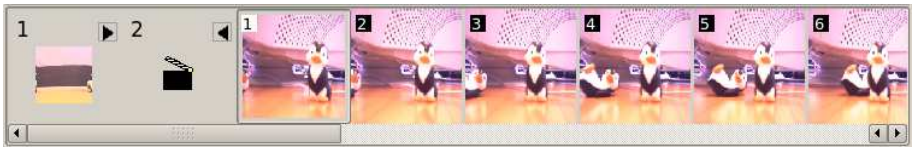

Figure 1: The FrameBar

0.1 Custom widgets

As a media application Stopmotion needed to display the pictures in various ways to the user. We searched around in the beginning, but we didn't find widgets using Qt and C++ which suited all our needs.

Therefore we needed to write some custom widgets our self, most notably the FrameBar and the FrameView.

0.1.1 FrameView widget

The FrameView is a widget which uses the SDL libraries to display images to the user. It can also perform Onionskinning, image differentiation and playback at requests. FrameView uses some paint code from an example widget from the SDL webpage[1], but has evolved greatly from this basic widget.

As an observer the FrameView widget is notified by the model every time something is changed in the model. It ignores most of these notifications, but paints the image for the frame with the number `frameNumber` when it receives one of the following update requests:

```
void FrameView::updateNewActiveFrame(int frameNumber);
void FrameView::updatePlayFrame(int frameNumber);
void FrameView::updateAnimationChanged(int frameNumber)
```

0.1.2 FrameBar widget

Figure 1 shows the FrameBar custom widget. The FrameBar is actually a package of five widgets. The FrameBar itself, the abstract ThumbView class, the FrameThumbView and SceneThumbView classes and the SceneArrowButton.

As shown in figure 2 the FrameBar class inherits from both the Observer interface and the QScrollView widget class. From QScrollView it gets the scrollbar as well as auto-scrolling[2] for free and by being an *Observer* it can, like the FrameView, be registered so that it receives notification when something is changed in the model.

The figure also shows that the FrameBar can contain a series of ThumbViews. There are two kinds of ThumbViews, the FrameThumbView which is the miniature

[1] http://www.libsdl.org/cvs/qtSDL.tar.gz

[2] The auto-scrolling is used by moving the mouse to the edge of the FrameBar while dragging a scene, a frame or a picture from another program.

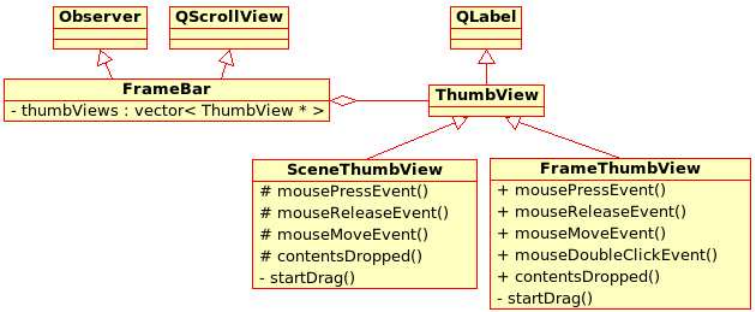

Figure 2: The FrameBar classes with some attributes and operations

picture of a frame as seen in figure 1 and the SceneThumbView which represent the beginning of a scene.

The FrameBar will react to almost all notifications from the model and will update the thumbviews to reflect these changes. When a frame or scene is added, removed or moved in the model the FrameBar will add, remove or move the corresponding ThumbView accordingly. As explained in the Qt documentation[3] a QScrollView class can't use layout managers if the contents can become larger than 4000 pixels horizontally or vertically. As the FrameBar class needs to use allot more than 4000 pixels for displaying all its data we had to move all the ThumbViews around by pixels using the function:

```
moveChild ( QWidget * child, int x, int y )
```

The following example shows how the ThumbView widgets are moved when the user wants to move a selection of frames backwards in the FrameBar:

[3]http://doc.trolltech.com/3.3/qscrollview.html

```
for(unsigned int i=movePosition; i<fromFrame; i++) {
  moveChild(thumbViews[i], childX(thumbViews[i]) +
          FRAMEBAR_HEIGHT*(toFrame-fromFrame+1), 0 );
  thumbViews[i]->
          setNumber(i+(toFrame-fromFrame)-activeScene);
}

for(unsigned int j=fromFrame; j<=toFrame; j++) {
  moveChild( thumbViews[j], childX(thumbViews[j]) -
          FRAMEBAR_HEIGHT*(fromFrame-movePosition), 0 );
  moveThumbView(j, j-(fromFrame-movePosition));
}
```

The first loop moves all the ThumbViews between the first selected frame (fromFrame) and the position to move to(movePosition) forward to make room for the new frames. The second loop then moves all the frames in the selection to their new position. This way of moving the ThumbViews around was very tedious and time consuming to program as ThumbViews needed to be moved for most kind of notifications, but because of the way QScrollView works we didn't have a choice.

When the user wants to move a Frame or a Scene this is done through drag and drop. If the user for instance grabs a frame and starts moving the mouse a drag will commence. This drag is a so called URI drag meaning it contains the path to the file. If a *dropEvent* occur in a FrameThumbView it will check if the drag originated inside the application. If it did it will tell the FrameBar to move the selected frames to the position it is located at and if the drag originated outside the application it will tell the FrameBar to add frames to its location in the FrameBar. This way we get an uniform way of handling drags whether the user wants to drag them frames to another application or move them around inside Stopmotion. We also get the QScrollBars auto-scroll functionality by using drag and drop inside the FrameBar.

The ThumbViews extend QLabel and the functionality from this is used to add borders and the FrameThumbViews use it to display the images. The paintEvent(..) functions however are overloaded in both FrameThumbView and SceneThumbView. FrameThumbView extends it to draw an number on top of the image and a note if the frame has sounds attached to it and the SceneThumbView use it to draw everything in it:

The following code is the paintEvent(..) function for the FrameThumbView:

```
void FrameThumbView::paintEvent(QPaintEvent *paintEvent){
    QLabel::paintEvent(paintEvent);
    QPainter painter( this );

    if(selected) {
        painter.fillRect(4, 5, textWidth, 14,
                QBrush(Qt::white));
        painter.setPen( Qt::black );
    }
    else {
        painter.fillRect(4, 5, textWidth, 14,
                QBrush(Qt::black));
        painter.setPen( Qt::white );
    }

    stringstream ss;
    ss << number+1;
    painter.drawText(5, 17, ss.str().c_str());

    if(this->hasSounds) {
        QPixmap noteIcon = QPixmap(note);
        painter.drawPixmap(width()-32, 0, noteIcon);
    }
}
```

The first thing that happens is that QLabel::paintEvent(..) is called thereby allowing it to paint the frame. Then the painter is set up and the background for the framenumber is painter. After this a STL stringstream is used to cast the framenumber to a char which is then displayed. At last a sound icon is displayed in the widget, provided this widget has a sound attached to it.