Technology Workshop 2011

서강대학교 영상대학원

서 용 덕

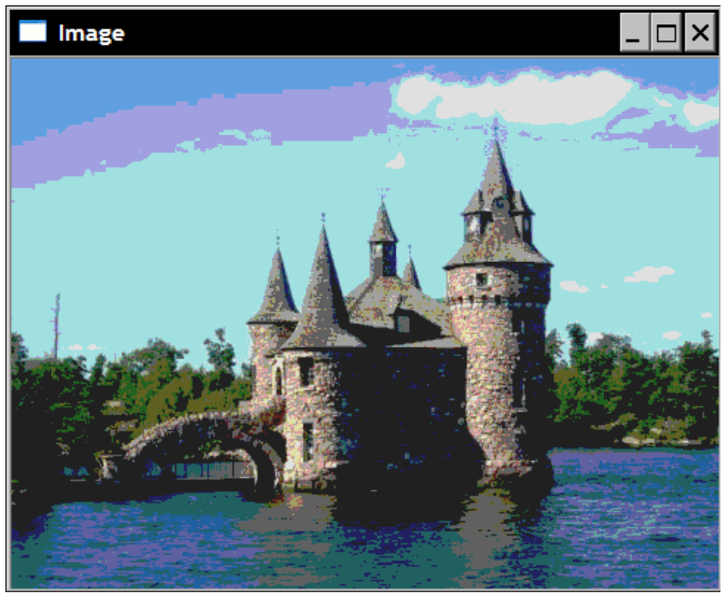
yndk@sogang.ac.kr

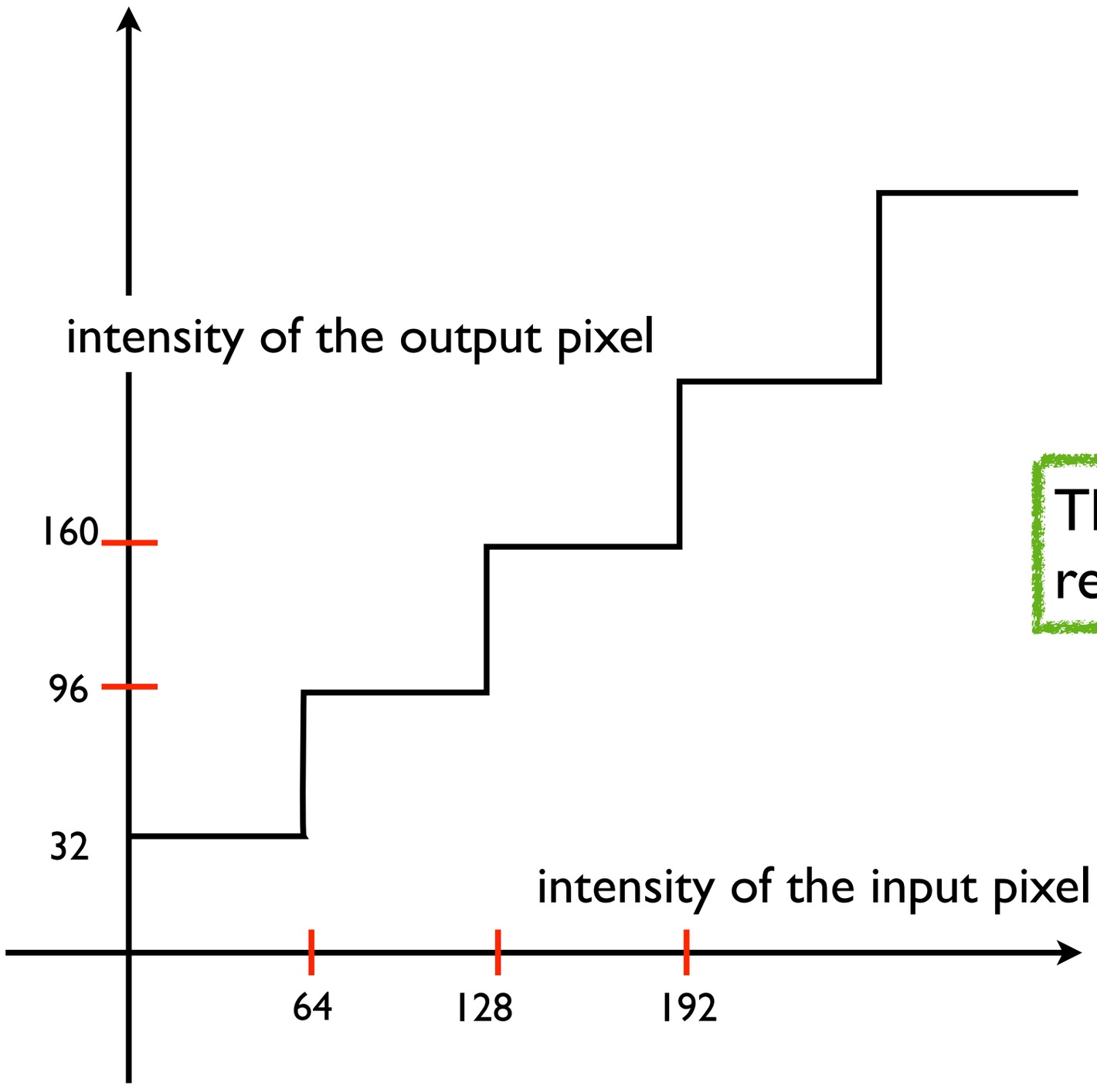
Color Reduction: Reduce the number of colors

```
#include <iostream>

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>

// using .ptr and []
void colorReduce0(cv::Mat &image, int div=64) {

      int nl= image.rows; // number of lines
      int nc= image.cols * image.channels(); // total number of elements per line

      for (int j=0; j<nl; j++) {

         uchar* data= image.ptr<uchar>(j);

          for (int i=0; i<nc; i++) {
            // process each pixel ---------------------

                  data[i]= data[i]/div*div + div/2;

            // end of pixel processing ----------------
            } // end of line
      }
}
```

```
div=64;
data[i]= data[i]/div*div + div/2;
```

```
div/2 => 32;
data[i]/div*div == (data[i]/div)*div + 32
```





