Unityはじめるよ

～よく人に質問されるTIPSと冬休みに勉強した技術紹介～

統合開発環境を内蔵したゲームエンジン
http://japan.unity3d.com/

※いろんな職業の方が見る資料なので説明を簡単にしてある部分があります。正確には本来の意味と違いますが上記理由のためです。ご了承ください。
この資料内の一部の画像、一部の文章はUnity公式サイトから引用しています。

© monolizm LLC

本日の内容

・よく質問されることとTIPS

・冬休みに勉強した技術

© monolizm LLC

よく質問されることとTIPS

・ポストプロセッシングスタックの使い方
・クリックした位置にUIを移動させたい
・FPS指定方法
・画面分割
・プレハブについて

© monolizm LLC

ポストプロセッシングスタックの使い方

© monolizm LLC

概要

以前はグラフィックのクオリティを上げる表現
・Bloom（強い光源から光が滲み出すような表現）
・Depth of Field（カメラのぼかし表現）
・SSAO（モデルの凹凸の奥まった部分位影をつける）
などなどは、 ImageEffectとして提供されていました。

現在は、
ポストプロセッシングスタックという名称で、
アセットストアやGithubから手に入れることができます。
※Githubの方が最新

© monolizm LLC

ポストプロセッシングスタックの使い方

OFF

ON

© monolizm LLC

何が変わった？

ImageEffectの頃から何が変わったかというと、
各機能ごとバラバラに提供されていたものが、
一元管理されるようになり、
速度面、画質面とも良くなったということです。

バージョンによって使い方がコロコロ変わるので、
とりあえず現時点での最新バージョンでの使い方を説明します。

© monolizm LLC

基本的な使い方

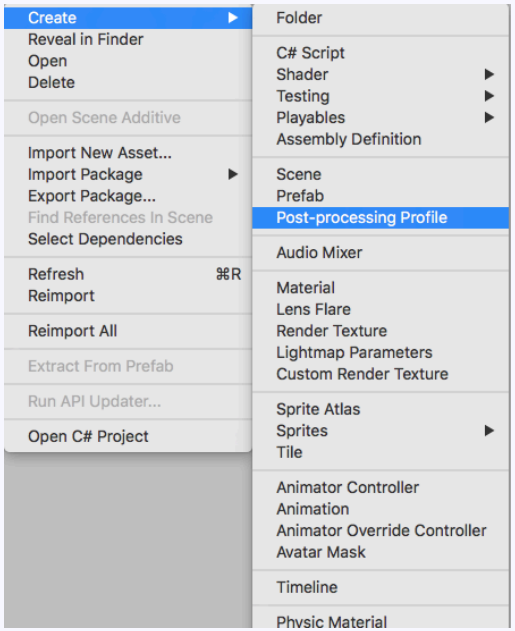
概念的には、
ポストプロセスの処理内容を設定したプロファイルを作って、
ポストプロセスのスクリプトにセットすることで利用します。

まずは、
https://github.com/Unity-Technologies/PostProcessing
から最新版をDL・解凍し、
中にあるPostProcessingフォルダをプロジェクトにインポートします。

© monolizm LLC

手順１

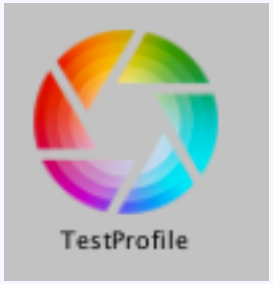
プロジェクトビューで右クリックして、
Create > Post-processing Profile
でプロファイルを作成します。

