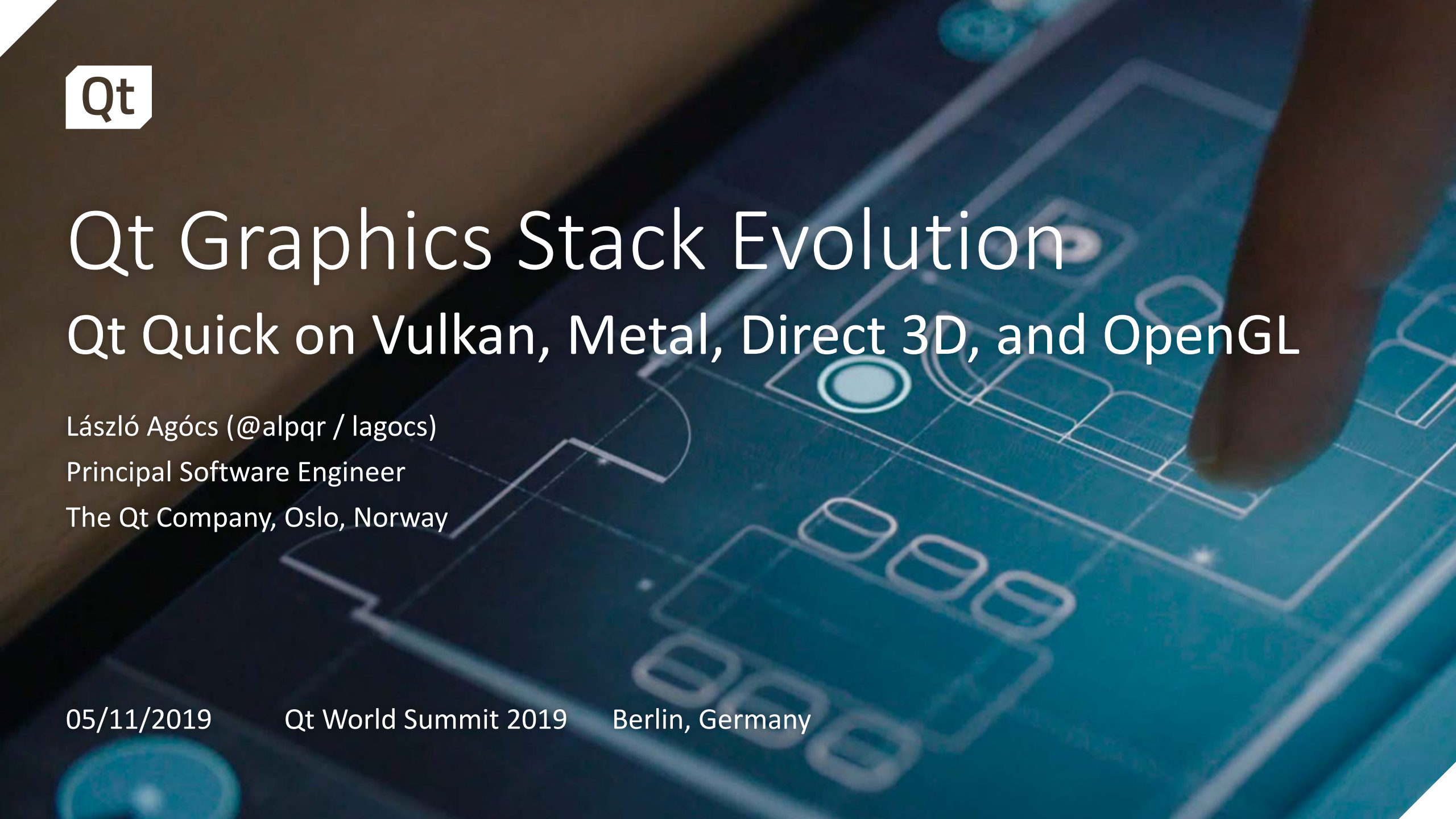


Qt Graphics Stack Evolution

Qt Quick on Vulkan, Metal, Direct 3D, and OpenGL

László Agócs (@alpqr / lagocs)
Principal Software Engineer
The Qt Company, Oslo, Norway

05/11/2019 Qt World Summit 2019 Berlin, Germany

Contents

- What?
- Why?
- How?

Qt

https://wiki.qt.io/New_Features_in_Qt_5.14

- Qt Quick
  - Added the first preview of the graphics API independent scenegraph renderer as an opt-in feature. This allows running qualifying Qt Quick applications on top of Vulkan, Metal, or Direct3D 11 instead of OpenGL. The supported platforms are currently Windows 10, Linux with X11 (xcb), macOS with MoltenVK, or Android 7.0+ for Vulkan, macOS for Metal, Windows 10 for D3D.

Demo

1. Qt Everywhere

2. Qt (Quick) application
+
external rendering

3. Shader pipeline renewal

4. Possible performance gains

+1. Enable new stuff

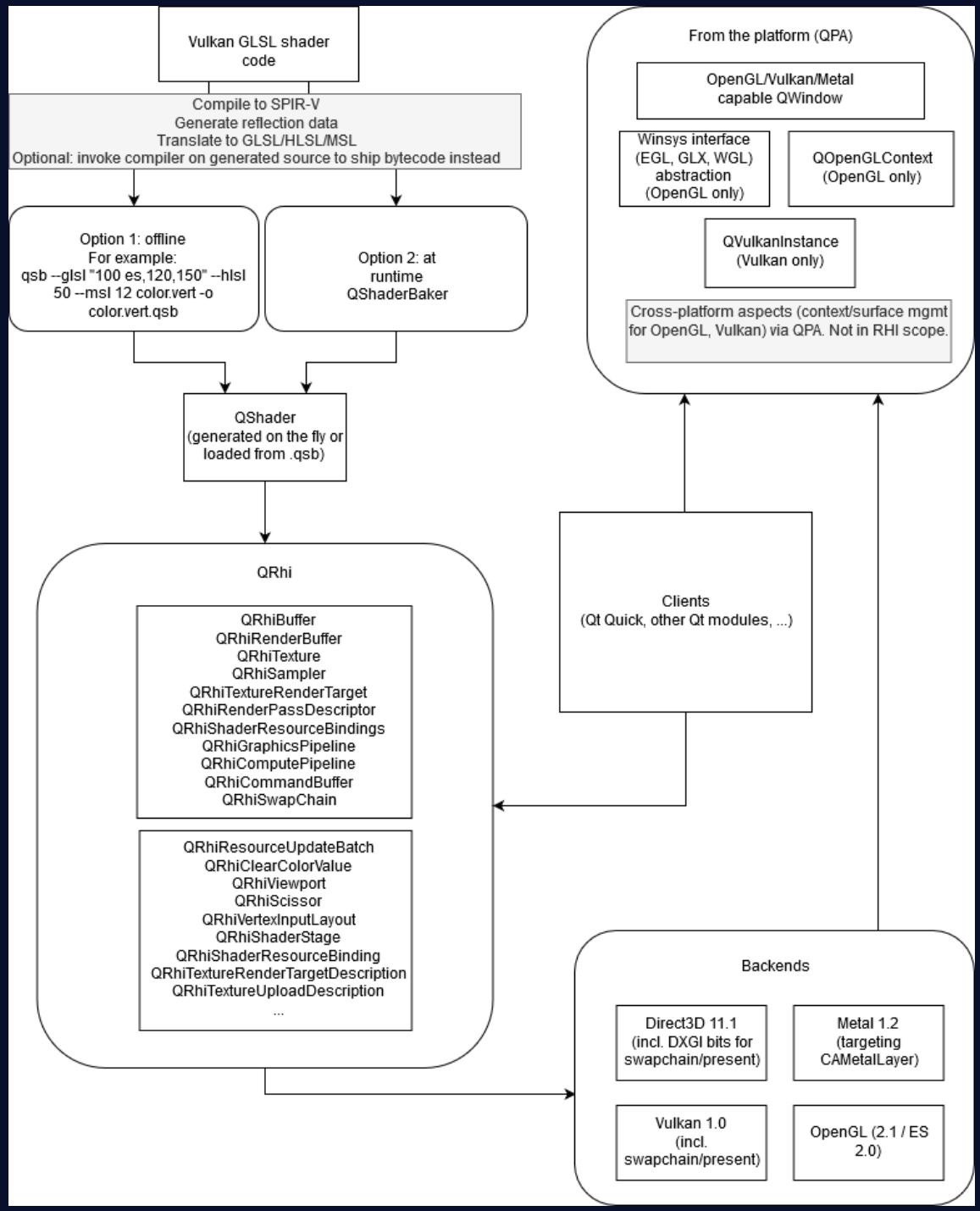
(for example, compute)

