Lab-2


# 图形绘制技术

Introduction and Prerequisite

**Lab2 Unity 自定义 Shadow Map**

2025 春季学期

# Abstract

该文档是南京大学计算机系 2025 春季学期《图形绘制技术》课程实验 2 的实验手册。

Lab2 的内容是在 Unity 中完成自定义阴影的渲染。旨在通过简单的实验让大家了解如何在 URP 渲染管线中**自定义渲染 Pass**，以及如何**修改 URP 底层源码**。

各位同学如果在 Lab2 实验过程中有任何疑问或想法，包括但不限于

- 实验框架的出现的 Bug
- 手册中叙述不完善的部分
- 对实验安排的建议
- 更好的框架设计

主讲老师：过洁 Email：guojie@nju.edu.cn

欢迎联系助教，帮助我们完善课程的实验部分。TAs：

- 杨铭 QQ：925882085 Email：925882085@qq.com
- 孙奇 QQ：745356675 Email：745356675@qq.com

**同时，课程的框架代码以及实验文档也会不断更新改进，请同学们关注助教发布的相关信息。**

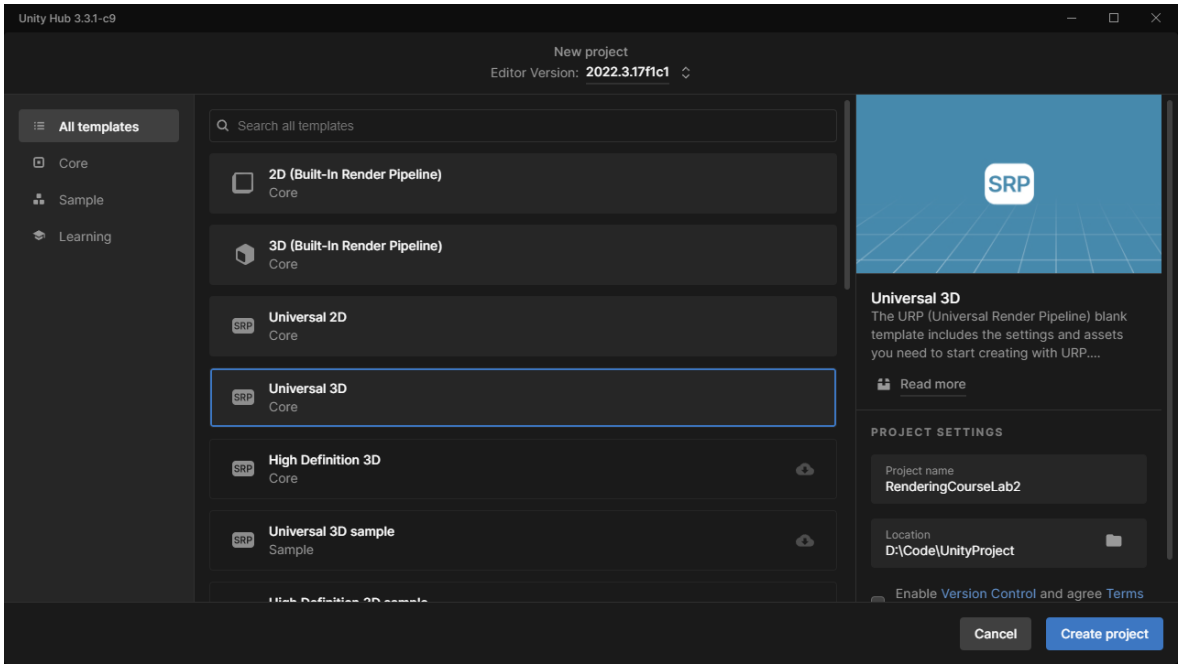
# 目录

# 1. 实验环境准备

## 1.1 Unity 与 URP 版本

请从官网获取 Unity 并使用 Personal License 进行本次实验。本次实验使用的 Unity 版本为 2022.3.17f1c1，URP 版本为 14.0.9。

安装完成后，选择 Universal 3D 模版创建实验项目：




## 1.2 URP 源码环境准备

由于我们需要修改 URP 源码，因此需要改变源码的引用方式：

将 Library/PackageCache/com.unity.render-pipelines.universal@14.0.9 拷贝至项目根目录的 Package 下

在 Package/packages-lock.json 中找到”com.unity.render-pipelines.universal”：

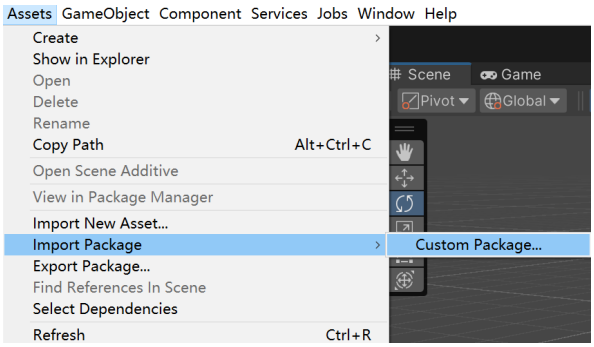
- 修改 “source” 为 “embedded”
- 修改 “version” 为 “file:com.unity.render-pipelines.universal@14.0.9”

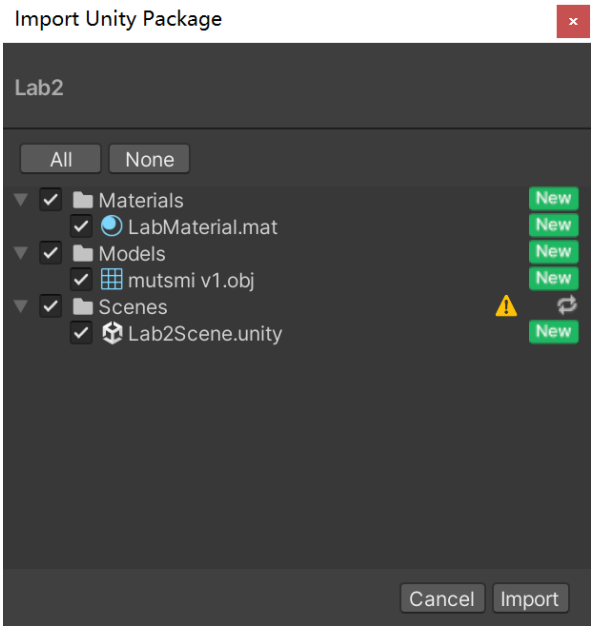
```
"com.unity.render-pipelines.universal": {
  "version": "file:com.unity.render-pipelines.universal@14.0.9",
  "depth": 0,
  "source": "embedded",
  "dependencies": {
    "com.unity.mathematics": "1.2.1",
```

```
        "com.unity.burst": "1.8.9",
        "com.unity.render-pipelines.core": "14.0.9",
        "com.unity.shadergraph": "14.0.9",
        "com.unity.render-pipelines.universal-config": "14.0.9"
      }
    },
```

