Qt World Summit 2017

October 10-12 | Berlin, Germany

Integrate external content in QtQuick

Giulio Camuffo, Software Engineer at KDAB

KDAB

Integrate external content in QtQuick

- **Integrate external content in QtQuick**
  - Sharing the content
  - Displaying the content

The problem(s)

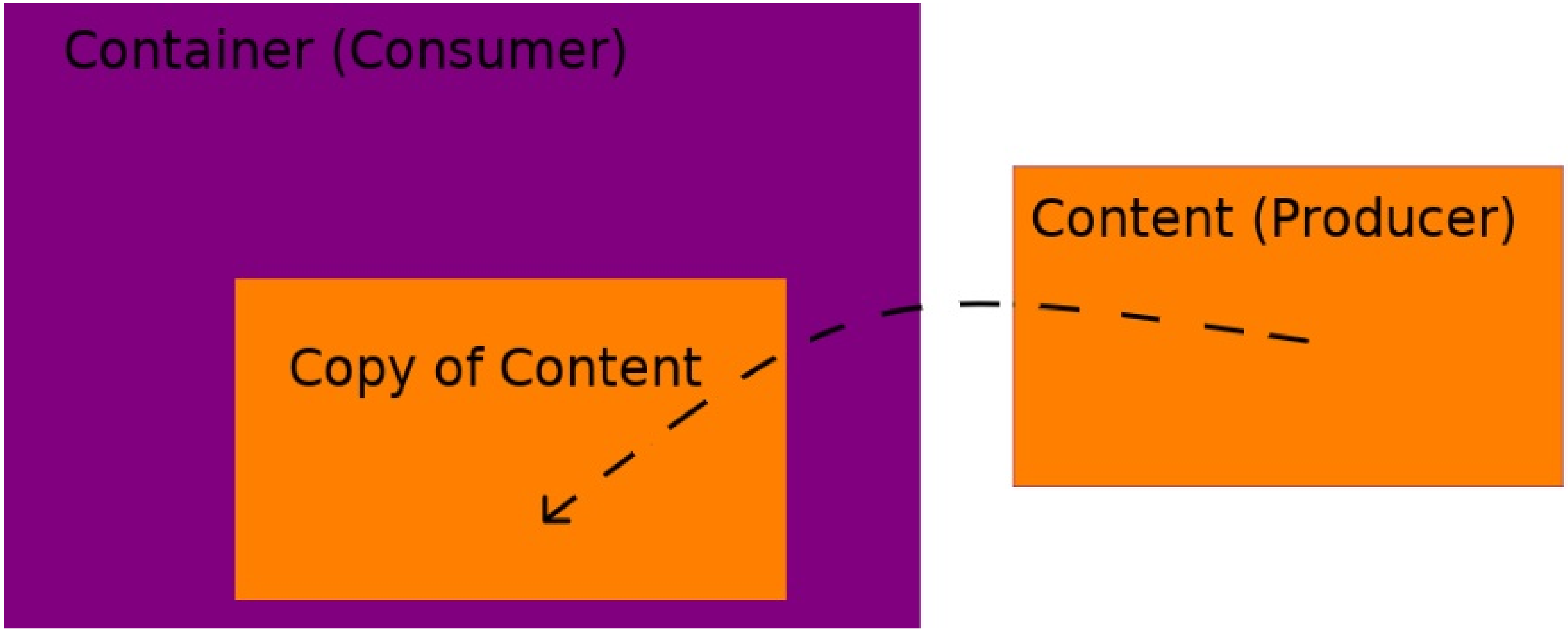
The problem(s):

- Losing frames;
- 3rd party content and no security;
- Rendering in parallel;

The solution

The solution:

- use a helper process to draw part of the content, and embed that content into your container application.



...But how do I actually do that? We can split the problem in two parts:

Integrate external content in QtQuick

- **Sharing the content**
  - Wayland
  - EGLStream
- Displaying the content

glReadPixels

```
// render your content
...
// transfer from GPU memory to CPU memory
char filename[] = "/tmp/tempfile-XXXXXX";
// create a temporary file
int fd = mkstemp(filename);
// map the file to access the memory backing it
char *pixels = mmap(nullptr, pixelsSize, PROT_READ | PROT_SIZE,
                    MAP_SHARED, fd, 0);
// fetch the rendered pixels from GPU memory to the file
glReadPixels(pixels);

// send the fd to the consumer via some IPC mechanism
send_fd(fd);
```

```
// on the consumer side:
int fd = receive_fd();

// map the memory from the fd, backing the same temporary file
char *pixels = mmap(nullptr, pixelsSize, PROT_READ, MAP_SHARED, fd, 0);
GLuint texture;
// create an OpenGL texture
glGenTexture(1, &texture);
// send the content of the file to GPU memory
glBindTexture(GL_TEXTURE_2D, texture);
glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0,
                format, GL_UNSIGNED_BYTE, pixels);

// use the texture...
```

