nDisplay Render in Unreal 5.6 (Low Level – Deep Dive)

Render Cycle

Everything starts in the main GameThread. In case you don't know, there are three main threads involved in rendering, these are:

GameThread (frame n) –> RenderThread (frame n+1) -> RHI (Render Hw Interface, frame n+1 or n+2)

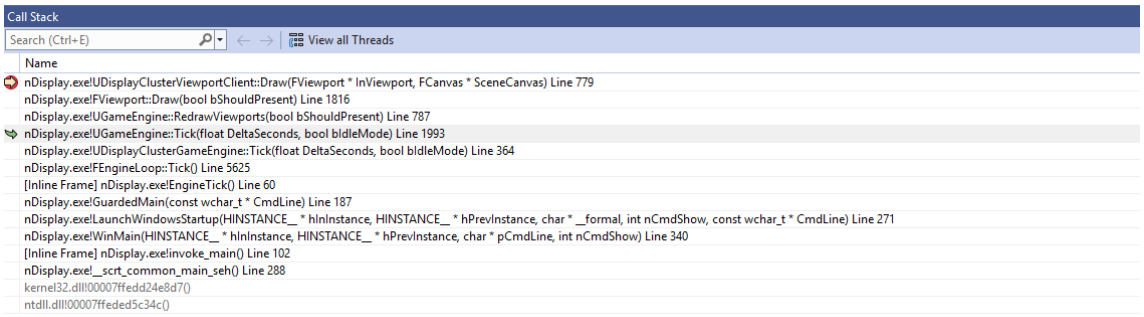
and for now we can ignore RHI because it is just a helper offloader for RenderFrame. They go in sync meaning that they are not independent. They can only be one frame off each other, so that if one blocks the next will block one frame afterwards. They do not share memory, because data is copied. For every class Data in GameThread there is a DataProxy in RenderThread.

This document is meant to help understand nDisplay render, but the general render pipeline is very similar and in fact shares a lot of code with the pipeline discussed below. Other topics relating to nDisplay –which is a broad topic- are not covered in this document.

As mentioned before, everything starts in GameThread, in either UGameViewportClient or some derived class; nDisplay uses a special subclass, which is required to implement a method call Draw():

```
8
9   UCLASS()
10  class DISPLAYCLUSTER_API UDisplayClusterViewportClient
11      : public UGameViewportClient
12  {
13      GENERATED_BODY()
14
15  public:
16      UDisplayClusterViewportClient(FVTableHelper& Helper);
17      virtual ~UDisplayClusterViewportClient();
18
19      virtual void Init(struct FWorldContext& WorldContext, UGameInstance*
          OwningGameInstance, bool bCreateNewAudioDevice = true) override;
20      virtual void Draw(FViewport* Viewport, FCanvas* SceneCanvas) override;
21
22      virtual FSceneViewport* CreateGameViewport(TSharedPtr<SViewport>
          InViewportWidget) override;
23
24  protected:
25  #if WITH_EDITOR
26      bool Draw_PIE(FViewport* InViewport, FCanvas* SceneCanvas);
27  #endif /*WITH_EDITOR*/
28  };
```

The GameEngine ticks (updates) the World (level) objects and calls to all viewports to draw (a viewport is like a render window)




```
774
775     void UGameEngine::RedrawViewports( bool bShouldPresent /*= true*/ )
776     {
777         SCOPE_CYCLE_COUNTER(STAT_RedrawViewports);
778         CSV_SCOPED_TIMING_STAT_EXCLUSIVE(ViewportMisc);
779         if ( GameViewport != NULL )
780         {
781             GameViewport->LayoutPlayers();
782             if ( GameViewport->Viewport != NULL )
783             {
784                 GameViewport->Viewport->Draw(bShouldPresent);
785             }
786         }
787     }
```

In our case it ends up in DisplayClusterViewportClient::Draw() which is a pretty long function, but the most important parts –at least for now- are:

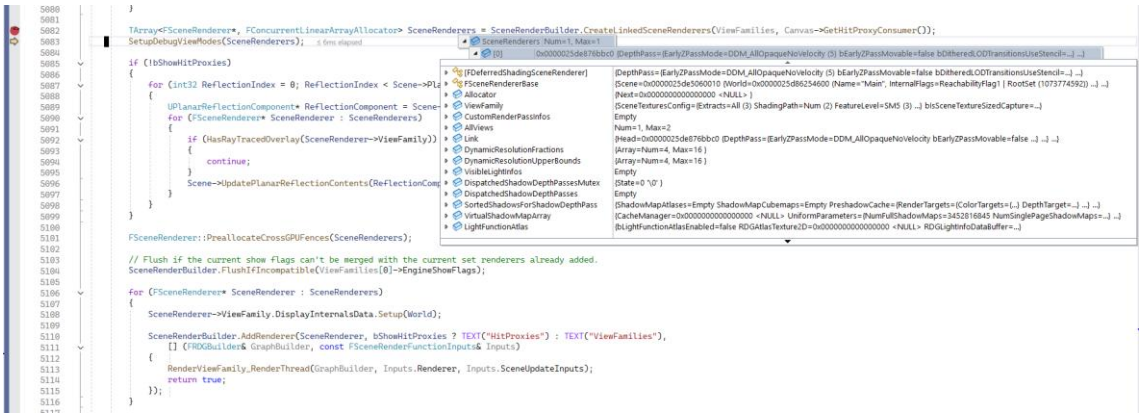
1) Create resources to render frame render (see below on this)

```
472
473     // Gather all view families first
474     TArray<FSceneViewFamilyContext*> ViewFamilies;
475
476     // Initialize new render frame resources
477     FDisplayClusterRenderFrame RenderFrame;
478     if (!DCRenderDevice->BeginNewFrame(InViewport, MyWorld, RenderFrame))
479     {
480         // skip rendering: Can't build render frame
481         return;
482     }
```

2) And Render of the ViewFamilies

Think of a SceneViewFamily as scene + a group of views (and think of a view as a render target texture, viewmatrix, and projection matrix) associated with a viewport render. It is not clear why many views are needed per family but it might be the case that one view correspond to the main POV, while other views might be used for light shadowsmaps, cubemap reflection views, etc.

In BeginRenderingViewFamilies a SceneRenderBuilder is used to create one or more Renderers, in our case a DeferredShadingSceneRenderer is created. Notice in the following picture that the renderers are first created (line 5082) and then added to the builder with a lambda that will be used to call RenderViewFamily_RenderThread in the RenderThread (line 5110). The use of lambdas to inject code meant for the RenderThread inside GameThread code is pervasive in Unreal code.



Once the SceneRenderBuilder has been filled with renderers, it is executed:

```
5105
5106        for (FSceneRenderer* SceneRenderer : SceneRenderers)
5107        {
5108            SceneRenderer->ViewFamily.DisplayInternalsData.Setup(World);
5109
5110            SceneRenderBuilder.AddRenderer(SceneRenderer, bShowHitProxies ? TEXT("HitProxies") : TEXT("ViewFamilies"),
5111                [] (FRDGBuilder& GraphBuilder, const FSceneRenderFunctionInputs& Inputs)
5112            {
5113                RenderViewFamily_RenderThread(GraphBuilder, Inputs.Renderer, Inputs.SceneUpdateInputs);
5114                return true;
5115            });
5116        }
5117
5118        SceneRenderBuilder.AddRenderCommand([SceneRenderers = MoveTemp(SceneRenderers)](FRHICommandListImmediate& RHICmdList)
5119        {
5120            CleanupViewFamilies_RenderThread(RHICmdList, SceneRenderers);
5121        });
5122
5123        SceneRenderBuilder.Execute();
5124
5125        // Force kick the RT if we've got RT polling on.
5126        // This saves us having to wait until the polling period before the scene draw starts executing.
5127        if (GRenderThreadPollingOn)
5128        {
5129            FTaskGraphInterface::Get().WakeNamedThread(ENamedThreads::GetRenderThread());
5130        }
5131    }
5132 }
5133
```

This is what execute does:

```
099     }
100
101     void FSceneRenderBuilder::Execute()
102     {
103     #if !USE_NULL_RHI
104         if (Processor)
105         {
106             Processor->Execute();
107             Processor = nullptr;
108         }
109     #endif
110     }
111
```

Processor is just a for loop in the GameThread that executes operations

```
void FSceneRenderProcessor::Execute()
{
    checkf(!bInsideGroup, TEXT("FSceneRenderBuilder::Execute called within scene render group scope %s. You must end the scope first."), *GroupNodes.GetTail()->Name);

#if WITH_GPUDEBUGCRASH [ Active Preprocessor Block ]
#endif

    UE::RenderCommandPipe::FSyncScope SyncScope;
    FUniformExpressionCacheAsyncUpdateScope AsyncUpdateScope;
    TOptional<UE::RenderCore::DumpGPU::FDumpScope> GpuDumpScope;
    TOptional<RenderCaptureInterface::FScopedCapture> GpuCaptureScope;

    for (FOp Op : Ops)
    {
        switch (Op.Type)
        {
        case FOp::EType::BeginGroup: { ... }
        break;
        case FOp::EType::EndGroup: { ... }
        break;
        case FOp::EType::FunctionCall:
        {
            (*Op.Data_FunctionCall)();
        }
        break;
        case FOp::EType::Render:
        {
            FRenderNode& RenderNode = *Op.Data_Render;

            ENQUEUE_RENDER_COMMAND(SceneRenderBuilder_Render)([this, &RenderNode] (FRHICommandListImmediate& RHICmdList)
            {
                LLM_SCOPE(ELLMTag::SceneRender);
                RenderState.GroupEvent.Begin(RenderState.Group, RHICmdList, FGroupEventLocation::SceneRenderCommand);

                ON_SCOPE_EXIT
                {
                    RenderState.GroupEvent.End(RHICmdList, FGroupEventLocation::SceneRenderCommand);
                };
```

We focus on **FOp::EType::Render** operations, which starts with ENQUEUE_RENDER_COMMAND lambda to enqueue work in the Render Thread (here happens the main transition from GameThread to Render Thread). This lambda will be called when the Render Thread gets to it a few moments later, and it will - among other things - create an FRDGBuilder GraphBuilder, and pass it to the function of the render operation. The render node operation is the lambda that we passed to the AddRenderer function above:

```
        if (Renderer.ViewFamily.EngineShowFlags.Rendering && !RenderState.bSceneUpdateConsumed)
        {
            SceneUpdateInputs.Emplace();
            SceneUpdateInputs->Scene = Scene;
            SceneUpdateInputs->FXSystem = Scene->FXSystem;
            SceneUpdateInputs->FeatureLevel = Scene->GetFeatureLevel();
            SceneUpdateInputs->ShaderPlatform = Scene->GetShaderPlatform();
            SceneUpdateInputs->GlobalShaderMap = GetGlobalShaderMap(SceneUpdateInputs->ShaderPlatform);
            SceneUpdateInputs->Renderers = ActiveRenderers;
            SceneUpdateInputs->ViewFamilies = ActiveViewFamilies;
            SceneUpdateInputs->Views = ActiveViews;
            SceneUpdateInputs->CommonShowFlags = CommonShowFlags;
        }

        const FSceneRenderFunctionInputs FunctionInputs(&Renderer, SceneUpdateInputs ? &SceneUpdateInputs.GetValue() : nullptr, *RenderNode.Name, *RenderState.FullPath);

        FRDGBuilder GraphBuilder(RHICmdList, RDG_EVENT_NAME("%s", FunctionInputs.FullPath), ERDGBuilderFlags::Parallel, Scene->GetShaderPlatform());
        FSceneRendererBase::SetActiveInstance(GraphBuilder, &Renderer);

    #if WITH_MGPU
        if (Renderer.ViewFamily.bForceCopyCrossGPU)
        {
            GraphBuilder.EnableForceCopyCrossGPU();
        }
    #endif

        if (!Renderer.ViewFamily.EngineShowFlags.HitProxies)
        {
            VISUALIZE_TEXTURE_BEGIN_VIEW(Scene->GetFeatureLevel(), Renderer.Views[0].GetViewKey(), FunctionInputs.FullPath, Renderer.Views[0].bIsSceneCapture);
        }

        const bool bRenderCalled = RenderNode.Function(GraphBuilder, FunctionInputs);
```

GraphBuilder is simply put a continer where all the renderpasses and resources (such as textures, meshes, etc) will be stored, prior to execution during the RenderNode.Function() call. After that it will be executed and the renderpasses will be run.

