Render Graph in Unity SRP

李知洋

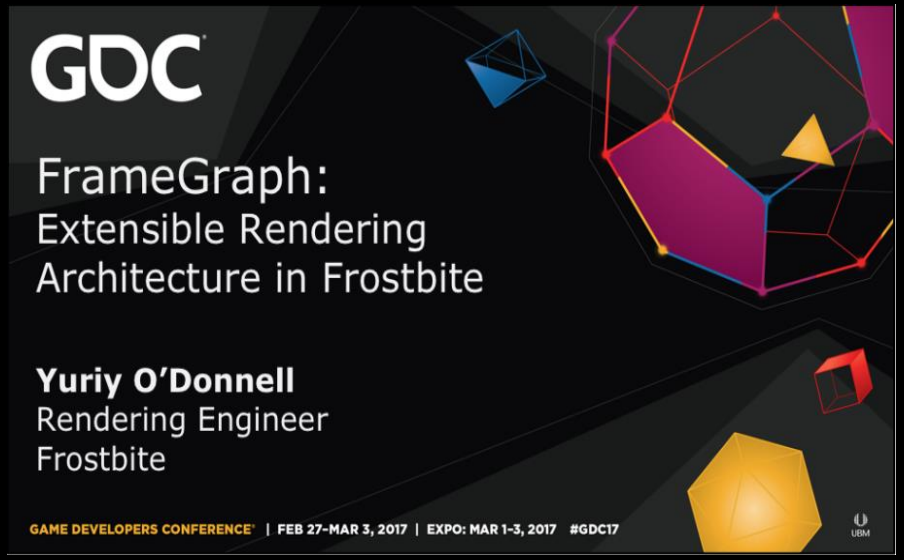
紫龙游戏

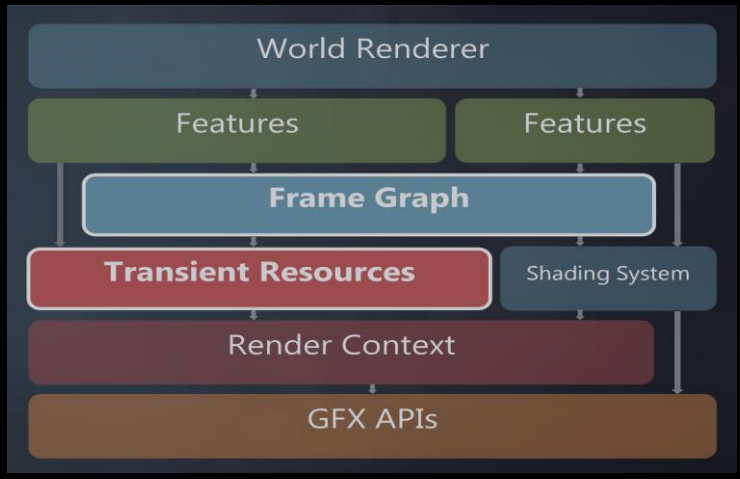
内容概要

- Frame Graph@Frostbite
- Unity SRP 接口
- SRP Render Graph 结构
- SRP Render Graph 具体实现遇到的问题和解决方法

Frame Graph@Frostbite

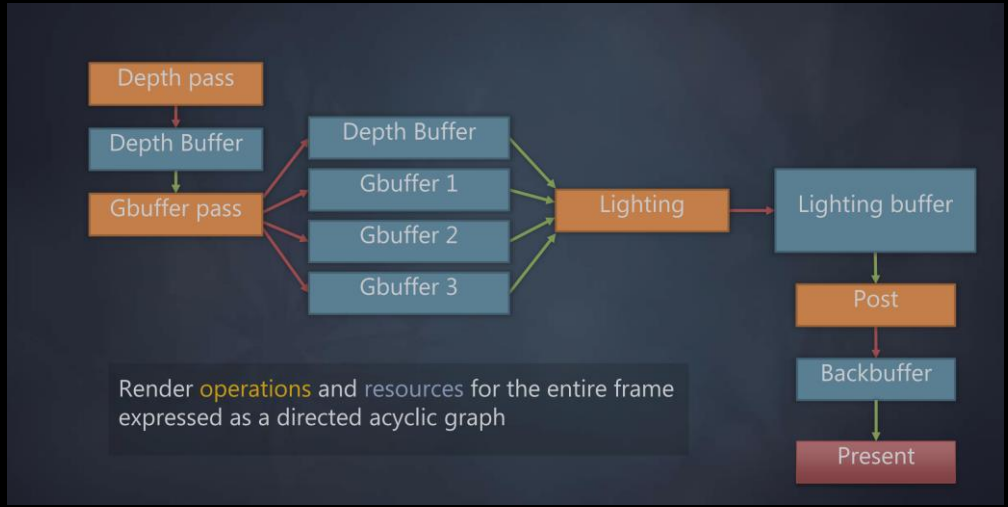
- Frostbite 作为EA DICE内部开发的引擎，服务许多项目：战地，FIFA，星战，极品飞车等，类型涵盖RPG，Racing，Sports，Action，FPS，TPS
- 渲染系统能够支持上述类型的游戏, 通用性要强
- 维护成本不随着不同团队的技术路线不同而提升，可维护性要强;
- 新项目大量新feature的需求，可扩展性要强;





