Ostbayerische
Technische Hochschule
Amberg-Weiden

Studienarbeit Mobile & Ubiquitous Computing Sommersemester 2020

Dozent: Prof. Dr.-Ing. Ulrich Schäfer

Bearbeiter: Nicolas Proske

Abgabe: 07.07.2020

Gliederung

1. Analyse der ursprünglichen AccelerometerPlay-App
2. Beschreibung der Studienarbeit

1. Analyse der ursprünglichen AccelerometerPlay-App

Die ursprüngliche AccelerometerPlay-App nutzt den Beschleunigungssensor des Smartphones und wandelt mit Hilfe der Verlet-Methode die Beschleunigung des Geräts in eine Position um. Zur Darstellung dieser errechneten Positionen werden Eisenkugeln simuliert, welche sich auf einer dargestellten Holzplatte frei bewegen können. Dabei wird die Neigung des virtuellen Tisches vom Beschleunigungssensor des Smartphones gesteuert.

Der Code ist in vier größere Teile unterteilt. Darunter befindet sich die Hauptklasse mit **`onCreate`**, **`onResume`** und **`onPause`**, zwei View-Klassen **`SimulationView`** und **`Particle`** und eine weitere Klasse namens **`ParticleSystem`**.

Als Resource-Files zum Anzeigen der Partikel und des Hintergrundes werden die unter „**`res/drawable-hdpi/`**“ liegenden Dateien „**`ball.png`**“ und „**`wood.jpg`**“ benutzt. Das App-Icon ist unter „**`res/mipmap-XXXdpi/`**“ gespeichert und wird in der **`AndroidManifest.xml`** gesetzt.

Die **AndroidManifest.xml** legt fest, mit welchen Standardwerten die App gebaut und gestartet werden soll. Dort wurde unter anderem die Bildschirmorientierung, das Theme, die minimale SDK Version und die benötigten Berechtigungen zum einwandfreien Benutzen der App definiert. Da es sich bei dieser Applikation um ein Spiel handelt wurde das Theme **@android:style/Theme.NoTitleBar.Fullscreen** gewählt, sodass standardmäßig die Titlebar ausgeblendet ist und die App im Vollbildmodus gestartet wird. Die Berechtigungen **android.permission.VIBRATE** und **android.permission.WAKE_LOCK** bedeuten, dass zum einen das Gerät vibrieren darf und zum anderen, dass die Anwendung den Energiezustand des Host-Geräts – also das Smartphone – steuern darf. Somit wird verhindert, dass sich das Handy von selbst abschaltet und die App dadurch pausiert wird.

Um dies steuern zu können wird die PowerManager-Systemdienstfunktion namens Wake-Locks benötigt, welche in der **onCreate**-Methode der Hauptklasse definiert wird:

```
mWakeLock = mPowerManager.newWakeLock(PowerManager.SCREEN_BRIGHT_WAKE_LOCK,
getClass().getName());
```

Zudem werden dort alle globalen Variablen initialisiert, sodass diese überall im restlichen Code zur Verfügung stehen. Um beim App-Start etwas auf dem Bildschirm zu sehen wird ein neues **SimulationView**-Objekt angelegt und diesem das Hintergrundbild „**wood.jpg**“ aus dem **drawable**-Ordner zugewiesen. Anschließend wird dieses **SimulationView**-Objekt als Hauptview angegeben, sodass beim App-Start der Holztisch als Hintergrund angezeigt wird.

Die Methoden **onResume** und **onPause** sorgen dafür, dass beim Starten/Aufwecken bzw. Pausieren die Simulation der Eisenkugeln gestartet bzw. gestoppt wird. Wichtig ist hier in der **onResume**-Methode **mWakeLock.acquire()** aufzurufen, sodass der Bildschirm eingeschalten bleibt, da der Benutzer wahrscheinlich nicht viel auf dem Bildschirm herumtippen wird, denn die App basiert schließlich nicht auf Touch-Eingabe, sondern auf den Daten der Bewegungssensoren.

