Mobile AR Application: Funiture

by

LEE Cheuk Tung

Submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science (Honours)
in Computer Science

Hong Kong Baptist University

April, 2020

Declaration

I hereby declare that all the work done in this Final Year Project is of my independent effort. I also certify that I have never submitted the idea and product of this Final Year Project for academic or employment credits.

_____

LEE Cheuk Tung

Date: _____

Hong Kong Baptist University
Computer Science Department

We hereby recommend that the Final Year Project submitted by LEE Cheuk Tung entitled "Mobile AR Application: Funiture" be accepted in partial fulfillment of the requirements for the degree of Bachelor of Science (Honours) in Computer Science.

Dr. CHOY, Martin Man Ting	Dr. HUANG, Xin
Supervisor	Observer
Date: _____	Date: _____

Abstract

Augmented Reality (AR) is a new technology in recent years that many people discuss. It can extend to the real world's content and enrich our experience. It is just like a glimpse of "Mirror World", which is a term first popularized by David Gelernter, a computer scientist at Yale University. It means that we can create a virtual world, known as "Mirror World", which is a duplicate of the real world, and we can control it, interact with it and feel it. Although it is only an imagination, AR is the basis of this dream, and there will be a chance to make this dream come true. Therefore, I am interested in AR development.

This project aims to create a mobile AR application that allows people to play around their home. They can place a furniture object into the real world and interact with it. ARKit is a tool produced by Apple Inc. in 2017 and it can provide a high-quality AR experience, so I will use it to develop the application in this project.

Acknowledgements

I would like to express my great gratitude towards my supervisor, Dr. CHOY, Martin Man Ting, who had given me invaluable advice to this project. Besides, I would also like to thank Dr. HUANG, Xin, who is the observer of this project, for his constructive comments during the development of the project.

Table of Contents

Chapter 1

Introduction

1.1 Background

Buying furniture is always annoying. Measuring size, choosing a style, considering the functions, going to a furniture shop, etc. If there is an application fulfilling all these needs, buying furniture will be more convenient. The technique which can best solve this problem is Augmented Reality (AR).

AR refers to adding virtual information to the user's sense and the user is in the real world. This technique can be traced back to the late 1960s. In 1968, Ivan Sutherland, who was a professor at Harvard University, constructed a 3D head-mounted display device with his colleagues. Wearing this device would project the information generated by the computer along with the real objects onto the wall. From the 1980s to the 1990s, only a few institutions researched AR, such as the US Air Force Armstrong Laboratory, the Massachusetts Institute of Technology, the University of North Carolina, etc. In 1992, Louis Rosenburg from the US Air Force Armstrong Laboratory created the first real operational augmented reality system, Virtual Fixtures. This system could be known as an early version of AR systems today. Later, as hardware costs fell, more research and applications about AR appeared.

Nowadays, AR has already been widely used in different areas, such as car manual, Google Glass, games, and even construction work. AR is a future

trend in Information Technology, and it makes our life better, so I choose to incorporate this technique with furniture buying.

1.2 Motivation

The idea of this project is inspired by IKEA Place which is a currently available application designed by famous furniture company IKEA. This application lets the users virtually place their products in an empty space. But this application has a problem. The virtual objects placed into the live camera scene do not look realistic enough. Therefore, in this project, I will try to apply lighting and physically based rendering in the AR application to insert more realistic looking virtual objects into the live camera scene.

1.3 Objective

The purpose of this project is to develop a mobile AR application called "Funiture", which assists the users to buy furniture. The name "Funiture" puts two words together, which are "Fun" and "Furniture". It means this application can provide the users a funny experience of buying furniture. It allows the users to:

- place more realistic looking virtual 3D furniture into the real environment
- interact with the virtual 3D furniture placed into the live camera scene by transformation, enlargement, reduction and rotation.
- search for suitable furniture in more advanced ways, by measurement of the real world and by type of plane

With the functions mentioned above, the users can place the virtual 3D furniture they want and enjoy a more realistic view. They can interact with the virtual 3D furniture, so the AR experience will be more interesting. They can enjoy a more advanced searching process, so the searching result can be more considerate.

Chapter 2

System Overview

2.1 System Development Tool

"Funiture" is an iOS AR application developed with Apple's frameworks. It runs on iOS devices that support ARKit and SceneKit.

2.1.1 ARKit

ARKit includes many different tools related to AR. Below are some features related to my project. First, ARKit can perform world tracking. ARKit can track the iOS device's position and orientation, so that we can augment the environment in front of the device.

Second, ARKit provides us with visual-inertial odometry. It acquires information from the motion-sensing device with the live camera scene first, then combines different pieces of information together to provide a precise position of the virtual 3D objects in the live camera scene.

Third, ARKit can recognize many types of surfaces in the real environment, like walls, floors, seats, etc. It first detects the feature points aligned in a real plane, then returns this piece of information as an object with basic geometry and type of plane for further usage.

2.1.2 SceneKit

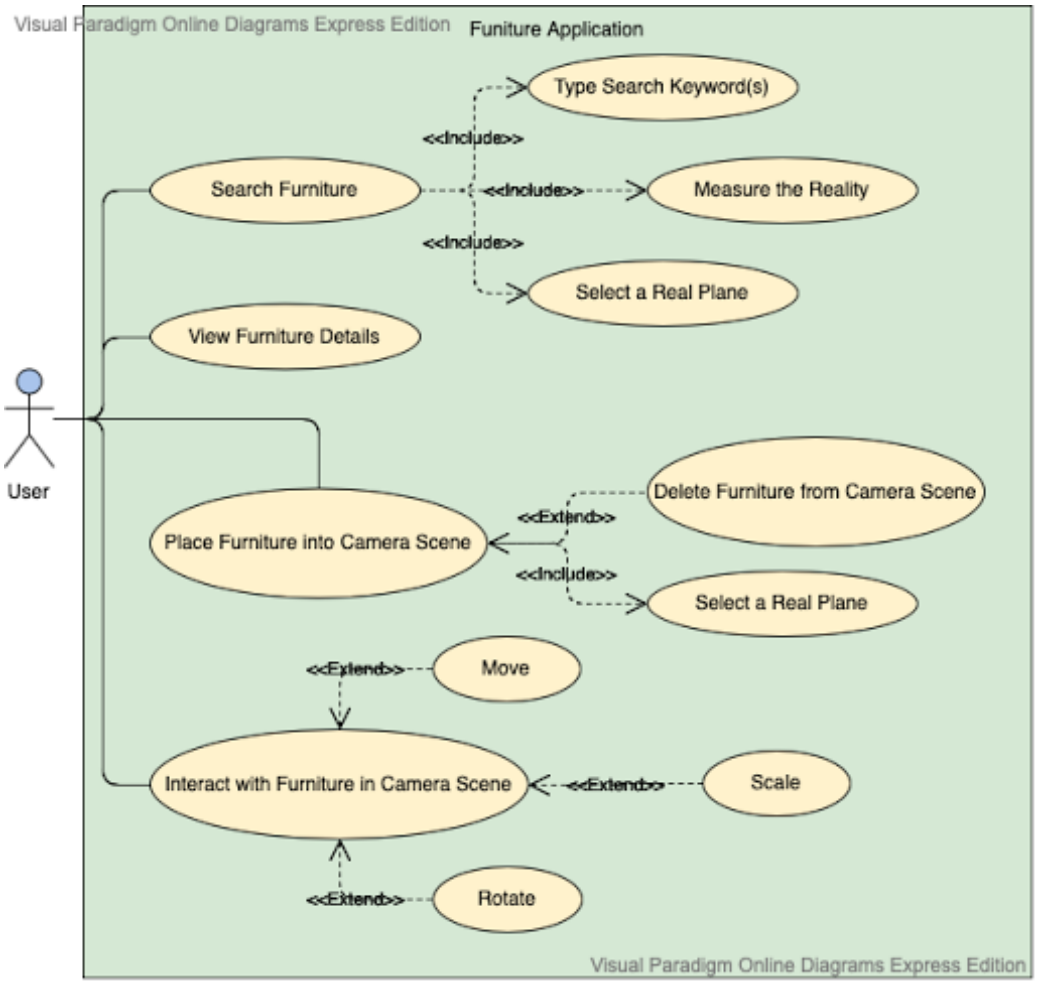
SceneKit can add 3D objects to applications. It is convenient to use because it only requires the developers to provide descriptions of the scene's actions they want. It does not require the developers to provide details of the rendering algorithms of 3D objects.

2.1.3 Scripting

In this project, I will program in the native language, Swift. And I will use Realm Studio to manage my database.

2.2 System Design

2.2.1 Use Case Diagram



2.2.1.1 Search Furniture by Keyword

Actor	User
Brief Description	The user needs to search for a furniture object before placing the object into the camera scene.
Preconditions	The user has entered the search page.

Flow of Events	1. The user enters the keyword(s) in the search bar. 2. A search result will be shown.
Postconditions	A search result is shown.
Alternative Flows and Exception	If the keyword(s) does not match any furniture object's name in the database, the system will show "No Result".

2.2.1.2 Search Furniture by Measurement

Actor	User
Brief Description	The user needs to search for a furniture object before placing the object into the camera scene.
Preconditions	The user has entered the measuring home page.
Flow of Events	1. The user clicks the "Length"/"Width"/"Height" button. < 2. The user measures the reality for the selected type of distance. 3. The user repeats step 1-2 to measure other types of distance. 4. The user clicks the "Search" button. 5. A search result will be shown.
Postconditions	A search result is shown.
Alternative Flows and Exception	1. If the measurement(s) does not match any furniture object's measurement in the database, the system will show "No Result". 2. If the user does not measure any distance then clicks the "Search" button, the system will show a warning and nothing will be searched.

2.2.1.3 Search Furniture by Plane Selection

Actor	User
Brief Description	The user needs to search for a furniture object before placing the object into the camera scene.
Preconditions	The user has entered the live camera scene showing each plane detected with its type.
Flow of Events	1. The user selects a plane detected with its type in the live camera scene. 2. The user clicks the "Search" button. 3. A search result will be shown.
Postconditions	A search result is shown.
Alternative Flows and Exception	1. If the plane type selected does not match any furniture object's plane type in the database, the system will show "No Result". 2. The "Search" button will only be shown after the user has selected a plane.

2.2.1.4 View Furniture Details

Actor	User
Brief Description	The user can view each furniture object's details.
Preconditions	The user has searched for at least one furniture object.
Flow of Events	1. The user clicks one of the furniture objects searched. 2. A detail page will be shown, displaying the corresponding details.
Postconditions	A detail page is shown, displaying the corresponding details.
Alternative Flows and Exception	If the furniture object does not have any detail in the database, the system will show "No Result".

2.2.1.5 Place Furniture into Camera Scene

Actor	User
Brief Description	The user can place a furniture object into the camera scene.
Preconditions	The user has entered the detail page of a furniture object.
Flow of Events	1. The user clicks the "Place" button below the furniture photo. 2. The system will go to the page of the live camera scene. 3. The system will detect existing planes in the live camera scene. 4. The user selects one of the detected planes to place the object. 5. The furniture object will be placed into the camera scene.
Postconditions	The furniture object is placed into the camera scene.
Alternative Flows and Exception	1. If the system fails to detect any existing plane in the live camera scene, the object cannot be placed there. 2. If the user does not select any plane detected, the object cannot be placed there.

2.2.1.6 Delete Furniture from Camera Scene

Actor	User
Brief Description	The user can delete the furniture objects from the camera scene.
Preconditions	The user has already entered the live camera scene and placed at least one furniture object into the camera scene.

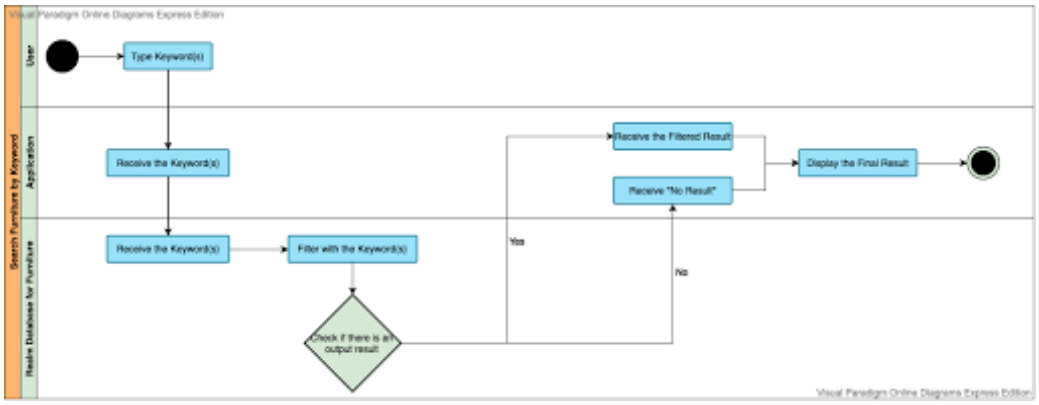
Flow of Events	1. The user selects a furniture object. 2. The user clicks the “Delete” button. 3. The furniture object will be deleted.
Postconditions	The furniture object is deleted.
Alternative Flows and Exception	1. The “Delete” button will only be shown after the user has selected a furniture object in the camera scene.

