第 3 章

渲 染 基 础

渲染(Render)在计算机图形学中是指将几何网格、纹理等数字元素按照一定的光照模型生成可视图像的过程。从本质上讲,计算机图形学采用数学的方法对现实世界建模,并按照物理学规律处理数字世界中各元素之间的相互关系,最后仿真现实世界的视觉表现。由于现实世界的复杂性,特别是光源、材质、反射、折射、衍射、透射现象的复杂多样性,导致精确的仿真在计算上变得非常困难。经过几十年的发展,人们通过将整个仿真过程分解成若干个紧密结合的子过程降低复杂性后比较好地解决了这个问题,并称整个相互连接、紧密合作的过程为图形渲染管线。

本章所述的渲染只是整个渲染管线的最后阶段,即使如此,渲染包含的内容仍然纷繁复杂,鉴于本书的目的,我们只能从很浅的层面对渲染进行阐述。但是,渲染又非常重要,因为它决定了 AR 应用最终的外观表现,了解一些基础概念有助于更好地理解相关过程。

3.1 材质纹理

在利用光照模型计算物体表面光照信息时,需要使用物体表面的各类参数,即物体与光交互的相关信息,这些记录物体表面与光交互的参数集合就叫材质(material)。材质参数通常包括物体表面色彩、纹理、光滑度、透明度、反射率、折射率、发光度等,材质决定了物体的外观质感。材质只是一组数字参数,由于光照模型不同、渲染引擎不同,相同物体所使用的材质参数也会不一样,如 RealityKit 使用的材质与 Unreal 使用的材质就不通用。

纹理(texture)是物体具体的外观表现,通常指最终可以以位图形式使用的 2D 图片,纹理用于表现物体的视觉外观,当把纹理按照特定的方式映射到模型表面时能使模型看上去更加真实,如图 3-1 所示。

图 3-1　纹理是用于表现物体外观的 2D 图像

在计算机图形渲染中，由于真实世界物体的精细性和计算机资源的有限性，无法将所有物体都建成高精度模型，为降低计算成本，也会使用图片模拟精细的模型外观表现，这类图像叫贴图（map），如法线贴图用于提供像素级别的表面法线。纹理侧重于描述物体表面外观特征，而贴图侧重于描述物体表面的视觉表现，这两者有区别，但在本节中不加以区分。

RealityKit 支持 PBR 渲染（PBR 渲染稍后会详细阐述），PBR 渲染使用了多种贴图模拟真实物体的外观表现，每一种贴图负责实现某种特定效果，这些贴图包括漫反射贴图、法线贴图、高度贴图、环境光遮挡贴图、自发光贴图、置换贴图、金属感贴图、粗糙度贴图、清漆贴图等。如渲染一个高置信的照片级地球可能需要使用其中很多种贴图，如图 3-2 所示。

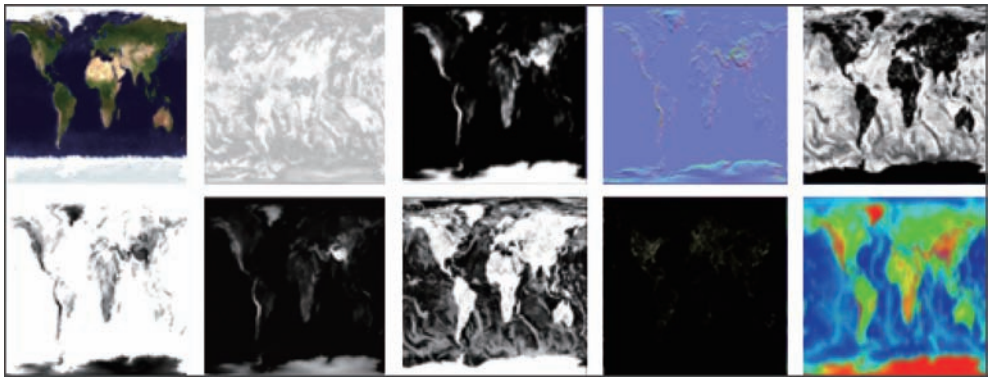

图 3-2　PBR 渲染使用多种贴图模拟物体的外观

1. 漫反射贴图（Diffuse map、Base Color map、Albedo map）

漫反射贴图用于表现物体的基本外观，赋予物体基本质感，呈现物体被光照而显出的颜色和强度。使用漫反射贴图的地球如图 3-3 所示，通过漫反射贴图，地球被赋予基本的外观。

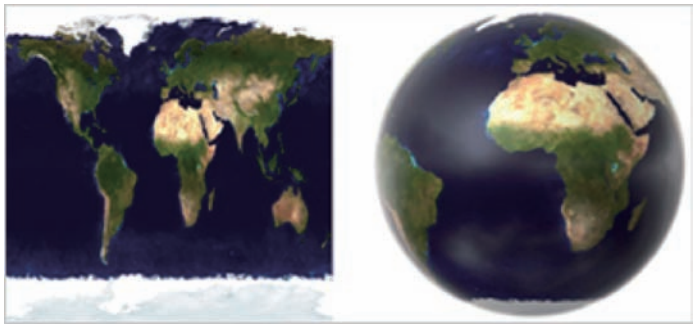

图 3-3　漫反射贴图表现物体的基本外观

2. 不透明贴图（Opacity map）

不透明贴图通常为一张黑白灰度图，用于控制透明区域，黑色部分为透明区域，而白色部分为保留区域。利用不透明贴图可以实现透明、半透明、镂空效果，如图 3-4 所示。应当注意的是，不透明贴图应谨慎使用，它可能会导致 GPU 性能恶化。

