O3DE

Working together
for an open 3D world

/* ACADEMY SOFTWARE FOUNDATION

open Source days '24

Understanding And Customizing The O3DE Render Pipeline

About Me

Galib Arrieta

- Active contributor to O3DE since 2018 (back then called Lumberyard).
- Discord: galibzon
- Github: @galibzon or @lumbermixalot

About O3DE

https://github.com/o3de/o3de.git
https://o3de.org/
https://discord.com/invite/o3de

Real-Time, High Fidelity PBR Pipeline For Games & Video

Although this presentation is focused on empowering developers, unfamiliar with O3DE, to get started with the Graphics APIs, it is important to remark that O3DE provides, out of the box, a Real-Time, AAA quality, PBR Material System. Besides games, the Main Render Pipeline is ready to support Computer Vision simulation for Robotics and Video Production. And, of course, it's all open source under MIT + Apache 2.0 Licensing.

Rendered with O3DE.

Platforms Supported By O3DE

https://www.docs.o3de.org/docs/welcome-guide/supported-platforms/

For Development:

- Linux: Vulkan.
- Windows: Vulkan, DX12, **OpenXR**(via Vulkan).
- MacOS is experimentally supported, and mostly used to build iOS applications made with O3DE.

For Runtime:

- Linux: Vulkan.
- Windows: Vulkan, DX12, **OpenXR**(via Vulkan).
- Android: Vulkan, OpenXR(via Vulkan, tested on Meta Quest devices.)
- MacOS (Metal)
- iOS (Metal)
- Regarding Console support, the engine has the infrastructure to allow anyone to extend it for console support.

Agenda

Agenda

1. Target audience.
2. Companion github repository for this presentation.
3. Big picture of the O3DE graphics architecture.
4. Hello World Triangle. (No C++).
5. Hello World Mesh. (No C++).
6. Adding custom Pass Templates. (C++).
7. When is C++ needed?
8. Creating a Feature Processor (C++).
9. Injecting Passes Into The Main Render Pipeline (C++).
10. Conclusions and what's next.

Target Audience

Welcome to O3DE

Target Audience.

Welcome to O3DE.

This presentation assumes you have at least basic knowledge of 3D Graphics, including familiarity with at least one of these APIs: Vulkan, DX12, Metal, OpenGL, etc.

Some experience working with game engines architected around the concepts of Entities/Actors and Components.

In particular, this presentation is about how to customize the O3DE Render Pipeline from the ground up. You'll know enough to even replace the PBR Material System.

Companion github repository for this presentation.

https://github.com/galibzon/siggraph2024

Companion Github Repository

https://github.com/galibzon/siggraph2024

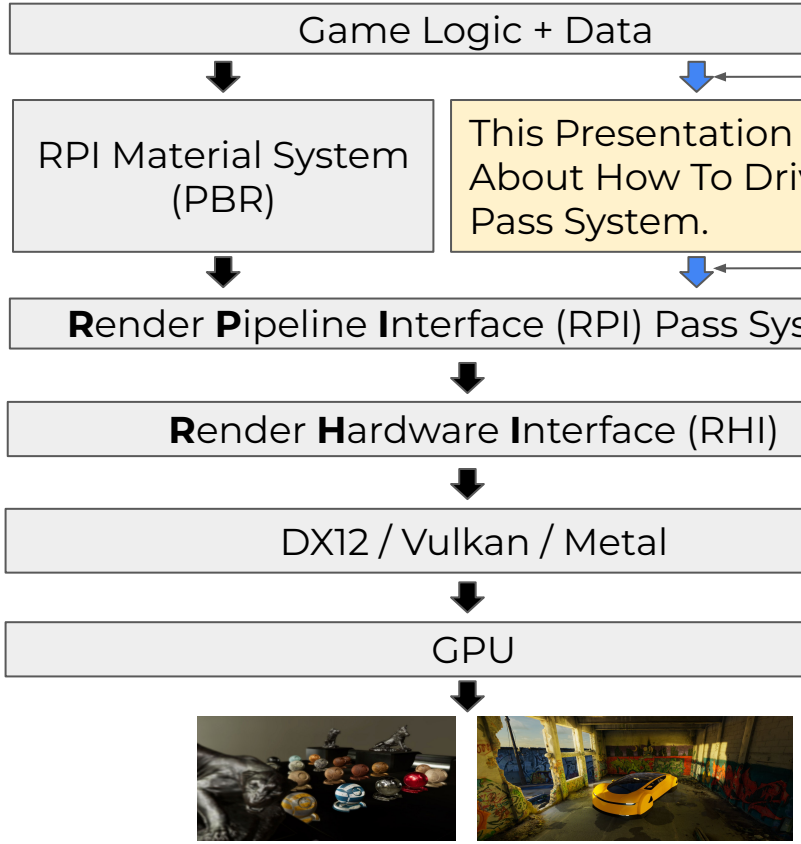
Contains two folders:

- **Gem/**. Contains code and assets. As a typical O3DE Gem, it can be referenced and used by other projects.
- **Project/**. Contains the sample levels.

Big picture of the O3DE graphics architecture

O3DE -> RPI -> RHI -> GPU.

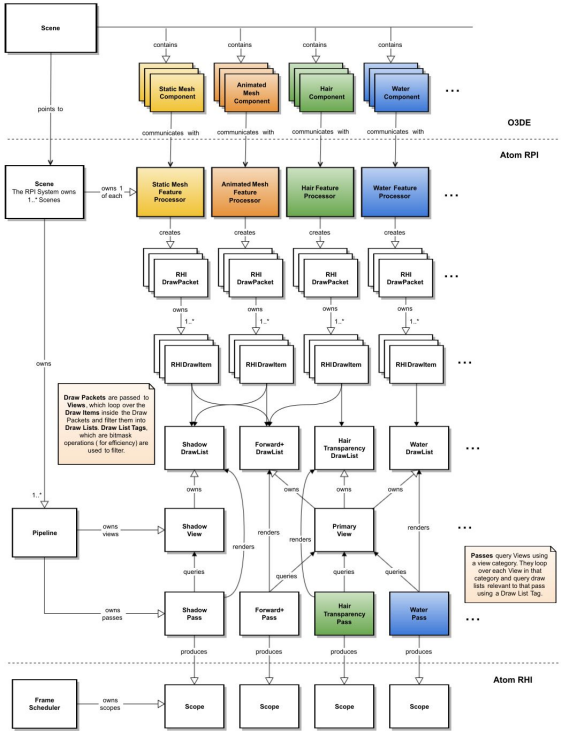
The End Game



The end game, pun intended, is to learn how to drive the PassSystem (docs).

You'll know enough about co-existing with the MaterialSystem (docs) or even replacing it altogether.

The Big Picture



https://www.docs.o3de.org/docs/atom-guide/dev-guide/frame-rendering/

O3DE: Entities and their Components push **data** into FeatureProcessors.

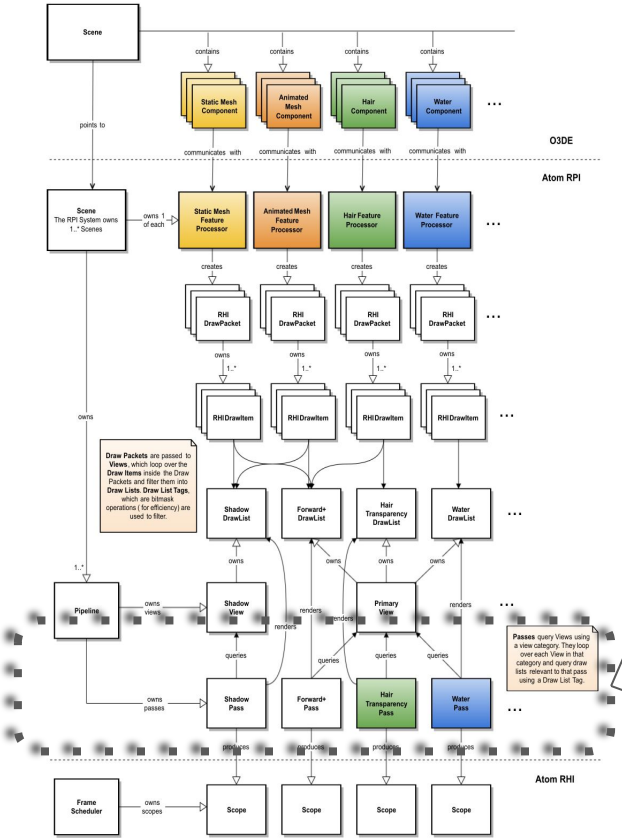
RPI: Feature Processors process the **data** received from Components and push DrawPackets into Views. Typically, Draw Packets are pushed into a View only if the View **contains** the DrawListTag from the Shaders that the Feature Processor cares about.

When a RenderPipeline is ticked, each of its Passes will:

- Pull DrawListViews from the View owned by the Render Pipeline. The View only returns Draw Lists that match a particular DrawListTag. The Draw Lists are **cached**.
- Insert themselves as Scopes into the FrameGraphBuilder (which is the same FrameScheduler) .

RHI: The Frame Scheduler invokes APIs on each Scope (Pass) to setup attachment dependencies, bind the Pass ShaderResourceGroup (if needed) and build CommandLists from Draw Lists that were **already cached**.

Render Pipelines Can Be Instantiated From Assets Or C++.



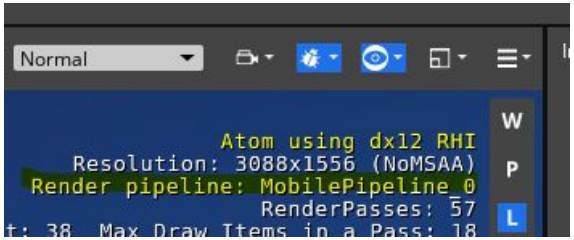
Asset

Or

C++

A Render Pipeline can be defined in an asset with extension "**.azasset**".
Upon startup, O3DE will assume the active Render Pipeline to be defined by the r_renderPipelinePath CVAR, with default value "passes/MainRenderPipeline.azasset".
For example, to replace the active Render Pipeline with the Mobile RenderPipeline you can pass the following command line argument to the Editor:

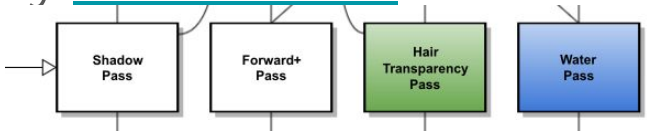
```
--r_renderPipelinePath=Passes/Mobile/MobileRenderPipeline.azasset
```



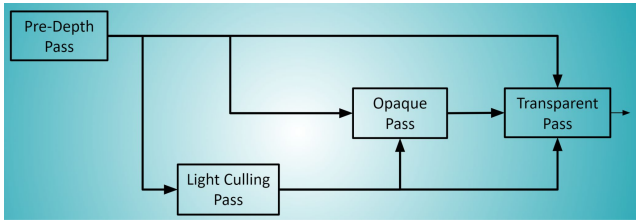
A Render Pipeline can be defined and instantiated at runtime with 100% C++ (or hybrid C++ & assets). There are **no performance advantages** in doing so.

Overview Of Render Pipeline Descriptor Assets.

A Render Pipeline Descriptor **“.azasset”** (JSON) file, defines a Directed Acyclic Graph (DAG) of PassRequests where the dependency, and order of execution, between Passes is defined by Attachments.



