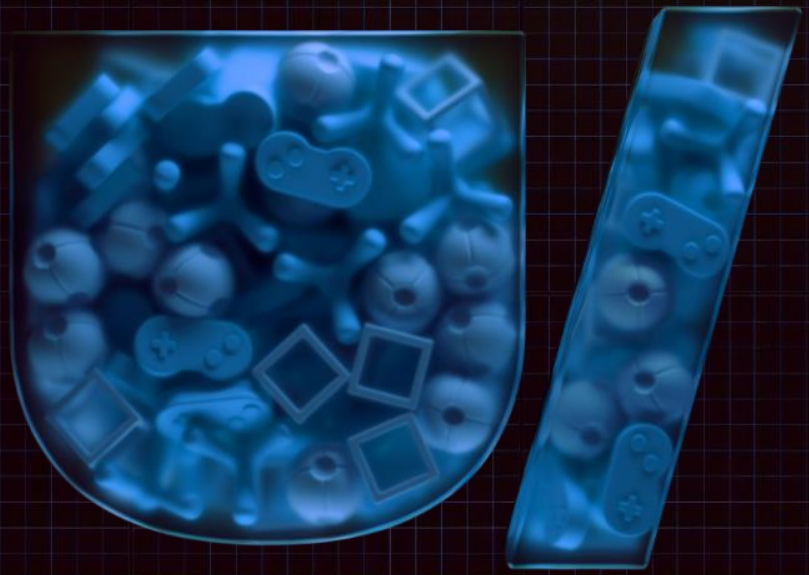


Unity6向け ShaderVariantガイド (ウォームアップとストリッピング)

黒河 優介

Partner Engineer

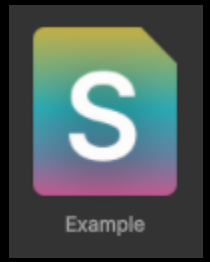
Agenda

- → ShaderVariantとは？
- → 使用しているShaderVariantを特定する
- → Warmupによるspike回避
- → ビルド時間や使用メモリを抑えるための ShaderVariantStrip
- → Dynamic shader variant loading

ShaderVariantとは？

ShaderVariant？

- → 1つのShaderを色んなシチュエーションで使いたいため、Keywordによる分岐処理を行います
- → 個々のKeywordの組み合わせに応じたものを ShaderVariantと呼びます
- → Unity内部でも様々な光源環境でも一つのShaderで対応できるように沢山のKeywordが定義されています
  - DirectionI Lightの有無、PointLightの数、ライトマップの有無など沢山のShaderVariantがあります



<no keyword>

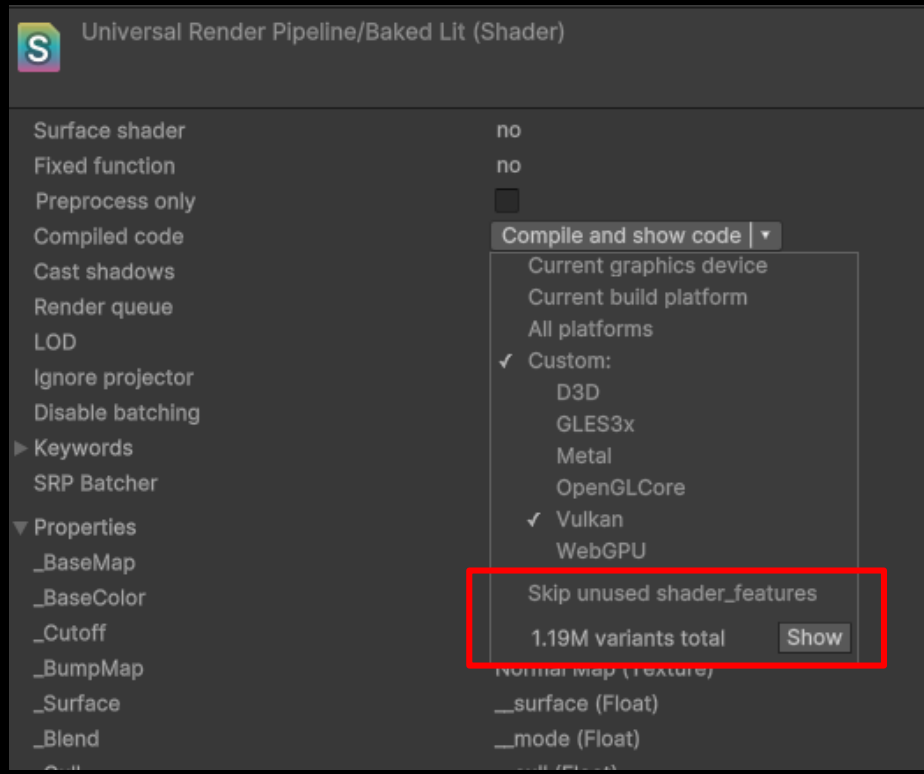
_NEGATIVE

Code example
Keywordの実装

```
// キーワードの宣言、 なし、もしくは _NEGATIVEで分岐
#pragma multi_compile _ _NEGATIVE
// QUALITY_LOW / QUALITY_MEDIUM / QUALITY_HIGH の内から三つ
#pragma multi_compile QUALITY_LOW QUALITY_MEDIUM QUALITY_HIGH

// ShaderFeatureを使うと、エンジンが利用状況を調べ、
// 必要に応じてStripされる。これも二つで分岐する
#pragma shader_feature _ _REDONLY

// 実際の利用
fixed4 frag () : SV_Target{
   fixed4 color ;
   // キーワード毎に処理を分岐
   #ifdef _NEGATIVE
        return fixed4(1-color.x, 1-color.y, 1-color.z, color.w);
   #else
        return color;
   #endif
   // 複数の場合
   #if QUALITY_LOW
   #elif QUALITY_MEDIUM
   #elif QUALITY_HIGH
   #endif
}

