



UNCUYO
UNIVERSIDAD
NACIONAL DE CUYO



**FACULTAD
DE INGENIERÍA**

Proyecto de Realidad Virtual

ASISTENTE DE NAVEGACIÓN EN INTERIORES

Autor: Facundo Arce Vieyra

Legajo: 10546

Profesor: Javier Rosenstein

Mendoza, Argentina

September 2021

Proyecto de Realidad Virtual
ASISTENTE DE NAVEGACIÓN EN INTERIORES

Índice

1. Introducción	2
1.1. Antecedentes	2
1.2. Propuesta	2
2. Descripción de la aplicación	3
2.1. Hardware	3
2.2. Software	4
3. Desarrollo de la aplicación	5
3.1. Configuración	5
3.1.1. Configuración de Unity	5
3.1.2. Configuración de la Raspberry Pi	5
3.2. Seguimiento de movimiento	6
3.3. Calibración	9
3.4. Navegación	11
3.4.1. Navegación en minimapa	11
3.4.2. Navegación en realidad aumentada	12
3.5. Descarga de datos	14
3.5.1. Servidor en Raspberry Pi	14
3.5.2. Cliente en teléfono	16
3.6. Interfaz de usuario	18
4. Resultados	20
5. Conclusiones	21

1. Introducción

1.1. Antecedentes

Aplicaciones basadas en posicionamiento satelital como Google Maps son muy útiles para obtener direcciones de navegación dentro de una ciudad, de un país, o incluso entre países. Pero a menor escala, para navegación dentro de edificios en donde puede haber numerosas habitaciones y pasillos, como un museo, un hospital, una planta industrial o un centro comercial, la tecnología GPS no podría aplicarse debido a la baja precisión, del orden del metro, y a la pobre recepción de señal dentro de edificios.

Existen distintos métodos para solucionar el problema de la navegación en interiores. Entre ellos, se pueden mencionar técnicas de triangulación con la intensidad de señales de Wi-Fi recibidas, estimación con sensores de odometría como acelerómetros, magnetómetros y giróscopos, detección directa de etiquetas de ubicación por infrarrojo, ultrasonido, Bluetooth o sensores ópticos y, por último, técnicas de visión artificial. Muchas veces estos métodos se combinan para lograr mejores resultados.

A nivel comercial, aún no existe alguna solución ampliamente utilizada para la navegación de interiores. Pero existen investigaciones y proyectos particulares que ofrecen diversas soluciones a este problema. Se puede mencionar a Mendez [1] y a Hardy [2] que han propuesto navegación usando métodos de SLAM con cámaras y sensores inerciales aprovechando la tecnología *ARCore* de Google. Sus trabajos sirven de guía para el presente proyecto.

1.2. Propuesta

En este proyecto, se propone realizar una aplicación de realidad aumentada para teléfonos para la asistencia de navegación en interiores.

En ella, el usuario selecciona a dónde quiere dirigirse dentro de un edificio y se le indicará las direcciones a seguir de dos formas. Por un lado, mediante flechas en un entorno de realidad mixta. Por otro lado, por medio de un minimapa virtual que le indicará la posición actual y el camino a seguir.

2. Descripción de la aplicación

2.1. Hardware

El hardware utilizado consiste en un teléfono inteligente de la marca Samsung modelo S9+ y una Raspberry Pi 3 modelo B.

Como se detallará más adelante, la cámara y los sensores inerciales del teléfono son utilizados para determinar su posición y orientación a medida que éste se mueve en el espacio. Al ser un sistema de localización relativo, surgen dos problemas, por un lado, la posición y orientación inicial son desconocidas y, por otro lado, los errores son acumulativos, por lo que el error entre la localización estimada y la real crecerá con el tiempo. Ambos problemas se resuelven con el uso de la cámara del teléfono que escanea códigos QR situados en distintos puntos del lugar a navegar y que sirven para determinar el estado inicial y para recalibrar la navegación.

La pantalla táctil del teléfono actúa tanto como salida y entrada de la interfaz de usuario. En ella se muestra la información de navegación y el usuario también puede interactuar con diversos elementos gráficos como menus, botones, listas y casillas de verificación para controlar y configurar la aplicación.

Al ser interiores de edificios los sitios a navegar, se propone que el mapa no se encuentre en el teléfono, sino en otro dispositivo dentro del edificio. De esta manera, cuando el usuario ingresa, descarga la información de navegación correspondiente. Para esto, el dispositivo seleccionado es un ordenador de placa única Raspberry Pi 3. Éste cuenta con conectividad Bluetooth incorporada que se utiliza para establecer una comunicación con el teléfono. La Raspberry Pi actúa como servidor y contiene los datos de navegación y el celular actúa como cliente y recibe y decodifica esta información.

La figura 1 ilustra la arquitectura de la aplicación explicada.

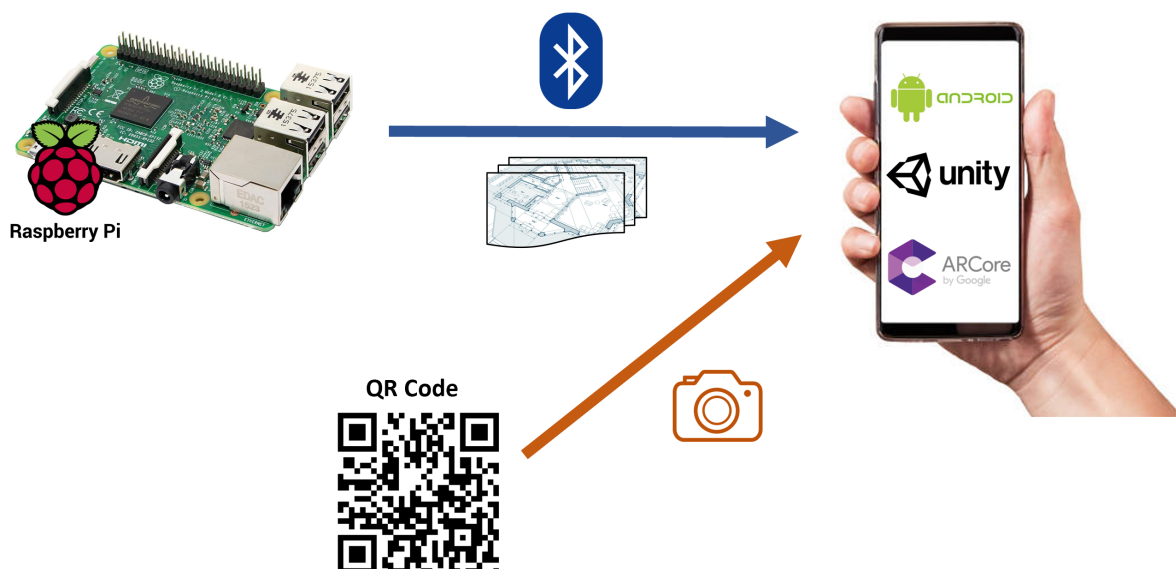


Figura 1: Arquitectura de la aplicación desarrollada.

