## 开发过程

### 1.1 打开视频文件

|  |  |
| --- | --- |
| 首先要设置几个全局变量，分别是用于保存视频图像信息的Mat对象、用于读取视频文件的VideoCapture对象、用于保存视频文件路劲和文件名的WCHAR数组对象以及相应的指针。 | |
|  | |
| 然后我们还定义了OpenVideoFile函数，该函数通过CoCreateInstance函数创建打开mp4或avi文件的对话框并获取文件完整路径，返回值为boolean类型。如果成功打开了mp4文件或avi文件，则通过GetDisplayName获取IShellItem对象的显示名称，fn接收指向检索到的显示名称的指针的地址，同时返回true；否则返回false。 | |
|  | |
| 由于VideoCapture的open方法接受的参数为string类型的引用传递，因此我们还需要定义WCHAR2String函数将WCHAR数组转换为字符串string。该函数主要通过windows提供的WideCharToMultiByte函数实现转换。 | |
|  | |
| 接着在WndProc函数中配置相应的WM\_COMMAND的消息。result用于接收获取视频文件路径的结果，result为true则表明视频文件路径合法，否则弹出错误提示框，表明视频未能打开。  于是我们再调用VideoCapture的open方法读取视频文件，并用opened变量接收读取结果。opened为true则表明视频读取成功，此时我们便直接通过重构的流运算符>>获取第一帧图像，调用InvalidateRect方法激发WM\_PAINT事件并重绘窗口。 | |
|  | |
| 最后再在WM\_PAINT消息中添加重绘窗口的代码。当接收到视频图像信息时（即mImg.rows>0或sImg.rows>0时），则绘制相应的视频画面。 | |
|  | |
| 由于直接绘制图像容易失真，因此在绘制之前我先调用resize方法重新调整图像的大小，然后再转换视频图像的颜色并依此计算一个像素的字节大小。 | |
|  | |
| 绘制视频画面的核心在于制作一个bitmapinfo的数据头，配置相应参数让其数据指针指向Mat类型数据mImg和sImg。 | |
|  | |
| 最后再调用StretchDIBits方法实现指向窗口句柄进行图像绘制。需要注意调整主视频和画中画视频的位置和大小。 | |
|  |  |

### 1.2 播放控制

|  |
| --- |
| 首先定义表示视频播放状态的枚举类型PlayState，然后再设置两个全局变量mPlayState和sPlayState，分别表示视频1和视频2的播放状态，其默认值为stopped。 |
|  |
| 接着处理WM\_COMMAND消息的响应事件，根据用户选择对视频的播放状态进行相应的改变。CAP\_PROP\_POS\_FRAMES参数表明接下来要解码/捕获的帧的基于0的索引，因此当用户选择“停止”的时候，我们设置VideoCapture的CAP\_PROP\_POS\_FRAMES参数为0，表明结束播放。 |
|  |
| 最后处理WM\_TIMER消息的响应事件，只有在视频文件读取成功并且视频的播放状态为playing时，才通过视频读取器VidCap1或VidCap2读取下一帧的画面，并调用InvalidateRect方法重绘窗口。 |
|  |
|  |
| 此外，实验任务还要求我们两个视频都能分别周而复始地播放。循环播放的效果也是在WM\_TIMER消息的响应中实现的，当抓取到的下一帧图像为空时，表明该视频已经播放结束，因此我们只需重新获取视频文件的第一帧图像即可。  除了代码中所使用的方法，我们同样可以使用设置CAP\_PROP\_POS\_FRAMES参数为0的方法来实现这一效果。 |
|  |

### 1.3 画中画效果

|  |  |
| --- | --- |
| 实现画中画的效果，其实就是在StretchDIBits改变画中画视频的播放位置。但是因为主视频跟画中画视频是在交替显示，因此仅通过这种方式，会导致画中画视频在播放时频繁地闪烁。 | |
|  |  |
| 所以在处理WM\_TIMER的响应事件时，在视频2打开并播放的情况下，我加入了画中画效果调整的处理函数来解决这一问题。 | |
|  | |
| 在视频2播放时，如果视频1也有视频图像，表明此时必定处于画中画模式。为了能够准确定位主视频的ROI区域位置，我们先通过resize函数重新调整视频1和视频2的图像大小。然后再使用copyTo方法将画中画图像sImg深复制到主视频图形mImg的ROI区域（即画中画的位置），事先将画中画图像合成到主视频图像上，解决了画中画画面的闪烁问题。 | |
|  | |

