Лекция 5. Разработка приложений для планшетных компьютеров

Планшетные компьютеры. Датчики. Сенсорный экран.

5.1. Введение

Последнее время наряду с ноутбуками и смартфонами набирает популярность еще одна разновидность мобильных компьютеров – планшетные. Наиболее известным примером такого компьютера является Apple iPad. От ноутбуков планшеты отличает отсутствие традиционной клавиатуры, от смартфонов – большие размеры. Они ориентированы в первую очередь на развлечение пользователя, пассивное восприятие информации. В линейке аппаратных платформ для ОС MeeGo планшеты занимают одно из главных мест.

В данной главе мы рассмотрим аппаратные особенности планшетов с точки зрения прикладного программиста. Во-первых, это сенсорный экран, который в планшете заменяет почти все традиционные устройства ввода, такие как «мышь» и клавиатура. В случае «мыши» эти замена почти эквивалента по природе самого сенсорного экрана – оба они являются pointing device, и мы сосредоточимся как раз на этом случае.

Во-вторых, планшет обладает набором датчиков, например, датчиком ориентации, освещения, акселерометром и т.п. Для поддержания такого разнородного набора датчиков требуется самостоятельный API. Датчики, в свою очередь, позволяют реализовать более удобный, более интуитивный интерфейс управления программой на планшете, который был бы невозможен на нетбуке.

Кроме аппаратных особенностей, у планшетов есть особенности пользовательского интерфейса, для которого и исчезновение клавиатуры, и изменение размера экрана критичны. Эти особенности также являются очень существенными, но мы их касаться не будем.

5.2. Датчики

Планшетные компьютеры, как и многие другие мобильные устройства обладают набором датчиков, например, датчиком ориентации, освещения, акселерометром и т.п. Для поддержания такого разнородного набора датчиков требуется самостоятельный API. Датчики, в свою очередь, позволяют реализовать более удобный, более интуитивный интерфейс управления программой на планшете, который был бы невозможен на нетбуке.

Датчики – неотъемлемая часть современного планшета или смартфона. В продвинутых телефонах они появились очень давно – например, датчик, позволяющий телефону определить, приложен ли он к уху или должен работать в режиме громкой связи. Сейчас количество датчиков и их возможности существенно возросли. Это касается как датчиков высокого уровня (получение текущей ориентации экрана (портрет, пейзаж)), так и низкого уровня, как, например, получение в режиме реального времени показаний акселерометра.

Вместе с тем набор датчиков меняется от устройства к устройству, более того, трудно предугадать, какие новые датчики появятся в будущем. Данные, получаемые от датчиков, неоднородны. Эти ограничения делают API датчиков относительно сложным.

5.3. Сенсорные возможности Android

Один из приятных аспектов работы с платформой Android заключается в возможности получить доступ к некоторым полезным компонентам самого устройства. До сих пор разработчиков мобильных устройств разочаровывала невозможность доступа к их внутреннему оборудованию. Хотя между вами и металлом все же остается прослойка Java-среды Android, команда разработчиков Android вывела многие возможности аппаратуры на поверхность. А так как Android – платформа с открытым исходным кодом, то можно засучить рукава и написать собственный код для решения своих задач. В Android существует много различных датчиков, используя которые,

разработчики могут создавать интересные и полезные программные решения.

5.4. Интерфейс традиционных датчиков в Android API

Рассмотрим более подробно работу с датчиками в Android. Пакет `android.hardware` представляет разработчику API, который может быть использован при необходимости в приложениях, опирающихся на аппаратные возможности устройства. К примеру, пакет представляет интерфейс управления камерой и другими датчиками устройства. Операционная система Android по умолчанию представляет программную абстракцию любого физического элемента устройства, будь то камера или датчик движения, но программируя приложения для работы с датчиками устройства, необходимо первоначально убедится, что требуемый датчик присутствует в мобильном устройстве. Чтобы обезопасить себя от таких ошибок нужно в манифест-файле программы использовать директиву <*uses-feature*>.

В табл. 5.4.1 приведено описание аппаратно-ориентированных интерфейсов Android.

Таблица 5.4.1.

Camera.AutoFocusCallback	Интерфейс, осведомляющий об окончании автофокусировки камеры. Регистрирует событие на программном уровне ОС.
Camera.ErrorCallback	Интерфейс, осведомляющий об ошибках. Регистрирует событие на программном уровне ОС.
Camera.FaceDetectionListener	Интерфейс, распознающий лицо в предварительном просмотре. Регистрирует событие на программном уровне ОС.
Camera.OnZoomChangeListener	Интерфейс, осведомляющий об изменениях зума камеры. Регистрирует событие на программном уровне ОС.

