Image-processing Library in C++

Chen Lujie

April 17, 2015

Contents

1 Introduction

C++, standard template library (STL), computer graphics and image processing, each of the areas is a big topic by itself; while they are indeed interrelated. This article aims to explore the interrelationship and document an image processing library in C++.

The article focuses on the input and output parameters of the library functions with some concise background information. Readers are encouraged to exam the source code where brief comments can be found explaining the implementation; however, code reading is not required for using the library. A complete package of the library files (*.h and *.cpp) may be requested from iiuufigtt@gmail.com. Please kindly note that this is not an open source or free library; however, two programs: UU and Fig, which are written based on the library, are free to the public. More information is available at `www.sutd.edu.sg/ChenLujie.aspx`.

1.1 Features

A variety of image processing libraries are available on the internet. The highlight of this library lies in the following features.

Simplicity. The library does not rely on any particular image class; so you can keep your data structure unchanged. (Many image processing libraries define their own image class and data structure conversion is required, which is likely to induce data-processing overhead.)

Flexibility. Based on C++ template mechanism, the library accepts all C++ built-in types as the image data, although certain functions are applicable to a subset of the built-in types. If you have integer and float image data to process, as an example, all you need to do is to generate two explicit template instantiations of a library function. It is a standard C++ technique and will be discussed in detail in Section 1.3.

Independency. Most of the image processing functions in the library are relatively independent. Combination of some functions may achieve sophisticated processing tasks.

Portability. The library is written in ANSI C++. It can be compiled by many compilers, such as Visual Studio C++ and g++.

Readability. Input and output of all functions are systemically organized. By following a few examples, you will be familiar with what to supply to a function and what can be expected from the output. Documentation is also provided in the source code, above each function's definition.

1.2 Source files

The source files of the library are listed in Table 1. To use the library, all cpp files should be compiled. Two header files ImgProc2D.h, and ImgProc3D.h should be included in whichever file that uses the library. The rest of the header files are included automatically in ImgProc2D.h, and ImgProc3D.h. It is important to know that the library classes and functions are defined in a CLJ namespace; and therefore after including ImgProc2D.h and/or ImgProc3D.h, you should write "using namespace CLJ;". The following code shows an example.

```
#include "PreCompiled.h"
#if !USE_PRECOMPILED_HEADER
 #include <iostream>
 #include <vector>
 using namespace std;
#endif
#include "ImgProc2D.h"
using namespace CLJ;

int main()
{
 int w = 256, h = 512;
 vector<float> vI2D1(w*h), vI2D2(w*h);
 ImgAssign(&vI2D1[0], w, h, CRect<int>(0,0,w,h), 50.f);
 ImgAssign(&vI2D2[0], w, h, CRect<int>(0,0,w,h), 100.f);
 cout<<"ImgAssign done.\n";
 ImgCopy(&vI2D1[0], w, h, CRect<int>(20,20,w-10,h-10),
         &vI2D2[0], w, h, CRect<int>(10,10,w-20,h-20));
 cout<<"ImgCopy done.\n";
 return 0;
}
```

Main files	Description
ImgPro2D.cpp and .h	2D image processing functions
ImgPro3D.cpp and .h	3D image processing functions
Shape2D.cpp and .h	2D geometry classes and functions
Shape3D.cpp and .h	3D geometry classes and functions
Support files	**Description**
PreCompiled.h	precompiled header file
Util.cpp and .h	utility functions source and header file
nr.h	"Numerical Recipes" (NR) header file
nrtypes_nr.h	NR data types
nrutil_nr.h	NR utilities
jacobi.cpp	NR eigenvalues and eigenvectors
lubksb.cpp	NR LU decomposition
ludcmp.cpp	NR LU decomposition
pythag.cpp	NR SVD
svdcmp.cpp	NR SVD

Table 1: Source files of the library.

There is a header file, PreCompiled.h, included at the first line of all .cpp files. It defines the ASSERT(...) macro used by the library for error reporting. If _DEBUG is defined, ASSERT(...) macro is the same as the C++ library function assert(...); otherwise, it does nothing. In PreCompiled.h, some C++ library headers are conditionally included by

`#define USE_PRECOMPILED_HEADER 1` (use precompilation) or

`#define USE_PRECOMPILED_HEADER 0` (not use precompilation).

If you use the precompilation feature of your C++ compiler, then make PreCompiled.h as the precompiled header of all your .cpp files. (Include PreCompiled.h at the first line of all your .cpp files.) Create a new file called PreCompiled.cpp, which contains just one line of code.

`#include "PreCompiled.h"`

Make PreCompiled.cpp as the precompiled source file of your project. This procedure is standard to making use of precompilation, which should be obvious to people who are familiar with compiler settings. However, if you are confused, you may simply

`#define USE_PRECOMPILED_HEADER 0`

1.3 Explicit template instantiation

An issue associated with using template functions and template classes is "explicit template instantiation". Basically, if you want to use a template function or a template class in a particular data type (for example: float), there should be an explicit template function or class instantiation of that type (float). The library's "explicit template instantiation" is put at the end of each cpp file, where you can find some code like:

```
// Code in Shape2D.cpp
template class CPt2D<int>;
template class CPt2D<float>;
template class CRect<int>;
template class CRect<float>;

// Code in ImgProc2D.cpp
template void ImgAssign<unsigned char>
(unsigned char *pImg, int w, int h, const CRect<int> &rcROI,
 unsigned char value);

template void ImgCopy<char, float>
(const char *pSrc, int w1, int h1, const CRect<int> &rcROI1,
 float *pDst, int w2, int h2, const CRect<int> &rcROI2);
```

```
template void ImgCopy<unsigned char, float>
(const unsigned char *pSrc, int w1, int h1, const CRect<int> &rcROI1,
 float *pDst, int w2, int h2, const CRect<int> &rcROI2);
```

If you need a built-in data type of a specific class or function that has not been instantiated, you can add an instantiation following the above coding style; otherwise you will get compiler errors.

1.4 Coding style

Clear and easy-to-follow coding styles and naming conventions are used in the library.
1. A pointer variable is prefixed by 'p'.
2. A one- or two-letter prefix, such as 'uc', 'c', 'n', and 'd', indicates the data type.
3. General purpose integer variables are usually defined as 'x', 'y', 'z', 'i', 'j' et al.

Example 1 (naming conventions)

char	*pcName = "Mary";
unsigned char	ucRed = 24;
int	nCount;
float	fRatio = 1.f;
double	dRadius = 30;

Example 2 (loop through an image)

```
int x, y, w = 100, h = 50;
float *pfImg = new float[w*h];
for(y=0; y<h; ++y)
    for(x=0; x<w; ++x)
        pfImg[y*w+x] = 3.f;
```

2 2D geometry classes and functions

The coordinate system of the library (see Fig. 1) follows the most widely used convention in computer graphics. The origin (0,0) is located at the left-top corner of an image (or a window, or the computer screen). The positive X axis extends horizontally to the right and the positive Y axis extends vertically downward. An anticlockwise angle with respect to the X axis is defined as positive.

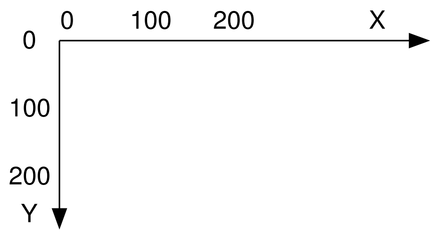


Figure 1: Coordinate system of the image processing library.

2.1 CPt2D: 2D point class

```
template <class T> class CPt2D
{
public:
 T x, y;

 CPt2D() : x(0), y(0) { }
 CPt2D(const CPt2D<T> &pt) : x(pt.x), y(pt.y) { }
 CPt2D(T tx, T ty)         : x(tx),   y(ty)   { }
 CPt2D<T>& operator =  (const CPt2D<T> &pt);
 CPt2D<T>& operator += (const CPt2D<T> &pt);
 CPt2D<T>& operator -= (const CPt2D<T> &pt);
 CPt2D<T>  operator +  (const CPt2D<T> &pt) const;
 CPt2D<T>  operator -  (const CPt2D<T> &pt) const;
 bool      operator == (const CPt2D<T> &pt) const;
 bool      operator != (const CPt2D<T> &pt) const;
 bool      operator <  (const CPt2D<T> &pt) const;

 void SetPoint(T tx, T ty) { x = tx; y = ty; }
 void Rotate(const CPt2D<T> &ptRot, double dAngle);
};
```

Variable	Description
x	x coordinate of the point
y	y coordinate of the point

Function	Description
operator =	Assignment operator
operator + =	*this = *this + pt
operator − =	*this = *this − pt
operator +	Return *this + pt
operator −	Return *this − pt
operator ==	Check if *this is equal to pt
operator !=	Check if *this is not equal to pt

Function	Description
SetPoint	Set the x and y coordinates of the point
Rotate	Rotate the point with respect to a point

2.2 CSize2D: 2D size class

```
template <class T> class CSize2D
{
public:
 T cx, cy;

 CSize2D() : cx(0), cy(0) { }
 CSize2D(const CSize2D<T> &size) : cx(size.cx), cy(size.cy) { }
 CSize2D(T tcx, T tcy)           : cx(tcx),     cy(tcy)     { }
 CSize2D<T>& operator =  (const CSize2D<T> &size);
 CSize2D<T>& operator += (const CSize2D<T> &size);
 CSize2D<T>& operator -= (const CSize2D<T> &size);
 CSize2D<T>  operator +  (const CSize2D<T> &size) const;
 CSize2D<T>  operator -  (const CSize2D<T> &size) const;
 bool        operator == (const CSize2D<T> &size) const;
 bool        operator != (const CSize2D<T> &size) const;

 void SetSize(T tcx, T tcy) { cx = tcx; cy = tcy; }
};
```

Variable	Description
cx	size in x direction
cy	size in y direction

Function	Description
operator =	Assignment operator
operator + =	*this = *this + size
operator − =	*this = *this − size
operator +	Return *this + size
operator −	Return *this − size
operator ==	Check if *this is equal to size
operator !=	Check if *this is not equal to size
operator <	compare two points for sorting

2.3 CLine2D: 2D line segment class

```
template <class T> class CLine2D
{
public:
 CLine2D() : m_ptStart(0,0), m_ptEnd(0,0) { }
 CLine2D(const CLine2D<T> &line);
 CLine2D(const CPt2D<T> &ptStart, const CPt2D<T> &ptEnd);
 CLine2D(T x1, T y1, T x2, T y2);
 CLine2D<T>& operator =  (const CLine2D<T> &line);
 bool        operator == (const CLine2D<T> &line) const;
 bool        operator != (const CLine2D<T> &line) const;

 CPt2D<T> Start()        const;
 CPt2D<T> End()          const;
 CPt2D<T> Center()       const;
 CRect<T> BoundingRect() const;
 double   Angle()        const;
 double   Length()       const;
 void     SetLine   (T x1, T y1, T x2, T y2);
 void     SetLine   (const CPt2D<T> &ptStart, const CPt2D<T> &ptEnd);
 void     SetStart  (T x, T y);
 void     SetStart  (const CPt2D<T> &pt);
 void     SetEnd    (T x, T y);
 void     SetEnd    (const CPt2D<T> &pt);
 void     SetCenter (const CPt2D<T> &pt);
 void     SetAngle  (const CPt2D<T> &ptRot, double dAngle);
 void     SetLength (double dLength, int nFix);
 void     Offset    (T x, T y);
 void     Offset    (const CPt2D<T> &ptOffset);
 void     Rotate    (const CPt2D<T> &ptRot, double dAngle);
 double   DistToPt  (const CPt2D<T> &pt, CPt2D<T> *ppt)     const;
 bool     GetY      (T x, T *pY)                            const;
 bool     GetPointAt(double dDist, CPt2D<T> *ppt)           const;
 bool     LineEqu   (double *pdA, double *pdB, double *pdC) const;

private:
 CPt2D<T> m_ptStart, m_ptEnd; // start and end point
};
```

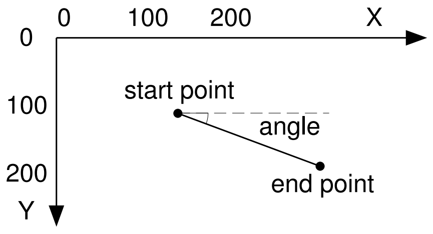


Figure 2: An example of a 2D line segment.

Function	Description
operator =	Assignment operator
operator ==	Check if *this is equal to line
operator !=	Check if *this is not equal to line

Function	Description
Start	Get the start point of the line
End	Get the end point of the line
Center	Get the center of the line
BoundingRect	Get the bounding box of the line
Angle	Get the angle formed by the line and the positive X axis
Length	Get the length of the line
SetLine	Set the start and the end points of the line
SetStart	Set the start point of the line
SetEnd	Set the end point of the line
SetCenter	Set the center of the line
SetAngle	Set the angle of the line The line is rotated with respect to ptRot to the specified angle, formed by the line and the positive X axis.
Offset	Offset the position of the line
Rotate	Rotate the line with respect to a point

SetLength	Set the length of the line To modify the line length, one can fix the start point, the end point, or the center. nFix specifies which point to fix. The angle of the line is always fixed. If the start point and the end point are the same, the line will be extended in the x direction only.
dLength	the new length of the line, which should be >= 0. If dLength is < 0, it is treated as 0.
nFix	0: the start point of the line is fixed; 1: the center of the line is fixed; 2: the end point of the line is fixed.

DistToPt	Get the distance from a point to the line segment
pt	reference point
ppt	the closest point on the line segment to pt. It is the foot of perpendicular from pt to the line, if the foot is inbetween two end points of the line segment; otherwise, it is one of the end points.
Return	distance from the reference point to the line segment

GetY	Given a point's x coordinate on the line, get its corresponding y coordinate.
x	input x coordinate
pY	output y coordinate
Return	true: y coordinate found; false: cannot find y coordinate.

GetPointAt	Get a point on the line that has a specified distance from the start point. Take the start point as a reference, dDist $>$ 0 indicates the point is at the same side as the end point; while dDist $<$ 0 indicates the point is at the opposite side of the end point.
dDist	distance of the point from the start point.
ppt	the point found
Return	true: point found; false: cannot find such a point.

LineEqu	Based on the line segment, retrieve a line equation in the form: A*x + B*y = C. This line equation encompasses all possible situations. Another choice is: cos(angle)*x + sin(angle)*y = R. However, it involves relatively slow trigonometric computation, so it is not used.
pdA	parameter A of the line equation
pdB	parameter B of the line equation
pdC	parameter C of the line equation
Return	true: line function found; false: line function not exist.

2.4 CRect: rectangle class

```
template <class T> class CRect
{
public:
 T left, top, right, bottom;

 CRect() : left(0), top(0), right(0), bottom(0) { }
 CRect(const CRect<T> &rect);
 CRect(const CPt2D<T> &ptLeftTop, const CSize2D<T> &size);
 CRect(const CPt2D<T> &ptLeftTop, const CPt2D<T> &ptRightBottom);
 CRect(T l, T t, T r, T b);
 CRect<T>& operator =  (const CRect<T>& rect);
 CRect<T>& operator &= (const CRect<T>& rect);
 CRect<T>& operator |= (const CRect<T>& rect);
 CRect<T>  operator &  (const CRect<T>& rect) const;
 CRect<T>  operator |  (const CRect<T>& rect) const;
 bool      operator == (const CRect<T>& rect) const;
 bool      operator != (const CRect<T>& rect) const;

 bool       IsEmpty()     const;
 T          Width()       const;
 T          Height()      const;
 T          Area()        const;
 CSize2D<T> Size()        const;
 CPt2D<T>   Center()      const;
 CPt2D<T>   LeftTop()     const;
 CPt2D<T>   RightTop()    const;
 CPt2D<T>   LeftBottom()  const;
 CPt2D<T>   RightBottom() const;
 void SetRect(const CPt2D<T> &ptLeftTop, const CPt2D<T> &ptRightBottom);
 void SetRect(T l, T t, T r, T b);
 void Inflate(T l, T t, T r, T b);
 void Deflate(T l, T t, T r, T b);
 void Inflate(T x, T y);
 void Deflate(T x, T y);
 void Offset (T x, T y);
 void Offset (const CPt2D<T> &pt);
 bool PtIn   (T x, T y)           const;
 bool PtIn   (const CPt2D<T> &pt) const;
 bool RectIn (const CRect<T> &rc) const;
 int  Partition(const CRect<T> &rc, vector<CRect<T> > *pvRc) const;
 void Regularize();
};
```

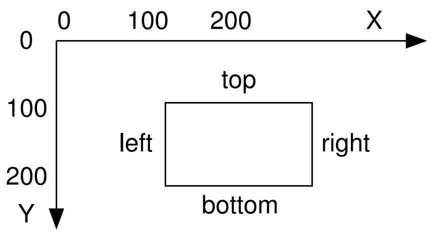


Figure 3: An example of a rectangle.

Variable	Description
left	left bound of the rectangle
top	top bound of the rectangle
right	right bound of the rectangle
bottom	bottom bound of the rectangle

Function	Description
operator =	Assignment operator
operator &=	*this = Intersection of two rectangles
operator |=	*this = Union of two rectangles
operator &	Return intersection of two rectangles
operator |	Return union of two rectangles
operator ==	Check if *this is equal to rect
operator !=	Check if *this is not equal to rect

Function	Description
IsEmpty	Check if it is an empty rectangle. A rectangle is empty if its width or height is <= 0.
Width	Get the width ($right - left$) of the rectangle
Height	Get the height ($bottom - top$) of the rectangle
Size	Get the size of the rectangle
Center	Return the center of the rectangle
LeftTop	Return the left-top point of the rectangle
RightTop	Return the right-top point of the rectangle
LeftBottom	Return the left-bottom point of the rectangle
RightBottom	Return the right-bottom point of the rectangle

Function	Description
SetRect	Set the left, top, right and bottom value of the rectangle
Inflate	Inflate the rectangle by l, t, r, b on the left, top, right, and bottom sides respectively.

continued on next page

	Inflate the rectangle by x on the left and right sides and by y on the top and bottom sides respectively.
Deflate	Deflate the rectangle by l, t, r, b on the left, top, right, and bottom sides respectively. Deflate the rectangle by x on the left and right sides and by y on the top and bottom sides respectively.
Offset	Offset the position of the rectangle
PtIn	Check if a point is inside the rectangle. A point is side a rectangle, if $pt.x \in [left, right)$ AND $pt.y \in [top, bottom)$,
RectIn	Check if a rectangle is completely inside or on *this.
Regularize	Make sure the rectangle has non-negative width and height. If $left > right$, swap left and right. If $top > bottom$, swap top and bottom.

Partition	Partition this rectangle by another one If rc completely encompasses or does not overlap *this, there is no partition; otherwise, *this is partitioned by rc into several pieces.
rc	rectangle to partition *this
pvRc	partitions of *this if exist
Return	-1: rc does not overlap *this, 0: rc completely encompasses *this, the number of partitions: rc overlaps *this.

2.5 CRectRot: rotating rectangle class

```
template <class T> class CRectRot
{
public:
 CRectRot() : m_ptCenter(0,0), m_width(0), m_height(0), m_dAngle(0) { }
 CRectRot(const CRectRot &rectRot);
 CRectRot(T x, T y, T width, T height, double dAngle = 0);
 CRectRot(const CPt2D<T> &ptCenter, T width, T height, double dAngle = 0);
 CRectRot(const CPt2D<T> &ptCenter, CSize2D<T> size, double dAngle = 0);
 CRectRot(const CRect<T> &rect, double dAngle = 0);
 CRectRot<T>& operator = (const CRectRot &rectRot);

 T         Width()  const;
 T         Height() const;
 CPt2D<T>  Center() const;
 double    Angle()  const;
 void      SetCenter(T x, T y);
 void      SetCenter(const CPt2D<T> &ptCenter);
 void      SetWidth (T width);
 void      SetHeight(T height);
 void      SetAngle (double dAngle);
 void      SetRectRot(T x, T y, T width, T height, double dAngle);
 void      Corner(CPt2D<T> ppt[4]) const;
 CRect<T>  BoundingRect()          const;
 void      Inflate(T w, T h);
 void      Deflate(T w, T h);
 void      Offset (T x, T y);
 void      Offset (const CPt2D<T> &ptOffset);
 int       PtIn   (T x, T y);          const;
 int       PtIn   (const CPt2D<T> &pt) const;
 void      Rotate (const CPt2D<T> &ptRot, double dAngle);

private:
 CPt2D<T> m_ptCenter; // center of the rectangle
 T        m_width;    // width  of the rectangle
 T        m_height;   // height of the rectangle
 double   m_dAngle;   // anticlockwise angle (radius) of the rectangle
};
```

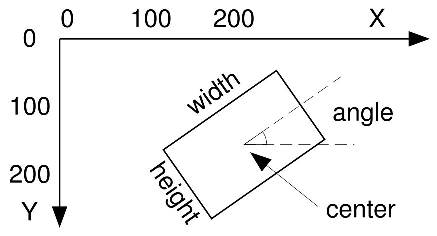


Figure 4: An example of a rotating rectangle.

Function	Description
operator =	Assignment operator
Width	Get the width of the rectangle
Height	Get the height of the rectangle
Center	Get the center of the rectangle
Angle	Get the angle of the rectangle, anticlockwise in radius
SetWidth	Set the width of the rectangle
SetHeight	Set the height of the rectangle
SetCenter	Set the center of the rectangle
SetAngle	Set the angle of the rectangle, anticlockwise in radius
SetRectRot	Set all parameters of the rectangle
Corner	Get 4 corner points of the rectangle
BoundingRect	Get the bounding box of the rectangle
Inflate	Increasing the width and height by w and h respectively.
Deflate	Decreasing the width and height by w and h respectively.
Offset	Offset the position of the rectangle
PtIn	Check if a point is inside the rectangle
Rotate	Rotate the rectangle with respect to a point

2.6 CFan: fan shape class

```
template <class T> class CFan
{
public:
 CFan() : m_ptCenter(0,0), m_Radius(0),
          m_dFanAngle(0),  m_dStartAngle(0) { }
 CFan(const CFan &fan);
 CFan(const CPt2D<T> &ptCenter, T radius,
      double dFanAngle, double dStartAngle);
 CFan(T x, T y, T radius, double dFanAngle, double dStartAngle);
 CFan<T>& operator = (const CFan &fan);

 CPt2D<T> Center()     const;
 T        Radius()     const;
 double   FanAngle()   const;
 double   StartAngle() const;
 CPt2D<T> StartPoint() const;
 CPt2D<T> EndPoint()   const;
 void     SetCenter(T x, T y);
 void     SetCenter(const CPt2D<T> &ptCenter);
 void     SetRadius(T radius);
 void     SetFanAngle  (double dAngle);
 void     SetStartAngle(double dAngle);
 void     SetStartPoint(const CPt2D<T> &pt);
 void     SetEndPoint  (const CPt2D<T> &pt);
 CRect<T> BoundingRect() const;
 void     Offset(T x, T y);
 void     Offset(const CPt2D<T> &ptOffset);
 int      PtIn  (T x, T y);          const;
 int      PtIn  (const CPt2D<T> &pt) const;
 void     Rotate(const CPt2D<T> &ptRot, double dAngle);

private:
 CPt2D<T> m_ptCenter;    // center of the fan
 T        m_Radius;      // radius of the fan
 double   m_dFanAngle;   // anticlockwise angle (radius) of the fan
 double   m_dStartAngle; // anticlockwise start angle (radius)
};
```

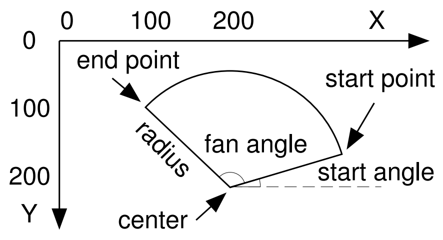


Figure 5: An example of a fan.

Function	Description
operator =	Assignment operator
Center	Get the center of the fan, which is the center of the full circle.
Radius	Get the radius of the fan
FanAngle	Get the angle of the fan, anticlockwise in radius
StartAngle	Get the start angle of the fan, anticlockwise in radius
StartPoint	Get the start point of the fan
EndPoint	Get the end point of the fan
SetCenter	Set the center of the fan
SetRadius	Set the radius of the fan
SetFanAngle	Set the angle of the fan, anticlockwise in radius
SetStartAngle	Set the start angle of the fan, anticlockwise in radius
SetStartPoint	Set the start point of the fan
SetEndPoint	Set the end point of the fan
BoundingRect	Get the bounding box of the fan
Offset	Offset the position of the fan
PtIn	Check if a point is inside the fan
Rotate	Rotate the fan with respect to a point

2.7 CCircle: circle class

```
template <class T> class CCircle
{
public:
 CCircle() : m_ptCenter(0,0), m_Radius(0) { }
 CCircle(const CCircle<T> &circle);
 CCircle(const CPt2D<T> &ptCenter, T radius);
 CCircle(T x, T y, T radius);
 CCircle<T>& operator = (const CCircle<T> &circle);

 CPt2D<T> Center() const;
 T        Radius() const;
 void     SetCenter(T x, T y);
 void     SetCenter(const CPt2D<T> &pt);
 void     SetRadius(T radius);
 CRect<T> BoundingRect() const;
 void     Offset(T x, T y);
 void     Offset(const CPt2D<T> &ptOffset);
 int      PtIn  (T x, T y)           const;
 int      PtIn  (const CPt2D<T> &pt) const;
 void     Rotate(const CPt2D<T> &ptRot, double dAngle);

private:
 CPt2D<T> m_ptCenter; // center of the circle
 T        m_Radius;   // radius of the circle
};
```

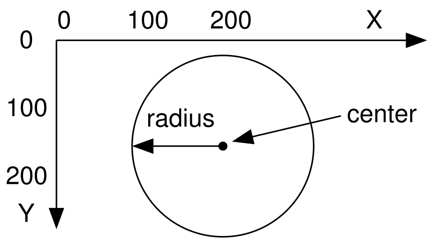


Figure 6: An example of a circle.

Function	Description
operator =	Assignment operator
Center	Get the center of the circle
Radius	Get the radius of the circle
SetCenter	Set the center of the circle
SetRadius	Set the radius of the circle
BoundingRect	Get the bounding box of the circle
Offset	Offset the position of the circle
PtIn	Check if a point is inside the circle
Rotate	Rotate the circle with respect to a point

2.8 CEllipse: ellipse class

```
template <class T> class CEllipse
{
public:
 CEllipse() : m_ptCenter(0,0), m_width(0), m_height(0), m_dAngle(0) { }
 CEllipse(const CEllipse &ellipse);
 CEllipse(T x, T y, T width, T height, double dAngle = 0);
 CEllipse(const CPt2D<T> &ptCenter, T width, T height, double dAngle = 0);
 CEllipse(const CPt2D<T> &ptCenter, CSize2D<T> size, double dAngle = 0);
 CEllipse<T>& operator = (const CEllipse &ellipse);

 T        Width()  const;
 T        Height() const;
 CPt2D<T> Center() const;
 double   Angle()  const;
 void     SetCenter(T x, T y);
 void     SetCenter(const CPt2D<T> &pt);
 void     SetWidth (T width);
 void     SetHeight(T height);
 void     SetAngle (double dAngle);
 double   DistToFoci(const CPt2D<T> &pt) const;
 CRect<T> BoundingRect()          const;
 void     Offset(T x, T y);
 void     Offset(const CPt2D<T> &ptOffset);
 int      PtIn  (T x, T y);          const;
 int      PtIn  (const CPt2D<T> &pt) const;
 void     Rotate(const CPt2D<T> &ptRot, double dAngle);
 bool     SetEqu(const double pdCoe[5]);

private:
 CPt2D<T> m_ptCenter; // center of the ellipse
 T        m_width;    // width  of the ellipse
 T        m_height;   // height of the ellipse
 double   m_dAngle;   // anticlockwise angle (radius) of the ellipse
};
```

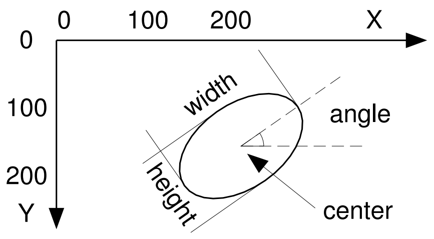


Figure 7: An example of an ellipse.

Function	Description
operator =	Assignment operator
Width	Get the width of the ellipse
Height	Get the height of the ellipse
Center	Get the center of the ellipse
Angle	Get the angle of the ellipse, anticlockwise in radius
SetWidth	Set the width of the ellipse
SetHeight	Set the height of the ellipse
SetCenter	Set the center of the ellipse
SetAngle	Set the angle of the ellipse, anticlockwise in radius
DistToFoci	Get the distance of a point to the two foci of the ellipse
BoundingRect	Get the bounding box of the ellipse
Offset	Offset the position of the ellipse
PtIn	Check if a point is inside the ellipse
Rotate	Rotate the ellipse with respect to a point

SetEqu	Set the ellipse based on an equation Ellipse equation: $Ax^2 + Bxy + Cy^2 + Dx + Ey = 1$
pdCoe	parameters, A, B, C, D and E, of the ellipse equation
Return	true: succeeded; false: the input equation is not an ellipse equation and no change is made.

2.9 CPoly2D: 2D polygon class

```
template <class T> class CPoly2D
{
public:
 CPoly2D() { }
 CPoly2D(const CPoly2D<T> &poly);
 CPoly2D(const vector<CPt2D<T> > &vpt);
 CPoly2D(int nCount, const CPt2D<T> &pt);
 CPoly2D(const CPt2D<T> *ppt, int nCount);
 CPoly2D<T>& operator = (const CPoly2D<T> &poly);

 int Count() const;
 const CPt2D<T>& Vertex(int idx) const;
 const vector<CPt2D<T> >& VertexArray() const { return m_vpt; }
       vector<CPt2D<T> >& VertexArray()       { return m_vpt; }
 void     SetPolygon  (const vector<CPt2D<T> > &vpt);
 void     AddVertex   (         const CPt2D<T> &pt);
 void     SetVertex   (int idx, const CPt2D<T> &pt);
 void     InsertVertex(int idx, const CPt2D<T> &pt);
 void     InsertVertex(int idx, const CPt2D<T> *ppt, int nStart, int nEnd);
 void     RemoveVertex(int idx);
 void     RemoveNearbyVertices(double dDist);
 void     Clear();
 CPt2D<T> Center()       const;
 CRect<T> BoundingRect() const;
 void     Offset (T x, T y);
 void     Offset (const CPt2D<T> &ptOffset);
 int      PtIn   (T x, T y)           const;
 int      PtIn   (const CPt2D<T> &pt) const;
 void     Rotate (const CPt2D<T> &ptRot, double dAngle);
 void     Rescale(double dXScale, double dYScale, const CPt2D<T> &ptRef);

private:
 vector<CPt2D<T> > m_vpt; // vertices of the polygon
};
```

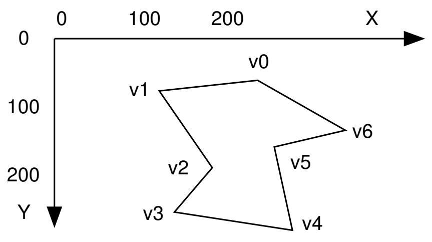


Figure 8: An example of a polygon.

Function	Description
operator =	Assignment operator
Count	Get the number of vertices of the polygon
Vertex	Get the idx-th vertex of the polygon
VertexArray	Get a reference to the vector that stores the vertices
SetPolygon	Set the polygon
AddVertex	Add a vertex at the end of the vertex list
SetVertex	Set the idx-th vertex value
InsertVertex	Insert a vertex at idx position of the vertex list
InsertVertex	Insert vertices before the one indexed by idx Vertices are copied from ppt and inserted to the position before idx. nStart and nEnd are indices specifying a section in ppt to copy from. The section includes nStart and nEnd. If nStart <= nEnd, vertices are copied in the same order as in ppt; if nStart > nEnd, vertices are copied in the reverse order as in ppt.
RemoveVertex	Remove the idx-th vertex from the vertex list
RemoveNearbyVertices	Remove vertices too close together, which makes sure that the length of each edge is >= dDist.
Clear	Remove all vertices of the polygon
Center	Get the centroid of the polygon
BoundingRect	Get the bounding box of the polygon

continued on next page

Offset	Offset the position of the polygon
PtIn	Check if a point is inside the polygon
Rotate	Rotate the polygon with respect to a point
Rescale	Rescale the polygon with respect to a reference point

2.10 CRing2D: 2D ring class

```
template <class T> class CRing2D
{
public:
 CRing2D() { }
 CRing2D(const CRing2D<T> &ring);
 CRing2D(const vector<CPt2D<T> > &vptO);
 CRing2D(const CPt2D<T> *ppt, int nCount);
 CRing2D<T>& operator = (const CRing2D<T> &ring);

 int CountIRim() const { return int(m_vvptI.size()); }
 const vector<CPt2D<T> >& ORim()      const { return m_vptO;     }
       vector<CPt2D<T> >& ORim()            { return m_vptO;     }
 const vector<CPt2D<T> >& IRim(int i) const { return m_vvptI[i]; }
       vector<CPt2D<T> >& IRim(int i)       { return m_vvptI[i]; }
 void     SetRing(      vector<CPt2D<T> > *pvpt);
 void     SetORim(const vector<CPt2D<T> > &vpt);
 void     SetORim(      vector<CPt2D<T> > *pvpt);
 void     SetIRim(const vector<CPt2D<T> > &vpt);
 void     AddIRim(      vector<CPt2D<T> > *pvpt);
 void     AddIRim(const CPt2D<T> *ppt, int nCount);
 void     Clear();
 void     Swap(CRing2D<T> &ring);
 double   Area()         const;
 CPt2D<T> Center()       const;
 CRect<T> BoundingRect() const;
 void     Offset   (T x, T y);
 void     Offset   (const CPt2D<T> &ptOffset);
 int      PtIn     (T x, T y)           const;
 int      PtIn     (const CPt2D<T> &pt) const;
 void     Rotate   (const CPt2D<T> &ptRot, double dAngle);
 void     Rescale  (double dXScale, double dYScale,
                    const CPt2D<T> &ptRef);
 void     GenIRim  (T width, int nHFL);
 int      Split    (double dA, double dB, double dC,
                    vector<CRing2D<T> > *pvRing);
 void     Partition(double dLimit, vector<CRing2D<T> > *pvRing);

private:
 vector<CPt2D<T> >          m_vptO;  // outer rim  of the ring
 vector<vector<CPt2D<T> > > m_vvptI; // inner rims of the ring
};
```

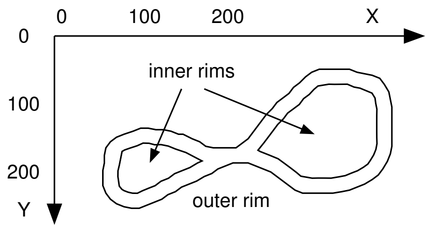


Figure 9: An example of a ring.

