Analysis of Unreal Rendering System (09) - Material System

Table of contents

9.1 Overview

Materialis a very important basic system in UE rendering system, which includes material blueprint, rendering status, geometry attributes and other data. Due to various cross-platform and detail encapsulation of UE, the material system appears to be large and complex. This article will analyze its technology, features and mechanisms from the surface to the bottom layer.

Many previous chapters have covered a lot of concepts, types and codes of materials. This article will explain its system in more depth and extensive. It mainly explains the following contents of UE:

- Basic concepts and basic types of materials.
- The implementation level of the Material.
- The implementation and principles of materials.
- How to use materials and use cases.

The material system is built on the Shader system and is the main bridge connecting graphics programs, TA, and artists.

9.2 Material Basics

This chapter will analyze the basic concepts and types of materials, and explain their basic relationships and usage methods.

9.2.1 UMaterial

UMaterial is a concept belonging to the engine layer, corresponding to the uasset resource file we edited in the material editor, which can be applied to the mesh to control its visual effect in the scene. It inherits from UMaterialInterface, and their definitions are as follows:

```
// Engine\Source\Runtime\Engine\Classes\Materials\MaterialInterface.h

//The basic interface class of materials defines a large number of material-related data and interfaces. Some interfaces are
empty or unimplemented interfaces. classUMaterialInterface: public UObject, public IBlendableInterface, public
IInterface_AssetUserData
{
    //Subsurface scattering profile (configuration)
    classUSubsurfaceProfile*SubsurfaceProfile; //A fence that
    keeps track of when a primitive is no longer used as a parent.
    FRenderCommandFence ParentRefFence;

protected:
    // LightmassOfflineGIset up.
    structFLightmassMaterialInterfaceSettings //Texture        LightmassSettings;
    streaming data.
    TArray<FMaterialTextureInfo> TextureStreamingData; //A list of
    user data stored in this material asset.
    TArray<UAssetUserData*> AssetUserData;

private:
    //Force compilation targetsFeature Level.
```

```
    uint32 FeatureLevelsToForceCompile;

public:
    //---- IInterface_AssetUserDatainterface----
    virtualvoid AddAssetUserData(UAssetUserData* InUserData) override; virtualvoid RemoveUserDataOfClass
    (TSubclassOf<UAssetUserData> InUserDataClass) override;

    virtual UAssetUserData*GetAssetUserDataOfClass(TSubclassOf<UAssetUserData> InUserDataClass)
    override;
    //---- IInterface_AssetUserDatainterface----

    //Compiled for all materialsFeaturelevelBit field. static
    uint32 FeatureLevelsForAllMaterials;
    void SetFeatureLevelToCompile(ERHIFeatureLevel::Type FeatureLevel,bool bShouldCompile);

    static void SetGlobalRequiredFeatureLevel(ERHIFeatureLevel::Type FeatureLevel,bool bShouldCompile);


    //---- UObjectinterface----
    virtualvoid BeginDestroy() override; virtualbool
    IsReadyForFinishDestroy() override; virtualvoid PostLoad()
    override;
    virtualvoid PostDuplicate(boolbDuplicateForPIE) override; virtualvoid
    PostCDOContruct() override; //---- UObjectinterface----


    //---- IBlendableInterfaceinterface----
    virtualvoid OverrideBlendableSettings(class FSceneView& View,floatWeight)const override;

    //---- IBlendableInterfaceinterface----

    //Go down the parent chain to find the base material that this instance belongs to.
    UMaterial*GetBaseMaterial(); //Get the
    material being instantiated
    virtual class UMaterial*GetMaterial() PURE_VIRTUAL
(UMaterialInterface::GetMaterial,return                                    NULL;);
    virtualconstclass UMaterial*GetMaterial()const PURE_VIRTUAL
(UMaterialInterface::GetMaterial,returnNULL;);
    //Get the material being instantiated (concurrent mode)
    virtualconstclass UMaterial*GetMaterial_Concurrent(TMicRecursionGuard RecursionGuard =
TMicRecursionGuard())const
PURE_VIRTUAL(UMaterialInterface::GetMaterial_Concurrent,returnNULL;);


    //Tests whether this material depends on the specified material.
    virtualbool IsDependent(UMaterialInterface* TestDependency); virtualbool
    IsDependent_Concurrent(UMaterialInterface* TestDependency, TMicRecursionGuard
RecursionGuard = TMicRecursionGuard());

    //Get the corresponding material for renderingFMaterialRenderProxy
    Examples. virtual class FMaterialRenderProxy*GetRenderProxy()const
PURE_VIRTUAL(UMaterialInterface::GetRenderProxy,returnNULL;);

    (.....)

    //Get the list of textures used to render this material.
    virtualvoid GetUsedTextures(TArray<UTexture*>& OutTextures, EMaterialQualityLevel::Type QualityLevel,bool
bAllQualityLevels, ERHIFeatureLevel::Type FeatureLevel,boolbAllFeatureLevels)const
```

```
        PURE_VIRTUAL(UMaterialInterface::GetUsedTextures,); //Get the list and index
    of textures used to render this material.
    virtualvoid GetUsedTexturesAndIndices(TArray<UTexture*>& OutTextures, TArray< TArray<int32> >&
OutIndices, EMaterialQualityLevel::Type QualityLevel, ERHIFeatureLevel::Type FeatureLevel)const;

    //Overwrites the specified texture.
    virtualvoid OverrideTexture(constUTexture* InTextureToOverride, UTexture* OverrideTexture,
ERHIFeatureLevel::Type InFeatureLevel)
PURE_VIRTUAL(UMaterialInterface::OverrideTexture,return;);

    //Override the default value of a given parameter (temporarily)
    virtualvoid OverrideVectorParameterDefault(constFHashedMaterialParameterInfo& ParameterInfo,const
FLinearColor& Value,boolbOverride, ERHIFeatureLevel::Type FeatureLevel)PURE_VIRTUAL
(UMaterialInterface::OverrideTexture,return;);

    //Checks for the specified material tag and returns it if it existstrue.
    virtualbool CheckMaterialUsage(constEMaterialUsage Usage) PURE_VIRTUAL
(UMaterialInterface::CheckMaterialUsage,returnfalse;);
    virtualbool CheckMaterialUsage_Concurrent(constEMaterialUsage Usage)const PURE_VIRTUAL
(UMaterialInterface::CheckMaterialUsage,returnfalse;);

    //Get the render texture of the material, you need to specifyFeatureLevel/QualityLevel.
    virtual FMaterialResource*GetMaterialResource(ERHIFeatureLevel::Type InFeatureLevel,
EMaterialQualityLevel::Type QualityLevel = EMaterialQualityLevel::Num);

    //Get the group sort priority.
    virtualbool GetGroupSortPriority(constFString& InGroupName, int32& OutSortPriority) const

        PURE_VIRTUAL(UMaterialInterface::GetGroupSortPriority,returnfalse;);

    //Get all data of various types of materials.
    virtualvoid GetAllScalarParameterInfo(TArray<FMaterialParameterInfo>& OutParameterInfo,
TArray<FGuid>& OutParameterIds)const
        PURE_VIRTUAL(UMaterialInterface::GetAllScalarParameterInfo,return;); virtualvoid
    GetAllVectorParameterInfo(TArray<FMaterialParameterInfo>& OutParameterInfo,
TArray<FGuid>& OutParameterIds)const
        PURE_VIRTUAL(UMaterialInterface::GetAllVectorParameterInfo,return;); virtualvoid
    GetAllTextureParameterInfo(TArray<FMaterialParameterInfo>& OutParameterInfo,
TArray<FGuid>& OutParameterIds)const
        PURE_VIRTUAL(UMaterialInterface::GetAllTextureParameterInfo,return;); virtualvoid
    GetAllRuntimeVirtualTextureParameterInfo(TArray<FMaterialParameterInfo>& OutParameterInfo, TArray<FGuid>&
OutParameterIds)const
        PURE_VIRTUAL(UMaterialInterface::GetAllRuntimeVirtualTextureParameterInfo, return;);

    virtualvoid GetAllFontParameterInfo(TArray<FMaterialParameterInfo>& OutParameterInfo, TArray<FGuid>&
OutParameterIds)const
        PURE_VIRTUAL(UMaterialInterface::GetAllFontParameterInfo,return;);

    //Get various types of data about the material.
    virtualbool GetScalarParameterDefaultValue(constFHashedMaterialParameterInfo& ParameterInfo,float&
OutValue,boolbOveriddenOnly =false,bool bCheckOwnedGlobalOverrides =false)const

        PURE_VIRTUAL(UMaterialInterface::GetScalarParameterDefaultValue,returnfalse;); virtualbool
    GetVectorParameterDefaultValue(constFHashedMaterialParameterInfo& ParameterInfo, FLinearColor&
OutValue,boolbOveriddenOnly =false,bool bCheckOwnedGlobalOverrides =false)const

        PURE_VIRTUAL(UMaterialInterface::GetVectorParameterDefaultValue,returnfalse;); virtualbool
    GetTextureParameterDefaultValue(constFHashedMaterialParameterInfo&
```

```
ParameterInfo, class UTexture*& OutValue,boolbCheckOwnedGlobalOverrides =false)const
        PURE_VIRTUAL(UMaterialInterface::GetTextureParameterDefaultValue,returnfalse;); virtualbool
    GetRuntimeVirtualTextureParameterDefaultValue(const FHashedMaterialParameterInfo& ParameterInfo, class
URuntimeVirtualTexture*& OutValue,bool bCheckOwnedGlobalOverrides =false)const

        PURE_VIRTUAL(UMaterialInterface::GetRuntimeVirtualTextureParameterDefaultValue, returnfalse;);

    virtualbool GetFontParameterDefaultValue(constFHashedMaterialParameterInfo& ParameterInfo, class
UFont*& OutFontValue, int32& OutFontPage,bool bCheckOwnedGlobalOverrides =false)const

        PURE_VIRTUAL(UMaterialInterface::GetFontParameterDefaultValue,returnfalse;);

    //Get hierarchical data.
    virtual int32GetLayerParameterIndex(EMaterialParameterAssociation Association,
UMaterialFunctionInterface * LayerFunction)const
        PURE_VIRTUAL(UMaterialInterface::GetLayerParameterIndex,returnINDEX_NONE;);

    //Gets the texture referenced by an expression, including nested functions.
    virtual TArrayView<UObject*const>GetReferencedTextures()const
        PURE_VIRTUAL(UMaterialInterface::GetReferencedTextures,return const>(););          TArrayView<UObject*


    //saveshaderStable key value.
    virtualvoid SaveShaderStableKeysInner(constclass ITargetPlatform* TP,conststruct FStableShaderKeyAndValue&
    SaveKeyVal)
        PURE_VIRTUAL(UMaterialInterface::SaveShaderStableKeysInner, );

    //Get material parameter information.
    FMaterialParameterInfoGetParameterInfo(EMaterialParameterAssociation FName                Association,
ParameterName, UMaterialFunctionInterface* LayerFunction)const;
    //Get the material association flag.
    ENGINE_API FMaterialRelevanceGetRelevance(ERHIFeatureLevel::Type InFeatureLevel) const;

    ENGINE_API    FMaterialRelevanceGetRelevance_Concurrent(ERHIFeatureLevel::Type
InFeatureLevel)       const;

#if!(UE_BUILD_SHIPPING || UE_BUILD_TEST) //Document
    materials and textures.
    ENGINE_API virtualvoid LogMaterialsAndTextures(FOutputDevice& Ar, int32 Indent)const
{}
#endif

private:
    int32    GetWidth()     const;
    int32    GetHeight()      const;

    // Lightmassinterface.
    constFGuid&GetLightingGuid()const; void
    SetLightingGuid();
    virtualvoid GetLightingGuidChain(boolbIncludeTextures, TArray<FGuid>& OutGuids) const;

    virtualbool UpdateLightmassTextureTracking(); inlinebool
    GetOverrideCastShadowAsMasked()const; inlinebool
    GetOverrideEmissiveBoost()const; (.....)


    //Data acquisition interface.
    virtualbool GetScalarParameterValue(constFHashedMaterialParameterInfo&
```

```
ParameterInfo,float& OutValue,boolbOveriddenOnly =false)const;
        virtualbool GetScalarCurveParameterValue(constFHashedMaterialParameterInfo& ParameterInfo,
FInterpCurveFloat& OutValue)const;
        virtualbool GetVectorParameterValue(constFHashedMaterialParameterInfo& ParameterInfo,
FLinearColor& OutValue,boolbOveriddenOnly =false)const;
        virtualbool GetVectorCurveParameterValue(constFHashedMaterialParameterInfo& ParameterInfo,
FInterpCurveVector& OutValue)const;
        virtualbool GetLinearColorParameterValue(constFHashedMaterialParameterInfo& ParameterInfo,
FLinearColor& OutValue)const;
        virtualbool GetLinearColorCurveParameterValue(constFHashedMaterialParameterInfo& ParameterInfo,
FInterpCurveLinearColor& OutValue)const;
        virtualbool GetTextureParameterValue(constFHashedMaterialParameterInfo& ParameterInfo, class
UTexture*& OutValue,boolbOveriddenOnly =false)const;
        virtualbool GetRuntimeVirtualTextureParameterValue(const FHashedMaterialParameterInfo& ParameterInfo, class
URuntimeVirtualTexture*& OutValue,bool bOveriddenOnly =false)const;

        virtualbool GetFontParameterValue(constFHashedMaterialParameterInfo& ParameterInfo,class UFont*&
OutFontValue, int32& OutFontPage,boolbOveriddenOnly =false) const;

        virtualbool GetRefractionSettings(float& OutBiasValue)const;

        //Access the override properties of the base material.
        virtualfloat GetOpacityMaskClipValue()const; virtualbool
        GetCastDynamicShadowAsMasked()const; virtual EBlendMode
        GetBlendMode()const;
        virtual FMaterialShadingModelFieldGetShadingModels()const; virtualbool
        IsShadingModelFromMaterialExpression()const; virtualbool IsTwoSided()const;

        virtualbool              IsDitheredLODTransition()const;              virtualbool
        IsTranslucencyWritingCustomDepth()const;                              virtualbool
        IsTranslucencyWritingVelocity()const;  virtualbool  IsMasked()const;  virtual
        bool IsDeferredDecal()const;

        virtual USubsurfaceProfile*GetSubsurfaceProfile_Internal()const; virtualbool
        CastsRayTracedShadows()const;

        //Forces the streaming system to ignore the normal logic of specifying a duration and instead always load all
        textures used by this material.miplevel. virtualvoid SetForceMipLevelsToBeResident(bool
OverrideForceMiplevelsToBeResident, boolbForceMiplevelsToBeResidentValue,floatForceDuration, int32
CinematicTextureGroups = 0,boolbFastResponse =false);

        //Re-cache unified expressions for all material interfaces.
        static void RecacheAllMaterialUniformExpressions(boolbRecreateUniformBuffer); virtualvoid
        RecacheUniformExpressions(boolbRecreateUniformBuffer)const; //Initialize all system default materials.

        static void InitDefaultMaterials();
        virtualbool IsPropertyActive(EMaterialProperty InProperty)const; staticuint32
        GetFeatureLevelsToCompileForAllMaterials()

        //Returns the number of used texture coordinates, and whether the vertex data is used in the shader graph.
        void AnalyzeMaterialProperty(EMaterialProperty InProperty, int32&
OutNumTextureCoordinates,bool& bOutRequiresVertexData);

        //Traverse allFeatureLevel,A callback can be
        specified. template <typename FunctionType>
        static void IterateOverActiveFeatureLevels(FunctionType InHandler);

        //Access to material sampler type cacheUEnumType information.
```

```
    staticUEnum*GetSamplerTypeEnum();

    bool  UseAnyStreamingTexture()        const;
    bool HasTextureStreamingData()        const;
    constTArray<FMaterialTextureInfo>&GetTextureStreamingData()const; FORCEINLINE
    TArray<FMaterialTextureInfo>&GetTextureStreamingData(); //Texture streaming interface.

    bool FindTextureStreamingDataIndexRange(FName TextureName, int32& LowerIndex, int32& HigherIndex)const;

    void SetTextureStreamingData(constTArray<FMaterialTextureInfo>&
    InTextureStreamingData);
    //Returns the scale of the texture (LocalSpace Unit / Texture),Used for texture streaming matrices.
    virtualfloat GetTextureDensity(FName TextureName,conststructFMeshUVChannelInfo& UVChannelData)const;


    //Pre-save.
    virtualvoid PreSave(constclass ITargetPlatform* TargetPlatform) override; //Sort texture stream data
    by name to speed up searching. Sort only when necessary.
    void SortTextureStreamingData(boolbForceSort,boolbFinalSort);

protected:
    uint32 GetFeatureLevelsToCompileForRendering()const;
    void UpdateMaterialRenderProxy(FMaterialRenderProxy&                Proxy);

private:
    static void PostLoadDefaultMaterials(); //Material
    sampler type cacheUEnumType information. static
    UEnum* SamplerTypeEnum;
};


// Engine\Source\Runtime\Engine\Classes\Materials\Material.h

//Material class, corresponding to a material resource file.
classUMaterial: public UMaterialInterface {

    //Physical materials.
    class    UPhysicalMaterial*PhysMaterial;
    class    UPhysicalMaterialMask*PhysMaterialMask;
    class    UPhysicalMaterial*PhysicalMaterialMap[EPhysicalMaterialMaskColor::MAX];

    //Material properties.
    FScalarMaterialInput        Metallic;
    FScalarMaterialInput        Specular;
    FScalarMaterialInput        Anisotropy;
    FVectorMaterialInput        Normal;
    FVectorMaterialInput        Tangent;
    FColorMaterialInput        EmissiveColor;

#if  WITH_EDITORONLY_DATA
    //transmission.
    FScalarMaterialInput        Opacity;
    FScalarMaterialInput        OpacityMask;
#endif

    TEnumAsByte<enum  EMaterialDomain> MaterialDomain; EBlendMode>
    TEnumAsByte<enum  BlendMode; EDecalBlendMode> DecalBlendMode;
    TEnumAsByte<enum  EMaterialDecalResponse> MaterialDecalResponse;
    TEnumAsByte<enum
```

```
    TEnumAsByte<enumEMaterialShadingModel> ShadingModel;
    UPROPERTY(AssetRegistrySearchable)
    FMaterialShadingModelField ShadingModels;

public:
    //Material properties.
    floatOpacityMaskClipValue;
    FVectorMaterialInput        WorldPositionOffset;
    FScalarMaterialInput        Refraction;
    FMaterialAttributesInput FScalarMMataetreirailaInlAptutrti bPuixtelsD;epthOffset;
    FShadingModelMaterialInput ShadingModelFromMaterialExpression;


#if  WITH_EDITORONLY_DATA
    FVectorMaterialInput        WorldDisplacement;
    FScalarMaterialInput        TessellationMultiplier;
    FColorMaterialInput         SubsurfaceColor;
    FScalarMaterialInput        ClearCoat;
    FScalarMaterialInput        ClearCoatRoughness;
    FScalarMaterialInput        AmbientOcclusion;
    FVector2MaterialInput         CustomizedUVs[8];
#endif
    int32 NumCustomizedUVs;

    //Material tags.
    uint8 bCastDynamicShadowAsMasked:1; uint8
    bEnableSeparateTranslucency:1; uint8
    bEnableResponsiveAA :1; uint8
    bScreenSpaceReflections :1; uint8
    bContactShadows :1; uint8 TwoSided :1;

    uint8 DitheredLODTransition :1; uint8
    DitherOpacityMask :1;
    uint8 bAllowNegativeEmissiveColor:1;

    //Transparency related.
    TEnumAsByte<enumETranslucencyLightingMode> uint8     TranslucencyLightingMode;
    bEnableMobileSeparateTranslucency:1; float
            TranslucencyDirectionalLightingIntensity;
    float   TranslucentShadowDensityScale;
    float   TranslucentSelfShadowDensityScale;
    float   TranslucentSelfShadowSecondDensityScale;
    FLinearColor TranslucentSelfShadowSecondOpacity;
            TranslucentBackscatteringExponent;
                    TranslucentMultipleScatteringExtinction;
    floatTranslucentShadowStartOffset;

    //Use tags.
    uint8 bDisableDepthTest :1; uint8
    bWriteOnlyAlpha :1;
    uint8 bGenerateSphericalParticleNormals:1; uint8
    bTangentSpaceNormal :1;
    uint8 bUseEmissiveForDynamicAreaLighting:1; uint8
    bBlockGI :1;
    uint8 bUsedAsSpecialEngineMaterial:1; uint8
    bUsedWithSkeletalMesh :1; uint8
    bUsedWithEditorCompositing:1; uint8
    bUsedWithParticleSprites :1;
```

```
uint8 bUsedWithBeamTrails :1; uint8
bUsedWithMeshParticles:1; uint8 bUsedWithNiagaraSprites
:1; uint8 bUsedWithNiagaraRibbons:1; uint8
bUsedWithNiagaraMeshParticles:1; uint8
bUsedWithGeometryCache:1; uint8
bUsedWithStaticLighting:1; uint8 bUsedWithMorphTargets
:1; uint8 bUsedWithSplineMeshes:1; uint8
bUsedWithInstancedStaticMeshes:1; uint8
bUsedWithGeometryCollections :1; uint8 bUsesDistortion :
1; uint8 bUsedWithClothing :1; uint32 bUsedWithWater :1;
uint32 bUsedWithHairStrands :1; uint32
bUsedWithLidarPointCloud :1; uint32
bUsedWithVirtualHeightfieldMesh:1; uint8
bAutomaticallySetUsageInEditor:1; uint8 bFullyRough :1;
uint8 bUseFullPrecision :1; uint8
bUseLightmapDirectionality :1; uint8
bUseAlphaToCoverage :1;



uint32 bForwardRenderUsePreintegratedGFForSimpleIBL:1; uint8
bUseHQForwardReflections :1; uint8 bUsePlanarForwardReflections :1;


//Reduces roughness based on screen space normal variations.
uint8 bNormalCurvatureToRoughness:1; uint8
AllowTranslucentCustomDepthWrites :1; uint8 Wireframe :1
; //Coloring frequency.

TEnumAsByte<EMaterialShadingRate> ShadingRate; uint8
bCanMaskedBeAssumedOpaque:1; uint8 bIsPreviewMaterial :
1; uint8 bIsFunctionPreviewMaterial:1; uint8
bUseMaterialAttributes :1; uint8 bCastRayTracedShadows :1;
uint8 bUseTranslucencyVertexFog:1; uint8
bApplyCloudFogging :1; uint8 bIsSky :1;



uint8 bComputeFogPerPixel :1; uint8
bOutputTranslucentVelocity:1; uint8
bAllowDevelopmentShaderCompile:1; uint8
bIsMaterialEditorStatsMaterial:1; TEnumAsByte<enum
EBlendableLocation> uint8 BlendableOutputAlphaB:1l;e ndableLocation;
uint8 bEnableStencilTest :1;
TEnumAsByte<EMaterialStencilCompare> uint8
StencilRefValue =0; TEnumAsByte<enum
ERefractionMode> int32 BlendablePriority;
                                        StencilCompare;

                                                RefractionMode;

uint8 bIsBlendable :1; uint32
            UsageFlagWarnings;
float     RefractionDepthBias;
FGuid StateId;
float     MaxDisplacement;

//When the renderer needs to obtain the parameter value, it represents the material to the renderer.FMaterialRenderProxyderivative.
```

```
    classFDefaultMaterialInstance*DefaultMaterialInstance;

#if  WITH_EDITORONLY_DATA
    //Editor parameters.
    TMap<FName, TArray<UMaterialExpression*> > EditorParameters; //Material Graph.
    Data behind the editor model.

    classUMaterialGraph*            MaterialGraph;
#endif

private:
    //Inline material resource serialized from the site.    By the game threadPostLoaddeal with.
    TArray<FMaterialResource> //A list of          LoadedMaterialResources;
    resources to use for rendering materials.
    TArray<FMaterialResource*>          MaterialResources;
#if  WITH_EDITOR
    //The material asset being cached or baked.
    TMap<constclassITargetPlatform*,TArray<FMaterialResource*>>
    CachedMaterialResourcesForCooking;

#endif

    //To ensure that this is used before cleaningUMaterialComplete the various
    resources inRT. FThreadSafeBool ReleasedByRT;
    FMaterialCachedExpressionData            CachedExpressionData;

public:
    //~ Begin UMaterialInterface Interface. virtual UMaterial*GetMaterial()
    override; virtualconstUMaterial*GetMaterial()constoverride;

    virtualconstUMaterial*GetMaterial_Concurrent(TMicRecursionGuard RecursionGuard = TMicRecursionGuard())
constoverride;
    virtualbool GetScalarParameterValue(...)constoverride; (......)

    void SetShadingModel(EMaterialShadingModel NewModel);
    virtualbool IsPropertyActive(EMaterialProperty InProperty)constoverride; //~ End UMaterialInterface
    Interface.

    //~ Begin UObject Interface
    virtualvoid PreSave(constclass ITargetPlatform* TargetPlatform) override; virtualvoid
    PostInitProperties() override; virtualvoid Serialize(FArchive& Ar) override;

    virtualvoid PostDuplicate(boolbDuplicateForPIE) override; virtualvoid PostLoad()
    override; virtualvoid BeginDestroy() override; virtualbool
    IsReadyForFinishDestroy() override; virtualvoid FinishDestroy() override;


    virtualvoid GetResourceSizeEx(FResourceSizeEx& CumulativeResourceSize) override; static void
    AddReferencedObjects(UObject* InThis, FReferenceCollector& Collector); virtualbool CanBeClusterRoot()const
    override;
    virtualvoid GetAssetRegistryTags(TArray<FAssetRegistryTag>& OutTags)constoverride; //~ End UObject Interface


    //Data acquisition interface.
    bool  IsDefaultMaterial()          const;              ReleaseResources();
    void  IsUsageFlagDirty(EMaterialUsage
    bool  IsCompilinUgsaOgreH)a;dCompileError (ERHIFeatureLevel::Type
    bool                                                                InFeatureLevel);

    (......)
```

```
private:
    //New version of the interface for getting material data.
    bool GetScalarParameterValue_New(...) const;
    bool GetVectorParameterValue_New(...) const;
    bool GetTextureParameterValue_New(...) const;
    bool GetRuntimeVirtualTextureParameterValue_New(...) const;
    bool GetFontParameterValue_New(...)const;

    FStringGetUsageName(constEMaterialUsage Usage)const; bool
    GetUsageByFlag(constEMaterialUsage Usage)const;
    bool SetMaterialUsage(bool&bNeedsRecompile,constEMaterialUsage Usage);

    (......)

private:
    virtualvoid FlushResourceShaderMaps(); //Buffers
    resources or data.
    void CacheResourceShadersForRendering(bool bRegenerateId);
    void CacheResourceShadersForCooking(...);
    void CacheShadersForResources(...);

public:
    //Static toolbox or operation interface.
    staticUMaterial*GetDefaultMaterial(EMaterialDomain Domain); static
        void UpdateMaterialShaders(...);
    static void BackupMaterialShadersToMemory(...);
    static void RestoreMaterialShadersFromMemory(...);
    static void CompileMaterialsForRemoteRecompile(...);
    static bool GetExpressionParameterName(constUMaterialExpression* Expression, FName& OutName);

    static bool CopyExpressionParameters(UMaterialExpression* Source, UMaterialExpression* Destination);

    static bool IsParameter(constUMaterialExpression*Expression); static bool IsDynamicParameter(
    constUMaterialExpression*Expression); static constTCHAR*GetMaterialShadingModelString
    (EMaterialShadingModel InMaterialShadingModel);

    staticEMaterialShadingModelGetMaterialShadingModelFromString(constTCHAR*
    InMaterialShadingModelStr);
    static constTCHAR*GetBlendModeString(EBlendMode InBlendMode); staticEBlendMode
    GetBlendModeFromString(constTCHAR* InBlendModeStr); virtual TArrayView<UObject*const>
    GetReferencedTextures()constoverride final;

    (......)
};
```

UMaterialInterfaceis a basic abstract class that defines a set of common material properties and interfaces. The main data declared by UMaterialInterface are:

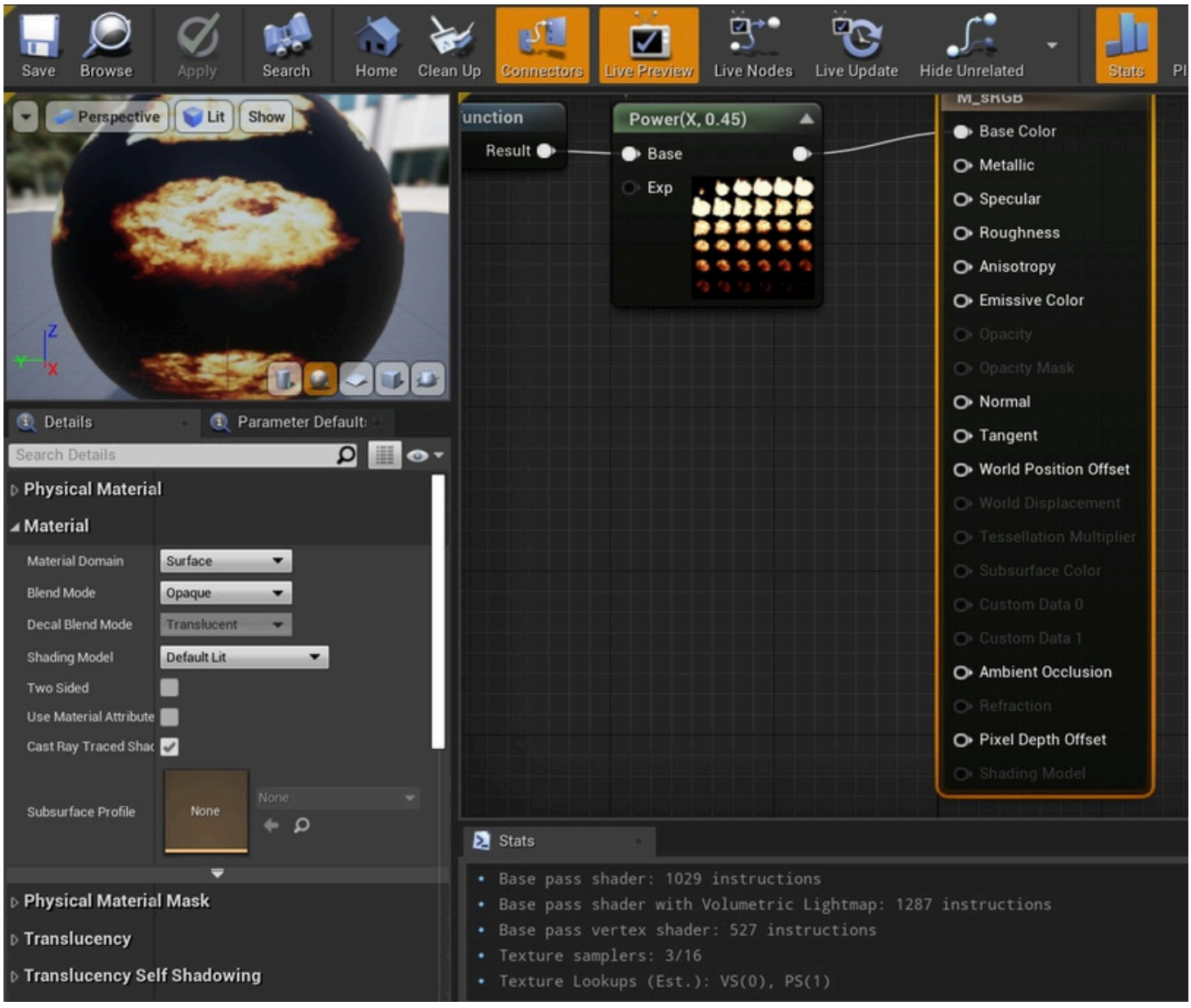
- USubsurfaceProfile* SubsurfaceProfile: Subsurface scattering profile, commonly used for materials such as skin and jade.
- TArray<UAssetUserData*> AssetUserData: user data, can store multiple data.
- FLightmassMaterialInterfaceSettings LightmassSettings: Offline GI data.
- TArrayTextureStreamingData: Texture streaming data.

