Getting started with the Gesture API & FrameAnimator API

Overview

12 Sep 2010

Series 40 6th Edition, Feature Pack 1 introduces Touch and Type UI, a new user interface concept which brings touch screen on Series 40 platform and combines it with the traditional keypad as user input methods. Along side with touch UI, two new Java APIs, Gesture API and Frame Animator API are introduced. These APIs enable creation of touch UI applications with enhanced user experiences. This article covers these two APIs and provides detailed steps in getting started with Java application development for Series 40 Touch and Type UI devices.

The Gesture API enables recognition of specific touch gestures like drag and flick by making the platform's gesture recognition engine available to MIDlet. This enables the MIDlet's user experience to match the native one on Series 40 Touch and Type UI devices.

The FrameAnimator API calculates motion interpolation for kinetic scrolling and linear animations. This can be used, for example, to implement list scrolling in response to a flick or drag gesture, enabling quick traversal of long lists. An upwards flick gesture would cause upward movement, which in the case of a scrolling list would result in list elements scrolling onto the screen from the bottom. Like Gesture API, FrameAnimator API uses the platform logic to control the scrolling of UI components. This ensures that a MIDlet has the same user experience as the core platform.

Gesture recognition

Gestures: :
- The Single tap is recognised by a quick touch down and release.
- The Long press is a touch, hold and release.
- The Long press repeated is a generated when a long press is held down.
- Drag and drop are defined as touch down, move the finger whilst keeping contact with the touch screen, stop and then release.
- The Flick gesture is defined as a touch down, move and release before stopping the finger movement.

Step 1

The Gesture API uses the Observer design pattern. To use this API, a MIDlet must first create a GestureInteractiveZone which defines a bounding rectangle for the Gesture event notifications. By default the bounding rectangle is the entire area taken up by the UI component. Only Gesture events that are initiated within the confines of the zone are passed to the MIDlet. The GestureInteractiveZone also defines the types of Gesture events to register for.

Once registered, all specified gesture events received on the container will be automatically routed to the GestureListener.

The gesture API supports the following events:

GESTURE_ALL - All gesture events.

GESTURE_TAP - Simple press and release.

GESTURE_LONG_PRESS - Press and hold for long press interval followed by release.

GESTURE_LONG_PRESS_REPEATED - Repeated long presses.

GESTURE_DRAG - Press and move.

GESTURE_DROP - Press and move followed by release. It is possible that a Flick gesture maybe recognised instead of the drop gesture.

GESTURE_FLICK - Press and move followed by release. It is possible that a Drop gesture maybe returned if Flick gesture cannot be recognised.

The application is limited to the number of rectangular gesture zones that the application can define per container.

The platform supports the application defining overlapping gesture zones, in this case the registered listener for each zone will receive the gesture event.

```
// Defines a GestureInteractiveZone for the whole screen and all Gesture types.
GestureInteractiveZone gizCanvas = new GestureInteractiveZone( GestureInteractiveZone.GESTURE_ALL );

// Defines a GestureInteractiveZone for the drag event only and on a restricted rectangle area
GestureInteractiveZone gizRectangle = new GestureInteractiveZone( GestureInteractiveZone.GESTURE_DRAG );
gizRectangle.setRectangle(myRectPosX, myRectPosY, myRectWidth, myRectHeight);
```

Step 2

The previously created zones must then be registered with the GestureRegistrationManager by passing in the container (either a Canvas or CustomItem) and the GestureInteractiveZone.

```
// Register the GestureInteractiveZones for the Canvas object myCanvas.
if (GestureRegistrationManager.register(myCanvas, gizCanvas))
 System.out.println("Gesture detection for the canvas added");

if (GestureRegistrationManager.register(myCanvas, gizRectangle))
```


(C) Copyright Nokia 2013. All rights reserved.

```
System.out.println("Gesture detection for the rectangle added");
```

Step 3

The MIDlet must then define a class that implements the GestureListener interface and set it as a Listener for a container with the GestureRegistrationManager by passing in the container (either a Canvas or a CustomItem) and the GestureListener. Each container can only have one listener associated with it.

```
// Set the GestureListener object myGestureListener for the Canvas object myCanvas.
GestureRegistrationManager.setListener(myCanvas, myGestureListener);
```

Now, all the gesture events detected for the 2 gesture zones previously defined will be sent to the listener myGestureListener.

