NUEVOS PARADIGMAS DE INTERACCIÓN

NUI para La Alhambra de Granada

La aplicación se ha desarrollado mediante android-studio con el lenguaje de programación kotlin y en la versión de Android 5.0 Se han implementado las clases que explicamos a continuación:

Clase monumento:

La clase monumento nos sirve para implementar una lista ordenada con todos los monumentos, de forma que la aplicación sugiera visitar lugares emblemáticos cuando nos encontremos cerca.

Contiene como parámetros:

- nombre del monumento(String)
- la latitud (Double)
- longitud (Double)
- distancia (Double)

Se inicializa pasándole el nombre, latitud y longitud.

Como métodos:

- fun setdist(d : Double) Simplemente pone como distancia el valor d que se le pasa

Clase menú:

Se ha utilizado la clase menú por defecto de android, la cual se ha modificado para que únicamente aparezcan las pestañas que nos interesan. Pide permisos para utilizar la galería de imágenes del dispositivo y el bluetooth. Las opciones disponibles en el menú son: - Retornar del menú a las gafas - Ver la información de ayuda de la aplicación - Compartir las fotos tomadas

Dentro de las funciones del menú por defecto de android se han añadido nuevos métodos para añadir los botones.

Clase Main:

La funcionalidad principal de la aplicación es la de servir como un guía personal virtual y permite al usuario disfrutar de una experiencia de realidad aumentada. Para ello se han implementado varias funciones que nos permiten utilizar la cámara para reconocer monumentos (en la práctica códigos QR), hacer diversos gestos en la pantalla táctil para una experiencia más interactiva y disponer de GPS y bŕujula para orientarnos y saber como llegar al siguiente monumento.

En el main hemos implementado directamente diversas variables y funciones que se pueden agrupar en tres clases: Gestos, Cámara y GPS/Brújula.

La clase Main utiliza AppCompatActivity(), GestureDetector.OnGestureListener, GestureDetector.OnDoubleTapListener, LocationListener, SensorEventListener.

Las funciones de Main implementadas que no se engloban en gestos, cámara o GPS/brújula son:

- override fun onCreate(savedInstanceState: Bundle?) Inicializa las vistas de la aplicación y llama a las funciones para activar los sensores, la cámara, vibración, la brújula y el GPS implementados.
- override fun onPause() Simplemente para la aplicación haciendo super.onPause() y stopCompass()
- override fun onResume() Simplemente reanuda la aplicación haciendo super.onResume() y startCompass()

GESTOS

Se ha implementado la siguiente serie de gestos: - Tap (un toque): abre el menú - DoubleTap (doble toque): muestra/oculta la brujula - LongPress (mantener pulsado) - OnScroll (pulsar y arrastrar el dedo) - OnFling (pulsar, arrastrar el dedo y levantarlo de la pantalla mientras este aun se esta moviendo) - OnScroll con dos dedos a derecha e izquierda: aceptar o denegar.

Una vez tenemos implementado el onScroll con dos dedos es fácil implementar el Tap con dos dedos y otra serie de gestos, pero como no los hemos usado hemos prescindido de añadirlos. De hecho hay gestos que están implementados pero no le hemos dado por ahora ningún uso, pero podría tener uso en alguna actualización futura.

Para la implementación de estos gestos hemos usado las clases GestureDetector.OnGestureListener y GestureDetector.OnDoubleTapListener, además de hacer override a la función onTouchEvent

Parámetros:

- gestureDetector: para la detección de gestos
- xPosIni, yPosIni: píxeles en los que se inician el gesto con dos dedos
- dosDedos: indica que se viene de una acción con dos dedos

Funciones:

- onTouchEvent: se llama cuando hay un evento touch. En dicha función esta implementado los gestos multitouch.
- onSingleTapConfirmed: abre menú.
- onDoubleTap: indica que se ha tocado dos veces la pantalla (sin uso).
- onDoubleTapEvent: muestra/oculta brújula.
- onDown: indica que se ha pulsado la pantalla (sin uso).
- onShowPress: se ha mantenido cierto tiempo presionado la pantalla (sin uso).
- onSingleTapUp: indica que se ha levantado el dedo de la pantalla (sin uso).
- onScroll: se ha pulsado y acto seguido desplazado el dedo por la pantalla (sin uso).