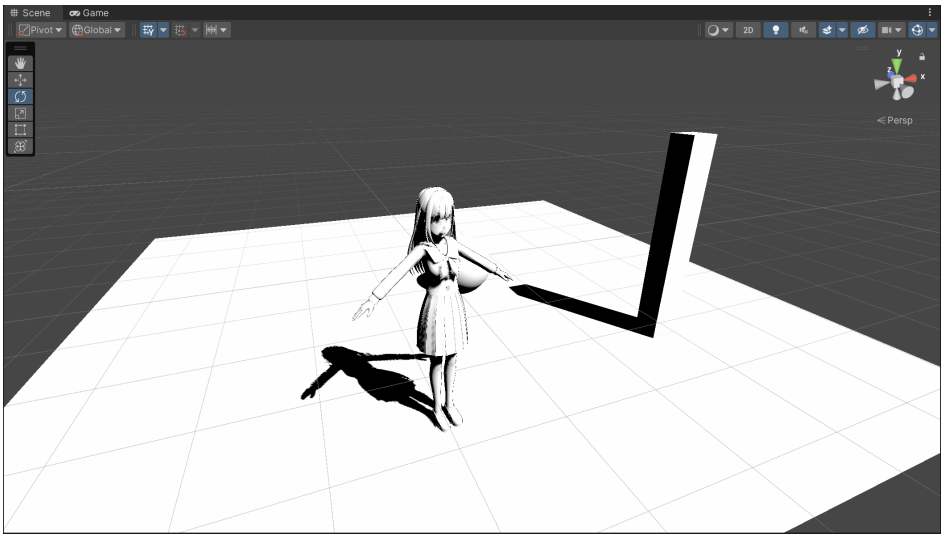
### 1.3 导入场景文件

本实验提供文件 Lab2.unitypackage，导入方式如下




随后进入 Scenes/Lab2Scene，可以看到：

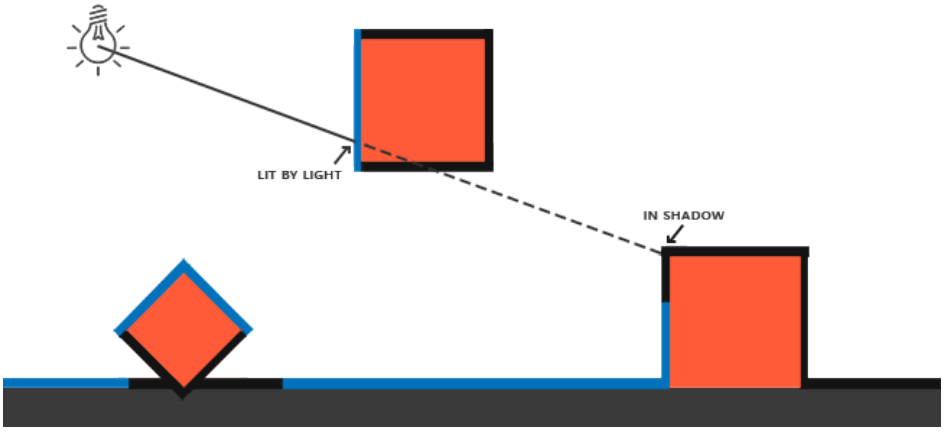



# 2. 生成自定义阴影贴图

## 2.1 Shadow 渲染原理

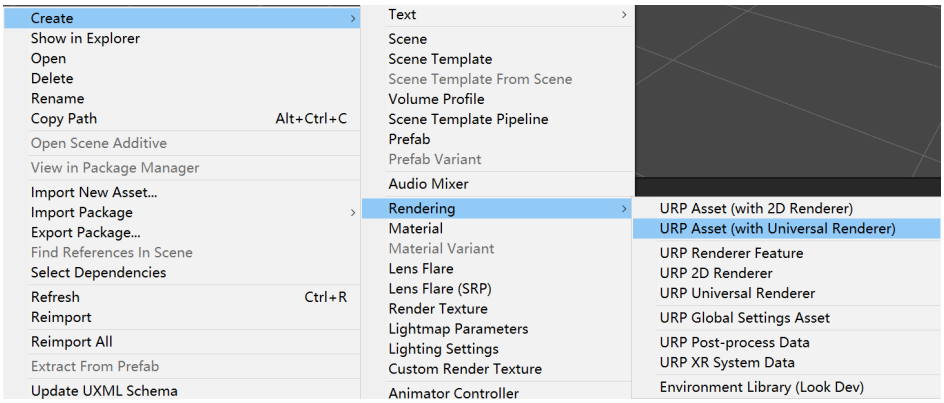
**光源视角渲染：**从光源视角渲染场景，将每个像素到光源的最近深度值（Z 值），记录在 ShadowMap。