```

ShaderVariantを何故勉強するのですか？

- → Keywordの組み合わせで増える ShaderVariantは、Keywordの数が増えると掛け算で増えていく
  - そのため数も膨大になりやすい
- → Unity側で明らかに使わそうな部分は予め省いてくれる
  - 実際には使用されない ShaderVariantがいくつか入る
- → どのShaderVariantを入れるかをもっと細かく指定したい場合APIで制御可能
  - 多すぎても、少なすぎても問題



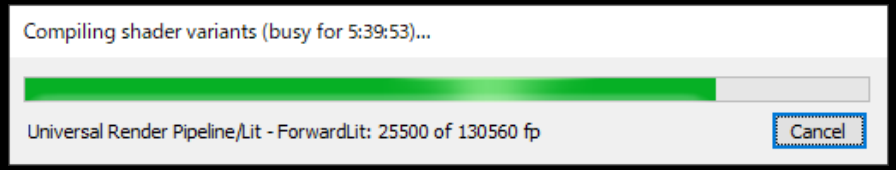
Unity6の**URP Lit**シェーダーには**100**万以上の**ShaderVaraint**があります。
(すべての **shader_features** あり**)**

ShaderVariantを多く入れすぎてしまったときの場合**(1)**

→ ShaderVariantが多いほど、ビルドに時間が掛かる

- ビルドした結果がキャッシュされるので、二回目以降のビルドでは問題ない

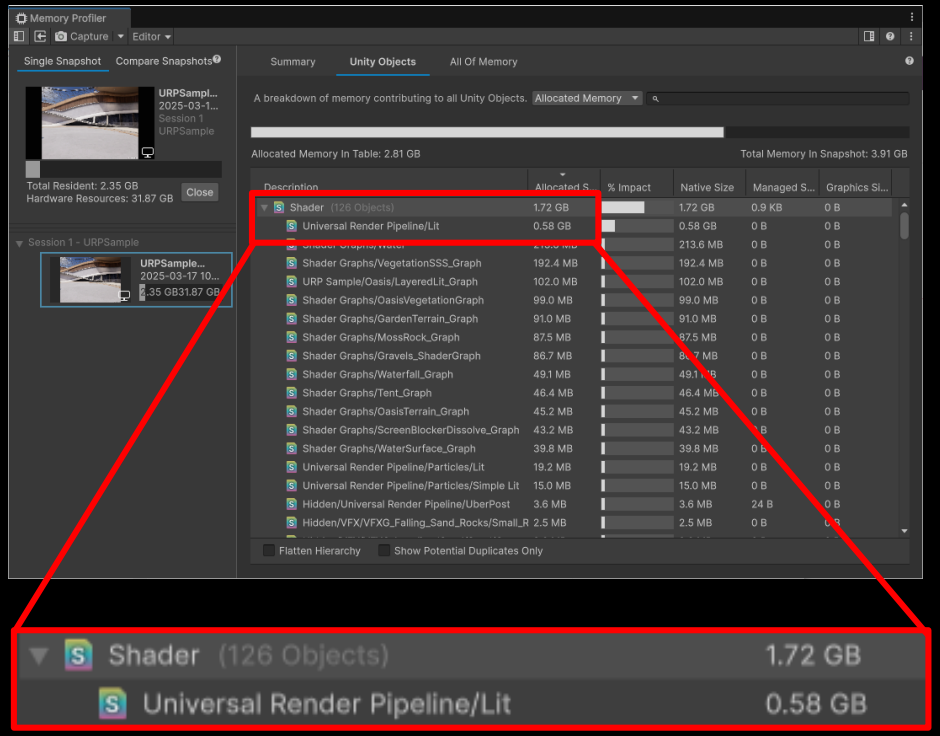
→ 事前にShaderVariniantをStripすれば、初回ビルドの時間は大きく減らせる



Compiling shader variantsが異常に長い場合は、気を付けると良い

ShaderVariantを多く入れすぎてしまったときの場合**(2)**

- → Shaderの持つShaderVariantが増えれば増えるほど、メモリの使用量が増えます
- → ShaderVariantを減らすことでメモリ使用量を減らせます



これは敢えて多くの**ShaderVariant**を含むようにした例です。
１つの**Shader**だけで**0.58GB**もメモリを使います。

ShaderVariantが少なすぎる場合(1)

→ 完全に一致するShaderVariantが見つからない場合、別のShaderVariantへFallbackするようになっている
  - その結果、見た目に差が出てしまう

