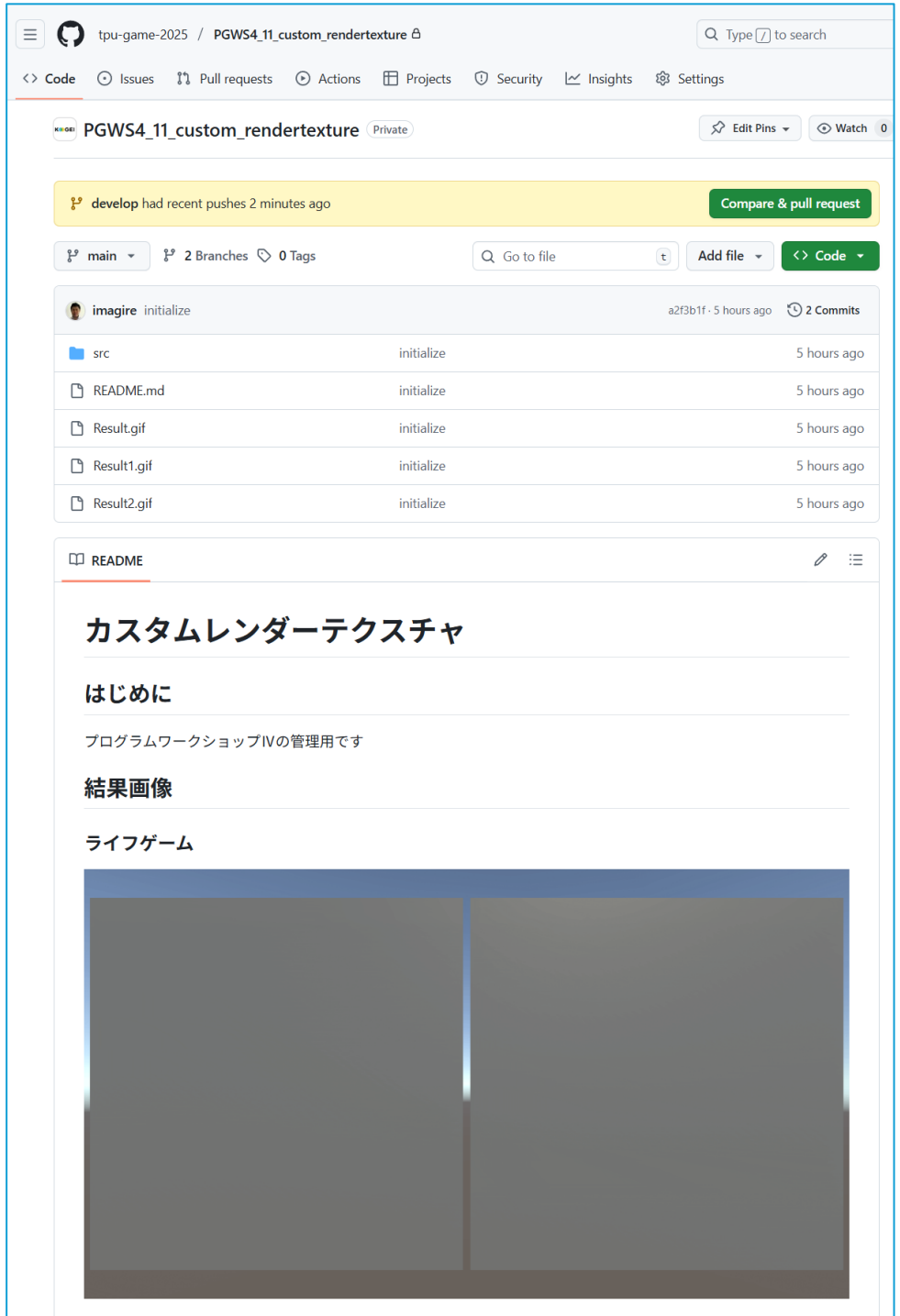
カスタム
レンダーテクスチャ

2025年度 プログラムワークショップIV（11）

今回のリポジトリ

- https://github.com/tpu-game-2025/PGWS4_11_custom_rendertexture



本日の内容

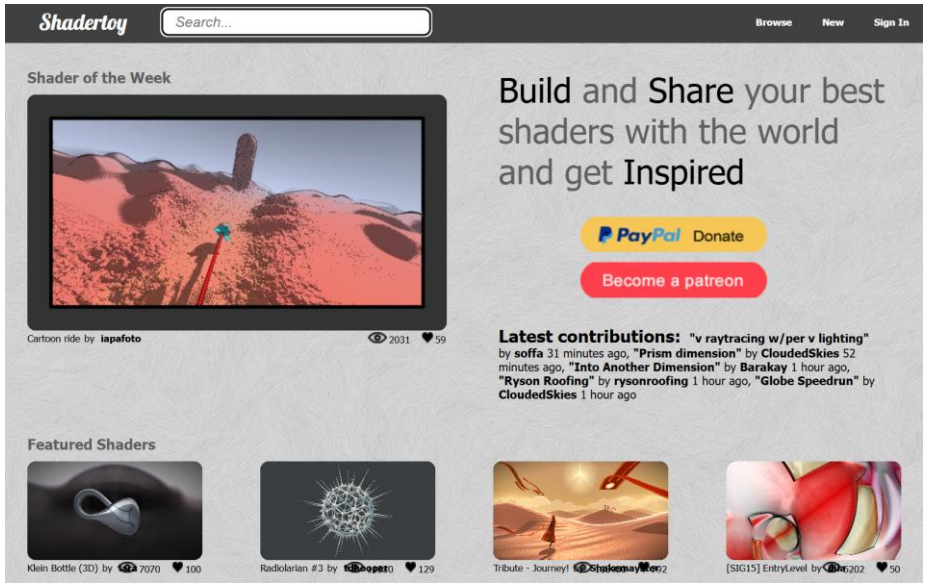
- カスタムレンダーテクスチャ
  - カスタムレンダーテクスチャの概要
  - ライフゲーム
  - カールノイズ

本日の内容

- カスタムレンダーテクスチャ
  - カスタムレンダーテクスチャの概要
  - ライフゲーム
  - カールノイズ

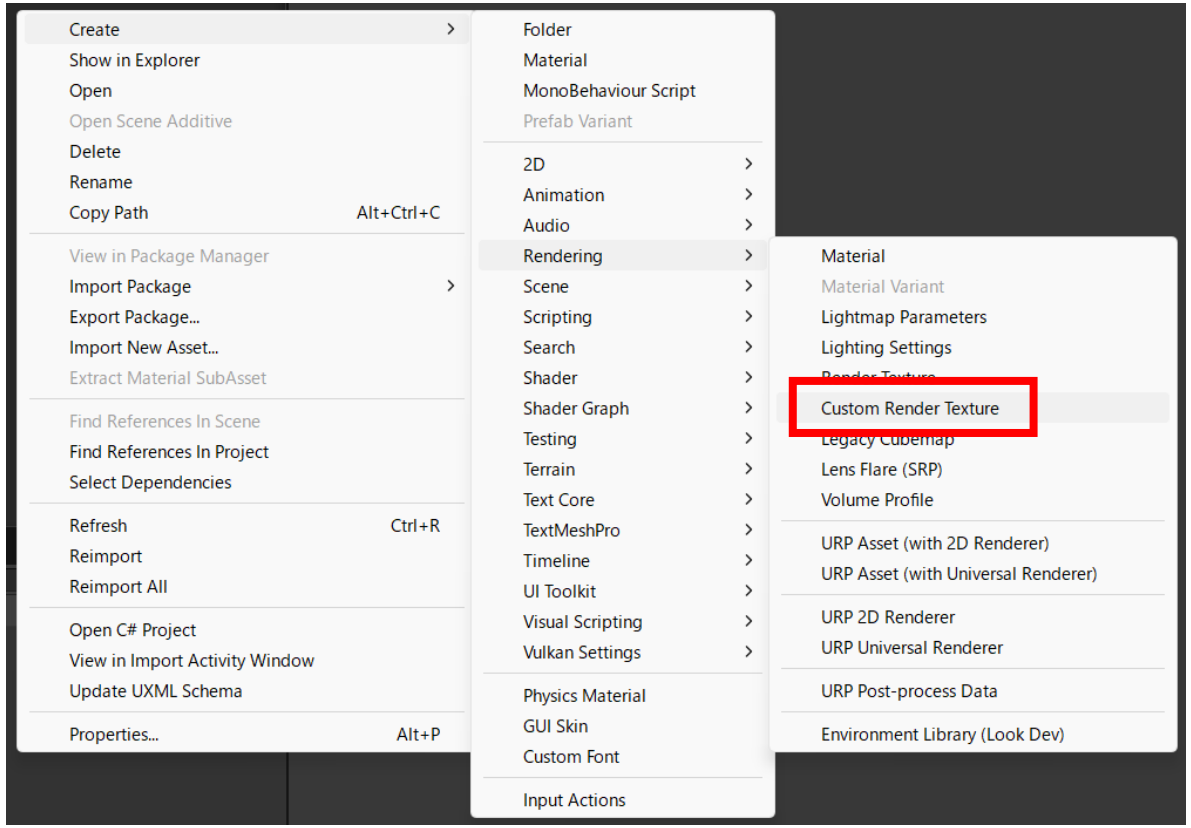
カスタムレンダーテクスチャの概要

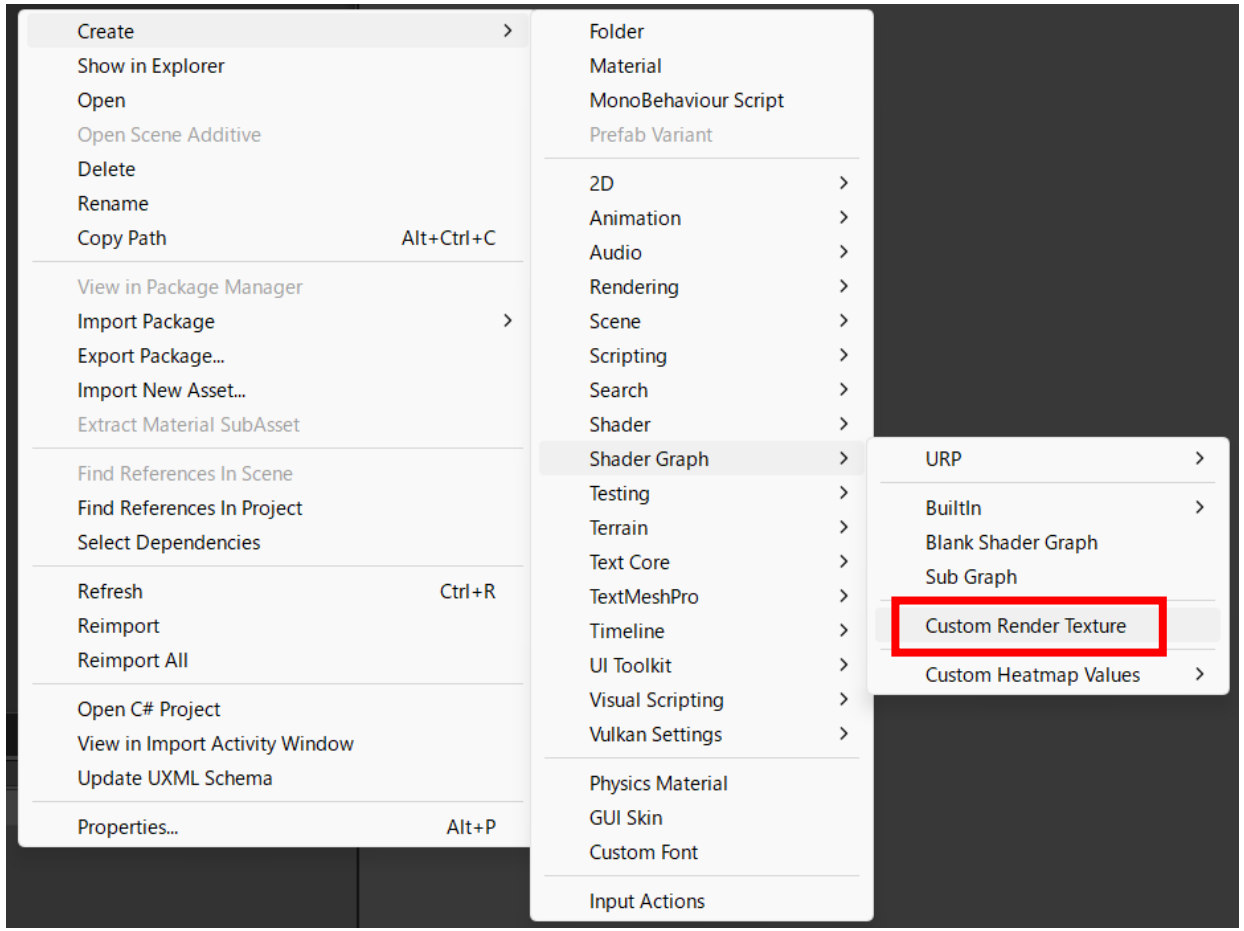
- レンダーターゲットの問題
  - 「カメラ」から見たものが描画される
  - レンダリングは必ずしもカメラに紐づかない
    - Shadertoyは全画面ポリゴンを一枚描画するだけ
    - 同様に全体を更新する仕組みが欲しくなる
- 画像全体の更新ならテクスチャ座標を使って、直接シェーダでレンダリングするのではだめなのか？
  - より複雑な表現が可能になる
    - 前のフレームの描画結果を使って実現する動的な処理



カスタムレンダーテクスチャ

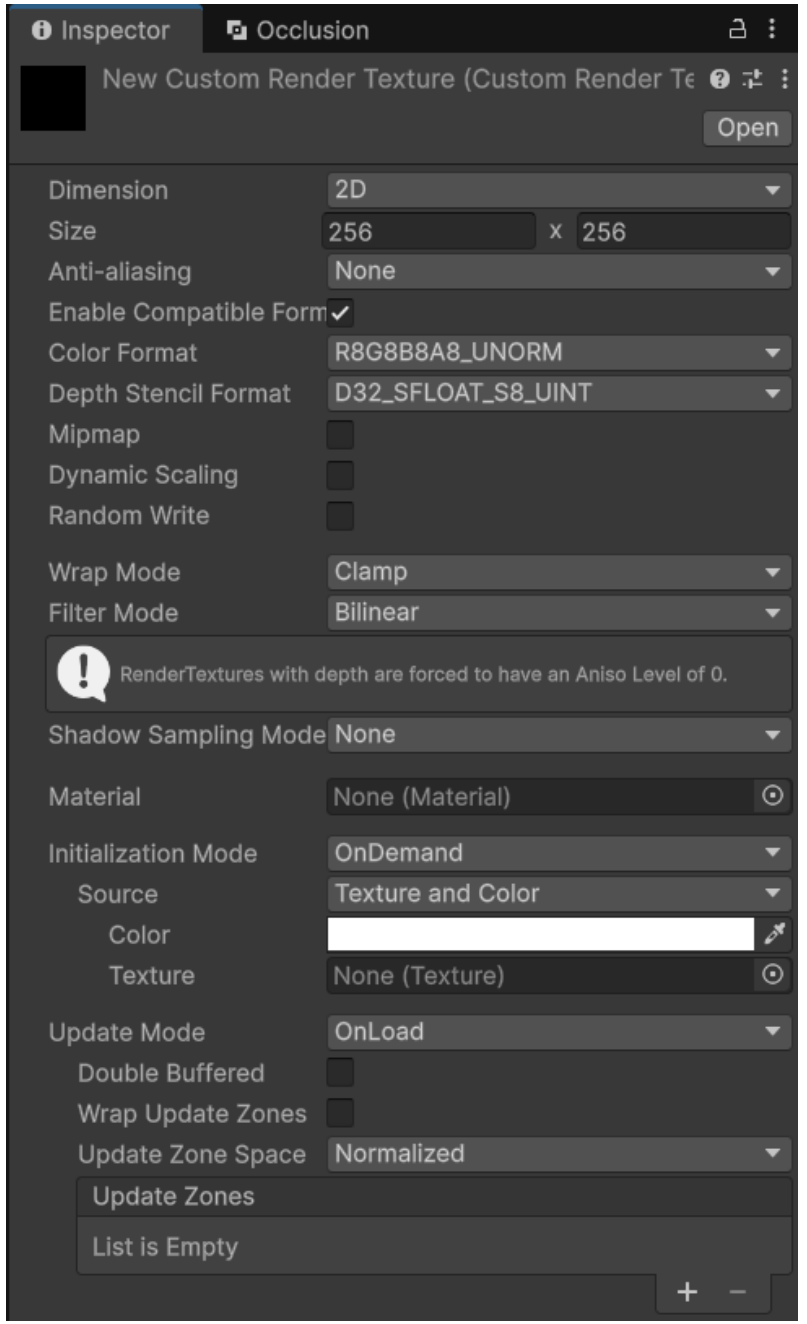
- 専用のテクスチャとシェーダ





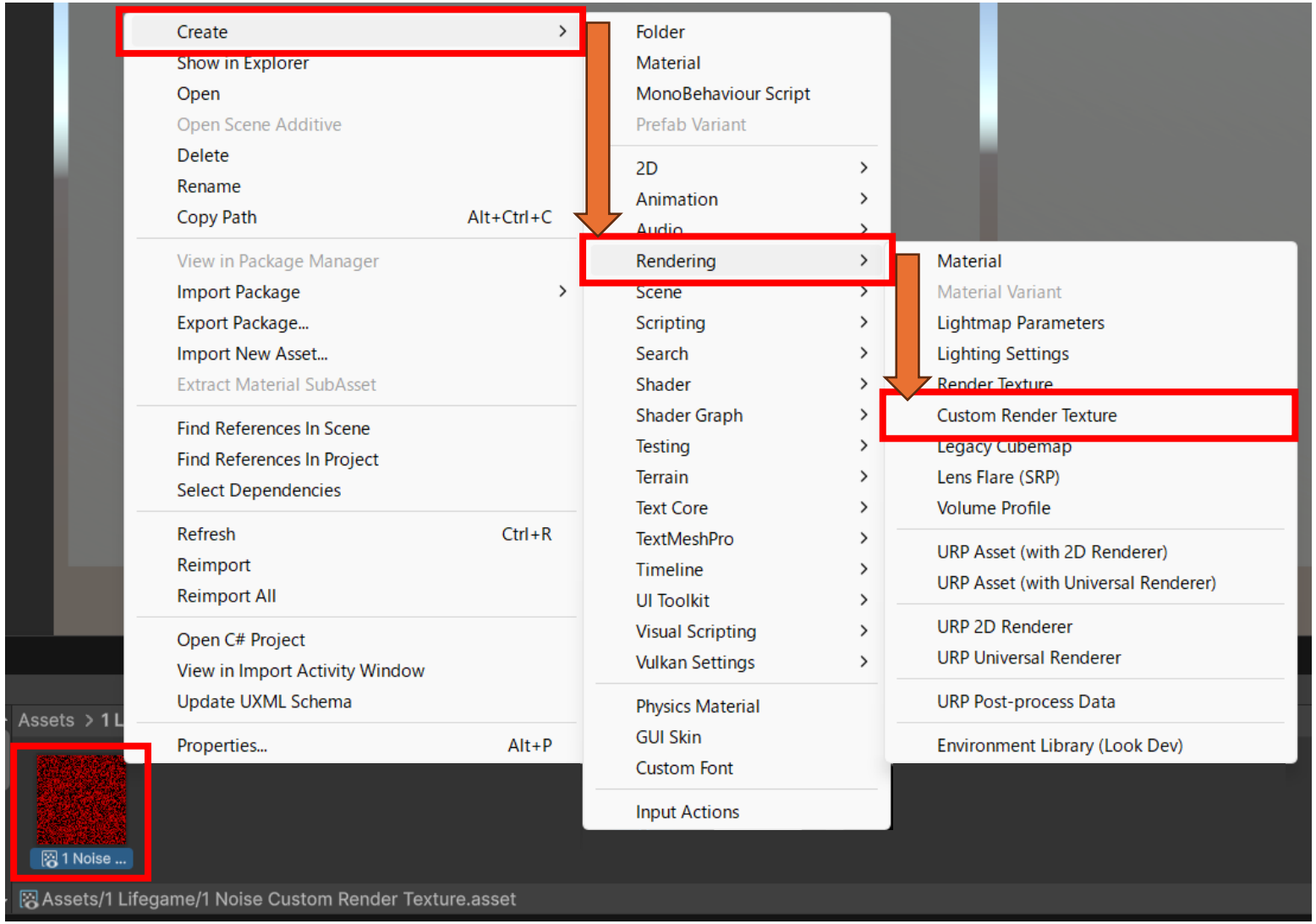
カスタムレンダーテクスチャ

- マテリアルの設定：更新するシェーダの指定
- Initialize Mode: 初期化タイミング
  - テクスチャやマテリアルを使って書き換えられる
    - On Load: 一度だけ更新（静的テクスチャ用）
    - Real Time: 毎フレーム更新
    - On Demand: スクリプトから更新（開始時に更新など）
- Update Mode: 更新タイミング
  - Double Buffered: 前フレームの結果をテクスチャとして読み込める
  - Update Zone: 更新範囲を指定



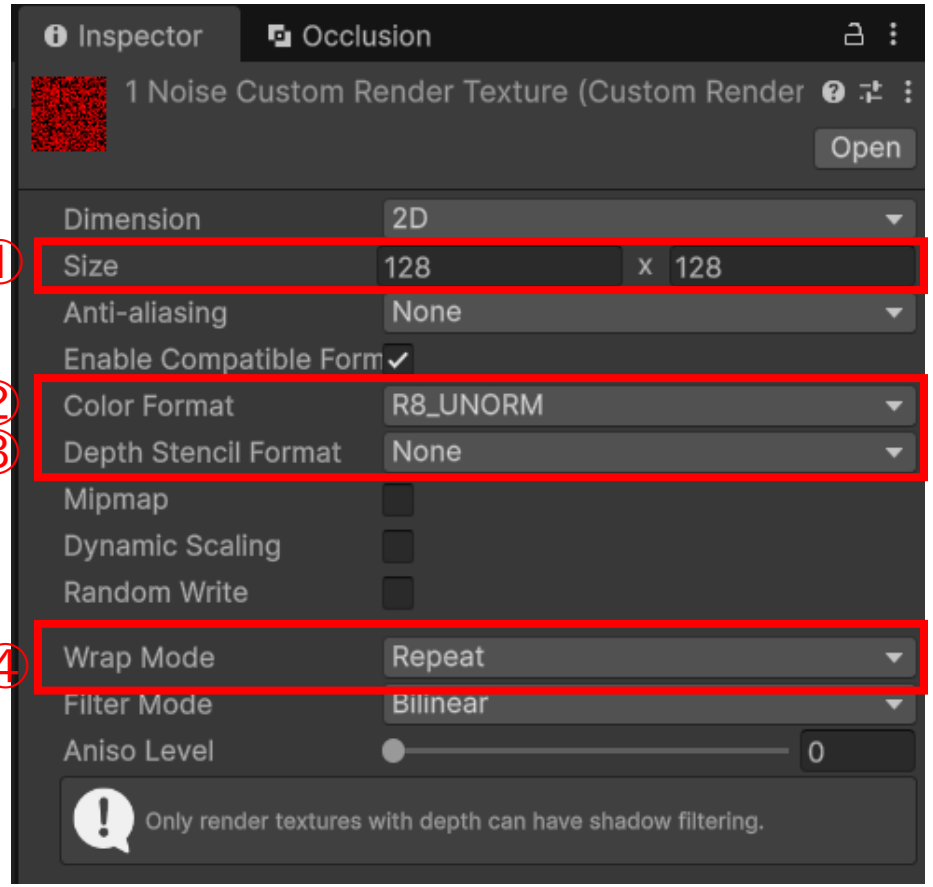
やってみよう：カスタムレンダーテクスチャの生成

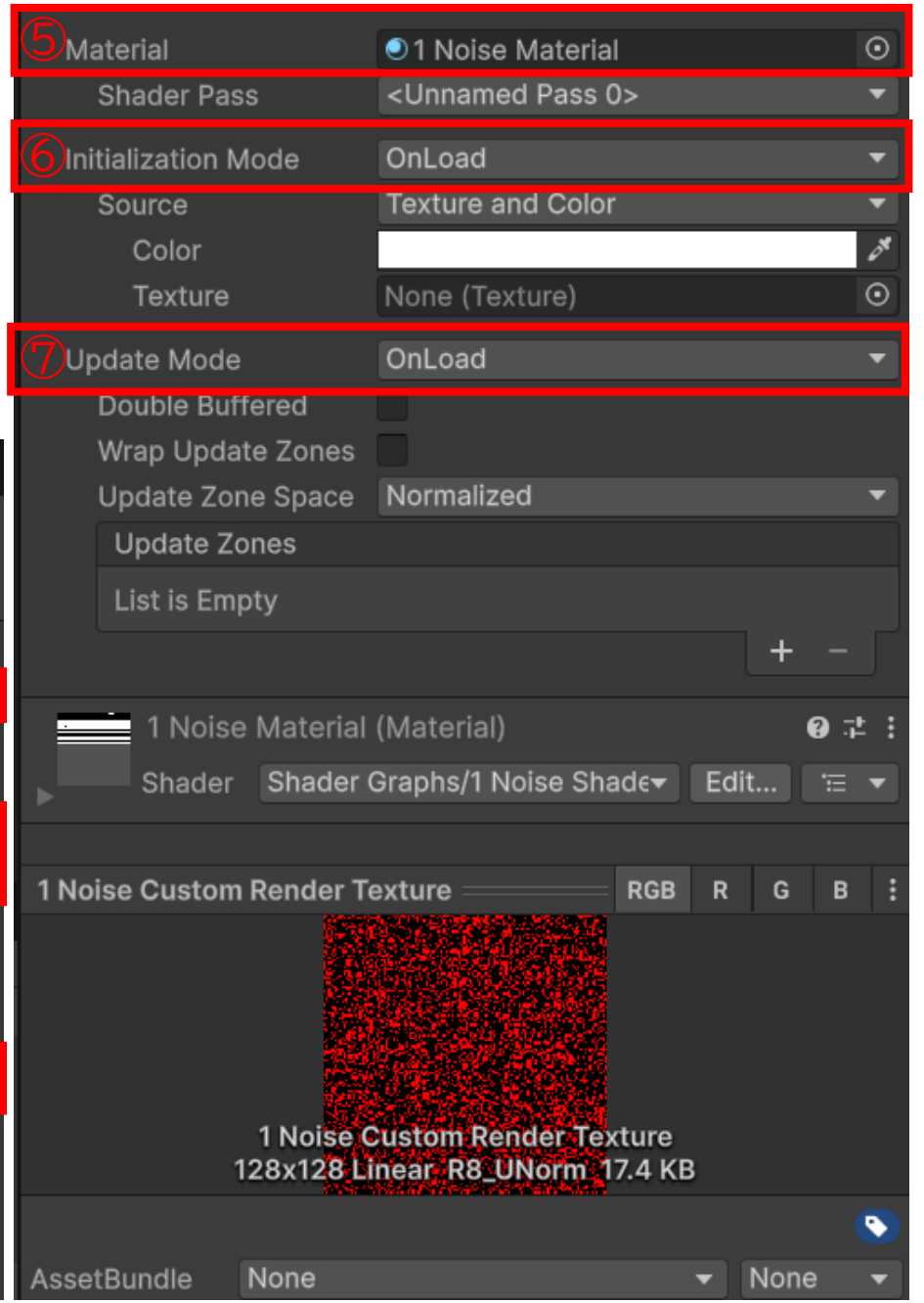
- 「Create」-「Rendering」-「Custom Render Texture」から生成
- 名前を設定
  - ここでは、「1 Lifegame」ディレクトリに「1 Noise Custom Render Texture」という名称



カスタムレンダーテクスチャの設定

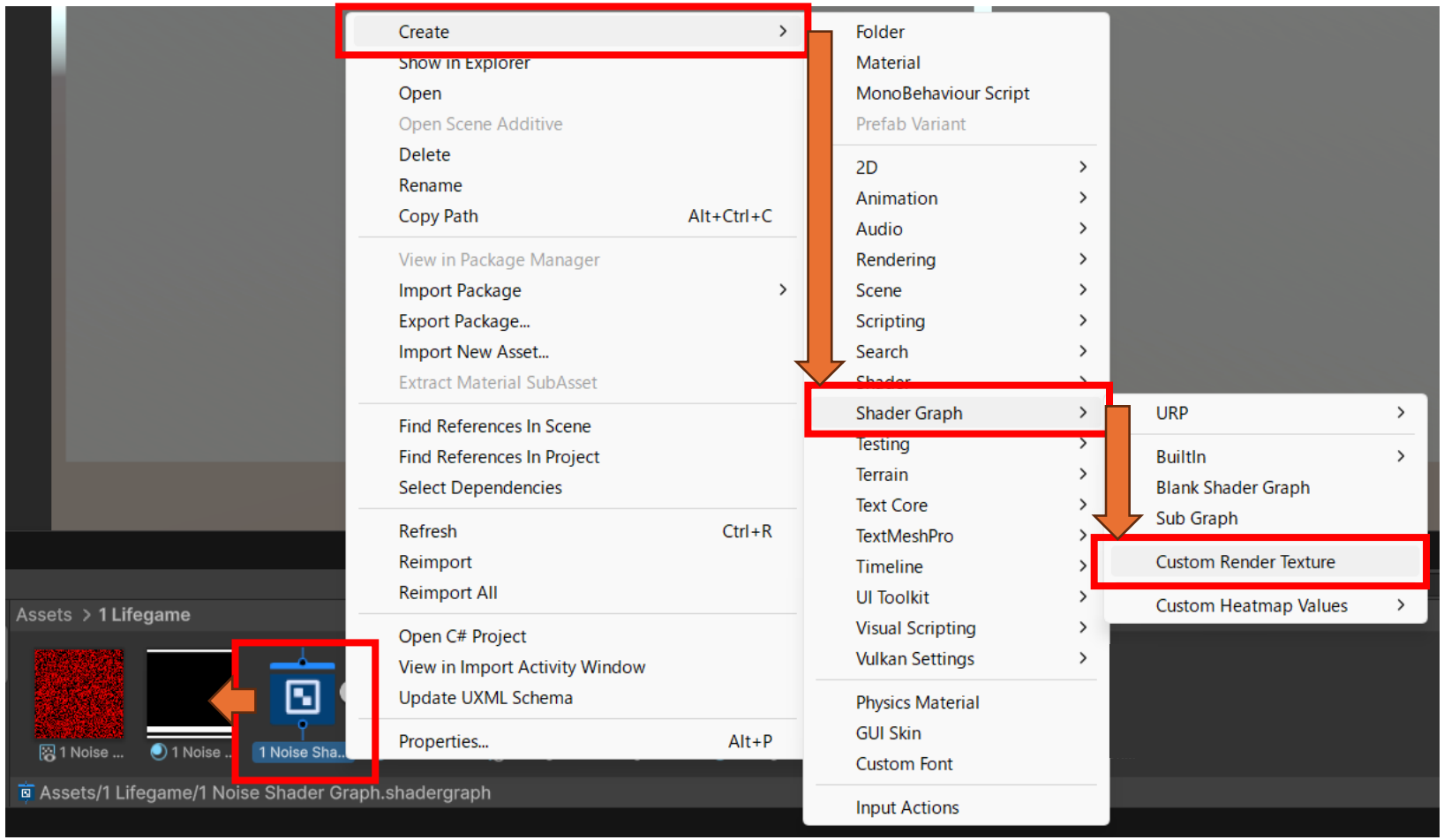
1. 128×128
2. 単色
3. 深度バッファなし
4. Wrapはリピート
5. 更新:「1 Noise Material」
   - 次で、シェーダグラフを作成
6. 初期化: 適当
   - Updateの後に上書きされないようにOnLoad
7. 更新: OnLoad
   - 一度だけ更新



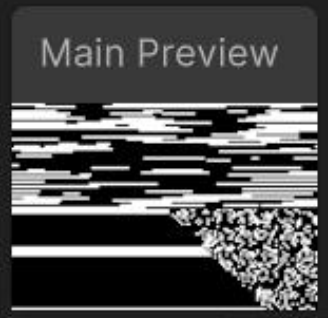


カスタムレンダーテクスチャ用シェーダグラフの生成

- 「Create」-「Shader Graph」-「Custom Render Texture」から生成
- 名前を設定
  - ここでは、「1 Lifegame」ディレクトリに「1 Noise Shader Graph」という名称
- マテリアルに設定
  - ここでは、作成済みの「1 Noise Material」



シェーダグラフ



プレビューは正しくない

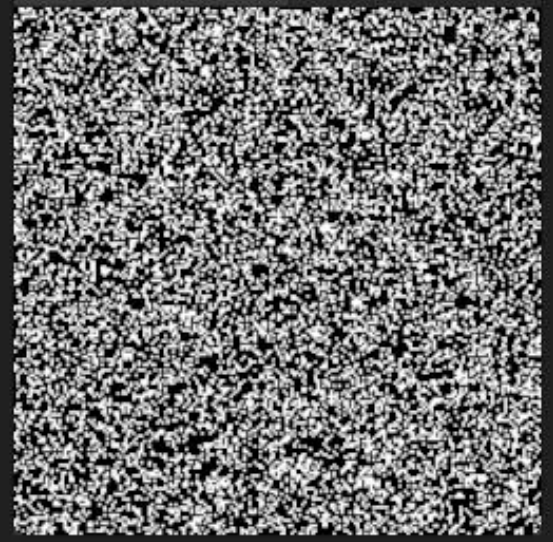
Vertex
頂点シェーダなし
Spacebar to Add Node

Fragment
Base Color(3)
Alpha(1)
X 1

Step
X 0.5
Edge(1)
In(1)
Out(1)

Simple Noise
UV0
UV(2)
Out(1)
X 1000
Scale(1)
1000
Hash Type
Determinist

とりあえず
大きな数

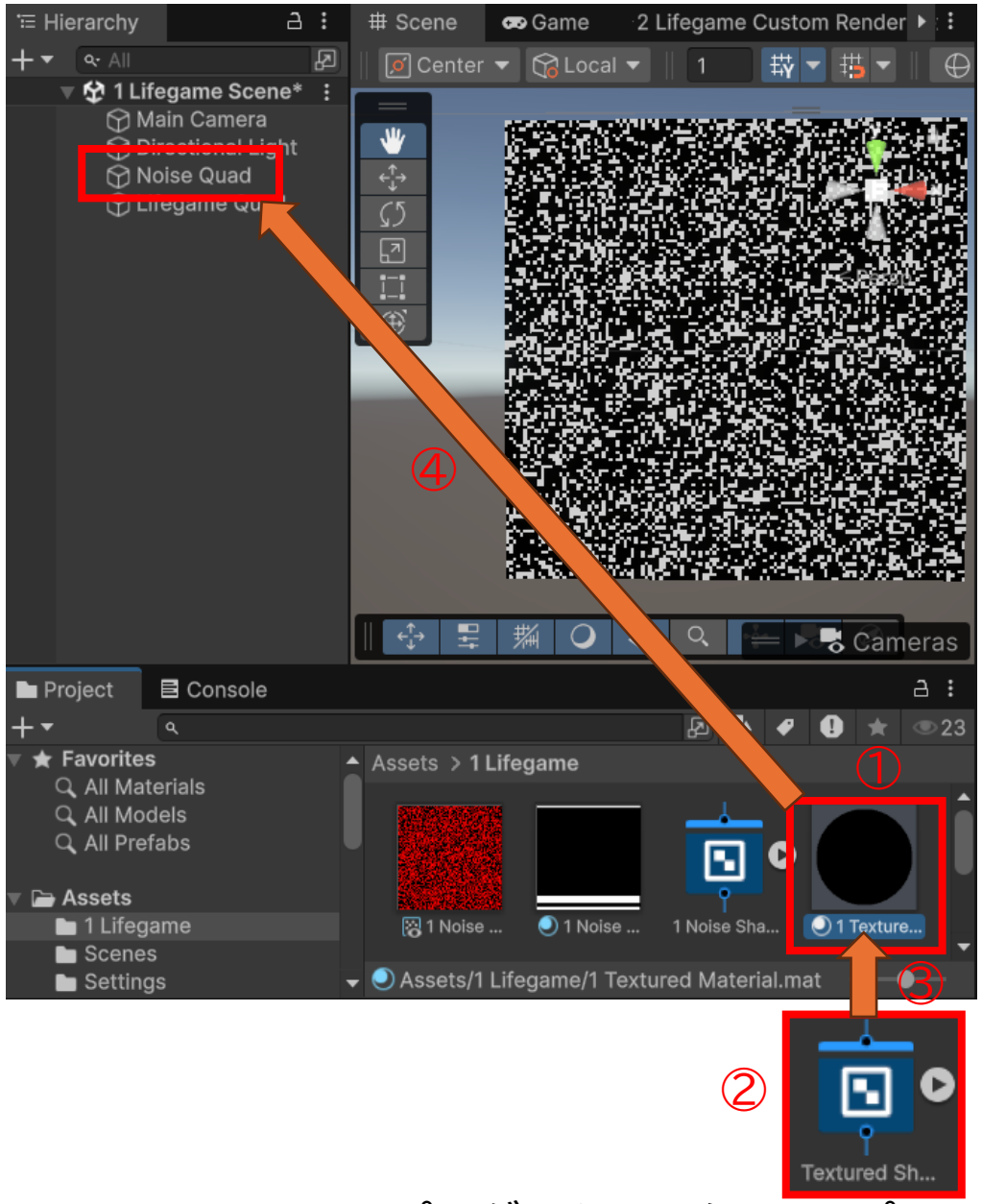


ここでは、白黒だけに
したかったので
Stepノードで分離

Graph Inspector
グローバル設定も特別
Node Settings
Graph Settings
Precision
Single
Target Settings
Active Targets
Custom Render Textur
Custom Render Texture
Material
Custom R
Custom Editor GUI

可視化

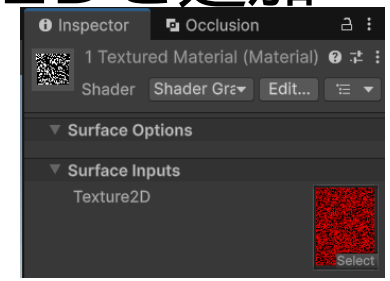
1. マテリアルの作成
   - 名称例:「1 Lifegame/1 Textured Material」
2. シェーダグラフの作成
   - Unlit Shader Graphで良い
   - 名称例:「Textured Shader Graph」
3. シェーダグラフをマテリアルに設定
4. マテリアルをバインド
   - 「Noise Quad」に



