WXSHAPEFRAMEWORK: AN EASY WAY FOR DIAGRAMS MANIPULATION IN C++ APPLICATIONS

MICHAL BLIŽŇÁK[1], TOMÁŠ DULÍK[2], VLADIMÍR VAŠEK[3]
Department of Informatics and Artificial Inteligence[1, 2], Department of Automation and Control Engineering[3]
Faculty of Applied Informatics, Tomas Bata University
Nad Stráněmi 4511, 760 05, Zlín
CZECH REPUBLIC
bliznak@fai.utb.cz[1], dulik@fai.utb.cz[2], vasek@fai.utb.cz[3]

Abstract: wxShapeFramework is new cross-platform software library written in C++ programming language which is suitable for creation of software applications manipulating diagrams, images and other graphical objects. Thanks to the underlying technologies such as wxWidgets toolkit and its XML-based persistent data container add-on called wxXmlSerializer it is an ideal solution for rapid and easy cross-platform visualisation software development. The paper reveals how the wxSF allows user to easily create applications able to interactively handle various scenes consisting of pre-defined or user-defined graphic objects (both vector- and bitmap-based) or GUI controls, 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, GUI, 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 It is a replacement for fairly out-of-date wxWidgets add-on library called OGL (Object Graphics Library) [3] which is not developed any more.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 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. Relation between wxSF and wxXS libraries is explained in figure 2.

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.

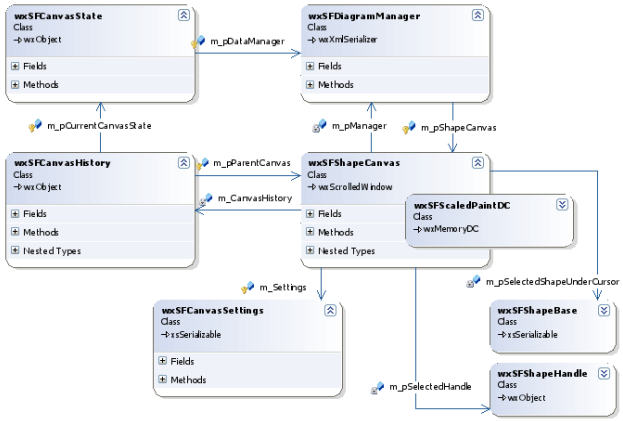

Figure 1: Main library classes and their cooperation

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).

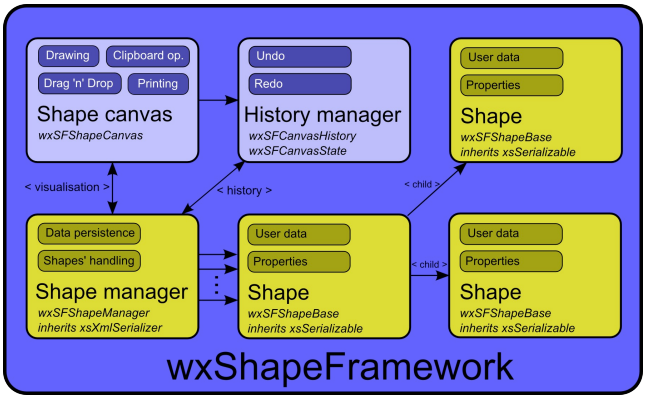

Figure 2: Logical library structure

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 (`wxSFRectShape, ...`),
- 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 it is or as a base for another more specific shape.

Class hierarchy diagram for the most important classes is shown in the figure 3.

Figure 3: Diagram objects (shapes) hierarchy

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:

```
// add wxWidgets header file
#include "wx/wx.h"
// add wxShapeFramework include file
#include "wx/wxsf/wxShapeFramework.h"

// declaration of main application window
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
                    ( wxMouseEvent& event );
};


// constructor of main application frame
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);

    m_pCanvas->AddStyle
        (wxSFShapeCanvas::sfsGRID_USE);

    // connect event handlers to the shape
    // canvas
    m_pCanvas->Connect(wxEVT_RIGHT_DOWN,
wxMouseEventHandler(wxSFSample1Frame::OnRigh
tClickCanvas), NULL, this);
}
```

