# P1 UE4 的入口Main函数

1. 在WinMain(在LaunchWindows.cpp) 入口函数里面调用GuardedMain(在Launch.cpp)函数，下面主要介绍这个函数里面的东西
2. 用Delegate的方式进行了预初始化：  
   FCoreDelegates::GetPreMainInitDelegate().Broadcast();
3. 定义一个结构体用于出作用域后自动析构，调用EngineExit()
4. 进行引擎预初始化：  
   int32 ErrorLevel = EnginePreInit( CmdLine );
5. 判断是否为EDITOR，如是则调用EditorInit(*GEngineLoop*)，否则调EngineInit()，其实两者内部都会调*GEngineLoop*.Init()，只是EditorInit还初始化了编辑器的一些东西
6. 随后进入循环，调用EngineTick()
7. 当检测到IsEngineExitRequested()时，退出循环，最后调用EditorExit()，且出作用域自动调EngineExit()

# P2 UE4的引擎循环FEngineLoop::Tick()函数

FEngineLoop::Tick()(在LaunchEngineLoop.cpp)函数是GuardedMain(在Launch.cpp)函数里面的EngineTick()调用的，相当于每帧都会执行的一些操作。主要实现了以下操作：

1. 内存跟踪、LLM和一些Ensure和Check以及Log等工作
2. FCoreDelegates进行开始这一帧的广播：  
   FCoreDelegates::OnBeginFrame.Broadcast();
3. 当考虑最大tick率的时候更新FApp的当前时间和时间差：  
   GEngine->UpdateTimeAndHandleMaxTickRate();
4. 这是一个奇怪的函数，没看懂为啥：  
   GEngine->SetGameLatencyMarkerStart(CurrentFrameCounter);
5. 对每个WorldContext的Scene的所有primitive的状态进行更新：  
   Scene->UpdateAllPrimitiveSceneInfos(RHICmdList);
6. 开始RHI渲染线程，此函数根据不同平台调用不同的虚函数：  
   BeginFrameRenderThread(RHICmdList, CurrentFrameCounter);
7. 开始新的一帧，在内部把InternalFrameIndex++，有一个循环，把每一个PrimitiveComponent的VelocityData进行更新，并把bNeedsUniformBufferUpdate设为true，要在每个片元移动后重新创建UniformBuffer：  
   Scene->StartFrame();
8. 忽略动态分辨率在一帧开始时的操作（仅Engine）：  
   GEngine->EmitDynamicResolutionEvent(EDynamicResolutionStateEvent::BeginFrame);
9. 更新性能监控、内存跟踪、STAT等
10. 计算FPS：当前FPS与上一个FPS进行加权：  
    GAverageMS = GAverageMS \* 0.75f + FrameTime \* 0.25f;
11. 在渲染线程处理一些逐帧任务：  
    FlushPendingDeleteRHIResources\_RenderThread();
12. 根据情况判断是否停止tick以及停止渲染，其值存给bIdleMode  
    bIdleMode = ShouldUseIdleMode();
13. 允许在每次运动控制器输入之前设置一次world-to-meters  
    extern ENGINE\_API float GNewWorldToMetersScale;
14. 处理一些平台文件的操作：  
    FPlatformFileManager::Get().TickActivePlatformFile();
15. 轮询一些游戏控制器或者面板输入：  
    SlateApp.PollGameDeviceState();
16. Media做一些引擎之前的工作：  
    MediaModule->TickPreEngine();
17. 主要的游戏和引擎Tick函数，以后会细致分析这个函数：  
    GEngine->Tick(FApp::GetDeltaTime(), bIdleMode);  
    GEngine是UEngine\*类型，主要有两个派生类：UEditorEngine和UGameEngine，根据多态的规则来确定调哪个Tick函数，两者Tick函数相似，最终都会调用Viewport->Draw()来绘制
18. 处理与movie的冲突，因为刚好在GEngine->Tick里面有可能会有用户代码，里面可能会涉及到movie的操作，所以紧随其后，等待movie操作完毕，主要根据判断是否有PreLoadScreen来看是先加载movie还是PreLoadScreen
19. 处理异步ShaderCompiling：  
    GShaderCompilingManager->ProcessAsyncResults(true, false);
20. 处理异步DistanceFieldAsyncQueue：  
    GDistanceFieldAsyncQueue->ProcessAsyncTasks();
21. Media做一些Slate之前的工作：  
    MediaModule->TickPreSlate();
22. 做一些Slate的输入的一些操作，主要是在网络同步之前做的：  
    ProcessLocalPlayerSlateOperations();
23. 用UDemoNetDriver构建虚拟网络驱动，用于记录或者回放。
24. 处理automation controller（仅Editor）和worker（仅Engine）  
    FModuleManager::GetModuleChecked<IAutomationControllerModule>(AutomationController).Tick();  
    FModuleManager::GetModuleChecked<IAutomationWorkerModule>(AutomationWorkerModuleName).Tick();
25. 处理RHI的操作，根据不同的平台调不同的虚函数：  
    RHITick( FApp::GetDeltaTime() ); // Update RHI.
26. 累加全局的帧计数，忽略前面的几帧不计入TotalTickTime：  
    GFrameCounter++;
27. 找到下一帧需要被清理的对象：  
    PendingCleanupObjects = GetPendingCleanupObjects();
28. 进行帧同步，即游戏线程和渲染线程的同步：  
    FrameEndSync.Sync( CVarAllowOneFrameThreadLag->GetValueOnGameThread() != 0 );
29. 处理一些关于core ticker, threads & deferred commands的操作，并且删除上一帧的删除对象PreviousPendingCleanupObjects
30. Media的一些后处理工作：  
    MediaModule->TickPostRender();
31. FCoreDelegates进行结束这一帧的广播：  
    FCoreDelegates::OnEndFrame.Broadcast();
32. 忽略动态分辨率在一帧结束时的操作（仅Engine）：  
    GEngine->EmitDynamicResolutionEvent(EDynamicResolutionStateEvent::EndFrame);
33. 结束RHI渲染线程：  
    EndFrameRenderThread(RHICmdList, CurrentFrameCounter);
34. 这是一个奇怪的函数，没看懂为啥：  
    GEngine->SetGameLatencyMarkerEnd(CurrentFrameCounter);
35. 最后设置一些STAT的参数，如CPU utilization、UObject count