3. 法线贴图（Normal map）

法线贴图用于提供逐像素的法线信息，精确控制像素的光照计算，能在不增加模型顶点数的情况下大

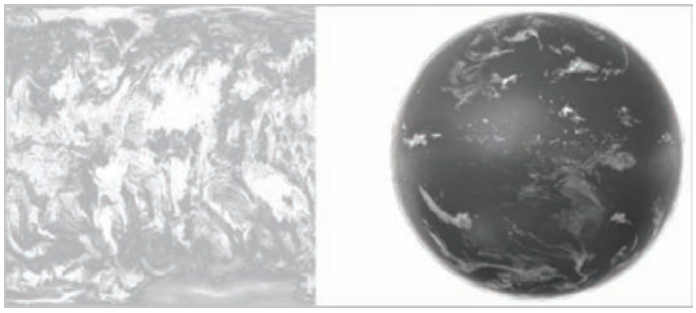

图 3-4　使用不透明贴图控制物体透明度

幅提高细节表现力，可以让细节程度较低的表面生成高细节程度的精确光照和反射效果，如使用法线贴图表现桌面木材细纹、表现瓷器凸凹不平的磨砂纹理，法线贴图一般呈现蓝紫色，如图 3-5 所示。

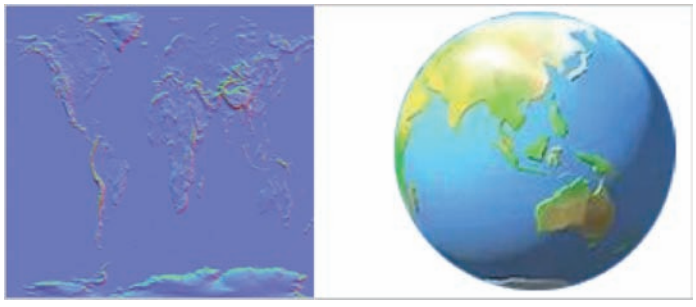

图 3-5　法线贴图

4. 高度贴图（Height map）

高度贴图是一张灰度图，用纯白表示最高的点，用纯黑表示最低的点，中间值表示对应的高度值，如图 3-6 所示。高度贴图一般用于调整模型顶点的位移，它不是 PBR 渲染需要的贴图，但通过高度贴图可以生成法线贴图。

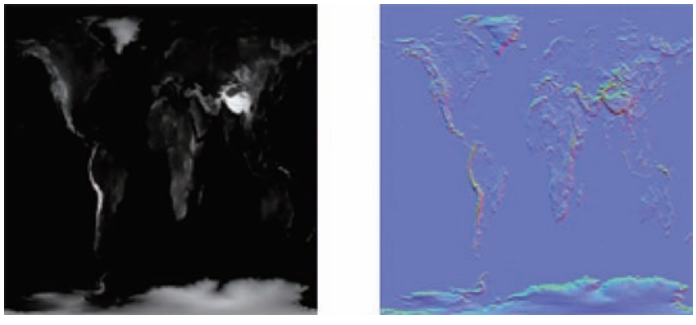

图 3-6　高度贴图可以转成法线贴图

5. 环境光遮挡贴图（Ambient Occlusion map）

在仅有环境光的情况下，物体的所有面接收的光照完全一样，渲染后就没有立体感和层次感，给人一种扁平的感觉，缺乏物体应有的明暗效果。为解决这个问题，引入了环境光遮挡贴图，模拟物体自身相互遮挡的效果，如凹面或者裂缝中接收的光照较少，渲染出来就会偏暗。环境光遮挡贴图也是灰度图，黑色区域阻

止所有的光照，而白色区域则允许光照，如图 3-7 所示。

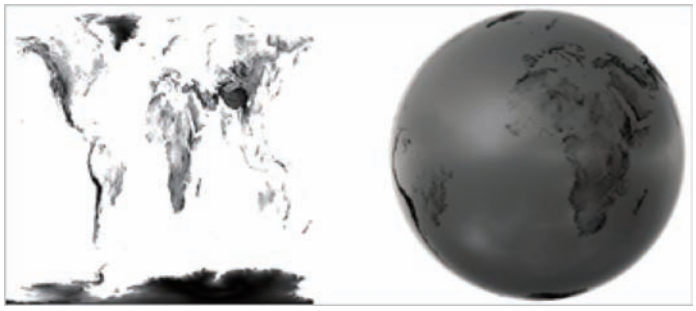

图 3-7　环境光遮挡贴图可以产生自我遮挡的效果

6. 发光贴图（Emission map，Emissive map）

在现实世界中，有类物体自带发光属性，如广告牌匾、荧光棒等。在 PBR 图形渲染中使用发光贴图模拟这类现象，发光贴图效果会叠加到其他贴图的光照效果后，因此可以渲染出光晕效果，如图 3-8 所示。发光贴图不是光源，无法照亮别的物体。

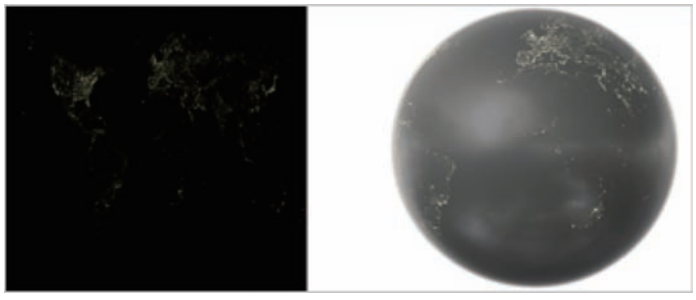

图 3-8　发光贴图可以营造物体自发光效果

7. 自照明贴图（Self-illumination map）

自照明贴图通常应用于所有效果之后，用于营造特殊视觉表现，如色彩化、亮化、暗化等，如图 3-9 所示。

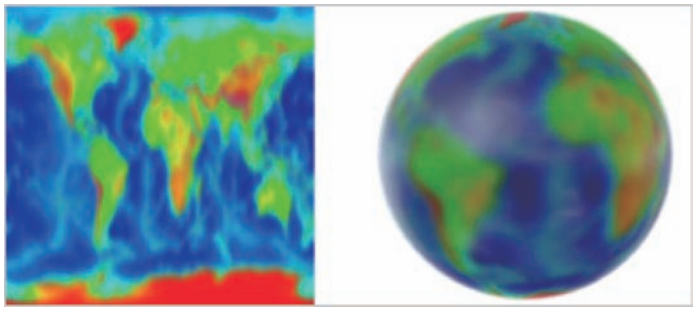

图 3-9　自照明贴图常用于营造特殊的外观表现

8. 置换贴图（Displacement map）

使用法线贴图模拟物体表面细节时，并不会改变模型顶点位置，因此，营造的是一种假象，其本质上是通过使用逐像素的法线信息参与光照计算模拟光感。采用置换贴图渲染时，会使用置换贴图中的数据真实地修改模型的顶点位置，如在图 3-10 中，通过使用置换贴图，地球不再是一个完美的圆，而是呈现凸凹不平

