Simulation of Desert Scenes Based on the URP Pipeline

2024 年 9 月 25 日

1 Abstract

This paper presents our work on enhancing visual effects in a desert environment using Unity's Universal Render Pipeline (URP). By adding custom functionalities to the URP, we aim to simulate realistic desert phenomena. Our approach includes reflective effect generating, realistic footprint simulation, and volumetric lighting with post-processing optimizations. These enhancements improve the visual quality and realism of scenes rendered in real-time, making them suitable for applications in gaming and interactive media.

2 Reflective Effect of Sand

Unity already has an EmissionMap, but its effect is somewhat different from what we desire. Therefore, I made some simple modifications to Unity' s Emission in the Material and Surface Shading stage, based on the URP rendering pipeline's Lit Shader, to achieve the effect of gravel sparkling under the sunlight.

2.1 Generating and Utilizing White Noise Emission Textures in Unity

In Unity, apart from simply controlling the emission effects of materials through uniform color and brightness settings, it is also possible to assign an emission map to this parameter. The emission color and brightness are controlled by the full color values of the texture map. Therefore, we first use Python to generate an image where every pixel is randomly assigned 0 or 1 and save it as the white noise image to serve as our desired emission texture.

2.2 Enhancing Emission Control in URP with Threshold and Range

In our URP process, we added two parameters to the Emission properties: EmissionThreshold and EmissionRange, which control the quantity and size of the glints, respectively.

In the SampleEmission function, we adjust the emission by sampling a texture channel for emission intensity and comparing it against a threshold to decide whether to apply the effect. This allows dynamic control of the object's emission based on the texture and threshold.

In the LitPassFragment function of the LitForward part, we calculate the dot product of the normal vector and the view direction to determine their alignment. We then raise this value to a power controlled by _EmissionPow, which adjusts the emission intensity based on the angle. When the view direction aligns closely with the normal, the emission remains bright; otherwise, it quickly fades. This processed value is applied across the RGB channels and multiplied by the original emission color.

Through this method, the code effectively restricts the area of light point display, making light points prominent only when the viewing angle is sufficiently close to the object's surface normal, thus achieving a view-dependent display effect.

图 1: Sand Reflection

3 Interaction of Footprints in the Desert

In this section, we will implement a technique to simulate realistic footprints in a desert environment. This involves using surface subdivision to increase the level of detail and vertex displacement to create the depressions and elevations representing footprints.

3.1 Surface Subdivision

To determine how to tessellate a triangular surface, we set four tessellation factors for the GPU: three for each edge of the triangle and one for its interior, thereby determining the level of detail of the patch. The main task of the Hull Shader is to subdivide the input patch into smaller patches based on the tessellation factors and to output the information of these control points. The Hull Shader is invoked multiple times, processing one control point at a time and outputting the new control point data. Our output is configured based on the vertex structure of the LitShader to serve as input for the tessellator.

3.2 Vertex Displacement

First, we use a RenderTexture to record the height variations of the character's footprints. Using a particle system, particles are emitted as the character moves, and color coding (red for depressions and green for

elevations) is used to record the footprints. Additionally, an extra camera is used to render these footprints onto the RenderTexture, and the camera's position and size are recorded.

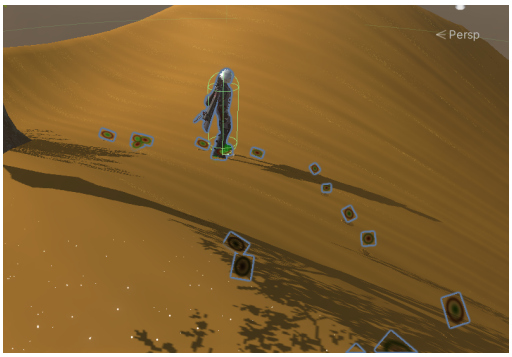

图 2: Red and Green Channel

In the Domain Shader after tessellation, the subdivided vertex positions and related attributes are calculated through interpolation methods. Barycentric coordinates are used to interpolate the position, normals, UVs, and other attributes of each subdivided vertex. Then, the UV coordinates of the vertices in the camera view are calculated, enabling the display of footprints within a certain range around the character.

By sampling the red and green channel information recorded in the RenderTexture as multiplication coefficients and using a noise texture to introduce subtle variations, the height offset for each vertex is obtained. The NormalFromHeight function (reused from ShaderGraph) is called to update the surface normal vectors based on the calculated height offsets, ensuring the normals reflect the terrain or surface detail changes.

After calculating the offset, it is converted into offset vectors in world coordinates and object coordinates. The offset vector is calculated based on the normal direction and then converted into object and world coordinate systems. These offset vectors are then added to the interpolated vertex positions to obtain the final displaced vertex positions.

In the fragment shader, the updated vertex positions and normal information are used to complete the final lighting and rendering calculations.