Die Klasse **SimulationView** und die dazugehörigen Unterklassen sorgen dafür, dass die Kugeln physikalisch korrekt angezeigt werden. Um dies zu gewährleisten werden anfangs alle benötigten Variablen definiert und im Konstruktor mit den korrekten Werten initialisiert. Die wichtigsten Variablen sind hierbei **mDstWidth**, **mDstHeight** und **sBallDiameter**. **mDstWidth-/Height** geben die Größe der Eisenkugeln abhängig von der Bildschirmgröße an. Dazu wird der Parameter **sBallDiameter** benötigt,

welcher den Durchmesser der Kugeln in Meter angibt. Daraus werden anschließend mit Hilfe der von Android zur Verfügung gestellten DisplayMetrics die Dots-Per-Inch (DPI) des Bildschirms entlang der x- & y-Achse ermittelt und zur Berechnung von Meter in Pixel benutzt. Zuletzt wird die Option „**inDither**“ festgelegt, sodass Bilder verbessert werden, indem eine Illusion von mehr Farben erzeugt wird.

Die Methode **onSizeChanged** bewirkt ein Neuberechnen des Ursprungs relativ zum Ursprung der gegebenen Bitmap.

Die Methode **onSensorChanged** prüft, ob sich die Bildschirmorientierung geändert hat. Das ist wichtig, da berücksichtigt werden muss, wie der Bildschirm in Bezug auf die Sensoren gedreht ist, da die Daten immer in einem Koordinatenraum zurückgegeben werden müssen, welches in seiner ursprünglichen Ausrichtung mit dem Bildschirm ausgerichtet ist. Wäre dies nicht der Fall, würden sich die Kugeln bei gedrehtem Bildschirm in ungewollte Richtungen bewegen.

Die Methode **onDraw** berechnet anhand der Daten des Beschleunigungssensors die Positionen aller Eisenkugeln neu und zeichnet diese mit Hilfe der **invalidate**-Methode auf die Oberfläche.

Die eigentlichen Eisenkugeln basieren auf der Klasse **Particle**, wobei ein „**Particle**“ eine Eisenkugel repräsentiert. Ein **Particle** besteht aus einer x- & y-Koordinate sowie einer x- & y-Velocity. In der Methode **computePhysics** wird anhand der x- & y-Daten sowohl die Position als auch die Geschwindigkeit der Eisenkugeln berechnet. Diese Methode wird ausgeführt, sobald die gesamten Kugeln neu berechnet werden. Zur Erkennung von Kollisionen mit dem Rand des Bildschirms wird der Verlet-Integrator in der Methode **resolveCollisionWithBounds** verwendet. Es muss sich lediglich eine kollidierende Eisenkugel so bewegen, dass die Einschränkungen, welche in dieser Methode festgelegt werden, erfüllt sind. Dazu werden die Maximalwerte mit der aktuellen Position der Kugel verglichen und entsprechend die Position & Velocity der Kugel geändert.

Die letzte Klasse **ParticleSystem** repräsentiert eine Sammlung von **Particles**, also eine Sammlung aller Eisenkugel - hier wird unter anderem die Anzahl der darzustellenden Kugeln festgelegt. Im Konstruktor der Klasse wird über die vordefinierte Anzahl der Partikel in der Variable **NUM_PARTICLES** iteriert und jeweils ein Partikel dazu angelegt und zur Haupt-View hinzugefügt.

