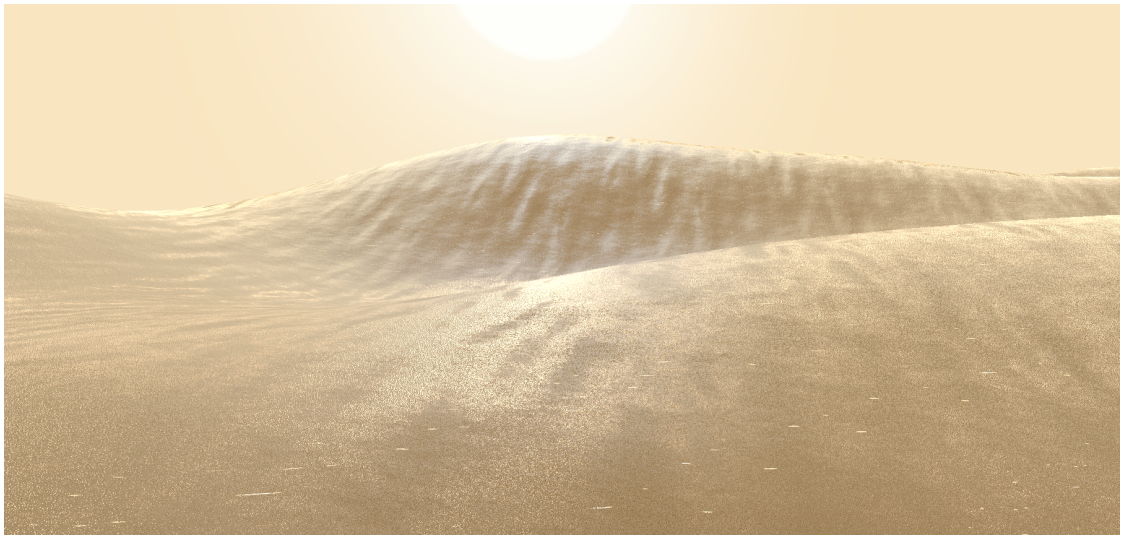
Glittery Sand/Snow Shader

Inspired By Journey

Written by

Daniel Eckmann
daec@itu.dk



Graphics Programming 2022
KGGRPRG1KU

Date: 03/06/2022

IT University of Copenhagen
2022

Table of Contents

Introduction

For my project, I have attempted to create a shader that could be used for fine, grainy materials, such as sand or snow, to give them a fluid-like, sparkly, artistic look. It is inspired by the sand shader seen in *Journey* (Sony Computer Entertainment, 2012), which was explored in detail by the game's lead engineer, John Edwards, in his 2013 GDC talk (Edwards & Thatgamecompany, 2013).

Theory

The shader setup uses *deferred rendering*, where lighting calculation happens separately, after the rendering of the geometry (LearnOpenGL - Deferred Shading, n.d.). This is especially beneficial in scenarios where multiple light sources are being calculated, since we are only computing lighting data on surfaces that are actually in sight. I chose this method because in some areas *Journey* featured segments that had many light sources present, and for those use cases deferred rendering would be a more ideal choice than forward rendering.

During the geometry pass, the shader processes the geometry attributes and renders the collected data into the following G-buffers:

- Albedo Buffer: stores the colour data from the objects' textures
- Normal Buffer: stores the normal data (in worldspace coordinates)
- Others Buffer: stores physical properties of objects (roughness and metalness)
- Depth Buffer: stores the depth data of all geometry in the scene

The lighting pass calculates the lighting data based on the received buffer textures prepared during the geometry pass (more on the actual methods in chapter *The Layers)*.

After the lighting pass is finished, the data collected during both phases gets stored into an Accumulation Buffer, which passes on the data to either one of the two helper shaders: the *Compose* shader or the *Copy* shader. When post-processing is enabled, the *Compose* shader processes the contents, by applying the post-processing data, before drawing the frame onto the screen. If post-processing is disabled however, the *Copy* shader just straight up copies the contents of the Accumulation Buffer, and draws it one to one.

The Layers

The lighting shader can be broken down into multiple layers. Each adds a different kind of lighting to the overall look and feel. In the following paragraphs, I am going to showcase them separately, and the theories that made them possible.