Function	Description
operator =	Assignment operator
CountIRim	Get the number of inner rims
ORim	Get the outer rim
IRim	Get an inner rim
SetRing	Set the ring
SetORim	Add the outer rim
AddIRim	Add an inner rim
Clear	Clear outer and inner rims
Swap	Swap two rings
Area	Get the area of the ring
Center	Get the centroid of the ring
BoundingRect	Get the bounding box of the ring
Offset	Offset the position of the ring
PtIn	Check if a point is inside the ring
Rotate	Rotate the ring with respect to a point
Rescale	Rescale the ring with respect to a reference point

GenIRim	Generate inner rims
width	width of the ring
nHFL	half-filter-length for ShiftCurve(..)

Split	Split the ring by a line If the ring does not have self-intersecting polygon and intersects the line, it will be empty after splitting and resultant rings are stored in pvRing. If no intersection, pvRing will be empty and *this is unchanged. If the ring has self-intersecting polygons and intersects the line, partial split may occur: neither pvRing nor *this will be empty.
dA, dB, dC	parameters of the line: A*x + B*y = C
pvRing	resultant rings
Return	1: the line splits the ring into multiple rings; 0: the line does not intersect the ring or the ring is ill-shaped and cannot be split; -1: the ring may be self-intersecting and split is partially done: neither pvRing nor *this is empty.

Partition	Partition the ring into multiple rings Partition the ring so that resultant rings are not longer than dLimit. The input pvRing should be empty; on output it stores partitioned rings.
dLimit	length limit
pvRing	partitioned rings are added to pvRing

2.11 CNCSpl: a natural cubic spline class

```
template <class T> class CNCSpl
{
public:
 CNCSpl() { }
 CNCSpl(const CNCSpl<T> &ncspl);
 CNCSpl<T>& operator = (const CNCSpl<T> &ncspl);

 int      Count() const;
 CPt2D<T> Knot(int idx) const;
 void     AddKnot   (         const CPt2D<T> &pt);
 void     SetKnot   (int idx, const CPt2D<T> &pt);
 void     InsertKnot(int idx, const CPt2D<T> &pt);
 void     RemoveKnot(int idx);
 void     Clear();
 CPt2D<T> Center() const;
 void     Offset(T x, T y);
 void     Offset(const CPt2D<T> &ptOffset);
 void     Rotate(const CPt2D<T> &ptRot, double dAngle);
 bool     GetPointAt(int nSec, double dFra, CPt2D<T> *ppt) const;
 void     Rescale(double dXScale, double dYScale, const CPt2D<T> &ptRef);
};
```

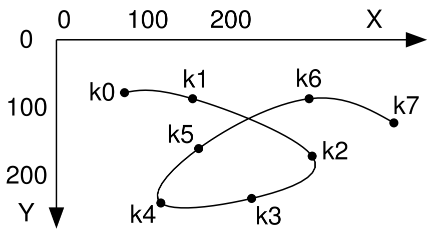


Figure 10: An example of a natural cubic spline.

Function	Description
operator =	Assignment operator
Count	Get the number of knots of the spline
Knot	Get the idx-th knot of the spline
AddKnot	Add a knot to the knot list
SetKnot	Set the idx-th knot value
RemoveKnot	Remove the idx-th knot from the knot list
Clear	Remove all knots of the spline
Center	Get the centroid of all knots
Offset	Offset the position of all knots
Rotate	Rotate the spline with respect to a point
Rescale	Rescale the spline with respect to a reference point

GetPointAt	Get a point at a specified place on the spline
nSec	the index of the section of the spline For example: i is the section between knot i and knot i+1.
dFra	fractional distance [0, 1] from the point to the start point of the section in x direction. For example: assume i-th section is chosen; fFraction = 0, returns the start point (x[i], y[i]). fFraction = 1, returns the end point (x[i+1], y[i+1]). fFraction = 0.3, returns the point (0.7*x[i]+0.3*x[i+1], y). y is calculated from the spline. dFra can also be set < 0 or > 1. In these cases, the point's x coordinate is < x[i] or > x[i+1].
ppt	stores output point if found
Return	true: point found; false: spline is not valid or nSec out of range, in that case ppt is unchanged.

2.12 CPackRc: a rectangle-packing class

```
enum EPackMode
{
 eBin,   // bin packing
 eStrip  // strip packing
};

enum EPackSequence
{
 eInput  = 0,  // input sequence
 eArea   = 1,  // area-descending sequence
 eHeight = 2,  // height-descending sequence
 eWidth  = 3,  // width-descending sequence
};

template <class T> class CPackRc
{
public:
 CPackRc();

 void Pack(EPackMode eMode, EPackSequence eSeq, int nBinW,
           int nBinH, const CSize2D<int> *pSz, int nCount);
 int  GetNumBins() const;
 int  GetBinH()    const;
 void GetPackedInfo( vector<CPt2D<int> > *pvpt,
                     vector<CRect<int> > *pvRc) const;
 void GetPackedRc(vector<vector<CRect<int> > > *pvvRc) const;
};
```

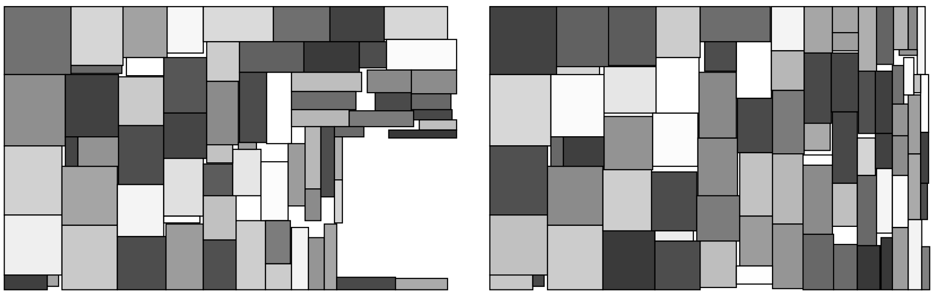


Figure 11: Two examples of rectangle packing.

Pack	Pack rectangles
eMode	bin packing or strip packing
eSeq	sequence of rectangles
nBinW	width of a bin or a strip container
nBinH	height of a bin (ignored in strip packing)
pSz	an array of rectangles' size
nCount	size of pSz

GetNumBins	Get the number of bins used in packing. In strip packing, the function always return 1 as only one bin is used.
GetBinH	Get bin height. In strip packing, it is the packed height. In bin packing, it is the specified bin height in Pack(..).

GetPackedInfo	Get information of the packed rectangles
pvpt	each packed rect's original index (x coord) and index of the bin it belongs to (y coord)
pvRc	each packed rect's position

GetPackedRc	Get the packed rectangles
pvvRc	packed rectangles. The outer list refers to bins. The inner list contains rectangles of each bin.

2.13 Distance

```
template <class T> double Distance
(const CPt2D<T> &pt1, const CPt2D<T> &pt2);
```

Description	Get the distance between two points
pt1	1st input point
pt2	2nd input point
Return	the distance between the two points

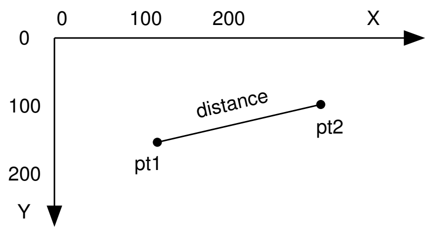


Figure 12: An example of the Distance function. Distance between two points.

2.14 DistPtLn

```
template <class T> double DistPtLn
(const CPt2D<T> &pt, double dA, double dB, double dC);
```

Description	Get the distance between a point and a line Line equation: A*x + B*y = C
pt	input point
dA, dB, dC	parameters of the line
Return	the distance between the point and the line

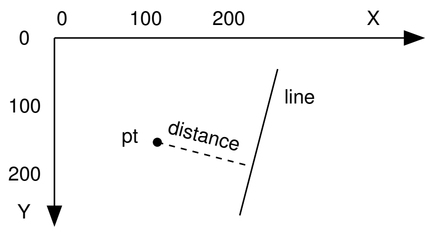


Figure 13: An example of the DistPtLn function. Distance between a point and a line.

2.15 DistLnLn

```
template <class T> double DistLnLn
(const CLine2D<T> &line1, const CLine2D<T> &line2);
```

Description	Get the distance between two line segments If the line segments intersect, the distance is 0; otherwise it is the closest distance from any point on a line segment to any point on the other.
line1	line segment 1
line2	line segment 2
Return	the distance between the two line segments

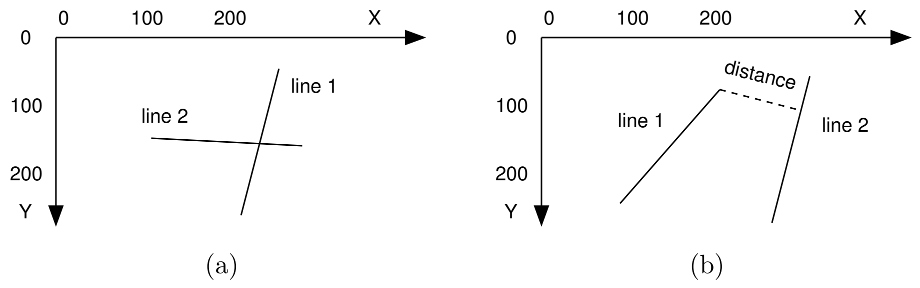


Figure 14: An example of the DistLnLn function. (a) Distance between two intersected line segments is zero. (b) If no intersection, the distance is the closest distance from any point on a line segment to that of the other.

2.16 FootOfPerpendicular

```
template <class T> CPt2D<T> FootOfPerpendicular
(const CPt2D<T> &pt, double dA, double dB, double dC);
```

Description	Get the foot of perpendicular from a point to a line. Line equation: A*x + B*y = C
pt	input point
dA, dB, dC	parameters of the line
Return	foot of perpendicular

2.17 MidPoint

```
template <class T> CPt2D<T> MidPoint
(const CPt2D<T> &pt1, const CPt2D<T> &pt2);
```

Description	Get the midpoint of two points
pt1	1st input point
pt2	2nd input point
Return	the midpoint

2.18 Area2

```
template <class T> double Area2
(const CPt2D<T> &pt1, const CPt2D<T> &pt2, const CPt2D<T> &pt3);
```

Description	Get twice the area enclosed by three points
pt1	1st input point
pt2	2nd input point
pt3	3rd input point
Return	twice the area enclosed by three points Use double type to prevent overflow. area < 0: pt1, pt2, pt3 are clockwise. area > 0: pt1, pt2, pt3 are anticlockwise. area = 0: three points are collinear.

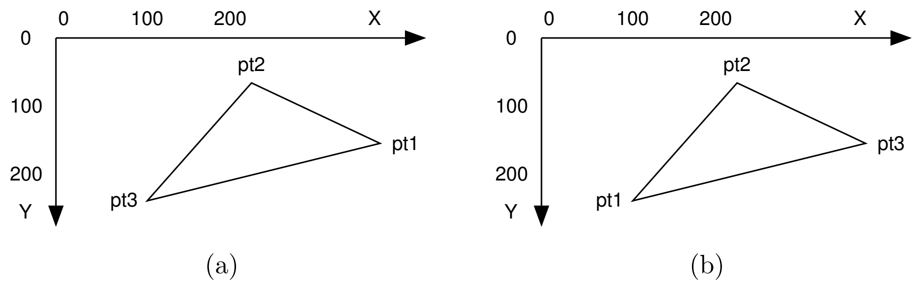


Figure 15: An example of the Area2 function. (a) Positive area. (b) Negative area.

```
template <class T> double Area2
(const CPt2D<T> *ppt, int nCount);
```

Description	Get twice the area enclosed by a polygon The polygon should be non-self-intersecting.
ppt	polygon vertices
nCount	size of ppt
Return	twice the area of a polygon Use double type to prevent overflow. area < 0: polygon vertices are clockwise. area > 0: polygon vertices are anticlockwise. area = 0: polygon vertices are on a poly line.

2.19 Angle

```
template <class T> double Angle
(const CPt2D<T> &pt1, const CPt2D<T> &pt2, const CPt2D<T> &pt3);
```

Description	Get the angle in $[-\pi, \pi]$, formed by pt1 $\leftarrow$ pt2 $\rightarrow$ pt3
pt1	1st input point
pt2	2nd input point (vertex of the angle)
pt3	3rd input point
Return	the angle formed by pt1 $\leftarrow$ pt2 $\rightarrow$ pt3 angle < 0: pt1, pt2, pt3 are clockwise. angle > 0: pt1, pt2, pt3 are anticlockwise. angle $= 0$, π or $-\pi$: three points are collinear.

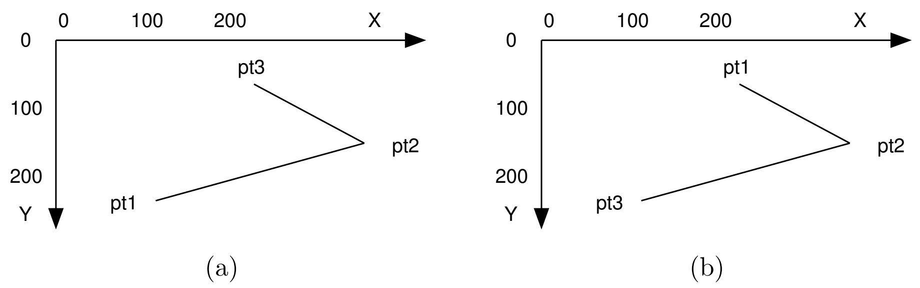


Figure 16: An example of the Angle function. (a) Positive angle. (b) Negative angle.

2.20 BoundingRect

```
template <class T> CRect<T> BoundingRect
(const CPt2D<T> *pptVtx, int nCount);
```

Description	Get the bounding rectangle of a set of points The bounding rectangle's left, right, top and bottom equal the min x, max x, min y and max y coordinates of the input points respectively.
ppt	an array of points
nCount	size of ppt
Return	the bounding rectangle

2.21 Centroid

```
template <class T> CPt2D<T> Centroid
(const CPt2D<T> *ppt, int nCount);
```

Description	Get the centroid of a set of points
ppt	an array of points
nCount	size of ppt
Return	the centroid of the input points

```
template <class T1, class T2> CPt2D<T1> Centroid
(const CPt2D<T1> *ppt, T2 *pWei, int nCount);
```

Description	Get the centroid of a set of points with weight
pWei	weight of each point

2.22 Length

```
template <class T> double Length
(const CPt2D<T> *ppt, int nCount);
```

Description	Get the length a poly line
ppt	vertices of the poly line
nCount	size of ppt
Return	length of the poly line

2.23 GetEdgePointAt

```
template <class T> int GetEdgePointAt
(const CPt2D<T> *ppt, int nCount, double dDist, CPt2D<T> *pptEdge);
```

Description	Get a poly line edge point that has a specified distance from the start point. The edge point may lie on a poly line segment or may be a vertex.
ppt	vertices of the poly line
nCount	size of ppt
dDist	distance of the edge point from the start point
pptEdge	the edge point found
Return	index of the start point of the line segment (edge) that contains the edge point. If dDist $<$ 0, return -1; if dDist $>$ length of the poly line, return nCount-1.

2.24 ClosestPoint

```
template <class T1, class T2> int ClosestPoint
(const CPt2D<T1> *ppt, int nCount, const CPt2D<T2> &pt, double *pd2 = 0);
```

Description	Search for the closest point to a given point
ppt	a point array where the closest point is searched
nCount	size of ppt
pt	the given reference point
pd2	store the square of distance from pt to the closest point in ppt. Input 0 to ignore.
Return	0-based index of the closest point in ppt to pt

2.25 ClosestPointPair

```
template <class T> void ClosestPointPair
(const CPt2D<T> *ppt, int nCount, int *pnIdx1, int *pnIdx2);
```

Description	Search for the closest point pair in a point set Index of the two points are stored in pnIdx1 and pnIdx2. *pnIdx1 is smaller than *pnIdx2 unless nCount==1; if so, both are 0.
ppt	point set
nCount	size of ppt
pnIdx1	index of the 1st point of the closest pair
pnIdx2	index of the 2nd point of the closest pair

```
template <class T> void ClosestPointPair
(const CPt2D<T> *ppt1, int nCount1, const CPt2D<T> *ppt2, int nCount2,
 int *pnIdx1, int *pnIdx2);
```

Description	Search in two set of points for the closest pair, one from each set.
ppt1	point set 1
nCount1	size of ppt1
ppt2	point set 2
nCount2	size of ppt2
pnIdx1	index of the closest point in ppt1
pnIdx2	index of the closest point in ppt2

2.26 ClosestEdgePoint

```
template <class T> int ClosestEdgePoint
(const CPt2D<T> *ppt, int nCount, bool bClosed, const CPt2D<T> &pt,
 CPt2D<T> *pptEdge, double *pdDist);
```

Description	Get the closest edge point of a polygon or a poly line to a point. The edge point may lie on a polygon or a poly line edge or may be a vertex.
ppt	vertices of a polygon or a poly line
nCount	size of ppt
bClosed	true for polygon, false for poly line
pt	reference point
pptEdge	the edge point found
pdDist	distance from pt to *pptEdge
Return	index of the start point of the closest line segment (edge) to the reference point

2.27 PointInPolygon

```
template <class T1, class T2> int PointInPolygon
(const CPt2D<T1> *ppt, int nCount, const CPt2D<T2> &pt);
```

Description	Check if a point is inside, on, or outside a polygon. Reference “Computational geometry in C” 7.4, Joseph O’Rourke. Ray-crossing algorithm. (Winding number algo is slower.)
ppt	polygon vertices
nCount	size of ppt
pt	point to be checked
Return	0: outside, 1: strictly inside, 2: on one of the edges but not on a vertex, 3: on one of the vertices.

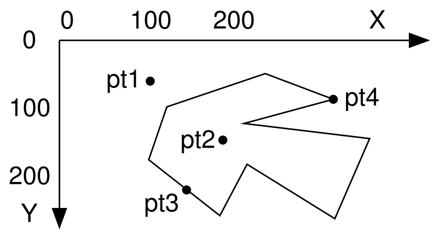


Figure 17: An example of the PointInPolygon function. pt1 is outside the polygon; pt2 is strictly inside; pt3 is on one of the edges but not on a vertex; pt4 is on one of the vertices.

2.28 Intersect

```
template <class T1, class T2> int Intersect
(const CLine2D<T1> &line1, const CLine2D<T1> &line2);
```

Description	Check if two line segments intersect
line1	1st input line segment
line2	2nd input line segment
Return	-2: invalid line parameters (e.g. two end points of a line are the same point), -1: two lines are the same line, or two line segments have overlapping sections. *ppt is unchanged. 0: no intersection (two lines are parallel), 1: intersection found. If the intersection is an end point of a line segment, the following bits of the return value are set. If so, the returned value is not 1. 3: intersection is line1.Start(), the second bit is set, 5: intersection is line1.End(), the third bit is set, 9: intersection is line2.Start(), the fourth bit is set, 17: intersection is line2.End(), the fifth is set.

```
template <class T1, class T2> int Intersect
(const CLine2D<T1> &line1, const CLine2D<T1> &line2, CPt2D<T2> *ppt);
```

Description	Get the intersection of two line segments (see above for details)
ppt	store the intersecting point if found; otherwise, its value is unchanged.

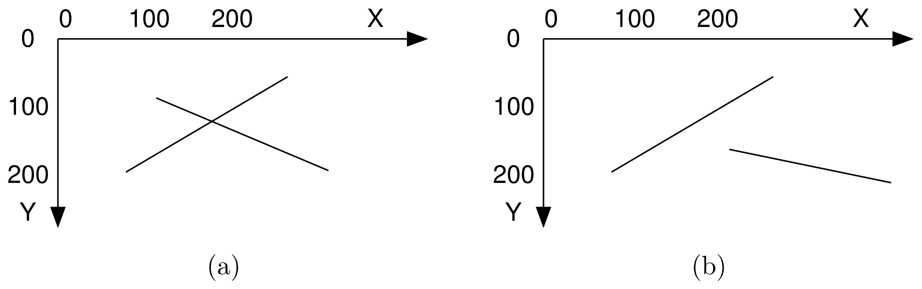


Figure 18: An example of the line segment Intersect function. (a) Two line segments intersect. (b) Two line segments do not intersect.

```
template <class T1, class T2> int Intersect
(const CLine2D<T1> &line1, double dA2, double dB2, double dC2,
 CPt2D<T2> *ppt);
```

Description	Get the intersection of a line segment with a line Line equation: A*x + B*y = C
line1	the line segment
dA2, dB2, dC2	parameters of the line
ppt	store the intersecting point if found. One of the line is derived from the line segment. If return true, the point is also on the line segment.
Return	-2: invalid line parameters (e.g. two end points of a line are the same point or dA2 = dB2 = 0), -1: two lines are the same line (*ppt is unchanged), 0: no intersection (two lines are parallel), 1: intersection found. If the intersection is an end point of a line segment, the following bits of the return value are set. If so, the returned value is not 1. 3: intersection is line1.Start(), the second bit is set, 5: intersection is line1.End(), the third bit is set.

```
template <class T> int Intersect
(double dA1, double dB1, double dC1,
(double dA2, double dB2, double dC2, CPt2D<T> *ppt);
```

Description	Get the intersection of two lines Line equation: A*x + B*y = C
dA1, dB1, dC1	parameters of the 1st input line
dA2, dB2, dC2	parameters of the 2nd input line
ppt	store the intersecting point if found; otherwise, its value is unchanged.
Return	-2: invalid line parameters (e.g. dA1 = dB1 = 0), -1: two lines are the same line (*ppt is unset), 0: no intersection (two lines are parallel), 1: intersection found.

2.29 IntsecLnRc

```
template <class T1, class T2> int IntsecLnRc
(const CLine2D<T1> &line, const CRect<T1> &rect,
 CPt2D<T2> *ppt1, CPt2D<T2> *ppt2)
```

Description	Get the intersections of a line segment with a rectangle If only one intersection is found, it is stored in ppt1. If two intersections are found, the one closer to the start of the line is ppt1. Improper intersections (touching) are treated as intersections. The x and y coordinates of *ppt1 and *ppt2 are within [left, right] and [top, bottom].
line	line segment
rect	rectangle
ppt1	1st intersection if exist
ppt2	2nd intersection if exist
Return	the number of intersections found (0, 1 or 2)

```
template <class T1, class T2> int IntsecLnRc
(double dA, double dB, double dC, const CRect<T1> &rect,
 CPt2D<T2> *ppt1, CPt2D<T2> *ppt2)
```

Description	Get two intersections of a line with a rectangle The x and y coordinates of *ppt1 and *ppt2 are within [left, right] and [top, bottom].
dA, dB, dC	parameters of the line: A*x + B*y = C
rect	rectangle
ppt1	1st intersection if exist
ppt2	2nd intersection if exist
Return	true: two intersections found; false: not found.

```
template <class T> bool IntsecLnRc
(double dA, double dB, double dC, const CRect<T> &rect,
 CLine2D<T> *pLine);
```

Description	Intersect a line with a rectangle The x and y coordinates of the resultant line segment's end points are within [left, right] and [top, bottom].
dA, dB, dC	parameters of the line: A*x + B*y = C
rect	rectangle
pLine	store the intersected line segment in rect if exists.
Return	true: pLine is in rect; false: input line does not intersect rect.

```
template <class T> bool IntsecLnRc
(const CRect<int> &rect, CLine2D<T> *pLine);
```

Description	Intersect a line segment with a rectangle A line segment is cut to fit inside a rect (integer type). The x and y coordinates of its end points are within [left, right) and [top, bottom). Note that the right and bottom borders are not inside.
rect	rectangle
pLine	on input, a line segment; on output, the intersected line segment in rect if exists; otherwise unchanged.
Return	true: pLine is in rect; false: pLine is out of rect.

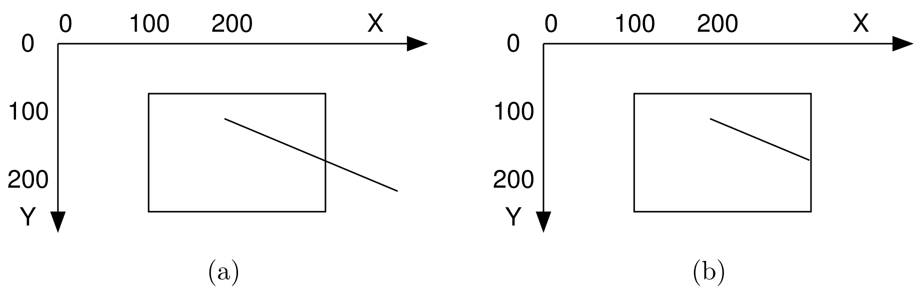


Figure 19: An example of the last version of IntsecLnRc function. (a) Before applying IntsecLnRc(...). (b) After applying IntsecLnRc(...).

2.30 IntsecLnPy

```
template <class T1, class T2> bool IntsecLnPy
(const CLine2D<T1> &line, const CPt2D<T1> *pptPy, int nCount,
 bool bClosed, int nIndex, CPt2D<T2> *ppt, int *pnIdx = 0);
```

Description	Get one intersection of a line segment with a polygon or a poly line. If the line segment or the line only touches an edge or a vertex of the polygon or the poly line, the contacting point is not considered as an intersection.
line	line segment
pptPy	vertices of the polygon or the poly line
nCount	size of ppt
bClosed	true for polygon, false for poly line
nIndex	index of the intersection to output. 1: the first intersection is output, 2: the second, etc; -1: the last, -2: the second last, etc. nIndex cannot be 0.
ppt	store an intersection
pnIdx	store index of the intersecting line segment on pptPy. For example, pptPy[*pnIdx] → [*pnIdx+1] intersects the input line or line segment. If no intersection, its value is unchanged.
Return	true: intersection found; false: no intersection.

```
template <class T1, class T2> bool IntsecLnPy
(double dA, double dB, double dC, const CPt2D<T1> *pptPy, int nCount,
 bool bClosed, int nIndex, CPt2D<T2> *ppt, int *pnIdx = 0);
```

Description	Get one intersection of a line with a polygon or a poly line
dA, dB, dC	parameters of the line: A*x + B*y = C

```
template <class T1, class T2> bool IntsecLnPy
(const CLine2D<T1> &line, const CPt2D<T1> *pptPy, int nCount,
 bool bClosed, vector<CPt2D<T2> > *pvpt, vector<int> *pvIdx = 0);
```

Description	Get all intersections of a line segment with a polygon or a poly line
pvpt	store intersections
pvIdx	store indices of the intersecting line segments on the poly line. For example, pptPy[pvIdx[0]] to pptPy[pvIdx[0]+1] is the 1st line segment that intersects the input line. Input 0 to ignore.
Return	the number of intersections found

```
template <class T1, class T2> bool IntsecLnPy
(double dA, double dB, double dC, const CPt2D<T1> *pptPy, int nCount,
 bool bClosed, vector<CPt2D<T2> > *pvpt, vector<int> *pvIdx = 0);
```

Description	Get all intersections of a line with a polygon or a poly line
dA, dB, dC	parameters of the line: A*x + B*y = C

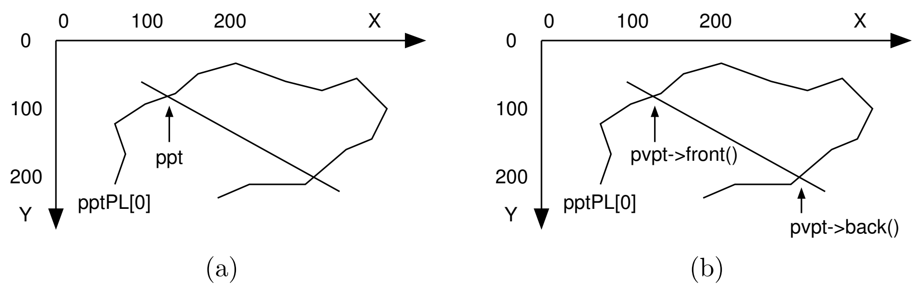


Figure 20: An example of the poly line Intersect function. (a) Locate the first intersection. (b) Locate all intersections.

2.31 IntsecPyPy

```
template <class T1, class T2> bool IntsecPyPy
(const CPt2D<T1> *ppt1, int nCount1, bool bClosed1,
 const CPt2D<T1> *ppt2, int nCount2, bool bClosed2,
 vector<CPt2D<T2> > *pvpt, vector<int> *pvIdx1 = 0, vector<int> *pvIdx2 = 0);
```

Description	Get the intersections of two polygons or poly lines Get the intersections of two polygons or poly lines. pvIdx1 and pvIdx2 store the index of the start point of the line segments in ppt1 and ppt2 that intersect each other. A line segment whose end point is the intersection and improper intersections (touching but not crossing) are ignored.
ppt1	vertices of the first polygon or poly line
nCount1	size of ppt1
bClosed1	if true, ppt1 is a polygon; or, a poly line.
ppt2	vertices of the second polygon or poly line
nCount2	size of ppt2
bClosed2	if true, ppt2 is a polygon; or, a poly line.
pvpt	intersections (along the sequence of ppt1)
pvIdx1	indices of the start points of the line segments in ppt1 that intersect ppt2. The indices are in order, from the start to the end of ppt1.
pvIdx2	indices of the start points of the line segments in ppt2 that intersect ppt1. The indices are not in order; each corresponds to the intersecting line segment in pvpt and pvIdx1.
Return	the number of intersections found

2.32 DensifyVertex

```
template <class T> void DensifyVertex
(vector<CPt2D<T> > *pvpt, T dist, bool bClosed);
```

Description	Densify vertex of a polygon or a poly line Add vertices so that the distance between each pair of adjacent vertices is not larger than dist.
pvpt	input vertices.On output, vertices are added wherever needed.
dist	maximum distance between vertices
bClosed	true for polygon, false for poly line

2.33 ReduceVertex

```
template <class T> void ReduceVertex
(vector<CPt2D<T> > *pvpt, double dAngThre, bool bClosed);
```

Description	Reduce the number of vertices of a polygon or a poly line. If the angle formed by a vertex and its adjacent neighbours differs from 0, PI or -PI by an amount less than dAngThre, the vertex is removed from the polygon or the poly line.
pvpt	input vertices. On output, vertices are removed based on the above criterion.
dAngThre	angular threshold
bClosed	true for polygon, false for poly line

2.34 RemoveSpike

```
template <class T> void RemoveSpike
(vector<CPt2D<T> > *pvpt, int nScanLen);
```

Description	Remove spikes on a polygon Within the scan length before and after each vertex, spikes are identified and removed; so that the closest points to this vertex in this range are its adjacent neighbours.
pvpt	input vertices of a polygon. On output, spikes are removed if any.
nScanLen	scan length for spike before and after a vertex

2.35 RoundCorner

```
template <class T> void RoundCorner
(vector<CPt2D<T> > *pvpt, T radius);
```

Description	Convert polygon or poly line vertices to round corners Each vertex is replaced by a number of vertices to round the vertex corner. If a vertex's two edges are too short, it is not converted to a round corner.
pvpt	input vertices. On output, vertices are converted to round corners if possible.
radius	radius of the round corner arc
bClosed	true for polygon, false for poly line

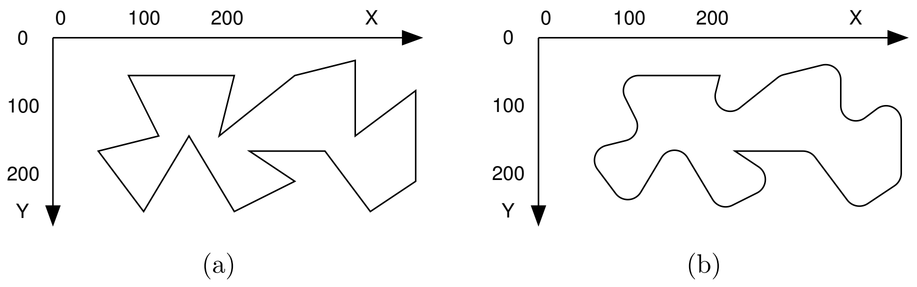


Figure 21: An example of the RoundCorner function. (a) Original polygon. (b) The polygon after processed by RoundCorner(...) with radius 10. Two vertices are not converted because their edges are too short.

2.36 SmoothVertex

```
template <class T> void SmoothVertex
(vector<CPt2D<T> > *pvpt, int nFltSize, bool bClosed);
```

Description	Smooth polygon or poly line vertices
pvpt	input vertices. On output, vertices are smoothed.
nFltSize	filter kernel size (e.g. 3: 3-point mean filter)
bClosed	true for polygon, false for poly line

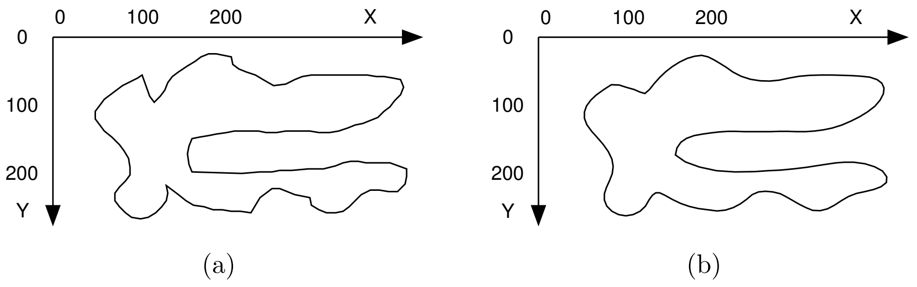


Figure 22: An example of the SmoothVertex function. (a) Original polygon. (b) The polygon after processed by SmoothVertex(...) with nFltSize 5.

