Faculdade de Ciências Exatas e da Engenharia

"2023 / 2024"

"Projeto/Estágio"

Licenciatura em Engenharia Informática

Trabalho realizado por:

Diogo Martins, 2082921

"junho de 2024"

Índice

1. Introdução

Foi-me proposto a implementação de uma aplicação *Android* que visa auxiliar a movimentação dentro da universidade da Madeira, com a ajuda de realidade aumentada.

a. Motivação

Sendo aluno de Engenharia Informática, procuro aumentar o meu conhecimento e experiência dentro das várias áreas.

Assim, decidi aceitar este projeto, proposto pela empresa *Ethical Alghorithm*, orientado pelo engenheiro Bernardo Luís e pelo professor Felipe Quintal, em que tive a oportunidade de desenvolver em Android e usar recursos de realidade aumentada, áreas novas em que não tinha qualquer experiência.

b. Problema

Passamos grande parte do nosso tempo dentro de quatro paredes, ainda assim, a navegação interna continua complexa em sítios como aeroportos, terminais, centros comerciais e até mesmo na universidade, levando a atrasos, más experiências e consumo excessivo de energia. A quantidade de pessoas, os vários caminhos e a pressa de chegar ao destino são fatores que aumentam a complexidade na navegação.

A realidade aumentada tem visto a sua popularidade a crescer, juntamente com a constante evolução das tecnologias moveis. Assim, diante este cenário, surgiu a ideia de criar uma ferramenta, juntando esta tecnologia com a capacidade de localização e orientação dos dispositivos móveis para auxiliar a navegação dos seus utilizadores, que irá auxiliar os utilizadores a obterem informações sobre a sua localização atual e que simplifique a sua navegação nestes espaços.

O resultado será, através da câmara do smartphone, mostrar uma camada virtual, sobreposta ao ambiente real, as informações sobre o espaço.

2. Solução

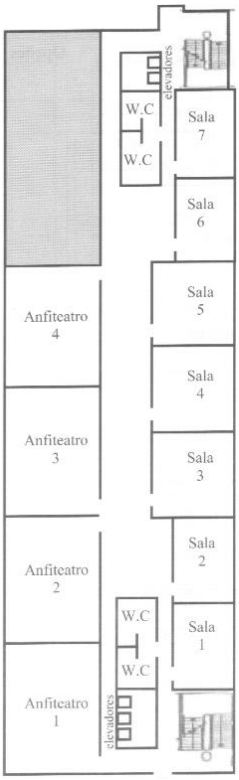
A app final consistirá num menu inicial onde serão apresentados uma lista de locais, onde o utilizador poderá escolher o seu destino, de seguida, através do seu local atual, apresentado um caminho para o utilizador poder seguir. Para a arquitetura do sistema, foi usado o modelo de arquitetura *MVVM (Model-View-ViewModel)*, que basicamente serve para separar a interface do utilizador da lógica de apresentação.

a. Setup da Câmara/deteção de QRCodes

O primeiro passo da solução foi o *setup* da câmara, através da biblioteca '*CameraX*,' que facilita o desenvolvimento de apps com recurso à câmara. De seguida, de modo ao utilizador receber informações sobre a sua posição atual, foi decidido utilizar um código QR, que o utilizador fará o scan, recebe a sua localização atual e de seguida calculado e apresentado um caminho até ao destino pretendido pelo utilizador.

b. Algoritmo para calcular um caminho

O passo seguinte foi, com base na posição atual do utilizador, e o destino escolhido, calcular o melhor caminho para o utilizador seguir. Primeiramente, foi necessário mapear um piso da universidade.




```
val floor = arrayOf(
    /*      x//0  1  2  3  4  //y */
    intArrayOf(0, 1, 1, 1, 0),//0
    intArrayOf(0, 1, 0, 1, 0),//1
    intArrayOf(0, 1, 0, 1, 0),//2
    intArrayOf(0, 1, 1, 1, 0),//3
    intArrayOf(0, 0, 1, 0, 0),//4
    intArrayOf(0, 0, 1, 0, 0),//5
    intArrayOf(0, 0, 1, 0, 0),//6
    intArrayOf(0, 0, 1, 0, 0),//7
    intArrayOf(0, 0, 1, 0, 0),//8
    intArrayOf(0, 0, 1, 0, 0),//9
    intArrayOf(0, 0, 1, 0, 0),//10
    intArrayOf(0, 0, 1, 0, 0),//11
    intArrayOf(0, 0, 1, 0, 0),//12
    intArrayOf(0, 1, 1, 1, 0),//13
    intArrayOf(0, 1, 0, 1, 0),//14
    intArrayOf(0, 1, 0, 1, 0),//15
    intArrayOf(0, 1, 1, 1, 0)//16
)
```

Na imagem à direita podemos ver um mapa do piso -2 da universidade, ilustrado na imagem à esquerda. No mapa, as posições com valores a 1 representam as posições onde é possível andar. O algoritmo de busca foi baseado no algoritmo 'Busca em Profundidade', um algoritmo que começa num nó raiz e explora um caminho até ao seu final, retrocede e explora novos caminhos. A ideia para esta busca será, em cada instante, dar valores às posições onde será possível se movimentar, escolhendo aquela com maior valor como próxima a ser explorada. A posição atual é colocada a zero a atual para evitar retornos. Também foi necessário guardar todas as posições já visitadas para evitar ciclos infinitos.

