The Phission phSimpleVision Tutorial

The phSimpleVision Class:

The phSimpleVision class encapsulates a very simple vision framework using Phission components. The driving purpose behind this class is to handle many of the mundane details of setting up a vision system in Phission. This allows the phSimpleVision class to require little or no prior vision processing or Phission experience to allow a beginner student to get started with vision processing in as little time as possible.

Phission was initially built to provide the best performance possible for the given hardware on which it is being run. The design of an entire system consists of sources (capturing), sinks (displays) and sink-source (pipelines) objects. Setting up a vision system requires a good deal of setup code which not all users are interested in knowing right away. There is also a good deal of error checking and memory/class allocations to perform, too. The phSimpleVision class performs all these tasks to set up the more common example of a simple vision processing system.

phSimpleVision's internal design:

Internal to the class there exists one capture class that provides input to two display classes and one pipeline class. The pipeline class supplies input to two display classes, too. The capture class (derived from a phImageCapture class) opens a connection to a device capabale of capturing images and continually captures images to supply to the display and pipeline classes. The pipeline class (phPipeline) allows one to add a series of filter processing classes (derived from phFilter) that will be run on the images that are supplied from the capture device.

The two displays (derived from the phDisplayInterface class) provide the ability to view images either on a local machine or across the network. When one is trying out the phSimpleVision code on a desktop, it makes sense to use facilities which will provide the best display performance locally. For this reason, the X11Display is used to display images when one is running locally. If one desires to run the vision system on a remote machine, for example a robot, the best performance is acheived by compressing the image being displayed and then sending it across the network (as opposed to using the X protocol to remote display uncompressed images). For these reasons, two display classes are provided for each output within the phSimpleVision class: from the capture device and the pipeline class.

phSimpleVision examples:

There are currently 2 phSimpleVision examples. One for use with Pyro and a PTZ capable robot and the other is a C++ example to demonstrate the features of phSimpleVision. The examples are located in the Phission directory (/usr/local/phission/examples/ or /usr/phission/examples/)

phSimpleVisionTest:

...phission/examples/cpp/phSimpleVisionTest

cd to that directory and run 'make' if necessary. The executable will be placed into the bin/ directory.

You can optionally copy the phSimpleVisionTest directory to your home directory:
cp -pR ...phission/examples/cpp/phSimpleVisionTest ~

Copy the Makefile.example file that it depends on:
cp ...phission/examples/cpp/Makefile.example ~/phSimpleVisionTest

Edit the phSimpleVisionTest Makefile and change topdir to be "." instead of ".." so it'll look within its own directory to include Makefile.example.

StateBasedSimpleVision.py:

...phission/examples/pyro/brains

cd to that directory and run:
pyro -r Aria -b StateBasedSimpleVision.py

phSimpleVision capabilities:

phSimpleVision provides the more basic capabilities needed to configure one's system in three areas: capturing, displaying and pipeline setup.

Capture settings:

int setPath (const char *path);

This allows one to set the path to the image capture device. The default is "/dev/video0" on Linux systems and you only need to call this if your device isn't located at "/dev/video0".

const char *getPath ( );

Returns the setting for the image capture device path.

int setChannel (uint32_t channel);

Sets the channel on an image capture device such as a TV-Tuner card to a specific channel. Usually, with Video4Linux, the lower

channels 0,1,2,etc. are for composite and s-video inputs.

uint32_t getChannel ( );

Returns the setting for the image capture device channel.

Display settings:

API Note:

For all the following methods, 'which' refers to which output
the particular display is connected to. Valid values include:

phSimpleVisionOutput_Capture
phSimpleVisionOutput_Pipeline

The short forms:

phSVO_Capture
phSVO_Pipeline

Performance Note:

A very useful feature is the ability to enable and disable the
displays. A display is really only needed during debug or
experimentation time. When it comes time to cut down on the amount
of resources the system is using (to get better performance)
the displays can all be disabled right before the main loop
has been entered and all setup is done.