It technically works

While this works it is very slow:

- It needs to first move the texture data from GPU to CPU memory, and back.
- glReadPixels() blocks until the rendering is done, making the usually asynchronous GPU rendering synchronous.

Can we do better?

Yes we can

There are various mechanisms to share GPU memory between processes:

- Wayland (Linux, some BSDs)
- ivi-share (Wayland protocol extension)
- EGLStream (Linux)
- gralloc (Android)
- dmabuf (Linux)
- Others...

We can leverage them to avoid the GPU ←→ CPU memory roundtrips.

Sharing the content

- **Wayland**
- EGLStream

Wayland (cont'd)

- Wayland is the boring case, as we have everything already ready in QtWayland:
- Your main application becomes a Wayland compositor, and the producer application a Wayland client.

Original code:

```
import QtWayland.Window 2.2
import QtQuick 2.6

Window {
    id: appWindow
    width: 1000
    height: 1000
    visible: true

    MyUiItem {
        id: uiItem

        ContentItem {
            ...
        }
    }
}
```

Wayland (cont'd)

New code:

```
import QtWayland.Compositor 1.0
import QtQuick.Window 2.2
import QtQuick 2.6

WaylandCompositor {
    WaylandOutput {
        sizeFollowsWindow: true
        // we have the original code put here, minus the content item
        window: Window {
            id: appWindow
            width: 1000
            height: 1000
            visible: true

            MyUiItem { id: uiItem }
        }
    }

    WlShell {
        // when a client creates a surface, create an item wrapping it, and make it
        // a child of uiItem
        onWlShellSurfaceCreated:
            itemComponent.createObject(uiItem, { "shellSurface": shellSurface })
    }
    Component { id: itemComponent; ShellSurfaceItem {} }
}
```

Wayland (cont'd)

The client application is then told to use Wayland, with one of the following methods:

- Passing the argument "-platform wayland", to the command line,
- or setting the environment variable "QT_QPA_PLATFORM=wayland".

Its QML code is taken out from the original application code to live on its own:

```
import QtQuick 2.6

ContentItem {
    ...
}
```

Sharing the content

- Wayland
- **EGLStream**

EGLStream (cont'd)

EGLStream, contrary to Wayland, has no ready made library to embed it in QtQuick apps.

```
#include <qpa/qplatformnativeinterface.h>

// get the EGL display used by Qt
EGLDisplay display = static_cast<EGLDisplay>(
                            QGuiApplication::platformNativeInterface()->
                                nativeResourceForIntegration("egldisplay"));
EGLStreamKHR stream = eglCreateStreamKHR(display);

//create a texture
GLuint texture;
glGenTexture(1, &texture);
glBindTexture(GL_TEXTURE_EXTERNAL_OES, texture);

//attach the texture to the consumer
eglStreamConsumerGLTextureExternalKHR(display, stream);

int fd = eglGetStreamFileDescriptorKHR(display, stream);

//send the fd to the producer using some IPC mechanism
send_fd(fd);

//acquire the stream to the texture
eglStreamConsumerAcquireKHR(display, stream);
```

EGLStream (cont'd)

```
//on the content producer side:
int fd = receive_fd();

// create a stream using the fd, this way they will refer to the same internal object
EGLStreamKHR stream = eglCreateStreamFromFileDescriptorKHR(display, fd);
EGLSurface surface = eglCreateStreamProducerSurfaceKHR(display, config,
                                                       stream, attributes);

//make the surface current and draw as usual with normal OpenGL
eglMakeCurrent(display, surface, surface, context);
...
eglSwapBuffers(display, surface);
```

A custom QPA plugin will be needed for a Qt client

Integrate external content in QtQuick

- Sharing the content
- **Displaying the content**
  - Behind the UI
  - Above the UI
  - Part of the UI

Displaying the content (cont'd)

Now that we have the content in a texture we need to show it, we can:

- Show it behind the QML ui;
- Show it above the QML ui;
- Show it as part of the QML ui;

