8

Detecting and Matching Interest Points

In this chapter, we will cover:

- Detecting Harris corners
- Detecting FAST features
- Detecting the scale-invariant SURF features
- Describing SURF features


After warping

Computing a homography between two images

$$\begin{bmatrix} sx' \\ sy' \\ s \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

We need several things related to this.

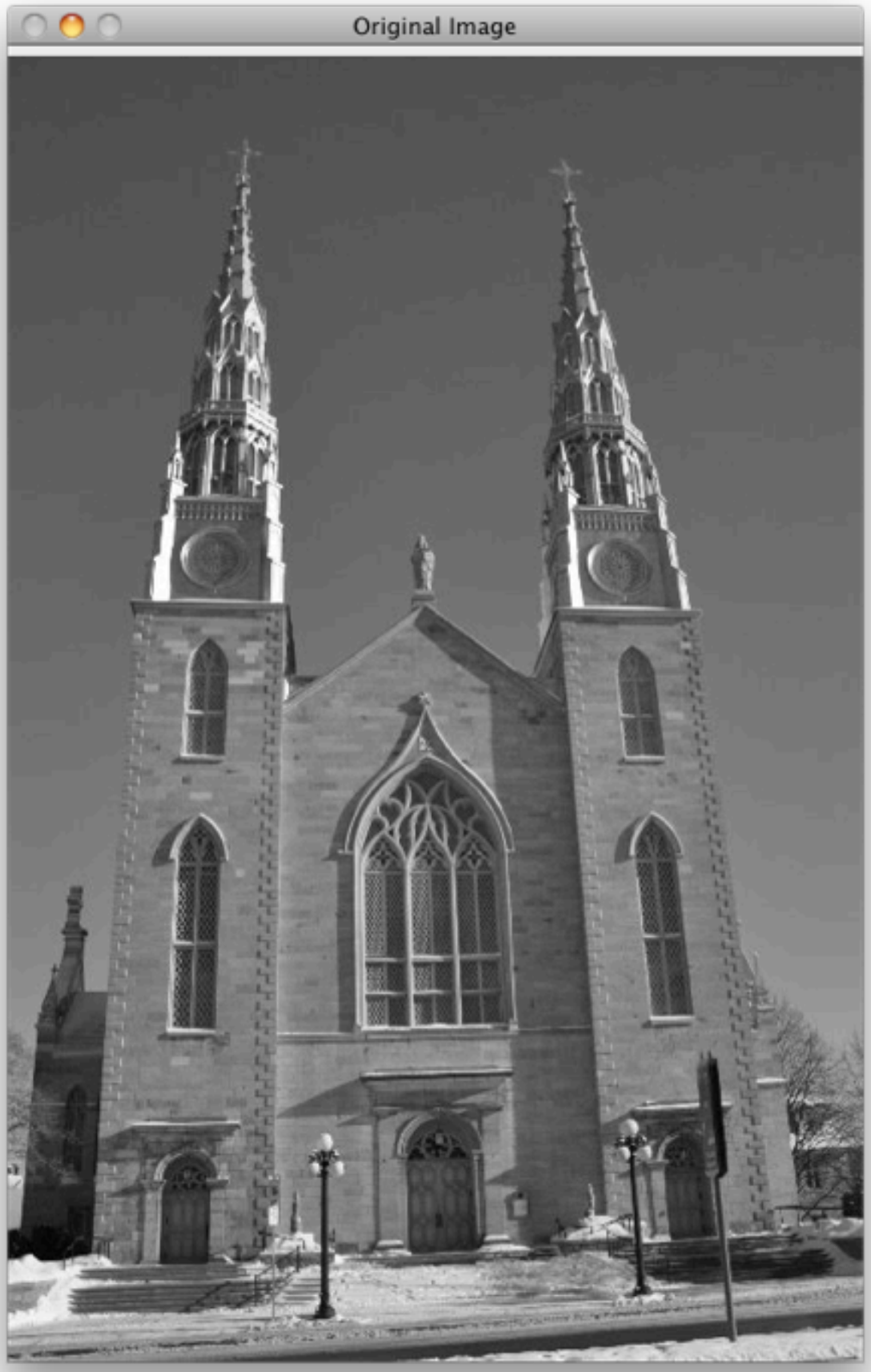
1. Projective transformation in 2D
2. How to compute it given correspondences (x, x')
3. How to find those correspondences?
4. How to build up the panorama image?

interest point, key point, feature point

It should

- be defined locally
- have something special compared to others
- be robust to some intensity variations such as illumination change

```
// Read input image
cv::Mat image= cv::imread("../images/church01.jpg",0);
if (!image.data)
    return 0;

 // Display the image
cv::namedWindow("Original Image");
cv::imshow("Original Image",image);
```



