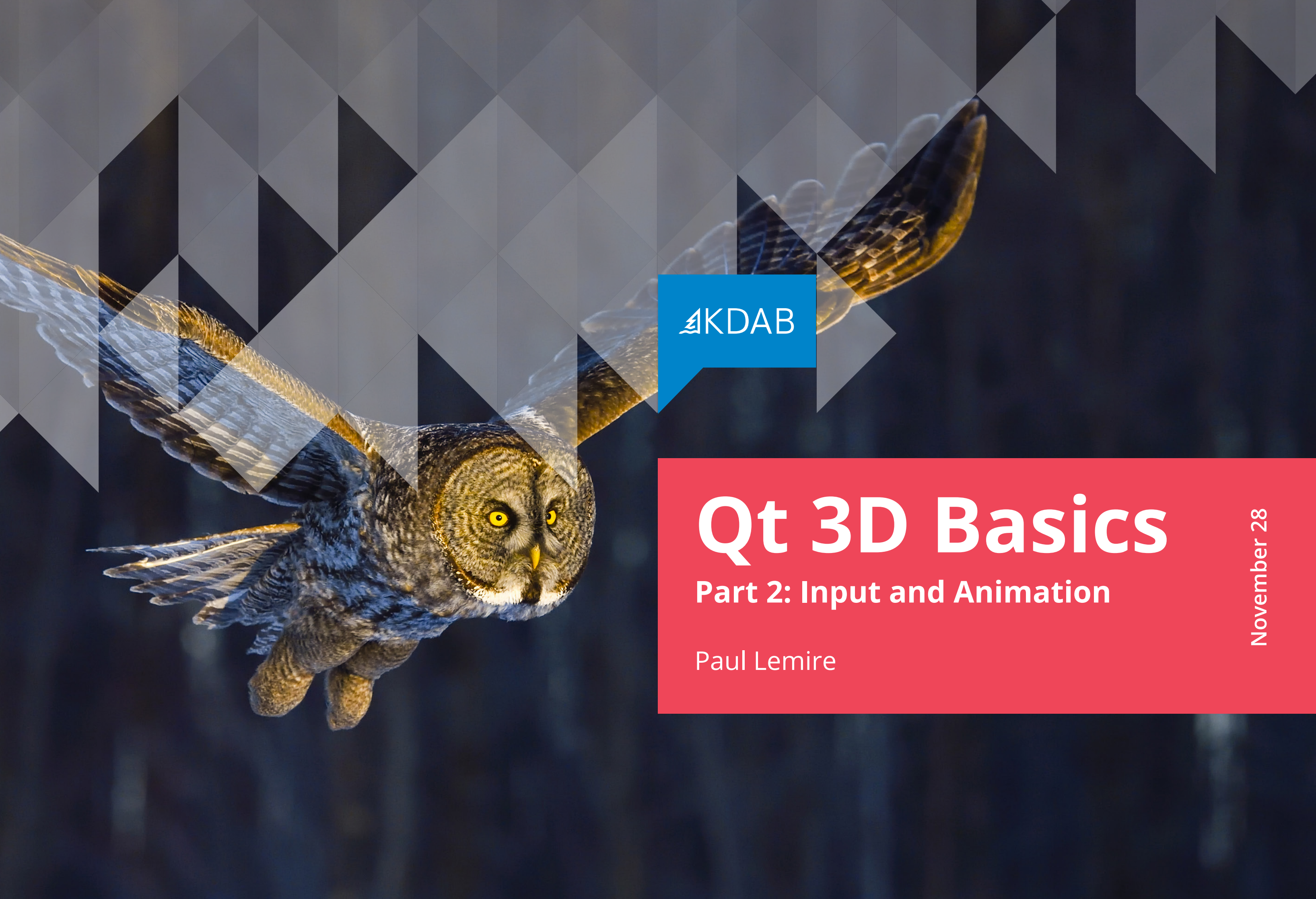


Qt 3D Basics

Part 2: Input and Animation

Paul Lemire

November 28

The ability to create 3D applications is a skillset that is increasingly in demand as it plays a crucial role in advanced UX design, virtual reality, game development, and more. While developing a modern 3D program requires many disparate skills, there are frameworks that help the developer stitch it all together. Qt 3D is one that we recommend.