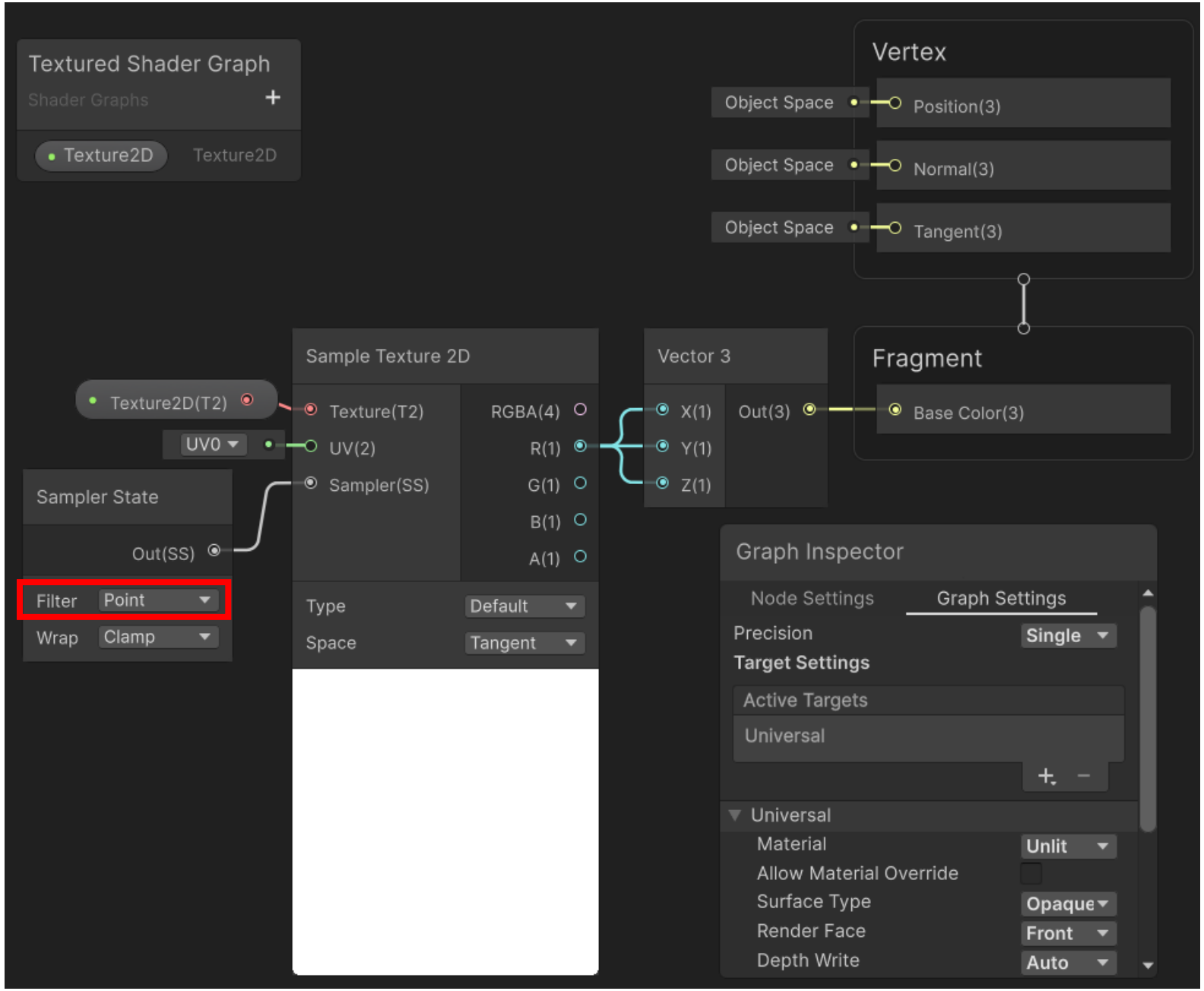
可視化のシェーダグラフ

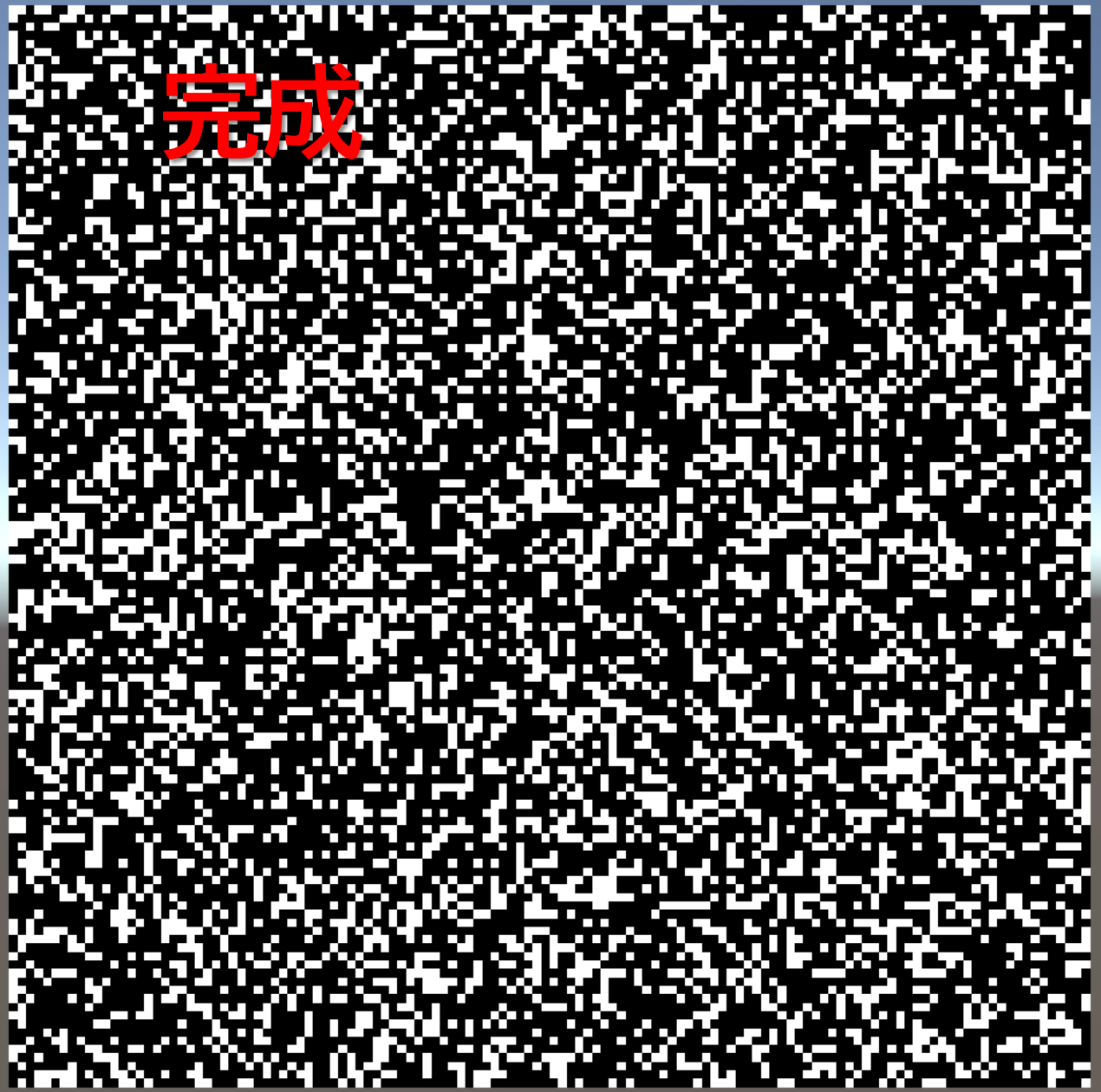
- Propertyに Texture2Dを追加

マテリアルで
カスタムレンダー
テクスチャを指定



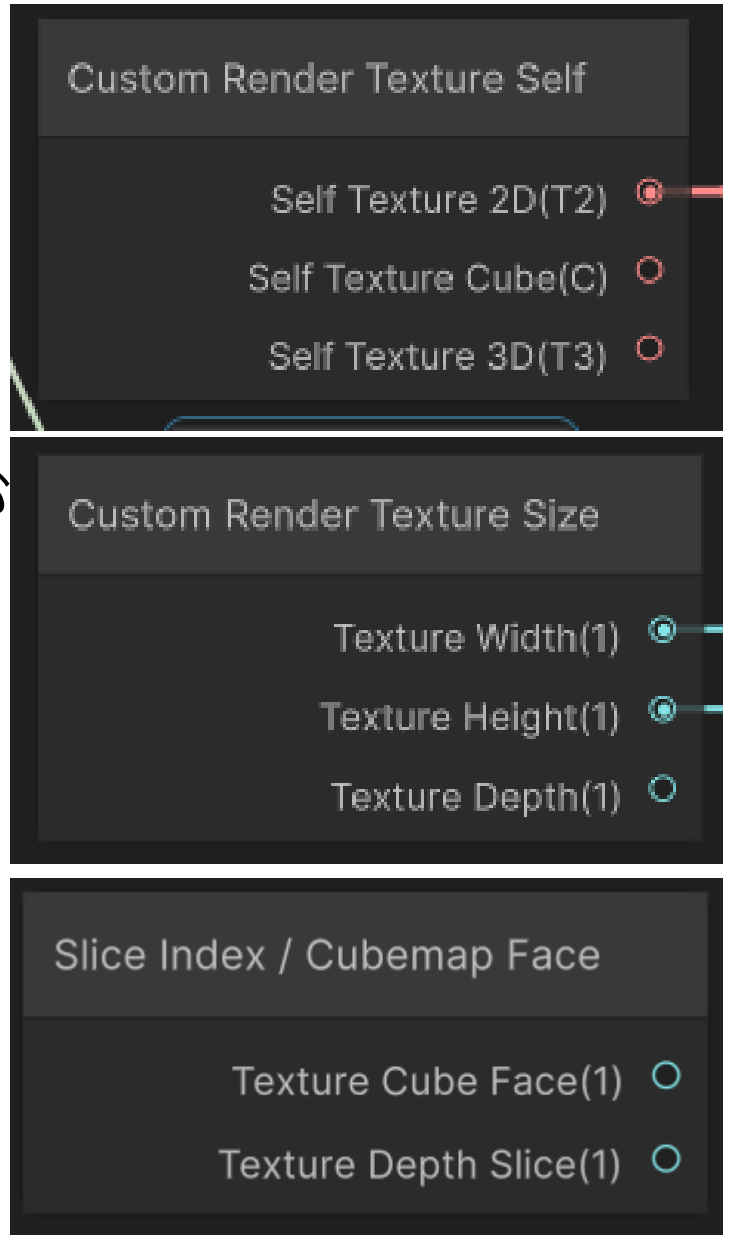
- サンプラーステート
  - ポイントサンプリング
    - くっきりと表示
- 赤成分をVector3の すべての成分に入れる
  - カスタムレンダーテクスチャが1成分なので




完成

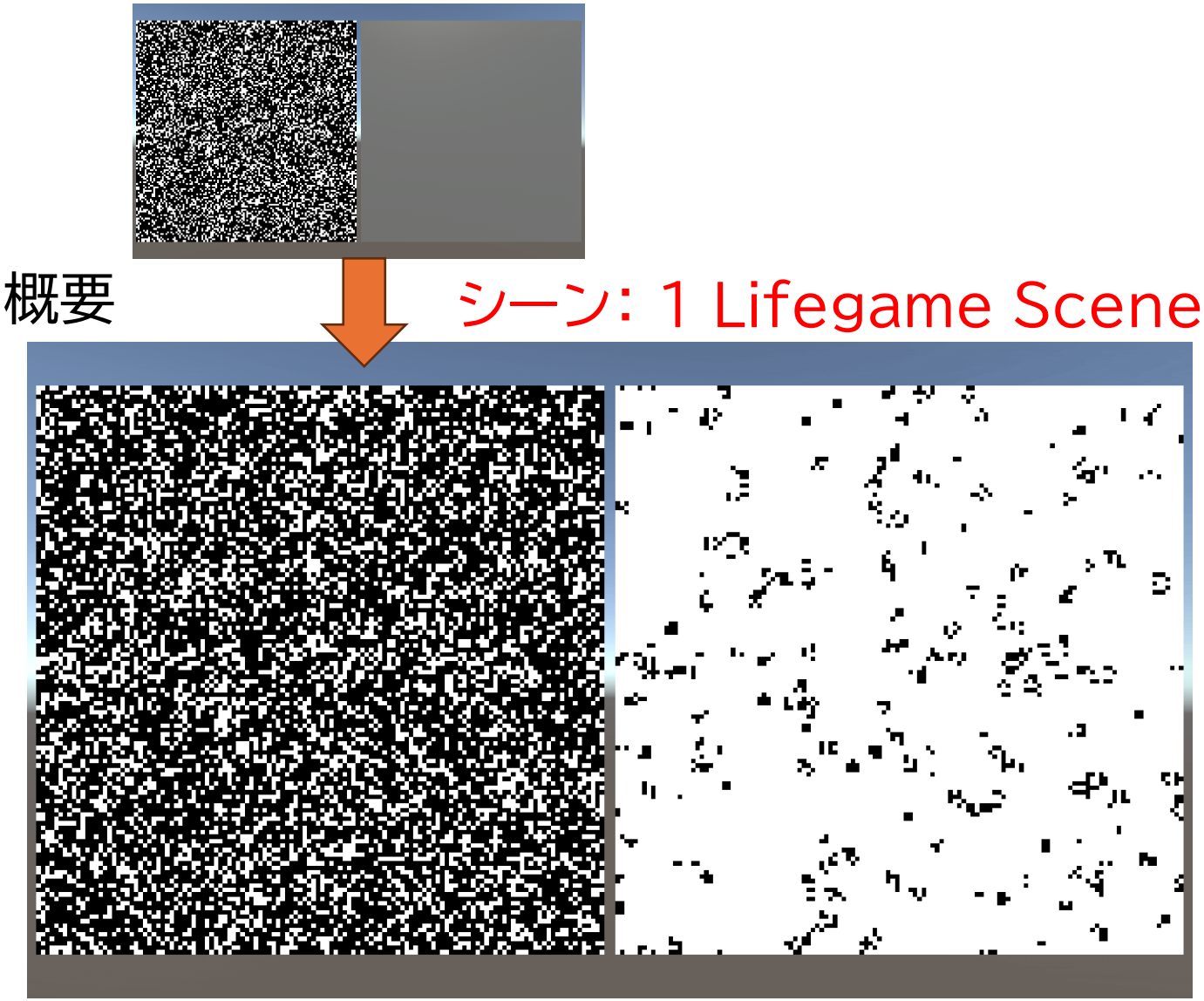
特別なノードの紹介

- Custom Render Texture Self
  - 直前の描画結果を読み込む
    - Double Bufferingを有効にして前のフレームを読み込む
- Custom Render Texture Size
  - 描画しようとしているカスタムレンダーテクスチャの大きさ
  - テクスチャ配列の場合は、Texture Depthとして、配列数を取得可能
- Slice Index / Cubemap Face
  - 描画しようとしているカスタムレンダーテクスチャのキューブマップの面や配列のインデックス



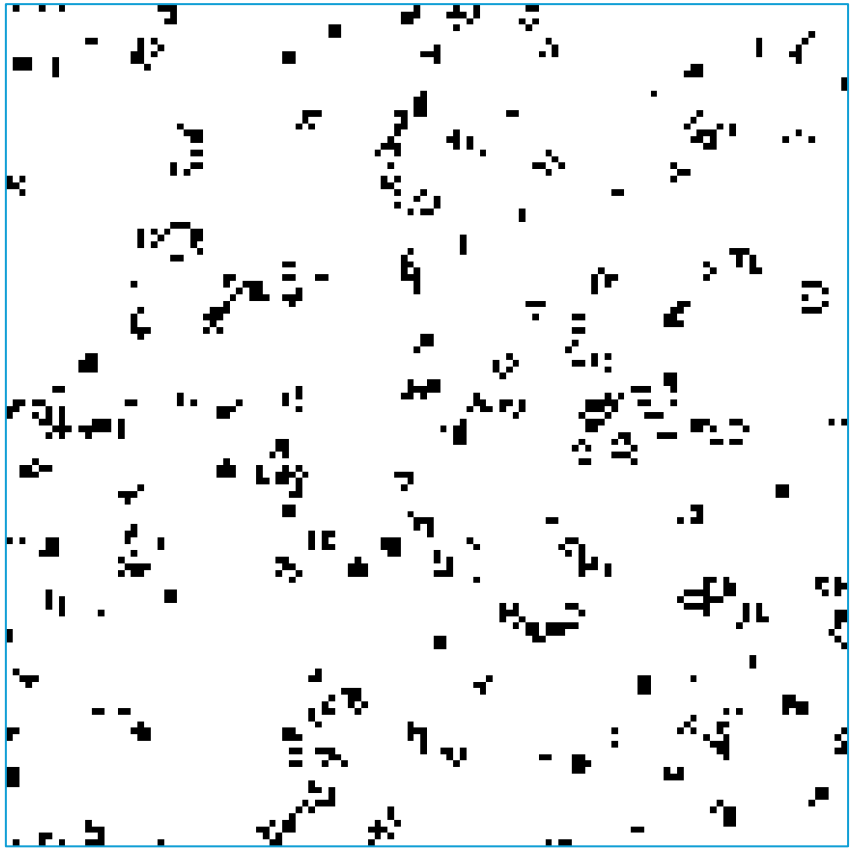
本日の内容

- カスタムレンダーテクスチャ
  - カスタムレンダーテクスチャの概要
  - ライフゲーム
  - カールノイズ

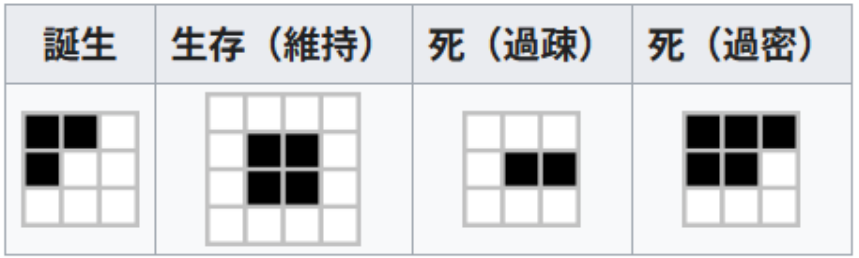


ライフゲーム

- イギリスの数学者ジョン・ホートン・コンウェイが考案した数理モデル（1970）
- セルオートマトンの例
  - 格子状のセルの更新を続ける離散的計算モデル
  - 複雑な動きのアニメーションを生じる



■:生きているセル、□:死んでいるセル



ライフゲームのルール

隣接する8つのセルの状態に応じて各セルを更新

- 誕生
  - 死んでいるセルに隣接する生きたセルがちょうど3つあれば、次の世代が誕生
- 生存
  - 生きているセルに隣接する生きたセルが2つか3つならば、次の世代でも生存
- 過疎
  - 生きているセルに隣接する生きたセルが1つ以下ならば、過疎により死滅
- 過密
  - 生きているセルに隣接する生きたセルが4つ以上ならば、過密により死滅

ライフゲームの更新の疑似コード

生きた隣接セル数	0, 1	2	3	4,5,6,7,8
■:生きている	□	■	■	□
□:死んでいる	□	□	■	□

- ライフゲームのルール
  - 誕生:死んでいるセルに隣接する生きたセルがちょうど3つあれば、次の世代が誕生
  - 生存:生きているセルに隣接する生きたセルが2つか3つならば、次の世代でも生存
  - 過疎:生きているセルに隣接する生きたセルが1つ以下ならば、過疎により死滅
  - 過密:生きているセルに隣接する生きたセルが4つ以上ならば、過密により死滅
- セルの更新は次の式で書ける(生きているセル:0, 死んでいるセル:1)

```
セル←lerp(
    (生きた隣接セル数 <= 1 ? 1 : 0) + (4 <= 生きた隣接セル数 ? 1 : 0),
    (生きた隣接セル数 <= 2 ? 1 : 0) + (4 <= 生きた隣接セル数 ? 1 : 0), 現在のセルの値)
```

「死んだ隣接セル数」の方が計算しやすいので変換

```
= lerp(
    (7 <= 死んだ隣接セル数 ? 1 : 0) + (死んだ隣接セル数 <= 4 ? 1 : 0),
    (6 <= 死んだ隣接セル数 ? 1 : 0) + (死んだ隣接セル数 <= 4 ? 1 : 0), 現在のセルの値)
```

三項演算子をstepに置き換え。浮動小数点数用に中間の0.5で切り替え

```
= lerp(
    step(6.5, 死んだ隣接セル数) + step(死んだ隣接セル数, 4.5),
    step(5.5, 死んだ隣接セル数) + step(死んだ隣接セル数, 4.5), 現在のセルの値)
```

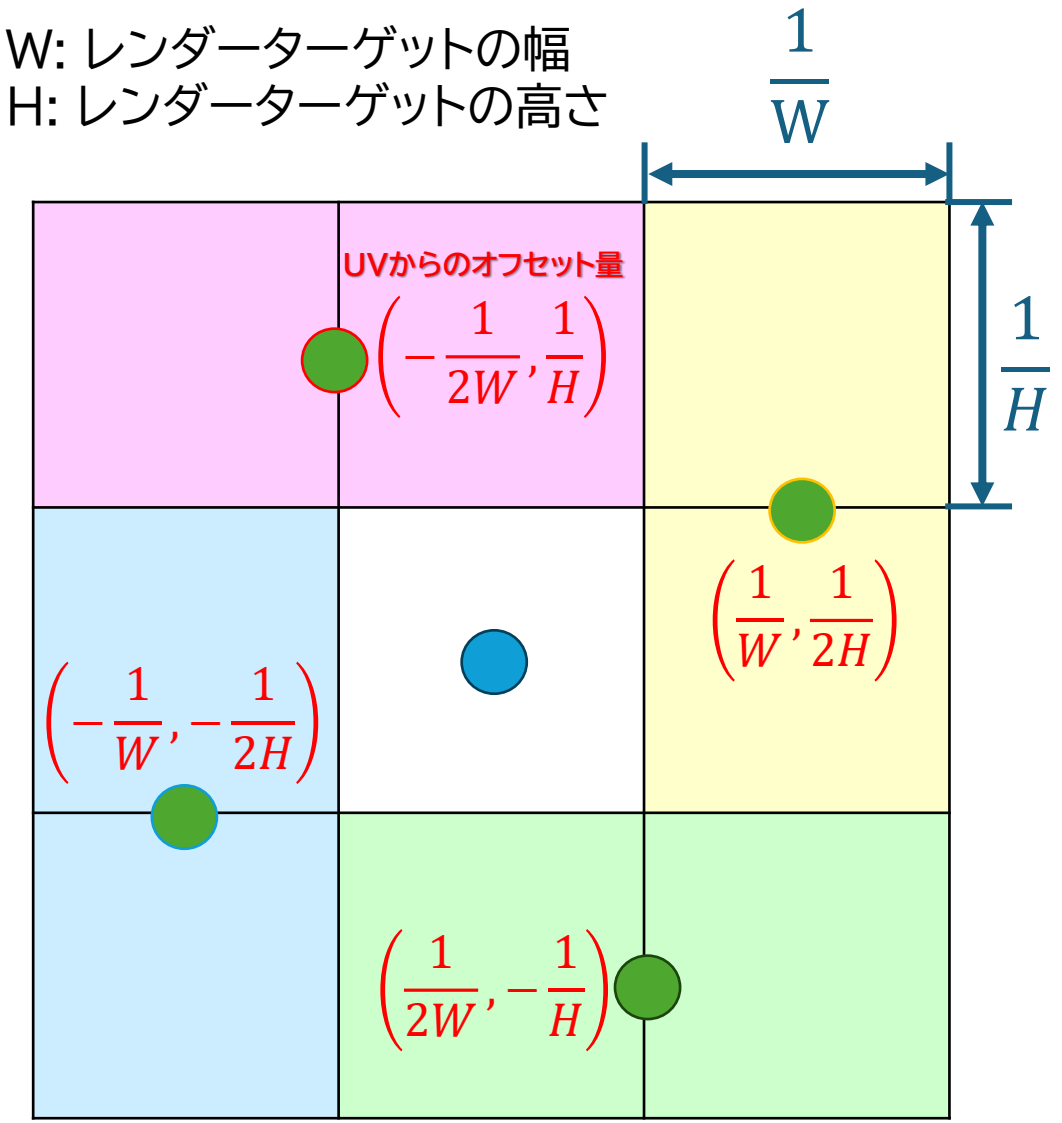
(引数の順番を入れ替えて判定を反転しているのに注意)

死んだ隣接セル数

周囲のテクセルの平均を求める

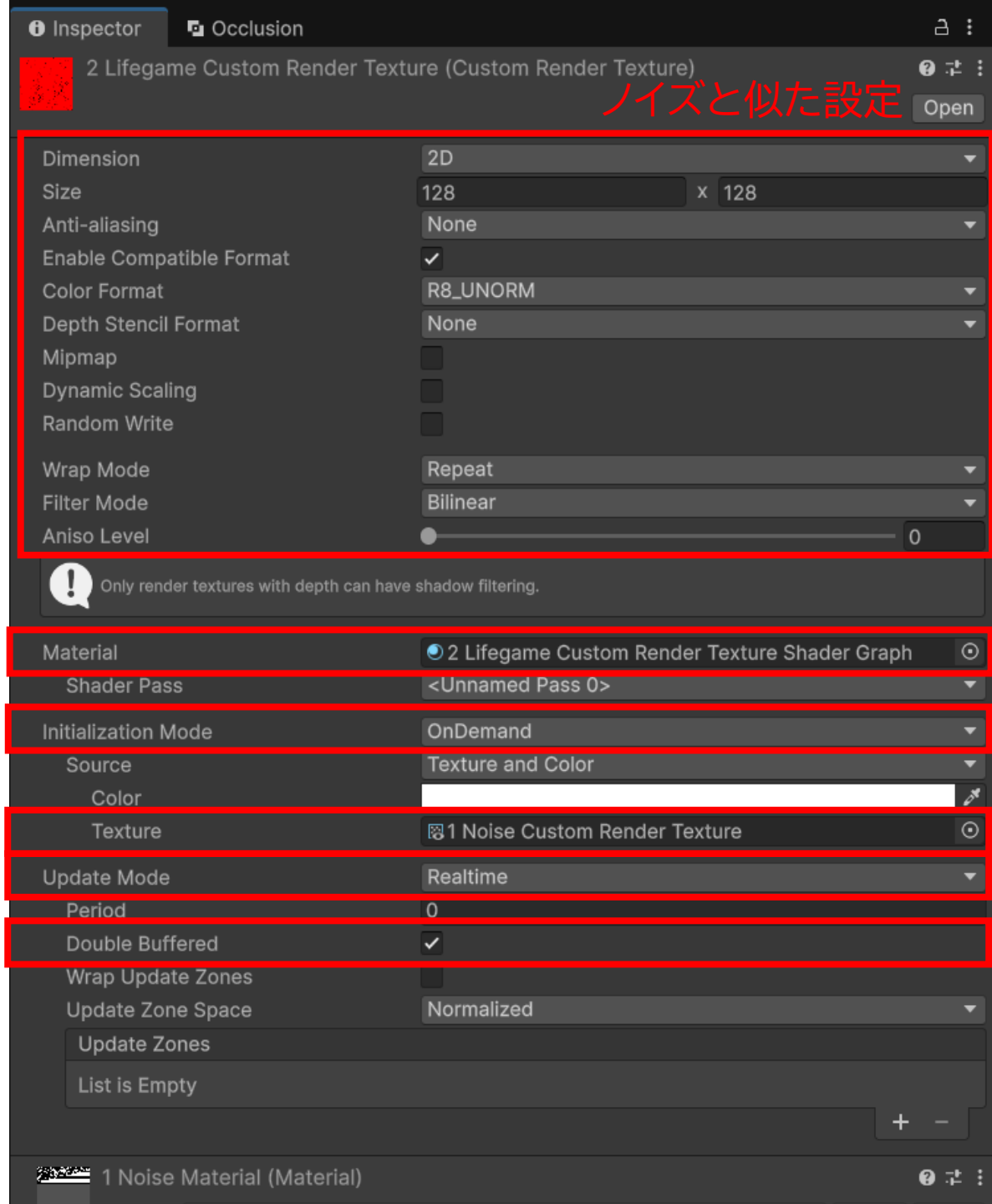
- ４つのサンプリング結果の平均
- サンプリング数を減らすために２つのテクセルの中間の場所をサンプリング
  - （ハードウェア機能の）バイリニアサンプリング機能を使うことで、テクセルを読み込んだだけで、２つのテクセルの平均が得られる

W: レンダーターゲットの幅
H: レンダーターゲットの高さ



やってみよう

- カスタムレンダーテクスチャの追加
- 名称例:「1 Lifegame/2 Lifegame Custom render Texture」



ノイズと似た設定

更新用マテリアル(後述)

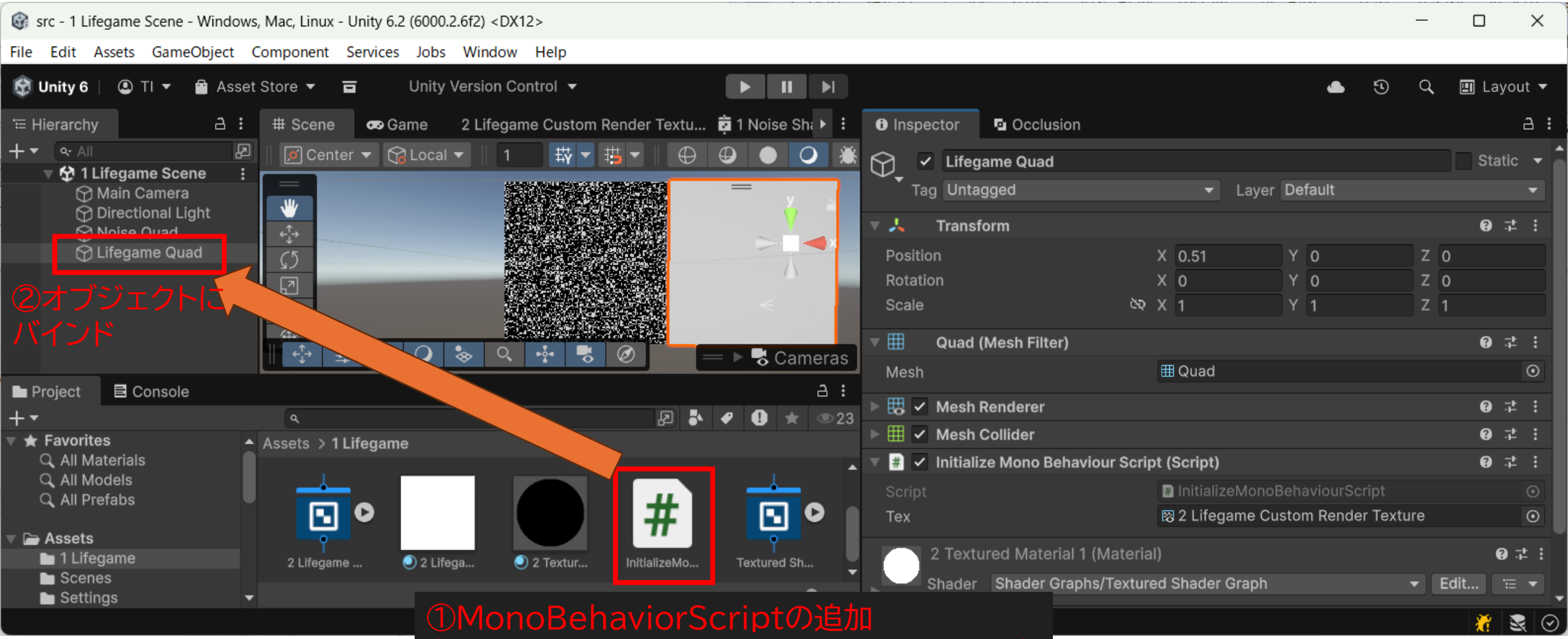
更新タイミングをスクリプトで指定

生成したテクスチャを指定

毎フレ更新

ダブルバッファ(読み書きを切り替えて使用)

更新用のスクリプトのバインド

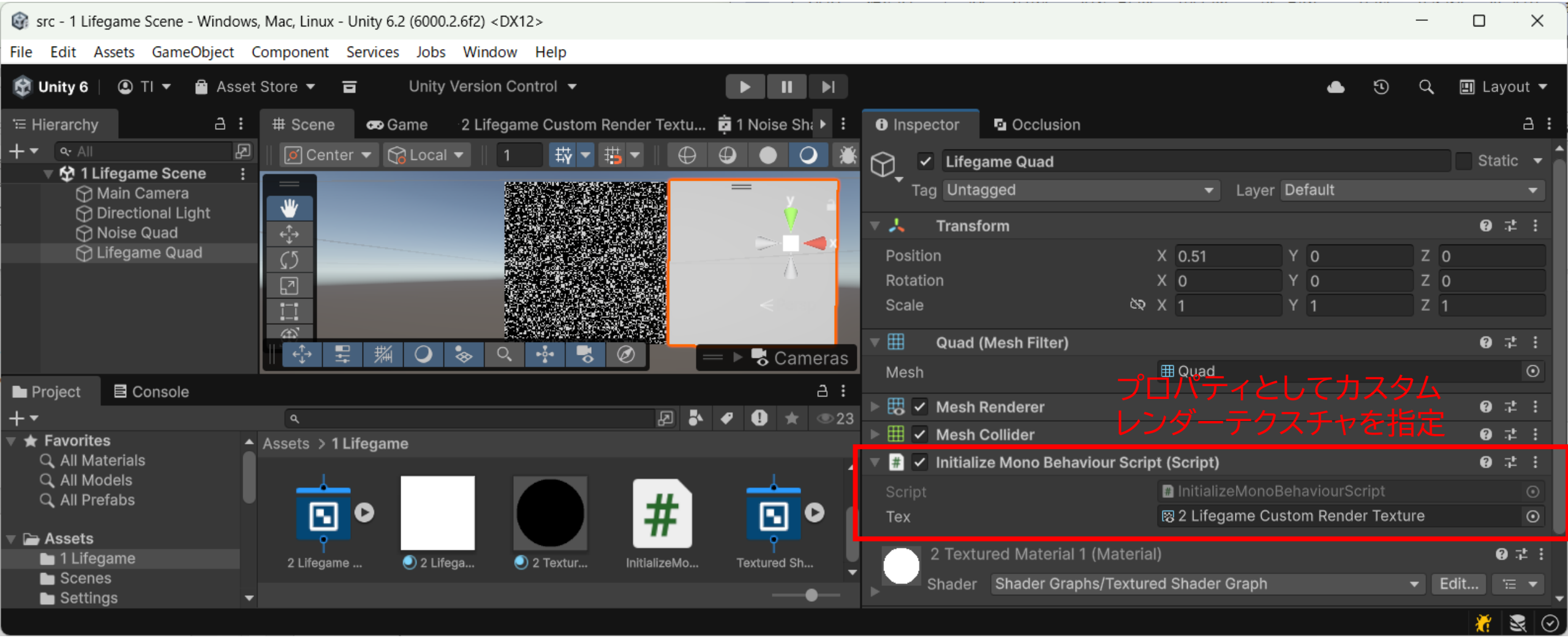


更新用のスクリプトコード

- CustomRender Textureクラス
  - Initializeメソッドで初期化呼び出しが可能
  - Updateメソッドで更新も可能(今回は未使用)

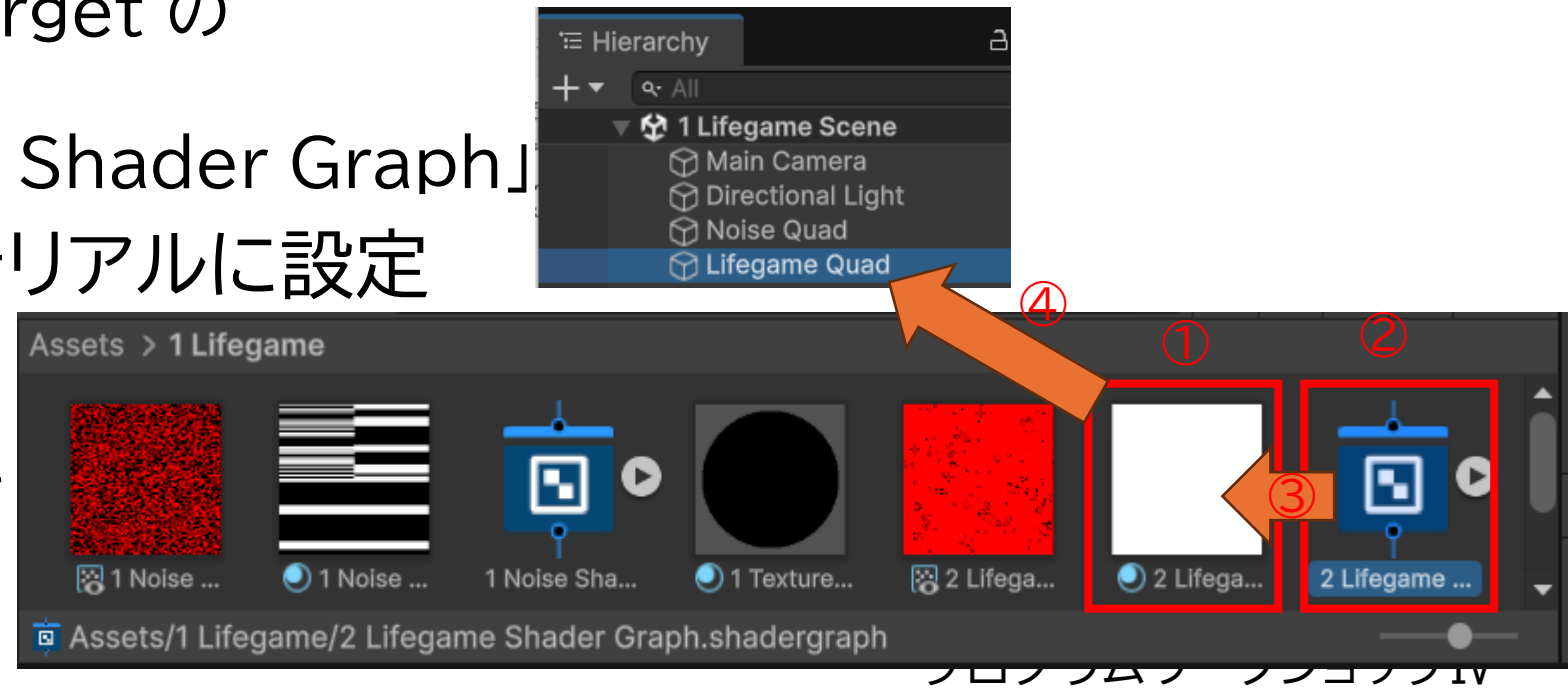
```
1          using UnityEngine;
2
           Unity スクリプト (1 件のアセット参照)|0 個の参照
3          public class InitializeMonoBehaviourScript : MonoBehaviour
4          {
5              [SerializeField] CustomRenderTexture tex = default!;
6
7              // Start is called once before the first execution of U
               Unity メッセージ|0 個の参照
8              void Start()
9              {
10                 tex.Initialize();
11             }
12
13             // Update is called once per frame
               Unity メッセージ|0 個の参照
14             void Update()
15             {
16
17             }
18         }
```

