Journal of Computer and Communications, 2025, 13(7), 319-334
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

# Camera Textures: A Study on Terrain Blending in Game Graphics Rendering

**Xianzhi Hui, Yiyu Deng***

Beijing Institute of Graphic Communication, Beijing, China
Email: shuimumushuimumu@gmail.com, *dengyiyu@bigc.edu.cn

**How to cite this paper:** Hui, X.Z. and Deng, Y.Y. (2025) Camera Textures: A Study on Terrain Blending in Game Graphics Rendering. *Journal of Computer and Communications*, **13**, 319-334.
https://doi.org/10.4236/jcc.2025.137016

**Received:** June 17, 2025
**Accepted:** July 26, 2025
**Published:** July 29, 2025

Copyright © 2025 by author(s) and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).
http://creativecommons.org/licenses/by/4.0/

## Abstract

The transitional blending between terrain and objects is a core challenge in enhancing the realism of game scenes. Aiming at the problems of poor universality, reliance on high-configuration devices, and insufficient workflow compatibility in existing technologies, this paper proposes a technical framework for Camera Textures. Inspired by deferred rendering, this framework uniformly stores core lighting channels such as depth, albedo, normal, roughness, metallic, and ambient occlusion in a view-space texture set, constructing a cross-object lighting information sharing mechanism. Through a dynamic calculation algorithm for blending factors, combined with depth-inferred world coordinates and distance metrics, it achieves adaptive blending of geometry and materials. Furthermore, an additional camera system is designed to support pre-baking channel information for complex regions, breaking through the single-viewpoint limitation. Experiments show that this solution achieves terrain blending with low labor costs and high real-time performance in the Unity engine, supports dynamic interactive scenes, and significantly improves cross-platform universality and developer workflow compatibility.

## Keywords

Camera Textures, Terrain Blending, Blending Factor, Additional Camera, Real-Time Rendering, Unity Engine

## 1. Introduction

### 1.1. Research Status of Terrain Blending

With the development of computer graphics, real-time rendering technology has gradually shifted from pursuing geometric accuracy in the early stage to the ultimate simulation of realism and immersion. The transitional blending between terrain and objects, as a key link in scene authenticity, still has significant bottlenecks

in existing technologies.

#### 1.1.1. Current Situation of Terrain Blending Issues

First, in the construction of early virtual scenes, the problem of "hard boundaries" often occurs.

The construction of real-time scenes relies on geometric mesh stitching, and terrain is mostly based on height maps or tiled meshes. At this stage, transitional blending technologies are relatively basic, and due to the lack of consistent processing of lighting and geometric details.

To solve the "hard boundary" problem at this stage, typical technologies include Texture Blending and Vertex Coloring. Texture Blending is a technology that superimposes transitional textures in the junction area between terrain and objects to alleviate vertex color. Vertex Coloring achieves local gradient by manually editing the color and transparency of junction vertices, but it is only suitable for static scenes and has a high editing cost.

Second, Physically-Based Rendering (PBR) was proposed [1], but it did not solve the "hard boundary" problem.

Based on physical rendering, it unifies the material parameters of terrain and objects, such as roughness and metallic. The BRDF lighting model is used to calculate the reflection and refraction consistency of the junction area.

To solve the "hard boundary" problem at this stage, Ambient Occlusion (AO) technology was born, and it evolved into Screen Space Ambient Occlusion (SSAO), which enhances the corner shadows of the junction area and simulates the occlusion effect of the real world [2].

In summary, the blending technology at the transition between terrain and objects remains a shortcoming of mainstream rendering solutions. However, as a bridge connecting macro-scenes and micro-details, its research is of great significance for improving the immersion and realism of virtual environments.

#### 1.1.2. Support for Terrain Blending in Mainstream Engines

First, Unreal Engine's Virtual Texturing (VT) is a texture management technology based on Tile Streaming. It solves the problem of traditional texture memory limitations and long-distance detail loss by dividing super-large textures into small tiles (Tiles) and loading them into video memory as needed [3]. The rendering effect is shown in **Figure 1**.

However, this solution of Unreal Engine has extremely difficult-to-replicate disadvantages, so other mainstream engines, such as Unity, have not followed up with this function.

Finally, in Unity, it uses a quadtree LOD system to dynamically switch the terrain mesh accuracy, combined with Frustum Culling to reduce rendering pressure. GPU instancing is used for repeated objects such as vegetation and rocks to reduce the number of Draw Calls.

Shader Graph, a visualization tool, is used to create custom blending logic, supporting multi-platform Shader variants such as HLSL and GLSL, which is conven-

ient for cross-platform deployment [4]. This tool is shown in Figure 2.

Figure 1. Terrain blending in unreal engine.

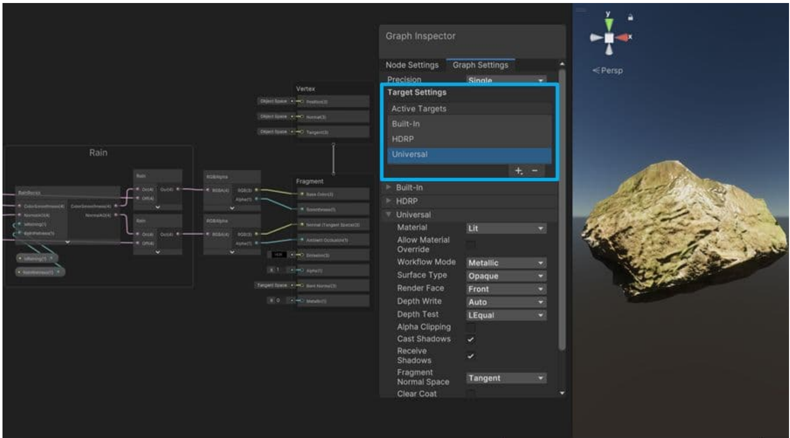


Figure 2. Unity's shader graph.

