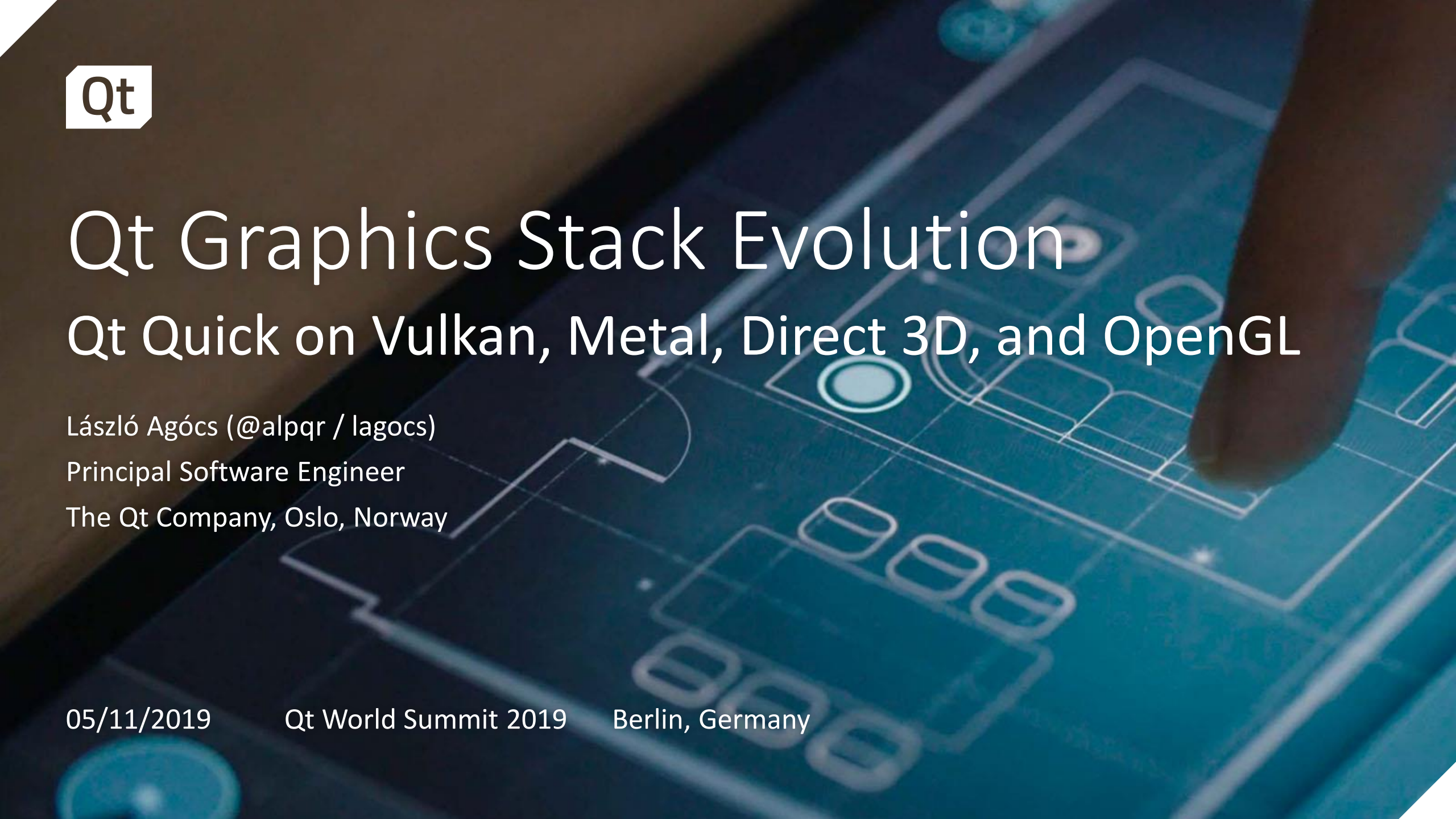


Qt Graphics Stack Evolution

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

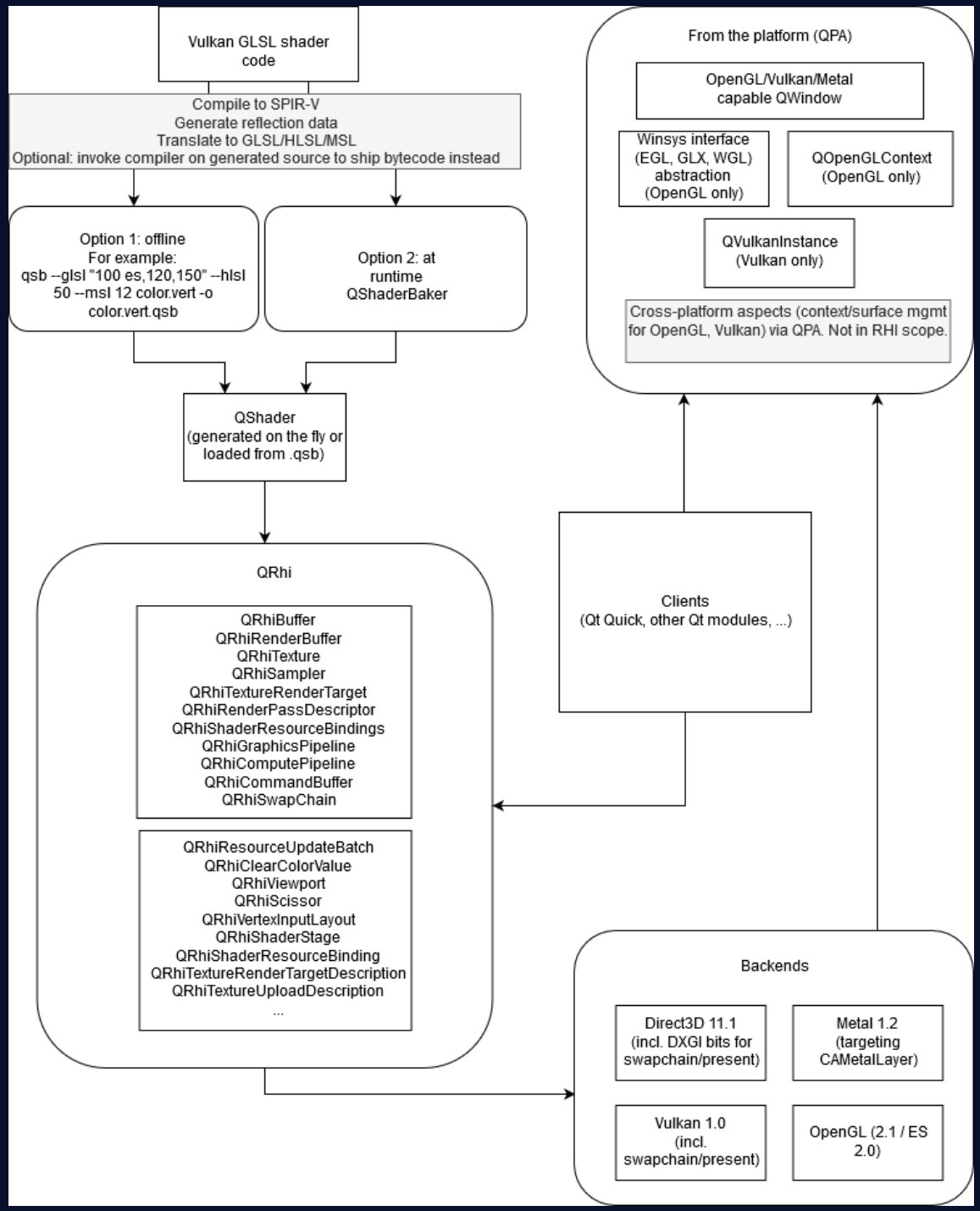
(for example, compute)

