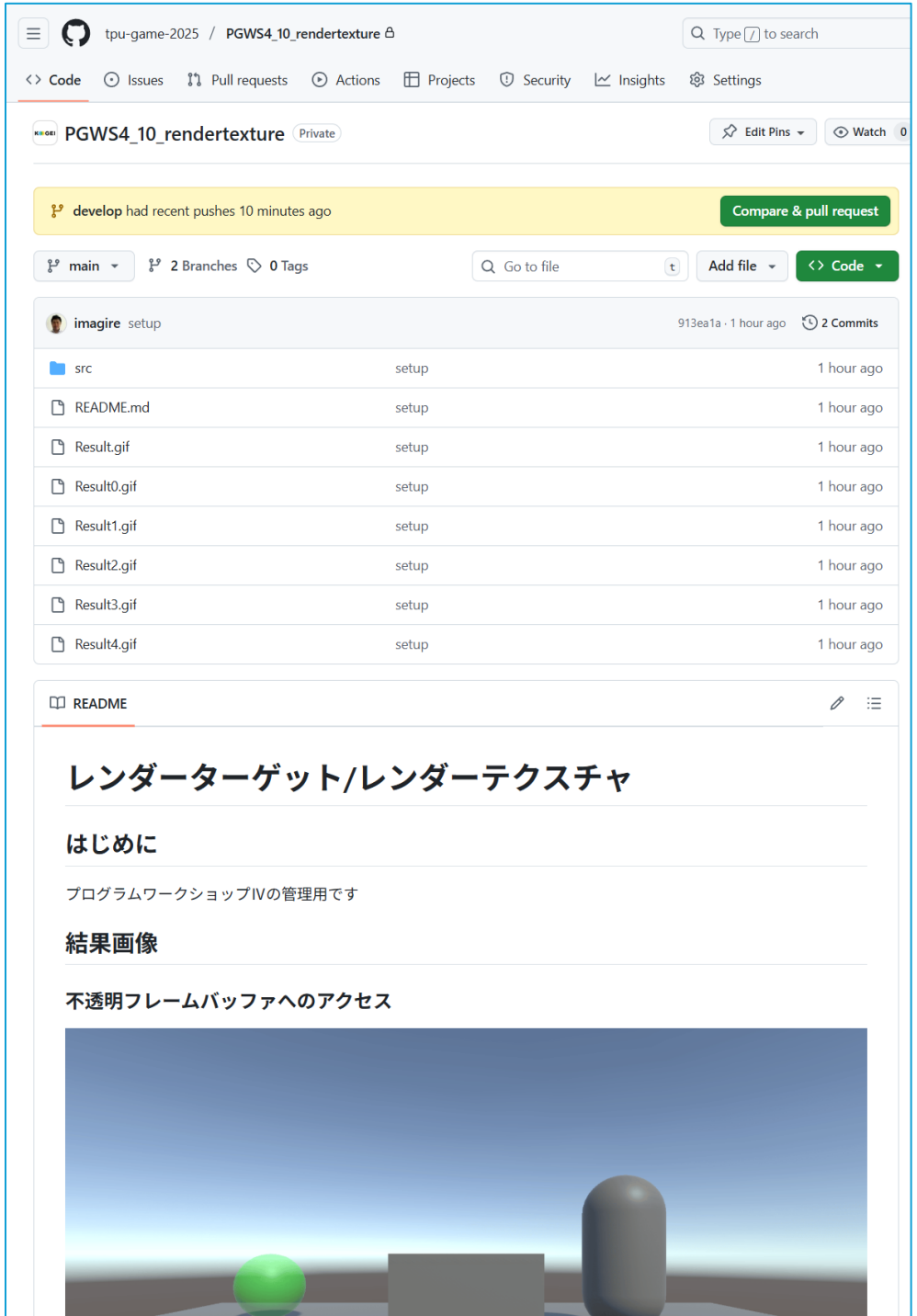
レンダーターゲット・レンダーテクスチャ

2025年度 プログラムワークショップIV（10）

今回のリポジトリ

- https://github.com/tpu-game-2025/PGWS4_10_rendertexture



KOGEI TOKYO POLYTECHNIC UNIVERSITY

本日の内容

- レンダーターゲット
  - レンダーターゲットの概要
  - 不透明フレームバッファへのアクセス
  - 深度からの位置・法線の復元
- レンダーテクスチャ
  - レンダーテクスチャの概要
  - ゲーム内モニター
  - バックミラー
  - 範囲内のオブジェクトだけ単色

本日の内容

- レンダーターゲット
  - レンダーターゲットの概要
  - 不透明フレームバッファへのアクセス
  - 深度からの位置・法線の復元
- レンダーテクスチャ
  - レンダーテクスチャの概要
  - ゲーム内モニター
  - バックミラー
  - 範囲内のオブジェクトだけ単色

レンダーターゲットの概要

レンダーターゲット：描画対象のバッファオブジェクト

- フォーマット種別
  - カラーバッファ
    - 「色」を出力する
    - 4の倍数バイトがきりが良いので、RGBに加えてα成分を持つ場合が多い
      - フォーマット次第
  - 深度バッファ
  - ステンシルバッファ
- バッファ種別
  - レンダーテクスチャ
    - 描画先とできるテクスチャ
    - 描画結果をテクスチャとして読み込むことができる
  - フレームバッファ
    - 表示に用いられるレンダーターゲット
    - テクスチャとして読み込めるかどうかは環境依存

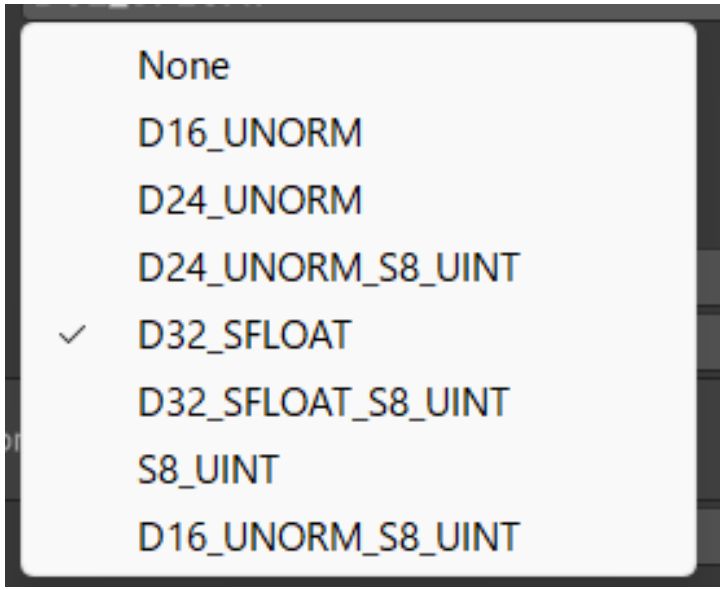
カラーバッファ

- 多くのフォーマット
  - デバイスごとに対応しているフォーマットが異なる
- バッファの種類
  - 成分: R (赤), G (緑), B (青), A(α)、E(指数)
  - 並び順: RGBA, ARGB, BGRA, …
  - 型:
    - 整数, 浮動小数点数
    - 符号付き(S)、符号なし(U)
  - サイズ:
    - 8,16,32ビット
    - 1, 2, 4, 5, 9, 19, 11のような特殊なサイズもある
      - まとめた際に2のべき乗のきりが良くなるように
  - PACK
    - 複数をまとめてatomicな型としてまとめる
  - 範囲制限
    - UNORM([0,1]), SNORM([-1,+1])
  - ガンマ補正
    - SRGB: sRGB色空間で作られた画像への対応
      - sRGB色空間: ディスプレイでみている色の明るさ
      - リニア色空間: 物理的な四則演算が成り立つ色空間

R8_SRGB
R8G8_SRGB
R8G8B8A8_SRGB
R8_UNORM
R8G8_UNORM
R8G8B8A8_UNORM
R8_SNORM
R8G8_SNORM
R8G8B8A8_SNORM
R8_UINT
R8G8_UINT
R8G8B8A8_UINT
R8_SINT
R8G8_SINT
R8G8B8A8_SINT
R16_UNORM
R16G16_UNORM
R16G16B16A16_UNORM
R16_SNORM
R16G16_SNORM
R16G16B16A16_SNORM
R16_UINT
R16G16_UINT
R16G16B16A16_UINT
R16_SINT
R16G16_SINT
R16G16B16A16_SINT
R32_UINT
R32G32_UINT
R32G32B32A32_UINT
R32_SINT
R32G32_SINT
R32G32B32A32_UINT
R32_SINT
R32G32_SINT
R32G32B32A32_SINT
R16_SFLOAT
R16G16_SFLOAT
R16G16B16A16_SFLOAT
R32_SFLOAT
R32G32_SFLOAT
R32G32B32A32_SFLOAT
B8G8R8A8_SRGB
B8G8R8A8_UNORM
B8G8R8A8_SNORM
B8G8R8A8_UINT
B8G8R8A8_SINT
R4G4B4A4_UNORM_PACK16
B4G4R4A4_UNORM_PACK16
R5G6B5_UNORM_PACK16
B5G6R5_UNORM_PACK16
R5G5B5A1_UNORM_PACK16
B5G5R5A1_UNORM_PACK16
A1R5G5B5_UNORM_PACK16
B9G9R9E5_UFLOAT_PACK32
B10G11R11_UFLOAT_PACK32
A2B10G10R10_UNORM_PACK32
A2B10G10R10_UINT_PACK32
A2B10G10R10_SINT_PACK32
A2R10G10B10_UNORM_PACK32
A2R10G10B10_UINT_PACK32
A2R10G10B10_SINT_PACK32

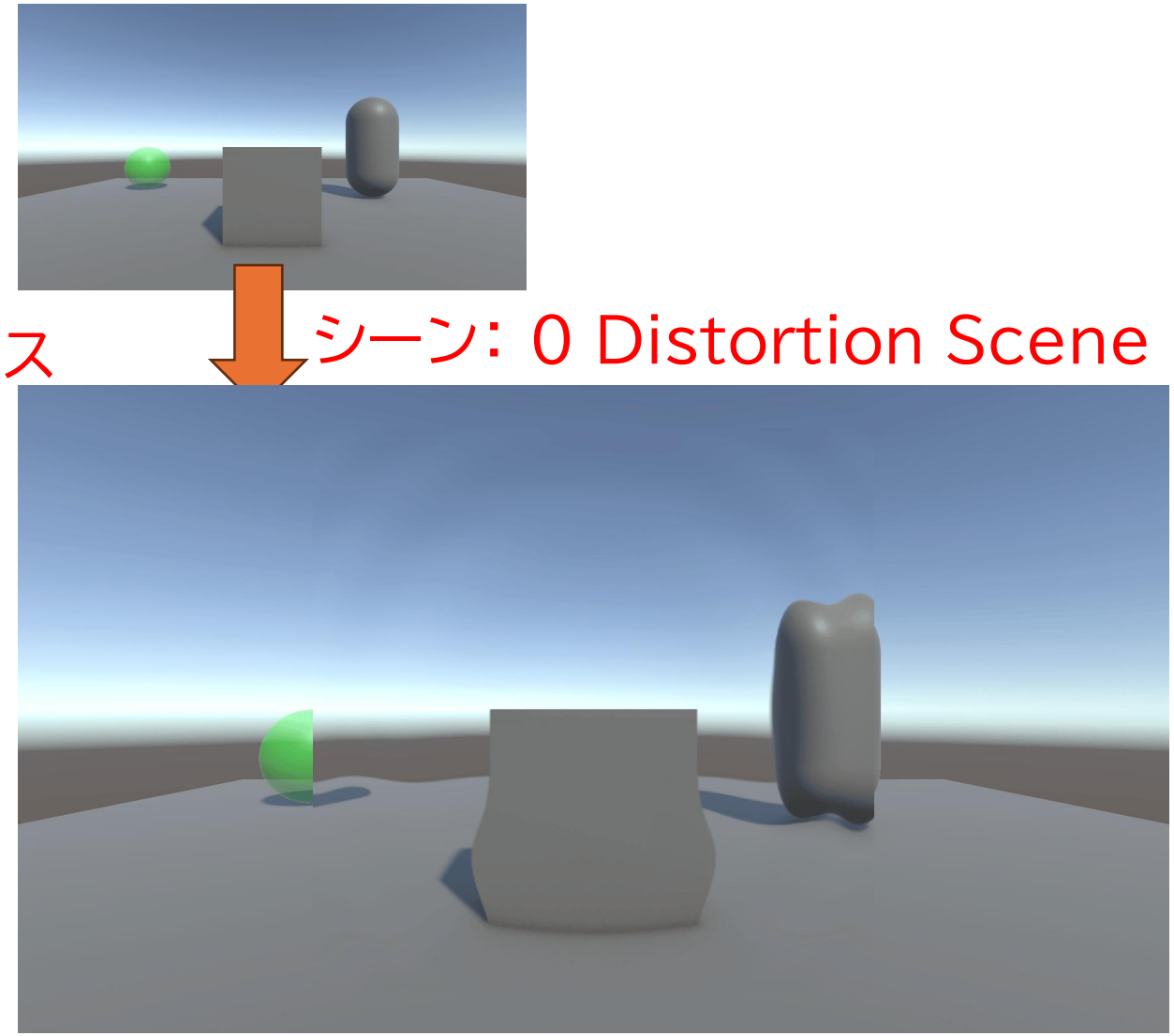
Depth Stencil Format

- 深度バッファとステンシルバッファ
  - 深度バッファ：描画するポリゴンの奥行き値を記録
    - 現代的な実装：手前は1で奥は0（Reversed-Z）
  - ステンシルバッファ：8ビットのバッファ
    - 値を比較しての描画のON/OFF
      - バッファ値と出力した値を比較して特定の時だけ描画
        - 大きい・小さい・等しい・等しくない
      - 特殊な書き込みが可能
        - インクリメント・デクリメント
        - ビット反転
        - 直値
    - 昔は深度バッファが24ビットだったので、余り成分の活用
    - シャドウボリュームという技法が昔はあった…
    - Shader Graphからアクセスできない…



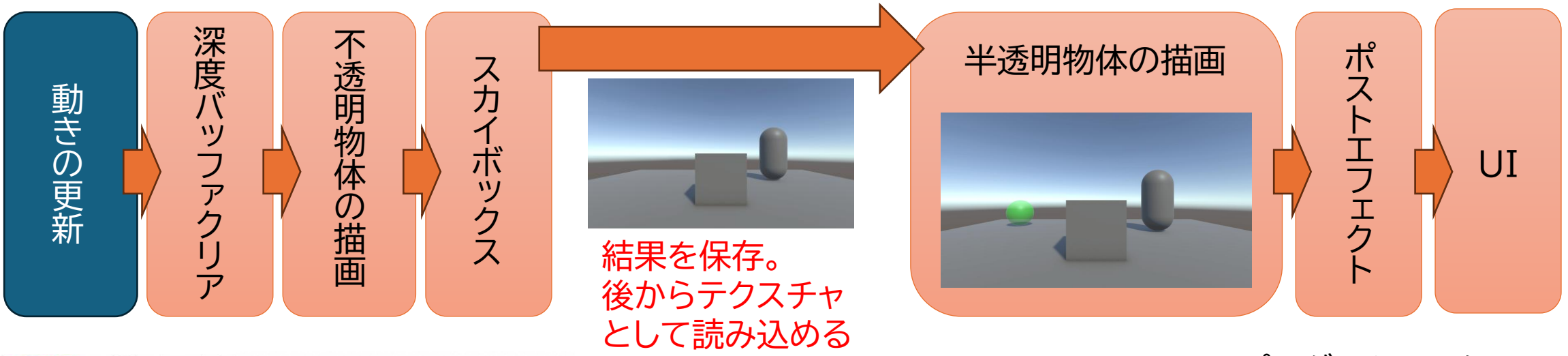
本日の内容

- レンダーターゲット
  - レンダーターゲットの概要
  - 不透明フレームバッファへのアクセス
  - 深度からの位置・法線の復元
- レンダーテクスチャ
  - レンダーテクスチャの概要
  - ゲーム内モニター
  - バックミラー
  - 範囲内のオブジェクトだけ単色



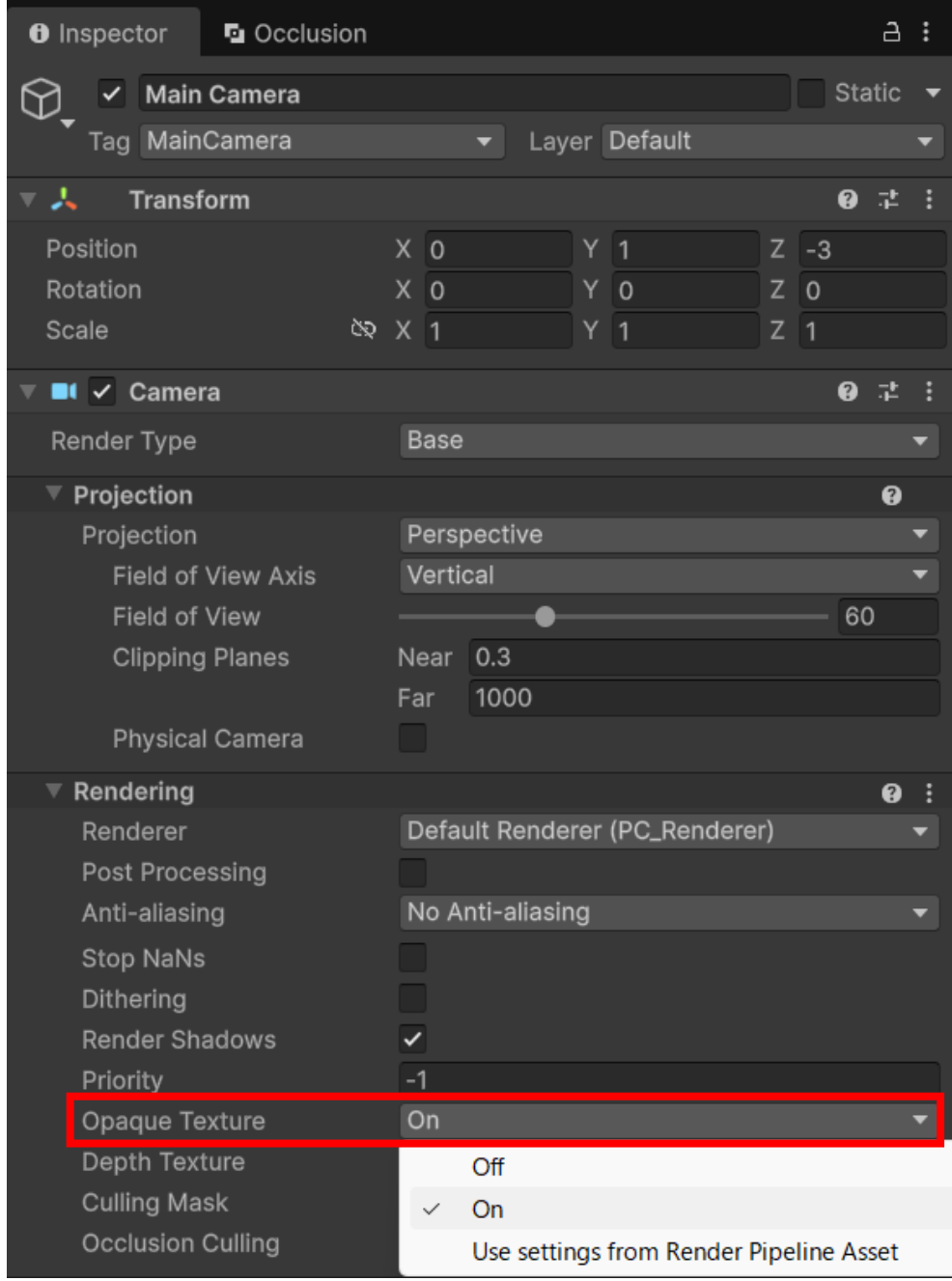
不透明フレームバッファへのアクセス

- 描画しているフレームバッファへはアクセスできない
  - 絵が完成していないからね
- 特別に半透明描画では、不透明描画の結果を参照できる
  - 不透明描画後（含むスカイボックス）の結果を裏で保存して再利用できる
    - 半透明の描画は順序依存するので安定したそれなりの値として



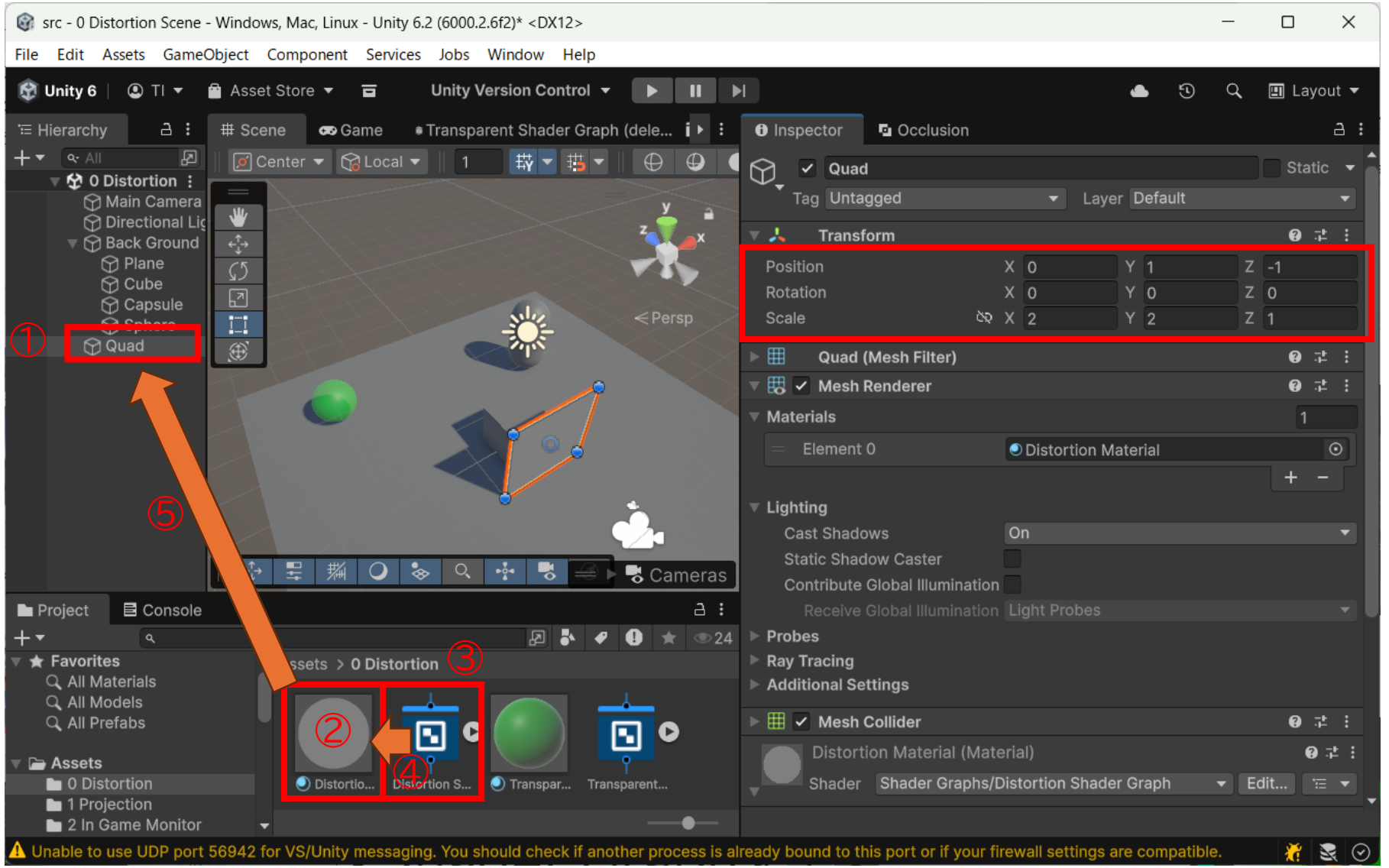
カラーバッファの保存

- カメラの設定で有効にする
  - 方法1.「Camera」-「Rendering」-「Opaque Texture」
  - 方法2. スクリプタブルレンダーパイプラインアセットで設定
    - 今回はこちらは行わない



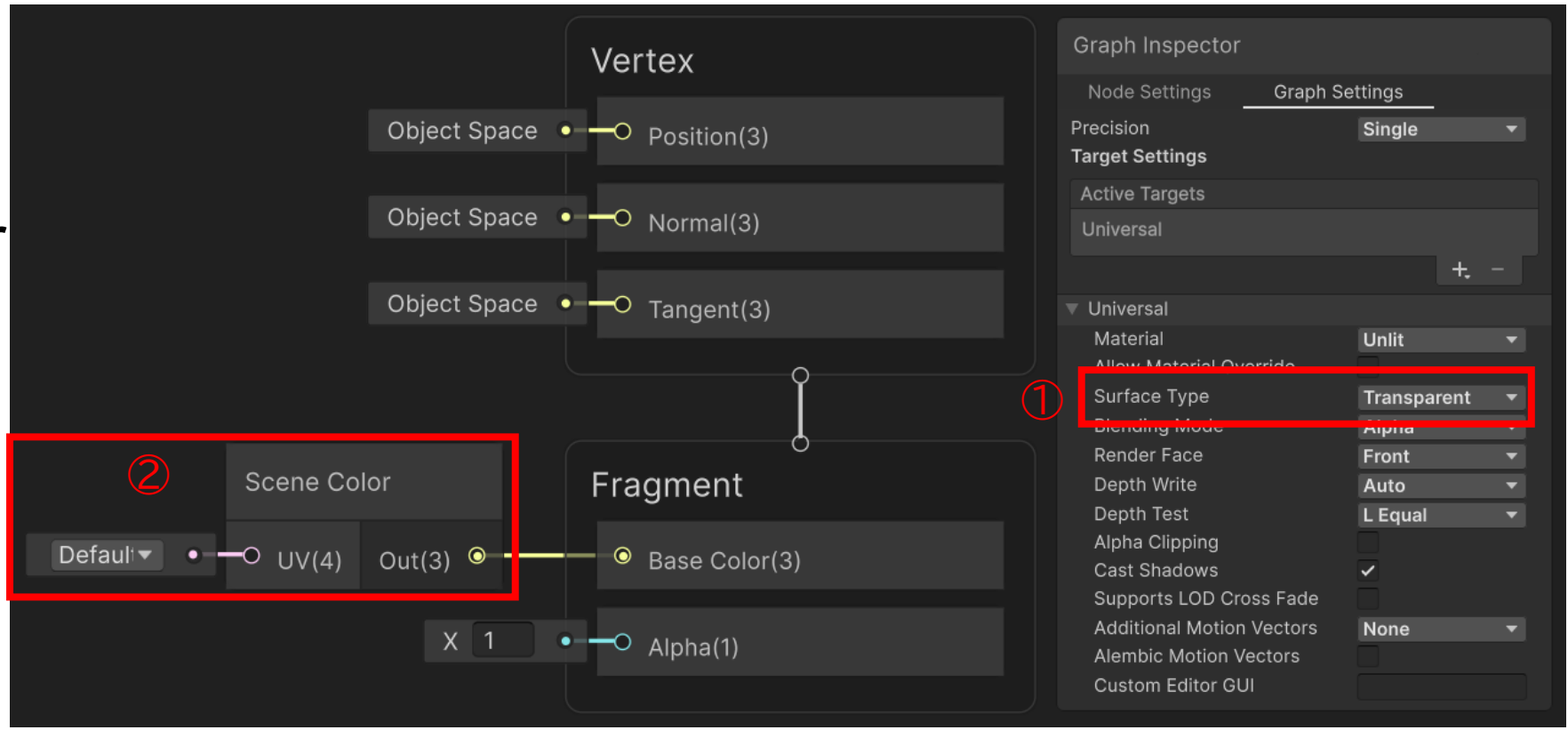
表示物の配置

1. Quadの追加
   - 位置: (0,1,-1)
   - 拡縮: (2,2,1)
2. マテリアル追加
   - 名称例:Distortion Material
3. Shader Graph追加
   - 名称例:Distortion Shader Graph
4. Shader Graphをマテリアルに設定
5. マテリアルをQuadに設定

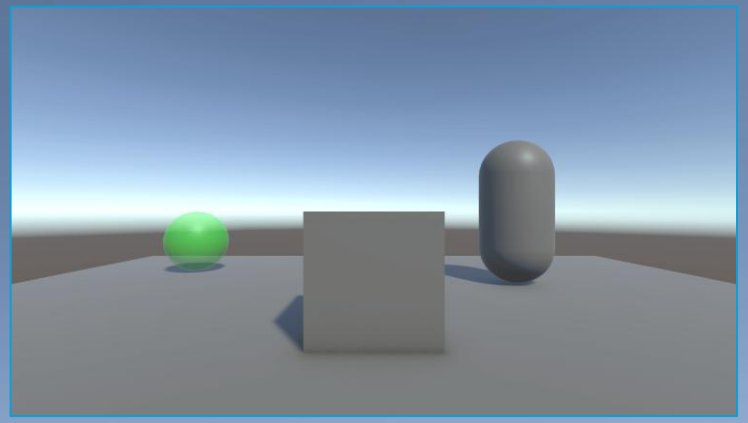


シェーダグラフ

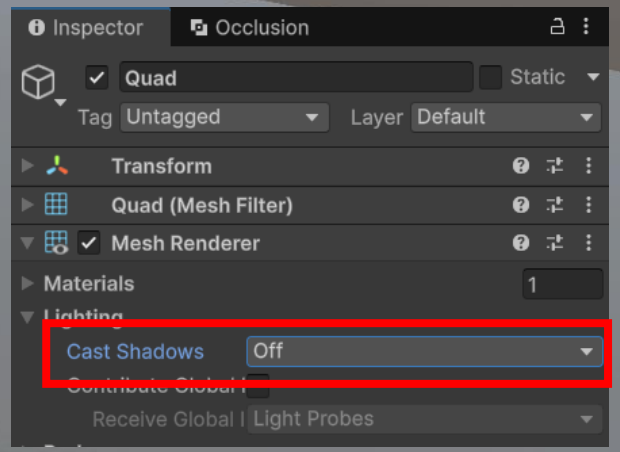
1. 半透明描画
   - でないと
     フレームバッファ
     が読み取れない
2. Scene Color
   ノードの追加



