The book about real examples of Qt Widgets usage

Preface

I guess that you, reader, know C++, know what is object-oriented programming and design. In this book, you won't find answers on the basics of C++, but you can find some practices, useful practices when developing with Qt. I guess that you understand the basics of Qt. This book is an explanation of the development processes of the real GUI projects, written on C++/Qt. You won't find explanations of public Qt API, as I guess that Qt API is very well described in Qt help.

Projects, described in this book are not so big and it's a very good start point to look at the working code, because a developer should read the code of another developer, this will improve your Qt skills. These projects are Open Source and you can become a part of these projects, you are welcome to make pull requests on GitHub with improvements.

The goal of this book is to introduce the reader with Qt Widgets, the best UI framework, in my opinion, for developing cross-platform, effective desktop applications on the real examples. You can look at the chapters of this book as on tutorials.

Back | Contents | Next

Chapter 1

I want to show on the real example how to create simple GIF editor with Qt widgets. Why Qt widgets and not QML? The answer is simple - I want to create a desktop application, and in my opinion for desktop applications, it's better to use widgets. Full sources of this example you can find here https://github.com/igormironchik/gif-editor

As backend of image processing, I will use Magick++ from ImageMagick.

Back | Contents | Next

Basics of the main window

Introduction

Each Qt widgets application should have one or more top-level widgets. For GIF editor we need one top-level window where we will display frames, current frame, toolbar with actions for editing, a menu bar with different actions. Qt has ready to use class QMainWindow which we can derive from and implement needed for our functionality.

Inheritance

Let's inherit from QMainWindow to have the ability to implement our functionality. We will start from basics, in the mainwindow.hpp we have:

```
#ifndef GIF_EDITOR_MAINWINDOW_HPP_INCLUDED
#define GIF_EDITOR_MAINWINDOW_HPP_INCLUDED

// Qt include.
#include <QMainWindow>
#include <QScopedPointer>


//
// MainWindow
//

class MainWindowPrivate;

//! Main window.
class MainWindow final
    :    public QMainWindow
{
    Q_OBJECT

public:
    MainWindow();
    ~MainWindow() noexcept override;

private:
    Q_DISABLE_COPY( MainWindow )

    QScopedPointer< MainWindowPrivate > d;
}; // class MainWindow

#endif // GIF_EDITOR_MAINWINDOW_HPP_INCLUDED
```

I publicly inherited from QMainWindow and in private section you can see usage of Q*OBJECT macros. This macro is needed by Qt's moc to generate auxiliary code for signals and slots. At this time we don't have any signals or slots, but it's a good practice to use Q* OBJECT macros in every class derived from QObject.

I use in my Qt applications private implementation idiom, for this I declared class MainWindowPrivate and in MainWindow I declared member - smart pointer to MainWindowPrivate. The private implementation is good for reducing compile time, it hides details of implementation from interface.

Implementation at this point is very simple (mainwindow.cpp):

```
// GIF editor include.
#include "mainwindow.hpp"
```

```
//
// MainWindowPrivate
//

class MainWindowPrivate {
public:
    MainWindowPrivate( MainWindow * parent )
        :    q( parent )
    {
    }

    //! Parent.
    MainWindow * q;
}; // class MainWindowPrivate


//
// MainWindow
//

MainWindow::MainWindow()
    :    d( new MainWindowPrivate( this ) )
{
}
```

For the future, I defined member to the parent object of MainWindow in MainWindowPrivate class. It can help us in the future to access MainWindow methods from data class (MainWindowPrivate).

Menu

Ok. We have the skeleton of our main window. Let's add File menu with open, save, save as and quit actions. We want to implement GIF editor and without such basic functions our application will cost nothing. First of all, let's define slots in MainWindow class for these actions.

```
private slots:
    //! Open GIF.
    void openGif();
    //! Save GIF.
    void saveGif();
    //! Save GIF as.
    void saveGifAs();
    //! Quit.
    void quit();
```

QMainWindow has a menu bar, status bar, central widget, etc. For such actions it's a good place in the File menu, as in almost all desktop applications. In the constructor of MainWindow we will add code to create the File menu and fill it with actions. Let's see:

```
MainWindow::MainWindow()
    :    d( new MainWindowPrivate( this ) )
{
    setWindowTitle( tr( "GIF Editor" ) );

    auto file = menuBar()->addMenu( tr( "&File" ) );
    file->addAction( QIcon( ":/img/document-open.png" ), tr( "Open" ),
        this, &MainWindow::openGif, tr( "Ctrl+O" ) );
    file->addSeparator();
    file->addAction( QIcon( ":/img/document-save.png" ), tr( "Save" ),
        this, &MainWindow::saveGif, tr( "Ctrl+S" ) );
    file->addAction( QIcon( ":/img/document-save-as.png" ), tr( "Save As" ),
        this, &MainWindow::saveGifAs );
    file->addSeparator();
    file->addAction( QIcon( ":/img/application-exit.png" ), tr( "Quit" ),
        this, &MainWindow::quit, tr( "Ctrl+Q" ) );
}
```

I set title of the main window and created File menu with actions and separators.

Quit from the application

The first slot that we will implement is quit from the editor and empty implementations of other slots.

```
void
MainWindow::openGif()
{

}

void
MainWindow::saveGif()
{

}

void
MainWindow::saveGifAs()
{

}

void
MainWindow::quit()
{
    if( isWindowModified() )
    {
        auto btn = QMessageBox::question( this, tr( "GIF was changed..." ),
            tr( "GIF was changed. Do you want to save changes?" ) );

        if( btn == QMessageBox::Yes )
            saveGif();
    }

    QApplication::quit();
}
```

QWidget, the parent of QMainWindow, has a mechanism to read/set a flag if something was changed in. Why not? In our editor we will set this flag on user's changes and clear it on saving. For the future I added in data class member m currentGif of QString type, where I will store the full path to the current GIF image.

```
    //! Current file name.
    QString m_currentGif;
    //! Parent.
    MainWindow * q;
}; // class MainWindowPrivate
```

Great. But application can be closed with the close button in the window's header. And it's a good idea to invoke MainWindow::quit() slot in handler of this event. For this case we will override closeEvent(), so in MainWindow:

```
protected:
    void closeEvent( QCloseEvent * e ) override;
```

And implementation:

```
void
MainWindow::closeEvent( QCloseEvent * e )
{
    quit();

    e->accept();
}
```

First launch of application

We created basic main window, let's have a look at it. We need a main() function to start the application.

```
// Qt include.
#include <QApplication>
#include <QTranslator>
#include <QLocale>

// GIF editor include.
#include "mainwindow.hpp"


int main( int argc, char ** argv )
{
    QApplication app( argc, argv );

    QIcon appIcon( ":/img/icon_256x256.png" );
    appIcon.addFile( ":/img/icon_128x128.png" );
    appIcon.addFile( ":/img/icon_64x64.png" );
    appIcon.addFile( ":/img/icon_48x48.png" );
    appIcon.addFile( ":/img/icon_32x32.png" );
    appIcon.addFile( ":/img/icon_22x22.png" );
    appIcon.addFile( ":/img/icon_16x16.png" );
    app.setWindowIcon( appIcon );

    QTranslator appTranslator;
    appTranslator.load( "./tr/gif-editor_" + QLocale::system().name() );
    app.installTranslator( &appTranslator );

    MainWindow w;
    w.resize( 800, 600 );
    w.show();

    return app.exec();
}
```

We created QApplication object, an icon of our application, translator, that will load translation according to system's locale, and created MainWindow object on the stack. Set default size, and invoked show() method. Voila, now we need to start application's event loop, what app.exec() do.

Back | Contents | Next

Plans

How do you see UI, the main UI, of the editor? I want to have a horizontal scrollable tape with frames of GIF at the bottom, frames should be checkable with checkbox, so the user will be able to delete some frames from the GIF. And resized to fit available space current frame in the centre of the main window. Frames should be clickable, so the user can select any frame. And for first Alpha version of the application I want to implement crop function. The crop will be accessible from toolbar, and in this mode user should be able to draw a rectangle to crop to, and on pressing Enter crop should do the job for all frames.

That's all. Sounds not so complicated, let's do it. And let's start from creating a widget that will represent a frame in the tape of frames.

I wrote the code for the next five chapters before continuing writing this book because it's very difficult to keep in mind all possible issues that could be during coding. So I wrote the code, debugged it, checked, and started to write an overview of my adventure. After reading next five chapters the editor will open GIF images, the user will see the tape with clickable frames, and on clicking in the centre of the window will be displayed selected frame. Believe me, it looks nice, and I spend only half a day on developing it, that is why I love Qt so much.

Back | Contents | Next

Frame

Class

Frame... This is an unit of GIF image. We need a thumbnail frame on the tape to display a sequence of frames in GIF, and a bigger one to display currently selected frame. The task of the frame is to display image, so why not to have one class for both cases? Hm, why not? But for the current frame we need the image to be scaled to the size of the available area with keeping aspect ratio, whereas for the frame on the tape we need an image scaled to the height of the tape. The image on the frame should automatically resize on parent resizing, and it should be clickable. I guess that this is enough for our application. Possibly we will need something additional in the future, possibly, but for the first attempt this is all that we need. Let declare a class of our frame.

```
#ifndef GIF_EDITOR_FRAME_HPP_INCLUDED
#define GIF_EDITOR_FRAME_HPP_INCLUDED

// Qt include.
#include <QWidget>
#include <QScopedPointer>


//
// Frame
//

class FramePrivate;

//! This is just an image with frame that fit the given size or height.
class Frame final
    :    public QWidget
{
    Q_OBJECT

signals:
    //! Clicked.
    void clicked();

public:
    //! Resize mode.
    enum class ResizeMode {
        //! Fit to size.
        FitToSize,
        //! Fit to height.
        FitToHeight
    }; // enum class ResizeMode

    Frame( const QImage & img, ResizeMode mode, QWidget * parent = nullptr );
    ~Frame() noexcept override;

    //! \return Image.
    const QImage & image() const;
    //! Set image.
    void setImage( const QImage & img );

    QSize sizeHint() const override;

protected:
    void paintEvent( QPaintEvent * ) override;
    void resizeEvent( QResizeEvent * e ) override;
    void mouseReleaseEvent( QMouseEvent * e ) override;

private:
    Q_DISABLE_COPY( Frame )

    QScopedPointer< FramePrivate > d;
}; // class Frame

#endif // GIF_EDITOR_FRAME_HPP_INCLUDED
```

Implementation

With private data class all is simple, it's better to see one time than hear thousand times.

```
class FramePrivate {
public:
    FramePrivate( const QImage & img, Frame::ResizeMode mode,
        Frame * parent )
        :    m_image( img )
        ,    m_mode( mode )
        ,    q( parent )
    {
    }

    //! Create thumbnail.
    void createThumbnail();
    //! Frame widget was resized.
    void resized();

    //! Image.
    QImage m_image;
    //! Thumbnail.
    QImage m_thumbnail;
    //! Resize mode.
    Frame::ResizeMode m_mode;
    //! Parent.
    Frame * q;
}; // class FramePrivate
```

I declared two methods to create a thumbnail of needed size and auxiliary method to do some stuff when widget will be resized, as creating a thumbnail, notifying layouts about size change and updating our frame.