In summary, the existing technical solutions have the disadvantages of poor universality, difficulty in being popularized on the device models of the majority of users, and difficulty in combining with developers' workflows.

## 1.2. Research Objectives of Terrain Blending

### 1.2.1. Improving Users' Terrain Blending Experience

First, the terrain blending achieved by existing technologies has extremely high configuration requirements for users' experience devices, and it is impossible to achieve universalization.

Second, existing technologies cannot meet the smooth transitional blending effects between terrain and objects such as buildings, vegetation, rocks, and other artificial or natural objects in real-time rendering applications with extremely high requirements for scene realism, such as open worlds and digital twins.

### 1.2.2. Facilitating Developers' Workflow

First, for 3D scene producers, it breaks through the "hard boundary" problem when traditional terrain and objects are interspersed, realizes seamless geometric

transition between terrain and objects, and reduces the cost of manual parameter adjustment.

Second, for technical art positions, in material writing, the blending effect of the terrain can be freely connected to any workflow. Furthermore, it is not just the terrain, but any mesh object that should be able to blend with each other.

In summary, this study aims to break through the limitations of traditional geometric stitching and texture blending through interdisciplinary technology integration and construct a blending technology system that takes into account realism, real-time performance, and universality.

## 2. Research Process

### 2.1. Detailed Design and Research

#### 2.1.1. Elaboration on Core Ideas

The core idea is that when rendering interspersed objects on the terrain, before the lighting calculation stage, they not only have their own lighting calculation information but also can obtain the lighting calculation information of the terrain under their feet. Then, the two are mixed, and the mixed values are passed into the lighting calculation to achieve blending.

Based on the accumulation of relevant technologies in the Unity engine, this engine is used for technical development.

#### 2.1.2. Underlying Rendering Logic as a Necessary Condition for Implementation

The Render Feature function in the Unity Scriptable Render Pipeline is used, which is an interface provided by Unity for developers to directly access the internal logic of the rendering pipeline [5].

### 2.2. Proposal and Implementation of Camera Textures

First, Camera Textures is a new concept proposed in this study.

Its theoretical basis comes from Deferred Rendering, which is a solution mainly used to solve the rendering of a large number of lights. Its essence is to summarize the channel information for lighting calculation from the model and save it in multiple Render Targets, and finally perform lighting calculation uniformly.

Similarly, with the help of the concept of deferred rendering, it is easier to understand what Camera Textures does.

That is, obtaining the channel information for lighting calculation from the perspective of the camera from many models in the scene. For PBR metal flow rendering, the cache set should include Depth cache, Albedo (or Base Color) cache, Normal cache, as well as Metallic, Roughness (or Smoothness), and Ambient Occlusion (AO) cache, as shown in the Fragment section of **Figure 3**.

In addition, in the rendering process, deferred rendering is to summarize the information channels at the end to calculate the lighting. The behavior of Camera Textures summarizing the information channels needs to be in a more forward position in the rendering process.

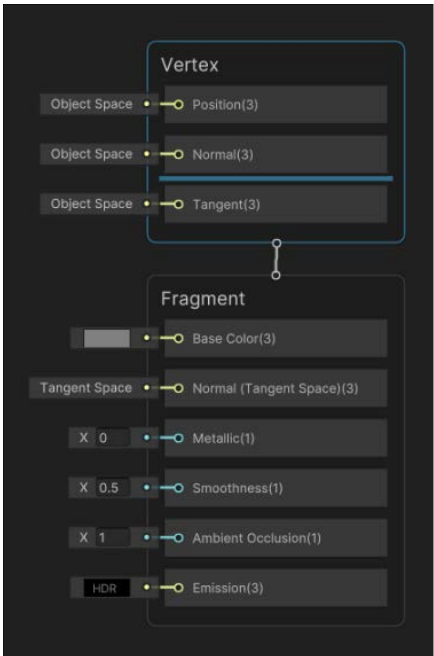


Figure 3. PBR channels.

And different from deferred rendering, which saves the results in multiple render targets. The medium used to save Camera Textures can be of different types, such as Texture 2D or Graphic Buffer.

### 2.2.1. Texture Storage Scheme

Textures have only four channels, namely red, green, blue, and transparency, as shown in Figure 4.

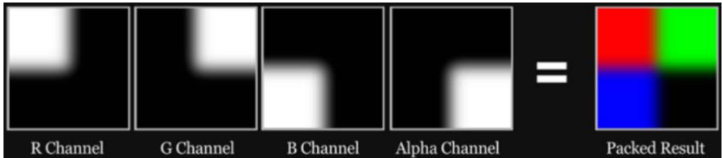


Figure 4. Texture channels.

First, when choosing a texture as the storage medium, although the depth is a floating-point number information and only occupies one channel, it is often drawn into the red channel of a separate render texture in the rendering pipeline, so it is not additionally processed and is separately saved as a depth texture, as shown in Figure 5.