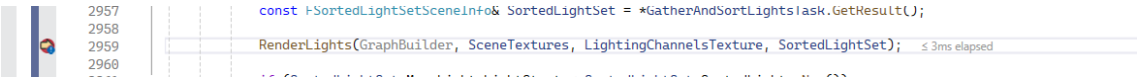
```
            // The final graph builder is responsible for flushing resources.
            if (RenderNode.Renderer == Renderers.Last())
            {
                GraphBuilder.SetFlushResourcesRHI();
            }

            GraphBuilder.Execute();

            Cleanup(RHICmdList, &Renderer);
        });
    }
```

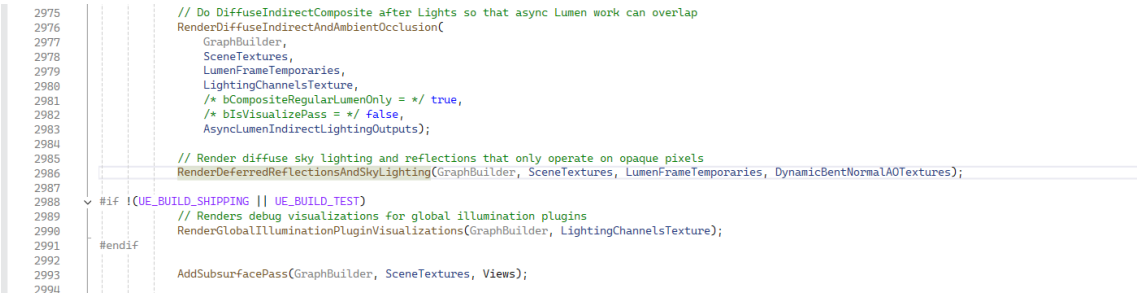
So what happens in RenderNode.Function() call?

A ton of things (thousands of lines, including calls to functions of thousands of lines, unreal rendering is very rich, see for yourself), but let’s state a few examples:

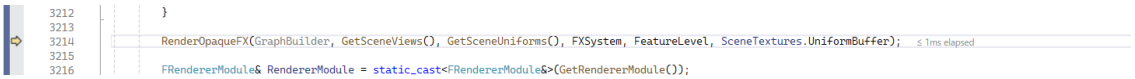
RenderBasePass

Lights and Lighting



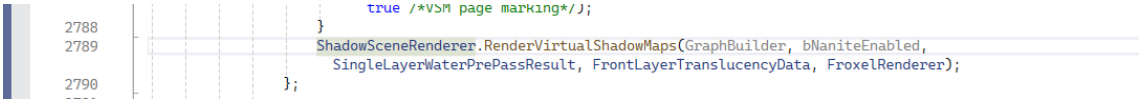


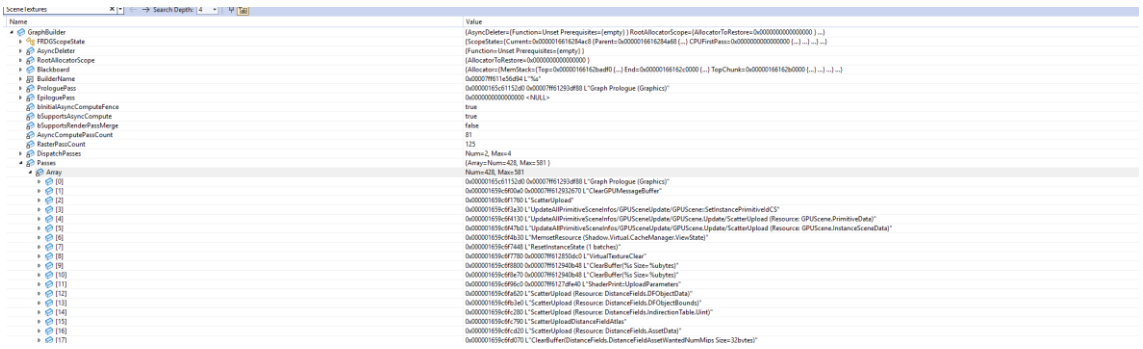
FX, etc.



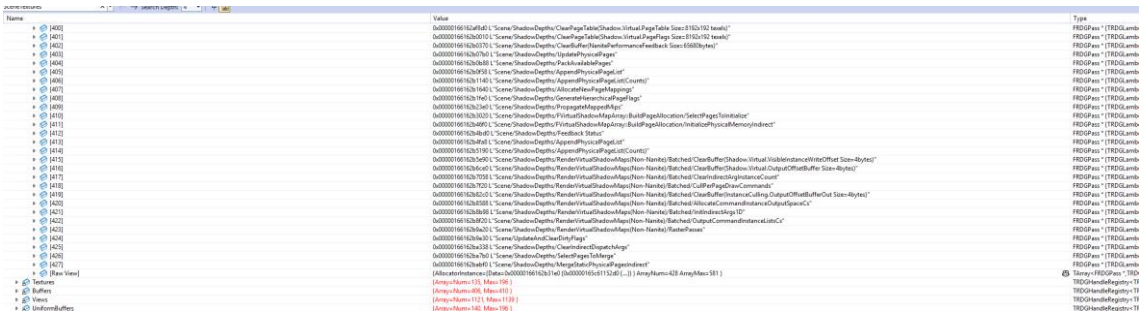
Notice GraphBuilder is the first parameter in each step. As mentioned GraphBuilder accumulates render passes and resources that are required for rendering. Notice this is only a “TODO” accumulation phase, nothing really gets rendered at this moment, not until GraphBuilder executes.

As an example of a single step, render shadow maps includes its passes:





…



Notice at the bottom of the last picture the accumulated resources such as Buffers and Textures.

Create resources for Frame Render

On every frame, on game thread DisplayClusterViewportClient::Draw() creates the rendertarget resources for each viewport (InViewport) at BeginNewFrame(…), and assigns its render target to the ViewFamily

```
475
476     // Initialize new render frame resources
477     FDisplayClusterRenderFrame RenderFrame;
478     if (!DCRenderDevice->BeginNewFrame(InViewport, MyWorld, RenderFrame))
479     {
480         // skip rendering: Can't build render frame
481         return
482     }
483
484     IDisplayClusterViewportManager* RenderFrameViewportManager = RenderFrame.GetViewportManager();
485     if (!RenderFrameViewportManager)
486     {
487         // skip rendering: Can't find render manager
488         return;
489     }
490
491     // Handle special viewports game-thread logic at frame begin
492     DCRenderDevice->InitializeNewFrame();
493
494     for (FDisplayClusterRenderFrameTarget& DCRenderTarget : RenderFrame.RenderTargets)
495     {
496         for (FDisplayClusterRenderFrameTargetViewFamily& DCViewFamily : DCRenderTarget.ViewFamilies)
497         {
498             // Create the view family for rendering the world scene to the viewport's render target
499             ViewFamilies.Add(new FSceneViewFamilyContext(RenderFrameViewportManager->CreateViewFamilyConstructionValues(
500                 DCRenderTarget,
501                 MyWorld->Scene,
502                 EngineShowFlags,
503                 false               // bAdditionalViewFamily  (filled in later, after list of families is known, and optionally reordered)
504             )));
505             FSceneViewFamilyContext& ViewFamily = *ViewFamilies.Last();
```

This in turns calls…

```
572
573 bool FDisplayClusterViewportManager::BeginNewFrame(FViewport* InViewport, FDisplayClusterRenderFrame& OutRenderFrame)
574 {
575     check(IsInGameThread());
```

…

```
679     UpdateSceneRenderTargetSize();
680
681     // Build new frame structure
682     if (!RenderFrameManager->BuildRenderFrame(InViewport, ImplGetCurrentRenderFrameViewports(), OutRenderFrame))   ≤ 3ms elapsed
683     {
684         return false;
685     }
686
687     // Allocate resources for frame
688     if (!RenderTargetManager->AllocateRenderFrameResources(InViewport, ImplGetCurrentRenderFrameViewports(), OutRenderFrame))
689     {
690         return false;
691     }
692
693     const FIntPoint RenderFrameSize = OutRenderFrame.FrameRect.Size();
```

During these calls a new render target resource is allocated for each viewport and assigned to it in each view of each family, with then resource enum **EDisplayClusterViewportResource::RenderTargets** …

```
130  bool FDisplayClusterRenderTargetManager::AllocateRenderFrameResources(FViewport* InViewport, const
       TArray<TSharedPtr<FDisplayClusterViewport, ESPMode::ThreadSafe>>& InViewports, FDisplayClusterRenderFrame&
       InOutRenderFrame)
131  {
132      bool bResult = true;
133
134      if(ResourcesPool->BeginReallocateResources(InViewport, Configuration->GetRenderFrameSettings()))
135      {
136          // ReAllocate Render targets for all viewports
137          for (FDisplayClusterRenderFrameTarget& FrameRenderTargetIt : InOutRenderFrame.RenderTargets)
138          {
139              if (FrameRenderTargetIt.bShouldUseRenderTarget)
140              {
141                  const EDisplayClusterViewportResourceSettingsFlags CustomFlags = ImplGetCustomFlags
                      (FrameRenderTargetIt.CaptureMode);
142                  const EPixelFormat CustomFormat = ImplGetCustomFormat(FrameRenderTargetIt.CaptureMode);
143
144                  // reallocate
145                  TSharedPtr<FDisplayClusterViewportResource, ESPMode::ThreadSafe> NewResource = ResourcesPool-
                      >AllocateResource(EmptyString, FrameRenderTargetIt.RenderTargetSize, CustomFormat,
                      EDisplayClusterViewportResourceSettingsFlags::RenderTarget | CustomFlags);
146                  if (NewResource.IsValid())
147                  {
148                      // Set RenderFrame resource
149                      FrameRenderTargetIt.RenderTargetResource = NewResource;
150
151                      // Assign for all views in render target families
152                      for (FDisplayClusterRenderFrameTargetViewFamily& ViewFamily : FrameRenderTargetIt.ViewFamilies)
153                      {
154                          for (FDisplayClusterRenderFrameTargetView& ViewIt : ViewFamily.Views)
155                          {
156                              if (FDisplayClusterViewport* ViewportPtr = static_cast<FDisplayClusterViewport*>
                                  (ViewIt.Viewport.Get()))
157                              {
158                                  // Array already resized in function FDisplayClusterViewport::UpdateFrameContexts() wi
                                      RenderTargets.AddZeroed(ViewportContextAmount);
159                                  check(ViewportPtr->GetViewportResources
                                      (EDisplayClusterViewportResource::RenderTargets).IsValidIndex(ViewIt.ContextNum));
160
161                                  ViewportPtr->GetViewportResourcesImpl(EDisplayClusterViewportResource::RenderTargets)
                                      [ViewIt.ContextNum] = NewResource;
162
```