# P3 UE4的UEditorEngine::Tick和UGameEngine::Tick函数

这两个函数主要是在LaunchEngineLoop.cpp的FEngineLoop::Tick()的GEngine->Tick通过多态调用了，还有其他的重载如：UUnrealEdEngine::Tick和UDisplayClusterEditorEngine::Tick以及UDisplayClusterGameEngine::Tick。此处主要分析UGameEngine::Tick并以此为基础来看与其他重载的不同之处。  
UGameEngine::Tick主要有以下实现：

1. 检查输入DeltaSeconds的有效性
2. 清除已经关闭的Viewports（三种情况下跳过：commandlet，dedicated server，headless）：  
   CleanupGameViewport();
3. 如果当前没有Viewports则Exit()（headless下跳过）
4. 根据DeltaSeconds来设置是否降低细节，其内部主要是计算了当前的framerate来与阈值比较：  
   GameViewport->SetDropDetail(DeltaSeconds);
5. 处理一些子系统，如：ProcessAsyncLoading，*EngineAnalytics和StudioAnalytics*
6. 保存当前对于Gworld的WorldList中的元素(WorldContext)为OriginalGWorldContext
7. 进入WorldList的循环，对每个元素执行以下操作：
8. 将元素存到GWorld：  
   GWorld = Context.World();
9. 处理travel以及一些NetNames（关于漫游和网络同步的一些操作）：  
   TickWorldTravel(Context, DeltaSeconds);
10. 对World实际进行Tick，实际上是对Levels进行tick，这个函数之后会详细分析  
    Context.World()->Tick( LEVELTICK\_All, DeltaSeconds );
11. 更新SkyCapture和ReflectionCapture：  
    USkyLightComponent::UpdateSkyCaptureContents(Context.World());  
    UReflectionCaptureComponent::UpdateReflectionCaptureContents(Context.World());
12. 加载一次Player，在加载一次世界时仅仅运行一次：  
    GamePlayer->Exec( GamePlayer->GetWorld(), \*(FString("CAUSEEVENT ") + InitialExec), \*GLog );
13. 更新过渡的方式：（判断现在处于什么过渡方式：connecting，pause等，并决定是否更改）  
    UpdateTransitionType(Context.World());
14. 根据需求来阻止异步加载：  
    BlockTillLevelStreamingCompleted(Context.World());
15. 更新LevelStreaming：  
    Context.World()->UpdateLevelStreaming();
16. 做一些切换地图的操作：  
    ConditionalCommitMapChange(Context);
17. 对tickable的对象进行处理：  
    FTickableGameObject::TickObjects(nullptr, LEVELTICK\_All, false, DeltaSeconds);
18. 将上面存储的OriginalGWorldContext又存到GWorld：  
    GWorld = GetWorldContextFromHandleChecked(OriginalGWorldContext).World();
19. 对Meida进行处理：  
    MediaModule->*TickPostEngine*();
20. 对Viewport进行Tick：（里面用delegate的方式进行广播）：  
    GameViewport->Tick(DeltaSeconds);
21. 隐藏闪屏并打开游戏窗口
22. Render所有的viewport并且做一些渲染后的操作：（此函数后续会进一步分析）：  
    RedrawViewports();（内部实际调用：GameViewport->Viewport->Draw）  
    GetRendererModule().PostRenderAllViewports();
23. 对资源流做一些处理：  
    IStreamingManager::Get().Tick( DeltaSeconds );
24. 更新音频：  
    GameAudioDeviceManager->UpdateActiveAudioDevices(bIsAnyNonPreviewWorldUnpaused);
25. 给渲染线程发送指令：  
    （仅Editor）处理一些Editor后面的操作和资产注册