Frame Graph@Frostbite

- 通过 图 组织一帧中的渲染逻辑
- “High level representation of render passes and resources”
- “Full knowledge of the frame”



Frame Graph@Frostbite

- Goals:

1. 简化渲染资源管理
2. 简化管线配置
3. 简化async compute和资源barrier
4. 允许自洽的，高效的渲染管线模块化
5. 可视化 以及 debugging 复杂管线

Untiy SRP

- Unity 的SRP ， 相比较于BuiltIn 自由度较高, 主要提供了两个接口类:

1. UnityEngine.Rendering.RenderPipeline

   调用Render() 接口进行管线绘制

   负责设置管线状态

   管理global管线资源

2. UnityEngine.Rendering.RenderPipelineAsset

   创建RenderPipeline

   管理默认资源

   管理RenderPipeline 的设置数据

UnitySRP - RenderPipeline

- Render() 是unity SRP 每一帧更新的接口
- ScriptableRenderContext 定义了SRP中需要的管线状态和提交绘制指令

  1. 负责Camera Culling

  2. 通过DrawRenderers(),ExecuteCommandBuffer()进行绘制

  3. Submit 会执行所有提交的绘制命令

- Cameras存放着当前帧获取到的所有Camera，不同camera需不同RenderPath

```
#region Overrides of RenderPipeline
/// <summary>
/// override unity SRP 的render 接口, SRP 每一帧更新的接口
/// </summary>      yangkun, 3 months ago • [SRP] 同步TAT, 修复tessellation decal材质
/// <param name="context"></param>
/// <param name="cameras"></param>
protected override void Render(ScriptableRenderContext context, Camera[] cameras)
```

Scriptable RenderContext

- **CullingResults**

  Camera剔除之后的抽象结果，包含绘制对象，可见光源等；

- **DrawSettings**

  怎么画，包括排序，绘制所需shaderPass，需要设置的perObjectData，是否开启Instancing，是否开启Dynamic Batching等；

- **FilterSettings**

  画什么，包括renderQueue，绘制的layerMask和renderingLayerMask，后者是一个只为渲染层服务的标识位，在light和Renderer上都可以设置

- **DrawRenderers**

  绘制CullingResults中的绘制对象，通过DrawSettings，FilterSettings，RenderingLayerMask来控制绘制什么，怎么绘制；