This corresponds to a 3-bit representation of image colors.

```
// using .ptr and * ++
void colorReduce1(cv::Mat &image, int div=64)
{
    int nl= image.rows; // number of lines
    int nc= image.cols * image.channels();

    for (int j=0; j<nl; j++) {
        uchar* data= image.ptr<uchar>(j);
        for (int i=0; i<nc; i++) {
            *data++= *data/div*div + div/2;
        } // end of line
    }
}
```

```
// using .ptr and * ++ and modulo
void colorReduce2(cv::Mat &image, int div=64)
{
    int nl= image.rows; // number of lines
    int nc= image.cols * image.channels();

    for (int j=0; j<nl; j++)
        {
        uchar* data= image.ptr<uchar>(j);
        for (int i=0; i<nc; i++) {
            int v= *data;
            *data++= v - v%div + div/2;
        } // end of line
    }
}
```

```
// using .ptr and * ++ and bitwise
void colorReduce3(cv::Mat &image, int div=64)
{
    int nl= image.rows; // number of lines
    int nc= image.cols * image.channels(); // total number of elements per line
    int n= static_cast<int>(log(static_cast<double>(div))/log(2.0));
    // mask used to round the pixel value
    uchar mask= 0xFF<<n; // e.g. for div=16, mask= 0xF0

    for (int j=0; j<nl; j++) {
        uchar* data= image.ptr<uchar>(j);
        for (int i=0; i<nc; i++) {
            *data++= *data&mask + div/2;
        } // end of line
    }
}
```

Integer number division can be done by shift, sometimes.