**场景渲染时对比深度：**渲染主场景时，将当前像素转换到**光源视角**下的坐标，读取 Shadowmap 中对应位置的深度值。若当前像素深度 > Shadowmap 中的深度（被遮挡），则处于阴影中，否则被照亮。

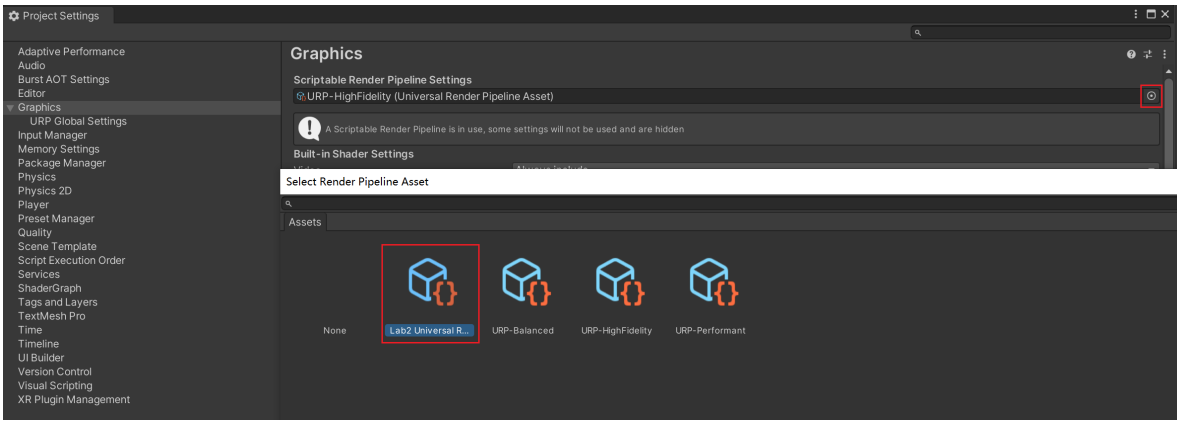


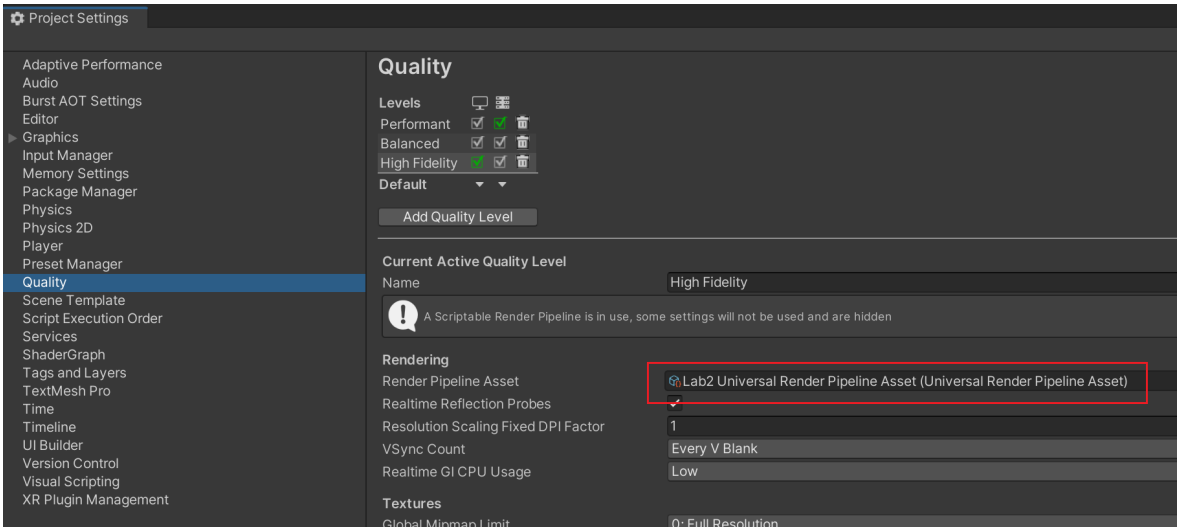

## 2.2 自定义渲染管线创建

在 Settings 目录下，新建渲染管线 Asset，命名为“Lab2 Universal Render Pipeline Asset”



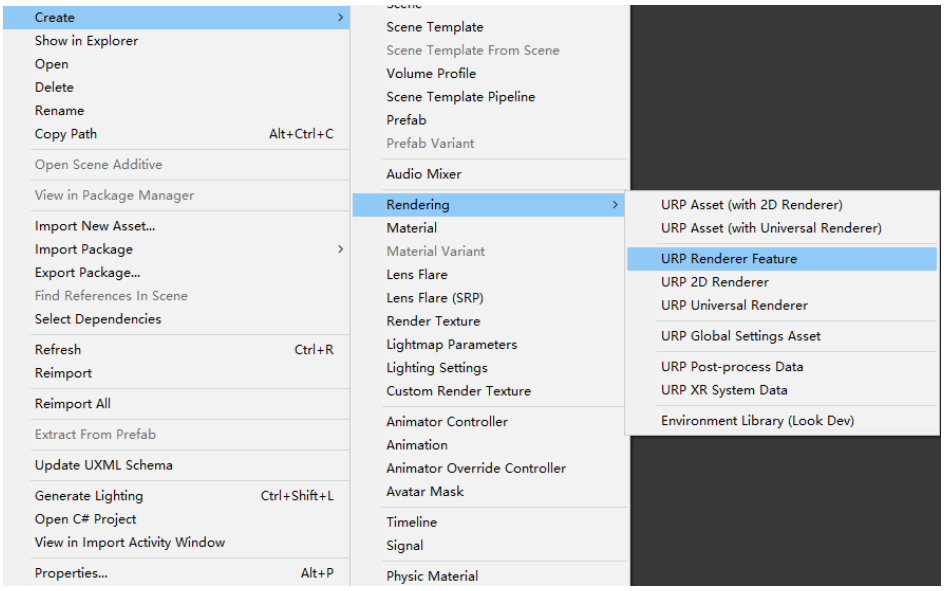

指定自定义管线为当前项目使用的管线。Edit→ Project Settings...→Graphics / Quality



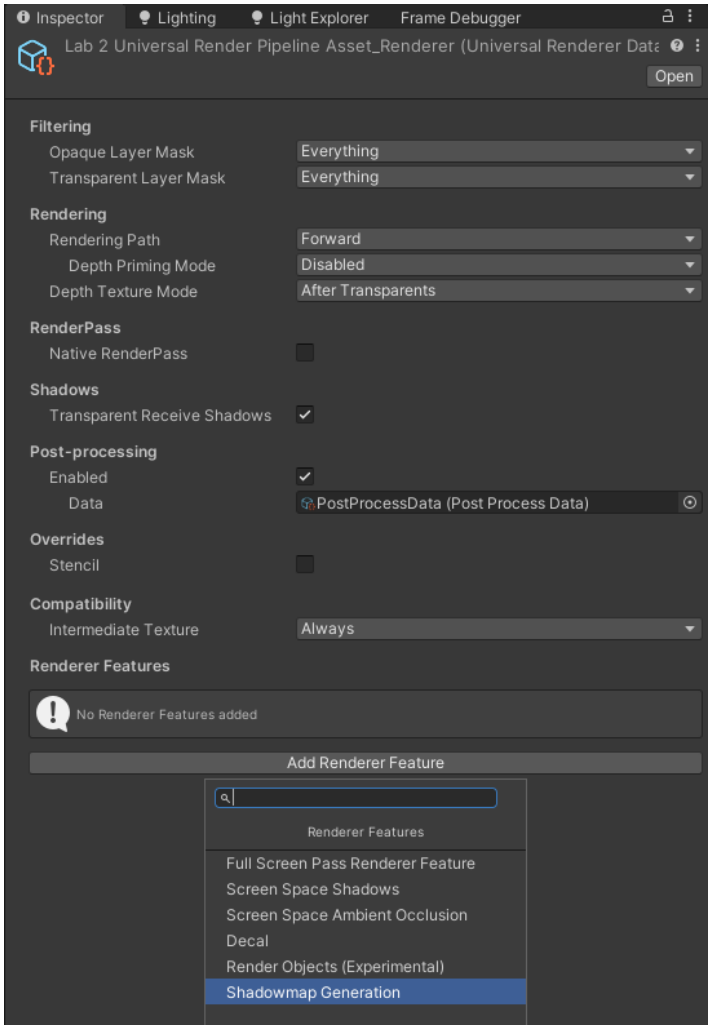

## 2.3 RenderFeature 创建

在 Scripts 目录下新建 RenderFeature，命名为“ShadowmapGeneration”




将 RenderFeature 添加到 Lab2 管线内:

- 在 Project 窗口中选中 Lab2 Universal Render Pipeline Asset_Renderer
- 在 Inspector 窗口中 Add Renderer Feature




## 2.4 创建阴影贴图并设置为 RenderTarget

双击进入 ShadowmapGeneration，可以观察到分别继承 ScriptableRendererFeature，ScriptableRenderPass 的两个类。

renderPassEvent 可以指定我们创建的 RenderFeature 在 URP 渲染管线的哪一步执行，这里我们可以改为 RenderPassEvent.**BeforeRenderingShadows**

