How to render programmatically a vtk item in qml?

Asked 3 years, 9 months ago Active 2 years, 3 months ago Viewed 3k times

3

1

So far, I understand that we have two threads in QML, our main application thread, and our "scene graph" thread : http://doc.qt.io/qt-5/qtquick-visualcanvas-scenegraph.html

I've implemented my own vtkQmlItem with the help of this link : http://doc.qt.io/qt-5/qtquick-scenegraph-openglunderqml-example.html

and I've noticed that my vtkscene is only rendered when the `afterrendering` signal is emitted by the qml flow.

So far, everything is ok and works perfectly, I can see my vtk scene and can even interract with it.

But I would like to also programmatically render my vtk scene as well, since I want to do an animation by moving the camera around a vtk object.

Calling `renderer->render()` directly shows a lot of vtk error, and does not seem to be the good way to do this.

Calling `this->window()->update()` seems to put the event in the eventLoop, when I want it to be handled instantly. The only way I've managed to make it work instantly is by using QApplication::processEvents(), which is a hack I don't like and would love another solution.

So the pseudocode of the working solution that I don't like is the following :

```
for (int i = 0; i < 50; i++)
{
   ChangeCameraPosition(i); // Change the position and orientation of the vtk camera
   this->window()->update();
   QApplication::processEvents(); // The hack I don't like
   QThread::msleep(500);
}
```

qt opengl events qml vtk

asked Jan 16 '17 at 22:14
asdfasdf
700 9 24

2 Answers

Active Oldest Votes

the problem is actually a bit complicated and if nothing changed in the past few months, there is still no support for QtQuick in VTK, which means no simple few lines solution. You can find derive support for qml. But mainly I recommend checking this thread for a discussion about this topic.

The key point is that QtQuick holds the openGL context for the qml window you are trying to render into in a dedicated thread and it won't let anything else get that context. So in order to render into it from VTK, you have to do it within that thread. This means:

1) Create your own vtkRenderWindow that overrides the Render() method such that it is ensured it happens in the qml's render thread.

2) Make that render window render into a framebuffer object provided by the qtquick (instance of QQuickFramebufferObject).

3) Interconnect vtk's rendering signals with the qt's rendering methods -> e.g. when the vtk render window calls makeCurrent, the qt's rendering thread "wakes up".

Here is my implementation based on Taylor Braun-Jones' template linked above. It might not be perfect, but it works for me (I have removed some parts specific to my app so it might not compile straight away, but it should put you on a path to some working solution):

qmlVtk.h:

```
#include <vtkEventQtSlotConnect.h>
#include <vtkGenericOpenGLRenderWindow.h>
#include <vtkRenderer.h>

#include <QtQuick/QQuickFramebufferObject>
// Use the OpenGL API abstraction from Qt instead of from VTK because vtkgl.h
// and other Qt OpenGL-related headers do not play nice when included in the
// same compilation unit
#include <QOpenGLFunctions>

#include <qqmlapplicationengine.h>

class QVTKFramebufferObjectRenderer;
class QVTKInteractorAdapter;
class vtkInternalOpenGLRenderWindow;
class QVTKFramebufferObjectRenderer;


class QVTKFrameBufferObjectItem : public QQuickFramebufferObject
{
    Q_OBJECT

public:
    QVTKFrameBufferObjectItem(QQuickItem *parent = 0);
    ~QVTKFrameBufferObjectItem();
    Renderer *createRenderer() const;
    vtkSmartPointer<vtkInternalOpenGLRenderWindow> GetRenderWindow() const;

protected:
    // Called once before the FBO is created for the first time. This method is
    // called from render thread while the GUI thread is blocked.
    virtual void init();

    vtkSmartPointer<vtkInternalOpenGLRenderWindow> m_win;
    QVTKInteractorAdapter* m_irenAdapter;
    // Convert the position of the event from openGL coordinate to native coordinate
    QMouseEvent openGLToNative(QMouseEvent const& event);

    virtual void mouseMoveEvent(QMouseEvent * event);
    virtual void mousePressEvent(QMouseEvent * event);
    virtual void mouseReleaseEvent(QMouseEvent * event);
    virtual void mouseDoubleClickEvent(QMouseEvent * event);
    virtual void wheelEvent(QWheelEvent *event);
    virtual void keyPressEvent(QKeyEvent* event);
    virtual void keyReleaseEvent(QKeyEvent* event);
    virtual void focusInEvent(QFocusEvent * event);
    virtual void focusOutEvent(QFocusEvent * event);


    protected Q_SLOTS:
    // slot to make this vtk render window current
    virtual void MakeCurrent();
    // slot called when vtk wants to know if the context is current
    virtual void IsCurrent(vtkObject* caller, unsigned long vtk_event, void*
client_data, void* call_data);
    // slot called when vtk wants to start the render
    virtual void Start();
    // slot called when vtk wants to end the render
    virtual void End();
    // slot called when vtk wants to know if a window is direct
    virtual void IsDirect(vtkObject* caller, unsigned long vtk_event, void*
client_data, void* call_data);
    // slot called when vtk wants to know if a window supports OpenGL
    virtual void SupportsOpenGL(vtkObject* caller, unsigned long vtk_event, void*
client_data, void* call_data);
};

/// <summary>
/// An extension of vktGenericOpenGLRenderWindow to work with Qt.
/// Serves to write VTK-generated render calls to a framebuffer provided and maintained
by Qt. It is meant to be used within Qt render loop, i.e. using Qt's render thread.
/// </summary>
/// <seealso cref="vtkGenericOpenGLRenderWindow" />
/// <seealso cref="QOpenGLFunctions" />
class vtkInternalOpenGLRenderWindow : public vtkGenericOpenGLRenderWindow, protected
QOpenGLFunctions
{

public:
    static vtkInternalOpenGLRenderWindow* New();
    vtkTypeMacro(vtkInternalOpenGLRenderWindow, vtkGenericOpenGLRenderWindow)

    virtual void OpenGLInitState();

    // Override to use deferred rendering - Tell the QSG that we need to
    // be rendered which will then, at the appropriate time, call
    // InternalRender to do the actual OpenGL rendering.
    virtual void Render();

    // Do the actual OpenGL rendering
    void InternalRender();

    // Provides a convenient way to set the protected FBO ivars from an existing
    // FBO that was created and owned by Qt's FBO abstraction class
    // QOpenGLFramebufferObject
    void SetFramebufferObject(QOpenGLFramebufferObject *fbo);

    QVTKFramebufferObjectRenderer *QtParentRenderer;

protected:
    vtkInternalOpenGLRenderWindow();
```

```
        // Prevent superclass destructors from destroying the framebuffer object.
        // QOpenGLFramebufferObject owns the FBO and manages it's lifecyle.
        this->OffScreenRendering = 0;
    }
};
```

