Extending the UE4 Mobile Renderer
by Example

Jack Porter
Dmitriy Dyomin
통역: 홍성진

Content

1. Unreal Engine 4 Mobile Renderer Overview
   - Mobile rendering pipeline
   - Renderer debugging
2. Extending the Mobile Renderer
   - Example: Implementing “Custom Depth”

PART 1

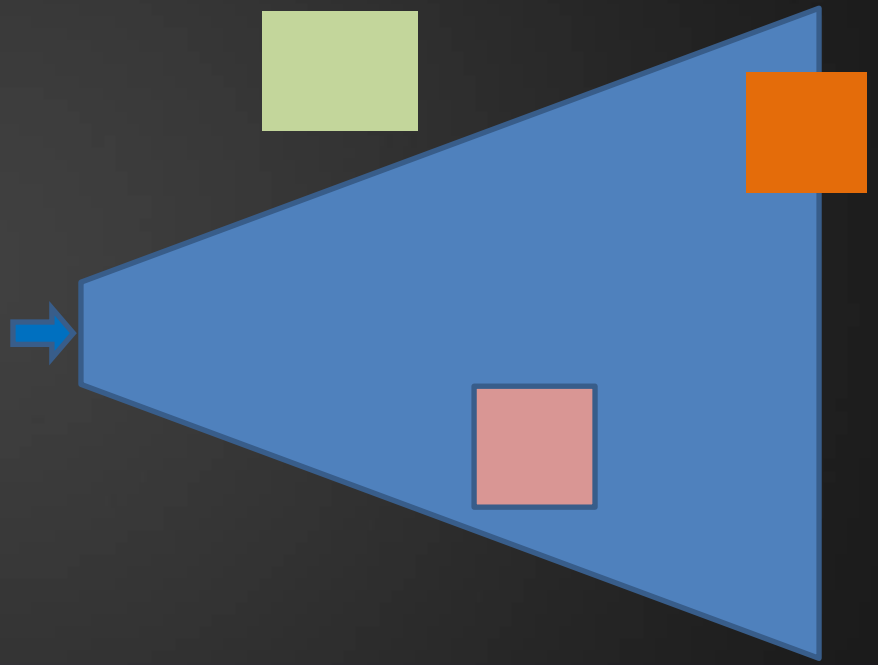
UE4 MOBILE RENDERER OVERVIEW

Forward Mobile Renderer

1. Views setup
2. GPU particles simulation
3. Shadow maps
4. Base pass
5. Deferred decals
6. Modulated shadows projections
7. Translucency
8. Post-process
9. HUD

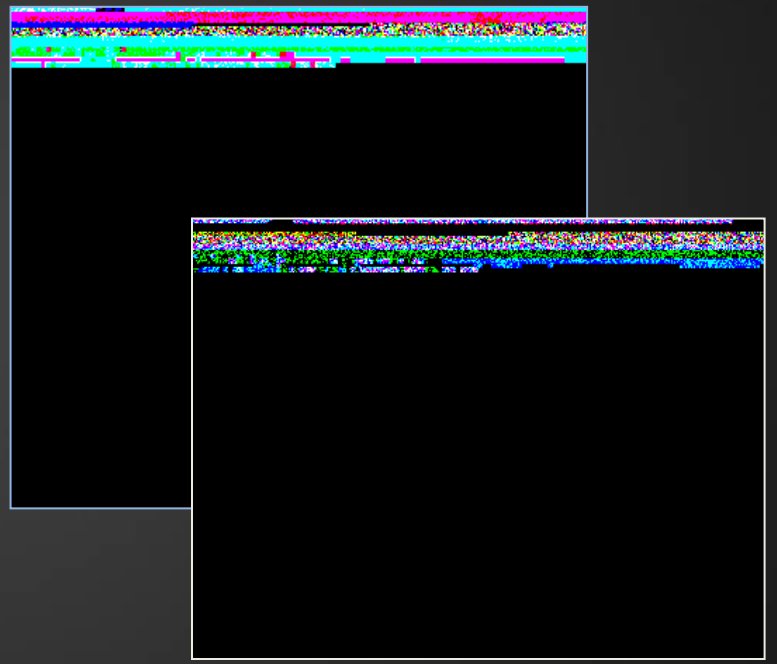
1. View setup

- Find out all visible objects
  - Frustum culling
  - Distance culling
  - Precomputed visibility
- View relevance
- Find out all visible shadows
- Gather dynamic mesh elements
- Update view uniform buffer



