README for OpenNI Python Wrapper

Created by Gabriele Nataneli (gamix)

20 January 2011

RELEASE NOTES

V0.3 alpha

* The current version provides partial support for OpenNI image and depth generators only.
* The purpose of the current preliminary version is to:
  + make image and depth streams available to OpenCV’s Python binding and to the Python Imaging Library (PIL)
  + Serve as a starting point for other open source developers to add more functionality.
* I developed and tested the code only on Windows with VS2008. However, the code is fully platform independent and should be easily portable to other platforms supported by OpenNI.
* Refer to the script *TestBench/testOpenNIPythonWrapper.py* for several examples on how to use the current functionality

ADDITIONAL NOTES

* I provided a project for Visual Studio 2008 that should simplify the build process in: *OpenNIPythonWrapper/ProjectFiles/VS2008.* Make sure you have all the required dependencies and environment variables set correctly for the project to build (See the sections below).
* If you want to use another compiler or development environment you are on your own for now!
* I haven’t tested the project with Visual Studio Express 2010, but I am pretty sure that it should work fine. It’s free, so please go ahead and let me know how it goes.
* For Python development, consider using Eclipse with the excellent PyDev Add-In. For one thing it has a really powerful debugger for Python.
* The code should be clean enough for someone with Boost Python experience to contribute, but the current [prelinary] release doesn’t come with any formal documentation yet. Sorry!
* Be sure to check out the Python sample code in the *TestBench* sub-directory to learn how to use the binding.
* In Windows, you must make sure that the Python interpreter can find the dlls for all the dependencies, otherwise Python would simply say that the module was not found. In the sample code, I show how to modify the Windows PATH at run-time to configure all the paths correctly from Python, but it is typically easier to make sure that all the necessary paths are found in the PATH environment variable. If you think that your PATH is set correctly, you may want to disable the environment setup phase in the sample by setting FLAG\_SETUP\_ENVIRONMENT=False.
* As a bonus I provided another project of mine called *PythonOutputStream* that uses some Boost trickery to easily print to the Python console directly from the binding code. This code enables a stream called PyCout that works exactly as std::cout.
* Even though in the binding code you will find some support for OpenNI meta-data objects, that code is deprecated and should not be used. The binding provides additional methods that let you grab meta-data directly from the exposed generator nodes.

DEPENDENCIES

*Mandatory*

* Python 2.x
* Boost Python
* OpenNI with a working driver for your compliant device

*Optional*

* Python Imaging Library
* OpenCV Python Binding
* Visual Studio 2008/2010 to use the provided project

HOW TO BUILD

*Windows*

* Make sure that all dependencies are installed correctly
* Remember to build Boost Python against the version of Python that you are planning to use!
* Define the following environment variables for the provided VS2008 project:
  + **BOOST\_ROOT**: The path to the Boost libraries. Example: *C:\Program Files\boost\boost\_1\_44*
  + **OPENNI\_DIR:** The path to the OpenNI library. Example: *C:\Program Files\OpenNI*
  + **PYTHON\_DIR:** The path to your Python installation. Example:*C:\Python26*
  + Build the project.

*HOW TO READ THE CODE*

* Unless you want to contribute, you can safely ignore most of the code in the binding. The most important stuff is in the file *wrapper.cpp* in the section that starts with *BOOST\_PYTHON\_MODULE*. Even if you are not familiar with Boost Python, you can use the following rules of thumb to understand what is currently exposed:
  + Sections delimited by the *class\_* keyword are exposed classes.
  + Statements that start with *.value* are constants.
  + Statements that start with *.def* are methods.
* Consider also using the following to take a look at Python’s automatically generated help:

import OpenNIPythonWrapper as ONIPY

help( ONIPY )

DESIGN RATIONALE

While SWIG is a great tool for generating quick bindings for large libraries, it tends be a little slippery. Bindings created in SWIG are somewhat unreliable and lead to unpleasant crashes wherever developers don’t specify the details of how objects must be converted from C++ to Python appropriately. A good alternative is Boost Python, which is what I used here. Boost Python provides less automation than SWIG, but it gives developers more fine grained control and it typically refuses to build if the binding code is not implemented correctly.

I originally developed this binding only to expose a small subset of OpenNI’s functionality for a specific computer vision project I am working on, in which case the overhead of using Boost Python is perfectly acceptable. I hope that the code that I’ve written can become the backbone for a more complete binding to OpenNI and NITE with contributions from the fantastic open source community surrounding these projects.

TROUBLESHOOTING

*Visual Studio can’t find the environment variables, but you just set them!*

Remember to either log off or restart your system for changes in the environment variables to take effect. I know that this is obvious, but people often forget :-)

*You get ImportError: DLL load failed: The specified module could not be found when you import the binding or run the sample code.*

* Make sure that the actual binding binary (e.g. in Windows is the pyd file that you built) is found by the Python interpreter at run-time.
* Make sure that all the dependencies (e.g. dlls) of the binding are found by the Python interpreter at run-time. You can discover the by yourself in Windows by using Dependency Walker. The key dependencies for version 0.3 alpha are:
  + boost\_python-vcxx-mt-gd-1\_xx.dll [Debug]
  + boost\_python-vcxx-mt-1\_xx.dll [Release]
  + openni.dll

*When you run the Visual Studio project (e.g. by pressing F5), A dialog pops up asking to “Please specify the name of the executable file to be used for the debug session.”*

I provided a *.user* file that should instruct Visual Studio to invoke your Python interpreter and run the test sample, but if you misplaced that file then Visual Studio will revert to the default debugging options, which don’t work for this project.

*I modified the code, but Python does not seem to have noticed.*

I set up the Visual Studio project, so that it copies the binding to the *TestBench* directory, where the sample code is located. However, in some circumstances, Visual Studio may not perform the copy (e.g. post-build events) during an incremental build. To resolve this either copy the pyd file manually from the Debug/Release directory to the *TestBench* directory, or perform a full rebuild.