2.2. Software

El software de la aplicación se divide en dos, por un lado el del teléfono inteligente, por otro lado, el de la Raspberry Pi.

Para el teléfono, se desarrolló una aplicación para Android en Unity. Unity es un motor de videojuego multiplataformas disponible como plataforma de desarrollo para Windows, Linux y Mac OS. Para este proyecto, se utilizó la versión de Unity 2019.4.22f1 en Windows 10.

ARCore es una plataforma de Google para el desarrollo de aplicaciones de realidad aumentada. Hay tres funciones principales que ofrece ARCore: seguimiento de movimiento, comprensión del entorno (por ejemplo detección de superficies) y estimación de la iluminación. De estas tres funciones, para esta aplicación se utiliza la de seguimiento de movimiento. Ésta se basa en la identificación y seguimiento en el tiempo de puntos interesantes capturados por la cámara del teléfono en combinación con las lecturas de los sensores inerciales del teléfono como acelerómetros y giróscopos. El seguimiento de movimiento no sólo se utiliza para la navegación, sino también para poder colocar objetos, como flechas y pines, integrados en el mundo real de manera tal que, aunque el usuario se mueva y enfoque la cámara en otra dirección, el objeto virtual permanece fijo en su posición en el mundo real. Se utiliza para esta aplicación el kit de desarrollo de ARCore para Unity en su versión 1.23.0.

Para la comunicación cliente-servidor con la Raspberry Pi, se utiliza el plugin de Unity *Android & Microcontrollers / Bluetooth* de TechTweaking que permite controlar fácilmente las capacidades Bluetooth del teléfono.

Navmesh es un conjunto de componentes que permiten la navegación en un ambiente tridimensional en Unity usando algoritmos de inteligencia artificial y en este proyecto se usa para encontrar el camino entre la ubicación actual y la ubicación deseada.

El escaneo de los códigos QR para el reposicionamiento del usuario se logra con la librería ZXing para Unity.

A la Raspberry Pi se le instala el sistema operativo Raspberry Pi OS basado en Debian. Para configurarla como servidor, se ejecuta un script de Python con el módulo *PyBluez* que permite acceder a recursos Bluetooth de la máquina. La comunicación se realiza con protocolo RFCOMM ("Radio Frequency Communication") que permite emular la comunicación a través de puertos serie RS-232. Este protocolo es ampliamente utilizado por su simplicidad y porque permite adaptar aplicaciones que usan puertos serie fácilmente.

3. Desarrollo de la aplicación

3.1. Configuración

3.1.1. Configuración de Unity

Para el correcto funcionamiento del entorno de desarrollo de la aplicación del presente proyecto, se deberá seguir las siguientes instrucciones para instalar y configurar Unity y los paquetes y plugins necesarios.

Descargar e instalar Unity Hub. Al iniciar Unity Hub, seleccionar la versión de Unity 2019.4.22f para descargar e instalar. Durante la instalación, incluir el módulo *Android Build Support* y, dentro de éste, asegurarse de incluir *Android SDK & NDK tools* y *OpenJDK*. Aunque el desarrollo es para Android, incluir también el módulo *iOS Build Support* para evitar errores en ARCore.

Descargar el paquete de ARCore SDK para Unity v1.23.0. Seguir la guía *Quickstart for Android* para instalarlo y configurarlo. En resumen, descargar e instalar Android Studio con Android SDK 7.0 o superior. Crear un nuevo proyecto 3D en Unity. Ir a *Windows > Package Manager* y descargar los paquetes *Multiplayer HLAPI* y *XR Legacy Input Helpers*. Ir a *Assets > Import Package > Custom Package* y seleccionar el archivo *.unitypackage* descargado previamente para importar el paquete ARCore. Abrir el proyecto de muestra *HelloAR* en *Assets/GoogleARCore/Examples/HelloAR/Scenes/*. Ir a *File > Build Settings* y seleccionar Android como plataforma, luego hacer click en *Player Settings*. Allí, configurar, dentro de *Other Settings*, *Minimum API Level* en 24, *Scripting Backend* en IL2CPP, *Target Architectures* en ARM64 y, dentro de *XR Settings*, habilitar la opción *ARCore Supported*.

Conectar el teléfono a la computadora y, estando en *Build Settings*, hacer click en *Build and Run*. Se genera un archivo *.apk* que se instala y ejecuta en el teléfono. Luego de comprobar que la aplicación de prueba *HelloAR* funciona correctamente, se la puede utilizar de plantilla para el desarrollo de la presente aplicación.

Descargar e instalar el plugin *Android & Microcontrollers / Bluetooth* desde la *Unity Asset Store*. Este plugin necesita modificar el archivo *AndroidManifest.xml* del proyecto, para eso, primero se lo debe generar en *File > Build Settings > Player Settings > Publishing Settings* habilitando la opción *Custom Main Manifest*. Luego, ir a *Tools > TechTweaking > Bluetooth Classic > Setup the BT Library* para iniciar la configuración automática del plugin.

Descargar la librería *ZXing* y colocar el archivo *zxing.unity.dll* en la carpeta */Plugins* del proyecto.

3.1.2. Configuración de la Raspberry Pi

Lo primero que se debe realizar es instalar un sistema operativo en la Raspberry Pi. Siguiendo la guía *Setting up your Raspberry Pi* [3], descargar y ejecutar en la computadora *Raspberry Pi Imager*. Conectar una tarjeta microSD de al menos 8GB a la computadora. En *Raspberry Pi Imager*, seleccionar la última versión de Raspberry Pi OS y la microSD en la que queremos instalar el sistema operativo y esperar a que finalice la descarga e instalación.

Conectar a la Raspberry Pi la tarjeta microSD, un teclado y un mouse a los puertos USB, un monitor al puerto HDMI y una fuente de alimentación de 5V 2,5A al puerto micro-USB. Es posible

también controlar la Raspberry desde la computadora por Wi-Fi realizando una configuración *headless* sin necesidad de teclado, mouse y monitor. Para ello, seguir la siguiente guía [4].