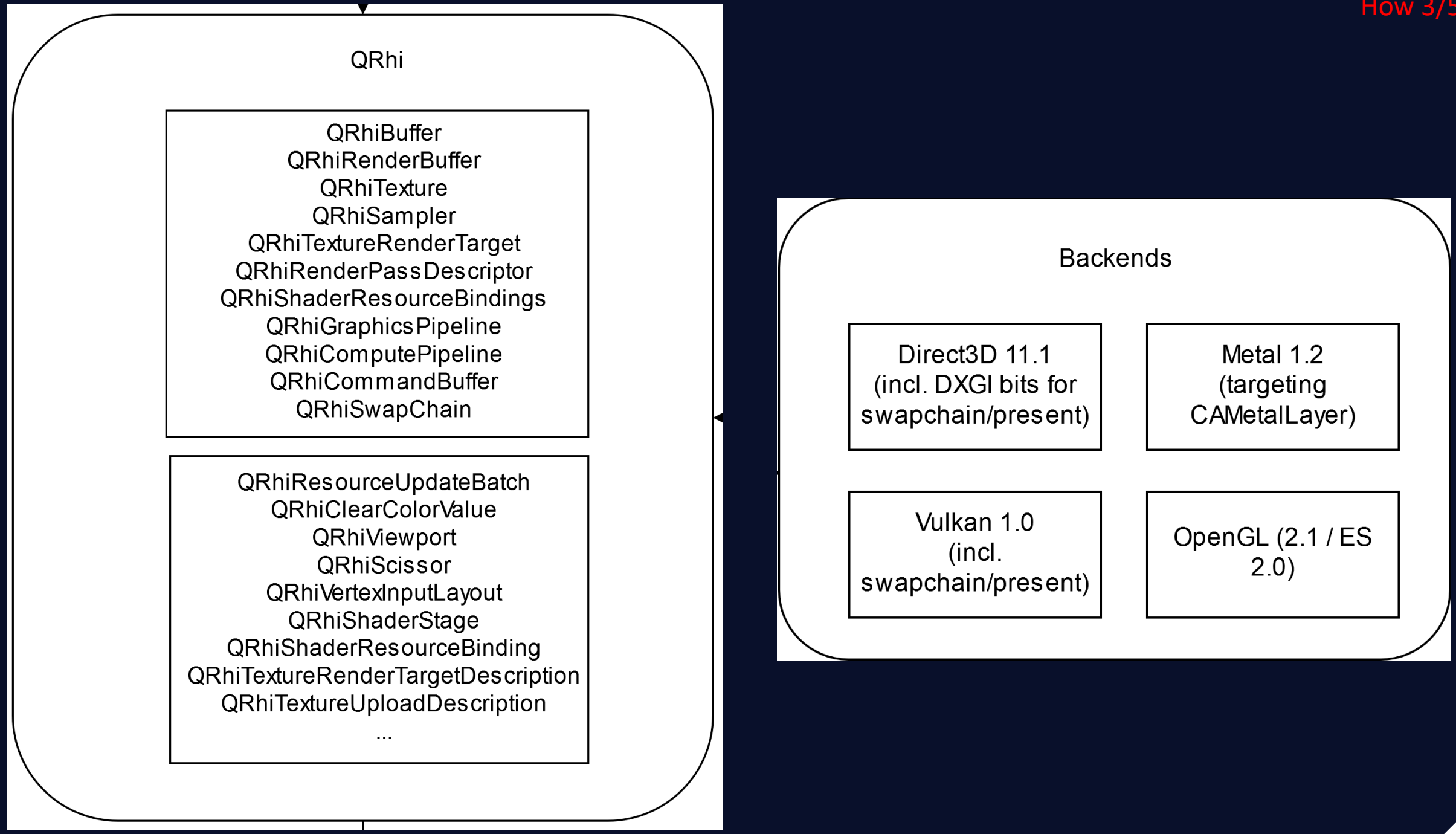
Components

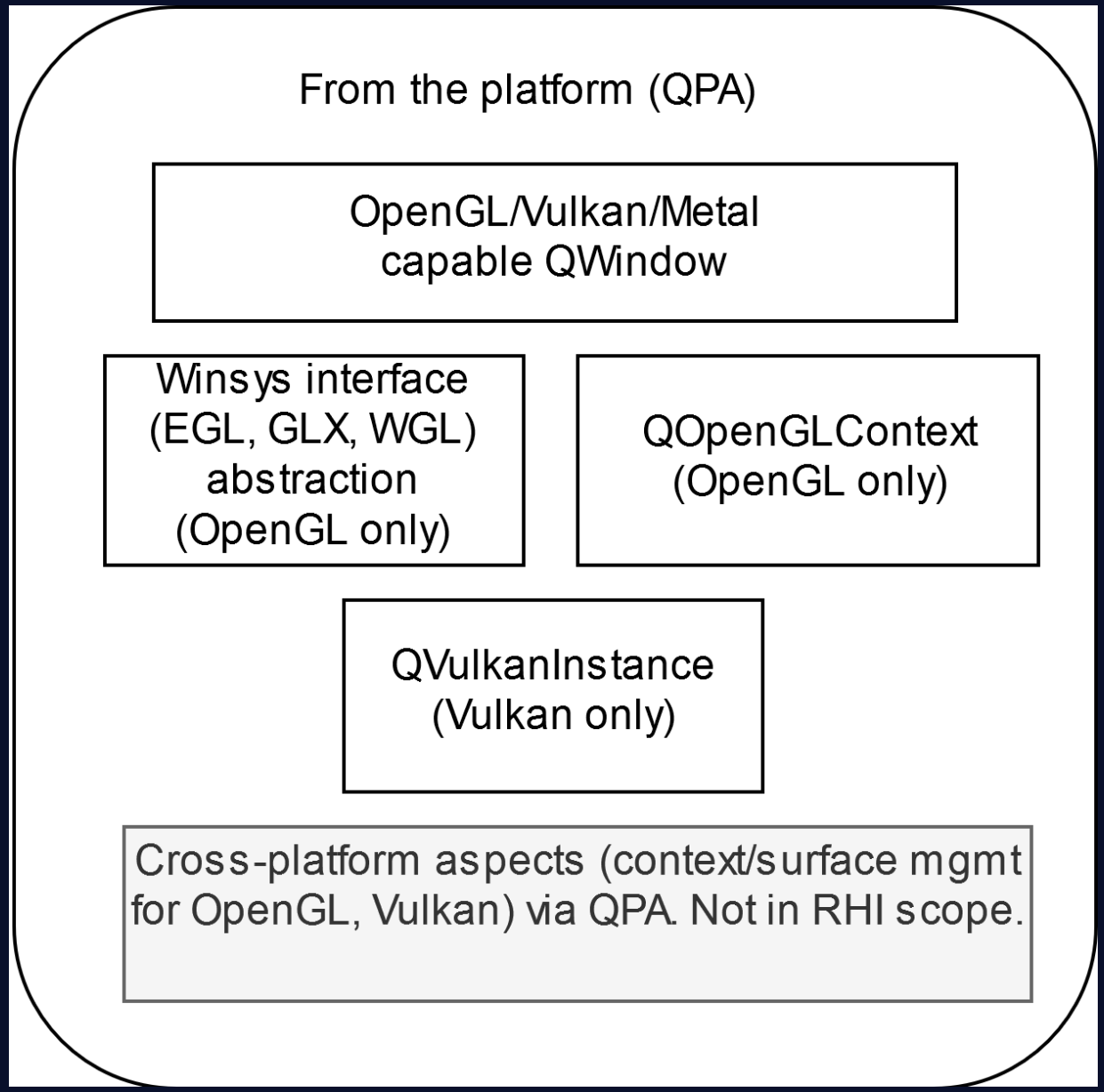
- Qt RHI
  - Part of the QtGui module
  - Private API
- Qt Shader Tools
  - qt-labs/qtshadertools
- Qt Quick port
  - Side-by-side

Components

- Qt RHI
  - Part of the QtGui module
  - Private API
- Qt Shader Tools
  - qt-labs/qtshadertools
- Qt Quick port
  - Side-by-side




QRhi
QRhiBuffer
QRhiRenderBuffer
QRhiTexture
QRhiSampler
QRhiTextureRenderTarget
QRhiRenderPassDescriptor
QRhiShaderResourceBindings
QRhiGraphicsPipeline
QRhiComputePipeline
QRhiCommandBuffer
QRhiSwapChain
QRhiResourceUpdateBatch
QRhiClearColorValue
QRhiViewport
QRhiScissor
QRhiVertexInputLayout
QRhiShaderStage
QRhiShaderResourceBinding
QRhiTextureRenderTargetDescription
QRhiTextureUploadDescription
...
Backends
Direct3D 11.1 (incl. DXGI bits for swapchain/present)
Metal 1.2 (targeting CAMetalLayer)
Vulkan 1.0 (incl. swapchain/present)
OpenGL (2.1 / ES 2.0)


From the platform (QPA)
OpenGL/Vulkan/Metal capable QWindow
Winsys interface (EGL, GLX, WGL) abstraction (OpenGL only)
QOpenGLContext (OpenGL only)
QVulkanInstance (Vulkan only)
Cross-platform aspects (context/surface mgmt for OpenGL, Vulkan) via QPA. Not in RHI scope.

Can we do a triangle in 50 lines?

```
// window is an exposed QWindow
// vertexShaderPack and fragmentShaderPack are the contents of .qsb files

    QRhiSwapChain *swapChain = rhi->newSwapChain();
    swapChain->setWindow(window);
    QRhiRenderPassDescriptor *rpDesc = swapChain->newCompatibleRenderPassDescriptor();
    swapChain->setRenderPassDescriptor(rpDesc);
    swapChain->buildOrResize();

    QRhiResourceUpdateBatch *updates = rhi->nextResourceUpdateBatch();
    static const float vertices[] = {
        -1.0f, -1.0f,
         1.0f, -1.0f,
         0.0f,  1.0f
    };
    QRhiBuffer *vbuf = rhi->newBuffer(QRhiBuffer::Immutable, QRhiBuffer::VertexBuffer, sizeof(vertices));
    vbuf->build();
    updates->uploadStaticBuffer(vbuf, vertices);
```

```
QRhiShaderResourceBindings *srb = rhi->newShaderResourceBindings();
srb->build();

QRhiGraphicsPipeline *pipeline = rhi->newGraphicsPipeline();
const QShader vs = QShader::fromSerialized(vertexShaderPack);
const QShader fs = QShader::fromSerialized(fragmentShaderPack);
pipeline->setShaderStages({ { QRhiShaderStage::Vertex, vs }, { QRhiShaderStage::Fragment, fs } });
QRhiVertexInputLayout inputLayout;
inputLayout.setBindings({ { 2 * sizeof(float) } });
inputLayout.setAttributes({ { 0, 0, QRhiVertexInputAttribute::Float2, 0 } });
pipeline->setVertexInputLayout(inputLayout);
pipeline->setShaderResourceBindings(srb);
pipeline->setRenderPassDescriptor(rpDesc);
pipeline->build();
```

```
rhi->beginFrame(swapChain);
QRhiCommandBuffer *cb = swapChain->currentFrameCommandBuffer();
QRhiRenderTarget *rt = swapChain->currentFrameRenderTarget();
const QSize outputSize = rt->pixelSize();

cb->beginPass(rt, Qt::green, { 1.0f, 0 }, updates);
cb->setGraphicsPipeline(pipeline);
cb->setViewport({ 0, 0, outputSize.width(), outputSize.height() });
QRhiCommandBuffer::VertexInput vbindings(vbuf, 0);
cb->setVertexInput(0, 1, &vbindings);
cb->draw(3);
cb->endPass();

rhi->endFrame(swapChain);
```

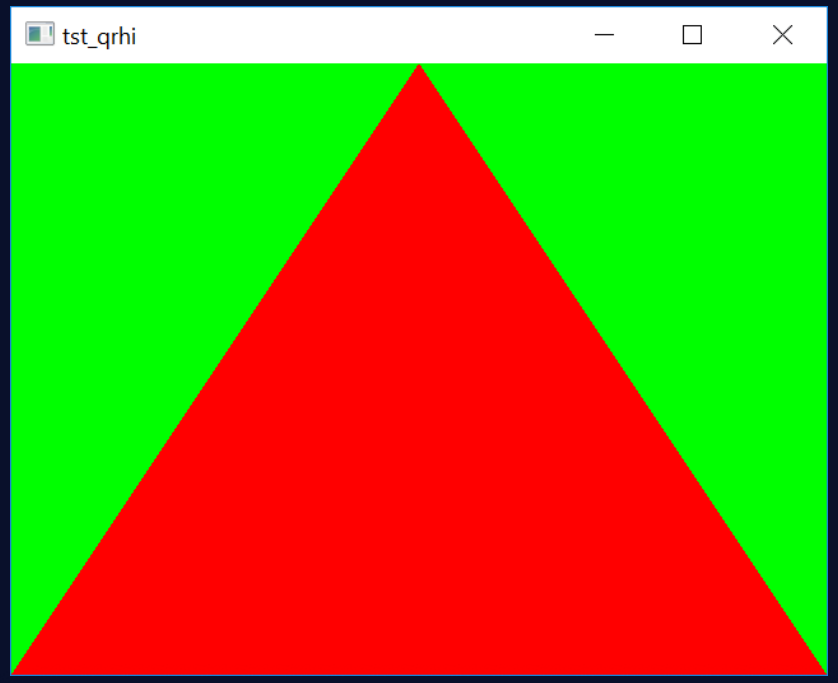
Assuming these vertex and fragment shaders…

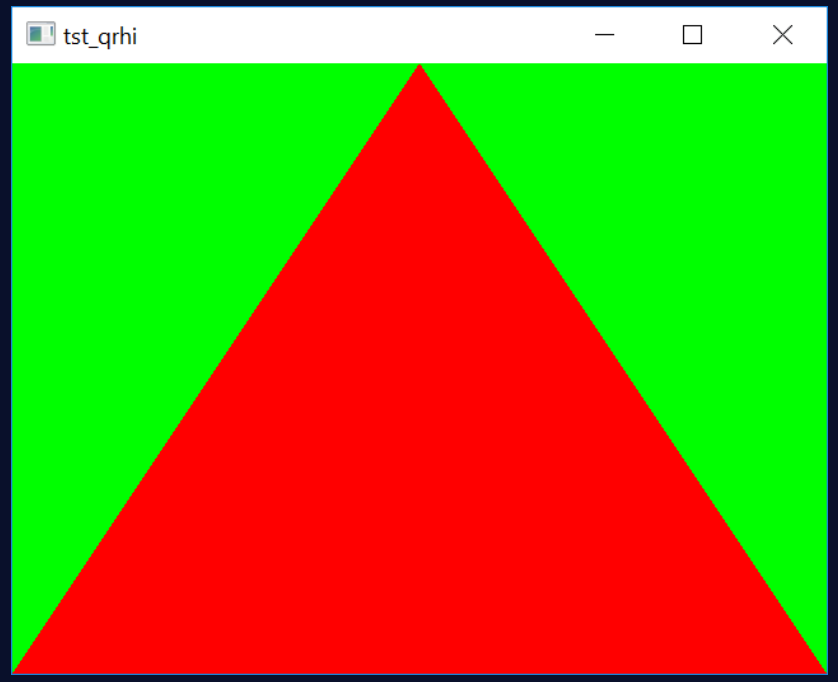
```
#version 440
layout(location = 0) in vec4 position;
out gl_PerVertex { vec4 gl_Position; };
void main()
{
    gl_Position = position;
}
```