Esta abordagem permite identificar o melhor caminho para que o utilizador possa seguir e chegar ao seu destino.

c. Visualização do caminho

Finalmente, foi implementado a realidade aumentada de modo a orientar os utilizadores através da visualização do caminho. Em primeiro lugar, os modelos 3D foram criados. Foi utilizando o Blender, um software de modelagem tridimensional para criar estes modelos simples. Para a execução da realidade aumentada, foi adotada a *framework 'Sceneview'*, desenvolvida de modo a dar continuidade à descontinuada '*Sceneform'* da *Google*. Esta ferramenta facilita a integração do *ARCore*, o *kit* de desenvolvimento de realidade aumentada para *Android*

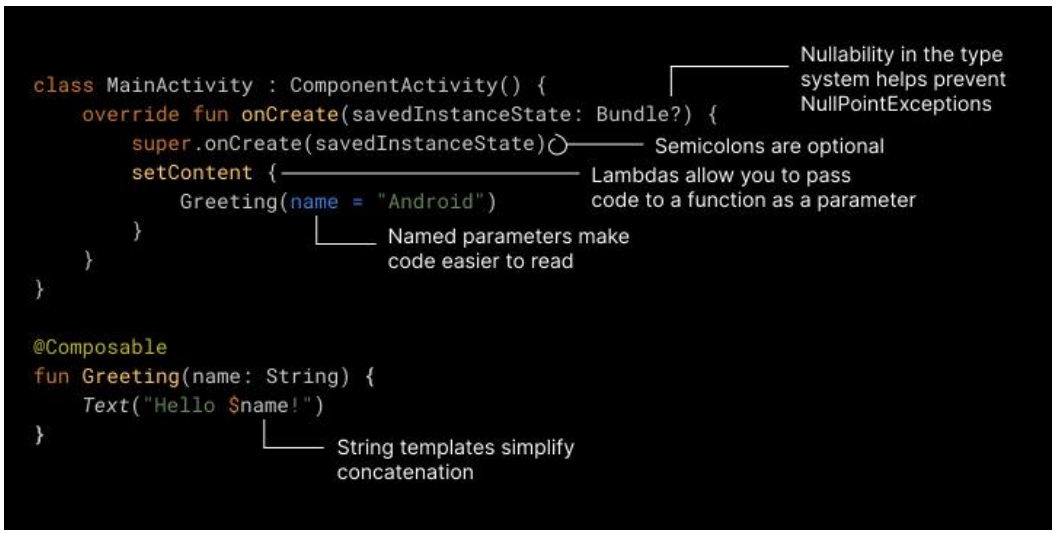
Em primeiro lugar foi necessário criar uma view espefica, "*ArSceneView*", onde se define tudo relacionado com a realidade aumentada. Comecei por definir o ciclo de vida, alinhado com o cilco de vida da view onde será implementado, depois o planeRenderer foi configurado, que, como o nome indica controla a renderização de planos. De seguida, a sessão do *ARCore* foi configurada para incluir deteção de profundidade, estimativa de luz e deteção de planos. Essas configurações são essenciais para melhorar a perceção do ambiente físico e garantir que os objetos virtuais sejam ancorados de forma precisa e estável.

Finalmente, a cada atualização da sessão, é capturado o *frame* da imagem e adicionado uma âncora à cena, onde serão anexados os modelos 3D seguindo o caminho previamente calculado. Cada posição é calculada a direção do modelo, com base na posição seguinte.

3. Revisão de Literatura

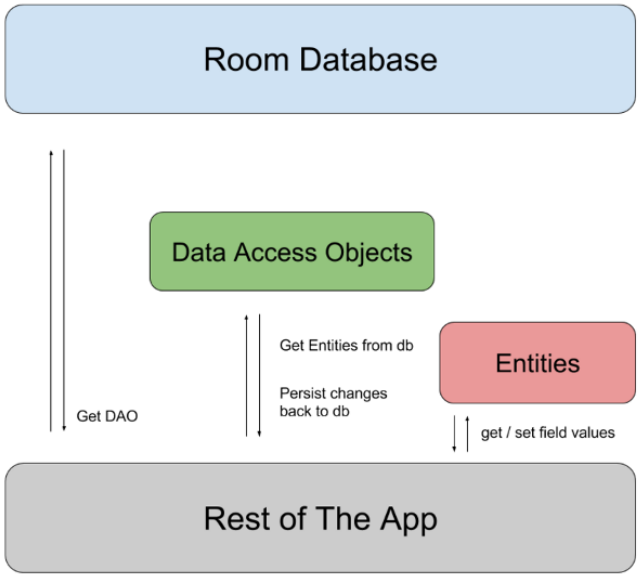
a. Kotlin

Kotlin é uma linguagem de programação moderna desenvolvida pela *JetBrains*. É uma linguagem concisa, orientada a objetos, criada para interoperar com o *Java*, permitindo a utilização de bibliotecas Java, mas com muitas melhorias a nível de sintaxe e funcionalidade. Em *Android*, é a principal linguagem de programação por suportar corrotinas, facilitando programação assíncrona e concorrente, inclui uma abordagem rigorosa para exceções *NullPointerException*, bastante problemáticas em aplicações *Java*, entre outras vantagens.



b. *Room Database*

De modo a dar uma lista ao utilizador dos locais disponíveis para que possa escolher o seu destino. Para tal, foi criada uma base de dados para guardar as localizações. Foi usado o *Room Database*, que proporciona uma camada de abstração sobre o *SQLite* permitindo um acesso fácil à base de dados, mas, aproveitando toda a capacidade do *SQLite*. No *Room*, existem 3 componentes principais:



- Entidades: Mapeiam as tabelas da base de dados da aplicação, definindo a estrutura e os campos dos dados que serão armazenados.
- *DAO (Data Access Object)*: Define as *querys* de modo a inserir, atualizar, consultar e excluir dados da base de dados. Serve como interface entre a aplicação e a base de dados.
- Classe da Base de Dados: Define a base de dados e serve como ponto de acesso para a conexão com os dados persistentes da app.

Após a definição da base de dados e a inserção dos locais, foi implementada uma *RecyclerView* para apresentar os locais aos utilizadores, juntamente com uma *search bar*. Desta forma, é oferecido ao utilizador uma forma eficiente e flexível de pesquisar e selecionar o local que deseja se dirigir.

c. *MLKit*

O *MLKit* é um kit de *machine learning* para desenvolvedores de dispositivos móveis de um modo avançado e fácil de utilizar, combinando vários modelos de *machine learnig* com *pipelines* de processamento avançado e *APIs.*

Algumas das *APIs* fornecidas são a *APIs Vision*, fornece análise de vídeo e imagem, permitindo rotular imagens, detetar código de barras, textos, rostos e objetos. *APIs* de linguagem natural, que fornece o processamento de linguagem natural que indetifica e traduz entre 58 idiomas e fornece sugestões de respostas.

