Qt 3D - A Data-Driven Renderer for Mortals

Sean Harmer*
Director (UK), KDAB

Figure 1: *Qt 3D rendering a Dodge Viper with a PBR pipeline and showing a 2D Qt Quick 2 UI as a blended overlay*

Abstract

Qt 3D is a soft real time simulation framework that provides visualization by way of a data-driven renderer. Qt 3D is part of the Qt cross platform development framework making it easy to integrate into your own code base. Qt 3D makes adding 3D content to your engineering, content creation, simulation, or business applications as simple as possible.

The renderer of Qt 3D is configurable in a data driven manner by way of a frame graph data structure. The frame graph encodes the rendering algorithm that should be applied and can be freely altered at runtime to allow dynamic adaptations for changing scene types or performance requirements.

In addition to a highly configurable renderer Qt 3D is a fully extensible framework that allows users to write their own subsystems providing new functionality (e.g. collision, positional audio, physics). Such subsystems can easily take advantage of Qt 3D's highly threaded architecture. Qt 3D provides intuitive APIs for both C++ and QML, a declarative language based on JavaScript, and can be trivially integrated with Qt Quick 2 to provide fluid 2D user interfaces.

Keywords: real time, data driven, framework, cross platform

Concepts: •**Computing methodologies** → **Graphics processors;**

1 Where does Qt 3D fit?

Qt has long provided a solid foundation for creating graphical user interfaces and other parts of application logic across multiple platforms. A long term omission from the repertoire of Qt has been a high level interface for adding 3D graphical content. Qt 3D adds support for cross platform 3D graphics capabilities to Qt via both C++ and QML APIs. QML is a very expressive, JavaScript-based, declarative language that allows binding object properties together in a very natural way. For example, configuring a camera for rendering that alters properties along with the screen size and is animated along a path is as simple as:

```
Entity {
  property real screenWidth
  property real screenHeight
  property alias progress: path.progress
  property bool explodedView: false
  property bool idleAnimationRunning: false

  Camera {
    projectionType: CameraLens.PerspectiveProjection
    fieldOfView: explodedView ? 55 : 30
    Behavior on fieldOfView {
      NumberAnimation {
        duration: 750
        easing.type: Easing.OutQuad
      }
    }

    aspectRatio: screenWidth / screenHeight
    nearPlane: 0.01
    farPlane: 1000.0
    viewCenter: Qt.vector3d(0.0, -0.5, 0.0)
    upVector: Qt.vector3d(0.0, 1.0, 0.0)
    position: {
      if (idleAnimationRunning)
        return Qt.vector3d(path.y, 2.0, path.x)
      else
        return Qt.vector3d(5.5, 2.0, 5.5)
    }
  }

  PathInterpolator { id: path; ... }
}
```

The `Behavior` element demonstrates how easy it is to animate properties using the Qt Quick 2 animation system and how properties can naturally be related to each other via property bindings.

*e-mail:sean.harmer@kdab.com

SIGGRAPH '16, July 24-28, 2016, Anaheim, CA,
ISBN: 978-1-4503-4282-7/16/07
DOI: http://dx.doi.org/10.1145/2897839.2927459

Qt 3D aims to provide a high-level framework for real time 3D graphics that can be used for all manner of applications including simulation, content creation, modelling, engineering, architectural visualization, data visualization, games, automotive, oil and gas, and aviation.

Building on the strengths of Qt's established cross-platform abstractions and signal-slot, property and reflection systems, Qt 3D exposes modern 3D capabilities to a wide user base. Integrating Qt 3D with your custom business logic is trivial from C++ and QML.

2 Architecture of Qt 3D

Qt 3D is built on top of OpenGL (or OpenGL ES on mobile targets) and exposes the full capabilities of these via an easy to learn programming interface. This talk will introduce how to build a scene directly in Qt 3D using QML or C++; how to add custom materials with your own GLSL shaders implementing a PBR workflow; and interfacing with your C++ business logic and data models.