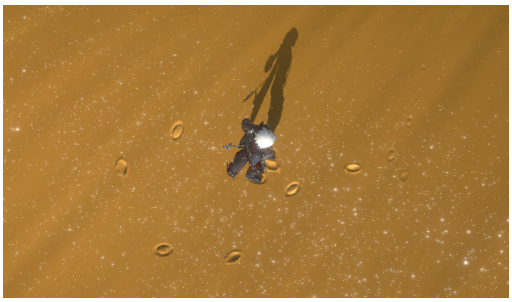

图 3: Footprints

4 Volumetric Light

In a game, when an object blocks light from a light source, the radiative leakage of light around it is referred to as volumetric lighting. In a desert, when the wind blows and light passes through the dust particles lifted by the wind, volumetric lighting is also produced. In this section, we implement volumetric light using ray marching, followed by post-processing optimization.

4.1 RayMarching

First, we need to reconstruct world space coordinates from screen space to determine if they will be affected by light. Therefore, we convert screen space coordinates to NDC coordinates, use the camera's inverse projection matrix to transform NDC coordinates into view space coordinates, and finally, apply the camera's world transformation inverse matrix to convert view space coordinates into world space coordinates.

Now we can proceed with ray marching in the fragment shader. We define the starting point of ray marching directly at the camera's position. The end position minus the starting position gives us the direction for marching, and we stop when obstructed or upon reaching the maximum distance. We start the ray marching loop, accumulating light intensity based on the set maximum number of steps and maximum marching distance. Ultimately, the calculated light intensity is combined with the camera's color.

Additionally, we need to make a small judgment on whether or not to add light intensity. We use a simple method to determine if the position is within an object's shadow. If it is in the shadow, the added intensity will be zero.

4.2 Gaussian blur[1]

Although the above method is simple, it also has some issues. For example, if the step size is too large, it can easily lead to a layering effect in the light, and if the step size is too small, it requires a significant amount of computation.

Therefore, we first render the volumetric light, then request a Render Texture (RT) to store this volumetric light RT. Next, we place the volumetric light RT into a Gaussian blur function to apply the blur, and finally output it to the screen. To expedite the implementation, we use a 3x3 Gaussian kernel convolution during the Gaussian blur process. The configuration of the Gaussian kernel is:

- Center pixel: 0.147761
- Direct adjacent pixels (4): 0.118318
- Diagonal adjacent pixels (4): 0.0947416

In this way, each pixel on the image is replaced by the weighted average color value of its surrounding pixels, thus achieving the blurring effect. However, applying Gaussian convolution to the entire screen can lead to a loss of detail. Therefore, we extract the luminance values of each part of the image and compare these with the original image's luminance to make decisions about color retention. Blurring is only permitted where there are significant differences in brightness.

4.3 Rayleigh Scattering[2]

In this section, we further optimize its performance. In the last section, we applied a linear attenuation to the intensity of light, which is clearly not

quite correct. Now, we discuss the energy loss during the propagation of light, using Rayleigh scattering as an example for implementation

First, we assume that scattering does not occur in a vacuum, meaning light does not lose energy; therefore, the intensity of light outside the atmosphere remains constant. When the light reaches point C, its intensity is at its maximum value, which we define as I_c; after attenuation, it reaches point P, where the intensity of light is I_p. At this point, our transmittance $T(C \to P)$ is defined as $\frac{I_p}{I_c}$ which leads to $I_p = T(C \to P) \times I_c$.[3] Since the step distance is very short, we here set $T(C \to P)$ as a constant 1.

Therefore, we obtain the first equation:

$$I_P = I_C \cdot T(\overline{CP})$$

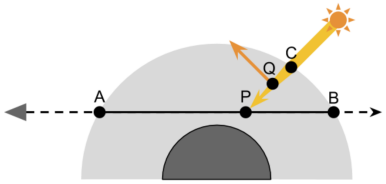

图 4: Light Refraction and Scattering

When the light reaches point P and scattering occurs, we need to know the intensity of the scattered light in the direction of the line of sight. Therefore, we introduce the scattering function[4]

$$S(\lambda, \theta, h)$$

where:

- λ: visible light wavelength

- θ: the angle between the direction of the scattered light and the line of sight (unity indicates the light source direction)

- h: height above sea level

The specific form of the S function changes according to different scattering types. For Mie scattering, it mainly depends on geometric characteristics rather than wavelength. Here, we study Rayleigh scattering, which is the scattering caused by small particles. The specific function is as follows:

$$S(\lambda, \theta, h) = \frac{\pi^2 (n^2 - 1)^2}{2} \frac{\rho(h)}{N} \frac{1}{\lambda^4} \left(1 + \cos^2 \theta\right)$$

where:

- $n = 1.00029$: refractive index
- $\rho(h) = 2.504 * 10^{25}$: density at height h
- $N = 8400$: number density of particles

At this point, we know the scattered intensity of light along the line of sight. Then, the light continues to travel forward to point A. The process is represented as follows:

$$I_{PA} = I_P S(\lambda, \theta, h) T(\overline{PA})$$

We substitute the first equation into this, noting that the function S will be discussed later:

$$I_{PA} = I_C T(\overline{CP}) S(\lambda, \theta, h) T(\overline{PA})$$

There are countless points like p along our line of sight. The illumination intensity we need is the total illumination at such points p.