The creation of a thumbnail is different for different resize modes of a frame. For fit to size mode we need to scale in both directions keeping the aspect ratio of the image, whereas for the fit to height mode we just need to scale to height keeping aspect ratio too. Let's have a look.

```
void
FramePrivate::createThumbnail()
{
    if( m_image.width() > q->width() || m_image.height() > q->height() )
    {
        switch( m_mode )
        {
            case Frame::ResizeMode::FitToSize :
                m_thumbnail = m_image.scaled( q->width(), q->height(),
                    Qt::KeepAspectRatio, Qt::SmoothTransformation );
            break;

            case Frame::ResizeMode::FitToHeight :
                m_thumbnail = m_image.scaledToHeight( q->height(),
                    Qt::SmoothTransformation );
            break;
        }
    }
    else
        m_thumbnail = m_image;
}

void
FramePrivate::resized()
{
    createThumbnail();

    q->updateGeometry();

    q->update();
}
```

Some methods' implementations of Frame class are quite simple and don't need an

explanation.

```
Frame::Frame( const QImage & img, ResizeMode mode, QWidget * parent )
    :    QWidget( parent )
    ,    d( new FramePrivate( img, mode, this ) )
{
    switch( mode )
    {
        case ResizeMode::FitToSize :
            setSizePolicy( QSizePolicy::Expanding, QSizePolicy::Expanding );
        break;

        case ResizeMode::FitToHeight :
            setSizePolicy( QSizePolicy::Fixed, QSizePolicy::Expanding );
        break;
    }
}

Frame::~Frame() noexcept
{
}

const QImage &
Frame::image() const
{
    return d->m_image;
}

void
Frame::setImage( const QImage & img )
{
    d->m_image = img;

    d->resized();
}

QSize
Frame::sizeHint() const
{
    return d->m_thumbnail.size();
}
```

Events

Painting needs just to draw a thumbnail in the center of the widget.

```
void
Frame::paintEvent( QPaintEvent * )
{
    const int x = ( width() - d->m_thumbnail.width() ) / 2;
    const int y = ( height() - d->m_thumbnail.height() ) / 2;

    QPainter p( this );
    QRect r = d->m_thumbnail.rect();
    r.moveTopLeft( QPoint( x, y ) );
    p.drawImage( r, d->m_thumbnail, d->m_thumbnail.rect() );
}
```

We want an image to be resized automatically on resizing of widget. That is why I overrided resizeEvent().

```
void
Frame::resizeEvent( QResizeEvent * e )
{
    if( d->m_mode == ResizeMode::FitToSize ||
        ( d->m_mode == ResizeMode::FitToHeight && e->size().height() != d->m_thumbnail.height() ) )
            d->resized();

    e->accept();
}
```

And mouseReleaseEvent() to notify about clicking on the frame.

```
void
Frame::mouseReleaseEvent( QMouseEvent * e )
{
    if( e->button() == Qt::LeftButton )
    {
        emit clicked();

        e->accept();
    }
    else
        e->ignore();
}
```

Great, a few hundred lines of code (with blank ones and comments) and we have a class that will display image of the frame, have different behaviour for different cases. Qt rocks!

Back | Contents | Next

Frame on tape

Well, we have a frame widget that will display a thumbnail image. But this is not enough for a frame on the tape. The frame on the tape should have a border, that should signal that this frame is current or not, the frame on the tape should have a checkbox to have the ability to remove some frames from the GIF, just deselect some frames, save file, and voila. And frame on the tape should have a counter, indicating the position of the frame on the tape.

Sounds like we can do it with standard widgets and layouts. We can create a widget, inherited from QFrame to have a border, QCheckBox for checkbox, QLabel for position indicator, and QVBoxLayout and QHBoxLayout for layout. Amazing, there is nothing better than reusing of the code, especially if this code written not by us.

The declaration of the new class looks like.

```
#ifndef GIF_EDITOR_FRAMEONTAPE_HPP_INCLUDED
#define GIF_EDITOR_FRAMEONTAPE_HPP_INCLUDED

// Qt include.
#include <QFrame>
#include <QScopedPointer>


//
// FrameOnTape
//

class FrameOnTapePrivate;

//! Frame on tape.
class FrameOnTape final
    :    public QFrame
{
    Q_OBJECT

signals:
    //! Clicked.
    void clicked( int idx );
    //! Checked.
    void checked( bool on );

public:
    FrameOnTape( const QImage & img, int counter, QWidget * parent = nullptr );
    ~FrameOnTape() noexcept override;

    //! \return Image.
    const QImage & image() const;
    //! Set image.
    void setImage( const QImage & img );

    //! \return Is frame checked.
    bool isChecked() const;
    //! Set checked.
    void setChecked( bool on = true );

    //! \return Counter.
    int counter() const;
    //! Set counter.
    void setCounter( int c );

    //! \return Is this frame current?
    bool isCurrent() const;
    //! Set current flag.
    void setCurrent( bool on = true );

private:
    Q_DISABLE_COPY( FrameOnTape )

    QScopedPointer< FrameOnTapePrivate > d;
}; // class FrameOnTape

#endif // GIF_EDITOR_FRAMEONTAPE_HPP_INCLUDED
```

Nothing difficult. We just added some auxiliary API to have access to the underlying full image, counter or position of the frame, ability to set and check if the current frame is current, and ability to check if this frame is checked.

Implementation really very simple. Look at private data class.

```
class FrameOnTapePrivate {
public:
    FrameOnTapePrivate( const QImage & img, int counter, FrameOnTape * parent )
        :    m_counter( counter )
        ,    m_current( false )
        ,    m_frame( new Frame( img, Frame::ResizeMode::FitToHeight, parent ) )
        ,    m_label( new QLabel( parent ) )
        ,    m_checkBox( new QCheckBox( parent ) )
        ,    q( parent )
    {
        m_checkBox->setChecked( true );

        m_label->setAlignment( Qt::AlignVCenter | Qt::AlignRight );
        m_label->setText( FrameOnTape::tr( "#%1" ).arg( m_counter ) );
    }

    //! Set current state.
    void setCurrent( bool on );

    //! Counter.
    int m_counter;
    //! Is current?
    bool m_current;
    //! Frame.
    Frame * m_frame;
    //! Counter label.
    QLabel * m_label;
    //! Check box.
    QCheckBox * m_checkBox;
    //! Parent.
    FrameOnTape * q;
}; // class FrameOnTapePrivate
```

We declared setCurrent() method as we will use this code more than once.

```
void
FrameOnTapePrivate::setCurrent( bool on )
{
    m_current = on;

    if( m_current )
        q->setFrameStyle( QFrame::Panel | QFrame::Sunken );
    else
        q->setFrameStyle( QFrame::Panel | QFrame::Raised );
}
```

We just changing a frame's style to indicate that this frame is currently selected.

And the implementation of the class is so simple that even doesn't need any comments.

```
FrameOnTape::FrameOnTape( const QImage & img, int counter, QWidget * parent )
    :    QFrame( parent )
    ,    d( new FrameOnTapePrivate( img, counter, this ) )
{
    auto vlayout = new QVBoxLayout( this );
    vlayout->setMargin( 0 );
    vlayout->addWidget( d->m_frame );

    auto hlayout = new QHBoxLayout;
    hlayout->setMargin( 0 );
    hlayout->addWidget( d->m_checkBox );
    hlayout->addWidget( d->m_label );

    vlayout->addLayout( hlayout );

    d->setCurrent( false );

    setLineWidth( 2 );
```

```
    setSizePolicy( QSizePolicy::Fixed, QSizePolicy::Expanding );

    connect( d->m_checkBox, &QCheckBox::stateChanged,
        [this] ( int state ) { emit this->checked( state != 0 ); } );
    connect( d->m_frame, &Frame::clicked,
        [this] ()
        {
            this->d->setCurrent( true );

            emit this->clicked( this->d->m_counter );
        } );
}

FrameOnTape::~FrameOnTape() noexcept
{
}

const QImage &
FrameOnTape::image() const
{
    return d->m_frame->image();
}

void
FrameOnTape::setImage( const QImage & img )
{
    d->m_frame->setImage( img );
}

bool
FrameOnTape::isChecked() const
{
    return d->m_checkBox->isChecked();
}

void
FrameOnTape::setChecked( bool on )
{
    d->m_checkBox->setChecked( on );
}

int
FrameOnTape::counter() const
{
    return d->m_counter;
}

void
FrameOnTape::setCounter( int c )
{
    d->m_counter = c;

    d->m_label->setText( tr( "#%1" ).arg( c ) );
}

bool
FrameOnTape::isCurrent() const
{
    return d->m_current;
}

void
FrameOnTape::setCurrent( bool on )
{
    d->setCurrent( on );
}
```

Tape

Ok, now we have FrameOnTape class, but this class can display a single frame. But animated GIF has several frames. And we should display all frames in a sequence - tape. The tape should be a horizontally scrollable widget with all available frames in the GIF. Scrollable? This is simple, in Qt we have QScrollArea class, and we just need a widget that should have ability to add, remove frames on it, and should grow in width on adding new frames, as well as it should reduce its width on removing a frame.

Let's reuse as much code as possible. We just need a QWidget with QHBoxLayout where we will add FrameOnTape objects.

So, as usuaul, let's have a look at class declarartion.

```
#ifndef GIF_EDITOR_TAPE_HPP_INCLUDED
#define GIF_EDITOR_TAPE_HPP_INCLUDED

// Qt include.
#include <QWidget>
#include <QScopedPointer>


class FrameOnTape;


//
// Tape
//

class TapePrivate;

//! Tape with frames.
class Tape final
    :    public QWidget
{
    Q_OBJECT

signals:
    //! Frame clicked.
    void clicked( int idx );
    //! Current frame changed.
    void currentFrameChanged( int idx );

public:
    Tape( QWidget * parent = nullptr );
    ~Tape() noexcept override;

    //! \return Count of frames.
    int count() const;
    //! Add frame.
    void addFrame( const QImage & img );
    //! \return Frame.
    FrameOnTape * frame( int idx ) const;
    //! \return Current frame.
    FrameOnTape * currentFrame() const;
    //! Set current frame.
    void setCurrentFrame( int idx );
    //! Remove frame.
    void removeFrame( int idx );
    //! Clear.
    void clear();

private:
    Q_DISABLE_COPY( Tape )

    QScopedPointer< TapePrivate > d;
}; // class Tape

#endif // GIF_EDITOR_TAPE_HPP_INCLUDED
```

API is intuitive, it doesn't need an explanation, so let's look at the implementation.

Private data class looks like.

```
//
// TapePrivate
//

class TapePrivate {
public:
    TapePrivate( Tape * parent )
        :    m_currentFrame( nullptr )
        ,    m_layout( new QHBoxLayout( parent ) )
        ,    q( parent )
    {
        m_layout->setMargin( 5 );
        m_layout->setSpacing( 5 );
    }

    //! Frames.
    QList< FrameOnTape* > m_frames;
    //! Current frame.
    FrameOnTape * m_currentFrame;
    //! Layout.
    QHBoxLayout * m_layout;
    //! Parent.
    Tape * q;
}; // class TapePrivate
```

We need access to all frames, so we have a data member of type QList< FrameOnTape >, a auxiliary member that will hold a pointer to the currently selected frame, and our layout.