2.37 TriangulatePolygon

```
template <class T> int TriangulatePolygon
(const CPt2D<T> *ppt, int nCount, vector<int> *pvIdx);
```

Description	Triangulate a simple polygon Reference "Computational geometry in C", Joseph O'Rourke.
ppt	vertices of the polygon
nCount	size of ppt
pvIdx	store indices of the triangles' vertices in ppt
Return	the number of triangles generated

```
template <class T> int TriangulatePolygon
(const CPt2D<T> *ppt, int nCount, vector<CPt2D<T> > *pvpt);
```

Description	Triangulate a simple polygon
vpt	store generated triangles' vertices

2.38 ConvexHull

```
template <class T> int ConvexHull
(const vector<CPt2D<T> > &vpt, vector<int> *pvIdx);
```

Description	Get convex hull points' index
Precondition	see `ConvexHull(vector<CPt2D<T> >*, vector<CPt2D<T> >*)`
vpt	input point set
pvIdx	indices of the convex hull points in vpt (clockwise)
Return	the number of convex hull points

```
template <class T> int ConvexHull
(vector<CPt2D<T> > *pvpt, vector<CPt2D<T> > *pvCH);
```

Description	Get the convex hull points in clockwise order Reference "Computational geometry in C", Joseph O'Rourke. The first point of pvpt must be the topmost of all points, meaning that it has the smallest y coordinate. If there are more than one topmost points, the first point should be the leftmost of them, with the smallest x coordinate.
pvpt	input points (the array will be modified)
pvCH	output the convex hull points
Return	the number of convex hull points

2.39 TopLeftPoint

```
template <class T> int TopLeftPoint
(const CPt2D<T> *ppt, int nCount);
```

Description	Find the top-left point in a point array The top-left point has the smallest y coordinate. If there are more than one topmost points, take the left-most points amony them, which has the smallest x coordinate.
ppt	input point array
nCount	size of ppt
Return	index of the top-left point

2.40 BoundingBox

```
template <class T> bool BoundingBox
(const vector<CPt2D<T> > &vpt, double dAngle,
 CPt2D<double> *pptCenter, CSize2D<double> *pSize);
```

Description	Get the fixed-angle bounding box of a set of points
vpt	input object points
dAngle	angle of the bounding box (radius, anticlockwise)
pptCenter	center of the box
pSize	size of the box
Return	true: succeeded; false: failed.

2.41 MinBoundingBox

```
template <class T> bool MinBoundingBox
(const vector<CPt2D<T> > *pvpt, CPt2D<double> *pptCenter,
 CSize2D<double> *pSize, double *pdAngle);
```

Description	Get the min bounding box of a set of points MinBoundingBox function is based on the "rotating caliper" algorithm. The output angle is anticlockwise; while some internal angles are clockwise. Since ConvexHull(...) outputs points in clockwise order, it is convenient to use clockwise angle internally.
Precondition	see `ConvexHull(vector<CPt2D<T> >*, vector<CPt2D<T> >*)`
pvpt	input points (the array will be modified)
pptCenter	center of the box
pSize	size of the box
pdAngle	angle of the box (radius, anticlockwise)
Return	true: succeeded; false: failed.

2.42 MinBoundingCircle

```
template <class T> bool MinBoundingCircle
(const vector<CPt2D<T> > *pvpt, CPt2D<double> *pptCenter, double *pdRadius);
```

Description	Get the min bounding circle of a set of points
1	Call the convex hull of the set of points H. Pick any side of H, say S.
2	For each vertex of H other than those of S, compute the angle subtended by S. The minimum such angle, α, occurs at vertex v. If $\alpha \geq 90$ deg, done! (The circle is the diametric circle of S.) If $\alpha < 90$ deg, check the remaining vertices of the triangle formed by S and v.
3	If no vertices are obtuse, done! (The circle is determined by the vertices of S and the vertex v.) If one of the other angles of the triangle formed by S and v is obtuse, then set S to be the side opposite the obtuse angle and go to step 2. (The new S may not be a side on the convex hull.)
Precondition	see `ConvexHull(vector<CPt2D<T> >*, vector<CPt2D<T> >*)`
pvpt	input points (the array will be modified)
pptCenter	center of the circle
pdRadius	radius of the circle
Return	true: succeeded; false: failed.

2.43 DelaunayTriangulation

```
template <class T> int DelaunayTriangulation
(const vector<CPt2D<T> > &vpt,
 vector<CLine2D<T> > *pvLine, vector<CPoly2D<T> > *pvPoly);
```

Description	Get the Delaunay triangulation of a set of points Reference "Computational geometry in C", Joseph O'Rourke. Delaunay triangulation in 2D is based on convex hull in 3D.
vpt	input point set
pvLine	edges of Delaunay triangles (input 0 to ignore)
pvPoly	Delaunay triangles (input 0 to ignore). Each polygon contains only 3 vertices.
Return	the number of triangles

2.44 ConstructKDTree

```
template <class T> CKDNode<T,2>* ConstructKDTree
(const CPt2D<T> *ppt, int nCount);
```

Description	Construct a 2-D tree
ppt	point set to construct the tree
nCount	size of ppt
Return	pointer to the root node of the tree. The caller is responsible for deleting the pointer after use.

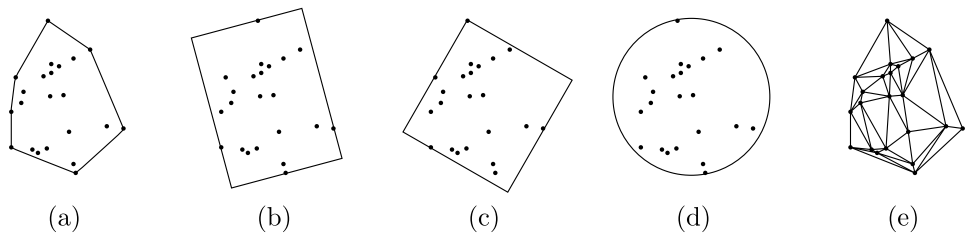


Figure 23: Examples of the ConvexHull, BoundingBox, MinBoundingBox, MinBoundingCircle and DelaunayTriangulation functions. (a) Convex hull. (b) Bounding box at a fixed angle of 15 deg. (c) Minimum-area bounding box. (d) Minimum-area bounding circle. (e) Delaunay triangulation.

2.45 KDTreePartition

```
template <class T> void KDTreePartition
(const CKDNode<T,2> *pNode, const CRect<int> &rect,
 vector<CLine2D<T> > *pvLine);
```

Description	Partition a region based on a 2-D tree
pNode	a tree node
rect	bounding rectangle of the region
pvLine	generated line partitions

2.46 FitLine

```
template <class T> bool FitLine
(const CPt2D<T> *ppt, int nCount, double *pdK, double *pd0, bool bYErr);
```

Description	Least squares fitting of a line Fit a line based on either y- or x-direction error. If y-direction error, the line equation is $y = kx + d0$; or the the line equation is $x = ky + d0$.
ppt	points to be fitted (size must be nCount)
nCount	size of ppt
pdK, pd0	parameters of the line equation
bYErr	true: y-direction, false: x-direction error.
Return	true: succeeded; false: failed.

```
template <class T> bool FitLine
(const CPt2D<T> *ppt, int nCount, double *pdA, double *pdB, double *pdC);
```

Description	Least squares fitting of a line Line equation: $Ax + By = C$. This function automatically determines if x- or y-direction error should be used when fitting a line. It compares the x and y coordinates of the first and last points. If the difference of the x coordinates is not smaller than that of the y coordinates, y-direction error is used; otherwise x-direction error is used.
ppt	points to be fitted (size must be nCount)
nCount	size of ppt
pdA, pdB, pdC	parameters of the line equation
Return	true: succeeded; false: failed.

```
template <class T> bool FitLine
(vector<CPt2D<T> > *pvpt, double dOutDist,
 double  dRMin,     double  dRMax,     int     nRSec,
 double  dAngleMin, double  dAngleMax, int     nAngleSec,
 double *pdA,       double *pdB,       double *pdC);
```

Description	Fit a line, not affected by outliers Least-squares line fitting is often affected by outliers: points that are far away from the real line. This function applies HoughLine(..) to find a line close to the real one, as Hough transform is insensitive to outliers. Then, based on a specified distance, outliers are removed. Finally, line fitting is applied to the remaining points. The input dRMin, dRMax, nRSec, dAngleMin, dAngleMax and nAngleSec are used in HoughLine(..).
pvpt	input points to be fitted. On output, the outliers are removed from the array.
dOutDist	outlier distance. Points, whose distance to the line found by HoughLine(..) is larger than dOutDist, are considered as outliers.
pdA, pdB, pdC	parameters of the line equation $Ax + By = C$
Return	true: succeeded; false: failed.

2.47 FitCircle

```
template <class T> bool FitCircle
(const CPt2D<T> *ppt, int nCount,
 CPt2D<double> *pptCenter, double *pdRadius);
```

Description	Least squares fitting of a circle Circle equation: $(x - a)^2 + (y - b)^2 = R^2$ Internally fit the modified equation: $Ax + By + C = x^2 + y^2$
ppt	points to be fitted (size must be nCount)
nCount	size of ppt
pptCenter	store the center of the circle if succeeded
pdRadius	store the radius of the circle if succeeded
Return	true: succeeded; false: failed.

2.48 FitEllipse

```
template <class T> bool FitEllipse
(const CPt2D<T> *ppt, int nCount, double pdCoe[5]);
```

Description	Least squares fitting of an ellipse Ellipse equation: $Ax^2 + Bxy + Cy^2 + Dx + Ey = 1$
ppt	points to be fitted (size must be nCount)
nCount	size of ppt
pdCoe	coefficients, A, B, C, D and E, of the ellipse equation
Return	true: succeeded; false: failed.

2.49 FitPolynomialCurve

```
template <class T> bool FitPolynomialCurve
(const CPt2D<T> *ppt, int nCount, vector<double> *pvCoe, int N);
```

Description	Least squares fitting of a polynomial curve Polynomial curve equation: $y = C[0] + coe[1] * x + C[2] * x^2 + ... + C[N] * x^N$
ppt	points to be fitted (size must be nCount)
nCount	size of ppt
pvCoe	stored coefficients. If succeed, its size is N+1.
N	the order of the curve equation (should be >= 0)
Return	true: succeeded; false: failed.

2.50 FitGauss

```
template <class T> bool FitGuass
(const CPt2D<T> *ppt, int nCount, double *pdA, double *pdB, double *pdC);
```

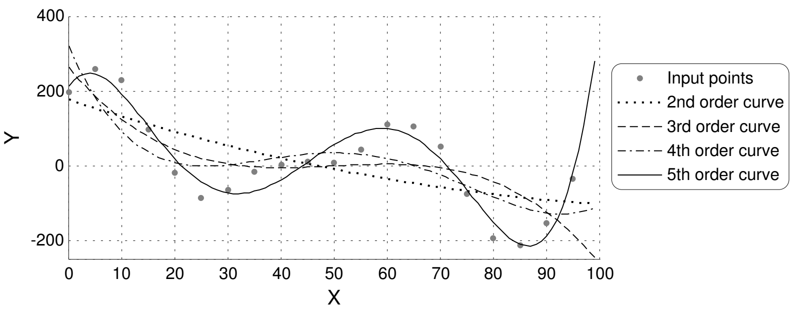


Figure 24: An example of the FitPolynomialCurve function.

Description	Least squares fitting of a Gaussian function Gaussian function: $y = Ae^{-(x-B)^2/(2C)}$. Fitting is based on a linear least squares method: the y coordinate of all points is converted to ln(y), which requires that y > 0.
ppt	points to be fitted (size must be nCount)
nCount	size of ppt
pdA	parameter A of the Gaussian equation
pdB	parameter B of the Gaussian equation
pdC	parameter C of the Gaussian equation
Return	true: succeeded; false: failed.

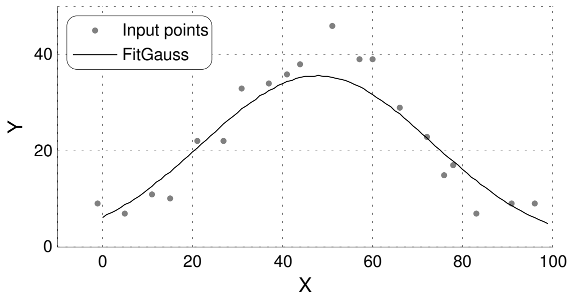


Figure 25: An example of the FitGauss function.

2.51 HoughLine

```
template <class T> int HoughLine
(const CPt2D<T> *ppt, int nCount,
 double dRMin,     double dRMax,     int nRSec,
 double dAngleMin, double dAngleMax, int nAngleSec,
 double *pdR, double *pdAngle);
```

Description	Hough transform line detection Detect one line with the maximum hough parameter. Line equation: R = x*cos(angle) + y*sin(angle) Resolution of R is (dRMax-dRMin)/nRSec. Resolution of angle is (dAngleMax-dAngleMin)/nAngleSec. If y axis points down, the angle is clockwise; otherwise, it is anticlockwise.
ppt	points to be analyzed (size must be nCount)
nCount	size of ppt
dRMin	minimum boundary of R [dRMin, dRMax)
dRMax	maximum boundary of R [dRMin, dRMax)
nRSec	number of sections in [dRMin, dRMax)

continued on next page

dAngleMin	minimum boundary of angle [dAngleMin, dAngleMax)
dAngleMax	maximum boundary of angle [dAngleMin, dAngleMax)
nAngleSec	number of sections in [dAngleMin, dAngleMax)
pdR	found R of the line equation
pdAngle	found angle (in radius) of the line equation
Return	0: line not found; >0 line found (return the number of points on the line).

2.52 HoughCircle

```
template <class T> int HoughCircle
(const CPt2D<T> *ppt, int nCount,
 double dXMin, double dXMax, int nXSec,
 double dYMin, double dYMax, int nYSec,
 double dRMin, double dRMax, int nRSec,
 CPt2D<double> *pptCenter, double *pdR);
```

Description	Hough transform circle detection Detect one circle with the maximum hough parameter. Circle equation: $R = (x - x0)^2 + (y - y0)^2$ Resolution of center x is (dXMax-dXMin)/nXSec. Resolution of center y is (dYMax-dYMin)/nYSec. Resolution of radius is (dRMax-dRMin)/nRSec.
ppt	points to be analyzed (size must be nCount)
nCount	size of ppt
dXMin	minimum boundary of the circle center x coordinate
dXMax	maximum boundary of the circle center x coordinate
dYMin	minimum boundary of the circle center y coordinate
dYMax	maximum boundary of the circle center y coordinate
dRMin	minimum boundary of R [dRMin, dRMax)
dRMax	maximum boundary of R [dRMin, dRMax)
nXSec	number of sections in [dXMin, dXMax)
nYSec	number of sections in [dYMin, dYMax)
nRSec	number of sections in [dRMin, dRMax)
pptCenter	found circle center
pdR	found circle radius
Return	0: circle not found; >0 circle found (return the number of points on the circle).

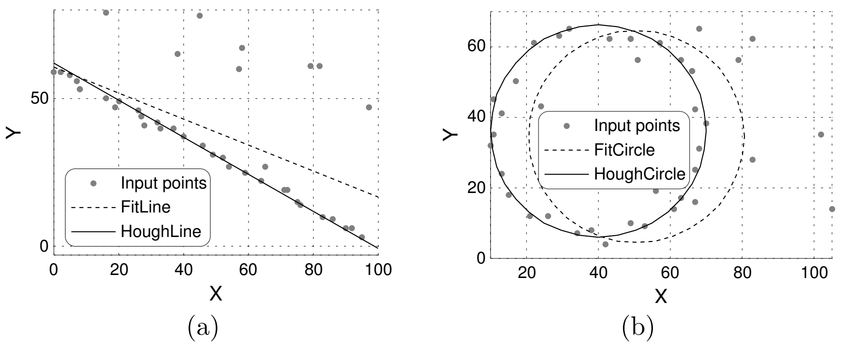


Figure 26: Examples of the FitLine, HoughLine, FitCircle and HoughCircle functions. (a) FitLine and HoughLine. (b) FitCircle and HoughCircle.

2.53 LinePoints

```
template <class T> int LinePoints
(CLine2D<T> &line, const CRect<int> &rcROI, vector<CPt2D<int> > *pvpt);
```

```
template <class T> int LinePoints
(const CPt2D<T> &ptStart, const CPt2D<T> &ptEnd,
 const CRect<int> &rcROI, vector<CPt2D<int> > *pvpt);
```

Description	Get points on a line within a bounding rect
line	input line, which may be modified.
ptStart	start point of the line
ptEnd	end point of the line
rcROI	bounding rect (only get points inside the rect)
pvpt	store points on the line
Return	the number of pixels on the line

```
template <class T> int LinePoints
(const CPt2D<T> &ptStart, const CPt2D<T> &ptEnd, vector<CPt2D<int> > *pvpt);
int LinePoints
(const CPt2D<int> &ptStart, const CPt2D<int> &ptEnd,
 vector<CPt2D<int> > *pvpt);
```

Description	Get points on a line Specialization for integer type based on Bresenham's line algorithm.
ptStart	start point of the line
ptEnd	end point of the line
pvpt	store points on the line
Return	the number of pixels on the line

2.54 CirclePoints

```
int CirclePoints
(const CPt2D<int> &ptCenter, int nRadius,
 vector<CPt2D<int> > *pvpt);
```

Description	Bresenham's circle algorithm
ptCenter	center of the circle
nRadius	radius of the circle
pvpt	store points on the circle
Return	the number of pixels on the circle

2.55 ShiftCurve

```
template <class T1, class T2> int ShiftCurve
(const CPt2D<T1> *ppt, int nCount, bool bClosed,
 const T1 *pShift, int nSize, int nHFL, bool bClockWise,
 vector<CPt2D<T2> > *pvpt);
```

Description	Shift a curve in its normal direction Different potions of the curve may have different shifts, which are specified by pShift. They are evenly distributed along the curve. The resultant curve may not have the same number of vertices.
ppt	vertices of the curve (poly line)
nCount	size of ppt
bClosed	true for polygon, false for poly line
pShift	an array of shifting distances
nSize	the pShift array size
nHFL	half-filter-length for fitting the tangential line to the curve
bClockWise	indicate if the shift is clockwise or anticlockwise. The order is determined by ppt[0] $\rightarrow$ ppt[nCount-1] $\rightarrow$ vpt.back() $\rightarrow$ vpt.front().
pvpt	store the shifted curve
Return	the number of vertices in pvpt

3 2D image processing functions

Some parameters are common to most functions. They are listed in Table 2.

Type	Name	Description
T*	pImg	Pointer to a continuous memory space which must be equal to or larger than w*h*sizeof(T) bytes. The y-th row, x-th column element can be retrieved by *(pImg+y*w+x) or pImg[y*w+x].
T*	pSrc	Pointer to the source image
T*	pDst	Pointer to the destination image
int	w, h	Image width and height
CRect<int>&	rcROI	Region of interest. Only image data within the ROI will be used or modified.
Pred	pred	a functional object. If pred(pSrc[y*w+x]) is true, (x,y) is an object point; otherwise, a background point.

Table 2: Common parameters of 2D image processing functions.

3.1 ImgAssign

```
template <class T> void ImgAssign
(T *pImg, int w, int h, const CRect<int> &rcROI, T value);
```

Description	Assign image data in an ROI to an input value
Parameters	See Table 2 for common parameters
value	value to be assigned to the image data

3.2 ImgAssignBorder

```
template <class T> void ImgAssignBorder
(T *pImg, int w, int h, const CRect<int> &rcROI,
 T value, int nLeft, int nTop, int nRight, int nBottom);
```

Description	Assign image data on the border of an ROI to an input value nLeft, nTop, nRight, nBottom: distance toward the center on the left, top, right and bottom border respectively. Pixels within the distance are assigned to the input value. The modified region is a rectangular ring.
Parameters	See Table 2 for common parameters
value	value to be assigned to the image data

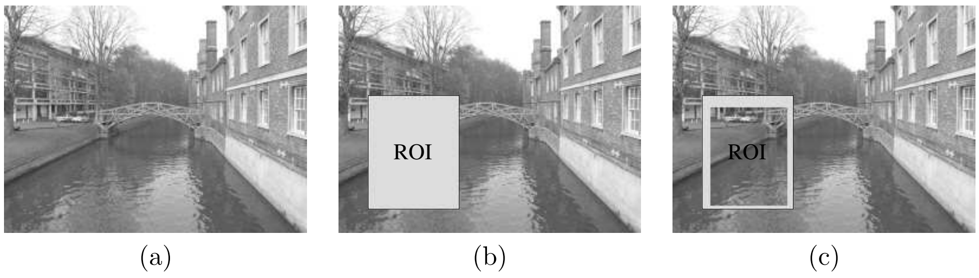


Figure 27: Examples of the ImgAssign and ImgAssignBorder functions. (a) Original image. (b) The ROI is assigned with value 220. (c) The border of the ROI is assigned with value 220. Left border: 7 pixels, top: 10 pixels, right: 5 pixels, bottom: 3 pixels.

3.3 ImgCopy

```
template <class T1, class T2> void ImgCopy
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2);
```

Description	Copy image data from source image ROI1 to destination image ROI2 Width and height of ROI1 and ROI2 must be the same.
Parameters	See Table 2 for common parameters

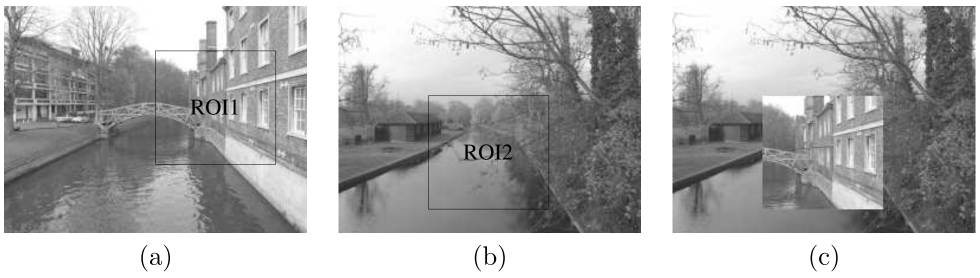


Figure 28: An example of the ImgCopy function. (a) Source image ROI1 is copied to (b) destination image ROI2. (c) Resultant image after ImgCopy.

3.4 ImgCopySubpixel

```
template <class T1, class T2> void ImgCopySubpixel
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2,
 const CPt2D<float> &ptOffset);
```

Description	Copy image data at subpixel accuracy Copy image ROI1 offset by ptOffset to ROI2. Each subpixel in ROI1 is linearly interpolated from interger pixels. Width and height of ROI1 and ROI2 must be the same.
Parameters	See Table 2 for common parameters
ptOffset	subpixel offset of ROI1

3.5 ImgBlend

```
template <class T1, class T2> void ImgBlend
(const T1 *pImg1, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pImg2, int w2, int h2, const CRect<int> &rcROI2,
 double dCoe1, double dCoe2);
```

Description	Blend img1 into img2 Blending equation: img2 = dCoe1*img1 + dCoe2*img2.
Parameters	See Table 2 for common parameters
dCoe1	coefficient multiplied to each pixel in ROI1
dCoe2	coefficient multiplied to each pixel in ROI2

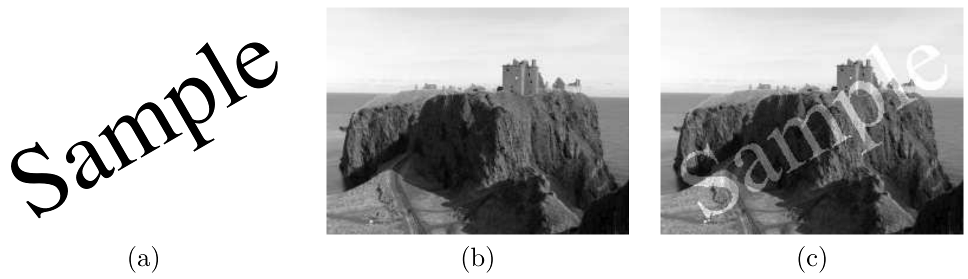


Figure 29: An example of the ImgBlend function. (a) Image 1. (b) Image 2. (c) The resultant image of blending image 1 into image 2.

3.6 ImgGradientX

```
template <class T1, class T2> void ImgGradientX
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2,
 int nGradLen);
```

Description	1D gradient in the x direction Take the image data gradient in the x direction. If the gradient length is 2, the x gradient pDst[x] = pSrc[x+1] - pSrc[x-1], meaning the gap between the two pixels used to calculate a gradient point is 2.
Parameters	See Table 2 for common parameters
nGradLen	length of the gradient operator

3.7 ImgGradientY

```
template <class T1, class T2> void ImgGradientY
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2,
 int nGradLen);
```

Description	1D gradient in the y direction Take the image data gradient in the y direction. If the gradient length is 2, the y gradient pDst[y*w+x] = pSrc[(y+1)*w+x] - pSrc[(y-1)*w+x], meaning the gap between the two pixels used to calculate a gradient point is 2.
Parameters	See Table 2 for common parameters
nGradLen	length of the gradient operator

3.8 ImgClamp

```
template <class T> void ImgClamp
(T *pImg, int w, int h, const CRect<int> &rcROI,
 T toMin, T toMax);
```

Description	Clamp image data to the range [toMin, toMax]
Parameters	See Table 2 for common parameters
toMin	minimum bound to clamp the image data
toMax	maximum bound to clamp the image data

3.9 ImgLinear

```
template <class T1, class T2> void ImgLinear
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2,
 T1 fromMin, T1 fromMax, T2 toMin, T2 toMax);
```

Description	Linear translate Src data in the range [fromMin, fromMax] to Dst data to the range [toMin, toMax]. Linear translation parameters a and b is determined by:
	a * fromMin + b = toMin;
	a * fromMax + b = toMax;
	Data points in Src that are < fromMin or > fromMax will be translate to toMin and toMax. pSrc and pDst may point to the same image buffer, if ROI1 and ROI2 are the same.

Parameters	See Table 2 for common parameters
fromMin	source minimum bound
fromMax	source maximum bound
toMin	destination minimum bound
toMax	destination maximum bound

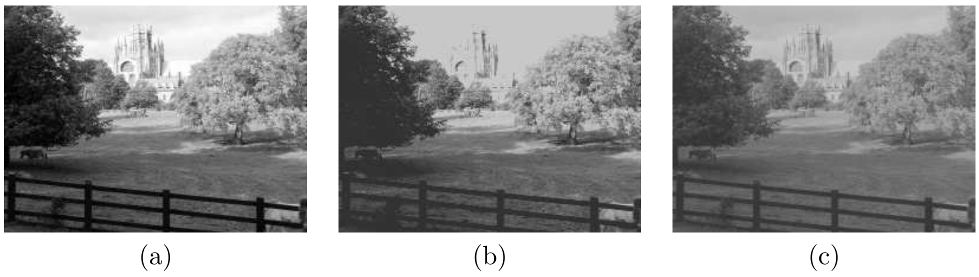


Figure 30: Examples of the ImgClamp and ImgLinear functions. (a) Source image. (b) Image data are clamped to [50, 200]. (c) Image data are linearly translated to [50, 200].

3.10 ImgGamma

```
template <class T1, class T2> void ImgGamma
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2,
 double dGamma);
```

Description	Apply gamma transform Gamma transform equation: $y = x^{gamma}$, where $x \in [0, 1]$. This function first maps all data from [min, max] to [0, 1]; then applies gamma transform; and finally maps the data back to [min, max]. pSrc and pDst may point to the same image buffer, if ROI1 and ROI2 are the same.
Parameters	See Table 2 for common parameters
dGamma	gamma value

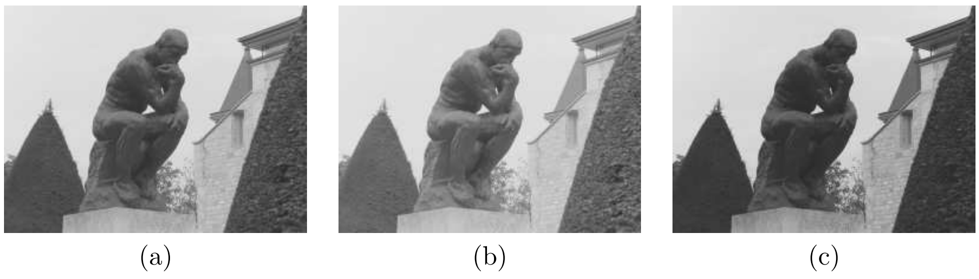


Figure 31: Examples of the ImgGamma function. (a) Source image. (b) The image after transformed by gamma = 0.7. (c) The image after transformed by gamma = 1.5.

3.11 ImgMin

```
template <class T> T ImgMin
(const T *pImg, int w, int h, const CRect<int> &rcROI);
```

Description	Get the minimum value of an ROI
Parameters	See Table 2 for common parameters
Return	the minimum value of an ROI

3.12 ImgMax

```
template <class T> T ImgMax
(const T *pImg, int w, int h, const CRect<int> &rcROI);
```

Description	Get the maximum value of an ROI
Parameters	See Table 2 for common parameters
Return	the maximum value of an ROI

3.13 ImgMinMax

```
template <class T> void ImgMinMax
(const T *pImg, int w, int h, const CRect<int> &rcROI,
 T *pMin, T *pMax);
```

Description	Get the minimum and maximum value of an ROI
Parameters	See Table 2 for common parameters
pMin	return the minimum value of an ROI
pMax	return the maximum value of an ROI

3.14 ImgMean

```
template <class T> T ImgMean
(const T *pImg, int w, int h, const CRect<int> &rcROI);
```

Description	Get the mean value of an ROI
Parameters	See Table 2 for common parameters
Return	the mean value of an ROI

3.15 ImgMedian

```
template <class T> T ImgMedian
(const T *pImg, int w, int h, const CRect<int> &rcROI);
```

Description	Get the median value of an ROI
Parameters	See Table 2 for common parameters
Return	the median value of an ROI

3.16 ImgVariance

```
template <class T> double ImgVariance
(const T *pImg, int w, int h, const CRect<int> &rcROI);
```

Description	Get the variance of an ROI
Parameters	See Table 2 for common parameters
Return	the variance of an ROI

3.17 ImgStdDev

```
template <class T> double ImgStdDev
(const T *pImg, int w, int h, const CRect<int> &rcROI);
```

Description	Get the standard deviation of an ROI
Parameters	See Table 2 for common parameters
Return	the standard deviation of an ROI

3.18 ImgStatistics

```
template <class T> void ImgStatistics
(const T *pImg, int w, int h, const CRect<int> &rcROI);
 T *pMin, T *pMax, T *pMean, double *pdStdDev);
```

Description	Get statistics of an ROI
Parameters	See Table 2 for common parameters
pMin	return the minimum value of an ROI
pMax	return the maximum value of an ROI
pMean	return the mean value of an ROI
pdStdDev	return the standard deviation of an ROI

3.19 ImgFindMin

```
template <class T> T ImgFindMin
(const T *pImg, int w, int h, const CRect<int> &rcROI);
 CPt2D<int> *ppt);
```

Description	Find the pixel of min value in an ROI If the ROI has multiple minima (pixels of equal value), the coordinate of the first one is returned. The scan is from left-to-right and top-to-bottom.
Parameters	See Table 2 for common parameters
pMin	coordinate of the pixel of min value

3.20 ImgFindMax

```
template <class T> T ImgFindMax
(const T *pImg, int w, int h, const CRect<int> &rcROI);
 CPt2D<int> *ppt);
```

Description	Find the pixel of max value in an ROI If the ROI has multiple maxima (pixels of equal value), the coordinate of the first one is returned. The scan is from left-to-right and top-to-bottom.
Parameters	See Table 2 for common parameters
pMax	coordinate of the pixel of max value

3.21 ImgFltMean

```
template <class T> void ImgFltMean
(const T *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T *pDst, int w2, int h2, const CRect<int> &rcROI2,
 int nKerW, int nKerH);
```

Description	Mean filter, kernel certer equal to kernel mean Half-filter-length data at the ROI boundary are filtered with reduced sized kernel.
Parameters	See Table 2 for common parameters
nKerW	filter kernel width
nKerH	filter kernel height
Example	nKerW*nKerH = 3*3 or 7*5

```
template <class T1, class T2, class T3> void ImgFltMean
(const T1 *pImg, int w1, int h1, const CRect<int> &rcROI1,
 const T2 *pWei, int w2, int h2, const CRect<int> &rcROI2,
       T3 *pDst, int w3, int h3, const CRect<int> &rcROI3,
 int nKerW, int nKerH);
```

Description	Weighted mean filter
Parameters	See Table 2 for common parameters
pWei	weighting. (rcROI1, rcROI2 and rcROI3 must be the same size)

3.22 ImgFltMedian

```
template <class T> void ImgFltMedian
(const T *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T *pDst, int w2, int h2, const CRect<int> &rcROI2,
 int nKerW, int nKerH);
```

Description	Median filter, kernel certer equal to kernel median Half-filter-length data at the ROI boundary are filtered with reduced sized kernel.
Parameters	See Table 2 for common parameters
nKerW	filter kernel width
nKerH	filter kernel height
Example	nKerW*nKerH = 3*3 or 7*5

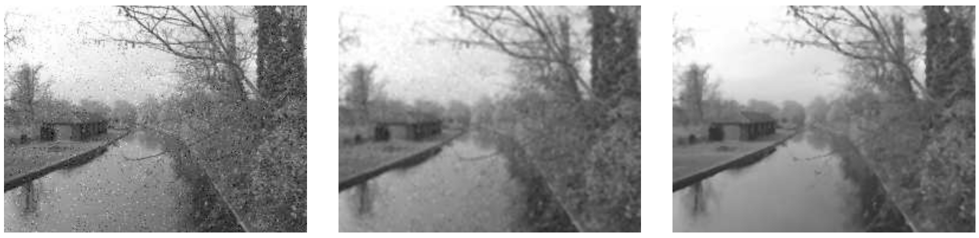

(a) (b) (c)

Figure 32: Examples of the ImgFltMean and ImgFltMedian functions. (a) Source image. (b) The image after filtered by ImgFltMean (3 by 3 kernel). (c) The image after filtered by ImgFltMedian (3 by 3 kernel).

3.23 ImgFltVariance

```
template <class T1, class T2> void ImgFltVariance
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2,
 int nKerW, int nKerH);
```

Description	Variance filter, kernel certer equal to kernel variance Half-filter-length data at the ROI boundary are filtered with reduced sized kernel.
Parameters	See Table 2 for common parameters
nKerW	filter kernel width
nKerH	filter kernel height
Example	nKerW*nKerH = 3*3 or 7*5

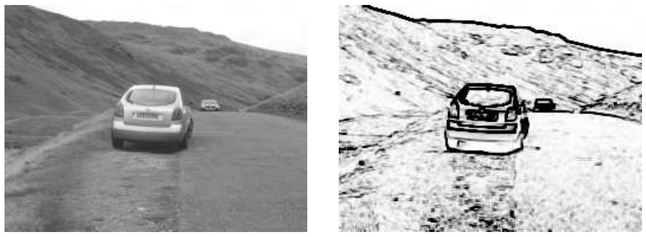

(a) (b)

Figure 33: An example of the ImgFltVariance function. (a) Source image. (b) The image after filtered by ImgFltVariance (3 by 3 kernel). Dark pixels indicate high variance; bright pixels indicate low variance.

