Plugin IrradianceDXR - Primer Pase Funcional (Resumen)

1. Introducción del Plugin

1.1 Contenido del plugin

Shaders HLSL de prueba (`RayTracingIrradiance.usf`): el programa gráfico que se ejecuta en la GPU. Se han implementado dos shaders: un ray-gen, y un closest-hit. Para el miss (no colisión) se ha usado un shader por defecto del motor. De momento no se trazan rayos como tal: tan sólo se escribe una textura (color rojo) en GPU. Estos shaders se declaran e implementan en C++ en `RayTracingIrradianceRGS.h/.cpp` (ray-gen) y en `RayTracingIrradianceCHS.h/.cpp` (closest-hit).

Módulo IrradianceDXR (`IrradianceDXR.cpp/.h`): el código principal del módulo. En él se registran tanto los comandos *TestPyrano, TestPyranoRead*, como el "Delegate" que "engancha" un pase de renderizado en la parte correcta del mismo (*OnPostOpaqueRender*).

- El comando *TestPyrano*, con muchas comillas "ejecuta" el shader (activa el pase de renderizado). CPU->GPU.
- El comando *TestPyranoRead* lee la textura que escribe el shader. GPU->CPU.

Clase IrradiancePass: encapsula toda la lógica del render y el readback; es decir, es donde ocurre casi todo. Tiene tres funciones principales:

- *AddPass_Render*: Construye el pass RDG + dispatch de rayos.
- *EnqueuePassAndReadback*: copia la textura de salida a CPU.
- *TryConsumeReadback*: devuelve el valor de la textura (log).

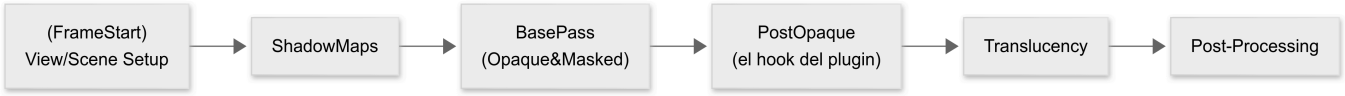
1.2 Flujo de ejecución

1. Se abre proyecto, se abre nivel y se da al play (*PlayInEditor*).
2. Se abre la consola y se ejecuta *TestPyrano*.
3. En el siguiente `PostOpaque` (fase de renderizado) se dispara `FIrradiancePass::RenderAndEnqueueReadback`.
4. Se construye textura de salida, y se encola un pass de RDG.
5. Se encola un readback GPU->CPU.
6. Tras 1-2 frames, se puede ejecutar *TestPyranoRead* para recibir el resultado.

2. Funcionamiento resumido del plugin

2.1 IrradianceDXR: *PostOpaque, Delegate*

Vamos primero a *IrradianceDXR.cpp*, código principal del plugin.
En cada frame, el pipeline de renderizado pasa por varias etapas. En el siguiente *flowchart* resumo las más relevantes.



En el plugin, nos enganchamos al hilo de renderizado en *PostOpaque*: en esta fase es donde se ejecutará nuestro shader. Está listo todo lo útil para la irradiancia, aunque no está listo aún **Translucency** (vidrios, partículas, agua; Volumetric Clouds no entraría en Translucency, ni siquiera es geometría "raytraceable" que pertenezca a la TLAS) ni **PostProcessing** (que nos conviene, pues "decora" la medida real). La propia API pública del motor facilita engancharse en esta parte del renderizado.

Así pues, se registra un `PostOpaqueRenderDelegate`, que llama a la función `OnPostOpaqueRender` en esa parte del proceso. Esto ocurre al inicio del módulo, en `FIrradianceDXRModule::StartupModule()`. En el inicio del módulo también se registran los comandos para ejecutar el shader y leer el resultado.

```
// Registrar el Post-Opaque delegate
    IRendererModule& RendererModule =
FModuleManager::LoadModuleChecked<IRendererModule>("Renderer");
    GPostOpaqueHandle = RendererModule.RegisterPostOpaqueRenderDelegate(
        FPostOpaqueRenderDelegate::CreateStatic(&OnPostOpaqueRender));
```

En la función `OnPostOpaqueRender`, se evalúa si el comando ha sido ejecutado. Si lo ha sido, se obtienen la *View* y la *Scene* (clases de UE necesarias para casi todo). Para el renderizado, hay que obtener la clase *FViewInfo*, que hereda de *FSceneView* e incluye la información necesaria del renderizado. Al final, se llama a la función `RenderAndEnqueueReadback`.

