**实验报告—智能剪刀算法**

小组成员：陈海龙 戴熹 杨琛璟 熊慧

1. 算法描述
2. 算法简介：

智能剪刀”( Intelligent Scissors) 是 Morten-son 和 Barrett 在 1995 年提出来的一种图像分割交互工具,它可用于 2D 图像分割 ,通过这个工具, 用户可以容易且精确地在图像中勾画出感兴趣的区域 ROI( Region Of Interest) 。

1. 算法原理

智能剪刀是一个新的，交互式的，用于图像分割和合成的工具。

智能剪刀可以通过鼠标的移动快速和精确地提取图像中的物体。当鼠标确定的位置接近一个物体的边缘的时候，一个live-wire边界捕捉并且包围了我们感兴趣的物体。live-wire是一种交互式分割方法，其基本思想是利用动态规划方法产生图像中给定两点间的最有路径，合理地构造代价函数和选择起始点和目标点，用以提取物体的边缘。Live-wire便捷检测将离散的动态规划问题规划为一个二维图像的搜索问题。动态规划提供了数学意义上最佳的便捷，同时也极大的减少了局部噪声和其他干扰结构的影响。

该算法选择的边界不是邻接边中的最强壮的边，而是与现在正在被跟踪的边的特定类型相符合的边，这一过程我们成为on-the-fly training，增强了算法的可靠性和智能剪刀工具的健壮性。通过智能剪刀提取出来的物体可以被放大和缩小，旋转，以及利用live-wire掩模可空间频率等值性组合成新的图像。空间频率等值是利用巴特沃斯低通滤波器实现的。

智能剪刀提供了一个用于物体提取和图像合成的精确并且高效的交互性工具，它不仅可以用于灰度图像，同时也可适用于任意复杂度的彩色图像，并且基于这个工作还有很多扩展应用。

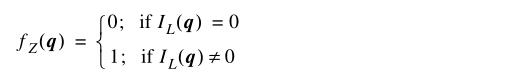
该算法分为以下几个部分：

1. **图像区域耗费值**
2. 图像区域耗费值(Image Local Cost) 是图像边界提取的依据。
3. Morten-son 和 Barrett 先后在1995年发表论文和 1998年发表论文。在两次的论文中，对于ILC 的取值有点不一样。
4. 现在一般而言，ILC 的取值一共由六项图像特征值加权和，分别是：

* 拉普拉斯交零点值(Laplacian zero-crossing)
* 梯度量值(Gradient magnitude)
* 梯度方向值(Gradient direction)
* 边界像素灰度值(Edge pixel value)
* 内部像素灰度值(Inside pixel value)
* 外部像素灰度值(Outside pixel value)

1. 这里写图片描述设l(p,q)表示ILC 的值，p 表示像素，q表示p 的相邻像素。（p、q 是一维的矢量，表示像素在图像中的坐标）
2. fZ - 拉普拉斯交叉零点

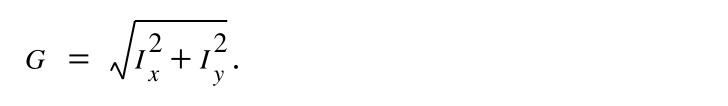
fZ的计算如下：



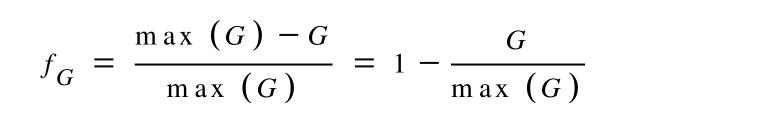
其中 IL 表示的是图像的拉普拉斯变换值。

1. fG - 梯度值

令 IX 和 IY 表示像素在 x 和 y 方向的梯度，则梯度 G 的计算如下：



为使高梯度产生低能量，fG的计算定义如下：

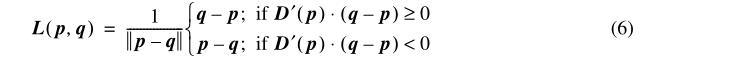


1. fD - 梯度方向值

假设 D(p) 表示的是像素点 p 的梯度方向，而 D’(p) 表示的与梯度方向垂直的向量（顺时针旋转90° 得来），那么 D(p) 和 D’(p) 的计算如下：

