PixInsightMagazine
01/2011

PixInsight Magazine Issue 01 February 2011

STAFF AND CONTRIBUTORS

Editor in Chief
Vicent Peris
Observatori Astronòmic de la Universitat de València

Editor Assistants
Carlos Milovic
Pontificia Universidad Católica de Chile
Niall Saunders
Clinterty Observatories, Aberdeen, Scotland

Design / Creative Directors
Juan Conejero
Pleiades Astrophoto S.L.
Vicent Peris
Observatori Astronòmic de la Universitat de València

Scientific Consultants
Fernando J. Ballesteros
Observatori Astronòmic de la Universitat de València
Joseph De Pasquale
Harvard-Smithsonian Center for Astrophysics
Chandra X-ray Observatory
Emilio J. García
Instituto de Astrofísica de Andalucía / CSIC
Robert Hurt
Jet Propulsion Laboratory / California Institute of Technology
Spitzer Space Telescope

Contributors
Jack Harvey / Steven Mazlin / Gerhard Bachmayer / Vicent Peris / Juan Conejero / Carlos Milovic / Máximo Ruiz / Jordi Gallego / Stephen Leshin / Stanislav Volskiy / Jon Talbot / Sergi Verdugo

Publisher
Pleiades Astrophoto S.L.
Apartado 204
46185 Pobla de Vallbona (Valencia)
Spain
www.pixinsight.com

ISSN: Pending

magazine.pixinsight.com
magazine@pixinsight.com

Copyright © 2011 Pleiades Astrophoto. All rights reserved.

PixInsight™, PixInsightMagazine™ and the PixInsight™ pixel logo are trademarks of Pleiades Astrophoto S.L. All other trademarks are the property of their respective owners.

Made on Linux® with PageStream™ 5.0 Pro desktop publishing software.
www.pagestream.org

Cover Image
NGC 2359 by Steven Mazlin and Jack Harvey at CTIO Observatory (Chile). 16-inch f/11.3 Ritchey-Chrétien telescope and Apogee U9 camera. Total exposure time: 26 hours (RGB 2 hours per channel, OIII 8 hours, Hα 12 hours)

ISSUE**011**

FEBRUARY**2011**

PixInsightMagazine

08
Editor's Letter
Vicent Peris

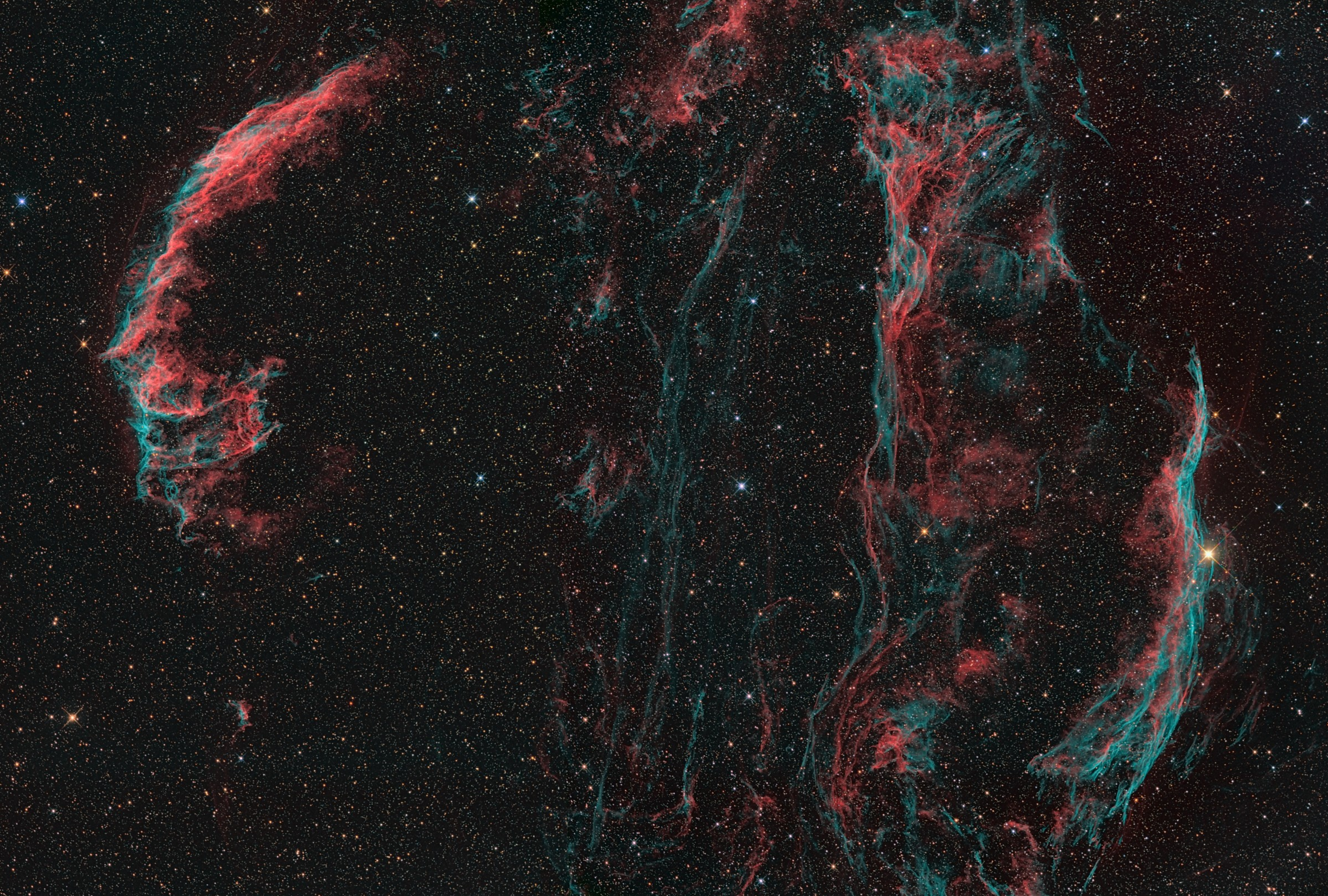
10
Astrophotography with Wavelets (I)
Vicent Peris

26
Introduction to PixInsight Module Development
Juan Conejero

58
PixInsight as a Research Platform
Carlos Milovic

64
Community News

72
Image Gallery



EDITOR'S **LETTER**

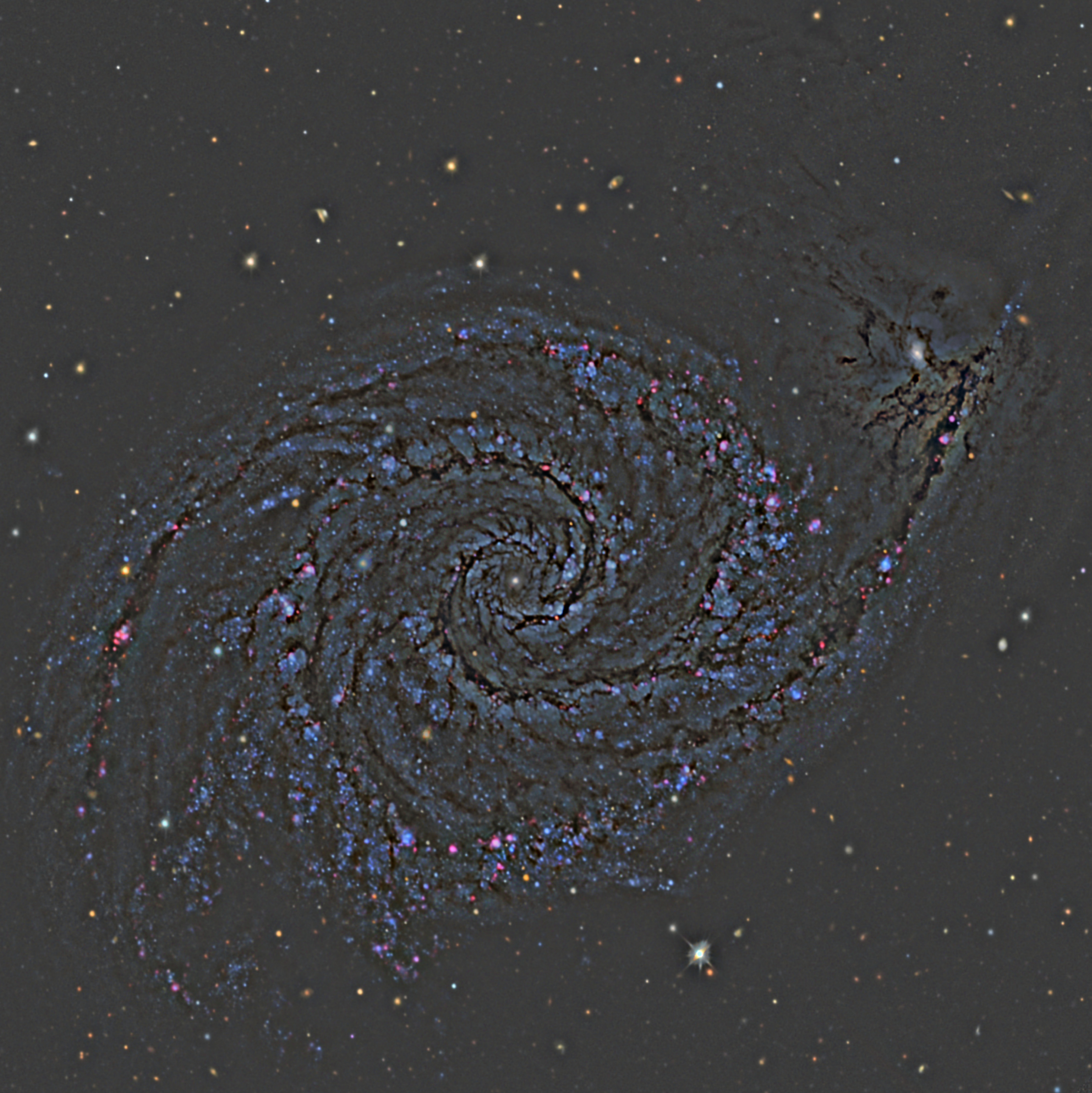
Here we start a new project, a new challenge. This magazine is a consequence of the creativity and innovation that is happening today in the dynamic and evolving community generated around PixInsight. Day to day, users and developers generate an increasing amount of resources including images, learning material, research works, development projects and a variety of technical documents. Therefore, all PixInsight users need an appropriate communication medium to share their works and knowledge, and we hope PixInsight Magazine will serve as a vehicle for that goal.

This doesn't mean that we are limited to the PixInsight community. With this magazine we want to establish an electronic publication open to the whole astrophotography world, suitable to gather multidisciplinary contributions from both the amateur and professional astronomy communities. For that purpose, PixInsight Magazine's subject matter is broad by design: astrophotography, image processing —from practical processing tutorials and examples to theoretical works—, software development, instrumentation, astronomy and astrophysics, science outreach and related disciplines. It is open to everybody, free and publicly available, with a focus on quality, modern design, innovation and advanced contents.

Great projects always start from the need. We feel that PixInsight users —including ourselves— and the astrophotography community need a publishing line like the one of this magazine. We hope you will find in PixInsight Magazine a source to cultivate and inspire your passion.

Vicent Peris, Editor in Chief

Pages 6 and 7
NGC 6960, NGC 6992, NGC 6995 and IC 1340 supernova remnant complex by Gerhard Bachmayer.
ASA 10N (10" f3.6) and FLI PL16803 camera. Total exposure time of 360 minutes (R, G, B, H-alpha and O-III filters).



Astrophotography with Wavelets (I)

Vicent Peris

This article series summarizes the work associated to multiscale processing techniques that I have been developing during the last decade. In this first installment I will give an overview with examples of the techniques that will be covered in coming articles.

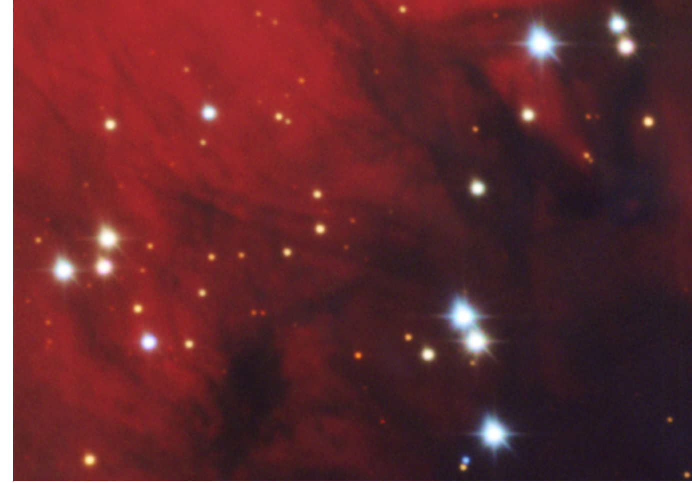
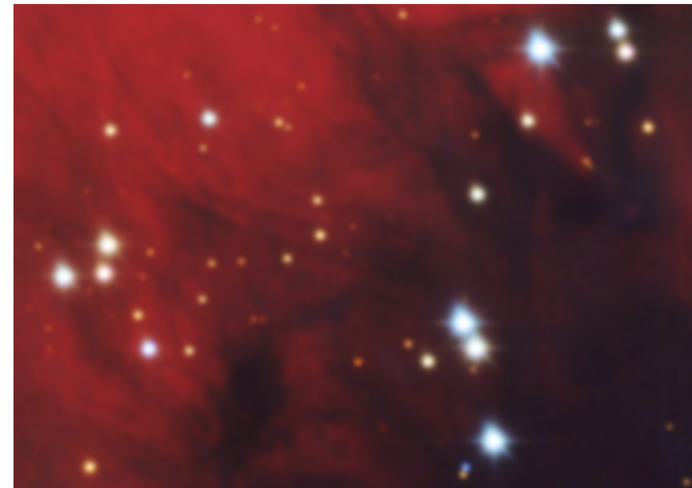
My work is focused on solving practical problems with new image processing algorithms and techniques. This connects directly with two key goals of any astrophotography: communicate the documentary value of the scene, and achieve that on an aesthetic and perceptual basis.

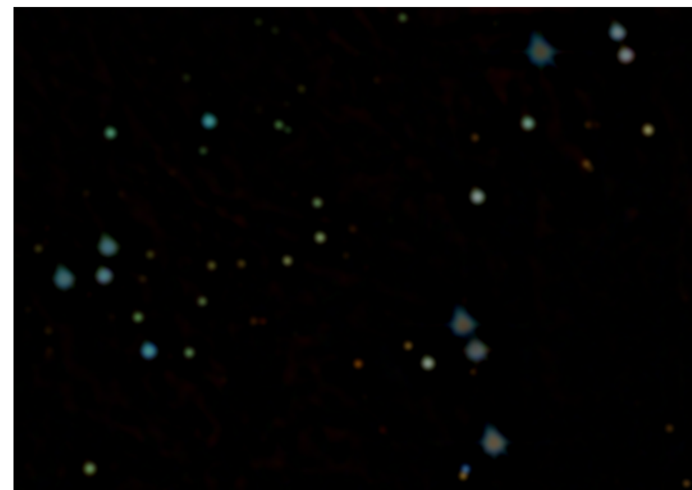
The Starting Point

Being a pianist, my research began from an aesthetic point of view. I started working with multiscale techniques before knowing anything about the multiscale concept. In 2003 I published an article on the former Pleiades Astrophoto's website, entitled *Masked Unsharp Mask* (MUSM). This was the first article I wrote on image processing, and the only one using Adobe® Photoshop®. Not very known at that time, through the years this article has had a large influence on how we understand image processing in astrophotography.

Most algorithms and techniques have their downsides, so learning them means controlling how they affect the image. As is well known, algorithms working with convolutions tend to suffer from the Gibbs effect, or *ringing*. Furthermore, if the algorithm performs a local contrast enhancement, high-contrast structures tend to saturate quickly. The «masked unsharp mask» technique used a mask to filter out the high-contrast, small-scale structures (such as the stars) when applying an unsharp mask filter. MUSM used what we know today as a «star mask» in order to prevent the Gibbs effect and saturation to happen around these structures.

The point of view of MUSM had nothing to do with the classical star shaping methods. The true revolutionary concept about MUSM was not in the mask, but in the conceptual change about image processing: structures with different properties require different processing methods.

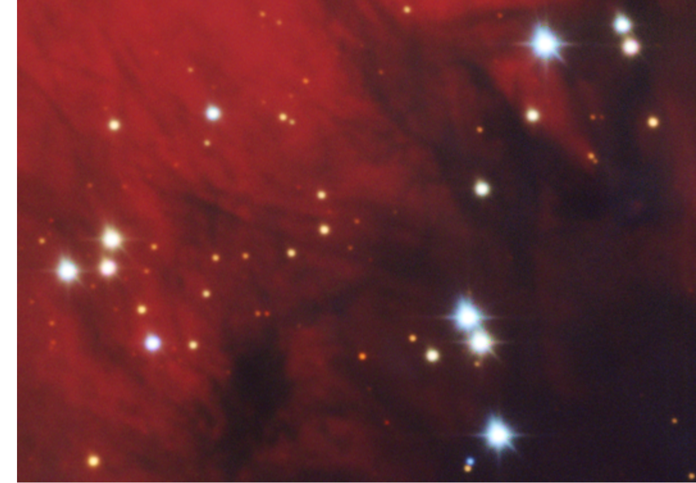
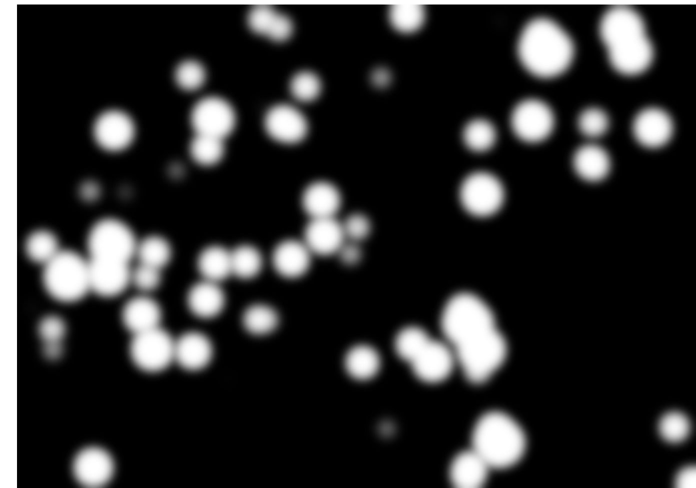
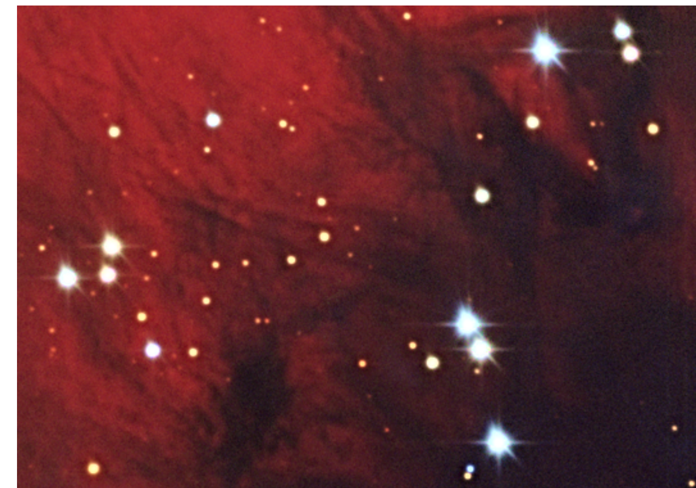
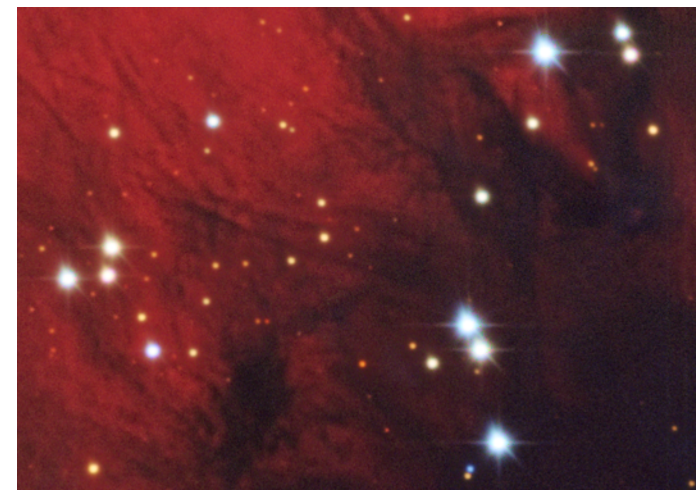





Page 10: Small scale structures of an image of Messier 51, isolated with the Local Contrast Normalization Function. Note that the dark rings around high contrast structures are complemented with the images of larger structure components.

Left: The generation of a star mask in Adobe® Photoshop® was carried out with what I called «differential Gaussian blur». Two duplicates of an image were convolved with Gaussian filters of different standard deviations. After subtraction of the more defocused image from the other one, the resulting image had only structures of a given size. This way, stars could be easily selected to generate a star mask. From top to bottom in this figure, first we have the original image. Starting from this image we generate two more images which are convolved with gaussian functions of different standard deviation (in this case, 2 and 8 pixels). The result of subtracting the *more defocused* image to the *less defocused* image is shown at bottom.

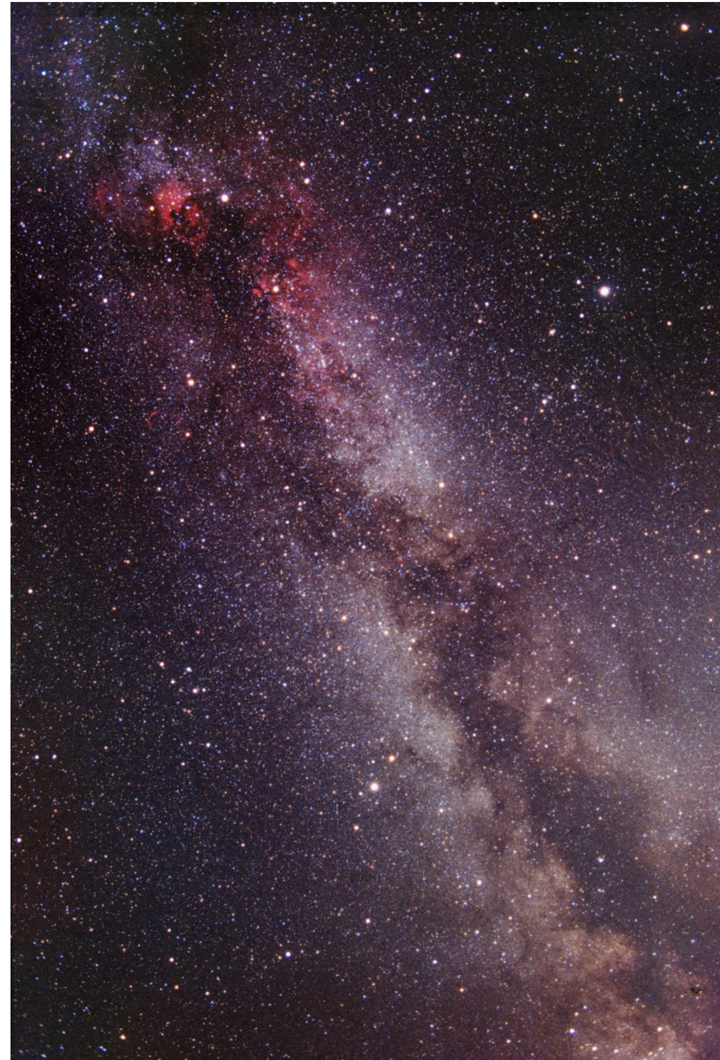
Right: The inverted mask is used to protect the stars from the USM filter. From top to bottom, first we have the original image; the second image is the associated star mask. The two lower images are comparison between the USM and MUSM techniques. The MUSM technique allows to control the Gibbs effect and the saturation of high brightness and high contrast structures in the image — the usual case of the stars in an astronomical image.

