直击苹果 ARKit 技术 - 简书

作者 程序员钙片吃多了 关注

2017.08.10 20:05* 字数 9934 阅读 1189喜欢 59

苹果在 WWDC2017 中推出了 ARKit，通过这个新框架可以看出苹果未来会在 AR 方向不断发展，本着学习兴趣，对此项新技术进行了学习，并在团队进行了一次分享，利用业余时间把几周前分享的内容整理成文档供大家交流学习。

本文并不是简单的介绍 ARKit 中的 API 如何使用，而是在介绍 ARKit API 的同时附上了一些理论知识，所以有些内容可能会不太容易理解。笔者能力有限，文章若有误请指出。团队分享时进行了一次直播，若文中有不理解的内容，欢迎观看下录播：ARKit 分享直播 。本文也附上分享的 PPT 地址：ARKit-keynote。

一、什么是 AR？

AR 全称 Augmented Reality(增强现实)，是一种在摄像机捕捉到的真实世界中加入计算机程序创造的虚拟世界的技术。下图是一个简单的 AR 的 Demo：

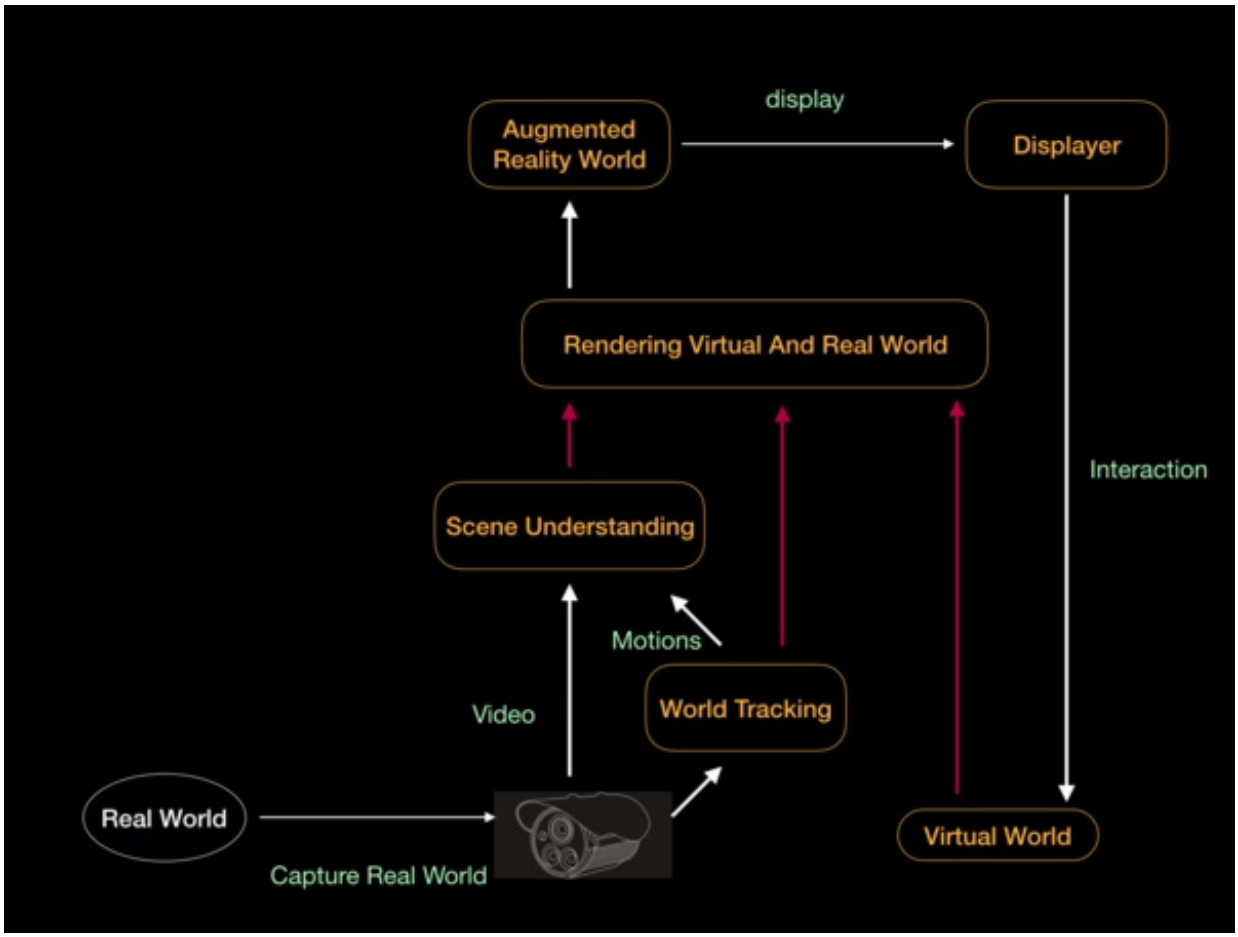
ARChair.gif

ARChair.gif

在上图中，椅子是计算机程序创建的虚拟世界的物体，而背景则是摄像机捕捉到的真实世界，AR 系统将两者结合在一起。我们从上图可以窥探出 AR 系统由以下几个基础部分组成：

1. 捕捉真实世界：上图中的背景就是真实世界，一般由摄像机完成。
2. 虚拟世界：例如上图中的椅子就是虚拟世界中的一个物体模型。当然，可以有很多物体模型，从而组成一个复杂的虚拟世界。
3. 虚拟世界与现实世界相结合：将虚拟世界渲染到捕捉到的真实世界中。
4. 世界追踪：当真实世界变化时(如上图中移动摄像机)，要能追踪到当前摄像机相对于初始时的位置、角度变化信息，以便实时渲染出虚拟世界相对于现实世界的位置和角度。
5. 场景解析：例如上图中可以看出椅子是放在地面上的，这个地面其实是 AR 系统检测出来的。
6. 与虚拟世界互动：例如上图中缩放、拖动椅子。(其实也属于场景解析的范畴)

根据上面的描述，我们可以得出 AR 系统的大致结构图：



ARKitSystem.png

Note：这里只介绍基于计算机显示器的 AR 系统实现方案，此外还有光学透视式和视频透视式方案。可参考增强现实-组成形式。

二、ARKit 简介

ARKitLogo.png

ARKit 是苹果 WWDC2017 中发布的用于开发iOS平台 AR 功能的框架。ARKit 为上一节中提到的 AR 系统架构中各个部分都提供了实现方案，并且为开发者提供了简单便捷的 API，使得开发者更加快捷的开发 AR 功能。

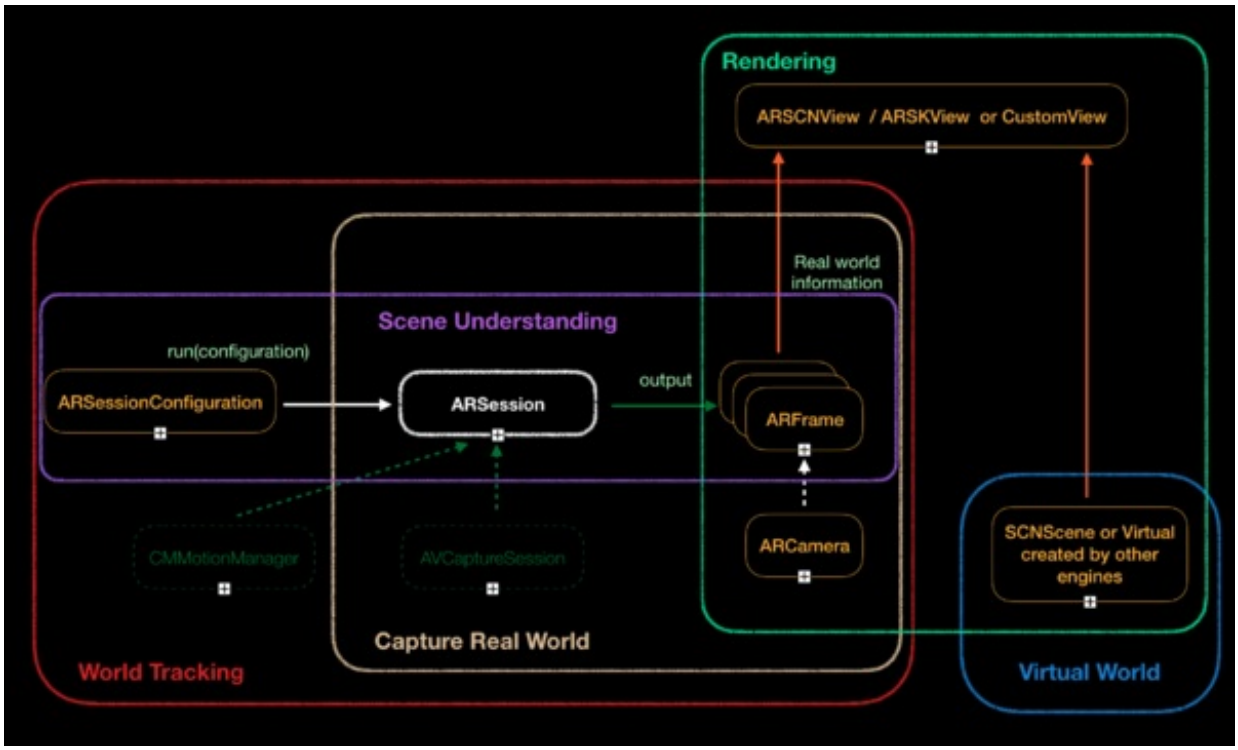
ARKit 的使用需要一定的软硬件设施：

- 软件：
  - 开发工具：Xcode9
  - iOS11
  - MacOS 10.12.4 及以上版本(为了支持 Xcode9)
- 硬件：
  - 处理器为 A9 及以上的 iPhone 或 iPad 设备(iPhone 6s 为 A9 处理器)

下面几节中，我们将逐步介绍 ARKit 中 AR 系统的各个组成部分。

三、ARKit 架构

ARKit 定义了一套简单易用的 API，API 中引入了多个类，为了更加清晰的理解 ARKit，我们从 AR 系统组成的角度对 ARKit API 进行了分类，如下图：



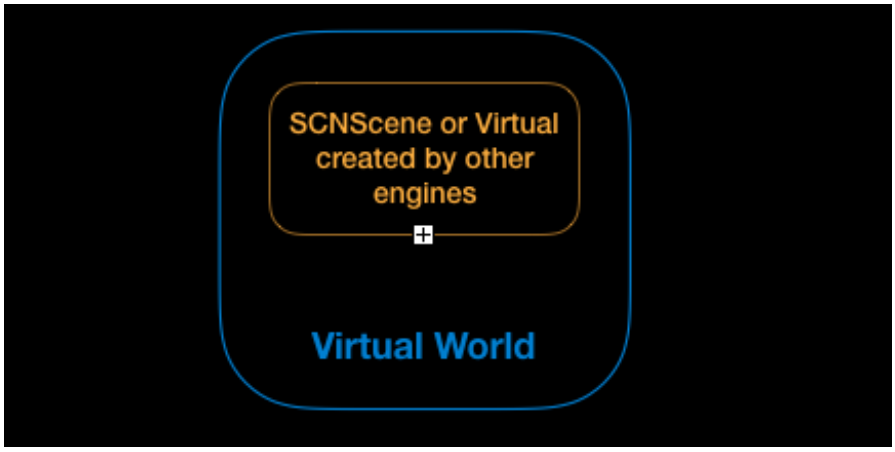
ARKitArchitecture

上图列出了 ARKit API 中的几个主要的类，如 ARSession、ARSessionConfiguration、ARFrame、ARCamera 等。并依据各个类的功能进行了模块划分：红色(World Tracking)、蓝色(Virtual World)、土色(Capture Real Wrold)、紫色(Scene Understanding)、绿色(Rendering)。