I have marked with breakpoints the allocation and the assignment. Also keep in mind the new (render target) resource that is assigned to the viewport with the enum **EDisplayClusterViewportResource::RenderTargets** is also returned via InOutRenderFrame through the iterator in line 149 (shown again highlighted here):

```
129
130  bool FDisplayClusterRenderTargetManager::AllocateRenderFrameResources(FViewport* InViewport, const TArray<TSharedPtr<FDisplayClusterViewport,
       ESPMode::ThreadSafe>>& InViewports, FDisplayClusterRenderFrame& InOutRenderFrame)
131  {
132      bool bResult = true;
133
134      if(ResourcesPool->BeginReallocateResources(InViewport, Configuration->GetRenderFrameSettings()))
135      {
136          // ReAllocate Render targets for all viewports
137          for (FDisplayClusterRenderFrameTarget& FrameRenderTargetIt : InOutRenderFrame.RenderTargets)
138          {
139              if (FrameRenderTargetIt.bShouldUseRenderTarget)
140              {
141                  const EDisplayClusterViewportResourceSettingsFlags CustomFlags = ImplGetCustomFlags(FrameRenderTargetIt.CaptureMode);
142                  const EPixelFormat CustomFormat = ImplGetCustomFormat(FrameRenderTargetIt.CaptureMode);
143
144                  // reallocate
145                  TSharedPtr<FDisplayClusterViewportResource, ESPMode::ThreadSafe> NewResource = ResourcesPool->AllocateResource(EmptyString,
                      FrameRenderTargetIt.RenderTargetSize, CustomFormat, EDisplayClusterViewportResourceSettingsFlags::RenderTarget | CustomFlags);
146                  if (NewResource.IsValid())
147                  {
148                      // Set RenderFrame resource
149                      FrameRenderTargetIt.RenderTargetResource = NewResource;
150
151                      // Assign for all views in render target families
```

This will allow it to be used when creating the view family later, see below. In fact as it says in the comments, these render targets resources are both the view family render destination, and can later be accessed via InternalRenderTargetResource, which will be used later

```
12
13      /**
14       * Internal types of DC viewport resources
15       * These types are used to point to actual instances of the resources in the viewport.
16       */
17      enum class EDisplayClusterViewportResource : uint8
18      {
19          // View family render to this resources
20          // This is the resource that can be pointed to using the enum values:
21          // - 'EDisplayClusterViewportResourceType::InternalRenderTargetEntireRectResource'
22          // - 'EDisplayClusterViewportResourceType::InternalRenderTargetResource'
23          RenderTargets,
24
```

Other viewport internal resources (for warping) are allocated at this step as well, see code.

At the very end of AllocateRenderFrameResources you see a call to AllocateFrameTargets

```
234             }
235
236             // Allocate frame targets for all visible  on backbuffer viewports
237             FIntPoint ViewportSize = InViewport ? InViewport->GetSizeXY() : FIntPoint(0, 0);
238
239             if (!AllocateFrameTargets(ViewportSize, InOutRenderFrame))
240             {
241                 UE_LOG(LogDisplayClusterViewport, Error, TEXT("DisplayClusterRenderTargetManager: Can't allocate frame
                      targets."));
242                 bResult = false;
243             }
244
245             ResourcesPool->EndReallocateResources();
246         }
247
248         return bResult;
249     }
```

```
bool FDisplayClusterRenderTargetManager::AllocateFrameTargets(const FIntPoint& InViewportSize, FDisplayClusterRenderFrame& InOutRenderFrame)
{
    // ICVFX internal viewports are invisible and do not use output frame resources.
    // In case of off-screen rendering this texture is not used.
    if (!Configuration->GetRenderFrameSettings().ShouldUseOutputFrameTargetableResources())
    {
        // Skip creating output resources if there are no visible viewports. This will save GPU memory.
        return true;
    }

    const uint32 FrameTargetsAmount = Configuration->GetRenderFrameSettings().GetViewPerViewportAmount();
    const FIntPoint DesiredRTTSize = Configuration->GetRenderFrameSettings().GetDesiredRTTSize(InViewportSize);
    FIntPoint TargetLocation(ForceInitToZero);

    // Offset for stereo rendering on a monoscopic display (side-by-side or top-bottom)
    const FIntPoint TargetOffset(
        DesiredRTTSize.X < InViewportSize.X ? DesiredRTTSize.X : 0,
        DesiredRTTSize.Y < InViewportSize.Y ? DesiredRTTSize.Y : 0
    );

    // Reallocate frame target resources
    TArray<TSharedPtr<FDisplayClusterViewportResource, ESPMode::ThreadSafe>> NewFrameTargetResources;
    TArray<TSharedPtr<FDisplayClusterViewportResource, ESPMode::ThreadSafe>> NewAdditionalFrameTargetableResources;

    for (uint32 FrameTargetsIt = 0; FrameTargetsIt < FrameTargetsAmount; FrameTargetsIt++)
    {
        TSharedPtr<FDisplayClusterViewportResource, ESPMode::ThreadSafe> NewResource = ResourcesPool->AllocateResource(EmptyString, InOutRenderFrame.FrameRect.Size(), PF_Unknown,
          EDisplayClusterViewportResourceSettingsFlags::RenderTargetableTexture);
        if (NewResource.IsValid())
        {
            if (EnumHasAnyFlags(NewResource->GetResourceState(), EDisplayClusterViewportResourceState::Initialized) == false)
            {
                // Log: New resource created
                UE_LOG(LogDisplayClusterViewport, Verbose, TEXT("Created new RenderFrame resource (%dx%d)"), NewResource->GetResourceSettings().GetSizeX(), NewResource-
                  >GetResourceSettings().GetSizeY());
            }

            // calc and assign backbuffer offset (side_by_side, top_bottom)
            NewResource->SetBackbufferFrameOffset(InOutRenderFrame.FrameRect.Min + TargetLocation);
            TargetLocation += TargetOffset;
            NewFrameTargetResources.Add(NewResource);
        }
```

Frame Target RenderTargetableTexture is also allocated, and stored in EDisplayClusterViewportResource::Output**Frame**TargetableResources

```
307
308     // Assign frame resources for all visible viewports
309     for (FDisplayClusterRenderFrameTarget& RenderTargetIt : InOutRenderFrame.RenderTargets)
310     {
311         for (FDisplayClusterRenderFrameTargetViewFamily& ViewFamilieIt : RenderTargetIt.ViewFamilies)
312         {
313             for (FDisplayClusterRenderFrameTargetView& ViewIt : ViewFamilieIt.Views)
314             {
315                 FDisplayClusterViewport* ViewportPtr = static_cast<FDisplayClusterViewport*>(ViewIt.Viewport.Get());
316                 if (ViewportPtr && ViewportPtr->GetRenderSettings().bVisible)
317                 {
318                     ViewportPtr->GetViewportResourcesImpl(EDisplayClusterViewportResource::OutputFrameTargetableResources) = NewFrameTargetResources;
319                     ViewportPtr->GetViewportResourcesImpl(EDisplayClusterViewportResource::AdditionalFrameTargetableResources) = NewAdditionalFrameTargetableResources;
320
321                     // Adjust viewports frame rects. This offset saved in 'BackbufferFrameOffset'
322                     if (ViewportPtr->GetContexts().IsValidIndex(ViewIt.ContextNum))
```

Now with all these resources, we turn to the view family creation. The first thing one has to figure out is how does the view family render on the

EDisplayClusterViewportResource::RenderTargets resources. After the creation of the frame resources, the view families are created:

```
493
494     for (FDisplayClusterRenderFrameTarget& DCRenderTarget : RenderFrame.RenderTargets)
495     {
496         for (FDisplayClusterRenderFrameTargetViewFamily& DCViewFamily : DCRenderTarget.ViewFamilies)
497         {
498             // Create the view family for rendering the world scene to the viewport's render target
499             ViewFamilies.Add(new FSceneViewFamilyContext(RenderFrameViewportManager->CreateViewFamilyConstructionValues(
500                 DCRenderTarget,
501                 MyWorld->Scene,
502                 EngineShowFlags,
503                 false              // bAdditionalViewFamily  (filled in later, after list of families is known, and optionally reordered)
504             )));
```

Notice how the DCRenderTarget corresponds to the RenderFrame.RenderTargets that is none other than the OutRenderFrame created in BeginNewFrame(…) which contains all these newly created resources, including of course the render target.