The subclasses of UMaterialInterface are not only UMaterial, but also UMaterialInstance (which will be analyzed later).

UMaterialdefines all the data and operation interfaces required by the material, and is responsible for opening up links to other associated types. Its core data includes:

- TArray<FMaterialResource*> MaterialResources: Material rendering resources. A material can have multiple rendering resource instances.
- FDefaultMaterialInstance* DefaultMaterialInstance: The default material rendering proxy, inherited from FMaterialRenderProxy.
- UPhysicalMaterial* PhysMaterial: Physical material.
- TArrayLoadedMaterialResources: Loaded material resources, usually loaded from disk and serialized by the game thread.
- Various properties and flags for materials.

Among them, FMaterialResource is the resource of the rendering material, which belongs to the type and data of the rendering layer, and will be analyzed in the following chapters.
A UMaterial corresponds to the resource of the material editor, and is also the object we come into contact with most often:



The material editor we use most often is the one that edits the UMaterial instance. The options in the property panel on the left are consistent with the UMaterial declaration.

UMaterial has only one special subclass, UPreviewMaterial, which is mainly used for editor-related material previews, such as the preview window in the upper left corner of the material editor, the material thumbnail of the browser, and so on.

9.2.2 UMaterialInstance

UMaterialInstance is a material instance and cannot exist independently. Instead, it needs to rely on a parent class of the UMaterialInterface type, which means that the parent class can be any subclass of UMaterialInterface, but the top-level parent class must be UMaterial. It is defined as follows:

```
classUMaterialInstance: public UMaterialInterface {

    //Physical materials.
    class    UPhysicalMaterial*        PhysMaterial;
    class    UPhysicalMaterial*        PhysicalMaterialMap[EPhysicalMaterialMaskColor::MAX];
    //Material father.
    classUMaterialInterface*             Parent;
    //When the renderer needs to get the parameter value, it represents theFMaterialRenderProxy
    Subclass of . classFMaterialInstanceResource*Resource;

    //Can partially coverParentThe properties ofUMaterialIn comparison, it's just a small part.
    uint8    bHasStaticPermutationResource:1;
    uint8    bOverrideSubsurfaceProfile:1;
    uint8 TwoSided :1;
    uint8 DitheredLODTransition :1; uint8
    bCastDynamicShadowAsMasked:1;
    uint8 bIsShadingModelFromMaterialExpression:1;
    TEnumAsByte<EBlendMode> BlendMode;
    floatOpacityMaskClipValue;
    FMaterialShadingModelField ShadingModels;

    //coverParentvarious types of data.
    TArray<struct      FScalarParameterValue> FVectoSrcPaalraarmPaertaemrVeatleureV>a lues;
    TArray<struct      FTextureParameterValue>      VectorParameterValues;
    TArray<struct      FRuntimeVirtualTextureParametеTreVxatulure>ParameterValues;
    TArray<struct
RuntimeVirtualTextureParameterValues;
    TArray<structFFontParameterValue> struct      FontParameterValues;
    FMaterialInstanceBasePropertyOverrides                    BasePropertyOverrides;

    (.....)

private:
    FStaticParameterSet StaticParameters;
    FMaterialCachedParameters CachedLayerParameters;
    TArray<UObject*> CachedReferencedTextures; //The loaded
    material resource.
    TArray<FMaterialResource>          LoadedMaterialResources;
    TArray<FMaterialResource*>         StaticPermutationMaterialResources;
    FThreadSafeBool ReleasedByRT;
```

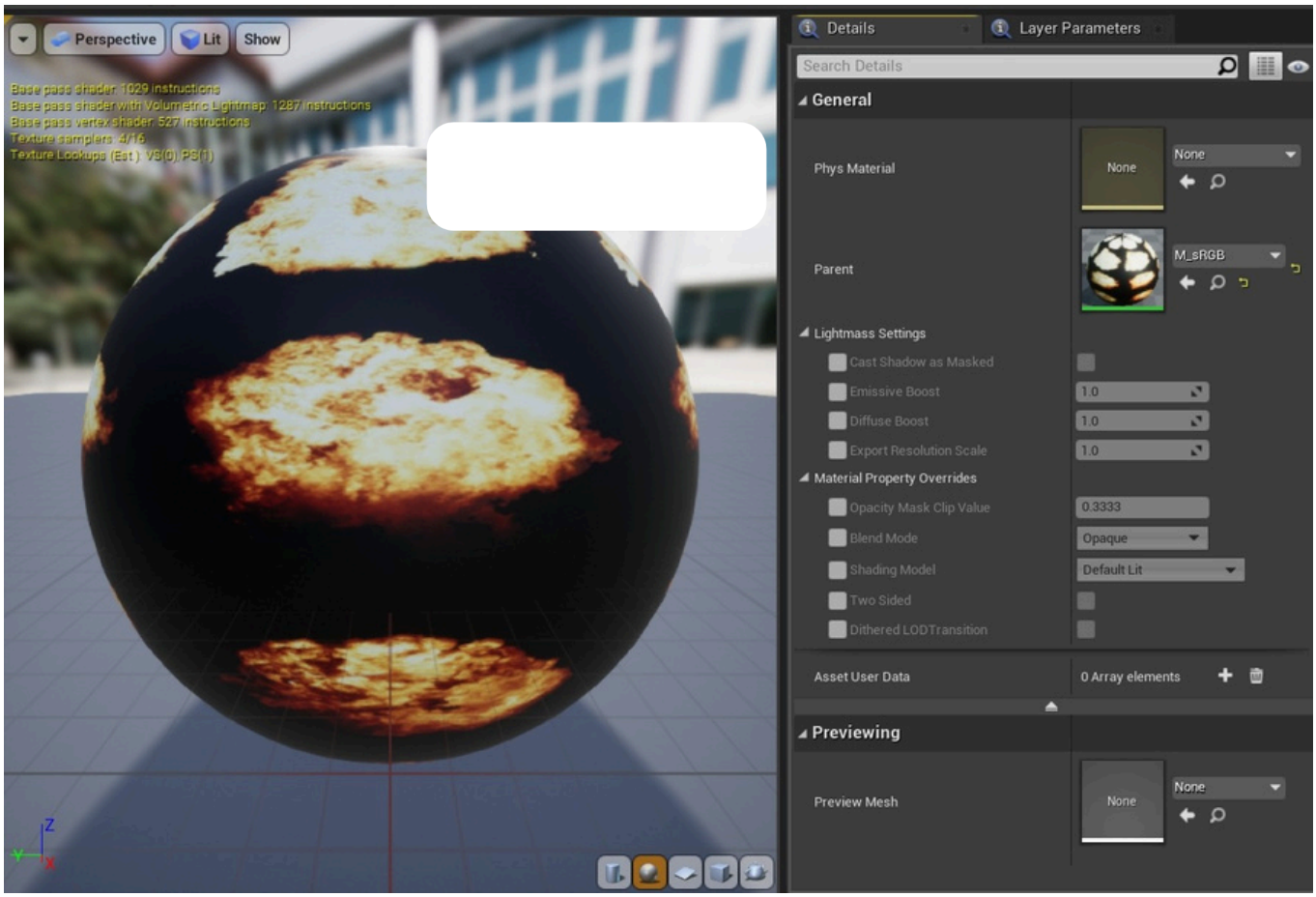
```
public:
	// Begin UMaterialInterface interface.
	virtual ENGINE_API UMaterial*GetMaterial() override;
	virtual ENGINE_APIconstUMaterial*GetMaterial()constoverride; virtual ENGINE_APIconstUMaterial*
	GetMaterial_Concurrent(TMicRecursionGuard RecursionGuard = TMicRecursionGuard())constoverride;

	virtual ENGINE_API FMaterialResource*AllocatePermutationResource(); (.....)

	//~ End UMaterialInterface Interface.

	//~ Begin UObject Interface.
	virtual ENGINE_APIvoid GetResourceSizeEx(FResourceSizeEx& CumulativeResourceSize) override;

	virtual ENGINE_APIvoid PostInitProperties() override; virtual ENGINE_APIvoid
	Serialize(FArchive& Ar) override; virtual ENGINE_APIvoid PostLoad() override;
	virtual ENGINE_APIvoid BeginDestroy() override; virtual ENGINE_APIbool
	IsReadyForFinishDestroy() override; virtual ENGINE_APIvoid FinishDestroy()
	override;

	ENGINE_APIstatic void AddReferencedObjects(UObject* InThis, FReferenceCollector& Collector);

	//~ End UObject Interface.

	void	GetAllShaderMaps(TArray<FMaterialShaderMap*>&		OutShaderMaps);
	void GetAllParametersOfType(EMaterialParameterType		Type,
TArray<FMaterialParameterInfo>& OutParameterInfo, TArray<FGuid>& OutParameterIds)const;

	(.....)

protected:
	void	CopyMaterialUniformParametersInternal(UMaterialInterface*			Source);
	bool UpdateParameters();
	ENGINE_APIvoid SetParentInternal(class UMaterialInterface* NewParent,bool RecacheShaders);


	(.....)

	//Initializes the resource for a material instance.
	ENGINE_APIvoid InitResources();

	//Cache resources.
	void	CacheResourceShadersForRendering(); CacheResourceShadersForRendering
	void (FMaterialResourceDeferredDeletionArray&
OutResourcesToFree);
	void CacheShadersForResources(...);
	void DeleteDeferredResources(FMaterialResourceDeferredDeletionArray&		ResourcesToFree);

	ENGINE_APIvoid CopyMaterialInstanceParameters(UMaterialInterface* Source);

	(.....)
};
```

UMaterialInstance is different from UMaterial, it needs to be attached to a parent instance, and the top-level parent must be a UMaterial instance. It can only cover a small part of the parameters of UMaterial, and is usually not created separately, but is created with its two subclasses **UMaterialInstanceConstant**and**UMaterialInstanceDynamic :**

```
// Engine\Source\Runtime\Engine\Classes\Materials\MaterialInstanceConstant.h

//Fixed Material Instance
classUMaterialInstanceConstant: public UMaterialInstance {

	//Editor Data
#ifWITH_EDITOR
	friend	class	UMaterialInstanceConstantFactoryNew;
	friend	class	UMaterialEditorInstanceConstant;
	virtual ENGINE_APIvoid PostEditChangeProperty(FPropertyChangedEvent&
PropertyChangedEvent) override;
#endif

	classUPhysicalMaterialMask*PhysMaterialMask;

	// Begin UMaterialInterface interface.
	virtualUPhysicalMaterialMask*GetPhysicalMaterialMask()constoverride; // End UMaterialInterface
	interface.

	float K2_GetScalarParameterValue(FName ParameterName);
	class UTexture*K2_GetTextureParameterValue(FName ParameterName); FLinearColor
	K2_GetVectorParameterValue(FName ParameterName);

	ENGINE_APIvoid PostLoad();

	(......)
};


// Engine\Source\Runtime\Engine\Classes\Materials\MaterialInstanceDynamic.h

//Dynamic Material Instances
classENGINE_API UMaterialInstanceDynamic: public UMaterialInstance {

	//To create a dynamic instance, you need to give a parent instance.
	staticUMaterialInstanceDynamic*Create(class UMaterialInterface* ParentMaterial, class UObject* InOuter);

	staticUMaterialInstanceDynamic*Create( class UMaterialInterface* ParentMaterial, class UObject* InOuter,
FName Name );

	//Copy from Material or Instance LevelUniformparameter((scalar, vector and texture).
	void	CopyMaterialUniformParameters(UMaterialInterface* Source);
	void	CopyInterpParameters(UMaterialInstance* Source); CopyParameterOverrides
	void	(UMaterialInstance* MaterialInstance);
	void CopyScalarAndVectorParameters(constUMaterialInterface& SourceMaterialToCopyFrom,
ERHIFeatureLevel::Type FeatureLevel);

	//Clean up parameters.
	void ClearParameterValues();

	//initialization.
	bool InitializeScalarParameterAndGetIndex(constFName& ParameterName,floatValue, int32&
	OutParameterIndex);
	bool InitializeVectorParameterAndGetIndex(constFName& ParameterName,const FLinearColor&
Value, int32& OutParameterIndex);

	//Data setting interface.
```

```
    void SetScalarParameterValue(FName ParameterName,floatValue);
    void SetScalarParameterValueByInfo(constFMaterialParameterInfo& ParameterInfo,float Value);

    bool SetScalarParameterByIndex(int32 ParameterIndex,floatValue);
    bool SetVectorParameterByIndex(int32 ParameterIndex,constFLinearColor& Value); void
    SetTextureParameterValue(FName ParameterName, class UTexture* Value);
    void SetTextureParameterValueByInfo(constFMaterialParameterInfo& ParameterInfo, class UTexture* Value);

    void SetVectorParameterValue(FName ParameterName, FLinearColor Value);
    void SetVectorParameterValueByInfo(constFMaterialParameterInfo& ParameterInfo, FLinearColor Value);

    void SetFontParameterValue(constFMaterialParameterInfo& ParameterInfo, class UFont* FontValue, int32
    FontPage);

    //Data acquisition interface.
    float K2_GetScalarParameterValue(FName ParameterName);
    float K2_GetScalarParameterValueByInfo(constFMaterialParameterInfo& ParameterInfo); class UTexture*
    K2_GetTextureParameterValue(FName ParameterName);
    class UTexture*K2_GetTextureParameterValueByInfo(constFMaterialParameterInfo& ParameterInfo);

    FLinearColor      K2_GetVectorParameterValue(FName ParameterName);
    FLinearColor      K2_GetVectorParameterValueByInfo(constFMaterialParameterInfo&
ParameterInfo);
    void K2_InterpolateMaterialInstanceParams(UMaterialInstance*                    Source A,
UMaterialInstance* SourceB,floatAlpha);
    void K2_CopyMaterialInstanceParameters(UMaterialInterface* Source,bool
bQuickParametersOnly =false);

    virtualbool HasOverridenBaseProperties()constoverride{returnfalse; } virtualfloat
    GetOpacityMaskClipValue()constoverride; virtualbool GetCastDynamicShadowAsMasked()const
    override; virtual FMaterialShadingModelFieldGetShadingModels()constoverride; virtualbool
    IsShadingModelFromMaterialExpression()constoverride; virtual EBlendModeGetBlendMode()
    constoverride; virtualbool IsTwoSided()constoverride;


    virtualbool IsDitheredLODTransition()constoverride; virtualbool IsMasked
    ()constoverride;

    //Each texture that has been renamed must be tracked in order to remap to the correct
    texture stream data. TMap<FName, TArray<FName> > RenamedTextures;

    //Gets the texture density.
    virtualfloat GetTextureDensity(FName TextureName,conststructFMeshUVChannelInfo& UVChannelData)const
    override; };
```

UMaterialInstanceConstant, as the name implies, is a fixed material instance, which is used for material instance resources that have been pre-created and edited by the editor:



An overview of the Material Instance Editor. The Material Property Overrides in the middle on the right shows the properties that can be edited and overridden by the Material Instance, which is quite limited.

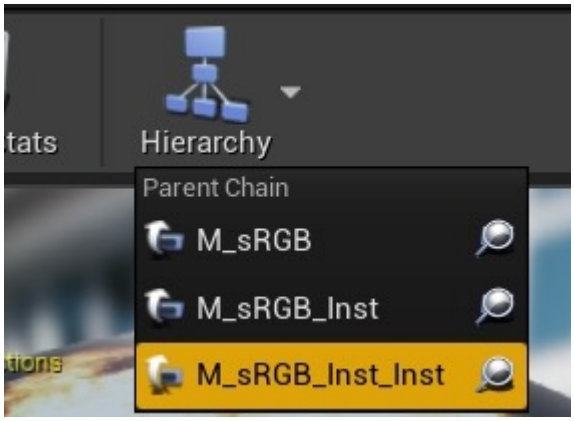
UMaterialInstanceConstant was created to avoid recompilation caused by modifying material parameters at runtime, which is why it has limited internal data coverage. Without recompilation, regular modifications to materials cannot be supported, so instances can only change the values of predefined material parameters. The parameters here are unique static definitions of name, type and default value defined in the material editor. In addition, it should be clearly noted that we cannot change the material properties of UMaterialInstanceConstant instances in runtime code (non-editor code). UMaterialInstanceConstant also has a subclass of ULandscapeMaterialInstanceConstant specifically for rendering terrain.

UMaterialInstanceDynamic is different from UMaterialInstanceConstant. It provides the function of dynamically creating and modifying material properties in runtime code, and it also does not cause the material to be recompiled. The UE built-in code contains a lot of UMaterialInstanceDynamic creation, setting and rendering code. Here is a use case:

```
// Engine\Source\Runtime\Engine\Private\Components\DecalComponent.cpp

classUMaterialInstanceDynamic*UDecalComponent::CreateDynamicMaterialInstance() {

    //Create a dynamic instance, whereDecalMaterialIs the parent of the dynamic instance, specified by the user
    UDecalComponentMaterial. UMaterialInstanceDynamic* Instance = UMaterialInstanceDynamic::Create(DecalMaterial,
    this);
```

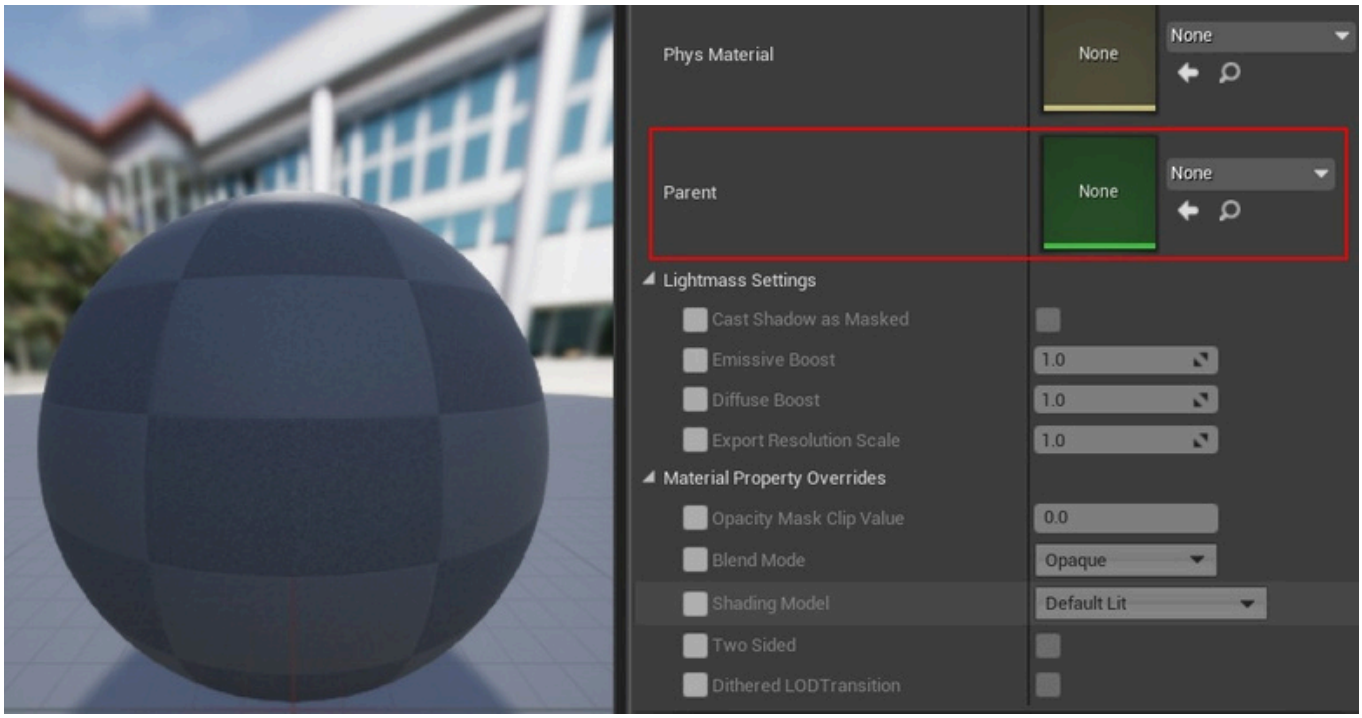
```
        //Overwrite with new decal material.
        SetDecalMaterial(Instance);

        returnInstance;
}
```

The inheritance relationship between UMaterialInstance can be any tree structure. Theoretically, there is no limit to the inheritance level (but too many inheritance levels increase the difficulty of maintenance and reduce operating efficiency). The top-level parent class must be a UMaterial instance, and the secondary and subsequent parent classes must be UMaterialInstance instances. The following figure shows the 4-level hierarchy displayed in the Material Instance Editor:



There are 4 levels of material instance hierarchy. The top level is UMaterial instance, and the 3 levels below are UMaterialInstance instances.

If you try to set the parent class of the top or middle layer to null, UE can tolerate it, but the rendering effect of the material will be abnormal:



9.2.3 FMaterialRenderProxy

We know that scene types such as primitives, meshes, and light sources have game thread representatives and rendering thread representatives, and materials are no exception. The rendering thread representative corresponding to the UMaterialInterface as the game thread representative is FMaterialRenderProxy. FMaterialRenderProxy is responsible for receiving data from the game thread representative and then passing it to the renderer for processing and rendering. The definition of FMaterialRenderProxy is as follows:

```
// Engine\Source\Runtime\Engine\Public\MaterialShared.h

classFMaterialRenderProxy: public FRenderResource {

public:
	//Cache data.
	mutable	FUniformExpressionCacheContainer UniformExpressionCache;
	mutable	FImmutableSamplerState ImmutableSamplerState;

	//Constructor/destructor.
	ENGINE_API	FMaterialRenderProxy(); virtual
	ENGINE_API	~FMaterialRenderProxy();

	//Evaluate the expression and store it inOutUniformExpressionCache.
	voidENGINE_APIEvaluateUniformExpressions(FUniformExpressionCache& OutUniformExpressionCache,const
FMaterialRenderContext& Context, class FRHICommandList* CommandListIfLocalMode = nullptr)const;


	// UniformExpressioninterface.
	voidENGINE_APICacheUniformExpressions(boolbRecreateUniformBuffer);
	voidENGINE_APICacheUniformExpressions_GameThread(boolbRecreateUniformBuffer); voidENGINE_API
	InvalidateUniformExpressionCache(boolbRecreateUniformBuffer); voidENGINE_API
	UpdateUniformExpressionCacheIfNeeded(ERHIFeatureLevel::Type InFeatureLevel)const;


	//Returns validFMaterialExamples.
	constclass FMaterial*GetMaterial(ERHIFeatureLevel::Type InFeatureLevel)const; //Find the code used to
	render thisFMaterialRenderProxyofFMaterialExamples.
	virtualconstFMaterial&GetMaterialWithFallback(ERHIFeatureLevel::Type InFeatureLevel,const
FMaterialRenderProxy*& OutFallbackMaterialRenderProxy)const=0;
	virtual FMaterial*GetMaterialNoFallback(ERHIFeatureLevel::Type InFeatureLevel)const {returnNULL; }

	//Get the correspondingUMaterialInterfaceExamples.
	virtual UMaterialInterface*GetMaterialInterface()const{returnNULL; }

	//Get the value of a material property.
	virtualbool GetVectorValue(constFHashedMaterialParameterInfo& ParameterInfo, FLinearColor*
OutValue,constFMaterialRenderContext& Context)const=0;
	virtualbool GetScalarValue(constFHashedMaterialParameterInfo& ParameterInfo,float* OutValue,const
FMaterialRenderContext& Context)const=0;
	virtualbool GetTextureValue(constFHashedMaterialParameterInfo& ParameterInfo,const UTexture** OutValue,
constFMaterialRenderContext& Context)const=0;
	virtualbool GetTextureValue(constFHashedMaterialParameterInfo& ParameterInfo,const
URuntimeVirtualTexture** OutValue,constFMaterialRenderContext& Context)const=0;

	bool	IsDeleted()const;
	void	MarkForGarbageCollection();
	bool	IsMarkedForGarbageCollection()		const;
```

```
    // FRenderResource interface.
    ENGINE_API virtualvoid InitDynamicRHI() override; ENGINE_API virtual
    void ReleaseDynamicRHI() override; ENGINE_API virtualvoid
    ReleaseResource() override;


    //Gets the static material rendering representation mapping table.
    ENGINE_APIstatic constTSet<FMaterialRenderProxy*>&GetMaterialRenderProxyMap();

    void SetSubsurfaceProfileRT(constUSubsurfaceProfile* Ptr); const
    USubsurfaceProfile*GetSubsurfaceProfileRT()const;

    ENGINE_APIstatic void UpdateDeferredCachedUniformExpressions(); staticinlinebool
    HasDeferredUniformExpressionCacheRequests();

    int32GetExpressionCacheSerialNumber()const{return
    UniformExpressionCacheSerialNumber; }

private:
    constUSubsurfaceProfile* SubsurfaceProfileRT; mutable int32
    UniformExpressionCacheSerialNumber =0;

    //Material tags.
    mutable int8 MarkedForGarbageCollection :1; mutable int8
    DeletedFlag :1; mutable int8 ReleaseResourceFlag:1;
    mutable int8 HasVirtualTextureCallbacks :1;



    //Tracks the render delegates for all materials in all scenes. Only accessible in the rendering thread. Used to propagate new shader
    maps to materials used in rendering. ENGINE_APIstaticTSet<FMaterialRenderProxy*> MaterialRenderProxyMap; ENGINE_APIstatic
    TSet<FMaterialRenderProxy*> DeferredUniformExpressionCacheRequests;
};
```

FMaterialRenderProxy is an abstract class that defines a static global material rendering proxy mapping table and an interface for obtaining FMaterial rendering instances. The specific logic is completed by subclasses, and its subclasses are:

- **FDefaultMaterialInstance**:The default representative instance for rendering UMaterial.
- **FMaterialInstanceResource**:A delegate for rendering a UMaterialInstance instance.
- FColoredMaterialRenderProxy: A material rendering proxy that overrides the material's color vector parameters.
- FLandscapeMaskMaterialRenderProxy: Landscape mask material rendering proxy.
- FLightmassMaterialProxy: Lightmass material rendering
- proxy. . . . . .

