Shading Shades

Frank Jargstorff

fjargstorff@nvidia.com

June 1, 2004

Abstract

Procedural material shaders continue to pop up since the advent of fragment programmability. Most of these shaders concentrate on interesting animated effects. The cg_ shades demo successfully explores the use of fragment shader programming for a different kind of materials: Highly repetitive, man-made materials.

The given code shows a variety of implementation options adapted to different usage scenarios, e.g. alpha-tested transparency vs. fragment kill. How to texture the "virtual geometry" and how to avoid artifacts inherent to the technique (i.e. clipping ends of bars and aliasing).

1 Introduction

Procedural material shader continue to pop up since the advent of fragment programmability. Most of these shaders concentrate on interesting animated effects. The procedural materials explored in this demo are of a different nature: Highly repetitive, man-made materials.

The demo explores a variety of implementation options for rectangular bars or slats similar to window blinds.

2 Motivation

Similarly to the usage of bump-maps the technique of the GrateShader demo allows displaying highly complex materials using the fragment processor instead of detailed vertex geometry.

Example: A grit covering a man-hole of 1m^2 with metal bars spaced at 1cm would require

$$100 \times 100 \times 10 = 100,000$$

quads! (See Figure 1)

Even if polygon counts were kept at a manageable level this approach is not practicable for several reasons:

1. Unless a smart level-of-detail method is deployed the hardware would have to render all this geometry, even if the man-hole from the example would only cover a handful of pixels on the screen. A per-pixel shader simulating the geometry would have the advantage of scaling with the amount of screen-space covered by the man-hole.

2. Even if the creation of the geometry was done by macros in the DCC applications the excessive geometry would increase file-sizes and load times in the final assets.

3. Writing such geometry-generator macros requires a lot of skill and integration into the DCC application is not trivial, e.g. the artist creates the man-hole and realizes that instead of spacing the bars at 0.3 inches he wants to space them at 0.5 inches. In this case the macro would have to delete a quarter million geometric primitives and create a similar amount of new ones. It is very unlikely that this would run interactively.

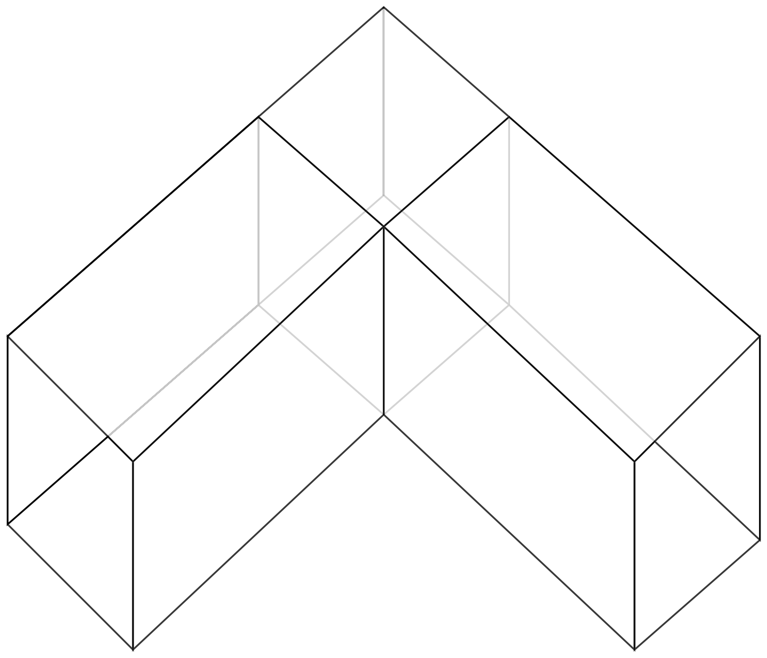

Figure 1: Grates used for cat walks or man-hole covers are made up from cells like this one. Each cell consists of 10 quads.

4. Highly repetitive geometric patterns are prone to expose aliasing patterns. This means that without full screen anti-aliasing the detailed geometry might look worse than necessary and at a very high cost.

3 Window Blinds and Prison Bars

All examples in the GrateShader demo shade a sequence of parallel, rectangular objects giving the look of a window with prison bars or window blinds.

In order to understand the demo's code a consistent nomenclature is used through this white paper and was carried over into the code as stringently as possible.

