A CROSS-PLATFORM SOFTWARE LIBRARY FOR DIAGRAMS
CREATION AND MANIPULATION

MICHAL BLIŽŇÁK1, TOMÁŠ DULÍK2, VLADIMÍR VAŠEK3
Department of Applied Informatics1,2, Department of Automation and Control Engineering3
Faculty of Applied Informatics, Tomas Bata University
Nad Stráněmi 4511, 760 05, Zlín
CZECH REPUBLIC
bliznak@fai.utb.cz1, dulik@fai.utb.cz2, vasek@fai.utb.cz3

Abstract: The aim of this paper is to introduce new cross-platform software library called wxShapeFramework (shortly wxSF) written in C++ language which is suitable for creation of software applications manipulating diagrams, images and other graphic objects. The library is based on open-source cross-platform GUI tool kit called wxWidgets and persistent XML-based data container called wxXmlSerializer. As shown in this paper the wxSF allows user to easily create cross-platform applications able to interactively handle various scenes consisting of pre-defined or user-defined graphic objects (both vector- and bitmap-based), store them to XML files, export them to bitmap images, print them etc. Moreover, thanks to applied software licence the library can be used for both open-source and commercial projects on all main target platforms including MS Windows, MacOS and Linux.

Keywords: Diagram, vector, bitmap, wxWidgets, wxXmlSerializer, wxShapeFramework, wxSF, C++

1 Introduction

Modern software applications often need the ability to graphically represent various data or logical structures, information flows, processes and similar abstract information types in the form of diagrams. Whatever the application does, the graphical representation of any problem is always more clear and understandable then a textual one.

The main goal of this paper is to introduce a new open-source cross-platform software library called wxShapeFramework (shortly wxSF) [1] written in C++ language. The library is based on well-known cross-platform GUI library wxWidgets [2] and is suitable for easy creation of software applications manipulating various diagrams and other graphic objects. It is a replacement for fairly out-of-date wxWidgets add-on library called OGL (Object Graphics Library) [3] which is not developed any more. The wxSF can be used as a base part of applications like various CASE tools, technological processes modeling tools, etc.

2 What the wxShapeFramework is

The library consists of a set of classes encapsulating so called shape canvas (a visual GUI control used for management of graphic objects and supporting serialization/deserialization to XML files, clipboard and drag&drop operations, undo/redo, export to BMP files, printing, etc.) and diagram graphic objects called shapes (including basic rectangular and elliptic shapes, line and curve shapes, polygonal shapes, static and in-place editable text, bitmap images, etc.). The wxSF allows to define relationship between various shape types (for example which shape can be a child of another one, which shape types can be connected together by which connector type, how various connections look like, etc.) and provides ability to interactively design diagrams composed of those shape objects.

3 A Technological Background And The Library's Structure

The library uses the wxWidgets API, so it is platform independent as far as the appropriate wxWidgets port is available for a required target platform. wxSF also uses the persistent data container provided by the wxXmlSerializer (shortly wxXS) software library [5]. wxXS allows users to easily serialize and deserialize hierarchically arranged class instances and their data members to an XML structure. The XML content can be stored to a disk file or to another output stream supported by wxWidgets. This functionality is used for saving and loading diagrams as well as a base for the clipboard and undo/redo operations provided by the wxSF.

wxSF consists of more than 40 classes which can be divided by their purpose into three main groups:

- classes implementing a diagram manager,
- classes implementing the shape canvas,
- diagram components classes.
The class diagram of main library classes is shown in figure 1.

![Diagram 1: Main library classes and their cooperation](image1)

The diagram manager encapsulated by `wxSFDiagramManager` class is a main persistent data container (inherited from `wxXmlSerializer` class provided by the wxXS library). It is responsible for management of included diagram components and for saving/loading them to/from various I/O streams. It also provides a set of member functions suitable for basic policy definition (user can define which shapes can be included into a specific diagram manager instance).