的地貌。置换贴图也是灰度图，黑色到中灰会造成顶点凹陷，而中灰到白色则会使顶点外凸，具体的值反映凸凹的程度。

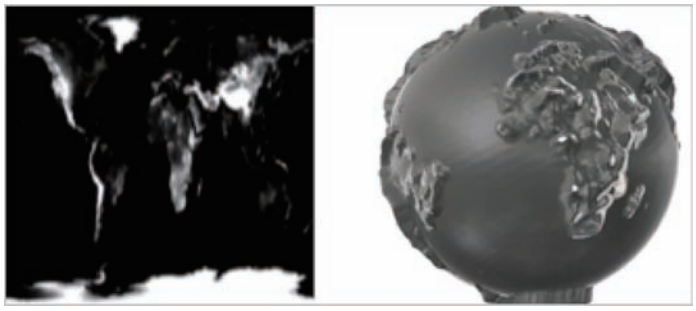

图 3-10　置换贴图用于位移模型顶点

9. 金属度贴图(Metalness Map，Metallic Map)

金属在光谱区有很强的光学吸收，而同时又有很大的反射率，金属值(Metal)用于描述物体表现的折射、反射、菲涅耳反射(Fresnel Reflection)，表现金属质感或高导电率外观。金属感贴图通常是黑白二值灰度图，黑色表示完全非金属，而白色表示全金属，如图 3-11 所示。

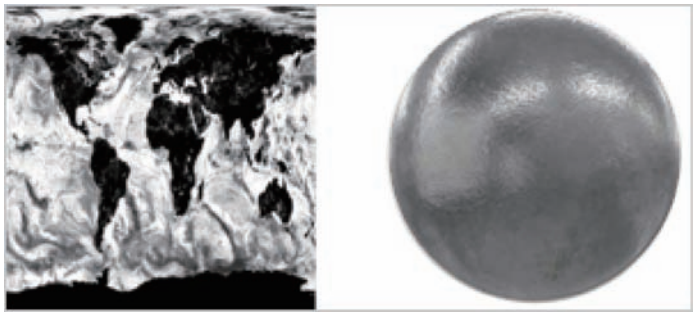

图 3-11　金属度贴图用于模拟物体的金属质感

10. 粗糙度贴图(Roughness Map)

粗糙度贴图用于模拟物体表面的粗糙与光滑表现，也是灰度图，白色表示粗糙表面，而黑色表示光滑表面，效果如图 3-12 所示。

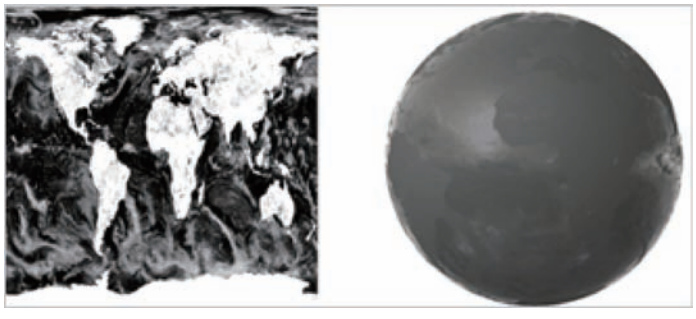

图 3-12　粗糙度贴图模拟物体表面的粗糙度质感

PBR 渲染中还有凸凹贴图(Bump Map)、高光贴图(Specular Map)、光泽度贴图(Glossiness Map)、视差贴图(Parallax Map)等贴图类型，由于这些贴图在 RealityKit 中不被支持，不再赘述，如有需要，读者需自行

查阅相关文档。

传统渲染中通常使用经验光照模型渲染物体，效果偏向理想化，而基于 PBR 的渲染则更倾向于依据物理规律对物体进行渲染，相对而言效果真实度更高、可信度更强，一个使用 PBR 渲染的地球如图 3-13 所示。

图 3-13　使用 PBR 渲染的地球

提示

PBR 渲染又分为 Metalness＋Roughness 工作流与 Specular＋Glossiness 工作流，RealityKit 支持 Metalness＋Roughness 工作流。传统渲染与 PBR 渲染、PBR 两种工作流中使用的贴图名字虽然相同，但实际贴图内容有很大区别，如有的会将阴影、高光存储到漫反射贴图中，有的则会要求剥离这些信息，在使用时需要根据具体需求对贴图进行调整。目前，RealityKit 与 Reality Composer 都不支持直接使用 PBR 贴图，只能通过导入使用了 PBR 贴图的 USDZ 或者 Reality 文件间接使用。

RealityKit 支持完整的 PBR 渲染，但目前通过程序化方式只能使用 SimpleMaterial、UnlitMaterial、OcclusionMaterial、VideoMaterial 4 种材质类型，而且支持的贴图非常有限，具体如表 3-1 所示。

表 3-1　RealityKit 可以程序化生成的材质

材　质	描　述
SimpleMaterial	该材质支持设置颜色(color)或者纹理(texture)类型的漫反射贴图、标量类型的 metallic、标量类型的 roughness，可供设置使用的纹理非常有限
UnlitMaterial	该材质只支持设置颜色(color)或者纹理(texture)类型的漫反射贴图，其特点是不参与光照计算，通常用于模拟带自发光属性的电视屏幕或者广告牌匾
OcclusionMaterial	该材质不需要设置贴图，其特点是遮挡应用该材质物体背后的物体，实现遮挡效果
VideoMaterial	该材质使用视频作为动态贴图，利用该材质可以实现动态的纹理效果

