Analysis of Unreal Rendering System (01) - Overview and Basics

Table of contents

1.1 Introduction to Unreal

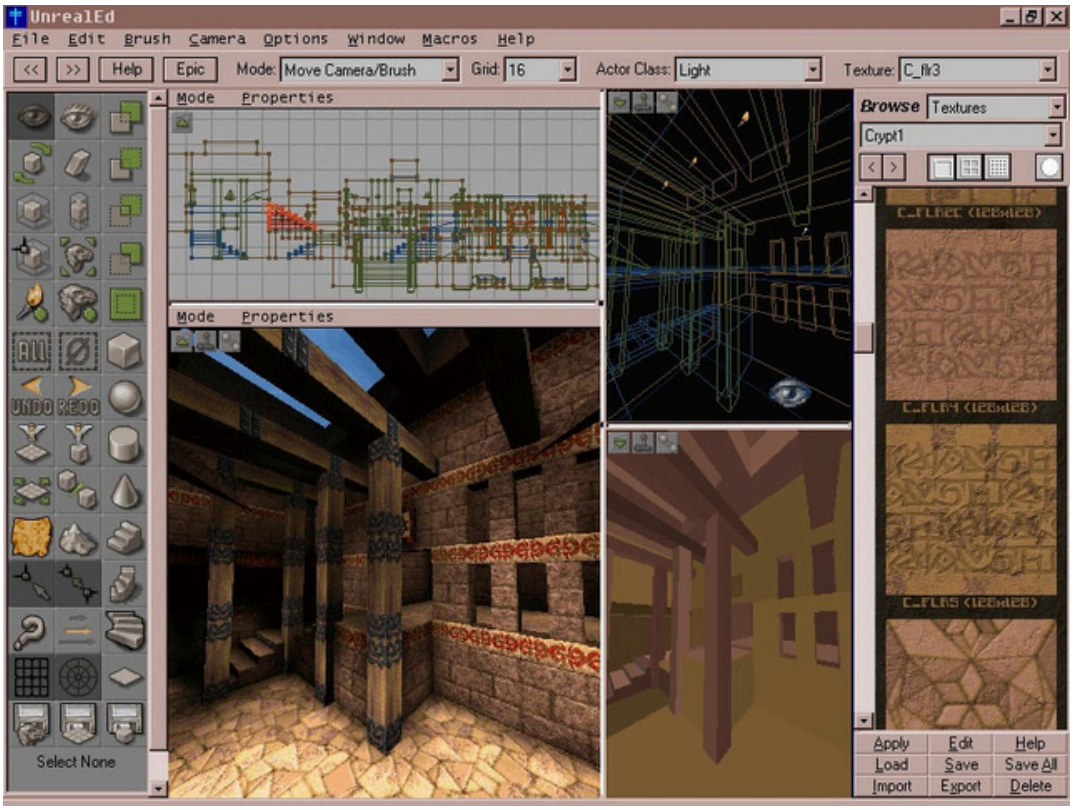
Unreal Engine (UE) is a commercial engine that integrates graphics rendering and development kits. After decades of development and accumulation, it has st and has become a global general-purpose commercial engine leading the field of real-time rendering. It is widely used in games, design, simulation, film and t industries. It comes from the game company Epic Games and was originally managed by Tim Sweeney. It started in the mid-1990s and has been in existence f version iterations.

1.1.1 Unreal Engine 1 (1995)

Tim Sweeney led the development in 1995, and the first game Unreal was developed in 1998. It was a third-person shooter game, which opened the door to t Engine.

As the first generation engine, it has the following features:

- Colored lighting.
- A limited form of texture filtering.
- Collision detection.
- Scenario Editor.
- Soft renderer (the CPU executes drawing instructions and then moves them to the hardware-accelerated Glide API).



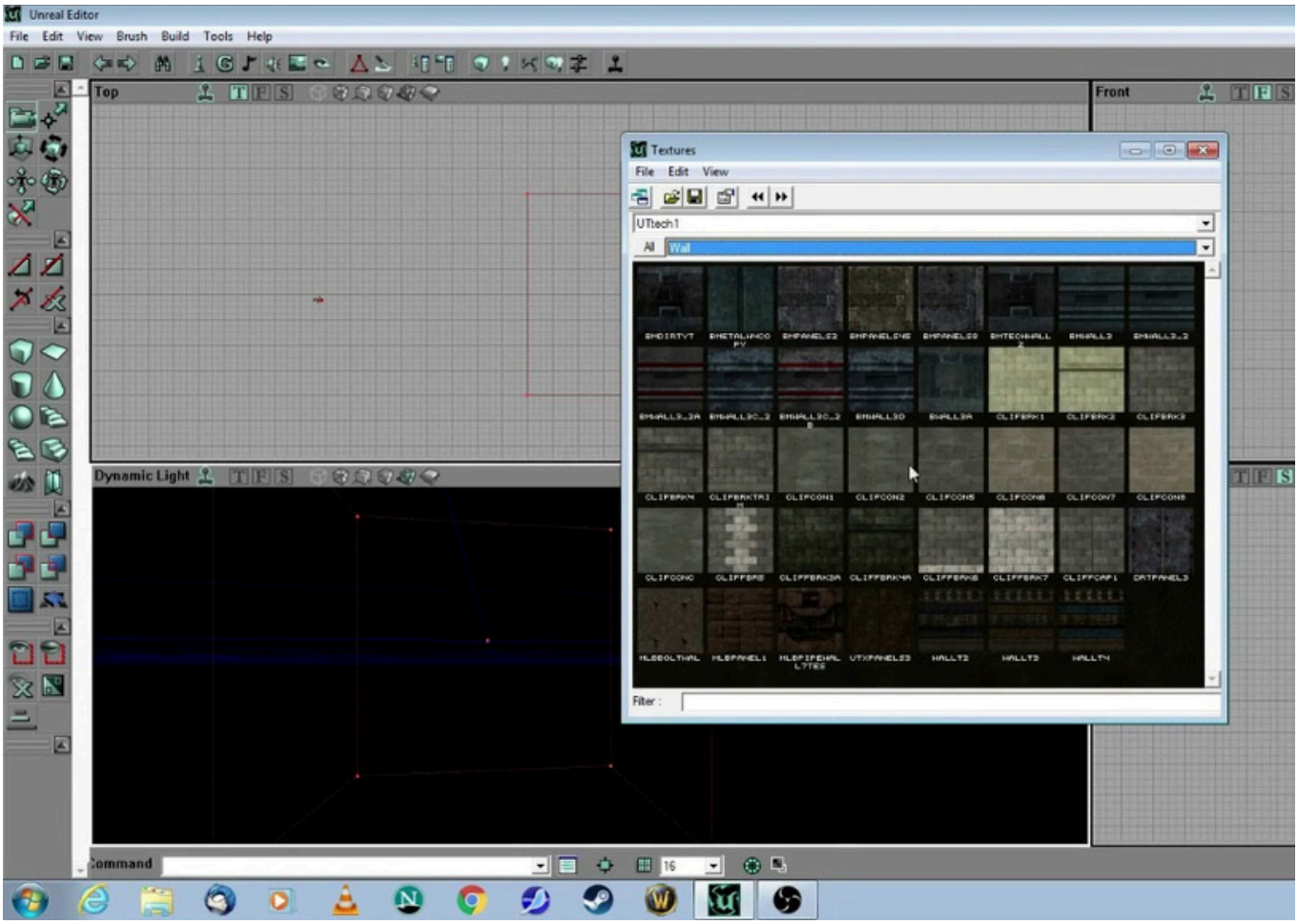
The interface of the first Unreal Engine editor.

1.1.2 Unreal Engine 2 (1998)

Unreal Engine 2 was still developed by Tim Sweeney. Development began in 1998, and the second version was completed in 2002. Several games were also de multiplayer shooting game America's Army.

Compared with the first generation, the features of the second generation of Unreal are mainly reflected in:

- A more complete tool chain.
- Cinematic editing toolchain.
- Particle system.
- Supports DCC skeletal animation export and other plug-ins.
- The wxWidgets toolkit based on C++.
- Physical simulation based on Karma physics engine: ragdoll collision, rigid body collision, etc.



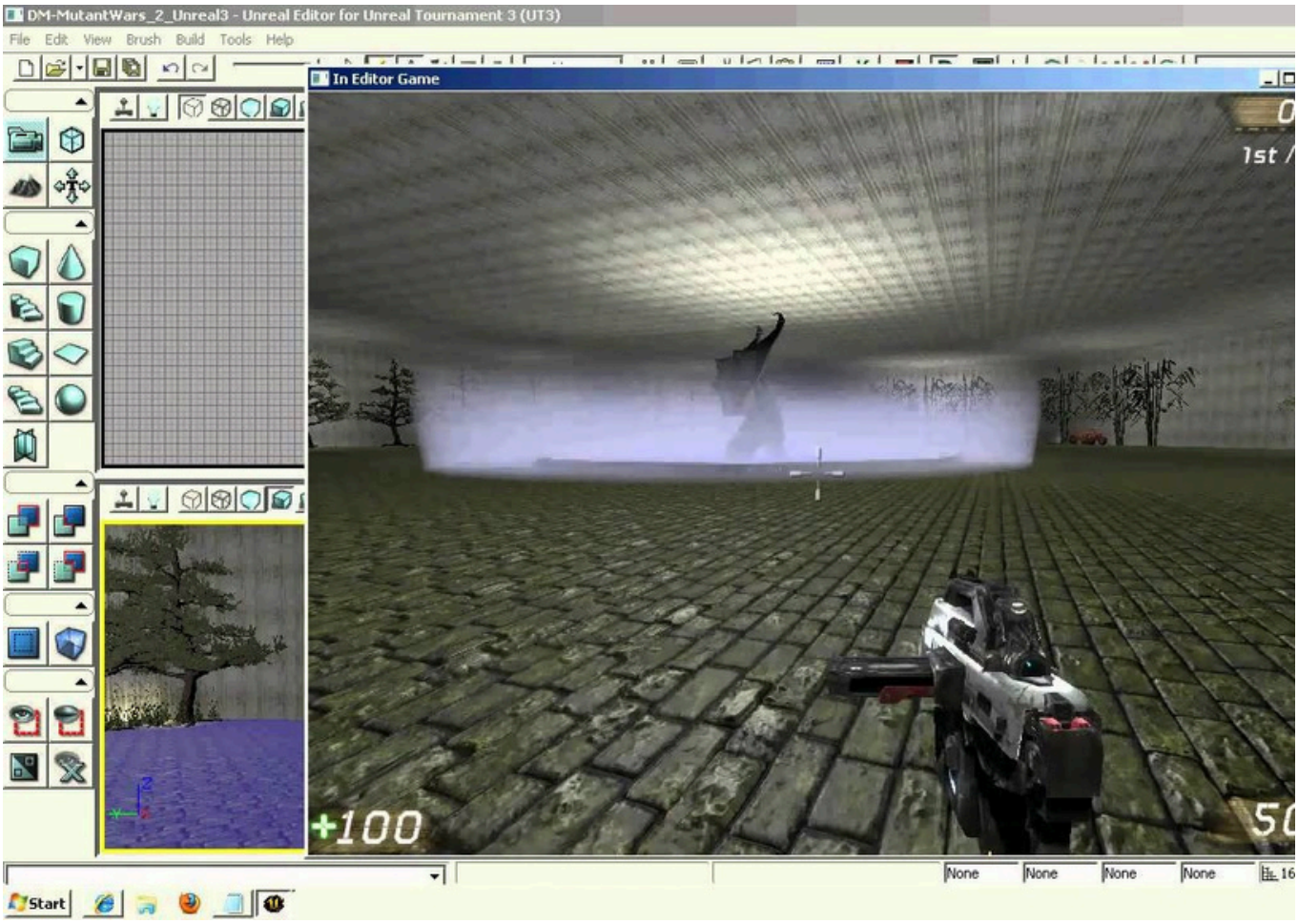
Unreal Engine 2 editor interface.

Screenshot of Killing Floor, a game developed based on Unreal Engine 2.

1.1.3 Unreal Engine 3 (2004)

Unreal Engine 3 was released in 2004 after a year and a half of closed-door development. This version also brought many new features to the industry, mainly

- Object-oriented design.
- Data driven scripts.
- Physical systems.
- Sound system.
- A new dynamic WYSIWYG toolchain.
- Programmable rendering pipeline.
- Pixel-by-pixel lighting and shading calculations.
- Gamma-corrected HDR renderer.
- Destructible environments.
- Dynamic software body simulation.
- Group role simulation.
- Real-time GI solution.



Unreal Engine 3 editor interface.

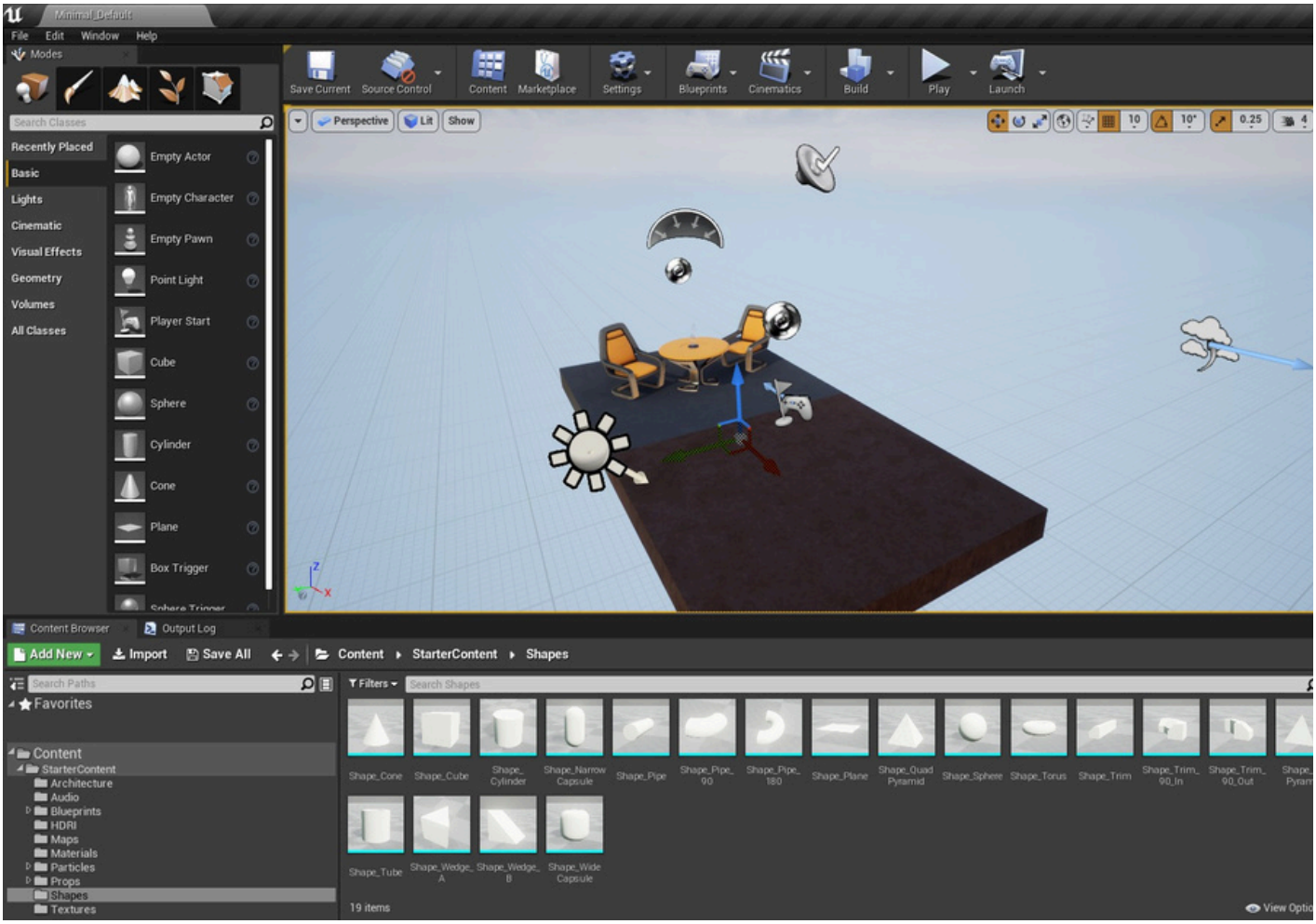
There are many representative games of Unreal 3, including Gear of War, RobotBlitz, Infinity Blade, Dungeon Defenders, Batman: Arkham City, Aliens: Colonial

A sneak peek at Batman: Arkham City gameplay footage.

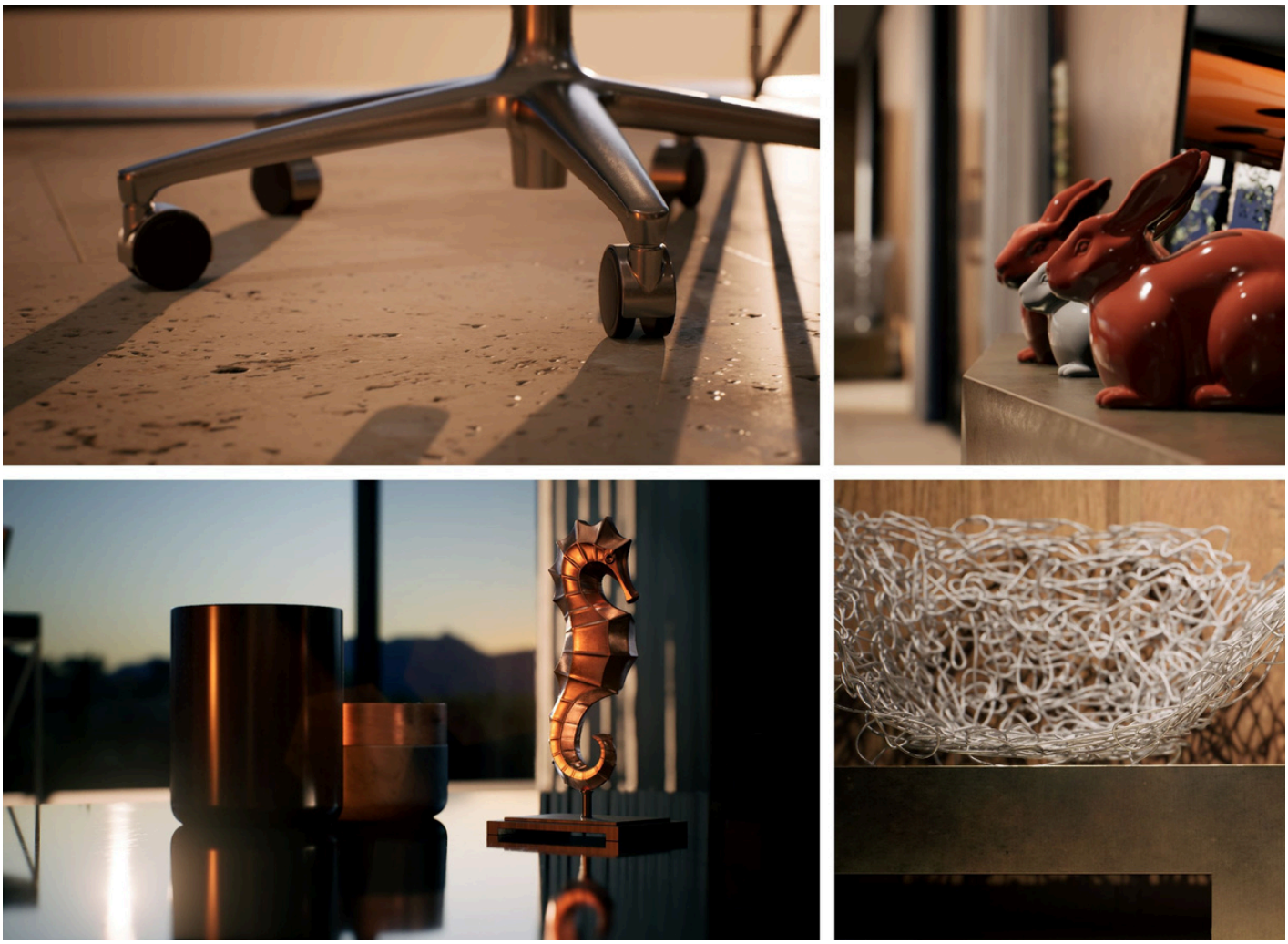
1.1.4 Unreal Engine 4 (2008)

Unreal Engine 4 was released as early as 2008, and it has been 12 years since then. It has gone through more than 20 versions and introduced countless amazi

- PBR rendering pipeline and supporting toolchain.
- Real-time ray tracing based on DXR and RTX. Blueprint
- system.
- Visual material editor.
- Deferred rendering pipeline.
- Lightweight rendering pipeline for mobile platforms. VR
- rendering pipeline.
- Niagara and other GPU particles.
- More realistic physics simulation (destruction, collisions, soft bodies, etc.).
- A more complete game and film and television production tool chain.
- Support more mainstream platforms.
- .....



A look at the UE4 Editor.



An overview of UE4.22 real-time ray tracing images.

With the development of UE4 and the change of Epic Games' strategy, a shocking decision was finally made in 2015: free for all users and open source code. F can study the source code of UE, and this series of articles was born.

There are more and more blockbuster games developed based on UE4, including Gears of War 4, Dead at Dawn, PlayerUnknown's Battlegrounds, Game for Pe Final Fantasy VII Remake, Code Vein, etc.

Final Fantasy 7 Remake has truly gorgeous and dynamic graphics.

In addition to the gaming industry, industries such as film and television, simulation, design, radio and television, and scientific visualization have also gradually produced and gradually improved the corresponding tool chain.

Film-quality virtual characters rendered by Unreal Engine 4.

1.1.5 Unreal Engine 5 (2021)

In May 2020, Unreal officially released a video "Lumen in the Land of Nanite" to showcase the rendering features of Unreal Engine 5. The video showcases Nan and Lumen technology for real-time global illumination, bringing a cinematic audio-visual experience to real -time games. This is not a game, it's a movie! I be by this video at the time, and it really amazed everyone. When I first saw this video, I was so excited that I watched it many times.

A frame from the "Lumen in the Land of Nanite" demonstration video.

According to the official introduction,**Nanite**supports hundreds of billions of polygons on the same screen, which means that traditional art production process mapping are no longer required, and high-poly rendering is directly adopted.**Lumen**is a fully dynamic global illumination solution that can respond to scene the need for dedicated ray tracing hardware. The system can render indirect specular reflections and diffuse reflections that can bounce infinitely in grand and

In addition, the video also shows new features such as the Chaos physics and destruction system, Niagara VFX, convolution reverb, and ambient stereo render

As for the release date of UE5, it is straightforward to quote the official statement:

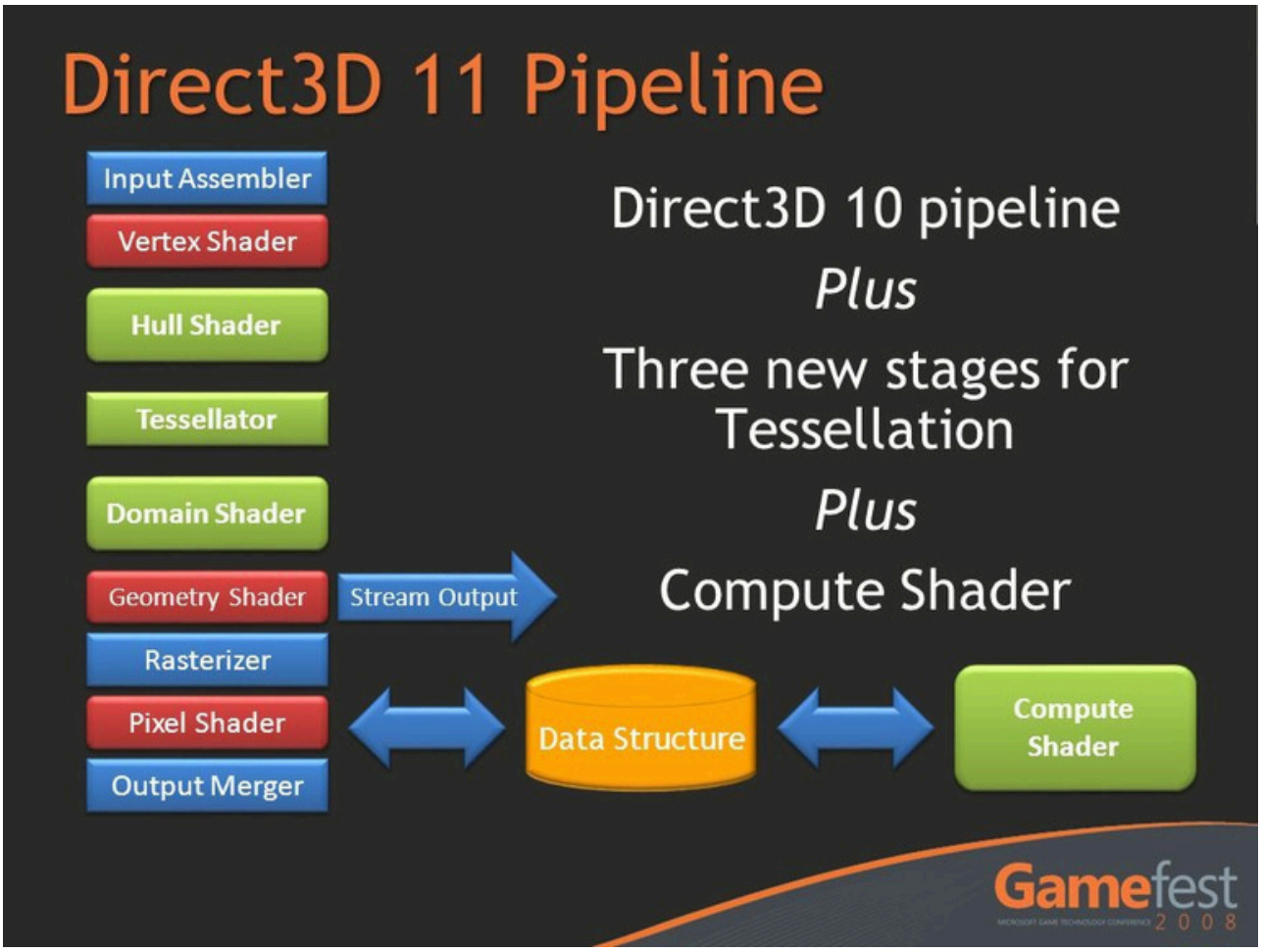
> **Unreal Engine 4 and 5 release schedule**
> Unreal Engine 4.25 already supports the next-generation console platforms from Sony and Microsoft. Epic is currently working closely with console manufac publishers to develop next-generation games using Unreal Engine 4.
> **Unreal Engine 5 will be released in preview in early 2021 and in full version in late 2021.**It will support next-generation consoles, current-generation co platforms.
>
> We are designing forward-compatible features so that you can start next-generation development in UE4 first and migrate your project to UE5 when the tim We will release Fortnite, developed with UE4, on next-generation consoles after they are released. In order to prove our industry-leading technology throug game to UE5 in mid-2021.