Cualquier teléfono debe ser capaz de descubrir y conectarse a la Raspberry Pi a través de Bluetooth para descargar los datos de navegación en cualquier momento. El tiempo en el que el dispositivo puede ser encontrado es, por defecto, tres minutos. Se configura este tiempo de expiración a cero para que siempre pueda encontrarse [5].

```
1 $ sudo nano /etc/bluetooth/main.conf
2 DiscoverableTimeout = 0
```

Se configura la Raspberry Pi para que al iniciarse, encienda el módulo Bluetooth y permita que otros dispositivos puedan encontrarla y conectarse.

```
1 $ sudo nano /etc/rc.local
2 sudo bluetoothctl <<EOF
3 power on
4 discoverable on
5 pairable on
6 EOF
```

Instalar Python 3 y PyBluez ejecutando las siguientes líneas en la terminal.

```
1 $ sudo apt update
2 $ sudo apt install python3 idle3
3 $ sudo apt-get install libbluetooth-dev python-dev libglib2.0-dev libboost-python-dev libboost-
  thread-dev
4 $ pip3 install pybluez
```

3.2. Seguimiento de movimiento

La plantilla *HelloAR* [6] contiene los objetos necesarios para el funcionamiento de las funciones de ARCore. *ARCore Device* contiene configuraciones de la sesión de realidad aumentada, ver figura 2. Se deshabilita la detección de planos ya que no se utilizará y se selecciona el modo de estimación de luz como intensidad ambiente para aplicar a los objetos de realidad aumentada, otras opciones demandan más recursos computacionales. *First Person Camera* está vinculada a la cámara trasera del teléfono y además de parámetros de cámara, también tiene asociadas configuraciones del seguimiento de movimiento, ver figura 3. Principalmente, se puede seleccionar seguimiento de posición solamente o posición y rotación. Seleccionar esta última opción, ya que si bien el seguimiento de rotación tiene un impacto negativo en la precisión del seguimiento de posición, es necesario para la integración de elementos de realidad aumentada.

Se crea el mapa del lugar a navegar en SolidWorks para obtener un archivo de geometría 3D a escala *.obj* como se ve en la figura 4. Este archivo se coloca en la carpeta */Prefabs* del proyecto y se inserta en la escena del proyecto. Asegurarse de que el mapa esté a escala 1:1, es decir que un metro en la vida real tiene que equivaler a 1 unidad de Unity. Esto puede lograrse, como se ve en la figura 5, con los assets de

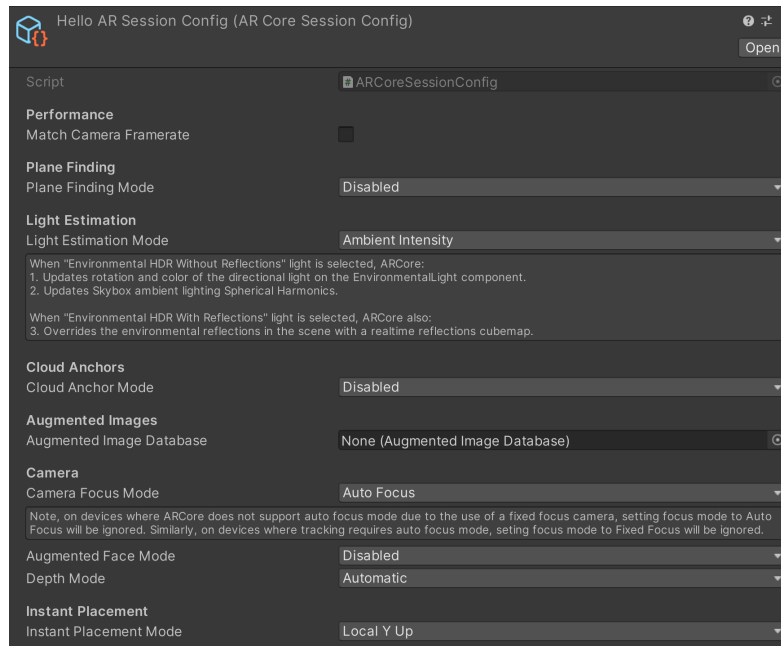


Figura 2: Configuración de la sesión de realidad aumentada.

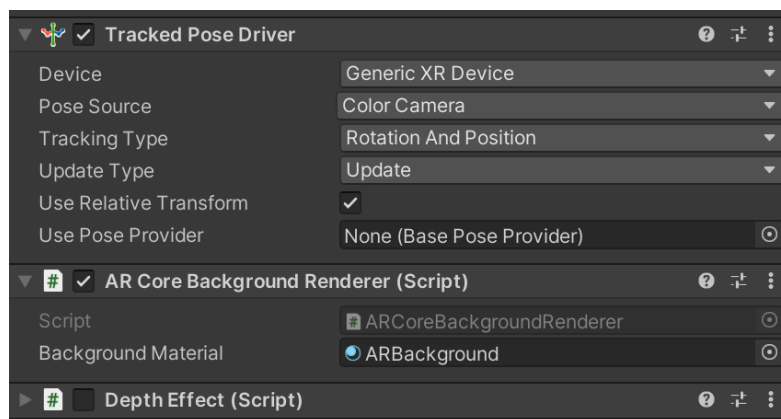


Figura 3: Configuración de la cámara en primera persona.

regla horizontal y vertical de *zzz Ruler* que pueden descargarse desde la *Unity Asset Store*

Para representar el usuario, referido como *Player*, se inserta un cilindro en la escena. Se añade también un *prefab* de una flecha encima del cilindro para indicar la orientación del usuario y un cubo invisible por delante del cilindro para utilizarlo en cálculos auxiliares que se explicarán más adelante. Se crea una cámara como hijo del cilindro que apunta hacia éste desde arriba para capturar su movimiento en el mapa. Se crea una *Render Texture* en la carpeta */Textures* del proyecto. En la propiedad *Target Texture* de la cámara mencionada, se selecciona la textura creada para poder usarla luego en un minimapa mostrado en la interfaz gráfica de la aplicación.

Se crea un script y se lo asocia al objeto *Player Controller* para controlar el movimiento del cilindro para que corresponda al movimiento real del usuario. El listado 1 muestra el código implementado. Se

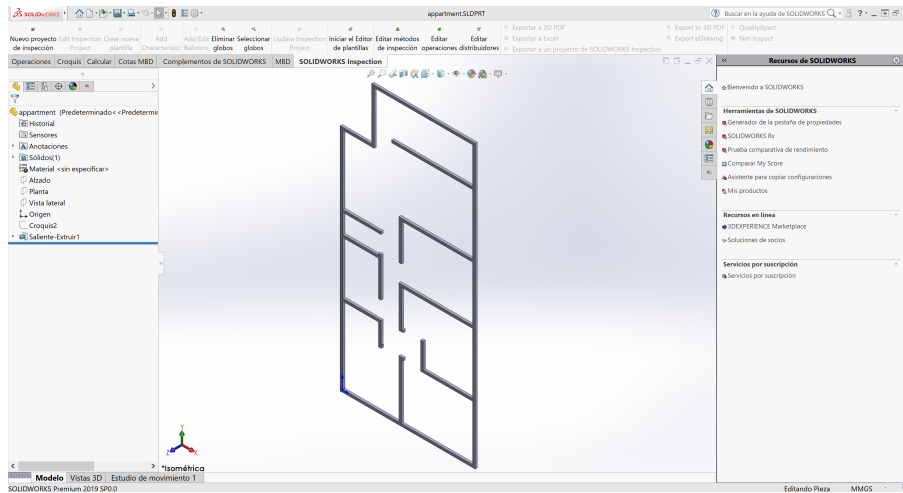


Figura 4: Mapa tridimensional del mapa del sitio a navegar hecho en SolidWorks.



