AR Core

Augmented Reality (AR) erweitert die physische Welt mit digitalen Elementen, welche in die reale Welt projiziert werden. ARCore[1] ist das von Google entwickelte SDK, welches die fundamentalen APIs für AR zur Verfügung stellt. ARCore verwendet OpenGL zur Darstellung der AR-Elemente. Um dies zu vereinfachen, wird die SDK SceneView verwendet, welche vereinfachtes Rendering mit Filament ermöglicht. Diese Seminararbeit, zusammen mit der Demo-App «CoreShow»[2], soll die Funktionalitäten von ARCore, zusammen mit SceneView untersuchen und beide SDKs und deren Funktionsweise genauer erklären. Dabei werden Code-Implementationen aufgegriffen und eine Demo-App mit folgenden Beispielen implementiert: platzieren von virtuellen Elementen, Bilderkennung und Objekterkennung. Die Demo-App soll dabei auf Jetpack Compose basieren, um zusätzlich herauszufinden, wie gut es mit ARCore kompatibel ist.

AR & ARCore Fundamentals

Zu Beginn ein kurzer Überblick der wichtigsten Begriffe im Thema ARCore und AR:

Bewegungserkennung (Motion Tracking)
ARCore nutzt die Sensoren wie Kamera und Gyroskop, um die Bewegungen im Raum zu erkennen. So bestimmt ARCore die Position und Orientierung des Geräts in Echtzeit. Dieser Prozess wird auch SLAM (Simultaneous Localization and Mapping) genannt.

Environmental Understanding
Durch das Analysieren der Kamerabilder können Oberflächen wie Böden, Wände und Tische erkannt werden. Diese Oberflächen werden als «Planes» bezeichnet. Zusammen mit dem Motion Tracking entsteht dadurch ein Verständnis für die Räume, in denen sich die User der App aufhalten.

Erkennung der Lichtverhältnisse (Light Estimation)
ARCore analysiert die Lichtverhältnisse im Raum, um virtuelle Objekte realistisch auszuleuchten. So wirken sie besser integriert und die App kann den Nutzer auf ungünstige Lichtverhältnisse hinweisen.

Tiefenerkennung (Depth Perception)
ARCore nutzt die Kamera des Geräts in Kombination mit anderen Sensoren und Softwarealgorithmen, um die Tiefeninformationen der Umgebung zu erfassen. Diese Informationen ermöglichen eine realistische Platzierung von virtuellen Objekten im Raum.

[1] https://developers.google.com/ar/develop
[2] https://gitlab.switch.ch/hslu/edu/bachelor-computer-science/moblab/projects/2024/2024-sa36-arcore

Anchors
Ein Anchor in ARCore ist ein Punkt im Raum, der an eine bestimmte Position in der physischen Welt gebunden ist. Anchors werden verwendet, um virtuelle Objekte in der AR-Szene zu platzieren, sodass diese auch «*verankert*» bleiben.

Hello AR!

Nach Einleitung der wichtigsten Begriffe im Thema Augmented Reality folgt in diesem Abschnitt mehr zu ARCore. Die Google Dokumentation[3] ist dabei der Startpunkt zur Entwicklung einer ersten AR-App. Dazu empfiehlt sich auch das Builden und Testen der Sample-Apps auf dem eigenen Android-Phone. Diese Sample-Apps sind in der SDK[4] von ARCore enthalten. Sie benutzen OpenGL[5] zur Darstellung und Rendering von Modellen. Um anzufangen, empfiehlt Google «hello_ar_kotlin» (im Ordner der Sample Apps). Diese Beispiel-App besteht aus drei Kotlin Files:

HelloArView: Enthält die UI-Elemente. Hier werden Abfragen bezüglich der AR-Funktionen als UI-Elemente implementiert. Darunter beispielsweise ein Pop-Up Fenster mit der Frage, ob die Erkennung der Tiefenebenen («Depth») ein- oder ausgeschaltet werden soll.