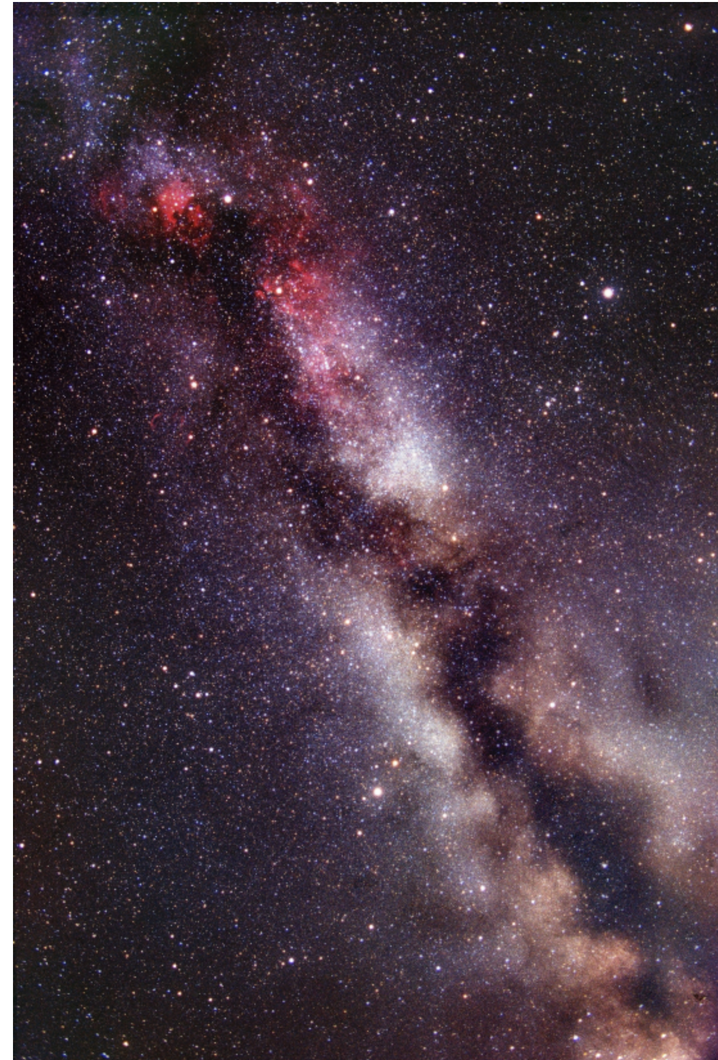





Large Scale Processing Techniques

From then on, my research continued with the newly born PixInsight platform, because at that moment, all members of the PixInsight development team needed to go further than what Photoshop® was allowing us. In my research there have been two key moments. The first one was with the new ATrousWaveletTransform tool when I had the idea to raise the bias of the 128-pixel scale to +0.2. Visually speaking, this simple action made a change in the image that I had never seen before, and focused my eye on features that were barely visible before. Thus, my first steps in multiscale processing were made developing techniques to process large scale structures. These techniques were focused on the problem of reflection of small structures through increasing scale wavelet layers, with the intention of recovering low surface brightness structures.

In some scenarios, large-scale processing can improve the image significantly. The typical case is a dense star field where all the stars are hiding a diffuse nebula. These techniques allow us to separate the large-scale structures —the nebulae— from the smaller ones —the stars and other small-scale objects—. We will see in coming articles the techniques that I developed to enhance the separation of objects by their scales. These techniques were the logical evolution of MUSM: we don't use masks to apply different processes to different objects; we use masks to better separate objects, and the result of this separation is two different images which we process separately.





Film image of the Summer Triangle. The right image shows the effect of increasing the bias of the 128 and 256-pixel scales by +0.25.

Dynamic Range Compression

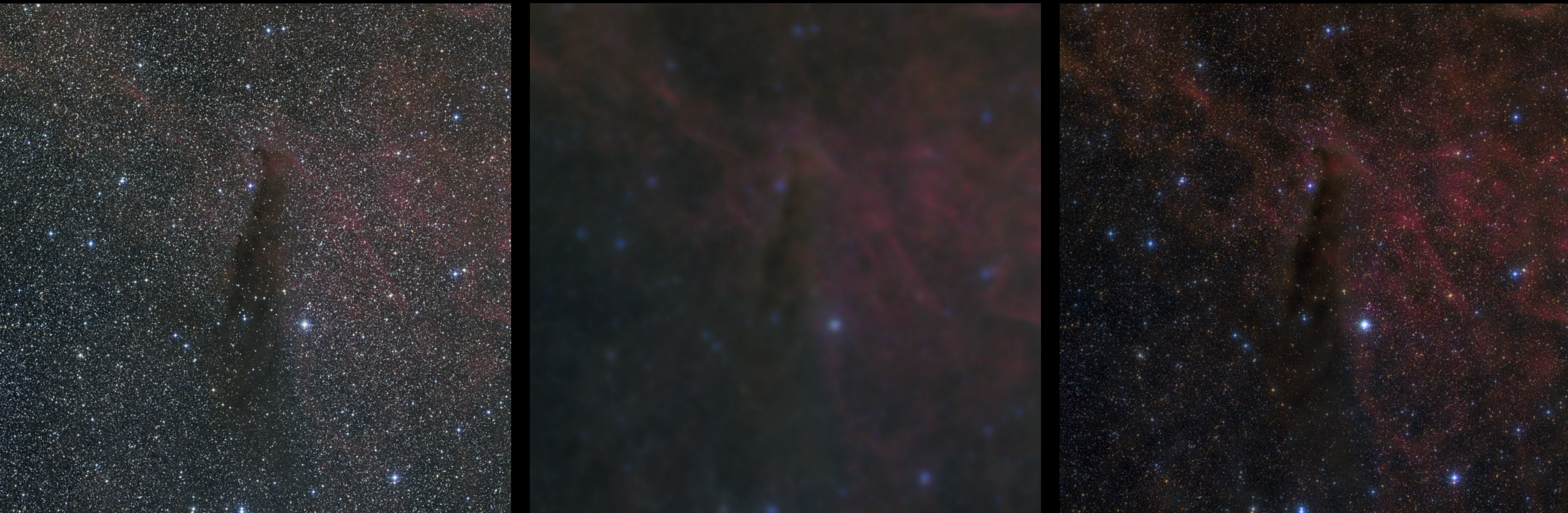
The second key moment in my research is associated with the dynamic range problem. It was at the time of making my first photo of the Orion Nebula. My intention —even before begin acquiring data— was to bypass the classical core-masking techniques to compress the dynamic range of the object. I started from a very specific premise: to have the object *as is*. And then, work on this basis to find a method to compress the dynamic range of the image.

The main difference between analog and digital photography is the linearity of data. This means that the relative intensity between two pixels is kept independently of their values. This is extremely important to understand what is documentary photography and, more specifically, astrophotography. The ability of the digital camera to count the light gives us the opportunity to have the objects themselves, before being translated into a visual image that our eyes can read. This has become the keystone in my work, because no adequate processing can be done if we don't start from an image that preserves the original properties of the objects.

Starting from this point, I designed a high dynamic range (HDR) composition algorithm to build a linear HDR image. The algorithm, now implemented in PixInsight (the HDRComposition tool), works in a pyramidal way, in the sense that saturated areas are replaced by linear data from unsaturated, shorter exposures.

The output of this algorithm is the best approximation to how the light of the photographed objects would be detected having a camera with a well depth large enough to capture the full range of light intensities in the scene. This is very useful in astrophotography, because starting from the object *as is* allows us to apply processing techniques specifically designed to work with linear data: noise reduction, deconvolution, color calibration, etc. In the example of pages 18 and 19, we are simulating a camera with an ADC of ~21 bits (a single image has 12 bit depth) and a well depth 720 times higher than the original sensor photosites.

The HDR linear image always needs to be accommodated to the dynamic range of the displaying media. I found the solution to this problem again within the multiscale context, working with the delinearized image. The core of this solution is the *Local Contrast Normalization Function* (LCNF). Contrarily to classical band-pass filtering solutions, where local contrast depends on pixel values from local structures at larger scales, this function makes these two factors independent each other. In simpler words, local contrast doesn't depend on local illumination levels; therefore, the dynamic range of an object can be compressed independently of its size. *//... Continued on page 20*



A GOOD EXAMPLE OF LARGE-SCALE PROCESSING is this image of Barnard 145 from POSS II plates. The image at left is the original one. From this image, we subtract the stars (at the center) to unveil the almost hidden nebulae. Actually, we are not unveiling any new data, but making it visible to the eye because now it is not lost among the high contrast stars that cover the whole field of view. The image without stars is then processed and merged with the original image. The result, at right, is an image in which the low surface brightness nebulae have been greatly enhanced, but having the original high-contrast structures relatively unmodified.

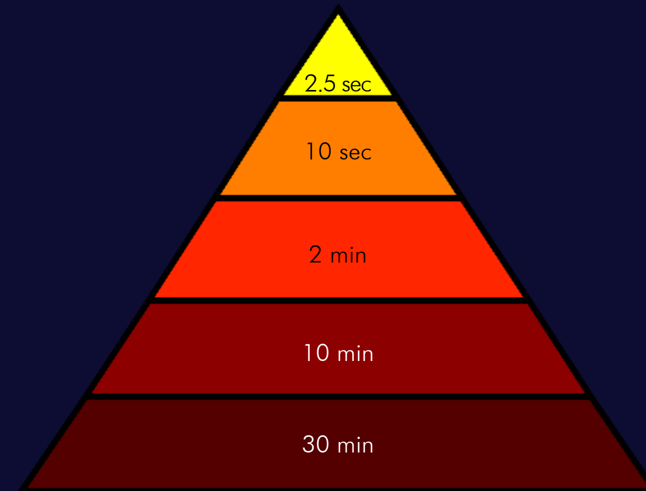
Dynamic Range Compression Techniques at a Glance

This page, bottom: The HDR composition algorithm is used to compose an image in which the subject has a larger brightness range than what the camera is able to capture. The Orion Nebula is the classical example, where extremely short exposures are needed to acquire the data around the Trapezium. In this case, exposures ranging from 2.5 seconds to half an hour were needed to record the full brightness interval of the objects in the scene.

This page, right: This simple graphic illustrates how the HDR composition algorithm works. Shorter exposures are superimposed over saturated areas of the longer ones. It is important to bear in mind that the result is a linear image. This allows us to apply processing techniques designed to work with linear data: deconvolution, color calibration or gradient extraction, among others.

Facing page: After the HDR composition, a histogram transformation allows us to see the darker areas of the scene. The Local Contrast Normalization Function is then used to compress the dynamic range of the nonlinear data. The facing page shows how the multiscale processing modifies the bright parts of the nebula to reveal the hidden structures in the core.

Original data: Vicent Peris / José Luis Lamadrid.



Towards a Multiscale Processing Methodology

The Local Contrast Normalization Function is not only the core of a dynamic range compression algorithm. It completes a multiscale-based processing methodology, as a result of merging this function with the large-scale processing techniques.

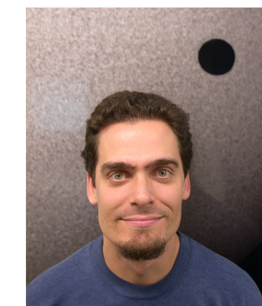
This methodology has three main stages. The first one is to divide the image into increasing scale components with the contrast normalization equation. After separation into scales, we have several images, each one containing different significant information of the image. All of these images are processed separately because structures with different properties require different processing methods. Finally, the processed images are recombined together with the original image. This methodology is briefly illustrated in the next pages with an image of NGC 6914.

Next Chapters

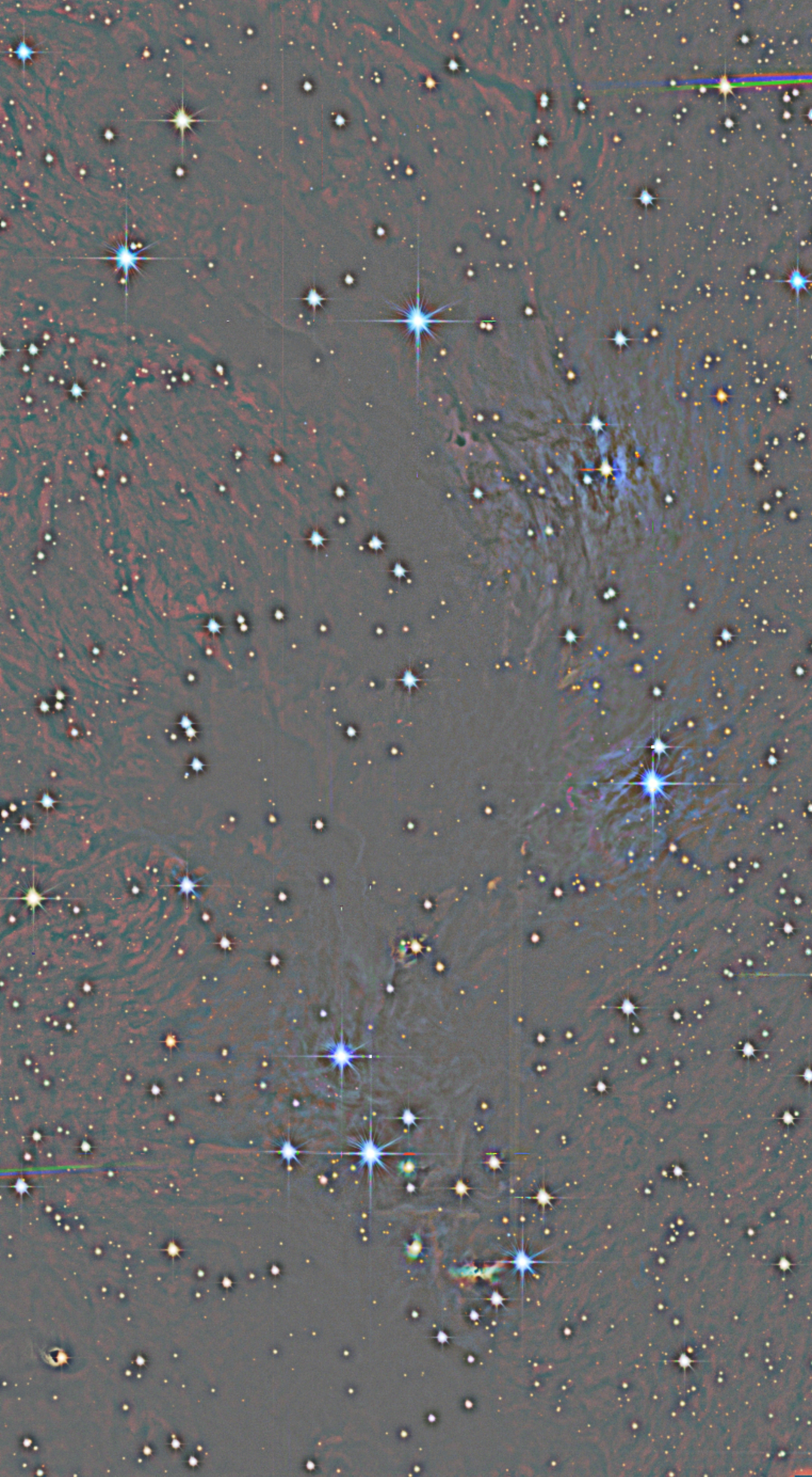
In the coming PixInsight Magazine issues we will do an in-depth review of these techniques. We'll begin with large scale processing techniques. Of great importance for this topic are the associated masking techniques, which are essential to process these structures correctly.

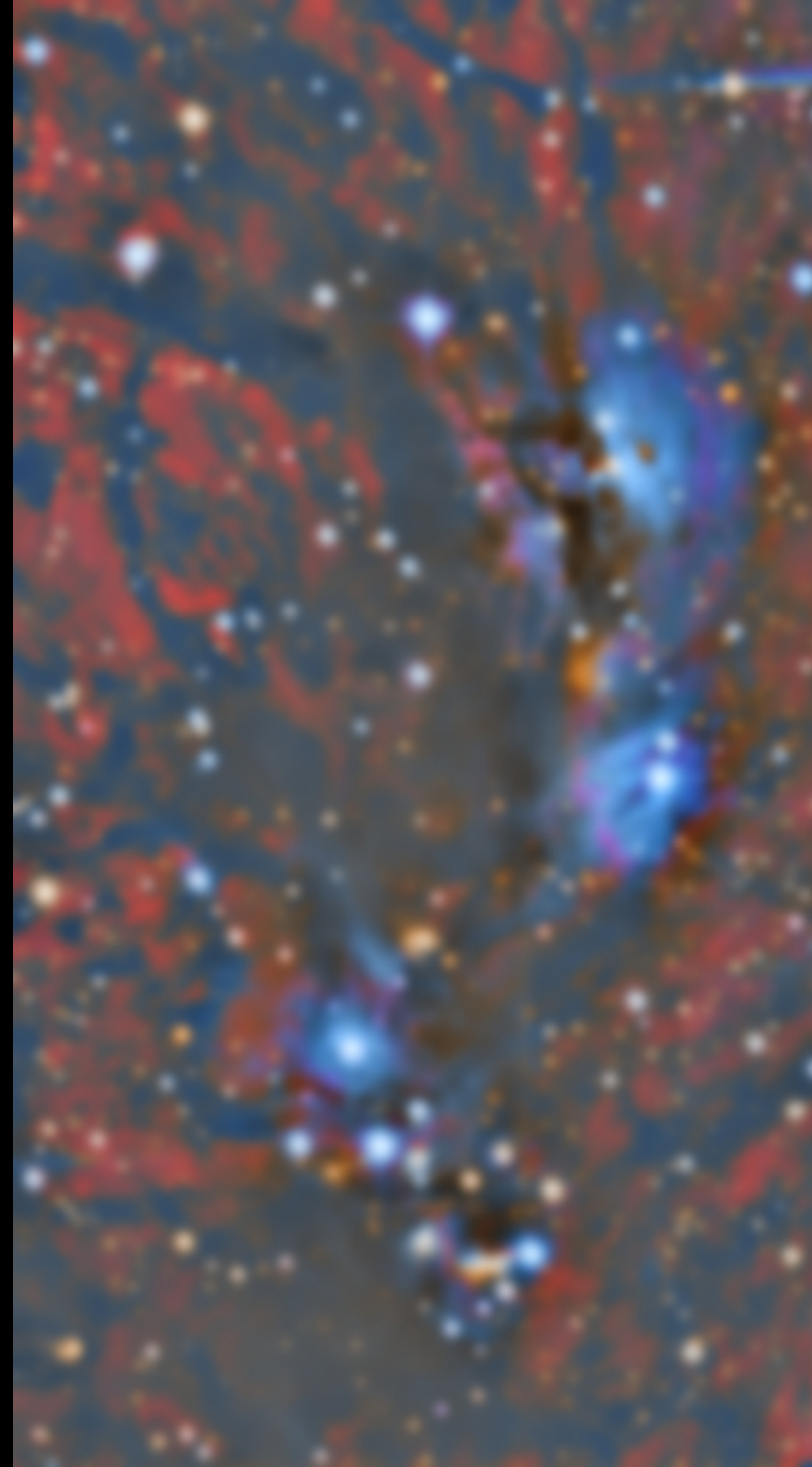
The articles will be illustrated with the required practical examples for a better understanding, because each image requires a different application of the techniques described. This means that they must not be taken as a mere collection of recipes. The goal of this article series is to provide the reader with a set of fresh ideas to find the appropriate methodology for each one of his/her works.

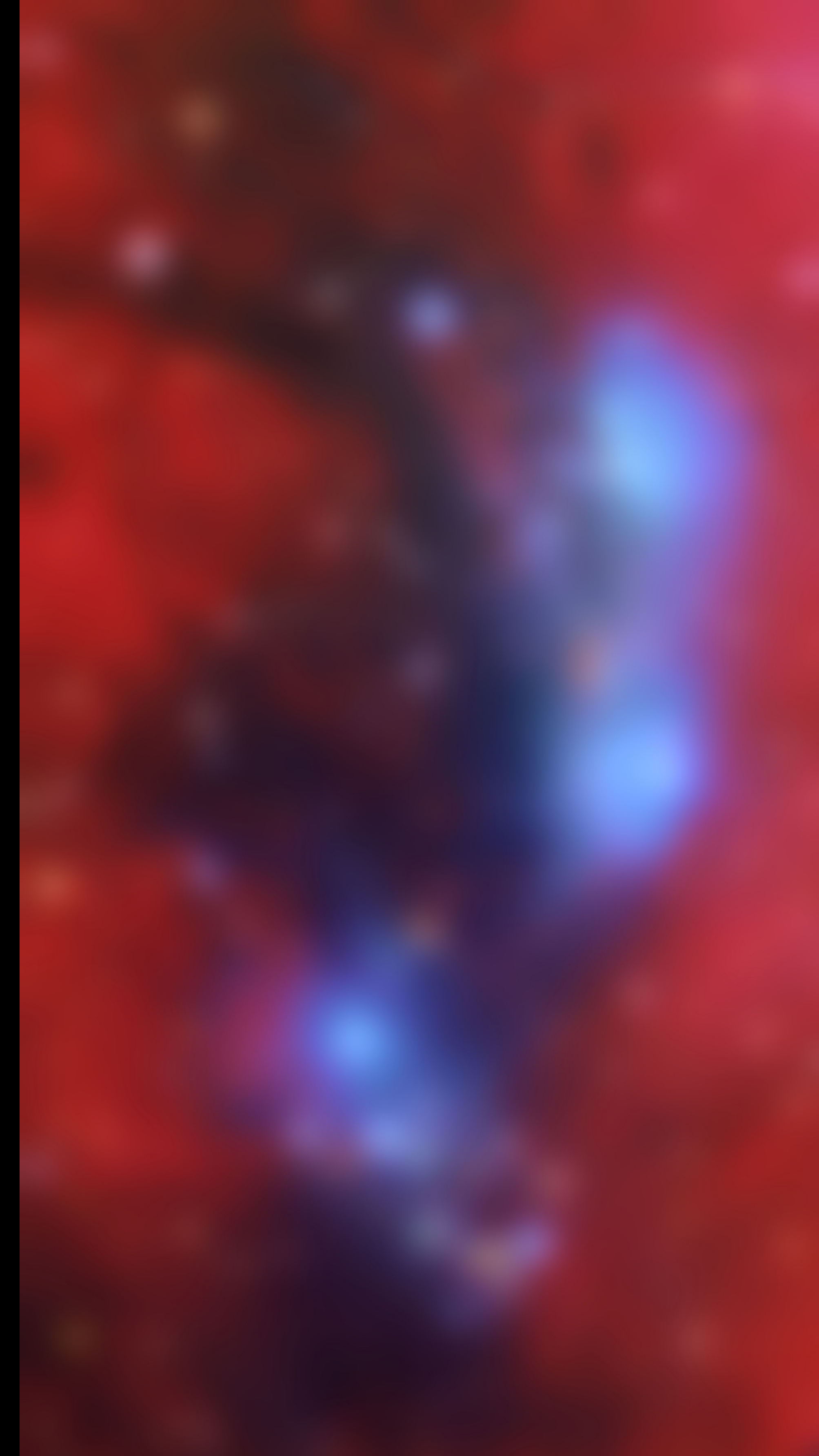
In this example of NGC 6914, a multiscale processing procedure has been applied to the right image. Pay attention to the areas with low-contrast detail. The processing recovered the detail inside the reflection nebulae, specially in their brighter areas. The dark structures around the bright star in NGC 6914b are greatly enhanced; also the small Hα emitting objects to the left of NGC 6914 are now much more visible. At the same time, there is a contrast increase in the Hα regions, where a lot of low-contrast structures become visible. Another advantage of this methodology is that we can manage color saturation much better, without an excessive increase in chrominance noise, because we work on the larger scale structures. Photo: CAHA / Descubre / DSA / OAUV.