Trivial methods.

```
Tape::Tape( QWidget * parent )
    :    QWidget( parent )
    ,    d( new TapePrivate( this ) )
{
}

Tape::~Tape() noexcept
{
}

int
Tape::count() const
{
    return d->m_frames.count();
}

FrameOnTape *
Tape::frame( int idx ) const
{
    if( idx >= 1 && idx <= count() )
        return d->m_frames.at( idx - 1 );
    else
        return nullptr;
}

FrameOnTape *
Tape::currentFrame() const
{
    return d->m_currentFrame;
}

void
Tape::clear()
{
    const int c = count();

    for( int i = 1; i <= c; ++i )
        removeFrame( 1 );
}
```

Just will say that indexes in our API start from 1. Let's look at addFrame() method.

```
void
Tape::addFrame( const QImage & img )
{
    d->m_frames.append( new FrameOnTape( img, count() + 1, this ) );
    d->m_layout->addWidget( d->m_frames.back() );
```

```
    connect( d->m_frames.back(), &FrameOnTape::clicked,
        [this] ( int idx )
        {
            if( this->currentFrame() )
                this->currentFrame()->setCurrent( false );

            this->d->m_currentFrame = this->frame( idx );

            this->d->m_currentFrame->setCurrent( true );

            emit this->currentFrameChanged( idx );

            emit this->clicked( idx );
        } );

    adjustSize();
}
```

We created new FrameOnTape object, added it to the list and to the layout. Connected clicked() signal to do stuff for the current frame. And resized the entire widget. So, when a new frame will be added the tape will grow in width.

setCurrentFrame() is quite simple.

```
void
Tape::setCurrentFrame( int idx )
{
    if( idx >= 1 && idx <= count() )
    {
        if( d->m_currentFrame )
            d->m_currentFrame->setCurrent( false );

        d->m_currentFrame = frame( idx );
        d->m_currentFrame->setCurrent( true );

        emit currentFrameChanged( idx );
    }
    else
        d->m_currentFrame = nullptr;
}
```

And some magic in the removeFrame() method. I implemented it so when current frame deletes, a new one will become current, so we always will have selected frame.

```
void
Tape::removeFrame( int idx )
{
    if( idx <= count() )
    {
        d->m_layout->removeWidget( d->m_frames.at( idx - 1 ) );
        d->m_frames.at( idx - 1 )->deleteLater();

        if( d->m_frames.at( idx - 1 ) == d->m_currentFrame )
        {
            d->m_currentFrame = nullptr;

            if( idx > 1 )
            {
                d->m_currentFrame = d->m_frames.at( idx - 2 );
                d->m_currentFrame->setCurrent( true );

                emit currentFrameChanged( idx - 1 );
            }
            else if( idx < count() )
            {
                d->m_currentFrame = d->m_frames.at( idx );
                d->m_currentFrame->setCurrent( true );

                emit currentFrameChanged( idx + 1 );
            }
            else
                d->m_currentFrame = nullptr;
        }

        d->m_frames.removeAt( idx - 1 );

        adjustSize();
    }
```

```
}
```

Back | Contents | Next

View

Now we have everything to display GIF, we just need to combine all together. We have Frame class that will display the current frame, we have Tape class that will display tape of frames. We need a widget that will combine the current frame with tape, that we will set as a central widget of our main window.

Declaration.

```
#ifndef GIF_EDITOR_VIEW_HPP_INCLUDED
#define GIF_EDITOR_VIEW_HPP_INCLUDED

// Qt include.
#include <QWidget>
#include <QScopedPointer>


class Tape;
class Frame;


//
// View
//

class ViewPrivate;

//! View with current frame and tape with frames.
class View final
    :    public QWidget
{
    Q_OBJECT

public:
    View( QWidget * parent = nullptr );
    ~View() noexcept override;

    //! \return Tape.
    Tape * tape() const;
    //! \return Current frame.
    Frame * currentFrame() const;

private slots:
    //! Frame selected.
    void frameSelected( int idx );

private:
    Q_DISABLE_COPY( View )

    QScopedPointer< ViewPrivate > d;
}; // class View

#endif // GIF_EDITOR_VIEW_HPP_INCLUDED
```

No magic at all, all is simple. Private data class.

```
class ViewPrivate {
public:
    ViewPrivate( View * parent )
        :    m_tape( nullptr )
        ,    m_currentFrame( new Frame( QImage(), Frame::ResizeMode::FitToSize, parent ) )
        ,    q( parent )
    {
    }

    //! Tape.
    Tape * m_tape;
    //! Current frame.
    Frame * m_currentFrame;
    //! Parent.
    View * q;
}; // class ViewPrivate
```

And again all is simple. Current frame will occupy all available space and initialized with an empty image.

You will not believe how implementation is simple. And again Qt rocks. Look.

```
View::View( QWidget * parent )
    :    QWidget( parent )
    ,    d( new ViewPrivate( this ) )
{
    QVBoxLayout * layout = new QVBoxLayout( this );
    layout->setMargin( 0 );
    layout->addWidget( d->m_currentFrame );

    QScrollArea * scroll = new QScrollArea( this );
    scroll->setVerticalScrollBarPolicy( Qt::ScrollBarAlwaysOff );
    scroll->setHorizontalScrollBarPolicy( Qt::ScrollBarAlwaysOn );
    scroll->setMinimumHeight( 150 );
    scroll->setMaximumHeight( 150 );
    scroll->setWidgetResizable( true );

    d->m_tape = new Tape( scroll );
    scroll->setWidget( d->m_tape );

    layout->addWidget( scroll );

    connect( d->m_tape, &Tape::currentFrameChanged,
        this, &View::frameSelected );
}

View::~View() noexcept
{
}

Tape *
View::tape() const
{
    return d->m_tape;
}

Frame *
View::currentFrame() const
{
    return d->m_currentFrame;
}

void
View::frameSelected( int idx )
{
    d->m_currentFrame->setImage( d->m_tape->frame( idx )->image() );
}
```

This is really simple. Now we just need to create an object of View class and set it as a central widget to the main window. UI part is ready to display a GIF image. And in the next chapter we will open GIF with Magick++ and use API of our UI classes to set the sequence of frames.

Back | Contents | Next

Reading

In the UI we use QImage, but Magic++ works with its own Image class. We need conversion method from Magick::Image to QImage. Great place for this is in the main window private data class. We need to create an object of View class and set it as a central widget of the main window, and again the place for it is main window private data class. So let's look at it.

```
class MainWindowPrivate {
public:
    MainWindowPrivate( MainWindow * parent )
        :    m_view( new View( parent ) )
        ,    q( parent )
    {
    }

    //! Clear view.
    void clearView();
    //! Convert Magick::Image to QImage.
    QImage convert( const Magick::Image & img );

    //! Current file name.
    QString m_currentGif;
    //! Frames.
    std::vector< Magick::Image > m_frames;
    //! View.
    View * m_view;
    //! Parent.
    MainWindow * q;
}; // class MainWindowPrivate
```

We will work in the future with a sequence of Magick::Image objects for editing, so we have a data member for it. When GIf opened and we want to open a new one we should clear the current view, so let's look at the clearView() method.

```
void
MainWindowPrivate::clearView()
{
    m_frames.clear();
    m_view->tape()->clear();
    m_view->currentFrame()->setImage( QImage() );
}
```

No comments.

Conversion from Magick::Image to QImage is simple as well.

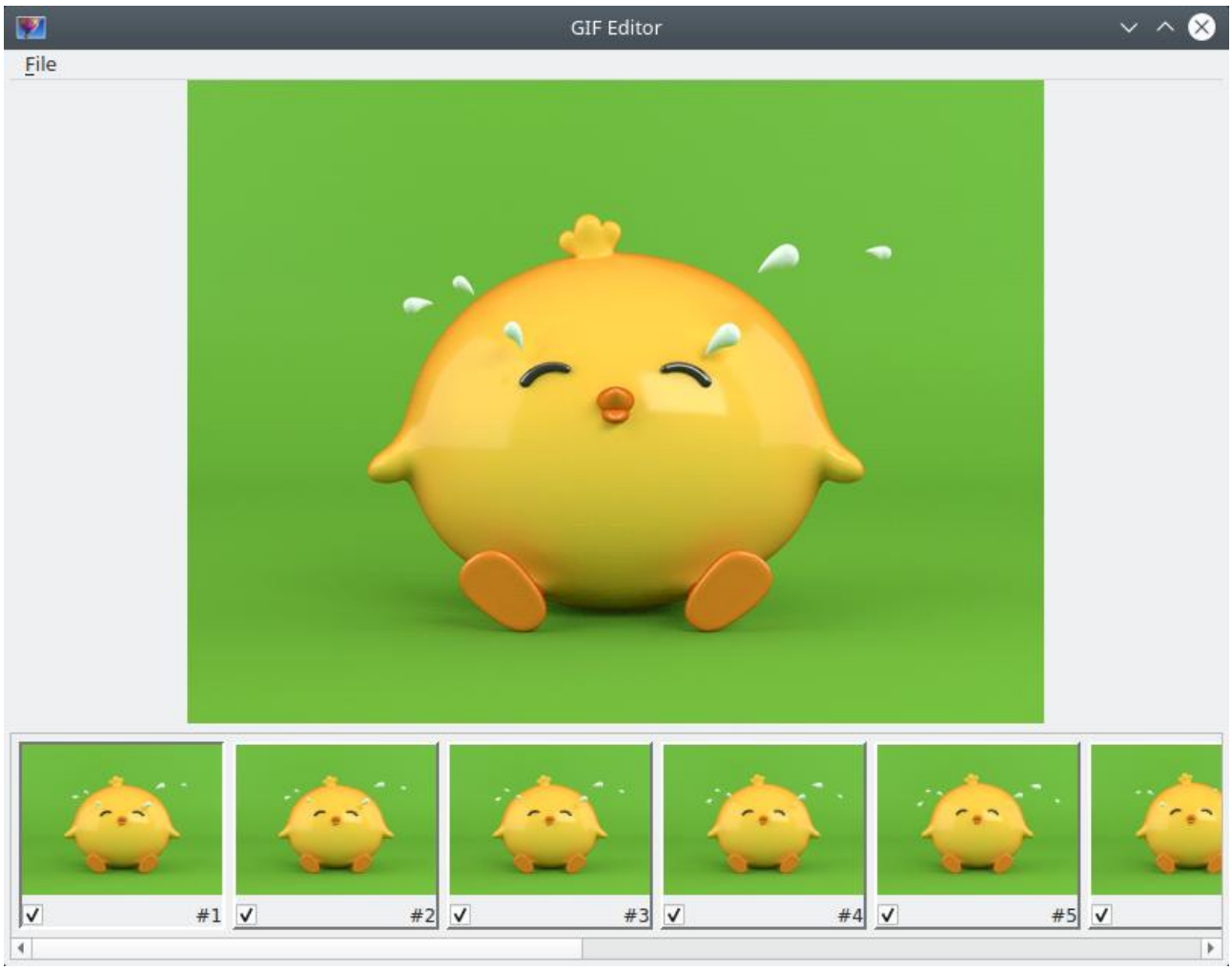
```
QImage
MainWindowPrivate::convert( const Magick::Image & img )
{
    QImage qimg( static_cast< int > ( img.columns() ),
        static_cast< int > ( img.rows() ), QImage::Format_RGB888 );
    const Magick::PixelPacket * pixels;
    Magick::ColorRGB rgb;

    for( int y = 0; y < qimg.height(); ++y)
    {
        pixels = img.getConstPixels( 0, y, static_cast< std::size_t > ( qimg.width() ), 1 );

        for( int x = 0; x < qimg.width(); ++x )
        {
            rgb = ( *( pixels + x ) );

            qimg.setPixel( x, y, QColor( static_cast< int> ( 255 * rgb.red() ),
                static_cast< int > ( 255 * rgb.green() ),
                static_cast< int > ( 255 * rgb.blue() ) ).rgb());
        }
    }
```

```
    return qimg;
}
```

Ok. We have empty openGif() slot in the MainWindow class. And to open GIF we should implement it.

```
void
MainWindow::openGif()
{
    const auto fileName = QFileDialog::getOpenFileName( this,
        tr( "Open GIF..." ), QString(), tr( "GIF (*.gif)" ) );

    if( !fileName.isEmpty() )
    {
        d->clearView();

        try {
            std::vector< Magick::Image > frames;

            Magick::readImages( &frames, fileName.toStdString() );

            Magick::coalesceImages( &d->m_frames, frames.begin(), frames.end() );

            std::for_each( d->m_frames.cbegin(), d->m_frames.cend(),
                [this] ( const Magick::Image & img )
                {
                    this->d->m_view->tape()->addFrame( this->d->convert( img ) );
                } );

            if( !d->m_frames.empty() )
                d->m_view->tape()->setCurrentFrame( 1 );
        }
        catch( const Magick::Exception & x )
        {
            d->clearView();

            QMessageBox::warning( this, tr( "Failed to open GIF..." ),
                QString::fromLocal8Bit( x.what() ) );
        }
    }
}
```

Simple, isn't it? You can believe, you can not believe, but editor now opens GIF images and displays all frames.