HelloArRenderer: In dieser Klasse wird die Darstellung der AR-Szene vorbereitet. Darunter wird das 3D-Objekt im Sample Ordner initialisiert. Es sind auch Berechnungen implementiert, die das Laden von Shadern oder Texturen des 3D-Modells vorgeben sowie die allgemeine Konfiguration des Rendering-Prozesses. In diesem kleinen Code-Ausschnitt wird die Verwendung von OpenGL deutlich:

```
override fun onSurfaceCreated(render: SampleRender) {
    val buffer: ByteBuffer =
      ByteBuffer.allocateDirect(dfgResolution * dfgResolution * dfgChannels * halfFloatSize)
    GLES30.glTexImage2D(
      GLES30.GL_TEXTURE_2D      // (more code… )
}
```

HelloArActivity[6]: Ist die MainActivity. In dieser Klasse wird, wie in den Google Docs vorgesehen, eine «ARCore Session» aufgesetzt. Dabei wird geprüft, ob ARCore korrekt installiert ist. Hier ist auch die Light Estimation implementiert. Zudem werden die vorhin erwähnten HelloArView und HelloArRenderer Klassen instanziiert. Diese App erlaubt es, das mit OpenGL vordefinierte Modell in der Umgebung zu platzieren. In zu dunklen Umgebungen wird eine Benachrichtigung eingeblendet und es ist ebenfalls möglich, das Tiefenbild über einen Button zu aktivieren.

[3] https://developers.google.com/ar/develop/getting-started
[4] https://github.com/google-ar/arcore-android-sdk
[5] https://www.opengl.org/
[6] https://github.com/google-ar/arcore-android-sdk/blob/main/samples/hello_ar_kotlin/app/src/main/java/com/google/ar/core/examples/kotlin/helloar/HelloArActivity.kt

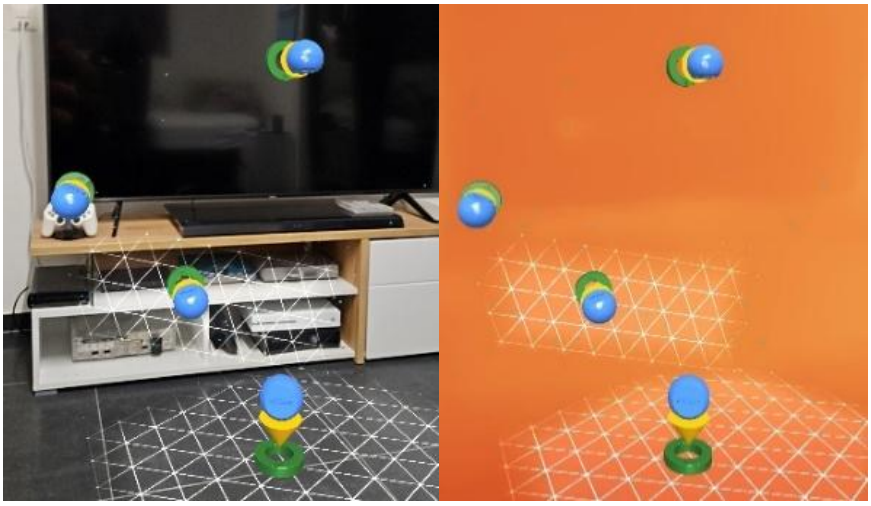

Abbildung 1: Bildschirmaufnahme Hello AR App mit und ohne Tiefenbild

Was bietet ARCore als SDK an und was ist möglich?

ARCore bietet grundlegende Funktionen an, welche das Platzieren von virtuellen Objekten ermöglichen. Darunter eine der essenziellen Funktionen: die Erkennung der Oberflächen («Planes»). Solche Funktionen werden innerhalb der instanziierten ARSession verwaltet:

```
// Configure the session, using Lighting Estimation, and Depth mode.
  fun configureSession(session: Session) { ❶
    session.configure(
      session.config.apply {
        lightEstimationMode = Config.LightEstimationMode.ENVIRONMENTAL_HDR ❷
        // Depth API is used if it is configured in Hello AR's settings.
        depthMode =
          if (session.isDepthModeSupported(Config.DepthMode.AUTOMATIC)) { ❸
      // …. for full code see hello_ar_kotlin App
}
```

Innerhalb der Session ❶ können verschiedene Einstellungen für die App vorgenommen werden, wie beispielsweise die Light Estimation ❷ oder eine Überprüfung, ob das Smartphone mit der Tiefenerkennung kompatibel ist ❸. Sollte ein Gerät dies nicht unterstützen, muss die App entsprechend reagieren und dies den Usern anzeigen können. ARCore ist weiterhin funktional, jedoch eingeschränkt und etwas ungenauer in der Erkennung von Flächen.

