[双缓冲法解决重绘和闪屏问题](https://www.cnblogs.com/renyuan/p/3474802.html)

https://www.cnblogs.com/renyuan/p/3474802.html

重绘导致原因：UpdateData、Invalidate、InvalidateRect和UpdateWindow函数。

1. UpdateData重绘控件函数

 UpdateData(TRUE)——刷新控件的值到对应的变量。(外部输入值交给内部变量)

 即：控件的值—>变量。

UpdateData(FALSE)　—— 拷贝变量值到控件显示。(变量的最终运算结果值交给外部输出显示)

 即：变量值—>控件显示。

2. Invalidate()

      该函数的作用是使整个窗口客户区无效。窗口的客户区无效意味着需要重绘，例如，如果一个被其它窗口遮住的窗口变成了前台窗口，那么原来被遮住的部分就是无效的，需要重绘。这时Windows会在应用程序的消息队列中放置WM\_PAINT消息。[MFC](http://baike.baidu.com/view/10918.htm" \t "_blank)为窗口类提供了WM\_PAINT的消息处理函数OnPaint，OnPaint负责重绘窗口。视图类有一些例外，在视图类的OnPaint函数中调用了OnDraw函数，实际的重绘工作由OnDraw来完成。参数bErase为TRUE时，重绘区域内的背景将被擦除，否则，背景将保持不变。

3. InvalidateRect

    用InvalidateRect函数只重绘部分区域，而且不重绘背景(第二个参数用FALSE)就可以解决大部分的屏闪问题。

    比如：CRect rect(10,47,10+120,47+70);  
    InvalidateRect(rect,FALSE);

 4. UpdateWindow函数  
UpdateWindow( )的作用是使窗口立即重绘。调用Invalidate等函数后窗口不会立即重绘，这是由于WM\_PAINT消息的优先级很低，它需要等消息队列中的其它消息发送完后才能被处理。调用UpdateWindow函数可使WM\_PAINT被直接发送到目标窗口，从而导致窗口立即重绘。**注意：函数绕过应用程序的**[**消息队列**](http://baike.baidu.com/view/262473.htm)**，直接发送WM\_PAINT消息给指定窗口的窗口过程，如果更新区域为空，则不发送消息。**

**解决方法：**

 双缓冲是一种基本的技术。我们知道,如果窗体在响应WM\_PAINT消息的时候要进行复杂的图形处理，那么窗体在重绘时由于过频的刷新而引起闪烁现象。解决这一问题的有效方法就是双缓冲技术。

      因为窗体在刷新时，总要有一个擦除原来图象的过程，它利用背景色填充窗体绘图区，然后在调用新的绘图代码进行重绘，这样一擦一写造成了图象颜色的反差。当WM\_PAINT的响应很频繁的时候，这种反差也就越发明显。于是我们就看到了闪烁现象。（

**当窗口由于任何原因需要重绘时，**  
**总是先用背景色将显示区清除，然后才调用OnPaint，而背景色往往与绘图内容**  
**反差很大，这样在短时间内背景色与显示图形的交替出现，使得显示窗口看起来**  
**在闪。如果将背景刷设置成NULL，这样无论怎样重绘图形都不会闪了。**  
**当然，这样做会使得窗口的显示乱成一团，因为重绘时没有背景色对原来**  
**绘制的图形进行清除，而又叠加上了新的图形。**） 我们会很自然的想到，避免背景色的填充是最直接的办法。但是那样的话，窗体上会变的一团糟。因为每次绘制图象的时候都没有将原来的图象清除，造成了图象的残留，于是窗体重绘时，画面往往会变的乱七八糟。所以单纯的禁止背景重绘是不够的。我们还要进行重新绘图，但要求速度很快，于是我们想到了使用BitBlt函数。它可以支持图形块的复制，速度很快。我们可以先在内存中作图，然后用此函数将做好的图复制到前台，同时禁止背景刷新，这样就消除了闪烁。以上也就是双缓冲绘图的基本的思路。

