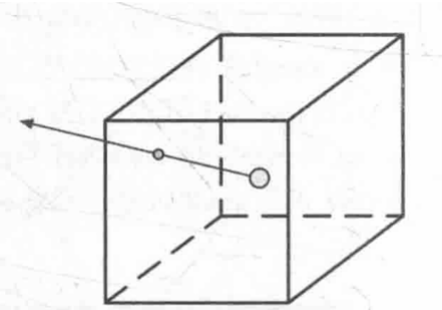
Lecture10 高级纹理

1. 立方体纹理 Cubemap

立方体纹理 Cubemap 是环境映射 Environment Mapping 的一种实现方式

它包含 6 张图像，对应立方体的 6 个面，我们对立方体的采样需要提供一个三维的纹理坐标

这个纹理坐标表示我们在世界空间下的一个 3D 方向，这个方向矢量从立方体的中心出发，当它向外延伸时，就会和立方体的 6 个纹理之一发生相交，采样结果就是由这些交点计算出来的



▲图 10.1　对立方体纹理的采样

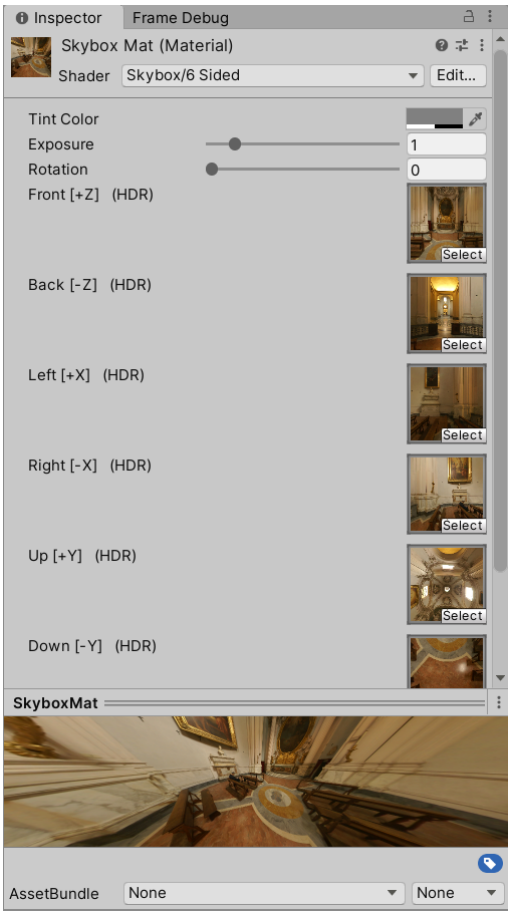
- 优点：简单快速，效果较好，可以反射环境
- 缺点：场景中引入新的物体、光源或者物体移动时，需要重新生成立方体纹理，不能反射使用该立方体的物体本身（不能模拟多次反射的结果）

天空盒 Skybox

模拟天空，整个场景被包裹在一个立方体内，在 Unity 中，天空盒是在所有不透明物体之后渲染的

在 Unity 中，使用天空盒非常简单，创建一个 Skybox 材质，把它赋值给场景的相关设置

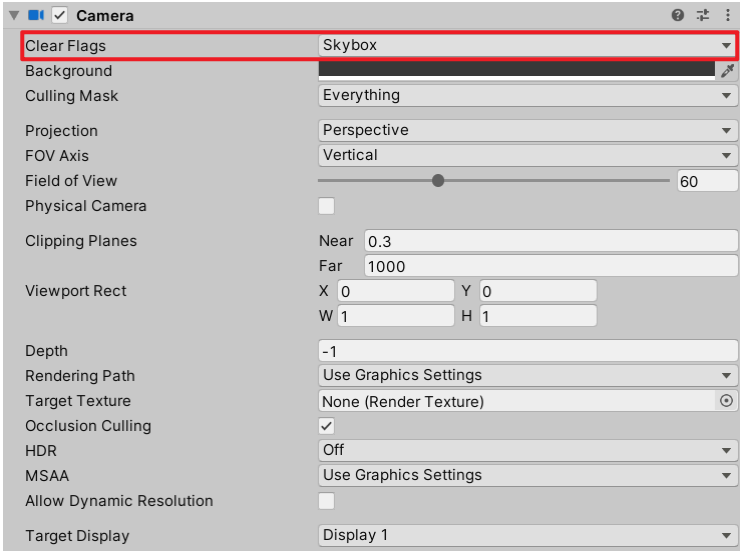
为了正确渲染天空盒，需要将 6 张图片的 Wrap Mode 设置为 **Clamp**，防止接缝处不匹配



除此之外还有 3 个属性

- Tine Color：控制材质整体颜色
- Exposure：控制天空盒亮度
- Rotation：控制天空盒沿 y 轴方向的旋转角度

为了让摄像机正常渲染天空盒，需要保证场景的 Camera 组件中的 Clear Flags 被设置为 **Skybox**



- 在 Window/Rendering/Lighting/Environment/Skybox Material 中设置的天空盒会应用到该场景的所有摄像机
- 如果我们希望某些摄像机使用不同的天空盒，可以向该摄像机添加 Skybox 组件覆盖掉之前的设置