ひとまず完成？



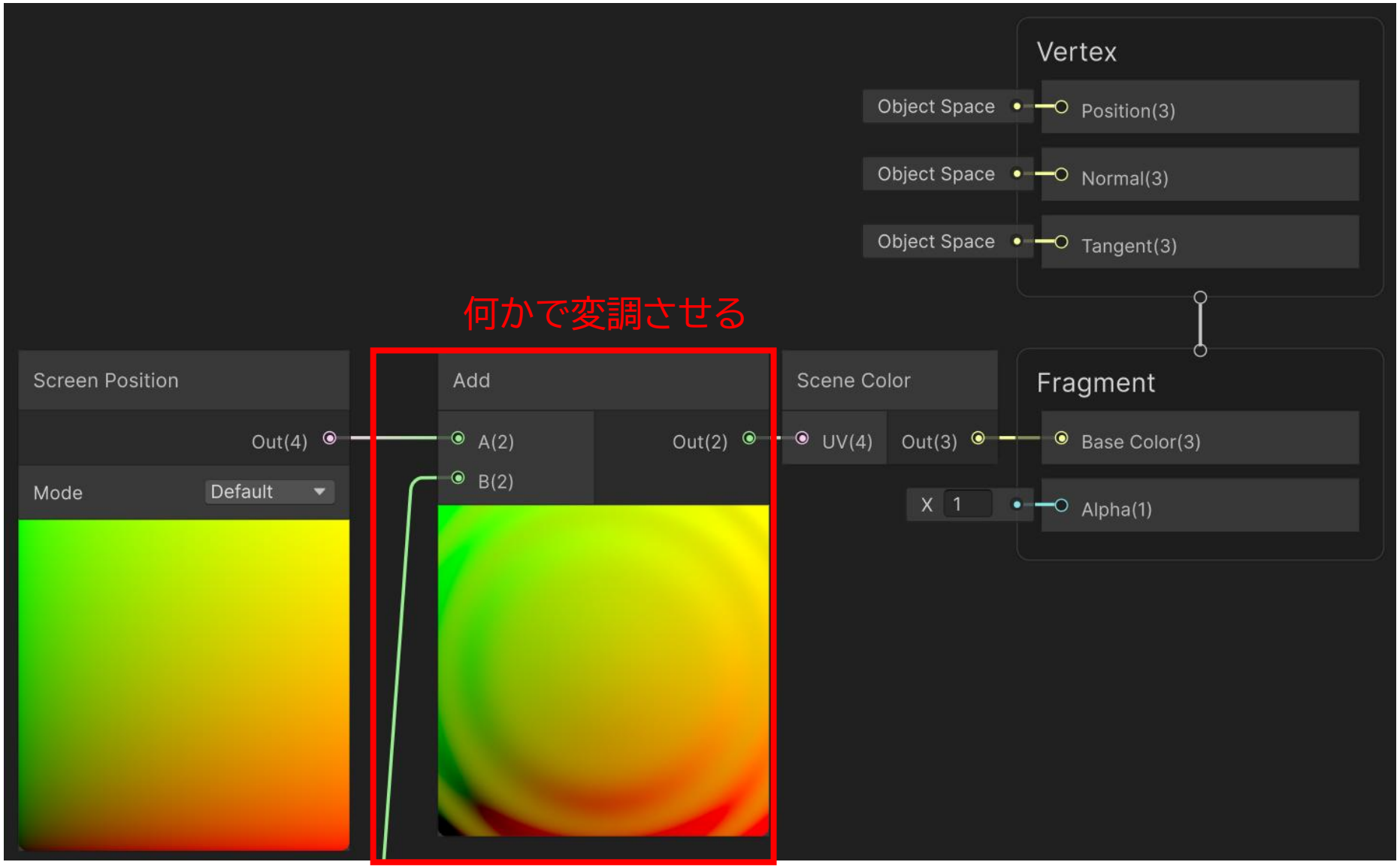
半透明描画の一部が消えている



影が増えている（QuadのCast Shadowsをオフにしよう）

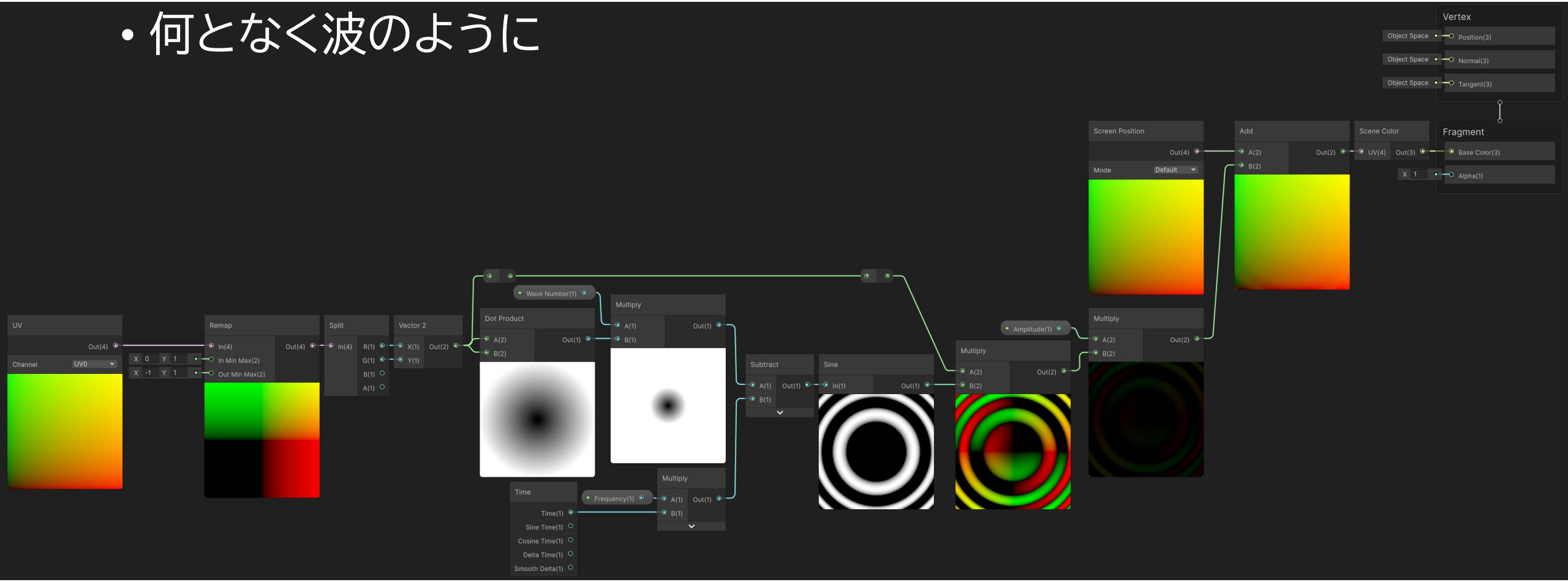
ゆがめる

- シーンカラーで読み取る位置を画素ごとにずらす



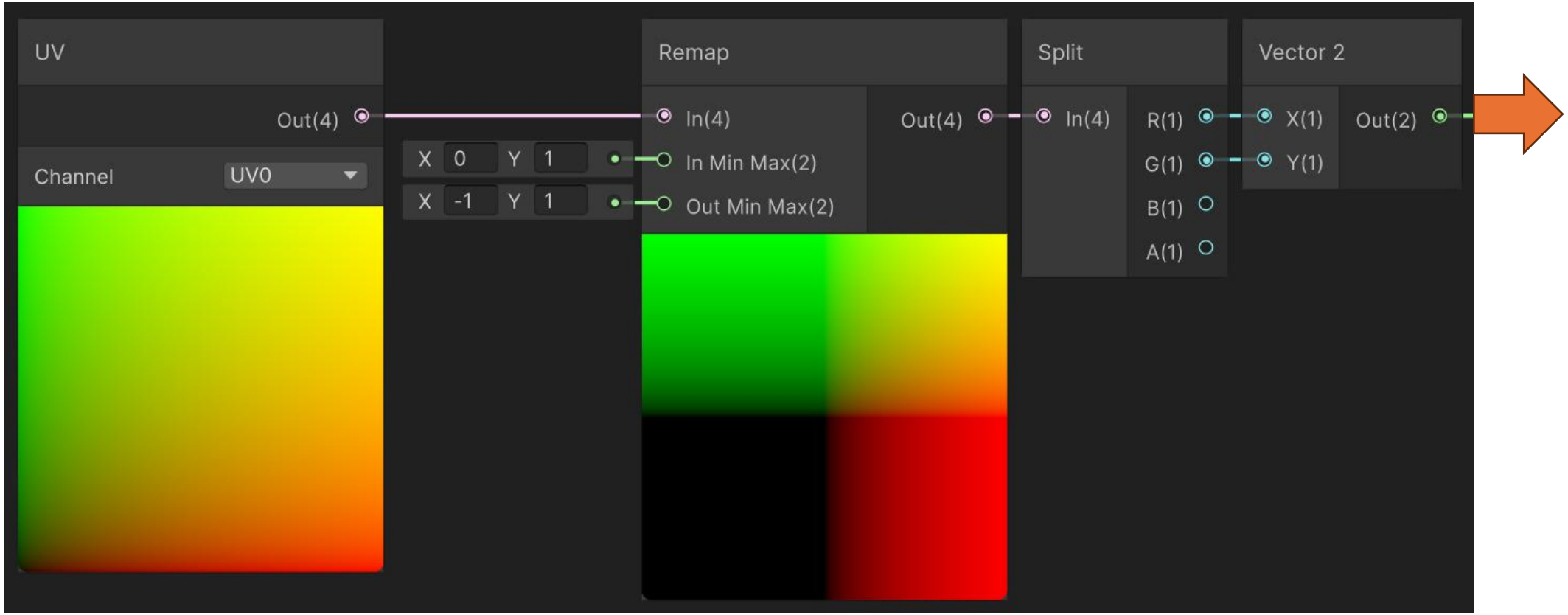
シェーダグラフを書き換える

- 何となく波のように

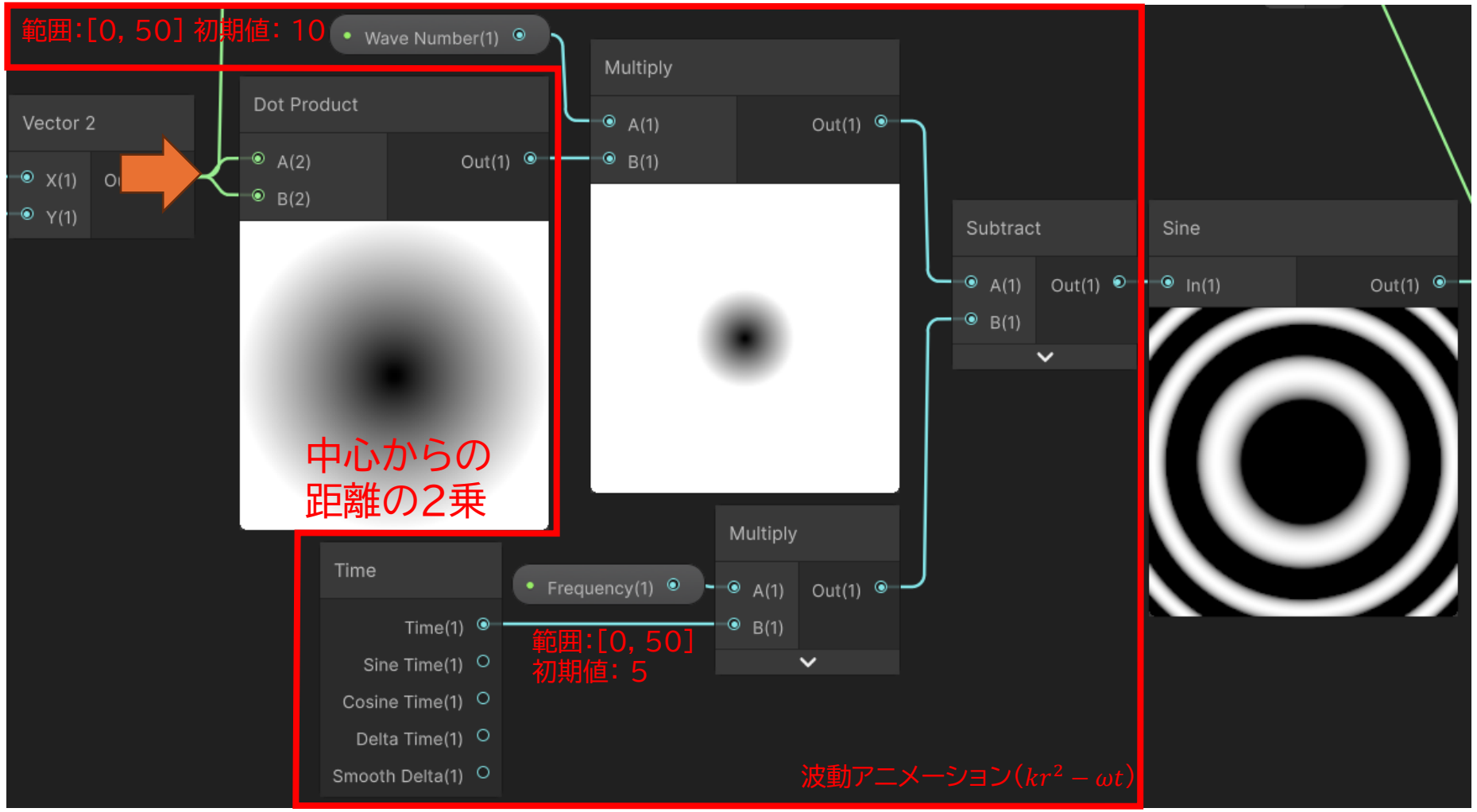


オブジェクトの真ん中を中心として演出

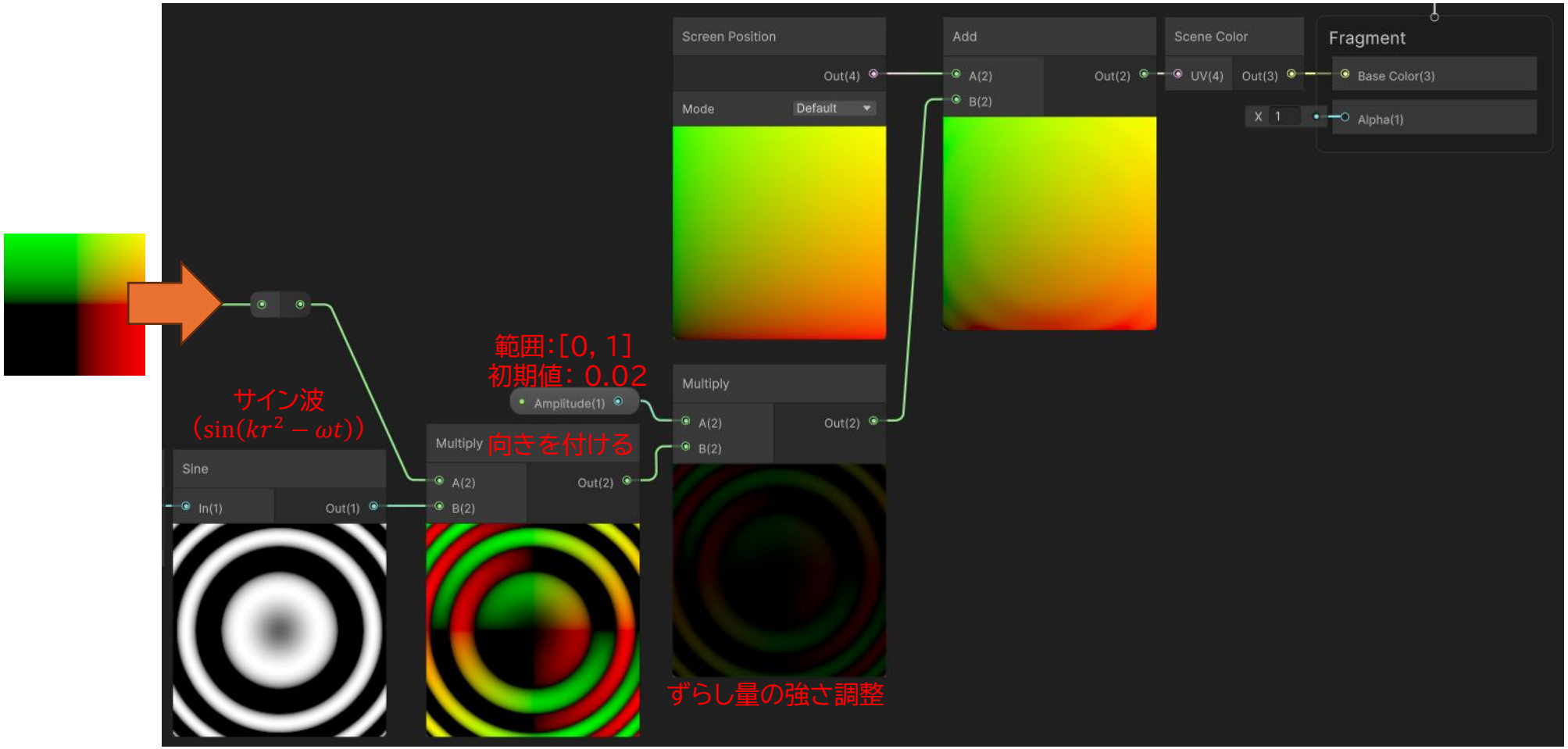
- テクスチャ座標を0.5だけずらす



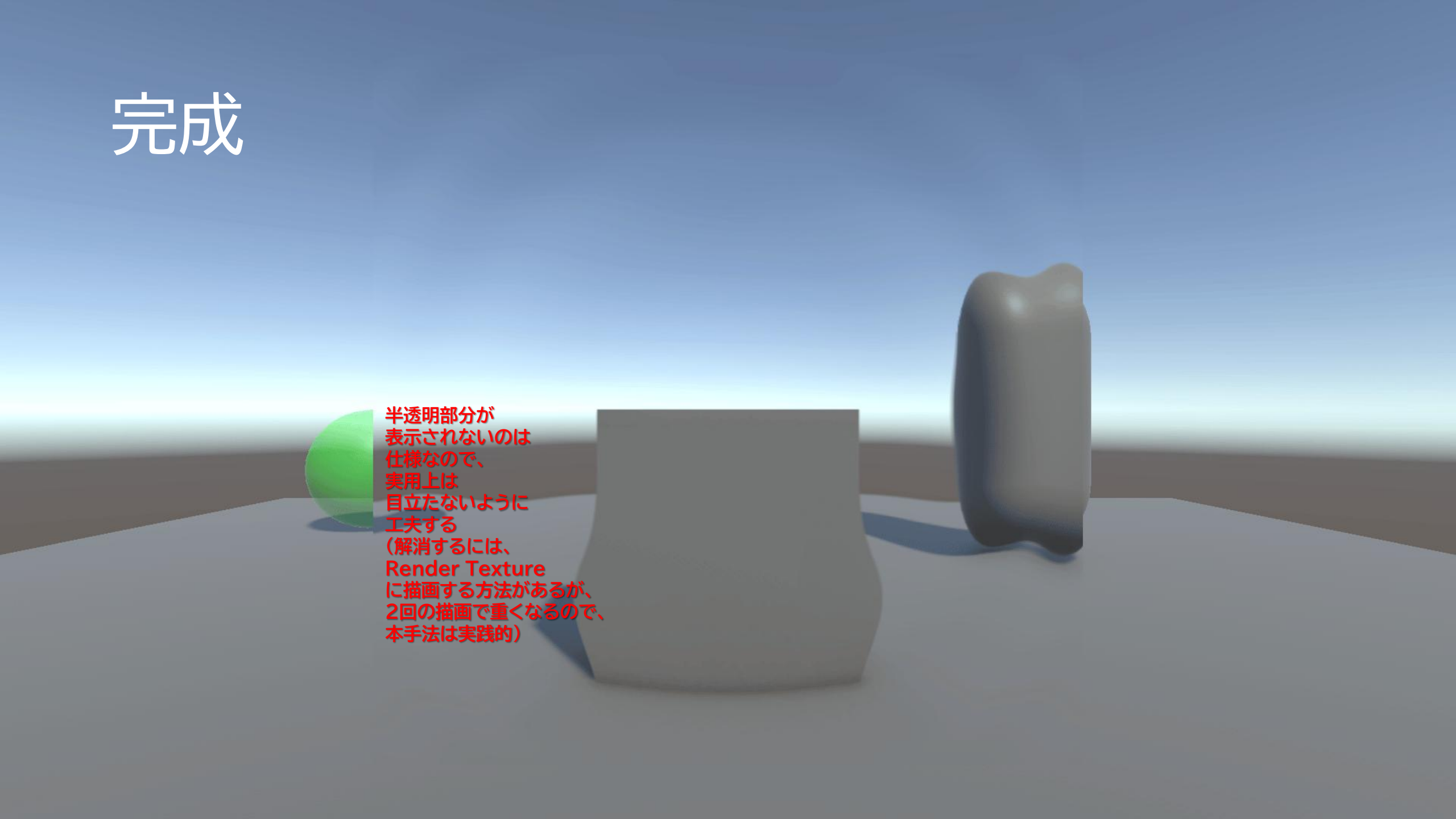
中心からの距離(の2乗)でアニメするsin



サンプリング点の変調

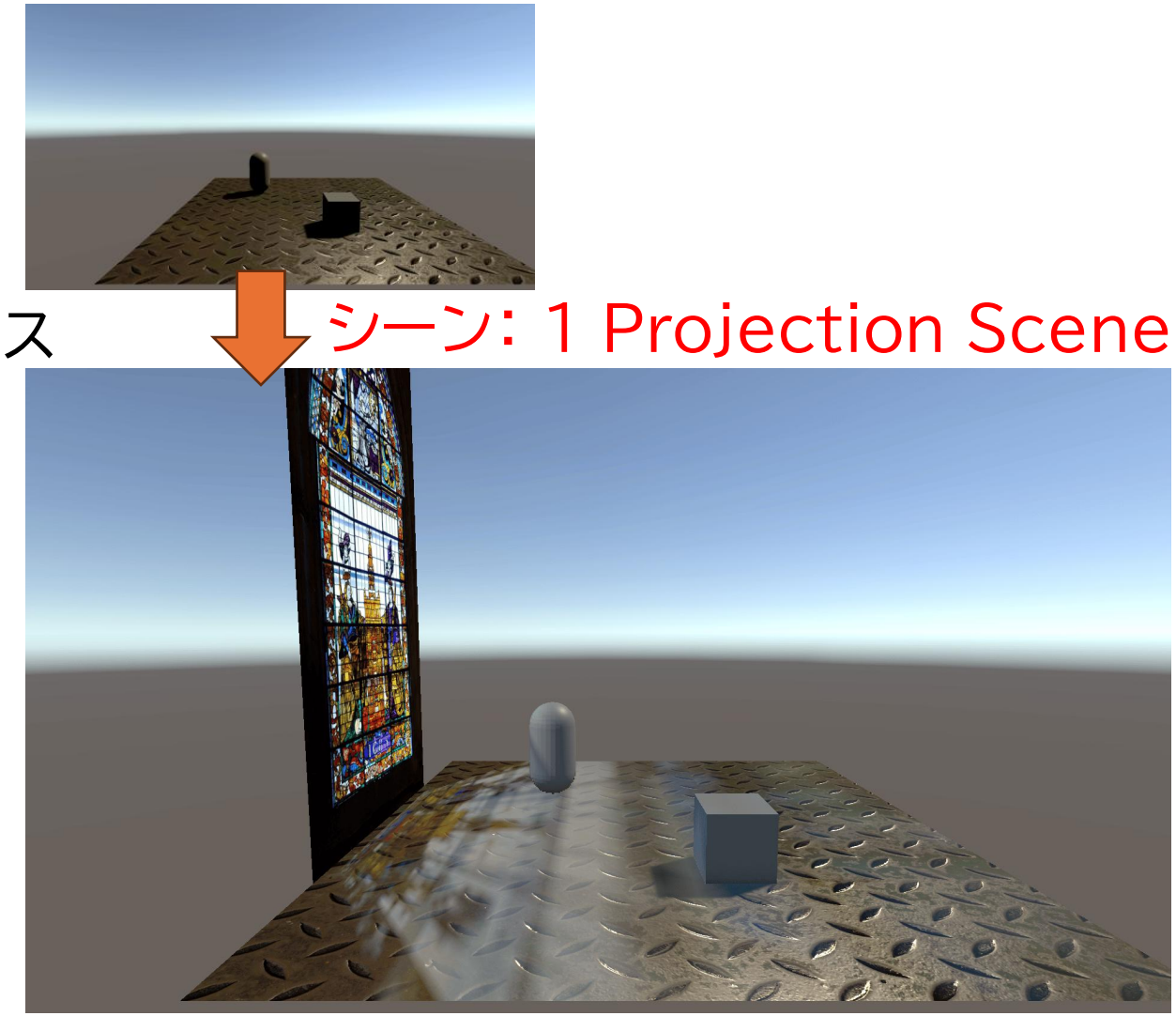

Screen Position
Out(4)
Mode
Default
Add
A(2)
B(2)
Out(2)
Scene Color
UV(4)
Out(3)
Fragment
Base Color(3)
X 1
Alpha(1)
範囲:[0, 1]
初期値: 0.02
Amplitude(1)
Multiply
A(2)
B(2)
Out(2)
サイン波
$(\sin(kr^2 - \omega t))$
Multiply 向きを付ける
Sine
In(1)
Out(1)
A(2)
B(2)
Out(2)
ずらし量の強さ調整

完成


半透明部分が
表示されないのは
仕様なので、
実用上は
目立たないように
工夫する
（解消するには、
Render Texture
に描画する方法があるが、
2回の描画で重くなるので、
本手法は実践的）

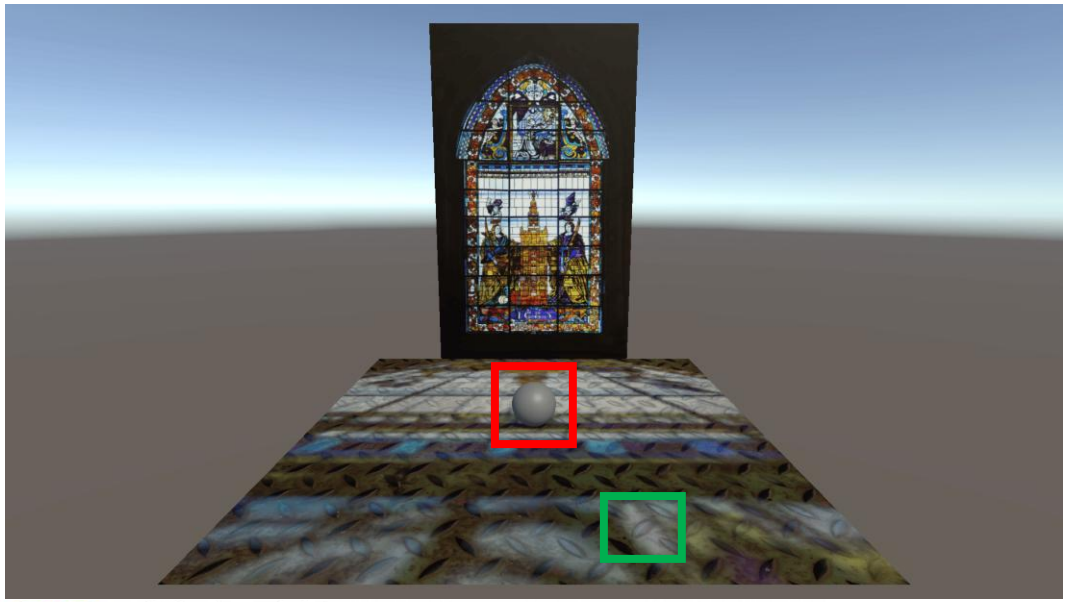
本日の内容

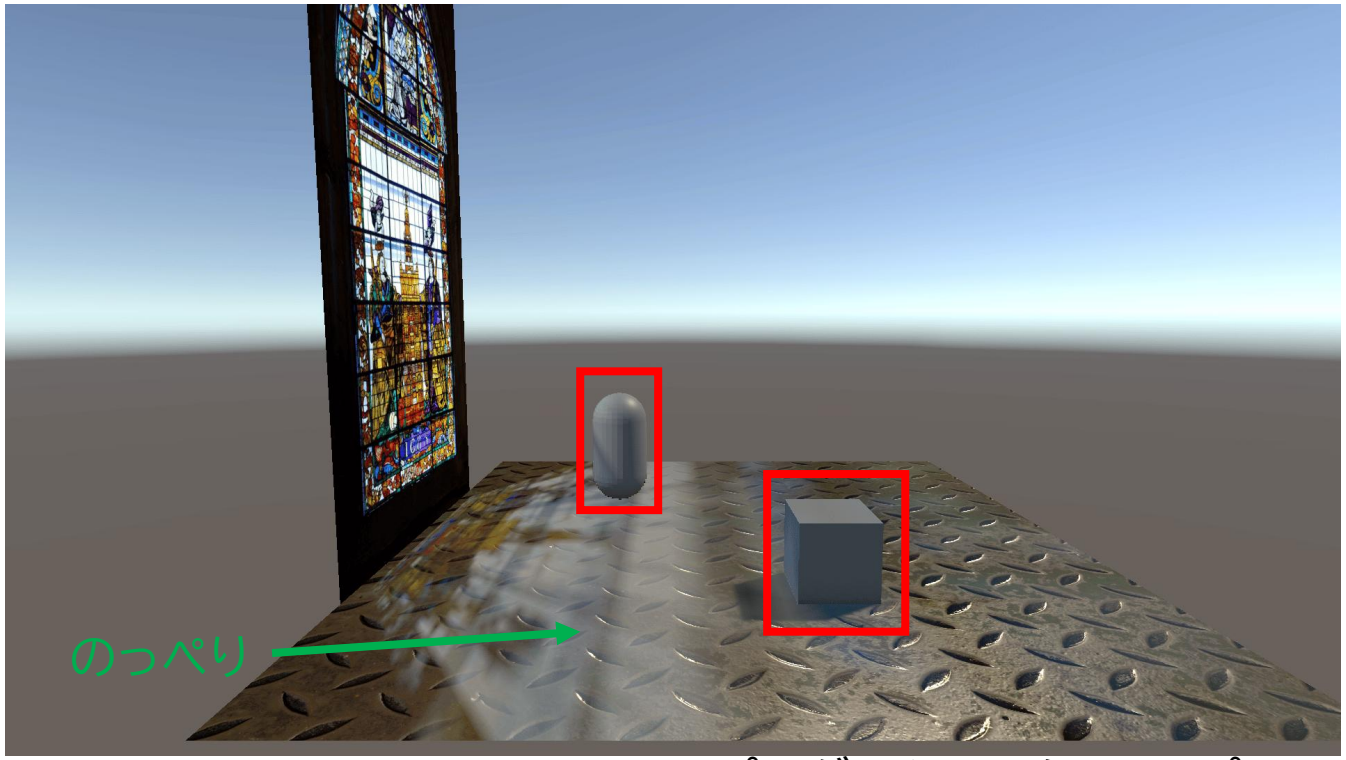
- レンダーターゲット
  - レンダーターゲットの概要
  - 不透明フレームバッファへのアクセス
  - 深度からの位置・法線の復元
- レンダーテクスチャ
  - レンダーテクスチャの概要
  - ゲーム内モニター
  - バックミラー
  - 範囲内のオブジェクトだけ単色



以前との違い

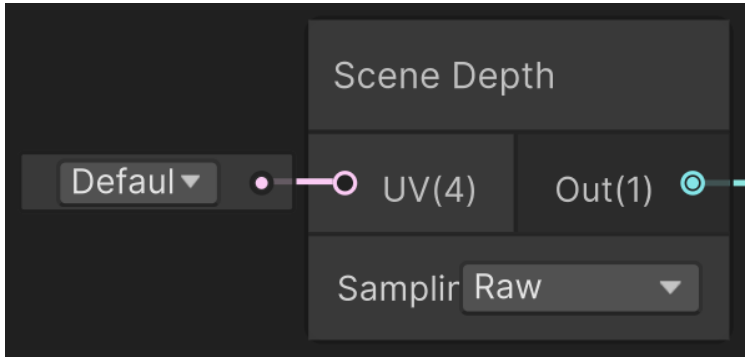
- 他のオブジェクトに投影されていなかった
  - 描画時ではなく、後から投影を重ねるのですべてに投影される
    - 背面には投影されない
- AOマップ等は反映されない

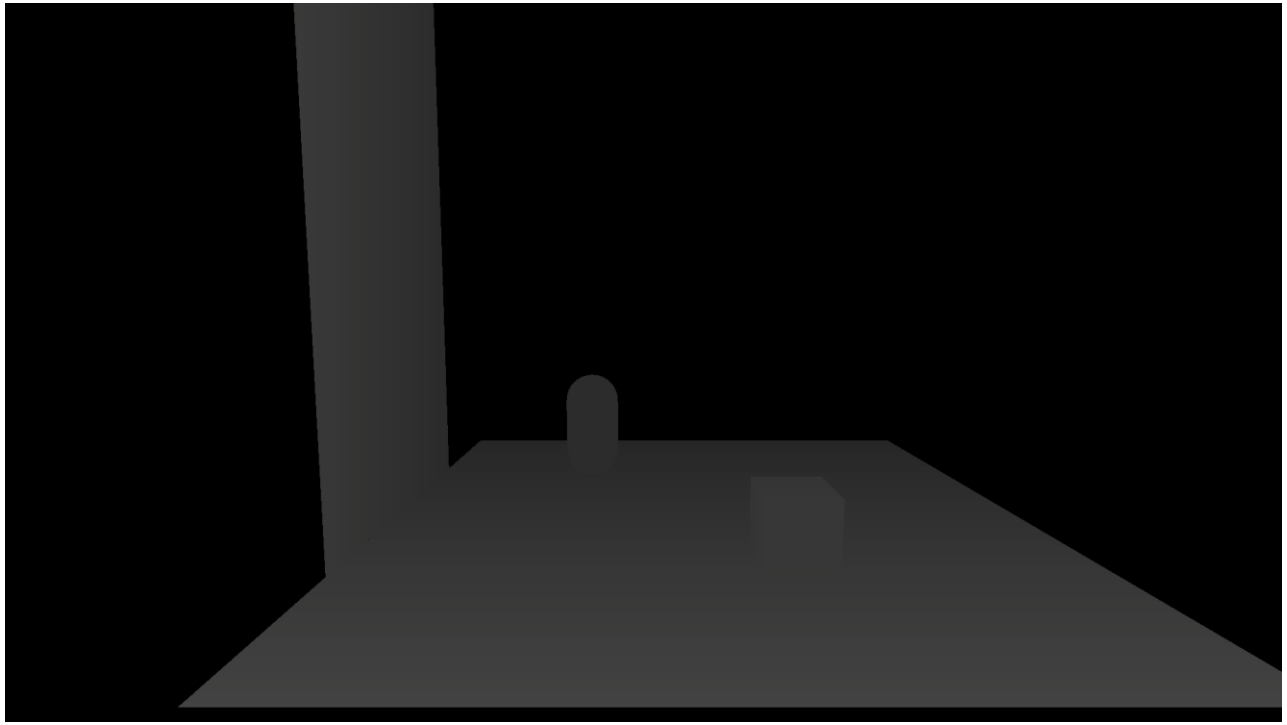


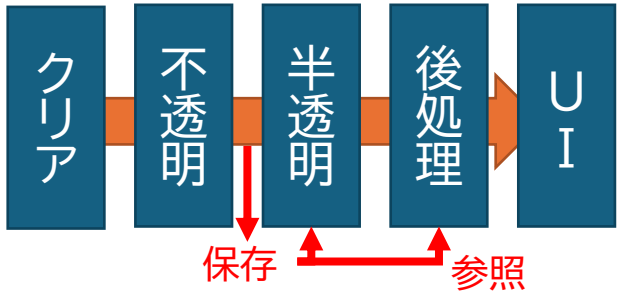


深度バッファ

- 描画するポリゴンの奥行き値を記録
  - 現代的な実装：手前は1で奥は0（Reversed-Z）
- 「Scene Depth」ノード
  - 不透明描画後の値