We will focus on two important subclasses: FDefaultMaterialInstance and FMaterialInstanceResource, which are defined as follows:

```
// Engine\Source\Runtime\Engine\Private\Materials\Material.cpp

//    For renderingUMaterialThe default rendering representation of , the default parameter values are stored inFMaterialUniformExpressionXxxParameterThe resource
elephant,
    value used to store the selected color.
classFDefaultMaterialInstance: public FMaterialRenderProxy
```

```
{
public:

	//Game thread destruction interface.
	void GameThread_Destroy() {

		FDefaultMaterialInstance* Resource = this;
		ENQUEUE_RENDER_COMMAND(FDestroyDefaultMaterialInstanceCommand)(
			[Resource](FRHICommandList& RHICmdList)
		{
			delete Resource;
		});
	}

	//  FMaterialRenderProxy interface. Get the
	//  material interface.
	virtualconstFMaterial&GetMaterialWithFallback(ERHIFeatureLevel::Type InFeatureLevel,const
FMaterialRenderProxy*& OutFallbackMaterialRenderProxy)const
	{
		constFMaterialResource* MaterialResource = Material-
> GetMaterialResource(InFeatureLevel);
		if(MaterialResource && MaterialResource->GetRenderingThreadShaderMap()) {

			return*MaterialResource;
		}

		OutFallbackMaterialRenderProxy = &GetFallbackRenderProxy();
		returnOutFallbackMaterialRenderProxy->GetMaterialWithFallback(InFeatureLevel,
		OutFallbackMaterialRenderProxy);
	}
	virtual FMaterial*GetMaterialNoFallback(ERHIFeatureLevel::Type InFeatureLevel)const {

		returnMaterial->GetMaterialResource(InFeatureLevel);
	}

	//Get the correspondingUMaterialInterfaceinterface.
	virtual UMaterialInterface*GetMaterialInterface()constoverride {

		returnMaterial;
	}

	//Gets the parameter values of a vector.
	virtualbool GetVectorValue(constFHashedMaterialParameterInfo& ParameterInfo, FLinearColor*
OutValue,constFMaterialRenderContext& Context)const
	{
		constFMaterialResource* MaterialResource = Material-
> GetMaterialResource(Context.Material.GetFeatureLevel()); if
(MaterialResource                    && MaterialResource->GetRenderingThreadShaderMap())
		{
			return    false;
		}
		else
		{
			return    GetFallbackRenderProxy().GetVectorValue(ParameterInfo, OutValue,
Context);
		}
	}
	//Get the value of a scalar parameter.
```

```
	virtualbool GetScalarValue(constFHashedMaterialParameterInfo& ParameterInfo,float* OutValue,const
FMaterialRenderContext& Context)const
	{
		constFMaterialResource* MaterialResource = Material-
> GetMaterialResource(Context.Material.GetFeatureLevel());
		if(MaterialResource && MaterialResource->GetRenderingThreadShaderMap()) {

			staticFNameNameSubsurfaceProfile(TEXT("__SubsurfaceProfile")); if
			(ParameterInfo.Name == NameSubsurfaceProfile) {

				constUSubsurfaceProfile* MySubsurfaceProfileRT =
GetSubsurfaceProfileRT();

				int32 AllocationId =0; if
				(MySubsurfaceProfileRT) {

					// can be optimized (cached)
					AllocationId =
GSubsurfaceProfileTextureObject.FindAllocationId(MySubsurfaceProfileRT);
				}
				else
				{
					// no profile specified means we use the default one stored at [0]
which is human skin
					AllocationId =0;
				}

				* OutValue = AllocationId /255.0f;

				returntrue;
			}

			return	false;
		}
		else
		{
			return	GetFallbackRenderProxy().GetScalarValue(ParameterInfo,		OutValue,
Context);
		}
	}
	//Gets the texture parameter value.
	virtualbool GetTextureValue(constFHashedMaterialParameterInfo& ParameterInfo,const UTexture** OutValue,
constFMaterialRenderContext& Context)const
	{
		constFMaterialResource* MaterialResource = Material-
> GetMaterialResource(Context.Material.GetFeatureLevel());
		if(MaterialResource && MaterialResource->GetRenderingThreadShaderMap()) {

			returnfalse;
		}
		else
		{
			return
GetFallbackRenderProxy().GetTextureValue(ParameterInfo,OutValue,Context);
		}
	}
	virtualbool GetTextureValue(constFHashedMaterialParameterInfo& ParameterInfo,const
URuntimeVirtualTexture** OutValue,constFMaterialRenderContext& Context)const
```

```
	{
		constFMaterialResource* MaterialResource = Material-
> GetMaterialResource(Context.Material.GetFeatureLevel());
		if(MaterialResource && MaterialResource->GetRenderingThreadShaderMap()) {

			return	false;
		}
		else
		{
			return	GetFallbackRenderProxy().GetTextureValue(ParameterInfo, OutValue,
Context);
		}
	}

	// FRenderResource interface.
	virtual FStringGetFriendlyName()const{returnMaterial->GetName(); }

	// Constructor.
	FDefaultMaterialInstance(UMaterial* InMaterial);

private:
	//Get the backup material rendering proxy.
	FMaterialRenderProxy&GetFallbackRenderProxy() {		const

		return*(UMaterial::GetDefaultMaterial(Material->MaterialDomain)-
> GetRenderProxy());
	}

	//The corresponding material instance.
	UMaterial* Material;
};


// Engine\Source\Runtime\Engine\Private\Materials\MaterialInstanceSupport.h

//RenderingUMaterialInstanceMaterial resources.
classFMaterialInstanceResource: public FMaterialRenderProxy {

public:

	//Stores name and value pairs for material instances.
	template <typename ValueType> struct
	TNamedParameter
	{
		FHashedMaterialParameterInfo		Info;
		ValueType Value;
	};

	FMaterialInstanceResource(UMaterialInstance* InOwner);

	void GameThread_Destroy() {

		FMaterialInstanceResource* Resource = this;
		ENQUEUE_RENDER_COMMAND(FDestroyMaterialInstanceResourceCommand)(
			[Resource](FRHICommandList& RHICmdList) {

				delete Resource;
			});
```

```
    }

    // FRenderResource interface.
    virtual FStringGetFriendlyName()constoverride {returnOwner->GetName(); }

    //  FMaterialRenderProxy interface. Get material
    //  rendering resources.
    virtualconstFMaterial&GetMaterialWithFallback(ERHIFeatureLevel::Type FeatureLevel, const
FMaterialRenderProxy*& OutFallbackMaterialRenderProxy)constoverride;
    virtual FMaterial*GetMaterialNoFallback(ERHIFeatureLevel::Type FeatureLevel)const override;

    virtual UMaterialInterface*GetMaterialInterface()constoverride;

    //Get the value of a material.
    virtualbool GetVectorValue(constFHashedMaterialParameterInfo& ParameterInfo, FLinearColor*
OutValue,constFMaterialRenderContext& Context)constoverride;
    virtualbool GetScalarValue(constFHashedMaterialParameterInfo& ParameterInfo,float* OutValue,const
FMaterialRenderContext& Context)constoverride;
    virtualbool GetTextureValue(constFHashedMaterialParameterInfo& ParameterInfo,const UTexture** OutValue,
constFMaterialRenderContext& Context)constoverride;
    virtualbool GetTextureValue(constFHashedMaterialParameterInfo& ParameterInfo,const
URuntimeVirtualTexture** OutValue,constFMaterialRenderContext& Context)constoverride;

    void GameThread_SetParent(UMaterialInterface* ParentMaterialInterface); void InitMIParameters(
    structFMaterialInstanceParameterSet& ParameterSet); void RenderThread_ClearParameters() {


        VectorParameterArray.Empty();
        ScalarParameterArray.Empty();
        TextureParameterArray.Empty();
        RuntimeVirtualTextureParameterArray.Empty();
        InvalidateUniformExpressionCache(false);
    }

    //Update parameters.
    template <typename ValueType>
    void RenderThread_UpdateParameter(constFHashedMaterialParameterInfo& ParameterInfo, constValueType&
    Value )
    {
        LLM_SCOPE(ELLMTag::MaterialInstance);

        InvalidateUniformExpressionCache(false); TArray<TNamedParameter<ValueType> >& ValueArray =
        GetValueArray<ValueType>(); constint32 ParameterCount = ValueArray.Num();

        for(int32 ParameterIndex =0; ParameterIndex < ParameterCount; ++ParameterIndex) {

            TNamedParameter<ValueType>& Parameter = ValueArray[ParameterIndex]; if
            (Parameter.Info == ParameterInfo) {

                Parameter.Value =       Value;
                return;
            }
        }
        TNamedParameter<ValueType>      NewParameter;
        NewParameter.Info       = ParameterInfo;
        NewParameter.Value      = Value;
        ValueArray.Add(NewParameter);
    }
```

```
        //Find the value of a parameter with the specified name.
        template <typename ValueType>
        constValueType*RenderThread_FindParameterByName(constFHashedMaterialParameterInfo& ParameterInfo)
        const
        {
                constTArray<TNamedParameter<ValueType> >& ValueArray = GetValueArray<ValueType>
();
                constint32 ParameterCount = ValueArray.Num();
                for(int32 ParameterIndex =0; ParameterIndex < ParameterCount; ++ParameterIndex) {

                        constTNamedParameter<ValueType>& Parameter = ValueArray[ParameterIndex]; if(Parameter.Info
                        == ParameterInfo) {

                                return&Parameter.Value;
                        }
                }
                returnNULL;
        }

private:
        template <typename ValueType> TArray<TNamedParameter<ValueType> >& GetValueArray();

        //The parent of the material instance.
        UMaterialInterface*          Parent;
        //The parent of the game thread.
        UMaterialInterface*
        //The material instance it belongs to.          GameThreadParent;

        UMaterialInstance*          Owner;

        //List of parameter values of various types.
        TArray<TNamedParameter<FLinearColor> > VectorParameterArray;
        TArray<TNamedParameter<float> > ScalarParameterArray;
        TArray<TNamedParameter<constUTexture*> > TextureParameterArray;
        TArray<TNamedParameter<constURuntimeVirtualTexture*> >
        RuntimeVirtualTextureParameterArray;
};
```

It is important to note that FMaterialRenderProxy will be processed by both the game thread and the rendering thread, so you need to be careful about the data access and interface calls between them. The one with GameThread is dedicated to the game thread, and the one with RenderThread is dedicated to the rendering thread. If there is no special instruction, it is generally (not absolutely) used for the rendering thread. If you mistakenly call an interface that should not be called or access data, a race condition will occur, causing random crashes , and increasing the difficulty of debugging exponentially.

9.2.4 FMaterial, FMaterialResource

FMaterialhas 3 functions:

- Represents the material-to-material compilation process and provides extensibility hooks (CompileProperty, etc.).
- Pass material data to the renderer and use functions to access material properties.

- Stores cached shader maps, and other transient outputs from compilation. This is necessary for asynchronous shader compilation.

Here is the definition of FMaterial:

```
// Engine\Source\Runtime\Engine\Public\MaterialShared.h

classFMaterial
{
public:
#if   UE_CHECK_FMATERIAL_LIFETIME
      uint32    AddRef()     const;
      uint32    Release()      const;
      inlineuint32GetRefCount()const{returnuint32(NumDebugRefs.GetValue()); }

      mutable FThreadSafeCounter NumDebugRefs;
#else
      FORCEINLINE uint32AddRef()const{return0u; } FORCEINLINE uint32
      Release()const{return0u; } FORCEINLINE uint32GetRefCount()const{
      return0u; }
#endif

      FMaterial();
      ENGINE_API virtual ~FMaterial();

      //cacheshader.
      ENGINE_APIbool CacheShaders(EShaderPlatform Platform,constITargetPlatform* TargetPlatform =
      nullptr);
      ENGINE_APIbool CacheShaders(constFMaterialShaderMapId& ShaderMapId, EShaderPlatform Platform,const
ITargetPlatform* TargetPlatform = nullptr);

      //Do you need to cache the specifiedshader typeof data.
      ENGINE_API virtualbool ShouldCache(EShaderPlatform Platform,constFShaderType* ShaderType,const
FVertexFactoryType* VertexFactoryType)const;
      ENGINE_APIbool ShouldCachePipeline(EShaderPlatform Platform,const FShaderPipelineType* PipelineType,
constFVertexFactoryType* VertexFactoryType)const;

      //Serialization.
      ENGINE_API virtualvoid LegacySerialize(FArchive& Ar); void
      SerializeInlineShaderMap(FArchive& Ar);

      // ShaderMapinterface.
      void   RegisterInlineShaderMap(bool            bLoadedByCookedMaterial);
      void   ReleaseShaderMap();
      void   DiscardShaderMap();

      //Material properties.
      ENGINE_API virtualvoid GetShaderMapId(EShaderPlatform Platform,const ITargetPlatform*
TargetPlatform, FMaterialShaderMapId& OutId)const;
      virtual EMaterialDomainGetMaterialDomain()const=0;// See EMaterialDomain. virtualbool IsTwoSided()
      const=0;
      virtualbool IsDitheredLODTransition()const=0;
      virtualbool IsTranslucencyWritingCustomDepth()const{returnfalse; } virtualbool
      IsTranslucencyWritingVelocity()const{returnfalse; } virtualbool IsTangentSpaceNormal()const{
      returnfalse; }
```

```
(……)

//Whether to save to disk.
virtualbool IsPersistent()const=0; //Get a material
instance.
virtual UMaterialInterface*GetMaterialInterface()const{returnNULL; }

ENGINE_APIbool HasValidGameThreadShaderMap()const; inlinebool
ShouldCastDynamicShadows()const; EMaterialQualityLevel::Type
GetQualityLevel()                                                    const

//Data access interface.
ENGINE_APIconstFUniformExpressionSet&GetUniformExpressions()const; ENGINE_API
TArrayView<constFMaterialTextureParameterInfo> GetUniformTextureExpressions
(EMaterialTextureParameterType Type)const;
ENGINE_API TArrayView<constFMaterialVectorParameterInfo>
GetUniformVectorParameterExpressions()const;
ENGINE_API TArrayView<constFMaterialScalarParameterInfo>
GetUniformScalarParameterExpressions()const;
inlineTArrayView<constFMaterialTextureParameterInfo> GetUniform2DTextureExpressions()
const{return GetUniformTextureExpressions(EMaterialTextureParameterType::Standard2D);
                                                                                    }
inlineTArrayView<constFMaterialTextureParameterInfo>
GetUniformCubeTextureExpressions()const{return
GetUniformTextureExpressions(EMaterialTextureParameterType::Cube); }
inlineTArrayView<constFMaterialTextureParameterInfo>
GetUniform2DArrayTextureExpressions()const{return
GetUniformTextureExpressions(EMaterialTextureParameterType::Array2D);          }
inlineTArrayView<constFMaterialTextureParameterInfo>
GetUniformVolumeTextureExpressions()const{return
GetUniformTextureExpressions(EMaterialTextureParameterType::Volume);          }
inlineTArrayView<constFMaterialTextureParameterInfo>
GetUniformVirtualTextureExpressions()const{return
GetUniformTextureExpressions(EMaterialTextureParameterType::Virtual);          }

constFStaticFeatureLevelGetFeatureLevel()const{returnFeatureLevel; } bool
GetUsesDynamicParameter()const;
ENGINE_APIbool RequiresSceneColorCopy_GameThread()const;ENGINE_APIbool
RequiresSceneColorCopy_RenderThread()const;ENGINE_APIbool
NeedsSceneTextures()const;ENGINE_APIbool NeedsGBuffer()const;ENGINE_API
bool UsesEyeAdaptation()const;

ENGINE_APIbool UsesGlobalDistanceField_GameThread()const;ENGINE_APIbool
UsesWorldPositionOffset_GameThread()const;

//Material tags.
ENGINE_APIbool MaterialModifiesMeshPosition_RenderThread()const;ENGINE_APIbool
MaterialModifiesMeshPosition_GameThread()const;ENGINE_APIbool
MaterialUsesPixelDepthOffset()const;ENGINE_APIbool
MaterialUsesDistanceCullFade_GameThread()const;ENGINE_APIbool
MaterialUsesSceneDepthLookup_RenderThread()const;ENGINE_APIbool
MaterialUsesSceneDepthLookup_GameThread()const;ENGINE_APIbool
UsesCustomDepthStencil_GameThread()const;ENGINE_APIbool
MaterialMayModifyMeshPosition()const;ENGINE_APIbool
MaterialUsesAnisotropy_GameThread()const;ENGINE_APIbool
MaterialUsesAnisotropy_RenderThread()const;

// shader mapinterface.
```

```
class FMaterialShaderMap*GetGameThreadShaderMap()const {

        returnGameThreadShaderMap;
    }
    void SetGameThreadShaderMap(FMaterialShaderMap* InMaterialShaderMap) {

        GameThreadShaderMap = InMaterialShaderMap;

        TRefCountPtr<FMaterialShaderMap> ShaderMap = GameThreadShaderMap;
        TRefCountPtr<FMaterial> Material = this;

        //Set the game threadshader mapSet to the rendering thread.
        ENQUEUE_RENDER_COMMAND(SetGameThreadShaderMap)([Material = MoveTemp(Material), ShaderMap
= MoveTemp(ShaderMap)](FRHICommandListImmediate& RHICmdList) mutable
        {
            Material->RenderingThreadShaderMap =        MoveTemp(ShaderMap);
        });
    }
    void SetInlineShaderMap(FMaterialShaderMap*             InMaterialShaderMap);
    ENGINE_API class FMaterialShaderMap*GetRenderingThreadShaderMap()const;ENGINE_APIvoid
    SetRenderingThreadShaderMap(constTRefCountPtr<FMaterialShaderMap>& InMaterialShaderMap);


    ENGINE_API virtualvoid AddReferencedObjects(FReferenceCollector& Collector);

    virtual TArrayView<UObject*const>GetReferencedTextures()const=0;

    //Getshader/shader pipeline.
    template<typename ShaderType>
    TShaderRef<ShaderType>GetShader(FVertexFactoryType* VertexFactoryType,consttypename
ShaderType::FPermutationDomain& PermutationVector,boolbFatalIfMissing =true)const;
    template <typename ShaderType>
    TShaderRef<ShaderType>GetShader(FVertexFactoryType* VertexFactoryType, int32 PermutationId =0,
boolbFatalIfMissing =true)const;
    ENGINE_API FShaderPipelineRefGetShaderPipeline(class FShaderPipelineType* ShaderPipelineType,
FVertexFactoryType* VertexFactoryType,boolbFatalIfNotFound =true) const;


    //Material interface.
    virtual FStringGetMaterialUsageDescription()const=0; virtualbool GetAllowDevelopmentShaderCompile()
    const{returntrue; } virtual EMaterialShaderMapUsage::TypeGetMaterialShaderMapUsage()const{return
    EMaterialShaderMapUsage::Default; }

    ENGINE_APIbool GetMaterialExpressionSource(FString& OutSource); ENGINE_APIbool
    WritesEveryPixel(boolbShadowPass =false)const; virtualvoid SetupExtaCompilationSettings(const
    EShaderPlatform Platform, FExtraShaderCompilerSettings& Settings)const;


    (.....)

protected:
    constFMaterialShaderMap*GetShaderMapToUse()const;

    virtual int32CompilePropertyAndSetMaterialProperty(EMaterialProperty Property, class FMaterialCompiler*
Compiler, EShaderFrequency OverrideShaderFrequency = SF_NumFrequencies, boolbUsePreviousFrameTime =false)const
=0;

    void SetQualityLevelProperties(ERHIFeatureLevel::Type InFeatureLevel,
```

```
EMaterialQualityLevel::Type InQualityLevel = EMaterialQualityLevel::Num);
        virtualEMaterialShaderMapUsage::TypeGetShaderMapUsage()const; virtual FGuid
        GetMaterialId()const=0;
        ENGINE_APIvoid GetDependentShaderAndVFTypes(EShaderPlatform Platform,
TArray<FShaderType*>& OutShaderTypes, TArray<constFShaderPipelineType*>&
OutShaderPipelineTypes, TArray<FVertexFactoryType*>& OutVFTypes)const;
        bool GetLoadedCookedShaderMapId()const;

private:
        //Game thread and rendering threadshader
        map. TRefCountPtr<FMaterialShaderMap> GameThreadShaderMap;
        TRefCountPtr<FMaterialShaderMap>            RenderingThreadShaderMap;

        //Quality grade.
        EMaterialQualityLevel::Type QualityLevel;
        ERHIFeatureLevel::Type FeatureLevel;

        //Special markings.
        uint32 bStencilDitheredLOD :1; uint32
        bContainsInlineShaders:1; uint32
        bLoadedCookedShaderMapId:1;

        bool BeginCompileShaderMap(
                const FMaterialShaderMapId&           ShaderMapId,
                const FStaticParameterSet             &StaticParameterSet,
                EShaderPlatform       Platform,
                TRefCountPtr<class          FMaterialShaderMap>&       OutShaderMap,
                constITargetPlatform* TargetPlatform = nullptr); void
        SetupMaterialEnvironment(
                EShaderPlatform       Platform,
                const    FShaderParametersMetadata& InUniformBufferStruct,
                FShaderCompilerEnvironment&  InUniformExpressionSet&& InUniformExpressionSet,
                                                    OutEnvironment
                    const;

        ENGINE_API TShaderRef<FShader>GetShader(class FMeshMaterialShaderType* ShaderType, FVertexFactoryType*
VertexFactoryType, int32 PermutationId,boolbFatalIfMissing =true) const;

};
```

As can be seen above, FMaterial is a collection of all data and operation interfaces, including materials, shaders, vertexfactories, shader pipelines, shader maps, etc. It is the distribution center of these data. However, it is just an abstract parent class , and specific functions need to be implemented by subclasses. Its only subclass is FMaterialResource:

```
//accomplishFMaterialInterface for renderingUMaterialorUMaterialInstance.
classFMaterialResource: public FMaterial {

public:
        ENGINE_API     FMaterialResource();
        ENGINE_API     virtual ~FMaterialResource();

        //Set the material.
        void SetMaterial(UMaterial* InMaterial, UMaterialInstance* InInstance, ERHIFeatureLevel::Type
InFeatureLevel, EMaterialQualityLevel::Type InQualityLevel = EMaterialQualityLevel::Num)
```

```
    {
        Material = InMaterial;
        MaterialInstance = InInstance;
        SetQualityLevelProperties(InFeatureLevel,                InQualityLevel);
    }

    ENGINE_API uint32GetNumVirtualTextureStacks()const;
    ENGINE_API virtual FStringGetMaterialUsageDescription()constoverride;

    // FMaterial interface.
    ENGINE_API virtualvoid GetShaderMapId(EShaderPlatform Platform,const ITargetPlatform*
TargetPlatform, FMaterialShaderMapId& OutId)constoverride;
    ENGINE_API virtualEMaterialDomainGetMaterialDomain()constoverride; ENGINE_API virtualbool
    IsTwoSided()constoverride; ENGINE_API virtualbool IsDitheredLODTransition()constoverride;
    ENGINE_API virtualbool IsTranslucencyWritingCustomDepth()constoverride; ENGINE_API virtualbool
    IsTranslucencyWritingVelocity()constoverride; ENGINE_API virtualbool IsTangentSpaceNormal()const
    override; ENGINE_API virtual EMaterialShadingRateGetShadingRate()constoverride;



    (.....)

    //Material interface.
    inlineconstUMaterial*GetMaterial()const{returnMaterial; }
    inlineconstUMaterialInstance*GetMaterialInstance()const{returnMaterialInstance;
}
    inlinevoid SetMaterial(UMaterial* InMaterial) { Material = InMaterial; } inlinevoid
    SetMaterialInstance(UMaterialInstance* InMaterialInstance) { MaterialInstance =
    InMaterialInstance; }

protected:
    //The corresponding material.
    UMaterial* Material;
    //The corresponding material instance.
    UMaterialInstance* MaterialInstance;


    //Compile the entry for specifying material properties.SetMaterialPropertyCall.
    ENGINE_API virtual int32CompilePropertyAndSetMaterialProperty(EMaterialProperty Property, class
FMaterialCompiler* Compiler, EShaderFrequency OverrideShaderFrequency, boolbUsePreviousFrameTime)const
override;

    ENGINE_API virtualbool HasVertexPositionOffsetConnected()constoverride; ENGINE_API virtualbool
    HasPixelDepthOffsetConnected()constoverride; ENGINE_API virtualbool
    HasMaterialAttributesConnected()constoverride;

    (.....)
};
```

FMaterialResource only implements the interface that FMaterial does not implement, and stores instances of UMaterial or UMaterialInstance. If both instances of UMaterialInstance and UMaterial are valid, then the overlapping data of UMaterialInstance will take precedence, for example:

```
//Get the shading model domain
FMaterialShadingModelFieldFMaterialResource::GetShadingModels() {                    const

    //PriorityMaterialInstanceof data.
```

```
    returnMaterialInstance ? MaterialInstance->GetShadingModels() : Material-
> GetShadingModels();
}
```

It should be noted that FMaterialResource must ensure that the UMaterial instance is valid, and MaterialInstance can be empty.

FMaterialResource also has a subclass FLandscapeMaterialResource, which corresponds to the material for rendering ULandscapeMaterialInstanceConstant.