Camera.PictureCallback	Интерфейс обратного вызова, используется для подачи данных после съемки.
Camera.PreviewCallback	Интерфейс обратного вызова используются, чтобы предоставить копии предварительного просмотра кадров как они отображаются на устройстве.
Camera.ShutterCallback	Интерфейс обратного вызова используется для обозначения момента фактического захвата изображения.
SensorEventListener	Интерфейс используется для получения уведомлений от менеджера датчиков (*SensorManager*), в тот момент, когда значение датчика изменилось.
SensorListener	Интерфейс реализован с помощью класса, который используется для вывода значений датчиков по мере их изменения в режиме реального времени. Приложение реализует этот интерфейс для мониторинга одного или нескольких имеющихся аппаратных датчиков.
Camera	Камера класс используется для установки настройки захвата изображения, старт/стоп предварительного просмотра, фотографии и извлечения кадров для кодирования видео.
Sensor	Класс представляет датчик.
SensorEvent	Класс представляет события датчика, а также содержит полезную информацию такую, как тип сенсора, временная метка, точность и данные сенсора.
SensorManager	Класс, обеспечивающий доступ к внутренним датчикам платформы

	Android.
Пакет android.os.*	Пакет, содержащий несколько полезных классов для взаимодействия с операционной средой, включая управление питанием, поиск файлов, обработчик и классы для обмена сообщениями. Как и многие другие портативные устройства, телефоны на базе Android могут потреблять достаточно много электроэнергии. Обеспечение "бодрствования" устройства в нужный момент, чтобы проконтролировать нужное событие, - важный аспект проектирования, заслуживающий особого внимания.

Рассмотрим более подробно несколько классов и интерфейсов.

SensorEventLstener. Используется для получения уведомлений от `SensorManager`, в тот момент, когда показания датчика меняются. Содержит описание двух основных методов, которые необходимо описать в реализующем классе.

1) `public abstract void onAccuracyChanged (Sensor sensor, int accuracy);`

Вызывается тогда, когда точность показаний датчика изменяется. Параметр `sensor` определяет датчик – объект класса `Sensor`. Параметр `accuracy` определяет точность измерений датчика. Точность может быть: высокая, низкая, средняя, ненадежные данные.

2) `public abstract void onSensorChanged (SensorEvent event, float values[]);`

Метод вызывается всякий раз, когда изменяется значение датчика. Этот метод вызывается только для датчиков контролируемых самим приложением. В число аргументов метода входит целое, которое указывает, что значение датчика изменилось, и массив значений с плавающей запятой, отражающих собственно значение датчика. Некоторые датчики выдают только одно значение данных, тогда как другие предоставляют три значения с плавающей запятой.

Датчики ориентации и акселерометр дают по три значения данных каждый.

SensorManager. Предоставляет доступ к различным датчикам устройства. Используя метод `getSystemService` с параметром SENSOR_SERVICE, можно получить экземпляр класса. Программируя приложения использующие датчики всегда необходимо убедиться в том, что датчики не функционирует, когда приложение приостановлено.

Пример демонстрирует основы работы с классом `SensorManager` и `Sensor`. Объект интересующего датчика может быть получен при помощи вызова метода, объекта класса `SensorManager, getDefaultSensor(Sensor.<тип>)`. Все типы датчиков Android описаны в виде констант в классе `Sensor`. Методы `onResume()` и `onPause()` необходимы для экономичного использования энергоресурсов устройства. Всякий раз, когда приложение приостанавливает свою работу, сбрасывайте слушатель датчика. Для взаимодействия с датчиком приложение должно зарегистрироваться на прием действий, связанных с одним или несколькими датчиками. Регистрация осуществляется с помощью метода `SensorManager.registerListener()`.

SensorEvent. Класс описывает различные типы событий датчиков, которые вычисляются устройством по-разному в зависимости от разновидности датчика. Будучи шаблоном, описывающим процесс функционирования различных датчиков (ускорения, ориентации…), класс представляет очень гибкое средство получения показаний от этих компонентов устройства. Рассмотрим более подробно структуру класса.

Класс имеет четыре основных поля, раскрывающих процесс взаимодействия датчика устройства с окружающим миром.

1) `Accuracy` – поле определяет точность показаний сенсора, обычно значение этой величины – константа, которая задается при регистрации слушателя на объект класса `Sensor`.
2) `Sensor` – объект сгенерировавший событие.
3) `Timestamp` – величина типа long, сообщающая время (в наносекундах) возникновения события.