Mit diesen Optionen ist es möglich, virtuelle Objekte in der physischen Welt auf Ebenen mit korrekter Beleuchtung zu platzieren (siehe Abbildung 1). Um mit ARCore erfolgreich 3D-Objekte zu platzieren, ist nicht nur das Verständnis der ARCore API gefragt, sondern auch die Programmierung in OpenGL. Dies zeigt sich in der hello_ar_kotlin App beispielsweise in der Verwendung von Shadern und Buffern zur Darstellung eines 3D-Modells (siehe auch HelloArRenderer File der hello_ar App). Der Fokus auf OpenGL-spezifische Details lenkt von den eigentlichen ARCore-Konzepten ab und kann das Verständnis der ARCore-Funktionalitäten erschweren.

Abhilfe durch SceneView (Filament)

Filament[7] und SceneView[8] sind Alternativen, welche den Einstieg in die AR-Entwicklung und das Rendering von Objekten vereinfachen. Es ist somit ebenfalls möglich, AR-Anwendungen ohne Kenntnisse über OpenGL zu entwickeln. Nun folgt eine Analyse von SceneView, sowie deren Möglichkeiten und Grenzen in der AR-Entwicklung. Auch die Demo-App CoreShow basiert auf SceneView.

Filament ist Googles eigene Rendering-Engine, welche spezifisch für mobile Geräte entwickelt wurde. Diese Engine wurde explizit entwickelt, um die Hürde des OpenGL zu überwinden. Diese Engine erlaubt es, die Entwicklung eine Abstraktionsebene höher anzusetzen.

SceneView ist ein Open-Source-Projekt, welches das eingestellte SceneForm[9] von Google ersetzt. SceneView basiert auf Filament und ARCore und erlaubt ebenfalls eine vereinfachte Implementation zur Darstellung von AR und 3D-Elementen. SceneView wird von freiwilligen Entwicklern auf GitHub aktuell gehalten und bietet gegenüber dem veralteten SceneForm auch aktuelle Gradle-Abhängigkeiten. Da SceneView ein Community-Projekt ist, wird es in den Google Dokumentationen nicht erwähnt. Stattdessen wird noch auf das *eingestellte* SceneForm verwiesen. Dennoch erleichtert SceneView die Entwicklung von AR-Apps, wie nachfolgend genauer aufgezeigt wird. Erwähnenswert ist, dass SceneView auf der ARCore SDK basiert und mit Filament die AR-Entwicklung erweitert. Viele Funktionen und Prinzipien sind ähnlich wie in ARCore. Beispielsweise die Implementierung der bereits erwähnten *configureSession* kann vollständig für SceneView übernommen werden.

SceneView vs. ARCore Implementation

Die Sample App[10] von SceneView auf GitHub verfolgt ein ähnliches Prinzip wie die hello_ar_kotlin App. Im Vergleich dazu kann ein «.glb» oder «.fbx» 3D-Modell importiert und in der AR-Szene platziert werden, ohne diese Modelle mit OpenGL vorgängig definieren zu müssen.

Im Beispiel der «Ar-Model-Viewer-Compose» App von SceneView, wird die *ARScene()* implementiert. Darin sind ebenfalls, wie bei ARCore, Einstellungen vorzunehmen wie die Depth Mode oder Light Estimation Mode. Der zu implementierende Code[11] unterscheidet sich vor allem durch das Weglassen der OpenGL Implementation gegenüber der hello_ar_kotlin App. Während bei der hello_ar_kotlin App von ARCore die App in drei Dateien aufgeteilt wurde, so reicht bei SceneView das Einbinden der Dependency und Instanziieren der *ARScene()* (sowie der Composable, falls mit Jetpack Compose gearbeitet wird). Innerhalb dieser ARScene können wichtige AR-Parameter, wie der *InstantPlacementMode* oder die *sessionFeatures*, direkt konfiguriert werden. Der Vorteil hierbei ist, dass beim Weglassen dieser Features, die von SceneView vordefinierten

[7] https://google.github.io/filament/
[8] https://github.com/SceneView
[9] https://developers.google.com/sceneform/develop
[10] https://github.com/SceneView/sceneview-android/blob/main/samples/ar-model-viewer-compose/src/main/java/io/github/sceneview/sample/armodelviewer/compose/MainActivity.kt
[11] https://github.com/SceneView/sceneview-android?tab=readme-ov-file#ar---arscene-filament--arcore

Standardwerte übernommen werden. Weitere optionale Callbacks, wie *onSessionCreated*, *onSessionResumed* und *onSessionPaused*, ermöglichen die Reaktion auf verschiedene Ereignisse im Lebenszyklus der AR-Sitzung. Auch die Fehlerbehandlung wird vereinfacht, z.B. durch den *onTrackingFailureChanged* Callback, der über Tracking-Probleme informiert.