In other words, if UE officials don't let down their guard, they will release the full version of UE5 in 2021. Let's wait and see.

1.2 Rendering Overview

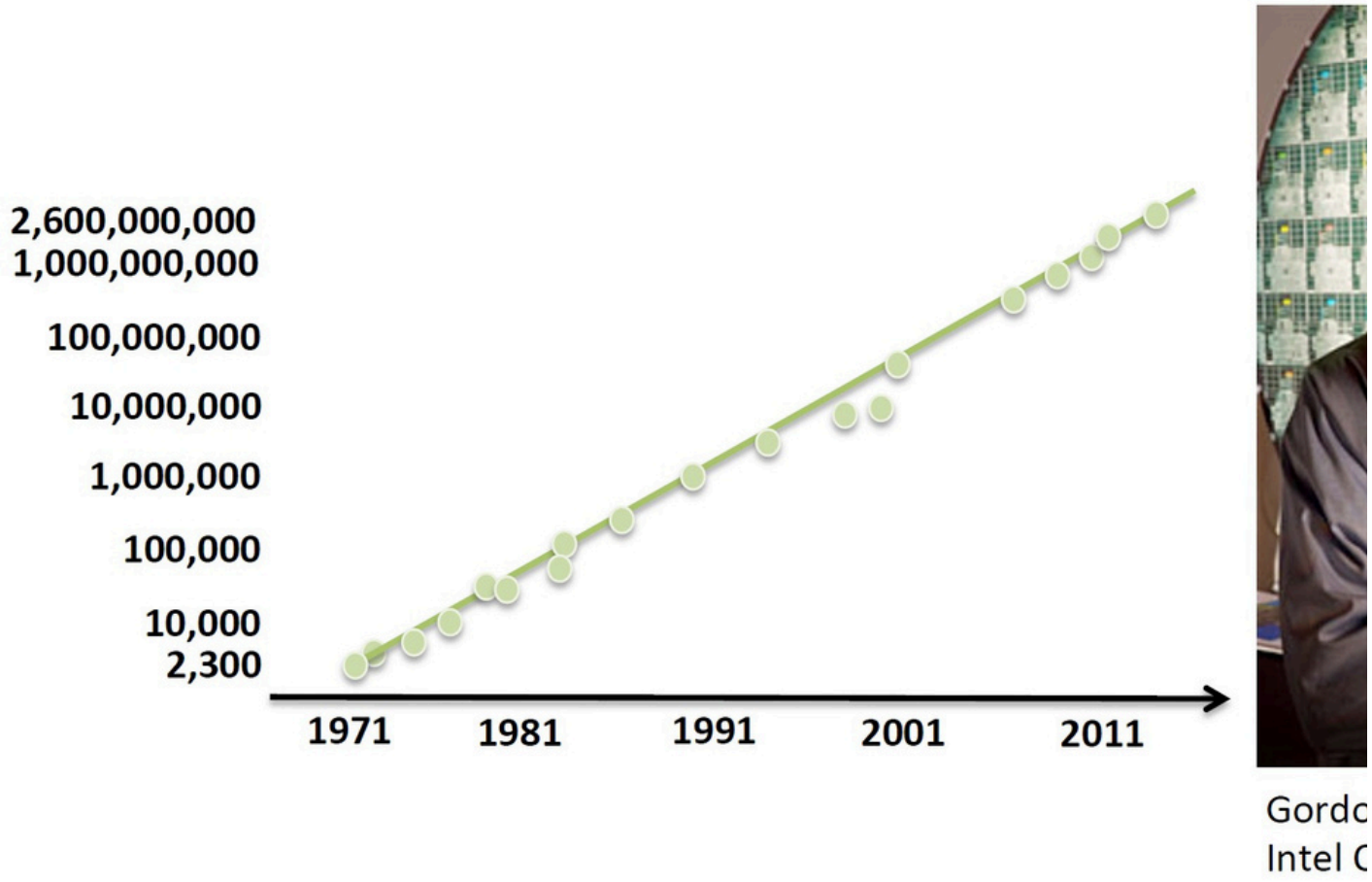
1.2.1 Evolution of Unreal Rendering

Looking at the development history of UE, UE is actually in line with the development trend of hardware technology and software technology, and is good at c hardware, plus the result of software engineering, operating system and other technical encapsulation. For example, with the development of hardware in the pipeline was added, and in 1998 it was increased to 32-bit RGBA; at the beginning of this century, after the emergence of hardware-based programmable rend fixed pipelines (now abandoned) and programmable rendering pipelines. In the following years, HDR emerged, delayed rendering pipelines were introduced, a etc. all followed this rule.

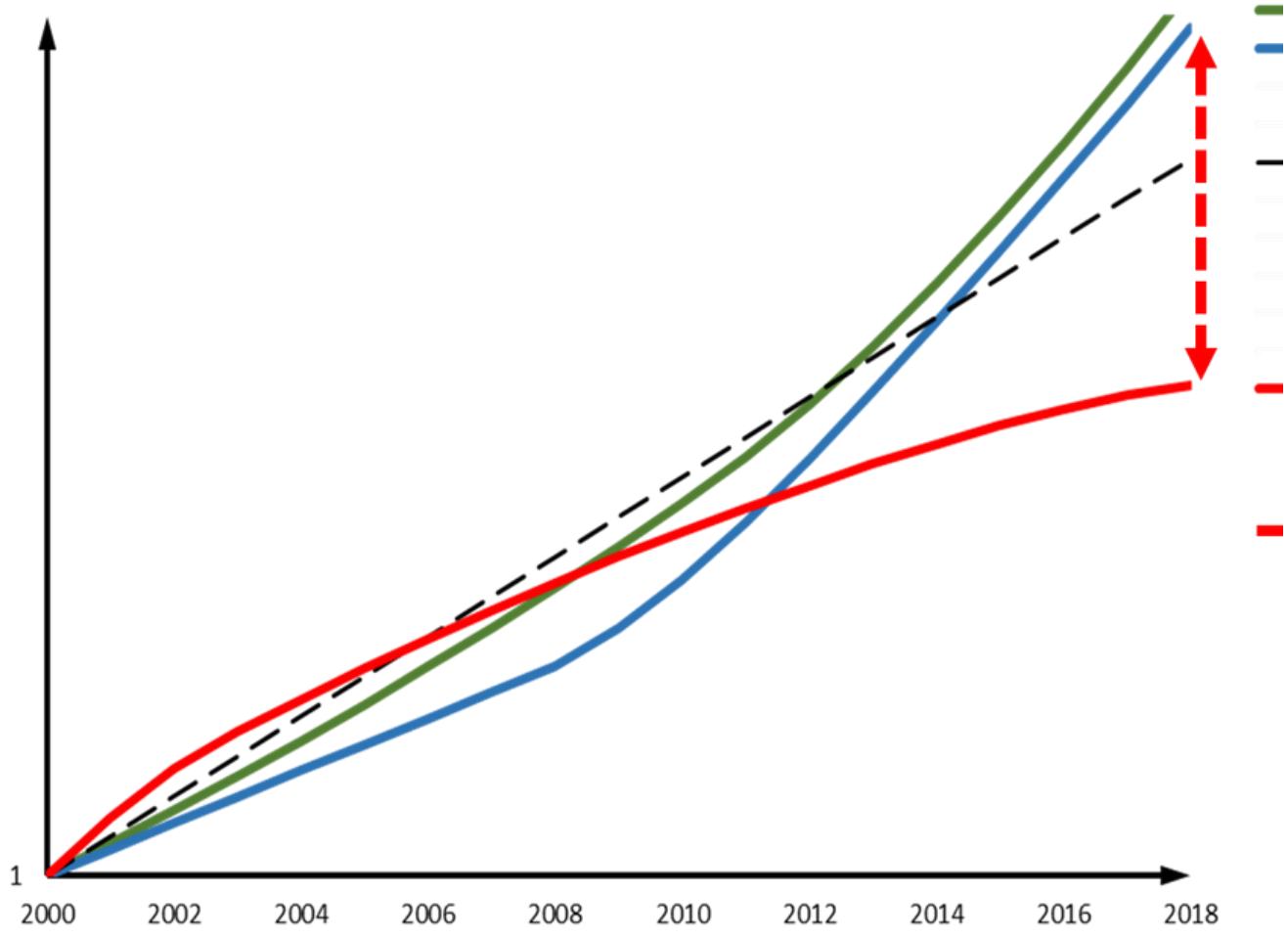


The picture shows the new features of DirectX 11, such as surface subdivision and compute shader.

Until a few years ago, Moore's Law of CPU reached its ceiling, and CPU manufacturers had to change their strategies to vigorously develop multi-core. As a res parallelism has been strongly developed, and with the emergence of lightweight, multi-threaded friendly graphics APIs such as Vulkan, DirectX12, and Metal, rendering to give full play to the performance advantages of modern CPU multi-core and GPU massive computing units.

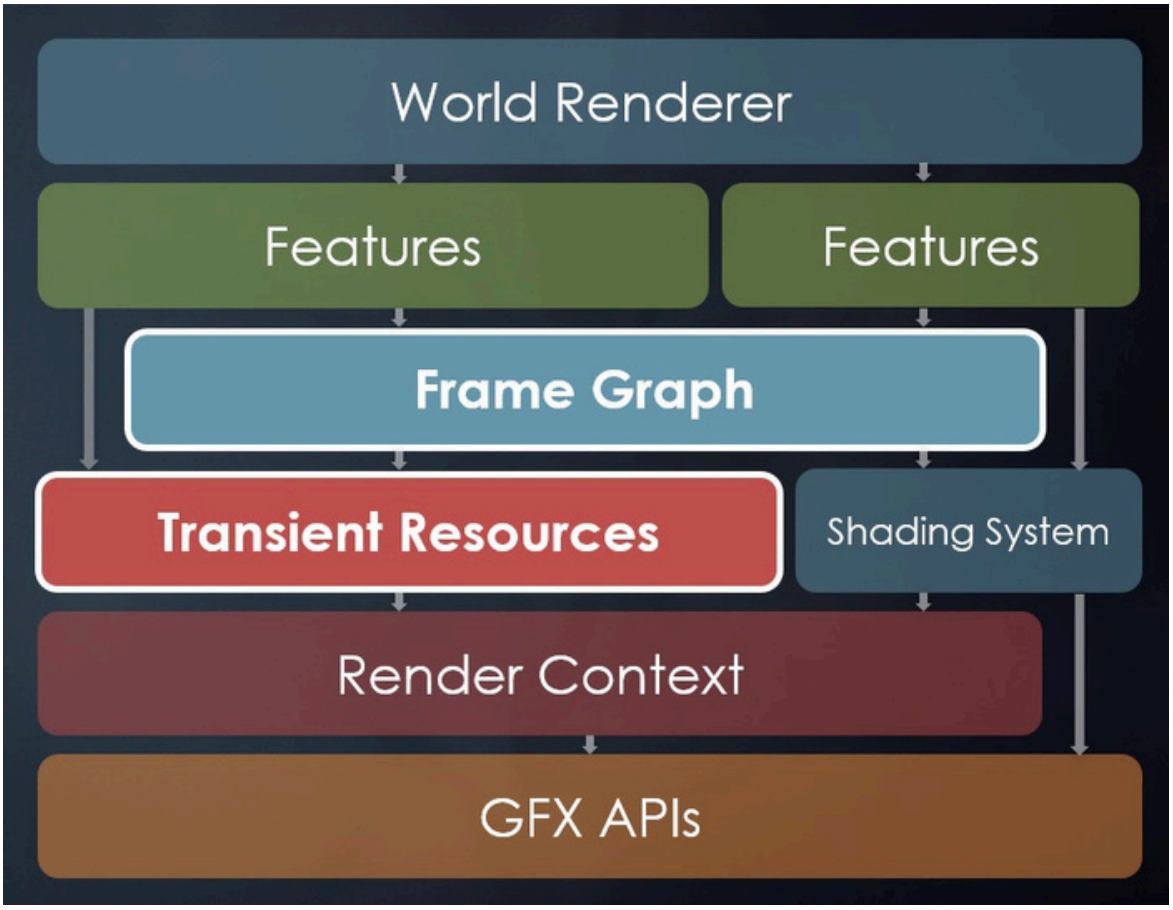


The CPU core frequency growth has maintained Moore's Law from 1970 to 2011, but with the stagnation of chip technology development, it has obviously not figure on the right is Intel founder Moore himself.)



Around 2006, CPU performance lagged significantly behind the Moore's Law curve, but at the same time, the number of CPU cores also increased.

Since UE needs to support many mainstream operating systems and encapsulate many graphics APIs and corresponding shaders, the rendering system of UE by layer, turning the originally simple API into the complex UE system today. For example, in order to cross multiple graphics APIs, the RHI system is added to APIs at the user level; in order to facilitate users to use and edit material effects, material templates and material editors are introduced, and a series of interme compile shaders to the corresponding hardware platform; in order to give full play to the advantages of multi -core, game threads, rendering threads, and RHI thread access conflicts and competitions, game thread representatives starting with U are introduced, and there are corresponding rendering thread representation rendering such as model batching and reducing DrawCall, dynamic and static rendering paths are added, and concepts such as FMeshPassProcessor, FMeshBa order to adapt to and make full use of new lightweight modern graphics APIs such as Vulkan and DirectX12, UE also introduced RDG (rendering dependency g



Frame Graph (or RDG) separates the engine functional modules and GPU resources, making the structure clearer and enabling targeted scheduling optimizatio

In the past few years, AI technology has risen with the trend and has been introduced into the field of graphics by industry researchers, giving full play to the f with the integration of Tensor Core and Raytrace Core in NVIDIA's Turing hardware architecture, and Microsoft's integration of the standard API for ray tracing has finally ushered in spring and has flourished. The first general commercial engine to integrate real-time ray tracing is UE 4.22, and the corresponding demo

When Epic Games released UE 4.22, it announced support for real-time ray tracing and released the demo video "Troll" in collaboration with Goodbye Kansas

In summary, the main reasons why the UE rendering system is in such a complex situation today are:

- Adapt to the development of software and hardware technology. Cater to the
- ideas and designs of object-oriented software engineering.
- The architecture is modularized to improve reusability and scalability and reduce coupling. Cross-
- platform, cross-compiler, cross-graphics API.
- Compatible with old functions, codes, and interfaces.
- Improve rendering efficiency, performance ratio, and robustness.
- Encapsulate the underlying details of the API and abstract away the details and complexity of the rendering system in order to reduce the learning and u programmers, artists, TAs, planners, etc.).
- In order to improve the versatility of the engine, it is necessary to add multi-level and multi-concept encapsulation.

Looking at the development of the entire graphics rendering industry, the goal of industry R&D personnel is the same, that is:**to make full use of limited har or more stylized images faster and better.**

1.2.2 Content Scope

Many people have written books (such as"A Brief Analysis of Unreal Engine Programming") or technical articles (such asUnreal Engine 4 Rendering series ,"F Rendering System Architecture" and many Zhihu articles) analyzing Unreal Rendering, but I think their articles can only reveal part of the UE rendering system. of articles that can fully analyze the whole picture of the UE rendering system. In view of this, I dare to take on this important task, but after all, my energy and mistakes or omissions, I sincerely ask readers to correct them.

This series of articles focuses on analyzing the UE rendering system. More specifically, it is mainly limited to the source code of the following UE directories:

- Engine\Source\Runtime\RendererCore.
- Engine\Source\Runtime\Renderer.
- Engine\Source\Runtime\RHI.
- Some RHI modules: D3D12RHI, OpenGLDrv, VulkanRHI, etc. Some
- basic modules: Core, CoreUObject, etc.

Of course, if necessary, code files not shown above will also be involved, but they will not be mentioned specifically later.

1.3 Basic Modules

This section mainly briefly describes some basic knowledge, concepts and systems commonly used in rendering systems, so as to provide a transition and entr or have a weak foundation. If you are a UE veteran, you can skip this section.

1.3.1 New Features of C++

This section briefly describes the new C++ features (C++11, C++14 and later versions) that are often involved in UE and rendering systems.

1.3.1.1 Lambda

C++ lambda is a feature of C++11, which is similar to closures and anonymous functions in scripting languages such as C# and Lua, but it is more complex an style of Native languages. It has several grammatical forms:

```
(1)     [ captures ] <tparams>(optional)(C++20) ( params ) specifiers exception attr -> retrequires(optional)(C++20) { body } [ captures ] ( params ) -> ret { body }
(2)
(3)     [ captures ] ( params ) { body } [ captures ]
(4)     { body }
```

Among them, the first one is the syntax supported by C++20, which is not used by UE yet. The other three are common forms. The most commonly used is to thread, such as FScene::AddPrimitive ((...) indicates that some codes are omitted, the same below):

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

void FScene::AddPrimitive(UPrimitiveComponent* Primitive) {

    (.....)

    FScene* Scene = this;

    TOptional<FTransform> PreviousTransform = FMotionVectorSimulation::Get().GetPreviousTransform(Primitive);

    ENQUEUE_RENDER_COMMAND(AddPrimitiveCommand)(
        [Params = MoveTemp(Params), Scene, PrimitiveSceneInfo, PreviousTransform = MoveTemp(PreviousTransform)](FRHICommandListImmediate& RHICmdL {

            FPrimitiveSceneProxy* SceneProxy = Params.PrimitiveSceneProxy;
            FScopeCycleCounter Context(SceneProxy->GetStatId());
            SceneProxy->SetTransform(Params.RenderMatrix, Params.WorldBounds, Params.LocalBounds, Params.AttachmentRootPosition);

            // Create any RenderThreadResources required. SceneProxy-
            >CreateRenderThreadResources();

            Scene->AddPrimitiveSceneInfo_RenderThread(PrimitiveSceneInfo, PreviousTransform);
        });
}
```

When generating a closure, there are multiple ways to capture variables in the current environment (scope): by value (directly put the variable into the capture before the variable and put it into the captures list). The above FScene::AddPrimitive uses the method of passing Lambda variables by value. Since the life cycle guaranteed by the coder, UE mostly uses the method of passing by value to prevent access to invalid memory.

For more detailed explanation, please refer to the official C++ website's description of Lambda:Lambda expressions .

1.3.1.2 Smart Pointer

Since C++11, the standard library has introduced a set of**smart pointers**:**unique pointer (unique_ptr)**,**shared pointer (shared_ptr)**,**weak pointer (weak_** programmers in memory allocation and tracking.

However, Unreal does not use this set of pointers directly, but implements its own set. In addition to the three mentioned above, UE can also add**shared refer** nullable shared pointers. Unreal Objects use a separate memory tracking system that is more suitable for game code, so these classes cannot be used at the sa
Their comparison and description are shown in the following table:

name	UE	C++	illustrate
Shared Pointers	TSharedPtr	shared_ptr	A shared pointer owns the object it refers to, indefinitely protecting that object from deletion, and eventually handling its deletion when no shared pointers or shared references refer to it. A shared pointer can be null, meaning it does not refer to any object. Any non-null shared pointer can generate a shared reference to the object it refers to.
Unique pointer	TUniquePtr	unique_ptr	A unique pointer will only explicitly own the object it refers to. There is only one unique pointer to a given resource, so unique pointers can transfer ownership, but they cannot be shared. Any attempt to copy a unique pointer will result in a compilation error. When a unique pointer goes out of scope, it automatically deletes the object it refers to.

name	UE	C++	illustrate
Weak Pointers	TWeakPtr	weak_ptr	The WeakPointer class is similar to a SharedPointer, but does not own the object it references, and therefore does not affect its lifetime. This property is useful because it breaks reference cycles, but it also means that a weak pointer can become null at any time without warning. Therefore, a weak pointer can generate a shared pointer to the object it references, ensuring that the programmer has safe temporary access to the object.
Shared References	TSharedRef	-	Shared references behave like shared pointers, that is, they own the object they reference. There is a difference in the case of null objects; a shared reference must always reference a non-null object. Shared pointers do not have this restriction, so a shared reference can always be converted to a shared pointer, and that shared pointer always references a valid object. Use shared references when you want to confirm that the referenced object is non-null, or to assert ownership of a shared object.

UE also provides a tool interface similar to C++ to build smart pointers better and faster:

name	UE	C++	illustrate
Construct a shared pointer from this	TSharedFromThis	enable_shared_from_this	Derive a class in which you add aAsSharedorShareThisFunction TSharedFromThis.This type of function allows you to get the object's TSharedRef.
Constructing a shared pointer	MakeShared, MakeShareable	make_shared	Creates a shared pointer from a regular C++ pointer.MakeSharedA new object instance and reference controller are allocated in a single block of memory, but requires the object to submit a public constructor. MakeShareableThis is less efficient, but will work even if the object's constructor is private. This allows you to own objects that you did not create yourself, and supports custom behavior when objects are deleted.
Static conversion	StaticCastSharedRef, StaticCastSharedPtr	-	Static casting utility function, typically used to cast down to derived types.
Fixed conversion	ConstCastSharedRef, ConstCastSharedPtr	-	Convertsconsta smart reference or smart pointer tomutablea smart reference or smart pointer, respectively.

In addition to providing basic functions such as memory management access and reference count tracking, the smart pointer library that comes with UE can a standard version in terms of efficiency and memory usage. In addition, it also provides thread-safe access modes:

- TSharedPtr<T, ESPMode::ThreadSafe>
- TSharedRef<T, ESPMode::ThreadSafe>
- TWeakPtr<T, ESPMode::ThreadSafe>
- TSharedFromThis<T, ESPMode::ThreadSafe>

However, since the thread-safe version relies on atomic reference counting, its performance is slightly slower than the non-thread-safe version, but its behavior

- Read and Copy are guaranteed to be thread-safe.
- Write and Reset must be synchronized to be safe.

These thread-safe smart pointers are widely used in the UE multi-threaded rendering architecture.

1.3.1.3 Delegate

A delegateis essentially a function type and representation, which facilitates the declaration, reference, and execution of specified member functions. The C+ delegates, but can achieve the effect of class delegation through obscure syntax.

Microsoft's built-in library implements thedelegate function. Similarly, due to the large number of delegate requirements and applications in UE, UE also imple There are three types of UE delegation:

- Single point entrustment
- Multicast delegation
    - event
- Dynamic Objects
    - UObject
    - Serializable

It is declared through a set of macros. The common declaration forms and corresponding function definitions are as follows:

Declaring Macros	Function definition or description
DECLARE_DELEGATE(DelegateName)	void Function()
DECLARE_DELEGATE_OneParam(DelegateName, Param1Type)	void Function(Param1)
DECLARE_DELEGATE_Params(DelegateName, Param1Type, Param2Type, ...)	void Function(Param1, Param2, ...)
DECLARE_DELEGATE_RetVal(RetValType, DelegateName)	Function()
DECLARE_DELEGATE_RetVal_OneParam(RetValType, DelegateName, Param1Type)	Function(Param1)
DECLARE_DELEGATE_RetVal_Params(RetValType, DelegateName, Param1Type, Param2Type, ...)	Function(Param1, Param2, ...)
DECLARE_MULTICAST_DELEGATE(_XXX)	Create a multicast delegate type (with parameters)
DECLARE_DYNAMIC_MULTICAST_DELEGATE()	Create a dynamic multicast delegate type (with parameters)

After the declaration, you can use the BindXXX and UnBind interfaces to bind and unbind the existing interfaces accordingly. For the bound delegate, you can