Displaying the content

- **Behind the UI**
- Above the UI
- Part of the UI

Behind the UI (cont'd)

This is a viable aproach for cases where the QML ui stays always only on top, such as in games.

```
QQuickView view(QUrl("myqmlfile.qml"));
connect(&view, &QQuickWindow::onBeforeRendering, [&]() {
    //when using EGLStream here is a good place to acquire the texture

    //draw the texture
    glBindTexture(GL_TEXTURE_2D, texture)
    glDrawArrays(...)

    //the next call is important to make sure the GL state is how the scenegraph
    //expects it
    view.resetOpenGLState();
}, Qt::DirectConnection); //must be direct connection, because the signal is emitted
                          //from the rendering thread
```

Displaying the content

- Behind the UI
- **Above the UI**
- Part of the UI

Above the UI (cont'd)

This is basically the same, but it's drawing the content on top of the UI.

```
QQuickView view(QUrl("myqmlfile.qml"));
connect(&view, &QQuickWindow::onAfterRendering, [&]() {
    //when using EGLStream here is a good place to acquire the texture

    //draw the texture
    glBindTexture(GL_TEXTURE_2D, texture)
    glDrawArrays(...)

    //the next call is important to make sure the GL state is how the scenegraph
    //expects it
    view.resetOpenGLState();
}, Qt::DirectConnection); //must be direct connection, because the signal is emitted
                          //from the rendering thread
```

Displaying the content

- Behind the UI
- Above the UI
- **Part of the UI**

Part of the UI (cont'd)

This is the most flexible approach because it allows to put the out of process content in the middle of the scenegraph. We will need to declare a few classes:

```
struct MyMaterialState
{
    GLuint texture;
};

class MyMaterial : public QSGSimpleMaterialShader<MyMaterialState>
{
    QSG_DECLARE_SIMPLE_SHADER(ShareMaterial, ShareBufferTexture)
public:
    // standard vertex shader for a textured rect
    const char *vertexShader() const override
    {
        return "attribute highp vec4 vertex;\n"
               "attribute highp vec2 texcoord;\n"
               "uniform highp mat4 qt_Matrix;\n"
               "varying highp vec2 tex;\n"
               "void main() {\n"
               "    gl_Position = qt_Matrix * vertex;\n"
               "    tex = texcoord;\n"
               "}";
    }
```

Part of the UI (cont'd)

```
// standard fragment shader for a textured rect
    const char *fragmentShader() const override
    {
        return "uniform mediump sampler2D texture;\n"
                "uniform lowp float qt_Opacity;\n"
                "varying highp vec2 tex;\n"
                "void main() {\n"
                "    gl_FragColor = texture2D(texture, tex) * qt_Opacity;\n"
                "}";
    }

    QList<QByteArray> attributes() const override
    {
        return { "vertex", "texcoord" };
    }
};
```

Part of the UI (cont'd)

We then use the new classes in a QQuickItem subclass:

```
class MyItem : public QQuickItem
{
    void updatePaintNode(QSGNode *oldNode, UpdatePaintNodeData *data) override
    {
        QSGGeometryNode *node = static_cast<QSGGeometryNode *>(oldNode);
        if (!node) {
            node = new QSGGeometryNode;
            node->setGeometry(
                new QSGGeometry(QSGGeometry::defaultAttributes_TexturedPoint2D(), 4));
            // MyMaterial is a custom material class
            node->setMaterial(MyMaterial::createMaterial());
        }

        QSGGeometry::updateTexturedRectGeometry(node->geometry(),
                                                boundingRect(), QRect(0, 0, 1, 1));
        node->markDirty(QSGNode::DirtyGeometry);

        static_cast<MyMaterialState *>(node->material())->texture = texture;
        node->markDirty(SGNode::DirtyMaterial);

        return node;
    }
};
```

Part of the UI (cont'd)

Then we register MyItem to the QML engine, and use it in QML code:

```
qmlRegisterType<MyItem>(uri, 1, 0, "MyItem");
```

```
import QtQuick.Window 2.2
import QtQuick 2.6

Window {
    id: appWindow
    width: 1000
    height: 1000

    MyUiItem {
        id: uiItem

        MyItem {
            ...
        }
    }
}
```

Conclusion

- Splitting up your app makes it more robust
- Don't use glReadPixels()!
- Using QtWayland saves you time
- But there are other solutions

Thank you!

KDAB

www.kdab.com

giulio.camuffo@kdab.com