/\*\*

\* OVERVIEW:

\* Filter which converts each tile to IplImage .

\*

\* Makes use of the ossim framework and opencv libraries.

\* The main operation starts with the getTile function.

\* Ossim splits up the image into tiles and for each tile,

\* the getTile function is called.

\*

\*/

// Include Blob Functions

#include "blob.h"

#include "BlobResult.h"

#include "ossimTileToIplFilter.h"

#include <ossim/base/ossimRefPtr.h>

#include <ossim/imaging/ossimU8ImageData.h>

#include <ossim/base/ossimConstants.h>

#include <ossim/base/ossimCommon.h>

#include <ossim/base/ossimKeywordlist.h>

#include <ossim/base/ossimKeywordNames.h>

#include <ossim/imaging/ossimImageSourceFactoryBase.h>

#include <ossim/imaging/ossimImageSourceFactoryRegistry.h>

#include <ossim/imaging/ossimImageChain.h>

#include <ossim/imaging/ossimImageDataFactory.h>

#include <ossim/base/ossimRefPtr.h>

#include <ossim/projection/ossimSensorModelTuple.h>

#include <ossim/projection/ossimImageViewTransform.h>

#include <ossim/projection/ossimImageViewProjectionTransform.h>

#include <ossim/projection/ossimMgrs.h>

#include <ossim/projection/ossimProjection.h>

#include <ossim/support\_data/ossimNitfFileHeaderV2\_0.h>

#include <ossim/support\_data/ossimNitfFileHeaderV2\_1.h>

#include <ossim/support\_data/ossimNitfFileHeaderV2\_X.h>

#include <ossim/support\_data/ossimNitfImageHeader.h>

#include <ossim/support\_data/ossimNitfImageHeaderV2\_0.h>

#include <ossim/support\_data/ossimNitfImageHeaderV2\_1.h>

#include <ossim/support\_data/ossimNitfRpcBase.h>

#include <ossim/support\_data/ossimNitfPiaimcTag.h>

#include <opencv/highgui.h>

#include <ossim/imaging/ossimImageSourceSequencer.h>

#include <time.h>

#include <map>

#include <stdio.h>

#include <vector>

#include <time.h>

#include <math.h>

RTTI\_DEF1(ossimTileToIplFilter, "ossimTileToIplFilter", ossimImageCombiner)

ossimTileToIplFilter::ossimTileToIplFilter(ossimObject\* owner)

: ossimImageCombiner(owner,

1,

0,

true,

false),

theTile(NULL)

{

xScale = 0;

yScale = 0;

numberOfImageInputs = 1;

}

ossimTileToIplFilter::~ossimTileToIplFilter()

{

}

void ossimTileToIplFilter::initialize()

{

ossimImageCombiner::initialize();

if(getInput(0))

{

// Force an allocate on the next getTile.

theTile = NULL;

}

// theTile = NULL;

//

// if(!isSourceEnabled())

// {

// return;

// }

//

// theTile = ossimImageDataFactory::instance()->create(this, this);

// if(theTile.valid())

// {

// theTile->initialize();

// }

}

void ossimTileToIplFilter::setScale(double scaleX, double scaleY)

{

xScale = scaleX;

yScale = scaleY;

}

ossimScalarType ossimTileToIplFilter::getOutputScalarType() const

{

if(!isSourceEnabled())

{

return ossimImageCombiner::getOutputScalarType();

}

return OSSIM\_UCHAR;

}

ossim\_uint32 ossimTileToIplFilter::getNumberOfOutputBands() const

{

if(!isSourceEnabled())

{

return ossimImageCombiner::getNumberOfOutputBands();

}

return 1;

}

bool ossimTileToIplFilter::saveState(ossimKeywordlist& kwl,

const char\* prefix)const

{

ossimImageCombiner::saveState(kwl, prefix);

return true;

}

bool ossimTileToIplFilter::loadState(const ossimKeywordlist& kwl,

const char\* prefix)

{

ossimImageCombiner::loadState(kwl, prefix);

return true;

}

double ossimTileToIplFilter::round(double x)

{

return floor(x + 0.5);

}

ossimRefPtr<ossimImageData> ossimTileToIplFilter::getTile(const ossimIrect& tileRect,

ossim\_uint32 resLevel)

{

cout << "Getting Tile !" << endl;

// Check input data sources for valid and null tiles

ossimImageSource \*imageSource = PTR\_CAST(ossimImageSource, getInput(0));

ossimRefPtr<ossimImageData> imageSourceData;

if (imageSource)

imageSourceData = imageSource->getTile(tileRect, resLevel);

if (!isSourceEnabled())

return imageSourceData;

if (!theTile.valid())

{

if(getInput(0))

{

theTile = ossimImageDataFactory::instance()->create(this, this);

theTile->initialize();