对于上图，ARSession 是核心整个ARKit系统的核心，ARSession 实现了世界追踪、场景解析等重要功能。而 ARFrame 中包含有 ARSession 输出的所有信息，是渲染的关键数据来源。虽然 ARKit 提供的 API 较为简单，但看到上面整个框架后，对于初识整个体系的开发者来说，还是会觉着有些庞大。没关系，后面几节会对每个模块进行单独的介绍，当读完最后时，再回头来看这个架构图，或许会更加明了一些。

四、构建虚拟世界

我们将 Virtual World 从 ARKit 架构图中抽出：



VirtualWorld.png

ARKit 本身并不提供创建虚拟世界的引擎，而是使用其他 3D/2D 引擎进行创建虚拟世界。iOS 系统上可使用的引擎主要有：

- Apple 3D Framework - SceneKit.
- Apple 2D Framework - SpriteKit.
- Apple GPU-accelerated 3D graphics Engine - Metal.
- OpenGl
- Unity3D
- Unreal Engine

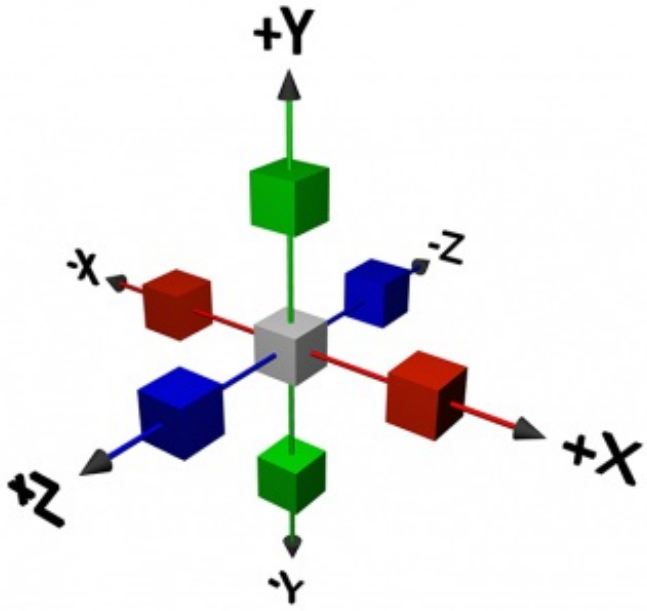
ARKit 并没有明确要求开发者使用哪种方式构建虚拟世界，开发者可以利用 ARKit 输出的真实世界、世界追踪以及场景解析的信息(存在于 ARFrame 中)，自己将通过图形引擎创建的虚拟世界渲染到真实世界中。值得一提的是，ARKit 提供了 ARSCNView 类，该类基于 SceneKit 为 3D 虚拟世界渲染到真实世界提供了非常简单的 API，关于 ARSCNView 会在最后的渲染部分进行介绍，所以下面我们来介绍下 SceneKit。

SceneKit 简介

这里不对 SceneKit 进行深入探讨，只简单介绍下基础概念。读者只需要理解 SceneKit 里虚拟世界的构成就可以了。

SceneKit 的坐标系

我们知道 UIKit 使用一个包含有 x 和 y 信息的 CGPoint 来表示一个点的位置，但是在 3D 系统中，需要一个 z 参数来描述物体在空间中的深度，SceneKit 的坐标系可以参考下图：



SceneKitCoordinate.png

这个三维坐标系中，表示一个点的位置需要使用(x,y,z)坐标表示。红色方块位于 x 轴，绿色方块位于 y 轴，蓝色方块

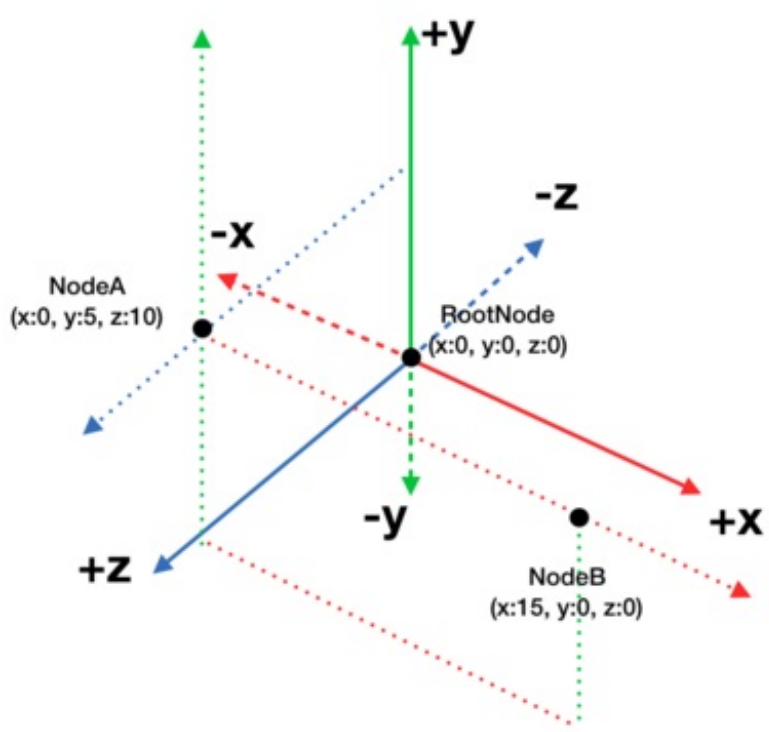
位于 z 轴，灰色方块位于原点。在 SceneKit 中我们可以这样创建一个三维坐标：

```
let position = SCNVector3(x: 0, y: 5, z:
10)
```

SceneKit 中的场景和节点

我们可以将 SceneKit 中的场景(SCNScene)想象为一个虚拟的 3D 空间，然后可以将一个个的节点(SCNNode)添加到场景中。SCNScene 中有唯一一个根节点(坐标是(x:0, y:0, z:0))，除了根节点外，所有添加到 SCNScene 中的节点都需要一个父节点。

下图中位于坐标系中心的就是根节点，此外还有添加的两个节点 NodeA 和 NodeB，其中 NodeA 的父节点是根节点，NodeB 的父节点是 NodeA：



SceneKitNode.png

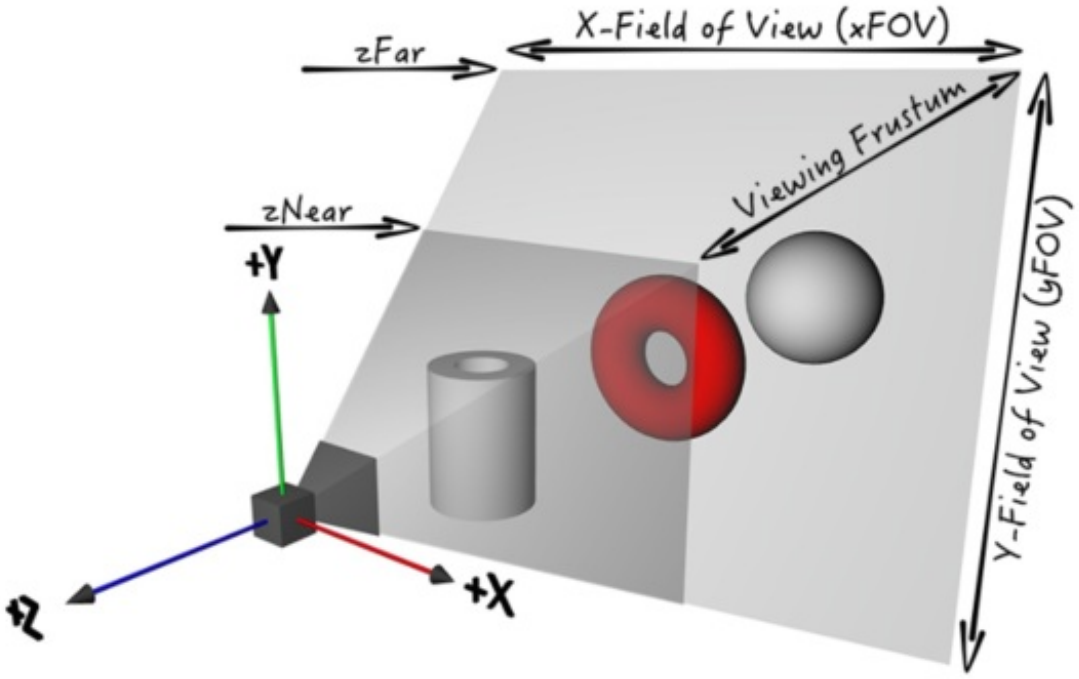
SCNScene 中的节点加入时可以指定一个三维坐标(默认为(x:0, y:0, z:0))，这个坐标是相对于其父节点的位置。这里说明两个概念：

- 本地坐标系：以场景中的某节点(非根节点)为原点建立的三维坐标系
- 世界坐标系：以根节点为原点创建的三维坐标系称为世界坐标系。

上图中我们可以看到 NodeA 的坐标是相对于世界坐标系(由于 NodeA 的父节点是根节点)的位置，而 NodeB 的坐标代表了 NodeB 在 NodeA 的本地坐标系位置(NodeB 的父节点是 NodeA)。

SceneKit 中的摄像机

有了 SCNScene 和 SCNNode 后，我们还需要一个摄像机(SCNCamera)来决定我们可以看到场景中的哪一块区域(就好比现实世界中有了各种物体，但还需要人的眼睛才能看到物体)。摄像机在 SCNScene 的工作模式如下图：



SceneKitCamera.png

上图中包含以下几点信息：

- SceneKit 中 SCNCamera 拍摄的方向始终为 z 轴负方向。
- 视野(Field of View)是摄像机的可视区域的极限角度。角度越小，视野越窄，反之，角度越大，视野越宽。
- 视锥体(Viewing Frustum)决定着摄像头可视区域的深度(z 轴表示深度)。任何不在这个区域内的物体将被剪裁掉(离摄像头太近或者太远)，不会显示在最终的画面中。

在 SceneKit 中我们可以使用如下方式创建一个摄像机：

```
let scene = SCNScene()
let cameraNode = SCNNode()
let camera = SCNCamera()
cameraNode.camera = camera
cameraNode.position = SCNVector3(x: 0, y: 0, z:
0)
scene.rootNode.addChildNode(cameraNode)
```