UEditorEngine::Tick与UGameEngine::Tick的不同之处（简称Editor和Game）：

1. Editor把GameViewport->Tick，Viewport->Draw(即Render)，GameViewport->SetDropDetail放在了WorldList的循环里面，而Game在外面；
2. Editor在没有viewport的时候调用的是EndPlayMap()，只是清理资源关闭了窗口，并没有完全退出主程序（编辑器还在），而Game则是直接退出主程；
3. Editor还有一个FEditorViewportClient::Tick，这个语句在一个Viewport的循环里面，对每一个视口主要做一个相机变换的更新。Game关于这个的实现暂时没找到
4. Editor还考虑了窗口被遮挡时的绘制，Game没有
5. 此外Editor还多了很多Game没有的操作，此处不赘述

# P4 UE4的关卡更新函数UWorld::Tick函数

此函数在LevelTick.cpp里面，主要负责关卡的更新，主要是在GEngine->Tick的内部进行调用。UWorld::Tick主要有以下实现：

1. 给render线程发送RHI指令BeginTickDraw：  
   FDrawEvent\* TickDrawEvent = BeginTickDrawEvent();
2. 进行广播WorldTick开始了：  
   FWorldDelegates::OnWorldTickStart.*Broadcast*(this, TickType, DeltaSeconds);
3. 游戏线程和其他线程的性能追踪器：  
   PerfTrackers->GetInGamePerformanceTracker((EInGamePerfTrackers)Tracker, EInGamePerfTrackerThreads::GameThread).Tick();  
   PerfTrackers->GetInGamePerformanceTracker((EInGamePerfTrackers)Tracker, EInGamePerfTrackerThreads::OtherThread).Tick();
4. XRSystem开始运作，此函数用于VR/AR/MR：  
   GEngine->XRSystem->OnStartGameFrame( GEngine->GetWorldContextFromWorldChecked( this ) )
5. 碰撞分析器开始运作：  
   Analyzer->*TickAnalyzer*(this);
6. 更新网络代码并且获取传入的数据包：  
   BroadcastTickDispatch(DeltaSeconds);  
   用传入的DeltaSeconds对一系列时间进行更新，主要有：  
   UWorld::RealTimeSeconds //不暂停，不缩放，总时间  
   UWorld::AudioTimeSeconds //暂停，不缩放，总时间  
   UWorld::DeltaTimeSeconds //缩放，时间差  
   UWorld::UnpausedTimeSeconds //不暂停，缩放，总时间  
   UWorld::TimeSeconds //暂停，缩放，总时间
7. 给高优先级和无缝切换的异步加载更多时间：  
   *ProcessAsyncLoading*(true, true, *GPriorityAsyncLoadingExtraTime* / 1000.0f);
8. 根据需求变换世界原点位置：  
   SetNewWorldOrigin(RequestedOriginLocation);
9. 更新子系统，如导航子系统：  
   NavigationSystem->Tick(DeltaSeconds);
10. 根据Ticktype，bIsPaused，NetDriver等情况定义一个bDoingActorTicks变量，决定后续的ActorTick，主要有以下操作：
11. 重设异步追踪和广播PreActorTick，并对Actor队列进行初始化：  
    ResetAsyncTrace();  
    FWorldDelegates::OnWorldPreActorTick.Broadcast(this, TickType, DeltaSeconds);
12. MovieSceneSequenceTick.*Broadcast*(DeltaSeconds);
13. 进入遍历LevelCollections的循环：
14. 从每个LevelCollection挑选合适的Level加入TArray<ULevel\*> LevelsToTick中
15. 通过一个辅助的RAII类来为某一个LevelCollection储存当前世界的信息，在构造函数里面存储，在析构函数里面又把这个暂存值还给当前的世界：  
    FScopedLevelCollectionContextSwitch LevelContext(i, this);
16. 对很多groups按顺序进行Tick，主要有七个group和一个标记group：  
    TG\_PrePhysics  
    TG\_StartPhysics  
    TG\_DuringPhysics  
    TG\_EndPhysics  
    TG\_PostPhysics  
    TG\_PostUpdateWork  
    TG\_LastDemotable  
    TG\_NewlySpawned
17. 在0-4和5-6之间插入了一些操作，这些操作只运行一次，且在source level collection里面运行：  
    处理一些潜在的操作：  
    CurrentLatentActionManager.ProcessLatentActions(nullptr, DeltaSeconds);
18. 对TimerManager进行更新：  
    GetTimerManager().Tick(DeltaSeconds);
19. 对用户自定义的Tickable的object进行Tick，这些对象往往在写游戏逻辑时需要补充Tick()和GetStatId()函数：  
    FTickableGameObject::TickObjects(this, TickType, bIsPaused, DeltaSeco
20. 更新每个playController的摄像机：  
    PlayerController->UpdateCameraManager(DeltaSeconds);
21. 根据player的位置，来更新LevelStreaming：  
    WorldComposition->*UpdateStreamingState*();
22. 广播ActorTick结束：  
    FWorldDelegates::OnWorldPostActorTick.*Broadcast*(this, TickType, DeltaSeconds);
23. PHYSX接口的相关操作，结束异步跟踪：  
    GPhysCommandHandler->Flush();  
    FinishAsyncTrace();
24. 更新网络相关的操作：  
    BroadcastTickFlush(RealDeltaSeconds); // note: undilated time is being used here  
    BroadcastPostTickFlush(RealDeltaSeconds); // note: undilated time is being used here
25. 更新SpeedTree wind，FXSystem：（不知道这个是啥）  
    Scene->UpdateSpeedTreeWind(TimeSeconds);  
    FXSystem->Tick(DeltaSeconds);
26. 垃圾回收，更新裁剪空间等其他操作：  
    GEngine->ConditionalCollectGarbage();  
    UpdateCullDistanceVolumes();
27. 给render线程发送指令EndTickDrawEvent：  
    EndTickDrawEvent(TickDrawEvent);

# P5 UE4的视口绘制函数FViewport::Draw

此函数在UnrealClient.cpp里面，功能是用ViewportClient->Draw的结果来更新视口的显示像素，主要是在GEngine->Tick的内部进行调用。FViewport::Draw主要有以下实现：