Vicent Peris works as an astrophotographer at the Astronomical Observatory of the University of Valencia, Spain. He is a founding member of the Documentary School of Astrophotography and an active PTeam member. Vicent has contributed numerous algorithms and techniques to PixInsight, including the remarkable HDRWaveletTransform tool.



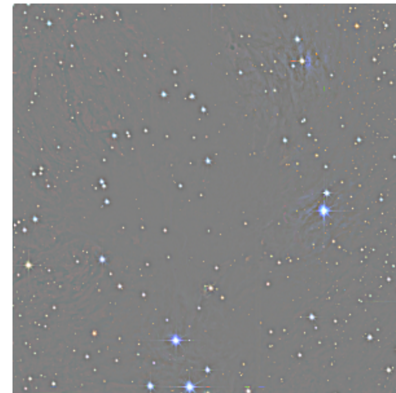


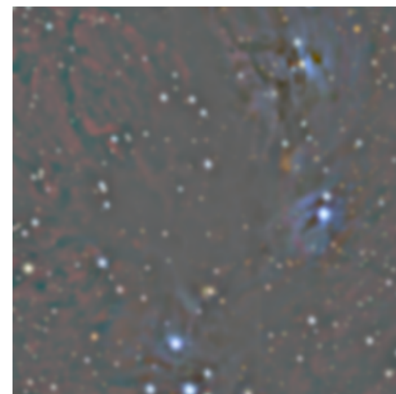


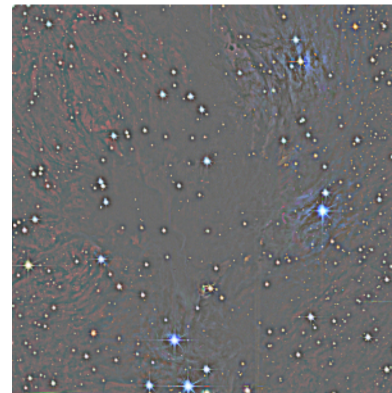
To perform the multiscale processing, the original image is divided in several images, each one containing structures of a given structure size interval. Here we can see the generated images with small, mid an large structure components. The scale separation has been made with the Local Contrast Normalization Function for the two left images. To generate the image at right, we remove the first eight wavelet layers (up to 128-pixel scale).

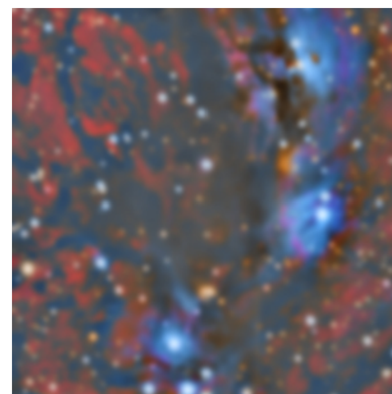
Summary of the Multiscale Processing Methodology

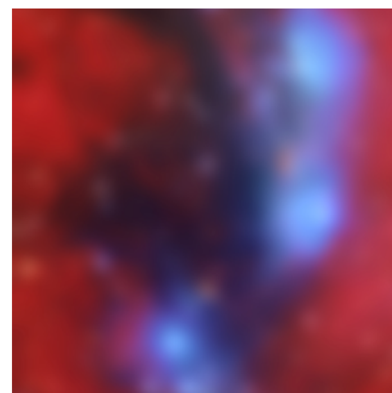
Under this text we can see the first stage of the multiscale processing methodology. The original image is divided into several images, each one containing structures of increasing sizes. The segmentation by structure sizes is highly dependent on the original image and on the properties of the photographed objects.



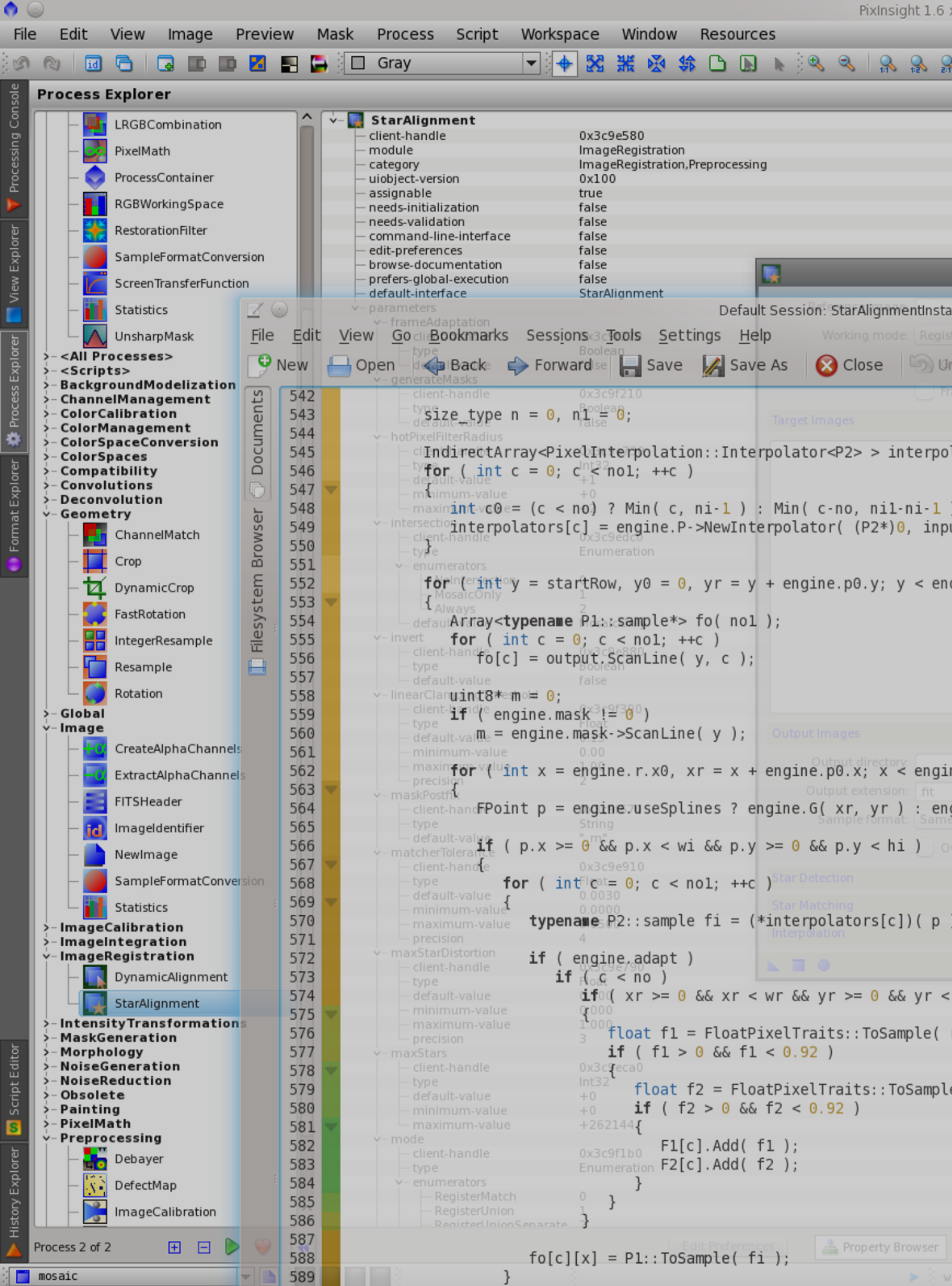








After scale division, we process all the images separately. Usually small scales are used to sharpen the image, and images containing larger scale structures are used to increase color saturation. Finally, the three scale-component images are mixed with the original image to create the final result. The original image is used in this combination as an equilibrium reference.



Introduction to PixInsight Module Development

Juan Conejero

PixInsight is both an image processing application and a development platform. PixInsight provides two different development frameworks: the PixInsight Class Library (PCL) and the PixInsight JavaScript Runtime (PJSR). PCL is a high-level C++ framework for development of PixInsight modules, while PJSR is an ECMA-262-3 compliant scripting environment embedded in the PixInsight Core application. This article is a general introduction to module development with PCL.

If you are a C++ developer willing to create your first PixInsight tools, then this article will help you start working in the right direction, providing you with the basic elements necessary to design and write your first modules correctly and efficiently. If you are a more experienced PixInsight/PCL developer, this article may also be of your interest as it provides a well-organized description of basic module development techniques and some key elements of PixInsight's architecture, difficult to find elsewhere. Finally, if you are a non-developer user, you may find this article also interesting to learn more about PixInsight, its architecture and some of its key design principles, which will provide you with a more complete perspective to understand and apply the different tools and resources available.

PixInsight Modules and PCL

PixInsight modules are shared objects —also known as shared libraries or dynamic load libraries on some platforms— that can be installed to add functionality to the PixInsight Core application, hereafter referred to as just «the core» for simplicity. You have a graphical description of PixInsight's modular architecture on Figure 1. As you can see in this figure, the core and all PixInsight modules communicate and work together following a client-server model. Communication between a module and the core is bidirectional: a module can make requests to the core and receive answers, or pass data to the core and retrieve data from the core. In all cases, valid communication must follow a set of strictly defined protocols and procedures,

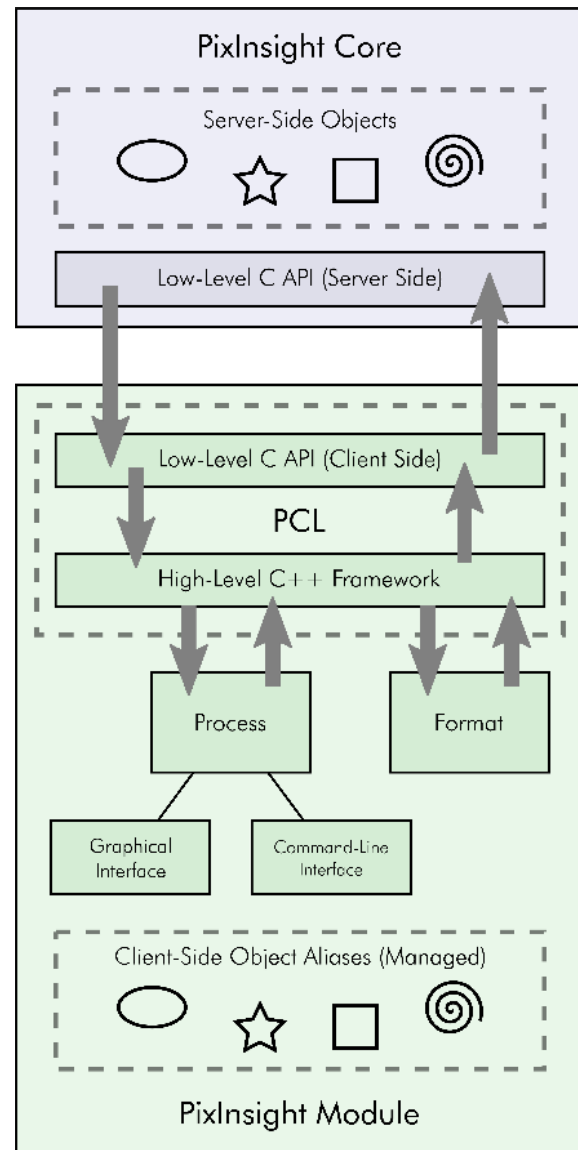


Figure 1— The modular architecture of PixInsight.

and use a set of predefined data types and data structures. These protocols, procedures, types and structures have been formally defined as a specific *application programming interface* (API). PixInsight's internal module communication API is a relatively complex set of functions and data structures defined in the C language that we collectively call *the low-level API*. As of writing this document, the low-level API has not been published; it is a closed-source, publicly undocumented part of PixInsight only known to the core development team, and there are no current plans to change its status. The client-side part of the low-level API is internal to PCL, and the server-side part is obviously internal to the core.

While modules and the core communicate using the low-level API behind the scenes, this is by no means how PixInsight modules are designed and developed. We have taken one step further with a dedicated framework written in the C++ language that provides a much higher level of abstraction. This high-level C++ framework has been implemented as the *PixInsight Class Library*, or PCL for short, and is publicly available for download at:
http://pixinsight.com/developer/pcl/download/

PCL has also a quite complete reference documentation in HTML format, which is included in all PCL distributions and is also available online at:
http://pixinsight.com/developer/pcl/doc/html/

Along with a core-module communication layer and a number of support classes (for example, platform-independent file access, container classes, thread support, etc.), PCL provides many ready-to-use implementations of image processing algorithms, including geometric transformations, intensity transformations (histograms, interpolated curves, color saturation, etc.), color management, colorimetrically defined RGB working spaces, morphological transformations, convolutions, pixel interpolations, fast Fourier transforms and wavelet transforms, among others. Most of these algorithms are provided as parallel multithreaded implementations. PCL is cross-platform: it is available on all platforms where PixInsight has been ported. Currently this includes FreeBSD, Linux, Mac OS X and Windows. Provided that reasonably good programming practices are observed, a module written around PCL is guaranteed to work on all supported platforms without changing a single line of source code.

PCL is completely free and royalty-free. It is released under a liberal software license, similar to the BSD license, that imposes no restrictions on distribution and availability of source code: PixInsight modules developed with PCL can be distributed as open-source or closed-source, freeware or commercial products. The PCL license has been included in the headers of all PCL source code files and can also be downloaded as a plain text file at:
http://pixinsight.com/developer/pcl/pcl-license-1.0.txt.

A module can implement basically two types of objects to add functionality to the PixInsight platform: *processes* and *formats*. A process can be thought of as a tool that usually —but not necessarily— provides some kind of interaction with the user. The actual functionality, roles and features of a PixInsight process, however, go far beyond what the typical tool or plug-in can do in most applications. Most PixInsight processes can modify images in some way, but they can also be *global processes* that don't work in the context of any particular image, *dynamic processes* that provide a higher level of interaction through the graphical interface, or *inspectors* and *observers* that gather information on images, on other running processes, on system objects, etc. A format implements an image format, which usually —although not necessarily— corresponds to a file format such as JPEG, TIFF, FITS, etc. There's nothing in the PixInsight/PCL architecture against a single module providing both processes and formats, although this is nonstandard practice and is not recommended for the sake of modularity, which is one of the main design principles of PixInsight. So we usually talk about *process modules* and *format modules* because their functionality is well defined and differenced.

Figure 1 also depicts the way some objects can be shared between modules and the core. Objects such as images, image views and windows, elements of the graphical user interface (windows, dialogs and controls such as buttons, check boxes, etc.), process instances (which are concrete representations of processes) and many more, can be managed, created and manipulated from a module. This object sharing mechanism plays a key role in the PixInsight platform and has been implemented through *managed aliases* in PCL. A managed alias is an abstraction of an actual object living in the core, that a module can see and manipulate as a high-level replica of the original.

This is just a general description; more complete and accurate information must be provided by means of practical examples to be useful, and this is what we'll do in successive articles on PixInsight development topics. So let's start by telling you how to set up your PCL development environment and how to write a simple module, to teach you the basics that you need to become a proficient PixInsight/PCL developer.

PCL Working Environment

The first thing you have to do is pretty obvious: download and install PCL. As we have said before, the PCL distribution is available for download from PixInsight's website:
http://pixinsight.com/developer/pcl/download/

Always keep your local PCL installation up-to-date with the latest version available. PCL is distributed as a unique archive that provides all necessary libraries and source code files for PixInsight/PCL module development on all supported platforms. Currently it is a standard gzip-compressed tar archive (tar.gz). Uncompressing and extracting tar.gz files are trivial operations on UNIX (FreeBSD and Mac OS X) and GNU/Linux platforms, but require a specific tool on Windows. For developers working on Windows platforms we recommend the excellent 7-Zip free utility, available at:
http://www.7-zip.org/.

Installing PCL is as simple as extracting the tar.gz archive on any directory where you have full read and write rights; typically on a directory under your home directory. You'll need also a working PixInsight Core application, which requires a valid software license. Both commercial and free trial licenses can be used for development.

PCL Distribution

Let's take a look at the PCL distribution's directory tree, which you can see on Figure 2. The following list describes the contents and purpose of each PCL distribution directory. We

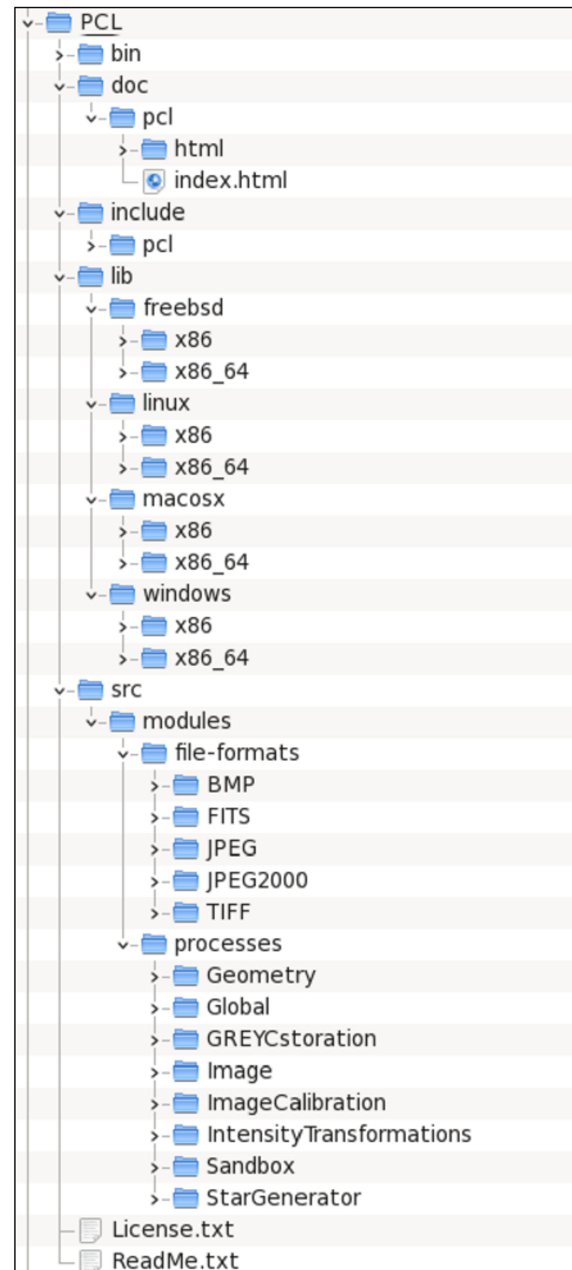


Figure 2— The PCL distribution tree.

assume that the PCL distribution package has been installed on a <PCL> directory of your local filesystem; for example:

UNIX and Linux: <PCL> = $HOME/PCL
Windows: <PCL> = C:\PCL

<PCL>/bin
Binary files. This subdirectory includes —or may include, after the corresponding compilation and building procedures— a number of utility programs, mainly for management of PCL source code and module projects. As it is distributed, this directory is initially empty.

<PCL>/doc/pcl
PCL Documentation files. Launch the file <PCL>/doc/pcl/index.html to browse the PCL Reference Documentation in HTML format. This documentation has been generated with the Doxygen tool from standard PCL header files.

<PCL>/include/pcl
Standard PCL C++ header files.

<PCL>/lib/freebsd
<PCL>/lib/linux
<PCL>/lib/macosx
<PCL>/lib/windows
PCL static libraries for all supported platforms and architectures. Currently PCL libraries are provided for the following compilers:

FreeBSD and Linux:
GCC C++ compiler version 4.4 and higher.
Mac OS X:
GCC C++ compiler version 4.2 and higher.
Windows:
Microsoft Visual C++ 2008.

<PCL>/src/modules/file-formats
A selection of PixInsight format modules, with complete source code.

<PCL>/src/modules/processes
A selection of PixInsight process modules, with complete source code. The source code is ready for compilation and forms an excellent set of development documentation and reference material.

Environment Variables

Once you have installed PCL, you must define a number of environment variables. These variables are necessary for the build system to locate the necessary library, source code and header files, as well as to generate the output modules (shared objects) in the correct place. All the required variables are described in the following list.

PCLDIR
PCL root directory. The value of this environment variable must be the full directory path where the PCL distribution has been installed on your computer. On FreeBSD, Linux and Mac OS X, this variable should normally be, assuming that you have installed PCL on a 'PCL' subdirectory of your home directory:

    $HOME/PCL

and on Windows, something like:

    C:\Users\<your-user-name>\PCL

or something perhaps more practical such as:

    C:\PCL

PCLBINDIR32
Optional on 64-bit platforms. Binary files subdirectory within the PCL directory tree, for output 32-bit executables and shared objects. On 32-bit platforms this variable is usually equal to $PCLDIR/bin. On 64-bit platforms, this variable is only necessary to carry out cross-platform 32-bit builds.

PCLBINDIR64
64-bit platforms only. Binary files subdirectory within the PCL directory tree, for output 64-bit executables and shared objects. On 64-bit platforms this variable is usually equal to $PCLDIR/bin. On 32-bit platforms this variable should not be defined, since 64-bit cross-platform builds are usually not possible on 32-bit operating systems.

PCLBINDIR
PCL binaries directory. This variable should be equal to either PCLBINDIR32 or PCLBINDIR64, respectively for 32-bit and 64-bit platforms. This variable points to the directory where output binaries must be generated for the host machine's native architecture.

PCLLIBDIR32
Optional on 64-bit platforms. 32-bit libraries subdirectory within the PCL directory tree. This variable should be equal to $PCLDIR/lib/<platform>/x86 on each platform. On 64-bit platforms, this variable is only necessary to carry out cross-platform 32-bit builds.

PCLLIBDIR64
64-bit platforms only. 64-bit libraries subdirectory within the PCL directory tree. This variable should be equal to $PCLDIR/lib/<platform>/x86_64 on each platform. On 32-bit platforms this variable should not be defined, since 64-bit cross-platform builds are usually not possible under 32-bit operating systems.

PCLLIBDIR
PCL library files directory. This variable should be equal to either PCLLIBDIR32 or PCLLIBDIR64, respectively for 32-bit and 64-bit platforms. This variable points to the directory where static PCL libraries are available for the host machine's native architecture.

PCLINCDIR
PCL include files directory. Should be equal to $PCLDIR/include.

PCLSRCDIR
PCL source files directory. Should be equal to $PCLDIR/src.

