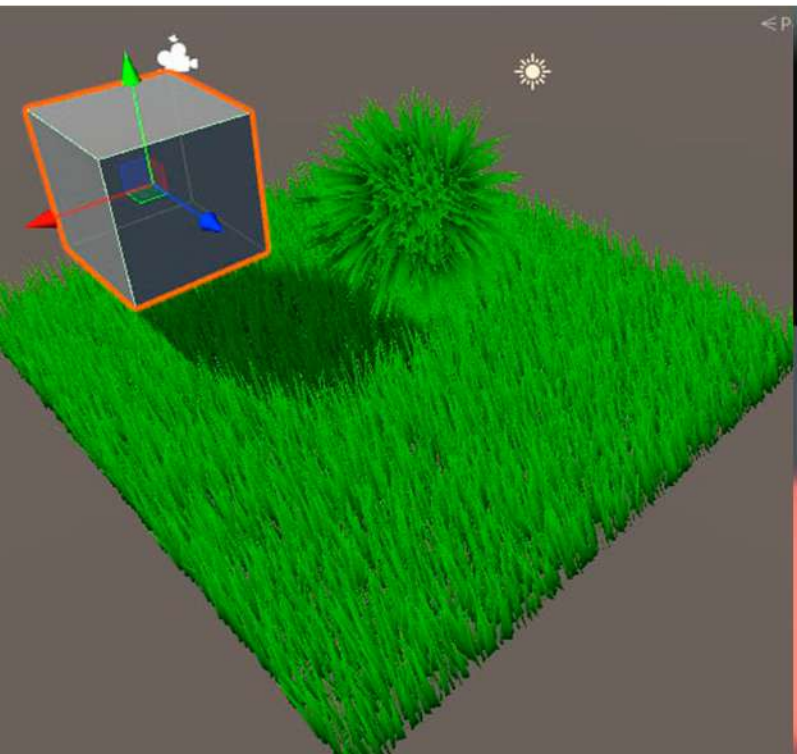
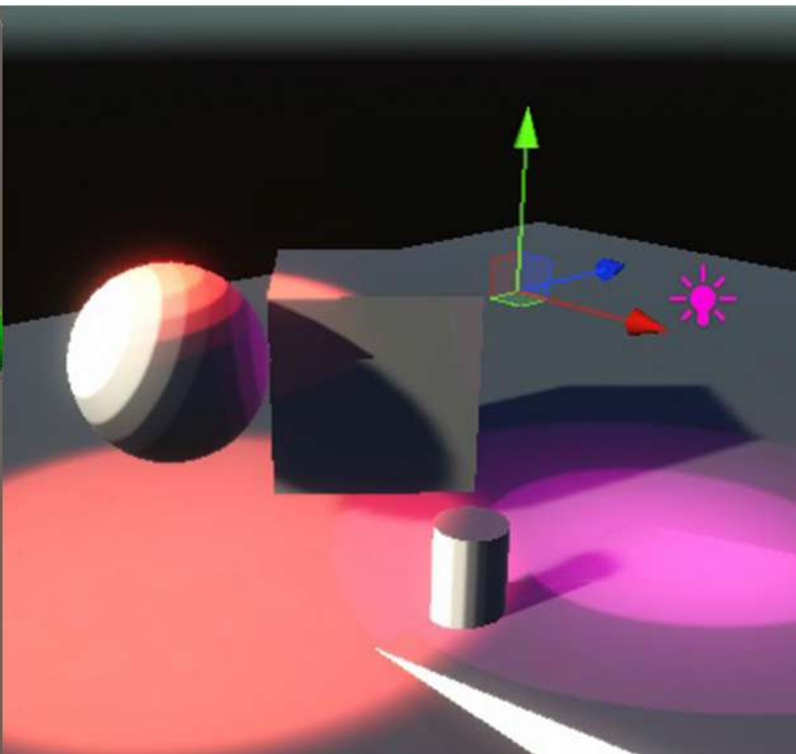
Technical Art Test

Tool Guide + Exercise 2 Document





How to use the tool **3**
1. Preparing the meshes to be painted (before placing on scene) 3
2. Preparing the meshes to be painted (on the scene) 3
3. Grass Tool Prefab and settings 4

Pipeline used **5**
Art 5
Blender 5
Photoshop 5
Embergen 6
After Effects 6
Programming 7
JetBrains Rider 7
Visual Studio Code 7

Work Done **8**
BrushEngine.cs 8
Variables 8
Functions 8
ClearTexture() 8
GetWhiteRenderTex() 9
DrawOnTexture() 9
Start Function 9
Update Function 10
BatchGrassOptimized.cs 11
ObjData Class 11
Variables 11
Functions 12
RenderBatches() 12
AddObj() 12
BuildNewBatch() 12
DrawOnRay() 13
Custom_Lightning.hlsl 14
DirectionalLightLambert() 14
CustomGI() 15
CalculateCustomLighting() 15
CustomLightModel_float() 16

Difficulties encountered & Decisions I had to take **17**
About instancing grass 17
About the mask tool 18
About the toon shader 18

Improvements Made **18**
Shader Improvements 18

Cursor Improvements 18

Suggestions 19

Multiple Meshes on the grass instancer: 19

Better Ramp control on the toon shader: 19

Better Fresnel control on the toon shader + reflection posterize: 19

Extras added 19

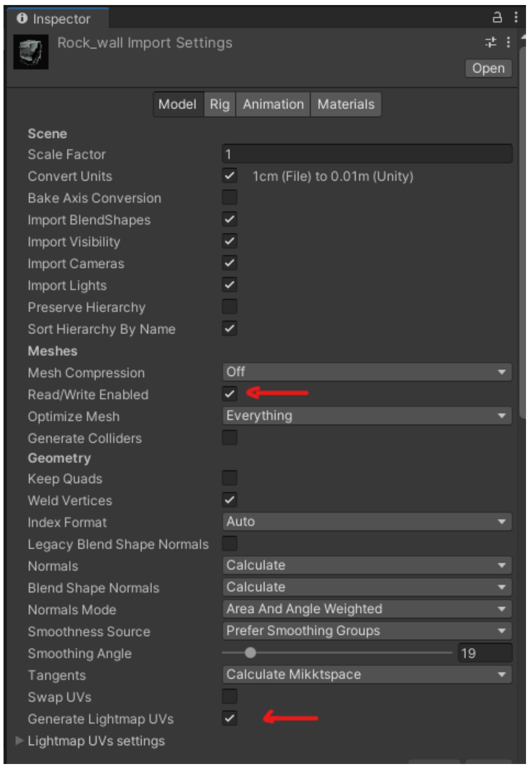
Billboard Shader and Bonfire VFX 19

Grass and leaves movement 20

References used 20

How to use the tool

1. Preparing the meshes to be painted (before placing on scene)


Inspector

Rock_wall Import Settings

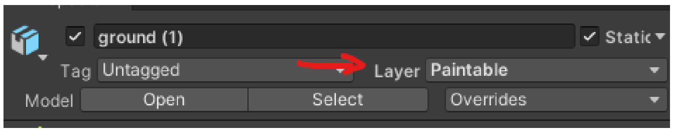
Open

Model | Rig | Animation | Materials

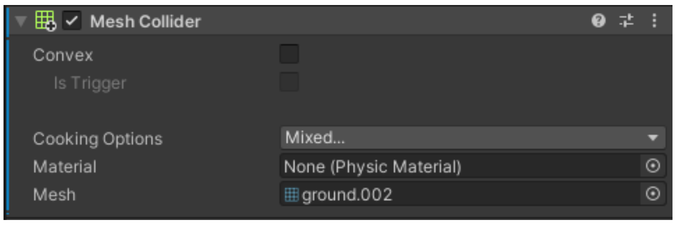
Scene	
Scale Factor	1
Convert Units	✓ 1cm (File) to 0.01m (Unity)
Bake Axis Conversion	
Import BlendShapes	✓
Import Visibility	✓
Import Cameras	✓
Import Lights	✓
Preserve Hierarchy	
Sort Hierarchy By Name	✓
Meshes	
Mesh Compression	Off
Read/Write Enabled	✓
Optimize Mesh	Everything
Generate Colliders	
Geometry	
Keep Quads	
Weld Vertices	✓
Index Format	Auto
Legacy Blend Shape Normals	
Normals	Calculate
Blend Shape Normals	Calculate
Normals Mode	Area And Angle Weighted
Smoothness Source	Prefer Smoothing Groups
Smoothing Angle	19
Tangents	Calculate Mikktspace
Swap UVs	
Generate Lightmap UVs	✓
Lightmap UVs settings	

Before doing anything, check that your mesh has **Read/Write enabled** and that the mesh has a **second uv channel with lightmap UV's**. If you didn't create the second uv channel in your modeling application, just check the **generate lightmap UVs** option.

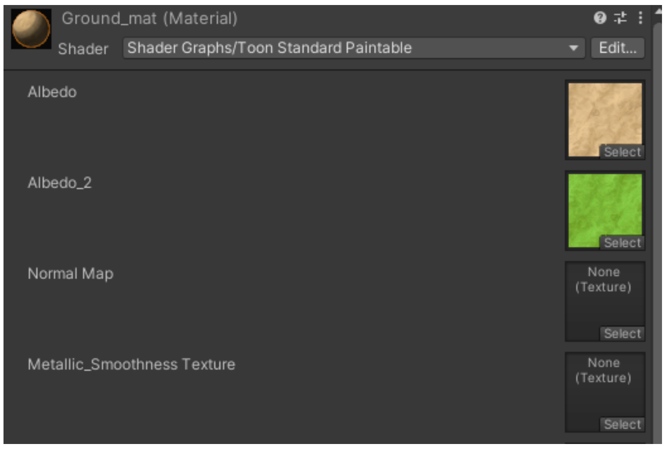
2. Preparing the meshes to be painted (on the scene)


✓ ground (1) ✓ Static

Tag	Untagged	Layer	Paintable
Model	Open	Select	Overrides


✓ Mesh Collider

Convex	
Is Trigger	
Cooking Options	Mixed...
Material	None (Physic Material)
Mesh	ground.002


Ground_mat (Material)

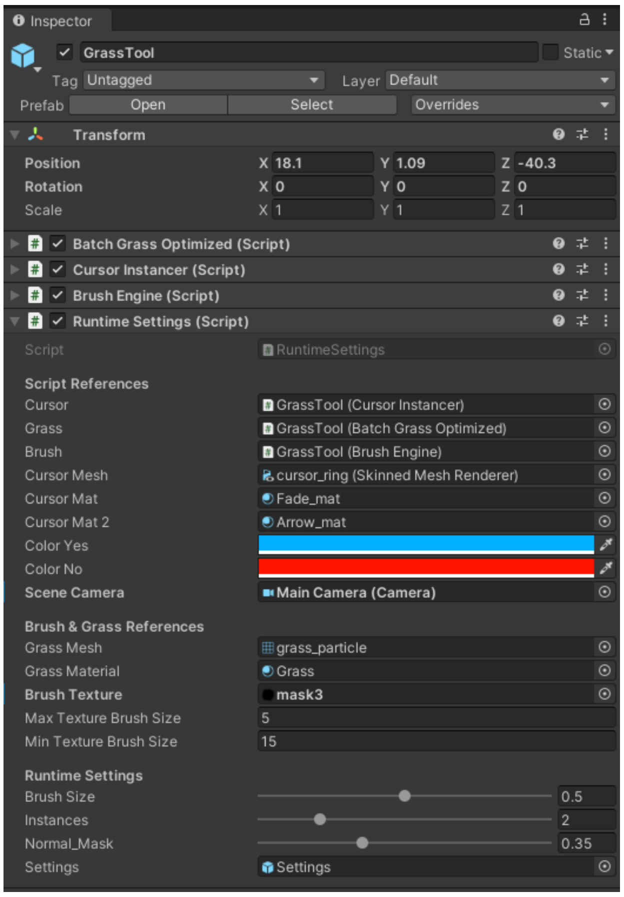
Shader: Shader Graphs/Toon Standard Paintable | Edit...

