MFC 连连看开发日志

2017 年 5 月 27 日 星期六
10:18

1.关于 MFC 对话框最小化后从任务栏还原就出发中断的问题

本文转自：http://blog.sina.com.cn/s/blog_4b44e1c00100mdkl.html

程序最小化后不能还原，一般原因是程序中至少存在一个 Popup 类型的窗口引起的，因为 Popup 类型的子窗口即使由于父窗口的隐藏而隐藏，其 WS_VISIBLE 属性仍然是可见的，当用户再次点击任务栏的程序图标时，Popup 窗口会拦截系统（还原）消息，使主程序框架无法接收到系统消息，从而导致主程序无法正常还原。如果将其修改为 Child 类型的窗口，那么主程序的最小化和还原的功能就可以正常了。不过在实际项目中，往往就需要一个 Popup 类型的窗口作为子窗口（Popup 类型的窗口也可以有父窗口），那么这又如何解决程序最小化后不能还原的问题呢？根据以上分析的原理，只要在主程序最小化时，相应也隐藏掉 Popup 窗口（ShowWindow(SW_HIDE)），这样系统消息就能够正确传递了；当主程序还原时，再将隐藏的 Popup 窗口显示出来，这样就既不影响程序的显示效果，又能解决问题了！具体方法如下：

首先需要在主程序（如 MainFrame）中拦截系统消息（响应最大化，最小化，还原，关闭等消息的地方）。其消息为 WM_SYSCOMMAND.如在 MainFrame.h 头文件中加入 afx_msg void OnSyscommand(UIND nID,LPARAM lParam);在 MainFrame.cpp 的 BEGIN_MAP 与 END_MAP 之间加入 ON_WM_SYSCOMMAND，响应函数为 void CMainFrame::OnSyscommand(UIND nID,LPARAM lParam){}。

其次根据系统消息对 Popup 窗口进行隐藏与显示操作，代码如下：

```
CWnd* m_pPopupWnd;/// Popup 类型的窗口指针
void CMainFrame::OnSyscommand(UIND nID,LPARAM lParam)
{
	static BOOL s_bDialogVisible = FALSE;
	/// 如果是最小化消息
	if(SC_MINIMIZE == nID)
	{
		if(NULL != m_pPopupWnd
&& ::IsWindow(m_pPopupWnd->m_hWnd))
		{
			if(::IsWindowVisible(m_pPopupWnd->m_hWnd))
			{
				s_bDialogVisible = TRUE;
				/// 隐藏 Popup 类型窗口
				m_pPopupWnd->ShowWindow(SW_HIDE);
			}
		}
	}
	else
	{
```

```
            if(NULL != m_pPopupWnd
&& ::IsWindow(m_pPopupWnd->m_hWnd))
            {
                  if(TRUE == s_bDialogVisible)
                  {
                        s_bDialogVisible = FALSE;
                        /// 显示 Popup 类型窗口
                        m_pPopupWnd->ShowWindow(SW_SHOW);
                  }
            }
      }
      CWnd::OnSyscommand(nID,lParam);
}
```

方法二：拦截系统的还原消息，对其进行自定义的操作，如先设置为活动窗口，然后继续执行还原操作。

```
BOOL PreTranslateMessage(MSG* pMsg)
{
            ASSERT(pMsg);
              /// 如果是激活窗口消息
            if(pMsg->message == WS_APPACTIVE)
                {
                        /// 如果是按下左键
                        if(pMsg->wParam == VK_LBUTTON)
                        {
                              ASSERT(AfxGetMainFrame());
                              /// 激活主窗口
                              SetActiveWindow(AfxGetMainFrame()->
m_hWnd);
                        }
                  }
            /// 可继续向基类传递消息
            return     C**APP::PreTranslateMessage(pMsg);
}
```

真正的问题所在：

问题出在背景图加载函数 InitBackground()上，将位图资源加载进 dc 内存后直接绘制图像，导致最小化窗口还原后无法重绘，要想正常重绘，必须将绘制图像的函数放进 OnPaint()函数；包括 CGameDlg 控制的游戏窗体类同理，加载游戏背景和游戏地图元素的逻辑，都是要先将位图加载进相应的 CDC 位图内存，然后执行各自的绘制或重绘，并且要保证游戏地图元素在游戏进行状态还原窗口后也能实现重绘，这就需要在 onPaint()中加入一个判断语句：

```
if (m_bPlaying)     //如果游戏处于开始状态，则需要重绘游戏地图，主要是为了窗口最小化还原后可以自动重绘元素
```

```
UpdateMap();     //如果是刚进入游戏界面，还没有点击“开始游戏”，则不需要加载游戏地图
```

