Qt 6 Shaders and materials for 2D/3D

László Agócs, Principal Software Engineer, The Qt Company, Oslo, Norway

Qt World Summit 2021 November 3, 2021

Where are application-provided shaders used in Qt?

- Qt Quick custom materials (C++) (QSGMaterial, QSGMaterialShader)
- **Qt Quick effects (QML) (ShaderEffect)**
- **Qt Quick 3D custom materials (QML) (CustomMaterial)**
- Qt Quick 3D post-processing effects (QML) (Effect)
- Others
  - Qt 3D, directly working with a graphics API, etc.

Material vs. effect

- Material: defines how a 3D object is rendered.
  - In practice the vertex and fragment shaders, plus some graphics pipeline state.
  - 3D object = a QSGGeometryNode in Quick or one sub-mesh of a Model in Quick3D.

- Effect: draws a textured quad using application-provided shaders that, typically, sample a texture containing a rendered, partial or full, 2D or 3D scene.

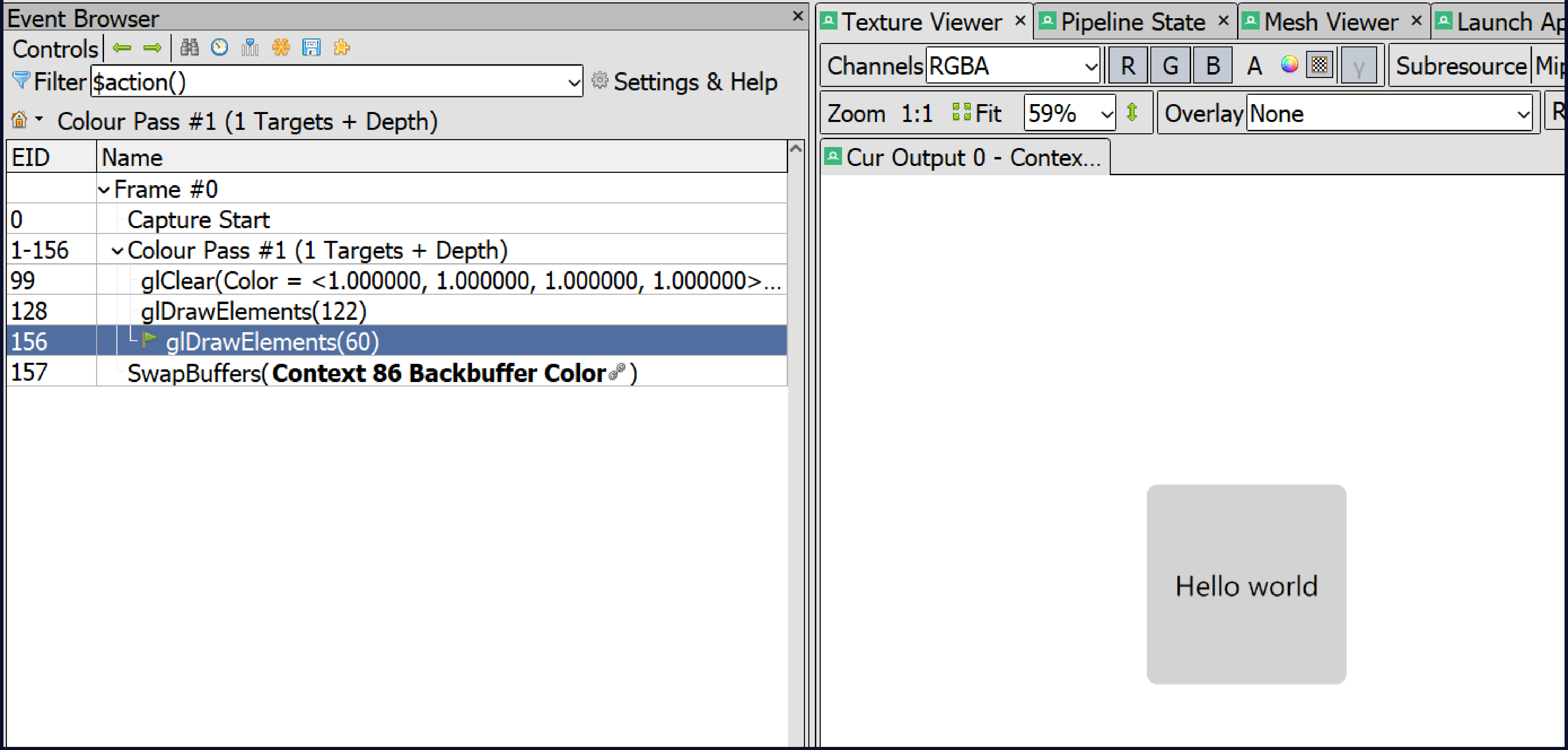
a bit simplified..

2D item layers: no layer

```
import QtQuick
Item {
    width: 1280; height: 720
    Item {
        width: 300; height: 300
        anchors.centerIn: parent
        Rectangle {
            id: effectItem
            anchors.fill: parent
            color: "lightgray"
            radius: 16
            Text {
                text: "Hello world"
                font.pointSize: 32
                anchors.centerIn: parent
            }
        }
    }
}
```