This FSceneViewFamilyContext is a SceneViewFamily

```
2572
2573 /**
2574  * A view family which deletes its views when it goes out of scope.
2575  */
2576 class FSceneViewFamilyContext : public FSceneViewFamily
2577 {
```

And it receives the render target via the Construction Values

```
493
494     for (FDisplayClusterRenderFrameTarget& DCRenderTarget : RenderFrame.RenderTargets)
495     {
496         for (FDisplayClusterRenderFrameTargetViewFamily& DCViewFamily : DCRenderTarget.ViewFamilies)
497         {
498             // Create the view family for rendering the world scene to the viewport's render target
499             ViewFamilies.Add(new FSceneViewFamilyContext(RenderFrameViewportManager-
                >CreateViewFamilyConstructionValue(
500               DCRenderTarget,
501               MyWorld->Scene,
502               EngineShowFlags,
503               false              // bAdditionalViewFamily  (filled in later, after list of families
                    is known, and optionally reordered)
504             )));
```

Construction values harvests the render target from the InFrameTarget

```
753
754 FSceneViewFamily::ConstructionValues FDisplayClusterViewportManager::CreateViewFamilyConstructionValues(
755     const FDisplayClusterRenderFrameTarget& InFrameTarget,
756     FSceneInterface* InScene,
757     FEngineShowFlags InEngineShowFlags,
758     const bool bInAdditionalViewFamily) const
759 {
760
761     bool bResolveScene = true;
762
763     // Sets the engine flags corresponding to the capture mode.
764     IDisplayClusterViewportManager::SetupEngineShowFlags(InFrameTarget.CaptureMode, InEngineShowFlags);
765
766     const FDisplayClusterRenderFrameSettings& RenderFrameSettings = Configuration->GetRenderFrameSettings();
767
768     // A special case for DCRA previewing in a scene to avoid double use of PP.
769     if(RenderFrameSettings.IsPostProcessDisabled()) { ... }
774
775     // A special use case is setting up alpha channel capture:
776     // (When using the DC viewport rendering pipeline).
777     switch (InFrameTarget.CaptureMode) { ... }
811
812     FRenderTarget* RenderTarget = InFrameTarget.RenderTargetResource.IsValid() ? InFrameTarget.RenderTargetResource-
          >GetViewportResourceRenderTarget() : nullptr;
813     return FSceneViewFamily::ConstructionValues(RenderTarget, InScene, InEngineShowFlags)
814         .SetResolveScene(bResolveScene)
815         .SetRealtimeUpdate(true)
816         .SetAdditionalViewFamily(bInAdditionalViewFamily);
817 }
```

And thus **EDisplayClusterViewportResource::RenderTargets** resource gets assigned to the view family as its render target.

```
2971
2972    FSceneViewFamily::FSceneViewFamily(const ConstructionValues& CVS)
2973        :
2974        ViewMode(VMI_Lit),
2975        RenderTarget(CVS.RenderTarget),
2976        RenderTargetDepth(CVS.RenderTargetDepth),
2977        Scene(CVS.Scene),
2978        EngineShowFlags(CVS.EngineShowFlags),
2979        Time(CVS.Time)
```

Shaders and Uniforms

A Renderer's (such as DeferredRenderer) View Family is an object of type FViewFamilyInfo, which includes SceneTextures, that are initialized in the Render Method

```
1442    void FDeferredShadingSceneRenderer::Render(FRDGBuilder& GraphBuilder, const FSceneRenderUpdateInputs* SceneUpdateInputs)
1443    {
1444        if (!ViewFamily.EngineShowFlags.Rendering)
1445        {
1446            return;
1447        }
```

…

```
1725
1726        FSceneTextures::InitializeViewFamily(GraphBuilder, ViewFamily, FamilySize);
1727        FSceneTextures& SceneTextures = GetActiveSceneTextures();
1728
```

For every render pass there might be one or more shader associated, and each shader needs different shader parameters. During DeferredShadingSceneRenderer::Render method, the code modifies SceneTextures.UniformBuffer struct to deliver to each render pass, for instance here a uniform buffer with scene color is created.

```
3181
3182        // Rebuild scene textures to include scene color.
3183        SceneTextures.UniformBuffer = CreateSceneTextureUniformBuffer(GraphBuilder, &SceneTextures, FeatureLevel,
              SceneTextures.SetupMode);
3184
3185        if (!bHasRayTracedOverlay)
```

SetupMode variable has the information of what elements (mostly textures) have to be present in the uniform buffer in each step. This work is done in SetupSceneTextureUniformBuffer

```
1100
1101    TRDGUniformBufferRef<FSceneTextureUniformParameters> CreateSceneTextureUniformBuffer(
1102        FRDGBuilder& GraphBuilder,
1103        const FSceneTextures* SceneTextures,
1104        ERHIFeatureLevel::Type FeatureLevel,
1105        ESceneTextureSetupMode SetupMode)
1106    {
1107        FSceneTextureUniformParameters* SceneTexturesParameters = GraphBuilder.AllocParameters<FSceneTextureUniformParameters>
              ();
1108        SetupSceneTextureUniformParameters(GraphBuilder, SceneTextures, FeatureLevel, SetupMode, *SceneTexturesParameters);
1109        return GraphBuilder.CreateUniformBuffer(SceneTexturesParameters);
1110    }
1111
```

If a texture is not needed / used in the shader, a placeholder is inserted

```
1005
1006    void SetupSceneTextureUniformParameters(
1007        FRDGBuilder& GraphBuilder,
1008        const FSceneTextures* SceneTextures,
1009        ERHIFeatureLevel::Type FeatureLevel,
1010        ESceneTextureSetupMode SetupMode,
1011        FSceneTextureUniformParameters& SceneTextureParameters)
1012    {
1013        const FRDGSystemTextures& SystemTextures = FRDGSystemTextures::Get(GraphBuilder);
1014
1015        SceneTextureParameters.PointClampSampler = TStaticSamplerState<SF_Point>::GetRHI();
1016        SceneTextureParameters.SceneColorTexture = SystemTextures.Black;
1017        SceneTextureParameters.SceneDepthTexture = SystemTextures.DepthDummy;
1018        SceneTextureParameters.ScenePartialDepthTexture = SystemTextures.DepthDummy;
1019        SceneTextureParameters.GBufferATexture = SystemTextures.Black;
1020        SceneTextureParameters.GBufferBTexture = SystemTextures.Black;
1021        SceneTextureParameters.GBufferCTexture = SystemTextures.Black;
1022        SceneTextureParameters.GBufferDTexture = SystemTextures.Black;
1023        SceneTextureParameters.GBufferETexture = SystemTextures.Black;
1024        SceneTextureParameters.GBufferFTexture = SystemTextures.MidGrey;
1025        SceneTextureParameters.GBufferVelocityTexture = SystemTextures.Black;
1026        SceneTextureParameters.ScreenSpaceAOTexture = GetScreenSpaceAOFallback(SystemTextures);
1027        SceneTextureParameters.CustomDepthTexture = SystemTextures.DepthDummy;
1028        SceneTextureParameters.CustomStencilTexture = SystemTextures.StencilDummySRV;
1029
1030        if (SceneTextures)
1031        {
1032            const EShaderPlatform ShaderPlatform = SceneTextures->Config.ShaderPlatform;
1033
1034            if (EnumHasAnyFlags(SetupMode, ESceneTextureSetupMode::SceneColor))
1035            {
1036                SceneTextureParameters.SceneColorTexture = SceneTextures->Color.Resolve;   ≤ 1ms elapsed
1037            }
1038
```

Here a Black texture placeholder is first set to SceneTextureParameters.SceneColorTexture (line 1016), but then because SetupMode has ‘SceneColor’ flag active, the actual Sceneextures->Color.Resolve texture is passed to the uniform buffer for the next set of shaders.

Rendering on the Render Target

During DeferredShadingSceneRenderer::Render, an RDGTexture family texture is created with the ViewFamily

```
2335
2336        FRDGTextureRef ViewFamilyTexture = TryCreateViewFamilyTexture(GraphBuilder, ViewFamily);
2337        FRDGTextureRef ViewFamilyDepthTexture = TryCreateViewFamilyDepthTexture(GraphBuilder, ViewFamily);
2338        if (RendererOutput == ERendererOutput::DepthPrepassOnly)
2339        {
```

The texture in fact is already there because it was created in BeginFrame above and associated to the ViewFamily, see above, it is in ViewFamily.RenderTarget and it includes a RenderTargetTexture, here it is obtained and registered in Graph RDGBuilder as external texture (so GraphBuilder does not try to delete it at the end of the frame).

```
32
33  FRDGTextureRef TryCreateViewFamilyTexture(FRDGBuilder& GraphBuilder, const FSceneViewFamily& ViewFamily)
34  {
35      FRHITexture* TextureRHI = ViewFamily.RenderTarget->GetRenderTargetTexture();
36      FRDGTextureRef Texture = nullptr;
37      if (TextureRHI
38      {
39          Texture = RegisterExternalTexture(GraphBuilder, TextureRHI, TEXT("ViewFamilyTexture"));
40          GraphBuilder.SetTextureAccessFinal(Texture, ERHIAccess::RTV);
41      }
42      return Texture;
43  }
```

This ViewFamilyTexture is added to the postprocess inputs

```
3513        FPostProcessingInputs PostProcessingInputs;
3514        PostProcessingInputs.ViewFamilyTexture = ViewFamilyTexture;
3515        PostProcessingInputs.ViewFamilyDepthTexture = ViewFamilyDepthTexture;
3516        PostProcessingInputs.CustomDepthTexture = SceneTextures.CustomDepth.Depth;
3517        PostProcessingInputs.ExposureIlluminance = ExposureIlluminance;
3518        PostProcessingInputs.SceneTextures = SceneTextures.UniformBuffer;
3519        PostProcessingInputs.bSeparateCustomStencil = SceneTextures.CustomDepth.bSeparateStencilBuffer;
3520        PostProcessingInputs.PathTracingResources = PathTracingResources;
3521
3522        FRDGTextureRef InstancedEditorDepthTexture = nullptr; // Used to pass instanced stereo depth data from primary to secondary views
3523
```

And attached at the very end of the postprocess pass to gather the results of postprocess

```
3576        }
3577
3578        AddPostProcessingPasses(
3579            GraphBuilder,
3580            View, ViewIndex,
3581            GetSceneUniforms(),
3582            bAnyLumenActive,
3583            ViewPipelineState.DiffuseIndirectMethod,
3584            ViewPipelineState.ReflectionsMethod,
3585            PostProcessingInputs,
3586            NaniteResults,
3587            InstanceCullingManager,
3588            &VirtualShadowMapArray,
3589            LumenFrameTemporaries,
3590            SceneWithoutWaterTextures,
3591            TSRFlickeringInput,
3592            InstancedEditorDepthTexture);
3593    }
```

First it is recovered inside the method

```
346
347 void AddPostProcessingPasses(
348     FRDGBuilder& GraphBuilder,
349     const FViewInfo& View,
350     int32 ViewIndex,
351     FSceneUniformBuffer& SceneUniformBuffer,
352     bool bAnyLumenActive,
353     EDiffuseIndirectMethod DiffuseIndirectMethod,
354     EReflectionsMethod ReflectionsMethod,
355     const FPostProcessingInputs& Inputs,
356     const Nanite::FRasterResults* NaniteRasterResults,
357     FInstanceCullingManager& InstanceCullingManager,
358     FVirtualShadowMapArray* VirtualShadowMapArray,
359     FLumenSceneFrameTemporaries& LumenFrameTemporaries,
360     const FSceneWithoutWaterTextures& SceneWithoutWaterTextures,
361     FScreenPassTexture TSRFlickeringInput,
362     FRDGTextureRef& InstancedEditorDepthTexture)
363 {
364     RDG_CSV_STAT_EXCLUSIVE_SCOPE(GraphBuilder, RenderPostProcessing);
365     QUICK_SCOPE_CYCLE_COUNTER(STAT_PostProcessing_Process);
366     using namespace UE::Renderer::PostProcess;
367
368     check(IsInRenderingThread());
369 #if DO_CHECK || USING_CODE_ANALYSIS
370     check(View.VerifyMembersChecks());
371 #endif
372     Inputs.Validate();
373
374     FScene* Scene = View.Family->Scene->GetRenderScene();
375
376     const FIntRect PrimaryViewRect = View.ViewRect;
377
378     const FSceneTextureParameters SceneTextureParameters = GetSceneTextureParameters(GraphBuilder, Inputs.SceneTextures);
379
380     const FScreenPassRenderTarget ViewFamilyOutput = FScreenPassRenderTarget::CreateViewFamilyOutput(Inputs.ViewFamilyTexture, View);
381     const FScreenPassRenderTarget ViewFamilyDepthOutput = FScreenPassRenderTarget::CreateViewFamilyOutput(Inputs.ViewFamilyDepthTexture, View);
```