Editor実行時

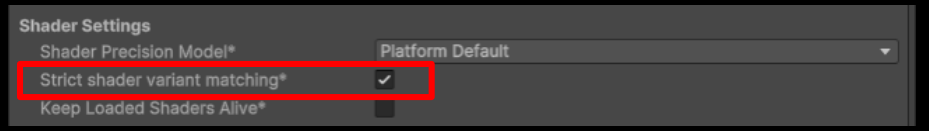
Build実行時

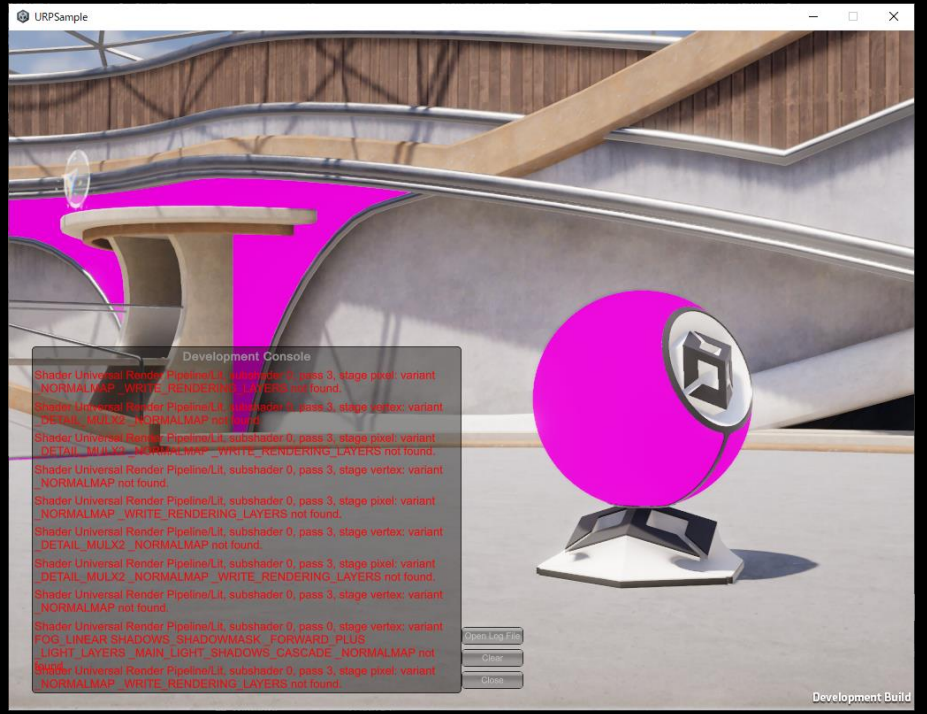
ShaderVariantが少なすぎる場合(2)

PlayerSettings.strictShaderVariantMatching

→ ShaderVariantがマッチしない時に、明確にエラーを出したい場合を有効にする

→ エラーログを出力し、ピンク色にフォールバックするようになる





ShaderVariant とは？

- → Shader内でのKeyword分岐によって出来るのがShaderVariant
- → ShaderVariantの数が多いと、ビルド時間やメモリ使用量に影響する
- → 少なすぎると見た目に影響
- → 適切に管理する必要がある