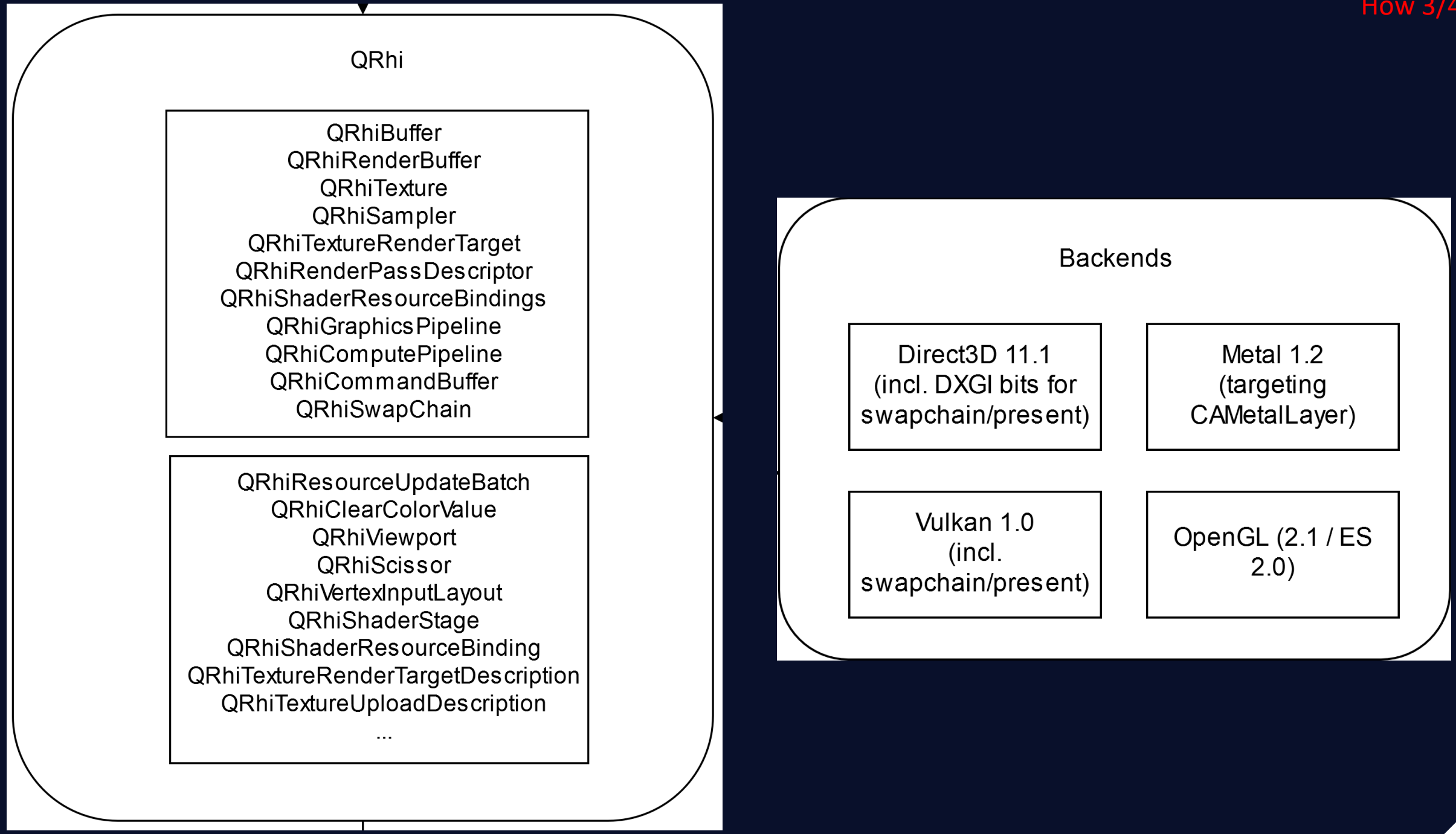
Components

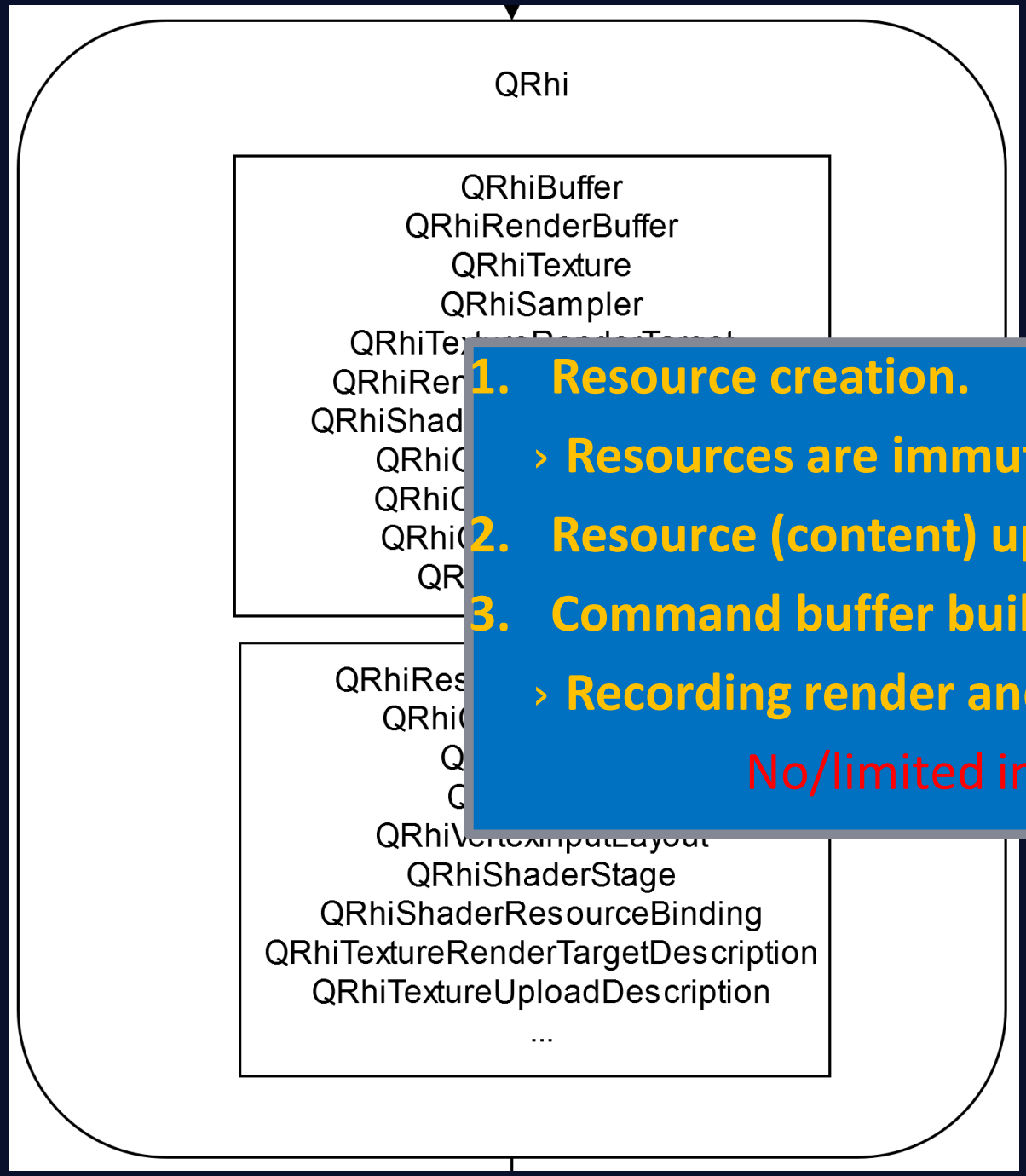
- Qt RHI
  - Part of the QtGui module
  - Private API
- Qt Shader Tools
  - qt-labs/qtshadertools
- Qt Quick port
  - Side-by-side