In this picture the Passes appear as running in parallel, in reality they are connected and ordered as a DAG.



A Render Pipeline asset can only reference C++ Pass classes registered with PassFactory**::AddPassCreator(...)**.
See PassFactory::AddCorePasses() or Common System Component::Activate() for examples.

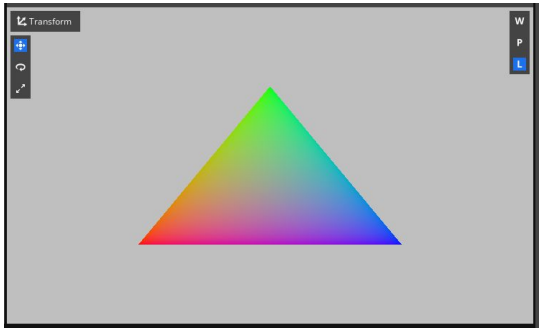
Hello World Triangle (No C++).

A Custom Render Pipeline With One Triangle.

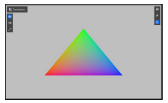
Hello World Triangle

Let’s create a single pass Render Pipeline, which will display a triangle. In the end, the user should be able to configure:

1. The background color (aka Clear Color).
2. Position and Color for each of the three vertices in the triangle.
3. We will replace the Main Render Pipeline used by the Editor, with our One Triangle Render Pipeline.



Summary Of Steps To Create The One Triangle Render Pipeline



1. Create the One Triangle Shader assets.
   1.1. **OneTriangle.azsl** (The actual shader code).
   1.2. **OneTriangle.shader** (Defines the shader entry points, blend states, compilation options, etc).
2. Create the Pass asset: **OneTriangle.pass**.
3. Create the Pipeline (aka Parent Pass) asset: **OneTrianglePipeline.pass**.
4. Create the RenderPipelineDescriptor asset: **OneTriangleRenderPipeline.azasset**
5. Create the **PassTemplates.azasset** asset.
6. Configure the **r_renderPipelinePath** CVAR as a command line argument so the Editor displays the One Triangle Render Pipeline.
   `--r_renderPipelinePath=Passes/Siggraph2024Gem/OneTriangleRenderPipeline.azasset`

OneTriangle.azsl - Shader Code

```
#include <Atom/Features/ColorManagement/TransformColor.azsli>
#include <scenesrg.srgi>
ShaderResourceGroup PassSrg : SRG_PerPass {
    float4x4 m_clipSpacePositions;
    float4x4 m_vertexColors;
}

struct VSInput {
    uint m_vertexID : SV_VertexID;
};
struct VSOutput {
    float4 m_position : SV_Position;
    float4 m_color : COLOR;
};
VSOutput MainVS(VSInput input) {
    VSOutput OUT;
    OUT.m_position = PassSrg::m_clipSpacePositions[input.m_vertexID];
    OUT.m_color = PassSrg::m_vertexColors[input.m_vertexID];
    return OUT;
}

struct PSOutput {
    float4 m_color : SV_Target0;
};
PSOutput MainPS(VSOutput input) {
    PSOutput OUT;
    OUT.m_color.xyz = TransformColor(input.m_color.xyz, ColorSpaceId::LinearSRGB, ColorSpaceId::SRGB);
    return OUT;
}
```

In O3DE, shader code is written in AZSL, a superset of HLSL. We'll go over details of the language later.

In terms of vertex data, all we get is the **SV_VertexID** semantics. It's a three vertices Draw Call invoked by FullscreenTrianglePass.

As part of this exercise, position and color for each vertex will come from **float4x4** shader constants (Which are reflected to the Pass System and defined in the Pass asset for further flexibility).

Vertex and Pixel shader functions (entry points) are trivial.

OneTriangle.shader

```
{
    "Source" : "OneTriangle.azsl",

    "DepthStencilState" : {
        "Depth" : { "Enable" : false, "CompareFunc" : "GreaterEqual" }
    },

    "DrawList" : "forward", // Irrelevant for this shader.

    "ProgramSettings":
    {
      "EntryPoints":
      [
        {
          "name": "MainVS",
          "type": "Vertex"
        },
        {
          "name": "MainPS",
          "type": "Fragment"
        }
      ]
    }
}
```

In O3DE, a ShaderSourceData (***.shader**) asset defines all the pieces required to compile a shader. AZSL source code, Depth/Stencil state, Raster state, Blend State, Entry point functions (VS, PS, etc) and compilation options.

At a bare minimum this asset says to disable Depth testing, the Vertex Shader entry point is called **MainVS,** and the Pixel Shader entry point is called **MainPS.**

Shader files should be placed at:

/GIT/siggraph2024/Gem/Assets/Shaders/Siggraph2024Gem/OneTriangle.azsl

/GIT/siggraph2024/Gem/Assets/Shaders/Siggraph2024Gem/OneTriangle.shader

OneTriangle.pass

```
{
    "Type": "JsonSerialization",
    "Version": 1, "ClassName": "PassAsset",
    "ClassData": {
        "PassTemplate": {
            "Name": "OneTrianglePassTemplate",
            "PassClass": "FullScreenTriangle",
            "Slots": [
                {
                    "Name": "ColorOutput", "SlotType": "Output",
                    "ScopeAttachmentUsage": "RenderTarget",
                    "LoadStoreAction": {
                        "ClearValue": { "Value": [ 0.75, 0.75, 0.75, 0.0 ] },
                        "LoadAction": "Clear"
                    }
                }
            ],
            "PassData": {
                "$type": "FullscreenTrianglePassData",
                "ShaderAsset": {
                    "FilePath": "Shaders/Siggraph2024Gem/OneTriangle.shader"
                },
                "BindViewSrg": true, // We don't use the ViewSrg, but this avoids a crash.
                "ShaderDataMappings": {
                    "Matrix4x4Mappings": [
                        {
                            "Name": "m_clipSpacePositions",
                            "Value": [ 0.0,  0.5, 0.0, 1.0, // X, Y, Z, W [Vertex 0]
                                      -0.5, -0.5, 0.0, 1.0, // X, Y, Z, W [Vertex 1]
                                       0.5, -0.5, 0.0, 1.0, // X, Y, Z, W [Vertex 2]
                                       0.0,  0.0, 0.0, 0.0 ] //Never used by the shader
                        },
                        {
                            "Name": "m_vertexColors",
                            "Value": [ 0.0, 1.0, 0.0, 1.0, // R, G, B, A [Vertex 0]
                                       1.0, 0.0, 0.0, 1.0, // R, G, B, A [Vertex 1]
                                       0.0, 0.0, 1.0, 1.0, // R, G, B, A [Vertex 2]
                                       0.0, 0.0, 0.0, 0.0 ] //Never used by the shader
                        }
                    ]
                }
            }
        }
    }
}
```

A PassAsset (*.pass)(doc) file describes a Pass Template. A Pass Template specifies the concrete C++ Pass class (FullscreenTriangle) that should be instantiated for a particular Pass, along with all the required attachments needed by **any Shader** that runs under this Pass.

For this simple case, we only need a Render Target. The Render Target will be cleared to a light Gray color when the Pass is started in the GPU.

The **“PassData”** property defines the **concrete Shader** that the Pass should execute, along with the values for the Shader Constants.

This file should be placed at:
/GIT/siggraph2024/Gem/Assets/Passes/Siggraph2024Gem/OneTriangle.pass

OneTrianglePipeline.pass

```
{
    "Type": "JsonSerialization",
    "Version": 1,
    "ClassName": "PassAsset",
    "ClassData": {
        "PassTemplate": {
            "Name": "OneTrianglePipelineTemplate",
            "PassClass": "ParentPass",
            "Slots": [
                {
                    // The Slot name must be exactly "PipelineOutput" because this the Parent Pass that
                    // Describes a Render Pipeline and the C++ code looks for a PassSlotBinding
                    // with this name, which will be connected to the SwapChain.
                    "Name": "PipelineOutput",
                    "SlotType": "InputOutput"
                }
            ],
            "PassRequests": [
                {
                    "Name": "OneTrianglePass",
                    "TemplateName": "OneTrianglePassTemplate",
                    "Enabled": true,
                    "Connections": [
                        {
                            "LocalSlot": "ColorOutput",
                            "AttachmentRef": {
                                "Pass": "Parent",
                                "Attachment": "PipelineOutput"
                            }
                        }
                    ]
                }
            ]
        }
    }
}
```

A Pipeline is just another “PassAsset”, in which the Pass class is “ParentPass”.

A Parent Pass Template lists all the Passes it needs (as **“PassRequests”**) and how they are interconnected by attachment dependencies (**“Connections”**).

In this particular case, only one child Pass is requested. The Child Pass Template is **“OneTrianglePassTemplate”**. The **“ColorOutput”** Slot Attachment of **“OneTrianglePass”** is connected to the **“PipelineOutput”** of this Parent Pass, which in turns is directly connected to the Swapchain thanks to the PassSystem.

This file should be placed at:
`/GIT/siggraph2024/Gem/Assets/Passes/Siggraph2024Gem/OneTrianglePipeline.pass`

OneTriangleRenderPipeline.azasset

```
{
    "Type": "JsonSerialization",
    "Version": 1,
    "ClassName": "RenderPipelineDescriptor",
    "ClassData": {
        "Name": "OneTrianglePipeline",
        "RootPassTemplate": "OneTrianglePipelineTemplate"
    }
}
```

A Render Pipeline Descriptor (*.azasset) asset file describes which Pass Template to load in order to build a Render Pipeline.

The “**RootPassTemplate**” property defines the name of the Pass Template with “**PassClass**”: “**ParentPass**”. This Parent Pass will become the **Root Pass** of the Pipeline.

This file should be placed at:
/GIT/siggraph2024/Gem/Assets/Passes/Siggraph2024Gem/OneTriangleRenderPipeline.azasset

PassTemplates.azasset

```
{
    "Type": "JsonSerialization",
    "Version": 1,
    "ClassName": "AssetAliasesSourceData",
    "ClassData": {
        "AssetPaths": [
            ...
            When not writing C++ code, you must list the same
            pass templates as in /GIT/o3de/Gems/Atom/Feature/Common/Assets/Passes/PassTemplates.azasset
            ...
            // List of custom pass templates from the Siggraph 2024 Gem.
            {
                "Name": "OneTrianglePassTemplate",
                "Path": "Passes/Siggraph2024Gem/OneTriangle.pass"
            },
            {
                "Name": "OneTrianglePipelineTemplate",
                "Path": "Passes/Siggraph2024Gem/OneTrianglePipeline.pass"
            }
        ]
    }
}
```

Any Pass Template referenced by other Pass Templates or a Render Pipeline Descriptor, must be registered with the Pass System.

By default, O3DE loads Pass Templates from an asset located at (Scan Folder Relative): "Passes/PassTemplates.azasset".

By default the Atom/Feature/Common Gem provides such asset with an extensive list of Pass Templates required by the Material System, Editor Gizmos, UI Fonts, etc.

This file should be placed at: `/GIT/siggraph2024/Project/Passes/PassTemplates.azasset`.

There is a C++ API to **add** Pass Templates to the Pass System. But in this "Hello World" example we are avoiding C++ at all costs. So the next best thing is to create a copy of `/GIT/O3DE/Gems/Atom/Feature/Common/Assets/Passes/PassTemplates.azasset` as `/GIT/siggraph2024/Project/Passes/PassTemplates.azasset` and append the new Pass Templates introduced in this presentation: "**OneTrianglePassTemplate**" and "**OneTrianglePipelineTemplate**".