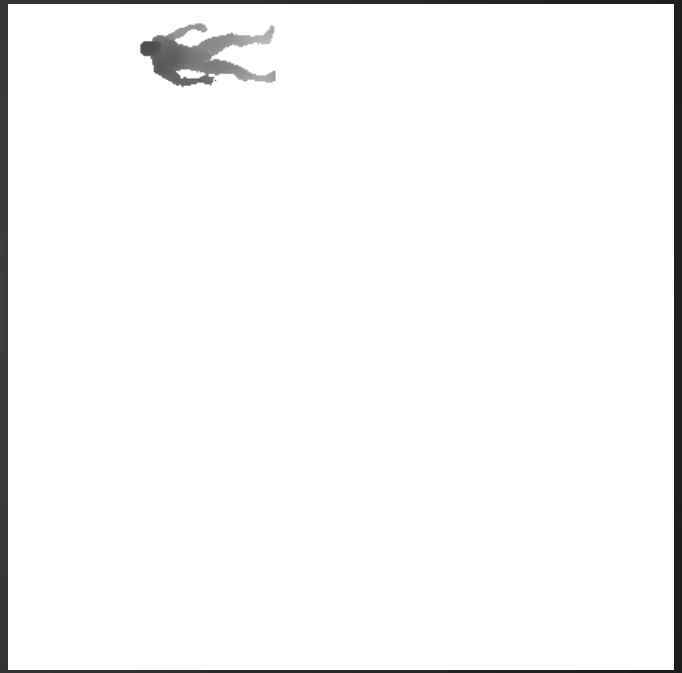
2. GPU particles simulation

- Requires device with ES 3.1 or Metal
- Simulates particle physics on GPU
- Writes particle Position to 128bpp target
- Writes particle Velocity to 64bpp target



3. Shadow maps

- Setup depth render target
- Find out which objects need shadows
- Render them using main light view
- Shadow map used later
  - base pass for CSM shadows
  - modulated shadow projections



4. Base pass (setup)

- Choose appropriate shaders depending on shadow and lighting properties
  - Unlit
  - Distance field shadows + LM
  - Distance field shadows + LM + CSM
  - Dynamic lighting (CSM)
  - etc

- For dynamic elements, this happens every frame

- For static elements this occurs once when the scene is created, and the mesh drawing state is added to a static draw list.
  - Primitives are grouped by material, vertex factory to reduce OpenGL state changes

4. Base pass (drawing)

Renders all objects with opaque materials

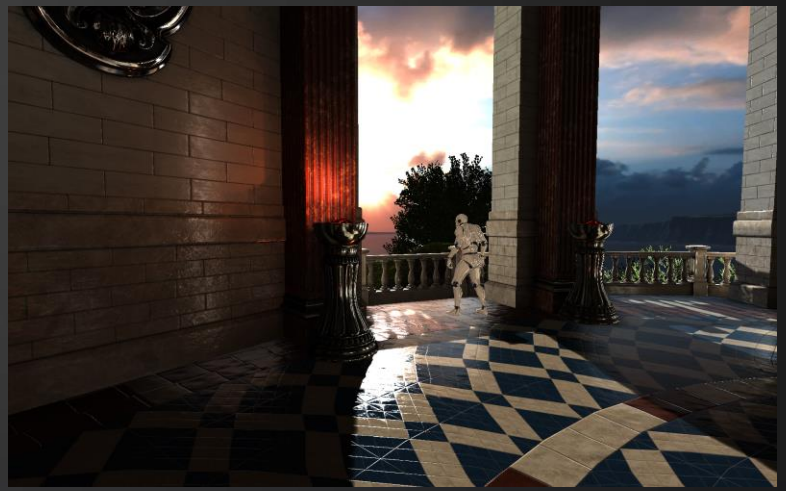
- static and dynamic directional and point lights (per-pixel)
- static distance field shadows
- dynamic CSM shadows

Drawing order depends on device

1. Draw without any reordering
   - Default option for ImgTec, which has “Hidden Surface Removal”
2. Reorder meshes front to back in each list
   - Minimizes state change
3. Reorder meshes front to back across all lists
   - Minimizes overdraw
   - Default option for non-ImgTec GPUs

Can be tweaked depending on your content

- r.ForwardBasePassSort = x



5. Deferred decals

- Requires scene depth fetch
  - Implementation depends on supported extensions
  - `GL_ARM_shader_framebuffer_fetch_depth_stencil`
  - `GL_EXT_shader_framebuffer_fetch`
  - `GL_OES_depth_texture`
    - depth buffer resolve
- Supports “Receives Decals” flag
  - Stencil operations
- Does not support lighting

6. Modulated shadow projections

- Similar to deferred decals
- Does not blend well with static lighting
- Static + CSM is a better option (4.12+)

7. Translucency

- Draw primitives with non-Opaque blend mode using the same shaders as the Base Pass
- Does not write to depth buffer
- Refraction supported
  - Requires full copy of scene color