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:

```
void wxSFSample1Frame::OnRightClickCanvas
    (wxMouseEvent& event)
{
    // add new rectangular shape to the
    // diagram:
    wxSFShapeBase* pShape =
m_Manager.AddShape(CLASSINFO(wxSFRectShape),
event.GetPosition());

    // set some shape's properties if
```

```
    // required:
    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:

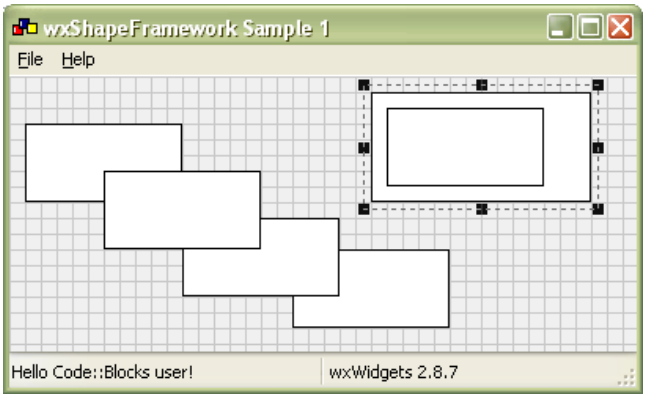

Figure 4: wxShapeFramework demonstration

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:

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

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

```
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:

```
// 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:

```
#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_CREAT
E(m_pText, wxT("title"));

        // assign the text shape to the
        // parent polygon shape
        AddChild(m_pText);
    }
}

// copy constructor
cStarShape::cStarShape(static cStarShape&
obj) : wxSFPolygonShape(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_CREAT
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 diagram manager (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.

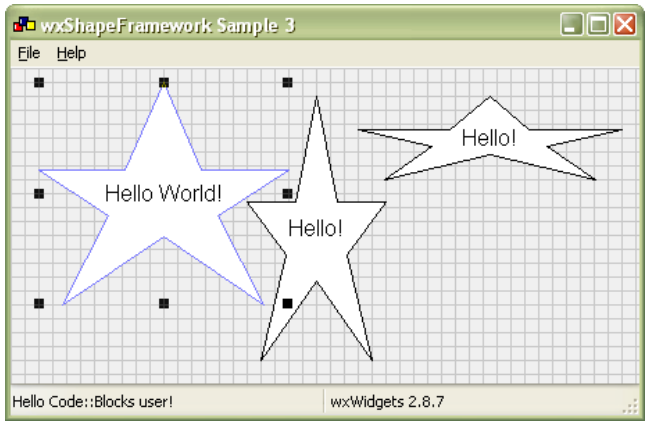

Figure 5: User-defined composed shapes

4.4 Handling of shape and canvas events

Definition and implementation of diagram structure is just only one development aspect of creation of any comprehensive diagram-manipulating application. The second and also very important task is handling of various events which can originate in both shape

canvas and shapes. The wxSF library allows users to use two different ways how to do that and it is only up to the user which way will choose – the functionality is the same.

In general, all events related to shape canvas/shapes can be handled via set of virtual functions declared in relevant classes or by using standard wxWidgets event system thanks to a set of special event types and classes provided by wxSF. Note that wxWidgets events produced by wxSF are mostly generated by default implementations of relevant virtual functions so if a concurrent usage of virtual functions and standard event system is requested then the user is responsible for invoking of standard virtual functions from his own overrided implementations.

4.4.1 Shape canvas events

Shape canvas base class (`wxSFShapeCanvas`) declaration contains set of virtual functions which can be used for handling of various mouse operations performed on the canvas and for handling of some operations performed on displayed shapes. Moreover, all standard events emitted from shapes are redirected to a shape canvas so they can be handled here as well.

At the first, lets demonstrate how standard mouse events performed on a shape canvas can be handled by using virtual functions.