Albedo	Select
Albedo_2	Select
Normal Map	None (Texture) Select
Metallic_Smoothness Texture	None (Texture) Select

Change the object **layer to "Paintable".** Add a **mesh collider** (Necessary to get lightmap coords using a raycast). Create and set a **Toon Standard Paintable material**, or instead add a texture2D to your custom shader with the **reference _PaintMap**.

Make sure that you maintain **constant texel size between lightmaps** to make the brush size stable.

3. Grass Tool Prefab and settings


Inspector

GrassTool — Static

Tag: Untagged — Layer: Default

Prefab: Open | Select | Overrides

Transform

Position	X 18.1	Y 1.09	Z -40.3
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Batch Grass Optimized (Script)

Cursor Instancer (Script)

Brush Engine (Script)

Runtime Settings (Script)

Script	RuntimeSettings
Script References	
Cursor	GrassTool (Cursor Instancer)
Grass	GrassTool (Batch Grass Optimized)
Brush	GrassTool (Brush Engine)
Cursor Mesh	cursor_ring (Skinned Mesh Renderer)
Cursor Mat	Fade_mat
Cursor Mat 2	Arrow_mat
Color Yes	🟦
Color No	🟥
Scene Camera	Main Camera (Camera)
Brush & Grass References	
Grass Mesh	grass_particle
Grass Material	Grass
Brush Texture	mask3
Max Texture Brush Size	5
Min Texture Brush Size	15
Runtime Settings	
Brush Size	0.5
Instances	2
Normal_Mask	0.35
Settings	Settings

The only script you need to modify in order to change the tools settings is the last one, **Runtime Settings**.

Make sure that you **reference your main camera on Scene Camera**, the grass mesh in Grass Mesh and the material in Grass Material.

You can change the brush texture on the brush texture reference.

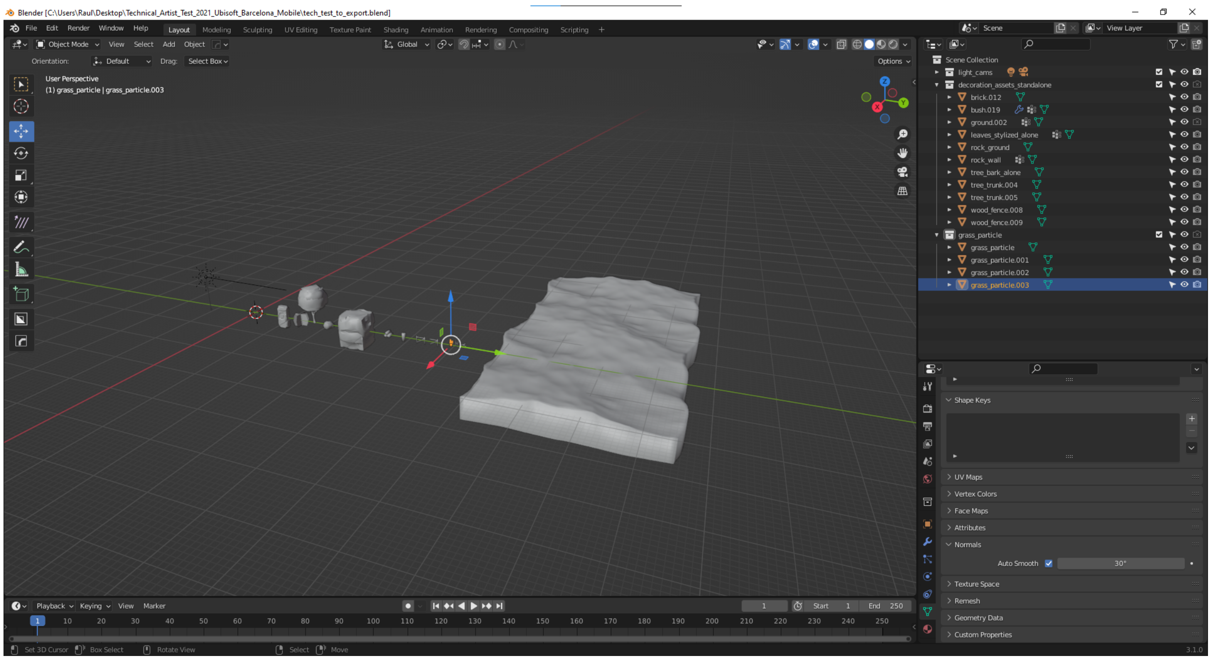
Max Texture Brush Size and Min Texture Brush Size changes the maximum and minimum size of the brush paint, **set it to values that match the grass instancer radius.**

Brush Size is controlled on the UI Sliders, Instances defines the density of the grass instancing and normal mask is a value that allows painting on only certain angles (also changed in UI).

Pipeline used

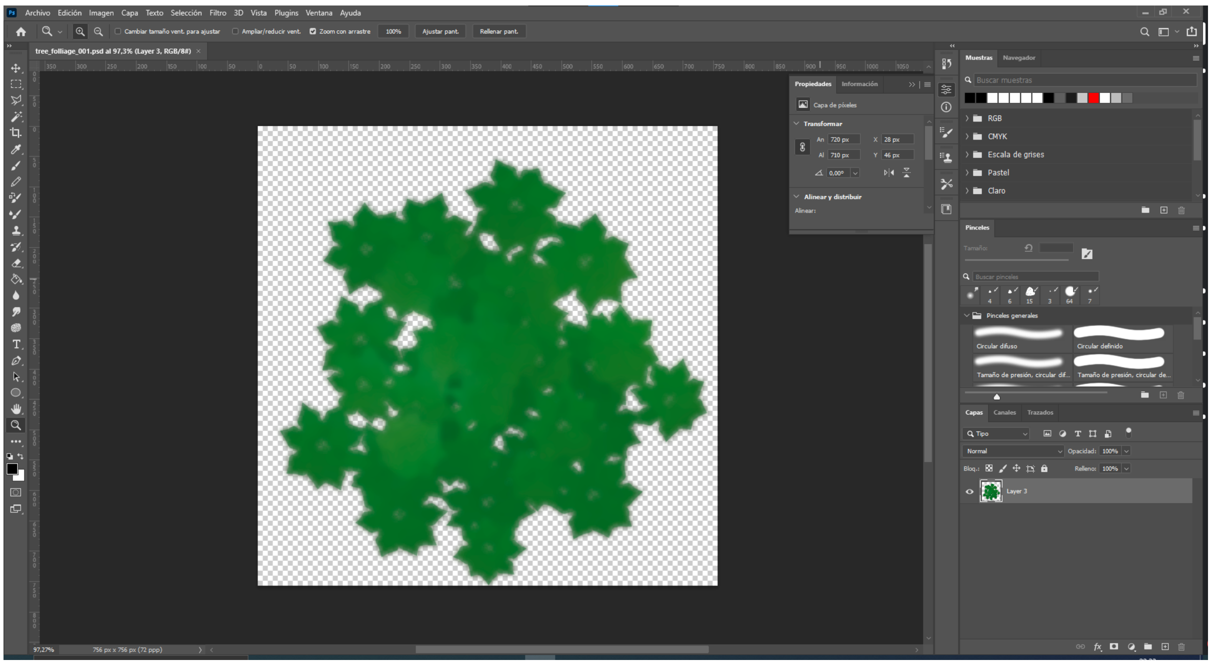
Art

Blender



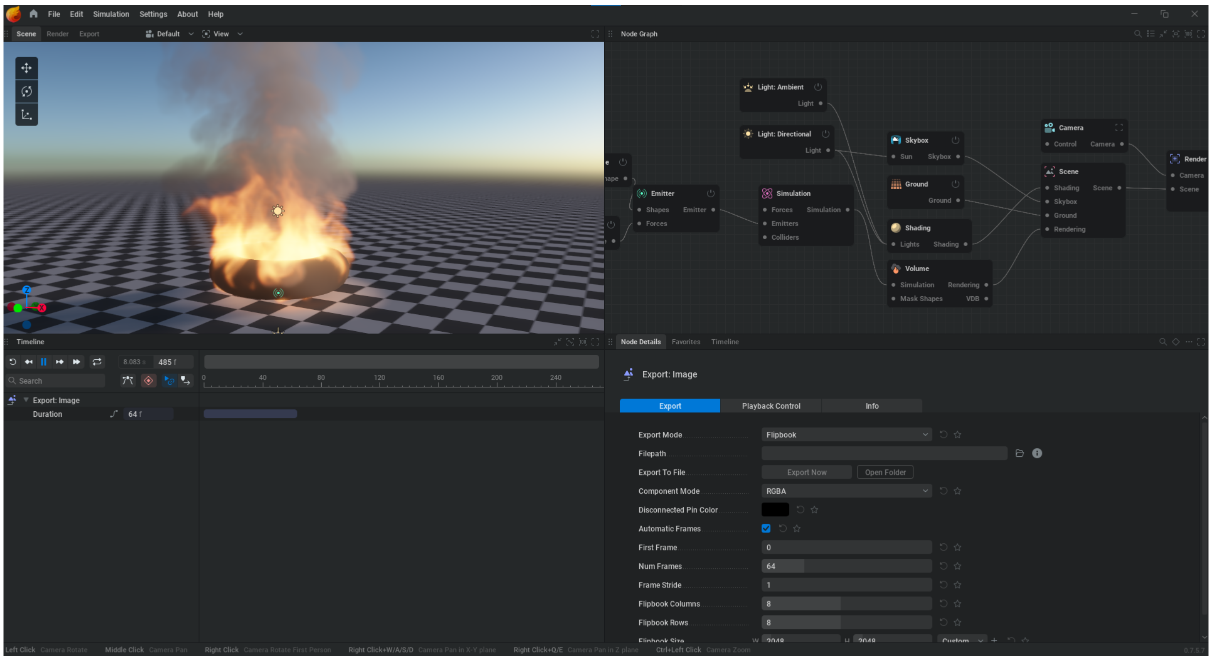
I use Blender for all of my 3D pipeline. I’ve been working with it since 2017 and I’m an avid Blender user and fan. I’ve used Blender in this project for UV editing, 3D modeling for the cursor and asset exporting.

Photoshop



I used Photoshop for texture creation, editing and export.