4) `Float values[]` – значения с датчика, которые отображают процесс взаимодействия устройства с окружающим миром. Этот параметр зависит от типа датчика.

```
public class SensorActivity extends Activity,
implements SensorEventListener {
    private final SensorManager mSensorManager;
    private final Sensor mAccelerometer;

    public SensorActivity() {
        mSensorManager =
(SensorManager)getSystemService(SENSOR_SERVICE);
        mAccelerometer =
mSensorManager.getDefaultSensor

(Sensor.TYPE_ACCELEROMETER);
    }

    protected void onResume() {
        super.onResume();

mSensorManager.registerListener(this,mAccelerometer,

SensorManager.SENSOR_DELAY_NORMAL);
    }

    protected void onPause() {
        super.onPause();
        mSensorManager.unregisterListener(this);
    }

    public void onAccuracyChanged(Sensor sensor, int
accuracy) {
    }

    public void onSensorChanged(SensorEvent event) {
    }
}
```

5.5. Датчик ориентации

Устройство с предустановленной ОС Android включает датчик ориентации, который используется для распознавания положения телефона в пространстве. Как трактуются координатные оси в Android продемонстрировано на рис. 5.5.1.

Ориентация в Android определяется тремя величинами:

- Азимут в градусах – угол между осью X и северным направлением
  0 <= azimuth <= 360.
- Высота в градусах – угол между осью У и горизонтальным положением устройства.
  -180<= pitch <= 180.
- Вращение в градусах – угол между осью X и горизонтальным положением устройства.
  -90 <= roll <= 90.

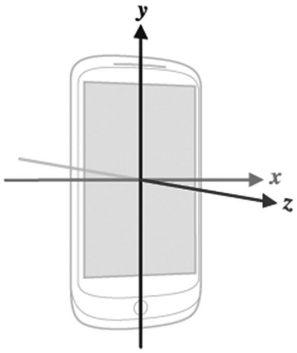

Рис. 5.5.1. Направление координатных осей в датчике ориентации

Используйте метод `getSystemService(Context.SENSOR_SERVICE)` чтобы инициализировать `SensorManager`. Далее используйте метод `getDefaultSensor(Sensor.TYPE_ORIENTATION)` чтобы создать объект класса *Sensor* и инициализировать его как датчик ориентации.

Для работы с датчиком ориентации необходимо реализовывать два метода, объявленные в интерфейсе: первый проверяет

изменение точности, второй вызывается всякий раз, когда происходит изменение показаний датчика.

5.6. Датчик акселерации

Android поддерживает большое число датчиков, которые могут быть использованы для получения информации о среде, окружающей мобильное устройство. Сейчас мы познакомимся, как в приложении использовать датчик ускорения.

Для того, чтобы использовать датчик ускорения, в программах необходимо реализовывать тот же интерфейс, что был описан выше. Параметр `rate` метода `registerlistener()` позволяет задавать точность показаний датчика ускорения устройства. Используйте метод `getSystemService(Context.SENSOR_SERVICE)` чтобы инициализировать `SensorManager`. Далее используйте метод `getDefaultSensor(Sensor.TYPE_ACCELEROMETER)` чтобы создать объект класса `Sensor` и инициализировать его как датчик акселерации. Рассмотрим поля объекта события метода `onSensorChanged(SensorEvent)`:

- `int accuracy` – определяет точность измерений.
- `long timestamp` – время в наносекундах, идентифицирующее начало события.
- `float[] values` – значения в системе СИ.
- `values[0]` – текущее ускорение по X минус ускорение свободного падения по оси X.
- `values[1]` – текущее ускорение по Y минус ускорение свободного падения по оси Y.
- `values[2]` – текущее ускорение по Z минус ускорение свободного падения по оси Z.

5.7. Датчик GPS

Android позволяет использовать показания GPS-датчика в тех случаях, когда логика приложения напрямую зависит от расположения устройства в пространстве, относительно Земли. GPS-датчик не является стандартным датчиком в Android, поэтому для его использования применяется немного другой подход.

`LocationManager` – менеджер управления службой GPS, основной класс, которой представляет GPS-датчик в Android. Класс предоставляет доступ к системной службе локации, позволяющей приложению получать периодические обновления в течении некоторого промежутка времени.

Для инициализации объекта класса `LocationManager` используйте метод `getSystemService()` с параметром `Location_Service`.

Чтобы запустить службу GPS используйте метод `requestLocationUpdate(String provider, long minTime, float minDistance, LocationListener listener)`.

Параметр provider задается в виде статической константы, которая определяет поставщика услуги GPS. Например, если указать `LocationManager.GPS_PROVIDER`, в качестве параметра, то поставщиком услуги будет спутник GPS, а если указать `LocationManager.NETWORK_PROVIDER` – то, поставщиком услуги будет сетевой протокол UDP или HTTP.

Второй параметр задает периодичность получения данных от GPS службы.

