoning from a gray input image using Mould (Contrast-Aware). It is an approximation to the original method. The priority list has been replaced by a random selection for the sake of efficiency of the algorithm. Nevertheless, the result is very similar to the original.

Input_image_0: grayscale image.

Channels: 1 channel → 1 channel

Parameters:

- kernel_size: 3–15. Default=7.
- exponent: 1–5. Default=2.6.

Reference: Li and Mould [9].

Example:

```
{
"type_filter": "HALFTONING_CAH",
"input_image_0": "combination4",
"output_image_0": "halftoning_cah",
"input_image_1": "NULL",
"kernel_size":"default",
"exponent":"default"
}
```

3.5.3 Ostromoukhov's Error Difussion Halftoning

Explanation:
This filter produces a halftone from a gray input image using Ostromoukhov's method.

Input_image_0: grayscale image.

Channels: 1 channel → 1 channel

Parameters:

Reference: Ostromoukhov [8].

Example:

```
{
"type_filter": "HALFTONING_OST",
"input_image_0": "combination4",
"output_image_0": "halftoning_ost",
"input_image_1": "NULL"
}
```

3.5.4 Space Filling Curve Halftoning

Explanation:
This filter produces a halftone from a gray input image using a space filling curve (see Wong and chi Hsu [11], Velho and Gomes [10]).

Input_image_0: grayscale image.

Channels: 1 channel → 1 channel

Parameters:

- cluster_size: 1–256. Default=9.

Reference: Velho and Gomes [10] and Wong and chi Hsu [11].

Example:

```
{
"type_filter": "HALFTONING_SFC",
"input_image_0": "GRAY",
"output_image_0": "halftoning_sfc",
"input_image_1": "NULL",
"cluster_size":"default"
}
```

3.6 Dot output control

3.6.1 Dot EBG

Explanation:
This filter uses scanned dots to draw each black pixel of the input image.

Input_image_0: Image with black pixels. Each black pixel will be drawn as a dot.
Input_image_1: Optional. Grayscale image that modulates the size of the dots.

Channels: 1 channel → 1 channel

Parameters:

- pixel_density: 300PPI, 600PPI, 1200PPI. The pixel density of the dots. Default=300PPI
- modulate_dot_size: true or false. The size of dots is modulated depending on the tone of the input image. Default=false.
- black_dots: true or false. if the dots are drawn in black color only. Default=false;
- black_treshold:0-255. Treshold that binarizes the image to black and to white. If the value of the input pixel is greater or equal to this threshold value, the output pixel will be black; otherwise it will become white. Default=220;

Reference: Martín et al. [6].

Example:

```
{
"type_filter":"DOT_EBS",
"input_image_0": "wcvd",
"output_image_0": "dot_ebs",
"input_image_1": "contrast",
"pixel_density":"300PPI",
"modulate_dot_size":"false",
"black_dots":"false",
"black_threshold":"default"
}
```

3.6.2 Dot SVG

Explanation:
This filter applies vectorial dots (SVG) to each black pixel of the input image. This can only be obtained by saving the result. The rendered image is an raster approximation (dots are displayed using OpenCV).

Input_image_0: Image with black pixels. Each black pixel will be drawn as a dot.
Input_image_1: Optional. Grayscale image used to modulate the size of the dots.

Channels: 1 channel $\rightarrow$ 1 channel

Parameters:

- dot_type: circle, star, at (@). Default=circle.
- modulate_dot_size: true or false. The size of dots is modulated depending on the tone of the input image. Default=false.

Reference:

Example:

```
{
"type_filter":"DOT_SVG",
"input_image_0": "wcvd",
"output_image_0": "dot_svg",
"input_image_1": "contrast",
"dot_type":"default",
"modulate_dot_size":"false"
}
```

3.7 Contrast

3.7.1 Contrast-Bright

Explanation:
This filter changes the contrast and bright of the input image.

Input_image_0: grayscale image.