For that reasons shape canvas class `wxSFShapeCanvas` declares these seven virtual functions:

- `virtual void OnLeftDown (wxMouseEvent &event)`
- `virtual void OnLeftDoubleClick (wxMouseEvent &event)`
- `virtual void OnLeftUp (wxMouseEvent &event)`
- `virtual void OnRightDown (wxMouseEvent &event)`
- `virtual void OnRightDoubleClick (wxMouseEvent &event)`
- `virtual void OnRightUp (wxMouseEvent &event)`
- `virtual void OnMouseMove (wxMouseEvent &event)`

In addition also one virtual function determined for handling of keyboard event is declared there:

- `virtual void OnKeyDown (wxKeyEvent &event)`

Note that all these virtual functions implement standard shape canvas behaviour so if a user wants to override them together with preservation of the default functionality then he is responsible for invoking of the originals from his own implementation code as illustrated in following example:

Example 3:

```
// custom shape canvas class
class MyCanvas : public wxSFShapeCanvas
{
    public:
    // overrided virtual functions
    virtual void OnLeftDown(wxMouseEvent
&event);
}

// overrided virtual event handler
void MyCanvas::OnLeftDown(wxMouseEvent
&event)
{
    // your custom code

    // invoke original handler
    wxSFShapeCanvas::OnLeftDown(event);
}
```

As can be seen from example 3 the original event handler is simply called like ordinary function from within the user-defined handler.

This way of user-defined event handlers implementation is quite straightforward but requests small additional programming overhead because a new class must be inherited from `wxSFShapeCanvas` base.

This drawback can be eliminated by using second possible approach, i.e. by using standard wxWidgets event system. All previously mentioned virtual event handlers are de facto called by standard wxWidgets event system so only thing the user has to do is to map desired mouse/keyboard events to his own handler routines in a standard way. Also in this case the user is responsible for calling of original handlers defined in wxSF if the default behaviour should be preserved. This approach is demonstrated in example 4:

Example 4:

```
// custom frame class
class MyFrame : public wxFrame
{
    public:
        MyFrame();
    protected:
        wxSFShapeCanvas *m_Canvas;
        // overrided virtual functions
        void OnLeftDown(wxMouseEvent
&event);
};

MyFrame::MyFrame()
{
    // create and initialize shape canvas
here...
```

```
    // bind desired events to user-defined
handlers
    m_pCanvas->Connect(wxEVT_LEFT_DOWN,
wxMouseEventHandler(MyFrame::OnLeftDown),
NULL, this);

}

// custom event handler
void MyFrame::OnLeftDown(wxMouseEvent
&event)
{
    // your custom code

    // invoke original handler
    event.Skip();
}
```

In this source code an application frame derived from `wxFrame` class is declared. Beside the class constructor it contains also pointer to shape canvas class and declaration of custom event handler function. Binding of desired mouse event to newly created handler is done by using `Connect()` function in the class constructor. Note that invoking of default event handler provided directly by `wxSFShapeCanvas` class is done by using `Skip()` member function which is the common way how to call default event handlers via standard wxWidgets event system.

Main advantage of the second approach is that there is no need of creation of derived canvas class; the original one can be used without any changes.

All previously mentioned virtual handler functions declared in `wxSFShapeCanvas` class have one common aspect: handled events are generated directly by wxWidgets library. But, there is also slightly different set of virtual functions which can be used for handling of events originated in wxSF library. The functions are:

- `virtual void OnTextChange (wxSFEditTextShape *shape)`
- `virtual void OnConnectionFinished (wxSFLineShape *connection)`
- `virtual void OnDrop (wxCoord x, wxCoord y, wxDragResult def, const ShapeList &dropped)`
- `virtual void OnPaste (const ShapeList &pasted)`

The main difference is that these events are generated by wxSF library itself under following conditions:

`OnTextChange` handler is called if any text-based shape's content has changed,

`OnConnectionFinished` handler is called when interactive connection creation is finished,

`OnDrop` handler is called if any shapes are dropped (by using Drag&Drop functionality) to the shape canvas,

and finally `OnPaste` handler is called if some shapes are pasted from system clipboard to active shape canvas.

Default implementations of these default virtual handlers emit relevant standard events which can be handled by standard wxWidgets event system like demonstrated above. For that reason a set of special event types and classes is declared in wxSF. For more details is highly recommended to study library's reference documentation and sample projects, which are parts of library source package distribution.

4.4.2 Shape events

In previous chapter shape canvas events have been discussed. Now lets focus to events which inform user about various operations performed on displayed shapes.