The diagram manager is not a visual object; it only stores and manages diagram components. The diagram is visualized by so called shape canvas encapsulated by `wxSFShapeCanvas` class. Shape canvas can be assigned to one instance of data manager and acts as its graphic user interface. This approach leads to possibility to process diagrams by an application without need of their displaying (which is useful for loading or creating of diagrams at the background). The shape canvas also provides clipboard and undo/redo functionality as well as a possibility to design the diagram interactively. Moreover, for better performance and drawing scalability it uses special double-buffered painting canvas (bitmap-based memory device context) encapsulated by `wxSFScaledDC` class. This drawing class can use both standard GDI functions and enhanced graphics engine encapsulated by `wxGraphicsContext` class as well.

The last mentioned class group encapsulates the diagram graphic components (shapes). Every shape class is inherited from the base shape class called `wxSFShapeBase` (inherited from `xsSerializable` class provided by the wxXS library). This class encapsulates a basic functionality like moving, drawing invocation, hit detection, and policies definition and includes set of virtual functions allowing a programmer to defined specific shape's properties and behaviour (like resizing or drawing). Finally, there is also a set of predefined shape classes encapsulating common diagram components like

- rectangles (`wxSFRRectShape, ...`),
- squares (`wxSFSquareShape`),
- ellipses (`wxSFEllipseShape`),
- circles (`wxSFCircleShape`),
- text objects (`wxSFTextShape, ...`),
- polygons (`wxSFPolygonShape, ...`),
- grid containers (`wxSFGrigShape, ...`),
- GUI container (`wxSFControlShape`),
- bitmaps (`wxSFBitmapShape`),
- and lines (`wxSFLineShape, ...`)

Every predefined shape can be used as is or as a base for another more specific shape.

Class hierarchy diagram for the most important classes is shown in the figure 2.

![Diagram 2: Diagram objects (shapes) hierarchy](image2)

### 4 Usage of wxShapeFramework

Now let’s take a look at an example how wxSF can be used for creation of simple graphics applications. The first example will demonstrate basic usage of diagram manager and shape canvas classes, the second example will focus to user-defined shape objects creation and manipulation. Note that in these examples only code fragments related to the wxSF are discussed and it is supposed that the reader is familiar with programming using wxWidgets.

#### 4.1 “Hello World” in Graphics

This simple example shows how to create an application displaying a “diagram” managed by one instance of diagram manager class. The diagram is visualized via the shape canvas class. Generally, there
are two ways how the data manager and shape canvas classes can be used; they can be used “as they are” or as bases for new classes. There is no need for inheriting new diagram manager class in our simple scenario so the class wxSFDiagramManager is used as it is and as a static class instance (dynamic diagram manager instances have sense for applications processing more than one diagram at the same time). On the other hand the shape canvas can be used in both ways (as it is or as a base class for further inheritance) in all application scenarios. It depends on the application requirements and programmer’s preferences only. In this paper only the simpler method (usage of original shape canvas class) is discussed.

The diagram manager instance and shape canvas should be declared and created during the application frame window initialization. The initialization code can be as follows:

Example 1:

```cpp
#include "wx/wx.h"
#include "wx/wxsf/wxShapeFramework.h"

class wxSFSample1Frame: public wxFrame
{
    public:
    wxSFSample1Frame(wxFrame *frame);
    ~wxSFSample1Frame();

    private:
    // create wxSF diagram manager
    wxSFDiagramManager m_Manager;
    // create pointer to wxSF shape // canvas
    wxSFShapeCanvas* m_pCanvas;
    // declare event handler for // wxSFShapeCanvas
    void OnRightClickCanvas ( wxMouseEventHandler event );
};

wxSFSample1Frame::wxSFSample1Frame(wxFrame *frame) : wxFrame(frame, -1, title)
{
    // set accepted shapes (accept only // wxSFRectShape)
    m_Manager.AcceptShape(wxT("wxSFRectShape"));
    // create shape canvas and associate it // with shape manager
    m_pCanvas = new wxSFShapeCanvas
                (m_Manager, this);
    // set shape canvas properties if // required:
    m_pCanvas->AddStyle
                (wxSFShapeCanvas::sfsGRID_SHOW);

    // add new rectangular shape to the // diagram:
    wxSFShapeBase* pShape =
        m_Manager.AddShape(CLASSINFO(wxSFRectShape),
                           event.GetPosition());
    if (pShape)
    {
        // set accepted child shapes for the // new shape ...
        pShape->AcceptChild
            (wxT("wxSFRectShape"));
    }
    // ... and then perform standard // operations provided by the shape // canvas:
    event.Skip();
}
```