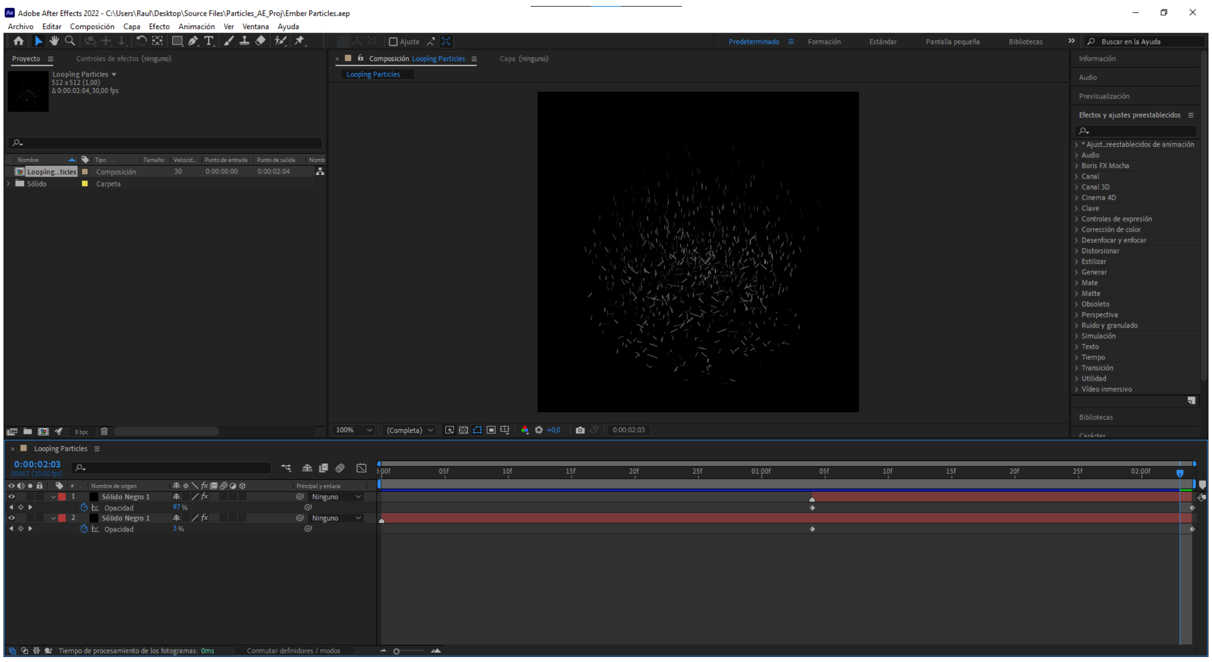
Embergen



Export Mode: Flipbook
Filepath
Export To File: Export Now / Open Folder
Component Mode: RGBA
Disconnected Pin Color
Automatic Frames: ✓
First Frame: 0
Num Frames: 64
Frame Stride: 1
Flipbook Columns: 8
Flipbook Rows: 8
Flipbook Size: W 2048 H 2048 Custom

Embergen is a specialized realtime fluid sim tool that I use for smoke simulations. It has options to export simulated fluids as flipbooks for games. I used Embergen for the Bonfire VFX.

After Effects



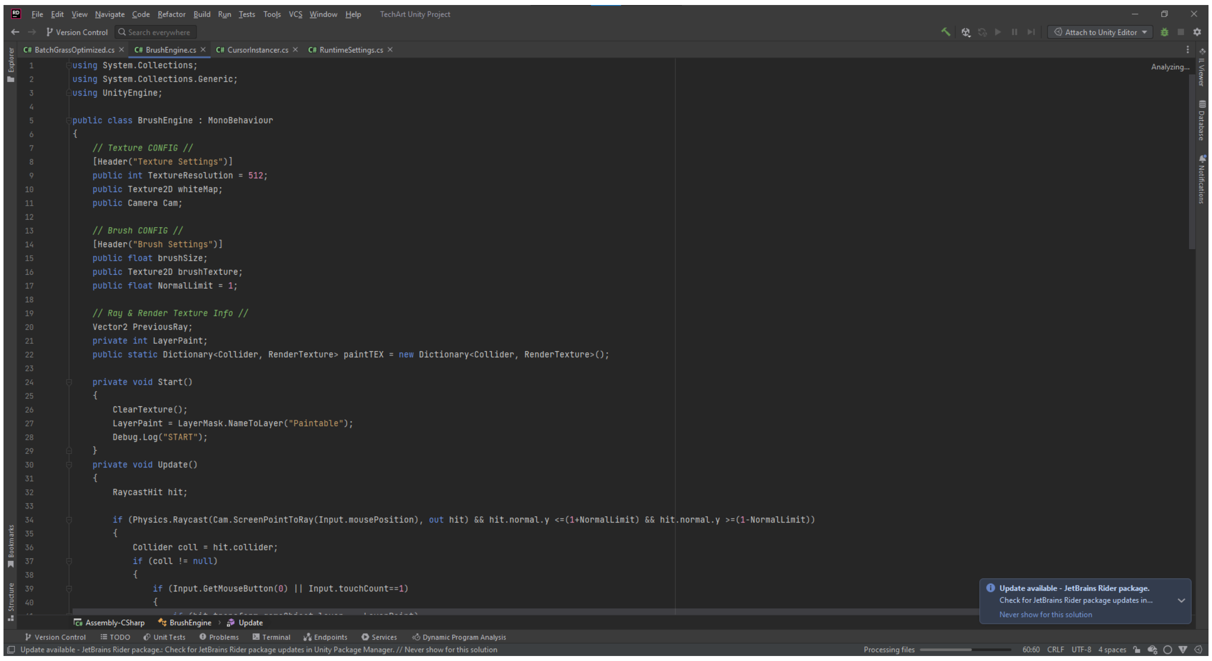
Adobe After Effects 2022 - C:\Users\Raul\Desktop\Source Files\Particles_AE_Proj\Ember Particles.aep
Archivo
Editar
Composición
Capa
Efecto
Animación
Ver
Ventana
Ayuda
Proyecto
Looping Particles
Efectos y ajustes preestablecidos

Looping Particles: 512 x 512 (1,00), Δ 0:00:02:04, 30,00 fps
0:00:02:03
1 Sólido Negro 1 — Opacidad 97 %
2 Sólido Negro 1 — Opacidad 3 %

After Effects particle systems are excellent for creating VFX for mobile games. I used After Effects to create the embers of the bonfire billboard in the scene.

Programming

JetBrains Rider

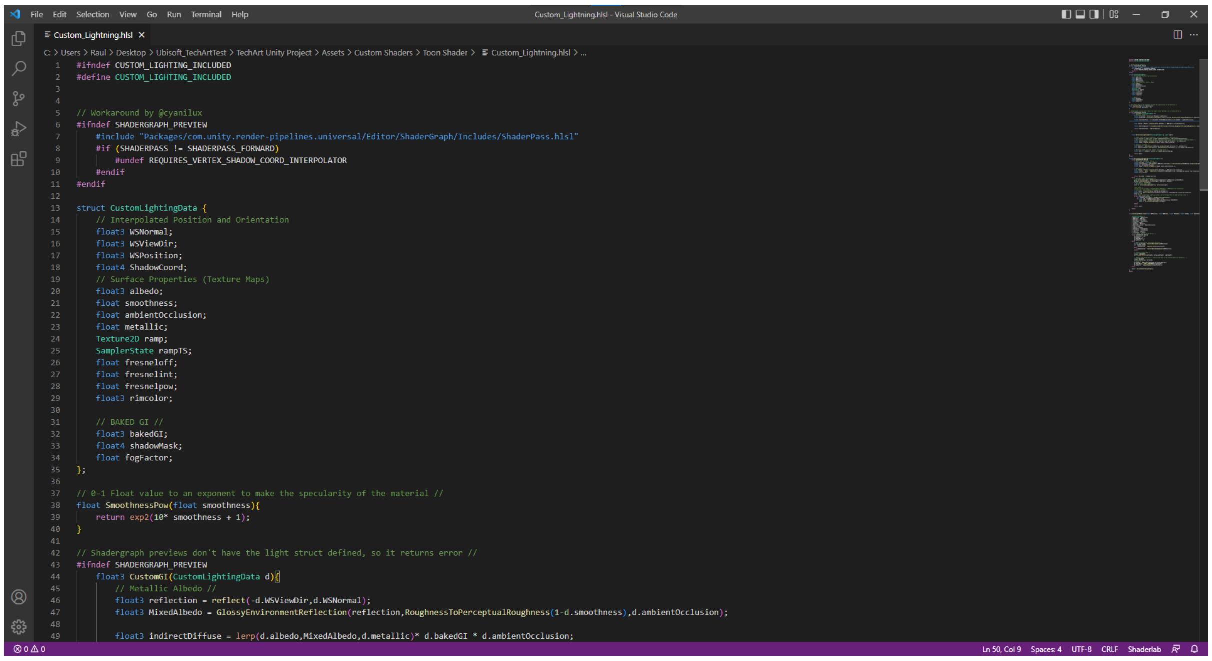


```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BrushEngine : MonoBehaviour
{
    // Texture CONFIG //
    [Header("Texture Settings")]
    public int TextureResolution = 512;
    public Texture2D whiteMap;
    public Camera Cam;

    // Brush CONFIG //
    [Header("Brush Settings")]
    public float brushSize;
    public Texture2D brushTexture;
    public float NormalLimit = 1;

    // Ray & Render Texture Info //
    Vector2 PreviousRay;
    private int LayerPaint;
    public static Dictionary<Collider, RenderTexture> paintTEX = new Dictionary<Collider, RenderTexture>();

    private void Start()
    {
        ClearTexture();
        LayerPaint = LayerMask.NameToLayer("Paintable");
        Debug.Log("START");
    }
    private void Update()
    {
        RaycastHit hit;

        if (Physics.Raycast(Cam.ScreenPointToRay(Input.mousePosition), out hit) && hit.normal.y <=(1+NormalLimit) && hit.normal.y >=(1-NormalLimit))
        {
            Collider coll = hit.collider;
            if (coll != null)
            {
                if (Input.GetMouseButton(0) || Input.touchCount==1)
                {
```

Update available - JetBrains Rider package.: Check for JetBrains Rider package updates in Unity Package Manager. // Never show for this solution

I used Rider for the scripts of the tool. Rider is the only IDE that I know that supports Shaderlab syntax highlighting, autocompletion and debugging.

Visual Studio Code



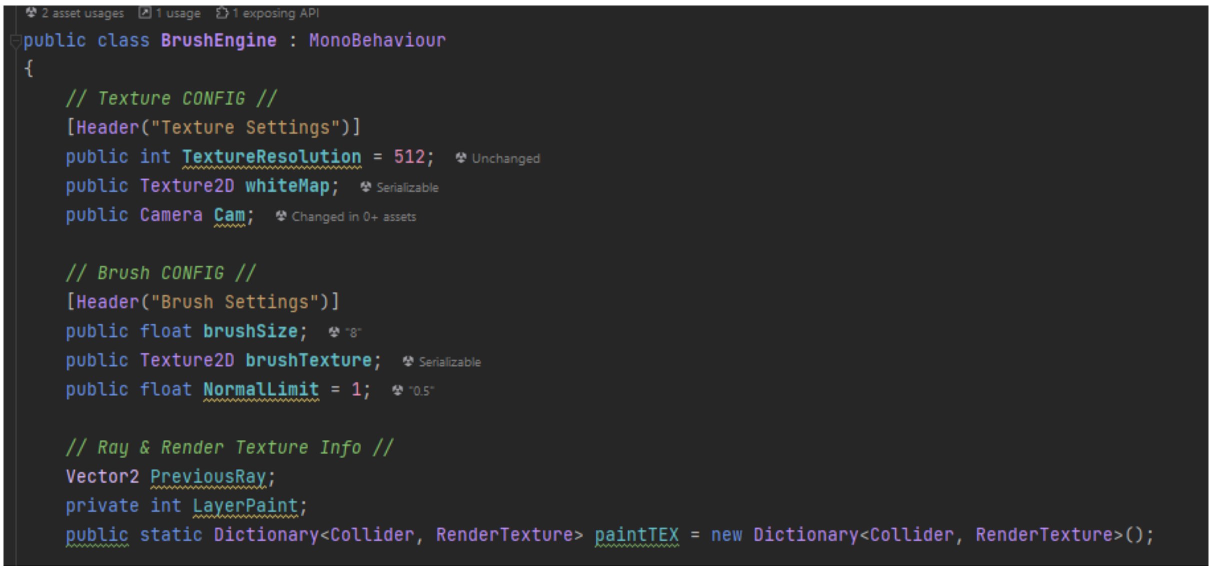
```
#ifndef CUSTOM_LIGHTING_INCLUDED
#define CUSTOM_LIGHTING_INCLUDED


// Workaround by @cyanilux
#ifndef SHADERGRAPH_PREVIEW
    #include "Packages/com.unity.render-pipelines.universal/Editor/ShaderGraph/Includes/ShaderPass.hlsl"
    #if (SHADERPASS != SHADERPASS_FORWARD)
        #undef REQUIRES_VERTEX_SHADOW_COORD_INTERPOLATOR
    #endif
#endif

struct CustomLightingData {
    // Interpolated Position and Orientation
    float3 WSNormal;
    float3 WSViewDir;
    float3 WSPosition;
    float4 ShadowCoord;
    // Surface Properties (Texture Maps)
    float3 albedo;
    float smoothness;
    float ambientOcclusion;
    float metallic;
    Texture2D ramp;
    SamplerState rampTS;
    float fresneloff;
    float fresnelint;
    float fresnelpow;
    float3 rimcolor;

    // BAKED GI //
    float3 bakedGI;
    float4 shadowMask;
    float fogFactor;
};

// 0-1 Float value to an exponent to make the specularity of the material //
float SmoothnessPow(float smoothness){
    return exp2(10* smoothness + 1);
}

// Shadergraph previews don't have the light struct defined, so it returns error //
#ifndef SHADERGRAPH_PREVIEW
    float3 CustomGI(CustomLightingData d){
        // Metallic Albedo //
        float3 reflection = reflect(-d.WSViewDir,d.WSNormal);
        float3 MixedAlbedo = GlossyEnvironmentReflection(reflection,RoughnessToPerceptualRoughness(1-d.smoothness),d.ambientOcclusion);

        float3 indirectDiffuse = lerp(d.albedo,MixedAlbedo,d.metallic)* d.bakedGI * d.ambientOcclusion;
```

I used Visual Studio Code for the toon custom function in the shadergraphs for the project. It has addons for HLSL syntax highlighting.