In addition to FMaterial, another core concept of rendering resources is FMaterialRenderContext, which saves the association pairing between FMaterialRenderProxy and FMaterial:

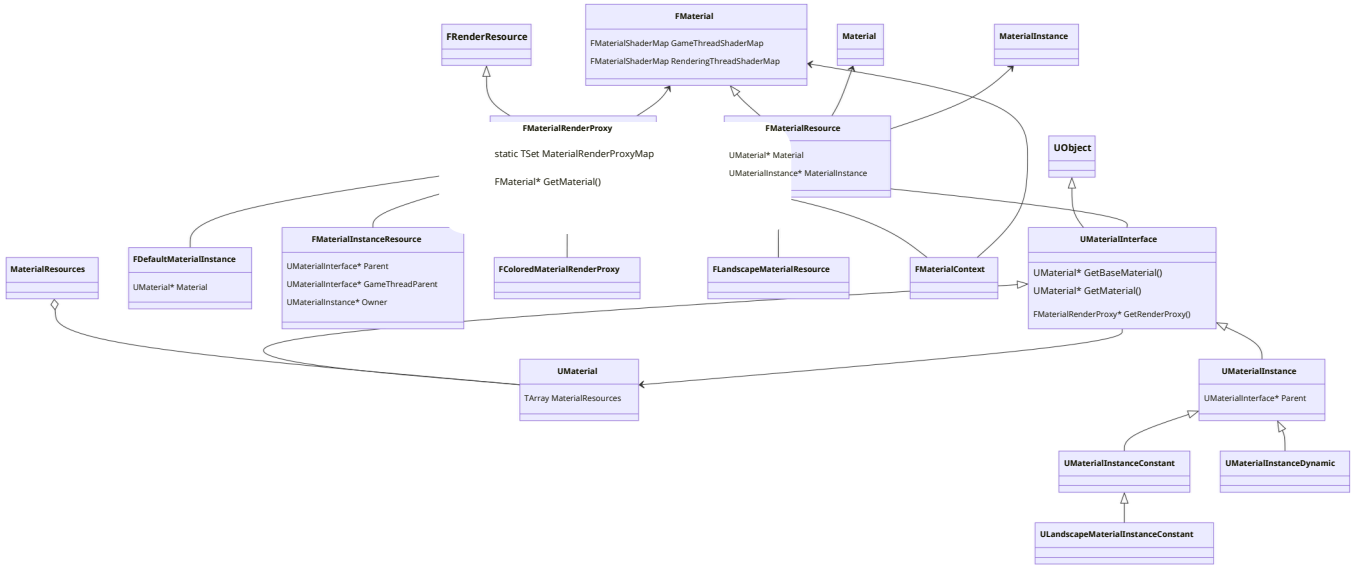
```
struct    ENGINE_API FMaterialRenderContext
{
    //    Used for materialshaderMaterial rendering representation of .
    const    FMaterialRenderProxy* MaterialRenderProxy;
    //Material rendering resources.
    constFMaterial&          Material;

    //Whether to display the selected color.
    boolbShowSelection;

    //Constructor.
    FMaterialRenderContext(constFMaterialRenderProxy* InMaterialRenderProxy,const FMaterial& InMaterial,
constFSceneView* InView); };
```

FMaterialRenderContext is mostly used for parameters of various types of material interfaces, such as:

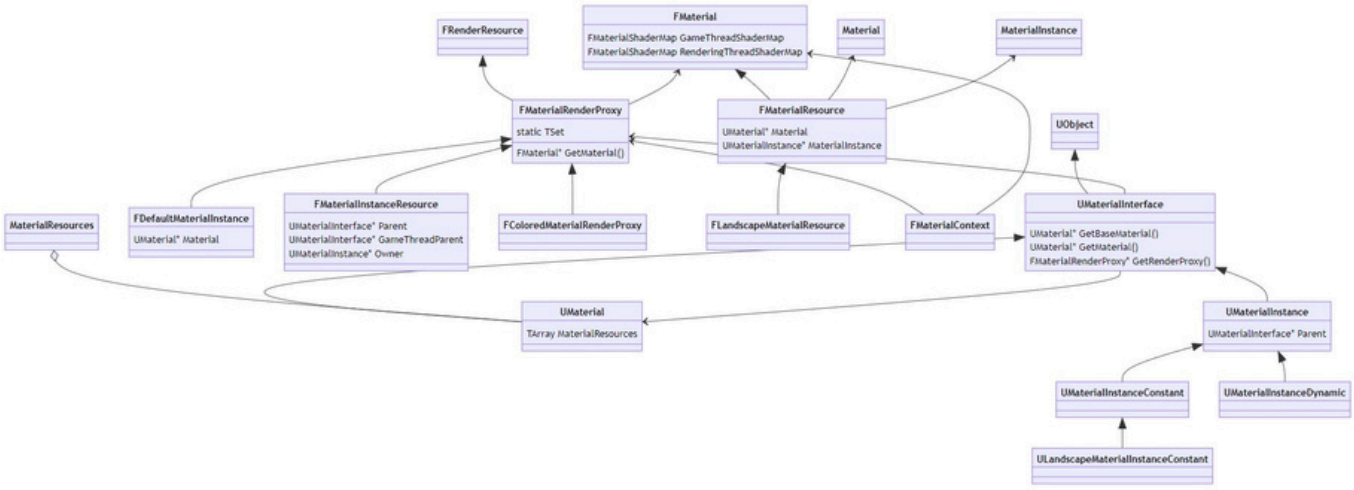
```
// FDefaultMaterialInstanceIn obtaining the vector parameter value, we useFMaterialRenderContextparameter.
virtualbool FDefaultMaterialInstance::GetVectorValue(constFHashedMaterialParameterInfo& ParameterInfo,
FLinearColor* OutValue,constFMaterialRenderContext& Context)const {

    constFMaterialResource* MaterialResource = Material-
> GetMaterialResource(Context.Material.GetFeatureLevel());

    if(MaterialResource && MaterialResource->GetRenderingThreadShaderMap()) {

        returnfalse;
    }
    else
    {
        returnGetFallbackRenderProxy().GetVectorValue(ParameterInfo, OutValue, Context);
    }
}
```

9.2.5 Material Overview

The previous sections have elaborated on the basic types, concepts and definitions of the material system. This section will directly show their UML diagrams to give an overview of their relationships:



If the text in the image above is too small to see clearly, you can click to enlarge the image version below:



Why are there so many concepts and types of materials in UE? What is the relationship between them? This section attempts to explain their relationship and function.

First, UMaterialInterface and its subclasses are explained. They are the representatives of the engine module in the game thread. UMaterialInterface inherits UOjbect and provides an abstract interface for materials, providing consistent behaviors and specifications for subclasses, and unifying the differences between different types of subclasses. The subclass UMaterial corresponds to the resources of the material blueprint generated by the material editor, saving various expression nodes and various parameters. Another subclass, UMaterialInstance , abstracts the interface of the material instance. It exists to support the modification of material parameters without triggering material recompilation. At the same time, it unifies and standardizes the data and behaviors of the two subclasses of fixed instances (UMaterialInstanceConstant) and dynamic instances (UMaterialInstanceDynamic). UMaterialInstanceConstant creates and modifies material parameters during the editor, and cannot be modified at runtime, improving the performance of data update and rendering; UMaterialInstanceDynamic can create instances and modify data at runtime, improving the scalability and customizability of materials, but its performance is worse than

UMaterialInstanceConstant. UMaterialInstance needs to specify a parent class, and the top-level parent class is required to be a UMaterial instance.

FMaterialRenderProxy is the representative of the rendering thread of UMaterialInterface, similar to the relationship between UPrimitiveComponent and FPrimitiveSceneProxy. FMaterialRenderProxy moves (copies) the data of the UMaterialInterface instance to the rendering thread, but it will also be accessed by the game thread at the same time. It is a coupling type of the two threads, and their data and interface calls need to be handled with caution. The subclass of FMaterialRenderProxy corresponds to the subclass of UMaterialInterface, so that the subclass data of UMaterialInterface
can be accurately moved (copied) to the rendering thread to avoid competition between the game thread and the rendering thread. FMaterialRenderProxy and its subclasses are all types of engine modules.

Since there is already a rendering thread delegate of FMaterialRenderProxy, why do FMaterial and FMaterialResource still exist? The answer is twofold:

- FMaterialRenderProxy and its subclasses are types of engine modules. They are capsule classes of the game thread and the rendering thread. The data and interface calls of the two threads need to be handled carefully, and the rendering module cannot really have full jurisdiction over it.

- The data of FMaterialRenderProxy is passed from UMaterialInterface, which means that the information of FMaterialRenderProxy is limited and cannot contain other information of the mesh using the material, such as vertex factory, ShaderMap, ShaderPipeline, FShader and various shader parameters.

Therefore, FMaterial came into being. FMaterial is also a type of engine module, but it stores two ShaderMaps for the game thread and the rendering thread, which means that the rendering module can freely access the ShaderMap of the rendering thread without affecting the access of the game thread. Moreover, FMaterial contains all the data required for rendering materials. As long as other places in the renderer get the FMaterial of the mesh, they can normally obtain the material data and submit drawing instructions. For example, the code of FBasePassMeshProcessor:: AddMeshBatch:

```
// Engine\Source\Runtime\Renderer\Private\BasePassRendering.cpp

void FBasePassMeshProcessor::AddMeshBatch(constFMeshBatch& RESTRICT MeshBatch, uint64 BatchElementMask,
constFPrimitiveSceneProxy* RESTRICT PrimitiveSceneProxy, int32 StaticMeshId)

{
	if(MeshBatch.bUseForMaterial) {

		constFMaterialRenderProxy* FallbackMaterialRenderProxyPtr = nullptr; //GetFMaterial
		Examples.
		constFMaterial& Material = MeshBatch.MaterialRenderProxy-
> GetMaterialWithFallback(FeatureLevel, FallbackMaterialRenderProxyPtr);
		constFMaterialRenderProxy& MaterialRenderProxy = FallbackMaterialRenderProxyPtr?
* FallbackMaterialRenderProxyPtr: *MeshBatch.MaterialRenderProxy;

		//passFMaterialInterface to obtain material data.
```

```
        constEBlendMode BlendMode = Material.GetBlendMode();
        constFMaterialShadingModelField ShadingModels = Material.GetShadingModels(); const bool
        bIsTranslucent = IsTranslucentBlendMode(BlendMode); constFMeshDrawingPolicyOverrideSettings
        OverrideSettings = ComputeMeshOverrideSettings(MeshBatch);

        constERasterizerFillMode MeshFillMode = ComputeMeshFillMode(MeshBatch, Material, OverrideSettings);

        constERasterizerCullMode MeshCullMode = ComputeMeshCullMode(MeshBatch, Material, OverrideSettings);


    (……)
}
```

9.3 Material Mechanism

This chapter mainly some underlying mechanisms of materials, such as material rendering, analysis compilation mechanism and process, material caching strategy, etc.

9.3.1 Material Rendering

This section will explain the data transfer, update, and rendering logic of the material. The originator of material data is still the resource on the game thread side, which is usually a binary resource loaded from disk and then serialized into a UMaterialInterface instance, or dynamically created and set by the runtime. However, most of them are loaded from disk. When a primitive component that uses a material is asked to collect mesh elements, the FMaterialRenderProxy corresponding to the UMaterialInterface it uses can be passed to FMeshBatchElement. The following takes StaticMesh as an example:

```
// Engine\Source\Runtime\Engine\Private\StaticMeshRender.cpp

bool FStaticMeshSceneProxy::GetMeshElement(
    int32    LODIndex,
    int32    BatchIndex,
    int32    SectionIndex,
    uint8    InDepthPriorityGroup,
    bool bUseSelectionOutline,
    bool bAllowPreCulledIndices,
    FMeshBatch& OutMeshBatch)const
{
    constERHIFeatureLevel::Type FeatureLevel = GetScene().GetFeatureLevel(); constFStaticMeshLODResources&
    LOD = RenderData->LODResources[LODIndex]; constFStaticMeshVertexFactories& VFs = RenderData-
    >LODVertexFactories[LODIndex]; constFStaticMeshSection& Section = LOD.Sections[SectionIndex]; const
    FLODInfo& ProxyLODInfo = LODs[LODIndex];


    //Get various instances of the material (includingUMaterialInterface, FMaterialRenderProxyandFMaterial)
    UMaterialInterface* MaterialInterface = ProxyLODInfo.Sections[SectionIndex].Material; FMaterialRenderProxy*
    MaterialRenderProxy = MaterialInterface->GetRenderProxy(); constFMaterial* Material = MaterialRenderProxy-
    >GetMaterial(FeatureLevel);
```

```
        FMeshBatchElement& OutMeshBatchElement = OutMeshBatch.Elements[0];

        //Handling Vertex Factories
        constFVertexFactory* VertexFactory = nullptr; if
        (ProxyLODInfo.OverrideColorVertexBuffer)
        {
            (......)
        }

        (......)

        if(NumPrimitives >0) {

            OutMeshBatch.SegmentIndex = SectionIndex;

            OutMeshBatch.LODIndex = LODIndex;

            //Assign materials and render delegates.
            OutMeshBatch.MaterialRenderProxy = MaterialRenderProxy;

            (......)
        }
}
```

Therefore, we can know that when the component collects the mesh elements, all types of material data are ready and can be accessed. This means that during the game thread phase, instances of various types of materials have been loaded, set, and created. Let's continue to dig deeper into when they are created. First, look at FMaterialRenderProxy. Different UMaterialInterface subclasses are slightly different, as shown in the following code:

```
// Engine\Source\Runtime\Engine\Private\Materials\MaterialInstance.cpp

void UMaterialInstance::PostInitProperties() {

        Super::PostInitProperties();

        if(!HasAnyFlags(RF_ClassDefaultObject)) {

            //createFMaterialRenderProxy.
            Resource = new FMaterialInstanceResource(this);
        }
}

FMaterialRenderProxy*UMaterialInstance::GetRenderProxy()const {

        returnResource;
}

// Engine\Source\Runtime\Engine\Private\Materials\Material.cpp

void UMaterial::PostInitProperties() {

        Super::PostInitProperties();
        if(!HasAnyFlags(RF_ClassDefaultObject))
```

```
    {
        //createFMaterialRenderProxy.
        DefaultMaterialInstance = new FDefaultMaterialInstance(this);
    }

    FPlatformMisc::CreateGuid(StateId);
}

FMaterialRenderProxy*UMaterial::GetRenderProxy()const {

    returnDefaultMaterialInstance;
}
```

It can be inferred that the FMaterialRenderProxy corresponding to UMaterialInstance is created in the PostInitProperties stage of the subclass.

We continue to find out which interface and member the corresponding FMaterial instance is obtained by UMaterialInterface:

```
// Engine\Source\Runtime\Engine\Private\Materials\Material.cpp

//GetUMaterialCorrespondingFMaterialResource(FMaterialInstance of a subclass of .
FMaterialResource*UMaterial::GetMaterialResource(ERHIFeatureLevel::Type                    InFeatureLevel,
EMaterialQualityLevel::Type QualityLevel)
{
    if(QualityLevel == EMaterialQualityLevel::Num) {

        QualityLevel = GetCachedScalabilityCVars().MaterialQualityLevel;
    }
    returnFindMaterialResource(MaterialResources, InFeatureLevel, QualityLevel,true);
}
```

From the above, we can know that UMaterial::MaterialResources is being searched, so let's continue to investigate when it is created:

```
FMaterialResource*        FindOrCreateMaterialResource(TArray<FMaterialResource*>&
MaterialResources,
    UMaterial* OwnerMaterial, UMaterialInstance*
    OwnerMaterialInstance, ERHIFeatureLevel::Type
    InFeatureLevel, EMaterialQualityLevel::Type
    InQualityLevel)
{
    (.....)

    FMaterialResource* CurrentResource = FindMaterialResource(MaterialResources, InFeatureLevel,
QualityLevelForResource,false);

    //  If the current resource list does not exist, create a new one.FMaterialResource
    if  Examples. (!CurrentResource)
    {
        //Prefer to use the material instance interface to create.
        CurrentResource = OwnerMaterialInstance? OwnerMaterialInstance-
> AllocatePermutationResource() : OwnerMaterial->AllocateResource(); CurrentResource-
        >SetMaterial(OwnerMaterial, OwnerMaterialInstance, QualityLevelForResource);          InFeatureLevel,
```

```
            //Add toFMaterialResourceList of instances.
            MaterialResources.Add(CurrentResource);
        }

        (……)

        returnCurrentResource;
}
```

When creating the FMaterialResource instance above, the valid OwnerMaterialInstance will be used first, and then the UMaterial interface will be used. Let's enter their interfaces to create FMaterialResource instances:

```
FMaterialResource*UMaterialInstance::AllocatePermutationResource() {

        returnnew FMaterialResource();
}

FMaterialResource*UMaterial::AllocateResource() {

        returnnew FMaterialResource();
}
```

Wow, the logic is the same, they all directly create a new FMaterialResource object and return it. Let's continue to track which interfaces will call FindOrCreateMaterialResource:

- ProcessSerializedInlineShaderMaps
- UMaterial::PostLoad
- UMaterial::CacheResourceShadersForRendering
- UMaterial::AllMaterialsCacheResourceShadersForRendering
- UMaterial::ForceRecompileForRendering
- UMaterial::PostEditChangePropertyInternal
- UMaterial::SetMaterialUsage
- UMaterial::UpdateMaterialShaders
- UMaterial::UpdateMaterialShaderCacheAndTextureReferences

The above interfaces will directly or indirectly call the FindOrCreateMaterialResource interface, thereby triggering the creation of the FMaterialResource object. However, in the runtime version, it is usually triggered by UMaterial::PostLoad, and the call stack is as follows:

- UMaterial::PostLoad
    - ProcessSerializedInlineShaderMaps
        - FindOrCreateMaterialResource

In addition, some interfaces of UMaterialInstance will also trigger the creation of FMaterialResource instances, which will not be tracked in this article.
We continue to study where and when FMaterial's GameThreadShaderMap and RenderingThreadShaderMap are set and passed:

```
//Direct settingsRenderingThreadShaderMap
void FMaterial::SetRenderingThreadShaderMap(const TRefCountPtr<FMaterialShaderMap>& InMaterialShaderMap)
{
	RenderingThreadShaderMap = InMaterialShaderMap;
}

//Setting up the game threadShaderMap.
void FMaterial::SetGameThreadShaderMap(FMaterialShaderMap* { InMaterialShaderMap)

	GameThreadShaderMap = InMaterialShaderMap;

	TRefCountPtr<FMaterialShaderMap> ShaderMap = GameThreadShaderMap;
	TRefCountPtr<FMaterial> Material = this;
	//Pushing settings to the rendering threadShaderMapInstructions.
	ENQUEUE_RENDER_COMMAND(SetGameThreadShaderMap)([Material = MoveTemp(Material), ShaderMap = MoveTemp(ShaderMap)](FRHICommandListImmediate& RHICmdList) mutable
	{
		Material->RenderingThreadShaderMap = MoveTemp(ShaderMap);
	});
}

//Setting InlineShaderMap
void FMaterial::SetInlineShaderMap(FMaterialShaderMap* { InMaterialShaderMap)

	GameThreadShaderMap = InMaterialShaderMap;
	bContainsInlineShaders = true;
	bLoadedCookedShaderMapId =true;

	TRefCountPtr<FMaterialShaderMap> ShaderMap = GameThreadShaderMap;
	TRefCountPtr<FMaterial> Material = this;
	//Pushing settings to the rendering threadShaderMapInstructions. ENQUEUE_RENDER_COMMAND(SetInlineShaderMap)([Material = MoveTemp(Material), ShaderMap = MoveTemp(ShaderMap)](FRHICommandListImmediate& RHICmdList) mutable
	{
		Material->RenderingThreadShaderMap = MoveTemp(ShaderMap);
	});
}
```

The above can set FMaterial's RenderingThreadShaderMap to have 3 interfaces, and continue to track which interfaces will call them:

- FMaterial::CacheShaders
    - FMaterial::SetGameThreadShaderMap
- FMaterialShaderMap::LoadForRemoteRecompile
    - FMaterial::SetGameThreadShaderMap
- ProcessSerializedInlineShaderMaps
    - FMaterial::SetInlineShaderMap
- SetShaderMapsOnMaterialResources_RenderThread

        - FMaterial::SetRenderingThreadShaderMap

Although many of the above interfaces will eventually be set to FMaterial's RenderingThreadShaderMap, in most cases, the call stack where the runtime RenderingThreadShaderMap is set as follows:

- UMaterial::PostLoad
    - ProcessSerializedInlineShaderMaps
        - FMaterial::SetInlineShaderMap

Once the RenderingThreadShaderMap of FMaterial is set correctly, many other data related to the material will be freely read by the rendering thread and renderer, just like a fish swimming carefree in the blue sea.

9.3.2 Material Compilation

In the previous article Shader system, the Shader compilation process has been explained, but this section describes the process and mechanism of how to convert material blueprints into HLSL code.

In the previous sections, we first understand the main types and concepts involved in the material blueprint compilation process.

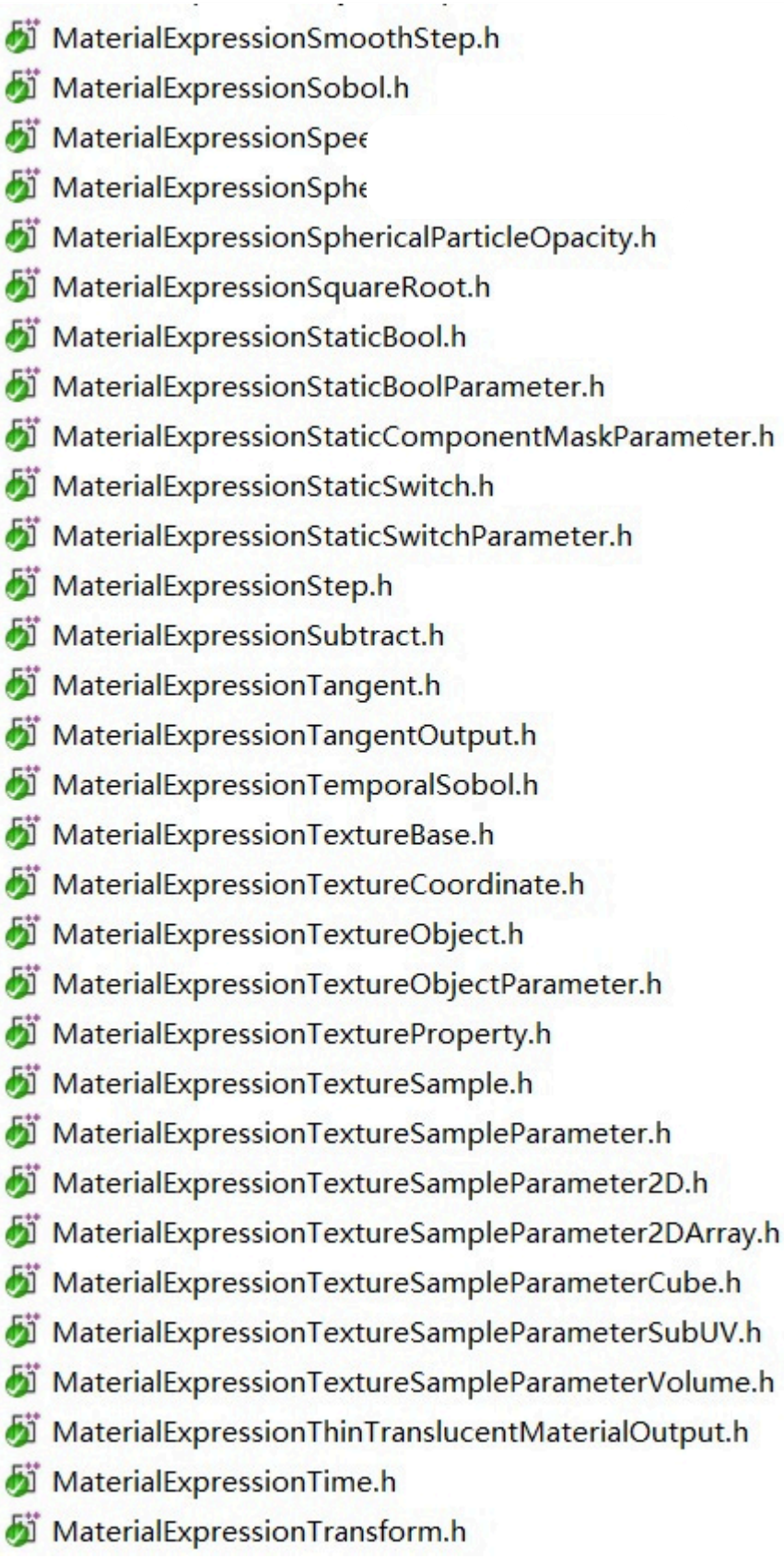
9.3.2.1 UMaterialExpression

UMaterialExpression is an expression. Each material node UMaterialGraphNode has a UMaterialExpression instance. Its main definition is as follows:

```
// Engine\Source\Runtime\Engine\Classes\Materials\MaterialExpression.h

classENGINE_API UMaterialExpression: public UObject {

#ifWITH_EDITORONLY_DATA
  int32          MaterialExpressionEditorX;
        int32 MaterialExpressionEditorY;
        //Material nodes.
        UEdGraphNode*         GraphNode;
#endif
        //    The material it belongs to.
        class     UMaterial*      Material;
        //The material function it belongs to.
        classUMaterialFunction*             Function;

        uint8 bIsParameterExpression:1;

        //Editor data and markup.
#if     WITH_EDITORONLY_DATA
        uint32     bShowInputs:1;
        TArray<FTextbShowOutputs:1;
        //Expression output.          MenuCategories;

        TArray<FExpressionOutput>          Outputs;
#endif
```

```
    //~ Begin UObject Interface.
    virtualvoid PostInitProperties() override;
    virtualvoid PostLoad() override;
    virtualvoid PostDuplicate(boolbDuplicateForPIE) override; virtualvoid Serialize
    (FStructuredArchive::FRecord Record) override; virtualbool IsEditorOnly()const {


        returntrue;
    }
    //~ End UObject Interface.

#if  WITH_EDITOR
    //Compile
    virtual int32Compile(class FMaterialCompiler* Compiler, int32 OutputIndex) {return INDEX_NONE; }


    virtual int32CompilePreview(class FMaterialCompiler* Compiler, int32 OutputIndex) { returnCompile(Compiler,
    OutputIndex); }
#endif

    //Data acquisition interface.
    virtualvoid GetTexturesForceMaterialRecompile(TArray<UTexture *> &Textures)const{ } virtual UObject*
    GetReferencedTexture()const{returnnullptr; } virtualbool CanReferenceTexture()const{returnfalse; }


#if  WITH_EDITOR
    //Get all input expressions.
    bool GetAllInputExpressions(TArray<UMaterialExpression*>&                    InputExpressions);

    //Parameter interface.
    virtualbool HasAParameterName()const{returnfalse; }
    virtualvoid ValidateParameterName(const boolbAllowDuplicateName =true); virtualbool
    HasClassAndNameCollision(UMaterialExpression* OtherExpression)const; virtualvoid
    SetValueToMatchingExpression(UMaterialExpression* OtherExpression) {}; virtual FNameGetParameterName()
    const{returnNAME_None; } virtualvoid SetParameterName(constFName& Name) {}


    (.....)
#endif// WITH_EDITOR };
```

There are a lot of subclasses that inherit from UMaterialExpression (about 200), because UE has many built-in material nodes. The following figure shows a small part of the subclasses:



The following two most commonly used subclasses are selected as cases for analysis:

```
// Engine\Source\Runtime\Engine\Classes\Materials\MaterialExpressionAdd.h

classUMaterialExpressionAdd: public UMaterialExpression {

    //Addition of two operands.
    FExpressionInput    A;
    FExpressionInput    B;
```

```
        //whenAandBAlternative value if
        illegal. floatConstA;
        float      ConstB;

        //~ Begin UMaterialExpression Interface
#if     WITH_EDITOR
        virtual int32Compile(class FMaterialCompiler* Compiler, int32 OutputIndex) override; virtualvoid GetCaption
        (TArray<FString>& OutCaptions)constoverride;
        virtual FTextGetKeywords()constoverride {returnFText::FromString(TEXT("+"));}
#endif// WITH_EDITOR
        //~ End UMaterialExpression Interface
};

// Engine\Source\Runtime\Engine\Private\Materials\MaterialExpressions.cpp

//Compile an expression.
int32UMaterialExpressionAdd::Compile(class FMaterialCompiler* Compiler, int32 OutputIndex)

{
        //Takes two operands.
        int32 Arg1 = A.GetTracedInput().Expression ? A.Compile(Compiler) : Compiler-
> Constant(ConstA);
        int32 Arg2 = B.GetTracedInput().Expression ? B.Compile(Compiler) : Compiler-
> Constant(ConstB);

        //Add and return the result.
        returnCompiler->Add(Arg1, Arg2);
}

//Get instructions.
void UMaterialExpressionAdd::GetCaption(TArray<FString>& OutCaptions)const {

        FString ret = TEXT("Add");

        FExpressionInput ATraced = A.GetTracedInput();
        FExpressionInput BTraced = B.GetTracedInput(); if(!
        ATraced.Expression || !BTraced.Expression) {

                ret += TEXT("(");
                ret += ATraced.Expression ? TEXT(",") : FString::Printf( TEXT("%.4g,"), ConstA); ret += BTraced.Expression ?
                TEXT(")") : FString::Printf( TEXT("%.4g)"), ConstB);
        }

        OutCaptions.Add(ret);
}

// Engine\Source\Runtime\Engine\Classes\Materials\MaterialExpressionDDX.h

classUMaterialExpressionDDX: public UMaterialExpression {

        FExpressionInput Value;

        //~ Begin UMaterialExpression Interface
#if     WITH_EDITOR
        virtual int32Compile(class FMaterialCompiler* Compiler, int32 OutputIndex) override; virtualvoid GetCaption
        (TArray<FString>& OutCaptions)constoverride;
#endif
```

```
	//~ End UMaterialExpression Interface
};

int32UMaterialExpressionDDX::Compile(class FMaterialCompiler* Compiler, int32 OutputIndex)

{
	int32 ValueInput = INDEX_NONE;

	if(Value.GetTracedInput().Expression) {

		ValueInput = Value.Compile(Compiler);
	}

	if(ValueInput == INDEX_NONE) {

		returnINDEX_NONE;
	}

	returnCompiler->DDX(ValueInput);
}

void UMaterialExpressionDDX::GetCaption(TArray<FString>& OutCaptions)const {

	OutCaptions.Add(FString(TEXT("DDX")));
}
```

The above two types call the compiler's Add and DDX when compiling. Now let's go into the implementation of these two interfaces of FMaterialCompiler (an abstract class implemented by the subclass FHLSLMaterialTranslator):

```
// Engine\Source\Runtime\Engine\Private\Materials\HLSLMaterialTranslator.cpp

int32FHLSLMaterialTranslator::Add(int32 A,int32 B) {

	if(A == INDEX_NONE || B == INDEX_NONE) {

		returnINDEX_NONE;
	}

	constuint64 Hash = CityHash128to64({ GetParameterHash(A), GetParameterHash(B) }); if
	(GetParameterUniformExpression(A) && GetParameterUniformExpression(B)) {

		returnAddUniformExpressionWithHash(Hash, new
FMaterialUniformExpressionFoldedMath(GetParameterUniformExpression(A),GetParameterUniformE
xpression(B),FMO_Add),GetArithmeticResultType(A,B),TEXT("(%s + %s)"),*GetParameterCode(A),*GetParameterCode(B));

	}
	else
	{
		returnAddCodeChunkWithHash(Hash, GetArithmeticResultType(A,B),TEXT("(%s + %s)"
		),*GetParameterCode(A),*GetParameterCode(B));
	}
}

int32FHLSLMaterialTranslator::DDX( int32 X ) {
```

```
    if(X == INDEX_NONE) {

        returnINDEX_NONE;
    }

    if(ShaderFrequency == SF_Compute)
    {
        // running a material in a compute shader pass (eg when using SVOGI) return
        AddInlinedCodeChunk(MCT_Float, TEXT("0"));
    }

    if(ShaderFrequency != SF_Pixel) {

        returnNonPixelShaderExpressionError();
    }

    returnAddCodeChunk(GetParameterType(X),TEXT("DDX(%s)"),*GetParameterCode(X));
}
```

Therefore, the compilation of material expressions is actually the serialization of parameters and corresponding functions into HLSL fragments.

