Principles of Rendering: Criminowl

Jack Diver*
National Centre for Computer Animation

Abstract

This paper will explore rendering techniques, using procedural Shaders and GLSL API. The aim was to create a photo realistic image of an object small enough to hold in your hand, using real time rendering techniques.

1 Introduction

I am attempting to create a brushed and oiled wood shader, with a large degree of wear. This material is found on a small carved owl and consists of several layers, all some form of noise which is relatively simple, but together create a detailed surface.

Figure 1: *Photo of the carved owl.*

I observed these properties from the owl:

1. Large carved circles for the eyes.
2. Variation in the base colour from dark to light brown.
3. A brushed pattern that shows up more in the light areas
4. Small and frequent rough scratches.
5. Larger, less frequent and vein-like scratches.
6. Lots of chips in the painted coats.

I envisioned the owl as a hero character, and also aimed to animate him. I ported some of the noise functions from a procedural OSL shader that I created prior to this, and used them as the basis for my surface patterns.

2 Method Overview

2.1 Eyes

I began by writing a concentric circle shader, which borrows *smoothpulse* and *smoothpulsetrain* from Larry Gritz [Larry Gritz 1999]. These functions produce a repeating pattern of smooth steps up and then down, similar to a sin wave but with flat peaks and dips. A *smoothpulse* is defined by subtracting or multiplying two *smoothsteps* [Fundza 2002]. I used the position vector to calculate distance from the origin, this is useful as all points lying on the edge of a circle are the same distance from the centre of that circle.

*e-mail:jackdiver@hotmail.co.uk

Algorithm 1 Concentric Circles

1: **procedure** CONCENTRIC(p, f, g, t)
2: $r \leftarrow \sqrt{p.x^2 + p.y^2}$ ▷ Get the distance from origin
3: $sum \leftarrow (p.x + p.y)/r$ ▷ Sum the normalized x and y
4: $period \leftarrow g * sum$
5: **return** $spte(t, t + period, f, period, r)$

Note that *spte* is a wrapper for Larry Gritz function that applies the same fuzz to each side of the *smoothpulse*. The circles in the eye are not concentric, they all share a single point. The circles do not grow linearly; their growth is exponential. To account for these properties I added two new parameters to the algorithm, a warp factor, and an exponent. When warp is zero we get concentric circles, when it is one we get the eye pattern. Exponent determines the growth factor of the circles.

Algorithm 2 Owl eye

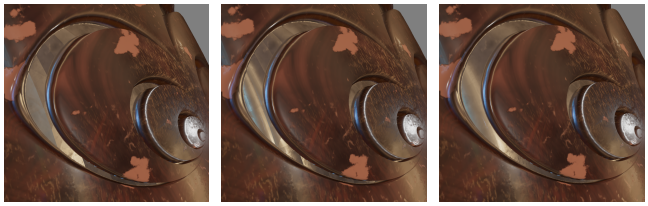
1: **procedure** EYE(p, f, g, t, w, e)
2: $recip \leftarrow 1/e$ ▷ Get the reciprocal of the exponent
3: $r \leftarrow \sqrt{p.x^2 + p.y^2}$ ▷ Get the distance from origin
4: $sum \leftarrow (p.x + p.y)/r$ ▷ Sum the normalized x and y
5: $period \leftarrow g * lerp(1, pow(sum, recip), w)$
6: **return** $spte(t, t + period, f, period, pow(r, recip))$

Here we make several modifications, to warp the circles away from being concentric we linearly interpolate between a uniform gap, and sum^{recip} which is what gives us the shared point. Similarly we use r^{recip} as the final parameter to our *smoothpulsetrain* which is what gives us the non-linear growth. On top of this, noise was added to the return value, and the position p, to give a more natural and imperfect result.

To perform the displacement I added geometry, tessellation control, and tessellation evaluation stages into my shader. The tessellation control used an expanded version of the mask used for the eyes, to only tessellate the geometry in the areas that would be displaced. The tessellation uses subroutines to allow the user to switch between different tessellation types, currently flat tessellation and phong tessellation are offered. Flat tessellation was performed by simply assigning positions to the new vertices using their bary-centric coordinates to blend between the positions of the parent vertices. Phong tessellation was used [Boubekeur and Alexa 2008], which first projects the vertex onto the three tangent planes of the parent vertices, then blends the resulting projections using bary-centric coordinates as we did with flat tessellation. I also added a uniform that allows the user to blend between flat and phong tessellation which means that they can get a smooth tessellation.