Our first Qt 3D whitepaper helped us understand the structure of a Qt 3D program – the nature of entities, components, and aspects, and how those are incorporated into the scene graph and frame graph. With those basics under our belt, we'll now tackle how to incorporate those elements into a functioning program by examining how to receive user input and how to animate objects in a scene.

Qt 3D input

Let's start with the most basic need for any non-trivial program: – user input. If your 3D program has a 2D overlay with buttons or controls, you can manage those widgets with the same APIs you would use for any standard 2D app. But how do you know where a click or touch occurs within the 3D scene?

That's done via the **QObjectPicker** class, which uses ray-cast picking. This technique traces the path of an imaginary ray from the screen

To see if the user is interacting with your 3D object, add a QObjectPicker to your QEntity and catch one of the available signals

pixel back into the 3D model. Qt 3D returns whatever object the ray intersects as a match. The algorithm is very similar to ray tracing in graphics rendering, except to find the object being pointed at we don't need to know the color, material, or lighting for the intersected point, we only need to know the object that the ray is intersecting.

To see if the user is interacting with your 3D object, add a **QObjectPicker** to your **QEntity** and catch one of the available signals. The signals **pressed(), released(),** and **clicked()** will be emitted when the program detects a click or part of a click. For a click-and-drag event, make sure that **dragEnabled** is true and that you're using the **moved()** signal. All of these signals provide a QPickEvent object that describes the intersection of the ray cast with the most common interpretations of the coordinate – either **position** (for screen space), **localIntersection** (for model space) or **worldIntersection** (for world space).

Using mice

The **pressed/released/clicked** signals in **QObjectPicker** naturally work with a mouse. But what if you want to see if the mouse is over your object? Simple: set hoverEnabled to true, and catch **entered()** and **exited()** signals. And if you just need a simple state-based check to see if the mouse is pointing to your entity, check that the **containsMouse** property equals **true**.

Here's a QML example to see mouse interaction techniques in action.

Code sample 1: PickableEntity.qml, catching mouse events

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

Entity {
    id: root
    signal pressed(var event)
    signal clicked(var event)
    signal released(var event)
    signal entered()
    signal exited()

    property alias position: transform.
        translation
    property color hoveredColor: "orange"
    property color pressedColor: "brown"
    readonly property bool containsMouse:
        objectPicker.containsMouse
    readonly property bool isPressed:
        objectPicker.pressed

    property GeometryRenderer mesh

    ObjectPicker {
        id: objectPicker
        hoverEnabled: true
        onPressed: root.pressed(pick)
        onReleased: root.released(pick)
        onEntered: root.entered()
        onExited: root.exited()
        onClicked: root.clicked(pick)
    }

    PhongMaterial {
        id: material
        diffuse: objectPicker.pressed
                ? pressedColor
                : objectPicker.containsMouse
                ? hoveredColor
                : "red"
    }

    components: [mesh, material,
        objectPicker]
}
```

Use QMouseDevice to get input for tasks like rotating the camera angle, translating a scene around, or reacting to mouse button states

Code sample 2: main.qml, reacting to mouse events

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

DefaultSceneEntity {
    id: scene

    SphereMesh {
        id: sphereMesh
        rings: 30
        slices: 30
    }

    PickableEntity {
        mesh: sphereMesh
        position.x: -2

        pressedColor: "blue"
        hoveredColor: "lightBlue"

        onClicked: console.log("Clicked
             sphere 1")
    }

    PickableEntity {
        mesh: sphereMesh
        position.x: 2

        pressedColor: "green"
        hoveredColor: "lightGreen"

        onClicked: console.log("Clicked
             sphere 2")
    }

    camera: Camera {
        position: Qt.vector3d(0, 16, 11)
        viewCenter: Qt.vector3d(0, 0, 0)
    }
}
```

Picking objects isn't the only thing you can do with a mouse. The mouse position can be used as input for tasks like rotating the camera angle or translating a scene around. Or when you want your application to react directly to mouse button states. In these cases, you'd access a **QMouseDevice** directly. To catch events from the **QMouseDevice**, you also need a **QMouseHandler**. Here's a very simple app that does nothing but catch mouse events and print them to the console.