Camera 数组

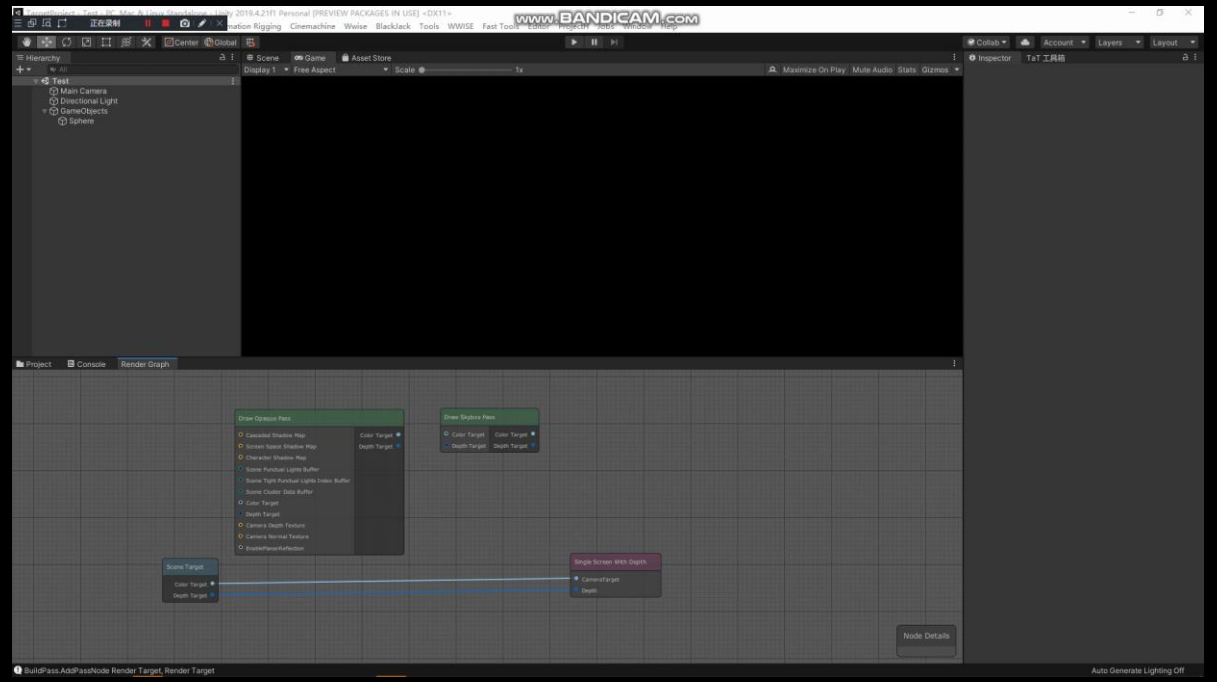
- Unity 提供的当前正在使用的camera
- Camera有不同类型：SceneView，Game，Reflection，Preview, VR
- Unity 中不同窗口对应的不同的type 的camera

```
data.m_preferGraphType = RenderGraphType.GameView;
#if UNITY_EDITOR
//Camera Preview in SceneView.
if (camera.cameraType == CameraType.Game)
{
    if (camera.name == "Preview Camera" && camera.hideFlags == HideFlags.HideAndDontSave)
    {
        data.m_preferGraphType = RenderGraphType.SceneViewPreviewCamera;
    }
}
else if (camera.cameraType == CameraType.SceneView)
{
    data.m_preferGraphType = RenderGraphType.SceneView;
}
else if (camera.cameraType == CameraType.Reflection)
{
    data.m_preferGraphType = m_lightmapBakeHandler.IsBakingSkyBox ? RenderGraphType.BakeSkyBox : RenderGraphType.BakeReflectionProbe;
}
else if (camera.cameraType == CameraType.Preview)
{
    data.m_preferGraphType = RenderGraphType.Preview;
}
#endif
```

渲染管线需求

- 需要同时支持多个项目，跨平台(mobile, pc, console)
- 高通用性(不同项目可以通用)
- 高可维护性(不同项目各自开发版本的维护成本需要降低)
- 高可扩展性(新feature的开发, 已有feature的拆分等)
- URP HDRP 不够自由，但是有参考价值

