**Geospatial data collection**

Contents

[1. introduction 1](#_Toc1689578953)

[1. The importance of geospatial data 1](#_Toc577198790)

[2. The demand for data acquisition is growing 2](#_Toc796430031)

[2. Detailed introduction of the App 3](#_Toc333959266)

[1. Main functions of the app 3](#_Toc1478614051)

[2. App architecture/technical introduction 7](#_Toc385592073)

[3. Data acquisition method 8](#_Toc1687807912)

[4. Data management 20](#_Toc876083761)

[5. Technical highlights and advantages 22](#_Toc1191887295)

## introduction

### The importance of geospatial data

Geospatial data is of great importance in many aspects, mainly reflected in the following points:

Providing a comprehensive perspective: Geospatial data can associate various information with geographical locations, presenting the spatial distribution and interrelationships of things to decision-makers, thereby formulating strategies that are more in line with actual situations.

Support scientific decision-making: Through the analysis and mining of geospatial data, the laws and trends hidden behind the data can be revealed, helping decision-makers make scientific predictions and evaluations and improving the accuracy and reliability of decisions.

Optimize resource allocation: Enterprises and governments can use geospatial data to understand the distribution and utilization efficiency of resources, so as to achieve a reasonable allocation of resources, improve production efficiency and reduce costs.

Help with business location selection: Retail enterprises can understand factors such as population density, consumption capacity, and transportation convenience by analyzing geospatial data, so as to select the best store location. Logistics enterprises can optimize distribution routes based on geospatial data, improve distribution efficiency and reduce logistics costs.

technologies such as satellite remote sensing and geographic information systems, it is possible to conduct long-term monitoring of environmental elements such as land cover, vegetation changes, water pollution, and air pollution. This allows for timely grasping of environmental change trends and provides a scientific basis for the formulation and implementation of environmental protection policies.

Reasonable utilization of resources: Geospatial data helps to comprehensively understand the distribution and reserves of natural resources, thereby realizing the rational development and protection of forests, minerals, water resources, etc., and promoting the sustainable utilization of resources.  
Effectively respond to natural disasters and emergencies.

### The demand for data acquisition is growing

The demand for data acquisition is growing for several reasons:

Business Intelligence and Decision Making

In today's competitive business environment, companies need accurate and up-to-date data to make informed decisions. Data acquisition provides the raw material for analytics and helps organizations understand customer behavior, market trends, and operational efficiencies.

By collecting data on various aspects such as sales, customer demographics, and website traffic, businesses can identify opportunities for growth, optimize marketing strategies, and improve product offerings.

Internet of Things (IoT)

The proliferation of IoT devices has led to a massive increase in the amount of data available. Sensors embedded in everything from smartphones and wearables to industrial machinery and smart home devices generate continuous streams of data.

This data can be used for applications such as predictive maintenance, energy management, and personalized experiences. For example, manufacturers can use data from sensors on production equipment to detect potential failures before they occur and schedule maintenance proactively.

Big Data and Advanced Analytics

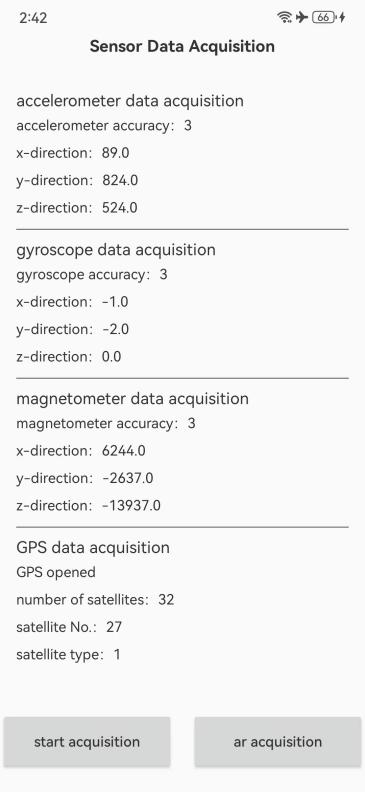
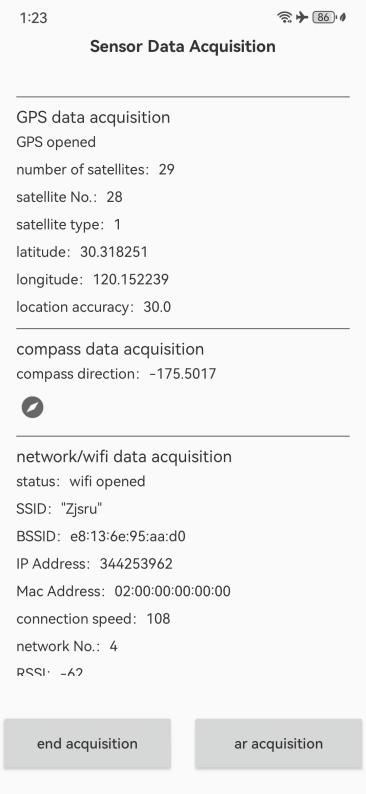
The advent of big data technologies has made it possible to process and analyze large volumes of data quickly and efficiently. This has led to an increased demand for data acquisition to feed these analytics engines.