环境映射 Environment Mapping

环境映射可以模拟出金属质感的材质，通常有几种创建环境映射

特殊布局纹理创建

提供一张具有特殊布局的纹理，然后将该纹理的 Texture Type 设置为 Cubemap 即可

基于物理的渲染中，通常会使用一张 HDR 图像来生成高质量的 Cubemap

通过脚本生成

之前的方法需要准备好立方体的纹理图像，我们希望物体根据场景中位置的不同，生成各自不同的立方体纹理

这可以使用 Unity 提供的 `Camera.RenderToCubemap` 函数实现，它能把从任意位置观察到的场景图像存储在 6 张图像中

本代码在 `RenderCubemapWizard.cs`

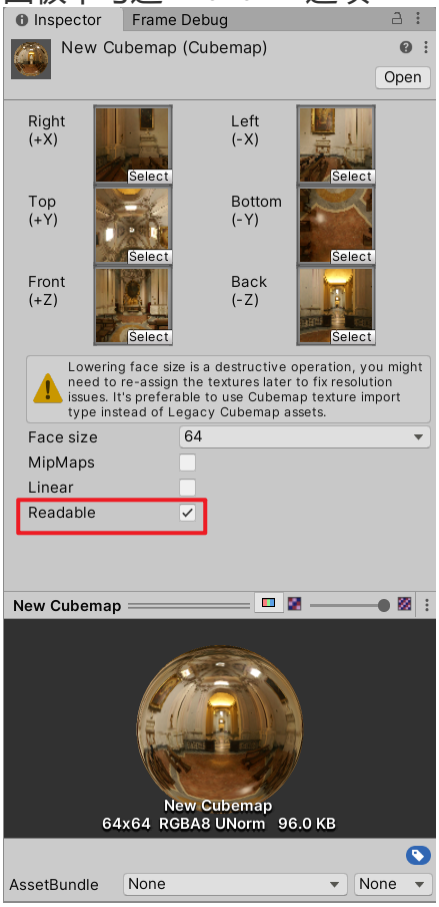
```
void OnWizardCreate () {
    // create temporary camera for rendering
    GameObject go = new GameObject( "CubemapCamera");
    go.AddComponent<Camera>();
    // place it on the object
    go.transform.position = renderFromPosition.position;
    // render into cubemap
    go.GetComponent<Camera>().RenderToCubemap(cubemap);
    // destroy temporary camera
    DestroyImmediate( go );
}
```

由于本代码需要添加菜单栏条目，因此我们把它放在 Editor 文件夹下才能正确执行

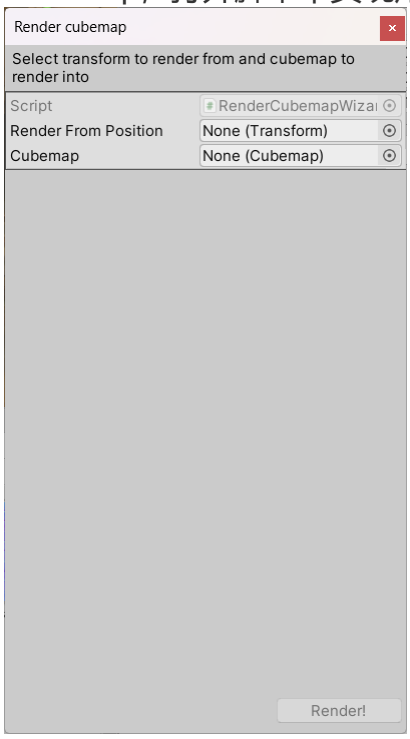
准备好上述代码后，进行如下步骤

1. 在有天空盒渲染的 Scene 下创建一个空的 GameObject 对象
2. 新建一个用于存储立方体的纹理，在 Project 视图下 Create -> Legacy -> Cubemap 创建

- 为了能让脚本顺利将图像渲染到该立方体纹理中，我们需要在它的面板中勾选 Readable 选项



3. Hierarchy 菜单中选中 GameObject 右键后选择 Render into Cubemap，打开脚本中实现用于渲染立方体纹理的窗口



4. 把创建的 GameObject 和 Cubemap 分别拖入 Render From Position 和 Cubemap 选项，点击窗口的 Render 按钮，即可完成渲染
5. 注意，Cubemap 的大小 Face size 选项值越大，渲染的立方体纹理越大，效果越好，占用内存越大

