Fulbright Navigator

Do Duc Quan
Final Project Report
ENG301 - Computer Vision
Prof. Phung Manh Duong

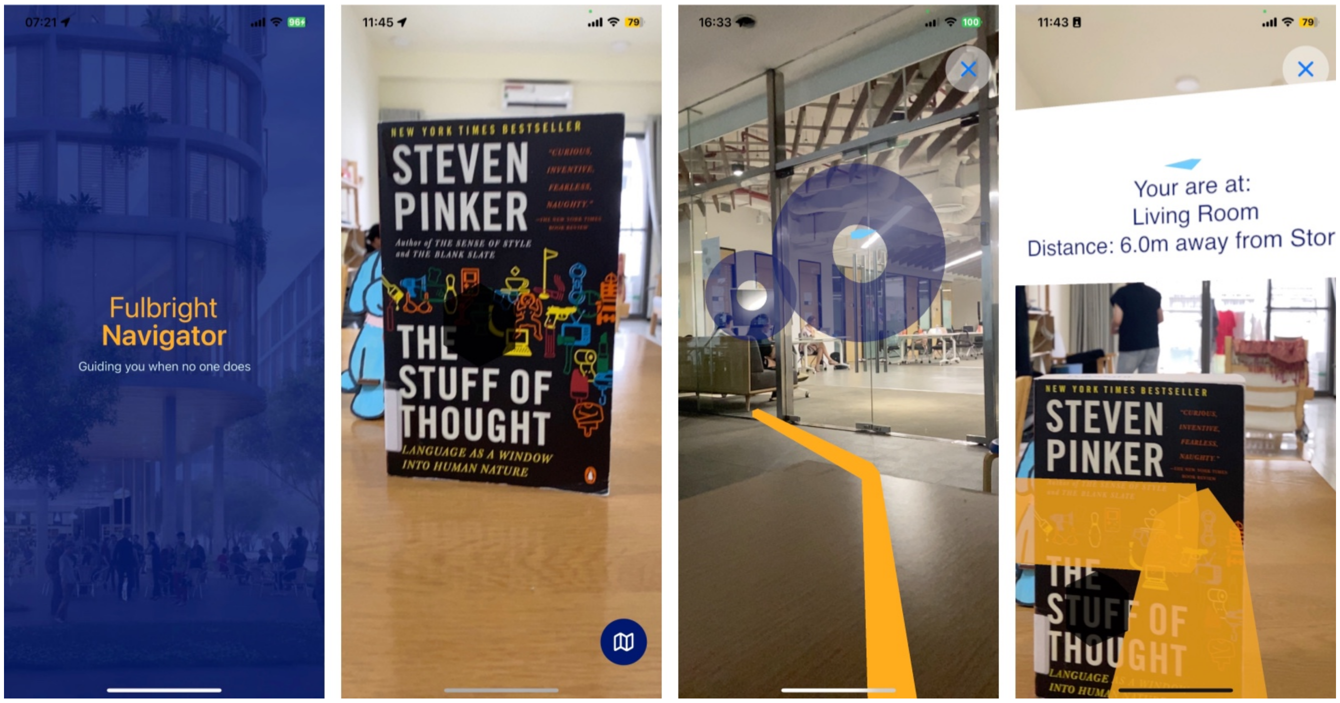


Table of Contents

1. Introduction
2. Methodology
3. Demonstration
4. Implementation
5. Final Remarks

Introduction

1. Project Description

This project is a software application for Fulbright that helps upcoming students find their way inside a campus building floor. It uses object recognition and 3D reconstruction to locate the user's position and provide shorest directions to their destination.

The program can also show information about the location and the estimated distance to the destination.

2. Features

The application's main features are as follow:

- Read a video capture and display the annotated scene onto the laptop's screen
- Detect the beacon and obtain its embedded location
- Allow manually selecting the destination (e.g. Classroom 1, Meeting room 3, etc.)
- Calculate the distance between the camera and the detected beacon. Using that information to estimate the camera/person location in the building.
- Determine the shortest path from the starting location to the destination
- Draw the route onto the video capture and display them onto the screen

3. Applications

The main application of this software is to provide guidance for students, staff, visitors, and guests to navigate around campus.