2.2 Wood

The wood texture is essentially lots of different types of noise mixed together. I used a large low frequency noise to blend the layers and turbulence was also used frequently. To produce the paint chips I defined a *slicednoise* function, which capped the noise at a given value using *smoothstep*. I found that this is very useful for blending my other layers together as most of them are limited to a few spots



(a) *Flat tessellation* **(b)** *Phong tessellation* **(c)** *Blended tessellation*

on the surface. The brush marks were created using turbulence, with noise and y-axis scaling applied to the position. The stretching produced long strands similar to brush marks. I defined another function which was used in the creation of the rough wood cracks and also the veins. This function again applies scaling in the y-axis to position, but then uses the *slicednoise* function I described earlier; we subtract the return of that function from one, which results in thin lines. These lines are the gaps between the spots of noise created by *slicednoise*. We also get lots of thin, faint lines within the spots themselves.

2.3 3D Textures

2.3.1 Albedo

Most of my methods involve the use of 3D noise to give proceduralism to the shader, however many successive calls to generate noise significantly hurt my performance so I decided to bake the layers of noise into a 3D texture. To do this I moved my code into a new shader, and setup a unit plane with orthographic projection, I then created a 3D texture and rendered to 2D slices of that texture using a frame buffer. Where each slice that was rendered, was given a z-depth value through a uniform, that offset the noise. Doing this increased my main shaders performance, however had several draw-backs, such as very high memory consumption which led to reduced resolutions, and also the user lost the ability to change the surface colour at runtime (unless they generated multiple maps).

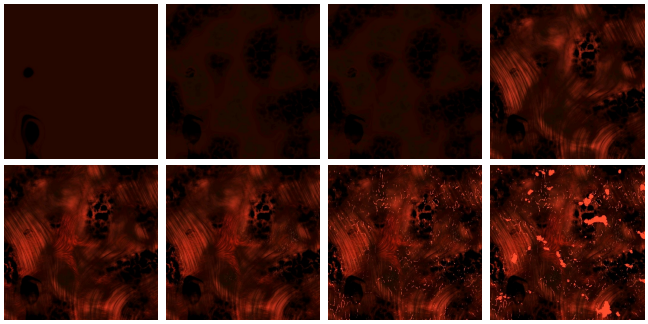


Figure 2: *Shows the layers of noise used to build up the albedo texture, this is a slice of the 3D texture.*

2.3.2 Normals

The generation of the 3D albedo map used float values from noise to blend colours together. In the original OSL shader, I summed these float values to use as my surface displacement, and so to keep this I stored the sum in the W component of the albedo texture. I didn't want to displace the surface of the owl in the same way as I did for the eyes as this would create very dense geometry and slow down performance, instead I wanted to use bump mapping. Bump mapping with a grey scale height map uses finite difference which involves at least four texture look-ups, seeing as the surface displacement is unchangeable due to it being baked previously, I decided to also bake my normal map, as this would reduce my texture look-ups. I used a very similar method to the one described for the albedo map, however this time the shader being rendered computed the finite difference between the W components of texels offset from the current fragment.

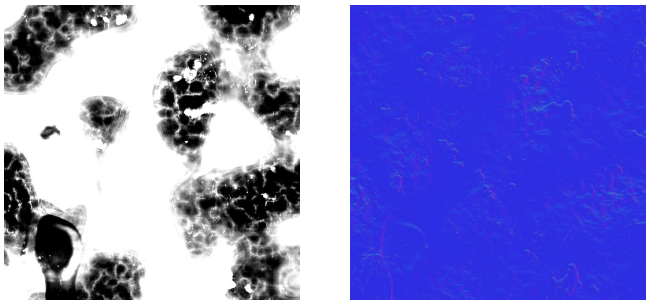


Figure 3: *Slice of the 3D height map on the left, and on the right is the normal map generated for it through finite difference.*

2.4 Animation

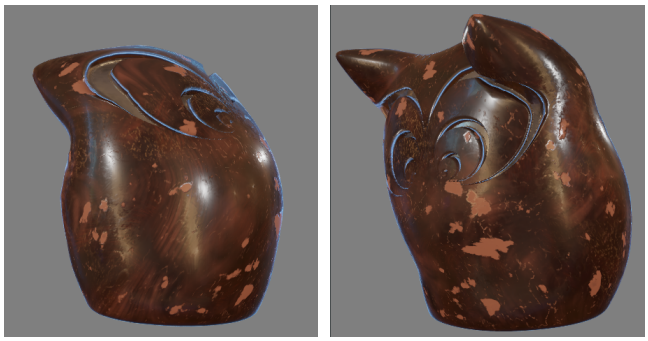
To animate the owl, I first created a rig for the owl (with much help from Miguel Goncalves), then key-framed some poses. I then exported an obj sequence of the entire animation. I wanted to be able to have no limits on the amount of poses that could be provided for the object, and so I decided to use a shader storage buffer object, as these can be as large as the GPU memory available. In my C++ code, I loaded all of the obj's, and then copied their vertex positions and normals into one large array. The positions and normals were padded to be four floats in size as there are alignment issues with passing vector3's. Instead of storing difference from the rest position, I just passed the absolute positions of every frame as this simplified my code. I then passed a blend float through a uniform, which could be in the range 0 to (N-1), where N is the number of frames, I then used this to smoothstep between floor(blend) and ceil(blend). To correctly index into my SSBO I also passed the target size, which was the amount of elements occupied by one morph target, and I also passed the normal offset, which represented how far into the SSBO, my normal data began. In the demo I am using 200 morph targets.