Advanced analytics techniques such as machine learning and artificial intelligence require large amounts of data to train and improve their models. The more data that is available, the more accurate and useful the insights generated can be.

## Detailed introduction of the App

### Main functions of the app

**Collect spatial data information through mobile phone hardware sensors.**

Accelerometer sensor collects data.

Accelerometer data is collected in real time through the mobile phone accelerometer sensor, including accelerometer accuracy, accelerometer x-direction data, accelerometer y-direction data, and accelerometer z-direction data.

Gyroscope sensor data acquisition.

Collect gyroscope data in real time through the mobile phone gyroscope sensor, including gyroscope accuracy, gyroscope x-direction data, gyroscope y-direction data, and gyroscope z-direction data.

Magnetometer sensor data acquisition.

Collect magnetometer data through the mobile phone magnetometer sensor, including magnetometer accuracy, magnetometer x-direction data, magnetometer y-direction data, and magnetometer z-direction data.

GPS data acquisition.

GPS data acquisition. Obtain latitude and longitude and satellite-related data through Android's native GNSS, including whether GPS is turned on, the number of satellites, satellite numbers, satellite types, latitude and longitude, and positioning accuracy.

Compass data acquisition.

Collect the compass direction in real time through the mobile phone compass sensor, and statistically calculate the current pointing direction of the mobile phone, and perceive the pointing direction of the mobile phone in real time through the compass icon.

Network/Wi-Fi data acquisition.

Obtain detailed information of the current network connection through Android's native network management class, including: whether Wi-Fi is turned on, Wi-Fi SSID, Wi-Fi BSSID, IP address, MAC address, network number, Wi-Fi RSSI, whether the SSID is hidden, and Wi-Fi accuracy.

Bluetooth data acquisition.

Collect Bluetooth connection information through Android's native Bluetooth management tool (BLE), including: whether Bluetooth is turned on, Bluetooth name, MAC address, and discoverable Bluetooth devices nearby.

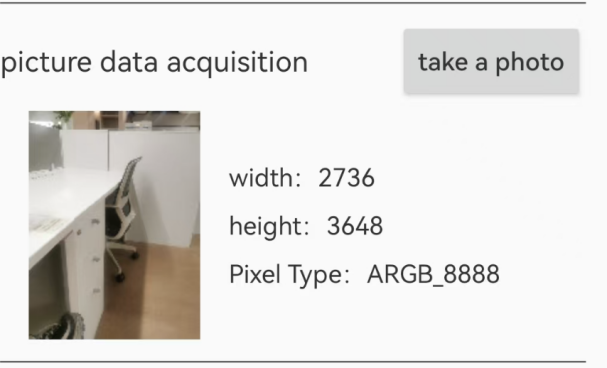
**Access the Google AR framework ARCore to realize AR acquisition of geospatial data.**



Access ARCore, render the surrounding environment through the mobile phone camera, and render a robot on a plane. The robot calculates the world coordinates of the plane through the pose tool provided by ARCore, and calculates the moving distance in real time when the mobile phone moves. After testing, the obtained moving distance has a high accuracy.

**Picture RGB data acquisition**

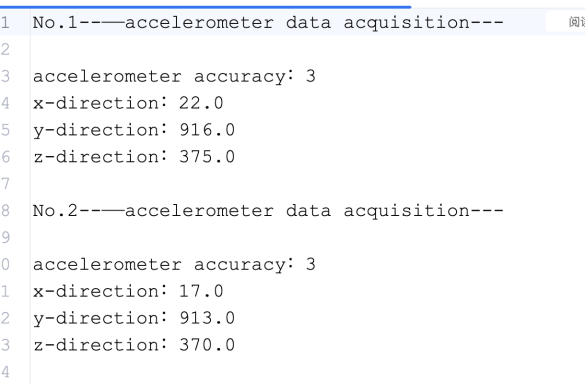
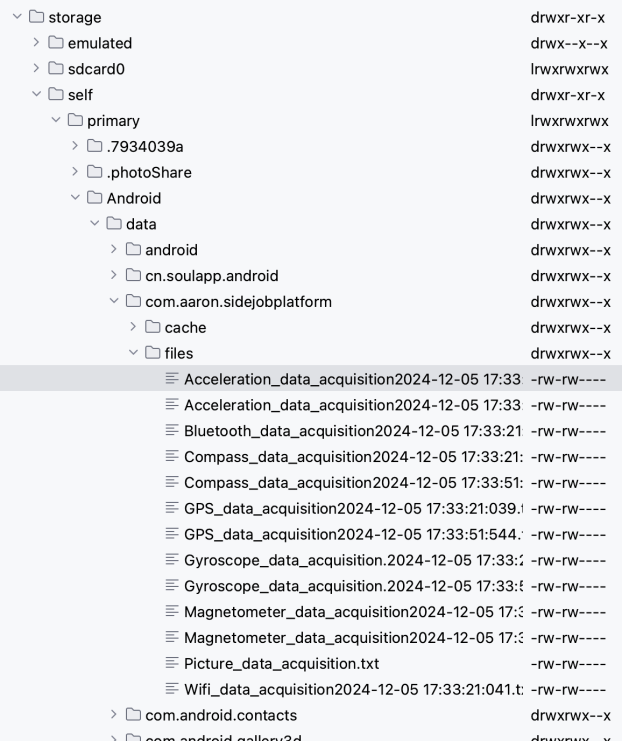
Click the "Take Photo" button to call the system camera to take a photo. After obtaining the captured photo, analyze the photo to obtain picture-related data, including picture width, picture height, and picture color coding format.