深度からワールド座標の復元

- 座標変換

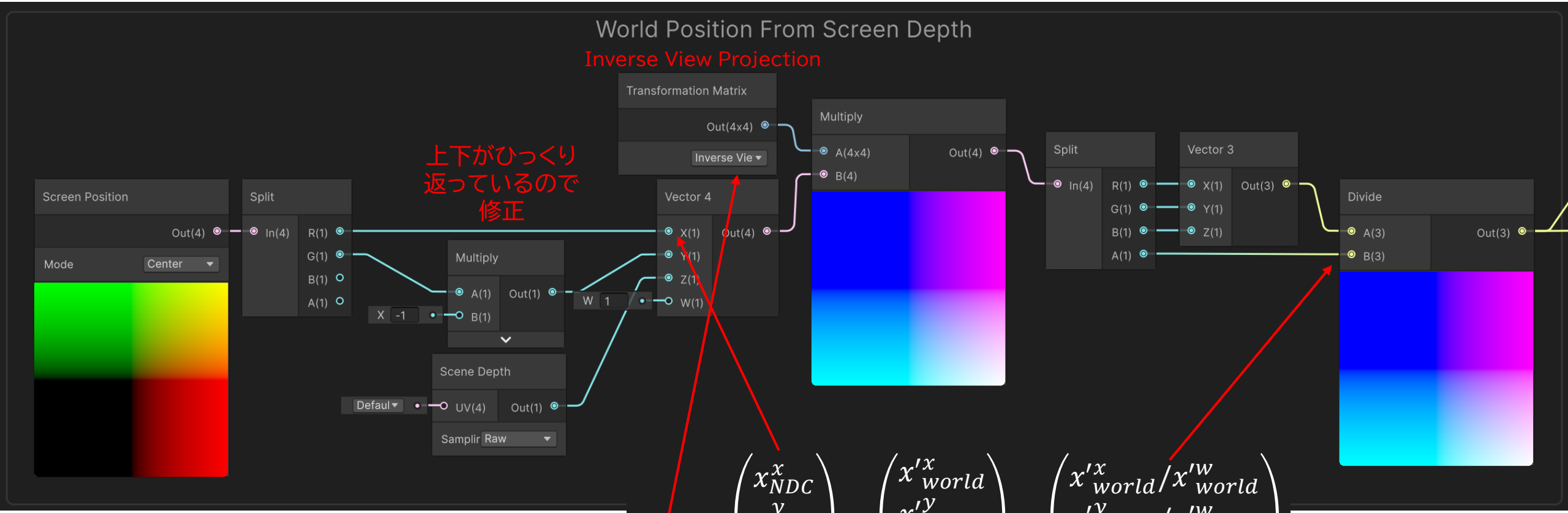
$$\begin{pmatrix} x^x_{NDC} \\ x^y_{NDC} \\ x^z_{NDC} \\ 1 \end{pmatrix} = \begin{pmatrix} x^x_{clip}/x^w_{clip} \\ x^y_{clip}/x^w_{clip} \\ x^z_{clip}/x^w_{clip} \\ 1 \end{pmatrix} \equiv \begin{pmatrix} x^x_{clip} \\ x^y_{clip} \\ x^z_{clip} \\ x^w_{clip} \end{pmatrix} = PV \begin{pmatrix} x^x_{world} \\ x^y_{world} \\ x^z_{world} \\ 1 \end{pmatrix} = PVW \begin{pmatrix} x^x_{model} \\ x^y_{model} \\ x^z_{model} \\ 1 \end{pmatrix}$$

x_{NDC}: 正規化デバイス座標系（X,Yの範囲が[-1, +1], Zの範囲が[1.0, 0.0]（Reversed-Z））
この座標系のZ値が深度として記録される（Scene DepthノードのRaw）

- ワールド座標値は逆変換として再構築できる

$$(PV)^{-1} \begin{pmatrix} x^x_{NDC} \\ x^y_{NDC} \\ x^z_{NDC} \\ 1 \end{pmatrix} = \begin{pmatrix} x'^x_{world} \\ x'^y_{world} \\ x'^z_{world} \\ x'^w_{world} \end{pmatrix} \equiv \begin{pmatrix} x'^x_{world}/x'^w_{world} \\ x'^y_{world}/x'^w_{world} \\ x'^z_{world}/x'^w_{world} \\ 1 \end{pmatrix}$$

深度バッファからワールド座標値の復元

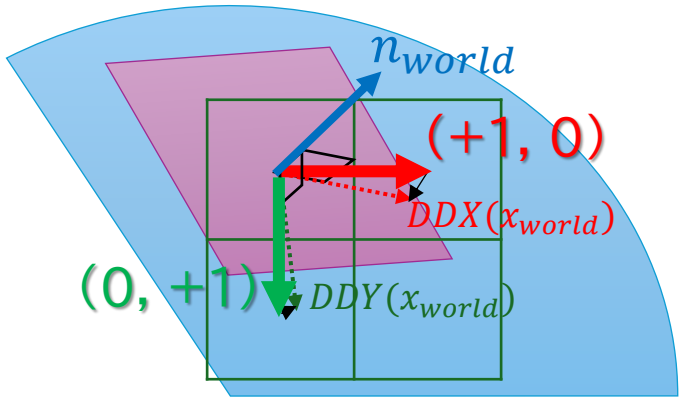


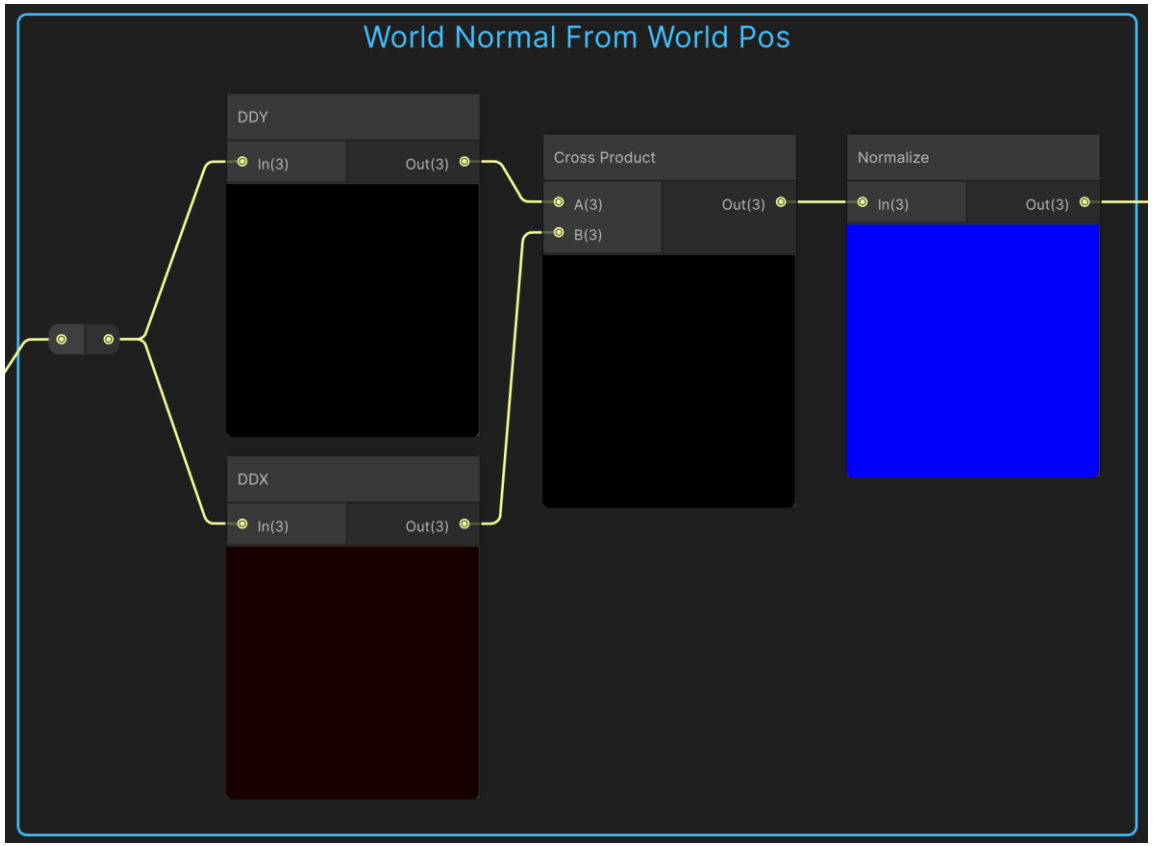
$$(PV)^{-1}\begin{pmatrix} x^{x}_{NDC} \\ x^{y}_{NDC} \\ x^{z}_{NDC} \\ 1 \end{pmatrix} = \begin{pmatrix} x'^{x}_{world} \\ x'^{y}_{world} \\ x'^{z}_{world} \\ x'^{w}_{world} \end{pmatrix} \equiv \begin{pmatrix} x'^{x}_{world}/x'^{w}_{world} \\ x'^{y}_{world}/x'^{w}_{world} \\ x'^{z}_{world}/x'^{w}_{world} \\ 1 \end{pmatrix}$$

ワールド座標から法線の復元

- DDX, DDY：隣接ピクセルへの値の変化
  - ワールド座標値のDDX,DDYは、ピクセル間隔程度の粒度での接平面の節ベクトルを与える
    - 2つの節ベクトルの外積を正規化したベクトルは法線ベクトル

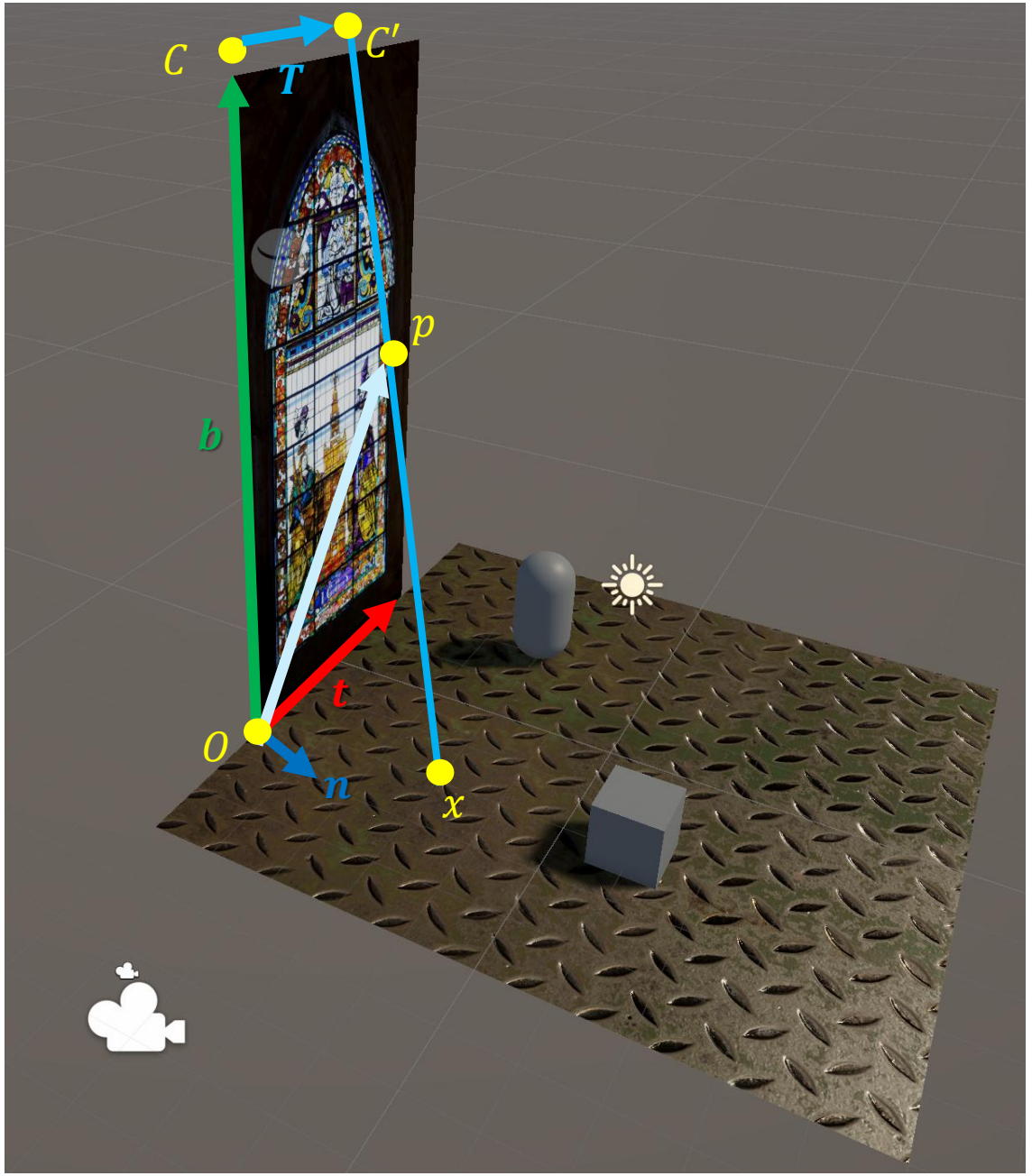
$$n_{world} \propto DDX(x_{world}) \times DDY(x_{world})$$





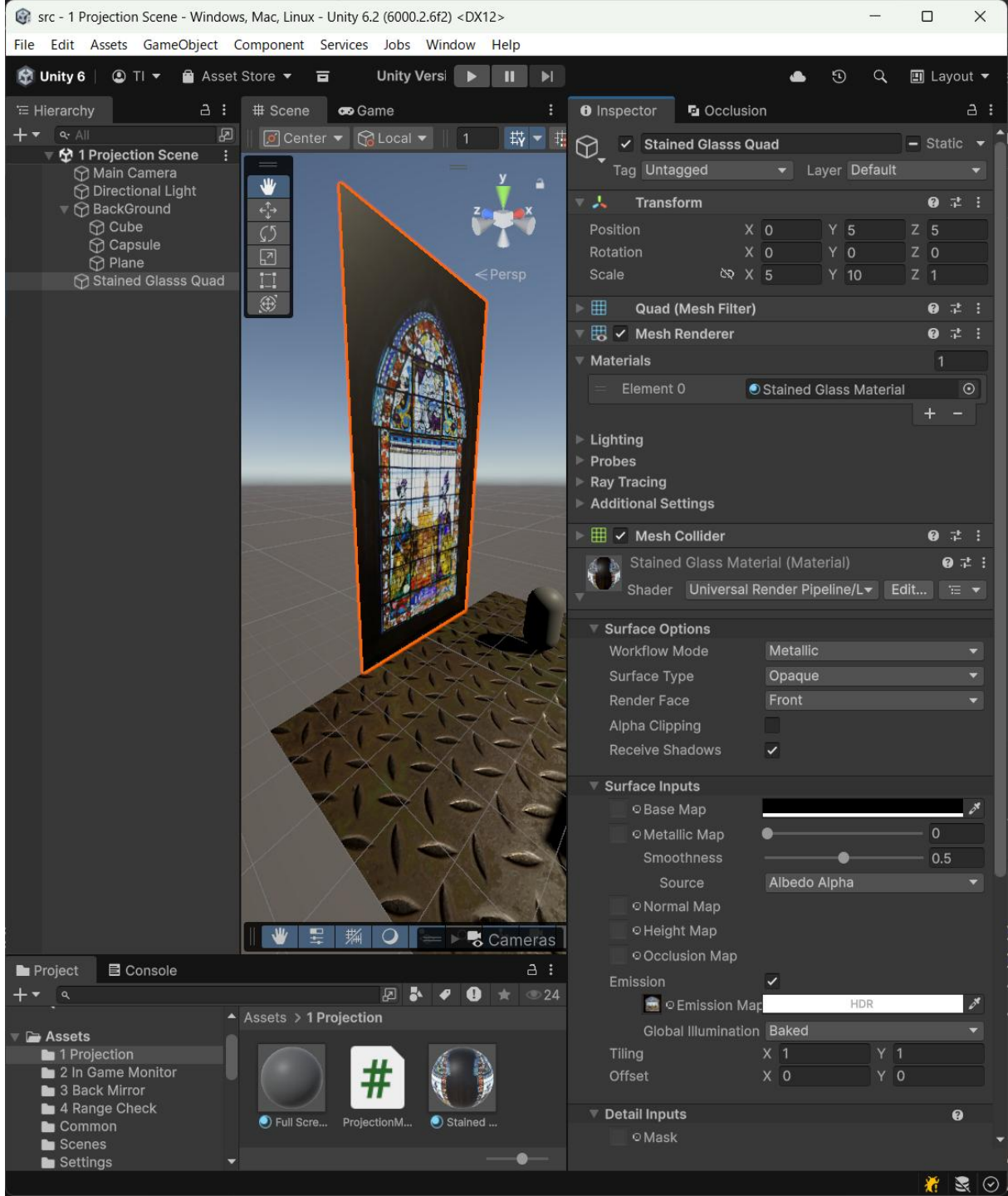
平面投影

- ライトの移動
  - $C' = C + T$
- ライトの位置C'から描画点Xへの線分のステンドグラスとの交点p
  - $p = x + \frac{(o-x)\cdot n}{(C'-x)\cdot n}(C' - x)$
  - $n = normalized(t \times b)$
- サンプリングするテクスチャ座標
  - $(u, v) = \left(\frac{(p-o)\cdot t}{t\cdot t}, \frac{(p-o)\cdot b}{b\cdot b}\right) = (p - o)\cdot\left(\frac{t}{\|t\|^2}, \frac{b}{\|b\|^2}\right)$



ステンドグラス オブジェクト

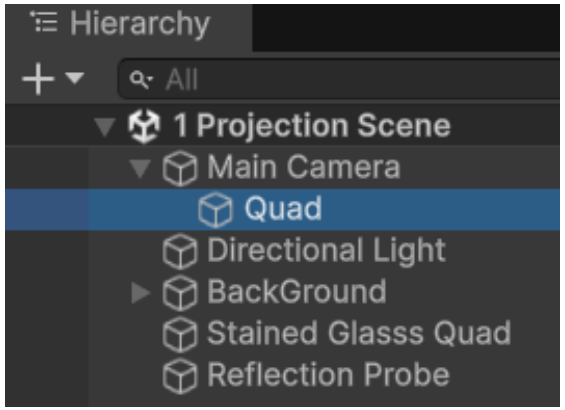
- Quadを追加
  - 名称例:Stained Glasss Quad
- 部屋の端に追加
  - ここでは、大きさ:(5,10,1)
  - 位置: (0,0,5)
- マテリアルを設定
  - Stained Glasss Material
  - ステンドグラスの画像を貼る
  - シェーディングに影響させないためには、Emissionに入れる
    - ベースカラーは黒



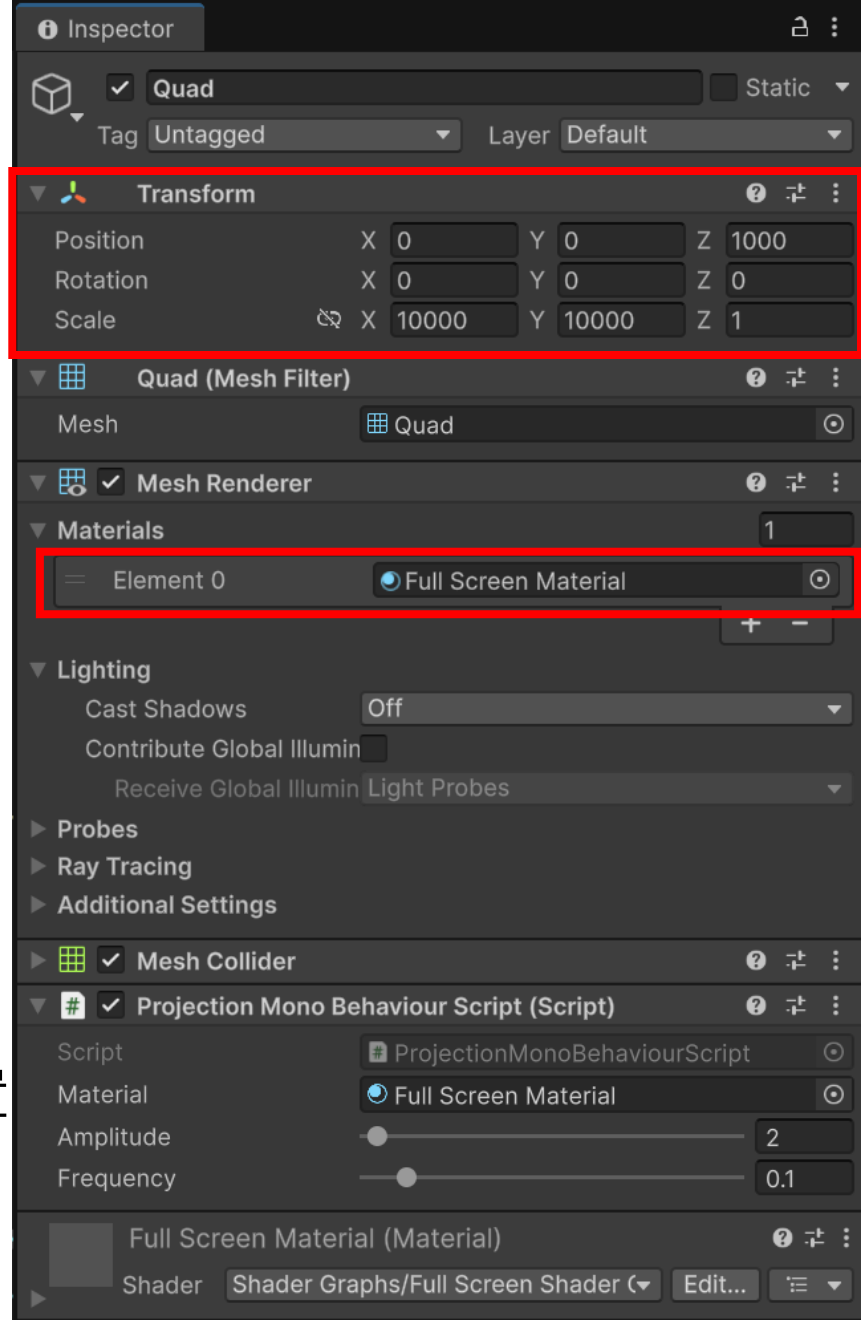
KOGEI TOKYO POLYTECHNIC UNIVERSITY

全画面描画用オブジェクト

- カメラの子供として平面を追加
  - 名称例：Quad

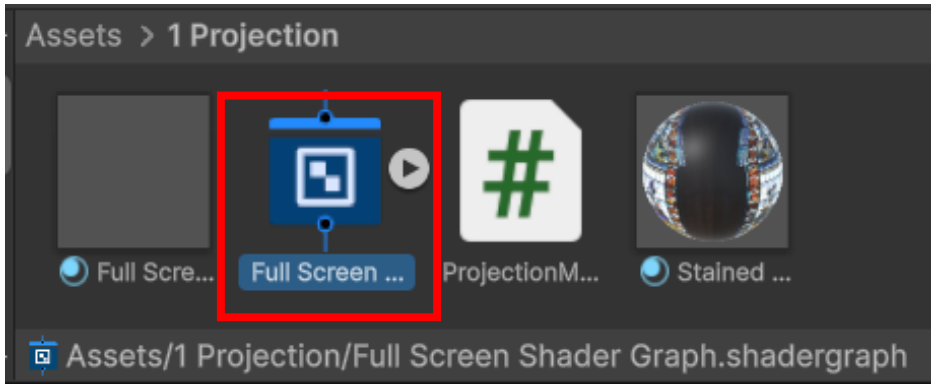


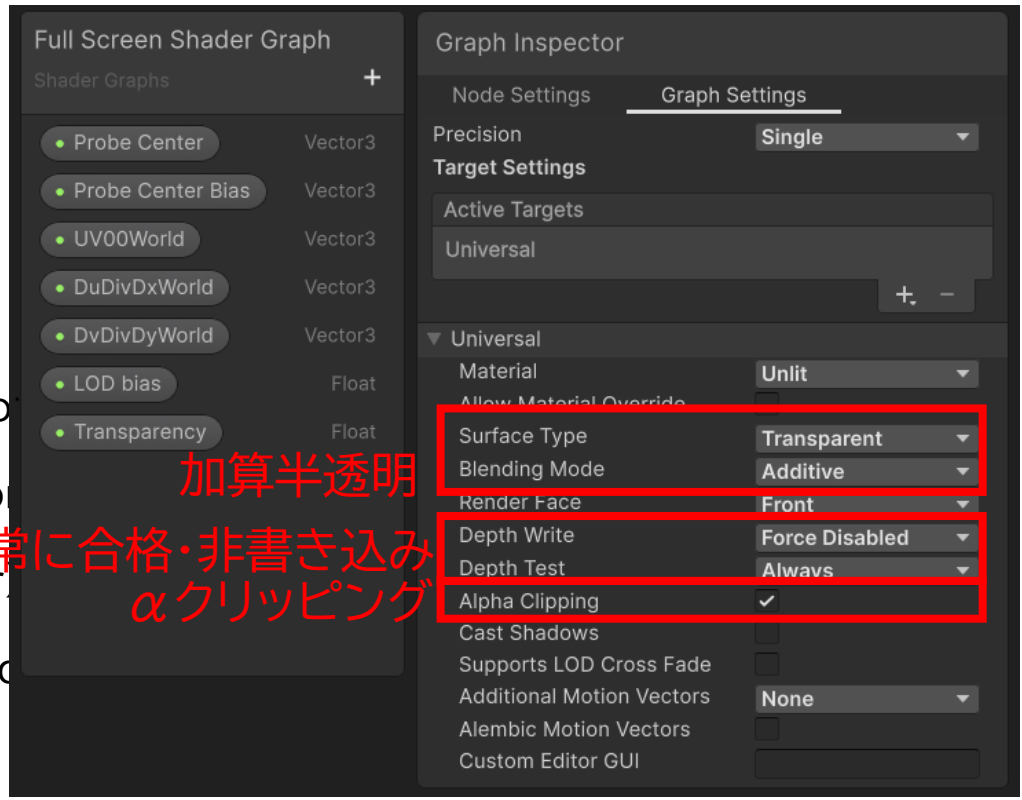
  - カメラの前に全画面を覆うように追加
    - ここでは、1000だけ前の位置に大きさ10000で配置
  - マテリアルを設定
    - 1 Projection/Full Screen Material



Shader Graph

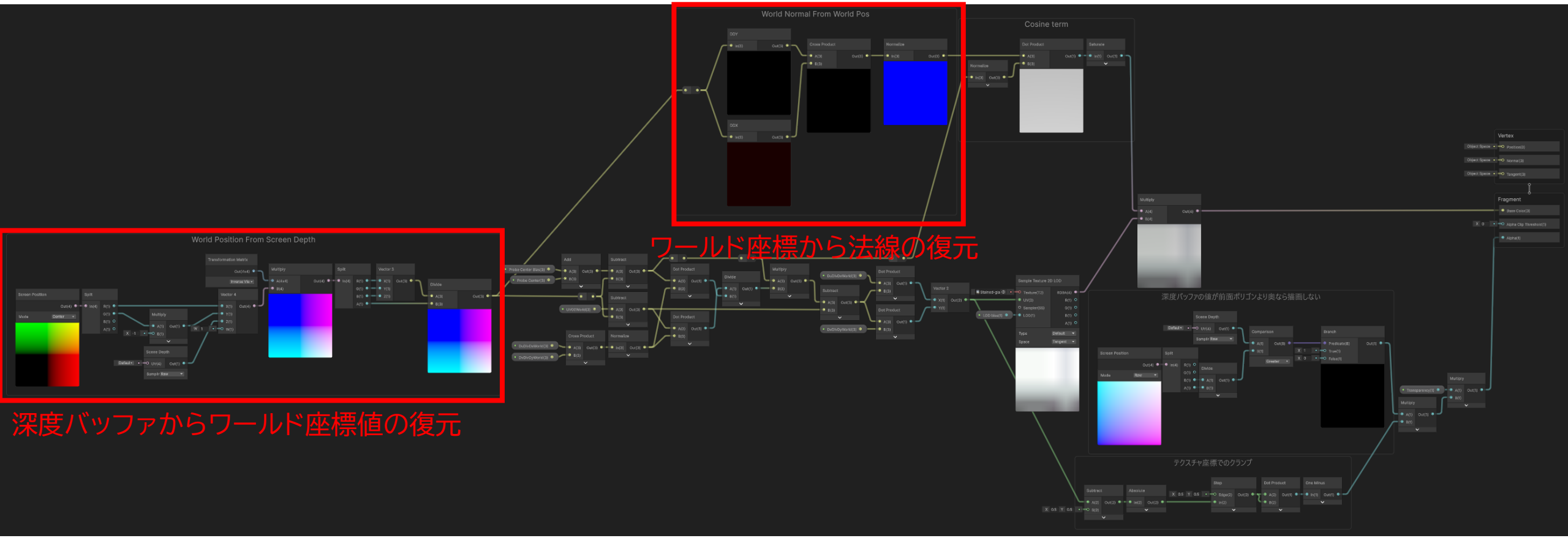
- Shader Graphを作成
  - 「1 Projection/Full Screen Material」に設定
- Shader Graphを実装(ノード構成は次頁)
  - 変数を7つ追加(次は設定例)
    - Probe Center: カメラ原点、Vector3型
      - 初期値: (0, 7, 6)
    - Probe Center Bias:カメラ移動量、Vector3型
      - 初期値: (0, 0, 0)
    - UV00World: ステンドグラスのUVの原点、Vector3型
      - 初期値: (-2.5 0, 5)
    - DuDivDxWorld:ステンドグラスのUが1になる位置の逆数、Vecto
      - 初期値: (0.2, 0, 0)
    - DvDivDyWorld:ステンドグラスのVが1になる位置の逆数、Vecto
      - 初期値: (0, 0.1, 0)
    - LOD bias:ステンドグラスの簡易ぼかし、Float型 (Mode: Slider)
      - 範囲: [0, 10], 初期値: 3
    - Transparency: 透明度(実際は負透過率)、Float型 (Mode: Slic
      - 範囲: [0, 1], 初期値: 0.3
  - 画像
    - ステンドグラスの模様 



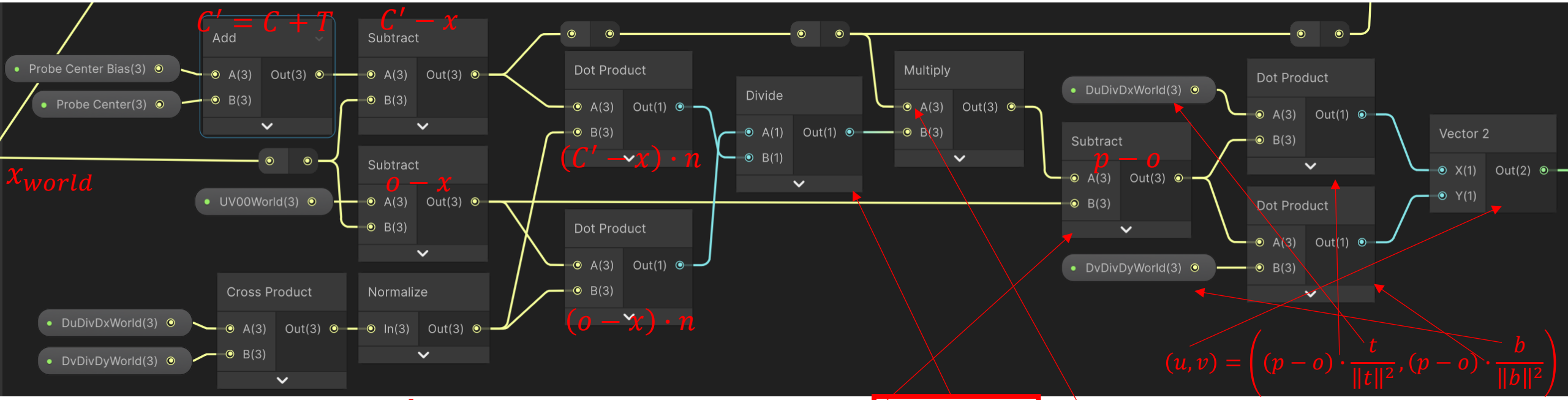


加算半透明

深度テスト常に合格・非書き込み
αクリッピング

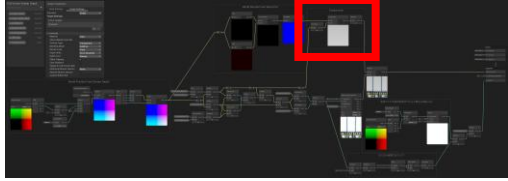

World Normal From World Pos
Cosine term
World Position From Screen Depth
ワールド座標から法線の復元
深度バッファからワールド座標値の復元
深度バッファの値が前面ポリゴンより奥なら描画しない
テクスチャ座標でのクランプ
Vertex
Fragment