Третий параметр определяет минимальный размер области пространства, в котором мы не хотим получать уведомления от GPS службы. Этот параметр определяется логикой приложения.

Четвертый параметр – слушатель, который реагирует на изменения локации.

Для того, чтобы сбросить слушатель используйте метод `removeUpdates(LocationListener listener)`.

Рассмотрим подробно интерфейс `LocationListener`. Он включает в себя следующие методы:

1) `onLocationChanged(Location location)` – метод вызывается всякий раз, когда происходит изменение показаний GPS датчика. Количество вызовов данного метода напрямую зависит от того, как вы на него подписались (параметры minDistance, minTime). Экземпляр класса `Location` содержит показания датчика. Его основные поля: `londitude` (долгота), `latitude` (широта),

`altitude` (высота над уровнем моря), `accuracy` (точность), `timestamp` (время идентификации события).

2) `onStatusChanged (String provider, int status, Bundle extras)` – метод вызывается всякий раз, когда GPS программно терпит изменения(плохой сигнал, устройство не отвечает итд). Например данный метод вызывается, когда GPS служба не в состоянии установить местоположение или или недавно стала доступной после периода блокировки.
3) `onProviderEnabled(String provider)` – метод вызовется, если приложению доступна служба GPS, или вызовется после того, как пользователь включит GPS.
4) `onProviderDisabled(String provider)` - метод вызывается, если приложению недоступен GPS, вообще говоря, если устройство никогда не использовало службу GPS, то этот метод после автоматического вызова должен предоставить пользователю возможность включить GPS.

5.8. Программирование сенсорного экрана

От ноутбуков планшеты отличает отсутствие традиционной клавиатуры, от смартфонов – большие размеры. Они ориентированы в первую очередь на развлечение пользователя, пассивное восприятие информации. В линейке аппаратных платформ на Intel Atom планшеты занимают одно из главных мест.

В этом разделе мы рассмотрим аппаратные такую особенность планшетов, как сенсорный экран, который в планшете заменяет почти все традиционные устройства ввода, такие как «мышь» и клавиатура. В случае «мыши» эти замена почти эквивалента по природе самого сенсорного экрана – оба они являются pointing device, и мы сосредоточимся как раз на этом случае. Для программирования будем использовать технологию *Qt*.

Qt – это инструментарий, включающий в себя программный фреймворк, библиотеку элементов графического интерфейса и набор программ для разработки, – который используется для разработки межплатформенных приложений с графическим пользовательским интерфейсом преимущественно на языке C++. Однако, в различное время были созданы интерфейсы,

позволяющие вести разработку с использованием *Qt* и на других языках программирования, таких как: Python – PyQt, PySide; Ruby – QtRuby; Java – QtJambi; PHP – PHP-Qt и другие.

Инструментарий *Qt* лежит в основе популярной среди пользователей Unix-подобных систем среды рабочего стола KDE, а также таких приложений, как Skype, VLC, Virtual Box и многих других.

Использование API *Qt* вместо других, специфичных для платформы, программных интерфейсов, позволяет создавать приложения, которые, во многих случаях, без всяческих доработок будет компилироваться и исполняться на любой из ОС поддерживаемых *Qt*, а в большинстве других случаев требовать лишь незначительной доработки. Среди таких ОС, помимо MeeGo – Windows, Mac OS X, различные дистрибутивы Linux, Solaris, использующие оконную систему X11, Symbian, Windows CE.

5.8.1. Краткие сведения о *Qt*

Разработка *Qt* как графического toolkit (библиотеки графических компонентов) была начата в 1991 году Гаавардом Нордом и Айриком Шамбе-Ингом, основавшими впоследствии компанию Quasar Technologies, затем переименованную в Trolltech. Идея разработки кроссплатфоменного toolkit появилась во время работы над графическим приложением для медицинской индустрии, которое должно было работать в ОС Windows и Unix. Буква Q появилась в названии фреймворка, поскольку Гааварду очень нравилось её начертание в шрифте, использовавшемся в редакторе Emacs. Буква t, за которой скрывается слово «toolkit», была добавлена по аналогии с Xt – X Toolkit, библиотекой для создания виджетов в оконной системе X.

Несколько лет проект разрабатывался без представления на рынке. Первый релиз Qt был сделан в 1995 году. Он включил в себя набор графических компонент для X11/Unix и Windows. В версии 3.0, вышедшей в 2001 году, появилась также поддержка Mac OS X.
В различное время фреймворк Qt распространялся под разными лицензиями. Если версия Qt для оконной системы X11 изначально выпускалась как под коммерческой, так и под бесплатной (хотя и не свободной) лицензией с открытым исходным кодом, то первые версии для Windows и Mac OS X существовали лишь в версии для

коммерческого использования. Особую остроту вопрос лицензирования технологии приобрел с ростом популярности оконной среды KDE среди пользователей Linux в конце 90-ых годов, когда стало очевидно, что одна из важнейших компонент наиболее популярной свободной ОС не является свободным ПО. Проблема лицензирования X11-версии была решена при помощи перехода на свободную лицензию QPL и основания KDE Free Qt Foundation — организации, гарантирующей, что в случае, если разработка свободной версии Qt будет приостановлено, последняя версия будет выпущена под лицензией типа BSD.

