Lomse library. Tutorial 3 for wxWidgets

In this third tutorial we are going to see how to add score playback capabilities to your application. You can download the full source code for this example from ../../examples/example_3_wxwidgets.cpp.

Table of content

1. How Lomse playback works

Lomse provides class `ScorePlayer`, that takes care of most of the work to do. By using it, playing an score is, basically, three tasks:

1. Load the score to play in the `ScorePlayer` instance.
2. Create a class, derived from `PlayerGui` for controling and defining the playback options (i.e. visual tracking, metronome settings, tempo speed, count-off, etc).
3. Ask `ScorePlayer` to play it, using the desired options.

This is an excerpt of the needed code (more on it later):

```
//step 1: prepare for playing back the score
ImoScore* pScore = ...
ScorePlayer* pPlayer = ...
PlayerGui* pPlayerGui = ...
pPlayer->load_score(pScore, pPlayerGui);

//step2: go on
bool fVisualTracking = true;
m_pPlayer->play(fVisualTracking, 0, spInteractor.get());
```

`PlayerGui` is an interface between your application playback GUI controls and the Lomse library. By "playback GUI controls" I am referring to things such as a button, a menu item, or a link to ask for playing back the score, check boxes for specifying playback options, such as "generate metronome clicks" or "start with count off clicks", a slider for specifying tempo speed, etc. The simplest way for specifying an option (i.e. tempo speed for playing back the score) would be to pass its value directly to Lomse when asking to play the score. But this approach has a drawback: the chosen values can not be changed by the user while the score is being played. Therefore, instead of passing Lomse the values for options, the idea is to have an interface

between Lomse and your application, so that Lomse can ask your application for these values when needed. More on this later.

The last line of previous code is invocation of `play` method. It will trigger all the playback machinery. But **Lomse will not generate any sounds**. Sound generation is always responsibility of your application. What Lomse will do is to generate *sound events*. Therefore, for playback, your application has three main responsibilities:

1. Create a class, derived from `PlayerGui` for defining the playback options.
2. Handle sound events
3. Generate sound for each sound event

Tasks 2 and 3 are accomplished by creating, in your application, a class derived from `MidiServerBase`. This base class defines the interface for processing MIDI events. Your application has to define a derived class and implement the virtual methods. Your class will receive the sound events and will have the responsibility of generating the sounds. Lomse does not impose any restriction about sound generation other than low latency. Perhaps, the simpler method to generate sounds is to rely on the MIDI synthesizer of the PC sound card. But any other alternatives are possible. By the way, I would like to modify Lomse library to add an interface to the JACK audio system (http://jackaudio.org/). If you are interested in taking this task you are welcome. Please post me a message. Thank you!

`ScorePlayer` constructor requires an instance of the `MidiServer` to use:

```
MidiServer* pMidi = new MidiServer();
ScorePlayer* pPlayer = m_lomse->create_score_player(pMidi);
```

With this, all playback machinery is in place.

As to visual tracking effects (highlighting notes being played and/or displaying a vertical tempo line marking beats), Lomse will also post *score highlight* events, to be handled by your application. By handling them you will have full control, and your application can create whatever fancy visual tracking effects you would like.

Nevertheless, Lomse implements standard visual effects so you can delegate in Lomse for generating visual effects. In this case, your application will just receive *update view* events when the display should be updated, and all you have to do is to copy the already updated rendering bitmap onto the window. All the required code for updating the display is already included in code from tutorial 2.

Finally, for controlling playback some GUI controls (buttons, menu items, etc. to trigger start, stop, pause actions) are required. Lomse gives your application two options for this:

1. The first one is to create your own play/pause/stop control mechanism. For instance, you could use menu items, buttons in the toolbar, etc. For linkinf these controls to Lomse you will have to define a class derived from `PlayerGui`. We will see how to do it later in this tutorial.
2. The other alternative is to display, embedded the document, near the score, a *player control* widget (the typical play/stop buttons, plus playback information that we see in many media players). Lomse has such control (class ScorePlayerCtrl) but this is an advanced topic that requires knowledge of Lomse controls and dynamic content creation. So, I will explain nothing about it in this tutorial.

So, as you can see, implementing score playback is simple, and the only burden for your application is coding a `MidiServer` class for generating the sounds.

2. Specifications

In this third tutorial we are going to see how to add score playback capabilities to your application. For controlling playback we are going to add some menu items (play, pause, stop) to the application main menu.

For sound generation we will use the wxMidi (http://www.lenmus.org/en/wxmidi/intro) component. It is just a wrapper for using the PortMidi library (http://sourceforge.net/apps/trac/portmedia/wiki/portmidi) in wxWidgets applications.