Mit Hilfe des Verlet-Integrators wird in der Methode **updatePositions** die Position jeder Eisenkugel aktualisiert. Zur Berechnung wird die Methode **computePhysics** der

Klasse **`Particle`** verwendet, welche als Parameter die x- & y-Daten des Beschleunigungssensors benötigen. Diese Daten repräsentieren die Erdbeschleunigung je nach Neigungswinkel und sorgen so für einen Richtungsvektor mit entsprechender Beschleunigung, sodass sich die Kugel bewegt. Die **`updatePositions`**-Methode wird beim Ausführen der **`update`**-Methode aufgerufen. In dieser **`update`**-Methode ist zudem eine Kollisionserkennung der einzelnen Eisenkugeln implementiert. Dort wird jede Kugel gegen jede andere Kugel auf Kollision getestet. Falls eine Kollision erkannt wurde, wird die Kugel mittels einer virtuellen Feder mit unendlicher Starrheit von der anderen Kugel wegbewegt. Um das Ganze noch natürlich wirken zu lassen wird zudem eine gewisse Unordnung (Entropie) hinzugefügt.

2. Beschreibung der Studienarbeit

Anhand der gegebenen Aufgabenstellung habe ich mich dazu entschieden den ursprünglichen Holztisch mit Eisenkugeln durch einen Golfplatz mit Goldbällen zu ersetzen.

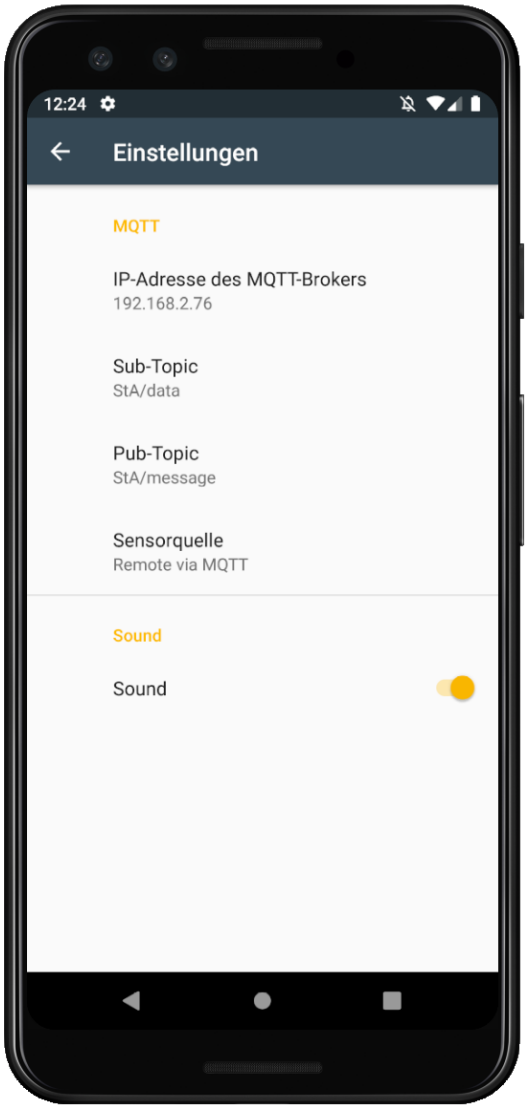
Das „Loch“, in das die Kugeln hineinfallen sollen habe ich jedoch etwas anders umgesetzt: Beim App-Start ist von Anfang an ein Kreis mit vordefinierter Größe vorhanden. Diesen Kreis kann man per Touch an eine beliebige Position auf dem Bildschirm verschieben.
Da der Kreis anfangs eine feste Größe besitzt, in der Aufgabenstellung der Studienarbeit jedoch steht, dass man diesen beim Erstellen beliebig groß „ziehen“ kann, ist es möglich den Kreis beim Klicken auf den von der BasicActivity erstellten FloatingActionButtom in kleinen Schritten zu vergrößern.

Der programmatische Aufbau des Android-Projekts meiner Studienarbeit folgt dem objektorientierten Ansatz und ist übersichtshalber in verschiedene Klassen unterteilt. Jede dieser Klassen ist jeweils für eine Hauptaufgabe zuständig und kümmert sich um deren Verwaltung.
Um Paho ins Projekt einbinden zu können wird die Jar-Datei benötigt, welche jeweils in **build.gradle (Module)** unter **repositories** und in **build.gradle (Project)** unter **dependencies** hinzugefügt wurde. Zudem wurde weiter oben in der **build.gradle (Module)** Datei die Java-Version festgelegt, sodass Lambda-Funktionen korrekt unterstützt werden:

```
compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}
```

In **AndroidManifest.xml** wurden die benötigten Berechtigungen für MQTT/Paho und die AccelerometerPlay-App hinzugefügt. Weiter unten wurde ebenfalls für Paho ein `<service>`-Tag erstellt, sodass der MqttService mittels Paho erkannt wird.