9.3.2.2 UMaterialGraphNode

UMaterialGraphNode is the material node we created in the material editor. The inherited parent classes are UMaterialGraphNode_Base and UEDGraphNode. Their definitions are as follows:

```
// Engine\Source\Runtime\Engine\Classes\EdGraph\EdGraphNode.h

classENGINE_API UEdGraphNode: public UObject {

public:
    //Pinout
    TArray<UEdGraphPin*> Pins;

    int32    NodePosX;
    int32    NodePosY;
    int32    NodeWidth;
    int32    NodeHeight;

    TEnumAsByte<ENodeAdvancedPins::Type> AdvancedPinDisplay;

private:
    //Status and tags.
    ENodeEnabledState       EnabledState;
    ESaveOrphanPinMode      OrphanedPinSaveMode;
    uint8    bDisableOrphanPinSaving:1;
    uint8    bDisplayAsDisabled:1;
    uint8    bUserSetEnabledState:1;
    uint8 bIsIntermediateNode :1; uint8
    bHasCompilerMessage:1;

    //interface.
    virtualbool IsInDevelopmentMode()const; bool
        IsAutomaticallyPlacedGhostNode()const;
    void MakeAutomaticallyPlacedGhostNode();
```

```
    virtualvoid Serialize(FArchive& Ar) override;

    (.....)
};

// Engine\Source\Editor\UnrealEd\Classes\MaterialGraph\MaterialGraphNode_Base.h

classUMaterialGraphNode_Base: public UEdGraphNode {

    // Pininterface.
    virtualvoid CreateInputPins() {}; virtualvoid CreateOutputPins() {}; virtual
    bool IsRootNode()const{returnfalse;} class UEdGraphPin*GetInputPin
    (int32 InputIndex)const;

    UNREALED_APIvoid GetInputPins(TArray<class UEdGraphPin*>& OutInputPins)const; class UEdGraphPin*
    GetOutputPin(int32 OutputIndex)const;
    UNREALED_APIvoid GetOutputPins(TArray<class UEdGraphPin*>& OutOutputPins)const; UNREALED_APIvoid
    ReplaceNode(UMaterialGraphNode_Base* OldNode);

    //Input interface.
    virtual int32GetInputIndex(constUEdGraphPin* InputPin)const{return-1;} virtual uint32GetInputType(
    constUEdGraphPin* InputPin)const;
    void InsertNewNode(UEdGraphPin* FromPin, UEdGraphPin* NewLinkPin, TSet<UEdGraphNode*>& OutNodeList);


    //~ Begin UEdGraphNode Interface. virtualvoid
    AllocateDefaultPins() override; virtualvoid ReconstructNode
    () override;
    virtualvoid RemovePinAt(constint32 PinIndex,constEEdGraphPinDirection PinDirection) override;

    virtualvoid AutowireNewNode(UEdGraphPin* FromPin) override;
    virtualbool CanCreateUnderSpecifiedSchema(constUEdGraphSchema* Schema)const override;

    virtual FStringGetDocumentationLink()constoverride; //~ End
    UEdGraphNode Interface.

protected:
    void ModifyAndCopyPersistentPinData(UEdGraphPin& TargetPin,constUEdGraphPin& SourcePin)const;

};

// Engine\Source\Editor\UnrealEd\Classes\MaterialGraph\MaterialGraphNode.h

classUMaterialGraphNode: public UMaterialGraphNode_Base {

    //Material Expression.
    classUMaterialExpression*MaterialExpression;

    bool  bPreviewNeedsUpdate;
    bool  bIsErrorExpression;
    bool  bIsPreviewExpression;

    FRealtimeStateGetter RealtimeDelegate; FSetMaterialDirty
    MaterialDirtyDelegate; FSimpleDelegate
    InvalidatePreviewMaterialDelegate;

public:
    UNREALED_APIvoid PostCopyNode();
```

```
    UNREALED_API    FMaterialRenderProxy*GetExpressionPreview(); void
    UNREALED_API RecreateAndLinkNode();
    UNREALED_API int32GetOutputIndex(constUEdGraphPin* OutputPin); uint32
    GetOutputType(constUEdGraphPin* OutputPin);

    //~ Begin UObject Interface
    virtualvoid PostEditChangeProperty(FPropertyChangedEvent& PropertyChangedEvent) override;

    virtualvoid PostEditImport() override;
    virtualvoid PostDuplicate(boolbDuplicateForPIE) override; //~ End UObject
    Interface

    //~ Begin UMaterialGraphNode_Base Interface virtualvoid
    CreateInputPins() override; virtualvoid CreateOutputPins()
    override;
    virtual UNREALED_API int32GetInputIndex(constUEdGraphPin* InputPin)constoverride; virtual uint32GetInputType(
    constUEdGraphPin* InputPin)constoverride; //~ End UMaterialGraphNode_Base Interface



    (.....)
};
```

The material node contains the information of the graphical interface and the corresponding expression, adopting the classic design mode of separating view and data.

9.3.2.3 UMaterialGraph

UMaterialGraph is a member of UMaterial, which is used to store material nodes and parameters generated by the editor. It and related types are defined as follows:

```
// Engine\Source\Runtime\Engine\Classes\EdGraph\EdGraph.h

classENGINE_API UEdGraph: public UObject {

public:
    //Graphic style.
    TSubclassOf<classUEdGraphSchema> Schema; //
    Graphics node.
    TArray<classUEdGraphNode*> Nodes;

    uint32    bEditable:1;
    uint32    bAllowDeletion:1;
    uint32    bAllowRenaming:1;

    (.....)

public:
    FDelegateHandleAddOnGraphChangedHandler(constFOnGraphChanged::FDelegate& InHandler
) ;
    void RemoveOnGraphChangedHandler( FDelegateHandle Handle ); //~ Begin
    UObject interface
    virtualvoid BuildSubobjectMapping(UObject* OtherObject, TMap<UObject*, UObject*>& ObjectMapping)const
    override;

    //Node operations.
    template <typename NodeClass>
```

```
        NodeClass*CreateIntermediateNode();
        void AddNode( UEdGraphNode* NodeToAdd,boolbUserAction =false,boolbSelectNewNode = ) ;
true
        bool RemoveNode( UEdGraphNode* NodeToRemove,boolbBreakAllLinks=true);

        (......)

protected:
        //Create a node.
        UEdGraphNode*CreateNode(TSubclassOf<UEdGraphNode> NewNodeClass,boolbFromUI,bool bSelectNewNode );

        UEdGraphNode*CreateNode(TSubclassOf<UEdGraphNode> NewNodeClass,boolbSelectNewNode = true)

        UEdGraphNode*CreateUserInvokedNode(TSubclassOf<UEdGraphNode> NewNodeClass,bool
bSelectNewNode =true)

private:
        FOnGraphChanged OnGraphChanged;
};

// Engine\Source\Editor\UnrealEd\Classes\MaterialGraph\MaterialGraph.h

classUNREALED_API UMaterialGraph: public UEdGraph {

        //The corresponding material instance.
        classUMaterial*                         Material;
        //Material functions.
        classUMaterialFunction* //Root
        node.                                   MaterialFunction;
        classUMaterialGraphNode_Root*
                                                    RootNode;

        //List of material inputs.
        TArray<FMaterialInputInfo> MaterialInputs;

        //Delegation.
        FRealtimeStateGetter RealtimeDelegate; FSetMaterialDirty
        MaterialDirtyDelegate; FToggleExpressionCollapsed
        ToggleCollapsedDelegate;

public:
        //Rebuild the material graph.
        void RebuildGraph();

        //Adds an expression to the material graph.
        class UMaterialGraphNode*               AddExpression(UMaterialExpression*          Expression,
bool bUserInvoked);
        classUMaterialGraphNode_Comment* bool   AddComment(UMaterialExpressionComment*      Comment,
bIsUserInvoked =false);

        //Connect all nodes.
        void   LinkGraphNodesFromMaterial();
        void   LinkMaterialExpressionsFromGraph()       const;

        (......)
};
```

9.3.2.4 FHLSLMaterialTranslator

FHLSLMaterialTranslator inherits from FMaterialCompiler and its function is to translate the material expression into HLSL code and fill it into the macros and empty code segments of MaterialTemplate.ush. Their definitions are as follows:

```
// Engine\Source\Runtime\Engine\Public\MaterialCompiler.h

classFMaterialCompiler {

public:
	virtual ~FMaterialCompiler() { } //Material
	property interface.
	virtualvoid SetMaterialProperty(EMaterialProperty InProperty, EShaderFrequency OverrideShaderFrequency =
SF_NumFrequencies,boolbUsePreviousFrameTime =false) =0;
	virtualvoid PushMaterialAttribute(constFGuid& InAttributeID) =0; virtual FGuid
	PopMaterialAttribute() =0; virtualconstFGuidGetMaterialAttribute() =0;

	virtualvoid SetBaseMaterialAttribute(constFGuid& InAttributeID) =0; virtualvoid PushParameterOwner(
	constFMaterialParameterInfo& InOwnerInfo) =0; virtual FMaterialParameterInfoPopParameterOwner() =0;



	//Calls a material expression.
	virtual int32CallExpression(FMaterialExpressionKey ExpressionKey,FMaterialCompiler* InCompiler) =0;



	//Platform and shading model dependent.
	virtual EShaderFrequencyGetCurrentShaderFrequency()const=0;
	virtualEMaterialCompilerTypeGetCompilerType()const; inlinebool
	IsVertexInterpolatorBypass()const; virtual EMaterialValueTypeGetType(int32 Code) =0;
	virtual EMaterialQualityLevel::TypeGetQualityLevel() =0; virtual
	ERHIFeatureLevel::TypeGetFeatureLevel() =0; virtual EShaderPlatform
	GetShaderPlatform() =0;

	virtualconstITargetPlatform*GetTargetPlatform()const=0; virtual FMaterialShadingModelField
	GetMaterialShadingModels()const=0;

	(.....)

	//The interface corresponding to the material expression.
	virtual int32AccessCollectionParameter(UMaterialParameterCollection* ParameterCollection,
int32 ParameterIndex, int32 ComponentIndex) =0;
	virtual int32ScalarParameter(FName ParameterName,floatDefaultValue) =0; virtual int32VectorParameter(FName
	ParameterName,constFLinearColor& DefaultValue) =
0;
	virtual int32Constant(floatX) =0; virtual int32Constant2(floatX,
	floatY) =0; virtual int32Sine(int32 X) =0; virtual int32Cosine
	(int32 X) =0; virtual int32Tangent(int32 X) =0; virtual int32
	ReflectionVector() =0;



	virtual int32If(int32 A,int32 B,int32 AGreaterThanB,int32 AEqualsB,int32 ALessThanB,int32
Threshold) =0;
	virtual int32VertexInterpolator(uint32 InterpolatorIndex) =0;

	virtual int32Add(int32 A,int32 B) =0; virtual int32Sub
	(int32 A,int32 B) =0;
```

```
        virtual int32Mul(int32 A,int32 B) =0; virtual int32Div
        (int32 A,int32 B) =0; virtual int32Dot(int32 A,int32 B) =0;
        virtual int32Cross(int32 A,int32 B) =0;


        virtual int32DDX(int32 X) =0; virtual int32
        DDY(int32 X) =0;

        (......)
};

// Engine\Source\Runtime\Engine\Private\Materials\HLSLMaterialTranslator.h

classFHLSLMaterialTranslator: public FMaterialCompiler {

protected:
        //Compiled materials.
        FMaterial* Material;
        //The compiled output will be stored inDDC.
        FMaterialCompilationOutput& MaterialCompilationOutput;

        //The resource string.
        FString ResourcesString;
        // MaterialTemplate.usfThe string content.
        FString MaterialTemplate;

        //Platform dependent.
        EShaderFrequency       ShaderFrequency;
        EShaderPlatform        Platform;
        EMaterialQualityLevel::Type ERHIFeatuQreuLaelivtyeLl:e:Tvyepl;e FeatureLevel;
        FMaterialShadingModelField ShadingModelsFromCompilation; const
        ITargetPlatform* TargetPlatform;


        //Compiled intermediate data.
        EMaterialProperty MaterialProperty; TArray<FGuid> MaterialAttributesStack;
        TArray<FMaterialParameterInfo> ParameterOwnerStack; TArray<FShaderCodeChunk>*
        CurrentScopeChunks; boolSharedPixelProperties[CompiledMP_MAX];
        TArray<FMaterialFunctionCompileState*> FunctionStacks[SF_NumFrequencies];
        FStaticParameterSet StaticParameters;



        TArray<FShaderCodeChunk> TArraSyh<aFrSehdaPdroeprCeortdyeCCohduenCkh>u nks[SF_NumFrequencies];
        TArray<TRefCountPtr<FMaterialUnUifnoifromrEmxEpxrpersessiosnio>n s;
        TArray<TRefCountPtr<FMaterialUniformExpression>                >UniformVectorExpressions;
        TArray<TRefCountPtr<FMaterialUniformExpressionTexture>         >UniformScalarExpressions;
        UniformTextureExpressions[NumMaterialTextureParameterTypes];          >

        TArray<TRefCountPtr<FMaterialUniformExpressionExternalTexture>>
        UniformExternalTextureExpressions;

        TArray<UMaterialParameterCollection*>                ParameterCollections;
        TArray<FMaterialCustomExpressionEntry> TArray<FSCtruinsgto>m Expressions;
        CustomOutputImplementations;
        TArray<UMaterialExpressionVertexInterpolator*>                  CustomVertexInterpolators;

        //Vertex Factory stack entry.
```

```
    TArray<FMaterialVTStackEntry> VTStacks;
    FHashTable VTStackHash;

    TBitArray<>     AllocatedUserTexCoords;
    TBitArray<>     AllocatedUserVertexTexCoords;

    (.....)

public:
    //implementHLSLTranslate.
    bool Translate();
    //Get the material environment.
    void GetMaterialEnvironment(EShaderPlatform InPlatform, FShaderCompilerEnvironment& OutEnvironment);


    void GetSharedInputsMaterialCode(FString& PixelMembersDeclaration, FString& NormalAssignment,
FString& PixelMembersInitializationEpilog);
    //Get the material shader code.
    FStringGetMaterialShaderCode();

protected:
    //Get all definitions.
    FStringGetDefinitions(TArray<FShaderCodeChunk>& CodeChunks, int32 StartChunk, int32 EndChunk)const;


    //Code blocks.
    int32AddCodeChunkInner(uint64 Hash,constTCHAR* FormattedCode, EMaterialValueType Type,boolbInlined);

    int32AddCodeChunk(EMaterialValueType Type,constTCHAR* Format, ...); int32AddCodeChunkWithHash(uint64
    BaseHash, EMaterialValueType Type,constTCHAR* Format, ...);

    int32AddInlinedCodeChunk(EMaterialValueType Type,constTCHAR* Format, ...); int32
    AddInlinedCodeChunkWithHash(uint64 BaseHash, EMaterialValueType Type,const TCHAR* Format, ...);


    int32AddUniformExpressionInner(uint64 Hash, FMaterialUniformExpression* UniformExpression,
EMaterialValueType Type,constTCHAR* FormattedCode);
    int32AddUniformExpression(FMaterialUniformExpression* UniformExpression, EMaterialValueType
Type,constTCHAR* Format, ...);
    int32AddUniformExpressionWithHash(uint64 BaseHash, FMaterialUniformExpression* UniformExpression,
EMaterialValueType Type,constTCHAR* Format, ...);

    //Material Expression.
    virtual int32Sine(int32 X) override; virtual int32Cosine(int32 X)
    override; virtual int32Tangent(int32 X) override; virtual int32
    Arcsine(int32 X) override; virtual int32ArcsineFast(int32 X)
    override; virtual int32Arccosine(int32 X) override; virtual int32
    Floor(int32 X) override; virtual int32Ceil(int32 X) override;
    virtual int32Round(int32 X) override; virtual int32Truncate
    (int32 X) override; virtual int32Sign(int32 X) override; virtual
    int32Frac(int32 X) override; virtual int32Fmod(int32 A, int32 B)
    override;
```

```
    (.....)
};
```

FHLSLMaterialTranslator implements all abstract interfaces of FMaterialCompiler. Its core members and interfaces are as follows:

- FMaterial* Material: The target material to compile.
- FMaterialCompilationOutput & MaterialCompilationOutput: The compiled result. FString
- MaterialTemplate: The MaterialTemplate.ush string to be filled or after being filled. Translate():
- Perform HLSL translation, translate the expression into a code block and save it in the corresponding attribute slot.
- GetMaterialShaderCode(): Fills the material's macros, properties, expressions and other data into MaterialTemplate.ush and returns the result.

There is a section later that specifically explains the translation process of FHLSLMaterialTranslator.

In addition, FMaterialCompiler also has a subclass FProxyMaterialCompiler, which is used for Lightmass renderer and material baking.

9.3.2.5 MaterialTemplate.ush

MaterialTemplate.usf is a material shader template, which contains a lot of %s vacancies and macros to be replaced, which are filled by FHLSLMaterialTranslator::GetMaterialShaderCode. Some of its original codes are as follows:

```
// Engine\Shaders\Private\MaterialTemplate.ush

#include "/Engine/Private/SceneTexturesCommon.ush" "/Engine/
#include Private/EyeAdaptationCommon.ush" "/Engine/Private/
#include Random.ush"
#include "/Engine/Private/SobolRandom.ush" "/Engine/Private/
#include MonteCarlo.ush" "/Engine/Generated/UniformBuffers/
#include Material.ush" "/Engine/Private/DepthOfFieldCommon.ush" "/
#include Engine/Private/CircleDOFCommon. ush"
#include
#include "/Engine/Private/GlobalDistanceFieldShared.ush" "/Engine/
#include Private/SceneData.ush"
#include "/Engine/Private/HairShadingCommon.ush"

////////////////////////////////////////////////// ////////////////////////// //! Must match ESceneTextureId


//Post-processing attribute macros.
#define PPI_SceneColor 0
#define PPI_SceneDepth 1
#define PPI_DiffuseColor 2
#define PPI_SpecularColor 3
#define PPI_SubsurfaceColor 4
#define PPI_BaseColor 5
#define PPI_Specular 6
#define PPI_Metallic 7
#define PPI_WorldNormal 8
#define PPI_SeparateTranslucency 9
```

```
#define PPI_Opacity 10
#define PPI_Roughness 11
#define PPI_MaterialAO 12
#define PPI_CustomDepth 13
#define
#define ////////////////////////////////////////////////////////////////////////////////
#define
(.....)
//Macro definitions to be filled.
#define NUM_MATERIAL_TEXCOORDS_VERTEX %s
#define NUM_MATERIAL_TEXCOORDS %s
#define NUM_CUSTOM_VERTEX_INTERPOLATORS %s
#define NUM_TEX_COORD_INTERPOLATORS %s

// Vertex interpolation position definitions.
%s

//File references and macro definitions.

#include"/Engine/Private/PaniniProjection.ush"

#ifndef USE_DITHERED_LOD_TRANSITION
    #if USE_INSTANCING
        #ifndefUSE_DITHERED_LOD_TRANSITION_FOR_INSTANCED
            #error"USE_DITHERED_LOD_TRANSITION_FOR_INSTANCED should have been defined"
        #endif
        #define USE_DITHERED_LOD_TRANSITION USE_DITHERED_LOD_TRANSITION_FOR_INSTANCED
    #else
        #ifndef USE_DITHERED_LOD_TRANSITION_FROM_MATERIAL
            #error"USE_DITHERED_LOD_TRANSITION_FROM_MATERIAL should have been defined"
        #endif
        #defineUSE_DITHERED_LOD_TRANSITION USE_DITHERED_LOD_TRANSITION_FROM_MATERIAL
    #endif
#endif

#ifndefUSE_STENCIL_LOD_DITHER
    #defineUSE_STENCIL_LOD_DITHER USE_STENCIL_LOD_DITHER_DEFAULT
#endif

#defineMATERIALBLENDING_ANY_TRANSLUCENT (MATERIALBLENDING_TRANSLUCENT ||
MATERIALBLENDING_ADDITIVE || MATERIALBLENDING_MODULATE)

(.....)

//Various structures of materials.
structFMaterialAttributes {

%s
};

structFPixelMaterialInputs {

%s
};

//Pixel parameters.
structFMaterialPixelParameters {
```

```
#if NUM_TEX_COORD_INTERPOLATORS
    float2 TexCoords[NUM_TEX_COORD_INTERPOLATORS];
#endif

    half4  VertexColor;
    half3 WorldNormal;
    half3 WorldTangent;
    half3  ReflectionVector;
    half3 CameraVector;
    half3  LightVector;
    float4   SvPosition;
    float4   ScreenPosition;
    half UnMirrored;
    half TwoSidedSign;
    half3x3   TangentToWorld;

#if USE_WORLDVERTEXNORMAL_CENTER_INTERPOLATION
    half3 WorldVertexNormal_Center;
#endif

    float3   AbsoluteWorldPosition;
    float3   WorldPosition_CamRelative;
    float3   WorldPosition_NoOffsets;
    float3   WorldPosition_NoOffsets_CamRelative;
    half3    LightingPositionOffset;
    float    AOMaterialMask;

#if LIGHTMAP_UV_ACCESS
    float2 LightmapUVs;
#endif

#if USE_INSTANCING
    half4 PerInstanceParams;
#endif

    uint PrimitiveId;

#if TEX_COORD_SCALE_ANALYSIS FTexCoordScalesParams
    TexCoordScalesParams;
#endif

    (.....)
};

//Vertex parameters.
structFMaterialVertexParameters {

    float3   WorldPosition;
    half3x3   TangentToWorld;
#ifUSE_INSTANCING
    float4x4    InstanceLocalToWorld;
    float3   InstanceLocalPosition;
    float4   PerInstanceParams;
    uint InstanceId;
    uint InstanceOffset;

#elifIS_MESHPARTICLE_FACTORY float4x4
  InstanceLocalToWorld;
```

```
#endif
        float4x4 PrevFrameLocalToWorld;

        float3      PreSkinnedPosition;
        float3      PreSkinnedNormal;

#if    GPU_SKINNED_MESH_FACTORY
        float3      PreSkinOffset;
        float3      PostSkinOffset;
#endif

        half4VertexColor;
#if    NUM_MATERIAL_TEXCOORDS_VERTEX
        float2 TexCoords[NUM_MATERIAL_TEXCOORDS_VERTEX];
        #ifES3_1_PROFILE
        float2      TexCoordOffset;
        #endif
#endif

        FMaterialParticleParameters Particle; uint
        PrimitiveId;

        (......)
};

//Data operation interface.
MaterialFloat3x3        GetLocalToWorld3x3(uint         PrimitiveId);
MaterialFloat3x3        GetLocalToWorld3x3();
float3      GetObjectWorldPosition(FMaterialPixelParameters                  Parameters);
float3      GetObjectWorldPosition(FMaterialTessellationParameters                   Parameters);
(......)

//Material expression interface.
float MaterialExpressionDepthOfFieldFunction(floatSceneDepth,intFunctionValueIndex); float2
            MaterialExpressionGetAtlasUVs(FMaterialPixelParameters Parameters);
float4      MaterialExpressionGetHairAuxilaryData(FMaterialPixelParameters Parameters);
float3MaterialExpressionGetHairColorFromMelanin(floatMelanin,floatRedness, float3 DyeColor);

(......)

//Material property lookup.
MaterialFloat4          ProcessMaterialColorTextureLookup(MaterialFloat4 TextureValue);
MaterialFloat4          ProcessMaterialVirtualColorTextureLookup(MaterialFloat4 TextureValue);
MaterialFloat4          ProcessMaterialExternalTextureLookup(MaterialFloat4 TextureValue);
MaterialFloat4          ProcessMaterialLinearColorTextureLookup(MaterialFloat4 TextureValue);
MaterialFloat       ProcessMaterialGreyscaleTextureLookup(MaterialFloat TextureValue);
(......)

//    Unity material expressions.
%s

//Material property acquisition interface.
half3GetMaterialNormalRaw(FPixelMaterialInputs PixelMaterialInputs);
half3GetMaterialNormal(FMaterialPixelParameters Parameters, FPixelMaterialInputs PixelMaterialInputs);

half3    GetMaterialTangentRaw(FPixelMaterialInputs PixelMaterialInputs); GetMaterialTangent
half3    (FPixelMaterialInputs PixelMaterialInputs); GetMaterialEmissiveRaw
half3    (FPixelMaterialInputs PixelMaterialInputs);
```

```
half3   GetMaterialEmissive(FPixelMaterialInputs PixelMaterialInputs);
half3   GetMaterialEmissiveForCS(FMaterialPixelParameters Parameters);
{ %s;
}
uint
half3 GetMaterialShadingModel(FPixelMaterialInputs PixelMaterialInputs);
half3   GetMaterialBaseColorRaw(FPixelMaterialInputs PixelMaterialInputs);
half    GetMaterialBaseColor(FPixelMaterialInputs PixelMaterialInputs);
{ %s; GetMaterialCustomData0(FMaterialPixelParameters                Parameters)
}
half
{ %s;
}       GetMaterialCustomData1(FMaterialPixelParameters                Parameters)
half
half
half2
(......)  GetMaterialAmbientOcclusionRaw(FPixelMaterialInputs GetMaterialAmPbixienlMtOactcelruiaslilonnputs);
          (FPixelMaterialInputs PixelMaterialInputs);
            GetMaterialRefraction(FPixelMaterialInputs PixelMaterialInputs);


//Calculate material parameters interface.
void    CalcMaterialParametersEx(
        in out FMaterialPixelParameters Parameters, in out
        FPixelMaterialInputs PixelMaterialInputs, float4
                    SvPosition,
        float4      ScreenPosition,
        FIsFrontFace        bIsFrontFace,
        float3      TranslatedWorldPosition,
        float3      TranslatedWorldPositionExcludingShaderOffsets);
void    CalcMaterialParameters(
        in out FMaterialPixelParameters Parameters, in out
        FPixelMaterialInputs PixelMaterialInputs, float4 SvPosition,

        FIsFrontFace bIsFrontFace);
void    CalcMaterialParametersPost(
        in out FMaterialPixelParameters Parameters, in out
        FPixelMaterialInputs PixelMaterialInputs, float4 SvPosition,

        FIsFrontFace bIsFrontFace);
float ApplyPixelDepthOffsetToMaterialParameters(inout                FMaterialPixelParameters
MaterialParameters, FPixelMaterialInputs PixelMaterialInputs, outfloatOutDepth);

(......)
```

As can be seen above, MaterialTemplate.ush contains a lot of data and interfaces, mainly in several categories:

- Base shader module reference.
- Macro definition to be filled.
- The interface implementation to be populated.
- Structure definitions for vertices, pixels, material properties, etc. Some structures need to be filled.

