Many 3D graphic visuals can be handled with the same no-frills rendering techniques software developers learn during an intro 3D course. However, what if you want to incorporate one of many advanced rendering techniques – effects like shadows, refractive surfaces, water effects, HDR lighting, lens flares, or focus blurs?

The previous two parts of our three part Qt 3D series focused on drawing a 3D model and accepting user input, which are necessary building blocks of 3D applications. For the last part of our Qt 3D whitepaper series, we’ll look at how to use Qt 3D and frame graphs to create some really sophisticated-looking graphics, including how to implement multi-pass rendering.

Qt 3D’s rendering engine

To get the most out of the Qt 3D renderer, we need to start by reviewing a few basics. Simply put, the renderer’s job is to determine the color of every pixel on the screen each time a new frame is drawn.

The basic components that the renderer uses are always the same: object geometries, object materials/textures, shaders, and uniforms. The geometry, materials, and textures control the placement and appearance of each object in a scene. Shaders are small programs that
describe how to translate a combination of attributes into a particular look, like how to handle glossy, matte, or transparent materials. **Uniforms** are variables the program passes to a shader for controlling general aspects of the program's behaviour, like light sources, fog, or focus effects. The renderer combines all of these elements in various ways as part of a GPU pipeline to specify the final look and feel of the graphical composition.

The primary mechanism used by Qt 3D to configure the graphical pipeline is the frame graph. A frame graph contains a number of nodes in a tree that define how the renderer will draw a scene, and is defined by creating a **FrameGraph** component. By customizing and reordering nodes within the FrameGraph, you can build the rendering algorithm of your choice. This architecture allows Qt 3D's rendering to be infinitely extensible.

**The benefits of FrameGraph**

Since the frame graph tree is entirely data-driven and can be modified dynamically at runtime, it gives you a lot of flexibility. You can use different frame graph trees for different platforms and hardware, and select the most appropriate at runtime. You can easily add and enable visual debugging in a scene. You can use different frame graph trees depending on what you need to render for particular regions of the scene. And finally, you can implement a new rendering technique without having to modify Qt 3D's internals.

This is just a taste of what FrameGraph can do – let's dig into the details to see how it works.

**FrameGraph operation**

The FrameGraph renderer is simply configured (relatively speaking) in QML (as well as in C++), making it easy for graphics experts and non-experts alike to modify a rendering technique. Even though it may be easy to configure, understanding a few important rules about how it works is necessary if you want your renderings to come out properly.

**Depth-first traversal** – The Qt 3D renderer visits frame-graph nodes using a depth-first traversal. That means a nested node will have its children visited before its siblings, which can make a significant difference in the rendered output. We'll look at some situations where node order matters (as well as when it doesn't) a bit later on.

- **RenderView construction** – The renderer constructs a RenderView for each leaf of the node tree that it encounters during the tree traversal. A RenderView contains all of the rendering state information from each leaf back to the tree's root node.

- **RenderCommand building** – Once a RenderView is created, the renderer adds RenderCommands to the RenderView to represent all the Entities within the SceneGraph that require rendering. RenderCommands are the translation of the scene graph objects into whatever OpenGL primitives (vertex arrays, shaders, uniforms, meshes, etc) are necessary for the GPU to display the object.

- **RenderView processing** – Each RenderView is processed by OpenGL to turn its associated list of RenderCommands into a frame buffer – an in-memory representation of the screen content.

- **Frame buffer display** – Once this process is repeated for all of the nodes within the frame graph, the frame is complete. At this point, the renderer calls swapBuffers() to exchange the newly drawn frame buffer with the current screen content and reads itself to be called for the next frame.
One standard way to organize the graphical pipeline is called forward rendering, which renders each object into the frame buffer in final form.

Because the processing load is heavily dependent on the total number of leaf nodes in a frame graph tree, reducing the number of leaf nodes to the absolute minimum required will make your renderer configuration more efficient. Moving state information that remains constant across multiple nodes closer to the frame graph tree's root can also help you optimize performance.

**FrameGraph nodes**

We've talked a bit about the nodes in a frame graph tree. But what exactly is a node for? Essentially, each node encapsulates a bit of functionality required by the renderer at each stage of execution. Qt 3D provides a number of pre-defined frame graph nodes for use in building your renderer. Although these nodes are all subclasses of the C++ class, Qt3D::QFrameGraphNode, they are accessible in QML too. By combining these simple node types it is often possible to configure the renderer without writing a single line of application-specific C++ rendering code.