Besides, it can also enhance and promote the university's accessibility, image and reputation as a modern and innovative institution.

4. Reasons for choosing topics

I find this program interesting because it involves computer vision and human-computer interaction. I enjoy learning about computer vision technologies and how they can be integrated to create a seamless and intuitive user experience. I also like the idea of improving the quality of life and education for the university community.

Methodology

1. Used Device

For this project, I use my **iPhone 11 Pro** that has 3 cameras. Although these camera has different focal length, they can be used as a stereo system to detect the depth of an object in pixel.

However, in order to make use of this devide, I must deploy my application as a `.swift` app.

2. Technology Stack

There are 4 layers in our technology stack, which are presentation, programming language, middleware, and Firebase.

For the presentation layer, we can use **UIKit** and **ARKit**. While UIKit provides a set of tools and components for building user interfaces for iOS devices, ARKit allows us to easily integrate augmented reality features into the app by placing virtual objects in the real world and tracking the user's position and orientation.

For the programming language, we use **Swift**, which is a modern and expressive language that is designed to work seamlessly with UIKit and ARKit.

For the middleware layer, we use **NodeJS**, which is a runtime environment that enables you to run JavaScript code on the server side. NodeJS is fast, scalable, and event-driven, which makes it suitable for handling concurrent requests and data-intensive operations.

For the database layer, we use **Firebase**, which is a platform that offers a suite of services for developing and maintaining mobile and web applications. Firebase provides a real-time database that syncs data across devices and users and an authentication system that supports various sign-in methods.

3. Inputs

The inputs for the application are:

- A real-time video capture from an external camera
- A computer-language floor plan
- Beacons' id and their hard-coded location with respect to the building

Demonstration

You can find our recorded demonstration on Google Drive. The demonstration is to find the shortest path from dining table to the storage room.

> Note: Since I have removed the database service key and Pod building file from the folder for security purpose, the app cannot run. Moreover, since Swift code also requires macOS system in order to run, I am willing to give another live demonstraion if you would like to see the project in more details.

Implementation Details

1. Authentication

We use Firebase Authentication Service and make an API call to log user in. If the input username and password is correct, user will be directed to the homescreen. If not, they are prompted with an error message and required to re-enter their username/password.

Below is the snapshot of the function that handles authentication using Firebase API for Swift. It is located in ViewLogin.swift.

```
@objc func handleLogin(){
    guard let email = emailTextField.text else {return}
    guard let password = passwordTextField.text else {return}

    logUserIn(withEmail: email, password: password, completion: {
result in
        switch result {
        case .success(_):
            self.navigationController?.popViewController(animated:
true)
```

```
                self.delegate!.handleLoginButton()
            case .failure(let error):
                self.errorLabel = self.view.errorLabel(text:
error.localizedDescription)
            }
            return
        })
}
```

2. Beacon Detection

After logging in, the system will check for permission to access to the camera. If the permission is already granted, we will continunously capture the frame every `0.3` second (`3 fps`). This is because we do not want to use too much resources for detecting the beacons and 0.3 second is a reasonable number as users may not notice the delay.

Having captured the frame, we validate whether it contains the beacon by comparing it with the storaged beacon image. If the simialarity is `0.3` (we set this quite low since our marker is a black book with colored lines and small details).

Below is the snapshot code for detecting the beacon after having the captured frame. It is located in ViewAR.swift.

```
if let imageAnchor = anchor as? ARImageAnchor{
    let size = imageAnchor.referenceImage.physicalSize
    let plane = SCNPlane(width: size.width, height: size.height)
    plane.firstMaterial?.diffuse.contents =
UIColor.white.withAlphaComponent(0)
    plane.cornerRadius = 0.005

    // Image plane
    let planeNode = SCNNode(geometry: plane)
    node.addChildNode(planeNode)

    // Detect beacon
    var shapeNode : SCNNode?
    beaconImageName = imageAnchor.referenceImage.name
    switch (beaconImageName) {
```

```
    case "book", "book2", "faculty":
        shapeNode =
dataModelSharedInstance!.getNodeManager().getbeaconNode()
    default:
        shapeNode = nil
    }

    // Adds the ARObject to the marker position if there is any
    node.addChildNode(shapeNode!)
}
```

And in testing, we found that there has quite a delay for detecting the beacon, yet it is not slow enough to make us annoyed. Therefore, we say that it is acceptable.

