(Unite2017)

本日の授業なんだけど…授業を始める前に一つ聞いておきたい。
Unite というイベントを知っていますか？Unity を使っていて、これを知らないのならば少し危機感を持ったほうが良い。
Unite ってのは、Unity の技術カンファレンスです。つまり CEDEC の Unity オンリーイベントみたいなもんです。

ちなみに Unite2017 に関してはスライドが公開されており、Unite2016 に関しては動画が Youtube にアップロードされたりしています。

Unite2017 のイベントスケジュールに、SlideShare されておりますので、気になる資料を各自見ておきましょう。
http://events.unity3d.jp/unite2017tokyo/session-lineup.html

僕の興味的にオススメなのは
https://www.slideshare.net/UnityTechnologiesJapan/unite-2017-tokyocunirx
UniRx のネタがもう来てる。ってのと「AI 記述のためのプログラミング言語を F#で実装」ってところです。

次に…
https://www.slideshare.net/UnityTechnologiesJapan/unite-2017-tokyo-75772316
ですが、日本語っていうか、関西弁なのは最初と最後だけで、あと全部英語っていう辛い資料ですが、ゲームプログラマ志すんだったら、まぁ、ね。

https://www.slideshare.net/UnityTechnologiesJapan/unite-2017-tokyorpgunity
ロマサガ好きなら見ておこう

https://www.slideshare.net/UnityTechnologiesJapan/unite-2017-tokyo-75775983
Unity の最適化について覚えておくべき技術(こういうの好きな人いるでしょ？)

【Unite 2017 Tokyo】スマートフォンでどこまでできる？3D ゲームをぐりぐり動かすテクニック講座
結構、数式とか出てくるんで気をつけましょう。Unity が云々かんぬんよりも、3DCG の基礎知識が大事なことがわかります。

DIY エフェクト実装：エンジニアレスでエフェクトを組み込める環境づくり
ゲーム会社のエンジニアって、アーティストのお役に立てるように動かないといけないのよね。それが自分を楽にする事に繋がります。

Unity で楽しむノンフォトリアルな絵づくり講座：トゥーンシェーダー・マニアクス
トゥーンシェーダとか好きやろ？

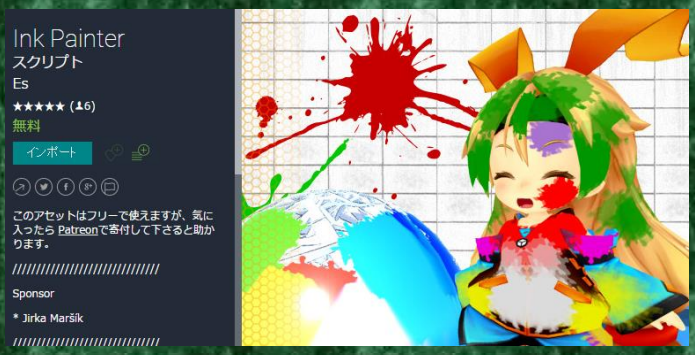
Unity×WebGL でゲームを開発・運用して見えてきたメリット・デメリット
みんな大好き DMM のお話だよー。Web ブラウザ系のゲーム開発に携わりたい人は必見

https://www.slideshare.net/Unite2017Tokyo/unite-2017-tokyo-75809779
高橋啓二朗大先生の資料です。スライド自体は 6 ページしかないですが、GitHub とかからソースコードダウンロードして研究しよう。

https://www.slideshare.net/Unite2017Tokyo/unite-2017-tokyo50cm3d
可愛い美少女を愛でる系のゲーム作るための技術力を技術者は徹底的につけておく必要があります。

ゆにてぃ目アセット科スクリプト属
インクペインタ
Ink Painter

イマーム君が「InkPainter」の中身を解析したいということなので、とりあえず舐めてみる程度に解析してみようと思います。



ちなみに、良かった場合は patreon で寄付してくれとのことですので、僕は寄付させていただこうかなーって思っています。

ちなみに似たようなアセットとして

UVPaint っていうスクリプトもあるのですが、これは$37.80+消費税=4657円なんですよね…(´・ω・`)。ちょっと高い書籍くらいにはなっています。また、2D 専用でも