And then it is made the end point of a pass sequence that overrides the final target

```
499     const TCHAR* PassNames[] =
500     {
501         TEXT("MotionBlur"),
502         TEXT("PostProcessMaterial (SceneColorBeforeBloom)"),
503         TEXT("Tonemap"),
504         TEXT("FXAA"),
505         TEXT("PostProcessMaterial (SceneColorAfterTonemapping)"),
506         TEXT("VisualizeLumenScene"),
507         TEXT("VisualizeDepthOfField"),
508         TEXT("VisualizeStationaryLightOverlap"),
509         TEXT("VisualizeLightCulling"),
510         TEXT("VisualizePostProcessStack"),
511         TEXT("VisualizeSubstrate"),
512         TEXT("VisualizeLightGrid"),
513         TEXT("VisualizeSkyAtmosphere"),
514         TEXT("VisualizeSkyLightIlluminanceMeter"),
515         TEXT("VisualizeLightFunctionAtlas"),
516         TEXT("VisualizeLevelInstance"),
517         TEXT("VisualizeVirtualShadowMaps_PreEditorPrimitives"),
518         TEXT("SelectionOutline"),
519         TEXT("EditorPrimitive"),
520         TEXT("VisualizeVirtualShadowMaps_PostEditorPrimitives"),
521         TEXT("VisualizeVirtualTexture"),
522         TEXT("VisualizeShadingModels"),
523         TEXT("VisualizeGBufferHints"),
524         TEXT("VisualizeSubsurface"),
525         TEXT("VisualizeGBufferOverview"),
526         TEXT("VisualizeLumenSceneOverview"),
527         TEXT("VisualizeHDR"),
528         TEXT("VisualizeLocalExposure"),
529         TEXT("VisualizeMotionVectors"),
530         TEXT("VisualizeTemporalUpscaler"),
531         TEXT("PixelInspector"),
532         TEXT("HMDDistortion"),
533         TEXT("HighResolutionScreenshotMask"),
534 #if UE_ENABLE_DEBUG_DRAWING
535         TEXT("DebugPrimitive"),
536 #endif
537         TEXT("PrimaryUpscale"),
538         TEXT("SecondaryUpscale"),
539         TEXT("AlphaInvert")
540     };
541
542     static_assert(static_cast<uint32>(EPass::MAX) == UE_ARRAY_COUNT(PassNames), "EPass does not match PassNames.");
543
544     TOverridePassSequence<EPass> PassSequence(ViewFamilyOutput);
545     PassSequence.SetNames(PassNames, UE_ARRAY_COUNT(PassNames));
```

Warping

Jump now back to GameThread, towards the end of UDisplayClusterViewportClient::**Draw**, well after View Families have been rendered –and all its lambdas have been enqueued in the render thread-, we find a final step (line 831):

```
825
826     {
827         //ensure canvas has been flushed before rendering UI
828         SceneCanvas->Flush_GameThread();
829
830         // After all render target rendered call nDisplay frame rendering
831         RenderFrameViewportManager->RenderFrame(InViewport);
832
833         OnDrawn().Broadcast();
834
835         // Allow the viewport to render additional stuff
836         PostRender(DebugCanvasObject);
837     }
```

Notice the InViewport that is passed. It is of type FViewport*

```
389
390 void UDisplayClusterViewportClient::Draw(FViewport* InViewport, FCanvas*
        SceneCanvas)
391 {
```

RenderFrame in turn calls ViewportManagerProxy::ImplRenderFrame_GameThread():

```
882
883 void FDisplayClusterViewportManager::RenderFrame(FViewport* InViewport)
884 {
885     LightCardManager->RenderFrame();
886
887     ViewportManagerProxy->ImplRenderFrame_GameThrea(InViewport);
888 }
```

Proxies are allowed to have GameThread methods as long as they do not touch their data members in that same thread, and in fact this last method will only enqueue a few more lambdas in the render thread. These lambdas come after all the previous ones, and s areince they run in order, you can imagine that when RenderThread gets to run these, all the previous RenderThread lambdas will be already executed. Now focus on UpdateDeferredResources_RenderThread:

```
191
192     void FDisplayClusterViewportManagerProxy::ImplRenderFrame_GameThread(FViewport* InViewport)
193     {
194         ENQUEUE_RENDER_COMMAND(DisplayClusterRenderFrame_Setup)(
195             [InViewportManagerProxy = SharedThis(this)](FRHICommandListImmediate& RHICmdList) { ... });
205
206         ENQUEUE_RENDER_COMMAND(DisplayClusterRenderFrame_CrossGPUTransfer)(
207             [InViewportManagerProxy = SharedThis(this), OutputViewport = InViewport](FRHICommandListImmediate&
                RHICmdList) { ... });
232
233         ENQUEUE_RENDER_COMMAND(DisplayClusterRenderFrame_UpdateDeferredResources)(
234             [ViewportManagerProxy = SharedThis(this)](FRHICommandListImmediate& RHICmdList)
235         {
236             RHI_BREADCRUMB_EVENT_STAT(RHICmdList, nDisplay_ViewportManager_UpdateDeferredResources,
                  "nDisplay_ViewportManager_UpdateDeferredResources");
237             SCOPED_GPU_STAT(RHICmdList, nDisplay_ViewportManager_UpdateDeferredResources);
238
239             // Update viewports resources: vp/texture overlay, OCIO, blur, nummips, etc
240             ViewportManagerProxy->UpdateDeferredResources_RenderThread(RHICmdList);
241         });
242
```

This method goes through every viewport proxy (remember we switched to render thread because it's running in the lambda) and updates its resources

```
288
289     void FDisplayClusterViewportManagerProxy::UpdateDeferredResources_RenderThread(FRHICommandListImmediate& RHICmdList) const
290     {
291         check(IsInRenderingThread());
292
293         // Viewports in the CurrentRenderFrameViewportProxies list are already sorted using GetPriority_RenderThread().
294         for (const TSharedPtr<FDisplayClusterViewportProxy, ESPMode::ThreadSafe>& ViewportProxy : ImplGetCurrentRenderFrameViewportProxies_RenderThread())
295         {
296             if (ViewportProxy.IsValid())
297             {
298                 ViewportProxy->UpdateDeferredResources(RHICmdList);
299             }
300         }
301     }
```

Inside of the update there is a texture copy with the following source and destination

```
255
256     EDisplayClusterViewportResourceType SrcResourceType = EDisplayClusterViewportResourceType::InternalRenderTargetResource;
257
258     // pre-Pass 0 (Projection policy):The projection policy can use its own method to resolve 'InternalRenderTargetResource' to 'InputShaderResource'
259     if (ProjectionPolicy.IsValid() && ProjectionPolicy->ResolveInternalRenderTargetResource_RenderThread(RHICmdList, this, &SourceViewportProxy))
260     {
261         SrcResourceType = EDisplayClusterViewportResourceType::InputShaderResource;
262     }
263
264     if (GetOpenColorIOMode() == EDisplayClusterViewportOpenColorIOMode::Resolved)
265         {
266         // Pass 0:  OCIO + Linear gamma
267         // At this point Resolver go to use RDG
268         OpenColorIO->AddPass_RenderThread(
269             FDisplayClusterShadersTextureUtilsSettings(),
270             GetShadersAPI().CreateTextureUtils_RenderThread(RHICmdList)
271                 ->SetInput(&SourceViewportProxy, SrcResourceType)
272                 ->SetOutput(this, EDisplayClusterViewportResourceType::InputShaderResource));
273         }
274     else
275     {
276         // Pass 0: Linear gamma
277         GetShadersAPI().CreateTextureUtils_RenderThread(RHICmdList)
278             ->SetInput(&SourceViewportProxy, SrcResourceType)
279             ->SetOutput(this, EDisplayClusterViewport  SrcResourceType | InternalRenderTargetResource (2 '\x2')
280             ->Resolve();
281     }
282
283     // (Opt) Pass 1: Generate blur postprocess effect for render target texture rect for all contexts
```

Source is of type InternalRenderTargetResource, but we will see in a moment that the actual resource used as input is type RenderTargets (i.e. the ViewFamily render target resouce, where postprocess rendering is performed) as it gets translated.

First GetTextureParametersFromViewport is called

```
445
446  TSharedRef<IDisplayClusterShadersTextureUtils> FDisplayClusterShadersTextureUtils::SetInput(
447      const IDisplayClusterViewportProxy* InViewportProxy,
448      const EDisplayClusterViewportResourceType InResourceType,
449      const int32 InContextNum)
450  {
451      const FDisplayClusterShadersTextureParameters NewTextureParameters = GetTextureParametersFromViewport(InViewportProxy, InResourceType);
452
453      if (InContextNum == INDEX_NONE)
454      {
455          // Override entire input parameters.
456          SetInputEncoding(NewTextureParameters.ColorEncoding);
457          for (const TPair<uint32, FDisplayClusterShadersTextureViewport>& TextureIt : NewTextureParameters.TextureViewports)
458          {
459              SetInput(TextureIt.Value, TextureIt.Key);
460          }
461      }
462      else if(NewTextureParameters.TextureViewports.Contains(InContextNum))
```

Then Resources (with Rects) are obtained from InViewportProxy

```
398
399  FDisplayClusterShadersTextureParameters FDisplayClusterShadersTextureUtils::GetTextureParametersFromViewport(
400      const IDisplayClusterViewportProxy* InViewportProxy,
401      const EDisplayClusterViewportResourceType InResourceType)
402  {
403      using namespace UE::DisplayClusterShaders::Private;
404
405      FDisplayClusterShadersTextureParameters OutTextureParameters;
406
407      TArray<FRHITexture*> Textures;
408      TArray<FIntRect> TextureRects;
409      if (InViewportProxy && InViewportProxy->GetResourcesWithRects_RenderThread(InResourceType, Textures, TextureRects)
410          && Textures.Num() == TextureRects.Num())
411      {
412          // Get resource color encoding
413          OutTextureParameters.ColorEncoding = InViewportProxy->GetResourceColorEncoding_RenderThread(InResourceType);
414
```