Second, the albedo is color information without transparency, with a total of 3 channels for red, green, and blue, and after acquisition, it occupies an albedo texture alone, as shown in Figure 6.

Again, the normal is a three-dimensional vector, that is, XYZ with a total of 3 channels, and is also stored separately as a normal texture. For the convenience of subsequent reading, the normal stored here is in the world space coordinate system, as shown in Figure 7.

Figure 5. Depth texture.

Figure 6. Albedo texture.

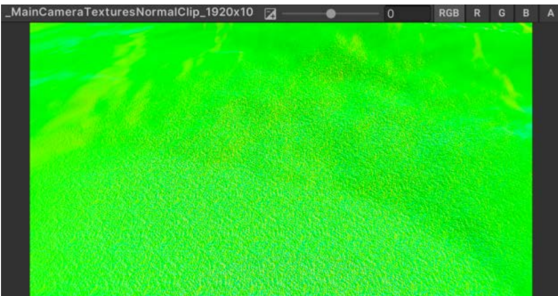


Figure 7. Normal texture.

Next, for roughness, metallic, and ambient occlusion, which are three floating-point numbers, they can be separately converged into a Mask Map. The arrangement of the channels is in accordance with the Unity rule standard, with metallic assigned to the red channel, ambient occlusion to the blue channel, the green channel left empty, and roughness in the alpha channel, as shown in Figure 8.

Finally, after this step is completed, Camera Textures are set to the global variables of the Shader for use in the subsequent interpolation and blending steps, as shown in Figure 9.

#### 2.2.2. Cache Storage Scheme

First, there is no limit to the number of channels in the cache because it is itself a collection of data structures. The data in each structure should include the lighting calculation information mentioned above, as shown in Figure 10.

Second, because the channel information obtained by the camera is itself a Render Texture (RT), using a cache to save it requires an additional step, that is, the transcription from texture to cache, as shown in Figure 11.

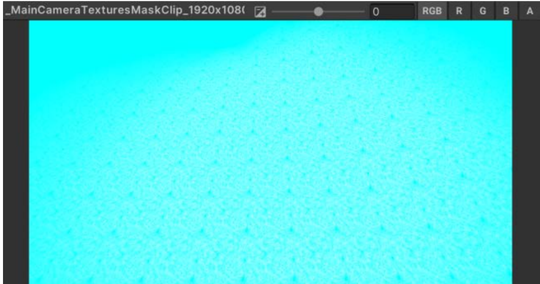


Figure 8. Mask texture.

| | | | | |
|---|---|---|---|---|
| _MainCameraTexturesAlbedoClip | fs | 1920x1080 | Tex2D | R32G32B32A32_SFloat |
| _MainCameraTexturesDepthClip | fs | 1920x1080 | Tex2D | R32_SFloat |
| _MainCameraTexturesEmissionClip | fs | 1920x1080 | Tex2D | R32G32B32A32_SFloat |
| _MainCameraTexturesMaskClip | fs | 1920x1080 | Tex2D | R32G32B32A32_SFloat |
| _MainCameraTexturesNormalClip | fs | 1920x1080 | Tex2D | R32G32B32A32_SFloat |

Figure 9. Set global variables.

```
struct Pixel
{
    float depth;
    float3 albedo;
    float3 normal;
    float3 trueNormal;
    float4 mask;
    float3 emission;
};
```

Figure 10. The data structure.

```
Texture2D tempClip = new(resolution.x, resolution.y, GraphicsFormat.R32G32B32A32_SFloat, TextureCreationFlags.None);

Color[] depthPixels = null, albedoPixels = null, normalPixels = null, trueNormalPixels = null, maskPixels = null, emissionPixels = null;

if (depthClip != null)
{
    RenderTexture.active = depthClip;
    tempClip.ReadPixels(new(0, 0, resolution.x, resolution.y), 0, 0);
    depthPixels = tempClip.GetPixels(0);
    RenderTexture.active = null;
    depthClip.Release();
}
if (albedoClip != null)
{
    RenderTexture.active = albedoClip;
    tempClip.ReadPixels(new(0, 0, resolution.x, resolution.y), 0, 0);
    albedoPixels = tempClip.GetPixels(0);
    RenderTexture.active = null;
    albedoClip.Release();
}
if (normalClip != null)
{
    RenderTexture.active = normalClip;
    tempClip.ReadPixels(new(0, 0, resolution.x, resolution.y), 0, 0);
    normalPixels = tempClip.GetPixels(0);
    RenderTexture.active = null;
    normalClip.Release();
}
if (trueNormalClip != null)
{
    RenderTexture.active = trueNormalClip;
    tempClip.ReadPixels(new(0, 0, resolution.x, resolution.y), 0, 0);
    trueNormalPixels = tempClip.GetPixels(0);
    RenderTexture.active = null;
    trueNormalClip.Release();
}
if (maskClip != null)
{
    RenderTexture.active = maskClip;
    tempClip.ReadPixels(new(0, 0, resolution.x, resolution.y), 0, 0);
    maskPixels = tempClip.GetPixels(0);
    RenderTexture.active = null;
    maskClip.Release();
}
if (emissionClip != null)
{
    RenderTexture.active = emissionClip;
    tempClip.ReadPixels(new(0, 0, resolution.x, resolution.y), 0, 0);
    emissionPixels = tempClip.GetPixels(0);
    RenderTexture.active = null;
    emissionClip.Release();
}

bakedStructs.Add(new(gameObject.name, resolution, viewProjMatrix, viewProjMatrix.inverse,
    depthPixels, albedoPixels, normalPixels, trueNormalPixels, maskPixels, emissionPixels));
```

Figure 11. Write pixel data to structure.