The GestureListener interface defines a single method, gestureAction, which gets called when the platform's gesture recognition engine detects a gesture in one of the registered GestureInteractiveZones. The gestureAction method will receive a GestureEvent instance each time it is called. This GestureEvent holds the properties of the recently recognized gesture such as the type (TAP, DRAG, etc).

```
public void gestureAction(Object container, GestureInteractiveZone gestureZone, GestureEvent gestureEvent) {
 if (gestureZone.equals(gizCanvas))
  handleGestureCanvas(container, gestureZone, gestureEvent);
 else if (gestureZone.equals(gizRectangle))
  handleGestureRect(container, gestureZone, gestureEvent);
}

public void handleGestureCanvas(Object container, GestureInteractiveZone gestureZone, GestureEvent gestureEvent) {
 // Which gesture has been recognized
 switch (gestureEvent.getType()) {
  case GestureInteractiveZone.GESTURE_TAP: {
   // Handle the TAP gesture
  };break;
  case GestureInteractiveZone.GESTURE_FLICK: {
   // Handle the FLICK gesture
  };break;
  default:
   System.out.println(" + HandleGestureCanvas() event ignored.");
 }
}

public void handleGestureRect(Object container, GestureInteractiveZone gestureZone, GestureEvent gestureEvent) {
 // Which gesture has been recognized
 switch (gestureEvent.getType()) {
  case GestureInteractiveZone.GESTURE_DRAG: {
   // Handle the DRAG gesture
  };break;
  default:
   System.out.println("handleGestureRect() event ignored.");
 }
}
```

Frame animation

The FrameAnimator API consists of a single class FrameAnimator and a single interface FrameAnimatorListener.

The MIDlet must provide its own class that implements FrameAnimatorListener. This interface provides a single method animate which will be called repeatedly for each frame update in the triggered animation. It is up to the MIDlet to redraw the UI on each call to animate. The FrameAnimator API will simply pass to animate the new x and y coordinates, the deltas from the last frame update and whether this is the last frame, or not, in the current animation.

This FrameAnimator API is independent of the Gesture API and so to trigger the animations from the Touch Gestures the MIDlet needs to also register for Gesture Events using the Gesture API as seen before.

The FrameAnimator API supports two types of animations; drag and kinetic scroll. These map to the drag and flick Gesture Events respectively.

Step 4

To use this API a MIDlet must first create an instance of the FrameAnimator class and register the listener. It is possible to register the same listener with more than one FrameAnimator.

As well as taking the FrameAnimatorListener instance, the register function also takes a reference x and y locations of the UI component which will be animated, the percentage of the platform's default value regarding the maximum number of frames per second (maxFps) and the percentage of the platform's default value regarding the maximum number of pixels per seconds (maxPps) to use. The maxFps and maxPps will control how often the animate method may be called and the distance in pixels between consecutive frame updates.

```
// Create the frame animator for the rectangle
FrameAnimator myRectangleAnimator = new FrameAnimator();
// Register the listener for this animator
myRectangleAnimator.register(myRectPosX, myRectPosY, maxFps, maxPps, myFrameAnimatorListener);
```

The platform's default values for Fps and Pps can be queried via the system properties com.nokia.mid.ui.frameanimator.fps and com.nokia.mid.ui.frameanimator.pps

```
defaultFps = (short) Integer.parseInt(System.getProperty("com.nokia.mid.ui.frameanimator.fps"));
defaultPps = (short) Integer.parseInt(System.getProperty("com.nokia.mid.ui.frameanimator.pps"));
```

If the value of maxFps and maxPps passed to register is set to 0 (default) or 100 (100%), then the default value will be used. Possible values: 1-200.

Warning, if you decide to retrieve the platform's default values for Fps and Pps to perform some checking and then decide those values are good for your application, don't pass them on to the register method since those default values can be 60 for example (60 fps) and you would then register your

(C) Copyright Nokia 2013. All rights reserved.

animator with 60% of those default values (36 fps)!

Step 5

The listener of the animations must provide the implementation of the "animate" method defined by the FrameAnimatorListener interface which, as seen before, will be called repeatedly for each frame update in the triggered animation.