Channels: 1 channel → 1 channel

Parameters:

- contrast: 0.1-5. Factor of contrast (multiplier weight). Default=1.0;
- bright: -255–255. Factor of brightness (additional intensity). Default=0;

Reference:

Example:

```
{
"type_filter": "CONTRAST",
"input_image_0": "GRAY",
"output_image_0": "contrast",
"input_image_1": "NULL",
"contrast":"default",
"bright":"default"
}
```

3.7.2 Retinex

Explanation:
This filter enhances the local contrast. Based on the retinex theory.

Input_image_0: color image.

Channels: 3 channels → 1 channel

Parameters:

- color_restoration_variance: 0-4. Default=1.

Reference: Land [12]

Example:

```
{
"type_filter": "RETINEX",
"input_image_0": "COLOR",
"output_image_0": "retinex",
"input_image_1": "NULL",
"color_restoration_variance":"default"
}
```

3.8 Edge detectors

3.8.1 Canny

Explanation:
This filter extracts the silhouettes from a gray image using Canny.

Input_image_0: grayscale image.

Channels: 1 channel $\rightarrow$ 1 channel

Parameters:

- kernel_size: 3, 5, 7. Default=3.
- threshold1: 0–255. Default=100.
- threshold2: 0–255. Default=200.

Reference: Canny [13].

Example:

```
{
"type_filter": "CANNY",
"input_image_0": "GRAY",
"output_image_0": "canny",
"input_image_1": "NULL",
"kernel_size":"default",
"threshold1":"default",
"threshold2":"default"
}
```

3.8.2 DoG

Explanation:
This filter produces a Difference of Gaussians (DoG).

Input_image_0: grayscale image.

Channels: 1 channel → 1 channel

Parameters:

- umbral: 0–255. Default=250.
- kernel_size_big: 0–255. Default=25.
- kernel_size_small: 0–255. Default=3.

Reference: Marr [14].

Example:

```
{
"type_filter": "DOG",
"input_image_0": "retinex",
"output_image_0": "dog_high",
"input_image_1": "NULL",
"umbral":"default",
"kernel_size_big":"default",
"kernel_size_small":"default"
}
```

3.9 Blur

3.9.1 Gaussian

Explanation:
This filter blurs the image by means of a Gaussian kernel.

Input_image_0: grayscale image.

Channels: 1 channel → 1 channel

Parameters:

- kernel_size:1-255. Default=1.

Reference:

Example:

```
{
"type_filter":"GAUSSIAN",
"input_image_0": "GRAY",
"output_image_0": "gaussian",
"input_image_1": "NULL",
"kernel_size":"default"
}
```

3.10 Effect

3.10.1 Bilateral

Explanation:
This is a bilateral filter.

Input_image_0: grayscale image.

Channels: 1 channel → 1 channel

Parameters:

- kernel_size:1-50. Default=1.
- iterations:0-15. Default=0.

Reference:

Example:

```
{
"type_filter":"BILATERAL",
"input_image_0": "GRAY",
"output_image_0": "bilateral",
"input_image_1": "NULL",
"kernel_size":"default",
"iterations":"default"
}
```

3.10.2 Dilation

Explanation:
This filter produces a dilation.

Input_image_0: grayscale image.

Channels: 1 channel → 1 channel

Parameters:

- kernel_size:1-50. Default=1.
- iterations:0-15. Default=0.

Reference:

Example:

```
{
"type_filter":"DILATION",
"input_image_0": "GRAY",
"output_image_0": "dilation",
"input_image_1": "NULL",
"kernel_size":"default",
"iterations":"default"
}
```

3.10.3 Distance field

Explanation:
This filter computes the distance field using the jump flooding method implemented in the CPU. When the distance fied is obtained it is converted to bands of white and black using the division between a value and the module 2. That is:

```
Value=Distance_field(x,y)
Value=Value/Line_width;
if ((int)Value%2==0) draw(Black)
else draw(White)
```

Input_image_0: grayscale image.

Channels: 1 channel $\rightarrow$ 1 channel

Parameters:

- line_width: 1–10. Indicates the width of the obtained black and white lines. Default=3.

Reference:

Example:

```
{
"type_filter": "DISTANCE_FIELD",
"input_image_0": "GRAY",
"output_image_0": "field_distance",
"input_image_1": "NULL",
"line_width":"default"
}
```

3.10.4 Erotion

Explanation:
This filter produces an erotion.