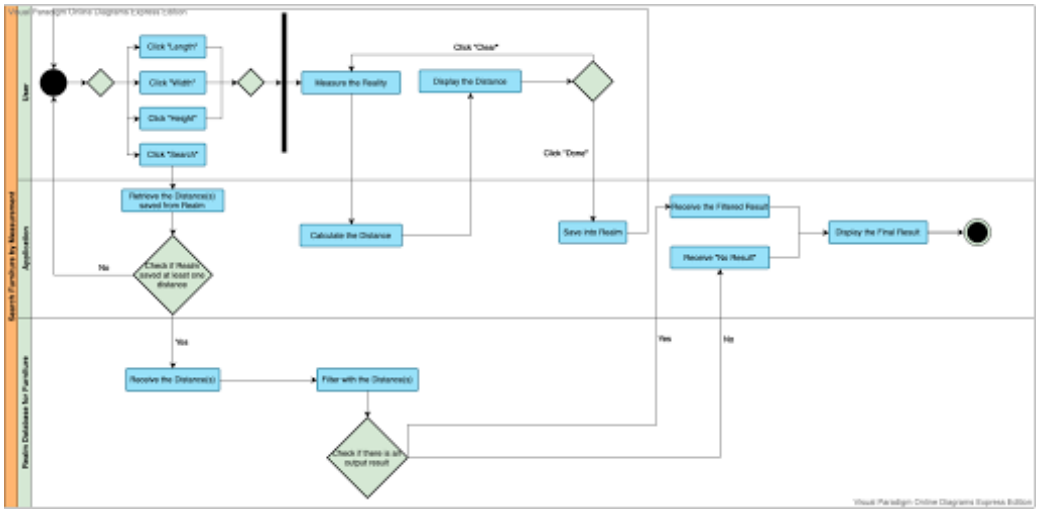
2.2.1.7 Interact with Furniture in Camera Scene

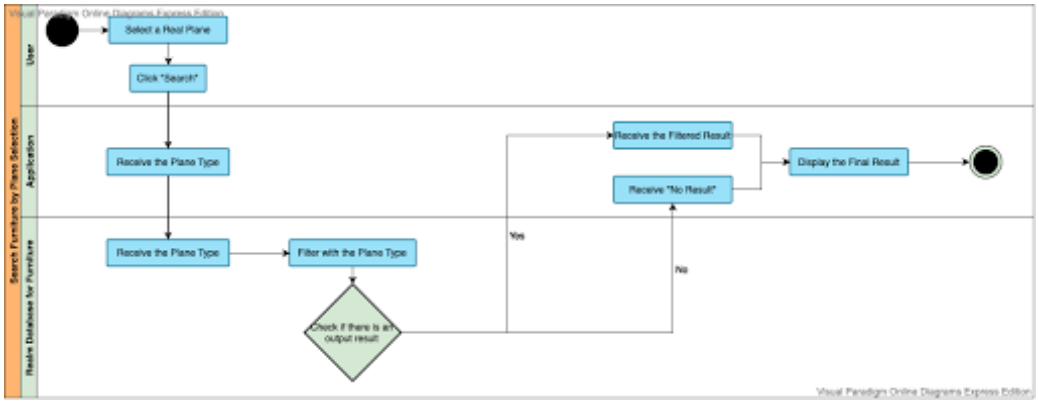
Actor	User
Brief Description	The user can move/scale/rotate the furniture objects in the camera scene.
Preconditions	The user has already entered the live camera scene and placed at least one furniture object into the camera scene.
Flow of Events	1. The user performs a pan/pinch/rotation gesture against one furniture object in the camera scene. 2. The system will move/scale/rotate the object according to the gesture. 3. The object will be moved/scaled/rotated in the camera scene.
Postconditions	The object is moved/scaled/rotated in the camera scene.

2.2.2 Activity Diagram

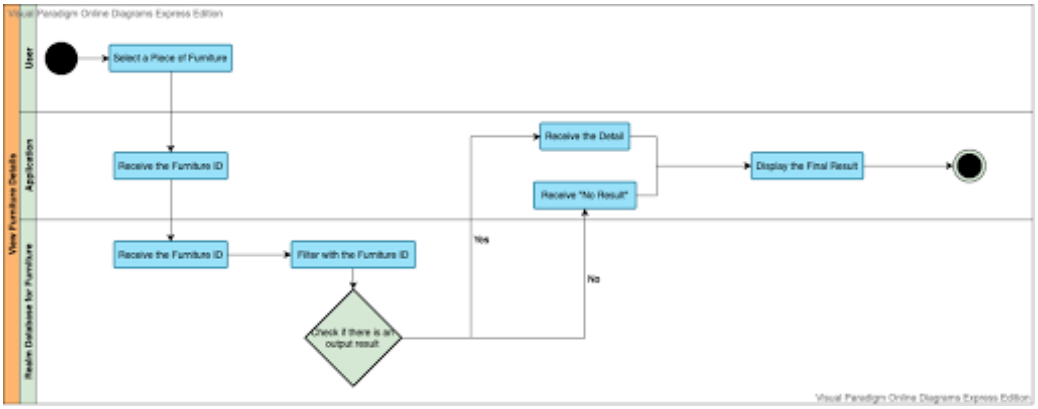
2.2.2.1 Searching



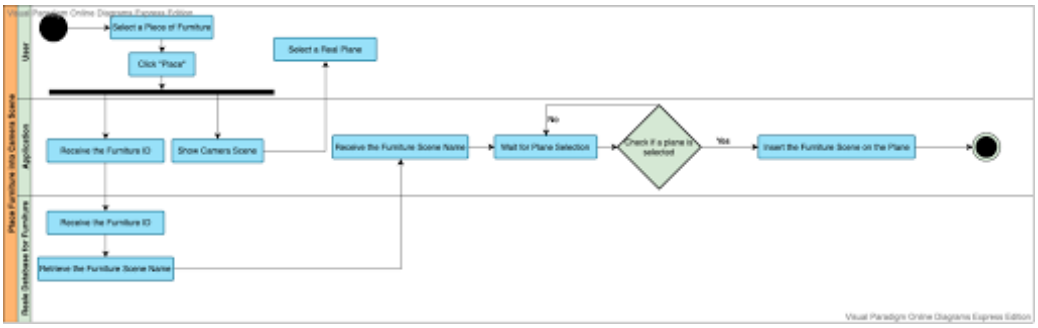




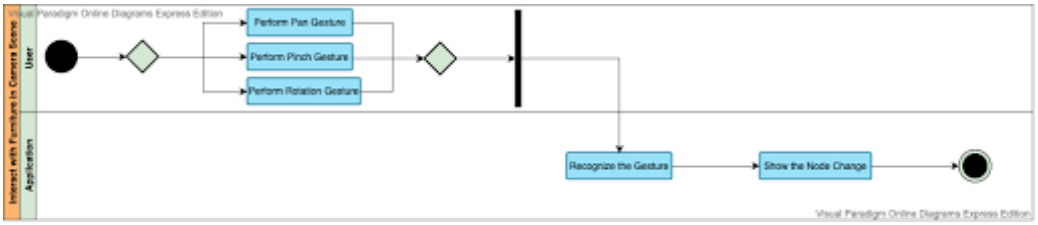
2.2.2.2 Viewing Details



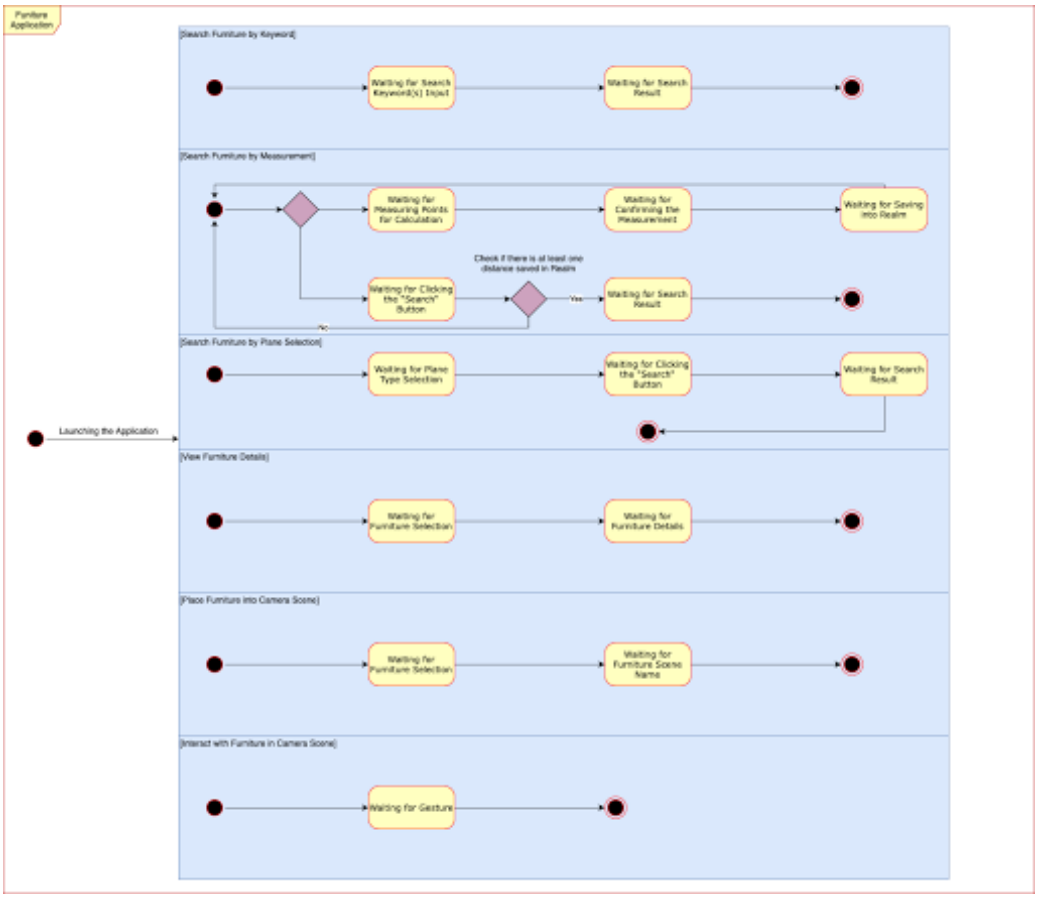
2.2.2.3 Placing Furniture

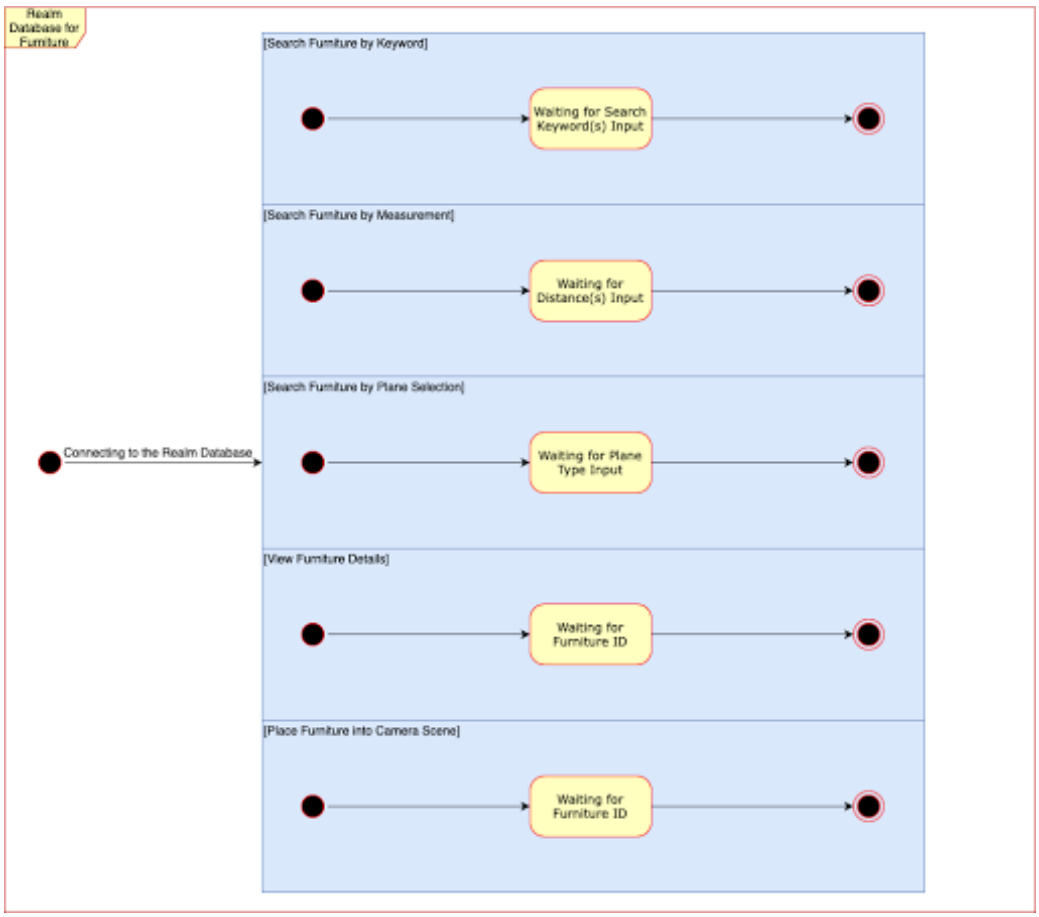


2.2.2.4 Interacting with Furniture

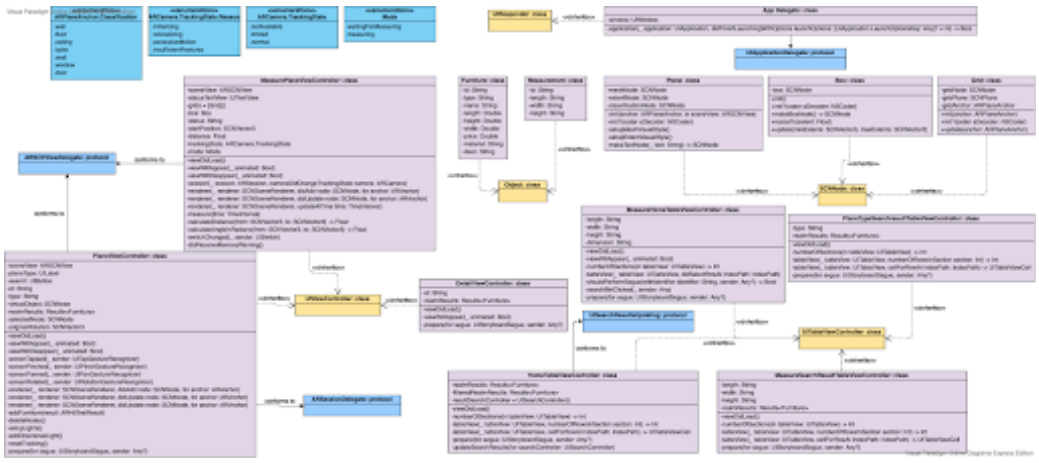


2.2.3 State Diagram





2.2.4 Class Diagram



2.2.5 Database

2.2.5.1 Realm Data Model

To show the furniture information and make use of it, I add a local database to this application. The `RealmSwift` library can be installed through CocoaPods, which is a dependency manager for iOS projects. I edit the Podfile to include `RealmSwift` as a dependency for my project.

I have prepared a Realm database file `Furniture.realm`. The model definition is as below:

```
class Furniture: Object {

    @objc dynamic var id: String? = nil
    @objc dynamic var type: String? = nil
    @objc dynamic var name: String? = nil

    let length = RealmOptional<Double>()
    let height = RealmOptional<Double>()
    let width = RealmOptional<Double>()
    let price = RealmOptional<Double>()

    @objc dynamic var material: String? = nil
    @objc dynamic var desc: String? = nil

}
```

Besides, I use the Realm for the temporary storage of measurement because the user can search after measuring all the length, width and height, more than one distance. In this process, the user will enter different views and the views will reload. To prevent the loss of the measurement when the views are reloading, I need the temporary storage to store the measured distances. The model definition is as below:

```
class Measurement: Object {
```

```
    @objc dynamic var id: String? = nil
    @objc dynamic var length: String? = nil
    @objc dynamic var width: String? = nil
    @objc dynamic var height: String? = nil

}
```

Chapter 3

Method and Implementation

3.1 Scene Preparation

Many 3D models can be found online. One of the popular websites is Free3D (https://free3d.com). In the furniture category, there are thousands of 3D models in commonly-used file formats, like *.obj*, *.dae*, *.blend*, etc. Besides, there are many sub-categories, such as lamps, desks, beds, etc. so it is user-friendly. I also used other websites like TurboSquid (https://www.turbosquid.com), CGTrader (https://www.cgtrader.com) which also provide many 3D models.

