# Efficient Cross-Platform Development: Real-Time Video Processing

Applications capable of running on many different types of hardware and software environments are usually called "cross-platform".

In the context of this article "cross-platform code" means source code written with a programming language and tools (e.g. libraries) capable of running on different Operating Systems (OS) and hardware platforms; cross-platform code thus can be compiled&executed or interpreted, without any change, on e.g. a Windows PC or a Linux workstation.

Cross-platform code widens the number of potential users of the software while maintaining a single code base, thus providing strong advantages over single-platform apps. Cross-platform code is typically associated with Java, given its popularity and the number of platforms for which a Java Virtual Machine is available. There is, however, a number of areas where Java is not the best solution and compiled languages are required because of their better runtime performances both in terms of memory footprint and execution speed.  C++ is often under-evaluated for cross-platform applications but thanks to the many cross-platforms C/C++-based libraries available nowadays, programming for multiple platforms in C++ is actually easy!

Moreover, C++ can easily be used both for applications loosely connected to their surrounding environment (think to e.g. word processors which mostly need to process just keyboard and mouse inputs and read/write to some file system) and for those applications requiring instead a much tighter integration with the operating system and possibly with the hardware (think to e.g. apps processing video streams, network-intensive software or apps interacting with external data acquisition devices). Ensuring that such applications are capable of running in very different software environments and capable of delivering the desired performances is challenging and often requires good knowledge of the underlying software layers.

Note that there are many cross-platform C/C++ libraries for Graphical User Interface (GUI) creation: GTK, Qt, wxWidgets just to name a few!

Many studies quantified the maximum latency which can be tolerated for different tasks [1-3]; it varies from 100ms for visual feedback to keyboard inputs (e.g. when writing text) up to few seconds for less frequent interactions.

In this article we consider as case of study a C++ cross-platform application which uses the wxWidgets library for its GUI, which processes a stream digitized by a camera using the OpenCV library and which needs to run under relatively-tight memory and speed constraints. This example covers many practical situations involving e.g. surveillance systems, embedded computer vision systems for autonomous robots, medical imaging systems involving large datasets, etc. Writing such a software requires many skills including: configuration of the dependencies (libraries), knowledge of different libraries and how they can be interfaced with each other (data conversion), parallel programming (at least at basic level), object-oriented and modular code programming. Even building and linking together heterogeneous code is often non-trivial! In this article some suggestions and guidelines for writing cross-platform C++ code will be provided, with emphasis on the problems of a) integration of cross-platform and platform-specific code b) efficiency in terms of both memory and speed.

## Cross-Platform, Responsive GUIs

Programming a good GUI is not an easy task. Users want the GUI to be a) intuitive (easy to use but still capable of providing access to all the application's features) and b) responsive. Responsiveness typically depends on the amount of processing which is triggered by user actions and on the number of layers through which the processed information needs to propagate before it is actually rendered on the screen. Of course for cross-platform applications the number of layers is higher than for native applications, but it is user-triggered processing that typically dominates the GUI latency. Multi-threading is very useful in this sense since it allows the GUI to be responsive to user actions even while performing CPU-intensive applications and thus is often absolutely necessary for data-intensive applications.

