UNIVERSITY OF CALIFORNIA
SANTA CRUZ

GeoAR
Android Application with POI Helper

A project submitted in partial satisfaction

of the requirements for the degree

of

Master of Science

in

Computer Science

by

Aishna Agrawal

December 2017

The project of Aishna Agrawal
is approved:

Professor Alex Pang, Chair

Professor Narges Norouzi

Contents

Abstract

Augmented Reality(AR) is an enhanced version of reality where live direct or indirect views of physical real-world environments are augmented with superimposed computer-generated images over a user's view of the real-world, thus enhancing one's current perception of reality. There are four types of AR: Marker based, Markerless, Projection based and Superimposition based.

In this project, we experiment with Markerless AR which relies on location data instead of physical markers. Points of interest are saved in the back-end as latitude and longitude information that are queried by the front-end Android application. The app integrates Google's Augmented Reality SDK called ARCore[1] to help render objects at various points of interest in the real world. This report describes the development of an open-source app that serves as a base for future work on markerless AR. To demonstrate the usage of the app, a UCSC tour was implemented.

1 Introduction

1.1 Motivation

Both video games and cell phones are driving the development of augmented reality. Everyone from tourists, to soldiers, to someone looking for the closest subway stop can now benefit from the ability to place computer-generated graphics in their field of vision. Both Apple and Google, have recently released augmented reality platforms called ARKit and ARCore respectively. Since it's a relatively new technology and has immense potential, the idea was to experiment with one of these and create a framework that can be made open-source to make it easier to develop AR apps. As the structure of any location based AR app has to be similar this project aims to be the go to plug and play AR app generator.

1.2 Scope

The application serves as a base for any developer to build their own Geolocation AR app. The basic concept of the app is to help a user identify a certain Point of Interest(POI). When the user points their camera at a building, a 3D object like a wooden sign representing the direction and name of the building, is augmented onto their camera view. The user can view this object from any angle, interact with it by touching it for further information about the point of interest. The sample app developed as proof of concept is a UCSC tour app that helps new UCSC students identify cafes, libraries, parking spots and academic building all around campus.

1.3 Report structure

This report aims to help the developer understand each building block thoroughly with description of each component as well as code documentation. In the sections below, we will discuss background and related work, approach to the front-end app development followed by results, conclusion and expected future work.

2 | Background

Though it's been in development since the early 2000s, Augmented Reality did not get popular until last year's release of Pokemon Go. Until recently, real AR applications required the use of a tracking image such as a QR code or image marker to function. The marker would be read and interpreted by camera-equipped hardware capable of running specialized software. The smartphone, translates that information into a 3D model or animation that maintains a consistent position within the scene, regardless of how or where the user moves.Though it works well, marker-based AR can be limiting and inconvenient. Markerless Augmented Reality is now ready for mainstream adoption, thanks to Apple's ARKit and Google's ARCore[1] that were released this year.

There are a lot of AR SDKs like Wikitude, Vuforia that integrate location to create Geo-location AR apps but all of these AR engines are licensed and do not integrate well with native app development. Also, these proposals neither provide insights into the functionality of such an engine nor its customization to a specific purpose. As ARcore is available for all users to view and use freely, projects developed with it can be made open-source. Since it does not support location-based AR in its preview version, this project is an experiment to integrate it with Android's Location and Sensor data to create a location supporting version of Google ARCore.

2.1 Related work

In (Geiger 2014)[8], the authors have discussed how to develop the core of a location-based augmented reality engine for Android and iOS mobile phones. The issue with this report is that it is only designed to render 2d points at POI and not 3d objects that creates augmented reality in its true essence. Also, this framework was created before Google and Apple released AR engines compatible with their native development.

There are a few location-based AR apps that are already in the market that were built using existing AR SDKs or their own AR engines. For example, Yelp Monocle[9], the monocle is a hidden feature on the Yelp app which opens up some cool features for users who discover it. If you point your camera, you'll see boxes pop up for businesses or services nearby that you are pointing towards. You can sort to just restaurants, just bars, places your friends have been, or everything.

3 | Android application

The application relies on GPS and built-in sensors to determine location and orientation of the device. When the user opens the app, camera view is displayed. Using the GPS, current location of the user is determined and if the user is within the geographic region that the app targets, an API call is made to the server to fetch markers that will be appropriately displayed. When the user points their camera at a certain location that has a corresponding virtual marker based on calculations using current location and sensor data, a 3D object is rendered on the camera view thereby augmenting the user's reality by meshing the physical and virtual world. The location is represented by a latitude and longitude, for larger areas such as trails or paths, a set of virtual markers can be organized. On tapping the object, information about the location will be displayed.