### 1.4 拖拽画中画视频

|  |
| --- |
| 首先设置两个全局变量draw\_\_mode和mouse\_L\_pressed，draw\_mode用于表示当前所处模式，其值为0时则表示无操作，为1表示绘图模式，为2表示拖拽模式；mouse\_L\_pressed则用于判断鼠标左键是否为按下状态，初始值为false，表示鼠标左键未按下。然后再设置5对全局变量，分别是：   1. 用于记录画中画视频左上角位置的cur\_x和cur\_y 2. 用于记录鼠标拖拽开始点相对画中画视频位置的rel\_x和rel\_y 3. 用于记录拖拽结束点相对于拖拽开始点的偏移位置offset\_x和offset\_y 4. 用于记录画中画视频左上角新位置的new\_x和new\_y 5. 用于记录当前鼠标位置的x和y |
|  |
| 接着再在WndProc函数中配置相应的响应事件。首先处理WM\_COMMAND消息，当选择“开始拖动”时，如果存在画中画视频画面，则设置draw\_mode的值为2；当选择“结束拖动”时，则令draw\_mode=0，表示现在处于不操作的模式。 |
|  |
| 然后处理鼠标响应的消息，当鼠标左键按下时，设置mouse\_L\_pressed为true，并通过GET\_X\_LPARAM(lParam)方法和GET\_Y\_LPARAM(lParam)方法读取当前鼠标位置x和y，再分别令x和y减去画中画视频左上角坐标cur\_x和cur\_y，得到鼠标拖拽开始点相对画中画视频位置的rel\_x和rel\_y。  当鼠标左键松开时，则设置mouse\_L\_pressed为false。 |
|  |
| 接着处理WM\_MOUSEMOVE消息。首先获取x和y的坐标值，如果鼠标左键按下并且处于拖动模式中，再判断rel\_x和rel\_y的值是否均在[0,200]的区间内。  如果是则表明拖拽点位于画中画区域，并通过(cur\_x+rel\_x)-x和(cur\_y+  rel\_y)-y计算出偏移位置offset\_x和offset\_y，再令视频窗口左上角位置减去偏移位置，得到新的视频窗口左上角位置new\_x和new\_y。此时我们再判断新的画中画位置是否超出原视频，如超出则拖动无效，未超出则更新cur坐标。 |
|  |
| 最后再修改WM\_TIMER和WM\_PAINT事件的相应代码，在主视频和画中画视频共存的情况下则修改主视频ROI区域的坐标；只有单独的画中画视频的情况下，则直接修改StretchDIBits方法的相应参数即可。 |
|  |
|  |

### 1.5 在视频画面上绘制线条

|  |
| --- |
| 绘制线条的实现方法大致与拖拽窗口相同。之前我们已经设置了两个全局变量draw\_\_mode和mouse\_L\_pressed，draw\_mode为1表示绘图模式。  然后处理WM\_COMMAND消息，当选择“开始绘制”时，如果主视频或画中画视频有图像信息，则设置draw\_mode的值为1；当选择“结束绘制”时，则令draw\_mode=0，表示现在处于不操作的模式。 |
|  |
| WM\_LBUTTONDOWN和WM\_LBUTTONUP的处理与拖拽视频位置相同，此处便不再赘述。我们接着处理WM\_MOUSEMOVE消息，当鼠标左键按下并且为绘图模式时，我认为画中画的优先级应该高于主视频，因此我们首先判断是否有画中画图像，如果则通过之前获取的**相对坐标**直接用circle函数画出一个半径为8的实心红点；然后再判断是否有主视频图像，如果有则直接使用**鼠标坐标**用circle函数画半径为8的实心红点。 |
|  |

### 1.6 边缘图滤镜

|  |
| --- |
| 首先定义表示滤镜效果的枚举类型VideoEffect，再分别设置两个全局变量mVidEffect和sVidEffect，表示视频1和视频2的画面效果，其默认值为no。 |
|  |
| 接着处理WM\_COMMAND消息中相应的响应事件，根据用户选择对视频的画面效果mVidEffect和sVidEffect进行相应的改变。 |
|  |
| 然后处理WM\_TIMER消息，在视频打开并处于播放状态时，创建两个Mat对象edgeY和edgeX，然后再用Sobel方法分别获取待处理图像x方向和y方向的边缘图像，并将其保存在edgeY和edgeX中。最后再通过重载运算符+直接令edgeX+edgeY，得到mImg或sImg的边缘图。 |
|  |

