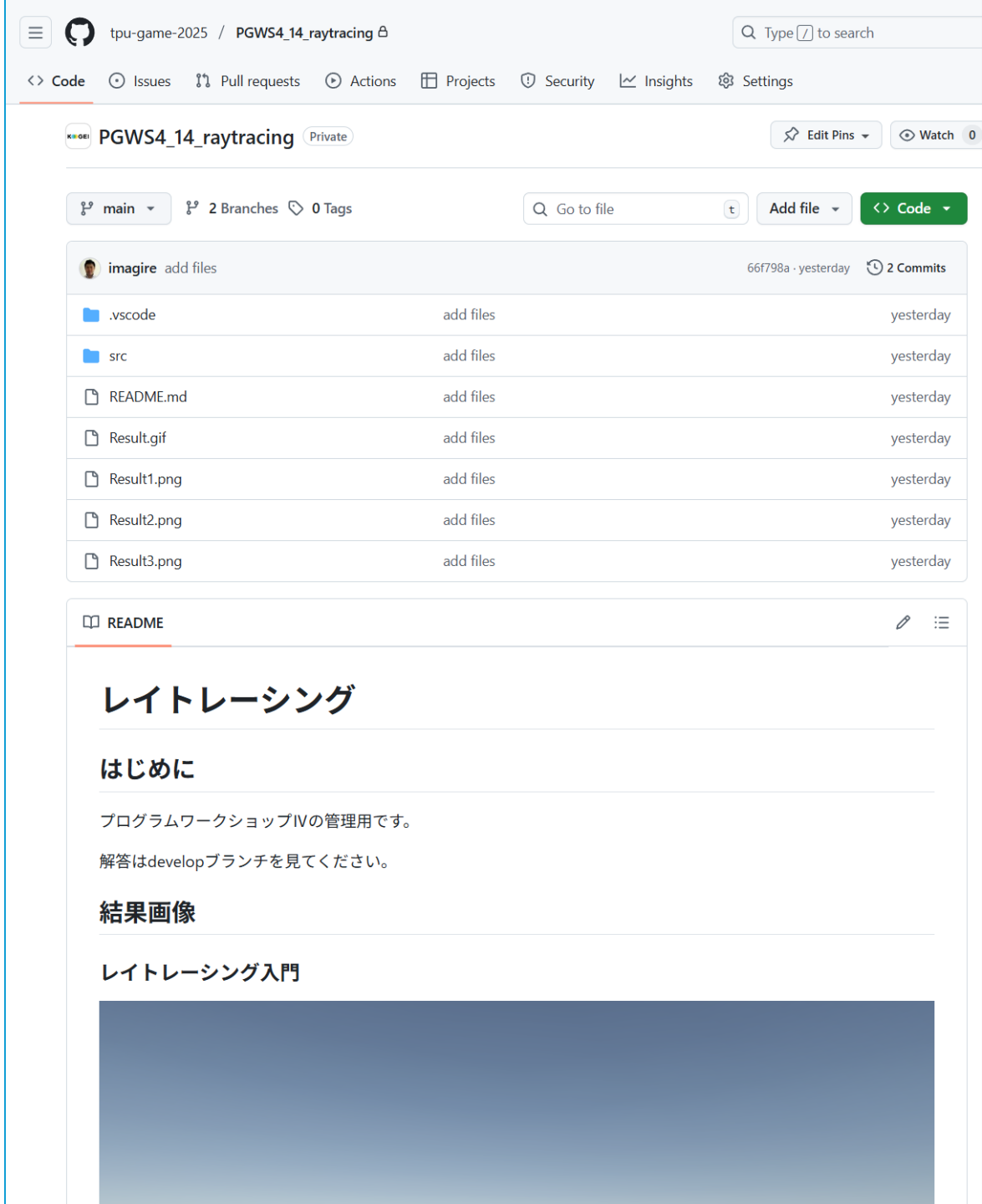
レイトレーシング

2025年度 プログラムワークショップIV（14）

今回のリポジトリ

- https://github.com/tpu-game-2025/PGWS4_14_raytracing



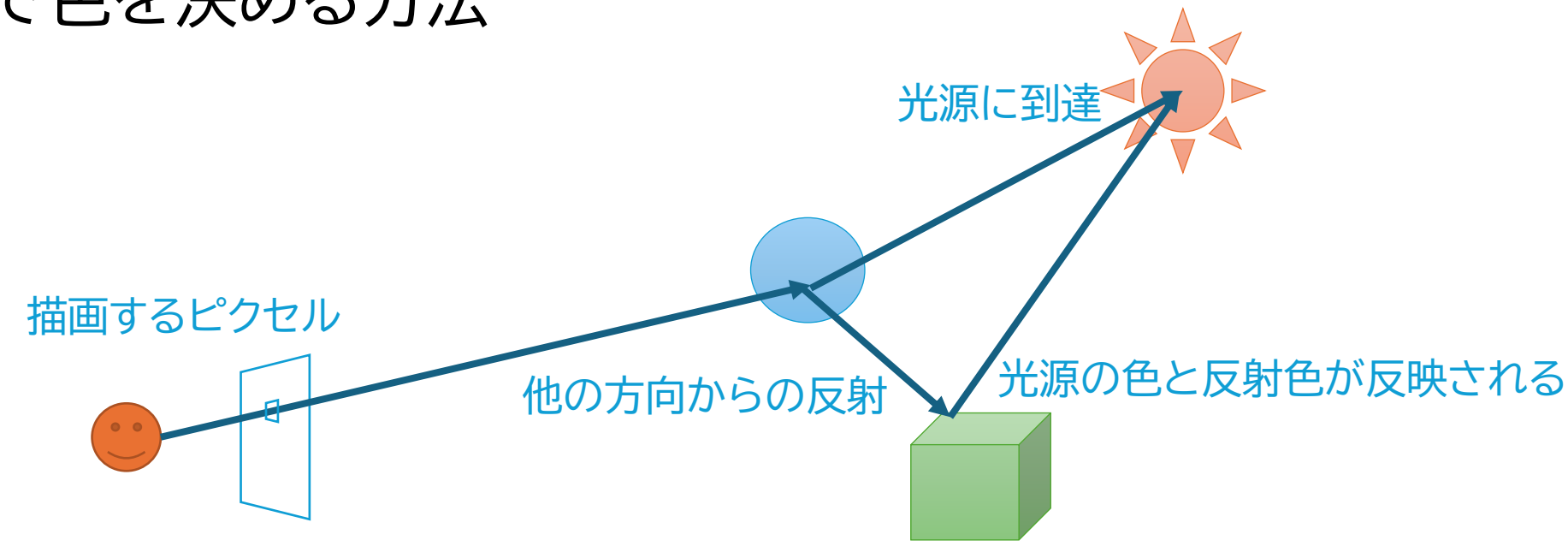
KOGEI TOKYO POLYTECHNIC UNIVERSITY

本日の内容

- レイトレーシング概要
- 簡単なレイトレーシング
- シーンの描画
- レイトレーシング影

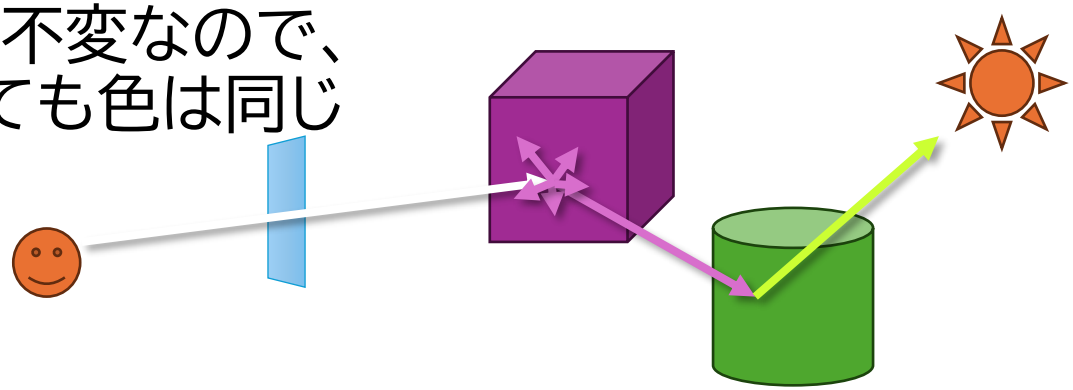
レイトレーシング

- 各画素の色を視点から仮想スクリーンの各画素へのレイ(線分)を追跡することで色を決める方法



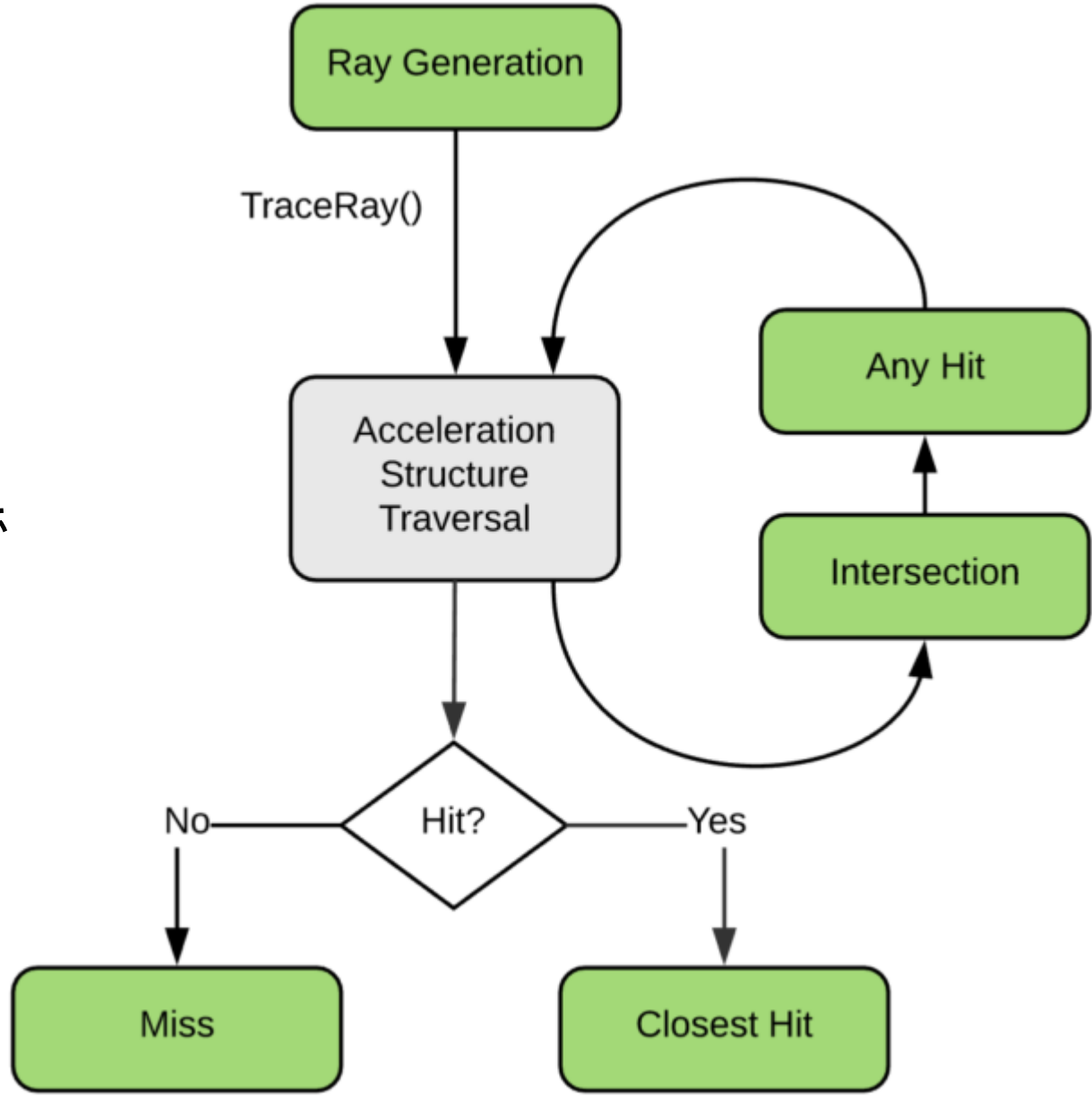
レイトレーシング

- 視点からレイ(光の進む線)を逆追跡
  - 光源にたどり取りついたらその色を参照
  - 光源にたどり着く前の物体の反射で光の色の変化を記録
    - 光の反射方向は表面の半球方向に反射
      - 膨大な数のレイが必要になるので、効率化の研究が進められている
- 光源から光を出しても良いが、各画素に到着するとは限らないので、視点から追跡する
  - (この範囲での)物理法則は時間反転不変なので、光源から追跡しても視点から追跡しても色は同じ



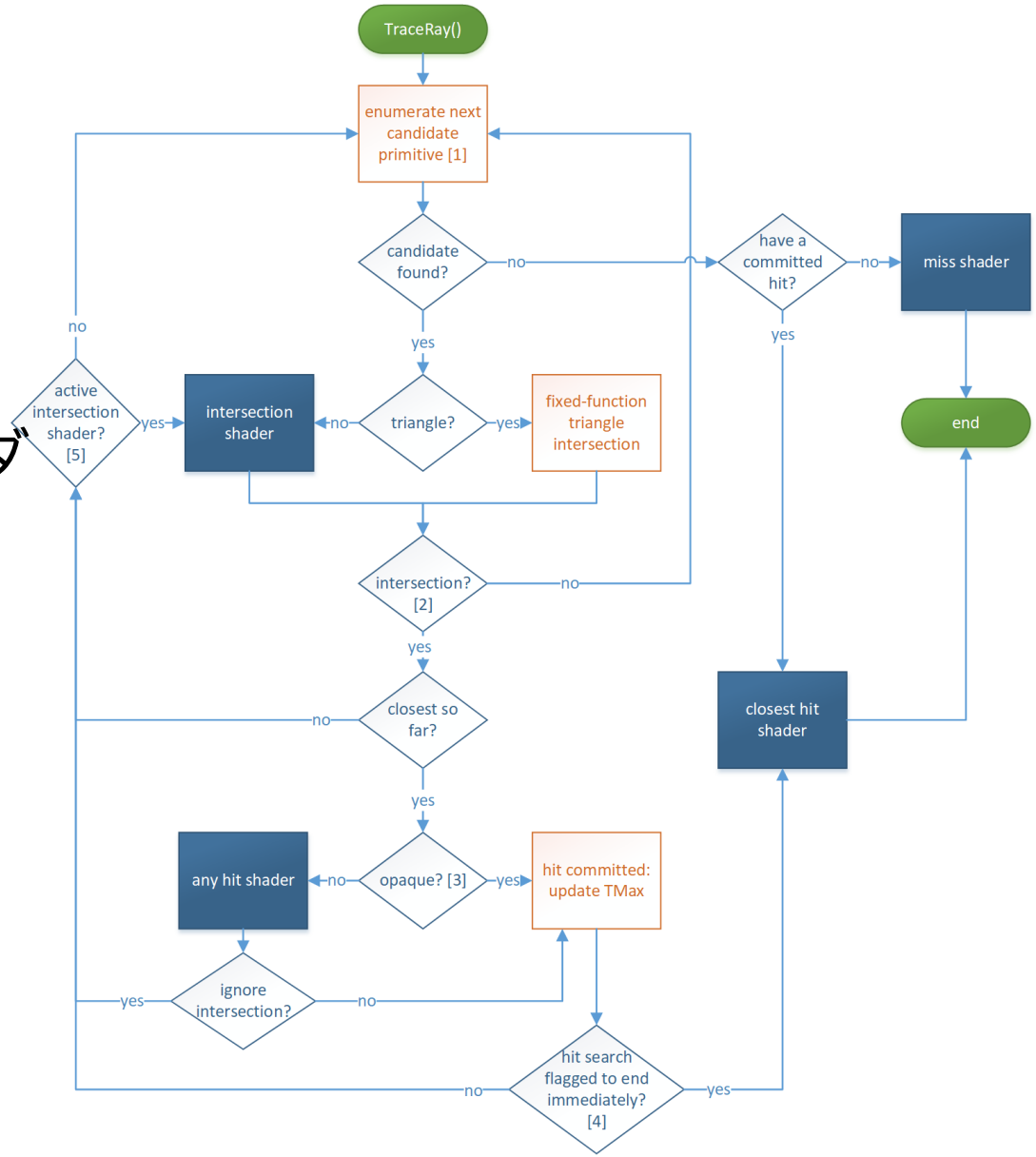
ハードウェア レイトレーシング

- GPUを使ってリアルタイムにレイトレーシングを実現
- パイプラインを変更し、シェーダーの組み合わせと追加の固定機能で実現
  - 変更できない処理がある
  - 交差判定は隠蔽されていて、変更できない
    - BVH(Bounding Volume Hierarchy)が使われているはず
      - 階層化されたAABBのグルーピング
- 複数のシェーダを利用