Similarly to the shape canvas class there are set of virtual functions declared in base shape class `wxSFShapeBase` called on relevant events. The functions are:

- `virtual void OnLeftClick (const wxPoint &pos)`
- `virtual void OnRightClick (const wxPoint &pos)`
- `virtual void OnLeftDoubleClick (const wxPoint &pos)`
- `virtual void OnRightDoubleClick (const wxPoint &pos)`
- `virtual void OnBeginDrag (const wxPoint &pos)`
- `virtual void OnDragging (const wxPoint &pos)`
- `virtual void OnEndDrag (const wxPoint &pos)`
- `virtual void OnBeginHandle (wxSFShapeHandle &handle)`
- `virtual void OnHandle (wxSFShapeHandle &handle)`
- `virtual void OnEndHandle (wxSFShapeHandle &handle)`
- `virtual void OnMouseEnter (const wxPoint &pos)`
- `virtual void OnMouseOver (const wxPoint &pos)`
- `virtual void OnMouseLeave (const wxPoint &pos)`

- `virtual bool OnKey (int key)`
- `virtual void OnChildDropped (const wxRealPoint &pos, wxSFShapeBase *child)`

The handlers listed above could be divided into several logical categories:

- **Mouse events** – events emitted when various mouse-related operations are performed on the shape,
- **Handle events** – events emitted when various operations are performed on shape handle,
- **Keyboard event** – event emitted when any key is pressed while the shape is selected,
- **Other events** – other events informing user about shape state change.

All those shape events are called internally by shape framework and their default handlers are responsible for emitting of relevant wxWidgets events which can be handled by using event tables in the standard way.

Note that wxWidgets events are emitted only if source shape contains `sfsEMIT_EVENTS` flag which is set off in default. This restriction is here due to possible congestion of event loop by events emitted from shapes in extend diagrams). All emitted events are internally redirected to a shape canvas where the source shape is displayed on so the wxWidgets event must be caught there.

Obviously, there are similar rules for working with shape event handlers like with shape canvas ones: if the default behavior should be preserved (i.e. wxWidgets events should be emitted) then the default implementations must be invoked from user-defined overrided handlers.

For better handling of shape event a set of special event classes, event types and mapping macros are provided by wxSF (`wxSFShapeEvent`, `wxSFShapeTextEvent`, `wxSFShapeDropEvent`, `wxSFShapePasteEvent`, `wxSFShapeHandleEvent`, `wxSFShapeKeyEvent`, `wxSFShapeMouseEvent` and `wxSFShapeChildDropEvent`). For further documentation please see the library reference manual or sample application wxSFSample1 which is shown in figure 6 demonstrating ability to bind specific shape events via built-in wxWidgets event system.

The following example shows how to enable emitting of shape events and how to handle them.

Example 5:

Assume that we have one shape which should emit wxWidgets events so we have to allow this functionality by setting of `sfsEMIT_EVENTS` flag:

```
// pShape is a pointer of type wxSFShapeBase or derived

pShape->AddStyle( wxSFShapeBase::sfsEMIT_EVENTS );
```

Now lets register new event handler for standard event emitted when a shape handle is dragged. Somewhere in an application initialization code the new event handler must be bound in this way:

```
// m_pCanvas is pointer to wxSFShapeCanvas class instance
m_pCanvas->Connect(wxEVT_SF_SHAPE_HANDLE, wxSFShapeHandleEventHandler(MyFrame::OnShapeHandleEvent), NULL, this);
```

and its implementation then could looks as follows:

```
// user-defined event handler
void MyFrame::OnShapeHandleEvent(wxSFShapeHandleEvent& event)
{
    // get reference to dragged shape handle
    wxSFShapeHandle &hnd = event.GetHandle();

    // perform desired operations here ...

    // invoke default handler if needed
    event.Skip();
}
```

