Scriptable Render Pipeline for Visual Perception Experiments

Yonatan Fihrer

214759047

EECS 4080 - Final Report

Introduction

The Scriptable Render Pipeline was released in 2018 as an API to be able to control the rendering pipeline of Unity, for a more flexible and customizable rendering experience. This API allows for many things that were either imposable or very hard and convoluted before as well as many fine control for a more tailored and optimized user experience. This also gave rise to Unity's two pre-built pipelines the Universal Rendering Pipeline and the High Definition Rendering Pipeline which have many advantages over the built-in render and also give the user the ability to easily hook in to, manipulate and organize the render pipeline.

Render Pipeline

Source Unity - Manual: Choosing and configuring a render pipeline and lighting solution

A render pipeline controls how the objects in the scene are displayed (not that actual rendering) and are done in 3 stages culling, rendering and post-processing. Culling is the process of determining the object that need to be rendered. For example those that are in view of the camera and unobstructed by other objects. Rendering is the process of drawing the culled objects, which takes in to account lighting, colouring those object and drawing shadows. Post-processing is a filter on the entire camera in which effects that are applied after the objects are rendering like colour grading, bloom and depth of field (camera focus)

Scriptable Render Pipeline

Source Unity - Manual: Creating a Render Pipeline Asset and Render Pipeline

Instance in a custom render pipeline (unity3d.com)

The Scriptable Render Pipeline (SRP) was released in 2018 as a high level API to be able to control the Unity's rendering loop as well as add additional features/passes with C#. SRP gives complete control of the render pipeline over to the developer, and therefore really require expertise and knowledge of graphics engineering to properly use.

To get started with a customized render pipeline utilizing SRP, the branch from the GitHub - Unity-Technologies/Graphics: Unity Graphics - Including Scriptable Render Pipeline repo corresponding to the version of the Unity Editor that is being used. The following packages should be imported in the listed order by clicking on the `+` button on the top left of the package manager and choosing "Add package from disk..." and selecting the `package.json` file in the root of the package which is located in the subfolder of the git repo: `com.unity.render-pipelines.core`, `com.unity.shadergraph` (optional), `com.unity.visualaffectgraph` (optional).

All SRPs utilize 2 basic C# classes `RenderPipeline` & `RenderPipelineInstance`. `RenderPipelineAsset` is attached to an asset that acts as a factory to produce the `RenderPipeline`. It must inherit from `RenderPipelineAsset` and override the `CreatePipeline` method that returns a `RenderPipeline` instance. This can be used to produce different graphics settings depending on the target or choice of the user. An example of a basic `RenderPipelineAsset` is

```
using UnityEngine;
using UnityEngine.Rendering;

// The CreateAssetMenu attribute lets you create instances of this
// class in the Unity Editor.
[CreateAssetMenu(menuName = "Rendering/ExampleRenderPipelineAsset")]
public class ExampleRenderPipelineAsset : RenderPipelineAsset
{
    // Unity calls this method before rendering the first frame.
    // If a setting on the Render Pipeline Asset changes, Unity
    // destroys the current Render Pipeline Instance and calls this
    // method again before rendering the next frame.
    protected override RenderPipeline CreatePipeline() {
        // Instantiate the Render Pipeline that this custom SRP uses
        // for rendering.
        return new ExampleRenderPipelineInstance();
    }
}
```

The child of `RenderPipeline` that is returned in this example is

```
using UnityEngine;
using UnityEngine.Rendering;

public class ExampleRenderPipelineInstance : RenderPipeline
{
    public ExampleRenderPipelineInstance() {
    }

    protected override void Render (ScriptableRenderContext context,
                                    Camera[] cameras) {
        // This is where you can write custom rendering code.
        // Customize this method to customize your SRP.
    }
}
```

Once those are created, a Render Pipeline Asset can be created by clicking on the `+` in the top left of the project view and select Rendering -> Example Render Pipeline Asset.

Then, the `RenderPipelineAsset` asset needs to be set as the render asset which can be done in the project settings under Graphics and dragging the asset in to the Scriptable Render Pipeline Settings box.