In this tuturial on score playback we will have to devote a lot of time to something not directly related to using Lomse: sound generation. Therefore, I will split this tutorial into two. In this tutorial we are not going to generate any visual tracking effects during playback. And we will do it in the fourth tutorial.

At program start we need to present user a dialog for choosing the midi synthesizer to use. We also have to add menu items for displaying again this settings dialog whenever the user likes to change Midi settings.

We are going to use the code from example 2 and add more functionality to it. The changes to do are the following:

1. Definitions for using wxMidi
2. Definition and implementation of class MidiServer, for generating sounds.
3. Initial dialog for Midi settings.
4. Menu items for displaying dialog and for score playback: play, pause, stop.
5. Modifications in `MyFrame` for displaying dialog at start up.
6. Modifications in `MyFrame` for creating `MidiServer` and `ScorePlayer`.
7. Modifications in `MyFrame` for dealing with 'Play', 'Pause' and 'Stop' menu items.
8. Modifications in `MyCanvas` for score playback and for defining the player GUI.

Let's start programming.

3. Changes for using wxMidi

First, download package wxMidi. It is not necessary to install anything, just download the sources. The recommended approach for using wxMidi is to include wxMidi sources in your application source tree. wxMidi is just two .cpp files and one header file. Place them in the same folder than example_3_wx.cpp file. For instance, you could have the following source tree:

```
tutorial-3
    |
    +-- example-3-wxwidgets.cpp
    +-- wxMidi
           |
           +-- wxMidi.cpp
           +-- wxMidiDatabase.cpp
           +-- wxMidi.h
```

Before compiling our example, take into account that it is necessary to install the PortMidi library and link with it. Perhaps this is the right time to ensure that you have PortMidi installed in your system. See wxMidi install instructions, included in the wxMidi package.

Now, let's start modifying code from tutorial 2 by adding `wxMidi` headers and a couple of additional Lomse headers:

```
#include <lomse_player_gui.h>
#include <lomse_score_player.h>

//wxMidi headers
#include <wxMidi.h>
```

4. Definition and implementation of class MidiServer

For using wxMidi we will define class `MidiServer`, derived from `MidiServerBase`- It will implement virtual methods, defined in base class, for creating sounds. Implementation of `MidiServer` is very simple as it is just using the services provided by `wxMidi` class. The definition of `MidiServerBase` class is in header lomse_score_player.h (add now this header to the example code). The class is as follows:

```
class MidiServerBase
{
public:
    MidiServerBase() {}
    virtual ~MidiServerBase() {}

    virtual void program_change(int channel, int instr) {}
    virtual void voice_change(int channel, int instr) {}
    virtual void note_on(int channel, int pitch, int volume) {}
    virtual void note_off(int channel, int pitch, int volume) {}
    virtual void all_sounds_off() {}
};
```

Our derived class is quite simple. We will just define two pointers to keep references to the relevant `wxMidi` objects: `wxMidiSystem`, that represents the whole Midi synthesizer, and `wxMidiOutDevice`, that represents the output device, that is, the device that will create the sounds. I also have defined a couple of variables to contain current Midi configuration, and have added three helper methods. Here is the definition:

```
class MidiServer : public MidiServerBase
{
protected:
    wxMidiSystem*  m_pMidiSystem;       //MIDI system
    wxMidiOutDevice*  m_pMidiOut;       //out device object

    //MIDI configuration information
    int     m_nOutDevId;
    int     m_nVoiceChannel;

public:
    MidiServer();
    ~MidiServer();

    //get number of available Midi devices
    int count_devices();

    //set up configuration
    void set_out_device(int nOutDevId);

    //create some sounds to test Midi
    void test_midi_out();

    //mandatory overrides from MidiServerBase
```

```
    void program_change(int channel, int instr);
    void voice_change(int channel, int instr);
    void note_on(int channel, int pitch, int volume);
    void note_off(int channel, int pitch, int volume);
    void all_sounds_off();
};
```

Implementation is very simple. The constructor is just initializing variables and getting the address of the `wxMidiSystem` object. This object represents the whole Midi interface. This is the code:

```
MidiServer::MidiServer()
    : m_pMidiSystem( wxMidiSystem::GetInstance() )
    , m_pMidiOut(NULL)
    , m_nOutDevId(-1)
    , m_nVoiceChannel(0)    // 0 based. So this is channel 1
{
}
```

The destructor is just closing the out device and deleting the `wxMidi` related objects:

```
MidiServer::~MidiServer()
{
    if (m_pMidiOut)
        m_pMidiOut->Close();

    delete m_pMidiOut;
    delete m_pMidiSystem;
}
```