And this is less than 1K lines of code!

Back | Contents | Next

Saving

We can open GIF, we can navigate through the frames, we can uncheck some frames. Let's do saving of GIF with regards to unchecked frames. This is a basic of any GIF editor. First of all we should notify user that file was changed when he checks/unchecks frames. For it we should connect to Tape's checkStateChanged() signal. Let's do it in the constructor of MainWindow.

```
connect( d->m_view->tape(), &Tape::checkStateChanged,
        this, &MainWindow::frameChecked );
```

I changed a little checkStateChanged() signal, so it looks like.

```
//! Frame checked/unchecked.
void checkStateChanged( int idx, bool checked );
```

MainWindow's frameChecked() slot is simple.

```
void
MainWindow::frameChecked( int, bool )
{
    setWindowModified( true );
}
```

We just notifying a user that GIF was changed. Let's have a look at saveGifAs() slot

```
void
MainWindow::saveGifAs()
{
    auto fileName = QFileDialog::getSaveFileName( this,
        tr( "Choose file to save to..." ), QString(), tr( "GIF (*.gif)" ) );

    if( !fileName.isEmpty() )
    {
        if( !fileName.endsWith( QLatin1String( ".gif" ), Qt::CaseInsensitive ) )
            fileName.append( QLatin1String( ".gif" ) );

        d->m_currentGif = fileName;

        QFileInfo info( fileName );

        setWindowTitle( tr( "GIF Editor - %1[*]" ).arg( info.fileName() ) );

        saveGif();
    }
}
```

So the main work is done in saveGif() slot.

```
void
MainWindow::saveGif()
{
    std::vector< Magick::Image > toSave;

    for( int i = 0; i < d->m_view->tape()->count(); ++i )
    {
        if( d->m_view->tape()->frame( i + 1 )->isChecked() )
            toSave.push_back( d->m_frames.at( static_cast< std::size_t > ( i ) ) );
    }

    if( !toSave.empty() )
    {
        try {
```

```
            Magick::writeImages( toSave.begin(), toSave.end(), d->m_currentGif.toStdString() );

            d->m_view->tape()->removeUnchecked();

            d->m_frames = toSave;

            setWindowModified( false );
        }
        catch( const Magick::Exception & x )
        {
            QMessageBox::warning( this, tr( "Failed to save GIF..." ),
                QString::fromLocal8Bit( x.what() ) );
        }
    }
    else
    {
        QMessageBox::information( this, tr( "Can't save GIF..." ),
            tr( "Can't save GIF image with no frames." ) );
    }
}
```

We just iterating through the frames and checking if they checked, saving all checked frames to the GIF, and if all is ok we updating UI. I added a new method to the Tape class to simplify this process.

```
void
Tape::removeUnchecked()
{
    const int c = count();
    int removed = 0;

    for( int i = 1; i <= c; ++i )
    {
        if( !frame( i - removed )->isChecked() )
        {
            removeFrame( i - removed );

            ++removed;
        }
        else
            frame( i - removed )->setCounter( i - removed );
    }
}
```

We just removing unchecked frames and updating counter.

To be more user friendly I changed a little openGif() slot.

```
void
MainWindow::openGif()
{
    const auto fileName = QFileDialog::getOpenFileName( this,
        tr( "Open GIF..." ), QString(), tr( "GIF (*.gif)" ) );

    if( !fileName.isEmpty() )
    {
        if( isWindowModified() )
        {
            const auto btn = QMessageBox::question( this,
                tr( "GIF was changed..." ),
                tr( "\"%1\" was changed.\n"
                    "Do you want to save it?" ) );

            if( btn == QMessageBox::Yes )
                saveGif();
        }

        d->clearView();

        setWindowModified( false );

        setWindowTitle( tr( "GIF Editor" ) );

        d->m_view->currentFrame()->setImage( QImage() );

        try {
            std::vector< Magick::Image > frames;
```

```
            Magick::readImages( &frames, fileName.toStdString() );

            Magick::coalesceImages( &d->m_frames, frames.begin(), frames.end() );

            QFileInfo info( fileName );

            setWindowTitle( tr( "GIF Editor - %1[*]" ).arg( info.fileName() ) );

            d->m_currentGif = fileName;

            std::for_each( d->m_frames.cbegin(), d->m_frames.cend(),
                [this] ( const Magick::Image & img )
                {
                    this->d->m_view->tape()->addFrame( this->d->convert( img ) );
                } );

            if( !d->m_frames.empty() )
                d->m_view->tape()->setCurrentFrame( 1 );
        }
        catch( const Magick::Exception & x )
        {
            d->clearView();

            QMessageBox::warning( this, tr( "Failed to open GIF..." ),
                QString::fromLocal8Bit( x.what() ) );
        }
    }
}
```

Just added a question, set window's title and a small exception safety.

Back | Contents | Next

What else

As said for the first Alpha version we need a crop function. The user should be able to enable cropping from some action in the tab bar or menu. When the crop is enabled user should be able to draw by dragging mouse cursor a rectangle on the current frame. After releasing the mouse cursor user should be able to adjust drawn rectangle, and on Enter key pressing crop function should crop all frames in the GIF. Interesting, isn't it?

Back | Contents | Next

Crop

As said we need to implement a crop function. First of all we need to implement a widget that will draw a rectangle on top of the current frame, that will show the crop region. I spent on this widget not so little time, a day, this is because this widget is very custom and complicated in implementation, not hard but complicated. There are a lot of calculations of regions of handles for adjusting the rectangle, mouse cursor handling, cursor overriding, etc. Let's have a look at this widget.

Widget

Declaration.

```
#ifndef GIF_EDITOR_CROP_HPP_INCLUDED
#define GIF_EDITOR_CROP_HPP_INCLUDED

// Qt include.
#include <QWidget>
#include <QScopedPointer>

class Frame;


//
// CropFrame
//

class CropFramePrivate;

//! Crop frame.
class CropFrame final
    :    public QWidget
{
    Q_OBJECT

public:
    CropFrame( Frame * parent = nullptr );
    ~CropFrame() noexcept override;

    //! \return Crop rectangle.
    QRect cropRect() const;

public slots:
    //! Start.
    void start();
    //! Stop.
    void stop();

private slots:
    //! Frame resized.
    void frameResized();

protected:
    void paintEvent( QPaintEvent * ) override;
    void mousePressEvent( QMouseEvent * e ) override;
    void mouseMoveEvent( QMouseEvent * e ) override;
    void mouseReleaseEvent( QMouseEvent * e ) override;
    void enterEvent( QEvent * e ) override;
    void leaveEvent( QEvent * e ) override;

private:
    Q_DISABLE_COPY( CropFrame )

    QScopedPointer< CropFramePrivate > d;
}; // class CropFrame

#endif // GIF_EDITOR_CROP_HPP_INCLUDED
```

API is simple, but let's look at what is under the hood.

Crop rectangle will have handles to change the geometry of the rectangle. And in the

code I defined a constant to store the size for it.

```
//! Size of the handle to change geometry of selected region.
static const int c_handleSize = 15;
```