テクスチャ座標値

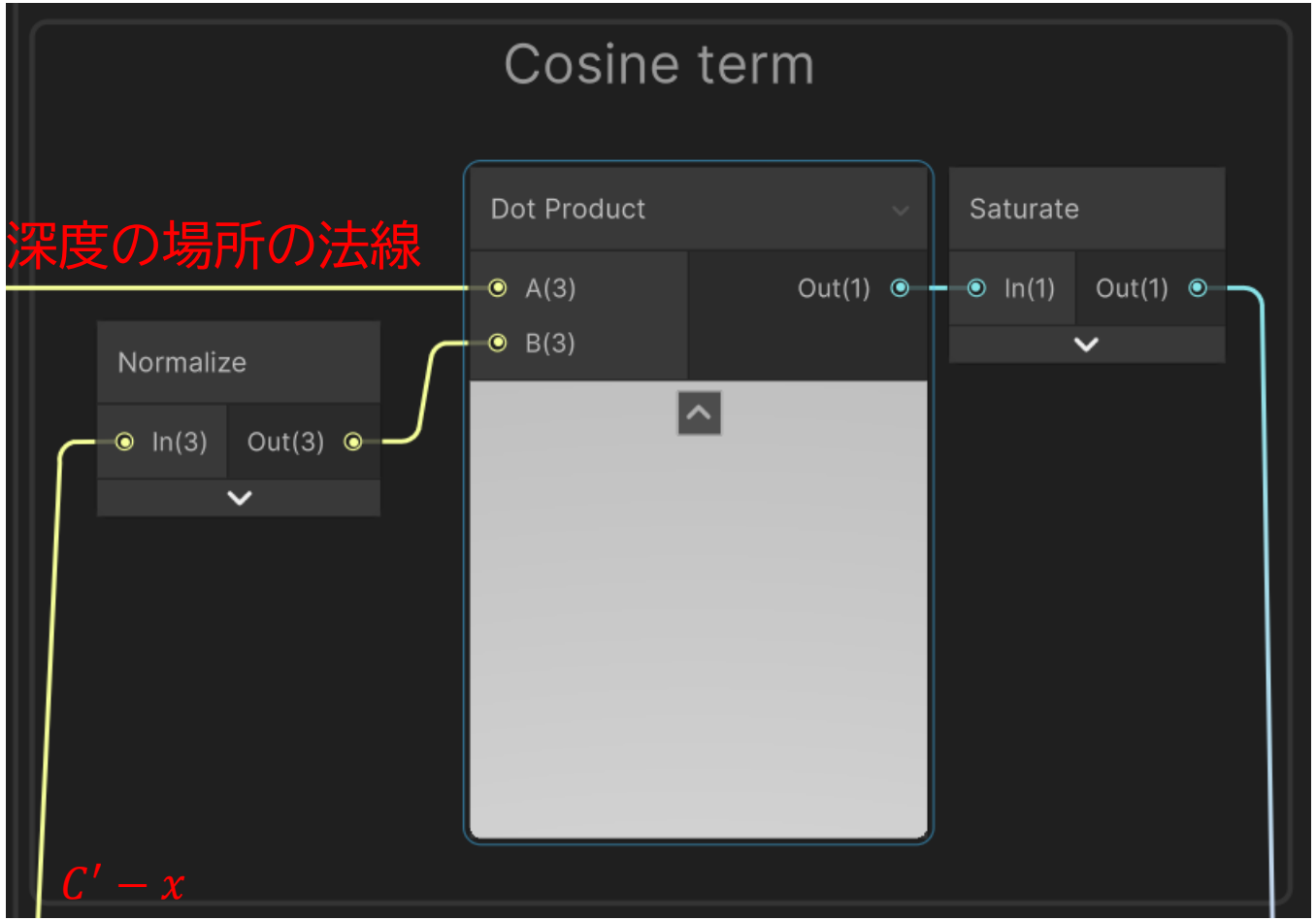


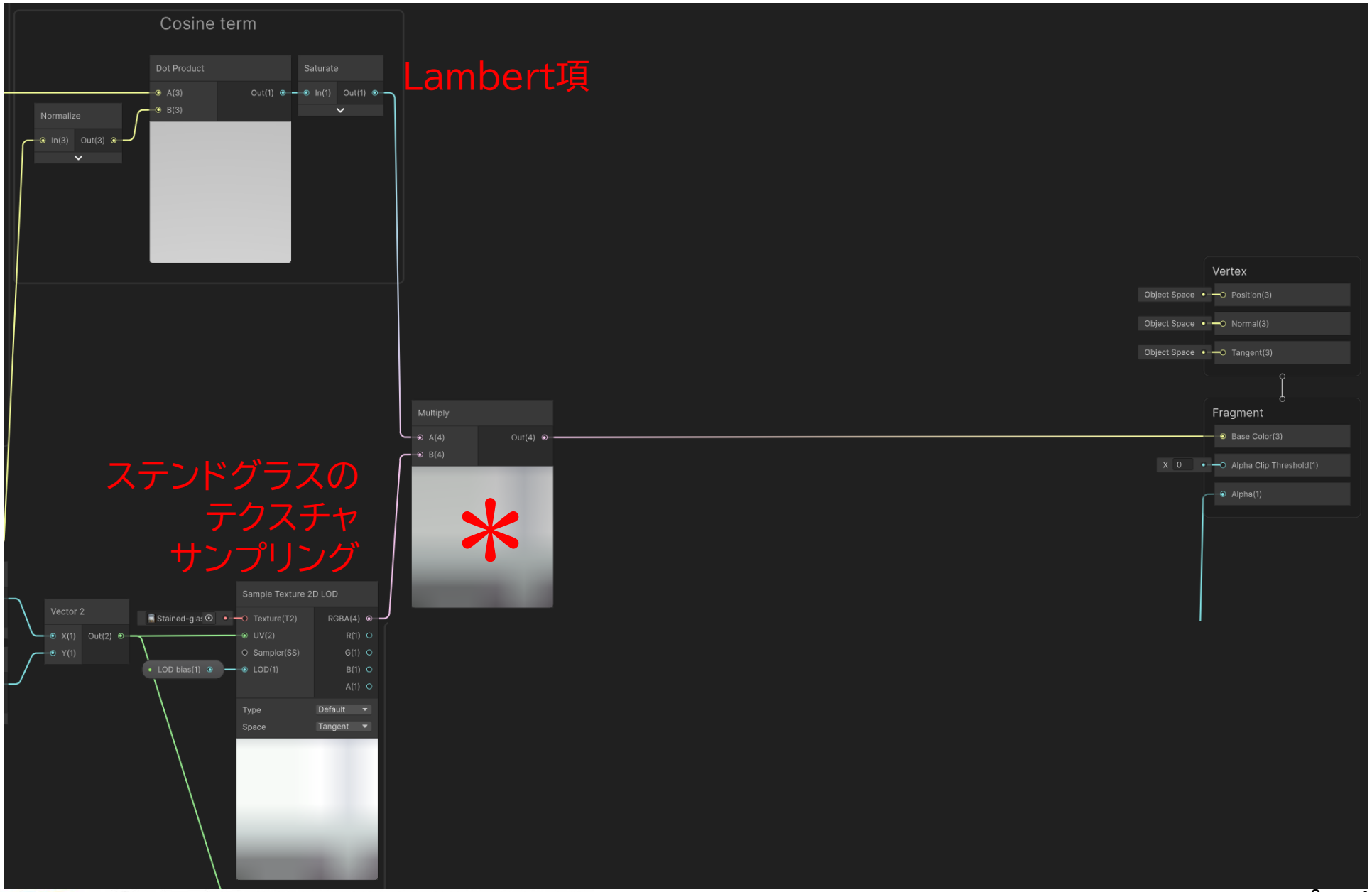
$$n = normalized(\frac{t}{\|t\|^2} \times \frac{b}{\|b\|^2})$$

$$p - o = -(o - x) + \frac{(o - x) \cdot n}{(C' - x) \cdot n}(C' - x)$$

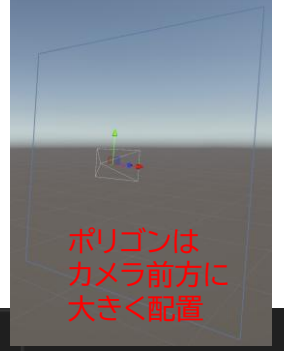


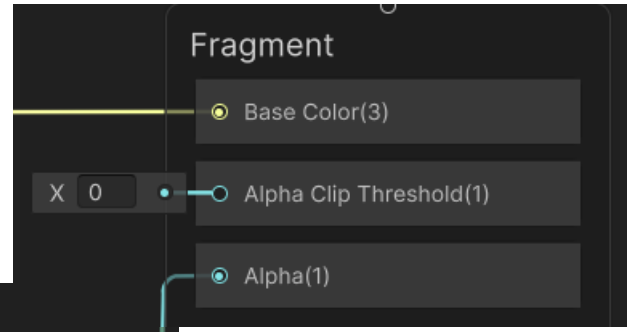
Lambert項


Cosine term
深度の場所の法線
Dot Product
A(3)
B(3)
Out(1)
Saturate
In(1)
Out(1)
Normalize
In(3)
Out(3)
$C' - x$

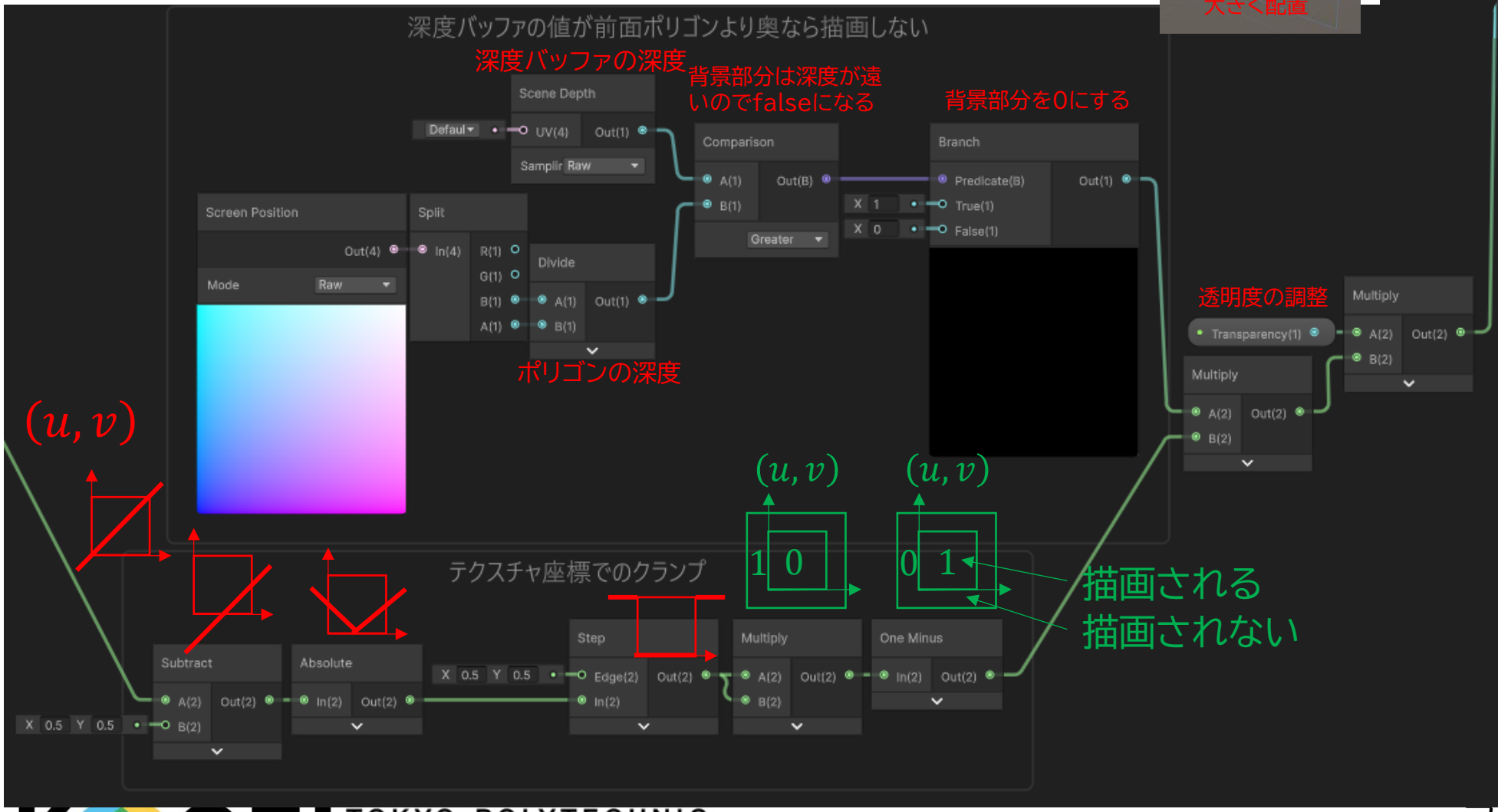

Cosine term
Dot Product
A(3)
B(3)
Out(1)
Saturate
In(1)
Out(1)
Normalize
In(3)
Out(3)
Lambert項
ステンドグラスの
テクスチャ
サンプリング
Multiply
A(4)
B(4)
Out(4)
Vertex
Object Space
Position(3)
Object Space
Normal(3)
Object Space
Tangent(3)
Fragment
Base Color(3)
X 0
Alpha Clip Threshold(1)
Alpha(1)
Vector 2
X(1)
Y(1)
Out(2)
Stained-glas
Sample Texture 2D LOD
Texture(T2)
UV(2)
Sampler(SS)
LOD(1)
RGBA(4)
R(1)
G(1)
B(1)
A(1)
LOD bias(1)
Type
Default
Space
Tangent

αクリッピング



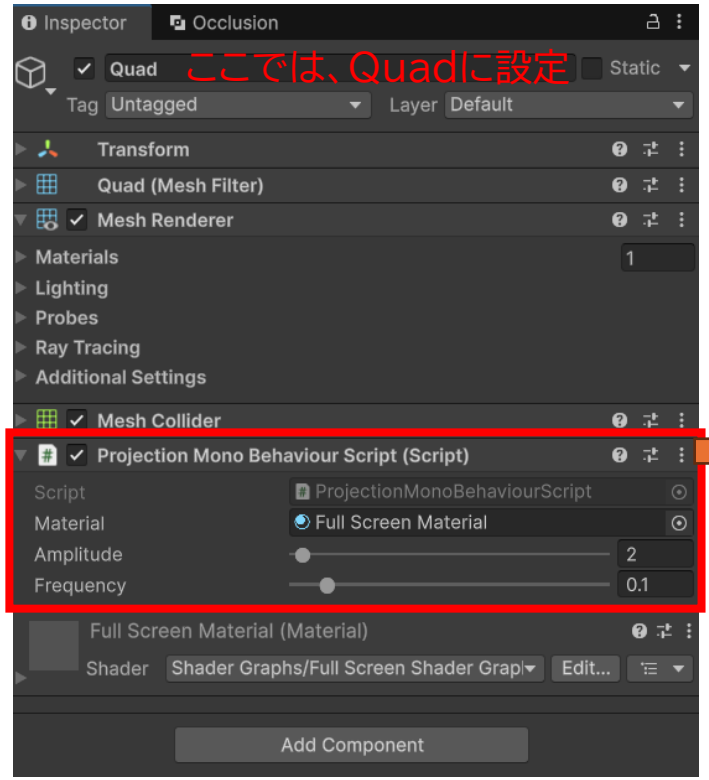


背景部分とテクスチャ座標が[0,1]に収まらない部分をクリッピングする



その他:光源を動かす

- いずれかのオブジェクトにスクリプトを追加
  - 「Full Screen Material」に設定できればどのオブジェクトでも良い



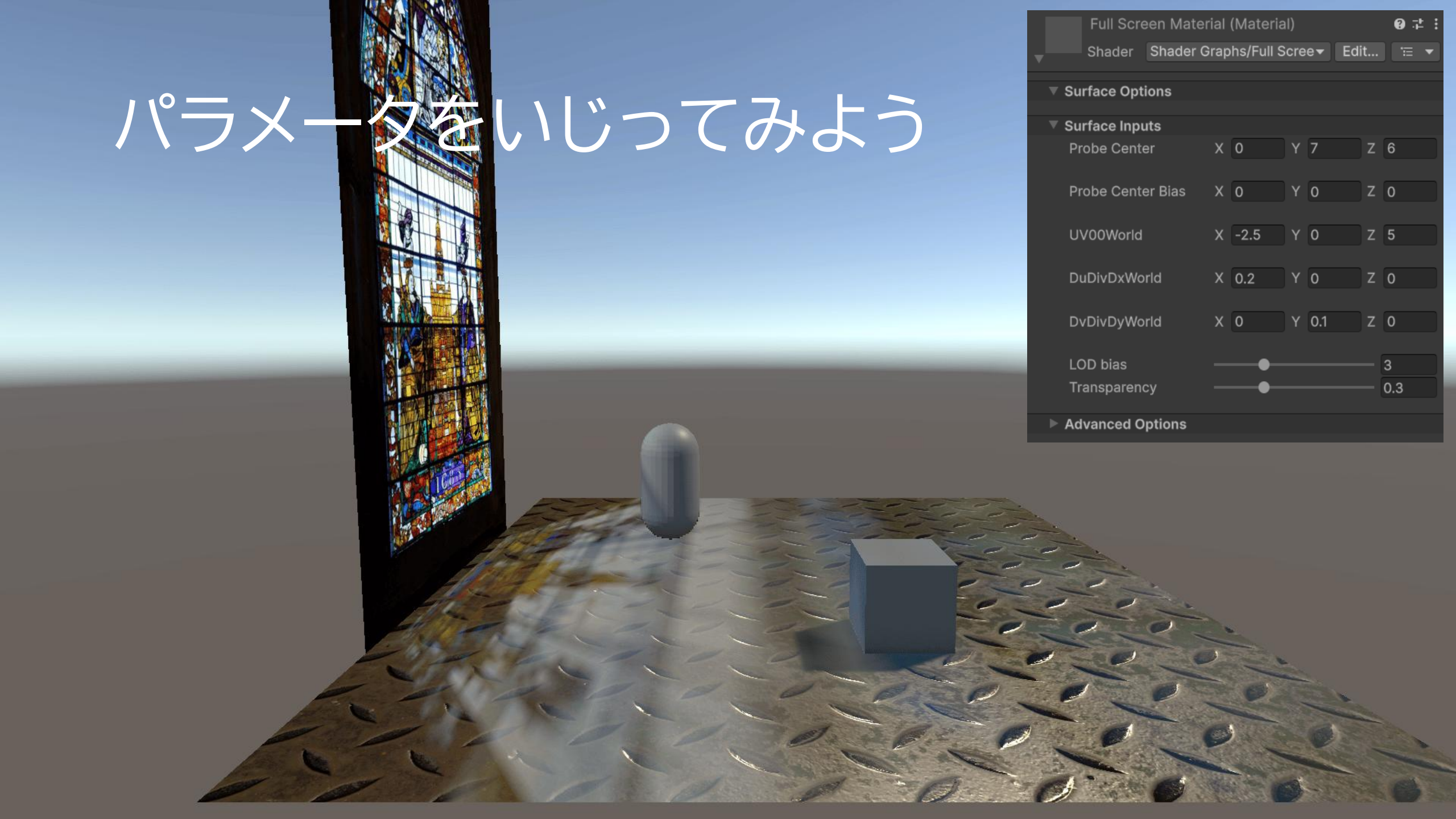

```
using UnityEngine;

Unity スクリプト (2 件のアセット参照)|0 個の参照
public class ProjectionMonoBehaviourScript : MonoBehaviour
{
    [SerializeField] Material material = default!;
    [SerializeField, Range(0.0f, 100.0f)] float Amplitude = 2.0f;
    [SerializeField, Range(0.0f, 1.0f)] float Frequency = 0.1f;
    float Angle = 0.0f;

    Unity メッセージ|0 個の参照
    void Update()
    {
        Angle += Time.deltaTime;

        float x = Amplitude * Mathf.Sin(2.0f * Mathf.PI * Frequency * Angle);
        material.SetVector("_Probe_Center_Bias", new Vector3(x, 0, 0));
    }
}
```

パラメータをいじってみよう



本日の内容

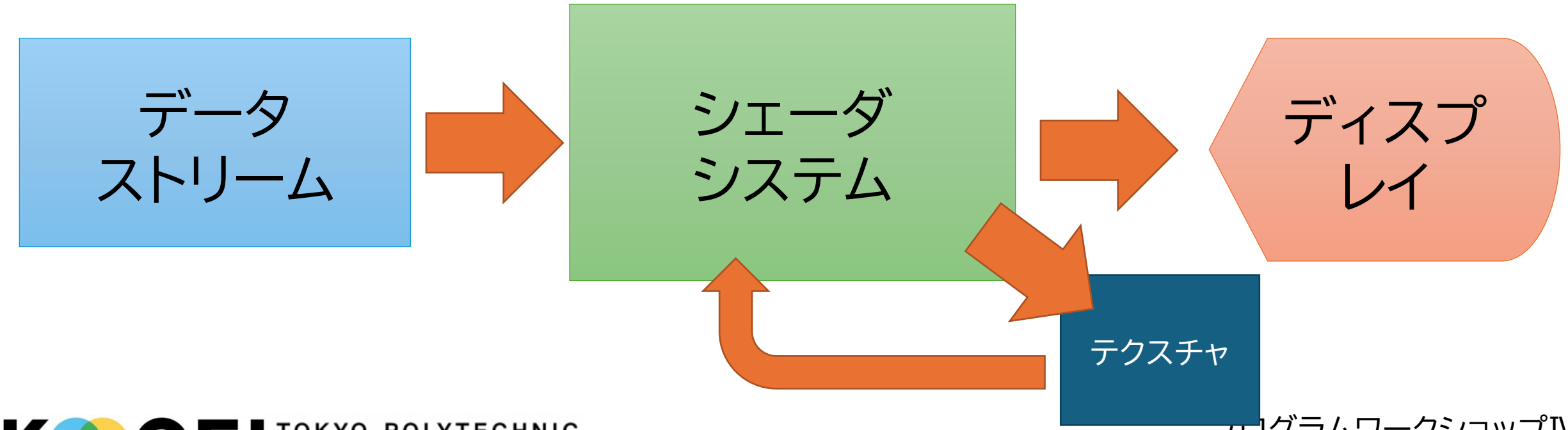
- レンダーターゲット
  - レンダーターゲットの概要
  - 不透明フレームバッファへのアクセス
  - 深度からの位置・法線の復元
- レンダーテクスチャ
  - レンダーテクスチャの概要
  - ゲーム内モニター
  - バックミラー
  - 範囲内のオブジェクトだけ単色

背景

- 通常のシェーダの出力はディスプレイ
- シェーダの結果を再利用したい！

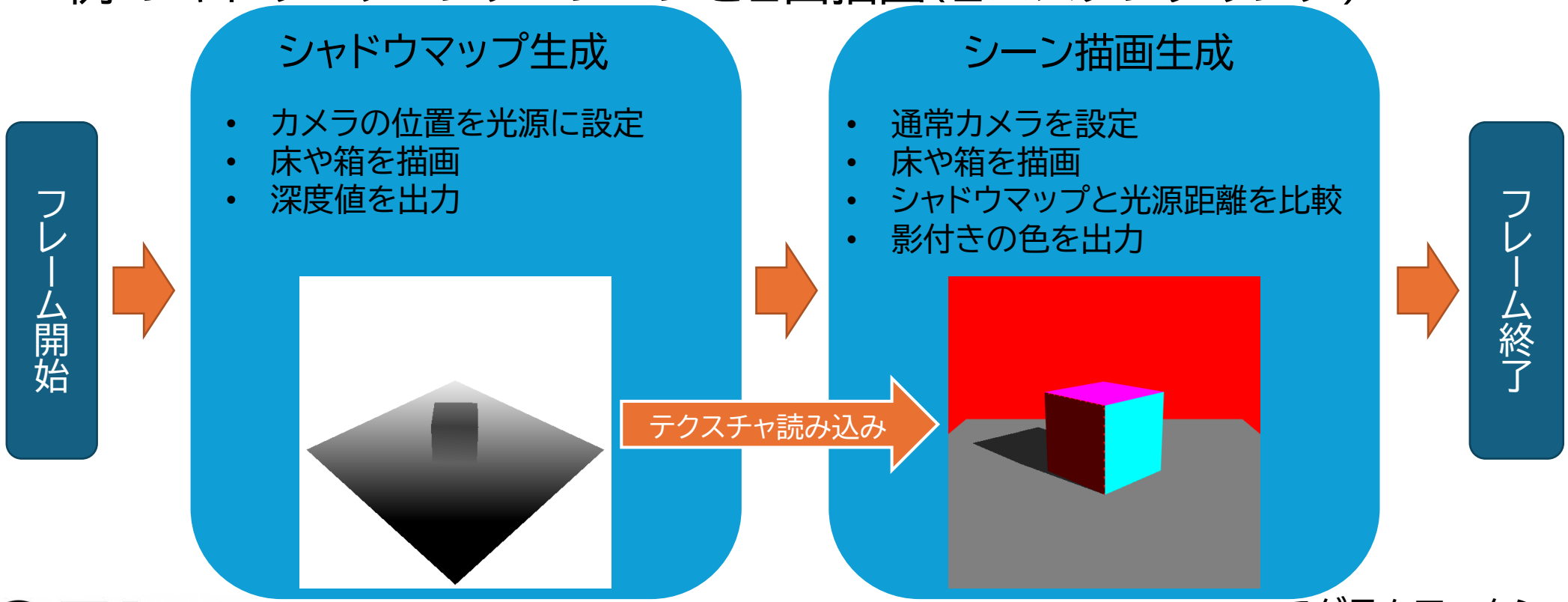
背景

- 通常のシェーダの出力はディスプレイ
- シェーダの結果を再利用したい！
  - テクスチャに結果を保存して過去の情報を参照する



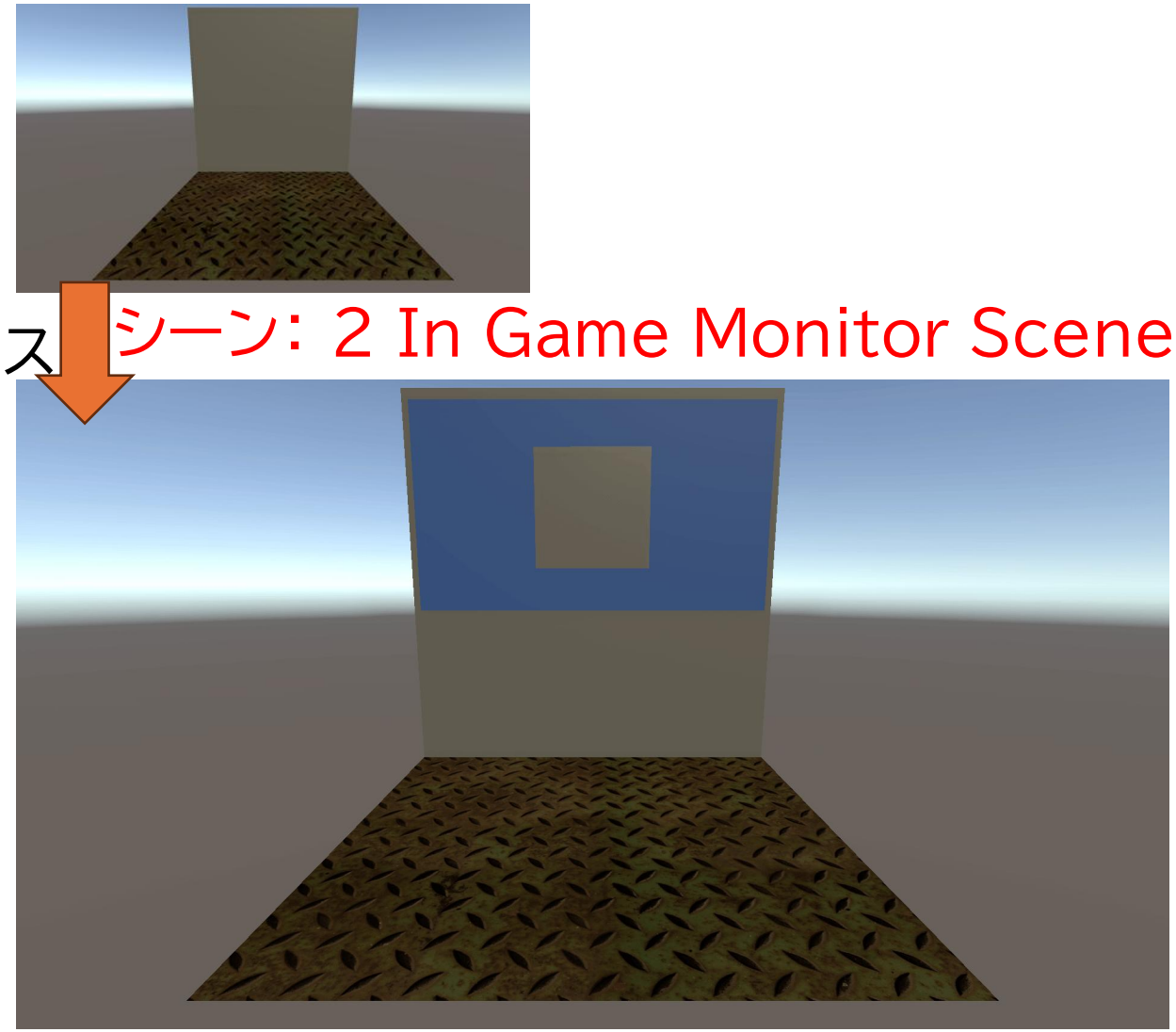
レンダーテクスチャ

- テクスチャへレンダリングする際の対象
  - 例:シャドウマッピング: シーンを2回描画(2パスレンダリング)



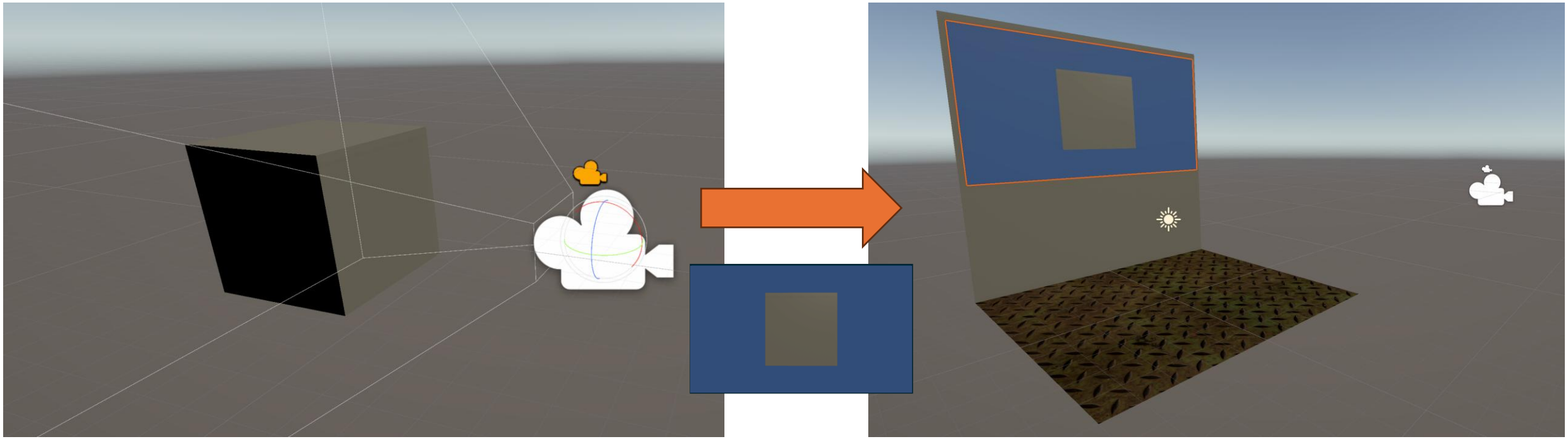
本日の内容

- レンダーターゲット
  - レンダーターゲットの概要
  - 不透明フレームバッファへのアクセス
  - 深度からの位置・法線の復元
- レンダーテクスチャ
  - レンダーテクスチャの概要
  - ゲーム内モニター
  - バックミラー
  - 範囲内のオブジェクトだけ単色



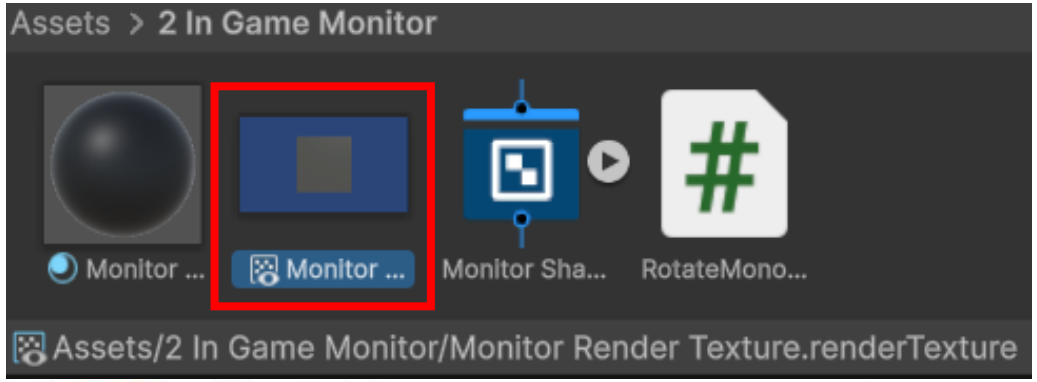
概要

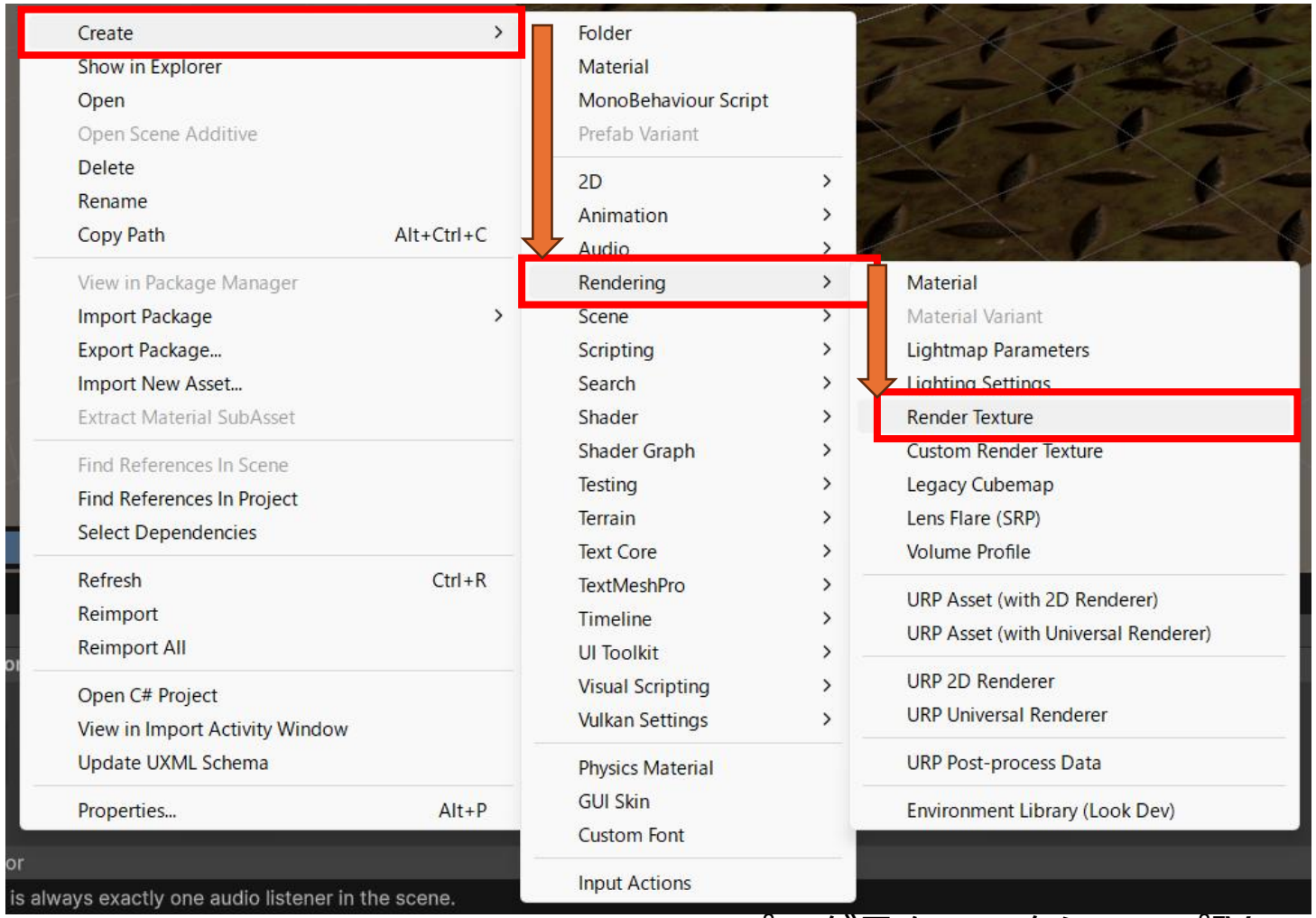
- 別のカメラでカメラに映るものを撮影
- 撮影結果をテクスチャとしてオブジェクトの描画で利用する



