Custom Qt Quick Components Using OpenGL

Kevin Funk, Software Engineer at KDAB
Integrating QtQuick with Qt

- Integrating QML with C++ (page 4)
  - Creating New QML Elements (page 5)
Integrating QtQuick with Qt

- Integrating QtQuick with Qt
  - Integrating QML with C++
Integrating QtQuick with Qt

- **Integrating QML with C++**
  - Creating New QML Elements
Integrating QML with C++

- Creating New QML Elements
  - Registering & Using Custom Elements
  - Creating GUI elements
Creating New QML Elements

Steps to define a new type in QML:

1. In C++: Subclass either QObject or QQuickItem.
2. In C++: Register the type with the QML environment.
3. In QML: Import the module containing the new item.
4. In QML: Use the item like any other standard item.
Creating New QML Elements

- Non-visual types are QObject subclasses.
- Visual types (items) are QQuickItem subclasses.
  - QQuickItem is the C++ equivalent of Item.
  - QQuickPaintedItem is for drawing items with QPainter.
Creating New QML Elements

- Registering & Using Custom Elements
- Creating GUI elements
Step 1: Implementing the Element

```cpp
#include <QObject>

class QTimer;

class RandomTimer : public QObject
{
    Q_OBJECT
public:
    RandomTimer(QObject *parent = 0);
private:
    QTimer* m_timer;
};
```
Implementing the Element

- **RandomTimer** is a QObject subclass.
- As with all QObjects, each item can have a parent.
- Non-GUI custom items do not need to worry about any painting.
Step 1: Implementing the Element

```cpp
#include "randomtimer.h"
#include <QTimer>

RandomTimer::RandomTimer(QObject *parent) :
    QObject(parent),
    m_timer(new QTimer(this))
{
    m_timer->setInterval(1000);
    m_timer->start();
}
```
Step 2: Registering the Element

```cpp
#include <QGuiApplication>
#include <QQuickView>
#include "randomtimer.h"

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);
    qmlRegisterType<RandomTimer>("CustomComponents", 1, 0, "RandomTimer");

    QQuickView view;
    view.setSource(QUrl("qrc:///main.qml"));
    view.show();
    return app.exec();
}
```

- RandomTimer registered as an element in module CustomComponents
- Automatically available to the main.qml file
Reviewing the Registration

```qml
qmlRegisterType<RandomTimer>("CustomComponents", 1, 0, "RandomTimer");
```

- This registers the `RandomTimer` C++ class.

- Available from the `CustomComponents` QML module
  - Version 1.0 (first number is major; second is minor)

- Available as the `RandomTimer` element
  - The `RandomTimer` element is an non-visual item.
  - A subclass of `QObject`
In the *main.qml* file:

```qml
import QtQuick 2.0
import CustomComponents 1.0

Rectangle {
    width: 500
    height: 360
    RandomTimer {
        id: timer
        ...
    }
}
```

*Demo qml-cpp-integration/ex_simple_timer*
Creating New QML Elements

- Registering & Using Custom Elements
- Creating GUI elements
Creating a Custom QML Element

- Canvas item (slow) [QML]
  - Provides a 2D canvas item
  - Allows to draw primitives using JavaScript (HTML5-like API)

- Painted items (slow) [C++]
  - Subclass QQuickPaintedItem
  - Implement paint(...)

- Scene graph items (hardware accelerated) [C++]
  - Subclass QQuickItem
  - Implement updatePaintNode(...)