- Material properties, data processing, expressions, and tool interface definitions. Some interfaces are yet to be filled.

9.3.2.6 Material compilation process

The above sections analyze in detail the core types involved in material compilation. This section will directly analyze the process of compiling material blueprints into HLSL code.

The compilation entry of the material ShaderMap is in the following two interfaces of FMaterial:

- FMaterial::BeginCompileShaderMap
- FMaterial::GetMaterialExpressionSource

However, BeginCompileShaderMap is in the main process and has stronger applicability. Let's take it as a starting point to analyze the compilation process of the material blueprint:

```
// Engine\Source\Runtime\Engine\Private\Materials\MaterialShared.cpp

bool FMaterial::BeginCompileShaderMap(constFMaterialShaderMapId& ShaderMapId,const FStaticParameterSet &StaticParameterSet,
  EShaderPlatform Platform, TRefCountPtr<FMaterialShaderMap>& OutShaderMap,const ITargetPlatform* TargetPlatform) {

      //Note that this is only executed while in the editor.
#ifWITH_EDITORONLY_DATA bool
      bSuccess =false; //Newshader
      map.
      TRefCountPtr<FMaterialShaderMap> NewShaderMap = new FMaterialShaderMap();

#if   WITH_EDITOR
      NewShaderMap->AssociateWithAsset(GetAssetPath());
#endif

      //   Generate MaterialsshaderCode.
      Output the results.
      //
      FMaterialCompilationOutput //        NewCompilationOutput;
      Converter.
      FHLSLMaterialTranslatorMaterialTranslator(this,                    NewCompilationOutput, StaticParameterSet, Platform, GetQualityLevel(), ShaderMapId.FeatureLevel, TargetPlatform);
      //Perform expression conversion, fill inMaterialTemplate.ush.
      bSuccess = MaterialTranslator.Translate();

      //Subsequent operations need to be performed only if the expression conversion is successful.
      if(bSuccess)
      {
            //Create a shader compilation environment for the material. All compilation jobs will share this material. TRefCountPtr<FShaderCompilerEnvironment> MaterialEnvironment = new FShaderCompilerEnvironment();
            MaterialEnvironment->TargetPlatform = TargetPlatform; //Get the material environment.
            MaterialTranslator.GetMaterialEnvironment(Platform, *MaterialEnvironment); //Get Materialshader Code.
            constFString MaterialShaderCode = MaterialTranslator.GetMaterialShaderCode();

            const boolbSynchronousCompile = RequiresSynchronousCompilation() ||
```

```
!GShaderCompilingManager->AllowAsynchronousShaderCompiling();

            //Contains virtual texture file paths.
            MaterialEnvironment-
> IncludeVirtualPathToContentsMap.Add(TEXT("/Engine/Generated/Material.ush"),
MaterialShaderCode);

            //Compiling MaterialsshaderCode.
            NewShaderMap->Compile(this, ShaderMapId, MaterialEnvironment,
NewCompilationOutput, Platform, bSynchronousCompile);

            if(bSynchronousCompile)//Synchronous compilation {

                //Synchronous mode, directly assign toOutShaderMap.
                OutShaderMap = NewShaderMap->CompiledSuccessfully() ? NewShaderMap : nullptr;
            }
            else//Asynchronous compilation
            {
                //FirstNewShaderMapPut it in the list waiting for compilation to end.
                OutstandingCompileShaderMapIds.AddUnique( NewShaderMap->GetCompilingId() ); //Asynchronous mode,
                OutShaderMapFirst set tonull,Will fall back to the default material. OutShaderMap = nullptr;

            }
        }

        returnbSuccess;
#else
        UE_LOG(LogMaterial,         Fatal,TEXT("Not         supported."));
        returnfalse;
#endif
}
```

The above interface is divided into several steps: initialize data and create a new compilation object, execute material blueprint translation, if the translation is successful, create a material shader compilation environment, and then compile the translated material blueprint shader code. Finally, it will be divided into different returns based on whether it is asynchronous. If it is synchronous, it will be returned directly. If it is asynchronous, it will be put into the list OutstandingCompileShaderMapIds waiting for the compilation to end.

The logical stack chain for handling OutstandingCompileShaderMapIds is as follows:

- UMaterial::Serialize
    - SerializeInlineShaderMaps
        - FMaterial::SerializeInlineShaderMap
            - FMaterial::FinishCompilation
                - FMaterial::GetShaderMapIDsWithUnfinishedCompilation

The code of FMaterial::GetShaderMapIDsWithUnfinishedCompilation is as follows:

```
void FMaterial::GetShaderMapIDsWithUnfinishedCompilation(TArray<int32>& ShaderMapIds) {

    //Add uncompiledShaderMapIdTo the list. (First detectGameThreadShaderMap,Retest
    OutstandingCompileShaderMapIds)
```

```
    if(GameThreadShaderMap && !GameThreadShaderMap->IsCompilationFinalized()) {

        ShaderMapIds.Add(GameThreadShaderMap->GetCompilingId());
    }
    else if(OutstandingCompileShaderMapIds.Num() !=0) {

        ShaderMapIds.Append(OutstandingCompileShaderMapIds);
    }
}
```

FMaterial::BeginCompileShaderMap also has several important interfaces of FHLSLMaterialTranslator that are not resolved. Let's cover them all:

```
// Engine\Source\Runtime\Engine\Private\Materials\HLSLMaterialTranslator.cpp

bool FHLSLMaterialTranslator::Translate() {

    bSuccess =true;

    //The compiled output needs to be saved toMaterialCompilationOutput,It can be automatically saved toDDC.

    Material->CompileErrors.Empty(); Material->ErrorExpressions.Empty();
    bCompileForComputeShader = Material->IsLightFunction(); int32
    NormalCodeChunkEnd =-1; int32 Chunk[CompiledMP_MAX];



    memset(Chunk, INDEX_NONE,sizeof(Chunk));

    //Translate all custom vertex interpolators before accessing primary attributes so that type information is available. {

        CustomVertexInterpolators.Empty();
        CurrentCustomVertexInterpolatorOffset               =0;
        NextVertexInterpolatorIndex             =0;
        MaterialProperty        = MP_MAX;
        ShaderFrequency     = SF_Vertex;

        TArray<UMaterialExpression*> Expressions; Material-
        >GatherExpressionsForCustomInterpolators(Expressions);
        GatherCustomVertexInterpolators(Expressions);

        //Resets the shared stack data.
        while(FunctionStacks[SF_Vertex].Num() >1) {

            FMaterialFunctionCompileState* Stack = FunctionStacks[SF_Vertex].Pop(false); delete Stack;

        }
        FunctionStacks[SF_Vertex][0]->Reset();

        //When expression lists are available, node count limits are applied.
        int32 NumMaterialLayersAttributes =0;
        for(UMaterialExpression* Expression : Expressions) {

            if(UMaterialExpressionMaterialAttributeLayers* Layers =
Cast<UMaterialExpressionMaterialAttributeLayers>(Expression))
            {
```

```
                    + + NumMaterialLayersAttributes;

                    if(NumMaterialLayersAttributes >1) {

                        Errorf(TEXT("Materials can contain only one Material Attribute Layers
node."));
                        break;
                    }
                }
            }
        }

        constEShaderFrequency NormalShaderFrequency =
        FMaterialAttributeDefinitionMap::GetShaderFrequency(MP_Normal);
        constEMaterialDomain Domain = Material->GetMaterialDomain(); const
        EBlendMode BlendMode = Material->GetBlendMode();

        //Collects implementations of any custom output expressions.
        TArray<UMaterialExpressionCustomOutput*> CustomOutputExpressions; Material-
        >GatherCustomOutputExpressions(CustomOutputExpressions); TSet<UClass*>
        SeenCustomOutputExpressionsClasses;

        //Some custom outputs must be pre-compiled so that they can be reused as shared inputs.
        CompileCustomOutputs(CustomOutputExpressions, SeenCustomOutputExpressionsClasses, true);


        //Normals are compiled first.
        {
            Chunk[MP_Normal]        =  Material->CompilePropertyAndSetMaterialProperty(MP_Normal,
this);
            NormalCodeChunkEnd        = SharedPropertyCodeChunks[NormalShaderFrequency].Num();
        }

        (.....)

        //The remaining material properties.
        Chunk[MP_EmissiveColor]                                    =Material-
> CompilePropertyAndSetMaterialProperty(MP_EmissiveColor                    , this);
Chunk[MP_DiffuseColor]                                  =Material-
> CompilePropertyAndSetMaterialProperty(MP_DiffuseColor                    , this);
Chunk[MP_SpecularColor]                                    =Material-
> CompilePropertyAndSetMaterialProperty(MP_SpecularColor                    , this);
Chunk[MP_BaseColor]                                    =Material-
> CompilePropertyAndSetMaterialProperty(MP_BaseColor                    , this);
Chunk[MP_Metallic]                                  =Material-
> CompilePropertyAndSetMaterialProperty(MP_Metallic                    , this);
Chunk[MP_Specular]                                  =Material-
> CompilePropertyAndSetMaterialProperty(MP_Specular                    , this);
Chunk[MP_Roughness]                                    =Material-
> CompilePropertyAndSetMaterialProperty(MP_Roughness                    , this);
Chunk[MP_Anisotropy]                                =Material-
> CompilePropertyAndSetMaterialProperty(MP_Anisotropy                    , this);
Chunk[MP_Opacity]                                  =Material-
> CompilePropertyAndSetMaterialProperty(MP_Opacity                    , this);
Chunk[MP_OpacityMask]                                =Material-
> CompilePropertyAndSetMaterialProperty(MP_OpacityMask                    , this);
Chunk[MP_Tangent]                                  =Material-
> CompilePropertyAndSetMaterialProperty(MP_Tangent                    , this);
```

```
        Chunk[MP_WorldPositionOffset]                   =Material-
> CompilePropertyAndSetMaterialProperty(MP_WorldPositionOffset                    , this);
Chunk[MP_WorldDisplacement]                    =Material-
> CompilePropertyAndSetMaterialProperty(MP_WorldDisplacement                     , this);
Chunk[MP_TessellationMultiplier]                =Material-
> CompilePropertyAndSetMaterialProperty(MP_TessellationMultiplier                     , this);

        //deal withshading model.
        Chunk[MP_ShadingModel]                    =Material-
> CompilePropertyAndSetMaterialProperty(MP_ShadingModel FMaterialShadingModelFiel,d t his);
        MaterialShadingModels = Material->GetShadingModels(); if(Material-
        >IsShadingModelFromMaterialExpression() && ShadingModelsFromCompilation.IsValid())

        {
                MaterialShadingModels = ShadingModelsFromCompilation;
        }
        ValidateShadingModelsForFeatureLevel(MaterialShadingModels);

        (……)

        //Self-defined data.
        Chunk[MP_CustomData0]                    =Material-
> CompilePropertyAndSetMaterialProperty(MP_CustomData0                    , this);
Chunk[MP_CustomData1]                    =Material-
> CompilePropertyAndSetMaterialProperty(MP_CustomData1                    , this);
Chunk[MP_AmbientOcclusion]                    =Material-
> CompilePropertyAndSetMaterialProperty(MP_AmbientOcclusion if                     , this);
        (IsTranslucentBlendMode(BlendMode) ||
        MaterialShadingModels.HasShadingModel(MSM_SingleLayerWater))
        {
                int32 UserRefraction = ForceCast(Material-
> CompilePropertyAndSetMaterialProperty(MP_Refraction, this), int32                MCT_Float1);
                RefractionDepthBias =
ForceCast(ScalarParameter(FName(TEXT("RefractionDepthBias")),                    Material-
> GetRefractionDepthBiasValue()), MCT_Float1);

                Chunk[MP_Refraction] = AppendVector(UserRefraction, RefractionDepthBias);
        }

        (……)

        ResourcesString         =  TEXT("");

        (……)

        //Code block generation completed.
        bAllowCodeChunkGeneration =false;

        //Handles various flags for compilation and materials.
        bUsesEmissiveColor = IsMaterialPropertyUsed(MP_EmissiveColor, Chunk[MP_EmissiveColor], FLinearColor(0,0,0,0),3);

        bUsesPixelDepthOffset = (AllowPixelDepthOffset(Platform) && IsMaterialPropertyUsed(MP_PixelDepthOffset,
Chunk[MP_PixelDepthOffset], FLinearColor(0,0, 0,0),1))

                || (Domain == MD_DeferredDecal && Material->GetDecalBlendMode() ==
                DBM_Volumetric_DistanceFunction);
        boolbUsesWorldPositionOffsetCurrent = IsMaterialPropertyUsed(MP_WorldPositionOffset,
Chunk[MP_WorldPositionOffset], FLinearColor(0,0,0,0),3);
```

```
        boolbUsesWorldPositionOffsetPrevious = IsMaterialPropertyUsed(MP_WorldPositionOffset,
Chunk[CompiledMP_PrevWorldPositionOffset], FLinearColor(0,0,0,0),3);
        bUsesWorldPositionOffset = bUsesWorldPositionOffsetCurrent ||
        bUsesWorldPositionOffsetPrevious;
        MaterialCompilationOutput.bModifiesMeshPosition = bUsesPixelDepthOffset ||
        bUsesWorldPositionOffset;
        MaterialCompilationOutput.bUsesWorldPositionOffset MaterialComp=il abtUiosneOsWutopruldt.PboUssietisoPniOxefflDsetp; thO
        bIsFullyRough = Chunk[MP_Roughness] != INDEX_NONE && IsMateriablPUrsoepsePrixtyeUlDsepdt(MhOPf_fRsoetu;ghness,
        Chunk[MP_Roughness], FLinearColor(1,0,0,0),1) == false;



        bUsesAnisotropy = IsMaterialPropertyUsed(MP_Anisotropy, Chunk[MP_Anisotropy], FLinearColor(0,0,0,0
),1);
        MaterialCompilationOutput.bUsesAnisotropy = bUsesAnisotropy; if
        (bUsesSceneDepth)
        {
                MaterialCompilationOutput.SetIsSceneTextureUsed(PPI_SceneDepth);
        }
        MaterialCompilationOutput.bUsesDistanceCullFade = bUsesDistanceCullFade;

        (......)

        boolbDBufferAllowed = IsUsingDBuffers(Platform);
        boolbDBufferBlendMode = IsDBufferDecalBlendMode((EDecalBlendMode)Material-
> GetDecalBlendMode());

        FString InterpolatorsOffsetsDefinitionCode; TBitArray<>
        FinalAllocatedCoords =
GetVertexInterpolatorsOffsets(InterpolatorsOffsetsDefinitionCode);

        MaterialCompilationOutput.NumUsedUVScalars = GetNumUserTexCoords() *2;
        MaterialCompilationOutput.NumUsedCustomInterpolatorScalars =
        CurrentCustomVertexInterpolatorOffset;


        //Let's process the normals code block first.
        {
                GetFixedParameterCode(
                        0,
                        NormalCodeChunkEnd,
                        Chunk[MP_Normal],
                        SharedPropertyCodeChunks[NormalShaderFrequency],
                        TranslatedCodeChunkDefinitions[MP_Normal],
                        TranslatedCodeChunks[MP_Normal]);


                if(TranslatedCodeChunkDefinitions[MP_Normal].IsEmpty()) {

                        TranslatedCodeChunkDefinitions[MP_Normal] =
GetDefinitions(SharedPropertyCodeChunks[NormalShaderFrequency],0,                NormalCodeChunkEnd);
                }
        }


        //Normals are ignored for the rest of the properties block.
        for(uint32 PropertyId =0; PropertyId < MP_MAX; ++PropertyId) {

                if(PropertyId == MP_MaterialAttributes || PropertyId == MP_Normal || PropertyId MP_CustomOutput)
==
                {
                        continue;
```

```
        }

        constEShaderFrequency PropertyShaderFrequency =
        FMaterialAttributeDefinitionMap::GetShaderFrequency((EMaterialProperty)PropertyId);

        int32 StartChunk =0;
        if(PropertyShaderFrequency == NormalShaderFrequency &&
        SharedPixelProperties[PropertyId])
        {
            StartChunk = NormalCodeChunkEnd;
        }

        GetFixedParameterCode(
            StartChunk,
            SharedPropertyCodeChunks[PropertyShaderFrequency].Num(),
            Chunk[PropertyId],
            SharedPropertyCodeChunks[PropertyShaderFrequency],
            TranslatedCodeChunkDefinitions[PropertyId],
            TranslatedCodeChunks[PropertyId]);
    }

    //Manipulate material properties.
    for(uint32 PropertyId = MP_MAX; PropertyId < CompiledMP_MAX; ++PropertyId) {

        switch(PropertyId)
        {
            caseCompiledMP_EmissiveColorCS:
                if(bCompileForComputeShader) {

                    GetFixedParameterCode(Chunk[PropertyId],
SharedPropertyCodeChunks[SF_Compute], TranslatedCodeChunkDefinitions[PropertyId],
TranslatedCodeChunks[PropertyId]);
                }
                break;
            caseCompiledMP_PrevWorldPositionOffset:
                {
                    GetFixedParameterCode(Chunk[PropertyId],
SharedPropertyCodeChunks[SF_Vertex], TranslatedCodeChunkDefinitions[PropertyId],
TranslatedCodeChunks[PropertyId]);
                }
                break;
            default: check(0);
                break;
        }
    }

    //Outputs the implementation of any custom output expressions.
    for(int32 ExpressionIndex =0; ExpressionIndex < CustomOutputImplementations.Num(); ExpressionIndex++)

    {
        ResourcesString += CustomOutputImplementations[ExpressionIndex] +"\r\n\r\n";
    }

    // UniformScalar expressions.
    for(constFMaterialUniformExpression* ScalarExpression : UniformScalarExpressions) {

        FMaterialUniformPreshaderHeader& Preshader =
        MaterialCompilationOutput.UniformExpressionSet.UniformScalarPreshaders.AddDefaulted_GetRef
```

```
();
            Preshader.OpcodeOffset =
            MaterialCompilationOutput.UniformExpressionSet.UniformPreshaderData.Num();
            ScalarExpression -
> WriteNumberOpcodes(MaterialCompilationOutput.UniformExpressionSet.UniformPreshaderData);
            Preshader.OpcodeSize =
MaterialCompilationOutput.UniformExpressionSet.UniformPreshaderData.Num() -
Preshader.OpcodeOffset;
        }

        // UniformVector expression.
        for(FMaterialUniformExpression* VectorExpression : UniformVectorExpressions) {

            FMaterialUniformPreshaderHeader& Preshader =
MaterialCompilationOutput.UniformExpressionSet.UniformVectorPreshaders.AddDefaulted_GetRef ();

            Preshader.OpcodeOffset =
            MaterialCompilationOutput.UniformExpressionSet.UniformPreshaderData.Num();
            VectorExpression -
> WriteNumberOpcodes(MaterialCompilationOutput.UniformExpressionSet.UniformPreshaderData);
            Preshader.OpcodeSize =
MaterialCompilationOutput.UniformExpressionSet.UniformPreshaderData.Num() -
Preshader.OpcodeOffset;
        }

        // UniformScalar expressions.
        for(uint32 TypeIndex =0u; TypeIndex < NumMaterialTextureParameterTypes; ++TypeIndex) {


MaterialCompilationOutput.UniformExpressionSet.UniformTextureParameters[TypeIndex].Empty(U
niformTextureExpressions[TypeIndex].Num());
            for(FMaterialUniformExpressionTexture* TextureExpression :
            UniformTextureExpressions[TypeIndex])
            {
                TextureExpression -
> GetTextureParameterInfo(MaterialCompilationOutput.UniformExpressionSet.UniformTexturePara
meters[TypeIndex].AddDefaulted_GetRef());
            }
        }

        //External texture.


MaterialCompilationOutput.UniformExpressionSet.UniformExternalTextureParameters.Empty(UniformExternalTextureExpressions.Num
        for(FMaterialUniformExpressionExternalTexture* TextureExpression :
        UniformExternalTextureExpressions)
        {
            TextureExpression -
> GetExternalTextureParameterInfo(MaterialCompilationOutput.UniformExpressionSet.UniformExt
ernalTextureParameters.AddDefaulted_GetRef());
        }

        //loadMaterialTemplate.ushThe code of the file. LoadShaderSourceFileChecked(TEXT("/Engine/
        Private/MaterialTemplate.ush"), GetShaderPlatform(), MaterialTemplate);


        //turn up'#line'The string position of .
        constint32 LineIndex = MaterialTemplate.Find(TEXT("#line"),
```

```
ESearchCase::CaseSensitive);

	//calculate'#line'The end of the line before the
	statement. MaterialTemplateLineNumber =
	INDEX_NONE; int32 StartPosition = LineIndex +1; do

	{
		MaterialTemplateLineNumber++;
		StartPosition = MaterialTemplate.Find(TEXT("\n"), ESearchCase::CaseSensitive, ESearchDir::FromEnd,
StartPosition -1);
	}
	while(StartPosition != INDEX_NONE);

	MaterialTemplateLineNumber +=3;

	//Material parameter sets.

MaterialCompilationOutput.UniformExpressionSet.SetParameterCollections(ParameterCollection s);

	//Create a material uniform buffer structure.
	MaterialCompilationOutput.UniformExpressionSet.CreateBufferStruct(); //Storage UnificationVTThe
	number of samples. MaterialCompilationOutput.EstimatedNumVirtualTextureLookups =
	NumVtSamples;

	//Clean up all function stacks.
	ClearAllFunctionStacks();

	returnbSuccess;
}

//Get the material's environment (macro definition).
void FHLSLMaterialTranslator::GetMaterialEnvironment(EShaderPlatform                    InPlatform,
FShaderCompilerEnvironment& OutEnvironment)
{
	if(bNeedsParticlePosition || Material->ShouldGenerateSphericalParticleNormals() ||
	bUsesSphericalParticleOpacity)
	{
		OutEnvironment.SetDefine(TEXT("NEEDS_PARTICLE_POSITION"),1);
	}

	if(bNeedsParticleVelocity || Material->IsUsedWithNiagaraMeshParticles()) {

		OutEnvironment.SetDefine(TEXT("NEEDS_PARTICLE_VELOCITY"),1);
	}

	(.....)

	OutEnvironment.SetDefine(TEXT("MATERIAL_ENABLE_TRANSLUCENCY_FOGGING"), Material-
> ShouldApplyFogging());
	OutEnvironment.SetDefine(TEXT("MATERIAL_ENABLE_TRANSLUCENCY_CLOUD_FOGGING"), Material-
> ShouldApplyCloudFogging());
	OutEnvironment.SetDefine(TEXT("MATERIAL_IS_SKY"), Material->IsSky());
	OutEnvironment.SetDefine(TEXT("MATERIAL_COMPUTE_FOG_PER_PIXEL"), Material-
> ComputeFogPerPixel());
	OutEnvironment.SetDefine(TEXT("MATERIAL_FULLY_ROUGH"), bIsFullyRough || Material-
> IsFullyRough());
	OutEnvironment.SetDefine(TEXT("MATERIAL_USES_ANISOTROPY"), bUsesAnisotropy);
```

```
(……)

//Material parameters.
for(int32 CollectionIndex =0; CollectionIndex < ParameterCollections.Num(); CollectionIndex++)

{
	constFString CollectionName = FString::Printf(TEXT("MaterialCollection%u"), CollectionIndex);

	FShaderUniformBufferParameter::ModifyCompilationEnvironment(*CollectionName,
ParameterCollections[CollectionIndex]->GetUniformBufferStruct(), InPlatform, OutEnvironment);

}
OutEnvironment.SetDefine(TEXT("IS_MATERIAL_SHADER"), TEXT("1"));

//deal withshading model. FMaterialShadingModelField ShadingModels = Material-
>GetShadingModels(); if(Material- >IsShadingModelFromMaterialExpression() &&
ShadingModelsFromCompilation.IsValid())

{
	ShadingModels = ShadingModelsFromCompilation;
}
if(ShadingModels.IsLit()) {

	intNumSetMaterials =0;
	if(ShadingModels.HasShadingModel(MSM_DefaultLit)) {

		OutEnvironment.SetDefine(TEXT("MATERIAL_SHADINGMODEL_DEFAULT_LIT"),
TEXT("1"));
		NumSetMaterials++;
	}
	if (ShadingModels.HasShadingModel(MSM_Subsurface))
	{
		OutEnvironment.SetDefine(TEXT("MATERIAL_SHADINGMODEL_SUBSURFACE"), TEXT("1"));
		NumSetMaterials++;
	}
	if (ShadingModels.HasShadingModel(MSM_PreintegratedSkin))
	{
		OutEnvironment.SetDefine(TEXT("MATERIAL_SHADINGMODEL_PREINTEGRATED_SKIN"),
TEXT("1"));
		NumSetMaterials++;
	}
	if(ShadingModels.HasShadingModel(MSM_SubsurfaceProfile)) {

		OutEnvironment.SetDefine(TEXT("MATERIAL_SHADINGMODEL_SUBSURFACE_PROFILE"),
TEXT("1"));
		NumSetMaterials++;
	}

	(……)
}
else
{
	OutEnvironment.SetDefine(TEXT("MATERIAL_SINGLE_SHADINGMODEL"), TEXT("1"));
	OutEnvironment.SetDefine(TEXT("MATERIAL_SHADINGMODEL_UNLIT"), TEXT("1"));
}

(……)
}
```

```
//Get the translated material expressionshader        (fillingMaterialTemplatebring%s)
Code. FStringFHLSLMaterialTranslator::GetMaterialShaderCode()
{
        //    Delayed formattingMaterialTemplateString.
        FLazyPrintf        LazyPrintf(*MaterialTemplate);

        //Assign slots to vertex interpolators.
        FString VertexInterpolatorsOffsetsDefinition; TBitArray<>
        FinalAllocatedCoords =
GetVertexInterpolatorsOffsets(VertexInterpolatorsOffsetsDefinition);

        constuint32 NumUserVertexTexCoords = GetNumUserVertexTexCoords(); constuint32
        NumUserTexCoords = GetNumUserTexCoords(); constuint32 NumCustomVectors =
        FMath::DivideAndRoundUp((uint32)CurrentCustomVertexInterpolatorOffset,
                                                                                2u);
        constuint32 NumTexCoordVectors = FinalAllocatedCoords.FindLast(true) +1;

        LazyPrintf.PushParam(*FString::Printf(TEXT("%u"),NumUserVertexTexCoords));
        LazyPrintf.PushParam(*FString::Printf(TEXT("%u"),NumUserTexCoords));
        LazyPrintf.PushParam(*FString::Printf(TEXT("%u"),NumCustomVectors));
        LazyPrintf.PushParam(*FString::Printf(TEXT("%u"),NumTexCoordVectors));

        LazyPrintf.PushParam(*VertexInterpolatorsOffsetsDefinition);

        FString MaterialAttributesDeclaration;

        //Serialize the material's floating point type intoHLSLtype.
        constTArray<FGuid>& OrderedVisibleAttributes =
        FMaterialAttributeDefinitionMap::GetOrderedVisibleAttributeList();
        for(constFGuid& AttributeID : OrderedVisibleAttributes) {

                constFString PropertyName =
                FMaterialAttributeDefinitionMap::GetAttributeName(AttributeID);
                constEMaterialValueType PropertyType =
                FMaterialAttributeDefinitionMap::GetValueType(AttributeID);
                switch(PropertyType)
                {
                caseMCT_Float1:caseMCT_Float: MaterialAttributesDeclaration += FString::Printf(TEXT(
"\tfloat %s;") LINE_TERMINATOR, *PropertyName);break;
                caseMCT_Float2: MaterialAttributesDeclaration += FString::Printf(TEXT("\tfloat2 %s;") LINE_TERMINATOR,
*PropertyName);break;
                caseMCT_Float3: MaterialAttributesDeclaration += FString::Printf(TEXT("\tfloat3 %s;") LINE_TERMINATOR,
*PropertyName);break;
                caseMCT_Float4: MaterialAttributesDeclaration += FString::Printf(TEXT("\tfloat4 %s;") LINE_TERMINATOR,
*PropertyName);break;
                caseMCT_ShadingModel: MaterialAttributesDeclaration += FString::Printf(TEXT("\tuint %s;"
) LINE_TERMINATOR, *PropertyName);break;
                }
        }
        LazyPrintf.PushParam(*MaterialAttributesDeclaration);

        FString      PixelMembersDeclaration;
        FString      NormalAssignment;
        FString      PixelMembersSetupAndAssignments;
        //Get the shared input material code.
        GetSharedInputsMaterialCode(PixelMembersDeclaration,                NormalAssignment,
        PixelMembersSetupAndAssignments);
```

```
	LazyPrintf.PushParam(*PixelMembersDeclaration);
	LazyPrintf.PushParam(*ResourcesString);

	if(bCompileForComputeShader) {

		LazyPrintf.PushParam(*GenerateFunctionCode(CompiledMP_EmissiveColorCS));
	}
	else
	{
		LazyPrintf.PushParam(TEXT("return 0"));
	}

	LazyPrintf.PushParam(*FString::Printf(TEXT("return 		%.5f"), 	Material-
> GetTranslucencyDirectionalLightingIntensity()));
	LazyPrintf.PushParam(*FString::Printf(TEXT("return 		%.5f"), 	Material-
> GetTranslucentShadowDensityScale()));
	LazyPrintf.PushParam(*FString::Printf(TEXT("return 		%.5f"), 	Material-
> GetTranslucentSelfShadowDensityScale()));
	LazyPrintf.PushParam(*FString::Printf(TEXT("return 		%.5f"), 	Material-
> GetTranslucentSelfShadowSecondDensityScale()));
	LazyPrintf.PushParam(*FString::Printf(TEXT("return 		%.5f"), 	Material-
> GetTranslucentSelfShadowSecondOpacity()));
	LazyPrintf.PushParam(*FString::Printf(TEXT("return 		%.5f"), 	Material-
> GetTranslucentBackscatteringExponent()));

	{
		FLinearColor Extinction = Material->GetTranslucentMultipleScatteringExtinction();
		LazyPrintf.PushParam(*FString::Printf(TEXT("return MaterialFloat3(%.5f, %.5f, %.5f)"), Extinction.R,
Extinction.G, Extinction.B));
	}

	LazyPrintf.PushParam(*FString::Printf(TEXT("return %.5f"), Material-
> GetOpacityMaskClipValue())); LazyPrintf.PushParam(*GenerateFunctionCode(MP_WorldPositionOffset));
	LazyPrintf.PushParam(*GenerateFunctionCode(CompiledMP_PrevWorldPositionOffset));
	am(*FString::Printf( TEXT("return %.5f"), Material-


> GetMaxDisplacement())); LazyPrintf.PushParam(*GenerateFunctionCode(MP_TessellationMultiplier));
	LazyPrintf.PushParam(*GenerateFunctionCode(MP_CustomData0));
	LazyPrintf.PushParam(*GenerateFunctionCode(MP_CustomData1));


	//Fill in custom texture coordinate allocations.
	FString CustomUVAssignments; int32
	LastProperty =-1;
	for(uint32 CustomUVIndex =0; CustomUVIndex < NumUserTexCoords; CustomUVIndex++) {

		if(CustomUVIndex ==0) {

			CustomUVAssignments += TranslatedCodeChunkDefinitions[MP_CustomizedUVs0 +
CustomUVIndex];
		}

		if(TranslatedCodeChunkDefinitions[MP_CustomizedUVs0 + CustomUVIndex].Len() >0) {

			if(LastProperty >=0) {

				check(TranslatedCodeChunkDefinitions[LastProperty].Len() ==
```

```
TranslatedCodeChunkDefinitions[MP_CustomizedUVs0 + CustomUVIndex].Len());
                }
                LastProperty = MP_CustomizedUVs0 + CustomUVIndex;
            }
            CustomUVAssignments += FString::Printf(TEXT("\tOutTexCoords[%u] = %s;") LINE_TERMINATOR,
CustomUVIndex, *TranslatedCodeChunks[MP_CustomizedUVs0 + CustomUVIndex]);
        }
        LazyPrintf.PushParam(*CustomUVAssignments);

        //Populate custom vertex shader interpolator assignments.
        FString CustomInterpolatorAssignments;
        for(UMaterialExpressionVertexInterpolator* Interpolator : CustomVertexInterpolators) {

            if(Interpolator->InterpolatorOffset != INDEX_NONE) {

                constEMaterialValueType Type = Interpolator->InterpolatedType == MCT_Float?
MCT_Float1 : Interpolator->InterpolatedType;
                constTCHAR* Swizzle[2] = { TEXT("x"), TEXT("y") }; constint32 Offset =
                Interpolator->InterpolatorOffset; constint32 Index = Interpolator-
                >InterpolatorIndex;

                CustomInterpolatorAssignments +=
FString::Printf(TEXT("\tOutTexCoords[VERTEX_INTERPOLATOR_%i_TEXCOORDS_X].%s =
VertexInterpolator%i(Parameters).x;") LINE_TERMINATOR, Index, Swizzle[Offset%2], Index);

                if(Type >= MCT_Float2) {

                    (.....)
                }
            }
        }
        LazyPrintf.PushParam(*CustomInterpolatorAssignments);

        // NormalRequired Initializers
        LazyPrintf.PushParam(*TranslatedCodeChunkDefinitions[MP_Normal]);
        LazyPrintf.PushParam(*NormalAssignment);
        //Finally, the remaining common code is followed by the assignment of values to each input.
        LazyPrintf.PushParam(*PixelMembersSetupAndAssignments); LazyPrintf.PushParam(*FString::Printf(TEXT("%u"
        ),MaterialTemplateLineNumber));

        returnLazyPrintf.GetResultString();
}
```