Diffuse Layer

Calculating the scattered, reflected light is done through a slightly modified implementation of the Lambert reflectance model (Losasso, 2004).
In the default implementation of the Lambertian diffuse, we take the dot product of the model's surface normal vector (*N*) and the normalised light-direction vector (*L*). Multiplying this with the colour of the model (*C*), and an intensity multiplier (*I*) gives us the diffuse reflection.

$$Lambert = N \cdot L * C * I$$

However, *Journey*'s implementation slightly differs in certain ways, and this is the version that I ended up using in my own project as well.

```
N.y *= 0.3;
saturate( 4 * dot( N, L ) );
```

First of all, they shrunk the up (y) axis of the normal vector, then they increased the intensity of the overall diffuse by multiplying the $N \cdot L$ product by 4. The $saturate()$ function of *Cg* (the shading language used during the development of *Journey*) translates to $clamp(x,\ 0.0,\ 1.0)$ in *OpenGL*, which makes sure that the increased value of the dot product remains within the boundaries of 0 and 1.

During the previously mentioned *GDC* talk, they also mentioned that the original plan was to use the *Oren-Nayar diffuse model* (Oren & Nayar, 1995), instead of Lambert. Even though this was discarded because it was computation-heavy for the *PlayStation* at the time, I implemented this one as well to demonstrate the difference (the diffuse model can be switched from the Debug menu).

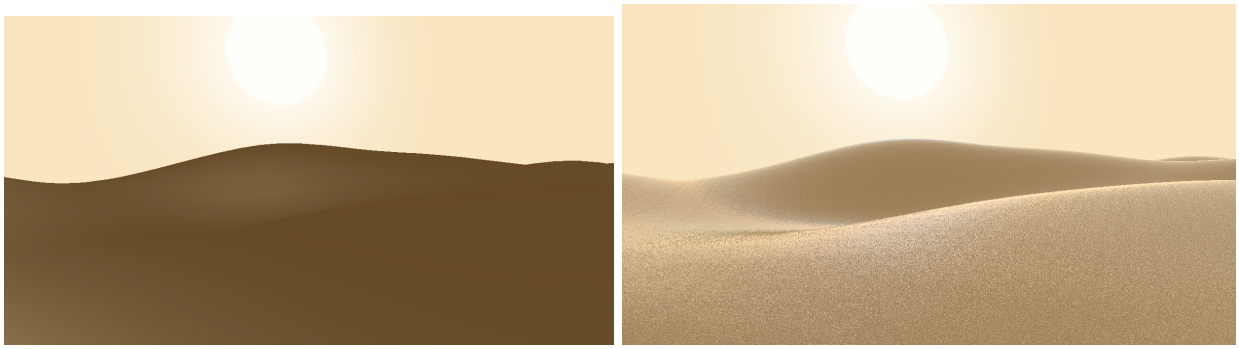
Glint Rendering

To achieve the grainy look sand has in real life, I added a glint layer which emulates this effect via a normal map.

In reality, sand is not as shiny as it looks in *Journey*, but the game went for a somewhat dream-like art direction, so it works perfectly. The normal noise map follows Gaussian normal distribution (Normal Distribution - Encyclopedia of Mathematics, n.d.), where most grains are pointing upwards, the others pointing in equally different, random directions to the sides.

This noisy map helps us simulate a surface that has microfacets, tiny fractions in the material that reflect the light in different directions. Each glint's orientation is determined by their pixel normal.

Adding glint rendering resulted in the look below:



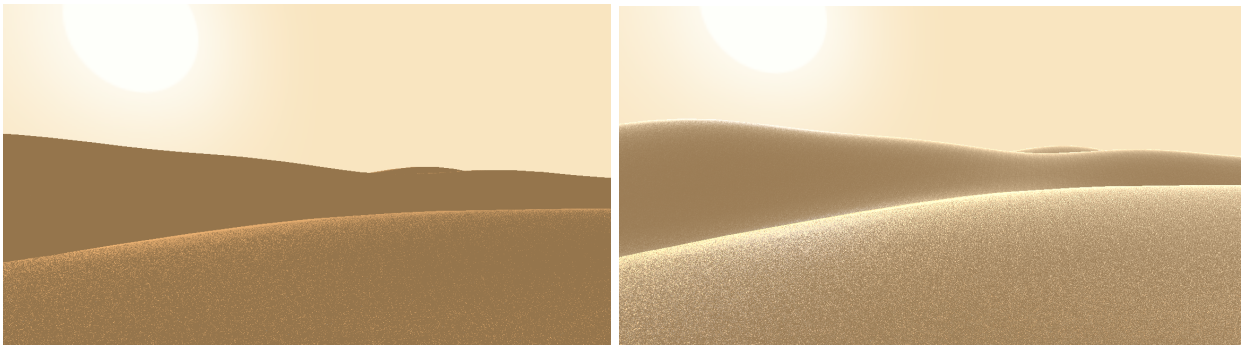
Rim Lighting

Sand dunes in *Journey* have a thin rim light on their top whenever they are being viewed in front of the sun or other light sources, giving them a bit of softness and a more realistic reflective behaviour. To achieve this effect, I have extended the calculation of specular reflection with the *Fresnel factor*. A very common way to do this is by implementing *Schlick's approximation* formula (Lettier, n.d.).

We start out with the unit vectors of the normal (N) and the view direction (V). The product of $N \cdot V$ is *0* when the two vectors are orthogonal ($cos(90°) = 0$), but since that is exactly the angle we want to receive the most light from, we reverse the result by subtracting it from 1. To control the sharpness of this reflection, we introduce the exponent to the equation, and finally multiply the whole with an intensity value to control the overall strength of the reflection.

$$Fresnel = (1 - N \cdot V)^{exponent} * I$$

Adding this layer achieved this look:



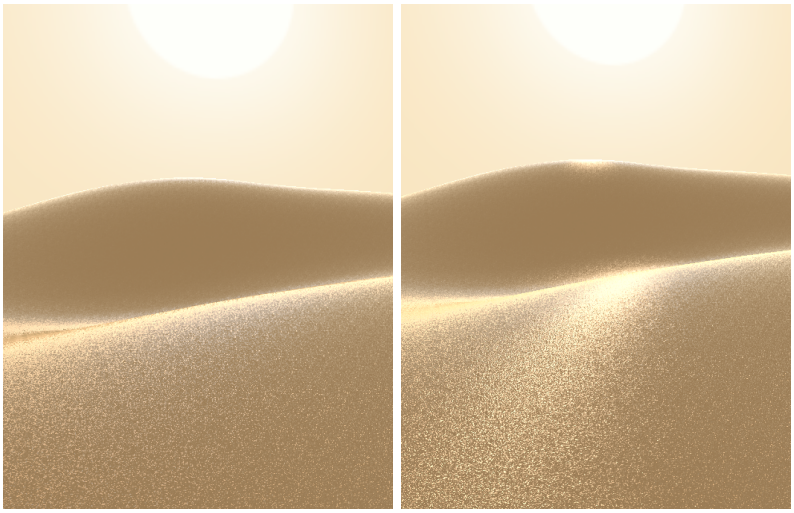
Ocean Specular

This layer creates the path of light beam on the surface leading towards the sun, a phenomenon that is typically observed on large bodies of water, during dawn or dusk. This effect does not happen in the desert or snowy areas in real life, but adding it to my shader strengthened the fluid-like feel of grainy materials that *Journey* was also going for.

To achieve this, I followed the *Blinn-Phong reflection model*'s implementation of specular reflection (LearnOpenGL - Advanced Lighting, n.d.). In the first part of the equation we take the half-vector (H), which is half-way between the view vector (V) and light direction vector (L), and is calculated as $H = normalize(L + V)$. When we take the dot product of the normal vector (N) and H, we make sure to disregard the negative values, as they are not relevant for lighting calculations. This is done through the $max(x, 0)$ function. Similarly to the rim lighting layer, we can control the intensity and strength of this reflection by introducing an exponent and a multiplier respectively.