更新用のスクリプトのプロパティ設定

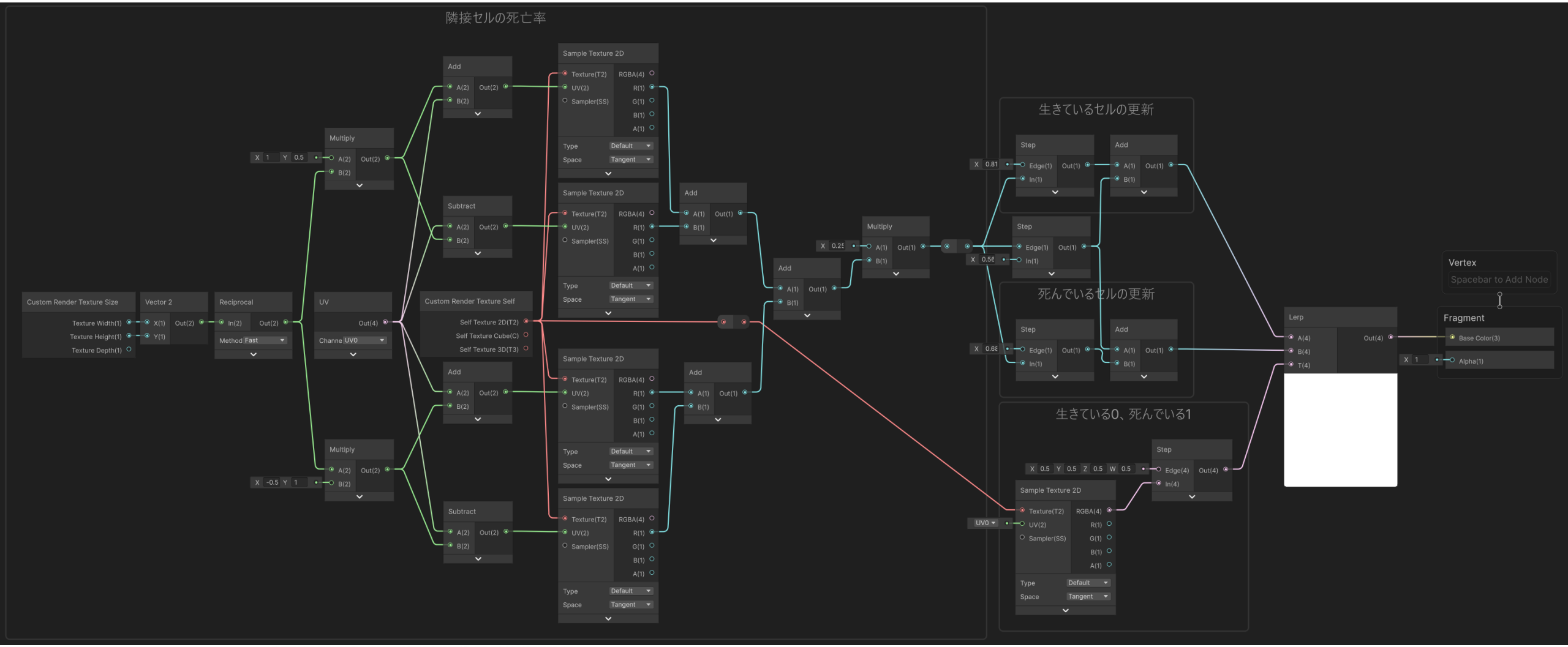


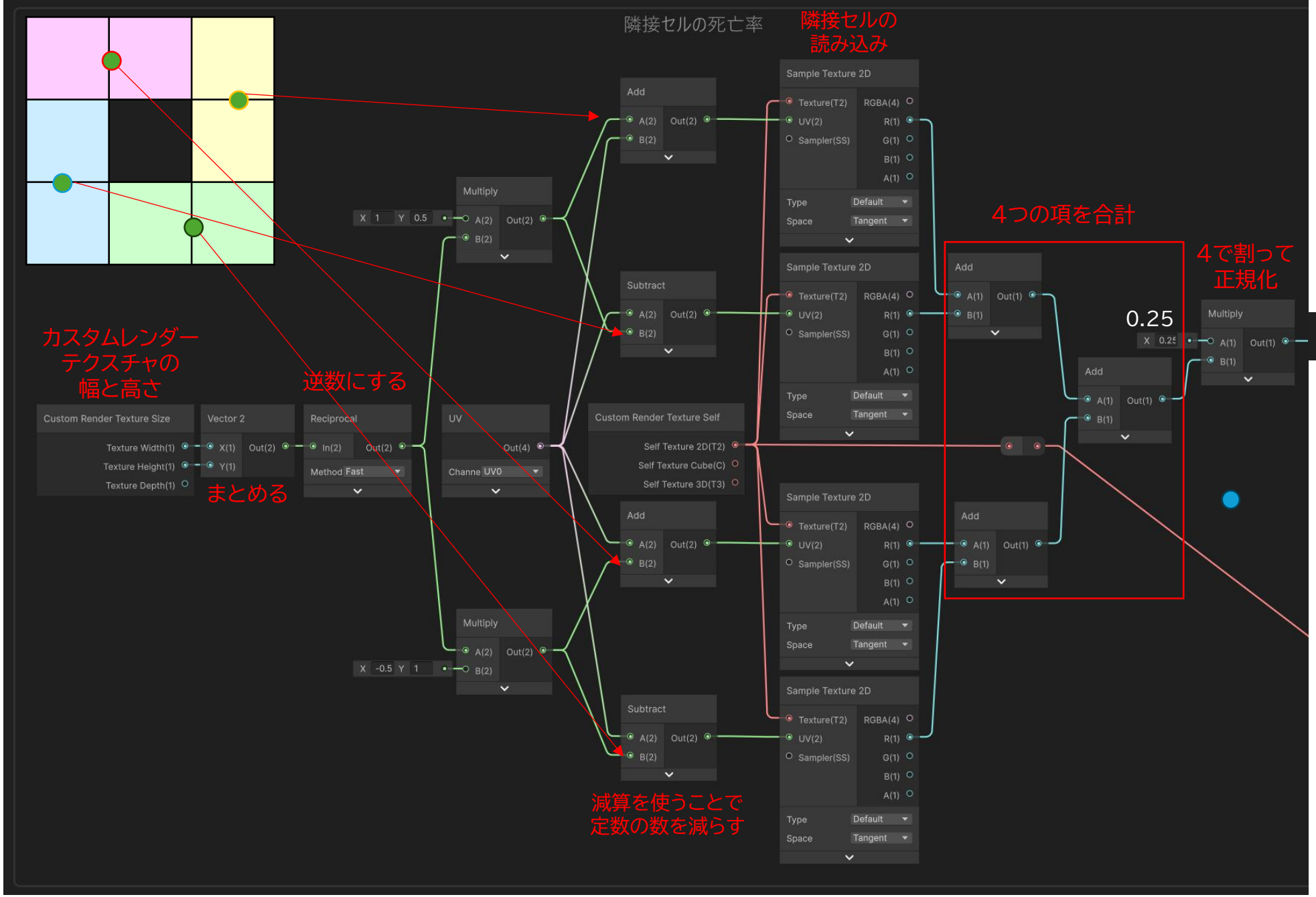
シェーダグラフの生成

1. マテリアルの作成
   - 名称例:「1 Lifegame/2 Lifegame Material」
2. シェーダグラフの作成
   - Custom Render Target の Shader Graph
   - 名称例:「2 Lifegame Shader Graph」
3. シェーダグラフをマテリアルに設定
4. マテリアルをバインド
   - 「Lifegame Quad」に

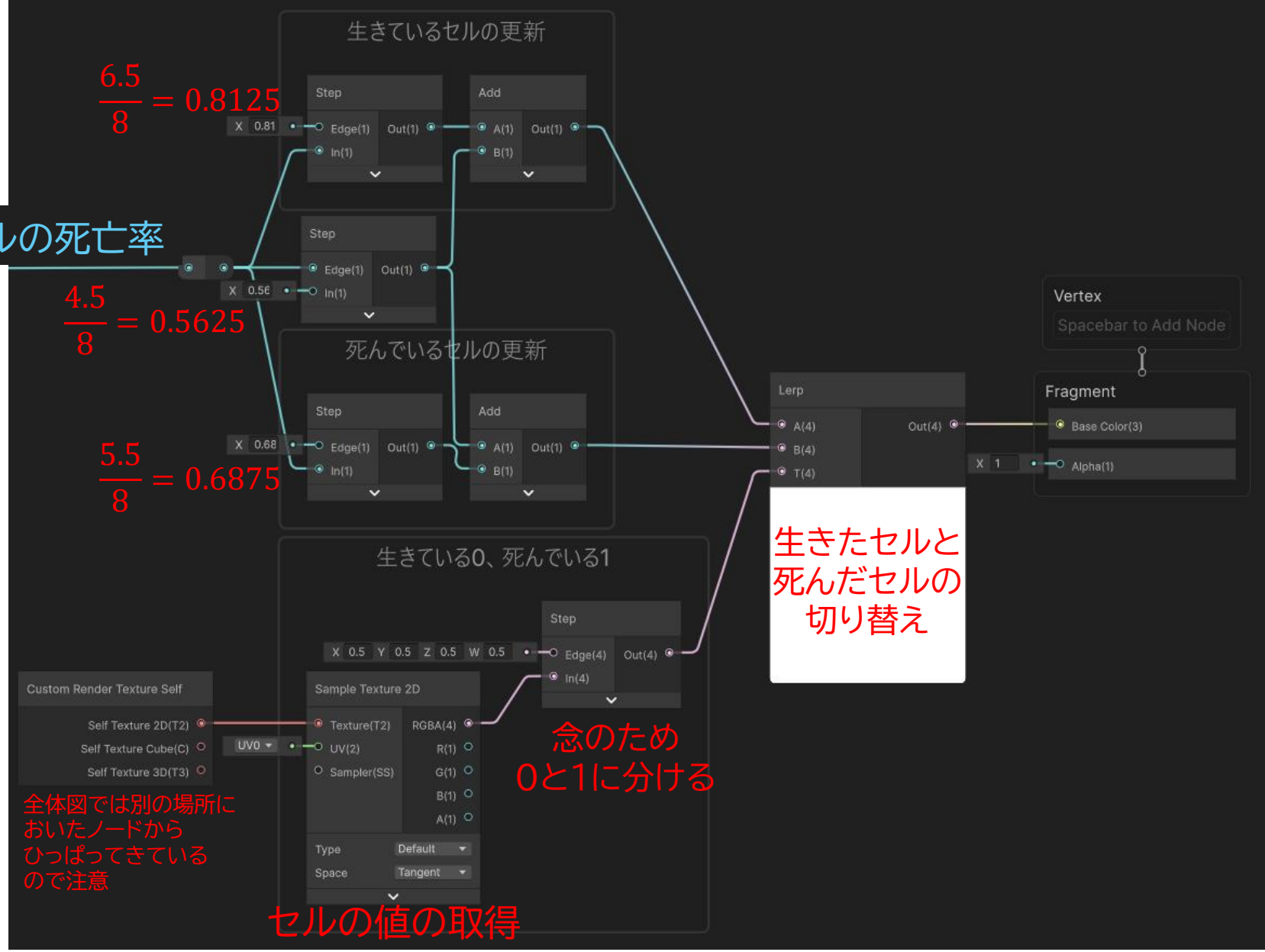


シェーダグラフ


隣接セルの死亡率
Custom Render Texture Size
Texture Width(1)
Texture Height(1)
Texture Depth(1)
Vector 2
X(1)
Y(1)
Out(2)
Reciprocal
In(2)
Out(2)
Method Fast
Multiply
A(2)
B(2)
Out(2)
UV
Out(4)
Channe UV0
Add
Subtract
Custom Render Texture Self
Self Texture 2D(T2)
Self Texture Cube(C)
Self Texture 3D(T3)
Sample Texture 2D
Texture(T2)
UV(2)
Sampler(SS)
RGBA(4)
R(1)
G(1)
B(1)
A(1)
Type Default
Space Tangent
Add
A(1)
B(1)
Out(1)
Multiply
Add
生きているセルの更新
Step
Edge(1)
In(1)
Out(1)
Add
死んでいるセルの更新
Step
Add
生きている0、死んでいる1
Step
Edge(4)
In(4)
Out(4)
Sample Texture 2D
UV0
Lerp
A(4)
B(4)
T(4)
Out(4)
Vertex
Spacebar to Add Node
Fragment
Base Color(3)
Alpha(1)


隣接セルの死亡率
隣接セルの
読み込み
4つの項を合計
4で割って
正規化
隣接セルの死亡率［0,1］
カスタムレンダー
テクスチャの
幅と高さ
逆数にする
まとめる
減算を使うことで
定数の数を減らす
Add
Subtract
Multiply
Sample Texture 2D
Custom Render Texture Size
Vector 2
Reciprocal
UV
Custom Render Texture Self
Texture Width(1)
Texture Height(1)
Texture Depth(1)
X(1)
Y(1)
In(2)
Out(2)
Out(4)
Method Fast
Channe UV0
Self Texture 2D(T2)
Self Texture Cube(C)
Self Texture 3D(T3)
Texture(T2)
UV(2)
Sampler(SS)
RGBA(4)
R(1)
G(1)
B(1)
A(1)
Type Default
Space Tangent
A(2)
B(2)
A(1)
B(1)
Out(1)
X 1 Y 0.5
X -0.5 Y 1
0.25
X 0.25

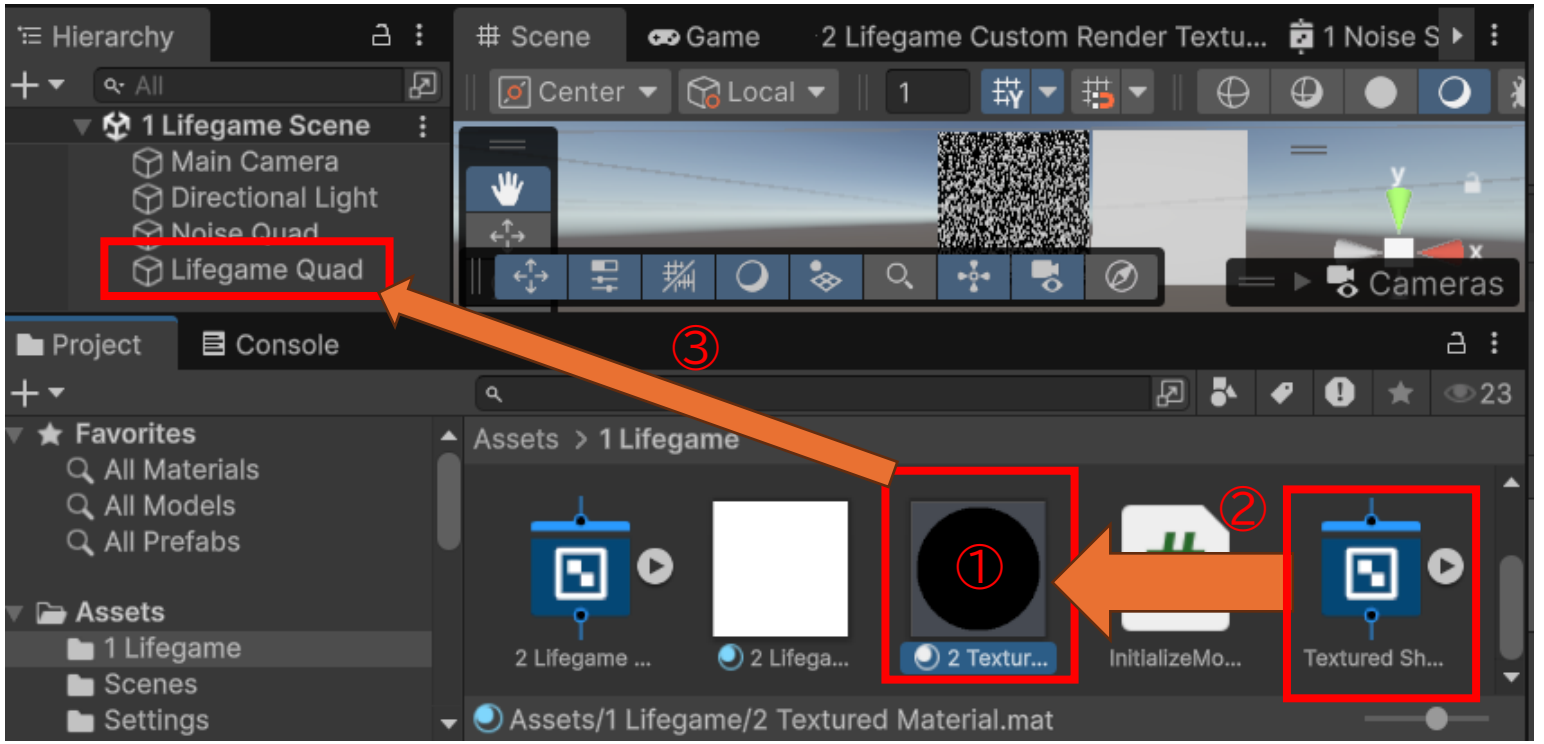
ライフゲームのルール

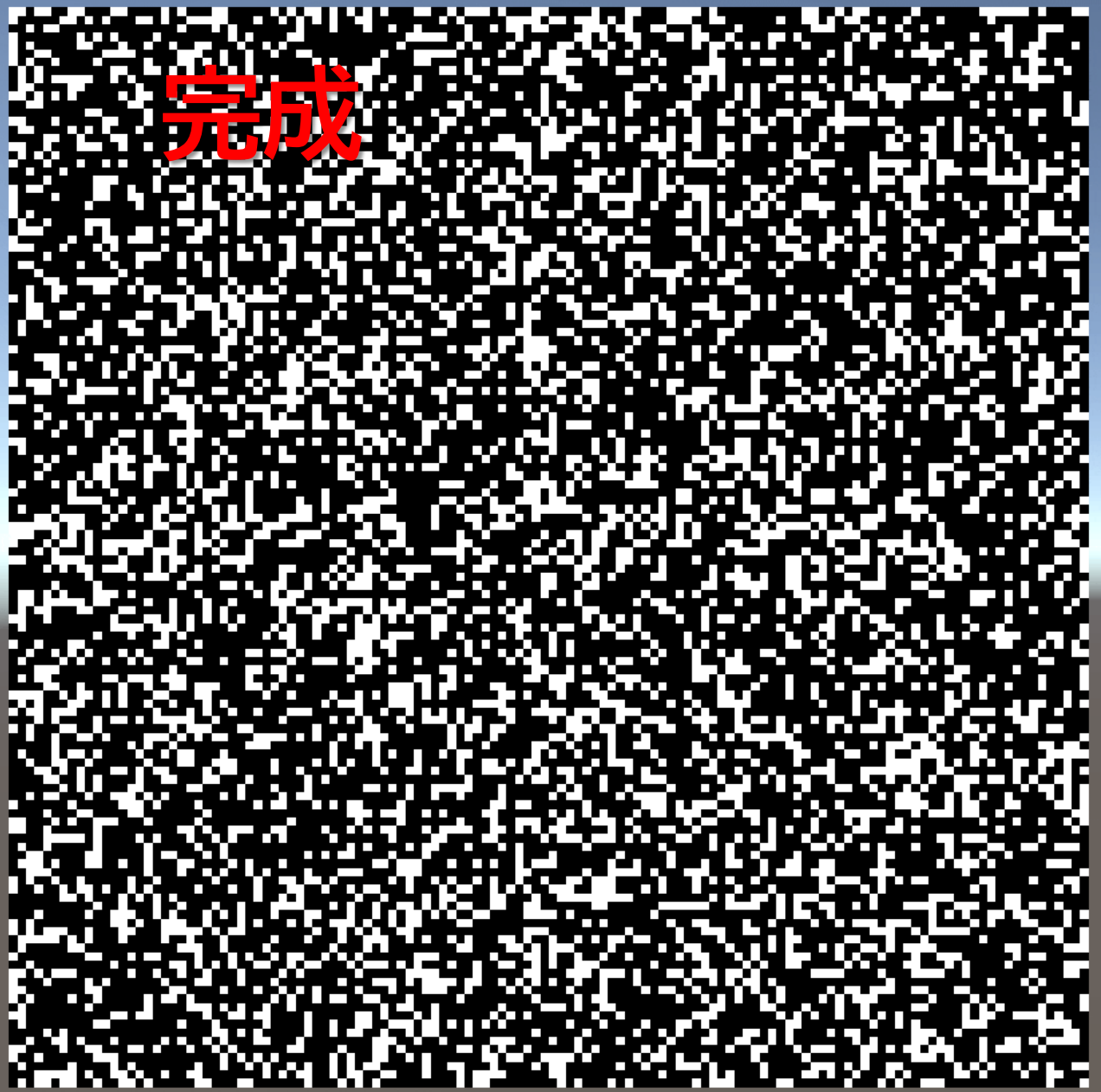


lerp(step(6.5, 死んだ隣接セル数) + step(死んだ隣接セル数, 4.5),
step(5.5, 死んだ隣接セル数) + step(死んだ隣接セル数, 4.5), 現在のセルの値)

可視化

1. マテリアルの作成
   - 名称例:「1 Lifegame/2 Textured Material」
2. シェーダグラフをマテリアルに設定
   - すでに作成した「Textured Shader Graph」のもの
3. マテリアルをバインド
   - 「Lifegame Quad」に



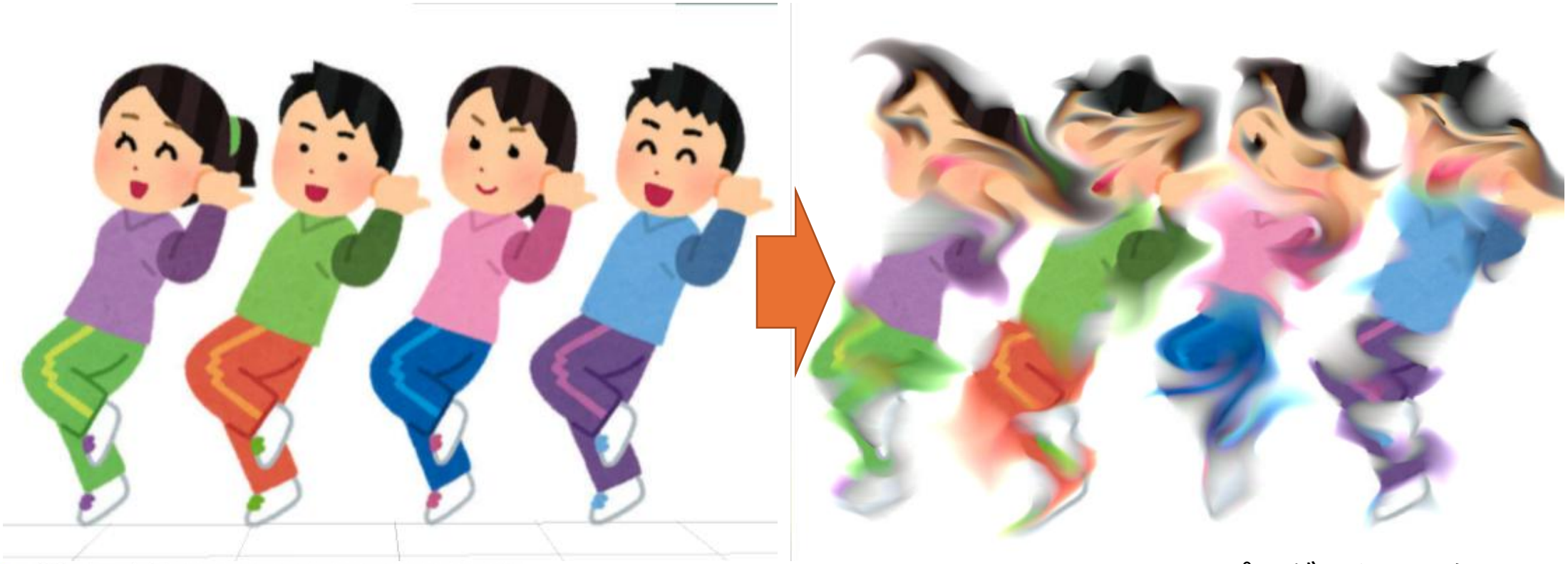

完成

本日の内容

- カスタムレンダーテクスチャ
  - カスタムレンダーテクスチャの概要
  - ライフゲーム
  - カールノイズ
    - ノイズを使って模様を流す
    - インタラクティブに動かす

今回のネタ2

- 画面を流す一つの方法



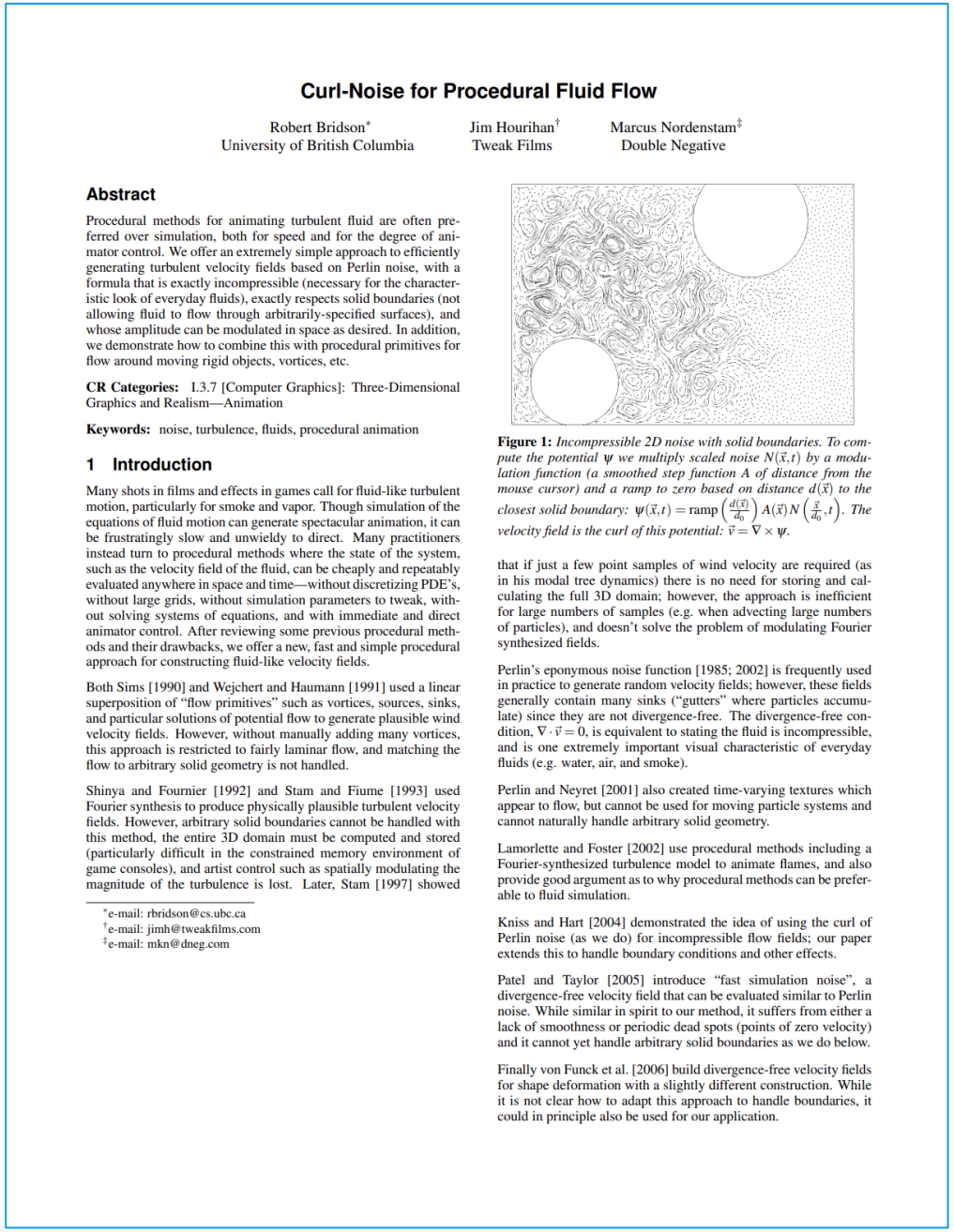
画面を流すには

- テクスチャを読む位置をずらす
  - 場所ごとに少しづつ変えながら

- 流体計算っぽいとかっこよさそう
  - しかし、流体計算は負荷が高い

Curl-Noise

- 流体としての質量を保存するノイズ
- Bridson, R., Houriham, J., & Nordenstam, M. (2007). Curl-noise for procedural fluid flow. ACM Transactions on Graphics (ToG), 26(3), 46-es.
- 日本語での解説例
  - Curl Noise書いてみた Qiita@nyamadandan
    - https://qiita.com/nyamadandan/items/2a8bc7a3639e7b5ce9c9
  - UnityのCompute ShaderでCurl Noiseを実装(流体編) - e.blog
    - https://edom18.hateblo.jp/entry/2018/01/18/081750



Curl-Noise for Procedural Fluid Flow

Robert Bridson* University of British Columbia
Jim Hourihan† Tweak Films
Marcus Nordenstam‡ Double Negative

Abstract

Procedural methods for animating turbulent fluid are often preferred over simulation, both for speed and for the degree of animator control. We offer an extremely simple approach to efficiently generating turbulent velocity fields based on Perlin noise, with a formula that is exactly incompressible (necessary for the characteristic look of everyday fluids), exactly respects solid boundaries (not allowing fluid to flow through arbitrarily-specified surfaces), and whose amplitude can be modulated in space as desired. In addition, we demonstrate how to combine this with procedural primitives for flow around moving rigid objects, vortices, etc.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation

Keywords: noise, turbulence, fluids, procedural animation

1 Introduction

Many shots in films and effects in games call for fluid-like turbulent motion, particularly for smoke and vapor. Though simulation of the equations of fluid motion can generate spectacular animation, it can be frustratingly slow and unwieldy to direct. Many practitioners instead turn to procedural methods where the state of the system, such as the velocity field of the fluid, can be cheaply and repeatably evaluated anywhere in space and time—without discretizing PDE's, without large grids, without simulation parameters to tweak, without solving systems of equations, and with immediate and direct animator control. After reviewing some previous procedural methods and their drawbacks, we offer a new, fast and simple procedural approach for constructing fluid-like velocity fields.

Both Sims [1990] and Wejchert and Haumann [1991] used a linear superposition of "flow primitives" such as vortices, sources, sinks, and particular solutions of potential flow to generate plausible wind velocity fields. However, without manually adding many vortices, this approach is restricted to fairly laminar flow, and matching the flow to arbitrary solid geometry is not handled.

Shinya and Fournier [1992] and Stam and Fiume [1993] used Fourier synthesis to produce physically plausible turbulent velocity fields. However, arbitrary solid boundaries cannot be handled with this method, the entire 3D domain must be computed and stored (particularly difficult in the constrained memory environment of game consoles), and artist control such as spatially modulating the magnitude of the turbulence is lost. Later, Stam [1997] showed

*e-mail: rbridson@cs.ubc.ca
†e-mail: jimh@tweakfilms.com
‡e-mail: mkn@dneg.com

Figure 1: *Incompressible 2D noise with solid boundaries. To compute the potential ψ we multiply scaled noise $N(\vec{x},t)$ by a modulation function (a smoothed step function A of distance from the mouse cursor) and a ramp to zero based on distance $d(\vec{x})$ to the closest solid boundary: $\psi(\vec{x},t) = \text{ramp}\left(\frac{d(\vec{x})}{d_0}\right) A(\vec{x}) N\left(\frac{\vec{x}}{d_0},t\right)$. The velocity field is the curl of this potential: $\vec{v} = \nabla \times \psi$.*

that if just a few point samples of wind velocity are required (as in his modal tree dynamics) there is no need for storing and calculating the full 3D domain; however, the approach is inefficient for large numbers of samples (e.g. when advecting large numbers of particles), and doesn't solve the problem of modulating Fourier synthesized fields.

Perlin's eponymous noise function [1985; 2002] is frequently used in practice to generate random velocity fields; however, these fields generally contain many sinks ("gutters" where particles accumulate) since they are not divergence-free. The divergence-free condition, $\nabla \cdot \vec{v} = 0$, is equivalent to stating the fluid is incompressible, and is one extremely important visual characteristic of everyday fluids (e.g. water, air, and smoke).

Perlin and Neyret [2001] also created time-varying textures which appear to flow, but cannot be used for moving particle systems and cannot naturally handle arbitrary solid geometry.

Lamorlette and Foster [2002] use procedural methods including a Fourier-synthesized turbulence model to animate flames, and also provide good argument as to why procedural methods can be preferable to fluid simulation.

Kniss and Hart [2004] demonstrated the idea of using the curl of Perlin noise (as we do) for incompressible flow fields; our paper extends this to handle boundary conditions and other effects.

Patel and Taylor [2005] introduce "fast simulation noise", a divergence-free velocity field that can be evaluated similar to Perlin noise. While similar in spirit to our method, it suffers from either a lack of smoothness or periodic dead spots (points of zero velocity) and it cannot yet handle arbitrary solid boundaries as we do below.

Finally von Funck et al. [2006] build divergence-free velocity fields for shape deformation with a slightly different construction. While it is not clear how to adapt this approach to handle boundaries, it could in principle also be used for our application.