8. Post-Process

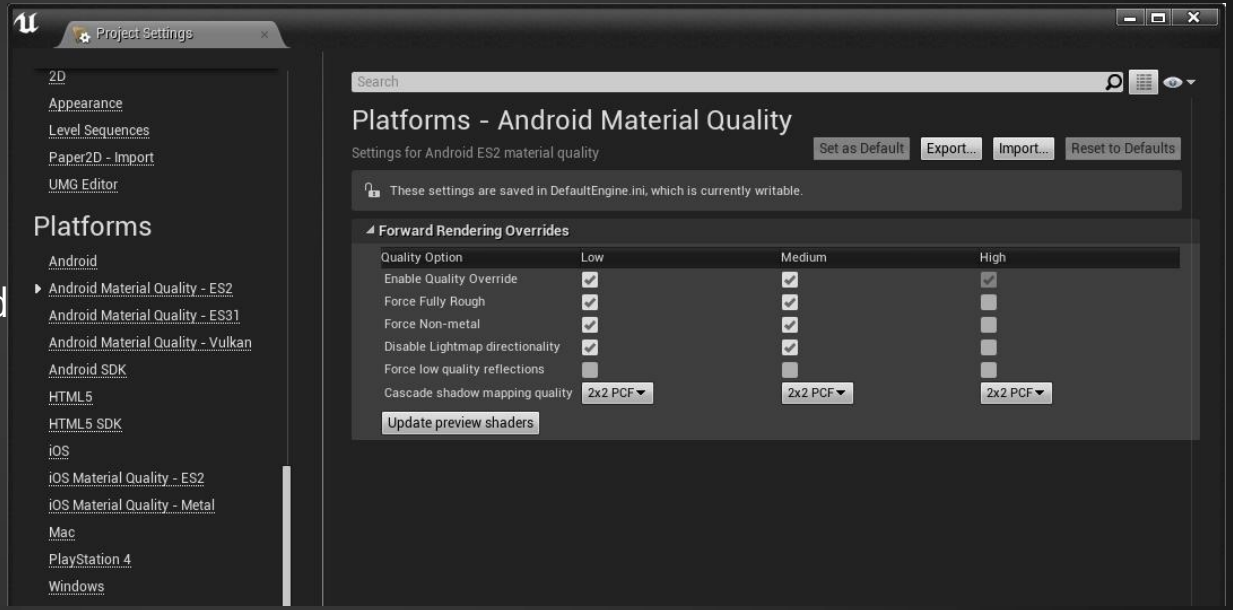
- Only supported when HDR is enabled
- Requires several passes depending on what effects are used
  - Depth of Field
  - Custom PostProcess Materials
- Tonemapper pass at the end
  - Maps HDR color to 8-bit per-channel RGB and writes it to backbuffer
  - Bloom is also applied at this stage

9. HUD

- Draw UI elements directly to backbuffer
  - Slate / Widget Blueprints
  - Canvas
- Swap backbuffer

Material Quality Levels

- Specifies Material shading quality and features
- Each QL is a new sets of shaders
- Which QL to use is decided at app startup
  - eg by Device Profiles
- Preview in Editor
  - Settings->Material Quality Level



Debugging Mobile Issues

- Make sure problem does not exists on PC (Deferred, PIE Mobile, feature level preview)
- Enable `r.DumpShaderDebugInfo` to see output of shader compilation
  - Will dump processed HLSL and cross compiled GLSL into `Saved\ShaderDebugInfo` folder
  - Useful for debugging!
- Watch log output, as it may report shader compilation errors
- Enable ENABLE_VERIFY_GL (disabled by default)
- Use vendor specific tools (Adreno Profiler, Mali Graphics Debugger, XCode, …)

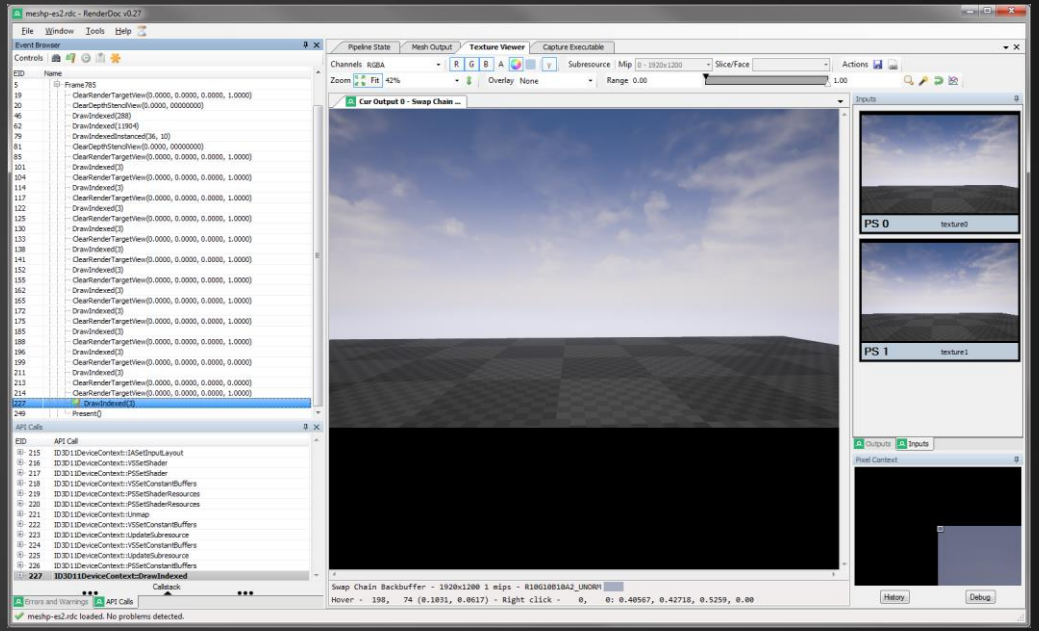
Renderer Debugging: RenderDoc

RenderDoc is very useful for debugging when adding to the renderer

- PC tool that captures Direct3D state
- show a breakdown of the scene drawcalls

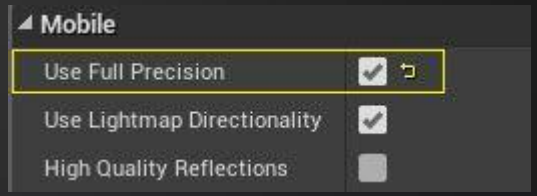
- Allows you to inspect the entire graphics pipeline
  - vertex buffer contents
  - vertex shader output
  - pixel shader input textures and output