3.24 ImgFltISEF

```
template <class T> void ImgFltISEF
(T *pImg, int w, int h, const CRect<int> &rcROI,
 double dStrength);
```

Description	Infinite symmetric exponential filter
Parameters	See Table 2 for common parameters
dStrength	ISEF filter strength [0, 1) (larger, smoother)

3.25 ImgLocalExtrema

```
template <class T1, class T2> int ImgLocalExtrema
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2,
 int nKerW, int nKerH, bool bMinima);
template <class T1, class T2> int ImgLocalExtrema
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
```

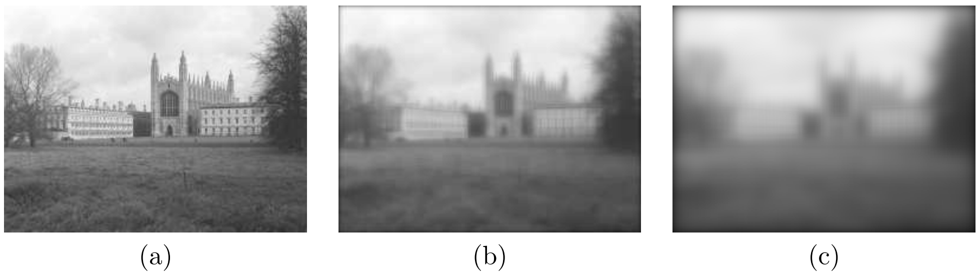


Figure 34: Examples of the ImgFltISEF function. (a) Source image. (b) The image after filtered by 0.5 filter strength. (c) The image after filtered by 0.8 filter strength.

```
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2,
 int nRadius, bool bMinima);
```

Description	Find local extrema Find local extrema (either minima or maxima, not both) in ROI1 and label such pixels in ROI2. Such a pixel is either the smallest or largest in a local region centered around it. The first version of the function takes a rectangular region; the second version takes a circular region. In pDst, extrema are labeled as 1; the rest are 0.
Parameters	See Table 2 for common parameters
nKerW	width of a rectangular region. An input even number will be forced to odd.
nKerH	height of a rectangular region. An input even number will be forced to odd.
nRadius	radius of a circular local region, which should be > 1.
bMinima	if true, find minima; otherwise, maxima.
Return	the number of found extrema

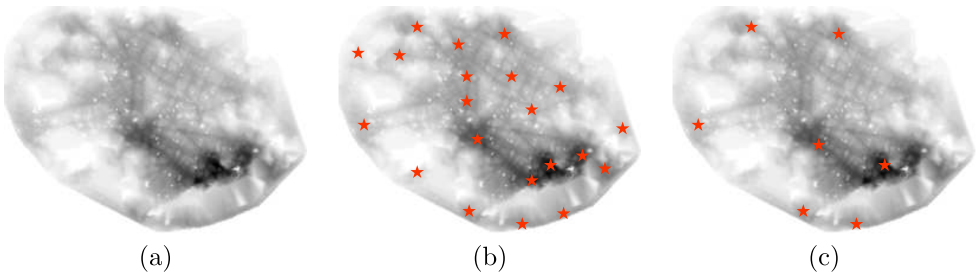


Figure 35: Examples of the ImgLocalExtrema function. (a) Source image. (b) and (c) are local minima of circular regions with radius 15 and 30 pixels respectively.

3.26 ImgConvX

```
template <class T1, class T2> void ImgConvX
(T1 *pImg, int w, int h, const CRect<int> &rcROI,
 const T2 *pMask, int nLen);
```

Description	1D convolution in x direction Convolution is applied on each row (x direction). Half-filter-length data at the ROI boundary are unchanged. The mask data array size must satisfy nLen+(nLen-1)/2 $\leq$ ROI.Width(); otherwise nothing is done.
Parameters	See Table 2 for common parameters
pMask	pointer to the convolution mask data array
nLen	mask array size

3.27 ImgConvY

```
template <class T1, class T2> void ImgConvY
(T1 *pImg, int w, int h, const CRect<int> &rcROI,
 const T2 *pMask, int nLen);
```

Description	1D convolution in y direction Convolution is applied on each column (y direction). Half-filter-length data at the ROI boundary are unchanged. The mask data array size must satisfy nLen+(nLen-1)/2 ≤ ROI.Height(); otherwise nothing is done.
Parameters	See Table 2 for common parameters
pMask	pointer to the convolution mask data array
nLen	mask array size

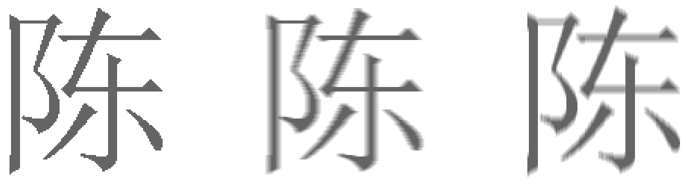

(a) (b) (c)

Figure 36: Examples of the ImgConvX and ImgConvY functions. (a) Source image. (b) The resultant image processed by ImgConvX. (c) The resultant image processed by ImgConvY. The convolution mask image is pMask[5] = {0.2, 0.2, 0.2, 0.2, 0.2}.

3.28 ImgConv

```
template <class T1, class T2> void ImgConv
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T1 *pDst, int w2, int h2, const CRect<int> &rcROI2,
 const T2 *pMask, int nKerW, int nKerH);
```

Description	2D convolution Half-filter-length data at the ROI boundary are unchanged.
Parameters	See Table 2 for common parameters
pMask	pointer to the convolution mask image
nKerW	mask image width
nKerH	mask image height

Example 1	Sobel convolution mask
	pMask[0] = -1, pMask[1] = 0, pMask[2] = 1
	pMask[3] = -2, pMask[4] = 0, pMask[5] = 2
	pMask[6] = -1, pMask[7] = 0, pMask[8] = 1

Example 2	Sharpening convolution mask
	pMask[0] = -0.2, pMask[1] = -0.5, pMask[2] = -0.2
	pMask[3] = -0.5, pMask[4] = +3.8, pMask[5] = -0.5
	pMask[6] = -0.2, pMask[1] = -0.5, pMask[2] = -0.2

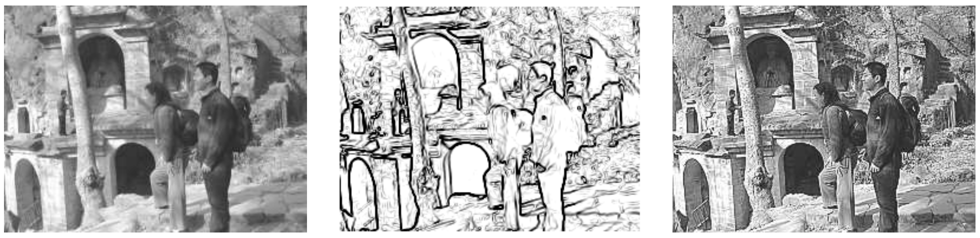

(a) (b) (c)

Figure 37: Examples of the ImgConv function. (a) Source image. (b) Image produced by Sobel convolution mask. Dark pixels indicate high values; bright pixels indicate low values. (c) Image produced by the sharpening convolution mask.

3.29 ImgResize

```
template <class T1, class T2> void ImgResize
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2,
 int nAlgo);
```

Description	Resize ROI1 of Src and copy to ROI2 of Dst If linear (nAlgo: 1) or spline (nAlgo: 2–4) is used, minification is always a downsampling process based on averaging. Magnification is done by the specified method. If nearest neighbour (nAlgo: 0) is used, both minification and magnification are based on nearest neighbour.
Parameters	See Table 2 for common parameters
nAlgo	interpolation algorithm 0: nearest neighbour interpolation, 1: linear interpolation, 2: cubic B-spline interpolation. 3: Catmull-Rom spline interpolation. 4: natural cubic spline interpolation.

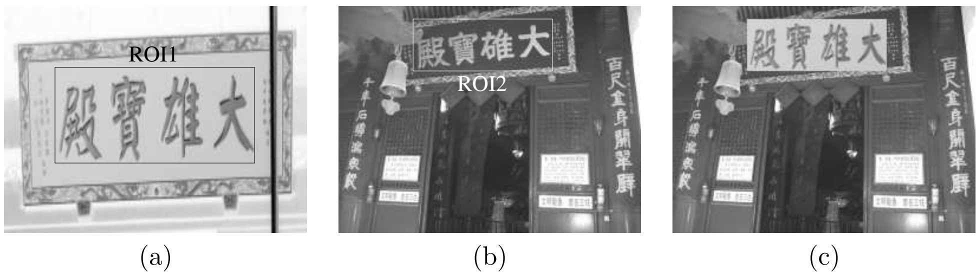


Figure 38: An example of the ImgResize function. (a) Source image. (b) Destination image. (c) Resized ROI1 is copied to ROI2.

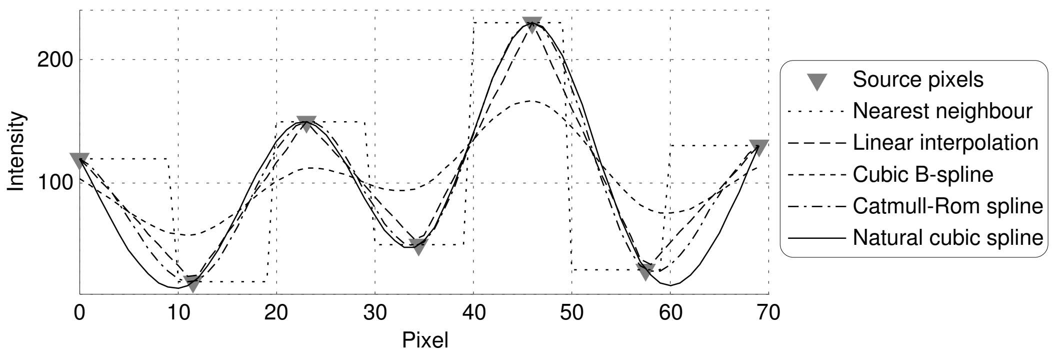


Figure 39: An example of different algorithms of the ImgResize function.

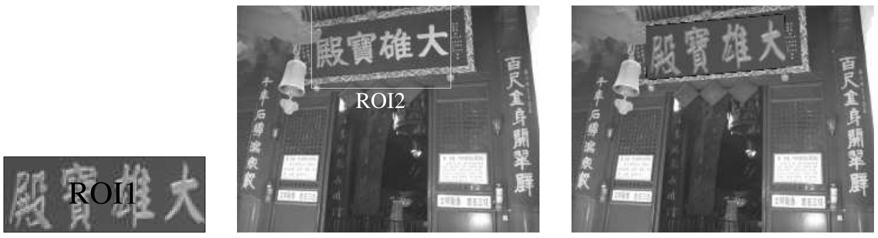
3.30 ImgRotate

```
template <class T1, class T2> void ImgRotate
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2,
 int nAngle, bool bLinear);
```

Description 1	Rotate ROI1 of Src and copy to ROI2 of Dst Rotation center is the center of ROI1. When copied to ROI2, the rotation center is matched with the center of ROI2.

continued on next page

2	If nAngle is not a multiple of 9000 (90 deg). The rotated points in ROI1 that do not fall in ROI2 are ignored. Those points in ROI2 that do not have counterpart in ROI1 are unchanged. Hence, there is no prerequisite on the size of ROI1 and ROI2.
3	If nAngle is a multiple of 9000 (90 deg), the size of ROI1 and ROI2 must be related accordingly. If the rotation angle is 0 or 180 deg, the size of ROI1 and ROI2 must be the same. If the rotation angle is 90 or 270 deg, the width and height of ROI1 and ROI2 must be exchanged.
Parameters	See Table 2 for common parameters
nAngle	anti-clock wise angle of rotation in 0.01 degree
bLinear	true for linear interpolation, false for nearest neighbour interpolation.



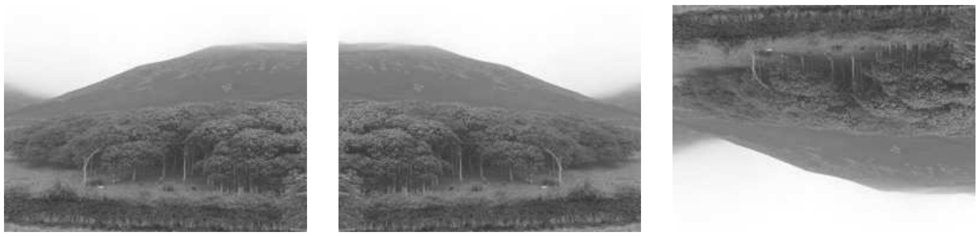
(a) (b) (c)

Figure 40: An example of the ImgRotate function. (a) Source image. (b) Destination image. (c) Rotated ROI1 is copied to ROI2.

3.31 ImgFlip

```
template <class T> void ImgFlip
(T *pImg, int w, int h, const CRect<int> &rcROI,
 bool bLeftRight);
```

Description	Flip image data in an ROI
Parameters	See Table 2 for common parameters
bLeftRight	true: left-right flip, false: top-bottom flip.



(a) (b) (c)

Figure 41: An example of the ImgFlip function. (a) Source image. (b) Image is flipped left to right. (c) Image is flipped top to bottom.

3.32 ImgFan

```
template <class T1, class T2> void ImgFan
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2,
 double dAngle, double dTopRadius, double dBottomRadius, bool bLinear);
```

Description	Create a fan-shaped image Empty region around the fan shape is unchanged.
Parameters	See Table 2 for common parameters
dAngle	angle in radius of the fan shape (0, 2π]
dTopRadius	radius of the top line in the source ROI
dBottomRadius	radius of the bottom line in the source ROI
bLinear	true for linear interpolation, false for nearest neighbour interpolation.

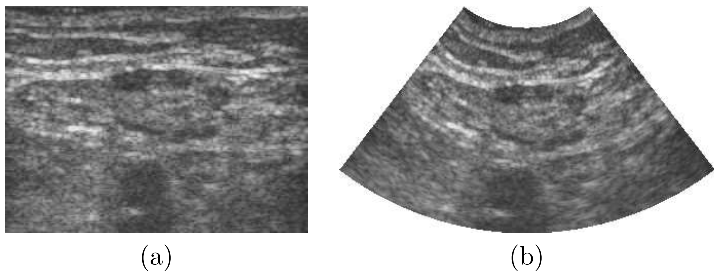


Figure 42: An example of the ImgFan function. (a) Source image. (b) Fan-shaped image.

3.33 ImgRadial

```
template <class T1, class T2> void ImgRadial
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2,
 const CPt2D<double> &ptCen, double dK);
```

Description	Create a radially distorted image ROI1 is radially distorted around ptCen based on equation $x_2 = ptCen.x + (1 + KR_1^2)(x_1 - ptCen.x)$ $y_2 = ptCen.y + (1 + KR_1^2)(y_1 - ptCen.y)$, where (x_1, y_1) and (x_2, y_2) is a pixel in ROI1 and ROI2 respectively, $R_1^2 = (x_1 - ptCen.x)^2 + (y_1 - ptCen.y)^2$. The distorted data are copied to ROI2. Pixels in ROI2 that do not have a counterpart in ROI1 are set to 0. Linear interpolation is used.
Parameters	See Table 2 for common parameters
ptCen	center of radial distortion in CRect(0,0,w1,h1) coordinate
dK	radial distortion coefficient

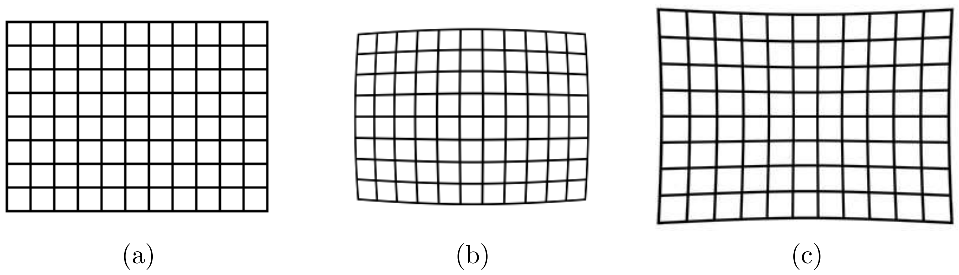
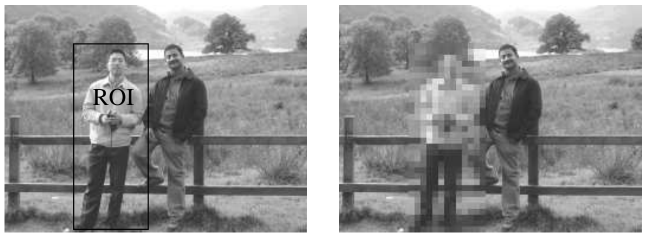


Figure 43: An example of the ImgRadial function. (a) Source image. (b) Barrel distortion. (c) Pincushion distortion.

3.34 ImgMosaic

```
template <class T> void ImgMosaic
(T *pImg, int w, int h, const CRect<int> &rcROI,
 int nKerW, int nKerH);
```

Description	Create mosaic effect
Parameters	See Table 2 for common parameters
nKerW	mosaic kernel width
nKerH	mosaic kernel height



(a) (b)

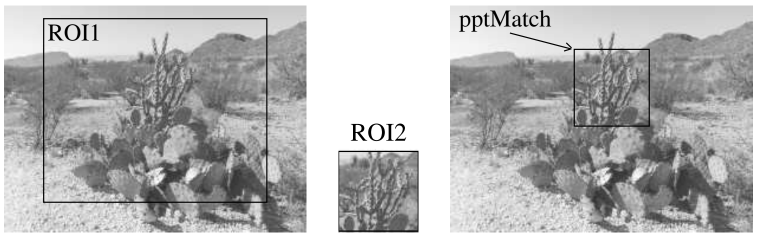
Figure 44: An example of the ImgMosaic function. (a) Source image. (b) The image data in the ROI are mosaicked.

3.35 ImgMatch

```
template <class T1, class T2> void ImgMatch
(const T1 *pMother, int w1, int h1, const CRect<int> &rcROI1,
 const T1 *pChild,  int w2, int h2, const CRect<int> &rcROI2,
 int nDSX, int nDSY, CPt2D<int> *pptMatch, T2 *pCorr);
```

Description	Search for ROI2 of pChild in ROI1 of pMother
	ROI1.Width() must be >= ROI2.Width()
	ROI1.Height() must be >= ROI2.Height()

1	Initially, ROI2 is matched to windows in ROI1 separated by nDSX and nDSY in the x and y directions. After the best-match window is found; nDSX and nDSY are halved.
2	Then ROI2 is matched to 9 windows, a 3 by 3 grid centered on the best-match window. The windows are separated by nDSX and nDSY. After a new best-match window is found, nDSX and nDSY are further halved.
3	Go back to 2 until both nDSX and nDSY equal to 1 pixel.
Parameters	See Table 2 for common parameters
nDSX	initial downsampling gap in x direction
nDSY	initial downsampling gap in y direction
	The downsampling gaps are used in shifting ROI1 across ROI2 and in accessing the pixels in ROI1 and ROI2,
pptMatch	the best-match point in ROI1 coordinate (left-top corner of the matched rect)
pCorr	normalized correlation coefficient, in [-1,1]. Its data type should be either float or double.



(a) (b)

Figure 45: An example of the ImgMatch function. (a) Mother image. (b) Child image. (c) Matched position of the child image in the mother image.

3.36 ImgMatchSubpixel

```
template <class T1, class T2> void ImgMatchSubpixel
(const T1 *pMother, int w1, int h1, const CRect<int> &rcROI1,
 const T1 *pChild,  int w2, int h2, const CRect<int> &rcROI2,
 int nLevel, CPt2D<float> *pptMatch, T2 *pCorr);
```

Description	Image matching at subpixel accuracy Subpixel search for ROI2 of pChild in ROI1 of pMother. The matching window is given by ROI2. ROI1 specifies the limit of the search region. Data matching is performed at several levels of increasing subpixel accruacy. At each level, 2 by 2 windows are searched. The best match is used as the center window of the next level search,
Parameters	See Table 2 for common parameters
nLevel	subpixel level (1: 0.5 pixel, 2: 0.25 pixel, etc)
pptMatch	On input, it represents the initial offset (eg. matching point at pixel accuracy). On output, it is the best match (left-top corner) of ROI2 in ROI1 at subpixel accuracy.
pCorr	normalized correlation coefficient, in [-1,1]. Its data type should be either float or double.

3.37 ImgMotion

```
template <class T1, class T2> bool ImgMotion
(const T1 *pImg1, int w1, int h1, const CRect<int> &rcROI1,
 const T1 *pImg2, int w2, int h2, const CRect<int> &rcROI2,
 int nCoarW, int nCoarH, int nCoarX, int nCoarY, int nCoarDSX, int nCoarDSY,
 int nFineW, int nFineH, int nFineX, int nFineY, int nFineDSX, int nFineDSY,
 int nWinX,  int nWinY, vector<CPt2D<int> > *pvptCenter,
 vector<CPt2D<int> > *pvptDisp, vector<T2> *pvCorr);
```

Description	Pixel flow or motion estimation Estimate the pixel motion from ROI2 in pImg2 to ROI1 in pImg1. ROI1 and ROI2 must be the same size.
1	There are two levels of search: a coarse search based on a few coarse windows and a quality-guided fine search on all fine windows. Results of the coarse search are used as the initial guess for the fine search.
2	A large coarse search window and search distance may be used to obtain robust initial estimates, and a small fine search window and search distance to obtain a high resolution. The downsampling gap at each level is set independently. See ImgMatch(...) for details of the downsampling gap.
3	Results are stored in pvptCenter, pvptDisp, pvCorr, which are arrays of the same size: nWinX * nWinY.
Parameters	See Table 2 for common parameters
nCoarW	coarse search window width
nCoarH	coarse search window height
nCoarX	coarse search distance in x direction
nCoarY	coarse search distance in y direction
nCoarDSX	coarse search downsampling gap in x direction
nCoarDSY	coarse search downsampling gap in y direction
nFineW	fine search window width
nFineH	fine search window height
nFineX	fine search distance in x direction
nFineY	fine search distance in y direction
nFineDSX	fine search downsampling gap in x direction
nFineDSY	fine search downsampling gap in y direction
nWinX	number of fine search windows in x direction

continued on next page

nWinY	number of fine search windows in y direction
pvptCenter	center of fine search windows in pImg2
pvptDisp	displacement of fine windows from pImg2 to pImg1 This includes the global shift from ROI2 to ROI1.
pvCorr	correlation coefficient of each fine window. Its data type should be either float or double.
Return	true: succeeded; false: failed.

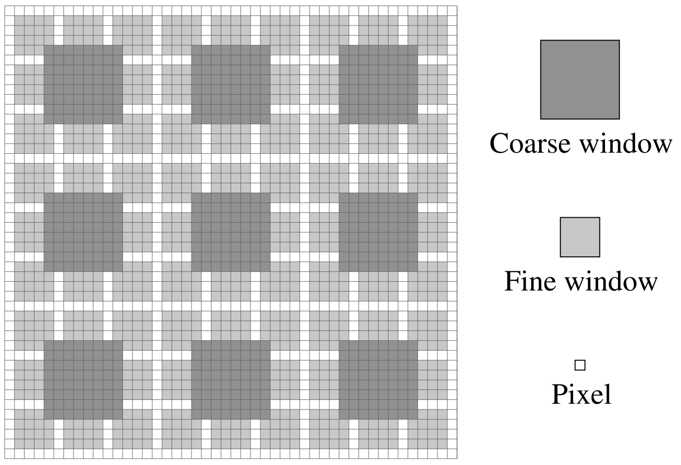


Figure 46: Illustration of the dual level search of the ImgMotion function.

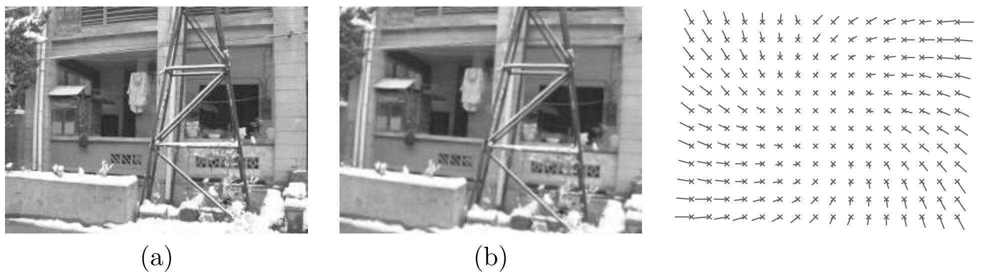


Figure 47: An example of the ImgMotion function. (a) Image 1. (b) Image 2. (c) Local motion vectors from image 1 to image 2.

3.38 ImgDIC

```
template <class T> bool ImgDIC
(const T *pImg1, int w1, int h1, const CRect<int> &rcROI1,
 const T *pImg2, int w2, int h2, const CRect<int> &rcROI2,
 const CRect<int> &rcAcc1, int nWinSize, int nWinX, int nWinY,
 vector<CPt2D<int> > *pvptCenter, vector<float> *pvDx,
 vector<float> *pvDy, vector<float> *pvCorr);
```

Description	Digital image correlation Calculate the subpixel displacement from ROI2 to ROI1. It applies ImgMotion(..) to obtain integer pixel displacement and then ImgMatchSubpixel(..) to obtain subpixel accuracy. Search parameters are estimated by the ROI size.
Parameters	See Table 2 for common parameters
rcAcc1	accessible region of pImg1, which should contain rcROI1.
nWinSize	window size for subpixel displacement search
nWinX	number of fine search windows in x direction

continued on next page

nWinY	number of fine search windows in y direction
pvptCenter	center of fine search windows in pImg2
pvDx	displacement in x direction
pvDy	displacement in y direction
pvCorr	correlation coefficient of each window
Return	true: succeeded; false: failed.

3.39 ImgDICStrain

```
template <class T> bool ImgDICStrain
(const T *pImg1, const T *pImg2, const T *pMask, int w, int h,
 int nWinSize, int nWinGap, int nFilter,
 vector<float> pvRes[5], float pfMin[5], float pfMax[5]);
```

Description Parameters	Calculate strain based on DIC See Table 2 for common parameters
pMask	mask image. Pixels > 0 indicate regions for calculation.
nWinSize	window size for subpixel displacement search
nWinGap	pixel gap between search windows
nFilter	number of filtering applied to strain fields
pvRes	store calculated displacement and strain fields. pvRes[0] and [1] are x and y displacement fields, [2], [3] and [4] are x, y normal and xy shear strain fields respectively. Their size is w/nWinGap by h/nWinGap on output.
pfMin	min displaying value for the corresponding pvRes
pfMax	max displaying value for the corresponding pvRes
Return	true: succeeded; false: failed.

3.40 ImgDICPtMapping

```
template <class T1, class T2> bool ImgDICPtMapping
(const T1 *pImg1, int w1, int h1, const CRect<int> &rcROI1,
 const T1 *pImg2, int w2, int h2, const CRect<int> &rcROI2,
 const CRect<int> &rcAcc1, int nWinSize, int nWinX, int nWinY,
 vector<CPt2D<T2> > *pvpt1, vector<CPt2D<T2> > *pvpt2,
 vector<float> *pvCorr = 0);
```

Description	Calculate point mapping based on DIC This function maps points in ROI2 of pImg2 to ROI1 of pImg1.
Parameters	See Table 2 for common parameters
rcAcc1	accessible region of pImg1, which should contain rcROI1.
nWinSize	window size for subpixel displacement search
nWinX	the number of fine search windows in x direction
nWinY	the number of fine search windows in y direction
pvpt1	mapped points in pImg1. Its array size is nWinX*nWinY.
pvpt2	mapped points in pImg2. Its array size is nWinX*nWinY.
pvCorr	correlation coefficient of each window
Return	true: succeeded; false: failed.

3.41 ImgHistogram

```
template <class T> void ImgHistogram
(const T *pImg, int w, int h, const CRect<int> &rcROI,
 T tMin, T tMax, int *pnHisto, int nLevel);
```

Description	Get histogram of the image data in an ROI
Parameters	See Table 2 for common parameters
tMin	minimum bound. Pixles < tMin contribute to pnHisto[0].
tMax	maximum bound. Pixels > tMax contribute to pnHisto[nLevel-1].
pnHisto	store histogram in nLevel entries. Each entry records the number of pixels falling into a specific range. Its size must be nLevel.
nLevel	level of histogram. The larger the nLevel, the finer is the resolution of the histogram.

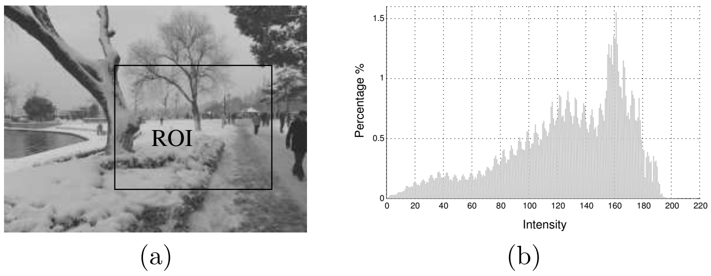


Figure 48: An example of the ImgHistogram function. (a) Source image. (b) Histogram of the ROI.

3.42 ImgHistoEqu

```
template <class T> void ImgHistoEqu
(T *pImg, int w, int h, const CRect<int> &rcROI,
 T tMin, T tMax, int nLevel);
```

Description	Equalize histogram of the image data in an ROI
Parameters	See Table 2 for common parameters
tMin	minimum bound. Pixles < tMin contribute to pnHisto[0].
tMax	maximum bound. Pixels > tMax contribute to pnHisto[nLevel-1].
nLevel	level of histogram. The larger the nLevel, the finer is the resolution of the histogram equalization.

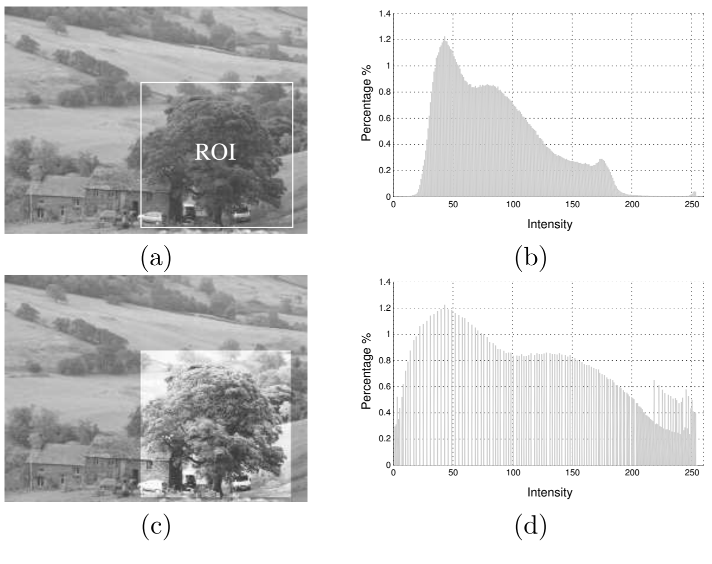


Figure 49: An example of the ImgHistoEqu function. (a) Source image ROI. (b) Histogram of the source image ROI. (c) Histogram-equalized ROI. (d) Histogram of the equalized ROI.

3.43 ImgClampExtreme

```
template <class T> void ImgClampExtreme
(T *pImg, int w, int h, const CRect<int> &rcROI,
 double dPerMin, double dPerMax, T *pMin = 0, T *pMax = 0);
```

Description	Clamp extreme values in an ROI The biggest value of the bottom "perMin" percent data is set as the minimum bound. The smallest value of the top "perMax" percent is set as the maximum bound. Image data that fall below the minimum bound or above the maximum are clamped to the bounds, respectively.
Parameters	See Table 2 for common parameters
dPerMin	percentage of minimum to be clamped. Should be in [0, 1].
dPerMax	percentage of maximum to be clamped. Should be in [0, 1].
pMin	the minimum value of the ROI after processing (input 0 to ignore)
pMax	the maximum value of the ROI after processing (input 0 to ignore)

3.44 ImgThre_TwoPeak

```
template <class T> T ImgThre_TwoPeak
(const T *pImg, int w, int h, const CRect<int> &rcROI);
```

Description	Get threshold of an ROI based on the two peaks method First, histogram of the ROI is computed. Then two peaks are located. The threshold is the mid value of the two peak values. Locate the max peak is straightforward. The 2nd peak is the max multiplication of the histogram values by the square of the distance from the first peak. This helps not to choose a pseudo-peak near the max (1st) peak.
Parameters	See Table 2 for common parameters
Return	threshold of an ROI

3.45 ImgThre_IterSel

```
template <class T> T ImgThre_IterSel
(const T *pImg, int w, int h, const CRect<int> &rcROI);
```

Description	Get threshold of an ROI based on the iterative selection method Take an initial guess of the threshold. Compute mean of all pixels below the threshold and the mean of all pixels above the threshold. The new threshold is (Tb+Ta)/2. Continue this process until there is no change in the threshold value.
Parameters	See Table 2 for common parameters
Return	threshold of an ROI

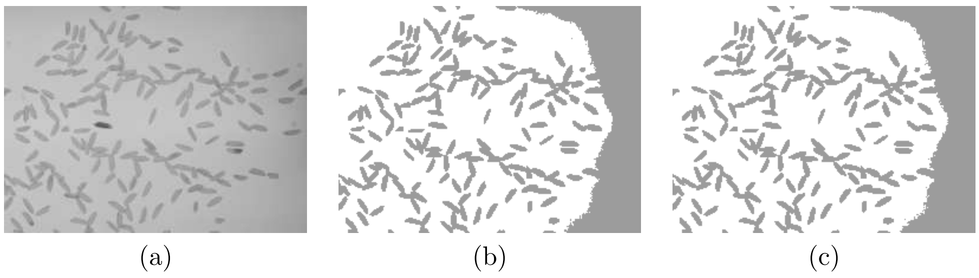


Figure 50: Examples of the ImgThre_TwoPeak and ImgThre_IterSel functions. (a) Source image. (b) Threshold obtained by the two-peak method. (c) Threshold obtained by the iterative selection method.