Components

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

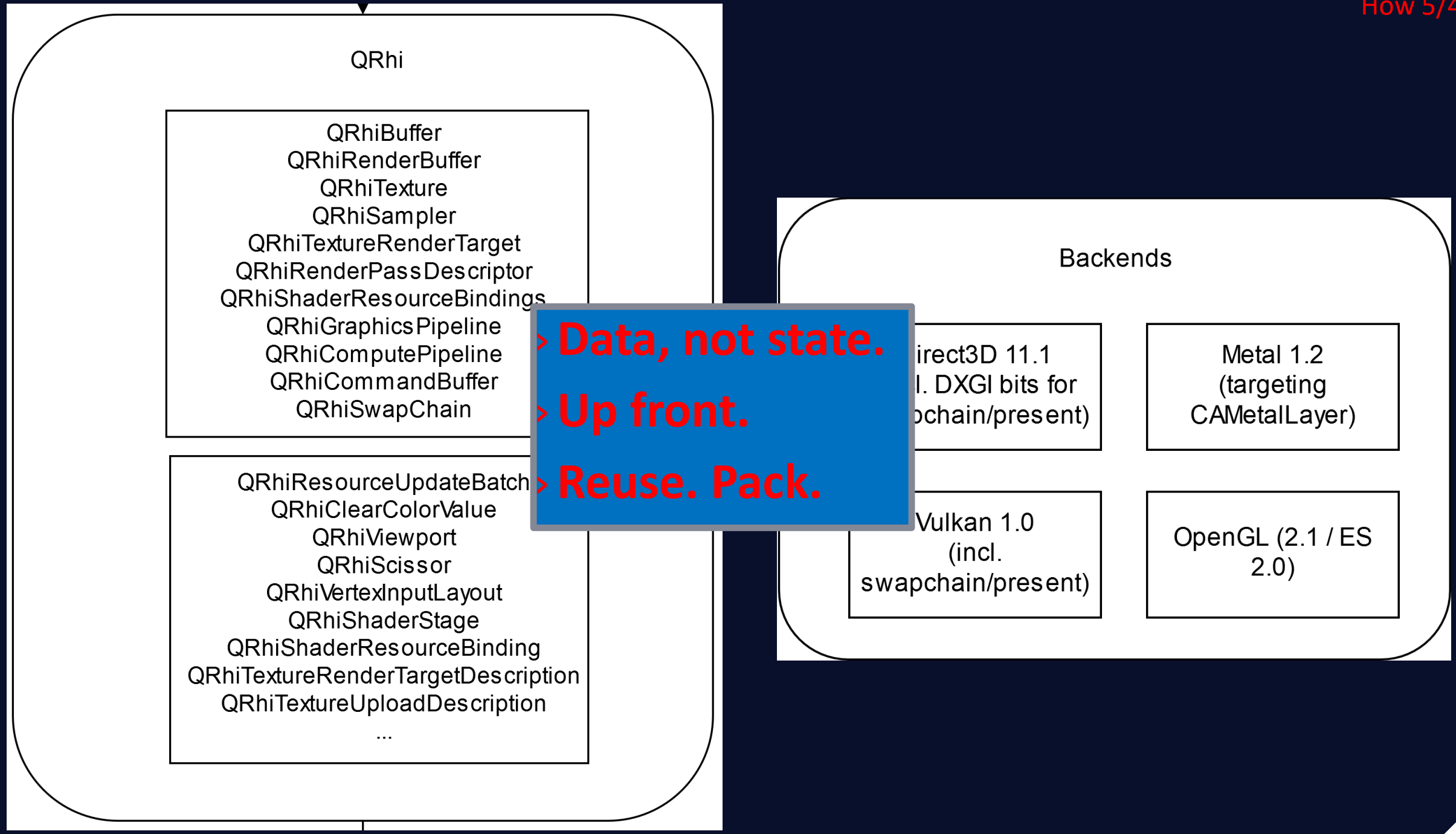


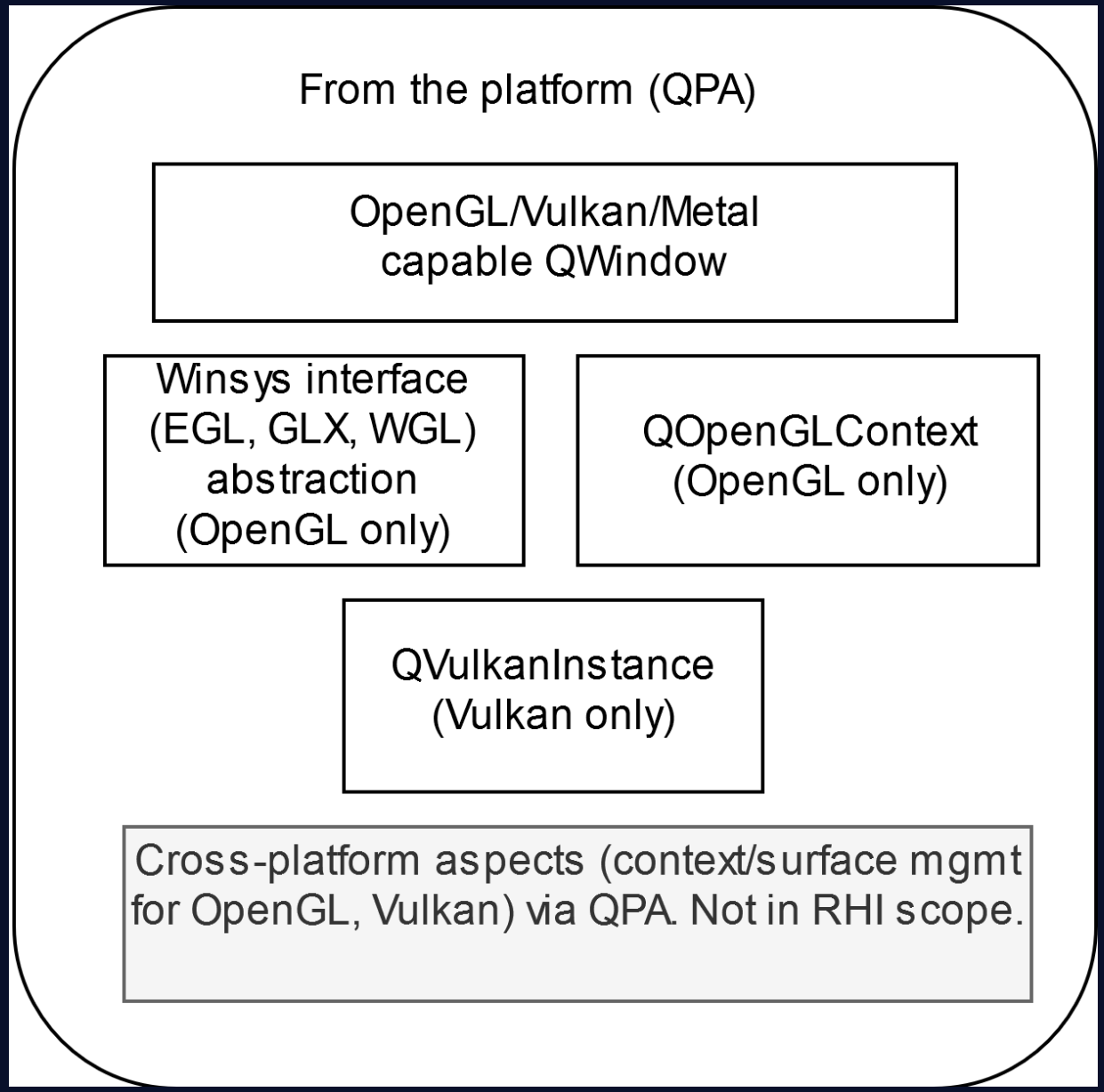

QRhi
QRhiBuffer
QRhiRenderBuffer
QRhiTexture
QRhiSampler
QRhiTextureRenderTarget
QRhiRenderPassDescriptor
QRhiShaderResourceBindings
QRhiGraphicsPipeline
QRhiComputePipeline
QRhiCommandBuffer
QRhiSwapChain
QRhiResourceUpdateBatch
QRhiClearColorValue
QRhiViewport
QRhiScissor
QRhiVertexInputLayout
QRhiShaderStage
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)



1. **Resource creation.**
   - **Resources are immutable. Contents are not.**
2. **Resource (content) updates.**
3. **Command buffer building.**
   - **Recording render and compute passes.**

No/limited intermixing of 2 & 3.


QRhi
QRhiBuffer
QRhiRenderBuffer
QRhiTexture
QRhiSampler
QRhiTextureRenderTarget
QRhiRenderPassDescriptor
QRhiShaderResourceBindings
QRhiGraphicsPipeline
QRhiComputePipeline
QRhiCommandBuffer
QRhiSwapChain
QRhiResourceUpdateBatch
QRhiClearColorValue
QRhiViewport
QRhiScissor
QRhiVertexInputLayout
QRhiShaderStage
QRhiShaderResourceBinding
QRhiTextureRenderTargetDescription
QRhiTextureUploadDescription
...
> Data, not state.
> Up front.
> Reuse. Pack.
Backends
irect3D 11.1
l. DXGI bits for
pchain/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);
```

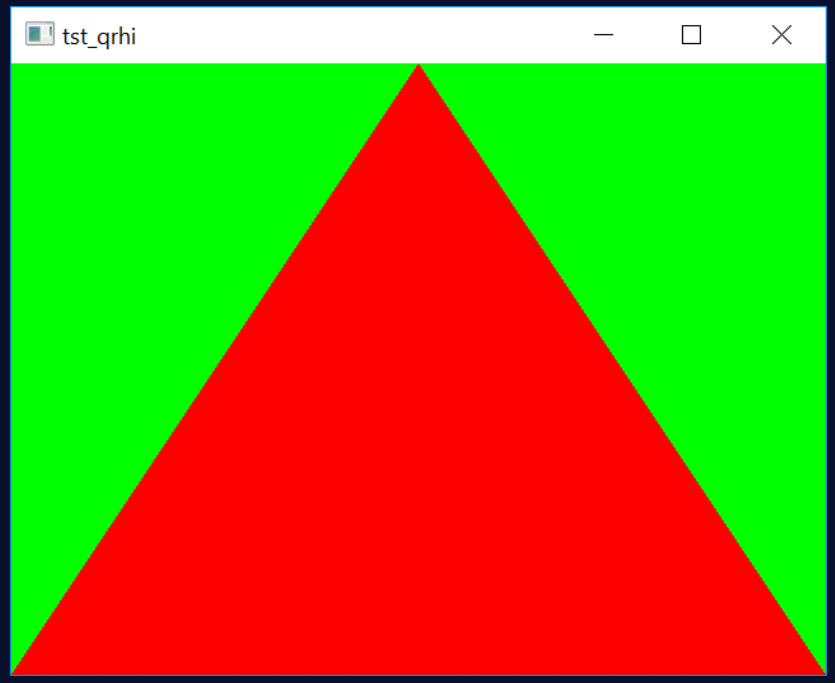
Assuming these vertex and fragment shaders…

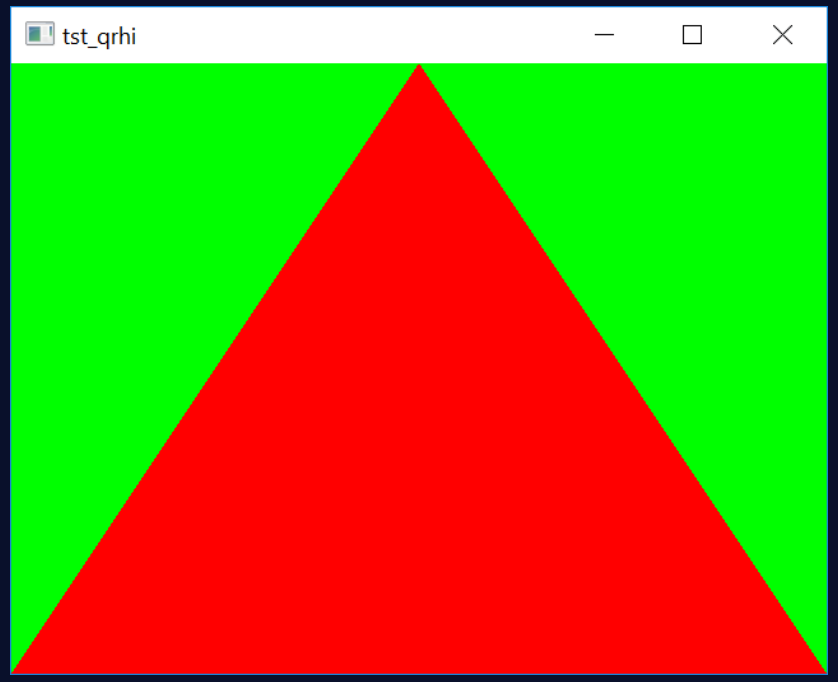
```
#version 440
layout(location = 0) in vec4 position;
out gl_PerVertex { vec4 gl_Position; };
void main()
{
    gl_Position = position;
}
```

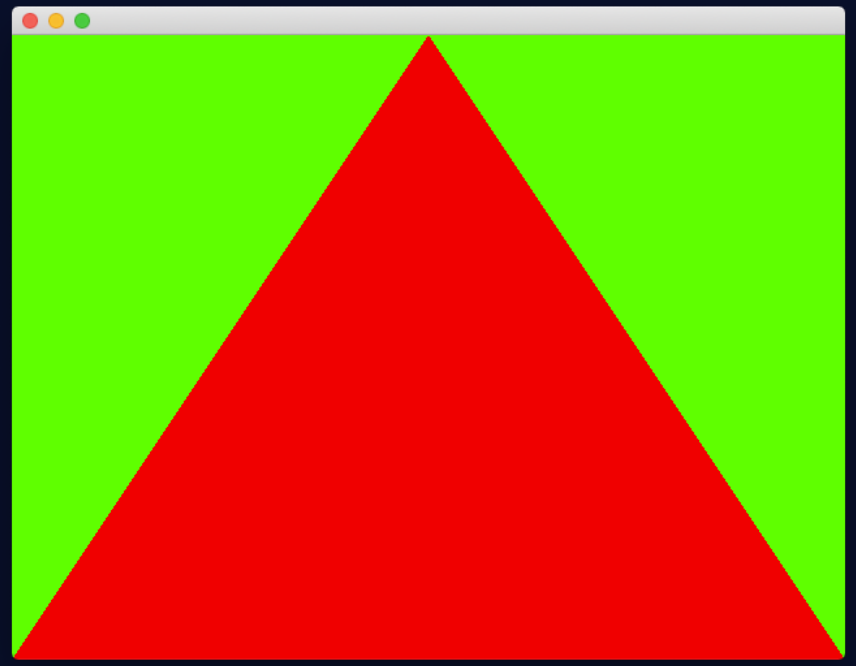
```
#version 440
layout(location = 0) out vec4 fragColor;
void main()
{
    fragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

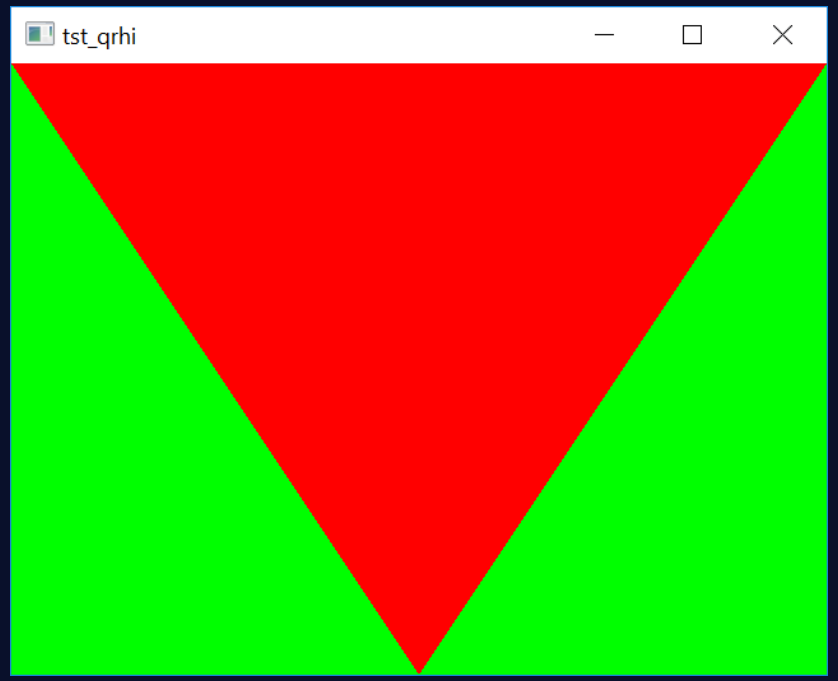
Qt

…this is what we get. Yay!