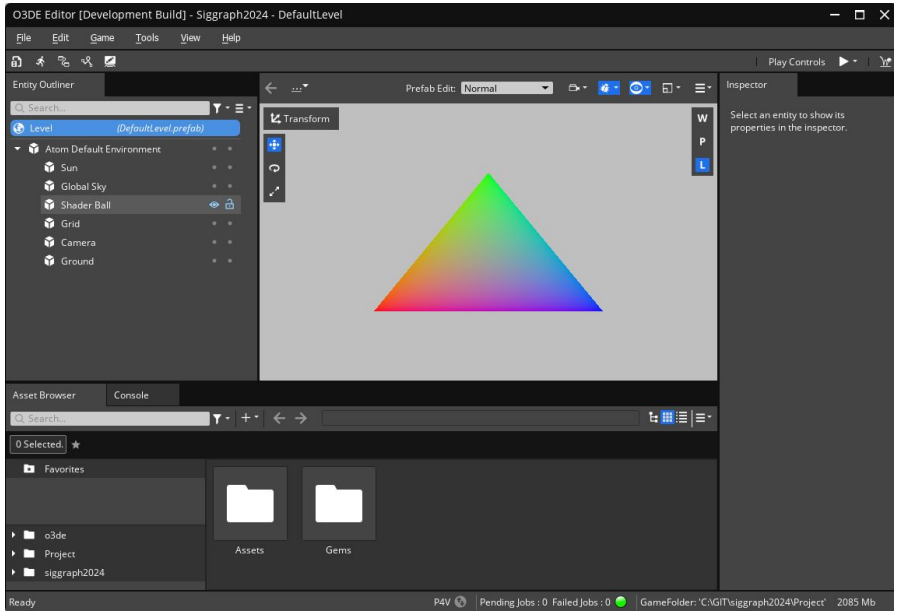
Unlike the other assets mentioned before, this one is placed under the "**GIT/Sigraph2024/Project**" folder instead of the "**GIT/Sigraph2024/Gem**" folder because the Project always has higher asset scan folder priority when asset paths collide. This can be changed but would only complicate this Presentation.

Running The Editor And Rendering The Triangle

Execute the Editor with the following command line argument:

`--r_renderPipelinePath=Passes/Siggraph2024Gem/OneTriangleRenderPipeline.azasset`

When prompted, pick any level.



We have replaced the Main Render Pipeline with a one pass pipeline that renders one triangle.

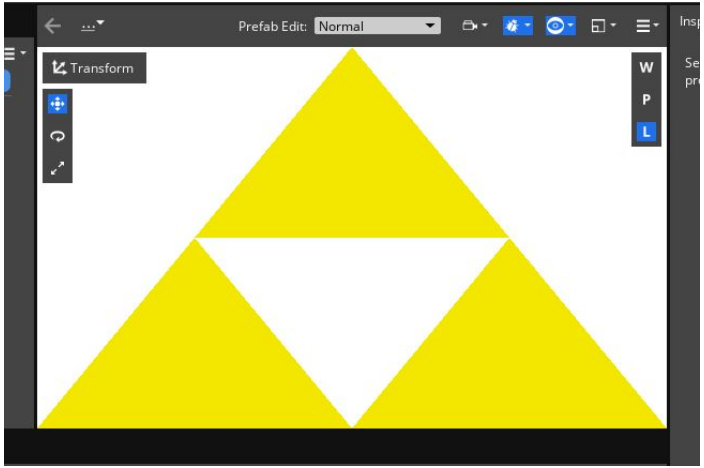
Regardless of which Entities you select, add or delete, the viewport won't change. There's no FPS counter, no gizmos, and no meshes because those are rendered by the Main Render Pipeline.

Mission accomplished! You have a data driven render pipeline with custom background color, and configurable triangle vertices and colors.

Feel free to modify the **"ClearValue"** or the **"ShaderDataMappings"** in **OneTriangle.pass** and you should see the changes updated on screen as soon as the Pass asset is reloaded (**No need to restart the Editor**).

Adding More Triangles

What about one more triangle? Simply add another “OneTrianglePassTemplate”, but the second pass should not clear the Render Target.
What about three triangles? Here is the TriforceRenderPipeline:
`--r_renderPipelinePath=Passes/Siggraph2024Gem/TriforceRenderPipeline.azasset`



Lessons Learned From One Triangle

1. O3DE is flexible enough to ingest Render Pipelines and Render Passes in the form of assets. This can be done in C++ too.

2. It only takes one command line argument to replace the Main Render Pipeline with your own.

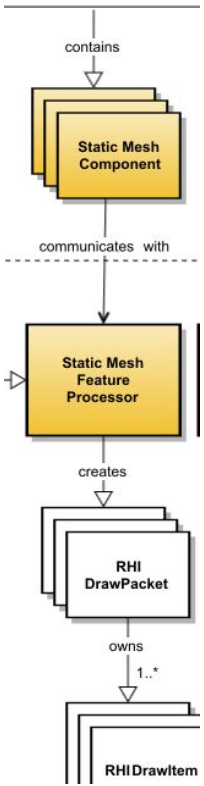
3. The Root/Parent Pass of a Render Pipeline can reference the Swapchain as an “InputOutput” attachment by the hard-coded name “**PipelineOutput**”.

4. Any concrete Pass based on **FullscreenTrianglePass** is eligible to three vertices, identifiable in the Vertex Shader by the **SV_VertexID** semantic.

DrawListTag Deep Dive.

Sending a message in a bottle to multiple public addresses.

DrawListTag Deep Dive



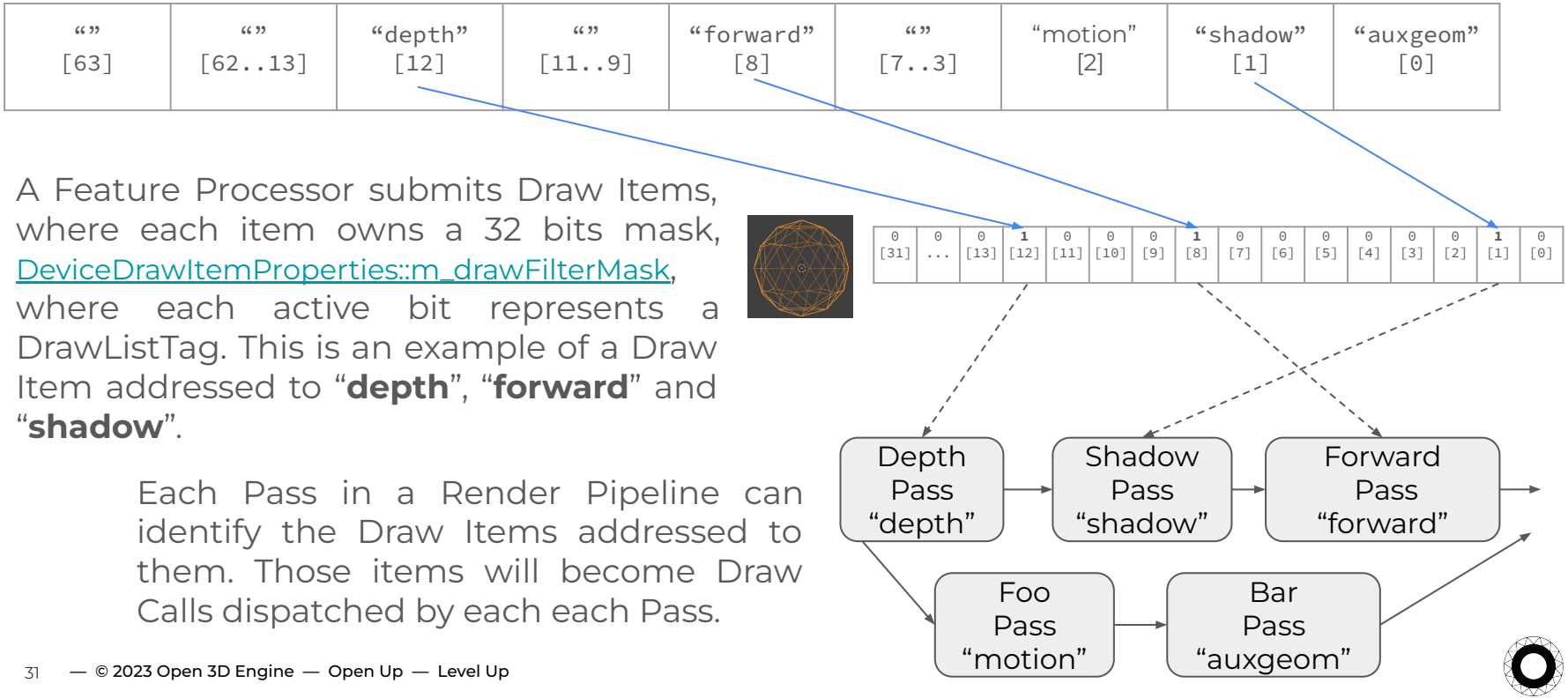
In order to render something more complex than a triangle, it's important to know how the higher layers of the engine submit Draw Packets to an arbitrary set of Passes.

At its core, a DrawListTag is an address, that serves as a layer of indirection that decouples Feature Processors from Passes. To a developer, a DrawListTag is identifiable by a string like "forward", "depth", "auxgeom", "motion", "shadow", etc. At **runtime**, a DrawListTag is an **index** related to one of those strings. There can be up to 64 (AZ::RHI::Pipeline::DrawListTagCountMax) unique DrawListTags.

Feature Processors submit DrawPackets to Views. Each Draw Item inside a Draw Packet contains a mask that groups all the DrawListTags of interest. In turn, Passes are **expected** (they can break the rule if they want to) to read only the Draw Items that match the DrawListTag they care about.

DrawListTag Deep Dive

At runtime, the RPI assigns an index to each DrawListTag string.



A Feature Processor submits Draw Items, where each item owns a 32 bits mask, DeviceDrawItemProperties::m_drawFilterMask, where each active bit represents a DrawListTag. This is an example of a Draw Item addressed to “**depth**”, “**forward**” and “**shadow**”.

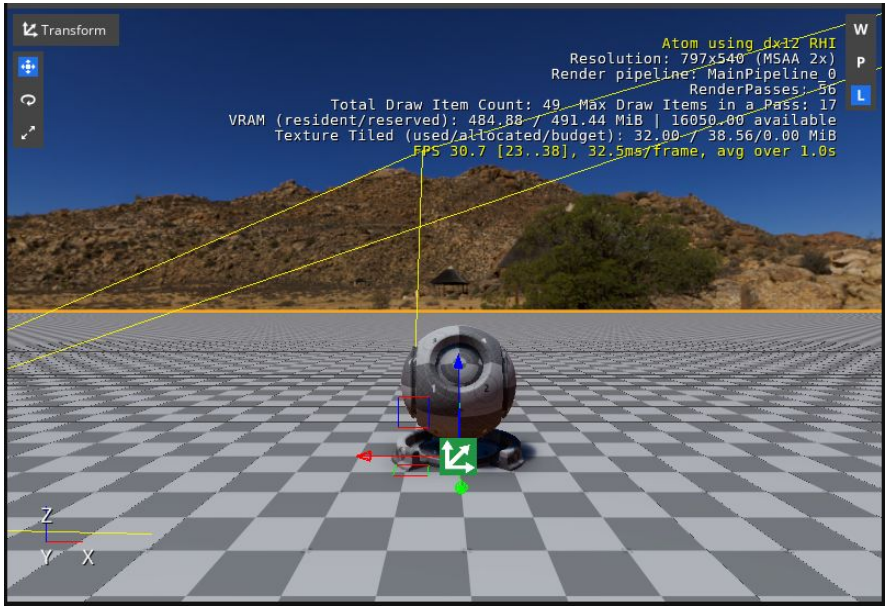
Each Pass in a Render Pipeline can identify the Draw Items addressed to them. Those items will become Draw Calls dispatched by each each Pass.