Curl-Noise

- 流体と同じような特性を持つ場をプロシージャルに生成

- 非圧縮性流体の特徴

$$\nabla \vec{v} = 0$$

$$\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} = 0$$

流れは何もないところから湧き出さない

- 上記の場を強制的に作り出す

$$\vec{v} = \nabla \times \vec{\varphi}$$

なぜなら $\nabla \cdot \boxed{\nabla \times} \vec{\varphi} = 0$

∇に直交する

Curl noiseの作り方(3次元)

- 3次元の場$\vec{\varphi}$を定義する
- 回転($\nabla \times \vec{\varphi}$)計算をして、速度場を作る

$$\vec{v} = \nabla \times \vec{\varphi} = \left(\frac{\partial \varphi_z}{\partial y} - \frac{\partial \varphi_y}{\partial z}, \frac{\partial \varphi_x}{\partial z} - \frac{\partial \varphi_z}{\partial x}, \frac{\partial \varphi_y}{\partial x} - \frac{\partial \varphi_x}{\partial y} \right)$$

- 速度に応じて、テクスチャ座標をずらしてテクスチャを読み込む

2次元のcurl noise

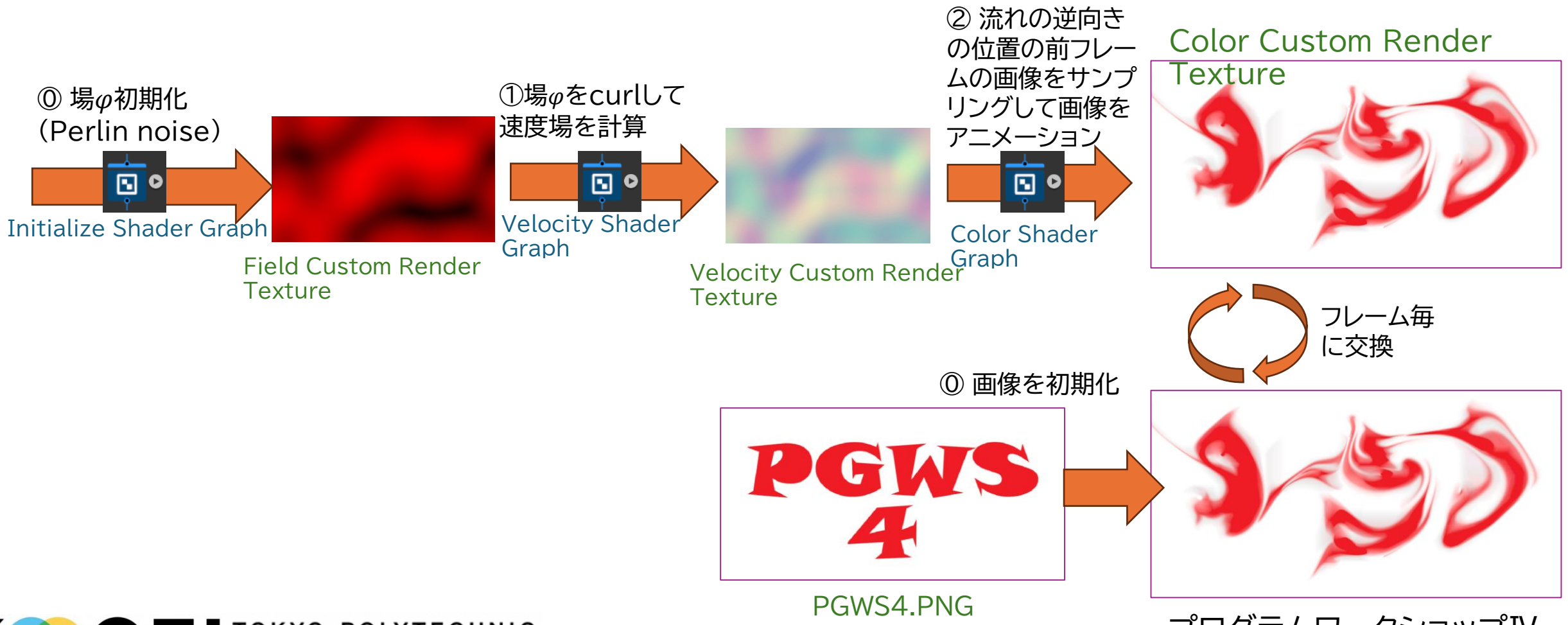
- 場は1次元

$$\varphi = \varphi(x, y)$$

- Curlは2次元平面での回転
  - 3次元のφ_zに対する計算

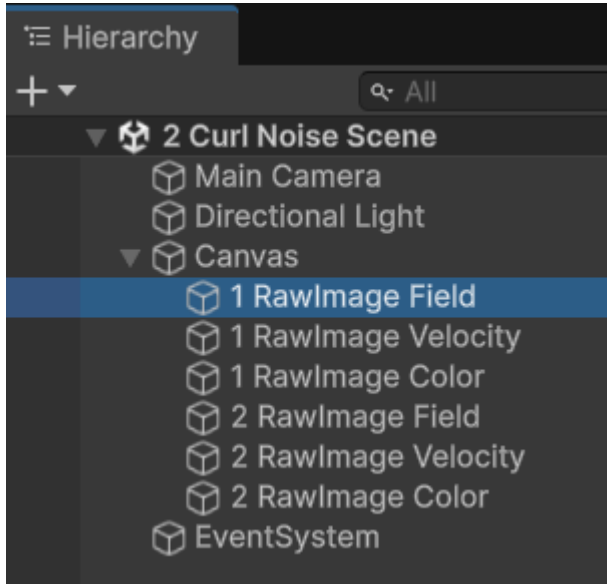
$$\vec{v} = \nabla \times \varphi = \left(\frac{\partial \varphi}{\partial y}, -\frac{\partial \varphi}{\partial x} \right)$$

実装手順


⓪ 場φ初期化
(Perlin noise)
Initialize Shader Graph
Field Custom Render Texture
①場φをcurlして速度場を計算
Velocity Shader Graph
Velocity Custom Render Texture
② 流れの逆向きの位置の前フレームの画像をサンプリングして画像をアニメーション
Color Shader Graph
Color Custom Render Texture
フレーム毎に交換
⓪ 画像を初期化
PGWS4.PNG
プログラムワークショップⅣ

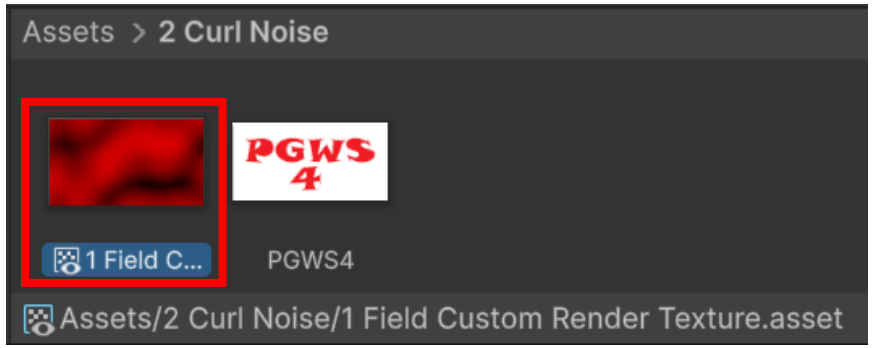
ステップ1：ノイズ（流れの元）

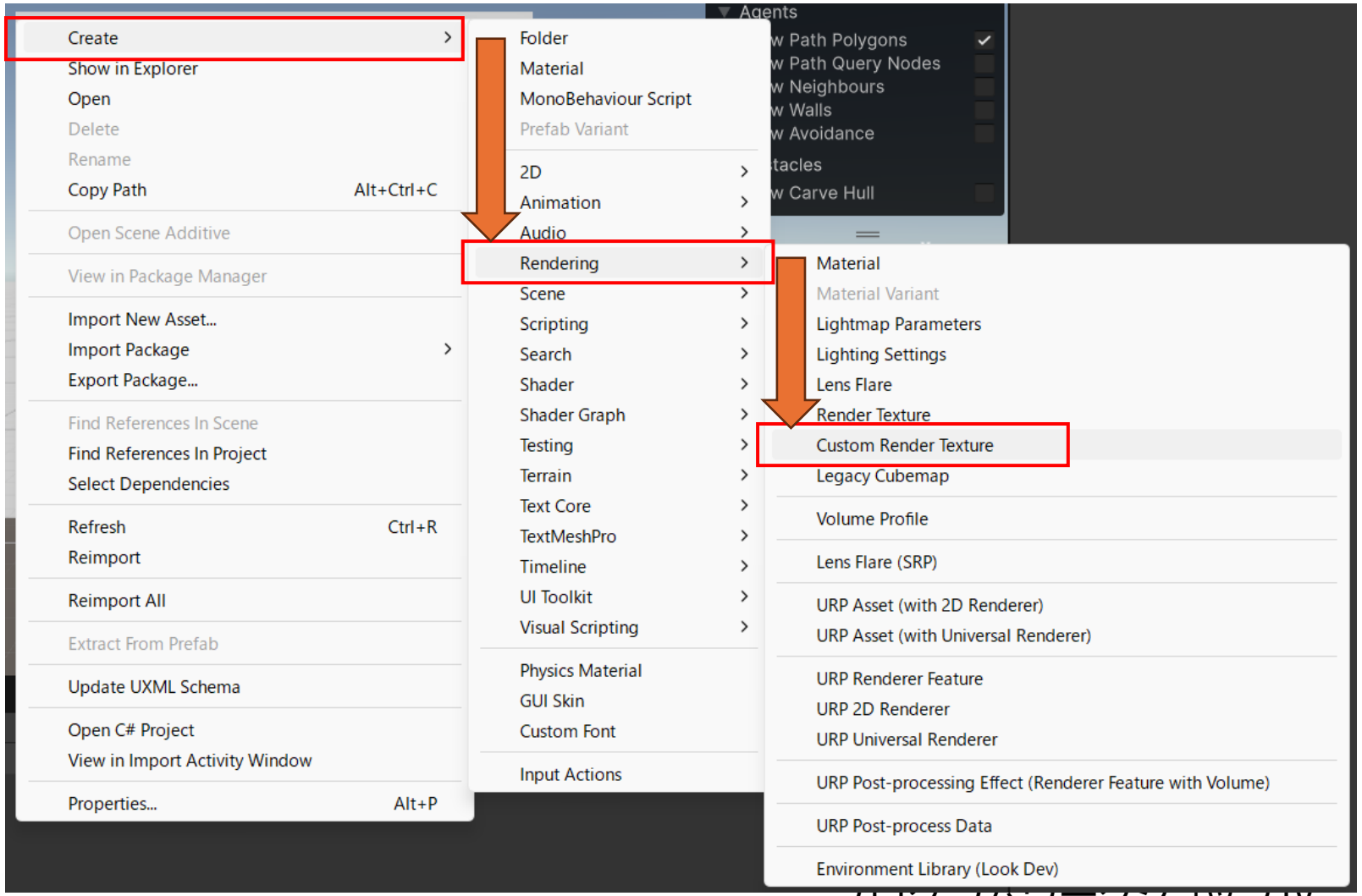
- 赤成分にPerlinノイズを当てはめる
  - 「1 RawImage Field」で可視化する
    - 「UI/Raw Image」オブジェクト



Custom Render Textureの追加

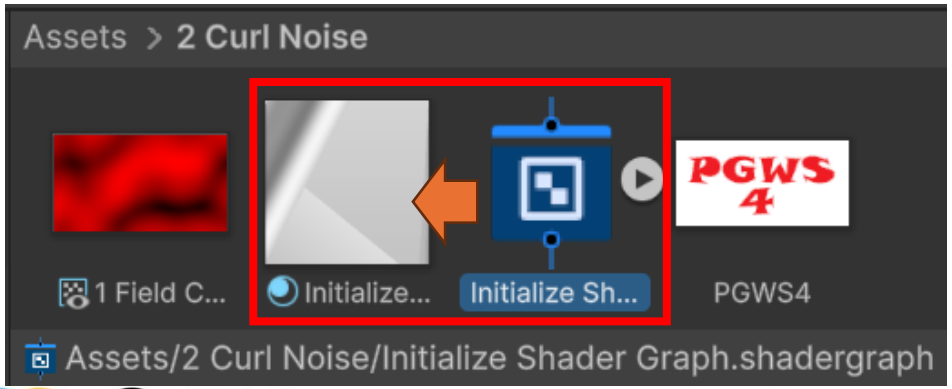
- 場所: Assets/2 Curl Noise
- 名称例:1 Field Custom Render Texture

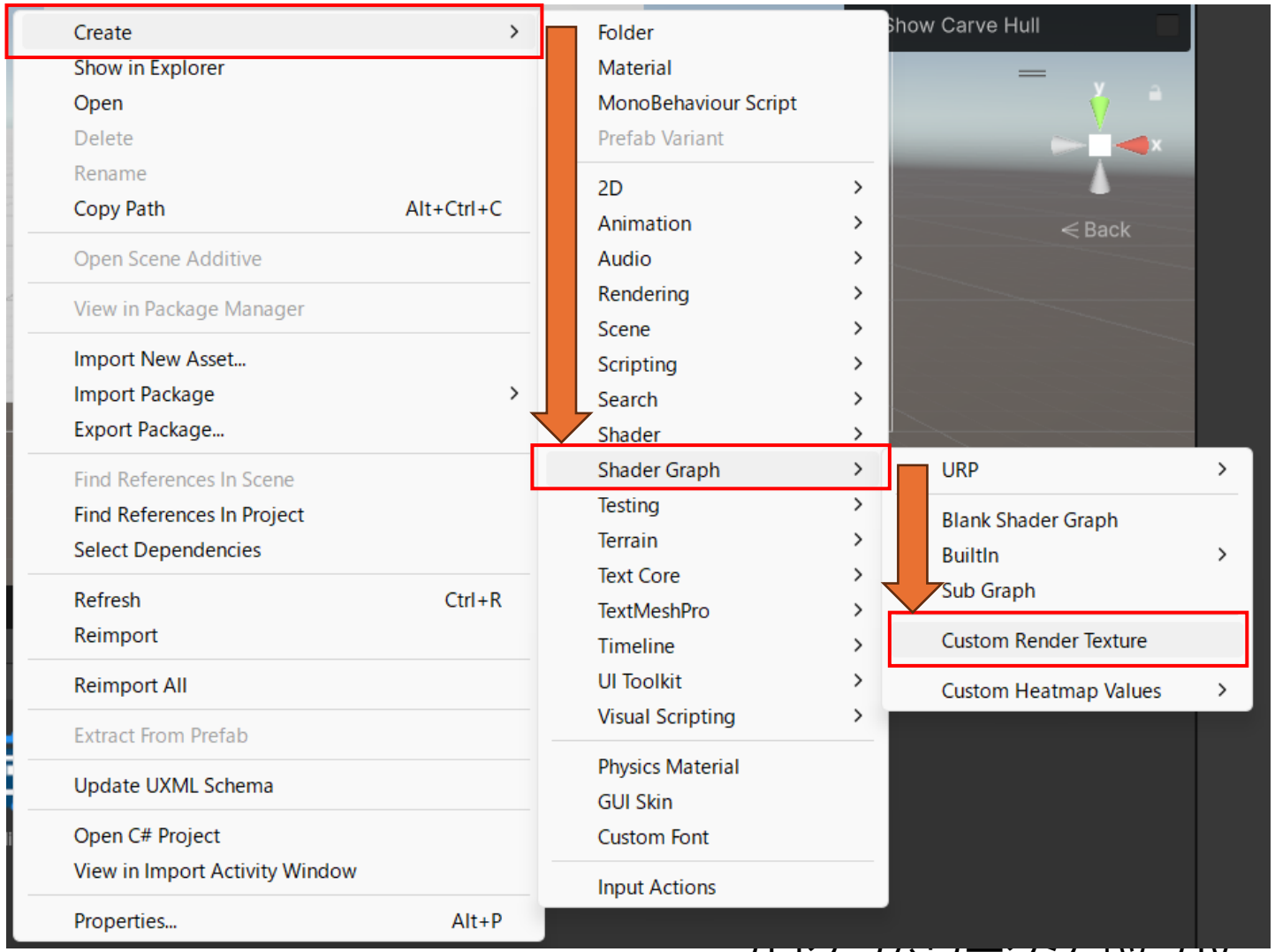




シェーダグラフの追加

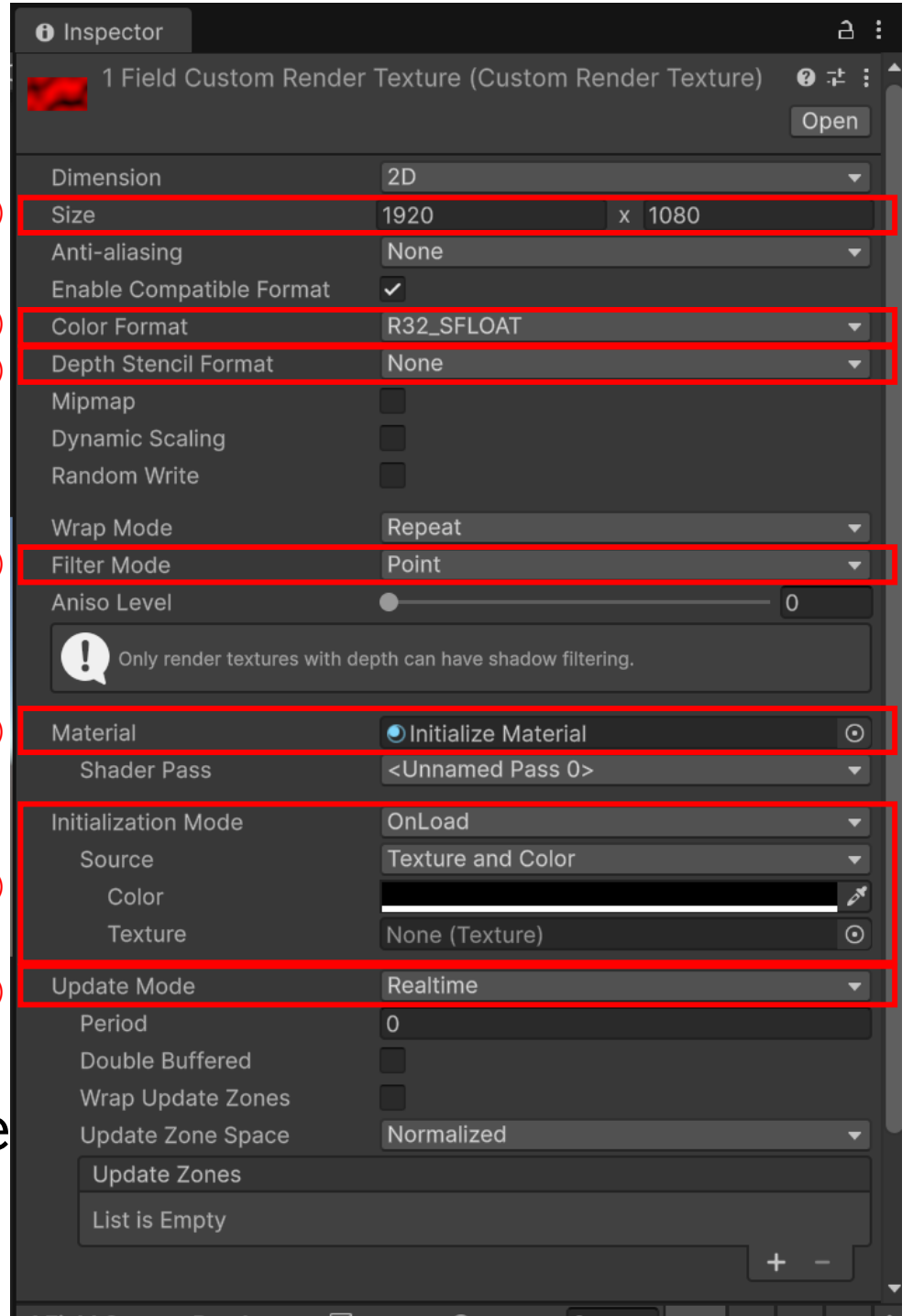
- 専用のアセット(Custom Render Texture)
  - 名称例: Initialize Shader Graph
- マテリアルも追加して設定
  - 名前例: Initialize Material





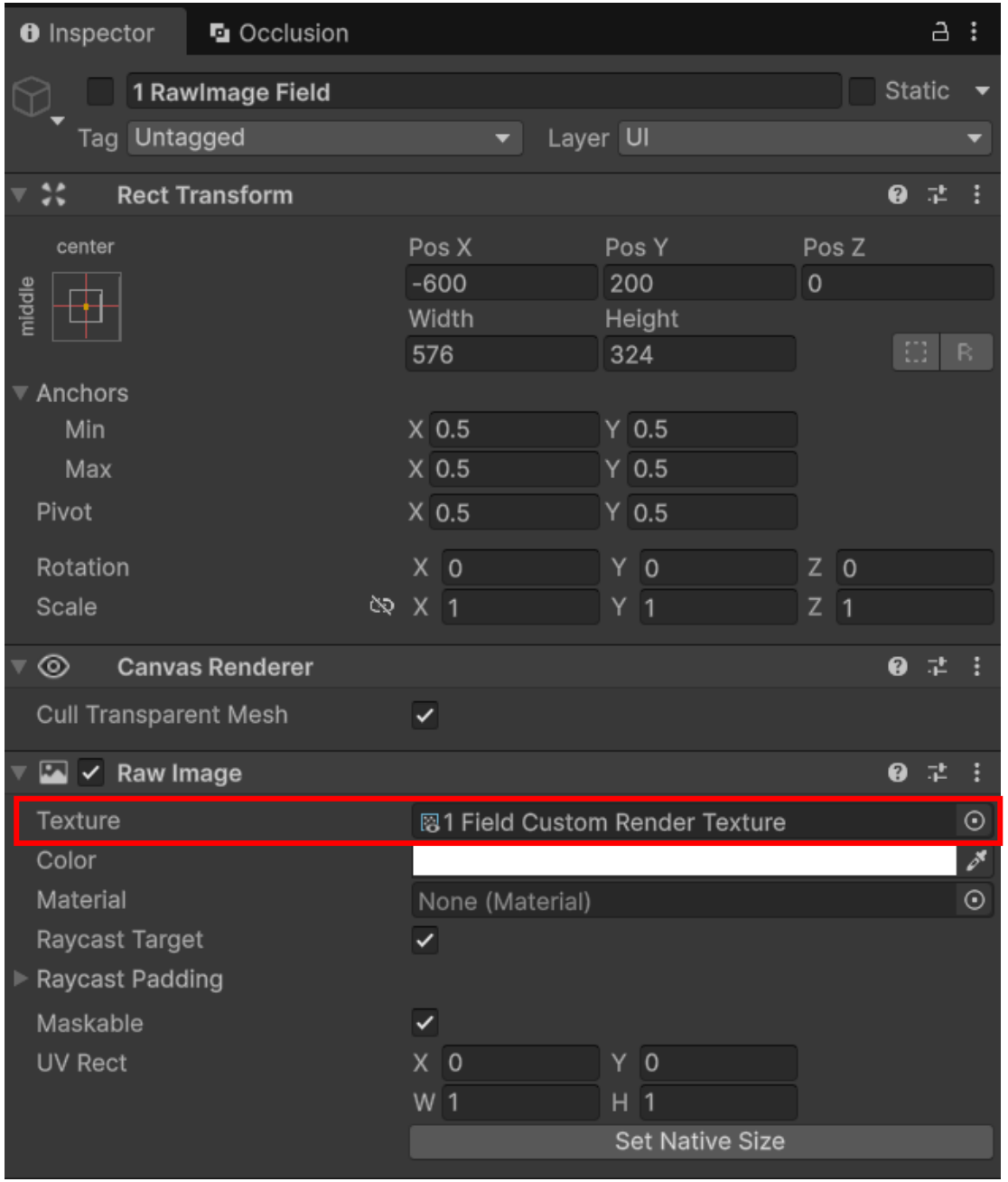
Custom Render Textureの設定

1. サイズ：フルHD(1920×1080)
   - アスペクト比が同じであれば、好みで良い
   - 小さいほど処理が軽いが、粗くなる
     - 時間があれば変更してみよう
2. フォーマット：R32_SFLOAT
   - 精度を高めるために浮動小数点数
     3. サンプラーは「Point」しか使えない(HW的に)
4. 深度バッファは不要
5. 更新用のマテリアルを設定
   6. Inspectorでのパラメータ調整を反映できるようにするために「Realtime」更新
7. 今回は初期化は適当で良い
   - Updateで更新しないでinitialization Modeで計算する方が実行時の負荷は低い



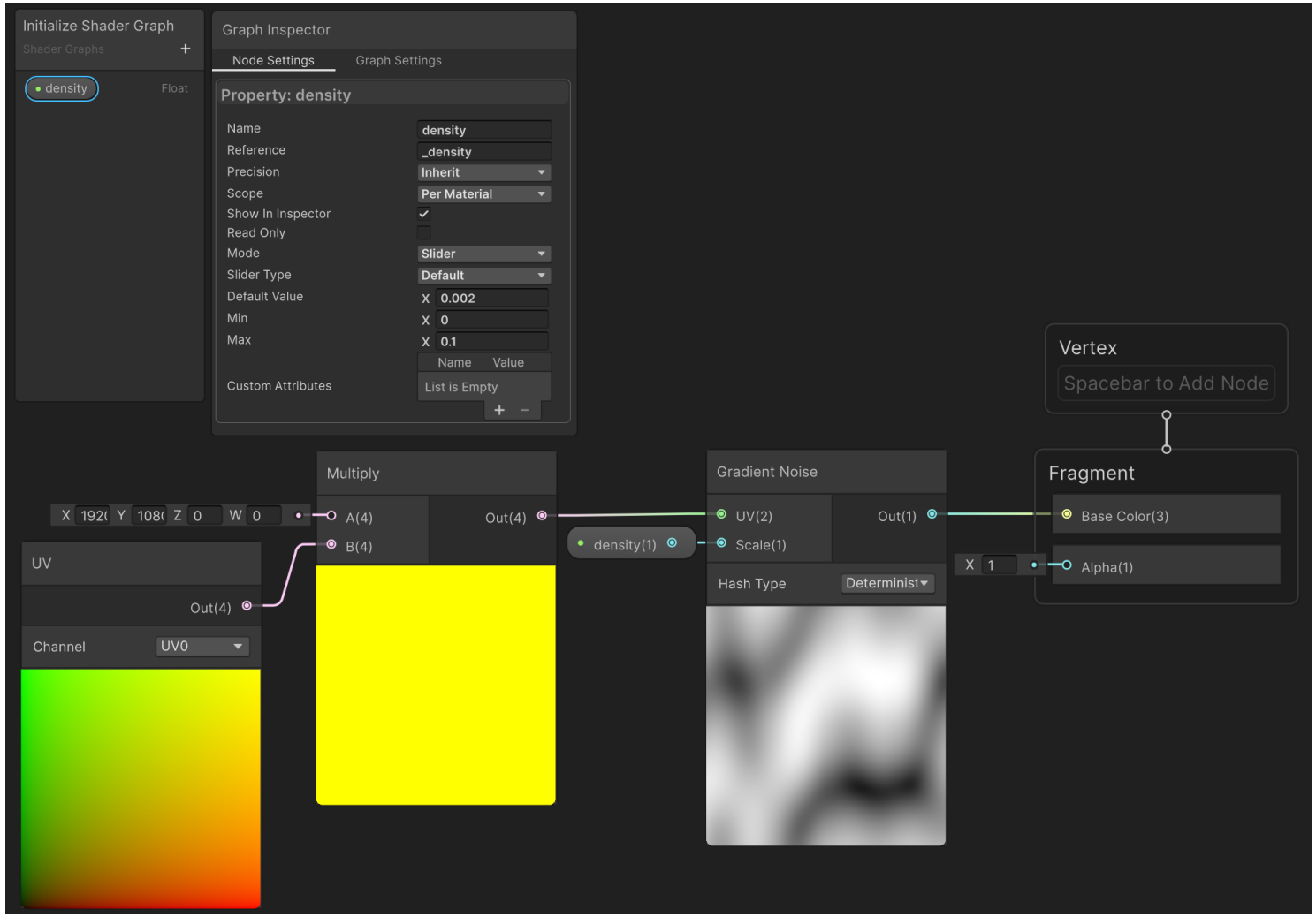
表示の設定

- オブジェクトのテクスチャに
  カスタムレンダーテクスチャを設定



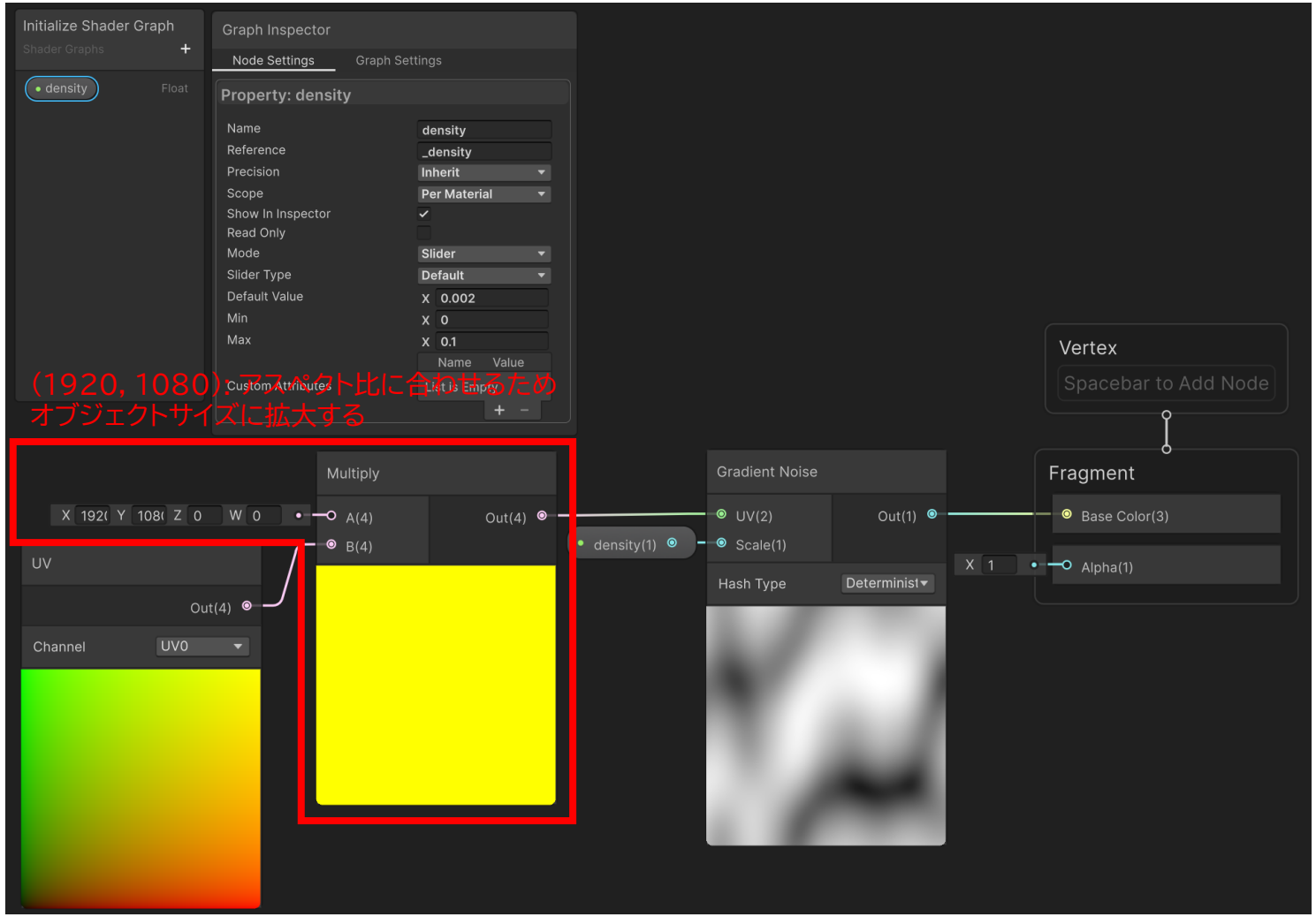
シェーダグラフ

Initialize Shader Graph



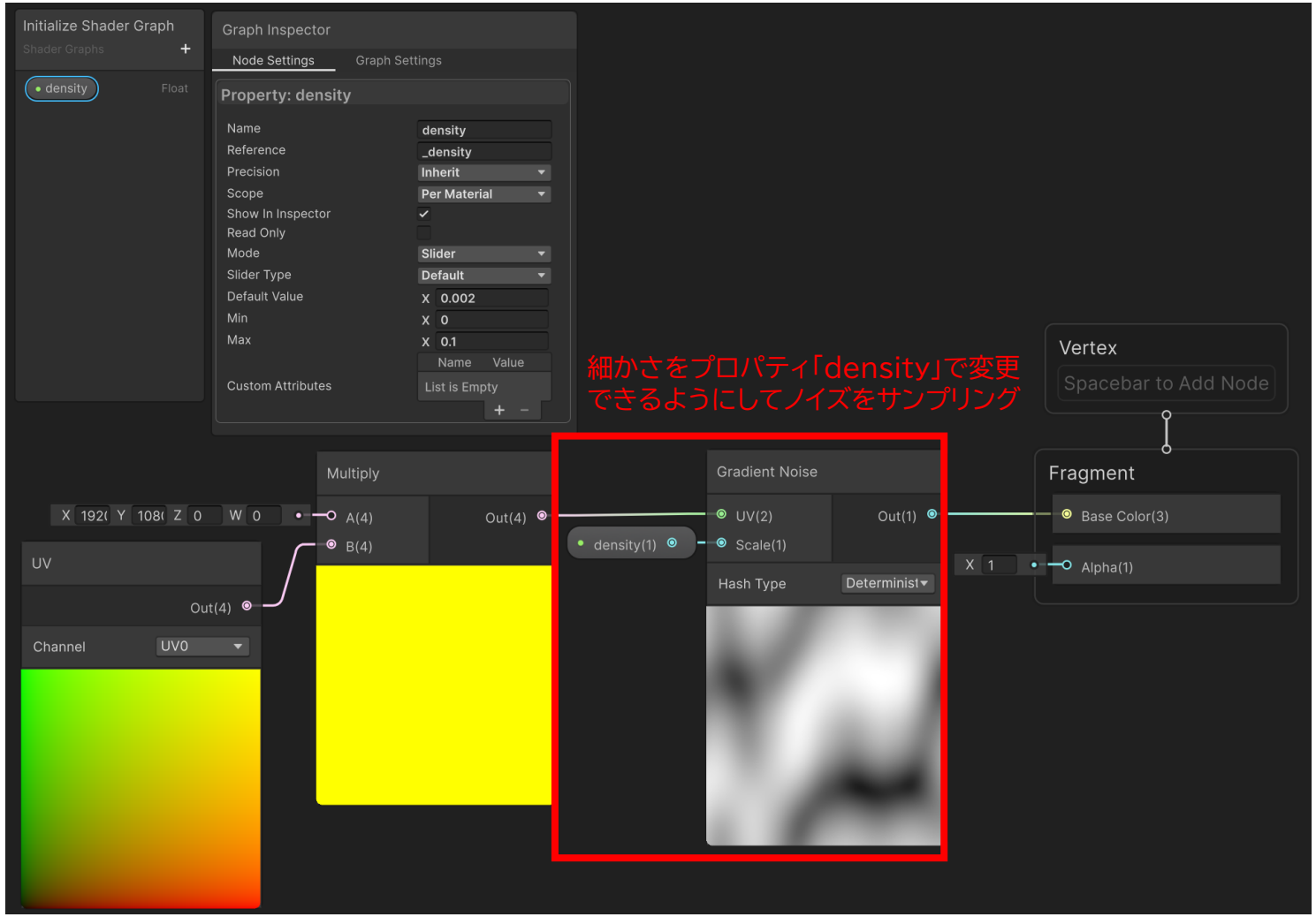
シェーダグラフ

Initialize Shader Graph



シェーダグラフ

Initialize Shader Graph

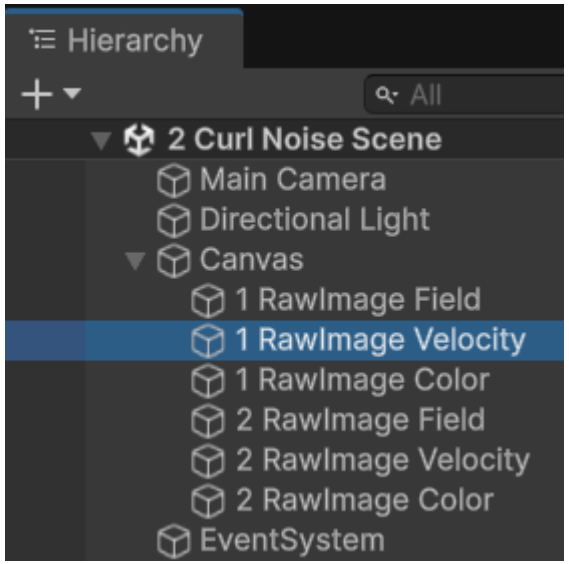


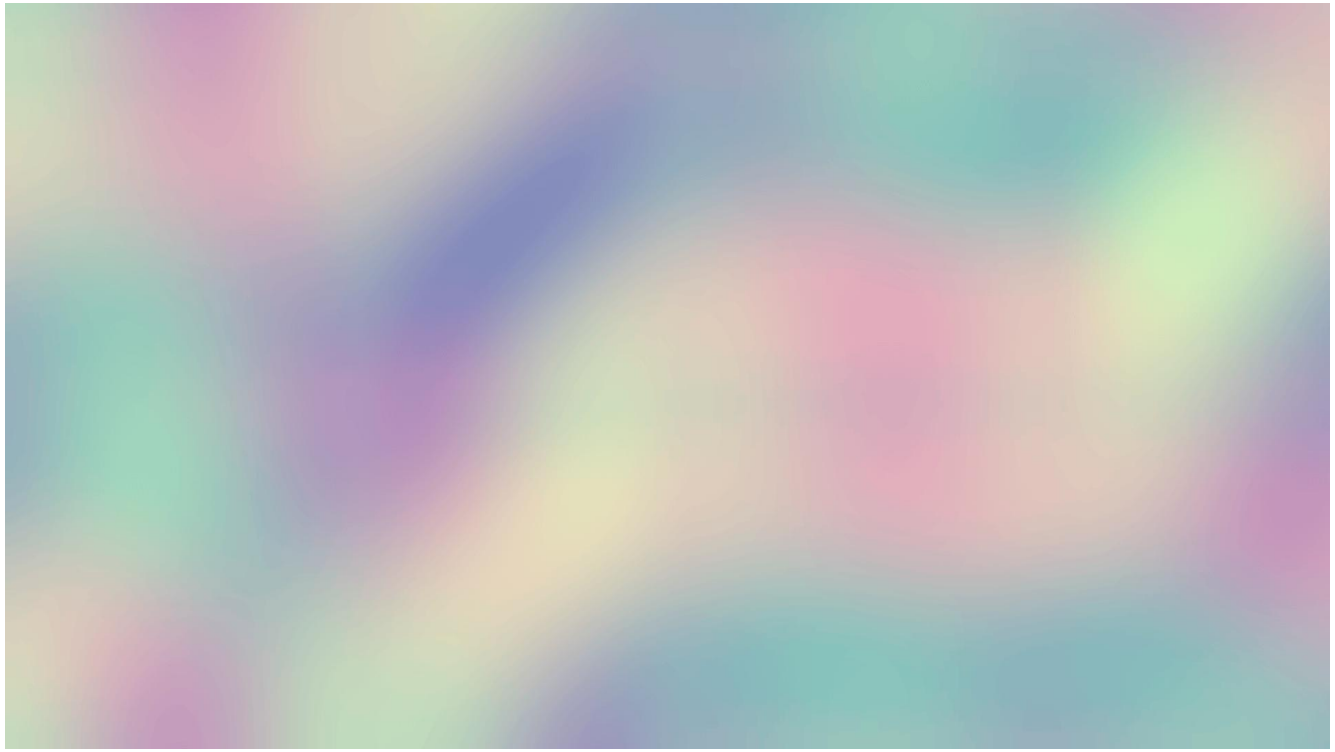
やってみよう
ステップ1:ノイズ(流れの元)