反射 Reflection

使用反射效果的物体通常看起来像镀了层金属，模拟反射效果只需要通过入射光线的方向和表面法线的方向来计算反射方向，再利用反射方向对立方体进行采样即可

```
Shader "MyShader/Light/S_14_ReflectionShader"
{
    Properties
    {
        _Color ("Color", Color) = (1,1,1,1)
        _ReflectColor("Reflection Color", Color) =
(1,1,1,1) // 反射颜色
        _ReflectAmount("Reflect Amount", Range(0,1)) = 1 //
反射程度
        _Cubemap("Reflection Cubemap", Cube) = "_Skybox" {}
// 模拟反射的环境映射纹理
    }
    SubShader
    {

        Pass
        {
            // Pass for ambient light & first pixel light
(directional light)
            Tags
            {
                "LightMode"="ForwardBase"
            }

            CGPROGRAM
            // Apparently need to add this declaration
            // 正确赋值光照衰减等光照变量
            #pragma multi_compile_fwdbase

            #pragma vertex vert
            #pragma fragment frag

            #include "Lighting.cginc"
            #include "AutoLight.cginc"

            fixed4 _Color;
            fixed4 _ReflectColor;
            fixed _ReflectAmount;
```

```
            samplerCUBE _Cubemap;


            struct a2v
            {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
            };

            struct v2f
            {
                float4 pos : SV_POSITION;
                float3 worldPos : TEXCOORD0;
                float3 worldNormal : TEXCOORD1;
                fixed3 worldViewDir : TEXCOORD2;
                fixed3 worldRefl : TEXCOORD3;
                SHADOW_COORDS(4)
            };

            v2f vert(a2v v)
            {
                v2f o;

                o.pos = UnityObjectToClipPos(v.vertex);
                o.worldNormal =
UnityObjectToWorldNormal(v.normal);
                o.worldPos = mul(unity_ObjectToWorld,
v.vertex).xyz;
                o.worldViewDir =
UnityWorldSpaceViewDir(o.worldPos);

                // 通过CG的reflect函数实现计算顶点反射方向
                o.worldRefl = reflect(-o.worldViewDir,
o.worldNormal);

                TRANSFER_SHADOW(o);

                return o;
            }

            fixed4 frag(v2f i) : SV_Target
            {
```

```
                fixed3 worldNormal =
normalize(i.worldNormal);
                fixed3 worldLightDir =
normalize(_WorldSpaceLightPos0.xyz);
                fixed3 worldViewDir =
normalize(i.worldViewDir);

                fixed3 ambient =
UNITY_LIGHTMODEL_AMBIENT.xyz;
                fixed3 diffuse = _LightColor0.rgb *
_Color.rgb * max(0, dot(worldNormal, worldLightDir));

                // 片元着色器中，利用反射方向来对立方体进行纹理采
样，立方体纹理采样用到texCUBE函数
                fixed3 reflection = texCUBE(_Cubemap,
i.worldRefl).rgb * _ReflectColor.rgb;

                UNITY_LIGHT_ATTENUATION(atten, i,
i.worldPos);

                // Mix the diffuse color with the reflected
color
                fixed3 color = ambient + lerp(diffuse,
reflection, _ReflectAmount) * atten;

                return fixed4(color, 1.0);
            }
            ENDCG
        }
    }
    FallBack "Diffuse"
}
```