在 RenderFeature 中添加成员变量，为创建 Shadowmap 做准备：

```
private RTHandle m_ShadowmapTexHandle; // Shadowmap指针
private const string k_ShadowmapTexName = "_ShadowMap"; // Shadowmap在Shader中的
    引用名
```

添加函数 SetupRenderPasses，在其中创建 Shadowmap RenderTexture：

```
public override void SetupRenderPasses(ScriptableRenderer renderer, in
    RenderingData renderingData){
```

```
    var desc = new RenderTextureDescriptor(1024, 1024, RenderTextureFormat.
    Shadowmap, 24);
    RenderingUtils.ReAllocateIfNeeded(ref m_ShadowmapTexHandle, desc, FilterMode.
    Bilinear, TextureWrapMode.Clamp, name: k_ShadowmapTexName);
    // 将Shadowmap传入RenderPass
    m_ShadowmapRenderPass.SetRTHandles(ref m_ShadowmapTexHandle);
}
```

在 RenderPass 中同样定义 m_ShadowmapTexHandle，并定义函数 SetRTHandles

```
public void SetRTHandles(ref RTHandle tex)
{
    m_ShadowmapTexHandle = tex;
}
```

在 RenderPass 中为 GPU 指定并初始化渲染目标（RenderTarget），Pixel Shader 的计算结果将直接写入渲染目标中：

```
public override void Configure(CommandBuffer cmd, RenderTextureDescriptor
    cameraTextureDescripor)
{
    ConfigureTarget(m_ShadowmapTexHandle);
    ConfigureClear(ClearFlag.Depth, Color.clear);
}
```

## 2.5 向 GPU 发送渲染指令

在 Execute 函数中获取 CommandBuffer，使用 DrawRenderers 进行绘制指令发送：

```
public override void Execute(ScriptableRenderContext context, ref RenderingData
    renderingData)
{
    CommandBuffer cmd = CommandBufferPool.Get("Shadowmap Generation");
    cmd.Clear();

    // ShaderTagId指定使用哪个Shader进行本次绘制
    var drawSettings = new DrawingSettings(new ShaderTagId("CustomShadowCaster"),
        new SortingSettings(renderingData.cameraData.camera) { criteria =
    SortingCriteria.CommonOpaque });
        var filterSettings = new FilteringSettings(RenderQueueRange.opaque);
    // 绘制指令
    context.DrawRenderers(renderingData.cullResults, ref drawSettings, ref
    filterSettings);

    context.ExecuteCommandBuffer(cmd);
    cmd.Clear();
    CommandBufferPool.Release(cmd);
}
```

当然,此处我们还没有传入任何参数,也没有定义 CustomShadowCaster 的 Shader pass，因此渲染还无法正常工作。

### 2.6 矩阵计算

我们需要渲染一张方向光（Directional Light）视角的深度图。你需要**自行完成 View 和 Projection Matrix 的计算**，随后在 DrawRenderers 之前，使用如下命令传入 Shader:

```
cmd.SetGlobalMatrix("_LightVPMatrix", LightVPMatrix);
```

可能用到的数据:

```
// .cs
SystemInfo.graphicsUVStartsAtTop
SystemInfo.usesReversedZBuffer
```

请尽量避免直接使用 Unity 内置的矩阵运算结果，建议手动推导与计算相关矩阵，以展示对底层原理的理解。手动计算将获得加分。

### 2.7 ShadowCasterShader

完成参数计算后，我们需要使用参数进行具体的渲染计算，这部分逻辑将在 Shader 中完成，具体任务是**输出光源视角的深度**。

在 Packages/com.unity.render-pipelines.universal@14.0.9/Shaders/Lit.shader 中

- 复制一份 ShadowCaster Pass 的代码，修改 Name 和 Tag 为 CustomShadowCaster(在 DrawRenderers 的参数中,我们曾指定了 Tag 为 CustomShadowCaster 进行绘制)。
- 注释以下代码:

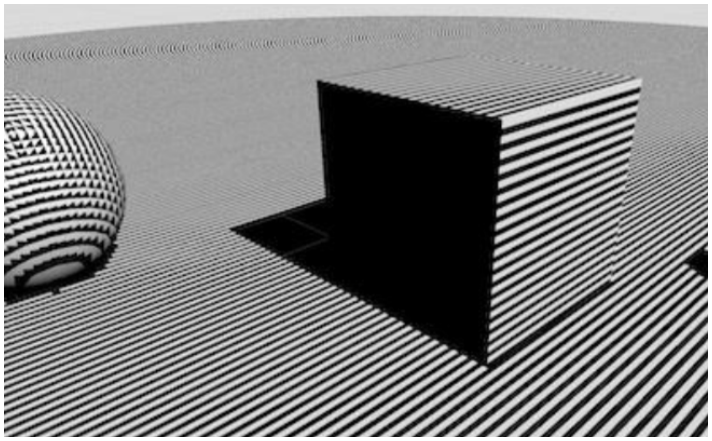
  ```
  #include "Packages/com.unity.render-pipelines.universal/Shaders/
      ShadowCasterPass.hlsl"
  ```

- 定义自己的 Shader，你需要**自行完成 VertexShader 中的逻辑**

```
HLSLPROGRAM
float4x4 _LightVPMatrix;

struct appdata
{
    float4 vertex : POSITION;
};

struct v2f
{
```

```
    float4 positionCS : SV_POSITION;
};


v2f ShadowPassVertex (appdata v)
{
    // 请实现顶点变换到光源空间的逻辑
}

half4 ShadowPassFragment (v2f i) : SV_Target
{
    // 我们只写入深度缓冲，因此Fragment Shader什么也不用做
    return 0;
}
ENDHLSL
```

## 2.8 ShadowBias



由于 ShadowMap 的分辨率有限，ShadowMap 中的一个像素，对应到场景上是一整片区域。

在阴影计算阶段，当我们将该区域内的点投影到 ShadowMap 中进行深度对比时，会发现其中一些点的深度大于 ShadowMap 的记录值，而其中一些小于，因此形成了亮暗条纹。

如下图，AB 为 ShadowMap 相机的 Near Plane，AB 线段对应 Shadowmap 上的一个像素，其覆盖的 CD 范围内的点，取到的 ShadowMap Depth 值均为 BG，而 CF 段着色点深度小于 BG，因而被照亮，但 FD 段则处在阴影中。若 ShadowMap 精度更高，记录下整个 CD 段的深度，则整个 CD 段都不应该存在阴影。

