Code less.
Create more.
Deploy everywhere.

Creating Cross-Platform Visualization UIs with Qt and OpenGL®

Abstract

Scientific visualization, medical imaging, flight simulation, flow modelling, animation, gaming and visual effects are areas in which high performance 2D and 3D graphics applications are in high demand. Standard graphics APIs like OpenGL® are ideal for complex graphics rendering, but offer little support for programming the user interfaces to serve these diverse markets.

In this paper, we will discuss the common challenges faced by developers of visualization software, and present techniques for easily integrating advanced 2D and 3D graphics into native, high-performance applications. Using the Qt® application framework, we will demonstrate how these techniques speed visualization development, and enable deployment across multiple operating systems from a single codebase.

NOKIA

Introduction

Most applications display information through tables, lists, images, graphs, maps and animations. Among these, the simpler visualization techniques don't require much graphics horsepower. They just map data into stock widgets. For example, the contents of a database can be displayed in a table widget and the contents of a log file can be displayed in a list view. But the more advanced visualization techniques often require low-level rendering technologies, like OpenGL®, and high-level APIs, like Scene graphs, Open Inventor, and Visualization ToolKit (VTK). These advanced techniques typically manipulate the Graphics Processor Unit (GPU) directly to obtain the graphics horsepower they need.

All widget toolkits support the simple visualization techniques with the standard set of GUI widgets, but the level of support for the advanced visualization techniques varies. In this whitepaper, we discuss how Qt provides support for the advanced visualization techniques, including out-of-the-box support for 2D and 3D rendering and tight integration with the native widgets.

Common Use Cases

For the purpose of this whitepaper, ***advanced visualization techniques*** refers to the visualization techniques required by the following domains:

- Medical imaging - Displaying ECG, MRI, CT scans
- Terrain visualization - Showing geography, weather and heat maps
- Flow and process visualization - Showing manufacturing processes
- Data mining - Visualizing history, log files
- Aerospace and Defence - Radar images
- Automotive - 3D modelling of vehicles, CAD

Standard Visualization APIs

The two main APIs for high performance visualization are:

1. **OpenGL** – OpenGL is a cross-platform, device-independent graphics API for rendering 2D and 3D graphics.
2. **Direct3D -** Direct3D is a graphics acceleration API that is part of the Microsoft® DirectX suite, available only on Windows®.

Both OpenGL and Direct3D are "low-level" graphics APIs. An application using either one must tell the computer how to draw a scene in terms of low level primitives such as vertices, lines and polygons. For example, to draw a cube, the cube's eight vertices must be specified. Imagine attempting to render something as complex as a dining table using one of these low-level APIs. Also, these APIs are highly sensitive to the order in which functions are called, making their use error prone. Consequently, high-level APIs that support object oriented 3D visualization have been developed. These APIs allow the programmer to define a scene (scene graph APIs) and then place objects in the scene.

For 2D visualization, there is no standard for scene graphs. The feature set and performance vary from toolkit to toolkit. In the following sections, we discuss Qt's 2D scene graph Graphics View and how one can easily render 2D scenes using OpenGL.

For 3D visualization, APIs like Open Inventor and VTK are considered standard. We will discuss their integration with Qt applications.

Common Challenges in Visualization Development

Some of the most common challenges faced when developing visualization programs are listed below:

- **Supporting the native user interface**
  Visualization APIs like OpenGL are for rendering items quickly in an area of the computer screen. They have no mechanisms for creating windows and widgets. Nor do they have mechanisms for trapping mouse, keyboard, or window system events. This means that "glue code" is required to embed an OpenGL rendered area into a window.

- **Supporting multiple platforms**
  OpenGL is cross-platform, but the choice of the widget toolkit that encapsulates the OpenGL scene might unnecessarily limit the application to a particular platform.