Хотя к 2003 году версии *Qt* для OS X и для X11 выпускались под свободными лицензиями, версия для Windows по-прежнему выпускалась лишь под коммерческой лицензией. Это привело к тому, что в 2002 группа независимых разработчиков начала работу по портированию X11-версии фреймворка, выпущенной под лицензией GPL, на Windows. Работа эта, впрочем, не была завершена, поскольку в 2005 была выпущена версия фреймворка 4.0, в действие лицензии GPL было распространено на версии для всех поддерживаемых платформ. Добавленное позднее специальное исключение в лицензию, сделало возможным использование GPL-версии *Qt* в проектах, использующих одну из целого ряда свободных лицензий, таких, как BSD License, Eclipse Public License и других.

В 2008 году компания Trolltech была приобретена компанией Nokia и переименована сперва в *Qt* Software, а впоследствии — в *Qt* Development Frameworks. Вскоре после этого была выпущена версия фреймворка для основной мобильной ОС, использующейся Nokia — Symbian S60. С развитием другой мобильной ОС, разрабатываемой Nokia — Maemo, в *Qt* была добавлена поддержка и этой платформы.

В версии *Qt* 4.5, вышедшей 14 января 2009 г., в фреймворк была добавлена третья опция лицензирования — LGPL, что сделало возможным использование «бесплатной» версии *Qt* в проектах с закрытым кодом (при выполнении некоторых условий).

Основным преимуществом программирования с *Qt* является в упрощении и унификации процесса разработки программного обеспечения для различных целевых платформ. Сложности, возникающие, при портировании проектов с одной платформы на

другую, очевидны. В силу различия архитектур ОС и отсутствия общепринятых стандартов и интерфейсов, код приложения оказывается насквозь пронизанным обращениями к специфичными для платформы API. Это становится особенно заметным в участках кода, отвечающих за графический пользовательский интерфейс, однако зачастую даже безобидные с виду участки, использующие стандартизированные API, оказываются труднопортируемыми.

Qt в значительной степени облегчает решение этой проблемы, предоставляя широчайший набор унифицированных программных интерфейсов. Вместо API операционной системы разработчик использует API *Qt*. API *Qt* реализован для каждой конкретной целевой архитектуры и опирается на нативные API операционной системы. В силу этого, приложение, написанное с использованием API Qt, фактически использует высокую производительность нативных интерфейсов целевой платформы; часто библиотеки Qt являются лишь тонкой прослойкой между приложением и API ОС.

Ещё одно несомненное преимущество *Qt* состоит в том, что его API позволяет скрыть сложные интерфейсы внешних библиотек. Порой для выполнения некоторой достаточно простой операции с использованием API ОС, программисту приходится изучать объёмную документацию и реализовывать тяжеловесные функции, инициализирующие и деинициализирующие применяемую библиотеку и т. д. Наглядным примером этой проблемы является создание оконных приложений с использованием Windows API.

Другой аспект использования *Qt* заключается в унификации кода приложения. Крупный проект зачастую использует значительное число внешних библиотек, многие из которых используют весьма специфичные по стилю интерфейсы, что порождает разнородные участки кода. API *Qt* используют единый стиль и подход, что позволяет сделать ваш код более легко читаемым и ясным.

Итак, *Qt* позволяет запускать написанное с его помощью ПО в большинстве современных операционных систем путём простой компиляции программы для каждой ОС без изменения исходного кода. Он включает в себя основные классы, которые могут потребоваться при разработке прикладного программного обеспечения, начиная от элементов графического интерфейса и заканчивая классами для работы с сетью, базами данных и XML.

Кратко опишем ключевые библиотеки, входящие в дистрибутив:

- `QtCore` – базовые примитивы, не имеющие отношения к GUI;
- `QtGui` – примитивы GUI Phonon и QtMultimedia – библиотеки для работы с мультимедиа;
- `QtNetwork` – работа с сетью;
- `QtOpenGL` – поддержка OpenGL-графики;
- `QtXml` – работа с XML;
- `QtSql` – работа с SQL-базами данных;
- `QtScript` – позволяет использовать скриптовый язык, аналогичный JavaScript в Qt-приложениях;
- `QtWebKit` – позволяет работать с веб-движком (библиоткой для обработки и отображения Web-страниц) `WebKit`.