SCNView

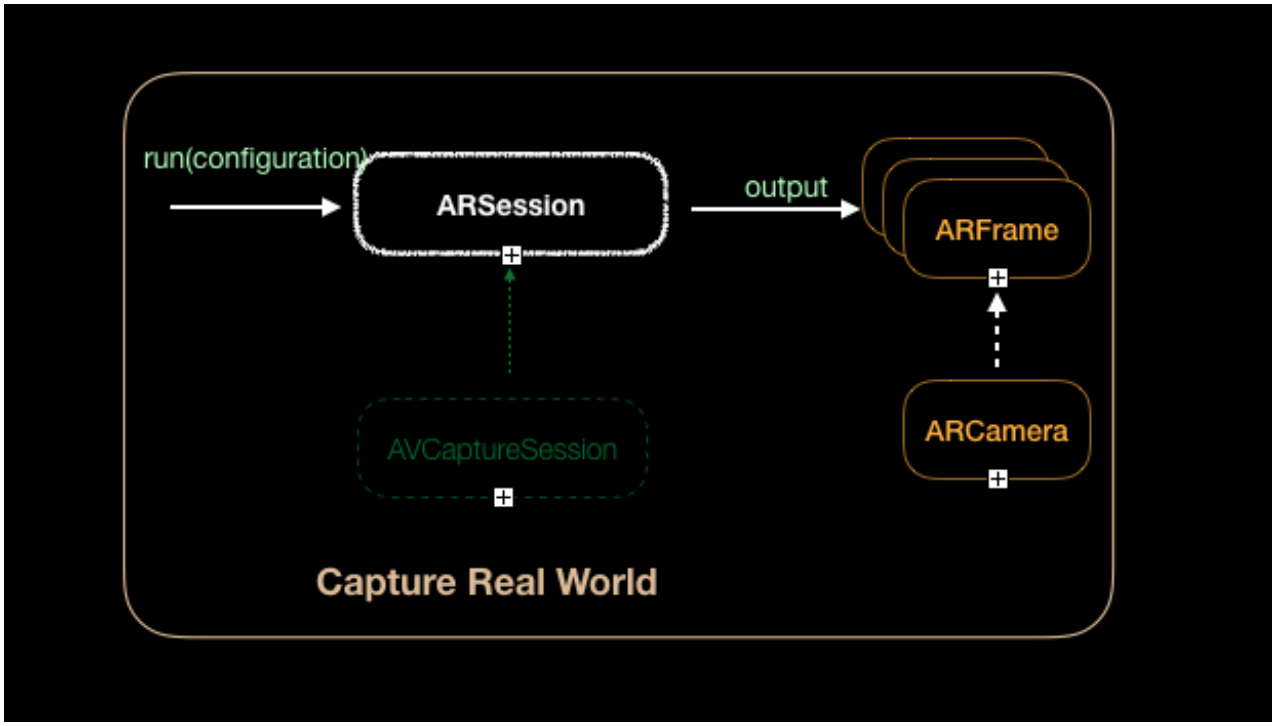
最后，我们需要一个 View 来将 SCNScene 中的内容渲染到显示屏幕上，这个工作由 SCNView 完成。这一步其实很简单，只需要创建一个 SCNView 实例，然后将 SCNView 的 scene 属性设置为刚刚创建的 SCNScene，然后将

SCNView 添加到 UIKit 的 view 或 window 上即可。示例代码如下：

```
let scnView = SCNView()
scnView.scene = scene
vc.view.addSubview(scnView)
scnView.frame =
vc.view.bounds
```

五、捕捉真实世界(Capture Real World)

捕捉真实世界就是为了将我们现实世界的场景作为 ARKit 显示场景的背景。为了方便阅读，我们首先将 Capture Real World 从 ARKit 架构图中抽取出：



CaptureRealWorld.png

ARSession

如果我们想要使用 ARKit，我们必须要创建一个 ARSession 对象并运行 ARSession。基本步骤如下：

```
let configuration =
ARWorldTrackingSessionConfiguration()

let session = ARSession()

session.run(configuration)
```

从上面的代码看，运行一个 ARSession 的过程是很简单的，那么 ARSession 的底层如何捕捉现实世界场景的呢？

- 首先，ARSession 底层使用了 AVCaputreSession 来获取摄像机拍摄的视频(一帧一帧的图像序列)。
- 然后，ARSession 将获取的图像序列进行处理，最后输出 ARFrame，ARFrame 中就包含有现实世界场景的所有信息。

ARFrame

从上一步骤得知 ARFrame 中包含了现实世界场景的所有信息，那么 ARFrame 中与现实世界场景有关的信息有哪些？

`var capturedImage:`
- `CVPixelBuffer`
  该属性是摄像机捕捉到的图像信息，就是构成我们现实世界场景中的一帧图像。顺便说明下，对于摄像机捕捉的每一帧图像，都会生成一个 ARFrame。

`var timestamp:`
- `TimeInterval`
  该属性是摄像机捕捉到的对应于 capturedImage 的一帧图像的时间。

`var camera:`
- `ARCamera`
  获取现实世界的相机信息。详细介绍见下。

ARCamera

ARCamera 是 ARFrame 中的一个属性，之因为单独拿出来说，是因为这里有必要介绍下相机的一些特性，ARCamera 中与现实世界场景有关的信息有两个：

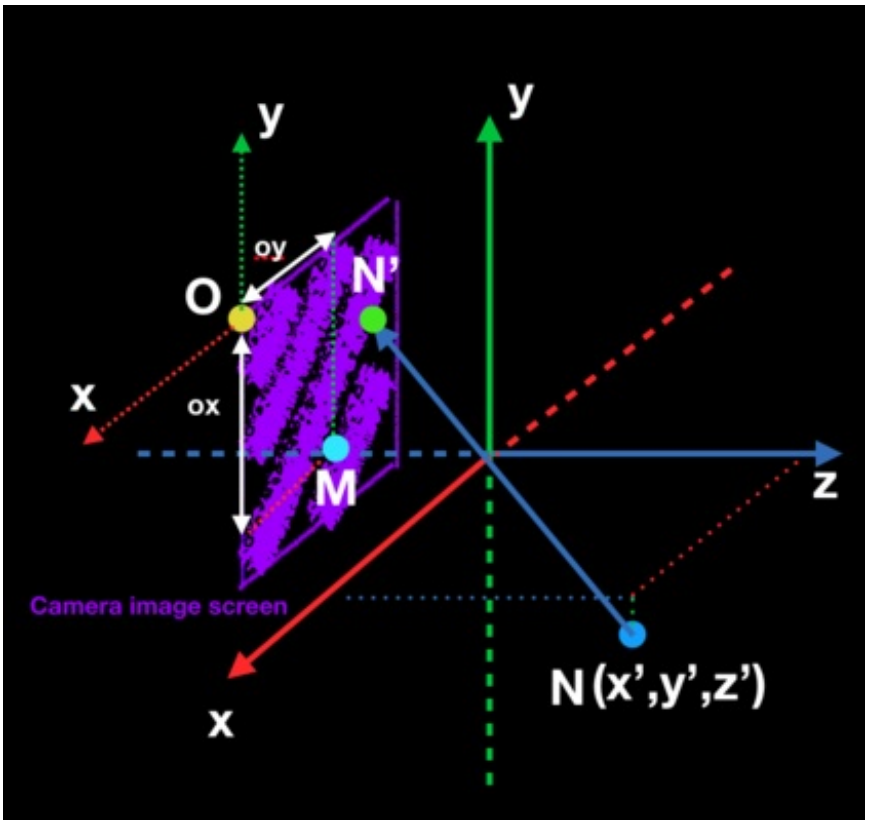
`var imageResolution:`
- `CGSize`
  该属性表示了相机捕捉到的图像的长度和宽度(以像素为单位)，可以理解成捕捉到的图像的分辨率。

`var intrinsics:`
- `matrix_float3x3`
  intrinsics 是一个 3x3 矩阵，这个矩阵将我们现实世界中三维坐标系的点映射到相机捕捉的图像中。有兴趣可看下面的详述。

Intrinsic Matrix

Intrinsic Matrix 是相机的一个固有属性，也就是说每个相机都会有 Intrinsic Matrix，因为所有的相机都需要将现实世界中三维空间的点映射到捕捉的图像中二维空间的点。

那么这个矩阵是如何工作的呢？我们先来看一个图片：



IntrinsicMatrix.png

上图包含如下基本信息：

- 一个三维坐标系(红色 x 轴，绿色 y 轴，蓝色 z 轴)。
- 空间中的一个点(蓝色的点 N，坐标为(x', y', z'))。
- 相机的成像平面(紫色的平行四边形)
- 成像平面与 z 轴的交点(点 M)
- 成像平面的原点(黄色的点 O)，也就是捕捉的二维图像的二维坐标系的原点。

现在我们需要将三维空间的点(x', y', z')映射到成像平面中的一个点(N')。下面我们来看下映射过程。

Intrinsic Matrix 一般是下图所示的样子：

$$K = \begin{bmatrix} fx & 0 & ox \\ 0 & fy & oy \\ 0 & 0 & 1 \end{bmatrix}$$

IntrinsicMatrixValue.png

上图中，fx 和 fy 是摄像机镜头的焦距，这里不做深究，ox 和 oy 则是点 M(成像平面与 z 轴交点)相对于点 O(成像平面二维坐标系原点)的 x 与 y 方向的偏移。

下图展示了利用 Intrinsic Matrix 将 N 映射 N' 的过程：

$$\begin{bmatrix} fx & 0 & ox \\ 0 & fy & oy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x' * fx + ox * z' \\ y' * fy + oy * z' \\ z' \end{bmatrix} \xrightarrow{/z'} \begin{bmatrix} (x' * fx)/z' + ox \\ (y' * fy)/z' + oy \\ 1 \end{bmatrix}$$

IntrinsicMatrixMap.png

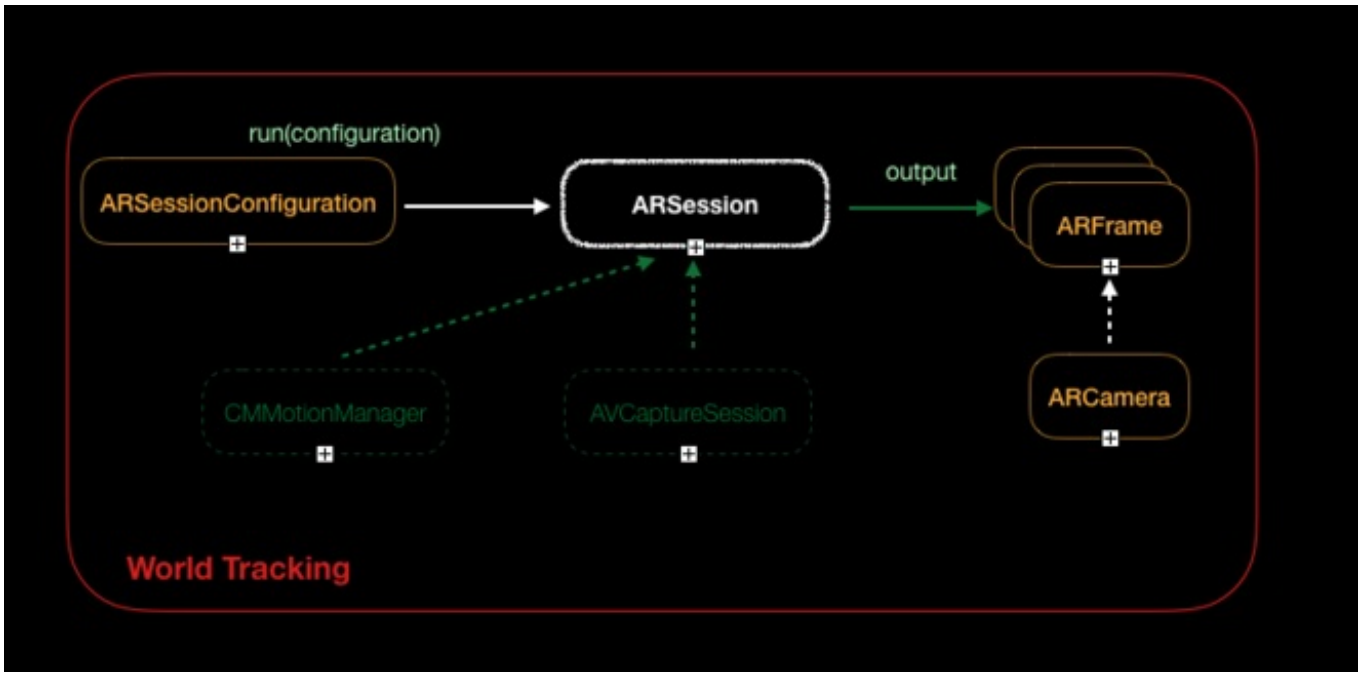
上图中，Intrinsic Matrix 与表示点 N 的向量相乘后，再除以 z'，就得到了一个 z 坐标为 1 的三维向量，
我们丢弃掉 z 坐标信息就得到了 N' 的坐标：((x' * fx)/z' + ox, (y' * fy)/z' + oy)。