Defining these environment variables requires different steps on each supported platform. On Windows you can define them with specific options available through the system's *Control Panel* application. For example, on Windows 7 and Windows Vista the following sequence will allow you to edit, add and remove environment variables: *Start > Control Panel > System > Advanced System Settings > System Properties > Advanced > Environment Variables.*

On Mac OS X you must create a property list file with the variable definitions. The path to this file must be *~/.MacOSX/environment.plist*. You'll find more information on the following section of Apple's Mac OS X Reference Library: *http://developer.apple.com/library/mac/#documentation/MacOSX/Conceptual/BPRuntimeConfig/Articles/EnvironmentVars.html#//apple_ref/doc/uid/20002093-113982*

On FreeBSD and Linux, you simply have to add the necessary variable declarations to your shell configuration file: *.bash_profile*, *.tcsh_profile*, or

the appropriate file for your shell. Figure 3 shows an example *.bash_profile* file for a 64-bit Linux system. Note that the configuration file shown in Figure 3 is just an example; you may need different declarations adapted to your particular working environment. In the example, the user has installed the PCL distribution on the `$HOME/PCL` directory and both 32-bit and 64-bit versions of the PixInsight Core application on the `$PIDIR32` and `$PIDIR64` directories, respectively. The `PCLBINDIR32` and `PCLBINDIR64` variables have been configured to generate module binaries (.so shared object files) directly on the bin distribution directory of each installed PixInsight Core application. The rest of PCL variables have been defined to point to the corresponding subdirectories within the PCL distribution tree.

```
# Get the default aliases and functions
if [ -f ~/.bashrc ]; then
      . ~/.bashrc
fi

# PixInsight installation directories
export PIDIR32=$HOME/PixInsight/x86
export PIDIR64=$HOME/PixInsight/x86_64
export PIDIR=$PIDIR64

# PCL environment variables
export PCLDIR=$HOME/PCL
export PCLBINDIR32=$PIDIR32/bin
export PCLBINDIR64=$PIDIR64/bin
export PCLBINDIR=$PCLBINDIR64
export PCLLIBDIR32=$PCLDIR/lib/linux/x86
export PCLLIBDIR64=$PCLDIR/lib/linux/x86_64
export PCLLIBDIR=$PCLLIBDIR64
export PCLINCDIR=$PCLDIR/include
export PCLSRCDIR=$PCLDIR/src

export PATH=$PATH:$PCLBINDIR
```

Figure 3— A *.bash_profile* configuration file for a 64-bit Linux PCL development environment.

```
 1 #ifndef __SandboxModule_h
 2 #define __SandboxModule_h
 3
 4 #include <pcl/MetaModule.h>
 5
 6 namespace pcl
 7 {
 8
 9 class SandboxModule : public MetaModule
10 {
11 public:
12
13    SandboxModule();
14
15    virtual const char* Version() const;
16    virtual IsoString Name() const;
17    virtual String Description() const;
18    virtual String Company() const;
19    virtual String Author() const;
20    virtual String Copyright() const;
21    virtual String TradeMarks() const;
22    virtual String OriginalFileName() const;
23    virtual void GetReleaseDate( int& year, int& month, int& day ) const;
24 };
25
26 } // pcl
27
28 #endif   // __SandboxModule_h
```

Listing 1 — SandboxModule.h

The Sandbox Module

No article related to practical computer programming topics should be without abundant source code examples, and this one won't be an exception. The Sandbox module has been included in all PixInsight Core and PCL distributions to ease creation of new process modules, especially for newcomer developers. It is a simple module that doesn't do anything useful, except providing a skeleton that can be used as the basis for a real project. We'll continue most of this article working with Sandbox as an idoneous tool to teach you the structure and functionality of a PixInsight process module. We'll leave the particularities of format modules for a future article.

In the PCL distribution, Sandbox can be found on *src/modules/processes/Sandbox* under the PCL root installation directory. Let's take a look at Sandbox's source files on Figure 4. As you can see, there are five groups of two files each with the following suffixes: *Instance*, *Interface*, *Module*, *Parameters* and *Process*, each group consisting of a .h file (for declarations) and a .cpp file (for definitions and implementations). This is the typical distribution of source code files in a process module. There's no rule against implementing a module in a different way, but this is the customary distribution that we recommend because it clearly identifies the different functional blocks, as we'll describe in the following sections.

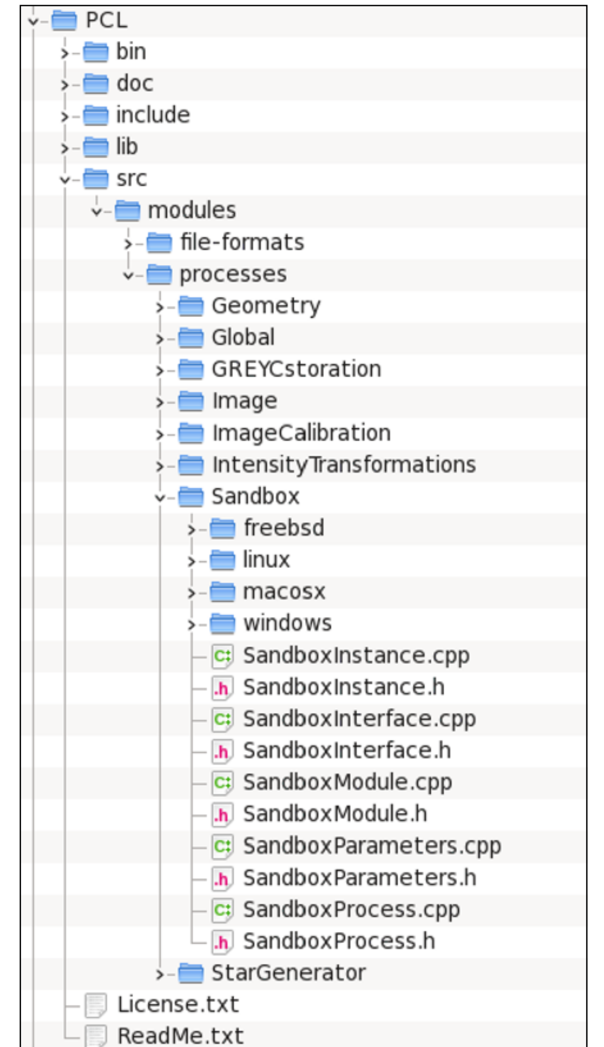


Figure 4— The Sandbox module in the PCL distribution tree.

Module Definition

A PixInsight module is a collection of components organized as a hierarchical tree structure, whose root element is a *unique* instance of a class derived from `MetaModule`. Such unique instance is mandatory in every PixInsight module, and is always defined as the `Module` global variable automatically by the PCL. The `MetaModule` class, besides acting as the root of the module's hierarchical structure, provides a formal description of the module, including some basic properties such as the module's name, version, copyright information, release date, etc. `MetaModule` and its unique instance `Module` are declared in the standard pcl/MetaModule.h header.

The SandboxModule.h file declares the `SandboxModule` class, which is a derived class of `MetaModule`. You can see the source code of this

```
 1 #define MODULE_VERSION_MAJOR     01
 2 #define MODULE_VERSION_MINOR     00
 3 #define MODULE_VERSION_REVISION  00
 4 #define MODULE_VERSION_BUILD     0049
 5 #define MODULE_VERSION_LANGUAGE  eng
 6
 7 #define MODULE_RELEASE_YEAR      2010
 8 #define MODULE_RELEASE_MONTH     12
 9 #define MODULE_RELEASE_DAY       14
10
11 #include "SandboxModule.h"
12 #include "SandboxProcess.h"
13 #include "SandboxInterface.h"
14
15 namespace pcl
16 {
17
18 SandboxModule::SandboxModule() : MetaModule()
19 {
20 }
21
22 const char* SandboxModule::Version() const
23 {
24    return PCL_MODULE_VERSION( MODULE_VERSION_MAJOR,
25                               MODULE_VERSION_MINOR,
26                               MODULE_VERSION_REVISION,
27                               MODULE_VERSION_BUILD,
28                               MODULE_VERSION_LANGUAGE );
29 }
30
31 IsoString SandboxModule::Name() const
32 {
33    return "Sandbox";
34 }
35
36 String SandboxModule::Description() const
37 {
38    return "PixInsight Sandbox Process Module"; // Replace with your own description
39 }
40
41 String SandboxModule::Company() const
42 {
43    return "Your company here";
44 }
45
46 String SandboxModule::Author() const
47 {
48    return "Your name here";
49 }
50
51 String SandboxModule::Copyright() const
52 {
53    return "Copyright (c) you, the year you wrote this";
54 }
55
56 String SandboxModule::TradeMarks() const
57 {
58    return "Your trade marks here";
59 }
60
61 String SandboxModule::OriginalFileName() const
62 {
63 #ifdef __PCL_FREEBSD
64    return "Sandbox-pxm.so";
```

Listing 2 — SandboxModule.cpp (1/2)

```
 65 #endif
 66 #ifdef __PCL_LINUX
 67    return "Sandbox-pxm.so";
 68 #endif
 69 #ifdef __PCL_MACOSX
 70    return "Sandbox-pxm.dylib";
 71 #endif
 72 #ifdef __PCL_WINDOWS
 73    return "Sandbox-pxm.dll";
 74 #endif
 75 }
 76
 77 void SandboxModule::GetReleaseDate( int& year, int& month, int& day ) const
 78 {
 79    year  = MODULE_RELEASE_YEAR;
 80    month = MODULE_RELEASE_MONTH;
 81    day   = MODULE_RELEASE_DAY;
 82 }
 83
 84 } // pcl
 85
 86 /*
 87  * Module installation routine
 88  */
 89 PCL_MODULE_EXPORT int InstallPixInsightModule( int mode )
 90 {
 91    new pcl::SandboxModule;
 92
 93    if ( mode == pcl::InstallMode::FullInstall )
 94    {
 95       new pcl::SandboxProcess;
 96       new pcl::SandboxInterface;
 97    }
 98
 99    return 0;
100 }
```

Listing 2 — SandboxModule.cpp (2/2)

file in Listing 1, and the corresponding definitions in SandboxModule.cpp, which we have included as Listing 2. All member functions of `SandboxModule` are pretty straightforward; most of them are optional since `MetaModule` provides default implementations, and all of them are virtual member functions. For the sake of coherence with standard modules, however, all of these member functions should always be reimplemented —failure to do so is considered bad programming practice. You can consider `SandboxModule` as a reference for defining and implementing a PCL module class.

The Module Installation Routine

An item that deserves a detailed explanation is the *module installation routine*, namely `InstallPixInsightModule()`. You can see this function implemented in SandboxModule.cpp, on Listing 2, line 89. This function is mandatory in every PixInsight module and has the following prototype:

```
PCL_MODULE_EXPORT int InstallPixInsightModule( int mode );
```

The `PCL_MODULE_EXPORT` macro is defined in the pcl/Defs.h standard header, which is automatically included by all PCL headers, and *must* be specified as a modifier to this function, just as shown in the prototype above. This macro provides the necessary declarations and compiler directives to export a function in a platform and compiler-independent way.

`InstallPixInsightModule()` will be called by the core when the module is being installed, either as a result of a specific user request, or as part of the module installation procedure that takes place upon application startup. The `mode` argument can be one of the following:

`InstallMode::FullInstall`
The module is being installed normally. In this mode all required objects for normal module operation should be initialized. In Listing 2, lines 93-97, note that the `SandboxProcess` and `SandboxInterface` instances are being initialized when this installation mode is passed to the module installation routine.

`InstallMode::QueryModuleInfo`
The module is being installed exclusively to retrieve module metadata. Besides an object inheriting from `MetaModule` and providing reimplementations of virtual member functions such as `Version()`, `Name()`, `ReleaseDate()` and so on, no additional objects such as processes, interfaces or formats need to be initialized in this installation mode.

`InstallMode::VerifyModule`
The module is being installed exclusively for validation and integrity verification. The same remarks and conditions that we have specified for `InstallMode::QueryModuleInfo` also apply to this installation mode.

The `InstallPixInsightModule` function must be defined in the global namespace —a common error is defining it in the `pcl` namespace; in such case it won't be exported with the correct signature and the core won't find it. This function should return zero to signal successful module installation; any other value will be interpreted as denoting installation failure. Note that this function, as happens with all PCL callbacks, can freely throw exceptions. All exceptions thrown will be catched by internal PCL routines and will lead to a nonzero return value (and hence the module won't install).

Module Version and Release Date

In SandboxModule.cpp, on Listing 2 line 22, you can see another important member function that we'll describe a little further, namely `SandboxModule::Version()`, which is a reimplementation of `MetaModule::Version()`. This virtual member function returns the address of a static character string that must specify the module version in a specific format that is thoroughly described with comments in the standard pcl/Module.h header. If the module version is not defined with the correct format, the core won't recognize the module shared object as a valid PixInsight module. This is part of a security mechanism to prevent problems caused by loading wrong shared objects from the core. To facilitate defining valid module version strings, PCL provides the `PCL_MODULE_VERSION` macro, defined in pcl/Module.h. The use of `MODULE_VERSION_` macro definitions (Listing 2, lines 1 to 5) isn't standard or strictly required, but it is customary for PCL development, and these macros are required by some PCL code maintenance utilities. The same is true for `MODULE_RELEASE_` macros to define module release dates. For compatibility with PCL code maintenance utilities, always use these macros in your modules, just as you have them included in SandboxModule.cpp (Listing 2).

Process Definition

The files SandboxProcess.h (on Listing 3) and SandboxProcess.cpp (on Listing 4) declare and define, respectively, the `SandboxProcess` class. This is a derived class of `MetaProcess`, which is the PCL class that must be used to implement the basic properties and functionality of all PixInsight processes. As happens with `MetaModule`, a number of virtual member functions of `MetaProcess` must be reimplemented by derived classes, while other non-critical members provide default implementations. Let's review some of these members.

Process Identifier

One of those critical member functions is `MetaProcess::Id()` (on Listing 3, line 15, and Listing 4, line 29). This member function returns the identifier of the process as an `IsoString` object. Every process must have a *unique* identifier, that is, two or more installed processes cannot have the same identifier. In PixInsight, process identifiers —*all* identifiers, actually— are case-sensitive (*fooProcess* is different from *FooProcess*), must be valid C identifiers (*a123* and *x_yz* are valid but *1a23* and '*x yz*' are not), and it is customary to form all process identifiers as concatenations of capitalized words, such as `UnsharpMask` or `HDRWaveletTransform` for example. In our example we are defining a process whose identifier is 'Sandbox', as you can see on line 31 of Listing 4.

```
 1 #ifndef __SandboxProcess_h
 2 #define __SandboxProcess_h
 3
 4 #include <pcl/MetaProcess.h>
 5
 6 namespace pcl
 7 {
 8
 9 class SandboxProcess : public MetaProcess
10 {
11 public:
12
13    SandboxProcess();
14
15    virtual IsoString Id() const;
16    virtual IsoString Category() const;
17
18    virtual String Description() const;
19
20    virtual const char** IconImageXPM() const;
21
22    virtual ProcessInterface* DefaultInterface() const;
23
24    virtual ProcessImplementation* Create() const;
25    virtual ProcessImplementation* Clone( const ProcessImplementation& ) const;
26
27    virtual bool CanProcessCommandLines() const;
28    virtual int ProcessCommandLine( const StringList& ) const;
29 };
30
31 PCL_BEGIN_LOCAL
32 extern SandboxProcess* TheSandboxProcess;
33 PCL_END_LOCAL
34
35 } // pcl
36
37 #endif   // __SandboxProcess_h
```

Listing 3 — SandboxProcess.h

Process Category

`MetaProcess::Category()` is another virtual member function that should be reimplemented by all process classes. The default implementation in `MetaProcess` returns an empty string, which is the same as not specifying any category. The set of installed processes forms a categorized tree in PixInsight. The core gathers all processes that don't specify a category in a default *`Etc`* group. Unless a process has a strong reason to exclude itself from the category tree, it should return a nonempty string from this member function. If the returned category already exists —because another process already specified it when its module was installed—, then the process will be added to the existing category. If the name of a nonexisting category is returned, a new category with that name is created and the process becomes its only member. Note that it is legal to return two or more different categories, separated by commas, from this function. For example, the standard `StarAlignment` process returns the string `"ImageRegistration, Preprocessing"` to inform that it belongs to those two categories.

```
1 #include "SandboxProcess.h"
2 #include "SandboxParameters.h"
3 #include "SandboxInstance.h"
4 #include "SandboxInterface.h"
5
6 #include <pcl/Console.h>
7 #include <pcl/Arguments.h>
8 #include <pcl/View.h>
9 #include <pcl/Exception.h>
10
11 namespace pcl
12 {
13
14 // #include "SandboxIcon.xpm" -- TODO: Define a process icon
15
16 SandboxProcess* TheSandboxProcess = 0;
17
18 SandboxProcess::SandboxProcess() : MetaProcess()
19 {
20    TheSandboxProcess = this;
21
22    new SandboxParameterOne( this ); // Instantiate process parameters
23    new SandboxParameterTwo( this );
24    new SandboxParameterThree( this );
25    new SandboxParameterFour( this );
26    new SandboxParameterFive( this );
27 }
28
29 IsoString SandboxProcess::Id() const
30 {
31    return "Sandbox";
32 }
33
34 IsoString SandboxProcess::Category() const
35 {
36    return IsoString(); // No category -- TODO: Define a category
37 }
38
39 String SandboxProcess::Description() const
40 {
41    return
42    "<html>"
43    "<p>Sandbox is just a starting point for development of PixInsight modules. "
44    "It is an empty module that does nothing but to provide the basic structure "
45    "of a module with a process and a process interface.</p>"
46    "</html>";
47 }
48
49 const char** SandboxProcess::IconImageXPM() const
50 {
51    return 0; // SandboxIcon_XPM; -- TODO: Define a process icon
52 }
53
54 ProcessInterface* SandboxProcess::DefaultInterface() const
55 {
56    return TheSandboxInterface;
57 }
58
59 ProcessImplementation* SandboxProcess::Create() const
60 {
61    return new SandboxInstance( this );
62 }
```

Listing 4 — SandboxProcess.cpp (1/3)

```
63
64 ProcessImplementation* SandboxProcess::Clone( const ProcessImplementation& p ) const
65 {
66    const SandboxInstance* instPtr = dynamic_cast<const SandboxInstance*>( &p );
67    return (instPtr != 0) ? new SandboxInstance( *instPtr ) : 0;
68 }
69
70 bool SandboxProcess::CanProcessCommandLines() const
71 {
72    return true;
73 }
74
75 static void ShowHelp()
76 {
77    Console().Write(
78 "<raw>"
79 "Usage: Sandbox [<arg_list>] [<view_list>]"
80 "\n"
81 "\n-p1=<n>"
82 "\n"
83 "\n      <n> is a real number that will be assigned to Sandbox's first parameter."
84 "\n"
85 "\n-p3[+|-]"
86 "\n"
87 "\n      Turns on|off the third parameter of the Sandbox process."
88 "\n"
89 "\n-p5=<s>"
90 "\n"
91 "\n      <s> is a string that will be assigned to Sandbox's fifth parameter."
92 "\n"
93 "\n--interface"
94 "\n"
95 "\n      Launches the interface of this process."
96 "\n"
97 "\n--help"
98 "\n"
99 "\n      Displays this help and exits."
100 "</raw>" );
101 }
102
103 int SandboxProcess::ProcessCommandLine( const StringList& argv ) const
104 {
105    ArgumentList args = ExtractArguments(
106             argv, ArgumentItemMode::AsViews, ArgumentOption::AllowWildcards );
107
108    SandboxInstance instance( this );
109
110    bool launchInterface = false;
111    int count = 0;
112
113    for ( ArgumentList::const_iterator i = args.Begin(); i != args.End(); ++i )
114    {
115       const Argument& arg = *i;
116
117       if ( arg.IsNumeric() )
118       {
119          if ( arg.Id() == "p1" )
120             instance.parameterOne = arg.NumericValue();
121          else
122             throw Error( "Unknown numeric argument: " + arg.Token() );
123       }
124       else if ( arg.IsString() )
```

Listing 4 — SandboxProcess.cpp (2/3)

```
125          {
126             if ( arg.Id() == "p5" )
127                instance.parameterFive = arg.StringValue();
128             else
129                throw Error( "Unknown string argument: " + arg.Token() );
130          }
131          else if ( arg.IsSwitch() )
132          {
133             if ( arg.Id() == "p3" )
134                instance.parameterThree = arg.SwitchState();
135             else
136                throw Error( "Unknown switch argument: " + arg.Token() );
137          }
138          else if ( arg.IsLiteral() )
139          {
140             if ( arg.Id() == "p3" ) // -p3 specified without +|-
141                instance.parameterThree = true;
142             else if ( arg.Id() == "-interface" )
143                launchInterface = true;
144             else if ( arg.Id() == "-help" )
145             {
146                ShowHelp();
147                return 0;
148             }
149             else
150                throw Error( "Unknown argument: " + arg.Token() );
151          }
152          else if ( arg.IsItemList() )
153          {
154             ++count;
155
156             if ( arg.Items().IsEmpty() )
157             {
158                Console().WriteLn( "No view(s) found: " + arg.Token() );
159                throw;
160             }
161
162             for ( StringList::const_iterator j = arg.Items().Begin();
163                   j != arg.Items().End(); ++j )
164             {
165                View v = View::ViewById( *j );
166                if ( v.IsNull() )
167                   throw Error( "No such view: " + *j );
168                instance.LaunchOn( v );
169             }
170          }
171       }
172
173       if ( launchInterface )
174          instance.LaunchInterface();
175       else if ( count == 0 )
176       {
177          if ( ImageWindow::ActiveWindow().IsNull() )
178             throw Error( "There is no active image window." );
179          instance.LaunchOnCurrentView();
180       }
181
182       return 0;
183 }
184
185 } // pcl
```

Listing 4 — SandboxProcess.cpp (3/3)

Process Instantiation

Unless you are writing an inspector or an observer tool, your process must be able to generate *process instances*. A process class, which we define as a derived class of `MetaProcess`, is a description of a process in abstract terms. A process instance is a concrete realization of a process class. For example, the standard `Statistics` process is an observer in PixInsight. `Statistics` cannot generate instances because they wouldn't make any sense; after all, `Statistics` does nothing at all that could belong to a processing history because it *cannot be executed*. The standard `CurvesTransformation` process, on the other hand, is an *instance generator*. An instance of `CurvesTransformation` can be executed on an image to apply a pixel intensity transformation.

An instantiable process must return newly created instances from a reimplementation of `MetaProcess::Create()`. Similarly, a new instance must also be returned from a reimplementation of `MetaProcess::Clone()`, constructed as a duplicate (a clone) of an existing instance. Note that both are pure virtual member functions, so they must be reimplemented as non-pure by all derived classes of `MetaProcess`. A process instance is an instance of a derived class of the `ProcessImplementation` PCL class. We'll study this class later, when we review the `SandboxInstance` class.

The `Sandbox` process is instantiable, mainly because in this way we are helping you in starting your own processes, which almost always will be instantiable processes —observers are nice, but most of the time you'll want to deal with executable image processing tasks in PixInsight. You can see how `SandboxProcess` declares its instance generation member functions in Listing 3, line numbers 24 and 25, respectively. The definitions of these functions can be seen on Listing 4, lines 59 and 64, respectively. What would these functions look like for an observer tool? Easy: both functions would just return zero. In this way the core knows that no instance can be generated, and the observer behaves as such.

In `SandboxProcess::Clone()`, note that we are using `dynamic_cast` to ensure that we are going to clone an instance of our process, that is, `SandboxInstance`. When the core invokes the `Clone` member function, there's no guarantee that the source instance (the `p` argument in Listing 4 line 64) is of the same class as the process being invoked —note that `p` is a reference to an instance of the generic `ProcessImplementation` class, so it can be a reference to *any* process instance—; `dynamic_cast` allows removing that uncertainty.

Command-Line Execution Routine

PixInsight provides three different user interfaces in the PixInsight Core application: a graphical user interface, a scripting interface —the JavaScript runtime—, and a command-line interface available from the Processing Console window. Along with a large set of internal commands, PixInsight's command-line interface allows running any installed process. By default, any installed process can be launched by just entering its identifier as a command; you don't need to do anything special to get this functionality. However, a process can implement its own command-line execution routine to customize its behavior on the command-line and to provide its own command-line arguments. That's what we have done with `SandboxProcess`, again to guide and help you. Implementing a custom command-line routine is always a good idea for most processes, as this improves their behavior and extends their scripting functionality.