Apart from the mandatory virtual methods, I have added three new methods. First one, `count_devices` is getting the number of Midi devices installed in your PC. We need it for the Midi settings dialog. This is the code:

```
int MidiServer::count_devices()
{
    return m_pMidiSystem->CountDevices();
}
```

Next one, `set_out_device`, is for choosing the device that will be used for generating sounds. We will use it also in the Midi settings dialog. The basic code is just creating a `wxMidiOutDevice` instance:

```
    m_pMidiOut = new wxMidiOutDevice(m_nOutDevId);
```

But the real code is just slightly more complex due to error checking and to allowing the user for changing Midi settings multiple times. This is the code:

```
void MidiServer::set_out_device(int nOutDevId)
{
    wxMidiError nErr;

    //if out device Id has changed close current device and open the new one
    if (!m_pMidiOut || (m_nOutDevId != nOutDevId))
    {
        //close current device
         if (m_pMidiOut)
         {
            nErr = m_pMidiOut->Close();
            delete m_pMidiOut;
```

```
            m_pMidiOut = NULL;
            if (nErr)
            {
                wxMessageBox( wxString::Format(
                    _T("Error %d while closing Midi device: %s \n")
                    , nErr, m_pMidiSystem->GetErrorText(nErr).c_str() ));
                return;
            }
        }

        //open new one
        m_nOutDevId = nOutDevId;
        if (m_nOutDevId != -1)
        {
            try
            {
                m_pMidiOut = new wxMidiOutDevice(m_nOutDevId);
                nErr = m_pMidiOut->Open(0, NULL);        // 0 latency, no driver user info
            }
            catch(...)      //handle all exceptions
            {
                wxLogMessage(_T("[MidiServer::set_out_device] Crash opening Midi device"));
                return;
            }

            if (nErr)
                wxMessageBox( wxString::Format(
                    _T("Error %d opening Midi device: %s \n")
                    , nErr, m_pMidiSystem->GetErrorText(nErr).c_str() ));
            else
                wxMessageBox(_T("Midi out device correctly set."));
        }
    }
}
```

Finally, I have added a test method for generating some sounds, to be used for checking that Midi settings are valid and the Midi interface works. This is useful for checking the Midi system if score playback doesn't produce sounds. Here is the code:

```
void MidiServer::test_midi_out()
{
    if (!m_pMidiOut) return;

    //Play a scale
    int scale[] = { 60, 62, 64, 65, 67, 69, 71, 72 };
    #define SCALE_SIZE 8

    for (int i = 0; i <SCALE_SIZE; i++)
    {
        m_pMidiOut->NoteOn(m_nVoiceChannel, scale[i], 100);
        ::wxMilliSleep(200);    // wait 200ms
        m_pMidiOut->NoteOff(m_nVoiceChannel, scale[i], 100);
    }
}
```

And now we will implement the required virtual methods. This is the simplest part. Here is the code:

```
void MidiServer::program_change(int channel, int instr)
{
    m_pMidiOut->ProgramChange(channel, instr);
```

```
}

void MidiServer::voice_change(int channel, int instrument)
{
    m_nVoiceChannel = channel;
    if (m_pMidiOut)
    {
        wxMidiError nErr = m_pMidiOut->ProgramChange(channel, instrument);
        if (nErr)
        {
            wxMessageBox( wxString::Format(
                _T("Error %d in ProgramChange:\n%s")
                , nErr, m_pMidiSystem->GetErrorText(nErr).c_str() ));
        }
    }
}

void MidiServer::note_on(int channel, int pitch, int volume)
{
    m_pMidiOut->NoteOn(channel, pitch, volume);
}

void MidiServer::note_off(int channel, int pitch, int volume)
{
    m_pMidiOut->NoteOff(channel, pitch, volume);
}

void MidiServer::all_sounds_off()
{
    m_pMidiOut->AllSoundsOff();
}
```

With this we have finished the longest part: creating our sound generator based on `wxMidi`. Now we have to modify `MyFrame` class for using it. We will modify `MyFrame` definition by adding a pointer to `MidiServer` instance and an accessor method for creating the instance, if not created, and returning it.

First, changes in `MyFrame` definition:

```
class MyFrame: public wxFrame
{
    ...
protected:
    ...
    //sound related
    MidiServer* get_midi_server();

    MidiServer* m_pMidi;
    ...
```

And now the implementation. First, we modify `MyFrame` constructor to initialize `m_pMidi` to `NULL`. and we modify destructor to delete it. Finally, we define the accessor method:

```
MidiServer* MyFrame::get_midi_server()
{
    if (!m_pMidi)
        m_pMidi = new MidiServer();
    return m_pMidi;
}
```

With this, we have all necessary changes for using Midi. The only missing issue is the initial dialog to set Midi device. We will deal with it in next section.

