دانشکده مهندسی کامپیوتر

استاد درس: دکتر ابوالفضل دیانت

بهار ۱۴۰۳

گزارش پروژه رودیوم
درس شبکه های تلفن همراه

فاطمه نیک پایان راد
شماره دانشجویی: ۹۹۵۲۲۲۱۲

آنیتا تلخابی
شماره دانشجویی: ۹۹۵۲۲۲۸۴

Rhodium

هدف از این پروژه ایجاد یک اپلیکیشن اندروید است که نقاط کور در پوشش شبکه‌های تلفن همراه را در محیط‌های درون بنا شناسایی و نمایش دهد. این برنامه به کاربران اجازه می‌دهد تا نقشه‌ای از محیط یا ساختمان موردنظر را به اپلیکیشن بدهند و سپس با حرکت در محیط و استفاده از سنسورهای مختلف گوشی، مسیر حرکت کاربر و پارامترهای مرتبط با توان دریافتی شبکه را ثبت کند. نتایج به دست آمده بر روی نقشه نمایش داده می‌شود و نقاط با توجه به کیفیت سیگنال دریافتی با رنگ‌های مختلف نشان داده می‌شوند.
پروژه اصلی در برنچ main قرار دارد که میتوان ران کرد. اما یک برنچ تست(gps-fined-location) هم برای استفاده از gps داریم که خب چونکه خطای آن بالا بود از آن استفاده نشده.

۱ طراحی و توسعه رابط کاربری

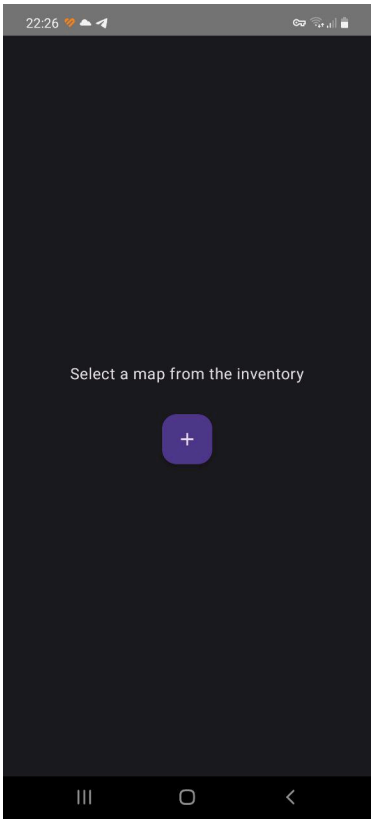
۱.۱ وارد کردن نقشه

کاربر باید بتواند یک نقشه از محیط یا ساختمان را به اپلیکیشن اضافه کند. این نقشه باید در صفحه گوشی به کاربر نمایش داده شود. یعنی ما یک نقشه به برنامه می دهیم مثلا یک عکس. این نقشه یک scale دارد. حالا به هر pixel نقشه یک مختصات نسبت میدهیم.

استفاده از Intent برای باز کردن گالری و انتخاب نقشه پس در ui خواهیم داشت:

```
Spacer(modifier = Modifier.height(16.dp))
FloatingActionButton(
    onClick = {
        val intent = Intent(Intent.ACTION_PICK, MediaStore.Images.Media.EXTERNAL_CONTENT_URI)
        launcher.launch(intent)
    },
    modifier = Modifier.align(Alignment.CenterHorizontally)
) {
    Icon(Icons.Default.Add, contentDescription = "Add Map")
}
```

شکل ۱: ورودی



شکل ۲: ورود عکس

در ابتدا لازم بوده یه سری متغیرهایی را تعریف بکنیم. استفاده از SensorEventListener جهت مدیریت رویدادهایی هست که توسط حسگرها مدیریت میشوند.

```
class MainActivity : ComponentActivity(), SensorEventListener {
    private lateinit var sensorManager: SensorManager
```

شکل ۳: مدیریت حسگرها

حال لزوم به تعریف متغیر های مهم داریم:
متغیرهای مربوط به حسگرها شامل ،accelerometer، gyroscope و magnetometer تعریف و مقداردهی اولیه شدند.
متغیر مربوط به پایگاه داده نیز با نام database تعریف و مقداردهی اولیه شد.

متغیر initialLocation برای ذخیره مکان اولیه.
متغیر currentLocation برای ذخیره مکان فعلی.
متغیر previousTime برای ذخیره زمان قبلی.
متغیرهای velocityX و velocityY برای ذخیره سرعت در جهت‌های X و Y.
متغیر accelValues برای ذخیره داده‌های شتاب‌سنج.
متغیر gyroValues برای ذخیره داده‌های ژیروسکوپ.
متغیر magnetValues برای ذخیره داده‌های مغناطیس‌سنج.
ثابت alpha برای فیلتر کردن داده‌های حسگر.
ثابت movementThreshold برای تشخیص حرکت.
ثابت updateInterval برای تنظیم فاصله زمانی به‌روزرسانی.
ثابت friction برای تنظیم مقدار اصطکاک.
ثابت deadZone برای تعیین نواحی بدون حرکت.
و در نهایت یک MutableState با نام routeState برای نگهداری اطلاعات مربوط به مسیر تعریف شد.

```
class MainActivity : ComponentActivity(), SensorEventListener {
    private lateinit var sensorManager: SensorManager
    private var accelerometer: Sensor? = null
    private var gyroscope: Sensor? = null
    private var magnetometer: Sensor? = null

    private lateinit var database: AppDatabase

    private var initialLocation: Pair<Float, Float>? = null
    private var currentLocation: Pair<Float, Float>? = null
    private var previousTime: Long = System.currentTimeMillis()
    private var velocityX = 0f
    private var velocityY = 0f
    private var accelValues = floatArrayOf(0f, 0f, 0f)
    private var gyroValues = floatArrayOf(0f, 0f, 0f)
    private var magnetValues = floatArrayOf(0f, 0f, 0f)

    // Constants for filtering and movement detection
    private val alpha = 0.8f // for low-pass filter
    private val movementThreshold = 20f // Increased threshold to reduce sensitivity
    private val updateInterval = 1000 // Update interval in milliseconds
    private val friction = 0.9f // Friction factor to gradually reduce velocity
    private val deadZone = 1.0f // Dead zone to ignore small movements

    // MutableState to hold the route
    private val routeState = mutableStateOf<List<Pair<Float, Float>>>(emptyList())
```