Mit der abschliessenden *createAnchorNode()* Funktion wird das Modell gerendert und in der Szene platziert. In dieser Funktion sind die Grösse des Modells definierbar oder die Sichtbarkeit der Bounding Box einstellbar.

«CoreShow» Demo-App

Die CoreShow Demo-App auf GitLab[12] demonstriert verschiedene Funktionen von SceneView kompakt in einer App. Diese App kann als Ergänzung zu dieser Arbeit betrachtet werden, um direkt ein visuelles Beispiel zum Text zu erhalten.

Google und SceneView setzen ebenfalls gezielt auf ihre Sample Apps, um Konzepte zu erklären. Dies bedeutet einerseits, dass es implementierten Code gibt, andererseits, dass oft weiterführende Erklärungen fehlen. In der CoreShow App wird mit Jetpack Compose gearbeitet, im Gegensatz zu den meisten Sample Apps, welche noch mit dem XML-Layout arbeiten. So kann auch herausgefunden werden, wie gut eine Entwicklung auf Jetpack Compose umsetzbar ist. Die Demo-App demonstriert drei Themen, welche auch im Rahmen dieser Arbeit aufgefasst werden: virtuelles Platzieren, Augmented Images und Object Detection.

1. Virtuelles Platzieren von 3D-Elementen mit SceneView

In diesem Abschnitt wird erklärt, wie die Plane Detection und das Platzieren von Objekten in SceneView funktioniert. Dabei werden einige bereits erklärte Konzepte aus dem vorherigen Kapitel wieder aufgegriffen. Die AR-Session der Demo-App wird durch das Implementieren der *ARScene()* ermöglicht. Hierbei schlägt Android Studio bereits während dem Tippen vor, welche Modifiers und Funktionen implementiert werden können. Weitere Methoden und Optionen können in der SceneView API[13] gefunden werden. Im «*sessionConfiguration*» Callback können sämtliche weitere Optionen für die AR-Szene konfiguriert werden, wie die bereits erwähnte Depth-Mode, Licht Erkennung oder das automatische Platzieren von Elementen. Mit dieser Vielfalt an Optionen scheint es sehr verlockend, alle möglichen Funktionen und Optionen zu aktivieren. Doch es gilt zu bedenken, dass auch im Jahre 2024 die AR-Funktionen noch nicht perfekt funktionieren, und je mehr Optionen aktiviert werden, desto mehr Rechenleistung wird dem Smartphone abverlangt. Resultat: die Erkennung der Planes dauert länger als üblich, oder die platzierten Objekte werden nicht sauber und realitätsnah in die Szene gesetzt. Deshalb ist hier abzuwägen, welche Funktionalitäten im Kontext einer App tatsächlich notwendig sind.

[12] https://gitlab.switch.ch/hslu/edu/bachelor-computer-science/moblab/projects/2024/2024-sa36-arcore
[13] https://sceneview.github.io/api/sceneview-android/arsceneview/io.github.sceneview.ar.arcore/-a-r-session/index.html

In der Demo kann die Szene komplett «geleert» werden, was bei vielen AR-Apps oft nicht möglich ist. Dies wird anhand eines «remember State»[14] erledigt. Beim Platzieren eines oder mehrerer virtueller Objekte wird «remember» auf «true» gesetzt. Dabei ist jedes platzierte Element ein sogenanntes «ChildNode». Jeder Node hat seine eigene Position in der physischen Welt. Mit Drücken des «Clear Scene» Buttons werden die ChildNodes geleert und somit die Szene wieder befreit. Dank dem remember State wird auch kein Objekt erneut automatisch platziert. Ähnlich wie im hello_ar_kotlin Beispiel, ist bei der CoreShow App das Platzieren eines Bugdroids möglich. Dieser hat etwa die Grösse einer Hand. Das Modell kann beispielsweise auf dem Schreibtisch platziert werden. Dank der bereits integrierten Optionen von ARCore ist es möglich, die Hand hinter dem virtuellen Objekt zu platzieren. So sieht es aus, als könnte der Bugdroid gestreichelt oder in die Hand genommen werden.