```
//Declare a delegate type
DECLARE_DELEGATE_OneParam(FOnEndCaptureDelegate, FRHICommandListImmediate*);


//Defining the delegate object
staticFOnEndCaptureDelegate GEndDelegates;


//Registration Entrustment
void RegisterCallbacks(FOnBeginCaptureDelegate InBeginDelegate, FOnEndCaptureDelegate InEndDelegate) {

	GEndDelegates = InEndDelegate;
}

//Execute the delegation (if binding exists)
void EndCapture(FRHICommandListImmediate* {		RHICommandList)

	if(GEndDelegates.IsBound()) {

		GEndDelegates.Execute(RHICommandList);
	}
}

//Unbinding
void UnregisterCallbacks() {

	GEndDelegates.Unbind();
}
```

UE's delegate implementation code is in TBaseDelegate:

```
// Engine\Source\Runtime\Core\Public\Delegates\DelegateSignatureImpl.inl

template <typename WrappedRetValType, typename... ParamTypes> class
TBaseDelegate: public FDelegateBase {

public:
	/** Type definition for return value type. */
	typedeftypename TUnwrapType<WrappedRetValType>::Type RetValType; typedef
	RetValTypeTFuncType(ParamTypes...);

	/** Type definition for the shared interface of delegate instance types compatible with this delegate class. */ typedefIBaseDelegateInstance<TFuncType>
	TDelegateInstanceInterface;
```

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

The implementation has a lot of source code, using templates and multiple inheritance, but in essence it also encapsulates objects, function pointers, etc.

1.3.1.4 Coding Standard

This section briefly describes the common coding standards and common sense that are officially recommended or mandatory by UE.

- **Naming convention**
  - Each word in a name (such as a type or variable) should have its first letter capitalized, and there are usually no underscores between words. For ex notlastMouseCoordinatesordelta_coordinates.
  - Type names are prefixed with an extra capital letter to distinguish them from variable names. For example,FSkinis a type name, andSkinisFS
    - Template classes are prefixed with T.
    - Classes that inheritUObjectfrom are prefixed with U.
    - Classes that inherit fromAActorare prefixed with A.
    - Classes that inheritSWidgetfrom are prefixed with S.
    - The prefix of interface class is I.
    - Enumerations are prefixed with E.
    - Boolean variables must be prefixed with b (egbPendingDestructionorbHasFadedIn).
    - Most other classes are prefixed with F, while some subsystems are prefixed with other letters.
    - Typedefs should be prefixed with whatever letter is appropriate for their type: F for structs,UobjectU for , and so on.
      - Typedefs of special template instantiations are no longer templates and should be prefixed accordingly, for example:

        ```
        typedefTArray<FMytype> FArrayOfMyTypes;
        ```

    - In C# the prefix is omitted.
    - In most cases, UnrealHeaderTool requires the correct prefix, so adding the prefix is important.
  - Types and variables are named as nouns.
  - Method names are verbs that describe the effect of the method or the return value that is not affected by the method.
  - Variable, method, and class names should be clear, concise, and descriptive. The larger the scope of the name, the more important it is to have ag abbreviations.
  - All functions that return a boolean should issue a true/false query, such asIsVisible()orShouldClearBuffer().
  - Procedures (functions without a return value) should use strongly conjugated verbs after Object. An exception is when the method's Object is the Object is understood from the context. Avoid beginning a method with "Handle" and "Process"; such verbs are ambiguous.
- **STL Whitelist**

  Although UE avoids using some STL libraries and implements its own code for reasons such as memory management and efficiency, the C++ standard i cross-platform valid modules have been added. Therefore, the official whitelist status of the following modules is maintained (allowed to use and will no
  - atomic
  - type_traits
  - initializer_list
  - regex
  - limits
  - cmath
- The declaration of a class should be from the user's perspective rather than the implementer's perspective, so it is usually necessary to declare the class's first, and then the private ones.

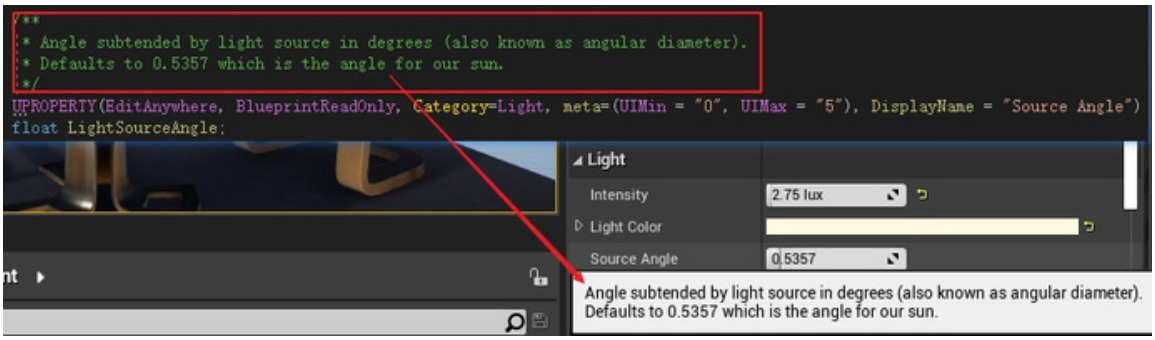
```
UCLASS()
classMyClass
{
public:
    UFUNCTION()
    void SetName(constFString& InName);
    UFUNCTION()
    FStringGetName()const;

private:
    void ProcessName_Internal(constFString& InName);
```

```
private:
	UPROPERTY()
	FString Name;
};
```

- Use const as much as possible, including parameters, variables, constants, function definitions, return values, etc.

```
void MyFunction(constTArray<Int32>& InArray, FThing& OutResult) {

	//This will not be modifiedInArray, but may be modifiedOutResult
}

void MyClass::MyFunction()const {

	//This code does not changeMyClassFor any member of , you can addconst
}

TArray<FString> StringArray; for(const
FString& :StringArray) {

	//The body of this loop does not modifyStringArray
}
```

- The code should be clearly and accurately commented. Specific comment formats can be used to generate tooltips for the editor using the automatic do



After the C++ component adds comments to the variables, their descriptions are captured by the UE compilation system and applied to the editor's pro

- **C++ New Syntax**
  - nullptr replaces the old NULL.
  - static_assert (static assertion)
  - override & final
  - Try to avoid using the auto keyword.
  - New traversal syntax

```
TMap<FString, int32> MyMap;

// Old style
for(autoIt = MyMap.CreateIterator(); It; ++It) {

	UE_LOG(LogCategory, Log, TEXT("Key: %s, Value: %d"), It.Key(), *It.Value());
}

// New style
for(TPair<FString, int32>& Kvp : MyMap) {

	UE_LOG(LogCategory, Log, TEXT("Key: %s, Value: %d"), *Kvp.Key, Kvp.Value);
}
```

  - New Enumeration

```
// Old enum
UENUM()
namespace EThing
{
	enum Type
	{
		Thing1,
		Thing2
	};
}

// New enum
UENUM()
```

```
enum classEThing: uint8 {

    Thing1,
    Thing2
}
```

- Move semantics. All UE4 built-in containers support move semantics and are usedMoveTempinstead of C++std::move.
- Initial values of class member variables.

```
UCLASS()
classUMyClass: public UObject {

    GENERATED_BODY()

public:

    UPROPERTY()
    floatWidth =11.5f;

    UPROPERTY()
    FString Name = TEXT("Earl Grey");
};
```

- Third-party library specific formats.

```
// @third party code - BEGIN PhysX
#include<physx.h>
// @third party code - END PhysX

// @third party code - BEGIN MSDN SetThreadName
// [http://msdn.microsoft.com/en-us/library/xcb2z8hs.aspx] // Used to set the
thread name in the debugger
. . .
//@third party code - END MSDN SetThreadName
```

For more complete coding standards, please refer to the UE official document:Coding Standard .

1.3.2 Container

Unreal Engine implements a set of basic containers and algorithm libraries, and does not use the STL standard library. However, some of them can be found in shown in the following table:

Container Name	UE4	STL	Analysis
Arrays	TArray	vector	A continuous array can add, delete, and sort elements. It is more powerful and convenient than the vector of stl. When adding an array, the memory will be reallocated as needed, and the length of the array will grow according to a certain strategy (see below for details on the growth strategy).
Tuple	TTuple	tuple	Stores a set of data. Its length cannot be changed after construction, but the element types can be different.
Linked List	TList	forward_list	One-way linked list, the operation and underlying implementation are similar to stl.
Doubly Linked List	TDoubleLinkedList	list	Doubly linked list, the operation and underlying implementation are similar to stl.
Mapping Table	TMap	map	The key-value one-to-one mapping table is ordered, implemented with TSet at the bottom layer, and stores a set of key-value pairing arrays.
Multi-value mapping table	TMultiMap	unordered_map	A key-to-multi-value mapping table, ordered, with the underlying implementation being similar to TMap, but adding elements will not delete existing values. The difference is that stl's unordered_map is unordered and uses a hash table for storage and indexing.
Ordered Mapping Table	TSortedMap	map	A key-value mapping table, ordered, implemented with a TArray that is sorted by key at the bottom layer, and stores a set of key-value pairing arrays. The memory occupied is half that of TMap, but the complexity of

Container Name	UE4	STL	Analysis
			adding and deleting elements is O(n), and the complexity of searching is O(Log n).
gather	TSet	set	A collection of keys, and the keys cannot overlap. The underlying The implementation uses TSparseArray, and the elements are stored in buckets. The number of buckets depends on the size of the elements, and the stored elements are linked with Hash values.
Hash Table	FHashTable	hash_map	Often used to index other arrays. Get the hash value of the specified ID based on other hash functions, and then store and search for elements in other arrays.
queue	TQueue	queue	Unbounded non-intrusive queue, implemented using lock-free linked lists. Supports multiple producers-single consumers (MPSC) and single producer-single consumers (SPSC) modes, both of which are thread-safe. Often used for data transmission and access between multiple threads.
Circular Queue	TCircularQueue	-	A lock-free circular queue, first-in-first-out, implemented using a circular array (TCircularBuffer), thread-safe in single producer-single consumer (SPSC) mode.
Looping through arrays	TCircularBuffer	-	The underlying implementation is TArray, which is boundless and requires a specified capacity when it is created, and the capacity cannot be changed later.
String	FString	string	A string whose content and size can be changed dynamically. It is similar to stl's string, but has more complete functions. It is implemented using TArray at the bottom layer. In addition, it also has optimized versions FText and FName.

The above correspondence between UE and STL is only considered from the perspective of the provided calling interface (user), but the actual underlying imp different. For example, let's analyze the element size growth strategy of TArray in detail (some macros and branches are simplified):

```
// Array.h

void TArray::ResizeGrow(SizeType OldNum) {

	ArrayMax = AllocatorInstance.CalculateSlackGrow(ArrayNum, ArrayMax,sizeof(ElementType));
	AllocatorInstance.ResizeAllocation(OldNum, ArrayMax,sizeof(ElementType));
}


// ContainerAllocationPolicies.h

SizeTypeCalculateSlackGrow(SizeType NumElements, SizeType NumAllocatedElements, SIZE_T NumBytesPerElement)const {

	returnDefaultCalculateSlackGrow(NumElements, NumAllocatedElements, NumBytesPerElement,true, Alignment);
}

template <typename SizeType>
SizeTypeDefaultCalculateSlackGrow(SizeType NumElements, SizeType NumAllocatedElements, SIZE_T BytesPerElement,boolbAllowQuantize, uint32 Align {

	constSIZE_T FirstGrow =4; constSIZE_T
	ConstantGrow =16;

	SizeType Retval;
	checkSlow(NumElements > NumAllocatedElements && NumElements >0);

	SIZE_T Grow = FirstGrow;// this is the amount for the first alloc

	if(NumAllocatedElements || SIZE_T(NumElements) > Grow) {

		// Allocate slack for the array proportional to its size.
		Grow = SIZE_T(NumElements) +3* SIZE_T(NumElements) /8+ ConstantGrow;
	}

	if(bAllowQuantize)
	{
		Retval = (SizeType)(FMemory::QuantizeSize(Grow * BytesPerElement, Alignment) / BytesPerElement);
	}
```

```
    else
    {
        Retval    =  (SizeType)Grow;
    }
    // NumElements and MaxElements are stored in 32 bit signed integers so we must be careful not to overflow here. if(NumElements > Retval) {


        Retval = TNumericLimits<SizeType>::Max();
    }

    returnRetval;
}
```

From the above, we can see the growth strategy of TArray memory length: when it is first allocated, it will grow by at least 4 elements; then it will grow propor and a fixed growth of 16. Then it will be adjusted to a multiple of 8 and memory alignment. It can be seen that its memory growth strategy is quite different fr

The above is just a small list of commonly used UE containers. There are dozens of UE containers. The complete list is inContainers .

1.3.3 Mathematical Library

Unreal Engine implements a set of mathematical libraries, the code is in the Engine\Source\Runtime\Core\Public\Math directory. The following lists the comm

type	name	Analysis
FBox	Bounding Box	Axis-parallel three-dimensional bounding boxes are often used for bounding volumes, collision volumes, visibility determination, etc.
FBoxSphereBounds	Sphere-Cube Bounding Box	Contains the bounding box data of a sphere and an axis-parallel cube, each of which is used for different purposes, such as the sphere is used for scene traversal acceleration structure, and the cube is used for collision detection. It is the type of bounding box for most visible objects.
FColor	Gamma space color	Stores the color values of RGBA8888 4 channels, which are in Gamma space and can be converted from FLinearColor in linear space.
FLinearColor	Linear Space Color	Stores the color values of RGBA4 channels. The precision of each channel is 32-bit floating point value. They are in linear space and can be converted from FColor in Gamma space.
FCapsuleShape	Capsule	Stores the data of two circles and a cylinder. The two circles are located at both ends of the cylinder, thus forming a capsule. Commonly used for physical collision capsules.
FInterpCurve	Interpolation curve	The template class stores a series of key frames and provides interfaces such as interpolation and derivative to facilitate external operation of curves.
FMatrix	4x4 Matrix	Contains 16 floating-point values used to store spatial transformations, such as rigid body transformations such as rotation, scaling, translation, and non-rigid body transformations such as shear.
FMatrix2x2	2x2 Matrix	Contains a 2x2 matrix for transformations in 2D space.
FQuat	Quaternions	Stores quaternion 4D data, which associates the rotation axis and rotation angle. Commonly used for operations such as rotation and rotation interpolation.
FPlane	flat	A plane in 3D space described by a point and an additional W value.
FRay	ray	Describe a ray in three-dimensional space using a point and a vector.
FRotationMatrix	Rotation Matrix	The rotation matrix without translation is inherited from the rotation matrix with translation FRotationTranslationMatrix.
FRotator	Rotator	Providing a rotation structure described by Pitch, Yaw, and Roll, which is more in line with the rotation description method of human perspective, making it easier for the logic layer to control the rotation of objects (such as cameras).
FSphere	Sphere	A sphere in three-dimensional space described by a point and a radius.

type	name	Analysis
FMath	Mathematics Toolbox	A collection of cross-platform, precision-compatible mathematical constant definitions and utility functions.
FVector	3D Vector	A vector in three-dimensional space, with each dimension being a floating point value, can also be used to describe a point.
FVector2D	2D Vector	Two-dimensional vectors can also describe 2D points.
FVector4	4D Vector	It stores vectors of four dimensions XYZW and can be used for homogeneous coordinates, projection transformation, etc.

In addition to the commonly used types listed above, the UE math library also provides floating-point numbers of different precisions, random numbers, boun management nodes, auxiliary classes and toolboxes derived from basic types, and other modules. For a complete list of math libraries, see the UE source code Math .

It is worth mentioning that UE provides several optimized versions of vector SIMD instructions, and different macros can be defined to enable the correspondi

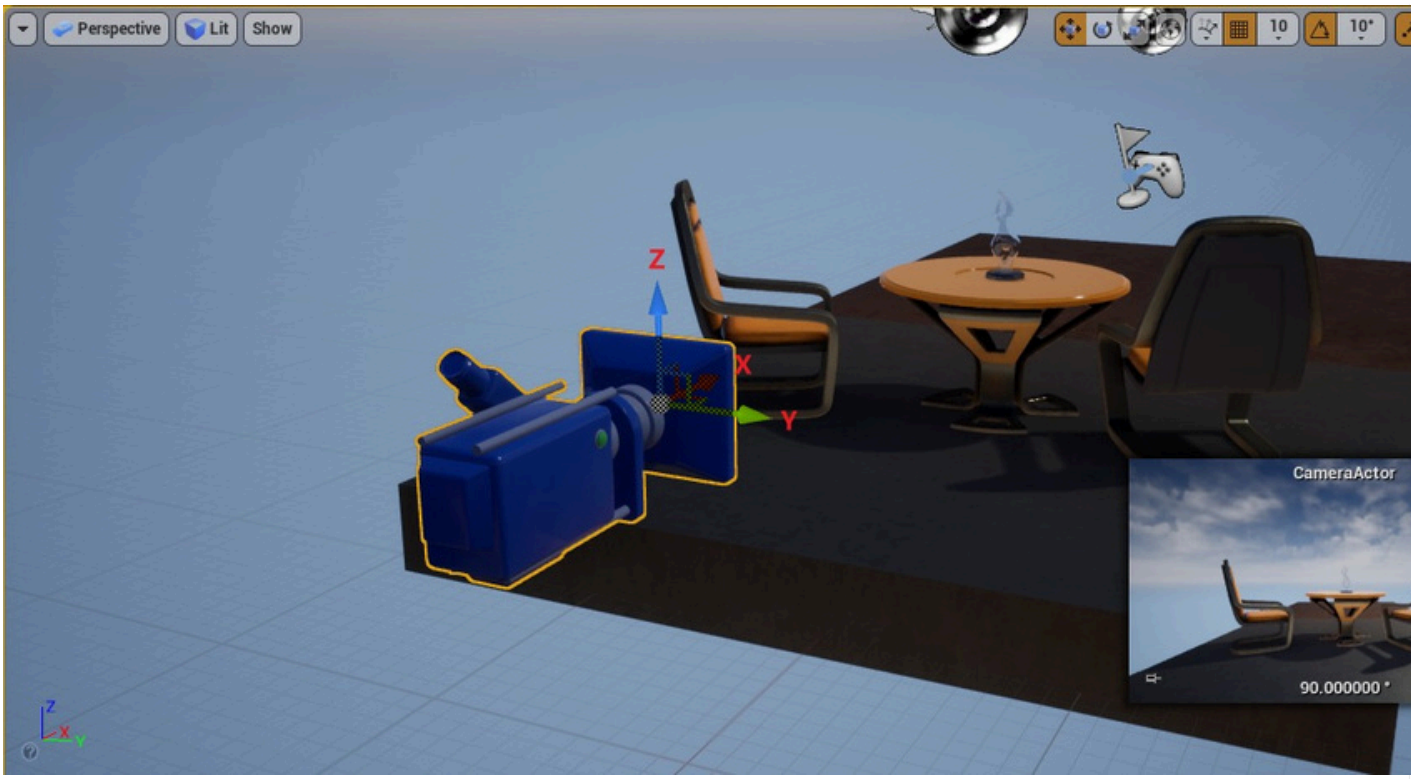
```
// Engine\Source\Runtime\Core\Public\Math\VectorRegister.h

// Platform specific vector intrinsics include.
#if WITH_DIRECTXMATH
	#define	SIMD_ALIGNMENT (16)
	#include	"Math/UnrealMathDirectX.h"
#elifPLATFORM_ENABLE_VECTORINTRINSICS
	#define	SIMD_ALIGNMENT (16)
	#include	"Math/UnrealMathSSE.h"
#elifPLATFORM_ENABLE_VECTORINTRINSICS_NEON
	#define	SIMD_ALIGNMENT (16)
	#include	"Math/UnrealMathNeon.h"
#else
	#define	SIMD_ALIGNMENT (4)
	#include	"Math/UnrealMathFPU.h"
#endif
```

From the above code, we can see that UE supports DirectX built-in library, Arm Neon instructions, SSE instructions, FPU and other versions.

> Designed by Arm,**Neon is a set of single instruction multiple data (SIMD) architecture extension technologies suitable for Arm Cortex-A and Cortex-SSE**(Stream SIMD Extensions) is an instruction set designed by Intel and first introduced in its computer chip Pentium3. It is an extended instruction set afte architectures. Currently, there are SSE2, SSE3, SSSE3, SSE4 and other instruction sets.
> **FPU**(Floating-point unit) is a floating-point calculation unit. It is part of the hardware structure of the CPU core and is the core unit of the CPU for processin

1.3.4 Coordinate Space

UE uses a left-handed coordinate system (same as DirectX, but OpenGL uses a right-handed coordinate system). In the default level (newly created scene) view is to the back of the view; but when you drag a CameraActor into the scene, the default view of the camera is Z axis upward, Y is to the right, and X is to the fro coordinate system is different from many other engines. You may not be used to it at first, but you won't feel hindered after using it for a long time.



The orientation of the default coordinate system under the UE's camera view is shown in the figure.

The coordinate space of UE is basically the same as the conversion of the 3D rendering pipeline, but there are also some unique concepts, as detailed in the fo

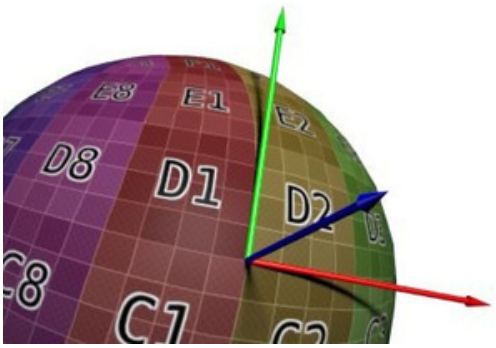
UE coordinate space	English name	Aliases	Analysis
Tangent	Tangent Space	-	Orthogonal (biased after interpolation), may be left-handed or right-handed. TangentToLocal contains only rotation, not position translation information, so it is OrthoNormal (the transposed matrix is also the inverse matrix).
Local	Local Space	ObjectSpace	Orthogonal, can be left-right or right-handed (meaning related to triangle clipping, need to be adjusted), LocalToWorld contains information such as rotation, scale, translation, etc. Scale may be negative, used for animation, wind direction simulation, etc.
World	World Space	-	The WorldToView matrix only contains rotation and translation, not scaling.
TranslatedWorld	World Space with Translation	-	TranslatedWorld=World+PreViewTranslation, PreViewTranslation is the reverse position of the Camera position, and TranslatedWorld is equivalent to the World matrix that does not contain the camera translation information. It is widely used in calculations such as BasePass, bone skinning, particle effects, hair, and noise reduction.
View	View Space	CameraSpace	View space is a coordinate space with the origin at the center of the camera's near clipping plane. The ViewToClip matrix includes x and y scaling, but no translation. It also scales and translates the depth value z, and usually also applies a projection matrix to transform to homogeneous projection space.
Clip	Clip Space	HomogeniousCoordinates, PostProjectionSpace, ProjectionSpace	Once the perspective projection matrix is applied, we can transform to homogeneous clip space. Note that W in clip space is equal to Z in view space.

UE coordinate space	English name	Aliases	Analysis
Screen	Screen Space	NormalizedDeviceCoordinates	After applying perspective division to the coordinates in clip space (xyz divided by the w component), the coordinates in screen space can be obtained. The horizontal coordinates in screen space are [-1, 1] from left to right, the vertical coordinates are [-1, 1] from bottom to top, and the depth is [0, 1] from near to far (but the depth of OpenGL RHI is [-1, 1]).
Viewport	Viewport Space	ViewportCoordinates (viewport coordinates), WindowCoordinates (window coordinates)	Map the screen coordinates to the pixel coordinates of the window. The horizontal coordinates are [0, width-1] from left to right, and the vertical coordinates are [0, height-1] from top to bottom (note that the vertical coordinates of the screen space increase from bottom to top).

In UE's C++ interface or Shader variables, there are widespread transformations from one space to another. Their names are X To Y (X and Y are both spatial no

- LocalToWorld
- LocalToView
- TangentToWorld
- TangentToView
- WorldToScreen
- WorldToLocal
- WorldToTangent
- .....

Tangent space is different from local space (model space). It uses the normal and tangent of each vertex (or pixel) as axes to construct an orthogonal coordina



Schematic diagram of the tangent space on the model's vertices. Each vertex has its own tangent space.

$$\mathbf{T} = \left(\frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u}\right)$$
$$\mathbf{B} = \mathbf{N} \times \mathbf{T}$$
$$\mathbf{N} = \left(\frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u}\right) \times \left(\frac{\partial x}{\partial v}, \frac{\partial y}{\partial v}, \frac{\partial z}{\partial v}\right)$$

A common formula for constructing three axes (tangent T, bitangent B, normal N) of an orthogonal tangent space from a vertex.

Why do we need tangent space when we already have local space?

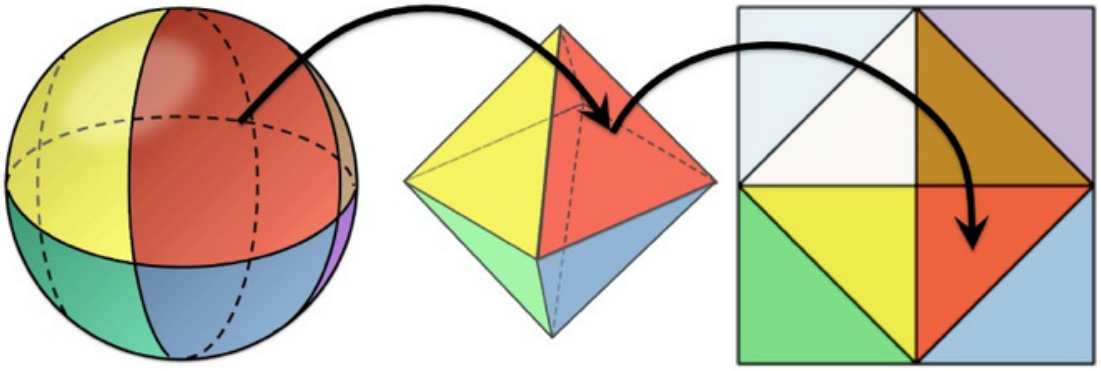
This can be answered from the role of tangent space. To sum up, there are mainly the following points:

- **Supports all kinds of animations**,including skinned skeletal animation, procedural animation, vertex animation, UV animation, etc. Since the normals o operation is performed, if the normals are not corrected in real time in the tangent space, incorrect lighting results will be generated.
- **Supports tangent space calculation of lighting.**You only need to convert the light source direction L and the line of sight V to the tangent space, and normal sampling to perform the lighting calculation and obtain the correct lighting result.
- **Normal maps can be reused**.Tangent space normal maps record relative normal information, which means that even if the normal map is applied to a relatively reasonable lighting result can be obtained. The same model can reuse normal maps multiple times, and different models can also reuse the sa model, only one map is needed to be used on all six faces.

- **Compressible**.Since the Z direction of the normal of the tangent space normal map is always toward the positive direction of the Z axis, the normal ma deduce the Z direction.

The compression of normal textures was mentioned above. By the way, the compression of unit vectors, which is widely used in the UE Shader layer, is also me

Zina H. Cigolleet al. published a paperSurvey of Efficient Representations for Independent Unit Vectors in 2014 , which proposed a method to compress thre dimensions. The compression process is to first map the unit sphere into an octahedron, and then project it into a two-dimensional cube, as shown in the figu



The decompression process is just the opposite. The UE shader code clearly records the specific process of compression and decompression:

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

//Compression: From3After the unit vector of the dimension is converted to the
octahedron, it returns2Dimensional results. float2UnitVectorToOctahedron( float3 N ) {

	N.xy /= dot(1,abs(N) ); if( Nz <=0)		//Convert unit sphere to octahedron

	{
		N.xy = (1-abs(N.yx) ) * ( N.xy >=0? float2(1,1) : float2(-1,-1) );
	}
	returnN.xy;
}

//Unzip: From2dimensional octahedron vector conversion to3
-dimensional unit vector. float3OctahedronToUnitVector( float2 Oct ) {

	float3 N = float3( Oct,1- dot(1,abs(Oct) ) ); if( Nz <0)

	{
		N.xy = (1-abs(N.yx) ) * ( N.xy >=0? float2(1,1) : float2(-1,-1) );
	}
	returnnormalize(N);
}
```