Now let's briefly summarize several important interfaces of FHLSLMaterialTranslator:

- **Translate**:Translate the material node expression of the material blueprint and fill the compiled results of all material properties into the FShaderCodeChunk of the grid.
- **GetMaterialEnvironment**:Handles the compilation environment of the material blueprint (macro definition).
- **GetMaterialShaderCode**:Fills in the missing code of MaterialTemplate.ush according to the material attribute interface corresponding to the FShaderCodeChunk compiled by Translate, as well as other macro definitions, structures, and tool class interfaces.

After being compiled by FHLSLMaterialTranslator, the complete material shader code will be obtained and sent to the FMaterialShaderMap::Compile interface for compilation. The compiled shader code is saved in FMaterialShaderMap.

The only interface for calling FMaterial::BeginCompileShaderMap is FMaterial::CacheShaders. The core call stack for calling FMaterial::CacheShaders is as follows:

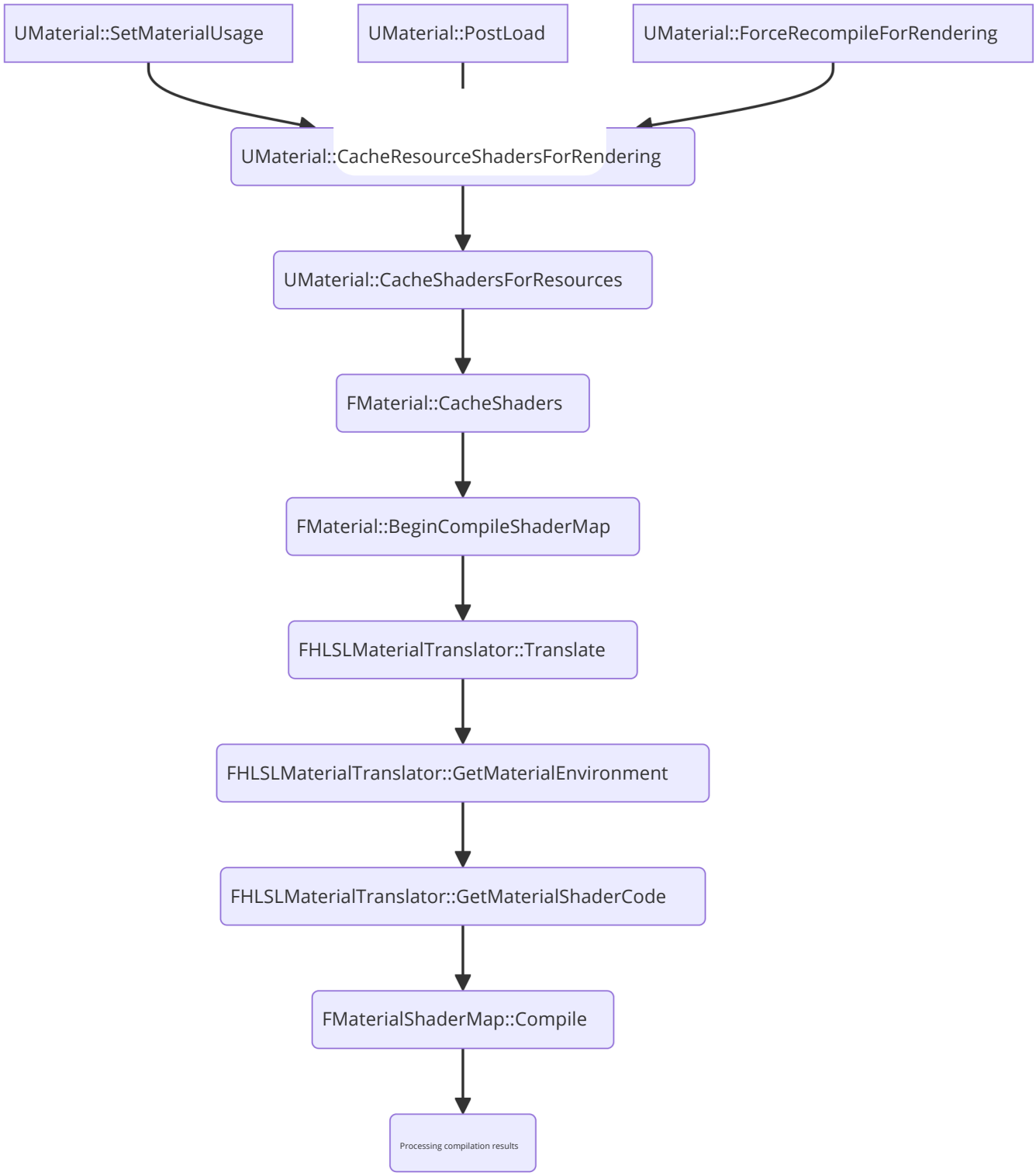
- UMaterial::CacheResourceShadersForRendering
    - UMaterial::CacheShadersForResources
        - FMaterial::CacheShaders
- UMaterialInstance::CacheResourceShadersForRendering
    - UMaterialInstance::CacheShadersForResources
        - FMaterial::CacheShaders

The CacheResourceShadersForRendering call logic is very large, including various setting interfaces of UMaterial and UMaterialInstance:

- SetMaterialUsage
- UpdateMaterialShaderCacheAndTextureReferences
- PostLoad
- PostEditChangePropertyInternal
- ForceRecompileForRendering
- AllMaterialsCacheResourceShadersForRendering

In the above interfaces, PostLoad must be called.

At this point, the entire process of material compilation is clear, and the flowchart is as follows (taking UMaterial as an example, the same is true for UMaterialInstance):



In fact, when compiling different types of shaders, the data required is not exactly the same:

type	composition
GlobalShader	Shader_x.usf
MaterialShader	Shader_x.usf + MaterialTemplate_x.usf
MeshMaterialShader	Shader_x.usf + MaterialTemplate_x.usf + VertexFactory_x.usf

in:

- Shader_x.usf: Existing files in the engine Shader directory, such as DeferredLightVertexShaders.usf and DeferredLightPixelShaders.usf. MaterialTemplate_x.usf: Code that fills in MaterialTemplate.ush
- after FHLSLMaterialTranslator
  compiles the material blueprint.
- VertexFactory_x.usf: Existing vertex factory file codes in the engine Shader directory, such as LocalVertexFactory.ush and GpuSkinVertexFactory.ush.

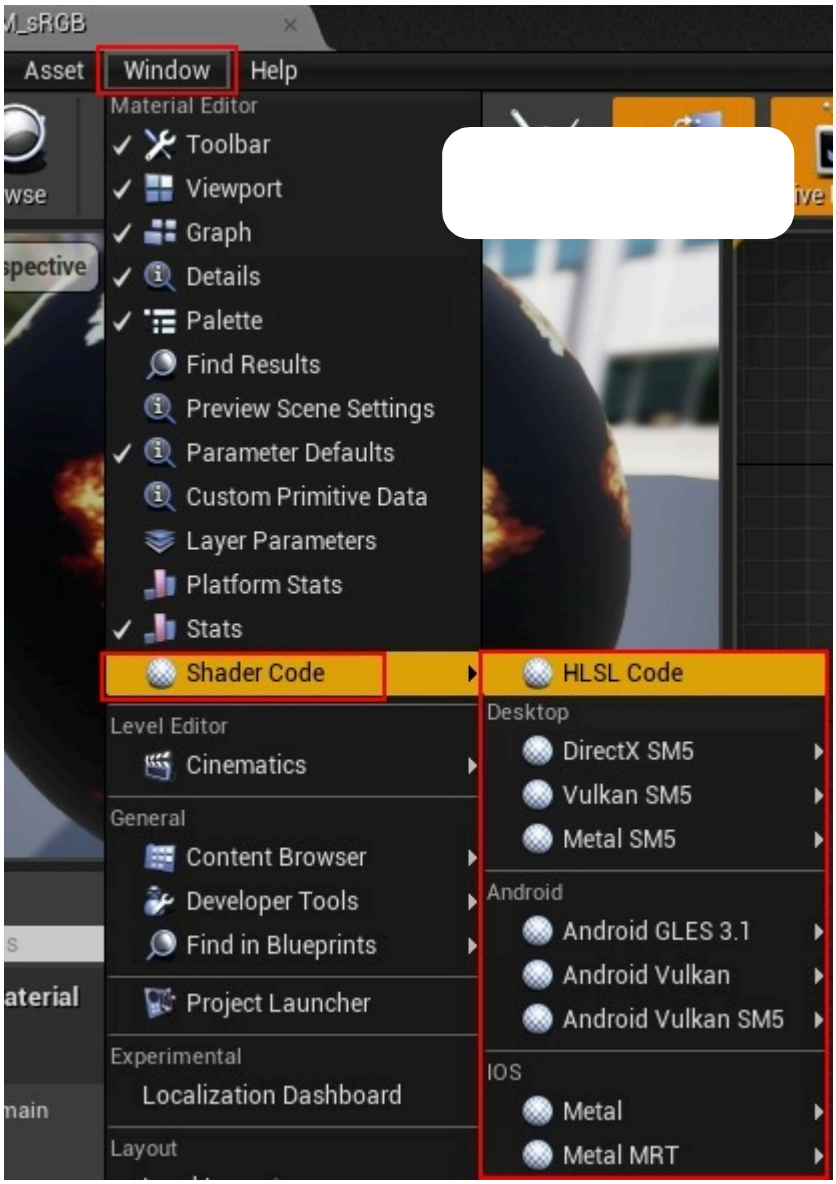
In addition, the data required to generate shader codes with different shading frequencies for the same material blueprint is also different:

Coloring frequency	**composition**
PixelShader	MaterialTemplate_x.usf + VertexFactory_x.usf + PixelShader_x.usf
VertexShader	MaterialTemplate_x.usf + VertexFactory_x.usf + VertexShader_x.usf
GeometryShader	MaterialTemplate_x.usf + VertexFactory_x.usf + VertexShader_x.usf

in:

- PixelShader_x.usf: PixelShader code of a rendering Pass, such as BasePassPixelShader.usf, ShadowDepthPixelShader.usf, DepthOnlyPixelShader.usf.
- VertexShader_x.usf: VertexShader code of a rendering pass, such as BasePassVertexShader.usf, ShadowDepthVertexShader.usf, DepthOnlyVertexShader.usf.

In addition, you can view the target platform code after filling the MaterialTemplate in the material editor:



The following figure shows the code preview interface after opening HLSL:

HLSL Code

Copy

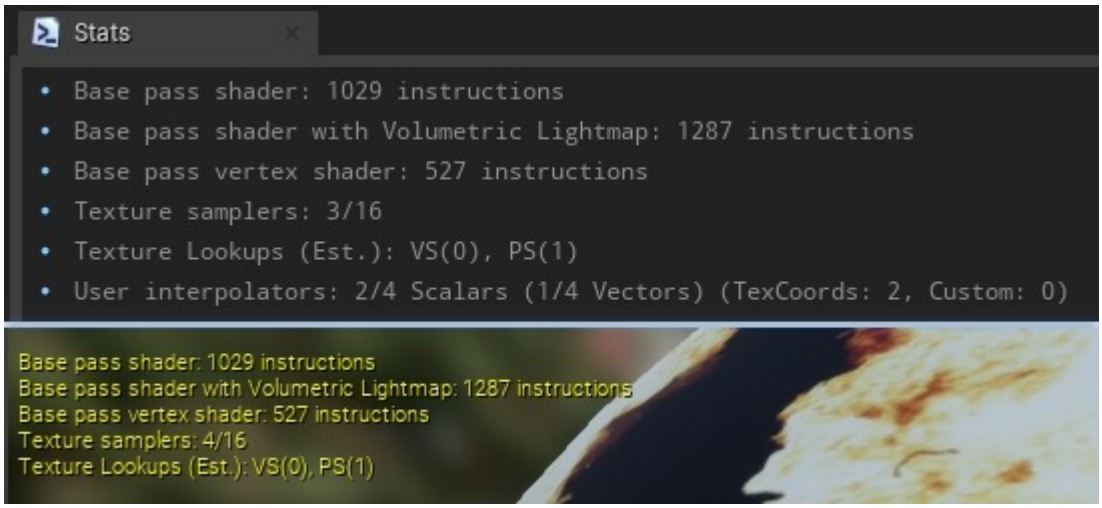
```
// Copyright Epic Games, Inc. All Rights Reserved.

/**
 * MaterialTemplate.usf: Filled in by FHL                                    alShaderCode for each material being compiled.
 */

#include "/Engine/Private/SceneTexturesCommon.ush"
#include "/Engine/Private/EyeAdaptationCommon.ush"
#include "/Engine/Private/Random.ush"
#include "/Engine/Private/SobolRandom.ush"
#include "/Engine/Private/MonteCarlo.ush"
#include "/Engine/Generated/UniformBuffers/Material.ush"
#include "/Engine/Private/DepthOfFieldCommon.ush"
#include "/Engine/Private/CircleDOFCommon.ush"
#include "/Engine/Private/GlobalDistanceFieldShared.ush"
#include "/Engine/Private/SceneData.ush"
#include "/Engine/Private/HairShadingCommon.ush"

#if USES_SPEEDTREE
    #include "/Engine/Private/SpeedTreeCommon.ush"
#endif

////////////////////////////////////////////////////////////////////////////////
//! Must match ESceneTextureId

#define PPI_SceneColor 0
#define PPI_SceneDepth 1
#define PPI_DiffuseColor 2
#define PPI_SpecularColor 3
#define PPI_SubsurfaceColor 4
#define PPI_BaseColor 5
#define PPI_Specular 6
#define PPI_Metallic 7
#define PPI_WorldNormal 8
#define PPI_SeparateTranslucency 9
#define PPI_Opacity 10
#define PPI_Roughness 11
#define PPI_MaterialAO 12
#define PPI_CustomDepth 13
#define PPI_PostProcessInput0 14
#define PPI_PostProcessInput1 15
#define PPI_PostProcessInput2 16
#define PPI_PostProcessInput3 17
#define PPI_PostProcessInput4 18
#define PPI_PostProcessInput5 19 // (UNUSED)
#define PPI_PostProcessInput6 20 // (UNUSED)
#define PPI_DecalMask 21
#define PPI_ShadingModelColor 22
#define PPI_ShadingModelID 23
#define PPI_AmbientOcclusion 24
#define PPI_CustomStencil 25
#define PPI_StoredBaseColor 26
#define PPI_StoredSpecular 27
#define PPI_Velocity 28
#define PPI_WorldTangent 29
#define PPI_Anisotropy 30

////////////////////////////////////////////////////////////////////////////////

#define NUM_MATERIAL_TEXCOORDS_VERTEX 1
#define NUM_MATERIAL_TEXCOORDS 1
#define NUM_CUSTOM_VERTEX_INTERPOLATORS 0
#define NUM_TEX_COORD_INTERPOLATORS 1

// Vertex interpolators offsets definition


#if NUM_VIRTUALTEXTURE_SAMPLES || LIGHTMAP_VT_ENABLED
    #include "/Engine/Private/VirtualTextureCommon.ush"
#endif
```

9.4 Material Development

This chapter will cover material development cases, debugging tips, and optimization techniques.

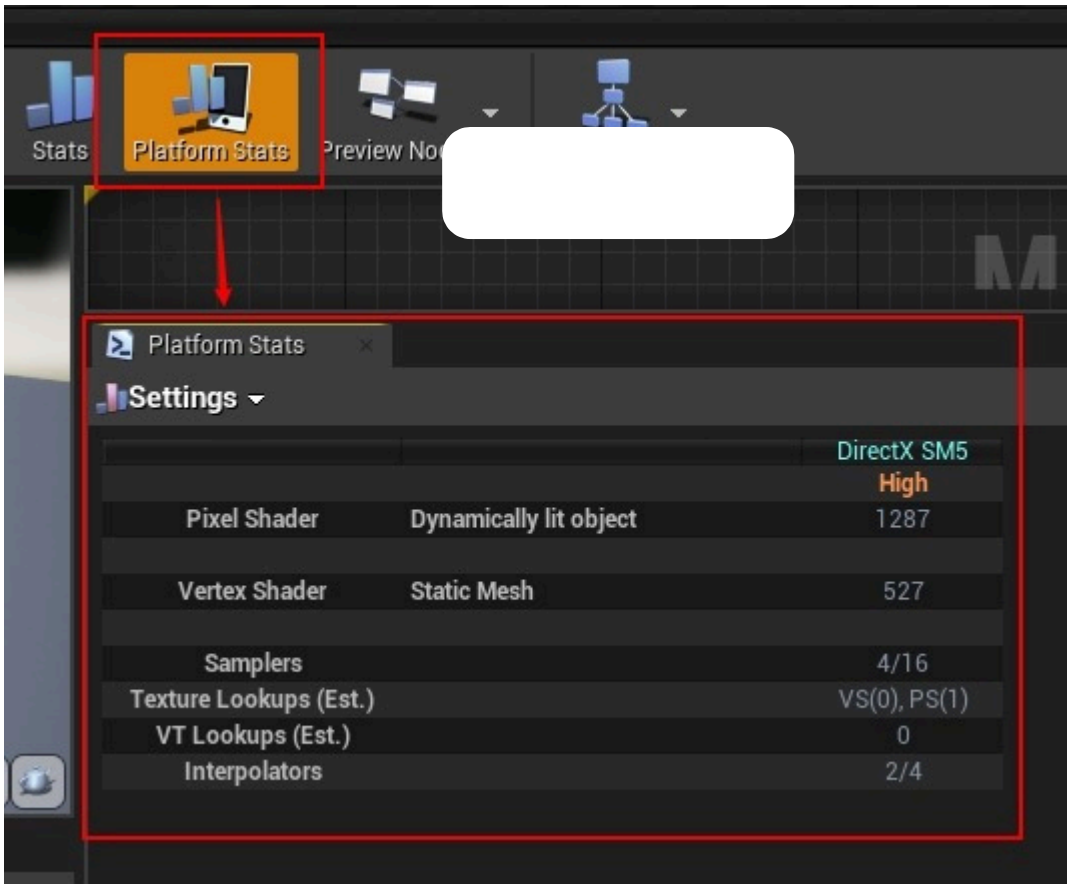
9.4.1 Material debugging and optimization

In the material and material instance editors, you can view the number of instructions, texture sampling number, interpolator data, etc. after the current material is compiled:

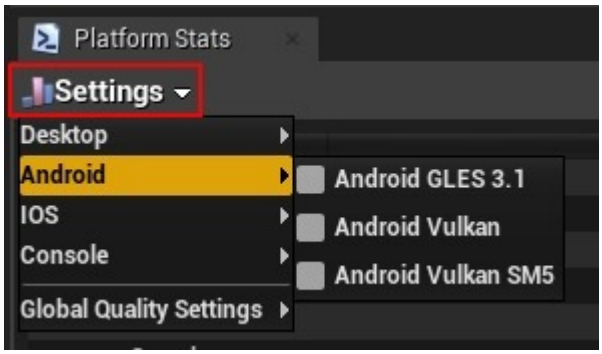


Top: Material Editor's Statistics window. Bottom: Material Instance Editor's Statistics.