5. Initial dialog for Midi settings

To reduce the size of the example code, instead of coding our own dialog, we will use function `::wxGetSingleChoiceIndex`. It pops up a dialog box containing a message, 'OK' and 'Cancel' buttons and a single-selection listbox. In this listbox the user will select the Midi output device. All other Midi settings will be defaulted to wxMidi default values. In a real application you will have to code a better dialog, including more Midi parameters and other options.

```
void MyFrame::show_midi_settings_dlg()
{
    wxArrayString outDevices;
    vector deviceIndex;

    //get available Midi out devices
    MidiServer* pMidi = get_midi_server();
    int nNumDevices = pMidi->CountDevices();
    for (int i = 0; i <nNumDevices; i++)
    {
        wxMidiOutDevice device(i);
        if (device.IsOutputPort())
        {
            outDevices.Add( device.DeviceName() );
            deviceIndex.push_back(i);
        }
    }

    int iSel = ::wxGetSingleChoiceIndex(
                    _T("Select Midi output device to use:"),    //message
                    _T("Midi settings dlg"),                    //window title
                    outDevices,
                    this                                        //parent window
                );
    if (iSel == -1)
    {
        //the user pressed cancel
        //
    }
    else
    {
        //set current selection
        MidiServer* pMidi = get_midi_server();
        int deviceID = deviceIndex[iSel];   //output device
        pMidi->SetOutDevice(deviceID);
    }
}
```

Now, I will modify `MyFrame` constructor to invoke this method:

```
MyFrame::MyFrame()
    : wxFrame(NULL, wxID_ANY, _T("Lomse sample for wxWidgets"),
              wxDefaultPosition, wxSize(850, 600))
    , m_pMidi(NULL)
{
    create_menu();
    initialize_lomse();
```

```
    show_midi_settings_dlg();
    ...
```

Next, we will add menu items for displaying the Midi settings dialog and for testing the Midi interface. All these new items will be arranged under a 'Sound' submenu. This is usual wxWidgets stuff. We have to define IDs for these new menu items (`k_menu_midi_test` and `k_menu_midi_settings`). After this, we modify our our `create_menu()` method as follows:

```
    ...
    wxMenu *soundMenu = new wxMenu;
    soundMenu->Append(k_menu_midi_settings, _T("&Midi settings"));
    soundMenu->Append(k_menu_midi_test, _T("Midi &test"));
    ...
    menuBar->Append(soundMenu, _T("&Sound"));
```

Now, we have to modify MyFrame class to add handlers for the new menu events. Remember that we also have to modify the events table:

```
    EVT_MENU(k_menu_midi_settings, MyFrame::on_midi_settings)
    EVT_MENU(k_menu_midi_test, MyFrame::on_sound_test)
```

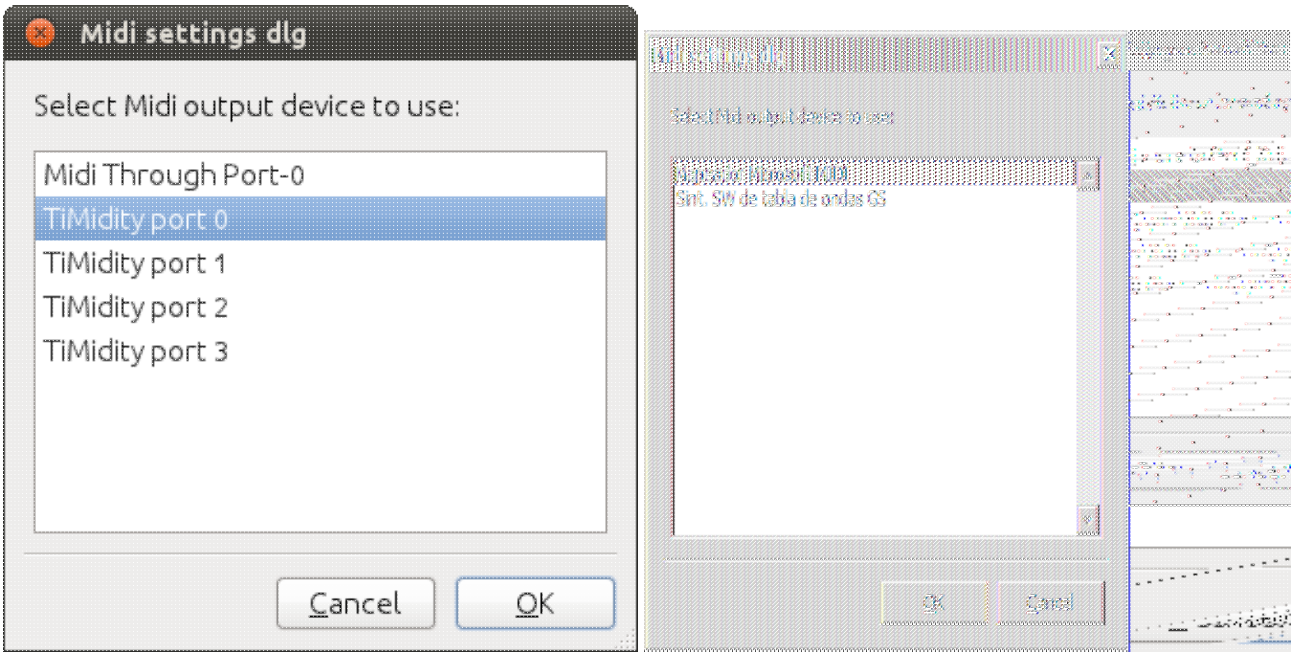
Finally, we code these two methods:

```
void MyFrame::on_midi_settings(wxCommandEvent& WXUNUSED(event))
{
    show_midi_settings_dlg();
}

void MyFrame::on_sound_test(wxCommandEvent& WXUNUSED(event))
{
    MidiServer* pMidi = get_midi_server();
    if (!pMidi) return;
    pMidi->test_midi_out();
}
```