Since the compressed vectors mentioned above are required to be of unit length, only vectors such as the incident light direction, line of sight, and normal can information such as color and light intensity cannot be accurately compressed.

In addition, UE also supports the encoding and decoding of half-octagonal surfaces:

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

// 3dimensional unit vector compressed into a half-octahedron2Dimensional vector
float2UnitVectorToHemiOctahedron( float3 N ) {

	N.xy /= dot(1,abs(N) );
	returnfloat2( Nx + Ny, Nx - Ny );
}

//Semi-octahedral2The vector is unpacked into3dimensional unit vector
float3HemiOctahedronToUnitVector( float2 Oct ) {

	Oct = float2( Oct.x + Oct.y, Oct.x - Oct.y ) *0.5; float3 N = float3( Oct,1-
	dot(1,abs(Oct) ) ); returnnormalize(N);

}
```

1.3.5 Basic macro definitions

In order to be compatible with the differences between various platforms and various compiler options, UE defines a variety of macro definitions, which are m Build.h files:

```
// Engine\Intermediate\Build\Win64\UE4Editor\Development\Launch\Definitions.h

#defineIS_PROGRAM 0
```

```
#define UE_EDITOR 1
#define ENABLE_PGO_PROFILE 0
#define USE_VORBIS_FOR_STREAMING 1
#define USE_XMA2_FOR_STREAMING 1
#define WITH_DEV_AUTOMATION_TESTS 1
#define WITH_PERF_AUTOMATION_TESTS 1
#define UNICODE 1
#define _UNICODE 1
#define __UNREAL__ 1
#define IS_MONOLITHIC 0
#define WITH_ENGINE 1
#define WITH_UNREAL_DEVELOPER_TOOLS 1
#define WITH_APPLICATION_CORE 1
#define WITH_COREUOBJECT 1
#define USE_STATS_WITHOUT_ENGINE 0
#define WITH_PLUGIN_SUPPORT 0
#define WITH_ACCESSIBILITY 1
#define WITH_PERFCOUNTERS 1
#define USE_LOGGING_IN_SHIPPING 0
#define WITH_LOGGING_TO_MEMORY 0
#define USE_CACHE_FREED_OS_ALLOCS 1
#define USE_CHECKS_IN_SHIPPING 0
#define WITH_EDITOR 1
#define WITH_SERVER_CODE 1
#define WITH_PUSH_MODEL 0
#define WITH_CEF3 1
#define WITH_LIVE_CODING 1
#define WITH_XGE_CONTROLLER 1
#define UBT_MODULE_MANIFEST "UE4Editor.modules"
#define UBT_MODULE_MANIFEST_DEBUGGAME "UE4Editor-Win64-DebugGame.modules"
#define UBT_COMPILED_PLATFORM Win64
#define UBT_COMPILED_TARGET Editor
#define UE_APP_NAME"UE4Editor"
#define NDIS_MINIPORT_MAJOR_VERSION 0
#define WIN32 1
#define _WIN32_WINNT 0x0601
#define WINVER 0x0601
#define PLATFORM_WINDOWS 1
#define PLATFORM_MICROSOFT 1
#define OVERRIDE_PLATFORM_HEADER_NAME Windows
#define RHI_RAYTRACING 1
#define NDEBUG 1
#define UE_BUILD_DEVELOPMENT 1
#define
#define UE_IS_ENGINE_MODULE 1
#define WITH_LAUNCHERCHECK 0
#define UE_BUILD_DEVELOPMENT_WITH_DEBUGGAME 0
#define UE_ENABLE_ICU 1
#define WITH_VS_PERF_PROFILER 0
#define WITH_DIRECTXMATH 0
#define WITH_MALLOC_STOMP 1
#define CORE_API DLLIMPORT
#define TRACELOG_API DLLIMPORT
#define
#define COREUOBJECT_API DLLIMPORT
#define INCLUDE_CHAOS 0
#define WITH_PHYSX 1
#define WITH_CHAOS 0
#define WITH_CHAOS_CLOTHING 0
#define WITH_CHAOS_NEEDS_TO_BE_FIXED 0
#define PHYSICS_INTERFACE_PHYSX 1
#define WITH_APEX 1
#define WITH_APEX_CLOTHING 1
#define WITH_CLOTH_COLLISION_DETECTION 1
#define WITH_PHYSX_COOKING 1
#define WITH_NVCLOTH 1
#define WITH_CUSTOM_SQ_STRUCTURE 0
#define WITH_IMMEDIATE_PHYSX 0
#define GPUPARTICLE_LOCAL_VF_ONLY 0
#define ENGINE_API DLLIMPORT
#define NETCORE_API DLLIMPORT
#define
#define APPLICATIONCORE_API DLLIMPORT
#define DDPI_EXTRA_SHADERPLATFORMS SP_XXX=32,
#defineDDPI_SHADER_PLATFORM_NAME_MAP { TEXT("XXX"), SP_XXX },
#define RHI_API DLLIMPORT
#define JSON_API DLLIMPORT
#define WITH_FREETYPE 1
#define SLATECORE_API DLLIMPORT
#define INPUTCORE_API DLLIMPORT
#define SLATE_API DLLIMPORT
#define WITH_UNREALPNG 1
#define WITH_UNREALJPEG 1
#define WITH_UNREALEXR 1
#define IMAGEWRAPPER_API DLLIMPORT
#define MESSAGING_API DLLIMPORT
#define MESSAGINGCOMMON_API DLLIMPORT
#define RENDERCORE_API DLLIMPORT
#define
#define ANALYTICSET_API DLLIMPORT
#define ANALYTICS_API DLLIMPORT
```

```
#define SOCKETS_PACKAGE 1
#define SOCKETS_API DLLIMPORT
#define ASSETREGISTRY_API DLLIMPORT
#define ENGINEMESSAGES_API DLLIMPORT
#define ENGINESETTINGS_API DLLIMPORT
#define SYNTHBENCHMARK_API DLLIMPORT
#define RENDERER_API DLLIMPORT
#define GAMEPLAYTAGS_API DLLIMPORT
#define PACKETHANDLER_API DLLIMPORT
#define RELIABILITYHANDLERCOMPONENT_API DLLIMPORT
#define AUDIOPLATFORMCONFIGURATION_API DLLIMPORT
#define MESHDESCRIPTION_API DLLIMPORT
#define STATICMESHDESCRIPTION_API DLLIMPORT
#define PAKFILE_API DLLIMPORT
#define RSA_API DLLIMPORT
#define NETWORKREPLAYSTREAMING_API DLLIMPORT


// Engine\Source\Runtime\Core\Public\Misc\Build.h

#ifndefUE_BUILD_DEBUG
    #defineUE_BUILD_DEBUG 0
#endif
#ifndefUE_BUILD_DEVELOPMENT
    #defineUE_BUILD_DEVELOPMENT 0
#endif
#ifndefUE_BUILD_TEST
    #defineUE_BUILD_TEST 0
#endif
#ifndefUE_BUILD_SHIPPING
    #defineUE_BUILD_SHIPPING 0
#endif
#ifndefUE_GAME
    #defineUE_GAME 0
#endif
#ifndefUE_EDITOR
    #defineUE_EDITOR 0
#endif
#ifndefUE_BUILD_SHIPPING_WITH_EDITOR
    #defineUE_BUILD_SHIPPING_WITH_EDITOR 0
#endif
#ifndefUE_BUILD_DOCS
    #defineUE_BUILD_DOCS 0
#endif

(.....)
```

The common basic macros and their descriptions are as follows:

Macro Name	Analysis	default value
UE_EDITOR	Whether the current program is an editor, the most commonly used	1
WITH_ENGINE	Whether to enable the engine. If not, it is similar to SDK that only provides basic APIs and many modules will not work properly.	1
WITH_EDITOR	Whether to enable the editor, similar to UE_EDITOR.	1
WIN32	Whether it is a win32 program.	1
PLATFORM_WINDOWS	Whether the operating platform is Windows.	1
UE_BUILD_DEBUG	Debug build mode.	0
UE_BUILD_DEVELOPMENT	Developer build mode.	1
UE_BUILD_SHIPPING	Release build mode.	0
UE_GAME	Game Build Mode.	0
UE_EDITOR	Editor build mode.	0
UE_BUILD_DEVELOPMENT_WITH_DEBUGGAME	Developer build mode with game debugging.	0
UE_BUILD_SHIPPING_WITH_EDITOR	Release build mode with editor.	0

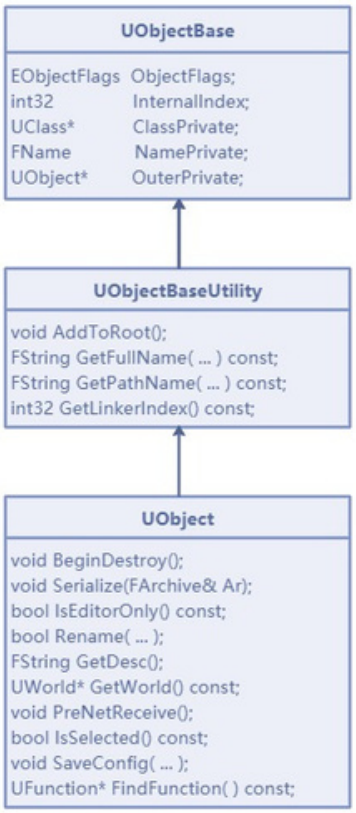
Macro Name	Analysis	default value
UE_BUILD_DOCS	Documentation build mode.	0
RHI_RAYTRACING	Whether to enable ray tracing	1

1.4 Engine Module

This section will go over the basic system and concepts of UE so that readers who are not familiar with UE can have a general understanding and better enter

1.4.1 Object, Actor, ActorComponent

UObject is the base class of all object types in UE. It inherits from UObjectBaseUtility, which in turn inherits from UObjectBase. It provides metadata, reflection serialization, some editor information, object creation and destruction, event callback and other functions. The specific type of the subclass is determined by th relationship is as follows:



AActor is the main and most important concept and type in the UE system. It inherits from UObject and is the base class of all objects that can be placed in th GameObject of the Unity engine. It provides network synchronization (Replication), creation and destruction of objects, frame update (Tick), component opera transformation and other functions. AActor objects can be nested AActor objects, and are supported by the following interfaces:

```
// Engine\Source\Runtime\Engine\Classes\GameFramework\Actor.h

void AttachToActor(AActor* ParentActor, ... ); void AttachToComponent
(USceneComponent* Parent, ... );
```

The above two interfaces are actually equivalent, because in fact the implementation code of AActor::AttachToActor also calls the RootComponent::AttachToCo

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

void AActor::AttachToActor(AActor* ParentActor,constFAttachmentTransformRules& AttachmentRules, FName SocketName) {

    if(RootComponent && ParentActor) {

        USceneComponent* ParentDefaultAttachComponent = ParentActor->GetDefaultAttachComponent(); if
        (ParentDefaultAttachComponent) {

            RootComponent->AttachToComponent(ParentDefaultAttachComponent, AttachmentRules, SocketName);
        }
```

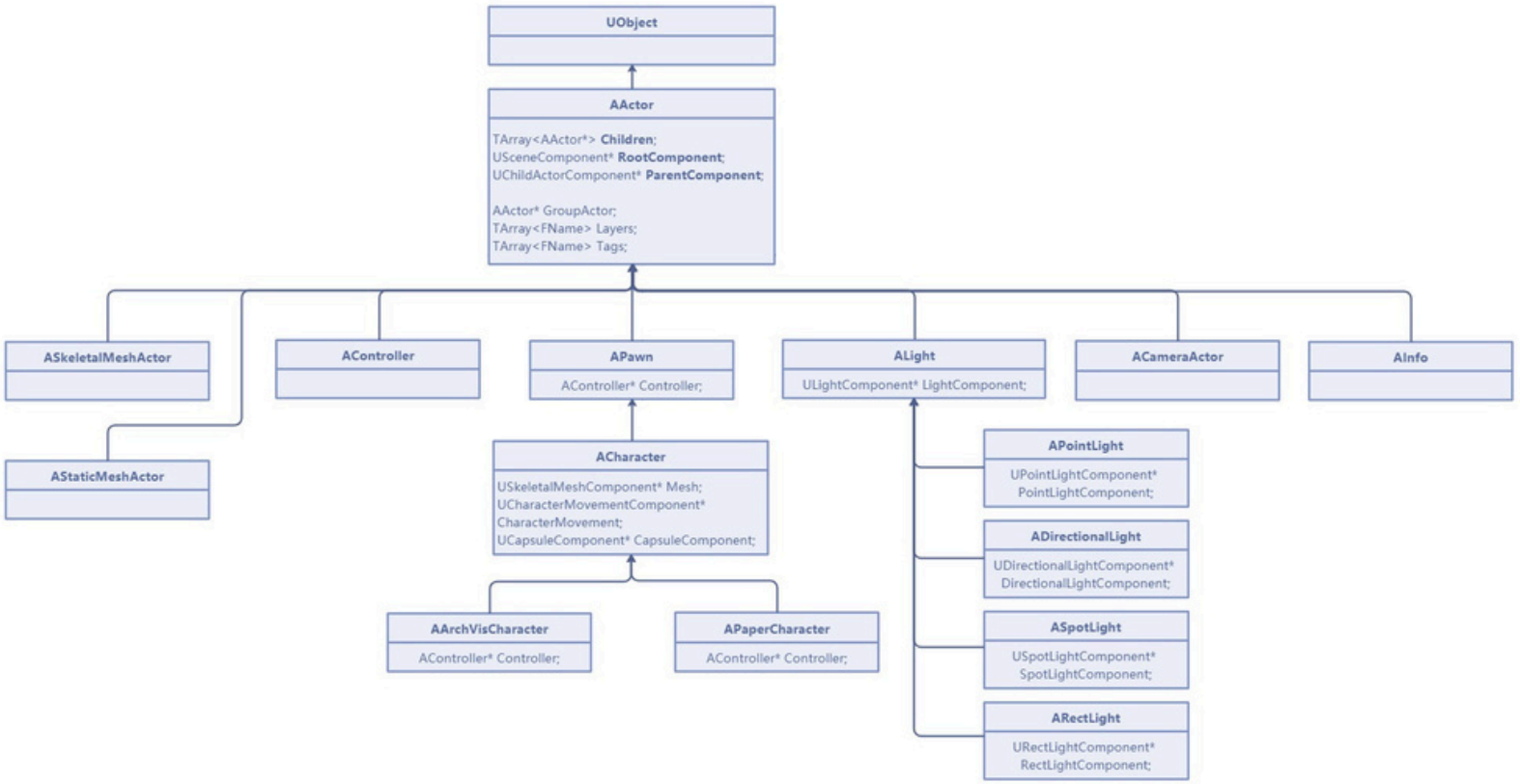
```
    }
}
```

This means that the Actor itself does not have nesting capabilities, but it can be achieved through the RootSceneComponent, which has a one-to-one relation

Common subclasses that inherit from Actor are:

- ASkeletalMeshActor: Skinned skeleton body, used to render dynamic models with skeletal skinning.
- AStaticMeshActor: static model.
- ACameraActor: Camera object.
- APlayerCameraManager: Camera manager, which manages all camera (ACameraActor) instances in the current world.
- ALight: Light object, from which point light source (APointLight), parallel light (ADirectionalLight), spotlight (ASpotLight), rectangular light (ARectLight) a
- AReflectionCapture: Reflection capturer, used to generate environment maps offline.
- AController: character controller. It also derives subclasses such as AAIController and APlayerController.
- APawn: describes dynamic characters or objects with AI. Its subclasses include ACharacter, ADefaultPawn, AWheeledVehicle, etc.
- AMaterialInstanceActor: Material instance body.
- ALightmassPortal: Global illumination portal, used to accelerate and improve offline global lighting efficiency and effects.
- AInfo: The base class for configuration information classes. Common subclasses that inherit from it include AWorldSettings, AGameModeBase, AAtmosp
- .....

The above only lists some of the subclasses of AActor. It can be seen that some of them can be put into the level, but some cannot be put into the level directl follows:

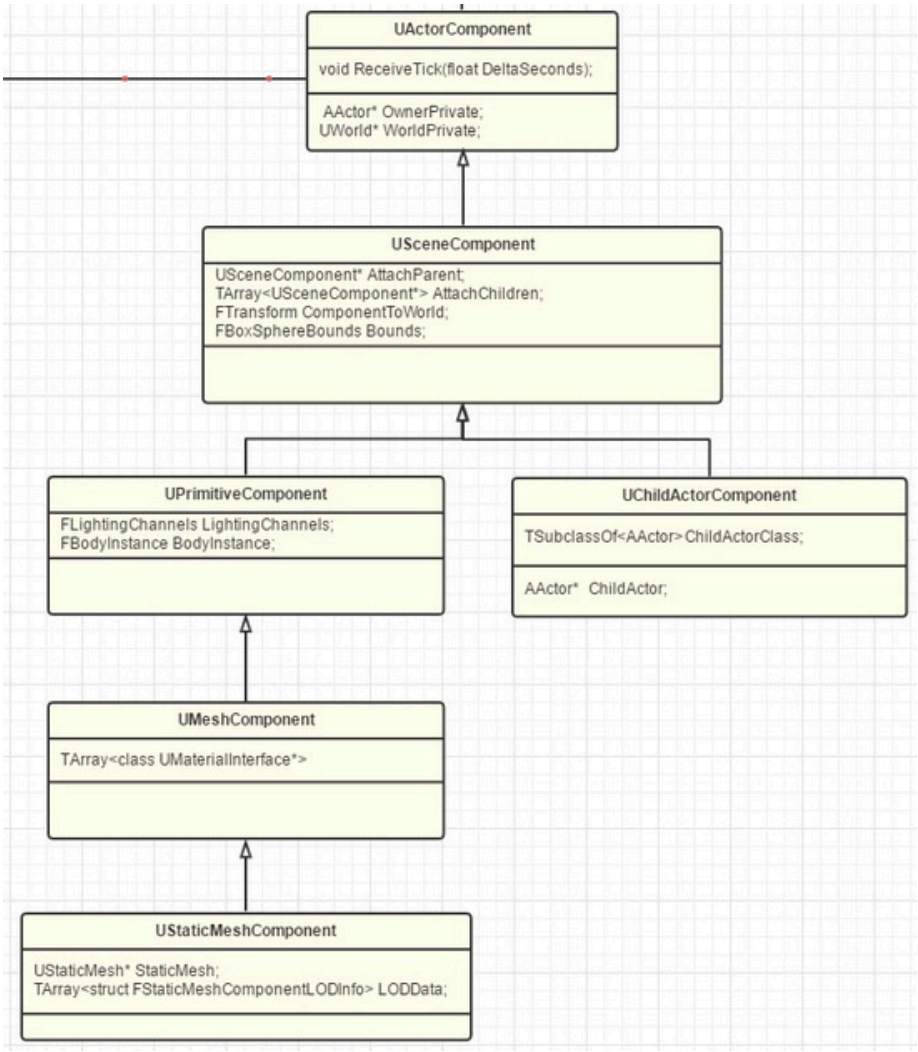


UActorComponent inherits from UObject and the interface IInterface_AssetUserData. It is the base class of all component types and can be added to an AActo more intuitively that an Actor can be regarded as a container containing a series of components. The functional characteristics and properties of an Actor are m attached to it.

The main commonly used UActorComponent subcomponent types are:

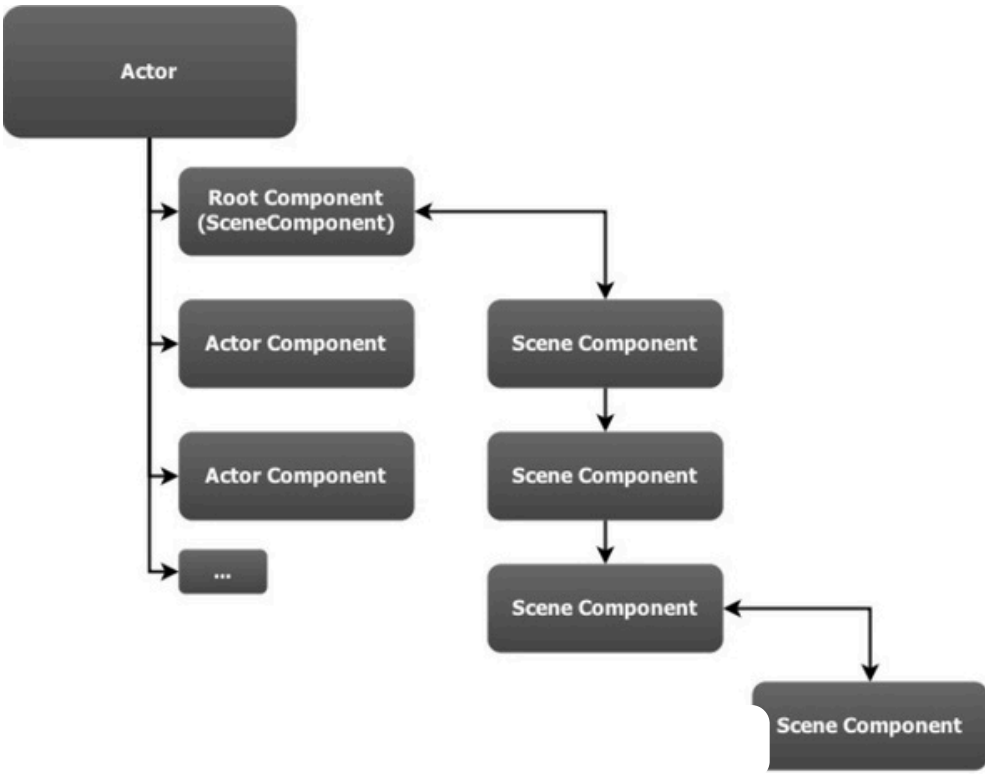
- USceneComponent: SceneComponents are ActorComponents that have transforms. The transform is the position in the scene, defined by position, rotat attached to each other in a hierarchy. The Actor's position, rotation, and scale are taken from the SceneComponent at the root of the hierarchy . UPrimitiveComponent: Inherits
- from SceneComponent, is the base class for all visible (renderable, such as meshes or particle systems) objects, and also and other functions.
- UMeshComponent: Inherited from UPrimitiveComponent, the base class for all renderable triangle mesh collections (static models, dynamic models, pro
- UStaticMeshComponent: Inherited from UMeshComponent, it is the geometry of the static mesh and is often used to create UStaticMesh instances.
- USkinnedMeshComponent: Inherits from UMeshComponent, a component that supports skinned mesh rendering, providing interfaces such as mesh, bo
- USkeletalMeshComponent: Inherited from USkinnedMeshComponent, usually used to create instances of USkeletalMesh resources with animation.