شکل ۴: متغیرها

در مرحله بعد استفاده میکنیم از متد OnCreate که خب برای درخواست مجوز از کاربر حیت ورود برای دسترسی به حافظه هایی مثل گالری کاربر است که ما لزوم آن را داریم برای آپلود عکس این اجازه صادر شود . در خطوط بعدی مقداردهی اولیه برای متغیرها رو داریم که در عکس قابل مشاهده است همچنین برای پایگاه داده که مخصوص به نقشه های ورودی ماست که در دیتابیس آن ذخیره میشود و بعدا میتوان دوباره از آن استفاده کرد.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    ActivityCompat.requestPermissions(
        this,
        arrayOf(Manifest.permission.READ_EXTERNAL_STORAGE),
        1
    )

    sensorManager = getSystemService(SENSOR_SERVICE) as SensorManager
    accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
    gyroscope = sensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE)
    magnetometer = sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD)

    database = Room.databaseBuilder(
        applicationContext,
        AppDatabase::class.java, "maps-database"
    ).build()
```

شکل ۵: متغیرها

در قسمت بعدی لازمه بدانیم MyScreen از دسته توابع composable هستند که برای طراحی ui درنظر گرفته شده و از RhodiumTheme توش استفاده کردیم.در نهایت هم دیتابیس نقشههای قبلی رو بهش پاس دادیم.

```
setContent {
    RhodiumTheme {
        Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
            MapScreen(
                database = database,
                modifier = Modifier.padding(innerPadding)
            )
```

شکل ۶: برای طراحی ui

در قسمت بعد برای فعال سازی و غیرفعال سازی حسگرهاست که در متدهای onResume و ،onPause ثبت و لغو ثبت لیسنر برای حسگرها برای مدیریت رویدادهای حسگرها زمانی که فعالیت فعال و غیرفعال می‌شود.مثلا در OnPause برای وقتی است که برنامه بسته میشه و قراره حسگرها غیرفعال شه و توی On-Resume برای اینه که حسگرها دوباره ریجیستر بشن.

```
override fun onResume() {
    super.onResume()
    accelerometer?.also { acc ->
        sensorManager.registerListener(this, acc, SensorManager.SENSOR_DELAY_NORMAL)
    }
    gyroscope?.also { gyro ->
        sensorManager.registerListener(this, gyro, SensorManager.SENSOR_DELAY_NORMAL)
    }
    magnetometer?.also { mag ->
        sensorManager.registerListener(this, mag, SensorManager.SENSOR_DELAY_NORMAL)
    }
}

override fun onPause() {
    super.onPause()
    sensorManager.unregisterListener(this)
}
```

شکل ۷: مدیریت حسگرها

در قسمت بعد onSensorChanged را داریم که کار اصلی آن این است که مثلا اون ایونت های شتاب سنج یا بقیه چیزها را میگیرد و در مقایسه با constant هایی که در بالاتر به آن متغیر ها اختصاص دادیم مقایسه میکند که مثلا اگر movement ما کمتر از عددی باشد در نظر گرفته نشود و برای به روزرسانی و مدیریت سرعت و موقعیت فعلی محاسباتی ریاضی انجام شده تا به بهترین حالت ممکن از خطاها جلوگیری و به حالت طبیعی شبیه سازی شود.

```
override fun onSensorChanged(event: SensorEvent?) {
    if (event == null || initialLocation == null) return

    val currentTime = System.currentTimeMillis()
    val deltaTime = (currentTime - previousTime) / 1000f // in seconds

    if (deltaTime < updateInterval / 1000f) {
        // Debounce updates to avoid too frequent updates
        return
    }

    previousTime = currentTime

    when (event.sensor.type) {
        Sensor.TYPE_ACCELEROMETER -> {
            // Apply high-pass filter to accelerometer data to remove gravity
            val gravity = 9.81f // approximate value of gravity
            accelValues[0] = alpha * accelValues[0] + (1 - alpha) * (event.values[0])
            accelValues[1] = alpha * accelValues[1] + (1 - alpha) * (event.values[1])
            accelValues[2] = alpha * accelValues[2] + (1 - alpha) * (event.values[2] - gravity)
            Log.d("SensorChanged", "Filtered Accelerometer: (${accelValues[0]}, ${accelValues[1]}, ${accelValues[2]})")
        }
        Sensor.TYPE_GYROSCOPE -> {
            gyroValues[0] = event.values[0]
            gyroValues[1] = event.values[1]
            gyroValues[2] = event.values[2]
            Log.d("SensorChanged", "Gyroscope: (${gyroValues[0]}, ${gyroValues[1]}, ${gyroValues[2]})")
        }
        Sensor.TYPE_MAGNETIC_FIELD -> {
```

شکل ۸: حساس به تغییرات

و همینطور در updatedRoute برای آپدیت لیست routeState است که در بالاتری تعریف شده که شامل ۲ عدد اعشاری یا همان x و y است که وقتی onsensorchaned حس بکنه حرکت کردیم میاد و یه نقطه جدید رو به این لیست اضافه میکنه.

```
private fun updateRoute(newLocation: Pair<Float, Float>) {
    val routeList = routeState.value.toMutableList()
    routeList.add(newLocation)
    routeState.value = routeList
}
```

شکل ۹: آپدیت نقطه