Die App besteht grundlegend aus zwei Activities – der **MainActivity** und der **SettingsActivity**. Die **MainActivity** ist die Hauptklasse und dient als Schnittstelle zwischen allen anderen Klassen. Hier werden in der **onCreate**-Methode sowohl alle Klassen instanziiert als auch alle Views und die dazu benötigten Ressourcen erstellt bzw. initialisiert. Die Hauptview ist dabei unter „**res/layout/activity_main.xml**“ zu finden. Diese besteht aus der Toolbar, einem **FloatingActionButton** und der restlichen freien View an sich. In der **MainActivity** wird zu dieser Hauptview die **SimulationView** angefügt, sodass unter der Toolbar das eigentliche Spiel angezeigt wird. In der **onResume**- & **onPause**-Methode wird festgelegt, was beim Starten bzw. Pausieren der App passieren soll – beim Starten muss sich die App zu dem in den Einstellungen hinterlegten MQTT-Broker verbinden und die ebenfalls in den Einstellungen hinterlegte Topic abonnieren, sodass Nachrichten empfangen werden können. Das Gegenteil passiert in der **onPause**-Methode. Der untere Teil der Klasse besteht aus **Gettern** und **Settern**, sodass auf die in der **MainActivity** angelegten Variablen von anderen Klassen aus zugegriffen werden kann.



Die **SettingsActivity** ist für die Anzeige und Verwaltung der Einstellungen zuständig. Dort wird sowohl die Settings-View „**res/layout/settings_activity.xml**“ als Content-View als auch über den **FragmentManager** die einzelnen Einstellungsmöglichkeiten wie Sound, Broker-IP oder die Sensorquelle gesetzt. Diese sind unter „**res/xml/root_preferences.xml**“ hinterlegt.

Um die Einstellungen beim Klicken auf das Toolbar-Segment „Einstellungen“ aufzurufen, wird in der **MainActivity** in der **onOptionsItemSelected**-Methode geprüft, ob das angeklickte Segment dem Einstellungsbutton entspricht und falls ja wird die neue Activity **SettingsActivity** gestartet:

```
if (id == R.id.action_settings) {
    final Intent intent = new Intent(this, SettingsActivity.class);
    startActivity(intent);
}
```

Um die Soundsteuerung kümmert sich die Klasse **AudioManager**. In den Einstellungen ist dafür ein Switch-Button hinterlegt, welcher die zwei Werte **true** (Sound an) oder **false** (Sound aus) entsprechen kann. Ist der Wert **true**, so wird über die Klasse **MainActivity** in der **onResume**-Methode die **unmute**-Methode der **AudioManager**-Klasse aufgerufen, sodass der Spieler alle Sounds hören kann. Falls der Wert **false** ist, wird die **mute**-Methode aufgerufen, welche dafür sorgt, dass alle Sounds auf lautlos gestellt werden.

Die Klasse **MQTTManager** kümmert sich um die Verbindung zu einem in den Einstellungen hinterlegten Broker. Über **SharedPreferences** werden im Konstruktor

die Variablen **broker**, **sub_topic** und **pub_topic** initialisiert. Falls der Index der dort angegebenen **SharedPreference** nicht existiert, wird auf einen **DefaultValue** zurückgegriffen.

Die **connect**-Methode baut eine Verbindung zum Broker auf. Falls der Verbindungsaufbau nicht innerhalb von fünf Sekunden möglich ist, wird der Verbindungsaufbauversuch abgebrochen und es kommt sowohl in der Konsole als auch auf dem Bildschirm eine Meldung, dass die Verbindung nicht hergestellt werden konnte.

