WP Computergrafik für Android - Framework

Philipp Jenke[1]

[1]Hochschule für Angewandte Wissenschaften (HAW) Hamburg

I. Einführung

In diesem Dokument werden die grundsätzlichen Funktionalitäten des Frameworks beschrieben. Das Framework besteht aus vier Modulen:

- **shared:** gemeinsamer Code für alle anderen Module
- **opengl:** Testmodul, um OpenGL-basierte Apps zu entwickeln. Kein AR-Support.
- **vuforia:** Unterstützung der Vuforia-Funktionalität. In diesem Modul werden die Aufgaben für das Praktikum umgesetzt.
- **arcore:** experimentelles Modul zur Unterstützung der Google-ARCore-Technologie[1].

Eine App hat immer die folgende Stuktur:

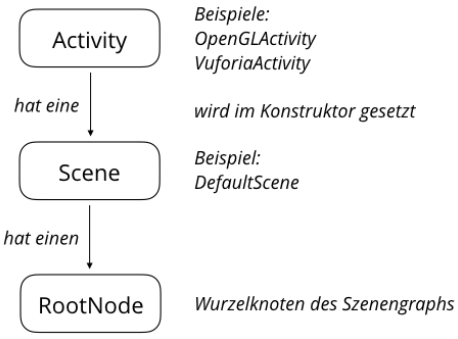

Abbildung 1. Grundsätzlicher Aufbau einer App im Framework.

II. Einrichtung

Das Framework kann von einem privaten Git-Repository mit der URL (*https://gitlab.informatik.haw-hamburg.de/wp_cg/wpcgar.git*) gekloned werden. Es ist Teil eines etwas umfangreicheren Repositories mit Frameworks für Java (Desktop), C# (Desktop) und eben diesem zur Android-Entwicklung. Diese Dokumentation bezieht sich nur auf den Android-Teil. Das Teilprojekt ist mit einem Gradle-Buildsystem konfiguriert. Die Entwicklung erfolgt über Android Studio (*https://developer.android.com/studio/index.html*). Zur Einrichtung sind folgende Schritte notwendig:

- Android Studio installieren (inklusive des Android SDKs, im Installer enthalten)
- Pfad-Variable *ANDROID_HOME* setzen (auf das Verzeichnis in dem das Android SDK installiert wurde)
- Framework aus dem Git-Repository klonen
- Android-Projekt in Android-Studio als Projekt öffnen

III. Projekte

Bei der Entwicklung mobiler Apps für Android stehen im Kern immer sogenannte *Activities*. In diesem Framework gibt es für jedes ausführbare Modul je eine *Activity*:

- opengl → *OpenGLActivity.java*
- vuforia → *VuforiaActivity.java*
- arcore → *ARCoreActivity.java*

[1]https://developers.google.com/ar/

Die Activities werden bei der Umsetzung eines neuen Projektes (beispielsweise bei der Bearbeitung einer neuen Praktikumsaufgabe kaum verändert. Jede der drei genannten Activities hat eine Objektvariable `scene` vom Typ `Scene`. Für ein neues Projekt implementiert man eine neue Szene, indem man eine Klasse schreibt, die von `Scene` erbt. Im Modul *ARCore* gibt es ein spezialisierte Variante der Szene, *ARCoreScene*, von der man erben muss. Die Szene verwaltet dabei alle relevanten Informationen, insbesondere den Szenengraph, dessen Wurzelknoten man in der Szene mit `getRoot()` bekommt. Die Integration der eigenen Szene ist denkbar einfach. Man instanziiert ein Objekt des eigenen Szenentyps und initialisiert damit die `scene`-Referenz im Konstruktor der *Activity*:

```
// Application specific scene
scene = new DefaultScene();
```

Bei der Implementierung einer eigenen Szenen-Klasse müssen insbesondere die folgende Methoden überschrieben werden:

- `setup()`: In dieser Methode baut man den Szenengraph auf.
- `timerTick`: In der Szene kann man einen Timer konfigurieren. Bei jedem Timeout des Timers wird diese Methode aufgerufen.
- `onSceneRedraw();()`: Diese Methode wird immer dann aufgerufen, wenn der Bildinhalt von OpenGL ES neu gezeichnet wird.

IV. Vuforia

Für die Entwicklung von AR-Anwendungen verwendet das Framework ein Marker-Tracking-System. Dieses ist durch eine externe Bibliothek (Vuforia) umgesetzt. Um mit Vuforia arbeiten zu können, muss man sich auf der Vuforia-Webseite (*https://developer.vuforia.com/*) als Entwickler (*Developer*) kostenlos registrieren. Dies hat zwei Gründe: Zum einen bekommt man damit eine ID, die im Code eingetragen wird, damit man die Bibliothek verwenden kann. Zum anderen generiert man Marker auf der Webseite aus Bildern, die dann in der Anwendung verwendet werden können.