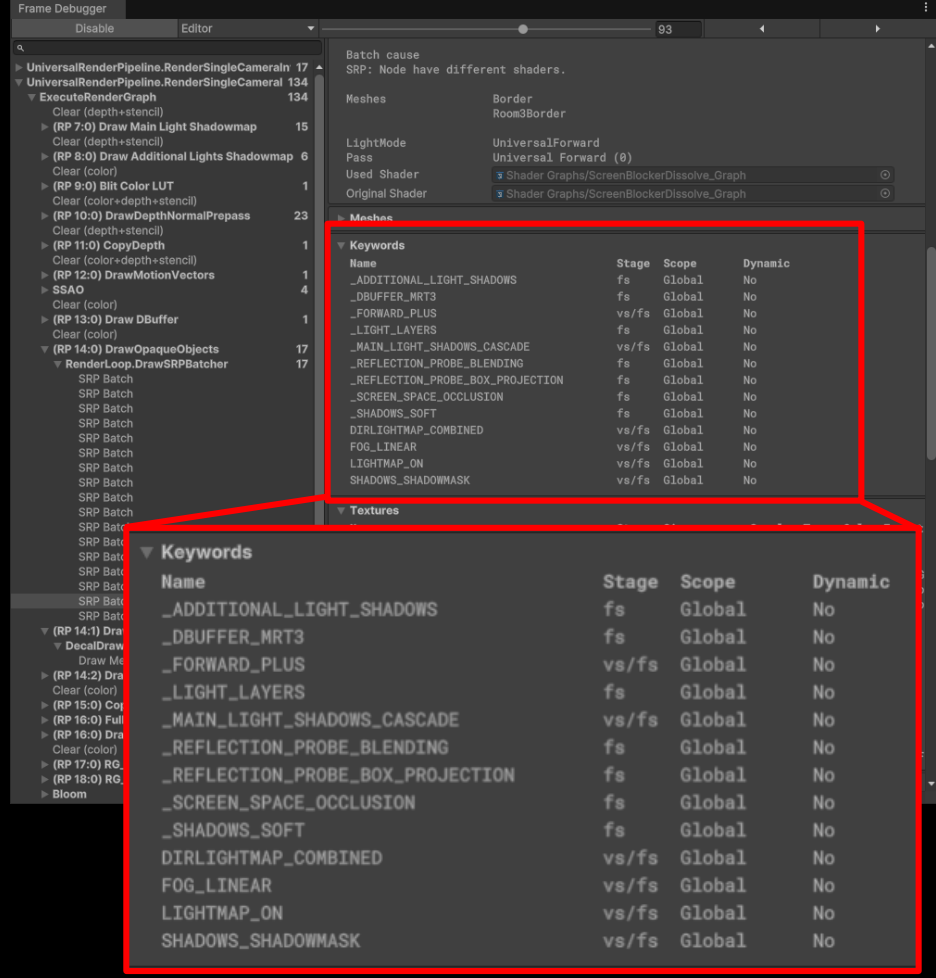
使用している ShaderVariantを特定する

使用している**ShaderVariant**を特定する

- → FrameDebuggerでKeywordを調べる
- → ProfilerでCreateGPUProgramのMetadataを見る
- → GraphicsStateCollection APIを使う(Unity 6.0 +)

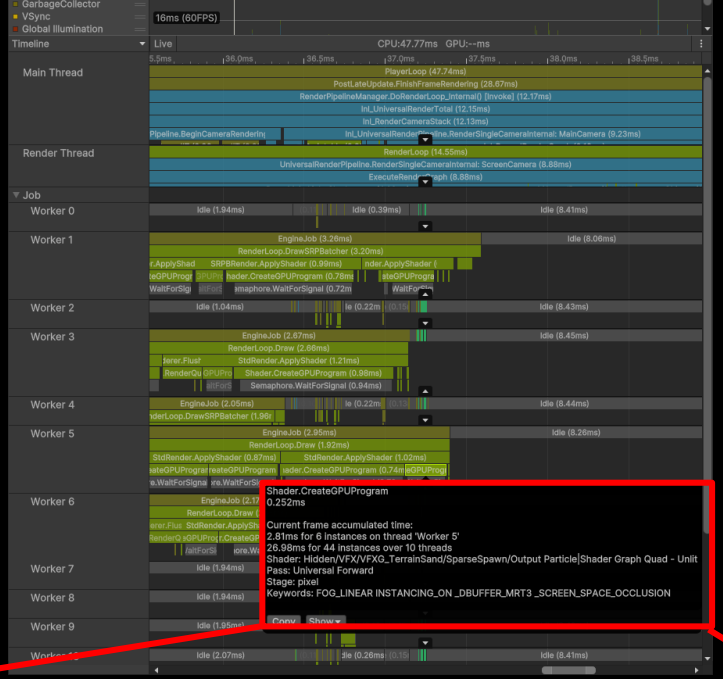
FrameDebuggerでKeywordを調べる

→ FrameDebuggerには Shaderのどの Keywordで描画しているかが表示される



ProfilerでCreateGPUProgramの Metadataを見る

- → はじめて描画するShaderVariantの場合、Warmup処理を行う
- → その時にCreateGPUProgramという名前でProfilerに表示される
- → Metadataをみると、Shader名・Pass・Stage・Keyword等の情報が確認出来る