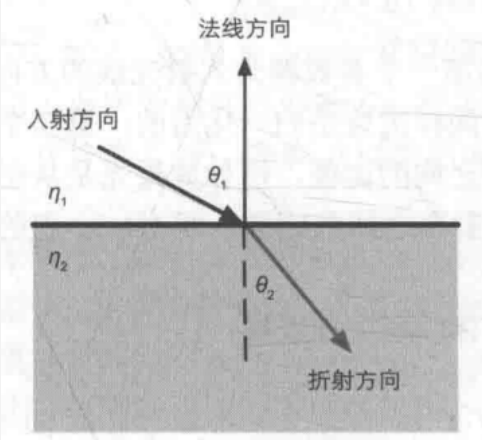
折射 Refraction

折射的原理比反射复杂一些，当光线从一种介质（例如空气）斜射入另一种介质（例如玻璃）时，传播方向会发生改变

当给定入射角时，我们可以使用**斯涅耳定律 Snell's Law** 计算反射角

斯涅耳定律 Snell's Law

当光从介质 1 沿着表面法线夹角为 θ_1 方向斜射入介质 2 的时候，我们可以使用如下公式计算折射光线与法线的夹角 θ_2



$$\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2$$

- η：两种介质的**折射率 index of reflection**，它是一个重要的物理常数，例如真空的折射率是 1，玻璃的折射率是 1.5

```
Shader "MyShader/Light/S_15_RefractionShader"
{
    Properties
    {
        _Color ("Color", Color) = (1,1,1,1)
        _RefractColor("Reflection Color", Color) =
(1,1,1,1) // 折射颜色
        _RefractAmount("Reflect Amount", Range(0,1)) = 1 //
折射程度
        _RefractRatio("Refraction Ratio", Range(0.1, 1)) =
0.5 // 透射比
        _Cubemap("Reflection Cubemap", Cube) = "_Skybox" {}
// 模拟反射的环境映射纹理
    }
    SubShader
    {

        Pass
        {
            // Pass for ambient light & first pixel light
(directional light)
            Tags
            {
                "LightMode"="ForwardBase"
            }

            CGPROGRAM
            // Apparently need to add this declaration
            // 正确赋值光照衰减等光照变量
```

```
            #pragma multi_compile_fwdbase

            #pragma vertex vert
            #pragma fragment frag

            #include "Lighting.cginc"
            #include "AutoLight.cginc"

            fixed4 _Color;
            fixed4 _RefractColor;
            fixed _RefractAmount;
            fixed _RefractRatio;
            samplerCUBE _Cubemap;


            struct a2v
            {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
            };

            struct v2f
            {
                float4 pos : SV_POSITION;
                float3 worldPos : TEXCOORD0;
                float3 worldNormal : TEXCOORD1;
                fixed3 worldViewDir : TEXCOORD2;
                fixed3 worldRefr : TEXCOORD3;
                SHADOW_COORDS(4)
            };

            v2f vert(a2v v)
            {
                v2f o;

                o.pos = UnityObjectToClipPos(v.vertex);
                o.worldNormal =
UnityObjectToWorldNormal(v.normal);
                o.worldPos = mul(unity_ObjectToWorld,
v.vertex).xyz;
                o.worldViewDir =
UnityWorldSpaceViewDir(o.worldPos);
```

```
                // 通过CG的refract函数实现计算顶点折射方向
                o.worldRefr = refract(-
normalize(o.worldViewDir), normalize(o.worldNormal),
_RefractRatio);

                TRANSFER_SHADOW(o);

                return o;
            }

            fixed4 frag(v2f i) : SV_Target
            {
                fixed3 worldNormal =
normalize(i.worldNormal);
                fixed3 worldLightDir =
normalize(_WorldSpaceLightPos0.xyz); //
_WorldSpaceLightPos0获取平行光方向（位置没有意义）

                fixed3 ambient =
UNITY_LIGHTMODEL_AMBIENT.xyz;
                fixed3 diffuse = _LightColor0.rgb *
_Color.rgb * max(0, dot(worldNormal, worldLightDir));
                // 片元着色器中，利用折射方向来对立方体进行纹理采
样，立方体纹理采样用到texCUBE函数
                fixed3 reflection = texCUBE(_Cubemap,
i.worldRefr).rgb * _RefractColor.rgb;

                UNITY_LIGHT_ATTENUATION(atten, i,
i.worldPos);

                // Mix the diffuse color with the reflected
color
                fixed3 color = ambient + lerp(diffuse,
reflection, _RefractAmount) * atten;

                return fixed4(color, 1.0);
            }
            ENDCG
        }
    }
    FallBack "Diffuse"
}
```

菲涅尔反射 Fresnel Reflection

在实时渲染中，我们经常使用菲涅尔反射来**根据视角方向控制反射程度**，菲涅尔反射描述了一种光学现象，当光线照射到物体表面时，一部分发生反射，一部分进入物体内部，发生折射。被反射的光和入射光之间存在一定的比率关系

- 一个例子是：当你站在湖边，直接低头看脚边的水面时，水几乎是透明的，你可以直接看清河底的小鱼和石子，但是，当你抬头看远处当你抬头看远处的水面时，会发现几乎看不到水下的情景，而只能看到水面反射的环境

在实时渲染中，我们选择一些近似公式来计算菲涅尔反射

Schlick 菲涅尔近似等式

$$F_{schlick}(\boldsymbol{v},\boldsymbol{n}) = F_0 + (1 - F_0)(1 - \boldsymbol{v} \cdot \boldsymbol{n})^5$$

- F_0: 反射系数，控制菲涅尔反射强度
- $\boldsymbol{v}$: 视角方向
- $\boldsymbol{n}$: 表面法线

Empricial 菲涅尔近似等式

$$F_{emprical}(\boldsymbol{v},\boldsymbol{n}) = \max(0, \min(1, bias + scale \times (1 - (\boldsymbol{v},\boldsymbol{n})^{power}))$$