## 2.3. Channel Interpolation Using Blending Factor

### 2.3.1. Calculating Blending Factor

After rendering and storing Camera Textures, the key difficulty lies in how to read the texture information in Camera Textures when rendering the interspersed objects on the terrain and blend it with the PBR channels of the interspersed objects themselves.

First, the matrix method is used, a technology to restore world coordinates from depth. From the code side, the inverse matrix of the main camera's view projection matrix (Inversed View and Projection Matrix) is set as the global matrix of the Shader.

Unity provides the useful function, it's shown in Figure 12.

```
void ComputeWorldSpacePosition_float(float2 positionNDC, float deviceDepth, float4x4 invViewProjMatrix, out float3 positionWS)
{
    positionWS = ComputeWorldSpacePosition(positionNDC, deviceDepth, invViewProjMatrix);
}

void ComputeWorldSpacePosition_half(half2 positionNDC, half deviceDepth, half4x4 invViewProjMatrix, out half3 positionWS)
{
    positionWS = ComputeWorldSpacePosition(positionNDC, deviceDepth, invViewProjMatrix);
}
```

Figure 12. Unity provides the compute world space position function.

The effect of restoring world coordinates from depth using the matrix method is shown in Figure 13.

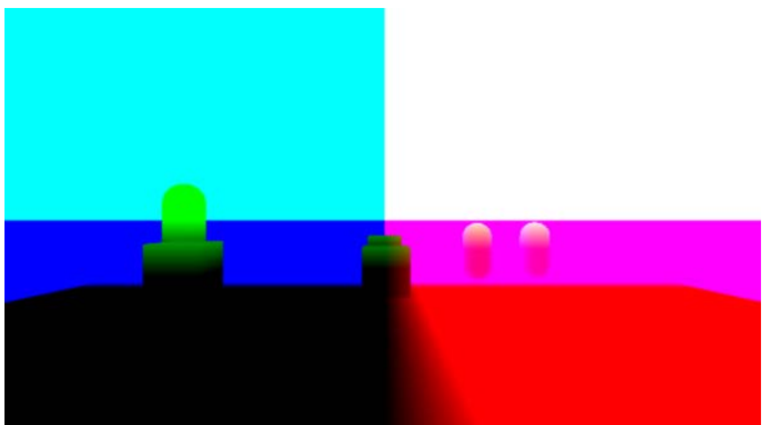

Figure 13. Compute world position from depth.

Thereafter, the depth cache in Camera Textures can be directly transformed into world coordinates. Then, the distance between this world coordinate and the world coordinate of the fragment pixel of the interspersed object itself is subjected to a Smoothstep algorithm to obtain a blending factor with a value range of 0 to 1.

By using the shader graph, the calculation of blending factor is shown in Figure 14.

The blending factor can be presented in color, as shown in Figure 15.

With this blending factor, linear interpolation (Lerp) algorithm can be used to blend the albedo, roughness, metallic, and ambient occlusion.

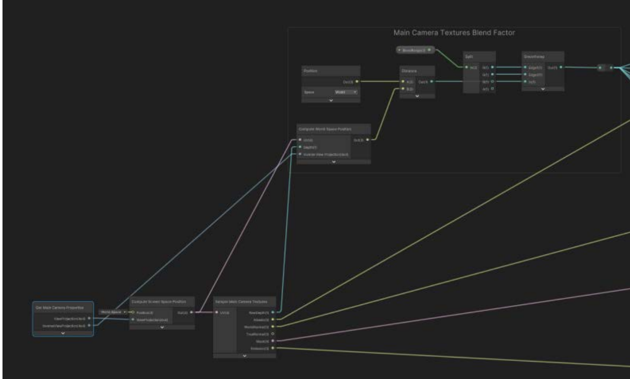


Figure 14. Calculation of blending factor.

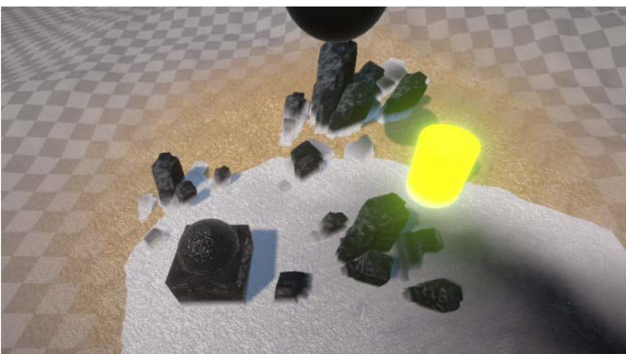

Figure 15. Show of blending factor.

### 2.3.2. Normal Interpolation

In particular, for normals, special algorithms such as Spherical Interpolation (Slerp) can be used to maintain the characteristic that their length is always 1 [6].