2.1 The Frame Graph

The renderer within Qt 3D is data-driven by way of a user-definable frame graph. This is parsed by the Qt 3D engine which uses the data to control which entities in your scene are rendered, which materials, techniques and render passes are used, which cameras and viewports to use and a raft of other options.

All of this can be changed dynamically at runtime to adapt the renderer to the time varying needs of your application. For example, to enable specialised rendering algorithms that use clip planes or stereoscopic configurations, or to switch between forward, deferred or forward+ approaches.

The abstraction of the frame graph allows developers to easily experiment with novel rendering algorithms without having to write low level C/C++ rendering code. The data oriented nature of the frame graph also lends itself well to being exposed to tooling in the future.

2.2 Aspects, Entities and Components

Qt 3D is based upon a highly threaded, Entity Component System (ECS) architecture. The renderer is just one aspect (albeit an important one). Qt 3D also offers a flexible input aspect that allows for easy customisation of input event handling and allows the system to be easily extended beyond the traditional mouse and keyboard input systems. For example, many CAD or content creation applications require the use of a 3D mouse. This is easily catered for within Qt 3D and if you need to add your own devices, that is fairly straight forward to do too.

The renderer and input aspects are independent of each other allowing either to be replaced or improved without disruption to the other. The framework for this and for allowing the user-visible frontend APIs to operate independently of the backend are quite involved and highly threaded.

We will introduce this architecture and explain how it is possible for users (and the Qt 3D authors) to extend the features in the future to include facilities for collision detection, AI, physics simulation, positional audio and more. The Qt 3D architecture also makes it a prime candidate for porting to more modern graphics APIs such as Vulkan, Direct3D 12 or Metal.

The object model of Qt 3D is built on top of Qt's established QObject systems. As a result, it is possible to use a variety of tools such as GammaRay for introspecting and debugging Qt 3D applications. This allows debugging of the scene graph, frame graph, state machines and many other subsystems.

The combination of the ECS architecture and data driven frame graph, allows the user to build scenes and control the rendering in ways that the previous generation scene graphs cannot easily match without deep and intrusive changes.

2.3 Integration with Qt

Qt 3D also interfaces very well with Qt Quick 2, an established technology stack for creating 2.5D user interfaces. Qt Quick 2 is based upon the same QML language and runtime as Qt 3D. This allows for powerful and versatile user interfaces to easily be combined with your 3D content, including overlays and underlays. Qt 3D can also be combined with more traditional widget based 2D user interfaces as shown in Fig. 2. KDAB has first-hand experience of doing this across a number of sectors including gaming, architectural visualisation, engineering, automotive, oil and gas and aviation.

Qt and Qt 3D also have attractive licensing options. They are available under both LGPLv3 and Commercial licenses without royalties making it suitable for use in both commercial and open source projects.

The entire Qt project is operated as an Open Governance project with peer review. If you need a feature adding, a bug fix, or some part optimising, you are more than welcome to participate and contribute code, documentation, examples or assets.

Figure 2: *Qt 3D integrated in a desktop application*

Acknowledgements

I would like to thank all those who have contributed directly, or indirectly to Qt 3D and to Qt in general and for making the Qt Project such a pleasant place to contribute. In particular, full and hearty appreciation to Paul Lemire, Kevin Ottens, James Turner, Mauro Persano, Giuseppe D'Angelo, Volker Krause, Marc Mutz and all the others that have contributed or been there to bounce ideas off. Thank you to my employer, KDAB, for indulging my fascination with all things 3D and allowing me to work on fun projects and travel the world.

References

KRAUSE, V., WINTER, A., WOLFF, M., FUNK, K., AND KREUZKAMP, A., 2016. Gammaray - Qt introspection and debugger tool. https://github.com/KDAB/GammaRay.

THE QT PROJECT, 2016. Qt cross platform toolkit. http://www.qt.io.