- Inspectorで値を変更してみよう

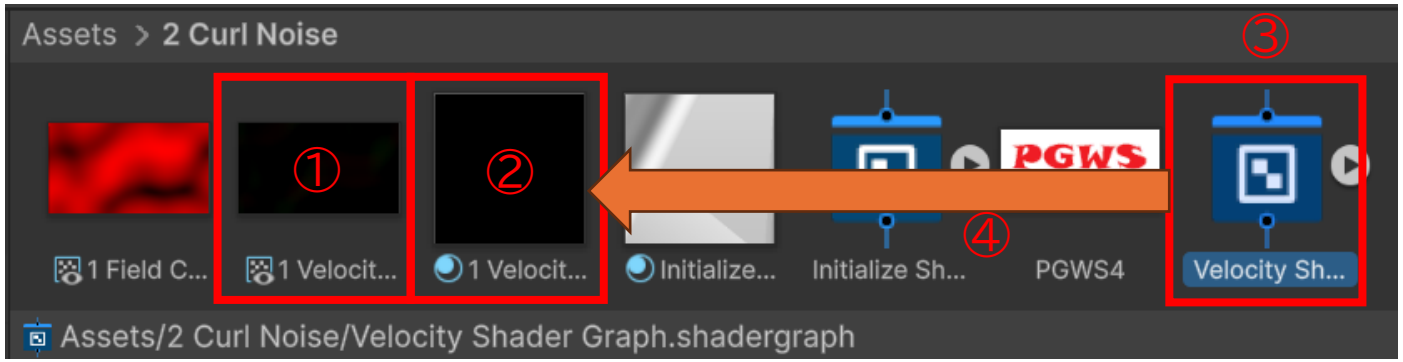
ステップ2：流れ場の生成

- ノイズによる場をCurlする
  - 「1 RawImage Velocity」で可視化する
    - 「UI/Raw Image」オブジェクト





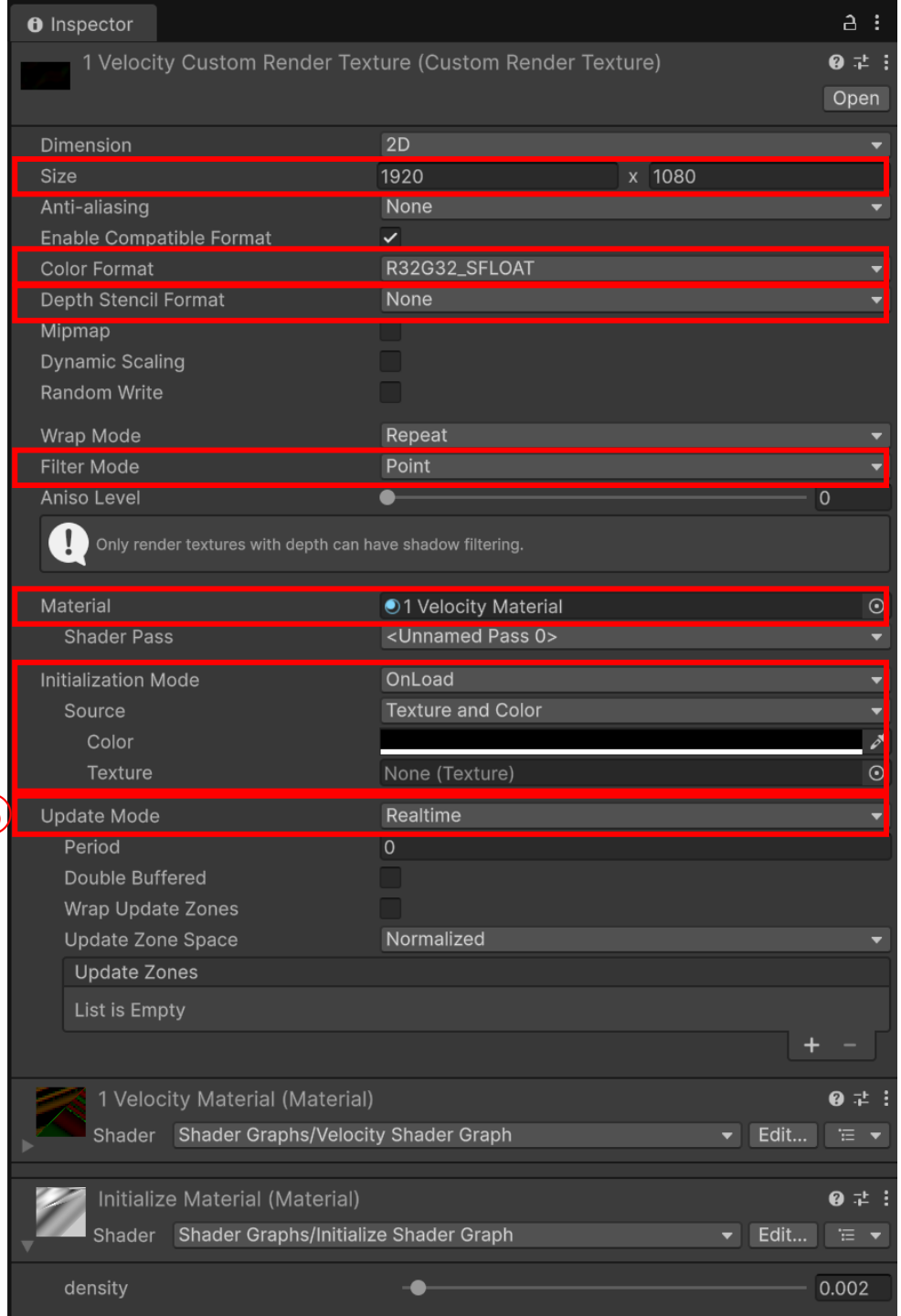
アセットの追加



1. カスタムレンダーテクスチャ
   - 名称例: 1 Velocity Custom Render Texture
2. マテリアル
   - 名称例: 1 Velocity Material
3. シェーダグラフ(Custom Render Texture)
   - 名称例: Velocity Shader Graph
   4. 「1 Velocity Material」に設定

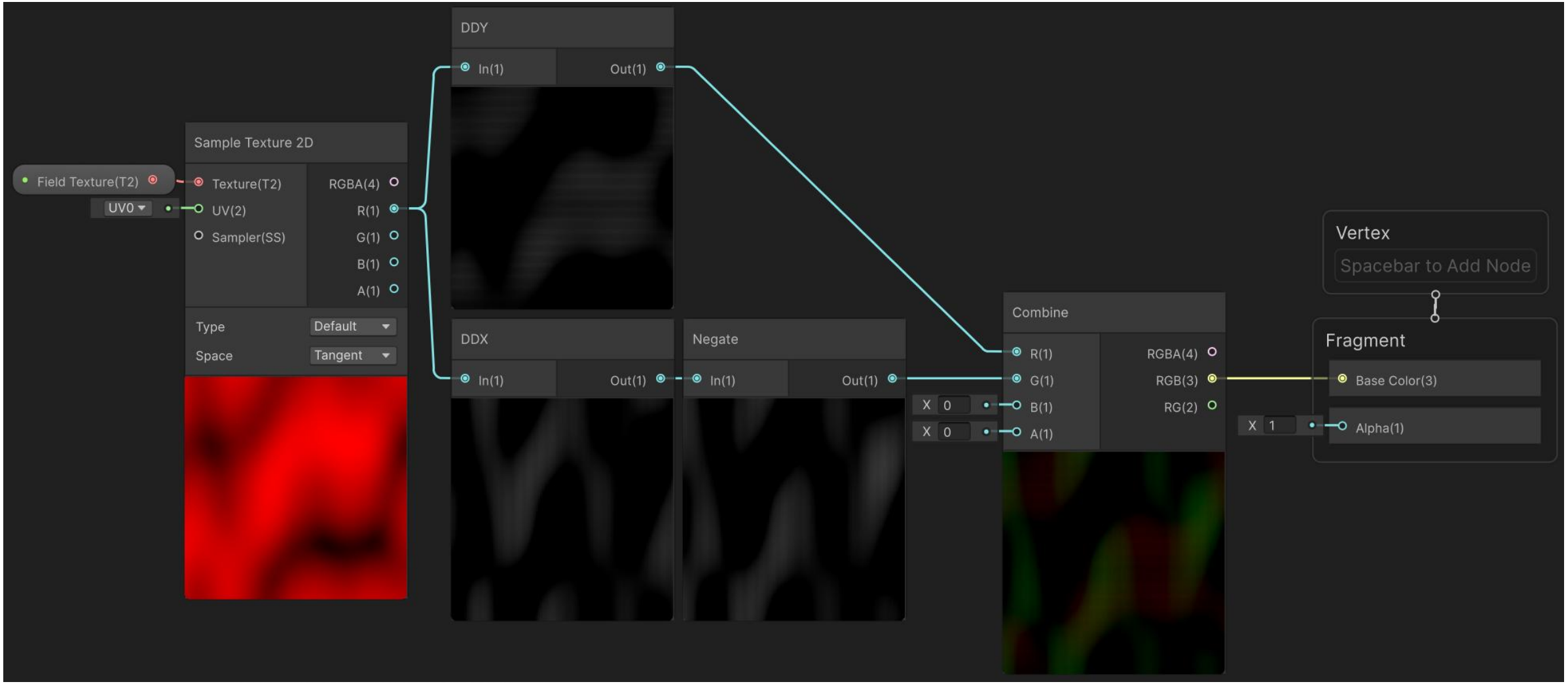
Custom Render Textureの設定

1. サイズ:フルHD(1920×1080)
   - Initialize Shader Graphと合わせる
     - 合わせなくても動くが、情報の無駄が少ない
2. フォーマット:R32G32_SFLOAT
   - 負の値も使うので符号付き
   - 精度を高めるために浮動小数点数
     3. サンプラーは「Point」しか使えない(HW的に)
4. 深度バッファは不要
5. 更新用のマテリアルを設定
   6. Inspectorでのパラメータ調整を反映できるようにするために「Realtime」更新
7. 今回は初期化は適当で良い



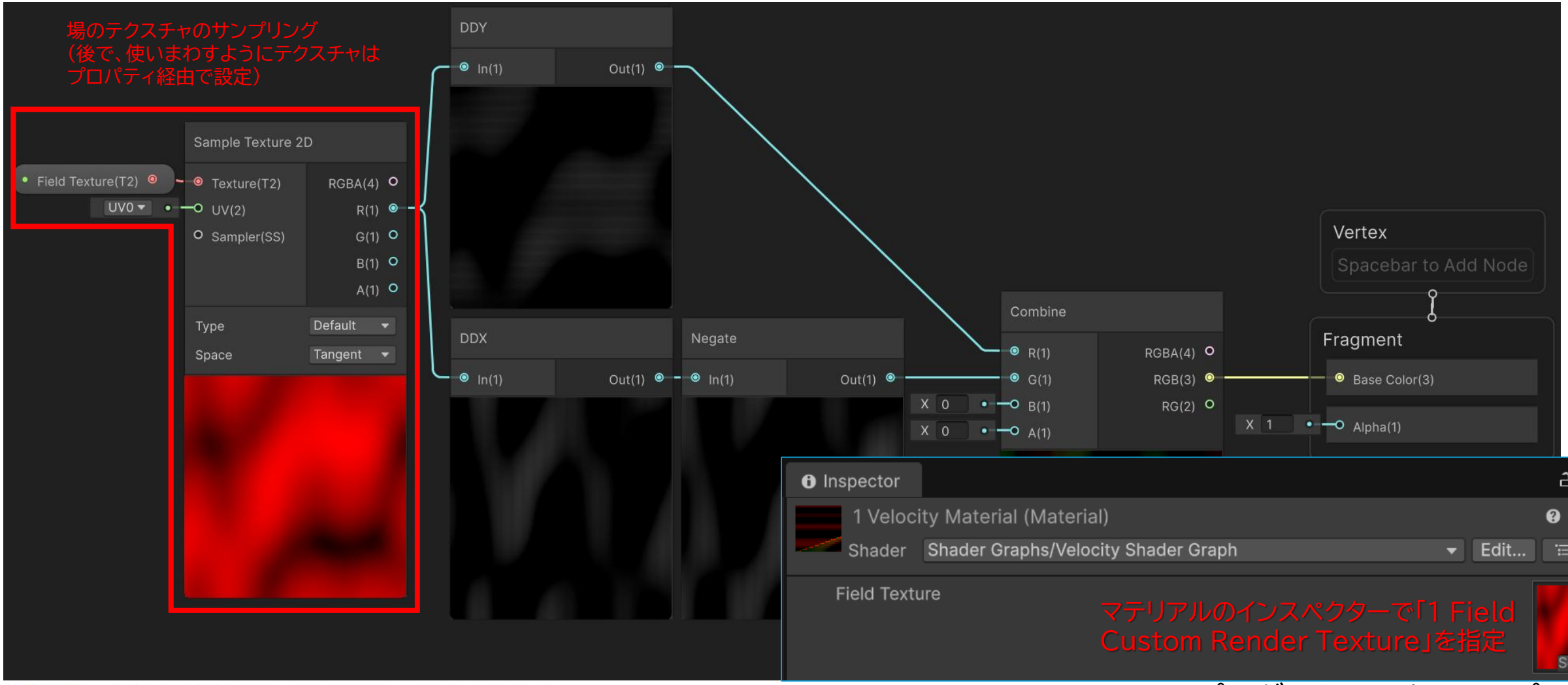
シェーダグラフ

Velocity Shader Graph



シェーダグラフ

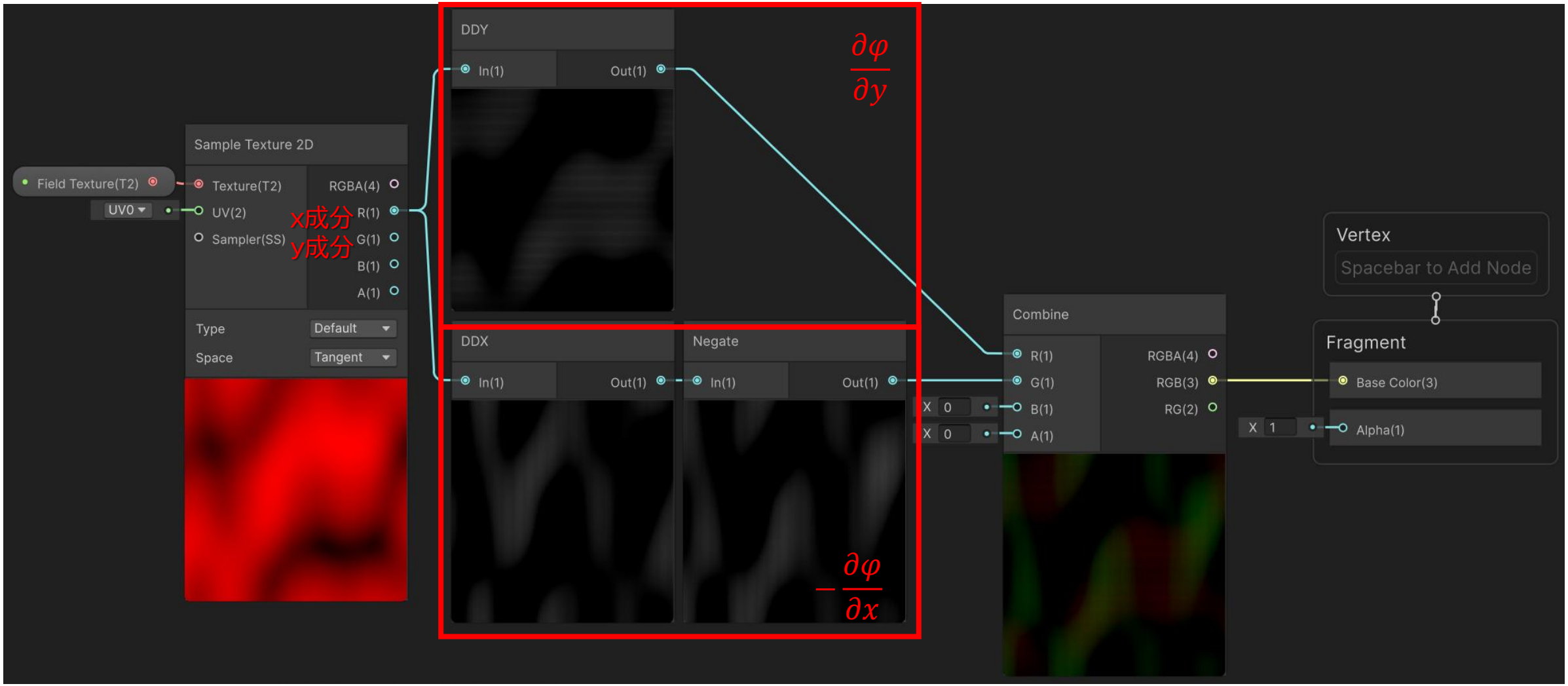
Velocity Shader Graph



シェーダグラフ

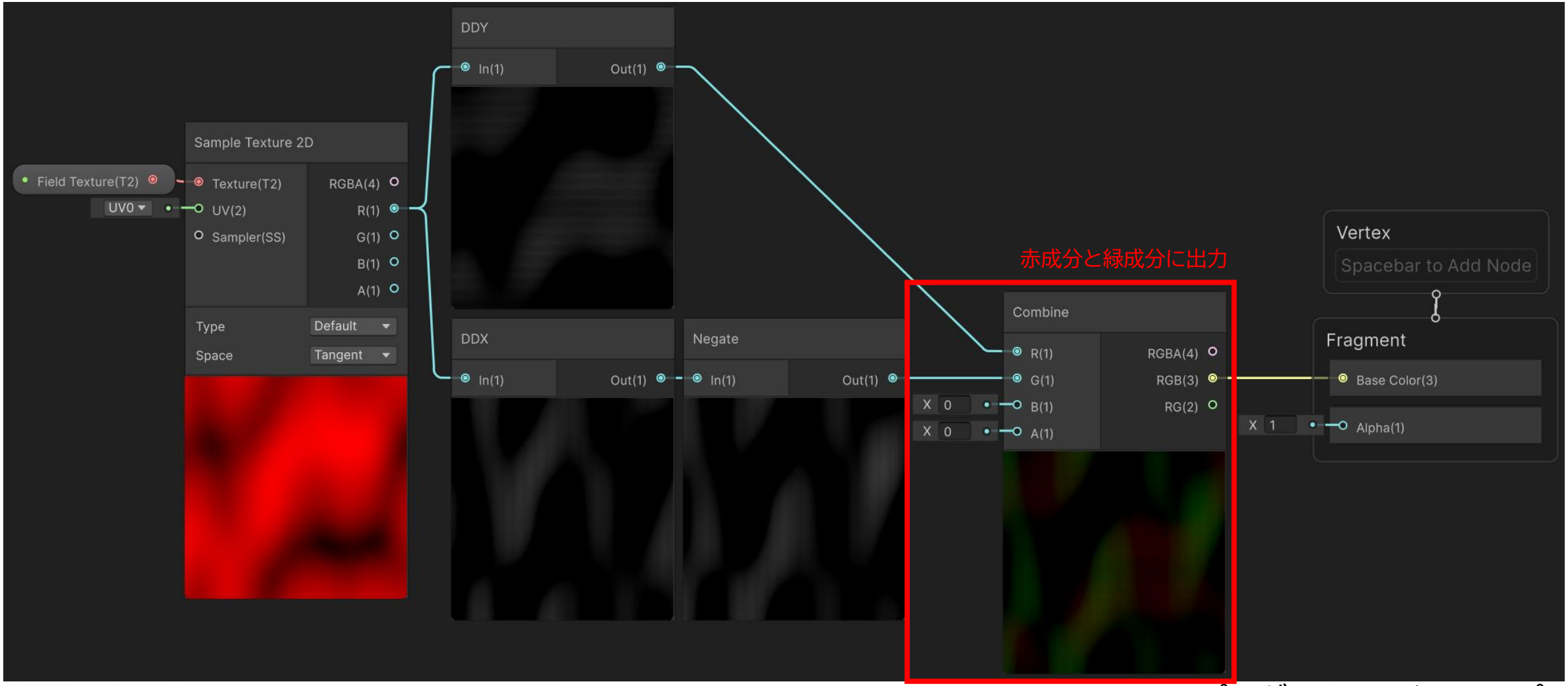
∇×φの計算

Velocity Shader Graph

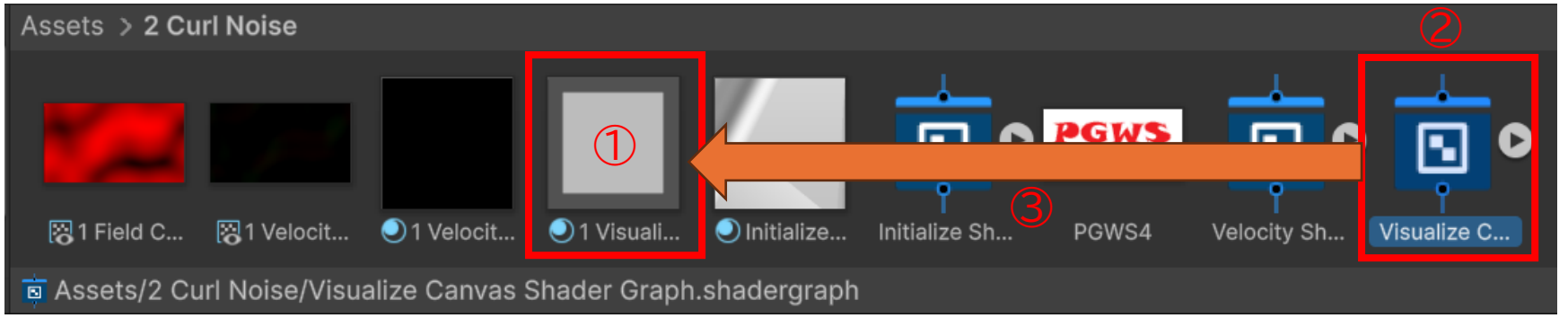


シェーダグラフ

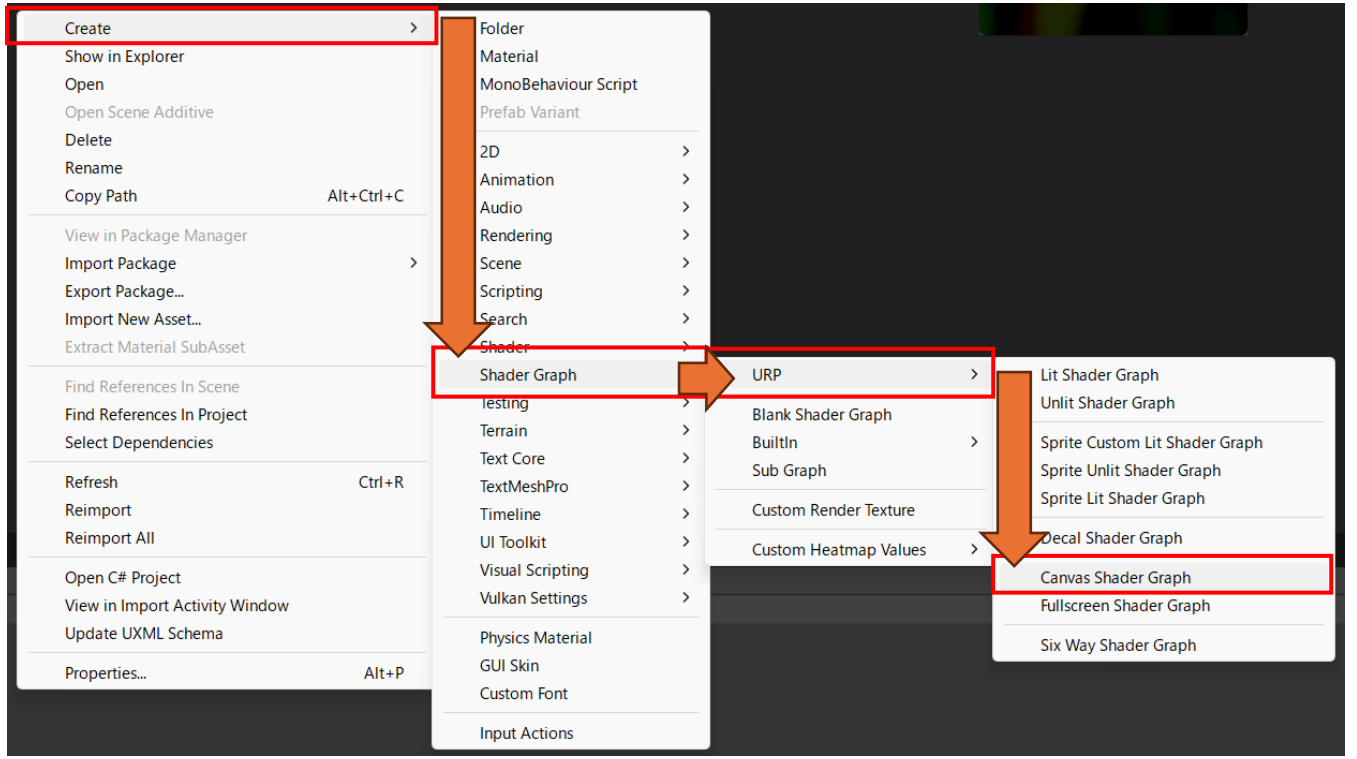
Velocity Shader Graph



可視化

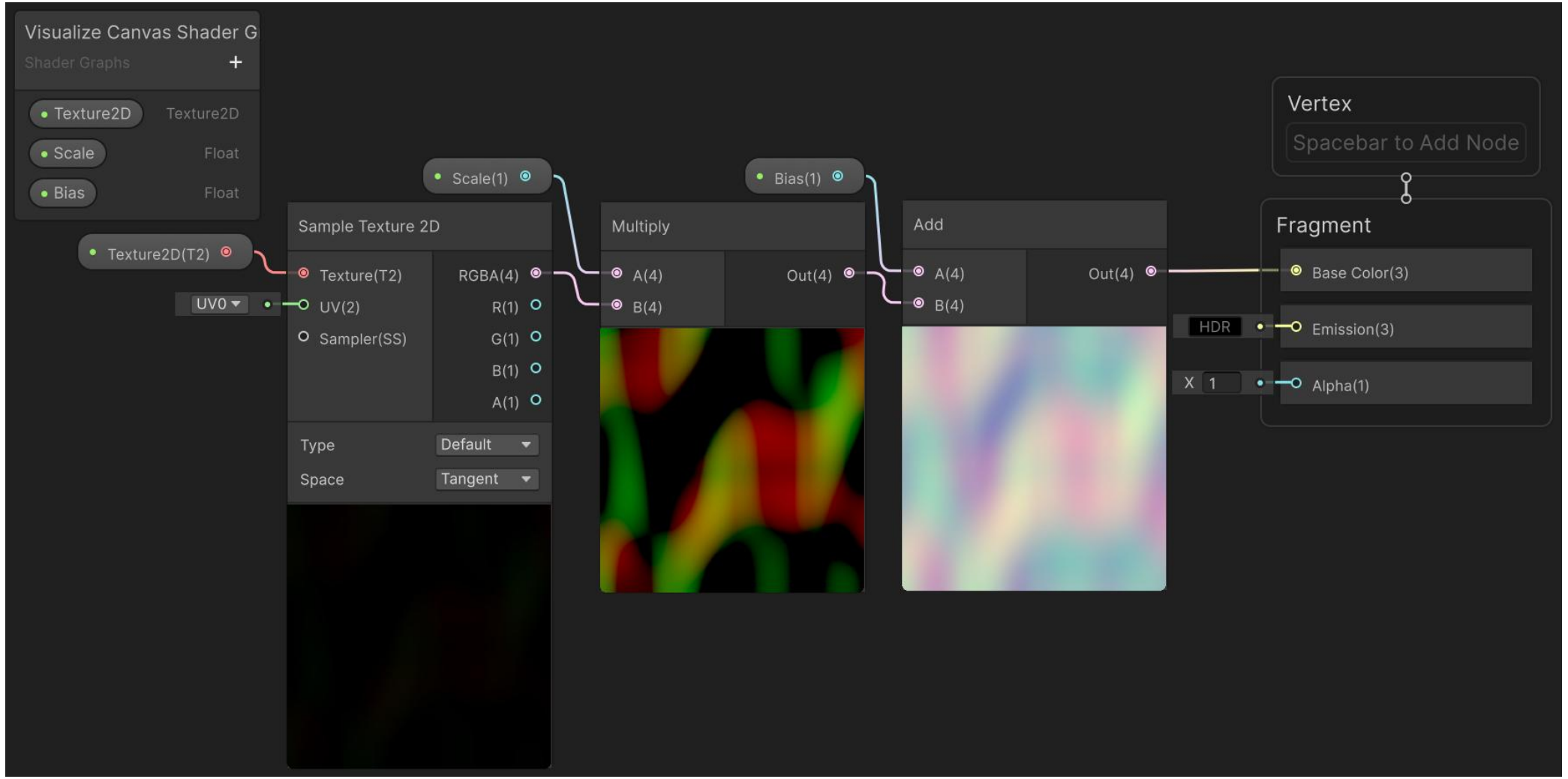


- 値が小さく、負の値を含むため、そのままの表示では見難い
- 可視化用のマテリアルとシェーダグラフを追加
  1. マテリアル
     - 名称例：1 Visualize Velocity Material
  2. シェーダグラフ
     - 種類：Canvas Shader Graph
       - UI用
     - 名称例：Visualize Canvas Shader Graph
  3. シェーダグラフをマテリアルに設定