- $bias, scale, power$: 控制项

2. 渲染纹理 Render Texture

之前的学习中，一个摄像机的渲染结果会输出到颜色缓冲中，并显示在我们屏幕上，

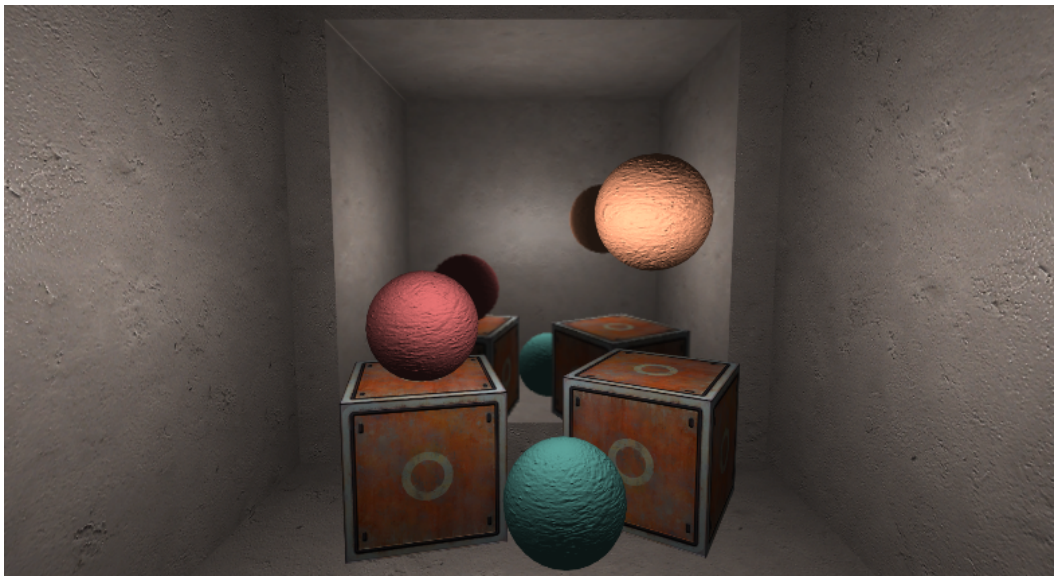
现代 GPU 允许我们使用

- **渲染目标纹理 Render Target Texture，RTT**：把整个三维场景渲染到颜色缓冲中，即，而不是传统的帧缓冲
- **多重渲染目标 Multiple Render Target， MRT**：GPU 允许我们把从场景同时渲染到多个渲染目标纹理中，不用再为每个目标纹理单独渲染完整的场景（延迟渲染）

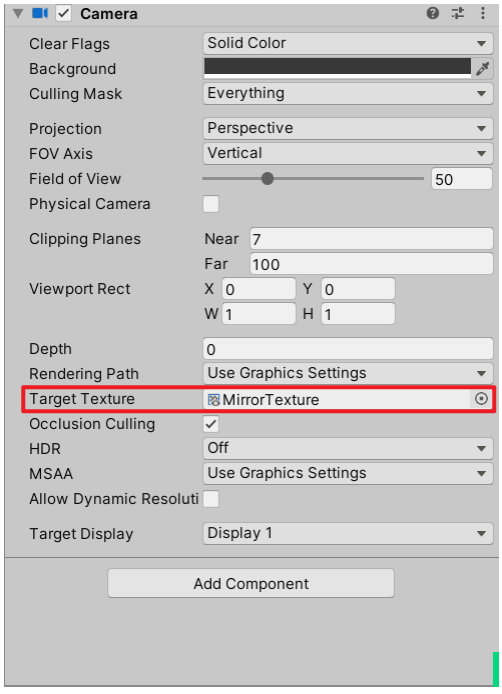
Unity 为渲染目标纹理定义了一种专门的**渲染纹理 Render Texture** 类型，通常有两种使用方式

- 在 Project 目录下创建一个渲染纹理，然后把摄像机的渲染目标设置成该渲染纹理（这样相机的渲染结果会实时更新到渲染纹理中），可以选择渲染纹理的分辨率、滤波模式等纹理属性
- 在屏幕后处理时使用 GrabPass 或者 OnRenderImage 函数获取当前屏幕图像，Unity 会把整个屏幕图像放到一张和屏幕分辨率等同的渲染纹理中，可以在自定义的 Pass 中将它当作普通的纹理处理，来实现各种屏幕特效

镜子效果



- 创建一个四边形 Quad，调整它的位置和大小，将它作为镜子
- 在 Project 视图下创建一个**渲染纹理**（右键单击 Create -> Render Texture）
- 为了从镜子出发观察到场景图像，我们需要创建一个摄像机，并且调整它的位置、裁剪平面、视角等，使得它现实的图像是我们希望的镜子图像，由于这个摄像机不需要直接显示在屏幕上，而是用于渲染到的纹理，因此我们把创建的渲染纹理拖拽到该摄像机的 **Target Texture** 上



镜子实现的原理很简单，使用一个渲染纹理作为输入属性，并把该渲染纹理在水平方向上反转后直接显示到物体上即可

```
Properties{
    _MainTex("Main Tex", 2D) = "white"{} // 镜子摄像机渲染得到的渲染纹理
```

```
    v2f vert(a2v v){
        v2f o;
        o.pos = mul(UNITY_MATRIX_MVP, v.vertex);
        o.uv = v.texcoord;
        // Mirror needs to flip x
        o.uv.x = 1 - o.uv.x;
        return o;
    }

    fixed4 frag(v2f i) : SV_Target{
        return tex2D(_MainTex, i.uv);
    }
}
```