Let’s discuss the code above in more details. The first code part declares application frame class with constructor, destructor, diagram manager static object, pointer to shape canvas and with one event handler further used by the shape canvas. The frame class constructor code does these initialization steps:

1. setting shape class objects accepted by the diagram manager (now only wxSFRectShape class instances are accepted),
2. creating the shape canvas as a child window of main application frame,
3. definition of shape canvas properties (a design grid is shown and used),
4. registering previously declared event handler in the shape canvas.

Implementation of registered event handler could be like this:

Continuing of Example 1:

```cpp
void wxSFSample1Frame::OnRightClickCanvas (wxMouseEventHandler event)
{
    // add new rectangular shape to the // diagram:
    wxSFShapeBase* pShape =
        m_Manager.AddShape(CLASSINFO(wxSFRectShape),
                           event.GetPosition());
    if (pShape)
    {
        // set accepted child shapes for the // new shape ...
        pShape->AcceptChild
            (wxT("wxSFRectShape"));
    }
    // ... and then perform standard // operations provided by the shape // canvas:
    event.Skip();
}
```

Event handler invoked at the right mouse button click creates new instance of rectangular shape encapsulated by the wxSFRectShape class and adds it to the canvas at a position read from the mouse event class object. Also some basic shape policy is set
here which tells the shape canvas that only wxSFRectShape class object are accepted as a child objects of newly created shape. The last command statement (event.Skip()) calls default event handler implemented in wxSFShapeCanvas class.

And this is all what the user needs to do for implementation of diagrams in his application. The application built from previous code could look like this:

![Diagram Example](image.png)

**Figure 3: wxShapeFramework demonstration**

4.2 Persistence of the diagram

wxShapeFramework library is based on the persistent hierarchical data container provided by wxXmlSerializer library and fully uses its built-in potential so operations like saving or loading of diagrams content can be implemented in very easy way. The diagram manager class inherits set of member functions suitable for serialization and deserialization of its content which can be used as follows.

A current content of diagram manager can be saved to a disk file (or any output stream provided by wxWidgets library) by single code line looking like this one:

```c++
m_Manager.SerializeToXML(wxT("data.xml"));
```

Loading of stored diagram is as easy as the saving and can be performed by this code line:

```c++
m_Manager.DeserializeFromXML(wxT("data.xml"));
```

4.3 User-defined Shapes? Why Not!

The example above uses only predefined shape object but the wxSF allows user to define completely unique shapes based on the most suitable ancestor. In the second example a star shape inherited from wxSFPolygonShape class with embedded editable text shape (wxSFEditTextShape class instance) is created and used in enhanced version of the first example.

The star shape class declaration can be as follows:

**Example 2:**

```c++
// include main wxSF header file
#include "wx/wxsf/wxShapeFramework.h"

class cStarShape : public wxSFPolygonShape {
public:
  // enable RTTI and cloneability
  XS_DECLARE_CLONABLE_CLASS(cStarShape);

  // default constructor used by RTTI
  cStarShape();

  // copy constructor Clone() function
  cStarShape( static cStarShape& obj);

  // destructor
  virtual ~cStarShape();

protected:
  // protected data members
  wxSFEditTextShape* m_pText;
};
```