2D item layers: no layer, single pass, as expected

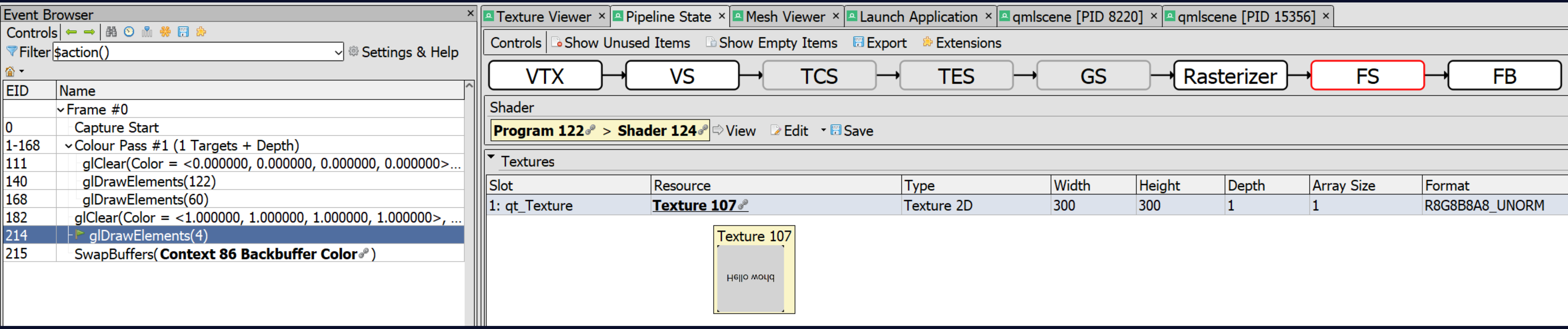
Frame capture with RenderDoc



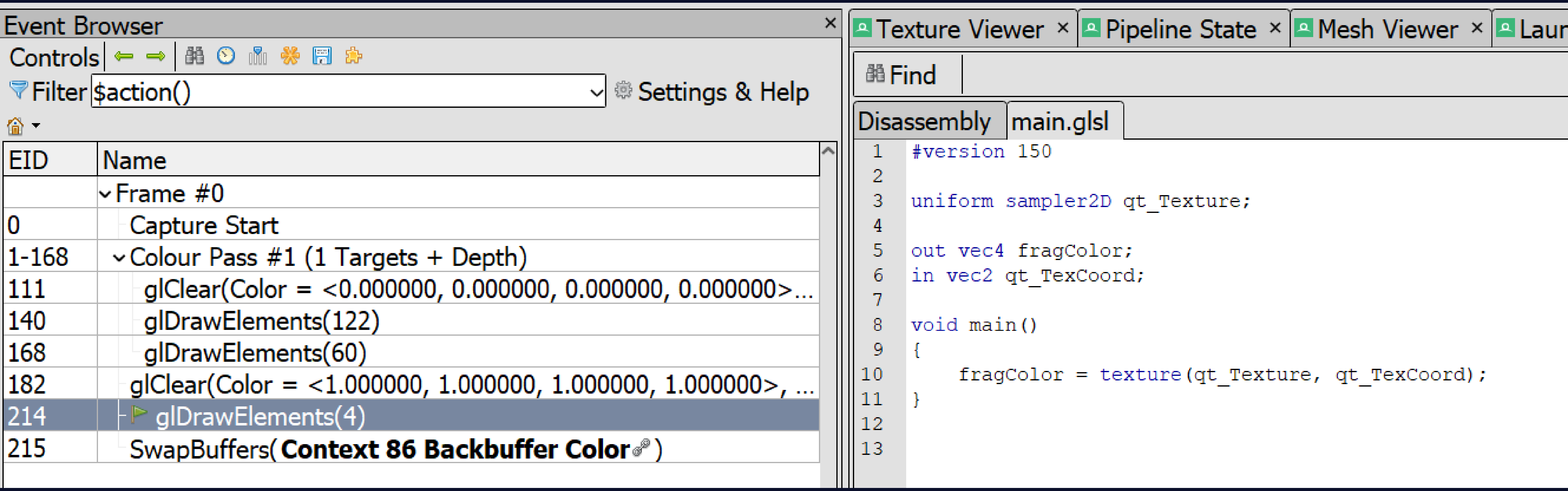
2D item layers: enable, looks the same...

```
import QtQuick
Item {
    width: 1280; height: 720
    Item {
        width: 300; height: 300
        anchors.centerIn: parent
        layer.enabled: true // !!
        Rectangle {
            id: effectItem
            anchors.fill: parent
            color: "lightgray"
            radius: 16
            Text {
                text: "Hello world"
                font.pointSize: 32
                anchors.centerIn: parent
            }
        }
    }
}
```

2D item layers: ...but two render passes



2D item layers: what is the fragment shader doing?



Towards ShaderEffect, 1

Let the user code specify the shader

- If we can render a subtree into a texture and then draw a textured quad, then the next step is to let the application do something more than just simply sampling the texture.

Towards ShaderEffect, 2

Let the user code specify the shader

- Expose QML properties to the shader with the same name and values, re-render when the values change.

In addition, any property that can be mapped to a GLSL type can be made available to the shaders. The following list shows how properties are mapped:

- bool, int, qreal -> bool, int, float - If the type in the shader is not the same as in QML, the value is converted automatically.
- QColor -> vec4 - When colors are passed to the shader, they are first premultiplied. Thus Qt.rgba(0.2, 0.6, 1.0, 0.5) becomes vec4(0.1, 0.3, 0.5, 0.5) in the shader, for example.
- QRect, QRectF -> vec4 - Qt.rect(x, y, w, h) becomes vec4(x, y, w, h) in the shader.
- QPoint, QPointF, QSize, QSizeF -> vec2
- QVector3D -> vec3
- QVector4D -> vec4
- QTransform -> mat3
- QMatrix4x4 -> mat4
- QQuaternion -> vec4, scalar value is w.
- Image -> sampler2D - Origin is in the top-left corner, and the color values are premultiplied. The texture is provided as is, excluding the Image item's fillMode. To include fillMode, use a ShaderEffectSource or Image::layer::enabled.
- ShaderEffectSource -> sampler2D - Origin is in the top-left corner, and the color values are premultiplied.

https://doc-snapshots.qt.io/qt6-dev/qml-qtquick-shadereffect.html

ShaderEffect (Qt 5)

```
import QtQuick 2.0
Item {
    width: 1280; height: 720
    Item {
        width: 300; height: 300
        anchors.centerIn: parent
        Rectangle {
            id: effectItem
            layer.enabled: true // !!
            visible: false // !!
            anchors.fill: parent
            color: "lightgray"
            radius: 16
            Text {
                text: "Hello world"
                font.pointSize: 32
                anchors.centerIn: parent
            }
        }
        ShaderEffect {
            anchors.fill: parent
            property variant source: effectItem
            fragmentShader: "
varying highp vec2 qt_TexCoord0;
uniform sampler2D source;
uniform lowp float qt_Opacity;
void main() {
    lowp vec4 c = texture2D(source, qt_TexCoord0);
    gl_FragColor = vec4(c.rgb * vec3(0.8, 0.2, 0.2), c.a) * qt_Opacity;
}"
        }
    }
}
```

ShaderEffect (Qt 6)

```
import QtQuick
Item {
    width: 1280; height: 720
    Item {
        width: 300; height: 300
        anchors.centerIn: parent
        Rectangle {
            id: effectItem
            layer.enabled: true // !!
            visible: false // !!
            anchors.fill: parent
            color: "lightgray"
            radius: 16
            Text {
                text: "Hello world"
                font.pointSize: 32
                anchors.centerIn: parent
            }
        }
        ShaderEffect {
            anchors.fill: parent
            property variant source: effectItem
            fragmentShader: "file:shader.frag.qsb" // !!!
        }
    }
}
```

API breaks in ShaderEffect in Qt 6

1. The vertexShader and fragmentShader properties are **not strings** anymore, but rather URLs.

2. They **must** refer to a file that is either local or in the resource system. (scheme must be file or qrc)

3. The file **must** be a .qsb file generated by the qsb command line tool.

Try the Qt 5 code with Qt 6

```
C:\Users\agocs\Documents\qtws21\code>qml shadereffect.qml
Failed to find shader "C:/Users/agocs/Documents/qtws21/code/\nvarying highp vec2 qt_TexCoord0;\nuniform sampler2D source;\nuni
form lowp float qt_Opacity;\nvoid main() {\n    lowp vec4 c = texture2D(source, qt_TexCoord0);\n    gl_FragColor = vec4(c.rgb
* vec3(0.8, 0.2, 0.2), c.a) * qt_Opacity;\n}"
ShaderEffect: Failed to deserialize QShader from C:/Users/agocs/Documents/qtws21/code/
varying highp vec2 qt_TexCoord0;
uniform sampler2D source;
uniform lowp float qt_Opacity;
void main() {
    lowp vec4 c = texture2D(source, qt_TexCoord0);
    gl_FragColor = vec4(c.rgb * vec3(0.8, 0.2, 0.2), c.a) * qt_Opacity;
}. Either the filename is incorrect, or it is not a valid .qsb file. In Qt 6 shaders must be preprocessed using the Qt Shader
Tools infrastructure. The vertexShader and fragmentShader properties are now URLs that are expected to point to .qsb files gen
erated by the qsb tool. See https://doc.qt.io/qt-6/qtshadertools-index.html for more information.
ShaderEffect: shader preparation failed for file:///C:/Users/agocs/Documents/qtws21/code/%0Avarying highp vec2 qt_TexCoord0;%0
Auniform sampler2D source;%0Auniform lowp float qt_Opacity;%0Avoid main() %7B%0A    lowp vec4 c = texture2D(source, qt_TexCoor
d0);%0A    gl_FragColor = vec4(c.rgb * vec3(0.8, 0.2, 0.2), c.a) * qt_Opacity;%0A%7D
```

ShaderEffect migration docs (Qt 6.1)

Migrating From Qt 5

For Qt 5 applications with ShaderEffect items the migration to Qt 6 involves:

- Moving the shader code to separate `.vert` and `.frag` files,
- updating the shaders to Vulkan-compatible GLSL,
- running the `qsb` tool on them,
- including the resulting `.qsb` files in the executable with the Qt resource system,
- and referencing the file in the vertexShader and fragmentShader properties.