Для разработки кросс-платформенных приложений для мобильных устройств компания *Qt* Software разработала дополниельную библиотеку `Qt Mobility`, пока не входящую в состав основного дистрибутива. `Qt Mobility` должен обеспечить удобную разработку приложений для мобильных платформ, поддерживающих *Qt* и, в первую очередь, ОС MeeGo.

`Qt Mobility` предоставляет интерфейс для функциональностей, специфичных для мобильных устройств, таких как, например:

- сервисы (GSM-связь, Bluetooth) ;
- записная книжка;
- мгновенные сообщения;
- органайзер;
- устройства позиционирования;
- сенсоры (акселерометр, датчик освещённости).

В пакете *Qt* SDK поставляется набор инструментов, которые облегчают разработку приложений с использованием фреймворка. Перечислим основные:

- `QtCreator` – кроссплатформенная IDE для работы с фреймворком *Qt*, разработанная *Qt* Software. Эта IDE была специально разработана для работы с *Qt*, имеет возможности удаленной отладки, расширения плагинами, встроенный

QtDesigner и QtAssistant и графический фронтенд для gdb. QtCreator входит в состав SDK.

- QtDesigner – инструмент для визуального дизайна графических интерфейсов. В результате работы Qt Designer создается xml файл, описывающий графический интерфейс.
- QtLinguist – локализация интерфейса.
- QtAssistant – система справки.
- QtSimulator – эмулятор мобильных устройств.
- qmake – система сборки.
- moc – метаобъектный компилятор , предварительная система обработки исходного кода. Позволяет использовать механизм слотов и сигналов. Утилита moc ищет в заголовочных файлах на C++ описания классов, содержащие макрос Q_OBJECT, и создаёт дополнительный исходный файл на C++, содержащий реализацию дополнительных методов.
- uic – компилятор графических интерфейсов, который получает на вход xml файл, сгенерированный QtDesigner, и по нему выдает код на C++.
- rcc – компилятор ресурсов.

qmake – программное средство, с помощью которого упрощается процесс сборки проекта при разработке для разных платформ. qmake автоматизирует создание файла сборки Makefile, используя для этого более простой и лаконичный файл *.pro.

Утилита создает Makefile, основываясь на информации в файле проекта. Файлы проекта обычно создаются разработчиком, однако для их первичного создания можно также использовать и саму утитлиту qmake, запуская её с аргументом -project. qmake содержит дополнительные возможности для поддержки разработки с *Qt*, включая автоматическое создание правил для moc и uic.

Рассмотрим простой пример работы с qmake. Допустим, что у вас уже завершена начальная реализация вашего приложения, и у вас имеются следующие файлы: hello.cpp, hello.h, main.cpp. Используя текстовый редактор, создайте файл с названием hello.pro. Теперь в этот файл следует добавить

строки, которые сообщают qmake об исходных файлах, файлах-заголовках, используемых библиотеках, которые следует прилинковать в процессе сборки и др.

На первом шаге добавим исходные файлы в файл проекта. Чтобы это сделать, нужно использовать переменную SOURCES. Надо написать новую строку с SOURCES += и добавить hello.cpp после нее. Должно получиться наподобие:

```
SOURCES += hello.cpp
```

Теперь нужно повторить эти действия для каждого исходного файла в проекте. В итоге в нашем примере получается следующее:

```
SOURCES += hello.cpp
SOURCES += main.cpp
```

Или в одну строку:

```
SOURCES = hello.cpp \ main.cpp
```

Кроме исходных файлов должны быть указаны файлы заголовка. Для их добавления используется переменная HEADERS. После изменений наш файл выглядит так:

```
HEADERS += hello.h
SOURCES += hello.cpp \ main.cpp
```

Имя файла результата сборки устанавливается автоматически; оно такое же, как и имя файла проекта, но с суффиксом, соответствующим платформе. Например, если файл проекта называется - hello.pro, результатом сборки будет файл hello.exe для Windows и hello для Unix. Другое имя файла для результата сборки может быть указано в переменной target. Например,

```
TARGET = helloworld
```

Далее установим переменную CONFIG, отвечающую за общую конфигурацию сборки. Так как наш приложение использует Qt, то нужно поместить qt в строке CONFIG для того, чтобы qmake добавил релевантные библиотеки и обеспечил встроенные строки для moc и uic, включаемые в создаваемый файл сборки. Если в переменной CONFIG указать значение debug, то будет создана отладочная версия программы.