Hello World Mesh Rendering (No C++).

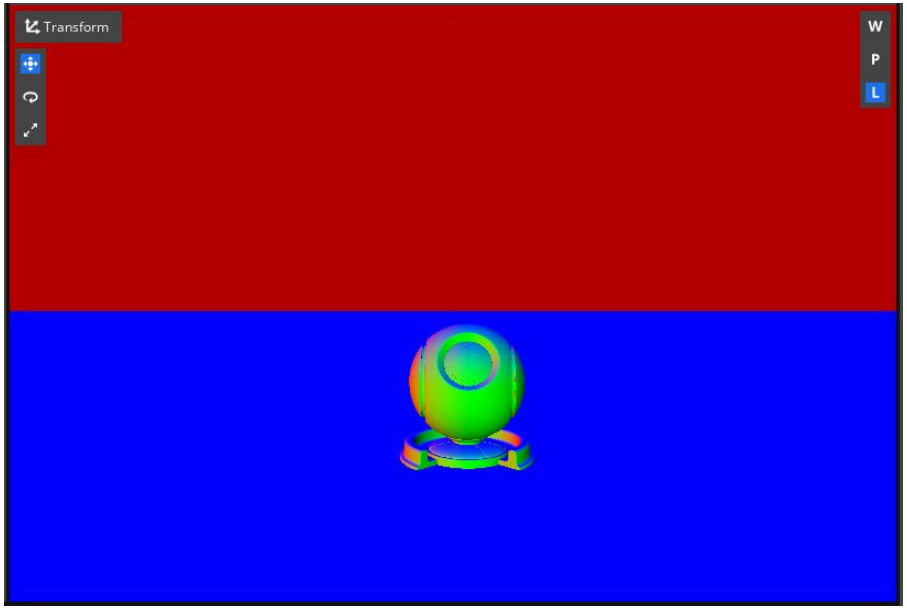
A custom Render Pipeline that piggybacks on the Mesh Feature Processor and the Material System.

Hello World Mesh Rendering (No C++).

Let's figure out how to put together a custom Render Pipeline that renders mesh assets from Mesh Components. The mesh Normals will be used as Fragment Colors.



No command line arguments, which uses MainRenderPipeline.azasset.

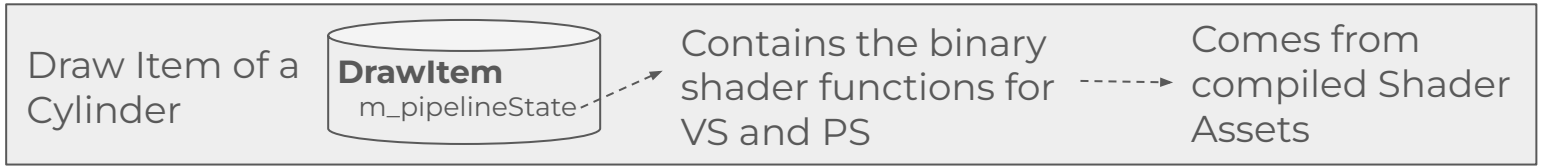


--r_renderPipelinePath=Passes/Siggraph2024Gem/SiggraphRenderPipeline.azasset

Is It Possible To Render Meshes Without Writing C++?

Yes, indeed. Let's figure out how to put together a custom Render Pipeline that renders mesh assets from Mesh Components.

Something We glossed over when talking about Feature Processors submitting Draw Items, is that a Draw Item is created with a Pipeline State Object, aka PSO. This means the Feature Processor must know what Shader Asset to use when building a Draw Item.



In particular, the Mesh Feature Processor, works in tandem with the Material System. The Mesh Component (doc) works with the Material Component (doc) to link a Model Asset with a Material Asset, which in turn describes a list of Shaders that should be used when rendering a Mesh.

We are going to create the most basic Material that uses our custom Shader. The custom Render Pipeline will have a Raster Pass that **will match the DrawListTag** of the custom Shader.

Creating A Custom Material

This is not a presentation about the O3DE PBR Material System, instead we will piggyback on the Material System as minimally as possible to be able to render entities with (Mesh + Material) Components.

In summary:

1. Create the Shader. With “DrawList”: “siggraph”.
2. Create the Material Type.
3. Create the Material.
4. Create the Pass asset. This will use the RasterPass with “DrawListTag”: “siggraph”.
5. Create the Pass asset for the single-pass pipeline.
6. Create the Render Pipeline Descriptor asset.
7. In the **Editor**, set the new Material asset to the **ShaderBall** and **Ground** entities.

NormalVectorDisplay.azsl

```
#include <scenesrg.srgi>
#include <viewsrg.srgi>
#include <Atom/Features/PBR/DefaultObjectSrg.azsli>

struct VSInput {
    float3 m_position : POSITION;
    float3 m_normal : NORMAL;
};
struct VSOutput {
    float4 m_position : SV_Position;
    float3 m_normal: NORMAL;
};
VSOutput MainVS(VSInput IN) {
    VSOutput OUT;
    float3 worldPosition = mul(ObjectSrg::GetWorldMatrix(), float4(IN.m_position, 1.0)).xyz;
    OUT.m_position = mul(ViewSrg::m_viewProjectionMatrix, float4(worldPosition, 1.0));
    OUT.m_normal = IN.m_normal;
    return OUT;
}

struct PSOutput {
    float4 m_color : SV_Target0;
};
PSOutput MainPS(VSOutput IN) {
    PSOutput OUT;
    OUT.m_color = float4(normalize(IN.m_normal), 1.0);
    return OUT;
}
```

Located at: /GIT/siggraph2024/Gem/Assets/Shaders/Siggraph2024Gem/Materials/NormalVectorDisplay.azsl

Minimalist shader that renders a mesh, using vertex Normals as fragment Colors.

The Material System automatically sets the shader constants related with the Scene, View, and Object.

The shader constant for the **Object World Matrix**, is available in the **ObjectSrg**. The shader constant for the View-Projection Matrix is in **ViewSrg::m_viewProjectionMatrix.**

NormalVectorDisplay.shader

```
{
    "Source" : "NormalVectorDisplay.azsl",

    "DepthStencilState" : {
        "Depth" : { "Enable" : true, "CompareFunc" : "GreaterEqual" }
    },

    "DrawList" : "siggraph",

    "ProgramSettings": {
      "EntryPoints":
      [
        {
          "name": "MainVS",
          "type": "Vertex"
        },
        {
          "name": "MainPS",
          "type": "Fragment"
        }
      ]
    }
}
```

This time we are enabling depth testing so we can render all Entities in the correct order of depth.

Most importantly, we are setting **"DrawList" : "siggraph"**, which should match the **“DrawListTag”** of our custom RasterPass template (**NormalVectorDisplay.pass**).

Located at: /GIT/siggraph2024/Gem/Assets/Shaders/Siggraph2024Gem/Materials/NormalVectorDisplay.shader

NormalVectorDisplay.materialtype

```
{
    "description": "For Siggraph2024. This material renders vertex normals as fragment colors",
    "version": 1,
    "shaders": [
        {
            "file": "Shaders/Siggraph2024Gem/Materials/NormalVectorDisplay.shader"
        }
    ]
}
```

Typically a Material Type lists all the Material Properties. The shader constants and resources related with Material Properties are contained in the **Material ShaderResourceGroup**, also known as **MaterialSrg** in Shader code.

In our case, We have no Material Properties (We will do that later).

A Material Type owns a list of Shaders. In this example we are using only on Shader, which will become part of the Draw Items Pipeline State Object for all Mesh Entities in the scene that use our custom Material.

Located at: /GIT/siggraph2024/Gem/Assets/Materials/Siggraph2024Gem/NormalVectorDisplay.materialtype

NormalVectorDisplay.material

```
{
    "materialType": "NormalVectorDisplay.materialtype",
    "materialTypeVersion": 1
}
```

A Material asset represents an instance of a Material Type. The **concrete** value of each Material Property is listed here.

In our case, We have no Material Properties.

This minimalist Material asset is simply saying that it is an instance of "**NormalVectorDisplay.materialtype**".

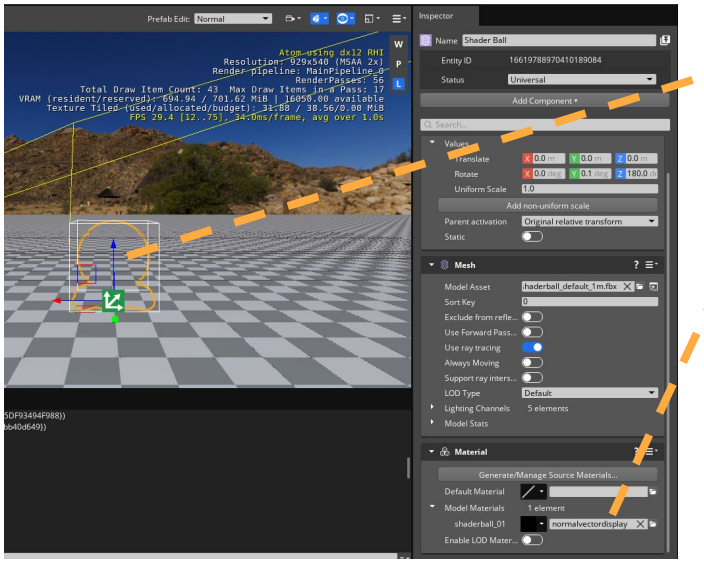
This is the asset we will pick later when adding the Material Component to Entities that have a Mesh Component.

Located at: /GIT/siggraph2024/Gem/Assets/Materials/Siggraph2024Gem/NormalVectorDisplay.material

Some Thoughts Before Writing The New Render Pipeline

At this moment we have a new Material asset called **NormalVectorDisplay.material.** We can use it on Entities with Mesh Components.

But, when using the Main Render Pipeline, the meshes that get linked to **NormalVectorDisplay.material** will become invisible because there's no RasterPass in the Main Render Pipeline looking for Draw Items with DrawListTag "**siggraph**".