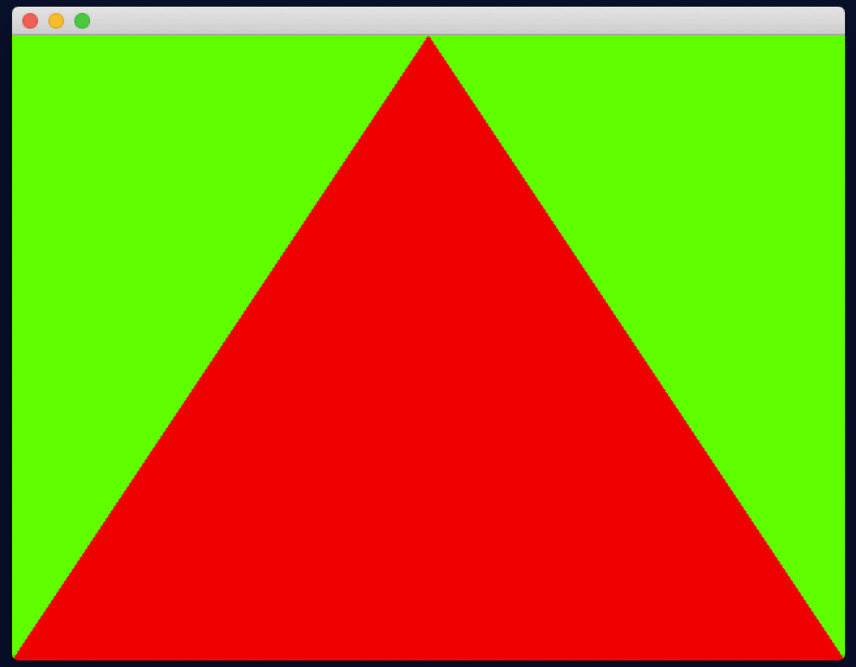
```
#version 440
layout(location = 0) out vec4 fragColor;
void main()
{
    fragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

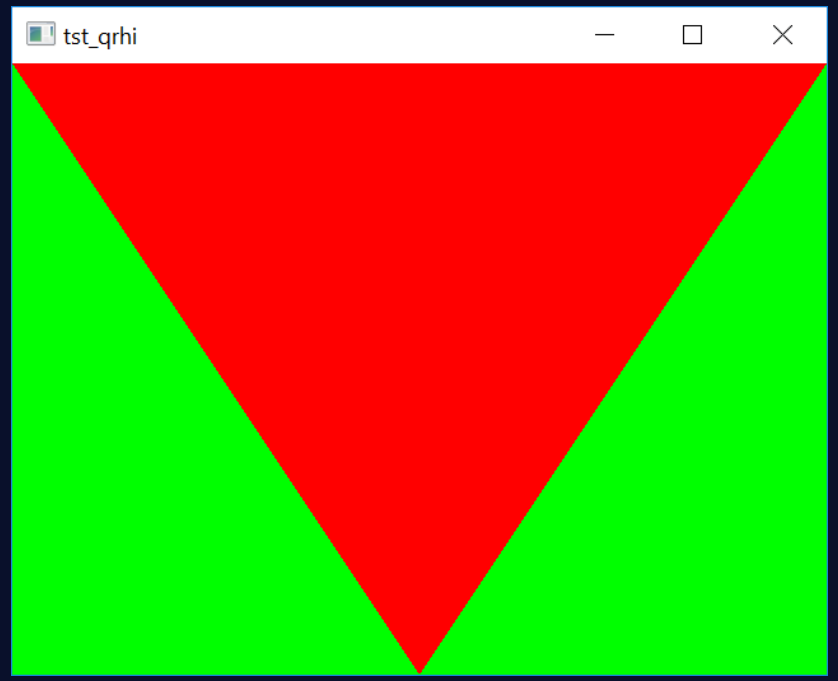
Qt

…this is what we get. Yay!



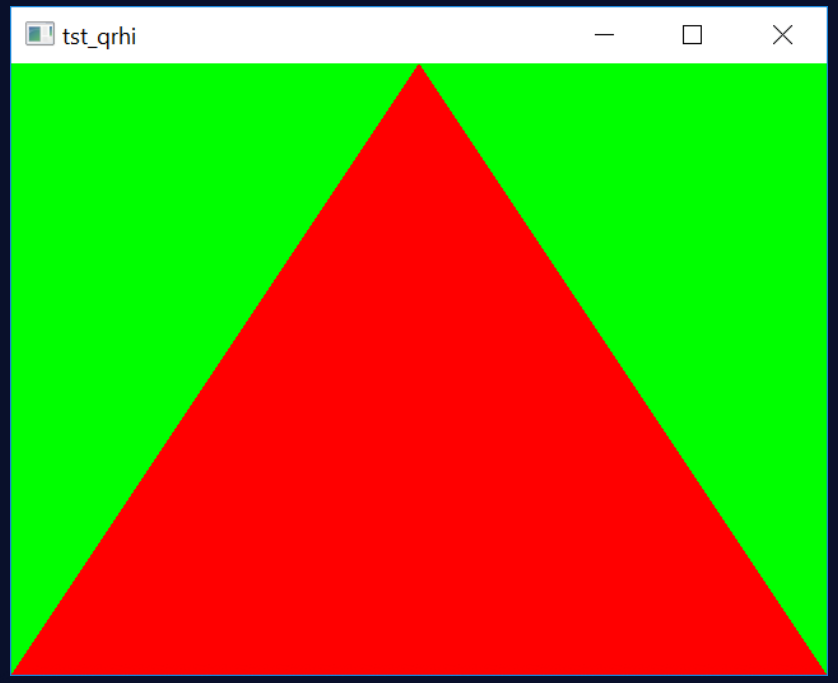


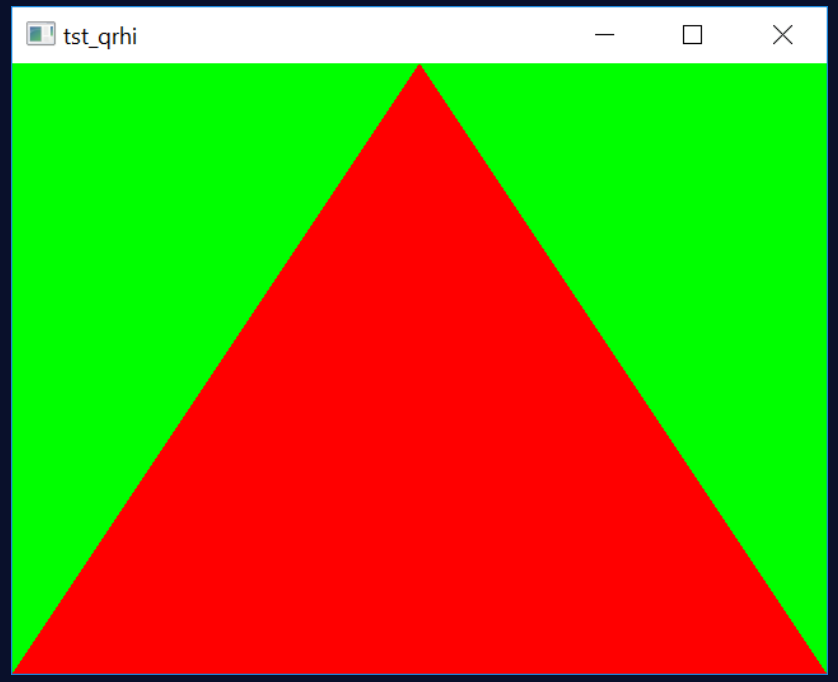


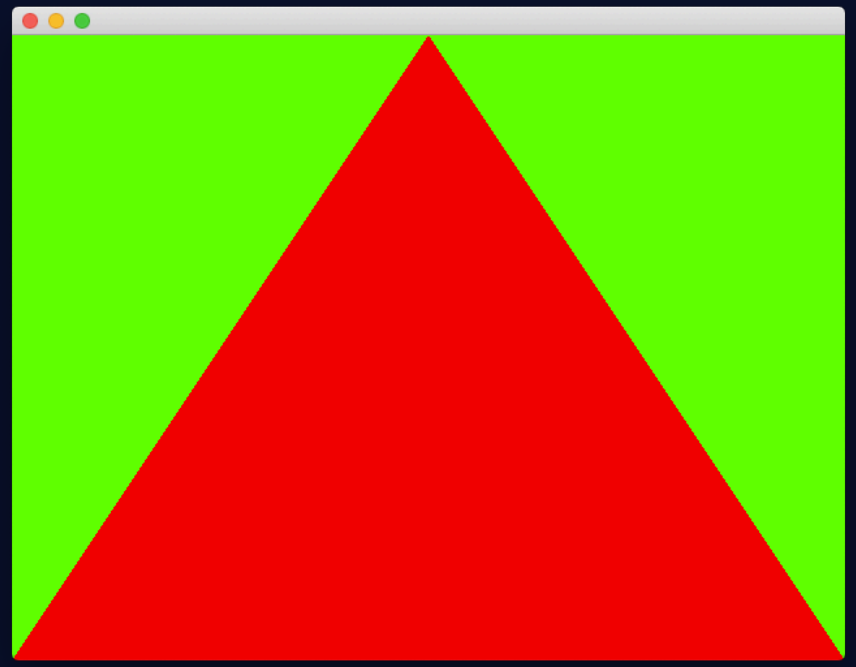


Qt

…this is what we get. Yay!







Yay?!

Qt

...this is what we get. Yay!

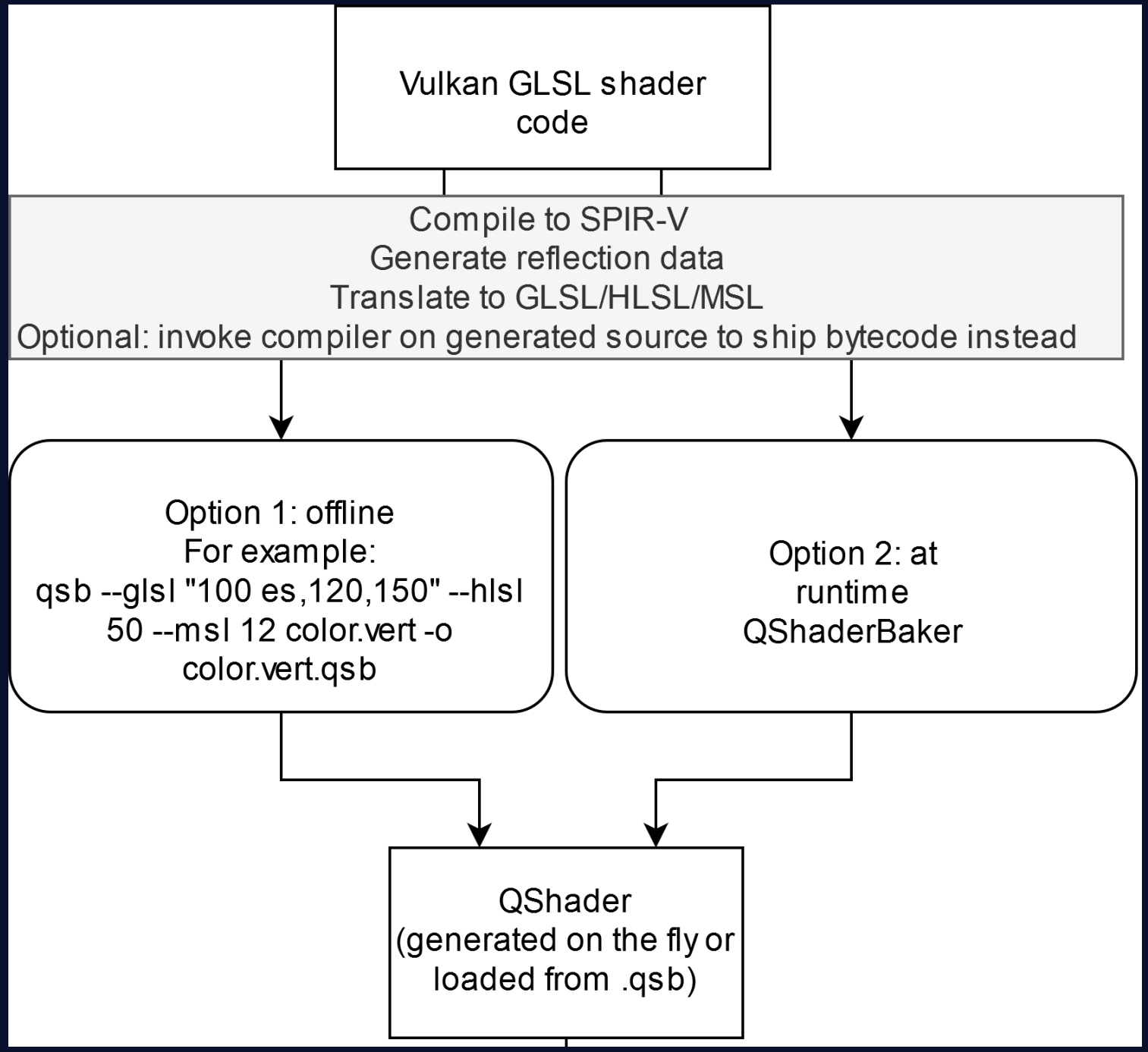
tst_qrhi

qrhi_p.h

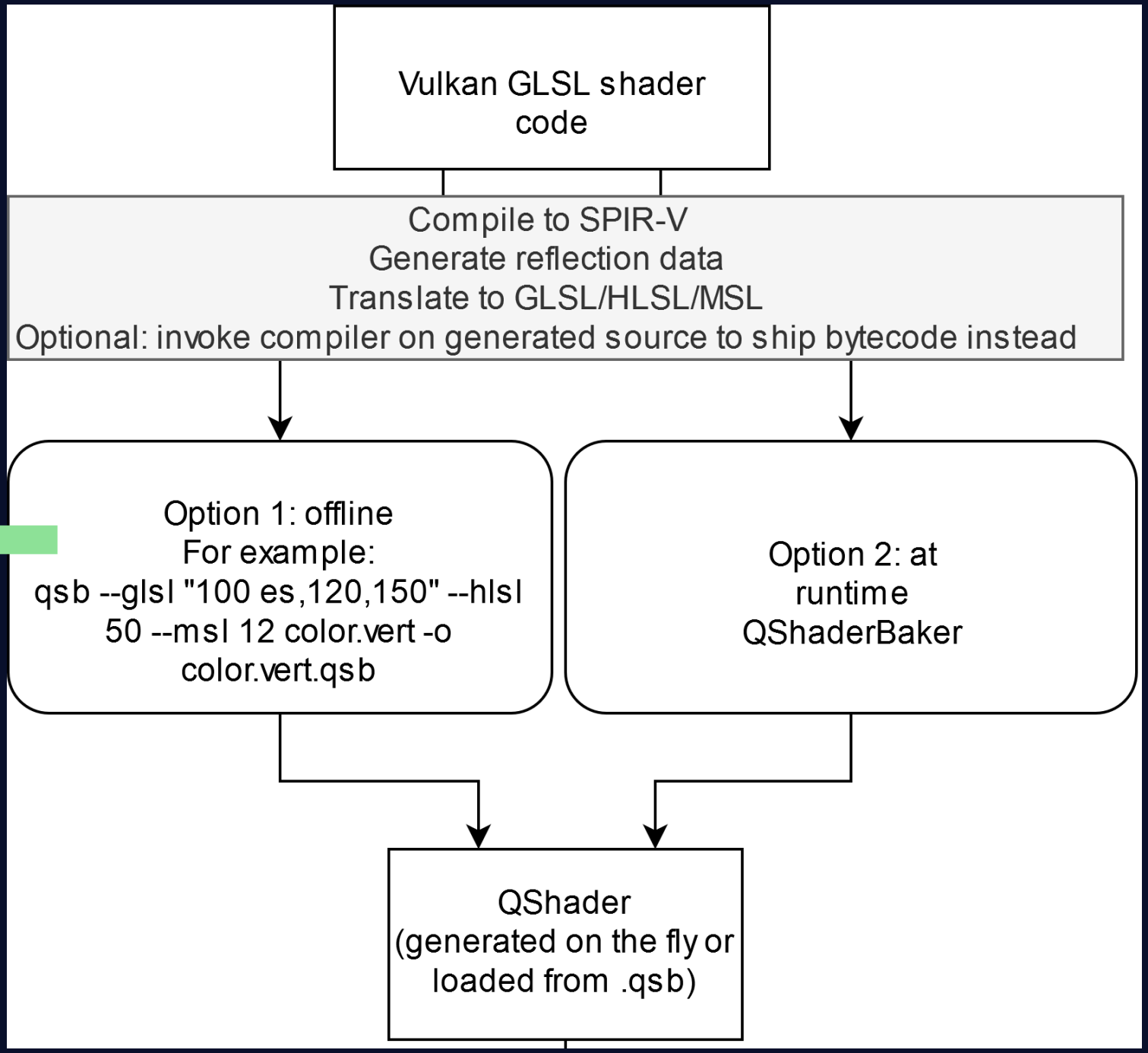
QRhi

