



Pruebas de ensayo de nuevas funcionalidades en KDE 4.4

ANIMADO

Qt 4.6 aporta una recopilación de nuevas funcionalidades a KDE 4.4.

Mostraremos el framework para animaciones y la nueva característica de soporte multi-touch de KDE. **POR JOHAN THELIN**

El escritorio KDE y el framework de aplicaciones Qt subyacente son desarrollos independientes aunque paralelos. Las necesidades de KDE derivan en nuevas funcionalidades de Qt, y las nuevas funcionalidades de Qt pronto se convierten en nuevas características de KDE. La última liberación de Qt en su versión 4.6 viene empaquetada con nuevas características dirigidas a los escritorios KDE de todo el mundo. Podemos ver muchas de estas mejoras en KDE 4.4. Gran parte de sus cambios se centran en mejorar la experiencia de usuario. Por ejemplo, la barra de tareas ha recibido mucha atención, y en esta era, la de las redes sociales, hace su debut el cliente de blogging *Blogilo*. Este artículo analiza detalladamente algunas de las bondades técnicas disponibles en KDE 4.4. Más específicamente, mostraremos un ejemplo sobre cómo usar el framework para animaciones de Qt y describiremos un escenario para la integración de tecnologías multi-touch. Excepto en el caso de los programadores de KDE, no hay

que interactuar directamente con estos componentes, pero viendo su funcionamiento interno, podemos tener una idea bastante precisa acerca de esta nueva generación de efectos especiales que pronto harán aparición en las futuras actualizaciones de nuestras aplicaciones KDE favoritas.

Framework para Animaciones

Gran parte de los esfuerzos de KDE 4.4 han ido dirigidos a proporcionar una experiencia de usuario más suave y pulida. Parte de este trabajo se ha basado en transiciones de animaciones. Por ejemplo, al pasar por encima de los botones del marco de la ventana, éstos se acercan y alejan, en lugar de limitarse a cambiar de estado.

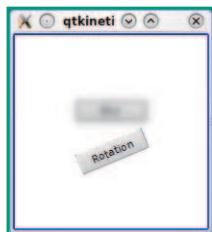


Figura 1: Las dos animaciones en acción.

Otra área de KDE en la que la suavidad es una de las prioridades es la del escritorio Plasma. El objetivo aquí es facilitar a los desarrolladores la adición de efectos y animaciones, los cuales harán de KDE un producto aún más intuitivo para el usuario. En Qt 4.6, este objetivo se ve satisfe-

cho a través de un framework de animaciones recién salido del horno basado en la clase *QAbstractAnimation*. Durante el desarrollo de Qt 4.6, se empezó a conocer dicho framework con el sobrenombre de *Qt Kînetic*.

El framework de animaciones se centra en el concepto de propiedades de animación. Las animaciones no se limitan al movimiento, la rotación y el escalado, sino que pueden afectar a cosas como la transparencia o el color. ¿Cómo se ve todo esto desde el punto de vista del desarrollador? Por algún motivo, el escritorio *Plasmoid* está creado alrededor de las clases dedicadas a la visualización de gráficos.

Como ejemplo de su funcionamiento, animaremos un conjunto de elementos gráficos en una escena. El código fuente más trascendente de este ejemplo se muestra en el Listado 1. Este listado describe dos animaciones diferentes de dos widgets que se muestran a través de vistas de gráficos. La ventana resultante, sin efectos, se presenta en la Figura 1.

El Listado 1 ilustra el constructor del widget de visualización, que hereda de *QGraphicsView*. La clase *QGraphicsView* se usa para mostrar los contenidos de una *QGraphicsScene*. Los contenidos de las escenas se crean a partir de objetos *QGraphicsItem*.

La línea 4 simplemente instala el renderizado de la vista, con anti-alias, para transformar los mapas de píxeles del modo más suave posible. De esta forma, se proporciona la mayor calidad de renderizado posible a un coste computacional asumible por la mayoría de las máquinas de escritorio.