The mesh of the "**Shader Ball**" Entity disappears as soon as the Material Component is set to "**NormalVectorDisplay.material**"

NormalVectorDisplay.pass

```
{
    "Type": "JsonSerialization", "Version": 1, "ClassName": "PassAsset",
    "ClassData": {
        "PassTemplate": {
            "Name": "NormalVectorDisplayPassTemplate", "PassClass": "RasterPass",
            "Slots": [
                { "Name": "DepthOutput",
                  "SlotType": "Output",
                  "ScopeAttachmentUsage": "DepthStencil",
                  "LoadStoreAction": {
                        "ClearValue": { "Type": "DepthStencil" },
                        "LoadAction": "Clear",
                        "LoadActionStencil": "Clear"
                    }
                },
                { "Name": "LightingOutput",
                  "SlotType": "Output",
                  "ScopeAttachmentUsage": "RenderTarget",
                  "LoadStoreAction": {
                        "ClearValue": { "Value": [ 0.7, 0.0, 0.0, 0.0 ] },
                        "LoadAction": "Clear"
                    }
                }
            ], "ImageAttachments": [
                {
                    "Name": "DepthAttachment",
                    "SizeSource": { "Source": { "Pass": "Parent", "Attachment": "PipelineOutput" } },
                    "ImageDescriptor": { "Format": "D32_FLOAT_S8X24_UINT", "SharedQueueMask": "Graphics" }
                }
            ], "Connections": [
                {
                    "LocalSlot": "DepthOutput",
                    "AttachmentRef": { "Pass": "This", "Attachment": "DepthAttachment" }
                }
            ], "PassData": {
                "$type": "RasterPassData",
                "DrawListTag": "siggraph",
                "BindViewSrg": true
            }
        }
    }
}
```

Located at: /GIT/siggraph2024/Gem/Assets/Passes/Siggraph2024Gem/NormalVectorDisplay.pass

The Pass Class is "RasterPass", with **"DrawListTag": "siggraph".** All Draw Items with this tag will be rendered under this Pass.

This Pass will ask the Pass System to create a transient GPU Image Attachment with name "**DepthAttachment**" that will serve as the **DepthStencil** attachment for Depth Testing.

When the Pass starts it will clear the Render Target with a dark red (0.7, 0.0, 0.0, 0.0).

In terms of Color attachment, this Pass Template will need the Parent Pass to connect the "**LightingOutput**" Slot with the correct Color attachment (Which later the Parent Pass template, **NormalVectorDisplayPipeline.pass**, will bind directly to the Swapchain).

NormalVectorDisplayPipeline.pass

```
{
    "Type": "JsonSerialization",
    "Version": 1,
    "ClassName": "PassAsset",
    "ClassData": {
        "PassTemplate": {
            "Name": "NormalVectorDisplayPipelineTemplate",
            "PassClass": "ParentPass",
            "Slots": [
                {
                    "Name": "PipelineOutput",
                    "SlotType": "InputOutput"
                }
            ],
            "PassRequests": [
                {
                    "Name": "NormalVectorDisplayPass",
                    "TemplateName": "NormalVectorDisplayPassTemplate",
                    "Enabled": true,
                    "Connections": [
                        {
                            "LocalSlot": "LightingOutput",
                            "AttachmentRef": {
                                "Pass": "Parent",
                                "Attachment": "PipelineOutput"
                            }
                        }
                    ]
                }
            ]
        }
    }
}
```

Our Single-Pass-Pipeline Pass template.

Explicitly says that the “**LightingOutput**” Slot in "**NormalVectorDisplayPass**" should be directly bound to the Swapchain.

Located at: /GIT/siggraph2024/Gem/Assets/Passes/Siggraph2024Gem/NormalVectorDisplayPipeline.pass

NormalVectorDisplayRenderPipeline.azasset

```
{
    "Type": "JsonSerialization",
    "Version": 1,
    "ClassName": "RenderPipelineDescriptor",
    "ClassData": {
        "Name": "NormalVectorDisplayPipeline",
        "MainViewTag": "MainCamera",
        "RootPassTemplate": "NormalVectorDisplayPipelineTemplate",
        "AllowModification": false,
        "RenderSettings": {
            "MultisampleState": {
                "samples": 1
            }
        }
    }
}
```

A regular Render Pipeline Descriptor asset, which references "**NormalVectorDisplayPipelineTemplate**" As the Root Pass Template.

Located at: /GIT/siggraph2024/Gem/Assets/Passes/Siggraph2024Gem/NormalVectorDisplayRenderPipeline.azasset

Extending PassTemplates.azasset

We are adding two more Pass Templates to the Pass System:
"**NormalVectorDisplayPassTemplate**" and
"**NormalVectorDisplayPipelineTemplate**".

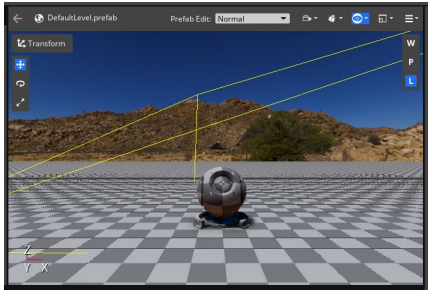
```
{
    "Type": "JsonSerialization",
    "Version": 1,
    "ClassName": "AssetAliasesSourceData",
    "ClassData": {
        "AssetPaths": [
            ...
            When not writing C++ code, you must list the same
            pass templates as in /GIT/o3de/Gems/Atom/Feature/Common/Assets/Passes/PassTemplates.azasset
            ...
            // List of custom pass templates from the Siggraph 2024 Gem.
            {
                "Name": "OneTrianglePassTemplate",
                "Path": "Passes/Siggraph2024Gem/OneTriangle.pass"
            },
            {
                "Name": "OneTrianglePipelineTemplate",
                "Path": "Passes/Siggraph2024Gem/OneTrianglePipeline.pass"
            },
            {
                "Name": "OneTriangleNoClearPassTemplate",
                "Path": "Passes/Siggraph2024Gem/OneTriangleNoClear.pass"
            },
            {
                "Name": "TriforcePipelineTemplate",
                "Path": "Passes/Siggraph2024Gem/TriforcePipeline.pass"
            },
            {
                "Name": "NormalVectorDisplayPassTemplate",
                "Path": "Passes/Siggraph2024Gem/NormalVectorDisplay.pass"
            },
            {
                "Name": "NormalVectorDisplayPipelineTemplate",
                "Path": "Passes/Siggraph2024Gem/NormalVectorDisplayPipeline.pass"
            }

        ]
    }
}
```

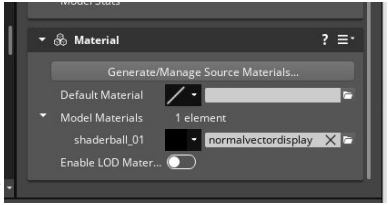
Located at: `/GIT/siggraph2024/Project/Passes/PassTemplates.azasset`.

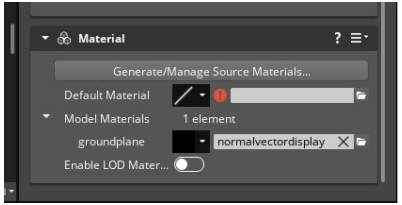
It’s Showtime. (Part 1)

Run the Editor (no command line arguments).
Open the level named “**DefaultLevel**”.

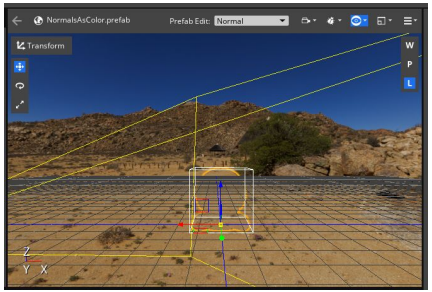


Save the level as “**NormalsAsColor**”. Add the Material Component to Entities “**Shader Ball**” and “**Ground**” and pick the “**NormalVectorDisplay.material**” asset for both of them.

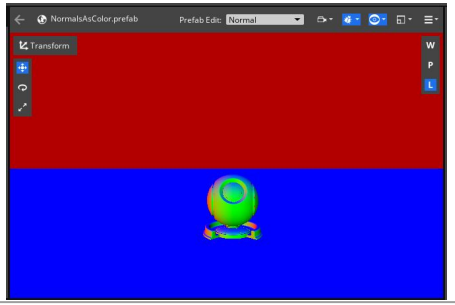




As expected, both Meshes disappear because there’s not a RasterPass looking for Draw Items with **DrawListTag** == **“siggraph”** .



Restart the Editor with argument:
--r_renderPipelinePath=Passes/Siggraph2024Gem/NormalVectorDisplayRenderPipeline.azasset
And load the new level named “**NormalsAsColor**”.



Adding Custom Pass Templates (C++).

The proper way of adding more Pass Templates to the Pass System.

Adding Custom Pass Templates.

There's a better way to register custom Pass Templates other than making a copy of the O3DE **PassTemplates.azasset**. The right way requires a little bit of C++.

Any Gem (or Project) can instantiate an EventHandler of type: RPI::PassSystemInterface::OnReadyLoadTemplatesEvent::Handler, which should be registered with the Pass System using the API: RPI::PassSystemInterface::ConnectEvent(handler).

This is a one time registration, which is typically done during a System Component Activate(). Examples:

1. Gems/Atom/Feature/Common/Code/Source/CommonSystemComponent.cpp.
2. Gems/Terrain/Code/Source/Components/TerrainSystemComponent.cpp.

When the time is right, the Pass System will invoke the callbacks of all handlers connected to the event RPI::PassSystemInterface::OnReadyLoadTemplatesEvent. During this callback, a custom System Component may invoke the API: AZ::RPI::PassSystemInterface::LoadPassTemplateMappings(String& assetTemplatePath).

Siggraph2024Gem Loads Custom Pass Templates

```
        ...
        // Load pass template mappings for this Gem.
        void LoadPassTemplateMappings();

        // We use this event handler to add our Pass Templates to the RPI::PassSystem.
        // The callback will invoke this->LoadPassTemplateMappings()
        AZ::RPI::PassSystemInterface::OnReadyLoadTemplatesEvent::Handler m_loadTemplatesHandler;

        ...
```

/GIT/siggraph2024/Gem/Code/Source/Clients/Siggraph2024GemSystemComponent.h

```
    ...
    void Siggraph2024GemSystemComponent::Activate() {
        ...

        // Register the event handler that helps us register our custom pass templates.
        m_loadTemplatesHandler = AZ::RPI::PassSystemInterface::OnReadyLoadTemplatesEvent::Handler(
            [&]() {
                LoadPassTemplateMappings();
            }
        );
        AZ::RPI::PassSystemInterface::Get()->ConnectEvent(m_loadTemplatesHandler);
    }

    void Siggraph2024GemSystemComponent::LoadPassTemplateMappings() {
        constexpr char passTemplatesFile[] = "Passes/Siggraph2024Gem/PassTemplates.azasset";
        AZ::RPI::PassSystemInterface::Get()->LoadPassTemplateMappings(passTemplatesFile);
    }
    ...
```

/GIT/siggraph2024/Gem/Code/Source/Clients/Siggraph2024GemSystemComponent.cpp

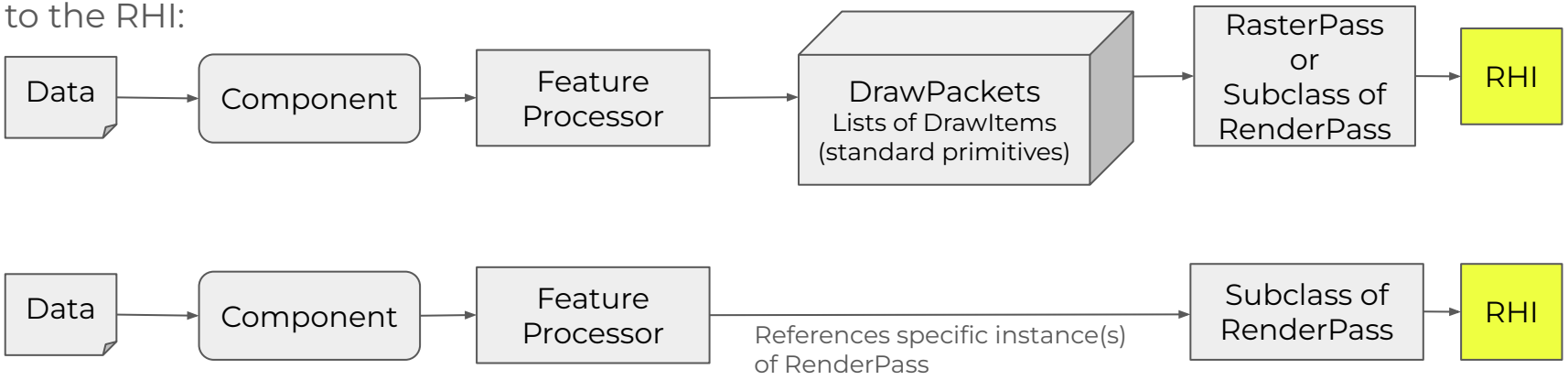
When Is C++ Needed?