The overall idea should now is clear. Next, we will explain the function S and function T in detail.

First, let's look at the function S. S only accounts for the light scattering in a specific direction and does not consider the total scattering at a point. Therefore, we perform spherical integration at point p.

$$\beta(\lambda, h) = \int_0^{2\pi} \int_0^{\pi} S(\lambda, \theta, h) \sin\theta \, d\theta d\phi$$

The final integration result is

$$\beta(\lambda, h) = \frac{\pi^2 (n^2 - 1)^2}{2} \frac{\rho(h)}{N} \frac{1}{\lambda^4} \int_0^{2\pi} \int_0^{\pi} \frac{8}{3} \theta \, d\theta d\phi$$

$$\beta(\lambda, h) = \frac{\pi^2 (n^2 - 1)^2}{2} \frac{\rho(h)}{N} \frac{1}{\lambda^4} \frac{16\pi}{3}$$

Since our step distance is relatively short, height has little effect. To save space, we use constant substitution in the code.

Another related function is γ, which is used to determine the laws of constant scattering.

$$\gamma(\theta) = \frac{S(\lambda, \theta, h)}{\beta(\lambda)}$$

$$= \frac{3}{16\pi}\left(1 + \cos^2\theta\right)$$

Next, we discuss the attenuation T.

From the above, we know that the energy of light decreases by a factor of $(1 - \beta)$ each time it scatters.

So,

$$I_1 = I_0(1 - \beta)$$

After traveling a certain distance, how do we decompose it?

For the convenience of calculation, we write $(1 - \beta)$ as the product of two terms $\left(1 - \frac{\beta}{2}\right)$.

So,

$$I_1 = I_0\left(1 - \frac{\beta}{2}\right)\left(1 - \frac{\beta}{2}\right)$$

$$I_2 = I_1\left(1 - \frac{\beta}{2}\right)$$

After the next attenuation?

$$I_n = I_0\left(1 - \frac{\beta}{n}\right)^n$$

Combining this with the limit $\left(1 + \frac{1}{x}\right)^x = e$, we have

$$I_n = e^{-\beta}$$

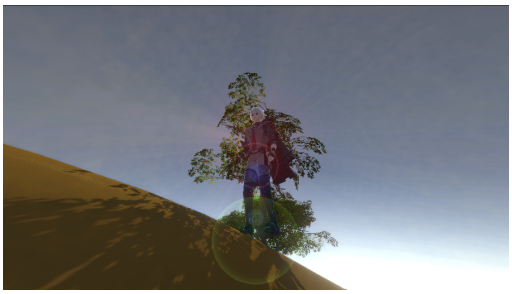

图 5: volumetric light effects

5 Future Work

There are several areas for improvement and expansion in our project that we aim to address in future work:

1. Currently, the character controller sometimes struggles to accurately determine whether the character's feet are in contact with the ground on steep slopes, resulting in a lack of footprints. Despite testing various character control packages, we have not yet found a solution that effectively addresses this issue. Future work will involve refining the character control system to ensure reliable footprint generation on all terrain types, including steep slopes.

2. Our volumetric lighting implementation create realistic volumetric lighting all the time, which is great when you look through leaves. However, similar effects are also generated when the character stands on open ground, which is not realistic. This occurs because we have not yet completed the effects for wind-blown sand. As a result, volumetric lighting appears in open spaces where it should not. Future work will focus on developing and integrating realistic wind-driven sand effects.

6 Resources Used and Collaborations

1. We used the "Genshin Impact - Wriothesley" character model from Sketchfab.

2. The "Invector Third Person Controller" from the Unity Asset Store was used to control the character.

3. The "HornedHolly Tree" from the Unity Asset Store was used to generate tree models.

Benhou Li is responsible for the implementation in section2 and section3.

Peiyang Zheng is responsible for the implementation in section4.

参考文献

[1] TutorialExample. (n.d.). *Understand Gaussian Blur Algorithm: A Beginner Guide.* Retrieved from https://www.tutorialexample.com/understand-gaussian-blur-algorithm-a-beginner-guide-deep-learning-tutorial/

[2] 冯乐乐. (2021, January 13). *unity_ 游戏中的大气散射和雾效 Unity 中实现推导*. CSDN. Retrieved from https://blog.csdn.net/weixin$_2$9315091/*article*/*details*/113331020

[3] 命运是沉睡的奴隶. (2021, March 29). *Unity* 大气散射实现（一*)*. 知乎专栏. Retrieved from https://zhuanlan.zhihu.com/p/360832044

[4] Zucconi, A. (2017, October 10). *Volumetric Atmospheric Scattering.* Retrieved from https://www.alanzucconi.com/2017/10/10/atmospheric-scattering-1/