NUM / 2 == NUM >> 1
NUM / 4 == NUM >> 2
NUM / 8 == NUM >> 3
NUM / 16 == NUM >> 4
NUM / 32 == NUM >> 5

log2 (2) = 1
log2 (4) = 2
log2 (8) = 3
log2 (16) = 4
log2 (32) = 5

```
#include <math.h>
#include <iostream>

int main()
{
    for (int i=0; i<10; i++)
        {
        double pow_2 = pow (2, i);
        double log_2 = log2(pow_2);
        int n_pow_2 = (int) pow_2;
        int n_log_2 = log2 (n_pow_2);
        printf ("2^%d = %.1lf -> %d, its log2 is %.1lf or %d\n",
                i, pow_2, n_pow_2, log_2, n_log_2);
        }
    return 0;
}
```

Scanning an image with neighbor access
image sharpening

Image sharpening = High Pass Filtering

Keywords: Filter Kernel, Convolution, Correlation

0	-1	0
-1	5	-1
0	-1	0

contrast.cpp

```
#include <iostream>

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

void sharpen(const cv::Mat &image, cv::Mat &result)
{
    result.create(image.size(), image.type()); // allocate if necessary

    for (int j= 1; j<image.rows-1; j++) { // for all rows (except first and last)

       const uchar* previous= image.ptr<const uchar>(j-1); // previous row
       const uchar* current= image.ptr<const uchar>(j);  // current row
       const uchar* next= image.ptr<const uchar>(j+1);      // next row

       uchar* output= result.ptr<uchar>(j); // output row

       for (int i=1; i<image.cols-1; i++) {
          output[i]= cv::saturate_cast<uchar>(
                              5*current[i]-current[i-1]-current[i+1]-previous[i]-next[i]
                              );
          //*output++= ...

       }
    }

    // Set the unprocess pixels to 0
    result.row(0).setTo(cv::Scalar(0));
    result.row(result.rows-1).setTo(cv::Scalar(0));
    result.col(0).setTo(cv::Scalar(0));
    result.col(result.cols-1).setTo(cv::Scalar(0));
}
```

C++: Type Casting

Converting an expression of a given type into another type.

Implicit conversion

Implicit conversions do not require any operator. They are automatically performed when a value is copied to a compatible type. For example:

```
short a=2000;
int b;
b=a;
```

Explicit conversion

C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion. We have already seen two notations for explicit type conversion: functional and c-like casting:

```
short a=2000;
int b;
b = (int) a;    // c-like cast notation
b = int (a);    // functional notation
```

static_cast

`static_cast` can also be used to perform any other non-pointer conversion that could also be performed implicitly, like for example standard conversion between fundamental types:

```
double d=3.14159265;
int i = static_cast<int>(d);
```

Or any conversion between classes with explicit constructors or operator functions as described in "implicit conversions" above.

http://cplusplus.com/doc/tutorial/typecasting/

```
void sharpen3(const cv::Mat &image, cv::Mat &result)
{
    cv::Mat_<uchar>::const_iterator it= image.begin<uchar>() + image.step;
    cv::Mat_<uchar>::const_iterator itend= image.end<uchar>() - image.step;
    cv::Mat_<uchar>::const_iterator itup= image.begin<uchar>();
    cv::Mat_<uchar>::const_iterator itdown= image.begin<uchar>() + 2*image.step;

    result.create(image.size(), image.type()); // allocate if necessary
    cv::Mat_<uchar>::iterator itout= result.begin<uchar>() + result.step;

    for ( ; it!= itend; ++it, ++itup, ++itdown) {

        *itout= cv::saturate_cast<uchar>(*it *5 - *(it-1)- *(it+1)- *itup - *itdown);
    }
}
```

```
void sharpen2D(const cv::Mat &image, cv::Mat &result) {

    // Construct kernel (all entries initialized to 0)
    cv::Mat kernel(3,3,CV_32F,cv::Scalar(0));
    // assigns kernel values
    kernel.at<float>(1,1)= 5.0;
    kernel.at<float>(0,1)= -1.0;
    kernel.at<float>(2,1)= -1.0;
    kernel.at<float>(1,0)= -1.0;
    kernel.at<float>(1,2)= -1.0;

    //filter the image
    cv::filter2D(image,result,image.depth(),kernel);
}
```

```
int main()
{
    cv::Mat image= cv::imread("boldt.jpg",0);
    if (!image.data)
       return 0;

    cv::Mat result(image.size(),image.type());

    sharpen(image, result);
    cv::namedWindow("Image");
    cv::imshow("Image",result);

    sharpen2D(image, result);
    cv::namedWindow("Image 2D");
    cv::imshow("Image 2D",result);

    cv::waitKey();

    return 0;
}
```

