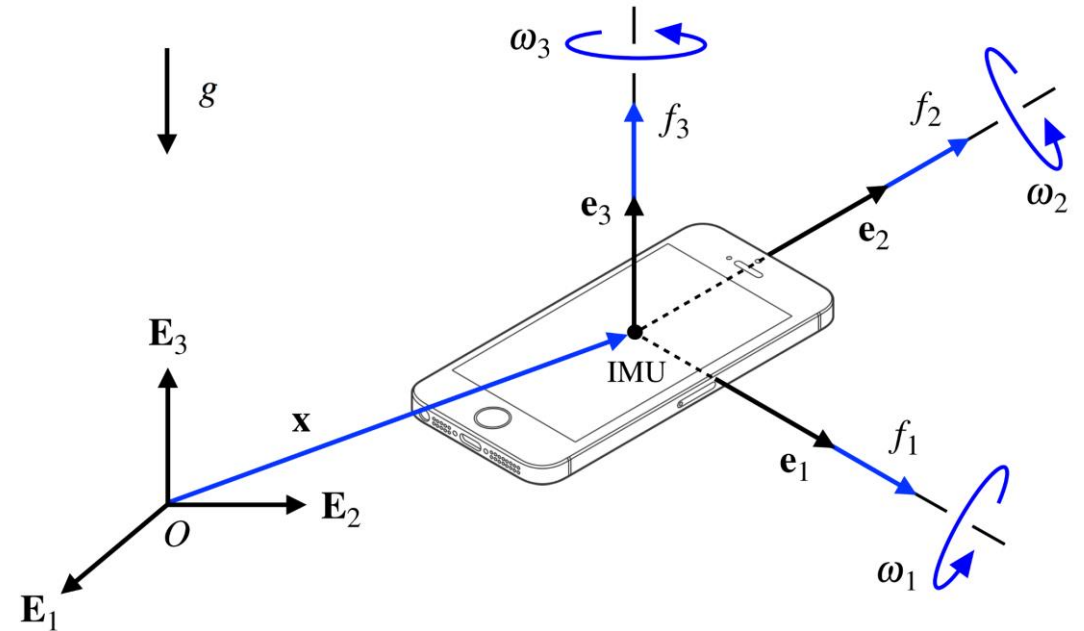


Работа с датчиками и API Android

Введение в работу с датчиками

Датчики играют ключевую роль в расширении функциональности мобильных приложений. Операционная система Android предоставляет разнообразные API для взаимодействия с датчиками устройства.

В этой лекции мы рассмотрим работу с датчиками, их типы, взаимодействие с устройствами и применение биометрии.



Основные типы датчиков

- **Акселерометр:** Измеряет ускорение устройства по трем осям.
- **Гироскоп:** Измеряет угловую скорость вращения устройства.
- **Магнитометр:** Измеряет магнитное поле, используется для определения направления.
- **Датчик освещенности:** Определяет уровень освещенности вокруг устройства.
- **Датчик приближения:** Определяет расстояние до ближайшего объекта.
- **Биометрические датчики:** Сканер отпечатков пальцев, распознавание лица и другие.



API для работы с датчиками

Android предоставляет API для работы с датчиками через `SensorManager`.

Основные классы и интерфейсы:

- **`SensorManager`**: Управляет доступом к датчикам.
- **`Sensor`**: Представляет конкретный датчик.
- **`SensorEvent`**: Содержит данные от датчика.
- **`SensorEventListener`**: Интерфейс для обработки событий датчиков.



Регистрация и отмена слушателя датчиков

Чтобы начать получать данные от датчика, необходимо зарегистрировать слушатель.

После завершения работы с датчиком важно снять регистрацию слушателя, чтобы избежать утечек памяти и снизить нагрузку на батарею.

Регистрация и снятие регистрации должны происходить в соответствующих методах жизненного цикла активности, например, **onResume()** и **onPause()**.