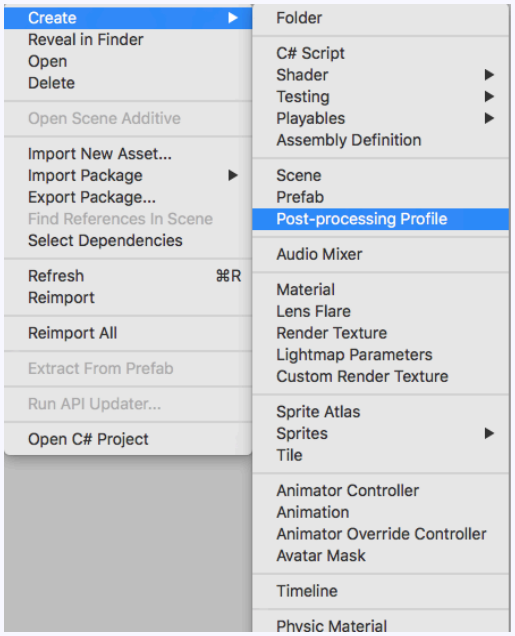


© monolizm LLC

手順２

プロジェクトビューで右クリックして、
Create > Post-processing Profile
でプロファイルを作成します。



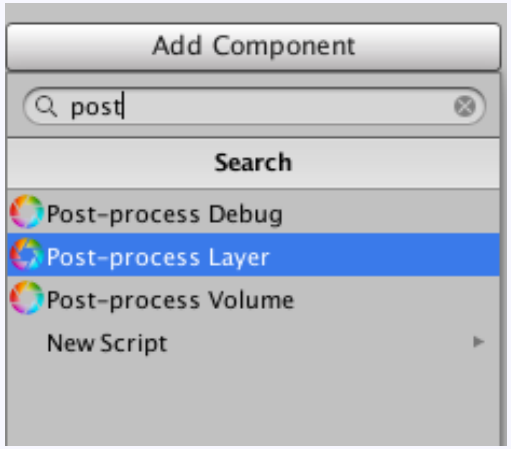


こんなファイルができればOK。

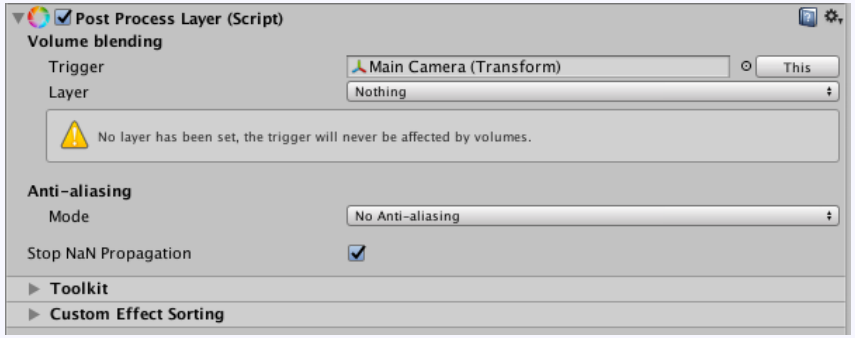
© monolizm LLC

手順３

ヒエラルキーで**MainCamera**を選んで
インスペクターの**AddComponent**ボタンを押し、
検索窓にpostと打つと、名前にpostが付く
コンポーネントが出てきます。
その中から**Post-processLayer**を選択。



Post-processLayerは、
どのレイヤーを対象とするか、アンチエイリアス処理を行うか、
などを設定するコンポーネントです。
今回はDefaultを選んでおきます。