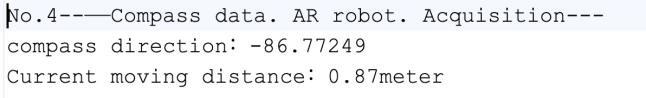
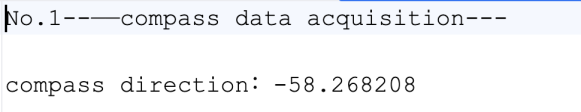
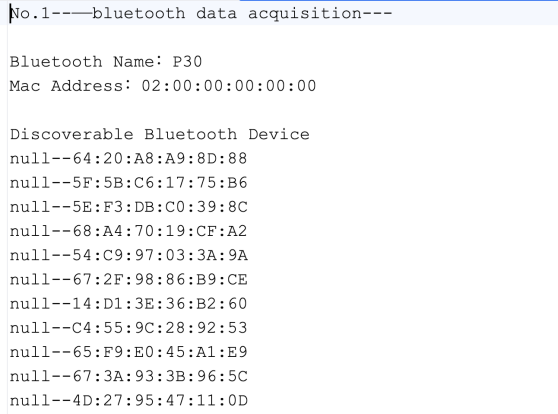
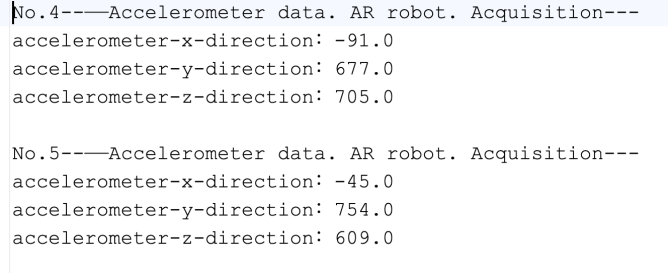


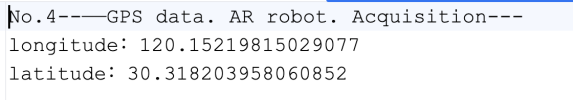
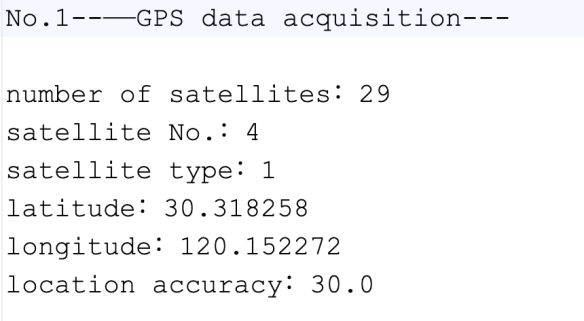
**Local storage of collected data.**

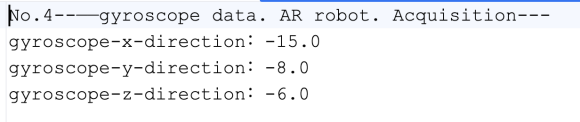
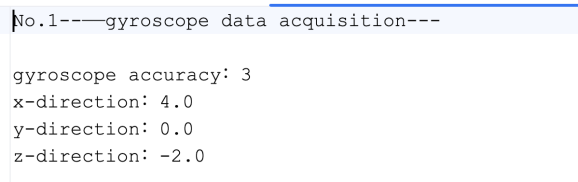
Sensor data acquisition and AR data acquisition write the collected data to the local storage of the mobile phone at a frequency of once per second.

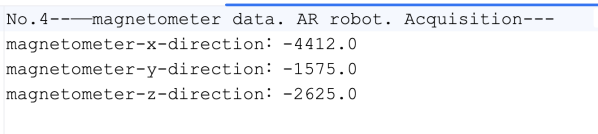
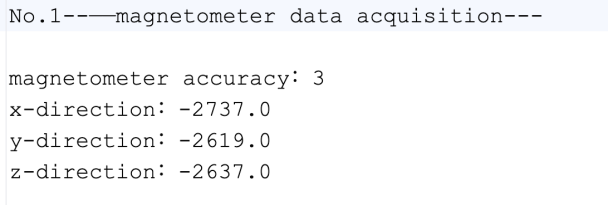
The collected data is stored in separate files according to sensor and AR acquisition, and is stored in separate files according to accelerometer, compass, magnetometer, gyroscope, GPS, Wi-Fi, Picture RGB，and Bluetooth data.

