Custom Qt Quick Components Using OpenGL

Kevin Funk, Software Engineer at KDAB

QtCon

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

```
1  #include <QObject>
2
3  class QTimer;
4
5  class RandomTimer : public QObject
6  {
7      Q_OBJECT
8  public:
9      RandomTimer(QObject *parent = 0);
10
11 private:
12     QTimer* m_timer;
13 };
```

Implementing the Element

- `RandomTimer` is a `QObject` subclass.
- As with all `QObject`s, each item can have a parent.
- Non-GUI custom items do not need to worry about any painting.

Step 1: Implementing the Element

```
1  #include "randomtimer.h"
2  #include <QTimer>
3
4  RandomTimer::RandomTimer(QObject *parent)
5      : QObject(parent),
6        m_timer(new QTimer(this))
7  {
8      m_timer->setInterval( 1000 );
9      m_timer->start();
10 }
```

Step 2: Registering the Element

```
1  #include <QGuiApplication>
2  #include <QQuickView>
3  #include "randomtimer.h"
4
5  int main(int argc, char *argv[])
6  {
7      QGuiApplication app(argc, argv);
8      qmlRegisterType<RandomTimer>("CustomComponents", 1, 0, "RandomTimer");
9
10     QQuickView view;
11     view.setSource(QUrl("qrc:///main.qml"));
12     view.show();
13     return app.exec();
14 }
```

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

Reviewing the Registration

```
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`

Step 3+4 Importing and Using the Element

In the *main.qml* file:

```
 1 import QtQuick 2.0
 2 import CustomComponents 1.0
 3
 4 Rectangle {
 5     width: 500
 6     height: 360
 7
 8     RandomTimer {
 9         id: timer
10         ...
11
12     }
13 }
```

Demo qml-cpp-integration/ex_simple_timer

Creating New QML Elements

- Registering & Using Custom Elements
- **Creating GUI elements**

Creating a Custom QML Element

KDAB

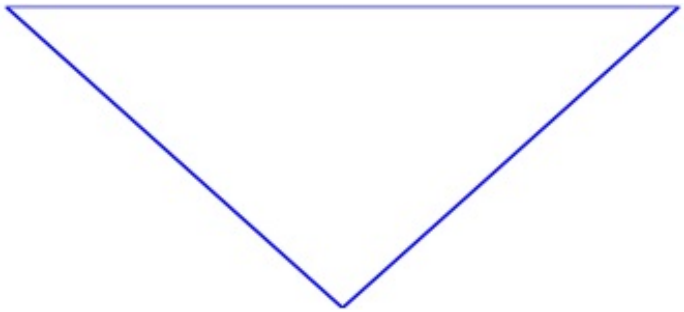
- Canvas item (slow) [QML]
  - Provides a 2D canvas item
  - Allows to draw primitives using JavaScript (HTML5-like API)
- Painted items (slow) [C++]
  - Subclass `QQuickPaintedItem`
  - Implement `paint(...)`
- Scene graph items (hardware accelerated) [C++]
  - Subclass `QQuickItem`
  - Implement `updatePaintNode(...)`
- 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:



```
1  Canvas {
2      id: triangle
3
4      onPaint: {
5        var ctx = getContext("2d");
6
7        ctx.beginPath();
8
9        ctx.moveTo(0, 0);
10       ctx.lineTo(triangleWidth, 0);
11       ctx.lineTo(0.5 * triangleWidth,triangleHeight);
12
13       ctx.closePath();
14
15 }
```

Creating a Custom Canvas item (cont'd)

In the *main.qml* file:

```
 1 Window {
 2     id: root
 3     visible: true
 4
 5     width: 800
 6     height: 600
 7
 8     TriangleCanvas {
 9         anchors.fill: parent
10         anchors.margins: 10
11     }
12
13 }
```

Demo qml-cpp-integration/ex-canvas-item

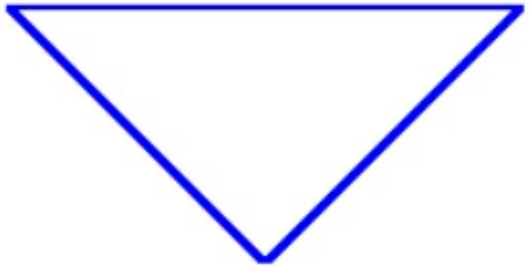
Qt Docs Qt Quick Canvas

Creating a Custom QML Element

Exporting a Painted Item Class

Exporting a Painted Item Class

