Vision

This page is **stale** and needs to be reviewed. It may be deleted or radically changed in the near future.

Core Functionality

- The vision system on the sub is completely managed by the vision module. The current daemon implementation allows for the vision module to be spawned independently or as a child of the command module. Be sure to comment out communicator initialization code before running any unit tests as the vision system will wait on the broker.
- Cameras are managed under an umbrella class named Capture which includes flycapture and opencv cameras in one interface for convenience. Refer to the source code for specifics on how to use the class as it is a bit hacky in order to get the build to work on OSX which does not have flycap support.
- The vision system's main functionality is implemented through the creation and use of vision processes. The VisionProcess class inherits all of the process elements from base class Process and adds convenient vision specific functions for accessing camera framebuffers from shared memory among other things.
- Vision processes utilize the FilterTree class for all basic image filtering operations. This allows a dynamic insertion of new algorithm implementations into a vision process as desired. It also allows the vision GUI tool to access and change parameters for the vision pipeline in each process.

Writing a Filter

- The building blocks of the algorithms in the vision system utilize the filter. A filter can be thought of as an operation that inputs and outputs an image. Filters are connected together in a pipe and filter architecture with the FilterTree class.
- In order to ensure proper serialization and deserialization at runtime and when tweaking algorithms within the GUI, you need to make sure that OpenCV has all of its required wrapper functions implemented. You can look at other filter implementations as an example on how it should be structured. Refer to OpenCV's filestorage documentation for details.

Writing a Vision Process

- Vision processes should be thought of as standalone vision threads that accomplish a reusable vision functionality for the sub to utilize at the AI level. They can utilize multiple cameras and trees if desired, although one tree and one camera is most common.
- The first thing you should do is create a .hpp and .cpp file for the process you want to make. Since settings file and tree loading are automatic, make sure you have an identically named settings and tree file in the settings folder or the process will not load and will log the error.

- You must then include and inherit from vision process and implement each function as necessary. You can look at other processes as examples on how it should look. NOTE: Since filtering trees operate on filters and filters only return images as output, you will have to run algorithms that output something other than an image outside of the tree. (Ex. Finding the center of an object and returning it.) There are filters which output colored visuals for algorithms such as hough lines, circles, and histograms, but they are mainly for the GUI and visual verification. Runtime implementations should be outside of the filtertree in vision processes for things such as this.
- To spawn the vision process you just created, you can send the appropriate spawn command to the vision system along with the name of the process. Refer to the communicator documentation for specifics on commands.

Cameras

See the Cameras page for more information on the details and use of the cameras.

Running the Vision System in ROS

```
roslaunch robosub vision.launch
```

To remap left and right camera topics, append

```
leftImage:=[newTopic]
```

and/or

```
rightImage:=[newTopic]
```

(bottom camera to be implemented) the topics for the simulator are /camera/(left|right|bottom).

To use simulator color parameters, append

```
simulated:=true
```

(This feature could change in the near future) After this, you will see the launch file spin up multiple nodes. The vision system is running!

If you would like to see the images that the system is using, you can run the following command:

```
rosparam set /{vision node name}/processing/doImShow true
```

You will then see many windows open each with a unique image.