　　BitBlt函数执行颜色数据的位块传送，从指定的源设备描述表向给定的一个目的设备描述表传送对应于一个象素矩形的颜色数据。

BOOL BitBlt(HDC hdcDest,  
　　int nXDest,  
　　int nYDest,  
　　int nWidth,  
　　int nHeight,  
　　HDC hdcSrc,  
　　int nXSrc,  
　　int nYSrc,  
　　DWORD dwRop)；

hdcDest：指向目标设备环境的[句柄](http://baike.baidu.com/view/194921.htm" \t "_blank)。

nXDest：指定目标矩形区域左上角的X轴逻辑坐标。

nYDest：指定目标矩形区域左上角的Y轴逻辑坐标。

nWidth：指定源和目标矩形区域的逻辑宽度。

nHeight：指定源和目标矩形区域的逻辑高度。

hdcSrc：指向源设备环境的句柄。

nXSrc：指定源矩形区域左上角的X轴逻辑坐标。

nYSrc：指定源矩形区域左上角的Y轴逻辑坐标。

dwRop：指定光栅操作代码。这些代码将定义源矩形区域的颜色数据，如何与目标矩形区域的颜色数据组合以完成最后的颜色。

BLACKNESS：表示使用与物理[调色板](http://baike.baidu.com/view/1120949.htm" \t "_blank)的索引0相关的色彩来填充目标矩形区域，（对缺省的物理调色板而言，该颜色为黑色）。

DSTINVERT：表示使目标矩形区域颜色取反。

MERGECOPY：表示使用[布尔](http://baike.baidu.com/view/46060.htm" \t "_blank)型的AND（与）操作符将源矩形区域的颜色与特定模式组合一起。

MERGEPAINT：通过使用[布尔](http://baike.baidu.com/view/46060.htm" \t "_blank)型的OR（或）操作符将反向的源矩形区域的颜色与目标矩形区域的颜色合并。

NOTSRCCOPY：将源矩形区域颜色取反，于拷贝到目标矩形区域。

NOTSRCERASE：使用[布尔](http://baike.baidu.com/view/46060.htm" \t "_blank)类型的OR（或）操作符组合源和目标矩形区域的颜色值，然后将合成的颜色取反。

PATCOPY：将特定的模式拷贝到目标位图上。

PATPAINT：通过使用[布尔](http://baike.baidu.com/view/46060.htm" \t "_blank)OR（或）操作符将源矩形区域取反后的颜色值与特定模式的颜色合并。然后使用OR（或）操作符将该操作的结果与目标矩形区域内的颜色合并。

PATINVERT：通过使用XOR（[异或](http://baike.baidu.com/view/674171.htm)）操作符将源和目标矩形区域内的颜色合并。

SRCAND：通过使用AND（与）操作符来将源和目标矩形区域内的颜色合并。

SRCCOPY：将源矩形区域直接拷贝到目标矩形区域。

SRCERASE：通过使用AND（与）操作符将目标矩形区域颜色取反后与源矩形区域的颜色值合并。

SRCINVERT：通过使用[布尔](http://baike.baidu.com/view/46060.htm" \t "_blank)型的XOR（异或）操作符将源和目标矩形区域的颜色合并。

SRCPAINT：通过使用[布尔](http://baike.baidu.com/view/46060.htm" \t "_blank)型的OR（或）操作符将源和目标矩形区域的颜色合并。

WHITENESS：使用与物理[调色板](http://baike.baidu.com/view/1120949.htm" \t "_blank)中索引1有关的[颜色填充](http://baike.baidu.com/view/1139815.htm" \t "_blank)目标矩形区域。（对于缺省物理[调色板](http://baike.baidu.com/view/1120949.htm" \t "_blank)来说，这个颜色就是白色）。