3.1 Software and hardware requirements

The code is written entirely in Java for Android, available on GitHub: `https://github.com/aishnacodes/Geolocation-ARCore`. Requirements to set up the project are as following:

- IDE - Android Studio 3.0
- Mobile phone - Currently works on Samsung S8, Google Pixel, Pixel XL, Pixel 2, Pixel 2 XL (as of Dec 2017)

3.2 Tools and components explained

The app involves AR, sensors, GPS, Computer Graphics and server communication. Each component has extensive libraries provided by Android, the following section explains which libraries were used and how they are integrated within the app.

3.2.1 Google ARcore

ARCore is a platform for building augmented reality apps on Android. It uses Motion Tracking and Light Estimation to integrate virtual content with the real world as seen through your phone's camera. Motion tracking allows the phone to understand and track its position relative to the world. As your phone moves through the world, ARCore uses a process called concurrent odometry and mapping(COM), to understand where the phone is relative to the world around it. ARCore detects visually distinct features

in the captured camera image called feature points and uses these points to compute its change in location. The visual information is combined with inertial measurements from the device's Inertial Measurement Unit(IMU) to estimate the pose (position and orientation) of the camera relative to the world over time. Light estimation allows the phone to estimate the environment's current lighting conditions.

(a) **Config** (public final class Config)
Holds settings that are used to configure the session.

- public static Config createDefaultConfig()
  Returns a sensible default configuration. Plane detection and lighting estimation are enabled, and blocking update is selected. This configuration is guaranteed to be supported on all devices that support ARCore.

```
mDefaultConfig = Config.createDefaultConfig();
```

(b) **Frame** (public final class Frame)
Provides a snapshot of AR state at a given timestamp.

(i) public Frame.TrackingState getTrackingState()
Returns the state of the AR tracking system.

```
if (frame.getTrackingState() == TrackingState.NOT_TRACKING) {
    return;
    }
```

(ii) public Pose getPose()
Returns the pose of the user's device in the world coordinate frame at the time of capture of the current camera texture. The position of the pose is located at the device's camera. The orientation of the pose matches the orientation of the display (considering display rotation) and uses OpenGL conventions (+X right, +Y up, -Z in the direction the camera looks).

```
Pose pose = frame.getPose();
```

(iii) public LightEstimate getLightEstimate()
Returns the current ambient light estimate. Returns the pixel intensity of the current camera view. Values are on the range (0.0, 1.0), with zero being black and one being white.

```
frame.getLightEstimate().getPixelIntensity();
```

(c) **Pose** (public final class Pose)
Represents an immutable rigid transformation from one coordinate frame to another. As provided from all ARCore APIs, Poses always describe the transformation from object's local coordinate frame to the world coordinate frame. That is, Poses from ARCore APIs can be thought of as equivalent to OpenGL model matrices. The transformation is defined using a quaternion rotation about the origin followed by a translation.

public Pose (float[] translation, float[] rotation)

- **Translation**
  Translation is the position vector from the destination (usually world) coordinate frame to the local coordinate frame, expressed in destination (world) coordinates.
- **Rotation**
  Rotation is a quaternion following the Hamilton convention. Assume the destination and local coordinate frames are initially aligned, and the local coordinate frame is then rotated counter-clockwise about a unit-length axis, k, by an angle, theta. The quaternion parameters are hence:

$$x = k.x * sin(\theta/2)$$

$$y = k.y * sin(\theta/2)$$

$$z = k.z * sin(\theta/2)$$

$$w = cos(\theta/2)$$

(d) **Anchor** (public final class Anchor)
Describes a fixed location and orientation in the real world. To stay at a fixed location in physical space, the numerical description of this position will update as ARCore's understanding of the space improves. Use getPose() to get the current numerical location of this anchor. This location may change any time update() is called, but will never spontaneously change.

- Pose getPose()
  Returns the pose of the anchor in the world coordinate frame. This pose may change each time update() is called. This pose should only be considered valid if getTrackingState() returns TRACKING.

```
Pose pose = anchor.getPose();
```

(e) **Session** (public final class Session)
Manages AR system state and handles the session lifecycle. This class is the main entry point to ARCore API. This class allows the user to create a session, configure it, start/stop it, and most importantly receive frames that allow access to camera image and device pose.