In this project, I download all 3D models online in `Collada (.dae)` file format. Collada, which stands for Collaborative Design Activity, defines an XML-based schema. The XML-based schema states clearly the geometry information about each corresponding 3D model, such as width, height, depth, position, Euler, scale, material, etc. The XML-based schema follows an open standard, so it makes Collada an interchange file format for 3D applications. Collada can transport 3D assets between applications more easily.

In this project, I use Xcode SceneKit Editor to edit the scenes. I convert the Collada file to a SceneKit file. I copy the Collada file to the `art.scnassets` folder, which is the default SceneKit Catalog. Then, I click the file, so the corresponding scene graph of the file will be displayed in the editor. In the

menu bar, I click “Editor”, then “Convert to SceneKit file format (.scn)”, so a scene file will be generated.

After the conversion, first, I remove all the unused camera and light nodes. Then, I set the material of the nodes in the material inspector. The details of the material setting can be found below at “3.6.3 Physically Based Rendering”. Next, in the scene graph, I create an empty node and the editor will add it to the center of our scene by default. Both the default position and Euler are (0, 0, 0) for (x, y, z). I rename it to “Body” and drag all other nodes underneath it. This newly-created node can act as a base node of the scene. It can group the nodes, so positioning or scaling the nodes will be easier as I do not need to perform those actions to the nodes one by one. Instead, I can perform those actions to the base node.

The bottom of a 3D object should be positioned exactly on top of the detected surface. SceneKit operates in a “Y-up” system, which means that the y-axis is pointing up. Each node shows a coordinate system. Dragging upwards or downwards the arrow pointing up is changing the y-value of the object’s position. The object will move up or down. I select all the nodes at the same time and dragged them upwards or downwards until the bottom of the object is touching the grid line in the editor. Eventually, the scene file is now ready to be used in ARKit.

Figure 3.1: Collada file

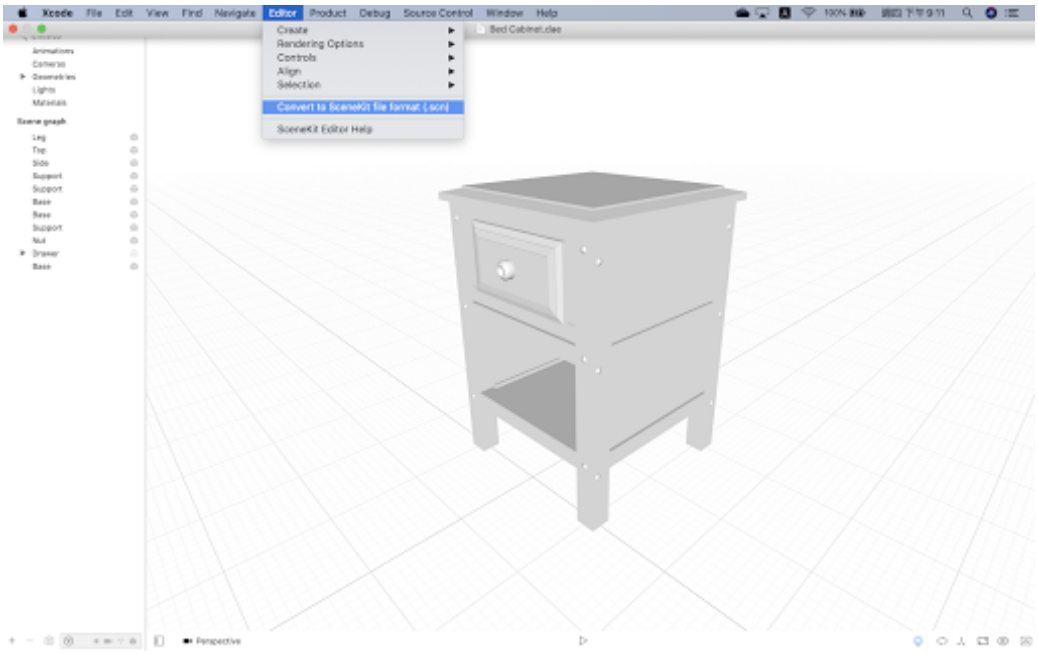


Figure 3.2: Convert the Collada file to a SceneKit file

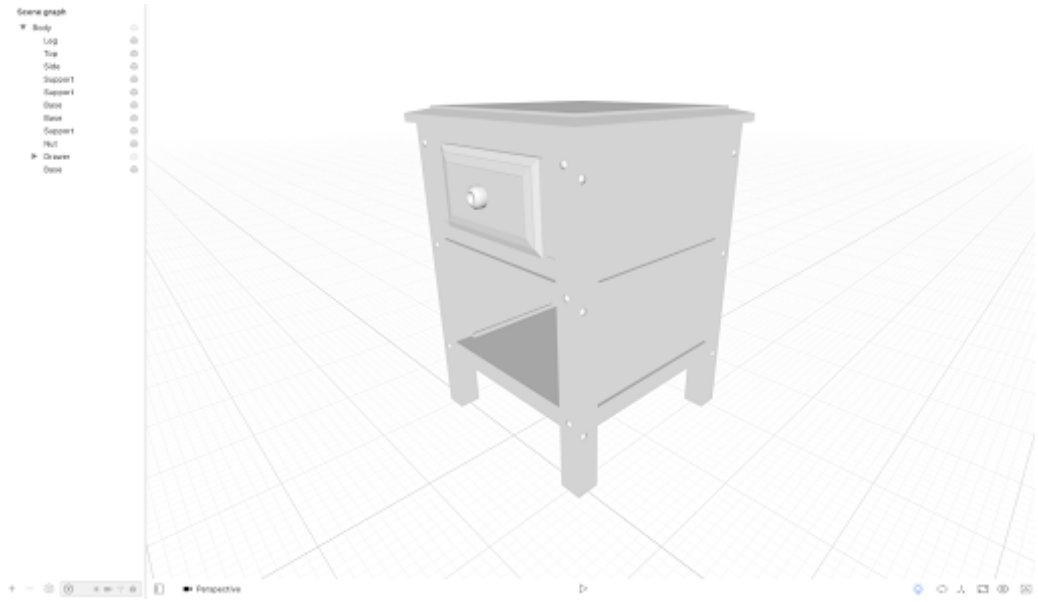


Figure 3.3: The base node is added to the scene

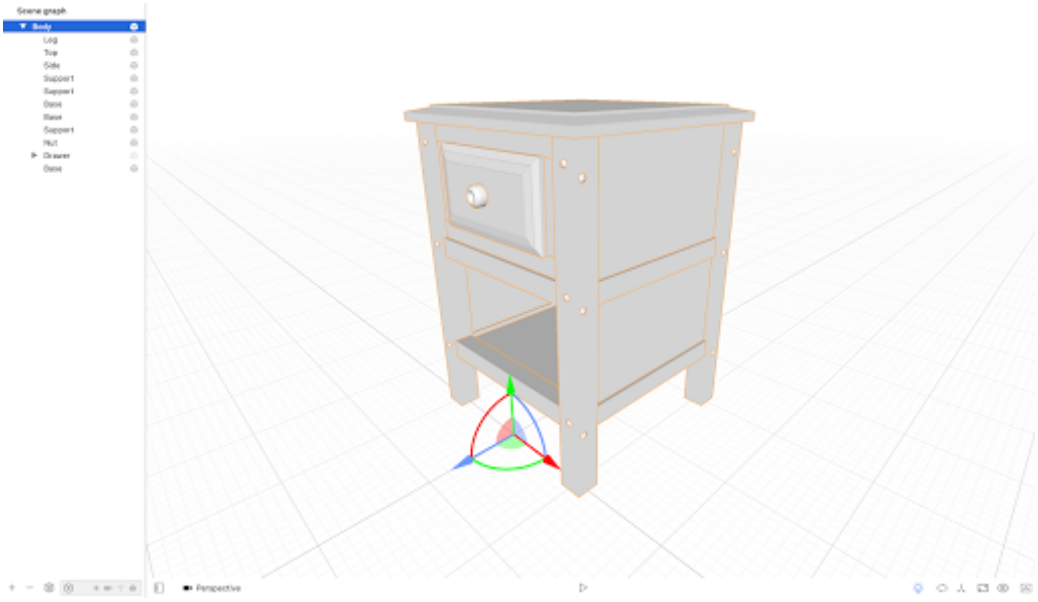


Figure 3.4: The coordinate of the base node

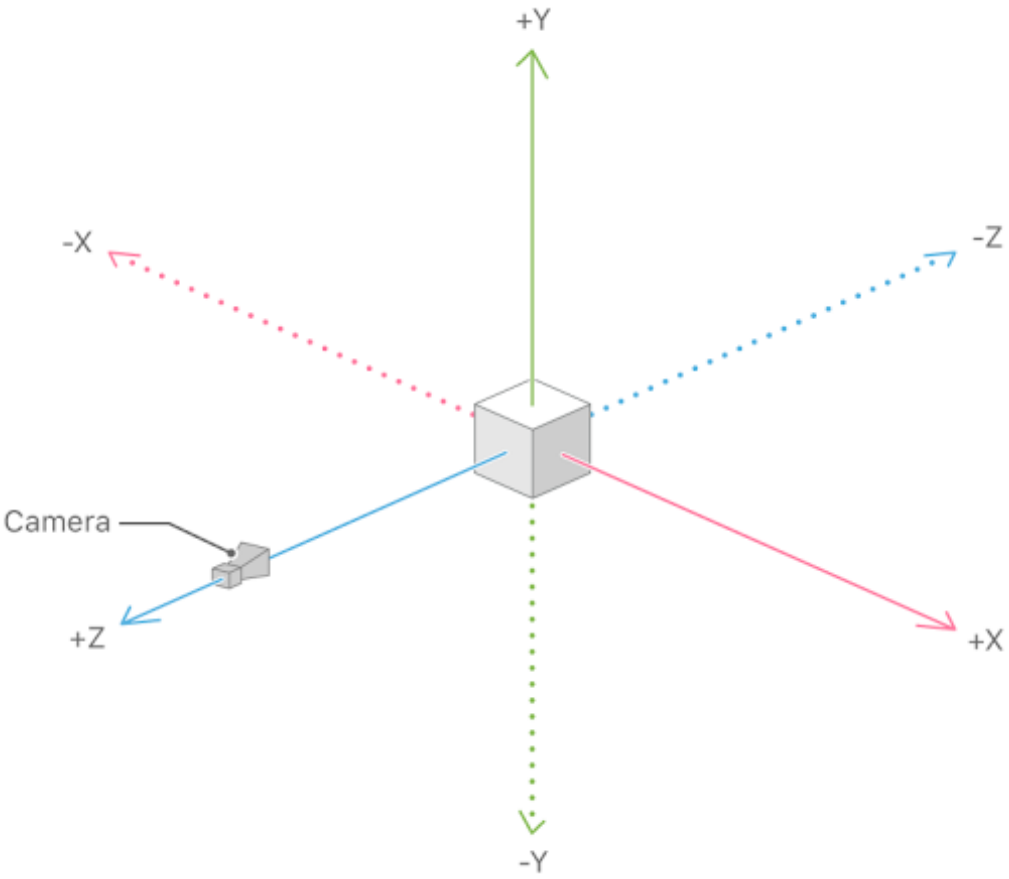


Figure 3.5: SceneKit coordinate system

3.2 Plane Detection

Before augmenting the virtual content, the first step of this application is to detect any flat surface, i.e. plane, in the real world. ARKit not only detects planes in the real world, but also recognizes many types of real-world surfaces on supported iOS devices that have an A9 or later processor. I will utilize this feature in my application.

Related class of this function:

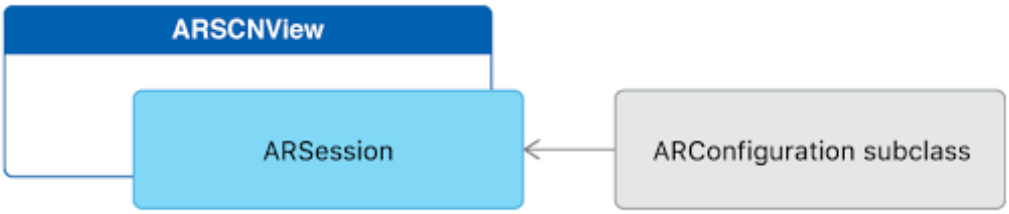
- `PlaneViewController` class
- `Plane` class

Figure 3.6: The application is performing plane detection with “Seat” detected

3.2.1 Session Configuration

`ARSession` class is the basis of controlling an AR experience. It is the main object for coordinating the major processes that ARKit performs to provide an AR experience.

First, in order to run an `ARSession` class, I provide a session configuration. In the `viewWillAppear(_ animated: Bool)` method of the `PlaneViewController` class, which is called every time the view is displayed, a session configuration is created using `ARWorldTrackingConfiguration` class that is a sub-class of `ARConfiguration` class. `ARWorldTrackingConfiguration` class can track the iOS device's position and orientation.



The relationship between ARSession and ARConfiguration subclass

Then, the `planeDetection` property, which is a type property under `ARWorldTrackingConfiguration` class, is used. This type property can find the real-world horizontal or vertical surfaces, and add them to the session as the `ARPlaneAnchor` objects.

Next, I define the value of the `planeDetection` property as both `.horizontal` and `.vertical`, so the session will attempt to automatically detect the horizontal and vertical flat planes in the live camera scene.

At last, the view's session is run by calling the `run(_:options:)` instance method.

```
override func viewWillAppear(_ animated: Bool) {

    super.viewWillAppear(animated)

    // Create a session configuration
    let configuration = ARWorldTrackingConfiguration()
```

```
        // Run the plane detection for horizontal and vertical planes
        configuration.planeDetection = [.horizontal, .vertical]

        // Enable the view's light estimation
        configuration.isLightEstimationEnabled = true

        // Run the view's session
        sceneView.session.run(configuration)

        // Set a delegate to track the number of plane anchors f
        // or providing UI feedback
        sceneView.session.delegate = self

        // Prevent the screen from being dimmed after a while
        UIApplication.shared.isIdleTimerDisabled = true

}
```

3.2.2 Plane Visualization and Classification

To visualize the planes detected, that is what the user can see a rectangle, the `Plane` class is created, which inherits from the `SCNNode` class.