Input_image_0: grayscale image.

Channels: 1 channel → 1 channel

Parameters:

- kernel_size:1-50. Default=1.
- iterations:0-15. Default=0.

Reference:

Example:

```
{
"type_filter":"EROTION",
"input_image_0": "GRAY",
"output_image_0": "erotion",
"input_image_1": "NULL",
"kernel_size":"default",
"iterations":"default"
}
```

3.10.5 Inversion

Explanation:
This filter inverts the intensity of the input image.

Input_image_0: grayscale image.

Channels: 1 channel → 1 channel

Parameters:

Reference:

Example:

```
{
"type_filter": "INVERSION",
"input_image_0": "GRAY",
"output_image_0": "inversion",
"input_image_1": "NULL"
}
```

3.11 Combination

3.11.1 Combination

Explanation:
This filter combines two input images using an arithmetic or logical operation.

Input_image_0: grayscale image.
Input_image_1: grayscale image.

Channels: 1 channel → 1 channel

Parameters:

- operation: product, add, and, or, sub. Default=product.

Reference:

Example:

```
{
"type_filter": "COMBINATION",
"input_image_0": "gaussian_low",
"output_image_0": "combination1",
"input_image_1": "gaussian_medium",
"operation":"add"
}
```

3.12 Measure

3.12.1 Measure SSIM and PSNR

Explanation:
This filter computes the index of the SSIM and the PSNR. It is used only for computing the measure information, so the input image is copied to the output image. The obtained values are shown in the parameters zone of the filter (in the Filters tab).

Input_image_0: grayscale image.
Input_image_1: grayscale image.

Channels: 1 channel → 1 channel

Parameters:

Reference: Wang et al. [16].

Example:

```
{
```

```
"type_filter": "MEASURE_SSIM_PSNR",
"input_image_0": "stippling_rwt",
"output_image_0": "measure_ssim_psnr",
"input_image_1": "GRAY"
}
```

3.13 Examples

Now some examples:

1. We want to obtain the stippling with the example-based method. It is necessary to produce a halftoning image.

```
{
  "effect":
  [
  { "type_filter": "HALFTONING_OST",
    "input_image_0": "GRAY",
    "output_image_0": "halftoning_ost",
    "input_image_1": "NULL"
  },
  { "type_filter": "STIPPLING_EBS",
    "input_image_0": "halftoning_ost",
    "output_image_0": "stippling_ebs",
    "input_image_1": "NULL",
    "displacement_ramdomness":"default"
  }
  ]
}
```

2. We want to improve the global contrast of the previous example

```
{
  "effect":
  [
  { "type_filter": "CONTRAST",
    "input_image_0": "GRAY",
    "output_image_0": "contrast",
    "input_image_1": "NULL",
    "contrast":"default",
    "bright":"default"
  },
  { "type_filter": "HALFTONING_OST",
```

```
    "input_image_0": "contrast",
    "output_image_0": "halftoning_ost",
    "input_image_1": "NULL"
  },
  { "type_filter": "STIPPLING_EBS",
    "input_image_0": "halftoning_ost",
    "output_image_0": "stippling_ebs",
    "input_image_1": "NULL",
    "displacement_ramdomness":"default"
  }
  ]
}
```

3. We want to compare Ostromoukhov's method for halftoning with Mould's method

```
{
  "effect":
  [
  { "type_filter": "HALFTONING_OST",
    "input_image_0": "GRAY",
    "output_image_0": "halftoning_ost",
    "input_image_1": "NULL"
  },
  { "type_filter": "HALFTONING_CAH",
    "input_image_0": "GRAY",
    "output_image_0": "halftoning_cah",
    "input_image_1": "NULL",
    "kernel_size":"default",
    "exponent":"default"
  }
  ]
}
```

4. We want to produce 2 DoG and to combine both

```
{
  "effect":
  [
  { "type_filter": "DOG",
    "input_image_0": "GRAY",
    "output_image_0": "dog_high",
    "input_image_1": "NULL",
    "umbral":"default",
    "kernel_size_big":"15",
    "kernel_size_small":"1"
```

```
    },
    { "type_filter": "DOG",
      "input_image_0": "GRAY",
      "output_image_0": "dog_low",
      "input_image_1": "NULL",
      "umbral":"default",
      "kernel_size_big":"45",
      "kernel_size_small":"23"
    },
    { "type_filter": "COMBINATION",
      "input_image_0": "gaussian_high",
      "output_image_0": "combination1",
      "input_image_1": "gaussian_low",
      "operation":"default"
    }
    ]
}
```

Programming a filter

4.1 How to program a filter

In this section, I will explain how to convert you own process or any other process to a filter that can be used by *StippleShop* .