1. 定义了一个智能指针，用途是让渲染线程暂停，以便让其他线程获取资源，此处的用途是Slate UI获取资源，是否使用取决于一个全局变量：GCaptureCompositionNextFrame  
   static TUniquePtr<FSuspendRenderingThread> GRenderingThreadSuspension;
2. 判断当前是否能被渲染，主要是bIsGameRenderingEnabled，若不能则只进行LevelStreaming，不进行渲染，若能渲染则执行后面的渲染操作：  
   World->UpdateLevelStreaming();
3. 获取是否截图或者高清截图的信息，以便于进行截图：  
   FScreenshotRequest::RequestScreenshot( FString(), bShowUI, bAddFilenameSuffix );  
   HighResScreenshot();
4. 判断画布的SizeX，Y>0以进行下一步，获取是否垂直同步的信息，存储在bLockToVsync，主要取决于cVar或者cinematicMode
5. 通过RHIlist告知渲染线程，开始渲染这一帧：  
   EnqueueBeginRenderFrame(bShouldPresent);
6. 计算游戏线程的时间，排除了idle的等待时间
7. 初始化画布Canvas、设置绘制矩形区域、保证没有上下翻转、调用ViewportClient->Draw(this, &Canvas)、给渲染线程发一堆指令：  
   FCanvas Canvas(this, nullptr, ViewportWorld, ViewportWorld ?ViewportWorld->FeatureLevel.GetValue() : GMaxRHIFeatureLevel, FCanvas::CDM\_DeferDrawing, ViewportClient->ShouldDPIScaleSceneCanvas() ? ViewportClient->GetDPIScale() : 1.0f);  
   Canvas.SetRenderTargetRect(FIntRect(0, 0, SizeX, SizeY));  
   {  
   // Make sure the Canvas is not rendered upside down  
   Canvas.SetAllowSwitchVerticalAxis(true);  
   ViewportClient->Draw(this, &Canvas);// 此函数有很多重载实现，后续会深入分析  
   }  
   Canvas.Flush\_GameThread();
8. Client的截图操作，不知道与上面的截图有何区别，后续补充：  
   ViewportClient->ProcessScreenShots(this);
9. 如有垂直同步，则给渲染线程发指令让其执行：  
   SetRequiresVsync(bLockToVsync); //此语句是一个空函数，推测目前只是作为注释的用途  
   EnqueueEndRenderFrame(bLockToVsync, bShouldPresent);
10. 重置相机采集的标签为0，这个标签通常会在每一帧被重置，这个值的设置会影响很多效果，如运动模糊等：  
    PlayerController->PlayerCameraManager->bGameCameraCutThisFrame = false;
11. 处理PresentAndStopMovieDelay与bIsGameRenderingEnabled的关系，如前者不为0，那么在每一帧减一置为0时，后者就会被置为true。如果前者在每一帧都设置为大于1，那么就可以让后者一直为false，就可以不执行渲染转而播放movie的操作
12. 在即将出作用域的时候，reset智能指针GRenderingThreadSuspension，当然得看全局变量GCaptureCompositionNextFrame是否被设为true

# P6 UE4的编辑器视口绘制函数FEditorViewportClient::Draw

此函数在EditorViewportClient.cpp里面，在编辑器Editor模式下，此函数就是P5中ViewportClient->Draw调用的函数，形参列表为FViewport\*, FCanvas\*。主要有以下实现：

1. 暂存this->Viewport在此函数结束时取回
2. 根据UseAppTime()来确实是用World还是App的Time，主要有三个TimeSeconds，RealTimeSeconds，DeltaTimeSeconds
3. 判断是否要进行立体渲染，比如用于VR的HMD，在渲染前进行相关设置：  
   const bool bStereoRendering = GEngine->IsStereoscopic3D( InViewport );
4. 取得this->EngineShowFlags，并存到UseEngineShowFlags，用其对一些函数进行重载：  
   OverrideShowFlagsFunc(UseEngineShowFlags);