In the `Plane` class's initializer `init(anchor: ARPlaneAnchor, in sceneView: ARSCNView)`, two nodes which are a mesh node to visualize the detected plane's estimated shape and an extent node to visualize the detected plane's bounding rectangle are created. They are the main components of plane visualization. The mesh node's geometry is the `ARSCNPlaneGeometry` class while the extent node's geometry is the `SCNPlane` class with the width of the x-value of the anchor's extent and the height of the z-value of the anchor's extent.

Then, because the extent node's geometry, which is the `SCNPlane` class, is vertically oriented in its local coordinate system, so it is rotated to let the

extent node match the orientation of `ARPlaneAnchor`.

At last, the two nodes are added as the child nodes using an instance method, which is `addChildNode(_:)`, so they appear in the scene.

```
// Create a mesh node to visualize the plane's estimated shape

guard let meshGeometry = ARSCNPlaneGeometry(device: sceneView.device!)
    else { fatalError("Can't create plane geometry") }

meshGeometry.update(from: anchor.geometry)

meshNode = SCNNode(geometry: meshGeometry)

// Create an extent node to visualize the plane's bounding rectangle

let extentPlane: SCNPlane = SCNPlane(width: CGFloat(anchor.extent.x),
                                     height: CGFloat(anchor.extent.z))
extentNode = SCNNode(geometry: extentPlane)
extentNode.simdPosition = anchor.center
extentNode.eulerAngles.x = -.pi / 2

super.init()
self.setupMeshNodeVisualStyle()
self.setupExtentNodeVisualStyle()

// Add the mesh node and extent node as the child nodes
// so they appear in the scene
addChildNode(meshNode)
addChildNode(extentNode)
```

Besides, a text node showing the plane type of each detected plane is created and added as a child node of the extent node. It is center-aligned in the bounding rectangle.

The plane classification feature is only available in iOS 12.0 or above,

so there is a version checking about the iOS device's version that is `if #available(iOS 12.0, \*), ARPlaneAnchor.isClassificationSupported`. `ARPlaneAnchor.Classification` is an enumeration in which the plane anchor's classification can be got. ARKit provides 7 types, which include *wall*, *floor*, *ceiling*, *table*, *seat*, *window* and *door*, for plane classification.

```
// Create a text node to show the plane type

if #available(iOS 12.0, *), ARPlaneAnchor.isClassificationSupported {

    let classification = anchor.classification.description

    let textNode = self.setupTextNodeVisualStyle(classification)

    classificationNode = textNode

    // Change the pivot of the text node to its center
    textNode.centerAlign()

    // Add the text node as a child node of the extent node
    extentNode.addChildNode(textNode)
```

Also, there are three private methods for setting up the visual style of the above mesh node, extent node and text node. Mainly, the mesh node and extent node will be made as semi-transparent by lowering the opacity's value to 0.6 so that the real-world placement can be clearly shown.

Moreover, because of the session configuration, ARKit will automatically add and update anchors in a session. In the `PlaneViewController` class, there is an optional instance method called `renderer(_:didAdd:for:)`. The view calls this method once for each new *ARPlaneAnchor* which had been added to the scene. In the `renderer(_:didAdd:for:)` method, the `Plane` class as mentioned above is instantiated each time with the `ARPlaneAnchor` detected and the current scene view to visualize the plane's mesh and extent. Then, the plane created is added as the child node using an instance method, which

is `addChildNode(_:)`, so they appear in the scene. Additionally, also in the `PlaneViewController` class, there is another optional instance method `renderer(_:didUpdate:for:)`. The view calls this method once for each updated `ARPlaneAnchor` in the scene. The method content is similar to the `renderer(_:didAdd:for:)` method above.

3.3 Furniture Search

In this project, I developed three search methods, which are by keyword, by measurement and by plane selection.

Figure 3.7: Search Furniture by Keyword

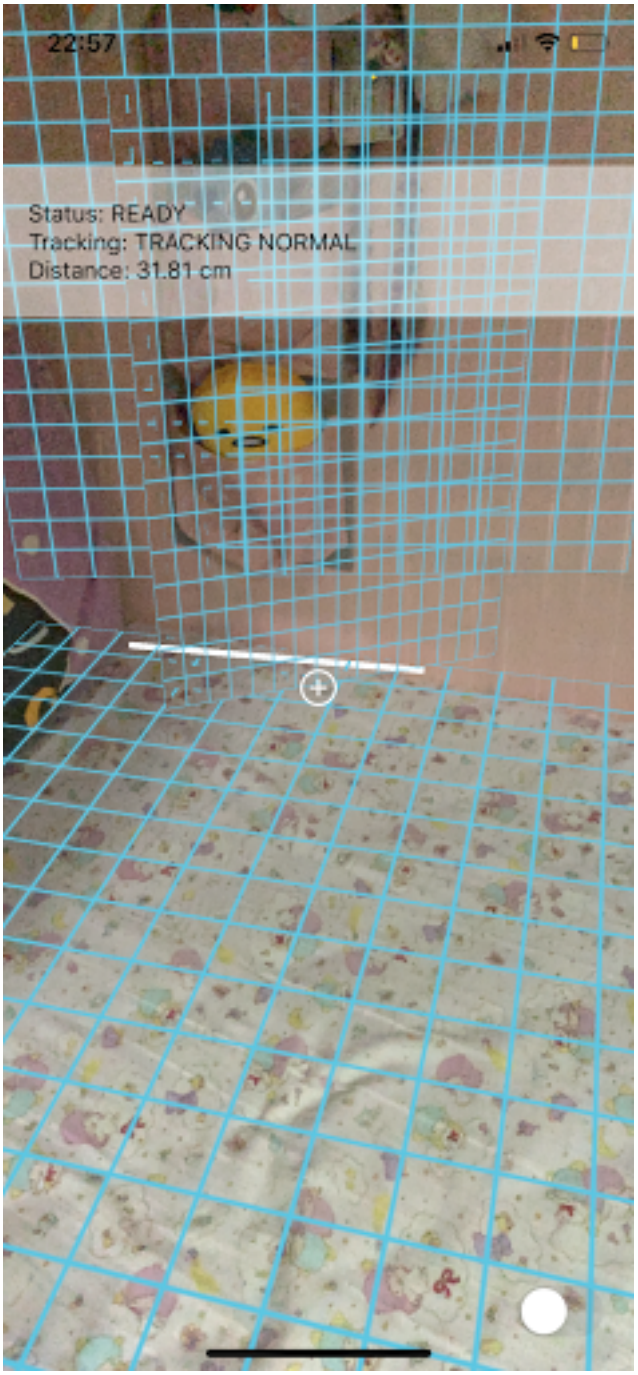


Figure 3.8: Search Furniture by Measurement

Figure 3.9: Search Furniture by Plane Selection

3.3.1 Simple Search for Furniture by Keyword

Related class of this function:

- `HomeViewController` class
- `Furniture` class

This function is done by the `HomeViewController` class. First, I create a variable which inherits from `UISearchController` class. It is the sub-class of `UIViewController` class. I added a `searchBar` property, which is an instance property of `UISearchController` class, in the `tableHeaderView`.

```
tableView.tableHeaderView = controller.searchBar
```

Then, I create a method called `updateSearchResults(for searchController: UISearchController)`. In this method, the Realm will retrieve the data in the `Furniture.realm` through the `Furniture` class, then the Realm result will be filtered by the text of the search bar.

```
// Read some data from the btrueundled Realm

filteredRealmResults = realm.objects(Furniture.self)
        .filter("name CONTAINS[c] '\(searchController.searchBar.text!)'")
```

To show the filtered result, in the instance method `tableView(_:cellForRowAt:)`, there is a checking of whether the search controller is active, which is `if (resultSearchController.isActive)`. When it is active, the filtered result will be displayed.

3.3.2 Search for Furniture by Measurement

Related class of this function:

- `MeasurePlaneViewController` class

- `MeasureHomeTableViewController` class
- `MeasureSearchResultTableViewController` class
- `Grid` class
- `Box` class
- `Measurement` class
- `Furniture` class

3.3.2.1 Session Configuration

First of all, plane detection would be enabled. Similar to the implementation of plane detection above, in the `viewWillAppear(_ animated: Bool)` function of the `MeasurePlaneViewController` class, a session with session configuration of plane detection will be created and run.

3.3.2.2 Grid Visualization

To visualize each plane detected as a grid, the `Grid` class is created, which inherits from the `SCNNode` class. It is similar to the `Plane` class as mentioned above.

In the `Grid` class's initializer `init(anchor: ARPlaneAnchor)`, a mesh node to visualize the detected plane's estimated shape is created. It is the main component of grid visualization. The mesh node's geometry is the `SCNPlane` class with the width of the x-value of the anchor's extent and the height of the z-value of the anchor's extent.

Also, I set up the grid's material using the `SCNMaterial` class. I set the diffuse contents of the material as an image of grid as it is more convenient for the users to make a point on a gird for the start and end position of the distance they want to measure.

Besides, the `physicsBody` of the mesh node is the `SCNPhysicsBody` class. Its type is *.static*, and its shape is the `SCNPhysicsShape` class. Also, the `categoryBitMask` of the `physicsBody` of this node is equal to 2, which means that this body is static, as the default value of `categoryBitMask` is

1, which means that the body is dynamic.

Moreover, I will use `SCNVector3Make(_:_:_:)` to set the mesh node's position. The x-value is the x-vlaue of the anchor's center, while the z-value is the z-vlaue of the anchor's center. The y-value is 0. It will return a three-component vector.

Next, same as before, because the mesh node's geometry, which is the `SCNPlane` class, is vertically oriented in its local coordinate system, so it is rotated to let the extent node match the orientation of `ARPlaneAnchor`.

At last, the mesh node is ready and it is added as the child node using an instance method, which is `addChildNode(_:)`, so it appears in the scene.

```
// Create a plane
gridPlane = SCNPlane(width: CGFloat(anchor.extent.x),
                height: CGFloat(anchor.extent.z))

// Set up the grid's material
let material = SCNMaterial()
material.diffuse.contents = UIImage(named: "overlay_grid.png")

gridPlane.materials = [material]

// Create a mesh node to visualize the plane's estimated shape
gridNode = SCNNode(geometry: gridPlane)
gridNode.physicsBody = SCNPhysicsBody(type: .static,
        shape: SCNPhysicsShape(geometry: gridPlane, options: nil))
gridNode.physicsBody?.categoryBitMask = 2
gridNode.position = SCNVector3Make(anchor.center.x, 0, anchor.center.z);
gridNode.transform
    = SCNMatrix4MakeRotation(Float(-Double.pi / 2.0), 1.0, 0.0, 0.0);

super.init()

// Add the mesh node as child node so it appears in the scene
```

```
addChildNode(gridNode)
```

Similar to the implementation of plane detection above, in the the `MeasurePlaneViewController` class, two optional instance methods, which are `renderer(_:didAdd:for:)` and `renderer(_:didUpdate:for:)`, will be run to add and update the grids in the scene view.

3.3.2.3 Measuring Path Visualization

The measuring path will being drawn when the user is measuring the reality.

First, the `Box` class is created for visualization of the measuring path. It inherits from the `SCNNode` class. In this class, first, there is a method for creating a box node. The geometry of the box node is the `SCNBox` class, which is a sub-class of `SCNGeometry`. It is a six-sided polyhedron geometry whose faces are all rectangles. It is created with the specified width, height and length and chamfer radius. The chamfer radius is the radius of the curvature for the edges and corners of the box. It is 0 here, which means that the box will not have any round edge. The material of the box is assigned as uniform shading with `UIColor` of white.

Then, the box node is ready and it is added as the child node using an instance method, which is `addChildNode(_:)`, so it appears in the scene.

```
func makeBoxNode() -> SCNNode {

let box = SCNBox(width: 0.005, height: 0.005, length: 0.005, chamferRadius: 0)

    for material in box.materials {
        material.lightingModel = .constant
        material.diffuse.contents = UIColor.white
    }

    let node = SCNNode(geometry: box)
    self.addChildNode(node)
```

```
    return node
}
```

Besides, in the `viewDidLoad()` method of the `MeasurePlaneViewController` class, the box node is instantiated using the `Box` class and also added as the child node using an instance method, which is `addChildNode(_:)`, so it appears in the scene. But it is hidden before the user starts the measurement.

To know when the user starts the measurement, I use a variable called `mode` to manage the state of waiting for measuring and measuring. I also create an enumeration, `Mode`, to contain two cases, `waitingForMeasuring` and `measuring`. The variable `mode` is `waiting for measuring` at the beginning. When the user starts the measurement by switching on the switch, the variable `mode` will change from `waiting for measuring` to `measuring`, so the property observer `didset` of `mode` will get called because new value is assigned to `mode`. Then, the private method `update(minExtents: SCNVector3, maxExtents: SCNVector3)` will call to update both the minimum and maximum extent position of the box to `SCNVector3Zero`, which means that every component of the vector is 0.0. It is to give an initial position to the box node. Also, the box, which is the measuring path, will become visible in the view.

```
enum Mode {

    case waitingForMeasuring
    case measuring

}

var mode: Mode = .waitingForMeasuring {

    didSet {
```

```
        switch mode {

            case .waitingForMeasuring:
            status = "NOT READY"

            case .measuring:

            box.update(minExtents: SCNVector3Zero, maxExtents: SCNVector3Zero)
            box.isHidden = false

            startPosition = nil
            distance = 0.0

            setStatusText()
        }

    }

}
```

Moreover, the measuring path will move according to the device's movement when the user is measuring the reality. Therefore, `resizeTo(extent: Float)` method is needed to update the path's size, especially the length.

```
func resizeTo(extent: Float) {

    var (min, max) = boundingBox

    max.x = extent

    update(minExtents: min, maxExtents: max)

}
```

The user will change direction when measuring the reality, the path's direc-

tion will also change, so `calculateAngleInRadians(from: SCNVector3, to: SCNVector3) -> Float` method is needed to method to get the direction changes and the rotation value of the path can be set with the angle in redials the method returned.

```
func calculateAngleInRadians(from: SCNVector3, to: SCNVector3) -> Float {

    let x = from.x - to.x
    let z = from.z - to.z

    return atan2(z, x)

}
```

3.3.2.4 Calculation of Distance

In the `MeasurePlaneViewController` class, I use the `measure(time: TimeInterval)` method to perform the measurement and distance calculation. The two variables, `startPosition` and `worldPosition` are the main parts of the measurement and distance calculation.

First of all, the `startPosition` is the screen center. The screen center can be found with `midX` and `midY` of scene view's bounds. A hit test is used to search for whether the screen center is within any existing plane's area. If the screen center is within any existing plane's area, given that the user has already switched on the switch and the `mode` is `measuring`, the measurement starts.

