KDAB

Qt 3D Basics

Paul Lemire

September 18

Learning to create 3D applications can be an overwhelming task even for seasoned developers because of all the new terminology, visual concepts, and advanced math. To simplify the job, many developers use a framework like Qt 3D.

Qt offers developers everything they need to make a cross-platform app with all the bells and whistles, while the 3D portion helps take care of all the graphical rendering details. Make no mistake, it's still a complex technology, but it's a great choice for easily creating rich 3D applications.

This is the first in a series of whitepapers intended to help developers create highly polished Qt 3D applications, including descriptions of the technology components, the rationale behind their designs, and straightforward examples.

This paper assumes that you already have experience with C++ and QML.

Qt 3D architectural decisions

To have a better understanding of how to use Qt 3D, it's important to examine the key requirements that drove its architecture:

- Be able to create, draw, and move 2D shapes and 3D meshes
- Handle a wide variety of visual techniques such as shadowing, occlusion, high-dynamic range, deferred rendering, and physical-based rendering
- Draw scenes in near real-time, with graphic performance that scales with the power of the GPU and available CPU cores

Qt 3D was designed using two fundamental principles: make it fast and make it flexible.

- Have an extensible framework that can handle other aspects of 3D objects like physics simulation, collision detection, positional audio, animation skeletons, path finding, particles, etc

Boiling down all of these demanding requirements leaves us with two fundamental principles: make it *fast* and make it *flexible*.

Let's get started with a high-level view of the frame graph and its related concept, the scene graph, both of which are necessary for building Qt 3D graphic images. Why are they part of Qt 3D? The scene graph is a data-driven description of *what to render* while the frame graph is a data-driven description of *how to render*. The frame graph allows developers to select a renderer, making it infinitely flexible yet as lightweight as possible for each application. Using a data-driven description in the frame graph allows developers to choose between using a simple forward renderer (including a z-fill pass), using a deferred renderer, describing how to render transparent or semi-transparent objects, and so on. And as the frame graph is data-driven, it's very easy to dynamically modify at runtime without a line of C++ code. However, it also allows you to implement any rendering algorithm – if Qt 3D doesn't provide what you need out of the box, you can always compose your own.

Qt 3D through Space Invaders

Let's explore the Qt 3D architecture by imagining we want to translate the old arcade classic Space Invaders into Qt 3D. This example is simple enough to easily understand, yet contains enough complexity to help us explore the necessary design concepts.

Essential elements in the Space Invaders game

We begin by enumerating some typical object types that might be found in an implementation of Space Invaders:

- Player's ground cannon
- Defensive barriers
- Enemy spaceships
- Enemy boss flying saucer
- Bullets from both enemy ships and the player's ground cannon

In a traditional C++ design these objects would likely end up implemented as classes arranged in an inheritance tree. Various branches in the inheritance tree might add additional functionality to the root class for features such as "accepts user input", "plays a sound", "has animation", "requires collision detection", and so on. However, designing an elegant inheritance tree for even such a simple example is not easy. An object model based on inheritance has a number of issues, including:

Qt 3D uses an entity-component-system to avoid the problems of complex inheritance trees, which may dictate awkward class hierarchies.

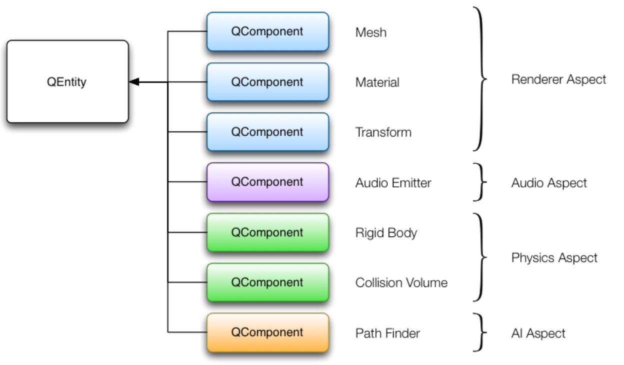
- Deep and wide inheritance hierarchies that are difficult to maintain and extend
- An inheritance taxonomy that is set in stone at compile time
- Class inheritance levels that can only classify upon a single criteria or axis
- Shared functionality that tends to "bubble up" throughout the class hierarchy over time
- The inevitability that library designers will never know all the things library users want to do

If you've worked with an inheritance tree of any size or complexity, you'll know that modifying one can be a huge hassle. Not only do you have to understand the original taxonomy, your proposed changes need to fit cleanly within it – which they often do not. The result is that the project can devolve into the original class structure plus a mess of ugly hacks on top.