Figura 5: Uso de *zzz Ruler* para asegurar la correcta escala del mapa en Unity.

puede apreciar que *Frame.Pose* contiene la posición y orientación dadas por ARCore relativas al punto inicial. Estos datos relativos se traducen a una posición y orientación absoluta en el mapa gracias a la calibración por códigos QR.

Listado 1: Código para el seguimiento de movimiento

```
1 // Current position in real-world
2 Vector3 currPos = Frame.Pose.position;
3 float currHeading = qrScanner.deltaOrientation + Frame.Pose.rotation.eulerAngles.y;
4
5 // Difference between positions
6 Vector3 deltaPos = -(currPos - prevPos);
7 float deltaHeading = - Mathf.DeltaAngle( prevHeading, currHeading );
8
9 // Update previous position
```

```
10 prevPos = currPos;
11 prevHeading = currHeading;
12
13 // Apply transform to Player (ignoring differences in height)
14 float x = deltaPos.x * Mathf.Cos( qrScanner.deltaOrientation * Mathf.Deg2Rad ) + deltaPos.z *
    Mathf.Sin( qrScanner.deltaOrientation * Mathf.Deg2Rad );
15 float z = deltaPos.z * Mathf.Cos( qrScanner.deltaOrientation * Mathf.Deg2Rad ) - deltaPos.x *
    Mathf.Sin( qrScanner.deltaOrientation * Mathf.Deg2Rad );
16 player.transform.Translate( x, 0.0f, z );
17
18 // Apply rotation to Arrow indicator and auxiliar Cube
19 arrow.transform.localRotation = Quaternion.Euler( new Vector3( 0, currHeading, 0 ) );
20 auxiliar.transform.RotateAround( player.transform.position, new Vector3( 0, 0, 1 ),
    deltaHeading );
```

3.3. Calibración

Al iniciar la aplicación, antes de poder comenzar a navegar, el usuario debe escanear un código QR correspondiente a un punto de referencia del mapa. Estos códigos contienen simplemente el nombre del punto de referencia y son creados y colocados en el sitio a navegar con antelación.

Los puntos de referencia son objetos vacíos en Unity cuyo nombre se usa para identificarlos, las componentes x y z de su posición reflejan la ubicación en el mapa y la rotación alrededor del eje y , la orientación.

El objeto *QR Scanner* contiene el script que maneja la calibración. Como muestra el listado 2, en cada fotograma, el lector de ZXing busca un código QR en la imagen de la cámara, cuando encuentra uno, se analiza el contenido del código y, si coincide con un punto de referencia, se procede con la reubicación.

Listado 2: Código para la lectura de códigos QR

```
1 /// Capture and scan the current frame
2 void Scan() {
3     System.Action<byte[], int, int> callback = (bytes, width, height) => {
4         if (bytes == null) {
5             // No image is available.
6             return;
7         }
8         // Decode the image using ZXing parser
9         try {
10             IBarcodeReader barcodeReader = new BarcodeReader();
11             var result = barcodeReader.Decode(bytes, width, height, RGBLuminanceSource.BitmapFormat.
                Gray8);
12             string resultText = result.Text;
13             // Relocate player
14             if (first) {
15                 Relocate(resultText);
```

```
16         first = false; // avoids relocating on every Update() to the same calibration point
17     }
18 }
19 catch (NullReferenceException) { //throws NullReferenceException if no QR was found
20     first = true;                // allows relocating next time it finds a qr code
21 }
22 };
23 CaptureScreenAsync(callback);
24 }
```

La reubicación, mostrada en listado 3, consiste en trasladar el cilindro que representa al usuario hasta la posición del punto de referencia y en calcular la diferencia en la orientación que luego se ajustará en el script de seguimiento de navegación como pudo verse en listado 1. Cabe resaltar que no se utilizó la siguiente línea

```
player.transform.position = child.position;
```

porque el componente *NavMeshAgent* que posee el usuario no permite el movimiento cuando entre la posición anterior y actual hay una pared. Se utiliza en cambio la función *Warp()* de dicho componente *NavMeshAgent*.

Listado 3: Código para la reubicación del usuario en el mapa

```
1 // Move the player indicator to the calibration point
2 private void Relocate(string text) {
3     text = text.Trim(); //remove spaces
4
5     //Find the correct calibration point scanned and move the player to that position
6     foreach (Transform child in calibrationPoints.transform) {
7         if ( child.name == text ) {
8             isRelocating = true;
9
10            if (!initialized) {
11                mapUI.SetActive(true); // show map UI
12                initialized = true;
13                surface.BuildNavMesh();
14            }
15
16            player.GetComponent<NavMeshAgent>().Warp( child.position );
17            deltaOrientation = Mathf.DeltaAngle( Frame.Pose.rotation.eulerAngles.y, child.
18                localRotation.eulerAngles.y );
19            isRelocating = false;
20        }
21    }
```

3.4. Navegación

La interfaz de usuario contiene un objeto *Dropdown*, o menú desplegable, en el que se enumeran los puntos de referencia. Cuando el usuario selecciona uno de estos puntos, el *Listener* del evento *onValueChanged* del menú se activa y llama a la función *SetTargetDestination()*, ver listado 4, que establece la posición objetivo y activa y posiciona los colisionadores para poder ubicar la flecha y el pin en realidad aumentada.

Listado 4: Código que se ejecuta al seleccionar un punto de referencia del menú desplegable

```
1 private void SetTargetDestination (int index) {  
2     targetDestination = destinations[ index ];  
3  
4     arrowCollider.SetActive( true );  
5     arrowCollider.transform.position = player.position;  
6     pinCollider.SetActive( true );  
7     pinCollider.transform.position = targetDestination.position;  
8 }
```

3.4.1. Navegación en minimapa

Para implementar esta funcionalidad se siguió el siguiente tutorial [7] que explica cómo utilizar los distintos componentes y funciones de NavMesh.

Una vez seleccionada la ubicación de destino, el listado 5 muestra cómo se calcula constantemente el camino entre la posición actual del usuario y la posición deseada teniendo en cuenta los obstáculos en el mapa con la función *CalculatePath()* de NavMesh y cómo se representa dicho camino en el minimapa utilizando el componente *LineRenderer*. Este código es parte del script del objeto *Path Finder*.

Listado 5: Código para el cálculo del camino y su representación gráfica

```
1 void Start() {  
2     path = new NavMeshPath();  
3     line = GetComponent<LineRenderer>(); // get LineRenderer component on current object  
4     line.enabled = false;  
5     // ....  
6 }  
7  
8 void Update() {  
9     if (targetDestination != null) {  
10         NavMesh.CalculatePath(player.position, targetDestination.position, NavMesh.AllAreas, path);  
11  
12         line.positionCount = path.corners.Length;  
13         line.SetPositions( path.corners );  
14         line.enabled = true;  
15     }  
16 }
```

Se agrega un objeto con el componente *NavMeshSurface*. La figura 6 muestra las configuraciones realizadas. Al hacer click en *Bake*, se genera la superficie navegable.