3.2　网格

在计算机图形学中，所有物体表面都由三角形组成的网格模拟(也可能包括点与直线)，由于三角形是最简单的基本图元，三角网格能在视觉精度和处理速度之间取得良好的平衡，随着计算机图形硬件的快速发展，目前已经能够每秒处理数百万甚至数千万三角形的渲染。

网格是 3D 虚拟世界中最基本的元素，有了网格，我们就可以进行渲染、碰撞检测等后续操作。三角网

格由三角形组成，而三角形由三个顶点按照一定的环绕方向确定，在计算机存储时，只存储所有的离散顶点及这些离散顶点间的关系，如图 3-14 所示。

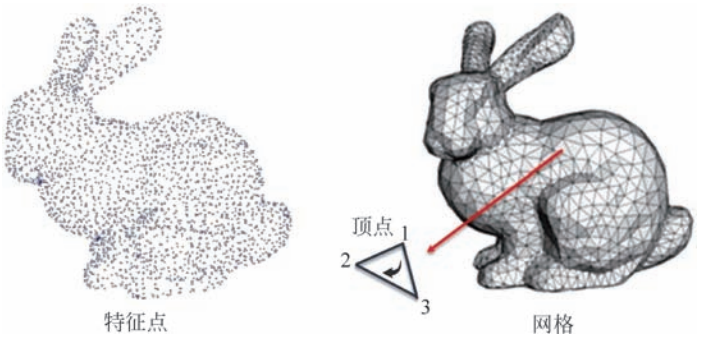

图 3-14　由离散顶点按照一定的规律构建成三角网格

在图 3-14 中，左图是离散顶点示意图，右图是由离散顶点构建的三角网格，在 GPU 渲染网格时，组成网格的三角形的 3 个顶点的位置是有序排列的，通常将如图 3-14 中所示的逆时针排序方式朝外的一面称为正面，朝内的一面称为背面，排序方式对后续的渲染、光照、背面剔除会产生很大的影响。在图 3-14 中，132 排列与 123 排列得到的三角形面的法线相反，因此阴影、光照、背面剔除也相反，当然，现在的建模软件会自动处理排序问题，但如果需要手动建立网格，或者由算法生成网格，则应当注意顶点环绕方向。

3.3　模型

模型是在计算机中对物体三维结构的数字表达，通常为众多点、线、三角形构成的网格，是一组能反映物体三维结构的网格。模型通常由三维建模软件制作生成，简单的模型也可以程序化生成，在 RealityKit 中，程序化模型生成由 MeshResource 类负责管理，目前可以生成立方体、球体、平面、文字 4 种类型的模型网格，如表 3-2 所示。

表 3-2　RealityKit 可以程序化生成的网格

网格生成方法	描　　述
generateBox()	3 个重载，支持生成带圆角的正方体、长方体网格
generatePlane()	3 个重载，支持在 *XY* 平面或者 *XZ* 平面内生成带圆角的平面网格
generateShpere()	1 个方法，支持按半径生成球体网格
generateText()	1 个方法，支持英文及汉字 3D 文字网格生成

在 RealityKit 中，程序化方式可以生成最基本的模型网格，在实际应用开发中，更多会导入由第三方软件生成的模型。目前，RealityKit 支持 USD 模型文件（包括. usd、. usda、. usdc、. usdz 文件）和 Reality 文件（. reality 文件，Reality 文件是由 Reality Composer 应用生成并导出的文件），Reality 格式由 USDZ 格式文件发展而来，通常可以提供更好的纹理压缩、动画、特效，性能相对而言更好。

RealityKit 允许从工程文件资源或者网络下载的 Bundle 中加载模型，并在 Entity 实体类中提供了一系列方法用于加载模型。由于模型文件大小不一、层次各异，RealityKit 同时提供了同步与异步两种加载方法。

1. 同步加载模型

同步加载使用 Entity.load(named:in:)方法从工程文件资源或者网络下载的 Bundle 中加载模型，同步加载方法会阻塞应用程序主线程直到模型加载完成，在模型文件较小、对模型实时性要求很高的情况下可以采用同步加载的方法，典型的使用方法如代码清单 3-1 所示。

代码清单 3-1

```
let entity = try? Entity.load(named: "MyEntity")
```

load(named:in:)方法也允许从工程文件资源中指定的位置加载模型，方法是先创建 URL 指定路径再进行模型加载，典型的使用方法如代码清单 3-2 所示。

代码清单 3-2

```
let url = URL(fileURLWithPath: "path/to/MyEntity.usdz")
let entity = try? Entity.load(contentsOf: url)
```

Entity 实体类的 load(named:in:)方法会加载模型的完整层次并返回根(root entity)实体对象，即如果一个模型文件包含很多层级，该方法会加载所有层级并保留它们相互之间的层级关系，如图 3-15 所示，RealityKit 中的所有模型加载方法都会遵循同样的原则，这允许我们使用具有复杂层次结构的模型，在加载成功后，可以使用 HasHierarchy 协议定义的方法访问各层级详细信息。

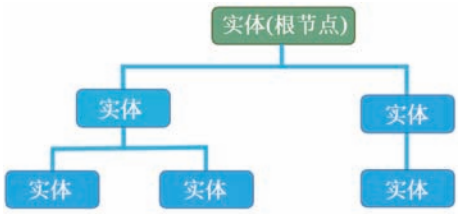

图 3-15 RealityKit 加载模型时会加载模型的完整层次结构

2. 异步加载模型

异步加载会在新创建的线程中执行模型加载操作，不会阻塞主线程，因此，为使用户界面更流畅，对模型文件较大、通过网络进行加载时应当使用异步加载方式，在 RealityKit 中，所有同步加载方法都有一个对应的异步加载方法，如 load(named:in:)方法的异步加载方法为 loadAsync(named:in:)，典型的异步加载模型用法如代码清单 3-3 所示。