<table>
<thead>
<tr>
<th>FrameGraph Node</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CameraSelector</td>
<td>Sets a camera to perform the rendering</td>
</tr>
<tr>
<td>ClearBuffers</td>
<td>Specifies OpenGL buffers to be cleared</td>
</tr>
<tr>
<td>LayerFilter</td>
<td>Specifies entities to render by filtering entities with a matching layer component</td>
</tr>
<tr>
<td>RenderPassFilter</td>
<td>Defines the filters used to select a render pass in a technique</td>
</tr>
<tr>
<td>RenderStateSet</td>
<td>Sets various rendering states</td>
</tr>
<tr>
<td>RenderSurfaceSelector</td>
<td>Specifies the surface to draw to</td>
</tr>
<tr>
<td>RenderTargetSelector</td>
<td>Specifies the rendering target (i.e. frame buffer)</td>
</tr>
<tr>
<td>SortPolicy</td>
<td>Controls how rendering commands are sorted</td>
</tr>
<tr>
<td>TechniqueFilter</td>
<td>Defines the filters used to select a technique in a material</td>
</tr>
<tr>
<td>Viewport</td>
<td>Defines the rectangular viewport where the scene is drawn</td>
</tr>
</tbody>
</table>

List of frame graph nodes. [Consult the Qt 3D documentation](https://doc.qt.io/qt-3d) to see additional node types not listed here.

**Forward rendering**

Let's put some of our newfound frame graph knowledge into practice by building a simple renderer. One standard and straightforward way to organize the graphical pipeline is called forward rendering, which renders each object into the frame buffer in final form. Forward rendering uses a vertex -> fragment -> pixel transformation process as follows:

1. Object vertices are transformed by the viewer's current viewport and viewing angle. These vertices represent the visible triangle meshes that wrap an object's volume. Triangles that face away from the viewer or are outside the screen boundary are discarded.

2. Triangles are rasterized into many individual fragments. Fragments are “hopeful” pixels – they may or may not end up being displayed, depending on the fragment shader algorithm (which may choose to discard the fragment), the fragment's transparency (which may make it invisible), and the presence of other overlapping objects (which may cover the fragment).

3. Fragment shaders operate on the fragments to calculate their color and/or transparency, based on surface texture, lighting, normal (the surface's angle to the light), and any other variables the app may require.

4. The GPU finds the fragments at the same pixel location sorted by distance from the viewer and uses the closest fragment to determine the final pixel color.

Incidentally, a forward renderer is the type of renderer used by Qt Quick 2. Even though Qt Quick 2 is limited to 2D, its content is rendered in 3D to take advantage of OpenGL hardware acceleration. Defining the frame graph needed for a forward renderer is pretty simple, especially in QML:
If the state collected at each leaf is the same between two FrameGraph trees, the resulting renderer will create visually identical results.

**Code sample 1: forward renderer**
```cpp
/* Creates a frame graph tree with Viewport | Clear Buffers | Camera */
RenderSettings {
  id: root
  activeFrameGraph: Viewport {
    normalizedRect: Qt.rect(0.0, 0.0, 1.0, 1.0)
    property alias viewCamera: viewCameraSelector.camera
    ClearBuffers {
      buffers: ClearBuffers.
      ColorDepthBuffer
      CameraSelector {
        id: viewCameraSelector
      }
    }
  }
}
```

This resulting frame graph tree has three nodes with a single leaf.

During the rendering process, this frame graph tree creates a single `RenderView` that sets the view port to fill the entire screen (using normalized unit coordinates), clears the color and depth buffers, and sets the camera to the exposed camera property. If the state collected at each leaf is the same between two frame graph trees, the resulting renderer will create visually identical results. As an example, here are two additional frame graph configurations that produce the same outcome as the renderer in code sample 1.