Like most modern toolkits, wxWidgets provides to the user all the tools required for multi-threaded GUI creation and user-event processing in a thread-safe manner (even if more advanced users may prefer to use the tools provided by specific libraries like e.g. Boost.Thread or even the now-standardized C++11 threading facilities). The basic concept is that wxWidgets internally handles an "event loop" (a.k.a. "message pump") in the "main thread" (the thread which runs the "main()" of the application and which is started by the OS). All the event handling code is called by such event loop. Listing 1 contains a simple, basic wxWidgets application which just handles clicks on the menu item "Compute"; this control triggers a long computation which is executed in the main thread, thus freezing the GUI! Listing 2 shows the same application written in the correct way: when the user clicks on the "Compute" menu item, a secondary thread is created and a `wxEVT_COMMAND_MYTHREAD_UPDATE` event informs the main thread that the computation is going on so that updated results can be shown to the user. (Note that the secondary thread in this case is implemented inside the same class object `MyFrame` using the `wxThreadHelper` model [6] which allows the developer to avoid the use of lots of pointers to connect the secondary thread with the main one.) The most important concept here is that the long operation needs to be coded in sequential steps (i.e. not as a single uninterruptible operation) because in secondary threads the `wxThread:: TestDestroy()` function needs to be called regularly. In our example the long computation is split in different steps using a "ComputationState" object and the `LongComputationSplitted()` function. Finally, note the need of synchronization objects like `wxMutex`, which is the cross-platform wxWidgets-equivalent of POSIX mutexes, and like the helper class `wxMutexLocker`, very useful to avoid forgetting a `wxMutex::Unlock()` call in some exit paths. The big difference of the example in Listing 2, compared to the one in Listing 1, is that while the computation is ongoing the GUI will be able to redraw, handle user clicks, show a progress bar (with some additional code), etc. For small applications, the wxThreadHelper, wxThreadEvent, wxThread and `wxMutex` classes are typically enough for building a responsive, cross-platform multithreaded GUI.

## Application Architecture

Building on the simple example of Listing 2, let us now consider a cross-platform video processing application. In particular, a video acquired from a camera attached to the computer is processed and rendered in real-time. Thanks to some cross-platform libraries many of the involved tasks can be implemented writing only