StainSystem ってのがありますが、これもだいたい2500円くらいですね。
それに比べれば、寄付しますよ、俺は。

って言いながら、実はUVPaintは既に買ってたんですよね…(´・ω・`)
カネを払う前にInkPainterに目をつけたイマームくんは偉い！！

さて、前フリはさておき、解析してみましょうか…

まず基本的な事を知るためにInkPainterの作者のHPを覗いておきましょう。
http://esprog.hatenablog.com/
その中でも
http://esprog.hatenablog.com/entry/2016/05/08/212355
がキモになる記事なのですが、全体的に見ておいたほうが良いので、この人のサイトは定期的に見ておきましょう。

割と専門用語バリバリで、読むの大変かもしれませんが、まぁイマーム君は理解しないとね！！
ちなみに僕は普段は Unity 使わないんで、ここで言ってるような計算は自前で式を書いてあげないといけないんですが、そこはUnity。かなりありがたい機能が多いです。

その一つに、RaycastHitって関数があって
https://docs.unity3d.com/ja/540/ScriptReference/RaycastHit.html
「レイを飛ばした際に、衝突したオブジェクトの情報を得るために使用されます。」とのことです。そういえば、この関数は僕も使ったことあります。スクリーン上で、3Dゲーム正解のオブジェクトを選択したり、TPSの照準をマウスで調整する時に使いました。

で、この関数の偉いところは衝突地点を返してくれるだけではなくて、当たった先に貼られているテクスチャのUV座標まで返してくれるんですわー。
texcoordってプロパティですけどね
https://docs.unity3d.com/ja/540/ScriptReference/RaycastHit-textureCoord.html
この↑の説明の23行目くらいで使用されています。

```
using UnityEngine;
using System.Collections;
```

```
public class ExampleClass : MonoBehaviour {
    public Camera cam;
    void Start() {
        cam = GetComponent<Camera>();
    }
    void Update() {
        if (!Input.GetMouseButton(0))
            return;

        RaycastHit hit;
        if (!Physics.Raycast(cam.ScreenPointToRay(Input.mousePosition), out hit))
            return;

        Renderer rend = hit.transform.GetComponent<Renderer>();
        MeshCollider meshCollider = hit.collider as MeshCollider;
        if (rend == null || rend.sharedMaterial == null ||
rend.sharedMaterial.mainTexture == null || meshCollider == null)
            return;

        Texture2D tex = rend.material.mainTexture as Texture2D;
        Vector2 pixelUV = hit.textureCoord;
        pixelUV.x *= tex.width;
        pixelUV.y *= tex.height;
        tex.SetPixel((int)pixelUV.x, (int)pixelUV.y, Color.black);
        tex.Apply();
    }
}
```

とりあえず、この関数の注意点としては
「コライダーがメッシュコライダーではなかった場合、 **Vector2.zero** が返ってきます。」
この部分ですね。メッシュコライダーを設定していない場合の **textureCoord** は無効であることは認識しておきましょう。

ともかくテクスチャやマテリアルを設定して、メッシュコライダーをきちんと設定しさえすれば「クリックした場所」を染めることができます。
実際にテクスチャに色を乗せているのが、**tex.SetPixel** の部分です。