```
1  #include <QQuickPaintedItem>
2
3  class TriangleItem : public QQuickPaintedItem
4  {
5      Q_OBJECT
6      Q_PROPERTY(QColor color READ color
7                 WRITE setColor NOTIFY colorChanged)
8
9  public:
10     explicit TriangleItem(QQuickItem *parent = 0);
11     void paint(QPainter *painter) override;
```



Exporting a Painted Item Class (cont'd)

```
 1 TriangleItem::TriangleItem(QQuickItem *parent)
 2     : QQuickPaintedItem(parent)
 3 {
 4 }
 5
 6 void TriangleItem::paint(QPainter *painter)
 7 {
 8     const auto rect = contentsBoundingRect();
 9     const QPolygonF trianglePolygon({
10         {0, 0},
11         {rect.width(), 0},
12         {0.5 * rect.width(), rect.height()},
13         {0, 0}
14     });
15     painter->drawPolygon(trianglePolygon);
16 }
```

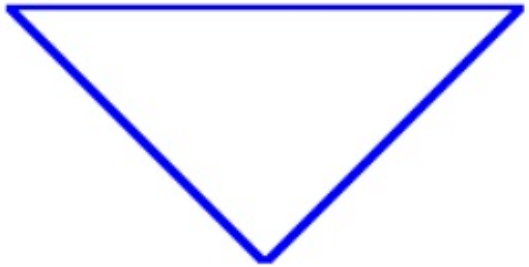
Exporting a Painted Item Class (cont'd)

```
#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:

```
 1 import Shapes 1.0
 2
 3 Item {
 4     width: 300; height: 200
 5
 6     Triangle {
 7         id: ellipse
 8         anchors.fill: parent
 9         anchors.margins: 50
10     }
11
12 }
```