Which after a couple jumps gets to this function

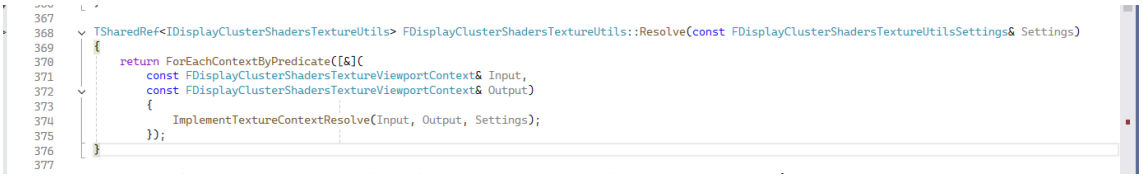
```
78
79  bool FDisplayClusterViewportProxy::ImplGetResources_RenderThread(const EDisplayClusterViewportResourceType InExtResourceType, TArray<FRHITexture*>&
        OutResources, const int32 InRecursionDepth) const
80  {
81      using namespace UE::DisplayCluster::ViewportProxy;
82      check(IsInRenderingThread());
83
84      const EDisplayClusterViewportResourceType InResourceType = GetResourceType_RenderThread(InExtResourceType);
85
86      // Override resources from other viewport
87      if (ShouldOverrideViewportResource(InResourceType))
88      {
89          if (InRecursionDepth < DisplayClusterViewportProxyResourcesOverrideRecursionDepthMax)
90          {
91              return GetRenderingViewportProxy().ImplGetResources_RenderThread(InExtResourceType, OutResources, InRecursionDepth + 1);
92          }
93
94          return false;
95      }
96
97      OutResources.Empty();
98
99      switch (InResourceType)
100     {
            InResourceType | InternalRenderTargetResource (2 '\x2')
101     case EDisplayClusterViewportResourceType::InternalRenderTargetEntireRectResource:
102     case EDisplayClusterViewportResourceType::InternalRenderTargetResource:
103     {
104         bool bResult = false;
105
106         if (Contexts.Num() > 0)
107         {
108
109             // 1. Replace RTT from configuration
110             if (!bResult && PostRenderSettings.Replace.IsEnabled()) { ... }
123
124             // 2. Replace RTT from UVLightCard:
125             if (!bResult && EnumHasAnyFlags(RenderSettingsICVFX.RuntimeFlags, EDisplayClusterViewportRuntimeICVFXFlags::UVLightcard)) { ... }
149
150             // 3. Finally Use InternalRTT
151             if (!bResult)
152             {
153                 bResult = Resources.GetRHIResources_RenderThread(EDisplayClusterViewportResource::RenderTargets, OutResources);
154             }
```

Notice in line 99 how InResourceType in the switch gets translated from **EDisplayClusterViewportResourceType::InternalRenderTargetResource**, to the queried **EDisplayClusterViewportResouce::RenderTargets**, just as the enum comment said it would be.

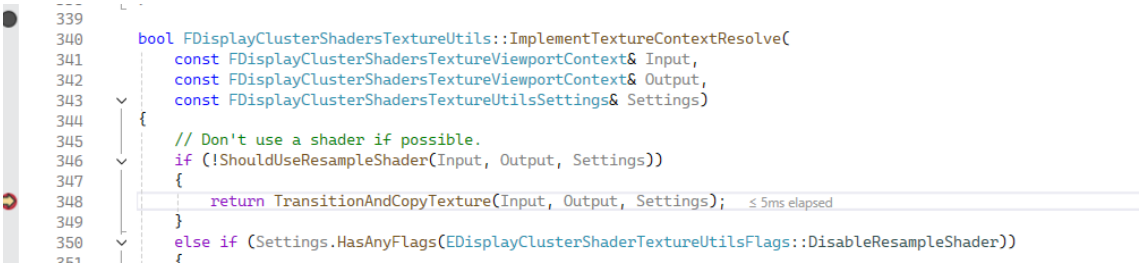
In the end, Resolve() just means a texture copy, where source is the render target color for the viewport as just shown, and output is the resource marked as the InputShaderResource type:

```
274     else
275     {
276         // Pass 0: Linear gamma
277         GetShadersAPI().CreateTextureUtils_RenderThread(RHICmdList)
278             ->SetInput(&SourceViewportProxy, SrcResourceType)
279             ->SetOutput(this, EDisplayClusterViewportResourceType::InputShaderResource)
280             ->Resolve();
281     }
282
```

As mentioned resolve just means a texture copy of its contents



```
TSharedRef<IDisplayClusterShadersTextureUtils> FDisplayClusterShadersTextureUtils::Resolve(const FDisplayClusterShadersTextureUtilsSettings& Settings)
{
    return ForEachContextByPredicate([&](
        const FDisplayClusterShadersTextureViewportContext& Input,
        const FDisplayClusterShadersTextureViewportContext& Output)
        {
            ImplementTextureContextResolve(Input, Output, Settings);
        });
}
```

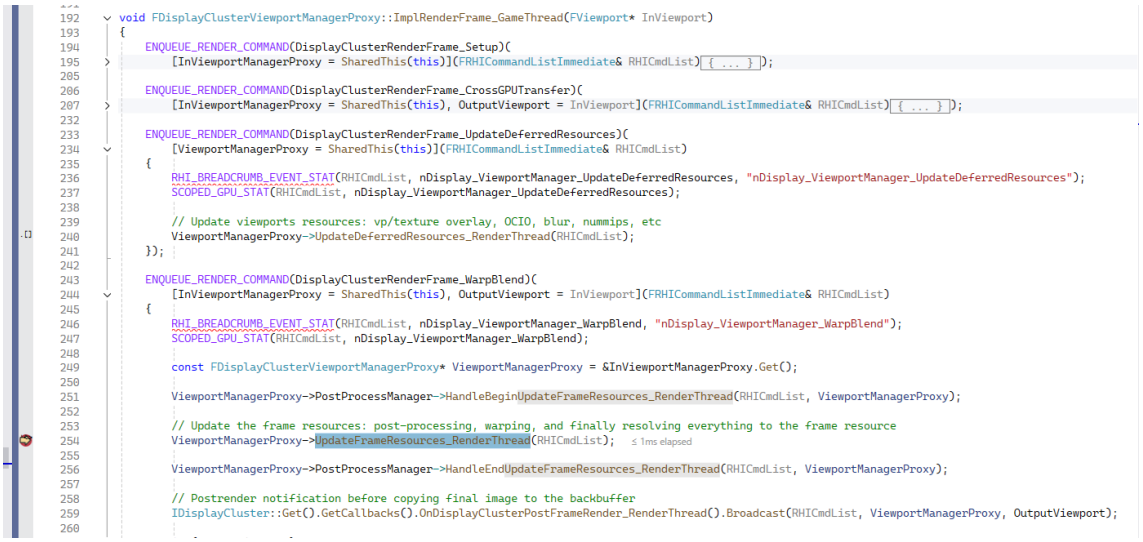


```
bool FDisplayClusterShadersTextureUtils::ImplementTextureContextResolve(
    const FDisplayClusterShadersTextureViewportContext& Input,
    const FDisplayClusterShadersTextureViewportContext& Output,
    const FDisplayClusterShadersTextureUtilsSettings& Settings)
{
    // Don't use a shader if possible.
    if (!ShouldUseResampleShader(Input, Output, Settings))
    {
        return TransitionAndCopyTexture(Input, Output, Settings);   ≤ 5ms elapsed
    }
    else if (Settings.HasAnyFlags(EDisplayClusterShaderTextureUtilsFlags::DisableResampleShader))
```

After a couple jumps it all ends in commands for copy texture

```
inline void TransitionAndCopyTexture(FRHICommandList& RHICmdList, FRHITexture* SrcTexture, FRHITexture* DstTexture, const FRHICopyTextureInfo& Info)
    ≤ 3ms elapsed
    check(SrcTexture && DstTexture);
    check(SrcTexture->GetNumSamples() == DstTexture->GetNumSamples());

    if (SrcTexture == DstTexture)
    {
        RHICmdList.Transition({
            FRHITransitionInfo(SrcTexture, ERHIAccess::Unknown, ERHIAccess::SRVMask)
        });
        return;
    }

    RHICmdList.Transition({
        FRHITransitionInfo(SrcTexture, ERHIAccess::Unknown, ERHIAccess::CopySrc),
        FRHITransitionInfo(DstTexture, ERHIAccess::Unknown, ERHIAccess::CopyDest)
    });

    RHICmdList.CopyTexture(SrcTexture, DstTexture, Info);

    RHICmdList.Transition({
        FRHITransitionInfo(SrcTexture, ERHIAccess::CopySrc,  ERHIAccess::SRVMask),
        FRHITransitionInfo(DstTexture, ERHIAccess::CopyDest, ERHIAccess::SRVMask)
    });
}
```

Hence the render target color is in its destination, everything is ready for warping, let's jump back to where we left off, to the next lambda:



```
void FDisplayClusterViewportManagerProxy::ImplRenderFrame_GameThread(FViewport* InViewport)
{
    ENQUEUE_RENDER_COMMAND(DisplayClusterRenderFrame_Setup)(
        [InViewportManagerProxy = SharedThis(this)](FRHICommandListImmediate& RHICmdList){ ... });

    ENQUEUE_RENDER_COMMAND(DisplayClusterRenderFrame_CrossGPUTransfer)(
        [InViewportManagerProxy = SharedThis(this), OutputViewport = InViewport](FRHICommandListImmediate& RHICmdList){ ... });

    ENQUEUE_RENDER_COMMAND(DisplayClusterRenderFrame_UpdateDeferredResources)(
        [ViewportManagerProxy = SharedThis(this)](FRHICommandListImmediate& RHICmdList)
    {
        RHI_BREADCRUMB_EVENT_STAT(RHICmdList, nDisplay_ViewportManager_UpdateDeferredResources, "nDisplay_ViewportManager_UpdateDeferredResources");
        SCOPED_GPU_STAT(RHICmdList, nDisplay_ViewportManager_UpdateDeferredResources);

        // Update viewports resources: vp/texture overlay, OCIO, blur, nummips, etc
        ViewportManagerProxy->UpdateDeferredResources_RenderThread(RHICmdList);
    });

    ENQUEUE_RENDER_COMMAND(DisplayClusterRenderFrame_WarpBlend)(
        [InViewportManagerProxy = SharedThis(this), OutputViewport = InViewport](FRHICommandListImmediate& RHICmdList)
    {
        RHI_BREADCRUMB_EVENT_STAT(RHICmdList, nDisplay_ViewportManager_WarpBlend, "nDisplay_ViewportManager_WarpBlend");
        SCOPED_GPU_STAT(RHICmdList, nDisplay_ViewportManager_WarpBlend);

        const FDisplayClusterViewportManagerProxy* ViewportManagerProxy = &InViewportManagerProxy.Get();

        ViewportManagerProxy->PostProcessManager->HandleBeginUpdateFrameResources_RenderThread(RHICmdList, ViewportManagerProxy);

        // Update the frame resources: post-processing, warping, and finally resolving everything to the frame resource
        ViewportManagerProxy->UpdateFrameResources_RenderThread(RHICmdList);   ≤ 1ms elapsed

        ViewportManagerProxy->PostProcessManager->HandleEndUpdateFrameResources_RenderThread(RHICmdList, ViewportManagerProxy);

        // Postrender notification before copying final image to the backbuffer
        IDisplayCluster::Get().GetCallbacks().OnDisplayClusterPostFrameRender_RenderThread().Broadcast(RHICmdList, ViewportManagerProxy, OutputViewport);
```