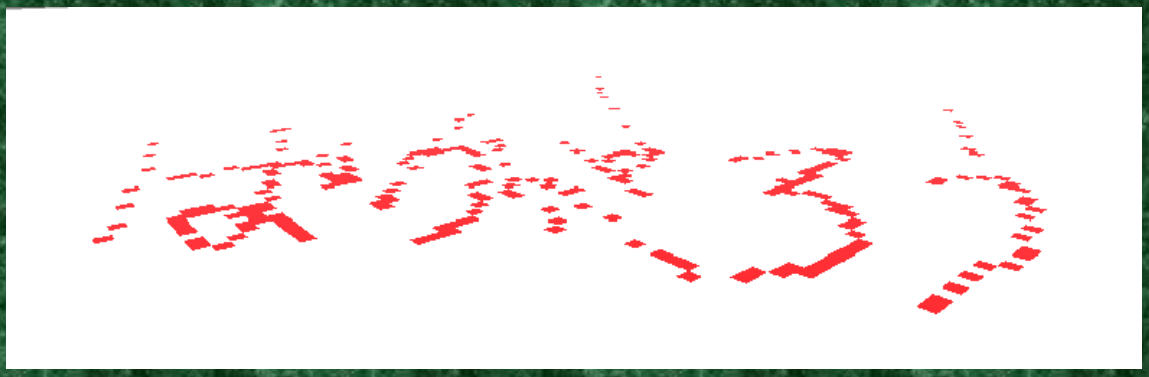
https://docs.unity3d.com/ja/540/ScriptReference/Texture2D.SetPixel.html
を見ましょう。また、SetPixels という配列を書き込む関数もありますので、色々できる事が期待されます。

注意点は

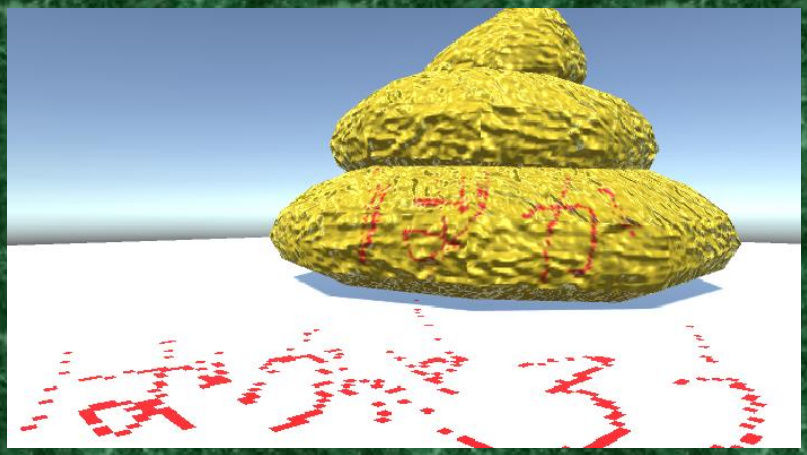
- オブジェクトにテクスチャ、マテリアルが設定されていること
- オブジェクトにはメッシュコライダーが設定されていること
- テクスチャは Read/Write Enable であること
- UV 展開は使い回ししていないこと(実質 Cube はダメ)