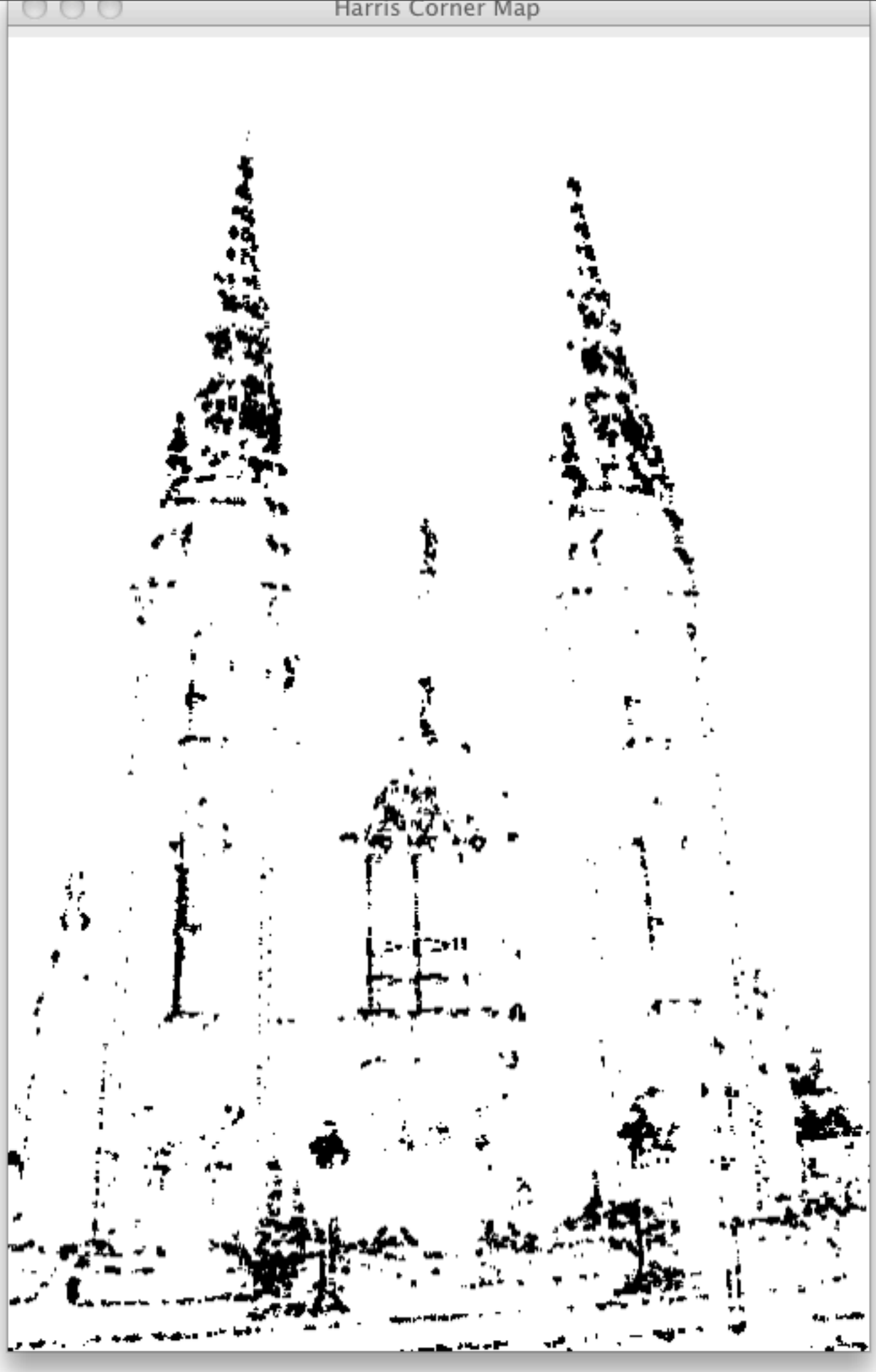
Interest Point Detectors

- Harris Detector
- Good Features to Track
- FAST
- SURF
- SIFT
- MSER

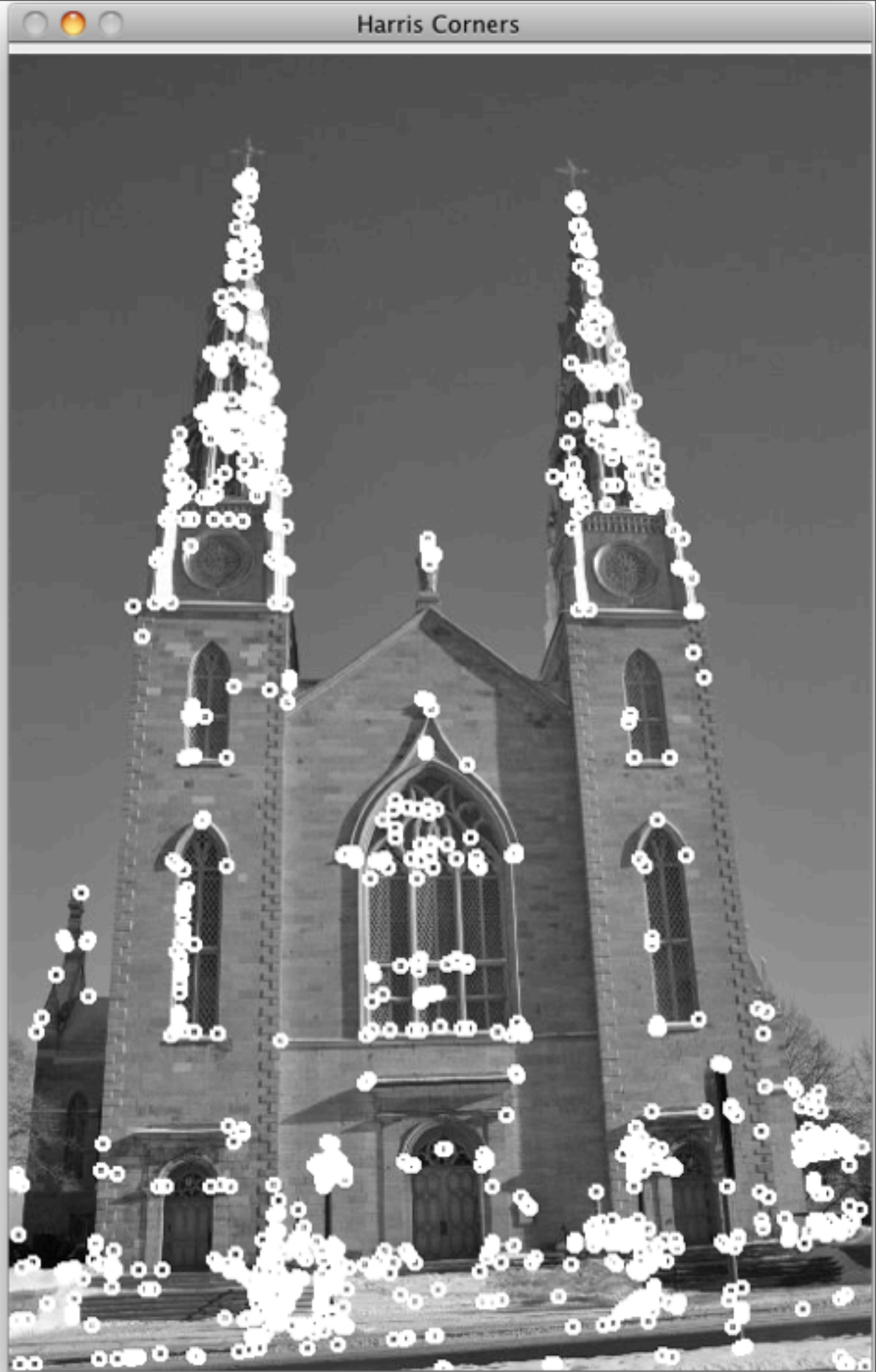
Scale Invariant: SURF, SIFT, MSER

Can be used for Feature Detection/Recognition

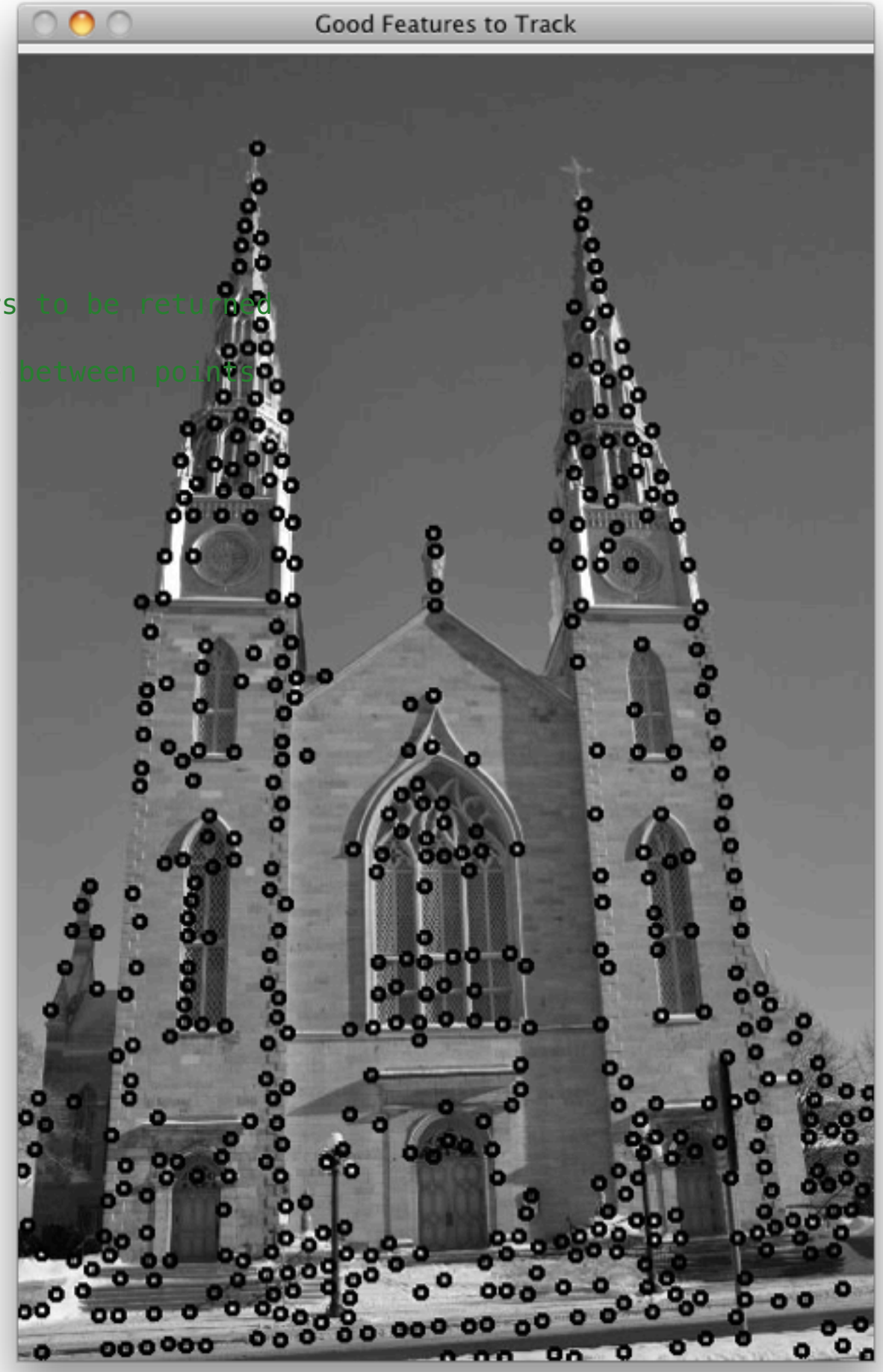
```
// Detect Harris Corners
cv::Mat cornerStrength;
cv::cornerHarris( image,cornerStrength,
                  3,     // neighborhood size
                  3,     // aperture size
                  0.01); // Harris parameter

// threshold the corner strengths
cv::Mat harrisCorners;
double threshold= 0.0001;