(i) public void getProjectionMatrix (float[] dest, int offset, float near, float far)
Returns a projection matrix for rendering virtual content on top of the camera image.

```
mSession.getProjectionMatrix(projmtx, 0, 0.1f, 100.0f);
```

(ii) Frame update()
Updates the state of the ARCore system. This includes: receiving a new camera frame, updating the location of the device, updating the location of tracking anchors

```
Frame frame = mSession.update();
```

(iii) Anchor addAnchor (Pose pose)
Defines a tracked location in the physical world.

```
Anchor anchor = mSession.addAnchor(frame.getPose());
```

3.2.2 Geolocation

Geolocation requires an internet connection and GPS-enabled mobile phone. The application accesses the network provider's internet and GPS to identify the user's current location. Location managers and listeners are used to constantly update the user's position and re-calculate its relative position with markers.

(a) **LocationManager** (android.location.LocationManager)
This class provides access to the system location services. These services allow applications to obtain periodic updates of the device's geographical location

```
this.mLocationManager = (LocationManager)
     this.getSystemService(this.LOCATION_SERVICE);
```

(i) getLastKnownLocation()
Returns a Location indicating the data from the last known location fix obtained from the given provider.

```
mLocationManager.getLastKnownLocation
    (LocationManager.NETWORK_PROVIDER);
```

(ii) requestLocationUpdates()
Register for location updates using the named provider, and a pending intent.

```
mLocationManager.requestLocationUpdates(
    LocationManager.NETWORK_PROVIDER,
    MIN_TIME_BW_UPDATES,
    MIN_DISTANCE_CHANGE_FOR_UPDATES, this
    );
```

(b) **LocationListener** (android.location.LocationListener)
Interface LocationListener is extended by the ARActivity and is used for receiving notifications from the LocationManager when the location has changed. These methods are called if the LocationListener has been registered with the location manager service using the requestLocationUpdates.

- onLocationChanged()
  Called when the location has changed. For each marker, we calculate the distance between the current location and the marker and update it in the 'distance' variable of a MarkerInfo Object using pre-defined function distanceTo() of the Location class.

```
public void onLocationChanged(Location location) {
  mLocation = location;
  MarkerInfo marker;

  for (int i = 0; i < mMarkerList.size(); i++) {
      marker = mMarkerList.get(i);
      marker.setDistance(location.
              distanceTo(marker.getLocation()));
      }
  }
```

3.2.3 Sensors

The application makes use of the mobile phone's built-in sensors that measure motion and orientation. These sensors are capable of providing raw data with high precision and accuracy, and are useful for monitoring three-dimensional device movement or positioning. To be able to identify if the camera is pointing towards a marker location, the direction of the phone and orientation is required. This data is constantly gathered from the motion sensors.

(a) **Sensor Event** (android.hardware.SensorEvent)
This class represents a Sensor event and holds information such as the sensor's type, the time-stamp, accuracy and of course the sensor's data.

```
if (sensorEvent.sensor.getType() == Sensor.TYPE_ROTATION_VECTOR) {
    //calculation
    }
```

The rotation vector represents the orientation of the device as a combination of an angle and an axis, in which the device has rotated through an angle θ around an axis (x, y, z). The reference coordinate system is defined as a direct orthonormal basis, where:

- X is defined as the vector product Y.Z (It is tangential to the ground at the device's current location and roughly points East).
- Y is tangential to the ground at the device's current location and points towards magnetic north.
- Z points towards the sky and is perpendicular to the ground.

(b) **SensorManager** (android.hardware.SensorManager)
SensorManager lets you access the device's sensors.

```
mSensorManager = (SensorManager)
    this.getSystemService(SENSOR_SERVICE);
```

(i) getRotationMatrixFromVector (float[] R, float[] rotationVector)
Helper function to convert a rotation vector to a rotation matrix.

```
SensorManager.getRotationMatrixFromVector
    (rotationMatrixFromVector, sensorEvent.values);
```

(ii) remapCoordinateSystem (float[] inR, int X, int Y, float[] outR)
Rotates the supplied rotation matrix so it is expressed in a different coordinate system.
The phone is assumed to be horizontal as if it were lying on a table with its screen facing upward and the top of the phone pointing away from the user. Since we are using the camera for our augmented reality application, the phone needs to be upright, for that we need to change our coordinate system.

```
SensorManager
    .remapCoordinateSystem(rotationMatrixFromVector,
        SensorManager.AXIS_X, SensorManager.AXIS_Y,
                            updatedRotationMatrix);
```