Neste caso, foi usado de modo a identificar um código QR, que terá a informação do local exato onde o utilizador se encontra.

d. *ARCore/ Sceneview framework*

O *ARCore* é o *SDK* de realidade aumentada da *Google* que oferece APIs para criar aplicações com recurso a realidade aumentada no *Android, iOS, Unity e Web.* Oferece ferramentas como rastreamento de movimento, âncoras, para garantir a fixação de um objeto no espaço, entendimento ambiental, ou seja, deteta o tamanho e localização de superfícies, processamento de profundidade e estimativa leve sobre intensidade e correção de cor do ambiente.

A *Sceneview* é um *framework* para criar aplicações com recursos 3D e AR utilizando *Jetpack Compose* e *Layout View* com base no *ARCore* e *Google Filament.* Foi criada para simplificar a integração de elementos virtuais no mundo real

4. Implementação

Nesta secção será abordada os detalhes de como foi feita a implementação da solução descrita previamente.

a. Requisitos

O primeiro passo foi definir alguns requisitos base que o sistema deverá de seguir para poder cumprir o seu objetivo e ter sucesso. Estes requisitos serão funcionais, de tecnologia e não funcionais.

- Requisitos funcionais:

Os requisitos funcionais descrevem as funções que o sistema deverá realizar e estabelecem o comportamento normal do sistema.

  - O sistema deverá mostrar uma lista de todos os locais disponíveis
  - O sistema deverá permitir ao utilizador escolher um local de destino
  - O sistema deverá permitir ao utilizador pesquisar um local específico
  - O sistema deverá apresentar a posição atual do utilizador
  - O sistema deverá apresentar o melhor caminho possível desde a posição do utilizador até ao destino escolhido

- Requisitos de tecnologia:

Os requisitos de tecnologia ditam as ferramentas que serão usadas para o desenvolvimento do sistema.

  - Deverá ser utilizado o *IDE Android Studio*
  - Deverá ser utilizado a linguagem de programação *Kotlin*
  - Deverá ser utilizado a *framework Sceneview*
  - Deverá ser usada a biblioteca *CameraX*

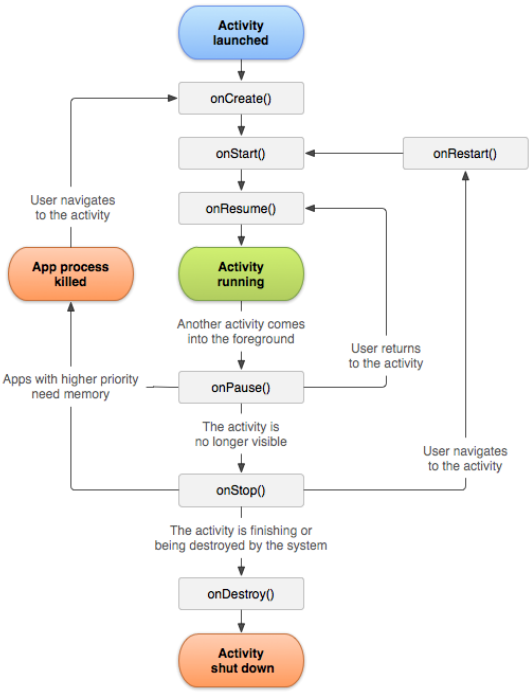
- Requisitos não funcionais:

Os requisitos não funcionais dizem respeito aos aspetos da app relacionados com desempenho, confidencialidade, segurança e disponibilidade.

  - O sistema deverá fornecer a posição atual do utilizador instantaneamente
  - O sistema deverá renderizar o caminho entre1 a 5 segundos depois de detetar o plano.
  - O sistema deverá fornecer uma forma de retroceder em caso de falhas na renderização do caminho.

b. Processo

Para desenvolver a solução, primeiro foi necessário começar por definir a atividade onde iria correr todo o fluxo da aplicação e os fragmentos e *view-model* correspondentes, que representam os comportamentos das *views* do sistema. A criação da atividade passou por definir o seu ciclo de vida, através do *override* dos métodos do ciclo de vida e estabelecimento de um *Fragment Container*, que irá tratar as transições dos fragmentos.



```kotlin
class MainActivity : AppCompatActivity() {
    private var _binding: ActivityMainBinding? = null
    private val navController: NavController by lazy {
        (supportFragmentManager
            .findFragmentById(R.id.home_fragment_container) as NavHostFragment).navController
    }
    Diogo Martins
    private val binding: ActivityMainBinding
        get() = _binding!!
    Diogo Martins *
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val binding = ActivityMainBinding.inflate(layoutInflater)
        _binding = binding
        setContentView(binding.root)
        setSupportActionBar(binding.toolbar)
        supportActionBar?.setDisplayHomeAsUpEnabled(true)
        binding.toolbar.setNavigationOnClickListener { it: View!
            onBackPressed()
        }
        setupActionBarWithNavController(navController)
    }
    Diogo Martins
    override fun onDestroy() {
        _binding = null
        super.onDestroy()
    }
    new *
    companion object {
        private val TAG = MainActivity::class.simpleName
    }
}
```

```xml
<androidx.fragment.app.FragmentContainerView
    android:id="@+id/home_fragment_container"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:name="androidx.navigation.fragment.NavHostFragment"
    app:defaultNavHost="true"
    app:navGraph="@navigation/main"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@id/toolbar"
    app:layout_constraintBottom_toBottomOf="parent"
    android:background="@color/white"/>
```

O primeiro fragmento é o menu inicial, onde é apresentado os locais disponíveis para o utilizador se dirigir. Como já foi explicado, os locais são guardados através do *Room Database*, e são apresentados com uma *RecyclerView*. Cada instância da tabela tem colunas mapeadas para os atributos que representam o nome e coordenadas, x, y e z, do local.

```kotlin
@Entity(tableName = TABLE_NAME)
data class LocationTable(@ColumnInfo(name = LOCAL_NAME) val name: String,
                         @ColumnInfo(name = LOCAL_X) val x: Int,
                         @ColumnInfo(name = LOCAL_Y) val y: Int,
                         @ColumnInfo(name = LOCAL_Z) val z: Int,) {
    @PrimaryKey(autoGenerate = true) @ColumnInfo(name = LOCAL_ID)
    var id: Int = 0
```

