

Join the Stack Overflow Community

Stack Overflow is a community of 6.6 million programmers, just like you, helping each other. Join them; it only takes a minute:

[Sign up](#)

Stabilize QWidget::paintEvent() calls frequency

As i understand, `paintEvent()` is executed in 'main loop' of `QApplication` object, and can spend time for its internal system tasks, delaying execution of queued slots or other events.

But what if i need to play very smooth animation and i notice periodic main loop delays on that animation? Can i create separate special *very stable* "main loop" and reassign `paintEvent()` calling to it?

P.S. Yes, a GPU, OpenGL and other nice technologies was invented for smooth game-like animations, i know, i know.

My program: http://www.youtube.com/watch?v=KRk_LNd7EBg

Solution

`paintEvent()` call frequency stabilization that i am looking for, GPU, OpenGL or hardware vsync will not help me! The issue is a normal behaviour until i calculate pixel's position in integer numbers. There always will be impulses of pixel movement speed. To solve my "problem" i have to measure coordinates in real numbers (double, float) and implement anti-aliasing algorithm.

c++ qt animation paintevent

edited Aug 30 '13 at 19:28

asked Aug 30 '13 at 11:26

 pavelkolodin
930 ● 14 ● 39

Why do people keep on asking questions with the note "I know there's a solution that was invented for exactly this problem, but I don't want to use it because, well because." – nijansen Aug 30 '13 at 11:35

Because sometimes people know those techniques, but are not experienced enough to use them. Maybe they are in a project with a time line and don't have time to get comfortable with them. So they hope there is a twist how to adjust things they know for their needs. I don't think questions like that are wrong. They have the potential to show alternative ways to solve a problem. – Greenflow Aug 30 '13 at 11:41

@Greenflow Touché. – nijansen Aug 30 '13 at 11:43

`paintEvent()` should be used only to paint current state of widget end of story. If you want to animate widget you should change state of widget and call `update` periodically (using `QTimer` or `Qt Animation framework`). – Marek R Aug 30 '13 at 12:04

1 1000/16 = 60 Hz as expected from average LCD. – Marek R Aug 30 '13 at 15:24

1 Answer

What you need to do is what you want, but in the opposite way. You propose a special "stable" main loop. What you want to do instead is to do everything but GUI "stuff" in the GUI thread. This will make the main event loop "stable".

`update()` adds an order like "please repaint!" to main loop, but main loop may be busy, so animation will lag

The main loop will *not* be busy doing anything unless it's running the code that you wrote and

that you have explicit control over. There's no magic to it at all. If you don't run code in the main loop, it won't be busy. Your comment above is not true in this respect. If you don't run stuff in the main loop, it won't be busy, and everything will happen right away - as soon as an `update()` is called. You might want to actually trace the execution of the code in the debugger to see it for yourself.

Qt by itself doesn't bog down the main event loop with unnecessary tasks unless you tell it to do so. What you want is to process everything but GUI interaction in another thread. Stuff like network access, file access, even `QSettings` access -- it should all be done in `QObject`s that live in a worker thread. Only the main, GUI thread should handle user interaction, and only in minimal fashion - it should only do what is directly needed to respond to events and to repaint stuff. Any other processing must be done outside of the GUI thread. That's how you get smooth animations.

Another important thing is that your animations should be driven by real time, not by assumed time. Thus when you step the animation, you should use `QElapsedTime` to measure how long it was since the last step, and use this time to calculate animated variables. The `QAbstractAnimation` and friends already handle this for you. If you don't use them, you'll need to do it yourself.

My hunch is that your code is just bad and does things in non-Qt-idiomatic way, and thus suffers. There are likely simple architectural reasons for why it's not smooth.

Below is a simple example of how you might do it in a `QWidget`. Note the conspicuous absence of anything related to time, except for the FPS calculation. That's the beauty of Qt. The `paintEvent()` is querying the animation's `currentValue()` directly. It could also store the value in the `newValue()` slot and use it instead, although that leaves a possibility of delay between the time the value was calculated and the time the value is used - say, due to preemption.

I've provided an [example that leverages Graphics View Framework](#) in another answer.

In the case of your application, you should be choosing where in the waveform to render the spectrum based on `QElapsedTime` since you've started the playback. That's all there's to it.

The example supports Qt 4/5 and leverages `QOpenGLWidget` on Qt 5.4 and later instead of the then-deprecated `QGLWidget`.



```
// https://github.com/KubaO/stackoverflow/tree/master/questions/widget-animation-18531776
#include <QtGlobal>
#if QT_VERSION >= QT_VERSION_CHECK(5,4,0)
#include <QtWidgets>
typedef QOpenGLWidget GLWidget;
#else if QT_VERSION >= QT_VERSION_CHECK(5,0,0)
#include <QtWidgets>
typedef QGLWidget GLWidget;
#else // Qt 4
#include <QtGui>
#include <QtOpenGL>
typedef QGLWidget GLWidget;
#endif

class Widget: public GLWidget
{
    QElapsedTimer m_timer;
    struct Animation : public QVariantAnimation {
        void updateCurrentValue(const QVariant &) {}
    } m_anim;
    QPolygonF m_polygon;
    qreal m_fps;
    void paintEvent(QPaintEvent *) {
        const qreal t = 0.05;
        qreal iFps = 1E9/m_timer.nsecsElapsed();
        m_fps = (1.0-t)*m_fps + t*iFps;
        int len = qMin(height(), width());
        QPainter p(this);
        p.drawText(rect(), QString("%1,%2 FPS").arg(m_fps, 0, 'f', 0).arg(iFps, 0, 'f',
```

```
0));
    p.translate(width()/2.0, height()/2.0);
    p.scale(len*.8, len*.8);
    p.rotate(m_anim.currentValue().toReal());
    p.setPen(QPen(Qt::darkBlue, 0.1));
    p.drawPolygon(m_polygon);
    p.end();
    m_timer.restart();
}
public:
    Widget(QWidget *parent = 0) : GLWidget(parent), m_fps(0.0) {
        m_anim.setDuration(2000);
        m_anim.setStartValue(0);
        m_anim.setEndValue(360);
        m_anim.setEasingCurve(QEasingCurve::InBounce);
        m_anim.setLoopCount(-1);
        m_anim.start();
        m_polygon.resize(4);
        m_polygon[0] = QPointF(-0.3, 0);
        m_polygon[1] = QPointF(-0.5, 0.3);
        m_polygon[2] = QPointF(0.5, 0);
        m_polygon[3] = QPointF(-0.5, -0.3);
        setAutoFillBackground(true);
        connect(&m_anim, SIGNAL(valueChanged(QVariant)), SLOT(update()));
    }
};

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();
    return a.exec();
}
```

edited Jun 1 '16 at 22:17

answered Aug 30 '13 at 17:46



Kuba Ober

50.3k ● 7 ● 49 ● 108

Thank you for your time and answer. My problem is much deeper: integer-based calculations. Even if `paintEvent()` will be called with very-very stable frequency it will not help me at all. The problem is that a pixel positions calculations are integer. To get what i want i should work with real numbers and sub-pixels. – [pavelkolodin](#) Aug 30 '13 at 18:16

Well, then what's stopping you? :) Thankfully `QPainter` is ready for your `qreal` coordinates. – [Kuba Ober](#) Aug 30 '13 at 18:38

The deal is that `paintEvent()` will not be called with a stable frequency. It will be done on a best-effort basis. Your code must deal with it. Integer calculations may not necessarily have anything to do with it. – [Kuba Ober](#) Aug 30 '13 at 18:41