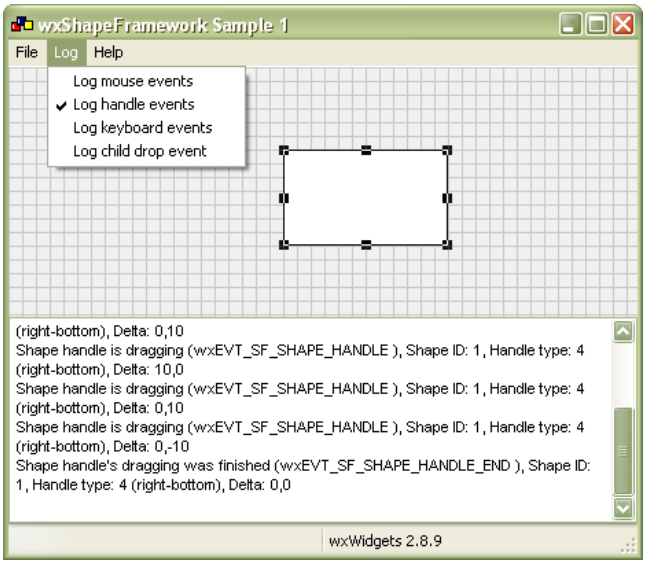

Figure 6: Shape events demonstration

4.5 GUI controls management

Another unique feature provided by wxShapeFramework library is an ability to manage various GUI controls by using a special shape type. This feature allows a dynamic re-organization of an application user interface and also displaying and

usage of any GUI control as a part of composed shapes.

All that features are possible thanks to special `wxSFControlShape` class. It is a simple rectangular shape which can act as a container for wxWidgets GUI controls. In addition to basic functionality provided by `wxSFRectShape` base class it allows user to define how GUI events will be processed and in which way the shape dimension will be controlled. All wxWidgets events emitted from managed GUI control can be processed either by the GUI control itself or by parent shape canvas and shape control dimensions can be updated to the managed GUI control ones or vice versa.

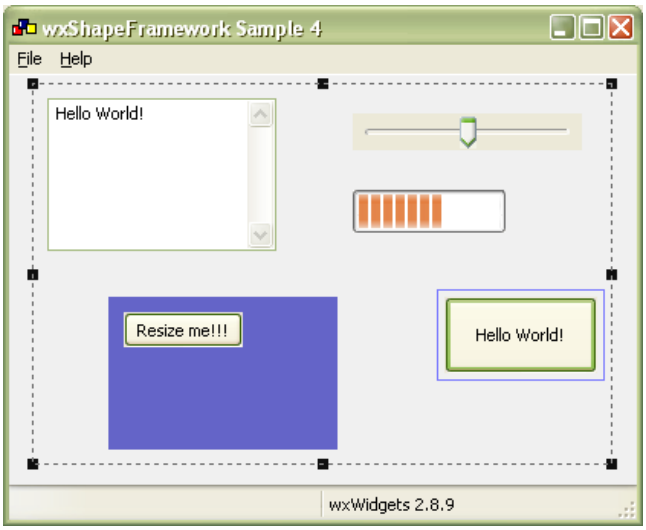

Figure 7: Shape events demonstration

Figure 7 shows sample project wxSFSample4 which demonstrates usage of `wxSFControlShape`. It allows user to place several GUI controls onto a shape canvas, move them around and resize them.

The basic idea of `wxSFControlShape` usage is as follows:

1. User creates an instance of `wxSFControlShape` and sets some visual aspects like GUI control offset or pen and brush used during modification of shape dimensions and position (in this case the GUI control is hidden during the modification process).
2. User have to assign an instance of GUI control which should be managed to the control shape. An authoritative dimensions can be set during this step, i.e. user can determine whether the control shape will be resized to fit managed GUI control or vice versa.
3. User should determine how GUI events will be processed: they can be processed by either shape canvas where the control shape is displayed on or by the GUI control itself. However, this behaviour can be modified any time during the control shape lifetime.

Following commented example illustrates how a control shape managing simple button can be created.