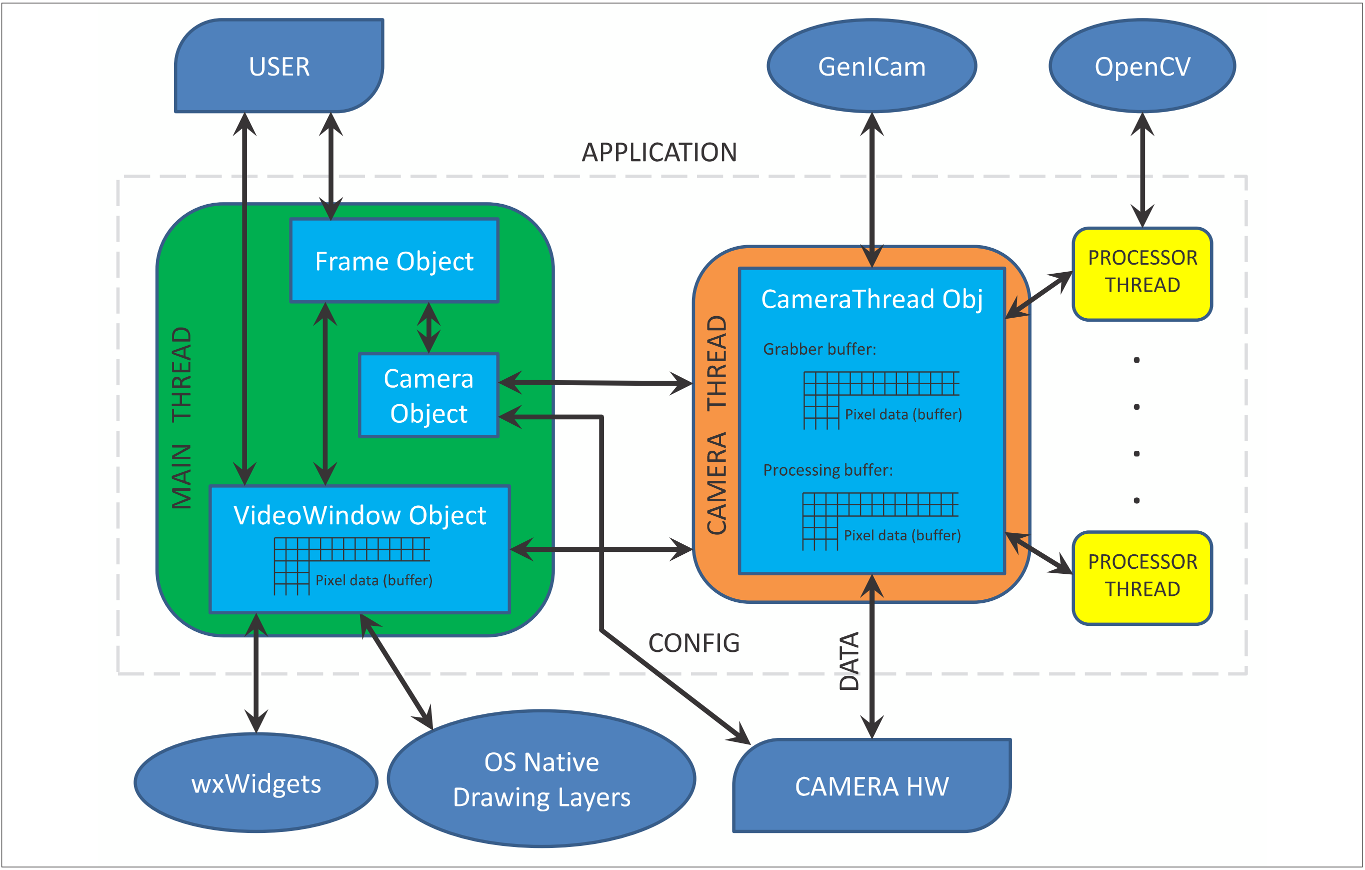

**Figure 1:** *Architecture of a cross-platform video-processing application*

**Listing 1:** *Basic single-threaded application*

```
#define myID wxID _ CUSTOM+1
class MyFrame : public wxFrame
{
public:
  MyFrame() :
    wxFrame(NULL, wxID _ ANY, "The Main Frame")
    {
    wxMenu* fileMenu = new wxMenu;
    wxMenuBar* menuBar = new wxMenuBar();
    fileMenu->Append(myID, "Compute!");
    menuBar->Append(fileMenu, "File");
    SetMenuBar(menuBar);
  Connect(myID, wxEVT _ COMMAND _ MENU _ SELECTED,
    wxCommandEventHandler(MyFrame::OnCompute));
  }
void OnCompute(wxCommandEvent& evt);
  ...
  };
void MyFrame::OnCompute(wxCommandEvent&)
{
  // the user clicked the menu item
  "Compute!"...
  // start the computation of the quantity of
  // interest:
  LongComputation();
    // PROBLEM: the GUI will freeze till the
    // computation is ongoing!
}
```

**Listing 2:** *Multi-threaded application*

```
#define myID
wxID_CUSTOM+1

wxDECLARE_EVENT(
  wxEVT_MYTHREAD_UPDATE,
  wxThreadEvent)

wxDEFINE_EVENT(
  wxEVT_MYTHREAD_UPDATE,
  wxThreadEvent)

class MyFrame : public wxFrame,
                public wxThreadHelper
{
public:
  MyFrame() :
    wxFrame(NULL, wxID _ ANY, "The Main Frame")
    {
    wxMenu* fileMenu = new wxMenu;
    wxMenuBar* menuBar = new wxMenuBar();
    fileMenu->Append(myID, "Compute!");
    menuBar->Append(fileMenu, "File");
    SetMenuBar(menuBar);

   Connect(myID, wxEVT_COMMAND_MENU_SELECTED,
      wxCommandEventHandler(MyFrame::OnCompute));
   Connect(wxID_ANY, wxEVT_MYTHREAD_UPDATE,
      wxCommandEventHandler(MyFrame::OnThreadUpd
      ate));
  }
  ...
```

```
  void OnCompute(wxCommandEvent& evt);

  void OnThreadUpdate(wxThreadEvent& evt);
  wxThread::ExitCode Entry();

  ComputationData m _ data;
  wxMutex m _ mutexData; // protects field above
  };

void MyFrame::OnCompute(wxCommandEvent&)
{
  // the user clicked the menu item
  "Compute!"...
  // start the computation of the quantity of
  // interest:
  GetThread()->Run();
  }

wxThread::ExitCode MyFrame::Entry()
{
 // IMPORTANT: this function gets executed in the
 // secondary thread context! here we do our long
 // task, periodically calling TestDestroy():

  ComputationState status;
  while (!GetThread()->TestDestroy() ||
        status.IsComputationComplete()) {
    status = LongComputationSplitted(status);

    {
  // when modifying the data we exchange with
  // the main thread we need to lock it!

    wxMutexLocker lock(m_mutexData);
    data = status.GetUpdatedData();
  }
  wxQueueEvent(this,
        new wxThreadEvent(
                wxEVT_MYTHREAD_UPDATE));
  }

  return 0;
}
void MyFrame::OnThreadUpdate(wxThreadEvent& evt)
{
  wxMutexLocker lock(m _ mutexData);
  pSomeStaticText->SetValue(m_data.GetAsText());
}
```