Na criação da classe da base de dados as entidades são especificadas com a anotação *@Database*

```kotlin
@Database(entities = [LocationTable::class], version = 1)
abstract class DataBase: RoomDatabase() {
    Diogo Martins
    abstract fun LocationDao(): LocationDao
    Diogo Martins
    companion object{
        private var instance: DataBase ?= null
        private const val dbName = "locations.db"
        Diogo Martins
        fun getDataBase(context: Context): DataBase {
            if (instance == null) {
                synchronized(DataBase::class) {
                    instance = Room
                        .databaseBuilder(context,
                            DataBase::class.java,
                            dbName)
                        .build()
                }
            }
            return instance!!
        }
```

Para a criação da *RecyclerView* é criado uma *view* específica para os elementos, estendendo o um *ViewHolder* específico da *RecyclerView* definindo a aparência e comportamento de cada elemento da lista.

```
class LocationsViewHolder(private val binding: RecyclerViewLocationBinding):
    RecyclerView.ViewHolder(binding.root) {
    var location : Location?= null
    Diogo Martins
    fun bind (locationsViewModel: LocationsViewModel) {
        binding.apply { this: RecyclerViewLocationBinding
            locationsViewModel.apply { this: LocationsViewModel
                locationName.text = name
                locationsFloor.text = floor
                cardView.setOnClickListener(this)
            }
        }
    }
    Diogo Martins *
    companion object {
        private val TAG = LocationsViewHolder::class.java.simpleName
        val LAYOUT_ID = R.layout.recycler_view_location
        Diogo Martins *
        fun create(viewGroup: ViewGroup): LocationsViewHolder =
            LocationsViewHolder(RecyclerViewLocationBinding.inflate(
                viewGroup.context.layoutInflater,
                viewGroup,
                attachToParent: false))
    }
}
```

Para associar os dados às *views*, usamos um adaptador, estendendo a classe *Adapter* da *RecyclerView*, que solicita e vincula as visualizações aos dados chamando os dados definidos nesta classe.

```
fun deleteItem(item: ItemViewModel) {
    val currentList = currentList.toMutableList()
    currentList.remove(item)
    submitList(currentList)
}
Diogo Martins
fun addNewItems(newItems: List<ItemViewModel>) {
    val currentList = currentList.toMutableList()
    currentList.addAll(newItems)
    submitList(currentList)
}
```

```
override fun getItemViewType(position: Int): Int =
    getItem(position).viewType
Diogo Martins *
override fun onCreateViewHolder(
    parent: ViewGroup,
    viewType: Int): RecyclerView.ViewHolder =
    when(viewType) {
        LocationsViewHolder.LAYOUT_ID ->
            LocationsViewHolder.create(parent)
        else ->
            throw IllegalArgumentException("$TAG, viewType Not Coded.")
    }

Diogo Martins *
override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position: Int) {
    val item = getItem(position)
    when {
        item is LocationsViewModel && holder is LocationsViewHolder ->
            holder.bind(item)
        else ->
            Log.w(TAG, msg: "ViewModel or ViewHolder not recognized.")
    }
}
```

Depois de criada a lista onde ficarão os locais, é necessário criá-los, adicionar à base de dados, chamá-los e adicionar à *RecyclerView*. Para tal, são chamados. no viewModel os métodos definidos no *DAO*, e adicionados através dos métodos do *Adapter*. Primeiramente, chamamos os métodos *DAO, by lazy*, que serve para fazer inicialização preguiçosa, ou seja, o valor da variável será calculado e atribuído apenas quando for acessado pela primeira vez, De seguida, criado uma instância da tabela e introduzido na base de dados.

```
private val Dao by lazy {
    DataBase.getDataBase(app.applicationContext).LocationDao()
}

val local1 = LocationTable( name: "Anfiteatro 1",  x: 0,  y: 16,  z: -2)
Dao.insert(local1)

 @Insert(onConflict = OnConflictStrategy.IGNORE)
 suspend fun insert(location: LocationTable)
```

Para apresentar a lista dos locais, chamamos um método que devolve a lista completa dos elementos da base de dados. De seguida, chamamos um método do *Adapter* para submeter essa lista e apresentar as *views* esses elementos.

```
val data = Dao.getLocals().map { LocationsViewModel(it) }
withContext(mainDispatcher) { this: CoroutineScope
    locationsAdapter.addNewItems(data)
}

@Query("SELECT * FROM ${LocationTable.TABLE_NAME} ORDER BY ${LocationTable.LOCAL_NAME}")
suspend fun getLocals(): List<LocationTable>
```

Para além desta lista, é possível encontrar locais através através de pesquisa. Esta pesquisa é implementada colocando um *observer* na *searchBar*, e, sempre que é registada alguma alteração, é chamado um método *DAO*, que procura na base de dados alguma entrada em que o nome contém os caracteres observados.

```
val onTextWritten: LiveData<String>
    get() = _onTextWritten
private val _onTextWritten: MutableLiveData<String> = MutableLiveData( value: "")

val searchTextWatcher = object : TextWatcher{
    override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int) = Unit

    override fun afterTextChanged(s: Editable?) = Unit

    override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {
        val value = s?.toString() ?: return
        _onTextWritten.value = value
    }
}
 Diogo Martins
override fun onEditorAction(v: TextView?, actionId: Int, event: KeyEvent?): Boolean {
    return when (actionId) {
        EditorInfo.IME_ACTION_SEARCH -> {
            // TODO: Search in Web
            Log.d(TAG,  msg: "final value: ${_onTextWritten.value}")
            false
        }
        else -> {
            Log.w(TAG,  msg: "This action is not supposed to happen!, Check your code!")
            true
        }
    }
}
```

```
onTextWritten.observe(viewLifecycleOwner){ name ->
    if (name.isNotEmpty()) {
        getDBdataByName(name)
    }
}

inputText.apply{ this: TextInputEditText
    addTextChangedListener(viewModel.searchTextWatcher)
    setOnEditorActionListener(viewModel)
}

@Query ("SELECT * FROM ${LocationTable.TABLE_NAME} " +
        "WHERE ${LocationTable.LOCAL_NAME}=:name " +
        "ORDER BY ${LocationTable.LOCAL_NAME}")
suspend fun getLocal(name: String): LocationTable
```