```
// MutableState to hold the route
private val routeState = mutableStateOf<List<Pair<Float, Float>>>(emptyList())
```

شکل ۱۰: لیست نقطه ها

در قسمت بعد myscreen را داریم که گفتیم از توابع composable هست و مدام داره صدا زده میشه و کال میشه حالا لازمه یه سری موارد رو که توی کد هست تعریف کنیم.
از remember برای مدیریت حالت‌های mapImageUri و tapLocation استفاده می‌شود. mapImageUri برای ذخیره آدرس تصویر نقشه و tapLocation برای ذخیره مکان ضربه‌های کاربر روی نقشه به کار می‌رود.

از LaunchedEffect برای ذخیره‌سازی تصویر نقشه در پایگاه داده استفاده می‌شود. این فرآیند تنها در صورتی انجام می‌شود که mapImageUri تغییر کند. با تغییر mapImageUri، LaunchedEffect فعال شده و تصویر نقشه به همراه اطلاعات آن در پایگاه داده ذخیره می‌شود.

از Column و LazyColumn برای نمایش لیستی از تصاویر نقشه استفاده می‌شود. LazyColumn لیستی از تصاویر نقشه را به صورت عمودی نمایش می‌دهد و کاربر می‌تواند از بین آن‌ها نقشه مورد نظر خود را انتخاب کند.

از Canvas برای رسم تصویر نقشه و مسیر کاربر با استفاده از داده‌های routeState استفاده می‌شود. Canvas امکان رسم نقشه و نمایش مسیر حرکت کاربر را فراهم می‌کند.

از Button برای بارگذاری تصویر نقشه استفاده می‌شود. با کلیک روی دکمه، کاربر می‌تواند یک تصویر نقشه جدید را از حافظه دستگاه انتخاب کرده و بارگذاری کند.

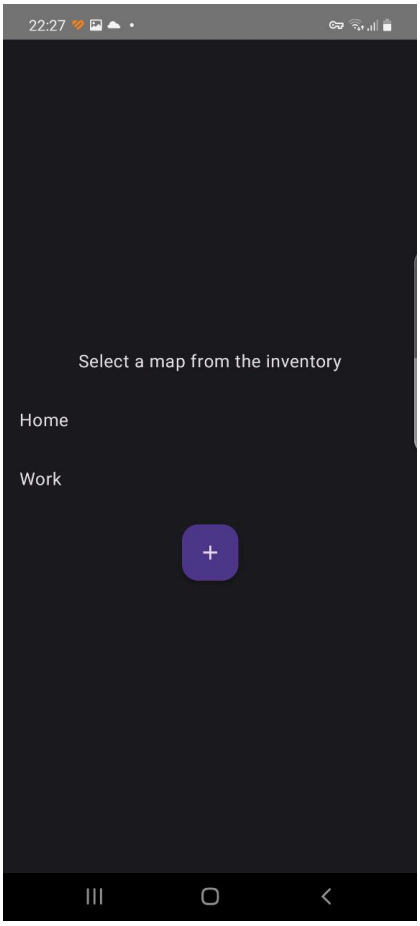
```
@SuppressLint("ProduceStateDoesNotAssignValue")
@Composable
fun MapScreen(database: AppDatabase, modifier: Modifier = Modifier) {
    var mapBitmap by remember { mutableStateOf<Bitmap?>(null) }
    val context = LocalContext.current
    var userLocation by remember { mutableStateOf<Pair<Float, Float>?>(null) }
    val scope = rememberCoroutineScope()

    var availableMaps by remember { mutableStateOf(emptyList<MapEntity>()) }

    LaunchedEffect(Unit) {
        withContext(Dispatchers.IO) {
            availableMaps = database.mapDao().getAll()
        }
    }

    var selectedMap by remember { mutableStateOf<String?>(null) }
    var showDialog by remember { mutableStateOf(false) }
    var newMapUri by remember { mutableStateOf<Uri?>(null) }
    var newMapName by remember { mutableStateOf("") }

    val launcher = rememberLauncherForActivityResult(ActivityResultContracts.StartActivityForResult()) { result ->
        if (result.resultCode == RESULT_OK) {
            val data: Intent? = result.data
            val uri = data?.data
            if (uri != null) {
                newMapUri = uri
                showDialog = true
            }
```

شکل ۱۱: my screen

در این قسمت برای اضافه کردن map هست که ازمون نوعش و اسم مپ رو میپرسه و توی دیتابیس ذخیره میکنه.که در عکس صفحه بعد خروجی در موبایل را میبینم:

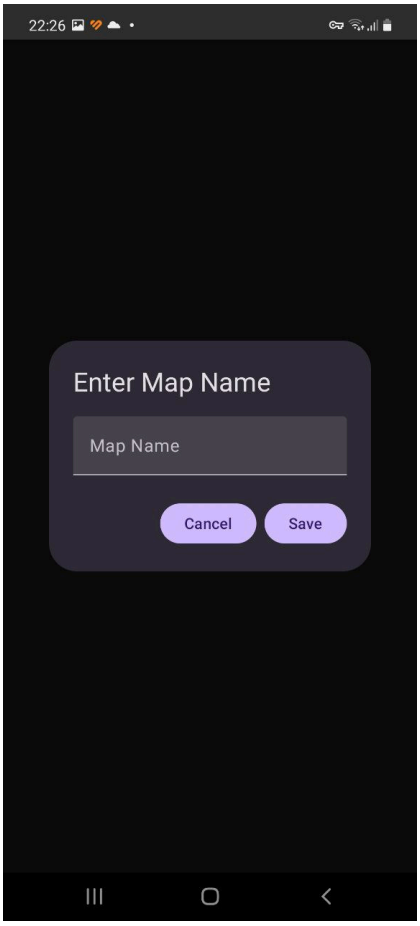
```
if (showDialog) {
    AlertDialog(
        onDismissRequest = { showDialog = false },
        title = { Text(text = "Enter Map Name") },
        text = {
            TextField(
                value = newMapName,
                onValueChange = { newMapName = it },
                label = { Text("Map Name") }
            )
        },
        confirmButton = {
            Button(
                onClick = {
                    val uri = newMapUri ?: return@Button
                    val inputStream = context.contentResolver.openInputStream(uri)
                    mapBitmap = BitmapFactory.decodeStream(inputStream)
                    scope.launch {
                        withContext(Dispatchers.IO) {
                            // Save map to the database with the user-provided name
                            database.mapDao().insert(MapEntity(name = newMapName, uri = uri.toString()))
                        }
                        saveMapUri(uri.toString(), context)
                        // Update availableMaps to include the newly added map
                        availableMaps = withContext(Dispatchers.IO) { database.mapDao().getAll() }
                    }
                    showDialog = false
```

شکل ۱۲: my screen



شکل ۱۳: ورودی نوع

همانطور که گفتیم composable ها درواقع eventbase هستند و اینقدر کال میشوند و هی لیسن میکنند اما اگر به یاد داشته باشیم در بالاتر در myscreen اومدیم و database رو صدا زدیم تا کوئری بزنه و مپ هارو برگردونه این با async انجام بشه و نمیشه از ui thread استفاده کرد که خب در withcontext –dispatch ers.io همینکار رو انجام میده که میره و توی یه thread دیگه ای این کار رو انجام میده و کوئری رو میزنه تا ui نایستد



شکل ۱۴: ورودی نام