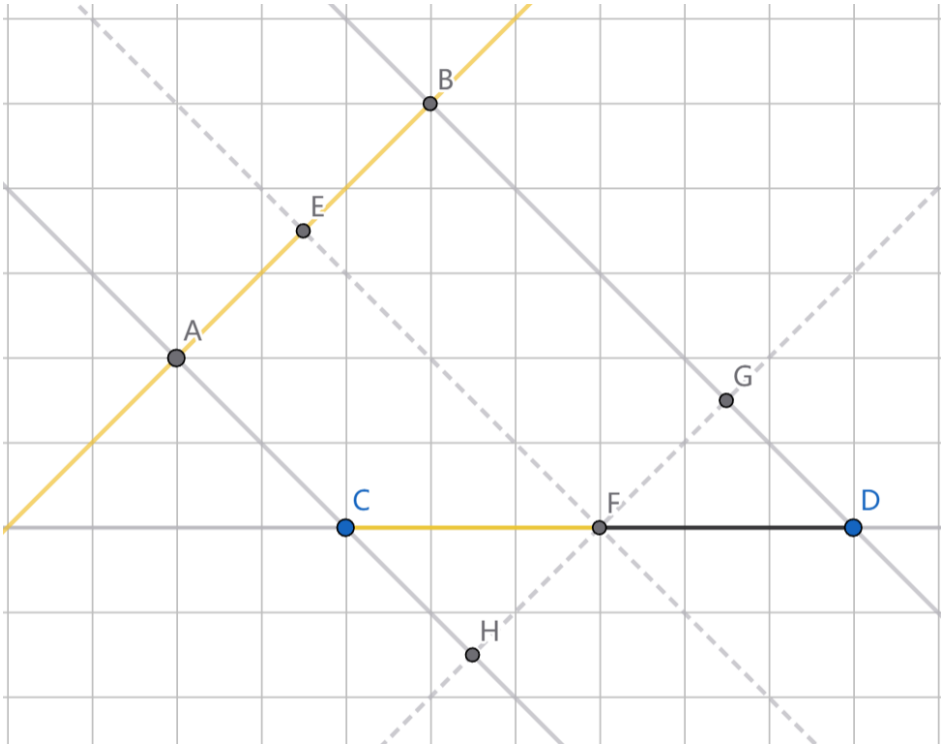
为此，我们可以把 ShadowMap 中的 Depth 向光源方向推 GD 的距离。