3.46 ImgDilation

```
template <class T> void ImgDilation
(const T *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T *pDst, int w2, int h2, const CRect<int> &rcROI2,
 const CPt2D<int> *ppt, int nCount);
```

Description	Morphology dilation Dilation is applied to intensity data rather than binary data. The input relative position array is used as structuring elements. A destination pixel is set to the max value (in ImgDilation) or min value (in ImgErosion) of the source pixels located by the relative positions. If a destnation pixel's all relative positions are outside the source region, it is set to the source region's global min (in ImgDilation) or max (in ImgErosion) value.
Parameters	See Table 2 for common parameters
ppt	relative position array (size must be nCount)
nCount	size of ppt
Example	Dilate in 135 deg: ppt[0]=(-1,-1); [1]=(1,1);

3.47 ImgErosion

```
template <class T> void ImgErosion
(const T *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T *pDst, int w2, int h2, const CRect<int> &rcROI2,
 const CPt2D<int> *ppt, int nCount);
```

Description	Morphology erosion Erosion is applied to intensity data rather than binary data. The input relative position array is used as structuring elements. A destination pixel is set to the max value (in ImgDilation) or min value (in ImgErosion) of the source pixels located by the relative positions. If a destnation pixel's all relative positions are outside the source region, it is set to the source region's global min (in ImgDilation) or max (in ImgErosion) value.
Parameters	See Table 2 for common parameters
ppt	relative position array (size must be nCount)
nCount	size of ppt
Example	Dilate in 135 deg: ppt[0]=(-1,-1); [1]=(1,1);

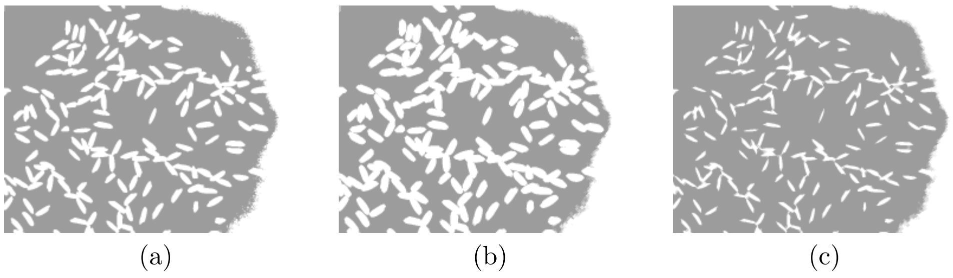


Figure 51: Examples of the ImgDilation and ImgErosion functions. (a) Source image. (b) Image dilation. (c) Image erosion.

3.48 ImgProjX

```
template <class T> void ImgProjX
(const T *pImg, int w, int h, const CRect<int> &rcROI,
       T *pProj);
```

Description	Get x direction projection of an ROI
Parameters	See Table 2 for common parameters
pProj	average of values along x direction. Its size must be >= ROI.Height(). Its index increases with y.

Example	`3,  5, 234, 90 | pvDst: 336/4`
	`7, 23,  34, 29 |        93/4`
	`12, 21,   9, 32 |        74/4`

3.49 ImgProjY

```
template <class T> void ImgProjY
(const T *pImg, int w, int h, const CRect<int> &rcROI,
      T *pProj);
```

Description	Get y direction projection of an ROI
Parameters	See Table 2 for common parameters
pProj	average of values along y direction. Its size must be >= ROI.Width(). Its index increases with x.

Example	`3, 5, 234, 90`
	`7, 23, 34, 29`
	`12, 21, 9, 32`
	`------------------------`
	`pvDst: 22/3, 49/3, 277/3, 151/3`

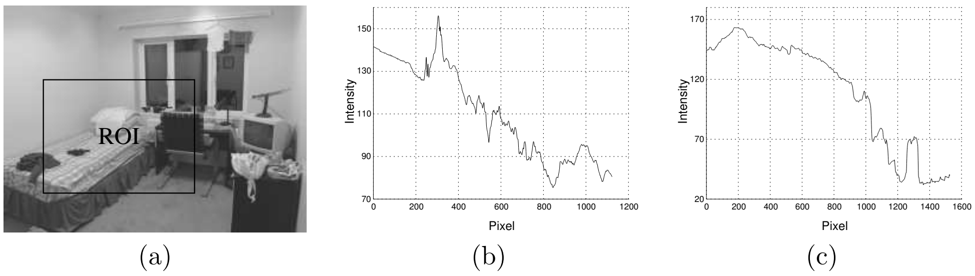


Figure 52: Examples of the ImgProjX and ImgProjY functions. (a) Source image. (b) Intensity projection of the ROI in the x direction. (c) Intensity projection of the ROI in the y direction.

3.50 ImgProjAny

```
template <class T> void ImgProjAny
(const T *pImg, int w, int h, const CRect<int> &rcROI,
 const CLine2D<int> &lnStart, const CLine2D<int> &lnEnd,
 T *pProj, int nLine);
```

Description	Get the projection of an arbitrary-shaped region The arbitrary-shaped region is specified by 2 lines. The projection sweeps from lnStart to lnEnd with "nLine" projection lines. If a line is outside the ROI, its projection value will be set to 0.
Parameters	See Table 2 for common parameters
lnStart	the start projection line
lnEnd	the end projection line (lnStart and lnEnd may intersect)
pProj	store average value of each projection line. Its size must be nLine.
nLine	number of projection lines

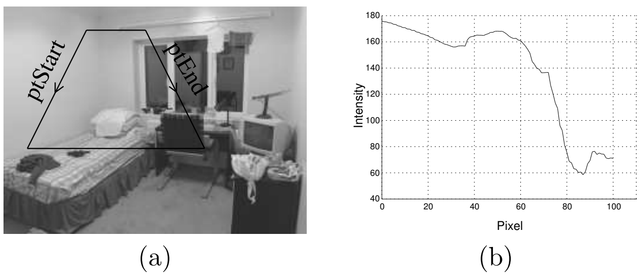


Figure 53: An example of the ImgProjAny function. (a) Source image with start and end projection lines. (b) Intensity projection of the ROI.

3.51 ImgRadon

```
template <class T1, class T2> void ImgRadon
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2,
 double dStartAngle, double dStepInc);
```

Description	Radon transform Radon transform is the integral of an image over straight lines. Parallel geometry is assumed. The projection width is ROI2.Width() and the number of angles is ROI2.Height(). The projection at each angle is stored in a line in ROI2.
Parameters	See Table 2 for common parameters
dStartAngle	the start angle of projection
dStepInc	the angular increase at each step

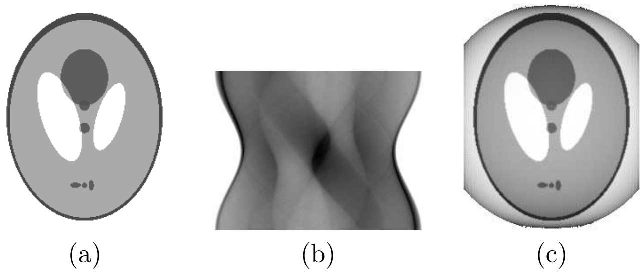


Figure 54: An example of the ImgRadon function. (a) Source image. (b) Radon transformed image. (c) Inverse Radon trasformed image by filtered backprojection.

3.52 ImgRadonVH

```
template <class T1, class T2, class Pred> void ImgRadonVH
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2,
 Pred pred, double dStartAngle, double dStepInc, T2 tObj);
```

Description	Radon transform for visual hull This function mimics non-penetrating light projection on objects. It is different from X-ray penetration of objects, modeled in ImgRadon(..). Parallel geometry is assumed. The projection width is ROI2.Width() and the number of angles is ROI2.Height(). The projection at each angle is stored in a line in ROI2.
Parameters	See Table 2 for common parameters
pDst	Pixels == 0 are definitely background, == tObj are definitely objects, in between are edge points.
dStartAngle	the start angle of projection
dStepInc	the angular increase at each step
tObj	value in pDst to indicate object pixels

3.53 ImgBackprojVH

```
template <class T1, class T2, class Pred> void ImgBackprojVH
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2,
 Pred pred, double *pdAngle, T2 tObj);
```

Description	Backprojection for visual hull Backprojection for visual hull (VH) is an erosive process in contrast to the accumulative process in the filtered backprojection for computed tomography (CT). At the start, all pixels in the backprojected field are set as objects. A background projection datum sets the pixels in the backprojected line to be background while an object datum does not change the backprojected field. Each line of ROI1 is the CP Radon transform projection at an angle (given by pdAngle). The size of pdAngle should be the same as ROI1.Height().
Parameters	See Table 2 for common parameters
pDst	Pixels == 0 are definitely background, == tObj are definitely objects, in between are edge points.
pdAngle	angle of each projection line
tObj	value in pDst to indicate object pixels

3.54 ImgIntegral

```
template <class T1, class T2> void ImgIntegral
(const T1 *pSrc,  int w1, int h1, const CRect<int> &rcROI1,
       T2 *pSum,  int w2, int h2, const CRect<int> &rcROI2);
   template <class T1, class T2> void ImgIntegral
(const T1 *pSrc,  int w1, int h1, const CRect<int> &rcROI1,
       T2 *pSum,  int w2, int h2, const CRect<int> &rcROI2,
       T2 *pSum2, int w3, int h3, const CRect<int> &rcROI3);
```

Description	Calculate two integral images The first version calculates the sum and the second version calculates both the sum and the sum of square. Each pixel in pSum (pSum2) is the sum (sum of square) of all pSrc data in the left-top region with respect to and including that pixel.
Parameters	See Table 2 for common parameters

3.55 ImgLabeling

```
template <class T1, class T2, class Pred> int ImgLabeling
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2,
 Pred pred, int nConnectivity, int nMinArea = 0, int nMaxArea = 0,
 vector<CRect<int> > *pvRect = 0, vector<int> *pvArea = 0);
```

Description	Connectivity labeling algorithm
Parameters	See Table 2 for common parameters
nConnectivity	Euclidean connectivity of an object. 0: 4-neighbour labeling algorithm, 1: 8-neighbour labeling algorithm, >1: pixels within this distance are one object.
nMinArea	Object fewer than nMinArea pixels are discarded.
nMaxArea	Object larger than nMaxArea pixels are discarded.
pvRect	bounding rectangle of each object (input 0 to ignore) It is with respect to ROI2. To obtain a bounding rect with respect to ROI1, each rect should be offset r1.LeftTop()-r2.LeftTop().
pvArea	number of points of each object (input 0 to ignore)
Return	the number of objects found
Example	The 1st object can be found by checking if(pDst[y*w+x]==1). The 1st object's bounding rect is (*pvRect)[0]. The 3rd object's number of points is (*pvArea)[2].

3.56 ImgFilling

```
template <class T, class Pred> void ImgFilling
(T *pImg, int w, int h, const CRect<int> &rcROI, T value,
```

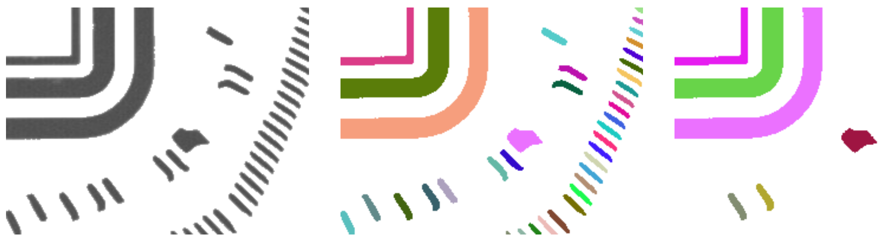
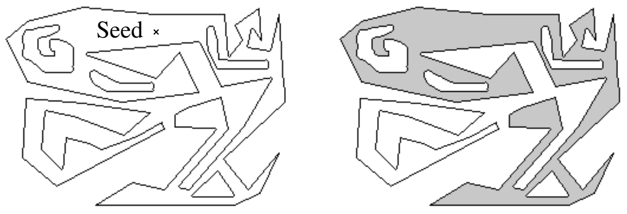

(a) (b) (c)

Figure 55: An example of the ImgLabeling function. (a) Source image. Dark pixels are to be labeled. (b) Labeled image. Each color represents an object and the background is white. (c) Labeled image with nMinArea = 150, Objects fewer than 150 pixels are discarded.

```
 Pred pred, CPt2D<int> ptSeed, int nConnectivity);
```

Description	Fill a region bounded by a predicate condition
Parameters	See Table 2 for common parameters
value	value to fill with
pred	if pred(pImg[y*w+x],value) is true, (x,y) is a boundary point. Only three predicates are valid: equal_to, greater_equal, less_equal. Other predicates, such as greater or less, may cause dead loop and should not be used.
ptSeed	initial seed point for filling
nConnectivity	Euclidean connectivity of a filling region. 0: 4-neighbour filling algorithm, 1: 8-neighbour filling algorithm, >1: pixels within this distance are filled.



(a) (b)

Figure 56: An example of the ImgFilling function. (a) Source image. The point is the seed of filling. (b) Filled image. Filling value: 200. pred (boundary condition): less_equal. Connectivity: 4-neighbour filling algorithm.

3.57 ImgThinning

```
template <class T, class Pred> void ImgThinning
(T *pImg, int w, int h, const CRect<int> &rcROI,
 Pred pred, T tBK);
```

Description	Thinning algorithm
Parameters	See Table 2 for common parameters
tBK	background value used to replace object pixels

3.58 ImgCentroid

```
template <class T, class Pred> bool ImgCentroid
(const T *pImg, int w, int h, const CRect<int> &rcROI,
 Pred pred, double *pdXCen, double *pdYCen);
```

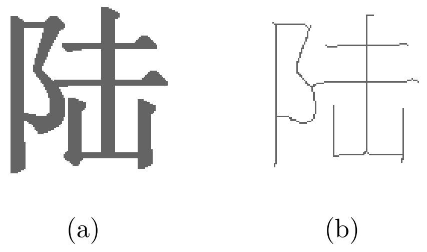


Figure 57: An example of the ImgThinning function. (a) Source image. (b) Thinned image.

Description	Get the centroid of an object
Parameters	See Table 2 for common parameters
pdXCen	x coordinate of the center, if found.
pdYCen	y coordinate of the center, if found.
Return	true: an object is found; otherwise, false.

3.59 ImgPerimeter

```
template <class T, class Pred> double ImgPerimeter
(const T *pImg, int w, int h, const CRect<int> &rcROI,
 Pred pred);
```

Description	Get the perimeter of an object
Parameters	See Table 2 for common parameters
Return	perimeter of an object if found; otherwise, 0.

3.60 ImgBoundaryOrdered

```
template <class T, class Pred> int ImgBoundaryOrdered
(const T *pImg, int w, int h, const CRect<int> &rcROI,
 Pred pred, vector<CPt2D<int> > *pvpt);
```

Description	Get clockwise ordered boundary points, based on 8-neighbour connectivity. Holes are not detected as boundary.
Parameters	See Table 2 for common parameters
pvpt	output the boundary points
Return	the number of boundary points

3.61 ImgBoundaryUnordered

```
template <class T, class Pred> int ImgBoundaryUnordered
(const T *pImg, int w, int h, const CRect<int> &rcROI,
 Pred pred, vector<CPt2D<int> > *pvpt);
```

Description	Get unordered boundary points by checking 4-way neighbours. Holes are detected as boundary.
Parameters	See Table 2 for common parameters
pvpt	output the boundary points
Return	the number of boundary points

3.62 ImgOuterCorners

```
template <class T, class Pred> int ImgOuterCorners
(const T *pImg, int w, int h, const CRect<int> &rcROI,
 Pred pred, vector<CPt2D<int> > *pvpt);
```

Description	Get the outer corner points of an object Outer corner points are used to calculate the convex hull.
Parameters	See Table 2 for common parameters
pvpt	stores the outer corner points, in clockwise order.
Return	the number of outer corner points

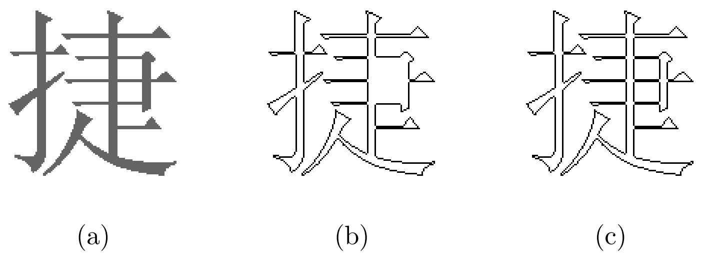


Figure 58: Examples of the ImgBoundaryOrdered and ImgBoundaryUnordered functions. (a) Source image. (b) Ordered boundary. Holes are ignored. (c) Unordered boundary. Holes are detected.

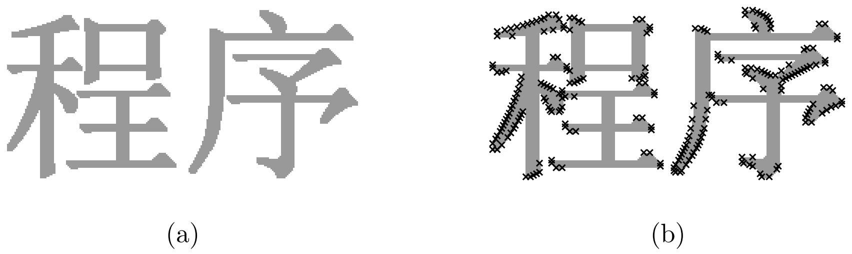


Figure 59: An example of the ImgOuterCorners function. (a) Source image. Dark pixels are considered as objects for detecting corner points. (b) Corner points of each object. Note that inner corners of a hole are not treated as corner points.

3.63 ImgConvexHull

```
template <class T, class Pred> int ImgConvexHull
(const T *pImg, int w, int h, const CRect<int> &rcROI,
 Pred pred, vector<CPt2D<int> > *pvCH);
```

Description	Get the convex hull points in clockwise order
Parameters	See Table 2 for common parameters
pvCH	output convex hull points (in clockwise order)
Return	the number of convex hull points

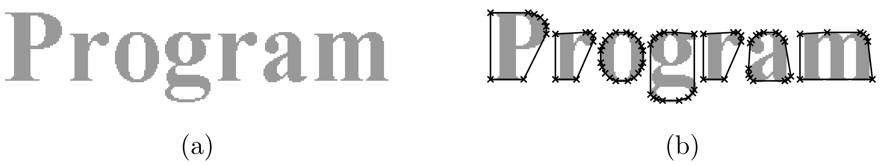


Figure 60: An example of the ImgConvexHull function. (a) Source image. Dark pixels are considered as objects for detecting convex hulls. (b) Convex hulls of each object.

3.64 ImgBoundingRect

```
template <class T, class Pred> CRect<int> ImgBoundingRect
(const T *pImg, int w, int h, const CRect<int> &rcROI, Pred pred);
```

Description	Get the bounding rectangle of an object
Parameters	See Table 2 for common parameters
Return	the bounding rectangle. The coordinate of all object points are within [rc.left, rc.right) and [rc.top, rc.bottom). If no object pixel exists, return an empty CRect.

3.65 ImgBoundingBox

```
template <class T, class Pred> bool ImgBoundingBox
(const T *pImg, int w, int h, const CRect<int> &rcROI, Pred pred,
 double dAngle, CPt2D<double> *pptCenter, CSize2D<double> *pSize);
```

Description	Get the fixed-angle bounding box of an object
Parameters	See Table 2 for common parameters
dAngle	angle of the bounding box (radius, anticlockwise)
pptCenter	center of the box
pSize	size of the box
Return	true: succeeded; false: failed (no object points).

3.66 ImgMinBoundingBox

```
template <class T, class Pred> bool ImgMinBoundingBox
(const T *pImg, int w, int h, const CRect<int> &rcROI, Pred pred,
 CPt2D<double> *pptCenter, CSize2D<double> *pSize, double *pdAngle);
```

Description	Get the minimum bounding box of an object
Parameters	See Table 2 for common parameters
pptCenter	center of the box
pSize	size of the box
pdAngle	angle of the box (radius, anticlockwise)
Return	true: succeeded; false: failed (no object points).

3.67 ImgMinBoundingCircle

```
template <class T, class Pred> bool ImgMinBoundingCircle
(const T *pImg, int w, int h, const CRect<int> &rcROI,
 Pred pred, CPt2D<double> *pptCenter, double *pdRadius);
```

Description	Get the minimum bounding circle of an object
Parameters	See Table 2 for common parameters
pptCenter	center of the circle
pdRadius	radius of the circle
Return	true: succeeded; false: failed (no object points).

3.68 ImgMinBoundingPolygon

```
template <class T, class Pred> bool ImgMinBoundingPolygon
(const T *pImg, int w, int h, const CRect<int> &rcROI,
 Pred pred, vector<CPt2D<int> > *pvpt);
```

Description	Get the minimum bounding polygon of an object Such polygon differs from a complete list of boundary points in that it removes points on any straight line parallel to X or Y axis. Only end points of a line are kept as the polygon vertices.
Parameters	See Table 2 for common parameters
pvpt	stores the polygon vertices, in clockwise order.
Return	the number of polygon vertices

3.69 ImgDistTrans

```
template <class T, class Pred> void ImgDistTrans
(const T       *pSrc,  int w1, int h1, const CRect<int> &rcROI1,
 unsigned long *pnDst, int w2, int h2, const CRect<int> &rcROI2,
 Pred pred);
```

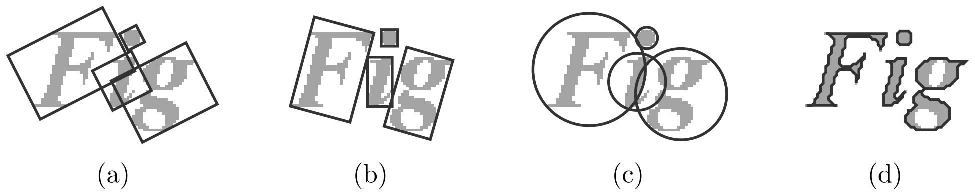


Figure 61: Examples of the ImgBoundingBox, ImgMinBoundingBox, ImgMinBoundingCircle, and ImgMinBoundingPolygon functions. (a) Bounding box at a fixed angle of 27 deg. (b) Minimum area bounding box. (c) Minimum area bounding circle. (d) Minimum area bounding polygon.

Description	Distance transform algorithm Object boundary pixels are set to 0. A nun-boundary pixel is set to the distance to its nearest object boundary pixel. A non-boundary object pixel gets a positive distance and a background pixel gets a negative distance. The generated distance value is 100 * the actual Euclidean distance. Reference "Euclidean Distance Mapping", P. Danielsson.
Parameters	See Table 2 for common parameters

3.70 ImgWatershed

```
template <class T1, class T2> int ImgWatershed
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2,
 int nDist, int nMinPts);
```

Description	Watershed algorithm for segmentation This function assumes the input pSrc a distance map, for example, obtained from ImgDistTrans(...). Negative pixels are background; otherwise, object. It outputs an object map similar to that produced by ImgLabeling(...). In the object map, previously connected regions are segmented.
1	Local maxima are located for each object.
2	The first-pass propagation estimates the number of points grown by each maxima within each object.
3	Maxima whose number of points are < nMinPts are discarded.
4	The second-pass propagation finishes the segmentation.
Parameters	See Table 2 for common parameters
nDist	the distance for merging local maxima. 0: 4-neighbour merging, 1: 8-neighbour merging, >1: local maxima within this distance are merged.
nMinPts	threshold of the number of points grown by a maximum.
	Both parameters are used to reduce over-segmentation.
Return	The number of segmented objects, if succeeded; otherwise -1.

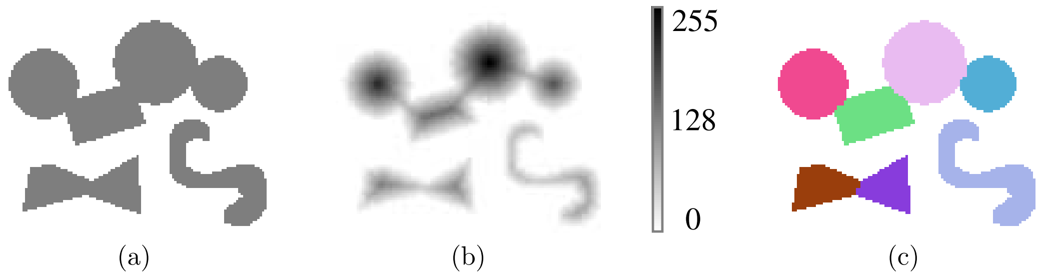


Figure 62: Examples of the ImgDistTrans and ImgWatershed functions. (a) Source image. (b) Distance transformed image. (c) Watershed segmented image.

3.71 ImgEdgeX

```
template <class T> int ImgEdgeX
(T *pImg, int w, int h, const CRect<int> &rcROI,
 T tThre, int nEdgeType, int nHFL, int nIndex,
 int nDSY, vector<CPt2D<int> > *pvptEdge);
```

Description	1D edge detection in x direction (description applicable to y direction) 1D edge detection based on the contrast of image data along x or y direction. The scan of edge point in x direction is left-to-right; the scan in y direction is top-to-bottom.
1	At each point, nHFL specifies how many pixels to check to the left and right (x direction) or to the top and bottom (y direction) of a point. The average intensity of the two sections are calculated. If their difference is >= tThre, the center point is considered as an edge point. nHFL-pixel boundary of the ROI is not scanned.
2	There are two types of edges: positive and negative. In the direction of scan (x direction: left-to-right, y direction: top-to-bottom), a positive edge indicates the data values are increasing; while a negative edge indicates decreasing.
3	One or multiple edge points at each line can be output to vptEdge. If nIndex is non-zero, one edge point is selected. For example, 1: the first edge point, 2: the second, etc; -1: the last, -2: the second last, etc. If nIndex is 0, all edge points are output.
4	If two edge points of the same type are found within 2*nHFL pixels, only the one with larger contrast is output because they are likely to be part of a steep edge.
Parameters	See Table 2 for common parameters
tThre	edge contrast threshold
nEdgeType	edge type (-1: negative, 1: positive, 0: both)
nHFL	half-filter-length at each side of a point
nIndex	index of the edge point to be output
nDSY	downsampling gap in y direction (eg. If 1, each line, or if 2, every other line is scanned.)
pvptEdge	detected edge points. Some line may not have any edge point.
Return	the number of edge points found

3.72 ImgEdgeY

```
template <class T> int ImgEdgeY
(T *pImg, int w, int h, const CRect<int> &rcROI,
 T tThre, int nEdgeType, int nHFL, int nIndex,
 int nDSX, vector<CPt2D<int> > *pvptEdge);
```

Description	1D edge detection in y direction (see ImgEdgeX for details)
Parameters	See Table 2 for common parameters
nDSX	downsampling gap in x direction (eg. If 1, each row, or if 2, every other row is scanned.)

3.73 ImgEdgeXSubpixel

```
template <class T> int ImgEdgeXSubpixel
(T *pImg, int w, int h, const CRect<int> &rcROI,
 T tThre, int nEdgeType, int nHFL, int nIndex,
 int nDSY, vector<CPt2D<double> > *pvptEdge);
```

Description	1D edge detection in x direction at subpixel accuracy All the parameters and the return value are the same as those of ImgEdgeX except for pvptEdge, whose x coord is at subpixel accuracy and y coord is still in integer pixel.

3.74 ImgEdgeYSubpixel

```
template <class T> int ImgEdgeYSubpixel
(T *pImg, int w, int h, const CRect<int> &rcROI,
 T tThre, int nEdgeType, int nHFL, int nIndex,
 int nDSX, vector<CPt2D<double> > *pvptEdge);
```

Description	1D edge detection in y direction at subpixel accuracy All the parameters and the return value are the same as those of ImgEdgeY except for pvptEdge, whose y coord is at subpixel accuracy and x coord is still in integer pixel.

3.75 ImgEdgeAny

```
template <class T> int ImgEdgeAny
(T *pImg, int w, int h, const CRect<int> &rcROI,
 const CLine2D<double> &lnGuide, T tThre, int nEdgeType, int nHFL,
 int nScanLen, int nNumScan, vector<CPt2D<double> > *pvptEdge,
 vector<CLine2D<double> > *pvlnScan = 0);
```

Description 1 2 3 4	Detect 1D edge based on a guiding line Multiple scan lines are generated perpendicular to the guiding line. Their direction is determined by lnGuide: lnGuide.Start() → lnGuide.End() → lnScan.Start() is anticlockwise and lnGuide.Start() → lnGuide.End() → lnScan.End() is clockwise. In the direction of lnScan, a positive edge indicates increasing data values; while a negative edge indicates decreasing data values. On each scan line, the maximum contrast (strongest) edge point is output, if its contrast is larger than tThre. Edge contrast is the difference of the average intensity of the nHFL pixels before and after an edge point.
Parameters	See Table 2 for common parameters
lnGuide	a guiding line for searching for edge points
tThre	edge contrast threshold
nEdgeType	edge type (-1: negative, 1: positive, 0: both)
nHFL	half-filter-length at each side of a point
nScanLen	scan length for an edge point
nNumScan	number of scan lines on the guiding line
pvptEdge	detected edge points at subpixel accuracy
pvlnScan	scan lines (input 0 to ignore)
Return	the number of edge points found

3.76 ImgEdgeOnLine

```
template <class T> int ImgEdgeOnLine
(T *pImg, int w, int h, const CRect<int> &rcROI,
 const CLine2D<int> &lnScan, T tThre, int nEdgeType, int nHFL,
 vector<CPt2D<int> > *pvptEdge);
```

Description	Detect 1D edge on a line Scan from lnScan.Start() to lnScan.End() for edge points. In the direction of lnScan, a positive edge indicates the data values are increasing; while a negative edge indicates decreasing. All edge points with contrast >= tThre are output to pvptEdge but if two edge points of the same type are found within 2*nHFL pixels, only the one with larger contrast is output because they are likely to be part of a steep edge. Edge contrast is the difference of the average intensity of the nHFL pixels before and after an edge point.

continued on next page

Parameters	See Table 2 for common parameters
lnScan	scan line for edge points
tThre	edge contrast threshold
nEdgeType	edge type (-1: negative, 1: positive, 0: both)
nHFL	half-filter-length at each side of a point
pvptEdge	detected edge points
Return	the number of edge points found

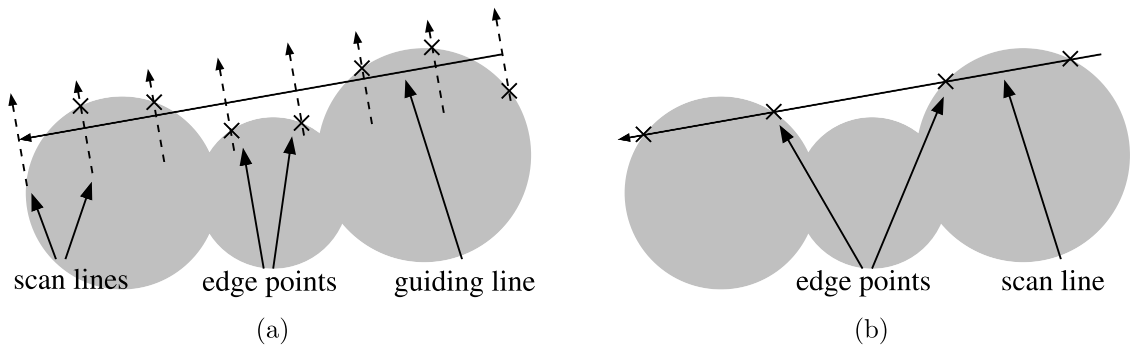


Figure 63: Examples of the ImgEdgeAny and ImgEdgeOnLine functions. (a) ImgEdgeAny and (b) ImgEdgeOnLine function.

3.77 ImgCannyEdge

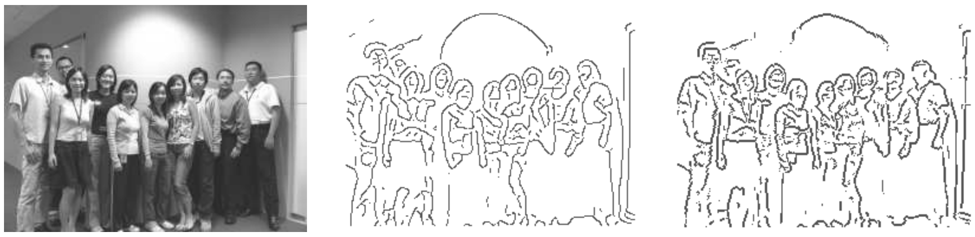
```
template <class T> void ImgCannyEdge
(T *pImg, int w, int h, const CRect<int> &rcROI,
 T lowThre, T highThre, int nFltLen);
```

Description	Canny edge detection algorithm The edge magnitude (gradient in the x and y directions) at each pixel is calculated and non-local-maximum pixels are set to 0. Then, a hysteresis thresholding procedure is applied to extract the edge pixels. Pixels $\geq$ lowThre and connected to any pixel $\geq$ highThre are thresholded as edge (set to 1). Those that are $<$ lowThre or $<$ highThre but not connected to any pixel $>$ highThre are set to 0.
Parameters	See Table 2 for common parameters
lowThre	low threshold for hysteresis thresholding
highThre	high threshold for hysteresis thresholding
nFltLen	Gaussian filter length ($\geq$3)
	If highThre $>$ lowThre, pImg stores edge (1) and non-edge (0). If highThre $\leq$ lowThre, pImg stores edge magnitude.

3.78 ImgShenCastanEdge

```
template <class T> void ImgShenCastanEdge
(T *pImg, int w, int h, const CRect<int> &rcROI,
 T lowThre, T highThre, double dStrength, int nWinSize);
```

Description	Shen and Castan edge detection algorithm
Parameters	See Table 2 for common parameters
lowThre	low threshold for hysteresis thresholding
highThre	high threshold for hysteresis thresholding
dStrength	ISEF filter strength [0, 0.9]
nWinSize	window size for calculating the edge contrast
	If highThre $>$ lowThre, pImg stores edge (1) and non-edge (0). If highThre $\leq$ lowThre, pImg stores edge magnitude.



(a) (b) (c)

Figure 64: Examples of the ImgCannyEdge and ImgShenCastanEdge functions. (a) Source image. (b) Canny edge. (c) ShenCastan edge.