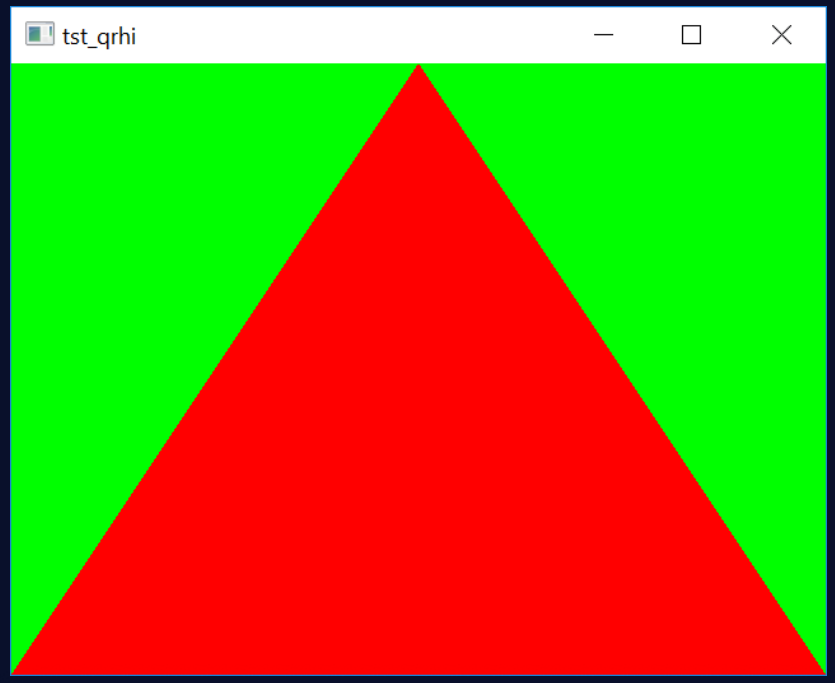


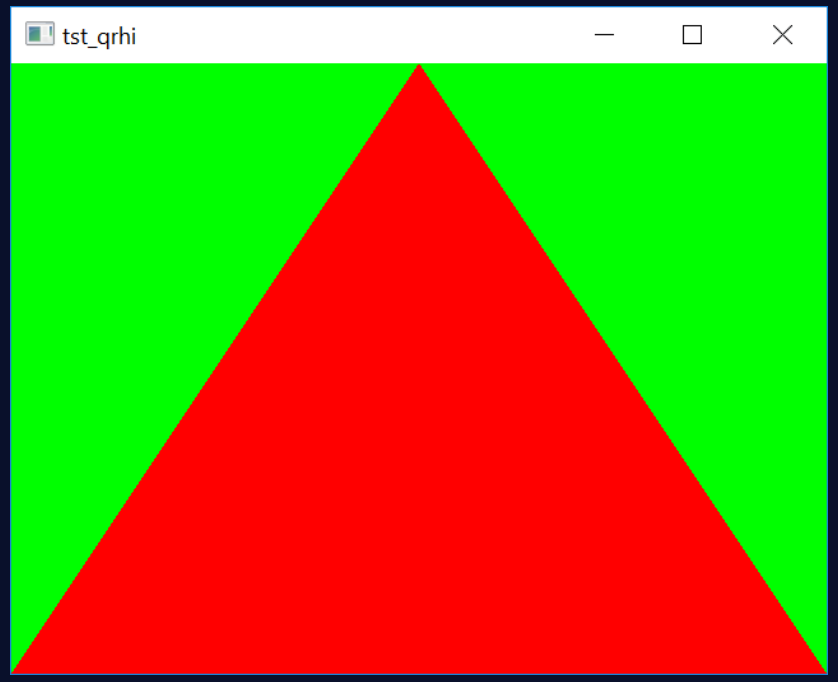


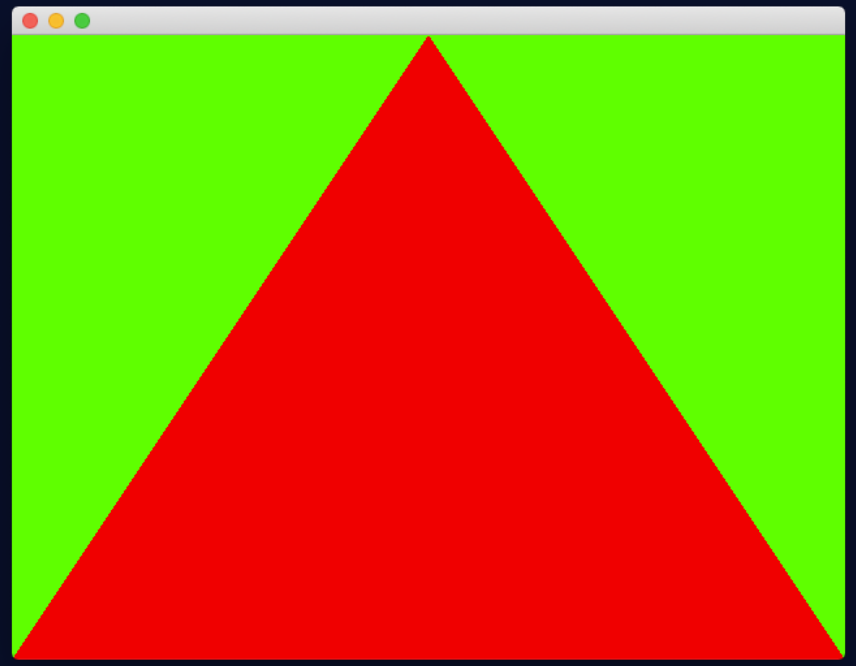


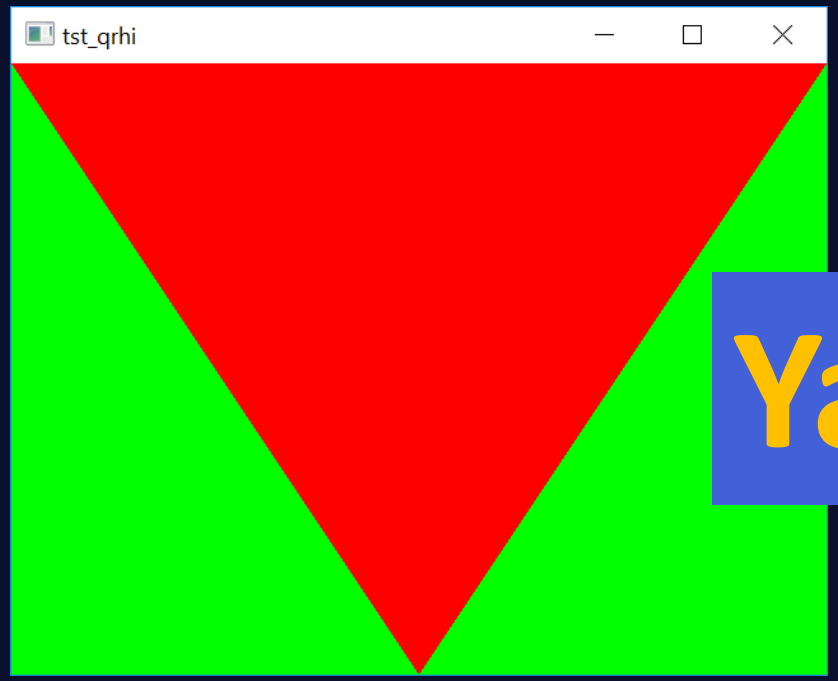
Qt

…this is what we get. Yay!