Code sample 3: creating a mouse handler

```
MouseDevice {
        id: mouseDevice
    }

Entity {
    components: [
        MouseHandler {
            sourceDevice: mouseDevice
            onReleased: {
            switch (mouse.button) {
                case Qt.LeftButton:
                    console.log("Left mouse
                          click");
                    break;
               case Qt.RightButton:
                    console.log("Right
                          mouse click");
                    break;
                }
            }
            onPositionChanged: {
                console.log("Mouse moved
                    (",mouse.x,
                    ",",mouse.y,")")
            }
        }
    ]
}
```

Qt Quick animation changes aren't synchronized with the engine frame rate, which can cause shearing or other visual glitches

Using keyboards

Now you'll note that we've only covered the mouse here – what about other forms of input? Input devices in Qt 3D are handled via the class **QAbstractPhysicalDevice**. There are two classes provided by default (**QKeyboardDevice** and **QMouseDevice**) that inherit from **QAbstractPhysicalDevice**.

We've already covered the basics of the mouse, so a brief example on keyboards is next. Here are the bare necessities.

Code sample 4: handling keyboard input

```
KeyboardDevice {
    id: keyboardDevice
}

Entity {
    components: [
        KeyboardHandler {
            sourceDevice: keyboardDevice
                focus: true
            onUpPressed: box.position.z
                -= 0.5
            onDownPressed: box.position.z
                += 0.5
            onLeftPressed: box.position.x
                -= 0.5
            onRightPressed: box.position.x
                += 0.5
        }
    ]
}
```

How do you allow the key presses in the above example to smoothly rotate the object instead of introducing jerky motion? For that, you'll need an accumulator – a class that tracks an input device's movement signals to smoothly deliver appropriate acceleration and velocity cues to an application. Any time you're managing the rotation or translation of objects, you'll probably want to incorporate an accumulator.

Unfortunately, we don't cover accumulators in this whitepaper due to space constraints. To get more information about accumulators, managing touchscreens, or interfacing with other non-default devices – among many other topics – refer to the documentation or take one of our Qt 3D training courses.

Qt 3D animation

In the last step of our previous Qt 3D whitepaper, we used Qt Quick animations to pulse the width of the lines of our trefoil knot example. Qt Quick animation can also be used to implement much more complex animations – changing object position, size, or other properties. So, is that all we need to know about animation?

Hardly. One problem with Qt Quick animation is that it's executed on the main thread. This stalls other tasks on the main event loop. It doesn't use our modern multi-core and hyper-threaded CPUs effectively. Another problematic aspect of using Qt Quick to animate Qt 3D is that animation changes aren't synchronized with the engine frame rate, which can cause shearing or other visual glitches. Not to mention that if you're using Qt Quick only for animation, you're pulling in a lot of logic and dependencies that you don't really need.

The solution is to use **Qt3DAnimation**::**QAnimationAspect**. This provides the APIs necessary to map your animation in a Qt 3D-approved way. **QtAnimationAspect** inherits from **AbstractAnimationClip**. Like a movie clip, an **AnimationClip** provides the necessary manipulation of object properties through a series of key frames. The key frames define properties such as position, size, rotation, or color and specify their values at the beginning

Qt 3D was designed to be compatible with Blender and shares the same method of defining animation curves

and ending of the animation, as well as any important transitions in-between. The bundling of the property type and its values over time is called a channel and the channels (stored in **AnimationClip.clipData**) dictate how properties are smoothly interpolated between key frames throughout the duration of the animation.

At this time, **AnimationClipData** instances can only be created in C++, so your QML code will need some extra C++ helper routines. Let's look at an example that defines the movement of an object's position.

Code sample 5: creating position-based animation data

```
auto data = Qt3DAnimation::
      QAnimationClipData();

// Add a channel for a Location animation
auto location = Qt3DAnimation::
      QChannel(QLatin1String("Location"));

auto locationX =
      Qt3DAnimation::QChannelComponent
      (QLatin1String("Location X"));
locationX.appendKeyFrame(Qt3DAnimation::
      QKeyFrame({0.0f, -2.0f},
      {-1.0f, 0.0f},{1.0f, 0.0f}));
locationX.appendKeyFrame(Qt3DAnimation::
      QKeyFrame({2.45f, 2.0f},
      {1.45f, 5.0f},{3.45f, 5.0f}));

auto locationY =
      Qt3DAnimation::QChannelComponent
      (QLatin1String("Location Y"));
locationY.appendKeyFrame(Qt3DAnimation::
      QKeyFrame({0.0f, 0.5f},
      {-1.0f, 0.5f},{1.0f, 0.5f}));

auto locationZ =
      Qt3DAnimation::QChannelComponent
      (QLatin1String("Location Z"));
locationZ.appendKeyFrame(Qt3DAnimation::
      QKeyFrame({0.0f, 0.0f},
      {-1.0f, 0.0f},{1.0f, 0.0f}));

location.appendChannelComponent(locationX);
location.appendChannelComponent(locationY);
location.appendChannelComponent(locationZ);

data.appendChannel(location);
```