To implement a custom command-line execution routine, a process must inform the core that it is able to manage command-line invocations; otherwise the custom routine will never be called. This must be done by returning `true` from a reimplementation of `MetaProcess::CanProcessCommandLines()`. You can see how `SandboxProcess` has reimplemented this function in Listing 4, line 70.

A process' command-line execution routine in PCL looks and behaves much like the `main` function of a command-line application. It receives a list of command-line arguments in a way very similar to the standard `argv` argument of `main`, but instead of an array of `char` pointers, it is a constant reference to `StringList`. A `StringList` object is a dynamic array of `String` objects (UTF-16 Unicode character strings), implemented as the `Array<String>` template instantiation in PCL. As happens with all PCL containers, `StringList` elements can be traversed with mutable iterators and constant iterators in the usual way, and also using integer array subscripts. As is customary in the UNIX

world, a PCL command-line routine *must* accept the standard `--help` argument to provide some help about its usage, valid syntax and supported arguments. The `ShowHelp()` routine (Listing 4 line 75) provides a good starting point to write a process command-line help. Naturally, all console output must be done by calling `Console::Write()` and similar functions. You may want to read the documentation for the `pcl::Console` class at this point. The `ShowHelp()` routine in SandboxProcess.cpp can be considered as a prototype to be followed in all implementations.

On Listing 4 line 103 you can see how a command-line routine looks like for a typical process. The `ExtractArguments` function (line 105) performs the argument parsing task efficiently, which saves you a lot of work. There's much more to say about command-line routines. To learn more you should consult the *Argument parsing routines and utilities* section of the PCL Reference Documentation, along with the documentation for the `pcl::Argument` class.

Default Interface

Along with command-line functionality, processes usually have one or more associated graphical interfaces. The term *default interface* refers to the process interface that the core will use to *launch* a process. For example, A process is launched when the user makes double click on a process icon, or when the user activates a process item on the Process Explorer window.

To specify a default interface, a process must reimplement the `MetaProcess::DefaultInterface` virtual member function to return the address of an object inheriting from the `ProcessInterface` class. A derived class of `ProcessInterface` represents a specialized top-level window that provides a graphical user interface for a particular process. A `ProcessInterface` descendant class typically implements a set of controls that can be used to edit process parameters, such as edit fields, sliders, check boxes, etc. The subject of process interfaces is complex and cannot be covered in this introductory work.

Process Icon

Every process has an associated icon in PixInsight. Icons are important graphical elements because they allow identifying the different processes and their instances in a variety of contexts within the core application. We expect that PixInsight/PCL developers will invest some time and care in designing and creating nice icons for their new tools. A good icon must be both graphically appealing and informative about the process task. A process icon can be specified in several ways. One obvious way is by reimplementing the `MetaProcess::IconImageFile` virtual member function. This function returns the absolute file path of an image file for the icon. Supported formats include PNG, XPM, JPEG, BMP and TIFF at least. However, using this function is discouraged, mainly because it generates an unnecessary dependency on an external resource, and also because providing a file path can be quite error-prone. The preferred way to specify a process icon is with a reimplementation of `MetaProcess::IconImageXPM()`. This member function returns the address of an image stored in the standard X Pixmap format (XPM) as a static array of `char*`. The XPM format is very nice for us developers because it allows us to encode image resources as static data blocks integrated with C/C++ source code:
http://en.wikipedia.org/wiki/X_PixMap
You can use a variety of image editing applications, including PixInsight, to create XPM files from PNG or TIFF images.

On Listing 3 line 20 you can see how `SandboxProcess` declares its `IconImageXPM()` reimplementation. On Listing 4 line 14, note the `#include` directive that should be uncommented when the SandboxIcon.xpm file is available. Similarly, `SandboxProcess::IconImageXPM()` on line 51 should return the address of the XPM structure (`SandboxIcon_XPM` in the code), when available.

Process icons must be 24x24 pixels, 32-bit ARGB images. If other dimensions are used, the core automatically resizes the icon, which may lead to suboptimal results. If a process doesn't specify an icon —as usually happens during initial development stages— the core automatically assigns a default icon (a *gear* icon in current core versions).

Instance Definition

The files SandboxInstance.h and SandboxInstance.cpp, in Listing 5 and Listing 6, respectively, implement the instance class of our process: `SandboxInstance`. This is a derived class of `ProcessImplementation`, which as we have seen earlier, is the class that represents a process instance in PCL. Recall that process instances are responsible for actual processing in PixInsight: a process is just an abstract definition, while a process instance is an actual object that can be executed to work directly with image data, among many other things.

The first thing we have to focus our attention on is `SandboxInstance`'s constructors. This class declares two constructors on lines 14 and 15 of SandboxInstance.h (Listing 5), respectively. Note that none of these constructors will be invoked directly by the core; instances will always be created by the process class (`SandboxProcess` in this case) in response to direct core requests, and those requests will always be translated into calls to reimplementations of `MetaProcess::Create()` and `MetaProcess::Clone()` —at this point you may want to recall these reimplementations from Listing 4, lines 59 and 64.

Despite the fact that it isn't strictly a default constructor per C++ syntax rules, the first constructor (Listing 5, line 14) is a *default instance*

```
1 #ifndef __SandboxInstance_h
2 #define __SandboxInstance_h
3
4 #include <pcl/ProcessImplementation.h>
5 #include <pcl/MetaParameter.h>
6
7 namespace pcl
8 {
9
10 class SandboxInstance : public ProcessImplementation
11 {
12 public:
13
14    SandboxInstance( const MetaProcess* );
15    SandboxInstance( const SandboxInstance& );
16
17    virtual void Assign( const ProcessImplementation& );
18
19    virtual bool CanExecuteOn( const View&, pcl::String& whyNot ) const;
20    virtual bool ExecuteOn( View& );
21
22    virtual void* LockParameter( const MetaParameter*, size_type tableRow );
23    virtual bool AllocateParameter( size_type sizeOrLength,
24                                    const MetaParameter* p, size_type tableRow );
25    virtual size_type ParameterLength( const MetaParameter* p, size_type tableRow ) const;
26
27 private:
28
29    float     parameterOne;   // Real parameters can be either float or double
30    int32     parameterTwo;   // Use intxx or uintxx - NEVER use int, long, etc
31    pcl_bool parameterThree; // pcl_bool for Boolean parameter - NEVER use bool
32    pcl_enum parameterFour;  // pcl_enum for enumerated parameters - NEVER use enum, int, etc
33    String    parameterFive;  // String parameters are UTF-16 strings
34
35    friend class SandboxEngine;
36    friend class SandboxProcess;
37    friend class SandboxInterface;
38 };
39
40 } // pcl
41
42 #endif   // __SandboxInstance_h
```

Listing 5 — SandboxInstance.h

constructor in PCL. It takes a single argument of type `MetaProcess*`. Note that this is the constructor called by the process class from its `MetaProcess::Create()` reimplementation. This constructor should provide a default initialization for a newly created instance; you have a typical example in Listing 6, line 11.

The second constructor is the equivalent to a C++ copy constructor for a PCL process instance: we call it a *clone instance constructor*. This constructor will be called by the process class from its reimplementation of `MetaProcess::Clone()`. Note that this constructor takes a single argument of type `ProcessImplementation*`. This constructor should initialize a newly created instance as a duplicate of an existing instance of the same class. Note that the implementation of this function in `SandboxInstance` (Listing 6 line 21) simply calls the `Assign` member function, which performs all the required data member assignments. This behavior is typical and highly customary in most PCL modules.

Process Parameters

We have reached a point where we must widen our field of view to understand how a process is formally connected to the core and to the rest of the PixInsight platform. This includes standard object-oriented functionality in PixInsight, such as automatic process scripting and serialization, abstract interprocess communication and encapsulation of process instances, which fall outside the scope of this introductory article.

From an object-oriented point of view, a process instance is characterized by three main elements: (1) a process class, (2) instance methods, and (3) instance properties. We already know how to define a process class in PCL by subclassing `MetaProcess`. Instance methods are relatively straightforward: they are just virtual member functions of `ProcessImplementation`, which must be reimplemented by a derived class; we'll see some of those relevant functions later. Let's concentrate on properties now.

At the PCL level, instance properties are just data members (of a derived class of `ProcessImplementation`) known to the core. The core knows the data type and the length and/or the size in bytes occupied by each instance property, as appropriate, and also knows how to read and write them through specialized methods. This allows for very powerful features such as automatic scripting and serialization of process instances. For example, as soon as you install a module that defines a new process, it can be instantiated from JavaScript code automatically (and in the future, also from other scripting languages, such as Python). Neither the user nor the developer need to do anything special to support this feature. Similarly, all process instances can be serialized in XPSM format (XML Process Set Module format) or as binary PSM files in a completely automatic and transparent way; again, besides designing and implementing your module correctly, you don't need to do anything special for this to be possible.

In all of these advanced PixInsight features, process parameters play a key role. A process parameter is a high-level object that provides a complete description of an instance property in abstract terms. Process parameters are implemented as derived classes of `MetaParameter`. More precisely, `MetaParameter` is the root of a subtree of PCL classes that implement process parameters of all supported data types. All of these classes are declared in the standard pcl/MetaParameter.h header.

Metaparameter Classes

As happens with all objects in PixInsight, a process parameter has a unique identifier, similar to module and process identifiers. The scope of a parameter is its parent process, so a parameter identifier must be unique within its parent process, but other parameters may have the same identifier if they are children of other processes. Reimplementations of the `MetaParameter::Id` pure virtual function return parameter identifiers as `IsoString` objects. There are other member functions of MetaParameter that you should know, but this is about all you need to start working. As always, we recommend you study the corresponding sections of PCL's reference documentation.

As we have said, `MetaParameter` is the root of a subtree of descendant classes that represent all process parameters supported by PixInsight. Let's review all of these classes briefly.

MetaBoolean

Represents a Boolean process parameter. The core expects to read and write to an `int32` variable whose value will be 0 for false and 1 for true. Don't use the `bool` C++ type to implement Boolean process parameters. Always use the `pcl_bool` first-class type, defined in pcl/MetaParameter.h.

```
 1 #include "SandboxInstance.h"
 2 #include "SandboxParameters.h"
 3
 4 #include <pcl/View.h>
 5 #include <pcl/StdStatus.h>
 6 #include <pcl/Console.h>
 7
 8 namespace pcl
 9 {
10
11 SandboxInstance::SandboxInstance( const MetaProcess* m ) :
12 ProcessImplementation( m ),
13 parameterOne( TheSandboxParameterOneParameter->DefaultValue() ),
14 parameterTwo( int32( TheSandboxParameterTwoParameter->DefaultValue() ) ),
15 parameterThree( TheSandboxParameterThreeParameter->DefaultValue() ),
16 parameterFour( SandboxParameterFour::Default ),
17 parameterFive( TheSandboxParameterFiveParameter->DefaultValue() )
18 {
19 }
20
21 SandboxInstance::SandboxInstance( const SandboxInstance& x ) :
22 ProcessImplementation( x )
23 {
24    Assign( x );
25 }
26
27 void SandboxInstance::Assign( const ProcessImplementation& p )
28 {
29    const SandboxInstance* x = dynamic_cast<const SandboxInstance*>( &p );
30    if ( x != 0 )
31    {
32       parameterOne = x->parameterOne;
33       parameterTwo = x->parameterTwo;
34       parameterThree = x->parameterThree;
35       parameterFour = x->parameterFour;
36       parameterFive = x->parameterFive;
37    }
38 }
39
40 bool SandboxInstance::CanExecuteOn( const View& view, pcl::String& whyNot ) const
41 {
42    if ( view.Image().IsComplexSample() )
43    {
44       whyNot = "Sandbox cannot be executed on complex images.";
45       return false;
46    }
47
48    whyNot.Clear();
49    return true;
50 }
51
52 class SandboxEngine
53 {
54 public:
55
56    template <class P>
57    static void Apply( Generic2DImage<P>& img, const SandboxInstance& instance )
58    {
59       /*
60        * Your magic comes here...
61        */
62       Console().WriteLn( "<end><cbr>Ah, did I mention that I do just nothing at all? :D" );
63    }
64 };
```

Listing 6 — SandboxInstance.cpp (1/3)

```
65
66 bool SandboxInstance::ExecuteOn( View& view )
67 {
68    try
69    {
70       view.Lock();
71
72       ImageVariant v;
73       v = view.Image();
74
75       StandardStatus status;
76       v.AnyImage()->SetStatusCallback( &status );
77
78       Console().EnableAbort();
79
80       if ( !v.IsComplexSample() )
81          if ( v.IsFloatSample() )
82             switch ( v.BitsPerSample() )
83             {
84             case 32: SandboxEngine::Apply(
85                         *static_cast<pcl::Image*>( v.AnyImage() ), *this ); break;
86             case 64: SandboxEngine::Apply(
87                         *static_cast<pcl::DImage*>( v.AnyImage() ), *this ); break;
88             }
89          else
90             switch ( v.BitsPerSample() )
91             {
92             case  8: SandboxEngine::Apply(
93                         *static_cast<pcl::UInt8Image*>( v.AnyImage() ), *this ); break;
94             case 16: SandboxEngine::Apply(
95                         *static_cast<pcl::UInt16Image*>( v.AnyImage() ), *this ); break;
96             case 32: SandboxEngine::Apply(
97                         *static_cast<pcl::UInt32Image*>( v.AnyImage() ), *this ); break;
98             }
99
100      view.Unlock();
101
102      return true;
103   }
104
105   catch ( ... )
106   {
107      view.Unlock();
108      throw;
109   }
110 }
111
112 void* SandboxInstance::LockParameter( const MetaParameter* p, size_type /*tableRow*/ )
113 {
114    if ( p == TheSandboxParameterOneParameter )
115       return &parameterOne;
116    if ( p == TheSandboxParameterTwoParameter )
117       return &parameterTwo;
118    if ( p == TheSandboxParameterThreeParameter )
119       return &parameterThree;
120    if ( p == TheSandboxParameterFourParameter )
121       return &parameterFour;
122    if ( p == TheSandboxParameterFiveParameter )
123       return parameterFive.c_str();
124    return 0;
125 }
126
127 bool SandboxInstance::AllocateParameter( size_type sizeOrLength,
128                                          const MetaParameter* p, size_type tableRow )
```

Listing 6 — SandboxInstance.cpp (2/3)

```
129 {
130    if ( p == TheSandboxParameterFiveParameter )
131    {
132       parameterFive.Clear();
133       if ( sizeOrLength > 0 )
134          parameterFive.Reserve( sizeOrLength );
135    }
136    else
137       return false;
138    return true;
139 }
140
141 size_type SandboxInstance::ParameterLength( const MetaParameter* p,
142                                             size_type tableRow ) const
143 {
144    if ( p == TheSandboxParameterFiveParameter )
145       return parameterFive.Length();
146    return 0;
147 }
148
149 } // pcl
```

Listing 6 — SandboxInstance.cpp (3/3)

`MetaEnumeration`
Represents an enumerated process parameter. An enumerated process parameter defines a finite set of *unique-identifier/value* associations. The implementation of an enumerated parameter must be a 32-bit signed integer variable (`int32`). The best and safest way to implement enumerated parameters is by using the `pcl_enum` type, defined in pcl/MetaParameter.h.

`MetaEnumeration` declares the `ElementId` and `ElementValue` pure virtual member functions, which return the unique enumeration identifier and value, respectively, for a given enumeration element.

`MetaNumeric`
This class is the root of a subtree of classes representing all numerical parameter types supported. `MetaNumeric` has two direct descendants: `MetaInteger` and `MetaReal`, respectively to represent integer and real (floating point) process parameters. In turn, `MetaInteger` roots the `MetaSignedInteger` and `MetaUnsignedInteger` subtrees. In summary, the following leaf classes —that is, classes directly usable to derive actual process parameters— are available: `MetaDouble`, `MetaFloat`, `MetaInt8`, `MetaInt16`, `MetaInt32`, `MetaInt64`, `MetaUInt8`, `MetaUInt16`, `MetaUInt32`, and `MetaUInt64`.

`MetaVariableLengthParameter`
This is the root class of a subtree representing process parameters that implement variable-length —that is, dynamic— collections of elements. There are two read-only properties to control the valid range of lengths for one of these parameters, accessible through the `MinLength` and `MaxLength` virtual functions. By default, both of these functions return zero, meaning that there's no specific length limit. Variable-length parameters can be character strings, tables and blocks. Let's describe them briefly in the following sections.

`MetaString`
Represents an UTF-16 string —or more formally, a Unicode string encoded as UTF-16. The core expects to find a contiguous list of valid Unicode 16-bit code points as the implementation of a string process parameter. Surrogate pairs (in the rare event that a 32-bit code point is necessary) are fully supported. Two consecutive zero terminating bytes are advisable but not necessary; the core always asks a process instance to provide the current length (in characters) of a string parameter.

`MetaBlock`
Represents a block process parameter. A block is a generic container that can be used to store virtually *anything* as the value of a process param-

```
 1 #ifndef __SandboxParameters_h
 2 #define __SandboxParameters_h
 3
 4 #include <pcl/MetaParameter.h>
 5
 6 namespace pcl
 7 {
 8
 9 PCL_BEGIN_LOCAL
10
11 class SandboxParameterOne : public MetaFloat
12 {
13 public:
14
15    SandboxParameterOne( MetaProcess* );
16
17    virtual IsoString Id() const;
18
19    virtual int Precision() const;
20
21    virtual double DefaultValue() const;
22    virtual double MinimumValue() const;
23    virtual double MaximumValue() const;
24 };
25
26 extern SandboxParameterOne* TheSandboxParameterOneParameter;
27
28 class SandboxParameterTwo : public MetaInt32
29 {
30 public:
31
32    SandboxParameterTwo( MetaProcess* );
33
34    virtual IsoString Id() const;
35
36    virtual double DefaultValue() const;
37    virtual double MinimumValue() const;
38    virtual double MaximumValue() const;
39 };
40
41 extern SandboxParameterTwo* TheSandboxParameterTwoParameter;
42
43 class SandboxParameterThree : public MetaBoolean
44 {
45 public:
46
47    SandboxParameterThree( MetaProcess* );
48
49    virtual IsoString Id() const;
50
51    virtual bool DefaultValue() const;
52 };
53
54 extern SandboxParameterThree* TheSandboxParameterThreeParameter;
55
56 class SandboxParameterFour : public MetaEnumeration
57 {
58 public:
59
60    enum { FirstItem,
61           SecondItem,
62           ThirdItem,
```

Listing 7 — SandboxParameters.h (1/2)

```
63           NumberOfItems,
64           Default = FirstItem };
65
66    SandboxParameterFour( MetaProcess* );
67
68    virtual IsoString Id() const;
69
70    virtual size_type NumberOfElements() const;
71    virtual IsoString ElementId( size_type ) const;
72    virtual int ElementValue( size_type ) const;
73    virtual size_type DefaultValueIndex() const;
74 };
75
76 extern SandboxParameterFour* TheSandboxParameterFourParameter;
77
78 class SandboxParameterFive : public MetaString
79 {
80 public:
81
82    SandboxParameterFive( MetaProcess* );
83
84    virtual IsoString Id() const;
85    virtual size_type MinLength() const;
86 };
87
88 extern SandboxParameterFive* TheSandboxParameterFiveParameter;
89
90 PCL_END_LOCAL
91
92 } // pcl
93
94 #endif   // __SandboxParameters_h
```

Listing 7 — SandboxParameters.h (2/2)

eter. A block parameter is usually implemented as a `ByteArray` object, which is the `Array<uint8>` template instantiation in PCL. The core expects to find a contiguous sequence of bytes of the length specified by the parent instance —we'll see later how parameter lengths are provided by process instances.

MetaTable

A table parameter is a collection of rows, where each row consists of a list of process parameters, or *column parameters*. If you take a look at the documentation for `MetaParameter`, you'll notice that this class —as well as all derived classes except `MetaTable`— has two constructors: one that takes a `MetaProcess*` argument, and another one that takes a `MetaTable*` argument. The second constructor tells us that any process parameter class can be used to define a table column, with just one exception: `MetaTable`. The existence of this exception means that you cannot use a table parameter as a table column parameter: nested tables are not supported. With only this limitation, `MetaTable` allows you to define *really* complex process instance layouts.

Note that unlike the rest of process parameter types, the core will never try to access a table parameter. This is because a table is only a *structural* component of a process instance; its contents are represented by instantiations of its columns, which the core will read and write as it does with regular (non-column) parameters.

Sandbox Parameters

The Sandbox process defines five process parameters. The corresponding classes and definitions are on Listing 7 and Listing 8. These five parameters have been implemented just as «placeholders» to illustrate several parameter types and their typical implementations in a module. Their identifiers are "sampleOne",

```
 1 #include "SandboxParameters.h"
 2
 3 namespace pcl
 4 {
 5
 6 SandboxParameterOne*   TheSandboxParameterOneParameter = 0;
 7 SandboxParameterTwo*   TheSandboxParameterTwoParameter = 0;
 8 SandboxParameterThree* TheSandboxParameterThreeParameter = 0;
 9 SandboxParameterFour*  TheSandboxParameterFourParameter = 0;
10 SandboxParameterFive*  TheSandboxParameterFiveParameter = 0;
11
12 // ----------------------------------------------------------------------------
13
14 SandboxParameterOne::SandboxParameterOne( MetaProcess* P ) : MetaFloat( P )
15 {
16    TheSandboxParameterOneParameter = this;
17 }
18
19 IsoString SandboxParameterOne::Id() const
20 {
21    return "sampleOne";
22 }
23
24 int SandboxParameterOne::Precision() const
25 {
26    return 3;
27 }
28
29 double SandboxParameterOne::DefaultValue() const
30 {
31    return 0;
32 }
33
34 double SandboxParameterOne::MinimumValue() const
35 {
36    return 0;
37 }
38
39 double SandboxParameterOne::MaximumValue() const
40 {
41    return 1;
42 }
43
44 // ----------------------------------------------------------------------------
45
46 SandboxParameterTwo::SandboxParameterTwo( MetaProcess* P ) : MetaInt32( P )
47 {
48    TheSandboxParameterTwoParameter = this;
49 }
50
51 IsoString SandboxParameterTwo::Id() const
52 {
53    return "sampleTwo";
54 }
55
56 double SandboxParameterTwo::DefaultValue() const
57 {
58    return 1;
59 }
60
61 double SandboxParameterTwo::MinimumValue() const
62 {
```

Listing 8 — SandboxParameters.cpp (1/3)

```
 63    return 1;
 64 }
 65
 66 double SandboxParameterTwo::MaximumValue() const
 67 {
 68    return 100;
 69 }
 70
 71 // ----------------------------------------------------------------------------
 72
 73 SandboxParameterThree::SandboxParameterThree( MetaProcess* P ) : MetaBoolean( P )
 74 {
 75    TheSandboxParameterThreeParameter = this;
 76 }
 77
 78 IsoString SandboxParameterThree::Id() const
 79 {
 80    return "sampleThree";
 81 }
 82
 83 bool SandboxParameterThree::DefaultValue() const
 84 {
 85    return false;
 86 }
 87
 88 // ----------------------------------------------------------------------------
 89
 90 SandboxParameterFour::SandboxParameterFour( MetaProcess* P ) : MetaEnumeration( P )
 91 {
 92    TheSandboxParameterFourParameter = this;
 93 }
 94
 95 IsoString SandboxParameterFour::Id() const
 96 {
 97    return "sampleFour";
 98 }
 99
100 size_type SandboxParameterFour::NumberOfElements() const
101 {
102    return NumberOfItems;
103 }
104
105 IsoString SandboxParameterFour::ElementId( size_type i ) const
106 {
107    switch ( i )
108    {
109    default:
110    case FirstItem:  return "FirstItem";
111    case SecondItem: return "SecondItem";
112    case ThirdItem:  return "ThirdItem";
113    }
114 }
115
116 int SandboxParameterFour::ElementValue( size_type i ) const
117 {
118    return int( i );
119 }
120
121 size_type SandboxParameterFour::DefaultValueIndex() const
122 {
123    return Default;
124 }
```

Listing 8 — SandboxParameters.cpp (2/3)

```
125
126 // ----------------------------------------------------------------------------
127
128 SandboxParameterFive::SandboxParameterFive( MetaProcess* P ) : MetaString( P )
129 {
130    TheSandboxParameterFiveParameter = this;
131 }
132
133 IsoString SandboxParameterFive::Id() const
134 {
135    return "sampleFive";
136 }
137
138 size_type SandboxParameterFive::MinLength() const
139 {
140    return 0;
141 }
142
143 // ----------------------------------------------------------------------------
144
145 } // pcl
```

Listing 8 — SandboxParameters.cpp (3/3)

"sampleTwo", ..., "sampleFive". In PixInsight, it is customary to define process parameter identifiers following the mixed case Java naming convention: concatenated words, where all words begin with a capital letter except the first one. Unless you have a strong reason to use different criteria, please define your parameters in this way for the sake of coherence.