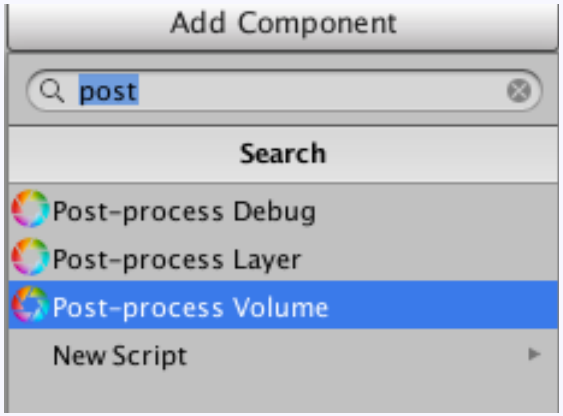


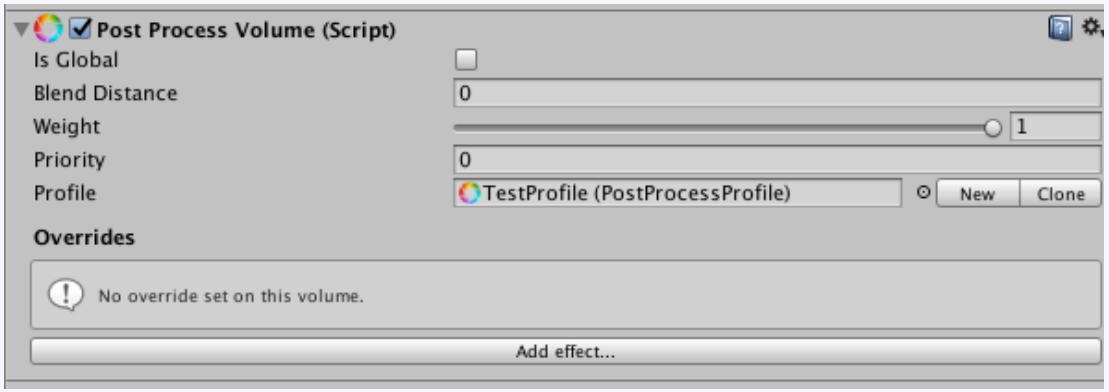
© monolizm LLC

手順４

もう一度**AddComponent**ボタンを押し、
Post-processVolumeを選択。

Post-processVolumeの**Profile**に、
先ほど作ったプロファイルをセットする。
isGloabalのチェックを入れておきます。

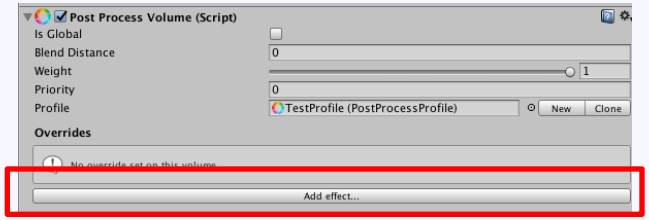


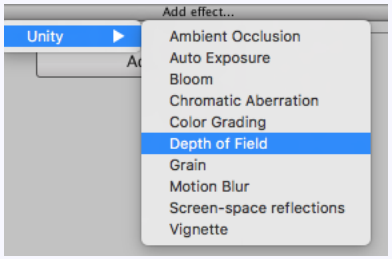


© monolizm LLC

手順５

Post-processVolumeの**Add effect**ボタンを押して、
利用したいエフェクトを追加します。





試しにDepth of Fieldを追加してみましょう。

© monolizm LLC

手順６

Depth of Fieldと書かれている部分をクリックすると、
隠れていたプロパティが表示されます。

設定したい項目の●をクリックすると値を変更をできます。

Depth of Fieldの場合、

Focus Distance：ピントを合わせたい位置(カメラからの距離mを指定)
Aperture：レンズの明るさ(数字が小さいほど明るくなりボケやすい)
FocalLength：焦点距離(数字が大きほどボケやすい)
MaxBlurSize：ボケの強さ

となります。

© monolizm LLC

ポストプロセッシングスタックを使えば、
手軽に画面のクオリティを上げることができます。

AntiAlias、Bloom、Depth of Field、Ambient Occlusion、
Color Gradingは、特に使いたくなるエフェクトです。
アクションやレースゲームならMotion Blurも効果的。

ただ、ポストプロセス全般的に GPU負荷が大きいので注意。
イベントシーンだけ使うとか、使い所は要検討してください。

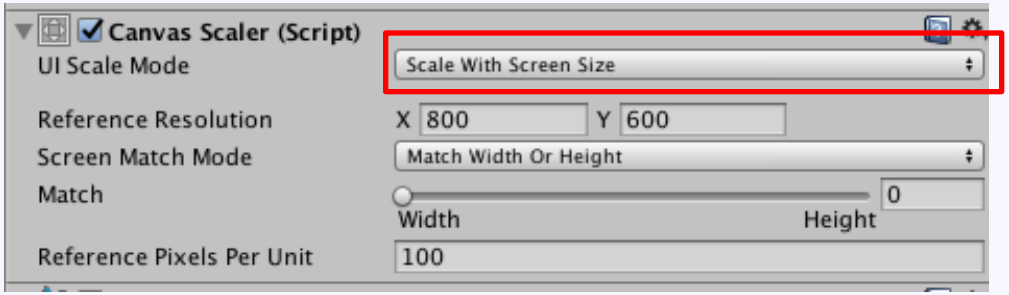
© monolizm LLC

クリックした位置にUIを移動させたい

※スクリプトから狙った位置にUIを移動させたい

© monolizm LLC

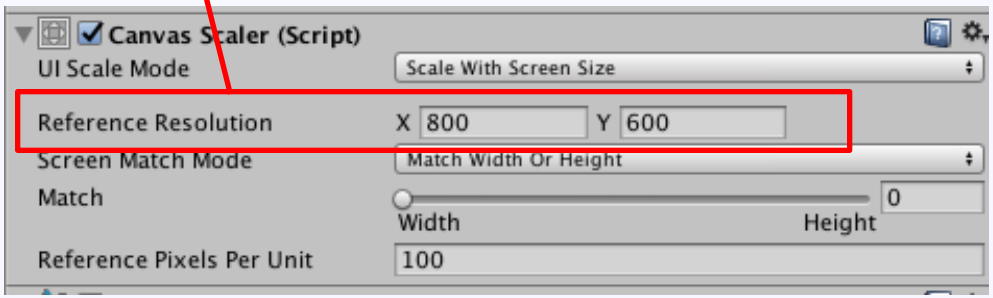
概要



今回は、
Canvasの**CanvasScaler**コンポーネントの
UIScaleModeが**ScaleWithScreenSize**の場合、かつ、
動かしたい画像のアンカーを中心とした場合の話です。