Depois do utilizador escolher o seu destino, é feito a navegação para o próximo fragmento, onde poderá fazer o scan de um código QR para obter a sua localização atual e de seguida visualizar o caminho até o seu destino. A navegação é feita num ficheiro *XML*, que gere as atividades, fragmentos e as navegações, tal como os argumentos que passarão de um fragmento para outro.

```
<fragment
    android:id="@+id/scanner_fragment"
    android:name="com.example.wayfinderar.ui.ScannerFragment"
    android:label="Scanner"
    tools:layout="@layout/fragment_scanner">
    <action
        android:id="@+id/action_scanner_fragment_to_start_route"
        app:destination="@+id/start_route_fragment"/>
    <argument
        android:name="finalLocation"
        app:argType="com.example.wayfinderar.Model.Location" />
    <action
        android:id="@+id/action_scanner_fragment_to_ar_fragment"
        app:destination="@id/ar_fragment" />
</fragment>
```

Neste caso, como temos um *RecyclerView*, e quero passar o local específico, a navegação será feita no *viewModel* destes locais em vez de no fragmento, ao selecionar o local de destino.

```
override fun onClick(v: View?) {
    location = transform.locationTable_to_location(locationTable)
    val route = HomeFragmentDirections.actionHomeToScannerFragment(location!!)
    when(v?.id){
        R.id.card_view->v.findNavController().navigate(route)
    }
}
```

Neste fragmento, começo por iniciar a câmera, configurando o *cameraSelector*, que define qual câmera será usada, neste caso a traseira, o *cameraProvider*, que vincula o ciclo de vida com o da atividade, o *imageAnalysis*, que irá analisar os *frames* da câmera, e o *previewBuilder*, configura a pré-visualização da câmera.

```
private val previewBuilder: Preview.Builder = Preview.Builder()
    .setResolutionSelector(ResolutionSelector.Builder()
        .setAspectRatioStrategy(AspectRatioStrategy.RATIO_4_3_FALLBACK_AUTO_STRATEGY)
        .build())

@Throws(NullPointerException::class)
fun startCamera(
    context: Context?,
    lifecycleOwner: LifecycleOwner?,
    cameraSelector: CameraSelector = CameraSelector.DEFAULT_BACK_CAMERA,
    imageAnalyzer: ImageAnalysis.Analyzer? = null,
    imageAnalysisBuilder: ImageAnalysis.Builder = ImageAnalysis.Builder()
        .setBackpressureStrategy(ImageAnalysis.STRATEGY_KEEP_ONLY_LATEST)
cameraProviderFuture.addListener({
    // Used to bind the lifecycle of cameras to the lifecycle owner
    cameraProvider = cameraProviderFuture.get()

cameraProvider?.apply { this: ProcessCameraProvider
    // Unbind use cases before rebinding
    unbindAll()
    camera = bindToLifecycle(
        lifecycleOwnerNotNull,
        cameraSelector,
        previewUseCase,
        imageAnalysisUseCase
    )
}
```

Quanto ao *ImageAnalysis*, em primeiro lugar é criado um objeto, *'QRCodeAnalyser'* para analisar os códigos QR, que será uma extensão de '*ImageAnalysis.Analyze'* e feito *override* da função '*analyse'* para processar cada frame de imagem capturado pela câmera e detectar códigos QR.

```
@SuppressLint("UnsafeOptInUsageError")
override fun analyze(image: ImageProxy) {
    val inputImg = image.image
    val rotation = image.imageInfo.rotationDegrees
    if (inputImg != null){
        val img =
            InputImage.fromMediaImage(inputImg, rotation)
        scanner.process(img)
            .addOnSuccessListener { qr ->
                qr.forEach{ code ->
                    listeners?.invoke(code.displayValue)
                }
                image.close()
            }
            .addOnFailureListener{ it: Exception
                image.close()
            }
    }
}
```

Então, é inicializado um objeto de *QRCodeAnalyser*, que irá ser ligado ao *imageAnalysis* da câmera. O método *tryEmit()*, diz respeito ao *SharedFlow*, que é, basicamente, uma funcionalidade que permite a emissão e coleta de valores de forma assíncrona, ficando o valor emitido disponível para qualquer observador do Sh*aredFlow*. Neste caso, o uso desta ferramenta é pensado para que se possa emitir eventos de forma assíncrona, sem se armazenar os valores.

```
qrCodeAnalyser = QRCodeAnalyser { string ->
    string?.let { result ->
        Log.d(TAG, msg: "RESULTADO:  $result")
        _result.tryEmit(result)
    }
}

// Setup Image Analyzer
val imageAnalysisUseCase =
    imageAnalysisBuilder.build().apply { this: ImageAnalysis
        if (imageAnalyzer != null) {
            setAnalyzer(
                cameraExecutor,
                imageAnalyzer
            )
        }
    }.also { it: ImageAnalysis
        it.setAnalyzer(cameraExecutor, qrCodeAnalyser!!)
    }
this.imageAnalysis = imageAnalysisUseCase
```