}

}

if (!imageSourceData.valid() || !theTile.valid())

return ossimRefPtr<ossimImageData>();

theTile->setOrigin(tileRect.ul());

if (theTile->getImageRectangle() != tileRect)

{

theTile->setImageRectangle(tileRect);

theTile->initialize();

}

IplImage \*input = cvCreateImage(cvSize(tileRect.width(), tileRect.height()),IPL\_DEPTH\_8U,3);

IplImage \*output = cvCreateImage(cvSize(tileRect.width(),tileRect.height()),IPL\_DEPTH\_8U,3);

cvZero(input);

cvZero(output);

// If 16 or 32 bits, downsample to 8 bits

ossimScalarType inputType = imageSourceData->getScalarType();

if(inputType == OSSIM\_UINT16 || inputType == OSSIM\_USHORT11)

CopyTileToIplImage(static\_cast<ossim\_uint16>(0), imageSourceData, input, tileRect);

else

CopyTileToIplImage(static\_cast<ossim\_uint8>(0), imageSourceData, input, tileRect);

cvCopy(input, output);

int bins = 256;

int hsize[] = {bins};

float binVal;

float sum=0;

int firstIndexFlag = 1;

/\*// Create histogram of image

CvHistogram \*hist;

hist = cvCreateHist(1, hsize, CV\_HIST\_ARRAY, 0, 1);

cvCalcHist(&input, hist, 0, 0);

cvNormalizeHist(hist, 100);

binVal = cvQueryHistValue\_1D(hist,1);

\*/

// Determine the actual height and width of each tile

ossimIrect fullImageRect;

fullImageRect = imageSource->getBoundingRect(0);

ossim\_int32 tileHeight, tileWidth, imageWidth, imageHeight;

tileHeight = tileRect.height();

tileWidth = tileRect.width();

imageWidth = fullImageRect.width();

imageHeight = fullImageRect.height();

ossim\_int32 totRows, totCols;

totRows = (ossim\_uint32)round(imageHeight / tileHeight);

totCols = (ossim\_uint32)round(imageWidth / tileWidth);

ossimIpt upperLeftTile = tileRect.ul();

if ((upperLeftTile.x + 1) > fullImageRect.ul().x + totCols \* tileWidth)

tileWidth = imageWidth - totCols \* tileWidth;

if ((upperLeftTile.y + 1) > fullImageRect.ul().y + totRows \* tileHeight)

tileHeight = imageHeight - totRows \* tileHeight;

//Begin Ship Detect Algorithim

//create a 1 channel image for grayscale image

//this is the same size as the input

IplImage\* grayImage =

cvCreateImage(

cvSize(input->width, input->height),

IPL\_DEPTH\_8U,

1);

//use gray image

//convert input image to grayscale image

cvCvtColor(input, grayImage, CV\_BGR2GRAY);

cvThreshold(grayImage, grayImage, 100, 255, CV\_THRESH\_BINARY);

//at this point we have a binary image of our detections

//so now we want to detect the blobs in the image

//these blobs are our targets

//perform a closing operatoin

cvDilate(grayImage, grayImage); //local max

cvErode(grayImage, grayImage); //local min

//now grayImage is "closed"

// Declare blob results

CBlobResult blobs;

// delare a single blob

CBlob Blob;

// get the blobs from the image, with no mask, using a threshold of 100

blobs = CBlobResult(grayImage, NULL, 100, true );

for(int index = 0; index <blobs.GetNumBlobs(); index++){

Blob = blobs.GetBlob(index);

cout << "Blob.Area(" << index << ")=" << Blob.Area() <<endl;

}

// Create sub-image to ignore zeros created by OSSIM

// ie, the tile is 512x512 but on the edges, the information is only in 512x10

CvRect subRect = cvRect(0, 0, tileWidth, tileHeight);

IplImage \*subImg = cvCreateImage(cvSize(tileWidth, tileHeight),IPL\_DEPTH\_8U,3);

cvSetImageROI(input, subRect);

cvCopy(input, subImg);

cvResetImageROI(input);

showImage(subImg,grayImage);

cvReleaseImage(&input);

cvReleaseImage(&output);

cvReleaseImage(&grayImage);

//don't use grayImage here or after

return theTile;

}

template<class T>

void ossimTileToIplFilter::CopyTileToIplImage(T dummyVariable, ossimRefPtr<ossimImageData> inputTile, IplImage \*output, ossimIrect neighborhoodRect)

{

ossimDataObjectStatus status = inputTile->getDataObjectStatus();

uchar \*outputData = (uchar \*)output->imageData;

int outputStep = output->widthStep/sizeof(uchar);

int outputChannels = output->nChannels;

ossimScalarType inputType = inputTile->getScalarType();

double scFactor;

if (inputType == OSSIM\_UINT16)

scFactor = 0.0039; // 255 / 65535

else if (inputType == OSSIM\_USHORT11)

scFactor = 0.1246; //255 / 2047

else if (inputType == OSSIM\_UINT8)

scFactor = 1;

else

scFactor = 1;

int pixVal;

if (status == OSSIM\_PARTIAL)