Then, the `worldPosition` will be got by `SCNVector3Make(_:_:_:)`. As the user will keep on moving the iOS device, like the iPhone, when measuring, which means that the screen center will keep on moving corresponding to the movement of the device, so the hit test result will keep updating and setting the variable `worldPosition` as the hit test location simultaneously. Also, the distance will keep on being calculated and updated at the same time. The calculation is using the Pythagoras Theorem. When the user stops

measuring by switching off the switch, the update of the hit test result and the distance calculation will stop. We can get the final distance eventually.

```
func measure(time: TimeInterval) {

    let screenCenter: CGPoint = CGPoint(x: self.sceneView.bounds.midX, y: self.

    let planeTestResults = sceneView.hitTest(screenCenter, types: [.existingPla

    if planeTestResults.first != nil {

        if let result = planeTestResults.first {

            status = "READY"

            if mode == .measuring {

                status = "MEASURING"

                let worldPosition = SCNVector3Make(result.worldTransform.columns

                if startPosition == nil {
                    startPosition = worldPosition
                    box.position = worldPosition
                }

                distance = calculateDistance(from: startPosition!, to: worldPos
                box.resizeTo(extent: distance)

                let angleInRadians = calculateAngleInRadians(from: startPositio
                box.rotation = SCNVector4(x: 0, y: 1, z: 0, w: -(angleInRadians

            }

        } else {
```

```
            status = "NOT READY"

        }

    }

    setStatusText()

}

func calculateDistance(from: SCNVector3, to: SCNVector3) -> Float {

    let x = from.x - to.x
    let y = from.y - to.y
    let z = from.z - to.z

    return sqrtf( (x * _x) + (y_ * y) + (z * z))
}
```

The measure() method is called in the optional instance method, `renderer(_:updateAtTime:)` in the `MeasurePlaneViewController` class. SceneKit calls this method exactly once per frame so that the distance will be updated per frame. Additionally, this method is called asynchronously using `DispatchQueue.main.async` to perform the heavy calculation task without slowing down the UI.

```
func renderer(_ renderer: SCNSceneRenderer, updateAtTime time: TimeInterval) {

    // Call the heavy method asynchronously without slowing down the UI
    DispatchQueue.main.async {
        self.measure(time: time)
    }
}
```

3.3.2.5 Search Result Filtering

The final measured distance will be saved to the temporary Realm using the `Measurement` class which is the Realm data model, and passed to the `MeasureHomeTableViewController` class using `segue.destination`. The `MeasureHomeTableViewController` class manages the scene showing a table view of the "Search", "Length", "Width", "Height" button. The user tells the application which type of distance he is going to measure by clicking different buttons.

When the user clicks "Search" and given that there is at least one distance of length, width and height are saved into Realm, the distance(s) will be the filtering parameter for searching for furniture and passed using `segue.destination` to the `MeasureSearchResultTableViewController` class as the measurement for searching for furniture. The Realm will retrieve the data in the `Furniture.realm` through the `Furniture` class, and the Realm result will be filtered by the length, width and/or height and the final result will be displayed.

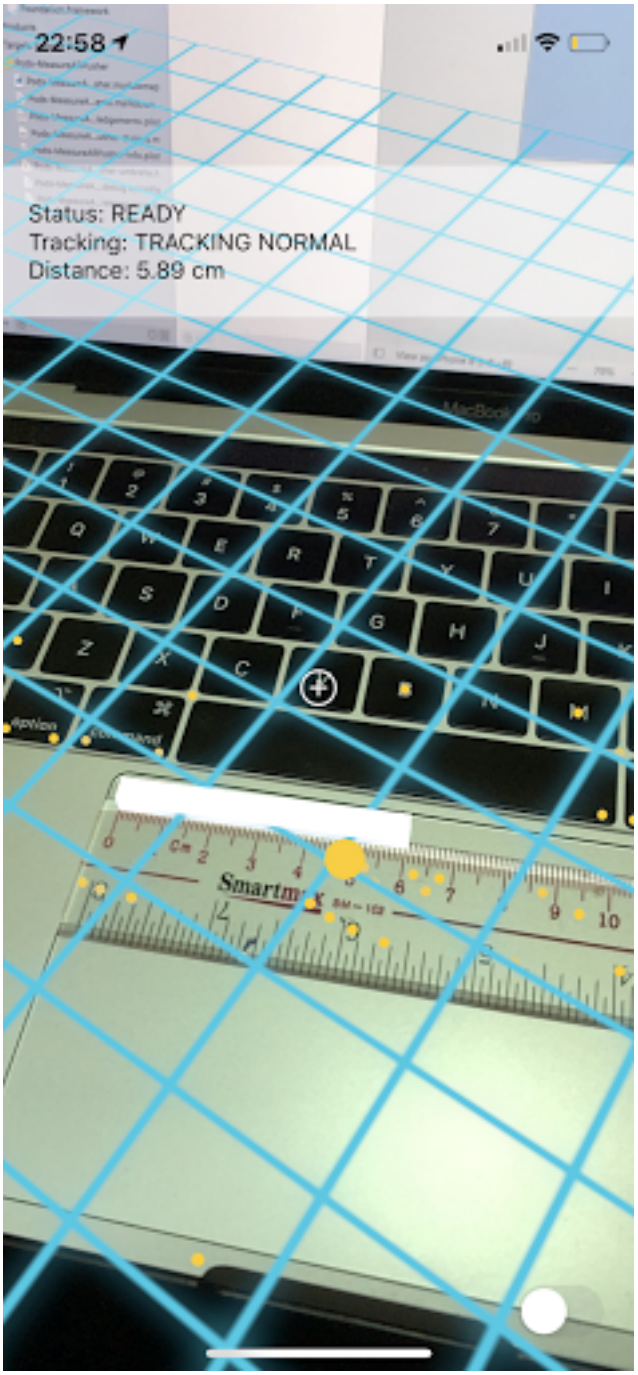


Figure 3.10: The measuring path and the distance result

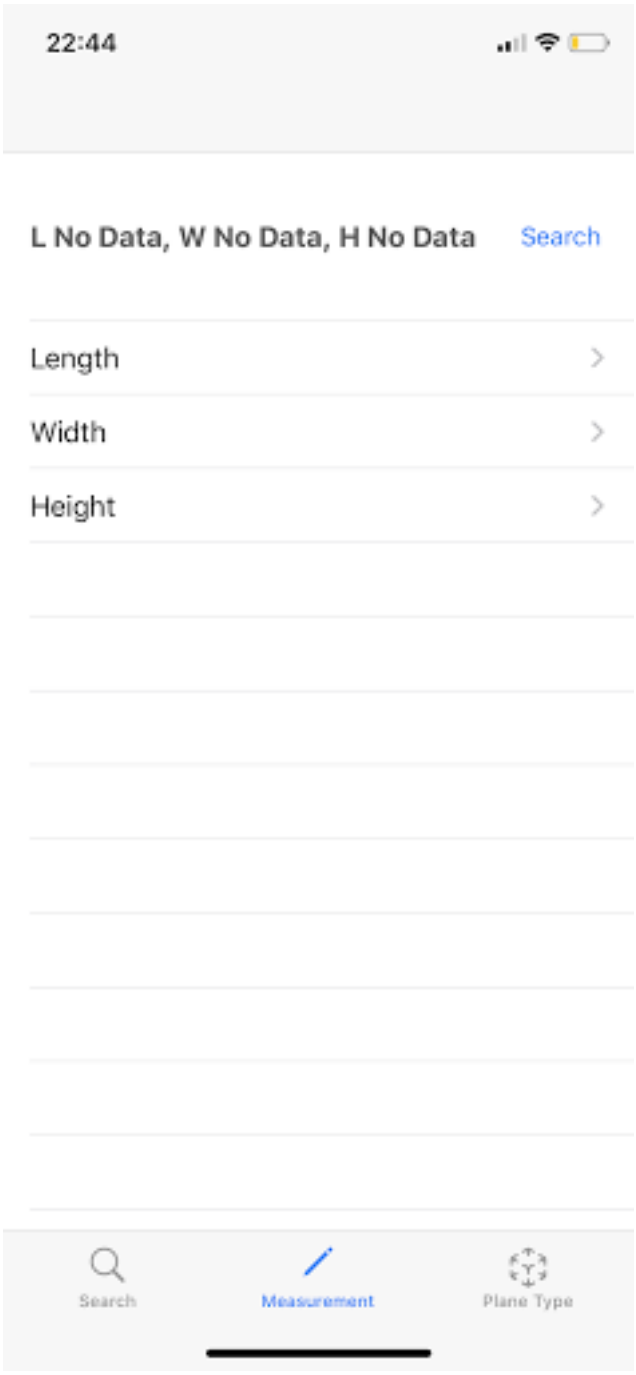


Figure 3.11: The home page of measuring

3.3.3 Search for Furniture by Plane Selection

Related class of this function:

- `PlaneViewController` class
- `PlaneTypeSearchResultTableViewController` class
- `Plane` class
- `Furniture` class

As mentioned above, ARKit provides plane classification for the real-world surfaces. ARKit can recognize 7 surfaces in total, namely *wall*, *floor*, *ceiling*, *table*, *seat*, *window* and *door*. When the plane anchor information is not enough for classification, the classification will return *none*. This feature is only available in iOS 12.0 or above, so in the `Plane` class, there is a version checking about the iOS device's version that is `@available(iOS 12.0, *)` before the extension of enumeration `ARPlaneAnchor.Classification`.

```
@available(iOS 12.0, *)
extension ARPlaneAnchor.Classification {

    var description: String {

        switch self {

            case .wall:
            return "Wall"

            case .floor:
            return "Floor"

            case .ceiling:
            return "Ceiling"

            case .table:
            return "Table"
```

```
            case .seat:
            return "Seat"

            case .window:
            return "Window"

            case .door:
            return "Door"

            case .none(.unknown):
            return "Unknown"

            default:

            return ""

        }
    }
}
```

When the user taps the scene view, the touch location will be got in the `PlaneViewController` class. The hit test of `existingPlaneUsingExtent` will search for any point which lies within the area defined by the plane's center and extent properties in the touch location. If the hit test successfully gets a point, which means that the user has tapped an existing plane area, an object is created using the `Plane` class and the anchor as `ARPlaneAnchor` of the hit test result will be got.

```
if let planeAnchor = result.anchor as? ARPlaneAnchor {

    if planeAnchor.classification.description != "Unknown" && planeAnchor.class

        type = planeAnchor.classification.description
```

```
            // Show the plane type
            planeType.text = "This is \(type)"

            // Show the search button
            search.isHidden = false

        }

}
```

Each `ARPlaneAnchor` has its classification description as specified in the `Plane` class. The classification description will be passed to the `PlaneTypeSearchResultTableViewController` class as the plane type for searching for furniture using `segue.destination`.

When the user clicks the "Search" button, in the `PlaneTypeSearchResultTableViewController` class, the Realm will retrieve the data in the `Furniture.realm` through the `Furniture` class, then the Realm result will be filtered by the plane type and the final result will be displayed.

```
// Read some data from the btrueundled Realm

realmResults = realmFurniture.objects(Furniture.self).filter("type == '\(self.t
```

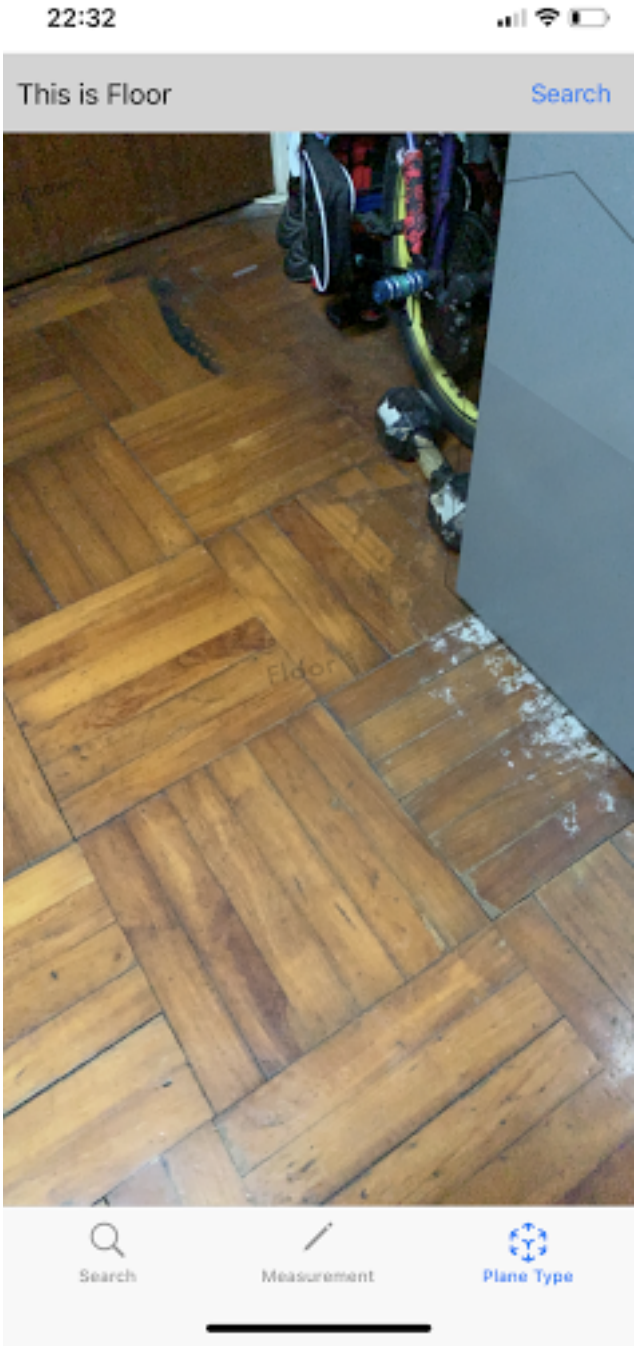


Figure 3.12: The “Search” button is located at the top right corner

3.4 Virtual Content Augmentation

3.4.1 Insertion of Virtual Content into the Camera Scene

Related class of this function:

- `PlaneViewController` class
- `DetailViewController` class

First of all, when the user has already searched for at least one furniture object, he can go to the corresponding detail page of that object. In the detail page, he can click the "Place" button at the bottom of the furniture photo, and in the `DetailViewController` class the furniture ID will be passed to the `PlaneViewController` class using `segue.destination`.

In `Main.StoryBoard`, the `Tap Gesture Recognizer` object is added to the `Plane Scene`. It is also added to the referencing outlet collection of the scene view . A sent action `@IBAction`, which is called `screenTapped`, is created to send an action to the corresponding custom class of the `Plane Scene`, which is the `PlaneViewController` class, as a method. In the `PlaneViewController`, the `sender` of the `@IBAction` is `UITapGestureRecognizer`.