**Code sample 2: Another forward renderer that works like Code sample 1**
```cpp
/* Swap camera and clear buffers: Viewport | Camera | Clear Buffers */
RenderSettings {
  id: root
  activeFrameGraph: Viewport {
    normalizedRect: Qt.rect(0.0, 0.0, 1.0, 1.0)
    property alias viewCamera: viewCameraSelector.camera
    CameraSelector {
      id: viewCameraSelector
    }
    ClearBuffers {
      buffers: ClearBuffers.
      ColorDepthBuffer
    }
  }
}
```

**Code sample 3: Yet one more forward renderer with the same result**
```cpp
/* An equivalent configuration: Camera | Viewport | Clear Buffers */
RenderSettings {
  id: root
  activeFrameGraph: CameraSelector {
    id: viewCameraSelector
    Viewport {
      normalizedRect: Qt.rect(0.0, 0.0, 1.0, 1.0)
      ClearBuffers {
        buffers: ClearBuffers.
        ColorDepthBuffer
      }
    }
  }
}
```
A slightly more complex example of a renderer draws a scene graph from four different viewpoints, each in a separate quadrant of the screen.

### Multiple viewport rendering
A slightly more complex example of a renderer draws a scene graph from four different viewpoints, each in a separate quadrant of the screen. This type of multi-viewpoint renderer might make sense in any application that requires multiple views of the same scene, such as a CAD or 3D printing visualization tool, a car racing game with a rear-view mirror, or a security monitoring station.

#### Code sample 4: Multi-viewport renderer

```cpp
RenderSettings {
  id: root
  activeFrameGraph: Viewport {
    id: mainViewport
    normalizedRect: Qt.rect(0, 0, 1, 1)
  }
  ClearBuffers {
    buffers: ClearBuffers.
    ColorDepthBuffer
  }
  Viewport {
    id: topLeftViewport
    normalizedRect: Qt.rect(0, 0, 0.5, 0.5)
  }
  Viewport {
    id: topRightViewport
    normalizedRect: Qt.rect(0.5, 0, 0.5, 0.5)
  }
  Viewport {
    id: bottomLeftViewport
    normalizedRect: Qt.rect(0, 0.5, 0.5, 0.5)
  }
  Viewport {
    id: bottomRightViewport
    normalizedRect: Qt.rect(0.5, 0.5, 0.5, 0.5)
  }
}
```

The frame graph tree for this example is a bit more complex, resulting in five leaves and five corresponding `RenderView` objects. The following diagrams show the construction for the first two `RenderViews` (the other three `RenderViews` aren’t shown but are just like the second `RenderView` albeit for the remaining viewports).

Although in our first example the nodes were arbitrarily arranged, in this example the order of the nodes matters a lot. If the `ClearBuffers` node was at the end instead of at the beginning, the rendering engine would create a black screen – everything that was carefully drawn would be erased as the last step. Similarly, the `ClearBuffers` node couldn’t be at the root of the frame graph tree or the screen would be cleared before each viewport was rendered, leaving only the last drawn viewport visible.
Complexities in light can be very tricky to display with a forward renderer; that’s where deferred rendering comes into play.

**Deferred rendering**

A forward renderer really is best when your lighting needs are simple. Scenes with multiple lights, reflected or refracted light, shadows, or other complexities in the light can be very tricky to display with a forward renderer. That’s where deferred rendering comes into play.

Deferred rendering takes multiple passes through a scene. An early pass sets state information such as normal vector, color, depth, or position that the renderer will use in subsequent passes. These pre-computed values are not displayed but stored in a buffer called the geometry buffer (or G-buffer) that resides in a texture. Once the object meshes have been drawn, the G-buffer contains everything that can be seen by the current camera. A second pass then renders the scene to a buffer with the final colors computed from the G-buffer information.

Qt 3D uses a dedicated thread to submit the work to the GPU. Although the generation of render commands is done in parallel for each render view, the thread submits OpenGL commands to the GPU in an order that ensures correct rendering.

**Parallel processing**

The declaration order for the frame graph is important in specifying its behaviour but it doesn’t imply a draw order. That’s because Qt 3D can process each RenderView in parallel, depending on the number of available CPU cores.

*Resulting display of the multi-viewport renderer*

*Multi-viewport render views operating in parallel*

*Visualization of referred renderer G-buffer contents: diffuse color (top left), depth buffer (top right), normal vectors (bottom left), and world position (bottom right)*
Output of deferred renderer’s second pass that combines all G-buffer elements

Deferred rendering decouples the generation of the scene’s geometry from any lighting effects so that the geometry for each object is only calculated once. Another advantage is that the shading and lighting computations, which are often complex, are dependent on the resolution of the screen rather than the number of objects in a scene. Deferred rendering does have some drawbacks – it doesn’t handle transparency well, it uses a lot of GPU memory bandwidth, and it doesn’t play well with hardware anti-aliasing. However, because this technique can easily render many lights in a scene and handle object shadowing, it’s often a popular choice for games.