Mat_

Template matrix class derived from **Mat** .

```
template<typename _Tp> class Mat_ : public Mat
{
public:
    // ... some specific methods
    //          and
    // no new extra fields
};
```

The class `Mat_<_Tp>` is a “thin” template wrapper on top of the `Mat` class.

While `Mat` is sufficient in most cases, `Mat_` can be more convenient if you use a lot of element access operations and if you know matrix type at the compilation time.

```
Mat_<double> M(20,20);
for(int i = 0; i < M.rows; i++)
    for(int j = 0; j < M.cols; j++)
        M(i,j) = 1./(i+j+1);
Mat E, V;
eigen(M,E,V);   // eigen decomposition of M
cout << E.at<double>(0,0)/E.at<double>(M.rows-1,0);
```

To use `Mat_` for multi-channel images/matrices, pass `Vec` as a `Mat_` parameter:

```
// allocate a 320x240 color image and fill it with green (in RGB space)
Mat_<Vec3b> img(240, 320, Vec3b(0,255,0));
// now draw a diagonal white line
for(int i = 0; i < 100; i++)
    img(i,i)=Vec3b(255,255,255);
// and now scramble the 2nd (red) channel of each pixel
for(int i = 0; i < img.rows; i++)
    for(int j = 0; j < img.cols; j++)
        img(i,j)[2] ^= (uchar)(i ^ j);
```

http://opencv.itseez.com/modules/core/doc/basic_structures.html#id5

filter2D

Convolves an image with the kernel.

C++: void **cv::filter2D**(InputArray **src**, OutputArray **dst**, int **ddepth**, InputArray **kernel**, Point**anchor**=Point(-1,-1), double **delta**=0, int **borderType**=BORDER_DEFAULT)

C: void **cvFilter2D**(const CvArr* **src**, CvArr* **dst**, const CvMat* **kernel**, CvPoint**anchor**=cvPoint(-1, -1))

Parameters:

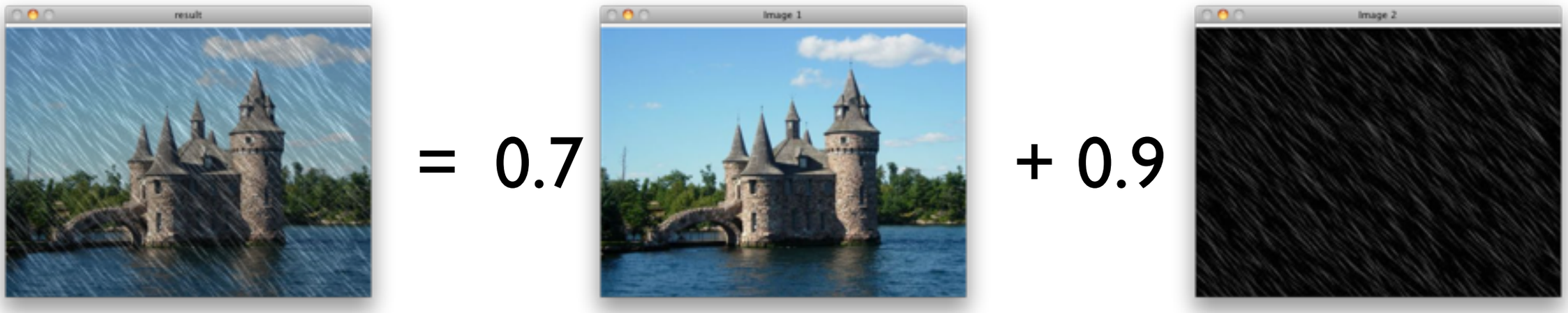
1. **src** – Source image.
2. **dst** – Destination image of the same size and the same number of channels as`src` .
3. **ddepth** – Desired depth of the destination image. If it is negative, it will be the same as `src.depth()` .
4. **kernel** – Convolution kernel (or rather a correlation kernel), a single-channel floating point matrix. If you want to apply different kernels to different channels, split the image into separate color planes using **split()** and process them individually.
5. **anchor** – Anchor of the kernel that indicates the relative position of a filtered point within the kernel. The anchor should lie within the kernel. The special default value (-1,-1) means that the anchor is at the kernel center.
6. **delta** – Optional value added to the filtered pixels before storing them in `dst` .
7. **borderType** – Pixel extrapolation method. See **borderInterpolate()** for details.