Private data class.

```
class CropFramePrivate {
public:
    CropFramePrivate( CropFrame * parent, Frame * toObserve )
        :    m_started( false )
        ,    m_nothing( true )
        ,    m_clicked( false )
        ,    m_hovered( false )
        ,    m_cursorOverriden( false )
        ,    m_handle( Handle::Unknown )
        ,    m_frame( toObserve )
        ,    q( parent )
    {
    }

    enum class Handle {
        Unknown,
        TopLeft,
        Top,
        TopRight,
        Right,
        BottomRight,
        Bottom,
        BottomLeft,
        Left
    }; // enum class Handle

    //! Bound point to available space.
    QPoint boundToAvailable( const QPoint & p ) const;
    //! Bound left top point to available space.
    QPoint boundLeftTopToAvailable( const QPoint & p ) const;
    //! Check and override cursor if necessary.
    void checkAndOverrideCursor( Qt::CursorShape shape );
    //! Override cursor.
    void overrideCursor( const QPoint & pos );
    //! Resize crop.
    void resize( const QPoint & pos ) ;
    //! \return Cropped rect.
    QRect cropped( const QRect & full ) const;
    //! \return Is handles should be outside selected rect.
    bool isHandleOutside() const
    {
        return ( qAbs( m_selected.width() ) / 3 < c_handleSize + 1 ||
            qAbs( m_selected.height() ) / 3 < c_handleSize + 1 );
    }
    //! \return Top-left handle rect.
    QRect topLeftHandleRect() const
    {
        return ( isHandleOutside() ?
            QRect( m_selected.x() - ( m_selected.width() > 0 ? c_handleSize : 0 ),
                m_selected.y() - ( m_selected.height() > 0 ? c_handleSize : 0 ),
                c_handleSize, c_handleSize ) :
            QRect( m_selected.x() - ( m_selected.width() > 0 ? 0 : c_handleSize ),
                m_selected.y() - ( m_selected.height() > 0 ? 0 : c_handleSize ),
                c_handleSize, c_handleSize ) );
    }
    //! \return Top-right handle rect.
    QRect topRightHandleRect() const
    {
        return ( isHandleOutside() ?
            QRect( m_selected.x() + m_selected.width() - 1 -
                    ( m_selected.width() > 0 ? 0 : c_handleSize ),
                m_selected.y() - ( m_selected.height() > 0 ? c_handleSize : 0 ),
                c_handleSize, c_handleSize ) :
            QRect( m_selected.x() + m_selected.width() -
                    ( m_selected.width() > 0 ? c_handleSize : 0 ) - 1,
                m_selected.y() - ( m_selected.height() > 0 ? 0 : c_handleSize ),
                c_handleSize, c_handleSize ) );
    }
    //! \return Bottom-right handle rect.
    QRect bottomRightHandleRect() const
    {
        return ( isHandleOutside() ?
            QRect( m_selected.x() + m_selected.width() - 1 -
                    ( m_selected.width() > 0 ? 0 : c_handleSize ),
                m_selected.y() + m_selected.height() -
                    ( m_selected.height() > 0 ? 0 : c_handleSize ),
                c_handleSize, c_handleSize ) :
            QRect( m_selected.x() + m_selected.width() -
```

```
                ( m_selected.width() > 0 ? c_handleSize : 0 ) - 1,
            m_selected.y() + m_selected.height() -
                ( m_selected.height() > 0 ? c_handleSize : 0 )  - 1,
            c_handleSize, c_handleSize ) );
}
//! \return Bottom-left handle rect.
QRect bottomLeftHandleRect() const
{
    return ( isHandleOutside() ?
        QRect( m_selected.x() - ( m_selected.width() > 0 ? c_handleSize : 0 ),
            m_selected.y() + m_selected.height() - 1 -
                ( m_selected.height() > 0 ? 0 : c_handleSize),
            c_handleSize, c_handleSize ) :
        QRect( m_selected.x() - ( m_selected.width() > 0 ? 0 : c_handleSize),
            m_selected.y() + m_selected.height() -
                ( m_selected.height() > 0 ? c_handleSize : 0 ) - 1,
            c_handleSize, c_handleSize ) );
}
//! \return Y handle width.
int yHandleWidth() const
{
    const int w = m_selected.width() - 1;

    return ( isHandleOutside() ? w :
        w - 2 * c_handleSize - ( w - 2 * c_handleSize ) / 3 );
}
//! \return X handle height.
int xHandleHeight() const
{
    const int h = m_selected.height() - 1;

    return ( isHandleOutside() ? h :
        h - 2 * c_handleSize - ( h - 2 * c_handleSize ) / 3 );
}
//! \return Y handle x position.
int yHandleXPos() const
{
    return ( m_selected.x() + ( m_selected.width() - yHandleWidth() ) / 2 );
}
//! \return X handle y position.
int xHandleYPos() const
{
    return ( m_selected.y() + ( m_selected.height() - xHandleHeight() ) / 2 );
}
//! \return Top handle rect.
QRect topHandleRect() const
{
    return ( isHandleOutside() ?
        QRect( yHandleXPos(), m_selected.y() - ( m_selected.height() > 0 ? c_handleSize : 0 ),
            yHandleWidth(), c_handleSize ) :
        QRect( yHandleXPos(), m_selected.y() - ( m_selected.height() > 0 ? 0 : c_handleSize ),
            yHandleWidth(), c_handleSize ) );
}
//! \return Bottom handle rect.
QRect bottomHandleRect() const
{
    return ( isHandleOutside() ?
        QRect( yHandleXPos(), m_selected.y() + m_selected.height() - 1 -
                ( m_selected.height() > 0 ? 0 : c_handleSize ),
            yHandleWidth(), c_handleSize ) :
        QRect( yHandleXPos(), m_selected.y() + m_selected.height() - 1 -
                ( m_selected.height() > 0 ? c_handleSize : 0 ),
            yHandleWidth(), c_handleSize ) );
}
//! \return Left handle rect.
QRect leftHandleRect() const
{
    return ( isHandleOutside() ?
        QRect( m_selected.x() - ( m_selected.width() > 0 ? c_handleSize : 0 ),
            xHandleYPos(), c_handleSize, xHandleHeight() ) :
        QRect( m_selected.x() - ( m_selected.width() > 0 ? 0 : c_handleSize ),
            xHandleYPos(), c_handleSize, xHandleHeight() ) );
}
//! \return Right handle rect.
QRect rightHandleRect() const
{
    return ( isHandleOutside() ?
        QRect( m_selected.x() + m_selected.width() - 1 -
                ( m_selected.width() > 0 ? 0 : c_handleSize ),
            xHandleYPos(), c_handleSize, xHandleHeight() ) :
        QRect( m_selected.x() + m_selected.width() - 1 -
                ( m_selected.width() > 0 ? c_handleSize : 0 ),
            xHandleYPos(), c_handleSize, xHandleHeight() ) );
}

//! Selected rectangle.
QRect m_selected;
//! Available rectangle.
QRect m_available;
//! Mouse pos.
```

```
    QPoint m_mousePos;
    //! Selecting started.
    bool m_started;
    //! Nothing selected yet.
    bool m_nothing;
    //! Clicked.
    bool m_clicked;
    //! Hover entered.
    bool m_hovered;
    //! Cursor overriden.
    bool m_cursorOverriden;
    //! Current handle.
    Handle m_handle;
    //! Frame to observe resize event.
    Frame * m_frame;
    //! Parent.
    CropFrame * q;
}; // class CropFramePrivate
```

Uhh, so many methods... I defined some methods in the class, these methods just returns rectangles of the handles, you could understand it from their names.

We will resize, move selection rectangle, and we don't want this rectangle to go out of frame boundary. And for this task we have two auxiliary methods.

```
QPoint
CropFramePrivate::boundToAvailable( const QPoint & p ) const
{
    QPoint ret = p;

    if( p.x() < m_available.x() )
        ret.setX( m_available.x() );
    else if( p.x() > m_available.x() + m_available.width() - 1 )
        ret.setX( m_available.x() + m_available.width() - 1 );

    if( p.y() < m_available.y() )
        ret.setY( m_available.y() );
    else if( p.y() > m_available.y() + m_available.height() - 1 )
        ret.setY( m_available.y() + m_available.height() - 1 );

    return ret;
}

QPoint
CropFramePrivate::boundLeftTopToAvailable( const QPoint & p ) const
{
    QPoint ret = p;

    if( p.x() < m_available.x() )
        ret.setX( m_available.x() );
    else if( p.x() > m_available.x() + m_available.width() - m_selected.width() - 1)
        ret.setX( m_available.x() + m_available.width() - m_selected.width() - 1 );

    if( p.y() < m_available.y() )
        ret.setY( m_available.y() );
    else if( p.y() > m_available.y() + m_available.height() - m_selected.height() - 1 )
        ret.setY( m_available.y() + m_available.height() - m_selected.height() - 1 );

    return ret;
}
```

When the user moves the mouse cursor over the widget, different regions we need to override cursor to help the user understand what he can do. Auxiliary methods to override cursor.

```
void
CropFramePrivate::checkAndOverrideCursor( Qt::CursorShape shape )
{
    if( QApplication::overrideCursor() )
    {
        if( *QApplication::overrideCursor() != QCursor( shape ) )
        {
            if( m_cursorOverriden )
                QApplication::restoreOverrideCursor();
            else
                m_cursorOverriden = true;
```

```
            QApplication::setOverrideCursor( QCursor( shape ) );
        }
    }
    else
    {
        m_cursorOverriden = true;

        QApplication::setOverrideCursor( QCursor( shape ) );
    }
}

void
CropFramePrivate::overrideCursor( const QPoint & pos )
{
    if( topLeftHandleRect().contains( pos ) )
    {
        m_handle = CropFramePrivate::Handle::TopLeft;
        checkAndOverrideCursor( Qt::SizeFDiagCursor );
    }
    else if( bottomRightHandleRect().contains( pos ) )
    {
        m_handle = CropFramePrivate::Handle::BottomRight;
        checkAndOverrideCursor( Qt::SizeFDiagCursor );
    }
    else if( topRightHandleRect().contains( pos ) )
    {
        m_handle = CropFramePrivate::Handle::TopRight;
        checkAndOverrideCursor( Qt::SizeBDiagCursor );
    }
    else if( bottomLeftHandleRect().contains( pos ) )
    {
        m_handle = CropFramePrivate::Handle::BottomLeft;
        checkAndOverrideCursor( Qt::SizeBDiagCursor );
    }
    else if( topHandleRect().contains( pos ) )
    {
        m_handle = CropFramePrivate::Handle::Top;
        checkAndOverrideCursor( Qt::SizeVerCursor );
    }
    else if( bottomHandleRect().contains( pos ) )
    {
        m_handle = CropFramePrivate::Handle::Bottom;
        checkAndOverrideCursor( Qt::SizeVerCursor );
    }
    else if( leftHandleRect().contains( pos ) )
    {
        m_handle = CropFramePrivate::Handle::Left;
        checkAndOverrideCursor( Qt::SizeHorCursor );
    }
    else if( rightHandleRect().contains( pos ) )
    {
        m_handle = CropFramePrivate::Handle::Right;
        checkAndOverrideCursor( Qt::SizeHorCursor );
    }
    else if( m_selected.contains( pos ) )
    {
        m_handle = CropFramePrivate::Handle::Unknown;
        checkAndOverrideCursor( Qt::SizeAllCursor );
    }
    else if( m_cursorOverriden )
    {
        m_cursorOverriden = false;
        m_handle = CropFramePrivate::Handle::Unknown;
        QApplication::restoreOverrideCursor();
    }
}
```

When user presses and moves handle selection rectangle should resize, so the method for it.

```
void
CropFramePrivate::resize( const QPoint & pos )
{
    switch( m_handle )
    {
        case CropFramePrivate::Handle::Unknown :
            m_selected.moveTo( boundLeftTopToAvailable(
                m_selected.topLeft() - m_mousePos + pos ) );
        break;

        case CropFramePrivate::Handle::TopLeft :
            m_selected.setTopLeft( boundToAvailable( m_selected.topLeft() -
                m_mousePos + pos ) );
        break;
```

```
        case CropFramePrivate::Handle::TopRight :
            m_selected.setTopRight( boundToAvailable( m_selected.topRight() -
                m_mousePos + pos ) );
        break;

        case CropFramePrivate::Handle::BottomRight :
            m_selected.setBottomRight( boundToAvailable( m_selected.bottomRight() -
                m_mousePos + pos ) );
        break;

        case CropFramePrivate::Handle::BottomLeft :
            m_selected.setBottomLeft( boundToAvailable( m_selected.bottomLeft() -
                m_mousePos + pos ) );
        break;

        case CropFramePrivate::Handle::Top :
            m_selected.setTop( boundToAvailable( QPoint( m_selected.left(), m_selected.top() ) -
                m_mousePos + pos ).y() );
        break;

        case CropFramePrivate::Handle::Bottom :
            m_selected.setBottom( boundToAvailable( QPoint( m_selected.left(),
                m_selected.bottom() ) - m_mousePos + pos ).y() );
        break;

        case CropFramePrivate::Handle::Left :
            m_selected.setLeft( boundToAvailable( QPoint( m_selected.left(),
                m_selected.top() ) - m_mousePos + pos ).x() );
        break;

        case CropFramePrivate::Handle::Right :
            m_selected.setRight( boundToAvailable( QPoint( m_selected.right(),
                m_selected.top() ) - m_mousePos + pos ).x() );
        break;
    }

    m_mousePos = pos;
}
```

We can draw a crop rectangle on the scaled frame, but for cropping we need to know rectangle to crop in the original frame's coordinates.

```
QRect
CropFramePrivate::cropped( const QRect & full ) const
{
    const auto oldR = m_available;

    const qreal xRatio = static_cast< qreal > ( full.width() ) /
        static_cast< qreal > ( oldR.width() );
    const qreal yRatio = static_cast< qreal > ( full.height() ) /
        static_cast< qreal > ( oldR.height() );

    QRect r;

    if( !m_nothing )
    {
        const auto x = static_cast< int >( ( m_selected.x() - oldR.x() ) * xRatio ) +
            full.x();
        const auto y = static_cast< int >( ( m_selected.y() - oldR.y() ) * yRatio ) +
            full.y();
        const auto dx = full.bottomRight().x() - static_cast< int >(
            ( oldR.bottomRight().x() - m_selected.bottomRight().x() ) * xRatio );
        const auto dy = full.bottomRight().y() - static_cast< int >(
            ( oldR.bottomRight().y() - m_selected.bottomRight().y() ) * yRatio );

        r.setTopLeft( QPoint( x, y ) );
        r.setBottomRight( QPoint( dx, dy ) );
    }

    return r;
}
```

You can ask how it's possible to write all these methods first and only then implement methods of the widget? I guess that this is impossible. I wrote a skeleton of widget and step by step wrote code, so these private data methods were born from time to time when they were needed. Developing is an iterative process. With some experience you will come to it, but I believe that you are an experienced C++ developer and just want to quickly look at working methods to develop on Qt's widgets. Let's go.