© monolizm LLC

ScaleWithScreenSizeの場合、
実行環境のディスプレイ解像度に関係なく、
仮想スクリーンサイズの座標系となります。



ScreenMatchModeで、
仮想スクリーンサイズとアスペクト比の異なるディスプレイに
対応する方法を選ぶ。

© monolizm LLC

何が難しく感じるのかというと、

座標を動かせる変数が複数あるので、
「どの値を変更すれば良いのかわからない」
ということ。

マウスの座標系とUIの座標系が異なるため、
「座標変換を行わなければならない」
ということ。

© monolizm LLC

UIの座標をスクリプトから操作するなら、

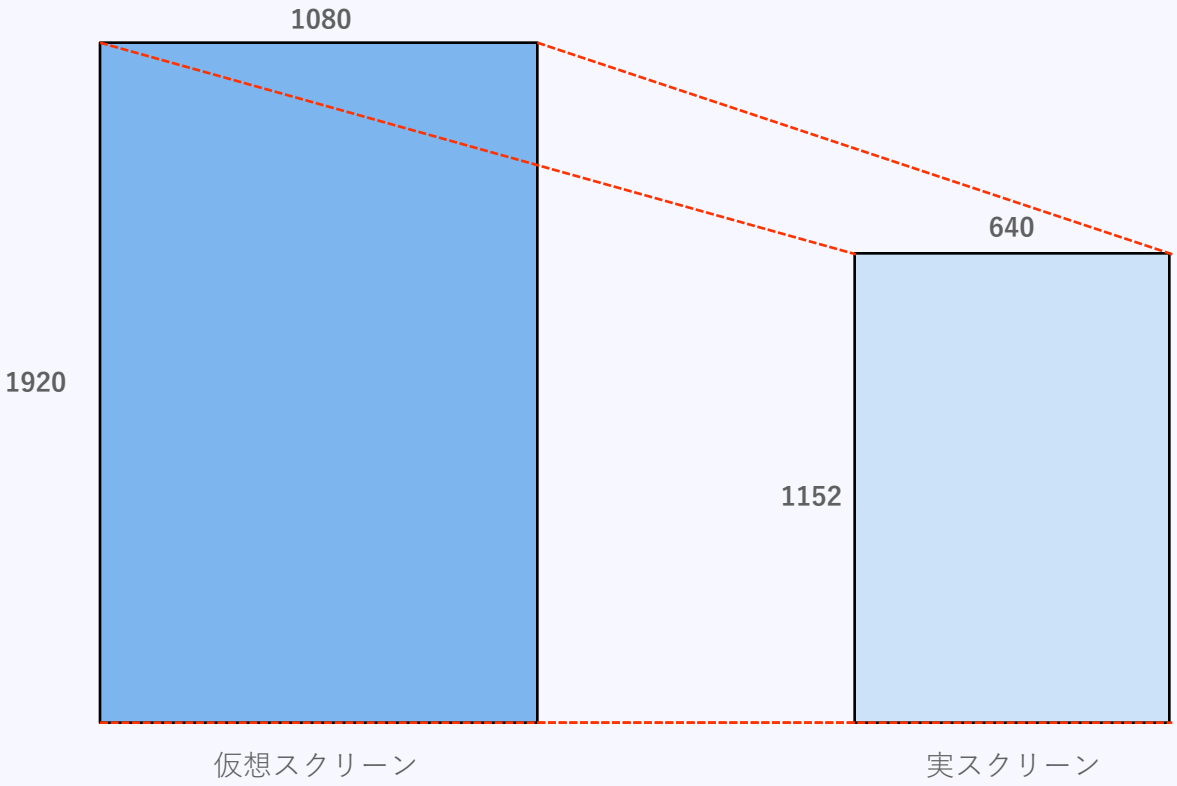
RectTransformのanchoredPosition

をいじるのがわかりやすいと感じます。

© monolizm LLC

手順１

実スクリーンと仮想スクリーンのサイズの違いをスケール値として出す。



スケール値

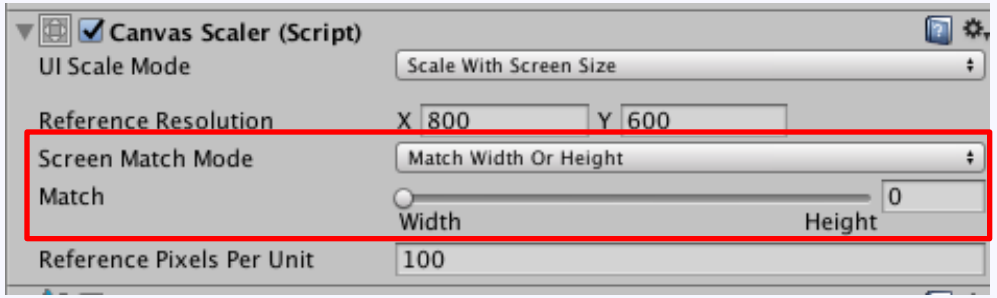
1080 / 640 = 1.6875

1920 / 1152 = 1.6667

© monolizm LLC

手順２

画面のアスペクト比対策、ScreenMatchModeの値に合わせた値の計算。



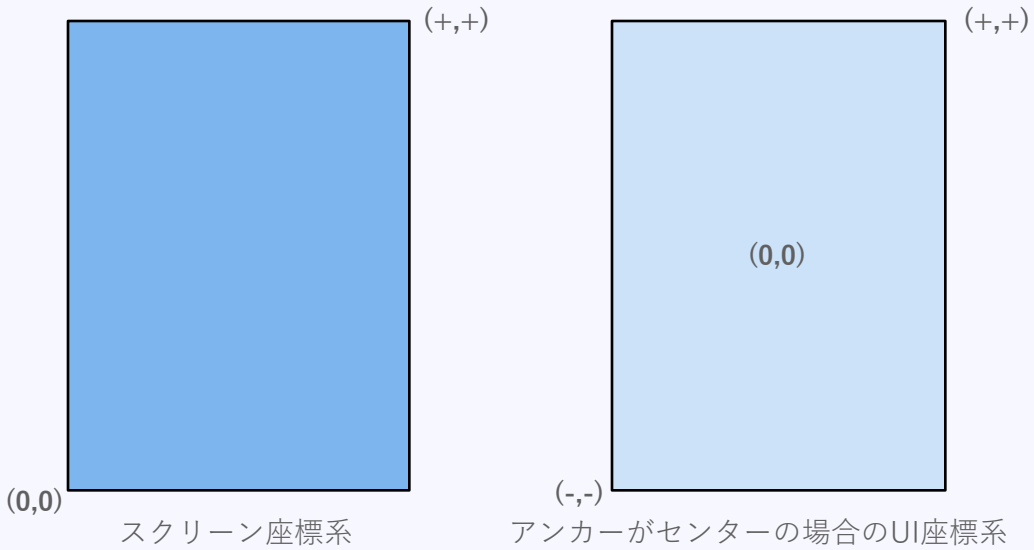
Matchのスライダーは、Widthが0、Heightが1となる。
アスペクト値を求めるのは下記の計算