6. Testing sound

Now, compile and build the example code (see section Compiling your code and building). It is not yet finished but we can test our Midi interface. When running the program the first thing you will see is the Midi settings dialog, a window similar to one of these:



In these images, you can see the Midi devices created by Timidity software in my Linux system and the Midi devices in my Windows XP system. In Windows, choose any of the devices. In Linux choose one of the Timidity ports 0 to 4, but not the midi through port!

After choosing the desired device and clicking on button 'Ok' you will get the main frame, displaying an score. In main menu, you will get a new item 'Sound'. Click on it and select item 'Test Midi'. If everything is ok you should hear an scale. If not, check the sound settings of your computer. The most frequent cause for not hearing any sound is that the Midi synthesizer output is disabled! If you are running the tutorial on Linux see "Problems with MIDI sound in Linux".

Once the Midi test is passed, we are ready to continue with more changes in our program.

7. Additional menu items

Now we will add menu items for score playback: 'Play', 'Pause' and 'Stop'. All these new items will be arranged under the 'Sound' submenu. This is usual wxWidgets stuff. We have to define IDs for these new menu items (`k_menu_play_start`, `k_menu_play_stop` and `k_menu_play_pause`). After this, we modify our our `create_menu()` method as follows:

```
wxMenu *soundMenu = new wxMenu;
soundMenu->Append(k_menu_play_start, _T("&Play"));
soundMenu->Append(k_menu_play_stop, _T("&Stop"));
soundMenu->Append(k_menu_play_pause, _T("Pause/&Resume"));
soundMenu->Append(k_menu_midi_settings, _T("&Midi settings"));
soundMenu->Append(k_menu_midi_test, _T("Midi &test"));
...
menuBar->Append(soundMenu, _T("&Sound"));
```

Now, we have to modify MyFrame class to add handlers for the new menu events. Remember that we also have to modify the events table:

```
EVT_MENU(k_menu_play_start, MyFrame::on_play_start)
EVT_MENU(k_menu_play_stop, MyFrame::on_play_stop)
```

```
    EVT_MENU(k_menu_play_pause, MyFrame::on_play_pause)
```

The methods for handling these events are quite simple as we just delegate on `MyCanvas`:

```
void MyFrame::on_play_start(wxCommandEvent& WXUNUSED(event))
{
    get_active_canvas()->play_start();
}

void MyFrame::on_play_stop(wxCommandEvent& WXUNUSED(event))
{
    get_active_canvas()->play_stop();
}

void MyFrame::on_play_pause(wxCommandEvent& WXUNUSED(event))
{
    get_active_canvas()->play_pause();
}
```

8. Creating ScorePlayer object

Before coding the necessary changes in `MyCanvas` it is necessary to create the `ScorePlayer` object. It should be a global object, accessible from any point at which it is necessary to play an score. In our simple example, I will put this object as member of `MyFrame` and will pass it to `MyCanvas` in the constructor. Therefore, I will change `MyFrame` declaration to define a member variable and a getter method:

```
class MyFrame: public wxFrame
{
    ...
protected:
    ...
    //sound related
    ...
    ScorePlayer* get_score_player();

    ScorePlayer* m_pPlayer;
    ...
```