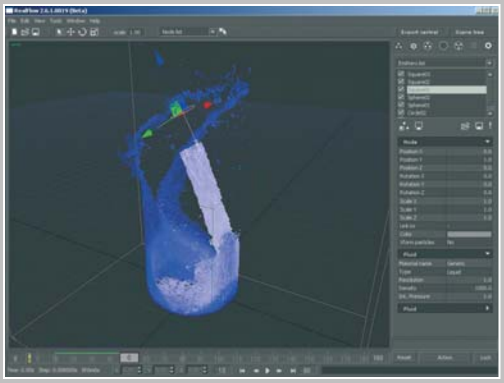
- **Supporting 2D graphics**
  Most applications require support for basic 2D painting, including drawing polygons, transformations, paths, gradients, and image manipulation. In addition, some applications require functionality built on top of the basic 2D painting, for example charting and plotting. In addition to basic, stateless 2D painting, some applications require the convenience of a 2D retained mode scene graph.

- **Supporting utilities**
  A visualization application's main task may be to render quality graphics, but it still requires additional APIs to print documents, parse XML, read databases, and use threads, networking, and so on. An ideal toolkit should support these additional utilities and be cross-platform at the same time.

- **Creating overlays**
  It is common to have "controller" buttons for a visualization application, for example to zoom in and out. The functionality to overlay native widgets on an OpenGL area is a feature of the widget toolkit.

In the following sections, we will see how Qt meets these challenges.

Qt in Use



Company: Next Limit Technologies
Application: RealFlow 4

When studios and visual effects houses including Disney and Pixar need to make the waters part, gush, and swirl, they often turn to Next Limit Technologies and its RealFlow simulation software. Next Limit won an Oscar for Technical Achievement in 2007 for its work, which lets movie makers animate the flows of water and other liquids with greater realism.

RealFlow was originally developed as a Windows application. As Real Flow began attracting potential customers who preferred Linux. So rather than port the Win32 code to Linux, Next Limit rebuilt its software on Qt.

To test Qt, Next Limit designed an easy application that would cover the most sensitive issues of RealFlow, including the ability to handle OpenGL in a fluent way and the ability to use computation threads that work while the user is manipulating the GUI. The test was a resounding success enabling Next Limit developers to port RealFlow to Qt in only two months.

"It was really easy," said Angel Tena, technical director for Next Limit's RealFlow product. "The Qt classes were well-designed and easy to use. We often didn't even have to use the documentation. We just figured out what the name of the function was and a great number of functions would be there."

http://www.realflow.com/

What is Qt?

Qt is a cross-platform application framework for desktop and embedded development. It includes an intuitive API and a rich C++ class library, integrated tools for GUI development and internationalization, and support for Java™ and C++ development.

Qt provides support for advanced 2D graphics using the QPainter and QGraphicsView APIs. In addition, Qt supports 3D graphics with its OpenGL module. Each of these technologies is presented in more detail in the following sections.

Solutions for Advanced Visualization

In this section, we discuss the technologies that Qt provides for easier development of advanced visualization applications.

Basic 2D Graphics: QPainter

QPainter provides a comprehensive 2D painting framework. In addition to basic features such as rendering polygons, painter paths, affine and non-affine transformations, it supports antialiasing, gradient brushes and alpha blending. Usage of QPainter is trivial:

```
void Widget::paintEvent(QPaintEvent *event)
{
            QPainter painter(this);
            painter.setBrush(Qt::red);
            painter.drawRect(rect());
     // create a pen from any brush!
            painter.setPen(QPen(QBrush(“texture.png”), 1);
            painter.drawLine(10, 30, 40, 503);
}
```

A QPainter can be used on any QpaintDevice. This includes widgets (QWidget), images (QImage), printers (QPrinter), and pixmaps (QPixmap). Note that the QPainter is just an API for 2D rendering. The actual painting is done using backends, which are implementations of QPaintEngine. Qt provides paint engines that use Raster Graphics (Windows), XRender (X11), OpenGL (all platforms), and PDF (all platforms). This means that the “painter.drawLine()” function call above is automagically converted to an OpenGL command, or to a PDF command, merely by letting QPainter use the appropriate paint engine, and by using a QPaintDevice that supports the paint engine. We will see in a later section how QPainter can be used to draw accelerated 2D graphics using OpenGL.