- onLongPress: se ha pulsado y mantenido por un periódo de tiempo considerable la pantalla (sin uso)
- onFling: cuando realizas un gesto onScroll y dejas de tocar la pantalla estando el puntero en movimiento (sin uso).

CÁMARA

La funcionalidad de la cámara es simular el uso de unas smart glasses mediante el uso de nuestro smartphone, además de tener implementado un detector de barras QR. La idea es que el propio dispositivo reconociera el monumento en sí, pero no hemos podido implementar dicha funcionalidad, por lo que hemos sustituido por un código QR. Otras funcionalidades serían poder sacar fotos con tus smart glasses y que se guarden en tu smartphone mediante bluetooth u otros medios, y también poder hacer otros tipos de acciones como usar el zoom.

Parámetros:

- MY_PERMISSIONS_REQUEST_CAMERA : booleano para comprobar si se tienen permisos de cámara
- token : token actual
- tokenanterior : token anterior, usado para no leer varias veces el mismo token seguidos
- QRdetectado : booleano que indica si se ha detectado un código QR.

Funciones:

- initQR: inicia la cámara y el detector QR.
- surfaceCreated: comprueba si se tiene permisos para abrir la cámara, y si es así abre la cámara.
- surfaceDestroyed: cierra la cámara.
- receiveDetections: indica que hacer cuando se detecta un código QR.

GPS/BRÚJULA

La funcionalidad del GPS y la brújula es saber donde estamos en todo momento de manera que se pueda realizar una visita interactiva, en la cual la aplicación sugiere los monumentos más cercanos y el usuario puede decidir cual desea visitar (en la práctica, la aplicación toma el monumento más cercano como objetivo). Además indica como llegar a dicho lugar mediante una brújula que marca la dirección en que se encuentra el monumento y la distancia que queda por recorrer hasta llegar a él.

Parámetros:

- sensorManager: de tipo SensorManager, para gestionar los sensores del GPS y la brújula.
- accelerometer, magnetometer y rotationVector: sensores que utiliza la brújula.
- haveSensorAccelerometer, haveSensorMagenotemeter y private var haveSensorRotationVector: booleanos para marcar si tenemos dichos sensores.

- lastAccelerometer, lastMagnetometer: último valor obtenido de dichos sensores.
- lastAccelerometerSet, lastMagnetometerSet: booleanos para marcar si hemos obtenido el último valor de los sensores.
- rotationMatrix, orientation y azimuth: respectivamente, el valor de la matriz de rotación de la brújula, la orientación de la brújula y el ángulo que rota la brújula.
- northLatitude, northLongitude, currentLatitude, currentLongitude, targetLatitude, targetLongitude: coordenadas del norte magnético, la posición actual y la posición del lugar al que se quiere llegar.
- angle: ángulo en radianes respecto al norte en el que se encuentra el objetivo. Ejemplo: si angle = 0, entonces el objetivo se encuentra al norte respecto nuestra posición actual; si angle = PI, el objetivo se encuentra al sur.

Funciones de la bŕujula y el GPS:

- private fun setAngle()

  Sirve para saber en que dirección se encuentra el monumento objetivo.

  Actualiza el valor de la variable angle, asignándole el valor del ángulo que forman los vectores u (localizaciónActual_localizaciónObjetivo) y v (localizaciónActual_localizaciónNorte).

  Para ello al vector u se le asignan las coordenadas del objetivo menos las coordenadas actuales, mientras que al vector v se le asignan las coordenadas del norte menos las coordenadas actuales. Luego obtenemos el aṅgulo que forman u y v haciendo arcocoseno(<u,v>/(|u||v|)), para ello utilizamos las funciones de Math. Como el arcocoseno da los valores entre [0, PI], para saber si el ángulo es negativo hacemos la función distancia con signo. Si (<u,n> < 0) entonces el ángulo es negativo, siendo n el giro de 90º antihorario de la variable v. Por último, igualamos la variable angle al ángulo obtenido.

- private fun distance(latitude: Double, longitude: Double): Double

  Devuelva la distancia en metros de la posición actual a la posición cuyas coordenadas se pasan como parámetros.