Пример: Регистрация и отмена слушателя

RegisterSensor.kt

Kotlin

```
val sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
val accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)

val sensorEventListener = object : SensorEventListener {
    override fun onSensorChanged(event: SensorEvent) {
        // Обработка данных от датчика
    }

    override fun onAccuracyChanged(sensor: Sensor, accuracy: Int) {
        // Обработка изменений точности
    }
}

sensorManager.registerListener(sensorEventListener, accelerometer,
    SensorManager.SENSOR_DELAY_NORMAL)
```

UnregisterSensor.kt

Kotlin

```
sensorManager.unregisterListener(sensorEventListener)
```

Пример: Получение и обработка данных от датчика

Метод **onSensorChanged()**
вызывается при каждом
изменении показаний датчика

HandleSensorsData.kt

Kotlin

```
override fun onSensorChanged(event: SensorEvent) {  
    val x = event.values[0]  
    val y = event.values[1]  
    val z = event.values[2]  
  
    // Логика обработки данных  
}
```

Практический пример: Шагомер на основе акселерометра

THRESHOLD установлено в 10.0, что соответствует минимальной величине ускорения, при котором считается, что пользователь сделал шаг. Порог может быть настроен в зависимости от устройства и условий использования

StepCounter.kt

Kotlin

```
private var stepCount = 0
private val THRESHOLD = 10.0

override fun onSensorChanged(event: SensorEvent) {
    val x = event.values[0]
    val y = event.values[1]
    val z = event.values[2]
    val magnitude = Math.sqrt((x * x + y * y + z * z).toDouble())

    if (magnitude > THRESHOLD) {
        stepCount++
    }
}
```


Управление энергопотреблением

Работа с датчиками может существенно влиять на энергопотребление устройства, особенно при постоянном получении данных.

Рекомендации по снижению энергопотребления:

- **Используйте минимально возможную частоту обновления данных:** Вместо **SENSOR_DELAY_FASTEST** используйте **SENSOR_DELAY_UI** или **SENSOR_DELAY_NORMAL**.
- **Отключайте ненужные датчики:** Если датчик не используется, снимайте регистрацию слушателя с помощью метода **unregisterListener()**.
- **Приостановите работу с датчиками, когда приложение не активно:** В методах **onPause()** Activity и Fragment'ов снимайте регистрацию слушателей датчиков, а затем регистрируйте их заново в **onResume()**.

Взаимодействие с устройствами

Датчики могут использоваться не только для сбора информации о состоянии устройства, но и для взаимодействия с внешними устройствами, такими как фитнес-трекеры, умные часы и другие устройства IoT (Интернет вещей).

Основные технологии для взаимодействия:

- **Bluetooth Low Energy (BLE):** Используется для низкоэнергетического подключения к внешним устройствам, например, для синхронизации данных с фитнес-браслетом.
- **NFC (Near Field Communication):** Применяется для обмена данными на коротких расстояниях, например, для быстрой авторизации или передачи данных.
- **Wi-Fi Direct:** Позволяет напрямую соединять устройства через Wi-Fi без необходимости в общей сети.

Пример: Подключение к BLE-устройству и обработка данных

BluetoothConnection.kt

Kotlin

```
val bluetoothAdapter: BluetoothAdapter? = BluetoothAdapter.getDefaultAdapter()
val deviceAddress = "00:11:22:33:44:55" // Адрес устройства
val device: BluetoothDevice? = bluetoothAdapter?.getRemoteDevice(deviceAddress)

val gattCallback = object : BluetoothGattCallback() {
    override fun onConnectionStateChange(gatt: BluetoothGatt, status: Int, newState: Int) {
        if (newState == BluetoothProfile.STATE_CONNECTED) {
            // Устройство подключено
            gatt.discoverServices()
        } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
            // Устройство отключено
        }
    }

    override fun onServicesDiscovered(gatt: BluetoothGatt, status: Int) {
        if (status == BluetoothGatt.GATT_SUCCESS) {
            // Сервисы обнаружены, можно начать взаимодействие
            val service = gatt.getService(UUID.fromString("0000180d-0000-1000-8000-00805f9b34fb"))
            val characteristic = service?.getCharacteristic(UUID.fromString("00002a37-0000-1000-8000-00805f9b34fb"))
            gatt.readCharacteristic(characteristic)
        }
    }

    override fun onCharacteristicRead(gatt: BluetoothGatt, characteristic: BluetoothGattCharacteristic, status: Int) {
        if (status == BluetoothGatt.GATT_SUCCESS) {
            // Данные успешно получены
            val data = characteristic.value
            // Пример обработки данных - преобразование байтов в строку
            val receivedData = data.joinToString(separator = " ") { byte -> "%02x".format(byte) }
            Log.d("BLE", "Received data: $receivedData")
        }
    }
}

device?.connectGatt(this, false, gattCallback)
```