The platform can call this method one or more times as a result of the application triggering an animation via FrameAnimator.drag(int newX, int newY) or FrameAnimator.kineticScroll(int started, int direction, int friction, float angle). In each call, the current state of the animation is described via x and y, which are absolute values, and computed from the initial coordinates passed to FrameAnimator.register(int x, int y, short maxFps, short maxPps, FrameAnimatorListener listener) and the actual animation-properties. After an animation has ended, the coordinates are still maintained as long as the FrameAnimator is registered. Therefore a new animation will start off where the last one has stopped.

```
public void animate(FrameAnimator animator, int x, int y, short delta, short deltaX, short deltaY, boolean lastFrame) {
 // Which animator is being used
 if(animator.equals(myRectangleAnimator)) {
  // Handle new coordinates and deltas
   ...
  // Refresh UI
   ...

  // If last frame of the animation
  if (lastFrame) {
   // Handle last frame case
     ...
  }
 }
 else {
  ...
 }
}
```

Step 6

Now one just has to trigger an animation with the drag or kineticScroll methods when the right gesture has been recognized to perform the desired animation.

For a drag and drop:

```
case GestureInteractiveZone.GESTURE_DRAG: {
 ...
 rectAnimator.drag(newX, newY);
}
```

For a flick:

```
case GestureInteractiveZone.GESTURE_FLICK: {
 ...
 rectAnimator.kineticScroll(gestureEvent.getFlickSpeed(),
     FrameAnimator.FRAME_ANIMATOR_FREE_ANGLE,
     FrameAnimator.FRAME_ANIMATOR_FRICTION_MEDIUM,
     gestureEvent.getFlickDirection());
}
```

Important note

If using the platform's default values for the maximum frames per second (maxFps) and for the maximum pixels per seconds (maxPps), the flick event will work quite well while the drag&drop might be ridiculously slow, too slow to be usable. The reason for that is because the “drag” method issues a linear drag-movement and calling this method will trigger the FrameAnimator listeners’ “animate” method to be called one or more times to describe the movement as it happens. The platform guarantees that those calls occur within the specified framerate. But, if this method is called repeatedly, some intermediate drags might be skipped, so only the last call may be fed back to “animate”. Therefore lots of drag distances are skipped and the final movement doesn’t match the touch position.

A solution to this problem is to change the maxFps and maxPps values during registration. The maxFps and maxPps will control how often the animate method may be called and the distance in pixels between consecutive frame updates. On certain devices, a good-looking drag animation might force you to use the max value for maxFps, that is to say 200%. The thing is that you can end up with 120fps like that (default 60 * 200%) and this is approximately the number of times the animate method can be called per second (assuming the platform supports it), and the view needing to be updated, it makes approximately 120 repaints. This might not be really satisfying either.

Solution: A solution to the previous problem is to implement your own counters

After recognition of the drag gesture, when the “gestureAction” method is called by the platform, add up all the drag distances since the last call of “animate” and then call the “drag” method with these counters.

```
case GestureInteractiveZone.GESTURE_DRAG: {
 // Add up the drag distances since last call to animate()
 dragCounterX += gestureEvent.getDragDistanceX();
 dragCounterY += gestureEvent.getDragDistanceY();
 // Call the drag method with the new coordinates
 int newX = (dragCounterX + getMyRectPosX());
 int newY = (dragCounterY + getMyRectPosY());
 rectAnimator.drag( newX, newY);
}
```

When the platform eventually calls the “animate” method the position can be updated normally (which takes into account all the drag distances this time) and then reset the counter. Like that the default values can remain for the maxFps and maxPps and the object being dragged does follow the real movement of the touching position.

(C) Copyright Nokia 2013. All rights reserved.

```
public void animate(FrameAnimator animator, int x, int y, short delta, short deltaX, short deltaY, boolean lastFrame) {
 // Update UI component with the new coordinates for the rectangle
 setMyRectPosX(x);
 setMyRectPosY(y);
 // Update the Gesture Interactive Zone for the rectangle
 gizRectangle.setRectangle(getMyRectPosX(), getMyRectPosY(), getMyRectWidth(), getMyRectHeight());
 // Refresh screen
 repaint();
 // Reset dragging counters
 resetDragCounterX();
 resetDragCounterY();
}
```

Example applications

This example uses a canvas on which a rectangle is displayed. Wherever the user clicks on the canvas a TAP event is recognized and a small red square is drawn. Drag and drops can also be performed on the rectangle which can be moved on the whole surface of the canvas. If one flicks the canvas the rectangle is thrown following this same direction and stops on the sides of the screen before getting out of the screen visible space.

Download the complete source code of the ready-to-use MIDlet described in this page: Media:GestureAndFrameAnimatorMIDlet.zip

You can also see three other examples in action by downloading the code contained in this package: TTTest1.zip

Related pages

- MIDP: Racer Game Example

(C) Copyright Nokia 2013. All rights reserved.

 (C) Copyright Nokia 2013. All rights reserved.