**D(p) = [ IX(p), IY(p) ]**   
**D’(p) = [ IY(p), -IX(p) ]**

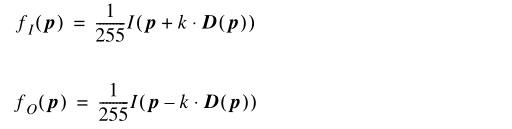
fD的计算：这里写图片描述

L 的计算也变为：****

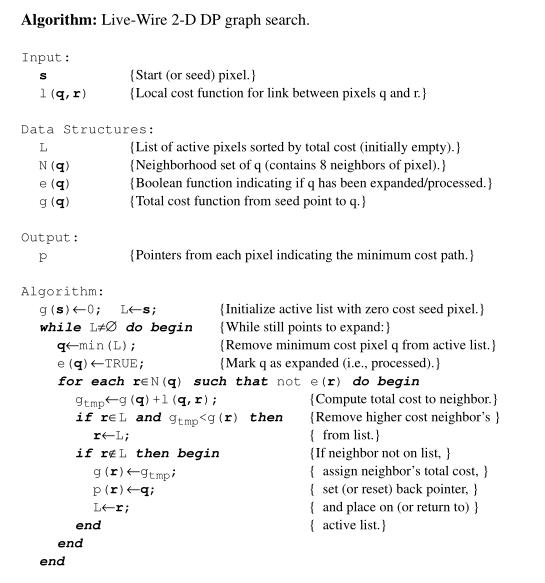
1. fP-边界像素值

这里写图片描述  
\*I(p) 表示该坐标的像素值

1. fI 和 fO

内部像素值和外部像素值分别是在像素点的梯度方向和反方向的偏移像素值。   


1. **最短路径生成算法**



1. 实现概述

整个算法的核心是计算ILC 值和路径的计算。认真分析ILC 值得计算，发现有些图像特征值的计算是不依赖于p 、 q 两点的关系的，那么对于这个特征值，有必要进行预处理。

* 1. 计算fZ

首先对整张图像预处理计算IL， 然后用来计算fZ的值。由于在后续的计算过程中不需要重复使用到IL，所以可以直接在同一个内存空间上先后完成IL和fZ的计算。

* 1. 计算fG和D’(p)

fG和D’(p) 都需要先计算出IX 和 IY，所以这两个值可以同时完成计算。

但是fG 并不完全依赖于q 点，所以在计算ILC 值还要判断p, q 点的位置关系。

* 1. 计算fP、fI 和 fO

fP 就等于p 点的像素值，直接取值就好。

fI 和 fO 的计算都很简单，在计算ILC 的时候，实时计算就行。实现版本如下，采用最近邻像素的方式：

* 1. 计算fD

在前面的计算中，D’(p) 已经预处理了，所以这里直接计算fD的值就好。

* 1. 计算ILC

上面的计算已经把各个图像的特征值都求出来了，这里直接进行加权求和就可以。

但是加权fG的时候，需要注意，还要对照p、 q 关系决定是不是需要除以根号2.

* 1. 计算路径

至此就可以循环计算路径了

1. 关键代码

注：本次实验参考了github上一个Intelligent Scissor项目，其地址如下

<https://github.com/frankdj412/Intelligent_Scissor>

1. 打开图片（Document.cpp）

BOOL CDoc::OnOpenDocument(LPCTSTR lpszPathName)

{

if (!CDocument::OnOpenDocument(lpszPathName))

return FALSE;

ReleaseAndInitialize();

if (lpszPathName != NULL)

{

// 将lpszPathName 转换成 std::string类型 文件名

#ifdef UNICODE

// 中文路径时会存在问题

size\_t i = 0;

char\* nstring = new char[(wcslen(lpszPathName)+1)\*2];

wcstombs\_s(&i, nstring, (wcslen(lpszPathName)+1)\*2, lpszPathName, (wcslen(lpszPathName)+1)\*2);

std::string filename(nstring);

delete[] nstring;

#else

std::string filename = std::string(lpszPathName);

#endif

// 加载原始的图片文件

o\_image = imread(filename, -1);