The slerp function is shown in Figure 16.

```
float3 FastOrthogonal(float3 v, bool normalize = true)
{
    float sqr = v.x * v.x + v.y * v.y;
    // (0,0,1) x (x,y,z)
    if (sqr > 0)
    {
        float im = normalize ? 1.0 / sqrt(sqr) : 1.0;
        return float3(-v.y * im, v.x * im, 0);
    }
    // (1,0,0) x (x,y,z)
    else
    {
        sqr = v.y * v.y + v.z * v.z;
        float im = normalize ? 1.0 / sqrt(sqr) : 1.0;
        return float3(0, -v. z* im, v. y* im);
    }
}

float3x3 QuaternionMatrix(float4 q)
{
    return float3x3(
    1 - 2 * q.y * q.y - 2 * q.z * q.z, 2 * q.x * q.y - 2 * q.z * q.w, 2 * q.x * q.z + 2 * q.y * q.w,
    2 * q.x * q.y + 2 * q.z * q.w, 1 - 2 * q.x * q.x - 2 * q.z * q.z, 2 * q.y * q.z - 2 * q.x * q.w,
    2 * q.x * q.z - 2 * q.y * q.w, 2 * q.y * q.z + 2 * q.x * q.w, 1 - 2 * q.x * q.x - 2 * q.y * q.y);
}

float3x3 AxisAngle(float3 axis, float rad)
{
    rad /= 2;
    axis *= sin(rad);
    float4 q = float4(axis.x, axis.y, axis.z, cos(rad));

    return QuaternionMatrix(q);
}
```

```
float3 Slerp(float3 lhs, float3 rhs, float t)
{
    float kPI = 3.1415926;

    float lhsMag = length(lhs);
    float rhsMag = length(rhs);

    if (lhsMag == 0 || rhsMag == 0)
        return lerp(lhs, rhs, t);

    float lerpedMagnitude = lerp(lhsMag, rhsMag, t);

    float d = dot(lhs, rhs) / (lhsMag * rhsMag);
    // direction is almost the same
    //if (d == 1.0)
    if (d >= 0.99999)
    {
        return lerp(lhs, rhs, t);
    }
    // directions are almost opposite
    //else if (d == -1.0)
    else if (d <= -0.99999)
    {
        float3 lhsNorm = lhs / lhsMag;
        float3 axis = FastOrthogonal(lhsNorm);
        float3x3 m = AxisAngle(axis, kPI * t);
        float3 slerped = mul(m, lhsNorm);
        slerped *= lerpedMagnitude;
        return slerped;
    }
    // normal case
    else
    {
        float3 axis = cross(lhs, rhs);
        float3 lhsNorm = lhs / lhsMag;
        axis = normalize(axis);
        float angle = acos(d) * t;

        float3x3 m = AxisAngle(axis, angle);
        float3 slerped = mul(m, lhsNorm);
        slerped *= lerpedMagnitude;
        return slerped;
    }
}
```

Figure 16. Slerp function.

The effect of spherical interpolation is shown in Figure 17.

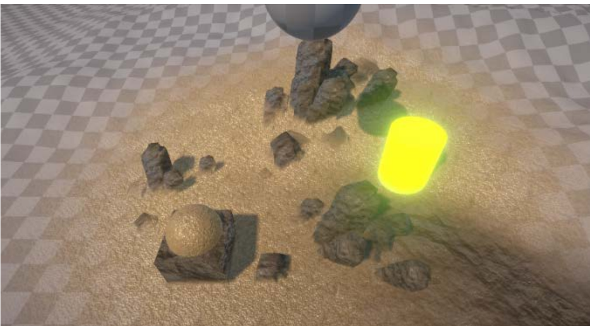

Figure 17. Slerp normal result.

#### 2.3.3. Using Noise to Perturb Blending Factor

By perturbing the blending factor with procedural noise or texturing, different blending effects can be achieved, as shown in Figure 18 and Figure 19.

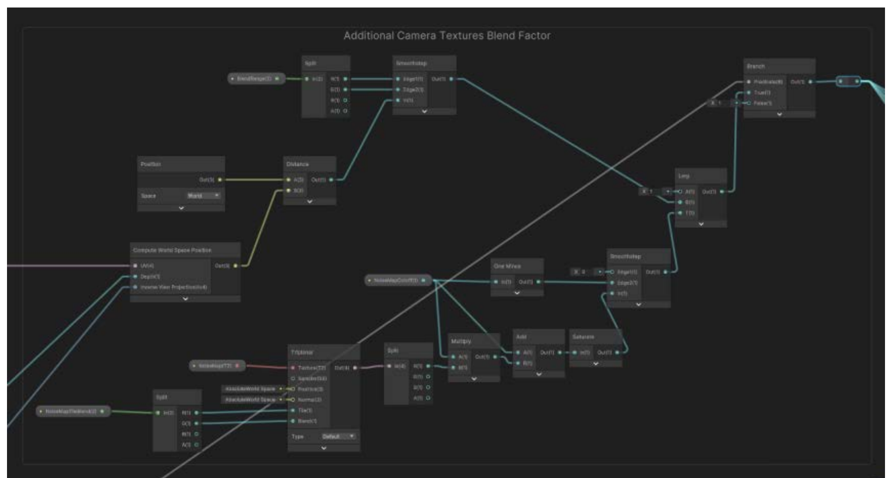


Figure 18. Perturb blending factor.