As described in the Qt Shader Tools module some of these steps can be automated by letting CMake invoke the `qsb` tool at build time. See Qt Shader Tools Build System Integration for more information and examples.

When it comes to updating the shader code, below is an overview of the commonly required changes.

Vertex shader in Qt 5

```
attribute highp vec4 qt_Vertex;
attribute highp vec2 qt_MultiTexCoord0;
varying highp vec2 coord;
uniform highp mat4 qt_Matrix;
void main() {
    coord = qt_MultiTexCoord0;
    gl_Position = qt_Matrix * qt_Vertex;
}
```

Vertex shader in Qt 6

```
#version 440
layout(location = 0) in vec4 qt_Vertex;
layout(location = 1) in vec2 qt_MultiTexCoord0;
layout(location = 0) out vec2 coord;
layout(std140, binding = 0) uniform buf {
    mat4 qt_Matrix;
    float qt_Opacity;
};
void main() {
    coord = qt_MultiTexCoord0;
    gl_Position = qt_Matrix * qt_Vertex;
}
```

The conversion process mostly involves updating the code to be compatible with GL_KHR_vulkan_glsl. It is worth noting that Qt Quick uses a subset of the features provided by GLSL and Vulkan, and therefore the conversion process for typical ShaderEffect shaders is usually straightforward.

- The `version` directive should state 440 or 450, although specifying other GLSL version may work too, because the GL_KHR_vulkan_glsl extension is written for GLSL 140 and higher.
- Inputs and outputs must use the modern GLSL `in` and `out` keywords. In addition, specifying a location is required. The input and output location namespaces are separate, and therefore assigning locations starting from 0 for both is safe.
- When it comes to vertex shader inputs, the only possibilities with ShaderEffect are location 0 for vertex position (traditionally named `qt_Vertex`) and location 1 for texture coordinates (traditionally named `qt_MultiTexCoord0`).
- The vertex shader outputs and fragment shader inputs are up to the shader code to define. The fragment shader must have a `vec4` output at location 0 (typically called `fragColor`). For maximum portability, vertex outputs and fragment inputs should use both the same location number and the same name. When specifying only a fragment shader, the texture coordinates are passed in from the built-in vertex shader as `vec2 qt_TexCoord0` at location 0, as shown in the example snippets above.
- Uniform variables outside a uniform block are not legal. Rather, uniform data must be declared in a uniform block with binding point 0.
- The uniform block is expected to use the std140 qualifier.
- At run time, the vertex and fragment shader will get the same uniform buffer bound to binding point 0. Therefore, as a general rule, the uniform block declarations must be identical between the shaders. This also includes members that are not used in one of the shaders. The member names must match, because with some graphics APIs the uniform block is converted to a traditional struct uniform, transparently to the application.
- When providing one of the shaders only, watch out for the fact that the built-in shaders expect `qt_Matrix` and `qt_Opacity` at the top of the uniform block. (more precisely, at offset 0 and 64, respectively) As a general rule, always include these as the first and second

https://doc-snapshots.qt.io/qt6-dev/qml-qtquick-shadereffect.html#migrating-from-qt-5

Turning shader.frag into shader.frag.qsb

?

```
#version 440
layout(location = 0) in vec2 qt_TexCoord0;
layout(location = 0) out vec4 fragColor;
layout(std140, binding = 0) uniform buf {
    mat4 qt_Matrix;
    float qt_Opacity;
};
layout(binding = 1) uniform sampler2D source;
void main() {
    vec4 c = texture(source, qt_TexCoord0);
    fragColor = vec4(c.rgb * vec3(0.8, 0.2, 0.2), c.a) * qt_Opacity;
}
```

Manual way

```
qsb --glsl 100es,120,150 --hlsl 50 --msl 12 -o shader.frag.qsb shader.frag
```

Qt

```
qsb --glsl                                                              der.frag
```


Qt Shader Tools

Introduction

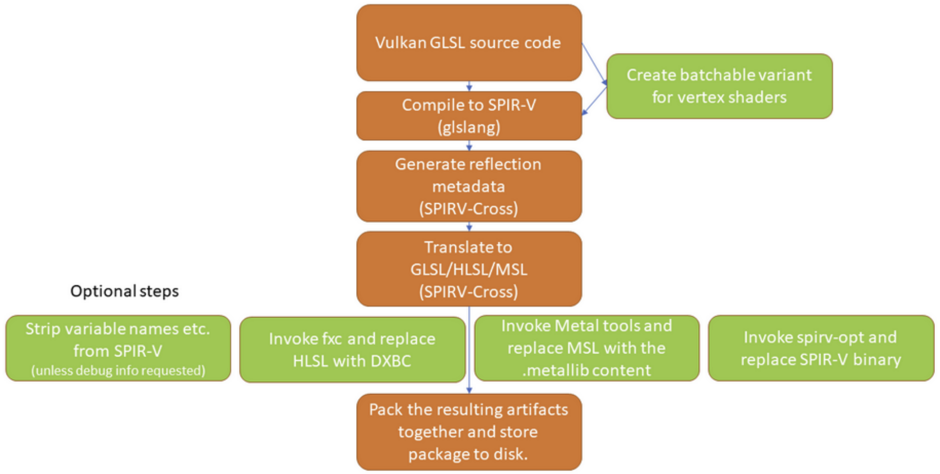
The Qt Shader Tools module builds on the SPIR-V Open Source Ecosystem as described at the Khronos SPIR-V web site. For compiling into SPIR-V glslang is used, while translating and reflecting is done via SPIRV-Cross.

In order to allow shader code to be written once in Qt applications and libraries, all shaders are expected to be written in a single language which is then compiled into SPIR-V. This shanding language is Vulkan-compatible GLSL at the moment. This is different from the OpenGL-style GLSL Qt 5.x expects. See the GL_KHR_vulkan_glsl specification for an overview of the differences.

Source code for other shading languages, such as, GLSL, HLSL and the Metal Shading Language, are then generated from translating the SPIR-V bytecode, together with reflection information (inputs, outputs, shader resources). For GLSL in particular, this also involves generating multiple variants, meaning source code suitable for different GLSL versions, such as, GLSL ES 100, GLSL ES 300, GLSL 120, 150, etc. This is then all packed into serializable QShader container objects, typically stored in disk in form of files an extension of `.qsb`. The Qt Rendering Hardware Interface consumes QShader instances directly, picking the shader source or bytecode that is best suited for the graphics API used at run time. A QShader object is typically deserialized from `.qsb` files shipped with the application or Qt itself in the Qt Resource System.

