Lomse library. Tutorial 1 for MS Windows

This is meant to be an introduction to using Lomse in a Windows program. Before starting, ensure that you have installed the Lomse library. See the installation page for detailed instructions.

In this first example we are just going to open a window and display the text "Hello world!" and an score on it. You can download the full source code for this example from ../../examples/example_1_win32.cpp. After building the program and running it you will see something as:

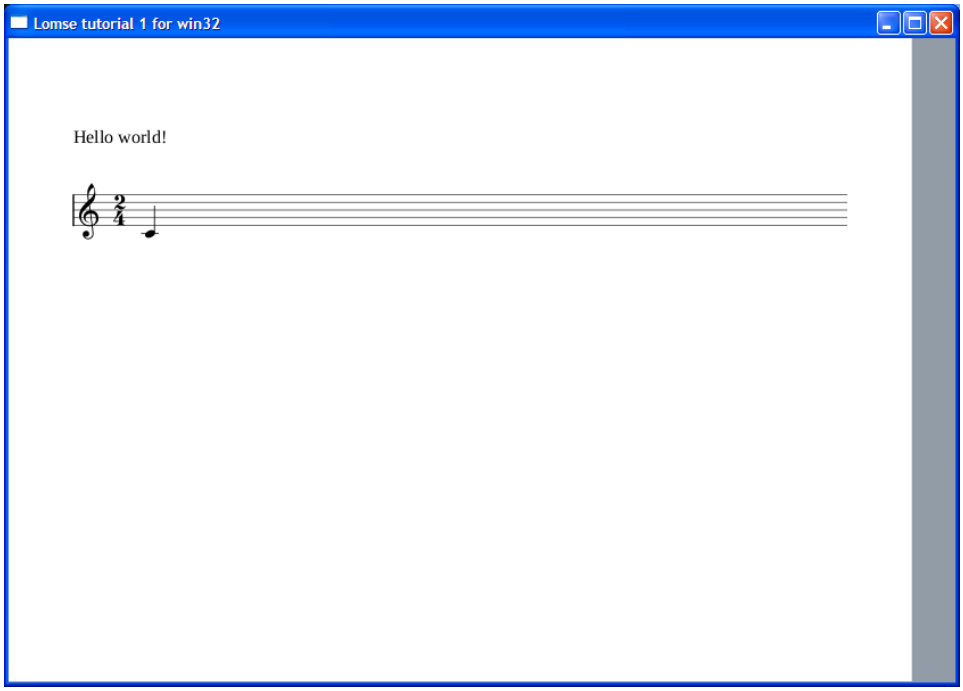


Table of content

1. How does Lomse work
2. Specifications: displaying a score
3. Header files to use Lomse
4. Helper class to create and manage bitmaps
5. Important variables
6. The application: main function
7. Initializing the Lomse library
8. Creating the score and the View
9. Creating the main window and the bitmap for the rendering buffer
10. Painting the window
11. Closing the application
12. Compiling your code and building
13. Conclusions

1. How does Lomse work

In this first example we are just going to open a window and display an score on it. The first and most

important thing to learn about Lomse is that is platform independent code, with no knowledge about Microsoft Windows or CWindow classes. Therefore, **Lomse can not directly render scores on any window object.**

Lomse works by rendering the scores on a bitmap buffer, that is, on an array of consecutive memory bytes. As this buffer is provided by the user application, it can be any type of memory, such as a real bitmap, a window's buffer, etc. This implies that before using Lomse you should decide what is the best approach for using Lomse in your application.

The simplest and usual way of rendering scores on a window is just passing Lomse a bitmap in memory, asking Lomse to render on that bitmap, and copying the bitmap onto the window. And this is the approach we will follow for our MS Windows application.

2. Specifications: displaying a score

In this first example we are just going to open a window and display an score on it. For displaying a score the work to do is minimal:

1. Initialize the Lomse library,
2. Pass Lomse the source code for the score to render and a bitmap. Lomse will render the score on the bitmap.
3. Finally, open a window and display the bitmap on it

In this example, the source code for the score is embedded in the code. In a real program you normally will read a file containing the score to display or you will create, by program, the source code for the score. We will do that in a more advanced tutorial.

With previous specifications, the structure of our program will be very simple. The application will be modeled by class MyApp, derived from wxApp. When the app starts running it will create the main frame (class MyFrame, derived from wxFrame) which, in turn, will create a window for displaying the score (class MyCanvas). For it, we will pass Lomse a bitmap, and will ask Lomse to create the score. Then we will render the bitmap on the window. And that's all.