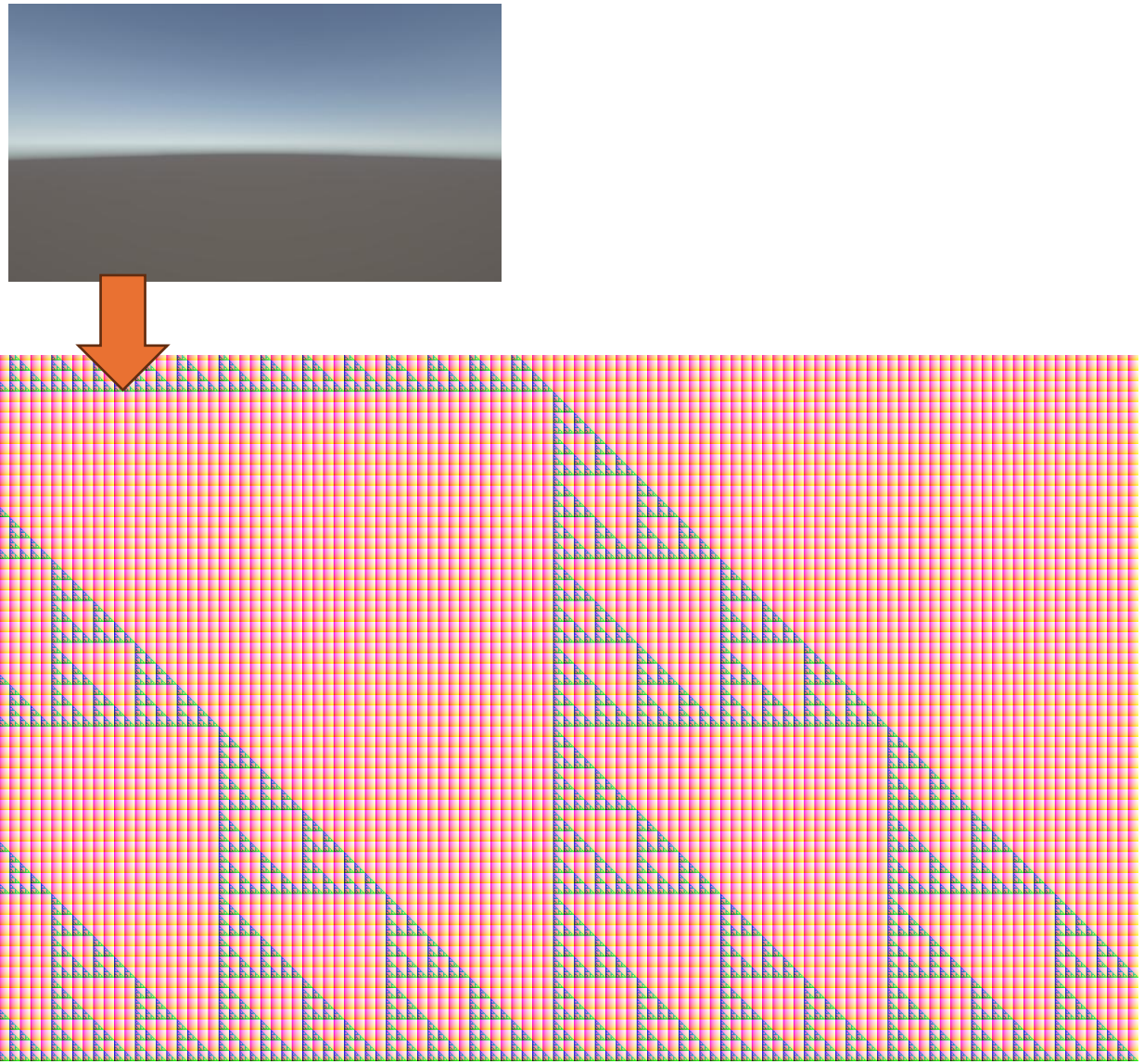
シェーダ

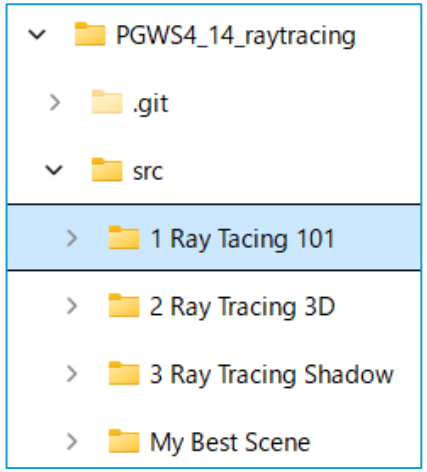
- Ray Generation Shader: レイ生成シェーダ
  - 各画素からレイを飛ばす
- Closest Hit Shader: 最近傍衝突シェーダ
  - 一番近い衝突点での処理
- Any Hit Shader: 任意衝突シェーダー
  - いずれかに衝突した際の処理
- Miss Shader: 失敗シェーダ
  - 衝突が起きなかった場合の処理
- Intersection Shader: 交差シェーダ
  - 三角形以外の衝突判定
- Callable Shader: 呼び出し可能シェーダ
  - 関数呼び出しとして機能する



本日の内容

- レイトレーシング概要
- 簡単なレイトレーシング
- シーンの描画
- レイトレーシング影





「1 Ray Tracing 101」ディレクトリ

参考

- https://blog.siliconstudio.co.jp/2024/12/1923/
- 今回のシェーダは、自動生成されたコードから改変
- Unityの仕様（?）が微妙に更新されている

Silicon Studio
Ideas × Art × Technology

Search ...

Unity6 + URP で動くパストレーサーを実装してみよう Part 1

By 川口 | 2024年12月12日

レイトレーシング

Unity, レイトレーシング, レンダリング

はじめに

こんにちは。研究開発室の川口です。
突然ですが皆さんはレイトレーシングを実装したいときにどんな環境を利用しますか？

C/C++ や Rust を使った CPU で動作するシンプルな実装はとても大切ですし、OptiX や最近登場した HIPRT を使った GPU で動作する実践的な実装も非常に興味深いです。
DXR や Vulkan Ray Tracing を使ってゲームに応用できるような実装を追求しても良いですし、お手軽に Shadertoy 上で動作する表現に挑戦するのも楽しいです。

そんな色々なレイトレの実装環境の中でもちょっと特殊といえる、 Unity 上で GPU レイトレを行う話を何回かにわけて書いていきたいと思います。
理論や設計などには深く言及せず、ざっくりとレイトレの実装例としてパストレーサーを作っていく過程を紹介していきます。
最終的には ReSTIR のような実践的なアルゴリズムを実装するところまで紹介できると良いなと考えています（が予定は未定です）。

対象読者：Unity、URP に触ったことのある方・GPU レイトレの実装に興味のある方

Unity6 + URP で動くパストレーサーを実装してみよ

人気記事

batファイルのあれこれ(コマンド編)

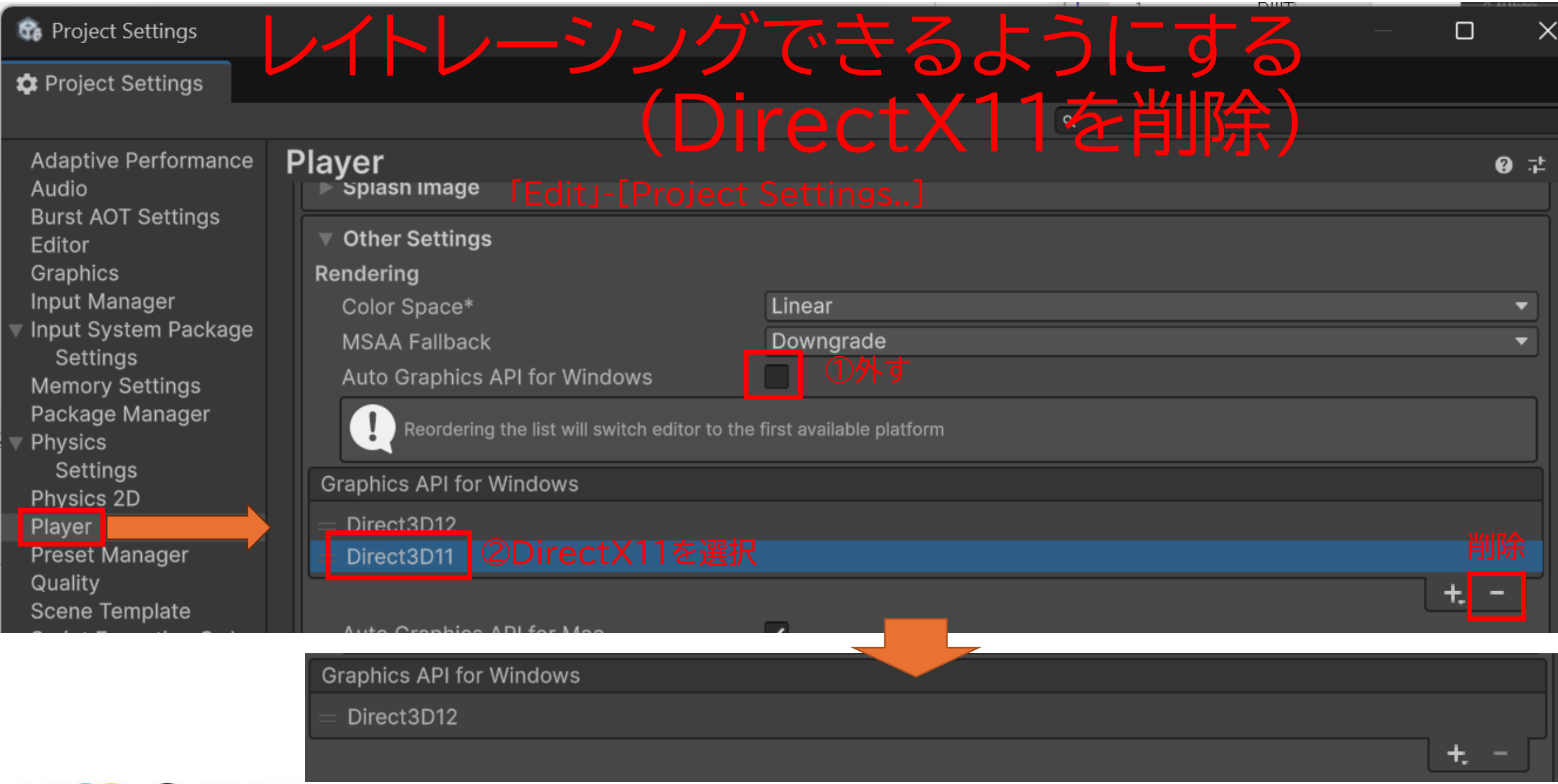
Unreal Engine 揺れもの物理プラグイン検証：Kawaii Physics & SPCR Joint Dynamics