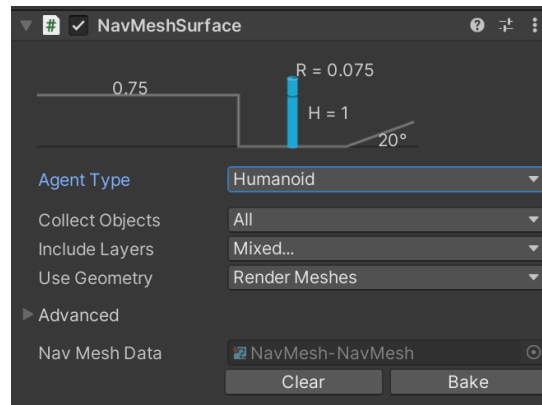


Figura 6: Configuraciones de *NavMeshSurface*.

Para que NavMesh reconozca las paredes como obstáculos a evitar, es necesario agregar el componente *NavMeshModifier* al mapa. En sus opciones, activar *Override Area* y seleccionar *Not Walkable* en *Area Type*. De esta manera, Navmesh sabrá que el agente no puede caminar por las paredes. Después de esto, es necesario hacer click en *Bake* nuevamente para que se reflejen los cambios.

Al objeto *Player* se le añade un componente *NavMeshAgent*. Por defecto, este componente se desactiva y el usuario puede moverse incluso fuera de las zonas caminables de NavMesh. Esto quiere decir que el usuario podría “atravesar paredes” por ejemplo. Esto es útil porque el seguimiento de navegación presenta, como se explicó anteriormente, errores acumulativos y, en lugares como pasillos angostos, el usuario podría “quedarse atrapado” en habitaciones en las que realmente no se encuentra. Si, aún así, se quiere que el usuario no pueda atravesar paredes, se activa entonces este componente. La opción para activar o desactivar el colisionado con paredes está incorporado en el menú de configuraciones de la interfaz de usuario.

3.4.2. Navegación en realidad aumentada

Además de las indicaciones en el minimapa, se agregan indicaciones en realidad aumentada. Se coloca una flecha en frente del usuario indicando la dirección a seguir durante la navegación y cuando llega a destino, se coloca un pin. Estos objetos se fijan en un punto del entorno físico de manera tal que, al moverse el usuario, los objetos quedan anclados a ese punto. De esta manera, el efecto de realidad aumentada es más inmersivo.

Para ello, primero se crean dos objetos con colisionadores del tipo *Capsule Collider*, uno para la flecha y otro para el pin. Al colisionador de la flecha se lo fija en la posición del usuario en un instante dado. Cuando la colisión entre el usuario y este colisionador se activa, se coloca frente al usuario una flecha. El dirección de la flecha se calcula como el ángulo entre la dirección en la que el usuario está mirando (representado por la recta que une el usuario y el cubo auxiliar) y la dirección en la que se encuentra el próximo punto del camino calculado (recta que une el usuario y el segundo punto del objeto *LineRenderer*, siendo el primer punto el correspondiente a la posición del usuario).

El script *ArrowPlacer*, mostrado en parte en el listado 6, se lo añade al objeto usuario, *Player*, ya que las colisiones que se buscan son entre *Player* y el colisionador de la flecha y *Player* y el colisionador del pin y las funciones utilizadas *OnTriggerEnter(Collider)* y *OnTriggerExit(Collider)* son llamadas cuando el objeto que contiene el script entra o sale de una colisión y tienen como argumento al colisionador con el que dicho objeto interactuó.

Cada vez que el usuario sale del volumen del colisionador de la flecha, la flecha actual se elimina y se crea una nueva enfrente del usuario hasta que éste entre en contacto con el colisionador del pin ubicado en la posición deseada. La flecha se elimina y se coloca el pin. Si el usuario sale del volumen del colisionador del pin, entrará nuevamente en el colisionador de la flecha.

Listado 6: Código para el manejo de elementos de realidad aumentada

```
1 private void OnTriggerEnter( Collider collider ) {
2     if ( collider.tag.Equals( "Arrow" ) && !hasEnteredPinCollider ) {
3         if ( line.positionCount > 0 ) {
4             // Position arrow a bit before the camera and a bit up
5             Vector3 pos = arcoreCamera.transform.position + arcoreCamera.transform.forward * 2.0f +
                        arcoreCamera.transform.up * 0.5f;
6             // Rotate arrow to neutral orientation
7             Quaternion rot = arcoreCamera.transform.rotation * Quaternion.Euler( 45, 180, 0 );
8             // Create new anchor
9             anchorArrow = Session.CreateAnchor( new Pose( pos, rot ) );
10            // Spawn arrow
11            spawnedArrow = GameObject.Instantiate( arrowPrefab, anchorArrow.transform.position,
                        anchorArrow.transform.rotation, anchorArrow.transform );
12            // Calculate arrow angle
13            Vector2 currPos = new Vector2( this.transform.position.x, this.transform.position.z );
14            Vector3 pathNodeAux = line.GetPosition( 1 );
15            Vector2 pathNode = new Vector2( pathNodeAux.x, pathNodeAux.y );
16            Vector2 auxNode = new Vector2( auxPos.position.x, auxPos.position.y );
17            float angle = Mathf.Rad2Deg * ( Mathf.Atan2( auxNode.y - currPos.y, auxNode.x - currPos.x )
                        - Mathf.Atan2( pathNode.y - currPos.y, pathNode.x - currPos.x ) );
18            // Apply calculated angle
19            spawnedArrow.transform.Rotate( 0, angle, 0, Space.Self );
20        }
21    }
22    else if ( collider.tag.Equals( "Pin" ) ) {
23        hasEnteredPinCollider = true;
24        // Position pin a bit before the camera and a bit up
25        Vector3 pos = arcoreCamera.transform.position + arcoreCamera.transform.forward * 3.0f +
                    arcoreCamera.transform.up * 0.5f;
26        // Create new anchor
27        anchorPin = Session.CreateAnchor( new Pose( pos, Quaternion.Euler( 0, 0, 0 ) ) );
28        // Spawn pin
29        spawnedPin = GameObject.Instantiate( pinPrefab, anchorPin.transform.position, anchorPin.
```

```
        transform.rotation, anchorPin.transform );  
30     // Destroy arrow  
31     Destroy( spawnedArrow );  
32     Destroy( anchorArrow );  
33 }  
34 }  
35  
36 private void OnTriggerExit( Collider collider ) {  
37     if ( collider.tag.Equals( "Pin" ) ) {  
38         Destroy( spawnedPin );  
39         Destroy( anchorPin );  
40         hasEnteredPinCollider = false;  
41     }  
42     else if ( collider.tag.Equals( "Arrow" ) ) {  
43         arrowCollider.transform.position = this.transform.position;  
44         Destroy( spawnedArrow );  
45         Destroy( anchorArrow );  
46     }  
47 }
```