In the `@IBAction`, which is the `screenTapped(_ sender: UITapGestureRecognizer)` method, when the user taps the scene view, the touch location will be got. The hit test of `existingPlaneUsingExtent` will search for any point which lies within the area defined by the plane's center and extent properties in the touch location. If the hit test successfully gets a point, which means that the user has tapped an existing plane area, the anchor as `ARPlaneAnchor` of the hit test result will be got and the function `addFurniture(result: ARHitTestResult)` will be run. In this function, a container called `furnitureScene`, which inherits from the `SCNScene` class, is created to specify the 3D content's location. Here, the location is `art.scnassets/scene/\(sceneName).scn`, in which the scene name is the

furniture name actually, and it is got using the furniture ID passed from the `DetailViewController` class. When the 3D content is got, a variable called `furnitureNode` is created with the `Body` node of the 3D content that the `furnitureScene` got.

Next, the position of the `furnitureNode` will be set by using the above hit test result to get the `worldTransform`, which is a matrix showing the position and orientation of the hit test result relative to the world coordinate system. The position of the `furnitureNode` is the x-value, y-value and z-value of column 3 of the matrix got.

Finally, the `furnitureNode` is ready and it is added as the child node using an instance method, which is `addChildNode(_:)`, so it appears in the scene.

```
func addFurniture(result: ARHitTestResult) {

    let sceneName = realmResults![0].name!

    let furnitureScene = SCNScene(named: "art.scnassets/scene/\(sceneName).scn"

    guard let furnitureNode = furnitureScene?.rootNode.childNode(withName: "Bod

    let planePosition = result.worldTransform.columns.3

    furnitureNode.position = SCNVector3(planePosition.x, planePosition.y, plane

    sceneView.scene.rootNode.addChildNode(furnitureNode)

    virtualObject = furnitureNode

}
```

Figure 3.13: A pillow is placed on the bed

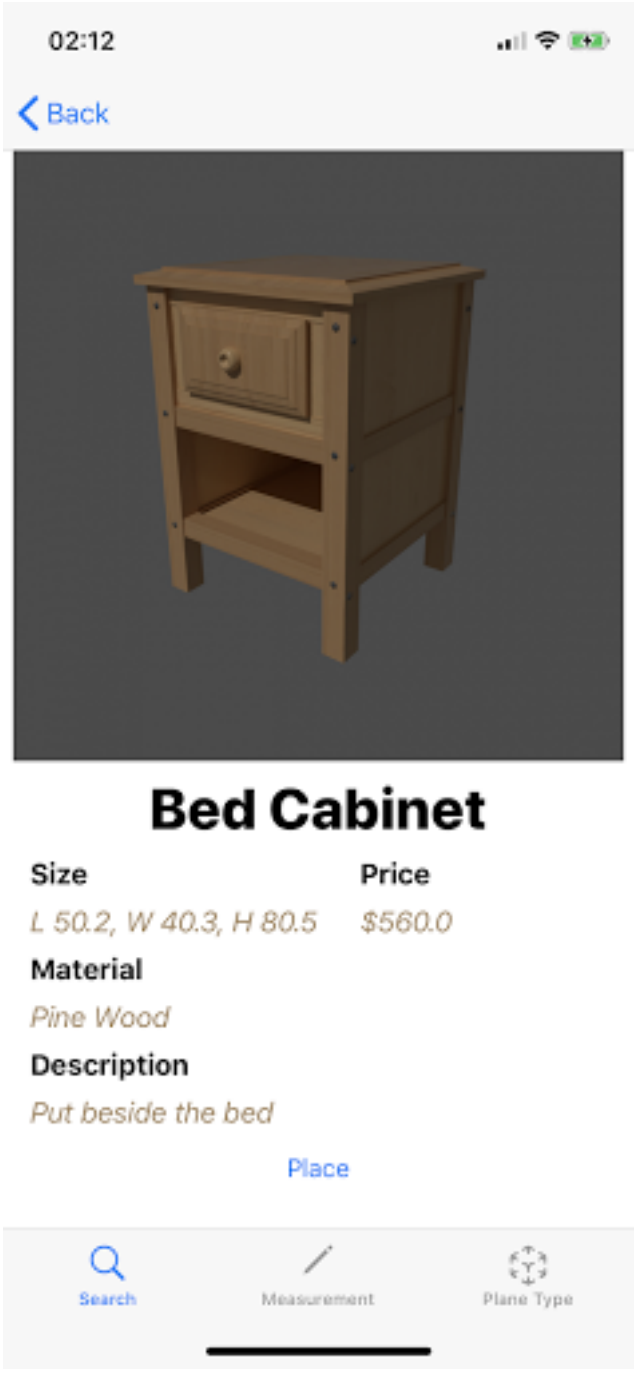


Figure 3.14: The detail page of the bed cabinet

3.4.2 Deletion of Virtual Content from the Camera Scene

Related class of this function:

- `PlaneViewController` class

When the user clicks the "Delete" button, given that he has already placed at least one furniture object in the scene view, all the nodes will be deleted.

First, when the user adds any node into the scene view, I store those nodes into an array of `SCNNode` called `nodes`. Therefore, when he clicks the "Delete" button, the corresponding method will call `removeFromParentNode()` of each member in the array. Then, `removeAll()` is called to remove the array. Finally, all the nodes are removed.

```
for nodeAdded in nodes {
    nodeAdded.removeFromParentNode()
}

nodes.removeAll()
```

3.5 User Interaction with Virtual Content

Related class of this function:

- `PlaneViewController` class

3.5.1 Pan Gesture for Transformation

In `Main.StoryBoard`, the `Pan Gesture Recognizer` object is added to the `Plane Scene`. It is also added to the referencing outlet collection of the scene view . A sent action `@IBAction`, which is called

`screenPanned`, is created to send an action to the corresponding custom class of the `Plane Scene`, which is the `PlaneViewController` class, as a method. In the `PlaneViewController`, the `sender` of the `@IBAction` is `UIPanGestureRecognizer`.

In the `screenPanned(_ sender: UIPanGestureRecognizer)` method, first, the sender's location is got, which means that the location of the continuous movement of the user's finger on the screen is got. Because this gesture recognizer only recognizes continuous gestures, any discrete event, such as a tap, does not report change.

Then, a switch statement is used. When the `sender's state` is `began`, the `selectedNode` is set to be the virtual object's node, which means that the user selects the virtual object as the node to move. On the other hand, when the `sender's state` is `changed`, which means the touches have been recognized as a continuous change, a hit test is called. The type of hit test is set to be the existing plane, so the hit test will search for the AR anchors and real-world objects of the existing plane detected in each hit. Next, `worldTransform` is used. It is a matrix, showing the position and orientation of the hit test result relative to the world coordinate system. The new position is set to be the x-value, y-value and z-value of column 3 of the matrix got. Each node has its `simdPosition`. At last, the `simdPosition` of the virtual object is set to be the new position.

```
@IBAction func screenPanned(_ sender: UIPanGestureRecognizer) {

    // Find the location in the view
    let location = sender.location(in: sceneView)

    switch sender.state {

        case .began:

        // Choose the node to move
        selectedNode = virtualObject
```

```
        case .changed:

        // Move the node based on the real world translation
        guard let result = sceneView.hitTest(location, types: .existingPlane).f

        let transform = result.worldTransform

        let newPosition = SIMD3<Float>(transform.columns.3.x, transform.columns
        selectedNode?.simdPosition = newPosition

        default:

        // Remove the reference to the node
        selectedNode = nil
    }
}
```

Therefore, when user pans the virtual object, the position of the virtual object will change according to the panning direction.

3.5.2 Pinch Gesture for Enlargement or Reduction

In `Main.StoryBoard`, the `Pinch Gesture Recognizer` object is added to the `Plane Scene`. It is also added to the referencing outlet collection of the scene view . A sent action `@IBAction`, which is called `screenPinched`, is created to send an action to the corresponding custom class of the `Plane Scene`, which is the `PlaneViewController` class, as a method. In the `PlaneViewController`, the `sender` of the `@IBAction` is `UIPinchGestureRecognizer`.

In the `screenPinched(_ sender: UIPinchGestureRecognizer)` method, when the `sender's state` is `changed`, which means that the user is pinching against the screen, either outwards or inwards, the virtual object's `simdScale` is got. Each object has its `simdScale`, which is the scale factor. At the same time, it is multiplied with the `sender's scale`, and this

calculation result is set to be the new scale. Then, the `simdScale` of the virtual object is set to be a new scale.

```
@IBAction func screenPinched(_ sender: UIPinchGestureRecognizer) {

    guard let node = virtualObject, sender.state == .changed

    else { return }

    let newScale = node.simdScale * Float(sender.scale)

    node.simdScale = newScale

    sender.scale = 1.0

}
```

Therefore, when the user pinches outwards against the virtual object, it is enlarged by the pinching scale; when the user pinches inwards against the virtual object, it is reduced by the pinching scale.

3.5.3 Rotation Gesture for Rotation

In `Main.StoryBoard`, the `Rotation Gesture Recognizer` object is added to the `Plane Scene`. It is also added to the referencing outlet collection of the scene view . A sent action `@IBAction`, which is called `screenRotated`, is created to send an action to the corresponding custom class of the `Plane Scene`, which is the `PlaneViewController` class, as a method. In the `PlaneViewController`, the `sender` of the `@IBAction` is `UIRotationGestureRecognizer`.

In the `screenRotated(_ sender: UIRotationGestureRecognizer)` method, a switch statement is used. When the `sender's state` is `began`, the `originalRotation` is set to be the Euler angle of the virtual object's node. On the other hand, when the `sender's state` is `changed`, which

means the user is performing rotation gesture against the virtual object, the y-value of the `originalRotation` is subtracted by the float of the `sender's rotation`. Then, the Euler angle is set to be the rotation value after subtraction.

```
@IBAction func screenRotated(_ sender: UIRotationGestureRecognizer) {

    guard let node = virtualObject else { return }

    switch sender.state {

        case .began:
        originalRotation = node.eulerAngles

        case .changed:
        guard var originalRotation = originalRotation else { return }
        originalRotation.y -= Float(sender.rotation)
        node.eulerAngles = originalRotation

        default:
        originalRotation = nil
    }

}
```

Therefore, when user rotates the virtual object, the Euler angle of the virtual object will change according to the rotation gesture.

3.6 Virtual Content Realism

3.6.1 Estimated Ambient Lighting

Related class of this function:

- `PlaneViewController` class

First, in the `viewDidLoad()` method of the `PlaneViewController` class, all the default lights that SceneKit adds would be turned off, including `autoenablesDefaultLighting` and `automaticallyUpdatesLighting`.

Then, the `lightingEnvironment` of the scene is set with the below image. This process is to create a general lighting environment for the scene. This method applies Image Based Lighting (IBL). The concept is using the images taken in the real world to be the light source of the environment. Therefore, the 3D objects will become much more realistic and accurate. Mostly, the environment map is spherical because the sphere is used to surround the 3D objects. Therefore, the environment map can illuminate them. Also, the environment map will be reflected by shiny materials, like metals. But without the environment map, the metals will not be shiny and reflective, and will become faked.

Many types of spherical environment maps can be found online, like Lugher Texture (http://www.lughertexture.com). The below image is an ordinary home lighting environment, and I will use this image as an environment lighting in this project.

```
func setupLights() {

    // Turn off all the default lights SceneKit adds since we are handling it
    sceneView.autoenablesDefaultLighting = false
    sceneView.automaticallyUpdatesLighting = false

    let env = UIImage(named: "./Assets.scnassets/spherical.jpg")
    sceneView.scene.lightingEnvironment.contents = env

}
```

In the `viewWillAppear(_ animated: Bool)` method of the `PlaneViewController` class, the Boolean value `isLightEstimationEnabled` is `true` in the scene configuration, which enables the light estimation.

```
configuration.isLightEstimationEnabled = true
```

Then, the method `renderer(_:updateAtTime:)`, which is called exactly once per frame will adjust the light intensity of the scene. I get the `lightEstimate` of the current frame, and modify the scene's intensity of `lightingEnvironment` to be the ambient intensity of the `lightEstimate`. The estimated ambient intensity is normalized by dividing with 1000.0 because 1000.0 intensity is neutral.

```
func renderer(_ renderer: SCNSceneRenderer, updateAtTime time: TimeInterval) {

    let estimate = sceneView.session.currentFrame?.lightEstimate

    if estimate == nil {
        return
    }

    let intensity = (estimate?.ambientIntensity ?? 0.0) / 1000.0
    sceneView.scene.lightingEnvironment.intensity = intensity

}
```

Change of lighting environment intensity:

Figure 3.15: The nuts of the bed cabinet are made of streaked metal which is shiny, so they reflects the environment map

Figure 3.16: Without environment map, the nuts become black in color and they look faked

Figure 3.17: Using this image to create a lighting environment

Figure 3.18: enter image description here

Figure 3.19: enter image description here

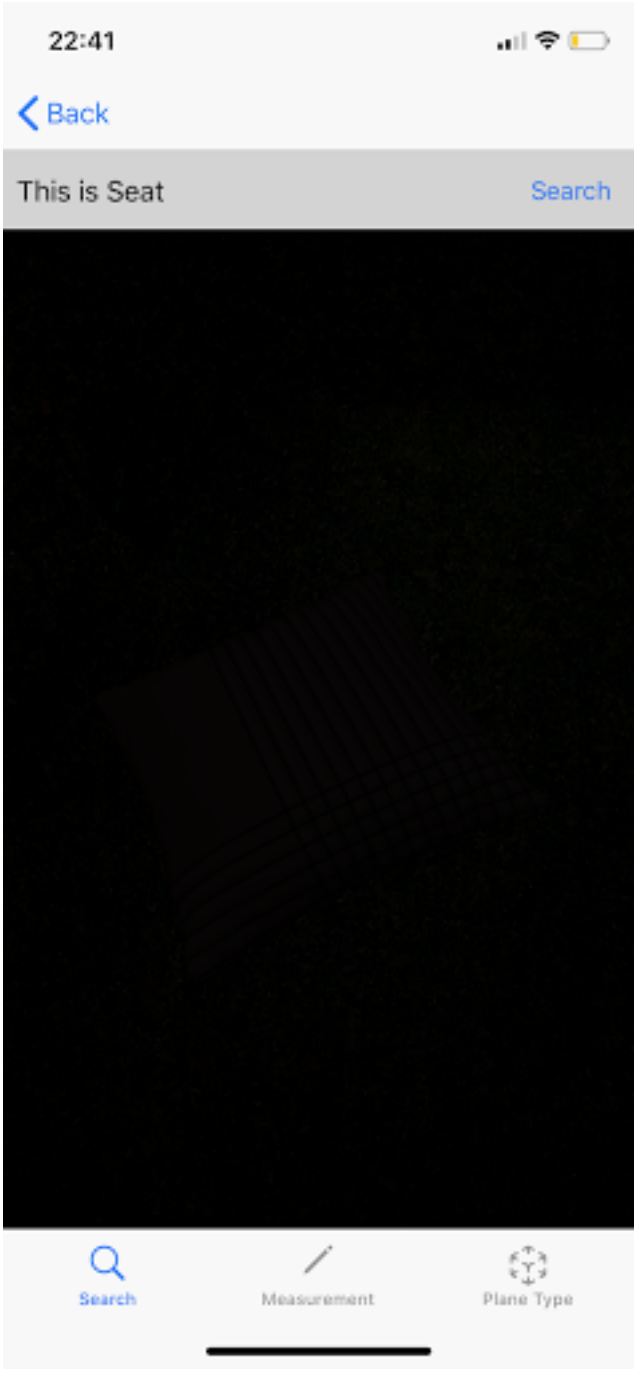


Figure 3.20: enter image description here

3.6.2 Shadows

3.6.2.1 Directional Lighting

Related class of this function:

- `PlaneViewController` class

Shadows can make the 3D objects looks more realistic. I add shadows to the scene by adding directional light. There are mainly 4 commonly used lights in SceneKit:

By adding directional light to the scene, the shadows will be cast on the objects and they will only be cast on the scene geometry, so there will not be "extra shadows" above or below the objects. The *addDireactionalLight()* method is called in the *viewDidLoad()* method of the *PlaneViewController* class. In this method, first, I create a light and set its type to *.directional* because only directional and spot lights can cast shadows and the directional lights are more natural than spot lights. The below images will prove it. The added light does not have any effect on the lighting environment, so I set the *intensity* to zero. Then, I set *castsShadow* to *true* and *shadowMode* to *.deferred*, so that the shadows are not applied when rendering the objects. Instead, it would be applied after finishing the rendering.