```
LaunchedEffect(Unit) {
    withContext(Dispatchers.IO) {
        availableMaps = database.mapDao().getAll()
    }
}
```

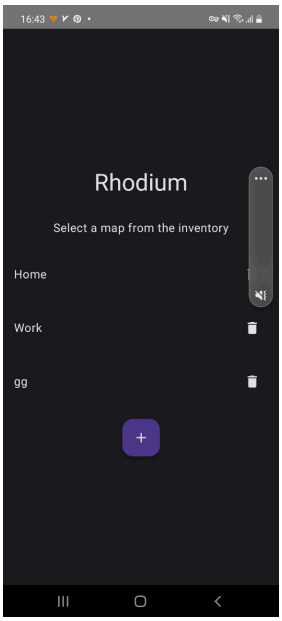
شکل ۱۵: withcontext dispatchers.io

از دیگر ویژگی هایی که حایز اهمیت است backhandler است که باید در هر شرایطی گزینه بازگشت به قبل برای کاربر فراهم باشد که در عکس زیر مشخص است.

```
) {
    if (mapBitmap != null) {
        BackHandler {
            mapBitmap = null
        }

        Box(
```

شکل ۱۶: back handler

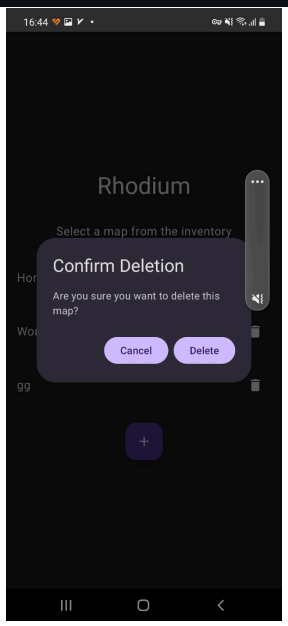
همانطور که در بالاتر گفتیم نقشه ها در دیتابیس دخیره میشود و هنگام ورود به کاربران نمایش داده میشود.



شکل ۱۷: نقشه های پیشین

همینطور امکان حذف کردن نقشه های پیشین برای کاربر فراهم شده تا از دیتابیس پاک کند و همچنین confirm هنگام پاک کردن داریم.

```
@Composable
fun DeleteConfirmationDialog(onConfirm: () -> Unit, onDismiss: () -> Unit) {
    AlertDialog(
        onDismissRequest = onDismiss,
        title = { Text("Confirm Deletion") },
        text = { Text("Are you sure you want to delete this map?") },
        confirmButton = {
            Button(onClick = onConfirm) {
                Text("Delete")
            }
        },
        dismissButton = {
            Button(onClick = onDismiss) {
                Text("Cancel")
            }
        }
```



شکل ۱۸: نقشه های پیشین

۲.۱ کلیک روی نقشه

کاربر میتواند روی نقشه کلیک کند تا نقطه موردنظر را مشخص کند. مختصات نقطه کلیک شده در دیتابیس ذخیره میشود.

در detectTapGestures برای تشخیص InitialLocation هست که با کلیک اول این InitialLoca-tion لاجیکش کال میشه و از اونجا به بعد شروع میشه به کشیدن بقیه، تا اولین کلیک صورت نگیرد بقیه جاها کشیده نمیشوند.

مورد بعدی که لازمه صحبت بشه راجب DrawRoute هست که همش در حال کال شدن است و روی لیست RouteState مدام کال میشود همان لیستی که شامل x و y بود پس به ازای اضافه شدن هرنقطه در لیست یک نقطه هم در صفحه اضافه میشه یعنی مدام در حال کشیدن آن لیست هستیم.

```
Box(
    modifier = Modifier
        .fillMaxSize()
        .pointerInput(Unit) {
            detectTapGestures { tapOffset ->
                userLocation = Pair(tapOffset.x, tapOffset.y)
                initialLocation = Pair(tapOffset.x, tapOffset.y)
                currentLocation = Pair(tapOffset.x, tapOffset.y)
                routeState.value = listOf(Pair(tapOffset.x, tapOffset.y))
                Log.d("MapScreen", "Initial Location: ($tapOffset.x, $tapOffset.y)")
            }
        }
) {
    Image(
        bitmap = mapBitmap!!.asImageBitmap(),
        contentDescription = "Map",
        modifier = Modifier.fillMaxSize()
    )

    DrawRoute(routeState.value)

    userLocation?.let { (x, y) ->
        Box(
            modifier = Modifier
                .offset(x.dp, y.dp)
                .size(10.dp)
                .background(Color.Green) // Initial location pin
        )
    }
```

شکل ۱۹: detectTapGestures

درون Box که نقشه را نمایش میدهد، LocationMarker برای نمایش مکان جاری کاربر فراخوانی میشود. در صورت وجود currentLocation، این مکان به LocationMarker ارسال میشود تا روی نقشه رسم شود.
در فایل marker.kt، LocationMarker تعریف شده است که به شکل زیر عمل میکند
از Canvas برای رسم نشانگر مکان استفاده میشود:

```
@Composable
fun LocationMarker(x: Float, y: Float) {
    Canvas(modifier = Modifier.size(40.dp)) {
        val radius = size.minDimension / 4
```

شکل ۲۰: marker.kt

رسم دایره‌های خارجی با افکت بلور:
ده دایره با رنگ آبی کمرنگ و شفافیت متفاوت رسم می‌شود تا افکت بلور ایجاد شود:

```
for (i in 1..10) {
    drawCircle(
        color = Color(0xFF007BB5).copy(alpha = 0.1f * (11 - i)),
        radius = radius * 1.7f * i / 10,
        center = Offset(x, y)
    )
}
```

شکل ۲۱: دایره

رسم دایره بیرونی:
یک دایره بیرونی با رنگ آبی روشن رسم می‌شود:

```
// Draw outer circle
drawCircle(
    color = Color(0xFFAEDFF7), // Light blue color
    radius = radius,
    center = Offset(x, y)
)
```

شکل ۲۲: دایره

رسم دایره داخلی:
یک دایره داخلی کوچکتر با رنگ آبی تیره‌تر رسم می‌شود:

```
    // Draw inner circle
    drawCircle(
        color = Color(0xFF007BB5), // Blue color
        radius = radius * 0.5f,
        center = Offset(x, y)
    )
}
```

شکل ۲۳: دایره

۲ ثبت مسیر حرکت و داده‌های مرتبط با شبکه

۱.۰.۲

لازم است بیان شود توضیحات مربوط به حرکت و جهت و چرخش و موقعیت یابی و همینطور حرکت روی نقشه در قسمت های آخر که مربوط به بیان مکان UE هست بیان شده است.

۱.۲ استفاده از سنسورهای گوشی

برنامه باید مسیر حرکت کاربر را با استفاده از سنسورهای مختلف گوشی (مانند GPS و شتاب‌سنج) تشخیص دهد و نمایش دهد.
از LocationManager برای دریافت موقعیت استفاده میکنیم و از SensorManager برای دریافت داده ها و سنسور ها.

۲.۲ رسم مسیر

DrawRoute.Composable.Function: تابعی که مسیر را بر اساس لیستی از زوج‌های مختصات Float رسم می‌کند.

```
@Composable
fun DrawRoute(route: List<Pair<Float, Float>>) {
    Canvas(modifier = Modifier.fillMaxSize()) {
        route.forEach { (x, y) ->
            drawCircle(
                color = Color.Red,
                radius = 7f,
                center = Offset(x, y),
                style = androidx.compose.ui.graphics.drawscope.Fill
            )
            Log.d("DrawRoute", "Drawing circle at: ($x, $y)")
        }
    }
}
```

```
    @Composable
    fun MapListItem(map: MapEntity, onClick: () -> Unit) {
        Text(
            text = map.name,
            modifier = Modifier
                .fillMaxWidth()
                .padding(16.dp)
                .clickable { onClick() }
        )
    }
}
```

شکل ۲۴: رسم

۳.۲ نمایش هر آیتم نقشه

MapListItem.Composable.Function: تابعی که مورد استفاده برای نمایش هر آیتم نقشه در لیست نقشه‌ها است.

```
    @Composable
    fun MapListItem(map: MapEntity, onClick: () -> Unit) {
        Text(
            text = map.name,
            modifier = Modifier
                .fillMaxWidth()
                .padding(16.dp)
                .clickable { onClick() }
        )
    }
}
```

شکل ۲۵: نمایش

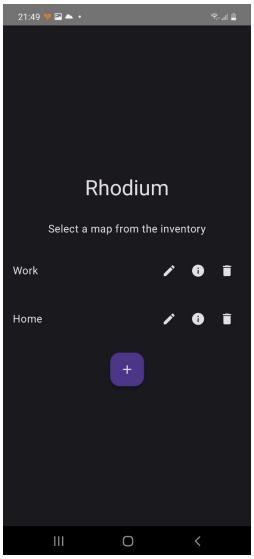
۴.۲ اصلاح مسیر توسط کاربر

۱.۴.۲ نمایش راهنما به کاربر

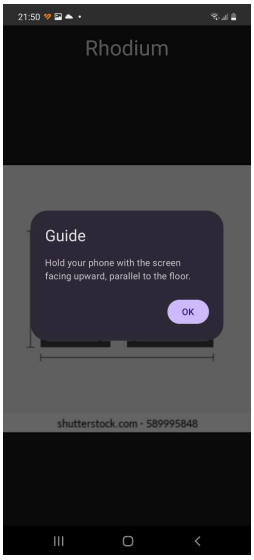
GuideDialog.Composable.Function: که برای نمایش راهنما به کاربر استفاده می‌شود.
در این قسمت وقتی که روی مداد کنار صفحه کلیک میشود همان نقشه نمایش داده میشود و به کاربر راهنمایی میشود که باید صفحه گوشی را چگونه تنظیم کند مثلا به صورت افقی بگیرد و در نهایت شروع ما با کلیک روی همین مداد آغاز میشود که باید به عنوان مکان اولیه یه کلیک رو صفحه بزنیم.

```
@Composable
fun GuideDialog(onDismiss: () -> Unit) {
    AlertDialog(
        onDismissRequest = onDismiss,
        title = { Text("Guide") },
        text = { Text("Hold your phone with the screen facing upward, parallel to the floor.") },
        confirmButton = {
            Button(onClick = onDismiss) {
                Text("OK")
            }
        }
```

شکل ۲۶: راهنما



شکل ۲۷: راهنما



شکل ۲۸: راهنما

اگر کاربر انحرافی در مسیر مشاهده کرد، می‌تواند با کلیک روی نقشه، نقطه درست را مجدداً مشخص کند. در کد به گونه ای تنظیم شده است با هر کلیک روی صفحه مکان کاربر آپدیت میشود.

۳ اندازه‌گیری پارامترهای شبکه

۱.۳ ثبت پارامترهای مرتبط با توان دریافتی

برنامه به صورت پیوسته پارامترهای مرتبط با توان دریافتی شبکه را اندازه‌گیری می‌کند. این پارامترها می‌توانند شامل توان دریافتی سلول خدمتگزار و سلول‌های همسایه باشند. به عنوان مثال:
RxLev، C۲، C۱ :GSM
EC/N۰، RSCP، :UMTS
CINR، RSRQ، RSRP :LTE
لازم است بیان شود باد توجه به نسخه استفاده شده اندروید امکان استخراج تمامی این پرامترها ممکن نخواهد بود.

لازم است از TelephonyManager استفاده کنیم برای دریافت اطلاعات شبکه مانند RSRQ، RSRP و هر آن چیز دیگری که لازم است از شبکه بگیریم. برای استفاده از بهره گیری از موقعیت و مکان کاربر و همینطور fusedLocationClient لازم است اطلاعات مربوط به این نقاط را از telephonymanager بگیریم پس در کد زیر درباره متغیر های لازم بر اساس لوکیشن و همینطور آپدیت چند ثانیه واسه به روز رسانی اطلاعات نوشته شده:

```
class MainActivity : ComponentActivity() {
    private lateinit var fusedLocationClient: FusedLocationProviderClient
    private lateinit var locationCallback: LocationCallback


    private val locationRequest = LocationRequest.create().apply { this: LocationRequest
        interval = 2000 // ثانیه 5
        fastestInterval = 2000 // ثانیه 2
        priority = LocationRequest.PRIORITY_HIGH_ACCURACY
    }

    private var locationState: MutableState<Location?> = mutableStateOf( value: null)
    private var cellInfoState: MutableState<List<String>?> = mutableStateOf( value: null)
    private var networkTypeState: MutableState<String> = mutableStateOf( value: "")
```

شکل ۲۹: مکان

۱.۱.۳ locationCallback

یک callback برای دریافت موقعیت مکانی تعریف می‌شود که با دریافت نتیجه موقعیت، اطلاعات سلولی نیز به‌روز می‌شود.

```
locationCallback = object : LocationCallback() {
    override fun onLocationResult(locationResult: LocationResult) {
        locationResult ?: return
        val location = locationResult.lastLocation
        locationState.value = location
        getCellInfo()
    }
}
```

شکل ۳۰: callback

در نهایت گرفتن اطلاعات مربوط به هر ۳ نسل ۲و۳و۴ بدین گونه خواهد بود و اطلاعات نهایی لوکیشن نهایی در callinfostate ذخیره میشه.

```
@SuppressLint("MissingPermission")
private fun getCellInfo() {
    val telephonyManager = getSystemService(Context.TELEPHONY_SERVICE) as TelephonyManager
    val cellInfoList = telephonyManager.allCellInfo
    val cellInfoStrings = mutableListOf<String>()

    cellInfoList?.forEach { cellInfo ->
        when (cellInfo) {
            is CellInfoLte -> {
                val cellIdentity = cellInfo.cellIdentity
                val cellSignalStrength = cellInfo.cellSignalStrength
                val cid = cellIdentity.ci
                val tac = cellIdentity.tac
                val rac = null
                val nci = null
                val rsrp = cellSignalStrength.rsrp
                val rsrq = cellSignalStrength.rsrq
                val sinr = if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
                    cellSignalStrength.rssnr
                } else {
                    "N/A"
                }
                val rss = cellSignalStrength.dbm // RSS (Received Signal Strength)
                cellInfoStrings.add("LTE - CID: $cid, TAC: $tac, RAC: $rac, NCI: $nci, RSRP: $rsrp, RSRQ: $rsrq, SINR: $sinr, RSS: $rss")
            }
```

شکل ۳۱: نسل ۴

```
is CellInfoWcdma -> {
    val cellIdentity = cellInfo.cellIdentity
    val cellSignalStrength = cellInfo.cellSignalStrength
    val cid = cellIdentity.cid
    val lac = cellIdentity.lac
    val rac = null
    val rscp = cellSignalStrength.dbm
      val ecio = if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
          cellSignalStrength.ecio
      } else {
          "N/A"
      }
    val ecio = "not possible to calculate"
    val rss = cellSignalStrength.dbm // RSS (Received Signal Strength)
    cellInfoStrings.add("WCDMA - CID: $cid, LAC: $lac, RAC: $rac, RSCP: $rscp, ECIO: $ecio, RSS: $rss")
}
```

شکل ۳۲: نسل ۳

```
is CellInfoGsm -> {
    val cellIdentity = cellInfo.cellIdentity
    val cellSignalStrength = cellInfo.cellSignalStrength
    val cid = cellIdentity.cid
    val lac = cellIdentity.lac
    val rac = null
    val rssi = cellSignalStrength.dbm
    val rss = cellSignalStrength.dbm // RSS (Received Signal Strength)
    cellInfoStrings.add("GSM - CID: $cid, LAC: $lac, RAC: $rac, RSSI: $rssi, RSS: $rss")
}
```

شکل ۳۳: نسل ۲

۲.۳ ثبت مکان و شناسه‌های سلول

اطلاعات مکان UE و شناسه‌های سلول خدمتگزار نظیر شناسه PLMN، LAC، RAC TAC و شناسه سلول به همراه پارامترهای اندازه‌گیری شده در پایگاه داده ذخیره می‌شوند.
درباره اطلاعات موقعیت حرکت آنچه لازم است گفته شود عدم استفاده از gps به علت خطای بسیار زیاد است و به همین علت از پارامتر هایی چون مغناطیس سنج و ژیروسکوپ استفاده شده.
مغناطیس‌سنج: از داده‌های مغناطیس‌سنج برای محاسبه آزیموت (جهت) استفاده می‌شود. آزیموت نشان‌دهنده زاویه بین شمال جغرافیایی و جهت رو به جلو دستگاه است.
ژیروسکوپ: در این قطعه کد به‌طور خاص داده‌های ژیروسکوپ استفاده نشده است، اما معمولاً ژیروسکوپ برای بهبود دقت جهت‌یابی استفاده می‌شود. این ممکن است در بخش‌های دیگر کد یا برای کاربردهای خاص‌تر مانند ردیابی دقیق چرخش‌ها استفاده شود.
محاسبه مکان: داده‌های شتاب‌سنج برای تشخیص حرکت (مثلاً راه رفتن) و به‌روزرسانی مکان فعلی استفاده می‌شود.

در ابتدا لازم است این متغیر های مقدار دهی شوند:

```
    )

    sensorManager = getSystemService(SENSOR_SERVICE) as SensorManager
    accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
    gyroscope = sensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE)
    magnetometer = sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD)

    database = Room.databaseBuilder(
```

شکل ۳۴: مقداردهی اولیه

۱.۲.۳ سنسورها

در onResume، سنسورها برای دریافت داده‌ها ثبت می‌شوند و در onSensorChanged پردازش آنها را خواهیم داشت: که کل کد آن بدین صورت است که در ادامه به توضیح قسمت های میپردازیم:

```
override fun onSensorChanged(event: SensorEvent?) {
    if (event == null || initialLocation == null) return

    when (event.sensor.type) {
        Sensor.TYPE_ACCELEROMETER -> {
            accelValues = event.values.clone()
            Log.d(
                "SensorChanged",
                "Accelerometer: (${accelValues[0]}, ${accelValues[1]}, ${accelValues[2]})"
            )
        }