The widget as is very simple with all these auxiliary methods, have a look.

```
CropFrame::CropFrame( Frame * parent )
    :    QWidget( parent )
    ,    d( new CropFramePrivate( this, parent ) )
{
    setAutoFillBackground( false );
    setAttribute( Qt::WA_TranslucentBackground, true );
    setMouseTracking( true );

    d->m_available = parent->imageRect();

    connect( d->m_frame, &Frame::resized,
        this, &CropFrame::frameResized );
}

CropFrame::~CropFrame() noexcept
{
    if( d->m_cursorOverriden )
        QApplication::restoreOverrideCursor();

    if( d->m_hovered )
        QApplication::restoreOverrideCursor();
}

QRect
CropFrame::cropRect() const
{
    return d->cropped( d->m_frame->image().rect() );
}

void
CropFrame::start()
{
    d->m_started = true;
    d->m_nothing = true;

    update();
}

void
CropFrame::stop()
{
    d->m_started = false;

    update();
}
```

I added to Frame class resized() signal to handle resizing and correctly resize selection region.

```
void
CropFrame::frameResized()
{
    d->m_selected = d->cropped( d->m_frame->imageRect() );

    setGeometry( QRect( 0, 0, d->m_frame->width(), d->m_frame->height() ) );

    d->m_available = d->m_frame->imageRect();

    update();
}
```

Painting of our widget.

```
void
CropFrame::paintEvent( QPaintEvent * )
{
    static const QColor dark( 0, 0, 0, 100 );

    QPainter p( this );
    p.setPen( Qt::black );
    p.setBrush( dark );

    if( d->m_started && !d->m_nothing )
    {
        QPainterPath path;
        path.addRect( QRectF( d->m_available ).adjusted( 0, 0, -1, -1 ) );
```

```
        if( d->m_available != d->m_selected )
        {
            QPainterPath spath;
            spath.addRect( QRectF( d->m_selected ).adjusted( 0, 0, -1, -1 ) );
            path = path.subtracted( spath );
        }
        else
            p.setBrush( Qt::transparent );

        p.drawPath( path );
    }

    p.setBrush( Qt::transparent );

    if( d->m_started && !d->m_clicked && !d->m_nothing &&
        d->m_handle == CropFramePrivate::Handle::Unknown )
    {
        p.drawRect( d->topLeftHandleRect() );
        p.drawRect( d->topRightHandleRect() );
        p.drawRect( d->bottomRightHandleRect() );
        p.drawRect( d->bottomLeftHandleRect() );
    }
    else if( d->m_started && !d->m_nothing &&
        d->m_handle != CropFramePrivate::Handle::Unknown )
    {
        switch( d->m_handle )
        {
            case CropFramePrivate::Handle::TopLeft :
                p.drawRect( d->topLeftHandleRect() );
            break;

            case CropFramePrivate::Handle::TopRight :
                p.drawRect( d->topRightHandleRect() );
            break;

            case CropFramePrivate::Handle::BottomRight :
                p.drawRect( d->bottomRightHandleRect() );
            break;

            case CropFramePrivate::Handle::BottomLeft :
                p.drawRect( d->bottomLeftHandleRect() );
            break;

            case CropFramePrivate::Handle::Top :
                p.drawRect( d->topHandleRect() );
            break;

            case CropFramePrivate::Handle::Bottom :
                p.drawRect( d->bottomHandleRect() );
            break;

            case CropFramePrivate::Handle::Left :
                p.drawRect( d->leftHandleRect() );
            break;

            case CropFramePrivate::Handle::Right :
                p.drawRect( d->rightHandleRect() );
            break;

            default:
                break;
        }
    }
}
```

The behaviour of crop region is like in Gimp. When user has drawn rectangle on mouse release he will see a transparent rectangle with darkening semi-transparent background on a non-selected region and corner handles. To access the top, bottom, left and right handles user should move the mouse cursor in the centres of the edges. And when the mouse cursor is on a handle, only this handle will be drawn and the mouse cursor will be overridden, like in Gimp.

And mouse handling.

```
void
CropFrame::mousePressEvent( QMouseEvent * e )
{
    if( e->button() == Qt::LeftButton )
    {
        d->m_clicked = true;
```

```
        if( !d->m_cursorOverriden )
            d->m_selected.setTopLeft( d->boundToAvailable( e->pos() ) );
        else
            d->m_mousePos = e->pos();

        update();

        e->accept();
    }
    else
        e->ignore();
}

void
CropFrame::mouseMoveEvent( QMouseEvent * e )
{
    if( d->m_clicked )
    {
        if ( !d->m_cursorOverriden )
        {
            d->m_selected.setBottomRight( d->boundToAvailable( e->pos() ) );

            d->m_nothing = false;
        }
        else
            d->resize( e->pos() );

        update();

        e->accept();
    }
    else if( !d->m_hovered )
    {
        d->m_hovered = true;

        QApplication::setOverrideCursor( QCursor( Qt::CrossCursor ) );
    }
    else if( d->m_hovered && !d->m_nothing )
    {
        d->overrideCursor( e->pos() );

        update();
    }
    else
        e->ignore();
}

void
CropFrame::mouseReleaseEvent( QMouseEvent * e )
{
    d->m_clicked = false;

    if( e->button() == Qt::LeftButton )
    {
        d->m_selected = d->m_selected.normalized();

        update();

        e->accept();
    }
    else
        e->ignore();
}

void
CropFrame::enterEvent( QEvent * e )
{
    if( d->m_started )
    {
        d->m_hovered = true;

        QApplication::setOverrideCursor( QCursor( Qt::CrossCursor ) );

        e->accept();
    }
    else
        e->ignore();
}

void
CropFrame::leaveEvent( QEvent * e )
{
    if( d->m_started )
    {
        d->m_hovered = false;

        QApplication::restoreOverrideCursor();

        e->accept();
```

```
    }
    else
        e->ignore();
}
```

Integrating crop frame into view

In the View private data class I added pointer to CropFrame widget.

```
//! Crop.
CropFrame * m_crop;
```

And two slots to start and stop crop operation.

```
void
View::startCrop()
{
    if( !d->m_crop )
    {
        d->m_crop = new CropFrame( d->m_currentFrame );
        d->m_crop->setGeometry( QRect( 0, 0,
            d->m_currentFrame->width(), d->m_currentFrame->height() ) );
        d->m_crop->show();
        d->m_crop->raise();
        d->m_crop->start();
    }
}

void
View::stopCrop()
{
    if( d->m_crop )
    {
        d->m_crop->stop();
        d->m_crop->deleteLater();
        d->m_crop = nullptr;
    }
}
```

So crop frame will be the same size as current frame widget and will be placed on top of it.

To access crop region I added a method.

```
QRect
View::cropRect() const
{
    if( d->m_crop )
        return d->m_crop->cropRect();
    else
        return QRect();
}
```

Nothing more.

Cropping

We need menu and tool bar to start, finish and cancel crop operation, so in MainWindow's constructor we added.

```
d->m_crop = new QAction( QIcon( ":/img/transform-crop.png" ),
    tr( "Crop" ), this );
```

```
d->m_crop->setShortcut( tr( "Ctrl+C" ) );
d->m_crop->setShortcutContext( Qt::ApplicationShortcut );
d->m_crop->setCheckable( true );
d->m_crop->setChecked( false );
d->m_crop->setEnabled( false );

d->m_applyEdit = new QAction( this );
d->m_applyEdit->setShortcut( Qt::Key_Return );
d->m_applyEdit->setShortcutContext( Qt::ApplicationShortcut );
d->m_applyEdit->setEnabled( false );

d->m_cancelEdit = new QAction( this );
d->m_cancelEdit->setShortcut( Qt::Key_Escape );
d->m_cancelEdit->setShortcutContext( Qt::ApplicationShortcut );
    d->m_cancelEdit->setEnabled( false );

addAction( d->m_applyEdit );
addAction( d->m_cancelEdit );

connect( d->m_crop, &QAction::triggered, this, &MainWindow::crop );
connect( d->m_applyEdit, &QAction::triggered, this, &MainWindow::applyEdit );
connect( d->m_cancelEdit, &QAction::triggered, this, &MainWindow::cancelEdit );

auto edit = menuBar()->addMenu( tr( "&Edit" ) );
edit->addAction( d->m_crop );

auto editToolBar = new QToolBar( tr( "Edit" ), this );
editToolBar->addAction( d->m_crop );

addToolBar( Qt::LeftToolBarArea, editToolBar );
```

Reaction on triggering crop action is simple.

```
void
MainWindow::crop( bool on )
{
    if( on )
    {
        d->enableFileActions( false );

        d->m_editMode = MainWindowPrivate::EditMode::Crop;

        d->m_view->startCrop();
    }
    else
    {
        d->m_view->stopCrop();

        d->m_editMode = MainWindowPrivate::EditMode::Unknow;

        d->enableFileActions();
    }
}
```

Where.

```
//! Enable file actions.
void enableFileActions( bool on = true )
{
    m_save->setEnabled( on );
    m_saveAs->setEnabled( on );
    m_open->setEnabled( on );

    m_applyEdit->setEnabled( !on );
    m_cancelEdit->setEnabled( !on );
}
```

Cancelling and applying crop operation.

```
void
MainWindow::cancelEdit()
{
    switch( d->m_editMode )
    {
        case MainWindowPrivate::EditMode::Crop :
        {
```

```
            d->m_view->stopCrop();

            d->enableFileActions();

            d->m_crop->setChecked( false );

            d->m_editMode = MainWindowPrivate::EditMode::Unknow;
        }
            break;

        default :
            break;
    }
}

void
MainWindow::applyEdit()
{
    switch( d->m_editMode )
    {
        case MainWindowPrivate::EditMode::Crop :
        {
            const auto rect = d->m_view->cropRect();

            if( !rect.isNull() && rect != d->m_view->currentFrame()->image().rect() )
            {
                QVector< int > unchecked;

                for( int i = 1; i <= d->m_view->tape()->count(); ++i )
                {
                    if( !d->m_view->tape()->frame( i )->isChecked() )
                        unchecked.append( i );
                }

                try {
                    auto tmpFrames = d->m_frames;

                    std::for_each( tmpFrames.begin(), tmpFrames.end(),
                        [&rect] ( auto & frame )
                        {
                            frame.crop( Magick::Geometry( rect.width(), rect.height(),
                                rect.x(), rect.y() ) );
                            frame.repage();
                        } );

                    const auto current = d->m_view->tape()->currentFrame()->counter();
                    d->m_view->tape()->clear();
                    d->m_frames = tmpFrames;

                    d->initTape();

                    d->m_view->tape()->setCurrentFrame( current );

                    for( const auto & i : qAsConst( unchecked ) )
                        d->m_view->tape()->frame( i )->setChecked( false );

                    setWindowModified( true );

                    cancelEdit();
                }
                catch( const Magick::Exception & x )
                {
                    cancelEdit();

                    QMessageBox::warning( this, tr( "Failed to crop GIF..." ),
                        QString::fromLocal8Bit( x.what() ) );
                }
            }
            else
                cancelEdit();
        }
            break;

        default :
            break;
    }
}
```

That's all. Now our editor can crop GIFs. So first Alpha version almost done.

Back | Contents | Next

About

And the last step. Let's add Help menu with about dialogues.

In the MainWindow constructor.

```
auto help = menuBar()->addMenu( tr( "&Help" ) );
help->addAction( QIcon( ":/img/icon_22x22.png" ), tr( "About" ),
    this, &MainWindow::about );
help->addAction( QIcon( ":/img/qt.png" ), tr( "About Qt" ),
    this, &MainWindow::aboutQt );
```

And slots.

```
void
MainWindow::about()
{
    QMessageBox::about( this, tr( "About GIF editor" ),
        tr( "GIF editor.\n\n"
            "Author - Igor Mironchik (igor.mironchik at gmail dot com).\n\n"
            "Copyright (c) 2018 Igor Mironchik.\n\n"
            "Licensed under GNU GPL 3.0." ) );
}

void
MainWindow::aboutQt()
{
    QMessageBox::aboutQt( this );
}
```

Have a good day!