手始めに練習として、真っ白なテクスチャを Plane に割り当てて、インポート設定を Read/Write Enable にして、みましょう。

で、さっきのコードはカメラのスクリプトにでもしてしまいましょう。実行して絵を描くと

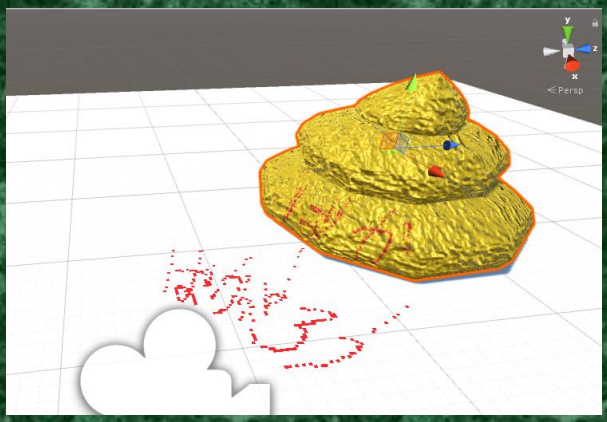


はい、Plane に絵が書かれたことがわかります。
次にメッシュを使います。メッシュコライダーは忘れないようにして、UV の設定にも気をつけましょう。



如何でしょうか？
簡単でしょう？

でも、こいつの難点は、割当テクスチャに直接色をのせていくため、一度書いてしまうと、その絵が残っちゃうんだよね。どういう事かというと…



ご覧のように、ゲームを終了しても残っちゃうんですよ。
これはひどい…
実は InkPainter はこれに対応しています。
http://esprog.hatenablog.com/entry/2016/05/08/212355
の中のコードに

```
//DynamicPaint 用 RenderTexture の生成
paintTexture = new RenderTexture(mainTexture.width, mainTexture.height, 0, RenderTextureFormat.ARGB32,
RenderTextureReadWrite.Default);
//メインテクスチャのコピー
Graphics.Blit(mainTexture, paintTexture);
```

という部分がありますが、ここでは元のテクスチャのコピーを動的に取っています。これにより元テクスチャに影響を与えず、かつ paintTexture はゲーム終了とともに開放されますのでまた、初期化されます。

今回の実験でわかったことは、DirectX をゴリゴリ書いてきた人間からすると、Unity の機能は便利すぎるってことですね。

もうここまでやってきました。早いですね。もう少しゆっくりさせていってほしいものです。

さぁ、頑張っていこうか！！

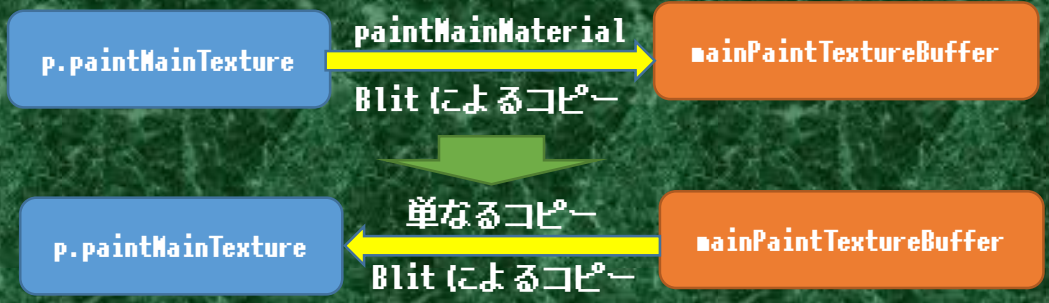
InkCanvas.cs全体をいきなり解説しても良いんですけど何しろ978行です。まぁ、大した行数ではないのですが、皆さん把握できないかと思いますので、たぶん、PaintUVDirect () 関数を見るのが分かりやすいでしょう。見てみます。
中を見るとやたら頻繁にGraphics.Blit () という関数が書かれていますね？
https://docs.unity3d.com/jp/540/ScriptReference/Graphics.Blit.html

概要
「元のテクスチャをシェーダーでレンダリングするテクスチャへコピーします。
これは主に image effects を実装するために使用されます。
Blit はレンダーターゲットとして dest を設定し、マテリアルに source _MainTex プロパティーを設定し、 フルスクリーンクワッドを描画します。」

引数の説明

source	ソーステクスチャ（コピー元のテクスチャ）
dest	コピー先の RenderTexture オブジェクト。null の場合、直接画面に転送する。
mat	使用するマテリアル。たとえば、マテリアルのシェーダーはいくつかのエフェクトを後処理できます。
pass	-1（デフォルト）の場合、マテリアルのすべてのパスを描画します。そうでなければ、指定されたパスだけを描画します。

とか書いてます。最後の引数は気にしなくていいですね。
必須なのは source と dest です。呼び出し部分を見ると
Graphics.Blit(p.paintMainTexture, mainPaintTextureBuffer, paintMainMaterial);
Graphics.Blit(mainPaintTextureBuffer, p.paintMainTexture);
こういった感じで記述されています。
前者はマテリアル(シェーダ)込で呼び出され、後者は単なるコピーです。図にするとこういう展開ですね。