The 2D Scene Graph: Qt Graphics View

Qt Graphics View provides a 2D scene graph API. It has a high-level API for placing objects (shapes) in a scene and for displaying the scene in multiple views. The strength of Graphics View lies in the fact that it exposes a very simple API, despite solving a very complex problem.

To get started with using Graphics View, all one needs to do is to create a QGraphicsScene (the scene) that contains the necessary QGraphicsItems (the shapes). The scene then can be visualized using QGraphicsView (the view).

```
QGraphicsScene scene;
    scene.addRect(QRectF(0, 0, 100, 100));
    // populate more items in the scene

    QGraphicsView *view = new QGraphicsView;
    view->setScene(&scene);
    view->show();
```

Custom items can be created by subclassing QGraphicsItem. Graphics View does not require items to be rectangular. For example, we may require a circular item that does not receive mouse events when clicked outside the "circle". In such a case, we let shape() return the shape of the item. The rectangle returned by boundingRect() is used by Graphics View to determine if an item requires repainting. The paint() function performs the actual drawing.

```
  QRectF CircularItem::boundingRect() const
    {
        return QRectF(0, 0, 100, 100);
    }

    QPainterPath CircularItem::shape() const
    {
        QPainterPath path;
        path.addEllipse(boundingRect());
        return path;
    }

    void CircularItem::paint(QPainter *painter,
                         const QStyleOptionGraphicsItem *option,
                         QWidget *Widget = 0)
    {
        painter->drawEllipse(QRectF(0, 0, 100, 100));
    }
```

Graphics View provides a complete framework for handling input, just as in the world of widgets. Every item can receive mouse and keyboard events. Events can be handled, or they can be ignored, in which case they are propagated upward to the parent, view, and, ultimately, the scene. Graphics View items also support drag and drop, custom cursors, tooltips and animations, just like widgets. In addition, Graphics View items can be grouped (items can then be moved/selected as a group) and support collision detection.

Note the QStyleOptionGraphicsItem argument to the paint() function of class CircularItem above. The option structure contains an interesting field called levelOfDetail that provides a hint about the zoom status of an item. When the item is sufficiently zoomed out, one can skip drawing the details. This optimizes the rendering speed of complex items.

One of the key features of Graphics View is its *out-of-the-box* support for transformations. Transformations can be done on a per item basis or on the entire scene. To add zooming capabilities to a scene, all one needs to do is to call scale(). In the example below, we press '+' to zoom in or '-' to zoom out. The code is written as:

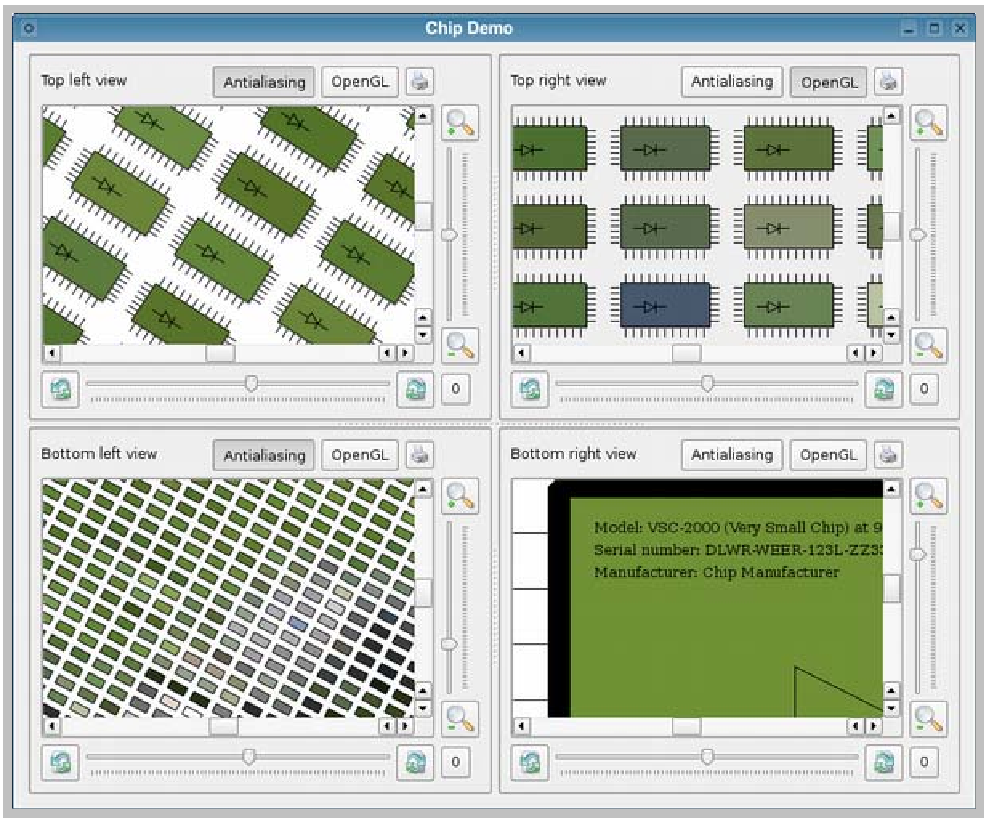
```
class ZoomableView : public QGraphicsView
{
public:
    ZoomableView(QWidget *parent = 0) : QGraphicsView(parent) { }

protected:
    void keyPressEvent(QKeyEvent *event) {
        if (event->key() == Qt::Key_Plus) {
            scale(1.5, 1.5);
        } else if (event->key() == Qt::Key_Minus) {
            scale(1/1.5, 1/1.5);
        } else {
            event->ignore();
        }
    }
};
```

Complex scenes with millions of items require considerable horsepower for rendering. Graphics View supports rendering a scene using OpenGL with a single line:

```
  view->setViewport(new QGLWidget(QGLFormat(QGL::SampleBuffers))); // SampleBuffers for
antialiasing
```

This means that in the case of the CircularItem above, the Circle is actually painted using OpenGL. As we will see in the section "OpenGL Paint Engine", this magic OpenGL rendering without writing a single line of OpenGL is made possible thanks to Qt's extensible painting system.

Qt ships with a Chip demo that visualizes 400000 chips using QGraphicsView. It demonstrates visualization of the chips using the Raster engine and OpenGL *(screenshot on next page)*.



Model: VSC-2000 (Very Small Chip) at 9
Serial number: DLWR-WEER-123L-ZZ3
Manufacturer: Chip Manufacturer

Fig. 1: 400000 chips displayed in 4 views. Each view can have a different transformation and optionally rendered using OpenGL.

More Advanced 2D Graphics: Widgets on Canvas

In the latest release of Qt -- Qt 4.4 -- Graphics View obtained the capability of placing native widgets on the scene. Widgets on the canvas behave just like normal items - they can even be transformed. And, unlike native widgets, Widgets on the canvas appear simultaneously in multiple views. Widgets on the canvas are created using QGraphicsProxyWidget:

```
QComboBox *modeCombo = new QComboBox;
    modeCombo->addItem("Landscape");
    QGraphicsProxyWidget *proxyWidget = new QGraphicsProxyWidget;
    proxyWidget->setWidget(modeCombo);
    scene.addItem(proxyWidget);
```

As of Qt 4.4, Graphics View also includes a layout system for QGraphicsItems. To enable placement of items in a Graphics View layout (QGraphicsLayout), items are subclassed from QGraphicsLayoutItem. One can then use QGraphicsLinearLayout or QGraphicsGridLayout to layout items linearly (horizontally/vertically) or in a grid.