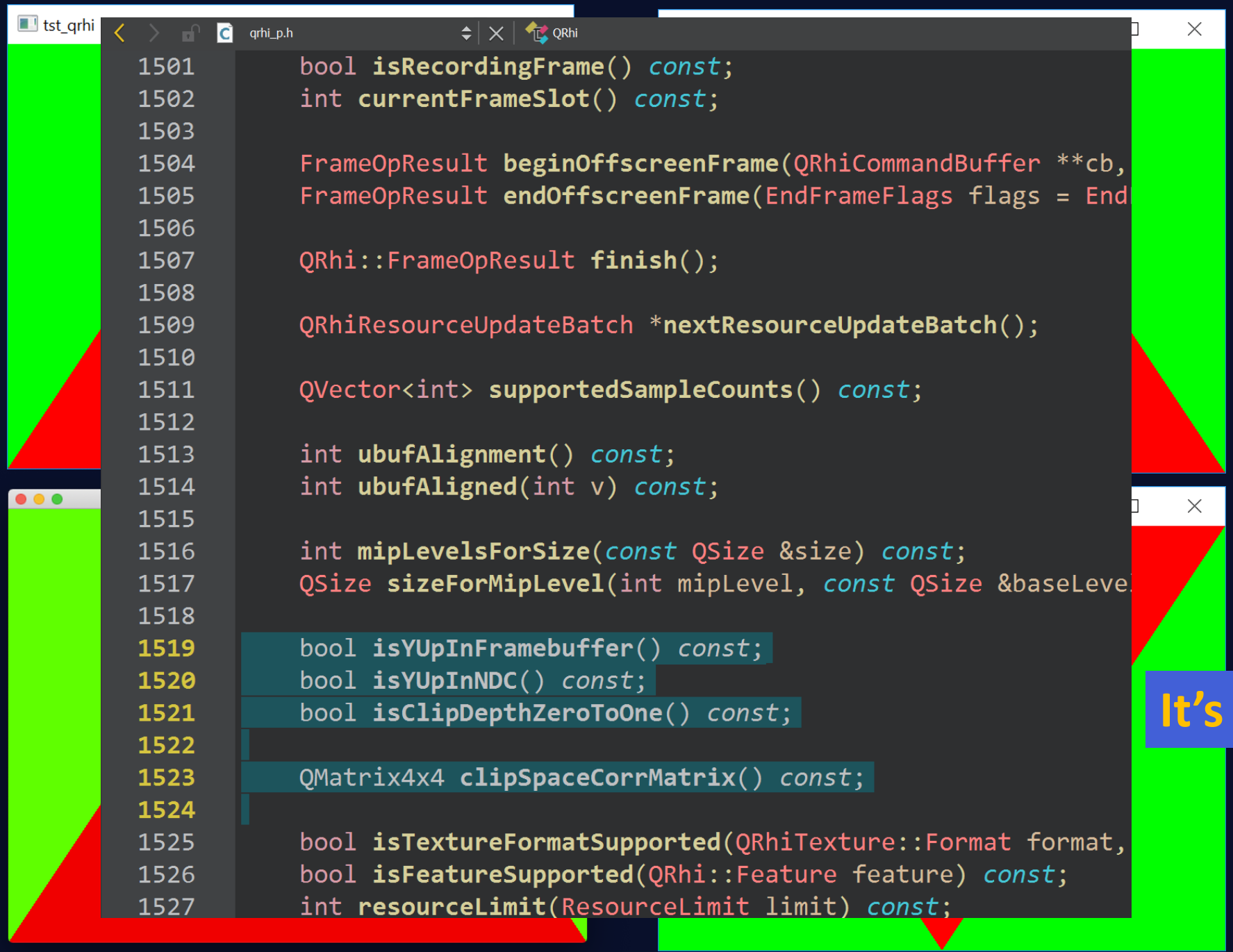




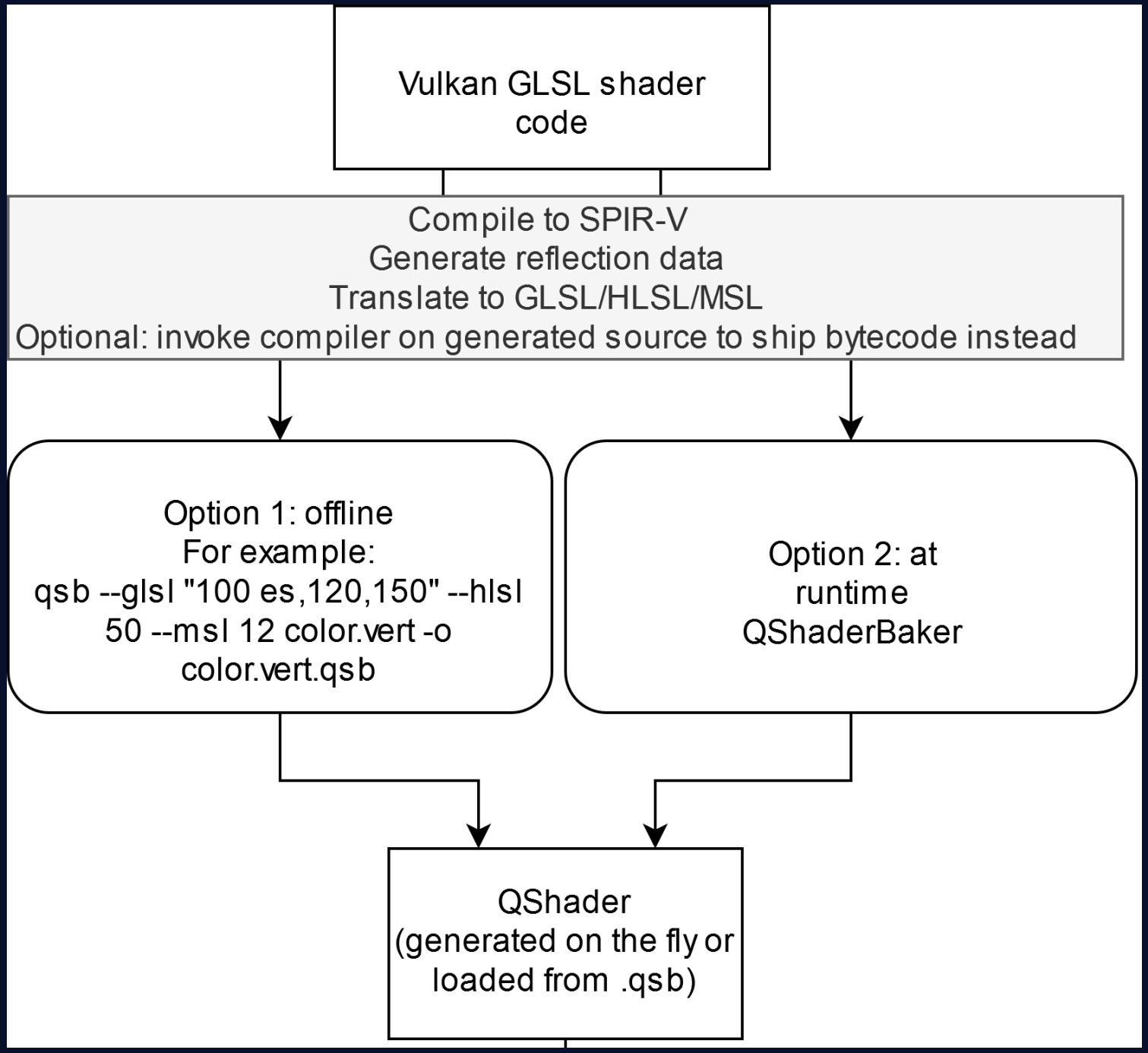
Yay?!

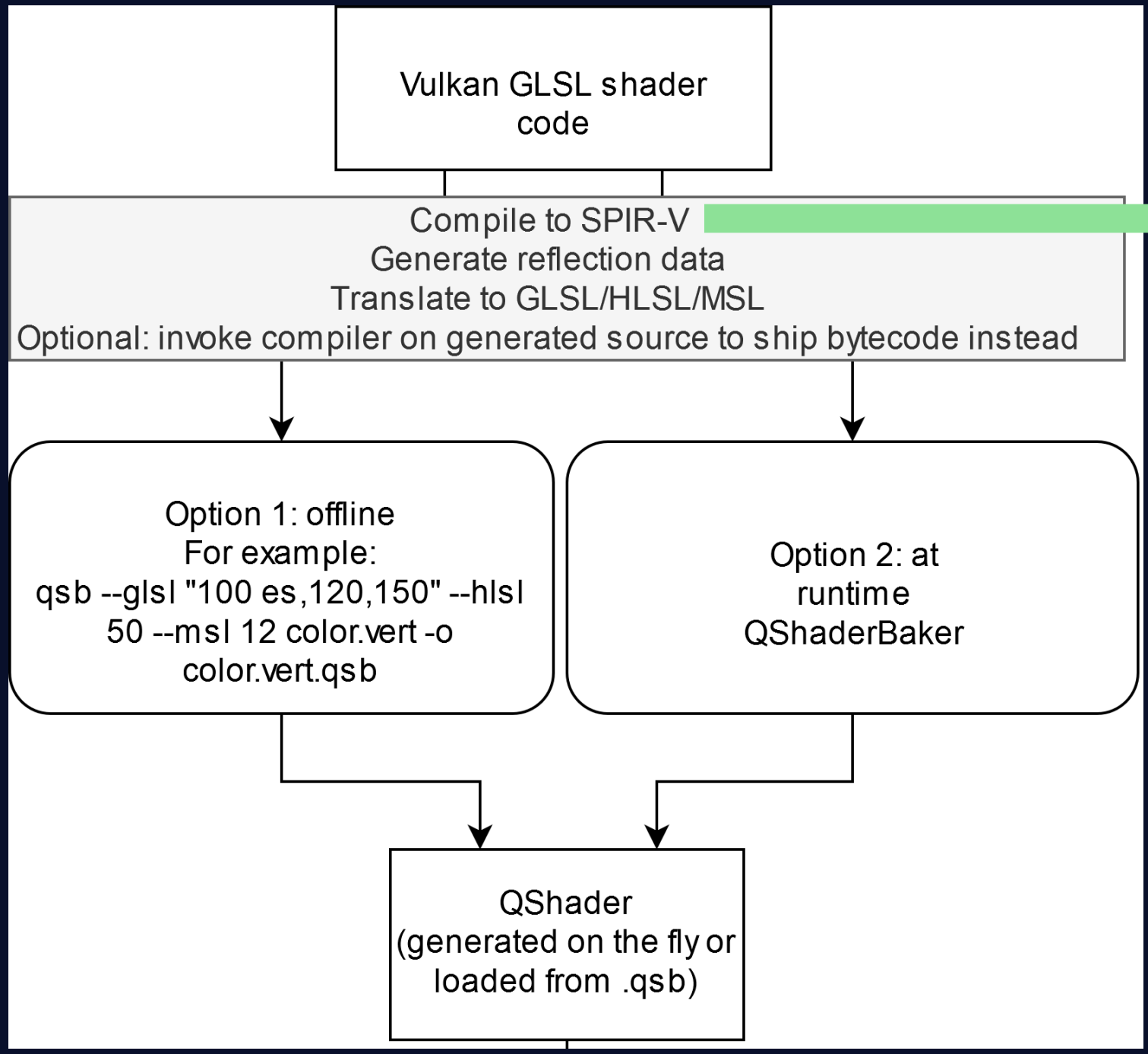
Qt

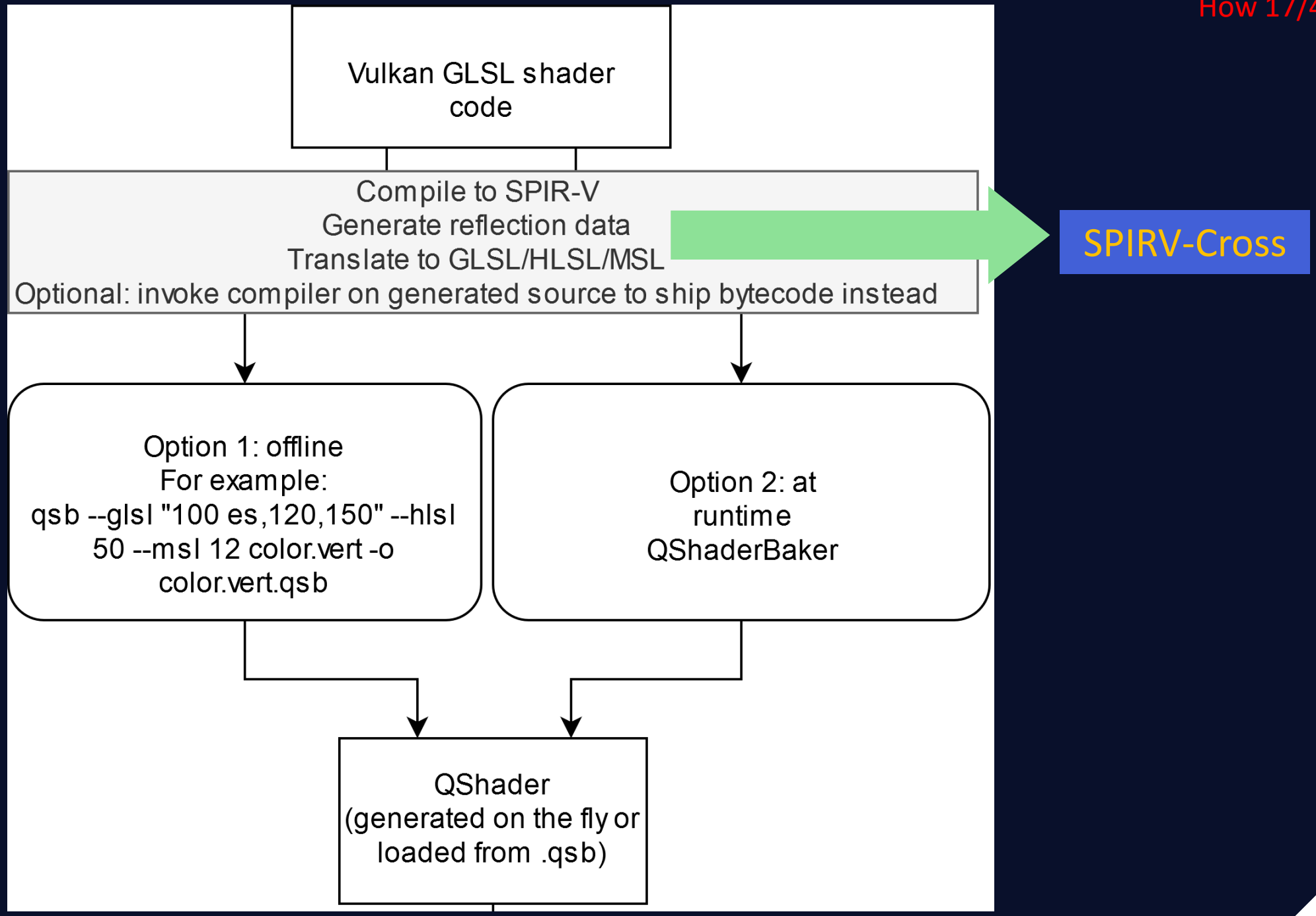
...this is what we get. Yay!