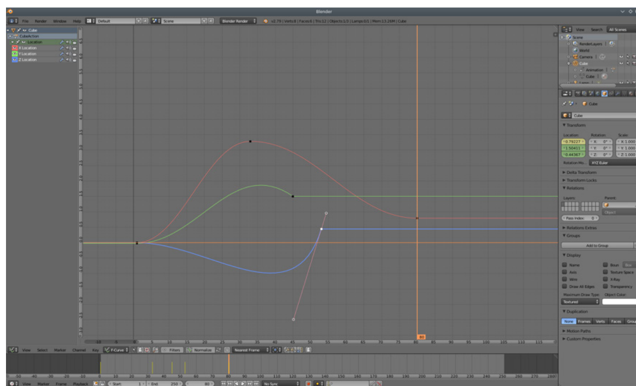
A clip contains several channels – one for each component being manipulated in the animation – and each channel contains the array of key frames that specify the animation. Each key frame consists of three 2D points, a position, and two control points. Multiple key frames together define a Bezier curve. Although you can use different interpolation algorithms, we'll assume Bezier curves for the sake of this explanation. The x-axis of the curve represents time – the time of the animation – while the y-axis represents the value of the parameter. In the position animation code sample above, we have a single point for y and z, so those coordinates won't change. However, the x coordinate uses a curve with two points (two key frames), so when we do the animation, that parameter will vary according to the height of the parameter curve.

The following screenshot from Blender shows how animation curves are defined. Thankfully, Qt 3D shares the same method of defining animation, so we can use it for a visual illustration that's easier to understand than a code snippet. (That similarity is not by mere chance, as Qt 3D was designed to be compatible with Blender.)

There are three curves being created in the application that describe a 3D point's position with x (red), y (green), and z (blue). Each black dot is associated with two control points – the key frame – that the Blender UI hides unless you're editing them. The control points are visible on the blue z curve, where the point is highlighted in white. The control points are the two hollow points connected to the white dot

Key frame positions and control points define a curve, which determines the values animating each parameter

via pink lines. The red x parameter has three key frames, while the green y and blue z each have two key frames – but you can use as many key frames as you need to smoothly define the motion of your object. The vertical orange line represents time: as it sweeps from left to right, the position of each curve at that position will control the associated parameter in the model.



Position animation curves in Blender

There isn't anything particularly special about these curves being associated with x, y, and z parameters in this screenshot. They represent the varying value of a parameter over time and could also represent any other parameter change – such as a color transformation with red, green, and blue components – with the same key frame data. Let's see what changes are needed in order to animate an object's color.

Code sample 6: creating color-based animation data

```
auto data = Qt3DAnimation::
       QAnimationClipData();

// Add a channel for a Color animation
auto color = Qt3DAnimation::
       QChannel(QLatin1String("Color"));

auto colorR = Qt3DAnimation::
       QChannelComponent(QLatin1String
       ("Color R"));
colorR.appendKeyFrame(Qt3DAnimation::
       QKeyFrame({0.0f, 0.05f},
       {-1.0f, 0.0f},{1.0f, 0.0f}));
colorR.appendKeyFrame(Qt3DAnimation::
       QKeyFrame({2.45f, 1.0f},
       {1.5f, 0.0f},{3.45f, 5.0f}));

auto colorG = Qt3DAnimation::
       QChannelComponent(QLatin1String
       ("Color G"));
colorG.appendKeyFrame(Qt3DAnimation::
       QKeyFrame({0.0f, 0.05f},
       {-1.0f, 0.0f},{1.0f, 0.0f}));

auto colorB = Qt3DAnimation::
       QChannelComponent(QLatin1String
       ("Color B"));
colorB.appendKeyFrame(Qt3DAnimation::
       QKeyFrame({0.0f, 0.05f},
       {-1.0f, 0.0f}, {1.0f, 0.0f}));

color.appendChannelComponent(colorR);
color.appendChannelComponent(colorG);
color.appendChannelComponent(colorB);

data.appendChannel(color);
```

Again, the key frame positions and control points define a curve that determines the values animating each parameter. It's just that this time the parameters are the red, green, and blue components of our image's color.