// 复制图片信息

o\_image.copyTo(image);

image.copyTo(tem1);

// 如果图片信息为空，则打开图片失败

if (!image.data)

{

MessageBox(NULL, lpszPathName, \_T("Fail to open the image."), MB\_OK);

return FALSE;

}

// 将放大的image存储在l\_img

Mat l\_img(image.rows+2, image.cols+2, CV\_8UC3);

larger(image, l\_img);

// 绘制Cost graph

nodes\_row = image.rows;

nodes\_col = image.cols;

nodes = new Node\*[image.rows];

for(int i = 0; i < image.rows; i++){

nodes[i] = new Node [image.cols];

}

cal\_local\_cost(nodes, l\_img);

mask = Mat(image.rows, image.cols, CV\_8UC3, Scalar(100, 100, 100));

tem1.copyTo(tem2);

// 创建Cost graph和Pixel graph

create\_pixel\_and\_cost\_graph(o\_image, m\_cvPixelGraph, m\_cvCostGraph, nodes);

m\_cvCostGraph.copyTo(m\_cvTreeMask);

m\_bInitial = true;

m\_bReload = true;

}

return TRUE;

}

1. 画图函数(View.cpp)

void CView::OnDraw(CDC\* pDC)

{

// 读取文件

CDoc\* pDoc = GetDocument();

ASSERT\_VALID(pDoc);

if (!pDoc)

return;

// 使重载按钮失效

pDoc->m\_bReload = false;

//判断是哪种工作模式

if (isWorkMode()) {

switch (m\_iWorkMode) {

case IMAGE\_ONLY:// 仅为图片，不能画点并选择区域

m\_cvImg = pDoc->image;

AdjustSize(m\_cvImg);

break;

case IMAGE\_CONTOUR: // 可以画点并选择区域，使用智能剪刀算法

if (pDoc->close) {// 如果路线闭合，则裁剪出闭合区域的图片

m\_cvImg = pDoc->mask;

} else {

m\_cvImg = pDoc->tem2;

}

break;

}

} else { // 其他模式，包括Pixel graph，Cost graph等

switch (m\_iDebugMode) {

case PIXEL\_NODE:

m\_cvImg = pDoc->m\_cvPixelGraph;

break;

case COST\_GRAPH:

m\_cvImg = pDoc->m\_cvCostGraph;

break;

case PATH\_TREE:

m\_cvImg = pDoc->m\_cvTreeMask;

break;

case MINI\_PATH:

m\_cvImg = pDoc->m\_cvMiniPath;

break;

}

}

// 将opencv-IplImage结构图像画到HDC上

HDC hDC = pDC->GetSafeHdc();

IplImage img = m\_cvImg;

CvvImage cimg;

cimg.CopyOf( &img );// 将IplImage复制为CvvImage

CRect rect;

rect.SetRect(0, 0, m\_cvImg.cols, m\_cvImg.rows);// 设置区域大小

cimg.DrawToHDC(hDC, &rect); // 在hdc上显示IpImage

ReleaseDC( pDC );// 释放hdc

}

1. 鼠标移动(View.cpp)

void CView::OnMouseMove(UINT nFlags, CPoint point)

{

CDoc\* pDoc = GetDocument();

ASSERT\_VALID(pDoc);

bool& close = pDoc->close;// 路径是否闭合

int& seed\_count = pDoc->seed\_count;// 标记点的个数

Mat& mask = pDoc->mask; // 被标记裁剪部分

Mat& tem2 = pDoc->tem2;

Vec2i& seed = pDoc->seed;

Vec2i& b\_seed = pDoc->b\_seed;

Vec2i& current = pDoc->current;

Vec2d& scale = pDoc->scale;// 缩放指数

bool is\_debug = (m\_iDebugMode == MINI\_PATH);

double x = point.x;

double y = point.y;

if (!is\_debug){ // 判断是否处于最短路径模式

current[0] = x / (scale[0] \* scale[1]);

current[1] = y / (scale[0] \* scale[1]);

}

else {

current[0] = x;

current[1] = y;

}

Mat temp;

if (is\_debug){

pDoc->m\_cvCostGraph.copyTo(temp);