## 难点解析

### 2.1 解决画中画闪烁问题

|  |
| --- |
| 实现画中画的效果非常容易，因为核心就是在WM\_PAINT响应事件中通过改变StretchDIBits函数的参数来改变画中画视频的播放位置。  这样处理的话，因为主视频跟画中画视频是在交替显示，所以会导致画中画视频在播放时频繁地闪烁。一开始我是想通过修改WM\_TIMER定时器的值，加快视频画面的切换速度来解决，但是发现这样也无济于事。  为了解决这一问题，我在网上搜索了一下其他画中画的实现方法。在<https://blog.csdn.net/dzhuang123/article/details/45054497>中，该作者虽然是使用yuv解码的方式实现，但是其核心思想在于通过设置ROI区域合并主视频与画中画视频的图像，从而实现画中画的效果。 |
|  |
| 因此在处理WM\_TIMER的响应事件时，在视频2打开并播放的情况下，我加入了画中画效果调整的处理函数来解决这一问题。 |
|  |
| 在视频2播放时，如果视频1也有视频图像，表明此时必定处于画中画模式。为了能够准确定位主视频的ROI区域位置，我们先通过resize函数重新调整视频1和视频2的图像大小。然后再使用copyTo方法将画中画图像sImg深复制到主视频图形mImg的ROI区域（即画中画的位置），事先将画中画图像合成到主视频图像上。  虽然这种方法并没有从原理上解决画中画画面的闪烁问题，由于此时是**合并了画中画视频图像的主视频图像**与**画中画视频图像**交替切换，因此在实现效果上该问题已得到解决。 |

### 2.2 实现拖拽画中画视频效果

|  |
| --- |
| 拖拽画中画视频实际上就是改变画中画视频的位置。为了更好地解释该原理，我在主视频中建立平面直角坐标系，定义并记录以下几个点：“画中画窗口”的定位点**(cur\_x,cur\_y)**；“鼠标点击并拖拽”的定位点**(rel\_x,rel\_y)**。  当用户点击鼠标开始拖拽时，首先会记录拖拽结束的新的点**(x,y)**，将(rel\_x,rel\_y)-(x,y)可以得出偏移量**(offset\_x,offset\_y)**，此时仅需要用“画中画窗口”的定位点(cur\_x,cur\_y)减去偏移量(offset\_x,offset\_y)，就可以得到“画中画窗口”左上角新的定位点**(new\_x,new\_y)**。  至此鼠标拖动结束，画中画视频窗口由原先的位置改变至新的位置。  在这个过程中，我们还需要注意两个细节。首先是在鼠标拖拽时，我们需要判断鼠标是否位于画中画窗口内，如果不是则不能称之为“拖动”。除此之外，我们还需要注意画中画视频拖拽的结束位置是否超出了原视频窗口位置，如果不是则不能称之为“画中画”。 |
|  |
| 因此，在理解了原理之后，我们需要处理的核心代码是鼠标响应的消息。  当鼠标左键按下时，我们可以并通过GET\_X\_LPARAM(lParam)方法和GET\_Y\_  LPARAM(lParam)方法读取当前鼠标位置x和y，再分别令x和y减去画中画视频左上角坐标cur\_x和cur\_y，得到鼠标拖拽开始点相对画中画视频位置的rel\_x和rel\_y。 |
|  |
| 接着处理WM\_MOUSEMOVE消息。首先获取x和y的坐标值，如果鼠标左键按下并且处于拖动模式中，再判断rel\_x和rel\_y的值是否均在[0,200]的区间内。  如果是则表明拖拽点位于画中画区域，并通过(cur\_x+rel\_x)-x和(cur\_y+  rel\_y)-y计算出偏移位置offset\_x和offset\_y，再令视频窗口左上角位置减去偏移位置，得到新的视频窗口左上角位置new\_x和new\_y。此时我们再判断新的画中画位置是否超出原视频，如超出则拖动无效，未超出则更新cur坐标。 |

### 2.3 实现动态绘制效果

虽然在“使用鼠标在视频画面上绘制线条”这一部分中仅实现了在静态画面中的绘制，但在撰写实验报告的时候已经有了如何实现动态画面绘制的思路了，由于时间和精力有限没能将想法呈现出来。

首先难以实现的原因是视频的播放实际上是多个静态画面的逐帧播放，在动态画面中绘制时，当前帧所绘制的图案，在下一帧就会被替换掉，导致图案不能保留在画面中。

为了解决这个问题，可以在进入“开始绘制”模式时，创建一个链表或其他数据结构。在用户绘制时，通过保存用户当前帧绘制的坐标点，在视频播放至下一帧时，复现上一帧保存的绘制过程，同时再次记录并保存这一帧绘制的坐标点，并再在下一帧中复现。以此类推，每一帧绘制的图案都能“记录”→“保存”→“在下一帧复现”，以此实现动态画面的绘制。

此外还可以在菜单栏中新增功能“清除画面”，当用户选择“清除画面”时，则释放该数据结构，清空所有保存的绘制图案数据，使视频能正常播放。