球面調和関数とCG. Part1, ~Light Probeは何を計算しているのか？~

QGISで数値樹冠高モデルから樹頂点を抽出してみよう Part2

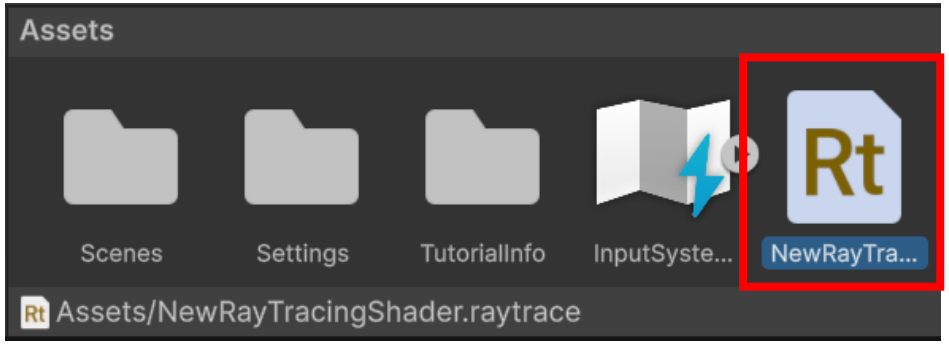
Slang で自動微分シェーダ開

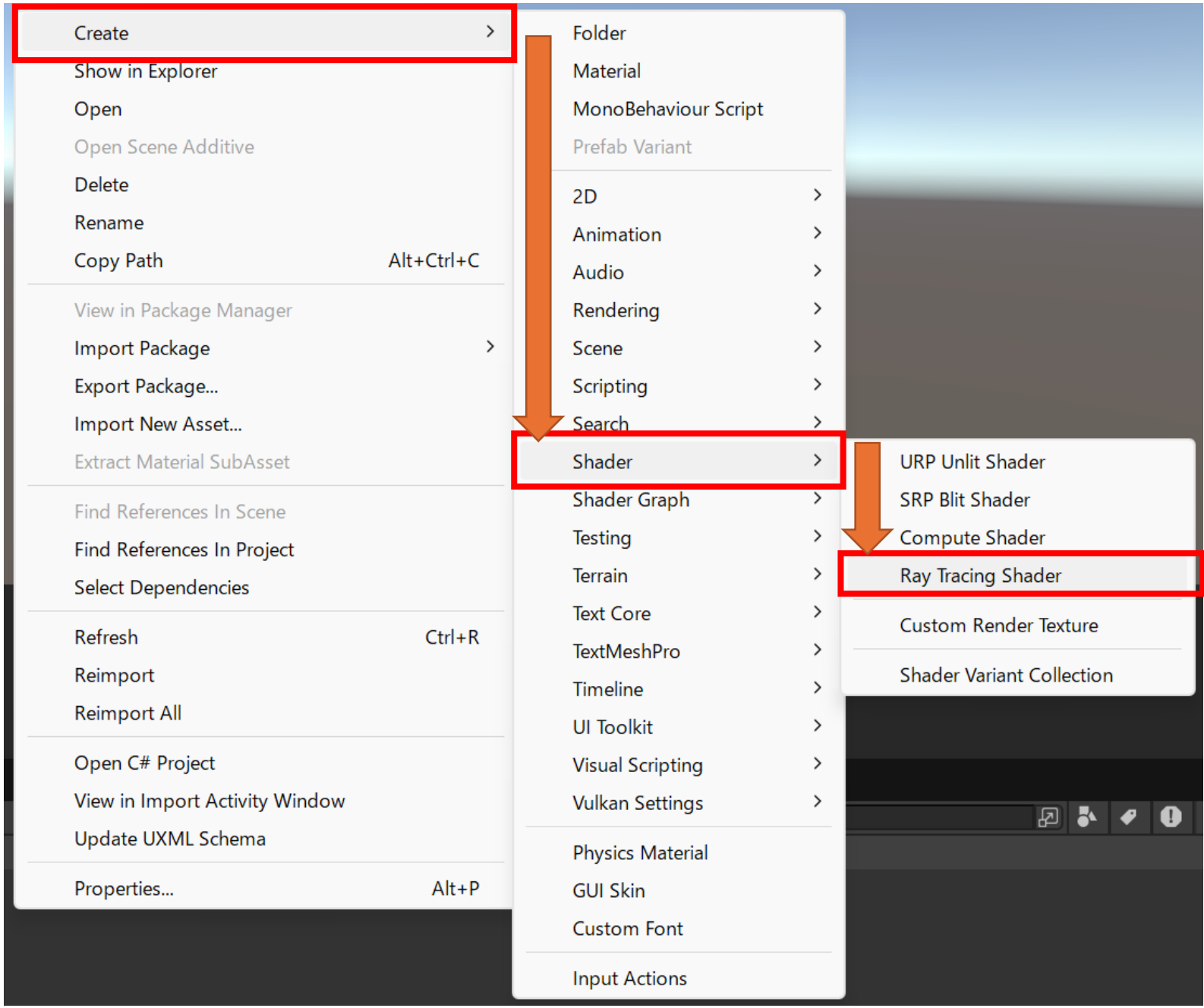
レイトレーシングできるようにする（DirectX11を削除）



シェーダの追加

- 「Ray Tracing Shader」
  - 名称例: NewRayTracingShader
    - デフォルト名





デフォルトのシェーダーコード

- レイ生成シェーダのみ
  - 画面分のレイを生成して飛ばす
  - 飛ばした後の処理を書いていないので、生成しただけで終わる
- 「RenderTarget」という名前のテクスチャに出力
  - DispatchRaysIndex()で各レイを区別

NewRayTracingShader.raytrace

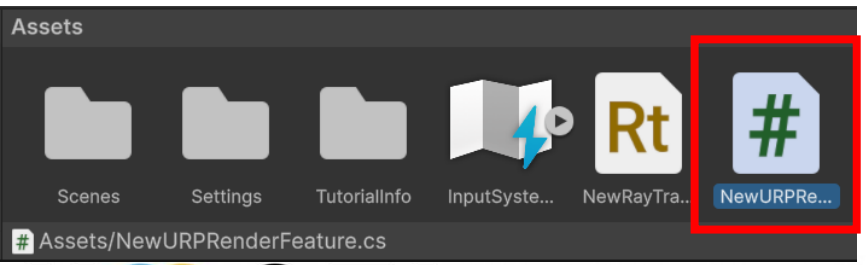
```
1   RWTexture2D<float4> RenderTarget;
2
3   // Uncomment this pragma for debugging the HLSL code in PIX. GPU performance will be impacted.
4   //#pragma enable_ray_tracing_shader_debug_symbols
5
6   #pragma max_recursion_depth 1
7
8   [shader("raygeneration")]
9   void MyRaygenShader()
10  {
11      uint2 dispatchIdx = DispatchRaysIndex().xy;
12
13      RenderTarget[dispatchIdx] = float4(dispatchIdx.x & dispatchIdx.y, (dispatchIdx.x & 15)/15.0, (dispatchIdx.y & 15)/15.0, 0.0);
14  }
```

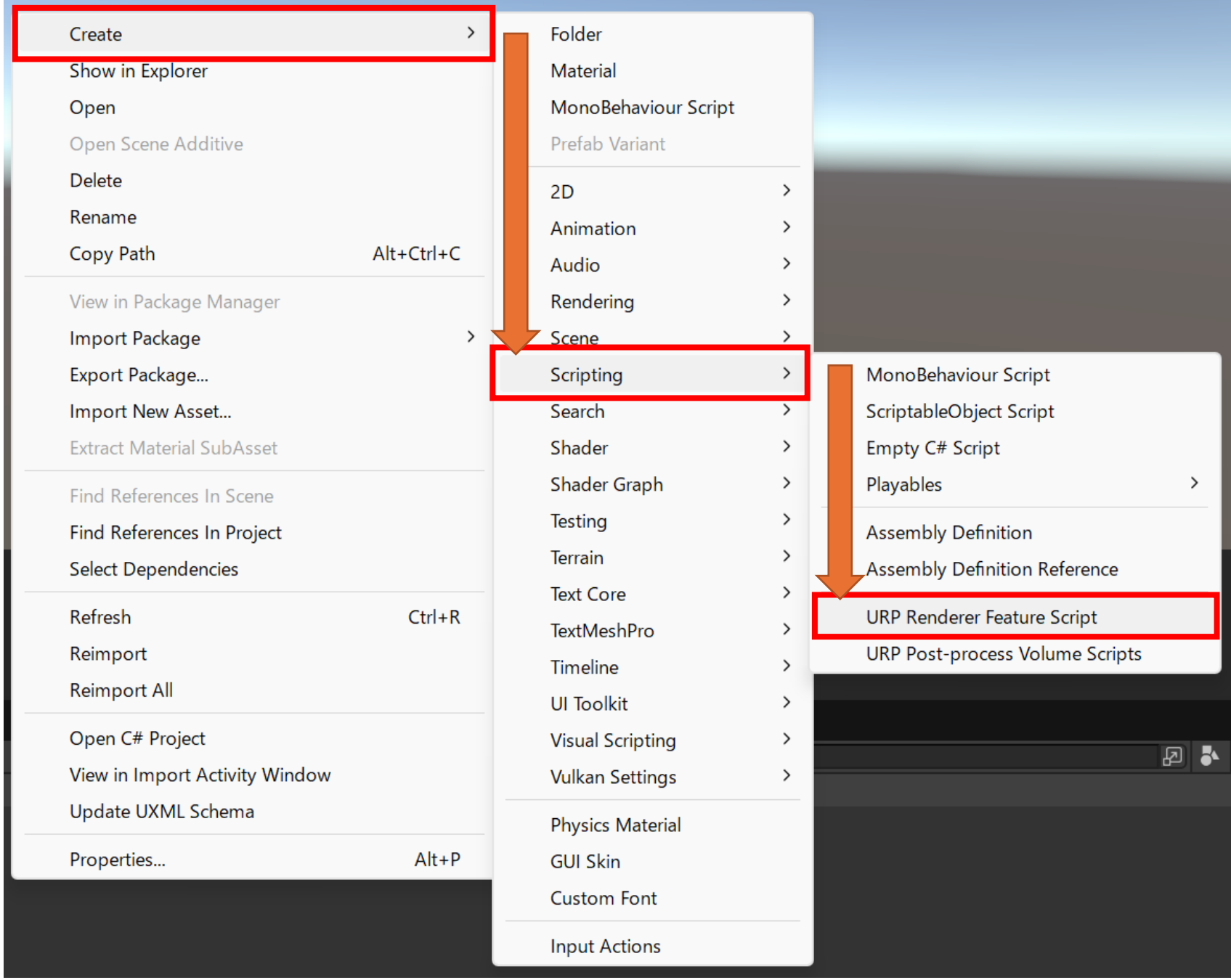
レンダラー

- 描画処理をするモジュール
- パイプライン
  - 影マップの生成・影マップの反映や、半透明・不透明で処理を分けたい
- レンダラーデータ
  - 各シェーダで描画するオブジェクトを選択できる
- レンダラーフィーチャー（機能）
  - マルチパス等を一つにまとめる
    - キューブマップの6面の描画を一つとみなせる存在

描画機能スクリプト

- レイトレーシングはラスタライズではないので、大きく描画の機能を変更する必要がある





生成コード

- 修正は後から

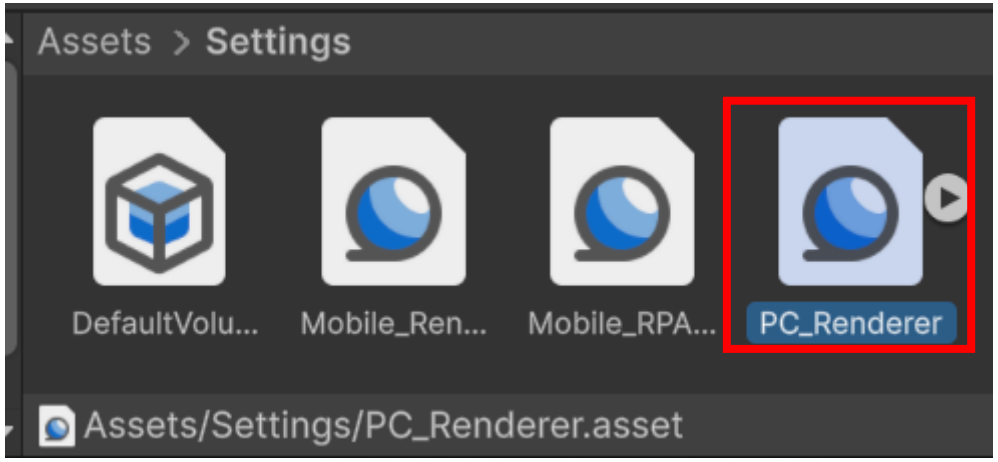
```
1   using System;
2   using UnityEngine;
3   using UnityEngine.Rendering;
4   using UnityEngine.Rendering.Universal;
5   using UnityEngine.Rendering.RenderGraphModule;
6
    Unity スクリプト | 1 個の参照
7   public class NewURPRenderFeature : ScriptableRendererFeature
8   {
9       [SerializeField] NewURPRenderFeatureSettings settings;
10      NewURPRenderFeaturePass m_ScriptablePass;
11
12      /// <inheritdoc/>
        0 個の参照
13      public override void Create()...
30
31      // Here you can inject one or multiple render passes in the renderer.
32      // This method is called when setting up the renderer once per-camera.
        0 個の参照
33      public override void AddRenderPasses(ScriptableRenderer renderer, ref RenderingData renderingData)...
37
38      // Use this class to pass around settings from the feature to the pass
39      [Serializable]
        7 個の参照
40      public class NewURPRenderFeatureSettings...
44
        7 個の参照
45      class NewURPRenderFeaturePass : ScriptableRenderPass
46      {
47          readonly NewURPRenderFeatureSettings settings;
48
            2 個の参照
49          public NewURPRenderFeaturePass(NewURPRenderFeatureSettings settings)...
53
54          // This class stores the data needed by the RenderGraph pass.
55          // It is passed as a parameter to the delegate function that executes the RenderGraph pass.
            7 個の参照
56          private class PassData...
60
61          // This static method is passed as the RenderFunc delegate to the RenderGraph render pass.
62          // It is used to execute draw commands.
            1 個の参照
63          static void ExecutePass(PassData data, RasterGraphContext context)...
67
68          // RecordRenderGraph is where the RenderGraph handle can be accessed, through which render passes can be added to the graph.
69          // FrameData is a context container through which URP resources can be accessed and managed.
            0 個の参照
70          public override void RecordRenderGraph(RenderGraph renderGraph, ContextContainer frameData)...
97      }
98  }
```

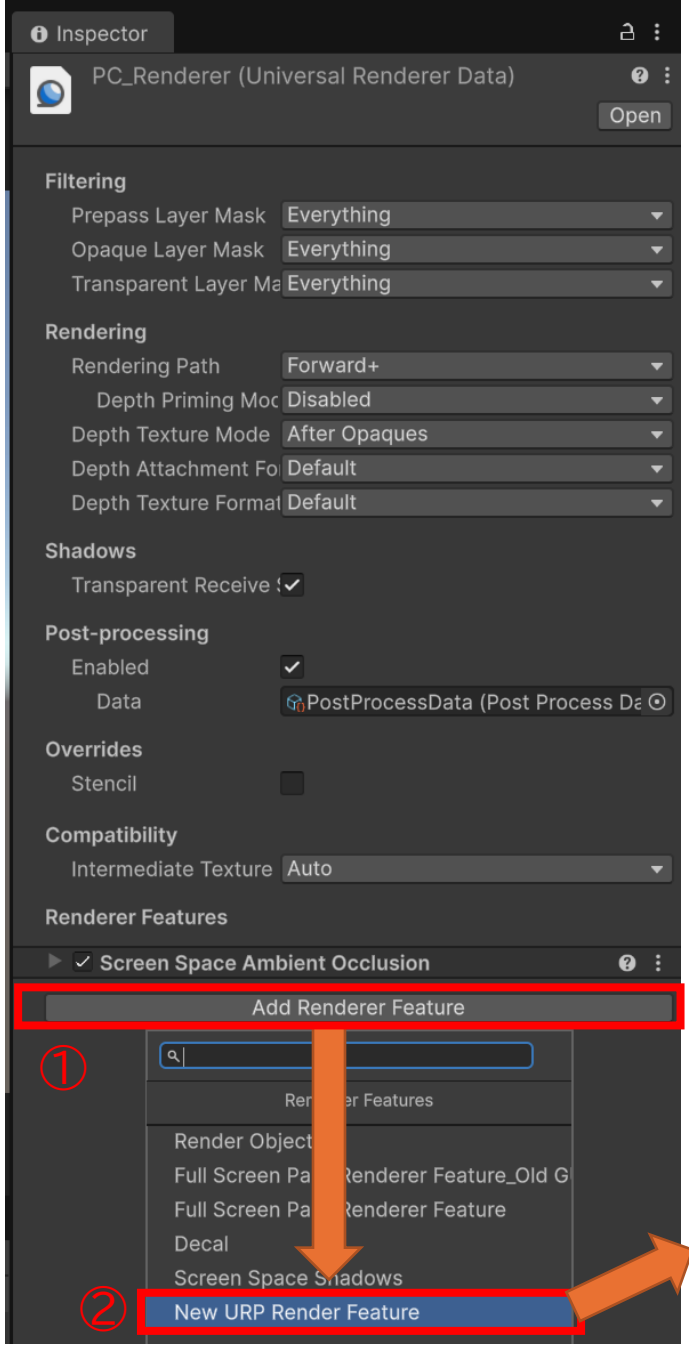
パスに渡す初期化情報

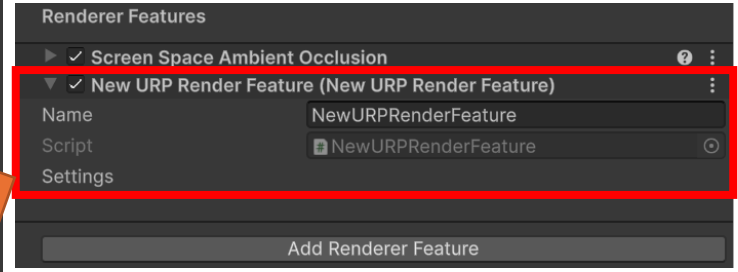
各パスの描画処理の設定

レンダラーデータに要素を追加

- 「PC_Renderer」に追加
  1. Add Renderer Featureを選択
  2. (作成した)「New URP Render Feature」を追加