**Code sample 5: Deferred renderer**

```cpp
Viewport {
  id: root
  normalizedRect: Qt.rect(0.0, 0.0, 1.0, 1.0)

  property GBuffer gBuffer
  property alias camera: sceneCameraSelector.camera

  // SceneEntities should reference one // of these to be incorporated into // correct layer
  readonly property Layer sceneLayer {} ➌

  readonly property Layer screenQuadLayer {}

  readonly property real windowWidth:
  surfaceSelector.surface !== null
  ? surfaceSelector.surface.width: 0

  readonly property real windowHeight:
  surfaceSelector.surface !== null
  ? surfaceSelector.surface.height: 0

  RenderSurfaceSelector {
    id: surfaceSelector

    // Fill G-Buffer
    LayerFilter {
      layers: sceneLayer
      RenderTargetSelector {
        id: gBufferTargetSelector ➌
        target: gBuffer

        ClearBuffers {
          buffers: ClearBuffers.
          ColorDepthBuffer ➍

          RenderPassFilter {
            id: geometryPass
            matchAny: FilterKey { name: "pass": value: "geometry" }

            CameraSelector {
              id: sceneCameraSelector ➏
            }
          }
        }
      }
    }

    TechniqueFilter {
      parameters:
      Parameter { name: "color": value: gBuffer.color },
      Parameter { name: "position": value: gBuffer.position },
      Parameter { name: "normal": value: gBuffer.normal },
      Parameter { name: "depth": value: gBuffer.depth },
      Parameter { name: "winSize": value: Qt.size(1024, 1024) }
    ]
```

Because deferred rendering can easily render many lights in a scene and handle object shadowing, it’s often a popular choice for games.
Shadow rendering, our last example, is the most complex, building on everything we’ve learned thus far.

The resulting renderer works like this, starting with the first render view:

1. Defines a viewport that fills the whole screen
2. Defines our layers for use within subsequent LayerFilter definitions
3. Sets the G-buffer as the active render target
4. Clears the color and depth on the G-buffer
5. Selects only entities in the scene that have a material and technique matching the render pass filter (“geometry”)
6. Specifies which camera should be used

Then the second render view:

7. Selects all entities that have a layer component matching screenQuadLayer
8. Clears the color and depth buffers on the currently bound frame buffer (the screen)
9. Selects only entities in the scene that have a material and technique matching the filter key in the render pass filter (“final”)

Shadow rendering

Our last example is the most complex, building on everything we’ve learned thus far. We’ll be creating a renderer that can handle the following scene, as well as all the code to create (and animate) the objects within. The source code for this example is in Qt Creator examples under shadow-map-qml.
This example shows us how to render shadows. It's important to note that the trefoil knot and the toy plane both cast shadows on themselves, and the plane also casts a shadow on the knot. How is this achieved within our renderer? By using a shadow mapping technique - this generates decent looking shadows without introducing a large performance hit.

Shadow mapping is normally implemented in two passes. The first pass generates the shadow information by identifying fragments on which the light is shining. The second pass renders the scene objects and uses the first pass shadow information to determine appropriate lighting for each fragment.

The first pass needs to determine which fragments are in a straight line from the light source. To accomplish this, we draw the scene with a camera at the light source – in other words, in light space – viewing the scene from the point-of-view of the light. We store the distance of the fragments that are closest to the light into a frame buffer object (FBO) with a depth texture. (Since we aren't drawing the fragments at this point, our FBO doesn't require color information, just depth.) OpenGL depth testing does exactly what we want here. Since OpenGL will automatically track the depth of the closest fragments, that gives us what we need – the first fragment at each point in the light space that's closest to the light source.

Fragments farther away will be occluded by closer fragments and should display in shadow.

The light's point-of-view rendering is our shadow map and only preserves the depth for each fragment. Darker colors are closer to the camera while lighter colors are farther away. From the first screenshot's perspective, the light source (the "sun") is outside the upper-right corner of the viewport. That means that the toy plane should be closer to the light and hence, darker/closer in the shadow map – and that's exactly what we see in the shadow map information.

Once the first pass has generated our shadow map, we go to the second rendering pass to draw our objects. We revert our viewport back to the normal camera and draw all our objects per usual. However, we can now incorporate the shadow map information to determine if the fragments are in light or shadow as we draw them.