Their inheritance relationship is as follows:



All Actors that can be placed in a level have a Root Component (a type of Scene Component), which can be any subclass of a Scene Component. The Scene Co
scale of the Actor in the world, and these properties affect all of the Actor's child objects.

Even an empty Actor has a "Default Scene Root" object, which is the simplest scene component. During the editor operation phase, when we place a new scene root object of the Actor will be replaced.



Actor, RootComponent, SceneComponent, ActorComponent hierarchical nesting diagram.

1.4.2 Level, World, WorldContext, Engine

ULevel is the level of UE, a collection of objects in the scene, storing a series of Actors, including visible objects (such as meshes, lights, special effects, etc.) and blueprints, level configurations, navigation data, etc.).

UWorld is a container of ULevel, which truly represents a scene, because ULevel must be placed in UWorld to display its content. Each UWorld instance must c may also contain several streaming levels (Streaming Level, optional, non-essential, can be dynamically loaded and unloaded on demand). In addition to level GameInstance, AISystem, FXSystem, NavigationSystem, PhysicScene, TimerManager and other information. It has the following types:

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

namespace    EWorldType
{
	enum Type
	{
		None,

		Game, Editor,
		PIE,
		EditorPreview,
		GamePreview,
		GameRPC,


		Inactive
	};
}
```

Common WorldTypes include Game, Editor, Editor Play (PIE), and Preview Mode (EditorPreview, GamePreview), etc. The scene in the editor we usually use is ac

FWorldContext is the device context for processing Level at the engine level, which is convenient for UEngine to manage and record information related to Wo not be directly operated by the logic layer. The data it stores includes World type, ContextHandle, GameInstance, GameViewport and other information.
UEngine controls and manages many internal systems and resources, and derives UGameEngine and UEditorEngine. It is a singleton global variable:

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

/** Global engine pointer. Can be 0 so don't use without checking. */ externENGINE_APIclass
UEngine*GEngine;
```

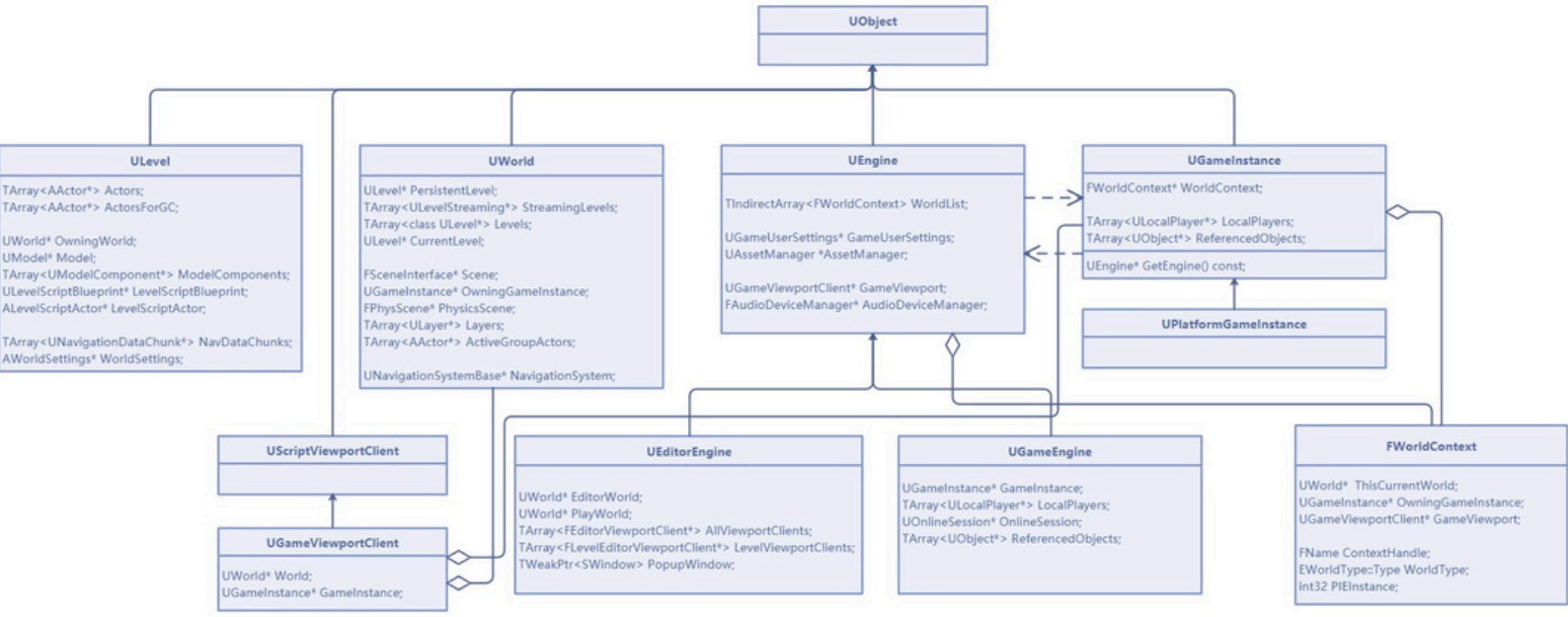
It is created and assigned in FEngineLoop::PreInitPostStartupScreen at the beginning of the program startup:

```
// Engine\Source\Runtime\Launch\Private\LaunchEngineLoop.cpp

int32FEngineLoop::PreInitPostStartupScreen(constTCHAR* CmdLine) {

	(......)

		if( GEngine == nullptr ) {

#if	WITH_EDITOR
			if( GIsEditor ) {

				FString EditorEngineClassName;
				GConfig->GetString(TEXT("/Script/Engine.Engine"), TEXT("EditorEngine"), EditorEngineClassName, GEngineIni); UClass* EditorEngineClass =
				StaticLoadClass( UEditorEngine::StaticClass(), nullptr, *EditorEngineClassName);

				//Create an editor engine instance
				GEngine = GEditor = NewObject<UEditorEngine>(GetTransientPackage(), EditorEngineClass);

				(......)
			}
			else
#endif
			{
				FString GameEngineClassName;
				GConfig->GetString(TEXT("/Script/Engine.Engine"), TEXT("GameEngine"), GameEngineClassName, GEngineIni);

				UClass* EngineClass = StaticLoadClass( UEngine::StaticClass(), nullptr, *GameEngineClassName);

				//Creating a game engine instance
				GEngine = NewObject<UEngine>(GetTransientPackage(), EngineClass);

				(......)
			}
		}

	(......)
```

```
    return0;
}
```

As you can see above, a UEditorEngine or UGameEngine instance will be created depending on whether it is in editor mode, and then assigned to the global v directly accessed by code from other places.

The inheritance, dependency, and reference relationships among ULevel, UWorld, FWorldContext, and UEngine are shown in the following figure:



1.4.3 Memory Allocation

UE's memory allocation system is large and complex, and the functions it provides are summarized as follows:

- Encapsulates the differences between system platforms and provides a unified interface.
- Efficiently creating and reclaiming memory according to certain rules can effectively improve memory operation efficiency.
- Supports multiple memory allocation schemes to meet your needs.
- Supports multiple calling methods to cope with different scenarios.
- Supports multi-thread-safe memory operations.
- TLS is partially supported (thread-local caching). Supports
- unified management of GPU memory. Provides memory
- debugging and statistics information.
- Good scalability.

So, how does UE achieve the above goals? We will reveal the secrets below.

1.4.3.1 Memory Allocation Basics

In order to better explain the memory allocation scheme later, this section first explains the basic concepts involved.

- **FFreeMem**

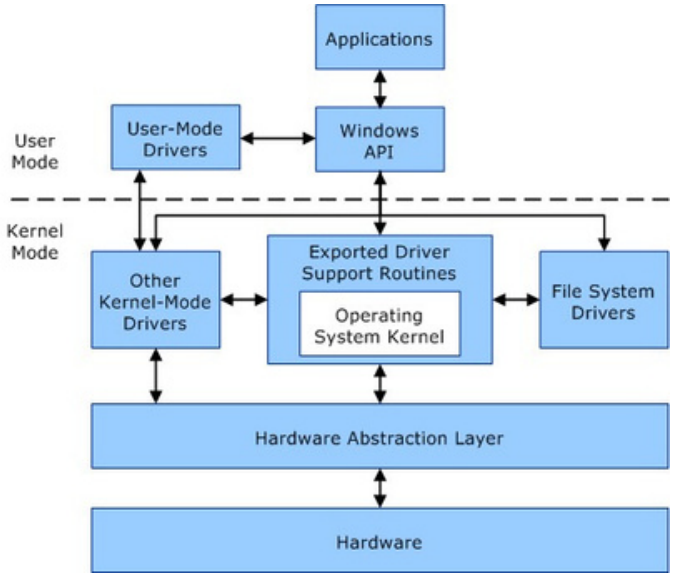
  The allocatable small block memory information record isFMallocBinneddefined as follows:

```
structFMallocBinned::FFreeMem {

    FFreeMem*    Next;          //Points to the next block of memory or the end. From the memory pool (poolManage and ensure order.
    uint32         NumFreeBlocks;     //The number of contiguous memory blocks that can be allocated
    uint32         Padding;       is at least1. //Let the structure16Byte-aligned padding bytes.
};
```

- **FPoolInfo**

  Memory pool, in common memory allocators, in order to reduce the system operation memory overhead, usually a large memory is allocated first, and several small blocks (UE's small blocks of memory are equal in size).

> **Why do we need to allocate a large block of memory and then divide it into several small blocks?**
> The Memory Management section in Chapter 5 of Game Engine Architecture gives an in-depth and clear answer. The reasons can be summarized as fo
> 1. The memory allocator usually operates in the heap, which is a slow process. It is a general-purpose device. If it is applied directly, it must handle allo lot of management overhead for the operating system.

> 2. On most operating systems, calling system memory operations will switch from user state to kernel state, and then switch back to user state after these states will take a lot of time.
>
> 
>
>
> *Schematic diagram of communication between user state and kernel state in Windows operating system. It can be seen that the communication betw layers of drivers.*
>
> This allocation method can effectively improve memory allocation efficiency and globally manage all memory operations (GC, optimization, defragme also has certain side effects, such as the inevitable waste of a certain proportion of memory space, the increase of instantaneous IO, the formation of m defragmented), etc.

FPoolInfo isFMallocBinneddefined as follows:

```
structFMallocBinned::FPoolInfo {

    uint16          Taken;          //The number of allocated memory blocks.
    uint16          TableIndex;     //LocatedMemSizeToPoolTableindex.
    uint32          AllocSize;      //The size of the allocated memory.

    FFreeMem*       FirstMem;       //If it is boxing mode, it points to the memory block available in the memory pool; if it is non-boxing mode, it points to the memory block directly allocated by the operating system.
    FPoolInfo*      Next;           //  Points to the next memory pool.
    FPoolInfo**       PrevLink;     //  Points to the previous memory pool.
};
```

Since the memory blocks in the memory pool are of equal size, the memory distribution diagram of the memory pool is as follows:

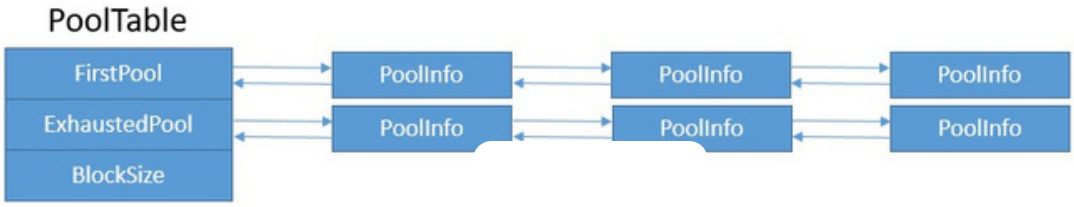
- **FPoolTable**

  The memory pool table uses a doubly linked list to store a group of memory pools. When a memory pool in the memory pool table cannot have allocat will be created and added to the doubly linked list.

  FPoolTable isFMallocBinneddefined as follows:

```
structFMallocBinned::FPoolTable {

    FPoolInfo*      FirstPool;      //  The initial memory pool is the header of the doubly linked list.
    FPoolInfo*      ExhaustedPool;  //  A linked list of memory pools that have been exhausted (no memory to allocate)
    uint32          BlockSize;      //  Memory block size
};
```

  Schematic diagram of the data structure of FPoolTable:

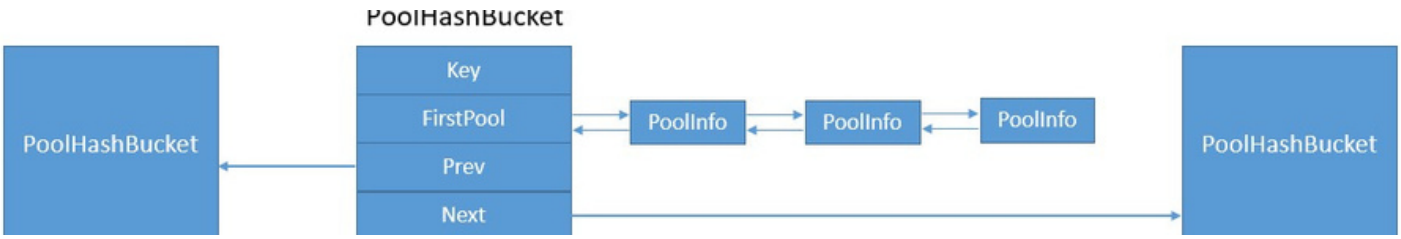


- **PoolHashBucket**

  The memory pool hash bucket is used to store all pool memorys corresponding to the keys hashed from the memory address. PoolHashBucket isFMall

```
structFMallocBinned::PoolHashBucket {

    UPTRINT          Key;        //  Hash Key
    FPoolInfo*       FirstPool;  //  Point to the first memory pool
    PoolHashBucket*  Prev;       //  Previous memory pool hash bucket
    PoolHashBucket*  Next;       //  Next memory pool hash bucket
};
```

  Its data structure diagram is as follows:



- **Memory size**

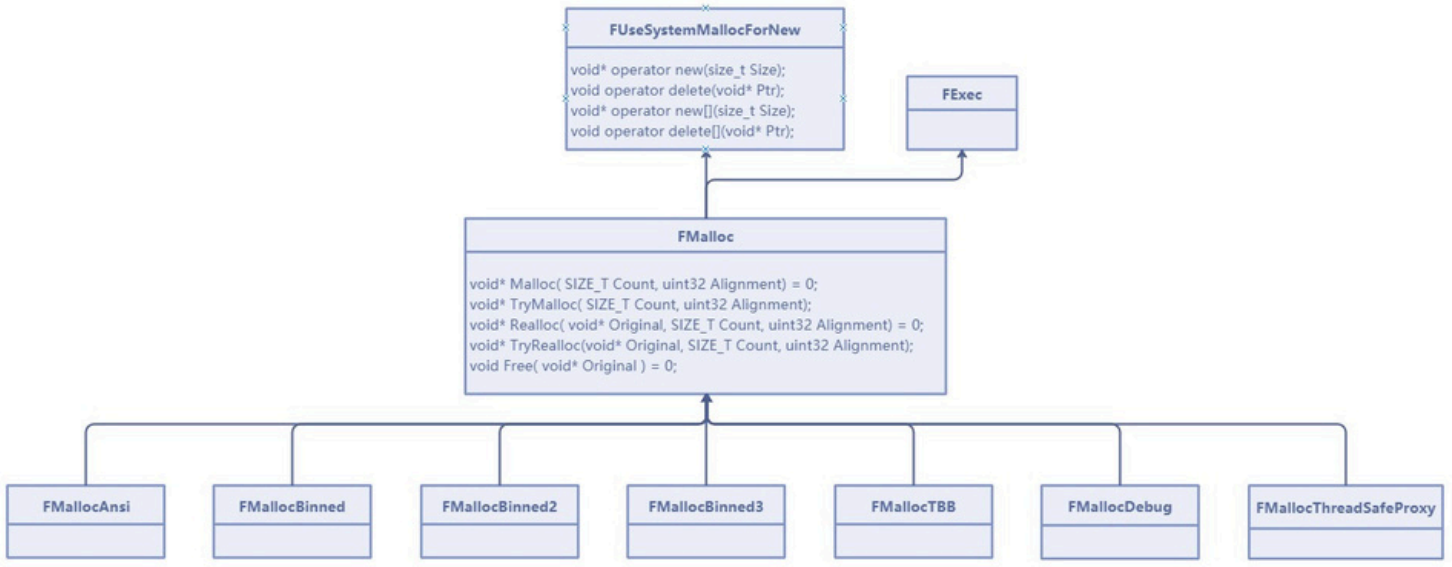
  The memory size of UE involves many parameters, including memory pool size (PoolSize), memory page size (PageSize) and memory block (BlockSize). T system platform, memory alignment, and caller. The following areFMallocBinnedthe sizes of some memory-related variables defined:

```
#if  PLATFORM_IOS
    #define  PLAT_PAGE_SIZE_LIMIT              16384
    #define  PLAT_BINNED_ALLOC_POOLSIZE        16384
    #define  PLAT_SMALL_BLOCK_POOL_SIZE        256
#else
    #define  PLAT_PAGE_SIZE_LIMIT              65536
    #define  PLAT_BINNED_ALLOC_POOLSIZE        65536
    #define  PLAT_SMALL_BLOCK_POOL_SIZE        0
#endif
```

  From this we can see that on the IOS platform, the memory page upper limit and memory pool size are 16k, and the boxed memory block size is 256 by upper limit and memory pool size are 64k, and the boxed memory block size is 0 bytes.

1.4.3.2 Memory Allocator

FMallocis the core class of UE memory allocator, controlling all memory allocation and release operations of UE. However, it is a virtual base class, which inhe FExec, and also has multiple subclasses, corresponding to different memory allocation schemes and strategies. The main inheritance relationship of FMalloc is



The above figure only shows some of the FMalloc subclasses. Other debugging and auxiliary classes are not included in this figure. The main classes of the FM follows:

- **FUseSystemMallocForNew**

  FUseSystemMallocForNew provides operator support for the new and delete keywords, and FMalloc inherits FUseSystemMallocForNew, which means th operations such as C++ keywords such as new and delete.

- **FMallocAnsi**

  The standard allocator directly calls C's malloc and free operations without any memory caching and allocation strategy management.

- **FMallocBinned**

  The standard (old) packing management method enables the memory pool table (FPoolTable), page memory pool table (FPagePoolTable) and memory p default memory allocation method of UE and also a memory allocation method supported by all platforms. Its core definition is as follows :

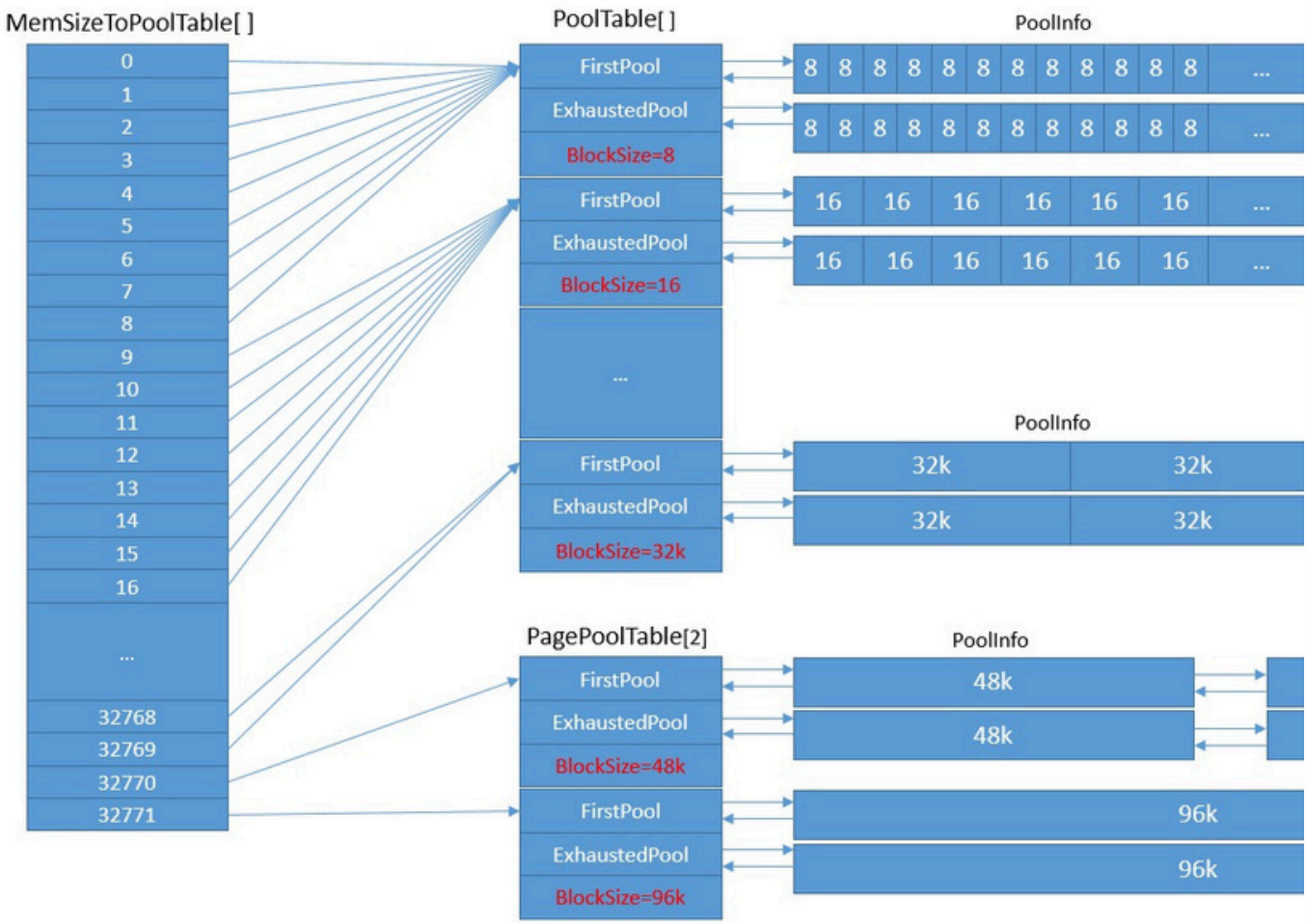
```
// Engine\Source\Runtime\Core\Public\HAL\MallocBinned.h

classFMallocBinned: public FMalloc {

private:
    enum{ POOL_COUNT =42};
    enum{ EXTENDED_PAGE_POOL_ALLOCATION_COUNT =2}; enum
    { MAX_POOLED_ALLOCATION_SIZE =32768+1};

    (......)

    FPoolTable      PoolTable[POOL_COUNT];          //List of all memory pool tables, a single memory poolBlockThe size is the same.
    FPoolTable      OsTable;        //A memory pool table that manages memory allocated directly by the system. However, after reading the
    FPoolTable      source code, I found that it is not used. PagePoolTable[EXTENDED_PAGE_POOL_ALLOCATION_COUNT];   //Memory pool table for memory pages (not
    FPoolTable*     small blocks of memory). MemSizeToPoolTable[MAX_POOLED_ALLOCATION_SIZE+EXTENDED_PAGE_POOL_ALLOCATION_COUNT];   //   Memory pool table indexed by size,   It will actually point toPoolT

    PoolHashBucket*     HashBuckets;        //  Memory pool hash bucket
    PoolHashBucket*     HashBucketFreeList;         //Allocatable memory pool hash buckets

    uint32              PageSize;       //Memory page size

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

In order to better understand the subsequent memory allocation mechanism, let's first analyze the initialization code of the memory allocator:

```
// Engine\Source\Runtime\Core\Private\HAL\MallocBinned.cpp

FMallocBinned::FMallocBinned(uint32 InPageSize, uint64 AddressLimit) {

    (......)