When do you need custom Components, Feature Processors, or Passes?

When Is C++ Needed?

There are times when writing C++ is inevitable. Let's review some of those situations.

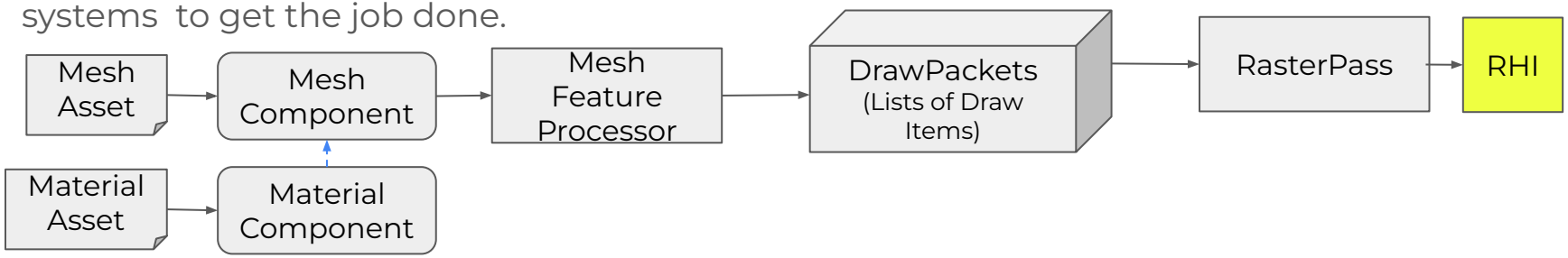
Conceptually O3DE and the RPI assume the following overarching data flows to submit work to the RHI:



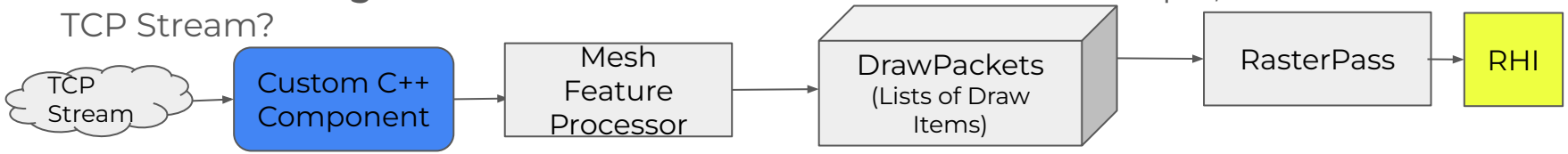
If a Render Pipeline can not get the job done with some combination of the Core/Generic Passes provided by O3DE, then custom C++ Pass class must be added, and perhaps new Feature Processors and their respective user facing Components. See "void PassFactory::AddCorePasses()".

Leveraging The Mesh Feature Processor

The Mesh and Material Systems are flexible enough that sometimes you don't need to write C++ (as demonstrated previously). And sometimes you can leverage some parts of those systems to get the job done.



What if the **triangle** data doesn't come from an Asset. For example, the data comes from a TCP Stream?



You can leverage the Mesh Feature Processor, but a custom C++ Component is needed.

Similarly, whenever the source data is dynamic in nature, like the WhiteBox Component (doc), where the data is made of standard primitives like triangles, a custom C++ Component is needed, and it can leverage the Mesh Feature Processor.

Other Common Cases That Require C++

- Some Shader Constants for which their values may change from frame to frame are managed by the Pass System (e.g. SceneSrg::m_time). If a Pipeline requires other kind of dynamic data for Shader Constants then C++ is needed.

- When not using the Material System, and needing to bind Texture resources to Shaders, then custom C++ Passes (and maybe custom Feature Processors are need).

- Non-standard primitive data, like Volumetrics always require custom Feature Processors or custom Passes or a combination of both.

Creating A Feature Processor (C++)

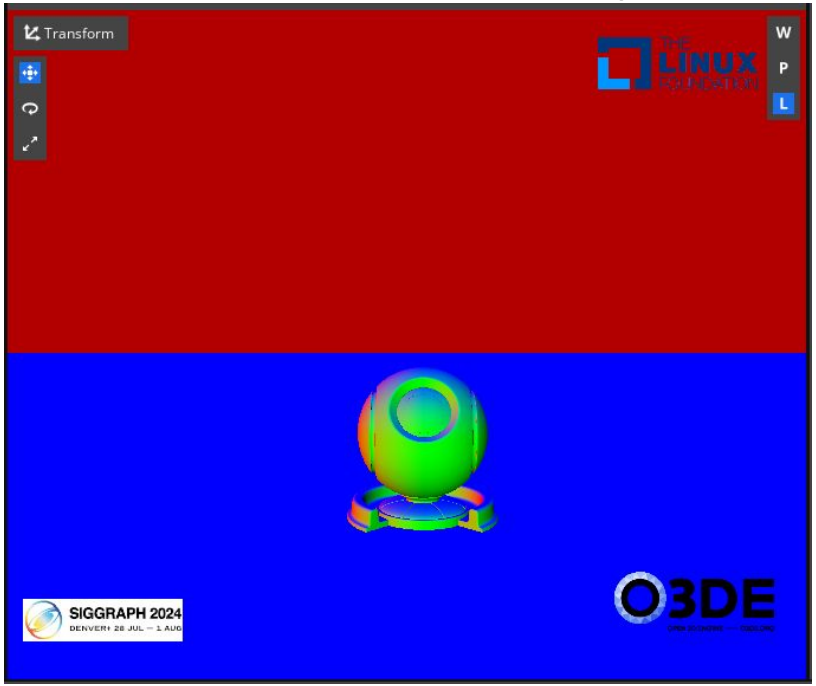
Everybody loves stickers, right?

A New Feature Processor (C++)

Everybody loves stickers, right?

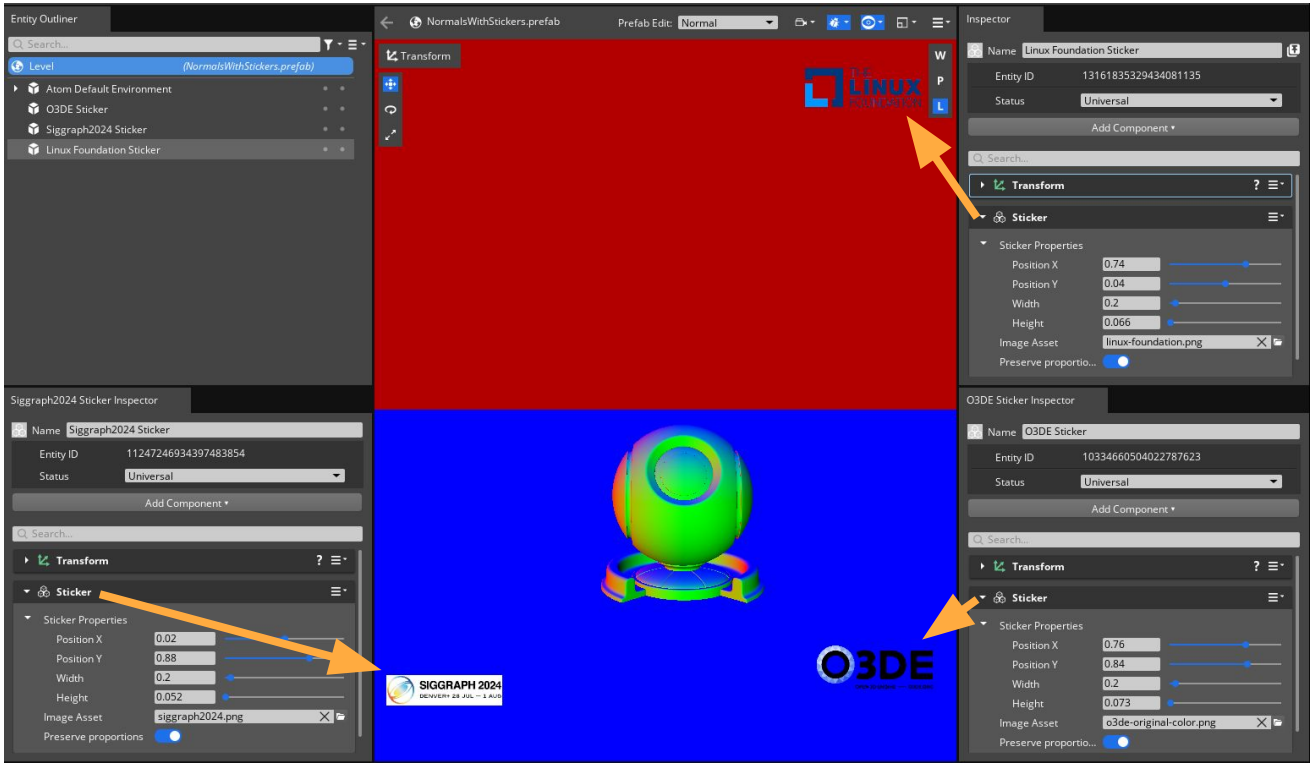
Generated with https://meta.ai. Prompt:
"Picture of a laptop with stickers"

Let's create a Feature Processor that renders stickers on the Viewport



A Feature Processor That Renders Stickers

In this level there are three entities with their own “**Sticker**” Component.

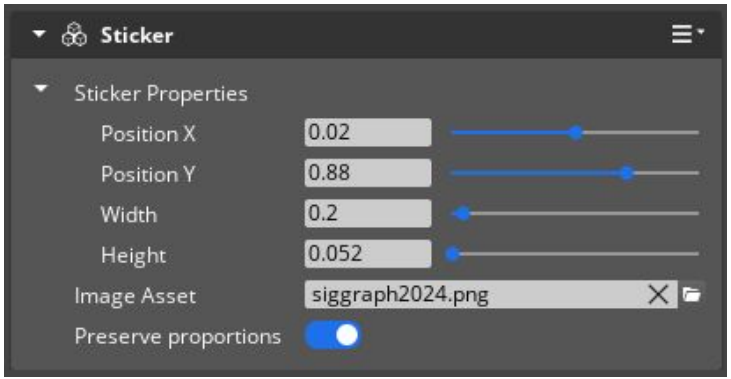


The Sticker Feature Processor.