In this lambda, first the render targets are cleared (remember that its contents have been just copied so its ok to clear it)

```
327
328 void FDisplayClusterViewportManagerProxy::UpdateFrameResources_RenderThread(FRHICommandListImmediate& RHICmdList) const
329 {
330     check(IsInRenderingThread());
331
332     // Do postprocess before warp&blend
333     PostProcessManager->PerformPostProcessViewBeforeWarpBlend_RenderThread(RHICmdList, this);
334
335     // Support viewport overlap order sorting:
336     TArray<TSharedPtr<FDisplayClusterViewportProxy, ESPMode::ThreadSafe>> SortedViewportProxy = ImplGetCurrentRenderFrameViewportProxies_RenderThread();
337     SortedViewportProxy.Sort(
338         [](const TSharedPtr<FDisplayClusterViewportProxy, ESPMode::ThreadSafe>& VP1, const TSharedPtr<FDisplayClusterViewportProxy, ESPMode::ThreadSafe>&
        VP2)
339         {
340             return  VP1->GetRenderSettings_RenderThread().OverlapOrder < VP2->GetRenderSettings_RenderThread().OverlapOrder;
341         }
342     );
343
344     // Clear Frame RTT resources before viewport resolving
345     const bool bClearFrameRTTEnabled = CVarClearFrameRTTEnabled.GetValueOnRenderThread() != 0;
346     if (bClearFrameRTTEnabled)
347     {
348         ImplClearFrameTargets_RenderThrea(RHICmdList);
349     }
350
351     IDisplayCluster::Get().GetCallbacks().OnDisplayClusterPreWarp_RenderThread().Broadcast(RHICmdList, this);
352
```

```
302
303 void FDisplayClusterViewportManagerProxy::ImplClearFrameTargets_RenderThread(FRHICommandListImmediate&
    RHICmdList) const
304 {
305     TArray<FRHITexture*> FrameResources;
306     TArray<FRHITexture*> AdditionalFrameResources;
307     TArray<FIntPoint> TargetOffset;
308     if (GetFrameTargets_RenderThread(FrameResources, TargetOffset, &AdditionalFrameResources))
309     {
310         for (FRHITexture* FrameResourceIt : FrameResources)
311         {
312             FDisplayClusterViewportProxy::FillTextureWithColor_RenderThread(RHICmdList,
                FrameResourceIt, FLinearColor::Black);
313         }
314     }
315 }
```

Render commands fill the render target texture with black

```
61
62 void FDisplayClusterViewportProxy::FillTextureWithColor_RenderThread(FRHICommandListImmediate& RHICmdList, FRHITexture* InRenderTargetTexture, const
    FLinearColor& InColor)
63 {
64     if (InRenderTargetTexture)  ≤ 2ms elapsed
65     {
66         FRHIRenderPassInfo RPInfo(InRenderTargetTexture, ERenderTargetActions::DontLoad_Store);
67         RHICmdList.Transition(FRHITransitionInfo(InRenderTargetTexture, ERHIAccess::Unknown, ERHIAccess::RTV));
68         RHICmdList.BeginRenderPass(RPInfo, TEXT("nDisplay_FillTextureWithColor"));
69         {
70             const FIntPoint Size = InRenderTargetTexture->GetSizeXY();
71             RHICmdList.SetViewport(0, 0, 0.0f, Size.X, Size.Y, 1.0f);
72             DrawClearQuad(RHICmdList, FLinearColor::Black);
73         }
74         RHICmdList.EndRenderPass();
75         RHICmdList.Transition(FRHITransitionInfo(InRenderTargetTexture, ERHIAccess::Unknown, ERHIAccess::SRVMask));
76     }
77 }
78
```

Then warping is invoked

```
353     // Handle warped viewport projection policy logic:
354     for (uint8 WarpPass = 0; WarpPass < (uint8)EWarpPass::COUNT; WarpPass++)  ≤ 1ms elapsed
355     {
356         // Update deferred resources for viewports
357         for (TSharedPtr<FDisplayClusterViewportProxy, ESPMode::ThreadSafe>& ViewportProxyIt : SortedViewportProxy)
358         {
359             // Iterate over visible viewports:
360             if (ViewportProxyIt.IsValid() && ViewportProxyIt->GetRenderSettings_RenderThread().bVisible)
361             {
362                 if (ViewportProxyIt->ShouldApplyWarpBlend_RenderThread())
363                 {
364                     const TSharedPtr<IDisplayClusterProjectionPolicy, ESPMode::ThreadSafe>& PrjPolicy = ViewportProxyIt->GetProjectionPolicy_RenderThread
                        ();
365                     switch ((EWarpPass)WarpPass)
366                     {
367                     case EWarpPass::Begin:
368                         IDisplayCluster::Get().GetCallbacks().OnDisplayClusterPreWarpViewport_RenderThread().Broadcast(RHICmdList, ViewportProxyIt.Get());
369                         PrjPolicy->BeginWarpBlend_RenderThread(RHICmdList, ViewportProxyIt.Get());
370                         break;
371
372                     case EWarpPass::Render:
373                         PrjPolicy->ApplyWarpBlend_RenderThrea(RHICmdList, ViewportProxyIt.Get());
374                         break;
375
376                     case EWarpPass::End:
377                         PrjPolicy->EndWarpBlend_RenderThread(RHICmdList, ViewportProxyIt.Get());
378                         IDisplayCluster::Get().GetCallbacks().OnDisplayClusterPostWarpViewport_RenderThread().Broadcast(RHICmdList, ViewportProxyIt.Get
                            ());
379                         break;
380
381                     default:
382                         break;
383                     }
384                 }
385             }
386         }
387     }
388
389     // per-frame handle
390     PostProcessManager->HandleUpdateFrameResourcesAfterWarpBlend_RenderThread(RHICmdList, this);
391
392     // Per-view postprocess
```

Notice that input for warping is the output of the previous RenderTargets copy. This was the output for the copy:

```
274         else
275         {
276             // Pass 0: Linear gamma
277             GetShadersAPI().CreateTextureUtils_RenderThread(RHICmdList)
278                 ->SetInput(&SourceViewportProxy, SrcResourceType)
279                 ->SetOutput(this, EDisplayClusterViewportResourceType::InputShaderResource)
280                 ->Resolve();
281         }
282
```

And this is the input for warping

```
221 void FDisplayClusterProjectionEasyBlendPolicy::ApplyWarpBlend_RenderThread(FRHICommandListImmediate& RHICmdList, const IDisplayClusterViewportProxy*
      InViewportProxy)
222 {
223     check(IsInRenderingThread());
224     check(InViewportProxy);
225
226     if (!PolicyViewDataProxy.IsValid())
227     {
228         return;
229     }
230
231     // Get in\out remp resources ref from viewport
232     TArray<FRHITexture*> InputTextures, OutputTextures;
233     if (InViewportProxy->GetResources_RenderThread(EDisplayClusterViewportResourceType::InputShaderResource, InputTextures)
234         && InViewportProxy->GetResources_RenderThread(EDisplayClusterViewportResourceType::AdditionalTargetableResource, OutputTextures))
235     {
236         check(InputTextures.Num() == OutputTextures.Num());
237         check(InViewportProxy->GetContexts_RenderThread().Num() == InputTextures.Num());
238
239         TRACE_CPUPROFILER_EVENT_SCOPE(FDisplayClusterProjectionEasyBlendPolicy::ApplyWarpBlend_RenderThread);
240
241         // Warp all viewport contexts
242         for (int32 ContextNum = 0; ContextNum < PolicyViewInfoProxy.Num(); ContextNum++)
243         {
244             if (InputTextures.IsValidIndex(ContextNum) && OutputTextures.IsValidIndex(ContextNum))
245             {
246                 PolicyViewDataProxy->ApplyWarpBlend_RenderThread(RHICmdList, PolicyViewInfoProxy[ContextNum], InputTextures[ContextNum], OutputTextures
                      [ContextNum], RHIViewportProxy);
247             }
248         }
249     }
250 }
```

Warping uses two textures - input (InputShaderResource, the RenderTargets color just copied) and output (AdditionalTargetableResource, a render target resource created in BeginNewFrame)- and the warping mesh, as well as the point of view. Notice how the output texture is first transitioned from unknown to Render (RTV) and then to RTV to Present.

```
151
152     // Setup In/Out EasyBlend textures
153     RHICmdList.Transition(FRHITransitionInfo(OutputTexture, ERHIAccess::Unknown, ERHIAccess::RTV));
154     // Insert an outer query that encloses the whole batch
155     RHICmdList.EnqueueLambda(
156     [ViewData = SharedThis(this), InputTexture = InputTexture, OutputTexture = OutputTexture, InViewLocation = InViewInfo.ViewLocation](FRHICommandList& ExecutingCmdList)
157     {
158         // Block multi-threaded access to EasyBlendMeshData
159         FScopeLock DataScopeLock(&ViewData->EasyBlendMeshDataAccessCS);
160         TSharedRef<FDisplayClusterProjectionEasyBlendLibraryDX12, ESPMode::ThreadSafe>& EasyBlendAPI = FDisplayClusterProjectionEasyBlendLibraryDX12::Get();
161
162         ID3D12DynamicRHI* D3D12DynamicRHI = GetID3D12DynamicRHI();
163         if (!D3D12DynamicRHI)
164         {
165             return;
166         }
167
168         // Ensure EasyBlend uses the same GPU device as the source texture to support multi-GPU configurations
169         const uint32 DevIndex = D3D12DynamicRHI->RHIGetResourceDeviceIndex(InputTexture);
170         ID3D12Device* D3D12Device = D3D12DynamicRHI->RHIGetDevice(DevIndex);
171         if (!D3D12Device)
172         {
173             return;
174         }
175
176         // Update view location for proxy
177         const EasyBlendSDKError Result1 = EasyBlendAPI->EasyBlendSDK_SetEyepoint(ViewData->EasyBlendMeshData.Get(), InViewLocation.X, InViewLocation.Y, InViewLocation.Z);
178
179         // Initialize EasyBlend internals
180         if (!ViewData->bIsRenderResourcesInitialized)
181         {
182             const EasyBlendSDKError Result = EasyBlendAPI->EasyBlendSDK_InitializeDX12_Device(ViewData->EasyBlendMeshDat.Get(), D3D12Device);
183             if (!EasyBlendSDK_SUCCEEDED(Result))
184             {
185                 ViewData->bIsRenderResourcesInitialized = false;
186
187                 return;
188             }
189
190             ViewData->bIsRenderResourcesInitialized = true;
191         }
192
193         ID3D12Resource* SrcTexture = D3D12DynamicRHI->RHIGetResource(InputTexture);
194         ID3D12Resource* DestTexture = D3D12DynamicRHI->RHIGetResource(OutputTexture);
195         const D3D12_CPU_DESCRIPTOR_HANDLE RTVHandle = D3D12DynamicRHI->RHIGetRenderTargetView(OutputTexture);
196         ID3D12GraphicsCommandList* D3D12GraphicsCommandList = D3D12DynamicRHI->RHIGetGraphicsCommandList(ExecutingCmdList, DevIndex);
197
198         if (SrcTexture && DestTexture && D3D12GraphicsCommandList)
199         {
200             // Render EasyBlend on proxy
201             const EasyBlendSDKError Result2 = EasyBlendAPI->EasyBlendSDK_TransformInputToOutputDX12_CommandList(ViewData->EasyBlendMeshData.Get(), SrcTexture, DestTexture, RTVHandle,
                  D3D12GraphicsCommandList);
202         }
203     });
204     RHICmdList.Transition(FRHITransitionInfo(OutputTexture, ERHIAccess::RTV, ERHIAccess::Present));
205
206     return true;
207 }
```