cv::threshold(cornerStrength, harrisCorners,
              threshold, 255,cv::THRESH_BINARY_INV);

// Display the corners
cv::namedWindow("Harris Corner Map");
cv::imshow("Harris Corner Map",harrisCorners);

cv::waitKey ();
```



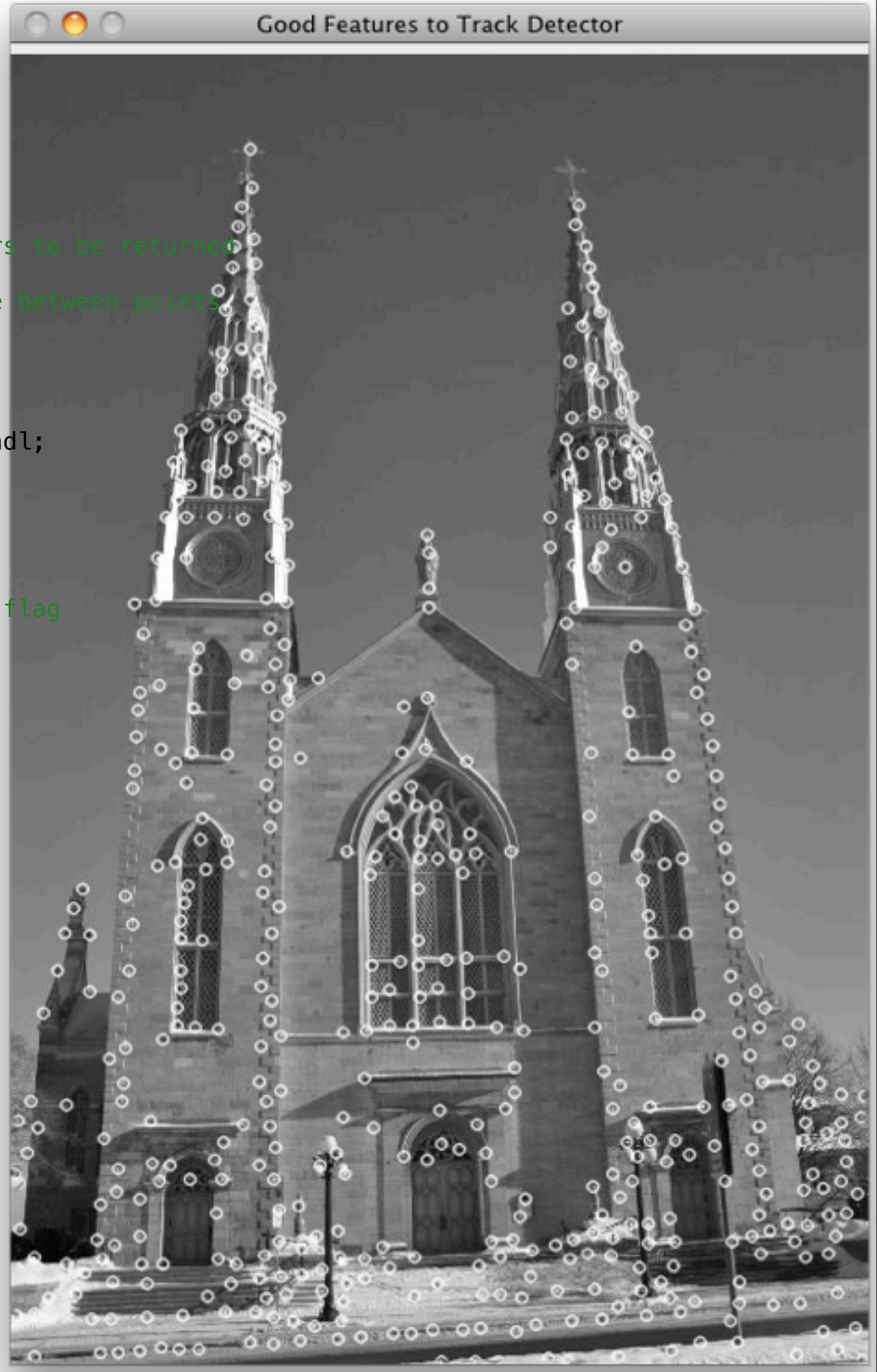
```
// Create Harris detector instance
HarrisDetector harris;
 // Compute Harris values
harris.detect(image);
 // Detect Harris corners
std::vector<cv::Point> pts;
harris.getCorners(pts,0.01);
// Draw Harris corners
harris.drawOnImage(image,pts);

 // Display the corners
cv::namedWindow("Harris Corners");
cv::imshow("Harris Corners",image);

 cv::waitKey ();
```