- To debug mobile, use a scene running in ES2 or ES3.1 feature level



Watch out - Precision

- Be careful when using mediump with functions like
  - `length()`
  - `normalize()`
  - `distance()`
- mediump is usually in: -2^{14} … 2^{14} range
  - `length = sqrt(v.x`^{2}`+ v.y`^{2}`+ v.z`^{2}`)`
- Make sure that vector magnitude is less 2^7 (128)
- `length (vec3(0, 129, 0))` may produce INF on some GPUs (eg Mali)
  - subsequent calculations using the INF value will also be INF or NaN
  - other GPUs seem to clamp
- From 4.13, Material can request full precision

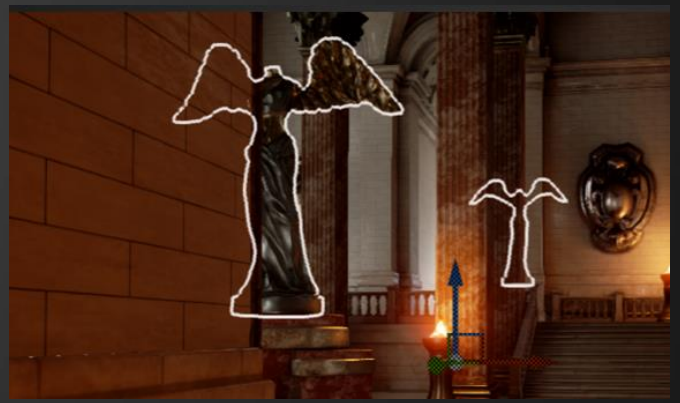


PART 2

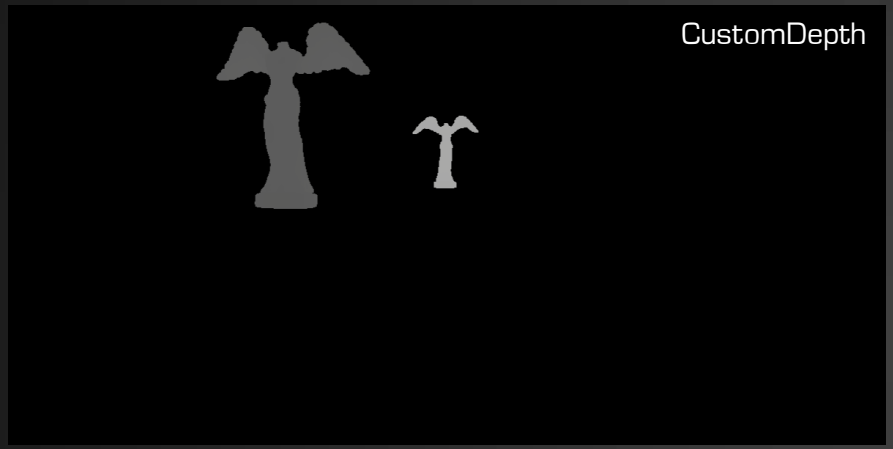
EXTENDING THE UE4 MOBILE RENDERER

Custom Depth

- Custom Depth is a feature of the desktop UE4 renderer we wanted to bring to mobile
- It renders a second depth buffer for some objects the user specifies
- Allows all kinds of useful effects when combined with custom post-process, eg:
  - Draw silhouettes for characters or objects occluded by geometry
  - Draw outlines around objects
  - Highlight or mask parts of the scene