Las líneas 5 y 6 crean una escena y garantizan su visualización a través de esta vista. A la escena se le pasa un rectángulo que se extiende a 200 píxeles de ancho y de alto desde las coordenadas de origen (-100, -100). Si no se especifica un rectángulo de escena, ésta crecerá a demanda, haciendo que sus dimensiones vengan dadas por las animaciones. Como consecuencia, se añadirán barras de desplazamiento o se moverán los contenidos (algo que queremos evitar).

En las líneas que van desde la 8 hasta la 10 se crea el primer elemento de la escena. Un elemento puede ser cualquier cosa que podamos imaginar; un mapa de bits, un dibujo vectorial en formato SVG, una forma básica como un rectángulo o un círculo, o incluso contenidos generados a partir de código propio. En este ejemplo incluiremos un widget, aunque son pesados y poco recomendados cuando el rendimiento es

crítico. Sin embargo, queremos tener acceso a la señal de clicado de modo que podamos interactuar con los eventos del ratón. Se podría conseguir directamente con algún elemento que no sea un widget, pero habría que hacer una sub-clase y capturar el evento que se produce al presionar el botón del ratón.

Primero se instancia el botón y luego se añade a la escena, que lo ubica automáticamente dentro del *QGraphicsProxyWidget*. Esta clase se encarga de todos los detalles necesarios para pasar los eventos y las operaciones de dibujo entre el widget y la escena de gráficos. Al añadir el botón, la llamada *setPos* lo centra y lo ubica ligeramente por encima de la mitad.

En las líneas de la 11 a la 13 entra en juego una mejora de Qt 4.6: los efectos gráficos. La última versión de Qt permite al desarrollador añadir efectos gráficos a cualquier widget o elemento. Los efectos estándar incluidos con Qt son *blur*, *colorize*, *drop shadow* y *add opacity* (desenfocado, coloreado, sombra arrojada y opacidad, respectivamente). Si no bastara con estos efectos, sería posible crear efectos personalizados o combinar los existentes. En el código fuente se aplica un efecto *blur* al elemento proxy que alberga el widget. Sin embargo, el radio de desenfoque es 0, por lo que en un primer momento no se aprecia.

Las líneas 15 a 18 se encargan de añadir otro botón a la escena con casi el mismo código usado para el primer botón. Este botón se sitúa debajo del anterior. De forma predeterminada, el punto de origen de la transformación se encuentra en la esquina superior izquierda, pero lo hemos movido al centro del botón. El título del botón es *Rotación*, lo que explica por qué hemos cambiado el punto de origen: la rotación se ve mejor si se hace desde el centro del botón y no desde la esquina superior izquierda del mismo.

Una vez creados ambos botones, ya podemos empezar con las clases para la animación. La línea 19 muestra a grosso modo el concepto de la nueva clase de propiedades de la animación. Dado un objeto de destino, que en este caso es *blurEffect*, y una propiedad a animar, *blurRadius*, el animador de la propiedad ya está listo. Lo único que faltan son algunos datos sobre el modo en que deberá cambiar la propiedad a lo largo del tiempo.

En todas las animaciones hay un valor temporal que va variando entre cero y uno. En las líneas 20 a 22 se definen los valores de *start* y *end*, pero también un valor clave

de 5.0 para el tiempo (es decir, a mitad de camino). El resultado es que el radio de desenfoque aumenta de cero a cinco, y luego hasta cero de nuevo. La línea 23 indica cuánto tiempo empleará el proceso: un segundo y medio.

En las líneas 26 a la 29 se instala otra animación para la propiedad de rotación del otro botón. El botón recorre un círculo completo durante dos segundos. Sin embargo, la línea 30 especifica la forma en que el valor de tiempo ha de ir desde 0 hasta 1. En este caso hemos usado *OutBounce*, de modo que la rotación finalizará con un pequeño rebote. De este modo conseguimos que la animación sea más aparente que si tuviese una velocidad de rotación lineal.

Concentrémonos ahora en la línea 31 (la última línea) para ver cómo se ajusta el framework de animaciones en las clases de

efectos gráficos. Nótese que para instalar la escena ha hecho falta casi tanto código como para instalar las animaciones.