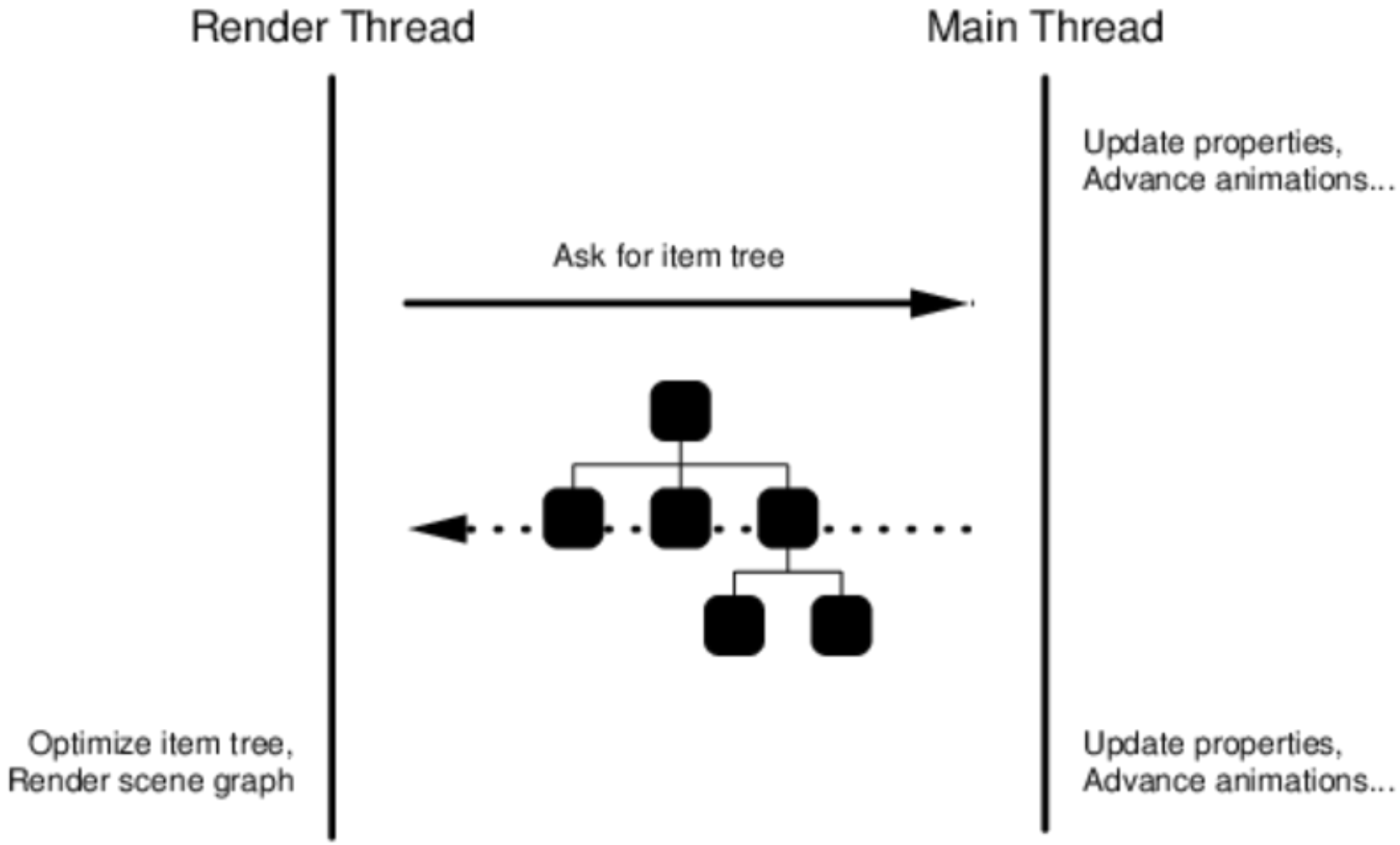
Demo qml-cpp-integration/ex-simple-item

Qt Docs QQuickPaintedItem **Qt Docs QPainter**

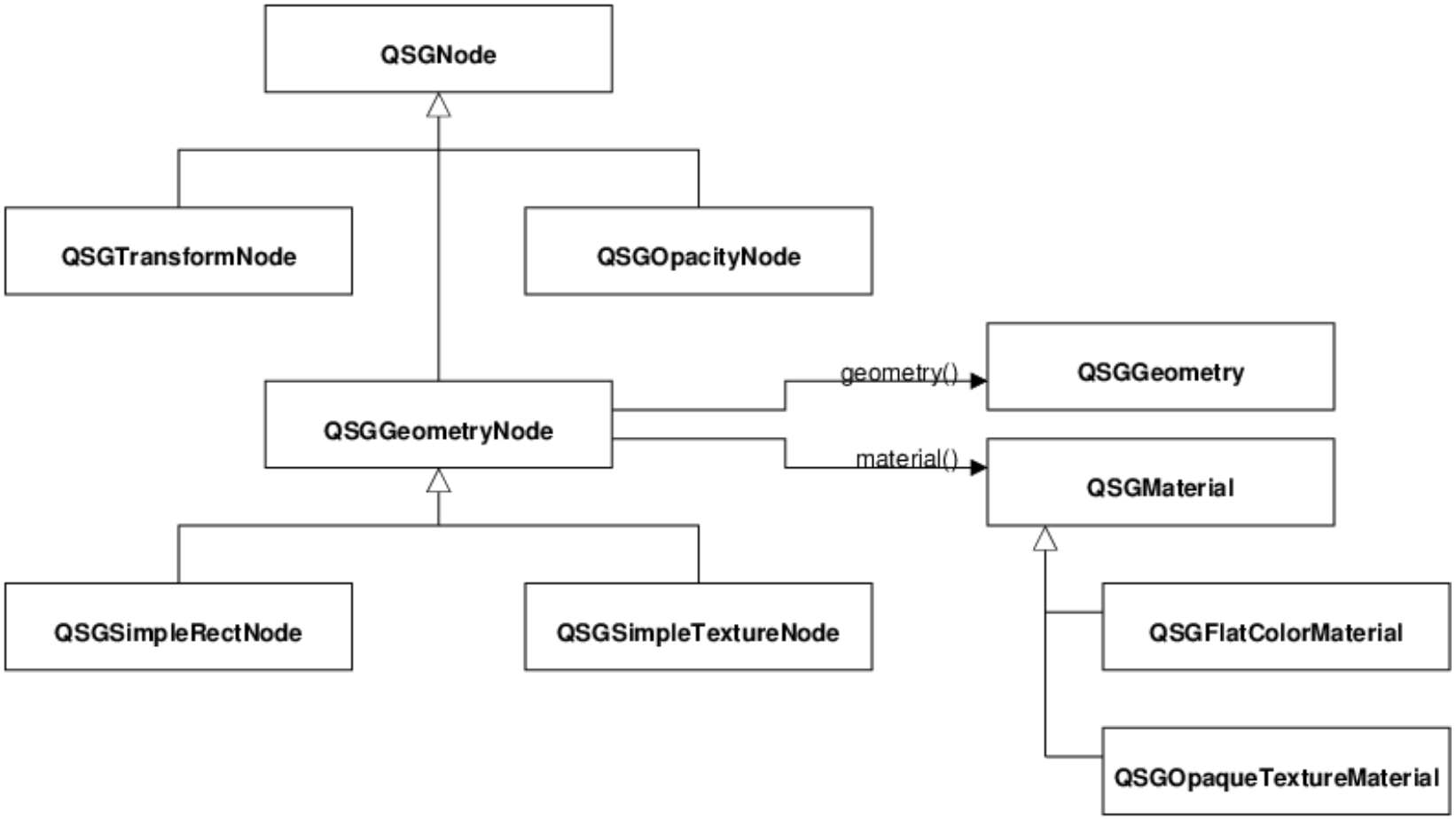
Creating a Custom QML Element

Exporting a Scene Graph Item Class

Introduction to the Qt Quick 2 Render Thread



Scene Graph API



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


```
1  #include <QQuickItem>
2
3  class TriangleItem : public QQuickItem
4  {
5      Q_OBJECT
6  public:
7      TriangleItem(QQuickItem *parent = 0);
8
9  protected:
10     QSGNode *updatePaintNode(QSGNode *oldNode,
11                              UpdatePaintNodeData *data) override;
12 };
```

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

Qt Docs QQuickItem **Qt Docs QSGNode** **Qt Docs QSGMaterial**

Exporting a Scene Graph Item Class (cont'd)

```
 1 TriangleItem::TriangleItem(QQuickItem *parent)
 2     : QQuickItem(parent)
 3 {
 4     setFlag(QQuickItem::ItemHasContents, true);
 5 }
 6
 7 QSGNode *TriangleItem::updatePaintNode(QSGNode *oldNode,
 8                                         UpdatePaintNodeData *)
 9 {
10     if (width() <= 0 || height() <= 0) {
11         delete oldNode;
12         return 0;
13     }
14
15     QSGGeometryNode *triangle = static_cast<QSGGeometryNode*>(oldNode);