5. 初始化并构造ViewFamily，这个类包含了对一个Scene的多个views，整个family有相同的ViewMode，RT，Scene，ShowFlags等：  
   FSceneViewFamilyContext ViewFamily(FSceneViewFamily::ConstructionValues(Canvas->GetRenderTarget(),GetScene(),UseEngineShowFlags)  
   .SetWorldTimes( TimeSeconds, DeltaTimeSeconds, RealTimeSeconds )  
   .SetRealtimeUpdate( IsRealtime() &&FSlateThrottleManager::Get().IsAllowingExpensiveTasks() )  
   .SetViewModeParam( ViewModeParam, ViewModeParamName ) );
6. UpdateDebugViewModeShaders();
7. 对ViewFamily的相关成员进行设置，有EngineShowFlags（及其部分子成员），bIsHDR，ViewExtensions，ViewMode，ExposureSettings，LandscapeLODOverride
8. 获取诸如bCanDisableTonemapper相关的参数，对ViewFamily.EngineShowFlags以引用方式传进去修改，以及据此修改光照参数：  
   EngineShowFlagOverride(ESFIM\_Editor, ViewFamily.ViewMode, ViewFamily.EngineShowFlags, bCanDisableTonemapper);  
   EngineShowFlagOrthographicOverride(IsPerspective(), ViewFamily.EngineShowFlags);  
   UpdateLightingShowFlags( ViewFamily.EngineShowFlags );
9. 进行screen percentage和upscaling：  
   ViewFamily.SecondaryViewFraction = GetDPIDerivedResolutionFraction();  
   GCustomEditorStaticScreenPercentage->SetupEditorViewFamily(ViewFamily, this);
10. 对ViewFamily的View进行计算（此处之后会深入分析，里面对ViewFamily的成员Views进行了添加），此处根据bStereoRendering来设置NumViews，true为2，否则为1，主要确定了View的相关变换矩阵以及根据EngineShowFlags设置着色模型的参数，另外也设置了安全框SafeFrame：  
    View = CalcSceneView( &ViewFamily, StereoPass );  
    SetupViewForRendering(ViewFamily,\*View);  
    FSlateRect SafeFrame;
11. 根据IsAspectRatioConstrained()选择是否在相机比例不匹配的时候加黑边：  
    Canvas->Clear(FLinearColor::Black);
12. 绘制3D场景，此函数里面调用了render函数（会在后面详细分析）：  
    GetRendererModule().BeginRenderingViewFamily(Canvas,&ViewFamily);
13. 绘制画布（内部画了一条string信息和Editor tool的HUD）和安全线框：  
    DrawCanvas( \*Viewport, \*View, \*Canvas );  
    DrawSafeFrames(\*Viewport, \*View, \*Canvas);
14. 清理暂存的用于调试的线：  
    World->LineBatcher->Flush();  
    World->ForegroundLineBatcher->Flush();
15. 绘制Widget，画的的HUD的text：  
    Widget->DrawHUD( Canvas );
16. 根据GetViewportType()，绘制坐标轴和比例尺。顺带提一下ELevelViewportType有八种模式：正交投影（六个对齐坐标轴+一个自由的）+透视投影。如LVT\_OrthoXY模式下：  
    DrawAxes(Viewport, Canvas, &XYRot, EAxisList::XY);  
    DrawScaleUnits(Viewport, Canvas, \*View);
17. 显示Stat信息：  
    DrawStatsHUD( World, Viewport, DebugCanvas, NULL, EmptyPropertyArray, GetViewLocation(), GetViewRotation() );
18. 判断是否为实时渲染，不是的话则等待渲染线程完成渲染：  
    FlushRenderingCommands();