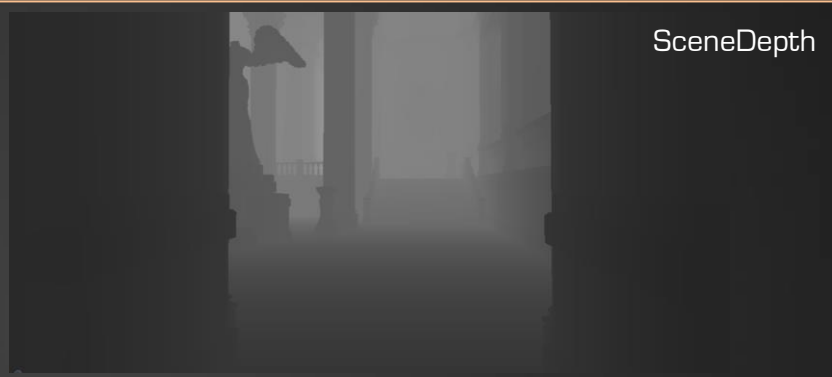


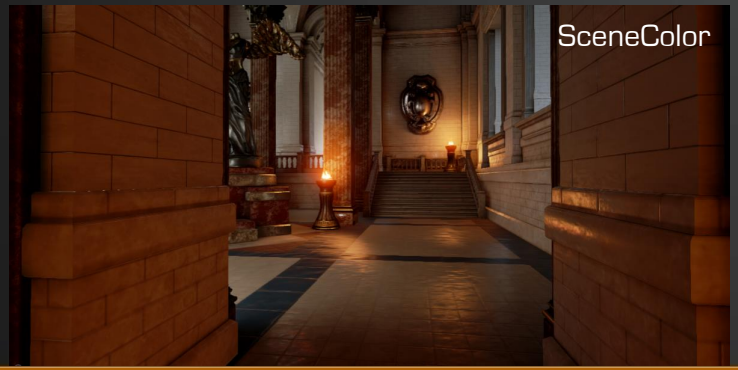
Example: Outline Effect



- Sample a pixel and the 4 neighboring pixels from CustomDepth
- Output highlight color if total difference is bigger than some threshold

Example: Hidden Silhouette Effect

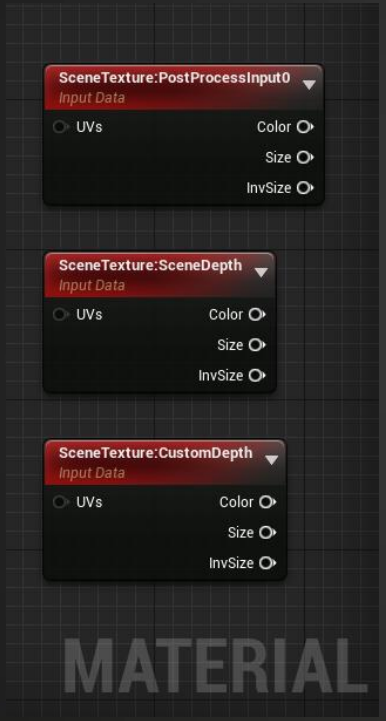




- Output highlight color if CustomDepth bigger than SceneDepth
- Output SceneColor otherwise

Example: Radar Scope Effect

Custom Post-process scene textures



- Scene color texture after post-processing
  - Mobile support added in 4.13
- Scene depth of all opaque and masked objects
  - Linear value between Near and Far plane
  - Mobile support added in 4.14
- Depth of objects that are marked as CustomDepth
  - Mobile support added in 4.14

Steps Required for Custom Depth

4. During custom post-processing, make use of an extra render target containing some depth values as a texture

…. and to do that we need to:

3. Add a render pass to draw some set of primitives (Components) to an extra render target, with a depth pixel shader

…. and to do that we need to:

2. Identify the primitives that will receive Custom Depth and store them in a list somewhere

…. and also

1. Allocate a new render target for Custom Depth

1. Identifying Primitives to Render

UE4 typically uses flags on either the Material or the Component to indicate it needs special handling by the renderer

PrimitiveComponent.h

UPrimitiveComponent

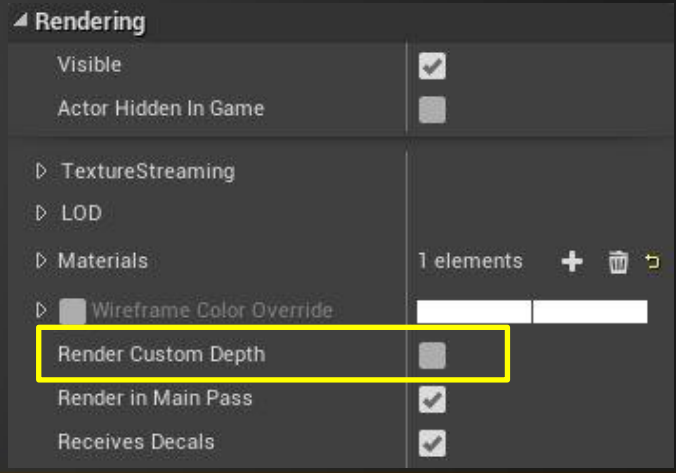
class ENGINE_API UPrimitiveComponent : public USceneComponent, public INavRelevantInterface

UE4

UPrimitiveComponent

```
/** If true, this component will be rendered in the CustomDepth pass (usually used for outlines) */
UPROPERTY(EditAnywhere, AdvancedDisplay, BlueprintReadOnly, Category=Rendering, meta=(DisplayName =
uint32 bRenderCustomDepth:1;
```

For Custom Depth, it's a `UPrimitiveComponent` property



1. Identifying Primitives to Render

- UE4 uses a separate Rendering Thread which has its own representation of `PrimitiveComponent` called `FPrimitiveSceneProxy`.
- This contains all the information required to render the component
- `FPrimitiveSceneProxy` also has a `bRenderCustomDepth` flag which is set by the constructor

View Relevance

Each frame, the `InitViews` function calculates visibility for all primitives.