En KDE 4.4 veremos muchas más animaciones. Hay un juego completo de ellas listas para ser incluidas en *plasmoids*. Por ejemplo, para hacer que un elemento lata, basta con usar el animador para crear una animación de latido, aplicarle un widget y conectarlo a una señal que lo inicie, como hemos hecho en el siguiente código:

```
Animation *pulseAnimation =
Animator::create(Animator::
PulseAnimation);
pulseAnimation->
setWidgetToAnimate(button);
connect(button,
SIGNAL(clicked()),
pulseAnimation, SLOT(start()));
```

Listado 1: Animación

```
01 QWidget::QWidget(QWidget *parent)
02 : QGraphicsView(parent)
03 {
04     setRenderHints(QPainter::Antialiasing |
05     QPainter::SmoothPixmapTransform);
06     QGraphicsScene *scene = new QGraphicsScene(-100, -100, 200, 200,
07     this);
08     setScene(scene);
09     QPushButton *blurButton = new QPushButton("Desenfocado");
10     QGraphicsProxyWidget *blurItem = scene->addWidget(blurButton);
11     blurItem->setPos(-blurButton->width()/2, -10-blurButton->height());
12     QGraphicsBlurEffect *blurEffect = new QGraphicsBlurEffect(this);
13     blurEffect->setBlurRadius(0);
14     blurItem->setGraphicsEffect(blurEffect);
15     QPushButton *rotateButton = new QPushButton("Rotación");
16     QGraphicsProxyWidget *rotateItem = scene->addWidget(rotateButton);
17     rotateItem->setPos(-rotateButton->width()/2, 10);
18     rotateItem->setTransformOriginPoint(rotateButton->width()/2,
19     rotateButton->height()/2);
20     QPropertyAnimation *blurAnimation = new
21     QPropertyAnimation(blurEffect, "blurRadius", this);
22     blurAnimation->setStartValue(0.0);
23     blurAnimation->setKeyValueAt(0.5, 10.0);
24     blurAnimation->setEndValue(0.0);
25     blurAnimation->setDuration(1500);
26     connect(blurButton, SIGNAL(clicked()), blurAnimation,
27     SLOT(start()));
28     QPropertyAnimation *rotateAnimation = new
29     QPropertyAnimation(rotateItem, "rotation", this);
30     rotateAnimation->setStartValue(0.0);
31     rotateAnimation->setEndValue(360.0);
32     rotateAnimation->setDuration(2000);
33     rotateAnimation->setEasingCurve(QEasingCurve::OutBounce);
34     connect(rotateButton, SIGNAL(clicked()), rotateAnimation,
35     SLOT(start()));
36 }
```

Multi-Touch

Otra tecnología que finalmente está apareciendo en Qt y KDE es multi-touch. Popularizada a través de los teléfonos avanzados, la tecnología multi-touch ha llegado hasta los tablet PCs y el software de escritorio. Con ella, el usuario puede manejar una pantalla táctil con dos dedos a la vez.

La introducción de multi-touch supone la aparición de dos problemas para el desarrollador. Primero, el usuario puede alterar varias partes de la interfaz de usuario a la vez, como si hubiese varios punteros al mismo tiempo, lo que limita el número de suposiciones que se pueden hacer en los casos en los que un widget depende de otro. No se trata de un problema que se pueda solucionar con Qt, ni siquiera con KDE; en lugar de eso, cada desarrollador de aplicaciones deberá tener en cuenta las implicaciones antes de habilitar el soporte para multi-touch.

En segundo lugar, la interpretación del significado de múltiples puntos de contacto se ha de resolver a través de gestos, los cuales permiten a Qt interpretar los movimientos de los puntos de contacto. Los gestos se pueden analizar a través de una API de más alto nivel, que siempre será más fácil que tratar de decodificar e interpretar manualmente la interacción entre los distintos puntos de contacto.

El Listado 2 incluye la parte más interesante de la clase *PinchWidget* (Figura 2), en la que el rectángulo rotado y escalado se controla a través del gesto de pellizcado (*pinch*). El rectángulo oscuro representa la

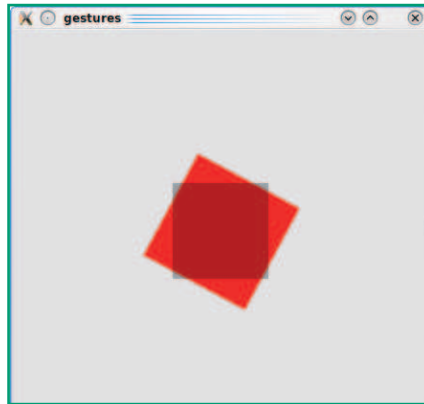


Figura 2: Widget pinch en acción.

ubicación original previa a la rotación y el escalado.