Биометрические данные и их использование

Биометрия — это технология распознавания пользователей по уникальным биологическим характеристикам.

Android поддерживает несколько типов биометрии:

- Отпечаток пальца
- Лицо
- Радужная оболочка глаза (ограниченно)

Биометрия широко используется для аутентификации в приложениях, подтверждении действий и многому другому.



Пример: Работа с биометрией

BiometricAuth.kt

Kotlin

```
val biometricPrompt = BiometricPrompt(
    this,
    ContextCompat.getMainExecutor(this),
    object : BiometricPrompt.AuthenticationCallback() {
        override fun onAuthenticationSucceeded(result: BiometricPrompt.AuthenticationResult) {

            // Успешная аутентификация
            Log.d("Biometric", "Authentication succeeded")
        }

        override fun onAuthenticationFailed() {
            // Неудачная аутентификация
            Log.d("Biometric", "Authentication failed")
        }

        override fun onAuthenticationError(errorCode: Int, errString: CharSequence) {
            super.onAuthenticationError(errorCode, errString)
            // Обработка ошибки аутентификации
            Log.d("Biometric", "Authentication error: $errString")
        }
    }
)

val promptInfo = BiometricPrompt.PromptInfo.Builder()
    .setTitle("Biometric login")
    .setSubtitle("Log in using your biometric credential")
    .setNegativeButtonText("Cancel")
    .build()

biometricPrompt.authenticate(promptInfo)
```

Применение биометрии в приложениях

Примеры использования биометрии:

- **Мобильный банкинг:** вход в банковские приложения и подтверждения финансовых операций.
- **Электронная коммерция:** подтверждения транзакций
- **Доступ к личным данным:** вход и открытия доступа к конфиденциальным данным
- **Умные дома:** управление доступом к умным устройствам (дверные замки или системы сигнализации).

Преимущества использования биометрии:


- Повышенная безопасность
- Скорость и удобство
- Снижение рисков фишинга

Тестирование работы с датчиками

Тестирование работы с датчиками является ключевым этапом разработки мобильных приложений, особенно если приложение активно использует данные с нескольких датчиков.

Тщательное тестирование работы с датчиками позволит создавать более надежное и стабильное приложение, которое будет корректно работать в любых условиях эксплуатации.





Рекомендации по тестированию

- **Использование физических устройств:** Эмуляторы Android имеют ограниченные возможности по эмуляции работы датчиков.
- **Тестирование в различных условиях:** Датчики могут работать по-разному в зависимости от условий эксплуатации.
- **Учет энергопотребления:** При работе с датчиками важно тестировать приложение на разных уровнях заряда батареи.
- **Тестирование на различных устройствах:** Разные модели устройств могут иметь различные характеристики датчиков
- **Тестирование с учетом жизненного цикла приложения:** Убедитесь, что работа с датчиками корректно останавливается и возобновляется при изменении состояния активности



Инструменты для тестирования

- **Использование Logcat:** Вывод данных с датчиков в лог с помощью **Logcat** поможет отладить их работу и увидеть, как данные изменяются в реальном времени.
- **Инструменты автоматизированного тестирования:** Для автоматизации тестирования можно использовать инструменты, такие как **Espresso** или **UI Automator**, которые помогут проверить работу приложения с датчиками в различных сценариях.