レンダラー要素の編集

- NewURPRenderFeature::NewURPRenderFeaturePass::ExecutePassで描画(DispatchRays)の実行
  - 引数をRasterGraphCotextからUnsafeGraphContextに変更
- 2種類のコマンドバッファ
  - native_cmd
    - CommandBufferクラス
    - 描画用
  - context.cmd
    - UnsafeCommandBufferクラス
    - GPUへの低レベルアクセス
- テクスチャに描画し、その結果をレンダーテクスチャに転送
  - 描画先:data.output_ColorTexture
  - レンダーテクスチャ:data.camera_ColorTarget

```
109             // This static method is passed as the RenderFunc delegate to the RenderGraph render pass.
110             // It is used to execute draw commands.
111             // ■この静的メソッドは、RenderGraphレンダーパスにRenderFuncデリゲートとして渡されます。
112             // ■描画コマンドを実行するために使用されます.
113         //  static void ExecutePass(PassData data, RasterGraphContext context)
                1 個の参照
114             static void ExecutePass(PassData data, UnsafeGraphContext context)// ■差し替え
115             {
116                 // ■ 追加
117                 // UnsafeGraphContext からネイティブの CommandBuffer （GPUの命令を溜めるバッファ） を取得
118                 CommandBuffer native_cmd = CommandBufferHelpers.GetNativeCommandBuffer(context.cmd);
119                 // レイトレシェーダーパスを設定
120                 native_cmd.SetRayTracingShaderPass(data.rayTracingShader, "NewURPRenderFeaturePass");
121                 // レイトレシェーダーに出力テクスチャを設定
122                 context.cmd.SetRayTracingTextureParam(data.rayTracingShader,
123                     Shader.PropertyToID("RenderTarget"), data.output_ColorTexture);
124                 // レイトレを実行
125                 context.cmd.DispatchRays(data.rayTracingShader, "MyRaygenShader",
126                     (uint)data.camera.pixelWidth, (uint)data.camera.pixelHeight, 1, data.camera);
127                 // 結果をカメラに書き戻す
128                 native_cmd.Blit(data.output_ColorTexture, data.camera_ColorTarget);
129             }
```

Bit-Blt (Bitmap Bit Transfer)の略

NewURPRenderFeature.cs

Execute passの呼び出し

- NewURPRenderFeaturePass::RecordRenderGraphに記述
- builder(IUnsafeRenderGraphBuilder)で描画関数として設定
-

```
131             // RecordRenderGraph is where the RenderGraph handle can be accessed, through which render passes can be added to the graph.
132             // FrameData is a context container through which URP resources can be accessed and managed.
133             // ■RecordRenderGraphは、RenderGraphハンドルにアクセスできる場所であり、これを通じてレンダーパスをグラフに追加できます。
134             // ■FrameDataは、URPリソースにアクセスおよび管理できるコンテキストコンテナです。
                0 個の参照
135             public override void RecordRenderGraph(RenderGraph renderGraph, ContextContainer frameData)
136             {
137                 const string passName = "Render Custom Pass";
                ⋮
166             // This adds a raster render pass to the graph, specifying the name and the data type that will be passed to the ExecutePass function.
167             // ■これにより、グラフにラスターレンダーパスが追加され、名前とExecutePass関数に渡されるデータタイプが指定されます。
168         //  using (var builder = renderGraph.AddRasterRenderPass<PassData>(passName, out var passData))
169             using (var builder = renderGraph.AddUnsafePass<PassData>(passName, out var passData))// ■差し替え
170             {
171                 // Use this scope to set the required inputs and outputs of the pass and to
172                 // setup the passData with the required properties needed at pass execution time.
173                 // ■このスコープを使用して、パスの必要な入力と出力を設定し、
174                 // ■パス実行時に必要なプロパティでpassDataを設定します。
175
176                 // ■ 追加
177                 passData.rayTracingShader = rayTracingShader;
178                 passData.output_ColorTexture = resultTex;
179                 passData.camera_ColorTarget = colorTexture;
180                 passData.rayTracingAccelerationStructure = rayTracingAccelerationStructure;
181                 passData.camera = cameraData.camera;
182                 builder.UseTexture(passData.output_ColorTexture, AccessFlags.Write);
183                 builder.UseTexture(passData.camera_ColorTarget, AccessFlags.ReadWrite);
184
185                 // Make use of frameData to access resources and camera data through the dedicated containers.
186                 // Eg:
187                 // ■frameDataを利用して、専用のコンテナを通じてリソースとカメラデータにアクセスします。
188                 // ■例：
189                 // UniversalCameraData cameraData = frameData.Get<UniversalCameraData>();
190
191         //      UniversalResourceData resourceData = frameData.Get<UniversalResourceData>(); // ■ 削除: 上に移動
192
193                 // Setup pass inputs and outputs through the builder interface.
194                 // Eg:
195                 // ■builderインターフェースを通じてパスの入力と出力を設定します。
196                 // ■例：
197                 // builder.UseTexture(sourceTexture);
198                 // TextureHandle destination = UniversalRenderer.CreateRenderGraphTexture(renderGraph, cameraData.cameraTargetDescriptor, "Destinat
199
200                 // This sets the render target of the pass to the active color texture. Change it to your own render target as needed.
201                 // ■パスのレンダーターゲットがアクティブなカラーテクスチャに設定されます。必要に応じて独自のレンダーターゲットに変更してください。
202         //      builder.SetRenderAttachment(resourceData.activeColorTexture, 0);// ■差し替え：不要
203
204                 // Assigns the ExecutePass function to the render pass delegate. This will be called by the render graph when executing the pass.
205                 // ■レンダーパスデリゲートにExecutePass関数を割り当てます。これは、パスを実行する際にレンダーグラフによって呼び出されます。
206         //      builder.SetRenderFunc((PassData data, RasterGraphContext context) => ExecutePass(data, context));// ■差し替え
207                 builder.SetRenderFunc((PassData data, UnsafeGraphContext context) => ExecutePass(data, context));
208             }
209         }
```

builderを自動破棄するためのusingステートメント
（引数の型は変更）

ExecutePassに渡すデータ(後述)を設定する
- ExecutePassで使っている変数
- 使い方を含めたテクスチャの利用の宣言

パスデータ

- NewURPRenderFeaturePass::PassData
- ExecutePassで使うデータ

```
95          // This class stores the data needed by the RenderGraph pass.
96          // It is passed as a parameter to the delegate function that executes the RenderGraph pass.
97          // ■このクラスは、RenderGraphパスに必要なデータを格納します。
98          // ■RenderGraphパスを実行するデリゲート関数にパラメーターとして渡されます。
            3 個の参照
99          private class PassData
100         {
101             public RayTracingShader rayTracingShader;// ■ 追加: レイトレシェーダー
102             public TextureHandle output_ColorTexture;// ■ 追加: レイトレ結果を書き出すテクスチャ
103             public TextureHandle camera_ColorTarget;// ■ 追加: カメラのカラーテクスチャ
104             public RayTracingAccelerationStructure rayTracingAccelerationStructure;// ■ 追加: AS
105                                                                                    加速構造:シーンのBVH。今回は使わない
106             public Camera camera;// ■ 追加: カメラ
107         }
```

パスデータで渡すデータの構築

- Builder作成前の処理

```
131             // RecordRenderGraph is where the RenderGraph handle can be accessed, through which render passes can be added to the graph.
132             // FrameData is a context container through which URP resources can be accessed and managed.
133             // ■RecordRenderGraphは、RenderGraphハンドルにアクセスできる場所であり、これを通じてレンダーパスをグラフに追加できます。
134             // ■FrameDataは、URPリソースにアクセスおよび管理できるコンテキストコンテナです。
                0 個の参照
135             public override void RecordRenderGraph(RenderGraph renderGraph, ContextContainer frameData)
136             {
137                 const string passName = "Render Custom Pass";
138
139                 // ■ 追加
140                 UniversalResourceData resourceData = frameData.Get<UniversalResourceData>();
141                 UniversalCameraData cameraData = frameData.Get<UniversalCameraData>();
142
143                 // ■ 追加: 現在のカメラで描画されたカラーフレームバッファを取得
144                 var colorTexture = resourceData.activeColorTexture;
145
146                 // ■ 追加: レイトレ結果を描き出すバッファを作成
147                 RenderTextureDescriptor rtdesc = cameraData.cameraTargetDescriptor;
148                 rtdesc.graphicsFormat = GraphicsFormat.R16G16B16A16_SFloat;
149                 rtdesc.depthStencilFormat = GraphicsFormat.None;
150                 rtdesc.depthBufferBits = 0;
151                 rtdesc.enableRandomWrite = true;
152                 var resultTex = UniversalRenderer.CreateRenderGraphTexture(renderGraph, rtdesc, "_RayTracedColor", false);
153
154                 // ■ 追加: Acceleration Structure を作成
155                 if (rayTracingAccelerationStructure == null)
156                 {
157                     var settings = new RayTracingAccelerationStructure.Settings();
158                     settings.rayTracingModeMask = RayTracingAccelerationStructure.RayTracingModeMask.Everything;
159                     settings.managementMode = RayTracingAccelerationStructure.ManagementMode.Automatic;
160                     settings.layerMask = 255;
161                     rayTracingAccelerationStructure = new RayTracingAccelerationStructure(settings);
162
163                     rayTracingAccelerationStructure.Build();// 今回は静的に構築
164                 }
```

関係するデータをフレーデータから取得

最終的に画面を作成する先

描画するレンダーテクスチャをここで作成

加速構造は「Build」メソッドで計算(軽くないイメージ)

NewURPRenderFeaturePassのメンバーや初期化・解放

- シェーダはコンストラクタで受け取る
  - FeaturePassは直接は、Inspectorに表示されない
    - 表示されるRenderer Feature Scriptから受け取る
- 加速構造は、自動で解放されないので、解放関数を作成
  - 外部から呼ばれる(後述)

```
72          class NewURPRenderFeaturePass : ScriptableRenderPass
73          {
74              readonly NewURPRenderFeatureSettings settings;
75
76              RayTracingShader rayTracingShader;// ■ 追加
77              RayTracingAccelerationStructure rayTracingAccelerationStructure;// ■ 追加
78
79
                1 個の参照
80              public NewURPRenderFeaturePass(NewURPRenderFeatureSettings settings)
81              {
82                  this.settings = settings;
83
84                  base.profilingSampler = new ProfilingSampler("NewURPRenderFeaturePass");
85                  rayTracingShader = settings.rayTracingShader;// ■ 追加
86              }
87
88              // ■ 追加
                1 個の参照
89              public void Cleanup()
90              {
91                  rayTracingAccelerationStructure?.Dispose();
92              }
```

RendererFeatureクラス

- 描画イベントのデフォルトからの変更
  - 不透明後ではなくポスト処理直前
- 解放関数の作成
  - 加速構造の解放

```
9           public class NewURPRenderFeature : ScriptableRendererFeature
10          {
11              [SerializeField] NewURPRenderFeatureSettings settings;
12              NewURPRenderFeaturePass m_ScriptablePass;
13
14              /// <inheritdoc/>
                0 個の参照
15              public override void Create()
16              {
17                  m_ScriptablePass = new NewURPRenderFeaturePass(settings);
18
19                  // Configures where the render pass should be injected.
20                  // ■レンダリングパスを挿入する場所を設定
21          //      m_ScriptablePass.renderPassEvent = RenderPassEvent.AfterRenderingOpaques; // ■差し替え
22                  m_ScriptablePass.renderPassEvent = RenderPassEvent.BeforeRenderingPostProcessing;
23
24                  // You can request URP color texture and depth buffer as inputs by uncommenting the line below,
25                  // URP will ensure copies of these resources are available for sampling before executing the render pass.
26                  // Only uncomment it if necessary, it will have a performance impact, especially on mobiles and other TBDR GPUs where it will break render pas
27                  // ■以下の行のコメントを外すことで、URPカラーテクスチャと深度バッファを入力として要求できます。
28                  // ■URPは、レンダリングパスを実行する前に、これらのリソースのコピーがサンプリング可能であることを保証します。
29                  // ■必要な場合のみコメントを外してください。特にモバイル端末やTBDR GPUではパフォーマンスに影響し、レンダリングパスが破綻する可能性があります。
30                  //m_ScriptablePass.ConfigureInput(ScriptableRenderPassInput.Color | ScriptableRenderPassInput.Depth);
31
32                  // You can request URP to render to an intermediate texture by uncommenting the line below.
33                  // Use this option for passes that do not support rendering directly to the backbuffer.
34                  // Only uncomment it if necessary, it will have a performance impact, especially on mobiles and other TBDR GPUs where it will break render pas
35                  // ■以下の行のコメントを外すことで、URPに中間テクスチャへのレンダリングを要求できます。
36                  // ■バックバッファへの直接レンダリングをサポートしないパスにこのオプションを使用してください。
37                  // ■必要な場合のみコメントを外してください。特にモバイル端末やTBDR GPUではパフォーマンスに影響し、レンダリングパスが破綻する可能性があります。
38                  //m_ScriptablePass.requiresIntermediateTexture = true;
39              }
40
41              // ■ 追加(ASの削除の為)
                0 個の参照
42              protected override void Dispose(bool disposing)
43              {
44                  if (disposing)
45                  {
46                      m_ScriptablePass?.Cleanup();
47                      m_ScriptablePass = null;
48                  }
49              }
```