- 把创建的渲染纹理拖拽到材质的 Main Tex 属性中
- 注意：渲染纹理的分辨率越高，会影响带宽和性能

玻璃效果

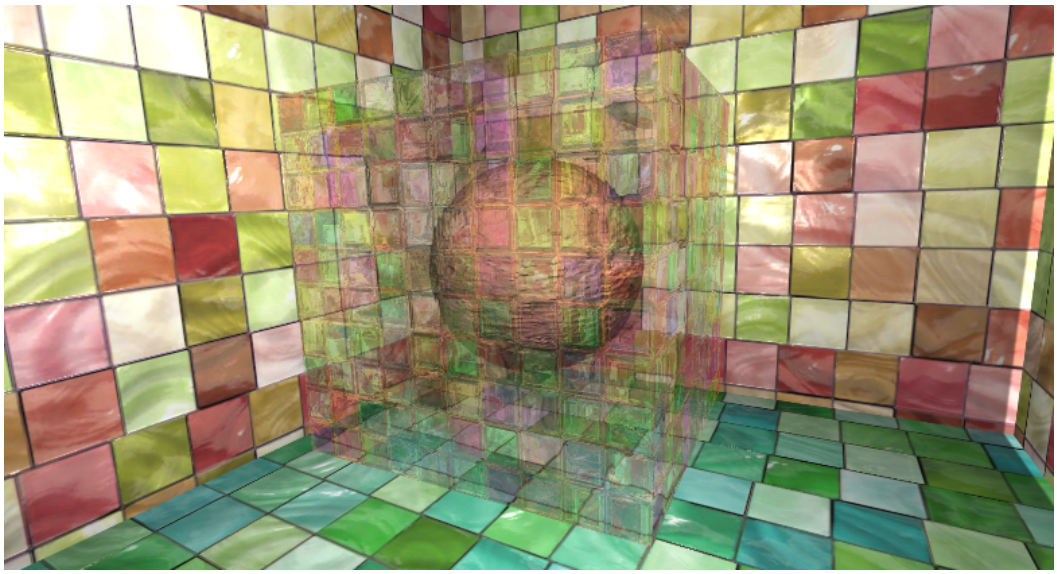
GrabPass

在 Unity 中，我们可以使用一种特殊的 Pass —— GrabPass

- 当我们在 Shader 中定义了 GrabPass 后，Unity 会把当前屏幕的图像绘制在一张纹理中，以便我们在后续的 Pass 中访问它
- 使用 GrabPass 可以让我们对该物体后面的图像进行更复杂的处理，例如使用法线来模拟折射效果

注意：使用 GrabPass 要额外小心物体的**渲染队列设置**，它通常用于渲染透明物体，尽管代码不包含混合之类，我们还要把物体的渲染队列设置成透明队列 `"Queue"="Transparent"`

实践 - 玻璃效果 Shader



```
Shader "Unity Shaders Book/Chapter 10/Glass Refraction" {
    Properties {
        _MainTex ("Main Tex", 2D) = "white" {} // 玻璃材质纹理
        _BumpMap ("Normal Map", 2D) = "bump" {} // 玻璃法线纹理
        _Cubemap ("Environment Cubemap", Cube) = "_Skybox" {}
// 模拟反射环境纹理
        _Distortion ("Distortion", Range(0, 100)) = 10 // 折射
时图像扭曲程度
        _RefractAmount ("Refract Amount", Range(0.0, 1.0)) =
1.0 // 折射程度
    }
    SubShader {
        // We must be transparent, so other objects are drawn
before this one.
         // Transparent 保证渲染"透过玻璃看到的图像"
         // RenderType 为了在使用着色器替换Shader Replacement时,
该物体可以在需要时被正确渲染，这发生在我们需要摄像机的深度和法线纹理时
        Tags { "Queue"="Transparent" "RenderType"="Opaque" }

        // This pass grabs the screen behind the object into
a texture.
        // We can access the result in the next pass as
_RefractionTex
        GrabPass { "_RefractionTex" } // 抓取屏幕图像的Pass，这
个Pass定义的字符串决定了我们抓取得到的屏幕图像会被存入哪个纹理中

        Pass {
            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            sampler2D _MainTex;
            float4 _MainTex_ST;
            sampler2D _BumpMap;
            float4 _BumpMap_ST;
            samplerCUBE _Cubemap;
            float _Distortion;
            fixed _RefractAmount;
            sampler2D _RefractionTex;
```

```
        float4 _RefractionTex_TexelSize; // 纹素大小，例如一个大小为256×512的纹理，它的纹素大小为(1/256, 1/512)

        struct a2v {
            float4 vertex : POSITION;
            float3 normal : NORMAL;
            float4 tangent : TANGENT;
            float2 texcoord: TEXCOORD0;
        };

        struct v2f {
            float4 pos : SV_POSITION;
            float4 scrPos : TEXCOORD0;
            float4 uv : TEXCOORD1;
            float4 TtoW0 : TEXCOORD2;
            float4 TtoW1 : TEXCOORD3;
            float4 TtoW2 : TEXCOORD4;
        };

        v2f vert (a2v v) {
            v2f o;
            o.pos = UnityObjectToClipPos(v.vertex);

            o.scrPos = ComputeGrabScreenPos(o.pos); // 得到被抓去屏幕图像的采样坐标

            // 计算_MainTex和_BumpMap的采样坐标
            o.uv.xy = TRANSFORM_TEX(v.texcoord, _MainTex);
            o.uv.zw = TRANSFORM_TEX(v.texcoord, _BumpMap);

            // 计算顶点对应从切线空间到世界空间的变换矩阵
            // w分量用于存储世界空间下的顶点坐标
            float3 worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;
            fixed3 worldNormal = UnityObjectToWorldNormal(v.normal);
            fixed3 worldTangent = UnityObjectToWorldDir(v.tangent.xyz);
            fixed3 worldBinormal = cross(worldNormal, worldTangent) * v.tangent.w;

            o.TtoW0 = float4(worldTangent.x, worldBinormal.x, worldNormal.x, worldPos.x);
```

```
            o.TtoW1 = float4(worldTangent.y,
worldBinormal.y, worldNormal.y, worldPos.y);
            o.TtoW2 = float4(worldTangent.z,
worldBinormal.z, worldNormal.z, worldPos.z);

            return o;
        }

        fixed4 frag (v2f i) : SV_Target {
            float3 worldPos = float3(i.TtoW0.w, i.TtoW1.w,
i.TtoW2.w); // 得到片元的世界坐标
            fixed3 worldViewDir =
normalize(UnityWorldSpaceViewDir(worldPos)); // 该片元的视角
方向

            // Get the normal in tangent space
            fixed3 bump = UnpackNormal(tex2D(_BumpMap,
i.uv.zw));

            // Compute the offset in tangent space
            // 使用_Distortion属性以及
_RefractionTex_TexelSize对屏幕图像的坐标进行偏移（模拟折射效果）
            float2 offset = bump.xy * _Distortion *
_RefractionTex_TexelSize.xy;
            i.scrPos.xy = offset * i.scrPos.z +
i.scrPos.xy; // 透视除法得到屏幕坐标
            fixed3 refrCol = tex2D(_RefractionTex,
i.scrPos.xy/i.scrPos.w).rgb; // 抓取屏幕图像进行采样。得到模拟折
射颜色

            // Convert the normal to world space

            bump = normalize(half3(dot(i.TtoW0.xyz, bump),
dot(i.TtoW1.xyz, bump), dot(i.TtoW2.xyz, bump))); // 法线方
向从切线空间变换到世界空间
            fixed3 reflDir = reflect(-worldViewDir, bump);
// 得到视角方向相对于法线方向的反射颜色
            fixed4 texColor = tex2D(_MainTex, i.uv.xy);
            fixed3 reflCol = texCUBE(_Cubemap, reflDir).rgb
* texColor.rgb; // 反射方向与Cubemap进行采样，并且与主纹理颜色相乘
后得到反射颜色
```

```
            fixed3 finalColor = reflCol * (1 -
_RefractAmount) + refrCol * _RefractAmount; // 反射颜色和折射
颜色进行混合，得到输出颜色

            return fixed4(finalColor, 1);
        }

        ENDCG
    }
}

FallBack "Diffuse"
}
```