`FPrimitiveViewRelevance` contains information about how we will render the primitive for the current view

Each primitive type implements `FPrimitiveSceneProxy::GetViewRelevance` to set this information

```
struct FPrimitiveViewRelevance
{
    /** The LightingProfile supported by this primitive, as a bitmask. */
    uint16 ShadingModelMaskRelevance;
    /** The primitive has one or more opaque or masked elements. */
    uint32 bOpaqueRelevance : 1;
    /** The primitive has one or more masked elements. */
    uint32 bMaskedRelevance : 1;
    /** The primitive has one or more distortion elements. */
    uint32 bDistortionRelevance : 1;
    /** The primitive has one or more elements that have SeparateTranslucency. */
    uint32 bSeparateTranslucencyRelevance : 1;
    /** The primitive has one or more elements that have MobileSeparateTranslucency. */
    uint32 bMobileSeparateTranslucencyRelevance : 1;
    /** The primitive has one or more elements that have normal translucency. */
    uint32 bNormalTranslucencyRelevance : 1;
    /** */
    uint32 bUsesGlobalDistanceField : 1;


    /** The primitive's static elements are rendered for the view. */
    uint32 bStaticRelevance : 1;
    /** The primitive's dynamic elements are rendered for the view. */
    uint32 bDynamicRelevance : 1;
    /** The primitive is drawn. */
    uint32 bDrawRelevance : 1;
    /** The primitive is casting a shadow. */
    uint32 bShadowRelevance : 1;
    /** The primitive should render to the custom depth pass. */
    uint32 bRenderCustomDepth : 1;
    /** The primitive should render to the base pass / normal depth / velocity rendering. */
    uint32 bRenderInMainPass : 1;
    /** The primitive is drawn only in the editor and composited onto the scene after post process
```

1. Identifying Primitives to Render

- The `ComputeRelevance` function in `SceneVisibility.cpp` calls `GetViewRelevance` for all visible primitives

- This gives us an opportunity to add visible Custom Depth primitives to a list we can use to render the custom depth pass later

SceneVisibility.cpp

FPrimitiveViewRelevance.bRenderCustomDepth — uint32 bRenderCustomDepth : 1

UE4

```
if (ViewRelevance.bRenderCustomDepth)
{
    // Add to set of dynamic distortion primitives
    CustomDepthSet.AddPrim(PrimitiveSceneInfo->Proxy);
}
```

2. Allocating a Render Target

- `FSceneRenderTargets` is a container for all render targets used for rendering the scene
  - Render targets are allocated from a fixed-size pool
  - We need to add a render target reference and an on-demand allocator
  - The reference be deallocated from pool to free memory on LRU basis

```
IPooledRenderTarget* FSceneRenderTargets::RequestCustomDepth(FRHICommandListImmediate& RHICmdList)
{
    if (!CustomDepth.GetReference() || BufferSize != CustomDepth->GetDesc().Extent)
    {
        FPooledRenderTargetDesc Desc(FPooledRenderTargetDesc::Create2DDesc(BufferSize, PF_DepthStencil, FClearValueBinding::DepthFar,
            TexCreate_None, TexCreate_DepthStencilTargetable, false));
        GRenderTargetPool.FindFreeElement(RHICmdList, Desc, CustomDepth, TEXT("CustomDepth"));
    }
    return CustomDepth;
}
```

3. Adding a Render Pass

- We need to render our custom depth pass from `FMobileSceneRenderer::Render`, which renders one frame

- When should we render custom depth pass?
  - Custom depth doesn't have any dependencies
  - We want to make use of the custom depth in post processing (end of frame)
  - Switching render targets is expensive on mobile

- So it makes sense to render the custom depth at the start of the frame

Rendering: FMobileSceneRenderer::Render

Calculate visibility

Render shadow depth buffers

Render Custom Depths

Set color RenderTarget

Render base pass

Render decals & mod shadows

Render translucency

Render post-processing

```
void FMobileSceneRenderer::Render(FRHICommandListImmediate& RHICmdList)
{
    //make sure all the targets we're going to use will be safely writable.
    GRenderTargetPool.TransitionTargetsWritable(RHICmdList);

    // Find the visible primitives.
    InitViews(RHICmdList);

    RenderShadowDepthMaps(RHICmdList);

    const bool bGammaSpace = !IsMobileHDR();

    if (!bGammaSpace)
    {
        RenderCustomDepthPass(RHICmdList);
    }

    // Begin rendering to scene color
    SceneContext.BeginRenderingSceneColor(RHICmdList, ESimpleRenderTargetMode::EClearColorAndDepth);

    RHICmdList.Clear(true, Views[0].BackgroundColor, false, (float)ERHIZBuffer::FarPlane, false, 0, FIntRect());

    RenderMobileBasePass(RHICmdList);

    ConditionalResolveSceneDepth(RHICmdList);

    RenderDecals(RHICmdList);

    RenderModulatedShadowProjections(RHICmdList);

    RenderTranslucency(RHICmdList);

    // Resolve the scene color for post processing.
    SceneContext.ResolveSceneColor(RHICmdList, FResolveRect(0, 0, ViewFamily.FamilySizeX, ViewFamily.FamilySizeY));

    if (!bKeepDepthContent)
    {
        RHICmdList.DiscardRenderTargets(true, true, 0);
    }

    GPostProcessing.ProcessES2(RHICmdList, Views[ViewIndex], bOnChipSunMask);

    RenderFinish(RHICmdList);
}
```