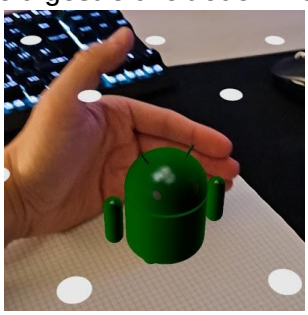

Abbildung 2: Platzieren eines Bugdroiden in der CoreShow App

Beim Drücken des «Stift»-Symbols wechselt das virtuelle Objekt. Die Tiefenerkennung ermöglicht es, sich hinter die virtuellen Objekte zu bewegen, sodass es perspektivisch korrekt wirkt. Die Illusion bricht jedoch, wenn man sich direkt davor positioniert. Dies wird «Okklusion»[15] genannt und ist standardmässig nicht aktiviert. Eine Anleitung zur Aktivierung und Implementierung von Okklusion befindet sich in den Google-Dokumentationen.[16] Die Integration dieser Tiefenfunktion ist dabei jedoch nicht über eine einfache Änderung der *configureSession*-Methode möglich. Aufgrund Googles Entscheidung, SceneForm aufzugeben, stoppte die Entwicklung auch in Bezug der Implementierung der Okklusion. Um die Okklusion zu aktivieren, müsste eine komplette Implementierung auf Basis von ARCore, OpenGL Shadern und Depth Maps durchgeführt werden. SceneView erlaubt durch Benutzen der *ARCore()*[17] Funktion das Aufrufen von ARCore Elementen. Ob dies auch im Falle der Implementation der Okklusion funktioniert, müsste näher geprüft werden.

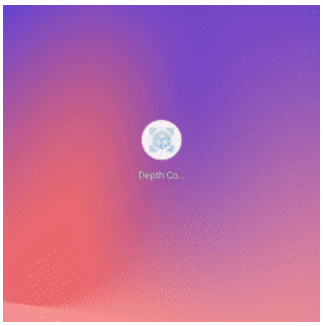

Abbildung 3: GIF aus dem Depth-Codelab von Google

[14] https://developer.android.com/develop/ui/compose/state-saving
[15] https://kumarahir.medium.com/occlusion-in-augmented-reality-2090911b3da1
[16] https://developers.google.com/ar/develop/depth
[17] https://sceneview.github.io/api/sceneview-android/arsceneview/io.github.sceneview.ar/index.html

2. Augmented Images (ARCore vs. SceneView)

Eine weitere Funktionalität, welche ARCore zur Verfügung stellt, sind die «Augmented Images». Also das Erkennen eines Bildes und die darauffolgende Darstellung eines virtuellen Objektes. In Googles Dokumentationen und Codelabs[18] ist es möglich, basierend auf einem Bild einer Weltkugel, ein 3D-Labyrinth (Maze) darzustellen. Zur Erstellung einer solchen Augmented-Image App ist ein Referenzbild notwendig. Dieses muss eindeutige Merkmale besitzen, sogenannte Feature Points. Diese Merkmale in einem Bild werden von der Software analysiert (Feature Extraction)[19]. ARCore stellt auch ein «arcoreimg» Tool[20] zur Verfügung, welches die Eignung des Bildes für die Augmented Reality-Nutzung angibt. Dies geschieht durch ein Bildqualitätsbewertungssystem, welches die Merkmale des Bildes analysiert und die Eignung in einer Skala von 0 bis 100 ausgibt.

```
val config = Config(arSession)
config.augmentedImageDatabase = createAugmentedImageDatabase(arSession) ❶
arSession.configure(config)
```

Um in ARCore die Session für Augmented Images zu konfigurieren, wird die bestehende Session mit der Datenbank der Augmented Images erweitert, bzw. konfiguriert ❶:

```
override fun onUpdate(frame: Frame) {
    for (augmentedImage in frame.getUpdatedTrackables(AugmentedImage::class.java)) {
        when (augmentedImage.trackingState) {
            TrackingState.TRACKING -> {
                // place 3D-Object
                placeObject(augmentedImage) ❶
            }
    }
```

Danach muss das Referenzbild zur Datenbank hinzugefügt werden. Nun ist die ARSession bereit, bei der Erkennung des Bildes ein Objekt zu platzieren ❶ (anhand dem obigen, vereinfachten Code-Beispiel).