To avoid these problems, Qt 3D uses an Entity Component System (ECS) to impart functionality to an instance of an object through aggregation. An entity represents an object that is devoid of any specific behaviour or characteristics. Behaviours are described in one or more QComponents, which are then aggregated to the entity.

What would that look like in our Space Invaders example? An enemy spaceship would be represented as a **QEntity** with several attached **QComponents** to provide the entity's behaviour: render, emit sound, detect collisions, and attack. The player's ground cannon would have similar components to the enemy space invader, except instead of the attack component it would have a component that accepts player input, allowing it to move side to side and fire bullets.

The entire game would be represented with a single scene graph that represents all of the objects as **QEntities** (enemy invaders, player's ground cannon, shields, bullets, etc).



Example QEntity/QComponent structure

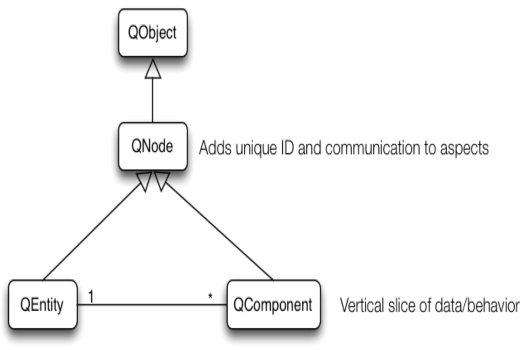
A key part of the Qt 3D ECS paradigm is Aspects. Objects that require a specific type of functionality belong to an aspect, and each aspect is registered with the various components that provide its specific behaviour. On every display frame, aspects are asked for a set of tasks to execute – which includes any dependent tasks. The aspects find all related components and execute their behaviour.

As this is getting a bit convoluted, some examples may help. **QEntities** that need to draw themselves must have **QComponents** with a rendering aspect. A renderer aspect looks for **QEntities** that have **Mesh**, **Material**, or **Transformation** components – components that require displaying. Similarly, a physics simulation aspect looks for entities that have a collision volume, mass, coefficient of friction, etc. An audio aspect finds entities with components that need to emit sounds. And so on.

A nice feature of Qt 3D's ECS is that we can dynamically change how an object behaves at runtime simply by adding or removing components.

Dynamic behaviour with the ECS

A nice feature of Qt 3D's ECS is that, because it uses aggregation rather than inheritance, we can dynamically change how an object behaves at runtime simply by adding or removing components. Want your cannon to be invincible to enemy bullets after shooting a power-up? No problem. Just temporarily remove the entity's collision volume component and, when the power-up times out, add the collision volume back in again. There is no need to make a special one-off subclass.

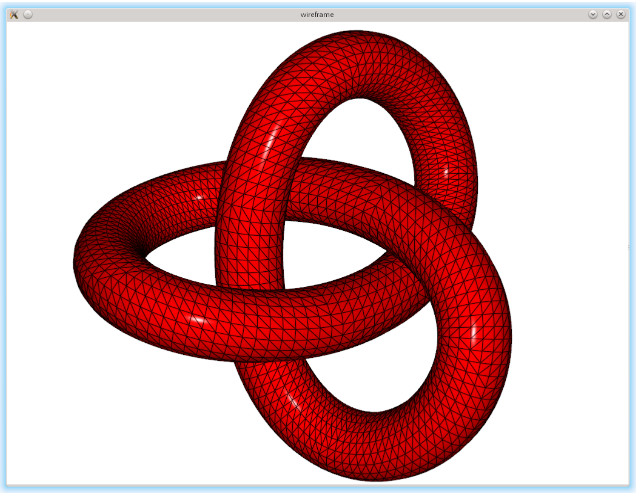

QObject
QNode
Adds unique ID and communication to aspects
QEntity
1
*
QComponent
Vertical slice of data/behavior

Relationship of QEntity and QComponent

QEntity aggregates zero or more **QComponents** to define the object's behaviour and, as we've described, can be dynamically changed. But how do you create custom behaviours? Simple – by creating a new aspect (or extending an existing one) to add the methods necessary for the corresponding components to do their work, and then adding the data needed to drive the aspect's behaviour to the component. For example, a renderer aspect utilizes a **Mesh** component to retrieve the per-vertex data that should be sent down the OpenGL pipeline.