Nota: El gesto de pellizcado es el gesto hecho con dos dedos con el que el iPhone se hizo famoso. Separando los dedos se amplía el zoom y juntándolos se reduce. Al rotar ambos dedos se rota la imagen. De todo esto se encarga PinchWidget, pero con un simple rectángulo.

Fuera del listado, en el constructor de *PinchWidget*, se hace una llamada a *grabGesture* (*Qt::PinchGesture*). Sin esta llamada, el widget no recibirá ningún evento de gestos. Al recibir el gesto, el programa lo puede interceptar como un evento.

El método del evento (líneas 1 a 6 del Listado 2) intercepta los eventos de gestos y los pasa al método *gestureEvent* (líneas 8 a 16), que a su vez determina si el gesto es un pellizco. De ser así, se lo pasa al método *pinchGesture*.

El método *pinchGesture* (líneas 18 a 38) interpreta el gesto y actualiza el estado del widget como corresponde. La clase *QPinchGesture* registra los cambios que se producen, algo que usamos en el primer *if* (línea 22) para comprobar si el ángulo de rotación ha cambiado. Si el ángulo cambia, el programa actualiza la variable *rotationAngle*. Si cambia la escala (línea 28), el programa actualiza *currentScaleFactor* (la escala de la operación de redimensionamiento actual).

El *if* de la línea 32 comprueba si el gesto ha finalizado. Si la respuesta es positiva, se actualiza *scaleFactor* y se pone *currentScaleFactor* a 1 para prepararlo para el próximo gesto de pellizcado. Independientemente de lo que ocurra después, el programa volverá a dibujar el widget en la línea 37.

Conclusión

El manejo de este tipo de eventos irá ganando importancia en las nuevas aplicaciones conforme se vayan actualizando los dispositivos de entrada. Sin embargo, para los desarrolladores de KDE no tiene por qué ser tan complicado. Por ejemplo, todos los Plasmoids se pueden pellizcar para rotarlos y escalarlos. Además del gesto de pellizcado, Qt 4.6 habilita el *panning* (o lo que es lo mismo, *scrolling* por contacto, por ejemplo para ir a la siguiente imagen en un visor de gráficos) y el *swiping*. Todos estos gestos están disponibles para los desarrolladores de aplicaciones KDE y, por extensión, para los usuarios. ■

Listado 2: Clase PinchWidget

```

01 bool PinchWidget::event(QEvent *event)                *gesture)
02 {                                                    19 {
03     if(event->type() == QEvent::Gesture)             20     QPinchGesture::ChangeFlags flags =
04         return                                       21     gesture->changeFlags();
05         gestureEvent(static_cast<QGestureEvent*>(event)); 22     if(flags & QPinchGesture::RotationAngleChanged)
06     return QWidget::event(event);                   23     {
07 }                                                    24         qreal value = gesture->rotationAngle();
08 bool PinchWidget::gestureEvent(QGestureEvent        25         qreal lastValue =
09 *event)                                             gesture->lastRotationAngle();
10 {                                                  26         rotationAngle += value - lastValue;
11     if(QGesture *pinch =                             27     }
12     event->gesture(Qt::PinchGesture))                28     if(flags & QPinchGesture::ScaleFactorChanged)
13     {                                               29     {
14         pinchGesture(static_cast<QPinchGesture*>(pinch)); 30         currentScaleFactor = gesture->scaleFactor();
15         return true;                                31     }
16     }                                               32     if(gesture->state() == Qt::GestureFinished)
17     return false;                                   33     {
18 void PinchWidget::pinchGesture(QPinchGesture        34         scaleFactor *= currentScaleFactor;
                                                    35         currentScaleFactor = 1;
                                                    36     }
                                                    37     update();
                                                    38 }

```