```
1501    bool isRecordingFrame() const;
1502    int currentFrameSlot() const;
1503
1504    FrameOpResult beginOffscreenFrame(QRhiCommandBuffer **cb,
1505    FrameOpResult endOffscreenFrame(EndFrameFlags flags = End
1506
1507    QRhi::FrameOpResult finish();
1508
1509    QRhiResourceUpdateBatch *nextResourceUpdateBatch();
1510
1511    QVector<int> supportedSampleCounts() const;
1512
1513    int ubufAlignment() const;
1514    int ubufAligned(int v) const;
1515
1516    int mipLevelsForSize(const QSize &size) const;
1517    QSize sizeForMipLevel(int mipLevel, const QSize &baseLeve
1518
1519    bool isYUpInFramebuffer() const;
1520    bool isYUpInNDC() const;
1521    bool isClipDepthZeroToOne() const;
1522
1523    QMatrix4x4 clipSpaceCorrMatrix() const;
1524
1525    bool isTextureFormatSupported(QRhiTexture::Format format,
1526    bool isFeatureSupported(QRhi::Feature feature) const;
1527    int resourceLimit(ResourceLimit limit) const;
```

It’s fine.


Vulkan GLSL shader code
Compile to SPIR-V
Generate reflection data
Translate to GLSL/HLSL/MSL
Optional: invoke compiler on generated source to ship bytecode instead
Option 1: offline
For example:
qsb --glsl "100 es,120,150" --hlsl 50 --msl 12 color.vert -o color.vert.qsb
Option 2: at runtime
QShaderBaker
QShader
(generated on the fly or loaded from .qsb)

Qt



To be integrated with the build system in 6.x

```
#version 440

layout(location = 0) in vec2 sampleCoord;
layout(location = 0) out vec4 fragColor;

layout(binding = 1) uniform sampler2D _qt_texture;

layout(std140, binding = 0) uniform buf {
    mat4 matrix;
    vec4 color;
    vec2 textureScale;
    float dpr;
} ubuf;

void main()
{
    vec4 glyph = texture(_qt_texture, sampleCoord);
    fragColor = vec4(glyph.rgb * ubuf.color.a, glyph.a);
}
```

qsb --glsl "150,120,100 es" --hlsl 50 --msl 12
-o textmask.frag.qsb textmask.frag

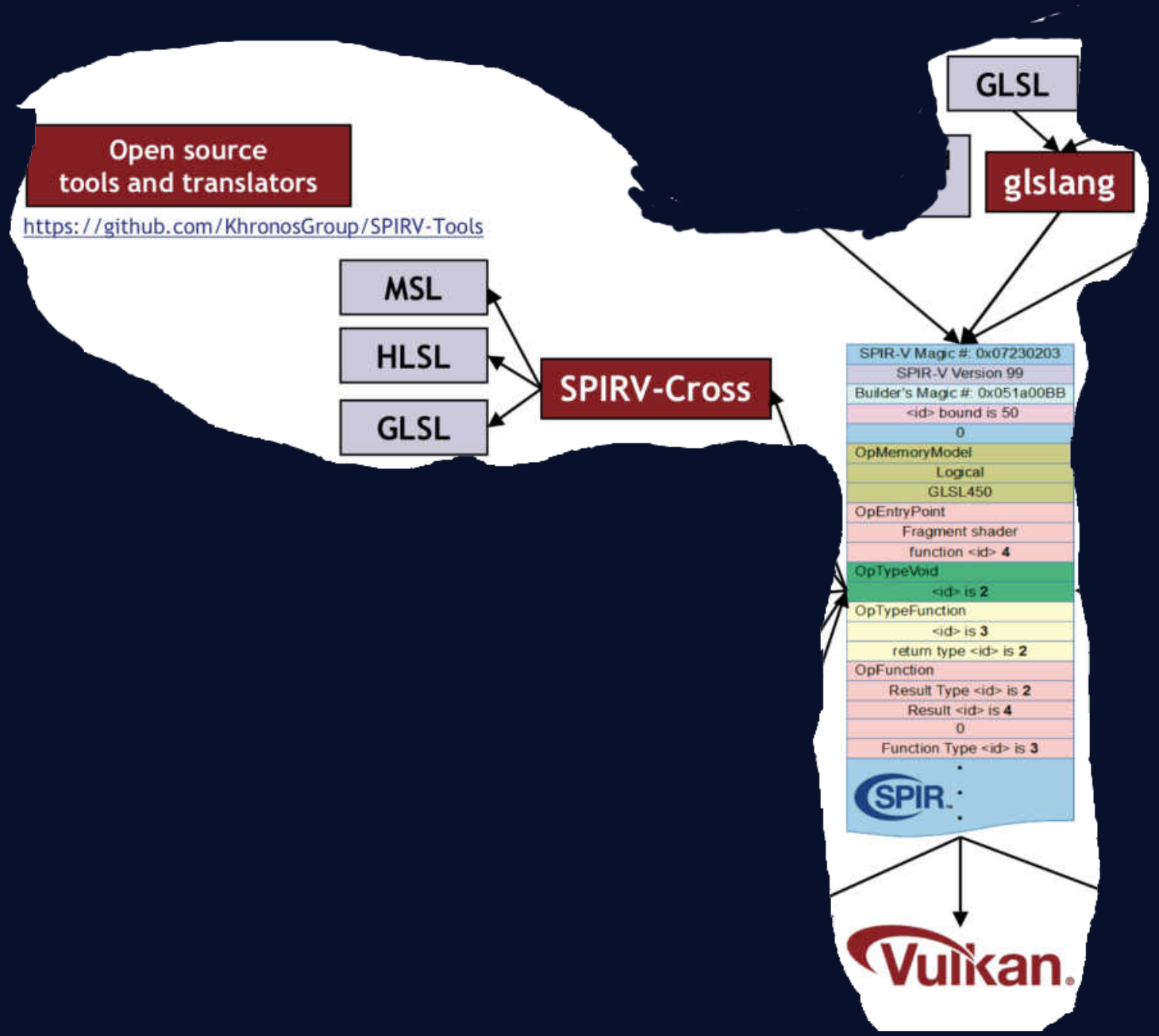
```
Stage: Fragment

Has 6 shaders: (unordered list)
  Shader 0: HLSL 50 [Standard]
  Shader 1: GLSL 150 [Standard]
  Shader 2: GLSL 100 es [Standard]
  Shader 3: GLSL 120 [Standard]
  Shader 4: SPIR-V 100 [Standard]
  Shader 5: MSL 12 [Standard]

Reflection info: {
    "combinedImageSamplers": [
        {
            "binding": 1,
            "name": "_qt_texture",
            "set": 0,
            "type": "sampler2D"
        }
    ],
    "inputs": [
        {
            "location": 0,
            "name": "sampleCoord",
            "type": "vec2"
        }
```