{

for( int band = 0; band < outputChannels; band++){

T\* inBuf = static\_cast<T\*>(inputTile->getBuf(band));

for (long y = 0; y < output->height; ++y)

{

for (long x = 0; x < output->width; ++x)

{

pixVal = (int)(\*inBuf);

if ((int)round(pixVal \* scFactor) > 255)

outputData[y \* outputStep + x\*outputChannels + band] = 255;

else if ((int)round(pixVal \* scFactor) < 0)

outputData[y \* outputStep + x\*outputChannels + band] = 0;

else

outputData[y \* outputStep + x\*outputChannels + band] = (uchar)round(pixVal \* scFactor);

++inBuf;

}

}

}

}

else

{

for(int band = 0; band < outputChannels; band++){

T\* inBuf = static\_cast<T\*>(inputTile->getBuf(band));

for (int y = 0; y < output->height; ++y)

{

for (int x = 0; x < output->width; ++x)

{

pixVal = (int)(\*inBuf);

if ((int)round(pixVal \* scFactor) > 255)

outputData[y \* outputStep + x\*outputChannels + band] = 255;

else if ((int)round(pixVal \* scFactor) < 0)

outputData[y \* outputStep + x\*outputChannels + band] = 0;

else

outputData[y \* outputStep + x\*outputChannels + band] = (uchar)round(pixVal \* scFactor);

++inBuf;

}

}

}

}

}

template<class T>

void ossimTileToIplFilter::CopyIplImageToTile(T dummyVariable, ossimRefPtr<ossimImageData> inputTile, IplImage \*output)

{

// Determine if tile is full or partially filled

ossimDataObjectStatus status = inputTile->getDataObjectStatus();

uchar \*outputData = (uchar \*)output->imageData;

int outputStep = output->widthStep/sizeof(uchar);

int pixVal;

long outputOffset = 0;

T maxPix = static\_cast<T>(getMaxPixelValue(0));

T minPix = static\_cast<T>(getMinPixelValue(0));

T np = static\_cast<T>(inputTile->getNullPix(0));

if (status == OSSIM\_PARTIAL)

{

for (ossim\_uint32 bandIdx = 0; bandIdx < inputTile->getNumberOfBands(); ++bandIdx)

{

T\* outBuf = static\_cast<T\*>(inputTile->getBuf(bandIdx));

if (outBuf)

{

outputOffset = 0;

for (long y = 0; y < output->height; ++y)

{

for (long x = 0; x < output->width; ++x)

{

if (!inputTile->isNull(outputOffset))

{

pixVal = (int)round(outputData[y \* outputStep + x]);

if (pixVal > maxPix)

\*outBuf = maxPix;

else if (pixVal < 0)

\*outBuf = minPix;

else

\*outBuf = static\_cast<T>(pixVal);

}

else

inputTile->setNull(outputOffset);

++outBuf;

++outputOffset;

}

}

}

}

}

else

{

for (ossim\_uint32 bandIdx = 0; bandIdx < inputTile->getNumberOfBands(); ++bandIdx)

{

T\* outBuf = (T\*)(inputTile->getBuf(bandIdx));

if (outBuf)

{

for (int y = 0; y < output->height; ++y)

{

for (int x = 0; x < output->width; ++x)

{

pixVal = (int)round(outputData[y \* outputStep + x]);

if (pixVal > maxPix)

\*outBuf = (maxPix);

else if (pixVal < 0)

\*outBuf = (minPix);

else

\*outBuf = static\_cast<T>(pixVal);

// Increment the output buffer to the next pixel value

++outBuf;

}

}

}

else

\*outBuf = np;

}

}

}

void ossimTileToIplFilter::showImage(IplImage\* src1, IplImage\* src2)

{

if (src1==NULL || src2==NULL)

cout << "ERROR ERROR \_\_\_\_\_ERROR " << endl;

// create window

cvNamedWindow("image1", CV\_WINDOW\_AUTOSIZE);

cvMoveWindow("image1", 100, 100);

cvNamedWindow("image2", CV\_WINDOW\_AUTOSIZE);

cvMoveWindow("image2", 600, 100);

// show the image

cvShowImage("image1", src1);

cvShowImage("image2", src2);

// wait for a key

int userKey = cvWaitKey(0);

if(userKey == 81 || userKey==113){ //exit if user presses q or Q

exit(1);

}

cvDestroyWindow("image1");

cvDestroyWindow("image2");

}

int ossimTileToIplFilter::GetTotalNumberTiles()

{

return totalTiles;

}