        Sensor.TYPE_MAGNETIC_FIELD -> {
            magnetValues = event.values.clone()
            Log.d(
                "SensorChanged",
                "Magnetometer: (${magnetValues[0]}, ${magnetValues[1]}, ${magnetValues[2]})"
            )
        }
    }

    // Check if the phone is held horizontally (z-axis should be close to 9.8 or -9.8 for horizontal)
    if (accelValues[2] < horizontalThreshold && accelValues[2] > -horizontalThreshold) {
        Log.d("SensorChanged", "Phone is not horizontal. Ignoring movement.")
        return
    }

    // Calculate rotation matrix and orientation angles
    SensorManager.getRotationMatrix(rotationMatrix, null, accelValues, magnetValues)
    SensorManager.getOrientation(rotationMatrix, orientationAngles)
    val azimuth = Math.toDegrees(orientationAngles[0].toDouble()).toFloat()
    Log.d("SensorChanged", "Azimuth: $azimuth")
```

شکل ۳۵: رفتار سنسور ها

```
val direction = when {
    azimuth in -45.0..45.0 -> "East"
    azimuth in 45.0..135.0 -> "South"
    azimuth < -45.0 && azimuth >= -135.0 -> "North"
    azimuth > 135.0 || azimuth < -135.0 -> "West"
    else -> "Unknown"
}
Log.d("SensorChanged", "Phone Direction: $direction")

// Smooth the accelerometer data using a low-pass filter
val alpha = 0.8f
smoothedAccelValues[0] = alpha * smoothedAccelValues[0] + (1 - alpha) * accelValues[0]
smoothedAccelValues[1] = alpha * smoothedAccelValues[1] + (1 - alpha) * accelValues[1]

// Calculate the magnitude of the smoothed accelerometer vector for x and y only
val accelMagnitude =
    sqrt(smoothedAccelValues[0] * smoothedAccelValues[0] + smoothedAccelValues[1] * smoothedAccelValues[1])
Log.d("SensorChanged", "Smoothed Accelerometer Magnitude: $accelMagnitude")

// Check if the user is walking
if (accelMagnitude > walkingThreshold) {
    // Update velocities based on azimuth (direction)
    when (direction) {
        "East" -> {
            velocityX = deltaMove
            velocityY = 0f
        }

        "South" -> {
            velocityX = 0f
            velocityY = deltaMove
        }

        "North" -> {
            velocityX = 0f
            velocityY = -deltaMove
        }

        "West" -> {
            velocityX = -deltaMove
            velocityY = 0f
        }
    }
    Log.d("SensorChanged", "User is walking. Updated Velocities: ($velocityX, $velocityY)")

    // Update the current location based on the velocities
    currentLocation?.let { (x, y) ->
        val newX = x + velocityX
        val newY = y + velocityY
        currentLocation = Pair(newX, newY)
        // Update the route state to add the new location
        updateRoute(Pair(newX, newY))
        Log.d("SensorChanged", "New Location: ($newX, $newY)")
    }
} else {
    // User is not walking, stop movement
    velocityX = 0f
    velocityY = 0f
    Log.d("SensorChanged", "User is not walking. Velocities set to zero.")
}
```

شکل ۳۶: رفتار سنسور ها

این قسمت مربوط به محاسبه ماتریس چرخش و زوایای جهت‌گیری است و جهت حرکت در آن مشخص میشود به گونه ای که رو نقشه چپ و راست و بالا و پایین حرکت میکند طبق آنچه حرکت کنیم:

```
SensorManager.getRotationMatrix(rotationMatrix, null, accelValues, magnetValues)
SensorManager.getOrientation(rotationMatrix, orientationAngles)
val azimuth = Math.toDegrees(orientationAngles[0].toDouble()).toFloat()
Log.d("SensorChanged", "Azimuth: $azimuth")
```

```
val direction = when {
    azimuth in -45.0..45.0 -> "East"
    azimuth in 45.0..135.0 -> "South"
    azimuth < -45.0 && azimuth >= -135.0 -> "North"
    azimuth > 135.0 || azimuth < -135.0 -> "West"
    else -> "Unknown"
}
Log.d("SensorChanged", "Phone Direction: $direction")
```

شکل ۳۷: جهت و چرخش

۳.۳ از بین بردن نویز با alpha

فیلتر پایین‌گذر برای صاف کردن داده‌های شتاب‌سنج : در کد پایین، داده‌های شتاب‌سنج با استفاده از یک فیلتر پایین‌گذر صاف می‌شوند. هدف از صاف کردن داده‌های شتاب‌سنج کاهش نویز و نوسانات ناخواسته در داده‌های خام است که از سنسور شتاب‌سنج به دست می‌آیند. این کار باعث می‌شود که سیگنال اصلی بهتر نمایان شود و حرکات واقعی کاربر با دقت بیشتری تشخیص داده شود.
مقدار alpha ضریبی بین ۰ و ۱ است که میزان تاثیر داده های جدید و قبلی را تعیین میکند در اینجا مقدار alphaa برابر با ۸.۰ است که به این معنی است که ۸۰ درصد از مقدار قبلی و ۲۰ درصد از مقدار جدید استفاده میشود
مقدار صاف‌شده محور X به صورت زیر محاسبه می‌شود:
smoothedAccelValues[۰] مقدار قبلی صاف‌شده برای محور X است. accelValues[۰] مقدار جدیدی است که از شتاب‌سنج دریافت شده است. این فرمول باعث می‌شود که مقدار جدید به مقدار قبلی اضافه شده و میانگین وزنی از آن‌ها گرفته شود.
مقدار صاف‌شده برای محور Y به همان روش محاسبه می‌شود:
smoothedAccelValues[۱] مقدار قبلی صاف‌شده برای محور Y است. accelValues[۱] مقدار جدیدی است که از شتاب‌سنج دریافت شده است.