Another thing appears in our testing: the object placement meshes up if our beacon is put laid down on the floor. Consequently, there is an assumption that we must make about the beacon setup as we will convert from camera coordinates to image coordinates to place AR objects onto the screen. Because in practice, people will scan something on the wall, not on the floor, to start navigation, we will assume that *the beacon is upright*. The code below contains the neccesary modification for our assumption.

```
// Assumes that the image is upright on a vertical surface.
let planeNode = SCNNode(geometry: plane)
planeNode.eulerAngles.x = -.pi / 2
```

4. Navigation

a. Destination Selection

After detecting the start location using the beacon, we now can choose our destination among all possible destinations shown on the screen. Then, having the end point, we query all possible routes from the database and fetch it into another function for determining the route.

The code snapshot for starting the navigation is as follow. For more details, please regard to ViewMaps.swift.

```
// Get starting and ending locations
let destinationName = locInfo.destination
let nodeList = locInfo.nodes.index
let referencedBeaconName = locInfo.beaconName

// Get all possible routes
dataModelSharedInstance!.getNodeManager().setNodeList(list: nodeList)
dataModelSharedInstance!.getNodeManager().setIsNodeListGenerated(isSet
: true)

// Begin navigation
DispatchQueue.main.async {

self.dataModelSharedInstance!.getMainVC().destinationFound(destination
: destinationName)
}
```

b. Distancing Calculation

One main section of our project is to calculate the distance between locations.

> Note: We have converted from camera coordinate system to the beacon coordinates system as above. Therefore, from now on, all notations about coordination will be in the latter system.

Since our sense of distance do not take into account of the `z` -distance, we will only consider the distance between 2 locations in `x` and `y` . With that beign said, below is the snapshot of how we can calculate the distance. For more details, the code is located in ViewAR.swift.

```
private func distance(firstNode: Index, secondNode: Index) -> Float {
    let xd: Float = firstNode.xOffset - secondNode.xOffset
    let yd: Float = firstNode.yOffset - secondNode.yOffset

    return Float(sqrt(xd * xd + yd * yd))
}
```

As you can see in the demonstration above, the calculated distance is quite off. After a while, we realized that object coordinates are recorded with regards to the previous one. For instances, the 2nd object's location is relative to the 1st's, the 3rd's is relative to the 2nd's, and so one. Therefore, we cannot directly perform calculation like above. Instead, we will calculate each distance and add them all as below.

```
private func sumDistance(fromIndex: Index) -> Float {
    let listIndex =
dataModelSharedInstance!.getNodeManager().getNodeList()
    var sumDistance: Float = 0
    var lastElement = listIndex[0]

    // Sum distances
    for (index, element) in listIndex.enumerated() {
        if element.descript == fromIndex.descript {return sumDistance}
        if index != 0 {
            let xd: Float = element.xOffset - lastElement.xOffset
            let yd: Float = element.yOffset - lastElement.yOffset
            sumDistance += Float(sqrt(xd * xd + yd * yd))
            lastElement = element
        }
    }

    return sumDistance
}
```

c. Shortest Path

Initally, we tried to build a map for Fulbright and used API to get the shorest path from the current position to another location. However, doing so was harder than we thought. Therefore, we come up with a simple algorithm as below.