这就是 Intrinsic Matrix 的作用过程，至于为何这么映射，则是相机原理的内容了，由于水平有限，就不做介绍了。如果不太好理解，我们这样简单理解为相机使用这个矩阵就可以将空间中的某个点映射到二维成像平面的一个点。

六、世界追踪(World Tracking)

在第一部分 AR 系统介绍时，我们看到虚拟椅子是放在地面上的，当我们移动时可以看到不同角度，我们也可以移动椅子，这些功能的实现都离不开世界追踪。总结来说，世界追踪用来为真实世界与虚拟世界结合提供有效信息，以便我们能在真实世界中看到一个更加真实的虚拟世界。

为了方便阅读，我们首先将 World Tracking 从 ARKit 架构图中抽取出：



WorldTracking.png

下面我们分析一下 ARKit 中与世界追踪相关的技术以及类。

ARSession

如果我们想要使用 ARKit，我们必须要创建一个 ARSession 对象并运行 ARSession。基本步骤如下：

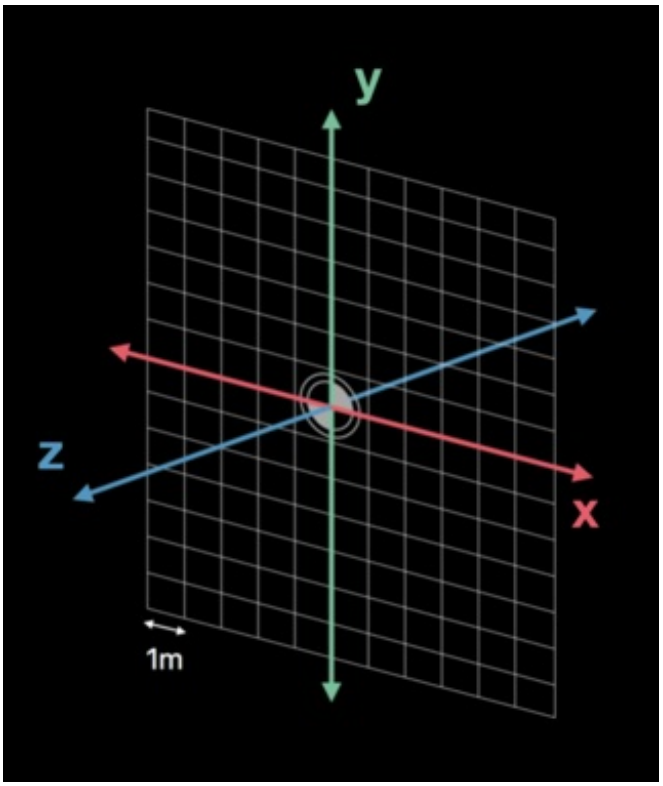
```
let configuration =
ARWorldTrackingSessionConfiguration()

let session = ARSession()

session.run(configuration)
```

从上面的代码看，运行一个 ARSession 的过程是很简单的，那么 ARSession 的底层如何进行世界追踪的呢？

- 首先，ARSession 底层使用了 AVCaputreSession 来获取摄像机拍摄的视频(一帧一帧的图像序列)。
- 其次，ARSession 底层使用了 CMMotionManager 来获取设备的运动信息(比如旋转角度、移动距离等)
- 最后，ARSession 根据获取的图像序列以及设备的运动信息进行分析，最后输出 ARFrame，ARFrame 中就包含有渲染虚拟世界所需的所有信息。

追踪什么？

那么世界追踪到底追踪了哪些信息？下图给出了 AR-World 的坐标系，当我们运行 ARSession 时设备所在的位置就是 AR-World 的坐标系原点。



WorldTrackingCoordinate.png

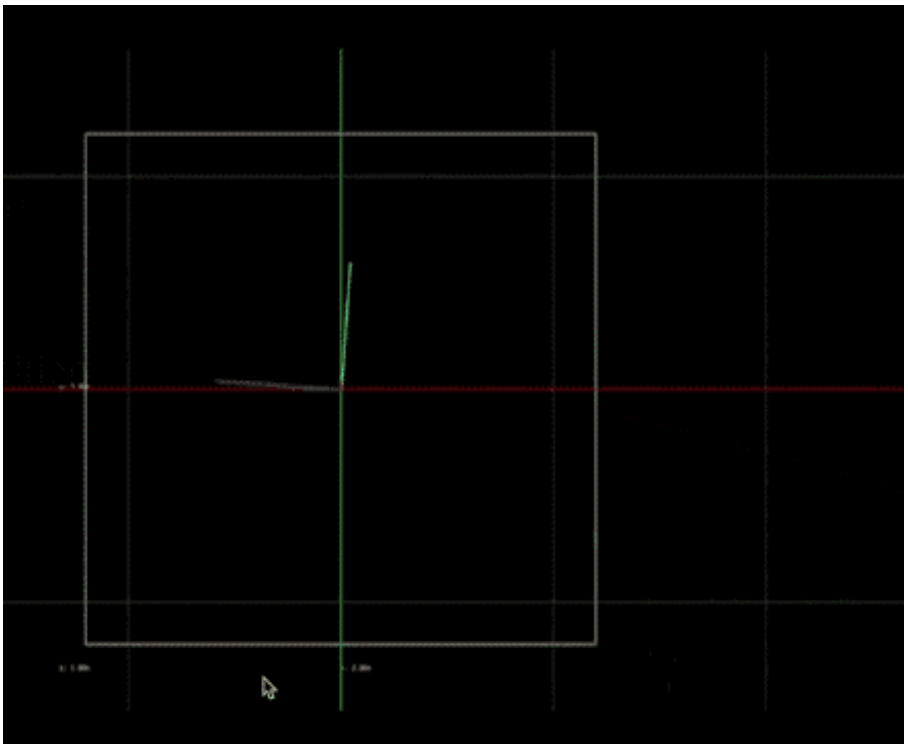
在这个 AR-World 坐标系中，ARKit 会追踪以下几个信息：

- 追踪设备的位置以及旋转，这里的两个信息均是相对于设备起始时的信息。
- 追踪物理距离(以“米”为单位)，例如 ARKit 检测到一个平面，我们希望知道这个平面有多大。
- 追踪我们手动添加的希望追踪的点，例如我们手动添加的一个虚拟物体。

世界追踪如何工作？

苹果文档中对世界追踪过程是这么解释的：ARKit 使用视觉惯性测距技术，对摄像头采集到的图像序列进行计算机视觉分析，并且与设备的运动传感器信息相结合。ARKit 会识别出每一帧图像中的特征点，并且根据特征点在连续的图像帧之间的位置变化，然后与运动传感器提供的信息进行比较，最终得到高精度的设备位置和偏转信息。

我们通过一个 gif 图来理解上面这段话：



WorldTrackingProcess.gif

- 上图中划出曲线的运动的点代表设备，可以看到以设备为中心有一个坐标系也在移动和旋转，这代表着设备在不断的移动和旋转。这个信息是通过设备的运动传感器获取的。
- 动图中右侧的黄色点是 3D 特征点。3D 特征点就是处理捕捉到的图像得到的，能代表物体特征的点。例如地

板的纹理、物体的边边角角都可以成为特征点。上图中我们看到当设备移动时，ARKit 在不断的追踪捕捉到的画面中的特征点。

- ARKit 将上面两个信息进行结合，最终得到了高精度的设备位置和偏转信息。

Configuration

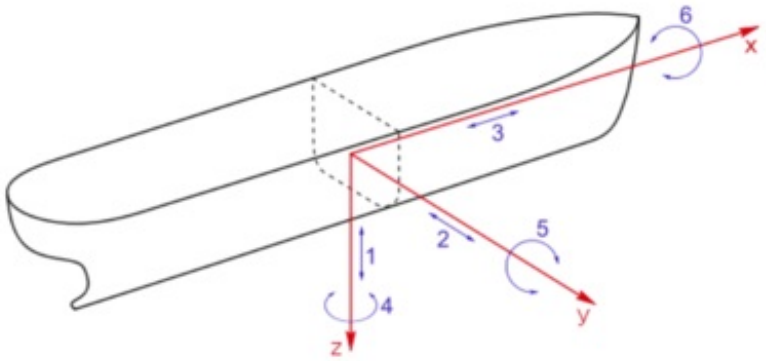
在运行 ARSession 时，我们必须要有一个 Configuration。Configuration 告诉 ARKit 应该如何追踪设备的运动。ARKit 为我们提供了两种类型的 Configuration：

- ARSessionConfiguration：提供 3DOF 追踪。
- ARWorldTrackingSessionConfiguration：提供 6DOF 追踪。

上面两个 Configuration 的差别就是 DOF 不一样，那么什么是 DOF？

1. DOF

   自由度(DOF,Degree Of Freedom)表示描述系统状态的独立参数的个数。6DOF 主要包括如下 6 个参数：



   DOF.png

   - 平移：
     1. 上下移动
     2. 左右移动
     3. 前后移动
   - 旋转：
     1. Yawing
     2. Pitching
     3. Rolling

其中 3DOF 中只包含旋转中的三个参数，平移的几个参数其实很好理解，为了更形象的理解旋转参数，我们看下面几个动图：

- Yawing 效果：

  yaw.gif

- Pitching 效果：

  pitch.gif

- Rolling 效果：

  roll.gif

1. 3DOF 与 6DOF 追踪的效果差异

   `ARWorldTrackingSessionConfiguration` 使用 3 个旋转参数和 3 个平移参数来追踪物理系统的状态，`ARSessionConfiguration` 使用 3 个旋转参数来追踪物理系统的状态。那么两者有什么效果差别？

   我们下面看两个图，下图一使用 3DOF 的 `ARSessionConfiguration` 进行世界追踪，下图二使用 6DOF 的 `ARWorldTrackingSessionConfiguration` 进行世界追踪：