    //The maximum size of the box is8k(IOS)or32k(NoiOSplatform).
    BinnedSizeLimit = Private::PAGE_SIZE_LIMIT/2;

    (......)

    //Initialize the memory pool of memory pages1,By default, itsBlockSizefor12k(IOS)or48k(NoiOS
    platform). PagePoolTable[0].FirstPool = nullptr; PagePoolTable[0].ExhaustedPool = nullptr;

    PagePoolTable[0].BlockSize = PageSize == Private::PAGE_SIZE_LIMIT ? BinnedSizeLimit+(BinnedSizeLimit/2) :0;

    //Initialize the memory pool of memory pages2,By default, itsBlockSizefor24k(IOS)or96k(NoiOS
    platform). PagePoolTable[1].FirstPool = nullptr; PagePoolTable[1].ExhaustedPool = nullptr;

    PagePoolTable[1].BlockSize = PageSize == Private::PAGE_SIZE_LIMIT ? PageSize+BinnedSizeLimit :0;

    //Used to create differentBlockSizeArrays of numbers that follow two rules:1.Try to be an integer divisor (factor) of the memory pool size to reduce memory waste;2.must16Bit
    alignment. static constuint32 BlockSizes[POOL_COUNT] = {

        8,      16,     32,     48,     64,     80,     96,     112,
        128,    160,    192,    224,    256,    288,    320,    384,
        448,    512,    576,    640,    704,    768,    896,    1024,
        1168,   1360,   1632,   2048,   2336,   2720,   3264,   4096,
        4672,   5456,   6544,   8192,   9360,   10912,  13104,  16384,
        21840,  32768
    };

    //Create a memory pool table for the memory block andBlockSizes
    initializationBlockSize for( uint32 i =0; i < POOL_COUNT; i++ ) {

        PoolTable[i].FirstPool = nullptr;
        PoolTable[i].ExhaustedPool      = nullptr;
        PoolTable[i].BlockSize      =   BlockSizes[i];
#if   STATS
        PoolTable[i].MinRequest     =   PoolTable[i].BlockSize;
#endif
    }

    //initializationMemSizeToPoolTable,Point memory pool tables of all sizes to
    PoolTable. for( uint32 i =0; i<MAX_POOLED_ALLOCATION_SIZE; i++ )
```

```
    {
        uint32 Index =0;
        while( PoolTable[Index].BlockSize < i ) {

            + + Index;
        }
        checkSlow(Index < POOL_COUNT); MemSizeToPoolTable[i]
        = &PoolTable[Index];
    }

    //Add the memory pool table for the memory page toMemSizeToPoolTable
    The end of the array. MemSizeToPoolTable[BinnedSizeLimit]=&PagePoolTable[0];
    MemSizeToPoolTable[BinnedSizeLimit+1]                    = &PagePoolTable[1];

    check(MAX_POOLED_ALLOCATION_SIZE -1== PoolTable[POOL_COUNT -1].BlockSize);
}
```

In order to more clearly and intuitively illustrate the relationship and memory distribution between MemSizeToPoolTable, PoolTable and PagePoolTable, t schematic diagram:



FMallocBinnedThe main code and analysis of memory allocation are as follows:

```
// Engine\Source\Runtime\Core\Private\HAL\MallocBinned.cpp

void*FMallocBinned::Malloc(SIZE_T Size, uint32 Alignment) {

    (......)

    //Handle memory alignment and adjust according to
    memory alignmentSize if(Alignment ==
    DEFAULT_ALIGNMENT) {
        //The default memory alignment is16byte
        Alignment = Private::DEFAULT_BINNED_ALLOCATOR_ALIGNMENT;
    }
    Alignment = FMath::Max<uint32>(Alignment, Private::DEFAULT_BINNED_ALLOCATOR_ALIGNMENT); SIZE_T SpareBytesCount =
    FMath::Min<SIZE_T>(Private::DEFAULT_BINNED_ALLOCATOR_ALIGNMENT, Size); Size = FMath::Max<SIZE_T>(PoolTable[0].BlockSize,
    Size + (Alignment - SpareBytesCount));

    (......)

    FFreeMem* Free = nullptr; bool
    bUsePools =true;                    //Default memory pool

    (......)
```

```
    if  (bUsePools)
    {
        //If the allocated size is smaller thanBinnedSizeLimit(32k),It indicates memory fragmentation, putMemSizeToPoolTableofFPoolTable
        middle. if( Size < BinnedSizeLimit) {

            // Allocate from pool.
            FPoolTable* Table = MemSizeToPoolTable[Size];
#ifdef      USE_FINE_GRAIN_LOCKS
            FScopeLockTableLock(&Table->CriticalSection);
#endif
            checkSlow(Size <= Table->BlockSize);

            Private::TrackStats(Table, (uint32)Size);

            FPoolInfo* Pool = Table->FirstPool; if( !Pool )

            {
                Pool = Private::AllocatePoolMemory(*this, Table, Private::BINNED_ALLOC_POOL_SIZE/*PageSize*/, Size);
            }

            Free = Private::AllocateBlockFromPool(*this, Table, Pool, Alignment);
        }
        //If the allocated size is inBinnedSizeLimit(32k)andPagePoolTable[0].BlockSize(48k)between, or inPageSize(64k)andPagePoolTable[1].BlockSize(96k)between else if( ((Size >=
        BinnedSizeLimit && Size <= PagePoolTable[0].BlockSize) ||
                (Size > PageSize && Size <= PagePoolTable[1].BlockSize)))
        {
            // Bucket in a pool of 3*PageSize or 6*PageSize uint32 BinType =
            Size < PageSize?0:1; uint32 PageCount =3*BinType +3;
            FPoolTable* Table = &PagePoolTable[BinType];
            USE_FINE_GRAIN_LOCKS
#ifdef
            FScopeLockTableLock(&Table->CriticalSection);
#endif
            checkSlow(Size <= Table->BlockSize);

            Private::TrackStats(Table, (uint32)Size);

            FPoolInfo* Pool = Table->FirstPool; if( !Pool )

            {
                Pool = Private::AllocatePoolMemory(*this, Table, PageCount*PageSize, BinnedSizeLimit+BinType);
            }

            Free = Private::AllocateBlockFromPool(*this, Table, Pool, Alignment);
        }
        //Exceeds the memory page size, the system directly allocates memory and puts it inHashBucketsIn the
        table. else
        {
            // Use OS for large allocations. UPTRINT AlignedSize =
            Align(Size,PageSize);
            SIZE_T ActualPoolSize;//TODO:use this to reduce waste?
            Free = (FFreeMem*)Private::OSAlloc(*this, AlignedSize, ActualPoolSize); if( !Free )

            {
                Private::OutOfMemory(AlignedSize);
            }

            void*AlignedFree = Align(Free, Alignment);

            // Create        indirect.
            FPoolInfo*       Pool;
            {
#ifdef      USE_FINE_GRAIN_LOCKS
                FScopeLockPoolInfoLock(&AccessGuard);
#endif
                Pool = Private::GetPoolInfo(*this, (UPTRINT)Free);

                if((UPTRINT)Free != ((UPTRINT)AlignedFree & ~((UPTRINT)PageSize -1))) {

                    // Mark the FPoolInfo for AlignedFree to jump back to the FPoolInfo for ptr.
                    for(UPTRINT i = (UPTRINT)PageSize, Offset =0; i < AlignedSize; i += PageSize, ++Offset) {

                        FPoolInfo* TrailingPool = Private::GetPoolInfo(*this, ((UPTRINT)Free) + i); check(TrailingPool);

                        //Set trailing pools to point back to first pool TrailingPool->SetAllocationSizes(0,0, Offset,
                        BinnedOSTableIndex);
                    }
                }
            }
            Free    = (FFreeMem*)AlignedFree;
            Pool->SetAllocationSizes(Size, AlignedSize, BinnedOSTableIndex, BinnedOSTableIndex);

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

```
    returnFree;
}
```

To sum up the above code, in the case of non-IOS platform and default page size (64k), the allocation strategy of FMallocBinned is briefly described as f

- The size of the memory to be allocated is (0, 32k), and is allocated and stored using PoolTable of MemSizeToPoolTable.
- The size of the memory to be allocated is [32k, 48K] or [64k, 96k], and is allocated and stored using the PoolTable of PagePoolTable.
- The size of other memory to be allocated is directly allocated by the system and placed in HashBuckets.

> **Why does UE directly allocate memory between (48k, 64k) and not use boxing?**
> Since the memory pool of FMallocBinned is divided equally, if the memory between (48k, 64k) is allocated by binning, it must be placed in the 6 memory waste between (0, 16k). In other words, in the worst case, each memory allocation in this interval will waste 16k of memory, and the me 33.33%. This is obviously unacceptable for the UE official team that emphasizes high performance. We chose the lesser of two evils and came up
> and cons.
> Of course, there is room for optimization here, that is, the memory between (48k, 64k) can be assembled from smaller blocks. For example, 50k with a BlockSize of 2k, occupying only 25 blocks (but at the same time it will increase the management complexity of the memory pool and mem

FMallocBinned and the FMallocBinned2 and FMallocBinned3 mentioned below actually allocate a large memory in advance, and then allocate appropria memory. Although these methods can improve the efficiency of memory allocation, the instantaneous IO pressure will increase, and memory waste will

> The memory waste of FMallocBinned is mainly reflected in the following points:
> 1. The newly allocated memory pool often cannot be fully utilized immediately, resulting in redundancy of certain programs.
> 2. Due to memory alignment and size alignment, many memory blocks of consecutive sizes are mapped upward to the memory pool table of the sam [9, 16] are mapped to the memory pool table with BlockSize of 16), which also leads to a certain proportion of memory waste.
> 3. Maintain the additional memory generated by the allocator's memory pool table, memory pool, hash bucket, memory block and other information.

- **FMallocBinned2**

  The new boxed memory allocation method. From the analysis of the source code, we can see that FMallocBinned2 is simpler than FMallocBinned. It will strategy based on the small block memory, alignment size and whether the thread cache is enabled (enabled by default).
- **FMallocBinned3**

  A new binned memory allocation method available only on 64-bit systems. The implementation is similar to FMallocBinned2 and supports thread cachin
- **FMallocTBB**

  FMallocTBB adopts the scalable_allocator allocator in the third-party memory allocator TBB. The interface provided by scalable_allocator is as follows:

```
// Engine\Source\ThirdParty\IntelTBB\IntelTBB-2019u8\include\tbb\scalable_allocator.h

void* __TBB_EXPORTED_FUNCscalable_malloc(size_tsize); void
        __TBB_EXPORTED_FUNCscalable_free(void*ptr);
void* __TBB_EXPORTED_FUNCscalable_realloc(void* ptr,size_tsize); void* __TBB_EXPORTED_FUNC
scalable_calloc(size_tnobj,size_tsize);
int__TBB_EXPORTED_FUNCscalable_posix_memalign(void** memptr,size_talignment,size_tsize); void* __TBB_EXPORTED_FUNC
scalable_aligned_malloc(size_tsize,size_talignment);
void* __TBB_EXPORTED_FUNCscalable_aligned_realloc(void* ptr,size_tsize,size_talignment); void__TBB_EXPORTED_FUNC
scalable_aligned_free(void*ptr); size_t__TBB_EXPORTED_FUNCscalable_msize(void*ptr);
```

FMallocTBB uses the abovescalable_aligned_mallocinterface to implement memory operations, and the allocation code is as follows:

```
// Engine\Source\Runtime\Core\Private\HAL\MallocTBB.cpp

void*FMallocTBB::TryMalloc( SIZE_T Size, uint32 Alignment ) {

    (......)

    void* NewPtr = nullptr;

    if( Alignment != DEFAULT_ALIGNMENT ) {

        Alignment = FMath::Max(Size >=16? (uint32)16: (uint32)8, Alignment); NewPtr =
        scalable_aligned_malloc( Size, Alignment );
    }
    else
    {
        // Fulfill the promise of DEFAULT_ALIGNMENT, which aligns 16-byte or larger structures to 16 bytes, // while TBB aligns to 8 by default.

        NewPtr = scalable_aligned_malloc( Size, Size >=16? (uint32)16: (uint32)8);
    }

     (......)

    returnNewPtr;
}
```

> TBB (Threading Building Blocks) is developed by Intel and provides SDK. Its features include:
>
> - Provides three allocation methods: tbb_allocator, scalable_allocator and cache_aligned_allocator. Parallel
> - algorithms and data structures.
> - Task-based memory scheduler.
> - It is multi-thread friendly and supports multiple threads to operate memory at the same time. scalable_allocator does not allocate memory in the consumption caused by multi-thread competition.
> - The cache processing efficiency is higher than other methods. cache_aligned_allocator solves the problem of false sharing by aligning the cache.

- **Other memory allocators**

  In addition to the commonly used basic memory allocators mentioned above, UE also comes with FMallocDebug (debugging memory), FMallocStomp (FMallocJemalloc (suitable for memory allocation management under multi-threading) and GPU video memory related allocation (FMallocBinnedGPU), e quite special and will not be described in detail here. Interested readers can study the source code by themselves.

1.4.3.3 Memory Operation Mode

The previous section explained the memory allocation method and strategy technology. Next, let's talk about memory usage. For the caller, there are several w

- **GMalloc**:GMalloc is a global memory allocator, whichFPlatformMemoryis created at the beginning of UE startup:

```
// Engine\Source\Runtime\Core\Private\HAL\UnrealMemory.cpp

static int FMemory_GCreateMalloc_ThreadUnsafe() {

	(......)

	GMalloc = FPlatformMemory::BaseAllocator();

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

FPlatformMemoryDifferent operating systems have different types. For example, in Windows, it is actually FWindowsPlatformMemory:

```
// Engine\Source\Runtime\Core\Public\Windows\WindowsPlatformMemory.h

structCORE_API FWindowsPlatformMemory: public FGenericPlatformMemory {

	(......)

	staticclass FMalloc*BaseAllocator();

	(......)
};

typedefFWindowsPlatformMemory FPlatformMemory;
```

As can be seen from the above code, GMalloc is actually an instance of FMalloc. DifferentFPlatformMemoryFMalloc subclasses are created in different op allocation strategies. The followingFWindowsPlatformMemory::BaseAllocatorThe code is analyzed:

```
// Engine\Source\Runtime\Core\Private\Windows\WindowsPlatformMemory.cpp

FMalloc*FWindowsPlatformMemory::BaseAllocator() {

#ifENABLE_WIN_ALLOC_TRACKING
	// This allows tracking of allocations that don't happen within the engine's wrappers. // This actually won't be compiled
	unless bDebugBuildsActuallyUseDebugCRT is set in the // build configuration for UBT.

	_CrtSetAllocHook(WindowsAllocHook);
#endif// ENABLE_WIN_ALLOC_TRACKING

	//  Adopt different memory allocation strategies according to macro definitions
	if  (FORCE_ANSI_ALLOCATOR)//-V517
	{
		AllocatorToUse = EMemoryAllocatorToUse::Ansi;
	}
	else if((WITH_EDITORONLY_DATA || IS_PROGRAM) && TBB_ALLOCATOR_ALLOWED)//-V517 {

		AllocatorToUse = EMemoryAllocatorToUse::TBB;
	}
#if  PLATFORM_64BITS
	else if((WITH_EDITORONLY_DATA || IS_PROGRAM) && MIMALLOC_ALLOCATOR_ALLOWED)//-V517 {

		AllocatorToUse = EMemoryAllocatorToUse::Mimalloc;
	}
	else if(USE_MALLOC_BINNED3) {

		AllocatorToUse = EMemoryAllocatorToUse::Binned3;
	}
#endif
```

```
    else if(USE_MALLOC_BINNED2) {

        AllocatorToUse        =  EMemoryAllocatorToUse::Binned2;
    }
    else
    {
        AllocatorToUse        =  EMemoryAllocatorToUse::Binned;
    }

#if  !UE_BUILD_SHIPPING
    // If not shipping, allow overriding with command line options, this happens very early so we need to use windows functions constTCHAR* CommandLine
    = ::GetCommandLineW();

    //  Adjust memory allocation strategy from command line
    if  (FCString::Stristr(CommandLine, TEXT("-ansimalloc")))
    {
        AllocatorToUse = EMemoryAllocatorToUse::Ansi;
    }
#if  TBB_ALLOCATOR_ALLOWED
    else if(FCString::Stristr(CommandLine, TEXT("-tbbmalloc"))) {

        AllocatorToUse = EMemoryAllocatorToUse::TBB;
    }
#endif
#if  MIMALLOC_ALLOCATOR_ALLOWED
    else if(FCString::Stristr(CommandLine, TEXT("-mimalloc"))) {

        AllocatorToUse = EMemoryAllocatorToUse::Mimalloc;
    }
#endif
#if  PLATFORM_64BITS
    else if(FCString::Stristr(CommandLine, TEXT("-binnedmalloc3"))) {

        AllocatorToUse = EMemoryAllocatorToUse::Binned3;
    }
#endif
    else if(FCString::Stristr(CommandLine, TEXT("-binnedmalloc2"))) {

        AllocatorToUse = EMemoryAllocatorToUse::Binned2;
    }
    else if(FCString::Stristr(CommandLine, TEXT("-binnedmalloc"))) {

        AllocatorToUse = EMemoryAllocatorToUse::Binned;
    }
#if  WITH_MALLOC_STOMP
    else if(FCString::Stristr(CommandLine, TEXT("-stompmalloc"))) {

        AllocatorToUse        = EMemoryAllocatorToUse::Stomp;
    }
#endif    //  WITH_MALLOC_STOMP
#endif    //  !UE_BUILD_SHIPPING

    //Create according to different typesFMallocA
    subclass object of . switch(AllocatorToUse) {

    caseEMemoryAllocatorToUse::Ansi:
        returnnew FMallocAnsi();
#if         WITH_MALLOC_STOMP
    case   EMemoryAllocatorToUse::Stomp: return
        new FMallocStomp();
#endif
#if  TBB_ALLOCATOR_ALLOWED
    caseEMemoryAllocatorToUse::TBB:
        returnnew FMallocTBB();
#endif
#ifMIMALLOC_ALLOCATOR_ALLOWED && PLATFORM_SUPPORTS_MIMALLOC case
 EMemoryAllocatorToUse::Mimalloc:
        returnnew FMallocMimalloc();
#endif
    caseEMemoryAllocatorToUse::Binned2:
        returnnew FMallocBinned2();
#ifPLATFORM_64BITS
    caseEMemoryAllocatorToUse::Binned3:
        return     new FMallocBinned3();
#endif
    default:            //  intentional fall-through
    caseEMemoryAllocatorToUse::Binned:
        returnnew FMallocBinned((uint32)(GetConstants().BinnedPageSize&MAX_uint32), (uint64)MAX_uint32 +1);
    }
}
```

From this we can see that GMalloc operates memory through a subclass of FMalloc. The following table shows the memory allocation methods supporte systems:

operating system	Supported memory allocation methods	Default memory allocation method
Windows	Ansi, Binned, Binned2, Binned3, TBB, Stomp, Mimalloc	Binned
Android	Binned, Binned2, Binned3	Binned
Apple(IOS, Mac)	Ansi, Binned, Binned2, Binned3	Binned
Unix	Ansi, Binned, Binned2, Binned3, Stomp, Jemalloc	Binned
HoloLens	Ansi, Binned, TBB	Binned

- **FMemory**:FMemory is a static tool class of UE. It provides many static methods for operating memory. The common APIs are as follows:

```
// Engine\Source\Runtime\Core\Public\HAL\UnrealMemory.h

struct     CORE_API FMemory
{
      //    Direct callcMemory allocation and deallocation interface.
      static void*SystemMalloc(SIZE_T Size); static void
      SystemFree(void* Ptr);


      //passGMallocObject Operation Memory
      static void*Malloc(SIZE_T Count, uint32 Alignment = DEFAULT_ALIGNMENT);
      static void*Realloc(void* Original, SIZE_T Count, uint32 Alignment = DEFAULT_ALIGNMENT); static void Free(void* Original);

      static void*MallocZeroed(SIZE_T Count, uint32 Alignment = DEFAULT_ALIGNMENT);

      //Memory auxiliary interface
      static void*Memmove(void* Dest,const void* Src, SIZE_T Count ); staticint32Memcmp(const void*
      Buf1,const void* Buf2, SIZE_T Count ); static void*Memset(void* Dest, uint8 Char, SIZE_T Count);
      static void*Memzero(void* Dest, SIZE_T Count);

      static void*Memcpy(void* Dest,const void* Src, SIZE_T Count); static void*BigBlockMemcpy(void*
      Dest,const void* Src, SIZE_T Count); static void*StreamingMemcpy(void* Dest,const void* Src, SIZE_T
      Count); static void Memswap(void* Ptr1,void* Ptr2, SIZE_T Size );



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

From the above code, we can seeFMemorythat both GMalloc and C-style memory operations are supported.

- **New/delete operators**:In addition to some classes overloading the new and delete operators, other global new and delete use the following declaratio

```
// Engine\Source\Runtime\Core\Public\Modules\Boilerplate\ModuleBoilerplate.h

#defineREPLACEMENT_OPERATOR_NEW_AND_DELETE \
      OPERATOR_NEW_MSVC_PRAGMA void* operator new ( size_t Size                                                    ) OPERATOR_NEW_THROW_SPEC          { return FMemory::Mallo
      OPERATOR_NEW_MSVC_PRAGMA void* operator new[]( size_t Size                                                   ) OPERATOR_NEW_THROW_SPEC          { return FMemory::Mallo
      OPERATOR_NEW_MSVC_PRAGMA void* operator new ( size_t Size, const std::nothrow_t& ) OPERATOR_NEW_NOTHROW_SPEC                                    { return FMemory::Mallo
      OPERATOR_NEW_MSVC_PRAGMA void* operator new[]( size_t Size, const std::nothrow_t& ) OPERATOR_NEW_NOTHROW_SPEC void operator delete               { return FMemory::Mallo
      ( void* Ptr )                                                                                                                                  OPERATOR_DELETE_THROW_SPEC OPERA{ TFOMRe_mDEoLryE:T:FEr_eTeH( RPOtrW
      void operator delete[]( void* Ptr )                                                                                                            OPERATOR_DELETE_NOTHROW_SPEC { FM{ FeMmeomryo::rFyr:e:Fer(e Pet(r P )t;r);
      void operator delete ( void* Ptr, const std::nothrow_t& ) void operator delete[]                                                               OPERATOR_DELETE_NOTHROW_SPEC { FMemory::Free( Ptr );
      ( void* Ptr, const std::nothrow_t& ) void operator delete ( void* Ptr, size_t Size )                                                           OPERATOR_DELETE_THROW_SPEC
                                                                                                                                                                                  { FMemory::Free( Ptr );
      void operator delete[]( void* Ptr, size_t Size )                                                                                               OPERATOR_DELETE_THROW_SPEC OPERA{ TFOMRe_mDEoLryE:T:FEr_eNeO( PTHtrR
      void operator delete ( void* Ptr, size_t Size, const std::nothrow_t& ) void operator delete[]( void* Ptr,                                      { FMemory::Free( Ptr ); OPERATOR_DELETE_NOTHROW_SPEC
      size_t Size, const std::nothrow_t& )                                                                                                           { FMemory::Free( Ptr );
```

As can be seen from the source code, the global operator memory alsoFMemorycompletes memory operations through calls.