The main idea is that a filter is like a box that transforms input images, one or two, to an output image. They must be raster images. If our algorithm works in that way, it can be easily included in *StippleShop* .

A filter must be created as a class that is derived from the base class _filter. This class has the basic components of any filter, particularly the pointers to the two possible input images, and to the output image. A filter must have at least one input image and one output image. Some filters can have two input images. Images are referenced by pointers. This implies that an image can be shared as an input by several filters. In such a way, for example, if our effect has 5 filters there will be five images, but they can be used more than five times. Each effect always have two fixed images that represent the initial input image that will be loaded by the user to obtain a final result: the COLOR and GRAY images, representing the color, 3 channels, and the gray, 1 channel, versions of the loaded image. If the loaded image is a color one, the system will create the gray version. In the loaded image is gray scale, the COLOR image will also be gray scale but with 3 channels.

For managing the images, we have used the class Mat of the OpenCV library. This is not only for having a container but also to allow applying all the functionality of the library, saving much work.

The process to get a functional result must follow these steps:

1. Include information in filter.h
   The information of the new filter is added to the file filter.h. Particularly it is necessary to add an identifier for the filter in the enum _type_filter. Follow the rule of using first the word FILTER_ and then the name of the filter. Some examples are FILTER_CANNY or FILTER_DOT_EBG.

   Then, you have to establish a relation between the identifier of the filter, as an integer, and the name of the filter, as a string. This is done with the data structure map called Type_filter_name. Some examples are {"CANNY",FILTER_CANNY} or {"DOT_EBG",FILTER_DOT_EBG}.

   The third step is to define some data for being used in the node editor. The data is included in the map Filter_name_parameters. For each filter is necessary to define:

   - the name of the filter as it will appear in the drawing block, in upper case. This name is the same for all the instances of the same filter.

- the name of the filter as it will appear in the drawing block, in lower case. This name is unique for each instance of the same filter. The names is completed with an unique number.
- number of parameters.
- number of input ports: one or two.
- number of channels of the input or inputs ports.

One example is {FILTER_CANNY,{"CANNY","canny",3,1,1}}.

Finally a helping text is included using the map Filter_name_text. This text will appear as a tooltip when the mouse is over the name of the filter. It is written using html tags. One example is {FILTER_CANNY,{"Canny"," This filter applies Canny edge detector Input: Grayscale image Output: Grayscale image "}}.

2. Create the class for the filter
   A .h and a .cc (or .hh and .cpp,...) files must be created to define and implement the filter. One easy way is to copy one of the created filters and use it as a template.

   - Definition
     It is important to take into account that a filter is made of three classes: the base class that implements the main process and maintains all the parameters that control it, the qtw class that implements the user interface based on the Qt widgets, and the ui class that connects the base class with the qtw class.

     For example, for the implementation of the Canny filter we have the _filter_canny class, the _filter_canny_ui class, and the _qtw_filter_canny class.

     The goal of this structure is to facilitate an easy change to other widgets library. It also allows to simplify the coding as we will comment below.

     base class
     The base class must have a descriptive name. The class must derive from the _filter class. Some functions are common to all the filters: reset_data(), change_output_image_size(), use_dots(), and update().

     The constructor must be defined and implemented. The other things that must be included are the parameters that control the process and the functions that allow to set and get the values for that parameters, particularly those that can be changed by the user. For example, in the Canny filter the parameters are: Canny_size, Threshold1, and Threshold2.

     ui class
     Most functions of this class are the same for all filters, because they control the widgets, allow to create the filter, and to save and load the parameters to files. We have also to define and implement the functions that connect the parameters of the base class with the parameters of the qtw class. This is done in a special way: the names of the functions are always the word parameter plus the number

of the parameter (as it is ordered). For example, if the base class have 3 parameters that control the process that can be changed by the user, then the ui class will have parameter1, parameter2, and parameter3 functions. Of course, there will be a function to set the value and another to get the value.