   3DOF.gif

   6DOF.gif

   上面两个图，都是先对设备进行旋转，再对设备进行平移。

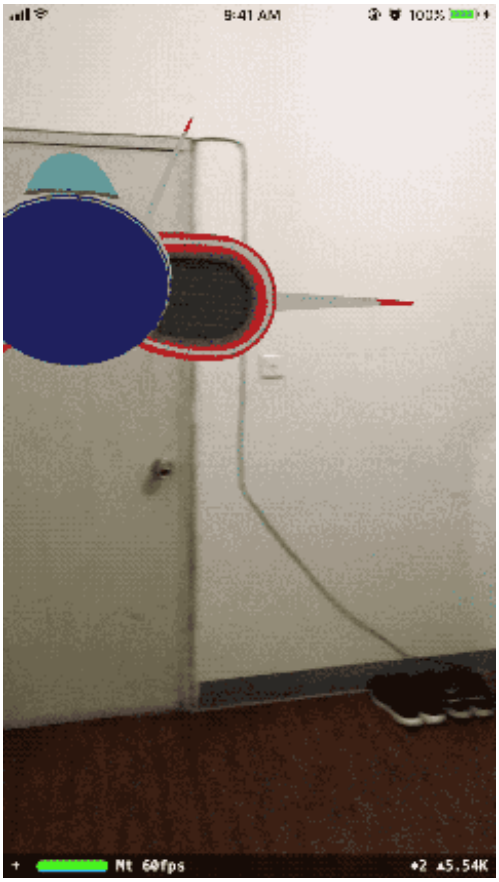
   那么，对于 3DOF 追踪，我们旋转设备时可以看到虚拟的飞机视角有所变化；但当平移时，我们可以看到飞机是随着设备进行移动的。

   对于 6DOF 追踪，我们旋转设备时可以看到虚拟的飞机视角有所变化(这点与 3DOF 追踪没有区别)；平移时，我们可以看到飞机的不同位置，例如向上平移看到了飞机的上表面，围着飞机平移可以看到飞机的四周，而 3DOF 没有提供这种平移的追踪。如果还是不理解两者区别，可以看动图的后半段，效果差异其实是非常明显的。

2. 判断当前设备是否支持某类 SessionConfiguration

```
class var isSupported:
Bool
```

示例代码如下：



6DOF.gif

```
if ARWorldTrackingSessionConfiguration.isSupported {
    configuration =
ARWorldTrackingSessionConfiguration()
} else {
    configuration = ARSessionConfiguration()
}
```

关于 `ARSessionConfiguration` 我们就介绍到这里，下面我们看一下 ARFrame。

ARFrame

ARFrame 中包含有世界追踪过程获取的所有信息，ARFrame 中与世界追踪有关的信息主要是：anchors 和 camera：

- camera: 含有摄像机的位置、旋转以及拍照参数等信息。

```
var camera:
[ARCamera]
```

- ahchors: 代表了追踪的点或面。

```
var anchors:
[ARAnchor]
```

至于 ARCamera 和 ARAnchor 是什么？下面分别进行介绍。

ARAnchor

ARAnchor.png

我们可以把 ARAnchor(AR 锚点) 理解为真实世界中的某个点或平面，anchor 中包含位置信息和旋转信息。拿到 anchor 后，可以在该 anchor 处放置一些虚拟物体。对于 ARAnchor 有如下几种操作：

- 我们可以使用 ARSession 的 add/remove 方法进行手动添加或删除 Anchor。例如，我们添加了一个虚拟物体到 ARKit 中，在之后的某个时候我们想要在刚才的虚拟物体上面再放置一个东西，那么我们可以为这个虚拟物体添加一个 anchor 到 ARSession 中，这样在后面可以通过 ARSession 获取到这个虚拟物体的锚点信息。
- 通过 ARSession 获取当前帧的所有 anchors：

```
let anchors =
session.currentFrame.anchors
```

- ARKit 自动添加 anchors。例如，ARKit 检测到了一个平面，ARKit 会为该平面创建一个 ARPlaneAnchor 并添加到 ARSession 中。
- ARSessionDelegate 可以监听到添加、删除和更新 ARAnchor 的通知。

ARAnchor 中的主要参数是 transform，这个参数是一个 4x4 的矩阵，矩阵中包含了 anchor 偏移、旋转和缩放信息。

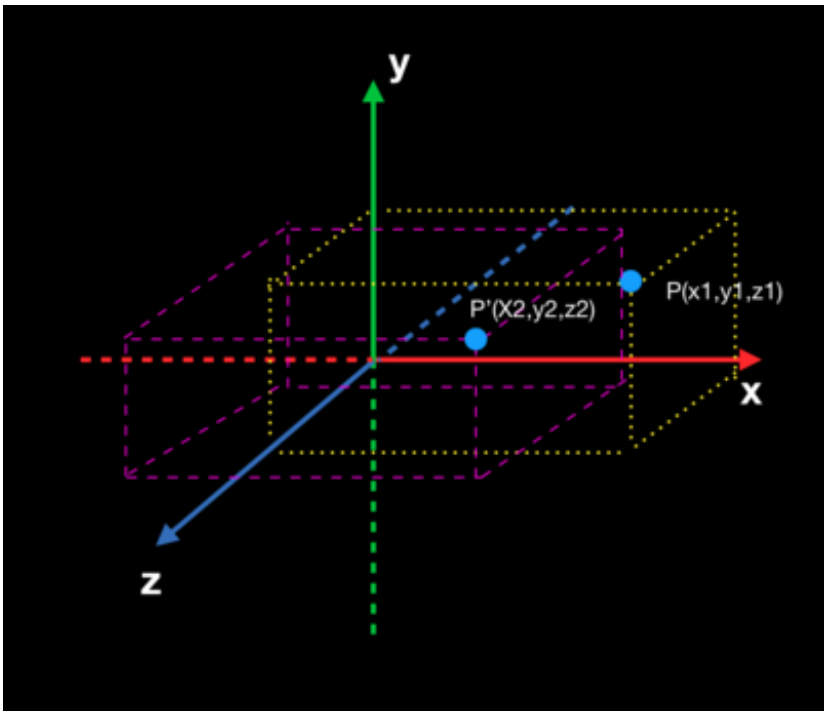
```
var transform: matrix_float4x4
```

这里可能存在的一个疑问就是，为何是一个 4x4 的矩阵，三维坐标系表示一个点不是用三个坐标就可以了吗？

4x4 矩阵？

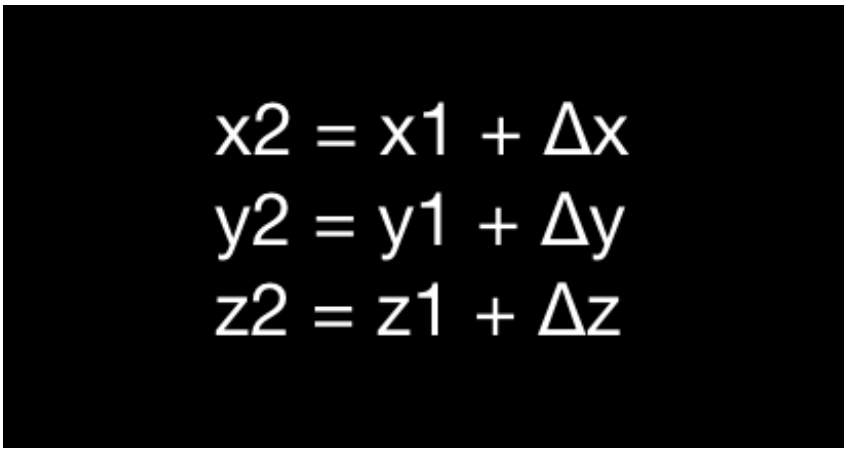
物体在三维空间中的运动通常分类两类：平移和旋转，那么表达一个物体的变化就应该能够包含两类运动变化。

- 平移



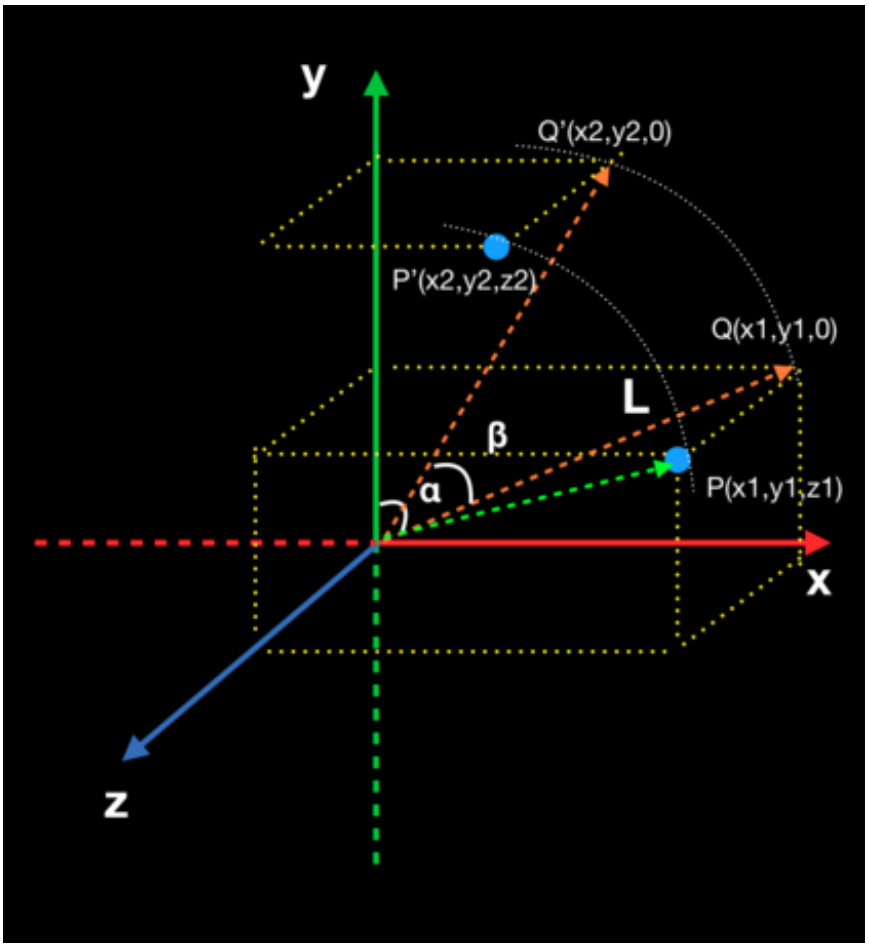
4x4Translation.png

首先看上图，假设有一个长方体(黄色虚线)沿 x 轴平移Δx、沿 y 轴平移Δy、沿 z 轴平移Δz 到了另一个位置(紫色虚线)。长方体的顶点 P(x1, y1, z1)则平移到了 P'(x2, y2, z2)，使用公式表示如下：



4x4TranslationExpression.png

- 旋转



4x4Rotation.png

在旋转之前，上图中包含以下信息：

- 黄色虚线的长方体
- P(x1, y1, z1)是长方体的一个顶点
- P 点在 xy 平面的投影点 Q(x1, y1, 0)
- Q 与坐标原点的距离为 L
- Q 与坐标原点连线与 y 轴的夹角是α

那么在旋转之前，P 点坐标可以表示为：

```
    x1 = L *
sinα
    y1 = L *
cosα
    z1 = z1
```

下面我们让长方体绕着 z 轴逆时针旋转β角度，那么看图可以得到以下信息：

- P 点会绕着 z 轴逆时针旋转β角度到达 P'(x2, y2, z2)
- P' 在 xy 平面投影点 Q'(x2, y2, 0)

- Q' 与 Q 在以 xy 平面原点为圆心，半径为 L 的圆上
- Q' 与原点连线与 Q 与原点连线之间的夹角为 β
- Q'与原点连线与 y 轴的角度是 α-β。