$$Specular = max(N \cdot H, 0)^{exponent} * I$$

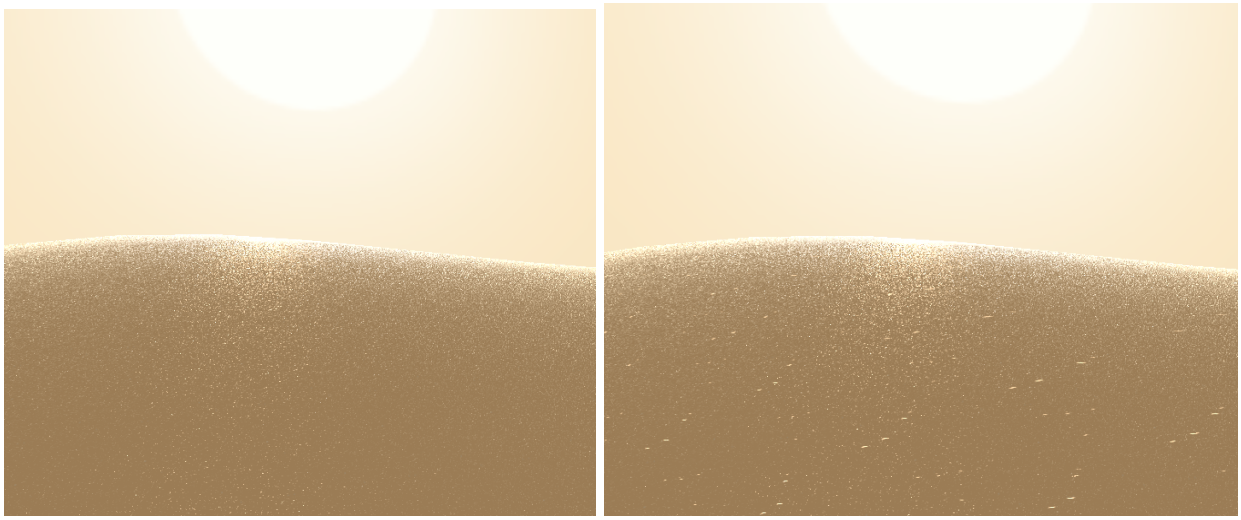
The ocean specular looks like this:



Larger Sparkles

By looking at larger bodies of sand in real life, we can observe that there are some sand particles that appear to shine brighter than the rest. This is due to them facing perfectly towards the viewer. I tried to emulate this effect by sampling a specular map with sparse particles. They are visible in darker areas of the ground, when facing the sun.
To implement this, first I calculate the reflection vector by using the built-in $reflect()$ function. I bounce the inverse of the light direction vector ($-L$) off of the normal (N): $reflect(-L, N)$. This way we reverse the illumination of the specular map, so the parts not lit by sunlight are going to be shining, and the ones covered in light disappear. Similarly to the ocean specular, we calculate the specular light by $max(dot(V, R), 0.0)$, and control its intensity through an exponent and multiplier.
(This feature is turned off on startup, and can be enabled from the Debug menu!)

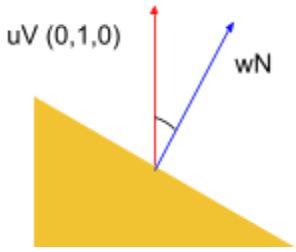
We can see this in action below:



Blended Detail Map

There are two additional normal maps that the shader is sampling on top of the glitter normal. One wavy texture with larger details, and one with smaller ones. These provide the wind-swept look of the dunes.
To compute this, we are using *linear interpolation* (“lerp”) on three axes, which is oftentimes called *spherical interpolation* (“slerp”), a concept explored by Kevin Shoemake (Shoemake, 1985). His formula takes an interpolation parameter between 0 and 1,

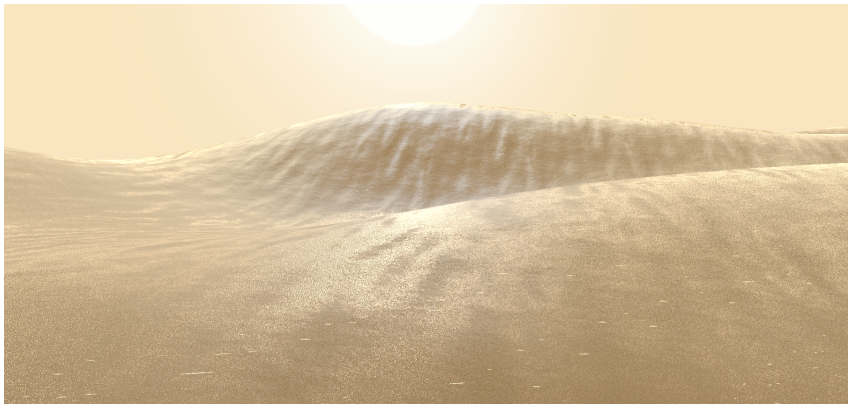
$$\mathrm{Slerp}(p_0, p_1; t) = \frac{\sin\left[(1-t)\Omega\right]}{\sin\Omega} p_0 + \frac{\sin[t\Omega]}{\sin\Omega} p_1.$$



which defines the speed/slope of the interpolation curve. For this parameter, we are using the steepness of the dunes, which can be calculated by taking the dot product of the world normal (wN) and the up vector (uV): $cos(steepness°) = wN \cdot uV$