2.2 IrradiancePass: *Pase RDG*

En este código se implementa el pase RDG, y se definen tres funciones importantes: `EnqueuePassAndReadback`, `AddPass_Render`, y `TryConsumeReadback`. En este .cpp se implementa el paso del shader a la GPU, usando la API de RDG. La estructura de capas del renderizado se puede ver así en Unreal Engine:

Por lo que en la práctica, hay dos formas de realizar tareas en GPU:

- Usando directamente RHI.
- Utilizando Render Dependency Graph (forma recomendada), la API que expone Unreal Engine y que gestiona automáticamente la sincronización de recursos, dependencias entre passes, memoria y ejecución en GPU.

Retomando el flujo del plugin, en `OnPostOpaqueRender` se llamaba a `RenderAndEnqueueReadback`. En él tenemos:

```
    FRDGTextureRef Out = AddPass_Render(
        GraphBuilder, Scene, View, Resolution,
        SensorOriginWS, SensorNormalWS, MaxBounces, TemporalSeed);

    if (Out) {
        EnqueuePassAndReadback(GraphBuilder, Out);
    }
```

Aquí aparecen las dos funciones más importantes: `AddPass_Render` se encarga de lanzar el pase / la orden al RDG para que se ejecute el shader. `EnqueuePassAndReadback` se encarga de lanzar otro pase al RDG (un Readback) que lea el valor de la textura del shader.

En `EnqeuePassAndReadback`, UE ya expone una función para lanzar la orden al RDG de que lea una textura: `AddReadbackTexturePass`.

```
            if (!TextureReadback.IsValid()) {
                TextureReadback = MakeUnique<FRHIGPUTextureReadback>
(TEXT("PyranometerReadback"));
            }
            TextureReadback->EnqueueCopy(RHICmdList, OutTex->GetRHI(), Rect);
        });
```

Por último, `AddPass_Render` es la función más importante: donde se crea y se lanza el pase del shader ray-gen. A diferencia de implementar un compute shader normal, implementar un shader ray tracing no es trivial, y requiere configurar bastantes cosas de forma manual. Explicaré el código en la siguiente sección, aunque el funcionamiento general del plugin ya está definido y son detalles menores.

2.3 AddPass_Render

De entrada, antes de la función, se construyen 2 estructuras y un método constructor para facilitar la estructura de la propia función.

`FIrradianceRTDispatch` guarda los parámetros que se pasarán al pase RDG.

```
struct FIrradianceRTDispatch
{
    FRayTracingIrradianceRGS::FParameters* Params = nullptr;
    FRHIRayTracingShader* RayGen = nullptr;
    FRHIRayTracingShader* ClosestHit = nullptr;
    FRHIUniformBuffer* SceneUB = nullptr;
    FRHIUniformBuffer* NaniteUB = nullptr;
    FIntPoint Resolution = FIntPoint::ZeroValue;
};
```

`FRTBuiltPipeline` guarda tanto el *PipelineState* (PSO), como la *Shader Binding Table* (SBT).

- *PipelineState*: se puede definir como la configuración de la tubería de ray tracing. Indica qué shaders se usan, qué payload manejan y cómo se organizan. Funciona a modo de "contrato" a seguir por la GPU.
- *SBT*: es una tabla en GPU que enlaza shaders concretos con los datos que necesitan.