アスペクト値.x = Mathf.Lerp(1.0f, スケール値.y / スケール値.x, Match);
アスペクト値.y = Mathf.Lerp(スケール値.x / スケール値.y, 1.0f, Match);

© monolizm LLC

手順３

スクリーン座標の座標系から、オフセット値を求める。
スクリーン座標は左下が原点で、右上方向に＋となる。



スクリーン座標系　　アンカーがセンターの場合のUI座標系

座標系を合わせるために、どれだけ原点がずれているかのオフセットを求める

オフセット値.x = -仮想スクリーンサイズ.x * 0.5f;
オフセット値.y = -仮想スクリーンサイズ.y * 0.5f;

© monolizm LLC

手順４

手順３までに求めた値を使って座標変換する。

変換後.x = (変換したい値.x * スケール値.x + オフセット値.x) * アスペクト値.x;
変換後.y = (変換したい値.y * スケール値.y + オフセット値.y) * アスペクト値.y;

© monolizm LLC

ソース

クリックした位置にUIを移動させたい

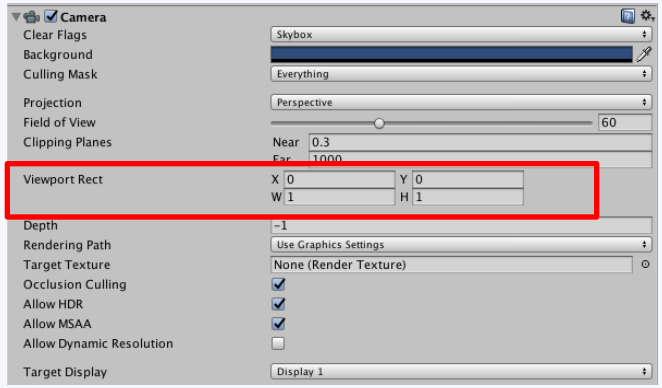
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;


/// <summary>
/// クリックした位置にImageを移動させるスクリプト.
/// Imageにアタッチして使用する.
/// </summary>
public class ImageController : MonoBehaviour
{

    [SerializeField]
    GameObject m_canvas;                // Canvasを受け取っておく

    [SerializeField]
    Text m_debugText;                   // デバッグ情報表示用TEXT

    Image m_image;                      // 移動させる画像

    // 1パターン目の方法で利用
    CanvasScaler m_canvasScaler;        // CanvasScaler
    Vector2 m_scale;                    // 画面解像度と仮想スクリーンサイズの拡大縮小率
    Vector2 m_aspect;                   // 画面のアスペクト比対策
    Vector2 m_offset;                   // マウスの座標系は左下が(0,0)なのでオフセット用の変数を用意しておく


    // Use this for initialization
    void Start ()
    {
        // Imageコンポーネントを取得
        m_image = GetComponent<Image>();


        // CanvasScalerを取得
        m_canvasScaler = m_canvas.GetComponent<CanvasScaler>();



        // 画面解像度と仮想スクリーンサイズの拡大縮小率を求める
        m_scale.x = m_canvasScaler.referenceResolution.x / Screen.width;
        m_scale.y = m_canvasScaler.referenceResolution.y / Screen.height;


        // 画面のアスペクト比対策
        m_aspect.x = Mathf.Lerp(1.0f, m_scale.y / m_scale.x, m_canvasScaler.matchWidthOrHeight);
        m_aspect.y = Mathf.Lerp(m_scale.x / m_scale.y, 1.0f, m_canvasScaler.matchWidthOrHeight);


        // マウスの座標系は左下が(0,0)なのでオフセットを求める
        m_offset.x = -m_canvasScaler.referenceResolution.x * 0.5f;
        m_offset.y = -m_canvasScaler.referenceResolution.y * 0.5f;
    }


    // Update is called once per frame
    void Update ()
    {
        // マウスの左ボタンを押したら
        if (Input.GetMouseButtonDown(0))
        {
            // 求めた座標に移動
            m_image.rectTransform.anchoredPosition = ConvertScreenToUICoordinate(Input.mousePosition);

            // デバッグ表示
            m_debugText.text =
            "マウス座標( " + Input.mousePosition.x + ", " + Input.mousePosition.y + " )" + "¥n" +
               "UI座標( " + m_image.rectTransform.anchoredPosition.x + ", " + m_image.rectTransform.anchoredPosition.y + " )";