In `MyFrame` constructor the new variable `m_pPlayer` is initialized to `NULL`, and the `ScorePlayer` creation is delayed until really needed:

```
ScorePlayer* MyFrame::get_score_player()
{
    if (!m_pPlayer)
    {
        MidiServer* pMidi = get_midi_server();
        m_pPlayer = m_lomse.create_score_player(pMidi);
    }
    return m_pPlayer;
}
```

Finally, we modify `MyCanvas` definition for receiving the `ScorePlayer` object in constructor:

```
class MyCanvas : public wxWindow
{
public:
    MyCanvas(wxFrame *frame, LomseDoorway& lomse, ScorePlayer* pPlayer);
    ...
```

```
protected:
    ...
    // for score playback
    ScorePlayer* m_pPlayer;


MyCanvas::MyCanvas(wxFrame *frame, LomseDoorway& lomse, ScorePlayer* pPlayer)
    : wxWindow(frame, wxID_ANY)
    , m_lomse(lomse)
    , m_pPresenter(NULL)
    , m_buffer(NULL)
    , m_pPlayer(pPlayer)
    , m_view_needs_redraw(true)
{
}
```

And we modify the code for passing the `ScorePlayer` object when `MyCanvas` instances are created, in `MyFrame` constructor:

```
MyFrame::MyFrame()
    : wxFrame(NULL, wxID_ANY, _T("Lomse sample for wxWidgets"),
              wxDefaultPosition, wxSize(850, 600))
    , m_pMidi(NULL)
    , m_pPlayer(NULL)
{
    ...
    // create our one and only child -- it will take our entire client area
    m_canvas = new MyCanvas(this, m_lomse, get_score_player());
    ...
```

With this, we have finished with the modifications required for `MyFrame`.

9. Creating the player GUI object

I have defined some items for score playback: 'Play', 'Pause' and 'Stop', in the 'Sound' submenu. But, as I would like to keep this tutorial simple, I have not created other GUI widgets for controlling playback. In a real application you, probably, would like to have additional feautures, such as checkboxes for deciding wether to have metronome clicks enabled, to produce some coount off metronome clicks, to have volume and tempo sliders, etc.

To inform Lomse about these controls and about the options fof playback you have to define a class, derived from `PlayerGui`, and implement its virtual methods. Let's see the definition of class `PlayerGui`:

```
class PlayerGui
{
protected:
    PlayerGui() {}

public:
    virtual ~PlayerGui() {}

    //mandatory overrides
    virtual void on_end_of_playback() = 0;
    virtual int get_play_mode() = 0;
    virtual int get_metronome_mm() = 0;
    virtual Metronome* get_metronome() = 0;
    virtual bool countoff_status() = 0;
    virtual bool metronome_status() = 0;
```

```
};
```

As you can see, it is just a few methods that will be invoked by Lomse to get the desired playback options. Let's explain the purpose of each method:

`metronome_status()` must return a boolean to indicate if the metronome is swicthed on or off. This method will be used by Lomse to determine if metronome clicks should be generated. As user can turn metronome on and off while the playback is taken place, this method will be invoked many times by Lomse during playback for pulling metronome state. For coding this method you could, for instance, access your application GUI control (i.e. the checkbox controlling the metronome status) and return its value. Or could always return `false` if your application doesn't require metronome sounds.

`countoff_status()` must return a bolean to indicate if playback will start with some metronome clicks for count off or not. This method will be invoked by Lomse only once, just before starting playback. Again, in your application you can code this method as it best suits your needs: return a fixed value, true o false; accsess a GUI control and return its status; or any other solution you would like.

`get_metronome_mm()` must return the tempo speed (in beats per minute) you would like for playback. Again, as tempo can be changed by the user while playback is taking place, this method will be invoked constantly by Lomse during playback.

`get_play_mode()` must return one of the following values defined in `lomse_player_gui.h` header file:

```
enum
{
    k_play_normal_instrument = 0,   //normal, (pitched instrument)
    k_play_rhythm_instrument,       //only rhythm (instrument, single pitch)
    k_play_rhythm_percussion,       //only rhythm (percussion, single pitch)
    k_play_rhythm_human_voice,      //only rhythm (solfege, human voice)
};
```

The returned value will be, for now, ignored by Lomse as only normal playback using a musical instrument is currently implemented.