The render pipeline class has to inherit from the `RenderPipeline` class and override that `Render` method. The `Render` method is used to execute the render pipeline. The `ScriptableRenderContext` is the API to execute the command and is an interface between C# and the low-level graphics code. `context` is a `ScriptableRenderContext` which defines state and drawing command that custom render pipelines use to schedule and submit state updates and drawing command to the GPU. The culling, rendering and post-processing is all scheduled manually.

The asset can be used to store data and the like. An example of a `RenderPipelineAsset` & `RenderPipeline` would be

```
using UnityEngine;
using UnityEngine.Rendering;

// The CreateAssetMenu attribute lets you create instances of this
// class in the Unity Editor.
[CreateAssetMenu(menuName = "Rendering/ExampleRenderPipelineAsset")]
public class ExampleRenderPipelineAsset : RenderPipelineAsset
{
    // This data can be defined in the Inspector for each Render
    // Pipeline Asset
    public Color exampleColor;
    public string exampleString;

        // Unity calls this method before rendering the first frame.
        // If a setting on the Render Pipeline Asset changes, Unity
        // destroys the current Render Pipeline Instance and calls
        // this method again before rendering the next frame.
    protected override RenderPipeline CreatePipeline() {
        // Instantiate the Render Pipeline that this custom SRP uses
        // for rendering, and pass a reference to this Render
        // Pipeline Asset. The Render Pipeline Instance can then
        // access the configuration data defined above.
        return new ExampleRenderPipelineInstance(this);
    }
}
```

Source Unity - Manual: Creating a simple render loop in a custom render pipeline

```
using UnityEngine;
using UnityEngine.Rendering;

public class ExampleRenderPipelineInstance : RenderPipeline
{
    // Use this variable to a reference to the Render Pipeline Asset
    // that was passed to the constructor
    private ExampleRenderPipelineAsset renderPipelineAsset;

    // The constructor has an instance of the
    // ExampleRenderPipelineAsset class as its parameter.
    public ExampleRenderPipelineInstance(
                        ExampleRenderPipelineAsset asset) {
        renderPipelineAsset = asset;
    }

    protected override void Render(ScriptableRenderContext context,
                                   Camera[] cameras) {
        // This is an example of using the data from the Render
        // Pipeline Asset.
        Debug.Log(renderPipelineAsset.exampleString);

        // This is where you can write custom rendering code.
        // Customize this method to customize your SRP.
    }
}
```

Code execution are primarily performed in 2 ways. Either a `CommandBuffer` can be passed to the `ScriptableRenderContext.ExecuteCommandBuffer` or the API calls can be made directly. The commands are then executed once `ScriptableRenderPipelineContext.Submit` is called. `CommandBuffers` can also be executed immediately by calling `Graphics.ExecuteCommandBuffer`. Calls to this API take place outside of the render pipeline.

Once the renderer script, the render pipeline script and the renderer pipeline asset are all set up, the renderer can render the scene in the `Render` Method. An example of a basic rendering the following `RenderPipeline` derivative is as follows

```
/*
This is a simplified example of a custom Scriptable Render Pipeline.
It demonstrates how a basic render loop works.
It shows the clearest workflow, rather than the most efficient runtime
*/

using UnityEngine;
using UnityEngine.Rendering;

public class ExampleRenderPipeline : RenderPipeline {
    public ExampleRenderPipeline() {
    }

    protected override void Render (ScriptableRenderContext context,
                                    Camera[] cameras) {
        // Create and schedule a command to clear the current render
        // target
        var cmd = new CommandBuffer();
        cmd.ClearRenderTarget(true, true, Color.black);
        context.ExecuteCommandBuffer(cmd);
        cmd.Release();

        // Iterate over all Cameras
        foreach (Camera camera in cameras)
        {
            // Get the culling parameters from the current Camera
            camera.TryGetCullingParameters(out var cullingParameters);

            // Use the culling parameters to perform a cull operation,
            // and store the results
            var cullingResults = context.Cull(ref cullingParameters);

            // Update the value of built-in shader variables, based on
            // the current Camera
            context.SetupCameraProperties(camera);

            // Tell Unity which geometry to draw, based on its
            // LightMode Pass tag value
            ShaderTagId shaderTagId = new("ExampleLightModeTag");

            // Tell Unity how to sort the geometry,
            // based on the current Camera
            var sortingSettings = new SortingSettings(camera);

            // Create a DrawingSettings struct that describes which
            // geometry to draw and how to draw it
            DrawingSettings drawingSettings = new(shaderTagId,
                                                  sortingSettings);

            // Tell Unity how to filter the culling results, to further
            // specify which geometry to draw. Use
            // FilteringSettings.defaultValue to specify no filtering
            FilteringSettings filteringSettings =
                                         FilteringSettings.defaultValue;

            // Schedule a command to draw the geometry,
            // based on the settings you have defined
            context.DrawRenderers(cullingResults, ref drawingSettings,
                                  ref filteringSettings);

            // Schedule a command to draw the Skybox if required
            if (camera.clearFlags == CameraClearFlags.Skybox &&
                RenderSettings.skybox != null)
            {
                context.DrawSkybox(camera);
            }

            // Instruct the graphics API
            // to perform all scheduled commands
            context.Submit();
        }
    }
}
```