3.79 ImgFitPlane

```
template <class T> bool ImgFitPlane
(const T *pImg, int w, int h, const CRect<int> &rcROI,
 double *pdKx, double *pdKy, double *pdZ0);
```

Description	Fit a plane to image data of an ROI Plane equation z(x,y) = Kx*x + Ky*y + Z0
Parameters	See Table 2 for common parameters
pdKx, ... pdZ0	parameters of the plane equation, if succeed; or undefined.
Return	true: succeeded; false: failed.

```
template <class T1, class T2> bool ImgFitPlane
(const T1 *pImg, int w1, int h1, const CRect<int> &rcROI1,
 const T2 *pWei, int w2, int h2, const CRect<int> &rcROI2,
 double *pdKx, double *pdKy, double *pdZ0);
```

Description	Fit a plane to weighted image data
Parameters	See Table 2 for common parameters
pWei	weighting. (rcROI1, rcROI2 must be the same size)

3.80 ImgFitPolynomialSurf

```
template <class T> bool ImgFitPolynomialSurf
(const T *pImg, int w, int h, const CRect<int> &rcROI,
 int N, vector<double> *pvCoe,
 vector<int> *pvPowOfX, vector<int> *pvPowOfY);
```

Description	Fit a polynomial surface to the image data in an ROI Polynomial surface equation: z(x,y) = $+C_0 + C_1x + C_3x^2 + C_6x^3 + ... + C_?x^N$ $+C_2y + C_4xy + C_7x^2y + ...$ $+C_5y^2 + C_8xy^2 + ...$ $+... + ... + ...$ $+C_{(N+1)(N+2)/2-1}y^N$
Parameters	See Table 2 for common parameters
N	the order of the surface equation (should be >= 0)
pvCoe	store coefficients, if succeed; or undefined.
pvPowOfX	store power of x of each term, if succeed; or undefined.
pvPowOfY	store power of y of each term, if succeed; or undefined.
	After processing, the size of pvCoe, pvPowOfX and pvPowOfY will be (N+1)(N+2)/2 and their elements are ordered according to the index of *C* in the equation.
Return	true: succeeded; false: failed.

```
template <class T1, class T2> bool ImgFitPolynomialSurf
(const T1 *pImg, int w1, int h1, const CRect<int> &rcROI1,
 const T2 *pWei, int w2, int h2, const CRect<int> &rcROI2,
 int N, vector<double> *pvCoe,
 vector<int> *pvPowOfX, vector<int> *pvPowOfY);
```

Description	Weight version of ImgFitPolynomialSurf
Parameters	See Table 2 for common parameters
pWei	weighting. (rcROI1, rcROI2 must be the same size)

3.81 ImgFitGauss

```
template <class T> bool ImgFitGauss
(const T *pImg, int w, int h, const CRect<int> &rcROI,
 T threshold, double pdCoe[5]);
```

Description	Fit a 2D Gaussian function to image data of an ROI Gaussian function: $z = Ae^{-(x-B)^2/(2C)-(y-D)^2/(2E)}$. Fitting is based on a linear least squares method: the z coord of all points is converted to ln(z), which requires that the image data > 0.
Parameters	See Table 2 for common parameters
threshold	image data < threshold are not used for fitting. The threshold must be > 0 because of logarithm.
pdCoe	parameters of the Gaussian function pdCoe[0], pdCoe[1], ... pdCoe[4] are A, B, ... E respectively.
Return	true: succeeded; false: failed.

3.82 ImgFitImg

```
template <class T> bool ImgFitImg
(const T1 *pImg1, int w1, int h1, const CRect<int> &rcROI1,
 const T2 *pImg2, int w2, int h2, const CRect<int> &rcROI2,
 int N, vector<double> *pvCoe);
```

Description	Fit a plane to image data of an ROI Let pImg1 and pImg2 be x and y respectively, the fitting equation is $y = C_0 + C_1x + C_2x^2 + ... + C_Nx^N$. The least squares method is applied to each pair of pixels in pImg1 and pImg2 to find the coefficients.
Parameters	See Table 2 for common parameters
N	the order of the polynomial equation (should be >= 0)
pvCoe	store coefficients, if succeed; or undefined.
Return	true: succeeded; false: failed.

3.83 ImgSurfNormal

```
template <class T1, class T2> bool ImgSurfNormal
(const T1 *pImg, int w, int h, const CRect<int> &rcROI,
 int nKerW, int nKerH, int nWinX, int nWinY,
 vector<T2> *pvNx, vector<T2> *pvNy);
```

Description	Get image surface normals Calculate surface normals of the image data. The x and y components are output by pvNx and pvNy. The z component is consistently 1 or -1, depending on which side of the surface is defined as the front.
Parameters	See Table 2 for common parameters
nKerW	kernel width for calculating the surface normal at each point
nKerH	kernel height for calculating the surface normal at each point
nWinX	number of surface normal in x direction. It should be > 1 and $\leq$ nRw.
nWinY	number of surface normal in y direction. It should be > 1 and $\leq$ nRh.
pvNx	output x component of the normal vectors (array size = nWinX*nWinY).
pvNy	output y component of the normal vectors (array size = nWinX*nWinY).
Return	true: succeeded; false: nWinX or nWinY not suitable.

3.84 ImgLineData

```
template <class T> int ImgLineData
(const T *pImg, int w, int h, const CRect<int> &rcROI,
 const CPt2D<int> &ptStart, const CPt2D<int> &ptEnd,
 vector<T> *pvDst, vector<CPt2D<int> > *pvpt = 0);
```

Description	Get image data on a line A point on the line is round off to integer sample accuracy.
Parameters	See Table 2 for common parameters
ptStart	start point of the line
ptEnd	end point of the line
pvDst	store image data on the line
pvpt	store points on the line (input 0 to ignore)
Return	the number of pixels on the line

3.85 ImgLineDataSubpixel

```
template <class T> int ImgLineDataSubpixel
(const T *pImg, int w, int h, const CRect<int> &rcROI,
 const CPt2D<double> &ptStart, const CPt2D<double> &ptEnd,
 vector<T> *pvDst, vector<CPt2D<double> > *pvpt = 0);
```

Description	Get subpixel image data on a line A subpixel version of ImgLineData(..).

3.86 ImgLineSum

```
template <class T> int ImgLineSum
(const T *pImg, int w, int h, const CRect<int> &rcROI,
 const CPt2D<int> &ptStart, const CPt2D<int> &ptEnd, double *pdSum);
```

Description	Get the sum of a line's data A point on the line is kept with subsample accuracy and obtains its datum by bilinear interpolation.
Parameters	See Table 2 for common parameters
ptStart	start point of the line
ptEnd	end point of the line
pdSum	sum of the line's data
Return	the number of pixels on the line

3.87 ImgContour

```
template <class T1, class T2> void ImgContour
(const T1 *pSrc, int w, int h, const CRect<int> &rcROI,
      T2 *pDst, int nContourLine);
```

Description	Get contour lines of the image data in an ROI
1	Compute each contour line's value.
2	For each pixel, check which contour line is the nearest.
3	Compare the pixel's 4-way neighbours' value. If the contour line's value is in between, then the pixel is a contour point.
Parameters	See Table 2 for common parameters
pDst	image of contour object. size must be w*h. value = i represent the ith contour. background is 0.
nContourLine	number of contour lines to extract

3.88 ImgMask

```
template <class T1, class T2, class Pred> void ImgMask
(const T1 *pMask, int w1, int h1, const CRect<int> &rcROI1,
      T2 *pImg,  int w2, int h2, const CRect<int> &rcROI2, T2 value);
```

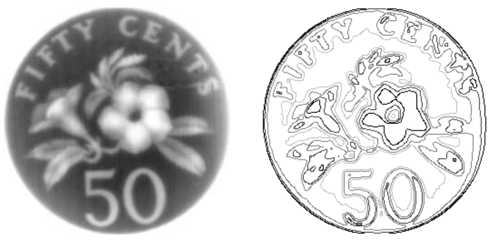

(a) (b)

Figure 65: An example of the ImgContour function. (a) Source image. (b) Contours of intensity. Each gray scale represents a level of intensity.

Description	Set image data based on a mask image
Parameters	See Table 2 for common parameters
pred	if pred(pMask[y*w+x],value) is true, pMask[y*w+x] is set to the mask value.
value	mask value

3.89 ImgMaskPolygon

```
template <class T1, class T2> void ImgMaskPolygon
(T1 *pImg, int w, int h, const CRect<int> &rcROI,
 const CPt2D<T2> *ppt, int nCount);
```

Description	Mask a polygonal region Create a mask image for a polygonal region by filling. This approach is faster than checking if each pixel is in or out of the polygon. After processing, pixels outside the polygon are set to 0; those inside are set to 1; those on the edges are set to 2.
Parameters	See Table 2 for common parameters
ppt	vertices of a polygon
nCount	size of ppt

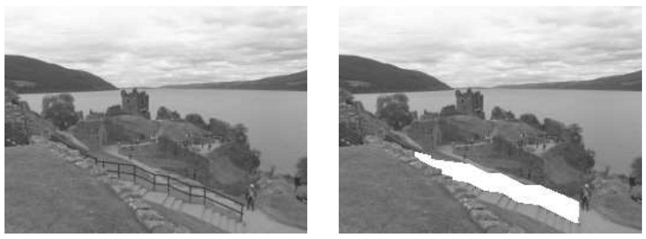

(a) (b)

Figure 66: An example of the ImgMaskPolygon function. (a) Source image. (b) A polygon region is masked with value 255.

3.90 ImgMaskRing

```
template <class T1, class T2> void ImgMaskRing
(T1 *pImg, int w, int h, const CRect<int> &rcROI,
 const CRing2D<T2> &ring);
```

Description	Mask a ring region see ImgMaskPolygon(..)
Parameters	See Table 2 for common parameters
ring	a ring region

3.91 ImgComplete

```
template <class T1, class T2> void ImgComplete
(const T1 *pMask, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pImg,  int w2, int h2, const CRect<int> &rcROI2,
 vector<int> *pvObjMap);
```

Description	Complete an image with regions of missing data Fill regions of missing data in pImg using the neighbouring data. The missing data are indicated by pixels <= 0 in pMask. The function checks a 3-by-3 neighbourhood of each missing data point and assigns the average of the neighbours to the point. A neighbour involved in averaging must be a real (non-missing) data point. An iterative process is applied; it stops when all missing data are filled.
Parameters	See Table 2 for common parameters
pMask	pixels <= 0 are missing data points. ROI1 is mapped to ROI2, both having the same size.
pvObjMap	object map, used internally. Its memory is allocated only if necessary.

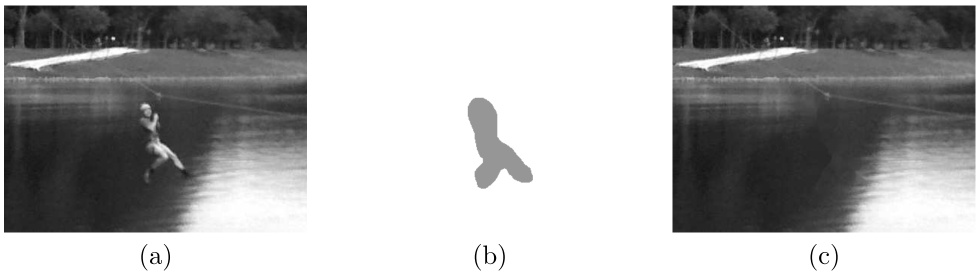


Figure 67: An example of the ImgComplete function. (a) Original image. (b) Mask image. (c) Resultant image.

3.92 ImgMandelbrot

```
template <class T> int ImgMandelbrot
(T *pImg, int w, int h, const CRect<int> &rcROI,
 double dX1, double dY1, double dX2, double dY2,
 int nMaxIteration);
```

Description	Create a Mandelbrot pattern
Parameters	See Table 2 for common parameters
pImg	output Mandelbrot image. Pixel value contains how many iterations used to decide that the point is in or out of the Mandelbrot set.
dX1	minimum bound of the Mandelbrot set in the x direction
dY1	minimum bound of the Mandelbrot set in the y direction
dX2	maximum bound of the Mandelbrot set in the x direction
dY2	maximum bound of the Mandelbrot set in the y direction
nMaxIteration	maximum iterations to be applied
Return	the min iterations used among all points to reach a decision that whether a point is in or out of the Mandelbrot set

3.93 ImgPhaseShifting

```
template <class T1, class T2> void ImgPhaseShifting
(const T1 *pSrc,   int w1, int h1, const CRect<int> &rcROI1,
       T2 *pPhase, int w2, int h2, const CRect<int> &rcROI2,
 const T2 *pPS, int nStep);
template <class T1, class T2> void ImgPhaseShifting
(const T1 *pSrc,   int w1, int h1, const CRect<int> &rcROI1,
       T2 *pPhase, int w2, int h2, const CRect<int> &rcROI2,
```

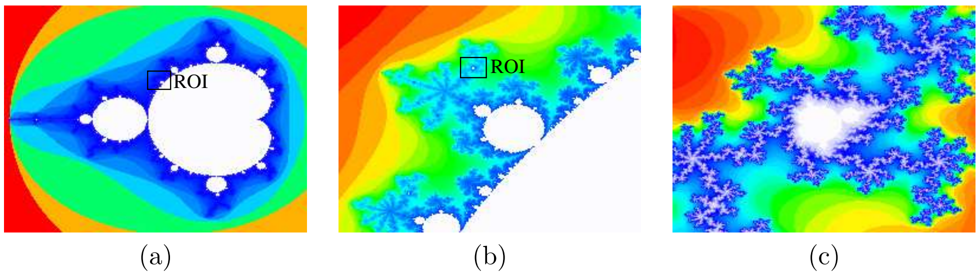


Figure 68: An example of the ImgMandelbrot function. (a) Large scale Mandelbrot image. dX1 = -2.05, dY1 = -1.3, dX2 = 0.7, dY2 = 1.3. (b) Zoom into the ROI shown in (a). (c) Zoom into the ROI shown in (b). The colors of the images are created by applying an intensity-to-color map.

```
       T2 *pBK,    int w3, int h3, const CRect<int> &rcROI3,
       T2 *pCon,   int w4, int h4, const CRect<int> &rcROI4,
 const T2 *pPS, int nStep);
```

Description	General phase-shifting algorithm The general phase-shifting algorithm takes an arbitrary number of (≥3) phase-shifted images to compute a wrapped phase image. The phase-shift angle of each image may also be arbitrary. All input images share the same ROI. T1 can be any integer or float type. T2 must be float or double to store the phase angles $[-\pi, \pi]$.
Parameters	See Table 2 for common parameters
pSrc	pointer to an array of images (size = nStep*w1*h1)
pPhase	output wrapped phase image
pBK	output background image
pCon	output contrast image
pPS	phase-shift value at each step (array size = nStep)
nStep	number of phase-shifting steps (at least 3)

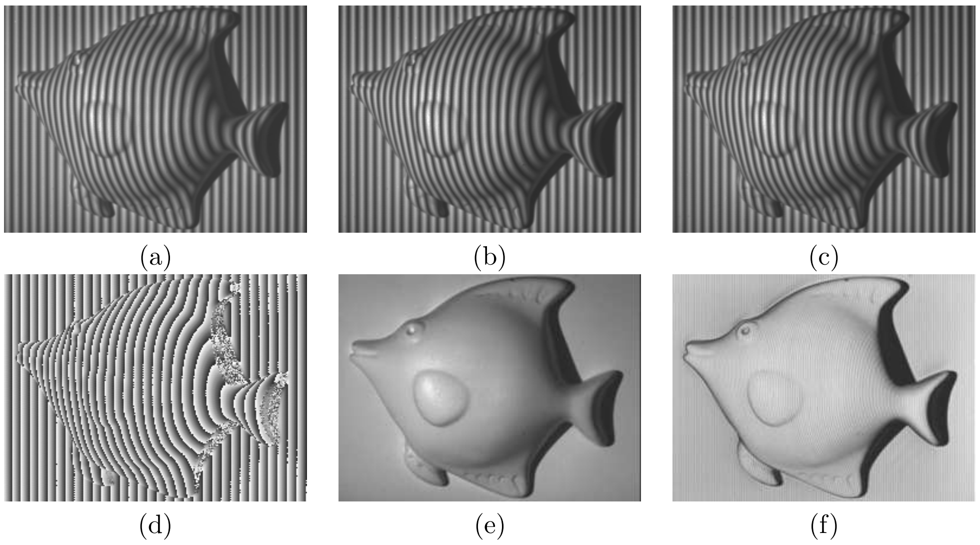


Figure 69: An example of the ImgPhaseShifting function. (a) (b) (c) Three phase-shifted images. Although they look similar, the fringe pattern in each image shifts a constant phase angle. (d) Wrapped phase image. (e) Background intensity image. (f) Fringe contrast image.

3.94 ImgPhaseShifting_Carre

```
template <class T1, class T2> T2 ImgPhaseShifting_Carre
(const T1 *pSrc,   int w1, int h1, const CRect<int> &rcROI1,
       T2 *pPhase, int w2, int h2, const CRect<int> &rcROI2);
```

```
template <class T1, class T2> void ImgPhaseShifting_Carre
(const T1 *pSrc,   int w1, int h1, const CRect<int> &rcROI1,
       T2 *pPhase, int w2, int h2, const CRect<int> &rcROI2,
       T2 *pBK,    int w3, int h3, const CRect<int> &rcROI3,
       T2 *pCon,   int w4, int h4, const CRect<int> &rcROI4);
```

Description	Carré phase-shifting algorithm Carré algorithm takes four images with a constant unknown phase shift. All input images share the same ROI. T1 can be any integer or float type. T2 must be float or double to store the phase angles $[-\pi, \pi]$.
Parameters	See Table 2 for common parameters
pSrc	pointer to an array of images (size = 4*w1*h1)
pPhase	output wrapped phase image
pBK	output background image
pCon	output contrast image
Return	the estimated constant phase shift at each step

3.95 ImgPhaseShifting_5Frame

```
template <class T1, class T2> T2 ImgPhaseShifting_5Frame
(const T1 *pSrc,   int w1, int h1, const CRect<int> &rcROI1,
       T2 *pPhase, int w2, int h2, const CRect<int> &rcROI2);
template <class T1, class T2> void ImgPhaseShifting_5Frame
(const T1 *pSrc,   int w1, int h1, const CRect<int> &rcROI1,
       T2 *pPhase, int w2, int h2, const CRect<int> &rcROI2,
       T2 *pBK,    int w3, int h3, const CRect<int> &rcROI3,
       T2 *pCon,   int w4, int h4, const CRect<int> &rcROI4);
```

Description	5-frame phase-shifting algorithm This algorithm takes five images with a constant unknown phase shift. All input images share the same ROI. T1 can be any integer or float type. T2 must be float or double to store the phase angles $[-\pi, \pi]$.
Parameters	See Table 2 for common parameters
pSrc	pointer to an array of images (size = 5*w1*h1)
pPhase	output wrapped phase image
pBK	output background image
pCon	output contrast image
Return	the estimated constant phase shift at each step

3.96 ImgPhaseUnw_LToR

```
template <class T> void ImgPhaseUnw_LToR
(const T *pWph, int w1, int h1, const CRect<int> &rcROI1,
       T *pUnw, int w2, int h2, const CRect<int> &rcROI2);
```

Description	Left-to-right phase unwrapping algorithm
Parameters	See Table 2 for common parameters
pWph	input wrapped phase
pUnw	output unwrapped phase

3.97 ImgPhaseUnw_QualGui

```
template <class T> void ImgPhaseUnw_QualGui
(const T *pWph,  int w1, int h1, const CRect<int> &rcROI1,
       T *pQual, int w2, int h2, const CRect<int> &rcROI2,
       T *pUnw,  int w3, int h3, const CRect<int> &rcROI3);
```

Description	Quality-guided phase unwrapping algorithm
Parameters	See Table 2 for common parameters
pWph	input wrapped phase
pQual	input quality map
pUnw	output unwrapped phase

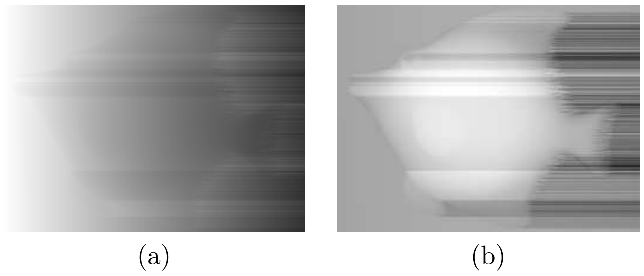


Figure 70: An example of the ImgPhaseUnw_LToR function. Wrapped phase image is as shown in Fig. 69(d). (a) Unwrapped phase image. (b) A linear carrier phase component is removed from the unwrapped phase image.

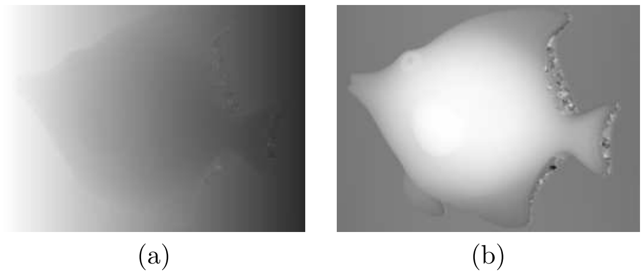


Figure 71: An example of the ImgPhaseUnw_QualGui function. Wrapped phase image is as shown in Fig. 69(d). (a) Unwrapped phase image. (b) A linear carrier phase component is removed from the unwrapped phase image.

3.98 ImgPhaseGradVar

```
template <class T> void ImgPhaseGradVar
(const T *pPhase,   int w1, int h1, const CRect<int> &rcROI1,
       T *pGradVar, int w2, int h2, const CRect<int> &rcROI2);
```

Description	Get phase-gradient-variance image
Parameters	See Table 2 for common parameters
pPhase	input phase
pGradVar	output the inverse of phase gradient variance

3.99 ImgPhaseUnw_MF

```
template <class T> T2 ImgPhaseUnw_MF
(T **ppSrc, int w1, int h1, const CRect<int> &rcROI1,
 T  *pUnw,  int w2, int h2, const CRect<int> &rcROI2,
 int nCount, double dAmp);
```

Description	Multi-frequency phase unwrapping The input wrapped phase images should be in the order of increasing frequencies.
Parameters	See Table 2 for common parameters
ppSrc	input wrapped phase images
pUnw	output unwrapped phase image
nCount	number of wrapped phase images
dAmp	fringe frequency amplification from one wrapped phase image to the next

3.100 ImgPhaseMapping

```
template <class T> void ImgPhaseMapping
(const T *pRef1, int w1, int h1, const CRect<int> &rcROI1,
 const T *pRef2, int w2, int h2, const CRect<int> &rcROI2,
```

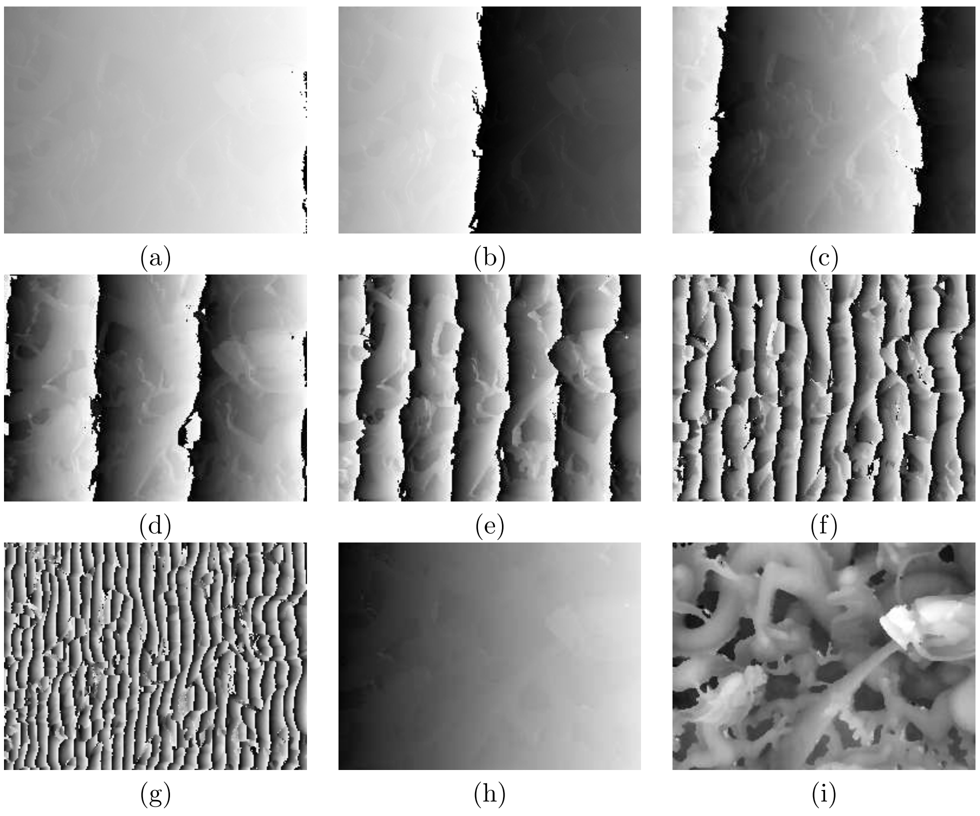


Figure 72: An example of the ImgPhaseUnw_MF function. (a–g) Wrapped phase images obtained from multi-frequency phase-shifted fringe patterns. (h) Unwrapped phase image. (i) Phase image after removing the carrier phase component.

```
 const T *pObj,  int w3, int h3, const CRect<int> &rcROI3,
       T *pHei,  int w4, int h4, const CRect<int> &rcROI4);
```

Description	Phase mapping algorithm Calculate an object surface height by mapping its phase value to those of two parallel reference planes. The input unwrapped phase maps should come from vertical fringe patterns with increasing phase values in x direction.
Parameters	See Table 2 for common parameters
pRef1	phase map of reference plane 1
pRef2	phase map of reference plane 2
pObj	phase map of an object
pHei	surface height of the object

3.101 ImgFaceDetection

```
template <class T1, class T2> void ImgFaceDetection
(const T1 *pImg, int w, int h, const CRect<int> &rcROI,
 T1 tContrast, int nMinFaceW, int nMaxFaceW,
 vector<T2> *pvSum, vector<T2> *pvSum2,
 vector<CLine2D<int> > *pvlnEye, vector<CRect<int> > *pvrcFace = 0);
```

Description	Face detection algorithm
Parameters	See Table 2 for common parameters
tContrast	min contrast of a valid face region
nMinFaceW	minimum possible face width in pixel
nMaxFaceW	maximum possible face width in pixel
pvlnEye	line from the left to the right eye ball of each detected face

continued on next page

pvrcFace	region of each detected face (For debugging purpose, input 0 to ignore.)
pvSum	sum of left-top region pixels
pvSum2	sum of square of left-top region pixels' square
	pvSum and pvSum2 are used internally. Their memory is allocated only if necessary.
Return	the number of faces detected

3.102 ImgSquareDetection

```
template <class T1, class T2> void ImgSquareDetection
(const T1 *pImg, int w, int h, const CRect<int> &rcROI,
 T1 tContrast, int nMinSqu, int nMaxSqu, vector<T2> *pvSum,
 vector<CRect<int> > *pvRect);
```

Description	Detect black squares in an image Any square smaller than 6 pixels in width or larger than the image width or height/1.4 is ignored.
Parameters	See Table 2 for common parameters
tContrast	min contrast of a valid square region
nMinSqu	minimum possible square size in pixel
nMaxSqu	maximum possible square size in pixel
pvSum	sum of left-top region pixels, used internally. Its memory is allocated only if necessary.
pvRect	region of each detected square
Return	the number of squares detected

3.103 ImgBlackSquareGrid

```
template <class T1, class T2> bool ImgBlackSquareGrid
(const T1 *pImg, int w, int h, const CRect<int> &rcROI,
 T1 tContrast, int nMinSqu, int nMaxSqu, vector<T2> *pvSum,
 vector<CPt2D<double> > *pvptCor, int *pnCx, int *pnCy,
 vector<CRect<int> > *pvRect = 0, vector<CPt2D<int> > *pvptEdge = 0,
 vector<CLine2D<int> > *pvlnEdge = 0, bool bRough = false);
```

Description	Detect a black square grid in an image In camera calibration, a black-square-grid pattern is often used. This function detects the corners of all the squares.
1	Detect squares using ImgSquareDetection(..).
2	Estimate corner locations by intersecting lines with outer squares. The lines are derived from the center of the four squares at the grid corner.
3	Refine corner locations of each square by intersecting edge lines, which are fitted to a square's edge points.
Parameters	See Table 2 for common parameters
tContrast	min contrast of a valid square region
nMinSqu	minimum possible square size in pixel
nMaxSqu	maximum possible square size in pixel
pvSum	sum of left-top region pixels, used internally. Its memory is allocated only if necessary.
pvptCor	the corner points of black square grid detected
pnCx	number of square corners in each row. It is twice the number of squares in each row.
pnCy	number of square corners in each column. It is twice the number of squares in each column.
pvRect	rough location of each square (input 0 to ignore)
pvptEdge	edge points on each square (input 0 to ignore)
pvlnEdge	edge lines of each square (input 0 to ignore)

continued on next page

bRough	indicate if search for rough corners only
Return	true: a black square grid is detected; false: squares are too small or the number of squares is not identical in each row or column.

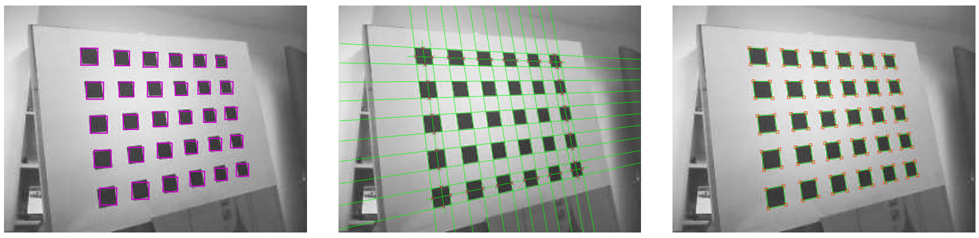

(a) (b) (c)

Figure 73: Examples of the ImgSquareDetection and ImgBlackSquareGrid functions. (a) ImgSquareDetection. (b) ImgBlackSquareGrid with bRough set to true. (c) ImgBlackSquareGrid with bRough set to false (retrieve refined corners).

3.104 ImgReadASCII

```
template <class T> bool ImgReadASCII
(const char *pcFileName, vector<T> *pvImg, int *pW, int *pH);
```

Description	Read image data from ASCII files Applicable to integer, float ...; Not to char, unsigned char.
Parameters	See Table 2 for common parameters
pcFileName	file name
pvImg	output image container
pW	image width
pH	image height
Return	true: succeeded; otherwise, false.

3.105 ImgSaveASCII

```
template <class T> bool ImgSaveASCII
(const T *pImg, int w, int h, const CRect<int> &rcROI,
 const char *pcFileName);
```

Description	Save image data to an ASCII file Int, float type will be saved as in numbers. char, unsigned char will be saved as characters, which may be undesired. So, if you want to save char value, convert it to int first.
Parameters	See Table 2 for common parameters
pcFileName	file name
Return	true: succeeded; otherwise, false.

3.106 ImgReadRaw

```
template <class T> bool ImgReadRaw
(T *pImg, int w, int h, const CRect<int> &rcROI,
 const char *pcFileName, int nOffset);
```

Description	Read image data from a binary file The raw image data must match the data type T. The size of the data to be read is ROI.Area(). If the file cannot be opened, pImg is unchanged.
Parameters	See Table 2 for common parameters
pcFileName	file name
nOffset	offset in bytes from the beginning of the file, where raw image data start to be read.
Return	true: succeeded; otherwise, false.

3.107 ImgSaveRaw

```
template <class T> bool ImgSaveRaw
(const T *pImg, int w, int h, const CRect<int> &rcROI,
 const char *pcFileName, bool bAppend = false);
```

Description	Save raw image data as a binary file
Parameters	See Table 2 for common parameters
pcFileName	file name
bAppend	whether append to the end of the file or create a new file. If 'pcFileName' does not exist, a new file will be created.
Return	true: succeeded; otherwise, false.

3.108 C360Conto: 360-degree contour-generation class

```
template <class T> class C360Conto
{
public:
 C360Conto();
 bool Gen(const T *pImg, int w, int h, const CRect<int> &rcROI,
          int nHor, int nVer, int nTop, int nBottom,
          double dScale, double dLimit, double dWidth, double dSlot,
          vector<CPoly2D<double> > *pvV, vector<CRing2D<double> > *pvH,
          vector<CPt2D<int> > pvptID[2], vector<double> pv3[2]);
};
```

GenRib	Generate contour from a 360-degree surface image Generates horizontal and vertical contour of a surface in the polar coordinate system.
1	The x and y coordinates of the input image denote the polar distance and height respectively. The width of the ROI is assumed to span 360 degrees.
2	A contour size is converted from pixel to mm by dScale.
3	dLimit specifies the longest possible contour size. Contours longer than dLimit are divided into shorter pieces.
4	dSlot specifies the slot size for interlocking the contours in two directions. It is the thickness of sheets from which the contours are cut.
Parameters	See Table 2 for common parameters
nHor	number of horizontal layers
nVer	number of vertical layers
nTop	start row of the top layer (pixel)
nBottom	end row of the bottom layer (pixel)
dScale	pixel scale (mm/pixel)
dLimit	size limit of a contour (mm)
dWidth	width of a contour (mm)
dSlot	slot size (mm)
pvV	store vertical contours
pvH	store horizontal contours
pvptID	ID of contours. pvptID[0] for vertical and [1] for horizontal contours. For each element, x coord is the 0-based index of the contour; y coord is the 0-based index of a segment of a contour. If a contour is complete (no segment), y is -1.
pv3	the 3rd dimenson information of the contours. pv3[0] for vertical and [1] for horizontal contours. pv3[0] is the radial angle. pv3[1] is the z coordinate.
Return	true: succeeded; otherwise, false.

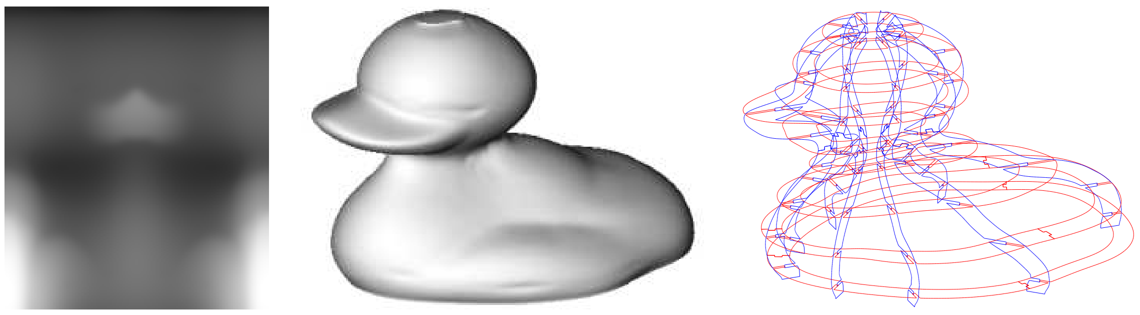

(a) (b) (c)

Figure 74: An example of the Gen function. (a) Source image. (b) Surface. (c) Contours.