- **Specific API**:In addition to the above three memory operation methods, UE also provides various interfaces for creating and destroying specific memo example:

```
structFPooledVirtualMemoryAllocator {

      void*Allocate(SIZE_T Size); void Free(void* Ptr,
      SIZE_T Size);
};

classCORE_API FAnsiAllocator {

      classCORE_API ForAnyElementType {

            void ResizeAllocation(SizeType PreviousNumElements, SizeType NumElements, SIZE_T NumBytesPerElement);
      };
};
```

```
classFVirtualAllocator {

    void*AllocateVirtualPages(uint32 NumPages,size_tAlignmentForCheck); void FreeVirtual(void* Ptr,
    uint32 NumPages);
};

classRENDERER_API FVirtualTextureAllocator {

    uint32Alloc(FAllocatedVirtualTexture* VT ); void Free
    (FAllocatedVirtualTexture* VT );
};

template<SIZE_T RequiredAlignment>classTMemoryPool {

    void*Allocate(SIZE_T Size); void Free(void*Ptr,
    SIZE_T Size);
};
```

From the caller's perspective, in most cases,new/deleteoperators andFMemorymethods are used to manipulate memory, and it is rare to directly apply for sy

1.4.4 Garbage Collection

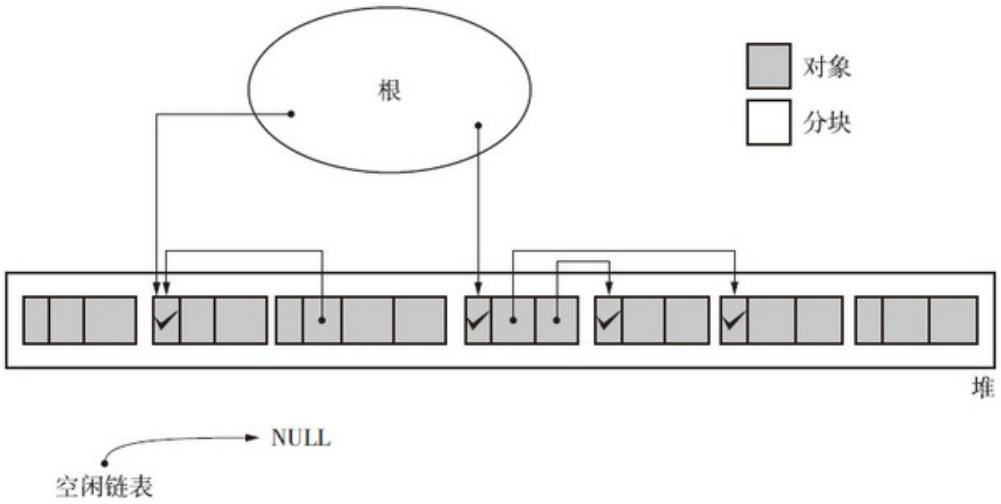
Garbage collectionis abbreviated as GC (Garbage Collection), which is a mechanism that recycles or reuses invalid resources with a certain strategy. It is often operating systems, etc.

1.4.4.1 GC Algorithm Overview

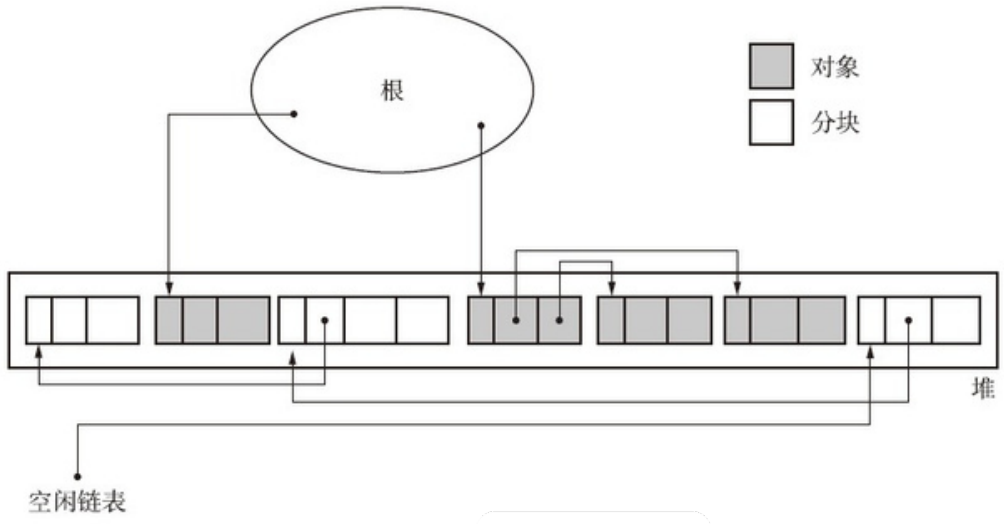
In the book"Garbage Collection Algorithms and Implementations", the GC algorithms mentioned are:

- **Mark-Sweep.**That is, the mark-clean algorithm, the algorithm is divided into two stages:

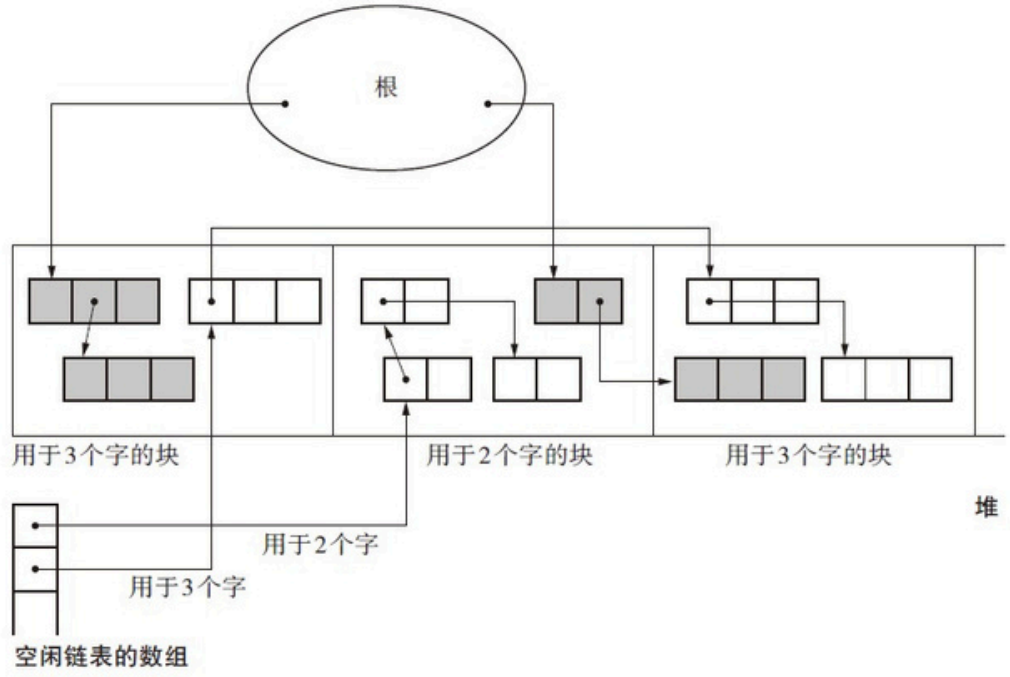
  The first phase is the Mark phase, which traverses the root's active object list and marks the heap objects pointed to by all active objectsTRUE.



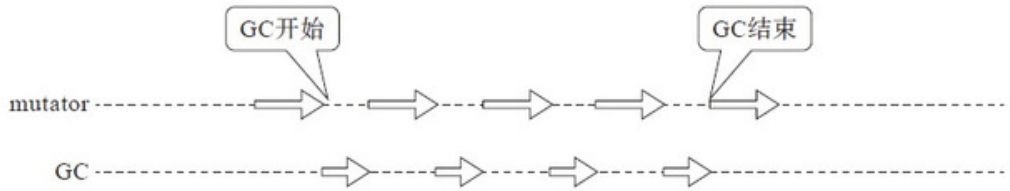
  The second phase is the Sweep phase, which traverses the heap list, releases all markedFALSEobjects to the allocatable heap, and resets the mark of th can be performed next time.



- **BiBOP**.The full name is Big Bag Of Pages. Its approach is to organize objects of similar size into fixed-size blocks for management, which FMallocBinned allocator.



- **Conservative GC.**Conservative GC is characterized by its inability to identify pointers and non-pointers. Since at the GC level, it is impossible to determine its memory value, many methods are derived to determine whether it is a pointer, which requires a certain cost. In contrast ,**Exact GC**can clearly identify
- **Generational GC**.Generational garbage collection, this method introduces the concept of age in objects and improves the efficiency of garbage collect likely to become garbage.
- **Incremental GC**.Incremental garbage collection is a method of controlling the maximum pause time of mutators by gradually advancing garbage collector



Schematic diagram of incremental garbage collection.

- **Reference Counting Immix**.Also known as RC Immix algorithm, it is a reference merging GC algorithm. Its purpose is to improve the behavior of refere improve GC throughput.

UE's GC algorithm is mainly based on Mark-Sweep, which is used to clean up UObject objects. Like the Mark-Sweep algorithm, UE also has the concept of Roo properties and static variables) from being cleaned up by GC, you can use UObject's AddToRoot interface.

1.4.4.2 UE GC

The main implementation code and analysis of the UE's GC module are as follows:

```
// Engine\Source\Runtime\CoreUObject\Private\UObject\GarbageCollection.cpp

void CollectGarbage(EObjectFlags KeepFlags,boolbPerformFullPurge) {

	//getGClock, preventGCThe process is operated by
	other threads AcquireGCLock();

	//implementGCprocess
	CollectGarbageInternal(KeepFlags, bPerformFullPurge);

	//releaseGCLock so that other threads can
	operate ReleaseGCLock();
}

//Real executionGCoperate.KeepFlags: Exclude cleanupUObjectmark,bPerformFullPurge: Whether to turn off
incremental updates void CollectGarbageInternal(EObjectFlags KeepFlags,boolbPerformFullPurge) {

	(.....)

	{
		FGCScopeLock GCLock;

		//Make sure the last incremental garbage cleanup has been completed, or simply do a full cleanup to prevent the previous callGCThere
		was some leftover garbage. if(GObjIncrementalPurgeIsInProgress || GObjPurgeIsRequired)
		{
```

```
            IncrementalPurgeGarbage(false);
            FMemory::Trim();
        }

        // This can happen if someone disables clusters from the console (gc.CreateGCClusters) if(!GCreateGCClusters &&
        GUObjectClusters.GetNumAllocatedClusters()) {

            GUObjectClusters.DissolveClusters(true);
        }

        (.....)

        // Fall back to single threaded GC if processor count is 1 or parallel GC is disabled // or detailed per class gc stats are
        enabled (not thread safe)
        // Temporarily forcing single-threaded GC in the editor until Modify() can be safely removed from HandleObjectReference. const boolbForceSingleThreadedGC =
        ShouldForceSingleThreadedGC();
        // Run with GC clustering code enabled only if clustering is enabled and there's actual allocated clusters const boolbWithClusters = !!
        GCreateGCClusters && GUObjectClusters.GetNumAllocatedClusters();


        {
            const doubleStartTime = FPlatformTime::Seconds(); FRealtimeGC
            TagUsedRealtimeGC;
            //Perform reachability analysis (i.e. marking)
            TagUsedRealtimeGC.PerformReachabilityAnalysis(KeepFlags,
                                                                    bForceSingleThreadedGC,          bWithClusters);
            UE_LOG(LogGarbage, Log, TEXT("%f ms for GC"), (FPlatformTime::Seconds() - StartTime) *1000);
        }

        // Reconstruct clusters if needed
        if(GUObjectClusters.ClustersNeedDissolving()) {

            const doubleStartTime = FPlatformTime::Seconds();
            GUObjectClusters.DissolveClusters();
            UE_LOG(LogGarbage, Log, TEXT("%f ms for dissolving GC clusters"), (FPlatformTime::Seconds() - StartTime) *1000);
        }

        // Fire post-reachability analysis hooks
        FCoreUObjectDelegates::PostReachabilityAnalysis.Broadcast();

        {
            FGCArrayPool::Get().ClearWeakReferences(bPerformFullPurge);

            //Collecting unreachable objects
            GatherUnreachableObjects(bForceSingleThreadedGC);

            if(bPerformFullPurge || !GIncrementalBeginDestroyEnabled) {

                //Remove unreachable objects from the hash table
                UnhashUnreachableObjects(/**bUseTimeLimit = */false);
                FScopedCBDProfile::DumpProfile();
            }
        }

        // Set flag to indicate that we are relying on a purge to be performed. GObjPurgeIsRequired =true;


        //Clean up all the garbage
        if(bPerformFullPurge || GIsEditor) {

            IncrementalPurgeGarbage(false);
        }

        //    Zoom outUObjectHash Table
        if    (bPerformFullPurge)
        {
            ShrinkUObjectHashTables();
        }

        // Destroy all pending delete linkers
        DeleteLoaders();

        //Free up memory.
        FMemory::Trim();
    }

    // Route callbacks to verify GC assumptions
    FCoreUObjectDelegates::GetPostGarbageCollect().Broadcast();

    STAT_ADD_CUSTOMMESSAGE_NAME( STAT_NamedMarker, TEXT("GarbageCollection - End") );
}
```

The marking phase`FRealtimeGC::PerformReachabilityAnalysis`is completed by the interface:

```
// Engine\Source\Runtime\CoreUObject\Private\UObject\GarbageCollection.cpp

classFRealtimeGC: public FGarbageCollectionTracer {
```

```
    void PerformReachabilityAnalysis(EObjectFlags KeepFlags,boolbForceSingleThreaded,boolbWithClusters) {

        (.....)

        /** Growing array of objects that require serialization */ FGCArrayStruct* ArrayStruct =
        FGCArrayPool::Get().GetArrayStructFromPool(); TArray<UObject*>& ObjectsToSerialize = ArrayStruct-
        >ObjectsToSerialize;


        //Reset the number of objects.
        GObjectCountDuringLastMarkPhase.Reset();

        // Make sure GC referencer object is checked for references to other objects even if it resides in permanent object pool
        if(FPlatformProperties::RequiresCookedData() && FGCObject::GGCObjectReferencer && GUObjectArray.IsDisregardForGC(FGCObject::GGCObjectRef {

            ObjectsToSerialize.Add(FGCObject::GGCObjectReferencer);
        }

        {
            const doubleStartTime = FPlatformTime::Seconds(); //Use the function of
            marking objects to mark the corresponding objects.
            (this->*MarkObjectsFunctions[GetGCFunctionIndex(!bForceSingleThreaded,                bWithClusters)])(ObjectsToSerialize,                KeepFlags);
            UE_LOG(LogGarbage, Verbose, TEXT("%f ms for Mark Phase (%d Objects To Serialize"), (FPlatformTime::Seconds() - StartTime) *1000, Obj
        }

        {
            const doubleStartTime = FPlatformTime::Seconds(); //Perform
            reachability analysis on an object.
            PerformReachabilityAnalysisOnObjects(ArrayStruct, bForceSingleThreaded,                bWithClusters);
            UE_LOG(LogGarbage, Verbose, TEXT("%f ms for Reachability Analysis"), (FPlatformTime::Seconds() - StartTime) *1000);
        }

        // Allowing external systems to add object roots. This can't be done through AddReferencedObjects // because it may require tracing objects (via
        FGarbageCollectionTracer) multiple times FCoreUObjectDelegates::TraceExternalRootsForReachabilityAnalysis.Broadcast(*this, KeepFlags,
        bForceSingleThreaded);


        FGCArrayPool::Get().ReturnToPool(ArrayStruct);

#ifUE_BUILD_DEBUG
        FGCArrayPool::Get().CheckLeaks();
#endif
    }
};
```

The aboveMarkObjectsFunctionssumPerformReachabilityAnalysisOnObjectsis actually a combined template function for whether to support parallel (Parallel)

```
// Engine\Source\Runtime\CoreUObject\Private\UObject\GarbageCollection.cpp

classFRealtimeGC: public FGarbageCollectionTracer {

    //statement
    MarkObjectsFn MarkObjectsFunctions[4]; ReachabilityAnalysisFn
    ReachabilityAnalysisFunctions[4];

    //initialization
    FRealtimeGC()
    {
        MarkObjectsFunctions[GetGCFunctionIndex(false,false)]   =   &FRealtimeGC::MarkObjectsAsUnreachable<false,false>;
        MarkObjectsFunctions[GetGCFunctionIndex(true,false)]    =    &FRealtimeGC::MarkObjectsAsUnreachable<true,false>;
        MarkObjectsFunctions[GetGCFunctionIndex(false,true)]    =    &FRealtimeGC::MarkObjectsAsUnreachable<false,true>;
        MarkObjectsFunctions[GetGCFunctionIndex(true,true)] = &FRealtimeGC::MarkObjectsAsUnreachable<true,true>;

        ReachabilityAnalysisFunctions[GetGCFunctionIndex(false,false)]    =    &FRealtimeGC::PerformReachabilityAnalysisOnObjectsInternal<false,false
        ReachabilityAnalysisFunctions[GetGCFunctionIndex(true,false)]    =   &FRealtimeGC::PerformReachabilityAnalysisOnObjectsInternal<true,false>;
        ReachabilityAnalysisFunctions[GetGCFunctionIndex(false,true)]    =   &FRealtimeGC::PerformReachabilityAnalysisOnObjectsInternal<false,true>;
        ReachabilityAnalysisFunctions[GetGCFunctionIndex(true,true)] = &FRealtimeGC::PerformReachabilityAnalysisOnObjectsInternal<true,true>;
    }
};
```

From the source code, we can see that UE's GC has the following characteristics:

- The main algorithm is Mark-Sweep. However, unlike the traditional Mark-Sweep algorithm which has only two steps, UE's GC has three steps:

  1. Index reachable object.

  2. Collect objects to be cleaned.

  3. Clean up the objects collected in step 2.

- Cleans up the UObject on the game thread.
- Thread-safe, supports multi-threaded parallel and cluster processing to improve throughput.
- Supports full cleanup, which is mandatory in editor mode; also supports incremental cleanup to prevent GC processing threads from being stuck for too
- You can specify that certain marked objects will not be cleaned up.

In fact, the GC mechanism and principle of UE are much more complicated than the above description. However, due to the limited space and topic, I will not i interested can study the UE source code or seek references.

1.4.5 Memory Barriers

Memory Barrieris also called**Membar**,**memory fence**or**fence instruction**.It is used to solve the problem of disordered memory access and asynchronous

Memory disorder problems can occur at compile time or runtime. Compile time disorder is caused by compiler optimization that leads to changes in instructio by out-of-order access of multiple processes and multiple threads.

1.4.5.1 Compile-time memory barriers

For compile-time memory disorder, for example, suppose there is the following C++ code:

```
sum = a + b + c;
print(sum);
```

After being compiled by the compiler, the assembly instruction sequence may become one of the following three:

```
//Instruction sequence1
sum = a + b;
sum = sum + c;

//Instruction sequence2
sum = b + c;
sum = a + sum;

//Instruction sequence3
sum = a + c;
sum = sum + b;
```

The above situations do not seem to affect the results, but the following code will produce different results:

```
sum= a + b +sum; print(
sum);
```

The quality after compilation is as follows:

```
//Instruction sequence1
sum = a + b;
sum = sum + sum;

//Instruction sequence2
sum = b + sum;
sum = a + sum;

//Instruction sequence3
sum = a + sum;
sum = sum + b;
```

Obviously, after compiling into assembly instructions, the three situations will get different results!!

In order to prevent out-of-order problems during compilation, it is necessary to explicitly add memory barriers between instructions, such as:

```
sum = a + b;
__COMPILE_MEMORY_BARRIER__;
sum = sum + c;
```

The above__COMPILE_MEMORY_BARRIER__has different implementations in different compilers, some compiler implementations are as follows:

```
// C11 / C++11
atomic_signal_fence(memory_order_acq_rel);

// Microsoft Visual C++
_ReadWriteBarrier();

// GCC
__sync_synchronize();

// GNU
asmvolatile(""":::"memory"); __asm__ __volatile__ (""":::
"memory");

// Intel ICC
__memory_barrier();
```

In addition, there are**combined barriers**,which combine different types of barriers into other operations (such as load, store, atomic increment, atomic comp barriers are added before or after them. It is worth mentioning that combined barriers are related to the CPU architecture, and will be compiled into different

and also rely on hardware memory ordering guarantees.

1.4.5.2 Runtime Memory Barriers

The above explains the memory disorder problem at compile time, and the following will explain the memory disorder problem at runtime.

Early processors were**in-order processors**.If there is no out-of-order problem at compile time, this type of processor can ensure that the processing order is the programmer.

In the era of modern multi-core processors, there are many**out-of-order processors**.The order in which the processor actually executes instructions is determ order written by the programmer. The results of the instruction execution are written to the register file only after the execution results of all earlier instruction the register file (the execution results**are reordered**to make the execution appear to be orderly).

In an out-of-order multiprocessor architecture, if there is no runtime memory barrier mechanism, many unexpected execution results will occur. Here is a speci

Assume that there are memory variables`x`and`f`,their values are initialized to 0, both processor #1 and processor #2 can access them, and the execution i

Processor #1:

```
while(f ==0); print(x);
```

Processor #2:

```
x = 42;
f  =  1;
```

One of the situations might be that the expected`x`value output by processor #1 is 42. However, this is not the case. Since processor #2 may be executing ou `x = 42`,and the value output by processor #1 is 0 instead of 42. Similarly, processor #1 may output`x`the value first and then execute`while`the statement, In order to avoid unexpected results caused by out-of-order execution, a runtime memory barrier can be added between the two processor instructions:

Processor #1:

```
while(f ==0);
_RUNTIME_MEMORY_BARRIAR_;//Add memory barriers to ensurefThe value can read the latest value of other processors, and then executeprint(x)
print(x);
```

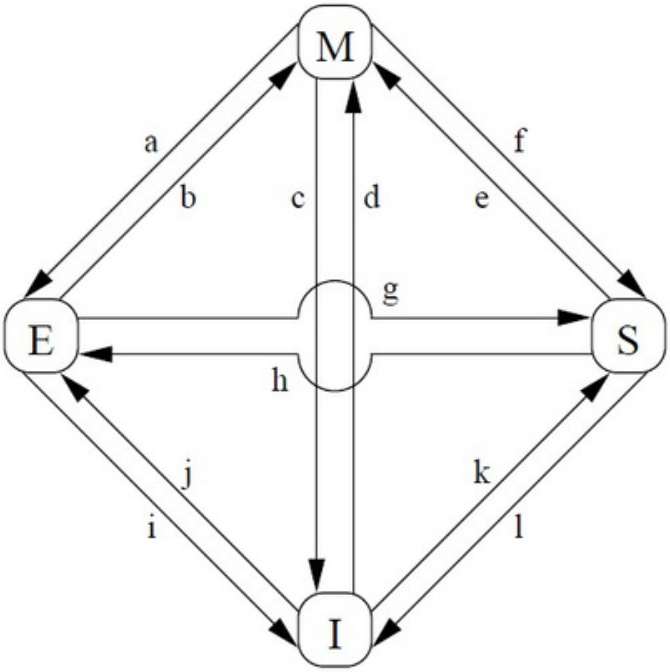
Processor #2:

```
x =42;
_RUNTIME_MEMORY_BARRIAR_;//Add memory barriers to ensurexVisible to other processors, it will be executedf=1 f =1;
```

The above`_RUNTIME_MEMORY_BARRIAR_`is a representation of the runtime memory barrier, which actually has different implementations on different hardware arc later.

At the hardware level, there are L1, L2, L3 caches, Store Buffers, and multi-core multi-threading. In order to keep the memory orderly, many states (such as ME are defined. There are more than a dozen combined interaction states between them, and they are related to the CPU hardware architecture. Obviously, it will directly access and manipulate these states.

> **The MESI protocol**is an invalidation-based cache consistency protocol and is the most commonly used protocol that supports write-back cache. It is often memory of multi-core CPUs.
> The basic states of the MESI protocol:**Modified**,**Exclusive**,**Shared**,and**Invalid**.
> MESI protocol messages: Read, Read Response, Invalidate, Invalidate Acknowledge, Read Invalidate, Writeback. The basic state transition of the MESI protocol is as follows:



Each basic state corresponds to a different meaning, but I will not elaborate on it here.

Other protocols similar to MESI include: Coherence protocol, MSI protocol, MOSI protocol, MOESI protocol, MESIF protocol, MERSI protocol, etc.

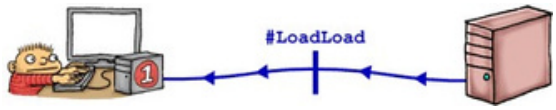
For more details about MESI, please refer to:

- MESI protocol
- Memory Barriers: a Hardware View for Software Hackers

So, smart people (such asDoug Lea) simplified these states and message passing mechanisms and combined them into 4 commonly used combined barriers ordering. Different CPUs have specific instructions, and these four can better match the instructions of real CPUs, although they are not completely matched. M combination of multiple types to achieve a specific effect.

Load Barrierand**Store Barrier**came into being. Inserting a Load Barrier before an instruction can invalidate the data in the cache and force the data to be rel Barrier is inserted after an instruction, the latest data in the cache can be written to the main memory so that it can be visible to other processor threads.
After arranging and combining Load Barrier and Store Barrier, four instructions can be formed:

- **LoadLoad**:It can prevent the disorder of read operations before and after the barrier caused by reordering.

  After adding the LoadLoad barrier, even if the CPU accesses it out of order, it will not jump before or after the LoadLoad barrier.

  Application examples:

```
if(IsValid)                 //  Load and testIsValid
{
    LOADLOAD_FENCE();       //  LoadLoadThe barrier prevents reordering between two loads.ValueBefore the data to be read by subsequent read operations is accessed, ensure thatIsValidThe data to be read before is
    returnValue;            //  read.Value
}
```

- **StoreStore**:It can prevent the disorder of write operations before and after the barrier caused by reordering.