代码清单 3-3

```
_ = Entity.loadAsync(named: "MyEntity")
    .sink(receiveCompletion: { loadCompletion in
        //错误处理
    }, receiveValue: { entity in
        //加载成功,进入正常处理流程
    })
```

异步加载操作返回一个 LoadRequest 类型实例，可以通过 sink(receiveCompletion:receiveValue:)使用闭包的方法处理返回结果。异步加载需要导入 Combine 框架，该框架提供了很多实用的加载处理方法，可以方便地处理各种异步加载需求，如可以使用 append(_:)和 collect()方法合并处理多个模型加载请求，如代码清单 3-4 所示。

代码清单 3-4

```
1.  _ = Entity.loadAsync(named: "MyEntity")
2.      .append(Entity.loadAsync(named: "MyOtherEntity"))
3.      .append(Entity.loadAsync(named: "MyThirdEntity"))
4.      .collect()
5.      .sink(receiveCompletion: { loadCompletion in
6.          //错误处理
7.      }, receiveValue: { entities in
8.          //处理整个集合
9.      })
```

load(named:in:)和 loadAsync(named:in:)方法不仅会加载模型网格,也会加载模型内置的骨骼动画信息,返回 Entity 类型实例,通常这是我们所希望的。加载模型也可以使用 loadModel(named:in:)方法,该方法会展平(flatten)模型结构,返回 ModelEntity 类型实例,更简单也更便于使用,并且模型使用效率更高。

除此之外,Entity 类还提供很多其他加载方法,分别用于从本地资源、网络中加载不同类型的资源,具体如表 3-3 所示。

表 3-3 RealityKit 提供的各类加载方法

方 法	类型	描 述
load(named:String,in:Bundle?)->Entity	同步	加载实体
load(contentsOf:URL,withName:String?)->Entity	同步	加载实体
loadAsync(named:String,in:Bundle?)->LoadRequest<Entity>	异步	加载实体
loadAsync(contentsOf:URL,withName:String?)->LoadRequest<Entity>	异步	加载实体
loadModel(named:String,in:Bundle?)->ModelEntity	同步	加载模型
loadModel(contentsOf:URL,withName:String?)->ModelEntity	同步	加载模型
loadModelAsync(named:String,in:Bundle?)->LoadRequest<ModelEntity>	异步	加载模型
loadModelAsync(contentsOf:URL,withName:String?)->LoadRequest<ModelEntity>	异步	加载模型
loadAnchor(named:String,in:Bundle?)->AnchorEntity	同步	加载 ARAnchor
loadAnchor(contentsOf:URL,withName:String?)->AnchorEntity	同步	加载 ARAnchor
loadAnchorAsync(named:String,in:Bundle?)->LoadRequest<AnchorEntity>	异步	加载 ARAnchor
loadAnchorAsync(contentsOf:URL,withName:String?)->LoadRequest<AnchorEntity>	异步	加载 ARAnchor
loadBodyTracked(named:String,in:Bundle?)->BodyTrackedEntity	同步	加载人体骨骼实体
loadBodyTracked(contentsOf:URL,withName:String?)->BodyTrackedEntity	同步	加载人体骨骼实体
loadBodyTrackedAsync(named:String, in:Bundle?) -> LoadRequest<BodyTrackedEntity>	异步	加载人体骨骼实体
loadBodyTrackedAsync(contentsOf:URL,withName:String?)->LoadRequest<BodyTrackedEntity>	异步	加载人体骨骼实体

在表 3-3 中,loadAnchor(named:in:)方法可以直接从文件中加载 ARAnchor 类的实体结构,该方法与 Entity.load(named:in:)方法的区别是,其返回 AnchorEntity 实例,可以直接添加到 Scene 节点中,典型用法如代码清单 3-5 所示。

代码清单 3-5

```
1.  if let anchor = try? Entity.loadAnchor(named: "MyEntity") {
2.      arView.scene.addAnchor(anchor)
3.  }
```

RealityKit 加载一个拥有很多层级的文件时，如果不需要获取其内部结构信息，可以将其内部和兄弟节点的层级展平到一个单一的实体中，相比于 load() 方法，使用 loadModel(named:in:) 或者 loadBodyTracked(named:in:)方法时会自动展平加载的实体对象，一个展平的实体更有利于使用和提高处理效率。

当然，展平后的实体将无法使用程序的方式访问内部层级，但很多时候，我们并不需要访问模型实体的内部层级，因此在大部分情况下，展平有利于提高性能。

利用 RealityKit 提供的方法，可以轻松加载各类模型，下面我们以同步及异步加载模型为例，演示如何进行模型加载，具体代码如代码清单 3-6 所示。

代码清单 3-6

```
1.  extension ARView : ARSessionDelegate{
2.      func loadModel(){
3.          let planeAnchor = AnchorEntity(plane:.horizontal)
4.          //同步加载
5.          do {
6.              let usdzPath = "toy_drummer"
7.              let modelEntity =   try ModelEntity.loadModel(named: usdzPath)
8.              print("加载成功!")
9.              planeAnchor.addChild(modelEntity)
10.         } catch {
11.             print("找不到文件")
12.         }
13.         //异步加载
14.         let usdzPath = "toy_drummer"
15.         var cancellable: AnyCancellable? = nil
16.         cancellable = ModelEntity.loadModelAsync(named: usdzPath)
17.             .sink(receiveCompletion: { error in
18.                 print("发生错误: \(error)")
19.                 cancellable?.cancel()
20.             }, receiveValue: { entity in
21.                 planeAnchor.addChild(entity)
22.                 cancellable?.cancel()
23.             })
24.         self.scene.addAnchor(planeAnchor)
25.     }
26. }
```