Rendering: RenderCustomDepthPass

Check if we have any CustomDepth primitives at all

Set CustomDepth render target

FCustomDepthPrimSet::DrawPrims

```
void FSceneRenderer::RenderCustomDepthPass(FRHICommandListImmediate& RHICmdList)
{
    // do we have primitives in this pass?
    bool bPrimitives = false;

    if(!Scene->World || (Scene->World->WorldType != EWorldType::EditorPreview && Scene->World->WorldType != EWorldType::Inactive))
    {
        for(int32 ViewIndex = 0; ViewIndex < Views.Num(); ++ViewIndex)
        {
            const FViewInfo& View = Views[ViewIndex];
            if(View.CustomDepthSet.NumPrims())
            {
                bPrimitives = true;
                break;
            }
        }
    }

    // Render CustomDepth
    FSceneRenderTargets& SceneContext = FSceneRenderTargets::Get(RHICmdList);
    if (SceneContext.BeginRenderingCustomDepth(RHICmdList, bPrimitives))
    {
        SCOPED_DRAW_EVENT(RHICmdList, CustomDepth);
        SCOPED_GPU_STAT(RHICmdList, Stat_GPU_CustomDepth);

        for(int32 ViewIndex = 0;ViewIndex < Views.Num();ViewIndex++)
        {
            SCOPED_CONDITIONAL_DRAW_EVENTF(RHICmdList, EventView, Views.Num() > 1, TEXT("View%d"), ViewIndex);

            FViewInfo& View = Views[ViewIndex];

            RHICmdList.SetViewport(View.ViewRect.Min.X, View.ViewRect.Min.Y, 0.0f, View.ViewRect.Max.X, View.ViewRect.Max.Y, 1.0f);

            // seems this is set each draw call anyway
            RHICmdList.SetRasterizerState(TStaticRasterizerState<>::GetRHI());
            RHICmdList.SetBlendState(TStaticBlendState<>::GetRHI());

            const bool bWriteCustomStencilValues = SceneContext.IsCustomDepthPassWritingStencil();

            if (!bWriteCustomStencilValues)
            {
                RHICmdList.SetDepthStencilState(TStaticDepthStencilState<true, CF_DepthNearOrEqual>::GetRHI());
            }

            View.CustomDepthSet.DrawPrims(RHICmdList, View, bWriteCustomStencilValues);
        }

        // resolve using the current ResolveParams
        SceneContext.FinishRenderingCustomDepth(RHICmdList);
    }
}
```

Rendering the Primitives

`FCustomDepthPrimSet::DrawPrims`

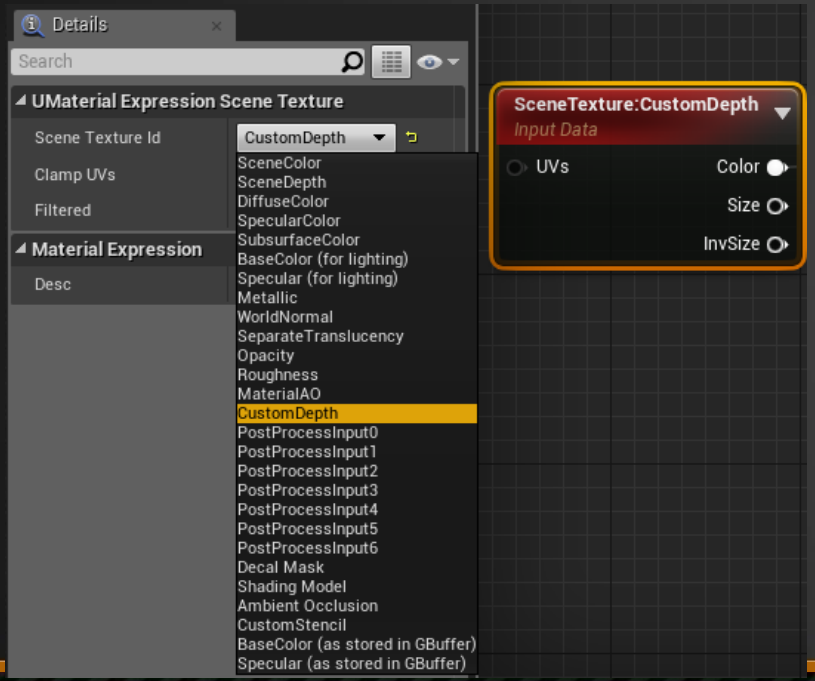
- For each primitive in the set, draw the primitive with `FDepthDrawingPolicyFactory`

`FDepthDrawingPolicyFactory`

- Sets the blend mode
- Sets the depth vertex and pixel shaders
  - Depth-only pixel shaders are generated for masked materials to evaluate the Opacity Mask input
  - Non-masked materials all share the same pixel shader
- Sets shader-specific uniforms
- Draws the primitive