In SandboxInstance.h (Listing 5), from lines 29 to 33, you can see the actual implementations of the five process parameters as data members of the `SandboxInstance` class. These members define the data that the core will read and write directly for each instance of the Sandbox process. Note how the types of these data members are coherent with the types declared by their `MetaProcess` descendant counterparts: `float` for `MetaFloat`, `int32` for `MetaInt32`, `pcl_bool` for `MetaBoolean`, `pcl_enum` for `MetaEnumeration`, and `String` for `MetaString`, respectively. Always use data types for which the bit and storage sizes are unambiguously known in a platform-independent way; for example, using the `int` and `long` C++ types, or even —heaven forbid— platform-dependent things such as `LONG` or `WORD`, is *wrong*. Use `int8`, `int16`, `int32`, `int64`, or the appropriate size-enforcing types, and implement your parameters with the corresponding `MetaParameter` descendants.

In SandboxInstance.cpp (Listing 6), line 112, we have a *parameter locking routine*. This routine is the `LockParameter` member function reimplementation of the `SandboxInstance` class. As you see, this function has two parameters: a pointer to `MetaParameter` and an integer row index. When the core calls this function for a particular process instance, the first function argument identifies a process parameter and the second argument, in case the parameter is a table column, identifies the row index ≥ 0 in the parameter's parent table parameter.

The core expects that the locking routine will return the address of a contiguous region of memory where a data item of the appropriate type and length, according to the corresponding `MetaParameter` description, can safely be accessed for read and write operations. If the `LockParameter` function returns zero, the core interprets that as an error condition and aborts the ongoing operation.

In line 127 of SandboxInstance.cpp, you can see the *parameter allocation routine*, implemented as the `AllocateParameter` member function of `SandboxInstance`. The allocation routine is invoked by the core when a new instance of a variable-length parameter must be generated. Allocation is thus only necessary for parameters declared as `MetaString`, `MetaBlock` and `MetaTable` descendants. In the case of Sandbox, only 'sampleFive' (`MetaString`) requires explicit allocation. The `AllocateParameter` function has three parameters. The first one is the requested allocation length; the other two are for parameter identification and table row indexing, as before. Note that the term length here refers to a count of *elements:* characters (not bytes!) for strings, bytes for blocks and rows for tables.

The last parameter management routine is in line 141 of SandboxInstance.cpp (Listing 6): `ParameterLength()`. This function is called by the core to learn the length of a variable-length parameter for a particular process instance. The meaning of «length» is the same that we have described above for parameter allocation: length refers to *elements*, which are bytes *only* for `MetaBlock` parameters.

Instance Execution

Now that we know how a process instance is defined in terms of its *properties* —parameters—, it's time to learn about its *behavior*. Unless a process is an observer tool, it must be able to be *executed*. This introduces the concept of *execution context* in PixInsight. There are two main execution contexts: the *view context* and the *global context*. The view context refers to a process instance being executed on a *target image* pertaining to a view owned by an image window. This is the context where most processes normally work. For example, standard processes such as HistogramTransformation, UnsharpMask and DynamicCrop can only be applied to images.

The global context refers to a process that is executed without a target image. Global processes are less frequent but not less important. The palmary example of a global process is Preferences, which can be used to modify a large set of global variables (or settings) that change the behavior of the whole PixInsight platform. Another, perhaps less obvious example of global process is NewImage. This process is executed globally to create a new image window. Although it certainly works on images, it cannot be executed on an existing image, so its working context is always global.

There are other processes that are able to work in both contexts. For example, the RGBWorkingSpace process can be executed on a view, to define the image's private RGBWS, or globally to define the default RGBWS in the PixInsight platform. A perhaps less known example of process that can work in both contexts is PixelMath. This process has a 'destination' parameter that selects whether the result will replace an existing target image, or if it will be sent to a newly created image window. In the second case PixelMath works as a global process.

The view execution routine is a reimplementation of `ProcessImplementation::ExecuteOn()`. SandboxInstance.h (Listing 5) declares it in line 20. Note that this function receives a reference to a `View` object as its unique argument, which is the target view where the instance is being executed. Before calling `ExecuteOn()` though, the core will always request permission by calling `CanExecuteOn()`, which should either return `true` to allow view execution, or `false` to prevent it. In the latter case, `CanExecuteOn()` *must* also tell why it doesn't allow execution on the specified target view, by providing a human-readable —and hopefully not too long— description of the reasons as the value of its `whyNot` argument.

The definition of `ExecuteOn()` for the `SandboxInstance` class, which you can see in line 66 of SandboxInstance.cpp (Listing 6), provides an example of how a process implements a *sample type independent* image processing routine in PixInsight. Note in line 72 how the view's image can be accessed through an `ImageVariant` object. `ImageVariant` acts much like a C union for all image types supported in PixInsight. Between lines 80 and 98, a decision tree is used to solve the `ImageVariant` abstraction through template instantiations covering all image types for which the process can work (in this case, all types except complex images). The actual processing tasks are implemented by a private template class, which we customarily call an *engine class* in PCL. These template-based techniques play an essential role in PixInsight; they are the foundations of PixInsight's transparent support for seven pixel data types: 8-bit, 16-bit and 32-bit integers, plus 32-bit and 64-bit real and complex floating point images. Now you know why PCL makes extensive use of complex C++ templates, which you should be prepared to deal with if you are going to develop on the PixInsight/PCL platform.

For simplicity, we haven't included global execution capabilities in `SandboxInstance`. This article is just a general introduction; we strongly

recommend that you study the corresponding sections of PCL's reference documentation, as well as the numerous open-source modules that come with PCL distributions.

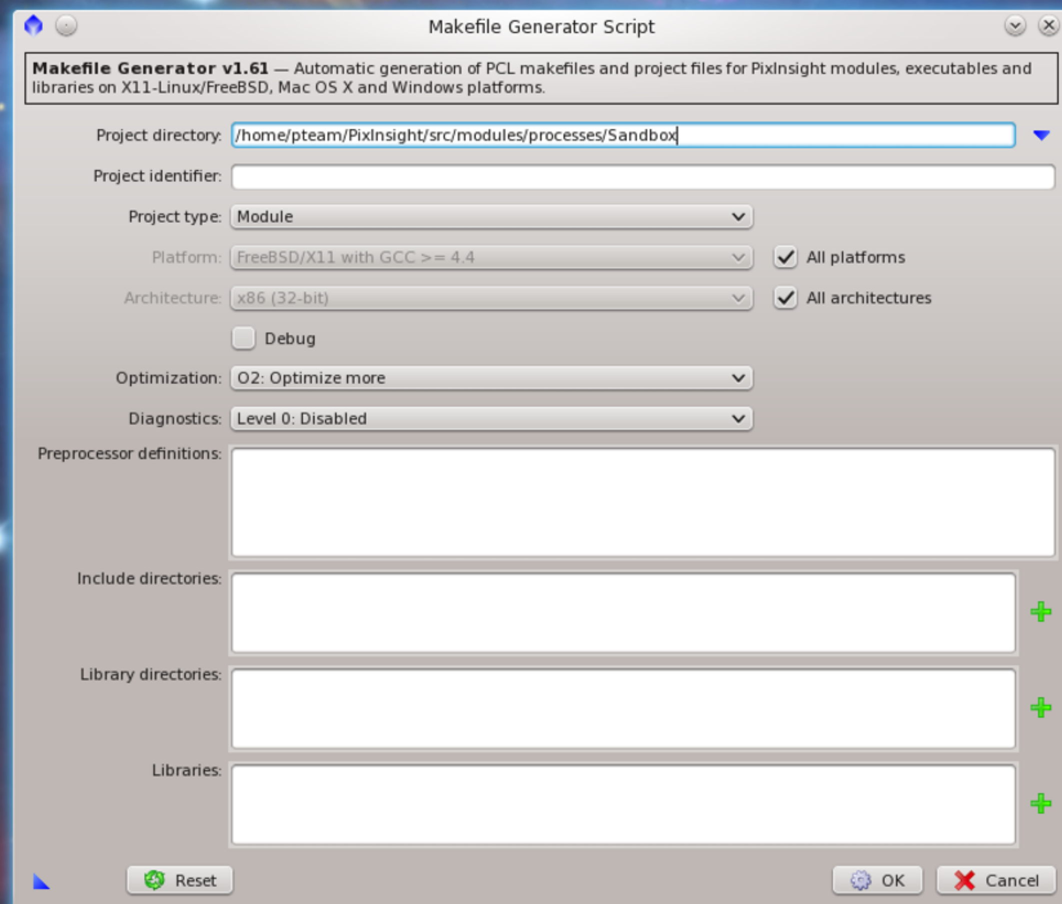
Process Interfaces

As I have said earlier, process interfaces are quite complex and cannot be covered properly in a general introductory article; we'll cover this subject in future development documents. For this reason we'll limit ourselves and won't include the source code for the SandboxInterface.h and SandboxInterface.cpp files in this article. You already have these files in all PCL distributions, and there are also many examples among the open-source standard modules that you can use to begin exploring PCL graphical interfaces. Along with all of that practical material, the PCL Reference Documentation is quite complete for the `ProcessInterface` class, which is the abstract base class for all PCL process interfaces.

Building Modules: The Makefile Generator Script

Makefile Generator is part of the standard set of scripts distributed with the PixInsight Core application. This JavaScript script is a PCL development utility for automatic generation of makefiles and project files on all supported platforms: FreeBSD, Linux, Mac OS X and Windows, including 32-bit and 64-bit builds. You can see a screenshot of the script's main dialog on Figure 5. Makefile Generator is rather easy to use. The script expects all the source code for your module located under a single directory; this is customary practice in PixInsight/PCL development. You just select the module's source directory as the *project directory*, and leave the rest of script parameters with default values. The module name —the name of the module's shared object— is the name of the project directory unless you specify a different *project identifier*.

Figure 5— The Makefile Generator script.



You can also specify a platform and an architecture, although you normally will leave the *All Platforms* and *All Architectures* check boxes with their default checked state. This will generate makefiles for all supported platforms and x86 and x86_64 architectures. Makefile Generator allows you to create makefiles and project files for several types of projects: modules, static libraries, dynamic libraries and standalone executables. There are other project types but they are for internal PTeam use exclusively.

Makefile Generator automatically selects all the necessary libraries and shared objects for all platforms and architectures, including all standard PCL libraries, so you normally don't need to change further script parameters. In case your project has specific requirements for custom include files and directories, additional libraries or preprocessor directives, you can specify them with the corresponding parameters, as you can see on Figure 5. There are other options for debug builds, custom optimizations and automatic PCL diagnostics code generation; these are well explained by tool tips on the script's dialog.

Note that Makefile Generator is an *instantiable script*. You know this from the blue triangle icon at the bottom left corner of the script's dialog. An instantiable script allows you to generate Script process icons by dragging the blue icon to the workspace, so you can save your settings for later use, and store them as standard XPSM files. This is particularly useful when you have custom settings such as preprocessor directives and special directories and libraries.

When you execute Makefile Generator, it creates a number of directories under your project's root directory, one for each platform: *freebsd*, *linux*, *macosx* and *windows*. Each of these directories is populated with the necessary makefiles and project files to build the module on each platform and architecture. For UNIX (FreeBSD, Mac OS X) and Linux platforms, standard GNU makefiles are generated. For example, in the Linux case the script will generate the following makefiles:

```
<module-dir>/linux/Makefile
<module-dir>/linux/makefile-x86
<module-dir>/linux/makefile-x86_64
```

So you just have to go to the linux subdirectory and enter the 'make' command to build your module on Linux, both 32-bit and 64-bit versions

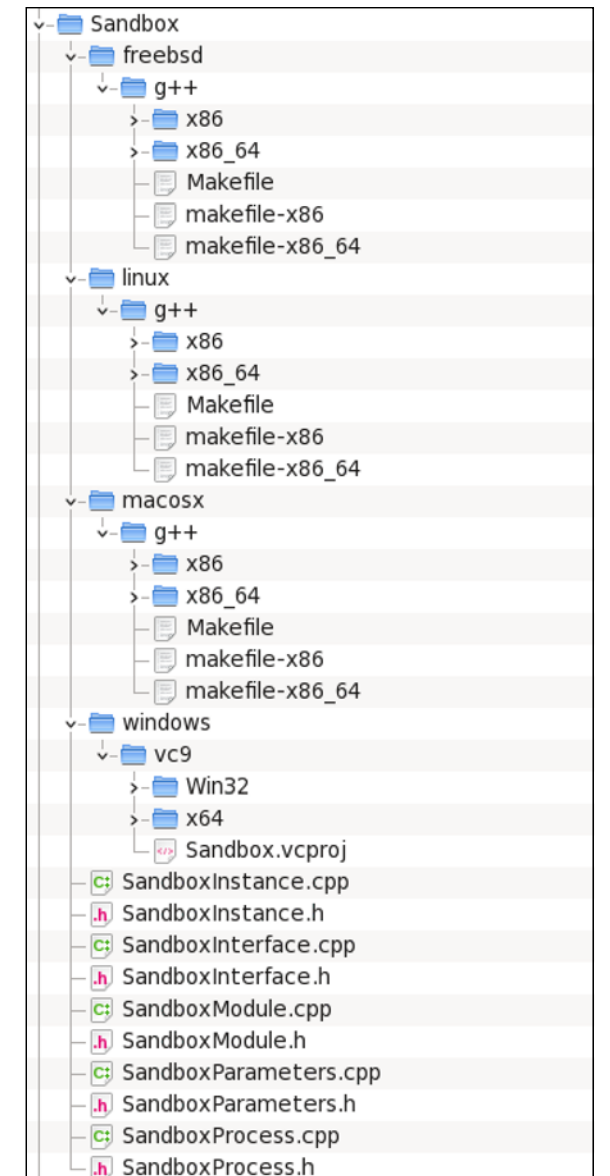


Figure 6— Makefiles for the Sandbox module.

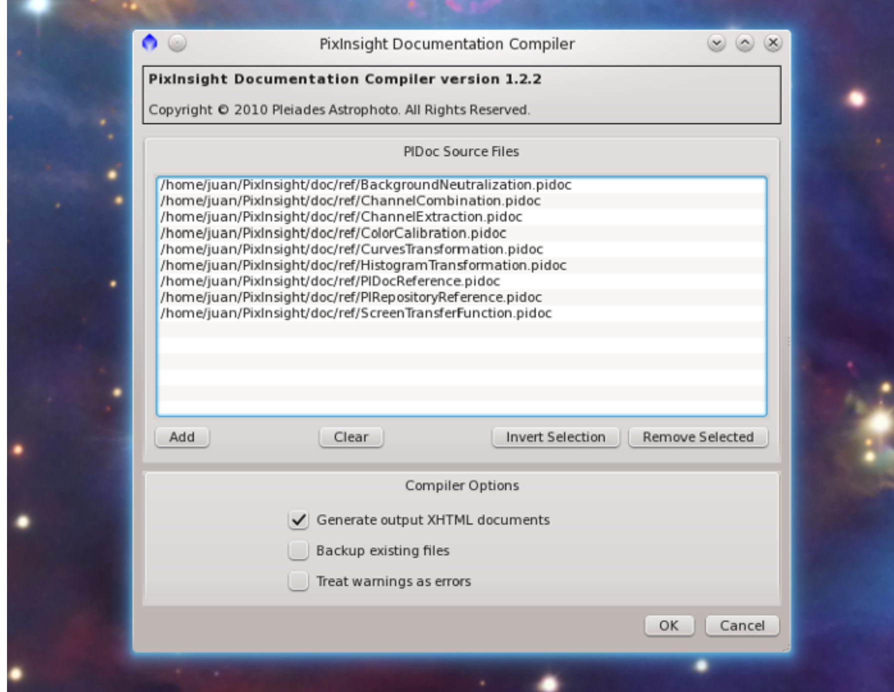
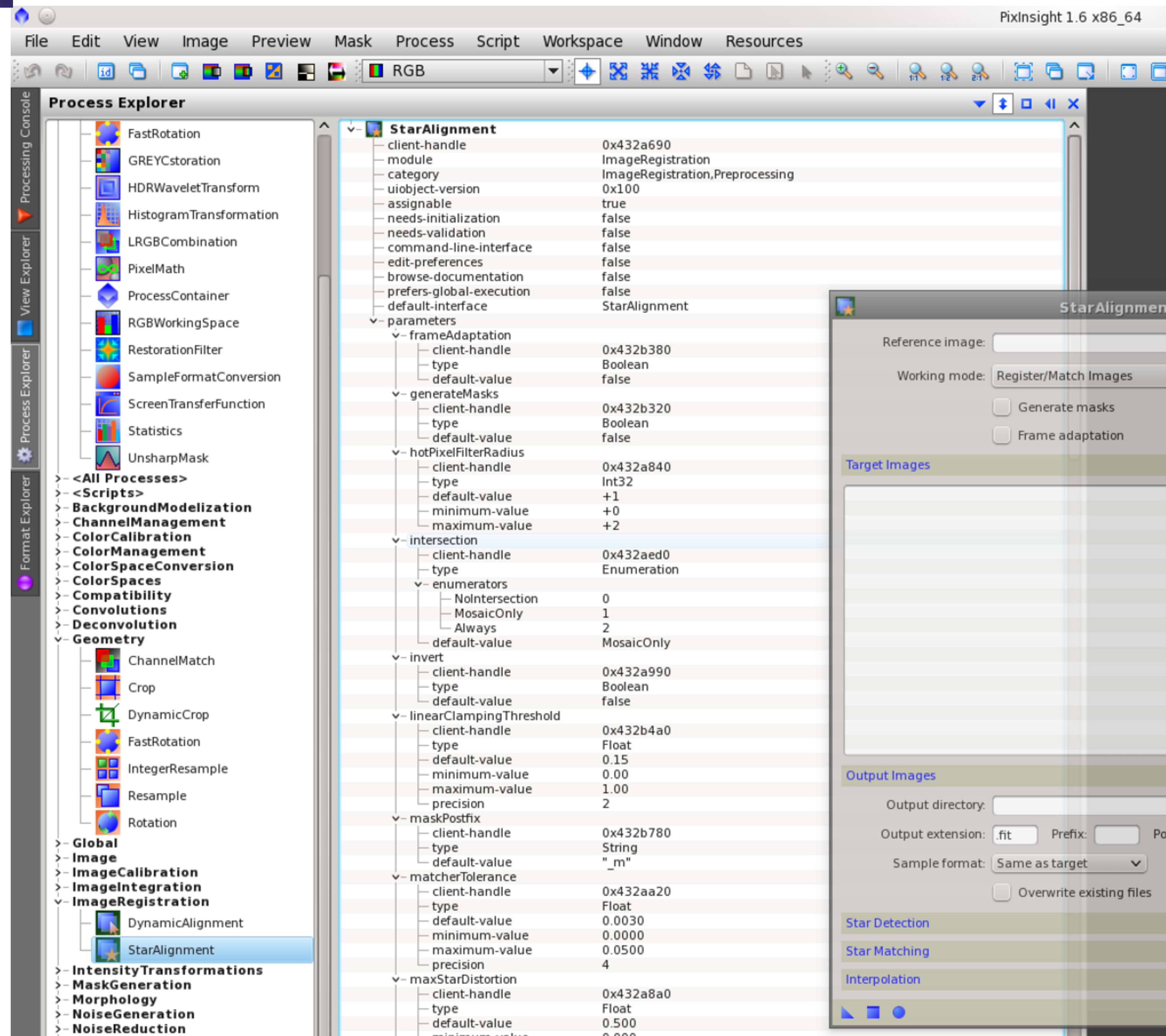


Figure 7— The Documentation Compiler script.

Juan Conejero is the author and principal developer of the PixInsight image processing application and the PixInsight Class Library (PCL). He is the co-founder and CEO of Pleiades Astrophoto, the Spanish software development company that produces PixInsight.

Figure 8— The Property Browser.



—note that we are assuming that you work on a 64-bit machine and operating system, as that is a necessary condition to build modules for 64-bit architectures. In case you just want to build the 64-bit version, for example, you should enter the following command:

```
$ make -f makefile-x86_64
```

Note that on FreeBSD you must use `gmake` instead of `make` to run the GNU make utility.

On Figure 6 you can see the set of makefiles and project files that the Makefile Generator script generates for the Sandbox module.

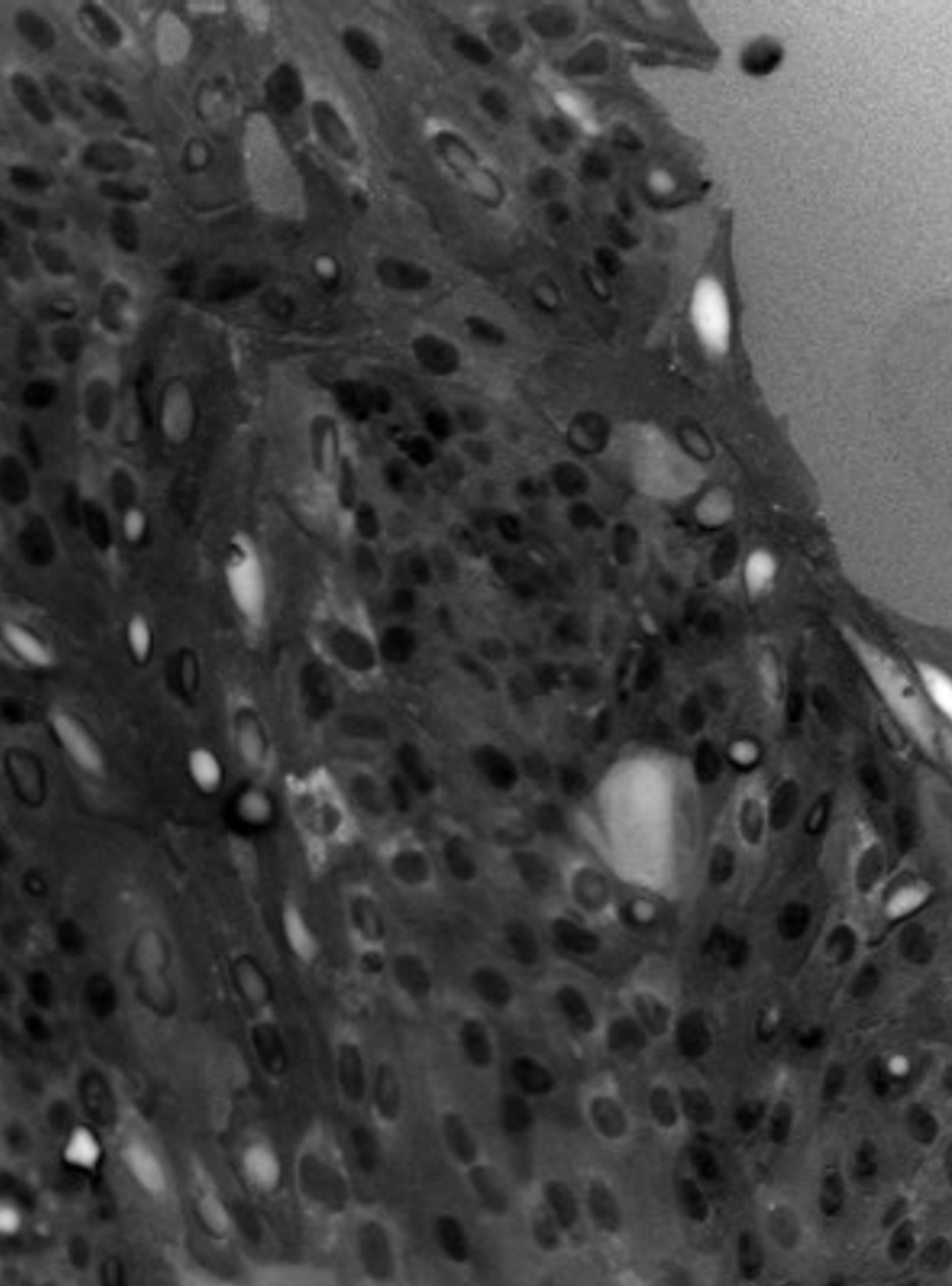
Inspecting Processes: The Property Browser

The Property Browser is a graphical inspection tool integrated with the Process Explorer window in the PixInsight Core application. It allows you to inspect a process and all of its properties, including process parameters. This can be very useful as a debug and analysis tool because it allows you to verify the actual state of a process as it is being seen by the core.