- 我们在这里使用了 `GrabPass { "_RefractionTex" }`，制定了一个字符串说明被抓取的屏幕图像将会存储在哪个名称的纹理中，GrabPass 支持两种形式
  - 直接使用 GrabPass，在后续的 Pass 中使用 _GrabTexture 来访问呢屏幕图像
    - 性能消耗较大，每一个使用它的物体 Unity 都会为它进行屏幕抓取操作
    - 这种方法每个物体可以得到不同的屏幕图像，取决于它们的渲染队列以及渲染它们时当前屏幕缓冲中的颜色
  - 使用 `GrabPass {"TextureName"}`：抓取屏幕，但是 Unity 只会在每一帧时为第一个使用名为 TextureName 的纹理执行抓取屏幕操作，这个纹理可以在其它 Pass 中被访问
    - 更加高效，每一帧 Unity 只会执行一次抓取工作
    - 所有物体都是用同一张屏幕图像

渲染纹理 vs GrabPass

	渲染纹理	GrabPass
优点	我们需要创建一个渲染纹理和一个额外的摄像机，再把摄像机的 Render Target 设置为新建的渲染纹理对象，最后把该渲染纹理传递给相应的 Shader	实现简单

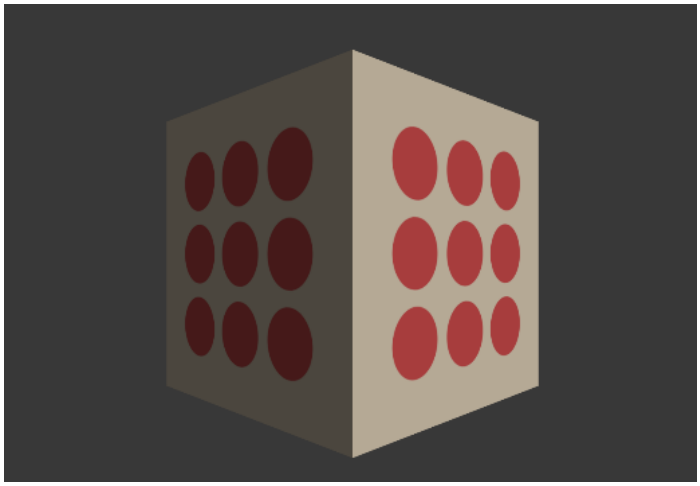
	渲染纹理	GrabPass
缺点	渲染纹理效率更高，尤其在移动设备上，可以自定义渲染纹理的大小	获得的分辨率与屏幕图像一致，在高分辨率的设备上可能会造成严重的影响，GrabPass 需要 CPU 直接读取后背缓冲的数据，破坏了 CPU 和 GPU 的并行性，耗时大，部分移动设备不支持