It is common to place widgets on the canvas to implement "controller" widgets, e.g. zoom in/out buttons. Widget items are no different from other canvas items. They transform when the canvas has a transform. This is most certainly undesirable. When the canvas is zoomed in, one doesn't expect the controller buttons to zoom along with the canvas. This undesirable effect is easily avoided using the following code:

```
// don't transform this widget with the canvas
  widgetItem->setFlags(QGraphicsItem::ItemIgnoresTransformations);
```

Support for Image Formats

Qt's QImage supports standard file formats. It has built in support for loading PNG and XPM files. Most visualization applications require support for loading custom file formats, since images are usually stored in a manner optimized for a particular domain. QImage supports a plugin architecture that enables one to add complete Qt support for a new file format. JPG, GIF and MNG files, for example, are implemented as plugins by subclassing QimageIOPlugin. Note that plugins can be statically compiled, if required.

Qt supports loading SVG 1.2 Tiny files using the Qt SVG Module. It provides a QsvgRenderer, which can be used to load and paint any SVG file>

```
class Widget : public QWidget
{
public:
    Widget(QWidget *parent = 0)
    {
        renderer = new QSvgRenderer(this);
        renderer->load(QString("sample.svg"));
        QObject::connect(renderer, SIGNAL(repaintNeeded()),
                         this, SLOT(update())); // animated SVG
    }

protected:
    void paintEvent(QPaintEvent *event)
    {
        QPainter painter(this);
        renderer->render(&painter);
    }

    QSvgRenderer *renderer;
};
```

In real applications, one may use the QSvgWidget, which displays an SVG file using the QSvgRenderer (as in the example above).

3D Graphics: Qt OpenGL Module

Qt developers can use OpenGL to draw 3D graphics in their GUI applications. Qt provides a separate Qt OpenGL Module that integrates the OpenGL visual with the native windowing system.

To use OpenGL with Qt, you subclass the QGLWidget. The key features of QGLWidget, demonstrated in the code below, are:

- initializeGL() is called to initialize the GL context. Qt provides convenience functions such as qglClearColor and qglColor to convert QColor to gl color.
- resizeGL() is called when the widget is resized. We can adjust our viewport and transforms accordingly.
- paintGL() is called when the OpenGL widget requires update.

```
class GLWidget : public QGLWidget
{
public:
    GLWidget(QWidget *parent = 0) : QGLWidget(parent)
    {
        setFormat(QGL::DoubleBuffer | QGL::DepthBuffer);
    }

protected:
    void initializeGL()
    {
        qglClearColor(Qt::white);
    }

    void resizeGL(int w, int h)
    {
        glViewport(0, 0, w, h);
        // setup project matrix
    }

    void paintGL()
    {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        // draw using gl
    }
};
```

A QGLWidget can be used like any normal QWidget. Multiple QGLWidgets can be created and placed in a layout with a specified size. Mouse and keyboard events are handled (just like QWidget) using mousePressEvent() and keyPressEvent(). For animations, all that is required is to start a QTimer and call updateGL().

Support for the GL frame buffer object and the GL pixel buffer is provided by QGLFrameBufferObject and QGLPixelBuffer respectively.

Qt provides convenience functions for loading images and binding them to textures. For example, this code loads an image and binds it to a texture>

```
texture = bindTexture(QPixmap(QString("image.png”)), GL_TEXTURE_2D);
```

Internally, Qt tracks the pixmaps/images that have been bound to textures. This allows for reuse of textures when using the same image file or pixmap.

The current scene can be rendered to a pixmap using renderPixmap(). The contents of the frame buffer can be grabbed using grabFrameBuffer(). The key difference between renderPixmap() and grabFrameBuffer() is that overlays are not captured when renderPixmap() is used.

Qt Embedded, the embedded version of Qt available for both Embedded Linux and Windows CE, supports OpenGL ES, an embedded systems variant of Open GL. With Qt Embedded, you can render your entire window using OpenGL by using QWSGLWindowSurface.