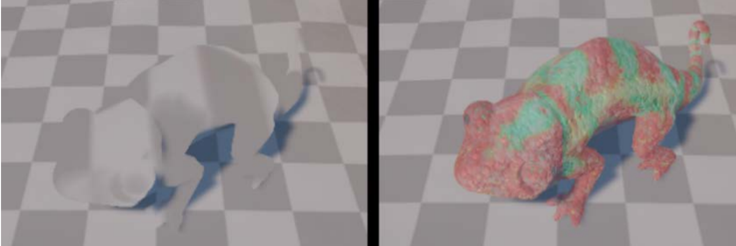

Figure 19. Perturb blending factor result.

### 2.4. Scene Baking Using Additional Cameras

The blending effect mentioned above is always based on the perspective of the main camera. With the help of another medium, that is, using a cache to store Camera Textures, different details in the scene can be baked, that is, Camera Textures of the Additional Camera are stored in hardware.

### 2.4.1. Directional Additional Camera

The Directional Additional Camera is responsible for pre-rendering textures containing depth, albedo, normal, roughness, metallic, and ambient occlusion in the position of the scene requiring high details through scripts in the editor development stage, then transcribing them into the cache, and after storing the required matrix of the additional camera at the same time, packaging and storing them in the asset. The asset structure is shown in Figure 20.

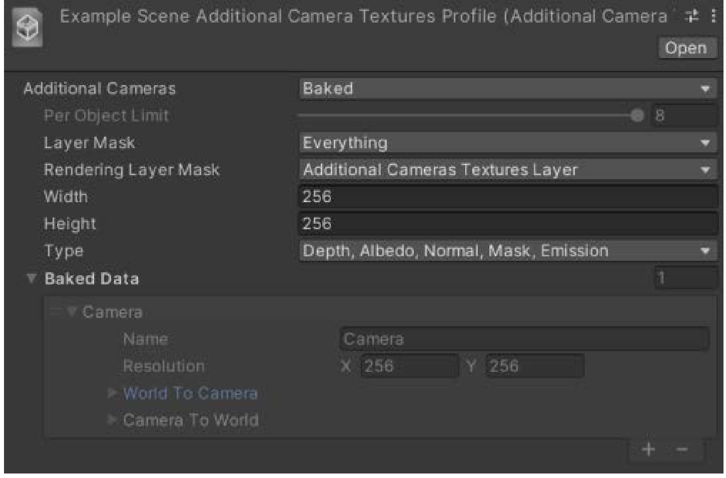


Figure 20. Asset used by additional camera.

In the rendering of interspersed objects, after blending Camera Textures of the main camera, Camera Textures of the additional camera need to be blended. The sequence is shown in Figure 21.

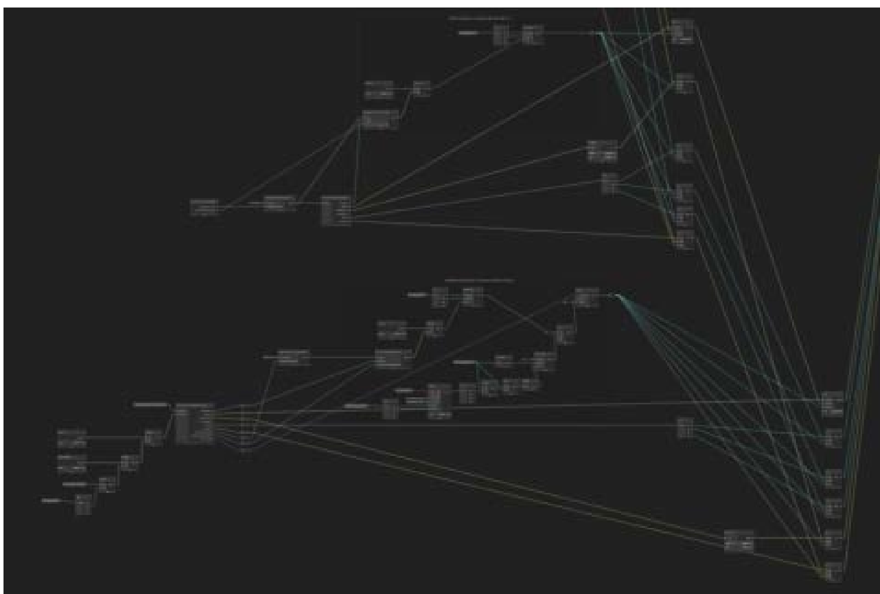

Figure 21. Blending additional camera after main camera.

Due to the use of cache as the data medium, there is an upper limit to the data

size of a single cache. At the same time, considering performance reasons, the upper limit of the number of additional cameras that can be mixed by a single object is tentatively set to 8.

Transform a large cache at once, can make unity editor crashed. The error it gives is shown in Figure 22.

```
[15:17:08] OutOfMemoryException: Out of memory
System.Collections.Generic.List`1[T].set_Capacity (System.Int32 value) (at <8ce0bd04a7a04b4b9395538239d3fdd8>:0)

OutOfMemoryException: Out of memory
System.Collections.Generic.List`1[T].set_Capacity (System.Int32 value) (at <8ce0bd04a7a04b4b9395538239d3fdd8>:0)
System.Collections.Generic.List`1[T].EnsureCapacity (System.Int32 min) (at <8ce0bd04a7a04b4b9395538239d3fdd8>:0)
System.Collections.Generic.List`1[T].InsertRange (System.Int32 index, System.Collections.Generic.IEnumerable`1[T] collection) (at
<8ce0bd04a7a04b4b9395538239d3fdd8>:0)
System.Collections.Generic.List`1[T].AddRange (System.Collections.Generic.IEnumerable`1[T] collection) (at
<8ce0bd04a7a04b4b9395538239d3fdd8>:0)
```

Figure 22. Out of memory.