        }
    }


    /// <summary>
    /// スクリーン座標をUI座標に変換する.
    /// 条件.UIScaleModeが ScaleWithScreenSize であること。
    /// </summary>
    /// <returns>The screen to UI Coordinate.</returns>
    /// <param name="pos">Position.</param>
    Vector2 ConvertScreenToUICoordinate(Vector2 pos)
    {
        // 座標を求める
        Vector2 ret;
        ret.x = (pos.x * m_scale.x + m_offset.x) * m_aspect.x;
        ret.y = (pos.y * m_scale.y + m_offset.y) * m_aspect.y;

        return ret;
    }

}
```

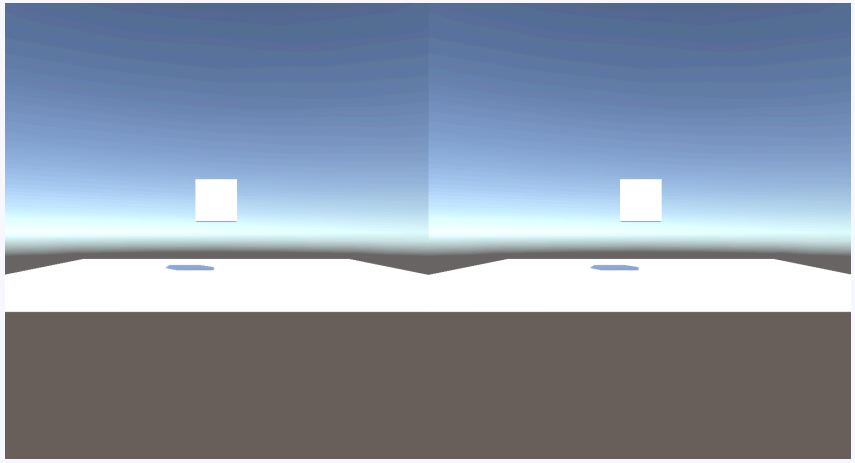
© monolizm LLC

画面分割

© monolizm LLC

カメラを２つ用意してそれぞれの**ViewportRect**を設定する。
左下が原点の正規化(0~1)された座標。





左右分割なら、
　カメラ１　X=0, Y=0, W=0.5, H=1
　カメラ２　X=0.5, Y=0, W=0.5, H=1
上下分割なら
　カメラ１　X=0, Y=0.5, W=1, H=0.5
　カメラ２　X=0, Y=0, W=1, H=0.5

© monolizm LLC

FPS指定

© monolizm LLC

FPS(フレーム/秒)を指定する方法。
モバイル環境だと、デフォルトは30FPSとなっている。
※電池節約のため

スクリプトから、

Application.targetFrameRate = 60;

と設定すれば変更可能。

注意点
QualitySettingsの**vSyncCount**が**Don't Sync**でないと、
targetFrameRateの設定が有効にならない。
スクリプトからは
QualitySettings.vSyncCount = 0;
でDon't Syncにできる。

© monolizm LLC

プレハブについて

© monolizm LLC

同じオブジェクトを大量に作る時に大変役に立つプレハブ。
プレハブを元にしたオブジェクトは、大元のプレハブを変更すれば
全てのオブジェクトに変更が反映される便利なもの。

しかし、

プレハブの中にプレハブを置いた場合（ネスト化）、
中のプレハブは大元の変更の影響を受けなくなる

ので注意が必要。

解決するアセットもあるようです。

© monolizm LLC

ミップマップについて

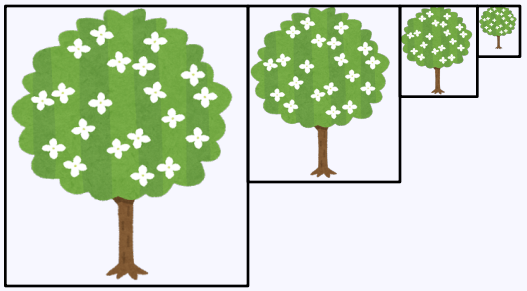
© monolizm LLC

ミップマップとは、