OpenGL Paint Engine – 2D Graphics with OpenGL

Recall that QPainter is just an API, and that the actual rendering is performed by a QPaintEngine. Qt has a QGLPaintEngine that provides the Open GL backend for QPainter. A QPainter that is created on a QGLWidget automatically gets the QGLPaintEngine as its paint engine. Any 2D graphics rendered using QPainter will then be translated to OpenGL automatically.

```
void myGLWidget::paintEvent(QPaintEvent *event)
{
            // First, create a QPainter on a QGLWidget
            QPainter painter(this);
            // That's it! All commands issued to QPainter are now
     // rendered using OpenGL!
            painter.setRenderHint(Qt::AntiAliasing);
            painter.drawLine(10, 203, 40, 30);
            painter.drawRect(QRectF(100, 100, 50, 50));
}
```

Using a QPainter instead of OpenGL commands has the advantage of easily switching the rendering from the platform default to OpenGL and vice versa. It also doesn't require the programmer to have knowledge of OpenGL. The sample code above shows how this trick used to render QGraphicsView using OpenGL.

QPainter does cause a minor performance hit, since each call to QPainter will have an additional overhead of a virtual function call compared to using OpenGL directly. But this is unnoticeable if the application does not require high graphics performance. The best way to find out is to try it.

Overlay Support

Overlays are visuals that are drawn on top of a scene. They should normally remain unaffected by the viewing transformations of a scene, and typically you can consider them independent from the model.

There are three mechanisms by which Qt can draw overlays on top of a QGLWidget:

1. **Using a QPainter**
   We can use QPainter to draw 2D overlays. Since a QGLWidget is a QWidget, it receives the paintEvent() when the widget requires an update. We can construct a QPainter in the paintEvent() and draw the overlay as follows:

```
GLWidget::GLWidget()
            {
                        // QPainter automatically calls glClear() unless
          // autoFillBackground is false.
                        setAutoFillBackground(false);
            }

            void GLWidget::paintEvent(QPaintEvent *event)
     {
                        makeCurrent(); // set the context for this widget
          paintGl();
                        QPainter painter(this);
                        drawOverlay(&painter);
                        // remember to swap buffers if double buffered!
            }
     void GLWidget::drawOverlay(QPainter *painter)
     {
         painter.drawRect(QRectF(50, 50, 100, 100),
                          QColor(255, 0, 0, 140 /* alpha */));
     }
```