Some platforms provide the option of compiling shader source code to an intermediate format similar to SPIR-V. This involves running platform-specific tools. With Direct 3D, the `qsb` tool provides the option to invoke the fxc tool from the Windows SDK once the HLSL source code has been generated. It then replaces the HLSL source code with the DXBC binary generated by `fxc`. This can have a positive effect on applications' run time performance since they no longer need to do the first phase of compilation (HLSL source to DXBC) themselves. For macOS and iOS the XCode SDK provides similar tools. The downside of this approach that running these tools is only possible on their respective platforms. Therefore, this is best suited for use in combination with qsb's CMake integration since doing shader conditioning at application build time implicitly comes with the knowledge needed about the target platform and what platform-specific tools can be invoked.

The following diagram describes the steps that happen during an invocation of the `qsb` tool:



Qt

qsb --gl … frag

https://doc-snapshots.qt.io/qt6-dev/qtshadertools-qsb.html

Contents

Previous
Shader Tools

Reference
All Qt C++ Classes
All QML Types
All Qt Modules
Qt Creator Manual
All Qt Reference Documentation

Getting Started
Getting Started with Qt
What's New in Qt 6
Examples and Tutorials
Supported Platforms

QSB Manual

qsb is a command-line tool provided by the Qt Shader Tools module. It integrates third-party libraries such as glslang and SPIRV-Cross, optionally invokes external tools, such as, fxc or spirv-opt, and generates .qsb files. Additionally, it can be used to inspect the contents of a .qsb package.

```
Usage: qsb [options] file
Qt Shader Baker (using QShader from Qt 6.2.0)

Options:
  -?, -h, --help               Displays help on commandline options.
  --help-all                   Displays help including Qt specific options.
  -v, --version                Displays version information.
  -b, --batchable              Also generates rewritten vertex shader for Qt
                               Quick scene graph batching.
  --zorder-loc <location>      The extra vertex input location when rewriting
                               for batching. Defaults to 7.
  --glsl <versions>            Comma separated list of GLSL versions to
                               generate. (for example, "100 es,120,330")
  --hlsl <versions>            Comma separated list of HLSL (Shader Model)
                               versions to generate. F.ex. 50 is 5.0, 51 is 5.1.
  --msl <versions>             Comma separated list of Metal Shading Language
                               versions to generate. F.ex. 12 is 1.2, 20 is 2.0.
  -g                           Generate full debug info for SPIR-V and DXBC
  -O                           Invoke spirv-opt to optimize SPIR-V for
                               performance
  -o, --output <filename>      Output file for the shader pack.
  -c, --fxc                    In combination with --hlsl invokes fxc to store
                               DXBC instead of HLSL.
  -t, --metallib               In combination with --msl builds a Metal library
                               with xcrun metal(lib) and stores that instead of
                               the source.
  -D, --define <name[=value]>  Define macro. This argument can be specified
                               multiple times.
  -p, --per-target             Enable per-target compilation. (instead of
                               source->SPIRV->targets, do source->SPIRV->target
                               separately for each target)
  -d, --dump                   Switches to dump mode. Input file is expected to
                               be a shader pack.
  -x, --extract <what>         Switches to extract mode. Input file is expected
                               to be a shader pack. Result is written to the
                               output specified by -o. Pass -b to choose the
                               batchable variant.
                               <what>=reflect|spirv,<version>|glsl,<version>|...
```

The smart way: let CMake do it

- Real world applications will have a proper C++ entry point, with (increasingly) CMake as the build system.
- Good, because there is built-in support for invoking qsb at build time and automatically packing the results into the resource system.

The smart way: let CMake do it

```
qt6_add_shaders(app "app_shaders"
    PREFIX
        "/"
    FILES
        "shader.frag"
)
```

The smart way: let CMake do it

- Will feel natural if used qt6_add_resources before.
- The result is a :/shader.frag.qsb (qrc:/shader.frag.qsb for URLs) ready to be used in the application.
- Unless there are errors in the shader in which case the build breaks.
  - no more hunting run time debug prints
- Available since Qt 6.0, works best in 6.2.

```
qt6_add_shaders(app "app_shaders"
    PREFIX
        "/"
    FILES
        "shader.frag"
)
```

The smart way: let CMake do it

- shader.frag is not needed once the build completes, no need to ship it.
- qt(6)_add_shaders has a set of qsb-specific options besides FILES/PREFIX/BASE. See docs.
- Tip: always specify BATCHABLE when listing vertex shaders (.vert) that are used with Qt Quick.
- Tip: set PREFIX to where the .qml file is in the resource system, and use relative paths, e.g. `fragmentShader: “shader.frag.qsb”`

```
qt6_add_shaders(app "app_shaders"
    PREFIX
        "/"
    FILES
        "shader.frag"
)
```

https://doc-snapshots.qt.io/qt6-dev/qtshadertools-build.html

Blog Contact Us

Wiki Documentation Forum Bug Reports Code Review Resource Center Qt Extensions

Qt Documentation Snapshots

Qt 6.3 › Shader Tools › Qt Shader Tools Build System Integration

Contents

Previous

Shader Tools

Reference

All Qt C++ Classes
All QML Types
All Qt Modules
Qt Creator Manual
All Qt Reference Documentation

Getting Started

Getting Started with Qt
What's New in Qt 6
Examples and Tutorials

Qt Shader Tools Build System Integration

Introduction

The Qt Shader Tools module provides a CMake macro file that provides useful functions applications can take into use in their `CMakeLists.txt`.

When using the `qt6_add_shaders` function, the qsb tool will get invoked automatically by the build system, and the resulting `.qsb` files get added to the resource system implicitly.

First Example

Let's look at a simple example. Assume that we have a Qt Quick application that wants to provides its own wobble effect via ShaderEffect. The fragment shader is implemented in `wobble.frag`. The ShaderEffect item's fragmentShader property refers to `wobble.frag.qsb`. How do we ensure this .qsb file gets generated at build time?