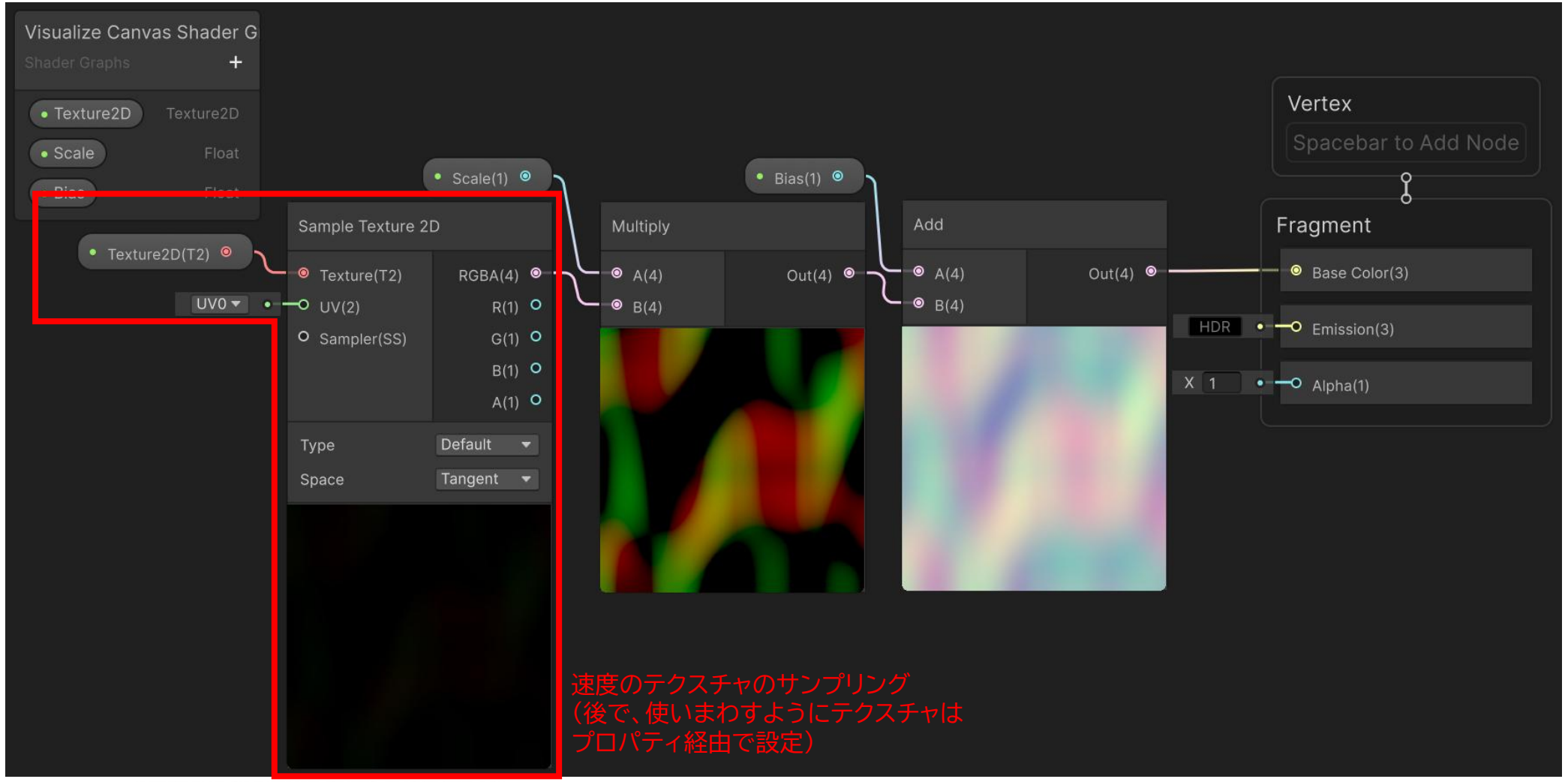
シェーダグラフ

Visualize Canvas Shader Graph



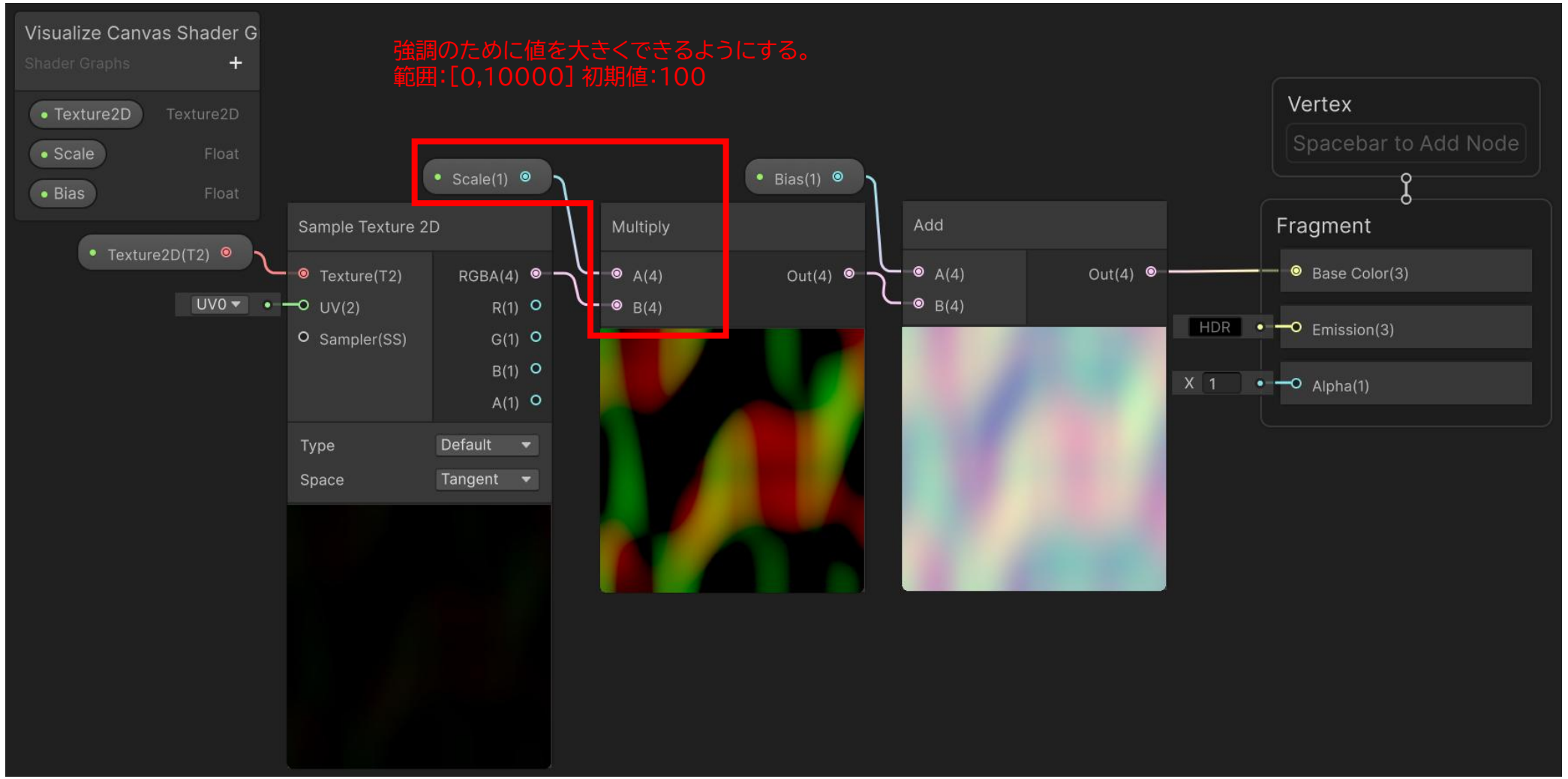
シェーダグラフ

Visualize Canvas Shader Graph



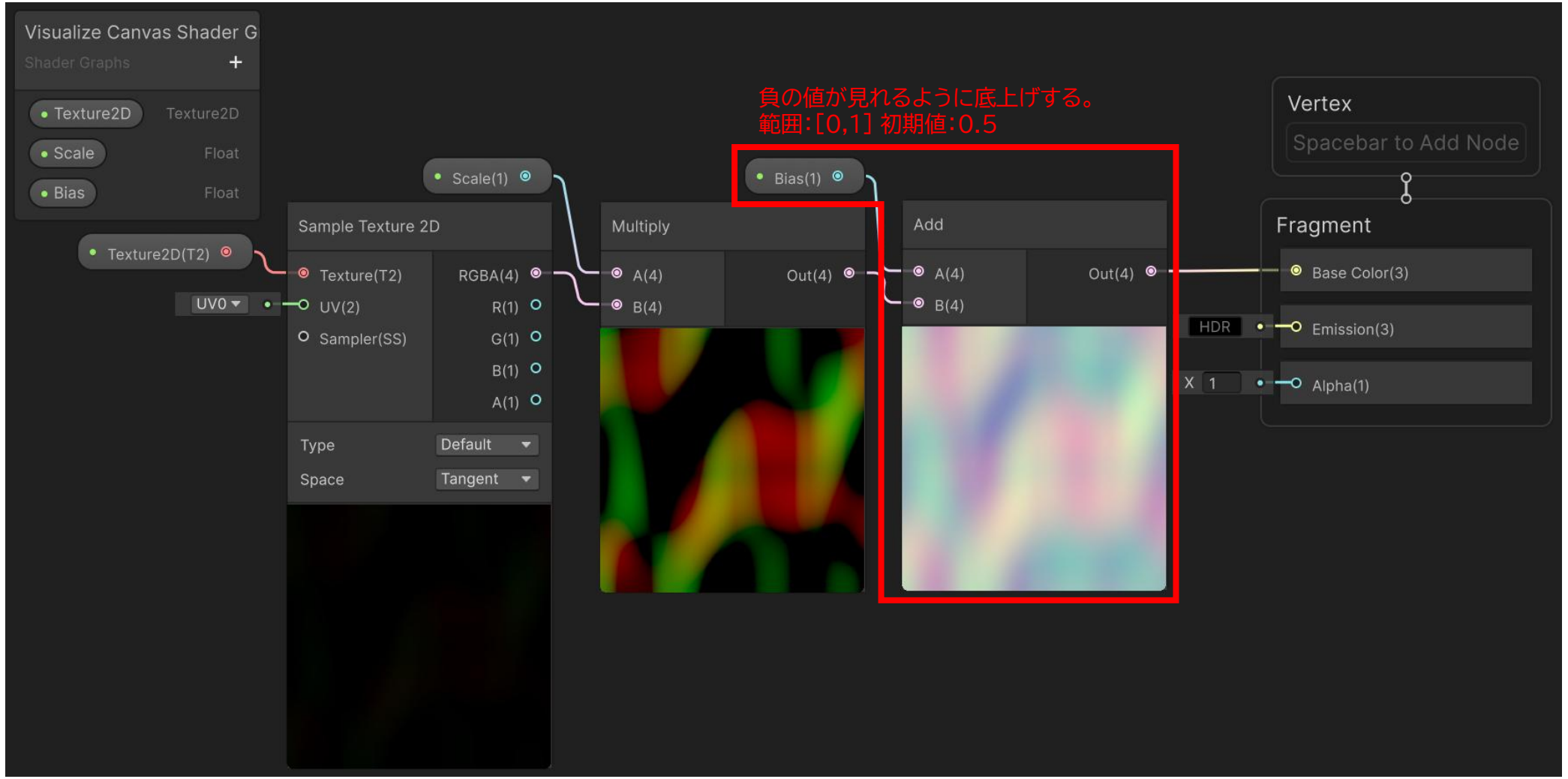
シェーダグラフ

Visualize Canvas Shader Graph



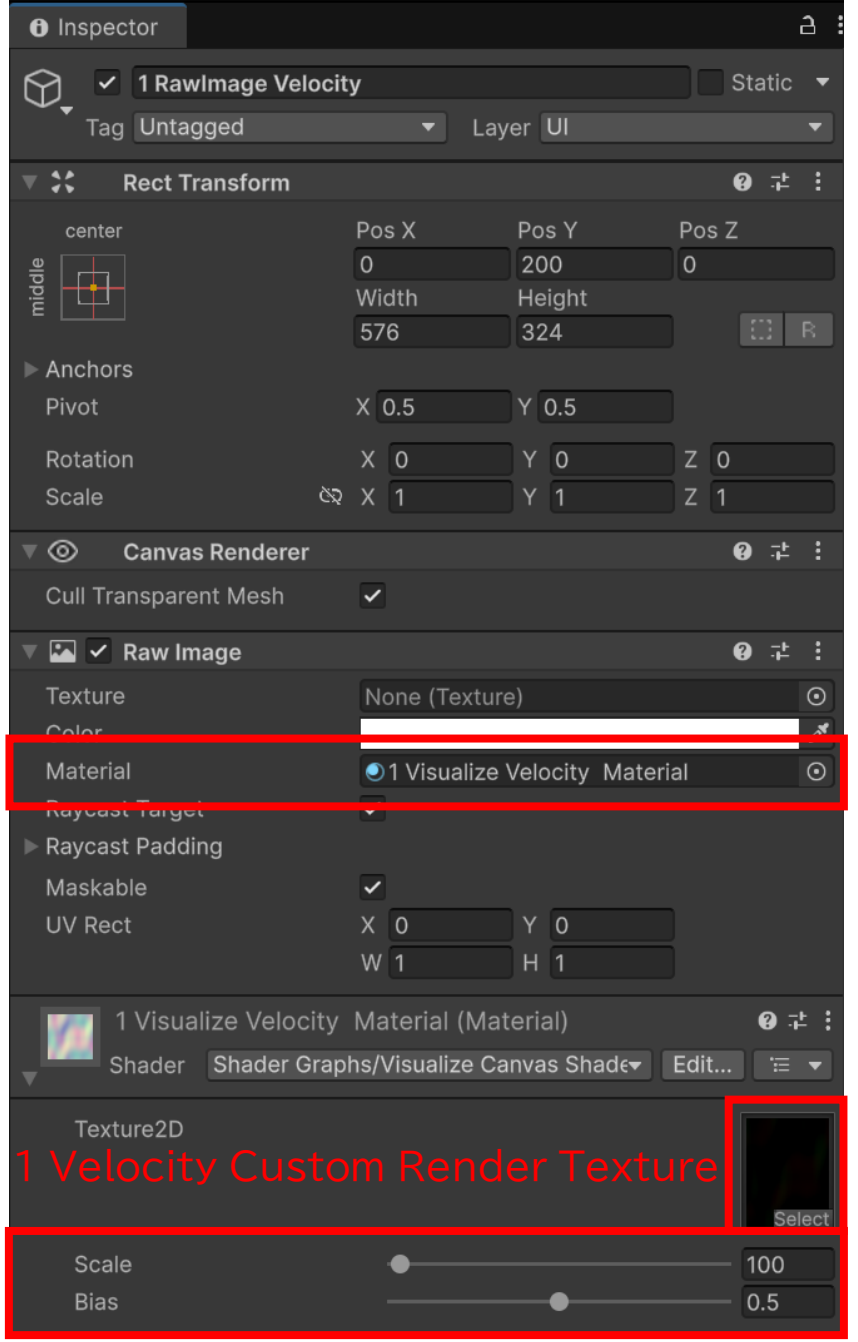
シェーダグラフ

Visualize Canvas Shader Graph



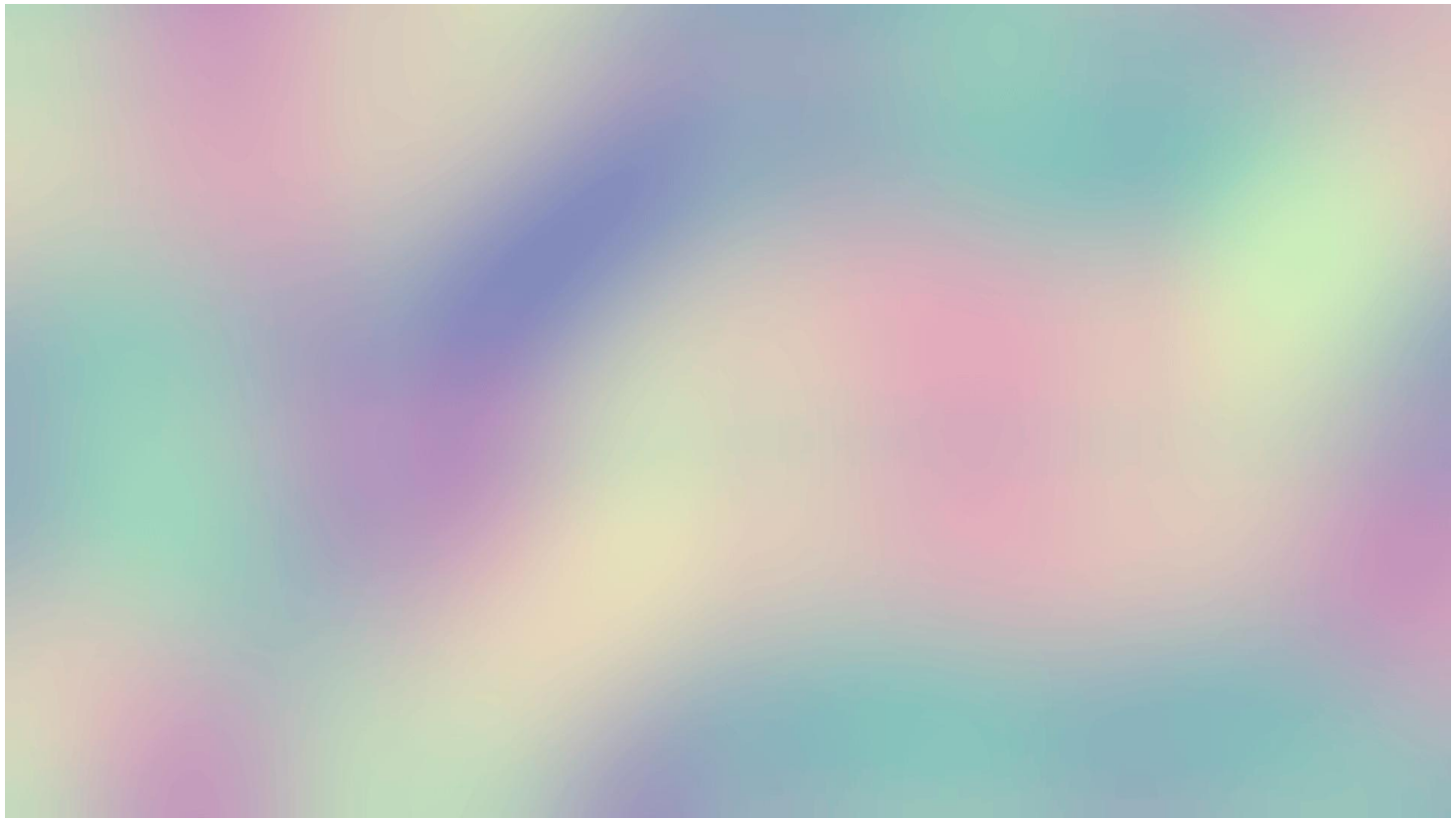
オブジェクトの設定

1. 描画にマテリアルを設定
2. マテリアルでテクスチャを設定
    3. 他のパラメータは、結果が見やすいように



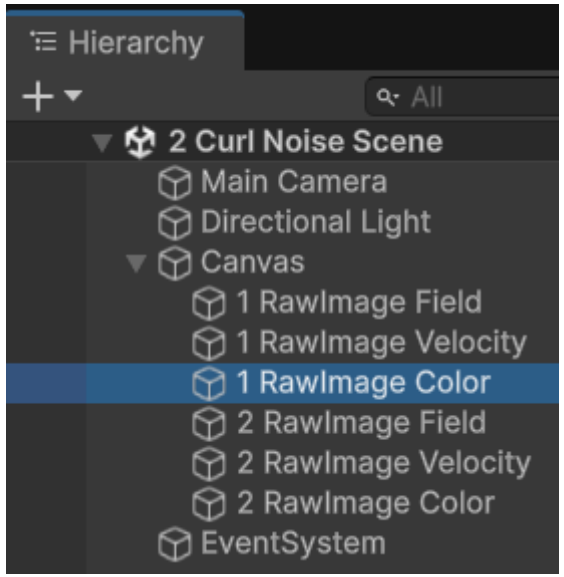
やってみよう
ステップ2：流れ場の生成

- Inspectorで値（Initialize Materialのdensity）を変更してみよう



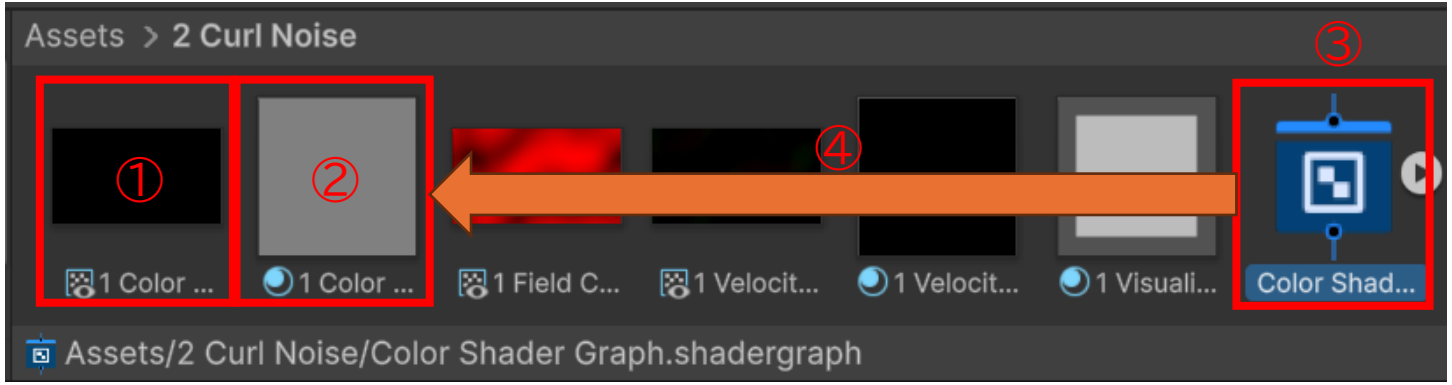
ステップ3：模様を流す

- 流れ場で模様を動かす：速度の量だけ反対向きの位置を読み込む
  - 「1 RawImage Color」で可視化する
    - 「UI/Raw Image」オブジェクト



PGWS 4

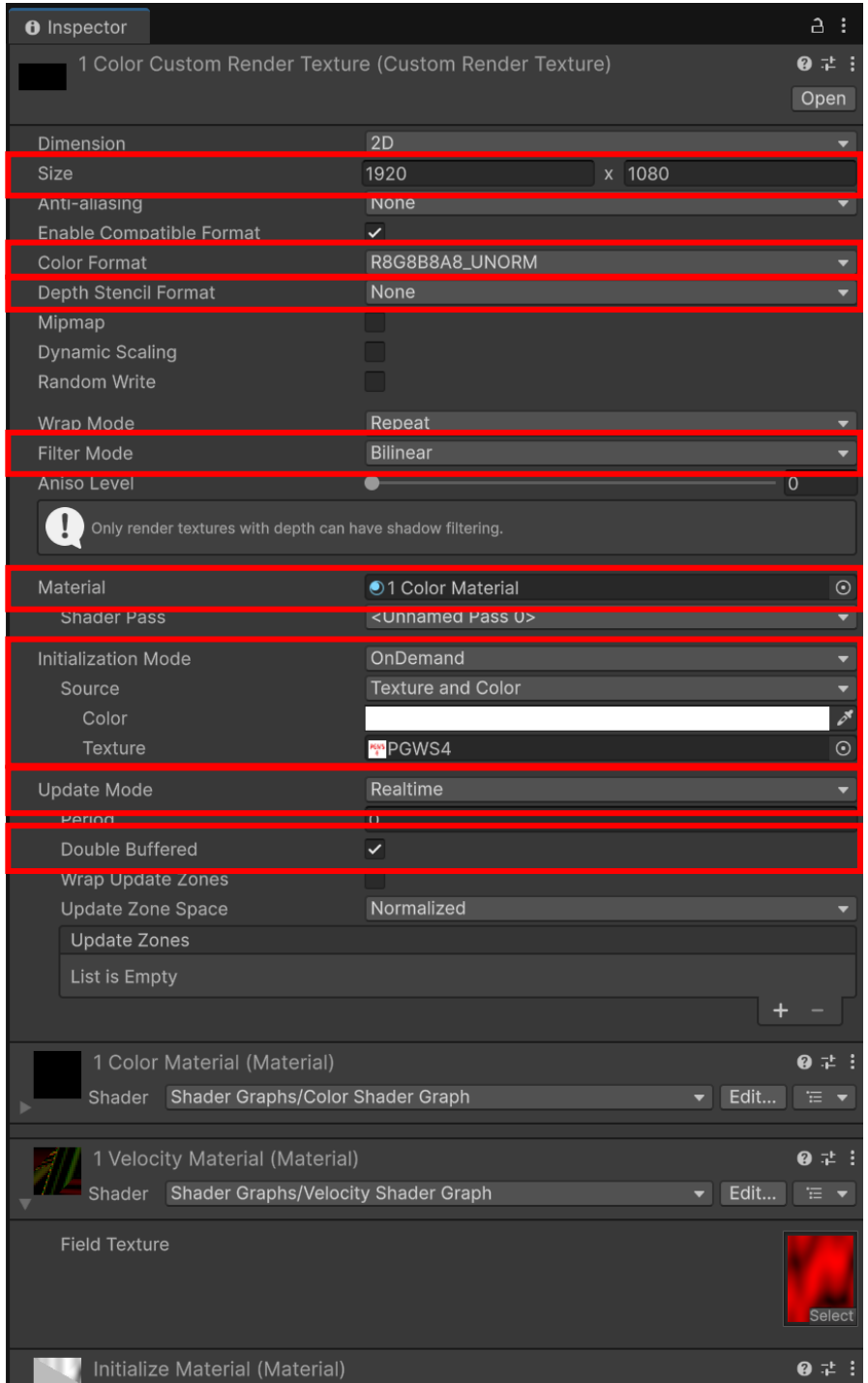
アセットの追加



1. カスタムレンダーテクスチャ
   - 名称例：1 Color Custom Render Texture
2. マテリアル
   - 名称例：1 Color Material
3. シェーダグラフ（Custom Render Texture）
   - 名称例：Color Shader Graph
   4. 「1 Color Material」に設定

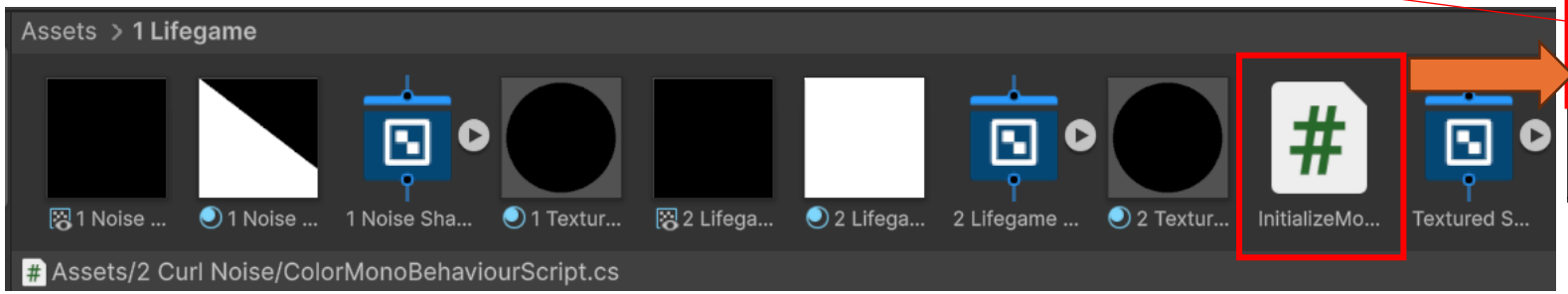
Custom Render Textureの設定

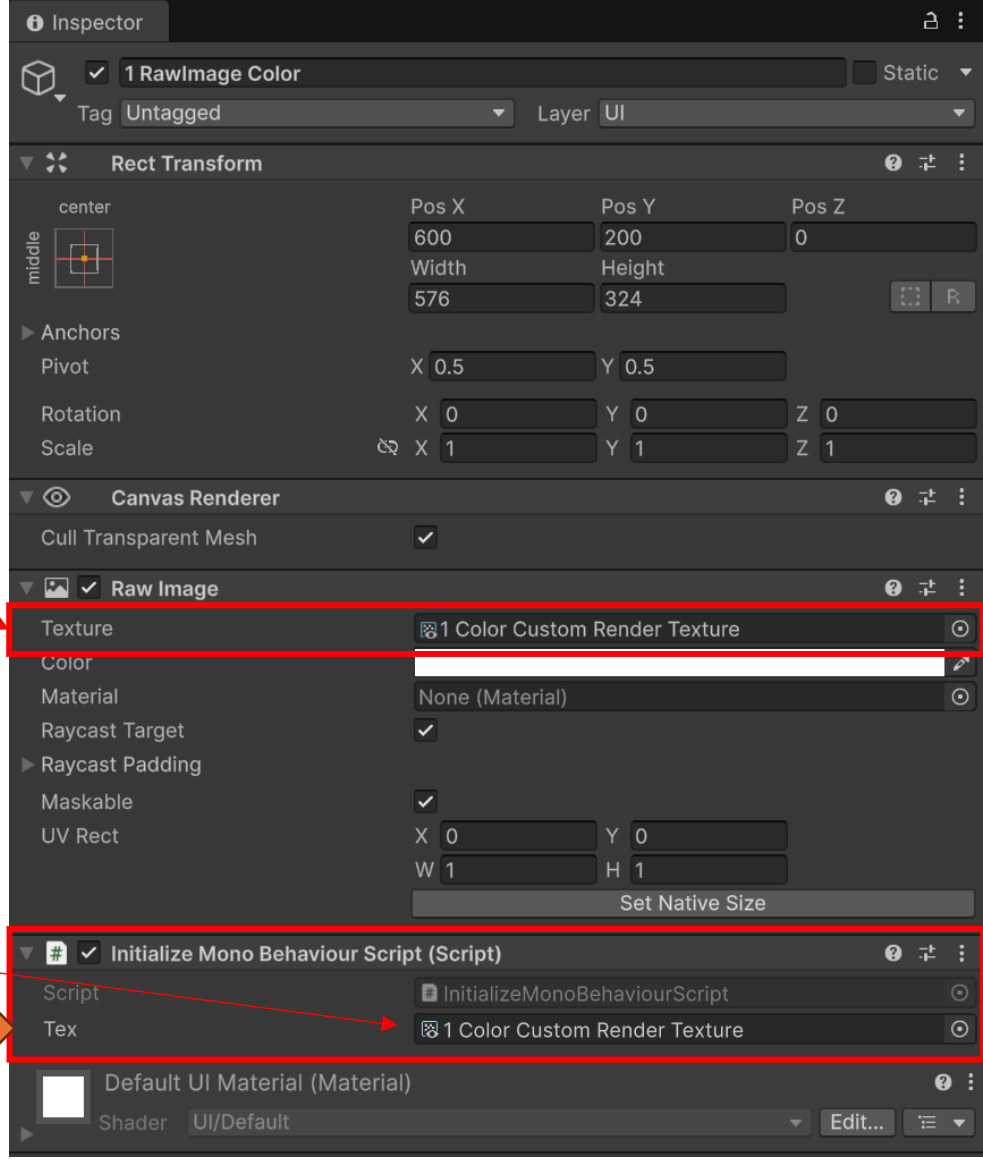
1. サイズ：フルHD(1920×1080)
   - 1 Velocity Custom Render Textureと合わせる
     - 合わせなくても動くが、情報の無駄が少ない
2. フォーマット：R8G8B8A8_UNORM
   - 表示されるものなので、高い精度は不要
   3. サンプラーは「Bilinear」が使える
4. 深度バッファは不要
5. 更新用のマテリアルを設定
   6. Inspectorでのパラメータ調整を反映できるようにするために「Realtime」更新
7. 初期化はスクリプトで実行(OnDemend)
   - 色(白)とテクスチャの乗算
8. ダブルバッファ



オブジェクトの設定

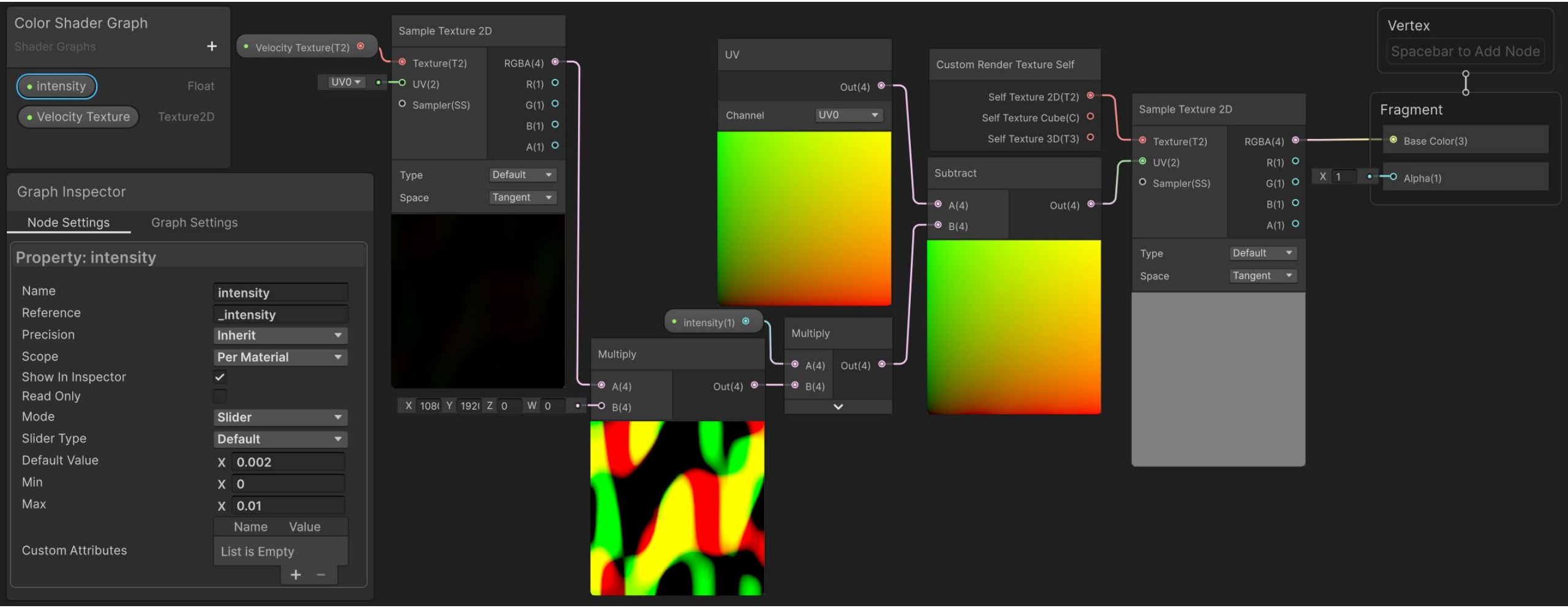
- オブジェクトのテクスチャに
  カスタムレンダーテクスチャを設定
- 更新用にスクリプトを追加
  - 「1 Lifegame」に作った「InitializeMono BehaviourScript」を利用
    - スクリプトのテクスチャにも「1 Color Custom Render Texture」を設定





シェーダグラフ

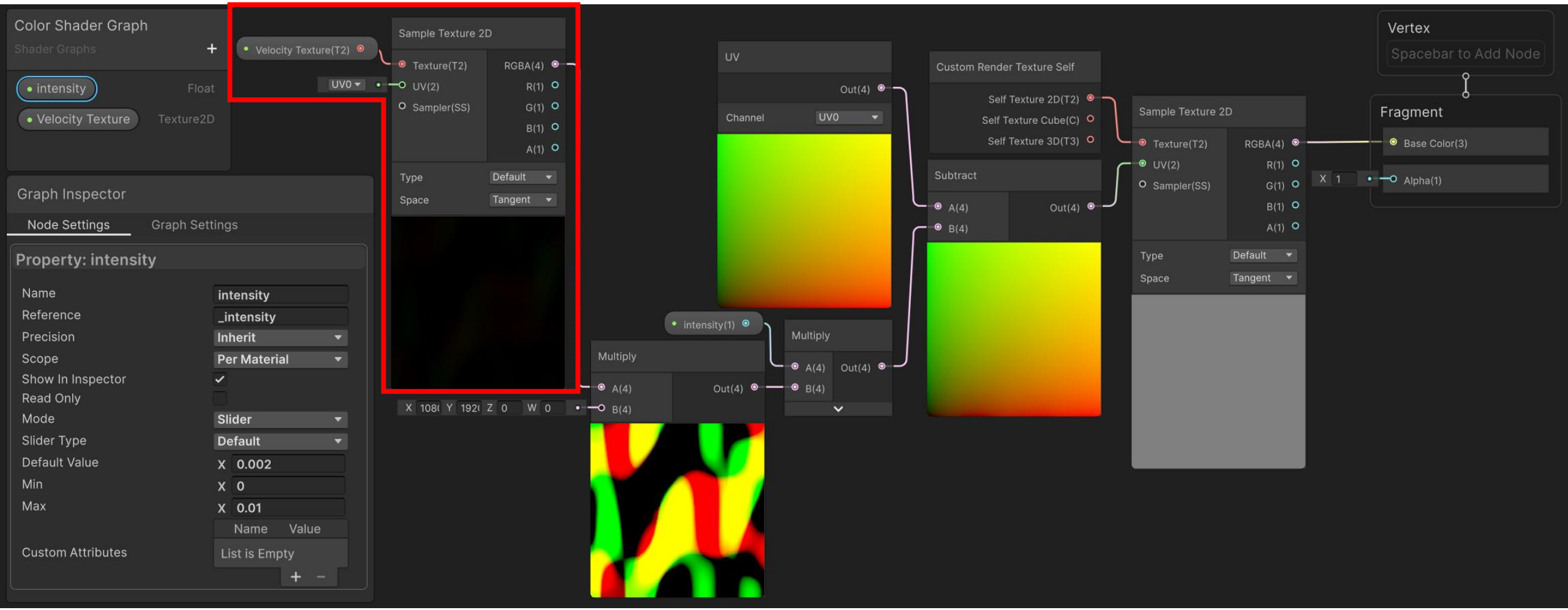
Color Shader Graph



シェーダグラフ

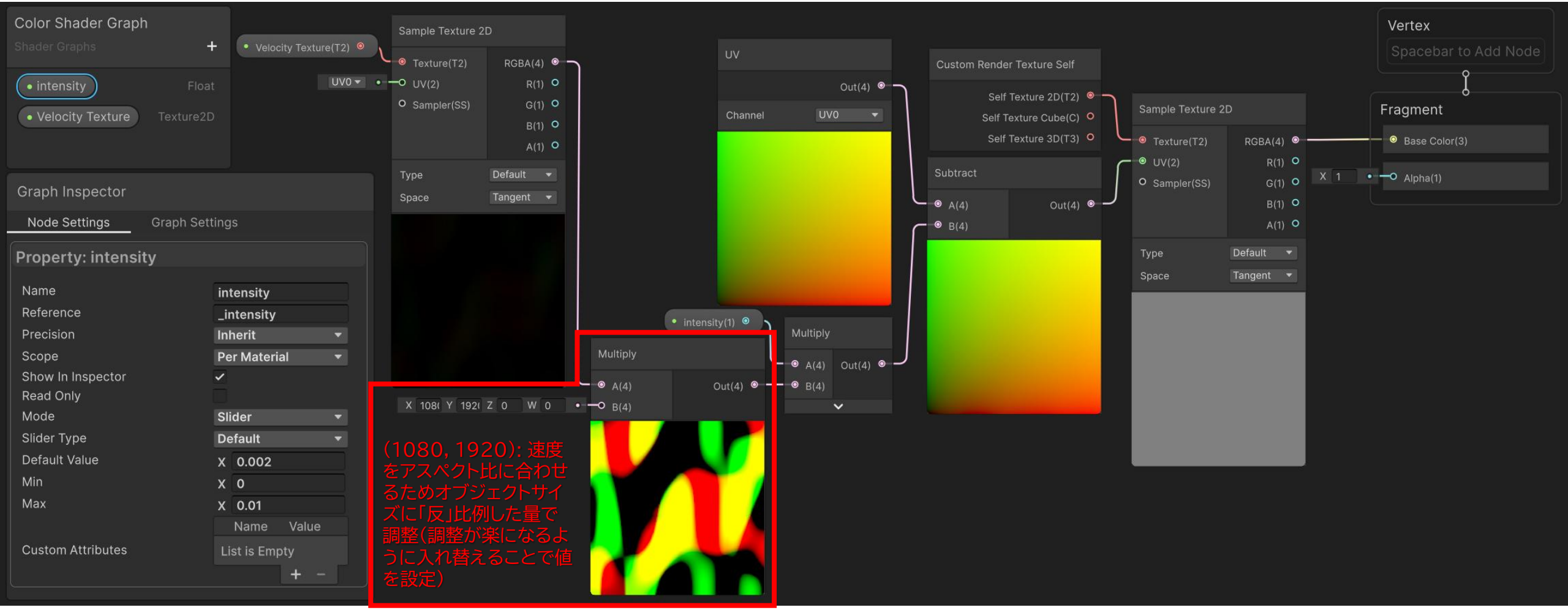
速度のテクスチャのサンプリング
（後で、使いまわすようにテクスチャは
プロパティ経由で設定）