Work Done

BrushEngine.cs

Variables

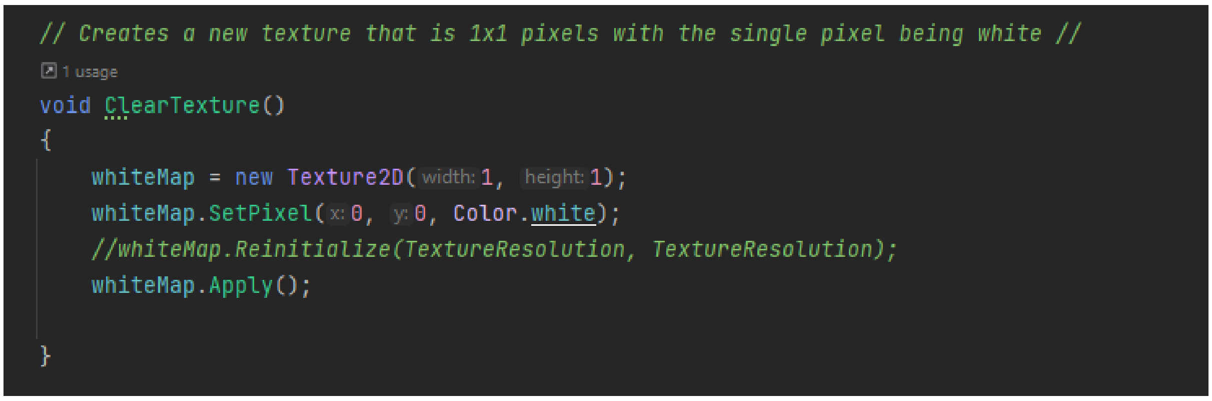


```
   2 asset usages   1 usage   1 exposing API
public class BrushEngine : MonoBehaviour
{
    // Texture CONFIG //
    [Header("Texture Settings")]
    public int TextureResolution = 512;   Unchanged
    public Texture2D whiteMap;   Serializable
    public Camera Cam;   Changed in 0+ assets

    // Brush CONFIG //
    [Header("Brush Settings")]
    public float brushSize;   "8"
    public Texture2D brushTexture;   Serializable
    public float NormalLimit = 1;   "0.5"

    // Ray & Render Texture Info //
    Vector2 PreviousRay;
    private int LayerPaint;
    public static Dictionary<Collider, RenderTexture> paintTEX = new Dictionary<Collider, RenderTexture>();
```

TextureResolution defines the resolution of the mask the script creates. whiteMap is a temporal texture 2D that holds the mask. The brush settings variables define the properties of the brush we are using to paint. Lastly, a vector2 for storing the last ray emitted from the camera, a private integer for the LayerMask of the paintable objects and a Dictionary for storing the render textures with the associated colliders.

Functions

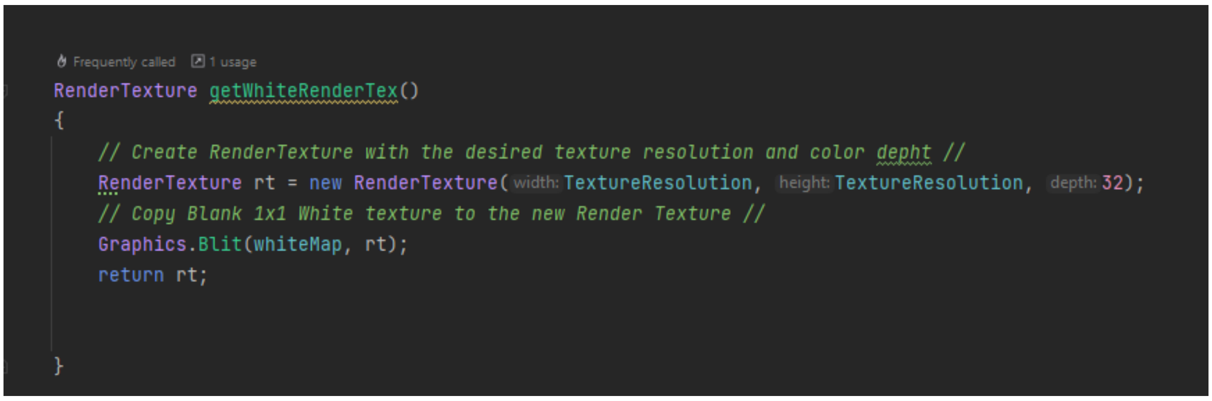
ClearTexture()



```
// Creates a new texture that is 1x1 pixels with the single pixel being white //
 1 usage
void ClearTexture()
{
    whiteMap = new Texture2D( width: 1,  height: 1);
    whiteMap.SetPixel( x: 0,  y: 0, Color.white);
    //whiteMap.Reinitialize(TextureResolution, TextureResolution);
    whiteMap.Apply();

}
```

As the comment says, it creates a 1x1 pixel white texture for the GetWhiteRenderTex function.

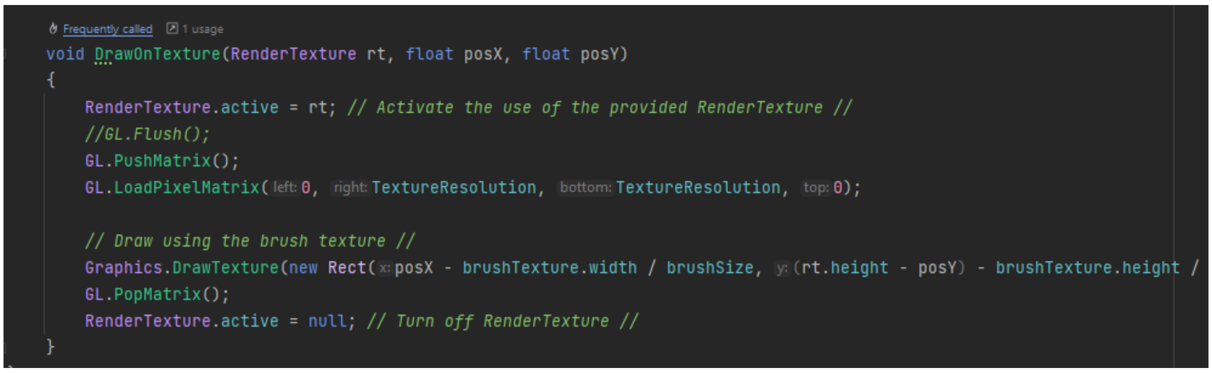
GetWhiteRenderTex()



```
Frequently called   1 usage
RenderTexture getWhiteRenderTex()
{
    // Create RenderTexture with the desired texture resolution and color depht //
    RenderTexture rt = new RenderTexture( width: TextureResolution,  height: TextureResolution,  depth: 32);
    // Copy Blank 1x1 White texture to the new Render Texture //
    Graphics.Blit(whiteMap, rt);
    return rt;


}
```

Creates a RenderTexture with the specified resolution and uses a blit command to copy the white texture to the RenderTexture, thus it returns a blank RenderTexture.

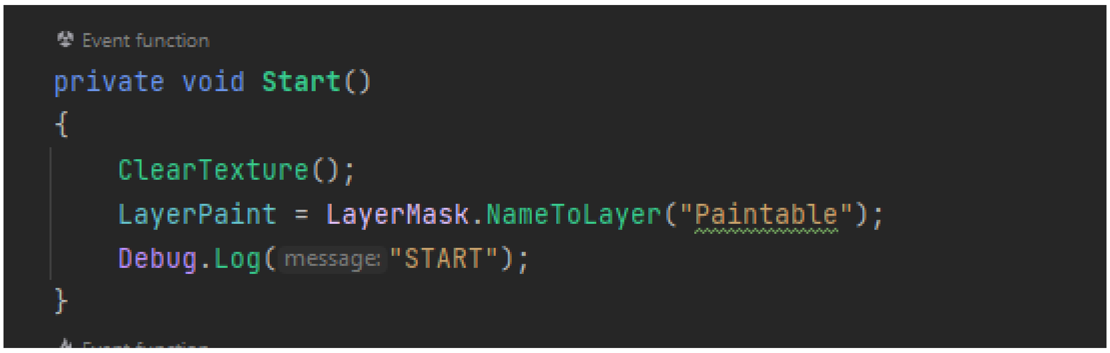
DrawOnTexture()



```
Frequently called   1 usage
void DrawOnTexture(RenderTexture rt, float posX, float posY)
{
    RenderTexture.active = rt; // Activate the use of the provided RenderTexture //
    //GL.Flush();
    GL.PushMatrix();
    GL.LoadPixelMatrix( left: 0,  right: TextureResolution,  bottom: TextureResolution,  top: 0);

    // Draw using the brush texture //
    Graphics.DrawTexture(new Rect( x: posX - brushTexture.width / brushSize,  y: (rt.height - posY) - brushTexture.height /
    GL.PopMatrix();
    RenderTexture.active = null; // Turn off RenderTexture //
}
```

The DrawOnTexture function receives the pixel coordinates where we are supposed to paint and a RenderTexture. We activate the provided RenderTexture and then we use the GL Unity libraries to draw on the texture using our mask.