パスの追加

- NewURPRenderFeature::AddRenderPass
- 表示するウィンドウを指定できる
  - ゲームで実行される画面にのみ描画

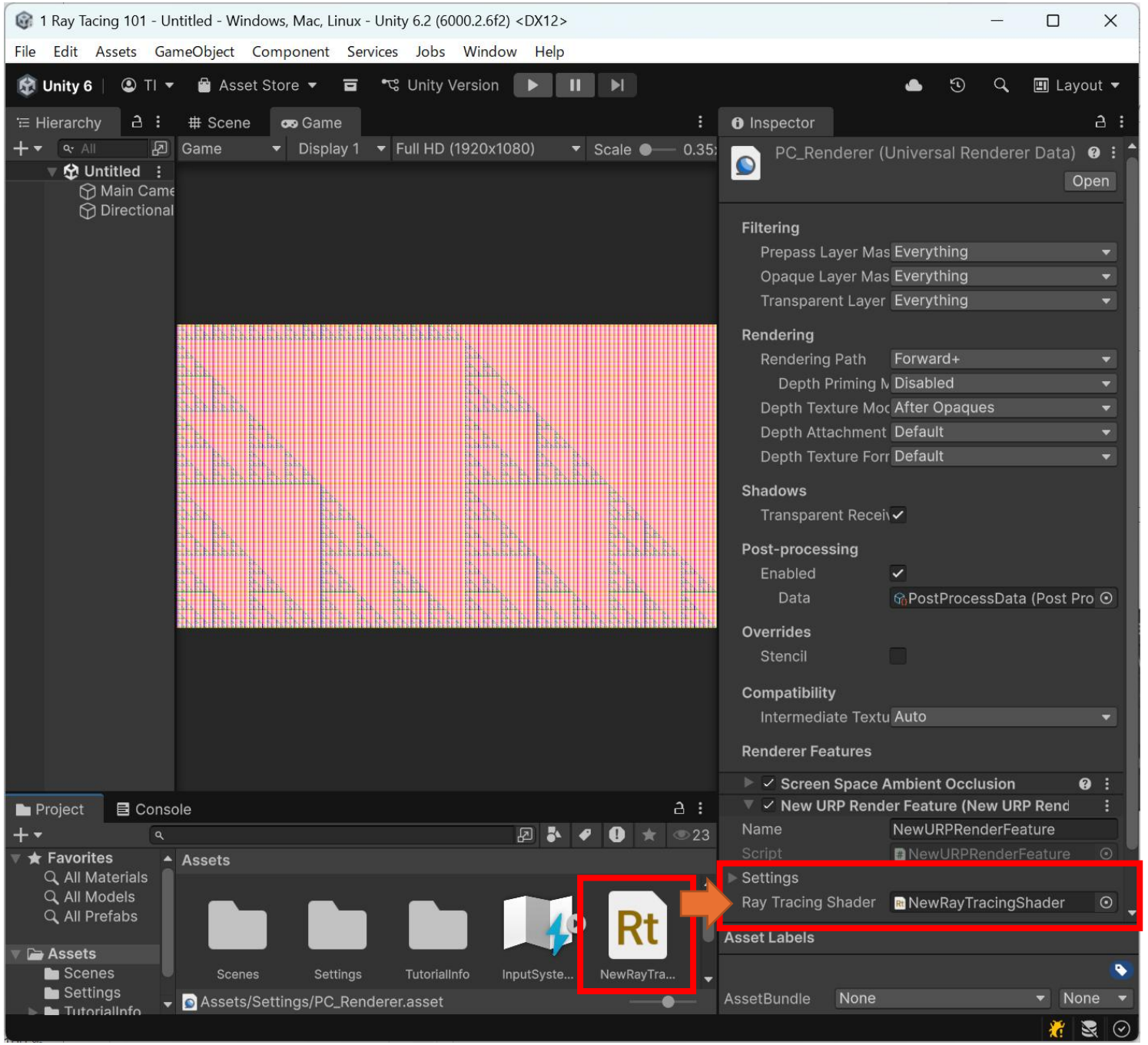
```
52          // Here you can inject one or multiple render passes in the renderer.
53          // This method is called when setting up the renderer once per-camera.
54          // ■ここで、レンダラーに1つまたは複数のレンダリングパスを挿入できます。
55          // ■このメソッドは、カメラごとに1回、レンダラーを設定する際に呼び出されます。
            0 個の参照
56          public override void AddRenderPasses(ScriptableRenderer renderer, ref RenderingData renderingData)
57          {
58              // ■追加: シーンビューとプレビューでは実行しない
59              if (renderingData.cameraData.isSceneViewCamera || renderingData.cameraData.isPreviewCamera) return;
60
61              renderer.EnqueuePass(m_ScriptablePass);
62          }
```

レンダーパスに渡すデータ

- NewURPRenderFeature::
  NewURPRenderFeatureSettings
- シェーダの設定

```
64          // Use this class to pass around settings from the feature to the pass
65          // ■このクラスを使用して、フィーチャーからパスに設定を渡します
66          [Serializable]
            3 個の参照
67          public class NewURPRenderFeatureSettings
68          {
69              public RayTracingShader rayTracingShader = default!;// ■追加
70          }
```

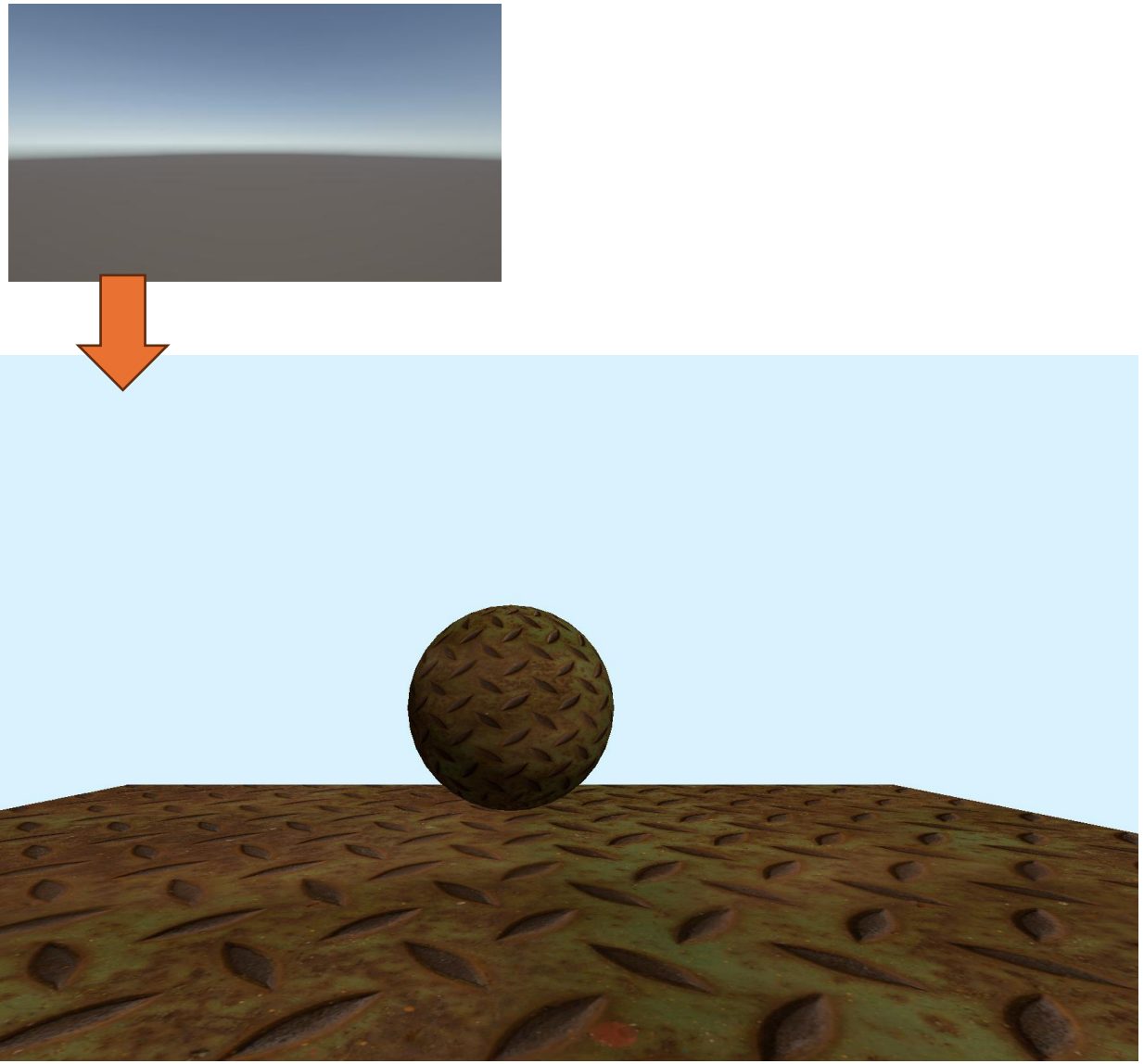
プロパティの設定

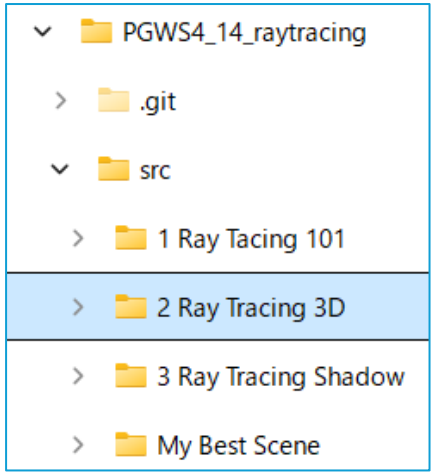

1 Ray Tacing 101 - Untitled - Windows, Mac, Linux - Unity 6.2 (6000.2.6f2) <DX12>
File Edit Assets GameObject Component Services Jobs Window Help
Unity 6
Asset Store
Unity Version
Layout
Hierarchy
Scene
Game
Inspector
Untitled
Main Came
Directional
Game
Display 1
Full HD (1920x1080)
Scale
PC_Renderer (Universal Renderer Data)
Open
Filtering
Prepass Layer Mas Everything
Opaque Layer Mas Everything
Transparent Layer Everything
Rendering
Rendering Path Forward+
Depth Priming M Disabled
Depth Texture Moc After Opaques
Depth Attachment Default
Depth Texture Forr Default
Shadows
Transparent Recei
Post-processing
Enabled
Data PostProcessData (Post Pro
Overrides
Stencil
Compatibility
Intermediate Textu Auto
Renderer Features
Screen Space Ambient Occlusion
New URP Render Feature (New URP Rend
Name NewURPRenderFeature
Script NewURPRenderFeature
Settings
Ray Tracing Shader NewRayTracingShader
Asset Labels
AssetBundle None None
Project
Console
Favorites
All Materials
All Models
All Prefabs
Assets
Scenes
Settings
TutorialInfo
Assets
Scenes
Settings
TutorialInfo
InputSyste...
NewRayTra...
Assets/Settings/PC_Renderer.asset