NetDisplay:

The NetDisplay functionality is disabled by default. You must
enable it with one of the NetDisplay control methods.

The NetDisplay's default port is 44444 for the Capture output and
44445 for the Pipeline output. There is a NetClientTest program
availible for monitoring the output from a NetDisplay class and
can be located in either "/usr/local" or "/usr" (depending upon installation):

phission/examples/cpp/NetClientTest/bin/NetClientTest

If the executable isn't already built, you can copy the directory
to your home directory using the following steps:

```
mkdir phission_examples
cd /usr/local/phission/examples/cpp
cp -pR NetClientTest ~/phission_examples
cp Makefile.example ~/phission_examples
```

The example requires the Makefile.example file to be one directory above (unless one wants to edit the Makefile in the NetClientTest directory to point to a different directory).

When the NetClientTest example is built, it can be executred as follows (from the NetClientTest directory):

```
./bin/NetClientTest 127.0.0.1 44444
```

NetDisplay control methods:

```
int enableNetDisplay ( int which );
```

This allows one to enable the NetDisplay class within the phSimpleVision class. All NetDisplay classes are disabled by defalut.

```
int disableNetDisplay ( int which );
```

Disable a NetDisplay instance to prevent it from using resources.

```
int isNetEnabled ( int which );
```

Returns whether a specific NetDisplay is enabled or not.

```
int setPort ( uint32_t port );
```

This sets the base port for the NetDisplays.
'port' will be used for the output of the capture class and
'port + 1' will be used for the output from the pipeline.

X11Display control methods:

```
int enableDisplay ( int which );
```

Enable a specific instance of the X11Display class.

```
int disableDisplay ( int which );
```

Disable a specific instance of the X11Display class to

prevent it from using resources.

int isDisplayEnabled (int which);

Returns whether a particular display is enabled or not.

phSimpleVision system control methods:

The vision processing system is not always running when an instance of the phSimpleVision class exists. Internal to all the Phission code exist many separate threads of execution to perform the many separate tasks required for image capture, processing and displaying. Before the system can do any processing it must be started and will then run concurrently with the normal application. Data and settings can be altered or retreived most the time while the threads are executing because careful care has been taken in implementing thread-safety within Phission.

phSimpleVision takes care of calling the methods that are responsible for starting up the Phission vision processing system in three methods:

int start ( );

This starts up the system. Any displays, capture classes (only one in this case) or pipelines (only one) will be started up and will begin processing frames immediately and updating the displays.

int pause ( );

This will close down the capture device and stop the pipeline. It also stops the live update thread within the displays but will leave the displays open in case the displayed images are useful for debugging purposes.

int stop ( );

Completely stop everything within the vision system.

int isStarted();

Returns whether the phSimpleVision system is started.
A non-zero value (1) is returned if the system is started.
Zero is returned if the system isn't started.

int isPaused( );

Returns whether the phSimpleVision system is paused.

A non-zero value (1) is returned if the system is paused.
Zero is returned if the system isn't paused.

int isStopped();

Returns whether the phSimpleVision system is stopped.
A non-zero value (1) is returned if the system is stopped.
Zero is returned if the system isn't stopped.

phSimpleVision Filtering control methods:

Singular filter methods:

These methods will run the respective filters by themselves within the pipeline. phSimpleVision handles all necessary pipeline accessing to remove the old filters and insert the new one.

int empty ( );

This runs the empty_Filter which performs no calculations on the incoming images.

int motion ( );

This runs the motion_Filter on the image to detect motion and output a black and white image. The white denotes motion and the black denotes no motion.

int canny ( );

This runs either the custom canny_Filter or the cv_canny_Filter. It executes the canny edge detection filter on the input image.

int sobel ( );

This runs the sobel_Filter sobel edge detection filter.

int gaussian ( );

This runs the gaussian3x3_Filter. It is a gaussian kernel of size 3 that is run over the image data to gaussian blur the image.

int mean ( );

This runs the mean filter with a kernel of size 7 within the pipeline.

int median ( );