Die Implementierung von Augmented Images mit der nativen ARCore SDK erfordert ebenfalls OpenGL für die Darstellung von 3D-Objekten. Google hat dies im Codebeispiel wieder durch die Implementierung einer speziellen Rendering-Klasse[21] gelöst. In der CoreShow App wurden als Unterschied dazu die Augmented Images mit SceneView implementiert. Dazu wird die *ARSceneView()* Klasse implementiert. Diese Klasse greift direkt auf die nativen ARCore Funktionen zurück, weshalb auf das Aufsetzen der *ARScene()*

[18] https://codelabs.developers.google.com/codelabs/augimg-intro#0
[19] https://www.researchgate.net/publication/338283777_Comparative_Analysis_of_the_Feature_Extraction_Performance_of_Augmented_Reality_Algorithms
[20] https://developers.google.com/ar/develop/augmented-images/arcoreimg
[21] https://github.com/google-ar/codelab-augmented-images-intro/blob/master/arcore-android-sdk-1.23.0/samples/augmented_image_java/app/src/main/java/com/google/ar/core/examples/java/common/rendering/ObjectRenderer.java

verzichtet werden kann (aus dem vorherigen Kapitel). Stattdessen kann im *configureSession{}* Callback folgende Funktion aufgerufen werden:

```
config.addAugmentedImage( ❶
    session, "mario", ❷
    context.assets.open("augmentedImages/mario.jpg") ❸
        .use(BitmapFactory::decodeStream)
)
```

Mit der *addAugmentedImage* ❶ Funktion wird bereits eine Session instanziiert. Wobei der Session einen Namen als String gegeben wird ❷ und das Referenzbild als Asset mitgegeben wird ❸. Anschliessend ist es wieder möglich, etliche Konfigurationen durchzuführen, wie die *lightEstimationMode* oder die *depthMode*. Im *onSessionUpdated{}* Callback wird die Session aufgerufen. Darin kann die Grösse des virtuellen Objektes in Float angegeben werden (*scaleToUnits*), sowie der Pfad. Als Standard wird das 3D-Modell immer im Zentrum des erkannten Bildes projiziert. Dies kann nach Belieben angepasst werden, wobei dabei mit dem kartesischen Koordinatensystem gearbeitet wird.

```
.apply {
    position = Position(x = 0.0f, y = -0.05f, z = 0.0f)
    rotation = Rotation(x = -90.0f, y = 0.0f, z = 90.0f)
}
```

Bei den Augmented Images ist es entscheidend, ob das Referenzbild immer an derselben Stelle ist (beispielsweise ein gedrucktes Bild an einer Wand), oder auf verschiedenen Bildschirmen angezeigt wird. Dies wird im Beispiel der CoreShow App mit dem Mario-Beispiel deutlich. Da die Grösse mit *scaleToUnits* festgelegt ist, könnte die Projektion zu gross oder zu klein sein. Dazu gibt es einige Foto- und Video-Beispiele im Demo-Media Ordner im GitLab der CoreShow App. Um mit Jetpack Compose eine Augmented Images Session starten zu können, muss *AndroidView()* eingesetzt werden. Die Einfachheit durch SceneView liegt vor allem darin, beliebig viele und verschiedene virtuelle Objekte zu laden und zu testen, ohne sich um das Rendering kümmern zu müssen.

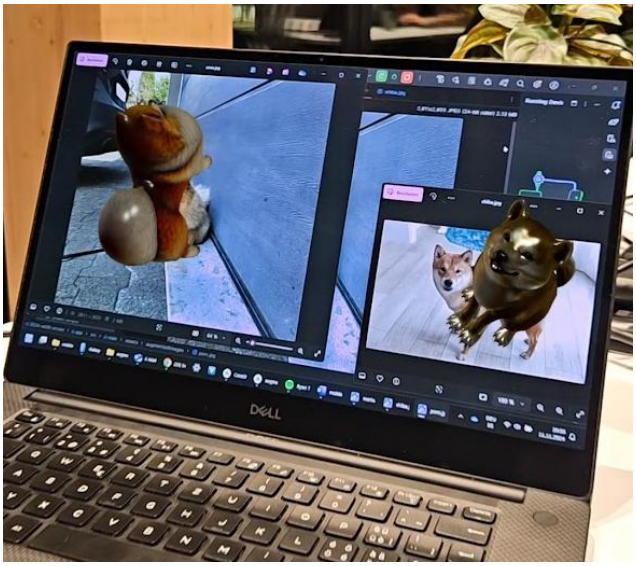

Abbildung 4: virtuelle 3D-Hunde werden gleichzeitig beim Erkennen des Referenzbildes angezeigt