  La distancia se calcula utilizando la fórmula 'harsevine' distancia = Rc *siendo c = 2*arcotangente(a$^{(1/2),(1-a)}$(1/2)) a = seno(phi/2)*seno(phi/2)+cos(phi1)*cos(phi2)*seno(lambda/2) R = radio de la Tierra en metros (6371000) phi1 = latitud_actual (en radianes) phi2 = latitude (en radianes) phi = phi2 - phi1 lambda = longitude - longitud_actual (en radianes)

- private fun setLocation()

  Actualiza la localización actual.

Primero comprobamos que tenemos los permisos del GPS para detectar la posición, en caso de no tenerlos se piden. A continuación contactamos con el mejor proveedor y le pedimos la última localización conocida. Si obtenemos la localización entonces actualizamos nuestra posición, en caso contrario mostramos un mensaje de error, "localización no disponible". Para obtener la posición utilizamos funciones predefinidas de android.

- override fun onLocationChanged(location: Location?)

  Si la posición detectada por el GPS ha cambiado entonces actualizamos la posición actual mediante setLocation(), obtenemos el monumento más cercano utilizando la función findMonuments que hemos implementado, cambiamos el objetivo a dicho monumento si se ha encontrado y está a menos de 20000 metros. Luego actualizamos el ángulo y la distancia con setAngle() y setDistance().

- override fun onSensorChanged(event: SensorEvent?)

  Si se detecta un cambio en el vector de rotación, en el acelerómetro o en el magnétometro, entonces se actualiza la información de dichos sensores y la brújula gira conforme al cambio detectado, señalando la aguja roja al norte y la aguja amarilla al objetivo. Para detectar los cambios de dichos sensores se utilizan funciones ya definidas en Android. Para que la aguja roja apunte al norte obtenemos la rotación de la brújula mediante el magnétometro y el acelerómetro, para que la amarilla apunte al objetivo le sumamos angle al ángulo de giro de la brújula. A continuación rotamos ambas imágenes modificando su parámetro rotation.

- private fun findMonuments(locX : Double, locY : Double) : Monument

  Devuelve el monumento más cercano a nuestra posición actual siempre que se encuentra a menos de 500 metros. Los monumentos que se tienen en cuenta son los que aparecen en el archivo coordenadas.txt (que se encuentra en app/assets).

  Primero obtiene los monumentos con sus nombres y coordenadas del documento coordenadas.txt y los mete en una lista. A continuación por cada monumento calcula la distancia respecto a la posición actual utilizando la función distance, si dicha distancia es menor de 500 metros lo mete en una lista ordenada según la distancia (de menor a mayor). Por último, devuelve el monumento más cercano, si ningún monumento se encuentra a una distancia menor de 500 metros entonces devuelve un monumento con nombre "No monument", coordenadas por defecto (0,0) y distancia de 1000 kilómetros.

- private fun startCompass()

  Pone a funcionar la brújula tomando los permisos de los sensores magnétometro, acelerómetro y vector de rotación. A continuación se pone a recibir la información de dichos sensores.

- private fun stopCompass()

  Deja de escuchar la información de los sensores para la brújula.

Diagrama de dependencias

Por último , introducimos un diagrama de dependencias que muestra lo descrito anteriormente en un diagrama

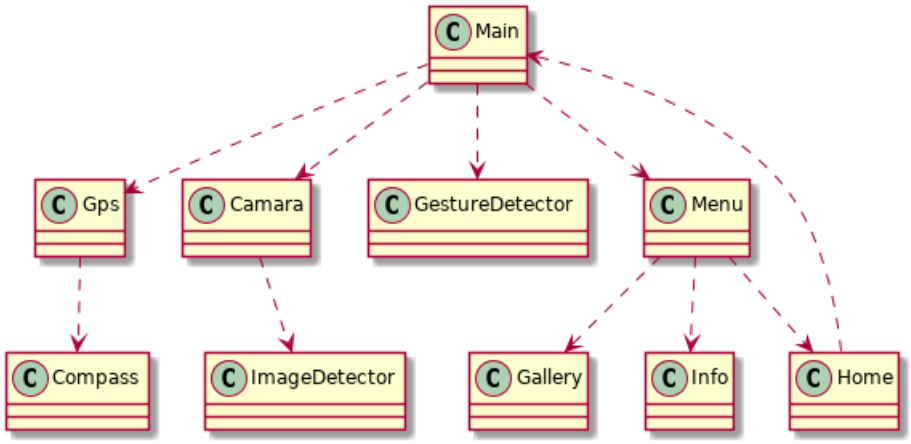

Figure 1: Diagrama