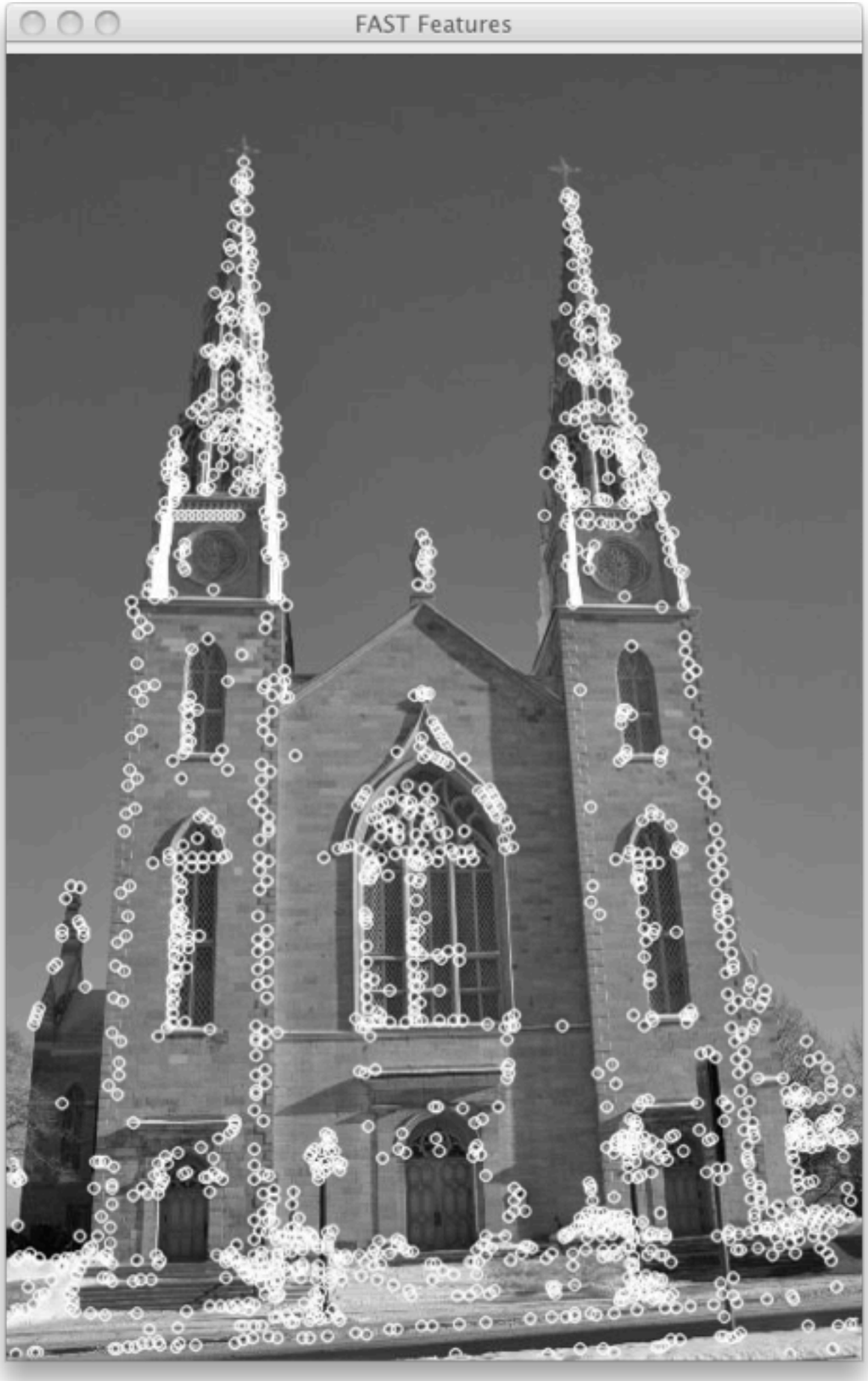
```
image= cv::imread("../images/church01.jpg",0);
if (!image.data) return 0;

// Compute good features to track
std::vector<cv::Point2f> corners;

cv::goodFeaturesToTrack(image,corners,
                        500,    // maximum number of corners to be returned
                        0.01,   // quality level
                        10);    // minimum allowed distance between points

 std::cerr << "gftt.corners() size()= "
           << corners.size()
           << std::endl;

// for all corners
std::vector<cv::Point2f>::iterator it= corners.begin();
while (it!=corners.end()) {
     // draw a circle at each corner location
     cv::circle(image,*it,3,cv::Scalar(5,255,5),2);
     ++it;
}

 // Display the corners
cv::namedWindow("Good Features to Track");
cv::imshow("Good Features to Track",image);

 cv::waitKey ();
```



```
// Read input image
image= cv::imread("../images/church01.jpg",0);

// vector of keypoints
std::vector<cv::KeyPoint> keypoints;
// Construction of the Good Feature to Track detector
cv::GoodFeaturesToTrackDetector gftt(
                                     500, // maximum number of corners to be returned
                                     0.01,     // quality level
                                     10); // minimum allowed distance between points

// point detection using FeatureDetector method
gftt.detect(image,keypoints);

std::cerr<< "GFTT Features Detected: " << keypoints.size() << std::endl;

cv::drawKeypoints(image,         // original image
                  keypoints,     // vector of keypoints
                  image,         // the resulting image
                  cv::Scalar(255,255,5),  // color of the points
                  cv::DrawMatchesFlags::DRAW_OVER_OUTIMG); //drawing flag

// Display the corners
cv::namedWindow("Good Features to Track Detector");
cv::imshow("Good Features to Track Detector",image);

cv::waitKey ();
```