A table of the test on different cache size, by bake different resolution of additional camera, is shown in Table 1. Both them baked 8 camera per-object.

Table 1. Test of cache size.

| 8 × 256 px × 256 px | 8 × 512 px × 512 px | 8 × 1028 px × 1028 px | 8 × 4096 px × 4096 px |
|---|---|---|---|
| 0.3s | 1s | 15s | Crashed |

The successful mixing effect is shown in Figure 23.

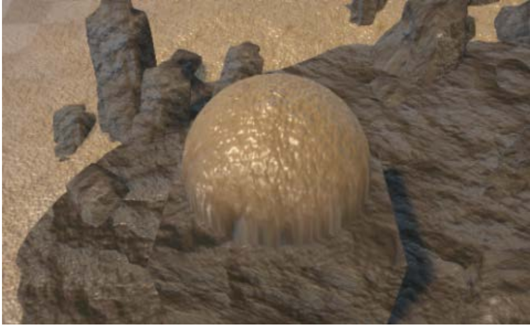

Figure 23. Directional additional camera result.

### 2.4.2. Omni Additional Camera

In the previous text, the additional camera only bakes with a fixed-direction perspective camera. The Omni Additional Camera bakes the full perspective of the upper, lower, left, right, front, and rear six sides at the same time with a 45-degree Field of View (FOV). Inspired by Unity's Reflection Probe technology [7], it does not bake the final rendering result but the various channel information used for lighting calculation. The effect of the omni additional camera is shown in Figure 24.

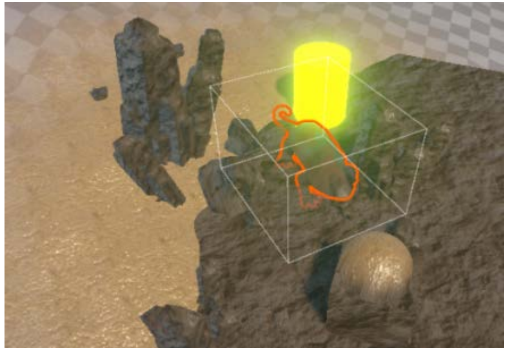

Figure 24. Omni additional camera result.

The omni additional camera uses the channels of the additional camera mentioned above in the rendering stage of interspersed objects. The omni additional camera is essentially six directional special additional cameras, and a single object occupies the number of six additional cameras that can be mixed at the same time. The omni additional camera bakes code can be shown in Figure 25.

```
if (directionMode == DirectionMode.Omni && Execute(out resolution, out RTHandle[] depthClips, out RTHandle[] albedoClips, out RTHandle[] normalClips,
    out RTHandle[] trueNormalClips, out RTHandle[] maskClips, out RTHandle[] emissionClips, out Matrix4x4[] viewProjMatrices))
{
    Texture2D tempClip = new(resolution.x, resolution.y, GraphicsFormat.R32G32B32A32_SFloat, TextureCreationFlags.None);

    Color[] depthPixels = null, albedoPixels = null, normalPixels = null, trueNormalPixels = null, maskPixels = null, emissionPixels = null;

    for (int face = 0; face < 6; face++)
    {
        if (depthClips[face] != null)
        {
            RenderTexture.active = depthClips[face];
            tempClip.ReadPixels(new(0, 0, resolution.x, resolution.y), 0, 0);
            depthPixels = tempClip.GetPixels(0);
            RenderTexture.active = null;
            depthClips[face].Release();
        }
        if (albedoClips[face] != null)
        {
            RenderTexture.active = albedoClips[face];
            tempClip.ReadPixels(new(0, 0, resolution.x, resolution.y), 0, 0);
            albedoPixels = tempClip.GetPixels(0);
            RenderTexture.active = null;
            albedoClips[face].Release();
        }
        if (normalClips[face] != null)
        {
            RenderTexture.active = normalClips[face];
            tempClip.ReadPixels(new(0, 0, resolution.x, resolution.y), 0, 0);
            normalPixels = tempClip.GetPixels(0);
            RenderTexture.active = null;
            normalClips[face].Release();
        }
        if (trueNormalClips[face] != null)
        {
            RenderTexture.active = trueNormalClips[face];
            tempClip.ReadPixels(new(0, 0, resolution.x, resolution.y), 0, 0);
            trueNormalPixels = tempClip.GetPixels(0);
            RenderTexture.active = null;
            trueNormalClips[face].Release();
        }
        if (maskClips[face] != null)
        {
            RenderTexture.active = maskClips[face];
            tempClip.ReadPixels(new(0, 0, resolution.x, resolution.y), 0, 0);
            maskPixels = tempClip.GetPixels(0);
            RenderTexture.active = null;
            maskClips[face].Release();
        }
        if (emissionClips[face] != null)
        {
            RenderTexture.active = emissionClips[face];
            tempClip.ReadPixels(new(0, 0, resolution.x, resolution.y), 0, 0);
            emissionPixels = tempClip.GetPixels(0);
            RenderTexture.active = null;
            emissionClips[face].Release();
        }

        bakedStructs.Add(new(gameObject.name, resolution, viewProjMatrices[face], viewProjMatrices[face].inverse,
            depthPixels, albedoPixels, normalPixels, trueNormalPixels, maskPixels, emissionPixels));
    }
}
```

Figure 25. Omni additional camera's code.