3. Objekterkennung – leichter gesagt als getan

Im folgenden Kapitel wird das Thema Objekterkennung näher betrachtet. Gemäss den Google Dokumentationen ist eine Integration der Objekterkennung möglich, indem der Bild-Stream der Kamera in eine Machine Learning (ML) Pipeline integriert wird. Das Abrufen des Bild Streams erlaubt ARCore bereits standardmässig[22]. Nebst dem Bild Stream wird zur Integration einer Objekterkennung nicht nur die SDK von ARCore benötigt, sondern ein weiteres Framework von Google: MLKit[23]. Die dazugehörige «VisionAPI» [24] erlaubt das Implementieren mehrerer Funktionen: Text- und Bilderkennung, Gesichtserkennung und eben auch die Objekterkennung und das Tracking der erkannten Objekte. Dabei ist der Unterschied zwischen der Objekterkennung (Object Detection) und der Object *Recognition* zu beachten. Bei der Object Recognition handelt es sich um die Klassifizierung und Bezeichnung von Objekten. Dazu müsste Googles MLKit zusammen mit der VisionAPI implementiert werden. Dazu kommt der API-Key für die Vision Dienste, welcher, je nach Anwendungsfall, nicht vollständig kostenlos von Google zur Verfügung gestellt wird[25]. Des Weiteren, funktioniert das MLKit offline, die VisionAPI ist vollständig von den Google Servern abhängig. Dabei ist zu erwähnen, dass auch ein eigenes Modell erlernt werden kann, welches offline in der App integriert werden kann[26].

[22] https://developers.google.com/ar/develop/java/depth/developer-guide
[23] https://developers.google.com/ml-kit
[24] https://cloud.google.com/vision?hl=en
[25] https://cloud.google.com/vision/pricing
[26] https://developers.google.com/ml-kit/custom-models

Abbildung 5: Beispiel einer Kombination von ARCore, MLKit und VisionAPI

Implementierung von MLKit in der CoreShow Demo App

Aufgrund der Komplexität von MLKit und Vision API, könnte diesem Thema ein eigener Bericht gewidmet werden. Um dem Umfang dieser Arbeit gerecht zu werden, fokussieren sich die folgenden Abschnitte, sowie die Demo App, auf die Object *Detection* mit MLKit, ohne Vision API. Ziel ist es, ein Objekt zu erkennen und mit ARCore, bzw. SceneView (AR), eine Bounding Box zu projizieren. Anhand der Google Developers Seite[27] wird das Implementieren von MLKit erklärt. In der vorliegenden Demo-Applikation wurde auf die Klassifizierung von Objekten verzichtet, sodass auf das Konfigurieren eines Models (mittels Firebase) verzichtet werden kann. Stattdessen wird direkt der Object Detector mit Schritt 2 konfiguriert. Implementiert wird dabei die «Live Detection and Tracking», um später in Echtzeit die Objekterkennung anzuzeigen. Mit Schritt 3 wird ARCore mitgeteilt, mit welchem Bild-Stream die Object Detection laufen soll. *«The object detector runs directly from a Bitmap, NV21 ByteBuffer or a YUV_420_888 media.Image. Constructing an InputImage from those sources is recommended if you have direct access to one of them.».* Das Abrufen der YUV_420_888 ist standardmässig möglich:

```
val image = InputImage.fromBitmap(bitmap, 0)
```

Hier kommt jedoch der Haken: ARCore mit SceneView und Jetpack Compose zu kombinieren macht diesen Schritt komplizierter als erwartet. Grund dafür ist, dass SceneView nicht auf ein Image zugreifen kann, wie es nativ mit ARCore möglich ist. Gemini und Infos aus dem ARCore GitHub[28] bieten hierzu Abhilfe. Die Image.toBitmap() Funktion wurde auch im Code der Demo-App entsprechend mit Kommentaren ergänzt.

```
private fun Image.toBitmap(): Bitmap {}
```

Anstelle einer Bitmap wird der Bild-Stream ausgewählt (welcher ebenfalls im YUV-Format vorliegt) und in eine Bitmap umgewandelt. Danach wird diese Bitmap wiederum ins NV21 Format umgewandelt. NV21 ist das korrekte Format für MLKit (siehe Fussnote 27). Mit der vorhandenen Bitmap und dem instanziierten Object Detector kann mit Schritt 4 der

[27] https://developers.google.com/ml-kit/vision/object-detection/custom-models/android
[28] https://github.com/google-ar/arcore-android-sdk/issues/510

Dokumentation fortgefahren werden. Die erkannten Objekte werden in einer Map gespeichert. Jedes erkannte Objekt erhält dabei eine Tracking ID, und die erkannten Koordinaten des Objektes werden in ein *Rect()* umgewandelt, um später mit *drawRect()* gezeichnet werden zu können. Dies kann dem *ObjectDetection.kt* File in der CoreShow App entnommen werden.