How we use the shadow map to identify if our fragments are lit requires a bit of matrix math – we have to remap the fragment's location back to its position in light space. Once we have a light space coordinate for the fragment, we refer back to the shadow map and compare the fragment's depth with the depth stored in the shadow map. If the fragment depth is greater (farther away) then the fragment is hidden by a closer object and will be in shadow. Otherwise, the fragment is the closest object to the light and it should be lit.

With that understanding of the shadow mapping technique, let's look at the code that implements it.

**Setting up the scene**

Before we worry about the details of configuring our shadow renderer, let's create the scene with all the necessary objects.
Classes needed for the shadow rendering example are in main.qml: camera, controller, light, and rendering effects, as well as knot, plane, and ground entities.

Code sample 6: main.qml

```qml
import QtQuick 2.1 as QQ2
import Qt3D.Core 2.0
import Qt3D.Render 2.0
import Qt3D.Input 2.0
import Qt3D.Extras 2.0

Entity { id: sceneRoot

Camera { id: camera
    projectionType: CameraLens.
    PerspectiveProjection
    fieldOfView: 45
    aspectRatio: _window.width / _window.height
    nearPlane: 0.1
    farPlane: 1000.0
    position: Qt.vector3d(0.0, 10.0, 20.0)
    viewCenter: Qt.vector3d(0.0, 0.0, 0.0)
    upVector: Qt.vector3d(0.0, 1.0, 0.0)
}

FirstPersonCameraController { camera: camera }

ShadowMapLight { id: light
    components: [
        ShadowMapFrameGraph { id: framegraph
            viewCamera: camera
            lightCamera: light.lightCamera
        },
        // Event Source will be set by
        // the Qt3DQuickWindow
        InputSettings { }
    ]

AdsEffect { id: shadowMapEffect
    shadowTexture: framegraph.shadowTexture
    light: light
}

// Trefoil knot entity
Trefoil { material: AdsMaterial {
    effect: shadowMapEffect
    specularColor: Qt.rgba(0.5, 0.5, 0.5, 1.0)
}

// Toyplane entity
Toyplane { material: AdsMaterial {
    effect: shadowMapEffect
    diffuseColor: Qt.rgba(0.9, 0.5, 0.3, 1.0)
    shininess: 75
}

// Plane entity
GroundPlane { material: AdsMaterial {
    effect: shadowMapEffect
    diffuseColor: Qt.rgba(0.2, 0.5, 0.3, 1.0)
    specularColor: Qt.rgba(0, 0, 0, 1.0)
}
```

1. The first component we create is a Camera, which represents the camera used for the final rendering.
2. We create a FirstPersonCameraController element so we can control the camera using a keyboard or mouse.
3. Our light is created with a ShadowMapLight entity, which represents our light sitting somewhere above the plane and looking down at the scene's origin.
4. We tie the light to our custom frame graph ShadowMapFrameGraph.
5. We set the rendering AdsEffect that is responsible for doing the shadow rendering.

Finally, we create entities for the objects in the scene: a trefoil knot 1, a toy aircraft 2, and the 2D plane representing the green ground 3.
Setting the light

Now we need to define the light source.

**Code sample 7: ShadowMapLight.qml**

```qml
import Qt3D.Core 2.0
import Qt3D.Render 2.0

Entity {
    id: root
    property vector3d lightPosition: Qt.vector3d(30.0, 30.0, 0.0)
    property vector3d lightIntensity: Qt.vector3d(1.0, 1.0, 1.0)
    readonly property Camera lightCamera:
        lightCamera
    readonly property matrix4x4 lightViewProjection:
        lightCamera.projectionMatrix.
times(lightCamera.viewMatrix)
    Camera {
        id: lightCamera
        objectName: "lightCameraLens"
        projectionType: CameraLens. PerspectiveProjection
        fieldOfView: 45
        aspectRatio: 1
        nearPlane : 0.1
        farPlane : 200.0
        position: root.lightPosition
        viewCenter: Qt.vector3d(0.0, 0.0, 0.0)
        upVector: Qt.vector3d(0.0, 1.0, 0.0)
    }
}
```

The light is a directional spotlight. Since in the first rendering pass we also need to use the light as a camera, we place a Camera sub-entity inside of it and expose it as a property. The light also exposes properties for position, color/intensity, and a transformation matrix. The transformation matrix is used to allow us to map fragments back to light space coordinates so we can test our shadow map during the second rendering pass.

Creating the frame graph

With the scene's basic objects in place, it's time to create the frame graph that will configure our renderer for shading.