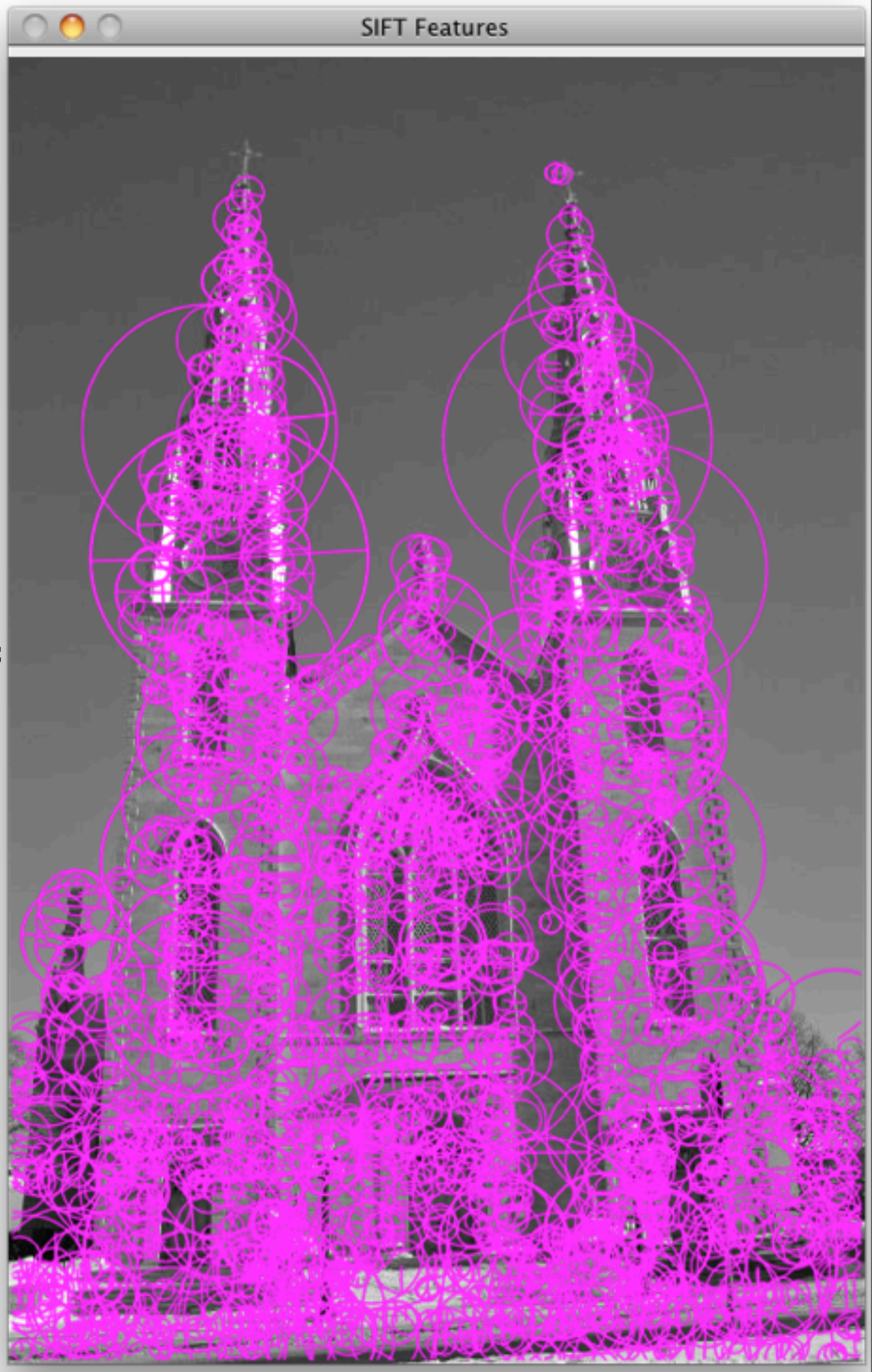


```
// vector of keypoints
std::vector<cv::KeyPoint> keypoints;

 // Read input image
image= cv::imread("../images/church01.jpg",0);

keypoints.clear();
cv::FastFeatureDetector fast(40);

fast.detect(image,keypoints);

 std::cerr << "FAST Features Detected: "
           << keypoints.size() << std::endl;

cv::drawKeypoints(image,keypoints,image,
                  cv::Scalar(255,5,255),
                  cv::DrawMatchesFlags::DRAW_OVER_OUTIMG);

 // Display the corners
cv::namedWindow("FAST Features");
cv::imshow("FAST Features",image);

 cv::waitKey ();
```



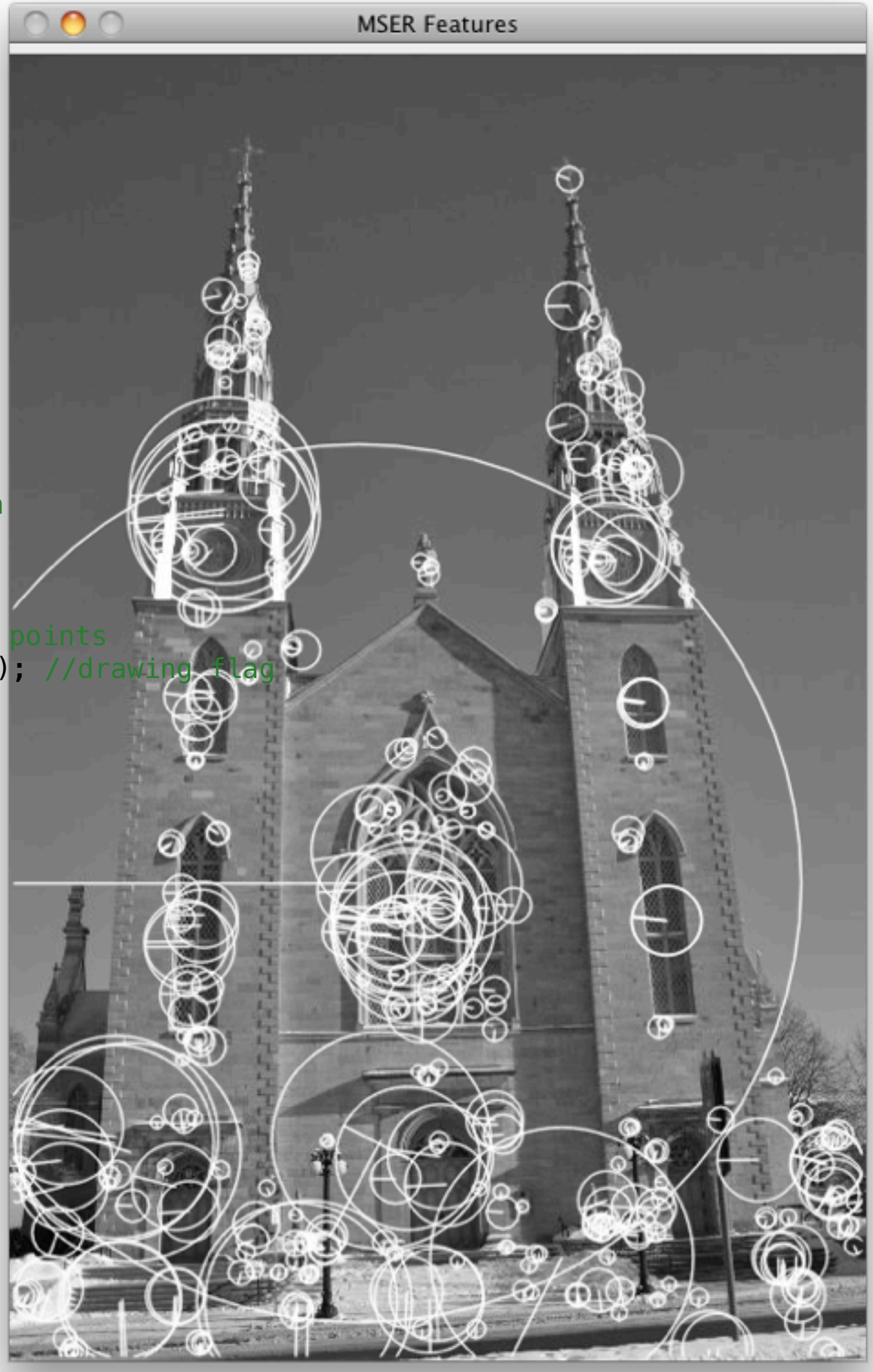
```
// Read input image
image= cv::imread("../images/church03.jpg",0);

keypoints.clear();

// Construct the SURF feature detector object
cv::SurfFeatureDetector surf(2500);

// Detect the SURF features
surf.detect(image,keypoints);

 std::cerr << "SURF Features Detected: "
           << keypoints.size() << std::endl;

cv::Mat featureImage;
cv::drawKeypoints(image,keypoints,featureImage,
                  cv::Scalar(5,255,255),
                  cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

 // Display the corners
cv::namedWindow("SURF Features");
cv::imshow("SURF Features",featureImage);

 cv::waitKey ();
```

```
// Read input image
image= cv::imread("../images/church01.jpg",0);

keypoints.clear();
// Construct the SURF feature detector object
cv::SiftFeatureDetector
 sift(
      0.03,  // feature threshold
      10.);  // threshold to reduce sensitivity to lines

// Detect the SURF features
sift.detect(image,keypoints);

 std::cerr << "SIFT Features Detected: "
           << keypoints.size() << std::endl;

cv::drawKeypoints(image,keypoints,featureImage,
                  cv::Scalar(255,5,255),
                  cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

 // Display the corners
cv::namedWindow("SIFT Features");
cv::imshow("SIFT Features",featureImage);

 cv::waitKey ();
```