To work with the Property Browser, open the Process Explorer window and extend its right panel (by default, the right panel is hidden) by clicking the double arrow button at the bottom. Then select the Property Browser by clicking the corresponding button, also at the bottom of the Process Explorer's right panel (by default, the Documentation Browser is selected instead). Once you have the Property Browser visible, select any process on the Process Explorer's left panel to inspect it. You have an example on Figure 8.

The Documentation Compiler Script

Last but not least, you should write documentation for all the processes and tools that you create. The PixInsight Reference Documentation System allows you to integrate your own documentation with the PixInsight Core application. For this purpose, you must write all your documentation in the PIDoc document definition language, then compile it with the Documentation Compiler script (Figure 7) to generate an XHTML document. For more information on PIDoc and the PixInsight Reference Documentation System, please refer to the following document: *http://pixinsight.com/doc/docs/PIDocReference/PIDocReference.html*



PixInsight as a Research Platform

Carlos Milovic

Since the beginning, the PixInsight project was aimed to be much more than a field-specific processing package. Many times we have referred to it as an image processing platform, and that's quite true. Despite the fact that many processes had indeed been developed with astrophotography in mind, most of them are general-purpose algorithms applicable to almost any imaging field. In addition, PixInsight's object oriented architecture allows developers and researchers to integrate the modules with scripts, or to create their own specific modules thanks to the PixInsight Class Library. Both facts allow PixInsight to become an invaluable research tool. In this brief article, I'm going to share my experiences using it at my daily research job at the Biomedical Imaging Center of the Pontificia Universidad Católica de Chile.

All the biological background of the project aside, what I do as an engineer is to analyze two set of images, showing the calcium concentration in a rat tissue sample, where a calcium wave propagates from a mechanically stimulated cell. Since they are using a fluorescence marker, the calcium concentration is calculated from the ratio of images at each of these sets, acquired at different wavelengths. The raw data is stored as 8-bit TIFF files and acquired with a CCD camera —yes, pretty much the same as in astrophotography. The goal was to design and implement a software that reads the raw data, calibrates it, and then extracts certain parameters to characterize the wave propagation, such as amplitude, speed, decay rate, etc.

My decision was to implement all of these as a new PixInsight module that contains several processes to address each stage of the analysis work. An alternative could be using a standard in the image processing field: MATLAB®. But, since it uses an interpreted language, which is slower, and I wanted something more user friendly and almost standalone, I discarded that option. PixInsight modules run native C++ code, which is pretty fast. Also, the PCL has been highly optimized and parallelized, so as long as I use already implemented functions, there is no much need for additional care on code optimization.

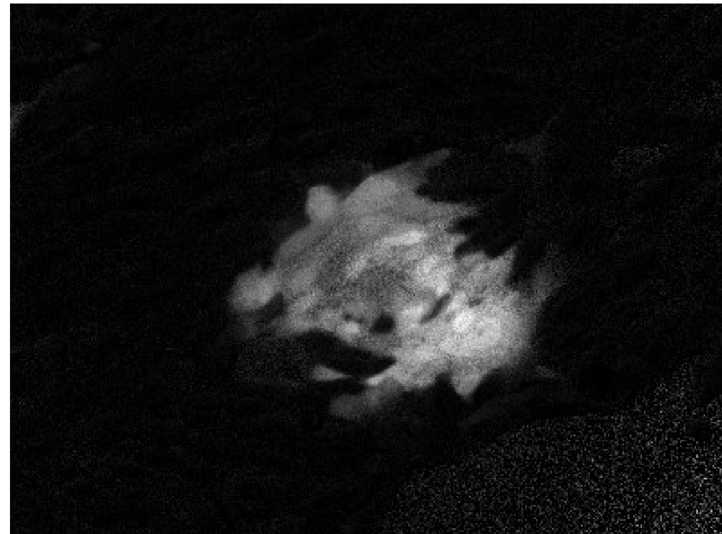
But writing a new module may be a time-consuming task. Testing new procedures or algorithms may take a while, and we may end wasting a lot of time just to get it working. Fortunately, PixInsight offers an alternative to modules: the PixInsight JavaScript Runtime (PJSR). I had almost no experience with PJSR and the JavaScript language. Luckily, it is an object-oriented language like C++ and its syntax is very similar to C, so even if I don't know exactly how it works, for me it is readable. There are also many examples that can be adapted to our own needs. Gathering pieces from here and there, using an existing script as a basis, I was able to quickly implement my algorithm and test

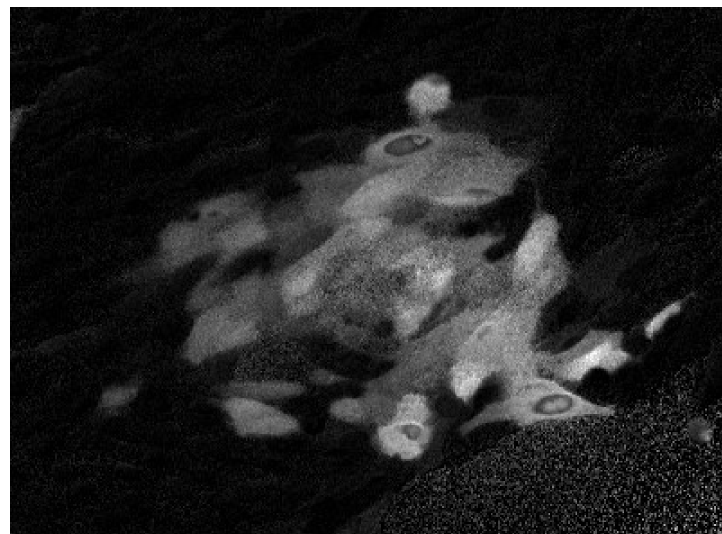
Previous page: A rat tissue sample. After calibration and correction for marker decay, the basal conditions are calculated.

On this page, top: At a certain time, a single cell is mechanically stimulated, and calcium is released to the media.

Middle: The calcium wave propagates, as other cells release their own stores.

Bottom: Eventually, the wave stops, and the cells reabsorb or expel the calcium slowly.





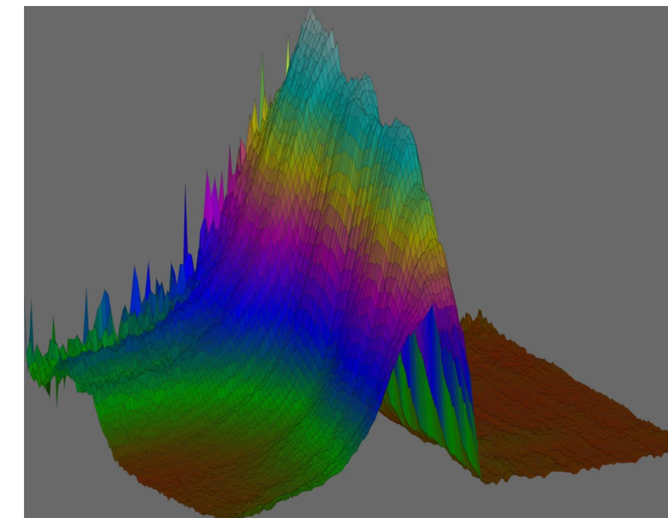
its results. Of course, it was slower than native C++ code, but it gave me a very nice idea of what happened, and allowed me to adjust the code for optimal results. Another benefit of the JavaScript engine is that one may interact with already installed modules. Creating and testing a morphological filter, for example, is easier in PJSR than in C++/PCL code.

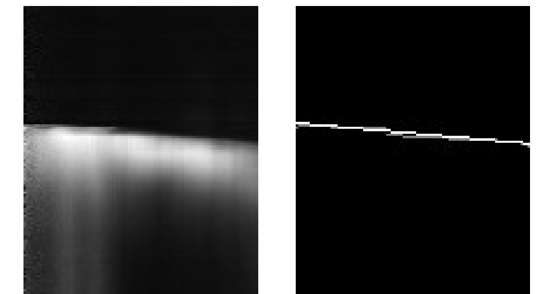
Another advantage is that the JavaScript engine is pretty much a clone of PCL, in the sense that it uses almost the same class definitions and functions, so porting JavaScript code into C++ doesn't require to rewrite everything again.

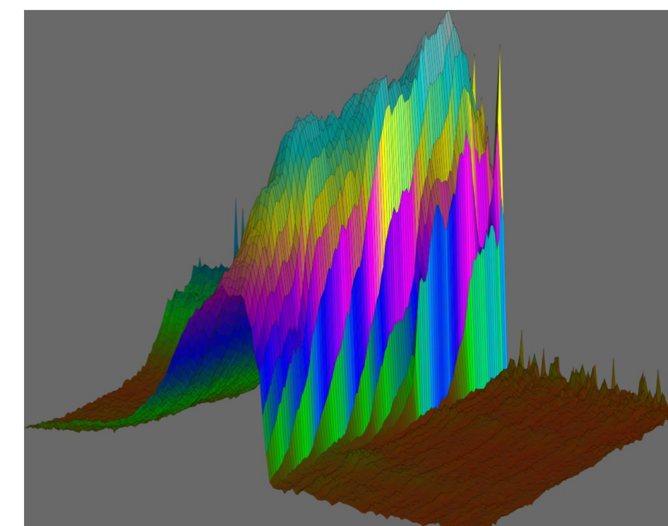
There are many JavaScript source code examples, and the same happens with modules and processes. This is an invaluable source of knowledge, and using an existing module as a starting point will save large amounts of time. With the time, you'll gather a library of code with your own solutions to specific problems, and writing each new module will be easier and easier.

Also, just like in any C++ code, you may use external libraries to help you. For instance, to solve nonlinear least square function fitting problems I used the CMinPack library. Everything works smoothly and is cross-platform, so my biologist partners use my module compiled on Windows, even if I write and test the code on Fedora Linux.

So, from a developer's point of view, PixInsight is really a very powerful platform that may host every one of our research projects.







Top row, middle: Since the wave propagation follows a radial propagation, integration of concentric one-pixel width rings around stimulus simplify the analysis, giving a graph of the radial progress over time of the concentration (vertical axis is time, horizontal is radius size).

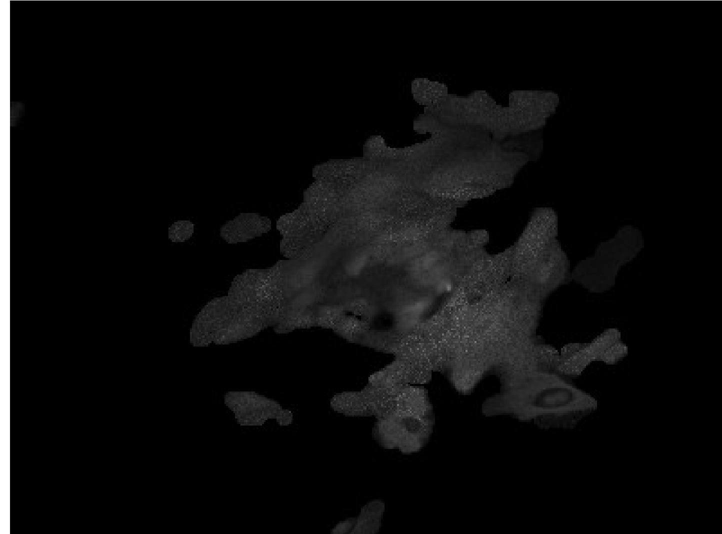
Top row, right: Studying the gradient of this graph allows us to follow the wavefront. The slope of a linear fit to the trajectory gives the velocity of the wave.

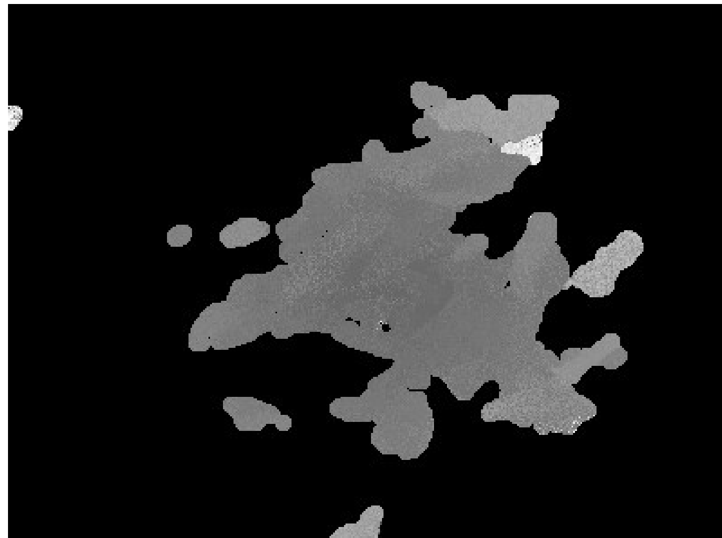
Left column: With the aid of the 3DPlot script, a better visualization of the graph allows for a deeper understanding of the data. See how the calcium in the inner regions decay at a slower rate, and there is still a remnant at the end of the experiment.

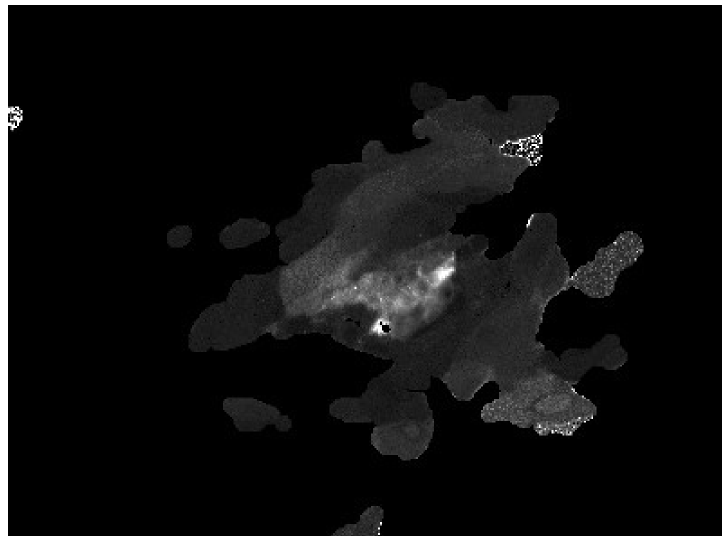
Top: Nonlinear modeling of decay functions gives more input of the wave behavior. Here is the fitted wave amplitude.

Middle: The model predicts the time of activation (darker means earlier). The cells activate almost as blocks, very fast, following a radial pattern from the stimulus.

Bottom: Finally, the decay rate is revealed to be much greater on the stimulated cell and its neighbors.

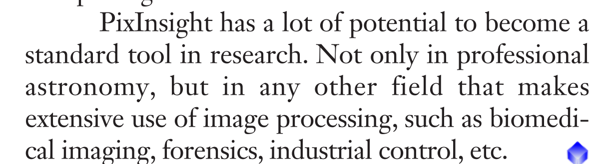






As I said earlier, PCL simplifies many tasks quite a lot. And not only from the performance point of view. Many algorithms like convolutions, morphological filters, wavelet transforms and many more, are already implemented as PCL classes. Using them with our modules is transparent and straightforward. The extensive use of templates also helps to manage any type of image, including any bit depth, or even complex floating point images.

From the user's point of view, once you understand PixInsight's GUI paradigm —as has been said somewhere else, if you aren't too accustomed to other software packages, understanding PixInsight's GUI is quite easy— the use of the modules is very intuitive, and may give the user a lot of control over the processes. PixInsight's console provides a wealth of information, and it may be quite useful to supervise the progress and to read results. On the other hand, the capability to manage 32 and 64-bit floating point images prevents any intermediate discretization problems, so the user can see the whole process from a more analytic point of view. Furthermore, even with a module that solves the user's specific needs, he/she may also use the standard modules and scripts to go even beyond, like 3D plotting for better representation of data, or use the ScreenTransferFunction process to inspect images. The Statistics and HistogramTransformation processes are also invaluable. Since PixInsight has been designed for image processing, management of this kind of data is much more pleasant than in other scientific packages.



PixInsight has a lot of potential to become a standard tool in research. Not only in professional astronomy, but in any other field that makes extensive use of image processing, such as biomedical imaging, forensics, industrial control, etc.

Carlos Milovic works as a research engineer at the Biomedical Imaging Center of the *Pontificia Universidad Católica* of Chile. He is an active PTeam member and PixInsight developer who has contributed numerous tools, including the AutomaticBackgroundExtractor process for automatic sky gradient modeling and correction.

NEWS COMMUNITY

The PixInsight Community has been very active lately, and PixInsight Forum's statistics confirm that. At the beginning of March 2010 we reached 10,000 posts. Right now we have over 18,000 forum messages, with a consistent average of more than 700 new messages per month. The highest peak of activity was reached during September 2010, when we got over 1300 new messages. Keep them coming! Forum membership is also growing at a steady rate. Right now we have over 650 registered members. Thanks for joining the cause, and welcome aboard! But these numbers don't quite reflect the extraordinary contributions to the project that our users have been making. Let's review some highlights from the past months.

Harry Page has become one of our most active contributors with his great series of PixInsight video tutorials. Watching his newbie and medium/advanced videos is a must for all PixInsight users:
http://harrysastroshed/pixinsighthome.html

Ken Hudson interviewed Juan Conejero and Harry Page on Share Astronomy:
http://www.shareastronomy.com/blog_posts/45
http://www.shareastronomy.com/blog_posts/46

Jordi Gallego has contributed with tutorials and presentations:
http://jordigallego.fotografiaastronomica.com/articles.html

Vicent Peris is co-author of a Letter to the Editor on Astronomy & Astrophysics, published May 11, 2010: *Herschel images of NGC6720: H_2 formation of dust grains,* by P.A.M. van Hoof et al. In this article Vicent has used PixInsight to carry out all the reduction of the H_2 infrared data acquired with the 3.5 meter telescope of Calar Alto Observatory. These data were used in combination with images acquired through the Herschel infrared spatial telescope.

Niall Saunders started a forum thread that led to a very nice compilation of PixelMath hints are tricks, implementing different blending methods:
http://pixinsight.com/forum/index.php?topic=2409.0
Niall is also the author of a new video tutorial on the StarGenerator tool:
http://pixinsight.com/forum/index.php?topic=2192.0

Rowland Cheshire compiled a very nice tutorial on DSLR image calibration, gathering different sources of information on the forum:
http://pixinsight.com/forum/index.php?topic=2570.0

Enzo de Bernardini published a DSLR tutorial in Spanish on his homepage:
http://astrosurf.com/astronosur/pixinsight/preprocessing-1.htm

Rogelio Bernal made a significant collaboration just before the PixInsight 1.6 release: The *Unofficial PixInsight Reference Guide*. This document goes through all standard tools included in PixInsight Core distributions, providing a brief description of its purpose and some details on parameters and interface options:
http://blog.deepskycolors.com/PixInsight/

In Rogelio's homepage you'll find also some PixInsight tutorials, like this brand new article about HDR composition:
http://blog.deepskycolors.com/archive/2011/01/19/HDR-Composition-with-PixInsight.html

Rogelio was also a speaker at the Advanced Imaging Conference (AIC) 2010, held on October 22-24 in Santa Clara, California. During his talk, Rogelio described some of his processing techniques with PixInsight. This year he won AIC's Pleiades Award.

Sander Pool gave a lecture on PixInsight at the Mid-West Advanced Imaging Conference (MWAIC)2010, held on July 23-24, in Chicago. He has also published a series of articles about PixInsight on the AstroPhoto Insight Magazine. Sander is a PixInsight/PCL developer; his Debayer module has been included in the latest releases as an official module, and there is a new process called DynamicProfile that provides a nice plot of the intensity of pixels along a line and works as a dynamic tool. This process is on the final stages of development. More details about that in our development news section.

Yuriy Toropin announced a new imaging project on May 2010: *Ultra Deep M51*. The goal of this project is to gather image sets from several amateur astronomers, to achieve the deepest amateur picture of this field ever. A preliminary result was published on June. See here for the announcement and the images:
http://pixinsight.com/forum/index.php?topic=1878.0
http://pixinsight.com/forum/index.php?topic=1988.0

PixInsight Workshop by Vicent Peris Adler Planetarium, Chicago, USA, September 10–12 2010

From left to right:
Geoffrey Stone, Cleon Wells, Larry Van Vleet, David Illig, Vicent Peris, Joe De Pasquale, Philip de Louraille, Robert Hurt and Max Mirot.

More information on this workshop on our website:
http://pixinsight.com/workshops/adler-2010/
and also on this thread of PixInsight Forum:
http://pixinsight.com/forum/index.php?topic=2333.0

Congreso Austral de Astrofotografía ESO Facilities, Santiago, Chile, November 26-27

Carlos Milovic gave three talks on the first Austral Astrophotographic Conference: an introduction to PixInsight, a talk about HDR techniques, and a beginner's introduction to histograms.

Some of Carlos' presentations are available on the official webpage of the event, along with more pictures:
http://www.astrofotografos.cl/

Photos courtesy of Leonardo Julio

NEW SCRIPTS

CanonBandingReduction by Georg Viehoever
http://pixinsight.com/forum/index.php?topic=1159.0
This script allows the user to correct a banding pattern that is generated by Canon's CMOS sensors.

BatchDebayer by Ken Pendlebury
http://pixinsight.com/forum/index.php?topic=1731.0
This script allows the user to select a large set of images to apply a deBayer process as a batch procedure.

Updated MaskedStretchTransform by Andrés Pozo
http://pixinsight.com/forum/index.php?topic=1737.0
Andrés modified David Serrano's original script, increasing the performance of the algorithm. The modified version runs significantly faster, and even if it lacks some accuracy with respect to the original, the results are close enough for a highly beneficial trade-off.