portable code: e.g. GenICam-compatible cameras usually provide a SDK for image acquisition on all major platforms and the cross-platform OpenCV library implements lots of computer vision algorithms making video processing very enjoyable! The critical aspect in such application thus reduces to the efficient interoperability among the different libraries. Unnecessary memory allocations and copies increase both the CPU load and the memory footprint and thus it is very important to minimize the number of auxiliary buffers used for pixel data temporary storage. In Fig. 1 a block diagram depicts the interactions of the application with the mentioned libraries, with its internal parts and with the buffers used for video storage ().

The main thread (green box) handles the GUI; for simplicity, we consider just a frame window containing a canvas, called `VideoWindow`, which renders on screen the results of the video processing. The `Camera` object takes care of camera configuration and, most importantly, grabs the incoming frames from the hardware through a secondary thread (the "camera thread", depicted as the orange box). Note that the main thread has no processing or data-intensive task so that a short latency in response to user inputs is ensured. Moreover, the only significant memory occupation for the main thread is represented by the pixel buffer stored in the `VideoWindow` instance; a common mistake when writing cross-platform code is to explicitly allocating such buffer... this is not necessary as modern OS native layers already allocate internal memory for such purpose (e.g. in the Windows GDI API under the name of the "window device context" [4]; in the Cairo API under the name of the "Cairo context" associated to a GDK window [5]); more details will be discussed later since, to keep as low as possible the computational burden on the main thread, it is important to operate on this buffer using native graphic layers.

The camera thread task is to keep polling the camera for new frames and notify the main thread (specifically, the `VideoWindow` in this example) each time a new frame has been processed. The camera thread (orange box in Fig 1) holds two buffers: the first one, the "grabber buffer" is used to store the raw, acquired frame and is thus filled with the data provided directly by the camera. The second buffer, the "processing buffer", is used instead to store the result of the video processing. These two buffers are both required since processing routines typically require a) the source image to be completely available in the RAM (since the operation to perform on a pixel or on a group of pixel often depends on all nearby pixels!) and b) a destination buffer different from the source one (many operations are non-local, i.e. cannot store their output in the source buffer). Note that both the buffers can and should be allocated only once, for obvious performance reasons, in the app initialization and later just be re-used for each new incoming frame.

Regarding video processing, which is the most demanding task of such applications, it is important to exploit all available computing power, i.e. all the cores of modern CPUs. For this reason, it is very useful, if possible, to divide what usually is a sequential task in different parallelizable tasks as represented in Fig 1 by the yellow boxes. Note that two approaches can be devised:

1. the same frame is split in different areas (e.g. horizontal or vertical "slices") to be processed by different threads;

2. different threads process different frames contiguous in time.