The implementation code is here:

```c++
#include "StarShape.h"

// implement RTTI information and Clone() // functions
XS_IMPLEMENT_CLONABLE_CLASS(cStarShape, wxSFPolygonShape);

// define star shape as an array of // wxRealPoint values
const wxRealPoint star[10] = {
  wxRealPoint(0,-50), wxRealPoint(15,-10),
  wxRealPoint(50, -10), wxRealPoint(22, 10),
  wxRealPoint(40, 50), wxRealPoint(0, 20),
  wxRealPoint(-40, 50), wxRealPoint(-22, 10),
  wxRealPoint(-50,-10), wxRealPoint(-15,-10)};

// default constructor
cStarShape::cStarShape()
{
  // disable serialization of polygon
  // vertices, because they are always
  // set in this constructor
  EnablePropertySerialization(wxT("vertices"), false);

  // set polygon vertices
  SetVertices(10, star);

  // polygon-based shapes can be connected
  // either to the vertices or to
  // the nearest border point (default
  // value is TRUE).
  SetConnectToVertex(false);

  // set accepted connections for the new
  // shape
  AcceptConnection(wxT("All"));
  AcceptSrcNeighbour(wxT("cStarShape"));
  AcceptTrgNeighbour(wxT("cStarShape"));

  // create associated child shape(s)
}
```
m_pText = new wxSFEditTextShape();
// set some properties
if (m_pText)
{
    // set text
    m_pText->SetText(wxT("Hello!"));
    // set alignment
    m_pText->SetVAlign(wxSFShapeBase::valignMIDDLE);
    m_pText->SetHAlign(wxSFShapeBase::halignCENTER);
    // set required shape style(s)
    m_pText->SetStyle(sfsALWAYS_INSIDE | sfsHOVERING);
    // components of composite shapes
    // created at runtime in parent
    // shape constructor cannot be
    // re-created by the serializer so
    // it is important to disable their
    // automatic serialization ...
    m_pText->EnableSerialization(false);
    // ... but their properties can be
    // serialized in the standard way:
    XS_SERIALIZE_DYNAMIC_OBJECT_NO_CREATE(m_pText, wxT("title"));
    // assign the text shape to the
    // parent polygon shape
    AddChild(m_pText);
}

// copy constructor
static cStarShape& obj)
{
    // clone source child text object..
    m_pText = (wxSFEditTextShape*)
                obj.m_pText->Clone();
    if( m_pText )
    {
        // ... and append it to this shapes
        // as its child
        AddChild(m_pText);
        // this object is created by the
        // parent class constructor and
        // not by the serializer (only its
        // properties are deserialized
        XS_SERIALIZE_DYNAMIC_OBJECT_NO_CREATE
        E(m_pText, wxT("title"));
    }
}

The implementation code may seems quite complex
but it also shows some interesting functionality like
creation of child shapes directly in the program code,
shape cloning or modification of shape layout and
behaviour.

This new shape object (instance of cStarShape
class) can be added to an existing diagram manager in
the way discussed in the first example. cStarShape
class as well as its embedded child (text shape class
wxSFEditTextShape) must be accepted by the
library (using wxSFDiagramManager::AcceptShape())
and then a new star shape can be created and added to
a diagram by the wxSFDiagramManager::AddShape() function.

There is also a possibility to connect several star
shapes by any type of connection line defined in the
wxSF as shown in the example. These connection
lines can be hard-coded or created interactively by
invocation of one of the following functions:

- wxSFDiagramManager::
  CreateConnection()
- wxSFShapeCanvas::
  StartInteractiveConnection()

Figure 4 shows the star shapes defined in the code
above which was added to the Example 1.

Figure 4: User-defined composed shapes

5 Conclusion

As can be seen from the examples, the
wxShapeFramework software library has sufficient
potential for effective development of various
software applications which use diagrams or other
form of visual communications. Note that only very
small fraction of all the functions provided by the
library has been discussed in this paper. For deeper
understanding of its principles and potential we would
recommend to go through the library reference
documentation and sample projects.

The library can be freely obtained from
SourceForge.net software repository[1] and is
distributed under wxWidgets license [4] so it can be
used for both open-source and commercial projects
without any restrictions. Up to the present day the
library has been downloaded more than 900 times and
only few bugs and patches were reported by the users
so it can be regarded (despite its relative youth) as
sufficiently mature software project. Of course, the
development of the wxSF is still in progress so new
features and improvements are continuously included.
to fulfil all requirements of modern cross-platform
diagram software library.

6 Acknowledgements
This work was supported by the Ministry of Education of the Czech Republic under grant No. MSM 7088352102.

7 References