qmlVtk.cpp:

```
#include "QVTKFramebufferObjectItem.h"

#include <QQuickFramebufferObject>
#include <QQuickWindow>
#include <QOpenGLFramebufferObject>
#include <QVTKInteractorAdapter.h>

#include <vtkRenderWindowInteractor.h>
#include <vtkObjectFactory.h>

#include <vtkSmartPointer.h>
#include <vtkCamera.h>
#include <vtkProperty.h>

#include <qglfunctions.h>


class QVTKFramebufferObjectRenderer : public QQuickFramebufferObject::Renderer
{
    friend class vtkInternalOpenGLRenderWindow;

public:
    QVTKFramebufferObjectRenderer(vtkSmartPointer<vtkInternalOpenGLRenderWindow> rw) :
        m_framebufferObject(0)
    {
        m_vtkRenderWindow = rw;

        m_vtkRenderWindow->QtParentRenderer = this;
    }

    ~QVTKFramebufferObjectRenderer()
    {
        m_vtkRenderWindow->QtParentRenderer = 0;
        glFrontFace(GL_CCW); // restore default settings
    }

    virtual void synchronize(QQuickFramebufferObject * item)
    {
        // the first synchronize call - right before the the framebufferObject
        // is created for the first time
        if (!m_framebufferObject)
        {
            QVTKFrameBufferObjectItem *vtkItem =
static_cast<QVTKFrameBufferObjectItem*>(item);
            vtkItem->init();
        }
    }

    virtual void render()
    {
        m_vtkRenderWindow->InternalRender(); // vtkXOpenGLRenderWindow renders the
scene to the FBO
    }

    QOpenGLFramebufferObject *createFramebufferObject(const QSize &size)
```

```
            m_vtkRenderWindow->SetFramebufferObject(m_framebufferObject);

            return m_framebufferObject;
        }

        vtkSmartPointer<vtkInternalOpenGLRenderWindow> m_vtkRenderWindow;
        QOpenGLFramebufferObject *m_framebufferObject;
};

vtkStandardNewMacro(vtkInternalOpenGLRenderWindow);

vtkInternalOpenGLRenderWindow::vtkInternalOpenGLRenderWindow() :
QtParentRenderer(0)
{
    vtkOpenGLRenderWindow::OpenGLInitContext();
}

void vtkInternalOpenGLRenderWindow::OpenGLInitState()
{
    this->MakeCurrent();
    vtkOpenGLRenderWindow::OpenGLInitState();
    // Before any of the gl* functions in QOpenGLFunctions are called for a
    // given OpenGL context, an initialization must be run within that context
    initializeOpenGLFunctions();
    glFrontFace(GL_CW); // to compensate for the switched Y axis
}

void vtkInternalOpenGLRenderWindow::InternalRender()
{
    vtkOpenGLRenderWindow::Render();
}

//
// vtkInternalOpenGLRenderWindow Definitions
//

void vtkInternalOpenGLRenderWindow::Render()
{
    this->QtParentRenderer->update();
}

void vtkInternalOpenGLRenderWindow::SetFramebufferObject(QOpenGLFramebufferObject *fbo)
{
    // QOpenGLFramebufferObject documentation states that "The color render
    // buffer or texture will have the specified internal format, and will
    // be bound to the GL_COLOR_ATTACHMENT0 attachment in the framebuffer
    // object"
    this->BackLeftBuffer = this->FrontLeftBuffer = this->BackBuffer = this->FrontBuffer
=
        static_cast<unsigned int>(GL_COLOR_ATTACHMENT0);

    // Save GL objects by static casting to standard C types. GL* types
    // are not allowed in VTK header files.
    QSize fboSize = fbo->size();
    this->Size[0] = fboSize.width();
    this->Size[1] = fboSize.height();
    this->NumberOfFrameBuffers = 1;
    this->FrameBufferObject = static_cast<unsigned int>(fbo->handle());
    this->DepthRenderBufferObject = 0; // static_cast<unsigned int>
(depthRenderBufferObject);
    this->TextureObjects[0] = static_cast<unsigned int>(fbo->texture());
    this->OffScreenRendering = 1;
    this->OffScreenUseFrameBuffer = 1;
    this->Modified();
}
```

```
    m_win->OpenGLInitState();
}

void QVTKFrameBufferObjectItem::End()
{
}


void QVTKFrameBufferObjectItem::MakeCurrent()
{
    this->window()->openglContext()->makeCurrent(this->window());
}

void QVTKFrameBufferObjectItem::IsCurrent(vtkObject*, unsigned long, void*, void*
call_data)
{
    bool* ptr = reinterpret_cast<bool*>(call_data);
    *ptr = this->window()->openglContext();
}