4. Using CustomDepth in Post Process Material

Material Editor / SceneTexture node



C++ (HLSLMaterialTranslator.h)

```
virtual int32 SceneTextureLookup(int32 UV, uint32 InSceneTextureId, bool bFiltered) override
{
    const bool bSupportedOnMobile = InSceneTextureId == PPI_PostProcessInput0 ||
                                    InSceneTextureId == PPI_CustomDepth ||
                                    InSceneTextureId == PPI_SceneDepth;

    if (!bSupportedOnMobile && ErrorUnlessFeatureLevelSupported(ERHIFeatureLevel::SM4) == INDEX_NONE)
    {
        return INDEX_NONE;
    }

    if (FeatureLevel >= ERHIFeatureLevel::SM4)
    {
        return AddCodeChunk(
            MCT_Float4,
            TEXT("SceneTextureLookup(%s, %d, %s)"),
            *TexCoordCode, (int)SceneTextureId, bFiltered ? TEXT("true") : TEXT("false")
            );
    }
    else
    {
        FString TexCoordCode = CoerceParameter(UV, MCT_Float2);
        return AddCodeChunk(MCT_Float4,
            TEXT("MobileSceneTextureLookup(Parameters, %d, %s)"),
            (int32)SceneTextureId, *TexCoordCode);
    }
}
```

4. Using CustomDepth in Post Process Material

- C++ generates HLSL code for the PP material
- The Scene Texture material node generates HLSL code that calls the function `MobileSceneTextureLookup()`
- We sample the appropriate texture for the `SceneTextureId` parameter passed in

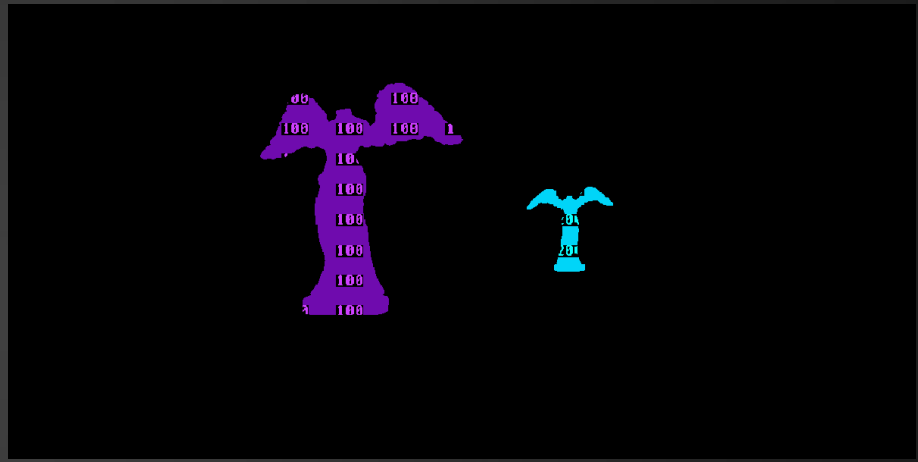
Engine\Shaders\MaterialTemplate.usf

```
MaterialFloat4 MobileSceneTextureLookup(inout FMaterialPixelParameters Parameters, int SceneTextureId, float2 UV)
{
#if (FEATURE_LEVEL <= FEATURE_LEVEL_ES3_1)
    //PPI_SceneDepth
    if (SceneTextureId == 1)
    {
        MaterialFloat Depth = ConvertFromDeviceZ(Texture2DSample(SceneDepthTexture, SceneDepthTextureSampler, UV).r);
        return MaterialFloat4(Depth.rrr, 0.f);
    }
    //PPI_CustomDepth
    else if (SceneTextureId == 13)
    {
        MaterialFloat Depth = ConvertFromDeviceZ(Texture2DSample(CustomDepthTexture, CustomDepthTextureSampler, UV).r);
        return MaterialFloat4(Depth.rrr, 0.f);
    }
    // PPI_PostprocessInput
    else if (SceneTextureId == 14)
    {
#if POST_PROCESS_MATERIAL
        MaterialFloat4 Input0 = Texture2DSample(PostprocessInput0, PostprocessInput0Sampler, UV);
#if POST_PROCESS_MATERIAL_BEFORE_TONEMAP
        Input0 = Decode32BPPHDR(Input0);
#endif
        // We need to preserve original SceneColor Alpha as it's used by tonemaper on mobile
        Parameters.BackupSceneColorAlpha = Input0.a;
        return Input0;
#endif// POST_PROCESS_MATERIAL
    }
#endif// FEATURE_LEVEL

    return MaterialFloat4(0.0f, 0.0f, 0.0f, 0.0f);
}
```

Possible Enhancement: Custom Stencil

- Desktop render can render extra information during CustomDepth pass
- Each object can output a specific byte value (0-255) into stencil buffer – CustomStencil
- Then use CustomStencil values in post-process



Possible Enhancement: Custom Stencil

- OpenGL ES2 does not support sampling stencil as a texture
- Possible with GL_EXT_texture_buffer extension
- Instead of sampling stencil we can do multiple post-process render passes using CustomStencil as a mask (stencil operations)

Possible Enhancement: CustomDepth in Base Pass

- Currently sampling custom depth is only allowed for translucent materials and post-processing

Q & A

- Thanks!