3. 程序纹理 Procedural Texture

程序纹理 Procedural Texture 指的是由计算机生成的图像

程序纹理可以使用各种参数来控制纹理的外观，比如颜色、图案、动画和视觉效果等

在 Unity 中实现简单的程序纹理



我们要创建一个 `.cs` 脚本来控制纹理

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

[ExecuteInEditMode] // 在编辑器模式下运行
public class ProceduralTextureGeneration : MonoBehaviour {

    public Material material = null; // 使用该脚本生成的程序纹理

    #region Material properties
    [SerializeField, SetProperty("textureWidth")]
    private int m_textureWidth = 512; // 纹理的大小（通常是2的整数幂）
    public int textureWidth {
        get {
            return m_textureWidth;
```

```
        }
        set {
            m_textureWidth = value;
            _UpdateMaterial();
        }
    }

    [SerializeField, SetProperty("backgroundColor")]
    private Color m_backgroundColor = Color.white; // 纹理背景颜色
    public Color backgroundColor {
        get {
            return m_backgroundColor;
        }
        set {
            m_backgroundColor = value;
            _UpdateMaterial();
        }
    }

    [SerializeField, SetProperty("circleColor")]
    private Color m_circleColor = Color.yellow; // 纹理圆点颜色
    public Color circleColor {
        get {
            return m_circleColor;
        }
        set {
            m_circleColor = value;
            _UpdateMaterial();
        }
    }

    [SerializeField, SetProperty("blurFactor")]
    private float m_blurFactor = 2.0f; // 模糊因子
    public float blurFactor {
        get {
            return m_blurFactor;
        }
        set {
            m_blurFactor = value;
            _UpdateMaterial();
        }
```

```
    }
    #endregion

    private Texture2D m_generatedTexture = null;

    // Use this for initialization
    void Start () {
        if (material == null) {
            Renderer renderer =
gameObject.GetComponent<Renderer>();
            if (renderer == null) {
                Debug.LogWarning("Cannot find a renderer.");
                return;
            }

            material = renderer.sharedMaterial;
        }

        _UpdateMaterial();
    }

    private void _UpdateMaterial() {
        if (material != null) {
            m_generatedTexture = _GenerateProceduralTexture();
            material.SetTexture("_MainTex",
m_generatedTexture);
        }
    }

    private Color _MixColor(Color color0, Color color1,
float mixFactor) {
        Color mixColor = Color.white;
        mixColor.r = Mathf.Lerp(color0.r, color1.r,
mixFactor);
        mixColor.g = Mathf.Lerp(color0.g, color1.g,
mixFactor);
        mixColor.b = Mathf.Lerp(color0.b, color1.b,
mixFactor);
        mixColor.a = Mathf.Lerp(color0.a, color1.a,
mixFactor);
        return mixColor;
    }
```

```
    private Texture2D _GenerateProceduralTexture() {
        Texture2D proceduralTexture = new
Texture2D(textureWidth, textureWidth);

        // The interval between circles
        float circleInterval = textureWidth / 4.0f;
        // The radius of circles
        float radius = textureWidth / 10.0f;
        // The blur factor
        float edgeBlur = 1.0f / blurFactor;

        for (int w = 0; w < textureWidth; w++) {
            for (int h = 0; h < textureWidth; h++) {
                // Initalize the pixel with background color
                Color pixel = backgroundColor;

                // Draw nine circles one by one
                for (int i = 0; i < 3; i++) {
                    for (int j = 0; j < 3; j++) {
                        // Compute the center of current circle
                        Vector2 circleCenter = new
Vector2(circleInterval * (i + 1), circleInterval * (j +
1));

                        // Compute the distance between the pixel
and the center
                        float dist = Vector2.Distance(new
Vector2(w, h), circleCenter) - radius;

                        // Blur the edge of the circle
                        Color color = _MixColor(circleColor, new
Color(pixel.r, pixel.g, pixel.b, 0.0f),
Mathf.SmoothStep(0f, 1.0f, dist * edgeBlur));

                        // Mix the current color with the
previous color
                        pixel = _MixColor(pixel, color, color.a);
                    }
                }

                proceduralTexture.SetPixel(w, h, pixel);
            }
        }
```

```
        proceduralTexture.Apply();

        return proceduralTexture;
    }
}
```

Unity 的程序材质 - Substance Designer （已弃用）

Unity 中有一类专门使用程序纹理的材质，叫做 Procedural Materials。这类材质使用的不是普通的纹理，且不是在 Unity 中创建的

它使用一个名为 **Substance Designer** 的软件在 Unity 的外部生成的，这些材质以 `.sbsar` 为后缀，可以直接把这些材质像其它资源一样拖入 Unity 项目中

当这些文件导入 Unity 之后，就会生成一个**程序纹理资源 Procedural Material Asset**，包含一个或多个程序材质