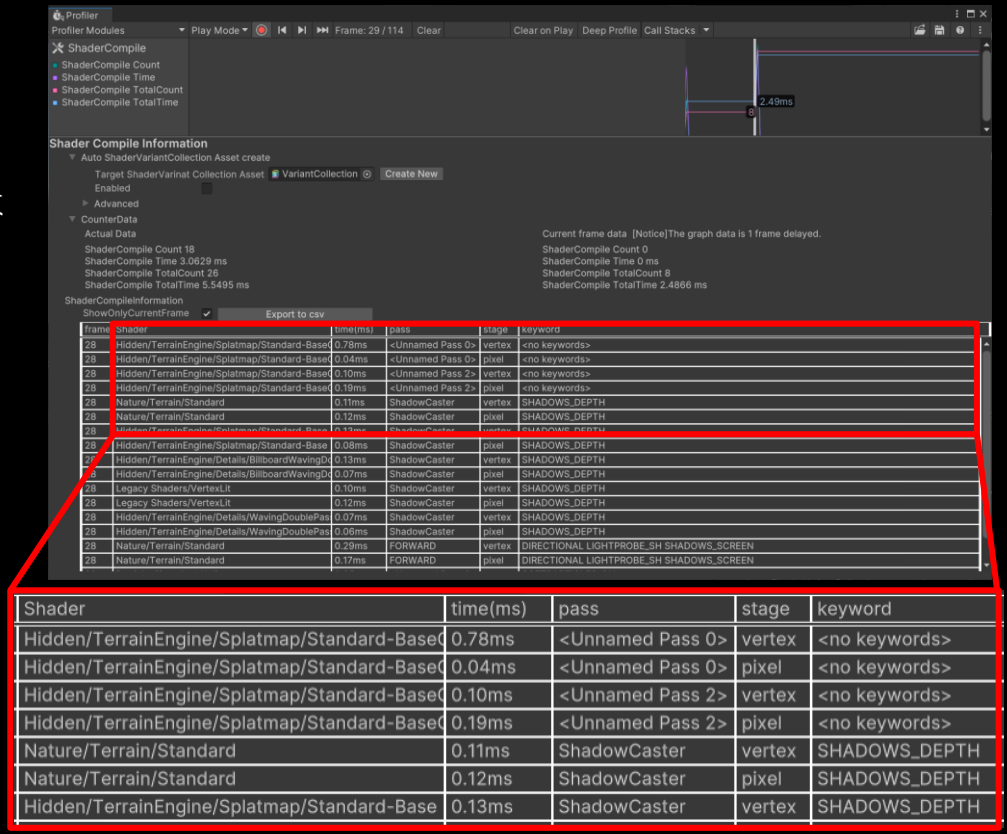
Shader.CreateGPUProgram
0.252ms

Current frame accumulated time:
2.81ms for 6 instances on thread 'Worker 5'
26.98ms for 44 instances over 10 threads
Shader: Hidden/VFX/VFXG_TerrainSand/SparseSpawn/Output Particle|Shader Graph Quad - Unlit
Pass: Universal Forward
Stage: pixel
Keywords: FOG_LINEAR INSTANCING_ON _DBUFFER_MRT3 _SCREEN_SPACE_OCCLUSION

ProfilerでCreateGPUProgramのMetadataを見る(2)

- → Profiler上で、一つずつ見つけていくのはとても大変
- → 一覧で見られるツールを作りました
  - ProfilerAPIを使ってリスト化

https://github.com/wotakuro/ProfilerModuleForShaderCompile/



Shader	time(ms)	pass	stage	keyword
Hidden/TerrainEngine/Splatmap/Standard-Base(	0.78ms	<Unnamed Pass 0>	vertex	<no keywords>
Hidden/TerrainEngine/Splatmap/Standard-Base(	0.04ms	<Unnamed Pass 0>	pixel	<no keywords>
Hidden/TerrainEngine/Splatmap/Standard-Base(	0.10ms	<Unnamed Pass 2>	vertex	<no keywords>
Hidden/TerrainEngine/Splatmap/Standard-Base(	0.19ms	<Unnamed Pass 2>	pixel	<no keywords>
Nature/Terrain/Standard	0.11ms	ShadowCaster	vertex	SHADOWS_DEPTH
Nature/Terrain/Standard	0.12ms	ShadowCaster	pixel	SHADOWS_DEPTH
Hidden/TerrainEngine/Splatmap/Standard-Base	0.13ms	ShadowCaster	vertex	SHADOWS_DEPTH

GraphicsStateCollection APIを使う(Unity 6.0 +)

"GraphicsStateCollection" API

- → Experimentalで提供中
- → 実行中にBeginTrace/EndTraceでShaderVariantを収集可能
- → SaveToFileでローカルに保存可能
- → SendToEditorで接続中のEditorに送信可能
  - *“.graphicsstate”* ファイルはそのままAssetとして認識される
- → DirectX11やOpenGLESでもTraceは可能

```
using UnityEngine;
using UnityEngine.Experimental.Rendering;
using UnityEngine.Networking.PlayerConnection;

public class SendToEditorExample : MonoBehaviour
{
    public GraphicsStateCollection graphicsStateCollection;
    public string fileName;

    void Start()
    {
        graphicsStateCollection = new GraphicsStateCollection();
        graphicsStateCollection.BeginTrace();
    }

    void OnDestroy()
    {
        graphicsStateCollection.EndTrace();
        if (PlayerConnection.instance.isConnected)
        {
            graphicsStateCollection.SendToEditor(fileName);
        }
        else
        {
            Debug.Log("No PlayerConnection found!");
        }
    }
}
```