完成

本日の内容

- レイトレーシング概要
- 簡単なレイトレーシング
- シーンの描画
- レイトレーシング影

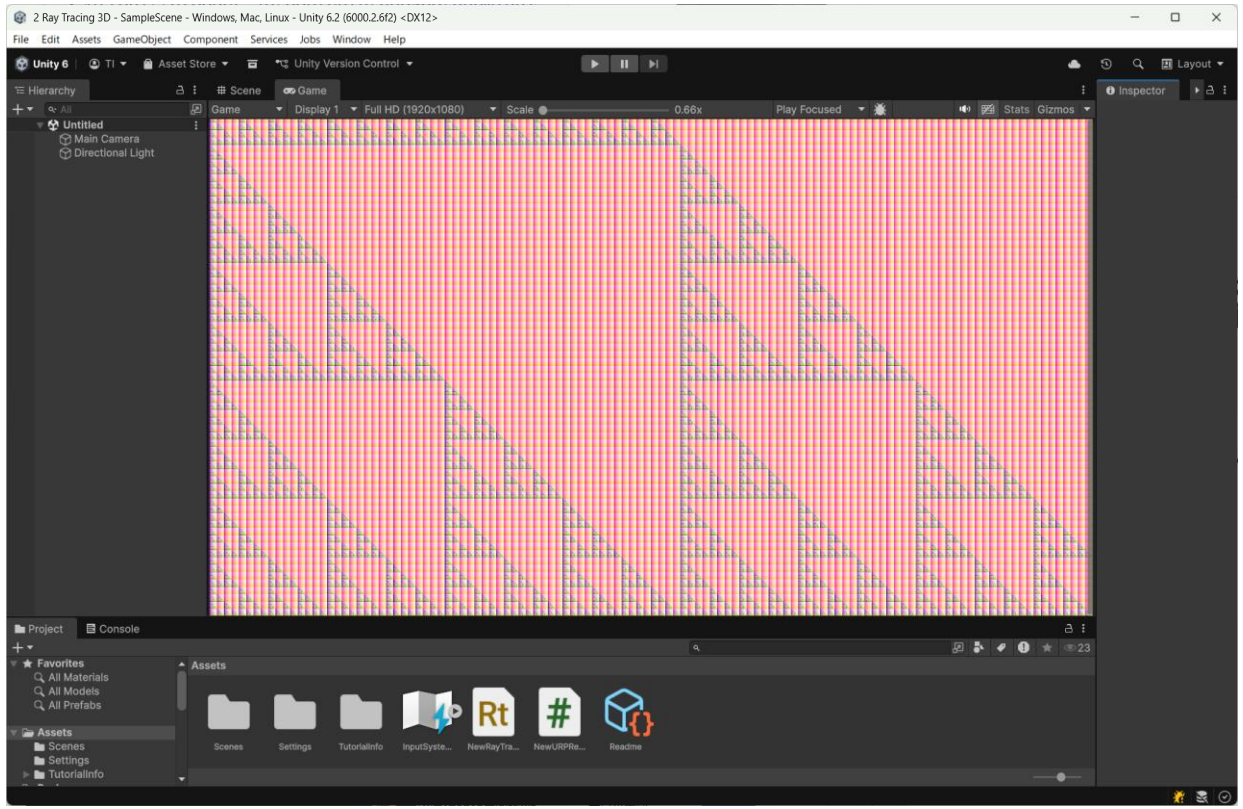




「2 Ray Tracing 3D」ディレクトリ

準備

- 先ほどと同じ状態まで進める
  - 「1 Ray Tracing 101」の中身を「2 Ray Tracing 3D」にコピーしても良い

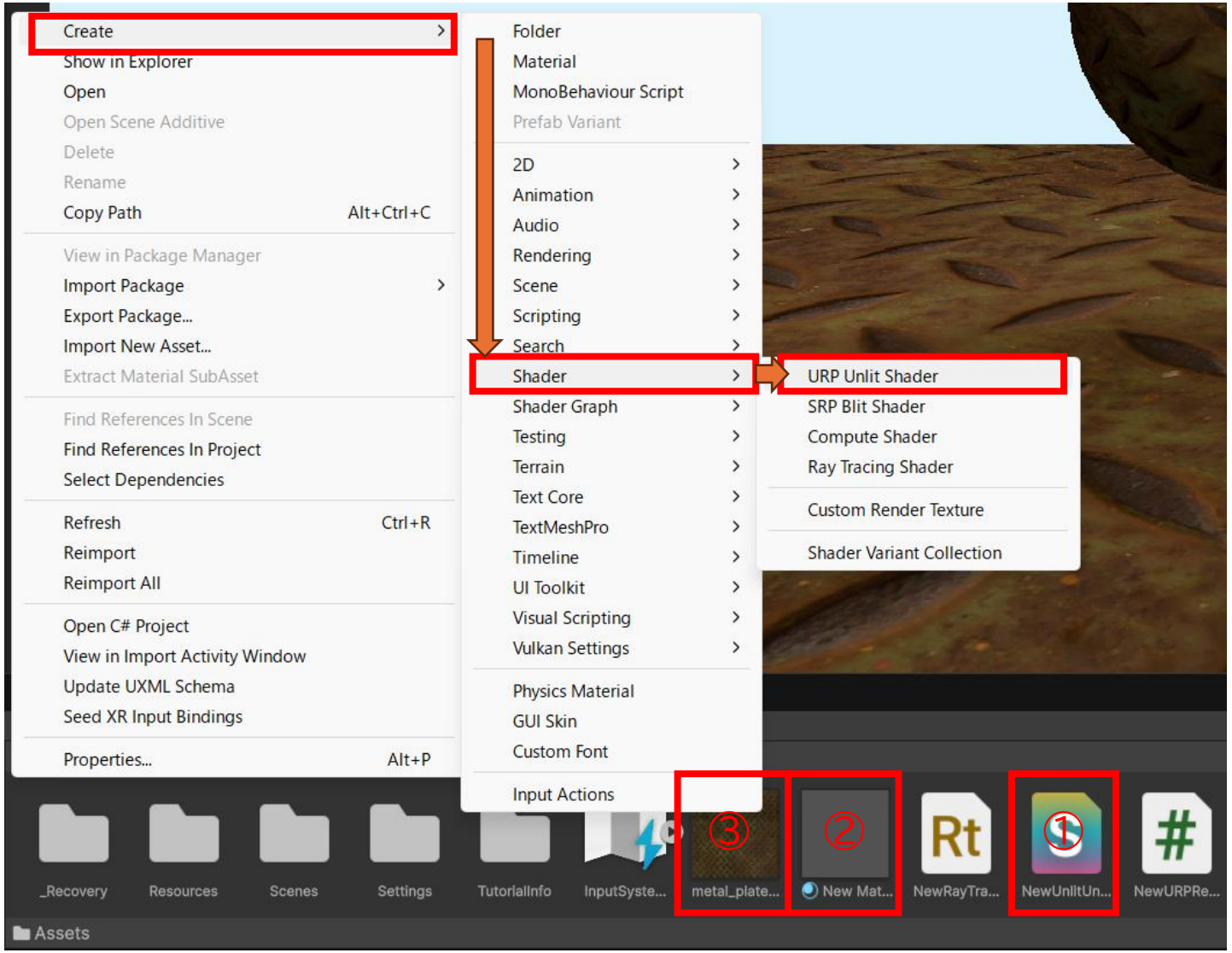


今回の特徴

- シェーダを分ける
  - Ray Tracing Shader: 全体で共通の処理
    - レイを生成する
    - 当らなかった時に背景色を与える
  - URP Unlit Shader: オブジェクトごとの処理
    - 交点の色を計算する
    - テクスチャなどのマテリアルの情報を使用する

アセットの追加

1. URP Unlit Shader
   - 名称例：NewUnlit UniversalRender PipelineShader
     - デフォルト
2. マテリアル
   - 名称例：New Material
     - デフォルト
   - NewUnlitUniversal RenderPipelineShaderを設定
3. テクスチャ
   - なんでもよい
   - ここでは、Poly HavenのMetal PlateのDiffuse
     - https://polyhaven.com/a/metal_plate



シェーダのインクルード・オブジェクト

- シーンの情報を使うために、シェーダライブラリをインクルード
- 加速構造のオブジェクトを追加

```
1   // ★追加
2   #include "Packages/com.unity.render-pipelines.core/ShaderLibrary/Common.hlsl"
3   #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/UnityInput.hlsl"
4
5   RaytracingAccelerationStructure SceneAS;// ★追加
6   RWTexture2D<float4> RenderTarget;
```

レイの生成(ピクセルごとの処理)

- ペイロード:レイの追跡において伝搬されるデータ
- TraceRay: レイを飛ばす

```
37          // ★追加
38          struct RayPayload
39          {
40              bool hit;
41              float3 radiance;
42          };
43
44          [shader("raygeneration")]
45          void MyRaygenShader()
46          {
47              uint2 dispatchIdx = DispatchRaysIndex().xy;
48              // ★追加
49              float2 clipPixel = (dispatchIdx + 0.5) / (float2)DispatchRaysDimensions();
50              clipPixel = clipPixel * 2.0 - 1.0;// NDC変換: [0,1] -> [-1,+1]
51              clipPixel.y = -clipPixel.y;// 上下反転
52              RayDesc ray = generateCameraRay(clipPixel);
53
54              RayPayload payload = (RayPayload)0;
55              TraceRay(SceneAS, 0, 0xFF, 0, 1, 0, ray, payload);
56              RenderTarget[dispatchIdx] = float4(payload.radiance, 1.0);
57          //  RenderTarget[dispatchIdx] = float4(dispatchIdx.x & dispatchIdx.y, (dispatchIdx.x & 15)/15.0, (dispatchIdx.y & 15)/15.0, 0.0);// ★削除
58          }
```

ピクセルの区別

(次々頁)

TraceRay 関数

- 加速構造内の交点を検索するためのレイを送信する

```
Template<payload_t>
void TraceRay(RaytracingAccelerationStructure AccelerationStructure,
              uint RayFlags,
              uint InstanceInclusionMask,
              uint RayContributionToHitGroupIndex,
              uint MultiplierForGeometryContributionToHitGroupIndex,
              uint MissShaderIndex,
              RayDesc Ray,
              inout payload_t Payload);
```

加速構造
探索の仕方をフラグで与える
オブジェクトのマスク情報を使って探索から外せる
衝突用のシェーダをいくつか持つ際の順番の指定
オブジェクトのヒットグループの数
ミスシェーダのインデックスでの選択
レイの情報
レイが伝搬する情報

```
struct RayDesc
{
    float3 Origin;    // 光線の原点
    float  TMin;      // 光線の最小範囲
    float3 Direction; // 光線の方向
    float  TMax;      // 光線の最大範囲
};
```

レイの情報の生成

- 射影行列から直接アスペクト比や視野角を取得する

$$M_{proj} = \begin{bmatrix} \frac{\cot\left(\frac{fov}{2}\right)}{aspect} & 0 & 0 & 0 \\ 0 & \cot\left(\frac{fov}{2}\right) & 0 & 0 \\ 0 & 0 & \frac{z_n}{z_n - z_f} & \frac{-z_n z_f}{z_n - z_f} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

```
13          // ★追加
14          RayDesc generateCameraRay(float2 pixel)
15          {
16              float4x4 view = unity_WorldToCamera;
17              float4x4 proj = unity_CameraProjection ;
18
19              // レイの初期化
20              RayDesc ray;
21              ray.Origin = _WorldSpaceCameraPos;
22              ray.TMin = 0.f;
23              ray.TMax = FLT_MAX;
24
25              // 射影行列からアスペクト比と視野角を展開する
26              float aspect = proj[1][1] / proj [0][0];
27              float tanHalfFovY = 1.0f / proj [1][1];
28
29              // ピクセルからレイの向きを計算する
30              ray.Direction = normalize(
31                  (pixel.x * view[0].xyz * tanHalfFovY * aspect) -
32                  (pixel.y * view[1].xyz * tanHalfFovY) + view[2].xyz);
33
34              return ray;
35          }
```

Missシェーダ

- レイが衝突しないときに呼び出されるシェーダ
- 背景色を設定する

```
60          // ★追加
61          [shader("miss")]
62          void MyMissShader(inout RayPayload payload : SV_RayPayload)
63          {
64              payload.radiance = float3(0.7, 0.9, 1.0);// 空の色
65              payload.hit = false;
66          }
```

加速構造の設定

- テクスチャと同様にUnsafeCommarBufferで指定

```
109             // This static method is passed as the RenderFunc delegate to the RenderGraph render pass.
110             // It is used to execute draw commands.
111             // ■この静的メソッドは、RenderGraphレンダーパスにRenderFuncデリゲートとして渡されます。
112             // ■描画コマンドを実行するために使用されます.
113         //  static void ExecutePass(PassData data, RasterGraphContext context)
                1 個の参照
114             static void ExecutePass(PassData data, UnsafeGraphContext context)// ■差し替え
115             {
116                 // ■ 追加
117                 // UnsafeGraphContext からネイティブの CommandBuffer (GPUの命令を溜めるバッファ) を取得
118                 var native_cmd = CommandBufferHelpers.GetNativeCommandBuffer(context.cmd);
119                 // レイトレシェーダーパスを設定
120                 native_cmd.SetRayTracingShaderPass(data.rayTracingShader, "NewURPRenderFeaturePass");
121                 // ★追加：レイトレシェーダーに加速構造を設定
122                 context.cmd.SetRayTracingAccelerationStructure(data.rayTracingShader,
123                     Shader.PropertyToID("SceneAS"), data.rayTracingAccelerationStructure);
124                 // レイトレシェーダーに出力テクスチャを設定
125                 context.cmd.SetRayTracingTextureParam(data.rayTracingShader,
126                     Shader.PropertyToID("RenderTarget"), data.output_ColorTexture);
127                 // レイトレを実行
128                 context.cmd.DispatchRays(data.rayTracingShader, "MyRaygenShader",
129                     (uint)data.camera.pixelWidth, (uint)data.camera.pixelHeight, 1, data.camera);
130                 // 結果をカメラに書き戻す
131                 native_cmd.Blit(data.output_ColorTexture, data.camera_ColorTarget);
132             }
```

URP Unlit Shader

- パス名の追加
  - NewURPRenderFeature::NewURPRenderFeature PassでRenderPass::ExecutePassで設定
- LightMode(パスの役割)のタグにもパス名を指定
- レイトレーシングの関数名を設定

```
Shader "Custom/NewUnlitUniversalRenderPipelineShader"
{
    Properties
    {
        [MainColor] _BaseColor("Base Color", Color) = (1, 1, 1, 1)
        [MainTexture] _BaseMap("Base Map", 2D) = "white"
    }

    SubShader
    {
        Tags { "LightMode" = "NewURPRenderFeaturePass"}// ★追加
//      Tags { "RenderType" = "Opaque" "RenderPipeline" = "UniversalPipeline"}// ★削除

        Pass
        {
            Name "NewURPRenderFeaturePass"// ★追加: SetRayTracingShaderPassで設定したパス名

            HLSLPROGRAM

//          #pragma vertex vert// ★削除
//          #pragma fragment frag// ★削除
            #pragma raytracing surface_shader// ★追加

            ENDHLSL
        }
    }
}
```

ラスタライズ用のシェーダは不要

- _BaseMap等の宣言は次頁で読み込むLitInput.hlsl内にある

```
29          #if false // ★削除
30                      struct Attributes
31                      {
32                          float4 positionOS : POSITION;
33                          float2 uv : TEXCOORD0;
34                      };
35
36                      struct Varyings
37                      {
38                          float4 positionHCS : SV_POSITION;
39                          float2 uv : TEXCOORD0;
40                      };
41
42                      TEXTURE2D(_BaseMap);
43                      SAMPLER(sampler_BaseMap);
44
45                      CBUFFER_START(UnityPerMaterial)
46                          half4 _BaseColor;
47                          float4 _BaseMap_ST;
48                      CBUFFER_END
49
50                      Varyings vert(Attributes IN)
51                      {
52                          Varyings OUT;
53                          OUT.positionHCS = TransformObjectToHClip(IN.positionOS.xyz);
54                          OUT.uv = TRANSFORM_TEX(IN.uv, _BaseMap);
55                          return OUT;
56                      }
57                      half4 frag(Varyings IN) : SV_Target
58                      {
59                          half4 color = SAMPLE_TEXTURE2D(_BaseMap, sampler_BaseMap, IN.uv) * _BaseColor;
60                          return color;
61                      }
62          #endif
```

構造体メンバー

- raytraceと同じペイロードを定義
- AttributeDataは、交差した位置の情報(重心座標
- raytraceと同じ加速構造を宣言
- 頂点宣言は使わないものも含めた

```
24      // #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl"// ★削除
25         #include "Packages/com.unity.render-pipelines.universal/Shaders/LitInput.hlsl"// ★追加
26         #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"// ★追加
27         #include "UnityRaytracingMeshUtils.cginc"// ★追加
28
29      #if false // ★削除...
62      #endif
63
64         // ★以降追加
65
66         struct RayPayload
67         {
68             bool hit;
69             float3 radiance;
70         };
71
72         struct AttributeData
73         {
74             float2 barycentrics;
75         };
76
77         RaytracingAccelerationStructure SceneAS;
78
79         struct Vertex
80         {
81             float3 position;
82             float3 normal;
83             float4 tangent;
84             float2 texCoord0;
85             float2 texCoord1;
86             float2 texCoord2;
87             float2 texCoord3;
88             float4 color;
89         };
```

頂点情報の読み込み

- 衝突したポリゴンのインデックス情報を通して頂点情報を取得

```
137             [shader("closesthit")]
138             void ClosestHitMain(inout RayPayload payload : SV_RayPayload, AttributeData attribs : SV_IntersectionAttributes)
139             {
140                 // レイトレの交点から頂点インデックスを取得
141                 uint3 triangleIndices = UnityRayTracingFetchTriangleIndices(PrimitiveIndex());
142
143                 // 頂点インデックスから三角形の3頂点を取得
144                 Vertex v0, v1, v2;
145                 v0 = FetchVertex(triangleIndices.x);
146                 v1 = FetchVertex(triangleIndices.y);
147                 v2 = FetchVertex(triangleIndices.z);
```

```
91              // Unity ビルトイン関数を使って頂点データを得る関数
92              Vertex FetchVertex(uint vertexIndex)
93              {
94                  Vertex v;
95                  v.position  = UnityRayTracingFetchVertexAttribute3(vertexIndex, kVertexAttributePosition);
96                  v.normal    = UnityRayTracingFetchVertexAttribute3(vertexIndex, kVertexAttributeNormal);
97                  v.tangent   = UnityRayTracingFetchVertexAttribute4(vertexIndex, kVertexAttributeTangent);
98                  v.texCoord0 = UnityRayTracingFetchVertexAttribute2(vertexIndex, kVertexAttributeTexCoord0);
99                  v.texCoord1 = UnityRayTracingFetchVertexAttribute2(vertexIndex, kVertexAttributeTexCoord1);
100                 v.texCoord2 = UnityRayTracingFetchVertexAttribute2(vertexIndex, kVertexAttributeTexCoord2);
101                 v.texCoord3 = UnityRayTracingFetchVertexAttribute2(vertexIndex, kVertexAttributeTexCoord3);
102                 v.color     = UnityRayTracingFetchVertexAttribute4(vertexIndex, kVertexAttributeColor);
103                 return v;
104             }
```