The function applies an arbitrary linear filter to an image. In-place operation is supported. When the aperture is partially outside the image, the function interpolates outlier pixel values according to the specified border mode.

The function does actually compute correlation, not the convolution.

Performing simple image arithmetic

```
Result(x,y) = a*Input1(x,y) + b*Input2(x,y) + c
```




```
cv::Mat image1;
cv::Mat image2;

image1= cv::imread("boldt.jpg");
image2= cv::imread("rain.jpg");
if (!image1.data)
   return 0;
if (!image2.data)
   return 0;

cv::Mat result;
cv::addWeighted(image1,0.7,image2,0.9,0.,result);
```

$$Result(x,y) = a*Input1(x,y) + b*Input2(x,y) + c$$

```
int main()
{
    cv::Mat image1;
    cv::Mat image2;

    image1= cv::imread("boldt.jpg");
    image2= cv::imread("rain.jpg");
    if (!image1.data)
       return 0;
    if (!image2.data)
       return 0;

    cv::namedWindow("Image 1");
    cv::imshow("Image 1",image1);
    cv::namedWindow("Image 2");
    cv::imshow("Image 2",image2);

    cv::Mat result;
    cv::addWeighted(image1,0.7,image2,0.9,0.,result);

    cv::namedWindow("result");
    cv::imshow("result",result);

    // using overloaded operator
    result= 0.7*image1+0.9*image2;

    cv::namedWindow("result with operators");
    cv::imshow("result with operators",result);
```

Image addition only on the BLUE channel

```
    image2= cv::imread("rain.jpg",0);

    // create vector of 3 images
    std::vector<cv::Mat> planes;

    // split 1 3-channel image into 3 1-channel images
    cv::split(image1,planes);

    // add to blue channel
    planes[0]+= image2;

    // merge the 3 1-channel images into 1 3-channel image
    cv::merge(planes,result);

    cv::namedWindow("Result on blue channel");
    cv::imshow("Result on blue channel",result);
```