```
Shader 0: HLSL 50 [Standard]
Entry point: main
Contents:
cbuffer buf : register(b0)
{
    row_major float4x4 ubuf_matrix : packoffset(c0);
    float4 ubuf_color : packoffset(c4);
    float2 ubuf_textureScale : packoffset(c5);
    float ubuf_dpr : packoffset(c5.z);
};

Texture2D<float4> _qt_texture : register(t1);
SamplerState __qt_texture_sampler : register(s1);
```

```
    "uniformBlocks": [
        {
            "binding": 0,
            "blockName": "buf",
            "members": [
                {
                    "matrixStride": 16,
                    "name": "matrix",
                    "offset": 0,
                    "size": 64,
                    "type": "mat4"
                },
                {
                    "name": "color",
                    "offset": 64,
                    "size": 16,
                    "type": "vec4"
                },
                {
                    "name": "textureScale",
                    "offset": 80,
                    "size": 8,
                    "type": "vec2"
                },
                {
                    "name": "dpr",
                    "offset": 88,
                    "size": 4,
                    "type": "float"
                }
            ],
            "set": 0,
            "size": 92,
            "structName": "ubuf"
        }
```

...

Unbutchered, original image at https://www.khronos.org/assets/uploads/apis/2018-spir-api-ecosystem.jpg

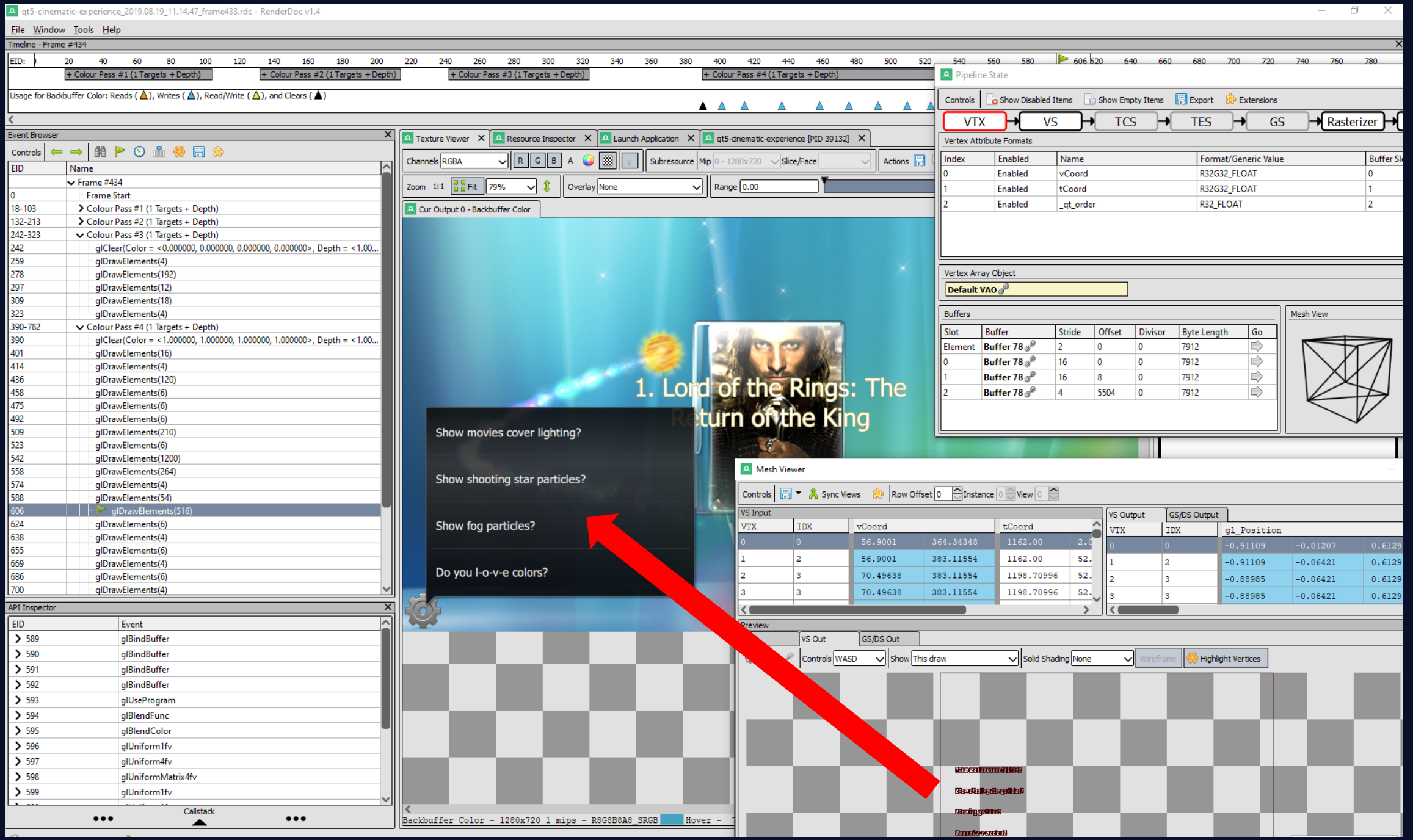


Qt Quick

- QQuickItem tree
  - described in QML
  - main thread
- QSGNode tree + material system
  - the “scene graph”
  - render thread
- OpenGL


```
[ ] RootNode 0x7fe48800f310  "" ) 1
 [ ] TransformNode( 0x7fe48800f390 identity "QQuickItem(QQuickRootItem:)"  ) 0
  [ ] TransformNode( 0x7fe488015fc0 identity "QQuickItem(QQuickItem:)"  ) 0
   [ ] TransformNode( 0x7fe488016220 identity "QQuickItem(MainView_QMLTYPE_8:)"  ) 0
    [ ] TransformNode( 0x7fe48808f450 identity "QQuickItem(QQuickItem:)"  ) 0
     [ ] TransformNode( 0x7fe48808f620 identity "QQuickItem(QQuickItem:)"  ) 0
      [ ] TransformNode( 0x7fe4881b70e0 identity "QQuickItem(QQuickImage_QML_2:)"  ) 0
       [ ] GeometryNode( 0x7fe4881b7c20 strip #V: 4 #I: 0 x1= 0 y1= 0 x2= 1280 y2= 720 materialtype= 0x7fe4c166ad32 ) "internalimage" 1000001 order 0
      [ ] TransformNode( 0x7fe4881b7290 identity "QQuickItem(QQuickParticleSystem:)"  ) 0
       [ ] TransformNode( 0x7fe4881b7480 identity "QQuickItem(QQuickImageParticle:)"  ) 0
       [ ] TransformNode( 0x7fe4881b7620 identity "QQuickItem(QQuickParticleEmitter:)"  ) 0
      [ ] TransformNode( 0x7fe48808f730 identity "QQuickItem(QQuickListView_QML_9:)"  ) 0
       [ ] TransformNode( 0x7fe4881ad830 translate 0 220 0 "QQuickItem(QQuickItem:)"  ) 0
        [ ] TransformNode( 0x7fe4881ad910 det= 0.444444 "QQuickItem(DelegateItem_QMLTYPE_0:)"  ) 0
         [ ] OpacityNode( 0x7fe4881ada10 opacity= 0.5 combined= 1  "" ) 1
          [ ] TransformNode( 0x7fe4881adf20 identity "QQuickItem(QQuickMouseArea:)"  ) 0
          [ ] TransformNode( 0x7fe4881ae0b0 translate 512 0 0 "QQuickItem(QQuickImage:)"  ) 0
           [ ] RootNode 0x7fe4881b0b90  "" ) 1
            [ ] GeometryNode( 0x7fe4881b1de0 strip #V: 4 #I: 0 x1= 0 y1= 0 x2= 256 y2= 256 materialtype= 0x7fe4c166ad32 ) "internalimage" 1000001 order 0
[ ] TransformNode( 0x7fe4881adaf0 det= 0.444444 "QQuickItem(DelegateItem_QMLTYPE_0:)"  ) 0
acity= 0.7  combined= 1  "" ) 1
 identity  QQuickItem(QQuickMouseArea:)"  ) 0
 translate 512 0 0 "QQuickItem(QQuickImage:)"  ) 0
) 1
0 strip #V: 4 #I: 0 x1= 0 y1= 0 x2= 256 y2= 256 materialtype= 0x7fe4c166ad32 ) "internalimage" 1000001 order 0
entity "QQuickItem(QQuickItem:)"  ) 0
et= 0.0948417 "QQuickItem(DelegateItem_QMLTYPE_0:)"  ) 0
acity= 0.825694 combined= 1  "" ) 1
 identity "QQuickItem(QQuickMouseArea:)"  ) 0
 translate 512 0 0 "QQuickItem(QQuickImage:)"  ) 0
) 1
0 strip #V: 4 #I: 0 x1= 0 y1= 0 x2= 256 y2= 256 materialtype= 0x7fe4c166ad32 ) "internalimage" 1000001 order 0
nslate 390 267 0 "QQuickItem(QQuickText_QML_10:)"  ) 0
entity "text"  ) 1
iangles #V: 140 #I: 210 x1= 20.0244 y1= 8.79277 x2= 479.55 y2= 107.761 materialtype= 0x7fe4c166b051 ) "glyphs" 1000003 order 0
nslate 480 528 0 "QQuickItem(QQuickImage_QML_11:)"  ) 0
```

```
Item {
    id: mainViewArea
    anchors.fill: parent

    Background {
        id: background
    }

    ListView {
        id: listView

        property real globalLightPosX: lightImage.x / root.width
        property real globalLightPosY: lightImage.y / root.height

        // Normal-mapped cover shared among delegates
        ShaderEffectSource {
            id: coverNmapSource
            sourceItem: Image { source: "images/cover_nmap.png" }
            hideSource: true
            visible: false
        }

        anchors.fill: parent
        spacing: -60
        model: moviesModel
        delegate: DelegateItem {
            name: model.name
```


qt5-cinematic-experience_2019.08.19_11.14.47_frame433.rdc - RenderDoc v1.4
File Window Tools Help
Timeline - Frame #434
+ Colour Pass #1 (1 Targets + Depth)
+ Colour Pass #2 (1 Targets + Depth)
+ Colour Pass #3 (1 Targets + Depth)
+ Colour Pass #4 (1 Targets + Depth)
Usage for Backbuffer Color: Reads ( ), Writes ( ), Read/Write ( ), and Clears ( )
Event Browser
Controls
EID Name
Frame #434
Frame Start
Colour Pass #1 (1 Targets + Depth)
Colour Pass #2 (1 Targets + Depth)
Colour Pass #3 (1 Targets + Depth)
glClear(Color = <0.000000, 0.000000, 0.000000, 0.000000>, Depth = <1.00...
glDrawElements(4)
glDrawElements(192)
glDrawElements(12)
glDrawElements(18)
glDrawElements(4)
Colour Pass #4 (1 Targets + Depth)
glClear(Color = <1.000000, 1.000000, 1.000000, 1.000000>, Depth = <1.00...
glDrawElements(16)
glDrawElements(4)
glDrawElements(120)
glDrawElements(6)
glDrawElements(6)
glDrawElements(6)
glDrawElements(210)
glDrawElements(6)
glDrawElements(1200)
glDrawElements(264)
glDrawElements(4)
glDrawElements(54)
glDrawElements(516)
glDrawElements(6)
glDrawElements(4)
glDrawElements(6)
glDrawElements(4)
glDrawElements(6)
glDrawElements(4)
API Inspector
EID Event
glBindBuffer
glBindBuffer
glBindBuffer
glBindBuffer
glUseProgram
glBlendFunc
glBlendColor
glUniform1fv
glUniform4fv
glUniformMatrix4fv
glUniform1fv
Callstack
Texture Viewer
Resource Inspector
Launch Application
qt5-cinematic-experience [PID 39132]
Channels RGBA
Subresource
Slice/Face
Actions
Zoom 1:1 Fit 79%
Overlay None
Range 0.00
Cur Output 0 - Backbuffer Color
1. Lord of the Rings: The Return of the King
Show movies cover lighting?
Show shooting star particles?
Show fog particles?
Do you l-o-v-e colors?
Backbuffer Color - 1280x720 1 mips - R8G8B8A8_SRGB
Hover -
Pipeline State
Controls
Show Disabled Items
Show Empty Items
Export
Extensions
VTX
VS
TCS
TES
GS
Rasterizer
Vertex Attribute Formats
Index Enabled Name Format/Generic Value Buffer Sl
0 Enabled vCoord R32G32_FLOAT 0
1 Enabled tCoord R32G32_FLOAT 1
2 Enabled _qt_order R32_FLOAT 2
Vertex Array Object
Default VAO
Buffers
Slot Buffer Stride Offset Divisor Byte Length Go
Element Buffer 78 2 0 0 7912
0 Buffer 78 16 0 0 7912
1 Buffer 78 16 8 0 7912
2 Buffer 78 4 5504 0 7912
Mesh View
Mesh Viewer
Controls
Sync Views
Row Offset 0
Instance 0
View 0
VS Input
VTX IDX vCoord tCoord
0 0 56.9001 364.34348 1162.00
1 2 56.9001 383.11554 1162.00
2 3 70.49638 383.11554 1198.70996
3 3 70.49638 383.11554 1198.70996
VS Output
GS/DS Output
VTX IDX gl_Position
0 0 -0.91109 -0.01207
1 2 -0.91109 -0.06421
2 3 -0.88985 -0.06421
3 3 -0.88985 -0.06421
Preview
VS Out
GS/DS Out
Controls WASD
Show This draw
Solid Shading None
Wireframe
Highlight Vertices

Qt Quick

- QQuickItem tree
  - described in QML
  - main thread
- QSGNode tree + material system
  - the “scene graph”
  - render thread
- Vulkan, Metal, Direct 3D, OpenGL

Materials

QSGMaterial creates a QSGMaterialShader
-> suitable for direct OpenGL