The idea is to make a undirected weighted graph in which vertices are the locations and the weights of the edge are the distance between 1 location to another. Then, we enumerate through the graph and for each node, we only consider the node with

- Smallest weight edge to the previous one
- Smaller distance to the destination node comparing to the previous one

Below are the code snapshots. Details can be found in ViewAR.swift and ViewMaps.swift.

```
func setUpNavigation(renderedBeaconNode: SCNNode) -> SCNNode {
    var buildingNode : SCNNode
    let list =
self.dataModelSharedInstance!.getNodeManager().getNodeList()

    // Sets the last referenced node (source node) to the marker node

dataModelSharedInstance!.getNodeManager().setLastReferencedNode(node:
renderedBeaconNode)

    // Traverses nodeList to find the node with least weight
    for (index, _) in list.enumerated() {
        buildingNode = self.placeNode(subNodeSource:
renderedBeaconNode, to: list[index])
    }

    // Have user reached destination?
    self.checkDestinationLoop()

    return buildingNode
}

private func placeNode(thisNode: SCNNode, to: Index) -> SCNNode {
    // Gets the last destination and last referenced node
    let destinationNode =
dataModelSharedInstance!.getNodeManager().getLastScnNode()
    let lastNode =
dataModelSharedInstance!.getNodeManager().getLastReferencedNode()

    // Get distance between last node, this node, and destination node
    let distanceOld = destinationNode!.distance(receiver: lastNode!)
    let distanceNew = destinationNode!.distance(receiver: thisNode)

    // Add node to route if distance is minimized
    if distanceNew < distanceOld {
        self.nodeList!.append(thisNode)

dataModelSharedInstance!.getNodeManager().setLastReferencedNode(node:
thisNode)
    }

    return thisNode
}
```

As you can see in the demo, it can find the shorest path to the storage room. Although there is not so much nodes as I only include one further node, which is the balcony, in the maps, the result is correct and we are satisfied.

However, we can only generate the path once and cannot update that based on our location. Therefore, one improvement can be made is to make the path re-generate each time we are off the navigation.

But, this improvement is time-consuming as we must decide what threshold is large enough to be called "off navigation" and deciding how can we switch the last route with the new one without replacing them (since the user might notice that the new one is longer, so they return to the old route). Consequently, we will not integrate this into our application.

5. AR Displacement

Having each checkpoint's location, its distance and information, we can now place them onto the view to help user navigating.

First, we place arrows above the nodes and draw the line connecting locations using the snapshot code below. As usual, below are code snapshots and full methods are located at ViewAR.swift.

```
private func placeArrowNodes(node1: SCNNode, node2: SCNNode){
    // Get transformation matrix
    let referenceNodeTransform = matrix_float4x4(node1.transform)
    var translation = matrix_identity_float4x4
    translation.columns.3.x = 0
    translation.columns.3.y = 0
    translation.columns.3.z =
Float(ArkitNodeDimension.arrowNodeXOffset) * -1

    // Place arrow
    let arrow =
dataModelSharedInstance!.getNodeManager().getArrowNode()
    arrow.simdTransform = matrix_multiply(referenceNodeTransform,
translation)
```

```
    sourceNode.addChildNode(arrow)
}

private func placeLine(sourceNode: SCNNode, from: SCNNode, to:
SCNNode){
    // Place line connecting 2 nodes
    let node = SCNGeometry.floorLine(from: from.position, to:
to.position, segments: 5)
    sourceNode.addChildNode(node)
    dataModelSharedInstance!.getNodeManager().addLineNode(node: node)
}
```

We continue to display the instruction for the this the current location as follow.

```
private func placeInstruction(sourceNode: SCNNode, fromIndex: Index) -
> CALayer {
    // Get destination
    let toIndex =
dataModelSharedInstance!.getNodeManager().getLastNode()

    // Create text frame
    let textFrame = CALayer()
    textFrame.frame = CGRect(x: 0, y: 0, width: 600, height: 240)
    textFrame.backgroundColor = UIColor.white.cgColor

    // Add location information
    let text = LCTextLayer()
    text.frame = textFrame.bounds
    text.string = "Your are at: \n \(fromIndex.descript) \n Distance:
\(sumDistance(fromIndex: fromIndex).truncate(places: 2))m away from \
(toIndex!.descript)."
    text.fontSize = 30.0
    text.alignmentMode = CATextLayerAlignmentMode.center
    text.foregroundColor =
AppThemeColorConstants.fulbrightBlue.cgColor
    text.display()
    textFrame.addSublayer(text)

    return textFrame
}
```

