1. Desarrollo de la aplicación

1.1. Configuración

1.1.1. 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.

1.1.2. 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* [**setupRPI**], 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 [**headlessRPI**].

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 [**bluetoothDiscoverable**].

```
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.

```
$ sudo nano /etc/rc.local
sudo bluetoothctl <<EOF
power on
discoverable on
pairable on
EOF
```

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

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

1.2. Seguimiento de movimiento

La plantilla *HelloAR* [**helloAR**] contiene los objetos necesarios para el funcionamiento de las funciones de ARCore. *ARCore Device* contiene configuraciones de la sesión de realidad aumentada, ver figura 1. 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 2. 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.

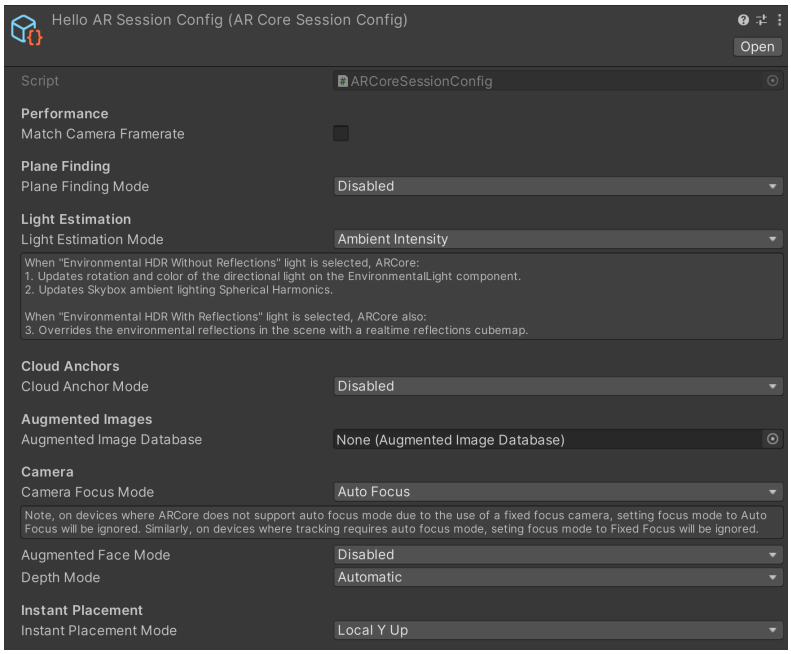


Figura 1: Configuración de la sesión de realidad aumentada.

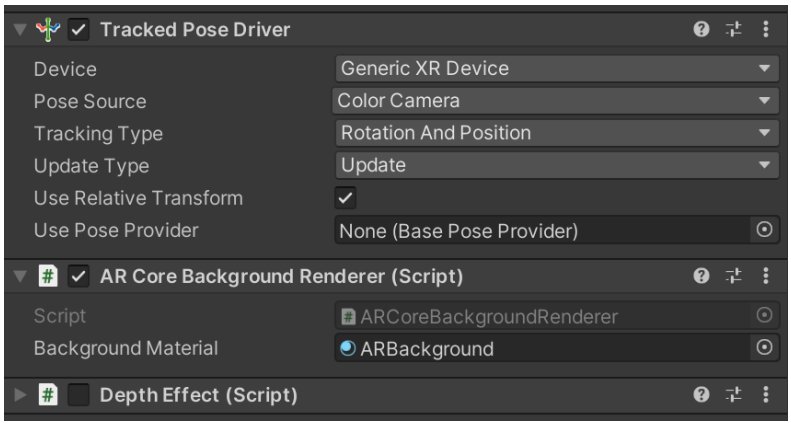


Figura 2: Configuración de la cámara en primera persona.

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 3. 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 4, con los assets de regla horizontal y vertical de *zzz Ruler* que pueden descargarse desde la *Unity Asset Store*

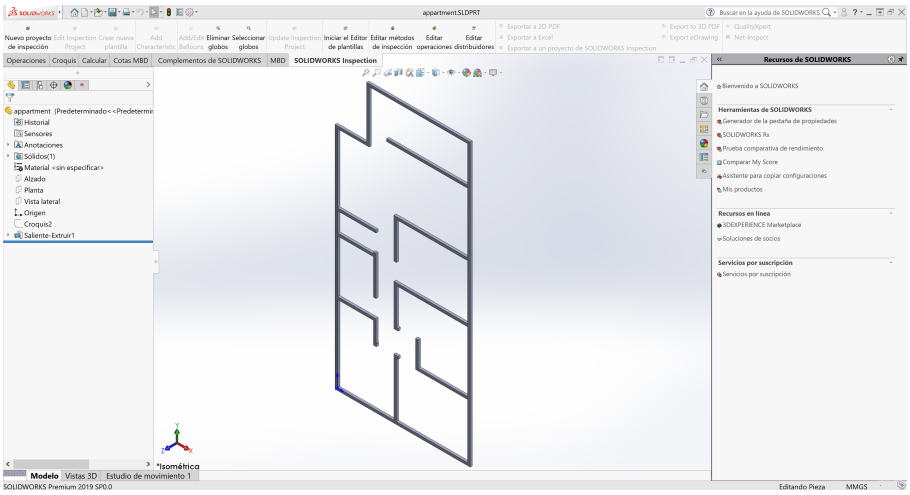


Figura 3: Mapa tridimensional del mapa del sitio a navegar hecho en SolidWorks.

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 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.

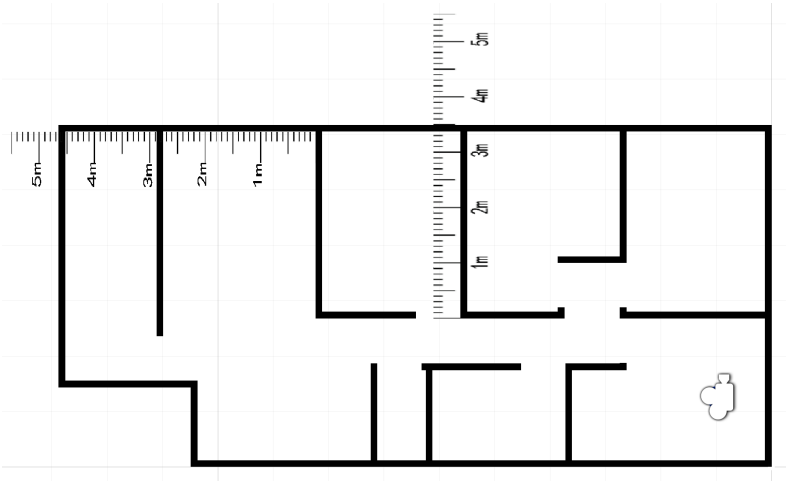


Figura 4: Uso de *zzz Ruler* para asegurar la correcta escala del mapa en Unity.

Listado 1: Código para el seguimiento de movimiento