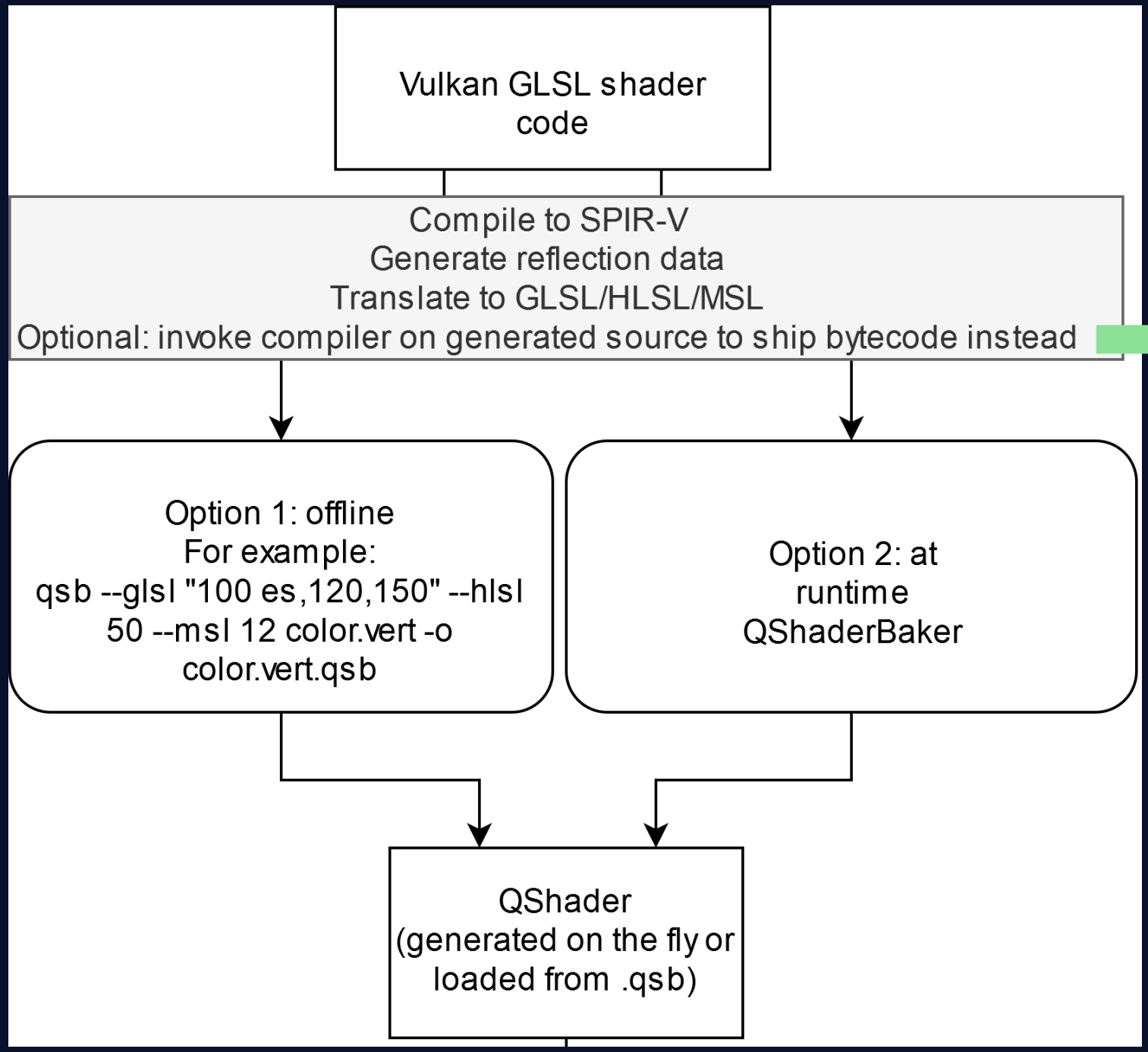


It’s fine.

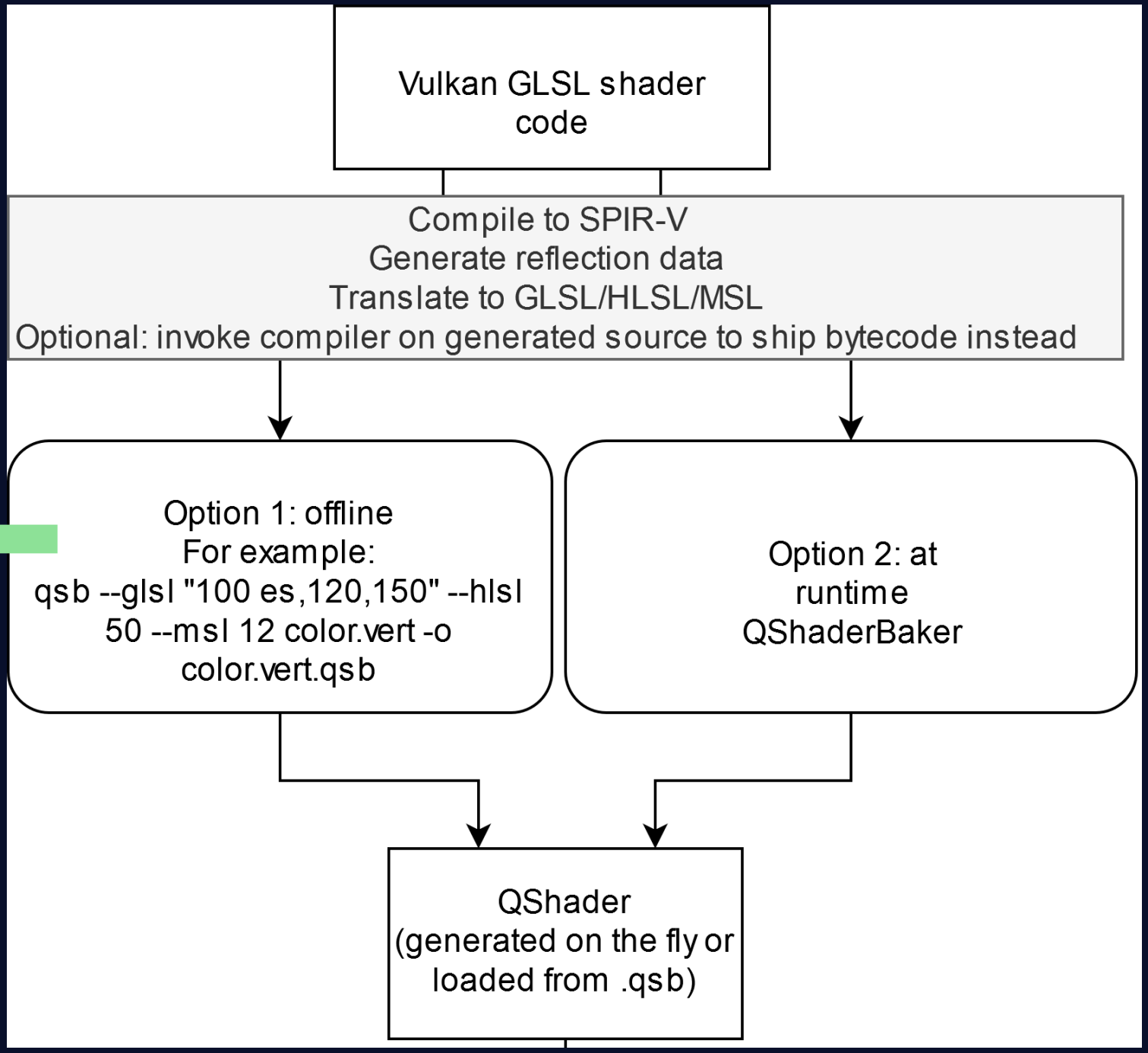

Vulkan GLSL shader code
Compile to SPIR-V
Generate reflection data
Translate to GLSL/HLSL/MSL
Optional: invoke compiler on generated source to ship bytecode instead
Option 1: offline
For example:
qsb --glsl "100 es,120,150" --hlsl 50 --msl 12 color.vert -o color.vert.qsb
Option 2: at runtime
QShaderBaker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


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
glslang
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)


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
SPIRV-Cross
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)