那么在旋转之后，P' 点的坐标可以表示为：

$$
\begin{aligned}
x2 &= L * \sin(\alpha - \beta) \\
&= L * \sin\alpha * \cos\beta - L * \cos\alpha * \sin\beta \\
&= \cos\beta * x1 - \sin\beta * y1 \\
\\
y2 &= L * \cos(\alpha - \beta) \\
&= L * \cos\alpha * \cos\beta + L * \sin\alpha * \sin\beta \\
&= \cos\beta * y1 + \sin\beta * x1 \\
\\
z2 &= z1
\end{aligned}
$$

4x4RotationExpression.png

使用矩阵来表示：

$$
\begin{bmatrix} x2 \\ y2 \\ z2 \end{bmatrix} = \begin{bmatrix} cos\beta & -sin\beta & 0 \\ cos\beta & sin\beta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x1 \\ y1 \\ z1 \end{bmatrix}
$$

4x4RotationExpression2.png

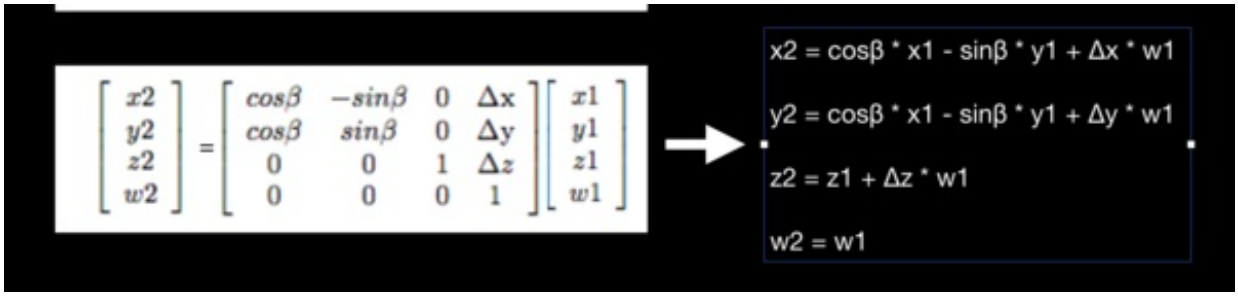
从上面的分析可以看出，为了表达旋转信息，我们需要一个 3x3 的矩阵，在表达了旋转信息的 3x3 矩阵中，我们无法表达平移信息，为了同时表达平移和旋转信息，在 3D 计算机图形学中引入了齐次坐标系，在齐次坐标系中，使用四维矩阵表示一个点或向量：

$$
P = \begin{bmatrix} x1 \\ y1 \\ z1 \\ w1 \end{bmatrix}
$$

w1 = 1 for point.
w1 = 0 for vector.

HomogeneousPoint.png

加入一个变化是先绕着 z 轴旋转 β 角度，再沿 x 轴平移Δx、沿 y 轴平移Δy、沿 z 轴平移Δz，我们可以用以下矩阵变化表示：



Homogeneous.png

最后，还有一种变化是缩放，在齐次坐标系中只需要在前三列矩阵中某个位置添加一个系数即可，比较简单，这里不在展示矩阵变换。从上面可以看出，为了完整的表达一个物体在 3D 空间的变化，需要一个 4x4 矩阵。

ARCamera

ARCamera 中的主要参数是：

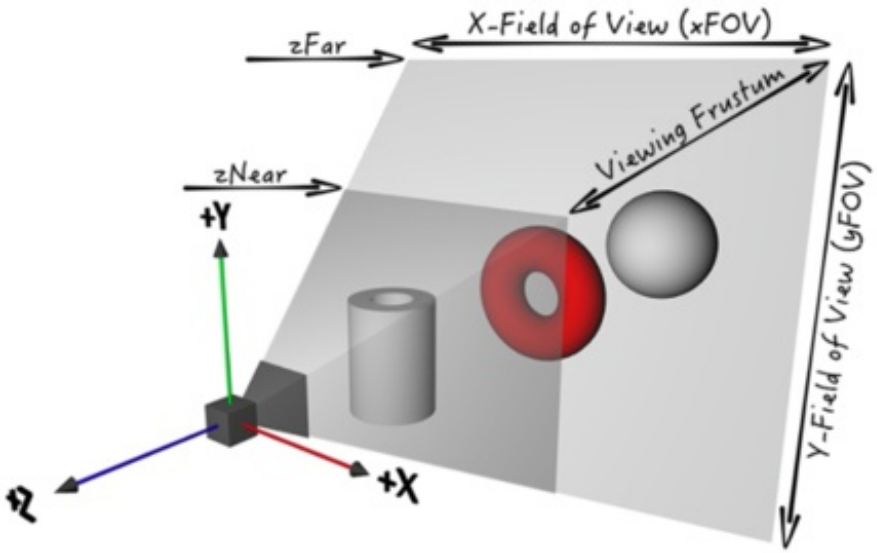
- transform: 表示摄像头相对于起始时的位置和旋转信息。至于为何是 4x4 矩阵，上面已经解释过了。

```
var transform: matrix_float4x4
```

- eulerAngles: 欧拉角，另一种表示摄像头的偏转角度信息的方式，与我们之前介绍的 3DOF 有关，欧拉证明了一个物体的任何旋转都可以分解为 yaw、pitch、roll 三个方向的旋转。

```
var eulerAngles: vector_float3
```

- projectionMatrix: 投影矩阵，其实这个很类似于上面介绍的 Intrinsic Matrix，但不同点是，投影矩阵是将 AR-world 中的物体投影到屏幕上，由于 AR-world 中采用的是齐次坐标，所以这里是一个 4x4 矩阵，投影矩阵除了决定 AR-world 中点应该映射到屏幕哪个点之外，还决定了哪些范围的点是不需要的，我们看下图：



ProjectionMatrix.png

上图中 Field-of-View 和 View-Frustum 影响了投影矩阵的部分参数，对于超过 Field-of-View 或者超出 View-Frustum 范围的点，ARKit 不会对其进行投影映射到屏幕。

此外，ARKit 还提供了一个接口让我们自定义 Field-of-View 和 View-Frustum：

```
func projectionMatrix(withViewportSize: CGSize,
                      orientation:
UIInterfaceOrientation,
                                zNear: CGFloat,
                                 zFar: CGFloat)
```

追踪质量：

世界追踪需要一定的条件才能达到较好的效果，如果达不到所需的条件要求，那么世界追踪的质量会降低，甚至会无法追踪。较好的世界追踪质量主要有以下三个依赖条件：

- 运动传感器不能停止工作。如果运动传感器停止了工作，那么就无法拿到设备的运动信息。根据我们之前提到的世界追踪的工作原理，毫无疑问，追踪质量会下降甚至无法工作。
- 真实世界的场景需要有一定特征点可追踪。世界追踪需要不断分析和追踪捕捉到的图像序列中特征点，如果图像是一面白墙，那么特征点非常少，那么追踪质量就会下降。
- 设备移动速度不能过快。如果设备移动太快，那么 ARKit 无法分析出不同图像帧之中的特征点的对应关系，也会导致追踪质量下降。

追踪状态

世界追踪有三种状态，我们可以通过 camera.trackingState 获取当前的追踪状态。

TrackingState.png

从上图我们看到有三种追踪状态：

- Not Available：世界追踪正在初始化，还未开始工作。
- Normal： 正常工作状态。
- Limited：限制状态，当追踪质量受到影响时，追踪状态可能会变为 Limited 状态。

与 TrackingState 关联的一个信息是 ARCamera.TrackingState.Reason，这是一个枚举类型：

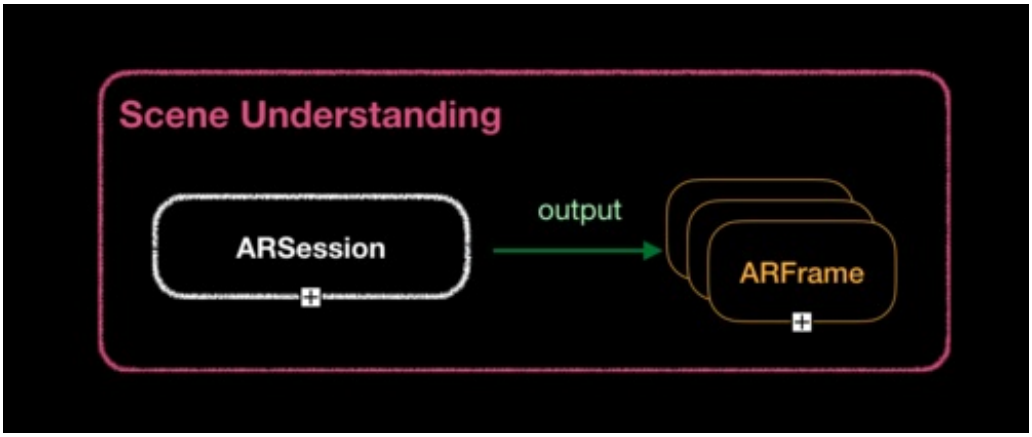
- case excessiveMotion：设备移动过快，无法正常追踪。
- case initializing：正在初始化。
- case insufficientFeatures：特征过少，无法正常追踪。
- case none：正常工作。

我们可以通过 ARSessionObserver 协议去获取追踪状态的变化，比较简单，可以直接查看接口文档，这里不做深入介绍。

到这里，ARKit 中有关于世界追踪的知识基本介绍完了，世界追踪算是 ARKit 中核心功能了，如果理解了本部分内容，相信去看苹果的接口文档也会觉着非常容易理解。如果没有看懂，可以去看一下分享的录播(本文开头有链接)。

七、场景解析(Scene Understanding)

为了方便阅读，我们首先将 Scene Understanding 从 ARKit 架构图中抽取出：



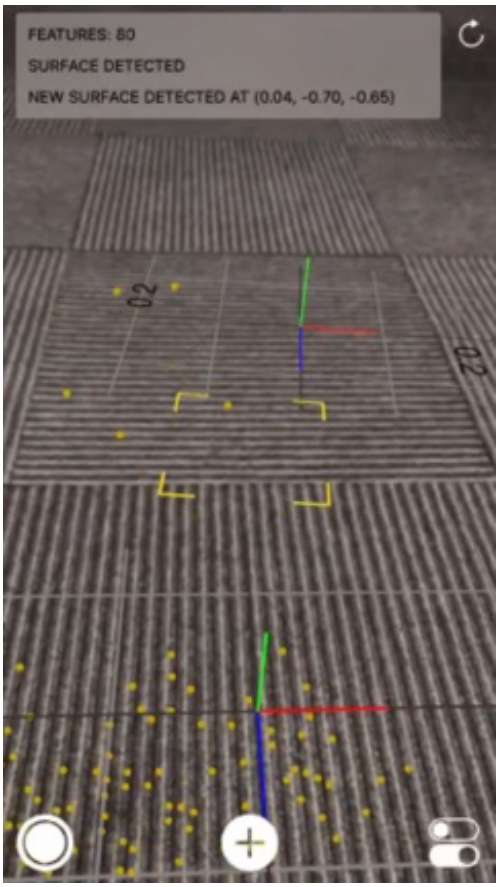
SceneUnderstanding.png

场景解析主要功能是对现实世界的场景进行分析，解析出比如现实世界的平面等信息，可以让我们把一些虚拟物体放在某些实物处。ARKit 提供的场景解析主要有**平面检测**、**场景交互**以及**光照估计**三种，下面逐个分析。