(iii) getOrientation (float[] R, float[] values)
Computes the device's orientation based on the rotation matrix.

```
SensorManager.getOrientation
    (updatedRotationMatrix, orientationValues);
```

When it returns, the array values are as follows:

- **Azimuth**(values[0]): angle of rotation about the -z axis.
  This value represents the angle between the device's y axis and the magnetic north pole. When facing north, this angle is 0, when facing south, this angle is π.
- **Pitch**(values[1]): angle of rotation about the x axis.
  This value represents the angle between a plane parallel to the device's screen and a plane parallel to the ground. Assuming that the bottom edge of the device faces the user and that the screen is face-up, tilting the top edge of the device toward the ground creates a positive pitch angle. The range of values is -π to π.
- **Roll**(values[2]): angle of rotation about the y axis.
  This value represents the angle between a plane perpendicular to the device's screen and a plane perpendicular to the ground. Assuming that the bottom edge of the device faces the user and that the screen is face-up, tilting the left edge of the device toward the ground creates a positive roll angle. The range of values is -$\pi/2$ to $\pi/2$.

(c) **SensorEventListener** (android.hardware.SensorEventListener)
SensorManager lets you access the device's sensors. onSensorChanged(SensorEvent event) is called every time the rotation vector event is changed. Here we check if a marker is in range of the camera view. We do this by calculating the bearing from user's current location to the marker's location using latitude and longitude for locations. The bearing is the angle that is between 2 locations with respect to magnetic north. If the Azimuth (phone's direction with respect to north) is equal to the bearing then the marker is in range. To account for the width of the marker's

corresponding building or location, we keep a buffer in the field of view for the marker. This buffer depends on the distance of the user from the marker. To make sure that the phone is mostly upright, pitch is also constrained to an angle of 45 to 90 degrees.

```
Range<Float> azimuthRange, pitchRange;
azimuthRange = new Range<>(bearing - buffer, bearing + buffer);
pitchRange = new Range<>(-90.0f, -45.0f);

if (azimuthRange.contains(azimuth) && pitchRange.contains(pitch)) {
    markerInRange = true;
}
```

3.2.4 OpenGL

Android includes support for high performance 2D and 3D graphics with the Open Graphics Library (OpenGL), specifically, the OpenGL ES 2.0 API. OpenGL is a cross-platform graphics API that specifies a standard software interface for 3D graphics processing hardware.

(a) **GLSurfaceView**
This class is a View where you can draw and manipulate objects using OpenGL API calls. You can use this class by creating an instance of GLSurfaceView and adding your Renderer to it.

```
mSurfaceView = (GLSurfaceView) findViewById(R.id.surfaceview);
mSurfaceView.setPreserveEGLContextOnPause(true);
mSurfaceView.setEGLContextClientVersion(2);
mSurfaceView.setEGLConfigChooser(8, 8, 8, 8, 16, 0);
mSurfaceView.setRenderer(this);
mSurfaceView.setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
```

(b) **GLSurfaceView.Renderer**
This interface defines the methods required for drawing graphics in a GLSurfaceView. An implementation of this interface is provided as a separate class and attached to GLSurfaceView instance using GLSurfaceView.setRenderer(). The GLSurfaceView.Renderer interface requires that you implement the following methods:

(i) onSurfaceCreated(): The system calls this method once, when creating the GLSurfaceView. We use this method to perform actions that need to happen only once, such as creating the texture and passing it to ARCore session to be filled during update() and preparing the rendering objects.

```
mBackgroundRenderer.createOnGlThread(/*context=*/this);
mSession.setCameraTextureName(mBackgroundRenderer.getTextureId());

mVirtualObject.createOnGlThread(/*context=*/this,
        "object.obj", "object_texture.png");
mVirtualObject.setMaterialProperties(0.0f, 3.5f, 1.0f, 6.0f);
```

(ii) onDrawFrame(): The system calls this method on each redraw of the GLSurfaceView. We see this method as the primary execution point for drawing (and re-drawing) graphic objects.

```
public void onDrawFrame(GL10 gl) {
    // code
    mVirtualObject.updateModelMatrix(mAnchorMatrix, scaleFactor);
    mVirtualObject.draw(viewmtx, projmtx, lightIntensity);
}
```

(iii) onSurfaceChanged(): The system calls this method when the GLSurfaceView geometry changes, including changes in size of the GLSurfaceView or orientation of the device screen. For example, the system calls this method when the device changes from portrait to landscape orientation.

```
public void onSurfaceChanged(GL10 gl, int width, int height) {
    GLES20.glViewport(0, 0, width, height);
    mSession.setDisplayGeometry(width, height);
}
```

3.2.5 Back-end linking

The app retrieves the virtual markers from the server based on the user's current location. To receive these markers a call is made to the POI helper server where the virtual markers are saved in JavaScript object notation(JSON) format.