4 3D geometry classes and functions

4.1 CPt3D: 3D point class

```
template <class T> class CPt3D
{
public:
 T x, y, z;

 CPt3D() : x(0), y(0), z(0) { }
 CPt3D(const CPt3D<T> &pt) : x(pt.x), y(pt.y), z(pt.z) { }
 CPt3D(T tx, T ty, T tz)   : x(tx),   y(ty),   z(tz)   { }
 CPt3D<T>& operator =  (const CPt3D<T> &pt);
 CPt3D<T>& operator += (const CPt3D<T> &pt);
 CPt3D<T>& operator -= (const CPt3D<T> &pt);
 CPt3D<T>  operator +  (const CPt3D<T> &pt) const;
 CPt3D<T>  operator -  (const CPt3D<T> &pt) const;
 bool      operator == (const CPt3D<T> &pt) const;
 bool      operator != (const CPt3D<T> &pt) const;
 bool      operator <  (const CPt3D<T> &pt) const;

 void GetCoord(T pC[3]) const    { pC[0] = x;  pC[1] = y;  pC[2] = z;  }
 void SetPoint(T tx, T ty, T tz) { x     = tx; y     = ty; z     = tz; }
 void Rotate(const CPt3D<double> &ptVec,
             const CPt3D<double> &ptArb, double dAngle);
 void   Normalize();
 double Norm()   const;
 double NormSq() const;
};

template <class T>
CPt3D<T> operator * (T val, const CPt3D<T> &pt);

template <class T>
CPt3D<T> operator * (const CPt3D<T> &pt, T val);
```

Variable	Description
x	x coordinate of the point
y	y coordinate of the point
z	z coordinate of the point

Function	Description
operator =	Assignment operator
operator + =	*this = *this + pt
operator − =	*this = *this − pt
operator +	Return *this + pt
operator −	Return *this − pt
operator ==	Check if *this is equal to pt
operator !=	Check if *this is not equal to pt
operator <	compare two points for sorting
operator *	Each x, y, z coordinate multiples a value

Function	Description
GetCoord	Get x, y and z coordinates and put them in an array
SetPoint	Set the x, y and z coordinates of the point
Rotate	Rotate the point with respect to an arbitrary line ptVec is the unit vector of the line direction. ptArb is a point on the line

continued on next page

	dAngle is the angle of rotation in rad (right-hand rule: the thumb points from pt in the direction of ptVec; the other four fingers indicate the rotating direction.)
Normalize	Normalize a vector
Norm	Get the norm of a vector
NormSq	Get the norm square of a vector

4.2 CSize3D: 3D size class

```
template <class T> class CSize3D
{
public:
 T cx, cy, cz;

 CSize3D() : cx(0), cy(0), cz(0) { }
 CSize3D(const CSize3D<T> &size);
 CSize3D(T tcx, T tcy, T tcz) : cx(tcx), cy(tcy), cz(tcz) { }
 CSize3D<T>& operator =  (const CSize3D<T> &size);
 CSize3D<T>& operator += (const CSize3D<T> &size);
 CSize3D<T>& operator -= (const CSize3D<T> &size);
 CSize3D<T>  operator +  (const CSize3D<T> &size) const;
 CSize3D<T>  operator -  (const CSize3D<T> &size) const;
 bool        operator == (const CSize3D<T> &size) const;
 bool        operator != (const CSize3D<T> &size) const;

 void SetSize(T tcx, T tcy, T tcz) { cx = tcx; cy = tcy; cz = tcz; }
};
```

Variable	Description
cx	size in x direction
cy	size in y direction
cz	size in z direction

Function	Description
operator =	Assignment operator
operator + =	*this = *this + size
operator − =	*this = *this − size
operator +	Return *this + size
operator −	Return *this − size
operator ==	Check if *this is equal to size
operator !=	Check if *this is not equal to size

4.3 CLine3D: 3D line segment class

```
template <class T> class CLine3D
{
public:
 CLine3D() : m_ptStart(0,0,0), m_ptEnd(0,0,0) { }
 CLine3D(const CLine3D<T> &line);
 CLine3D(T x1, T y1, T z1, T x2, T y2, T z2);
 CLine3D(const CPt3D<T> &ptStart, const CPt3D<T> &ptEnd);
 CLine3D<T>& operator =  (const CLine3D<T> &line);
 bool        operator == (const CLine3D<T> &line) const;
 bool        operator != (const CLine3D<T> &line) const;

 CPt3D<T> Start()        const;
 CPt3D<T> End()          const;
 CPt3D<T> Center()       const;
 CPt3D<T> BoundingCube() const;
 double   Length()       const;
 void     SetLine   (T x1, T y1, T z1, T x2, T y2, T z2);
 void     SetLine   (const CPt3D<T> &ptStart, const CPt3D<T> &ptEnd);
 void     SetStart  (T x, T y, T z);
 void     SetStart  (const CPt3D<T> &pt);
 void     SetEnd    (T x, T y, T z);
 void     SetEnd    (const CPt3D<T> &pt);
 void     SetCenter (const CPt3D<T> &pt);
 void     SetLength (double dLength, int nFix);
 void     Offset    (T x, T y, T z);
 void     Offset    (const CPt3D<T> &ptOffset);
 bool     GetPointAt(double dDist, CPt3D<T> *ppt) const;

private:
 CPt3D<T> m_ptStart, m_ptEnd; // Start and end point
};
```

Function	Description
operator =	Assignment operator
operator ==	Check if *this is equal to line
operator !=	Check if *this is not equal to line

Function	Description
Start	Get the start point of the line
End	Get the end point of the line
Center	Get the center of the line
BoundingCube	Get the bounding cube of the line
Length	Get the length of the line
SetLine	Set the start and the end points of the line
SetStart	Set the start point of the line
SetEnd	Set the end point of the line
SetCenter	Set the center of the line
Offset	Offset the position of the line

SetLength	Set the length of the line To modify the line length, one can fix the start point, the end point, or the center. nFix specifies which point to fix. The angle of the line is always fixed. If the start point and the end point are the same, the line will be extended in the x direction only.
dLength	the new length of the line, which should be >= 0. If dLength is < 0, it is treated as 0.
nFix	0: the start point of the line is fixed; 1: the center of the line is fixed; 2: the end point of the line is fixed.

GetPointAt	Get a point on the line that has a specified distance from the star point. Take the start point as a reference, dDist > 0 indicates the point is at the same side as the end point; while dDist < 0 indicates the point is at the opposite side of the end point.
dDist	the distance of the point from the start point.
ppt	the point found
Return	true: point found; false: cannot find such a point.

4.4 CCube: cube class

```
template <class T> class CCube
{
public:
 T left, top, front, right, bottom, back;

 CCube() : left(0), top(0), front(0), right(0), bottom(0), back(0) { }
```

```
  CCube(const CCube<T> &cube);
  CCube(T lt, T tp, T ft, T rt, T bm, T bk);
  CCube(const CPt3D<T> &ptLTF, const CPt3D<T> &ptRBB);
  CCube(const CRect<T>& rect, T ft, T bk);
  CCube<T>& operator =  (const CCube<T> &cube);
  CCube<T>& operator &= (const CCube<T> &cube);
  CCube<T>& operator |= (const CCube<T> &cube);
  CCube<T>  operator &  (const CCube<T> &cube) const;
  CCube<T>  operator |  (const CCube<T> &cube) const;
  bool      operator == (const CCube<T> &cube) const;
  bool      operator != (const CCube<T> &cube) const;

  bool       IsEmpty() const;
  T          Width()   const;
  T          Height()  const;
  T          Depth()   const;
  T          Volume()  const;
  CSize3D<T> Size()    const;
  CPt3D<T>   Center()  const;
  CPt3D<T>   LTF() const;
  CPt3D<T>   LTB() const;
  CPt3D<T>   LBF() const;
  CPt3D<T>   LBB() const;
  CPt3D<T>   RTF() const;
  CPt3D<T>   RTB() const;
  CPt3D<T>   RBF() const;
  CPt3D<T>   RBB() const;
  void       SetCube(T lt, T tp, T ft, T rt, T bm, T bk);
  void       Inflate(T lt, T tp, T ft, T rt, T bm, T bk);
  void       Deflate(T lt, T tp, T ft, T rt, T bm, T bk);
  void       Inflate(T x, T y, T z);
  void       Deflate(T x, T y, T z);
  void       Offset (T x, T y, T z);
  void       Offset (const CPt3D<T> &ptOffset);
  bool       PtIn   (T x, T y, T z)      const;
  bool       PtIn   (const CPt3D<T>& pt) const;
  bool       Regularize();
};
```

Variable	Description
left	left bound of the cube
top	top bound of the cube
front	front bound of the cube
right	right bound of the cube
bottom	bottom bound of the cube
back	back bound of the cube

Function	Description
operator =	Assignment operator
operator &=	*this = Intersection of two cubes
operator |=	*this = Union of two cubes
operator &	Return intersection of two cubes
operator |	Return union of two cubes
operator ==	Check if *this is equal to cube
operator !=	Check if *this is not equal to cube

Function	Description
IsEmpty	Check if it is an empty cube. A cube is empty if its width, height or depth is <= 0.

continued on next page

Width	Get the width ($right - left$) of the cube
Height	Get the height ($bottom - top$) of the cube
Depth	Get the depth ($back - front$) of the cube
Size	Get the size of the cube
Center	Get the center of the cube
Volume	Get the volume of the cube
LTF	Get the left-top-front point of the cube
LTB	Get the left-top-back point of the cube
LBF	Get the left-bottom-front point of the cube
LBB	Get the left-bottom-back point of the cube
RTF	Get the right-top-front point of the cube
RTB	Get the right-top-back point of the cube
RBF	Get the right-bottom-front point of the cube
RBB	Get the right-bottom-back point of the cube

Function	Description
SetCube	Set the left, top, front, right, bottom and back value of the cube
Inflate	Inflate the cube by lt, tp, ft, rt, bm, bk on the left, top, front, right, bottom and back sides respectively. Inflate the cube by x on the left and right sides, by y on the top and bottom sides, and by z on the front and back sides respectively.
Deflate	Deflate the cube by lt, tp, ft, rt, bm, bk on the left, top, front, right, bottom and back side respectively. Deflate the cube by x on the left and right sides, by y on the top and bottom sides, and by z on the front and back sides respectively.
Offset	Offset the position of the cube
PtIn	Check if a point is inside the cube. A point is side a cube, if $pt.x \in [left, right)$ AND $pt.y \in [top, bottom)$ AND $pt.z \in [front, back)$,
Regularize	Make sure the cube has non-negative width, height and depth. If $left > right$, swap left and right. If $top > bottom$, swap top and bottom. If $front > back$, swap front and back.

4.5 CTri3D: 3D triangle class

```
template <class T> class CTri3D
{
public:
 CTri3D() { }
 CTri3D(const CTri3D<T> &triangle);
 CTri3D<T>& operator = (const CTri3D<T> &triangle);
 CPt3D<T>   operator [] (int idx) const { return m_ppt[idx]; }
 CPt3D<T>&  operator [] (int idx)       { return m_ppt[idx]; }
 bool       operator == (const CTri3D<T> &tri);

 const CPt3D<T>& Vertex(int idx) const;
       CPt3D<T>  Center()        const;
 void GetCoord (T pCoord[9]) const;
 void SetVertex(int idx, const CPt3D<T> &pt);
 void Offset   (T x, T y, T z);
 void Offset   (const CPt3D<T> &ptOffset);
 void Rescale  (double dXScale, double dYScale, double dZScale,
               const CPt3D<T> &ptRef);
 void ReverseVertexSequence();

private:
 CPt3D<T> m_ppt[3];
};
```

Function	Description
operator =	Assignment operator
operator [ ]	Get vertex operator. Valid index is 0, 1, and 2. The 1st operator [ ] is for retrieving a vertex value. The 2nd operator [ ] is for setting a vertex value.
operator ==	Check if *this is equal to tri
Vertex	Get a vertex value. Valid index is 0, 1, and 2.
GetCoord	Get vertices' coordinates and put them in an array
SetVertex	Set a vertex value. Valid index is 0, 1, and 2.
Offset	Offset the position of the triangle
Rescale	Rescale the triangle with respect to a reference point
ReverseVertexSequence	Reverse vertex sequence. Example: Old sequence: tri[0] = pt1, tri[1] = pt2, tri[2] = pt3. New sequence: tri[0] = pt3, tri[1] = pt2, tri[2] = pt1.

4.6 CPoly3D: 3D polygon class

```
template <class T> class CPoly3D
{
public:
 CPoly3D() { }
 CPoly3D(const CPoly3D<T> &poly);
 CPoly3D(const vector<CPt3D<T> > &vpt);
 CPoly3D(int nCount, const CPt3D<T> &pt);
 CPoly3D<T>& operator = (const CPoly3D<T> &poly);

 int   Count() const { return int(m_vpt.size()); }
 const CPt3D<T>& Vertex(int idx) const;
 void  AddVertex(        const CPt3D<T> &pt);
 void  SetVertex(int idx, const CPt3D<T> &pt);
 void  Clear() { m_vpt.clear(); }
 void  ReverseVertexSequence();

private:
 vector<CPt3D<T> > m_vpt;
};
```

Function	Description
operator =	Assignment operator
Count	Get the number of vertices of the polygon
Vertex	Get the idx-th vertex of the polygon
AddVertex	Add a vertex at the end of the vertex list
SetVertex	Set the idx-th vertex value
Clear	Remove all vertices of the polygon
ReverseVertexSequence	Reverse vertex sequence

4.7 CRing3D: 3D ring class

```
template <class T> class CRing3D
{
public:
 CRing3D() { }
 CRing3D(const CRing3D<T> &ring);
 CRing3D(const vector<CPt3D<T> > &vptO);
 CRing3D(const CPt3D<T> *ppt, int nCount);
 CRing3D<T>& operator = (const CRing3D<T> &ring);

 int CountIRim() const { return int(m_vvptI.size()); }
 const vector<CPt3D<T> >& ORim()      const { return m_vptO;     }
       vector<CPt3D<T> >& ORim()            { return m_vptO;     }
 const vector<CPt3D<T> >& IRim(int i) const { return m_vvptI[i]; }
       vector<CPt3D<T> >& IRim(int i)       { return m_vvptI[i]; }
 void     SetRing(      vector<CPt3D<T> > *pvpt);
 void     SetORim(const vector<CPt3D<T> > &vpt);
 void     SetORim(      vector<CPt3D<T> > *pvpt);
 void     SetORim(const vector<CPt3D<T> > &vpt);
 void     AddIRim(      vector<CPt3D<T> > *pvpt);
 void     AddIRim(const CPt3D<T> *ppt, int nCount);
 void     Clear();
 CCube<T> BoundingCube() const;
 void     Partition(double dLimit, vector<CRing3D<T> > *pvRing) const;

private:
 vector<CPt2D<T> >          m_vptO;  // outer rim  of the ring
 vector<vector<CPt2D<T> > > m_vvptI; // inner rims of the ring
};
```

Function	Description
operator =	Assignment operator
CountIRim	Get the number of inner rims
ORim	Get the outer rim
IRim	Get an inner rim
SetRing	Set the ring
SetORim	Add the outer rim
AddIRim	Add an inner rim
Clear	Clear outer and inner rims
BoundingCube	Get the bounding cube of the ring
Partition	Partition the ring into multiple rings Partition is achieved by rotating the ring to the XY plane, partitioning a 2D ring, and rotating back the rings to 3D. For details see CRing2D::Partition(..).

4.8 CCylinder: cylinder class

```
template <class T> class CCylinder
{
public:
 CCylinder() : m_Radius(0) { }
 CCylinder(const CCylinder<T> &cylinder);
 CCylinder(const CLine3D<T> &lnCenter, T radius);
 CCylinder<T>& operator = (const CCylinder<T> &cylinder);

 CLine3D<T> CenterLine() const { return m_lnCenter; }
 T          Radius()     const { return m_Radius;   }
 void       SetCenterLine(const CLine3D<T> &lnCenter);
 void       SetRadius(T radius) { m_Radius = radius; }
 void       SetCylinder(const CLine3D<T> &lnCenter, T radius);

private:
 CLine3D<T> m_lnCenter; // center line of the cylinder
 T          m_Radius;   // radius of the cylinder
};
```

Function	Description
operator =	Assignment operator
CenterLine	Get the center line of the cylinder
Radius	Get the radius of the cylinder
SetCenterLine	Set the center line of the cylinder
SetRadius	Set the radius of the cylinder
SetCylinder	Set the center line and radius of the cylinder

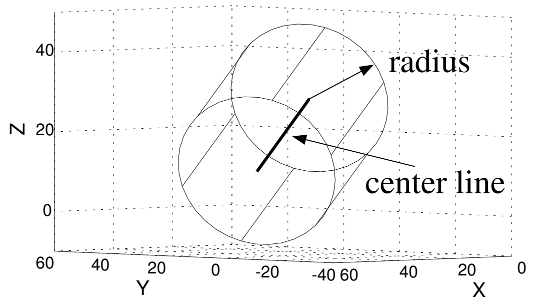


Figure 75: An example of a cylinder.

4.9 CrossProduct

```
template <class T> CPt3D<T> CrossProduct
(const CPt3D<T> &pt1, const CPt3D<T> &pt2);
```

Description	Get the cross product of two vectors
pt1	1st vector
pt2	2nd vector
Return	cross product of the two vectors

4.10 DotProduct

```
template <class T> double DotProduct
(const CPt3D<T> &pt1, const CPt3D<T> &pt2);
```

Description	Get the dot product of two vectors
pt1	1st vector
pt2	2nd vector
Return	dot product of the two vectors

4.11 Distance

```
template <class T> double Distance
(const CPt3D<T> &pt1, const CPt3D<T> &pt2);
```

Description	Get the distance between two points
pt1	1st point
pt2	2nd point
Return	the distance between the two points

4.12 DistPtLn

```
template <class T> double DistPtLn
(const CPt3D<T> &pt, const CPt3D<T> &ptArb1, const CPt3D<T> &ptArb2);
```

Description	Get the distance from a point to a plane
pt	the point
ptArb1	first arbitrary point on the line
ptArb2	second arbitrary point on the line
Return	distance from the point to the line

4.13 DistPtPl

```
template <class T> double DistPtPl
(const CPt3D<T> &pt, const CPt3D<T> &ptVecPl, const CPt3D<T> &ptArbPl);
```

Description	Get the distance from a point to a plane
pt	the point
ptVecPl	normal vector of the plane
ptArbPl	an arbitrary point on the plane
Return	distance from the point to the plane

4.14 FootOfPerpendicular

```
template <class T> CPt3D<T> FootOfPerpendicular
(const CPt3D<T> &pt, const CPt3D<T> &ptVecPl, const CPt3D<T> &ptArbPl);
```

Description	Get the foot of perpendicular from a point to a plane.
pt	input point
ptVecPl	normal vector of the plane
ptArbPl	an arbitrary point on the plane
Return	foot of perpendicular

4.15 MidPoint

```
template <class T> CPt3D<T> MidPoint
(const CPt3D<T> &pt1, const CPt3D<T> &pt2);
```

Description	Get the midpoint of two points
pt1	1st point
pt2	2nd point
Return	midpoint of two points

4.16 BisectingPlane

```
template <class T> bool BisectingPlane
(const CPt3D<T> &pt1, const CPt3D<T> &pt2,
 double *pdA, double *pdB, double *pdC, double *pdD);
```

Description	Get the bisecting plane of two points
pdA, . . . pdD	parameters of the plane equation A*x + B*y + C*z = D
Return	true: bisecting plane found; otherwise, false.

4.17 Area2

```
template <class T> double Area2
(const CPt3D<T> &pt1, const CPt3D<T> &pt2, const CPt3D<T> &pt3);
```

Description	Get twice the area enclosed by three points
pt1, pt2, pt3	input points
Return	the area enclosed by three points

```
template <class T> double Area2
(const CPt3D<T> *ppt, int nCount, const CPt3D<T> &ptNV);
```

Description	Get twice the area enclosed by a polygon
ppt	vertices of a polygon
nCount	size of ppt
ptNV	unit normal vector of the polygon
Return	the area enclosed by the polygon

4.18 Angle

```
template <class T> double Angle
(const CPt3D<T> &pt1, const CPt3D<T> &pt2);
```

Description	Get the angle in $[0, \pi]$ formed by two vectors
pt1	1st vector
pt2	2nd vector
Return	angle formed by the two vectors

4.19 Collinear

```
template <class T> bool Collinear
(const CPt3D<T> &pt1, const CPt3D<T> &pt2, const CPt3D<T> &pt3);
```

Description	Check collinearity of three points
pt1, pt2, pt3	input points
Return	true: collinear; otherwise, false.

4.20 NormalVector

```
template <class T> CPt3D<T> NormalVector
(const CPt3D<T> &pt1, const CPt3D<T> &pt2, const CPt3D<T> &pt3);
```

Description	Get the normal vector of a plane Get the normal vector of a plane specified by three points. The direction of the normal vector follows the right-hand rule: pt1 $\rightarrow$ pt2 $\rightarrow$ pt3. If the three points are collinear, return (0,0,0).
pt1, pt2, pt3	input points
Return	normal vector (unnormalized)

```
template <class T> CPt3D<T> NormalVector
(const CPt3D<T> *ppt, int nCount);
```

Description	Get the normal vector of a polygon If a surface is specified by a general polygon, the Newell's method is a good way to calculate the normal. It produces an average normal if the polygon is not quite planar. It also is not confused by collinear vertices.
ppt	an array of points
nCount	size of ppt
Return	normal vector (normalized)

4.21 Volume

```
template <class T> double Volume
(const CPt3D<T> &pt0, const CPt3D<T> &pt1,
 const CPt3D<T> &pt2, const CPt3D<T> &pt3);
```

Description	Get the volume of a tetrahedron The tetrahedron's four vertices are given by pt0, ... pt3. If right-hand rule applies (Four fingers curve in the order of pt1, pt2, pt3; and the thumb points to pt0.), the volume is positive; otherwise the volume is negative.
pt0, ... pt3	tetrahedron's 4 vertices
Return	volume of the tetrahedron
Note	Internal volume computation is based on double; therefore, for integer data type, it is better to determine the volume sign by testing if the volume is $>$ 0.5 or $<$ -0.5 rather than $>$ 0 or $<$ 0. This helps to prevent float precision error.

4.22 RotationMat

```
template <class T> void RotationMat
(const CPt3D<T> &ptAxis, T angle, T pR[9]);
```

Description	Calculate rotation matrix Calculate rotation matrix based on an axis of rotation and an angle of rotation. T should be float or double.
ptAxis	a unit vector specifying the axis of rotation
dAngle	angle of rotation
pR	calculated 3-by-3 rotation matrix

4.23 BoundingCube

```
template <class T> CCube<T> BoundingCube
(const T *pCoord, int nCount);
template <class T> CCube<T> BoundingCube
(const CPt3D<T> *pptVtx, int nCount);
```

Description	Get the bounding cube of a set of points The bounding cube's left, right, top, bottom, front and back equal the min x, max x, min y, max y, min z and max z coordinates of the input points respectively.
pCoord	coordinates of points (x1, y1, z1, x2, y2, z2, ...)
ppt	an array of points
nCount	size of pCoord or ppt
Return	the bounding cube

4.24 Centroid

```
template <class T> CPt3D<T> Centroid
(const CPt3D<T> *ppt, int nCount);
```

Description	Get the centroid of a set of points
ppt	an array of points
nCount	size of ppt
Return	the centroid of the input points

```
template <class T1, class T2> CPt3D<T1> Centroid
(const CPt3D<T1> *ppt, T2 *pWei, int nCount);
```

Description	Get the centroid of a set of points with weight
pWei	weight of each point

4.25 ClosestPoint

```
template <class T1, class T2> int ClosestPoint
(const CPt3D<T1> *ppt, int nCount, const CPt3D<T2> &pt, double *pd2 = 0);
```

Description	Search for the closest point to a given point
ppt	a point array where the closest point is searched
nCount	size of the point array
pt	the given reference point
pd2	store the square of distance from pt to the closest point in ppt. Input 0 to ignore.
Return	0-based index of the closest point in ppt to pt

4.26 ClosestPointPair

```
template <class T> void ClosestPointPair
(const CPt3D<T> *ppt1, int nCount1, const CPt3D<T> *ppt2, int nCount2,
 int *pnIdx1, int *pnIdx2);
```

Description	Search in two set of points for the closest pair, one from each set.
ppt1	point set 1
nCount1	size of point set 1
ppt2	point set 2
nCount2	size of point set 2
pnIdx1	index of the closest point in ppt1
pnIdx2	index of the closest point in ppt2

4.27 IntsecLnPl

```
template <class T1, class T2> int IntsecLnPl
(const CLine3D<T1> &line, const CPt3D<T1> &ptVecPl,
 const CPt3D<T1> &ptArbPl, CPt3D<T2> *ppt);
```

Description	Get the intersection of a line segment with a plane
line	line segment
ptVecPl	normal vector of the plane
ptArbPl	an arbitrary point on the plane
ppt	store the intersecting point if found
Return	-2: invalid line segment or plane vector, -1: the line is on the plane (*ppt is unchanged), 0: no intersection, 1: intersection found. If the intersection is an end point of a line segment, the following bits in the returned value are set. If so, the returned value is not 1. 3: intersection is line.Start(), the second bit is set, 5: intersection is line.End(), the third bit is set.

```
template <class T1, class T2> int IntsecLnPl
(const CPt3D<T1> &ptVecLn, const CPt3D<T1> &ptArbLn,
 const CPt3D<T1> &ptVecPl, const CPt3D<T1> &ptArbPl, CPt3D<T2> *ppt);
```

Description	Get the intersection of a line with a plane
ptVecLn	directional vector of the line
ptArbLn	an arbitrary point on the line
ptVecPl	normal vector of the plane
ptArbPl	an arbitrary point on the plane
ppt	store the intersecting point if found
Return	-2: invalid line or plane vector, -1: the line is on the plane (*ppt is unchanged), 0: no intersection (the line is parallel to the plane), 1: intersection found.

4.28 IntsecLnTri

```
template <class T1, class T2> int IntsecLnTri
(const CLine3D<T1> &line, const CPt3D<T1> pptTri[3], CPt3D<T2> *ppt);
```

Description	Get the intersection of a line segment with a triangle
line	line segment
pptTri	vertices of the triangle
ppt	store the intersecting point if found
Return	-2: invalid line segment or triangle, -1: the line is on the triangle plane (*ppt is unchanged), 0: no intersection, 1: intersection inside the triangle but not an end point of the line, 2: intersection inside the triangle and is line.Start(), 3: intersection inside the triangle and is line.End(), 4: intersection on one of the edges but not on a vertex but not an end point of the line, 5: intersection on one of the edges but not on a vertex, and is line.Start(), 6: intersection on one of the edges but not on a vertex, and is line.End(), 7: intersection on one of the vertices but not an end point of the line, 8: intersection on one of the vertices and is line.Start(), 9: intersection on one of the vertices and is line.End().

4.29 IntsecLnCube

```
template <class T1, class T2> int IntsecLnCube
(const CLine3D<T1> &line, const CCube<T1> &cube,
 CPt3D<T2> *ppt1, CPt3D<T2> *ppt2);
```

Description	Get the intersection of a line segment with a cube If only one intersection is found, it is stored in ppt1. If two intersections are found, the one closer to the start of the line is ppt1. Improper intersections (touching) are treated as intersections. The x, y and z coordinates of *ppt1 and *ppt2 are within [left, right], [top, bottom] and [front, back] respectively.
line	line segment
cube	cube
ppt1	1st intersection if exist
ppt2	2nd intersection if exist
Return	the number of intersections found (0, 1 or 2)

```
template <class T> bool IntsecLnCube
(const CCube<int> &cube, CLine3D<T> *pLine);
```

Description	intersect a line segment with a cube A line segment is cut to fit inside a cube (integer type). The x, y and z coordinates of its end points are within [left, right), [top, bottom) and [front, back) respectively. Note that the right, bottom and back borders are not inside.
cube	cube
pLine	on input, a line segment; on output, the intersected line segment in cube if exists; otherwise unchanged.
Return	true: pLine is in cube; false: pLine is out of cube.

4.30 IntsecTriPl

```
template <class T1, class T2> bool IntsecTriPl
(const CPt3D<T1> pptTri[3], const CPt3D<T1> &ptVecPl,
 const CPt3D<T1> &ptArbPl, CLine3D<T2> *pLine);
```

Description	Get the intersections of a triangle with a plane If they intersect, the intersection is a line. The start and end points of the line may be the same (a vertex).
pptTri	vertices of the triangle
ptVecPl	normal vector of the plane
ptArbPl	an arbitrary point on the plane
pLine	intersecting line
Return	true: intersections found; false: no intersection.

4.31 PointInMesh

```
template <class T1, class T2> int PointInMesh
(const T1 *pCoord, int nCount, const CCube<T1> &cbMesh, const CPt3D<T2> &pt);
```

Description	Check if a point is in a closed triangle mesh The input mesh must be closed, or the check is meaningless.
pCoord	coordinates of the mesh (x1, y1, z1, ...)
nCount	size of pCoord
cbMesh	bounding cube of the mesh. It is supplied as an input, not calculated in the functino, to save repeated calculation when testing many points.
pt	point to be checked
Return	-1: cannot determine due to degeneracy 0: outside, 1: strictly inside, 2: on one of the faces but not on an edge or a vertex, 3: on one of the edges but not on a vertex, 4: on one of the vertices.

4.32 RotateToXY

```
template <class T> void RotateToXY
(const CPt3D<T> *ppt, int nCount, vector<CPt2D<T> > *pvpt);
```

Description	Rotate points on a 3D plane to the XY plane
ppt	points on a 3D plane
nCount	size of ppt
pvpt	rotated corresponding points on the XY plane

```
template <class T> void RotateToXY
(const CRing3D<T> &r3D, CRing2D<T> *pr2D, T *pR = 0, T *pZ = 0);
```

Description	Rotate a 3D ring to a 2D ring on the XY plane
r3D	3D ring
pr2D	a rotated corresponding ring on the XY plane
pR	rotation matrix (input 0 to ignore)
pZ	z coordinate of the XY plane (input 0 to ignore)

4.33 ReduceVertex

```
template <class T> void ReduceVertex
(vector<CPt3D<T> > *pvpt, double dAngThre, bool bClosed);
```

Description	Reduce the number of vertices of a polygon or a poly line. If the angle formed by a vertex and its adjacent neighbours differs from 0 or PI by an amount less than dAngThre, the vertex is removed from the polygon or the poly line.
pvpt	input vertices. On output, vertices are removed based on the above criterion.
dAngThre	angular threshold
bClosed	true for polygon, false for poly line

4.34 RemoveIntsec

```
template <class T> void RemoveIntsec
(vector<CPt3D<T> > *pvpt);
```

Description	Remove intersection in a polygon.
pvpt	input vertices. On output, intersections are removed.

4.35 ConvexHull

```
template <class T> bool ConvexHull
(vector<CPt3D<T> >   *pvptIn, vector<CPt3D<T> >   *pvptOut,
 vector<CLine3D<T> > *pvLine, vector<CPoly3D<T> > *pvPoly,
 bool bTriFace);
```

Description	Get the 3D convex hull of a set of points Reference “Computational geometry in C”, Joseph O’Rourke. Based on the algorithm in the book, the convex hull is composed of triangles, including coplanar triangles. I add a post-processing step to output non-coplanar polygon faces of the hull. ConstructPolygon(..) is for this purpose.
pvptIn	input point set; (modified during processing)
pvptOut	output convex hull vertexes (input 0 to ignore)
pvLine	output convex hull edges (input 0 to ignore)
pvPoly	output convex hull polygon faces (must NOT be 0)
bTriFace	if true, output triangular faces that may be coplanar. This is used for DelaunayTriangulation. if false, output non-coplanar polygon faces.
Return	true: succeeded; false: failed (points are coplanar).

4.36 ConstructKDTree

```
template <class T> CKDNode<T,3>* ConstructKDTree
(const CPt3D<T> *ppt, int nCount);
```

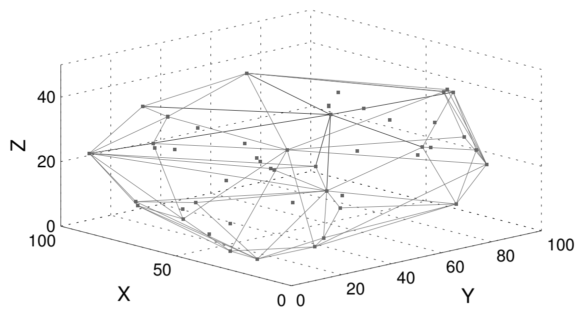


Figure 76: An example of the ConvexHull function.

Description	Construct a 3-D tree
ppt	point set to construct the tree
nCount	size of ppt
Return	pointer to the root node of the tree. The caller is responsible for deleting the pointer after use.

4.37 FitPlane

```
template <class T1, class T2> bool FitPlane
(const CPt3D<T1> *ppt, const T2 *pWei, int nCount,
 double *pdKx, double *pdKy, double *pdZ0);
```

Description	Least squares fitting of a plane Plane equation z(x,y) = Kx*x + Ky*y + Z0
ppt	points to be fitted (size must be nCount)
pWei	weighting factor array (input 0 to ignore)
nCount	size of ppt array and weighting array
pdKx, . . . pdZ0	parameters of the plane equation, if succeed; or undefined.
Return	true: succeeded; false: failed.

```
template <class T1, class T2, class T3> bool FitPlane
(const CPt3D<T1> *ppt, const T2 *pWei, int nCount,
 CPt3D<T1> *pptCen, CPt3D<T3> *pptNV);
```

Description	Least squares fitting of a plane The fitted plane passes through the centroid of the input points and obtains a normal vector. This function is able to fit a plane parallel to the Z axis.
ppt	points to be fitted (size must be nCount)
pWei	weighting factor array (input 0 to ignore)
pptCen	centroid of the input point
pptNV	unit normal vector of the plane
Return	true: succeeded; false: failed.

4.38 FitSphere

```
template <class T> bool FitSphere
(const CPt3D<T> *ppt, int nCount,
 CPt3D<double> *pptCenter, double *pdRadius);
```

Description	Least squares fitting of a sphere Sphere equation: $(x-a)^2 + (y-b)^2 + (z-c)^2 = R^2$ Internally fit the modified equation: $Ax + By + Cz + D = x^2 + y^2 + z^2$
ppt	points to be fitted (size must be nCount)
nCount	size of ppt array
pptCenter	store the center of the sphere if succeeded
pdRadius	store the radius of the sphere if succeeded
Return	true: succeeded; false: failed.