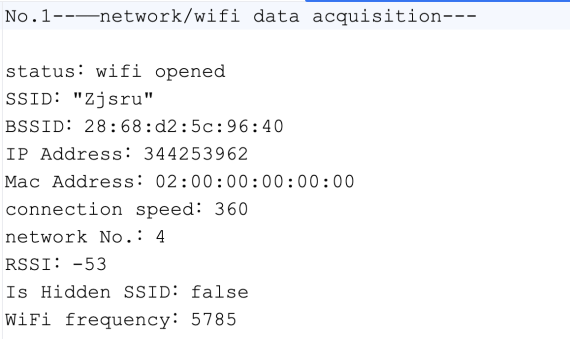
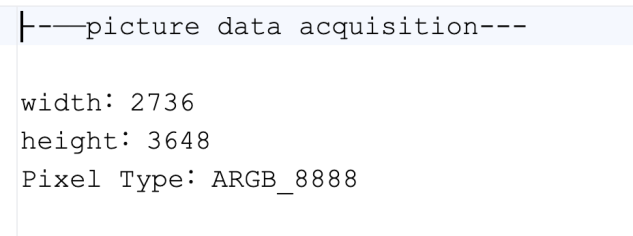












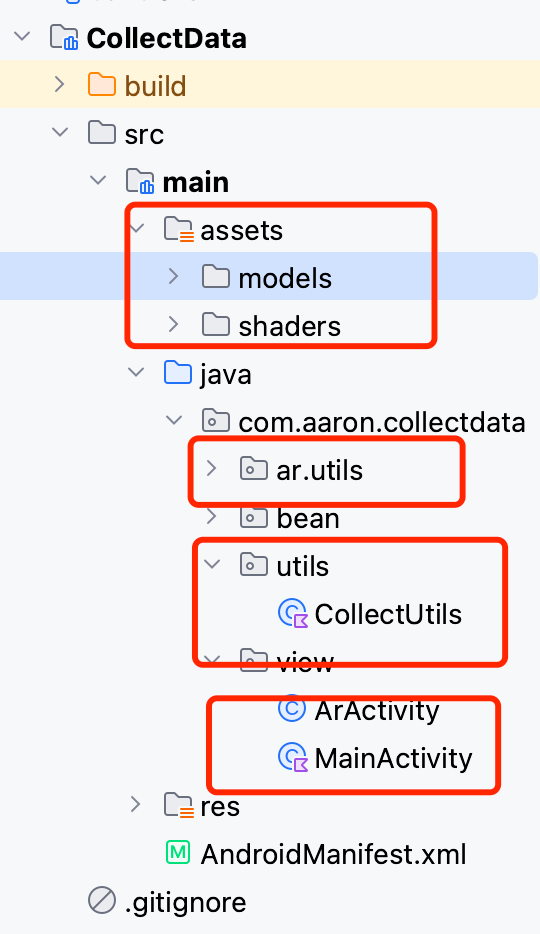
### App architecture/technical introduction

The app is developed natively for Android using Kotlin + Gradle. High-performance coroutines are added to handle asynchronous data processing.

Kotlin is a modern programming language with characteristics such as simplicity, safety, and high efficiency. Gradle provides powerful build and dependency management functions. Coroutines can effectively handle asynchronous operations, improve the responsiveness and performance of applications, and make it more efficient and concise when handling asynchronous tasks such as data loading and network requests.

Access ARCore to achieve AR functions. Use the Pose tool class provided by ARCore to achieve real-time calculation of spatial positions and motion tracking.

The project architecture is as follows.



The "assets" directory stores ARCore models.

The "ar.utils" directory stores AR rendering tools. It mainly renders AR models based on OpenGL.

The CollectUtils utility class implements sensor listening, Bluetooth processing, network listening, image decoding and acquisition, and GNSS listening.

ArActivity implements the AR acquisition interface.

MainActivity implements the rendering of the sensor data, Bluetooth, network, and GNSS acquisition interface.

### Data acquisition method

**Sensor data acquisition.**

Accelerometer sensor data acquisition implementation.

Define the sensor management object.

*private lateinit var sensorManager: SensorManager*

*//Obtain the sensor manager and sensor.  
sensorManager = context.getSystemService(Context.SENSOR\_SERVICE) as SensorManager  
//Obtain the accelerometer sensor.  
accSensor = sensorManager.getDefaultSensor(Sensor.TYPE\_ACCELEROMETER)!!  
//listen  
sensorManager.registerListener(acceListener, accSensor, SensorManager.SENSOR\_DELAY\_NORMAL)*

Implement listen.

*private lateinit var acceListener: SensorEventListener*

*acceListener = object: SensorEventListener {  
 override fun onSensorChanged(event: SensorEvent?) {  
  
 acceBean = AccelerometerBean(event.accuracy,  
 Math.round(event.values[0] \* 100).toFloat(),  
 Math.round(event.values[1] \* 100).toFloat(),  
 Math.round(event.values[2] \* 100).toFloat())  
  
 }  
  
 override fun onAccuracyChanged(sensor: Sensor?, accuracy: Int) {  
  
 }  
}*

Compass sensor data acquisition implementation.

Define the sensor management object.

*private lateinit var sensorManager: SensorManager*

*//Obtain the sensor manager and sensor.  
sensorManager = context.getSystemService(Context.SENSOR\_SERVICE) as SensorManager*

*//Obtain the compass sensor.  
val sensor1 : Sensor = sensorManager.getDefaultSensor(Sensor.TYPE\_ACCELEROMETER)!!  
val sensor2 : Sensor = sensorManager.getDefaultSensor(Sensor.TYPE\_MAGNETIC\_FIELD)!!  
//listen  
sensorManager.registerListener(compassListener, sensor1, SensorManager.SENSOR\_DELAY\_GAME)  
sensorManager.registerListener(compassListener, sensor2, SensorManager.SENSOR\_DELAY\_GAME)*

Implement listen.