ステップ1：テクスチャへのレンダリング

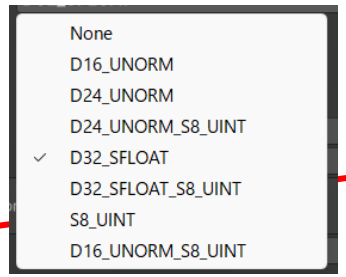
- レンダーテクスチャの生成
  - Project 内を右クリックした際の Createから選択
    - 「Custom Render Texture」もあるが、そちらではない
  - 命名例：「Monitor Render Texture」





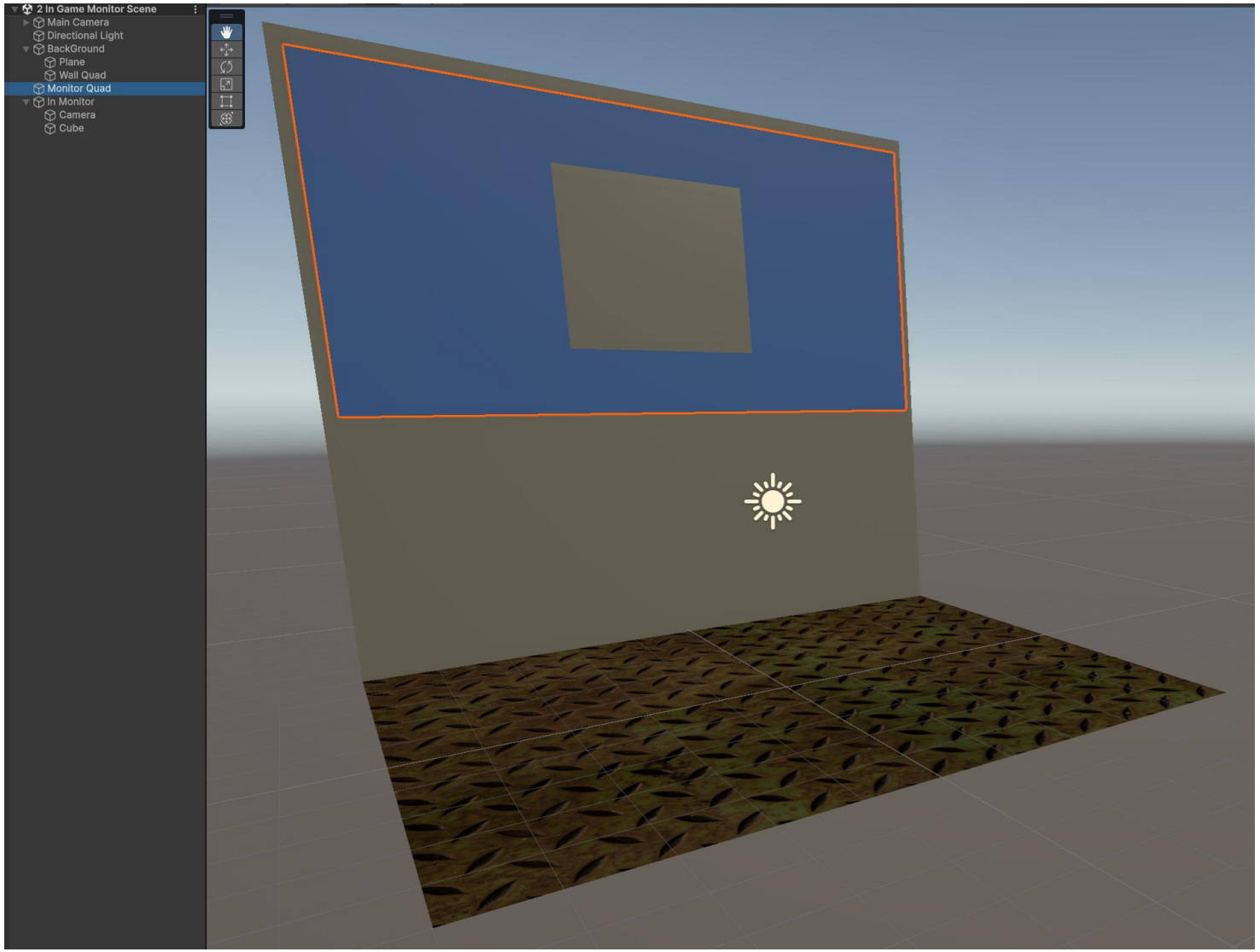
Render Textureの設定

- サイズは表示する際のアスペクト比に合わせる
  - でないと太ったり引き延ばされる
- 描画するので「Depth Buffer」が必要
  - 今回は精度はどうでもよい
  - ステンシルバッファは使わない
- Wrap Modeは「Clamp」
  - Repeat等にすると、画面に貼る際に等倍以外で外周のピクセルに反対の絵が入りこむ



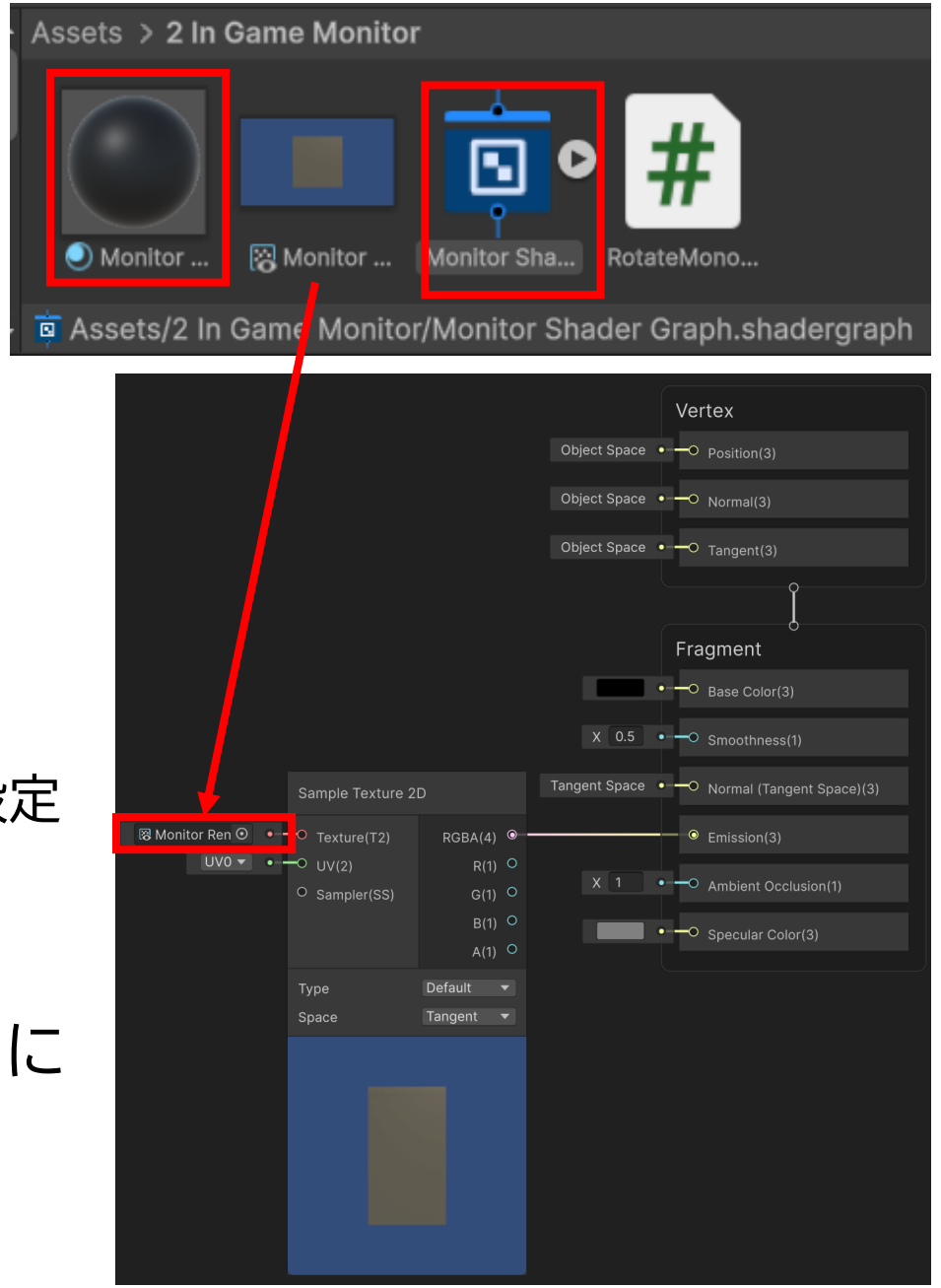
表示オブジェクト

- モニター
  - Quadオブジェクト
  - 右図の名前例:
    「Monitor Quad」
- 背景(設定済み)
  - 床
  - 壁



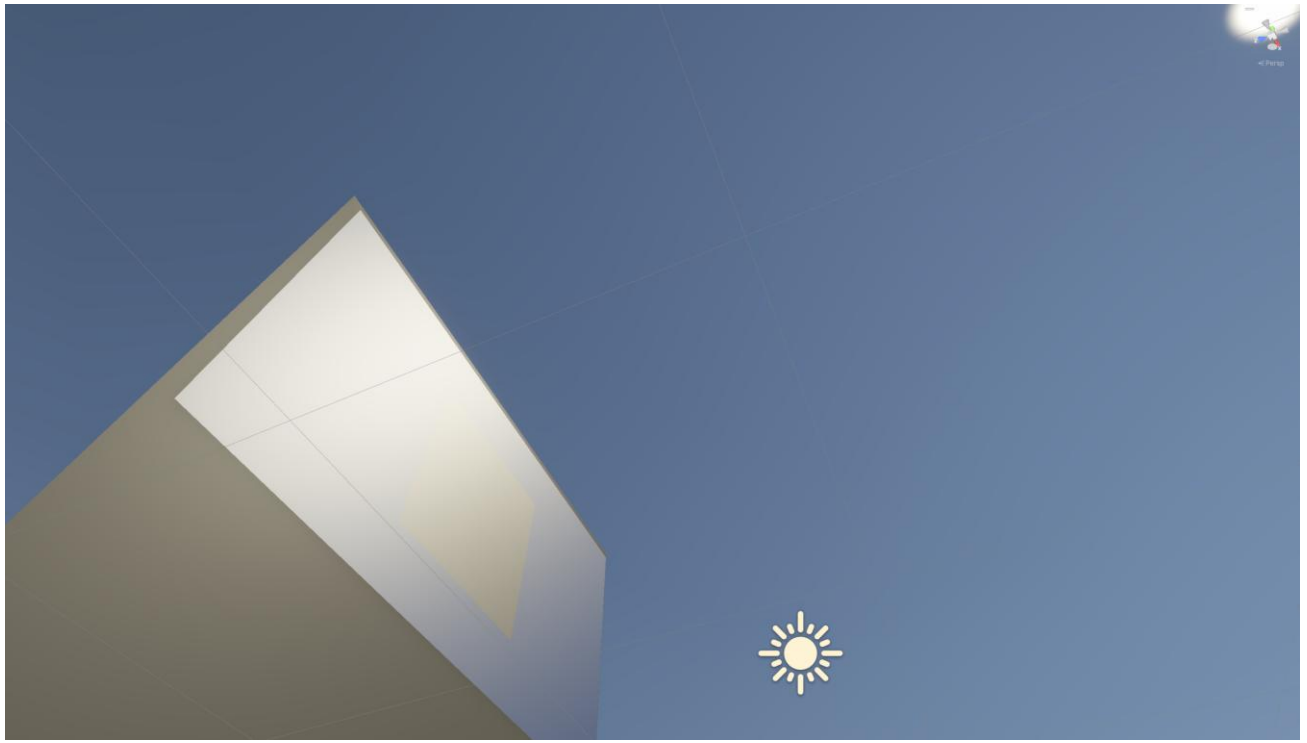
マテリアル

- Shader Graph
  - 下で説明されるマテリアルに設定
  - 命名例：Monitor Shader Graph
  - 内容：Lit Shader Graphでテクスチャの結果をEmissionにつなげる
    - テクスチャは「Monitor Render Texture」を設定
    - Base Colorは黒
- マテリアル
  - モニター用のオブジェクト(Monitor Quad)にバインドする
  - 命名例：Monitor Material



Lit Shader Graphが良いの？

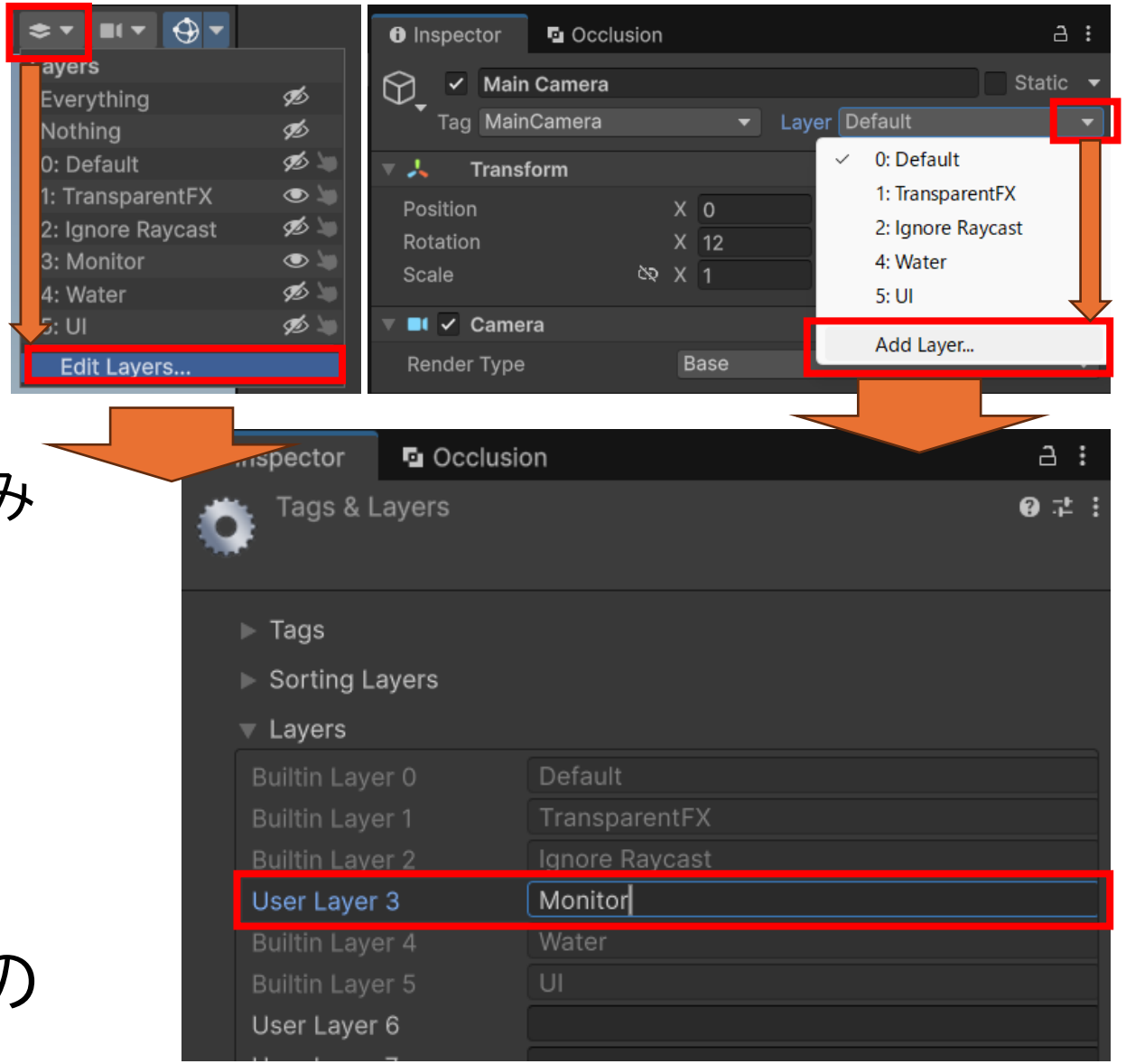
- Unlit Shader Graphのカラーで出力しても良い
- LitのEmissionに出力するとスペキュラは適応されるのでモニターの表面感がでる
  - それが良いかどうかは作りたいもの次第



レイヤー

- 表示するテクスチャ内に描画するオブジェクトは「レイヤー」で選ぶ
  - 表示物を切り分けるUnityの仕組み
- シーンの右上のセレクターから塚
  - 「Edit Layers」を選択
- Inspectorの「Layer」から追加
  - 「Add Layer」を選択

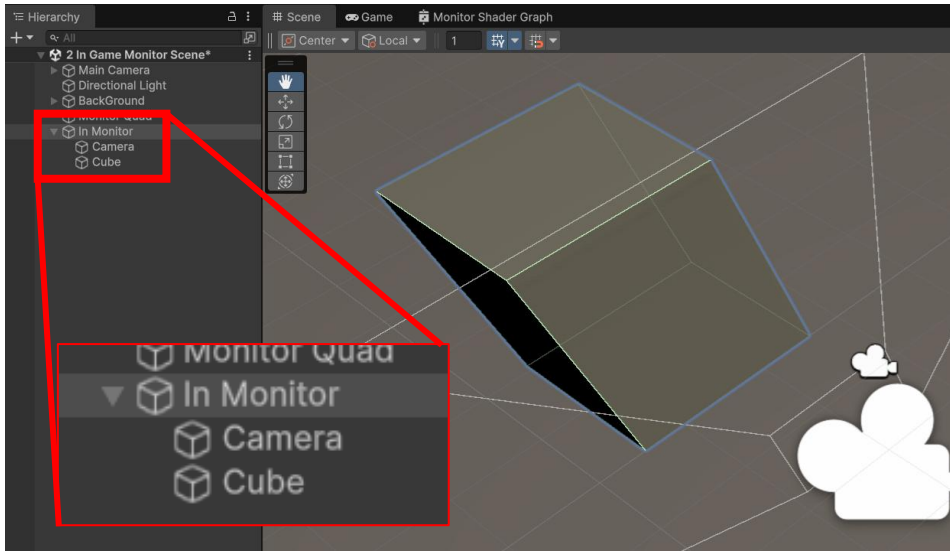
- 表示するレイヤーは、シーン右上のセレクターから切り替えられる

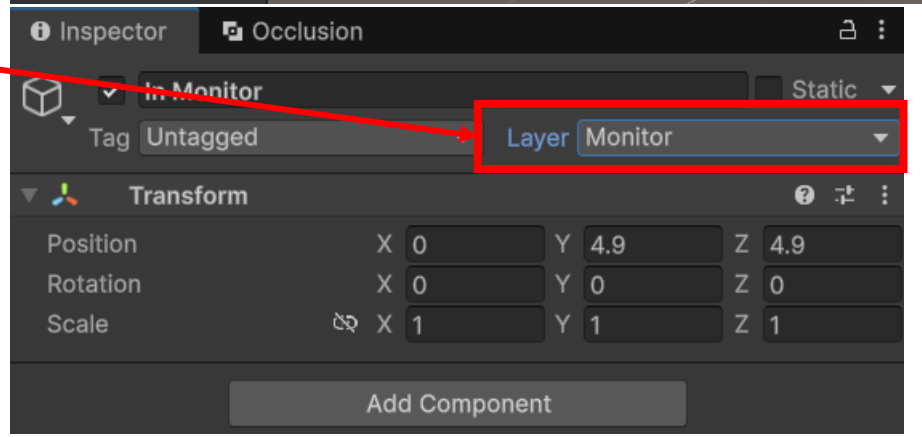


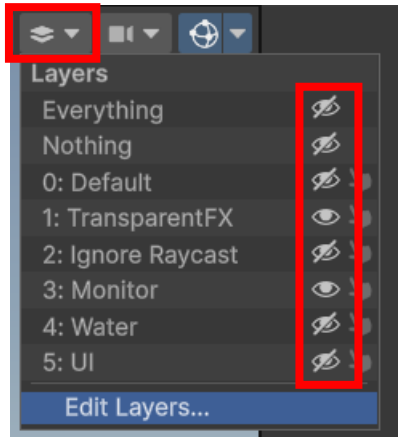
オブジェクトの追加

1. 見やすくするためのEmptyオブジェクト
   - 名称：In monitor
   - Layerとして「Monitor」を設定
     - 子供のオブジェクトのLayerも設定できる
2. カメラ
3. テクスチャに表示されるオブジェクト
   - ここでは立方体（名称：Cube）
     - Emptyオブジェクトから引き継いでレイヤーを「Monitor」にする

- 表示するレイヤーは、シーン右上のセレクターから切り替えられる

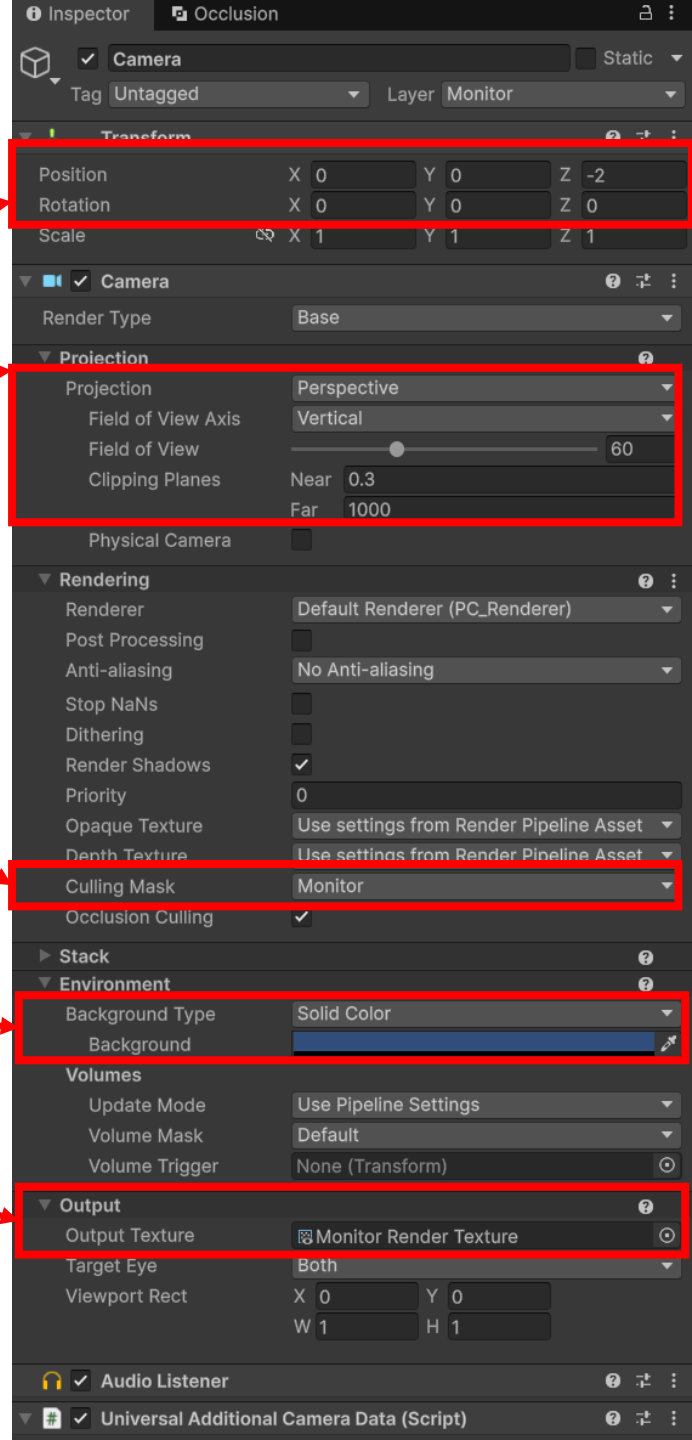






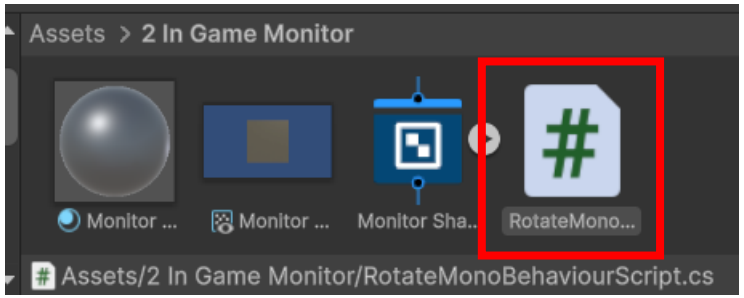
テクスチャ用カメラの設定

- カメラの位置、向きやProjectionはオブジェクトがうまく入る形に設定する
  - あえてパースをつけない平行投影でもよいかもね
- カリングマスクで表示するレイヤーを設定
- 背景は通常のシーンと別の設定が可能
  - ここでは固定色
- 出力先はレンダーテクスチャ



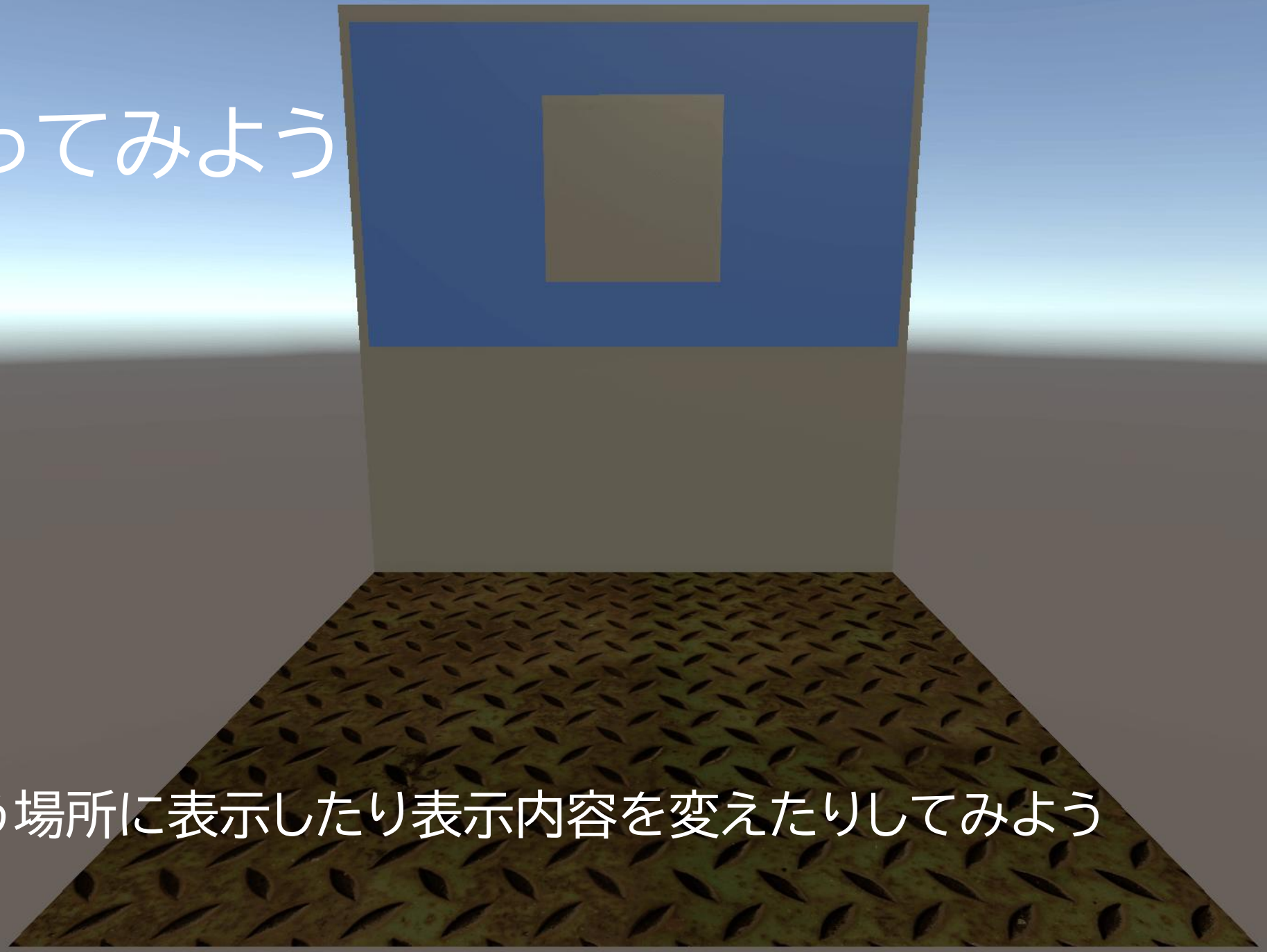
テクスチャに表示されるオブジェクト

- Cubeにスクリプトを付ける
  - スクリプト名例:RotateMonoBehaviourScript
  - Transform の Rotate メソッドで回転




```
1          using UnityEngine;
2
           Unity スクリプト (1 件のアセット参照)|0 個の参照
3          public class RotateMonoBehaviourScript : MonoBehaviour
4          {
5              // Start is called once before the first execution of Update aft
               Unity メッセージ|0 個の参照
6              void Start()
7              {
8
9              }
10
11             // Update is called once per frame
               Unity メッセージ|0 個の参照
12             void Update()
13             {
14                 this.transform.Rotate(0.0f, 20.0f * Time.deltaTime, 0.0f);
15             }
16         }
```