For the above example to work, the following custom shader must be saved, as the regular shader as the built-in render pipeline are usable. There must also be a objects in the scene to be scene, with a material associated with the objects and the following shader associated with the material.

```
// This defines a simple unlit Shader object that is compatible with a
// custom Scriptable Render Pipeline. It applies a hardcoded color, and
// demonstrates the use of the LightMode Pass tag.
// It is not compatible with SRP Batcher.

Shader "Examples/SimpleUnlitColor"
{
    SubShader
    {
        Pass
        {
            // The value of the LightMode Pass tag must match the
            // ShaderTagId in ScriptableRenderContext.DrawRenderers
            Tags { "LightMode" = "ExampleLightModeTag"}

            HLSLPROGRAM
            #pragma vertex vert
            #pragma fragment frag

    float4x4 unity_MatrixVP;
            float4x4 unity_ObjectToWorld;

            struct Attributes
            {
                float4 positionOS   : POSITION;
            };

            struct Varyings
            {
                float4 positionCS : SV_POSITION;
            };

            Varyings vert (Attributes IN)
            {
                Varyings OUT;
                float4 worldPos = mul(unity_ObjectToWorld,
                                      IN.positionOS);
                OUT.positionCS = mul(unity_MatrixVP, worldPos);
                return OUT;
            }

            float4 frag (Varyings IN) : SV_TARGET
            {
                return float4(0.5,1,0.5,1);
            }
            ENDHLSL
        }
    }
}
```

This works because the, the renderer gets all the shaders with the tag "ExampleLightModeTag" to render their materials and thus objects. The colour of the objects can be adjust by adjusting, the RBGA float vector on the 6th last line of the shader code above (see attached project titled "Customized SRP" for demo)

One of the primary advantages of using the SRP is that custom render passes and fine grain control of what is rendered and when it is rendered. Some of these include camera stacking (rendering multiple cameras over each other to avoid collisions for guns/weapons or in a cockpit for example), easy and built-in post processing render passes and applying post processing at a specific stage in the render pipeline.

Because the full pipeline if controlled by the script, and Unity provides the ability to both tag shaders with IDs and get and render all the shaders of a specific tag at any point in the render pipeline. This allows post processing affects to only be applied to certain objects (opaque for example).

Another advantage of having full control of the pipeline is that operations (sorting objects) that are computationally expensive and aren't necessary can be removed from the pipeline when unnecessary to save time.

Pre-built Render Pipelines Based the SRP

Unity provides 2 pre-built SRPs, he High Definition Render Pipeline & and the Universal Render Pipeline.

Both the URP and HDRP have 2 primary ways of being used in a project. Both the HDRP and URP have templates that a project can be created with in the Unity Hub (only from 2021 LTS editor version, not 2020 LTS). There are also sample scenes for each of the pre-built SRPs. A project can also be upgraded to use either the HDRP or URP from the package manager, by installing either the HDRP or URP package. Upgrading a project from the built-in rendered isn't an easily downgradable action.