Example 6:

```
// create new control shape
wxSFControlShape* pShape =
(wxSFControlShape*)
m_Manager.AddShape(CLASSINFO(wxSFControlShap
e), event.GetPosition());

// set properties
if(pShape)
{
    // disable accepted child shapes
    pShape->ClearAcceptedChilds();

    // set some visual aspects here:
    pShape->SetControlOffset(5);
    //pShape->SetModBorder
(*wxTRANSPARENT_PEN);
    //pShape->SetModFill(wxBrush(*wxRED,
wxCROSSDIAG_HATCH));

    // Assign managed GUI control to the
canvas. Remember the control shape now owns
the GUI control and it is
    // deleted by the shape control in its
destructor. If you want to remove the GUI
control from the parent shape
    // just assign a new control or NULL
value to it. You can also specify whether
managed GUI control
    // is resized in accordance to
dimensions of its parent control shape or
vice versa.
    pShape->SetControl( new
wxButton(m_pCanvas, wxID_ANY, wxT("Hello
World!")), sfFIT_SHAPE_TO_CONTROL);
    //pShape->SetControl( new
wxButton(m_pCanvas, wxID_ANY, wxT("Hello
World!")), sfFIT_CONTROL_TO_SHAPE);

    // You can specify whether events
generated by the managed control are
processed by the shape canvas
    // or/and the widget as well. Note that
GUI controls differ in a way how they
process events
    // so the behaviour can be different for
various widgets.
    pShape->SetEventProcessing
(wxSFControlShape::evtMOUSE2CANVAS |
wxSFControlShape::evtKEY2CANVAS);
    //pShape->SetEventProcessing
(wxSFControlShape::evtMOUSE2GUI |
wxSFControlShape::evtKEY2GUI);
}
```

4.6 Undo/Redo and Clipboard

The previous chapters dealt with basic functionality and operations. In addition, wxSF provides even more advanced functionality which could please many

developers (and application users as well, of course) such as built-in support for undo/redo and clipboard operations.

All these operations use built-in serialization functionality provided by underlying wxXmlSerializer library [5] in default so no additional programming overhead is required. The operations can be simply invoked by following `wxSFShapeCanvas` member functions:

- `Undo()`, `Redo()`, `Copy()`, `Cut()` and `Paste()`

There are also set of member functions defined in shape canvas class which allow user to determine whether requested operation has a sense at the moment (i.e. any shape is selected or the clipboard contains data understandable for wxSF library). The functions are following:

- `CanUndo()`, `CanRedo()`, `CanCopy()`, `CanCut()` and `CanPaste()`.

Undo/redo operations are internally provided by so called history manager (encapsulated by `wxSFCanvasHistory` class) which can be used for further tuning of their functionality like setting of a history depth (number of stored canvas states) or clearing of whole history. Reference to the history manager can be obtained from shape canvas by `wxSFShapeCanvas::GetHistoryManager()` member function.

4.7 Printing and image export

In addition to the undo/redo and clipboard operations the library has also printing and image exporting capabilities. All these operations are encapsulated by `wxSFShapeCanvas` class (in cooperation with set of other helper classes) and can be simply invoked by its member functions like `Print()`, `PrintPreview()` and `SaveCanvasToBMP()`.

Moreover, printing can be tuned in more details by `SetPrintHAlign()`, `SetPrintVAlign()` and `SetPrintMode()` functions in application source code or interactively by an application user via dialog window which can be displayed by using `PageSetup()` function.

Shape canvas content can be also exported to a bitmap file (in BMP format) by using `SaveCanvasToBMP()` function which will make a snapshot of current canvas drawing.

5 Conclusion

As can be seen from the document and given 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 wxCode repository [6] 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 4500 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

[1] wxShapeFramework library website, 2008: http://sourceforge.net/projects/wxsf

[2] Smart, J., Hock, K. *Cross-Platform GUI Programming with wxWidgets*, Prentice Hall, 2006

[3] wxOGL code repository at wxCode website, 2008,: http://wxcode.sourceforge.net/showcomp.php?name=ogl

[4] wxWidgets license documents, 2008: http://www.wxwidgets.org/about/newlicen.htm

[5] Bližňák, Michal, Dulík, Tomáš, Vašek, Vladimír, *A Persistent Cross-Platform Class Objects Container for C++ and wxWidgets* , WSEAS TRANSACTIONS on COMPUTERS, Issue 5, Volume 8, May 2009, p.778-787, ISSN 1109-2750.

[6] wxCode repository (official website for wxWidgets add-ons), 2010, https://sourceforge.net/projects/wxcode/files/