ROI defined by Rect

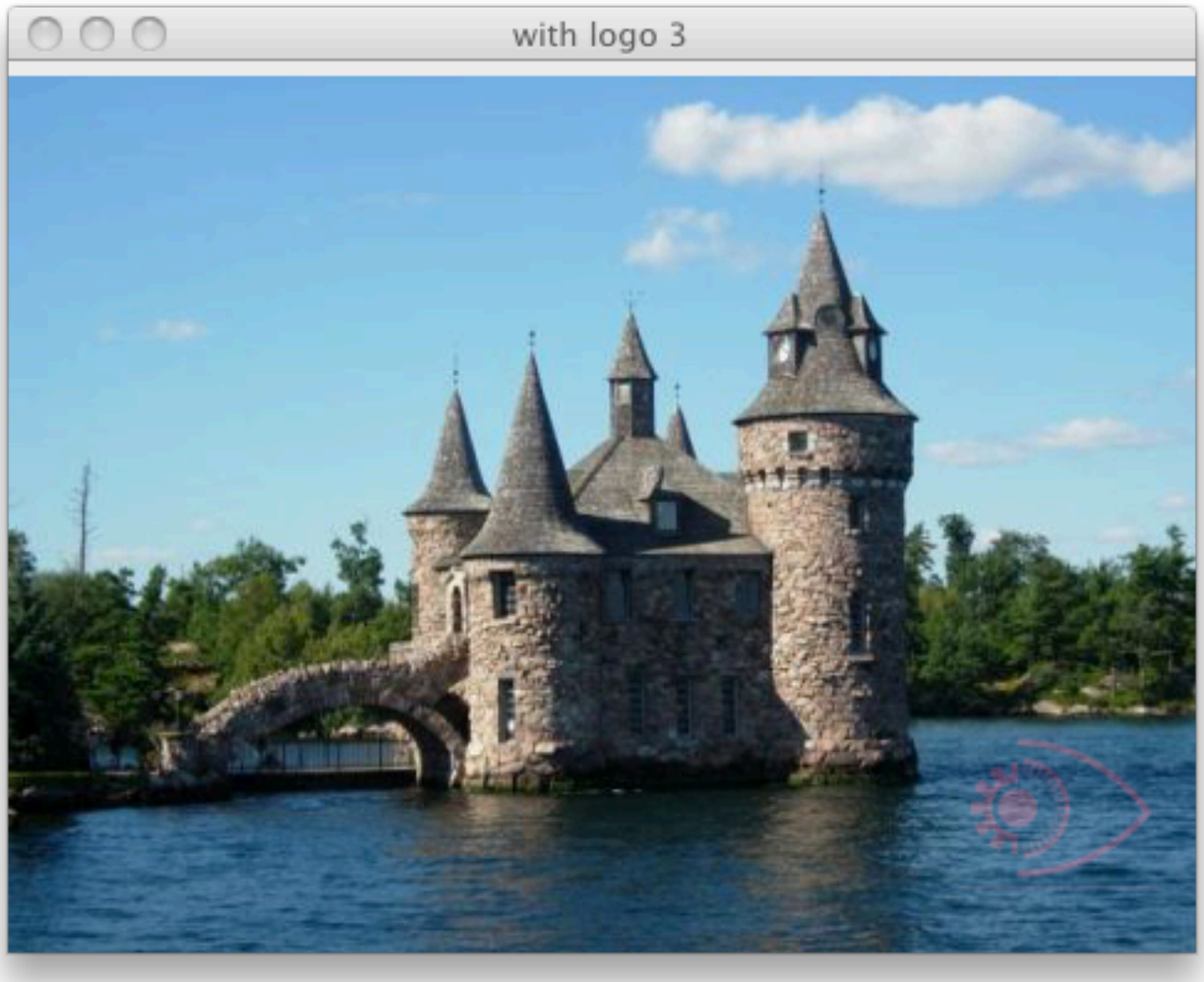
```
// read images
cv::Mat image= cv::imread("boldt.jpg");
cv::Mat logo= cv::imread("logo.bmp");

// define image ROI
cv::Mat imageROI;
imageROI= image(cv::Rect(385,270,logo.cols,logo.rows));

// add logo to image
cv::addWeighted(imageROI,1.0,logo,0.3,0.,imageROI);

// show result
cv::namedWindow("with logo");
cv::imshow("with logo",image);
```

```
// read images
image= cv::imread("boldt.jpg");
logo= cv::imread("logo.bmp");

// define ROI
imageROI= image(cv::Rect(385,270,logo.cols,logo.rows));

// load the mask (must be gray-level)
cv::Mat mask= cv::imread("logo.bmp",0);

// copy to ROI with mask
logo.copyTo(imageROI,mask);

// show result
cv::namedWindow("with logo 2");
cv::imshow("with logo 2",image);
```

```
// read images
logo= cv::imread("logo.bmp",0);
image1= cv::imread("boldt.jpg");

// split 3-channel image into 3 1-channel images
std::vector<cv::Mat> channels;
cv::split(image1,channels);

cv::Mat red_channel = channels[2];
imageROI = red_channel(cv::Rect(385,270,logo.cols,logo.rows));

cv::addWeighted(imageROI,1.0,logo,0.5,0.,imageROI);

cv::merge(channels,image1);

cv::namedWindow("with logo 3");
cv::imshow("with logo 3",image1);
```