SRP Render Graph

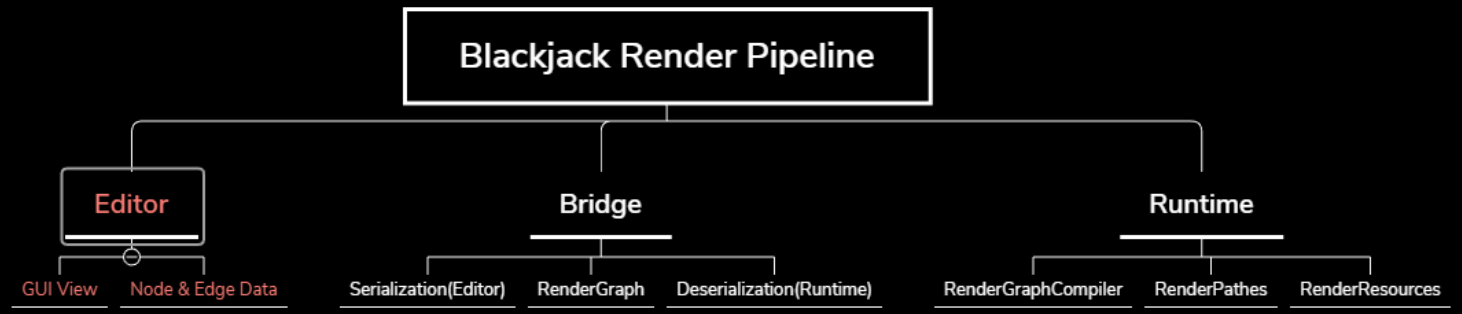


SRP Render Graph 结构

- 编辑器阶段

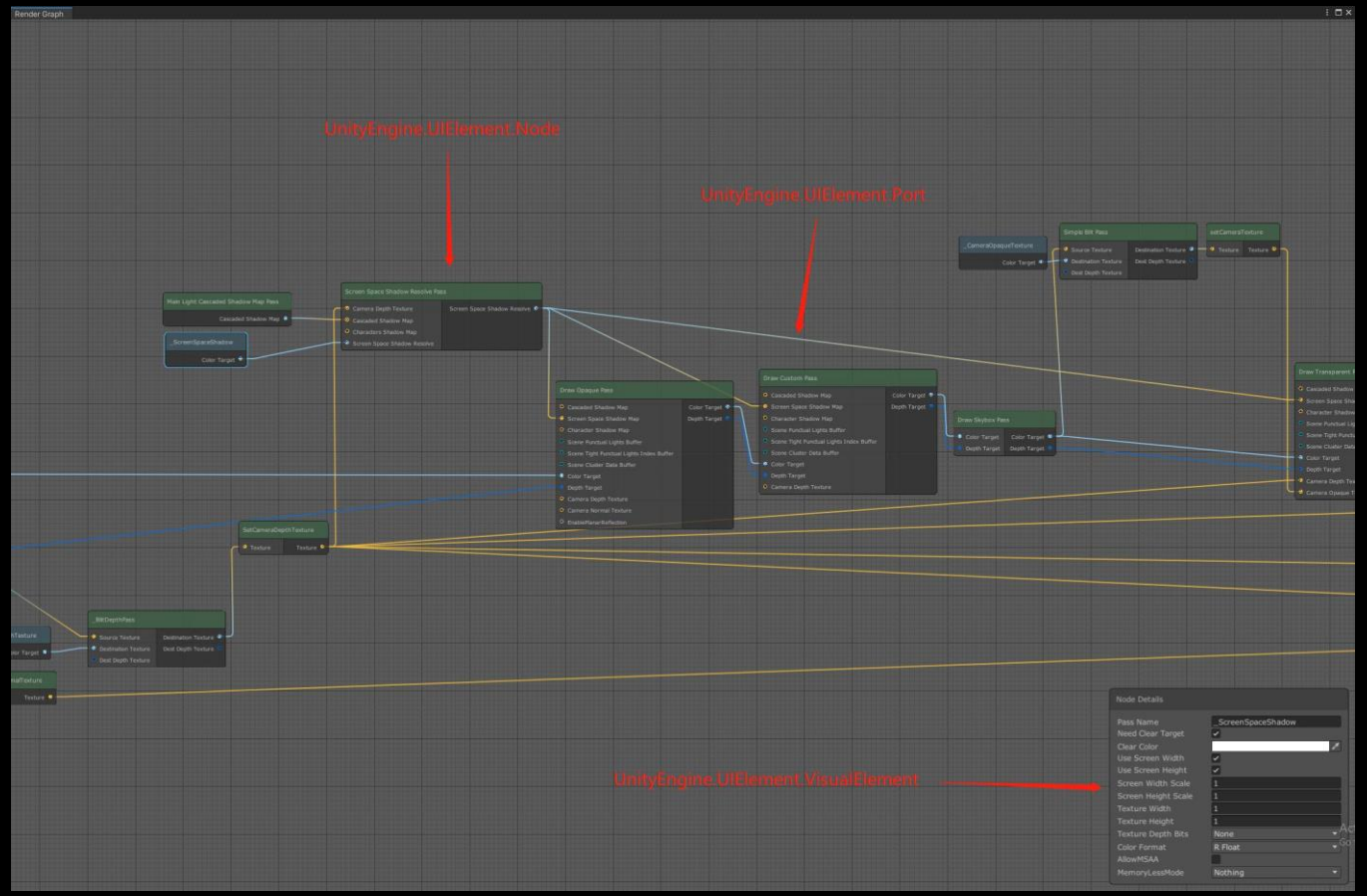
  1. GUI View： render graph 编辑器界面的绘制

  2. Backend Data: 存储着render graph中Node和Edge的数据



Editor

- 编辑器阶段使用Unity的UIElement 接口进行Graph绘制



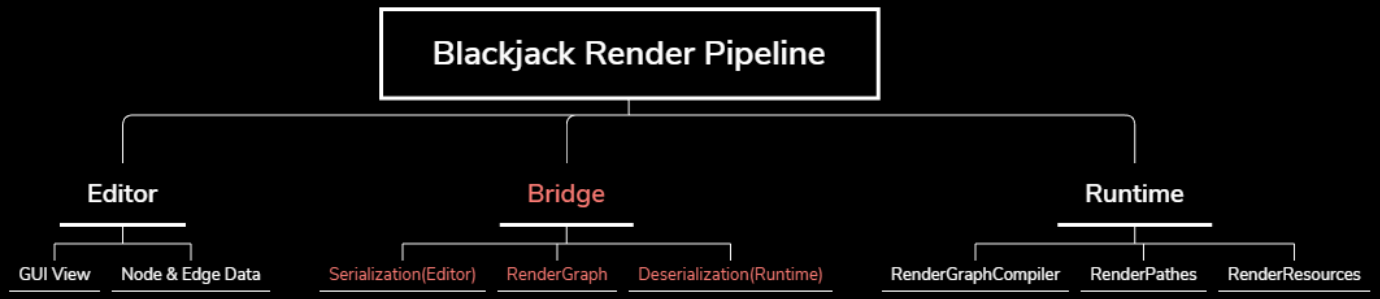
SRP Render Graph 的结构

- 桥接阶段

  1. Serialization：包括以上数据的序列化辅助数据和序列化方法

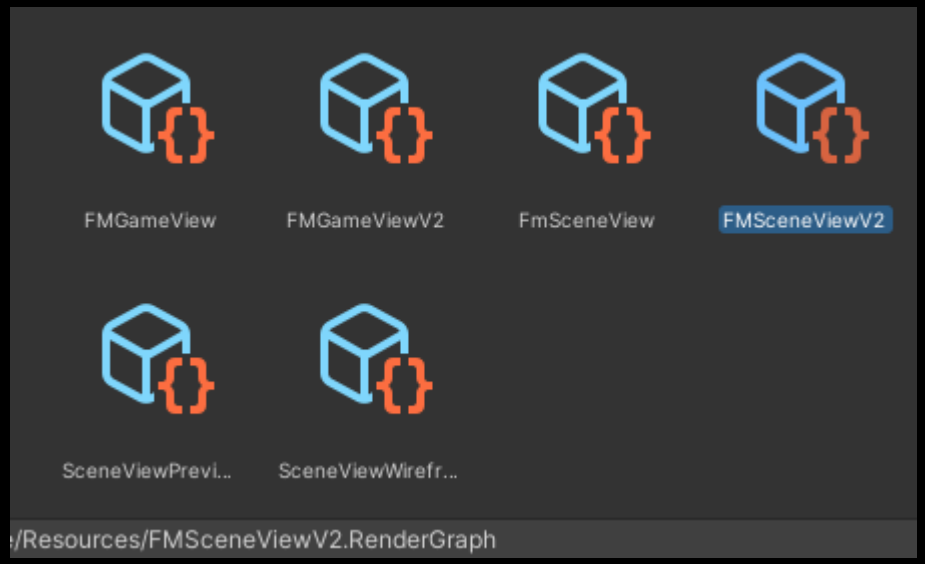
  2. RenderGraph: 序列化对象

  3. Deserialization: 将Render Graph反序列化，抽取数据