`on_end_of_playback()` is a method that will be invoked by Lomse when playback has finished, either because the last note has been played or because the user stopped the playback. In this method your application should do whatever you need, and it is mainly intended for ensuring GUI coherence with playback status. For instance, imagine an application having a "Play" button for starting playback. When the user clicks on it, the application will render the button as 'pushed' and invokes the Lomse `play` method. The "Play" button will remain pushed forever and the application could use the `on_end_of_playback()` for render again the button as 'released'.

I will not comment on the purpose of method `Metronome* get_metronome()` as it is intended for more advanced uses. So, for now, just code a method returning `NULL`.

Now that we understand the purpose of `PlayerGui` interface it is time for deciding how to do it in our sample application. As I wouldn't like to create a complex tutorial, the only GUI controls for controlling playback are the menu items 'Play', 'Pause' and 'Stop', in the 'Sound' submenu. Therefore, there are no ways for the user to decide on options. Instead, the application has a fized set of values that the user can not change. Let's decide the following values for the options:

- Always play with metronome clicks enabled.
- Always start playback with count off clicks.
- Tempo speed will always be 60 beats per minute.

Having decided this, it would be simple to code the required methods as it is just returning a fixed value. But for application using fixed values, such as this sample, Lomse provides a derived class `PlayerNoGui` that receives the desired fixed values in the constructor and implements the required methods.

A last important issue: the PlayerGui object need to be available while score is beign played. Therefore, it can not be a local object defined in the method invoking the Lomse `play`. In this sample, we will just derive `MyCanvas` from `PlayerNoGui`. So, lets start with the modifications:

Firts, the definition of `MyCanvas` class, for deriving from `PlayerNoGui`:

```
class MyCanvas : public wxWindow, public PlayerNoGui
```

Next, in the implementation we add a line to initialize `PlayerNoGui`:

```
MyCanvas::MyCanvas(wxFrame *frame, LomseDoorway& lomse, ScorePlayer* pPlayer)
    : wxWindow(frame, wxID_ANY)
    , PlayerNoGui(60L /*tempo 60 MM*/, true /*count off*/, true /*metronome clicks*/)
    , m_lomse(lomse)
        , m_pPresenter(NULL)
        , m_buffer(NULL)
        , m_pPlayer(pPlayer)
        , m_view_needs_redraw(true)
{
}
```

With these changes, our the player GUI object is created and initialized.

10. Other modifications in MyCanvas

In `MyFrame` we delegated all playback operations on `MyCanvas`. Therefore, we have now to code these new methods. The first one `play_start` has to load the score in the `ScorePlayer` object (received in constructor) and invoke its `play` method, as was explained in section "How lomse playback works".

The main doubt you should have is how to get the score to play. In these tutorials I never explained how access parts of a document, such as a paragraph or a score. In a future tutorial I will explain this in detail but, for now, you should learn that `Document` class has member methods for traversing the document and accessing its components. One of these methods is `get_content_item(int index)` that takes as argument the index to the desired content item. Index 0 is the first item in the document, index 1 the second one, and so on. Therefore, as we know that the document only contains one score, getting the score to play is just getting the document and invoking `get_content_item(0)` method:

```
    Document* pDoc = m_pPresenter->get_document_raw_ptr();
    ImoScore* pScore = dynamic_cast( pDoc->get_imodoc()->get_content_item(0) );
```

With this knowledge we can now code the `play_start` method:

```
void MyCanvas::play_start()
{
    if (SpInteractor spInteractor = m_pPresenter->get_interactor(0).lock())
```

```
    {
        Document* pDoc = m_pPresenter->get_document_raw_ptr();

        //AWARE: Then next line of code is just an example, in which it is assumed that
        //the score to play is the first element in the document.
        //In a real application, as the document could contain texts, images and many
        //scores, you shoud get the pointer to the score to play in a suitable way.
        ImoScore* pScore = static_cast( pDoc->get_imodoc()->get_content_item(0) );

        if (pScore)
        {
            m_pPlayer->load_score(pScore, this);
            m_pPlayer->play(k_do_visual_tracking, 0, spInteractor.get());
        }
    }
}
```

The other required methods, `play_stop` and `play_pause`, are trivial:

```
void MyCanvas::play_stop()
{
    m_pPlayer->stop();
}

void MyCanvas::play_pause()
{
    m_pPlayer->pause();
}
```

And this is all. Our code is now finished and our application is ready for building and testing. But before doing it I will change code in `open_test_score` to put an score more interesting for playback. You can download the full source code for this example from ../../examples/example_3_wxwidgets.cpp.

11. Compiling your code and building

At this point, we could compile and tests all these changes. Do not forget to install the PortMidi libraries. For detailed instructions you should see the PortMidi documentation. Nevertheless, on most Linux systems it is just installing the PortMidi package. For instance, execute the following command on a terminal window (enter your password as needed):

```
sudo apt-get install libportmidi-dev
```

Headers will be placed at /usr/include/ and libraries at /usr/lib/.