Homework: make C/C++ program to obtain the same result using direct access to pixels.

```
void imageOverlay (cv::Mat& result, cv::Mat& input, cv::Mat& logo, int x0, int y0)

// Even though logo is assumed to be of a small size
// you need to consider out-of-boundary cases in the loop.
// Use your own input image and logo!
```

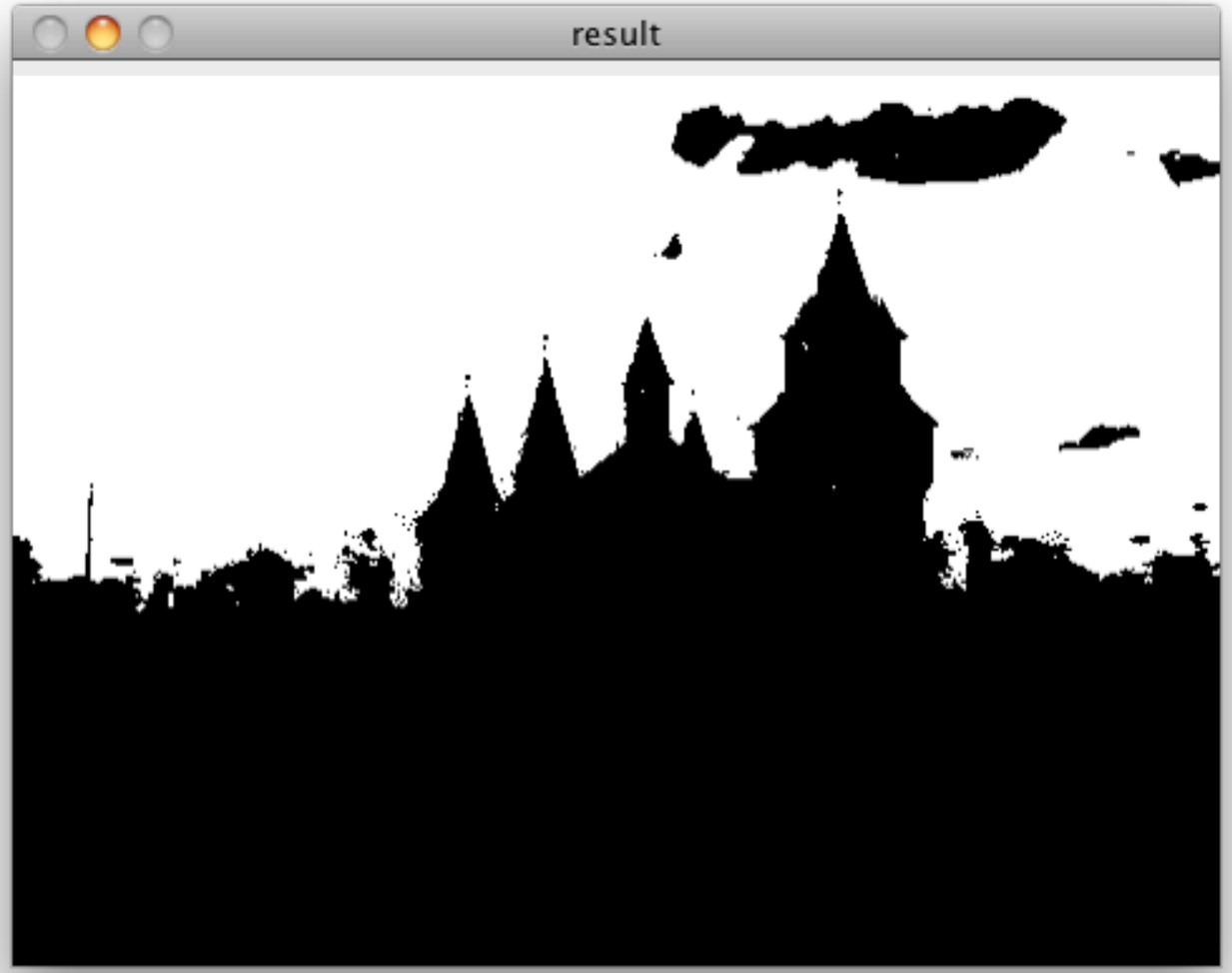


3

Processing Images with Classes

Color Detection

Given a color value, the black area represents those pixels of similar color.



colorDetection.cpp

```
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>

#include "colordetector.h"

int main()
{
    // Create image processor object
    ColorDetector cdetect;

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

    // set input parameters
    cdetect.setTargetColor(130,190,230); // here blue sky

    // Read image, process it and display the result
    cv::Mat result = cdetect.process(image);
    cv::namedWindow("result");
    cv::imshow("result", result);

    cv::waitKey();

    return 0;
}
```

colordetector.h

```
class ColorDetector {
private:
    // minimum acceptable distance
    int minDist;
    // target color
    cv::Vec3b target;
    // image containing resulting binary map
    cv::Mat result;
    // inline private member function
    // Computes the distance from target color.
    int getDistance(const cv::Vec3b& color) const {
        return abs(color[0]-target[0])+
               abs(color[1]-target[1])+
               abs(color[2]-target[2]);
    }

public:
    // empty constructor
    ColorDetector() : minDist(100) {
        // default parameter initialization here
        target[0]= target[1]= target[2]= 0;
    }
```

colordetector.h

```
        // Getters and setters

        // Sets the color distance threshold.
        // Threshold must be positive, otherwise distance threshold
        // is set to 0.
        void setColorDistanceThreshold(int distance) {
            if (distance<0)
                distance=0;
            minDist= distance;
        }
        // Gets the color distance threshold