Next, I create a black color with 50% opacity and set it as the *shadowColor*. This will make the casting shadows look more grey and realistic. I also increase its *shadowSampleCount* to create smoother shadows with and higher

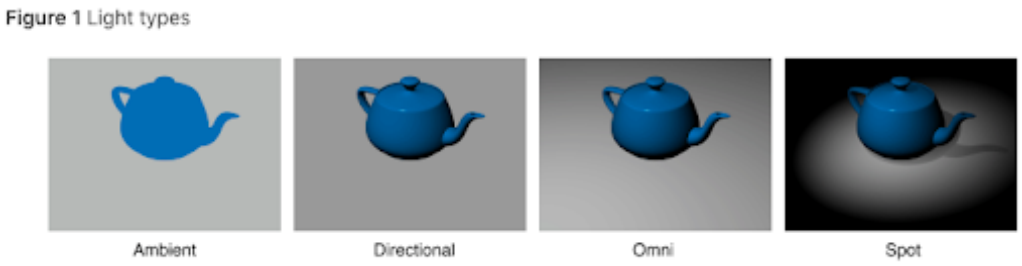


Figure 3.21: Common light types in SceneKit

resolution. Finally, I create a node called “directionalLightNode”, attach the above light to it, and rotate it so that it is facing the floor at a slightly downward angle. The “directionalLightNode” is ready and it is added as the child node using an instance method, which is *addChildNode(_:)*, so it appears in the scene. The images below will show the result.

```
func addDirectionalLight() {

    // Create a directional light
    let directionalLight = SCNLight()

    directionalLight.type = .directional
    directionalLight.intensity = 0
    directionalLight.castsShadow = true

    directionalLight.shadowMode = .deferred
    directionalLight.shadowColor = UIColor(red: 0, green: 0, blue: 0, alpha: 0.
    directionalLight.shadowSampleCount = 10

    // Create a directional light node
    let directionalLightNode = SCNNode()
    directionalLightNode.light = directionalLight
    directionalLightNode.rotation = SCNVector4Make(1, 0, 0, -Float.pi / 3)

    sceneView.scene.rootNode.addChildNode(directionalLightNode)

}
```

Result:

1. The spot light does not provide a natural view.
2. The bed cabinet with directional lighting will have a shadow inside the drawer. In comparison, the bed cabinet without any lighting does not have a shadow inside the drawer.

3.6.2.2 Shadow Baking uisng Blender

Normally, there are shadows below each objects in the daily life. If there is not any shadow below an object, the object will be like it is floating in the air. I used Blender, which is a free 3D computer graphics software, to bake a shadow for each furniture object.

The below image shows the Blender software.

According to the above image, in the left editor, which is the default scene, I create a plane below the object first. The bottom editor is UV/Image Editor, where I can render a shadow on it. And I save the rendering result as an image:

But I only want the shadow on the plane. The right editor is node editor, where I can set the image texture when I bake the shadows. I set the image texture to be black color.

After I got the shadow image, I go to Xcode SceneKit Editor and create a plane below the 3D object. Then, I go to the material inspector of that plane, and set its diffuse map to the shadow image. And I change the opacity of the shadow image to 0.5-0.7 to make it semi-transparent.

Result:

Figure 3.22: With spot light: the bed cabinet is less natural and its surface has higher contrast

Figure 3.23: With spot light: the bed cabinet's surface becomes even darker with dimmer ambient lighting in the real environment

Figure 3.24: With directional lighting

Figure 3.25: Without any lighting

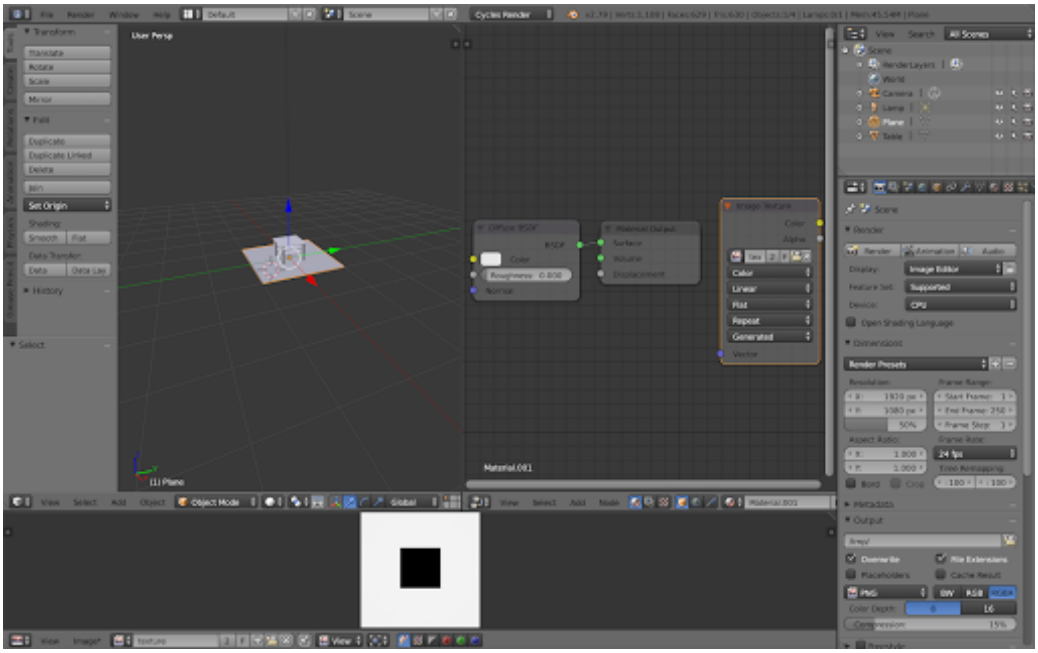


Figure 3.26: Blender software

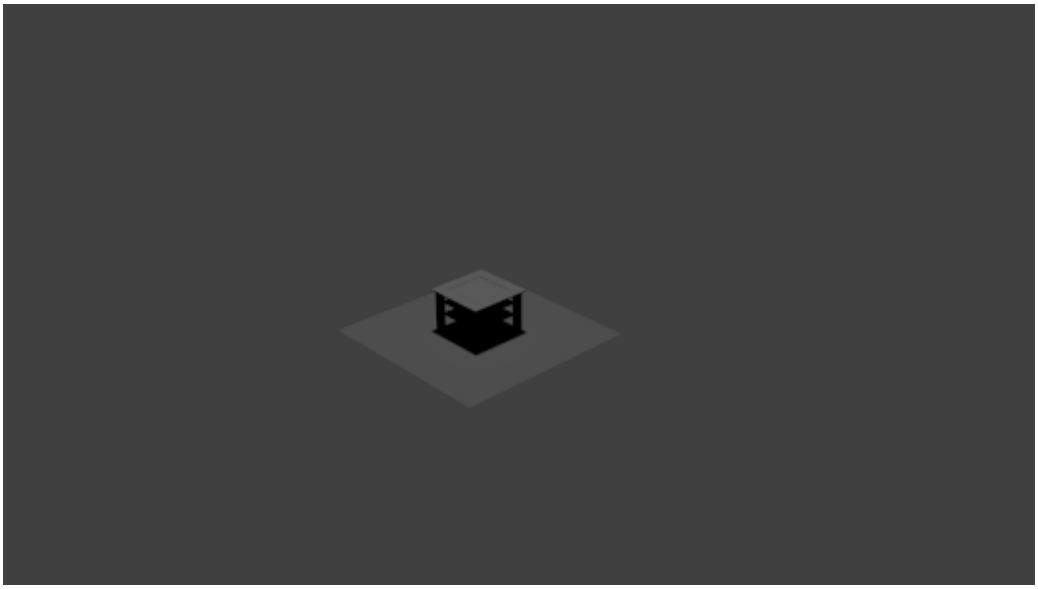


Figure 3.27: Rendering shadow

Figure 3.28: The shadow baked by Blender

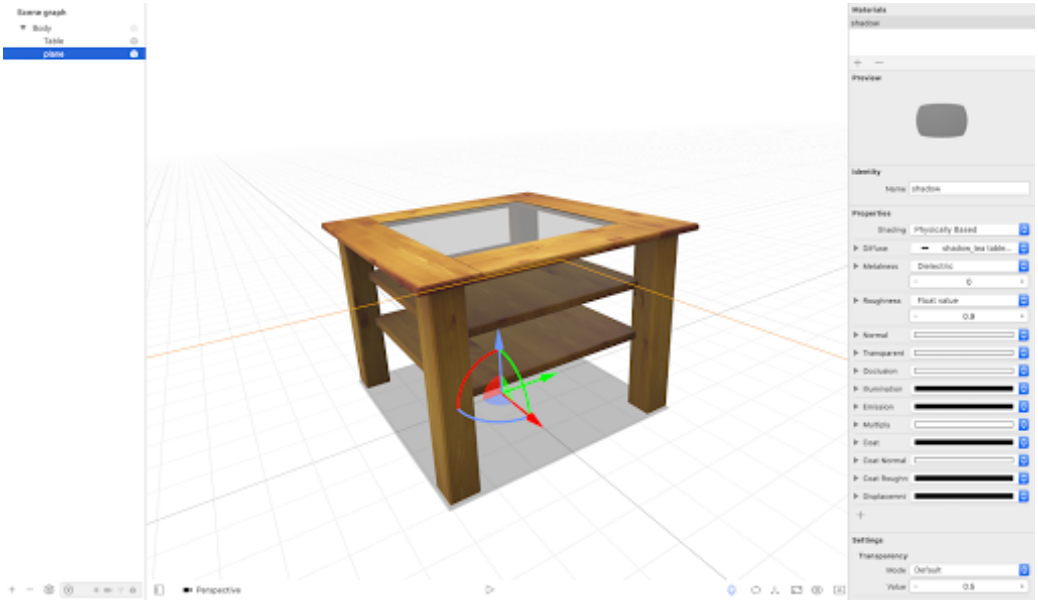


Figure 3.29: Adding a shadow below the tea table

Figure 3.30: With shadow below

Figure 3.31: Without shadow below

3.6.3 Physically Based Rendering

Physically Based Rendering (PBR) is a collection of render techniques to generate objects which look more realistic. When we add material to an object using PBR, we usually provide the following information as parameters:

- Albedo
  It is also known as the diffuse map. It is one of the factors that determines the amount of light reflected is material because it measures how much light that hits a surface is reflected without being absorbed. The light reflected would become visible.

- Roughness
  It describes how rough the material will be. The rougher the material, the dimmer the reflection, vice versa.

- Metalness
  It describes how metallic the material will be. It is an important parameter in different rendering systems. Firstly, metals (conductors) are much more reflective than insulators (non-conductors). The high reflectivity prevents most light from reaching the interior and scattering, so metals look shiny in general. The more metallic the material, the shinier the reflection, vice versa. In SceneKit, the meatless property can be set with a color vale. Lower values, which means darker colors, will make the material to less metallic, such as wood. Higher values, which means brighter colors, will cause the surface to appear more metallic, like iron. The below image shows the streaked metal material of the nuts. The metalness property is white color, so the material is very shiny.

- Normal
  It describes the nominal orientation of a surface at each point for use in lighting. In general, we use a texture image as a normal map for the normal property. When SceneKit uses the texture image for the normal property, it treats the R, G, and B components of each pixel of the image as the X, Y, and Z components of a surface normal vector. Because a normal map can store much more detailed surface information

than a geometry, we can set a texture image to the normal property to simulate rough surfaces such as stone and wood by adding protrusion in the normal map.

In Xcode SceneKit Editor, we can select a specific node and set the node's material in the material inspector. We can select an image for each property, such as diffuse, metalness, roughness, normal, etc.