O código QR contém os dados da localização no formato “nome/x/y/z”, sendo os valores separados pelas “/” e criado um objeto do tipo ‘*Location’* e enviado para o próximo fragmento, juntamente com o destino, da mesma forma que foi navegado desde o fragmento inicial até ao atual. Sendo o *‘result’* um ‘*SharedFlow’,* como já foi mencionado, será necessário um ‘*LifecycleScope’,* uma vez que os dados são emitidos de forma assíncrona. O ‘*throttleFirst’* indica que durante um período de 2 segundos, apenas será emitido o primeiro evento.

```
result
    .throttleFirst( windowMillis: 2000)
    .collectOnLifecycleScope(viewLifecycleOwner) { value ->
    //QRCode format --> NAME/X/Y/Z
    val valueArray = value.split( ...delimiters: "/").toTypedArray()
    if (value.isNotEmpty()) {
        startLocation = Location(
            valueArray[0],
            valueArray[1].toInt(),
            valueArray[2].toInt(),
            valueArray[3].toInt()
        )
        val route2 = ScannerFragmentDirections
            .actionScannerFragmentToArFragment(startLocation!!, finalLocation!!)
        findNavController().navigate(route2)
    }
}
```

Já no fragmento seguinte, que irá apresentar o caminho ao utilizador, a primeira ação será calcular o caminho até ao destino. O cálculo, como já foi explicado no capítulo da 'Solução', o algoritmo foi baseado no algoritmo de 'Busca em Profundidade' em que temos um mapa bidimensional em que as posições que representam onde se pode andar estão representadas com 1. O primeiro passo é percorrer o mapa e atribuir ao destino um valor absurdamente grande, de modo a ser possível distinguir o destino das restantes posições. Ainda foi dado valores extremamente baixos às posições inacessíveis, por segurança, para evitar problemas, como o caminho seguir por essas posições.

```
for (i in floor_02.indices){
    for (j in floor_02[i].indices){
        if (i == destination.y && j == destination.x) {
            floor_02[i][j] = 1000
        }
        if (floor_02[i][j] == 0){
            if (!(i == destination.y && j == destination.x)){
                floor_02[i][j] = -1000
            }
        }
    }
}
```

De seguida começa a pesquisa pelo caminho. Em cada instância, a posição atual é guardada e calculado os valores da posição atual e das posições nos arredores da atual. À posição atual é subtraído 17 valores e as restantes é somado 8, escolhendo a posição com maior valor para a próxima a explorar.

```
while ((actualPos.x != destination!!.x || actualPos.y != destination!!.y) && (checkVisited(actualPos) == 0)){
    nodesVisited += actualPos
    path += actualPos
    value = updateFloor(floor_02, actualPos.x, actualPos.y)
    actualPos = nextMove(floor_02, value, actualPos,  control: 1) //is possible to add to positionsSaved
    logFloor02()
}

private fun updateFloor(floor: Array<IntArray>, x: Int, y: Int): Int{
    var value = 0
    Log.d(TAG,  msg: "X: $x, Y: $y")
    floor[y][x] = floor[y][x] - 17

    if (x<floor[y].size-1) {
        floor[y][x + 1] += 8
        value = floor[y][x + 1]
    }
    if (y<floor.size-1) {
        floor[y + 1][x] += 8
        if (floor[y + 1][x]>value) value = floor[y + 1][x]
    }
    if (x>0) {
        floor[y][x - 1] += 8
        if ( floor[y][x - 1] > value) value =  floor[y][x - 1]
    }
    if (y>0) {
        floor[y - 1][x] += 8
        if (floor[y - 1][x] > value) value = floor[y - 1][x]
    }
    Log.d(TAG,  msg: "VALUE:.....$value")
    return value
}
```

Sempre que é há a possibilidade de ir por dois caminhos diferentes, ou seja, duas posições têm o mesmo valor, a posição atual é guardada para mais tarde explorar a segunda alternativa. Guardar a posição depende se passamos na posição pela primeira vez ou se estamos a voltar a trás e se a posição já foi previamente guardada ou não.

```
var actualPos = nextMove(floor_02, value, start, Control)//the possibility to add to positionsSaved depends
                                                         //possible if it´s the first time
                                                         //not possible if it´s called when returning a position
                                                         //then it's always 0

var c = control
for (i in positionsSaved) {
    if (i.x == position.x && i.y == position.y)
        c = 0
    }
}
if (c == 1) {
    positionsSaved += position
}
```

Para além disso, foi necessário guardar todas as posições visitadas para evitar andar aos círculos.

```
private fun checkVisited(position: Location): Int{
    var visited = 0
    for (i in nodesVisited){
        if (i.x == position.x && i.y == position.y){
            Log.d(TAG, msg: "ALREADY VISITED: $position")
            visited = 1
        }
    }
    return visited
}
```

Quando se chega ao destino final, caso seja a primeira vez, é guardado à parte, e as vezes seguintes é comparado o tamanho dos caminhos e aquele com menor caminho é guardado. A busca acaba caso não aja nós para retornar, caso aja, é voltado atrás no caminho atual.

```
while ((actualPos.x != destination!!.x || actualPos.y != destination!!.y) && (checkVisited(actualPos) == 0)){
    nodesVisited += actualPos
    path += actualPos
    value = updateFloor(floor_02, actualPos.x, actualPos.y)
    actualPos = nextMove(floor_02, value, actualPos, control: 1) //is possible to add to positionsSaved
    logFloor02()
}
if ((path.size < finalPath.size || finalPath.isEmpty()) && (checkVisited(actualPos) == 0)){
    finalPath = path
}
```

```
if (positionsSaved.isNotEmpty()){
    floor_02[destination!!.y][destination!!.x] = 1000
    actualPos = positionsSaved[positionsSaved.size-1]
    positionsSaved = positionsSaved.dropLast( n: 1).toMutableList()
    for ((i, value) in path.withIndex()){
        if (value.x == actualPos.x && value.y == actualPos.y){
            path = path.toMutableList().subList(0, i+1)          //return to the crossing
            break
        }
    }
    if ((actualPos.x == destination!!.x && actualPos.y == destination!!.y)) path = mutableListOf(start)
    path(actualPos)  //doesn't add the position to the savedPositions
}
```