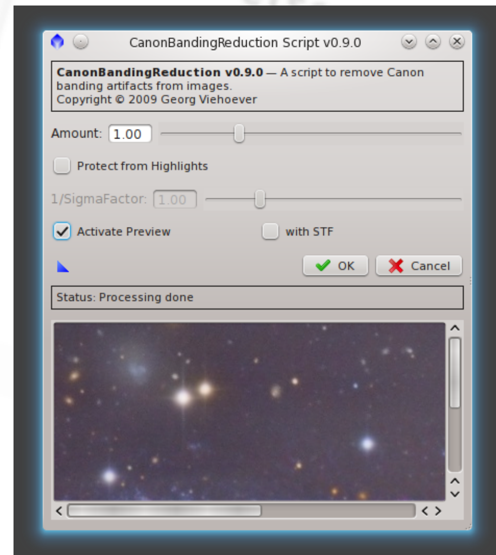
SubstituteWithPreview by Juan M. Gómez
http://pixinsight.com/forum/index.php?topic=1747.0
This is a nice tool to replace the pixel contents of an image with the contents of a preview generated on another image. Very useful to build better star masks.

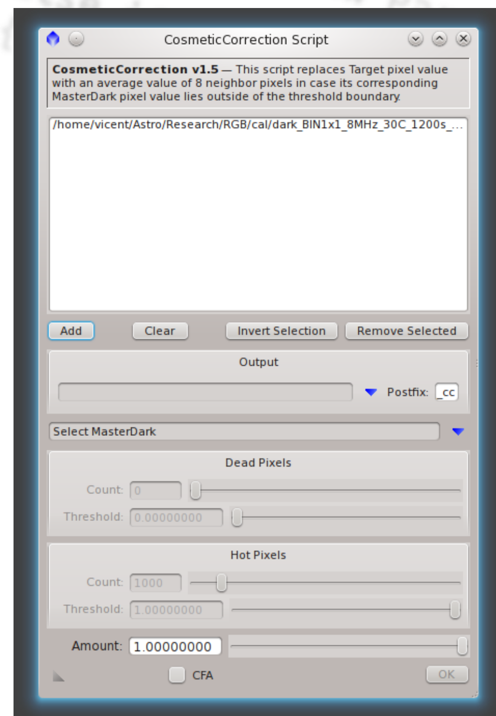
LensCorrection by Juan M. Gómez
http://pixinsight.com/forum/index.php?topic=1794.0
Still under development, this script implements geometrical corrections for pincushion and barrel deformations. This script will be useful to fix the distortions generated by optical systems, especially wide angle lenses.

NarrowbandCombination by Juan M. Gómez
http://pixinsight.com/forum/index.php?topic=1714.0
Another script still under development. This script has shown promising results combining data from narrowband filters and monochrome cameras.

BackgroundEnhance by Juan M. Gómez
http://pixinsight.com/forum/index.php?topic=1921.0
A simple way to deal with background enhancement. Nice interface and intuitive controls.

CosmeticCorrection by Nikolay Volkov
http://pixinsight.com/forum/index.php?topic=1828.0
This script uses a dark frame as a defect map to fix hot and cold pixels. A great companion to the ImageCalibration process.





StarHaloReducer by Juan M. Gómez
http://pixinsight.com/forum/index.php?topic=1427.0
A very useful tool to reduce the halos generated by bright stars. Manual selection is required, but the script automates most of the process.

MergeMosaic by Georg Viehoever
http://pixinsight.com/forum/index.php?topic=2124.0
For Windows only, this script uses a Poisson solver to merge mosaic images in the gradient domain, yielding seamless results. Needs the solver installation to work.

StarTrailsBlending by Enzo De Bernardini
http://pixinsight.com/forum/index.php?topic=2122.0
Enzo's first script automates the creation of long star trails from a series of short exposures. Not only saves the final result, but also intermediate steps, allowing new forms of animations to be done.

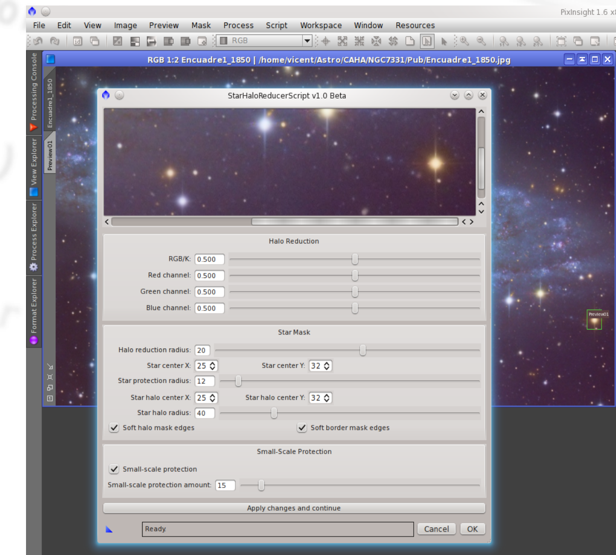
PropagatePreviews by Enzo De Bernardini
http://pixinsight.com/forum/index.php?topic=2190.0
Simple tool to transport previews to several images.

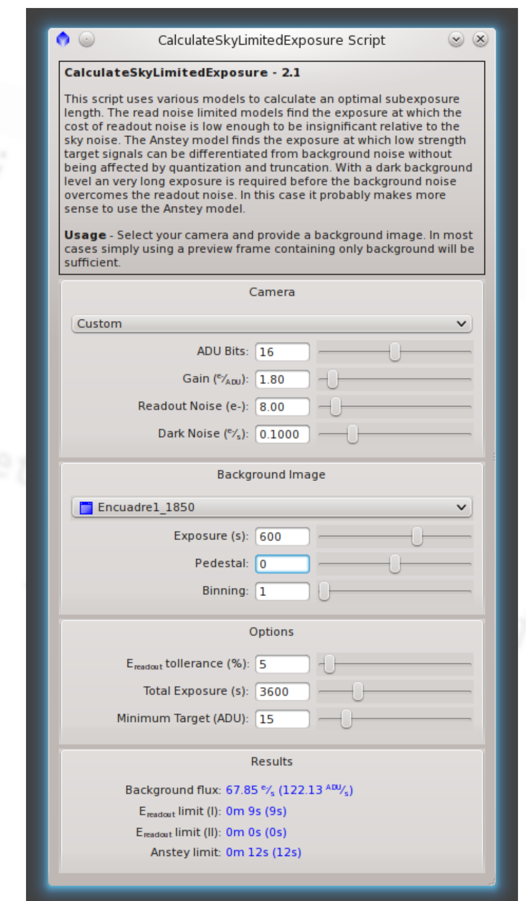
Animation by Nikolay Volkov
http://pixinsight.com/forum/index.php?topic=1895.150
A script that animates several files, enabling a fast way to perform blinking between images. Latest update (2.9.1) is avalaible from the forum, following the link above. Here's a movie where this script is the protagonist:
http://pixinsight.com/videos/blink-script/blink-the-movie-1.mov

LotsOfConvolution by Christoph Puetz
http://pixinsight.com/forum/index.php?topic=2261.0
33 common convolution matrices, compiled by Christoph. Low-pass, high-pass, edge detectors and many other useful filters.

BatchChannelExtraction by John Brown
http://pixinsight.com/forum/index.php?topic=2476.0
Extracts user selected channels from RGB images, and writes them as new grayscale images.

CalculateSkyLimitedExposure by Sean Houghton
http://pixinsight.com/forum/index.php?topic=2584.0
With the CCD users in mind, Sean wrote this script to inspect raw data images and help you in the calculation of the sky limited exposure for a given site.





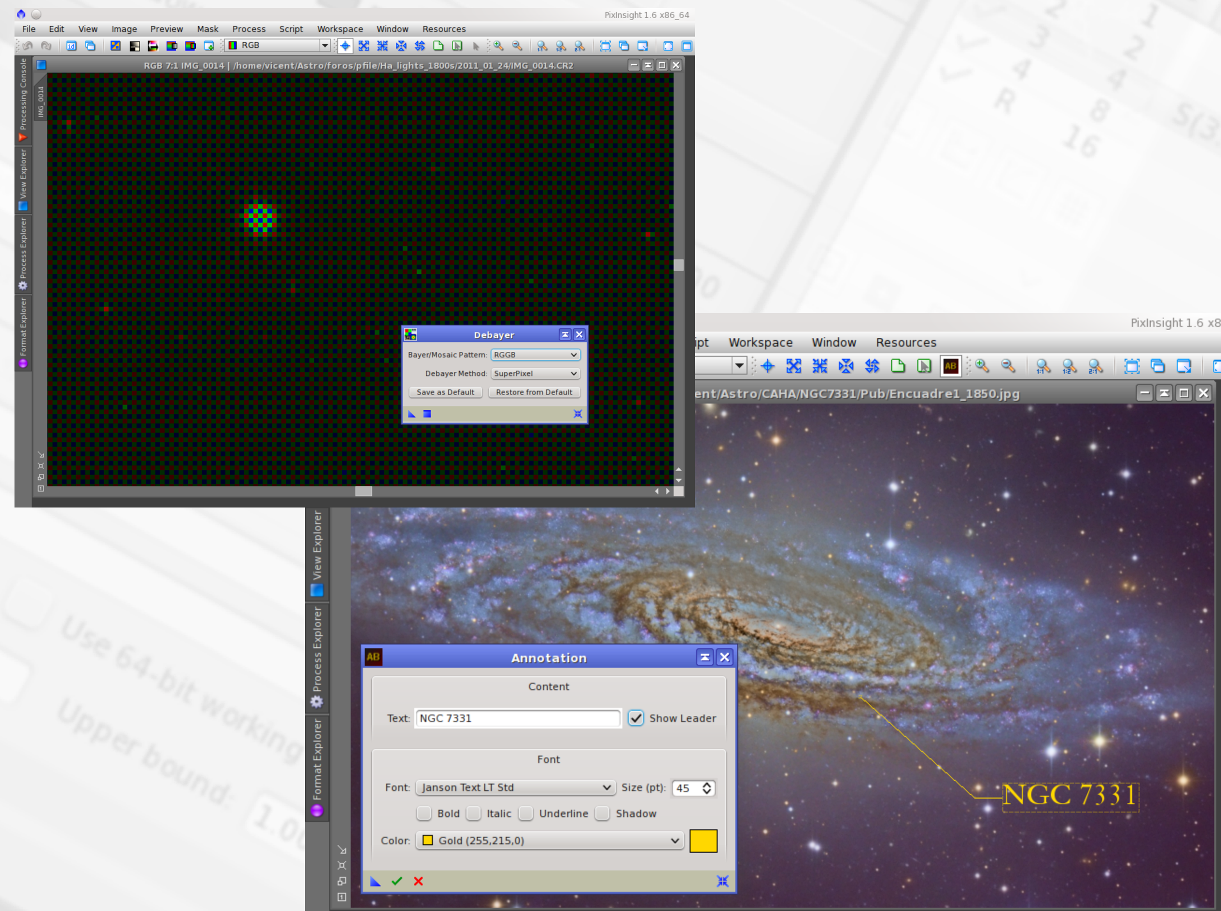
NEWMODULES

Debayer by Sander Pool
Many times a manual deBayer procedure is needed, and this is the tool of choice for this task. The latest version is included as an official module with PixInsight Core distributions.

DynamicProfile by Sander Pool
http://pixinsight.com/forum/index.php?topic=2247.0
Still under development, this is a dynamic process that reads pixel values along a user defined line across the image, and plots the intensities on its interface. A very useful tool for visual inspection of image changes. No beta releases are available right now.

AssistedColorCalibration & Annotation by Zbynek Vrastil
http://pixinsight.com/forum/index.php?topic=2577.0
Two very useful modules with excellent implementation and design. The first one allows to perform a manual color calibration (white balance) with previewing capabilities. The Annotation module is an interactive text rendering tool. It renders a single line of text on an image, with user-selectable font, font style, color, shadow, opacity and leader line.

ImageAcquisition by David Raphael
http://pixinsight.com/forum/index.php?topic=2681.0
This is an exciting and promising new development project. David is working on a camera control and image acquisition module. This project is still in its early development stages.

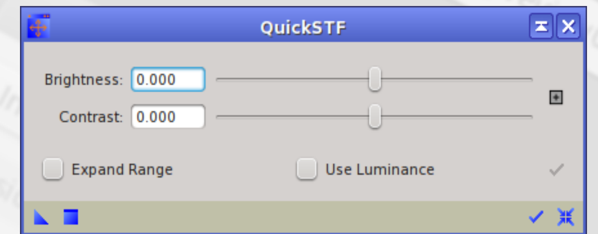


Development Modules by Carlos Milovic
http://pixinsight.com/forum/index.php?topic=2275.0
Since they are still not included in the standard set of processes, I'll briefly describe several tools that are avalaible for download for 32-bit and 64-bit Windows and 64-bit Linux.

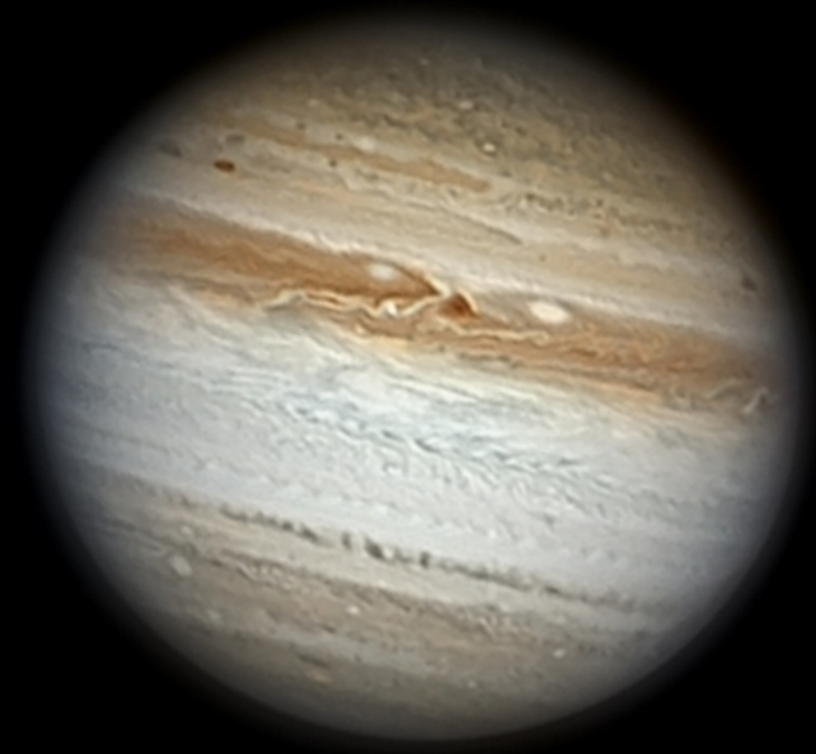
CMConv Module. A collection of convolution processes.
• Blur: Very simple implementation of "average type" low-pass filters.
• GaussianBlur: Blurring with a user-defined Gaussian function. User-defined standard deviation, kurtosis, asymmetry, and rotation angle.
• GradientMapper: This process calculates the gradient of an image (with user-defined filters) and creates a new image with the gradient modulus. Gradient angle output is optional.
• Sharpen: Simple implementation of Laplacian high-pass filters. Does a good job with small-scale contrast enhancement.

CMGeneral Module. A collection of general-purpose processes, mainly for preprocessing tasks.
• FFTRegister: This is an implementation of the FFTRegistration script by Juan Conejero, with a wavelet-based optimization. In some cases, isolating a single wavelet layer generates better results, especially when noise is prominent at high frequencies.
• SelectiveBlend: Compose a color image from grayscale channel images with user-defined percentages. Useful to blend narrowband images.

CMIntensity Module – A collection of intensity transformations.
• GammaStretch: A simple gamma stretch, with real-time preview.
• HistogramEqualization: The goal of this process is to compute a transformation function that yields a flat histogram. This means that pixel values will be distributed equally throughout the available numeric range. Fast and simple algorithm that works well with many daytime pictures. Future development will include histogram specification, that will allow for better results in other applications.
• QuickSTF: Instead of changing the STF by black and white points and a midtones balance, this process defines brightness and contrast parameters. An interactive mode lets the user to change the STF by clicking on the image, and the pixel coordinates will modify the SFT parameters.



CMSegmentation Module – A collection of processes related to mask creation, object detection or feature extraction.
• CHT: CHT stands for Circular Hough Transform. This algorithm transforms the image into a parametric space, where the intensity of the pixels is a measure of the probability of having a circle centered at that position. Transforms for different radii are calculated into different alpha channels.
• ColorRange: This process creates a new binary image that reflects the selection of a cube or spherical selection around a user-defined "color". This selection may be defined in various color spaces, providing a lot of freedom and flexibility.
• LHT: LHT stands for Linear Hough Transform. This algorithm transforms the image into a parametric space, where the intensity of each pixel is a measure of the probability of having a straight line at the pixel's location. The X axis gives the inclination angle, and the Y axis the absolute minimal distance from the origin.
• ReadPSF: Very simple implementation of the StarStatistics class, to evaluate the PSF by fitting Gaussian functions to star shapes.
• Seed: More commonly known as "Magic Wand". This algorithm plants a seed and through local and global comparisons, determines a similar neighborhood.



Jupiter by Máximo Ruiz | 2010/09/19 - 02:30 UT | Meade LX 200 10" ACF | Imaging Source DMK 21A

IMAGE**GALLERY**

Van den Bergh 152 by Jordi Gallego | Total exposure time: 13.25 h (LRGB) | Takahashi TOA 150 | SBIG STL-11000M

NGC 1333 by Jordi Gallego | Total exposure time: 20 h (LRGB) | Takahashi TOA 150 | SBIG STL-11000M

NGC 6951 by Stephen Leshin
Total exposure time: 18.7 h (LRGB)
RCOS 14.5″ f/9 | SBIG STL-11000XM

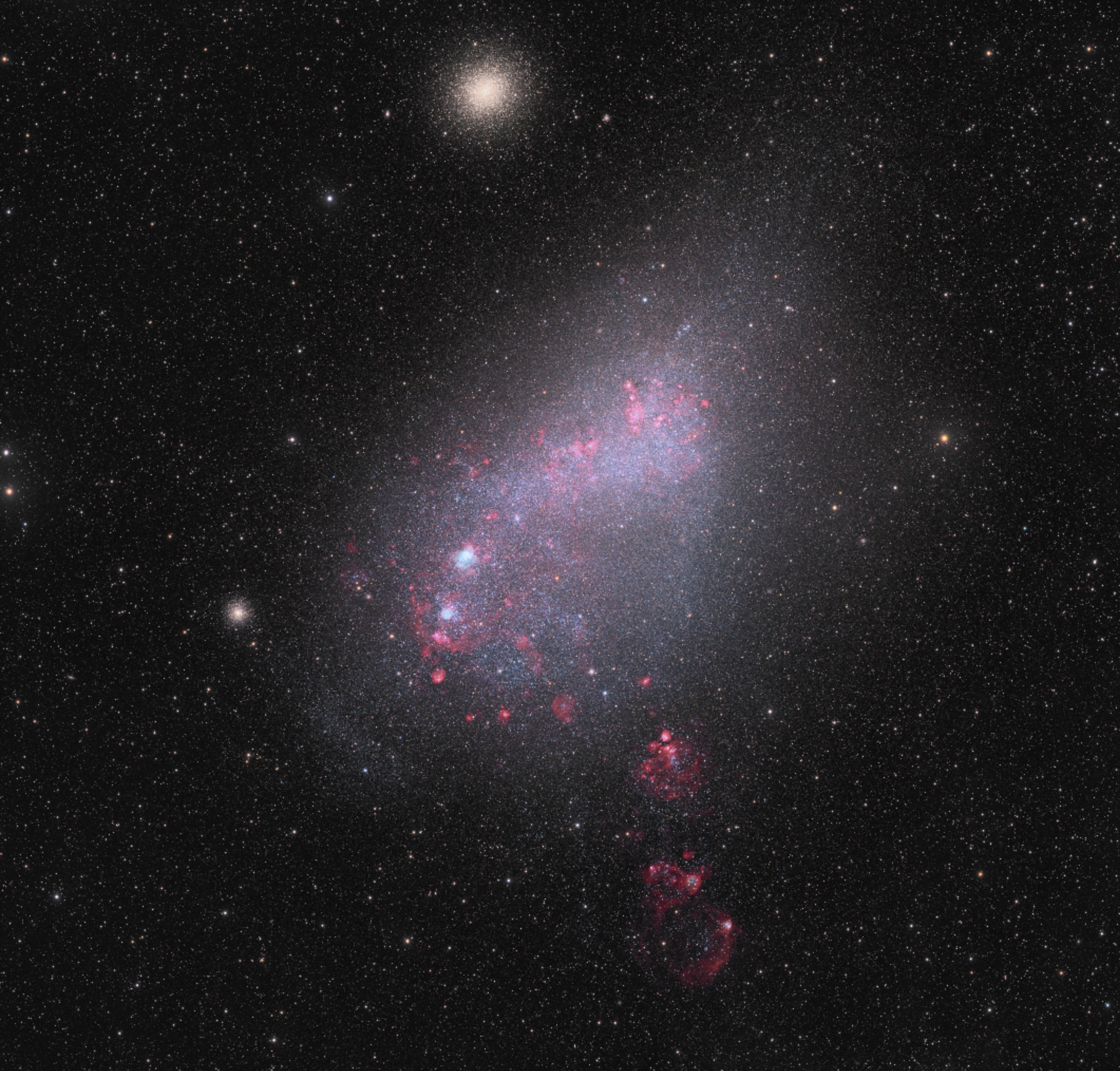
Van den Bergh 141 by Jordi Gallego
Total exposure time: 12.75 h (LRGB)
Takahashi TOA 150 | SBIG STL-11000M

NGC 772 by Stephen Leshin
Total exposure time: 16.67 h (LRGB)
RCOS 14.5″ f/9 | SBIG STL-11000XM

NGC 660 by Stephen Leshin
Total exposure time: 15 h (LRGB)
RCOS 14.5″ f/9 | SBIG STL-11000XM

Messier 51 — RGB by Vicent Peris (OAUV), Jack Harvey (SSRO), Steven Mazlin (SSRO), Juan Conejero (PixInsight) and Carlos Sonnenstein (Valkanik); CAHA, Descubre Foundation, OAUV, DSA. Total exposure time: 24 h (RGB) | Zeiss 1.23 m f/8 | SITe 2Kx2K, 24 um pixel CCD sensor

Messier 51 — HaRGB by Vicent Peris (OAUV), Jack Harvey (SSRO), Steven Mazlin (SSRO), Juan Conejero (PixInsight) and Carlos Sonnenstein (Valkanik); CAHA, Descubre Foundation, OAUV, DSA. Total exposure time: 35 h (HaRGB) | Zeiss 1.23 m f/8 | SITe 2Kx2K, 24 um pixel CCD sensor



Small Magellanic Cloud by Stanislav Volskiy
Total exposure time: 73 h (HaLRGB, 6 panel mosaic) | FSQ 106ED | SBIG STL-11000XM
Zoomed crop at right

Messier 31 by Jon Talbot
Continuum subtracted, narrowband H-alpha image
Total exposure time: 16 h (H-alpha & red cont.)
Stellarvue SV80ST2 | QSI 583

IC 1805 by Máximo Ruiz
Total exposure time: 15 h (HaRGB)
Takahashi FSQ 85ED | Cooled Canon 50D

Orion's Belt by Sergi Verdugo
Total exposure time: 6.67 h (RGB, two panel mosaic)
William Optics Megrez 88FD with flattener/reducer Borg DG-L | Amp-off Modded & Cooled Canon 350D

NGC 6914 by Vicent Peris (OAUV), Jack Harvey (SSRO), Juan Conejero (PixInsight); CAHA, Descubre Foundation, OAUV, DSA.
Total exposure time: 23 h (HaRGB, two panel mosaic) | Zeiss 1.23 m f/8 | SITe 2Kx2K, 24 um pixel CCD sensor