Bridge

- 将图的数据保存到RenderGraph(Scriptable Object)上，Scriptable object 只负责图的信息的存 和 取



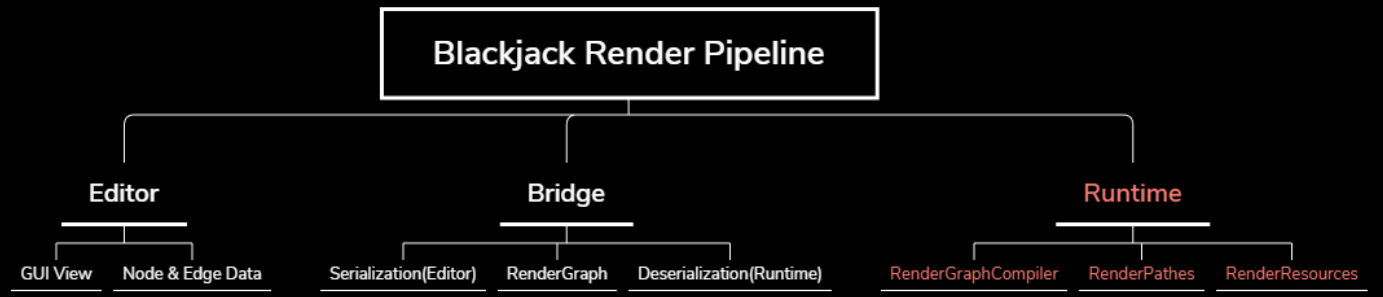
SRP Render Graph 的结构

- 运行时阶段

  1. RenderGraphCompiler: 运行时对于RenderGraph进行编译

  2. RenderPathes: 执行管线渲染逻辑

  3. RenderResources: 提供管线渲染资源



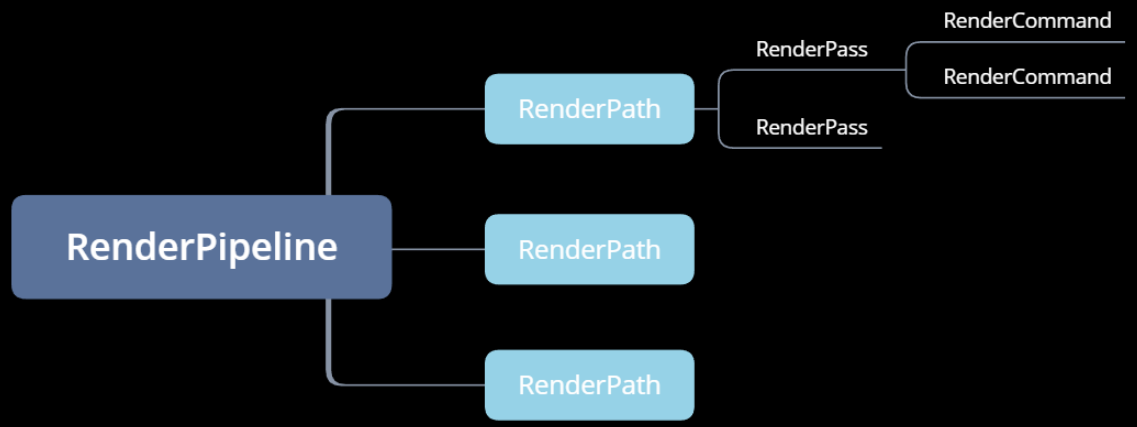
Runtime

Init

- Compile graph 反序列化后的数据
- 构建RenderPass

Update

- 执行RenderPass
- 第一次被使用时生成管线资源



SRP Render Graph

- 编译需要尽可能地较少RT 切换

  在编译期将RT 减少切换作为编译的影响因子

- 尽可能减少不必要的RenderCommand执行

  每个render pass执行的时候，其中的RenderCommand 实际是由一个RenderCommandExecuter 来执行的，Executer将会维护一个RenderCommand执行上下文，如果遇到重复的无效RenderCommand执行，则可以跳过这个RenderCommand

SRP Render Graph

- Temporal 的编译支持

  History Buffer 的数据不能够被当前帧的其他pass 使用，所以编译时没法通过实际引用依赖逻辑来处理

  通过在Graph 上增加特殊Node来单独处理

SRP Render Graph

- 每个RenderPass 对于RT 资源有三种行为：创建，读取，写入
- 在编译的时候能够确定每个RT 资源最后一次被使用的时机
- 每个RenderPass 不需要关注自己使用的RT resource 实体，而只需要在执行时通过RT 资源的句柄去获取实际RT 资源即可
- 通过虚拟资源实现RT 资源复用