Se construye con `BuildRTPSOAndSBT`.

```
struct FRTBuiltPipeline
{
    FRayTracingPipelineState* Pipeline = nullptr;
    FShaderBindingTableRHIRef  SBT;
};
```

Pasando a la función AddPass_Render:
Primero, se crea la textura de salida (que luego retornará la propia función).

```
    // Creates output texture
    FRDGTextureDesc Desc = FRDGTextureDesc::Create2D(
        Resolution,
        PF_FloatRGBA,
        FClearValueBinding::Black,
        TexCreate_UAV | TexCreate_ShaderResource);
    FRDGTextureRef OutTex = GraphBuilder.CreateTexture(Desc,
TEXT("Pyranometer.Out"));
```

Posteriormente se obtienen los shaders, y se enlazan los parámetros del shader (y definidos en C++ en las clases `FRayTracingIrradianceCHS`, `FRayTracingIrradianceRGS`) con los parámetros reales de la escena.

```
    // Parameter binding
    FRayTracingIrradianceRGS::FParameters* Params =
GraphBuilder.AllocParameters<FRayTracingIrradianceRGS::FParameters>();
Params->OutRadiance = GraphBuilder.CreateUAV(OutTex);
Params->TLAS = Scene-
>RayTracingScene.GetLayerView(ERayTracingSceneLayer::Base);
Params->View = View.ViewUniformBuffer;
Params->SceneTextures = CreateSceneTextureShaderParameters(
        GraphBuilder, View, ESceneTextureSetupMode::All);
```

Luego se obtienen los Uniform Buffers, buffers estáticos necesarios para que el binding sea correcto y que necesita la GPU. Son matrices vista/proyección, posición de cámara, parámetros de iluminación global...

```
// Uniform buffers
FRHIUniformBuffer* SceneUniformBuffer = GetSceneUniformBufferRef(GraphBuilder,
View)->GetRHI();
FRHIUniformBuffer* NaniteUB =
Nanite::GetPublicGlobalRayTracingUniformBuffer()->GetRHI(); // necesario en UE
5.6.1
```

Por último, se rellena el Dispatch con los parámetros, y se pasa el dispatch al `GraphBuilder.Add_Pass`, lo que realmente crea el pase. Dentro de él, se crea el PSO+SBT (con el método anterior). También se reservan parámetros temporales en un buffer GPU, y se meten los parámetros RDG (los mismos del shader, y de las clases C++).

```
FRHIBatchedShaderParameters& Globals =
RHICmdList.GetScratchShaderParameters(); // necesario en 5.6.1; non-batched
parameters planned for deprecation
SetShaderParameters(Globals, View.ShaderMap-
>GetShader<FRayTracingIrradianceRGS>(), *Dispatch->Params);
```

Se enlazan los buffers estáticos:

```

// Binding
TOptional<FScopedUniformBufferStaticBindings> StaticUniformBufferScope =
RayTracing::BindStaticUniformBufferBindings(View, Dispatch->SceneUB, Dispatch-
>NaniteUB, RHICmdList); // w/o nanite -> error

```

Y por último, se añaden los shaders, se asocia el SBT y se hace dispatch.

```
RHICmdList.SetDefaultRayTracingHitGroup(Built.SBT, Built.Pipeline, 0);
RHICmdList.SetRayTracingMissShader(Built.SBT, 0, Built.Pipeline, 0, 0,
nullptr, 0);
RHICmdList.CommitShaderBindingTable(Built.SBT);
RHICmdList.RayTraceDispatch(
    Built.Pipeline,
    Dispatch->RayGen,
    Built.SBT,
    Globals,
    Dispatch->Resolution.X,
    Dispatch->Resolution.Y); // API 5.6
```

Nota adicional: RHICmdList se refiere a RHI Command List: representa la lista de comandos de GPU que se están grabando en ese momento. Todo lo que se quiera hacer en GPU, se emite como comandos dentro de esta lista.

3. Siguientes pasos

- Comprobar si es posible reutilizar el Path Tracer del motor para hacer el shader (lanzar primer rayo y gestionar rebotes con código UE).
- Si no es posible o no merece la pena, implementar cálculo de irradiancia con RT normal.
- Implementar método de Scene Capture con Path Tracing (capturar 5 vistas con Path Tracing, evaluar píxeles).
- Implementar métodos secundarios más sencillos, pero a modo de "vista rápida": SceneCubes con Lumen, obtener rayo directo...