3.5. Descarga de datos

3.5.1. Servidor en Raspberry Pi

Para el script del server, se usa como plantilla el ejemplo *rfcomm-server.py* de PyBluez [8]. El script modificado para adaptarla a esta aplicación se puede ver en listado 7. Puede observarse en el script que, para crear la comunicación RFCOMM, es necesario indicar un UUID (*Universally Unique Identifier*) que es un número de 128 bits que identifica un servicio particular provisto por un dispositivo Bluetooth. Para esta comunicación se usa el servicio *Generic Attribute Profile* (GATT) que provee las bases para la transferencia bidireccional de pequeños fragmentos de datos para aplicaciones de bajo consumo. Para encontrar el UUID del servicio GATT de la Raspberry Pi, se ejecuta el siguiente comando en consola y se busca la línea UUID: **Generic Attribute Profile (#UUID)**.

```
1 $ bluetoothctl  
2 # show
```

Se añade un bucle para reintentar la conexión cuando esta falla. Una vez establecida la conexión, se espera a que el cliente comience la comunicación. Cuando se recibe un “*Hello*”, se envía al cliente la lista de puntos de referencia codificados como [<<nombre>>;<<posición en x>>;<<posición en z>>;<<rotación en y>>]. Al final del mensaje se envía “*Done*” para indicarle al cliente que ha finalizado la comunicación.

Se guarda el script, se lo hace ejecutable

```
1 $ chmod +x bt_server.py
```

y se configura para que se ejecute automáticamente en el arranque del sistema.

Listado 7: Código para el servidor en la comunicación Bluetooth

```
1 import bluetooth
2
3 server_sock = bluetooth.BluetoothSocket(bluetooth.RFCOMM)
4 server_sock.bind(("", bluetooth.PORT_ANY))
5 server_sock.listen(1)
6
7 port = server_sock.getsockname()[1]
8 uuid = "00001801-0000-1000-8000-00805f9b34fb"
9 bluetooth.advertise_service(server_sock, "SampleServer", service_id=uuid,
10                             service_classes=[uuid, bluetooth.SERIAL_PORT_CLASS],
11                             profiles=[bluetooth.SERIAL_PORT_PROFILE],
12                             )
13 while True:
14     print("Waiting for connection on RFCOMM channel", port)
15
16     client_sock, client_info = server_sock.accept()
17     print("Accepted connection from", client_info)
18
19     try:
20         while True:
21             data = client_sock.recv(1024).decode('utf-8')
22             if not data:
23                 break
24             elif "Hello" in data:
25                 client_sock.send("Ready\n".encode('utf-8'))
26                 client_sock.send("[Salida;-1.29;-10.03;0]\n".encode('utf-8'))
27                 # ....
28                 client_sock.send("[Baño 2;-0.34;-4.56;-90]\n".encode('utf-8'))
29                 client_sock.send("Done\n".encode('utf-8'))
30             print("Received: ", data)
31
32     except OSError:
33         pass
34
35     print("Disconnected.")
36     client_sock.close()
37     print("Trying to reconnect.")
38
39 server_sock.close()
```


3.5.2. Cliente en teléfono

El objeto *Map Download Menu* contiene un script que se encarga de la lógica detrás de los eventos vinculados con los elementos gráficos del menú para la descarga de los datos de navegación así como también de implementar las funciones para el cliente de la comunicación Bluetooth. El listado 8 muestra las funciones para la lectura y el envío de mensajes con el servidor usando como base el ejemplo *Bluetooth Terminal* del plugin *Android & Microcontrollers / Bluetooth*. Puede verse que los datos recibidos se almacenan en `_processText.text` tanto para mostrar en la interfaz de usuario, como para procesar los datos recibidos internamente.

Listado 8: Código para el cliente en la comunicación Bluetooth

```
1  /// Send a message to bluetooth device
2  void send( string msg ) {
3      if ( device != null && !string.IsNullOrEmpty( msg ) ) {
4          device.send( System.Text.Encoding.ASCII.GetBytes( msg ) );
5      }
6  }
7
8  // Read message from bluetooth device
9  IEnumerator ManageConnection( BluetoothDevice device ) {
10     while ( device.IsReading ) {
11         byte[] msg = device.read();
12         if ( msg != null ) {
13             // Convert byte array to string.
14             string content = System.Text.Encoding.UTF8.GetString( msg );
15             // Split the string into lines. '\n','\r' are character used to represent new line.
16             string[] lines = content.Split( new char[] { '\n', '\r' } );
17             // Add those lines to the processText
18             _processText.text += content;
19         }
20         yield return null;
21     }
22 }
```

El listado 9 muestra cómo se establece la comunicación cliente-servidor y cómo se tratan los datos recibidos para incorporarlos al mapeado. Como se explicó anteriormente, luego de establecer la conexión, el cliente envía el mensaje “Hello” y espera hasta recibir “Done”. Se eliminan los puntos de referencia existentes en caso de que ya se hayan descargados otros datos previamente. Se decodifica el mensaje usando *Regex*, expresiones regulares, para comprobar si una cadena de caracteres se corresponde con un patrón. De esta manera, se identifican los puntos de referencia individualmente separados por paréntesis y, para cada uno, el nombre y coordenadas separados por puntos y comas. Luego, se crea un objeto para cada punto de referencia como hijo de un objeto *parent* que los agrupa a todos y se modifican sus propiedades con los datos decodificados. Se crean también etiquetas para mostrarse en el punto correspondiente del minimapa con los nombres de los puntos de referencia utilizando el paquete *TextMeshPro* incluido en

Unity que permite manejar texto con una gran flexibilidad.

Listado 9: Código para la decodificación de datos recibidos y añadido de los puntos de referencia

```
1 IEnumerator getCalibrationPoints() {
2     // Start communication with server and wait until response
3     yield return new WaitForSeconds( 3.0f );
4     do {
5         send( "Hello" );
6         yield return new WaitForSeconds( 2.0f );
7     } while ( !_processText.text.Contains( "Done" ) );
8
9     // Clean existing calibration points
10    List<GameObject> children = new List<GameObject>();
11    foreach ( Transform child in parent ) children.Add( child.gameObject );
12    children.ForEach( child => Destroy( child ) );
13
14    // Message parsing
15    string pattern1 = @"\[([^\[\]]+)\]"; // Separate calibration points from rest of message
16    string pattern2 = @"^[,;[\]\n\r]+" ; // Split each calibration point into name and coordinates
17    foreach ( Match m1 in Regex.Matches( _processText.text, pattern1 ) ) {
18        Match mName = Regex.Match( m1.Groups[ 0 ].Value, pattern2 );
19        Match mX = mName.NextMatch();
20        Match mZ = mX.NextMatch();
21        Match mRY = mZ.NextMatch();
22
23        // Check if point is repeated
24        bool isNameRepeated = false;
25        foreach ( Transform child in parent ) {
26            isNameRepeated |= child.gameObject.name == mName.Groups[ 0 ].Value;
27        }
28        if ( !isNameRepeated ) {
29            // Add calibration points
30            GameObject calibPoint = new GameObject( mName.Groups[ 0 ].Value );
31            calibPoint.transform.parent = parent;
32            calibPoint.transform.name = mName.Groups[ 0 ].Value;
33            calibPoint.transform.localPosition = new Vector3( float.Parse( mX.Groups[ 0 ].Value ), 0.0f, float.Parse( mZ.Groups[ 0 ].Value ) );
34            calibPoint.transform.localEulerAngles = new Vector3( 0.0f, float.Parse( mRY.Groups[ 0 ].Value ), 0.0f );
35
36            // Add labels in minimap
37            GameObject label = GameObject.Instantiate( labelPrefab, labelParent );
38            label.transform.localPosition += calibPoint.transform.localPosition;
39            label.GetComponent<TextMeshPro>().SetText( calibPoint.transform.name );
40        }
41    }
42    pathFinder.Clear(); // populate dropdown with calibration points
```