何でこんなまどろっこしいことやってんのかなーって思いますが、元画像に対して加工を行った画像をテンポラリ画像である mainPaintTextureBuffer に書き込みます。「自分自身」に書き込むことは出来ませんので～。

要はマテリアル（シェーダ）適用しての画像加工を行いたいがためだけに、テンポラリ画像を使ってるんですよね。ここまでは大丈夫？ところでどんなシェーダを使ってるの？

それでは次のポイントですが
SetPaintMainData(brush, uv);

これの中身を見てみましょう。中身を見ると色々とゴチャゴチャ書いてますけれども
```
paintMainMaterial.SetVector(paintUVPropertyID, uv);
paintMainMaterial.SetTexture(brushTexturePropertyID, brush.BrushTexture);
paintMainMaterial.SetFloat(brushScalePropertyID, brush.Scale);
paintMainMaterial.SetVector(brushColorPropertyID, brush.Color);
```
これがポイントです。他人のソースコードを解析する時は「何処がポイントなのか」を押さえるのが大事です。

paintMainMaterial はマテリアル型です。コイツに対して SetVector だの SetTexture だの色々と書かれてますね？
マテリアルの説明を見てみましょう
https://docs.unity3d.com/jp/540/ScriptReference/Material.html
SetVector の説明は…
https://docs.unity3d.com/jp/540/ScriptReference/Material.SetVector.html
です。プロパティを名前か ID かどちらかで指定して、それに対してベクタ値を設定します。
SetTexture は
https://docs.unity3d.com/jp/540/ScriptReference/Material.SetTexture.html
ですね。Vector と同じですが、型が Texture ってだけです。
その他
SetFloat
https://docs.unity3d.com/jp/540/ScriptReference/Material.Set

Float.html
これも同様ね。
まぁ、だいたいやってることは分かるでしょ？
今一度

```
paintMainMaterial.SetVector(paintUVPropertyID, uv);
paintMainMaterial.SetTexture(brushTexturePropertyID, brush.BrushTexture);
paintMainMaterial.SetFloat(brushScalePropertyID, brush.Scale);
paintMainMaterial.SetVector(brushColorPropertyID, brush.Color);
```

を見ると、全部指定はIDでやってそうだということがわかります。でもIDだとよくわかんないので、検索してみましょうか…

そこでInitPropertyIDという関数が見つかりました。
こいつはAwake関数で呼ばれています。AwakeはStart関数の呼び出し直前で呼ばれるものですので、コンストラクタ関数みたいなもんだろうね。

さて、翻ってInitPropertyID関数ですけど

```
paintUVPropertyID = Shader.PropertyToID("_PaintUV");
brushTexturePropertyID = Shader.PropertyToID("_Brush");
brushScalePropertyID = Shader.PropertyToID("_BrushScale");
brushColorPropertyID = Shader.PropertyToID("_ControlColor");
```

こんな感じです。シェーダの中の変数からIDを生成して変数に入れています。これはなんちゃないです。
つまりシェーダ側のその名前の変数に値が設定されると言うだけの話です。

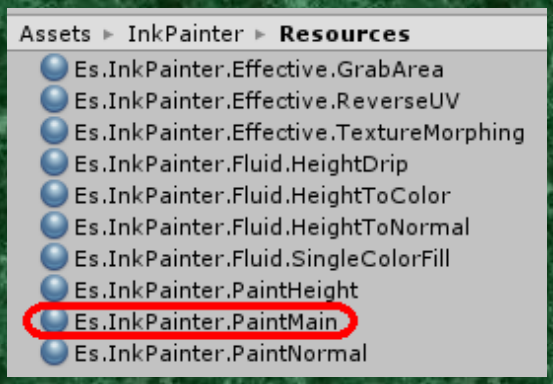
はい、ここにきてシェーダを見てみましょう（やっとか）。
この設定が、どのシェーダに紐付いているんでしょうか？