}

else {

pDoc->tem1.copyTo(temp);

}

if (/\* 当工作模式处于可以选择点使用智能剪刀算法裁剪图片并且debug模式处于最短路径模式\*/

(m\_iWorkMode == IMAGE\_CONTOUR || m\_iDebugMode == MINI\_PATH) &&

/\* x、y坐标在正常范围，标记点大于0，路径还未闭合\*/

x >= 0 && x < temp.cols-1 &&

y >= 0 && y < temp.rows-1 &&

seed\_count > 0 && !m\_bClicked &&

pDoc->m\_bInitial && !pDoc->close){

// 画出从b\_seed到当前鼠标点击点的贴合边缘的路径

if (is\_debug){

Mat& m\_cvMiniPath = pDoc->m\_cvMiniPath;

pDoc->m\_cvCostGraph.copyTo(m\_cvMiniPath);

draw\_path(m\_cvMiniPath, temp, pDoc->cost\_map, current, false, 1, is\_debug);

}

else {

pDoc->tem1.copyTo(tem2);

draw\_path(tem2, mask, pDoc->cost\_map, current, false, scale[0] \* scale[1]);

}

double dis = sqrt((double)((current[0]-b\_seed[0])\*(current[0]-b\_seed[0]) +

(current[1]-b\_seed[1])\*(current[1]-b\_seed[1])));

//如果两个标记点的距离小于5，则视为覆盖

int minimum\_gap = 5;

if (dis < minimum\_gap) {

draw\_circle(tem2,current,0.5,false, scale);

}

Invalidate(TRUE);

}

CView::OnMouseMove(nFlags, point);

}

1. 点击鼠标标记点（View.cpp）

void CView::OnLButtonDown(UINT nFlags, CPoint point)

{

CDoc\* pDoc = GetDocument();

ASSERT\_VALID(pDoc);

// 图片是否被加载

m\_bClicked = true;

Mat& o\_image = pDoc->o\_image;

Mat& image = pDoc->image;

Mat& tem1 = pDoc->tem1;

Mat& tem2 = pDoc->tem2;

Mat& mask = pDoc->mask;

double x = point.x;

double y = point.y;

//当工作模式处于可以选择点使用智能剪刀算法裁剪图片并且debug模式处于最短路径模式

// x、y坐标在正常范围，标记点大于0，路径还未闭合

if (pDoc->m\_bInitial && m\_iWorkMode == IMAGE\_CONTOUR && !pDoc->close && // pDoc->seed\_count != 0 &&

(pDoc->seed\_count != 0 || (pDoc->seed\_count == 0 && (nFlags & MK\_CONTROL))) &&

x >= 0 && x < tem2.cols && y >= 0 && y < tem2.rows) {

int& seed\_count = pDoc->seed\_count;// 标记点的个数

bool& close = pDoc->close;// 路径是否闭合

Vec2i& seed = pDoc->seed;

Vec2i& b\_seed = pDoc->b\_seed;// 上一个标记点的坐标

Vec2i& current = pDoc->current;// 当前点击的点坐标

Vec2d& scale = pDoc->scale;// 缩放的指数

stack<node\*\*>& map\_stack = pDoc->map\_stack;

stack<Mat>& mask\_stack = pDoc->mask\_stack;

stack<Mat>& tem1\_stack = pDoc->tem1\_stack;

stack<Vec2d>& scale\_stack = pDoc->scale\_stack;

stack<vector<node\*>>& node\_list\_stack = pDoc->node\_list\_stack;

tem1.copyTo(tem2);

current[0] = x;

current[1] = y;

seed\_count++;

seed[0] = current[0] = x / (scale[0]\*scale[1]);

seed[1] = current[1] = y / (scale[0] \* scale[1]);

if (seed\_count == 1) {

b\_seed[0] = x / (scale[0] \* scale[1]);

b\_seed[1] = y / (scale[0] \* scale[1]);

}

// 判断路径是否闭合

if (!close) {

//不闭合

// 如果标记点个数大于1

if (seed\_count > 1 ) {

// 计算鼠标点击的点与最开始标记点的距离

double dis = sqrt((double)((seed[0]-b\_seed[0])\*(seed[0]-b\_seed[0]) +

(seed[1]-b\_seed[1])\*(seed[1]-b\_seed[1])));