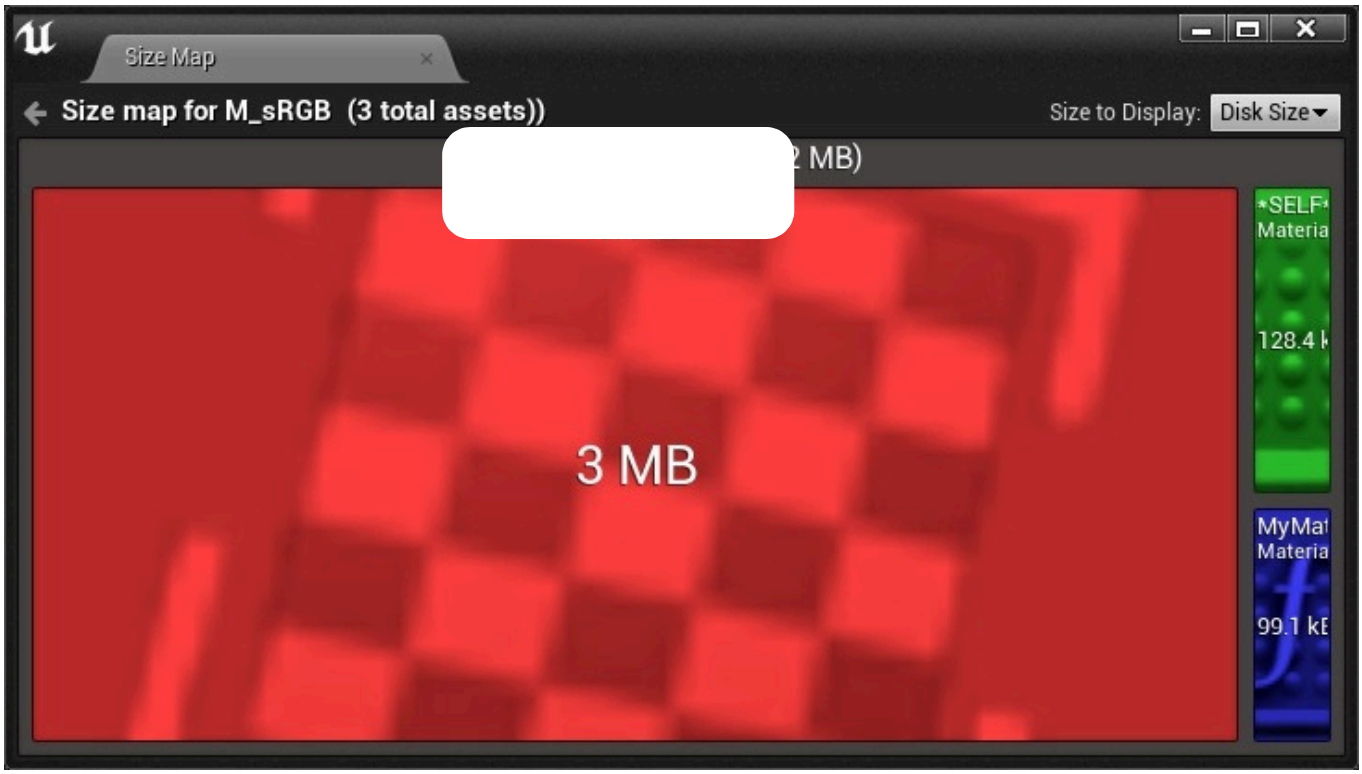
After opening the Platform Stats on the upper side of the Material Editor, you can view more detailed data for the specified platform (PS, VS, samplers, etc.):



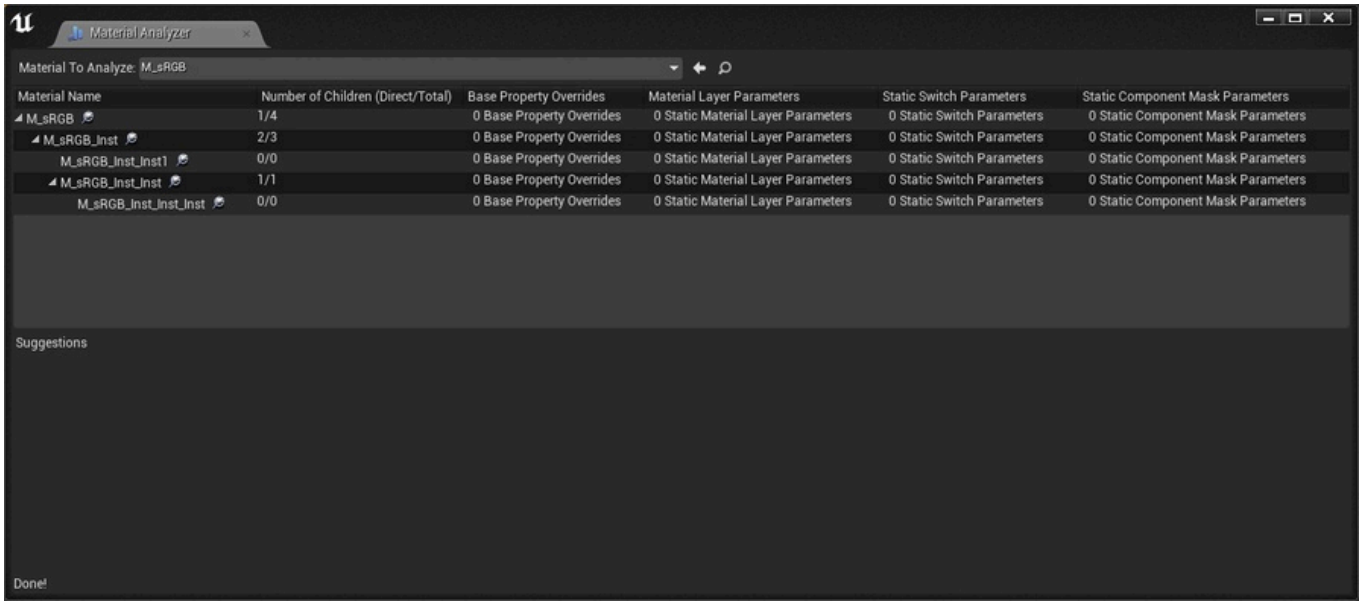
Click the Settings button in the picture above to view data from other platforms.



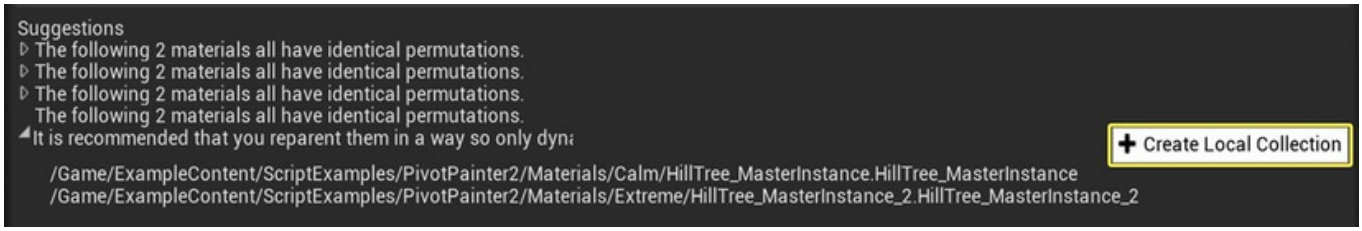
Open the menu Asset / Size Map of the material editor to open the layout of the space occupied by the material resources (disk and memory):



Open the menu Windows / Developer Tools / Material Analyzer of the material editor to view the level of the specified material and the number of various types of attributes:



If some data is abnormal or can be improved, the system will prompt and give modification suggestions. For example, the following picture prompts that multiple materials have the same arrangement:



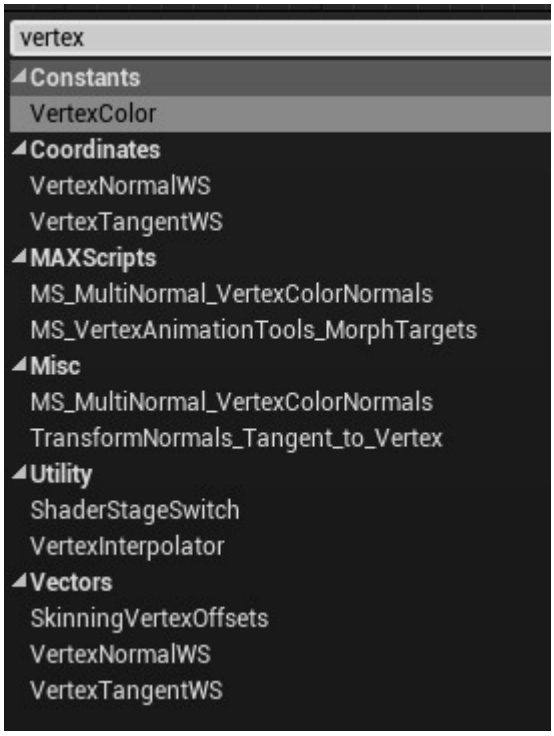
You can click Create Local Collection to place all related instances into a local collection so that they can be easily found and updated for a more efficient material parameter setup.

Of course, the material analyzer above can only manually select one material at a time, which is very slow. In fact, batch inspection tools can be developed based on this, and even certain rules can be customized to automatically create local parameter sets to improve the efficiency and effect of material optimization.

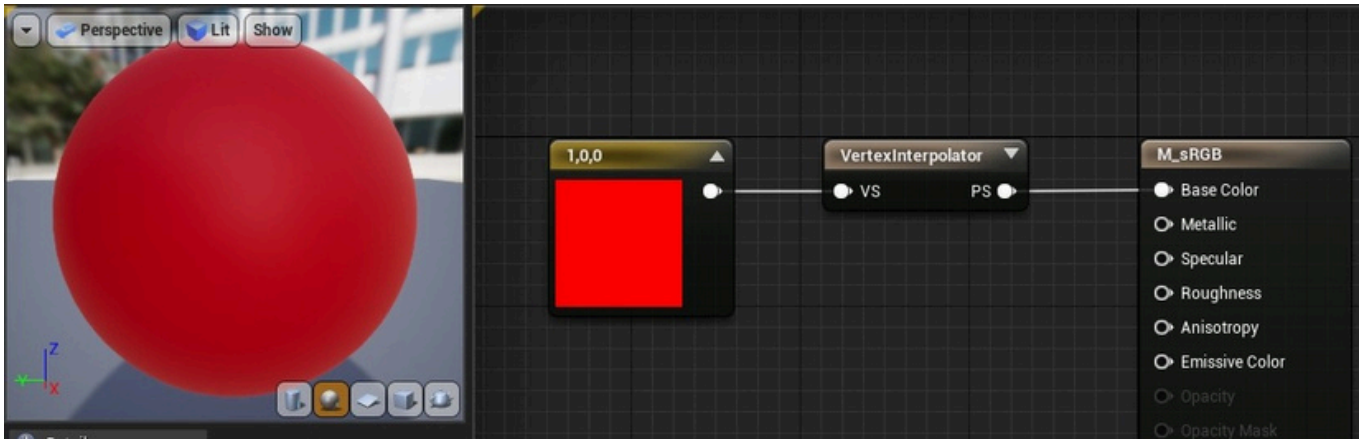
In addition, when editing the material blueprint, you can pay attention to the following points:

1. Pay attention to the number of static variables, switches, levels, and FeatureLevels, which usually increase the number of permutations.
2. Merge material parameters to reduce or avoid unnecessary material nodes.
3. Reasonably add annotations, modularize and abstract into material functions, which is very necessary for complex materials, conductive to maintenance and expansion, and improves material reuse rate.

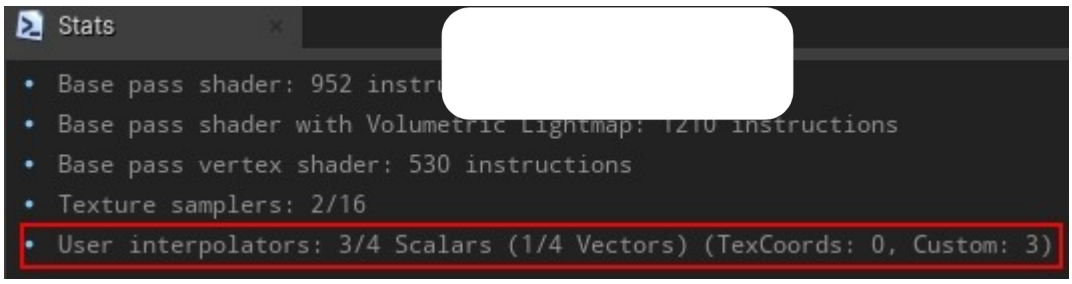
4. Although most of the material nodes are PS nodes, there are also a few VS nodes, as shown below:



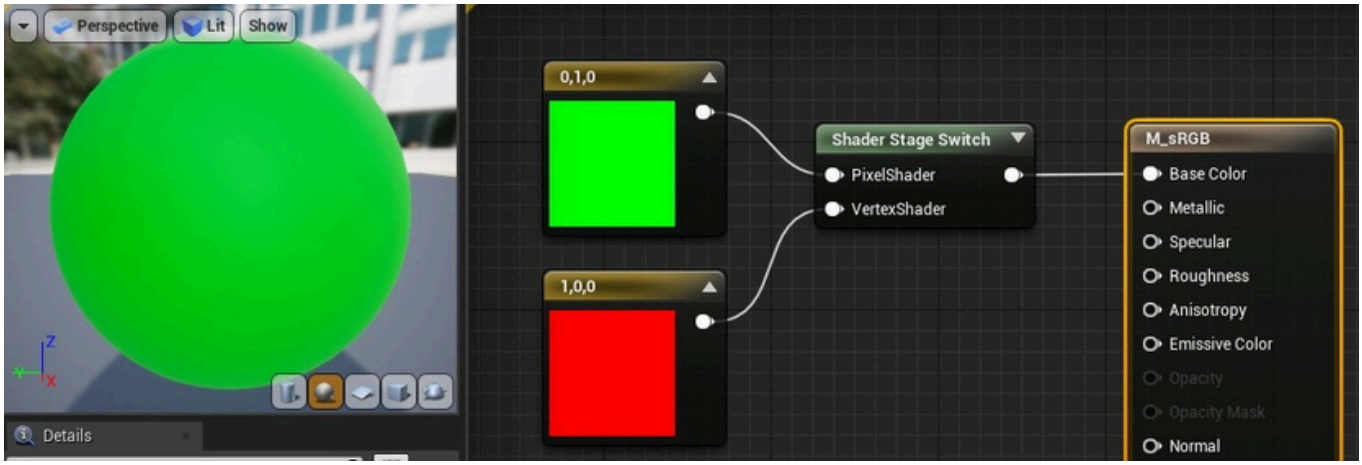
In particular, the VertexInterpolator node can put part of the logic into the VS calculation, and then use the VertexInterpolator to perform hardware interpolation and output the PS value:



VertexInterpolator supports Float~Float4 interpolation and supports up to 4 Float4 interpolations. If you want to know the current number of VertexInterpolators, you can see it in the Stats window:



In addition, Shader Stage Switch can also treat VS and PS values differently:



9.4.2 Material Development Case

9.4.2.1 Adding Material Nodes

This section takes addingMyCustomOperationa material node named as an example.

First, you need to add relevant interfaces and implementations in FMaterialCompiler and FHLSLMaterialTranslator:

```
// Engine\Source\Runtime\Engine\Public\MaterialCompiler.h

classFMaterialCompiler {

public:
    virtual int32MyCustomOperation(int32 A, int32 B) {return0; }//Do not write it as an abstract interface to prevent other subclasses from reporting errors.

    (.....)
};

// Engine\Source\Runtime\Engine\Private\Materials\HLSLMaterialTranslator.h

classFHLSLMaterialTranslator: public FMaterialCompiler {

public:
    virtual int32MyCustomOperation(int32 A, int32 B) override;

    (.....)
```

```
};

//    Engine\Source\Runtime\Engine\Private\Materials\HLSLMaterialTranslator.cpp

int32FHLSLMaterialTranslator::MyCustomOperation(int32 A, int32 B) {

        //Note that both operands are indices, not values!!
        if(A == INDEX_NONE || B == INDEX_NONE) {

                returnINDEX_NONE;
        }

        constuint64 Hash = CityHash128to64({ GetParameterHash(A), GetParameterHash(B) }); if
        (GetParameterUniformExpression(A) && GetParameterUniformExpression(B)) {

                //Since it is a custom operation node, you can specify anyHLSLCode snippet of standard syntax^_^ return
                AddUniformExpressionWithHash(Hash, new
FMaterialUniformExpressionFoldedMath(GetParameterUniformExpression(A),GetParameterUniformE
xpression(B),FMO_Add),GetArithmeticResultType(A,B),TEXT("(%s + %s * 0.5)"
),*GetParameterCode(A),*GetParameterCode(B));
        }
        else
        {
                returnAddCodeChunkWithHash(Hash, GetArithmeticResultType(A,B),TEXT("(%s + %s * 0.5)"
                ),*GetParameterCode(A),*GetParameterCode(B));
        }
}
```

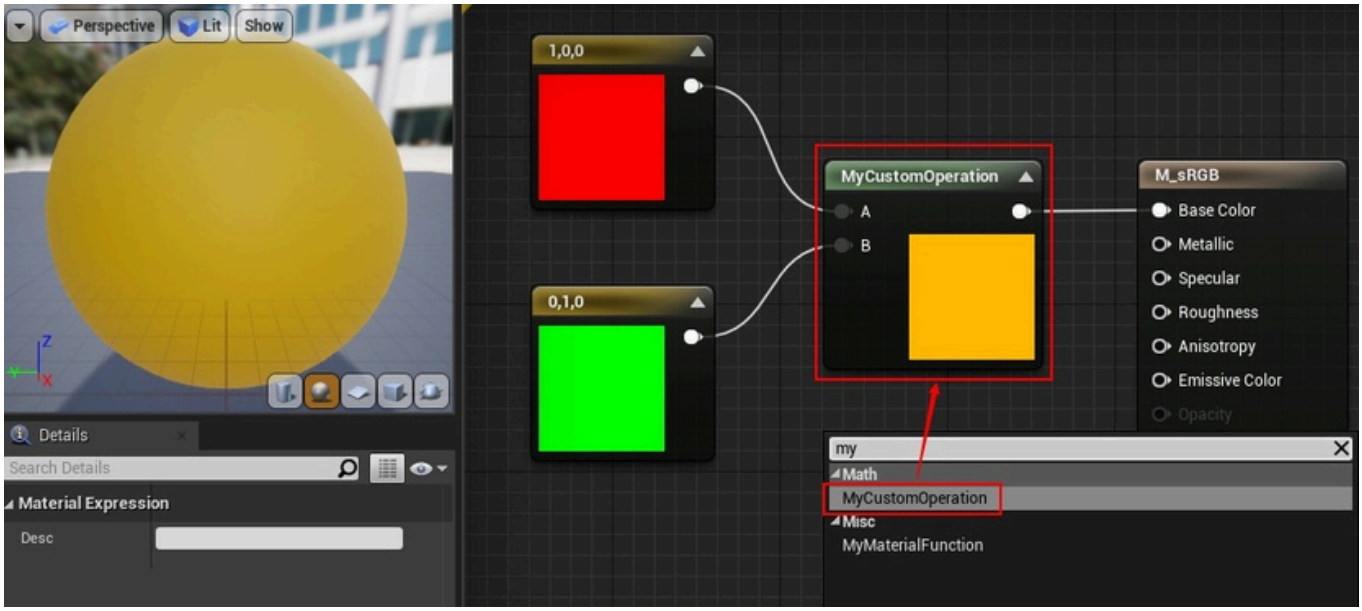
After implementing the above interface, you need to add the corresponding material expression type and file:

```
// Engine\Source\Runtime\Engine\Classes\Materials\MaterialExpressionMyCustomOperation.h

#Pragmasonce

#include     "CoreMinimal.h"
#include     "UObject/ObjectMacros.h"
#include     "MaterialExpressionIO.h" "Materials/
#include     MaterialExpression.h"
//UBTCompiled header files,    Can't be missed.
#include"MaterialExpressionMyCustomOperation.generated.h"

UCLASS (MinimalAPI)
classUMaterialExpressionMyCustomOperation: public UMaterialExpression {

        GENERATED_UCLASS_BODY()

        UPROPERTY(meta = (RequiredInput ="false", ToolTip ="Defaults to 'ConstA' if not specified"))

        FExpressionInput A;

        UPROPERTY(meta = (RequiredInput ="false", ToolTip ="Defaults to 'ConstB' if not specified"))

        FExpressionInput B;

        UPROPERTY(EditAnywhere, Category=MaterialExpressionAdd, meta=(OverridingInputProperty
```

```
="A"))
	floatConstA;

	UPROPERTY(EditAnywhere, Category=MaterialExpressionAdd, meta=(OverridingInputProperty "B"))
=
	floatConstB;

	//~ Begin UMaterialExpression Interface
#if	WITH_EDITOR
	virtual int32Compile(class FMaterialCompiler* Compiler, int32 OutputIndex) override; virtualvoid GetCaption
	(TArray<FString>& OutCaptions)constoverride;
#endif// WITH_EDITOR
	//~ End UMaterialExpression Interface
};

// Engine\Source\Runtime\Engine\Private\Materials\MaterialExpressions.cpp

//Add header file reference for new node.
#include"Materials/MaterialExpressionMyCustomOperation.h"


//Default constructor.
UMaterialExpressionMyCustomOperation::UMaterialExpressionMyCustomOperation(const
FObjectInitializer& ObjectInitializer)
	: Super(ObjectInitializer)
{
	// Structure to hold one-time initialization struct
	FConstructorStatics {

		FText NAME_Math;
		FConstructorStatics()
			: NAME_Math(LOCTEXT("Math","Math"))
		{
		}
	};
	staticFConstructorStatics ConstructorStatics;

	ConstA	=	0.0f;
	ConstB	=	1.0f;

#if	WITH_EDITORONLY_DATA
	MenuCategories.Add(ConstructorStatics.NAME_Math);
#endif
}

//Compile.
int32UMaterialExpressionMyCustomOperation::Compile(class FMaterialCompiler* Compiler, int32 OutputIndex)

{
	int32 Arg1 = A.GetTracedInput().Expression ? A.Compile(Compiler) : Compiler-
> Constant(ConstA);
	int32 Arg2 = B.GetTracedInput().Expression ? B.Compile(Compiler) : Compiler-
> Constant(ConstB);

	returnCompiler->MyCustomOperation(Arg1, Arg2);
}

//illustrate.
```

```
void UMaterialExpressionMyCustomOperation::GetCaption(TArray<FString>& OutCaptions)const {

	FString ret = TEXT("MyCustomOperation");

	FExpressionInput ATraced = A.GetTracedInput();
	FExpressionInput BTraced = B.GetTracedInput();
	if(!ATraced.Expression || !BTraced.Expression) {

		ret += TEXT("(");
		ret += ATraced.Expression ? TEXT(",") : FString::Printf( TEXT("%.4g,"), ConstA); ret += BTraced.Expression ?
		TEXT(")") : FString::Printf( TEXT("%.4g)"), ConstB);
	}

	OutCaptions.Add(ret);
}
```

After adding the above code, compile the engine code, start the UE editor, open the material editor, and you can search for the MyCustomOperation material node in the node and use it (see the figure below).



9.4.2.2 Extending custom nodes

Although the Custom material node that comes with UE can write any code, it is limited to a certain function. For example, the following code is written in Custom:

```
return1.0;
```

In fact, the HLSL compiler will compile it into code similar to the following:

```
float CustomExpression0() {

	return1.0;
}
```

Being limited to the function body will greatly limit our performance. For example, we cannot define functions and structures in the Custom node, nor can we reference other ush files.

The common and quick way is to add curly braces around the Custom stage:

```
return0.0; }//Matches the compiler's {

floatMyParameter =1.0;

//Normal code.
float MyFunction()
{
	returnMyParameter;
//You don't need to add } here, because the compiler will add it later
```

The generated HLSL code looks like this:

```
float CustomExpression0() {

	return1.0;
}

floatMyParameter =1.0;

float MyFunction()
{
	returnMyParameter;
}
```

But this method is very inelegant, and the result obtained by the material node is the value of the first return. Here is another slightly more elegant way that supports the definition of multiple functions and variables:

```
structFMyStruct
{
	float3 ColorDensity;

	float3ColorOperation(float3 Color) {

		returnColor * ColorDensity;
	}

	float3	Out()
	{
		//	InColoris the function input parameter.
		return	ColorOperation(InColor);
	}
};

FMyStruct MyStruct;
FMyStruct.ColorDensity = float3(0.5,0.5,0);

returnMyStruct.Out();
```

The above code uses the feature of HLSL that local structures can be defined within a function body, supporting the definition of any number of functions and variables.

9.4.2.3 Extending Material Templates

MaterialTemplate.ush defines a large number of global macros, structures, and interfaces. We can of course modify it and add the required code or data, such as:

- Add global static variables.
- Add macro definition.
- Add reference files.
- Add the missing code of %s to fill it in the HLSL compiler. Add
- structure.
- Add custom functions.

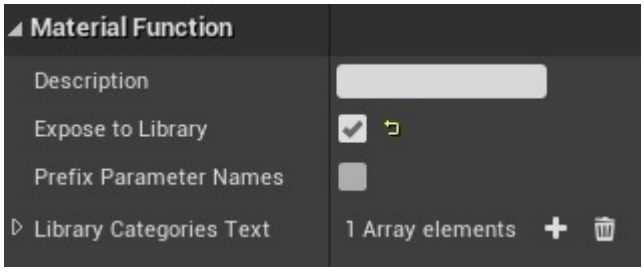
All the above data and interfaces can be combined with Custom nodes and MaterialFunction and exposed to material blueprints for access.
For example, let's add a global static constant to MaterialTemplate.ush:

```
// Engine\Shaders\Private\MaterialTemplate.ush

//Next toincludeafter.
static const floatMyGlobalParameter =0.5;
```

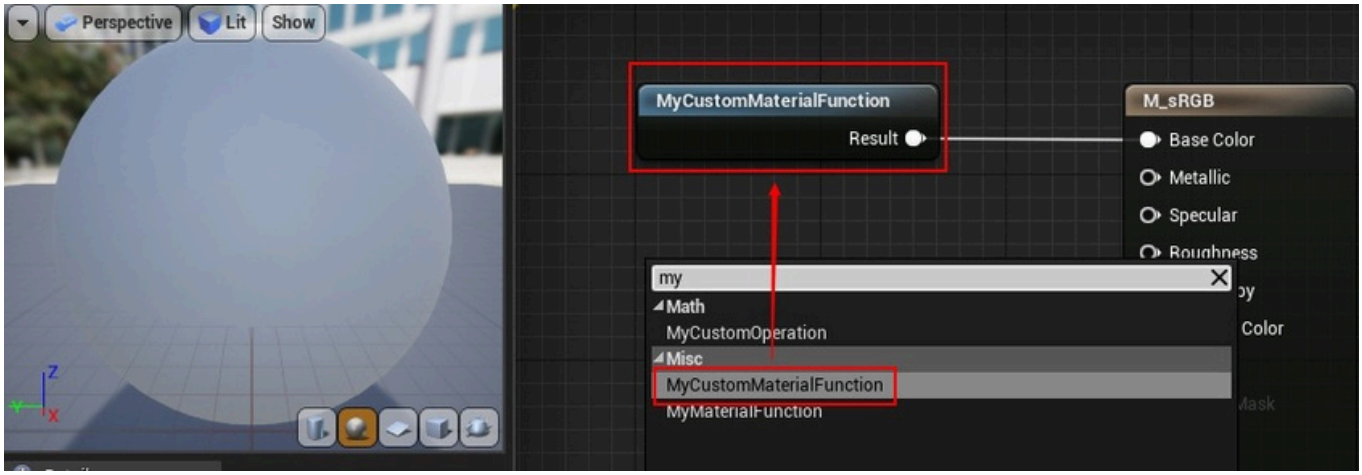
Compile the engine, start it and open the material editor, add a new material function, and add a Custom node to the material function. The code of the Custom node is as follows:

```
returnMyGlobalParameter;
```

Then check Expose to Library for the material function:



Now you can search and apply the material function in the material node:



9.5 Summary

This article mainly explains the basic concepts, types, and mechanisms of UE's material system. I hope that after studying this article, you will no longer be unfamiliar with UE's materials and will be able to easily master, apply, and expand them.

9.5.1 Thoughts on this article

As usual, this article also arranges some small thoughts to help understand and deepen the mastery and understanding of the UE material system:

- What are the connections and differences between UMaterialInterface, FMaterialRenderProxy, and FMaterial? Why are there so many material types?
- What is the data update process when rendering materials?
- What is the process of compiling a material blueprint? What are the main interface functions of the compiler?
- When editing materials, what performance data should I pay attention to? How can I view it? Added
- a new Shading Model to support 2D style rendering.

-
-
-
-
-

References

- Unreal Engine Source
- Rendering and Graphics
- Materials
- Graphics Programming

- Analysis of UE4 rendering module
- UE4 Render System Sheet
- 【UE4 Renderer】<03> PipelineBase
- UE4 material system source code analysis: UMaterial and material node introduction UE4
- material system source code analysis: material compiled into HLSL CODE
- UE4 HLSL and Shader Development Guide and Tips
- Material Analyzer
- Unreal Engine4 Custom Function
- Unreal Engine 4 Rendering Part 6: Adding a new Shading Model

https://github.com/pe7yu