Back | Contents | Next

Chapter 2

In this chapter I will show how to work with QCamera, QAbstractVideoSurface. How to detect motion with OpenCV, capture with QCamera frames and store them in some place on the disk. This chapter is based on the real project SecurityCam that places on GitHub https://github.com/igormironchik/security-cam

In this chapter I will describe only the most interesting parts of the code of SecurityCam, I won't show you how I save configuration file, how I organized configuration of the application. Only a few words, that for reading/saving configuration file I used cfgfile. This is Open Source library for reading and saving configurations. It places on GitHub https://github.com/igormironchik/cfgfile

SecurityCam is an application that connects to USB camera and tries to detect motions in the frame, as soon as motion is detected camera starts to capture images and store them in the configured folder with yyy/MM/dd hierarchy. SecurityCam can do clean at the configured time and delete folders with images that stored more than N days.

The window of the application displays stream from the camera and on closing minimizes to tray, so the application works in the background and protects the entrusted territory.

Back | Contents | Next

View

In this project we want to detect motion in the frame, so we need to have access to each frame in the camera's stream. So the only solution is QAbstractVideoSurface. And we want to display stream from a camera in some case of viewfinder. We need to tie together QAbstractVideoSurface and any viewfinder. I see only one solution - is to transmit QImage with the current frame from QAbstractVideoSurface to custom viewfinder, that will display the current frame.

So let's do such a view finder.

```
#ifndef SECURITYCAM_VIEW_HPP_INCLUDED
#define SECURITYCAM_VIEW_HPP_INCLUDED

// Qt include.
#include <QWidget>
#include <QScopedPointer>


namespace SecurityCam {

//
// View
//

class ViewPrivate;

//! View of the video data from the camera.
class View final
    :    public QWidget
{
    Q_OBJECT

public:
    explicit View( QWidget * parent );
    ~View() noexcept override;

public slots:
    //! Draw image.
    void draw( const QImage & image );

protected:
    void paintEvent( QPaintEvent * ) override;
    void resizeEvent( QResizeEvent * e ) override;

private:
    Q_DISABLE_COPY( View )

    QScopedPointer< ViewPrivate > d;
}; // class View

} /* namespace SecurityCam */

#endif // SECURITYCAM_VIEW_HPP_INCLUDED
```

Private data class.

```
class ViewPrivate {
public:
    explicit ViewPrivate( View * parent )
        :    m_resized( false )
        ,    q( parent )
    {
    }

    //! Image.
    QImage m_image;
    //! Resized?
    bool m_resized;
    //! Parent.
    View * q;
}; // class ViewPrivate
```

In the data class I store the current frame and a flag that current frame was resized. This is the main trick, draw() slot will be connected to video surface signal and will receive frames at maximum speed in the background, where we will just copy frame and set resized flag to false, and will trigger an update of the widget. GUI part of the view will draw a new frame when it can do it, so we will not have a long queue of frames to draw, we will quickly process this queue. Let's look.

```
View::View( QWidget * parent )
    :    QWidget( parent )
    ,    d( new ViewPrivate( this ) )
{
    d->init();
}

View::~View() noexcept
{
}

void
View::draw( const QImage & image )
{
    d->m_resized = false;

    d->m_image = image;

    update();
}

void
View::paintEvent( QPaintEvent * )
{
    if( isVisible() )
    {
        QPainter p( this );

        if( !d->m_image.isNull() )
        {
            if( !d->m_resized )
            {
                d->m_image = d->m_image.scaled( size(), Qt::KeepAspectRatio );

                d->m_resized = true;
            }

            const int x = rect().x() + ( size().width() - d->m_image.width() ) / 2;
            const int y = rect().y() + ( size().height() - d->m_image.height() ) / 2;

            p.drawImage( x, y, d->m_image );
        }
    }
}

void
View::resizeEvent( QResizeEvent * e )
{
    e->accept();

    d->m_resized = false;

    update();
}
```

We do actual resize of the frame only in paint event and only if it was not done before. Believe me, in the running application I don't see any flickering. This simple code does what it was designed for. Memory and CPU usage is constant and very low.

Back | Contents | Next

Video surface

As said we need to access each frame in the camera's stream. For such cases in Qt is QAbstractVideoSurface. Custom video surface can be set to QCamera as viewfider, what we will do. But the video surface doesn't draw anything, it just got access to video frames. Painting of frames will do view that was described in the previous section.

When deriving from QAbstractVideoSurface developer should understand that present() method will be invoked from the non-GUI thread. And very important to return correct list of supported formats from supportedPixelFormats() method. Video frames can come from the device in various formats, but we want to convert QVideoFrame to QImage, so the format of video frame should be compatible with QImage format. Qt can do the trick by pre-converting of video frames format to supported by our video surface format, so we will return just convertible to QImage pixel formats.

Our video surface will detect motions and notify the application about it. Surface will emit a signal with new frames with QImage, but for performance reasons we will emit every frame only if the motion is detected, otherwise we will emit only keyframes. Surface will have abilities to transform video frame before emitting for painting, such as mirroring and rotating.

Detection of motions is based on comparing keyframe with the current one. Surface will emit a signal about difference value of the current image and keyframe. If this value (L2 relative error) is bigger than a threshold then the motion is detected. Each device has its own parameters of noise in the frames, so the threshold is configurable.

Declaration.

```
#ifndef SECURITYCAM_FRAMES_HPP_INCLUDED
#define SECURITYCAM_FRAMES_HPP_INCLUDED

// Qt include.
#include <QAbstractVideoSurface>
#include <QTransform>
#include <QMutex>
#include <QTimer>

// SecurityCam include.
#include "cfg.hpp"


namespace SecurityCam {

//! Count of processed frames when key farme changes.
static const int c_keyFrameChangesOn = 10;


//
// Frames
//

//! Frames listener.
class Frames final
    :    public QAbstractVideoSurface
{
    Q_OBJECT

signals:
    //! New frame arrived.
    void newFrame( QImage );
    //! Motion detected.
    void motionDetected();
    //! No more motions.
    void noMoreMotions();
    //! Images difference.
```

```
    void imgDiff( qreal diff );
    //! No frames.
    void noFrames();
    //! FPS.
    void fps( int v );

public:
    Frames( const Cfg::Cfg & cfg, QObject * parent );

    //! \return Rotation.
    qreal rotation() const;
    //! Set rotation.
    void setRotation( qreal a );

    //! \return Mirrored?
    bool mirrored() const;
    //! Set mirrored.
    void setMirrored( bool on );

    //! \return Threshold.
    qreal threshold() const;
    //! Set threshold.
    void setThreshold( qreal v );

    //! Apply new transformations.
    void applyTransform( bool on = true );

    bool present( const QVideoFrame & frame ) override;

    QList< QVideoFrame::PixelFormat > supportedPixelFormats(
        QAbstractVideoBuffer::HandleType type =
            QAbstractVideoBuffer::NoHandle ) const override;

private:
    //! Detect motion.
    void detectMotion( const QImage & key, const QImage & image );

private slots:
    //! No frames timeout.
    void noFramesTimeout();
    //! 1 second.
    void second();

private:
    Q_DISABLE_COPY( Frames )

    //! Key frame.
    QImage m_keyFrame;
    //! Frames counter.
    int m_counter;
    //! Motions was detected.
    bool m_motion;
    //! Mutex.
    mutable QMutex m_mutex;
    //! Transformation applied.
    bool m_transformApplied;
    //! Transformation.
    QTransform m_transform;
    //! Threshold.
    qreal m_threshold;
    //! Rotation.
    qreal m_rotation;
    //! Mirrored.
    bool m_mirrored;
    //! Timer.
    QTimer * m_timer;
    //! 1 second timer.
    QTimer * m_secTimer;
    //! FPS.
    int m_fps;
}; // class Frames

} /* namespace SecurityCam */

#endif // SECURITYCAM_FRAMES_HPP_INCLUDED
```

This is a formalization in C++ delcaration of what was said above. Some methods are trivial.

```
static const int c_noFramesTimeout = 3000;

//
// Frames
//
```

```
Frames::Frames( const Cfg::Cfg & cfg, QObject * parent )
    :    QAbstractVideoSurface( parent )
    ,    m_counter( 0 )
    ,    m_motion( false )
    ,    m_threshold( cfg.threshold() )
    ,    m_rotation( cfg.rotation() )
    ,    m_mirrored( cfg.mirrored() )
    ,    m_timer( new QTimer( this ) )
    ,    m_secTimer( new QTimer( this ) )
    ,    m_fps( 0 )
{
    if( cfg.applyTransform() )
        applyTransform();

    m_timer->setInterval( c_noFramesTimeout );
    m_secTimer->setInterval( 1000 );

    connect( m_timer, &QTimer::timeout, this, &Frames::noFramesTimeout );
    connect( m_secTimer, &QTimer::timeout, this, &Frames::second );

    m_secTimer->start();
}

qreal
Frames::rotation() const
{
    return m_rotation;
}

void
Frames::setRotation( qreal a )
{
    m_rotation = a;
}

bool
Frames::mirrored() const
{
    return m_mirrored;
}

void
Frames::setMirrored( bool on )
{
    m_mirrored = on;
}

qreal
Frames::threshold() const
{
    QMutexLocker lock( &m_mutex );

    return m_threshold;
}

void
Frames::setThreshold( qreal v )
{
    QMutexLocker lock( &m_mutex );

    m_threshold = v;
}

void
Frames::applyTransform( bool on )
{
    QMutexLocker lock( &m_mutex );

    if( on )
    {
        m_transform = QTransform();

        m_transform.rotate( m_rotation );

        if( qAbs( m_rotation ) > 0.01 )
            m_transformApplied = true;

        if( m_mirrored )
        {
            m_transform.scale( -1.0, 1.0 );

            m_transformApplied = true;
        }
    }
    else
    {
        m_transformApplied = false;

        m_transform = QTransform();
    }
```

```
}
```

The main work is done in present() method.

```
bool
Frames::present( const QVideoFrame & frame )
{
    if( !isActive() )
        return false;

    QMutexLocker lock( &m_mutex );

    QVideoFrame f = frame;
    f.map( QAbstractVideoBuffer::ReadOnly );

    QImage image( f.bits(), f.width(), f.height(), f.bytesPerLine(),
        QVideoFrame::imageFormatFromPixelFormat( f.pixelFormat() ) );

    f.unmap();

    if( m_counter == c_keyFrameChangesOn )
        m_counter = 0;

    QImage tmp = ( m_transformApplied ? image.transformed( m_transform )
        :    image.copy() );

    if( m_counter == 0 )
    {
        if( !m_keyFrame.isNull() )
            detectMotion( m_keyFrame, tmp );

        m_keyFrame = tmp;

        emit newFrame( m_keyFrame );
    }
    else if( m_motion )
        emit newFrame( tmp );

    ++m_counter;
    ++m_fps;

    m_timer->start();

    return true;
}
```

We are converting QVideoFrame to QImage, applying transformation if needed, detecting motion on each keyframe, updating counters and emitting frames for drawing. Important to connect to newFrame() signal as queued one, as present() method invoked in non-GUI thread. And very important to emit a full copy of image because if we will emit temporary image object the data in it will be destroyed as original QImage uses data from QVideoFrame directly and in the slot we will try to access destroyed memory.