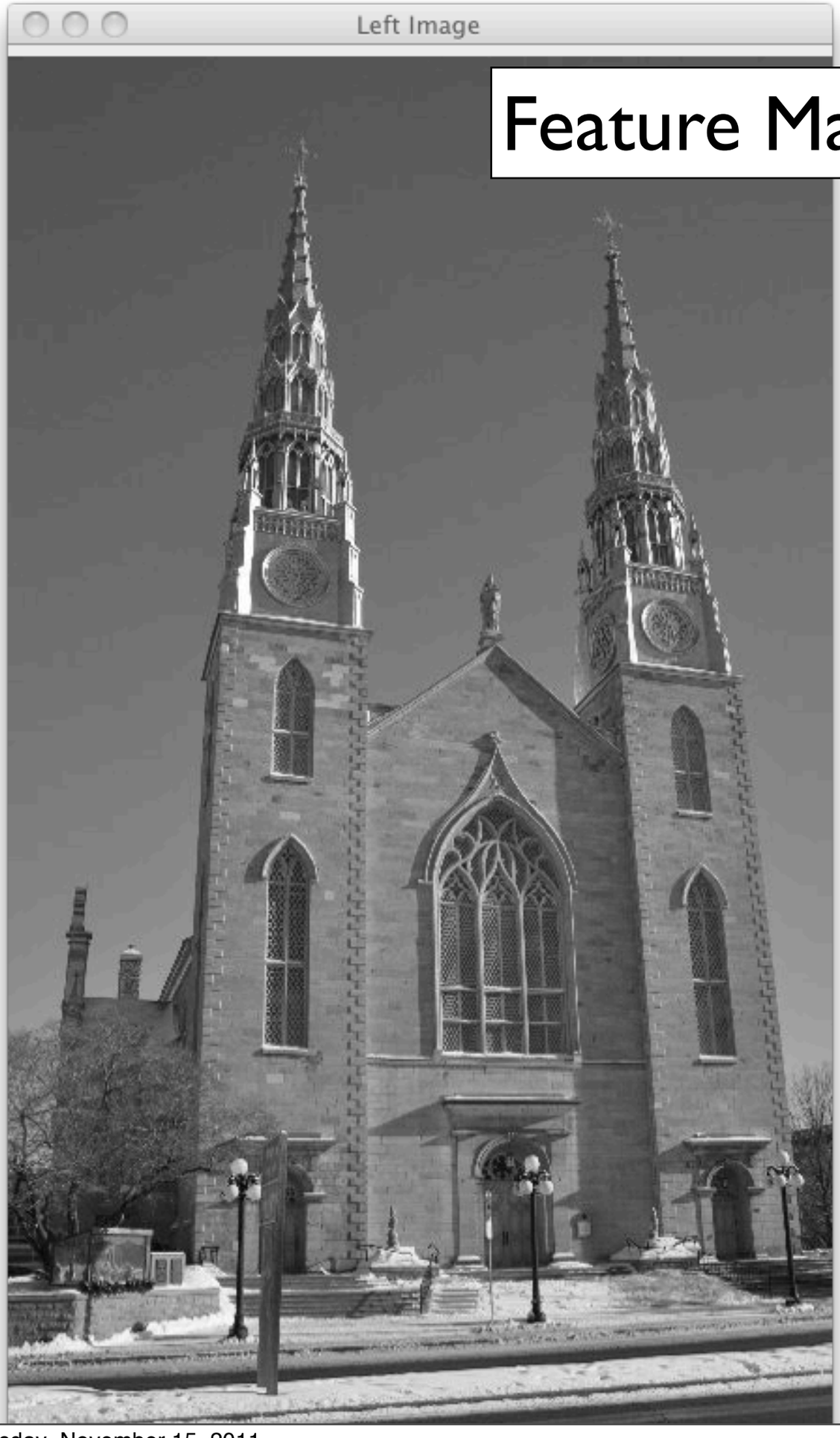


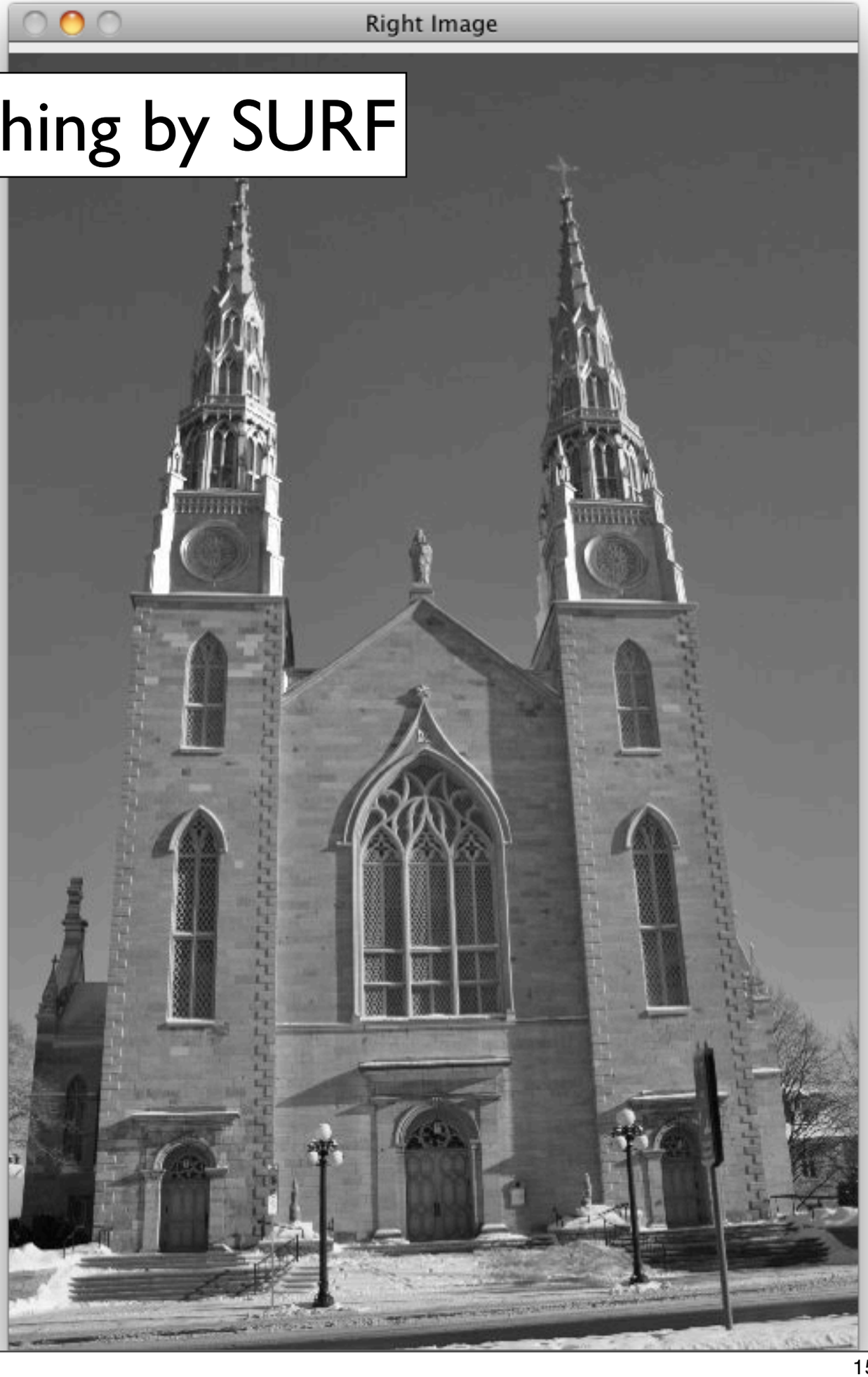
```
// Read input image
image= cv::imread("../images/church01.jpg",0);

keypoints.clear();
// Construct the SURF feature detector object
cv::SiftFeatureDetector
 sift(
      0.09,  // feature threshold
      10.);  // threshold to reduce sensitivity to lines

// Detect the SURF features
sift.detect(image,keypoints);

 std::cerr << "SIFT Features Detected: "
           << keypoints.size() << std::endl;

cv::drawKeypoints(image,keypoints,featureImage,
                  cv::Scalar(255,5,255),
                  cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

 // Display the corners
cv::namedWindow("SIFT Features");
cv::imshow("SIFT Features",featureImage);

 cv::waitKey ();
```