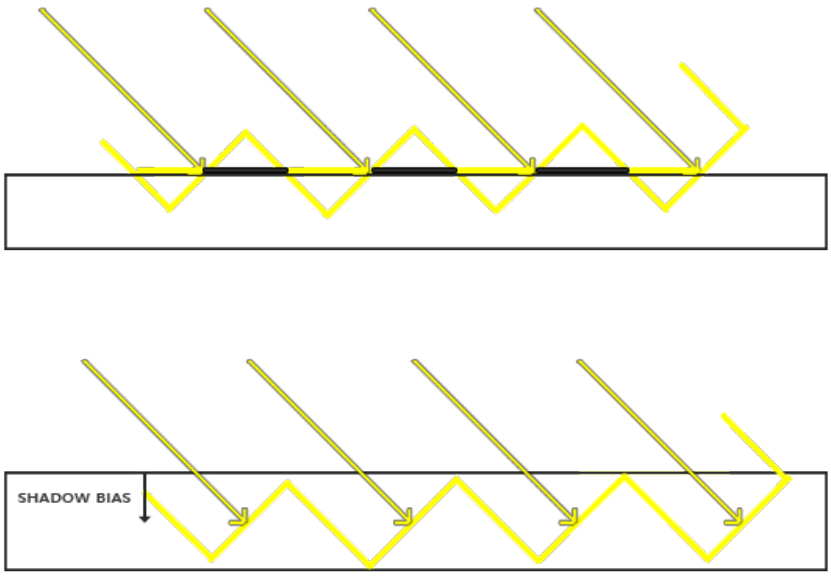
然而实际上在大多数引擎的实现中，并不会这么精确的去计算每个点的 Bias 数

值，而是对所有的点执行一个固定的 Depth Bias。

当角 BDC 变得无限小，GD 也将变得无限大，趋于失效，使用 Normal 方向的偏移作为代替，能缓解这个问题。

在本次实验中，我们仅使用光源方向的偏移。



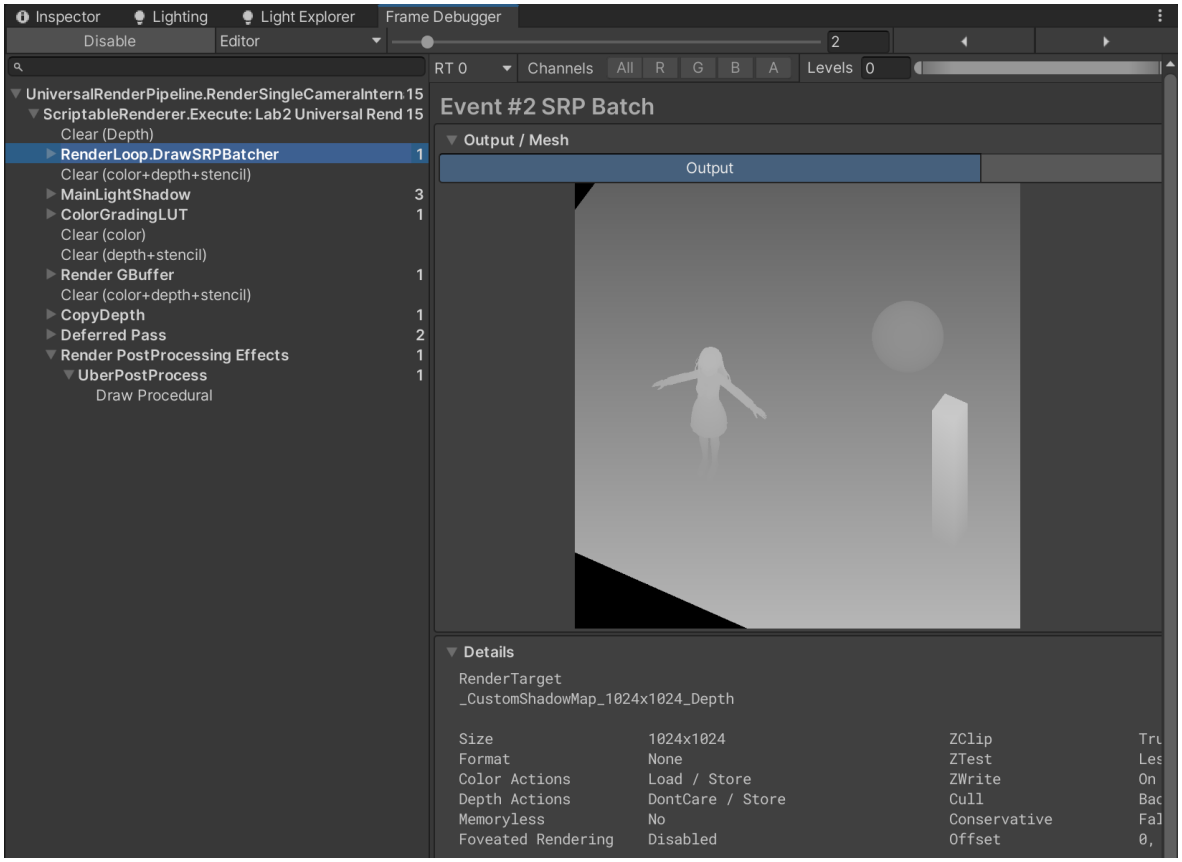




我们需要给 CustomShadowCaster Pass 传入 _CustomShadowBias 参数，朝光源方向偏移。

最终，使用以下光源相机的参数：

- near = 0.01
- far = 15
- orthographicSize = 6
- shadowBias = -0.77

如果实现正确，你将能在 Unity Frame Debugger 中看到：




# 3. 使用 Shadowmap 进行阴影计算

URP 的 Directional Light 阴影入口在 MainLightShadow 函数：

Packages/com.unity.render-pipelines.universal@14.0.9/ShaderLibrary/Shadows.hlsl

我们可以设置 _CUSTOM_SHADOW_ON_ 控制使用自定义的逻辑进行 Shadow 计算。

```
#ifdef _CUSTOM_SHADOW_ON
    half realtimeShadow = CustomShadow();
#else
```

```
    half realtimeShadow = MainLightRealtimeShadow(shadowCoord);
#endif
```

同时在 Assets/Shaders 中新建 CustomShadow.hlsl

```
real CustomShadow(float3 positionWS)
{
    // 暂时设置为0
    return (real)0;
}
```

在 Shadows.hlsl 中添加对 CustomShadow.hlsl 的引用

```
#include "Assets/Shaders/CustomShadow.hlsl"
```

我们可以在 RenderFeature 中启用 _CUSTOM_SHADOW_ON_ 关键字：

```
class ShadowmapRenderPass : ScriptableRenderPass
{
    private const string shadowKeyWord = "_CUSTOM_SHADOW_ON";
    public override void Execute(ScriptableRenderContext context, ref
    RenderingData renderingData)
    {
        cmd.EnableShaderKeyword(shadowKeyWord);
        ……
        context.ExecuteCommandBuffer(cmd);
    }
}
```

同时分别为 Defer 和 Forward 管线添加对关键字的定义

Defer：

Packages/com.unity.render-pipelines.universal@14.0.9/Shaders/Utils/StencilDeferred.shader

```
Pass
{
    Name "Deferred Directional Light (Lit)"
    HLSLPROGRAM
    #pragma multi_compile _ _CUSTOM_SHADOW_ON
    ENDHLSL
}
```

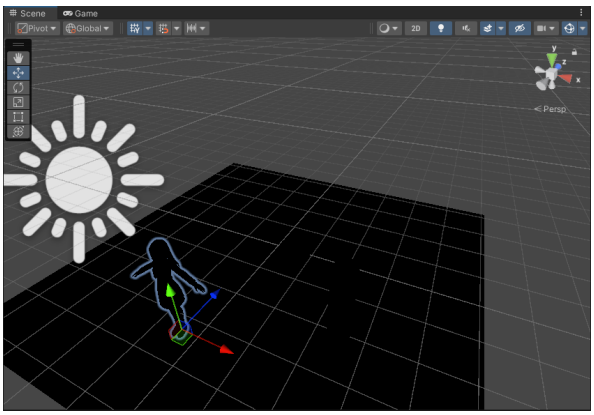
Forward：

Packages/com.unity.render-pipelines.universal@14.0.9/Shaders/Lit.shader

```
Pass
{
    Name "ForwardLit"
```

```
    Tags
    {
        "LightMode" = "UniversalForward"
    }
    HLSLPROGRAM
        #pragma multi_compile _ _CUSTOM_SHADOW_ON
    ENDHLSL
}
```

至此，你的画面会变黑：



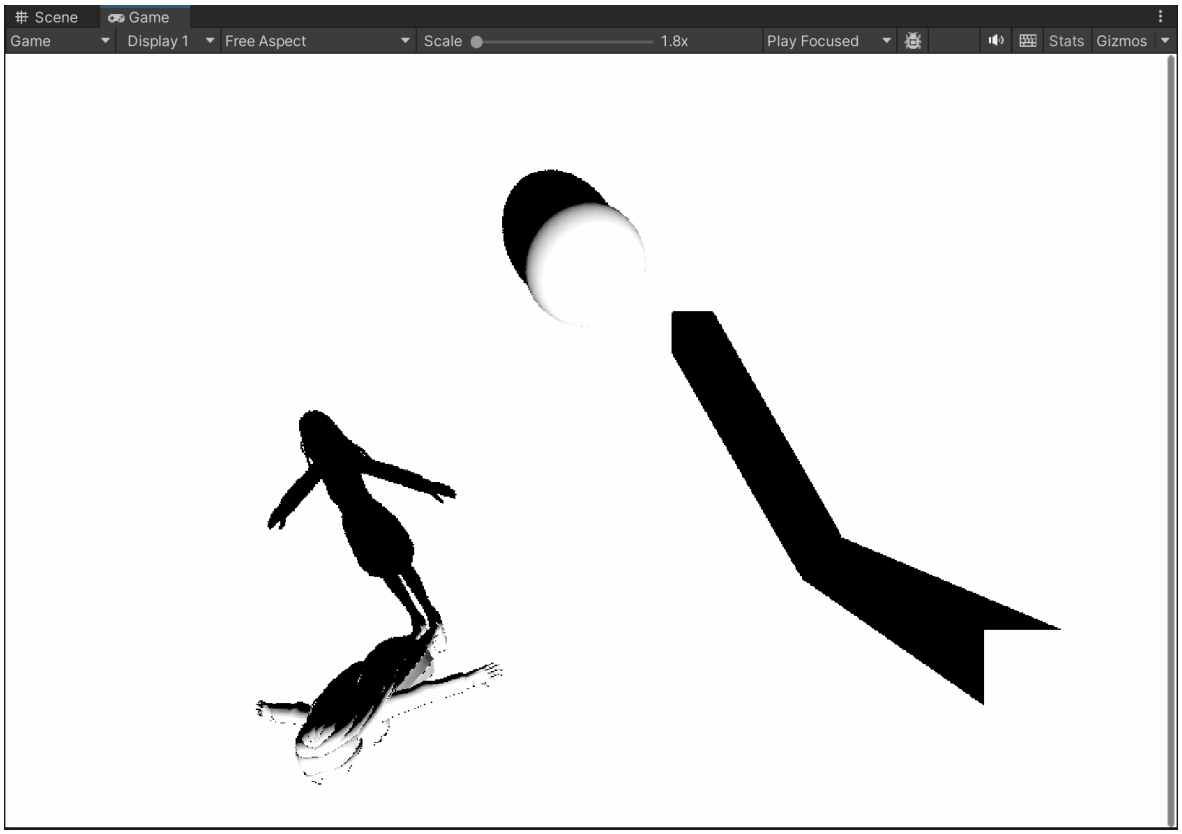
我们开始采样阴影贴图绘制真正的阴影，别忘了在 CPU 端传入 ShadowMap：

```
cmd.SetGlobalTexture(m_ShadowmapTexHandle.name, m_ShadowmapTexHandle);
```