Depois de calcular o caminho, o próximo passo é apresentá-lo aos utilizadores. Começamos por calcular as direções dos objetos em cada posição. Em cada posição é verificado a posição seguinte, e dado uma direção, caso seja a última posição, é dado a direção da anterior.

```
fun calculateDirection(path:MutableList<Location>):Array<String>{
    var directions = arrayOf<String>()
    var direction = ""
    var i = 0
    for (i in path) Log.d(TAG,  msg: "directions: ${i.x}, ${i.y}")
    while (i<path.size-1){
        if(path[i+1].x > path[i].x) {
            direction = "RIGHT"
            orientationH = 0f
            orientationL = 0f
        }
        if(path[i+1].x < path[i].x) direction = "LEFT"
        if(path[i+1].y > path[i].y) direction = "DOWN"
        if(path[i+1].y < path[i].y) direction = "UP"
        i++
        directions += direction
    }
    directions += directions[i-1]
    return directions
}
```

Depois, foi implementada uma '*ArSceneView'* no fragmento, onde toda a realidade aumentada referente ao caminho apresentado ao utilizador será exibida.

```
<io.github.sceneview.ar.ARSceneView
    android:id="@+id/scene_view"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Então, foi necessário configurar esta *view*, primeiro associamos o ciclo do fragmento ao da *view*, depois, foi configurado o *'planeRenderer',* que controla a renderização de planos e controlar se os '*Renderables'* fazem sombra sobre os planos,

neste caso, o '*planeRenderer*' e as sombras foram ativadas, mas a sua visibilidade foi desativada. Foi também configurada a sessão, ativando o modo de profundidade, caso seja suportado é configurado como automático, caso contrário fica desativado, o modo de deteção do plano e o modo de estimativa da luz. Ainda foi ativada a oclusão em profundidade e em caso de falha, a razão é capturada e armazenda.

```
sceneView.apply { this: ARSceneView
    lifecycle = this@ArFragment.viewLifecycleOwner.lifecycle
    planeRenderer.apply { this: PlaneRenderer
        isEnabled = true
        isShadowReceiver = true
        isVisible = false
    }
    configureSession { session, config ->

        config.depthMode =
            if (session.isDepthModeSupported(Config.DepthMode.AUTOMATIC)) {
                Config.DepthMode.AUTOMATIC
            } else {
                Config.DepthMode.DISABLED
            }
        config.planeFindingMode = Config.PlaneFindingMode.HORIZONTAL_AND_VERTICAL
        config.lightEstimationMode = Config.LightEstimationMode.ENVIRONMENTAL_HDR
    }

cameraStream?.isDepthOcclusionEnabled = true
onTrackingFailureChanged = { reason ->
    this@ArFragment.trackingFailureReason = reason
}
```

Depois de configurada a *view*, o próximo passo será renderizar os modelos e apresentar o caminho ao utilizador. Para tal, é ativado um *callback* sempre que a sessão é atualizada, e é obtido o *frame* atual e, por cada elemento no caminho, é criado um nó com o modelo com a direção correta. Para alterar a direção do modelo é aplicado rotação sobre o mesmo. As distâncias entre os modelos são, também, alteradas consoante a direção de cada modelo.

```
onSessionUpdated = { _, frame ->
    frame.getUpdatedPlanes() (Mutable)Collection<Plane!>
        .firstOrNull { it.type == Plane.Type.HORIZONTAL_UPWARD_FACING } Plane?
        ?.let { plane ->
            lifecycleScope.launch { this: CoroutineScope
                if((ang > orientationL && ang < orientationH) || n > 0 ) {
                    if (n < pathSize) {
                        addAnchorNode(
                            plane.createAnchor(plane.centerPose),
                            directions[n]
                        )
                    } else if (pathSize == n) addAnchorNode(
                        plane.createAnchor(plane.centerPose),
                        directions[pathSize - 1],
                        final: true
                    )
                }
                n++
            }
        }
}
```

```
suspend fun addAnchorNode(anchor: Anchor, direction: String, final: Boolean = false) {
    binding?.sceneView?.apply { this: ARSceneView
        addChildNode(
            AnchorNode(engine, anchor)
                .apply { this: AnchorNode
                    isEditable = false
                    buildModelNode(direction, final)?.let { addChildNode(it) }
                    val attachmentManager =
                        ViewAttachmentManager(
                            this@ArFragment.requireContext(),
                            this@ArFragment.binding?.sceneView
                        )
                    attachmentManager.onResume()
                    anchorNode = this
                }
        )
    }
}
```

```
binding?.sceneView?.apply { this: ARSceneView
    modelLoader.loadModelInstance(
        modelNode
    )?.let { modelInstance ->
        val modelNode = ModelNode(
            modelInstance = modelInstance,
            //scaleToUnits = 0.1f,
        )
        modelNode.isTouchable = false
        modelNode.transform = transform
        return modelNode
    }
}
```

```
val transform = when (direction) {
    "RIGHT" -> Transform(Position(xModel, yModel, zModel), Rotation( x: 0f,  y: 270f,  z: 0f), Scale( v: 0.1f))
    "LEFT" -> Transform(Position(xModel, yModel, zModel), Rotation( x: 0f,  y: 90f,  z: 0f), Scale( v: 0.1f))
    "UP" -> Transform(Position(xModel, yModel, zModel), Rotation( x: 0f,  y: 0f,  z: 0f), Scale( v: 0.1f))
    "DOWN" -> Transform(Position(xModel, yModel, zModel), Rotation( x: 0f,  y: 180f,  z: 0f), Scale( v: 0.1f))
    else -> Transform(Position(xModel, yModel, zModel), Rotation( x: 0f,  y: 0f,  z: 0f), Scale( v: 0.1f))
}
```

```
val distanceX = 4.5f
val distanceZ = 1f
zModel += when (direction) {
    "RIGHT" -> distanceZ
    "LEFT" -> -distanceZ
    else -> 0f
}

xModel += when (direction) {
    "UP" -> distanceX
    "DOWN" -> -distanceX
    else -> 0f
}
```

Assim, é criado um caminho que o utilizador pode seguir para chegar ao seu destino. No entanto ainda era necessário fazer com os modelos aparecessem numa orientação específica. Então, foi utilizado o acelerómetro para obter com exatidão a direção em que a câmera está apontada. Com essa informação, os modelos são renderizados apenas quando o utilizador está virado na direção correta.