Let's start programming.

3. Header files to use Lomse

Before we get into the heart of our basic example, we will include the needed headers. After the usual stuff and headers for MS Windows, I've included the Lomse needed headers. At the time of writing this the Lomse API is not yet fixed; therefore there is not a single header file (or set of headers) to include. Instead, the headers to include will depend on the classes and functions you would like to use. Anyway, with current API you will always include:

```
#include <lomse_doorway.h>
#include <lomse_document.h>
#include <lomse_graphic_view.h>
#include <lomse_interactor.h>
#include <lomse_presenter.h>
#include <lomse_events.h>

using namespace Lomse;
```

`LomseDoorway` is the main interface with the Lomse library. `Document` represents the score to display and is part of the Lomse Model-View-Controller (MVC) architecture. `GraphicView` is a kind of `View` (the window in which the score is going to be displayed). `Interactor` is the controller for the `View`. `Presenter` is also part of the MVC model, and is responsible for maintaining the relationships between a `Document` and its different views and associated interactors. Finally, `lomse_events.h` is required to deal with events received from the Lomse library.

4. Helper class to create and manage bitmaps

As Lomse works by rendering the scores on a bitmap buffer, there are two tasks your application have to do:

1. Create a new empty bitmap when necessary, and
2. Drawing the bitmap on your window

For performing these tasks I opted for creating a `Bitmap` class, enclosing the necessary methods and knowledge. Although the Windows API provides many functions for creating and managing bitmaps, my knowledge of managing bitmaps using the Windows API is null. So to write this tutorial I opted to borrow code from the AGG project, instead of finding documentation and studying how to use the Windows API functions. If you have good knowledge of the Windows API probably you would prefer a different solution for managing bitmaps. In that case, I would appreciate if you could help me to improve this tutorial by sharing your cleaner code. Thank you.

So, after the Lomse headers I have declared the auxiliary `Bitmap` class for creating and managing bitmaps. The code is borrowed from AGG project, but I simplified the class removing all methods not needed for using Lomse.

Next, after `Bitmap` class declaration, our real tutorial example starts.

5. Important variables

In this first example we are just going to display an score on the main window. For this, we need to define some Lomse related variables:

```
LomseDoorway    m_lomse;        //the Lomse library doorway
Presenter*      m_pPresenter;   //relates the View, the Document and the Interactor
Interactor*     m_pInteractor;  //to interact with the View
```

`m_lomse` is an important variable as it is the main interface with the Lomse library. As we will see later, we have to use it for specifying certain Lomse initialization options. The two other variables, `m_pPresenter` and `m_pInteractor` are pointers to two important components of the Lomse Model-View-Controller (MVC) architecture. The `Interactor` is a kind of controller for the view. And the `Presenter` is responsible for maintaining the relationships between a Document and its different Views and associated interactors. Later, we will learn more about them.

Next we are going to declare a rendering buffer and its associated bitmap:

```
RenderingBuffer     m_rbuf_window;
Bitmap              m_bitmap;
```

As you know, Lomse knows nothing about MS Windows, so the Lomse View renders the music scores on a bitmap. To manage the bitmap, Lomse associates it to a `RenderingBuffer` object. As Lomse only renders

on bitmaps, it is your application responsibility to do whatever is needed with it: rendering it on a window, exporting it as a file, printing it, etc. In our simple application, we are going to render the bitmap on the application main window.

There are some more variables defined but we will see them later, when having to use them. After the variables, I've also forward declared the main function. With this, we have finished the declarations. Lets's now move to the implementation.

6. The application: main function

Let's move to near line 690 for looking at the `WinMain` function. It is the main entry point and it is very simple:

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
    register_window_class(hInstance);
    initialize_lomse();

    //create a music score and a View. The view will display the score
    //when a paint event is received, once the main windows is
    //shown and the event handling loop is started
    open_test_document();

    //initialize and show main window
    if (!create_main_window(nCmdShow))
        return FALSE;

    //enter the main event handling loop
    int retcode = handle_events();