Approach #1 grants the minimum processing latency and thus, when practical, is typically the best solution. Examples of common video processing include face recognition, edge detection, pattern matching and feature tracking; most of these algorithms can be parallelized at least partially, thanks to OpenCV being *thread-aware* (i.e. OpenCV functions can be called by different threads and the reference counting for image buffers, that is cv::mat objects in OpenCV dialect, is implemented with atomic operations). Moreover, thanks to the number of algorithms implemented in OpenCV these tasks usually reduce to few function calls! Note however that great care has to be put in the image formats required by the different algorithms, in image encapsulation in the library-specific data structures and, as always in parallel programming, in thread synchronization and concurrent data access.

## Video Acquisition and Processing

Skipping the details related to the specific hardware and camera configuration, in Listing 3 we show the skeleton of the video acquisition code (the orange box of Fig. 1), which in our example is implemented using some GenICam-compatible SDK. First, in CameraThread::Entry() the processor threads are allocated; then the main loop of the camera thread begins. Vendors typically provide customized SDKs with their own class definitions but the basic video acquisition pattern remains the same: when a new frame is available, it is copied on the PC memory (by the GenICam transport layer) and the user application:

1. asks the device the address of such buffer, which is what we called "grabber buffer" so far; in Listing 3 we assume that for the specific camera model adopted a `GenICam _ DeviceClass` is available and exposes a function: `GenICam _ DeviceClass::AcquireNewFrame()`
2. processes the buffer; note that the processing may also take modify the source buffer... in the next frame it will be overwritten, so that is not an issue; in Listing 3, the camera threads just offloads this processing to a number `NUM _ THREADS` of `ProcessorThread` objects with the call `ProcessorThread::StartProcessingNewFrame()`. Then, after all of the processor threads are started in parallel on different slices of the source frame (thus following the approach #1 previously mentioned), `CameraThread` waits that all of them complete

**Listing 3:** *Video stream acquisition*

```
class CameraThread : public wxThread

{
public:
  CameraThread() : wxThread(wxTHREAD _ JOINABLE) { /* ... allocate m _ pProcBuffer ... */ }
  ...
  wxThread::ExitCode Entry();
  void RedrawHappened() { m _ cWaitForRedraw.Signal(); }
  GenICam _ DeviceClass m _ device;
    // the camera device using the manufacturer-provided SDK!
  unsigned char* m _ pProcBuffer;
  VideoWindow* m _ pVideoWin;
  wxCondition m _ cWaitForRedraw;
  std::vector<ProcessorThread*> m _ threads;
};
wxThread::ExitCode CameraThread::Entry()
{
  // create all the processor threads
  for (unsigned int i=0; i < NUM _ PROCESSOR _ THREADS; i++) {
    ProcessorThread* p = new ProcessorThread();
    m _ threads.push _ back(p);
    p->Run();
  }
  m _ device.StartAcquisition();
  while (!TestDestroy()) {
    if (!m _ device.IsNewFrameAvailable())
    continue;
    unsigned char* pGrabberBuff = m _ device.AcquireNewFrame();
    unsigned int rowsPerThread = m _ device.GetImageSize().y/NUM _ PROCESSOR _ THREADS;
    unsigned int stride = m _ device.GetImageSize().x;
    for (unsigned int i=0; i < m _ threads.size(); i++)
    m _ threads[i]->StartProcessingNewFrame(pGrabberBuff[stride*rowsPerThread*i],
    m _ pProcBuffer[stride*rowsPerThread*i]);
    // now that all threads are processing (different portions of) this frame;
    //wait for their completion
    for (unsigned int i=0; i < m _ threads.size(); i++)
    m _ threads[i]->WaitForFrameProcessing();
    // draw the new, processed frame in the main thread context
    wxThreadEvent ev(wxEVT _ COMMAND _ NEW _ FRAME);
    ev.SetClientData((void*)m _ pProcBuffer);
    wxQueueEvent(m _ pVideoWin, ev.Clone());
    // wait for acquisition window to redraw itself...
    // in the meanwhile we must ensure that the m _ pProcBuffer pointer remains valid
    m _ cWaitForRedraw.Wait();
    // tell the device we're done with this buffer!
    m _ device.ReleaseFrame(pGrabberBuff);
  }
}
```