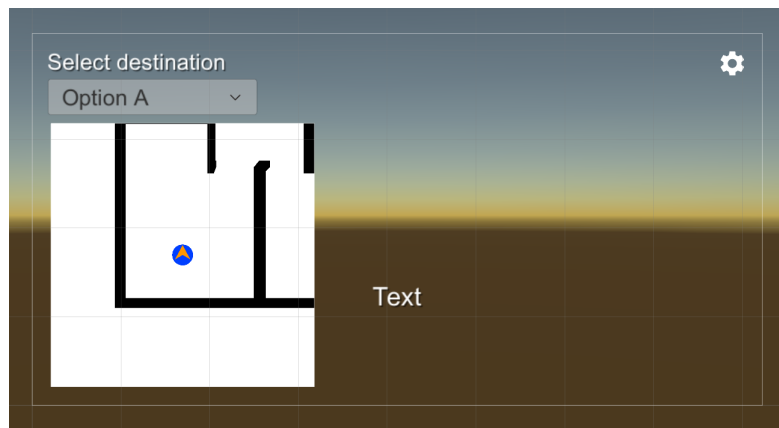
```
42 | message.text = "Please scan QR code to start";  
43 | }
```

3.6. Interfaz de usuario

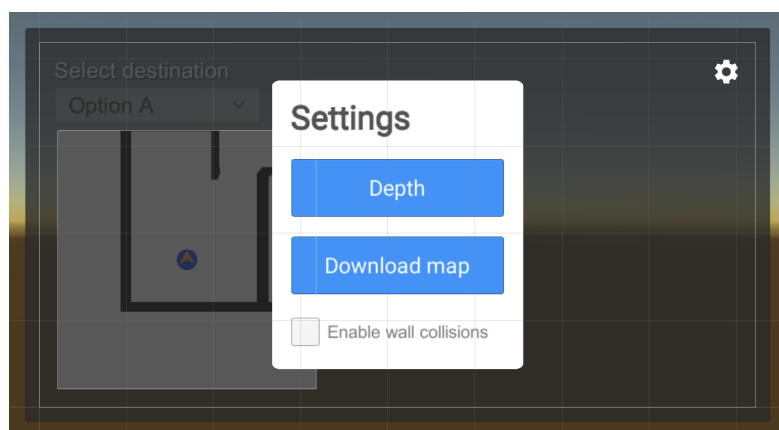
Para la interfaz de usuario que se muestra en la pantalla del teléfono, se crea un objeto *Canvas* en donde se colocan todos los objetos gráficos. En la interfaz principal, figura 7a, hay un menú desplegable en la esquina superior izquierda con los puntos de referencia para elegir el destino, el minimapa en la esquina inferior izquierda, un botón para el menú de configuraciones en la esquina superior derecha y, en el centro, una etiqueta de texto para mostrar mensajes al usuario.

El minimapa es un objeto con el componente *Raw Image*. En la propiedad *Texture*, se selecciona la textura creada por la cámara que sigue al cilindro que representa al usuario. Para aprender más sobre minimapas, ver el siguiente tutorial [9]. Se añade un script basado en [2] para permitir controlar la cámara según gestos con los dedos en la pantalla táctil. Moviendo un dedo por la pantalla, la cámara se mueve en el sentido contrario. Al elegir una opción del menú desplegable, el usuario realiza ese mismo movimiento, por lo que se debe detectar cuando el menú está desplegado y no mover la cámara. Con dos dedos, se puede acercar o alejar la cámara del usuario modificando su campo de visión, *field of view*. Con dos toques rápidos, la cámara vuelve a su estado inicial.

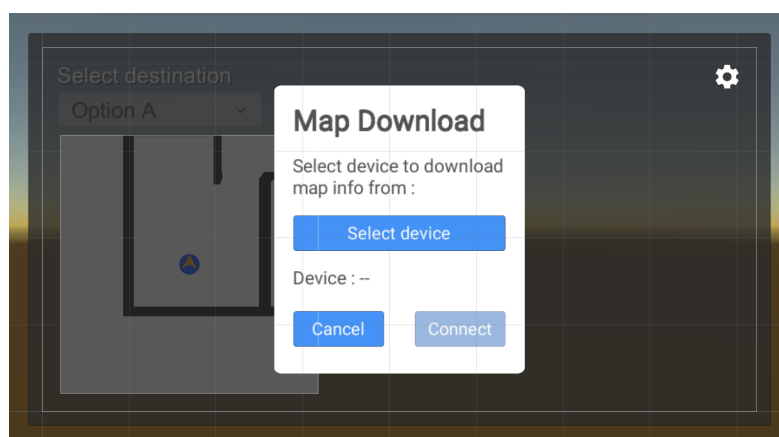
Al tocar el botón de configuraciones, se abre el menú mostrado en figura 7b. El botón *Depth* abre el menú ya presente en la plantilla *HelloAR* que permite activar o desactivar opciones de profundidad para realidad aumentada de modo que si el objeto virtual está ubicado detrás de un objeto real, se oculta la parte tapada para aumentar la inmersión de la experiencia. Se deja esta opción, sin embargo no es relevante para esta aplicación. El botón *Download map* abre el menú que se ve en figura 7c. Hay un botón que abre la interfaz de Android para seleccionar un dispositivo Bluetooth. Una vez seleccionado, se habilita el botón *Connect* que intentará establecer conexión con el dispositivo y obtener los puntos de referencia.



(a) Pantalla principal.



(b) Menú de configuraciones.



(c) Menú para descarga de datos del mapa.

Figura 7: Interfaz gráfica de la aplicación para el teléfono.

4. Resultados

Una vez que la aplicación en Unity está lista, se compila y ejecuta en el teléfono *File > Build and Run*. Se asegura que la Raspberry Pi esté ejecutando el script de servidor y que los códigos QR estén colocados adecuadamente en el sitio a navegar.