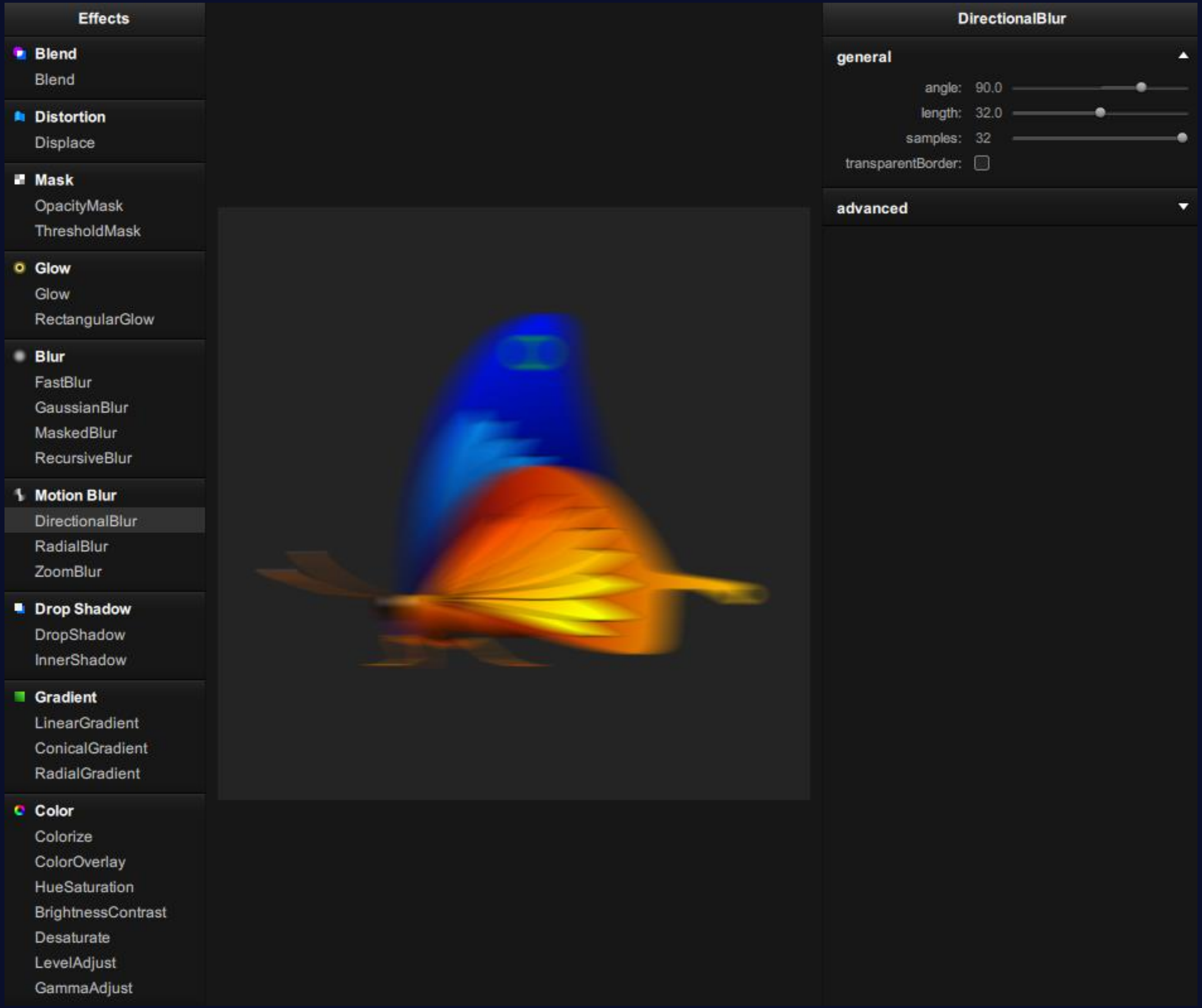
```
...
project(exampleapp LANGUAGES CXX)
...
find_package(Qt6 COMPONENTS ShaderTools)
...
qt6_add_executable(exampleapp
    main.cpp
)
...
qt6_add_resources(exampleapp "exampleapp"
    PREFIX
        "/"
    FILES
        "main.qml"
)

qt6_add_shaders(exampleapp "exampleapp_shaders"
    PREFIX
        "/"
    FILES
        "wobble.frag"
)
```

https://doc.qt.io/qt-6/qtshadertools-build.html

QtWS21QtWS21QtWS21QtWS21QtWS21QtWS21QtWS21QtWS21QtWS21QtWS21QtWS21QtWS21QtWS21

Qt Graphical Effects in Qt 5



- 25 effects
- No need to write shader code
- Some fairly complex and expensive (blurs)
- Combining multiple effects quickly becomes expensive

Why is combining expensive?

```
render to texture1

-> draw a quad to texture2 with custom shaders sampling texture1

-> draw a quad to texture3 with custom shaders sampling texture2

-> …

-> draw a quad to the backbuffer textured with textureN
```

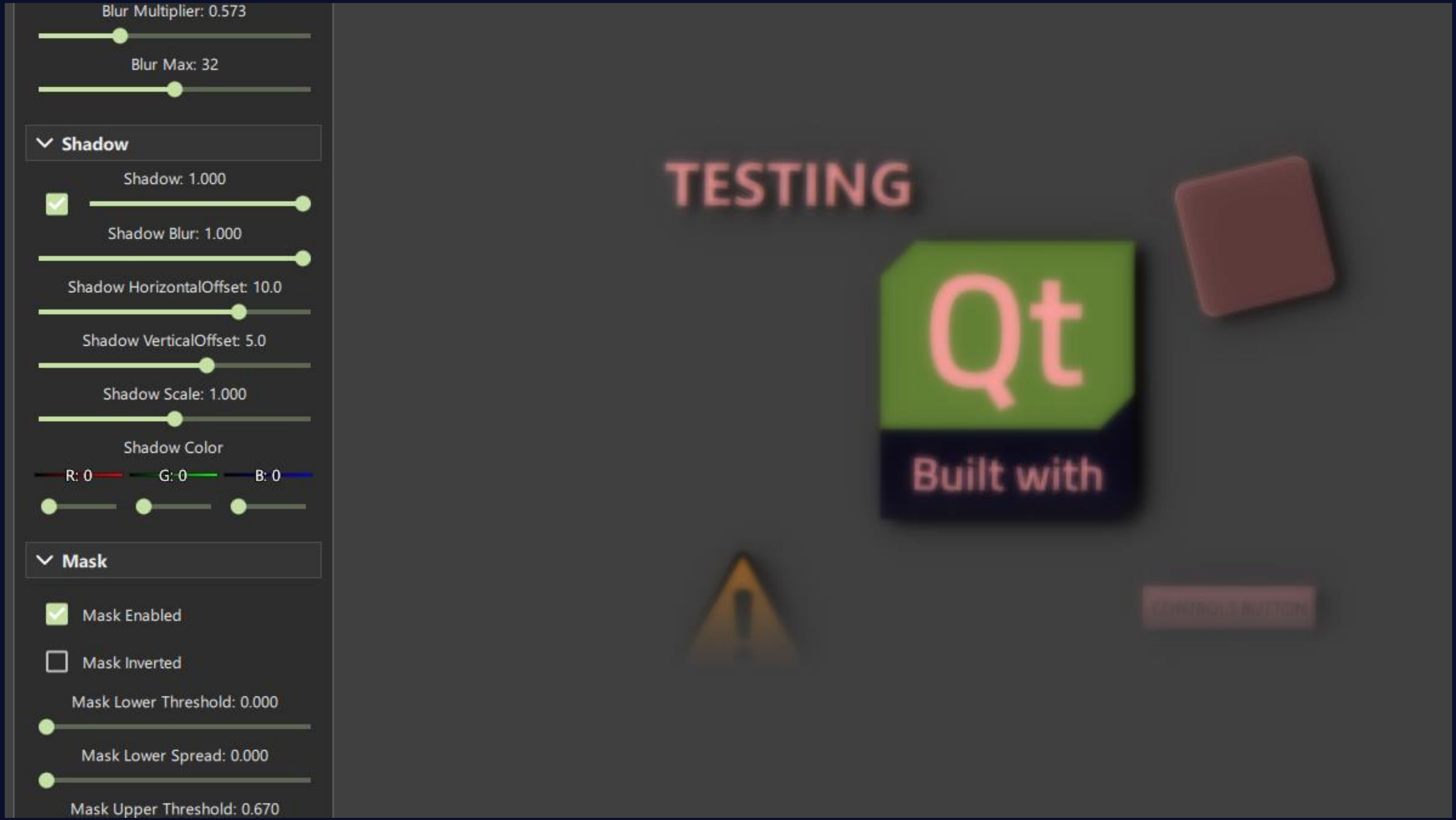
```
import QtQuick 2.0
import QtGraphicalEffects 1.0
Item {
    width: 1280; height: 720
    Item {
        width: 300; height: 300
        anchors.centerIn: parent
        Rectangle {
            id: effectItem
            visible: false
            anchors.fill: parent
            color: "lightgray"
            radius: 16
            Text {
                text: "Hello world"
                font.pointSize: 32
                anchors.centerIn: parent
            }
        }
        Colorize {
            id: colorizedItem
            anchors.fill: parent
            source: effectItem
            visible: false
        }
        DropShadow {
            anchors.fill: parent
            source: colorizedItem
            horizontalOffset: 8
            verticalOffset: 8
            color: "gray"
        }
    }
}
```