However the real presentation to the backbuffer happens later and it is a bit involved. After warping a post resolve step is executed

```
395     // Post resolve to Frame RTT
396     // All warp&blend results are now inside AdditionalTargetableResource. Viewport images of other
          projection policies are still stored in the InputShaderResource.
397     for (TSharedPtr<FDisplayClusterViewportProxy, ESPMode::ThreadSafe>& ViewportProxyIt :
          SortedViewportProxy)
398     {
399         // Iterate over visible viewports:
400         if (ViewportProxyIt.IsValid() && ViewportProxyIt->GetRenderSettings_RenderThread().bVisible)
401         {
402             ViewportProxyIt->PostResolveViewport_RenderThrea(RHICmdList);
403         }
404     }
405
406     IDisplayCluster::Get().GetCallbacks().OnDisplayClusterPostWarp_RenderThread().Broadcast(
          RHICmdList, this);
407
408     PostProcessManager->PerformPostProcessFrameAfterWarpBlend_RenderThread(RHICmdList, this);
409 }
```

In this step the texture result of warping is copied from AfterWarpBlendTargetableResource to OutputTargetableResources

```
160
161 void FDisplayClusterViewportProxy::PostResolveViewport_RenderThread(FRHICommandListImmediate& RHICmdList) const
162 {
163     // resolve warped viewport resource to the output texture
164     ResolveResources_RenderThread(RHICmdList, EDisplayClusterViewportResourceType::AfterWarpBlendTargetableResource,
          EDisplayClusterViewportResourceType::OutputTargetableResource);
165
166     // Implement ViewportRemap feature
167     ImplViewportRemap_RenderThread(RHICmdList);
168 }
169
```

When in fact AfterWarpBlendTargetableResource becomes our friend InputShaderResource in yet another Unreal's sleight of hand:

```
98
99  EDisplayClusterViewportResourceType FDisplayClusterViewportProxy::GetResourceType_RenderThread(const
      EDisplayClusterViewportResourceType& InResourceType) const
100 {
101     check(IsInRenderingThread());
102
103     switch (InResourceType)
104     {
105     case EDisplayClusterViewportResourceType::BeforeWarpBlendTargetableResource:
106         return EDisplayClusterViewportResourceType::InputShaderResource;
107
108     case EDisplayClusterViewportResourceType::AfterWarpBlendTargetableResource:
109         return ShouldApplyWarpBlend_RenderThread() ?
              EDisplayClusterViewportResourceType::AdditionalTargetableResource :
              EDisplayClusterViewportResourceType::InputShaderResource;
110
```

This step is just a texture copy again:

```
228
229     bool FDisplayClusterViewportProxy::ImplResolveResources_RenderThread(FRHICommandListImmediate& RHICmdList, FDis
            EDisplayClusterViewportResourceType InExtResourceType, const EDisplayClusterViewportResourceType OutExtResour
230     {
231         using namespace UE::DisplayCluster::ViewportProxy;
232
233         check(IsInRenderingThread());
234         check(SourceProxy);
235
236         const EDisplayClusterViewportResourceType InResourceType = SourceProxy->GetResourceType_RenderThread(InExtR
237         const EDisplayClusterViewportResourceType OutResourceType = GetResourceType_RenderThread(OutExtResourceType
238
239         if (InResourceType == EDisplayClusterViewportResourceType::MipsShaderResource)
240         {
241             // RenderTargetMips not allowved for resolve op
242             return false;
243         }
244
245         FDisplayClusterShadersTextureUtilsSettings TextureUtilsSettings;
246         // The mode used to blend textures
247         if (OutResourceType == EDisplayClusterViewportResourceType::OutputPreviewTargetableResource)
248         {
249             // The preview texture should use only RGB colors and ignore the alpha channel.
250             // The alpha channel may or may not be inverted in third-party libraries.
251             TextureUtilsSettings.OverrideAlpha = EDisplayClusterShaderTextureUtilsOverrideAlpha::Set_Alpha_One;
252         }
253
254         TSharedRef<IDisplayClusterShadersTextureUtils> TextureUtils =
255             GetShadersAPI().CreateTextureUtils_RenderThread(RHICmdList)
256                 ->SetInput(SourceProxy, InExtResourceType)
257                 ->SetOutput(this, OutExtResourceType);
258
259         if (InExtResourceType == EDisplayClusterViewportResourceType::AfterWarpBlendTargetableResource
260             && OutExtResourceType == EDisplayClusterViewportResourceType::OutputTargetableResource
261             && DisplayDeviceProxy.IsValid()
262             && DisplayDeviceProxy->HasFinalPass_RenderThread())
263         {
264             // Custom resolve at external Display Device
265             DisplayDeviceProxy->AddFinalPass_RenderThread(TextureUtilsSettings, TextureUtils);
266         }
267         else
268         {
269             // Standard resolve:
270             TextureUtils->Resolve(TextureUtilsSettings);   ≤ 1ms elapsed
271         }
272
273         return true;
274     }
```

Then in ResolveFrameTargetToBackBuffer the texture is finally copied to the true and only viewport rendertarget, with OutputViewport->GetRenderTargetTexture() as destination

```
253         // Update the frame resources: post-processing, warping, and finally resolving everything to the frame resource
254         ViewportManagerProxy->UpdateFrameResources_RenderThread(RHICmdList);
255
256         ViewportManagerProxy->PostProcessManager->HandleEndUpdateFrameResources_RenderThread(RHICmdList, ViewportManagerProxy);
257
258         // Postrender notification before copying final image to the backbuffer
259         IDisplayCluster::Get().GetCallbacks().OnDisplayClusterPostFrameRender_RenderThread().Broadcast(RHICmdList, ViewportManagerProxy, OutputViewport);
260
261         if (OutputViewport)
262         {
263             if (FRHITexture* FrameOutputRTT = OutputViewport->GetRenderTargetTexture())
264             {
265                 // For quadbuf stereo copy only left eye, right copy from OutputFrameTarget
266                 //@todo Copy QuadBuf_LeftEye/(mono,sbs,tp) to separate rtt, before UI and debug rendering
267                 //@todo QuadBuf_LeftEye copied latter, before present
268                 if(ViewportManagerProxy->ConfigurationProxy->GetRenderFrameSettings().ShouldUseStereoRenderingOnMonoscopicDisplay())
269                 {
270                     ViewportManagerProxy->ResolveFrameTargetToBackBuffer_RenderThread(RHICmdList, 1, 0, FrameOutputRTT, FrameOutputRTT->GetSizeXY());
271                 }
272
273                 ViewportManagerProxy->ResolveFrameTargetToBackBuffer_RenderThread(RHICmdList, 0, 0, FrameOutputRTT, FrameOutputRTT->GetSizeXY());   ≤ 2ms elapsed
274
275                 // Finally, notify about backbuffer update
276                 IDisplayCluster::Get().GetCallbacks().OnDisplayClusterPostBackbufferUpdated_RenderThread().Broadcast(RHICmdList, OutputViewport);
277 PRAGMA_DISABLE_DEPRECATION_WARNINGS
278                 IDisplayCluster::Get().GetCallbacks().OnDisplayClusterPostBackbufferUpdate_RenderThread().Broadcast(RHICmdList, ViewportManagerProxy, OutputViewpo
279 PRAGMA_ENABLE_DEPRECATION_WARNINGS
280             }
281         }
```

And the frame target is the source

```cpp
bool FDisplayClusterViewportManagerProxy::ResolveFrameTargetToBackBuffer_RenderThread(FRHICommandListImmediate& RHICmdList, const
 uint32 InContextNum, const int32 DestArrayIndex, FRHITexture* DestTexture, FVector2D WindowSize) const
{
    check(IsInRenderingThread());

    if (!DestTexture)
    {
        return false;
    }

    // Output texture must supports quadbuffer stereo.
    if (DestArrayIndex > 0)
    {
        // Check if the backbuffer texture has a second slice for stereo.
        const bool bStereoTexture = DestTexture->GetDesc().Dimension == ETextureDimension::Texture2DArray && DestTexture->GetDesc
          ().ArraySize > 1;
        if (!bStereoTexture)
        {
            return false;
        }
    }

    TArray<FRHITexture*>   FrameResources;
    TArray<FIntPoint>        TargetOffsets;
    if (GetFrameTargets_RenderThread(FrameResources, TargetOffsets) && FrameResources.IsValidIndex(InContextNum))
    {
        // Use internal frame textures as source
        if (FRHITexture* FrameTexture = FrameResources[InContextNum])
        {
            FIntRect SrcRect(FIntPoint::ZeroValue, FrameTexture->GetDesc().Extent);

            const FIntPoint& DestOffset = TargetOffsets[InContextNum];
            FIntRect DestRect(DestOffset, DestOffset + FrameTexture->GetDesc().Extent);

            // Check if resources with the specified regions can be resolved.
            if (!FDisplayClusterViewportHelpers::GetValidResourceRectsForResolve(FrameTexture->GetDesc().Extent, DestTexture->GetDes
              ().Extent, SrcRect, DestRect))
            {
                // The SrcRect or DestRect is invalid.
                return false;
            }

            FRHICopyTextureInfo CopyInfo;

            CopyInfo.SourceSliceIndex = 0;
            CopyInfo.DestSliceIndex = DestArrayIndex;

            CopyInfo.SourcePosition = FIntVector(SrcRect.Min.X, SrcRect.Min.Y, 0);
            CopyInfo.DestPosition = FIntVector(DestRect.Min.X, DestRect.Min.Y, 0);

            CopyInfo.Size = FIntVector(DestRect.Width(), DestRect.Height(), 0);

            TransitionAndCopyTexture(RHICmdList, FrameTexture, DestTexture, CopyInfo);  ≤ 1ms elapsed

            return true;
        }
    }
```

Tadah! And an image is produced.

TODO: Analyze whether FDisplayClusterMeshProjectionRenderer could be an example source for custom warping.