SIFT Algorithm in C++

Intro

This is a C++ implementation of the SIFT algorithm, which was originally presented by David G. Lowe in the International Journal of Computer Vision 60 in January 2004. This algorithm is mostly implemented after the principles described in Lowe's paper. Also some elements were taken from the lecture of Dr. Mubarak Shah, which was held at the University of Central Florida.

Installation

Requirements

- Vigra: A generic C++ library for image analysis(used for most of the calculations and image transformations)
- OpenCV: Open Source Computer Vision library(used for visualisation of the found sift features)
- Boost program_options: An easy to use layer for handling program arguments. Part of the Boost Library.
- CMake: A cross-platform open-source make system.

Installation Process

With the needed libraries installed and the help of CMake, the installation is as easy as three commands. First download this repository by cloning or whatever way you prefer. Switch into the directory afterwards. Now make an extra directory for your buildfiles. For Example:

```
mkdir build
```

Switch into the created directory

```
cd build
```

Then create your preferred make files with CMake. On all Unix like systems this will be GNU Makefiles.

```
cmake -G "Unix Makefiles" ..
```

For other supported build systems check the official documentation of CMake.
The final step is to build the executable from the Makefiles

```
make
```

There should now be an executable named sift in the build directory. Please refer to the next section to check how it is used and which possibilities you have, by executing it.

User Guide

The easiest way to start of is just giving an image and get a new image back, with the sift features drawn on it. The file is called `[file]_features.png`
and can be found in the same directory as the original file. The command to produce this file is
`./sift path/to/file.type`
But there are many possible parameters on which you can screw the values. The following list shows the possibilities

```
--help                          Print help messages
-i [ --img ] arg                The image on which sift will be executed
-s [ --sigma ] arg (=1.60000002) The sigma value of the Gaussian calculations
-k [ --k ] arg (=1.41421354)    The constant which is calculated on sigma
                                for the DoGs
-o [ --octaves ] arg (=4)       How many octaves should be calculated
-d [ --dogsPerEpoch ] arg (=3)  How many DoGs should be created per epoch
-p [ --subpixel ] arg (=0)      Starts with the doubled size of initial
                                image
-r [ --result ] arg (=0)        Print the resulting InterestPoints in a file
```

This overview can also be called by
`./sift --help`
The first flag is the shorthand and can generally be written by
`./sift path/to/file.jpg -[shorthand] [value]`
The second value inside the square brackets is the longhand and may be more readable, but it's also more typing. The general theme looks like this:
`./sift path/to/file.jpg --[longhand]=[value]`
The following argument is the default value, which is set, if no argument is given by the user. These default values are based on the studies of David Lowe's paper and seem to give the most stable results overall. The following chapters cover every flag in detail.

-i [--img] arg

The flag is the only non-optional as seen in the first example of the user guide. This is also the only possible option which can be handled without a flag. It takes an image file(tested: jpg, png) in RGB or Greyscale. Currently the algorithm gets slower by the size of the image dramatically. A small list with different image sizes shows the calculation time on my PC(i7 vPro) - ~300x300 px: ~0.7 seconds - ~600x600 px: ~15 seconds - ~1500x1500 px: ~11 minutes

-s [--sigma] arg (=1.60000002)

sigma is the standard deviation of the Gaussian curve. It is used extensively throughout the algorithm. For example when creating the Difference of Gaussian(DoG) pyramid.

-k [--k] arg (=1.41421354)

k is the constant, which is calculated onto sigma in each step of the Gaussian creation process. For example the process in the first octave of the algorithm looks like the following:

```
image 1: sigma
image 2: sigma * k ^ 1 = sigma * k
image 3: sigma * k ^ 2 = ...
```

-o [--octaves] arg (=4)

The count of octaves to be calculated for the DoG pyramid.

-d [--dogsPerEpoch] arg (=3)

The count of DoGs created per epoch.
The minimum is 3, because we need at least one DoG with an upper and lower neighbour to do further calculation steps of the algorithm.

-p [--subpixel] arg (=0)

Sets the subpixel flag to on(1) or off(0). If it is set to on, the algorithm works on subpixel accuracy. This is accomplished through doubling the size of the initial image and blur it with a Gaussian window with sigma=1.0 afterwards. Like mentioned in the paper, a base sigma of 0.5 is assumed in the original image. Every further calculation is based on the doubled version. If this flag is set to off, the algorithm starts with the initial image.

-r [--result] arg (=0)

Writes a sift.txt with a table like listing of all found interest points. The listed data are: positions, scale, orientation and their descriptors.

API

A full Class and Namespace Reference can be found here