//如果两个标记点的距离小于5，则视为覆盖

int minimum\_gap = 5;

if (dis < minimum\_gap) {

close = true;

}

}

}

//如果画完最新一个点以后，路径闭合

if (close) {

//画出路径和裁剪区域

draw\_path(tem1, mask, pDoc->cost\_map, b\_seed, true, scale[0] \* scale[1]);

draw\_circle(tem1, b\_seed, 0.5, false, scale);

//保存裁剪区域

save\_stack(mask,mask\_stack);

save\_stack(tem1,tem1\_stack);

save\_mask(mask, o\_image, scale);

save\_scale(scale, scale\_stack);

save\_node\_list\_stack(pDoc->node\_list, pDoc->node\_list\_stack);

}

// 画完新的点和路径，路径不闭合

else {

if(seed\_count > 1) {//标记点大于1个，画出标记点形成的路径

draw\_path(tem1, mask, pDoc->cost\_map, seed, true, scale[0] \* scale[1]);

pDoc->cost\_map = new\_tree(pDoc->cost\_map, image.rows, image.cols);

}

draw\_circle(tem1, seed, 0.5, true, scale);

save\_stack(mask, mask\_stack);

save\_stack(tem1, tem1\_stack);

save\_scale(scale, scale\_stack);

generate\_tree(pDoc->cost\_map, seed, image.rows, image.cols, pDoc->node\_list);

map\_stack.push(pDoc->cost\_map);

save\_node\_list\_stack(pDoc->node\_list, pDoc->node\_list\_stack);

}

Invalidate(TRUE);

}

m\_bClicked = false;

CView::OnLButtonDown(nFlags, point);

}

1. 放大图片(View.cpp)

void CView::ZoomIn() {

CDoc\* pDoc = GetDocument();

ASSERT\_VALID(pDoc);

if (isWorkMode()) {

Mat& tem1 = pDoc->tem1;

bigger(tem1);

tem1.copyTo(pDoc->tem2);

}

Vec2d& scale = pDoc->scale;

scale[1] = scale[1] \* 2.0;

m\_bZoomin = true;

}

1. 缩小图片(View.cpp)

void CView::ZoomOut() {

CDoc\* pDoc = GetDocument();

ASSERT\_VALID(pDoc);

if (isWorkMode()) {

Mat& tem1 = pDoc->tem1;

smaller(tem1);

tem1.copyTo(pDoc->tem2);

}

Vec2d& scale = pDoc->scale;

scale[1] = scale[1] / 2.0;

m\_bZoomout = true;

}

1. 撤销上一步操作(View.cpp)

void CView::Undo() {

CDoc\* pDoc = GetDocument();

ASSERT\_VALID(pDoc);

int& seed\_count = pDoc->seed\_count;// 标记点个数

// 如果标记点个数不为0

if(seed\_count != 0) {

bool& close = pDoc->close;// 是否闭合

Vec2i& seed = pDoc->seed;//标记点

Vec2i& b\_seed = pDoc->b\_seed;//上一个标记点

Vec2d& scale = pDoc->scale; //缩放指数

Mat& o\_image = pDoc->o\_image;//原始图像

Mat& image = pDoc->image;

Mat& tem1 = pDoc->tem1;

Mat& tem2 = pDoc->tem2;

Mat& mask = pDoc->mask;

stack<node\*\*>& map\_stack = pDoc->map\_stack;

stack<Mat>& mask\_stack = pDoc->mask\_stack;

stack<Mat>& tem1\_stack = pDoc->tem1\_stack;

stack<Vec2d>& scale\_stack = pDoc->scale\_stack;

//如果标记点个数为1，则停止标记的动作

if(seed\_count == 1) {

stop(tem1, o\_image, mask, scale[1], map\_stack, tem1\_stack, mask\_stack, pDoc->cost\_map, image.rows, image.cols, close, seed\_count, scale\_stack, pDoc->node\_list\_stack);