```
class Q_QUICK_EXPORT QSGMaterialShader
{
public:
    class Q_QUICK_EXPORT RenderState {
    public:
        inline bool isMatrixDirty() const { return m_dirty & DirtyMatrix; }
        ...
        QMatrix4x4 combinedMatrix() const;
        ...
    };

    virtual void updateState(const RenderState &state,
                             QSGMaterial *newMaterial,
                             QSGMaterial *oldMaterial);
    virtual char const *const *attributeNames() const = 0;

    virtual void initialize();
    virtual void activate();
    virtual void deactivate();

    void setShaderSourceFile(QOpenGLShader::ShaderType type, const QString &sourceFile);

    virtual const char *vertexShader() const;
    virtual const char *fragmentShader() const;
    virtual void compile();
```

QSGMaterial creates a QSGMaterialRhiShader
-> suitable for QRhi-based rendering

```
class Q_QUICK_EXPORT QSGMaterialRhiShader
{
public:
    class Q_QUICK_EXPORT RenderState {
    public:
        inline bool isMatrixDirty() const { ... }
        ...
        QMatrix4x4 combinedMatrix() const;
        QByteArray *uniformData();
        ...
    };
    enum Flag {
        UpdatesGraphicsPipelineState = 0x0001
    };
    enum Stage {
        VertexStage,
        FragmentStage,
    };

    virtual bool updateUniformData(RenderState &state,
                                   QSGMaterial *newMaterial, QSGMaterial *oldMaterial);

    virtual void updateSampledImage(RenderState &state, int binding, QSGTexture **texture,
                                    QSGMaterial *newMaterial, QSGMaterial *oldMaterial);

    virtual bool updateGraphicsPipelineState(RenderState &state, GraphicsPipelineState *ps,
                                             QSGMaterial *newMaterial,QSGMaterial *oldMaterial);

    void setFlag(Flags flags, bool on = true);

    // filename is for a file containing a serialized QShader.
    void setShaderFileName(Stage stage, const QString &filename);
```

```
class Q_QUICK_EXPORT QSGMaterialRhiShader
{
public:
    class Q_QUICK_EXPORT RenderState {
    public:
        inline bool isMatrixDirty() const { ... }
        ...
        QMatrix4x4 combinedMatrix() const;
        QByteArray *uniformData();
        ...
    };
    enum Flag {
        UpdatesGraphicsPipelineState = 0x0001
    };
```

A material (be it built-in or custom) provides data, and only data.
No graphics API access.

```
    virtual bool updateUniformData(RenderState &state,
                                   QSGMaterial *newMaterial, QSGMaterial *oldMaterial);

    virtual void updateSampledImage(RenderState &state, int binding, QSGTexture **texture,
                                    QSGMaterial *newMaterial, QSGMaterial *oldMaterial);

    virtual bool updateGraphicsPipelineState(RenderState &state, GraphicsPipelineState *ps,
                                             QSGMaterial *newMaterial,QSGMaterial *oldMaterial);

    void setFlag(Flags flags, bool on = true);

    // filename is for a file containing a serialized QShader.
    void setShaderFileName(Stage stage, const QString &filename);
```

```
class Q_QUICK_EXPORT QSGMaterialRhiShader
{
public:
    class Q_QUICK_E
    public:
        inline bool
        ...
        QMatrix4x4
        QByteArray
        ...
    };
    enum Flag {
        UpdatesGrap
    };

    virtual bool up

    virtual void up

    virtual bool up

    void setFlag(Fl

    // filename is
    void setShaderFileName(Stage stage, const QString &filename);
```

```
struct Q_QUICK_EXPORT GraphicsPipelineState {
    enum BlendFactor {
        Zero,
        One,
        SrcColor,
        ...
    };

    enum ColorMaskComponent {
        R = 1 << 0,
        G = 1 << 1,
        B = 1 << 2,
        A = 1 << 3
    };
    Q_DECLARE_FLAGS(ColorMask, ColorMaskComponent)

    enum CullMode {
        CullNone,
        CullFront,
        CullBack
    };

    bool blendEnable;
    BlendFactor srcColor;
    BlendFactor dstColor;
    ColorMask colorWrite;
    QColor blendConstant;
    CullMode cullMode;
```

A material (be , and only data.

ShaderEffect

```
ShaderEffect {
    id: shaderItem

    fragmentShader: "qrc:/qt-project.org/imports/QtQuick/Controls.2/Material/shaders/RectangularGlow.frag"

    x: (parent.width - width) / 2.0
    y: (parent.height - height) / 2.0
    width: parent.width + rootItem.glowRadius * 2 + cornerRadius * 2
    height: parent.height + rootItem.glowRadius * 2 + cornerRadius * 2

    function clampedCornerRadius() {
        var maxCornerRadius = Math.min(rootItem.width, rootItem.height) / 2 + rootItem.glowRadius;
        return Math.max(0, Math.min(rootItem.cornerRadius, maxCornerRadius))
    }

    property color color: rootItem.color
    property real inverseSpread: 1.0 - rootItem.spread
    property real relativeSizeX: ((inverseSpread * inverseSpread) * rootItem.glowRadius + cornerRadius * 2.
    property real relativeSizeY: relativeSizeX * (width / height)
    property real spread: rootItem.spread / 2.0
    property real cornerRadius: clampedCornerRadius()
}
```

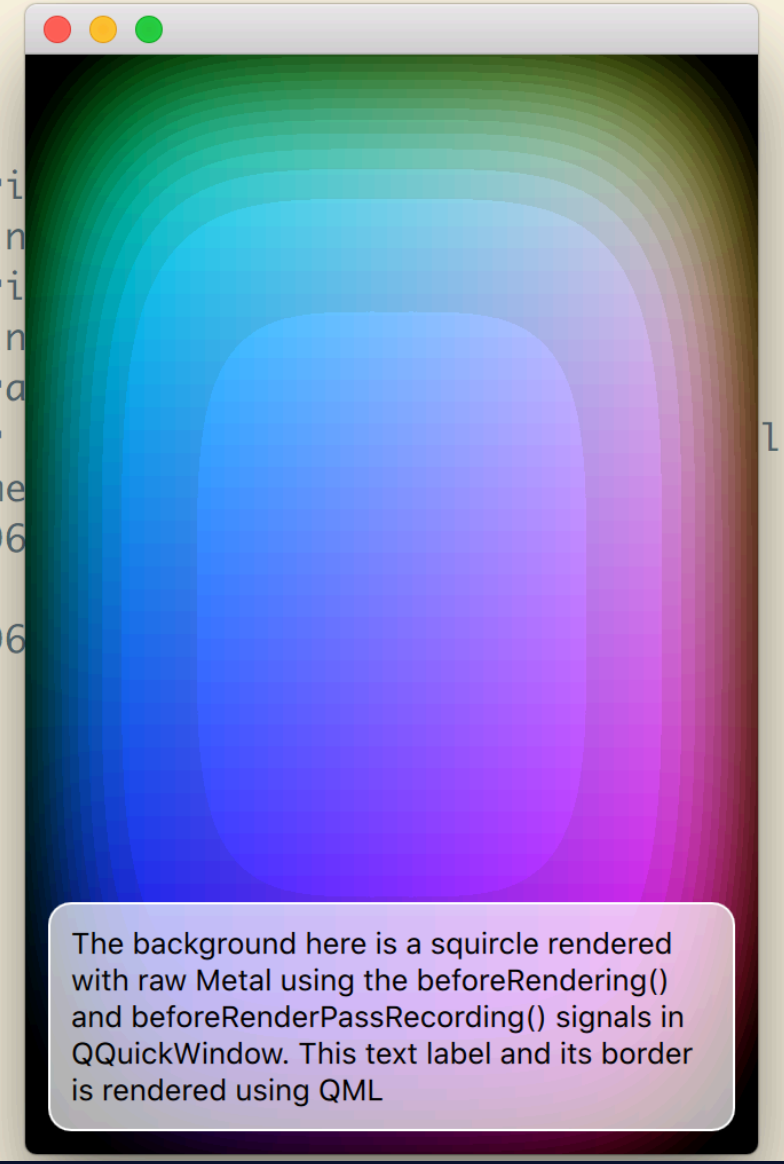
```
ShaderEffect {
    id: shaderItem

    fragmentShader: "qrc:/qt-project.org/imports/QtQuick/Controls.2/Material/shaders/RectangularGlow.frag"

    x: (parent
    y: (parent
    width: par
    height: pa

    function c
        var ma                                                    adius;
        returr
    }

    property c
    property r
    property r                                                    rnerRadius * 2.
    property real relativeSizeY: relativeSizeX * (width / height)
    property real spread: rootItem.spread / 2.0
    property real cornerRadius: clampedCornerRadius()
}
```

```
~/qtquickcontrols2_dev/src/imports/controls/material/shaders $ ls -lR
total 8
drwxr-xr-x  3 agocs  staff   96 Jul 10  2018 +glslcore
drwxr-xr-x  3 agocs  staff   96 Aug 17 15:31 +qsb
-rw-r--r--  1 agocs  staff  660 Jul 10  2018 RectangularGlow.frag

./+glslcore:
total 8
-rw-r--r--  1 agocs  staff  611 Jul 10  2018 RectangularGlow.frag

./+qsb:
total 8
-rw-r--r--  1 agocs  staff  2321 Aug 17 15:31 RectangularGlow.frag
```

Integrating custom Vulkan/Metal/D3D11/OpenGL rendering

1. metalunderqml