Qt Graphical Effects in Qt 6 (Qt5Compat module)

- 17 effects
- Meant as a porting aid
- Complex effects removed
- Works with all supported graphics APIs
- Added in Qt 6.1

```
import QtQuick
import Qt5Compat.GraphicalEffects
Item {
    width: 1280; height: 720
    Item {
        width: 300; height: 300
        anchors.centerIn: parent
        Rectangle {
            id: effectItem
            visible: false
            anchors.fill: parent
            color: "lightgray"
            radius: 16
            Text {
                text: "Hello world"
                font.pointSize: 32
                anchors.centerIn: parent
            }
        }
        Colorize {
            id: colorizedItem
            anchors.fill: parent
            source: effectItem
            visible: false
        }
        DropShadow {
            anchors.fill: parent
            source: colorizedItem
            horizontalOffset: 8
            verticalOffset: 8
            color: "gray"
        }
    }
}
```

Qt Quick MultiEffect



- Qt 5 version available on the Qt Marketplace
- Qt 6 version coming soon
- Blur, shadow, brightness, contrast, saturation, colorize, mask
- **Combining multiple effects has no additional cost**

```
Item {
    width: 1280; height: 720
    Item {
        width: 300; height: 300
        anchors.centerIn: parent
        Rectangle {
            id: effectItem
            visible: false
            anchors.fill: parent
            color: "lightgray"
            radius: 16
            Text {
                text: "Hello world"
                font.pointSize: 32
                anchors.centerIn: parent
            }
        }
        QuickMultiEffect {
            anchors.fill: parent
            source: effectItem
            colorizeEnabled: true
            colorizeColor: "red"
            colorize: 0.5
            shadowEnabled: true
            shadowHorizontalOffset: 8
            shadowVerticalOffset: 8
            blurEnabled: true
            blur: 0.5
        }
    }
}
```

NB! API subject to change

- The 3 effects (colorize, blur, dropshadow) are now combined.
- So more like 1 ShaderEffect, not 3 chained ones.

(blur/shadow are multi-pass depending on the properties hence all the additional render passes before the main one)

EID	Name
	⌄Frame #1
0	Capture Start
60	glClear(Color = <0.000000, 0.000000, 0.000000, 0.000000>, ...
89	glDrawElements(4)
104	glClear(Color = <0.000000, 0.000000, 0.000000, 0.000000>, ...
137	glDrawElements(6)
152	glClear(Color = <0.000000, 0.000000, 0.000000, 0.000000>, ...
184	glDrawElements(6)
199	glClear(Color = <0.000000, 0.000000, 0.000000, 0.000000>, ...
231	glDrawElements(6)
246	glClear(Color = <0.000000, 0.000000, 0.000000, 0.000000>, ...
278	glDrawElements(6)
288	glClear(Color = <0.250980, 0.250980, 0.250980, 1.000000>, ...
344	glDrawElements(6)
345	SwapBuffers(**Backbuffer Color**)

API Inspector

EID	Event
› 289	glDisable(GL_SCISSOR_TEST)
› 290	glDisable(GL_CULL_FACE)
› 291	glFrontFace(GL_CCW)

Disassembly | main.glsl

```
uniform sampler2D blurSrc3;
uniform sampler2D blurSrc4;
uniform sampler2D blurSrc5;
uniform sampler2D src;

in vec2 texCoord;
in vec2 shadowTexCoord;
out vec4 fragColor;

void main()
{
    vec4 color = texture(blurSrc1, texCoord) * _23.blurWeight1.x;
    color += (texture(blurSrc2, texCoord) * _23.blurWeight1.y);
    color += (texture(blurSrc3, texCoord) * _23.blurWeight1.z);
    color += (texture(blurSrc4, texCoord) * _23.blurWeight1.w);
    color += (texture(blurSrc5, texCoord) * _23.blurWeight2.x);
    vec3 _92 = ((color.xyz - vec3(0.5 * color.w)) * (1.0 + _23.contrast)) + vec3(0.5 * color.w);
    color = vec4(_92.x, _92.y, _92.z, color.w);
    vec3 _104 = color.xyz + vec3(_23.brightness * color.w);
    color = vec4(_104.x, _104.y, _104.z, color.w);
    float gray = dot(color.xyz, vec3(0.2989999949932098388671875, 0.58700001239776611328125, 0.114000000059604644775390625));
    vec3 _131 = ((_23.colorizeColor.xyz * gray) * _23.colorizeColor.w) + (color.xyz * (1.0 - _23.colorizeColor.w));
    color = vec4(_131.x, _131.y, _131.z, color.w);
    vec3 _143 = mix(vec3(gray), color.xyz, vec3(1.0 + _23.saturation));
    color = vec4(_143.x, _143.y, _143.z, color.w);
    float shadow = 0.0;
    shadow = texture(blurSrc1, shadowTexCoord).w * _23.shadowBlurWeight1.x;
    shadow += (texture(blurSrc2, shadowTexCoord).w * _23.shadowBlurWeight1.y);
    shadow += (texture(blurSrc3, shadowTexCoord).w * _23.shadowBlurWeight1.z);
    shadow += (texture(blurSrc4, shadowTexCoord).w * _23.shadowBlurWeight1.w);
    shadow += (texture(blurSrc5, shadowTexCoord).w * _23.shadowBlurWeight2.x);
    shadow *= _23.shadowColor.w;
    float aa = (1.0 - color.w) * (1.0 - shadow);
    vec3 _218 = mix(_23.shadowColor.xyz * shadow, color.xyz, vec3(color.w + aa));
    color = vec4(_218.x, _218.y, _218.z, color.w);
    color.w = 1.0 - aa;
    fragColor = color * _23.qt_Opacity;
```

Qt Quick 3D

```
import QtQuick
import QtQuick3D
Item {
    width: 1280
    height: 720
    View3D {
        anchors.fill: parent
        environment: SceneEnvironment {
            backgroundMode: SceneEnvironment.Color
            clearColor: "black"
        }
        PerspectiveCamera { z: 600 }
        DirectionalLight { }
        Model {
            source: "#Cube"
            scale: Qt.vector3d(2, 2, 2)
            eulerRotation.x: 30
            materials: CustomMaterial { }
         }
    }
}
```