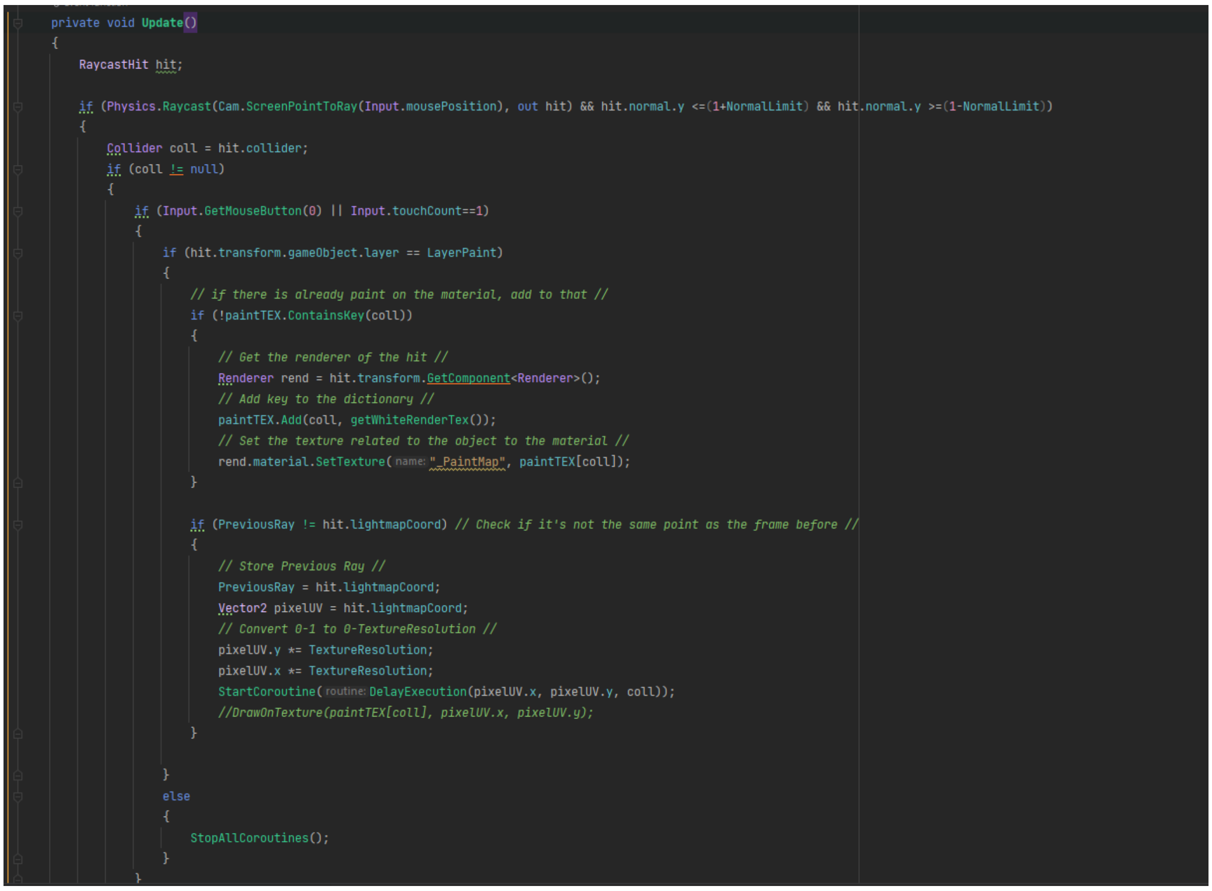
<u>**Start Function**</u>



```
Event function
private void Start()
{
    ClearTexture();
    LayerPaint = LayerMask.NameToLayer("Paintable");
    Debug.Log( message: "START");
}
```

We call the ClearTexture function to initialize the white 1x1 temporal texture. We get the bit value of the “Paintable” LayerMask.

Update Function



```
private void Update()
{
    RaycastHit hit;

    if (Physics.Raycast(Cam.ScreenPointToRay(Input.mousePosition), out hit) && hit.normal.y <=(1+NormalLimit) && hit.normal.y >=(1-NormalLimit))
    {
        Collider coll = hit.collider;
        if (coll != null)
        {
            if (Input.GetMouseButton(0) || Input.touchCount==1)
            {
                if (hit.transform.gameObject.layer == LayerPaint)
                {
                    // if there is already paint on the material, add to that //
                    if (!paintTEX.ContainsKey(coll))
                    {
                        // Get the renderer of the hit //
                        Renderer rend = hit.transform.GetComponent<Renderer>();
                        // Add key to the dictionary //
                        paintTEX.Add(coll, getWhiteRenderTex());
                        // Set the texture related to the object to the material //
                        rend.material.SetTexture( name: "_PaintMap", paintTEX[coll]);
                    }

                    if (PreviousRay != hit.lightmapCoord) // Check if it's not the same point as the frame before //
                    {
                        // Store Previous Ray //
                        PreviousRay = hit.lightmapCoord;
                        Vector2 pixelUV = hit.lightmapCoord;
                        // Convert 0-1 to 0-TextureResolution //
                        pixelUV.y *= TextureResolution;
                        pixelUV.x *= TextureResolution;
                        StartCoroutine( routine: DelayExecution(pixelUV.x, pixelUV.y, coll));
                        //DrawOnTexture(paintTEX[coll], pixelUV.x, pixelUV.y);
                    }

                }
                else
                {
                    StopAllCoroutines();
                }
            }
```

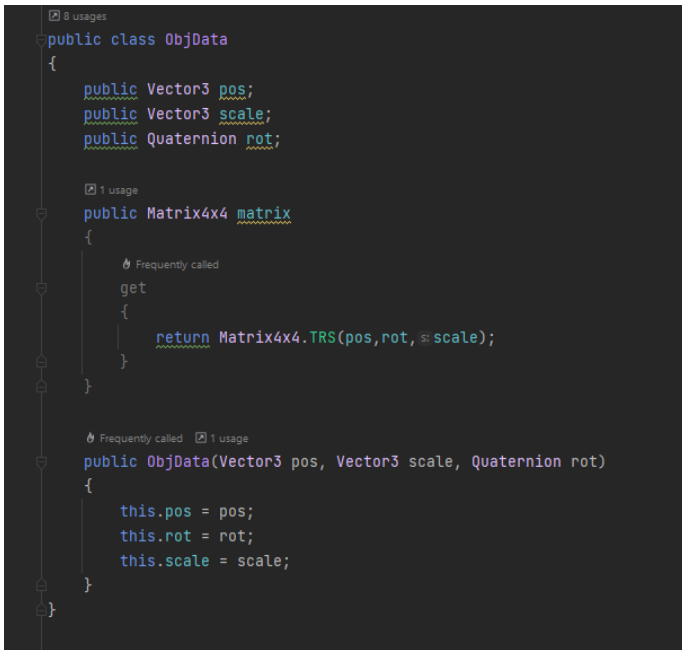
We start by doing a Raycast from the camera using the ScreenPointToRay function and we check if it’s between the normal limits of the brush.

If it hits a collider and we use the primary mouse button (0) and the LayerMask of the hit is what we specified we check if the collider of the hit exists in our dictionary as a key. If it is not defined in the paintTEX dictionary, we get the renderer component of the object, we add it to paintTEX and set the mask on the material.

If our ray is not in the same point as before, we store the current ray and check the Lightmap UV coordinates of the hit. We transform the coordinate from UV’s to pixel coordinates using the defined texture resolution and we pass the information to a coroutine that executes the DrawOnTexture function to draw on the desired point (the coroutine executes at 30 FPS).

BatchGrassOptimized.cs

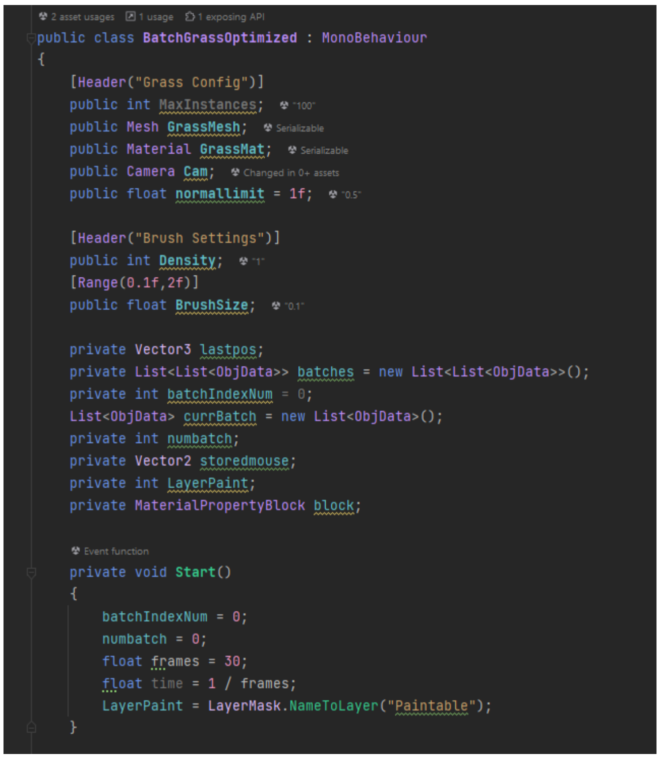
ObjData Class



```csharp
public class ObjData
{
    public Vector3 pos;
    public Vector3 scale;
    public Quaternion rot;

    public Matrix4x4 matrix
    {
        get
        {
            return Matrix4x4.TRS(pos,rot, scale);
        }
    }

    public ObjData(Vector3 pos, Vector3 scale, Quaternion rot)
    {
        this.pos = pos;
        this.rot = rot;
        this.scale = scale;
    }
}
```

A class that stores the instanced objects position, rotation and scale and constructs a Transform matrix for the DrawMeshInstanced function.

Variables



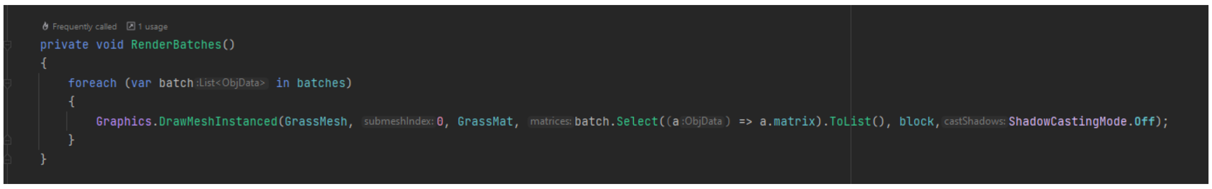
```csharp
public class BatchGrassOptimized : MonoBehaviour
{
    [Header("Grass Config")]
    public int MaxInstances;
    public Mesh GrassMesh;
    public Material GrassMat;
    public Camera Cam;
    public float normallimit = 1f;

    [Header("Brush Settings")]
    public int Density;
    [Range(0.1f,2f)]
    public float BrushSize;

    private Vector3 lastpos;
    private List<List<ObjData>> batches = new List<List<ObjData>>();
    private int batchIndexNum = 0;
    List<ObjData> currBatch = new List<ObjData>();
    private int numbatch;
    private Vector2 storedmouse;
    private int LayerPaint;
    private MaterialPropertyBlock block;

    private void Start()
    {
        batchIndexNum = 0;
        numbatch = 0;
        float frames = 30;
        float time = 1 / frames;
        LayerPaint = LayerMask.NameToLayer("Paintable");
    }
```

The Grass Config and Brush Settings of the variables are the settings and references to use in the instancing. A mesh and a material for the grass, density and brush size for the instancing radius.

The complex part, the variables that refer to the batches used for the instancing. We have a list of lists called batches that uses the ObjData class, we will use batches to do the instancing because the DrawMeshInstanced has a limit of 1024 objects per call.

Functions

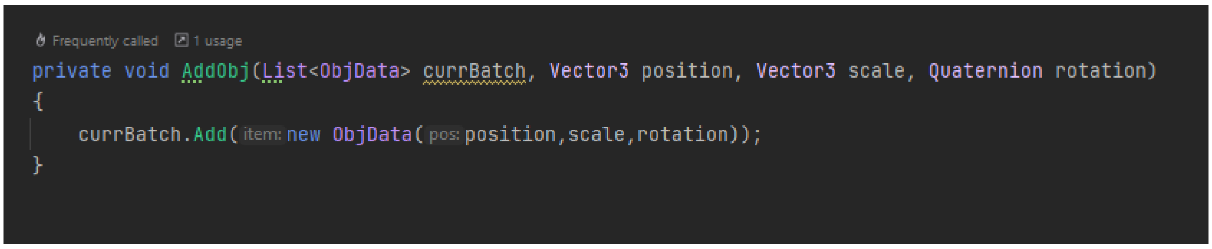
RenderBatches()



```
Frequently called   1 usage
private void RenderBatches()
{
    foreach (var batch :List<ObjData> in batches)
    {
        Graphics.DrawMeshInstanced(GrassMesh, submeshIndex: 0, GrassMat, matrices: batch.Select((a :ObjData ) => a.matrix).ToList(), block, castShadows: ShadowCastingMode.Off);
    }
}
```

We retrieve the batches and we call DrawMeshInstance for each one.