```
// Current position in real-world
Vector3 currPos = Frame.Pose.position;
float currHeading = qrScanner.deltaOrientation + Frame.Pose.rotation.eulerAngles.y;

// Difference between positions
Vector3 deltaPos = -(currPos - prevPos);
float deltaHeading = - Mathf.DeltaAngle( prevHeading, currHeading );

// Update previous position
prevPos = currPos;
prevHeading = currHeading;

// Apply transform to Player (ignoring differences in height)
float x = deltaPos.x * Mathf.Cos( qrScanner.deltaOrientation * Mathf.Deg2Rad ) + deltaPos.z *
    Mathf.Sin( qrScanner.deltaOrientation * Mathf.Deg2Rad );
float z = deltaPos.z * Mathf.Cos( qrScanner.deltaOrientation * Mathf.Deg2Rad ) - deltaPos.x *
    Mathf.Sin( qrScanner.deltaOrientation * Mathf.Deg2Rad );
player.transform.Translate( x, 0.0f, z );

// Aply rotation to Arrow indicator and auxiliar Cube
arrow.transform.localRotation = Quaternion.Euler( new Vector3( 0, currHeading, 0 ) );
auxiliar.transform.RotateAround( player.transform.position, new Vector3( 0, 0, 1 ),
    deltaHeading );
```

1.3. 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

```
/// Capture and scan the current frame
void Scan() {
  System.Action<byte[], int, int> callback = (bytes, width, height) => {
    if (bytes == null) {
      // No image is available.
      return;
    }

    // Decode the image using ZXing parser
    try {
      IBarcodeReader barcodeReader = new BarcodeReader();
      var result = barcodeReader.Decode(bytes, width, height, RGBLuminanceSource.BitmapFormat.
          Gray8);
      string resultText = result.Text;

      // Relocate player
      if (first) {
        Relocate(resultText);
        first = false; // avoids relocating on every Update() to the same calibration point
      }
    }
    catch (NullReferenceException) { //throws NullReferenceException if no QR was found
      first = true;                 // allows relocating next time it finds a qr code
    }
  };

  CaptureScreenAsync(callback);
}
```

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