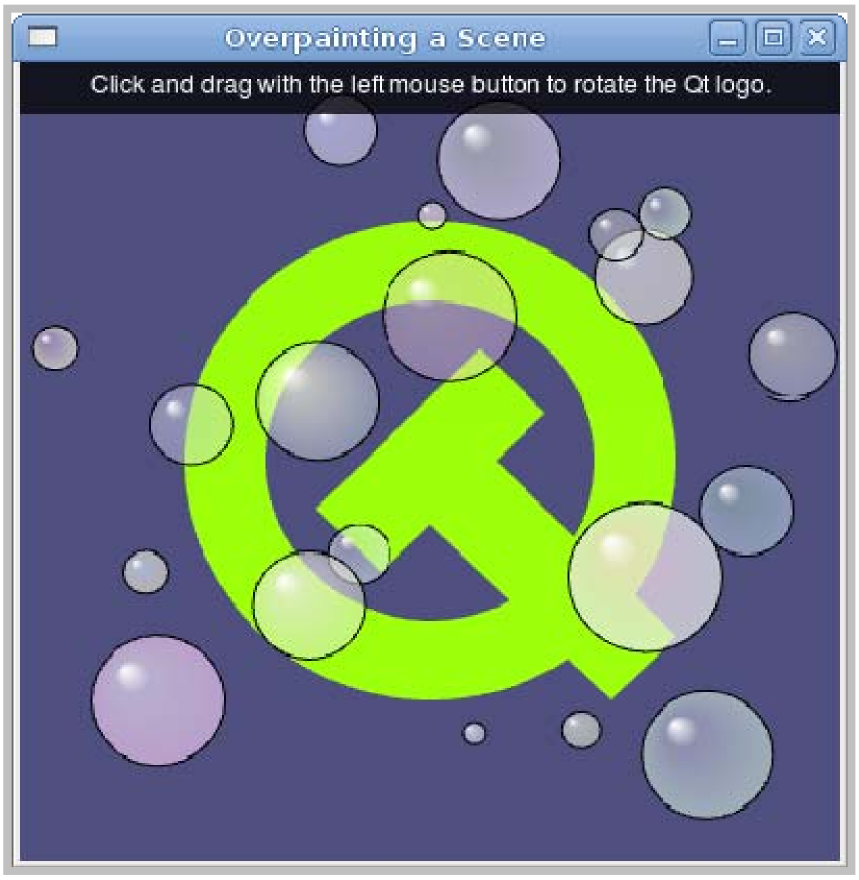

Overpainting a Scene
Click and drag with the left mouse button to rotate the Qt logo.

Fig. 2: An example of painting overlays using QPainter is included with Qt. The text "Qt" is drawn in the main plane, and the bubbles are overlays drawn with a QPainter.

It is also possible to interleave OpenGL commands and QPainter, provided we save and restore the OpenGL state before and after issuing any QPainter commands (using glPushAttrib). Because the scene must be redrawn to update the overlay, this technique is useful only when the overlay is not updated frequently.

2. **Using Framebuffer Objects**
   Sometimes, frequent update of the overlay is required. For example, a rubber band selection requires redrawing the rubber band when the mouse is dragged. Using the QPainter approach, the scene must be redrawn each time the overlay is redrawn.

   One solution is to cache the current scene rendering as a Frambuffer object. Qt provides QGLFramebufferObject to create offscreen surfaces. You can use QGLFramebufferObject::bind() to start painting on the surface and QGLFramebufferObject::release() to end it. Once the scene is rendered to the Framebuffer object, it can be used as a texture for the scene rectangle. Every time the scene is updated, the Framebuffer object is updated. The overlay is drawn over the scene rectangle using QPainter or plain GL commands.

3. **Using OpenGL Overlays**
   OpenGL overlays are a feature of the windowing system interface, which helps bind the OpenGL and the windowing system. OpenGL overlays are supported by WGL on Windows, and by GLX
   on X11.

Qt provides a cross-platform interface for creating overlays. QGLFormat::hasOpenGLOverlays() can be used to detect at runtime if the platform supports overlays.

To create a QGLWidget that has an overlay, we specify it in the QGLFormat. QGLWidget provides callback function for drawing overlays similar to the ones for the main plane: initializeOverlayGL(), resizeOverlayGL(), paintOverlayGL() and updateOverlayGL(). The makeOverlayCurrent() function can be used to mark the Overlay context as the current context.

```
class GLWidget
{
      GLWidget(QWidget *parent)
             : QGLWidget(QGLFormat(QGL::HasOverlay)))
  {
  }

protected:
    void initializeOverlayGL()
    {
        // initialize the overlay
    }

    void resizeOverlayGL(int w, int h)
    {
        // overlay was resized
    }

    void paintGL()
    {
        // draw the overlay
    }
};
```

Supporting Utilities

Qt is not just about providing a cross-platform user interface. It has a whole range of cross-platform utilities that are required by most programs. For example:

- **Qt XML module** – Provides a DOM, SAX, Stream parser for XML. The XmlPatterns module provides XQuery and XPath support.
- **Qt Multimedia module** – Provides support for playing audio and video files.
- **Qt Database module** – Supports SQL queries using pluggable database drivers
- **Qt Network module** – Support for sockets and common network protocol implementations

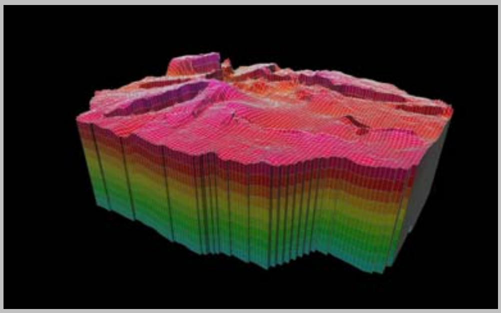
This is only the tip of the iceberg. Qt also includes support for printing (to a printer or to a PDF), threading, look and feel customization, and more. See the Qt Documentation for more information (http://doc.trolltech.com).