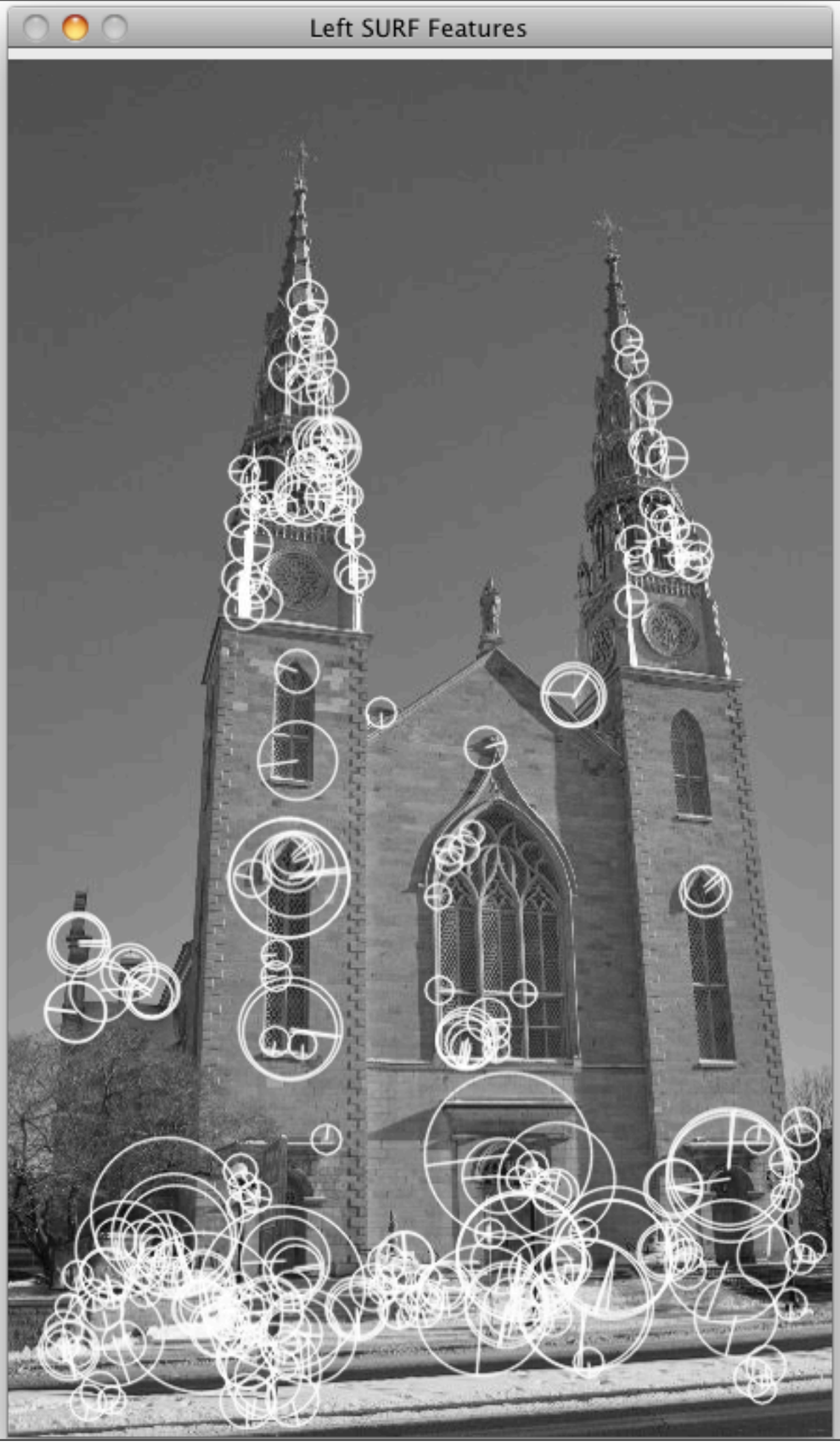
```
// Read input image
image= cv::imread("../images/church01.jpg",0);

keypoints.clear();

cv::MserFeatureDetector mser;

mser.detect(image,keypoints);

 std::cerr << "MSER Features Detected: "
           << keypoints.size() << std::endl;

// Draw the keypoints with scale and orientation information
cv::drawKeypoints(image,    // original image
                  keypoints,    // vector of keypoints
                  featureImage, // the resulting image
                  cv::Scalar(255,255,255),  // color of the points
                  cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS); //drawing flag

 // Display the corners
cv::namedWindow("MSER Features");
cv::imshow("MSER Features",featureImage);

cv::waitKey();
```