        int getColorDistanceThreshold() const {
            return minDist;
        }
        // Sets the color to be detected
        void setTargetColor(unsigned char red, unsigned char green, unsigned char blue) {
            target[2]= red;
            target[1]= green;
            target[0]= blue;
        }
        // Sets the color to be detected
        void setTargetColor(cv::Vec3b color) {
            target= color;
        }
        // Gets the color to be detected
        cv::Vec3b getTargetColor() const {
            return target;
        }
        // Processes the image. Returns a 1-channel binary image.
        cv::Mat process(const cv::Mat &image);
    }; // class ColorDetector
```

colordetector.cpp

```
#include "colordetector.h"

cv::Mat ColorDetector::process(const cv::Mat &image)
{
      // re-allocate binary map if necessary
      // same size as input image, but 1-channel
      result.create(image.rows, image.cols, CV_8U);

      // get the iterators
      cv::Mat_<cv::Vec3b>::const_iterator it= image.begin<cv::Vec3b>();
      cv::Mat_<cv::Vec3b>::const_iterator itend= image.end<cv::Vec3b>();
      cv::Mat_<uchar>::iterator itout= result.begin<uchar>();

      // for each pixel
      for ( ; it!= itend; ++it, ++itout) {

       // process each pixel ---------------------

         // compute distance from target color
         if (getDistance(*it)<minDist)
            *itout= 255;
         else
            *itout= 0;

        // end of pixel processing -----------------
      }
      return result;
}
```

Homework: make C/C++ program to obtain the same result using direct access to pixels.

```
cv::Mat ColorDetector::process_direct_access(const cv::Mat &image)
{
    ...
}
```

4

Counting the Pixels with Histograms

In this chapter, we will cover:

- Computing the image histogram
- Applying look-up tables to modify image appearance
- Equalizing the image histogram
- Backprojecting a histogram to detect specific image content
- Using the mean shift algorithm to find an object
- Retrieving similar images using histogram comparison