平面检测(Plane detection)

主要功能：

- ARKit 的平面检测用于检测出现实世界的水平面。



PlaneDetection.png

上图中可以看出，ARkit 检测出了两个平面，图中的两个三维坐标系是检测出的平面的本地坐标系，此外，检测出的平面是有一个大小范围的。

- 平面检测是一个动态的过程，当摄像机不断移动时，检测到的平面也会不断的变化。下图中可以看到当移动摄像机时，已经检测到的平面的坐标原点以及平面范围都在不断的变化。

MultipleFramePlane.gif

- 此外，随着平面的动态检测，不同平面也可能会合并为一个新的平面。下图中可以看到已经检测到的平面随着摄像机移动合并为了一个平面。

PlaneMerge.gif

开启平面检测

开启平面检测很简单，只需要在 run ARSession 之前，将 ARSessionConfiguration 的 planeDetection 属性设为 true 即可。

MultipleFramePlane.gif

PlaneMerge.gif

```
let configuration =
ARWorldTrackingSessionConfiguration()
configuration.planeDetection = .horizontal

let session = ARSession()

session.run(configuration)
```

平面的表示方式

当 ARKit 检测到一个平面时，ARKit 会为该平面自动添加一个 `ARPlaneAnchor`，这个 `ARPlaneAnchor` 就表示了

一个平面。

`ARPlaneAnchor` 主要有以下属性：

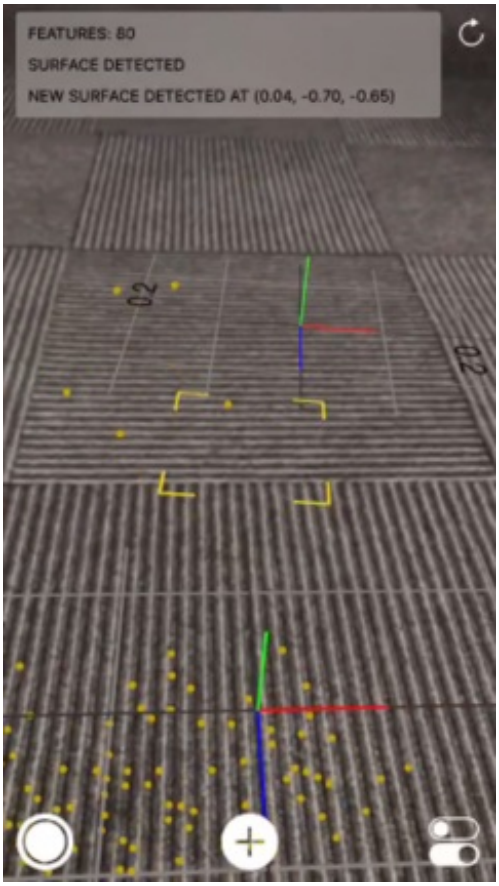
- alignment: 表示该平面的方向，目前只有 horizontal 一个可能值，表示这个平面是水平面。ARKit 目前无法检测出垂直平面。

  ```
  var alignment:
  ARPlaneAnchor.Alignment
  ```

- center: 表示该平面的本地坐标系的中心点。如下图中检测到的平面都有一个三维坐标系，center 所代表的就是坐标系的原点：

  ```
  var center:
  vector_float3
  ```

PlaneDetection.png



- extent: 表示该平面的大小范围。如上图中检测到的屏幕都有一个范围大小。

  ```
  var extent: vector_float3
  ```

ARSessionDelegate

当 ARKit 系统检测到新平面时，ARKit 会自动添加一个 ARPlaneAnchor 到 ARSession 中。我们可以通过 ARSessionDelegate 获取当前 ARSession 的 ARAnchor 改变的通知，主要有以下三种情况：

- 新加入了 ARAnchor

  ```
  func session(_ session: ARSession, didAdd anchors:
  [ARAnchor])
  ```

  对于平面检测来说，当新检测到某平面时，我们会收到该通知，通知中的 ARAnchor 数组会包含新添加的平面，其类型是 ARPlaneAnchor，我们可以像下面这样使用：

```
func session(_ session: ARSession, didAdd anchors: [ARAnchor])
{
    for anchor in anchors {
        if let anchor = anchor as? ARPlaneAnchor {
            print(anchor.center)
            print(anchor.extent)
        }
    }
}
```

- ARAnchor 更新

```
func session(_ session: ARSession, didUpdate anchors:
[ARAnchor])
```

从上面我们知道当设备移动时，检测到的平面是不断更新的，当平面更新时，会回调这个接口。

- 删除 ARAnchor

```
func session(_ session: ARSession, didRemove anchors:
[ARAnchor])
```

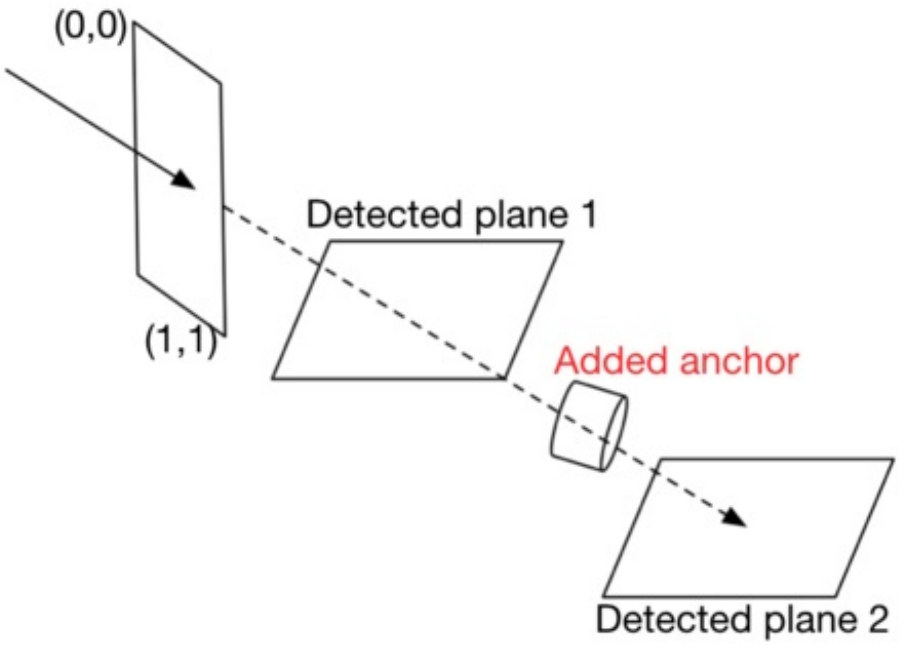
当手动删除某个 Anchor 时，会回调此方法。此外，对于检测到的平面来说，如果两个平面进行了合并，则会删除其中一个，此时也会回调此方法。

场景交互(Hit-testing)

Hit-testing 是为了获取当前捕捉到的图像中某点击位置有关的信息(包括平面、特征点、ARAnchor 等)。

工作原理

先看一下原理图：



Hit-testing.png

当点击屏幕时，ARKit 会发射一个射线，假设屏幕平面是三维坐标系中的 xy 平面，那么该射线会沿着 z 轴方向射向屏幕里面，这就是一次 Hit-testing 过程。此次过程会将射线遇到的所有有用信息返回，返回结果以离屏幕距离进行排序，离屏幕最近的排在最前面。

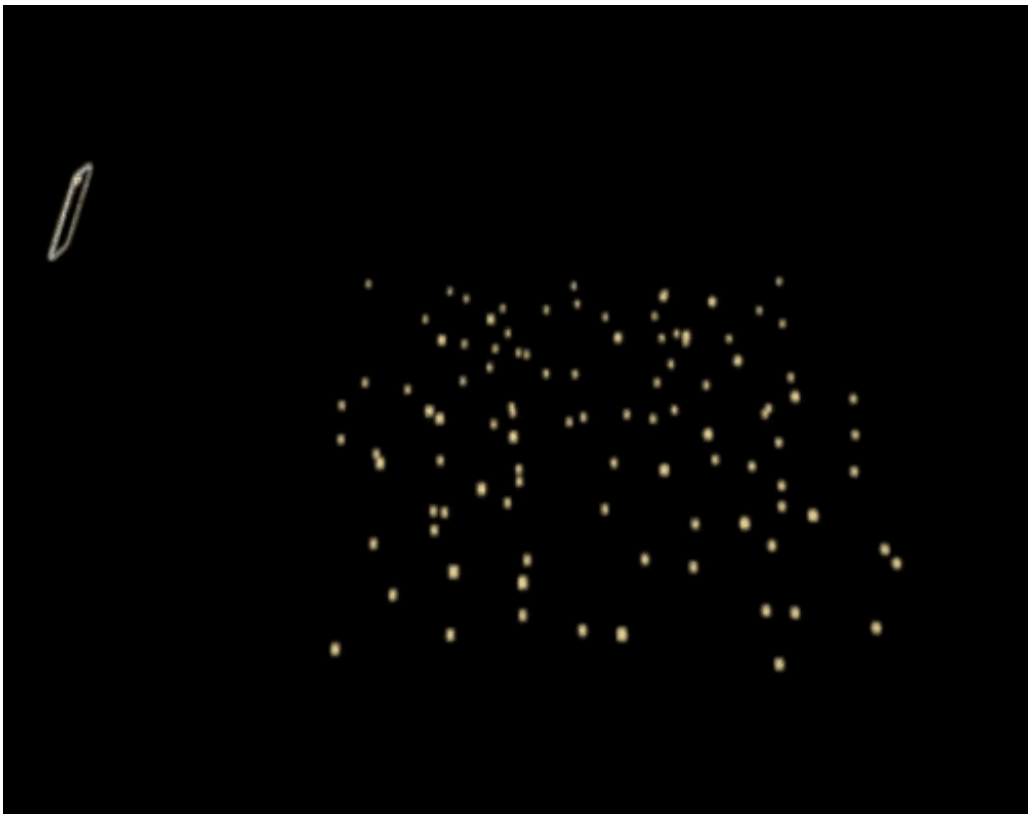
ResultType

ARFrame 提供了 Hit-testing 的接口：

```
func hitTest(_ point: CGPoint, types: ARHitTestResult.ResultType) ->
[ARHitTestResult]
```