GraphicsStateCollection から ShaderVariantを取得する

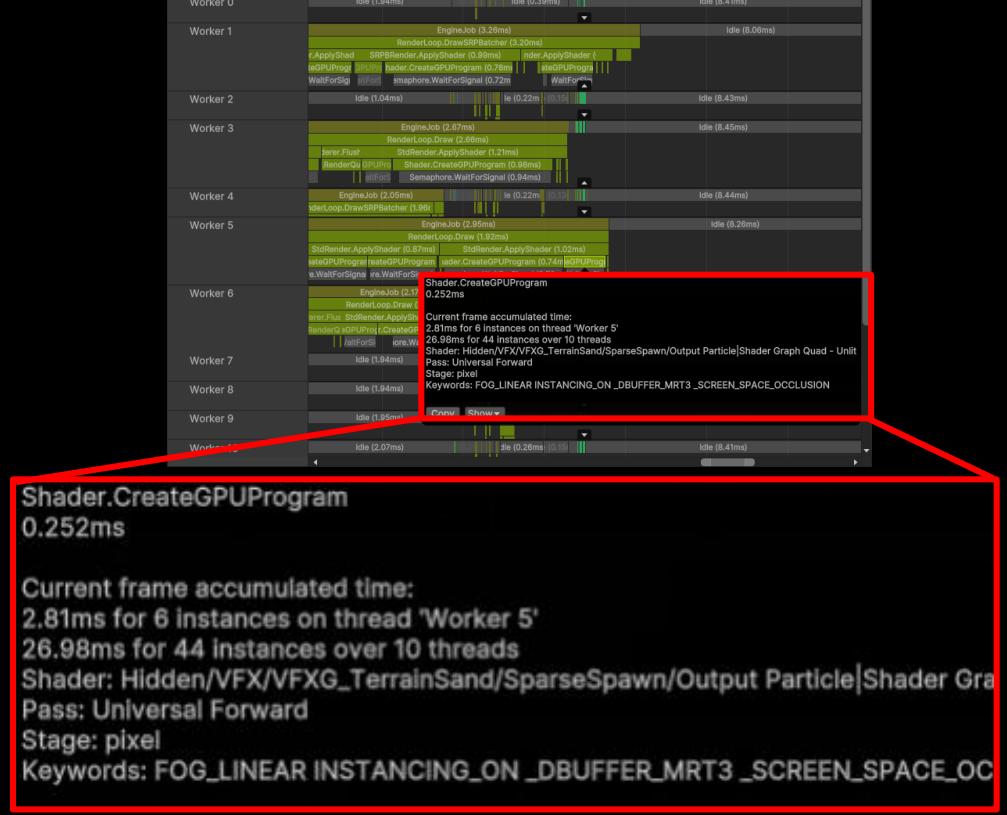
GraphicsStateCollection.GetVariants で ShaderVariant一覧を取得することが可能

```
using System.Collections.Generic;
using System.Text;
using UnityEngine;
using UnityEngine.Experimental.Rendering;

public class GetVariantsExample : MonoBehaviour
{
    public GraphicsStateCollection graphicsStateCollection;

    void Start()
    {
        var variants = new List<GraphicsStateCollection.ShaderVariant>();
        graphicsStateCollection.GetVariants(variants);
        foreach(var variant in variants)
        {
            Debug.Log( variant.shader.name + "::" + GetKeywordString(variant) );
        }
    }

    private static string GetKeywordString( GraphicsStateCollection.ShaderVariant variant)
    {
        var sb = new StringBuilder();
        if (variant.keywords == null || variant.keywords.Length == 0)
        {
            return "";
        }
        foreach (var keyword in variant.keywords)
        {
            sb.Append(keyword.name).Append(" ");
        }
        return sb.ToString();
    }
}
```

Warmupによるspike回避

Shader.CreateGPUProgram

- → はじめてShaderVariantを描画する時にWarmup処理が走る
- → ライティング環境の変化などにより、使用するShaderKeywordが変わり、意図しないタイミングでWarmup処理が走りSpikeとなってしまう事がある



Shader.CreateGPUProgram
0.252ms

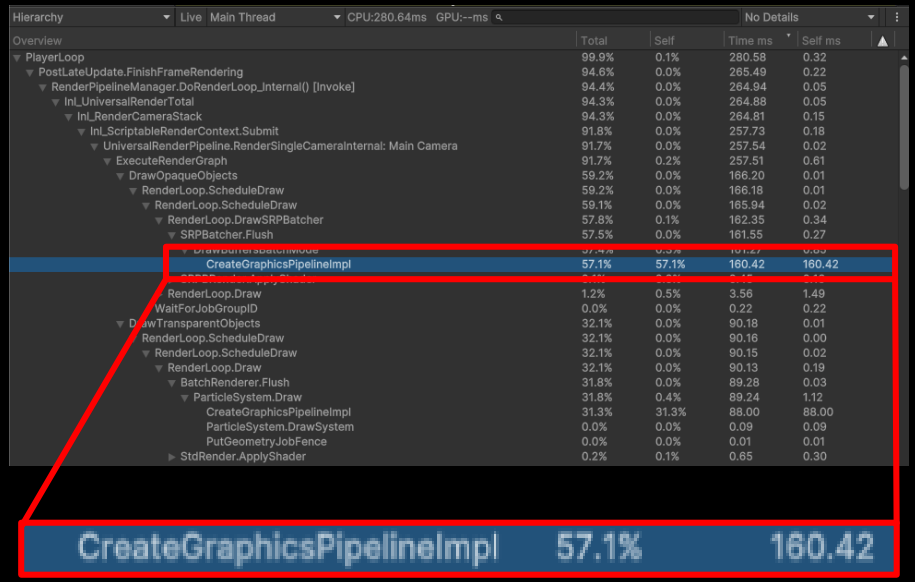
Current frame accumulated time:
2.81ms for 6 instances on thread 'Worker 5'
26.98ms for 44 instances over 10 threads
Shader: Hidden/VFX/VFXG_TerrainSand/SparseSpawn/Output Particle|Shader Gra
Pass: Universal Forward
Stage: pixel
Keywords: FOG_LINEAR INSTANCING_ON _DBUFFER_MRT3 _SCREEN_SPACE_OC

CreateGPUPipelineImp

→ DirectX12 / Metal / Vulkanでは PSO(Pipeline State Object)が異なっていてもWarmup処理が走る

→ 同じShaderVariantでも 使用する頂点レイアウトが異なるオブジェクトを描画するとWarmupしなおす必要がある

- 頂点レイアウト：
  Position - UV1 - Normal
  Position - UV1 -UV2- Normal等



DirectX11やOpenGLES等での事前Warmup処理

- → DirectX11やOpenGLES等では"CreateGPUProgram"によるSpikeのみを考慮すればよい
- → ShaderVariantCollectionアセットを予め構築し、ランタイム上でShaderVariantCollection.Warmupを呼び出す
  - これによって"CreateGPUProgram"を事前に行ってしまい、意図しないタイミングでのSpikeを防ぐことが可能