Figure 3.32: A diffuse map of bamboo wood

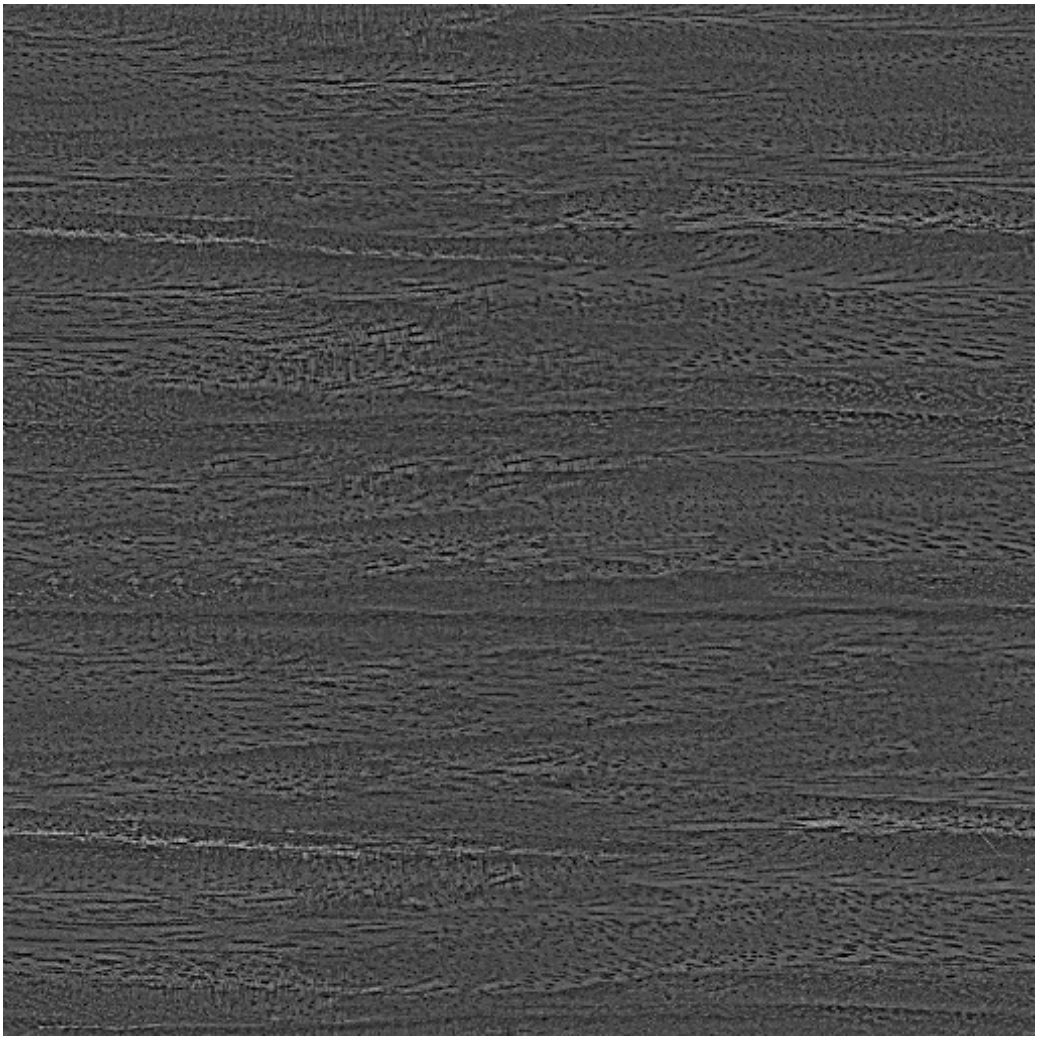


Figure 3.33: A roughness map of bamboo wood

Figure 3.34: The nuts of the bed cabinet are using streaked metal as the material

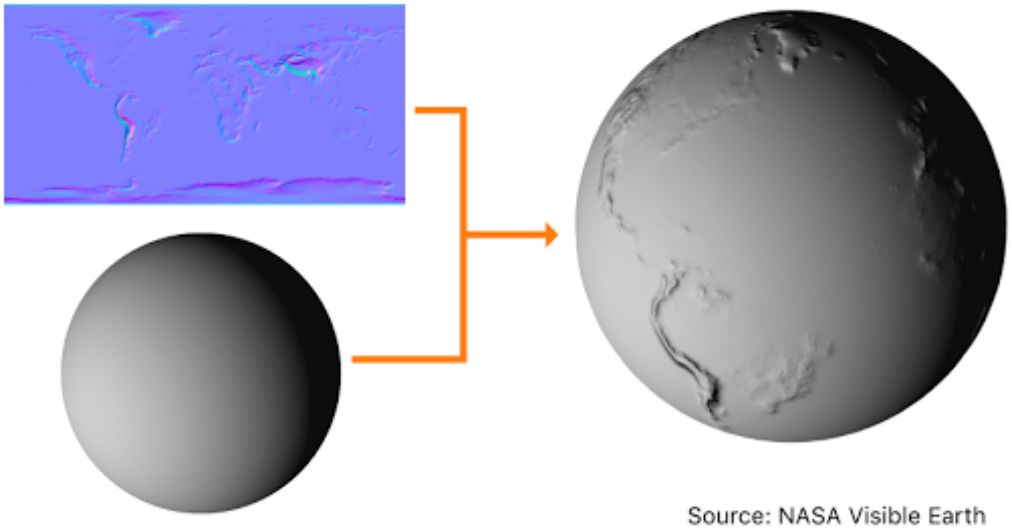


Figure 3.35: Adding protrusion by a normal map to the diffuse content

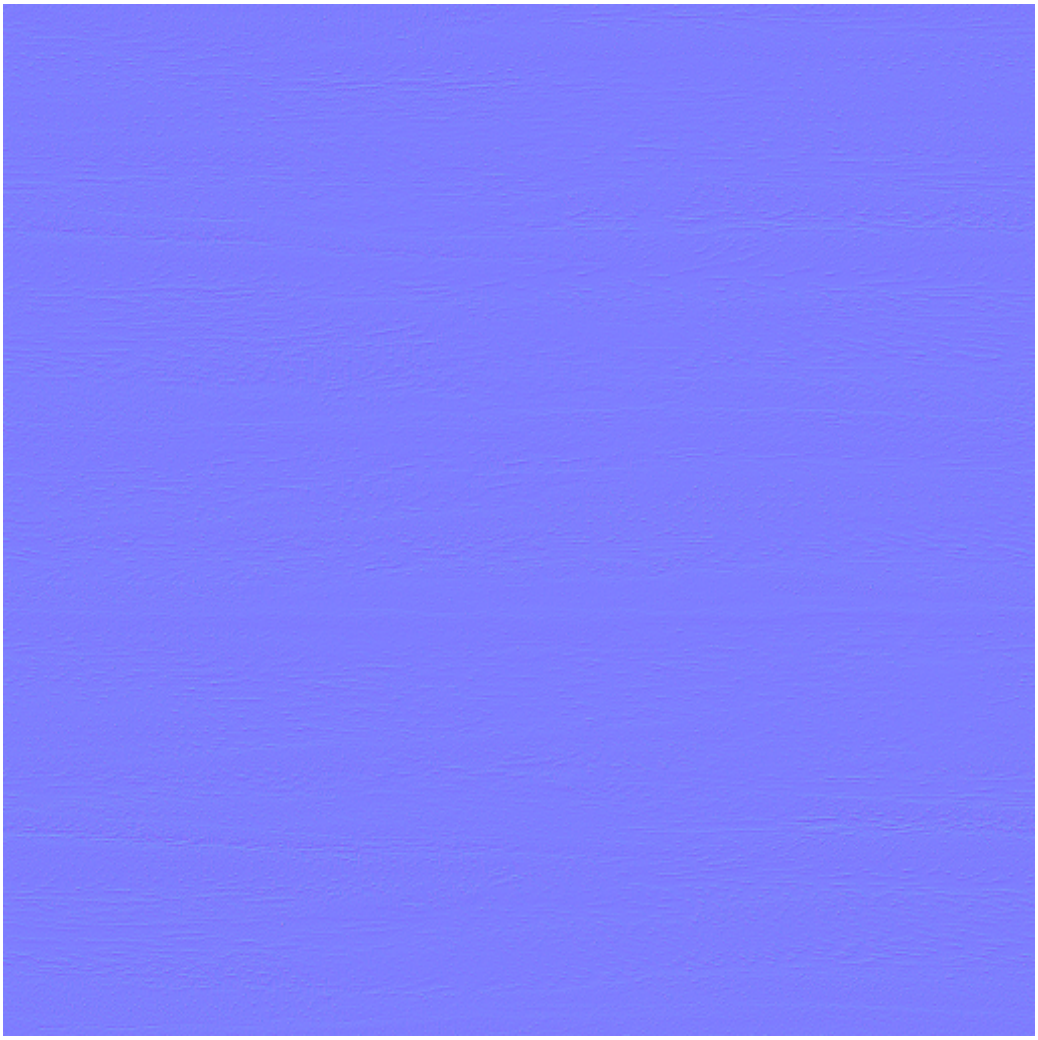


Figure 3.36: A normal map of bamboo wood with some protrusion on it

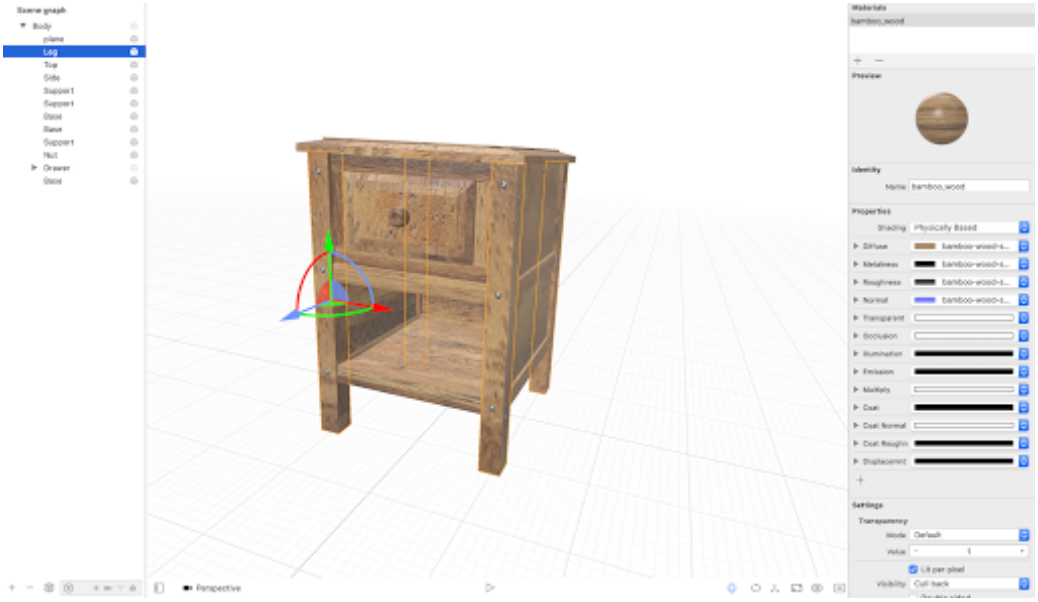


Figure 3.37: The material inspector of the bed cabinet is mainly using the PBR model of bamboo wood

Chapter 4

Results

4.1 Impact of Virtual Content Realism

The three components of virtual content realism are estimated ambient lighting, shadows and physically based rendering, which can provide a more realistic view for the users. The estimated ambient lighting can apply to an object according to the actual change of light. The shadows can apply to the object at its body by directional lighting and below it by the image of baked shadow. The physical based rendering can render different materials for the objects. It textures different materials, so the materials will be similar to what we see in real life.

4.2 Impact of Measurement

The measurement searching is able to allow the users to search for the furniture objects which are suitable for their measurements. It is able to allow the users to measure for more than one distance for each search, and the measurements can be stored temporarily in the Realm. The measuring path is dynamic, which can change size and direction according to the user's finger movement. It is more convenient than asking the user to point the start point and the endpoint, because the user needs to walk towards the endpoint

and point it. But if we use the measuring path, moving the device has been already sufficient to do the measurement.

Chapter 5

Discussions and Conclusions

5.1 Difficulties and Limitations

During the developement of Funiture Application, I have a deep revision about the pan and rotation gesture. I have tried what Apple Developer provided which is `raycast(_:)`, but it is complicated. According to the Apple Developer, ray casting a enables one-time hit test for pan gesture, so we do not need refined the position results over time for the continuous requests. Also, ray casting gives us the orientation information about the surface at a given screen point. Therefore, we do not need to keep on subtracting the gesture's rotation from the current object rotation for rotation gesture. The `raycast(_:)` method can provide a smoother interaction with the objects, but due to time limit of this project, I use the ordinary method to perform the pan and rotation gesture activities.

5.2 Further Development

5.2.1 Scanning and Detecting 3D Objects

Apple Developer provides us a resource to scan and record the spatial features of the real-world objects, and the records can be saved as an `.arobject` file.

It can be used as an after-sale service of a furniture shop. For example, after the user buys a denim chair from the shop and put it at home, he wants to know the information of this denim chair, like the maintenance period, he can scan the denim chair. As the denim chair has a record as a reference object before he buys it, so when he scans the chair, the application can compare the scanning object with the reference object, and find out the related information.

5.2.2 User Community

A user community can be a platform for different users to share the furniture objects. They can make comments on other shared posts too.

5.2.3 User Favorite

In the first progress report of this project, I wanted to add a page for the users to save their favorite furniture objects. However, due to the time limit, I decided to drop this function. I still believe that it can help the users to record what they like, and the preference information can provide great help for search suggestions by the application.

5.2.4 Material Customization

To make the application more interactive, the material customization can be added to the functions. The users can change the material of an object. They can also adjust different properties, like adjusting the reflection, metalness, etc.

Appendix 1: System Setup

This application can only run on iOS devices.

1. Download Xcode.
2. Download the project folder.
3. Open the `.xcworkspace` file inside the project folder.
4. Change the signing to your team and bundle identifier.
5. Connect to an iOS device that have an A9 or later processor and version iOS 12.0 or above. ARKit is not available in iOS Simulator.
6. Click “Run”.

References

[1] Apple Developer. (n.d.). Tracking and Visualizing Planes. Retrieved March 2020, from https://developer.apple.com/documentation/arkit/world_tracking/tracking_

[2] Herrera, E. (2017, September 4). Build a realtime measuring app with ARKit. Retrieved March 2020, from https://pusher.com/tutorials/realtime-measuring-arkit

[3] Dawson, M. (2017, June 18). ARKit by Example - Part 4: Realism - Lighting & PBR. Retrieved March 2020, from https://blog.markdaws.net/arkit-by-example-part-4-realism-lighting-pbr-b9a0bedb013e

[4] Korouei, K. (2019, November 26). Realistic Rendering of 3D Photogrammetry Model in ARKit. Retrieved April 2020, from https://medium.com/appcoda-tutorials/realistic-rendering-of-3d-photogrammetry-model-in-arkit-a2ebec48e923

[5] Wilson, J. (2018, November 1). Physically-Based Rendering, And You Can Too! Retrieved April 2020, from https://marmoset.co/posts/physically-based-rendering-and-you-can-too/#albedo

[6] Russell, J. (2018, November 1). Basic Theory of Physically-Based Rendering. Retrieved April 2020, from https://marmoset.co/posts/basic-theory-of-physically-based-rendering/

[7] Ibrahim, M. (2018, July 3). An Introduction to ARKit 2 - Object Scanning. Retrieved April 2020, from https://blog.usejournal.com/an-introduction-to-arkit-2-object-scanning-68963b9be43a

[8] Assouline, A. (2016, August 9). Amazing Physically Based Ren-

dering Using the New iOS 10 SceneKit. Retrieved April 2020, from https://medium.com/(**???**)