3.4 动画

动画是增强虚拟元素真实感和生动性的重要方面，RealityKit 支持变换动画(Transform Animation)和骨骼动画(Skeletal Animation)两种动画模式。变换动画一般程序化地执行，支持基本的平移、旋转、缩放，更复杂的动画通常由第三方模型制作软件采用骨骼绑定的方式生成，独立或者内置于模型文件中。USDZ 和 Reality 文件格式都支持动画，在使用时，可以直接由该类文件将动画导入场景中。

变换动画可以实现对虚拟元素常见的基本操作，如平移、旋转、缩放，在执行时，通常使用实体类的move(to:relativeTo:duration:)方法，该方法参数 duration 用于指定动画时间，基本使用方法如代码清单 3-7 所示。

代码清单 3-7

```
var cubeEntity : ModelEntity?
var gestureStartLocation: SIMD3<Float>?

extension ARView :ARSessionDelegate{
    func createPlane(){
        let planeAnchor = AnchorEntity(plane:.horizontal)
        do {
            let cubeMesh = MeshResource.generateBox(size: 0.1)
            var cubeMaterial = SimpleMaterial(color:.white,isMetallic: false)
            cubeMaterial.baseColor = try .texture(.load(named: "Box_Texture.jpg"))
            cubeEntity = ModelEntity(mesh:cubeMesh,materials:[cubeMaterial])
            cubeEntity!.generateCollisionShapes(recursive: false)
            cubeEntity?.name = "this is a cube"
            planeAnchor.addChild(cubeEntity!)
            self.scene.addAnchor(planeAnchor)
            self.installGestures(.all,for:cubeEntity!).forEach{
                $0.addTarget(self, action: #selector(handleModelGesture))
            }
        } catch {
            print("找不到文件")
        }
    }

    @objc func handleModelGesture(_ sender: Any) {
        switch sender {
        case let rotation as EntityRotationGestureRecognizer:
            rotation.isEnabled = false
            var transform = rotation.entity!.transform
            transform.rotation =  simd_quatf(angle: .pi*1.5, axis: [0, 1, 0])
            rotation.entity!.move(to: transform, relativeTo: nil, duration: 5.0)
            rotation.isEnabled = true
        case let translation as EntityTranslationGestureRecognizer:
            translation.isEnabled = false
            var transform = translation.entity!.transform
            transform.translation = SIMD3<Float>(x: 0.8, y: 0, z: 0)
            translation.entity!.move(to:transform,relativeTo:nil,duration:5.0)
            translation.isEnabled = true
        case let Scale as EntityScaleGestureRecognizer:
            Scale.isEnabled = false
            var scaleTransform = Scale.entity!.transform
            scaleTransform.scale = SIMD3<Float>(x: 2, y: 2, z: 2)
            Scale.entity!.move(to:scaleTransform,relativeTo:nil,duration:5.0)
            Scale.isEnabled = true
        default:
```

```
            break
        }
    }

    @objc func handleScaleGesture(_ sender : EntityScaleGestureRecognizer){
        print("in scale")
    }
}
```

在代码清单 3-7 中，我们对平移、旋转、缩放的变换动画都进行了演示，在使用 move()方法进行变换动画之前，应当先设置需要达到的目标，利用 duration 参数控制动画时长。

move()方法另一个重载 move(to:relativeTo:duration:timingFunction:)版本，其参数 timingFunction 为 AnimationTimingFunction 类型，通过它可以指定动画效果，如线性(linear)、缓入(easeIn)、缓出(easeOut)、缓入缓出(easeInOut)、三次贝赛尔曲线(cubicBezier)，通过使用该方法可以改善动画体验。

变换动画只适合于执行相对简单的动画操作，如控制灯光沿圆形轨道移动、用户单击模型时出现弹跳效果等，对于复杂的动画，一般使用第三方软件(Maya、3ds MAX 等)预先制作好骨骼动画，然后导出为 USDZ 或 Reality 格式文件供 ARKit 使用。在 RealityKit 中，使用骨骼动画的典型代码如代码清单 3-8 所示。

代码清单 3-8

```
extension ARView : ARSessionDelegate{
    func CreateRobot(){
        let planeAnchor = AnchorEntity(plane:.horizontal)
        do {
            let robot =   try ModelEntity.load(named: "toy_drummer")
            planeAnchor.addChild(robot)
            robot.scale = [0.01,0.01,0.01]
            self.scene.addAnchor(planeAnchor)
            print("Total animation count : \(robot.availableAnimations.count)")
            robot.playAnimation(robot.availableAnimations[0].repeat())
        } catch {
            print("找不到 USDZ 文件")
        }
    }
}
```

在代码清单 3-8 中，我们在 load()方法加载 USDZ 文件后，使用实体类的 playAnimation()方法播放骨骼动画。需要注意的是，在播放动画之前应当先通过实体的 availableAnimations.count 检查加载的模型所包含的动画数量，再确定播放哪个可用动画。playAnimation()方法返回一个 AnimationPlaybackController 类实例，可以利用这个实例控制动画暂停(pause)、继续播放(resume)、停止(stop)，也可以使用 stopAllAnimations()方法停止所有动画播放。

3.5　RealityKit 渲染

在计算机图形学中，3D 渲染又称为着色(Shading)，是指对 3D 模型进行纹理与光照处理并光栅化成像素，以可视化图像的方式展示数字场景的过程。图 3-16 直观地展示了 3D 渲染过程。在 3D 渲染中，顶点着色器(Shader)从来没有真正渲染过线框模型，相反，它只是定位并对顶点进行着色，然后输入到片元着色器进行光照计算和阴影处理，3D 渲染的最后一步称为光栅化，即生成每个像素的颜色信息，随后这些像素被输出到帧缓存中并由显示器进行显示。