An example

To give you a concrete example of how to draw something in Qt 3D using the QML API, we'll draw a single entity – a trefoil knot. To make it slightly more interesting, we'll use a custom set of shaders that implement a single-pass wireframe rendering method. This example is contained in the Qt Example projects. To follow along, load up the Qt IDE, go to **Welcome | Examples**, and search for **Qt 3D: Wireframe QML Example**.



Example trefoil knot

Let's start with the code, followed by an explanation of what each chunk does.

Qt Creator provides lots of Qt 3D examples like the wireframe trefoil knot used here that cover many aspects of application development.

We use Entity as the root element of our custom Trefoil knot type, exposing our custom properties just as you would with any other type in QML.

Code sample 1: TrefoilKnot.qml

```
import Qt3D.Core 2.0                       A
import Qt3D.Render 2.0

Entity {
    id: root

    property real x: 0.0
    property real y: 0.0
    property real z: 0.0                        B
    property real scale: 1.0
    property real theta: 0.0
    property real phi: 0.0
    property Material material

    components: [ transform, mesh,            C
                  root.material ]

    Transform {
        id: transform
        translation:
            Qt.vector3d(root.x,
                  root.y,root.z)
        rotation:
            fromEulerAngles(theta,
                  phi, 0)                         D
        scale: root.scale
    }

    Mesh {
        id: mesh
        source:
            "assets/obj/trefoil.obj"
    }
}
```

A. What's going on here? We start off by importing the `Qt3D.Core 2.0` module that provides the `Entity` type and value type helpers like `Qt.vector3d()`. Similarly, we also import the `Qt3D.Render 2.0` for the renderer aspect. (If we were using components from other aspects, we would also need to import their corresponding QML module here too.)

B. We use `Entity` as the root element of our custom Trefoil knot type, exposing our custom properties (`x`, `y`, `scale`, `theta`, `phi`, and `material`) just as you would with any other type in QML. (You'll note that in QML, we use **Entity** and **Component** – rest assured that these refer to the same classes as their underlying C++ equivalents, **QEntity** and **QComponent**.)

D. In addition to aggregating components, **Entity** objects can be used to group child objects together (just as Qt Quick 2 uses the *Item* object), such as the `Transform` and `Mesh` components. The `Mesh` component uses its source property to load in a static set of geometry (such as vertex positions, normal vectors, and texture coordinates,) from a file in the Wavefront Obj format. (This data was exported from the excellent and free Blender application.) The `Transform` component specifies how the renderer should transform the geometry when it is drawn with the OpenGL pipeline. (In addition to creating objects through a `Mesh` element, Qt 3D also allows dynamic generation of per-vertex attribute data through C++ hooks called by the task-based engine.)

C. But we skipped over something – the `components` property, what is that? Simply instantiating **Components** is not enough to allow them to have special behaviours. The **Entity** must aggregate the **Components** using its components property, which defines the subcomponents and allows them to be shared between multiple entities. So we take our transform and mesh components, and list them under components so they are accessible. We also list a component of type `root.Material` that we haven't defined within TrefoilKnot.qml – this is a property that allows users to easily customise the appearance of the entity, something that we will make use of shortly.

Now that we have defined a custom **Entity**, here's how to use it to actually get our desired result.

The Qt 3D material system allows multiple rendering passes with different state sets, provides mechanisms for overriding parameters at different levels, and easily switching shaders.

Code sample 2: main.qml

```
import Qt3D.Core 2.0
import Qt3D.Render 2.0
import Qt3D.Input 2.0
import Qt3D.Extras 2.0

Entity {
    id: root

    // Render from the mainCamera
    components: [
        RenderSettings {
            activeFrameGraph:
ForwardRenderer {
                id: renderer
                camera: mainCamera                A
            }
        },
        // Event Source will be set
        // by the Qt3DQuickWindow
        InputSettings { }
    ]

    BasicCamera {
        id: mainCamera
        position:                                  B
          Qt.vector3d(0.0,0.0,15.0)
    }

    FirstPersonCameraController { camera:
mainCamera }

    WireframeMaterial {
        id: wireframeMaterial
        effect: WireframeEffect {}
        ambient:                                  C
          Qt.rgba(0.2,0.0,0.0,1.0)
        diffuse:
          Qt.rgba(0.8,0.0,0.0,1.0)
    }

    TrefoilKnot {
        id: trefoilKnot
        material: wireframeMaterial
    }
}
```

We start off again with some import statements and the same overall structure as in Code sample 1, adding a couple for `Input` and `Extras`. We also again use `Entity` as a root element container.