16     if (!triangle) {
17         triangle = new QSGGeometryNode;
18         triangle->setFlag(QSGNode::OwnsMaterial, true);
19         triangle->setFlag(QSGNode::OwnsGeometry, true);
20     }
21
22     [...]
```

Exporting a Scene Graph Item Class (cont'd)

```
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)

```
1  #include <QGuiApplication>
2  #include <QQuickView>
3  #include "triangleitem.h"
4
5  int main(int argc, char *argv[])
6  {
7      QGuiApplication app(argc, argv);
8      qmlRegisterType<TriangleItem>("Shapes", 1, 0, "Triangle");
9
10     QQuickView view;
11     view.setSource(QUrl("qrc:/main.qml"));
12     view.setResizeMode(QQuickView::SizeRootObjectToView);
13     view.show();
14     return app.exec();
15 }
```

Exporting a Scene Graph Item Class (cont'd)

In the *main.qml* file:

```
 1 import QtQuick 2.0
 2 import Shapes 1.0
 3
 4 Item {
 5     width: 300; height: 200
 6
 7     Triangle {
 8         anchors.centerIn: parent
 9         width: parent.width/2
10         height: parent.height/2
11     }
12 }
```

A Slightly More Elaborate Example

- Builds on the previous example
- Creates a small tree of `QSGNode`s
  - 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