Работа с несколькими датчиками одновременно

В некоторых приложениях требуется одновременное использование нескольких датчиков (определения ориентации устройства и отслеживания движений пользователя).

Основные моменты при работе с несколькими датчиками:

- **Синхронизация данных:** Обработка данных с нескольких датчиков должна быть синхронизирована, чтобы избежать некорректных вычислений или задержек.
- **Оптимизация производительности:** Одновременное использование нескольких датчиков может увеличивать нагрузку на процессор и потребление энергии.
- **Учет особенностей различных датчиков:** Датчики могут иметь разные частоты обновления и точности, что требует внимательного подхода к обработке их данных.

Пример: обработка данных от акселерометра и гироскопа

MultipleSensors.kt


Kotlin

```
val sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
val accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
val gyroscope = sensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE)

val sensorEventListener = object : SensorEventListener {
    override fun onSensorChanged(event: SensorEvent) {
        when (event.sensor.type) {
            Sensor.TYPE_ACCELEROMETER -> {
                val x = event.values[0]
                val y = event.values[1]
                val z = event.values[2]
                // Обработка данных от акселерометра
                Log.d("SensorData", "Accelerometer - X: $x, Y: $y, Z: $z")
            }
            Sensor.TYPE_GYROSCOPE -> {
                val rotX = event.values[0]
                val rotY = event.values[1]
                val rotZ = event.values[2]
                // Обработка данных от гироскопа
                Log.d("SensorData", "Gyroscope - rotX: $rotX, rotY: $rotY, rotZ: $rotZ")
            }
        }
    }

    override fun onAccuracyChanged(sensor: Sensor, accuracy: Int) {
        // Обработка изменений точности
    }
}

sensorManager.registerListener(sensorEventListener, accelerometer,
    SensorManager.SENSOR_DELAY_NORMAL)
sensorManager.registerListener(sensorEventListener, gyroscope,
    SensorManager.SENSOR_DELAY_NORMAL)
```



Современные технологии в работе с датчиками

- **Датчики отпечатков пальцев под экраном:** Позволяют размещать сканер отпечатков непосредственно под экраном устройства. Они работают на основе ультразвуковых или оптических технологий.
- **Лидары (Light Detection and Ranging):** Используются для точного измерения расстояний до объектов.
- **Улучшенные гироскопы и акселерометры:** Они обеспечивают более точное отслеживание движений и ориентации устройства.
- **Температурные и атмосферные датчики:** Эти датчики позволяют измерять температуру, влажность, давление и другие параметры окружающей среды.

Пример: использования лидаров для измерения расстояния в AR-приложении

ArDistance.kt

Kotlin

```
// Настройка AR-фрейма
arFragment.arSceneView.scene.addOnUpdateListener { frameTime ->
    val frame = arFragment.arSceneView.arFrame ?: return@addOnUpdateListener

    // Получение списка точек глубины
    val depthPoints = frame.getUpdatedTrackables(DepthPoint::class.java)

    for (point in depthPoints) {
        val distance = point.pose.tx() // Получение расстояния до объекта
        Log.d("ARDepth", "Distance to object: $distance meters")

        // Пример использования данных для добавления виртуального объекта
        if (distance < 2.0) { // Если объект близко
            // Логика для добавления виртуального объекта или взаимодействия
        }
    }
}
```



Заключение

Работа с датчиками на Android представляет собой важный и многогранный аспект разработки мобильных приложений. Датчики позволяют создавать приложения, которые взаимодействуют с физическим миром, обеспечивая уникальные возможности для пользователей.

От простых шагомеров до сложных приложений дополненной реальности — датчики помогают разработчикам воплощать в жизнь самые разнообразные идеи.



True Engineering

630128, г. Новосибирск,
ул. Кутателадзе, 4г

(383) 363-33-51, 363-33-50
info@trueengineering.ru
trueengineering.ru

**Новосибирский
Государственный
Университет**