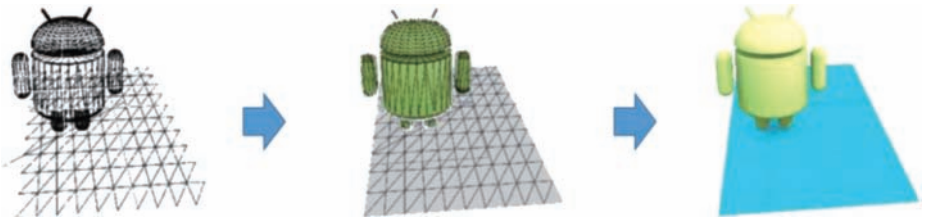

图 3-16　AR 渲染物体的过程

提示

这里讨论的渲染过程是指利用 DirectX 或 OpenGL 在设备的 GPU 上进行标准的实时渲染过程，目前已有一些渲染方式采用另外的渲染架构，但那不在我们的讨论范围之内。

3.5.1　立方体贴图

立方体贴图(Cubemap)由于其独特的性质通常用于环境映射，也常被用于具有反射属性物体的反射源，在 RealityKit 中，内置了一个简单的天空盒(Skybox，在 RealityKit 中实质上使用了 IBL 技术实现，第 6 章会详细介绍)，所有带反射材质的物体默认会对天空盒产生反射，如图 3-17 所示，天空盒的实现使用了立方体贴图，也常称为环境贴图。

图 3-17　RealityKit 内置了一个天空盒用于基础反射

立方体贴图是一个由 6 个独立的正方形纹理组成的纹理集合，包含了 6 个 2D 纹理，每个 2D 纹理为立方体的一个面，6 个纹理组成一个有贴图的立方体，如图 3-18 所示。

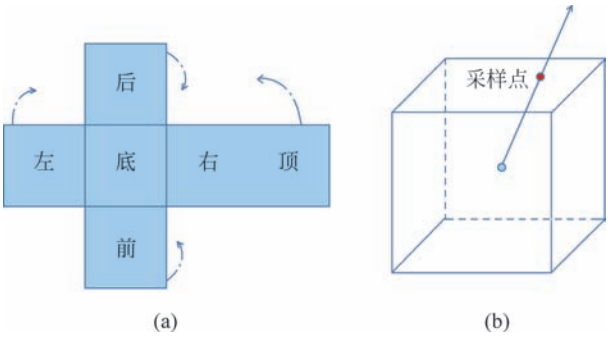

图 3-18　立方体贴图展开与采样示意图

在图 3-18(a)中，沿着虚线箭头方向可以将这 6 个面封闭成一个立方体，形成一个纹理面向内的贴图集合，这也是立方体贴图名字的由来。立方体贴图最大的特点是构成了一个 720°全封闭空间，因此如果组成立方体贴图的 6 张纹理选择连续无缝贴图就可以实现 720°无死角的纹理采样，形成完美的天空盒效果。

与 2D 纹理采样使用 UV 坐标不同，立方体贴图需要一个 3D 查找向量进行采样，查找向量是一个原点位于立方体中心点的 3D 向量，如图 3-18(b)所示，在 3D 找查向量与立方体相交处的纹理就是需要采样的纹理。在 GLSL、HLSL、Cg、Metal 中都定义了立方体贴图采样函数，可以非常方便地进行立方体贴图的采样操作。

立方体贴图因其 720°封闭特性常用来模拟在某点处的周边环境实现反射、折射效果，如根据赛车位置实时更新立方体贴图可以模拟赛车车身对周边环境的反射效果。在 AR 中，我们也是利用同样的原理实现虚拟物体对真实环境的反射，在 RealityKit 中实现实时环境反射方法将在第 6 章讲述。

立方体贴图需要 6 个无缝纹理，使用静态纹理可以非常好地模拟全向场景，但静态纹理不能反映动态的物体变化，如赛车车身对周围环境的反射，如果使用静态纹理将不能反射路上行走的人群和闪烁的霓虹灯，这时就需要使用实时动态生成的立方体贴图，这种方式能非常真实地模拟赛车对环境的反射，但性能开销比较大，需要谨慎使用。

3.5.2　PBR 渲染

就算法理论基础而言，光照模型分为两类：一类基于物理理论，另一类基于经验模型。基于物理理论的光照模型，偏重于使用物理的度量和统计方法，比较典型的有 ward BRDF 模型，其中不少参数需要由仪器测量，使用这种光照模型的好处是效果非常真实，但是计算复杂，实现起来也较为困难。PBR(Physically Based Rendering，基于物理渲染)渲染也是基于物理模型，PBR 对物体表面采用微平面进行建模，利用辐射度，加上光线追踪技术进行渲染，但 PBR 渲染并不是纯物理渲染，也使用了部分简化模型。经验模型更加偏重于使用特定的概率公式，使其与一组表面材质类型相匹配，所以经验模型大多比较简洁，但效果偏向理想化。物理模型与经验模型两者之间的界限并非清晰到“非黑即白”的程度，无论何种光照模型本质上还是基于物理的，只不过在求证方法上各有偏重。通常来讲，经验模型更简单、对计算更友好，而物理模型更复杂且计算量更大，但效果会更真实。

PBR 渲染是在不同程度上都与现实世界的物理原理更相符的基本理论所构成的渲染技术的集合。正因为基于物理的渲染目的是为了使用一种更符合物理学规律的方式来模拟光线交互，因此这种渲染方式与传统使用的 Phong 或者 Blinn-Phong 光照模型算法相比总体上看起来要更真实一些。除了看起来更真实以

外，由于使用物理参数调整模拟效果，因此可以编写出通用的算法，通过修改物理参数模拟不同的材质表面，而不必依靠经验修改或调整让光照效果看上去更自然。使用基于物理参数的方法编写材质还有一个更大的好处，就是无论光照条件如何，这些材质看上去都会是正确的，如 Unity 引擎内置的 Standard Shader 就是一个万能的基于物理的着色器，可以通过不同的参数设置模拟各种材质表面属性，从木质到金属都可以仅由一个着色器模拟。在使用 PBR 进行渲染时，可以通过调整 Metallic 和 Smoothness 值模拟从非金属到金属的所有材质，如图 3-19 所示。