*compassListener = object: SensorEventListener {  
 override fun onSensorChanged(event: SensorEvent?) {  
  
 if (event!!.sensor.type === Sensor.TYPE\_ACCELEROMETER) {  
   
 accelerometerValues = event!!.values.clone()  
 } else if (event!!.sensor.type === Sensor.TYPE\_MAGNETIC\_FIELD) {  
 magneticValues = event!!.values.clone()  
 }  
  
 val R = FloatArray(9)  
 val values = FloatArray(3)  
 SensorManager.getRotationMatrix(  
 R, null, accelerometerValues,  
 magneticValues  
 )  
 SensorManager.getOrientation(R, values)  
  
  
 norPoint = -(Math.toDegrees(values[0].toDouble()).toFloat())  
  
  
 }  
  
 override fun onAccuracyChanged(sensor: Sensor?, accuracy: Int) {  
  
 }  
}*

Gyroscope sensor data acquisition implementation.

Define the sensor management object.

*private lateinit var sensorManager: SensorManager*

*//Obtain the sensor manager and sensor.  
sensorManager = context.getSystemService(Context.SENSOR\_SERVICE) as SensorManager*

/*/Obtain the gyroscope sensor.*  
gyroscopeSensor = sensorManager.getDefaultSensor(Sensor.TYPE\_GYROSCOPE)!!  
//*listen*

sensorManager.registerListener(gyroscopeListener, gyroscopeSensor, SensorManager.SENSOR\_DELAY\_NORMAL)

Implement listen.

*gyroscopeListener = object: SensorEventListener {  
 override fun onSensorChanged(event: SensorEvent?) {  
  
 gyroscopeBean = AccelerometerBean(event.accuracy,  
 Math.round(event.values[0] \* 100).toFloat(),  
 Math.round(event.values[1] \* 100).toFloat(),  
 Math.round(event.values[2] \* 100).toFloat())  
  
 }  
  
 override fun onAccuracyChanged(sensor: Sensor?, accuracy: Int) {  
  
 }  
}*

Magnetometer sensor data acquisition implementation.

Define the sensor management object.

*private lateinit var sensorManager: SensorManager*

*//Obtain the sensor manager and sensor.  
sensorManager = context.getSystemService(Context.SENSOR\_SERVICE) as SensorManager*

/*/Obtain the magnetometer sensor.*  
*magnetometerSensor = sensorManager.getDefaultSensor(Sensor.TYPE\_MAGNETIC\_FIELD)!!*

//*listen*

*sensorManager.registerListener(magnetometerListener, magnetometerSensor, SensorManager.SENSOR\_DELAY\_NORMAL)*

Implement listen.

*magnetometerListener = object: SensorEventListener {  
 override fun onSensorChanged(event: SensorEvent?) {  
  
 magnetometerBean = AccelerometerBean(event.accuracy,  
 Math.round(event.values[0] \* 100).toFloat(),  
 Math.round(event.values[1] \* 100).toFloat(),  
 Math.round(event.values[2] \* 100).toFloat())  
  
 }  
  
 override fun onAccuracyChanged(sensor: Sensor?, accuracy: Int) {  
  
 }  
}*

GNSS data acquisition implementation.

*private lateinit var locationManager: LocationManager  
private lateinit var gnssMeasurementsEvent: GnssMeasurementsEvent.Callback  
private lateinit var locationListener: LocationListener  
private lateinit var mGnssStatusCallback: GnssStatus.Callback*

*locationManager = context.getSystemService(Context.LOCATION\_SERVICE) as LocationManager  
  
 mGnssStatusCallback = object : GnssStatus.Callback(){  
  
 override fun onSatelliteStatusChanged(status: GnssStatus) {  
 super.onSatelliteStatusChanged(status)  
  
 var satelliteCount: Int = status.getSatelliteCount();  
 var sat\_id = status.getSvid(1);  
 var constellationType = status.getConstellationType(1);  
// v = " Scan: "+ (count)+ " ,Sat-count=" +(satelliteCount) +"id="+ sat\_id +" type"+ constellationType + " ,Eacc= "+accuracy+ "\n\n";  
// v += tv1.getText();  
// tv1.setText(v);  
// count= count + 1;  
  
 gpsCollectData = GPSCollectData(gpsCollectData.status, satelliteCount, sat\_id, constellationType, gpslatitude, gpslongitude, gpsaccuracy)  
  
  
 }  
 }  
  
 locationManager.registerGnssStatusCallback(mGnssStatusCallback);  
  
 gnssMeasurementsEvent = object : Callback(){  
  
 override fun onGnssMeasurementsReceived(eventArgs: GnssMeasurementsEvent?) {  
 super.onGnssMeasurementsReceived(eventArgs)  
  
 "GNNS listen callback".showLog()  
  
 }  
  
 override fun onStatusChanged(status: Int) {  
 super.onStatusChanged(status)  
 }  
 }  
  
 locationManager.registerGnssMeasurementsCallback(gnssMeasurementsEvent);  
  
 locationListener = object : LocationListener{  
 override fun onLocationChanged(location: Location) {  
  
 if(null!=location){  
 gpslatitude = location.getLatitude();  
 gpslongitude = location.getLongitude();  
 gpsaccuracy = location.getAccuracy();  
 }  
  