具体实现：在窗口类中定义成员变量与成员函数：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103 | CBrush m\_brush;   CDC m\_memDC;//画在内存上   CBitmap m\_Bmp;   CWnd \*m\_pDrawWnd;     void InitialDBB();   void DrawOnMem(double \*pdData, unsigned long DataLen);   void DrawOnStaticArea(double \*pdData, unsigned long DataLen);    在该类的初始化函数中添加    BOOL CdaexpDlg::OnInitDialog()    {  。。。。。。  m\_brush.CreateSolidBrush(RGB(25,200,25));   m\_pDrawWnd=GetDlgItem(IDC\_PICTURE);   InitialDBB();  。。。。。。  }    在重绘函数中添加    void CdaexpDlg::OnPaint()  {  。。。  //已经有图像保存在缓冲区了    PAINTSTRUCT ps;    CRect rt;    m\_pDrawWnd->GetClientRect(&rt);    CDC\* pDC=m\_pDrawWnd->BeginPaint(&ps);    pDC->BitBlt(0, 0, rt.Width(), rt.Height(), &m\_memDC, 0, 0, SRCCOPY);    m\_pDrawWnd->EndPaint(&ps);    。。。  }    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*                                        初始化内存和画布  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/    void CdaexpDlg::InitialDBB()  {   CRect rt;      m\_pDrawWnd->GetClientRect(&rt);      CDC\* sDC = m\_pDrawWnd->GetDC();   // 为屏幕DC创建兼容的内存DC      if(!m\_memDC.CreateCompatibleDC(sDC))//   {     ::PostQuitMessage(0);   }      m\_nHeight = rt.bottom - rt.top;   m\_nWidth = rt.right - rt.left;   // 创建位图,不能是m\_memDC，否则无颜色   m\_Bmp.CreateCompatibleBitmap(sDC, rt.Width(), rt.Height());//m\_memDC   // 相当于选择画布,m\_pDrawWnd->      ::SelectObject(m\_memDC.GetSafeHdc(), m\_Bmp);      m\_pDrawWnd->ReleaseDC(sDC);  }    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*                                        绘制输出波形至内存  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  void CdaexpDlg::DrawOnMem( )  {   unsigned long i;  //---------------------------------------------------   //获取绘图区域      CRect rect;      m\_pDrawWnd->GetClientRect(&rect);    //----------------------------------------------------   //将背景涂黑      COLORREF crl = RGB(0,0,0);      m\_memDC.FillSolidRect(rect, crl);  //----------------------------------------------------   //设置画笔为黑色      CPen pen(PS\_SOLID,1,RGB(25,200,25));      m\_memDC.SelectObject(&pen);          // ======================================================    // = 画出波形     m\_memDC.MoveTo(  ,  );      m\_memDC.LineTo(  ,  );    }    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*                                        将内存内的波形绘制到界面  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  void CdaexpDlg::DrawOnStaticArea( )  {      CWnd\* pWnd = GetDlgItem(IDC\_PICTURE);//获得静态文本框的窗口对象      CRect rect;      pWnd->GetClientRect(&rect);      CDC\* pDC = pWnd->GetDC();      DrawOnMem( );      // 一次性的将内存设备环境上绘制完毕的图形"贴"到屏幕上      pDC->BitBlt(0, 0, rect.Width(), rect.Height(), &m\_memDC, 0, 0, SRCCOPY);      pWnd->ReleaseDC(pDC);  } |

**如何提高绘图的效率**  
    电力系统的网络图形的CAD软件，在一个窗口中往往要显示成千上万个电力元件，而每个元件又是由点、线、圆等基本图形构成。如果真要在一次重绘过程重画这么多元件，可想而知这个过程是非常漫长的。如果加上了图形的浏览功能，鼠标拖动图形滚动时需要进行大量的重绘，速度会慢得让用户将无法忍受。怎么办？只有再研究研究MFC的绘图过程了。  
    实际上，在OnDraw(CDC \*pDC)中绘制的图并不是所有都显示了的，例如：你  
在OnDraw中画了两个矩形，在一次重绘中虽然两个矩形的绘制函数都有执行，但是很有可能只有一个显示了，这是因为MFC本身为了提高重绘的效率设置了裁剪区。裁剪区的作用就是：只有在这个区内的绘图过程才会真正有效，在区外的是无效的，即使在区外执行了绘图函数也是不会显示的。因为多数情况下窗口重绘的产生大多是因为窗口部分被遮挡或者窗口有滚动发生，改变的区域并不是整个图形而只有一小部分，这一部分需要改变的就是pDC中的裁剪区了。因为显示（往内存或者显存都叫显示）比绘图过程的计算要费时得多，有了裁剪区后显示的就只是应该显示的部分，大大提高了显示效率。但是这个裁剪区是MFC设置的，它已经为我们提高了显示效率，在进行复杂图形的绘制时如何进一步提高效率呢？那就只有去掉在裁剪区外的绘图过程了。可以先用pDC->GetClipBox()得到裁剪区，然后在绘图时判断你的图形是否在这个区内，如果在就画，不在就不画。  
如果你的绘图过程不复杂，这样做可能对你的绘图效率不会有提高。