}

else {//如果标记点个数不为1，则撤回上一步标记

pDoc->cost\_map = undo(tem1, mask, map\_stack, mask\_stack, tem1\_stack, pDoc->scale\_stack, scale, pDoc->node\_list\_stack, pDoc->node\_list);

freopen("input.txt", "w", stdout);

cout << "The number of nodes: " << pDoc->node\_list.size() << endl;

seed\_count--;

}

tem1.copyTo(tem2);

m\_bUndo = true;

}

}

1. 清除画布(View.cpp)

void CView::ClearScreen() {

CDoc\* pDoc = GetDocument();

ASSERT\_VALID(pDoc);

bool& close = pDoc->close;// 是否闭合

int& seed\_count = pDoc->seed\_count;// 标记个数

Vec2i& seed = pDoc->seed;

Vec2i& b\_seed = pDoc->b\_seed;

Vec2d& scale = pDoc->scale;

Mat& o\_image = pDoc->o\_image;

Mat& image = pDoc->image;

Mat& tem1 = pDoc->tem1;

Mat& tem2 = pDoc->tem2;

Mat& mask = pDoc->mask;

stack<node\*\*>& map\_stack = pDoc->map\_stack;

stack<Mat>& mask\_stack = pDoc->mask\_stack;

stack<Mat>& tem1\_stack = pDoc->tem1\_stack;

stack<Vec2d>& scale\_stack = pDoc->scale\_stack;

// 停止标记的动作

stop(tem1, o\_image, mask, scale[1], map\_stack, tem1\_stack, mask\_stack, pDoc->cost\_map, image.rows, image.cols, close, seed\_count, scale\_stack, pDoc->node\_list\_stack);

// 缩小，放大的按钮变为无效；点击的按钮有效

m\_bZoomin = false;

m\_bZoomout = false;

m\_bClicked = true;