**Code sample 8: ShadowMapFrameGraph.qml**

```qml
import QtQuick 2.2 as QQ2
import Qt3D.Core 2.0
import Qt3D.Render 2.0

RenderSettings {
    id: root
    property alias viewCamera:
        viewCameraSelector.camera
    property alias lightCamera:
        lightCameraSelector.camera
    readonly property Texture2D shadowTexture: depthTexture
    activeFrameGraph: Viewport {
        normalizedRect: Qt.rect(0.0, 0.0, 1.0, 1.0)
        RenderSurfaceSelector {
            RenderPassFilter {
                matchAny: [ FilterKey { name: "pass"; value: "shadowmap" } ]
            }
            RenderTargetSelector {
                target: RenderTarget {
                    attachments: [
                        RenderTargetOutput {
                            objectName: "depth"
                            attachmentPoint: RenderTargetOutput.Depth
                            texture: Texture2D {
                                id: depthTexture
                                width: 1024
                                height: 1024
                                format: Texture. DepthFormat
                                generateMipMaps: false
                                magnificationFilter: Texture.Linear
                                minificationFilter: Texture.Linear
                            }
                        }
                    ]
                }
            }
        }
    }
}
```

Because the light view is also needed as a camera for the shadow renderer's first pass, we add and expose a Camera property to it.
Filter entities allow you to selectively enable or disable portions of the FrameGraph tree using selector entities.

```json
wrapMode {
  x: WrapMode.ClampToEdge
  y: WrapMode.ClampToEdge
}
comparisonFunction:
  Texture.CompareLessEqual
comparisonMode: Texture.CompareRefToTexture
}
}
}
]
ClearBuffers {
  buffers: ClearBuffers.
  DepthBuffer
  CameraSelector {
    id: lightCameraSelector
  }
}
}
}

RenderPassFilter {
  matchAny: [ FilterKey { name: "pass"; value: "forward" } ]
ClearBuffers {
  clearColor: Qt.rgba(0.0, 0.4, 0.7, 1.0)
  buffers: ClearBuffers.
  ColorDepthBuffer
  CameraSelector {
    id: viewCameraSelector
  }
}
}
}

We have two paths from the topmost Viewport entity, and each path corresponds to a pass of the shadow map technique.

One aspect of frame graphs that we’ve used in earlier examples but haven’t yet addressed in detail is filters. A filter entity allows you to selectively enable or disable portions of the frame graph tree using selector entities. In our case, the two alternate paths for each pass of the shadow renderer are enabled and disabled using a RenderPassFilter. Our render pass filter will filter based on a string we’ve defined: in our case the string “pass”. This lets us modify the frame graph configuration for each pass appropriately. Other lines of interest to highlight:

1 In the first pass to generate the shadow map, we need to render to an off-screen surface (our FBO) with a depth texture attachment.
2 The Texture2D entity configures the texture storage that we use to keep the shadow map.
3 We also need to render using the light’s camera, so we set the CameraSelector entity to the camera that our light exports.

The second pass is much more straightforward: we simply render to the screen using the main camera.
The effect

The bulk of the work happens in the AdsEffect.qml file, where our main Effect entity is defined. The effect implemented is using the ADS shading model (ambient diffuse specular, in other words, Phong shading) with our shadow-map generated shadows.

Code sample 9: AdsEffect.qml

```qml
import Qt3D.Core 2.0
import Qt3D.Render 2.0

Effect {
    id: root

    property Texture2D shadowTexture
    property ShadowMapLight light

    // These parameters act as default values
    // for the effect. They take priority
    // over any parameters specified in the
    // RenderPasses below (none provided in
    // this example). In turn these parameters
    // can be overwritten by specifying them
    // in a Material that references this
    // effect. The priority order is:
    // Material -> Effect -> Technique ->
    // RenderPass -> GLSL default values

    parameters: [
        Parameter { name: "lightViewProjection"; value: root.light.lightViewProjection },
        Parameter { name: "lightPosition"; value: root.light.lightPosition },
        Parameter { name: "lightIntensity"; value: root.light.lightIntensity },
        Parameter { name: "shadowMapTexture"; value: root.shadowTexture }
    ]

    techniques: [
        Technique {
            graphicsApiFilter {
                api: GraphicsApiFilter.OpenGL
                profile: GraphicsApiFilter.CoreProfile
            }
        }
    ]