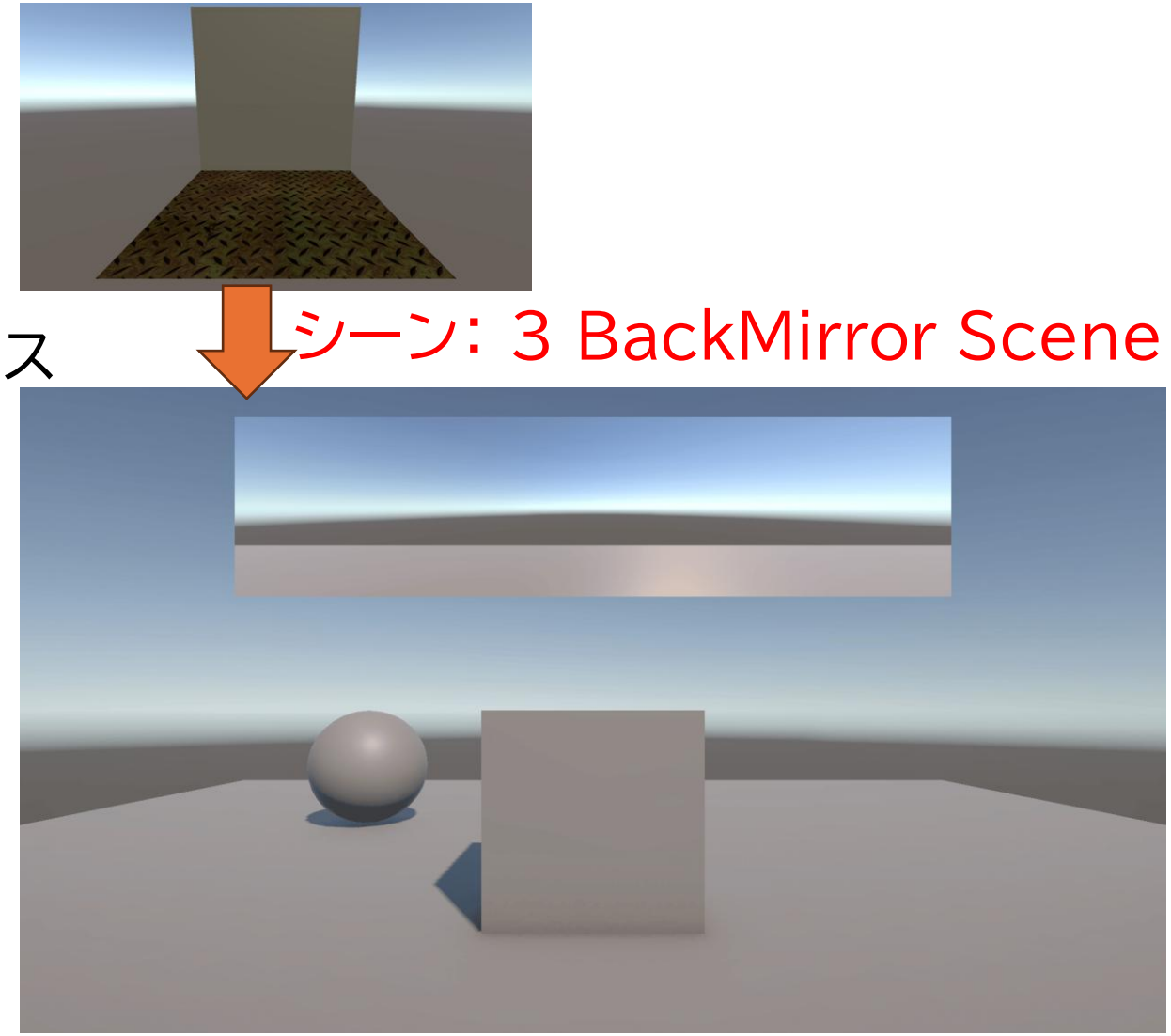
やってみよう



- 違う場所に表示したり表示内容を変えたりしてみよう

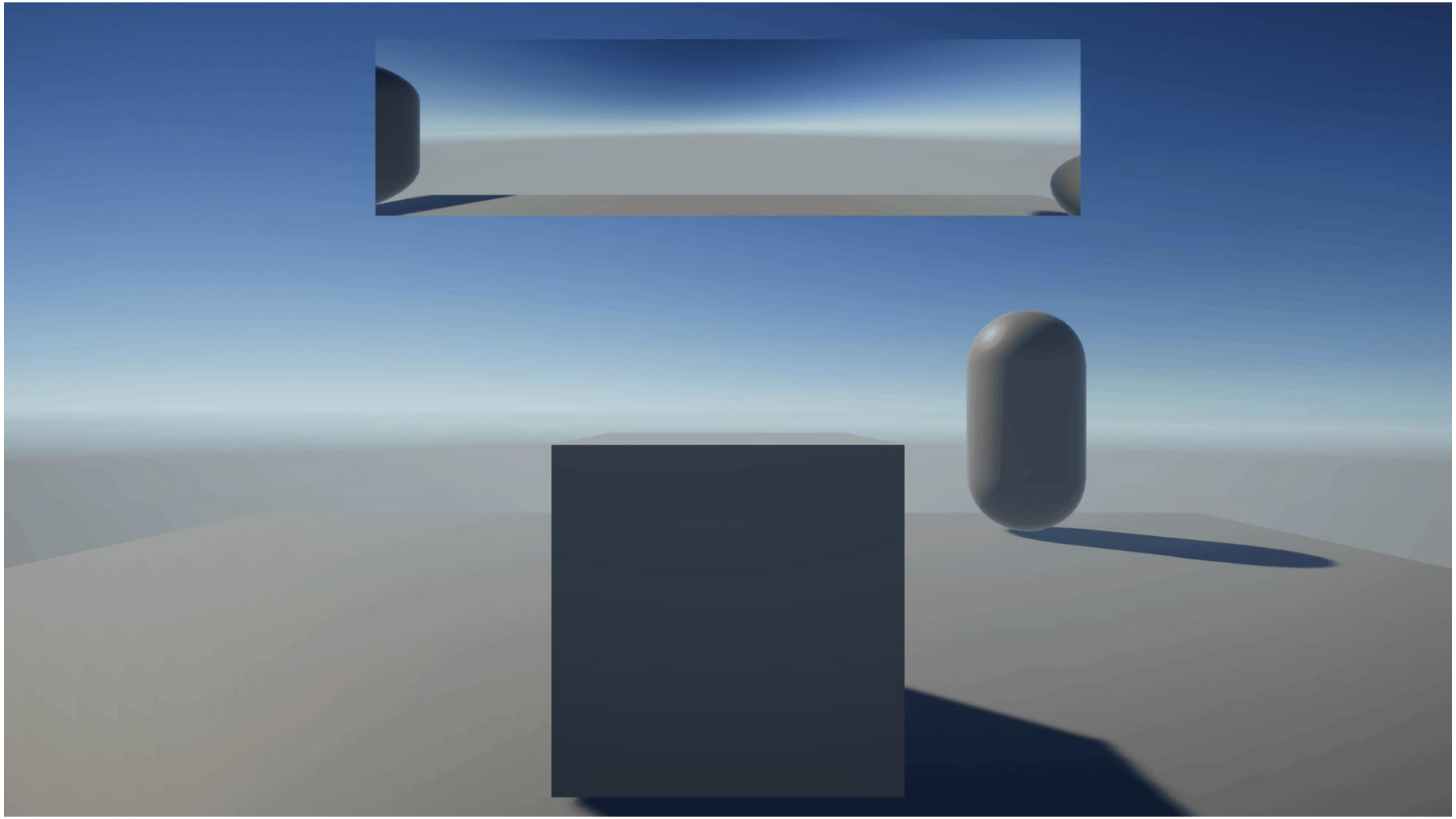
本日の内容

- レンダーターゲット
  - レンダーターゲットの概要
  - 不透明フレームバッファへのアクセス
  - 深度からの位置・法線の復元
- レンダーテクスチャ
  - レンダーテクスチャの概要
  - ゲーム内モニター
  - バックミラー
  - 範囲内のオブジェクトだけ単色



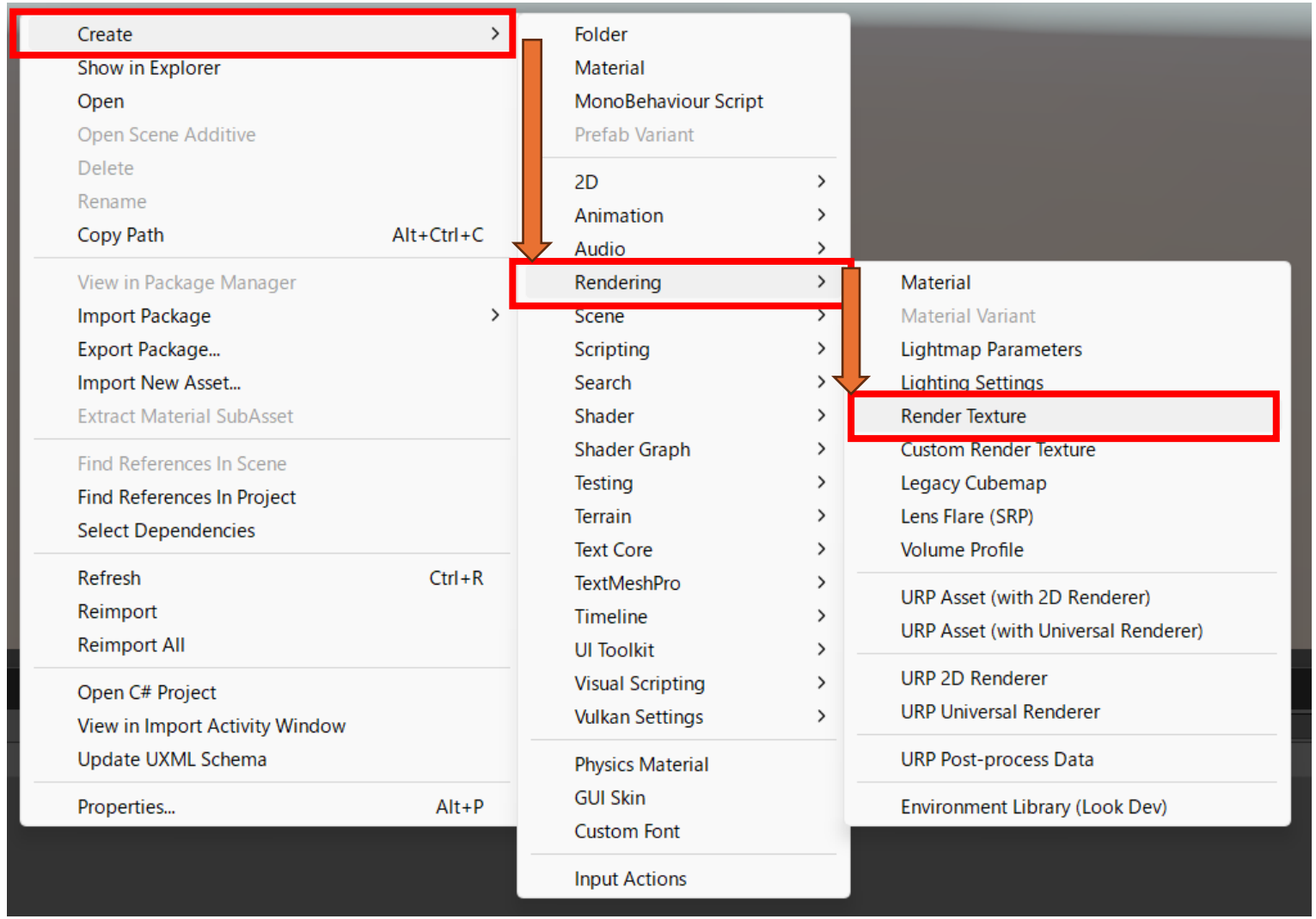
バックミラー

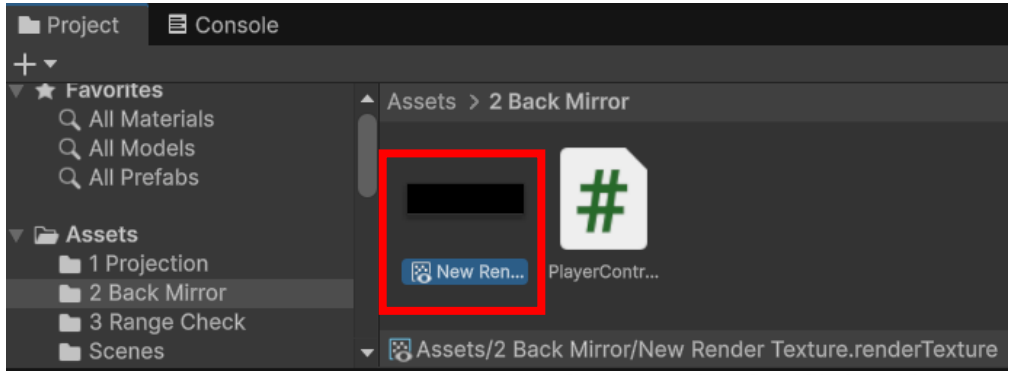
- 後方の景色をUIとして映す



用意するもの

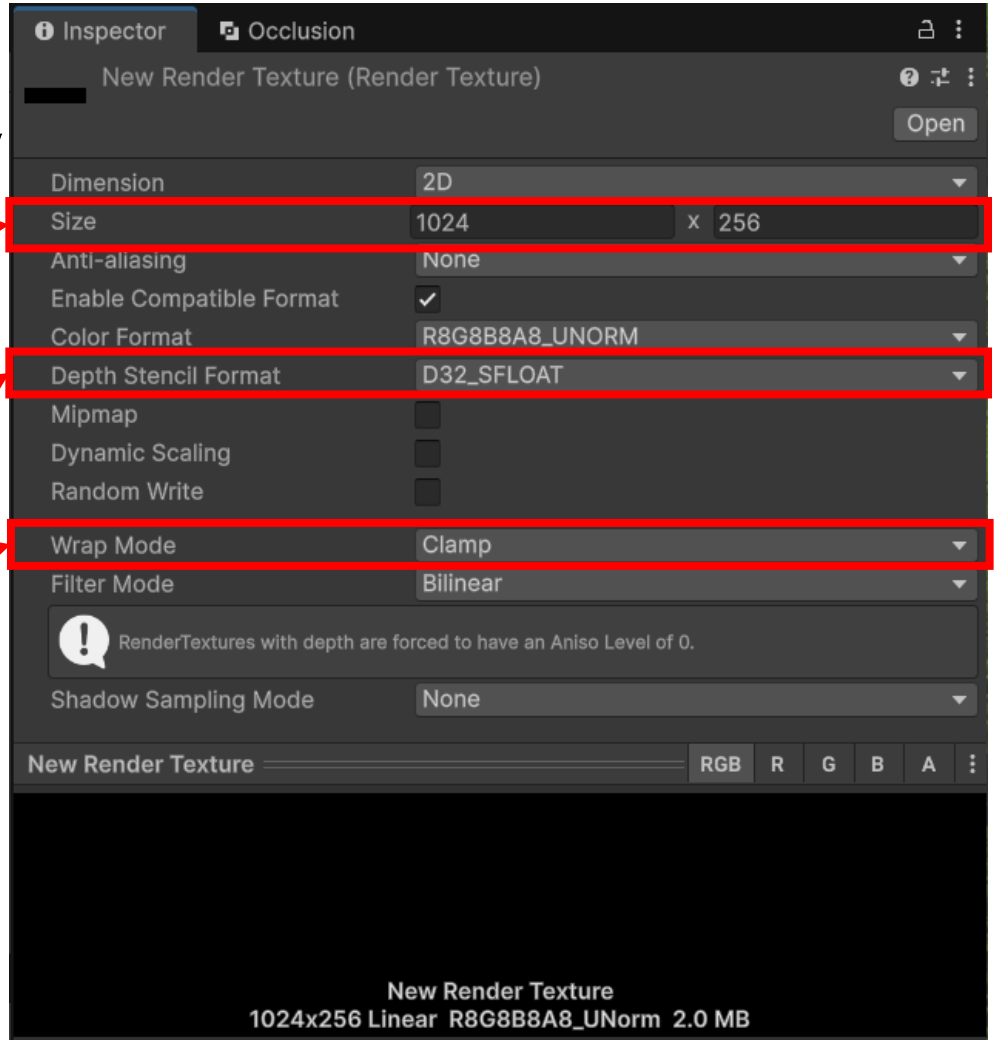
- Render Texture
  - Project 内を右クリックした際の Create から選択
    - 「Custom Render Texture」もあるが、そちらではない

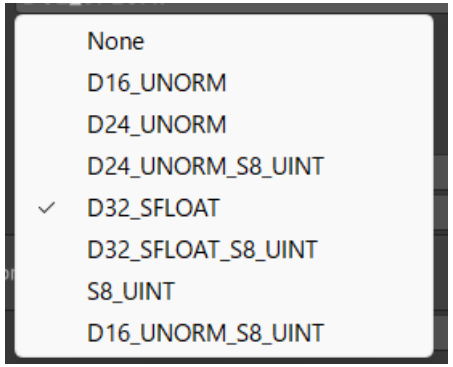




Render Textureの設定

- サイズをイイ感じに調整
- 描画するので「Depth Buffer」が必要
  - 今回は精度はどうでもよい
  - ステンシルバッファは使わない
- Wrap Modeは「Clamp」
  - Repeat等にすると、画面に貼る際に等倍以外で外周のピクセルに反対の絵が入りこむ

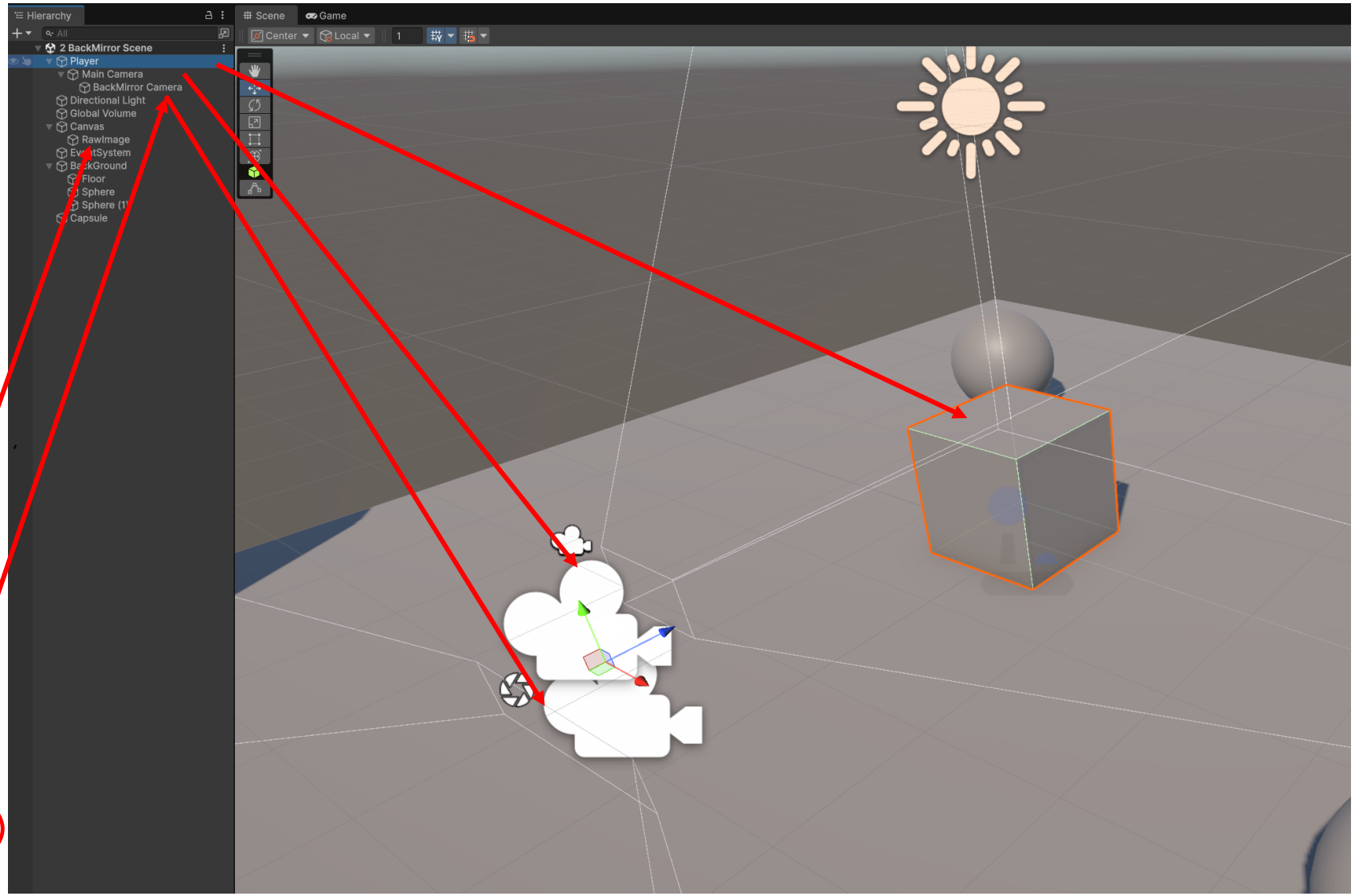




Depth Stencil Format

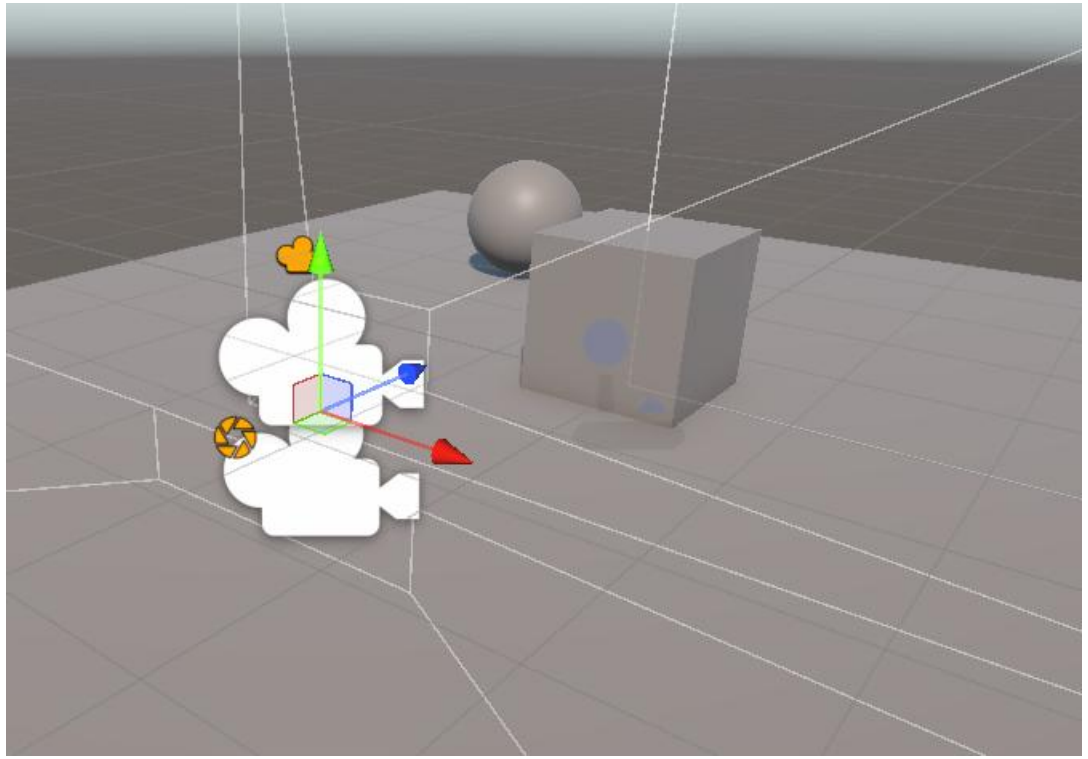
オブジェクト

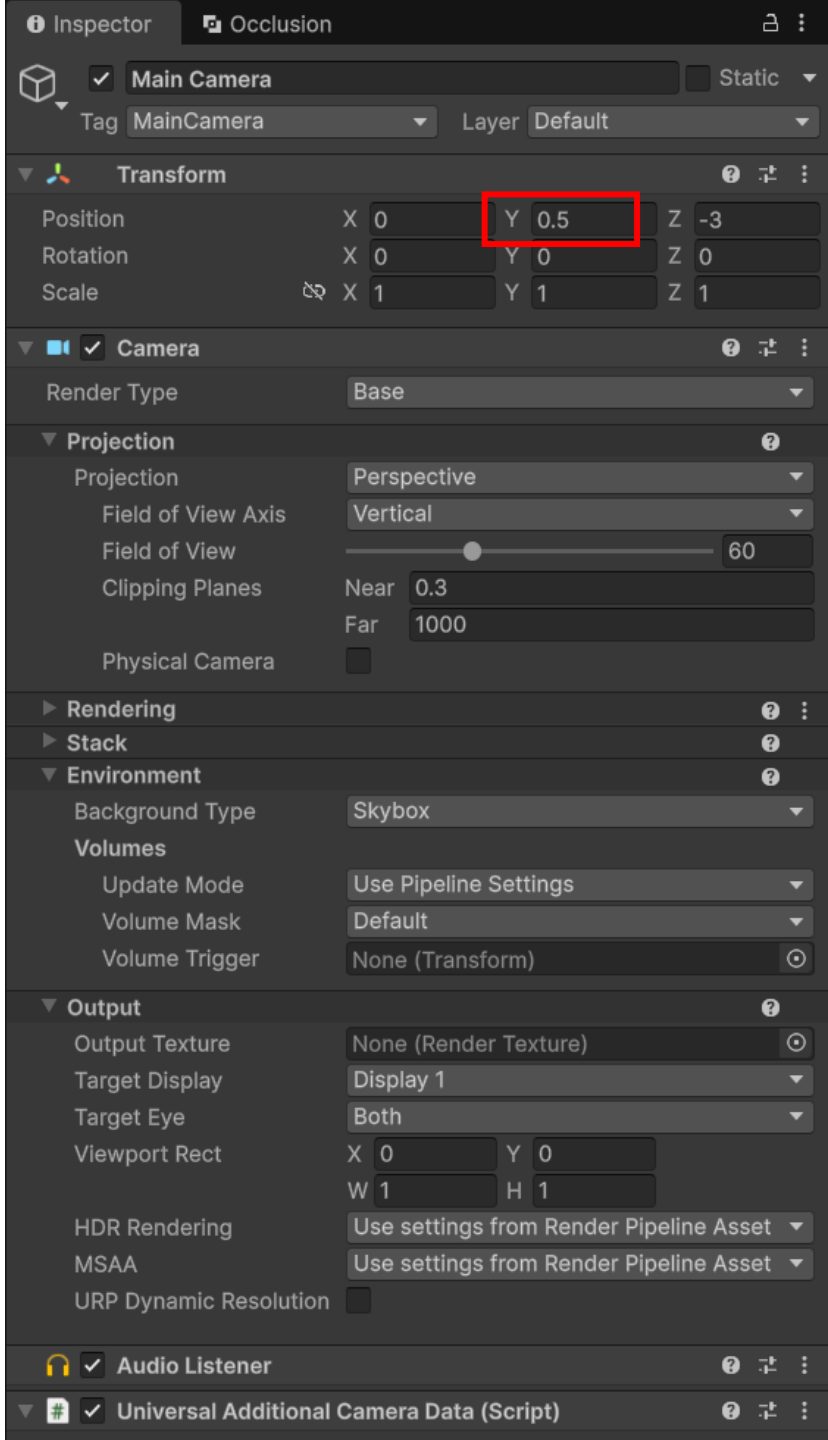
- プレイヤー
- メインカメラ
  - シーンを描画
  - プレイヤーの子にすることで、常にプレイヤーの後ろに位置する
- バックミラー矩形
  - 追加する
    - UI/Raw Image
  - 描画したテクスチャを貼る
- バックミラーカメラ
  - 追加する(Camera)
  - 後方を見る



カメラ

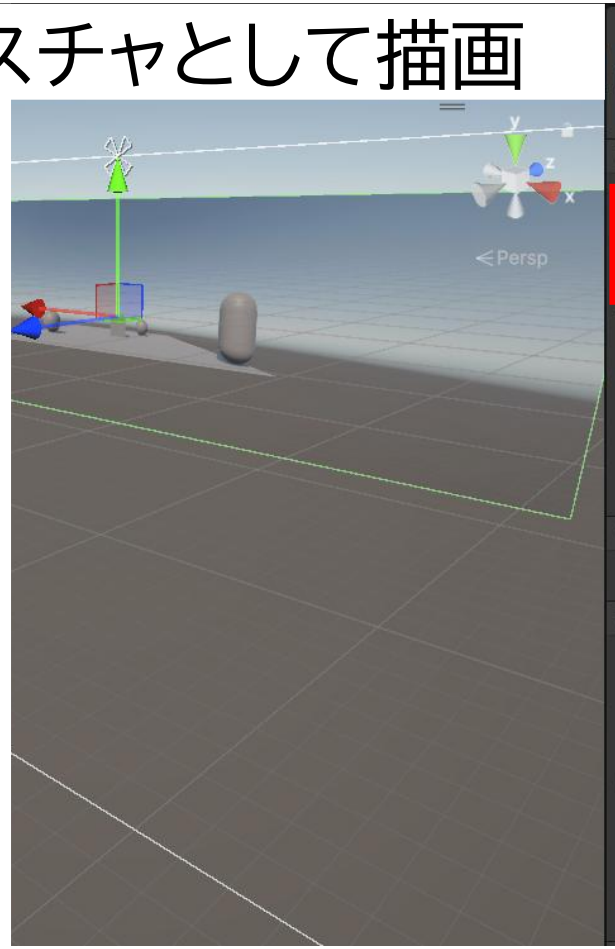
- プレイヤーの後方上部に配置

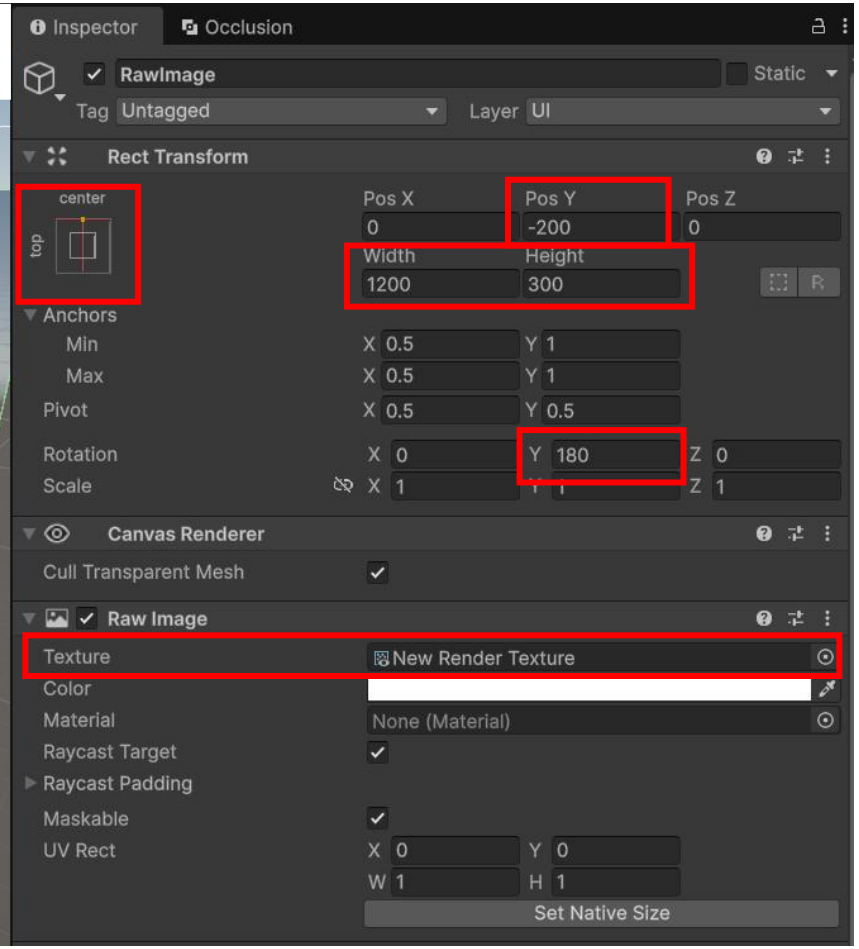




バックミラー矩形

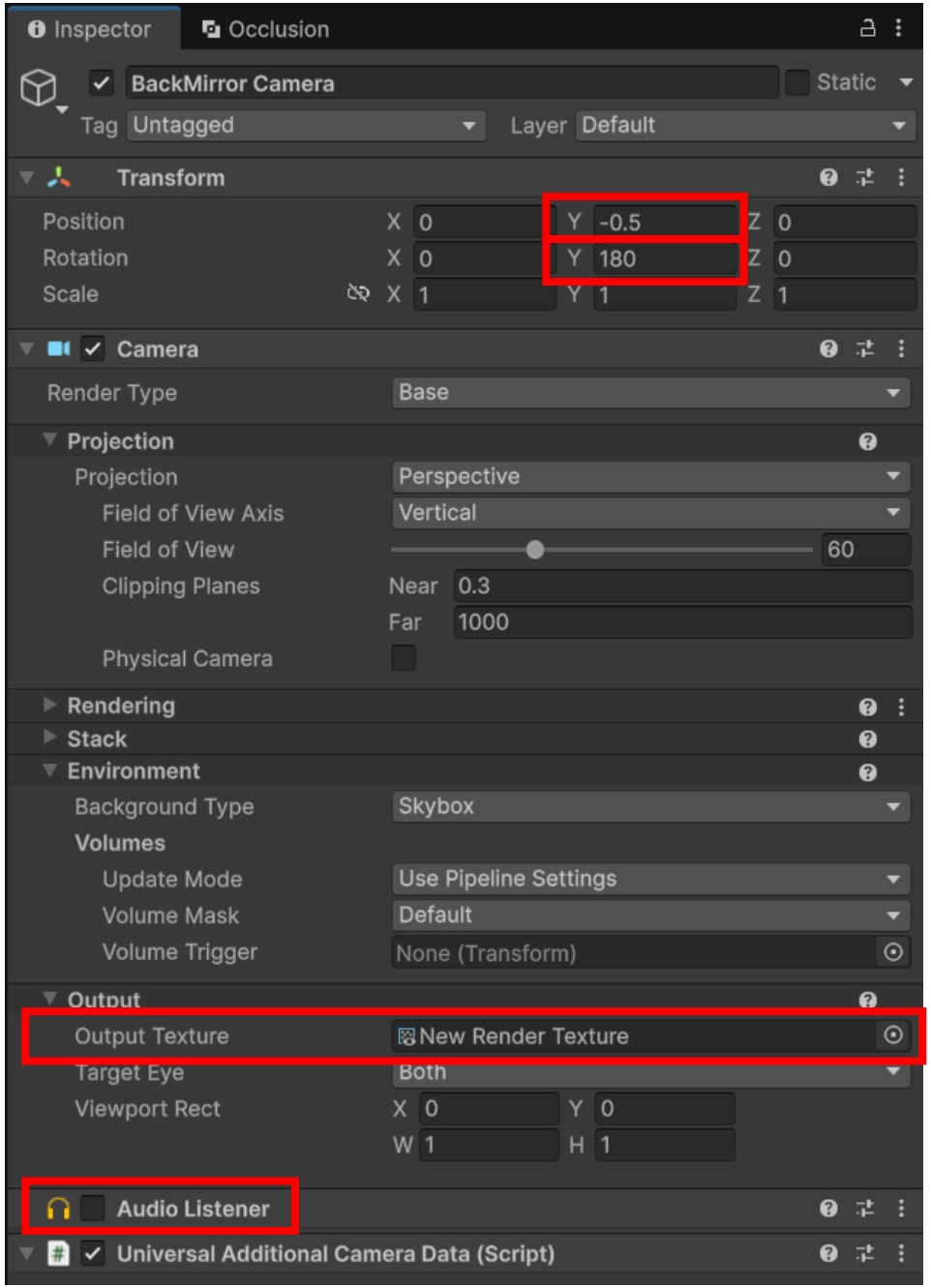
- レンダーテクスチャをテクスチャとして描画
  - アンカーは中央上部
  - 位置はいい感じの高さに
  - サイズはレンダーテクスチャのアスペクト比のに合わせる事
  - RotationはY軸180度
    - 鏡として反転して表示する
  - Textureにレンダーテクスチャを設定する