Die **subscribe**-Methode abonniert die in den Einstellungen angegebene Sub-Topic. Sobald eine neue Nachricht empfangen wurde, wird die Nachricht mittels der **split**-Methode in zwei String aufgeteilt, zu einem **Float** konvertiert und als x- bzw. y-Beschleunigung gesetzt.

Die **publish**-Methode dient zum Senden einer als Parameter angebbaren Nachricht an den verbundenen Broker. Als Topic wird hier die Pub-Topic genommen, sodass das Python-GUI auf der „anderen Seite“ diese Topic abonnieren und die Nachricht empfangen kann. Falls dabei ein Fehler auftritt wird dieser in der Konsole ausgegeben.

Die **disconnect**-Methode deabonniert die Topic und trennt die Verbindung zum Broker. Auch hier wird beim Auftreten eines Fehlers eine Meldung in der Konsole ausgegeben.

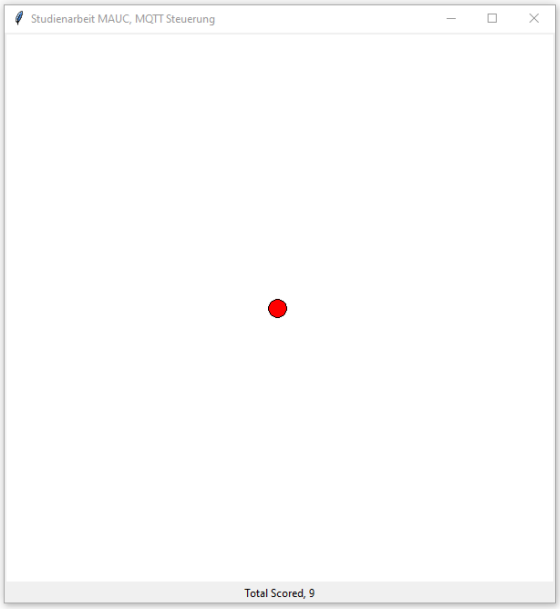
Um die korrekte Anzeige und die Animation der Eisenkugeln kümmert sich die Klasse **SimulationView**. Grundlegend wurden die meisten Dinge der ursprünglichen AccelerometerPlay-App übernommen, allerdings wurden die Variablennamen geändert und es wurde zudem hinzugefügt, dass beim Zeichnen der View ein Kreis in der Mitte des Bildschirms angezeigt wird. In der Aufgabenstellung der Studienarbeit wurde verlangt, dass man mittels Touch einen Kreis zeichnen soll, allerdings gab es damit ein paar Komplikationen, sodass ich mich nach langem Probieren dazu entschieden habe, es so zu lösen, dass von Anfang an ein kleiner Kreis mit vordefiniertem Radius auf dem Bildschirm angezeigt wird:

```
if (paintCircleX == 0f || paintCircleY == 0f) {
    paintCircleX = getWidth() / 2f; // X-Mitte des Views
    paintCircleY = getHeight() / 2f; // Y-Mitte des Views
}
canvas.drawCircle(paintCircleX, paintCircleY, paintCircleRadius, paintCircle);
```