Color Shader Graph



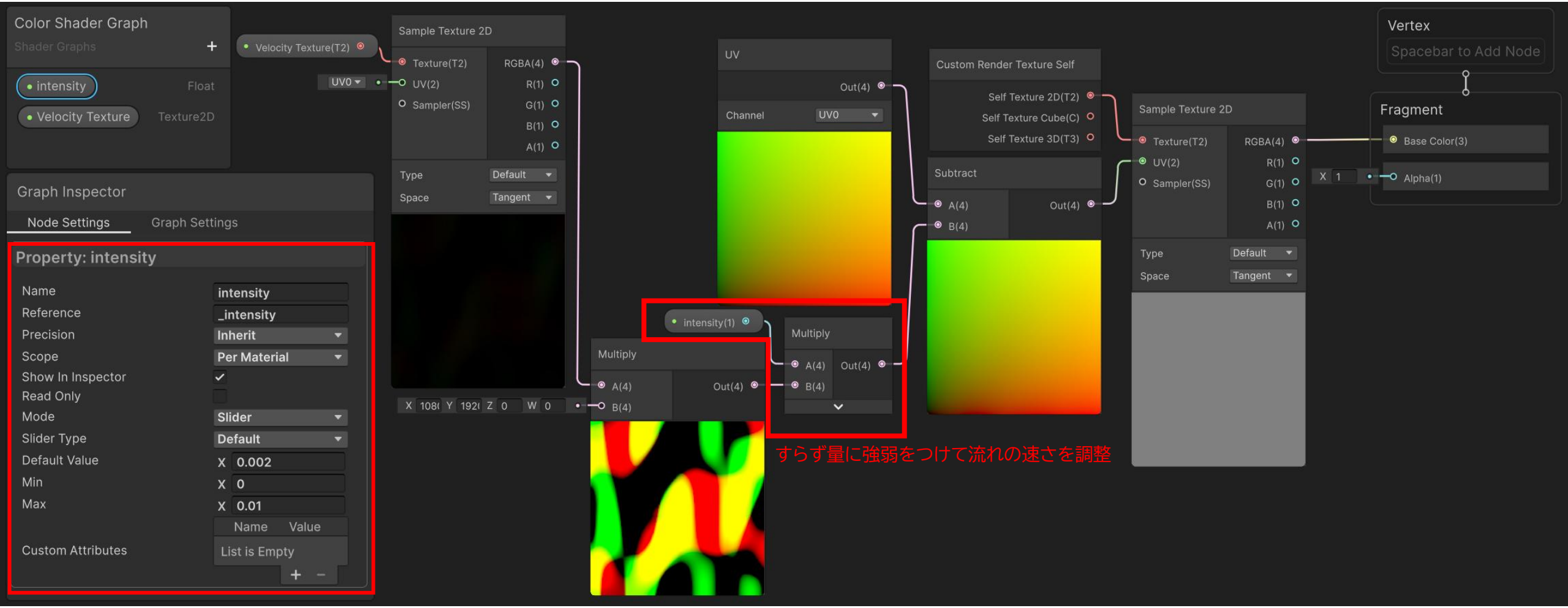
シェーダグラフ

Color Shader Graph



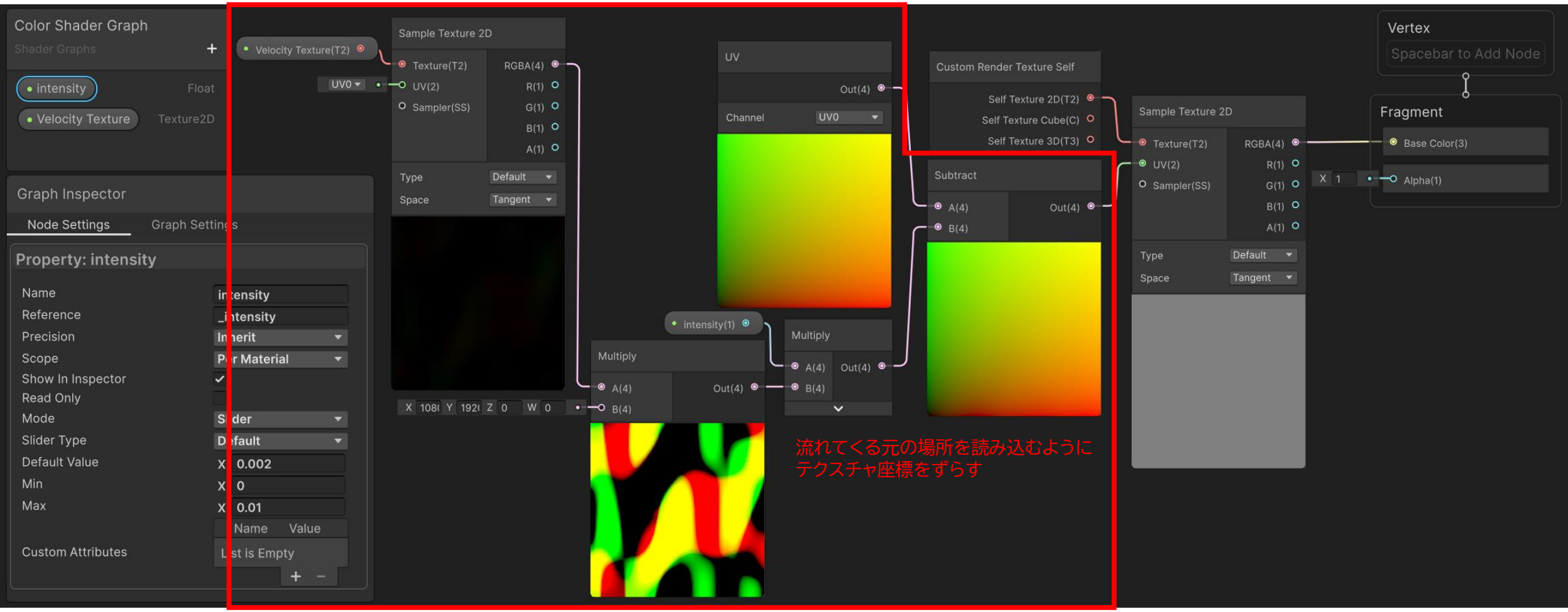
シェーダグラフ

Color Shader Graph



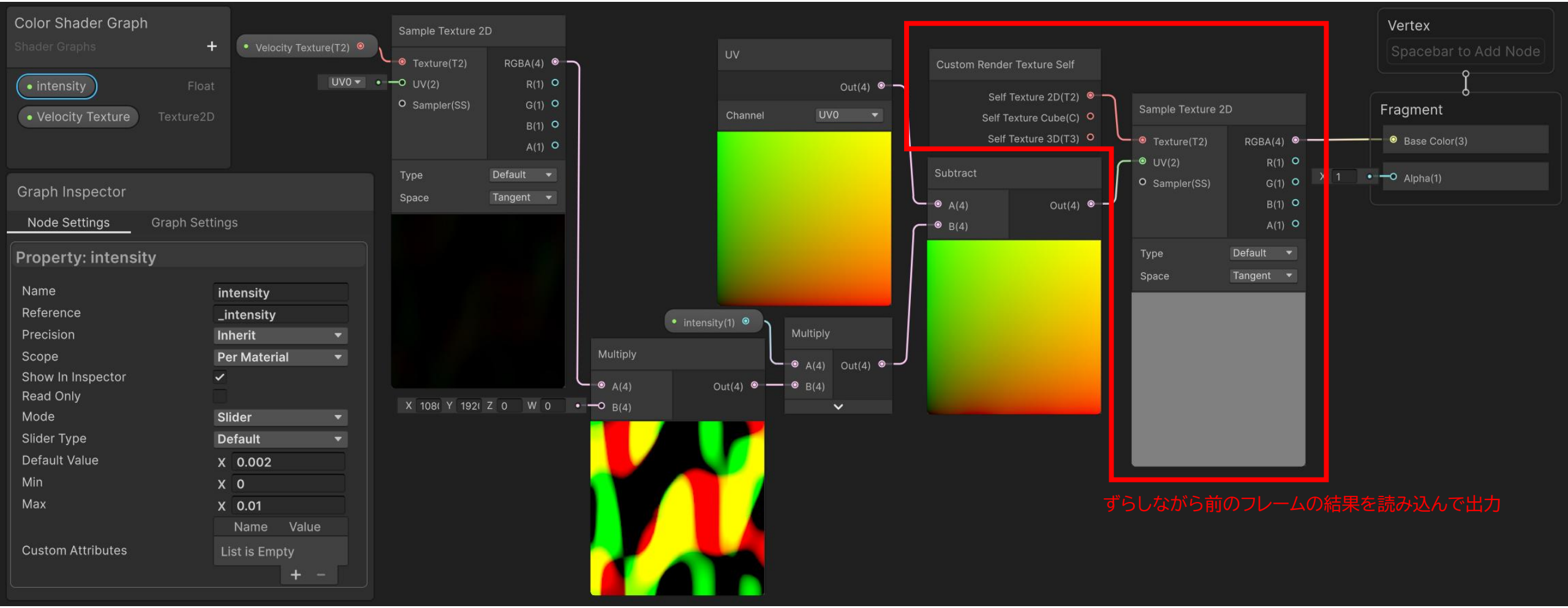
シェーダグラフ

Color Shader Graph

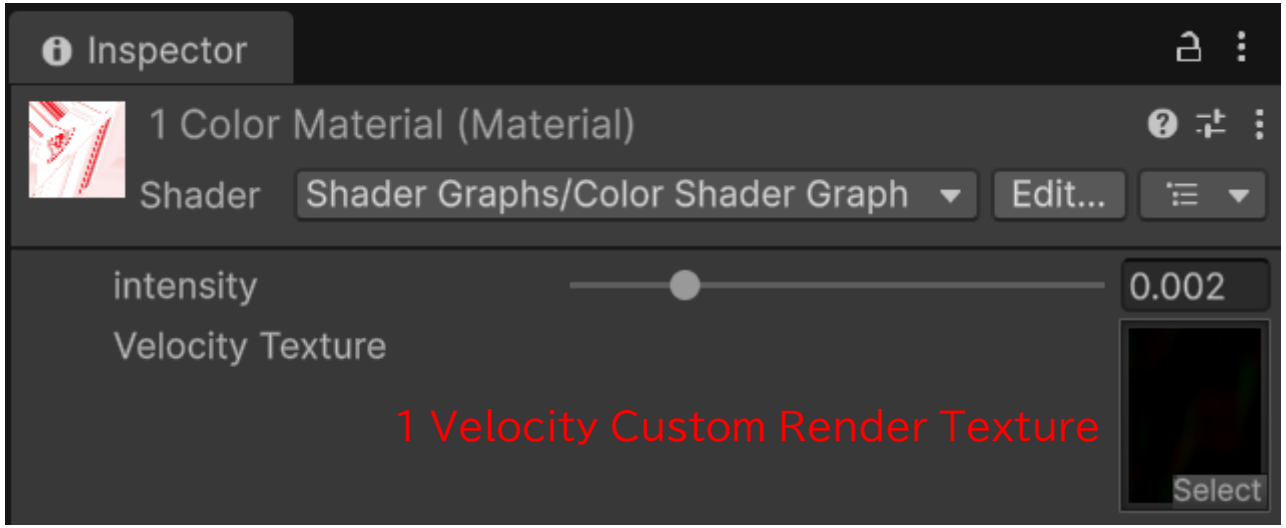


シェーダグラフ

Color Shader Graph



マテリアルの設定

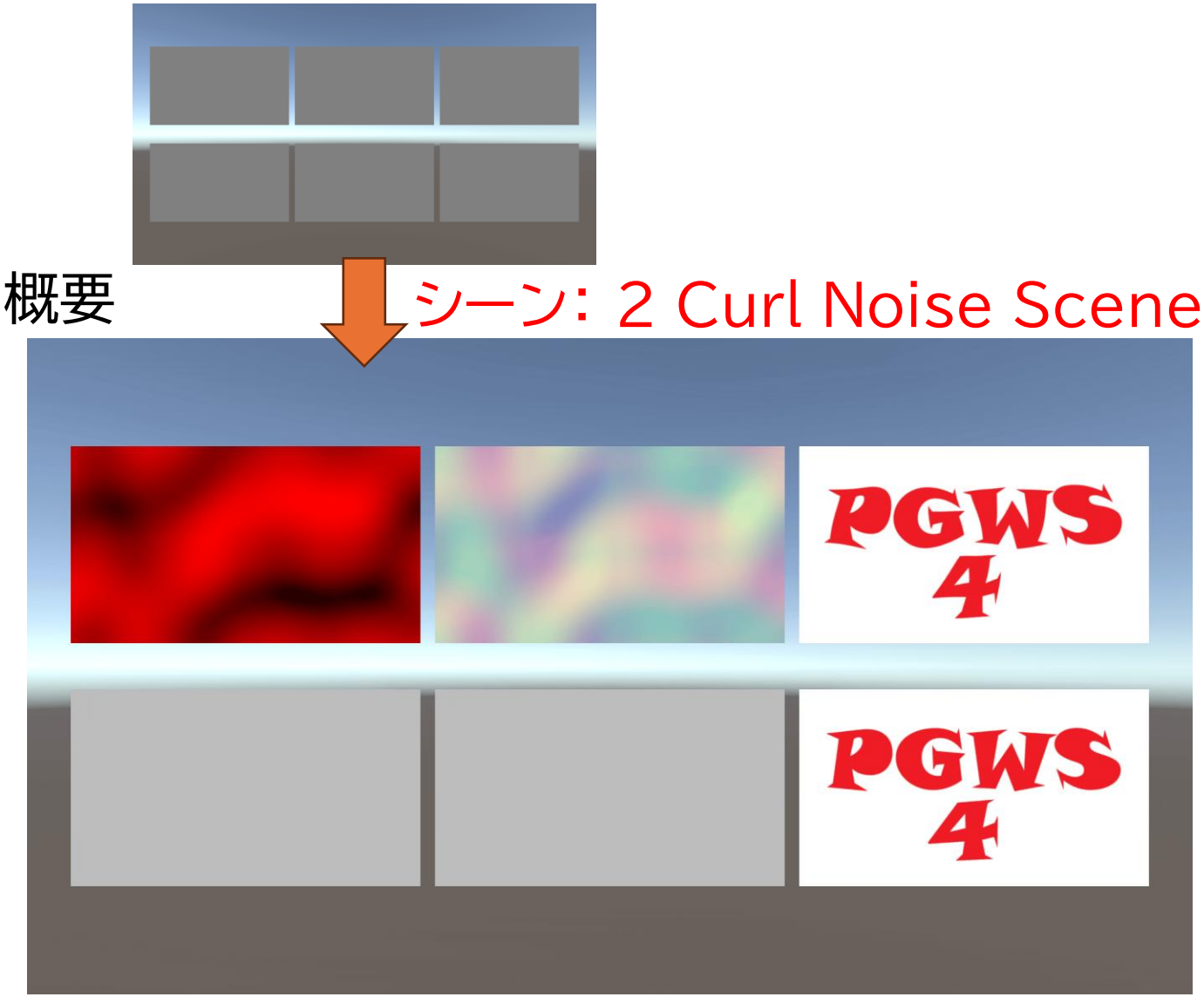

Inspector
1 Color Material (Material)
Shader
Shader Graphs/Color Shader Graph
Edit...
intensity
0.002
Velocity Texture
1 Velocity Custom Render Texture
Select

やってみよう
ステップ3：模様を流す

- Inspectorで値(intensityなど)を変更してみよう
- 初期化用のテクスチャを
  自分の好きなものに
  変えてみよう

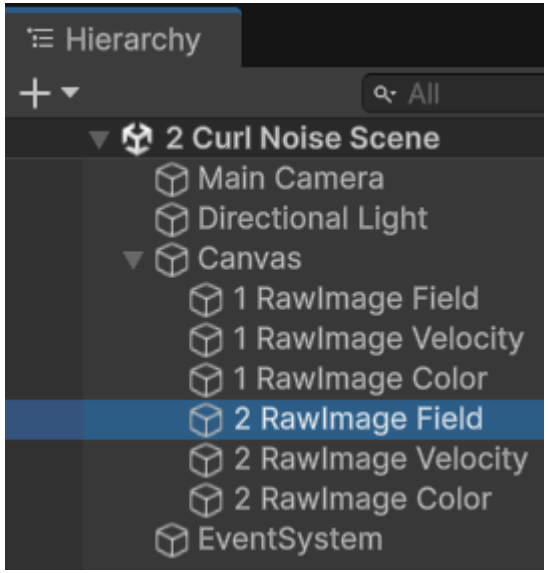
本日の内容

- カスタムレンダーテクスチャ
  - カスタムレンダーテクスチャの概要
  - ライフゲーム
  - カールノイズ
    - ノイズを使って模様を流す
    - インタラクティブに動かす



ステップ4:ドラッグで初期化

- ドラッグでカーソルが合った場所を赤くする
  - 「2 RawImage Field」で可視化する
    - 基準色は灰色だが利用は赤成分のみ
    - 「UI/Raw Image」オブジェクト



インタラクティブに動かす

- 画面上でドラッグした際にその動きで模様を動かす
  1. ドラッグの場所を検出する
  2. ドラッグした場所の場を変化させる
  3. 先ほどと同じ要領で流れを作成して、模様を流す

ドラッグの場所を検出する

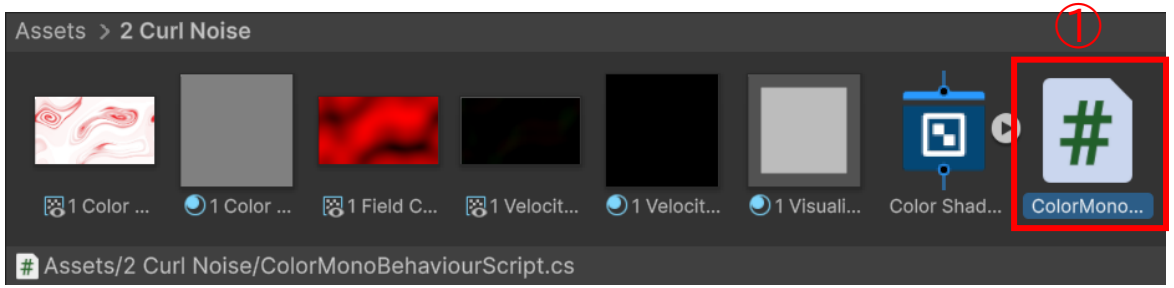
イベントにより処理

1. スクリプトを追加して、イベント時の処理を作成
2. ドラッグ対象のオブジェクトにイベントを追加
   - OnDrag()
3. イベントに応じてシェーダにパラメータを送る
4. シェーダではパラメータに応じて場を作成する

1. スクリプトを追加して、イベント時の処理を作成

1. MonoBehaviourスクリプトの追加

- 名称例:ColorMono BehaviourScript



2. メソッドを追加

```
public void OnDrag(BaseEventData e)
```

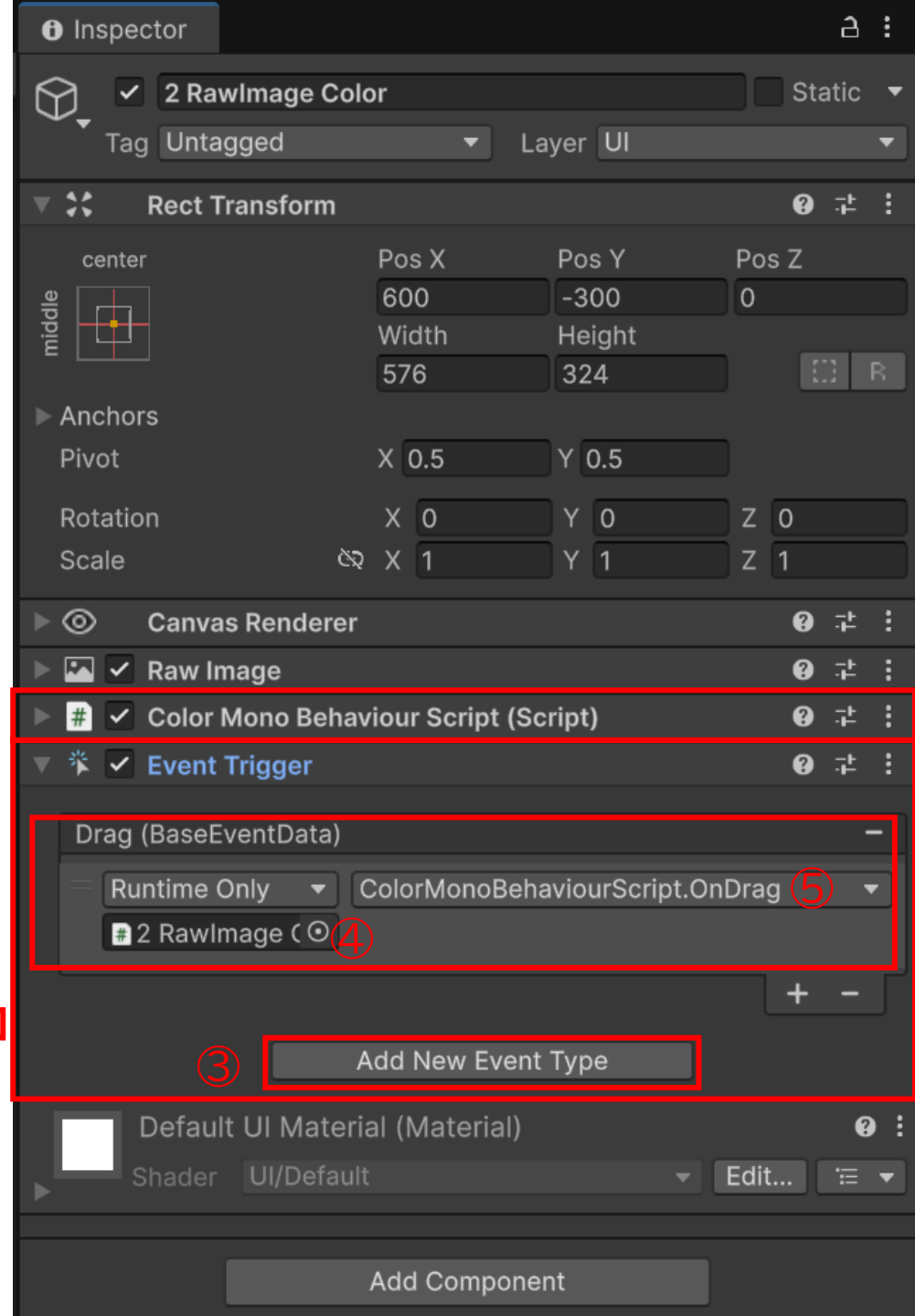
ColorMonoBehaviourScript.cs

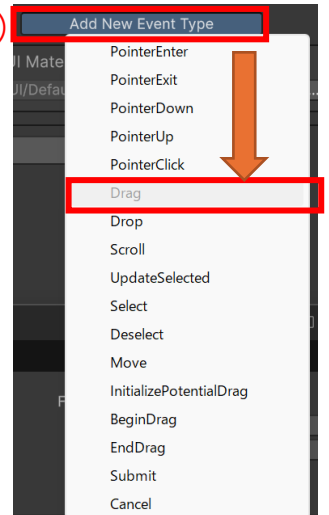
```
1   using UnityEngine;
2   using UnityEngine.EventSystems;
3   using UnityEngine.InputSystem;
4   using UnityEngine.UIElements;
5
        Unity スクリプト|0 個の参照
6   public class ColorMonoBehaviourScript : MonoBehaviour
7   {
8       // Start is called once before the first execution
            Unity メッセージ|0 個の参照
9       void Start()...
14
15      // Update is called once per frame
            Unity メッセージ|0 個の参照
16      void Update()...
21
            0 個の参照
22  ②   public void OnDrag(BaseEventData e)...
33  }
```

2.ドラッグ対象のオブジェクトにイベントを追加

1. ドラッグ対象のオブジェクトにスクリプトを追加
2. ドラッグ対象のオブジェクトに「Event Trigger」コンポーネントを追加
3. Event Triggerコンポーネントの「Add New Event Type」から「Drag」イベントを追加

- Dragイベントを設定
    4. 自分自身(2 Raw Inage Color オブジェクト)
    5. ColorMonoBehaviour Script.OnDragメソッド





KOGEI TOKYO POLYTECHNIC UNIVERSITY

3. イベントに応じてシェーダにパラメータを送る

ColorMonoBehaviourScript.cs

```
1   using UnityEngine;
2   using UnityEngine.EventSystems;
3   using UnityEngine.InputSystem;
4   using UnityEngine.UIElements;
5
    Unity スクリプト | 0 個の参照
6   public class ColorMonoBehaviourScript : MonoBehaviour
7   {
8       [SerializeField] Material matField = default!; シェーダーのマテリアル
9       float intensity = 0.0f; ドラッグ時以外に0を与えるためのメンバー
10
11      // Start is called once before the first execution of Update after the MonoBehaviour is created
        Unity メッセージ | 0 個の参照
12      void Start()...
17
18      // Update is called once per frame
        Unity メッセージ | 0 個の参照
19      void Update()
20      {
21          matField.SetFloat("_Intensity", intensity); Updadeで値(強さ)を送る
22          intensity = 0.0f; 送ったら0にして次のフレームでは0を送る
23      }
24
        0 個の参照
25      public void OnDrag(BaseEventData e)
26      {
27          RectTransform rt = transform as RectTransform;
28          Vector2 lp;
29          if (RectTransformUtility.ScreenPointToLocalPointInRectangle(rt, (e as PointerEventData).position, null, out lp))
30          {
31              Vector2 cood = lp / rt.rect.size + rt.pivot; ローカル座標の取得
32              matField.SetVector("_Position", new Vector4(cood.x, cood.y, 0, 0)); カーソルのローカル座標値を送る
33              intensity = 0.0001f;// 次の更新で確実に一回だけ値を設定する
34          }
            強さはひとまずマジックナンバー(よくない！)
35      }
36  }
```

スクリプトにおける他の処理

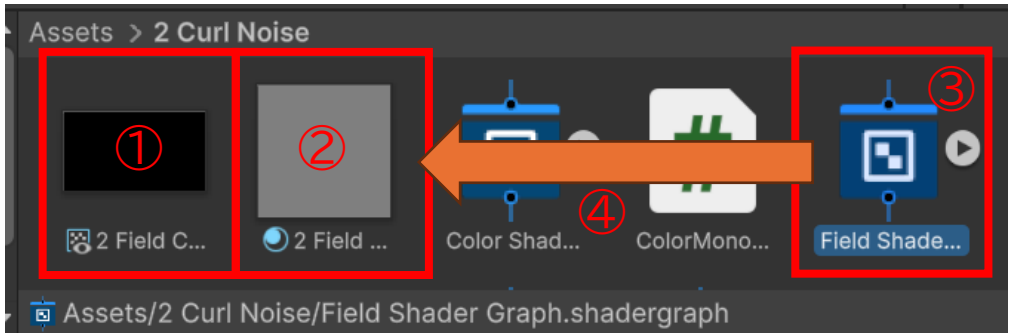
- 2つのカスタムレンダーテクスチャの初期化
  - 表示用と元となる場

ColorMonoBehaviourScript.cs

```
1   using UnityEngine;
2   using UnityEngine.EventSystems;
3   using UnityEngine.InputSystem;
4   using UnityEngine.UIElements;
5
        Unity スクリプト | 0 個の参照
6   public class ColorMonoBehaviourScript : MonoBehaviour
7   {
8       [SerializeField] CustomRenderTexture texField = default!;
9       [SerializeField] CustomRenderTexture texColor = default!;
10      [SerializeField] Material matField = default!;
11      float intensity = 0.0f;
12
13      // Start is called once before the first execution of Updat
            Unity メッセージ | 0 個の参照
14      void Start()
15      {
16          texField.Initialize();
17          texColor.Initialize();
18      }
```

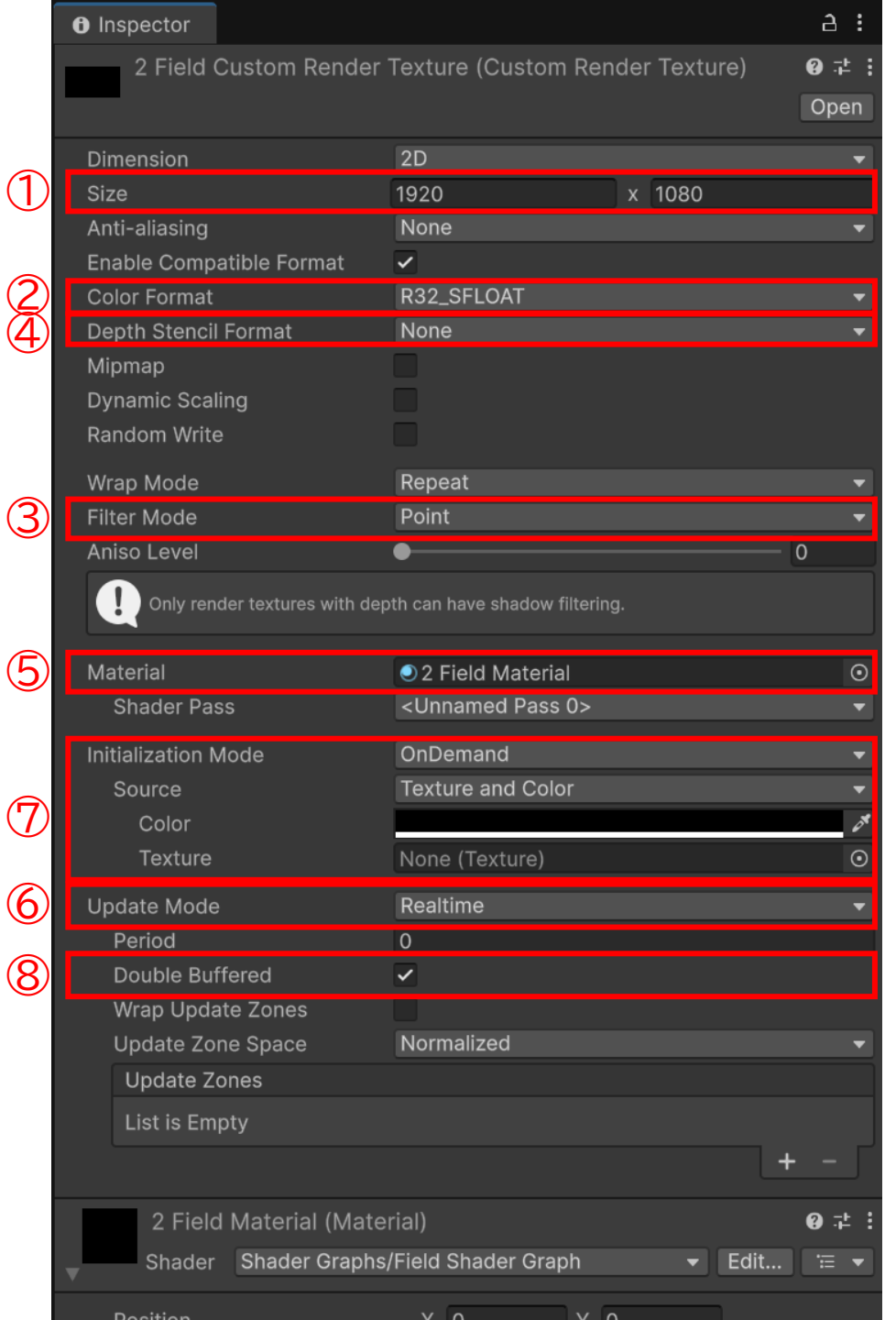
4. シェーダではパラメータに応じて場を作成する

- アセットの追加
  1. カスタムレンダーテクスチャ
     - 名称例：2 Field Custom Render Texture
  2. マテリアル
     - 名称例：2 Field Material
  3. シェーダグラフ(Custom Render Texture)
     - 名称例：Field Shader Graph
     4. 「2 Field Material」に設定



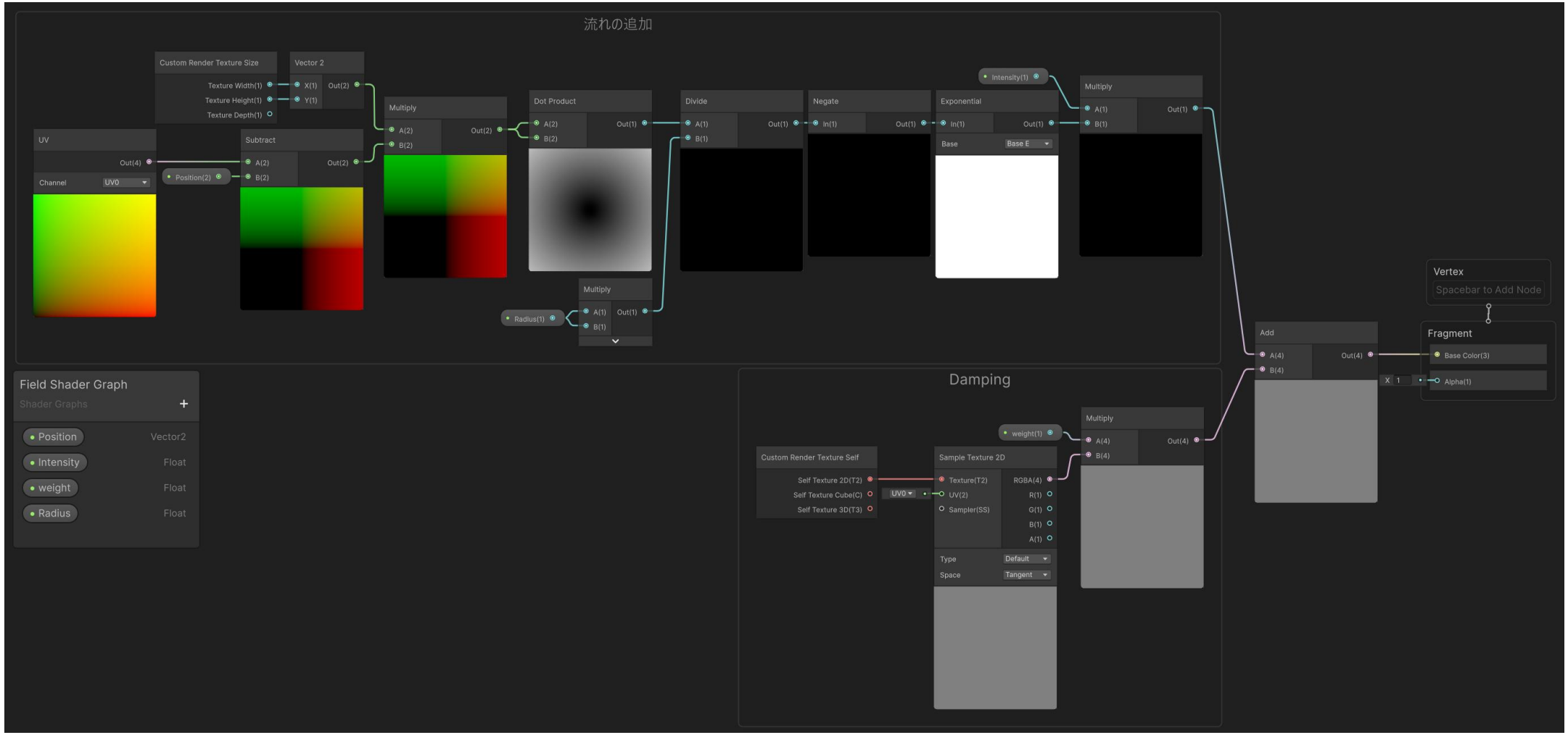
Custom Render Textureの設定

1. サイズ：フルHD（1920×1080）
   - アスペクト比が同じであれば、好みで良い
   - 小さいほど処理が軽いが、粗くなる
     - 時間があれば変更してみよう
2. フォーマット：R32_SFLOAT
   - 精度を高めるために浮動小数点数
3. サンプラーは「Point」しか使えない（HW的に）
4. 深度バッファは不要
5. 更新用のマテリアルを設定
6. Inspectorでのパラメータ調整を反映できるようにするために「Realtime」更新
7. 初期化はスクリプトで実行（OnDemend）
   - 色（黒）
8. ダブルバッファ



シェーダグラフ

Field Shader Graph

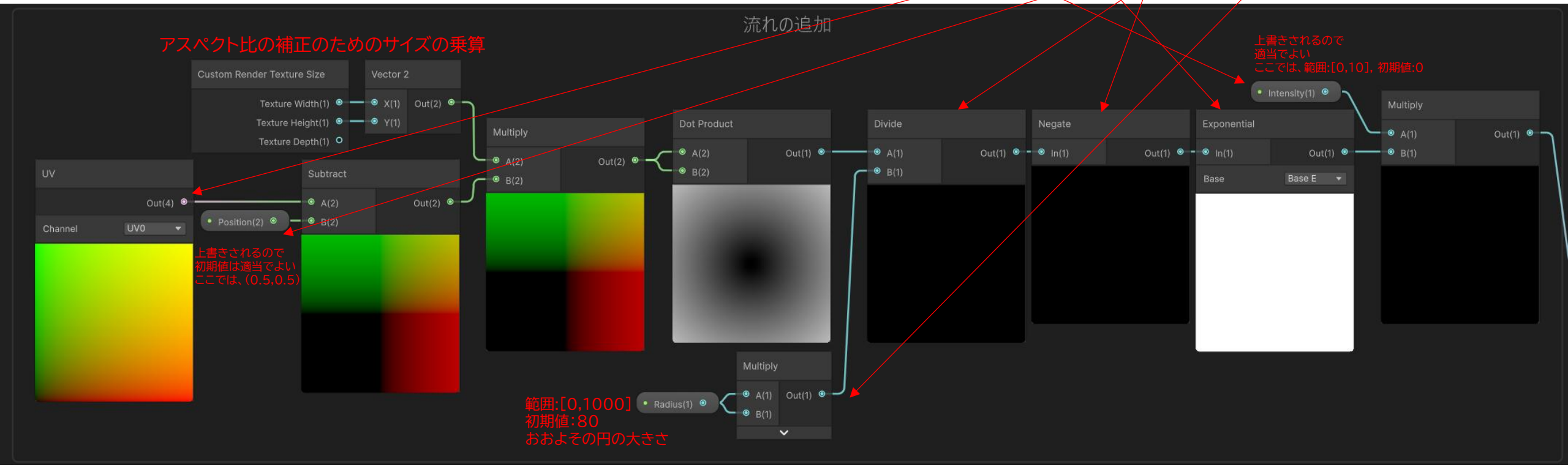


シェーダグラフ

Field Shader Graph

- なだらかな円としてガウス関数

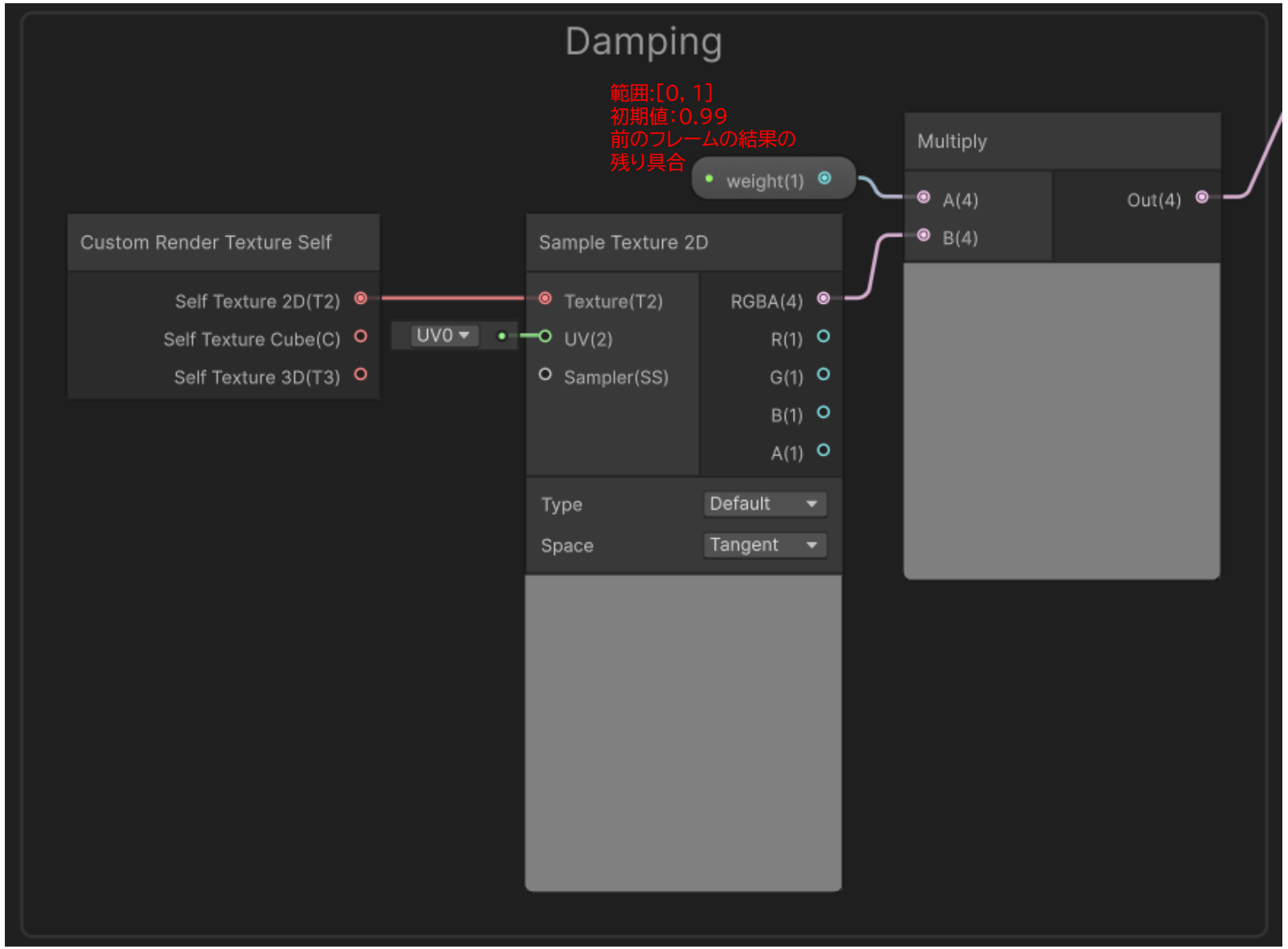
$$Intensity * \exp\left(-\frac{\|x - position\|^2}{Radius^2}\right)$$