```
~/qtdeclarative_dev/examples/quick/scenegraph/metalunderqml $ ./metalunderqml.app/Contents/MacOS/metalunderqml
qt.scenegraph.general: Using QRhi with backend Metal
  graphics API debug/validation layers: 0
  QRhi profiling and debug markers: 0
qt.scenegraph.general: threaded render loop
qt.scenegraph.general: Using sg animation dri
qt.scenegraph.general: Animation Driver: usin
qt.scenegraph.general: Using sg animation dri
qt.scenegraph.general: Animation Driver: usin
qt.rhi.general: Metal device: Intel(R) HD Gra
qt.scenegraph.general: MSAA sample count for                    l requested = no.
qt.scenegraph.general: rhi texture atlas dime
qt.rhi.general: got CAMetalLayer, size 640x96
init
qt.rhi.general: got CAMetalLayer, size 640x96
```



void QQuickWindow::beforeRenderPassRecording() [signal]

This signal is emitted before the scenegraph starts recording commands for the main render pass. (Layers have their own passes and are fully recorded by the time this signal is emitted.) The render pass is already active on the command buffer when the signal is emitted.

This signal is applicable when using the RHI graphics abstraction with the scenegraph. It is emitted later than beforeRendering() and it guarantees that not just the frame, but also the recording of the scenegraph's main render pass is active. This allows inserting commands without having to generate an entire, separate render pass (which would typically clear the attached images). The native graphics objects can be queried via QSGRendererInterface.

When not running with the RHI (and using OpenGL directly), the signal is emitted after the renderer has cleared the render target. This makes it possible to create applications that function identically both with and without the RHI.

Note: Resource updates (uploads, copies) typically cannot be enqueued from within a render pass. Therefore, more complex user rendering will need to connect to both beforeRendering() and this signal.

Warning: This signal is emitted from the scene graph rendering thread. If your slot function needs to finish before execution continues, you must make sure that the connection is direct (see Qt::ConnectionType).

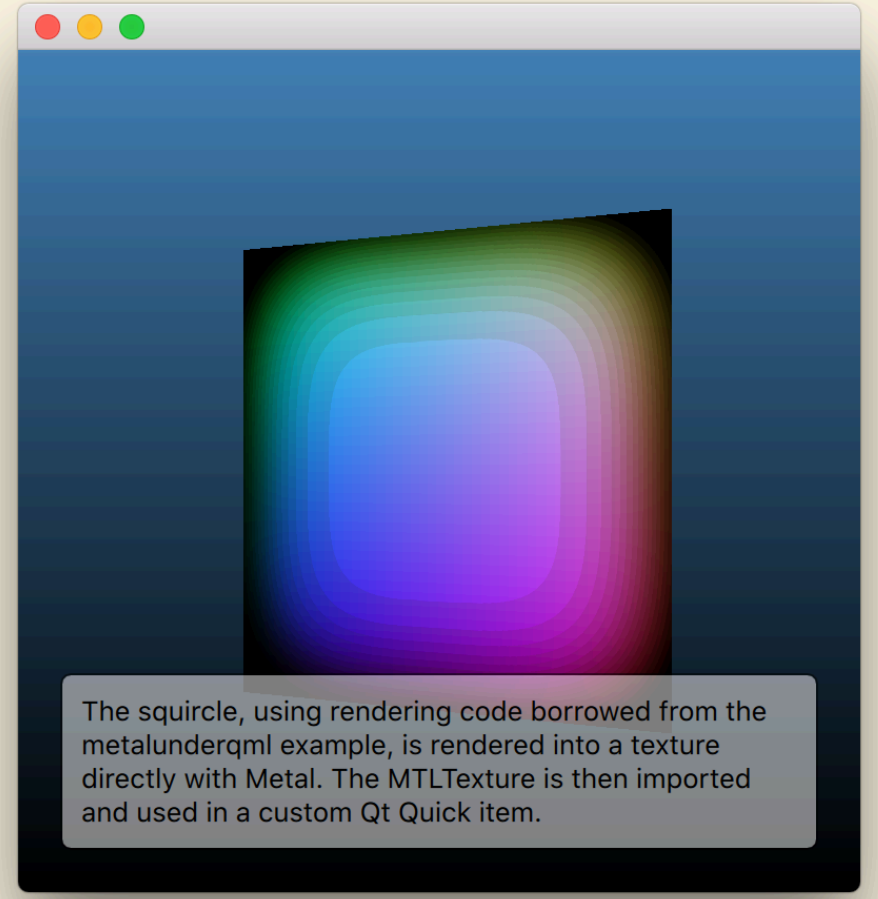
This function was introduced in Qt 5.14.

void QQuickWindow::beforeRendering() [signal]

This signal is emitted before the scene starts rendering.

1. metaltextureimpo

```
~/qtdeclarative_dev/examples/quick/scenegraph/metaltextureimport $ ./metaltextureimport.app/Contents/MacOS/metaltextureimport
renderer created
Got QSGTexture wrapper QSGPlainTexture(0x7f88c26587b0) for an MTLTexture of size QSize(800, 800)
resources initialized
```



```
QSGRendererInterface *rif = m_window->rendererInterface();
m_device = (id<MTLDevice>) rif->getResource(m_window, QSGRendererInterface::DeviceResource);
Q_ASSERT(m_device);

MTLTextureDescriptor *desc = [[MTLTextureDescriptor alloc] init];
desc.textureType = MTLTextureType2D;
desc.pixelFormat = MTLPixelFormatRGBA8Unorm;
desc.width = m_size.width();
desc.height = m_size.height();
desc.mipmapLevelCount = 1;
desc.resourceOptions = MTLResourceStorageModePrivate;
desc.storageMode = MTLStorageModePrivate;
desc.usage = MTLTextureUsageShaderRead | MTLTextureUsageRenderTarget;
m_texture = [m_device newTextureWithDescriptor: desc];
[desc release];

QSGTexture *wrapper = m_window->createTextureFromNativeObject(QQuickWindow::NativeObjectTexture,
                                                              &m_texture,
                                                              0,
                                                              m_size);

qDebug() << "Got QSGTexture wrapper" << wrapper << "for an MTLTexture of size" << m_size;
```

Key takeaways from porting the Qt Quick Renderer onto QRhi

a.k.a.

The Seven Commandments

Porting to QRhi 1

No intermixed copy operations, state changes, and draw calls.

Porting to QRhi 2

Must collect and submit resource updates before starting to record a render or compute pass.

Qt

```
void nextFrame()
{
    QRhiResourceUpdateBatch *resourceUpdates = m_rhi->nextResourceUpdateBatch();
    if (m_initialUpdates) {
        resourceUpdates->merge(m_initialUpdates);
        m_initialUpdates->release();
        m_initialUpdates = nullptr;
    }

    QMatrix4x4 triMvp = m_triBaseMvp;
    triMvp.rotate(m_triRot, 0, 1, 0);
    resourceUpdates->updateDynamicBuffer(m_triUbuf, 0, 64, triMvp.constData());

    cb->beginPass(rt, Qt::black, { 1.0f, 0 }, resourceUpdates);
```

QRhi-based code path

```
// depth test stays enabled but no need to write out depth from the
// transparent (back-to-front) pass
m_gstate.depthWrite = false;

QVarLengthArray<PreparedRenderBatch, 64> alphaRenderBatches;
if (Q_LIKELY(renderAlpha)) {
    for (int i = 0, ie = m_alphaBatches.size(); i != ie; ++i) {
        Batch *b = m_alphaBatches.at(i);
        PreparedRenderBatch renderBatch;
        bool ok;
        if (b->merged)
            ok = prepareRenderMergedBatch(b, &renderBatch);
        else if (b->isRenderNode)
            ok = prepareRhiRenderNode(b, &renderBatch);
        else
            ok = prepareRenderUnmergedBatch(b, &renderBatch);
        if (ok)
            alphaRenderBatches.append(renderBatch);
    }
}

if (m_visualizer->mode() != Visualizer::VisualizeNothing)
    m_visualizer->prepareVisualize();

QRhiCommandBuffer *cb = commandBuffer();
cb->beginPass(renderTarget(), m_pstate.clearColor, m_pstate.dsClear, m_resourceUpdates);
m_resourceUpdates = nullptr;

for (int i = 0, ie = opaqueRenderBatches.count(); i != ie; ++i) {
    PreparedRenderBatch *renderBatch = &opaqueRenderBatches[i];
    if (renderBatch->batch->merged)
        renderMergedBatch(renderBatch);
    else
        renderUnmergedBatch(renderBatch);
}
```

QRhi-based code path

```
// depth test stays enabled but no need to write out depth from the
// transparent (back-to-front) pass
m_gstate.depthWrite = false;

QVarLengthArray<PreparedRenderBatch, 64> alphaRenderBatches;
if (Q_LIKELY(renderAlpha)) {
    for (int i = 0, ie = m_alphaBatches.size(); i != ie; ++i) {
        Batch *b = m_alphaBatches.at(i);
        PreparedRenderBatch renderBatch;
        bool ok;
        if (b->merged)
            ok = prepareRenderMergedBatch(b, &renderBatch);
        else if (b->isRenderNode)
            ok = prepareRhiRenderNode(b, &renderBatch);
        else
            ok = prepareRenderUnmergedBatch(b, &renderBatch);
        if (ok)
            alphaRenderBatches.append(renderBatch);
    }
}

if (m_visualizer->mode() != Visualizer::VisualizeNothing)
    m_visualizer->prepareVisualize();

QRhiCommandBuffer *cb = commandBuffer();
cb->beginPass(renderTarget(), m_pstate.clearColor, m_pstate.dsClear, m_resourceUpdates);
m_resourceUpdates = nullptr;

for (int i = 0, ie = opaqueRenderBatches.count(); i != ie; ++i) {
    PreparedRenderBatch *renderBatch = &opaqueRenderBatches[i];
    if (renderBatch->batch->merged)
        renderMergedBatch(renderBatch);
    else
        renderUnmergedBatch(renderBatch);
}
```

Prepare vertex, index, uniform buffers, shader res.binding and pipeline state objects.

Materials provide **data**, and only data. (no messing with OpenGL)

QRhi-based code path

```
// depth test stays enabled but no need to write out depth from the
// transparent (back-to-front) pass
m_gstate.depthWrite = false;

QVarLengthArray<PreparedRenderBatch, 64> alphaRenderBatches;
if (Q_LIKELY(renderAlpha)) {
    for (int i = 0, ie = m_alphaBatches.size(); i != ie; ++i) {
        Batch *b = m_alphaBatches.at(i);
        PreparedRenderBatch renderBatch;
        bool ok;
        if (b->merged)
            ok = prepareRenderMergedBatch(b, &renderBatch);
        else if (b->isRenderNode)
            ok = prepareRhiRenderNode(b, &renderBatch);
        else
            ok = prepareRenderUnmergedBatch(b, &renderBatch);
        if (ok)
            alphaRenderBatches.append(renderBatch);
    }
}

if (m_visualizer->mode() != Visualizer::VisualizeNothing)
    m_visualizer->prepareVisualize();

QRhiCommandBuffer *cb = commandBuffer();
cb->beginPass(renderTarget(), m_pstate.clearColor, m_pstate.dsClear, m_resourceUpdates);
m_resourceUpdates = nullptr;

for (int i = 0, ie = opaqueRenderBatches.count(); i != ie; ++i) {
    PreparedRenderBatch *renderBatch = &opaqueRenderBatches[i];
    if (renderBatch->batch->merged)
        renderMergedBatch(renderBatch);
    else
        renderUnmergedBatch(renderBatch);
}
```

Start the renderpass, clear color/depth/stencil.

QRhi-based code path

```
// depth test stays enabled but no need to write out depth from the
// transparent (back-to-front) pass
m_gstate.depthWrite = false;

QVarLengthArray<PreparedRenderBatch, 64> alphaRenderBatches;
if (Q_LIKELY(renderAlpha)) {
    for (int i = 0, ie = m_alphaBatches.size(); i != ie; ++i) {
        Batch *b = m_alphaBatches.at(i);
        PreparedRenderBatch renderBatch;
        bool ok;
        if (b->merged)
            ok = prepareRenderMergedBatch(b, &renderBatch);
        else if (b->isRenderNode)
            ok = prepareRhiRenderNode(b, &renderBatch);
        else
            ok = prepareRenderUnmergedBatch(b, &renderBatch);
        if (ok)
            alphaRenderBatches.append(renderBatch);
    }
}

if (m_visualizer->mode() != Visualizer::VisualizeNothing)
    m_visualizer->prepareVisualize();

QRhiCommandBuffer *cb = commandBuffer();
cb->beginPass(renderTarget(), m_pstate.clearColor, m_pstate.dsClear, m_resourceUpdates);
m_resourceUpdates = nullptr;

for (int i = 0, ie = opaqueRenderBatches.count(); i != ie; ++i) {
    PreparedRenderBatch *renderBatch = &opaqueRenderBatches[i];
    if (renderBatch->batch->merged)
        renderMergedBatch(renderBatch);
    else
        renderUnmergedBatch(renderBatch);
}
```

Record draw calls

Porting to QRhi 3

Must create up front and then reuse resources like:

- pipeline state objects
- objects describing the shader resource bindings (which uniform buffers, textures, etc. are visible to which shader stages)

Porti

Must cre

› pipeli

› object

textur

m buffers,