This runs the median filter with a kernel of size 5 within the pipeline.

int inverse ( );

This runs the inverse_Filter in the pipeline.
This inverses all the values, which are in the range of 0..255, in the image. 0 becomes 255, 1 becomes 254, 2 becomes 253, etc.

int threshold ( );

Runs the threshold_Filter which thresholds some image channel at some value and outputs the thresholded image.
When running on RGB data, the results can be somewhat useless. The threshold_Filter is more useful when run in a pipeline along with the subtract_Filter to detect motion.

int add ( );

Runs the add_Filter which will add 3 sequential frames.

int subtract ( );

Runs the subtract_Filter which subtracts one previous frame from the current frame being input.

Histogramming - color training methods:

int trainingRGB ( );

Setup the pipeline with the following filters:
1.) gaussian
2.) convert(RGB) and
3.) histogram_Filter

The histogrammed (or trained) color/threshold data returned after a call to 'train' will be representative of the RGB colorspace.

Possible return values:
phSuccess

phFail

int trainingHSV ( );

Setup the pipeline with the following filters:
1.) gaussian
2.) convert(HSV) and
3.) histogram_Filter

The histogrammed (or trained) color/threshold data returned after a call to 'train' will be representative of the HSV colorspace.

Possible return values:
phSuccess
phFail

int train (int colorspace, int add_or_set);

The train method is responsible for retrieving the data from the histogram_Filter running within the pipeline. It will collect n histogram samples, store the color/threshold values and then average them in an attempt to remove any histogram noise that could result from a CCD camera.

'train()' will set up the pipeline for histogramming if the pipeline hasn't already been setup with a call to trainingHSV/RGB. If the system isn't running, it will be started.

The color and threshold values are only valid since the last call to the 'train' method and can be retrieved with the 'getColor()' and 'getThreshold()' methods.

Any other histogram data that you may want to get, for example the MaxBin color/threshold values, can be retreived by calling 'getHistogramData()' which returns a copy of a phHistogramData object. You shouldn't need to do this since the returned color and threshold data are likely to be more accurate and useful.

You can specify what color space to use and whether train() should take the averaged color and threshold values and immediately send them to the blob_Filter. A trained color can either be added after the other colors that have been previously trained on or it can replace all previously trained colors with the latest. The blob_Filter will not be run until 'track' is called, but the values in the blob_Filter will be ready immediately after the train call if either the add or set option is used.

The colorspace and add_or_set parameters are optional. You can call the method without them: "train();". The default values are marked below with *.

Valid values for colorspace are:
phSimpleVision_RGB
phSimpleVision_HSV (*)

Valid values for add_or_set are:
phSimpleVision_ADD
phSimpleVision_SET
Otherwise: do nothing (*)

Possible return values:
phSuccess
phFail

int trainSet (int colorspace = -1);

This will call train() with phSimpleVision_SET in place for the 'add_or_set' variable. This is useful for being immediately ready to make a call to 'track()' without alering any color/threshold values.

Valid values for colorspace are:
phSimpleVision_RGB
phSimpleVision_HSV (*)

Possible return values:
phSuccess
phFail

int trainAdd (int colorspace = -1);

This will call train() with phSimpleVision_ADD in place for the 'add_or_set' variable. This can be useful if one doesn't desire to alter the threshold values and attempt to match several different colors.

Be careful how often you call this method if you are training in a loop. You could potentially flood the blob_Filter with so many colors to match that it takes too long to run the algorithm for it to be useful for anything. It would be best to call 'train()' if you are using a loop for that simple reason and then call 'applyTraining' after the loop has ended.

Valid values for colorspace are:
phSimpleVision_RGB
phSimpleVision_HSV (*)

Possible return values:
phSuccess
phFail

phColor getColor ( );

This returns a phColor value for the trained color.
It is only valid as of the last invocation of "train()"

phColor getThreshold ( );

This returns a phColor value for the trained threshold.
It is only valid as of the last invocation of "train()"

int setColor (phColor color);

If one desires to put in a custom setting for the color, this method will set the trained color using the 'color' parameter's values. This doesn't apply the color to the blob_Filter.