SRP Render Graph

- SRP Core package 提供了VolumeComponent 的机制来进行逐场景的Global 或Local的数据的设置，我们使用VolumeComponent来管理各个RenderPass的参数运行时设置
- 通过RenderPassParameterProvider来封装各个RenderPass 对应的VolumeComponent，来提供对接客户端应用层的数据修改请求

SRP Render Graph

- 运行时资源重建

  一些RT资源需要能够在编辑期或运行时被重建，例如Bloom Prefilter Down Sample 的大小，Shadowmap 更改 Resolution 或者Cascade 数量的时候

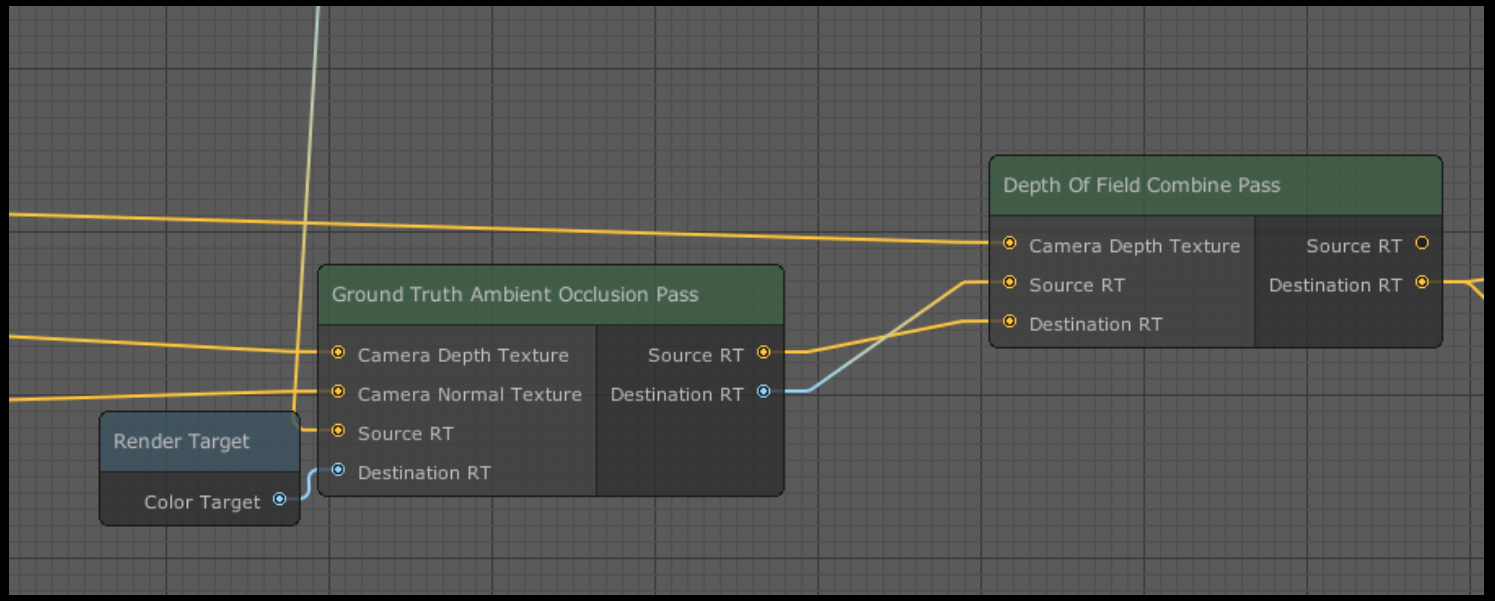
- 只有资源的创建者才能够在运行时重建资源

- RTHandles 能够处理随屏幕变化的RT变化需求

  在Unity Scene View 窗口，每次调整窗口大小的时候都需要更新RT 的大小，之前自己在管线中手动维护，现在通过RTHandles来处理窗口resizing的问题

SRP Render Graph

- 编译完成的管线需要在运行时开关某些pass，并且会导致编译期确定的资源引用 发生错误
  通过虚拟资源指针Switch 来解决



开发者鸣谢

- Itainter
- solid
- Avery
- 郑也歌
- 蒋蒋
- 棉花糖
- 伊重霄
- 404
- Feynman
- 弹头
- 阿里

如果你也感兴趣的话

Thank You

ZLONGAME