**Listing 4:** *Video stream processing*

```
class ProcessorThread : protected wxThread

{
public:
  ProcessorThread() : wxThread(wxTHREAD _ JOINABLE) { ... }
  void StartProcessingNewFrame(unsigned char* pGrabberBufferSlice,
                               unsigned char* pProcBufferSlice);
```

```
 void WaitForFrameProcessing();
 wxThread::ExitCode Entry();
 wxSize m _ szImage;
 unsigned char* m _ pGrabberBufferSlice;
 unsigned char* m _ pProcBufferSlice;
 wxMutex m _ mutexBufferSlices; // protects m _ pGrabberBufferSlice, m _ pProcBufferSlice
 wxCondition m _ cNewFrameToProcess,
        m _ cFrameProcessingStarted;
};
void ProcessorThread::WaitForFrameProcessing()
{
 m _ mutexBufferSlices.Lock();
  // wait until the image processor thread unlocks the m _ mutexBufferSlices mutex
  // (which means it has completed processing and is blocked
  // in m _ cNewFrameToProcess.WaitFor-Event())
}
void ProcessorThread::StartProcessingNewFrame(unsigned char* pGrabberBufferSlice,
                             unsigned char* pProcBufferSlice)
{
 WaitForFrameProcessing(); // ensure processing of previous frame has been finished!
 // we are now allowed to modify the buffer pointers!
 m _ pGrabberBufferSlice = pGrabberBufferSlice;
 m _ pProcBufferSlice = pProcBufferSlice;
 m _ mutexBufferSlices.Unlock(); // we're done with them
 m _ cNewFrameToProcess.Signal(); // inform the processor thread that it's clear to start
 // wait for it to effectively start the new frame processing
 // (if we don't do this, then it may happen that calling StartProcessingNewFrame() and
 // then WaitForFrameProcessing() blocks the processor thread and prevents it from effectively
 // processing the new frame data!)
 m _ cFrameProcessingStarted.Wait();
 // we are now sure that the processor thread has locked m _ mutexBufferSlices and is
 // processing the image data...
}
wxThread::ExitCode ProcessorThread::Entry()
{
 while (!TestDestroy()) {
  m _ cNewFrameToProcess.Wait(); // block until we have new data to process...
  {
   // signal that we're modifying the memory areas previously set for this thread
   wxMutexLocker lock(m _ mutexBufferSlices);
   // signal that we have started the processing of a new frame
   m _ cFrameProcessingStarted.Signal();
   // wrap the source & destination buffers in OpenCV-specific data objects
   cv::Mat source _ img(m _ szImage.y, m _ szImage.x, CV _ 8UC4, m _ pGrabberBufferSlice);
   cv::Mat dest _ img(m _ szImage.y, m _ szImage.x, CV _ 8UC4, m _ pProcBufferSlice);
   // process the new frame
   cv::adaptiveThreshold(source _ img, dest _ img,
    255 /* max value in the filtered result */,
    cv::ADAPTIVE _ THRESH _ MEAN _ C, cv::THRESH _ BINARY _ INV,
    20 /* block size */,
    0 /* constant subtracted by adaptive threshold */);
   /* ...do a lot of useful things on the dest _ img buffer... */
  }
 }
 return 0;
}
```

their task, calling `ProcessorThread::WaitForFrameProcessing();`

3. gives the control of the buffer back to the device object (in our example with the call `GenICam _ DeviceClass::ReleaseFrame()`).