    //terminate the application
    free_resources();
    return retcode;
}
```

As you can see the process is quite simple:

1. Usual MS Windows stuff: registering the class
2. Initialize the Lomse library. It is necessary to do it before using Lomse.
3. Create an score and a Lomse View to display it.
4. Create the main window, and the rendering buffer for Lomse, connecting this window to the Lomse rendering buffer.
5. Finally, run the main events handling loop, for processing events, until the user request to close the application. On of the firsts events will be a 'paint' event. And the event handler will display the Lomse bitmap on the main window. That's all!

Let's see in detail these steps.

7. Initializing the Lomse library

The first interesting line in `WinProc` function is the initialization of the Lomse library. As Lomse renders music scores on a bitmap it is necessary to inform Lomse about the bitmap format to use, the resolution and the y-axis orientation.

So, the first thing to do is to decide which bitmap format we are going to use. For MS Windows applications you can use, for instance, pixel format BGRA, 32 bits per pixel, as this is one of the native formats for MS Windows.

Next, we have to decide about resolution. As in our application the scores are going to be shown on screen, we can use a value of 96ppi, typical for MS Windows systems. In a real application, probably you should get this value by invoking some operating system related method to get the screen resolution.

As to the y-axis orientation, Lomse needs to know if your presentation device follows the standard convention used in screen displays in which the y coordinates increases downwards, that is, y-axis coordinate 0 is at top of screen and increases downwards to bottom of screen. This convention is just the opposite of the normal convention for geometry, in which 0 coordinate is at bottom of paper and increases upwards. Lomse follows the standard convention used in displays (y-axis 0 coordinate at top and increases downwards). Therefore, in our application, we have to inform Lomse that the y-axis follows the standard convention for screens and, therefore, we won't Lomse to reverse it.

One we have decided on the values to use, let's see the code to initialize Lomse:

```
void initialize_lomse()
{
    // Lomse knows nothing about windows. It renders everything on a bitmap and the
    // user application uses this bitmap. For instance, to display it on a window.
    // Lomse supports a lot of bitmap formats and pixel formats. Therefore, before
    // using the Lomse library you MUST specify which bitmap formap to use.
    //
    // For native MS Windows applications you can use, for instance, pixel format
    // BGRA, 32 bits. Screen resolution, in MS Windows, is 96 pixels per inch.
    // Let's define the requiered information:

        //the pixel format
        int pixel_format = k_pix_format_bgra32;  //BGRA, 32 bits
        m_bpp = 32;                              //32 bits per pixel

        //the desired resolution. For MS Windows use 96 pixels per inch
        int resolution = 96;    //96 ppi

        //Lomse default y axis direction is 0 coordinate at top and increases
        //downwards. You must specify if you would like just the opposite behaviour.
        //For MS Windows the Lomse default behaviour is the right behaviour.
        bool reverse_y_axis = false;

    //initialize the Lomse library with these values
    m_lomse.init_library(pixel_format, resolution, reverse_y_axis);
}
```

8. Creating the score and the View

After initializing the Lomse library we can use it. The next line we find in `WinProc` function is a call to `open_test_document();`. This function is the equivalent for the typical `open_document` method in which your application opens a dialog for requesting the file to open, and then, processes and displays it. In our example, the score is in a string, so the only thing to do is to request Lomse to create a new document with the specified content. When creating a document, Lomse automatically, creates a View to display it and an Interactor (a kind of Controller for the View). The `open_test_document()` method is as follows:

```
void open_test_document()
```

```
{
    //Normally you will load the content of a file. But in this simple example we
    //will create an empty document and define its content from a text string

    //first, we will create a 'presenter'. It takes care of creating and maintaining
    //all objects and relationships between the document, its views and the interactors
    //to interact with the view
    delete m_pPresenter;
    m_pPresenter = m_lomse.new_document(ViewFactory::k_view_vertical_book,
        "(lenmusdoc (vers 0.0)"
            "(content "
                "(para (txt \"Hello world!\"))"
                "(score (vers 1.6) "
                    "(instrument (musicData (clef G)(key C)(time 2 4)(n c4 q) )))"
            ")"
        ")" );

    //now, get the pointers to the relevant components
    m_pInteractor = m_pPresenter->get_interactor(0);