Al iniciar la aplicación, se pide en primer lugar descargar los datos de navegación. La figura 8a muestra la selección del dispositivo bluetooth para la descarga de datos de navegación. En la figura 8b se ve el mensaje recibido con dichos datos. En la figura 8c el usuario selecciona el destino. Ya se puede ver el minimapa en la esquina inferior izquierda. Las figuras 8d y 8e muestran la asistencia a la navegación tanto en el minimapa como con objetos de realidad aumentada.

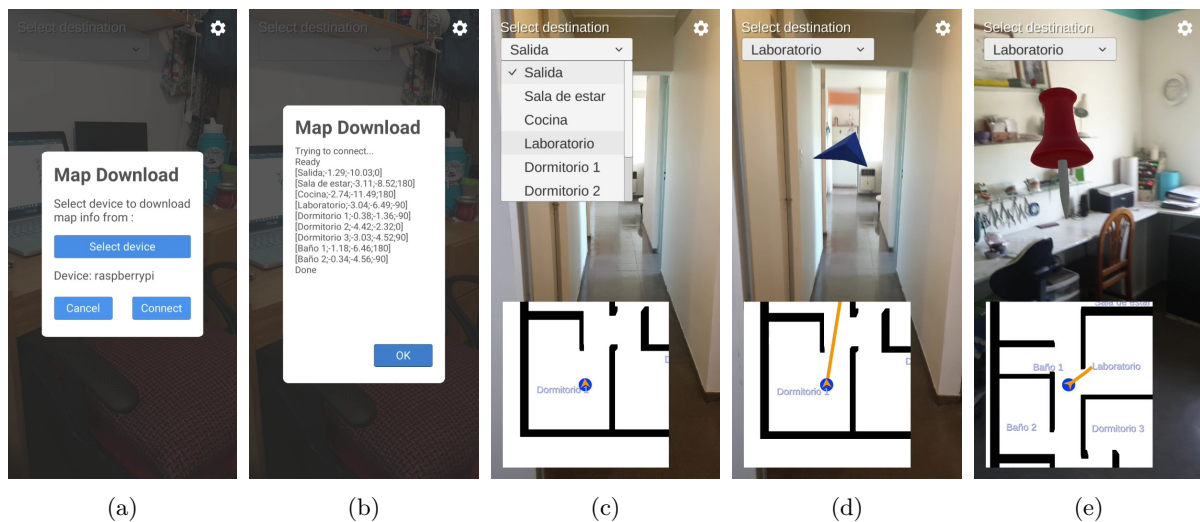


Figura 8: Captura de la aplicación en ejecución.

Puede comprobarse que, de manera general, la aplicación funciona correctamente pero, como se dijo anteriormente, el seguimiento de movimiento tiene errores acumulativos. Y estos se agravan cuando hay poca iluminación, cuando se realizan movimientos bruscos y cuando hay pocos puntos reconocibles, por ejemplo, cuando se apunta a la pared de un pasillo.

También se comprobó el efecto de activar o desactivar el componente *NavMeshAgent* del objeto usuario. Cuando está desactivado, el usuario puede atravesar paredes debido a los errores inherentes del seguimiento de movimiento. Esto produce indicaciones de navegación erróneas. Cuando está habilitado, el usuario no puede atravesar paredes, pero puede quedar atrapado en una habitación equivocada y produciendo también indicaciones falsas pero más consistentes.

Para todas estas situaciones en las que la navegación puede fallar, se soluciona rápidamente con recalibración escaneando un nuevo código QR para la relocalización del usuario.

5. Conclusiones

En el presente proyecto se desarrolló una aplicación para teléfonos Android en el entorno Unity en lenguaje C# para la asistencia a la navegación en interiores.

El abordaje del desarrollo de la aplicación se dividió en varios subsistemas. En primer lugar, un sistema de seguimiento de posición provisto por la plataforma ARCore basada en algoritmos de localización y mapeo simultáneos SLAM a partir del procesamiento de imágenes de la cámara y de datos de sensores inerciales. También se cuenta con un sistema de calibración por decodificación de códigos QR para la ubicación inicial y subsecuentes relocalizaciones para corregir errores de posicionamiento. Se implementó un sistema de navegación para encontrar el camino hasta la ubicación. El camino es indicado al usuario, por un lado, marcando la línea a seguir en el minimapa y, por otro lado, colocando objetos de realidad aumentada como flechas para indicar la dirección a seguir y un pin para indicar que se ha llegado a destino. Se realizó también un sistema de comunicación cliente-servidor a través de Bluetooth para descargar datos de navegación desde otro dispositivo de manera remota. El último subsistema es la interfaz gráfica que se diseñó para que el usuario pueda controlar y entender las funcionalidades de la aplicación de forma fácil e intuitiva.

Se probó la aplicación en el departamento del autor del proyecto y pudo comprobarse su correcto funcionamiento, así como las dificultades y errores intrínsecos a la solución elegida para la navegación de interiores.

Se deja como trabajo futuro la posibilidad de que el mapa completo se descargue desde un dispositivo externo y no solamente los puntos de referencia, la adición de indicaciones de voz para la navegación, la mejora del sistema de calibración calculando la distancia y orientación del usuario respecto al código QR para una reubicación más precisa, y sobre todo, la mejora de la precisión del sistema de seguimiento. Las técnicas de SLAM con cámaras son todavía muy recientes y se encuentran en constante mejora. Es un deseo del autor que, en un futuro cercano, esta tecnología esté lo suficiente madura para el uso comercial de asistentes de navegación en interiores.

Referencias

- [1] Roberto Lopez Mendez. *Navigation with SLAM on Your Mobile*. <https://community.arm.com/developer/tools-software/graphics/b/blog/posts/indoor-real-time-navigation-with-slam-on-your-mobile>. 2018.
- [2] Jan Hardy. *Creating an ARCore powered indoor navigation application in Unity*. <https://blog.raccoons.be/arcore-powered-indoor-navigation-unity>. 2019.
- [3] Raspberry Pi Foundation. *Setting up your Raspberry Pi*. <https://projects.raspberrypi.org/en/projects/raspberry-pi-setting-up>.
- [4] Patrick Fromaget. *Install your Raspberry Pi Zero without Keyboard and Monitor*. <https://raspberrytips.com/pi-zero-setup-without-keyboard/>. 2018.
- [5] Daniel Ellis. *How to keep Bluetooth discoverable (RPI / Unix)*. <https://medium.com/cemac/keep-bluetooth-discoverable-rpi-unix-bbe1c9ecbdb6>. 2020.
- [6] Google Developers. *Explore the HelloAR sample app code*. <https://developers.google.com/ar/develop/unity/tutorials/hello-ar-sample?hl=en>. 2021.
- [7] Brackeys. *Unity NavMesh*. <https://learn.unity.com/tutorial/unity-navmesh>. 2020.
- [8] Pybluez. *Python extension module allowing access to system Bluetooth resources*. <https://pybluez.github.io/>. 2019.
- [9] Brackeys. *How to make a Minimap in Unity*. <https://www.youtube.com/watch?v=28JTTXqMvOU>. 2017.