4.39 FitPolynomialSurf

```
template <class T1, class T2> bool FitPolynomialSurf
(const CPt3D<T1> *ppt, const T2 *pWei, int nCount, int N,
 vector<double> *pvCoe, vector<int> *pvPowOfX, vector<int> *pvPowOfY);
```

Description	Least squares fitting of a polynomial surface Polynomial surface equation: z(x,y) = $+C_0 + C_1x + C_3x^2 + C_6x^3 + ... + C_?x^N$ $+C_2y + C_4xy + C_7x^2y + ...$ $+C_5y^2 + C_8xy^2 + ...$ $+... + ... + ...$ $+C_{(N+1)(N+2)/2-1}y^N$
ppt	points to be fitted (size must be nCount)
pWei	weighting factor array (input 0 to ignore)
N	the order of the surface equation (should be >= 0)
pvCoe	store coefficients, if succeed; or undefined.
pvPowOfX	store power of x of each term, if succeed; or undefined.
pvPowOfY	store power of y of each term, if succeed; or undefined.
	After processing, the size of pvCoe, pvPowOfX and pvPowOfY will be (N+1)(N+2)/2 and their elements are ordered according to the index of C in the equation.
Return	true: succeeded; false: failed.

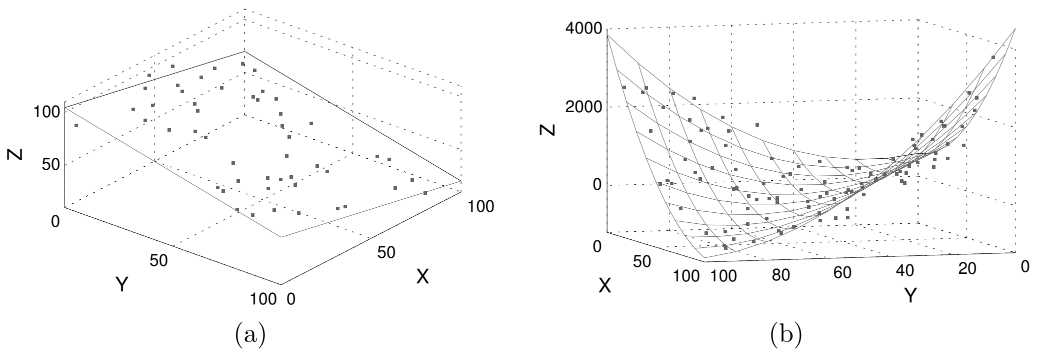


Figure 77: Examples of the (a) FitPlane and the (b) FitPolynomialSurf functions.

4.40 FitGauss

```
template <class T> bool FitGauss
(const CPt3D<T> *ppt, int nCount, double pdCoe[5]);
```

Description	Least squares fitting of a 2D Gaussian function Gaussian function: $z = Ae^{-(x-B)^2/(2C)-(y-D)^2/(2E)}$. Fitting is based on a linear least squares method: the z coordinate of all points is converted to ln(z), which requires that z > 0.
ppt	points to be fitted (size must be nCount)
nCount	size of ppt array
pdCoe	parameters of the Gaussian function pdCoe[0], pdCoe[1], ... pdCoe[4] are A, B, ... E respectively.
Return	true: succeeded; false: failed.

4.41 PtcICP

```
template <class T1, class T2> int PtcICP
(const CPt3D<T1> *ppt1, int nCount1, const CPt3D<T1> *ppt2, int nCount2,
 T2 pR[9], T2 pT[3]);
```

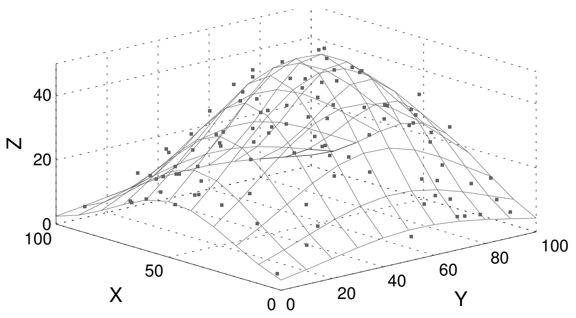


Figure 78: An example of the 2D FitGauss function.

Description	ICP algorithm for point cloud registration Apply ICP algorithm to find the rotation matrix and the translation vector that minimize the RMS error between two point clouds. The point cloud with fewer points is mapped to the coordinate of the one with more points.
ppt1	point array 1
nCount	size of ppt1
ppt2	point array 2
nCount	size of ppt1 and ppt2
pR	rotation matrix if found
pT	translation vector if found
Return	0: registration failed; 1: ppt1 is mapped to ppt2's coordinate ppt2 = R*ppt1 + T; 2: ppt2 is mapped to ppt1's coordinate ppt1 = R*ppt2 + T.

4.42 PtcCalcRT

```
template <class T1, class T2> bool PtcCalcRT
(const CPt3D<T1> *ppt1, const CPt3D<T1> *ppt2, int nCount,
 T2 pR[9], T2 pT[3]);
```

Description	Calculate rotation matrix and translation vector Apply the Kabsch algorithm to find the rotation matrix and the translation vector that minimize the RMS error between two point clouds. The results are used to transform ppt1 to ppt2's coordinate: ppt2 = R * ppt1 + T.
ppt1	point array 1
ppt2	point array 2
nCount	size of ppt1 and ppt2
pR	rotation matrix if found
pT	translation vector if found
Return	true: succeeded; false: failed if nCount < 3.

4.43 PtcFltMean

```
template <class T> void PtcFltMean
(const CPt3D<T> *ppt1, int nCount, int nNeighbour, CPt3D<T> *ppt2);
```

Description	Point cloud mean filtering Each point in the point cloud is replaced by the mean position of its neighbourhood.
pvpt1	a point array
nCount	size of ppt1 and ppt2
nNeighbour	number of closest neighbours to calculate for mean position
ppt2	output smoothed points. Its size must be nCount.

4.44 PtcFltDist

```
template <class T> void PtcFltDist
(const CPt3D<T> *ppt1, int nCount, int nNeighbour,
 double dDistNei, double dDistSurf, vector<int> *pvIdx);
```

Description	Point cloud distance filtering If the distance from a point to its farthest neighbour is larger than dDistNei, its index is added to pvIdx. Similarly, if the distance from a point to its local surface plane is larger than dDistSurf, its index is added to pvIdx.
pvpt	a point array
nCount	size of ppt
nNeighbour	number of closest neighbours to calculate for mean position
dDistNei	threshold of distance to the farthest neighbour
dDistSurf	threshold of distance to the surface plane
	Input a non-positive value to ignore a threshold.
pvIdx	index of the points above the thresholds

4.45 PtcMerge

```
template <class T1, class T2> void PtcMerge
(vector<CPt3D<T1> > *pvpt1, vector<T2> *pvWei1,
 vector<CPt3D<T1> > *pvpt2, vector<T2> *pvWei2, int nNear);
```

Description	Merge two point clouds Merge two point clouds by removing overlapping regions. On input, pvpt1 and pvpt2 are the source point clouds. On output, some overlapping points are removed from pvpt1 and some for pvpt2. For any two overlapping points, the one with larger weight is kept and the other is removed.
pvpt1	first point cloud/array
pvWei1	weight of each point in pvpt1
pvpt2	second point cloud/array
pvWei2	weight of each point in pvpt2
nNear	number of closest neighbours to define vicinity

4.46 PtcToSurf

```
template <class T> int PtcToSurf
(const CPt3D<T> *ppt, int nCount, int nNeighbour, float fFill,
 vector<CTri3D<float> > *pvTri);
```

Description	Generate surface from a point cloud Reference PhD thesis “Surface reconstruction from unorganized points”, Hugues Hoppe.
ppt	a point array
nCount	size of ppt
nNeighbour	number of closest neighbours. It is used to estimate each point’s local span. The larger is the value, the smoother is the surface but the longer is the processing time.
fFill	hole-filling multipler to each point’s local span estimated based on nNeighbour. It can be any value larger than 0. If larger than 1, it increases the hole-filling capability.
pvTri	generated surface triangles
Return	the number of surface triangles

```
template <class T> int PtcToSurf
(const CPt3D<T> *ppt, const CPt3D<T> *pptNV, int nCount,
 int nNeighbour, float fFill, vector<CTri3D<float> > *pvTri);
```

Description	Generate surface from a point cloud, in which each point has an associated unit normal vector. Reference “Surface reconstruction from unorganized points”, Hugues Hoppe.
pptNV	unit normal vector of each point in ppt
	For other parameters, see PtcToSurf(...)
Return	the number of surface triangles

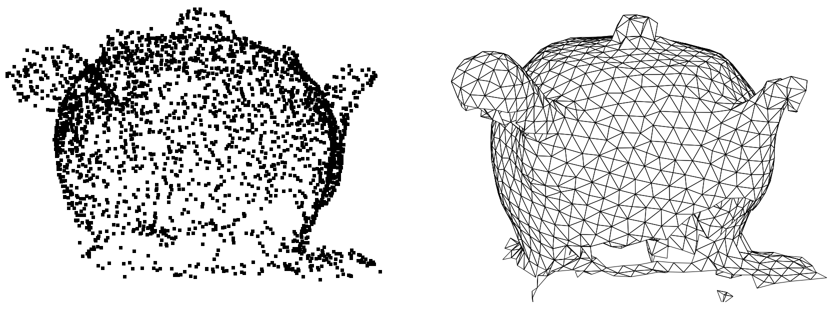

(a) (b)

Figure 79: An example of the PtcToSurf function. (a) Input point cloud. (b) Reconstructed surface.

4.47 ReadSTL

```
bool ReadSTL(const char *pcFileName, vector<float> *pvCoord);
```

Description	Read ASCII or binary STL file
pcFileName	name of the STL file
pvCoord	3D coordinate of all the triangles. For example, pvCoord[0], [1], [2] are x, y and z coordinates of the 1st vertex of the 1st triangle; [9*i], [9*i+1], [9*i+2] are x, y and z coordinates of the 1st vertex of the i-th triangle.
Return	true: succeeded; false: failed.

4.48 SaveBinarySTL

```
bool SaveBinarySTL(const CTri3D<float> *pTri, int nCount, const char *pcFileName);
```

Description	Save binary STL file
pvTri	an array of triangles to save in the STL file
nCount	size of pvTri
pcFileName	name of the STL file
Return	true: succeeded; false: failed.

5 3D image processing functions

Some parameters are common to most functions. They are listed in Table 3.

Type	Name	Description
T*	pImg	Pointer to a continuous memory space which must be equal to or larger than w*h*d*sizeof(T) bytes. The z-th frame, y-th row, x-th column element can be retrieved by *(pImg+(z*h+y)*w+x) or pImg[(z*h+y)*w+x].
T*	pSrc	Pointer to the source image
T*	pDst	Pointer to the destination image
int	w, h, d	Image width, height and depth
CCube<int>&	cbROI	Region of interest. Only image data within the ROI will be used or modified.
Pred	pred	a functional object. If pred(pSrc[(z*h+y)*w+x]) is true, (x,y,z) is an object point; otherwise, a background point.

Table 3: Common parameters of 3D image processing functions.

5.1 ImgAssign3D

```
template <class T> void ImgAssign3D
(T *pImg, int w, int h, int d, const CCube<int> &cbROI, T value);
```

Description	Assign image data in an ROI to the input value
Parameters	See Table 3 for common parameters
value	value to be assigned to the image data

5.2 ImgAssignBorder3D

```
template <class T> void ImgAssignBorder3D
(T *pImg, int w, int h, int d, const CCube<int> &cbROI, T value,
 int nLeft, int nTop, int nFront, int nRight, int nBottom, int nBack);
```

Description	Assign image data in an ROI to the input value nLeft, nTop, nFront, nRight, nBottom, nBack: distance toward the center on the left, top, front right, bottom and back border respectively. Pixels within the distance are assigned to the input value.
Parameters	See Table 3 for common parameters
value	value to be assigned to the image data

5.3 ImgCopy3D

```
template <class T1, class T2> void ImgCopy3D
(const T1 *pSrc, int w1, int h1, int d1, const CCube<int> &cbROI1,
       T2 *pDst, int w2, int h2, int d2, const CCube<int> &cbROI2);
```

Description	Copy image data from source image ROI1 to destination image ROI2 Width, height and depth of ROI1 and ROI2 must be the same.
Parameters	See Table 3 for common parameters

5.4 ImgCopyXY_2D

```
template <class T1, class T2> void ImgCopyXY_2D
(const T1 *pSrc, int w1, int h1, int d1, const CRect<int> &rcROI1, int z,
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2);
```

Description	Copy ROI1 in a X-Y slice of a 3D image to ROI2 in a 2D image
Parameters	See Tables 2 and 3 for common parameters
z	z-direction index of the X-Y slice in the 3D image

5.5 ImgCopyXZ_2D

```
template <class T1, class T2> void ImgCopyXZ_2D
(const T1 *pSrc, int w1, int h1, int d1, const CRect<int> &rcROI1, int y,
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2);
```

Description	Copy ROI1 in a X-Z slice of a 3D image to ROI2 in a 2D image
Parameters	See Table 2 and 3 for common parameters
y	y-direction index of the X-Z slice in the 3D image

5.6 ImgCopyYZ_2D

```
template <class T1, class T2> void ImgCopyYZ_2D
(const T1 *pSrc, int w1, int h1, int d1, const CRect<int> &rcROI1, int x,
       T2 *pDst, int w2, int h2, const CRect<int> &rcROI2);
```

Description	Copy ROI1 in a Y-Z slice of a 3D image to ROI2 in a 2D image
Parameters	See Table 2 and 3 for common parameters
x	x-direction index of the Y-Z slice in the 3D image

5.7 ImgCopy2D_XY

```
template <class T1, class T2> void ImgCopy2D_XY
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pDst, int w2, int h2, int d2, const CRect<int> &rcROI2, int z);
```

Description	Copy ROI1 in a 2D image to ROI2 in a X-Y slice of a 3D image
Parameters	See Tables 2 and 3 for common parameters
z	z-direction index of the X-Y slice in the 3D image

5.8 ImgCopy2D_XZ

```
template <class T1, class T2> void ImgCopy2D_XZ
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pDst, int w2, int h2, int d2, const CRect<int> &rcROI2, int y);
```

Description	Copy ROI1 in a 2D image to ROI2 in a X-Z slice of a 3D image
Parameters	See Tables 2 and 3 for common parameters
y	y-direction index of the X-Z slice in the 3D image

5.9 ImgCopy2D_YZ

```
template <class T1, class T2> void ImgCopy2D_YZ
(const T1 *pSrc, int w1, int h1, const CRect<int> &rcROI1,
       T2 *pDst, int w2, int h2, int d2, const CRect<int> &rcROI2, int x);
```

Description	Copy ROI1 in a 2D image to ROI2 in a Y-Z slice of a 3D image
Parameters	See Tables 2 and 3 for common parameters
x	x-direction index of the Y-Z slice in the 3D image

5.10 ImgLinear3D

```
template <class T1, class T2> void ImgLinear3D
(const T1 *pSrc, int w1, int h1, int d1, const CCube<int> &cbROI1,
       T2 *pDst, int w2, int h2, int d2, const CCube<int> &cbROI2,
 T1 fromMin, T1 fromMax, T2 toMin, T2 toMax);
```

Description	Linear translate Src data in the range [fromMin, fromMax] to Dst data to the range [toMin, toMax]. Linear translation parameters a and b is determined by:

continued on next page

	a * fromMin + b = toMin;
	a * fromMax + b = toMax;
	Data points in Src that are < fromMin or > fromMax will be translate to toMin and toMax. pSrc and pDst may point to the same image buffer, if ROI1 and ROI2 are the same.

Parameters	See Table 3 for common parameters
fromMin	source minimum bound
fromMax	source maximum bound
toMin	destination minimum bound
toMax	destination maximum bound

5.11 ImgMin3D

```
template <class T> T ImgMin3D
(const T *pImg, int w, int h, int d, const CCube<int> &cbROI);
```

Description	Get the minimum value of an ROI
Parameters	See Table 3 for common parameters
Return	the minimum value of an ROI

5.12 ImgMax3D

```
template <class T> T ImgMax3D
(const T *pImg, int w, int h, int d, const CCube<int> &cbROI);
```

Description	Get the maximum value of an ROI
Parameters	See Table 3 for common parameters
Return	the maximum value of an ROI

5.13 ImgMinMax3D

```
template <class T> void ImgMinMax3D
(const T *pImg, int w, int h, int d, const CCube<int> &cbROI,
 T *pMin, T *pMax);
```

Description	Get the minimum and maximum value of an ROI
Parameters	See Table 3 for common parameters
pMin	return the minimum value of an ROI
pMax	return the maximum value of an ROI

5.14 ImgMean3D

```
template <class T> T ImgMean3D
(const T *pImg, int w, int h, int d, const CCube<int> &cbROI);
```

Description	Get the mean value of an ROI
Parameters	See Table 3 for common parameters
Return	the mean value of an ROI

5.15 ImgVariance3D

```
template <class T> double ImgVariance3D
(const T *pImg, int w, int h, int d, const CCube<int> &cbROI);
```

Description	Get the variance of an ROI
Parameters	See Table 3 for common parameters
Return	the variance of an ROI

5.16 ImgFltMean3D

```
template <class T> void ImgFltMean3D
(const T *pSrc, int w1, int h1, int d1, const CCube<int> &cbROI1,
       T *pDst, int w2, int h2, int d2, const CCube<int> &cbROI2,
 int nKerW, int nKerH, int nKerD);
```

Description	Mean filter, kernel certer equal to kernel mean Half-filter-length data at the ROI boundary are filtered with reduced sized kernel.
Parameters	See Table 3 for common parameters
nKerW	filter kernel width
nKerH	filter kernel height
nKerD	filter kernel depth
Example	nKerW*nKerH*nKerD = 3*3*3 or 7*5*3

5.17 ImgFltVariance3D

```
template <class T1, class T2> void ImgFltVariance3D
(const T1 *pSrc, int w1, int h1, int d1, const CCube<int> &cbROI1,
       T2 *pDst, int w2, int h2, int d2, const CCube<int> &cbROI2,
 int nKerW, int nKerH, int nKerD);
```

Description	Variance filter, kernel certer equal to kernel variance Half-filter-length data at the ROI boundary are filtered with reduced sized kernel.
Parameters	See Table 2 for common parameters
nKerW	filter kernel width
nKerH	filter kernel height
nKerD	filter kernel depth
Example	nKerW*nKerH = 3*3 or 7*5

5.18 ImgConvZ

```
template <class T1, class T2> void ImgConvZ
(T1 *pImg, int w, int h, int d, const CCube<int> &cbROI,
 const T2 *pMask, int nLen);
```

Description	1D convolution in z direction Convolution is applied on each trace (z direction). Half-filter-length data at the ROI boundary are unchanged. The mask data array size must satisfy nLen+(nLen-1)/2 $\leq$ ROI.Depth(); otherwise nothing is done.
Parameters	See Table 3 for common parameters
pMask	pointer to the convolution mask data array
nLen	mask array size

5.19 ImgResize3D

```
template <class T1, class T2> void ImgResize3D
(const T1 *pSrc, int w1, int h1, int d1, const CCube<int> &cbROI1,
       T2 *pDst, int w2, int h2, int d2, const CCube<int> &cbROI2,
 int nAlgo);
```

Description	Resize ROI1 of Src and copy to ROI2 of Dst If linear (nAlgo: 1) or spline (nAlgo: 2–4) is used, minification is always a downsampling process based on averaging. Magnification is done by the specified method. If nearest neighbour (nAlgo: 0) is used, both minification and magnification are based on nearest neighbour.

continued on next page

Parameters	See Table 3 for common parameters
nAlgo	interpolation algorithm 0: nearest neighbour interpolation, 1: linear interpolation, 2: cubic B-spline interpolation, 3: Catmull-Rom spline interpolation, 4: natural cubic spline interpolation.

5.20 ImgRotate3D

```
template <class T1, class T2> void ImgRotate3D
(const T1 *pSrc, int w1, int h1, int d1, const CCube<int> &cbROI1,
       T2 *pDst, int w2, int h2, int d2, const CCube<int> &cbROI2,
 int nAxis, int nAngle);
```

Description	Rotate ROI1 of Src and copy to ROI2 of Dst The size of ROI1 and ROI2 must satisfy certain relationship based on the angle of rotation.
Parameters	See Table 3 for common parameters
nAxis	with respect to which axis to rotate. 0: X axis (points to the right), 1: Y axis (points to the bottom), 2: Z axis (points outside the screen).
nAngle	anti-clock wise angle (right-hand rule) of rotation in degree. Only multiples of 90-degree are accepted.

5.21 ImgFlip3D

```
template <class T> void ImgFlip3D
(T *pImg, int w, int h, int d, const CCube<int> &cbROI,
 int nDirection);
```

Description	Flip image data in an ROI
Parameters	See Table 3 for common parameters
nDirection	0: left-right flipped, 1: top-bottom flipped, 2: front-back flipped.

5.22 ImgMatch3D

```
template <class T1, class T2> void ImgMatch3D
(const T1 *pMother, int w1, int h1, int d1, const CCube<int> &cbROI1,
 const T1 *pChild,  int w2, int h2, int d2, const CCube<int> &cbROI2,
 int nDSX, int nDSY, int nDSZ, CPt3D<int> *pptMatch, T2 *pCorr);
```

Description	Search for ROI2 of pChild in ROI1 of pMother
	ROI1.Width() must be >= ROI2.Width()
	ROI1.Height() must be >= ROI2.Height()
	ROI1.Depth() must be >= ROI2.Depth()

1	Initially, ROI2 is matched to windows in ROI1 separated by nDSX, nDSY and nDSZ in the x, y and z directions. After the best-match window is found; nDSX, nDSY and nDSZ are halved.
2	Then ROI2 is matched to 27 windows, a 3*3*3 grid centered on the best-match window. The windows are separated by nDSX, nDSY and nDSZ. After a new best-match window is found, nDSX, nDSY and nDSZ are further halved.
3	Go back to 2 until nDSX, nDSY and nDSZ equal to 1 pixel.
Parameters	See Table 3 for common parameters
nDSX	initial downsampling gap in x direction
nDSY	initial downsampling gap in y direction
nDSZ	initial downsampling gap in z direction
	The downsampling gaps are used in shifting ROI1 across ROI2 and in accessing the pixels in ROI1 and ROI2,

continued on next page

pptMatch	the best-match point in ROI1 coordinate (left-top-front corner of the matched cube)
pCorr	normalized correlation coefficient, in [-1,1]. Its data type should be either float or double.

5.23 ImgMotion3D

```
template <class T1, class T2> bool ImgMotion3D
(const T1 *pImg1, int w1, int h1, int d1, const CCube<int> &cbROI1,
 const T1 *pImg2, int w2, int h2, int d2, const CCube<int> &cbROI2,
 int nCoarW,   int nCoarH,   int nCoarD,
 int nCoarX,   int nCoarY,   int nCoarZ,
 int nCoarDSX, int nCoarDSY, int nCoarDSZ,
 int nFineW,   int nFineH,   int nFineD,
 int nFineX,   int nFineY,   int nFineZ,
 int nFineDSX, int nFineDSY, int nFineDSZ,
 int nWinX,    int nWinY,    int nWinZ,
 vector<CPt3D<int> > *pvptCenter,
 vector<CPt3D<int> > *pvptDisp,
 vector<T2>          *pvCorr);
```

Description	Pixel flow or motion estimation Estimate the pixel motion from ROI2 in pImg2 to ROI1 in pImg1. ROI1 and ROI2 must be the same size.
1	There are two levels of search: a coarse search based on a few coarse windows and a quality-guided fine search on all fine windows. Results of the coarse search are used as the initial guess for the fine search.
2	A large coarse search window and search distance may be used to obtain robust initial estimates, and a small fine search window and search distance to obtain a high resolution. The downsampling gap at each level is set independently. See ImgMatch3D(...) for details of the downsampling gap.
3	Results are stored in pvptCenter, pvptDisp, pvCorr, which are arrays of the same size: nWinX * nWinY * nWinZ.
Parameters	See Table 3 for common parameters
nCoarW	coarse search window width
nCoarH	coarse search window height
nCoarD	coarse search window depth
nCoarX	coarse search distance in x direction
nCoarY	coarse search distance in y direction
nCoarZ	coarse search distance in z direction
nCoarDSX	coarse search downsampling gap in x direction
nCoarDSY	coarse search downsampling gap in y direction
nCoarDSZ	coarse search downsampling gap in z direction
nFineW	fine search window width
nFineH	fine search window height
nFineD	fine search window depth
nFineX	fine search distance in x direction
nFineY	fine search distance in y direction
nFineZ	fine search distance in z direction
nFineDSX	fine search downsampling gap in x direction
nFineDSY	fine search downsampling gap in y direction
nFineDSZ	fine search downsampling gap in z direction
nWinX	number of fine search windows in x direction
nWinY	number of fine search windows in y direction
nWinZ	number of fine search windows in z direction
pvptCenter	center of fine search windows in pImg2

continued on next page

pvptDisp	displacement of fine windows from pImg2 to pImg1. This includes the global shift from ROI2 to ROI1
pvCorr	correlation coefficient of each fine window. Its data type should be either float or double.
Return	true: succeeded; false: failed.

5.24 ImgLabeling3D

```
template <class T1, class T2, class Pred> int ImgLabeling3D
(const T1 *pSrc, int w1, int h1, int d1, const CCube<int> &cbROI1,
       T2 *pDst, int w2, int h2, int d2, const CCube<int> &cbROI2,
 Pred pred, int nConnectivity, int nMinVol = 0, int nMaxVol = 0,
 vector<CCube<int> > *pvCube = 0, vector<int> *pvVol = 0);
```

Description	Connectivity labeling algorithm
Parameters	See Table 3 for common parameters
nConnectivity	Euclidean connectivity of an object. 0: 6-neighbour labeling algorithm, 1: 26-neighbour labeling algorithm, >1: pixels within this distance are one object.
nMinVol	Object fewer than nMinVol pixels are discarded.
nMaxVol	Object larger than nMaxVol pixels are discarded.
pvCube	bounding cube of each object (input 0 to ignore) It is with respect to ROI2. To obtain a bounding cube with respect to ROI1, each cube should be offset cb1.LTF()-cb2.LTF().
pvVol	number of points of each object (input 0 to ignore)
Return	the number of objects found
Example	The 1st object can be found by checking if(pDst[(z*h+y)*w+x]==1). The 1st object's bounding cube is (*pvCube)[0]. The 3rd object's number of points is (*pvVol)[2].

5.25 ImgFilling3D

```
template <class T, class Pred> void ImgFilling3D
(T *pImg, int w, int h, int d, const CCube<int> &cbROI,
 T value, Pred pred, CPt3D<int> ptSeed, int nConnectivity);
```

Description	Fill a region bounded by a predicate condition
Parameters	See Table 3 for common parameters
value	value to fill with
pred	if pred(pImg[(z*h+y)*w+x],value) is true, (x,y,z) is a boundary point. Only three predicates are valid: equal_to, greater_equal, less_equal. Other predicates, such as greater or less, may cause dead loop and should not be used.
ptSeed	initial seed point for filling
nConnectivity	Euclidean connectivity of a filling region. 0: 6-neighbour filling algorithm, 1: 26-neighbour filling algorithm, >1: pixels within this distance are filled.

5.26 ImgBoundaryUnordered3D

```
template <class T, class Pred> int ImgBoundaryUnordered
(const T *pImg, int w, int h, int d, const CCube<int> &cbROI,
 Pred pred, vector<CPt3D<int> > *pvpt);
```

Description	Get unordered boundary points by checking 6-way neighbours. Holes are detected as boundary.
Parameters	See Table 3 for common parameters
pvpt	output the boundary points
Return	the number of boundary points

5.27 ImgDistTrans3D

```
template <class T, class Pred> void ImgDistTrans3D
(const T       *pSrc,  int w1, int h1, int d1, const CCube<int> &cbROI1,
 unsigned long *pnDst, int w2, int h2, int d2, const CCube<int> &cbROI2,
 double dZSpacing, Pred pred);
```

Description	Distance transform algorithm Object boundary pixels are set to 0. A nun-boundary pixel is set to the distance to its nearest object boundary pixel. A non-boundary object pixel gets a positive distance and a background pixel gets a negative distance. The generated distance value is 100 * the actual Euclidean distance. Reference “Euclidean Distance Mapping”, P. Danielsson.
Parameters	See Table 3 for common parameters
dZSpacing	frame spacing in z direction. The pixel spacing in the x and y directions is assumed to be 1.

```
template <class T1, class T2, class Pred> void ImgDistTrans3D
(const T1 *pSrc, int w1, int h1, int d1, const CCube<int> &cbROI1,
       T2 *pDst, int w2, int h2, int d2, const CCube<int> &cbROI2,
 Pred pred);
```

Description	Signed distance transform Distance code 22-31-38. (Pixel width 22, diagonal: 31, double diagonal: 38.) Object pixels are assigned positive distances and background pixels are assigned negative distances. For the sake of coding efficiency, the 1-pixel border in x and y directions do not have accurate distance value. Reference “Distance transformations in arbitrary dimensions”, Gunilla Borgefors.
Parameters	See Table 3 for common parameters
pDst	should be a signed data type (e.g. char, int)

5.28 ImgWatershed3D

```
template <class T1, class T2> int ImgWatershed3D
(const T1 *pSrc, int w1, int h1, int d1, const CCube<int> &cbROI1,
       T2 *pDst, int w2, int h2, int d2, const CCube<int> &cbROI2,
 int nDist, int nMinPts);
```

Description	Watershed algorithm for segmentation This function assumes the input pSrc a distance map, for example, obtained from ImgDistTrans3D(...). Negative pixels are background; otherwise, object. It outputs an object map similar to that produced by ImgLabeling3D(...). In the object map, connected regions are segmented.
1	Local maxima are located for each object.
2	The first-pass propagation estimates the number of points grown by each maxima within each object.
3	Maxima whose number of points are < nMinPts are discarded.
4	The second-pass propagation finishes the segmentation.
Parameters	See Table 3 for common parameters
nDist	the distance for merging local maxima. 0: 6-neighbour merging, 1: 26-neighbour merging, >1: local maxima within this distance are merged.
nMinPts	threshold of the number of points grown by a maximum.
	Both parameters are used to reduce over-segmentation.
Return	The number of segmented objects, if succeeded; otherwise -1.

5.29 ImgMaskLine3D

```
template <class T1, class T2> void ImgMaskLine3D
(T1 *pImg, int w, int h, int d, const CCube<int> &cbROI,
 CLine3D<T2> *pLine, T1 value);
```

Description	Set pixels on a line to a mask value
Parameters	See Table 3 for common parameters
pLine	line to be masked. It may be shortened so that it is completely within the image cube.
value	mask value

5.30 ImgMaskTriangle

```
template <class T1, class T2> void ImgMaskTriangle
(T1 *pImg, int w, int h, int d, const CCube<int> &cbROI,
 const CPt3D<T2> pptTri[3], T1 value);
```

Description	Set pixels on a triangle to a mask value
Parameters	See Table 3 for common parameters
pptTri	vertices of the triangle
value	mask value

5.31 ImgMaskMesh

```
template <class T1, class T2> void ImgMaskMesh
(T1 *pImg, int w, int h, int d, const CCube<int> &cbROI,
 const T2 *pCoord, int nCount, bool bFill);
```

Description	Mask a triangle mesh region Create a mask image for a triangle mesh region by filling. This approach is faster than checking if each pixel is in or out of the mesh. After processing, pixels outside the mesh are set to 0; those inside are set to 1 if filling is applied; those on the surface of the mesh are set to 2.
Parameters	See Table 3 for common parameters
pCoord	coordinates of the mesh (x1, y1, z1, ...)
nCount	size of pCoord
bFill	indicate if filling the interior of the mesh. If the mesh is not closed, the entire ROI will be filled.

5.32 ImgIsoSurface

```
template <class T1, class Pred> int IsoSurface
(const T *pImg, int w, int h, int d, const CCube<int> &cbROI,
 double dXSpacing, double dYSpacing, double dZSpacing,
 Pred pred, int nDS, int nFltSize, vector<CTri3D<float> > *pvTri);
```

Description	Generate isosurface from any 3D data Reference “Regularised marching tetrahedra: improved iso-surface extraction”, Graham Treece et al.
1	Resize the image, if needed, to make pixel spacing identical in x, y and z directions.
2	Apply distance transform.
3	Smooth the distance map, if nFltSize is $>$ 1.
4	Use body-centered cube layout (partitioned by tetrahedra) to generate isosurface.
Parameters	See Table 3 for common parameters
dXSpacing	pixel spacing in x direction
dYSpacing	pixel spacing in y direction
dZSpacing	q spacing in z direction
pred	if pred(pImg[(z*h+y)*w+x]) is true, point (x,y,z) is an object point; otherwise background.
nDS	downsampling gap in x, y, z directions
nFltSize	filter kernel size (ignored if <= 1)
pvTri	output triangles of the isosurface
Return	the number of triangles of the isosurface

```
template <class T> int IsoSurface
(const T *pImg, int w, int h, int d, const CCube<int> &cbROI,
 vector<CTri3D<float> > *pvTri);
```

Description	Generate isosurface from a distance map Surfaces at the two-pixel boundary are ignored. If the distance map is obtained from a 3D image or a point cloud, it can be expanded by two pixels at the boundaries to include the surfaces of the original data. A maximum value of the distance data type is used to signal pixels that should be ignored. For example, if the data type is char, pixels equal to 127 are ignored in processing.
pImg	should be a signed distance map
pvTri	output triangles of the isosurface
Return	the number of triangles of the isosurface

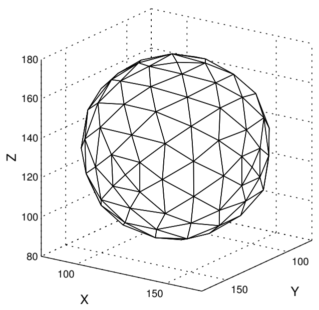


Figure 80: An example of the ImgIsoSurface function. Isosurface of a sphere.

5.33 ImgReadRaw3D

```
template <class T> bool ImgReadRaw3D
(T *pImg, int w, int h, int d, const CCube<int> &cbROI,
 const char *pcFileName, int nOffset);
```

Description	Read image data from a binary file The raw image data must match the data type T. The size of the data to be read is ROI.Volume(). If the file cannot be opened, pImg is unchanged.
Parameters	See Table 3 for common parameters
pcFileName	file name
nOffset	offset in bytes from the beginning of the file, where raw image data start to be read.
Return	true: succeeded; otherwise, false.

5.34 ImgSaveRaw3D

```
template <class T> bool ImgSaveRaw3D
(const T *pImg, int w, int h, int d, const CCube<int> &cbROI,
 const char *pcFileName, bool bAppend = false);
```

Description	Save raw image data as a binary file
Parameters	See Table 3 for common parameters
pcFileName	file name
bAppend	whether append to the end of the file or create a new file. If 'pcFileName' does not exist, a new file will be created.
Return	true: succeeded; otherwise, false.