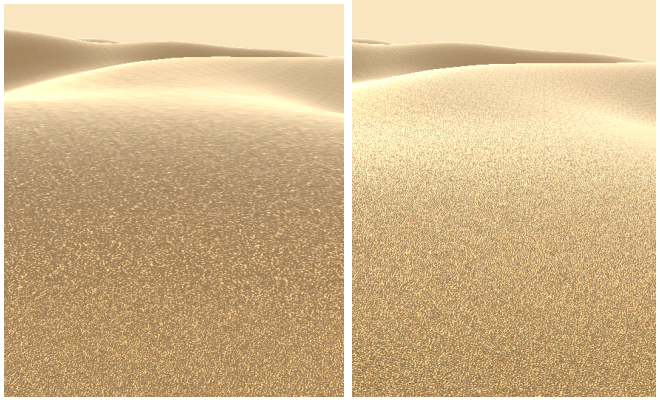
(This feature is turned off on startup, and can be enabled from the Debug menu!)

The blended detail maps look like this:



Additional Technical Feats

- Anisotropic filtering is enabled on all textures, using the $GL\_TEXTURE\_MAX\_ANISOTROPY\_EXT$ parameter. This led to crisper mipmap sampling when viewed from smaller angles:



- Apart from this, additional post-processing effects can be turned on from the Debug menu.

Future Improvements

I would like to iterate on the current implementation of the shader in the future. These further improvements include:

- Using a compute shader to generate the glitter normal map on startup, by implementing some form of Gaussian noise generation formula. A completely randomised noise generation was already part of my project, but it had to be cut due to it being a pure C++ implementation (running on the processor) and taking way too long to generate;
- Implementing the shader inside a commercial game engine, e.g. Unity or Unreal. To use it in my own projects for sand, snow, or even water shading with slight modifications. Adding footstep deformations to create snow tracks, or trails in sand, further inspired by *Journey*;

References

Bibliography

- *Journey* (PlayStation 3 version). (2012). [Video game]. Sony Computer Entertainment. https://thatgamecompany.com/journey/
- Edwards, J. & Thatgamecompany. (2013, March 25–29). *Sand Rendering in Journey* [Conference presentation]. Game Developers Conference, San Francisco, California, United States. https://www.gdcvault.com/play/1017742/Sand-Rendering-in
- *LearnOpenGL - Deferred Shading*. (n.d.). Retrieved June 1, 2022, from https://learnopengl.com/Advanced-Lighting/Deferred-Shading
- Losasso, F. (2004). Surface reflection models. NVIDIA Corporation.
- Oren, M., & Nayar, S. K. (1995). Generalization of the Lambertian model and implications for machine vision. *International Journal of Computer Vision, 14*(3), 227–251. https://doi.org/10.1007/bf01679684
- *Normal distribution - Encyclopedia of Mathematics*. (n.d.). Encyclopedia of Mathematics. Retrieved June 1, 2022, from https://encyclopediaofmath.org/wiki/Normal_distribution
- Lettier, D. (n.d.). *Fresnel Factor*. 3D Game Shaders For Beginners. Retrieved June 1, 2022, from https://lettier.github.io/3d-game-shaders-for-beginners/fresnel-factor.html
- *LearnOpenGL - Advanced Lighting*. (n.d.). Retrieved June 1, 2022, from https://learnopengl.com/Advanced-Lighting/Advanced-Lighting
- Shoemake, K. (1985, July). Animating rotation with quaternion curves. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques* (pp. 245-254).

Used Content

- The project reused some of the code of my solutions to *Exercise 9* that was assigned to us during the semester. This code included the camera, model loader, part of the deferred rendering buffer setup, and the post-processing shader.
- The modified Oren-Nayar implementation is a translation of *Journey*'s cloth shader that I reverse engineered from their Cg implementation.

Manual

The Debug menu can be opened/closed by pressing [Space].
The camera can be moved around with the [W] [A] [S] [D] keys.
The application can be closed by pressing [Esc]

Disclaimer: On some CPU architectures, using the integrated video card can lead to glitchy, red tinted textures (happened on my AMD laptop). In those cases, I suggest assigning the dedicated video card to be used by the build (e.g. *NVidia* has it under *"Manage 3D settings"*).