# P7 UE4的游戏视口绘制函数UGameViewportClient::Draw

P6里面分析FEditorViewportClient::Draw，下面来分析UGameViewportClient::Draw。两者实现差不多，主要讲一下后者多出来的操作：

根据GetEngineShowFlags()->PathTracing的状态对ShowFlag的标签进行设置：  
EngineShowFlags.SetPathTracing(true);

根据AllowDebugViewmodes()处理缓存的可视化命令：  
NewBufferVisualizationMode = ModeName;

可能有多个LocalPlayers，每个PlayerController都自己计算一次View并添加到ViewFamily，并且更新音频位置，且如果有光线追踪则启动，以及把view信息传给资源加载类（可能在小视场进行优化）：  
FSceneView\* View = LocalPlayer->CalcSceneView(&ViewFamily, ViewLocation, ViewRotation, InViewport, nullptr, PassType);  
AudioDevice->SetListener(MyWorld, ViewportIndex, ListenerTransform, (View->bCameraCut ? 0.f : MyWorld->GetDeltaSeconds()));  
View->SetupRayTracedRendering();  
IStreamingManager::Get().AddViewInformation(View->ViewMatrices.GetViewOrigin(), View->UnscaledViewRect.Width(), View->UnscaledViewRect.Width() \* View->ViewMatrices.GetProjectionMatrix().M[0][0], StreamingScale);

更新Level streaming：  
MyWorld->UpdateLevelStreaming();

找到所有Views的包围盒，并计算总面积，根据包围盒的信息来产生黑边或者清空Canvas，若此时清空没成功，后面还有一次清理：  
int32 BlackBorders = FMath::Clamp(CVarSetBlackBordersEnabled.GetValueOnGameThread(), 0, 10);  
SceneCanvas->Clear(FLinearColor::Transparent);

在渲染场景之前，开启动态分辨率，结束后在渲染UI之前关闭动态分辨率：  
GEngine->EmitDynamicResolutionEvent(EDynamicResolutionStateEvent::EndDynamicResolutionRendering);

打出粒子系统的debug信息：  
MyWorld->FXSystem->DrawDebug(SceneCanvas);  
绘制PlayerController的HUD，在绘制前后把变换矩阵push和pop到Canvas上面，并进行变换和逆变换：

SceneCanvas->PushAbsoluteTransform(FTranslationMatrix(CanvasOrigin));  
CanvasObject->ApplySafeZoneTransform();  
PlayerController->MyHUD->PostRender();  
CanvasObject->PopSafeZoneTransform();  
SceneCanvas->PopTransform();