(a) **Retrofit**
Retrofit is a type-safe HTTP client for Android and Java. Retrofit turns the HTTP API into a Java interface. It is used to make API calls to the POI helper server. HTTP's response body contains any information and Retrofit parses the data and maps it into a defined Java class.

(i) GSON
The converter dependency provides a GsonConverterFactory class. Retrofit automatically takes care of mapping the JSON data to Java objects (MarkerInfo).

```
mRetrofit = new Retrofit.Builder()
                .baseUrl(mBaseUrl)
                .addConverterFactory(GsonConverterFactory.create())
                .build();

        mMarkerApi = mRetrofit.create(MarkerApi.class);
```

(ii) Request
The URL contains Start latitude, end latitude, start longitude, end longitude to define the geographic region that the markers will be confined to.

```
public interface MarkerApi {
    @GET("36.97398389105355/37.00942677981021/-122.08119844562987/-122.047381
```

```
        Call<List<MarkerInfo>> getMarkers();
    }
```

(iii) Call

```
    Call<List<MarkerInfo>> call = mMarkerApi.getMarkers();

    call.enqueue(new Callback<List<MarkerInfo>>() {

    @Override
    public void onResponse(Call<List<MarkerInfo>> call,
            Response<List<MarkerInfo>> response) {
                mMarkerList.addAll(response.body());
        }

    // on failure code
    });
```

(b) **POJO**

POJO means Plain Old Java Object. It refers to a Java object (instance of definition) that isn't bogged down by framework extensions.

```
public class MarkerInfo {

    @SerializedName("_id")
    @Expose
    private String id;
    @SerializedName("time")
    @Expose
    private LocationTime time;
    @SerializedName("name")
    @Expose
    private String name;
    @SerializedName("category")
    @Expose
    private String category;
    @SerializedName("location")
    @Expose
    private MarkerLocation markerLocation;

    // setter and getters
}
```

4 | Results

The app fulfills all of the expectations and goals that were set in the initial stages of development. Since the app depends on GPS and sensors, its behavior is variable and not always consistent with the expected results. As Google ARcore improves its understanding of the world, the rendered object's pose will be more stable. A short video of the working can be viewed on YouTube[6] and the code is available on GitHub[7].

Figure 4.1: Jack Baskin Engineering 1

Figure 4.2: Jack Baskin Engineering 2

Figure 4.3: Science and Engineering Library

Figure 4.4: Perk's Coffee

5 | Conclusion and future work

The app is able to fetch virtual markers from the back-end server, render 3d objects in the user's camera view based on marker locations and display additional information about the point of interest. Creating this AR app has been interesting and challenging. The lack of existing work with the brand new library, drove the need to try multiple approaches and settling on the best. I look forward to see other developers using this platform to create unique and fun AR apps.

The app and the corresponding back-end that was created by Aakash Thakkar serve as a good foundation to create a Geolocation based AR app. Since the app was built using a preview version of Google ARCore, once the version 1.0 is released, the project would be updated. The GitHub project would be regularly maintained.

Bibliography

[1] Google ARCore: Getting started with Android Studio
https://developers.google.com/ar/develop/java/getting-started

[2] Google ARCore API Reference
https://developers.google.com/ar/reference/

[3] Android API Guides
https://developer.android.com/guide/index.html

[4] Outware insights: Which direction am I facing?
http://www.outware.com.au/insights/which-direction-am-i-facing
-using-the-sensors-on-your-android-phone-to-record-where-you-are-facing/

[5] Latitude and Longitude finder
https://www.latlong.net/

[6] YouTube: Android Geolocation Augmented Reality using Google ARCore
https://www.youtube.com/watch?v=RAg6u2AZ1fI

[7] GitHub: Geolocation-ARCore repository
https://github.com/aishnacodes/Geolocation-ARCore

[8] Location-based Mobile Augmented Reality Applications Challenges, Examples, Lessons Learned (2014)
Philip Geiger, Marc Schickler, Rudiger Pryss, Johannes Schobel, Manfred Reichert
Institute of Databases and Information Systems, University of Ulm, James-Franck-Ring, Ulm, Germany

[9] Yelp Monocle: Support page
https://www.yelp-support.com/article/What-is-Yelp-s-Monocle-feature