Ergebnisse und Ausbaumöglichkeiten

Abbildung 6: Real Life Bildschirmaufnahme aus der Object Detection Szene von CoreShow

Mit dem aktuellen Stand der Object Detection innerhalb der Demo-App ist es möglich, ein «Haupt»-Objekt in den Fokus zu nehmen. Die App zeichnet dann eine Bounding Box um das Objekt. Bei jedem weiteren erkannten Objekt, verschwindet diese Box und wird dem nächsten zugeteilt. Eine Möglichkeit ist es, anhand der Koordinaten, diese mit den «Anchors» in der Szene zu verankern und als ChildNodes somit alle Bounding Boxes stehen zu lassen. Aufgrund der aufgetretenen Schwierigkeiten bei dieser Implementierung, kann die Demo App jedoch lediglich ein Objekt zur gleichen Zeit erkennen und mit einer Bounding Box aufzeichnen. Weiter wäre die Anbindung an Firebase und / oder Googles (Vision) API möglich, um auch die Objekte beschriften und bezeichnen zu können, wie im Beispiel von Abbildung 5.

Fazit ARCore und SceneView

Anhand der implementierten Beispiele aus der Demo App konnte ich feststellen, dass SceneView die Entwicklung von AR-Apps erleichtern kann, jedoch auch Nachteile mit sich bringt. Zum einen fehlen Dokumentationen, sodass sich Entwickler auf Beispiel-Apps stützen müssen, welche wenig Kommentare oder zusätzliche Erklärungen enthalten. Zum anderen basiert SceneView auf ARCore. Demnach kann es ARCore zwar erweitern oder vereinfachen, aber viele Funktionen oder Methoden müssen dennoch aus der ARCore SDK in SceneView implementiert werden. Dies ist teilweise unklar dokumentiert. Beispiel: Frame.acquireDepthImage, aus dem Beispiel der Object Detection und Okklusion. Diese Funktion ist im Mix zwischen Jetpack Compose und SceneView nicht mehr aufrufbar. Und deshalb auch der letzte Punkt: wer Jetpack Compose einbinden möchte, erschwert sich in einigen Fällen definitiv das Leben. Dennoch ist mit *AndroidView()* ein Workaround möglich. Abschliessend jedoch empfand ich, dass SceneView und Jetpack definitiv für kleinere App eine geeignete Alternative zum nativen ARCore ist. Für komplexere Apps sollte auf eine Einführung für OpenGL und ARCore gesetzt werden, um das volle Potential von AR auf Android auszuschöpfen. Denn Dinge wie die Okklusion sind es, die AR faszinierend machen. Leider ist somit (noch?) nicht alles mit SceneView umsetzbar.

Rückblick Seminararbeit

Ich fand es sehr spannend, die Technologie, welche Augmented Reality auf Android ermöglicht, zu erkunden. Besonders deshalb, da ich eigene Ideen und kleinere Konzepte in der Demo-App umsetzen konnte. Dennoch war der Einstieg komplexer, als zunächst erwartet. Besonders die bereitgestellten Sample Apps mit OpenGL sorgten zu Beginn für viel Verwirrung. Auch wird mit näherem Befassen klar, dass das Absetzen von SceneForm eine Lücke hinterlassen hat, die nur teilweise durch SceneView geschlossen wird. SceneView hat es mir erlaubt, die ARCore Demo ohne das Implementieren von OpenGL zu entwickeln. Das Thema Objekterkennung von MLKit ist ebenfalls eine Herausforderung für sich gewesen. Deshalb war auch die Implementierung der Object Detection Funktion nicht leicht, besonders in Kombination mit SceneView und Jetpack Compose.

Betrachtet über das gesamte Projekt hinweg, empfand ich die Implementierung in der Praxis komplexer, als es zunächst durch Dokumentationen, Beispiel-Apps oder Googles Homepage vermittelt wurde. Auch die meisten Tutorials zu ARCore auf YouTube sind oft veraltet und somit nur teilweise hilfreich, insbesondere in Kombination mit Jetpack Compose. So ist es umso erfreulicher, dass nun dennoch eine funktionierende Demo-App bereit liegt.

Hinweis: AR-Apps können leider nicht im Emulator getestet werden. Es ist also zwingend ein Android Phone notwendig, um die Demo-App zu testen. Dabei muss das Gerät mindestens Android 8.1[29] unterstützen. Mehrere Demo-Videos und Bilder sind deshalb auf GitHub im Ordner «Demo-Media» hinterlegt. Diese basieren auf der aktuellen Version der App (Stand 28. November 2024).

[29] https://developers.google.com/ar/devices