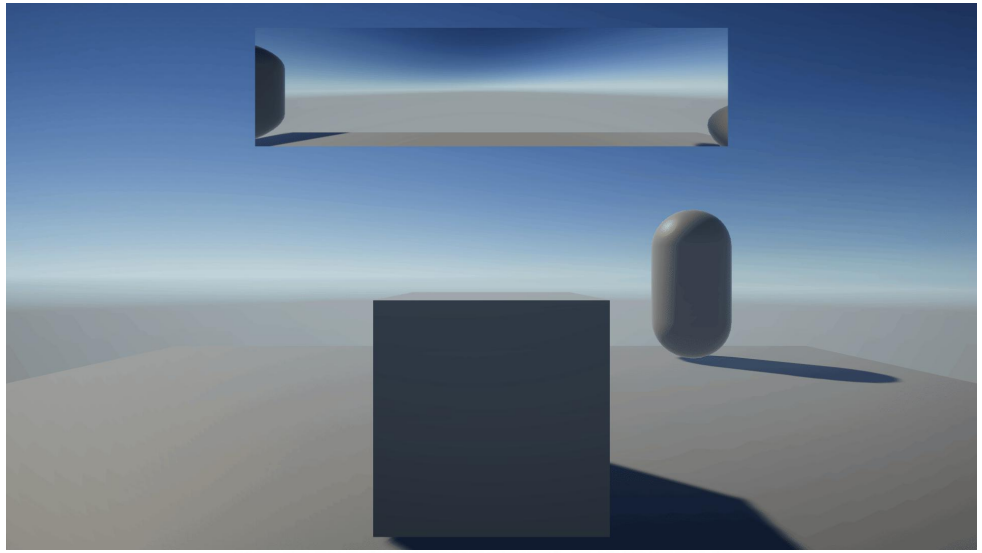
バックミラーカメラ

- 高さはいい感じに
- 反対向き
  - Y軸:180度回転
- テクスチャに描画
  - “Output” の“Output Texture”にレンダーテクスチャを設定
- Audio Listenerのチェックは外す
  - 忘れるとエラーが出る



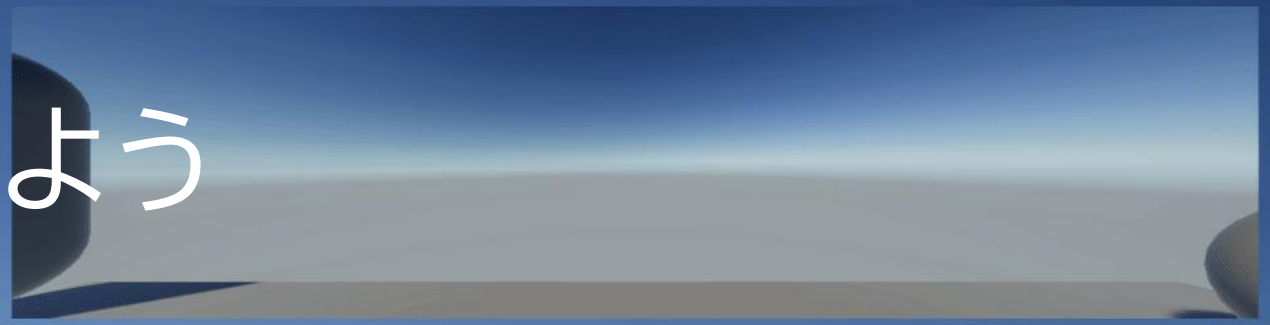
プレイヤー制御

- MonoBehaviourScript追加
- Playerオブジェクトにスクリプトを付ける
- WASDで移動
  - お好きな感じで



```
1   using UnityEngine;
2   using UnityEngine.InputSystem;
3
    Unity スクリプト (1 件のアセット参照)|0 個の参照
4   public class PlayerControllerMonoBehaviourScript : MonoBehaviour
5   {
6       [SerializeField] float FORWARD_ACCELARATION = 30.0f; // 加速
7       [SerializeField] float SIDE_ACCELARATION = 1000.0f;
8       [SerializeField] float FORWARD_DAMPING = 4.0f; // 自動減速
9       [SerializeField] float SIDE_DAMPING = 1.0f;
10
11      float forwardSpeed = 0.0f;
12      float sideSpeed = 0.0f;
13
        Unity メッセージ|0 個の参照
14      void Update()
15      {
16          var current = Keyboard.current;
17          if (current == null) return;
18
19          // 入力取得
20          float forwardInput = 0.0f;
21          if(current.wKey.isPressed) forwardInput += 1.0f;
22          if(current.sKey.isPressed) forwardInput -= 1.0f;
23          float sideInput = 0.0f;
24          if(current.aKey.isPressed) sideSpeed -= 1.0f;
25          if(current.dKey.isPressed) sideSpeed += 1.0f;
26
27          forwardSpeed += forwardInput * FORWARD_ACCELARATION * Time.deltaTime;
28          sideSpeed += sideInput * SIDE_ACCELARATION * Time.deltaTime;
29
30          // 移動
31          transform.Translate(Vector3.forward * forwardSpeed * Time.deltaTime);
32          transform.Rotate(Vector3.up * sideSpeed * Time.deltaTime);
33
34          // 減速
35          forwardSpeed -= forwardSpeed * FORWARD_DAMPING * Time.deltaTime;
36          sideSpeed -= sideSpeed * SIDE_DAMPING * Time.deltaTime;
37      }
38  }
```

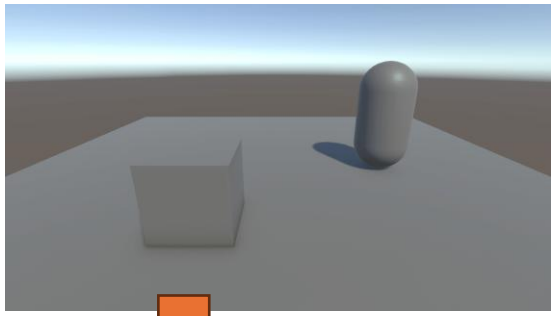
やってみよう



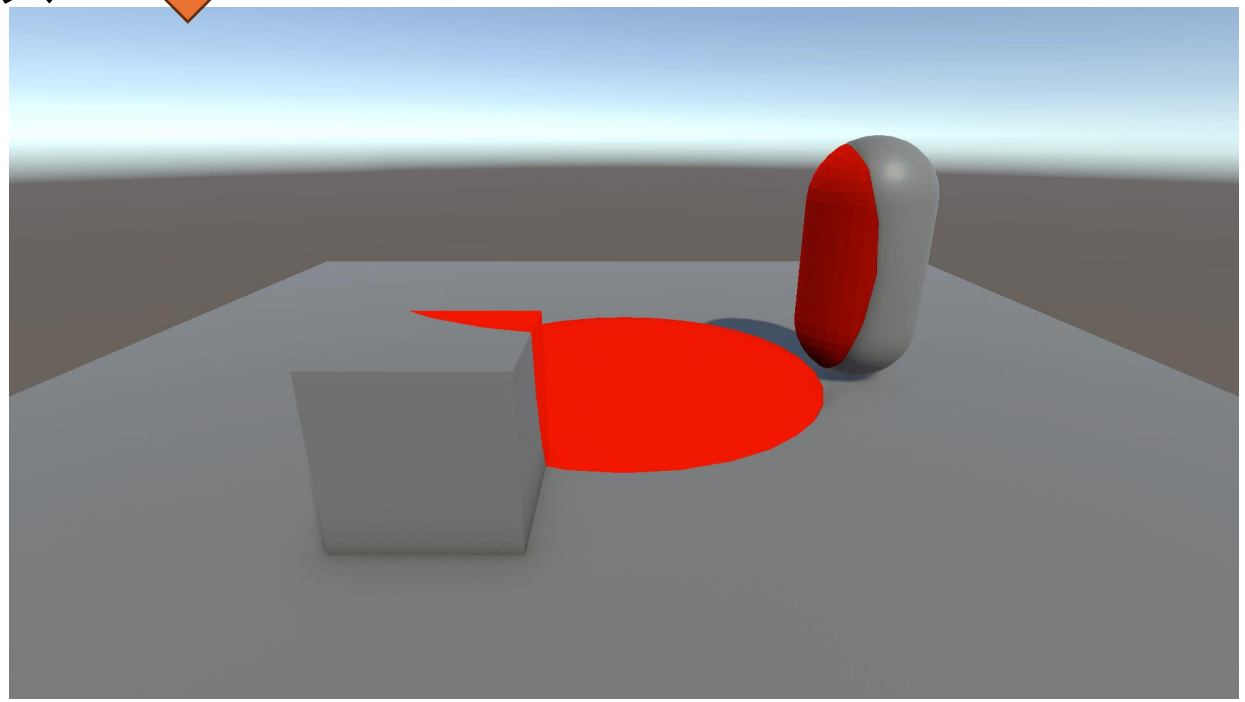
- 時間があれば
  サイドミラーも
  つけてみよう

本日の内容

- レンダーターゲット
  - レンダーターゲットの概要
  - 不透明フレームバッファへのアクセス
  - 深度からの位置・法線の復元
- レンダーテクスチャ
  - レンダーテクスチャの概要
  - ゲーム内モニター
  - バックミラー
  - 範囲内のオブジェクトだけ単色

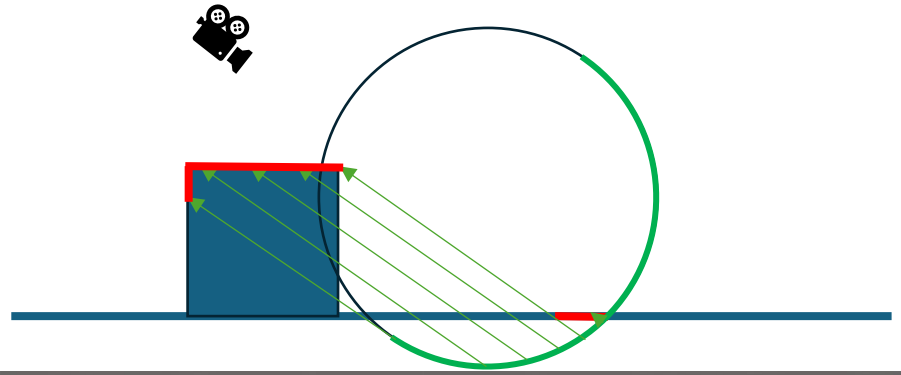


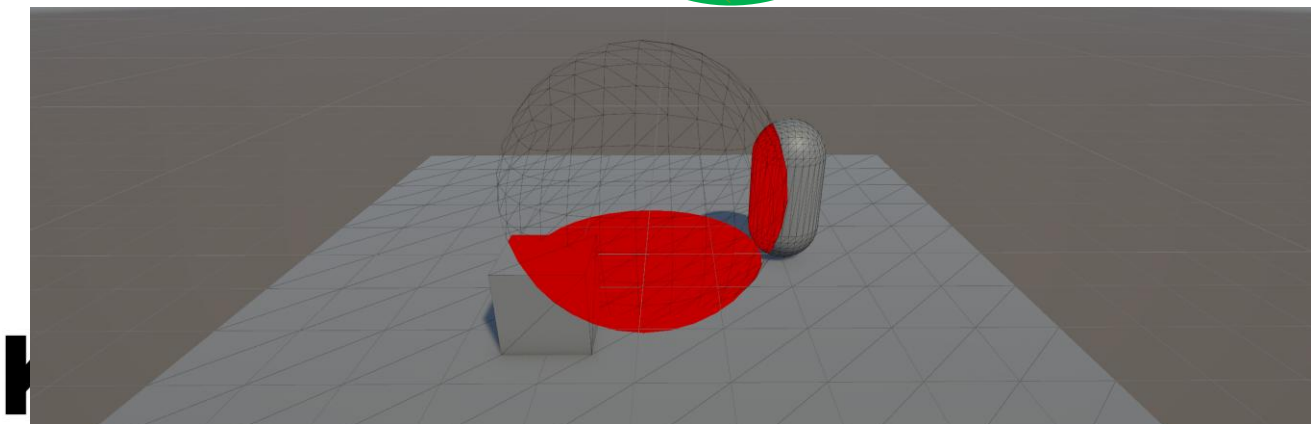
シーン:
4_2 Range Check 3D Scene



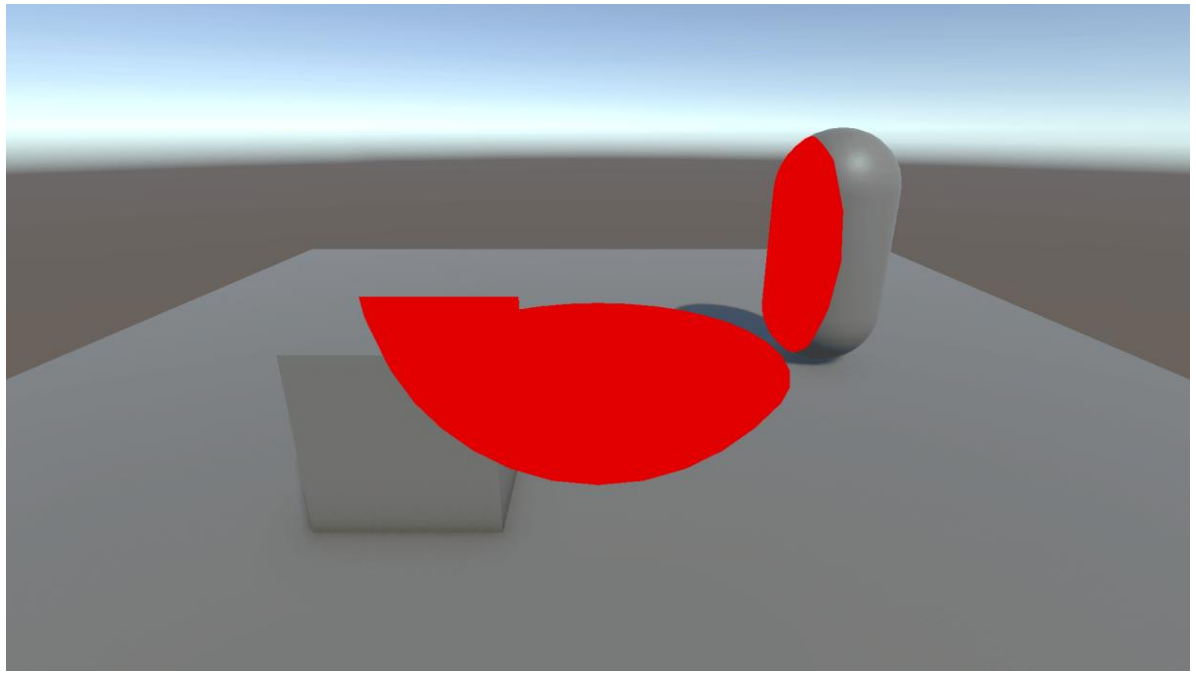
特定の範囲に入ったら固定色にする

- 物体を描画した際に表示を切り替えるのではなく、球を描画した際に、その内部に入っている部分を固定色で塗りつぶす
  - 背面を描画し、深度テストに

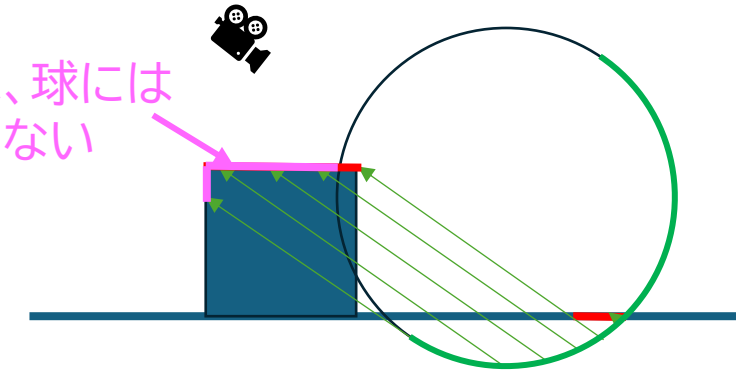




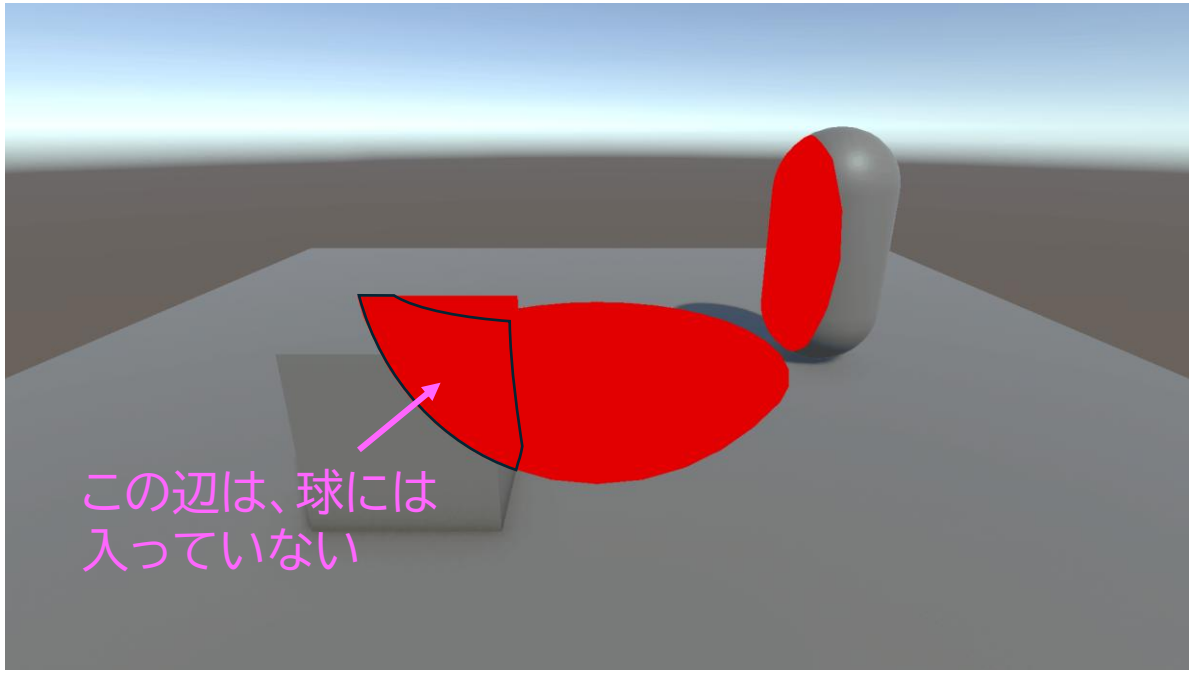
4_1 Range Check Back Scene

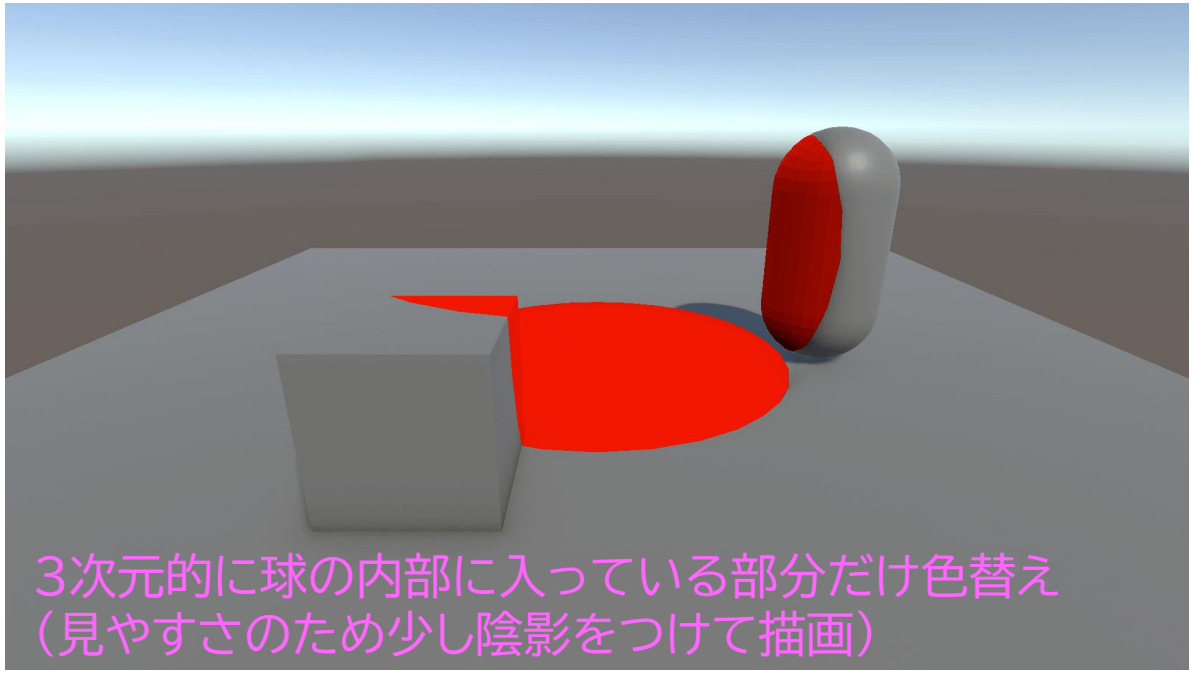


背面を描画した手法の問題



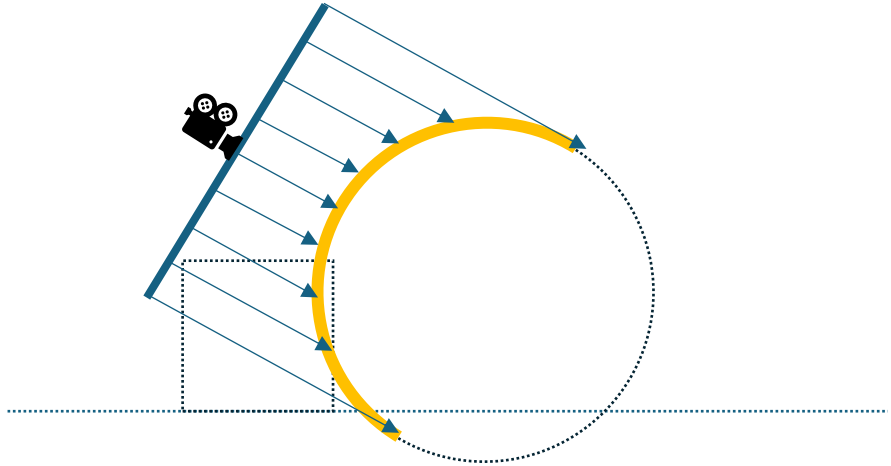
- 3D的には形状(球)に入っていない部分も範囲に入っているかのように処理される
  - 球の表面の情報を使っていないのでこのような表示なる

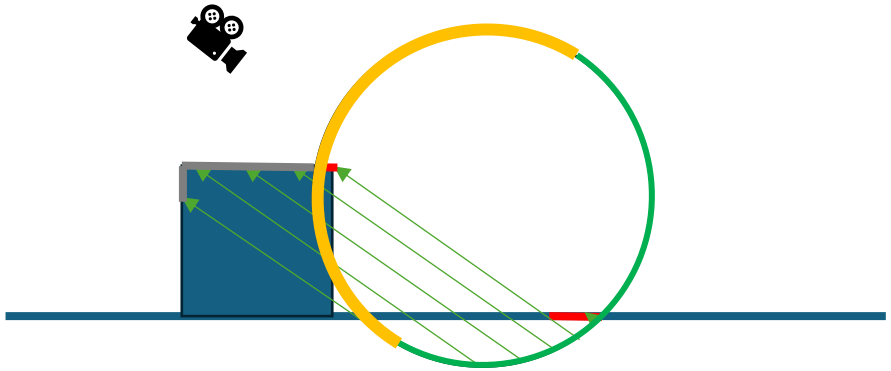




アルゴリズム

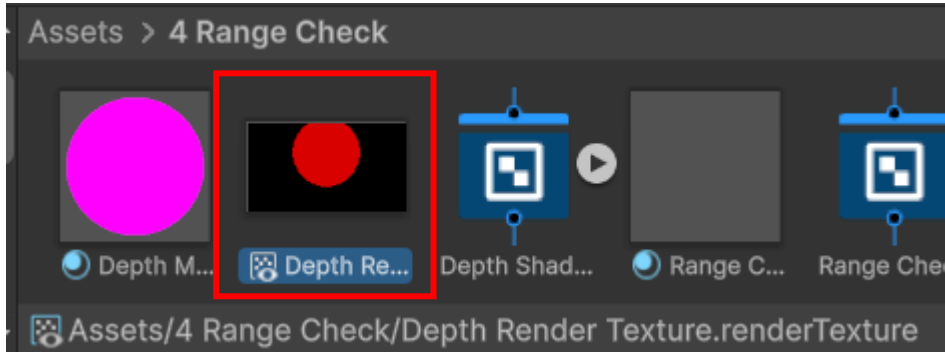
1. カメラからの深度を記録
   - 範囲の形状(球)の前面
2. 球の裏面を描画
   - 球の前面の深度と深度バッファの深度を比較
     - 球の前面の深度＜深度バッファ深度
       - 他の物体が球の中にあるので、固定色で描画
     - 球の前面の深度≧深度バッファ深度
       - 球の外に他の物体があるので、固定色を描画しない

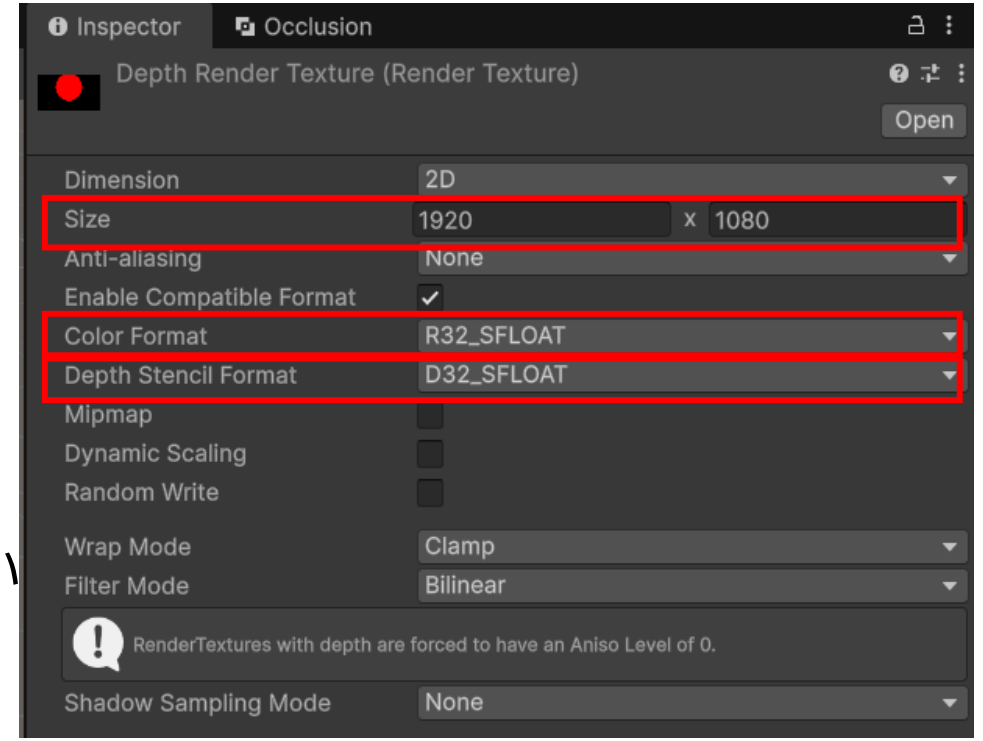




範囲形状の深度の記録

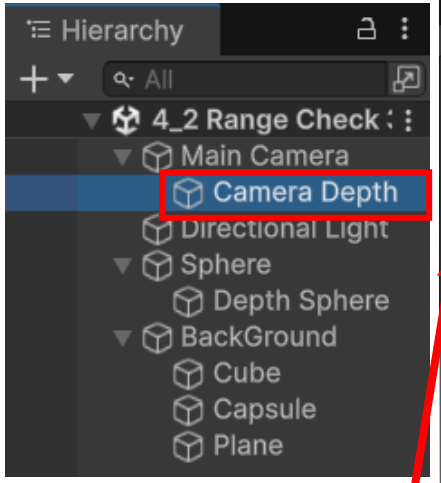
- 深度を描画する
- レンダーテクスチャの生成
  - 命名例: Depth Render Texture
  - サイズ:フレームバッファに合わせる
  - フォーマット: 深度バッファに合わせる
  - 深度バッファ:フレームバッファに合わせる
    - ただし、今回は球しか描画しないのでなくても良い

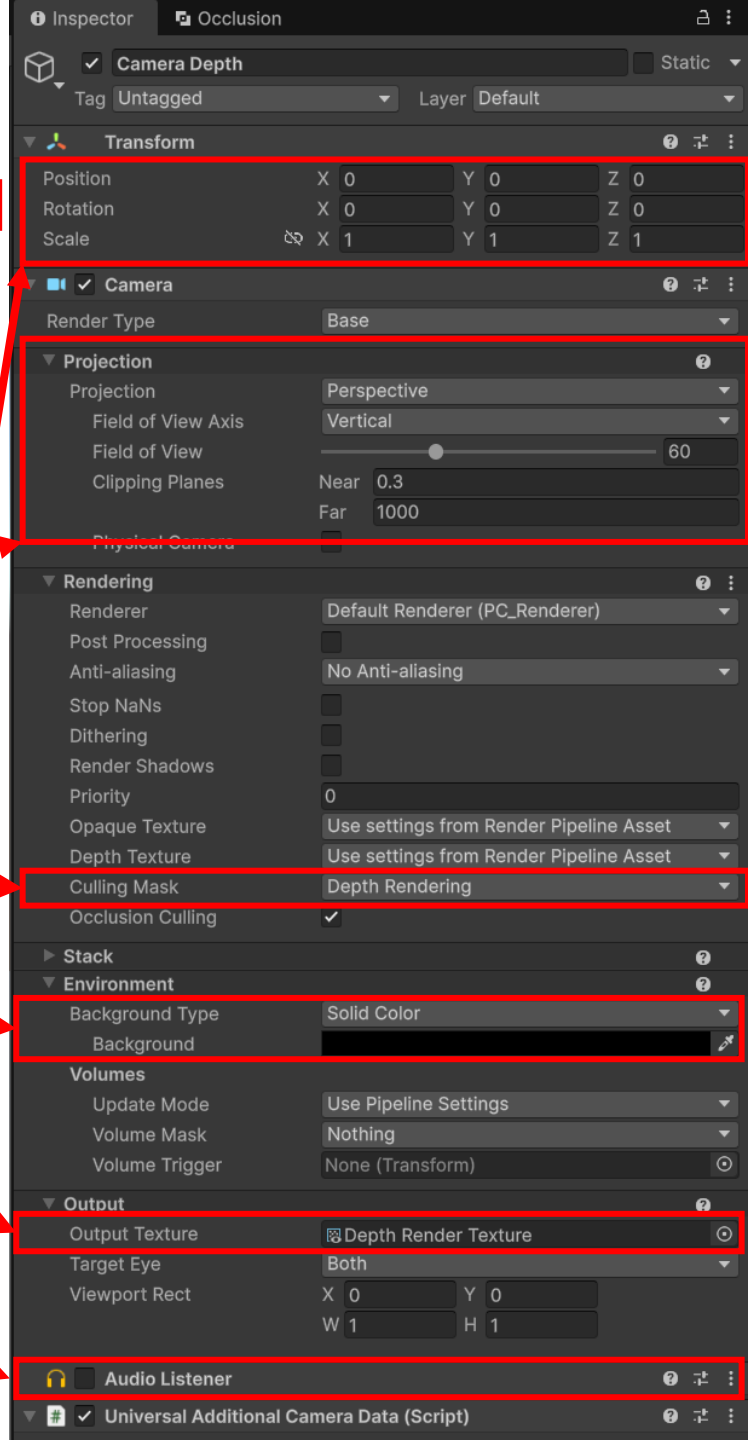




記録用カメラ

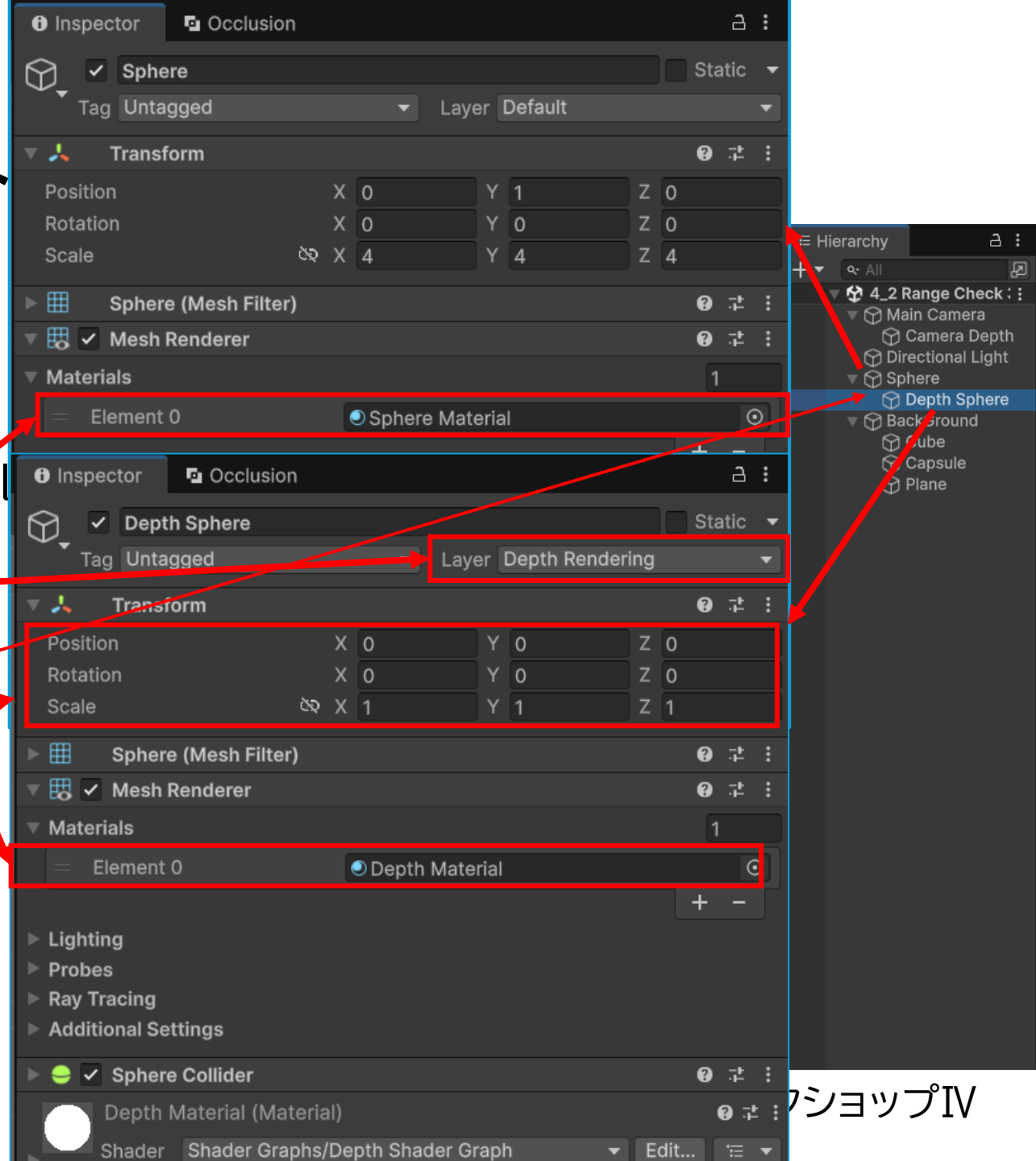
- メインカメラの子供にカメラを追加
  - 名称例：Camera Depth
  - 位置：メインカメラと同じ場所（原点、無回転、非拡大）
  - 投影：メインカメラに合わせる
  - クリップマスク：深度記録用のレイヤーを追加
    - レイヤー名例：Depth Rendering
  - 背景色：黒で初期化
  - 描画先：追加したレンダーテクスチャ
  - Audio Listenerは外す