カメラとの距離に応じて参照するテクスチャを切り替える技術

のことです。

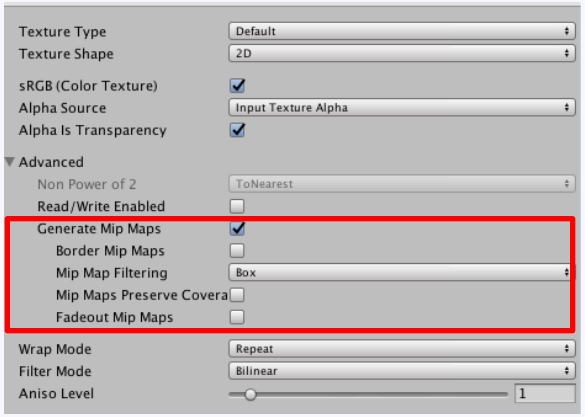
解像度の違うテクスチャを用意し、
遠くに行くにつれて解像度の低いテクスチャに切り替える。
効果は、遠くのテクスチャが綺麗になじむ、処理負荷軽減など。



ミップマップのイメージ

© monolizm LLC

Unityの場合、
テクスチャのインポート設定で自動でミップマップを
生成することができます。

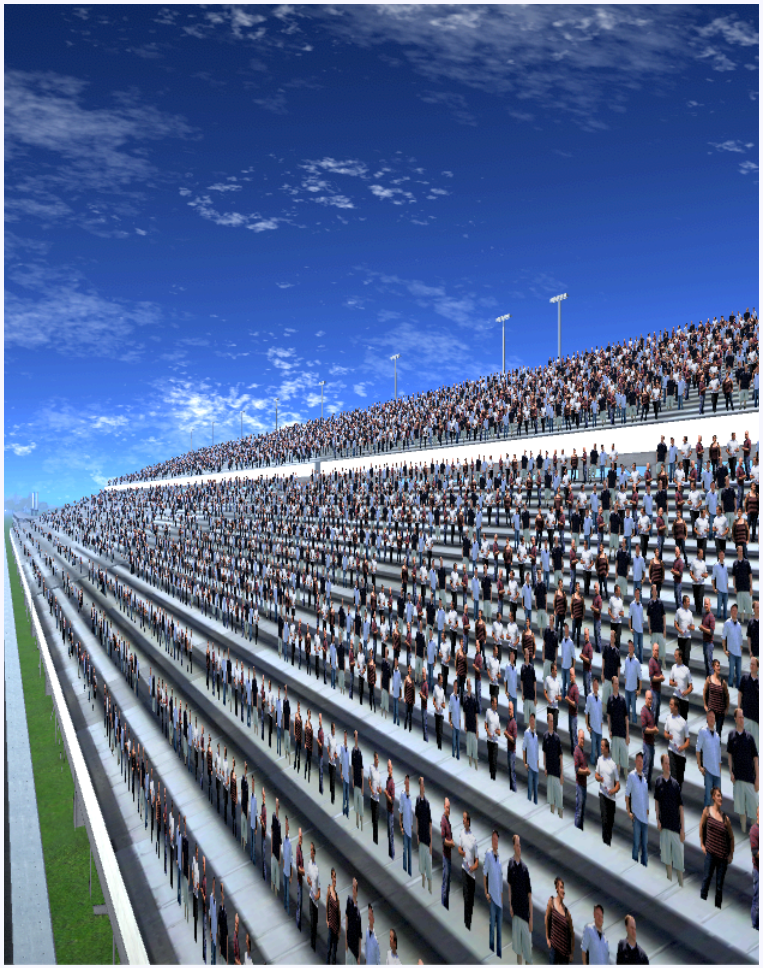


私の場合、モバイル環境向けの開発が多いため、
メモリや容量を食わないようミップマップを生成しないことが
多かったのですが、テクスチャが細かい図柄だと、
思った以上に効果があることを知りました。

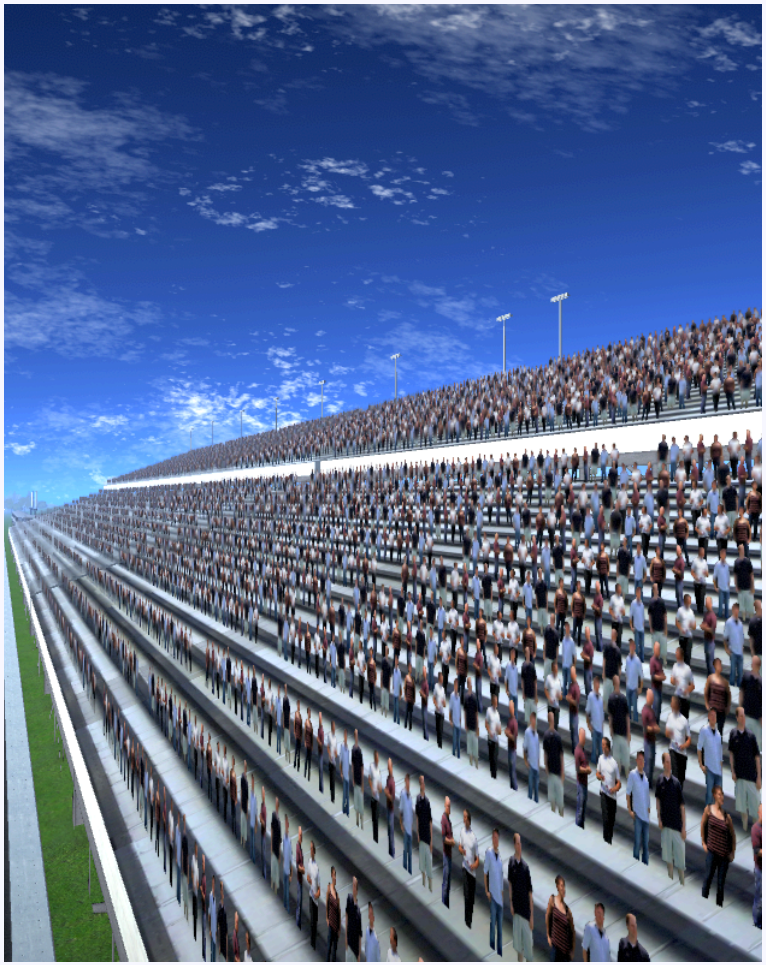
ザラザラした感じやモアレが大きく軽減できます。

© monolizm LLC

ミップマップなし



ミップマップあり



静止画だとわかりにくいかもしれませんが、ミップマップ無しの場合、実行のザラザラがかなり気になります。

© monolizm LLC

冬休みに勉強した技術

※本当は冬休みどころではない

実物を見せながら紹介します

© monolizm LLC

【１日の流れ】

・ゲームの世界の時刻を管理するクラスを作成。
時間に影響するものは、全てそのクラスを参照する。

・ディレクショナルライトのX軸回転で、
空の色と影の向き＆長さを作っている

・水平線より太陽が下がった時に影が上に向かわないように、
太陽水平線に近づくにつれて影を薄くしている

・時刻に合わせて、ライトの色と強さを、空の色の濃さを変更

・天気も変化する。不自然な天気の変化が起きないように、
過去の天気のバッファを保持して、
天気の変化の確率をコントロールしている。

© monolizm LLC

【シェーダーについて】
・利用しているシェーダーのほとんどはいじっている、
　または、作っている。
　不要な処理を消して高速化＆足りない処理を追加

【テクスチャについて】
　高速化のため、なるべく共通のテクスチャを使うようにしている。
　地面(256x256を3枚 + RGBAカラーマップ512x512を1枚)
　UI(2048x2048)
　モブキャラは全体で共通(2048x2048)
　建物や植物などは全体で共通の
　　不透明用テクスチャ(2048x2048)
　　半透明用テクスチャ(2048x2048)
　　の２枚だけ

© monolizm LLC

【地面の表現】
地面の表現方法は悩みました。

×**頂点カラー**
ポリゴン数を増やさないと表現が乏しくなる

×**頂点カラー　＋　テクスチャ**
ポリゴンのつなぎ目のテクスチャがくっきりしすぎて不自然

×**頂点カラー値を利用したテクスチャブレンド　＋**
頂点アンビエントオクルージョン
×テクスチャを変えられるのがポリゴン単位となってしまう

○**頂点カラー ＋ RGBカラーマップを利用したテクスチャブレンド　＋**