    renderPasses: [
        RenderPass {
            filterKeys: [ FilterKey { name: "pass"; value: "shadowmap" } ]
            shaderProgram: ShaderProgram {
                vertexShaderCode: loadSource("qrc:/shaders/shadowmap.vert")
                fragmentShaderCode: loadSource("qrc:/shaders/shadowmap.frag")
            }
            renderStates: [
                PolygonOffset { scaleFactor: 4; depthSteps: 4 },
                DepthTest { depthFunction: DepthTest.Less }
            ],
        }
        RenderPass {
            filterKeys: [ FilterKey { name: "pass"; value: "forward" } ]
            shaderProgram: ShaderProgram {
                vertexShaderCode: loadSource("qrc:/shaders/ads.vert")
                fragmentShaderCode: loadSource("qrc:/shaders/ads.frag")
            }
        }
        RenderPass {
            filterKeys: [ FilterKey { name: "pass"; value: "shadowmap" } ]
            shaderProgram: ShaderProgram {
                vertexShaderCode: loadSource("qrc:/shaders/ads.vert")
                fragmentShaderCode: loadSource("qrc:/shaders/ads.frag")
            }
            renderStates: [
                PolygonOffset { scaleFactor: 4; depthSteps: 4 },
                DepthTest { depthFunction: DepthTest.Less }
            ],
        }
        Technique {
            graphicsApiFilter {
                api: GraphicsApiFilter.OpenGL
                profile: GraphicsApiFilter.CoreProfile
            }
        }
    ]
}
```

The Effect entity is implemented using Phong shading with the ambient, diffuse, and specular components of light.
You can customize shaders by using multiple Technique elements, for example OpenGL versus OpenGL ES or different versions of OpenGL.

```cpp
shaderProgram: ShaderProgram {
    vertexShaderCode: loadSource("qrc:/shaders/es3/shadowmap.vert")
    fragmentShaderCode: loadSource("qrc:/shaders/es3/shadowmap.frag")
}
renderStates: [
    PolygonOffset { scaleFactor: 4;
    depthSteps: 4 },
    DepthTest { depthFunction: DepthTest.Less }
],
RenderPass {
    filterKeys: [FilterKey { name: "pass"; value: "forward" }]
}
}
}
```

➊ The parameters list defines default values for the effect. Those values are mapped to OpenGL shader program uniforms so we can access them within the shaders. What's needed in the shaders is information from the Light entity (position, intensity, and its view/projection matrix), as well as the shadow map texture exposed by the frame graph.

➋ In order to be able to adapt the implementation to different hardware or OpenGL versions, an Effect is implemented by providing one or more Technique elements. In our case, we provide two techniques, one for OpenGL 3.2 Core or greater and one for OpenGL ES 3.0 or greater. (The two techniques allow us to use customized shaders for each platform since the GLSL variant is slightly different between those two OpenGL versions.)

➌ Inside each technique we define our two rendering passes. We set the filter values for each so they're associated with our frame graph rendering passes.

➍ For the shadow map pass, we load GLSL shaders that very simply project the mesh coordinates from model space into clip space. The fragment shader is empty because we aren't generating color information and OpenGL will automatically capture the depth. We also set some custom OpenGL states for polygon offset (used to help prevent fragments from Z-fighting) and depth testing (to make sure we're properly sorting objects front to back).

➎ The second pass is instead a normal forward renderer using Phong shading. The code really just loads the fragment and vertex shaders that will be used when drawing.

**The shaders**

The full explanation of the shader code requires some fluency in GLSL shader language and is out of scope of this whitepaper. However, we will point out the key parts that are necessary to make our shadow mapper work.

**Code sample 10: ads.vert**

```cpp
#version 150 core
in vec3 vertexPosition;
in vec3 vertexNormal;
out vec4 positionInLightSpace;
out vec3 position;
out vec3 normal;
uniform mat4 lightViewProjection;
uniform mat4 modelMatrix;
uniform mat4 modelView;
uniform mat3 modelViewNormal;
uniform mat4 mvp;
void main() {
```

KDAB | the Qt, OpenGL and C++ experts

14
Converting from normalized device coordinates to texture map coordinates allows for easy shadow map value lookups within the fragment shader.