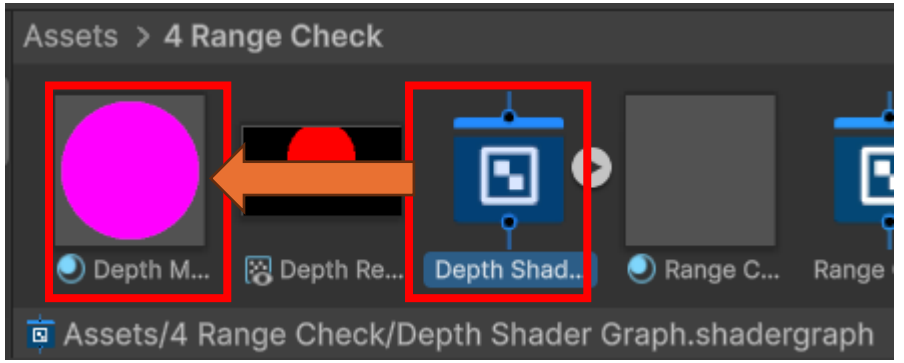
範囲形状オブジェクト

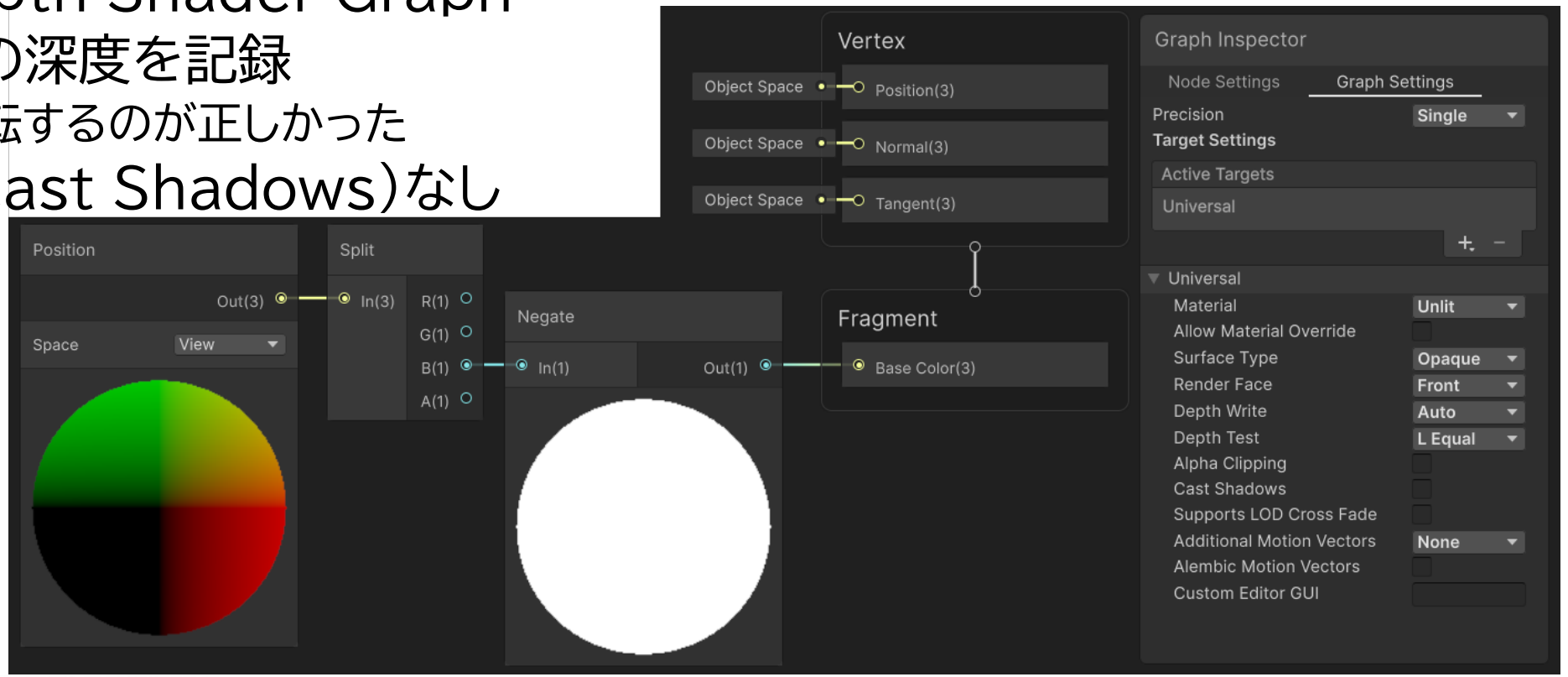
- 2つの球オブジェクトの追加
  - Sphere: 裏面の描画
    - マテリアル追加: Sphere Material
  - Depth Sphere: 深度の描画
    - Layerを設定
      - レイヤー: Depth Rendering
    - Sphereの子オブジェクト
      - 位置は原点
    - マテリアル追加: Depth Material



Depth Material

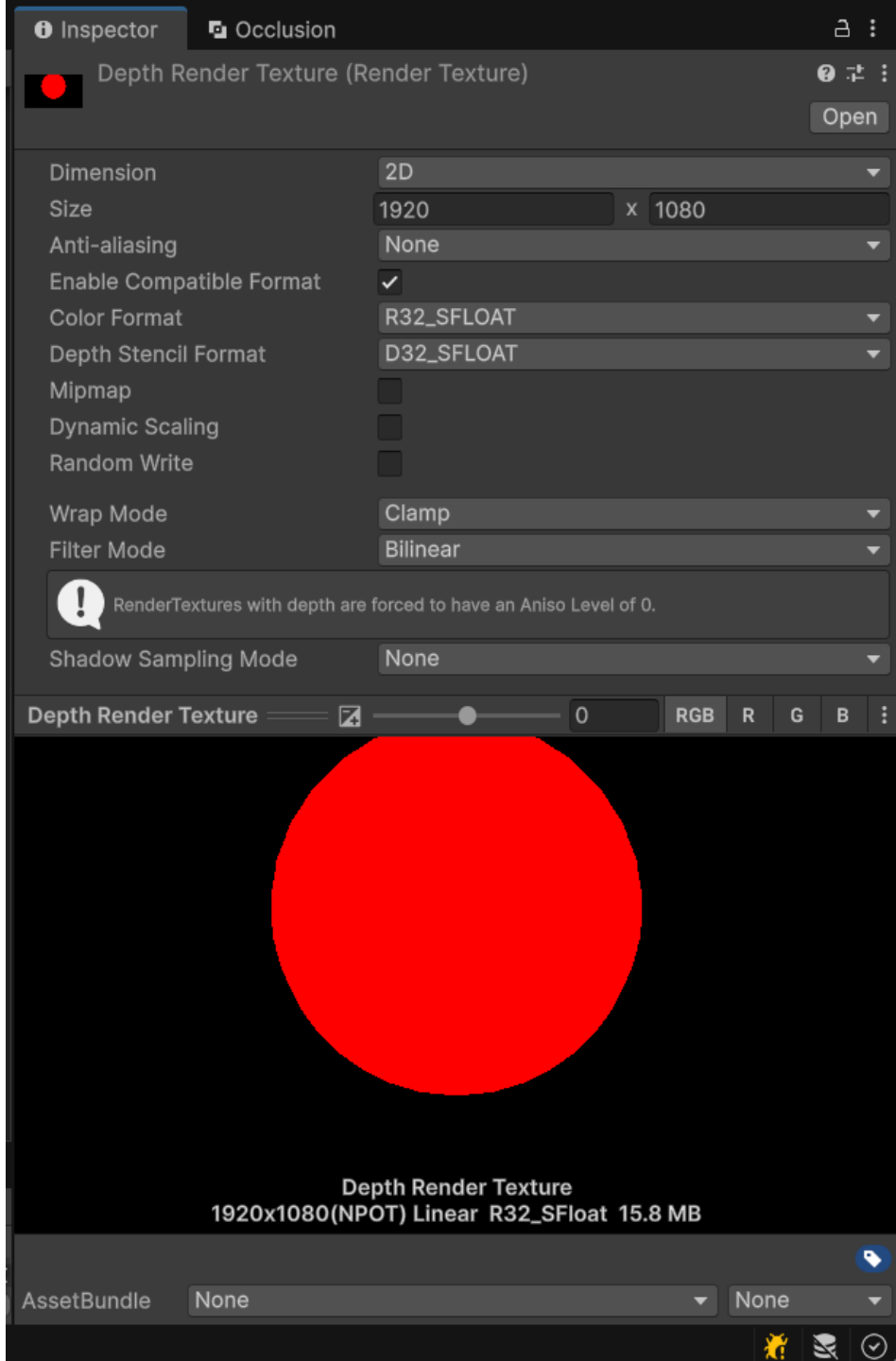
- Unlit Shader Graphも追加して設定
  - 命名例:Depth Shader Graph
  - カメラからの深度を記録
    - 符号を反転するのが正しかった
  - 影の処理(Cast Shadows)なし



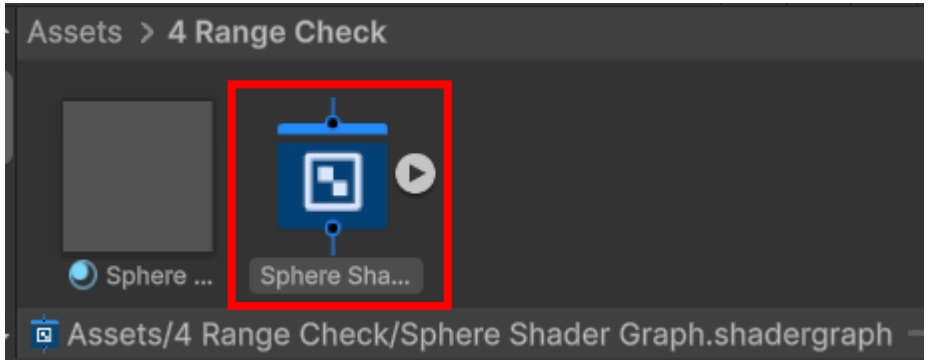


やってみよう

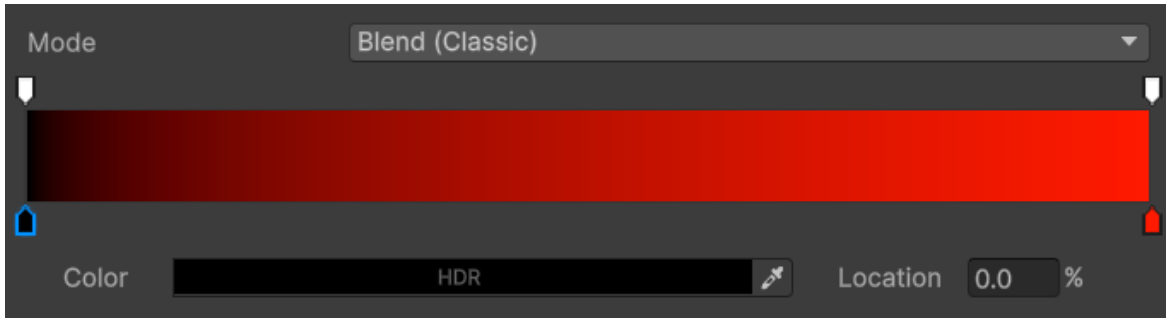
- 実行するとレンダーテクスチャのInspectorで描画結果を確認できる
- R成分しか持っていないので、可視化されたものは大きな値が赤く描画される
  - 後から描く固定色の赤色ではない

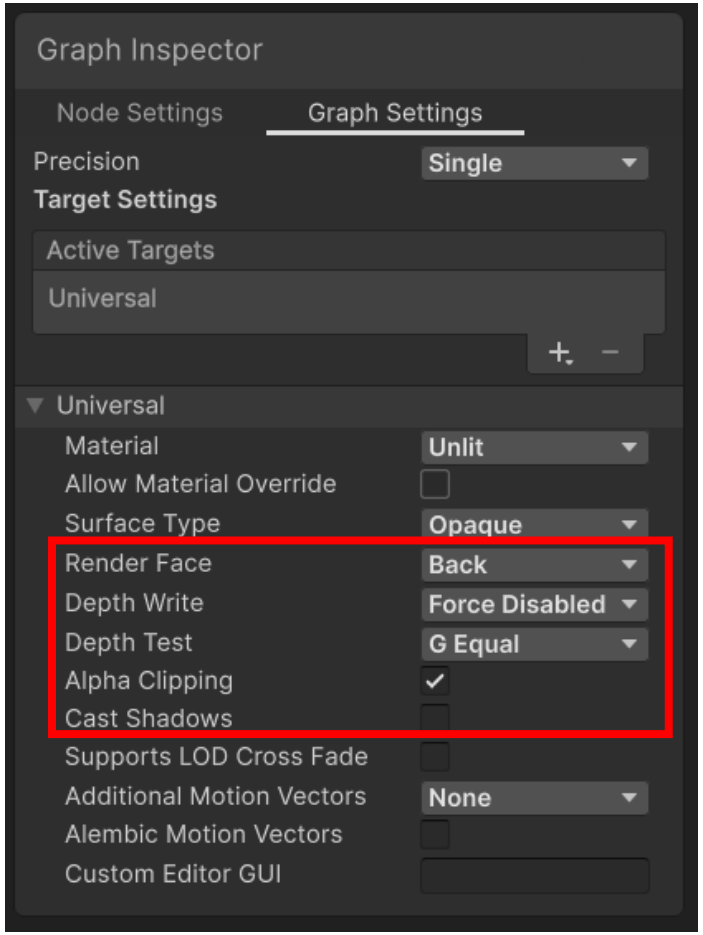


球の最終描画



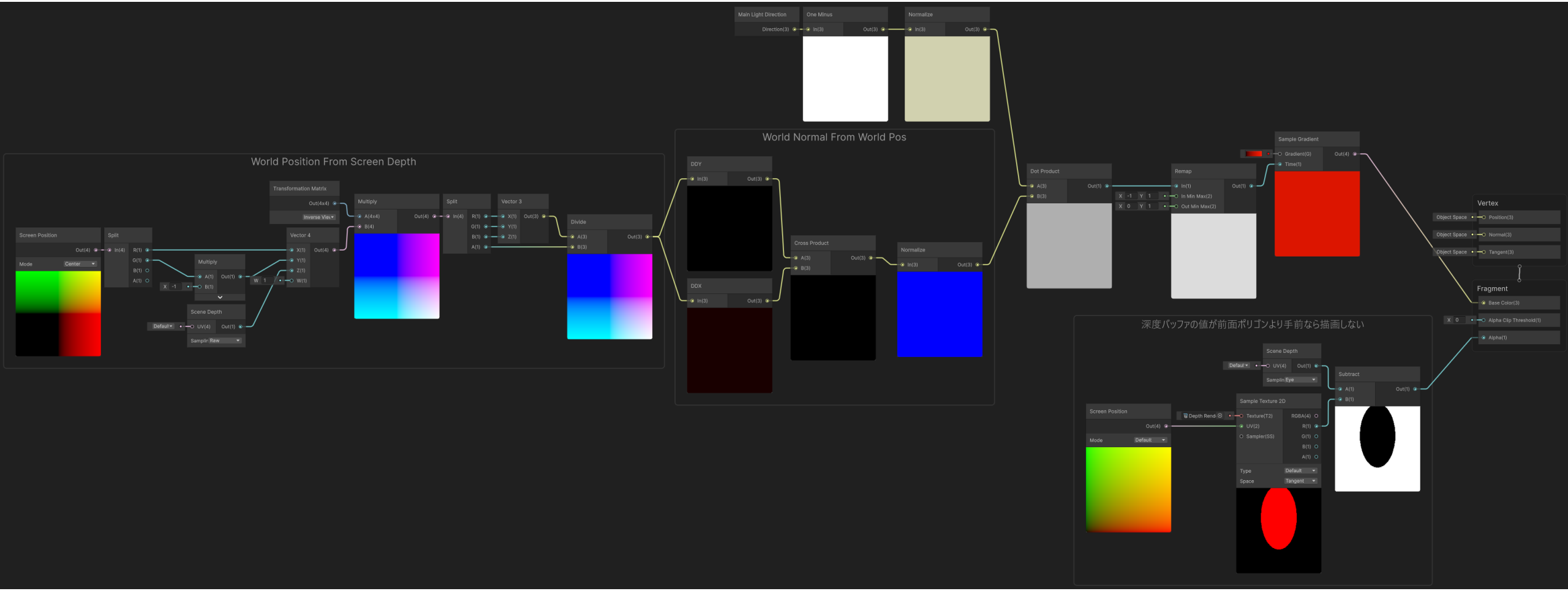
- Shader Graphを作成
  - 「4 Range Check/Sphere Material」に設定
    - なければ追加してSphereオブジェクトに追加
- Shader Graphを実装(ノード構成は次頁)
  - グラデーションを追加
    - 色に変化を与える
    - 固定色でも良い





裏面描画
深度書き込み不要
裏面用にテスト反転
αクリッピング
影用の処理はなし

球のシェーダグラフ


World Position From Screen Depth
World Normal From World Pos
深度バッファの値が前面ポリゴンより手前なら描画しない
Main Light Direction
One Minus
Normalize
Dot Product
Remap
Sample Gradient
Vertex
Fragment
Transformation Matrix
Multiply
Split
Vector 3
Vector 4
Divide
Screen Position
Scene Depth
DDY
DDX
Cross Product
Subtract
Sample Texture 2D

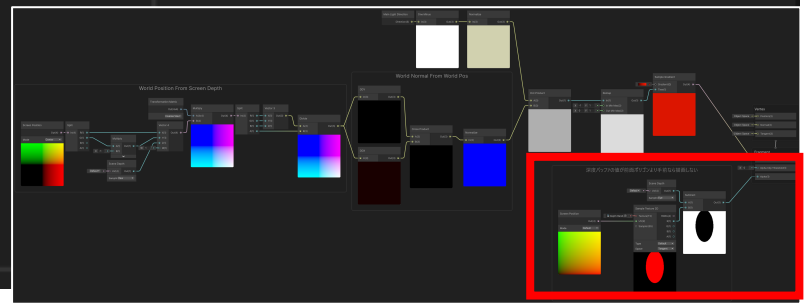
深度バッファの値が前面ポリゴンより手前なら描画しない

αクリッピングの閾値は0

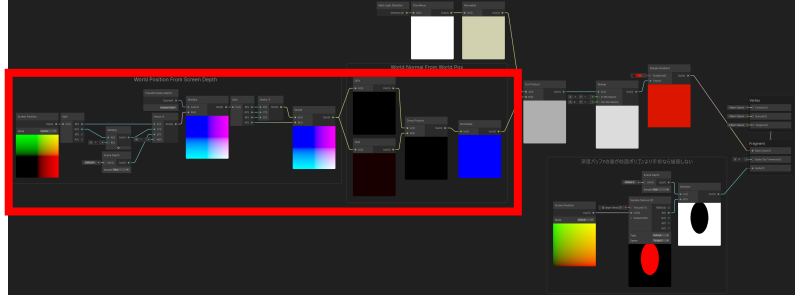
視点からの画素の距離を取得

描画時のピクセルの
位置からテクスチャ
を読み込み

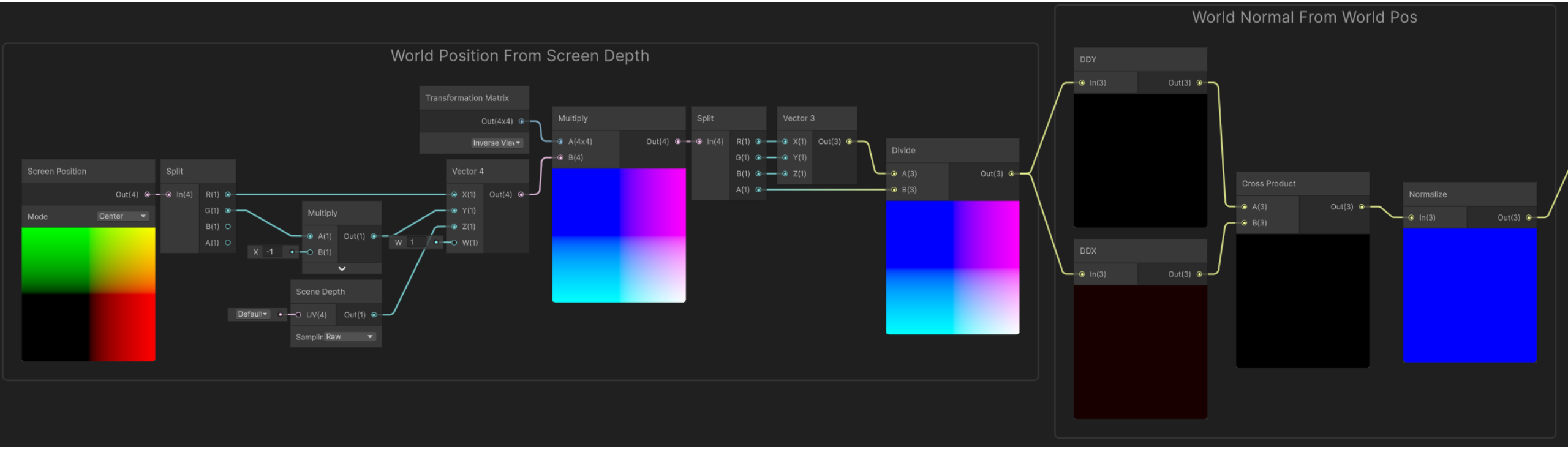
深度バッファに記録された現在の深度と
レンダーターゲットに記録した深度を比較し
レンダーターゲットの深度が遠ければ
クリッピングする



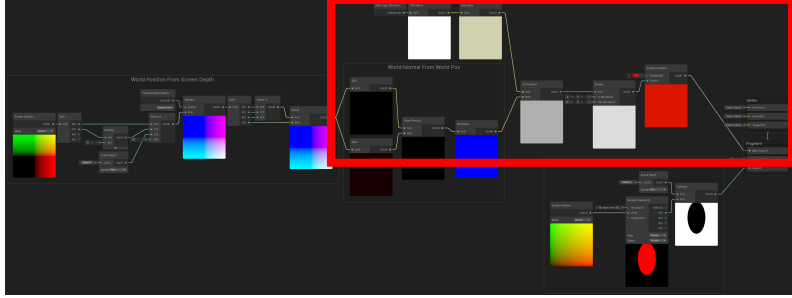
描画用の法線の導出

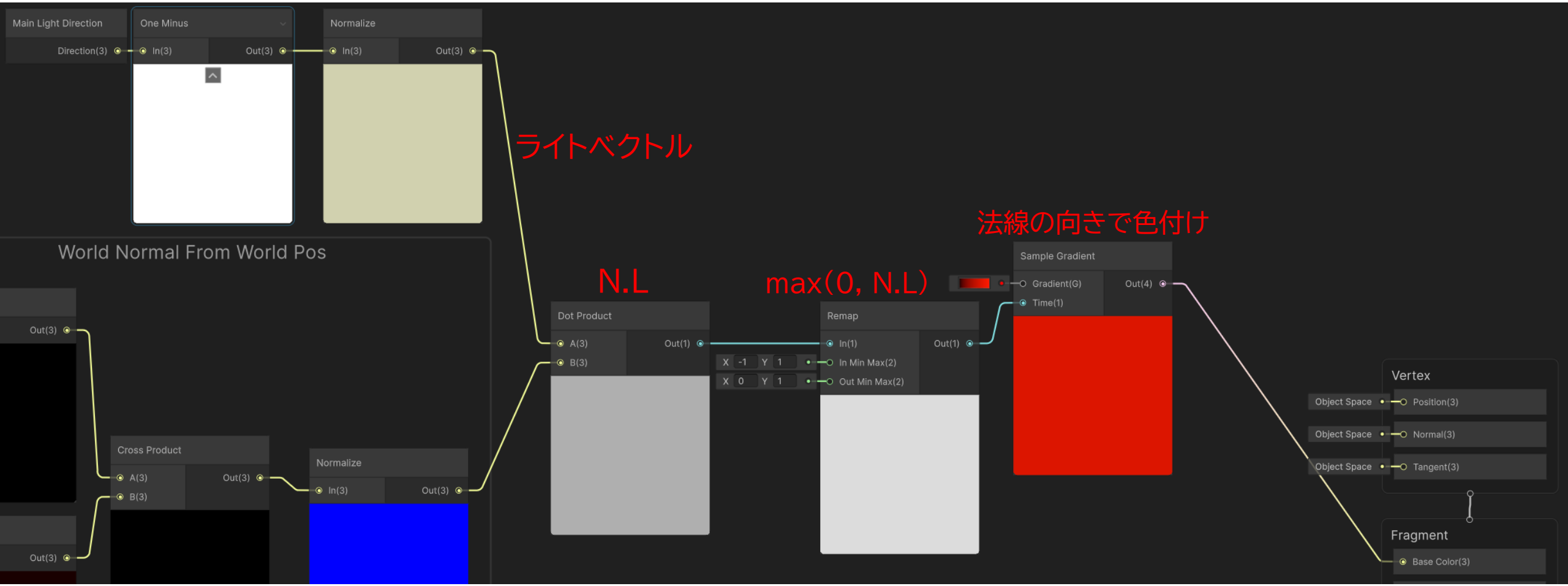


- ワールド座標系での位置を計算し、その勾配から法線を求める
  - 先に紹介した手法と同じ



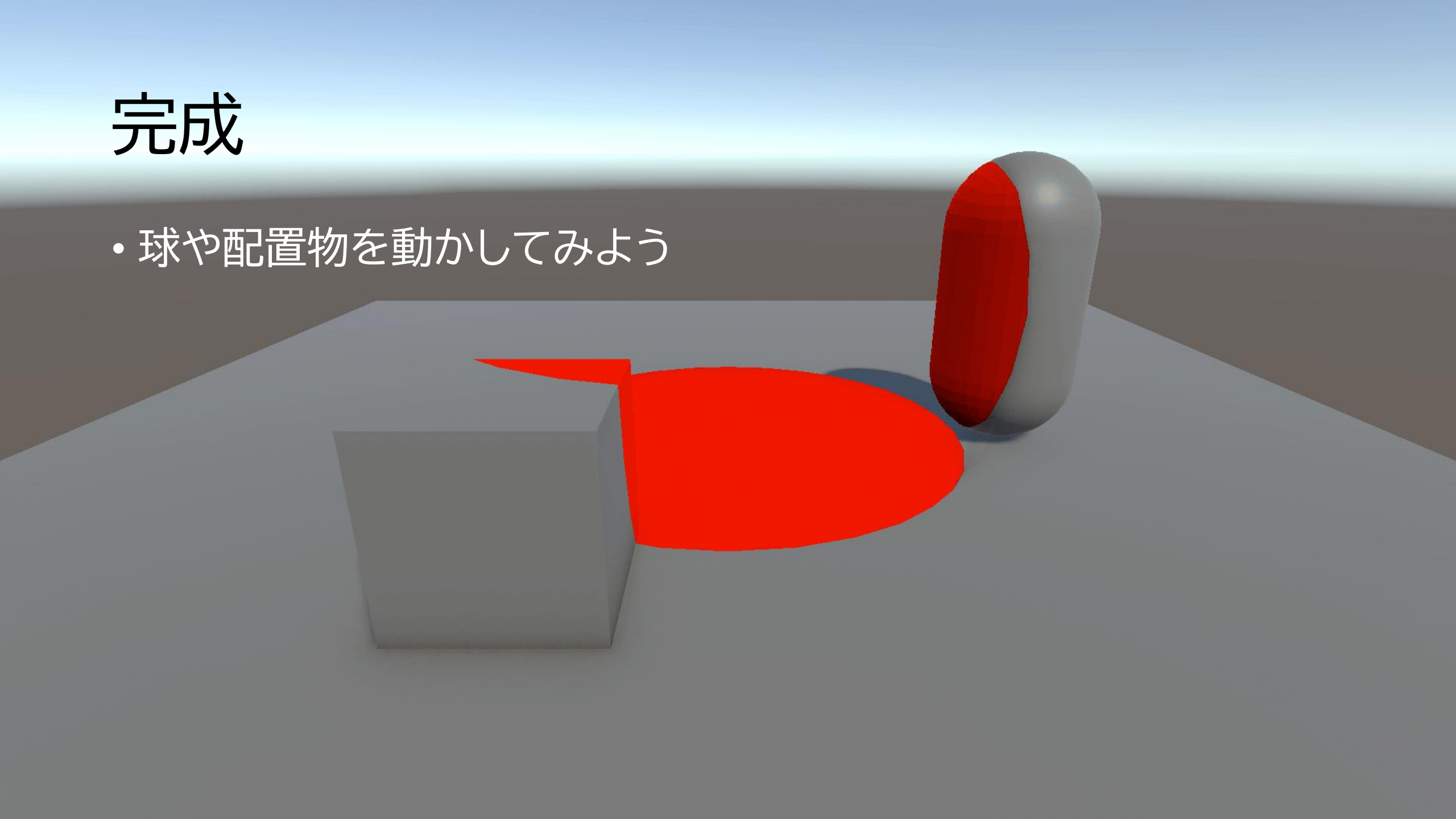
色の出力




Main Light Direction
Direction(3)
One Minus
In(3)
Out(3)
Normalize
In(3)
Out(3)
ライトベクトル
World Normal From World Pos
Out(3)
Out(3)
Cross Product
A(3)
B(3)
Out(3)
Normalize
In(3)
Out(3)
N.L
Dot Product
A(3)
B(3)
Out(1)
max(0, N.L)
Remap
In(1)
Out(1)
X -1 Y 1
In Min Max(2)
X 0 Y 1
Out Min Max(2)
法線の向きで色付け
Sample Gradient
Gradient(G)
Time(1)
Out(4)
Vertex
Object Space
Position(3)
Object Space
Normal(3)
Object Space
Tangent(3)
Fragment
Base Color(3)

完成

- 球や配置物を動かしてみよう



まとめ

- レンダーターゲット
  - レンダーターゲットの概要
  - 不透明フレームバッファへのアクセス
    - 空間をゆがませてみた
  - 深度からの位置・法線の復元
    - ポストエフェクトとして光の投影を実現
- レンダーテクスチャ
  - レンダーテクスチャの概要
  - ゲーム内モニター
    - ゲーム内に別空間の表示
  - バックミラー
    - 同じ空間の別の場所からの描画を表示
  - 範囲内のオブジェクトだけ単色
    - 今までの例の3次元空間的表現
    - オブジェクトのシェーダを複雑にせずに実現