Переменная QT позволяет указать, какие модули Qt использует. Для тестового файла укажем, что мы хотим использовать библиотеки ядра (core), XML (xml) и библиотеки работы с сетью (net). В результате в файл будет записана следующая строка:

```
QT += core xml network
```

Переменная LIBS перечисляет внешние библиотеки, которые мы хотим прилинковать к приложению, в виде ключей для линковщика. В примере прилинкуем к приложению библиотеку `ncurses`:

```
LIBS += -lncurses
```

Зачастую возникает необходимость собирать приложение в разных вариантах, например, для разных целевых платформ. В `qmake` для поддержки различных видов сборки существует механизм scopes, который позволяет создавать условные блоки и в зависимости от выполнения условий переходить в те или иные состояния. Добавим небольшой пример и в наш файл. Введем условные блоки, которые в зависимости от целевой платформы будут добавлять исходные файлы в проект. Так, например для windows будет добавлен файл `hellowin.cpp` :

```
win32 {
    SOURCES += hellowin.cpp
}
```

А для unix `hellounix.cpp`:

```
unix {
    SOURCES += hellounix.cpp
}
```

Также при помощи простой функции `exists` проверим, существует ли файл `main.cpp`:

```
!exists( main.cpp ) {
    error( "No main.cpp file found" )
}
```

Запишем полностью получившийся `qmake` файл:

```
CONFIG += qt
QT += core xml network
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
LIBS += -lncurses
win32 {
    SOURCES += hellowin.cpp
}
unix {
    SOURCES += hellounix.cpp
```

```
}
!exists( main.cpp ) {
    error( "No main.cpp file found" )
}
```

Теперь qmake можно использовать для создания файла сборки приложения. В командной строке в каталоге с проектом нужно написать:

```
qmake hello.pro
```

Затем может быть запущена утилита make или nmake для сборки проекта.

Сигналы и слоты используются для обмена сообщениями между объектами. Механизм сигналов и слотов является особенностью *Qt*. Необходимость в подобном механизме возникает, когда требуется, чтобы при изменении одного объекта, оповещался другой. Так, например, при разработке графического интерфейса, при нажатии на кнопку «Закрыть» вызывается метод окна `close()`.

Техника сигналов и слотов реализована следующим образом: сигнал вырабатывается, когда происходит определенное событие, а слот — это функция, которая вызывается в ответ на определенный сигнал. Каждый класс может объявлять сигналы, которые он будет отправлять и слоты, которые можно ассоциировать с конкретными сигналами. При этом сигналы и слоты слабо связаны. Класс, который вырабатывает сигнал, не знает и не заботится о том, какие слоты его получат.

Сигналы и слоты могут иметь аргументы. Механизм сигналов и слотов Qt гарантирует, что если мы подключим сигнал к слоту, слот будет вызван с параметрами сигнала в нужное время.

Система слотов и сигналов реализована как надстройка над синтаксисом C++. Исходный файл обрабатывается метакомпилятором `moc`, который генерирует вспомогательные файлы. При этом ограничения метакомпилятора накладывают определенные ограничения на классы, использующие слоты и сигналы для взаимодействия. Так, например, такие классы не могут использовать механизм шаблонов C++.

5.9. Сенсорный экран (touchscreen)

Сенсорный экран – универсальное устройство ввода, которое заменило в планшете и клавиатуру, и «мышь». По принципу действия он очень схож с мышью – пользователь прикосновением пальца к экрану указывает на нем точку и задает вид действия. Для ввода текста применяются разного рода экранные клавиатуры, которые являются самостоятельными программными решениями, мы не будем их рассматривать.

Сенсорный экран может быть использован различными способами. Самый простой и частый способ – использовать стандартные виджеты *Qt*, которые уже адаптированы для работы с сенсорным экраном и программисту не надо ничего специально делать для этого. Этот метод плохо подходит в ситуации, когда пользовательский интерфейс сложный или нестандартный – например, в играх. В этом случае целесообразно перехватывать системные события *Qt*, относящиеся к сенсорному экрану, и самостоятельно их обрабатывать.

Эти события бывают двух видов – события собственно от сенсорного экрана и события gestures. Gesture (буквально «жест») – это комбинация действий с сенсорным экраном, которые обозначают определенное событие для пользовательского интерфейса. Например, если человек установил два пальца на мультисенсорный экран и развел их в стороны – это соответствует gesture для увеличения масштаба содержимого виджета. Gesture могут быть как встроенные в систему, так и задаваемые программистом. Встроенные gesture служат для унификации опыта пользователя при работе с сенсорным экраном.

На более низком уровне абстрагирования находятся события непосредственно от сенсорного экрана, содержащие информацию о прикосновениях к нему. В *Qt* эти события и gesture могут появляться одновременно. Обработка этих событий позволяет получить наиболее полную информацию от сенсорного экрана в рамках его штатных возможностей.