When upgrading a project from the built-in render pipeline to either the HDRP or URP using the package manager, the render pipeline needs to be configured to work properly. When using the HDRP, the render pipeline wizard will pop up with the option to automatically fix any incompatibility issues with the current project. After the are all fixed there will be an option create a `RenderPipelineAsset` (see above description of purpose and functionality). For the URP, a render pipeline asset and renderer will have to be created from the Project View the same way any other asset is created under Create -> Rendering -> Universal Render Pipeline -> Pipeline Asset

(Forward Renderer).

From the Quality settings in the Project settings, different quality settings can be set, which each have a `RenderPipelineAsset` file associated with them, and the default can be set. These can be used to set different qualities for different target platforms and can be swapped programmatically as well with a regular C# script that inherits from `MonoBehaviour`. For each of these assets, the quality can be set as well as features can be toggled, like, for example, shadows. When opening the asset in the inspector view, it shows the customization and extensions possible with in the given pre-built SRP. For the each renderer asset, multiple renderers can be created with one as default and other can be switched to programmatically.

High Definition Render Pipeline (HDRP)

HDRP offers advanced rendering and shading features and is designed for PC and advanced console projects that require a high degree of visual fidelity. This pipeline is a hybrid deferred/forward tile/cluster renderer. this allow easy and performant transparent materials and offers both faster processing of the lighting and considerable reduction in bandwidth consumption compared to the Built-in Render Pipeline (BRP).

HDRP is intended to be considerably more realistic than the other render pipelines. This shows in a number of ways. The camera can be set up using real world aperture and focal length as well as light intensities use lumens to better reflect the real world. Lighting is available in more shapes with HDRP, such as rectangle, tube and disk. HDRP comes with realistic shaders for materials and hair, for example and reflections take surface smoothness in to account. There is a wider range of material types that are available. For example, anisotropy, iridescence, metallic, specular colour, subsurface scattering and translucent.

HDRP works off of a volume framework. The volume framework allows for splitting up a scene in to different sections, each with their own settings. Each volume is a game object that just stores a reference to a Volume Profile (explained below). Each volume is either local, which only takes effect when the camera is within the actual volume object, or global, which applies everywhere.

The Volume Profile for each volume is saved to disk and can therefore be shared between many different volumes. The profile begins with a set of default properties and which can be edited by adding a Volume Override which can be used to override the settings for the following affect Exposure, fog, lighting (ambient occlusion, indirect lighting controller, screen space global illumination, screen space reflection and screen space refraction), diffusion profile (settings for subsurface scattering), post-processing (discussed later), shadowing, sky (cloud layer, volumetric clouds) and visual environment (sky type).

3 variables control which volume is used, priority, blending and weight. Blending is a smoothing curve that unity applies to ease the application of the volume to the camera.

This is controlled by the blend distance which is the distance that unity starts to apply the volume's overrides. A value of 0 means an instant transition, while other positive values mean the the volume overrides begin blending once the camera enters a specified range. The priority of a volume is used when 2 volumes overlap (local and local or local and global) to decide which volume to use. Lastly, weight is used be unity to as a multiplier to the values that are calculated based on camera position and blend distance.

There is a similar type of volume that the HDRP uses for custom passes but without a weight value and fade radius referring to blending distance. These volume apply one of 2 passes to the screen at some point in the render pipeline. There are 2 extra properties of a custom pass volume with are injection point, the point in the render pipeline to render the pass, and custom passes, which is a list of custom passes to render. The 2 types of custom render passes available by default are FullScreen Custom Pass, execute an affect on the camera view, or a DrawRenderers Custom Pass, execute an affect on the visible objects. An example of a FullScreen Custom Pass is drawing red to the screen to simulate low health. An example of a DrawRenderers Custom Pass is outlining objects on the screen. Both of these need a material with a shader to be specified. Other custom passes can be scripted in C# by inheriting from `CustomPass` and implementing the following methods

```
protected override void Setup(ScriptableRenderContext renderContext, Co

protected override void Execute(CustomPassContext ctx)

protected override void Cleanup
```

See Scripting your own Custom Pass in C# | High Definition RP | 10.8.1 for more details.