For example, to set and get the parameter Canny_size, which we have assigned the first position, we have these functions: void parameter1(int Value) and int parameter1(). The same for the other parameters.

qtw class

This class implements the interaction with the user using Qt widgets. Depending on the parameter of the base class, we can change its value by using one of the several widgets that Qt has, as for example, sliders, buttons, spinbox, etc., depending on the user needs.

It is important to note that every widget that controls a parameter is included in a QGroupBox that allow to assign a descriptive text. There is a global QGroupBox that includes all the other QGroupBoxes. This allows to show or remove all the widgets of the filter depending on the filter selected by the user.

The parameters that control and the text that appears in the widgets are defined in the .h file, inside the namespace of the filter, using the same procedure of using the order numbers for assigning the names. For example, the namespace of the Canny filter is _f_canny_ns. The parameter Canny_size is the first one and it is controlled with a slider. The corresponding QGroupBox is Group_box_parameter1. The corresponding widgets are Slider_parameter1 and Line_edit_parameter1. The text of the slider and minimum and maximum values, as well as other parameters, are defined in the namespace.

Of course, it is necessary to implement the functions that set and get the values of the parameters of the widgets. For setting the values we have the numbered functions. For example, for the first parameter the function is void set_parameter1(int Value). The change of a parameter in a Qt widget is managed with a slot function (signals and slots are the particular way of managing events in Qt). So, we need to define a slot function for each parameter. For example, for the first parameter the slot function is void set_parameter1_slot(int Value).

These set and get (slot) functions are connected to the set and get functions of the ui class as well as the class that manages all the filters.

- Implementation

  base class

  We must implement the constructor in the corresponding .cc file. This function must do the initialization of the common parameters of all filters as well as the particular parameters of the current filter. Particularly, it is necessary to initialize the filter type using the definition made in filter.h.

  The reset_data function must reset the values of the own parameters.

  The most important function is update. As its name implies, it will do the work of updating the result of the filter by applying the particular process of the

filter. This function realizes some checking with the input and output images, as well as the appropriate conversion to the desired number of channels. Then, the function that applies the own process is called. For example, to apply the Canny filter a function called canny could be called. In our implementation, as OpenCV has such implementation we only need to make the call. If it is necessary, the number of channels of the result is adjusted to fit to the output image.

For the base class, it is also necessary to implement the functions to set and get the values of the parameters.

ui class

The important functions to implement in the ui class are those that allow to read and save the parameters from/to the disk files. We have used the JSON format to do it. In the case of reading the values of parameters there are two possibilities: to get them from the node editor or from a file. In the first case the values are the default ones defined in the namespace of the .h file. In the second case, the values are read from a file, and so, we must check that the values are correct. In a JSON file, each parameter is defined as couple of a name and its value. The name identifies the parameter to which the value is assigned. One possibility for the value is the word default: in such case the default value will be assigned.

The function for writing the values is easier as it only converts the values to strings.

qtw class

The constructor of the qtw class is responsible of creating all the needed widgets as well as to connect the signals and the slots.

The slot functions are called by the system when the user interacts with one of the widgets, for example, when she moves one slider. In such case, the slider will shoots a signal which makes that the function assigned to the corresponding slot to be called. If a slot function is called, implies that we must change one of the parameter of the filter and update the result of such filter. But as we can have connected several filters, the change in one of them could imply a chain effect of change, in such a way that we need to update all the filters that are affected by the current change. This is achieved in the GL_widget class, which is the main responsible of all the management and visualization processes.

For example, if the user moves the slider corresponding to the Canny's parameter Canny_size, the corresponding slot function set_parameter1_slot(int Value) is called by the system. This function converts the int value to a string that is also shown in the widget as feedback information. The value of the parameter Canny_size is changed, but it cannot be done directly but using the intermediate function defined in the ui class. Finally, we call to the GL_widget class with the identifier of the filter that need to be updated. GL_widget class will update all the necessary filters in the correct order to obtain the final result.