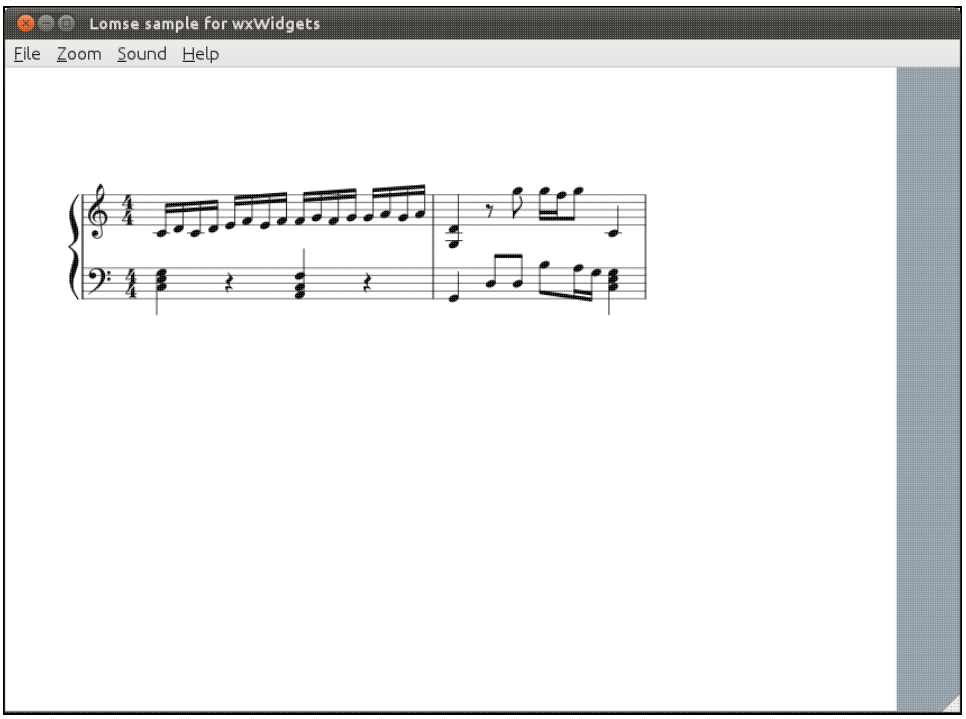
Once PortMidi libraries are installed, you can proceed to build your application. For building, you can use the makefile or build command used for tutorials 1 or 2. But you will have to modify it as follows:

- You have to add source files from wxMidi package.
- You have to add search directories to look for wxMidi and PortMidi headers. For instance, in Linux you will have to add '/usr/include/ and './wxMidi/'.
- You will have to link with PortMidi libraries. For instance, in Linux you will have to add libportmidi.so and libporttime.so.
- And you will have to add the paths for these libraries. For instance, /usr/lib/

With these changes, your build command in Linux will be something as (adjust paths if necessary):

```
gcc example_3_wxwidgets.cpp ./wxMidi/wxMidi.cpp ./wxMidi/wxMidiDatabase.cpp -o example-3-wx \
    `wx-config --cflags` `pkg-config --cflags liblomse`  -I ./wxMidi/    \
    `pkg-config --libs liblomse` -lstdc++ -lportmidi -lporttime `wx-config --libs`
```

Run the program. You should see something as:



Click on the 'Sound > Midi test' menu item, and check that sounds are produced. When executing example-3 and playing an score you will hear the music but you will not see any visual effects. We will learn in next tutorial how to add visual tracking effects. In Windows systems there should not be any problems with sound but in many Linux distros a MIDI synthesizer is not installed by default. If you have problems with sound, please read next section.

12. Problems with MIDI sound in Linux

When testing the code in Linux you might find problems with sound as a MIDI synthesizer is not installed by default in some Linux distros. If this is your case have to install a Midi synthesizer. Here you have some references for this:

https://help.ubuntu.com/community/Midi/SoftwareSynthesisHowTo
https://help.ubuntu.com/community/Midi/HardwareSynthesisSetup
http://timidity.sourceforge.net/#info

13. Conclusions

In this long tutorial I have introduced score playback. Unfortunately, most of the tutorial is not directly related to using the Lomse library but to generating sounds. In the fourth tutorial I will continue with this subject and we will study how to add visual tracking effects during playback.

If you would like to contribute with more tutorials or by adapting this tutorial for other platforms, you are welcome!. Join the Lomse list and post me a message.