This is also useful for tweaking values retrieved from 'getColor', where the pixel channel values (i.e. either R,G,B or H,S,V values) can be altered to some point to better match the color.

Possible return values:
phSuccess
phFail

int setThreshold (phColor threshold);

This sets the trained threshold value within the class. It doesn't apply it to the blob_Filter for tracking.

Generally, the threshold values returned from the training session are going to need to be tweaked. Especially in the HSV color space. This

Possible return values:
phSuccess
phFail

int applyTraining (int add_or_set);

This applies the trained color and threshold values to the blob_Filter. If the color or threshold values were not originally added or set (by passing phSimpleVision_SET/ADD to the train method or calling one of the convenience methods: trainSet/Add), the color/threshold values can be add or set using this method.

This is useful for constantly training and then only applying the values to blob_Filter when an input (variable or robot bumper) has been set/triggered to end the training loop.

Possible return values:
- phSuccess
- phFail

Blobbing - color tracking methods:

int trackingRGB ( );

Setup the pipeline with the following filters:
1.) gaussian
2.) convert(RGB) and
3.) blob_Filter

This method merely sets the pipeline up with the proper filters. If the pipeline is currently running, then the blob_Filter will begin blobbing the values it was given by one of the training methods.

Possible return values:
- phSuccess
- phFail

int trackingHSV ( );

Setup the pipeline with the following filters:
1.) gaussian
2.) convert(HSV) and
3.) blob_Filter

This method merely sets the pipeline up with the proper filters. If the pipeline is currently running, then the blob_Filter will begin blobbing the values it was given by one of the training methods.

Possible return values:
- phSuccess
- phFail

int track (int colorspace);

'track' is responsible for retreiving the lastest phBlobData.
Since track is generally to be used in a main sensing loop, This is
a non-blocking method that will return with a specific value should
no new blob data objects be available (returns phSimpleVision_NOUPDATE).
'getMaxBlob', 'getBlobCount' and 'getBlobData' will only current as of
the last call to this method.

This method will set up the pipeline for blobbing if it hasn't
already been done. If the system isn't running, it will be started.

The color space will specify what colorspace to blob on. This
may cause trouble if you trained on a different colorspace and are
trying to blob on another colorspace(i.e. trained on RGB and blobbing
on HSV). The blob_Filter is unlikely to allow this so don't use the
colorspace parameter unless you are sure what colorspace you should
be using.

Valid values for colorspace are:
phSimpleVision_RGB
phSimpleVision_HSV (*)

Possible return values:
phSimpleVision_NOUPDATE
There is no new blob ready. You can procede to loop
and process other sensors until a new blob shows up.

phSimpleVision_UPDATED
There is a new blob ready to be interpreted/analyzed.

uint32_t getBlobCount ( );
uint32_t getBlobCount (uint32_t minsize);

Returns the number of blobs given the minsize. phSimpleVision has
it's own private minsize that is used in the 'getBlobCount( )'
method that takes no parameter.

The value returned is only valid as of the last successful call
to the 'track' method.

phblob getMaxBlob ( );

Returns the most current max blob as of the most recent successful

call to the 'track' method.

If you desire to view more blobs than the maximum blob, you can retreive a phBlobData object by calling 'getBlobData' and calling 'getBlobData(n)' where 'n' is the index for the blob. This index will be less than the value that 'phBlobData::getBlobCount(0)' returns for a given instance.

int resetTrackingData ( );

This resets the phBlobData information and the blob_Filter information.

phSimpleVision internal object access methods:

The following methods will return pointers to instances of internal objects contained in the phSimpleVision class. Sometimes the control methods provided by phSimpleVision just aren't enough and one needs more control. These should only be used if you know how to use the class objects being returned.