  Application examples:

```
Value = x;                  //  WriteValue
STORESTORE_FENCE();         //  StoreStoreThe barrier prevents reordering between two writes.IsValidBefore subsequent write operations are performed, ensureValueWrite operations are visible to other
IsValid =1;                 //  processors.IsValid
```

- **LoadStore**:It can prevent the reordering of load operations before the barrier and store operations after the barrier. Application examples:

```
if(IsValid)                 //  Load and testIsValid
{
    LOADSTORE_FENCE();      //  LoadStoreThe barrier prevents reordering between loads and writes.ValueBefore subsequent write operations are flushed, ensureIsValidThe data to be read has been read.
```

```
    Value = x;                  //WriteValue
}
```

- **StoreLoad**:It can prevent the reordering of write operations before the barrier and load operations after the barrier. In most CPU architectures, it is a un other three memory barriers, but it also has the largest overhead. Application examples:

```
Value = x;              //  WriteValue
STORELOAD_FENCE();      //  existIsValidBefore all subsequent read operations are performed, ensureValueWrites to are visible to all
if(IsValid)             //  processors. Load and detectIsValid
{
    return1;
}
```

In the**Symmetric Multiprocessing (SMP)**micro-architecture, the memory access consistency model can be divided into:

- **Sequential consistency**:All read and write operations are sequential.
- **Relaxed consistency**:Loose consistency (or understood as partial consistency), Load after Load, Store after Store, Store after Load, and Load after Store **Weak**
- **consistency**:Weak consistency, all read and write operations may cause reordering unless there is an explicit memory barrier.

The following figure is a table of reordering of some common CPU architectures in different states:

Memory ordering in some architectures[8][9]

Type	Alpha	ARMv7	MIPS	RISC-V		PA-RISC	POWER	SPARC			x86 [a]	AMD64	IA-64	z/Architectu
				WMO	TSO			RMO	PSO	TSO				
Loads can be reordered after loads	Y	Y	depend on implementation	Y		Y	Y	Y					Y	
Loads can be reordered after stores	Y	Y		Y		Y	Y	Y					Y	
Stores can be reordered after stores	Y	Y		Y		Y	Y	Y	Y				Y	
Stores can be reordered after loads	Y	Y		Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic can be reordered with loads	Y	Y		Y			Y	Y					Y	
Atomic can be reordered with stores	Y	Y		Y			Y	Y	Y				Y	
Dependent loads can be reordered	Y													
Incoherent instruction cache pipeline	Y	Y		Y			Y	Y	Y	Y	Y		Y	

Runtime memory barriers have different implementations on different hardware architectures. The following lists the implementations of common architecture

```
// x86, x86-64
lfence (asm),void_mm_lfence(void)//Read barrier sfence (asm),void
_mm_sfence(void)//Write barrier mfence (asm),void_mm_mfence(
void)//Read and write barrier


// ARMv7
dmb (asm)// Data Memory Barrier,Data memory barrier
dsb (asm)// Data Synchronization Barrier,Data synchronization barrier isb (asm)//
Instruction Synchronization Barrier,Instruction synchronization barrier


//   POWER
dcs   (asm)

//PowerPC
sync (asm)

// MIPS
sync (asm)

//   Itanium
mf  (asm)
```

Memory barriers are a broad topic. Due to space and subject limitations, we cannot fully present its technology and mechanisms. However, we can recommen

- Memory ordering
- Memory Barriers Are Like Source Control Operations
- Understanding Memory Barrier from a hardware perspective

1.4.5.3 UE Memory Barrier

UE's memory barriers are encapsulated inFGenericPlatformMisc and its subclasses. The implementation of common operating systems is posted below:

```
structFGenericPlatformMisc {

    (.....)

    /**
      * Enforces strict memory load/store ordering across the memory barrier call.
      * /
    static void MemoryBarrier();
```

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

// Windows
structFWindowsPlatformMisc: public FGenericPlatformMisc {

        (.....)

        static void MemoryBarrier() {

                _mm_sfence();
        }

        (.....)
};
#if    WINDOWS_USE_FEATURE_PLATFORMMISC_CLASS typedef
        FWindowsPlatformMisc FPlatformMisc;
#endif

// Android
structFAndroidMisc: public FGenericPlatformMisc {

        (.....)

        static void MemoryBarrier() {

                __sync_synchronize();
        }

        (.....)
};
#if    !PLATFORM_LUMIN
        typedefFAndroidMisc            FPlatformMisc;
#endif

// Apple
structFApplePlatformMisc: public FGenericPlatformMisc {

        (.....)

        static void MemoryBarrier() {

                __sync_synchronize();
        }

        (.....)
};

// Linux
structFLinuxPlatformMisc: public FGenericPlatformMisc {

        (.....)

        static void MemoryBarrier() {

                __sync_synchronize();
        }

        (.....)
};
#if    !PLATFORM_LUMIN
        typedefFLinuxPlatformMisc              FPlatformMisc;
#endif
```

Except for Windows, which uses x86 architecture instructions, other systems use GCC's memory barrier instructions. What's strange is that Windows uses runti seem to use compile-time memory barriers. I was confused at first, but later found the answer in the referenceMemory ordering:

> Compiler support for hardware memory barriers
> Some compilers support builtins that emit hardware memory barrier instructions:
>
> - GCC, version 4.4.0 and later, has__sync_synchronize.
> - Since C11 and C++11 anatomic_thread_fence()command was added. The
> - Microsoft Visual C++ compiler hasMemoryBarrier().
> - Sun Studio Compiler Suite has__machine_r_barrier,__machine_w_barrierand __machine_rw_barrier .

That is to say, the compile-time memory barriers of some compilers will also trigger the hardware (runtime) memory barriers, including that of the GCC compi

With UE's polymorphic encapsulation of system platforms, callers do not need to pay attention to which system it is. TheyFPlatformMisc::MemoryBarrier()can barriers to the code without any brain calls. The sample code is as follows:

```
// Engine\Source\Runtime\RenderCore\Private\RenderingThread.cpp
```

```
void RenderingThreadMain(FEvent* TaskGraphBoundSyncEvent) {

	LLM_SCOPE(ELLMTag::RenderingThreadMemory);

	ENamedThreads::Type RenderThread = ENamedThreads::Type(ENamedThreads::ActualRenderingThread);

	ENamedThreads::SetRenderThread(RenderThread);
	ENamedThreads::SetRenderThread_Local(ENamedThreads::Type(ENamedThreads::ActualRenderingThread_Local));

	FTaskGraphInterface::Get().AttachToThread(RenderThread);

	//Add system memory barriers
	FPlatformMisc::MemoryBarrier();

	// Inform main thread that the render thread has been attached to the taskgraph and is ready to receive tasks if( TaskGraphBoundSyncEvent !=NULL) {


		TaskGraphBoundSyncEvent->Trigger();
	}

	// set the thread back to real time mode
	FPlatformProcess::SetRealTimeMode();

#if	STATS
	if(FThreadStats::WillEverCollectData()) {

		FThreadStats::ExplicitFlush();// flush the stats and set update the scope so we don't flush again until a frame update, this helps preve
	}
#endif

	FCoreDelegates::PostRenderingThreadCreated.Broadcast();
	check(GIsThreadedRendering);
	FTaskGraphInterface::Get().ProcessThreadUntilRequestReturn(RenderThread);

	//Add system memory barriers
	FPlatformMisc::MemoryBarrier();

	check(!GIsThreadedRendering);
	FCoreDelegates::PreRenderingThreadDestroyed.Broadcast();

#if	STATS
	if(FThreadStats::WillEverCollectData()) {

		FThreadStats::ExplicitFlush();// Another explicit flush to clean up the ScopeCount established above for any stats lingering since the l
	}
#endif

	ENamedThreads::SetRenderThread(ENamedThreads::GameThread);
	ENamedThreads::SetRenderThread_Local(ENamedThreads::GameThread_Local);

	//Add system memory barriers
	FPlatformMisc::MemoryBarrier();
}
```

From this we can see that UE directly encapsulates and uses runtime memory barriers, but does not encapsulate compile-time memory barriers.

In addition to the system's memory barrier, UE also encapsulates and uses the memory barrier of the graphics API layer:

```
// Direct3D / Metal
FPlatformMisc::MemoryBarrier();

// OpenGL
glMemoryBarrier(Barriers);

// Vulkan
typedef structVkMemoryBarrier{
	(.....)
} VkMemoryBarrier;

typedef structVkBufferMemoryBarrier{
	(.....)
} VkBufferMemoryBarrier;

typedef structVkImageMemoryBarrier{
	(.....)
} VkImageMemoryBarrier;
```

1.4.6 Engine startup process

Readers who have learned programming for Windows and other operating systems should know that for each application, there are different entrances in diff program entrance for Windows isWinMain,while for Linux isMain.The following will take the Windows PC platform entrance as the analysis process, and its

```
// Engine\Source\Runtime\Launch\Private\Windows\LaunchWindows.cpp

int32 WINAPIWinMain(_In_ HINSTANCE hInInstance, _In_opt_ HINSTANCE hPrevInstance, _In_char*, _In_ int32 nCmdShow ) {

	TRACE_BOOKMARK(TEXT("WinMain.Enter"));

	SetupWindowsEnvironment();

	int32 ErrorLevel 		=0;
	hInstance 		= hInInstance;
	constTCHAR* CmdLine = ::GetCommandLineW();

	//Processing command lines
	if( ProcessCommandLine() ) {

		CmdLine = *GSavedCommandLine;
	}

	if( FParse::Param( CmdLine, TEXT("unattended") ) ) {

		SetErrorMode(SEM_FAILCRITICALERRORS | SEM_NOGPFAULTERRORBOX | SEM_NOOPENFILEERRORBOX);
	}

	(......)

	//Depending on whether there is exception handling and error level, you will enter different entrances, but you will eventually enterGuardedMainfunction.
#if	UE_BUILD_DEBUG
	if(true&& !GAlwaysReportCrash )
#else
	if( bNoExceptionHandler || (FPlatformMisc::IsDebuggerPresent() && !GAlwaysReportCrash ))
#endif
	{
		//EnterGuardedMainMain Entrance ErrorLevel =
		GuardedMain(CmdLine);
	}
	else
	{
		(......)

		 {
			GIsGuarded =1; //Enter
			GuardedMainMain Entrance
			ErrorLevel = GuardedMainWrapper( CmdLine ); GIsGuarded
			=0;
		}

		(......)
	}

	//Exit Program
	FEngineLoop::AppExit();

	(......)

	returnErrorLevel;
}
```

The above main branches will eventually enterGuardedMianthe interface. The code (excerpt) is as follows:

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

int32GuardedMain(constTCHAR* CmdLine ) {

	(......)

	//Ensure that you can callEngineExit
	structEngineLoopCleanupGuard {

		~EngineLoopCleanupGuard() {

			EngineExit();
		}
	} CleanupGuard;

	(......)

	//Engine Pre-Initialization
	int32 ErrorLevel = EnginePreInit(CmdLine); if( ErrorLevel !=0||
	IsEngineExitRequested() ) {

		returnErrorLevel;
	}

	{
		(......)
```

```
#ifWITH_EDITOR
            if(GIsEditor)
            {
                //Editor initialization
                ErrorLevel =        EditorInit(GEngineLoop);
            }
            else
#endif
            {
                //Engine (non-editor) initialization
                ErrorLevel = EngineInit();
            }
        }

        (.....)

        while(      !IsEngineExitRequested() )
        {
            //  Engine frame update
            EngineTick();
        }

#if     WITH_EDITOR
        if( GIsEditor           )
        {
            //Editor Exit
            EditorExit();
        }
#endif
        returnErrorLevel;
}
```

It is not difficult to see that this logic mainly has four steps: engine pre-initialization (EnginePreInit), engine initialization (EngineInit), engine frame update (Eng

1.4.6.1 Engine Pre-initialization

UE engine pre-initialization mainly involves a lot of initialization and basic core-related module work during the startup page.

Its main code is as follows:

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

int32EnginePreInit(constTCHAR* CmdLine ) {

        //CallGEngineLoopPre-initialization.
        int32 ErrorLevel = GEngineLoop.PreInit(CmdLine);

        return( ErrorLevel );
}


// Engine\Source\Runtime\Launch\Private\LaunchEngineLoop.cpp

int32FEngineLoop::PreInit(constTCHAR* CmdLine) {

        //Start the progress bar of the small window
        constint32 rv1 = PreInitPreStartupScreen(CmdLine); if(rv1 !=0)

        {
            PreInitContext.Cleanup(); return
            rv1;
        }

        constint32 rv2 = PreInitPostStartupScreen(CmdLine); if(rv2 !=0)

        {
            PreInitContext.Cleanup(); return
            rv2;
        }

        return0;
}
```

The pre-initialization phase will initialize the random seed, load the CoreUObject module, start the FTaskGraphInterface module and attach the current game t modules of UE (Engine, Renderer, SlateRHIRenderer, Landscape, TextureCompressor, etc.), which isLoadPreInitModulescompleted by:

```
void    FEngineLoop::LoadPreInitModules()
{
#if    WITH_ENGINE
        FModuleManager::Get().LoadModule(TEXT("Engine"));
        FModuleManager::Get().LoadModule(TEXT("Renderer"));
        FModuleManager::Get().LoadModule(TEXT("AnimGraphRuntime"));


        FPlatformApplicationMisc::LoadPreInitModules();

#if    !UE_SERVER
        if(!IsRunningDedicatedServer() {                    )

                if(!GUsingNullRHI)
                {
                        // This needs to be loaded before InitializeShaderTypes is called
                        FModuleManager::Get().LoadModuleChecked<ISlateRHIRendererModule>("SlateRHIRenderer");
                }
        }
#endif

        FModuleManager::Get().LoadModule(TEXT("Landscape"));
        FModuleManager::Get().LoadModule(TEXT("RenderCore"));

#if    WITH_EDITORONLY_DATA
        FModuleManager::Get().LoadModule(TEXT("TextureCompressor"));
#endif

#endif// WITH_ENGINE

#if(WITH_EDITOR && !(UE_BUILD_SHIPPING || UE_BUILD_TEST))
        FModuleManager::Get().LoadModule(TEXT("AudioEditor"));
        FModuleManager::Get().LoadModule(TEXT("AnimationModifiers"));
#endif
}
```

Then, we will process the configuration log, loading progress information, TLS (thread-local scope) cache of memory allocator, setting some global states, proc basic core modules (FModuleManager, IFileManager, FPlatformFileManager, etc.). Another important point is to process the game thread, set the currently exe (main thread) and record the thread ID. This code is as follows:

```
int32FEngineLoop::PreInitPreStartupScreen(constTCHAR* CmdLine) {

        (.....)

        GGameThreadId = FPlatformTLS::GetCurrentThreadId();
        GIsGameThreadIdInitialized =true;

        FPlatformProcess::SetThreadAffinityMask(FPlatformAffinity::GetMainGameMask());
        FPlatformProcess::SetupGameThread();

        (.....)
}
```

Then set the Shader source code directory mapping, process the network token, initialize some basic modules (FCsvProfiler, AppLifetimeEventCapture, FTracing pool and a specified number of threads based on whether the platform supports multi-threading:

```
int32FEngineLoop::PreInitPreStartupScreen(constTCHAR* CmdLine) {

        (.....)

        if(FPlatformProcess::SupportsMultithreading()) {

                {
                        TRACE_THREAD_GROUP_SCOPE("IOThreadPool"
                        SCOPED_BOOT_TIMING("GIOThreadPool->Create");
                        GIOThreadPool = FQueuedThreadPool::Allocate();
                        int32 NumThreadsInThreadPool = FPlatformMisc::NumberOfIOWorkerThreadsToSpawn(); if
                        (FPlatformProperties::IsServerOnly()) {


                                NumThreadsInThreadPool =2;
                        }
                        verify(GIOThreadPool->Create(NumThreadsInThreadPool,96*1024, TPri_AboveNormal));
                }
        }

        (.....)
}
```

Then initialize or process UGameUserSettings, Scalability, rendering thread (if turned on), FConfigCacheIni, FPlatformMemory, game physics, RHI, RenderUtils,

In the late pre-initialization stage, the engine processes modules such as SlateRenderer, IProjectManager, IInstallBundleManager, MoviePlayer, PIE preview dev

1.4.6.2 Engine Initialization

The engine initialization is divided into two modes: editor and non-editor. The non-editor executesFEngineLoop::Init,and the editor executesEditorInit+ the initialization logic executed by the non-editor.

The process of engine initializationFEngineLoop::Initis completed by, and its main process is as follows:

- Create the corresponding game engine instance according to the configuration file and store itGEngine.The instance will be used extensively laterGEn
- Determine whether to create an EngineService instance based on whether multithreading is supported.
- implementGEngine->Start().
- Load the Media, AutomationWorker, AutomationController, ProfilerClient, SequenceRecorder, SequenceRecorderSections modules.
- Enable thread heartbeat FThreadHeartBeat.
- Register the external profiler FExternalProfiler.

1.4.6.3 Engine frame update

The process of engine initializationFEngineLoop::Tick is completed by, and its main process is as follows:

- Enable thread and thread hook heartbeat.
- Updates objects (FTickableObjectRenderThread instances) that the rendering module can update every frame.
- Profiler (FExternalProfiler) frame synchronization.
- Execute the console's callback interface.
- FlushRenderingCommands. If a separate rendering thread is not enabled, the rendering commands will be executed on the game thread, and then called command queue is submitted for drawing. A rendering fence (FRenderCommandFence) will be added at the end.

```
// Engine\Source\Runtime\RenderCore\Private\RenderingThread.cpp

void FlushRenderingCommands(boolbFlushDeferredDeletes) {

    (......)

    if(!GIsThreadedRendering
        &&!FTaskGraphInterface::Get().IsThreadProcessingTasks(ENamedThreads::GameThread) !
        && FTaskGraphInterface::Get().IsThreadProcessingTasks(ENamedThreads::GameThread_Local))
    {
        FTaskGraphInterface::Get().ProcessThreadUntilIdle(ENamedThreads::GameThread);
        FTaskGraphInterface::Get().ProcessThreadUntilIdle(ENamedThreads::GameThread_Local);
    }

    ENQUEUE_RENDER_COMMAND(FlushPendingDeleteRHIResourcesCmd)(
        [bFlushDeferredDeletes](FRHICommandListImmediate& RHICmdList)
    {
        RHICmdList.ImmediateFlush(
            bFlushDeferredDeletes ?
            EImmediateFlushType::FlushRHIThreadFlushResourcesFlushDeferredDeletes                    :
            EImmediateFlushType::FlushRHIThreadFlushResources);
    });

    AdvanceFrameRenderPrerequisite();

    FPendingCleanupObjects* PendingCleanupObjects = GetPendingCleanupObjects();

    FRenderCommandFence     Fence;
    Fence.BeginFence();
    Fence.Wait();

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

- Triggers the OnBeginFrame event.
- Flush the thread log.
- Use GEngine to refresh time and handle maximum frame rate.
- Traverse the current World of all WorlContexts and update the PrimitiveSceneInfo of the scenes in the World. It may be easier to understand if you paste

```
for(constFWorldContext& Context : GEngine->GetWorldContexts()) {

    UWorld* CurrentWorld = Context.World(); if
    (CurrentWorld)
    {
        FSceneInterface* Scene = CurrentWorld->Scene;
        ENQUEUE_RENDER_COMMAND(UpdateScenePrimitives)(
            [Scene](FRHICommandListImmediate& RHICmdList)
            {
                Scene->UpdateAllPrimitiveSceneInfos(RHICmdList);
```

```
                });
        }
}
```

- Process the beginning of the RHI frame.
- Call all scenesStartFrame.
- Handles performance analysis and data statistics.
- Handles per-frame tasks on the rendering thread.
- Handles world scale scaling (WorldToMetersScale).
- Updates the active platform's files.
- Processes Slate module input.
- GEngine's Tick event. This is the main frame update, and a lot of logic will be handled here. The following is UGameEngine::Tick the main process:
    - If the time interval is sufficient, refresh the Log.
    - Clears the closed game viewport.
    - Updates a subsystem.
    - Updated FEngineAnalytics and FStudioAnalytics modules. (If
    - Chaos is enabled) Update ChaosModule.
    - Handles WorldTravel frame updates.
    - Handles all World frame updates.
    - Update the skylight component (USkyLightComponent) and the reflection ball component (UReflectionCaptureComponent). These two componen Handle the player object (ULocalPlayer).
    - Handles level streaming.
    - Updates all updateable objects. Here, the object updated is FTickableGameObject.
    - Update the GameViewport. Handles a window in windowed mode. Draw the Viewport.
    - Updated IStreamingManager and FAudioDeviceManager modules.
    - Update rendering related modules such as GRenderingRealtimeClock, GRenderTargetPool, FRDGBuider, etc.
- Handles asynchronous compilation results of GShaderCompilingManager.
- Handles asynchronous tasks for GDistanceFieldAsyncQueue (distance field asynchronous queue).
- Parallel processing of Slate-related task logic.
- Handle replicated properties (ReplicatedProperties).
- Use FTaskGraphInterface to handle parallel tasks stored in ConcurrentTask.
- Waiting for unresolved rendering tasks in the rendering queue. Maybe I didn't understand it correctly, so I'll post the code:

```
ENQUEUE_RENDER_COMMAND(WaitForOutstandingTasksOnly_for_DelaySceneRenderCompletion)(
    [](FRHICommandList& RHICmdList) {

        QUICK_SCOPE_CYCLE_COUNTER(STAT_DelaySceneRenderCompletion_TaskWait);
        FRHICommandListExecutor::GetImmediateCommandList().ImmediateFlush(EImmediateFlushType::WaitForOutstandingTasksOnly);
    });
```

- Update AutomationWorker module.
- Update RHI module.
- Handles frame count (GFrameCounter) and total frame update time (TotalTickTime).
- Collect objects that need to be cleaned up in the next frame.
- Handle the frame end synchronization event (FFrameEndSync).
- Updated Ticker, FThreadManager, and GEngine's TickDeferredCommands.
- Triggers OnEndFrame on the game thread.
- Trigger the EndFrame event in the rendering module.

1.4.6.4 Engine Exit

When the engine exits, if it is not in editor mode, it directly returns the ErrorLevel value; if it is in editor mode, it executes the EditorExit logic. Its main work is t engine module:

```
// Engine\Source\Editor\UnrealEd\Private\UnrealEdGlobals.cpp
```

```
void EditorExit()
{
	TRACE_CPUPROFILER_EVENT_SCOPE(EditorExit);

	GLevelEditorModeTools().SetDefaultMode(FBuiltinEditorModes::EM_Default);
	GLevelEditorModeTools().DeactivateAllModes();// this also activates the default mode

	// Save out any config settings for the editor so they don't get lost GEditor->SaveConfig();

	GLevelEditorModeTools().SaveConfig();

	// Clean up the actor folders singleton
	FActorFolders::Cleanup();

	// Save out default file directories
	FEditorDirectories::Get().SaveLastDirectories();

	// Allow the game thread to finish processing any latent tasks.
	// Some editor functions may queue tasks that need to be run before the editor is finished.
	FTaskGraphInterface::Get().ProcessThreadUntilIdle(ENamedThreads::GameThread);

	// Cleanup the misc editor
	FUnrealEdMisc::Get().OnExit();

	if( GLogConsole ) {

		GLogConsole->Show(false);
	}

	delete  GDebugToolExec;
	GDebugToolExec=NULL;
}
```

-
-
-
-
-

References

- Unreal Engine 4 Sources
- Unreal Engine 4 Documentation
- Rendering and Graphics
- Materials
- Graphics Programming
- Unreal Engine
- Unreal Engine 4 Rendering
- List of Unreal Engine games
- Dissecting Unreal Engine's Hyper-Realistic Human Rendering Techniques Part 1 - Overview and Skin Rendering
- Explore ray tracing technology and its implementation in UE4
- In-depth understanding of GPU hardware architecture and operation mechanism
- A First Look at Unreal Engine 5
- "The Elephant is Invisible-A Brief Analysis of Unreal Engine Programming"
- Fang Yanliang - Analysis of Unreal 4 Rendering System Architecture
- UE4 Render System Sheet
- The End of Moore's Law and the Return of Cleverness
- FrameGraph Extensible Rendering Architecture in Frostbite
- Unreal Engine 4 Guide for Unity Developers

- Unreal Engine Math
- Neon
- Streaming SIMD Extensions
-
- Floating-point unit
- Inside UE4
- Exploring in UE4
- c++ lambda
- Why do we need tangent space?
- Survey of Efficient Representations for Independent Unit Vectors
- UE4 memory management (1)
- UE4 source code analysis: MallocBinned
- UE4 memory optimization knowledge points summary
- Game Design Patterns - Memory Pool Management
- Garbage Collection Algorithms and Implementations
- Game Engine Architecture
- Memory stomp allocator for Unreal Engine 4
- Intel® Threading Building Blocks
- Adding memory tracking to Unreal Engine 4
- Memory ordering
- Memory barrier
- Introduction to Memory Barrier
- Understanding Memory Barrier from a hardware perspective
- Using JDK 9 Memory Order Modes
- Memory Barriers Are Like Source Control Operations
- Memory Barriers
- Memory Barriers: a Hardware View for Software Hackers
- MESI protocol
- User mode and kernel mode
- UE4 Garbage Collection
- UE4 Garbage Collection (Part 2) GC Cluster
- Analysis of Unreal 4 Garbage Collection

https://github.com/pe7yu