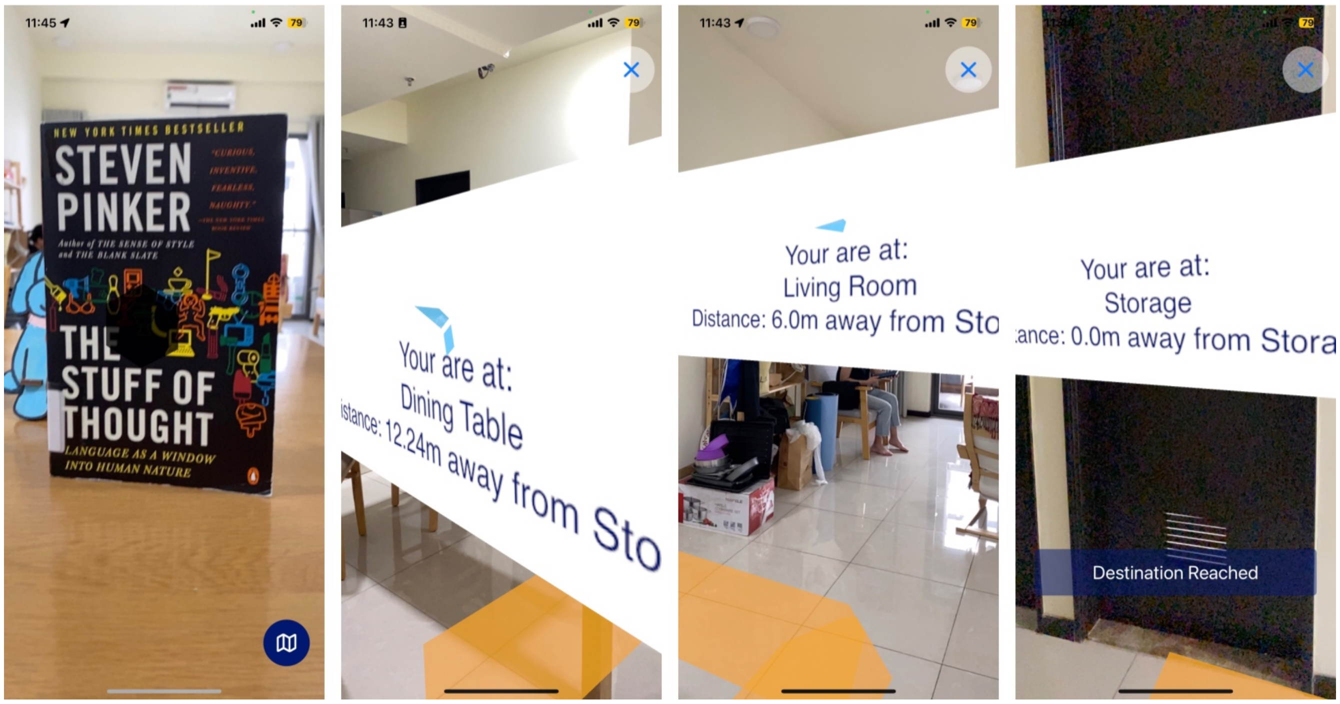
As you can see in the demo above, the placement is good enough for the user can navigate the route.

However, one major disadvantage of this approach is that there must be an origin node to calculate the location of other nodes since all locations are only relative. In the demo, we set this to be the starting point, yet we just said earlier that user may want to change their route and we may allow it in the future. But, if we stick with this origin set up, the displacement can be off if user change their route as there is no reference to the origin anymore. Besides, if we set the reference location to our location in realtime, the computional resources will be expensive. Therefore, the left option is to set it to be the destination point.

Final Remarks

After pointing out the limitations of the previous demo, we try to solve some, especially the distance calculation.

Below is the results after modification. I believe it quite interesting as the distance with regards to the origin (now becomes the destination point) is calculated correct (with error of ± 50cm as measured, which is reasonble as it doesn't affect user navigation activity).



You can find our recording for the final demo on Google Drive. The demonstration task is same as above: to find the shortest path from dining table to the storage room.