The interfaces to the following objects can be found in header files where Phission is installed. This is usually either going to be "/usr/phission/include" or "/usr/local/phission/include". Until the rest of the documentation for these objects is completed, it is suggested you look into the header files. Most the methods are self explainatory and there may be some enlightening comments, if any at all.

blob_Filter *getBlobFilter ( );

A pointer to the blob_Filter class that does all the tracking functionality.

* The interface to blob_Filter can be found in 'blob_Filter.h'

phBlobData getBlobData ( );

More detailed information of blobs that are being blobbed can be retreived from the phBlobData object that this copies out of the blob_Filter instance and copies back to the caller.

The phBlobData is only valid as of the last successful call to 'track'.

* The interface to phBlobData can be found in 'blob_Filter.h'

histogram_Filter *getHistogramFilter ( );

Returns a pointer to the histogram_Filter class.

* The interface to histogram_Filter can be found in 'histogram_Filter.h'

phHistogramData getHistogramData ( );

Returns a class that contains more detailed information on the histogramming result from the training methods.

The phHistogramData is only valid as of the last call to 'train'.

* The interface to phHistogramData can be found in 'histogram_Filter.h'

phSystem *getSystem ( );

Returns a pointer to the phSystem contained within the phSimpleVision class. The phSystem class is responsible greatly simplifying the management of starting all the displays, pipelines and captures.

* The interface to phSystem can be found in phSystem.h

phImageCapture *getCapture ( );

Returns a pointer to the phImageCapture instance.
This class accesses the image capture hardware.

* The interface to phImageCapture can be found in 'phImageCapture.h' and 'phCaptureInterface.h'

phPipeline *getPipeline ( );

Returns a pointer to the phPipeline instance that processes all the images fed to it by running phFilter derived classes with the image as input to those phFilter classes.

* The interface to phPipeline can be found in 'phPipeline.h'

phDisplayInterface *getDisplay (int which);

Returns a generic pointer to a phDisplayInterface class that provides access to the X11Display that is being used internally. 'which' can be phSVO_Capture or phSVO_Pipeline.

* The interface to phDisplayInterface can be found in

'phDisplayInterface.h'

NetDisplay *getNetDisplay (int which);

Returns a pointer to the specific instance of the NetDisplay class.
'which' can be phSVO_Capture or phSVO_Pipeline.

* The interface to NetDisplay can be found in
'phDisplayInterface.h' and 'NetDisplay.h'

phSimpleVision example code:

The examples are locate where Phission is installed. This directory will usually be "/usr/phission/examples" or "/usr/local/phission/examples".

C++ examples:

examples/cpp/phSimpleVisionTest:

This demos all the singular filters when supplied the '--demo' parameter and will run the 'train' and 'track' filters otherwise. Pass it '--help' or see the source file for more command line switches.

Pyro examples:

examples/pyro/brains/StateBasedSimpleVision.py
examples/pyro/brains/StateBasedSimpleVision.sh :

This demonstrates a pan/tilt camera tracking a color that is trained on. When using Pyro, the command interface will accept 'self.dotrain = 1' to allow training without needing to press the bumper on the robot.

This also demonstrates the use of a State Based brain for color training tracking and searching.

You'll likely want to run this program as such:
"pyro -r Aria -b StateBasedSimpleVision.py"

examples/pyro/brains/phSimpleVisionDemo.py
examples/pyro/brains/phSimpleVisionDemo.sh :

The phSimpleVisionDemo example demonstrates all the singular filter methods of the phSimpleVision class. Closing the display that contains the output of the pipeline will cycle through the filters.

This is a very convenient demo because it has two displays side by side to show the original live image and the output image.

When using Pyro, the command interface will accept 'self.switchfilter = 1' to cycle through the filters, too.

examples/pyro/brains/phSimpleVisionTest.py
examples/pyro/brains/phSimpleVisionTest.sh :

The phSimpleVisionTest example demonstrates training and tracking functionality of the phSimpleVision class without the robotic requisites. Closing the Pipeline output display will toggle between the train and track methods.

When using Pyro, the command interface will accept 'self.switchfilter = 1' to toggle between train and track methods