シェーダグラフ

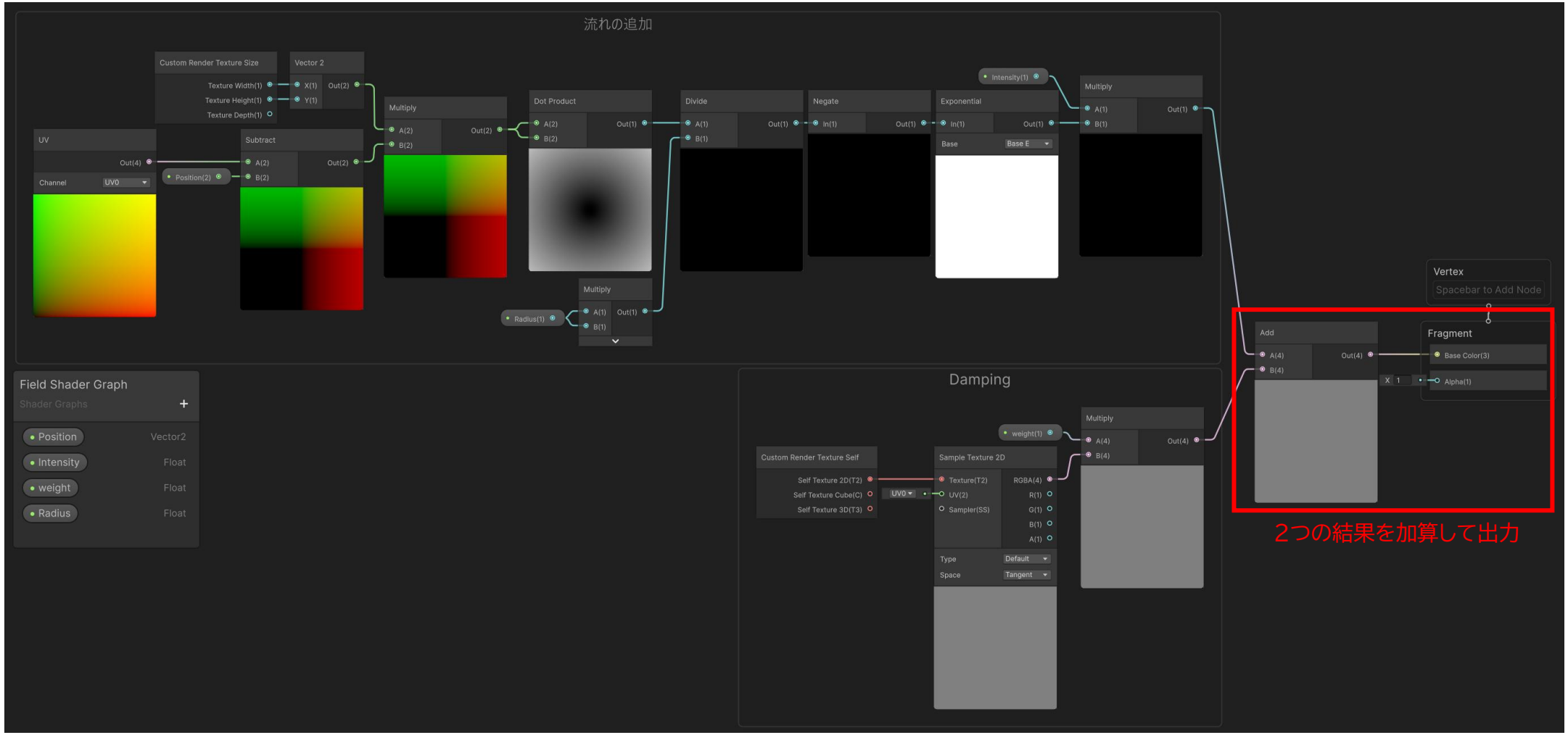
- 放置すると
  少しずつ減衰(止まる)

$$x \leftarrow 0.99x$$



シェーダグラフ

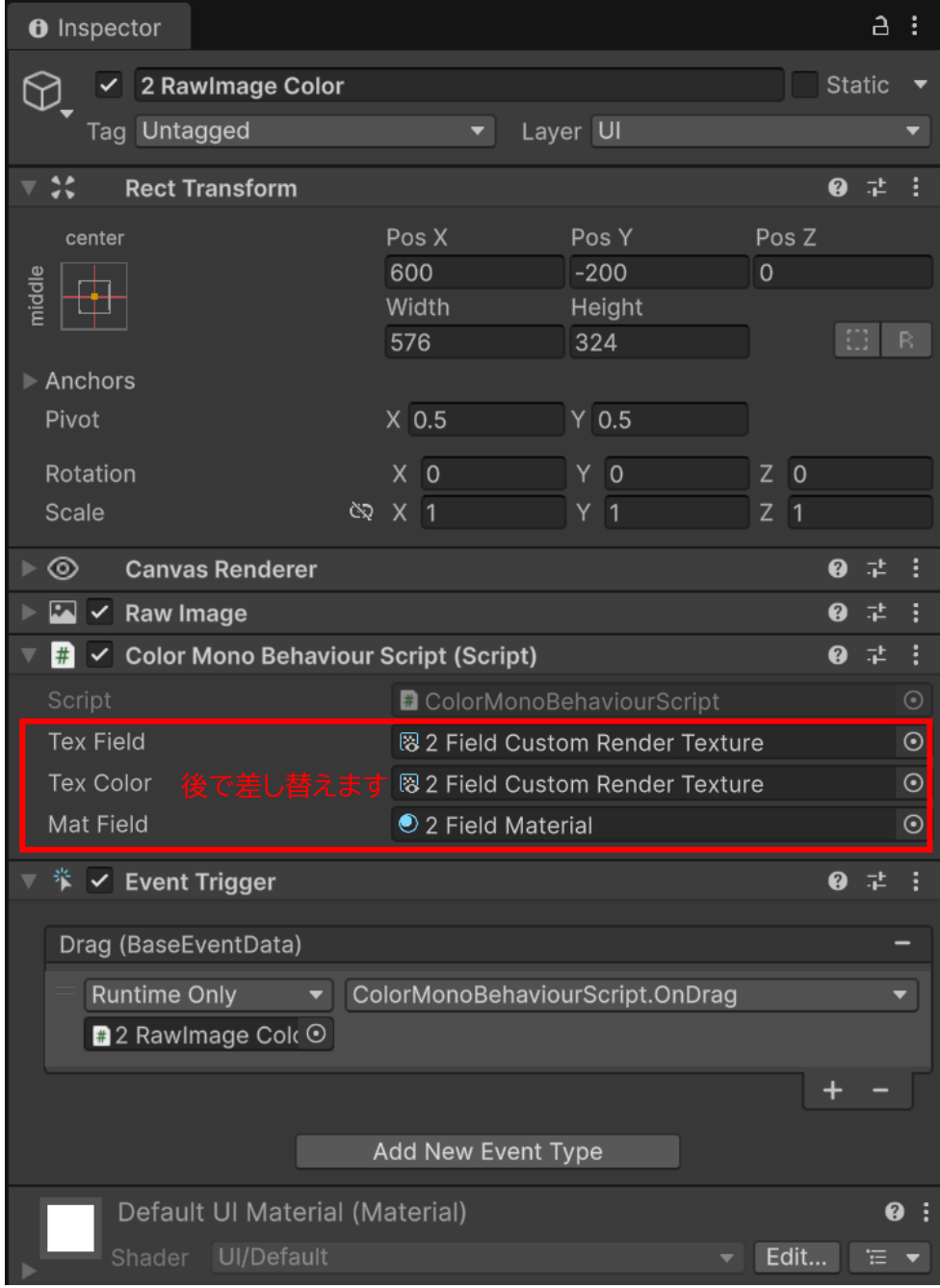
Field Shader Graph



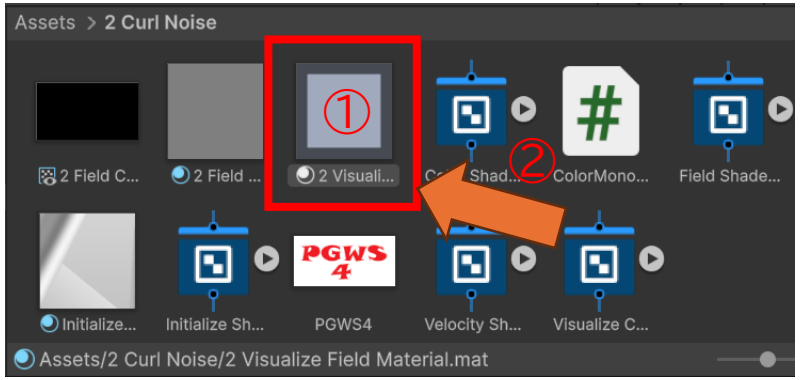
2つの結果を加算して出力

スクリプトのメンバー設定

- レンダーターゲットの初期化とシェーダの値の設定としてマテリアルを設定
  - テクスチャは共に「2 Field Custom Render Texture」
    - 片方(Tex Color)は違うが、設定しないとエラーになるので、仮のものとして設定し、後で差し替える

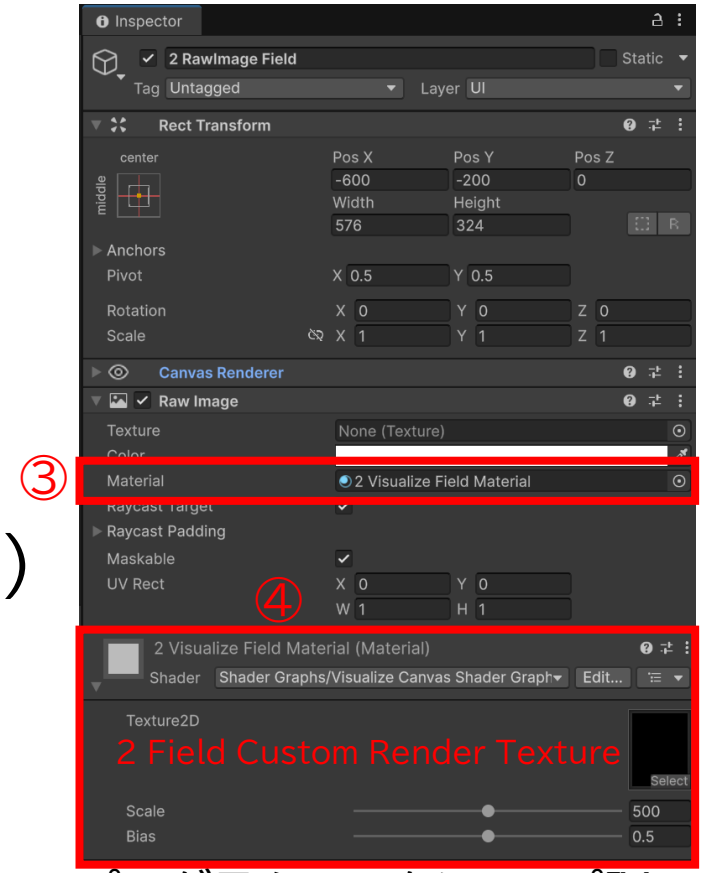


可視化



値が小さくなるので、強調する可視化のシェーダを通す

1. マテリアルの追加
   - 名称例:「2 Visualize Field Material」
2. シェーダの登録
   - 「2 Visualize Field Material」に「Visualize Canvas Shader Graph」を登録
3. オブジェクトのマテリアルに設定
   - 「2 RawImage Field」の「Raw Image」-「Material」に「2 Visualize Field Material」を設定
4. パラメータの調整(2 Visualize Field Material)
   - Texture 2D:「2 Field Custom Render Texture」
   - Scale: 500 など
   - Bias: 0.5

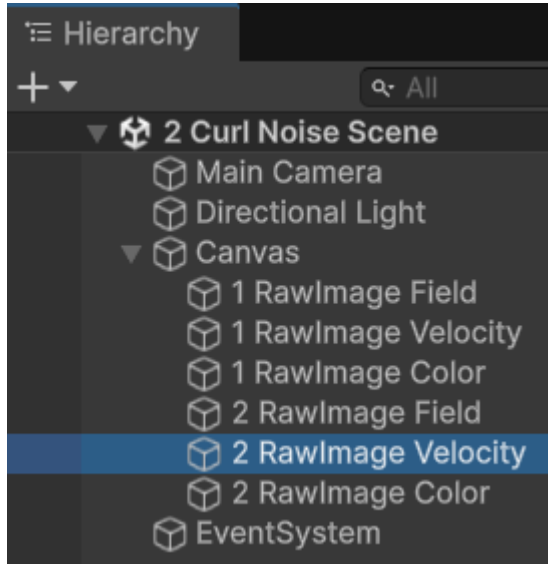


やってみよう
ステップ4:ドラッグで初期化

- ぐりぐりして動作を見てみよう
  - まだ右下の模様の流れは実装していません

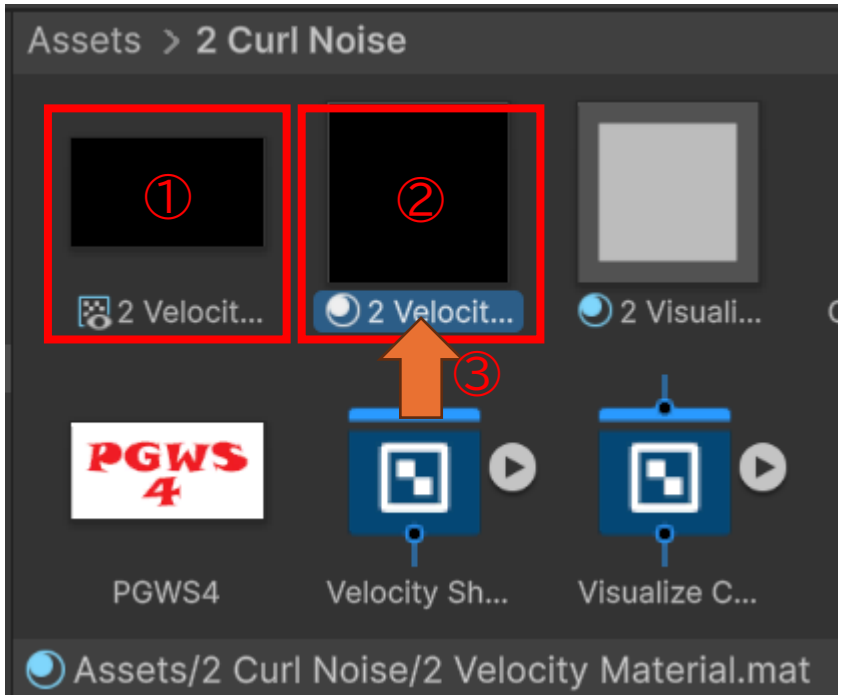
ステップ5：流れ場の生成

- ノイズによる場をCurlする
  - 「2 RawImage Velocity」で可視化する
    - 「UI/Raw Image」オブジェクト



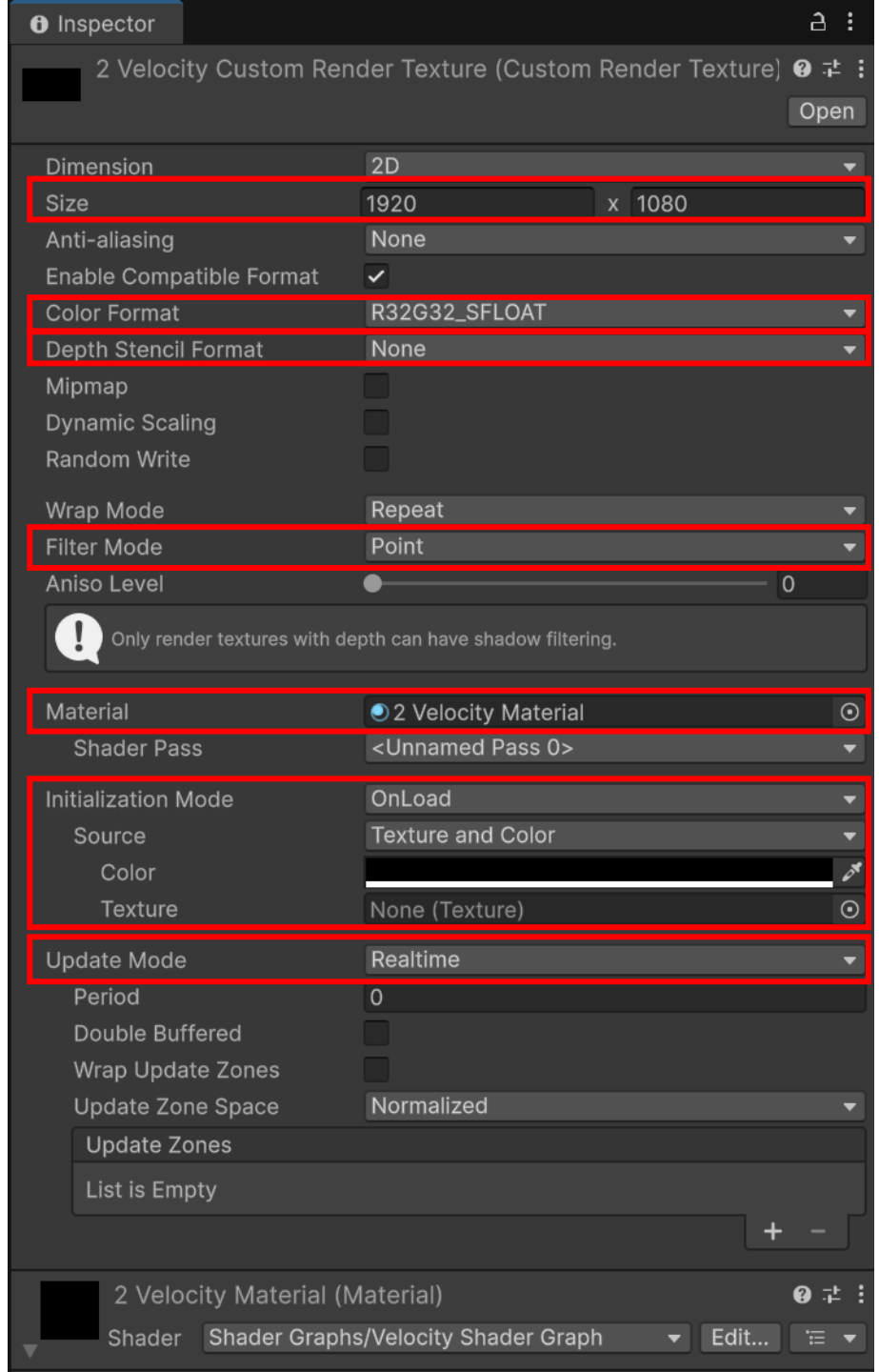
アセットの追加

1. カスタムレンダーテクスチャ
   - 名称例：2 Velocity Custom Render Texture
2. マテリアル
   - 名称例：2 Velocity Material
3. シェーダグラフをマテリアルに設定
   - 「Velocity Shader Graph」を「2 Velocity Material」に



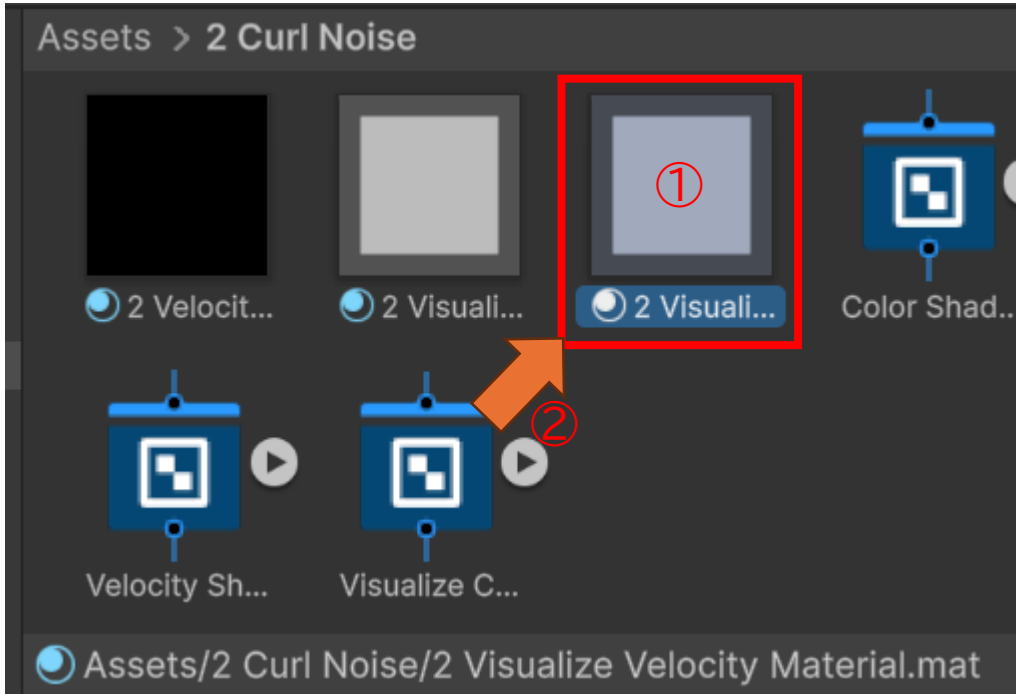
Custom Render Textureの設定

1. サイズ：フルHD(1920×1080)
   - Initialize Shader Graphと合わせる
     - 合わせなくても動くが、情報の無駄が少ない
2. フォーマット：R32G32_SFLOAT
   - 負の値も使うので符号付き
   - 精度を高めるために浮動小数点数
     3. サンプラーは「Point」しか使えない(HW的に)
4. 深度バッファは不要
5. 更新用のマテリアルを設定
   6. Inspectorでのパラメータ調整を反映できるようにするために「Realtime」更新
7. 今回は初期化は適当で良い



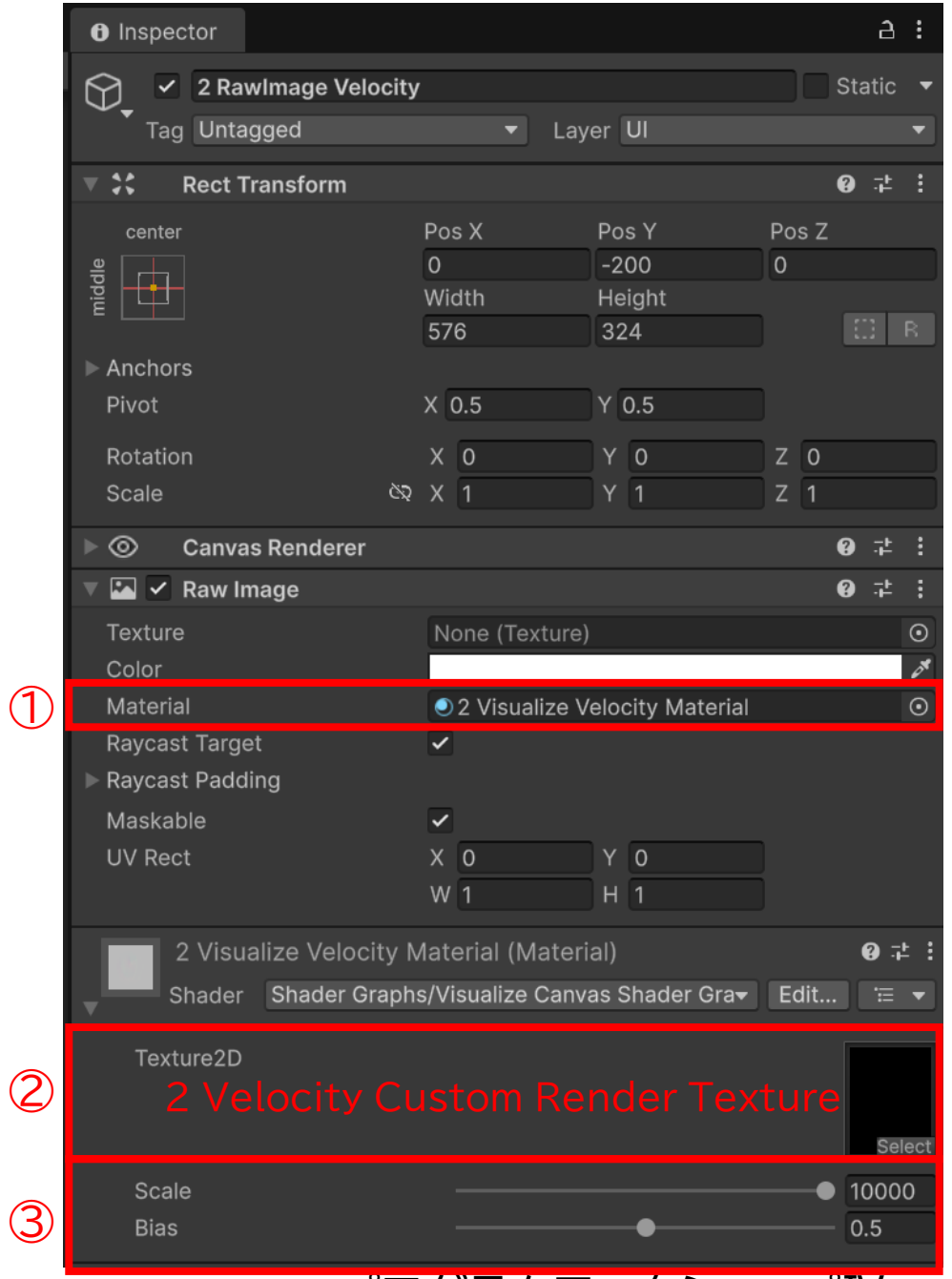
可視化

- 値が小さく、負の値を含むため、そのままの表示では見難い
- 可視化用のマテリアルを追加
  1. マテリアルを追加
     - 名称例：2 Visualize Velocity Material
  2. シェーダグラフをマテリアルに設定
     - Visualize Canvas Shader Graph を 2 Visualize Velocity Materialに設定



オブジェクトの設定

1. 描画にマテリアルを設定
2. マテリアルでテクスチャを設定
    3. 他のパラメータは、結果が見やすいように

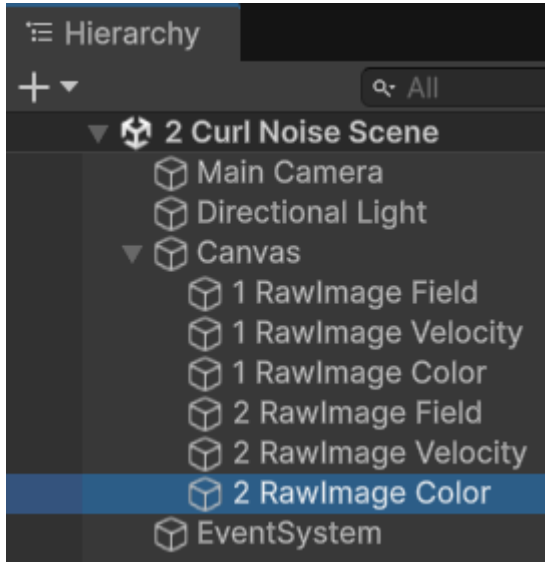


やってみよう
ステップ5：流れ場の生成

- ぐりぐりして動作を見てみよう
  - まだ右下の模様の流れは実装していません

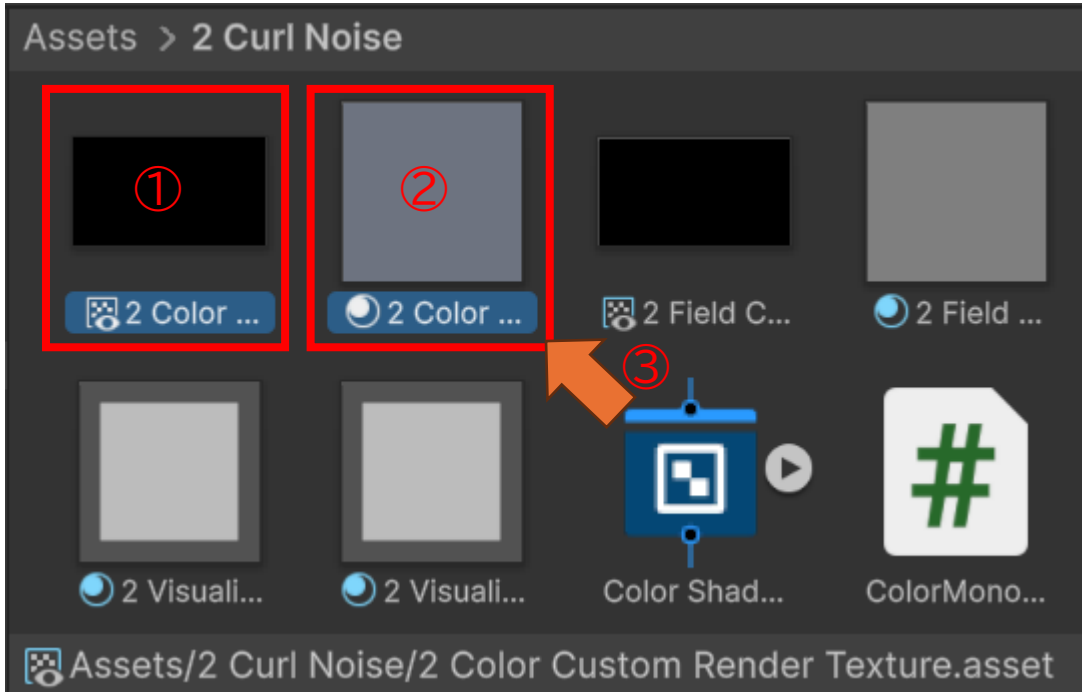
ステップ6：模様を流す

- 流れ場で模様を動かす：速度の量だけ反対向きの位置を読み込む
  - 「2 RawImage Color」で可視化する
    - 「UI/Raw Image」オブジェクト



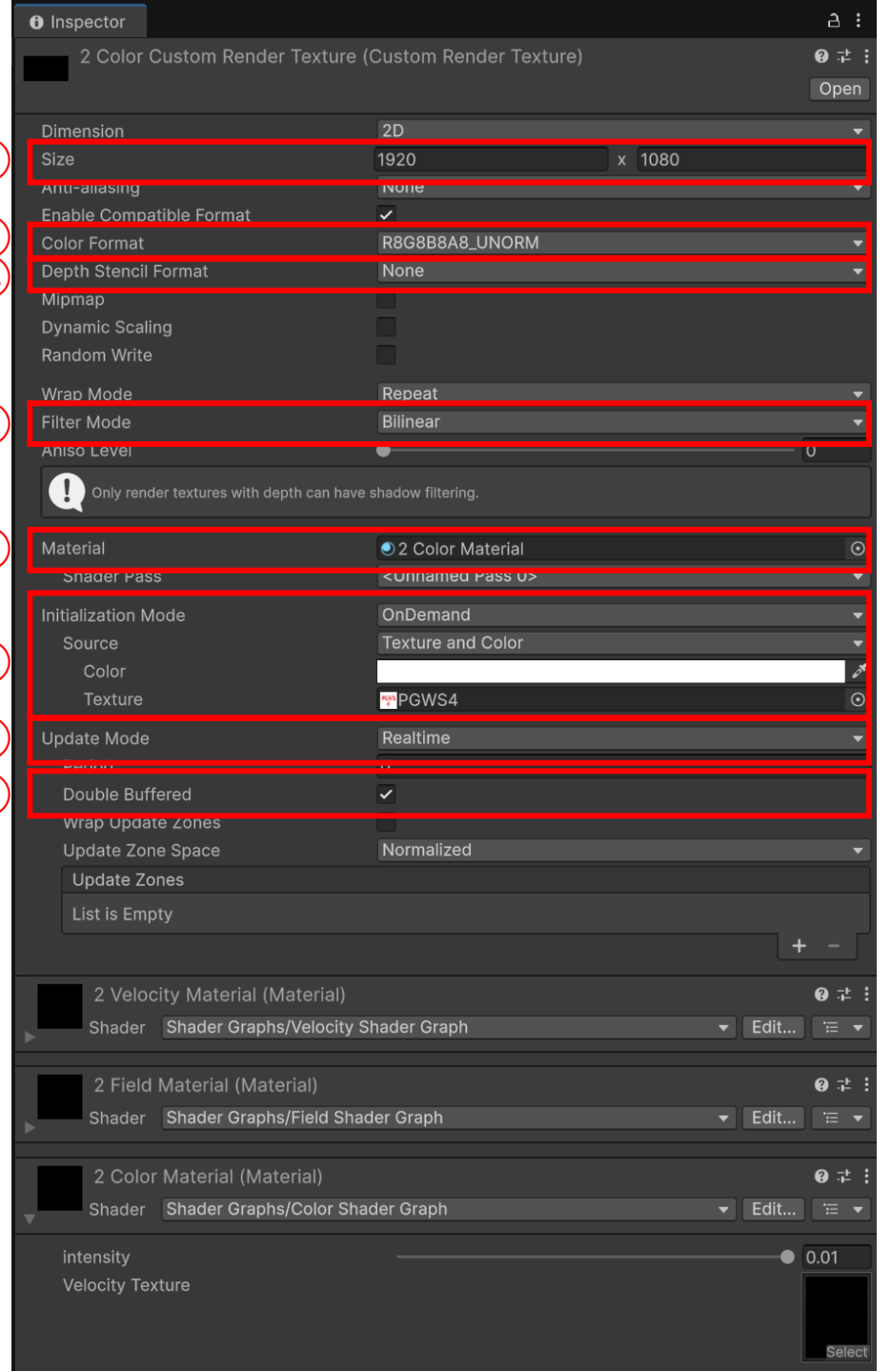
アセットの追加

1. カスタムレンダーテクスチャ
   - 名称例：2 Color Custom Render Texture
2. マテリアル
   - 名称例：2 Color Material
3. Color Shader Graphを
   「2 Color Material」に設定



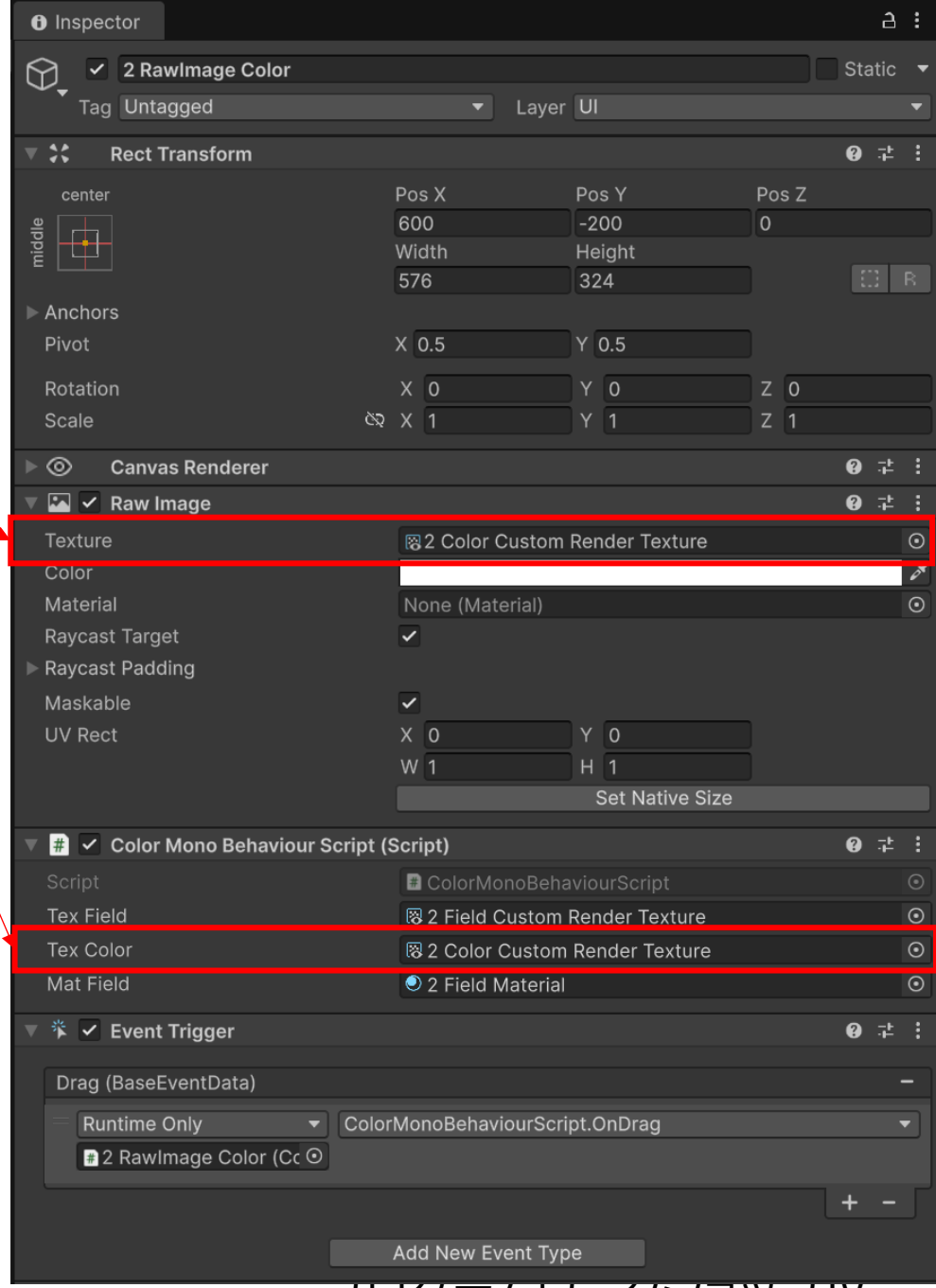
Custom Render Textureの設定

1. サイズ：フルHD(1920×1080)
   - 1 Velocity Custom Render Textureと合わせる
     - 合わせなくても動くが、情報の無駄が少ない
2. フォーマット：R8G8B8A8_UNORM
   - 表示されるものなので、高い精度は不要
   3. サンプラーは「Bilinear」が使える
4. 深度バッファは不要
5. 更新用のマテリアルを設定
   6. Inspectorでのパラメータ調整を反映できるようにするために「Realtime」更新
7. 初期化はスクリプトで実行(OnDemend)
   - 色(白)とテクスチャの乗算
8. ダブルバッファ

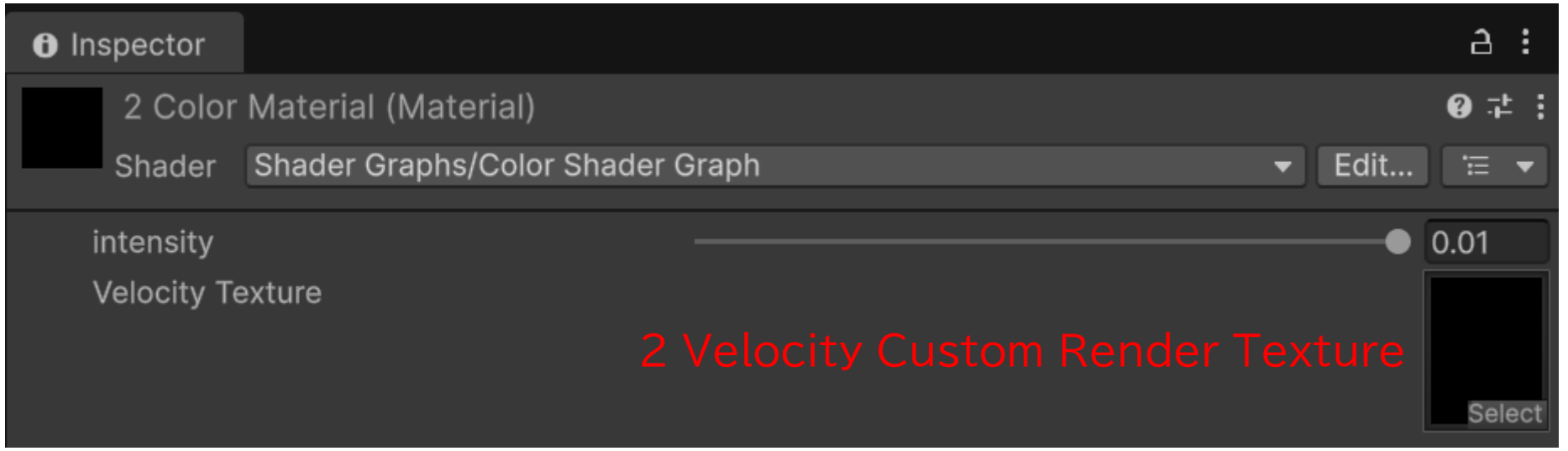


オブジェクトの設定

- オブジェクトのテクスチャに
  カスタムレンダーテクスチャを設定
- 更新用のスクリプトの変更
  - 「Tex Color」プロパティを「2 Color Custom Render Texture」に差し替え



マテリアルの設定



移流の強さも
いい感じに調整する

やってみよう
ステップ6：模様を流す

- Inspectorで値(intensityなど)を変更してみよう
- 初期化用のテクスチャを
  自分の好きなものに
  変えてみよう

まとめ

- カスタムレンダーテクスチャ
  - カスタムレンダーテクスチャの概要
  - ライフゲーム
    - GPU内で動的にテクスチャを更新する
  - カールノイズ
    - 流体のような表現を行う
    - 複数のカスタムレンダーテクスチャを連携させる