```cpp
const mat4 shadowMatrix = 
mat4( 0.5, 0.0, 0.0, 0.0, 
0.0, 0.5, 0.0, 0.0, 
0.0, 0.0, 0.5, 0.0, 
0.5, 0.5, 0.5, 1.0);

positionInLightSpace = shadowMatrix * 
lightViewProjection * modelMatrix * 
vec4(vertexPosition, 1.0);

normal = normalize(modelViewNormal * 
vertexNormal); 
position = vec3(modelView * 
vec4(vertexPosition, 1.0));
gl_Position = mvp * vec4(vertexPosition, 
1.0);
```

Here is where we compute the coordinates of each vertex in light space. We have to adjust the coordinates slightly by multiplying with `shadowMatrix`. This lets us convert from normalized device coordinates (ranging between -1 and 1) to texture map coordinates (between 0 and 1) so we can easily look up values in our shadow map during the fragment shader. On to the fragment shader...

**Code sample 11: ads.frag**

```cpp
#version 150 core

uniform mat4 viewMatrix;
uniform vec3 lightPosition;
uniform vec3 lightIntensity;
uniform vec3 ka; // Ambient reflectivity
uniform vec3 kd; // Diffuse reflectivity
uniform vec3 ks; // Specular reflectivity
uniform float shininess;
// Specular shininess factor

uniform sampler2DShadow shadowMapTexture;
in vec4 positionInLightSpace;
in vec3 position;
in vec3 normal;
out vec4 fragColor;

vec3 dsModel(const in vec3 pos, const in vec3 n) { 
  // Calculate the vector from the light 
  // to the fragment 
  vec3 s = normalize(vec3(viewMatrix * 
    vec4(lightPosition, 1.0)) - pos);
  // Calculate the vector from the fragment 
  // to the eye position (origin since this 
  // is in "eye" or "camera" space)
  vec3 v = normalize(-pos);
  // Reflect the light beam using the 
  // normal at this fragment 
  vec3 r = reflect(-s, n);

  // Calculate the diffuse component
  float diffuse = max(dot(s, n), 0.0);
  // Calculate the specular component
  float specular = 0.0;
  if (dot(s, n) > 0.0)
    specular = pow(max(dot(r, v), 0.0), shininess);
  // Combine the diffuse and specular 
  // contributions (ambient is taken 
  // into account by the caller)
  return lightIntensity * (kd * diffuse 
    + ks * specular);
}

void main() {
  float shadowMapSample = 
    textureProj(shadowMapTexture, 
        positionInLightSpace);
  vec3 ambient = lightIntensity * ka;
  vec3 result = ambient;
  if (shadowMapSample > 0) 
    result += dsModel(position, 
      normalize(normal));
  fragColor = vec4(result, 1.0);
}
```

The job of the fragment shader is conceptually easy – map the current fragment back to the shadow map, look up what’s in the shadow...
The flexibility of the Qt 3D renderer allows you to implement many sophisticated effects.

map at that position, and if it's positive the fragment is on a lit surface so add in our lighting component. That's exactly what happens here: we sample the shadow map ➊, and if it's lit ➋, we add in our light component ➌ using our Phong lighting model. ➍ Otherwise if it's not lit, the only lighting comes from our ambient source.

Although there's a decent amount of code in this example to pull it off, this example shows just how flexible and configurable the Qt 3D rendering engine is.

Summary
The flexibility of the Qt 3D renderer allows you to pretty easily experiment with a variety of sophisticated effects. With this introduction to frame graphs, frame graph nodes, and how the frame graph tree operates, you'll have some idea about how the Qt 3D engine uses frame graph and its assorted components to configure renderers – and hopefully some idea of how to go about building your own. However, if you get stuck trying to implement some amazing-looking new graphical technique, we're always happy to help!

About the KDAB Group
The KDAB Group is the world's leading software consultancy for architecture, development and design of Qt, C++ and OpenGL applications across desktop, embedded and mobile platforms. KDAB is the biggest independent contributor to Qt and is the world's first ISO 9001 certified Qt consulting and development company. Our experts build run-times, mix native and web technologies, solve hardware stack performance issues and porting problems for hundreds of customers, many among the Fortune 500. KDAB's tools and extensive experience in creating, debugging, profiling and porting complex applications help developers worldwide to deliver successful projects.

KDAB's trainers, all full-time developers, provide market leading, hands-on, training for Qt, OpenGL and modern C++ in multiple languages. Founded in 1999, KDAB has offices throughout North America and Europe.

www.kdab.com

© 2019 the KDAB Group. KDAB is a registered trademark of the KDAB Group. All other trademarks belong to their respective owners.