Kuesa loads scenes straight out of Blender or Autodesk 3ds Max into Qt 3D without requiring intermediate translation steps

Last but not least, what if you want to rotate the object? That's done by creating a set of channels for the key frame animation for a quaternion rotation.

Code sample 7: creating rotation-based animation data

```
auto data = Qt3DAnimation::
       QAnimationClipData();

// Add a channel for a Rotation animation
auto rotation =
       Qt3DAnimation::QChannel
       (QLatin1String("Rotation"));

auto rotationW =
       Qt3DAnimation::QChannelComponent
       (QLatin1String("Rotation W"));
rotationW.appendKeyFrame(Qt3DAnimation::
       QKeyFrame({0.0f, 1.0f},
       {-1.0f, 1.0f},{1.0f, 1.0f}));
rotationW.appendKeyFrame(Qt3DAnimation::
       QKeyFrame({2.45f, 0.0f},
       {1.5f, 0.0f}, {3.45f, 0.0f}));

auto rotationX =
       Qt3DAnimation::QChannelComponent
       (QLatin1String("Rotation X"));
rotationX.appendKeyFrame(Qt3DAnimation::
       QKeyFrame({0.0f, 0.0f},
       {-1.0f, 0.0f}, {1.0f, 0.0f}));

auto rotationY =
       Qt3DAnimation::QChannelComponent
       (QLatin1String("Rotation Y"));
rotationY.appendKeyFrame(Qt3DAnimation::
       QKeyFrame({0.0f, 0.0f},
       {-1.0f, 0.0f}, {1.0f, 0.0f}));

auto rotationZ =
       Qt3DAnimation::QChannelComponent
       (QLatin1String("Rotation Z"));
rotationZ.appendKeyFrame(Qt3DAnimation::
       QKeyFrame({0.0f, 0.0f},
       {-1.0f, 0.0f}, {1.0f, 0.0f}));
rotationZ.appendKeyFrameQt3DAnimation::
       QKeyFrame({2.45f, -1.0f},
       {1.5f, -1.0f}, {3.45f, -1.0f}));
rotation.appendChannelComponent(rotationW);
rotation.appendChannelComponent(rotationX);
rotation.appendChannelComponent(rotationY);
rotation.appendChannelComponent(rotationZ);

data.appendChannel(rotation);
```

If you're not new to 3D development, you'll know that a quaternion specifies the rotational angle of the object. Quaternions are great little bits of math that perfectly implement three-dimensional rotations. If you need a primer, there is the math-heavy Wikipedia article, but you can also try some coder friendly summaries here and here. (The next time you want to impress people at a party, you can tell them you're doing 3D math with four-dimensional complex numbers.)

Despite this math fun, it's not that easy to maintain data directly inline like this. That's especially the case if you've got designers or artists who are providing the models and animation data, or if you're defining models with any more than trivial complexity. You'll want to use a 3D modelling tool like Blender or Adobe 3ds Max to build your objects and scenes; this will be able to help you test your animations within the tool. That's when you'll need the **AnimationClipLoader** class, which can load key frame clip data from a JSON file. And, of course, a plug-in is available for Blender that exports animations in the required format.

There's also another option to consider. We've created a tool called Kuesa that helps bridge the gap between designer and developer. Kuesa loads scenes straight out of Blender or Adobe 3ds Max into Qt 3D without requiring any intermediate translation steps and preserves the geometry, materials, and animations so you don't have to mess around with JSON or hardcoding your scenes.

Summary

We've only room to scratch the surface of Qt 3D input and animation here. For in-depth learning and the docs to fill in the missing gaps, follow up with a training course. And check out the many sample demos that ship with Qt Creator.

There's still more to cover in our Qt 3D whitepaper series. Now that we've touched on the basics, we can take look at how Qt 3D can handle specialized rendering techniques in our next instalment.

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.

www.kdab.com

© 2018 the KDAB Group. KDAB is a registered trademark of the KDAB Group. All other trademarks belong to their respective owners.