你需要自行完成 CustomShadow 函数：

```
TEXTURE2D(_ShadowMap);
float4x4 _LightVPMatrix;

real CustomShadow(float3 positionWS)
{
    // 可能用到的函数
    SAMPLE_TEXTURE2D(_ShadowMap, sampler_LinearClamp, uv);
    // 可能用到的宏
    UNITY_UV_STARTS_AT_TOP
    UNITY_REVERSED_Z
}
```

如果一切顺利，你将看到：

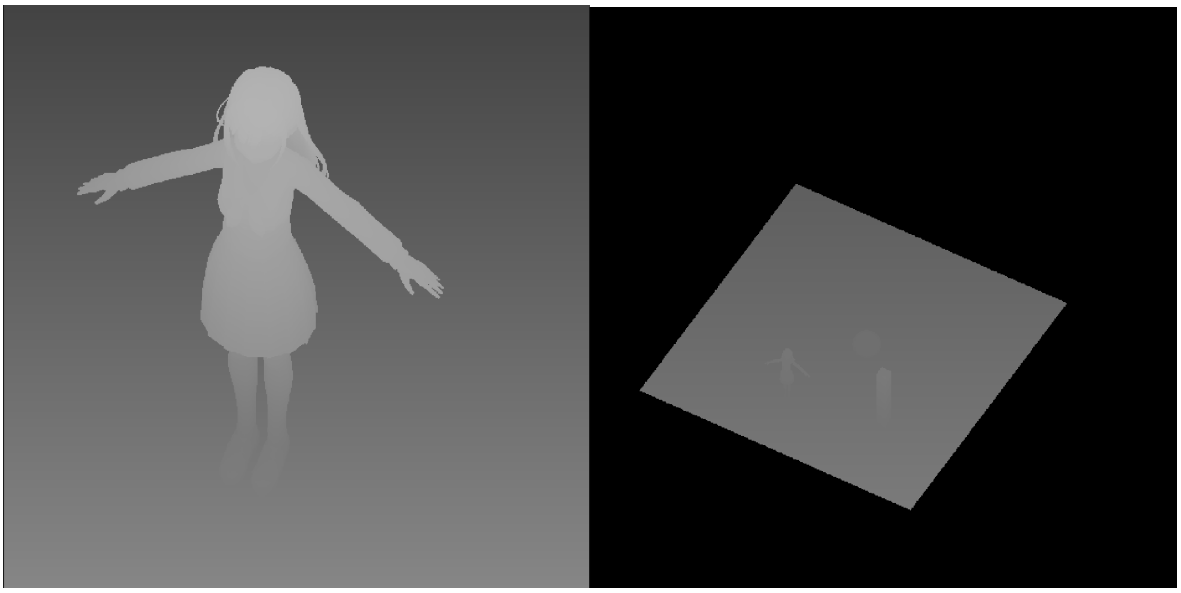



# 4. 选做课题

## 4.1 ShadowMap 自适应包围盒

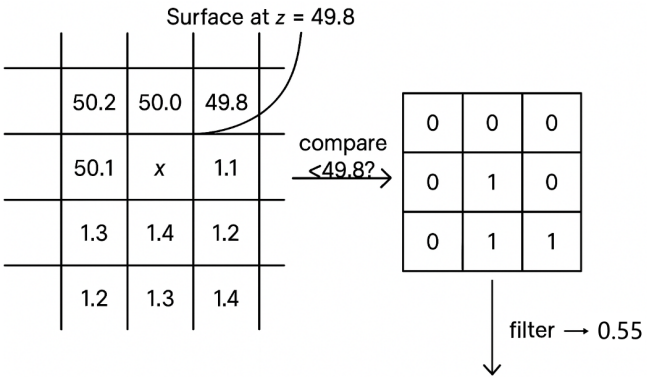
ShadowMap 的参数将直接影响生成 Shadow 的质量。右图为整个场景的 ShadowMap，精度较低。

对于距离视野较近，精度要求较高的角色阴影，低精度的 ShadowMap 难以满足用户的视觉需求，因此可以通过角色的包围盒，确定 Shadow 相机的参数，为角色单独生成更高精度的 ShadowMap。



## 4.2 软阴影 PCSS

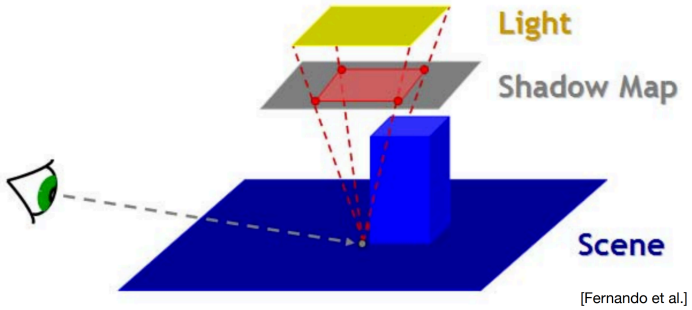
软化的阴影边缘可以通过对 shadowMap 进行 Filter 得到。



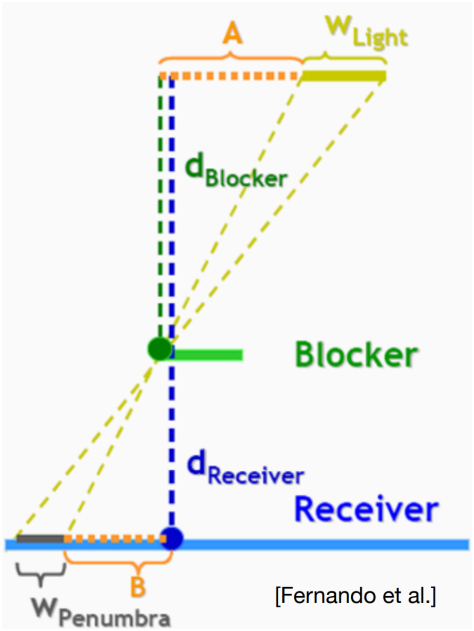

根据下图我们可以观察到，Shading Point 与**遮挡物距离越近，其阴影越硬**。这意味着我们需要根据 Shading Point 与遮挡物的距离，动态调整软化半径。