The `ProcessorThread` implementation is shown in Listing 4. `ProcessorThread` is a `wxThread`-derived class whose task is to process video frames using OpenCV; the function `StartProcessingNewFrame()` is executed in the context of the camera thread (see Listing 3) but triggers the execution of code in `ProcessorThread::Entry()`, i.e. code in the "processor thread" context. A few comments regarding the synchronization code:

1. it is very important to ensure that the buffer slices processed with OpenCV in `ImageProcessorThread::Entry()` remain valid till the processing is complete… this is accomplished using the `m _ mutexBufferSlices` mutex;
2. the `wxCondition` object `m _ cNewFrameToProcess` is used to "sleep" the `ProcessorThread` while it is waiting for a new frame to process;
3. the `wxCondition` object `m _ cFrameProcessingStarted` is used to ensure that when the StartProcessingNewFrame() function returns, the `Entry()` function has locked the `m _ mutexBufferSlices` mutex, so that subsequent calls to `WaitForFrameProcessing()` will behave as expected;

Turning our attention to the inner code of `ProcessorThread::Entry()`, we see that `ProcessorThread` avoids any data copy and just wraps the incoming slice of the grabber buffer (and of the processing buffer) into an OpenCV `cv::Mat` object. The conversion is pretty easy using the `cv::Mat` constructor which accepts the buffer size, type and memory address. Note that the `cv::Mat` destructors will not free the memory buffers they wrap, when constructed with the syntax of Listing 4; this allows to reuse the same buffers and, as already mentioned, is very important for optimization reasons. As example, the method `cv::adaptiveThreshold()` is then called on the `source _ img` buffer slice resulting in the update of the `dest _ img` buffer slice. Obviously, in a real application ProcessorThreads will perform some other operation on the source buffer and will extract some kind of useful information (e.g. the number of persons detected in the frame, trajectory of moving cars or, as recent consoles do, the gestures of the user of the software) and eventually post events to the main thread to trigger further actions.

Finally, when all `ProcessorThreads` allocated and run by the `CameraThread` object complete their operations, the processing buffer needs to be rendered on the screen; this is accomplished by `VideoWindow`, which is notified by the CamerThread object through a `wxEVT _ COMMAND _ NEW _ FRAME` event (see Listing 3).

**Listing 5:** *Video stream rendering*

```
class VideoWindow
: public wxPanel

{
  ...
  void DrawFrame(unsigned char* pProcBuffer,
const wxSize procBufferSize);
  CameraThread *m _ pGrabberThread;
};

void VideoWindow::DrawFrame(unsigned char*
pProcBuffer, const wxSize procBufferSize)
{

#if defined( _ _ WXMSW _ _ )

  wxSize viewPort = GetClientSize();
  HDC hDC = GetDC((HWND)GetHandle());
  BITMAPINFO bmpInfo;

  /* ... bmpInfo initialization ... */

  StretchDIBits(hDC,
    /* destination rect */
    0, 0,
    viewPort.x, viewPort.y,
    /* source rect */
    0, 0,
    procBufferSize.x, procBufferSize.y,
    pProcBuffer, bmpInfo, DIB _ RGB _ COLORS,
SRCCOPY);

#elseif defined( _ _ WXGTK _ _ )

  const unsigned int bpp = 4;
  cairo _ surface _ t* surf =
    cairo _ image _ surface _ create _ for _
data(
    pProcBuffer, CAIRO _ FORMAT _ RGB24,
    procBufferSize.x, procBufferSize.y,
width*bpp);

  c a i r o _ t * c r =
    g d k _ c a i r o _create
      (GTKGetDrawingWindow());
  cairo _ set _ source _ surface(cr, surf, 0, 0);
  cairo _ paint(cr);
  cairo _ destroy(cr);

  cairo _ surface _ finish(surf);

#endif

  // tell the grabber thread, which is blocked in
  // that we finished painting the new frame!
  m _ pGrabberThread->RedrawHappened();
}
```

## References

1. Card, S. K., Robertson, G. G., and Mackinlay, J. D. (1991). The information visualizer: An information workspace. Proc. ACM CHI'91 Conf. (New Orleans, LA, 28 April-2 May), 181-188.
2. Miller, R. B. (1968). Response time in man-computer conversational transactions. Proc. AFIPS Fall Joint Computer Conference Vol. 33, 267-277.
3. Myers, B. A. (1985). The importance of percent-done progress indicators for computer-human interfaces. Proc. ACM CHI'85 Conf. (San Francisco, CA, 14-18 April), 11-17.
4. http://msdn.microsoft.com/en-us/library/dd144871(v=vs.85).aspx
5. http://developer.gnome.org/gdk3/stable/gdk3-Cairo-Interaction.html
6. http://docs.wxwidgets.org/trunk/classwx_thread_helper.html