3頂点の情報を重心座標で補間

- 重心座標は2成分が得られる
  - 3成分の合計が1であることから残りを計算

```
106             // 重心座標での補間関数
107             float2 barycentricInterpolate2(float2 v0, float2 v1, float2 v2, float3 barycentrics)
108             {
109                 return v0 * barycentrics.x + v1 * barycentrics.y + v2 * barycentrics.z;
110             }
111
112             float3 barycentricInterpolate3(float3 v0, float3 v1, float3 v2, float3 barycentrics)
113             {
114                 return v0 * barycentrics.x + v1 * barycentrics.y + v2 * barycentrics.z;
115             }
116
117             float4 barycentricInterpolate4(float4 v0, float4 v1, float4 v2, float3 barycentrics)
118             {
119                 return v0 * barycentrics.x + v1 * barycentrics.y + v2 * barycentrics.z;
120             }
121
122             // レイトレのヒットした位置から頂点データを補間する関数
123             Vertex InterpolateVertices(Vertex v0, Vertex v1, Vertex v2, float3 barycentrics)
124             {
125                 Vertex v;
126                 v.position  = barycentricInterpolate3(v0.position,  v1.position,  v2.position,  barycentrics);
127                 v.normal    = barycentricInterpolate3(v0.normal,    v1.normal,    v2.normal,    barycentrics);
128                 v.tangent   = barycentricInterpolate4(v0.tangent,   v1.tangent,   v2.tangent,   barycentrics);
129                 v.texCoord0 = barycentricInterpolate2(v0.texCoord0, v1.texCoord0, v2.texCoord0, barycentrics);
130                 v.texCoord1 = barycentricInterpolate2(v0.texCoord1, v1.texCoord1, v2.texCoord1, barycentrics);
131                 v.texCoord2 = barycentricInterpolate2(v0.texCoord2, v1.texCoord2, v2.texCoord2, barycentrics);
132                 v.texCoord3 = barycentricInterpolate2(v0.texCoord3, v1.texCoord3, v2.texCoord3, barycentrics);
133                 v.color     = barycentricInterpolate4(v0.color,     v1.color,     v2.color,     barycentrics);
134                 return v;
135             }
```

```
137             [shader("closesthit")]
138             void ClosestHitMain(inout RayPayload payload : SV_RayPayload, AttributeData attribs : SV_IntersectionAttributes)
139             {
140                 // レイトレの交点から頂点インデックスを取得
141                 uint3 triangleIndices = UnityRayTracingFetchTriangleIndices(PrimitiveIndex());
142
143                 // 頂点インデックスから三角形の3頂点を取得
144                 Vertex v0, v1, v2;
145                 v0 = FetchVertex(triangleIndices.x);
146                 v1 = FetchVertex(triangleIndices.y);
147                 v2 = FetchVertex(triangleIndices.z);
148
149                 // 3頂点からの情報を交点の重心座標で補間
150                 float3 barycentricCoords = float3(
151                     1.0 - attribs.barycentrics.x - attribs.barycentrics.y,
152                     attribs.barycentrics.x,
153                     attribs.barycentrics.y);
154                 Vertex v = InterpolateVertices(v0, v1, v2, barycentricCoords);
```

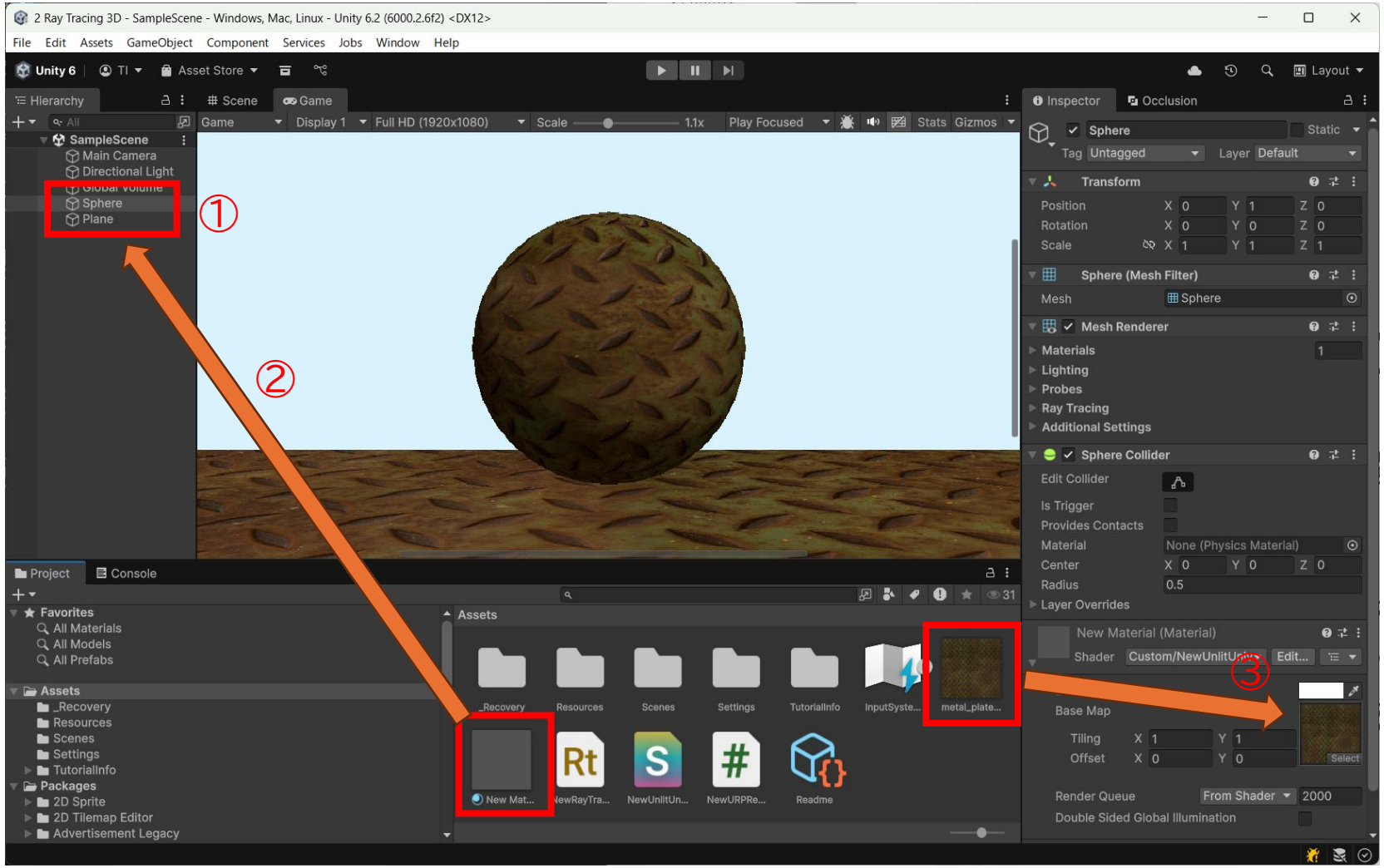
色の計算

- 情報が得られれば照明計算できる
- ペイロードで出力

```
137             [shader("closesthit")]
138             void ClosestHitMain(inout RayPayload payload : SV_RayPayload, AttributeData attribs : SV_IntersectionAttributes)
139             {
140                 // レイトレの交点から頂点インデックスを取得
141                 uint3 triangleIndices = UnityRayTracingFetchTriangleIndices(PrimitiveIndex());
142
143                 // 頂点インデックスから三角形の3頂点を取得
144                 Vertex v0, v1, v2;
145                 v0 = FetchVertex(triangleIndices.x);
146                 v1 = FetchVertex(triangleIndices.y);
147                 v2 = FetchVertex(triangleIndices.z);
148
149                 // 3頂点からの情報を交点の重心座標で補間
150                 float3 barycentricCoords = float3(
151                     1.0 - attribs.barycentrics.x - attribs.barycentrics.y,
152                     attribs.barycentrics.x,
153                     attribs.barycentrics.y);
154                 Vertex v = InterpolateVertices(v0, v1, v2, barycentricCoords);
155
156                 // ベースマップのテクスチャから色を取得
157                 // _BaseMap などのプロパティは LitInput.hlsl を include すれば使用できる
158                 float3 color = _BaseColor.rgb * SAMPLE_TEXTURE2D_LOD(_BaseMap, sampler_BaseMap, v.texCoord0, 0).rgb;
159                 // 簡易ライティング計算
160                 float3 normal = TransformObjectToWorldNormal(v.normal);
161                 color *= lerp(0.1, 1.0, max(0, dot(normal, _MainLightPosition)));// [0.1, 1.0]の範囲
162                 color = _MainLightColor * color;
163
164                 payload.hit = true;
165                 payload.radiance = color;
166 //              payload.radiance = v.normal * 0.5 + 0.5;// 法線を色に変換して表示する場合
167             }
```

オブジェクトの追加

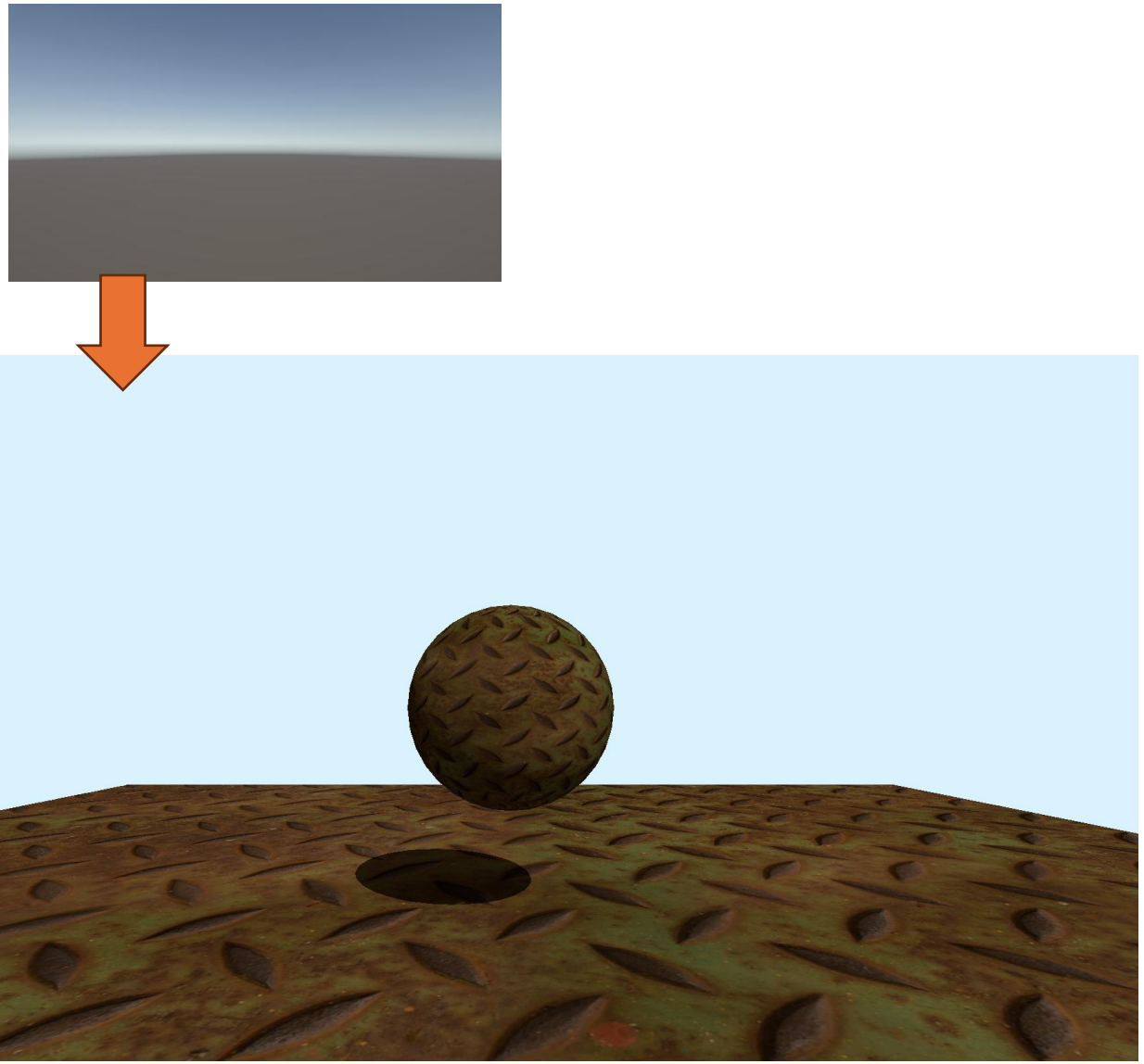
1. 球や床を追加
   - Sphere
   - Plane
2. マテリアルを設定
   - 両方のオブジェクトに設定する
3. マテリアルにテクスチャを設定