paintMainMaterialで検索すると…

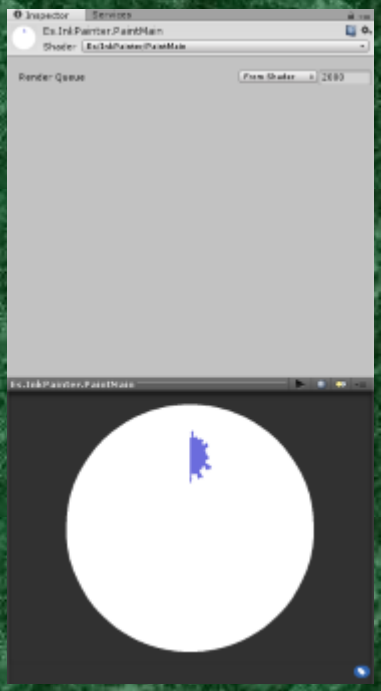
```
paintMainMaterial = Resources.Load<Material>("Es.InkPainter.PaintMain");
```

というプログラムが出てきます。(^ω^) おっ、求めるものが見つかったお！！

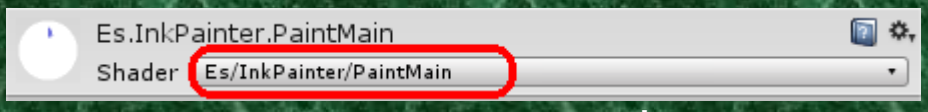
さて、**Es.InkMainter.PaintMain** というマテリアルを探しましょう。



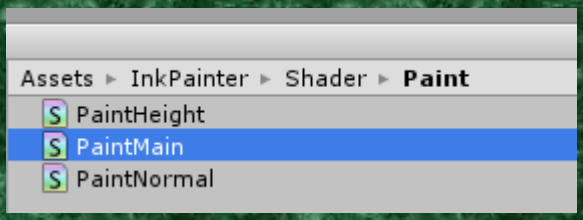
あったー！！さてマテリアルを見てみましょう。



なるほど、特に特筆すべき所はないですが、みたいのはシェーダです。シェーダの部分には



と、書かれています。探しますが、**Es** なんていうフォルダはありません。



こういうフォルダとシェーダは見つけたので、これなんでしょうね。たぶん、シェーダ自

体の定義として

```
Shader "Es/InkPainter/PaintMain"{
        Properties{
                [HideInInspector]
                _MainTex("MainTex", 2D) = "white"
                [HideInInspector]
                _Brush("Brush", 2D) = "white"
                [HideInInspector]
                _BrushScale("BrushScale", FLOAT) = 0.1
                [HideInInspector]
                _ControlColor("ControlColor", VECTOR) = (0,0,0,0)
                [HideInInspector]
                _PaintUV("Hit UV Position", VECTOR) = (0,0,0,0)
                [HideInInspector]
                [KeywordEnum(USE_CONTROL, USE_BRUSH, NEUTRAL, ALPHA_ONLY)]
                INK_PAINTER_COLOR_BLEND("Color Blend Keyword", FLOAT) = 0
        }
```

などという具合に定義づけられているからでしょう。
はい、案の定

- _PaintUV
- _Brush
- _BrushScale
- _ControlColor

がありますね？

```
sampler2D _MainTex;
sampler2D _Brush;
float4 _PaintUV;
float _BrushScale;
float4 _ControlColor;
```