void QVTKFrameBufferObjectItem::IsDirect(vtkObject*, unsigned long, void*, void*
call_data)
{
    int* ptr = reinterpret_cast<int*>(call_data);
    *ptr = QGLFormat::fromSurfaceFormat(this->window()->openglContext()-
>format()).directRendering();
}

void QVTKFrameBufferObjectItem::SupportsOpenGL(vtkObject*, unsigned long, void*, void*
call_data)
{
    int* ptr = reinterpret_cast<int*>(call_data);
    *ptr = QGLFormat::hasOpenGL();
}


QVTKFrameBufferObjectItem::QVTKFrameBufferObjectItem(QQuickItem *parent) :
QQuickFramebufferObject(parent)
{
    setAcceptedMouseButtons(Qt::AllButtons);

    m_irenAdapter = new QVTKInteractorAdapter(this);
    m_win = vtkSmartPointer<vtkInternalOpenGLRenderWindow>::New();

    // make a connection between the vtk signals and qt slots so that an initialized
and madeCurrent opengl context is given to the vtk
    // we probably need only the Start(), MakeCurrent() and End() one, but just to be
sure...
    mConnect = vtkSmartPointer<vtkEventQtSlotConnect>::New();
    mConnect->Connect(m_win, vtkCommand::WindowMakeCurrentEvent, this,
SLOT(MakeCurrent()));
    mConnect->Connect(m_win, vtkCommand::WindowIsCurrentEvent, this,
SLOT(IsCurrent(vtkObject*, unsigned long, void*, void*)));
    mConnect->Connect(m_win, vtkCommand::StartEvent, this, SLOT(Start()));
    mConnect->Connect(m_win, vtkCommand::EndEvent, this, SLOT(End()));
    mConnect->Connect(m_win, vtkCommand::WindowIsDirectEvent, this,
SLOT(IsDirect(vtkObject*, unsigned long, void*, void*)));
    mConnect->Connect(m_win, vtkCommand::WindowSupportsOpenGLEvent, this,
SLOT(SupportsOpenGL(vtkObject*, unsigned long, void*, void*)));
}

QVTKFrameBufferObjectItem::~QVTKFrameBufferObjectItem()
{
    mConnect->Disconnect(); // disconnect all slots
    if (m_irenAdapter)
        delete m_irenAdapter;
```

```
{
   return new QVTKFramebufferObjectRenderer(m_win);
}

vtkSmartPointer<vtkInternalOpenGLRenderWindow>
QVTKFrameBufferObjectItem::GetRenderWindow() const
{
   return m_win;
}

void QVTKFrameBufferObjectItem::init()
{
}

// theoretically not needed now - the Y is being flipped in render and devicePixelRatio
will almost always be = 1 on a PC anyway...but lets keep it to be sure
QMouseEvent QVTKFrameBufferObjectItem::openGLToNative(QMouseEvent const& event)
{
    QPointF localPos(event.localPos());
    localPos.setX(localPos.x() * window()->devicePixelRatio());
    localPos.setY(localPos.y() * window()->devicePixelRatio());
    QMouseEvent nativeEvent(event.type(), localPos, event.button(), event.buttons(),
event.modifiers());
    return nativeEvent;
}

void QVTKFrameBufferObjectItem::mouseMoveEvent(QMouseEvent * event)
{
    m_win->GetInteractor()->SetSize(this->width(), this->height());
    QMouseEvent nativeEvent = openGLToNative(*event);
    m_irenAdapter->ProcessEvent(&nativeEvent, this->m_win->GetInteractor());
}

void QVTKFrameBufferObjectItem::mousePressEvent(QMouseEvent * event)
{
    m_win->GetInteractor()->SetSize(this->width(), this->height());
    QMouseEvent nativeEvent = openGLToNative(*event);
    m_irenAdapter->ProcessEvent(&nativeEvent, this->m_win->GetInteractor());
}

void QVTKFrameBufferObjectItem::mouseReleaseEvent(QMouseEvent * event)
{
    m_win->GetInteractor()->SetSize(this->width(), this->height());
    QMouseEvent nativeEvent = openGLToNative(*event);
    m_irenAdapter->ProcessEvent(&nativeEvent, this->m_win->GetInteractor());
}

void QVTKFrameBufferObjectItem::wheelEvent(QWheelEvent *event)
{
    m_irenAdapter->ProcessEvent(event, this->m_win->GetInteractor());
}


void QVTKFrameBufferObjectItem::keyPressEvent(QKeyEvent* event)
{
    m_irenAdapter->ProcessEvent(event, this->m_win->GetInteractor());
}

void QVTKFrameBufferObjectItem::keyReleaseEvent(QKeyEvent* event)
{
    m_irenAdapter->ProcessEvent(event, this->m_win->GetInteractor());
}
void QVTKFrameBufferObjectItem::focusInEvent(QFocusEvent * event)
{
    m_irenAdapter->ProcessEvent(event, this->m_win->GetInteractor());
}
```

```
    m_irenAdapter->ProcessEvent(event, this->m_win->GetInteractor());
}
```