## Presenting the Result to the User: Rendering

In Listing 5 we present a basic implementation of the `VideoWindow` class, whose code, differently from the previously discussed code, is platform-specific and thus may be tricky to write and requires knowledge of the different graphic layer which are usually "hidden" by wxWidgets abstraction layers. The task of `VideoWindow` is to render the buffer processed by the camera thread and render it on the screen canvas. Note that for this reason the buffer held by `VideoWindow` is of the same size of the canvas on screen, which in general will be different from the size of the frame acquired and processed (think to e.g. a canvas which is user-resizable, which can be zoomed in/out, etc).

Let us start to analyze the first block of Listing 5, which is compiled only under Windows (a.k.a. "wxMSW port" in wxWidgets dialect). In the GDI API, i.e. the historical API of Windows for 2-D drawing, the fastest function to transfer the content of one buffer to another of different size is `StretchDIBits()`. In this case the conversion wrapping the processing buffer in a form suitable for use with the native Windows API is easy and just requires filling the various fields of the `BITMAPINFO` structure; in fact, in the GDI language the processing buffer is a Device Indipendent Bitmap (DIB) and thus can be copied on device-dependent buffers (the so-called "device contexts") without any intermediate operation. In this case, the device context where the processing buffer is copied, is the one associated directly with the `VideoWindow HWND`, i.e. with the video memory reserved by the OS for the `VideoWindow` window.

Let us move on the second block, which is compiled only under GTK+ (a.k.a. "wxGTK port"). GTK+ uses Cairo for rendering and thus requires some basic knowledge about the wxWidgets-GTK-GDK-Cairo interactions. In short, on Linux the video hardware is managed by kernel modules (a.k.a. drivers); X11 server (a.k.a. Xorg) connects to the hardware in user space; Cairo interacts with X11 (through Xlib); GDK and GTK use Cairo for rendering the controls and window contents; finally, wxWidgets wraps the GTK API. As users of these libraries and software "ecosystems", we are mostly interested in Cairo functions given that the direct use of X11 API is very difficult and is not recommended by the X11 developers! Moreover, Cairo API is better documented and is easier and safer to use. Looking at Listing 5 we note different names but the same drawing pattern used in the Windows-only block: the processing buffer is wrapped in a cairo_surface_t, which is then selected as the "source surface" for the cairo drawing context associated with the `VideoWindow`. The copy is performed by the `cairo _ paint()` call. Note that also here the destination buffer is directly associated with the memory reserved by the Cairo library for the `VideoWindow` canvas.

## Conclusions

The analysis of the rendering stage concludes our analysis of the key points of the app. In this article, we discussed some guidelines and basic concepts for the development of a cross-platform video-processing application. The listings 3-5 cover some of the key points of such an application and are meant as starting points for more in-depth study of the mentioned libraries.

Indeed, thanks to great open-source libraries like wxWidgets, OpenCV, Boost, etc, cross-platform C++ programming nowadays is easy and very powerful, and the operations critical for performances can be highly optimized. Of course, as mentioned through the paper, when using together heterogeneous code and data structures from different libraries, great care has to be taken in memory management and in designing the entire application in order to avoid at all costs unnecessary memory operations and data processing. Indeed, there is often a steep learning curve involved in this process... so, keep your favorite debugger close at hand and start writing cross-platform, efficient code!

### Francesco Montorsi

Francesco Montorsi is a PhD student in electrical engineering in the University of Modena and Reggio Emilia, Italy.

He has been working for many years in cross-platform open source projects regarding user interfaces, computer algebra, electrical circuit simulation and video processing. He is a developer of wxWidgets project.

He can be contacted at: francesco.montorsi@gmail.com