 }  
  
 override fun onStatusChanged(provider: String?, status: Int, extras: Bundle?) {  
 "onStatusChanged".showLog()  
 }  
  
 override fun onProviderDisabled(provider: String) {  
 "onProviderDisabled".showLog()  
 gpsCollectData = GPSCollectData(provider+"disabled")  
 }  
  
 override fun onProviderEnabled(provider: String) {  
 "onProviderEnabled".showLog()  
 gpsCollectData = GPSCollectData(provider+"opened")  
 }  
  
 override fun onFlushComplete(requestCode: Int) {  
 "onFlushComplete".showLog()  
 }  
  
 override fun onLocationChanged(locations: MutableList<Location>) {  
 "onLocationChanged".showLog()  
 }  
  
 }  
  
// var criteria: Criteria = Criteria()  
// criteria.accuracy = Criteria.ACCURACY\_COARSE  
// criteria.isAltitudeRequired = true  
// criteria.isBearingRequired = true  
// criteria.isCostAllowed = true  
// criteria.powerRequirement = Criteria.POWER\_LOW  
// val bestProvider: String? = locationManager.getBestProvider(criteria, true)  
  
 val location: Location? = locationManager?.getLastKnownLocation((LocationManager.NETWORK\_PROVIDER))  
  
 location?.let {  
  
 gpslatitude = it.getLatitude();  
 gpslongitude = it.getLongitude();  
 gpsaccuracy = it.getAccuracy();  
 }  
  
 locationManager.requestLocationUpdates(LocationManager.GPS\_PROVIDER, 30000L, 0F, locationListener)*

Bluetooth data acquisition implementation.

*private val ble: BluetoothAdapter = BluetoothAdapter.getDefaultAdapter()  
//private lateinit var bluetoothManager: BluetoothManager  
private lateinit var scanCallback: ScanCallback*

*ble?.let {  
  
 if(!ble.isEnabled) return  
  
 scanCallback = object : ScanCallback(){  
  
 override fun onScanFailed(errorCode: Int) {  
 super.onScanFailed(errorCode)  
 }  
  
 override fun onScanResult(callbackType: Int, result: ScanResult?) {  
 super.onScanResult(callbackType, result)  
  
 if(null!=result && null != result.device){  
  
 bleCollectData.otherDev.add(result.device)  
 bleCollectData = BleCollectData(false, "", bleCollectData.name, bleCollectData.address, bleCollectData.otherDev)  
 }  
 }  
  
 override fun onBatchScanResults(results: MutableList<ScanResult>?) {  
 super.onBatchScanResults(results)  
 }  
 }  
  
 ble.bluetoothLeScanner.startScan(scanCallback)  
  
}*

WIFI data acquisition implementation.

*private lateinit var wifiManager: WifiManager*

*wifiManager = context.getSystemService(Context.WIFI\_SERVICE) as WifiManager  
  
//wifi Status  
var wifiState = wifiManager.wifiState  
//Obtain detailed information of WiFi.  
var wifiInfo = wifiManager.connectionInfo  
  
var wifiStateStr = ""  
  
if(wifiState == WifiManager.WIFI\_STATE\_ENABLED){  
 wifiStateStr = "wifi opened";  
} else if(wifiState == WifiManager.WIFI\_STATE\_DISABLED){  
 wifiStateStr = "wifi closed";  
} else if(wifiState == WifiManager.WIFI\_STATE\_ENABLING){  
 wifiStateStr = "wifi opening";  
} else if(wifiState == WifiManager.WIFI\_STATE\_DISABLING){  
 wifiStateStr = "wifi closing";  
} else{  
 wifiStateStr = "wifi unknown";  
}  
  
return WifiCollectData(wifiStateStr, wifiInfo)*

Picture data acquisition implementation.

*var photo: Bitmap? = BitmapFactory.decodeFile(photoFile.absolutePath)  
  
var rgbBuiler: StringBuilder = StringBuilder()  
  
rgbBuiler.append(loadString(R.string.label\_34))  
rgbBuiler.append("\n\r")  
  
xBing?.imgW?.text = loadString(R.string.label\_35)+photo?.width  
rgbBuiler.append(loadString(R.string.label\_35)+photo?.width)  
rgbBuiler.append("\n")  
xBing?.imgH?.text = loadString(R.string.label\_36)+photo?.height  
rgbBuiler.append(loadString(R.string.label\_36)+photo?.height)  
rgbBuiler.append("\n")  
xBing?.imgConfig?.text = loadString(R.string.label\_37)+photo?.config.toString()  
rgbBuiler.append(loadString(R.string.label\_37)+photo?.config.toString())  
rgbBuiler.append("\n\r")  
  