```
fun startSensors(context: Context){
    sensorManager = (context).getSystemService(Service.SENSOR_SERVICE) as SensorManager
    sensor = sensorManager!!.getDefaultSensor(Sensor.TYPE_ROTATION_VECTOR)
}
Diogo Martins
fun onResume() {
    sensorManager!!.registerListener( listener: this, sensor, SensorManager.SENSOR_DELAY_UI)
}
Diogo Martins
fun onPause() {
    sensorManager!!.unregisterListener( listener: this, sensor)
}
Diogo Martins
override fun onSensorChanged(event: SensorEvent?) {
    val angle = Math.toDegrees(Math.atan2(event!!.values[1].toDouble(), event.values[0].toDouble())).toFloat()
    _angle.value = angle
    Log.d(TAG,  msg: "angle1: $angle")
}
```

```
startSensors(requireContext())
angle.observe(viewLifecycleOwner) { angle ->
    ang = angle
    homeButton.text = ang.toString()
}

//UP: -55 -> -69
//DOWN: 19 -> 35
//LEFT: -14 -> -27
//RIGHT: 63-> 77
if((ang > orientationL && ang < orientationH) || n > 0 ) {
    if (n < pathSize) {
        addAnchorNode(
            plane.createAnchor(plane.centerPose),
            directions[n]
        )
    } else if (pathSize == n) addAnchorNode(
        plane.createAnchor(plane.centerPose),
        directions[pathSize - 1],
         final: true
    )
}
```

5. Avaliação

Nesta secção, será realizado um teste à aplicação e avaliado o seu desempenho para verificar quão bem ela cumpre os objetivos propostos.

a. Teste

Em primeiro lugar, foi pensado num caso de uso para poder realizar o teste seguindo esse caso: O utilizador encontra-se no piso -2 da universidade, junto das máquinas de café e deseja se deslocar para a 'sala 6'. Então abre a aplicação, pesquisa pela 'sala 6' através da barra de pesquisa e selecionado o seu destino. De seguida o utilizador faz *scan* do código QR daquela zona e percorre o caminho que lhe será apresentado.

Depois, foi realizado noutro caso, com um caminho mais curto: O utilizador encontra-se no piso -2 da universidade, junto das máquinas de café e deseja se deslocar para 0 'anfiteatro 1". Então abre a aplicação, procura o seu destino e seleciona-o. De seguida o utilizador faz *scan* do código QR daquela zona e percorre o caminho que lhe será apresentado.

b. Performance

O utilizador abriu a aplicação, pesquisou e selecionou a sala e fez o scan sem quaisquer problemas. Mas, teve alguns problemas ao tentar seguir o caminho, uma vez que os modelos nem sempre renderizava com a orientação correta. Mas, acabou por conseguir ao repetir o *scan*, mas com distâncias grandes, os modelos deslocavam-se.

Quanto ao segundo teste, o utilizador abriu a aplicação e encontrou logo o destino pretendido, novamente teve problemas com a orientação dos modelos, mas, repetindo o *scan* acabou por conseguir realizar o teste proposto.

6. Discussão

Os objetivos propostos para este projeto foram implementados, embora o processo, no início, tenha apresentado desafios, uma vez que estava a seguir uma *sample* da *Google* sobre *'ARCore'* que estava confusa e complexa para o meu nível de experiência, como engenheiro informático inexperiente que pela primeira vez estava a implementar para Android e com realidade aumentada, perdendo imenso tempo. Então, foi decidido focar, primeiramente, na restante lógica da aplicação e depois voltar à realidade aumentada, possivelmente, com uma abordagem mais simples.

A implementação teve algumas falhas. Por exemplo, como foi notado no teste, a orientação dos modelos nem sempre é a correta e com os modelos a se deslocarem quando a distância é longa, provavelmente devido a um problema na deteção de planos à medida que a distância aumenta. Para além disso, algumas formas de implementação que podem não ter sido as mais corretas, como o facto de o *scan* e a apresentação do caminho estarem em fragmentos diferentes e não em um fluxo único, isto deveu-se ao facto de já ter a implementação da '*CameraX'* e do *scan* de códigos QR feita antes de implementar a realidade aumentada e não foi possível ligar as duas tecnologias, então, decidi esta abordagem para não desperdiçar muito tempo e não refazer o que estava feito.

7. Conclusão

Concluindo, realizar este projeto, proposto pelo engenheiro Bernardo Luís, ao qual agradeço pela oportunidade e pelo suporte contínuo ao longo do semestre, foi uma excelente experiência e oportunidade para expandir os meus conhecimentos nesta vasta área.

O plano proposto era uma aplicação que auxiliasse o utilizador a se navegar num espaço *indoor* e que lhe desse informações sobre a sua posição atual. Como já foi dito, não foi possível dar essas informações com 100% de eficácia. O próximo passo seria continuar a investigar sobre o tema para perceber melhor a razão dos problemas identificados e resolvê-los.

Apesar destas falhas, fiquei satisfeito com este projeto, do ponto de vista do conhecimento ganho ao longo do projeto e sinto que, com o conhecimento obtidos, estou preparado para enfrentar novos desafios e realizar projetos semelhantes com maior confiança e competência, embora reconheça as falhas e necessidade de resolver estas lacunas.

8. Referências

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

[2] https://github.com/SceneView/sceneview-android

[3] https://developer.android.com/training/data-storage/room

[4] https://medium.com/geekculture/getting-started-android-camerax-a84e138e2c00

[5] https://developers.google.com/ml-kit

[6] https://kotlinlang.org/docs/getting-started.html

[7] https://pt.wikipedia.org/wiki/Busca_em_profundidade

[8] https://pt.wikipedia.org/wiki/Realidade_aumentada

[9] https://developers.google.com/ar/develop/samples

[10]https://developer.android.com/develop/sensors-and-location/sensors/sensors_motion

[11] FERMÉ, Eduardo. Inteligência Artificial: Busca Cega [PowerPoint slides]. Slides 24-28.

[12] https://sceneview.github.io/api/sceneview-android/sceneview/