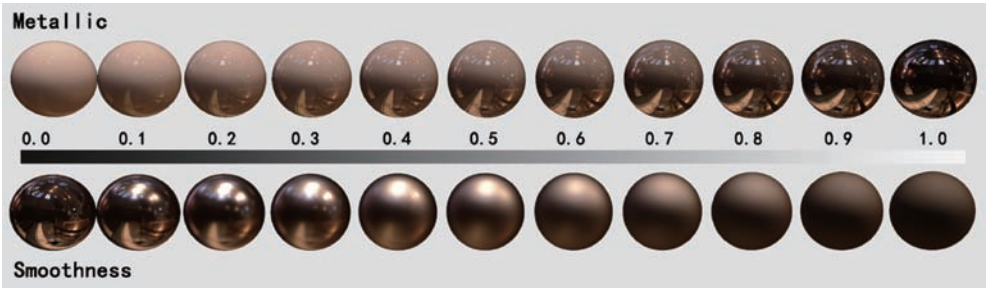

图 3-19　PBR 流程中 Metallic 与 Smoothness 对材质外观的质感影响

虽然如此，基于物理的渲染仍然只是对现实世界物理规律的一种近似，这也就是为什么它被称为基于物理的渲染而非物理渲染的原因。判断一种 PBR 光照模型是否基于物理，必须满足以下 3 个条件：基于微平面(Microfacet)的表面模型；能量守恒；应用基于物理的 BRDF。

3.5.3　清漆贴图

清漆贴图(Clear Coat)是一种高级材质贴图，清漆贴图源于瓷器工艺，通常在瓷器制作完成后会在其表面再刷一层清漆用作保护，刷过清漆的瓷器表面会呈现一种玻璃样通透的质感，如图 3-20 所示，所以清漆贴图通常用于模拟光滑玻璃样表面。

RealityKit 支持清漆贴图与清漆粗糙度贴图(Clear Coat Roughness)，在具体执行中，RealityKit 对粗糙度采用多次散射(Multi-scattering)算法，对材质表面粗糙度的模拟更细致，结合清漆粗糙度贴图，可以精细地模拟物体表面的散光，效果更真实可信，如图 3-21 所示。

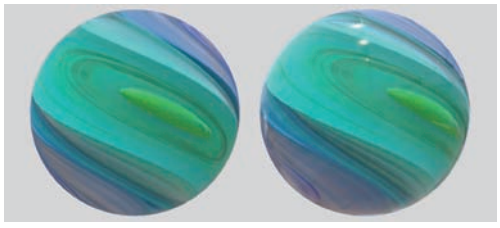

图 3-20　清漆对材质外观质感的影响

图 3-21　多次散射与清漆贴图使用对物体外观表现对比

清漆贴图和清漆粗糙度贴图对物体表面的影响如图 3-22 所示，它们的取值范围均为[0,1]，从 0 到 1，清漆贴图对物体表面影响越来越弱，清漆粗糙度贴图则会让物体对光照反射越来越模糊。

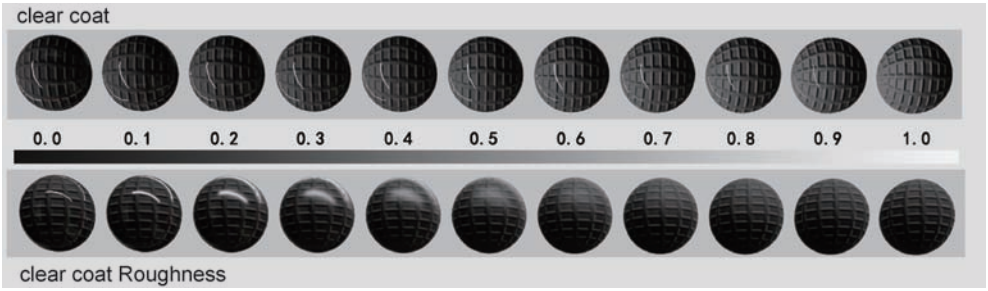

图 3-22　清漆贴图与清漆粗糙度贴图对物体表面质感的影响

RealityKit 中 PBR 渲染支持清漆贴图和清漆粗糙度贴图这两种贴图，与其他材质结合使用，可以让模型更生动真实。

功能技术篇

本篇为 ARKit 技术各功能点详细讨论篇，对 ARKit 各个功能技术点进行全面深入的剖析，在讲述功能点时，特别注重技术的实际应用，每一个功能点都配有详尽的可执行代码及代码的详细说明。

功能技术篇包括以下章节。

第 4 章　图像与物体检测跟踪

对 2D 图像与 3D 物体的检测识别跟踪进行阐述，并对检测跟踪过程中的性能优化、注意事项进行讨论。

第 5 章　人脸检测跟踪

对人脸检测、人脸表情捕捉、人脸特效相关技术进行讨论，并实现同时开启前后摄像头，利用前置摄像头捕捉的人脸表情信息驱动后置摄像头 AR 场景中模型的功能。

第 6 章　光影特效

对光照模型、光照一致性、光照估计、环境光反射等光影特效相关知识进行学习，讨论 AR 中实现光照估计和环境反射的原理及基本步骤。

第 7 章　肢体动捕与人形遮挡

对 2D、3D 人体姿态估计及人形遮挡相关知识进行阐述，实现人体动作捕捉、利用捕捉的人体骨骼关节点信息驱动模型、人形遮挡、人形区域提取等功能。

第 8 章　持久化存储与多人共享

对锚点、持久化存储、AR 多人体验共享相关原理进行学习，重点对 ARWorldMap、协作 Session 及 RealityKit 中同步共享技术进行深入探究。

第 9 章　物理模拟

对在 RealityKit 中利用物理引擎进行物理模拟进行深入探讨，并对触发器及触发域的使用进行阐述。

第 10 章　Reality Composer

详细介绍 Realtiy Composer 使用方法、操作技巧，对自定义行为中触发器和动作序列使用进行详细说明，对 Reality Composer 与 Xcode 代码交互进行深入探究。