Qt Quick 3D

CustomMaterial

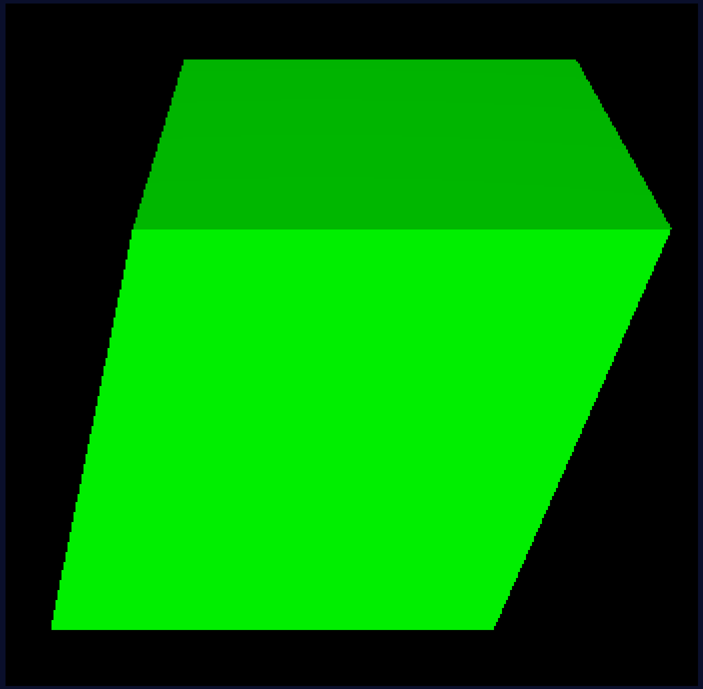
- More control over how a 3D model is rendered.
- Two modes
  - Shaded – augment a PrincipledMaterial.
  - Unshaded – loses all standard shading (lighting), up to the shader how it generates the final fragment color.
- A shaded CustomMaterial is similar to Godot’s spatial shaders or Unity’s surface shaders.
- An unshaded CustomMaterial is similar to Godot’s unshaded spatial shaders or Unity’s vertex/fragment shaders.

Qt Quick 3D

```
materials: CustomMaterial {
    vertexShader: "material.vert"
    fragmentShader: "material.frag"
    property real uTime
    property real uAmplitude: 50
    NumberAnimation on uTime { from: 0; to: 100; duration: 10000; loops: -1 }
}
```

```
void MAIN()
{
    VERTEX.x += sin(uTime + VERTEX.y) * uAmplitude;
}
```

```
void MAIN()
{
    BASE_COLOR = vec4(0.0, 1.0, 0.0, 1.0);
}
```



Qt Quick 3D

CustomMaterial

- Like with ShaderEffect, properties get mapped to uniforms.

Exposing data to the shaders

The dynamic properties of the CustomMaterial can be changed and animated using QML and Qt Quick facilities, and the values are exposed to the shaders automatically. This in practice is very similar ShaderEffect. The following list shows how properties are mapped:

- bool, int, real -> bool, int, float
- QColor, color -> vec4, and the color gets converted to linear, assuming sRGB space for the color value specified in QML. The built-in Qt colors, such as `"green"` are in sRGB color space as well, and the same conversion is performed for all color properties of DefaultMaterial and PrincipledMaterial, so this behavior of CustomMaterial matches those. Unlike Qt Quick, for Qt Quick 3D linearizing is essential as there will typically be tonemapping performed on the 3D scene.
- QRect, QRectF, rect -> vec4
- QPoint, QPointF, point, QSize, QSizeF, size -> vec2
- QVector2D, vector2d -> vec2
- QVector3D, vector3d -> vec3
- QVector4D, vector4d -> vec4
- QMatrix4x4, matrix4x4 -> mat4
- QQuaternion, quaternion -> vec4, scalar value is w
- TextureInput -> sampler2D - Textures referencing image files and Qt Quick item layers are both supported. Setting the enabled property to false leads to exposing a dummy texture to the shader, meaning the shaders are still functional but will sample a texture with opaque black image content. Pay attention to the fact that properties for samplers must always reference a TextureInput object, not a Texture directly. When it comes to the Texture properties, the source, tiling, and filtering related ones are the only ones that are taken into account implicitly with custom materials, as the rest (such as, UV transformations) is up to the custom shaders to implement as they see fit.

https://doc-snapshots.qt.io/qt6-dev/qml-qtquick3d-custommaterial.html#exposing-data-to-the-shaders

Qt Quick 3D

CustomMaterial

- Unlike Qt Quick (2D) materials and effects, the “shaders” here are really only snippets that contribute to the full shader the engine generates behind the scenes.

- Vulkan-compatible GLSL.
- Upper-case special keywords.

Qt Quick 3D

The 4 lines from material.frag are just 4 out of 252

Disassembly | main.glsl

```
in vec3 qt_varBinormal;
in vec2 qt_varTexCoord0;
in vec2 qt_varTexCoord1;
out vec4 fragOutput;

void qt_customMain(out vec4 BASE_COLOR, vec3 EMISSIVE_COLOR, float METALNESS, float ROUGHNESS, float SPECULAR_AMOUNT, float FRESNEL_POWER, vec3 NORMAL, vec3 TANGENT, vec3 BINORMAL, vec2 U
{
    BASE_COLOR = vec4(0.0, 1.0, 0.0, 1.0);
}

vec3 qt_F0(float metalness, float specular, vec3 baseColor)
{
    float dielectric = (0.1599999964237213134765625 * specular) * specular;
    return mix(vec3(dielectric), baseColor, vec3(metalness));
}

vec3 qt_principledMaterialFresnel(vec3 N, vec3 viewDir, float metalness, float specular, vec3 baseColor, float roughness, float fresnelPower)
{
    float nDotV = clamp(dot(N, viewDir), 0.0, 1.0);
    float param = metalness;
    float param_1 = specular;
    vec3 param_2 = baseColor;
    vec3 f0 = qt_F0(param, param_1, param_2);
    vec3 F = f0 + ((max(vec3(1.0 - roughness), f0) - f0) * pow(1.0 - nDotV, fresnelPower));
    return F;
}

float qt_schlick(float value)
{
    float n = 1.0 - value;
    float n2 = n * n;
    return (n2 * n2) * n;
}

vec4 qt_diffuseBurleyBSDF(vec3 normal, vec3 lightDirection, vec3 viewVector, vec3 lightDiffuse, float roughness)
{
    vec3 H = normalize(viewVector + lightDirection);
    float cLdotH = max(0.0, dot(lightDirection, H));
```

A custom vertex or fragment shader snippet is expected to provide one or more functions with pre-defined names, such as `MAIN`, `DIRECTIONAL_LIGHT`, `POINT_LIGHT`, `SPOT_LIGHT`, `AMBIENT_LIGHT`, `SPECULAR_LIGHT`. For now let's focus on `MAIN`.

As shown here, the end result with an empty MAIN() is exactly the same as before.

Before making it more interesting, let's look at an overview of the most commonly used special keywords in custom vertex shader snippets. This is not the full list. For a full reference, check the CustomMaterial page.