HDRP has a subsurface scattering type for materials. This is used for materials where the light is supposed to refract through the material, but is not transparent. Common materials are skin or leaves. For example. when one shines a light through a finger, the light is noticeable on the other side and the light can even have a different colour. This is also happens in bones and some shells. In order for sub surface scattering to be configured, a diffusion profile has to be created (project view -> create -> rendering -> diffusion profile) The the intensity, colour as well as other properties of the scattering can be chosen. (See attached project "HDRP") (**Source**: Tutorial: Subsurface Scattering in Unity 2019 (HDRP SSS))

In general, the HDRP give a lot of power to control practically anything and everything in the project, however a lot more tuning is required because of the high level of control for the project.

Universal Render Pipeline (URP)

Source: Evolving game graphics with the Universal Render Pipeline | Unite Now 2020

URP is a fast single pass forward renderer and has been designed to additionally support lower-end devices lacking support for compute shader technology, such as older smartphones, tables and AR/VR/MR devices. URP, however, can also deliver higher quality graphics for midrange devices such as consoles and PC, sometimes for a lower performance cost than the BRP. The lights are culled per-object and allow for the lighting to be computed in one single pass, which results in reduced draw calls compared to the Built-In Render Pipeline but limits the possibilities with multiple realtime shadowing lights.

Note: The video is made for an older version of the Unity Editor which mean that it some of the detail were modified for the aforementioned LTS Unity Editor version

In URP, custom render passes are easily added. For each render asset, a list of renderers are chosen in the inspector view. For the each renderer the settings and overrides can be set in the inspector view, as well as Render Features can be added.

A Render Feature is a customer render pass in URP. Examples are creating realistic wave and other watering affects, show a silhouette of an object obstructed by another object, pixelation, outlining objects and Ambient Occlusion (see below). URP provides 2 pre-built render features.

Render Objects is a feature that only renders a material given a condition is true, for example, when an object is obstructed by another object. This can be used to give the appearance of X-Ray vision. (See attach Unity project titled "URP"). This is done by assigning the material to a separate layer that is to be drawn by when the condition is true and telling the Render Objects feature to render that layer when the specified condition is true.

The other render pass is ambient occlusion which effect darkens creases, holes, intersections and surfaces that are close to each other. In the real world, such areas tend to blocked out from or are occluded from ambient light, so they appear darker.

To add a custom render feature, one can be created by right clicking in the project view and selecting Create -> Rendering -> Universal Render Pipeline -> Render Feature which creates a C# script that has a custom render pass class and the basic mechanics to render that custom pass. The render feature is added to the renderer script by click on the script, clicking on Add Render Feature in the inspector view and selecting the desired render feature.

For all render features, the point of insertion in to the render pipeline can be specified, to only apply the render pass to the already rendered objects. This can be use, for example to not apply pixelation to transparent objects, or only drawing outlines around opaque objects.

Another feature of URP is camera stacking, where other views can be stack on top of the base camera. This can be used, for example, to render UI or a gun in a FPS game. This can be done by setting the stacked camera as overlay and adding the other cameras to the camera stack list in the inspector view.

URP also uses volumes (as discussed above in the HDRP section) for the same post-processing affect as the HDRP. This is also the same way it is done in the BRP with the post-processing stack v2 package from Unity.

Shader Graph

Shader graph is a node based shader editor for HDRP & URP that can act as an accessible way to build complex shaders. A Shader create can be created from the project view under Create -> Shader -> Blank Shader Graph. A right click or space bar will bring up a searchable list of the nodes that are applicable if the edge of a node was dragged from (attempting to be connected to another node) but wasn't connect, a contextual list of node with matching datatype will be shown. The 2 sections of nodes that are by default on the graph of a blank shader graph are the vertex and fragment shader values. Unity provides lots of commonly used functionality to build upon. It is also possible to build nodes from HLSL shader code that can be used as part of shader graph and of course, regular shaders can be implemented entirely with HLSL. The second type of shader graph is a subgraph which is intended to be a reusable graph as part of a larger shader graph.

The blackboard is an overlay on the left side of the shader graph application that lists the properties of the shader (inputs). These are inputs, such as floating point numbers, colours etc. to the shader that can be specified on the inspector view of the associated material.

As of Unity Editor version 2021.2 and above, Shader Graph is available in the built-in render pipeline.