    //connect the View with the window buffer
    m_pInteractor->set_rendering_buffer(&m_rbuf_window);
}
```

The `Presenter` is the key object that relates a Document with its Views and Interactors. Also is the access point to get pointers to the Document and its Interactors. Deleting the Presenter also deletes all other related objects.

For creating the Presenter (and associated objects) we invoke LomseDoorway method `new_document()`, passing as arguments, the type of View to create and the content for the document (note: there are other methods, oriented to create the View from a file or programatically, but we will not study them in this simple example).

The View type is just a Lomse enum. In this example, value `ViewFactory::k_view_vertical_book` means that we would like to display the score as book pages, one page after the other in a vertical layout. Other View formats are possible out-of-the-box, such as horizontal book or not paginated (the score in a single system) but, in any case, its not complex to develop your own View format.

The next parameter is a C string containing the score. It is written in LenMus LDP language. Let's split it into lines, and let's analyse it:

```
(lenmusdoc (vers 0.0)
    (content
        (para (txt "Hello world!"))
        (score (vers 1.6)
            (instrument
                (musicData
                    (clef G)
                    (key C)
                    (time 2 4)
                    (n c4 q)
                )
            )
        )
    )
)
```

First line means that it is a LenMus document, with version 0.0 format. Next line describes the content of the document. The content is just two elements: a paragraph ('para' element) containing text "Hello world!" and a 'score' element. Other types of content are possible: headers, images, tables, lists, etc. You can see LenMus documents as HTML documents, but allowing also a new type of content: scores.

The score element contains one instrument (this implies, by default, one staff). Finally, element 'musicData' describes the content for this instrument. In the example, a G clef, a C key signature, a 2/4 time signature and a quarter C4 note.

For a detailed description of the LDP language see the LDP Reference Manual. I have plans for supporting scores in MusicXML format, as well as to move the LDP language to XML syntax. But other more urgent task force me to always postpone these objectives. Any help is welcome!

Once the Document and a View for it are created, we just get pointers to the Interactor, so that we can 'communicate' with the Document and its View:

```
    //next, get the pointers to the relevant components
    m_pInteractor = m_pPresenter->get_interactor(0);
```

Lomse architecture is based on the Model-View-Controller pattern, and supports multiple simultaneous Views for a Document. By default, when creating a Document also a View and its associated `Interactor` are created. So, parameter `'0'` in `get_interactor(0)` refers to first `Interactor`, in this case, the only one created.

Once we've got the Interactor we have one **important** task to do. It is to inform the Interactor about the rendering buffer that must be used for its associated View:

```
    //connect the View with the window buffer
    m_pInteractor->set_rendering_buffer(&m_rbuf_window);
```

In the previous line, we pass to the interactor the address of the rendering buffer but, we have not yet created any bitmap. Don't worry, the bitmap will not be used until we ask Lomse to render something, so we can delay its creation until really needed. We will see later hao the bitmap is created.

9. Creating the main window and the bitmap for the rendering buffer

The next step, in `WinProc` function, is to create the main window. The code is as follows:

```
BOOL create_main_window(int nCmdShow)
{
    m_hWnd = CreateWindow( "Lomse_Example1"                 //class name
                         , "Lomse tutorial 1 for win32"     //window caption
                         , WS_OVERLAPPEDWINDOW              //wflags
                         , CW_USEDEFAULT                    //pos-x
                         , 0                                //pos-y
                         , 840                              //width
                         , 600                              //height
                         , NULL                             //parent Window
                         , NULL                             //menu, or windows id if child
                         , m_hInst
                         , NULL                             //ptr to window specific data
            );
```

```
    if (!m_hWnd)
        return FALSE;

    //display the window
    ShowWindow(m_hWnd, nCmdShow);
    UpdateWindow(m_hWnd);
    return TRUE;
}
```

Creating the main window and displaying it is typical MS Windows stuff, so there is nothing to comment unless you are a beginer. In this case, please notice that functions `ShowWindow` and `UpdateWindow` don't display anything. They just generate events (i.e. window resize, window paint) that we will have to process at due course.

After creating the main window `WinMain()` enters in the main loop for handling events (function `handle_events`) and remains in that loop until a 'quit application' event arrives. This loop is the typical MS Windows stuff and when an event is received, the `DispatchEvent` function sends it to our `WndProc` function.

As a consequence of having created the main window, a `WM_SIZE` event arrives to our `WndProc` function. Here is the code for handling it:

```
case WM_SIZE:
    create_bitmap_for_the_rendering_buffer(LOWORD(lParam), HIWORD(lParam));
    break;
```

The only thing we have to do is to create the bitmap for lomse. We couldn't create it before as we didn't know the window size. And this is the right place for creating it, as we will have to create a new bitmap whenever the window size changes, and it is the place at which we are informed about the new window size. The code for creating the rendering buffer is as follows:

```
void create_bitmap_for_the_rendering_buffer(unsigned width, unsigned height)
{
    //creates a bitmap of specified size and associates it to the rendering
    //buffer for the view. Any existing buffer is automatically deleted

    m_bitmap.create(width, height, m_bpp);
    m_rbuf_window.attach(m_bitmap.buf(),
                         m_bitmap.width(),
                         m_bitmap.height(),
                         -m_bitmap.stride()
                        );
    m_view_needs_redraw = true;
}
```

The code is simple: we create a new bitmap, ask the rendering buffer to use it, raise a flag to signal that the bitmap is empty, that is, that Lomse has to paint something on it before displaying the bitmap on the window. Painting the bitmap takes place when a `WM_PAINT` event arrives to our `WndProc` function. We will see this in next section.

10. Painting the window

For handling a paint event we have to do two things: 1) to ask Lomse to render the score in the bitmap, and 2) to display the bitmap in the window. Here is the code:

```
case WM_PAINT:
{
    update_rendering_buffer_if_needed();

    PAINTSTRUCT ps;
    HDC paintDC = ::BeginPaint(m_hWnd, &ps);
    display_view_content(paintDC);
    ::EndPaint(m_hWnd, &ps);
    break;
}
```

Firts, notice that there is no need to ask Lomse to paint the bitmap whenever a paint event arrives. These events are generated because several reasons. The most frequent is when our window image is damaged (i.e. another window covering our window has moved). But in these cases the image is preserved in the bitmap so it is enough to re-display the bitmap. Other cases for receiving paint events are because the window has changed: when the window is created or when it is resized or when our program changes its content (i.e. because the if the user asks to open a different score). In our application this last case is not possible and so, the only source for additional paint event comes from size events. And we have a flag to signal when the bitmap needs to be repainted by Lomse. Therefore, the logic for updating the rendering buffer is simple: if flag `m_view_needs_redraw` is raised, ask Lomse to render the document on the bitmap. Here is the code:

```
void update_rendering_buffer_if_needed()
{
    //request the view to re-draw the bitmap

    if (!m_pInteractor) return;

    if (m_view_needs_redraw)
        m_pInteractor->force_redraw();
    m_view_needs_redraw = false;
}
```

Once we have ensured that the bitmap has the right content, we create a DC and copy the bitmap on it:

```
void copy_buffer_on_dc(HDC dc)
{
    m_bitmap.draw(dc);
}
```

An that's all. Our sample score should be now visible on the user screen.

11. Closing the application

Finally, the last important point to comment is to remind you that, to avoid memory leaks, it is necessary to delete the rendering buffer and the Presenter (which in turn will delete all Lomse related objects, such as the View, the Document and the Interactor). This code has been included in the last line of our `WinMain` function:

```
void free_resources()
{
    //delete the Presenter.
    //This will also delete the Interactor, the Document and the View
    delete m_pPresenter;
}
```

With this, I finish the explanation of the code. You can download the full source code for this example from ../../examples/example_1_win32.cpp. In the next section we are going to build and run our sample.

12. Compiling your code and building

Now the sometimes troubling part: compiling your code and running it! Your proyect makefile must include the path for Lomse header files. For instance:

```
C:\Program Files\LenMus\lomse\include\
```

Some Lomse headers include references to Boost and FreeType2 libraries. Therefore, it is necessary to include the headers from these libraries. For me, guessing were things are installed in Windows is a nightmare. Therefore, replace the paths I use here for the proper ones:

```
C:\Program Files\boost\
C:\Program Files\freetype\include
C:\Program Files\freetype\include\freetype2
```

As to the libraries to link, you will have to include Lomse and the required support libraries. Take into account that appart from any library required by your application, the Lomse library needs to be linked with some libraries (boost, zlib, libpng and freetype2). In summary, you need to link with the following libraries:

- libboost_date_time-vc71-mt-sgd-1_42.lib (or later)
- zlib.lib
- libpng.lib
- freetype.lib

Also, you will have to specify the paths for these libraries, for instance:

```
C:\Program Files\boost\lib
C:\Program Files\freetype\bin
C:\Program Files\freetype\lib
C:\Program Files\zlib
C:\Program Files\libpng
```

Warning

A common problem of building for Microsoft Windows is caused because there are several C++ run-time libraries available. If you have to link your code against two or more libraries, you have great chances that each library is using a different run-time. And you will have a problem (LNK4098 - defaultlib "library" conflicts with use of other libs; use /NODEFAULTLIB:library).

The "solution" is to rebuild all the required libraries, ensuring that they are built with the same options.

You are warned!

Using MS Visual Studio, the steps to create the project file and build our example code are the following:

- From Visual Studio's *File* menu, select *New* > *Project...*
- In the left-hand pane of the resulting *New Project* dialog, select *Visual C++* > *Win32*.
- In the right-hand pane, select *Win32 Project*.
- In the *name* field, enter "example-1-win32"

- Right-click **example-1-wx** in the *Solution Explorer* pane and select *Properties* from the resulting pop-up menu
- In *Configuration Properties* > *C/C++* > *General* > *Additional Include Directories*, enter the path to the required include directories, for example

```
C:\Program Files\LenMus\lomse\include
C:\Program Files\boost
C:\Program Files\freetype\include
C:\Program Files\freetype\include\freetype2
```

- In *Configuration Properties* > *C/C++* > *Precompiled Headers*, change *Use Precompiled Header (/Yu)* to *Not Using Precompiled Headers*.
- In *Configuration Properties* > *Linker* > *General* > *Additional Library Directories*, enter the path to Lomse library and to the required libraries. For instance:

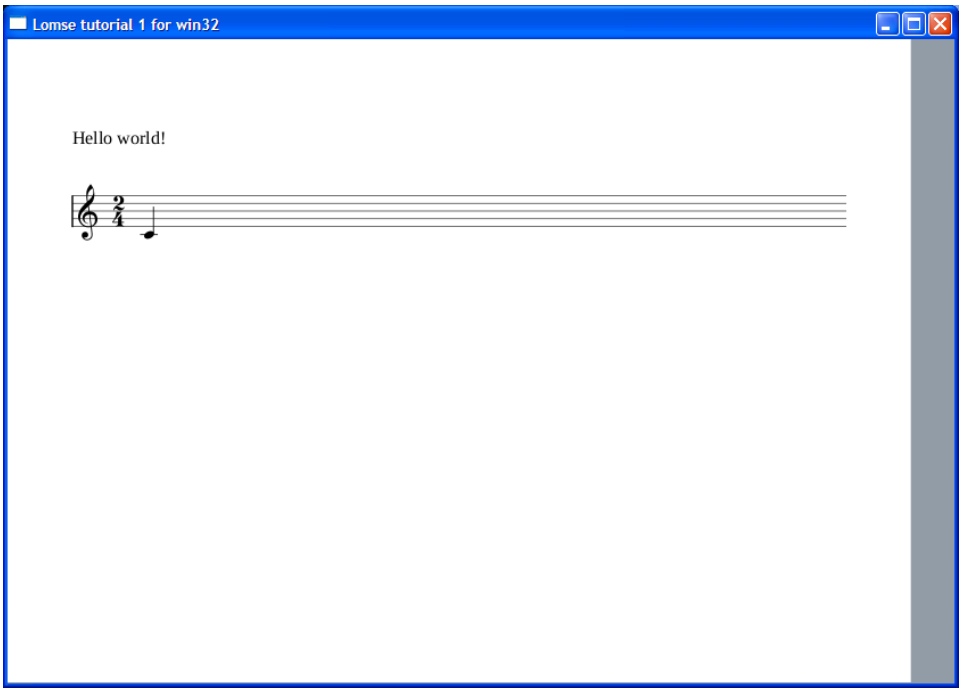
```
C:\Program Files\LenMus\lomse\bin
C:\Program Files\boost\lib
C:\Program Files\freetype\bin
C:\Program Files\freetype\lib
C:\Program Files\zlib
C:\Program Files\libpng
```

- In *Configuration Properties* > *Linker* > *Input* > *Additional Dependencies*, enter the names of the required libraries:

```
lomse.lib
freetype.lib
libpng.lib
zlib.lib
```

- Replace the contents of the `example-1-win32.cpp` generated by the IDE with the code of our example_1_win32.cpp file.
- From the *Build* menu, select *Build Solution*.

When running the program you should see something as:



13. Conclusions

This document is a very basic introduction. In the second tutorial I will add more code to our sample application for interacting with the score (zooming, dragging, selecting objects, etc.).

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.