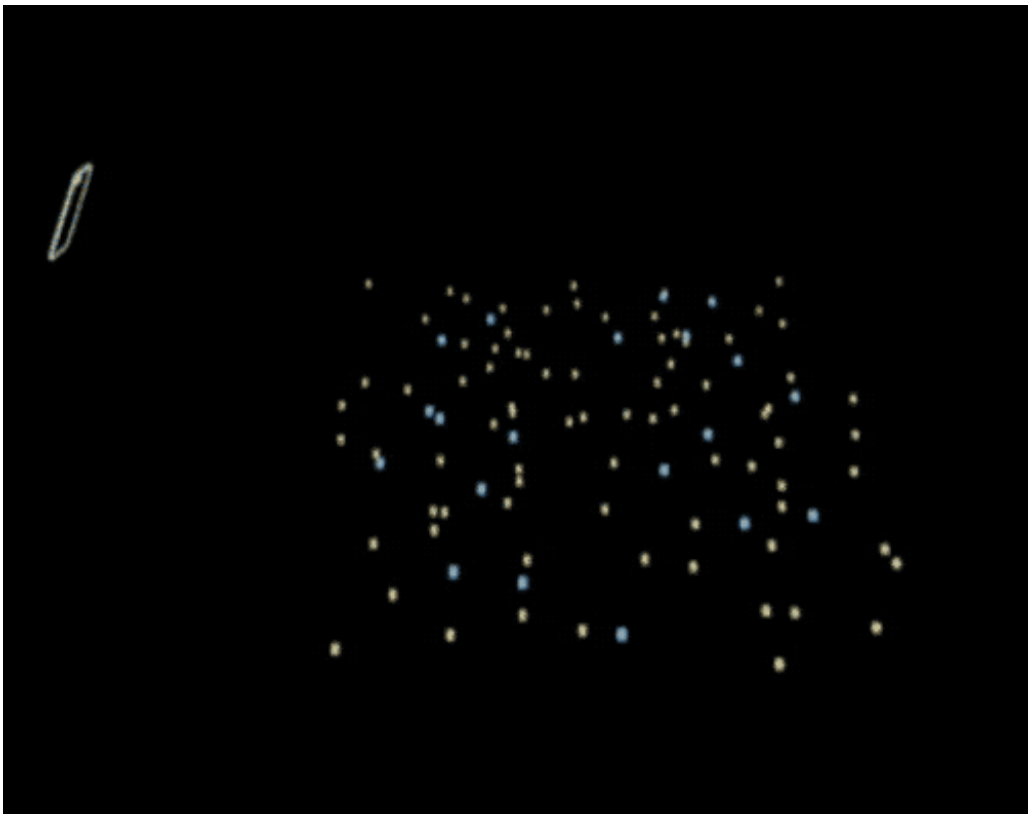
上述接口中有一个 types 参数，该参数表示此次 Hit-testing 过程需要获取的信息类型。ResultType 有以下四种：

- featurePoint
  表示此次 Hit-testing 过程希望返回当前图像中 Hit-testing 射线经过的 3D 特征点。如下图：



HitFeaturePoint.gif

- estimatedHorizontalPlane
  表示此次 Hit-testing 过程希望返回当前图像中 Hit-testing 射线经过的预估平面。预估平面表示 ARKit 当前检测到一个可能是平面的信息，但当前尚未确定是平面，所以 ARKit 还没有为此预估平面添加 ARPlaneAnchor。
  如下图：



HitEstimatedPlane.gif

- existingPlaneUsingExtent
  表示此次 Hit-testing 过程希望返回当前图像中 Hit-testing 射线经过的有大小范围的平面。

HitExistingPlaneUseExtent.gif

上图中，如果 Hit-testing 射线经过了有大小范围的绿色平面，则会返回此平面，如果射线落在了绿色平面的外面，则不会返回此平面。

- existingPlane
  表示此次 Hit-testing 过程希望返回当前图像中 Hit-testing 射线经过的无限大小的平面。

HitExistingPlane.gif

上图中，平面大小是绿色平面所展示的大小，但 exsitingPlane 选项表示即使 Hit-testing 射线落在了绿色平面外面，也会将此平面返回。换句话说，将所有平面无限延展，只要 Hit-testing 射线经过了无限延展后的平面，就会返回该平面。

使用方法

下面给出使用 Hit-testing 的示例代码：

```
let point = CGPoint(x: 0.5, y: 0.5)


let results = frame. hitTest(point, types: [.featurePoint,
.estimatedHorizontalPlane])


if let closestResult = results.first {

    anchor = (transform: closestResult.worldTransform)

    session.add(anchor: anchor)
}
```

上面代码中，Hit-testing 的 point(0.5, 0.5)代表屏幕的中心，屏幕左上角为(0, 0)，右下角为(1, 1)。 对于 featurePoint 和 estimatedHorizontalPlane 的结果，ARKit 没有为其添加 ARAnchor，我们可以使用 Hit-testing 获取信息后自己为 ARSession 添加 ARAnchor，上面代码就显示了此过程。

光照估计(Light estimation)

LightEstimation.png

上图中，一个虚拟物体茶杯被放在了现实世界的桌子上。

当周围环境光线较好时，摄像机捕捉到的图像光照强度也较好，此时，我们放在桌子上的茶杯看起来就比较贴近于现实效果，如上图最左边的图。但是当周围光线较暗时，摄像机捕捉到的图像也较暗，如上图中间的图，此时茶杯的亮度就显得跟现实世界格格不入。

针对这种情况，ARKit 提供了光照估计，开启光照估计后，我们可以拿到当前图像的光照强度，从而能够以更自然的光照强度去渲染虚拟物体，如上图最右边的图。

光照估计基于当前捕捉到的图像的曝光等信息，给出一个估计的光照强度值(单位为 lumen，光强单位)。默认的光照强度为 1000lumen，当现实世界较亮时，我们可以拿到一个高于 1000lumen 的值，相反，当现实世界光照较暗时，我们会拿到一个低于 1000lumen 的值。

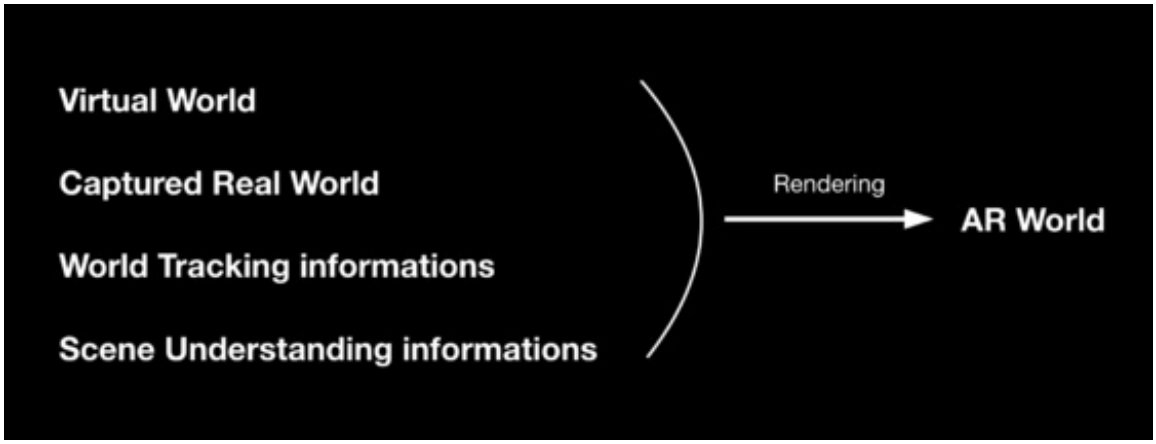
ARKit 的光照估计默认是开启的，当然也可以通过下述方式手动配置：

```
configuration.isLightEstimationEnabled =
true
```

获取光照估计的光照强度也很简单，只需要拿到当前的 ARFrame，通过以下代码即可获取估计的光照强度：

```
let intensity = frame.lightEstimate?.ambientIntensity
```

八、渲染(Rendering)



Rendering.png

渲染是呈现 AR world 的最后一个过程。此过程将创建的虚拟世界、捕捉的真实世界、ARKit 追踪的信息以及 ARKit 场景解析的的信息结合在一起，渲染出一个 AR world。渲染过程需要实现以下几点才能渲染出正确的 AR world：

- 将摄像机捕捉到的真实世界的视频作为背景。
- 将世界追踪到的相机状态信息实时更新到 AR world 中的相机。
- 处理光照估计的光照强度。
- 实时渲染虚拟世界物体在屏幕中的位置。

如果我们自己处理这个过程，可以看到还是比较复杂的，ARKit 为简化开发者的渲染过程，为开发者提供了简单易用的使用 SceneKit(3D 引擎)以及 SpriteKit(2D 引擎)渲染的视图`ARSCNView`以及`ARSKView`。当然开发者也可以使用其他引擎进行渲染，只需要将以上几个信息进行处理融合即可。这里只介绍下`ARSCNView`，对于使用其他引擎渲染，可参考 WWDC2017-ARKit 最后的 Metal 渲染示例。

ARSCNView 的功能

ARSCNView 帮我们做了如下几件事情：

- 将摄像机捕捉到的真实世界的视频作为背景。
- 处理光照估计信息，不断更新画面的光照强度。
- 将 SCNNode 与 ARAnchor 绑定，也就是说当添加一个 SCNNode 时，ARSCNView 会同时添加一个 ARAnchor 到 ARKit 中。
- 不断更新 SceneKit 中的相机位置和角度。
- 将 SceneKit 中的坐标系结合到 AR world 的坐标系中，不断渲染 SceneKit 场景到真实世界的画面中。

关于 ARSCNView 的各个属性这里不再进行一一介绍了，如果已经掌握了之前章节的内容，相信直接看 ARSCNView 的接口文档不会有什么问题。下面对 `ARSCNViewDelegate` 做一下简单介绍。

ARSCNViewDelegate

我们在介绍场景解析时，已经介绍过了 `ARSessionDelegate`，而 `ARSCNViewDelegate` 其实与 `ARSessionDelegate` 是有关系的，下面我们再来看下 `ARSessionDelegate` 的三个回调：

- 新加入了 ARAnchor

```
 func session(_ session: ARSession, didAdd anchors:
[ARAnchor])
```

当 ARKit 新添加一个 ARAnchor 时，ARSessionDelegate 会收到上述回调。此时，ARKit 会回调 ARSCNViewDelegate 的下面一个方法询问需要为此新加的 ARAnchor 添加 SCNNode。

```
func renderer(_ renderer: SCNSceneRenderer, nodeFor: ARAnchor) ->
SCNNode?
```

当调用完上个方法之后，ARKit 会回调 ARSCNViewDelegate 的下面一个方法告知 delegate 已为新添加的 ARAnchor 添加了一个 SCNNode。

```
func renderer(_ renderer: SCNSceneRenderer, didAdd: SCNNode, for:
ARAnchor)
```

- ARAnchor 更新

```
func session(_ session: ARSession, didUpdate anchors:
[ARAnchor])
```

当某个 ARAnchor 更新时，ARSessionDelegate 会收到上述回调，此时，ARSCNViewDelegate 会收到以下回调：

```
func renderer(_ renderer: SCNSceneRenderer, willUpdate: SCNNode, for:
ARAnchor)
func renderer(_ renderer: SCNSceneRenderer, didUpdate: SCNNode, for: ARAnchor)
```

- 删除 ARAnchor

```
func session(_ session: ARSession, didRemove anchors:
[ARAnchor])
```

当某个 ARAnchor 删除时，ARSessionDelegate 会收到上述回调，此时，ARSCNViewDelegate 会收到以下回调：

```
func renderer(_ renderer: SCNSceneRenderer, didRemove: SCNNode, for:
ARAnchor)
```

本章节没有对渲染进行太过深入的介绍，主要考虑到渲染过程中用到的知识点，在之前大部分已经介绍过了，剩下的只是一些接口的使用方法，相信使用过苹果各种 Kit 的开发者在掌握以上章节内容后，直接查看开发者文档，并参考苹果的官方 demo，使用起来应该不会遇到太多困难。

介绍完渲染之后，本文也就算是结束了。文中的内容涉及到了多个图形学的有关知识，有些不太好理解的欢迎交流，或者查看分享的录播视频：ARKit 分享直播

欢迎转载本文，请声明原文地址及作者，多谢。

九、参考文档