Keyword	Type	Description
MAIN		void MAIN() is the entry point. This function must always be present in a custom vertex shader snippet, there is no point in providing one otherwise.
VERTEX	vec3	The vertex position the shader receives as input. A common use case for vertex shaders in custom materials is to change (displace) the x, y, or z values of this vector, by simply assigning a value to the whole vector, or some of its components.
NORMAL	vec3	The vertex normal from the input mesh data, or all zeroes if there were no normals provided. As with VERTEX, the shader is free to alter the value as it sees fit. The altered value is then used by the rest of the pipeline, including the lighting calculations in the fragment stage.
UV0	vec2	The first set of texture coordinates from the input mesh data, or all zeroes if there were no UV values provided. As with VERTEX and NORMAL, the value can altered.
MODELVIEWPROJECTION_MATRIX	mat4	The model-view-projection matrix. To unify the behavior regardless of which graphics API rendering happens with, all vertex data and transformation matrices follow OpenGL conventions on this level. (Y axis pointing up, OpenGL-compatible projection matrix) Read only.
MODEL_MATRIX	mat4	The model (world) matrix. Read only.
NORMAL_MATRIX	mat3	The transposed inverse of the top-left 3x3 slice of the model matrix. Read only.
CAMERA_POSITION	vec3	The camera position in world space. In the examples on this page this is `(0, 0, 600)`. Read only.
CAMERA_DIRECTION	vec3	The camera direction vector. In the examples on this page this is `(0, 0, -1)`. Read only.
CAMERA_PROPERTIES	vec2	The near and far clip values of the camera. In the examples on this page this is `(10, 10000)`. Read only.
POINT_SIZE	float	Relevant only when rendering with a topology of points, for example because the custom geometry provides such a geometry for the mesh. Writing to this value is equivalent to setting pointSize on a PrincipledMaterial.
POSITION	vec4	Like `gl_Position`. When not present, a default assignment statement is generated automatically using `MODELVIEWPROJECTION_MATRIX` and `VERTEX`. This is why an empty MAIN() is functional, and in most cases there will be no need to assign a custom value to it.

Let's look at some of the commonly used keywords in fragment shaders. This is not the full list, refer to the CustomMaterial documentation for a complete reference. Many of these are read-write, meaning they have a default value, but the shader can, and often will want to, assign a different value to them.

As the names suggest, many of these map to similarly named PrincipledMaterial properties, with the same meaning and semantics, following the metallic-roughness material model. It is up the custom material implementation to decide how these values are calculated: for example, a value for BASE_COLOR can be hard coded in the shader, can be based on sampling a texture, or can be calculated based on QML properties exposed as uniforms or on interpolated data passed along from the vertex shader.

Keyword	Type	Description
BASE_COLOR	vec4	The base color and alpha value. Corresponds to PrincipledMaterial::baseColor. The final alpha value of the fragment is the model opacity multiplied by the base color alpha. The default value is `(1.0, 1.0, 1.0, 1.0)`.
EMISSIVE_COLOR	vec3	The color of self-illumination. Corresponds to PrincipledMaterial::emissiveFactor. The default value is `(0.0, 0.0, 0.0)`.
METALNESS	float	Metalness value in range 0-1. Default to 0, which means the material is dielectric (non-metallic).
ROUGHNESS	float	Roughness value in range 0-1. The default value is 0. Larger values soften specular highlights and blur reflections.
SPECULAR_AMOUNT	float	The strength of specularity in range 0-1. The default value is `0.5`. For metallic objects with `metalness` set to 1 this value will have no effect. When both SPECULAR_AMOUNT and METALNESS have values larger than 0 but smaller than 1, the result is a blend between the two material models.
NORMAL	vec3	The interpolated normal in world space, adjusted for double-sidedness when face culling is disabled. Read only.
UV0	vec2	The interpolated texture coordinates. Read only.
VAR_WORLD_POSITION	vec3	Interpolated vertex position in world space. Read only.

Qt Quick 3D

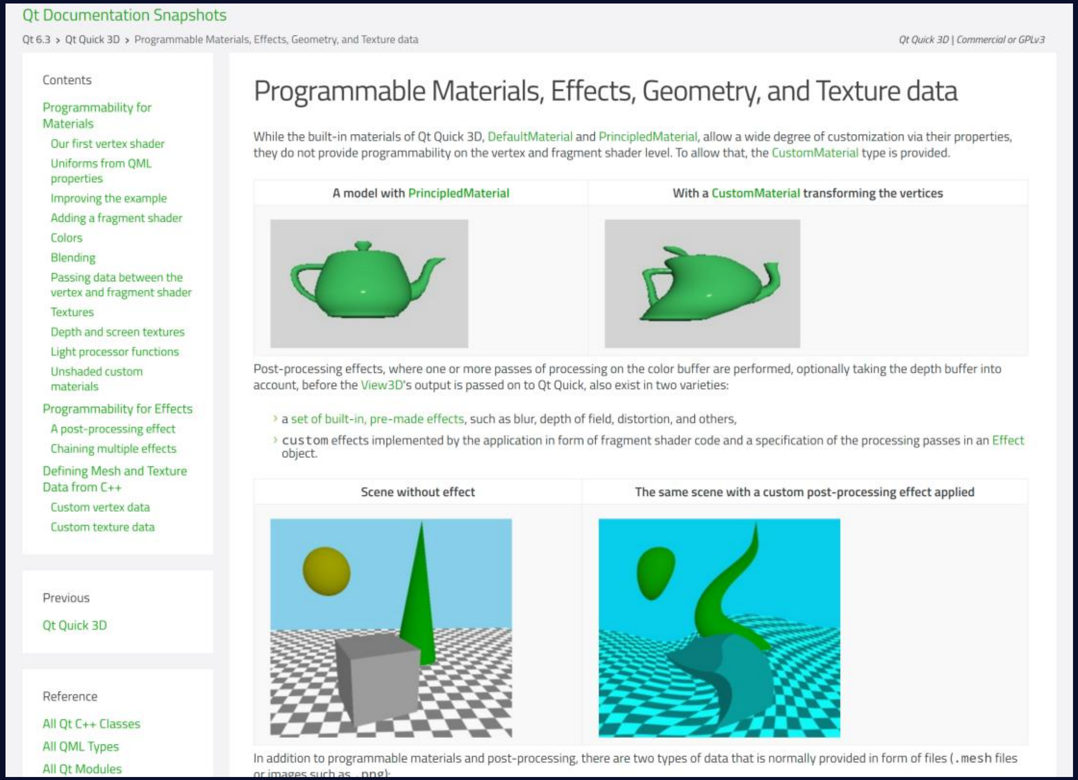
CustomMaterial

- The vertexShader and fragmentShader properties are URLs (scheme must be file or qrc).
- However, the files are *plain text shader snippets*.
- Assembling and processing via QtShaderTools *happens at run-time.*
  - Has pros and cons.
  - There are ways to work this around if absolutely needed (“build time materials”)
    - Also has pros and cons.

Qt Quick 3D

https://doc-snapshots.qt.io/qt6-dev/qtquick3d-custom.html

https://doc-snapshots.qt.io/qt6-dev/qml-qtquick3d-custommaterial.html



Thank you!