Feature Matching by SURF






Left SURF Features


Right SURF Features

Matches

```
int main()
{
    // Read input images
    cv::Mat image1= cv::imread("../images/church01.jpg",0);
    cv::Mat image2= cv::imread("../images/church02.jpg",0);
    if (!image1.data || !image2.data)
        return 0;

    // Display the images
    cv::namedWindow("Right Image");
    cv::imshow("Right Image",image1);
    cv::namedWindow("Left Image");
    cv::imshow("Left Image",image2);

    // vector of keypoints
    std::vector<cv::KeyPoint> keypoints1;
    std::vector<cv::KeyPoint> keypoints2;

    // Construction of the SURF feature detector
    cv::SurfFeatureDetector surf(3000);

    // Detection of the SURF features
    surf.detect(image1,keypoints1);
    surf.detect(image2,keypoints2);

    std::cout << "Number of SURF points (1): " << keypoints1.size() << std::endl;
    std::cout << "Number of SURF points (2): " << keypoints2.size() << std::endl;

    // Draw the kepoints
    cv::Mat imageKP;
    cv::drawKeypoints(image1,keypoints1,imageKP,cv::Scalar(255,255,255),
                      cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
    cv::namedWindow("Right SURF Features");
    cv::imshow("Right SURF Features",imageKP);
    cv::drawKeypoints(image2,keypoints2,imageKP,cv::Scalar(255,255,255),
                      cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
    cv::namedWindow("Left SURF Features");
    cv::imshow("Left SURF Features",imageKP);
```

```
    // Construction of the SURF descriptor extractor
    cv::SurfDescriptorExtractor surfDesc;

    // Extraction of the SURF descriptors
    cv::Mat descriptors1, descriptors2;
    surfDesc.compute(image1,keypoints1,descriptors1);
    surfDesc.compute(image2,keypoints2,descriptors2);

    std::cout << "descriptor matrix size: " << descriptors1.rows
              << " by " << descriptors1.cols << std::endl;

    // Construction of the matcher
    cv::BruteForceMatcher< cv::L2<float> > matcher;

    // Match the two image descriptors
    std::vector<cv::DMatch> matches;
    matcher.match(descriptors1,descriptors2, matches);

    std::cout << "Number of matched points: " << matches.size() << std::endl;

    std::nth_element(matches.begin(),    // initial position
                     matches.begin()+24, // position of the sorted element
                     matches.end());     // end position
    // remove all elements after the 25th
    matches.erase(matches.begin()+25, matches.end());

    cv::Mat imageMatches;
    cv::drawMatches(image1,keypoints1,  // 1st image and its keypoints
                    image2,keypoints2,  // 2nd image and its keypoints
                    matches,            // the matches
                    imageMatches,       // the image produced
                    cv::Scalar(255,255,255)); // color of the lines
    cv::namedWindow("Matches");
    cv::imshow("Matches",imageMatches);

    cv::waitKey();
    return 0;
}
```

Constructor & Destructor Documentation

```
features_2d::SurfDescriptorExtractor::SurfDescriptorExtractor ( int    octaves = 3,
                                                                int    octave_layers = 4,
                                                                bool   extended = false
                                                              )
```

Definition at line 6 of file surf_descriptor.cpp.

Member Function Documentation

```
void features_2d::SurfDescriptorExtractor::compute ( const cv::Mat &                  image,
                                                     std::vector< cv::KeyPoint > &    keypoints,
                                                     cv::Mat &                        descriptors
                                                   )                                  const [virtual]
```

Compute the descriptors for a set of keypoints in an image.

Must be implemented by the subclass.

Parameters:

[in]	*image*	The image.
[in,out]	*keypoints*	The keypoints. Keypoints for which a descriptor cannot be computed are removed.
[out]	*descriptors*	The descriptors. Row i is the descriptor for keypoint i.

Implements features_2d::DescriptorExtractor.

Definition at line 12 of file surf_descriptor.cpp.

http://www.ros.org/doc/api/features_2d/html/classfeatures__2d_1_1SurfDescriptorExtractor.html#a3bfbdd2b6a64fd2ee12cb22ee97aec85

class **BruteForceMatcher**

Brute-force descriptor matcher. For each descriptor in the first set, this matcher finds the closest descriptor in the second set by trying each one. This descriptor matcher supports masking permissible matches of descriptor sets.

class **DMatch**

Class for matching keypoint descriptors: query descriptor index, train descriptor index, train image index, and distance between descriptors.

```
struct DMatch
{
    int queryIdx; // query descriptor index
    int trainIdx; // train descriptor index
    int imgIdx;   // train image index

    float distance;

    // less is better
    bool operator<( const DMatch &m ) const;
   // some more ...
};
```

Sort element in range

Rearranges the elements in the range *[first,last)*, in such a way that the element at the resulting *nth* position is the element that would be in that position in a sorted sequence, with none of the elements preceding it being greater and none of the elements following it smaller than it. Neither the elements preceding it nor the elements following it are guaranteed to be ordered.

The elements are compared using `operator<` for the first version, and *comp* for the second.

```
// nth_element example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool myfunction (int i,int j) { return (i<j); }

int main () {
  vector<int> myvector;
  vector<int>::iterator it;

  // set some values:
  for (int i=1; i<10; i++) myvector.push_back(i);   // 1 2 3 4 5 6 7 8 9

  random_shuffle (myvector.begin(), myvector.end());

  // using default comparison (operator <):
  nth_element (myvector.begin(), myvector.begin()+5, myvector.end());

  // using function as comp
  nth_element (myvector.begin(), myvector.begin()+5, myvector.end(),myfunction);

  // print out content:
  cout << "myvector contains:";
  for (it=myvector.begin(); it!=myvector.end(); ++it)
    cout << " " << *it;

  cout << endl;

  return 0;
}
```

```
myvector contains: 3 1 4 2 5 6 9 7 8
```