Motion detection is made with help of OpenCV and is quite simple, look.

```
inline cv::Mat QImageToCvMat( const QImage & inImage )
{
    switch ( inImage.format() )
    {
        case QImage::Format_ARGB32:
        case QImage::Format_ARGB32_Premultiplied:
        {
            cv::Mat mat( inImage.height(), inImage.width(),
                CV_8UC4,
                const_cast< uchar* >( inImage.bits() ),
                static_cast< size_t >( inImage.bytesPerLine() ) );

            return mat;
        }

        case QImage::Format_RGB32:
        case QImage::Format_RGB888:
        {
            QImage swapped;
```

```
            if( inImage.format() == QImage::Format_RGB32 )
                swapped = inImage.convertToFormat( QImage::Format_RGB888 );

            swapped = inImage.rgbSwapped();

            return cv::Mat( swapped.height(), swapped.width(),
                CV_8UC3,
                const_cast< uchar* >( swapped.bits() ),
                static_cast< size_t >( swapped.bytesPerLine() ) ).clone();
        }

        default:
            break;
    }

    return cv::Mat();
}

void
Frames::detectMotion( const QImage & key, const QImage & image )
{
    bool detected = false;

    try {
        const cv::Mat A = QImageToCvMat( key );
        const cv::Mat B = QImageToCvMat( image );

        // Calculate the L2 relative error between images.
        const double errorL2 = cv::norm( A, B, CV_L2 );
        // Convert to a reasonable scale, since L2 error is summed across
        // all pixels of the image.
        const double similarity = errorL2 / (double)( A.rows * A.cols );

        detected = similarity > m_threshold;

        emit imgDiff( similarity );
    }
    catch( const cv::Exception & )
    {
    }

    if( m_motion && !detected )
    {
        m_motion = false;

        emit noMoreMotions();
    }
    else if( !m_motion && detected )
    {
        m_motion = true;

        emit motionDetected();
    }
}
```

And auxiliary trivial methods.

```
QList< QVideoFrame::PixelFormat >
Frames::supportedPixelFormats( QAbstractVideoBuffer::HandleType type ) const
{
    Q_UNUSED( type )

    return QList< QVideoFrame::PixelFormat > ()
        << QVideoFrame::Format_ARGB32
        << QVideoFrame::Format_ARGB32_Premultiplied
        << QVideoFrame::Format_RGB32
        << QVideoFrame::Format_RGB24;
}

void
Frames::noFramesTimeout()
{
    QMutexLocker lock( &m_mutex );

    m_timer->stop();

    emit noFrames();
}

void
Frames::second()
{
    QMutexLocker lock( &m_mutex );

    emit fps( m_fps );
```

```
    m_fps = 0;
}
```

If to set this video surface to QCamera as viewsfinder and connect newFrame() signal to View's draw() slot then we will see the stream from the camera in View widget.

Back | Contents | Next

Camera

We have the video surface that can be used as viewfinder for QCamera, we have the view that will display stream from the camera, now we need to initialize the camera.

```
void
MainWindowPrivate::initCamera()
{
    if( !m_cfg.camera().isEmpty() )
    {
        auto infos = QCameraInfo::availableCameras();

        if( !infos.isEmpty() )
        {
            QCameraInfo info;

            foreach( auto & i, infos )
            {
                if( i.deviceName() == m_cfg.camera() )
                {
                    info = i;

                    m_currCamInfo.reset( new QCameraInfo( info ) );

                    break;
                }
            }

            if( !info.isNull() )
            {
                m_cam = new QCamera( info, q );

                QObject::connect( m_cam, &QCamera::statusChanged,
                    q, &MainWindow::camStatusChanged );

                m_cam->setViewfinder( m_frames );

                m_capture = new QCameraImageCapture( m_cam, q );

                m_cam->setCaptureMode( QCamera::CaptureStillImage );

                q->setStatusLabel();

                m_cam->start();
            }
            else
            {
                QTimer::singleShot( c_cameraReinitTimeout,
                    [&] () { q->cameraError(); } );
            }
        }
    }
}
```

m *cfg.camera() is saved device name configured in the options dialog, when the application started it reads configuration and initializes camera, we just are looking for saved camera in the system, and if found allocating new QCamera, QCameraImageCapture objects, set view finder - our video surface (m* frames).

We connected to QCamera::statusChanged signal to set a resolution of the camera, we need to do it exactly in QCamera::LoadedStatus state, as only there we can ask the camera for supported viewfinder settings.

```
void
MainWindow::camStatusChanged( QCamera::Status st )
{
    if( st == QCamera::LoadedStatus )
    {
        const auto settings = d->m_cam->supportedViewfinderSettings();

        QCameraViewfinderSettings toApply;

        for( const auto & s : settings )
```

```
        {
            if( s.resolution().width() == d->m_cfg.resolution().width() &&
                s.resolution().height() == d->m_cfg.resolution().height() &&
                qAbs( s.maximumFrameRate() - d->m_cfg.resolution().fps() ) <= 0.001 )
            {
                toApply = s;

                break;
            }
        }

        if( !toApply.isNull() )
        {
            d->m_cam->stop();

            d->m_cam->setViewfinderSettings( toApply );

            setStatusLabel();

            d->m_cam->start();
        }
    }
}
```

On any error in the camera we do.

```
void
MainWindow::cameraError()
{
    d->stopCamera();

    d->m_view->draw( QImage() );

    QTimer::singleShot( c_cameraReinitTimeout, this,
        [&] () { d->initCamera(); } );
}
```

Where d->m view is our view. And.

```
void
MainWindowPrivate::stopCamera()
{
    m_stopTimer->stop();

    if( m_cam )
    {
        m_cam->stop();

        m_capture->deleteLater();

        m_capture = Q_NULLPTR;

        m_cam->deleteLater();

        m_cam = Q_NULLPTR;

        m_currCamInfo.reset();
    }
}
```

This allow us to have always initialized camera (if this is possible) with correct resolution and frame rate.

Back | Contents | Next

Capture images

When video surface detects motion.

```
void
MainWindow::motionDetected()
{
    if( d->m_cam )
    {
        if( !d->m_isRecording )
        {
            d->m_isRecording = true;

            takeImage();

            d->m_timer->start( d->m_takeImageInterval );
        }
        else
            d->m_stopTimer->stop();
    }
}
```

We start to capture images from camera with configured interval.

```
void
MainWindow::takeImage()
{
    if( d->m_capture )
    {
        const QDateTime current = QDateTime::currentDateTime();

        QDir dir( d->m_cfg.folder() );

        const QString path = dir.absolutePath() +
            current.date().toString( QLatin1String( "/yyyy/MM/dd/" ) );

        dir.mkpath( path );

        d->m_cam->searchAndLock();

        d->m_capture->capture( path +
            current.toString( QLatin1String( "hh.mm.ss" ) ) );
    }
}
```

Where d->m timer->timeout() connected to.

```
MainWindow::connect( m_timer, &QTimer::timeout,
        q, &MainWindow::takeImage );
```

And when there is no mo motion in the frame.

```
void
MainWindow::noMoreMotion()
{
    if( d->m_cam )
    {
        if( d->m_isRecording )
            d->m_stopTimer->start( d->m_takeImagesYetInterval );
    }
}

void
MainWindow::stopRecording()
{
    if( d->m_cam )
    {
        d->m_stopTimer->stop();
```

```
        d->m_timer->stop();

        d->m_isRecording = false;
    }
}
```

Where d->m stopTimer->timeout() connected to MainWindow::stopRecording().

Thus, we will have pictures of the attackers on the protected area.

Back | Contents | Next

Chapter 3

As you can saw operations like open, save and crop in GIF editor, described in chapter 1, can long very much. And during these processes UI is frozen. This is sad. Why not add some busy animation during these operations? Good! But animations should work in the main thread. Well, we can dilute code of operations with QApplications::processEvents(), and move Magick++ operations in a separate thread. Amazing, let's do it.

Want to add that multithreading, especially in GUI, is not a panacea. I saw in my practice that very much amount of threads in the application can only slow down the performance, and very much. But the approach described above sounds very good. Our application will be very responsive.

In Qt there are a lot of mechanisms of multithreading, like QThread, QThreadPool, QRunnable, QtConcurrent, queued connections of signals and slots. So let's look at the implementation.

Back | Contents | Next

Implementation

Long Magick++ operations like readImages(), coalesceImages(), writeImages() can longs very much. And during these operations and another UI preparations I'd like to show busy animation. I moved the view of the application to the QStackedWidget, that was set as a central widget of QMainWindow, and in this stacked widget I added a page with busy animation. During long operations I will show the page with animation, and when all is done I will show the ready result.

With Qt's stuff all is simple, I just dilute the code with QApplication::processEvents(), like.

```
//! Initialize tape.
void initTape()
{
    std::for_each( m_frames.cbegin(), m_frames.cend(),
        [this] ( const Magick::Image & img )
        {
            this->m_view->tape()->addFrame( this->convert( img ) );

            QApplication::processEvents();
        } );
}
```

But Magick++ operation should work in separate thread as we can't change the code of Magick++ functions. I decided to run these functions with QRunnable on QThreadPool. Magick++ can throw exceptions, so I declared the base class for all my runnables.

```
class RunnableWithException
    :    public QRunnable
{
public:
    std::exception_ptr exception() const
    {
        return m_eptr;
    }

    ~RunnableWithException() noexcept override = default;

protected:
    std::exception_ptr m_eptr;
}; // class RunnableWithException
```

And let's look at the implementation of the readImages() as runnable object.

```
class ReadGIF final
    :    public RunnableWithException
{
public:
    ReadGIF( std::vector< Magick::Image > * container,
        const std::string & fileName )
        :    m_container( container )
        ,    m_fileName( fileName )
    {
        setAutoDelete( false );
    }

    void run() override
    {
        try {
            Magick::readImages( m_container, m_fileName );
        }
        catch( ... )
        {
            m_eptr = std::current_exception();
        }
```

```
    }

private:
    std::vector< Magick::Image > * m_container;
    std::string m_fileName;
}; // class ReadGIF
```

Voila. And when I need to read GIF.

```
std::vector< Magick::Image > frames;

ReadGIF read( &frames, fileName.toStdString() );
QThreadPool::globalInstance()->start( &read );

d->waitThreadPool();

if( read.exception() )
    std::rethrow_exception( read.exception() );
```

Where d->waitThreadPool() is.

```
//! Wait for thread pool.
void waitThreadPool()
{
    while( !QThreadPool::globalInstance()->waitForDone( 100 / 6 ) )
        QApplication::processEvents();
}
```

That's all. Now GIF editor shows busy animation during long operations, UI is responsive.

I decided to disable all actions during such operations, even quit from the application. But what if the user wants to exit from the application during opening? We can allow to do it on the close button in the window's title click.

```
void
MainWindow::closeEvent( QCloseEvent * e )
{
    if ( d->m_busyFlag )
    {
        const auto btn = QMessageBox::question( this, tr( "GIF editor is busy..." ),
            tr( "GIF editor is busy.\nDo you want to terminate the application?" ) );

        if( btn == QMessageBox::Yes )
            exit( -1 );
        else
            e->ignore();
    }
    else
        e->accept();

    quit();
}
```

Where d->m busyFlag is a bool that I set to true when showing busy animation.

Wonderful, UI is always responsive and the user can terminate the application during the long operation at any time.

Back | Contents | Next

Links

- Qt - https://www.qt.io/
- ImageMagick - https://www.imagemagick.org/script/index.php
- OpenCV - https://opencv.org/

Contents