アルファ値アンビエントオクルージョン
RGBカラーマップの分のテクスチャが必要となってしまう
RGB値でテクスチャをブレンドしているので、
地面に使えるテクスチャが３種類まで。
256段階でブレンドしているので、128段階でブレンドにすれば、
6枚いけるかな。けどマップを作るのが大変そう。

© monolizm LLC

【空の表現】
空はUnity標準のプロシージャルSkyBoxを調整して利用。
太陽とレンズフレアも標準機能を利用。

雲と星空はドーム状のオブジェクトにテクスチャを貼って描画。

雲は、UVアニメーション＋グレースケールを利用した半透明度調整で雲っぽく見せている。
入道雲は単なるテクスチャ。

雨や雪は高速化の為に頂点シェーダーで動くものを作った。
けど屋根を突き抜けて屋内に入ってきちゃうからボツ予定。

【建物や植物について】
頂点カラー　＋　テクスチャ　＋　フォグ　＋　ライトカラー　＋
光の強さを設定できるシェーダー
を不透明用と半透明の２種類作成。
草や花は、上記　＋　風　の表現できるシェーダーを作成。

© monolizm LLC

【勉強中＆これから勉強すること】

・LODの使い方
　動かないものは問題無し
　動くものでBlenderで自作のものがうまくいかん

・シェーダーLOD
　カメラに近い時はリッチな表現
　離れた時は速度優先処理にする技術

・アニメーション管理
Animatorの管理の大変がもうイヤ
（思った通りのアニメーションに遷移してくれない制御の難しさ）
もちろんノンスクリプトで使える便利なところもある
キャラクターのような複雑なステートの制御では使いたくない
Unity社がシンプルなアニメーション管理ができる
「SimpleAnimation」というスクリプトを公開している
https://github.com/Unity-Technologies/SimpleAnimation
テラシュールブログが参考になります
http://tsubakit1.hateblo.jp/entry/2017/11/13/233334
http://tsubakit1.hateblo.jp/entry/2017/12/08/014153

© monolizm LLC

・遠くのオブジェクトの更新頻度を下げる
　モンハンワールドでもやってるね

・特許とか怖い
　知らぬ間に行なっていたことが特許を侵害していたら・・・

・アプリ内通貨の扱いの法律的な話（資金決済法）とか
　参考、アプリ開発の雑記帳
　http://yard1829.hatenablog.com/entry/2017/01/11/163410

© monolizm LLC

まとめ

技術的に知っていたとしても、
Unityではどうやってやるんだろう、とか、
なんでこれでうまくいかないんだ？、とか、
結構ありますね。
もちろん知らないこともいっぱいあるので、
常に勉強は続けなくてはと思いました。

効率よく開発するため、クオリティアップのため、
工数の見積もりの正確性をあげるため、などなど。

© monolizm LLC

ご清聴ありがとうございました

© monolizm LLC