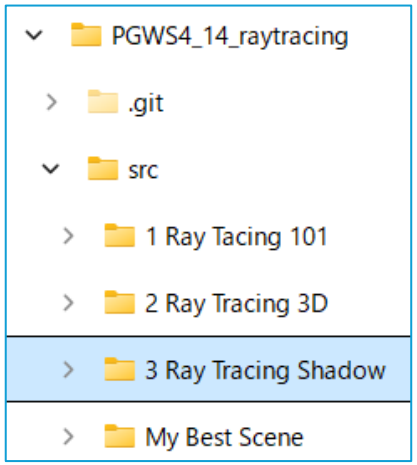


完成

本日の内容

- レイトレーシング概要
- 簡単なレイトレーシング
- シーンの描画
- レイトレーシング影

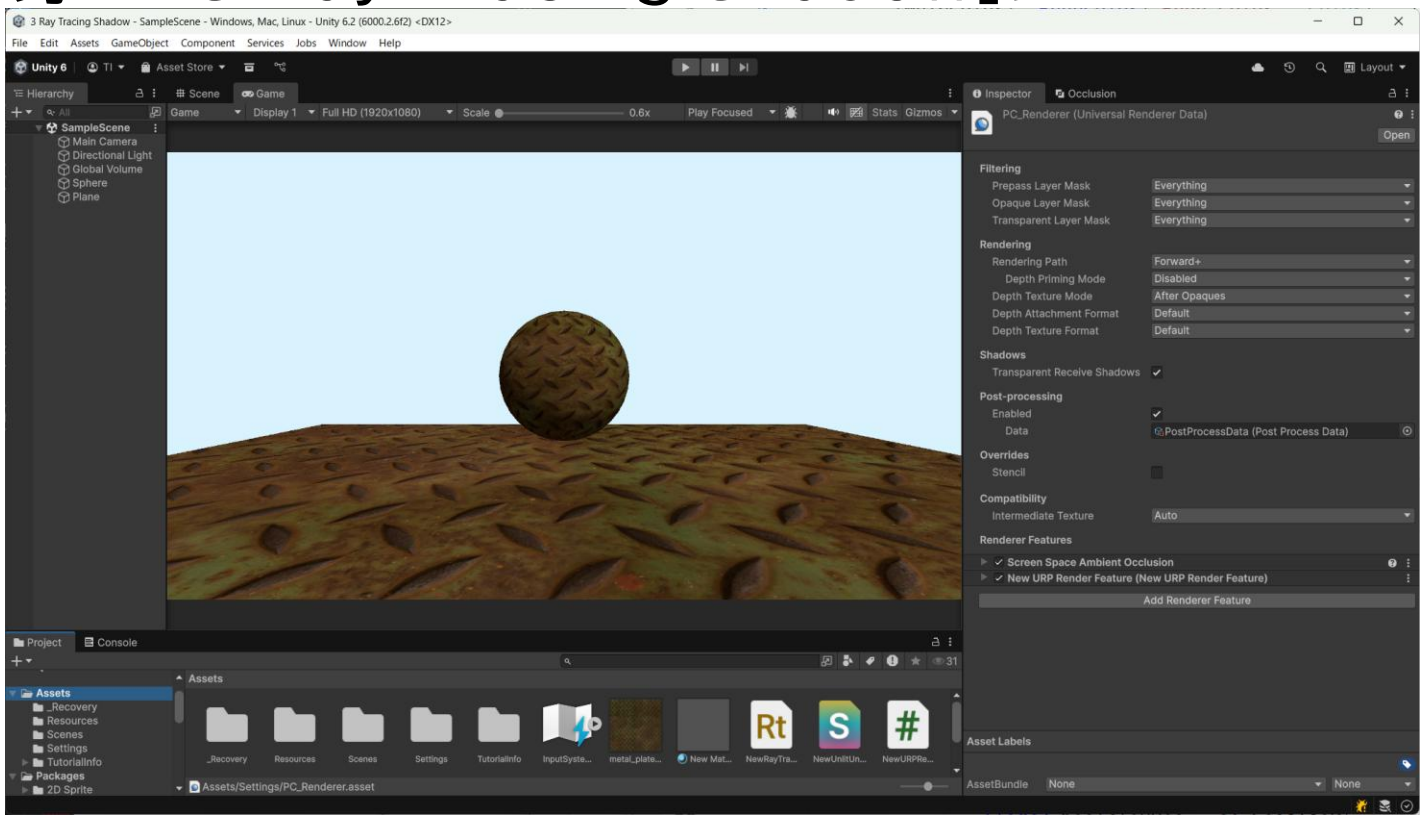




「3 Ray Tracing Shadow」ディレクトリ

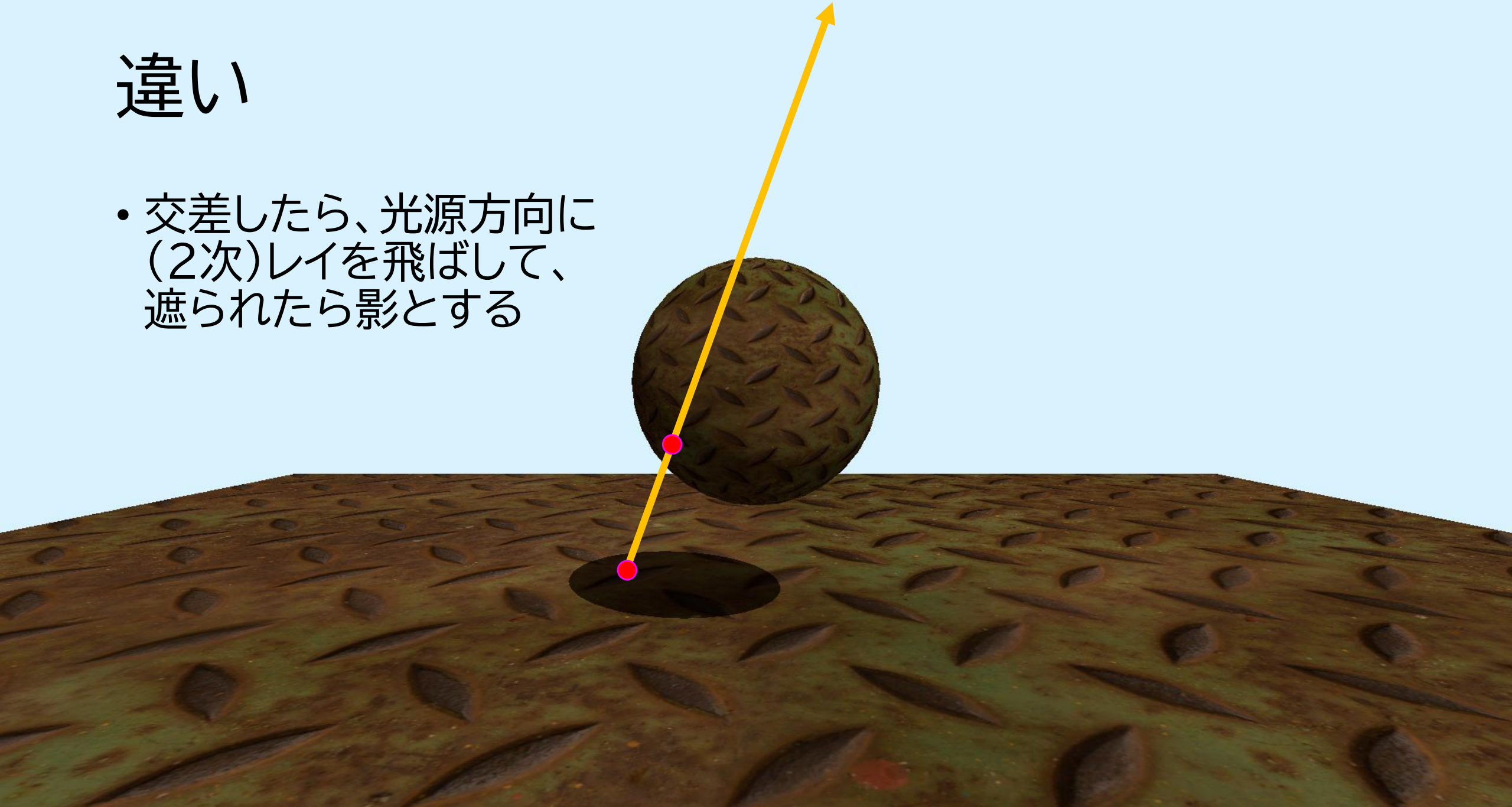
準備

- 先ほどと同じ状態まで進める
  - 「2 Ray Tracing 3D」の中身を「3 Ray Tracing Shadow」にコピーしても良い



違い

- 交差したら、光源方向に(2次)レイを飛ばして、遮られたら影とする



レイトレースのシェーダの変更

- レイを飛ばす上限を2段階
- ミスシェーダの名称を変更
  - ミスシェーダのインデックスが名前のアルファベット順になる
  - 2つのミスシェーダを使うので、確実にアルファベット順にする
    1. Miss00_MyMissShader
    2. Miss01_ShadowMissShader
- 影用のペイロードは色が不要なので、小さくできる
  - 衝突したかどうかだけ

```
1   // ★追加
2   #include "Packages/com.unity.render-pipelines.core/ShaderLibrary/Common.hlsl"
3   #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/UnityInput
4
5   RaytracingAccelerationStructure SceneAS;// ★追加
6   RWTexture2D<float4> RenderTarget;
7
8   // Uncomment this pragma for debugging the HLSL code in PIX. GPU performance wil
9   //#pragma enable_ray_tracing_shader_debug_symbols
10
11  #pragma max_recursion_depth 2 // ●2メインのレイとシャドウレイ
12
13  // ★追加
14  RayDesc generateCameraRay(float2 pixel)
15  {...
35  }
36
37  // ★追加
38  struct RayPayload
39  {...
42  };
43
44  [shader("raygeneration")]
45  void MyRaygenShader()
46  {...
57  //  RenderTarget[dispatchIdx] = float4(dispatchIdx.x & dispatchIdx.y, (dispatchI
58  }
59
60  // ★追加
61  [shader("miss")]
62  void Miss00_MyMissShader(inout RayPayload payload : SV_RayPayload)// ●名称変更
63  {...
66  }
67
68  // ●追加
69  struct ShadowPayload
70  {
71      bool hit;
72  };
73
74  // ●追加
75  [shader("miss")]
76  void Miss01_ShadowMissShader(inout ShadowPayload payload : SV_RayPayload)
77  {
78      payload.hit = false;
79  }
```

オブジェクトのシェーダ

- 色の計算で追加でレイを飛ばして(次頁のTraceRay)その結果で陰影をつける
  - 位置はワールド座標系

```
159                 [shader("closesthit")]
160                 void ClosestHitMain(inout RayPayload payload : SV_RayPayload, AttributeData attribs : SV_IntersectionAttributes)
161                 {
162                     // レイトレの交点から頂点インデックスを取得
163                     uint3 triangleIndices = UnityRayTracingFetchTriangleIndices(PrimitiveIndex());
164
165                     // 頂点インデックスから三角形の3頂点を取得
166                     Vertex v0, v1, v2;
167                     v0 = FetchVertex(triangleIndices.x);
168                     v1 = FetchVertex(triangleIndices.y);
169                     v2 = FetchVertex(triangleIndices.z);
170
171                     // 3頂点からの情報を交点の重心座標で補間
172                     float3 barycentricCoords = float3(
173                         1.0 - attribs.barycentrics.x - attribs.barycentrics.y,
174                         attribs.barycentrics.x,
175                         attribs.barycentrics.y);
176                     Vertex v = InterpolateVertices(v0, v1, v2, barycentricCoords);
177
178                     // ベースマップのテクスチャから色を取得
179                     // _BaseMap などのプロパティは LitInput.hlsl を include すれば使用できる
180                     float3 color = _BaseColor.rgb * SAMPLE_TEXTURE2D_LOD(_BaseMap, sampler_BaseMap, v.texCoord0, 0).rgb;
181                     // 簡易ライティング計算
182                     float3 normal = TransformObjectToWorldNormal(v.normal);
183                     // ●修正
184                     float3 position = TransformObjectToWorld(v.position);
185                     bool is_shadow = TraceShadow(_MainLightPosition, normal, position);
186                     float shade = is_shadow ? 0.1 :
187                         lerp(0.1, 1.0, max(0, dot(normal, _MainLightPosition)));// [0.1, 1.0]の範囲
188                     color = _MainLightColor * color * shade;
189
190                     payload.hit = true;
191                     payload.radiance = color;
192         //          payload.radiance = v.normal * 0.5 + 0.5;// 法線を色に変換して表示する場合
193                 }
```

シャドウレイ

- すぐに衝突しないように法線方向に微妙にずらして探索

```
137                 // ●追加
138                 struct ShadowPayload
139                 {
140                     bool hit;
141                 };
142
143                 // ●追加: シャドウレイ用ペイロード
144                 bool TraceShadow(float3 lightDirection, float3 normal, float3 position)
145                 {
146                     RayDesc ray;
147                     ray.Origin = position + 1e-6 * normal;
148                     ray.Direction = lightDirection;
149                     ray.TMin = 1e-06;
150                     ray.TMax = 10000.0;
151
152                     ShadowPayload payload = (ShadowPayload)0;
153                     payload.hit = true;
154                     TraceRay(SceneAS, RAY_FLAG_SKIP_CLOSEST_HIT_SHADER | RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH,
155                             0xFF, 0, 1, 1 /* Miss01_ShadowMissShader */, ray, payload);
156                     return payload.hit;
157                 }
```

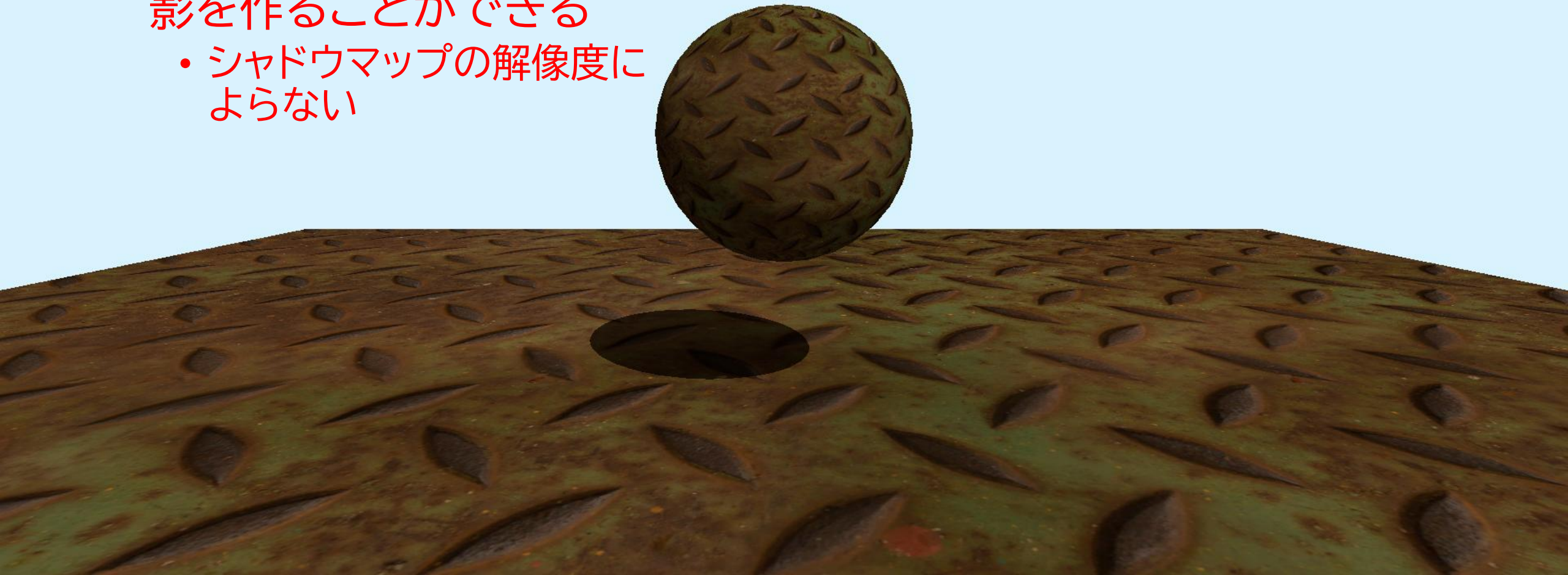
光源方向に追跡

Closestシェーダは使わない

一回でも交差したら終了(高速化)

完成

- 今回の例では利点が見えにくいが、ポリゴンエッジのぱっきりした影を作ることができる
  - シャドウマップの解像度によらない



まとめ

- レイトレーシング概要
- 簡単なレイトレーシング
  - デフォルトで生成されるレイトレーシングのコードを表示する
- シーンの描画
  - シーンに配置されたオブジェクトをレイトレーシングで表示する
- レイトレーシング影
  - 2次レイを飛ばしてレイトレーシングならではのくっきりした影を実現