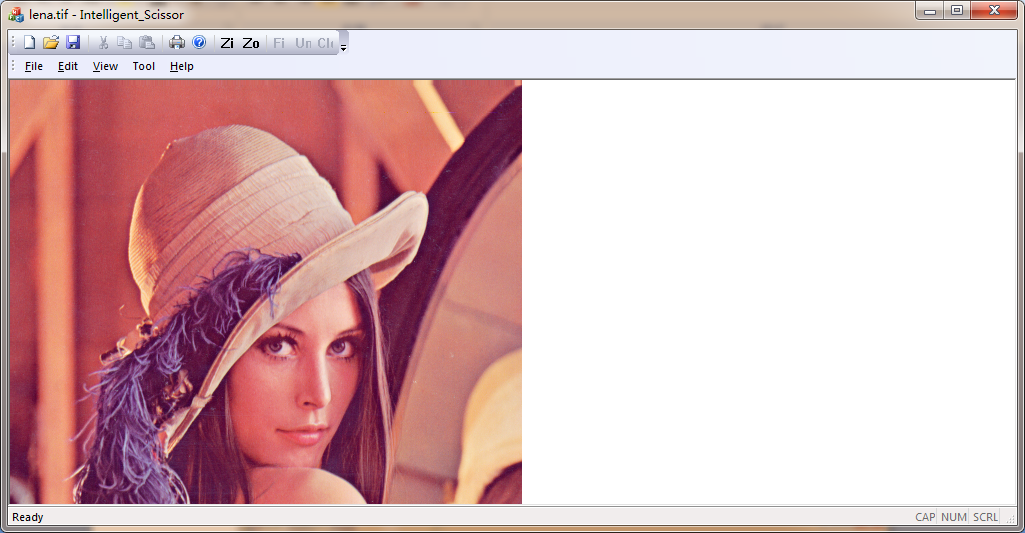
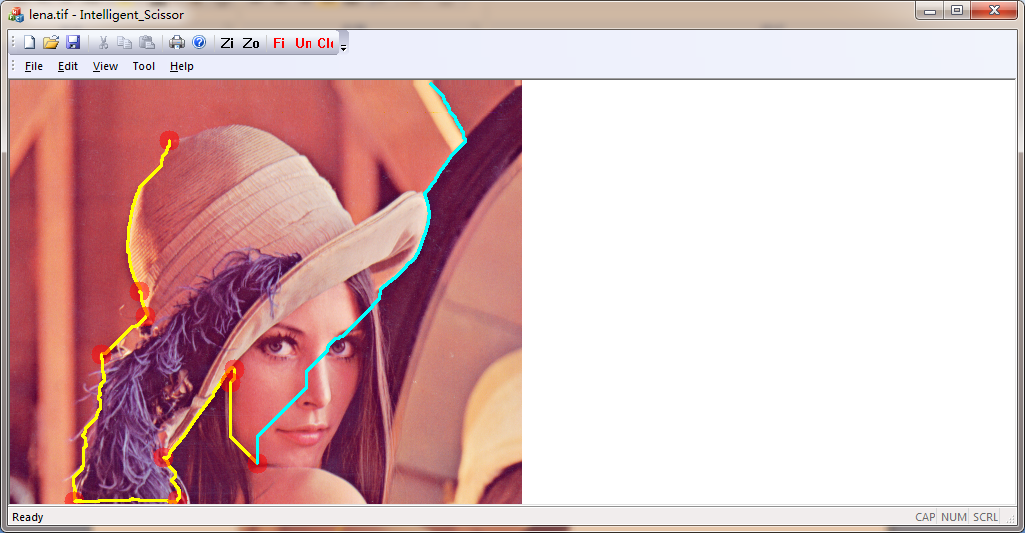
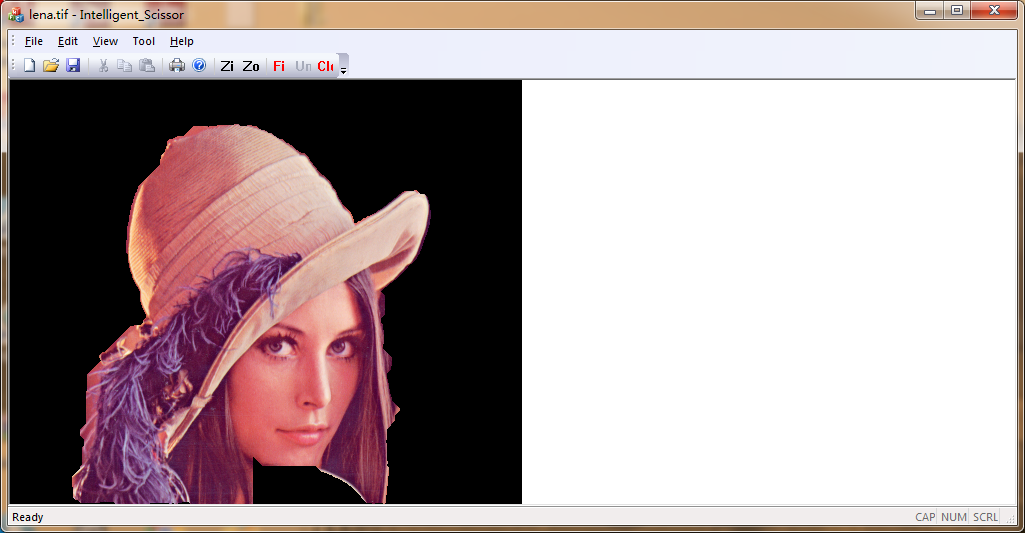
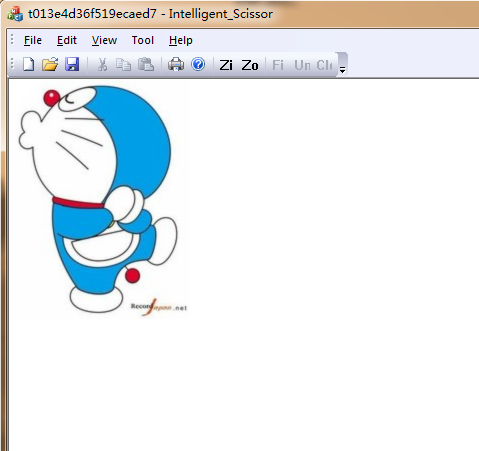
// 设置工作模式为可以设置标记点的模式

m\_iWorkMode = IMAGE\_CONTOUR;

SwitchToWorkMode();

tem1.copyTo(tem2);

}

1. 实验结果
2. 截图
3. 
4. 
5. 
7. 
8. 