# 3. Conclusion

## 3.1. Conclusion about New Technologies

First, the concept of Camera Textures is proposed, and a multi-channel lighting

information blending framework is constructed. Inspired by deferred rendering, for the first time, the core channels required for lighting calculation of terrain and objects, such as depth, albedo, normal, roughness, metallic, and ambient occlusion, are uniformly stored in the view-space texture set to form a standardized lighting information interaction medium. A cross-object lighting information sharing mechanism is constructed, so that the material properties of the terrain and objects in the junction area, such as roughness differences and normal direction mutations, can be quantitatively calculated and mixed using blending factors, laying a data foundation for "physically real" blending.

Second, the dynamic calculation method of blending factors based on matrix transformation realizes the intelligent connection of geometry and materials. By inferring world coordinates from depth textures and combining the distance metric of the object's own fragment coordinates, adaptive blending weights are generated to replace traditional manual parameter adjustment or fixed-function mixing. It does not require pre-baked masks or rely on the engine's built-in layer system, can respond in real time to dynamic interactions (such as object movement and terrain deformation), and automatically generates blending boundaries that conform to the geometric distance attenuation law, significantly reducing manual editing costs.

Again, the "additional camera" and "omni additional camera" systems solve the problem of blending accuracy for extreme perspectives and complex details. A solution to balance cache data size and performance is proposed. A single object can be mixed with up to 8 additional cameras, breaking through the traditional single-viewpoint dependence and providing a universal solution for the omni-directional seamless connection between dynamically rotating objects, such as rotatable mechanical devices, and terrain.

Then, while the research has made the above progress, there are still some areas for improvement. For example, the upper limit of the number of additional cameras that can be mixed by a single object is still 8. It is believed that with the development of hardware devices and further iterations, the problem of cache transmission scale will also be solved.

In summary, this study focuses on the core problem of transitional blending between terrain and objects. Through interdisciplinary technology integration and algorithm innovation, the universality and portability of terrain blending technology are achieved. It has made multiple contributions at the theoretical method, technical implementation, and engineering application levels.

### 3.2. Performance Comparison

A performance comparison is provided. All results are based on the following configurations.

13th Gen Intel(R) Core(TM) i9-13900HX

NVIDIA GeForce RTX-4070 Laptop GPU

RAM 64.0GB

The test scene is shown in Figure 26.

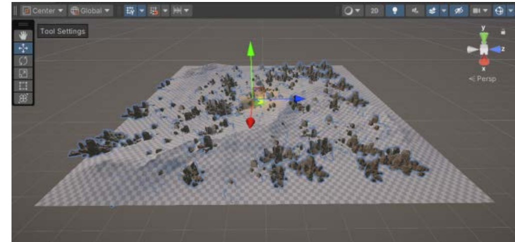

Figure 26. Test scene.

First, a table of the new technologies performance, includes platform in Unity Editor and on Windows Standalone, is shown in Table 2.

Table 2. Performance comparison on different platform.

| Platform | FPS | Memory Usage | GPU Cost |
|---|---|---|---|
| Unity Editor | 110 | 25.1/63.7 GB (37%) | 0% |
| Windows Standalone | 160 | 20.8/63.7 GB (33%) | 8% |

Second, the test scene' Statistics in Unity Editor is shown in Figure 27.

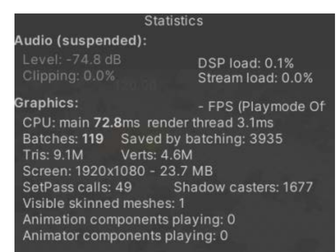


Figure 27. Statistics in unity editor.

Third, the test scene's performance on Windows Standalone is shown in Figure 28.

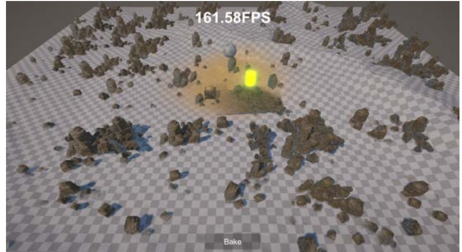


Figure 28. Performance on windows standalone.

## Acknowledgements

Design and Implementation of Immersive Experience and VR Technology Application Interactive Software with the Theme of Starry Sky Culture (Project No.: 22150725062).

## Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

## References

[1] Hillaire, S. (2016) Physically Based Rendering in Real-Time. ACM SIGGRAPH Courses.

[2] Karis, B. (2013) Real Shading in Unreal Engine 4. SIGGRAPH Presentation.

[3] Epic Games (2020) Virtual Texturing in Unreal Engine 4. Technical Documentation. Unreal.

[4] Unity Technologies (2023) Shader Graph: Visual Shader Editing Tool. Unity Manual.

[5] McGuire, M. (2017) Computer Graphics Techniques in Games. CRC Press.

[6] Lagarde, S. (2014) Spherical Gaussian Encoding for Efficient Rendering. *Journal of Computer Graphics Techniques* (*JCGT*), **3**, 1-30.

[7] Unity Technologies (2022) Reflection Probes Technical Guide. Unity Blog.