Über das in der Klasse **SimulationView** erstellte **onTouchEvent**-Event kann man diesen Kreis über „Press & Hold“ auf dem Bildschirm beliebig verschieben. Um den Kreis zu vergrößern wurde der mit der **BasicActivity** standardmäßig erstellte **FloatingActionButton** verwendet. In der **onCreate**-Methode der **MainActivity** ist ganz unten ein **setOnClickListener** implementiert, der beim Klick auf den Button dafür sorgt, dass der Wert der Variable **paintCircleRadius** in der **SimulationView** um jeweils 10 erhöht wird. Anschließend wird ein **postInvalidate** ausgeführt, um die **SimulationView** neu zu zeichnen und den vergrößerten Kreis anzuzeigen.
Im **ParticleManager** hat sich bis auf zwei Kleinigkeiten nichts zur AccelerometerPlay-App geändert. Es wurde einerseits hinzugefügt, dass beim Updaten der Positionen in der **updatePositions**-Methode eine zusätzliche if-Abfrage dazugekommen ist. Diese überprüft, ob sich der aktuelle Partikel (also die Kugel) innerhalb des Kreises befindet. Falls dies der Fall ist, wird diese Kugel aus der **SimulationView** und der Liste aller Kugeln entfernt und der Score um eins erhöht. Abschließend wird über den **MqttManager** eine Nachricht mit den aktuell erzielten Treffern an den Broker gesendet. Andererseits wurde eine Änderung an dem **balls**-Array vorgenommen. Das Problem eines normalen Java-Arrays ist, dass es keine direkte Möglichkeit gibt, ein Objekt der Liste zu entfernen. Daher wurde das Java-Array **balls[]** mit einer **CopyOnWriteArrayList** ersetzt, welche als Listeninhalt ein **Particle**-Object verlangt. CopyOnWriteArrayLists sind gegenüber normalen ArrayLists zum einen performanter und haben den großen Vorteil, dass diese threadsicher sind.

Zum Steuern der Kugeln wird ein in Python programmiertes GUI verwendet. Um eine Verbindung zum Broker herzustellen kommt Paho als Schnittstelle zum Einsatz. Der Python-Code basiert auf der Vorlage des SenseHat-Mqtt-Projekts und wurde für die Zwecke der Studienarbeit angepasst.
Um einen Punkt unter der aktuellen Mauszeigerposition zu erstellen, wird in der **paint**-Methode zuerst einmal die aktuelle Mauszeigerposition in einer x- und y-Variable gespeichert. Anschließend wird das Oval gezeichnet – dazu wird auf die aktuelle x- und y-Position der Radius addiert bzw. subtrahiert, um die maximalen äußeren Werte zu erhalten. Zur Verschönerung des Kreises dient **outline** und **fill**, welche den Rand und das innere des Kreises mit der angegebenen Farbe einfärben. Um den Mauszeiger zu verstecken wird in der Tkinter-Config der Wert auf **cursor='none'** gesetzt.

Da die Android-App die errechnete Beschleunigung zum korrekten Berechnen der Kugeln erhalten muss, wird beim Bewegen der Maus in der **`paint`**-Methode ebenfalls der Wert der virtuellen Erdbeschleunigung anhand der x- und y-Position des Mauszeigers berechnet. Dazu muss erst einmal der Mittelpunkt des GUIs bestimmt und dessen Koordinaten auf 0, 0 gesetzt werden. Da die maximalen x- und y- Werte jeweils -4.905 und 4.905 entsprechen sollen, muss man die aktuellen Koordinaten von Pixel in numerische Werte zwischen -4.905 und 4.905 konvertieren.
Das GUI abonniert in der **`on_connect`**-Methode die in der Android-App eingestellten Sub-Topic, sodass Nachrichten über diese Topic empfangen werden können. Da die Android-App beim Treffen einer Kugel eine Nachricht an den Broker sendet, wird diese vom Python-Client entgegengenommen und in der **`on_message`**-Methode verarbeitet. Zunächst wird auf die global angelegte Variable **`message`** referenziert, welche das Label und den dazugehörigen Text enthält. Die erhaltene Nachricht vom Broker wird anschließend als Text des Labels gesetzt und auf der Konsole ausgegeben.



Die Studienarbeit wurde mit folgenden Versionen getestet:

Python:

- Python Interpreter: 3.8
- paho-mqtt: 1.5.0

Mosquitto:

- 1.6.10 (64-bit) (https://mosquitto.org/files/binary/win64/mosquitto-1.6.10a-install-windows-x64.exe)

Android Studio:

- Android-Studio: 4.0
- Emulator: Pixel 3 (1080x2160px), Android 9.0 (API-Version 28)
- Java: Version 1.8