CollectUtils.getInstance(this@MainActivity).savedata2File(CollectUtils.getInstance(this@MainActivity).picRgbPath,  
 rgbBuiler.toString())  
  
val matrix: Matrix = Matrix();  
  
val ei: ExifInterface = ExifInterface(photoFile.absolutePath);  
var orientation = ei.getAttributeInt(ExifInterface.TAG\_ORIENTATION, ExifInterface.ORIENTATION\_NORMAL)  
  
if(orientation == ExifInterface.ORIENTATION\_ROTATE\_90){  
  
 matrix.postRotate(90F)  
} else if(orientation == ExifInterface.ORIENTATION\_ROTATE\_180){  
  
 matrix.postRotate(180F)  
} else if(orientation == ExifInterface.ORIENTATION\_ROTATE\_270){  
  
 matrix.postRotate(270F)  
} else {  
 matrix.postRotate(0F)  
}*

*var scale: Float = Math.max(128 / photo?.width!!.toFloat(), 128 / photo?.height!!.toFloat())  
  
matrix.postScale(scale, scale)  
  
//  
xBing?.showPhoto?.setImageBitmap(Bitmap.createBitmap(photo!!, 0, 0, photo.getWidth(), photo.getHeight(), matrix, false));*

**AR data acquisition.**

*//Load the AR model.*

*pointCloudShader =  
 Shader.createFromAssets(  
 render,  
 "shaders/point\_cloud.vert",  
 "shaders/point\_cloud.frag",  
 /\* defines= \*/ null)  
 .setVec4(  
 "u\_Color", new float[] {31.0f / 255.0f, 188.0f / 255.0f, 210.0f / 255.0f, 1.0f})  
 .setFloat("u\_PointSize", 5.0f);  
// four entries per vertex: X, Y, Z, confidence  
pointCloudVertexBuffer =  
 new VertexBuffer(render, /\* numberOfEntriesPerVertex= \*/ 4, /\* entries= \*/ null);  
final VertexBuffer[] pointCloudVertexBuffers = {pointCloudVertexBuffer};  
pointCloudMesh =  
 new Mesh(  
 render, Mesh.PrimitiveMode.POINTS, /\* indexBuffer= \*/ null, pointCloudVertexBuffers);  
  
// Virtual object to render (ARCore pawn)  
virtualObjectAlbedoTexture =  
 Texture.createFromAsset(  
 render,  
 "models/andy.png",  
 Texture.WrapMode.CLAMP\_TO\_EDGE,  
 Texture.ColorFormat.SRGB);  
virtualObjectAlbedoInstantPlacementTexture =  
 Texture.createFromAsset(  
 render,  
 "models/andy.png",  
 Texture.WrapMode.CLAMP\_TO\_EDGE,  
 Texture.ColorFormat.SRGB);  
Texture virtualObjectPbrTexture =  
 Texture.createFromAsset(  
 render,  
 "models/andy.png",  
 Texture.WrapMode.CLAMP\_TO\_EDGE,  
 Texture.ColorFormat.LINEAR);  
  
virtualObjectMesh = Mesh.createFromAsset(render, "models/andy.obj");  
virtualObjectShader =  
 Shader.createFromAssets(  
 render,  
 "shaders/environmental\_hdr.vert",  
 "shaders/environmental\_hdr.frag",  
 /\* defines= \*/ new HashMap<String, String>() {  
 {  
 put(  
 "NUMBER\_OF\_MIPMAP\_LEVELS",  
 Integer.toString(cubemapFilter.getNumberOfMipmapLevels()));  
 }  
 })  
 .setTexture("u\_AlbedoTexture", virtualObjectAlbedoTexture)  
 .setTexture("u\_RoughnessMetallicAmbientOcclusionTexture", virtualObjectPbrTexture)  
 .setTexture("u\_Cubemap", cubemapFilter.getFilteredCubemapTexture())  
 .setTexture("u\_DfgTexture", dfgTexture);*

*//Implement AR motion tracking.*

*camera.getProjectionMatrix(projectionMatrix, 0, Z\_NEAR, Z\_FAR);  
  
// Get camera matrix and draw.  
camera.getViewMatrix(viewMatrix, 0);  
  
// Visualize tracked points.  
// Use try-with-resources to automatically release the point cloud.  
try (PointCloud pointCloud = frame.acquirePointCloud()) {  
 if (pointCloud.getTimestamp() > lastPointCloudTimestamp) {  
 pointCloudVertexBuffer.set(pointCloud.getPoints());  
 lastPointCloudTimestamp = pointCloud.getTimestamp();  
 }  
 Matrix.multiplyMM(modelViewProjectionMatrix, 0, projectionMatrix, 0, viewMatrix, 0);  
 pointCloudShader.setMat4("u\_ModelViewProjection", modelViewProjectionMatrix);  
 render.draw(pointCloudMesh, pointCloudShader);  
}  
  
// Visualize planes.  
planeRenderer.drawPlanes(  
 render,  
 session.getAllTrackables(Plane.class),  
 camera.getDisplayOrientedPose(),  
 projectionMatrix);  
  
// -- Draw occluded virtual objects  
  
// Update lighting parameters in the shader  
updateLightEstimation(frame.getLightEstimate(), viewMatrix);  
  
// Visualize anchors created by touch.  
render.clear(virtualSceneFramebuffer, 0f, 0f, 0f, 0f);  
for (WrappedAnchor wrappedAnchor : wrappedAnchors) {  
 Anchor anchor = wrappedAnchor.getAnchor();  
 Trackable trackable = wrappedAnchor.getTrackable();  
 if (anchor.getTrackingState() != TrackingState.TRACKING) {  
 continue;  
 }  
  