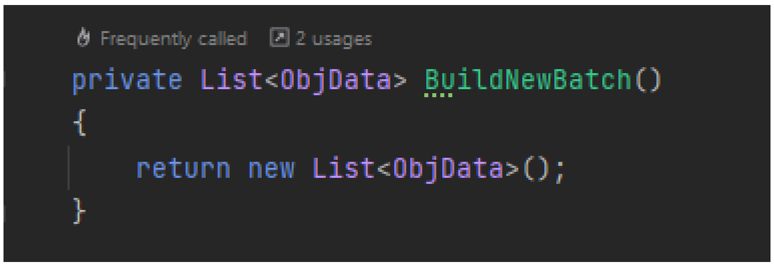
AddObj()



```
Frequently called   1 usage
private void AddObj(List<ObjData> currBatch, Vector3 position, Vector3 scale, Quaternion rotation)
{
    currBatch.Add( item: new ObjData( pos: position,scale,rotation));
}
```

Adds an object with its transform values to the current batch.

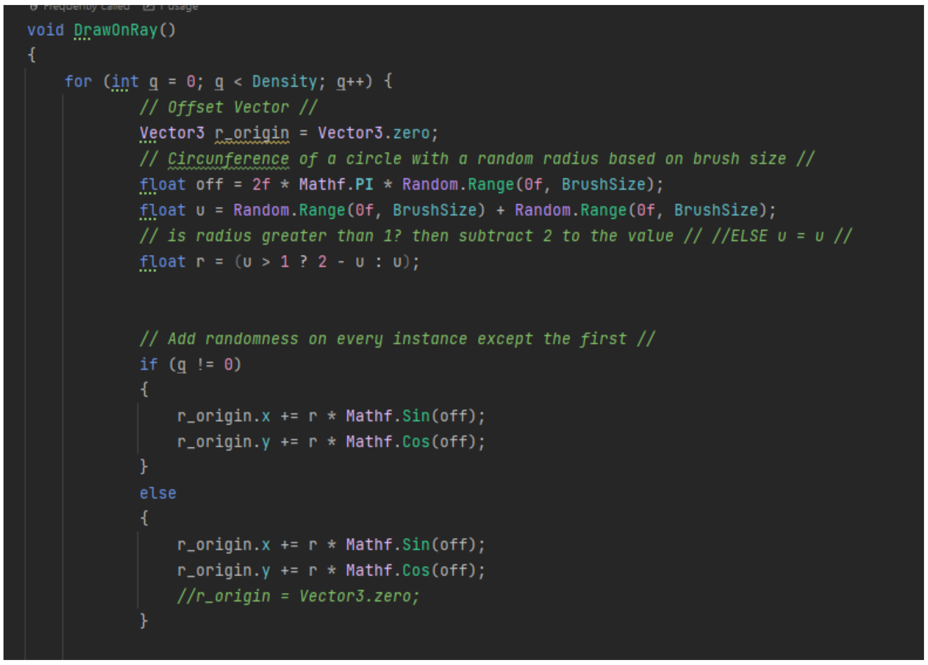
BuildNewBatch()



```
Frequently called   2 usages
private List<ObjData> BuildNewBatch()
{
    return new List<ObjData>();
}
```

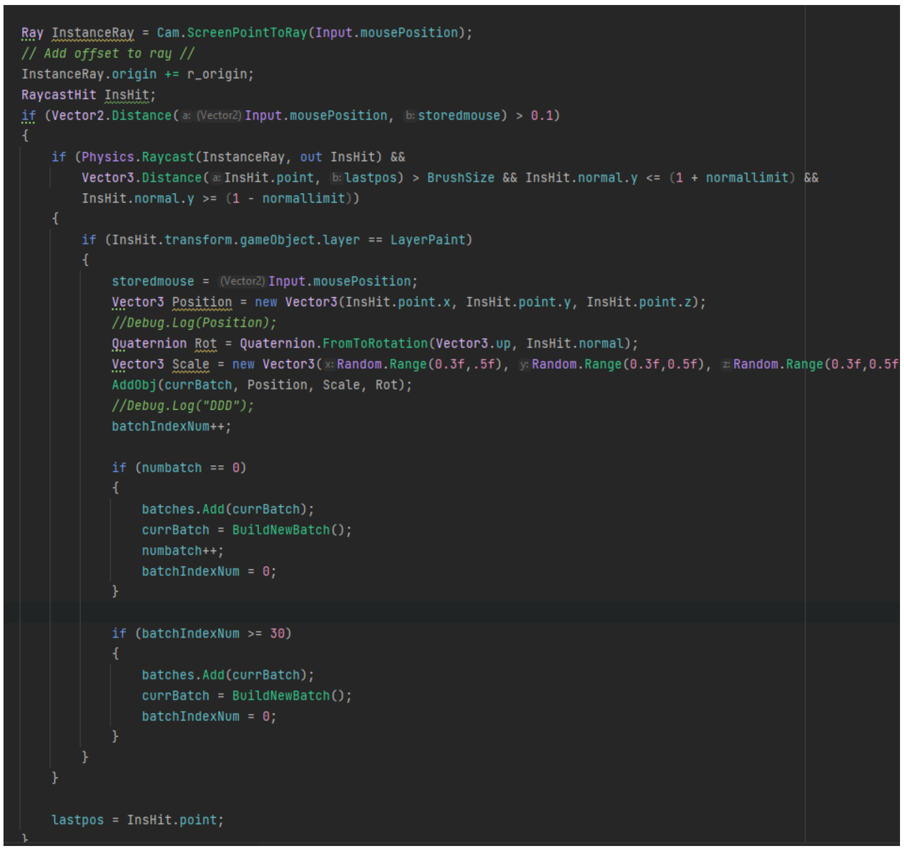
Builds a new batch when called.

DrawOnRay()



```
void DrawOnRay()
{
    for (int q = 0; q < Density; q++) {
            // Offset Vector //
            Vector3 r_origin = Vector3.zero;
            // Circunference of a circle with a random radius based on brush size //
            float off = 2f * Mathf.PI * Random.Range(0f, BrushSize);
            float u = Random.Range(0f, BrushSize) + Random.Range(0f, BrushSize);
            // is radius greater than 1? then subtract 2 to the value // //ELSE u = u //
            float r = (u > 1 ? 2 - u : u);


            // Add randomness on every instance except the first //
            if (q != 0)
            {
                r_origin.x += r * Mathf.Sin(off);
                r_origin.y += r * Mathf.Cos(off);
            }
            else
            {
                r_origin.x += r * Mathf.Sin(off);
                r_origin.y += r * Mathf.Cos(off);
                //r_origin = Vector3.zero;
            }
```

This function repeats multiple times based on the density integer. The first part adds a random offset to the instance position based on the brush radius.



```
Ray InstanceRay = Cam.ScreenPointToRay(Input.mousePosition);
// Add offset to ray //
InstanceRay.origin += r_origin;
RaycastHit InsHit;
if (Vector2.Distance( a: (Vector2) Input.mousePosition,  b: storedmouse) > 0.1)
{
    if (Physics.Raycast(InstanceRay, out InsHit) &&
        Vector3.Distance( a: InsHit.point,  b: lastpos) > BrushSize && InsHit.normal.y <= (1 + normallimit) &&
        InsHit.normal.y >= (1 - normallimit))
    {
        if (InsHit.transform.gameObject.layer == LayerPaint)
        {
            storedmouse = (Vector2) Input.mousePosition;
            Vector3 Position = new Vector3(InsHit.point.x, InsHit.point.y, InsHit.point.z);
            //Debug.Log(Position);
            Quaternion Rot = Quaternion.FromToRotation(Vector3.up, InsHit.normal);
            Vector3 Scale = new Vector3( x: Random.Range(0.3f,.5f),  y: Random.Range(0.3f,0.5f),  z: Random.Range(0.3f,0.5f
            AddObj(currBatch, Position, Scale, Rot);
            //Debug.Log("DDD");
            batchIndexNum++;

            if (numbatch == 0)
            {
                batches.Add(currBatch);
                currBatch = BuildNewBatch();
                numbatch++;
                batchIndexNum = 0;
            }

            if (batchIndexNum >= 30)
            {
                batches.Add(currBatch);
                currBatch = BuildNewBatch();
                batchIndexNum = 0;
            }
        }
    }

    lastpos = InsHit.point;
}
```

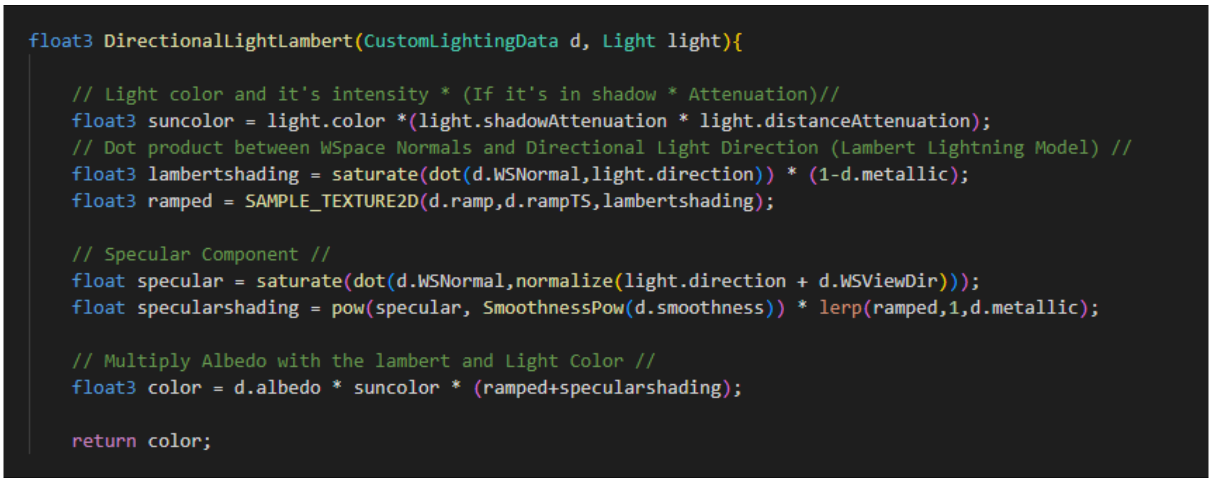
We cast a ray from the camera using the mouse coordinates. we check if the mouse position has moved more than a 0.1 threshold to check if we are not maintaining the mouse in place.

If the ray hits and the distance from the last hit is greater than the threshold defined by the brush radius and the normal of the surface is between the limits established, we then check if it’s the LayerMask that we specified.

If all of the conditions mentioned above are true, we store the mouse position for the next check and we define the position, rotation and scale of the mesh we are going to add to the current batch of instances. If the batch index (the number of batches created) is greater than the number we specified (in this case 30) we add the batch to the batches list and we reset the batch index to 0. We do batches of 30 batches because increasing the number introduces lag to the instancing rendering.