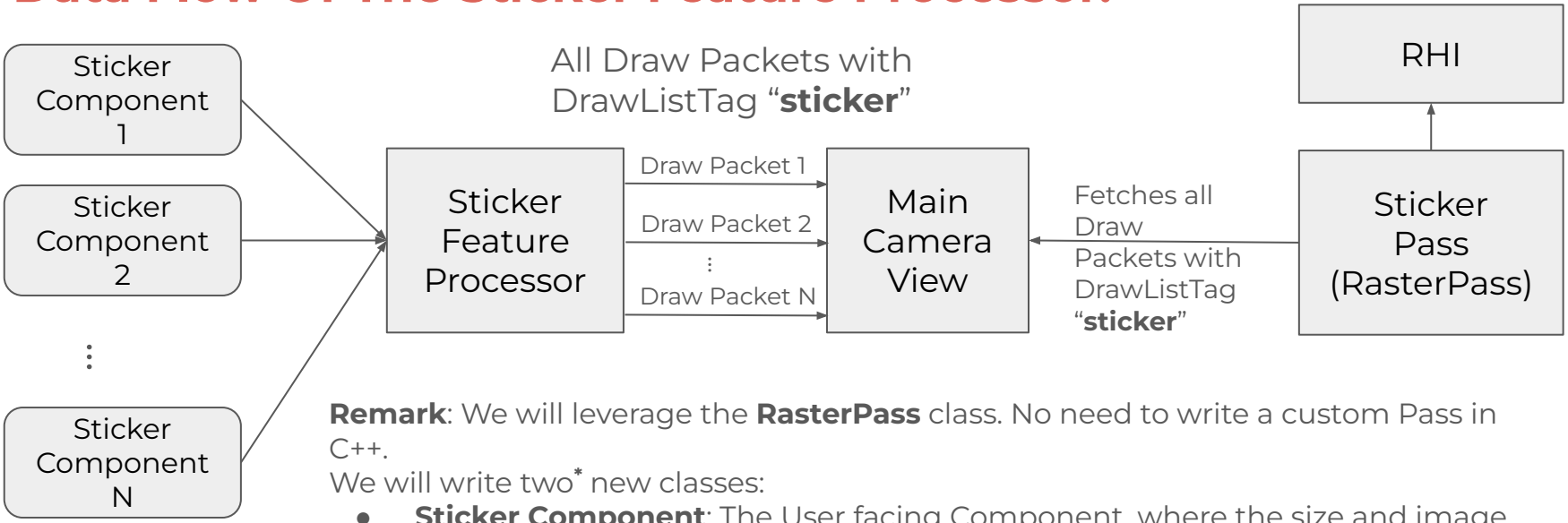
The purpose of this Feature Processor is to render an **arbitrary amount** of Images (aka Stickers), overlayed on the viewport.

The images can have Alpha color.

The user can specify sticker location and size, using **normalized coordinates** (between 0.0 and 1.0) and also preserve the original proportion of the image if needed.



Data Flow Of The Sticker Feature Processor.



Remark: We will leverage the **RasterPass** class. No need to write a custom Pass in C++.
We will write two* new classes:

- **Sticker Component**: The User facing Component, where the size and image asset of the Sticker can be configured.
- **The Sticker Feature Processor**: Gets data from each Sticker Component and submits Draw Packets with the "**sticker**" DrawListTag to the main View of the Render Pipeline.

* Other helper classes will be added.

Sticker.pass Is Just A Vanilla Raster Pass

```
{
    "Type": "JsonSerialization",
    "Version": 1,
    "ClassName": "PassAsset",
    "ClassData": {
        "PassTemplate": {
            "Name": "StickerPassTemplate",
            "PassClass": "RasterPass",
            "Slots": [
                {
                    "Name": "ColorOutput",
                    "SlotType": "Output",
                    "ScopeAttachmentUsage": "RenderTarget",
                    "LoadStoreAction": {
                        "LoadAction": "Load"
                    }
                }
            ],
            "PassData": {
                "$type": "RasterPassData",
                "DrawListTag": "sticker",
                "BindViewSrg": true
            }
        }
    }
}
```

The “**StickerPassTemplate**” is just a vanilla “**RasterPass**” will render Draw Items with the “**sticker**” DrawListTag.

The content of the Render Target **won’t be cleared** when this Pass starts. The Draw Items (Stickers) will be drawn on top of the pixel content.

Located at: /GIT/siggraph2024/Gem/Assets/Passes/Siggraph2024Gem/Sticker.pass

StickerPipeline.pass Render Pipeline

```
{
    . . .
        "PassTemplate": {
            "Name": "StickerPipelineTemplate",
            "PassClass": "ParentPass",
             . . .
            "PassRequests": [
                {
                    "Name": "NormalVectorDisplayPass",
                    "TemplateName": "NormalVectorDisplayPassTemplate",
                    . . .
                },
                {
                    "Name": "StickerPass",
                    "TemplateName": "StickerPassTemplate",
                    "Enabled": true,
                    "Connections": [
                        {
                            "LocalSlot": "ColorOutput",
                            "AttachmentRef": {
                                "Pass": "Parent",
                                "Attachment": "PipelineOutput"
                            }
                        }
                    ]
                }
            ]
        }
    }
}
```

This asset is based on "**NormalVectorDisplayPipeline.pass**", it differs from the original because of the addition of a second Pass called "**StickerPass**".

"**StickerPass**" is based on the Template "**StickerPassTemplate**" (shown in the previous slide).

Now the Pipeline contains two passes.

The "**StickerPass**" will render the stickers directly into the Swapchain.

Located at: /GIT/siggraph2024/Gem/Assets/Passes/Siggraph2024Gem/StickerPipeline.pass

StickerRenderPipeline.azsset Render Pipeline Descriptor

```
{
    "Type": "JsonSerialization",
    "Version": 1,
    "ClassName": "RenderPipelineDescriptor",
    "ClassData": {
        "Name": "StickerPipeline",
        "MainViewTag": "MainCamera",
        "RootPassTemplate": "StickerPipelineTemplate",
        "AllowModification": false
    }
}
```

Just a standard Render Pipeline Descriptor asset, which references our new Pipeline from template “**StickerPipelineTemplate**”.

Located at: /GIT/siggraph2024/Gem/Assets/Passes/Siggraph2024Gem/StickerRenderPipeline.azasset

Registering The New Pass Templates

```
{
    "Type": "JsonSerialization",
    "Version": 1,
    "ClassName": "AssetAliasesSourceData",
    "ClassData": {
        "AssetPaths": [
            . . .
            {
                "Name": "NormalVectorDisplayPipelineTemplate",
                "Path": "Passes/Siggraph2024Gem/NormalVectorDisplayPipeline.pass"
            },
            {
                "Name": "StickerPassTemplate",
                "Path": "Passes/Siggraph2024Gem/Sticker.pass"
            },
            {
                "Name": "StickerPipelineTemplate",
                "Path": "Passes/Siggraph2024Gem/StickerPipeline.pass"
            }
        ]
    }
}
```

The new pass templates must be registered to avoid crashes whenever “**StickerRenderPipeline.azasset**” is made as the active/default Render Pipeline.

Located at: /GIT/siggraph2024/Gem/Assets/Passes/Siggraph2024Gem/PassTemplates.azasset

Remark: If the Editor is started with the command line argument:
--r_renderPipelinePath=Passes/Siggraph2024Gem/StickerRenderPipeline.azasset everything should work just fine.

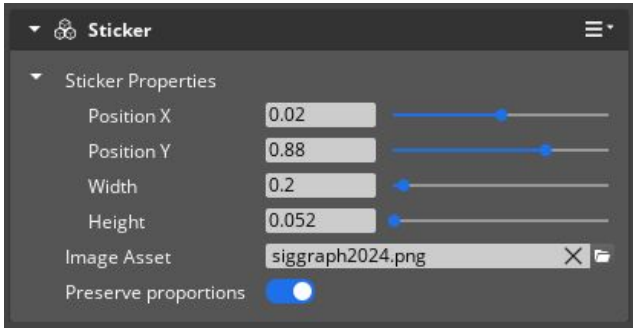
There will be a second Pass at the end of the Pipeline **doing nothing** called “**StickerPass**”.

The reason nothing new will happen is because there’s no Feature Processor (yet) submitting Draw Packets with the “**sticker**” DrawListTag.

StickerComponentController

This class is the liaison between the **StickerComponent** and the **StickerFeatureProcessor**.

When this class is Activated it calls StickerFeatureProcessor::**AddSticker**() to instantiate a DrawPacket that contains all the data required to render a Sticker on screen.



Each time the User makes changes to the properties of this Component (Position, Size or Texture changes) this class calls StickerFeatureProcessor::**UpdateStickerGeometry**() Or ::**UpdateStickerTexture**().

When the User deletes this Component, this class is Deactivated() and it will call StickerFeatureProcessor**::RemoveSticker**().

Located at:
/GIT/siggraph2024/Gem/Code/Source/Components/StickerComponentController.h
/GIT/siggraph2024/Gem/Code/Source/Components/StickerComponentController.cpp

StickerFeatureProcessor (struct StickerRenderData)

This is the class responsible for submitting Draw Packets. One Draw Packet is created per Sticker.

Copy of the original data from the Sticker Component (Also contains the Texture Image)

All the CPU and GPU render specific data and Buffers.

The Draw Packet, it references the GPU Buffer Views and the DrawSrg (Container of the Shader Constants and SRVs)

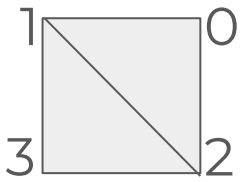
```
struct StickerFeatureProcessor::StickerRenderData
{
    StickerProperties m_properties;
    AZStd::array<StickerVertex, VertexCountPerSticker> m_cpuVertices;
    AZ::Data::Instance<AZ::RPI::Buffer> m_gpuVertexBuffer;
    AZStd::array<AZ::RHI::StreamBufferView, 1> m_gpuVertexBufferView;
    AZ::Data::Instance<AZ::RPI::ShaderResourceGroup> m_drawSrg;
    AZ::RHI::ConstPtr<AZ::RHI::DrawPacket> m_drawPacket;
    bool m_needsSrgUpdate = true;
};

AZStd::unordered_map<AZ::EntityId, StickerRenderData> m_stickers;
```

All the Stickers are conveniently organized in a Map, addressable by EntityId.

StickerFeatureProcessor (The Sticker As A Primitive)

In terms of drawing, a Sticker is a Triangle Strip with 4 vertices, representing one Quad made of two triangles. Using the right hand rule, the triangles should be facing out of the screen: (0, 1, 2), (1, 3, 2):




```
struct StickerVertex
{
    AZStd::array<float, 3> m_position;
    AZStd::array<float, 2> m_uv;
};
```

The layout and content of the primitive is defined in this function

```
void InitCpuVertices(AZStd::array<StickerVertex, 4>& cpuVertices, const StickerProperties& properties);
```

The Shader, Sticker.shader (Sticker.azsl), does not transform the vertices. So they are expected to be in Normalized Device Coordinates with W component at (1.0) and Z (0.0).