 // Get the current pose of an Anchor in world space. The Anchor pose is updated  
 // during calls to session.update() as ARCore refines its estimate of the world.  
 anchor.getPose().toMatrix(modelMatrix, 0);  
  
 // Calculate model/view/projection matrices  
 Matrix.multiplyMM(modelViewMatrix, 0, viewMatrix, 0, modelMatrix, 0);  
 Matrix.multiplyMM(modelViewProjectionMatrix, 0, projectionMatrix, 0, modelViewMatrix, 0);  
  
 // Update shader properties and draw  
 virtualObjectShader.setMat4("u\_ModelView", modelViewMatrix);  
 virtualObjectShader.setMat4("u\_ModelViewProjection", modelViewProjectionMatrix);  
  
 if (trackable instanceof InstantPlacementPoint  
 && ((InstantPlacementPoint) trackable).getTrackingMethod()  
 == InstantPlacementPoint.TrackingMethod.SCREENSPACE\_WITH\_APPROXIMATE\_DISTANCE) {  
 virtualObjectShader.setTexture(  
 "u\_AlbedoTexture", virtualObjectAlbedoInstantPlacementTexture);  
 } else {  
 virtualObjectShader.setTexture("u\_AlbedoTexture", virtualObjectAlbedoTexture);  
 }  
  
 render.draw(virtualObjectMesh, virtualObjectShader, virtualSceneFramebuffer);  
}  
  
// Compose the virtual scene with the background.  
backgroundRenderer.drawVirtualScene(render, virtualSceneFramebuffer, Z\_NEAR, Z\_FAR);*

*// Calculate spatial geographic location in real time.*

*Anchor anchor = hit.createAnchor();  
  
if(null == rootPose){  
  
 rootPose = anchor.getPose();  
  
} else {  
  
  
 //Calculate displacement.*

*Pose curPose = anchor.getPose();  
  
 float difX = curPose.tx() - rootPose.tx();  
 float difY = curPose.ty() - rootPose.ty();  
 float difZ = curPose.tz() - rootPose.tz();  
 distance = Math.sqrt((double) (difX \* difX + difY \* difY + difZ \* difZ));  
}  
  
  
  
wrappedAnchors.add(new WrappedAnchor(anchor, trackable));  
// For devices that support the Depth API, shows a dialog to suggest enabling  
// depth-based occlusion. This dialog needs to be spawned on the UI thread.  
this.runOnUiThread(this::showOcclusionDialogIfNeeded);*

### Data management

The data is saved on the local disk of the mobile phone. Location:

*context.getExternalFilesDir("")!!.absolutePath+File.separator*

Method for saving data.

*if(TextUtils.isEmpty(filePath) || TextUtils.isEmpty(data)) return  
  
Observable.create<Boolean> {  
  
 var writeSink: Sink? = null;  
 var bufferedSink: BufferedSink? = null;  
  
 try{  
  
 var file: File = File(filePath)  
 if(!file.parentFile.exists()){  
 file.parentFile.mkdirs()  
 }  
  
 if(!file.exists()){  
 file.parentFile.createNewFile()  
 }  
  
 //Obtain sink.  
 writeSink = file.sink(append = true)  
 bufferedSink = writeSink.buffer()  
 //Append and write data.  
 bufferedSink.writeUtf8(data)  
  
  
  
 }catch (e: Exception){  
 e.printStackTrace()  
 } finally {  
  
 try{  
  
 bufferedSink?.close()  
 writeSink?.close()  
  
 }catch (e: Exception){  
 e.printStackTrace()  
 }  
 }  
  
  
}.subscribeOn(Schedulers.io())  
 .subscribe()*

### Technical highlights and advantages

Open source and customizability: The Android system is based on the open source Linux kernel. Its open source feature enables developers to deeply understand the system architecture and underlying code and deeply customize the system according to their own needs, such as modifying the system interface, optimizing performance, and adding personalized functions, to meet the diverse needs of different users and enterprises.

Efficient multithreading and asynchronous processing: Supports multithreading programming, allowing developers to perform time-consuming operations in the background, such as network requests, data loading, file downloads, etc., without blocking the main thread, thus ensuring the fluency and responsiveness of the application. At the same time, various asynchronous processing mechanisms and tools such as AsyncTask, Handler, ThreadPoolExecutor, etc. are provided to facilitate developers to manage multithreading and asynchronous tasks.

Sensor and hardware interaction: It can easily interact with various sensors of the device, such as acceleration sensor, gyroscope sensor, light sensor, geographic location sensor, etc. Developers can use these sensor data to implement various innovative functions, such as motion monitoring, environmental perception, navigation and positioning, etc., to provide users with more intelligent and personalized services.

Access the powerful ARCore to build an AR virtual scene and realize motion tracking and real-time calculation of spatial geographic location, making data acquisition more realistic and intelligent.

## References

1. mobile sensor

<https://developer.android.google.cn/develop/sensors-and-location/sensors/sensors_motion?hl=zh-cn>

1. google ArCore

<https://developers.google.cn/ar/develop/getting-started?hl=zh-cn>

1. ArPose

<https://developers.google.cn/ar/reference/c/group/ar-pose>