```
// Move the player indicator to the calibration point
private void Relocate(string text) {
  text = text.Trim(); //remove spaces

  //Find the correct calibration point scanned and move the player to that position
  foreach (Transform child in calibrationPoints.transform) {
    if ( child.name == text ) {
      isRelocating = true;

      if (!initialized) {
        mapUI.SetActive(true); // show map UI
        initialized = true;
```

```
13        surface.BuildNavMesh();
14      }
15
16      player.GetComponent<NavMeshAgent>().Warp( child.position );
17      deltaOrientation = Mathf.DeltaAngle( Frame.Pose.rotation.eulerAngles.y, child.
            localRotation.eulerAngles.y );
18      isRelocating = false;
19    }
20  }
21 }
```

1.4. 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 }
```

1.4.1. Navegación en minimapa

Para implementar esta funcionalidad se siguió el siguiente tutorial [**navMesh**] 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 5 muestra las configuraciones realizadas. Al hacer click en *Bake*, se genera la superficie navegable.

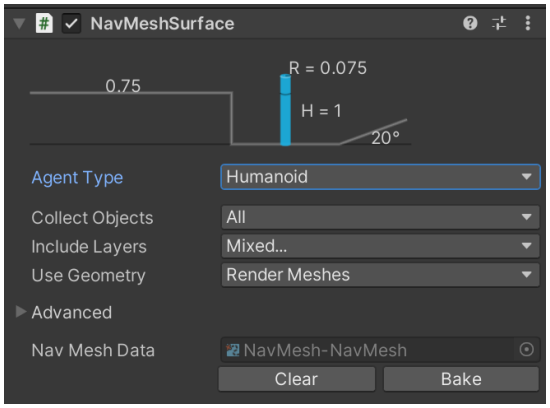


Figura 5: Configuraciones de *NavMeshSurface*.

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.

1.4.2. 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

```
private void OnTriggerEnter( Collider collider ) {
  if ( collider.tag.Equals( "Arrow" ) && !hasEnteredPinCollider ) {
    if ( line.positionCount > 0 ) {
      // Position arrow a bit before the camera and a bit up
      Vector3 pos = arcoreCamera.transform.position + arcoreCamera.transform.forward * 2.0f +
          arcoreCamera.transform.up * 0.5f;
      // Rotate arrow to neutral orientation
      Quaternion rot = arcoreCamera.transform.rotation * Quaternion.Euler( 45, 180, 0 );
      // Create new anchor
      anchorArrow = Session.CreateAnchor( new Pose( pos, rot ) );
      // Spawn arrow
      spawnedArrow = GameObject.Instantiate( arrowPrefab, anchorArrow.transform.position,
          anchorArrow.transform.rotation, anchorArrow.transform );
      // Calculate arrow angle
      Vector2 currPos = new Vector2( this.transform.position.x, this.transform.position.z );
      Vector3 pathNodeAux = line.GetPosition( 1 );
      Vector2 pathNode = new Vector2( pathNodeAux.x, pathNodeAux.y );
      Vector2 auxNode = new Vector2( auxPos.position.x, auxPos.position.y );
      float angle = Mathf.Rad2Deg * ( Mathf.Atan2( auxNode.y - currPos.y, auxNode.x - currPos.x
          ) - Mathf.Atan2( pathNode.y - currPos.y, pathNode.x - currPos.x ) );
      // Apply calculated angle
      spawnedArrow.transform.Rotate( 0, angle, 0, Space.Self );
    }
  }
  else if ( collider.tag.Equals( "Pin" ) ) {
    hasEnteredPinCollider = true;
    // Position pin a bit before the camera and a bit up
    Vector3 pos = arcoreCamera.transform.position + arcoreCamera.transform.forward * 3.0f +
        arcoreCamera.transform.up * 0.5f;
    // Create new anchor
    anchorPin = Session.CreateAnchor( new Pose( pos, Quaternion.Euler( 0, 0, 0 ) ) );
    // Spawn pin
    spawnedPin = GameObject.Instantiate( pinPrefab, anchorPin.transform.position, anchorPin.
        transform.rotation, anchorPin.transform );
    // Destroy arrow
    Destroy( spawnedArrow );
    Destroy( anchorArrow );
  }
}

private void OnTriggerExit( Collider collider ) {
  if ( collider.tag.Equals( "Pin" ) ) {
    Destroy( spawnedPin );
    Destroy( anchorPin );
    hasEnteredPinCollider = false;
  }
  else if ( collider.tag.Equals( "Arrow" ) ) {
    arrowCollider.transform.position = this.transform.position;
    Destroy( spawnedArrow );
    Destroy( anchorArrow );
```

```
  }
}
```

