

Archived:Combining Qt Animation and State Machine Frameworks



Archived: This article is **archived** because it is not considered relevant for third-party developers creating commercial solutions today. If you think this article is still relevant, let us know by adding the template {{ReviewForRemovalFromArchive|user=~~~~|write your reason here}}.

Qt Quick should be used for all UI development on mobile devices. The approach described in this article (using C++ for the Qt app UI) is deprecated.

This article demonstrates how to combine Qt's Animation Framework and State Machine Framework to animate standard widgets.

Firstly it shows how to implement and application's logic using a state machine. Then it demonstrates how to use the animation and state machine frameworks together, animating standard widgets rather than graphics items.

Animation Framework Concepts

Qt's animation framework is quite interesting and the concepts are easy to understand. The framework is based on [QObject](#) and Qt's property system. The simplest approach is to create a [QPropertyAnimation](#) for each property of each [QObject](#) we want to animate, and to give the property animation a duration, an initial value, and a final value.

In this example we give a duration of *5000 milliseconds* and initial and final geometry specified as [QRect](#). When the animation starts the object is immediately set to the initial geometry and then its geometry will be changed over a 5 second period to reach the final geometry. So if the initial width is 50 pixels and the final width is 360 pixels then the width would be 112 pixels after 1 second, 174 pixels after 2 seconds, 236 pixels after 3 seconds, 298 pixels after 4 seconds and finally 360 pixels after 5 seconds. In this case the increment is 62 pixels per second - calculated from the difference between the final and initial widths divided by the duration ie $(360 - 50)/5 = 62$.

Qt uses a much finer time granularity than seconds, so the actual change of width might be from 50 pixels to 52 pixels to 54 pixels and so on. We will also use the [QEasingCurve](#) class which offers over forty different interpolation graphs.

State Machine Framework Concepts

Using Qt's state machine framework we can define the states of an application and the transitions and triggers between them. Like the animation framework, the state machine framework is heavily dependent on [QObject](#) and Qt's property system.

To set up a state machine we start by creating a [QStateMachine](#). Then we create the states we need ([QState](#) or [QFinalState](#)). For each state we assign property values (*QObject, property, value*), these are the values that the state machine will change the properties to when in the state.

Once the states have been set up then we create the transitions which define how the state machine changes from one state to another. When ever there is a change in the state it emits an `exited()` signal to the state it left behind, it emits `entered()` signal to the state it enters and emits a `finished ()` signal if the state is completed. Now when everything is set up we tell the state machine which state to use as its initial state and then call `qstateMachine::start()` to start the machine.

The state machine offers a lot more functionality - we've only used the very basic functionality here.

Basic Idea

The example is a state machine with three states, **s1**, **s2** and **s3**, where **s1** is the initial state and **s3** is the final state. The state machine has a single [QPushbutton](#) which has a different position in each state. When the button is clicked, the state machine moves to the next state and animates the button to its next position.

The three screenshots below were recorded while changing the states.



Class Definition

```
#ifndef STATEMACHINEQT_H
#define STATEMACHINEQT_H
#include <QMainWindow>
#include "QPushButton"
#include "QVBoxLayout"
#include "QStateMachine"
#include <QPropertyAnimation>
#include "QEventTransition"
#include " QMessageBox"
namespace Ui {
    class StateMachineQt;
}
class StateMachineQt : public QMainWindow
{
    Q_OBJECT
public:
    explicit StateMachineQt(QWidget *parent = 0);
    ~StateMachineQt();
private:
    Ui::StateMachineQt *ui;
public:
    QPushButton *iButton;
    QStateMachine *machine;
    QPropertyAnimation *animation;
public slots:
    void animate1();
    void animate2();
    void animate3();
};
#endif // STATEMACHINEQT_H
```

The slots `animate1/2/3()` are used to animate the button from one position to another.

The [QStateMachine](#) class provides a hierarchical finite state machine. It is part of [The State Machine Framework](#). The [QPropertyAnimation](#) class animates Qt properties. [QPropertyAnimation](#) interpolates over Qt properties. As property values are stored in [QVariants](#), the class inherits [QVariantAnimation](#), and supports animation of the same variant types as its super class.

Class Implementation

First we create a button of size 50x50.

```
iButton = new QPushButton(this);
iButton->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
iButton->setFixedSize(50, 50);
iButton->show();
```

Then we create the state machine and states:

```
machine = new QStateMachine(this);
QState *s1 = new QState();
s1->assignProperty(iButton, "text", "S1");
QState *s2 = new QState();
s2->assignProperty(iButton, "text", "S2");
QState *s3 = new QState();
s3->assignProperty(iButton, "text", "S3");
```

The `QState::assignProperty()` function is used to have a state set a property of a `QObject` when the state is entered.

Then we create the transitions by using the `QState::addTransition()` function.

```
s1->addTransition(iButton, SIGNAL(clicked()), s2);
s2->addTransition(iButton, SIGNAL(clicked()), s3);
s3->addTransition(iButton, SIGNAL(clicked()), s1);
```

Next, we add the states to the machine and set the machine's initial state:

```
machine->addState(s1);
machine->addState(s2);
machine->addState(s3);
machine->setInitialState(s1);
```

Finally, we start the state machine:

```
machine->start();
```

The `QState::entered()` signal is emitted when the state is entered:

```
QObject::connect(s1, SIGNAL(entered()), this, SLOT/animate1());
QObject::connect(s2, SIGNAL(entered()), this, SLOT/animate2());
QObject::connect(s3, SIGNAL(entered()), this, SLOT/animate3());
```

In the following snippet, the widget `animation()` slot will be called when states are entered.

```
animation = new QPropertyAnimation(iButton, "geometry");
```

```
animation->setDuration(5000);
animation->setStartValue(QRect(0,0, iButton->width(),iButton-> height()));
animation->setEndValue(QRect(310,135, iButton->width(),iButton->height()));
animation->setEasingCurve(QEasingCurve::OutBounce);
animation->start();
```

The button is animated from (0,0) position to (310,135) position while entering to *state 1* (similar animations happen when entering the other states). We have also added an easing effect with `setEasingCurve()` function which makes the animation "bounce" before it settles. Finally `start()` is called to start the animation.

Source Code

The full source code for this article is available here: [File:StateMachineQt.zip](#)

Related Articles

- [How to use QStateMachine in Qt](#)
- [Archived:Using QStateMachine and QState](#)