```
3236  pool Renderer::ensurePipelineState(Element *e, const ShaderManager::Shader *sms) // RHI
3237  {
3238      // In unmerged batches the srbs in the elements are all compatible layout-wise.
3239      const GraphicsPipelineStateKey k { m_gstate, sms, renderPassDescriptor(), e->srb };
3240
3241      // See if there is an existing, matching pipeline state object.
3242      auto it = m_pipelines.constFind(k);
3243      if (it != m_pipelines.constEnd()) {
3244          e->ps = *it;
3245          return true;
3246      }
3247
3248      // Build a new one. This is potentially expensive.
3249      QRhiGraphicsPipeline *ps = m_rhi->newGraphicsPipeline();
3250      ps->setShaderStages(sms->programRhi.shaderStages);
3251      ps->setVertexInputLayout(sms->programRhi.inputLayout);
3252      ps->setShaderResourceBindings(e->srb);
3253      ps->setRenderPassDescriptor(renderPassDescriptor());
3254
3255      QRhiGraphicsPipeline::Flags flags = 0;
3256      if (needsBlendConstant(m_gstate.srcColor) || needsBlendConstant(m_gstate.dstColor))
3257          flags |= QRhiGraphicsPipeline::UsesBlendConstants;
3258      if (m_gstate.usesScissor)
3259          flags |= QRhiGraphicsPipeline::UsesScissor;
3260      if (m_gstate.stencilTest)
3261          flags |= QRhiGraphicsPipeline::UsesStencilRef;
3262
3263      ps->setFlags(flags);
3264      ps->setTopology(qsg_topology(m_gstate.drawMode));
3265      ps->setCullMode(m_gstate.cullMode);
3266
3267      QRhiGraphicsPipeline::TargetBlend blend;
3268      blend.colorWrite = m_gstate.colorWrite;
3269      blend.enable = m_gstate.blending;
3270      blend.srcColor = m_gstate.srcColor;
```

Porting to QRhi 4

Must declare up front certain usages of resources like textures.

Qt

Po

Mus

qrhi_p.h | QRhiTexture

```
class Q_GUI_EXPORT QRhiTexture : public QRhiResource
{
public:
    enum Flag {
        RenderTarget = 1 << 0,
        CubeMap = 1 << 2,
        MipMapped = 1 << 3,
        sRGB = 1 << 4,
        UsedAsTransferSource = 1 << 5,
        UsedWithGenerateMips = 1 << 6,
        UsedWithLoadStore = 1 << 7
    };
    Q_DECLARE_FLAGS(Flags, Flag)
```

Porting to QRhi 5

Uniform buffers. No individual uniforms.

Porting to QRhi 5

Uniform buffers. No individual uniforms.

From QRhi API perspective. Backends may not actually operate with native uniform buffers.

Porting to QRhi 5

Uniform buffers. No individual uniforms.

From QRhi API perspective. Backends may not actually operate with native uniform buffers. (hello OpenGL ES 2.0 and broken ES 3.x implementations!)

Porting to QRhi 6

No target buffer (color/depth/stencil) clears at arbitrary times.

Qt

Port

No targ

```
if (!(clipType & ClipState::StencilClip)) {
    if (!m_clipProgram.isLinked()) {
        QSGShaderSourceBuilder::initializeProgramFromFiles(
            &m_clipProgram,
            QStringLiteral(":/qt-project.org/scenegraph/shaders/stencilclip.vert"),
            QStringLiteral(":/qt-project.org/scenegraph/shaders/stencilclip.frag"));
        m_clipProgram.bindAttributeLocation("vCoord", 0);
        m_clipProgram.link();
        m_clipMatrixId = m_clipProgram.uniformLocation("matrix");
    }

    glClearStencil(0);
    glClear(GL_STENCIL_BUFFER_BIT);
    glEnable(GL_STENCIL_TEST);
    glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
    glDepthMask(GL_FALSE);

    m_clipProgram.bind();
    m_clipProgram.enableAttributeArray(0);

    clipType |= ClipState::StencilClip;
}

glStencilFunc(GL_EQUAL, m_currentStencilValue, 0xff); // stencil test, ref, test mask
glStencilOp(GL_KEEP, GL_KEEP, GL_INCR); // stencil fail, z fail, z pass

const QSGGeometry *g = clip->geometry();
Q_ASSERT(g->attributeCount() > 0);
const QSGGeometry::Attribute *a = g->attributes();
```

Qt

Port

No targ

```
if (!(clipType & ClipState::StencilClip)) {
    if (!m_clipProgram.isLinked()) {
        QSGShaderSourceBuilder::initializeProgramFromFiles(
            &m_clipProgram,
            QStringLiteral(":/qt-project.org/scenegraph/shaders/stencilclip.vert"),
            QStringLiteral(":/qt-project.org/scenegraph/shaders/stencilclip.frag"));
        m_clipProgram.bindAttributeLocation("vCoord", 0);
        m_clipProgram.link();
        m_clipMatrixId = m_clipProgram.uniformLocation("matrix");
    }

    glClearStencil(0);
    glClear(GL_STENCIL_BUFFER_BIT);
    glEnable(GL_STENCIL_TEST);
    glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
    glDepthMask(GL_FALSE);

    m_clipProgram.bind();
    m_clipProgram.enableAttributeArray(0);

    clipType |= ClipState::StencilClip;
}

glStencilFunc(GL_EQUAL, m_currentStencilValue, 0xff); // stencil test, ref, test mask
glStencilOp(GL_KEEP, GL_KEEP, GL_INCR); // stencil fail, z fail, z pass

const QSGGeometry *g = clip->geometry();
Q_ASSERT(g->attributeCount() > 0);
const QSGGeometry::Attribute *a = g->attributes();
```

!#$&^#@(*

Porting to QRhi 7

Two buffers are good, one is better.

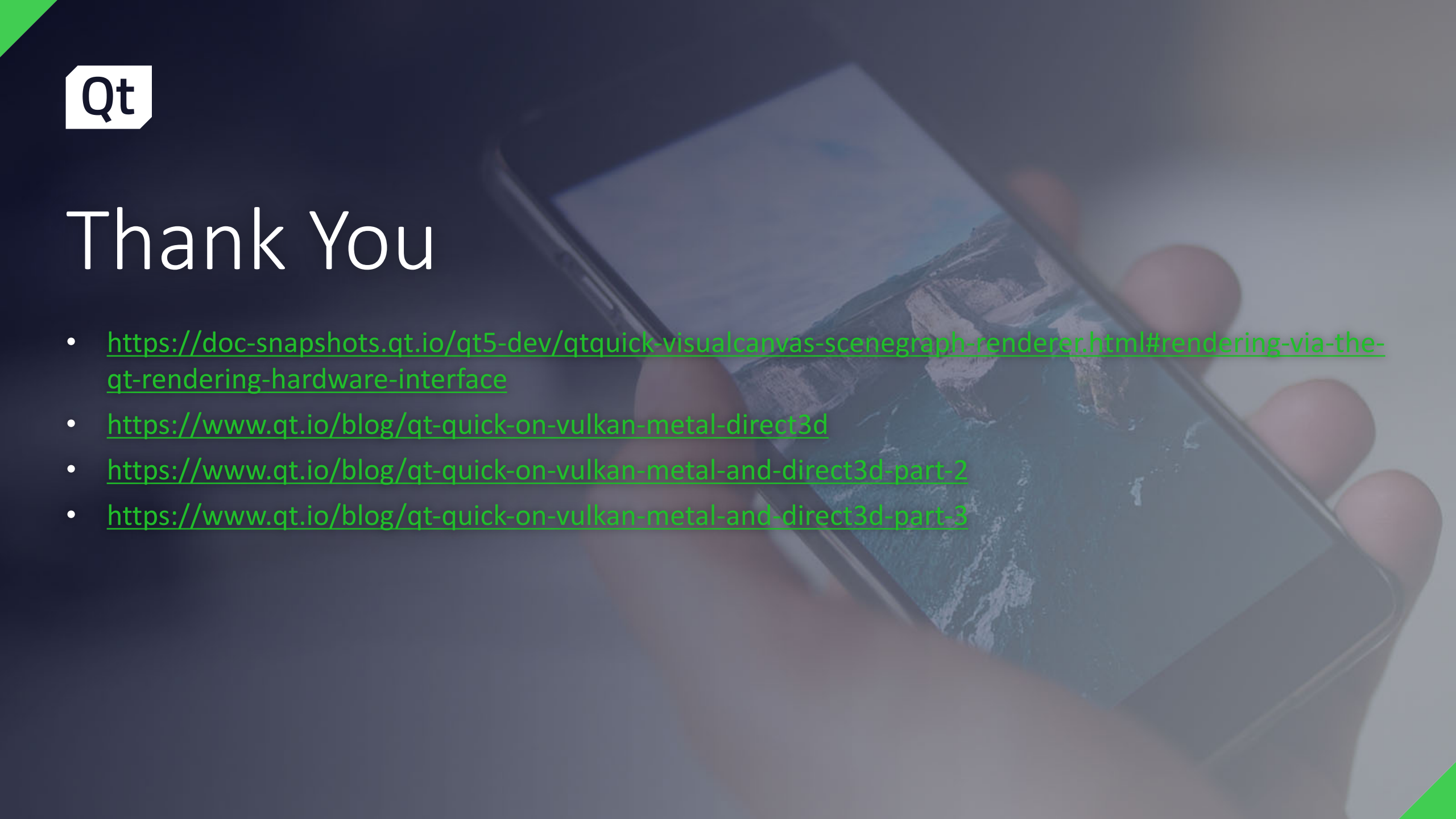
```
drawCall.vbufOffset = aligned(vOffset, 4);
const int vertexByteSize = g->sizeOfVertex() * g->vertexCount();
vOffset = drawCall.vbufOffset + vertexByteSize;

int indexByteSize = 0;
if (g->indexCount()) {
    drawCall.ibufOffset = aligned(iOffset, 4);
    indexByteSize = g->sizeOfIndex() * g->indexCount();
    iOffset = drawCall.ibufOffset + indexByteSize;
}

drawCall.ubufOffset = aligned(uOffset, m_ubufAlignment);
uOffset = drawCall.ubufOffset + StencilClipUbufSize;

QMatrix4x4 matrixYUpNDC = m_current_projection_matrix;
if (clip->matrix())
    matrixYUpNDC *= *clip->matrix();

m_resourceUpdates->updateDynamicBuffer(batch->stencilClipState.ubuf,
                                       drawCall.ubufOffset, 64, matrixYUpNDC.constData());
m_resourceUpdates->updateDynamicBuffer(batch->stencilClipState.vbuf,
                                       drawCall.vbufOffset, vertexByteSize, g->vertexData());
if (indexByteSize)
    m_resourceUpdates->updateDynamicBuffer(batch->stencilClipState.ibuf,
                                           drawCall.ibufOffset, indexByteSize, g->indexData());
```

Consequence

Typically, a **render** step becomes **prepare** + **render**.

Consequence

- Prepare: Gather all data (geometry, pipeline states, shader res.) needed for the current frame, enqueue buffer (vertex, index, uniform) and texture resource updates.
- Render: start the pass, record binding ia/shader/pipeline stuff, record draw call, change bindings if needed, record draw call, …, end pass.
- Submit and present.



Thank You

- https://doc-snapshots.qt.io/qt5-dev/qtquick-visualcanvas-scenegraph-renderer.html#rendering-via-the-qt-rendering-hardware-interface
- https://www.qt.io/blog/qt-quick-on-vulkan-metal-direct3d
- https://www.qt.io/blog/qt-quick-on-vulkan-metal-and-direct3d-part-2
- https://www.qt.io/blog/qt-quick-on-vulkan-metal-and-direct3d-part-3