Наконец, в исключительных случаях возможно работа с сенсорным экраном на уровне его драйвера в ядре MeeGo. Это наиболее сложный и трудоемкий способ и он имеет смысл только в том случае, если у конкретной модели сенсорного экрана есть

какие-то необходимые нам возможности, которые не поддерживаются *Qt*. Необходимо также учитывать проблемы с переносимостью созданной таким образом программы.

Далее мы будем говорить о событиях сенсорного экрана. Для перехвата событий необходимо применить традиционную для Qt схему – создать свой собственный виджет, унаследовав его от какого-нибудь стандартного, и в нем переопределить метод `event()`:

```
bool MyWidget::event(QEvent *event) {
    if (event->type() == QEvent::SomeEvent) {
        QSomeEvent *e =
static_cast<QSomeEvent*>(event);
          // ...
        e->accept();
        return true;
    }
    return BaseWidget::event(event);
}
```

В этом фрагменте кода важны четыре момента. Во-первых, нужно выделить только те события, которые нам интересны, не нужно пытаться обработать все события, которые получает виджет. Для этого в начале мы проверяем тип события, в данном случае `QEvent::SomeEvent`.

Во-вторых, сам по себе базовый класс событий `QEvent` малоинформативен, после того, как мы определились с типом события, надо привести его к соответствующему типу (в примере `QSomeEvent`). В *Qt* для этого рекомендуется использовать `static_cast`.

В-третьих, после обработки события его следует пометить как обработанное с помощью метода `accept()` и возврата `true`.

В-четвертых, все события, которые не были обработаны в новом обработчике, должны быть переданы обработчику базового класса.

Для сенсорного экрана выделены следующие типы событий: `QEvent::TouchBegin` (начало прикосновения), `QEvent::TouchUpdate` (продолжение прикосновения), `QEvent::TouchEnd` (завершение прикосновения). Класс `QTouchEvent` появился недавно, в *Qt* 4.6, для поддержки сенсорных экранов и сенсорных площадок (touchpad). Вложенный

класс `QTouchEvent::TouchPoint` описывает точку прикосновения к сенсорному экрану. В `QTouchEvent` имеется список таких точек – например, если пользователь касается экрана четырьмя пальцами, то в списке будут четыре точки. В простейшем случае достаточно рассматривать первую точку списка. Кроме координат, с каждой точкой ассоциировано ее состояние, для простейшего анализа оно также не важно.

Таким образом, можно выстроить несколько стратегий обработки событий сенсорного экрана. В простейшем случае (клик) достаточно реагировать только на `TouchBegin` или `TouchEnd`, в более сложных случаях необходимо анализировать данные в `TouchUpdate` на лету или накапливать их для последующего анализа при завершении прикосновения.

По умолчанию виджет не получает события от сенсорного экрана. Чтобы начать их получать, необходимо выполнить вызов `setAcceptTouchEvents(true)`. Другая проблема, которая может возникнуть с сенсорным экраном – обработка событий gesture других виджетов может приводить к нежелательным эффектам. В этом случае необходимо также начать получать события желаемых gestures в своем виджете, выполнив, например, `grabGesture(Qt::PanGesture)`, после чего в обработчике событий отмечать эти события как обработанные. Другие виджеты перестанут получать события и нежелательные эффекты прекратятся.

5.10. Выводы

В этой лекции были рассмотрены две основных аппаратных особенности планшетных компьютеров – сенсорный экран как универсальное устройство ввода и набор датчиков. Обе особенности отражают тот факт, что планшет – устройство в первую очередь для развлечения, а не для сложной работы.

Аппаратные особенности рассмотрены с точки зрения их API в *Qt*, обсуждены типовые способы их использования. Из интересных смежных тем, не отраженных в этом тексте, можно выделить использование и создание gestures, разного рода экранные клавиатуры и их заменители, пользовательские интерфейсы, ориентированные на сенсорные экраны, а также API для

подключения новых датчиков.

Список литературы

1. Инструкция для разработчика под Android. http://developer.android.com/guide/index..html
2. Android APIs. http://developer.android.com/reference/packages.html
3. http://developer.android.com/reference/android/hardware/SensorManager.html
4. http://developer.android.com/reference/android/location/LocationManager.html
5. Using Multi-Touch and Gestures with Qt. http://www.slideshare.net/qtbynokia/using-multitouch-and-gestures-with-qt
6. QTouchEvent Class Reference. http://doc.qt.nokia.com/latest/qtouchevent.html
7. Инструкция для разработчика под Android. http://developer.android.com/guide/index..html
8. Android APIs. http://developer.android.com/reference/packages.html
9. http://developer.android.com/reference/android/hardware/SensorManager.html
10. http://developer.android.com/reference/android/location/LocationManager.html