1.5. Descarga de datos

1.5.1. Servidor en Raspberry Pi

Para el script del server, se usa como plantilla el ejemplo *rfcomm-server.py* de PyBluez [**pybluez**]. 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)`.

```
$ bluetoothctl
# 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

```
$ 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

```
import bluetooth

server_sock = bluetooth.BluetoothSocket(bluetooth.RFCOMM)
server_sock.bind(("", bluetooth.PORT_ANY))
server_sock.listen(1)

port = server_sock.getsockname()[1]
uuid = "00001801-0000-1000-8000-00805f9b34fb"
bluetooth.advertise_service(server_sock, "SampleServer", service_id=uuid,
                        service_classes=[uuid, bluetooth.SERIAL_PORT_CLASS],
                        profiles=[bluetooth.SERIAL_PORT_PROFILE],
                        )
while True:
    print("Waiting for connection on RFCOMM channel", port)

    client_sock, client_info = server_sock.accept()
    print("Accepted connection from", client_info)

    try:
        while True:
            data = client_sock.recv(1024).decode('utf-8')
            if not data:
```

```
                break
            elif "Hello" in data:
                client_sock.send("Ready\n".encode('utf-8'))
                client_sock.send("[Salida;-1.29;-10.03;0]\n".encode('utf-8'))
                # ....
                client_sock.send("[Baño 2;-0.34;-4.56;-90]\n".encode('utf-8'))
                client_sock.send("Done\n".encode('utf-8'))
            print("Received: ", data)

    except OSError:
        pass

    print("Disconnected.")
    client_sock.close()
    print("Trying to reconnect.")

server_sock.close()
```

1.5.2. 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

```
/// Send a message to bluetooth device
void send( string msg ) {
  if ( device != null && !string.IsNullOrEmpty( msg ) ) {
    device.send( System.Text.Encoding.ASCII.GetBytes( msg ) );
  }
}

// Read message from bluetooth device
IEnumerator ManageConnection( BluetoothDevice device ) {
  while ( device.IsReading ) {
    byte[] msg = device.read();
    if ( msg != null ) {
      // Convert byte array to string.
      string content = System.Text.Encoding.UTF8.GetString( msg );
      // Split the string into lines. '\n','\r' are charachter used to represent new line.
      string[] lines = content.Split( new char[] { '\n', '\r' } );
      // Add those lines to the processText
      _processText.text += content;
    }
    yield return null;
  }
}
```

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

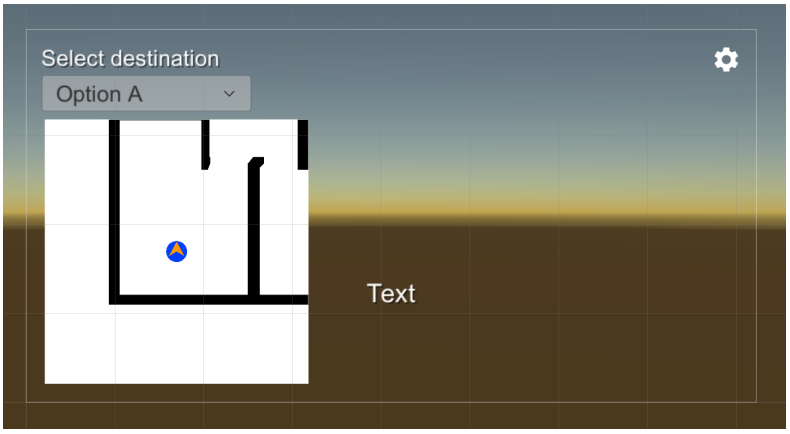
```
IEnumerator getCalibrationPoints() {
  // Start communication with server and wait until response
  yield return new WaitForSeconds( 3.0f );
  do {
    send( "Hello" );
    yield return new WaitForSeconds( 2.0f );
  } while ( !_processText.text.Contains( "Done" ) );

  // Clean existing calibration points
  List<GameObject> children = new List<GameObject>();
  foreach ( Transform child in parent ) children.Add( child.gameObject );
  children.ForEach( child => Destroy( child ) );

  // Message parsing
  string pattern1 = @"\[([^\[\]]+)\]"; // Separate calibration points from rest of message
  string pattern2 = @"[^,;\[\]\n\r]+"; // Split each calibration point into name and coordinates
  foreach ( Match m1 in Regex.Matches( _processText.text, pattern1 ) ) {
    Match mName = Regex.Match( m1.Groups[ 0 ].Value, pattern2 );
    Match mX = mName.NextMatch();
    Match mZ = mX.NextMatch();
    Match mRY = mZ.NextMatch();

    // Check if point is repeated
    bool isNameRepeated = false;
    foreach ( Transform child in parent ) {
      isNameRepeated |= child.gameObject.name == mName.Groups[ 0 ].Value;
    }
    if ( !isNameRepeated ) {
      // Add calibration points
      GameObject calibPoint = new GameObject( mName.Groups[ 0 ].Value );
      calibPoint.transform.parent = parent;
      calibPoint.transform.name = mName.Groups[ 0 ].Value;
      calibPoint.transform.localPosition = new Vector3( float.Parse( mX.Groups[ 0 ].Value ), 0.0
          f, float.Parse( mZ.Groups[ 0 ].Value ) );
      calibPoint.transform.localEulerAngles = new Vector3( 0.0f, float.Parse( mRY.Groups[ 0 ].
          Value ), 0.0f );

      // Add labels in minimap
      GameObject label = GameObject.Instantiate( labelPrefab, labelParent );
      label.transform.localPosition += calibPoint.transform.localPosition;
      label.GetComponent<TextMeshPro>().SetText( calibPoint.transform.name );
  }
  pathFinder.Clear(); // populate dropdown with calibration points
  message.text = "Please scan QR code to start";
}
```