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
fxc, dxc,
xcrun metal(lib),
...
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"
        }
```

...

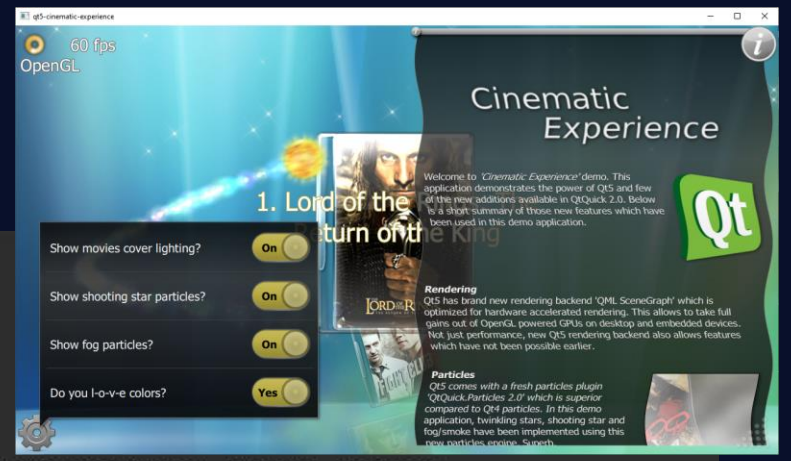
Qt Quick

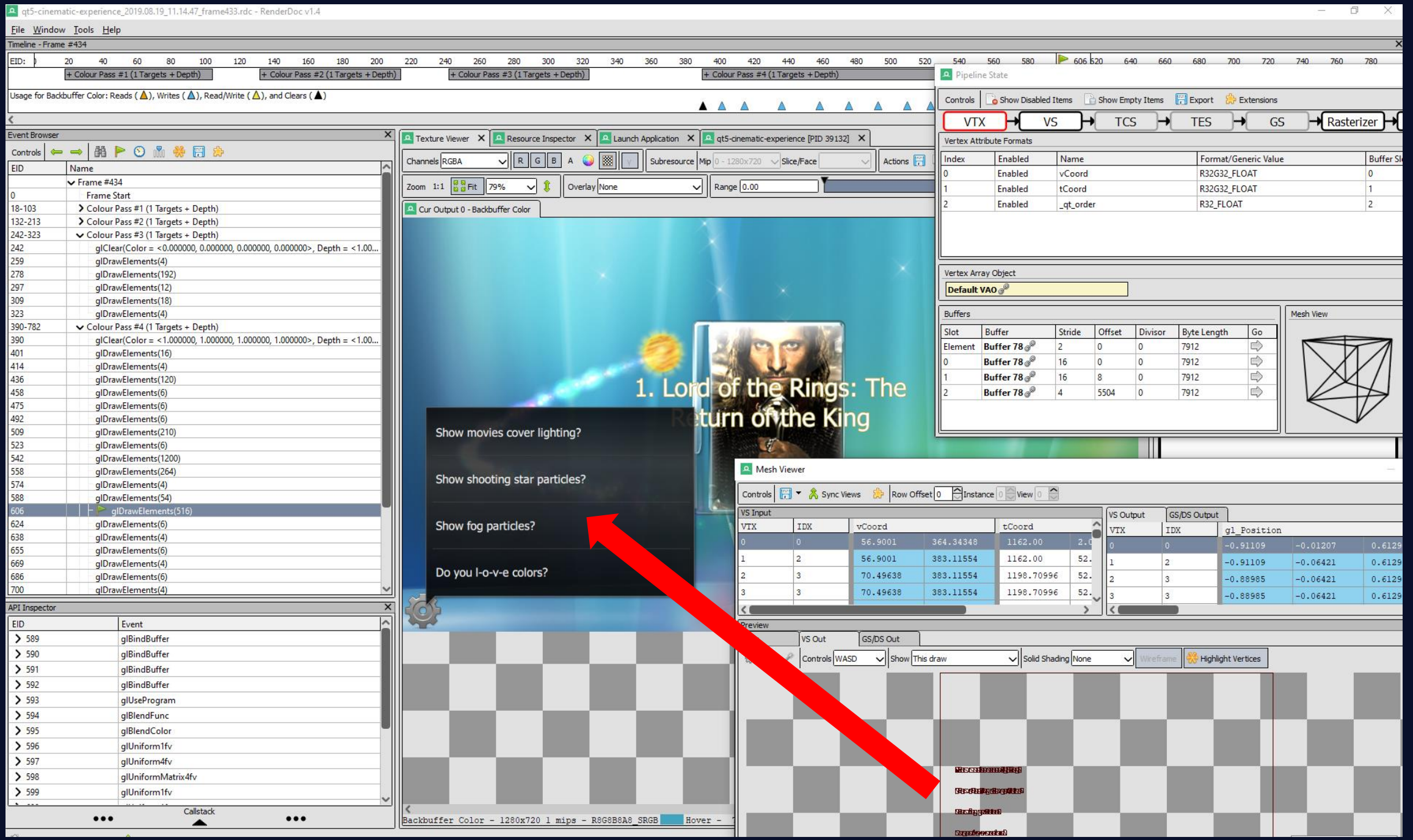
- QQuickItem tree
  - described in QML
  - main thread
- QSGNode tree + material system
  - the “scene graph”
  - render thread
- OpenGL

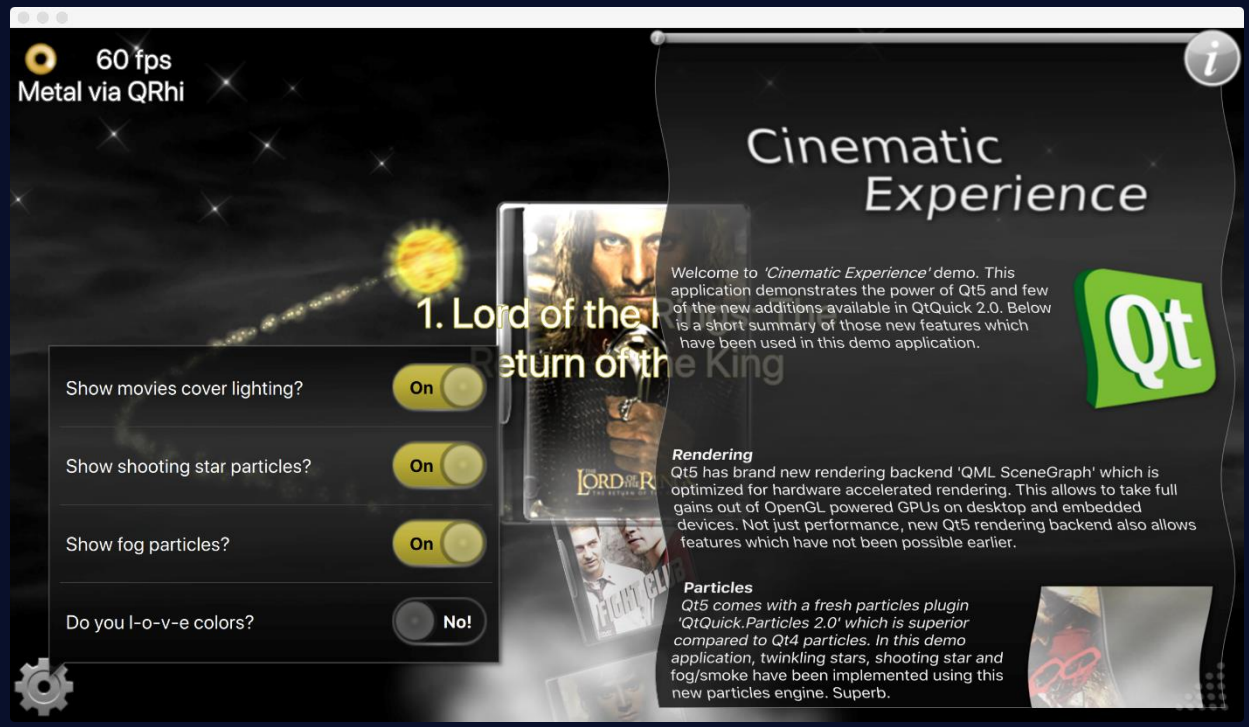
Qt

```
[ ] 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
 translat  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
t= 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
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
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
0 Enabled vCoord R32G32_FLOAT
1 Enabled tCoord R32G32_FLOAT
2 Enabled _qt_order R32_FLOAT
Vertex Array Object
Default VAO
Buffers
Slot Buffer Stride Offset Divisor Byte Length
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();
```

```
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();
```

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

```
void QSGVertexColorMaterialShader::updateState(const RenderState &state, QSGMaterial * , QSGMaterial *)
{
    if (state.isOpacityDirty())
        program()->setUniformValue(m_opacity_id, state.opacity());

    if (state.isMatrixDirty())
        program()->setUniformValue(m_matrix_id, state.combinedMatrix());
}
```

```
    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.

```
class Q_QUICK_EXPORT QSGMaterialRhiShader
{
public:
    class Q_QUICK_EXPORT RenderState {
```

```
bool QSGVertexColorMaterialRhiShader::updateUniformData(RenderState &state, QSGMaterial *, QSGMaterial *)
{
    bool changed = false;
    QByteArray *buf = state.uniformData();

    if (state.isMatrixDirty()) {
        const QMatrix4x4 m = state.combinedMatrix();
        memcpy(buf->data(), m.constData(), 64);
        changed = true;
    }

    if (state.isOpacityDirty()) {
        const float opacity = state.opacity();
        memcpy(buf->data() + 64, &opacity, 4);
        changed = true;
    }

    return changed;
}
```

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

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

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()
}
```

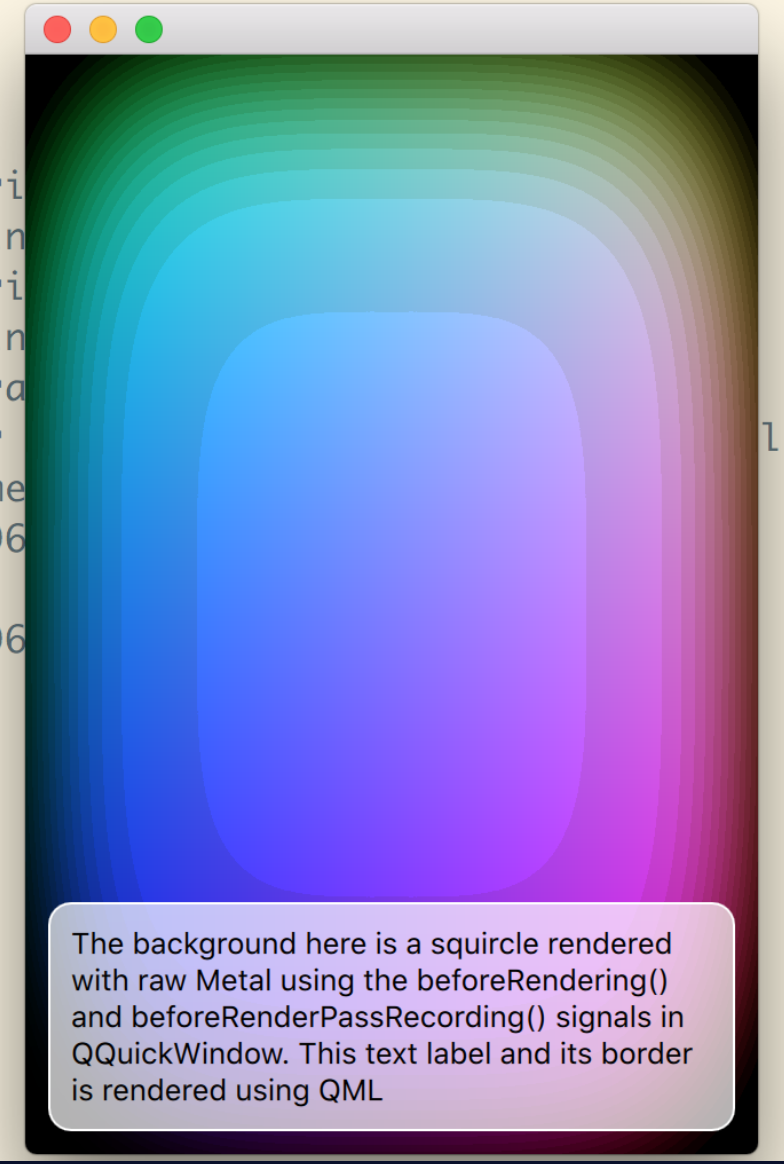
```
ShaderEffect {
    id: shaderItem