Custom_Lightning.hlsl

DirectionalLightLambert()



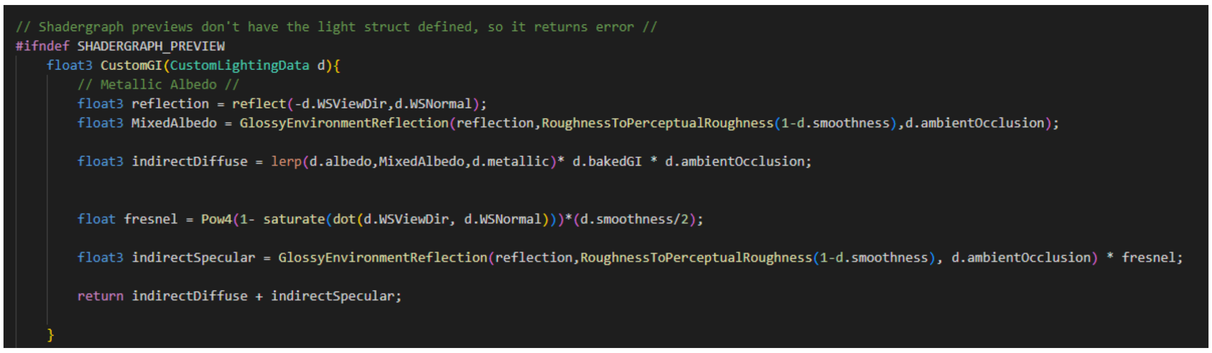
```
float3 DirectionalLightLambert(CustomLightingData d, Light light){

    // Light color and it's intensity * (If it's in shadow * Attenuation)//
    float3 suncolor = light.color *(light.shadowAttenuation * light.distanceAttenuation);
    // Dot product between WSpace Normals and Directional Light Direction (Lambert Lightning Model) //
    float3 lambertshading = saturate(dot(d.WSNormal,light.direction)) * (1-d.metallic);
    float3 ramped = SAMPLE_TEXTURE2D(d.ramp,d.rampTS,lambertshading);

    // Specular Component //
    float specular = saturate(dot(d.WSNormal,normalize(light.direction + d.WSViewDir)));
    float specularshading = pow(specular, SmoothnessPow(d.smoothness)) * lerp(ramped,1,d.metallic);

    // Multiply Albedo with the lambert and Light Color //
    float3 color = d.albedo * suncolor * (ramped+specularshading);

    return color;
```

The function used to shade the object. Diffuse shading, or in the function, Lambert shading is produced by the dot product of the world space normals and the light direction multiplied by the inverse of the metallic value (we consider that the object reflects all of the direct light energy if it’s a perfect mirror). Direct diffuse shading is remapped sampling a Ramp texture.

Specular shading is the dot product of the world space normals and the normalized combination of the light direction and the world space view direction.

The result that the function returns is the albedo color multiplied by the light color (and shadow and intensity) multiplied by the sum of the final diffuse shading and specular shading.

CustomGI()

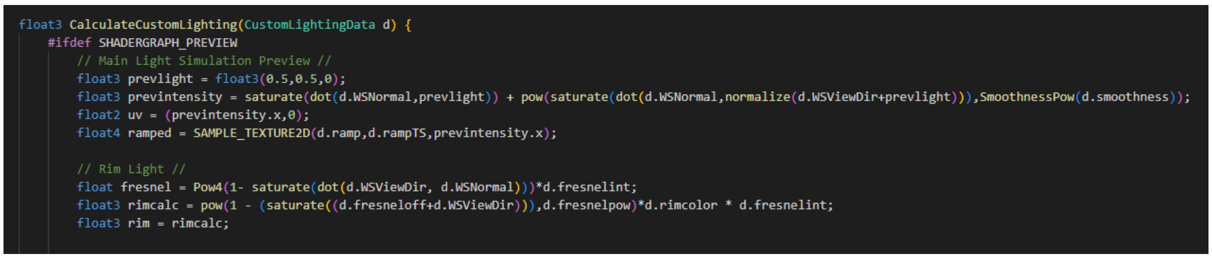


```
// Shadergraph previews don't have the light struct defined, so it returns error //
#ifndef SHADERGRAPH_PREVIEW
    float3 CustomGI(CustomLightingData d){
        // Metallic Albedo //
        float3 reflection = reflect(-d.WSViewDir,d.WSNormal);
        float3 MixedAlbedo = GlossyEnvironmentReflection(reflection,RoughnessToPerceptualRoughness(1-d.smoothness),d.ambientOcclusion);

        float3 indirectDiffuse = lerp(d.albedo,MixedAlbedo,d.metallic)* d.bakedGI * d.ambientOcclusion;


        float fresnel = Pow4(1- saturate(dot(d.WSViewDir, d.WSNormal)))*(d.smoothness/2);

        float3 indirectSpecular = GlossyEnvironmentReflection(reflection,RoughnessToPerceptualRoughness(1-d.smoothness), d.ambientOcclusion) * fresnel;

        return indirectDiffuse + indirectSpecular;

    }
```

Calculates the indirect diffuse multiplying the albedo with the light probes or sampled lightmaps GI. The indirect specular is generated by sampling the nearest cubemap reflection probe.

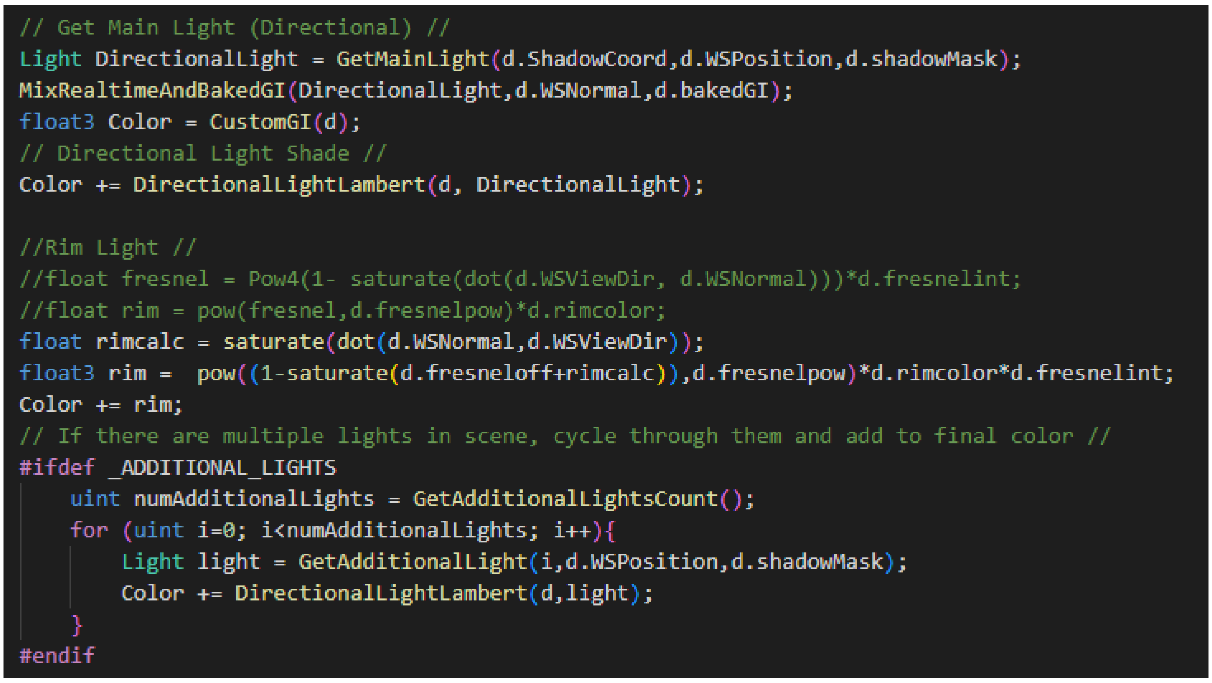
CalculateCustomLighting()

Shadergraph Preview:



```
float3 CalculateCustomLighting(CustomLightingData d) {
    #ifdef SHADERGRAPH_PREVIEW
        // Main Light Simulation Preview //
        float3 prevlight = float3(0.5,0.5,0);
        float3 previntensity = saturate(dot(d.WSNormal,prevlight)) + pow(saturate(dot(d.WSNormal,normalize(d.WSViewDir+prevlight))),SmoothnessPow(d.smoothness));
        float2 uv = (previntensity.x,0);
        float4 ramped = SAMPLE_TEXTURE2D(d.ramp,d.rampTS,previntensity.x);

        // Rim Light //
        float fresnel = Pow4(1- saturate(dot(d.WSViewDir, d.WSNormal)))*d.fresnelint;
        float3 rimcalc = pow(1 - (saturate((d.fresneloff+d.WSViewDir))),d.fresnelpow)*d.rimcolor * d.fresnelint;
        float3 rim = rimcalc;
```

Function:



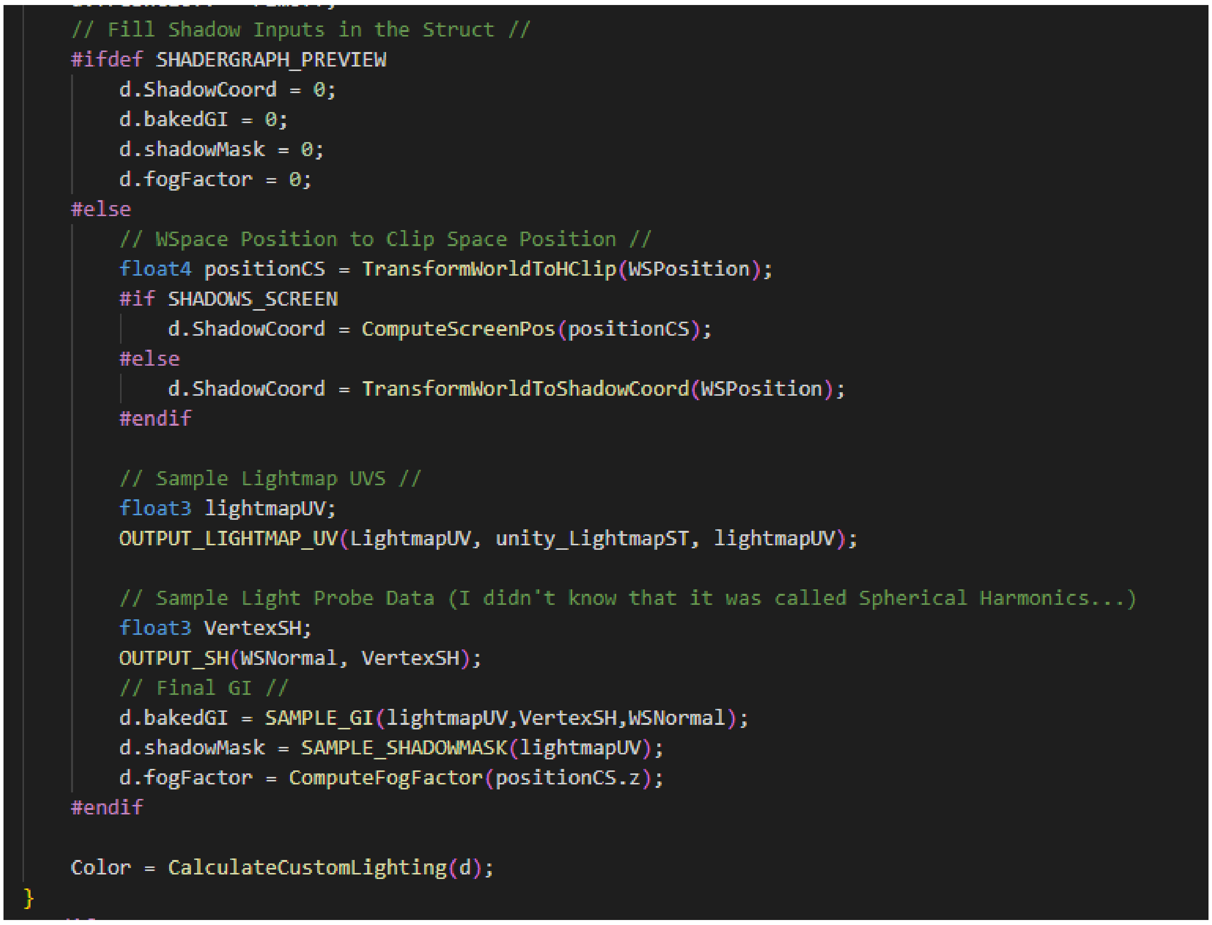
```
// Get Main Light (Directional) //
Light DirectionalLight = GetMainLight(d.ShadowCoord,d.WSPosition,d.shadowMask);
MixRealtimeAndBakedGI(DirectionalLight,d.WSNormal,d.bakedGI);
float3 Color = CustomGI(d);
// Directional Light Shade //
Color += DirectionalLightLambert(d, DirectionalLight);

//Rim Light //
//float fresnel = Pow4(1- saturate(dot(d.WSViewDir, d.WSNormal)))*d.fresnelint;
//float rim = pow(fresnel,d.fresnelpow)*d.rimcolor;
float rimcalc = saturate(dot(d.WSNormal,d.WSViewDir));
float3 rim =  pow((1-saturate(d.fresneloff+rimcalc)),d.fresnelpow)*d.rimcolor*d.fresnelint;
Color += rim;
// If there are multiple lights in scene, cycle through them and add to final color //
#ifdef _ADDITIONAL_LIGHTS
    uint numAdditionalLights = GetAdditionalLightsCount();
    for (uint i=0; i<numAdditionalLights; i++){
        Light light = GetAdditionalLight(i,d.WSPosition,d.shadowMask);
        Color += DirectionalLightLambert(d,light);
    }
#endif
```

We create a Light struct and use the GetMainLight function to get the struct from the directional light, we pass the shadow coordinates that we create in the wrapper function. We initialize the color using the GI color obtained by the CustomGI function.