はい、いい感じに変数も定義されていますね？これらがどう使われているのかを見ていきましょう。

頂点シェーダは特に見る必要はないので、ピクセルシェーダ(ここでは frag) を見ます。ちなみに frag は fragment の略でしょうね。フラグメントシェーダ…聞いたことありますか？これは OpenGL とかで言うところのピクセルシェーダです。
だから、frag という関数名にしてるのです。
短いんで、中でやってることまるまるコピペしてみましょう。

```
float4 frag(v2f i) : SV_TARGET {
        float h = _BrushScale;
        float4 base = SampleTexture(_MainTex, i.uv.xy);
        float4 brushColor = float4(1, 1, 1, 1);

        if (IsPaintRange(i.uv, _PaintUV, h)) {
                float2 uv = CalcBrushUV(i.uv, _PaintUV, h);
                brushColor = SampleTexture(_Brush, uv.xy);

                return INK_PAINTER_COLOR_BLEND(base, brushColor, _ControlColor);
        }
        return base;
}
```

意外と難しいことやってないんですよね…。その中でもポイントはここ

```
if (IsPaintRange(i.uv, _PaintUV, h)) {
        float2 uv = CalcBrushUV(i.uv, _PaintUV, h);
        brushColor = SampleTexture(_Brush, uv.xy);

        return INK_PAINTER_COLOR_BLEND(base, brushColor, _ControlColor);
}
```

ちなみに CalcBrushUV と IsPaintRange 関数に関しては Lib の中の
InkPainterFoundation.cginc
って所に実装が書かれています。.cginc ってのは、C 言語で言う include みたいなもんだと思ってください。

んで、そこに_PaintUV って入れてますけど、これってなんでしたっけ？
paintMainMaterial.SetVector(paintUVPropertyID, uv);
を思い出しましょう。
uv を入れてましたね？そしてこの UV はなんでしたっけ？まぁ、前回やったあの UV と思っておけばいいでしょう。
つまりクリックした場所に対応する UV です。

で、

```
bool IsPaintRange(float2 mainUV, float2 paintUV, float brushScale) {
return
        paintUV.x - brushScale < mainUV.x &&
        mainUV.x < paintUV.x + brushScale &&
        paintUV.y - brushScale < mainUV.y &&
        mainUV.y < paintUV.y + brushScale;
}
```

このブラシってのが模様な？

ああ、ちなみに h は_BrushScale が入っています。また、この Scale を辿っていくと、かなり辿って行き着いた先で 0.1 であることがわかります。0.1 ってのは、書き込み先テクスチャサイズに対する 0.1 です。この辺はたぶんどっかで調整可能です。

ともかく、IsPaintRange でこのブラシによる合成を適用すべきかどうかチェックしています。指定された UV からの幅が-0.1～0.1 の範囲であれば、合成を適用しようとしています。

あと、CalcBrushUV ですけど

```
float2 CalcBrushUV(float2 mainUV, float2 paintUV, float brushScale) {
#if UNITY_UV_STARTS_AT_TOP
        return (mainUV - paintUV) / brushScale * 0.5 + 0.5;
#else
        return (paintUV - mainUV) / brushScale * 0.5 + 0.5;
#endif
```

}
これで計算されたUVがブラシテクスチャのUVとなりますので、どの色(今回は白か透明か)で塗りつぶされることになります。

ちなみにこのままだと白色でしか塗りつぶされませんが、カラー値をあとでブレンドしているため色がつきます。ふつうに乗算でいいんじゃないでしょうか？

ゴチャゴチャ言いましたけど、ポイントはGraphics.Blit関数がシェーダを適用して画像のコピーができるということを利用して、テクスチャに直接色を塗っているということです。

今回サンプルプログラムを作るのが間に合いませんでしたが、それぞれの関数の意味さえ把握すれば、やってやれないということはわかります。
次回はサンプルとして、自前で直接塗りをするところまで実演、解説します。

実演します。既にSetPixelとかでの実装はやっておりますので、今回はBlitを使った実演をやります。いきなりブラシを使うのは大変なので、Blitを使って、ただただ周辺を塗りつぶすということをまずはやってみましょう。

RaycastでUVを取ってきて、ヒットした部分のマテリアルを取ってくるまでは同じです。

で、やってみた感想ですが、やっぱり難しかったです。

何点か予想外のことがあったためだいぶ、手間取って、結果的にうまくいってないです。