To use it, define an instance of the framebuffer in your qml form and stretch it across the window you want to render into, e.g. like this (assuming you registered the QVTKFrameBufferObjectItem as a QVTKFrameBuffer in qml e.g. like this `qmlRegisterType<QVTKFrameBufferObjectItem>("VtkQuick", 1, 0, "QVTKFrameBuffer");`):

```
import VtkQuick 1.0
QVTKFrameBuffer
{
    id: renderBuffer
    anchors.fill : parent
    Component.onCompleted :
    {
        myCppDisplay.framebuffer = renderBuffer // tell the c++ side of your app that
this is the framebuffer into which it should render
    }
}
```

You then use the vtkRenderWindow you get by `myCppDisplay.framebuffer.GetRenderWindow()` the same way you would use any other vtkRenderWindow if you were rendering into a vtk-managed window, i.e. you can assign vtkRenderer to it, assign actors to that renderer, call theWindow.Render() as you wish and it will all be rendered into the qml component to which you assigned the framebuffer.

Two notes: 1) the vtk and qt use different coordinate system, you need to flip the y-coordinate...I am doing it by assigning a scale transformation to the camera, but there is plenty of other ways to do it:

```
vtkSmartPointer<vtkTransform> scale = vtkSmartPointer<vtkTransform>::New();
scale->Scale(1, -1, 1);
renderer->GetActiveCamera()->SetUserTransform(scale);
```

2) things get quite tricky once you start using multiple threads - you have to make sure that you are not trying to render in two different threads, because they would compete for that one QtQuick's rendering thread. This does not mean only not calling renderWindow.Render() in parallel - that is easy to avoid - but you have to realize that that qt thread is used also for rendering the GUI, so you might get into trouble this way (updating GUI while doing VTK rendering).

edited Sep 20 '17 at 9:01

answered Feb 3 '17 at 12:23
tomj
909 6 16

6

For people looking for a solution for this using Qt QuickControls 2 and VTK 8, you can find a working example in this repository https://github.com/nicanor-romero/QtVtk with building instructions in the README.

Nicanor Romero Venier
61 1 1

How did you manage to overlay qml and vtk? I always had my application crash until I added the line : _putenv_s("QML_BAD_GUI_RENDER_LOOP", "1"); which I didn't see in your code – asdfasdf Jul 11 '18 at 20:33