Third Party Software

OpenGL is a low-level API, which is hard to use directly for building complex visualizations. Consequently, many 3D object-oriented APIs have been developed that build on OpenGL:

- **Coin3D and Quarter** – Coin3D, a product of Systems in Motion, is a high-level, retained-mode toolkit for effective 3D graphics development. It is an alternate implementation to the Open Inventor API. Applications that use Coin can use the SoQt library to integrate with Qt. Quarter provides a QuarterWidget (derived from QGLWidget) that provides functionality for rendering of Coin scenegraphs and translation of QEvents into SoEvents.
- **VTK** – VTK is a platform-independent graphics engine with parallel rendering support. VTK can be compiled to use Qt using the VTK_USE_GUISUPPORT and VTK_USE_QVTK options. See http://wiki.qtcentre.org/index.php?title=VTK_with_Qt for more details.

Qt in Use



Company: Midland Valley Exploration Ltd.
Application: 4DVista

The Midland Valley Exploration team was planning on a major re-design of its product suite, a sophisticated structural geology application that provides insight into subsurface exploration using 3D visualization. They wanted the new application to run on Linux and Windows as well as their legacy UNIX systems and to integrate seamlessly with their graphics toolkit, Coin 3D. Qt enabled the team to streamline its UI development process for faster coding. The results were impressive, with the new application appearing native on all platforms without any code tweaks. Additionally, Qt worked seamlessly with Midland Valley's established the graphics toolkit, Coin 3D. The MVE team also leveraged the selection of widgets and third party add-ons and libraries available with Qt.

"It was clear to us that Qt is a mature platform with mature OpenGL module support, and a tried and tested C++ solution – two things we absolutely needed," said Colin Dunlop, Principal Software Engineer, Midland Valley Exploration Ltd.

http://www.mve.com/

Summary

Visualization applications require 2D and 3D APIs that integrate well with the widgets of the underlying windowing system. For 2D graphics, Qt provides the sophisticated Graphics View framework for placing objects in a scene and for displaying the scene in multiple views. For 3D graphics using OpenGL, Qt provides the Qt OpenGL module for integrating native widgets with an OpenGL scene. Because Qt is completely cross-platform, applications that use the Qt OpenGL module are also cross-platform, which means they can be maintained using a single source code base.

Learn More and Download

To learn more about the Qt cross-platform application framework and to review further customer case studies and Qt-based applications, please visit the Qt website at http://trolltech.com/.

Free evaluation versions of Qt are available for the Windows, Mac, Linux, Embedded Linux and Windows CE platforms. Evaluations come with available source code and technical support during your evaluation period. To download Qt, please visit http://trolltech.com/downloads.

About Qt Software

Qt Software (formerly Trolltech) is a global leader in cross-platform application frameworks. Nokia acquired Trolltech in June 2008, renamed it to Qt Software as a group within Nokia. Qt allows open source and commercial customers to code less, create more and deploy everywhere. Qt enables developers to build innovative services and applications once and then extend the innovation across all major desktop, mobile and other embedded platforms without rewriting the code. Qt is also used by multiple leading consumer electronics vendors to create advanced user interfaces for Linux devices. Qt underpins Nokia strategy to develop a wide range of products and experiences that people can fall in love with.

About Nokia

Nokia is the world leader in mobility, driving the transformation and growth of the converging Internet and communications industries. We make a wide range of mobile devices with services and software that enable people to experience music, navigation, video, television, imaging, games, business mobility and more. Developing and growing our offering of consumer Internet services, as well as our enterprise solutions and software, is a key area of focus. We also provide equipment, solutions and services for communications networks through Nokia Siemens Networks.

Code less.
Create more.
Deploy everywhere.

Oslo, Norway +47 21 60 48 00 ▪ **Redwood City, CA, USA** +1 650-551-1676 ▪ **Beijing, China** +86 10 6588 8836 ▪ qt-info@nokia.com **www.trolltech.com**