    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

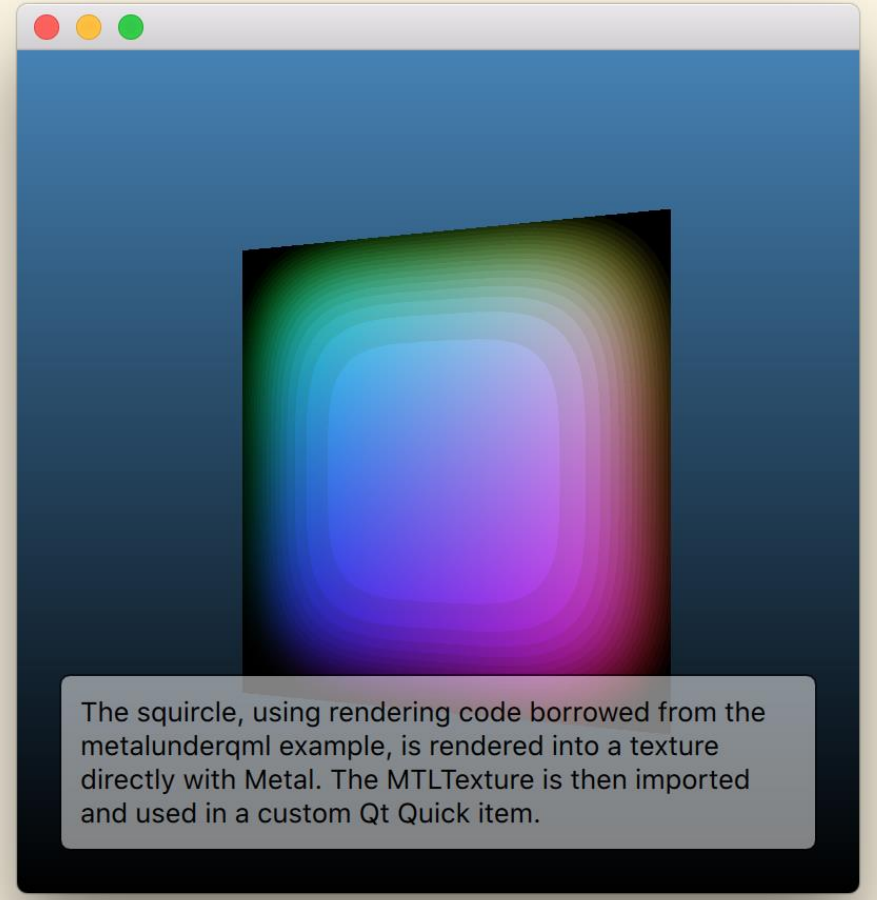
```
~/qtdeclarative_dev/examples/quick/scenegraph/metalunderqml $ ./metalunderqml.app/Contents/MacOS/metalunderqml
qt.scenegraph.general: Using QRhi with backend Metal
  graphics API debug/validation layers: 0
  QRhi profiling and debug markers: 0
qt.scenegraph.general: threaded render loop
qt.scenegraph.general: 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
```



```cpp
void VulkanSquircle::sync()
{
    if (!m_renderer) {
        m_renderer = new SquircleRenderer;
        // Initializing resources is done before starting to record the
        // renderpass, regardless of wanting an underlay or overlay.
        connect(window(), &QQuickWindow::beforeRendering, m_renderer,
                &SquircleRenderer::frameStart, Qt::DirectConnection);
        // Here we want an underlay and therefore connect to
        // beforeRenderPassRecording. Changing to afterRenderPassRecording
        // would render the squircle on top (overlay).
        connect(window(), &QQuickWindow::beforeRenderPassRecording, m_renderer,
                &SquircleRenderer::mainPassRecordingStart, Qt::DirectConnection);
    }
    m_renderer->setViewportSize(window()->size() * window()->devicePixelRatio());
    m_renderer->setT(m_t);
    m_renderer->setWindow(window());
}
void SquircleRenderer::mainPassRecordingStart()
{
    const QQuickWindow::GraphicsStateInfo &stateInfo(m_window->graphicsStateInfo());
    QSGRendererInterface *rif = m_window->rendererInterface();

    VkDeviceSize ubufOffset = stateInfo.currentFrameSlot * m_allocPerUbuf;
    void *p = nullptr;
    VkResult err = m_devFuncs->vkMapMemory(m_dev, m_ubufMem, ubufOffset, m_allocPerUbuf, 0, &p);
    if (err != VK_SUCCESS || !p)
        qFatal("Failed to map uniform buffer memory: %d", err);
    float t = m_t;
    memcpy(p, &t, 4);
    m_devFuncs->vkUnmapMemory(m_dev, m_ubufMem);

    m_window->beginExternalCommands();

    // Must query the command buffer _after_ beginExternalCommands(), this is
    // actually important when running on Vulkan because what we get here is a
    // new secondary command buffer, not the primary one.
    VkCommandBuffer cb = *reinterpret_cast<VkCommandBuffer *>(
                rif->getResource(m_window, QSGRendererInterface::CommandListResource));
```

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 takeaway from porting the Qt Quick Renderer onto QRhi

qsgbatchrenderer.cpp | Renderer::renderBatches() -> void

```
4003
4004        if (Q_LIKELY(renderOpaque)) {
4005            for (int i=0; i<m_opaqueBatches.size(); ++i) {
4006                Batch *b = m_opaqueBatches.at(i);
4007                if (b->merged)
4008                    renderMergedBatch(b);
4009                else
4010                    renderUnmergedBatch(b);
4011            }
4012        }
4013
4014        glEnable(GL_BLEND);
4015        if (m_useDepthBuffer)
4016            glDepthMask(false);
4017        glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
4018
4019        if (Q_LIKELY(renderAlpha)) {
4020            for (int i=0; i<m_alphaBatches.size(); ++i) {
4021                Batch *b = m_alphaBatches.at(i);
4022                if (b->merged)
4023                    renderMergedBatch(b);
4024                else if (b->isRenderNode)
4025                    renderRenderNode(b);
4026                else
4027                    renderUnmergedBatch(b);
4028            }
4029        }
```

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);
}
```

```
// 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 QRhi or graphics API)

```
// 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.

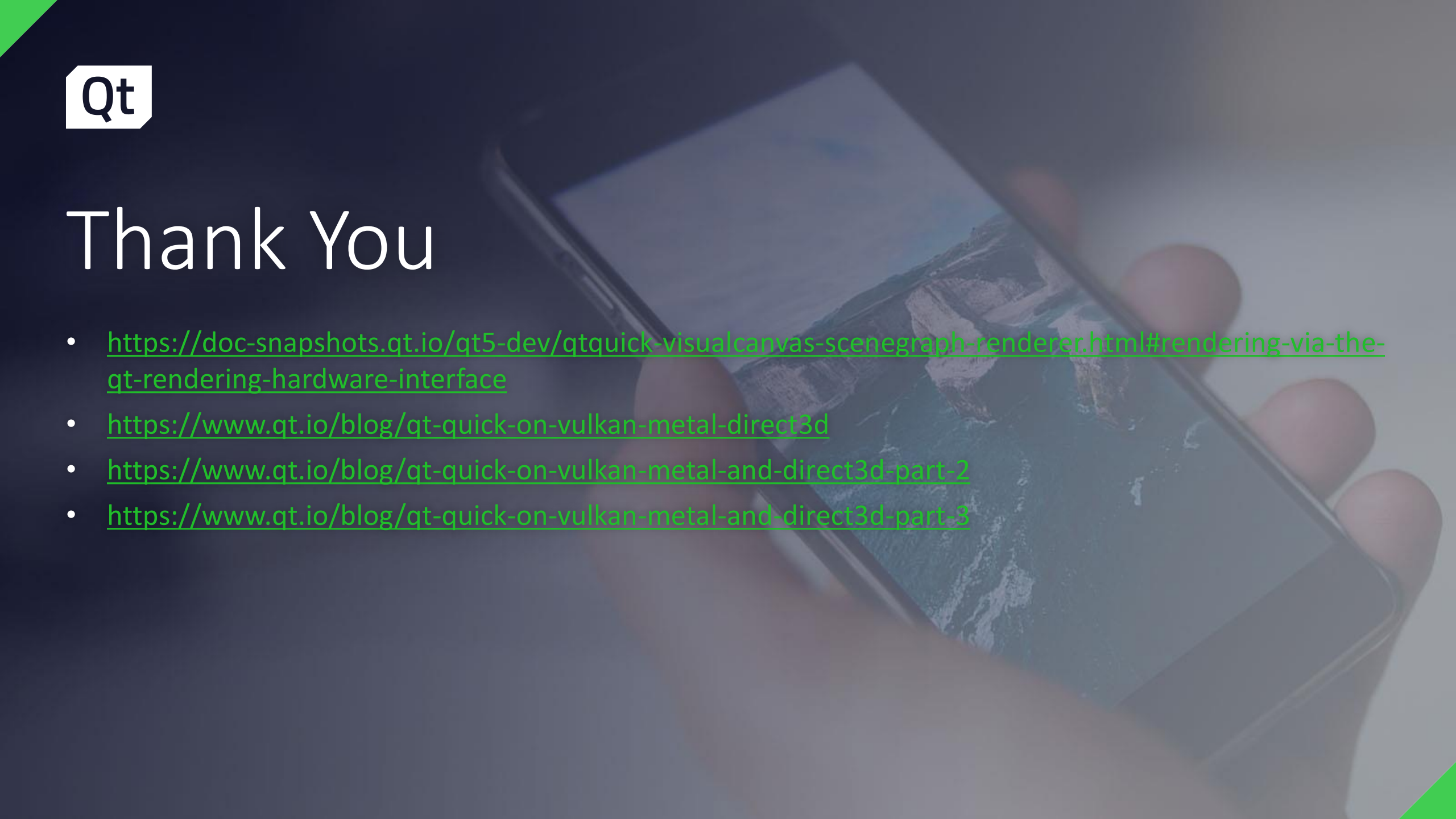
```
// depth test stays enabled but no need to write out depth from the
// transparent (back-to-front) pass
m_gstate.depthWrite = false;

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

```
// 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);
```

- **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.

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



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