- Raw OpenGL items [C++]
  - Subclass QQuickFrameBufferObject
  - Implement createRenderer(...)
  - → Briefly handled in this talk!
Creating a Custom QML Element

Creating a custom Canvas item
Creating a Custom Canvas item

In the `TriangleCanvas.qml` file:

```qml
Canvas {
    id: triangle

    onPaint: {
        var ctx = getContext("2d");
        ctx.beginPath();
        ctx.moveTo(0, 0);
        ctx.lineTo(triangleWidth, 0);
        ctx.lineTo(0.5 * triangleWidth, triangleHeight);
        ctx.closePath();
    }
}
```
Creating a Custom Canvas item (cont'd)

In the `main.qml` file:

```qml
Window {
    id: root
    visible: true
    width: 800
    height: 600
    TriangleCanvas {
        anchors.fill: parent
        anchors.margins: 10
    }
}
```

Demo `qml-cpp-integration/ex-canva-sitem`

Qt Docs Qt Quick Canvas
Creating a Custom QML Element

Exporting a Painted Item Class
Exporting a Painted Item Class

```cpp
#include <QQuickPaintedItem>

class TriangleItem : public QQuickPaintedItem
{
    Q_OBJECT
    Q_PROPERTY(QColor color READ color WRITE setColor NOTIFY colorChanged)

public:
    explicit TriangleItem(QQuickItem *parent = 0);
    void paint(QPainter *painter) override;
};
```
Exporting a Painted Item Class (cont'd)

```cpp
TriangleItem::TriangleItem(QQuickItem *parent)
  : QQuickPaintedItem(parent)
{
}

void TriangleItem::paint(QPainter *painter)
{
  const auto rect = contentsBoundingRect();
  const QPolygonF trianglePolygon({
    {0, 0},
    {rect.width(), 0},
    {0.5 * rect.width(), rect.height()},
    {0, 0}
  });
  painter->drawPolygon(trianglePolygon);
}
```
Exporting a Painted Item Class (cont'd)

```cpp
#include "triangleitem.h"

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);
    qmlRegisterType<TriangleItem>("Shapes", 1, 0, "Triangle");

    QQuickView view;
    view.setResizeMode(QQuickView::SizeRootObjectToView);
    view.setSource(QUrl("qrc:triangle1.qml"));
    view.show();
    return app.exec();
}
```
Exporting a Painted Item Class (cont'd)

In the `triangle1.qml` file:

```qml
import Shapes 1.0

Item {
  width: 300; height: 200

  Triangle {
    id: ellipse
    anchors.fill: parent
    anchors.margins: 50
  }
}
```

Demo [qml-cpp-integration/ex-simple-item](https://example.com/qml-cpp-integration/ex-simple-item)

Qt Docs [QQuickPaintedItem](https://examples.org/qml/QQuickPaintedItem)
Qt Docs [QPainter](https://docs.org/qml/QPainter)
Creating a Custom QML Element

Exporting a Scene Graph Item Class
Introduction to the Qt Quick 2 Render Thread

Render Thread

Main Thread

Ask for item tree

Optimize item tree, Render scene graph

Update properties, Advance animations...

Creating GUI elements
Scene Graph API

Creating GUI elements
Scene Graph API

- **QSGNode** is the basic scene graph building block.
  - QSGGeometryNode is a sub-class providing geometry and material for rendering.
    - QSGSimpleRectNode is a convenience class for solid filled rectangles.
    - QSGSimpleTextureNode is a convenience class for textured content.
  - QSGOpacityNode is a sub-class applying its opacity to its sub-tree.
  - QSGTransformNode is a sub-class applying a transformation to its sub-tree.

- QSGGeometry allows storing of vertex data to create graphics primitives.

- **QSGMaterial** is the base class used to specify materials.
  - QSGFlatColorMaterial is a convenience class for solid color fills.
  - QSGOpaqueTextureMaterial is a convenience class using QSGTexture for textured fills.
Exporting a Scene Graph Item Class

```cpp
#include <QQuickItem>

class TriangleItem : public QQuickItem {
    Q_OBJECT

public:
    TriangleItem(QQuickItem *parent = 0);

protected:
    QSGNode *updatePaintNode(QSGNode *oldNode,
                              UpdatePaintNodeData *data) override;
};
```

Demo qml-cpp-integration/ex-simple-item-scenegraph

Qt Docs QQuickItem  Qt Docs QSGNode  Qt Docs QSGMaterial
TriangleItem::TriangleItem(QObject *parent)
    : TriangleItem(parent)
{
    setFlag(QQuickItem::ItemHasContents, true);
}

QSGGeometryNode *TriangleItem::updatePaintNode(QSGGeometryNode *oldNode,
                                               UpdatePaintNodeData *)
{
    if (width() <= 0 || height() <= 0) {
        delete oldNode;
        return 0;
    }

    QSGGeometryNode *triangle = static_cast<QSGGeometryNode*>(oldNode);
    if (!triangle) {
        triangle = new QSGGeometryNode;
        triangle->setFlag(QSGGeometryNode::ownsMaterial, true);
        triangle->setFlag(QSGGeometryNode::ownsGeometry, true);
    }
    [...]