```
// Smooth the accelerometer data using a low-pass filter
val alpha = 0.8f
smoothedAccelValues[0] = alpha * smoothedAccelValues[0] + (1 - alpha) * accelValues[0]
smoothedAccelValues[1] = alpha * smoothedAccelValues[1] + (1 - alpha) * accelValues[1]

// Calculate the magnitude of the smoothed accelerometer vector for x and y only
val accelMagnitude =
    sqrt(smoothedAccelValues[0] * smoothedAccelValues[0] + smoothedAccelValues[1] * smoothedAccelValues[1])
Log.d("SensorChanged", "Smoothed Accelerometer Magnitude: $accelMagnitude")
```

شکل ۳۸: از بین بردن نویز

در نهایت به ترتیب بررسی میشود که حرکت داریم یا خیر و در نهایت به‌روزرسانی سرعت‌ها بر اساس آزیموت (جهت)

```
// Check if the user is walking
if (accelMagnitude > walkingThreshold) {
    // Update velocities based on azimuth (direction)
    when (direction) {
        "East" -> {
            velocityX = deltaMove
            velocityY = 0f
        }

        "South" -> {
            velocityX = 0f
            velocityY = deltaMove
        }

        "North" -> {
            velocityX = 0f
            velocityY = -deltaMove
        }

        "West" -> {
            velocityX = -deltaMove
            velocityY = 0f
        }
    }
    Log.d("SensorChanged", "User is walking. Updated Velocities: ($velocityX, $velocityY)")
```

شکل ۳۹: شرط حرکت

در ابتدا به ترتیب به‌روزرسانی مکان فعلی بر اساس سرعت‌ها و به‌روزرسانی وضعیت مسیر با افزودن مکان جدید را داریم:

```
    // Update the current location based on the velocities
    currentLocation?.let { (x, y) ->
        val newX = x + velocityX
        val newY = y + velocityY
        currentLocation = Pair(newX, newY)
        // Update the route state to add the new location
        updateRoute(Pair(newX, newY))
        Log.d("SensorChanged", "New Location: ($newX, $newY)")
    }
} else {
    // User is not walking, stop movement
    velocityX = 0f
    velocityY = 0f
    Log.d("SensorChanged", "User is not walking. Velocities set to zero.")
}
```

شکل ۴۰: شرط حرکت

۴ نمایش داده‌ها بر روی نقشه

۱.۴ نقشه Offline

کاربر می‌تواند داده‌های ذخیره شده را بر روی نقشه به صورت برون خط مشاهده کند. که در اینجا اطلاعاتی که مخصوص هرنسل داشتیم با توجه به موقعیت کاربر آپدیت شده و نمایش داده میشود.
اطلاعات این مسیر و نقشه در دیتابیس ذخیره شده اند و با هربار ورود به این نقشه در دیتابیس میتوان اطلاعات قبلی را مشاهده نمود. ولی در نقشه اطلاعاتی چون موقعیت و توان دریافتی نمایش داده میشود و بقیه پارامتر ها در دیتابیس ذخیره میشوند همگی.

۲.۴ نمایش نقاط رنگی

مکان‌های مختلف به صورت نقاط رنگی بر روی نقشه نشان داده می‌شوند. رنگ هر نقطه نشان‌دهنده کیفیت سیگنال دریافتی در آن مکان است:
سبز: سیگنال خوب
زرد: سیگنال متوسط
نارنجی: سیگنال ضعیف
قرمز: سیگنال خیلی ضعیف
سیاه: عدم موفقیت در اندازه‌گیری

برای تعیین این رنگ ها هربازه ای از rss یا توان دریافتی به رنگی اختصاص داده شده که اگر توان ما در آن بازه قرار بگیرد رنگ آن تغییر خواهد کرد و همینطور راهنمایی برای رنگ ها در گوشه صفحه در نظر گرفته شده است و همینطور خود rss در گوشه صفحه نمایش داده میشود و میتوان مقدار آن را دید.

۵ توضیحات برنامه با شروع از نقطه آغاز تا انتها

۱.۵ ورود نقشه که به صورت عکس است.

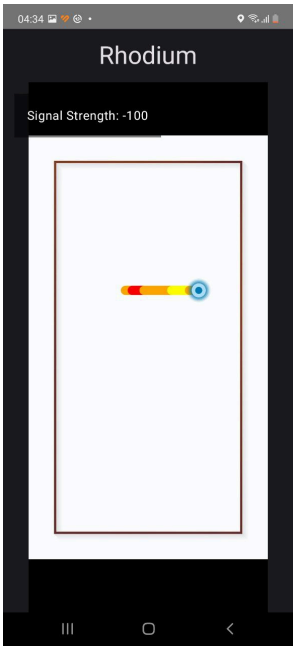
۲.۵ امکان انتخاب نقشه های پیشین و گرفتن دیتاهای پیشین.

۳.۵ شروع با کلیک روی مداد یا آپلود عکس جدید.

۴.۵ باید نقطه شروع را کلیک کنیم.

۵.۵ گوشی باید صاف افقی گرفته شود و کج نشود چونکه ممکنه موجب حرکت های اشتباه شود و دچار خطا شویم پس گوشی افقی باشد و در دست میگیریم و حرکت میکنیم.

۶.۵ خروجی توان دریافتی و راهنما نقشه و رنگ های راهنما گوشه نقشه نشان داده شده است.



شکل ۴۱: نقشه

همانطور که مشاهده میشود در ازای حرکت و تغییر سیگنال ورودی که بالا مشخص شده رنگ ها بر اساس قدرت سیگنال تغییر میکنند و اساس این تغییر این چنین است.

```
import androidx.compose.ui.graphics.Color

enum class RouteColor(val color: Color) {
    GREEN(Color.Green),
    YELLOW(Color.Yellow),
    ORANGE(Color(0xFFFFA500)), // Orange color
    RED(Color.Red),
    BLACK(Color.Black),
    NOCOLOR(Color.Transparent)
}

fun getColorByIndex(index: Int): Color {
    return when (index) {
        1 -> RouteColor.GREEN.color
        2 -> RouteColor.YELLOW.color
        3 -> RouteColor.ORANGE.color
        4 -> RouteColor.RED.color
        5 -> RouteColor.BLACK.color
        else -> RouteColor.NOCOLOR.color
    }
}
```

شکل ۴۲: نگ ها

۷.۵ این اطلاعات در دیتابیس ذخیره میشوند.