DirectX12 / Metal / Vulkanでの事前Warmup処理(1)

→ Unity6より前では、事前のWarmup処理は困難だった

- 全てのオブジェクトの頂点レイアウトとShaderVariantの組み合わせを事前に調べることが困難
- Editor上で調べてもBuildの最適化の過程で頂点レイアウトが変わってしまう事がある
- 実機上で動かして、RenderDoc等で地道に調査していく必要があった

DirectX12 / Metal / Vulkanでの事前Warmup処理(2)

→ Unity6では ***GraphicsStateCollection*** APIを使う

- 実機上で事前にGraphicsStateCollectionのBeginTrace/EndTraceをして、GraphicsStateCollectionを構築しておく
- GraphicsStateCollectionのWarmup処理を事前にする

```
using UnityEngine;
using UnityEngine.Experimental.Rendering;
using Unity.Jobs;

public class WarmUpSynchronousExample : MonoBehaviour
{
    public GraphicsStateCollection graphicsStateCollection;

    void Start()
    {
        JobHandle handle = graphicsStateCollection.WarmUp();
        handle.Complete();
    }
}
```

ビルド時間や使用メモリを抑えるための ShaderVariantStrip

ShaderVariant Stripを行うAPI

- → ShaderKeywordFilter
  - 特定のキーワードを全く使用しないと言ったケースで利用
  - RenderPipelineそのものを開発しないならば、直接触れることはない
- → IPreProcessShaders.OnProcessShader
  - ビルド時にCallbackされる
  - 引数で渡されるShaderVariantのListを編集し、どのShaderVariantをビルドに含めるかを決定する
  - RenderPipeline内に実装があるが、ユーザー側でも追加で実装可能

Code example
ShaderKeywordFilterの実装

```
using UnityEditor.ShaderKeywordFilter;

// RenderPipelineAssetの変数にAttributeを付けることで実装します
public class UniversalRenderPipelineAsset : RenderPipelineAsset {

  // trueの場合、キーワード “FeawtureA” を含むVariantはビルドの対象外とします
  [RemoveIf(false, keywordNames: "FeatureA")]
  bool enableFeatureA;

  // trueの場合、キーワード “FeawtureB” を含むVariantのみをビルド対象にします
  [SelectIf(true, keywordNames: "FeatureB")]
  bool forceFeatureB;

  // OpenGLESで trueの場合、キーワード “FeatureC” を含むVariantのみをビルド対象にします
  [ApplyRulesIfNotGraphicsAPI(GraphicsDeviceType.OpenGLES3 )]
  [SelectIf(true, keywordNames: "FeatureC")]
  bool forceFeatureC;
}
```

Code example
IPreProcessShaders.OnProcessShaderの実装

```
1  // IPreprocessShadersを継承したクラスを定義します
2  public class MyShaderProcessor : IPreprocessShaders{
3      // IPreprocessShadersは複数出来るので、コールバック順を設定します
4      public int callbackOrder { get { return 0; } }
5
6      // ビルド時に呼び出されるコールバック
7      // 引数「IList<ShaderCompilerData> data」の中身を変更し、ビルドに含むShaderVariantを減らします
8      public void OnProcessShader(
9          Shader shader, ShaderSnippetData snippet, IList<ShaderCompilerData> data){
10         // ShaderKeyword _FEATURE_A _FEATURE_B を定義
11         var keyword1 = new ShaderKeyword("_FEATURE_A");
12         var keyword2 = new ShaderKeyword("_FEATURE_B");
13         // 全てのShaderVariant候補を確認します
14         for (int i = data.Count-1; i >=0 ; --i){
15             // _FEATURE_A _FEATURE_B の二つが有効なShaderVariantはビルドから除外します
16             if (data[i].shaderKeywordSet.IsEnabled( keyword1 ) &&
17                 data[i].shaderKeywordSet.IsEnabled( keyword2 )){
18                 // ログ書き出しと、削除処理
19                 var foundKeywordSet = string.Join(" ", data[i].shaderKeywordSet.GetShaderKeywords());
20                 Debug.Log("RemoveFromBuild " + foundKeywordSet);
21                 data.RemoveAt(i);
22             }
23         }
24     }
25 }
```

ShaderVariant Stripで実際にあった効果

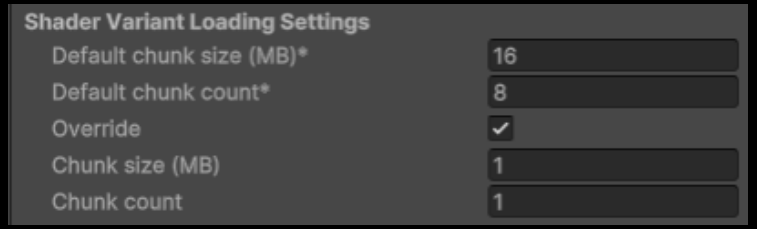
→ 特定のShaderが 30MB以上のメモリを使用していた
  - ShaderVariantを減らした事で、使用メモリ量が10MB以下になった
  - LitやUberなどのShaderは多くメモリを使用することが多いので効果が大きい