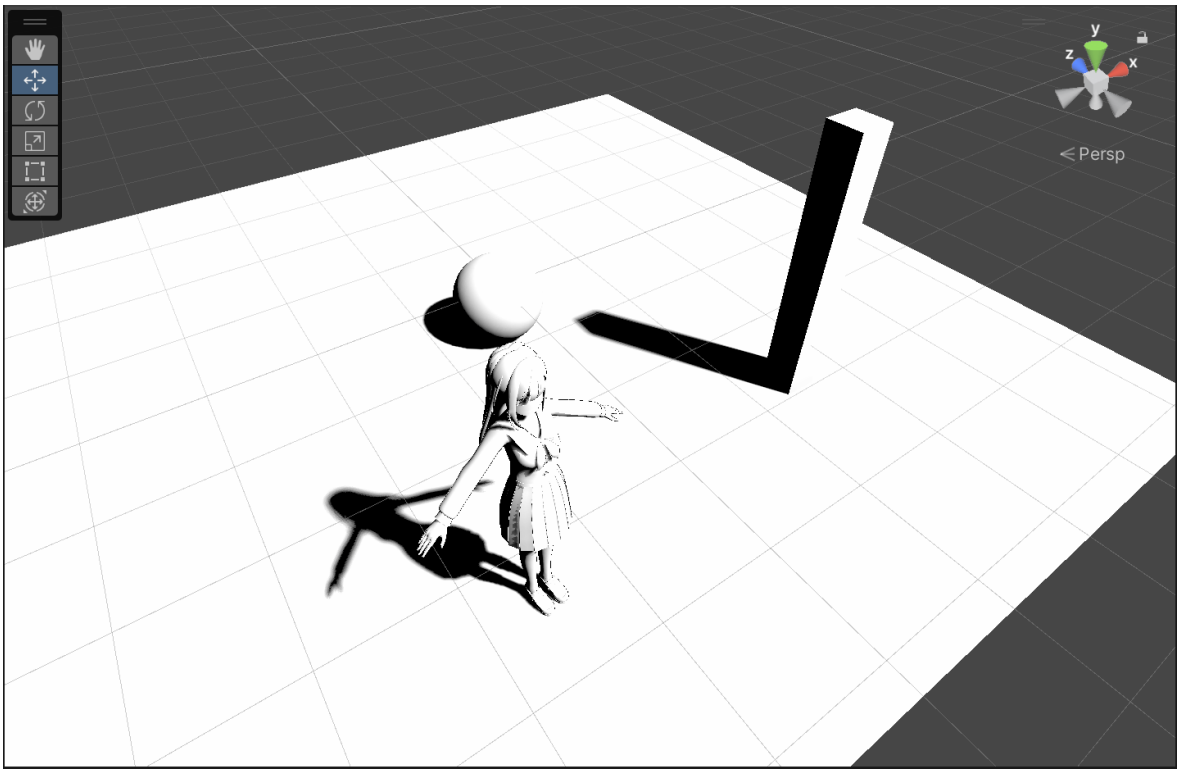
遮挡物的平均距离需要对一定半径内的 ShadowMap 进行 Filter，此半径与 Light

面积、Light 到 ShadowMap 的距离（Near）、Shading Point 到 ShadowMap 的距离有关。




计算出遮挡物的平均距离后，即可根据相似三角形计算软阴影的 Filter 半径。






### 4.3 PCSS 开销优化

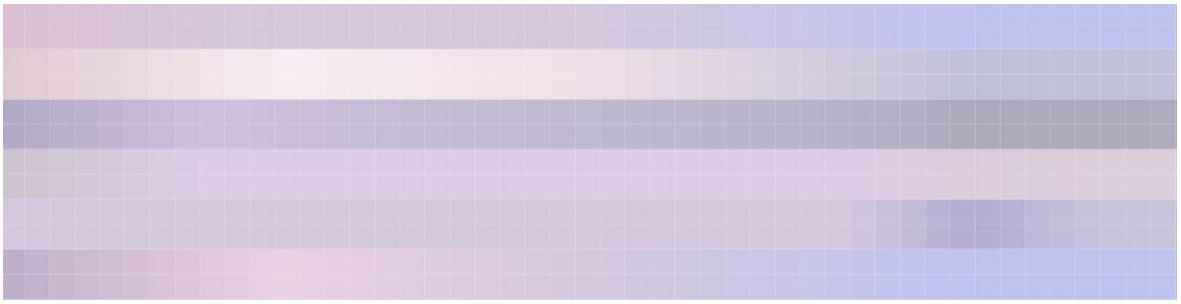
PCSS 计算开销较大，每个像素需要大量纹理采样。但很多不在半影区域的像素并不需要 PCSS 计算，其阴影值非 0 即 1。

因此，生成一张半影遮罩纹理，在屏幕空间标记出需要进行 PCSS 软阴影计算的像素，只有遮罩值不为 0 的区域才执行 PCSS 计算，从而减少无效采样。

### 4.4 软阴影溢色

软阴影溢色常见于风格化渲染中，在阴影的边缘根据倾斜角度，存在不同程度的颜色溢出。

可以通过在阴影边缘采样 Ramp 图实现。



### 4.5 风格化 SDF 面部阴影

由于模型面片精度较高，如果直接使用常规的半兰伯特漫反射进行阴影渲染，会出现如左图一样凌乱的阴影效果。在风格化渲染中，常用 SDF 进行面部阴影渲染，相关的技术在面片云渲染中也有使用。

不同于 3D SDF 在物体表面内为负值，物体外为正值。2D SDF，记录的是光影边界信息，例如，边界为 0.5，边界外 >0.5，边界内 <0.5。

每个光照角度都对应一张 01 贴图记录边界信息，由于角色面部在光照角度连续变化的情况下，暗部和亮部的边界也是连续变化的，因此我们可以把每个光照角度的 01 贴图累加为一张图，即可通过阈值筛选出任意角度的光照情况。这也是使用 2D 纹理记录动画的巧妙思路。

# 5. 作业提交

## 5.1 分数占比

- Shadowmap 生成: 50%
- 使用 Shadowmap 渲染阴影: 40%
- （选做）Shadowmap 自适应包围盒: 10%
- （选做）PCSS 软阴影: 20%
- （选做）PCSS 开销优化: 10%
- （选做）软阴影溢色: 5%
- （选做）风格化 SDF 面部阴影: 10%
- 除了以上列出的工作外，我们鼓励大家探索其他阴影方案，根据实现难度和效果给予额外加分:5%-20%

## 5.2 提交方式

在教学立方提交 **pdf** 格式实验报告，包括以下内容:

- 详细的代码实现思路。
- 实验结果图片。
- 遇到的问题和困难，对实验的建议和吐槽。

# 6. 参考文献

本文参考了以下资料：引用 Shadow Mapping 原理：[1]，OpenGL 实现：[2]，NVIDIA 技术文档：[3], [4]，Unity 的官方说明 [5]。

[1] Wikipedia, Shadow Mapping, *Wikipedia*. Accessed: April 2025.

[2] Joey de Vries, LearnOpenGL: Shadow Mapping, *LearnOpenGL*. Accessed: April 2025.

[3] Randima (Randy) Fernando, Shadow Map Antialiasing, *GPU Gems Chapter 11*, NVIDIA, 2004.

[4] NVIDIA, Percentage-Closer Soft Shadows (PCSS), NVIDIA Shader Library, 2007.

[5] Unity Technologies, Shadow Mapping, *Unity Manual*. Accessed: April 2025.