1.6. Interfaz de usuario

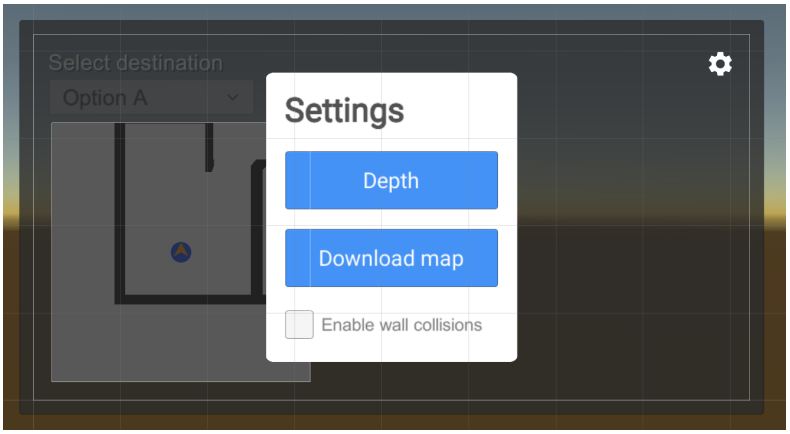
Para la interfaz de usuario que se muestra en la pantalla del teléfono, se crea un objeto *Canvas* en donde se colocan todos los objetos gráficos. En la interfaz principal, figura 6a, hay un menú desplegable en la esquina superior izquierda con los puntos de referencia para elegir el destino, el minimapa 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 [**miniMap**]. Se añade un script basado en [**indoorNavRacoons**] 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.

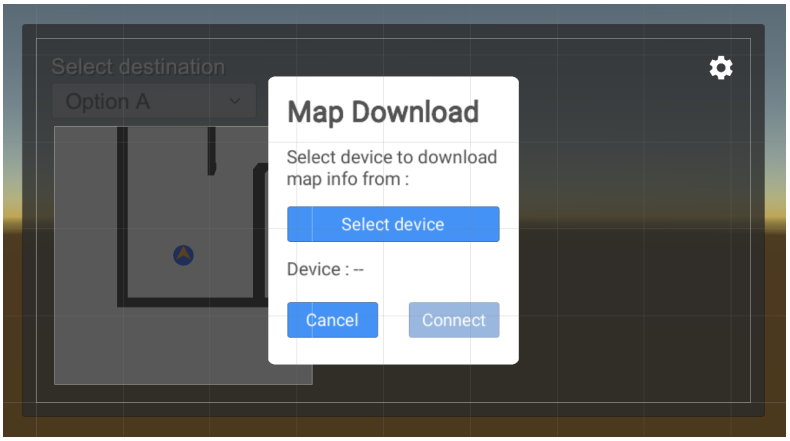
Al tocar el botón de configuraciones, se abre el menú mostrado en figura 6b. El botón *Depth* abre el menú ya presente en la plantilla *HelloAR* que permite activar o desactivar opciones de profundidad para realidad aumentada de modo que si el objeto virtual está ubicado detrás de un objeto real, se 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 6c. 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 6: Interfaz gráfica de la aplicación para el teléfono.