→ 初回のビルドに数時間掛かっていたプロジェクト
  - ShaderVariantを減らしたことで 初回ビルドが 1 時間以内で終わるようになった

Dynamic Shader Variant Loading

Dynamic Shader Variant Loading

- → PlayerSettingsのShaderVariantLoadingSettingsより設定可能
- → Shaderのデータは Chunkに分割して保存される
  - Chunk毎にLz4圧縮される
- → Shaderロード時に全てのChunkをLz4展開を行わない
  - 圧縮された状態でメモリにロードされるためメモリ使用量が減る
  - ChunkCountが0の時は全てのChunkを展開する



Platform毎に設定が可能

Chunkの詳細を知るには？

- → WebExtract.exe , binary2text.exeを利用してAssetBundleファイルなどビルドして作られたファイルをText化して確認する
- → 圧縮されたChunkサイズが compressedLengths に配列として入る
- → 展開後のChunkサイズが decompressedLengths に配列として入る
- → Platform/Shader実装によって異なるが、圧縮率が高いのがわかる
  - これだけに頼らず減らせるVariantは出来るだけ減らした方が良い

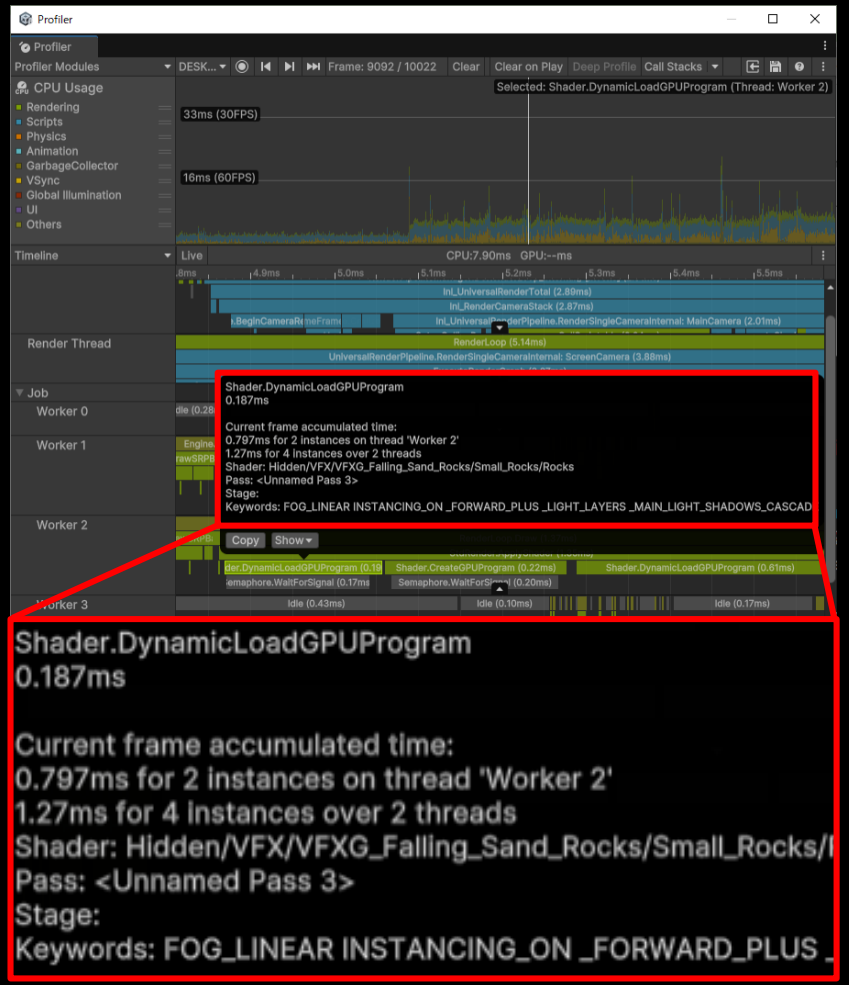
```
compressedLengths  (vector)
  size 1 (int)
    data  (vector)
      size 8 (int)
      data (unsigned int) #0: 7633 7745 17279 62708
      data (unsigned int) #4:  61389 76597 89272 87496

decompressedLengths  (vector)
  size 1 (int)
    data  (vector)
      size 8 (int)
        data (unsigned int) #0: 12112 524000 485568 514108
        data (unsigned int) #4: 510648 505596 504124 493404
```

DirectX11向けにビルドしたデータより抜粋

Chunkの展開による負荷

→ 展開されていないChunkにあるShaderVariantを利用しようとしたタイミングでChunkの展開処理が走る

- Shader.DynamicLoadGPUProgramがProfilerに負荷として出る
  - これがあるからと言って、必ず展開処理が走っているとは限らない
- ChunkCountを超えた数のChunkが展開されていた場合、LRUによってアクセスがもっとも古い展開済みChunkが破棄される



まとめ

- → Shader内でのKeyword分岐によって出来るのがShaderVariant
- → ShaderVariantを事前にWarmupする事で意図しないSpikeを回避
- → ShaderVariantを必要最小限にすることで、Build時間やRuntimeのメモリ使用量を減らせる
- → 全てを洗い出すのが困難ならば、DynamicShaderLoadingを利用してRuntimeのメモリ使用量を減らせる

Thank you


This is for a QR Code.
To replace it with your own:
1– Click this square
2– Click “Replace image” in toolbar above