We compute the first light with our DirectionalLightLambert function and add the result to the color. For the rim light we use the dot product of the world space normals and the WS view direction. We invert the result and we use the pow function to power the rim to the defined value and multiply the result by the rim intensity and color. We add the result to the color.

In order to receive light from additional lights (ex: point lights, spot, etc). We retrieve the number of additional lights and cycle through them in a for loop and use the same function as the directional light to add the extra lights to the color.

<u>CustomLightModel_float()</u>



```
    // Fill Shadow Inputs in the Struct //
    #ifdef SHADERGRAPH_PREVIEW
        d.ShadowCoord = 0;
        d.bakedGI = 0;
        d.shadowMask = 0;
        d.fogFactor = 0;
    #else
        // WSpace Position to Clip Space Position //
        float4 positionCS = TransformWorldToHClip(WSPosition);
        #if SHADOWS_SCREEN
            d.ShadowCoord = ComputeScreenPos(positionCS);
        #else
            d.ShadowCoord = TransformWorldToShadowCoord(WSPosition);
        #endif

        // Sample Lightmap UVS //
        float3 lightmapUV;
        OUTPUT_LIGHTMAP_UV(LightmapUV, unity_LightmapST, lightmapUV);

        // Sample Light Probe Data (I didn't know that it was called Spherical Harmonics...)
        float3 VertexSH;
        OUTPUT_SH(WSNormal, VertexSH);
        // Final GI //
        d.bakedGI = SAMPLE_GI(lightmapUV,VertexSH,WSNormal);
        d.shadowMask = SAMPLE_SHADOWMASK(lightmapUV);
        d.fogFactor = ComputeFogFactor(positionCS.z);
    #endif

    Color = CalculateCustomLighting(d);
}
```

The wrapper function that shadergraph uses. It fills the struct feeded to the light functions above and returns the shading as a float3 calling the CalculateCustomLightning function.

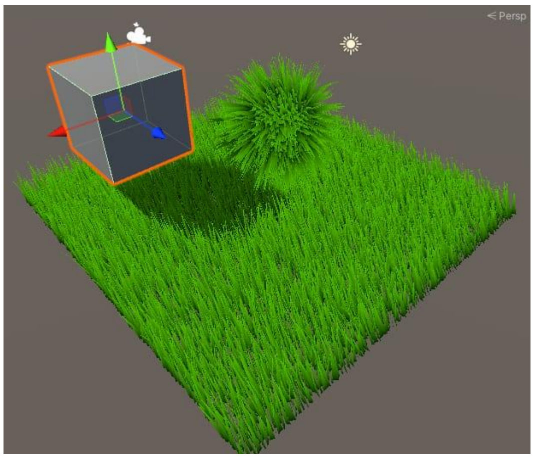
Difficulties encountered & Decisions I had to take

Most of the difficulties I encountered while producing this test project were a product of my basic knowledge of HLSL, ShaderLab and the limits of shaders on mobile platforms.

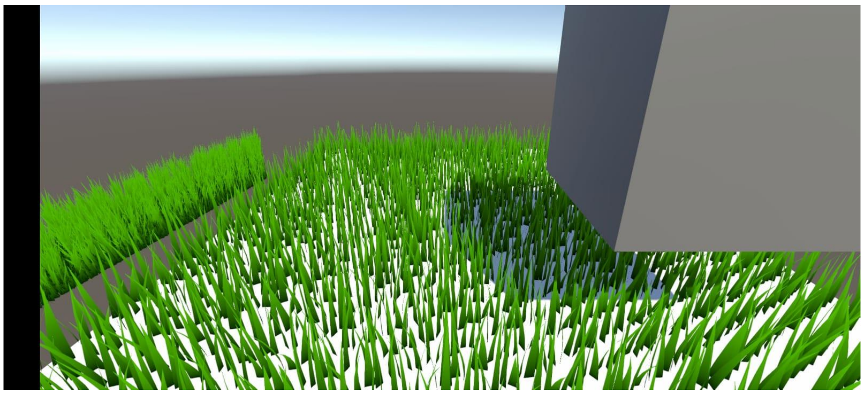
About instancing grass

I spent a lot of time (that ultimately was wasted) on researching compute shaders and geometry shaders. I used to think that geometry shaders were my best bet on creating efficient grass because of its procedural nature, so I spent time researching how they worked using multiple tutorials, the few pieces of documentation I found and URP guides on shaders.

Visually, the shader I created looked great, it was efficient on mobile and worked great with my painting script because I could sample the mask on the geometry shader for grass masking.



But soon I found out something horrible. Geometry shaders work great on DirectX, but fail to work correctly on Vulkan and don't work at all in OpenGL (even though it is technically supported). I tried working with compute shaders, but the lack of documentation made it impossible to use. I didn't want to throw the towel, so I had to find a workaround.



The workaround I found was creating batches and drawing the provided grass mesh using the DrawMeshInstanced function that Unity provides. It is not the best way, but it's the only one I found and the one I decided to use.

About the mask tool

I'm currently working on a personal project that focuses on recreating Splatoon ink bomb mechanics and VFX. I did a script to paint on lightmaps, the script created on this project (BrushEngine.cs) uses concepts of that script I created for my personal work. It is modified to work within the specifications of this test, but at its core, it is very similar.

I originally thinked of using the first channel of UV coordinates, but I decided to continue to use lightmaps because I can easily modify the texel size of all of the assets without needing to modify the UV coordinates in blender.

About the toon shader

Until now, I only had basic knowledge about the inner workings of shaders and the programming languages used to create them in Unity (CG, HLSL). Most of my shader creations for my personal projects on my portfolio are made using entirely Shadergraph, what forced me to learn more about the abyss that shader programming is was my curiosity about geometry shaders and custom functions in shadergraph.

Because it was my first time doing custom lighting all by code I had to research for guides, tutorials and documentation about writing shaders in HLSL. Ultimately I followed some tutorials, mostly for knowing how to use the included Unity HLSL libraries, but the final shaders are greatly modified and have multiple improvements, so a big part of the code is mine.

Improvements Made

Shader Improvements

I implemented metallic compatibility on my toon shaders. Most toon shaders don't consider metallic materials, but I wanted a shader that used the standard textures used for PBR materials and uses them for a cartoony style.

Another addition on my toon shader was the inclusion of a Ramp texture for remapping the lambert shading. A ramp texture allows for a better artistic control of the shadows and lights of the environment, you can use them to reduce shadows, to flatten them, etc.

Cursor Improvements

I wanted to have an interactive cursor that gives feedback to players about the possible interactions. My cursor changes color depending if a surface is paintable or not (the colors are changeable in the script for accessibility reasons) , the arrow also helps with the tracking of the cursor on the screen.

Suggestions

Multiple Meshes on the grass instancer:

Currently the script that instances the grass can only instance one type of mesh. My suggestion is to expand the code to create and classify different types of batches using different meshes. The multiple types of lists would be passed to the corresponding DrawMeshInstanced functions.

Better Ramp control on the toon shader:

Currently the toon shader simply samples the ramp using the light color as the horizontal UV axis. It would be beneficial to add an offset to better control the remap that the ramp offers.

Better Fresnel control on the toon shader + reflection posterize:

Currently fresnel is not controllable. It would be interesting to introduce a multiplication factor to the mix in order control the fresnel specular visibility on the materials.

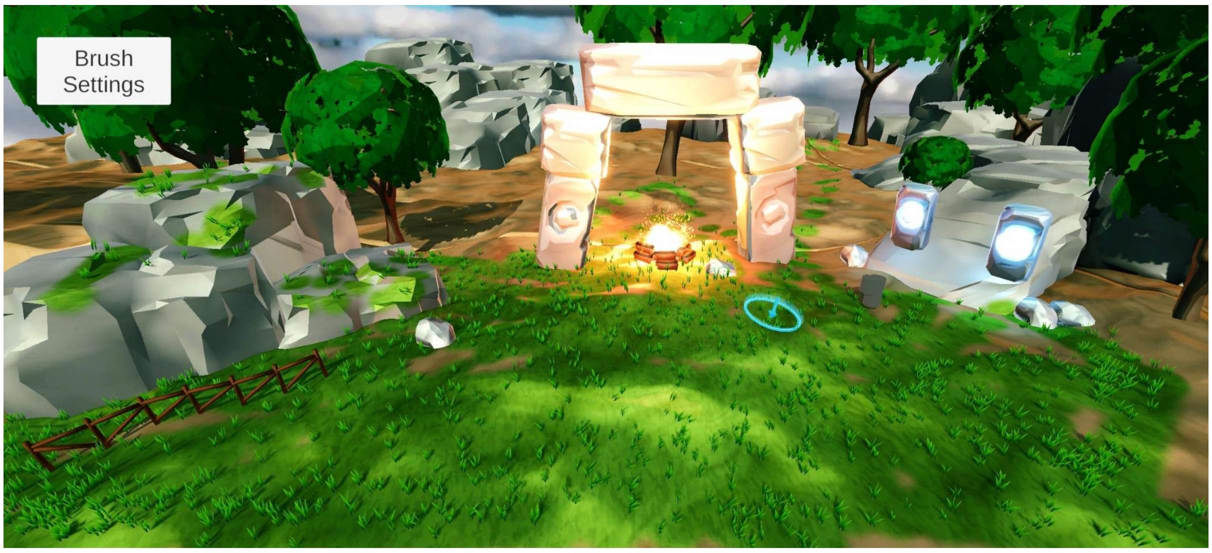
Right now the reflection probes are sampled how they are in the cubemap texture, I want to add a posterize effect to the reflections in order to give a more hand painted style.

Extras added

Billboard Shader and Bonfire VFX

A billboard shader made with Shadergraph meant to be used with flipbook animations to add additive VFX on scenes. It is currently used in the bonfire for the fire and After Effects particles.

Grass and leaves movement



In the final scene grass and tree leaves move using a gradient noise that adds to their object position. The tree leaves move without any necessary masking, but the grass has the bottom masked using the object space y-up coordinates.

References used

Stylised Grass with Shaders in URP (danielilett.com)

Writing Shader Code in Universal RP (v2) | Cyanilux

(12759) Making Zelda: Breath of the Wild Stylised Grass in Unity URP - YouTube

Geometry-Shader Object - Win32 apps | Microsoft Docs

(12959) Getting Started with Compute Shaders in Unity - YouTube

(12959) Unity GPU Instancing in less than 7 minutes! - YouTube

(12959) Custom Lighting in Unity URP Shader Graph! Ready for Toony Lights! ✔️ 2021.1 | Game Dev Tutorial - YouTube

Unity - Manual: Unity User Manual 2020.3 (LTS) (unity3d.com)