Located at:

/GIT/siggraph2024/Gem/Code/Source/Render/StickerFeatureProcessor.h
/GIT/siggraph2024/Gem/Code/Source/Render/StickerFeatureProcessor.cpp

Sticker.shader

```
{
    "Source" : "Sticker.azsl",
    "DepthStencilState" : {
        "Depth" : { "Enable" : false, "CompareFunc" : "GreaterEqual" }
    },
    "GlobalTargetBlendState" : {
      "Enable" : true,
      "BlendSource" : "One",
      "BlendAlphaSource" : "One",
      "BlendDest" : "AlphaSourceInverse",
      "BlendAlphaDest" : "AlphaSourceInverse",
      "BlendAlphaOp" : "Add"
    },
    "DrawList" : "sticker",
    "ProgramSettings":
    {
      "EntryPoints":
      [
        {
          "name": "MainVS",
          "type": "Vertex"
        },
        {
          "name": "MainPS",
          "type": "Fragment"
        }
      ]
    }
}
```

As expected, this shader does not enable Depth Testing.

The Blend State is setup to support “Add” Alpha Blending Operation.

The StickerFeatureProcessor will use the DrawListTag “**sticker**” from this Shader asset as the DrawListTag of the Draw Packets.

Located at: /GIT/siggraph2024/Gem/Assets/Shaders/Siggraph2024Gem/Sticker.shader

Sticker.azsl

```
#include <Atom/Features/SrgSemantics.azsli>
#include <Atom/Features/ColorManagement/TransformColor.azsli>
ShaderResourceGroup DrawSrg : SRG_PerDraw {
    Texture2D<float4> m_texture;
    Sampler LinearSampler;
}

struct VSInput {
    float3 m_position : POSITION; float2 m_uv : UV0;
};
struct VSOutput {
    float4 m_position : SV_Position; float2 m_uv : UV0;
};
VSOutput MainVS(VSInput input) {
    VSOutput OUT;
    OUT.m_position = float4(input.m_position, 1);
    OUT.m_uv = input.m_uv;
    return OUT;
}

struct PSOutput {
    float4 m_color : SV_Target0;
};
PSOutput MainPS(VSOutput input) {
    PSOutput OUT;
    float4 color = DrawSrg::m_texture.Sample(DrawSrg::LinearSampler, input.m_uv);
    const float3 srgbColor = TransformColor(color.xyz, ColorSpaceId::LinearSRGB, ColorSpaceId::SRGB);
    OUT.m_color = float4(srgbColor, color.a);
    return OUT;
}
```

This is a simple Vertex and Pixel Shader Functions Combo.

The Vertex Shader is just a passthrough for the data to be interpolated and sent to the Pixel Shader.

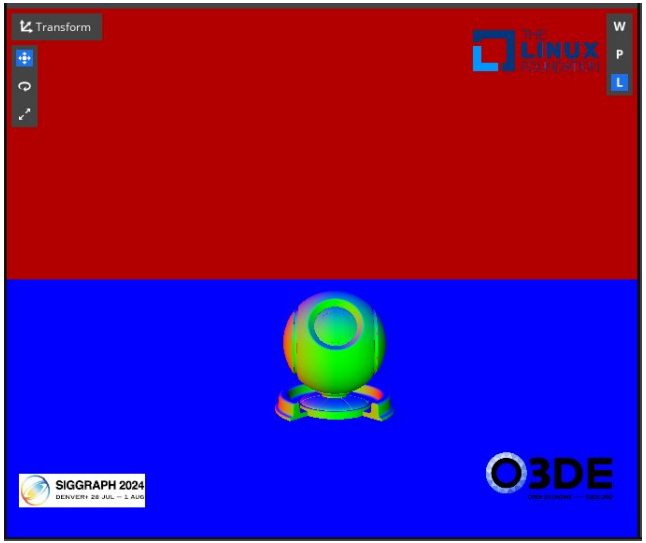
The Sticker image, **DrawSrg::m_texture,** is linearly sampled as the output color that will be Alpha Blended (Add Op) with the Render Target.

Located at: /GIT/siggraph2024/Gem/Assets/Shaders/Siggraph2024Gem/Sticker.azsl

Showtime

Run the Editor with argument:
--r_renderPipelinePath=Passes/Siggraph2024Gem/StickerRenderPipeline.azasset

Load the level named “**NormalsWithStickers**”.





Injecting Passes Into The Main Pipeline (C++)

How to modify another Render Pipeline at runtime.

How To Inject A Custom Pass Into The Main Pipeline

We have learned how to work with **our own** Render Pipelines.

But, is it possible to inject our own Passes into The Main Pipeline? **Without** manually copy/pasting the original **MainRenderPipeline.azasset**? Can We add the **Sticker** Pass to the Main Pipeline?

Yes, it is possible. Thanks to the FeatureProcessor function:

```
virtual void AddRenderPasses(RenderPipeline* pipeline) {}
```

This function works as a Callback. It is invoked only if **@pipeline** allows modifications.

When a Feature Processor overrides AddRenderPasses, it can search for other Passes using AZ::RPI::PassFilters and, based on the results, decide whether to add Passes to **@pipeline.**

StickerFeatureProcessor::AddRenderPasses(AZ::RPI::RenderPipeline* pipeline)

```
const auto StickerPassName = AZ::Name("StickerPass");
const auto UiPassName = AZ::Name("UIPass");
// Early exit if there's no "UIPass".
{
    AZ::RPI::PassFilter passFilter = AZ::RPI::PassFilter::CreateWithPassName(UiPassName, pipeline);
    AZ::RPI::Pass* existingPass = AZ::RPI::PassSystemInterface::Get()->FindFirstPass(passFilter);
    if (existingPass == nullptr)
    {
        // Can't find "UIPass"
        return;
    }
}

// Early exit if the Sticker Pass already exists
{
    ...
}

// We can safely add the StickerPass to the pipeline
static constexpr bool AddBefore = false;
AddPassRequestToRenderPipeline(pipeline,
   "Passes/Siggraph2024Gem/StickerPassRequest.azasset",
   UiPassName.GetCStr(), AddBefore);
// Optionally, validate we succeeded with another search.
{
   ...
}
```

This function is called by the PassSystem only if **@pipeline** allows modifications.

The StickerFeatureProcessor searches for a Pass named "**UIPass**". If the Pass doesn't exist then it can not inject the "**StickerPass**" into the Pipeline.

"StickerPass" only works well if it is the last Pass that writes to the Render Target. It should be added **after** "UIPass"**.** REMARK: As will be seen in the next slide, the proper attachment connection must be done between "UIPass" and "StickerPass" to enforce the correct order of execution: "**StickerPass must start AFTER UIPass**".

StickerPassRequest.azasset

```
{
    "Type": "JsonSerialization",
    "Version": 1,
    "ClassName": "PassRequest",
    "ClassData": {
        "Name": "StickerPass",
        "TemplateName": "StickerPassTemplate",
        "Enabled": true,
        "Connections": [
            {
                "LocalSlot": "ColorOutput",
                "AttachmentRef": {
                    "Pass": "UIPass",
                    "Attachment": "InputOutput"
                }
            }
        ]
    }
}
```

A PassRequest asset is only relevant when referenced by a call to **`AddPassRequestToRenderPipeline(...)`**

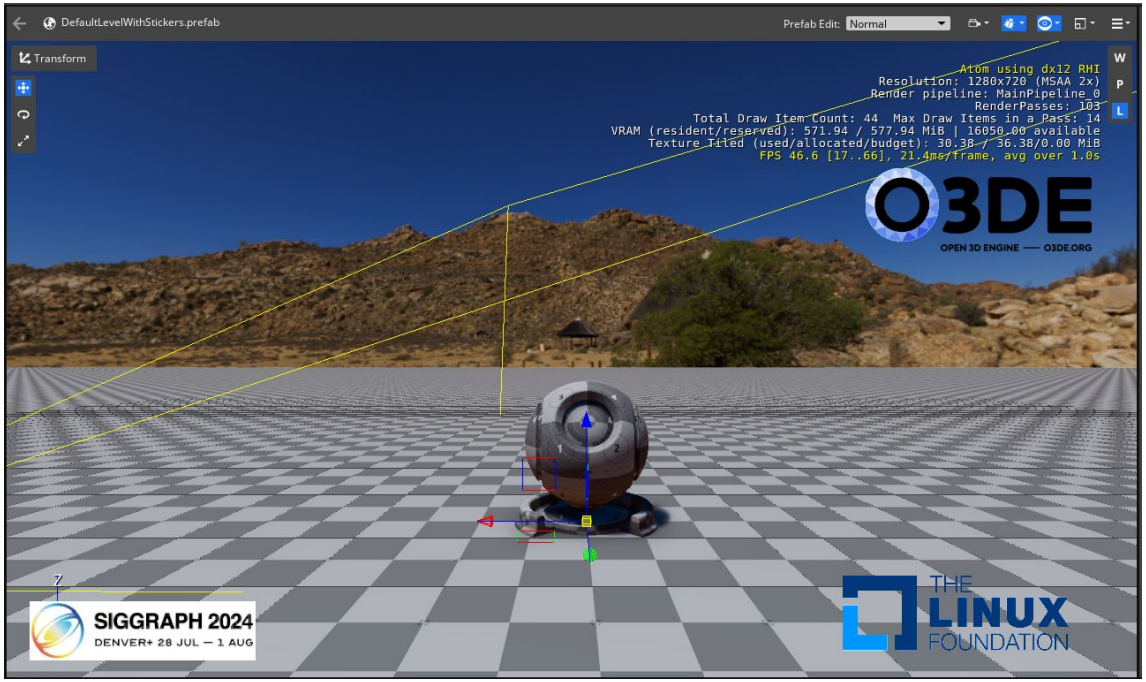
This asset looks like a slice of a Pipeline asset.

In particular, this asset assumes that “**UIPass**” exists, and that its Render Target slot is called “**`InputOutput`**”. This is why the C++ Runtime in **StickerFeatureProcessor** searches first if such pass named “**UIPass**” exists.

The request says that the “**`ColorOutput`**” Slot of “**StickerPass**” must be connected to the "**`InputOutput`**" Slot of the “**UIPass**” . This will guarantee the correct order of execution.

The Main Pipeline With Stickers.

We can now run the Editor **without** command line arguments. It will default to the Main Render Pipeline. But this time around, the **StickerFeatureProcessor** is smart enough to inject the **Sticker** Pass. Load the level named "**DefaultLevelWithStickers**".



Conclusions And What's Next.

The End Is Just A New Beginning.

Conclusions

In terms of what the O3DE RPI and RHI can do, We just skimmed the surface.

The O3DE Graphics APIs are data driven, yet fully customizable with C++ if required.

The O3DE Material System is not restricted to support only the Main Render Pipeline (PBR Based), instead it is an engine of its own that is ready for creating unique/proprietary Materials, Render Pipelines and Shaders.

The Main Render Pipeline is a Forward+ PBR pipeline, but should be clear by now that it is straightforward to build a Deferred PBR Pipeline that can co-exist with the Main Pipeline and allow the developer to switch back and forth.

What's Next?

- Fork O3DE and contribute: https://github.com/o3de/o3de
- Develop VR applications with the OpenXR Gem: https://github.com/o3de/o3de-extras
- Documentation available at: https://www.docs.o3de.org/docs/
- Join the O3DE Discord Server: https://discord.gg/nV5m4mJr
    - Join sig-graphics-audio for all things related with 3D Graphics in O3DE.

Rendered with O3DE.

Thank You!

Questions?