Exporting a Scene Graph Item Class (cont'd)
Exporting a Scene Graph Item Class (cont'd)

```cpp
20 [...]  
21    const QRectF rect = boundingRect();  
22    QSGGeometry *geometry = new QSGGeometry(  
23        QSGGeometry::defaultAttributes_Point2D(), 3);  
24    
25    QSGGeometry::Point2D *points = geometry->vertexDataAsPoint2D();  
26    points[0].x = rect.left();  
27    points[0].y = rect.top();  
28    points[1].x = rect.left() + rect.width() / 2.0;  
29    points[1].y = rect.bottom();  
30    points[2].x = rect.right();  
31    points[2].y = rect.top();  
32    
33    triangle->setGeometry(geometry);  
34    
35    QSGFlatColorMaterial *material = new QSGFlatColorMaterial;  
36    material->setColor(Qt::blue);  
37    triangle->setMaterial(material);  
38    
39    return triangle;  
40 }
```
Exporting a Scene Graph Item Class (cont'd)

```cpp
#include <QGuiApplication>
#include <QQuickView>
#include "triangleitem.h"

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);
    qmlRegisterType<TriangleItem>("Shapes", 1, 0, "Triangle");
    QQuickView view;
    view.setSource(QUrl("qrc:/main.qml"));
    view.setResizeMode(QQuickView::SizeRootObjectToView);
    view.show();
    return app.exec();
}
```
Exporting a Scene Graph Item Class (cont'd)

In the *main.qml* file:

```qml
import QtQuick 2.0
import Shapes 1.0

Item {
    width: 300; height: 200

    Triangle {
        anchors.centerIn: parent
        width: parent.width/2
        height: parent.height/2
    }
}
```
A Slightly More Elaborate Example

- Builds on the previous example
- Creates a small tree of QSGNodes
  - Adds a QSGTransformNode as a parent of the QSGGeometryNode to rotate
  - Periodically changes per-vertex data (color)

Demo qml-cpp-integration/ex-animated-item-scenegraph
Exporting a Raw OpenGL-based Item

- Reuse your existing OpenGL code in a few steps.
- **Inherit** `QQuickFramebufferObject::Renderer`.
  - Reimplement `createFramebufferObject()` to control the FBO format.
  - Reimplement `render()` to call your OpenGL code.
  - Ensure that your OpenGL code is compatible with the `QOpenGLContext` used by `QQuickWindow` (see `setFormat(QSurfaceFormat)`).
- **Inherit** `QQuickFramebufferObject`.
  - Reimplement `createRenderer()` to return your own renderer.
- Export the new type as usual.

Qt Demo qml-cpp-integration/textureinsgnode
Recommendations

- **QML: Canvas item**
  - Only useful for non-complex items
  - Useful if you don't intend to write any C++ ;)

- **C++: QQuickPaintedItem**
  - Using QQuickPaintedItem uses an indirect 2D surface to render its content
    - So the rendering is a two-step operation. First rasterize the surface, then draw the surface.
  - Does not implement edge antialiasing, only QPainter-based antialiasing

- **C++: Reimplementing QQuickItem::updatePaintNode()**
  - Relatively easy to use, hardware accelerated
  - Building blocks for creating custom geometries & materials

- **C++: QQuickFrameBufferObject**
  - Useful for over-/underlaying complete OpenGL scenes
  - Tricky to make sure custom OpenGL code does not interfere with QtQuick renderer
Thank you!

www.kdab.com

kevin.funk@kdab.com