An issue I faced with this method was that the 3D positions used for texture look-up and to calculate the eyes, were now moving and this made the owl look like it was sliding through the textures. To resolve this I passed the base position and normals through the shader pipeline along with the morphed ones, but only used the base ones for texture look-up and eye calculations.

2.5 Shading and reflectance model

I implemented the Physically based, Cook-Torrence shading model. I chose to use Trowbridge-Reitz GGX that is also used by Epic [Brian Karis 2013], as it was well documented in their course notes. I also used the Schlick GGX, geometric shadowing matched to Smith, as this was in the course notes and also very efficient. The Fresnel is also Schlick. I then implemented image based lighting which gave a huge visual boost, this was done by converting a HDR spherical map into a cube map, by rendering each face to texture using a framebuffer and projection shader. Next I convoluted the resulting cube map, using importance sampling to create an irradiance map. Then I then used a split sum approximation detailed by Epic to generate a BRDF map and a pre-filtered environment

map, that stores varying roughness levels inside of it's mip-maps. Joey De Vries explains the entire process very well [Joey De Vries 2017].



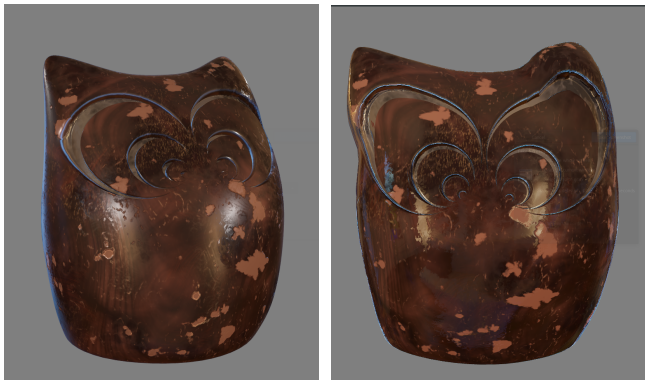
(a) *Looking surprised* **(b)** *Looking sad*

Figure 4: *Two of the key poses from the animation sequence.*

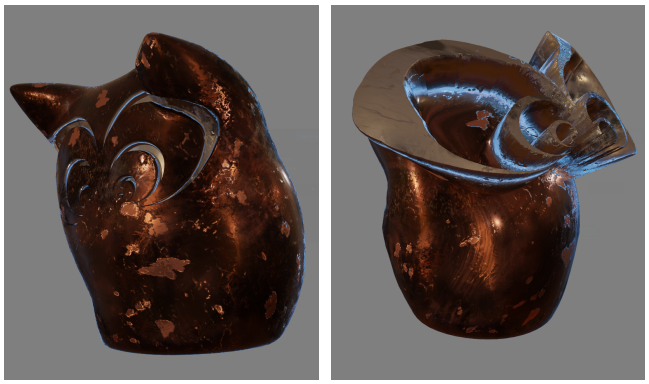
3 Results

The images on the right show some of the results that can be achieved with the shader parameters.

There are several aspects of this shader that I would improve on. The first would be to improve the memory footprint, by using some method such as sparse textures, to turn my 3D textures into voxels, and only store the ones that intersect with my geometry. I would also have liked to generate an Ambient occlusion map as this would have improved the self shadowing. I would also have improved on my placement of the wood chips by biasing them around the eyes. Overall I am happy with my results, especially the customisablity achieved through the shader parameters, that can be leveraged to make interesting results.



(a) *Trying to match the object* **(b)** *Morphed and high spec*



(c) *Morphed and metallic* **(d)** *Crazy*

Figure 5: *Adjusting shader parameters to get different looks.*

References

BOUBEKEUR, T., AND ALEXA, M. 2008. Phong tessellation. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia 2008) 27*, 5.

BRIAN KARIS, 2013. Real shading in unreal engine 4. `http://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf`.

FUNDZA, 2002. Rsl, using smoothstep. `http://www.fundza.com/rman_shaders/smoothstep/`.

JOEY DE VRIES, 2017. Specular-ibl. `https://learnopengl.com/PBR/IBL/Specular-IBL`.

LARRY GRITZ, 1999. Advanced renderman: Creating cgi for motion pictures. `http://www.larrygritz.com/arman/materials.html`.