3. Add the information of the filter to the selection tab

For adding the new created filter to the list in the tab of the node editor, it is necessary to add some code to the function create_filter_list() of the _window class in the window.cc file. The asy way is to copy the code of one of the implemented filters.

4. Include the filter in the main process
   The _gl_widget class does all the creation, edition and visualization of the filters and the corresponding images. Once we have created the code of a filter, we must include it in the function create_filters_from_blocks() that is in glwidget.cc. We will include a new case for the switch instruction using the identifier of the filter. Each filter is created as a pointer and saved to map structure that maintains all the created filters. For most filters, the creation and saving is achieved with only one instruction. Only when the size of the input/s image/s is/are different of the output image some more code is needed.

5. Add to the project
   Finally, the .h and .cc files must be included in the .pro file.

Bibliography

[1] Secord, A.. Weighted Voronoi stippling. In: Proc. NPAR. New York: ACM; 2002, p. 37–43. doi: 10.1145/508530.508537

[2] Kopf, J., Cohen-Or, D., Deussen, O., Lischinski, D.. Recursive Wang tiles for real-time blue noise. ACM Transactions on Graphics 2006;25(3):509–518. doi: 10.1145/1141911.1141916

[3] Balzer, M., Schlömer, T., Deussen, O.. Capacity-constrained point distributions: A variant of Lloyd's method. ACM Transactions on Graphics 2009;28(3):86:1–86:8. doi: 10.1145/1531326.1531392

[4] Li, H., Mould, D.. Structure-preserving stippling by priority-based error diffusion. In: Proc. Graphics Interface. Waterloo, ON, Canada: Canadian Information Processing Society; 2011, p. 127–134. doi: 10.20380/GI2011.17

[5] Kim, S., Maciejewski, R., Isenberg, T., Andrews, W.M., Chen, W., Sousa, M.C., et al. Stippling by example. In: Proc. NPAR. New York: ACM; 2009, p. 41–50. doi: 10.1145/1572614.1572622

[6] Martín, D., Arroyo, G., Luzón, M.V., Isenberg, T.. Example-based stippling using a scale-dependent grayscale process. In: Proc. NPAR. New York: ACM; 2010, p. 51–61. doi: 10.1145/1809939.1809946

[7] Martín, D., Arroyo, G., Luzón, M.V., Isenberg, T.. Scale-dependent and example-based stippling. Computers & Graphics 2011;35(1):160–174. doi: 10.1145/1809939.1809946

[8] Ostromoukhov, V.. A simple and efficient error-diffusion algorithm. In: Proc. SIGGRAPH. New York: ACM; 2001, p. 567–572. doi: 10.1145/383259.383326

[9] Li, H., Mould, D.. Contrast-aware halftoning. Computer Graphics Forum 2010;29(2):273–280. doi: 10.1111/j.1467-8659.2009.01596.x

[10] Velho, L., Gomes, J.d.M.. Digital halftoning with space filling curves. SIGGRAPH Comput Graph 1991;25(4):81–90. doi: 10.1145/127719.122727

[11] Wong, T.T., chi Hsu, S.. Halftoning with selective precipitation and adaptive clustering. In: Paeth, A.W., editor. Graphics Gems V. Academic Press; 1995, p. 302–313.

[12] Land, E.H.. The retinex theory of color vision. Scientific American 1977;237(6):108–128.

[13] Canny, J.. A computational approach to edge detection. IEEE Transactions on Pattern Analysis and Machine Intelligence 1986;8(6):679–698. doi: 10.1109/TPAMI.1986.4767851

[14] Marr, D.. Vision. The MIT Press; 1982. ISBN 978-0-262-51462-0.

[15] Kang, H., Lee, S., Chui, C.K.. Flow-based image abstraction. IEEE Visualization & Computer Graphics 2009;15(1):62–76. doi: 10.1109/TVCG.2008.81

[16] Wang, Z., Bovik, A.C., Sheikh, H.R., Simoncelli, E.P.. Image quality assessment: From error visibility to structural similarity. Trans Img Proc 2004;13(4):600–612. doi: 10.1109/TIP.2003.819861