2.图游戏片元素的组织问题

一开始没能理解老师给的操作步骤原理，看网上别人写的连连看都是给图片编号，根据游戏地图数组存储的随机编号调用显示相关图片。经过分析，原来课件中的思想是，把所有游戏图片元素组合在一起，相当于一个一维图片组，根据游戏地图中的图片编号确定图片在元素图片组和掩码图片组中的位置，然后利用 BitBit()函数将其提取并做位运算处理，之后显示在游戏地图对应位置。

3.游戏相关的 C++类的组合问题：CGameDlg、CGameControl、CGameLogic、CGameException

CGameDlg 类负责游戏界面的交互和消息事件响应，其中包含 CGameControl 类的对象，用于实现游戏的控制，包括初始化游戏地图、设置选中点的信息、消子判断等；CGameControl 类中创建了 CGameLogic 类的对象，用于实现游戏的逻辑控制，包括随机生成游戏地图、游戏图片的连通判断、连通路径的记录和消子等，逻辑性强、算法最复杂。CGameException 类贯穿各个类，用于处理游戏中的一些异常事件。

4.游戏胜负判断的算法优化

老师给的游戏胜负判断的参考算法是在 m_GameLogic 对象中用 IsBank(int** pGameMap)来遍历游戏地图二维数组，判断其中的元素是否全部置为空，然后在 m_GameC 类中用 IsWin()调用 IsBank(pGameMap)函数，判断胜负。在每次选中两张图片并判断可以消除后，都要调用一次 IsWin()来判断胜负，也就是要每次都遍历一次 10*16 的二维数组，效率非常低。因此，可以在 CGameControl 类中定义一个 int 变量 clearPic 用于记录消除的图片数，每次消除后 clearPic 自加 2，与图片总数比较，若相等则说明所有图片消除完毕，玩家胜利，这样就避免了遍历二维数组带来的时空效率的浪费。

```
/*根据消除的图片数判定胜负*/
bool CGameControl::IsWin(void)
{
    /*如果消除的图片数与原有图片数相等，则判定玩家取胜；优化了每次遍历二维地图数组带来的时空复杂度*/
    if (clearPic == s_nRows*s_nCols)
    {
        clearPic = 0;   //重置计数器，为下一轮做准备
        return true;
    }
    else
        return false;
}
```

5.游戏地图元素数据利用两个随机数重排使游戏进程阻塞的问题

CGameLogic 类中的 RerankGraph()函数中使用了如下代码随机生成两个坐标：

```
do {
    // 随机得到第一个坐标
    int nIndex1 = rand() % nVertexNum;
    x1 = nIndex1 / nCols; y1 = nIndex1 % nCols;
} while (pGameMap[x1][y1] != BLANK);   //直到第一个元素数据不为空

do {
    // 随机得到第二个坐标
    int nIndex2 = rand() % nVertexNum;
    x2 = nIndex2 / nCols; y2 = nIndex2 % nCols;
} while (pGameMap[x2][y2] != BLANK);                    //直到第二个元素数据不为空
```

出现的问题是每当点击“重排”按钮后都会很长时间无响应，不会真正实现重排，而且导致了游戏其他进程阻塞。开始我以为是两个随机数生成的筛选条件可能有点苛刻，导致长时间无法生成 2 个符合要求的随机数才导致运行变慢。然而开始游戏后，直接点击“重排”，进度条本来正在加载阶段，之后直接停在中间不动了。于是，我把其中一个坐标只用一次随机数生成，另一个任然用 while 循环生成地图数据不为空的坐标，游戏开始后 10s，进度条加载完毕，正常计时，这时点击“重排”，可以实现重排。如果游戏一开始就重排，仍然会导致阻塞。进一步测试，如果点击“重排”后，值生成 2 个简单随机数，不进行复杂的筛选，仍然会在进度条加载阶段卡死。所以问你题在于进度条的加载导致阻塞，而 2 个严格的随机数筛选也是游戏运行变慢、有效地图坐标命中率低下无法实现重排的因素。那么就要解决进度条加载的问题。

6.边缘图片的消除和内部图片区域外引线消除情况的统一解决方案

游戏地图数组外层加一“圈”，并初始化为 BLANK（-1），需要同步修改的其余部分有：加载游戏图片是需要从编号为 1 的行列开始，遍历数组寻找通路时扩展到 0 至 nRows+1 和 0 至 nCols+1；其余部分可直接复用。

```
// 游戏地图开辟内存空间
int** pGameMap = new int*[nRows + 2];
if(NULL == pGameMap)
{
    throw new CGameException(_T("内存操作异常！"));
}
else
{
    for (int i = 0; i < nRows + 2; i++)
    {
        pGameMap[i] = new int[nCols + 2];
        if(NULL == pGameMap)
        {
            throw new CGameException(_T("内存操作异常！"));
        }
```

```
            memset(pGameMap[i], BLANK, sizeof(int) * (nCols + 2));  //初
            始化数组为BLANK(-1)
        }
    }
```