A. The **FrameGraph** component uses a `ForwardRenderer` to completely configure the renderer without touching any C++ code at all. There's a lot more specialty rendering that you can do with a **FrameGraph**, but we'll save that for future whitepapers.

B. The `BasicCamera` element is a trivial wrapper around the built-in **Camera**. The **Camera** represents a virtual camera with properties for things like near and far planes, field of view, aspect ratio, projection type, position, orientation, and more. Here, we wrap it just to allow us to set the initial position.

C. Next up we have the `WireframeMaterial` element, a custom type that wraps up the built-in `Material` type and sets the effect to `WireFrameEffect`. A built-in wireframe rendering effect is handy and an example of Qt 3D's innate flexibility. The Qt 3D material system can handle different rendering approaches, different platforms, and different OpenGL versions. This allows for multiple rendering passes with different state sets, provides mechanisms for overriding parameters at different levels, and easily switches shaders — all from either C++ or QML property bindings. Qt 3D also supports all of the OpenGL programmable rendering pipeline stages: Vertex, tessellation control and evaluation, geometry, fragment, and compute shaders.

Qt 3D objects can be easily animated in position, colour, style, material, etc with Qt Quick 2 animations – all within QML with no C++ code.

Displaying – and animating – our knot

Instantiating the `TrefoilKnot` and setting its material is simplicity itself with a couple of lines at the bottom of main.qml. Once we do that, the Qt 3D engine in conjunction with the renderer aspect has enough information to finally render our mesh using the material we specified.

We can easily make things a little more interesting by adding some Qt Quick 2 animations to Code sample 2 – as highlighted in the blue boxes below.

Code sample 3: main.qml, with animation code added

```
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: root

    // Render from the mainCamera
    components: [
        RenderSettings {
            activeFrameGraph:
ForwardRenderer {
                id: renderer
                camera: mainCamera
            }
        },
        // Event Source will be set by the
Qt3DQuickWindow
        InputSettings { }
    ]

    BasicCamera {
        id: mainCamera
        position:
            Qt.vector3d( 0.0, 0.0, 15.0 )
    }

    FirstPersonCameraController { camera:
mainCamera }

    WireframeMaterial {
        id: wireframeMaterial
        effect: WireframeEffect {}
        ambient:
            Qt.rgba( 0.2, 0.0, 0.0, 1.0 )
        diffuse:
            Qt.rgba( 0.8, 0.0, 0.0, 1.0 )

        QQ2.SequentialAnimation {
            loops: QQ2.Animation.Infinite
            running: true

            QQ2.NumberAnimation {
                target: wireframeMaterial;
                property: "lineWidth";
                duration: 1000;
                from: 0.8
                to: 1.8
            }

            QQ2.NumberAnimation {
                target: wireframeMaterial;
                property: "lineWidth";
                duration: 1000;
                from: 1.8
                to: 0.8
            }

            QQ2.PauseAnimation {
                 duration: 1500
            }
        }
    }

    TrefoilKnot {
        id: trefoilKnot
        material: wireframeMaterial
    }
}
```

This time we also added in a namespace import for the `Qt Quick 2.1` module.

The `QQ2.SequentialAnimation` block takes care of the entire 3D animation – using what are otherwise standard QML animation techniques to make property updates directly to the `wireFrameMaterial`. The property

Qt 3D lets developers focus on app-specific details instead of the myriad of OpenGL structures, math, and calls needed to get things working.

updates are noticed by the **QNode** base class and are automatically sent through to the corresponding objects in the renderer aspect. The renderer then takes care of translating the property updates through to new values for uniform variables in the GLSL shader programs.

Our animation simply pulses the width of the wireframe lines – but all the heavy lifting is done by the GPU. We'll show Qt 3D key frame based animations in a later paper.

Summary

In this whitepaper, we've looked at the basic structure of Qt 3D and created a simple application. The app hides a huge amount of the complexity that would be needed for a raw OpenGL/C++ app. Qt 3D lets us focus on our app-specific details instead of the myriad of OpenGL structures, math, and calls needed to get things working. Future whitepapers will start to dive in deeper, looking at some specific techniques developers can use to create beautifully rendered 3D scenes, and peeking underneath the hood a bit more for Qt 3D.

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.

KDAB

CERTIFIED ISO 9001:2015 COMPANY

www.kdab.com