Hat man eine ID generiert, dann wird sie an dieser Stelle im Code benötigt: Modul *vuforia* → Klasse *Session* → Methode *onSurfaceCreated*,

```
Vuforia.setInitParameters(mActivity,
  mVuforiaFlags, hier-die-ID-eintragen);
```

Im Framework wird an der Stelle auf `VuforiaKey.KEY` verwiesen. Sie müssen also zunächst eine entsprechende Klasse `VuforiaKey` im Modul *vuforia* anlegen und in der Klasse den Schlüssel als statisches Feld ablegen:

```
public static final String KEY =
  <copy key here>;
```

Auch die verwendeten Marker werden über die Vuforia-Developer-Seite verwaltet. Zum Erstellen eines Markers muss zunächst ein Bild ausgewählt werden. Bilder sind unterschiedlich gut geeignet. Beim Eintragen eines Bildes auf der Vuforia-Seite bekommt das Bild eine Sterne-Wertung. Optimalerweise hat das Bild fünf Sterne und ist damit optimal geeignet. Die Marker werden in einer sogenannten *Target Database* verwaltet, die dann in die Android-Studio Anwendung integriert wird. Dazu sind folgende Schritte notwendig:

- Anlegen einer *Target Database* auf der Vuforia-Developer-Seite unter *Develop* → *Target Manager*
- Einfügen eines oder mehrerer Bilder mit *Add Target*
- Exportieren der *Target Database* mit *Download Database (All)* = Zip-Datei
- Ablegen der entpackten ZIP-Datei im Ordner *assets* des Moduls *vuforia*

- Auflisten der XML-Datei aus der *Target Database* im Konstruktor der Klasse `AndroidActivity`: `mDatasetStrings.add("CGforAR.xml");`

Welcher der Marker aus einer *Target Database* in der Anwendung verwendet wird, wird über die `VuforiaMarkerNode` gesteuert. Als Identifikation dient der Name (*target name*). Die maximale Anzahl parallel getrackter Marker wird in der `VuforiaActivity` in dem Feld `MAX_NUM_TARGETS` festgelegt (Default: 2).

V. Dreiecksnetze

Dreiecksnetze werden im Framework durch das Interface `ITriangleMesh` beschrieben. Eine Implementierung des Interfaces ist bereits umgesetzt. In einem `TriangleMesh` werden die Vertices und Dreiecke als Index-Listen verwaltet. Ein Dreiecksnetz kann einfach durch das Hinzufügen von Vertices und Dreiecken erzeugt werden.

Eine Alternative für das Erzeugen von Dreiecksnetzen bietet die Factory-Klasse `TriangleMeshFactory`. Damit lassen sich einige einfache Körper wie Kugeln erzeugen.

Die mächtigste Möglichkeit, um Dreiecksnetzen zu generieren, ist das Einlesen mit dem `ObjReader`. Der ist in der Lage, Wavefront-OBJ-Dateien[2] (*.obj*) zusammen mit Materialinformationen (*.mtl*) einzulesen und daraus eine Liste von Dreiecksnetze zu generieren (teilweise werden mehrere Meshes aus einer OBJ-Datei erzeugt):

```
ObjReader reader = new ObjReader();
List<ITriangleMesh> meshes =
  reader.read("meshes/square.obj");
```

VI. Texturen

Texturen werden bei den Resourcen *src/main/res* im Verzeichnis *drawable* abgelegt. Üblicherweise werden PNG-Bilder oder JPG-Bilder verwendet. Die Auflösung der Bilder sollte so klein wie möglich sein (ohne Qualitätseinbußen).

Zur Verwaltung der Texturen steht die Singleton-Klasse *TextureManager* zur Verfügung. Diese verwaltet die Texturen über ihren Namen (Dateinamen)

```
Texture tex = TextureManager.getInstance().
  getTexture("lego");
```

Auf die Angabe der Endung (hier: .png") kann verzichtet werden. Wird ein Dreiecksnetz aus einer OBJ-Datei eingelesen, dann steht der Texturname in der zugehörigen Materialdatei. Bei der Verwendung von Buttons wird der Texturname als Argument an den Konstruktor übergeben.

VII. Buttons

Zur Interaktion der Anwenderin mit der App können auch Buttons umgesetzt werden. Dies sind rechteckige Felder auf dem Bildschirm. Bei einem Touch-Ereignis auf ein solches Button-Feld wird ein Ereignis ausgelöst, auf das reagiert werden kann. Buttons sind in der Klasse *Button* umgesetzt.

Zum Erzeugen eines Buttons benötigt man eine Textur. Außerdem gibt man die Position und Größe des Buttons auf dem Bildschirm an. Dazu wird das folgende Koordinatensystem verwendet: Die x-Achse verläuft horizontal und die y-Achse verläuft vertikal. Der Ursprung des Koordinatensystems ist die Mitte des Displays. Auf beiden Achsen ist der Bereich von -1 bis 1 abgedeckt. Die linke untere Ecke des Displays hat die Koordinaten $(-1, -1)$ und die rechte obere Ecke des Koordinatensystems hat die Koordinaten $(1, 1)$. Der folgende Button

```
Button button = new Button(
  "button.png",
   -1, -0.8, 0.3,
   handler);
```

hat seine linke obere Ecke bei $(-1, -0.8)$ und eine Breite von 0.3. Die Höhe wird automatisch so angepasst, dass der Button quadratisch erscheint.

Zur Umsetzung des Handlers implementiert man das Interface *ButtonHandler*. Dies kann beispielsweise über eine innere Klasse

```
new ButtonHandler() {
  @Override
  public void handle() {
    Log.i(Constants.LOGTAG,
      "Button 2 pressed!");
  }}
```

oder einen Lambda-Ausdruck erfolgen:

```
() -> Log.i(Constants.LOGTAG,
  "Button 1 pressed!")
```

[2] https://de.wikipedia.org/wiki/Wavefront_OBJ