2 gives a first impression of how the shader works. In order to shade the bars one renders a single quad. The bars are assumed to sit flush underneath the quad. The orientation of the bars is determined by the direction of the texture coordinates (u, v) so that (at least in theory) the shader could be

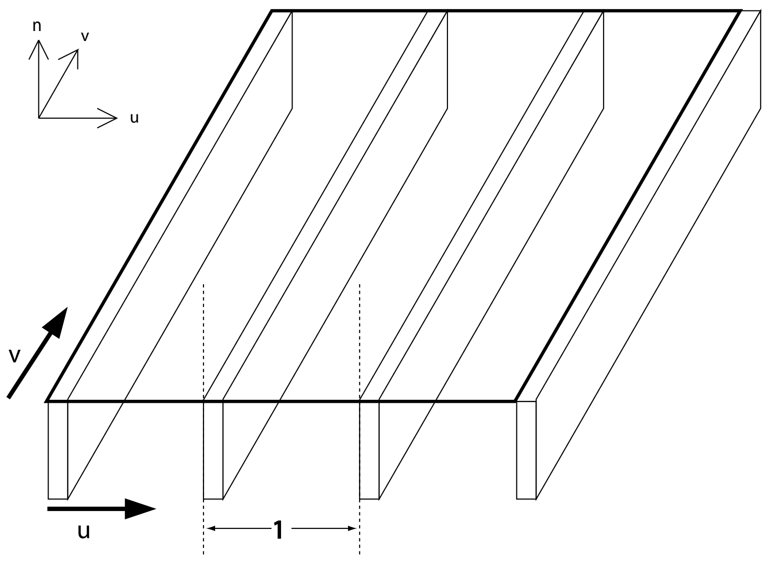

Figure 2: Quad with (U,V) texture coordinates and slats.

applied to arbitrary tessellations of the quad or even curved surfaces.

The shader code is based on the assumption that the spacing between bars is 1 unit in texture coordinate terms. This allows an artist to easily change the spacing of the bars by changing the texture coordinate mapping.

Similarly to bump-mapping the GrateShader requires a local coordinate system. In the example code this coordinate system is given as the basis of vectors $\{\vec{u}, \vec{v}, \vec{n}\}$ as shown in the figure. It is not handed to the shader as per-vertex attributes but as a set of `uniform` parameters to the vertex shader.

4 Shading the Bars

The fragment shader has to perform the following two tasks:

1. Determine whether the slat was hit by the eye-beam or not. If the slat was not hit the fragment can be discarded or alternatively a completely transparent color can be return (alpha = 0).

   If the slat was hit the program must evaluate a lighting equation that determines the color of the light that is reflected off the slat. Evaluating this equation usually involves a surface normal. The slats have three relevant surface normals: up, if the top of the slat was hit, in u-direction, if the right side of the slat was hit, or in negative u-direction, if the left side of the slat was hit. So in order to determine the normal the fragment program needs to find out which of the sides of the slat were hit.

2. Evaluate the lighting equation and return the resulting color.

4.1 Geometry

As stated above we assume that $u \in [0, 1[$ (i.e. all numbers between 0 and 1 without 1). Since the slats are spaced at a distance of 1 in $\vec{u}$-direction this means we need to only think about a single cell.

As shown in Figure 3 it is fairly simple to determine if and where the eye-beam hits. The simplest case is $u \geq s$ in this case the beam must hit the top of the slat. In case $u < s$ we have to distinguish three sub cases:

1. the beam hits the right side of the slat to the left,
2. the beam hits the left side of the slat to the right, or
3. the beam hits nothing at all.

Looking at Figure 3 it is also obvious that for the projected slat-corner c being to the right of u indicates hitting the left slat, i.e. $(u < c')$ we hit left slat. For negative c' we know that we can only hit the right slat. In this case we have to project the lower-left corner of the slat which results in the projection point of $c' - w = c' - (1 - s) = c' + s - 1$. In order to shift this negative value into our $[0, 1[$ interval we define $c'' := c' + s$ which gives the projection of the lower-left corner of the slat to the right. If $c'' = c' + s < u$ we hit the slat to the right. If neither case is true we didn't hit the slats at all.

c' can be easily determined using similar triangles and the two scalar products $\langle \vec{e}, \vec{n} \rangle$ and $\langle \vec{e}, \vec{u} \rangle$[1]. It is

$$\frac{c'}{n} = \frac{\langle \vec{e}, \vec{u} \rangle}{\langle \vec{e}, \vec{n} \rangle} \Leftrightarrow c' = h \frac{\langle \vec{e}, \vec{u} \rangle}{\langle \vec{e}, \vec{n} \rangle}$$

[1]In the following we use $\langle \cdot, \cdot\cdot \rangle$ to denote the scalar product (or dot product) between two vectors.

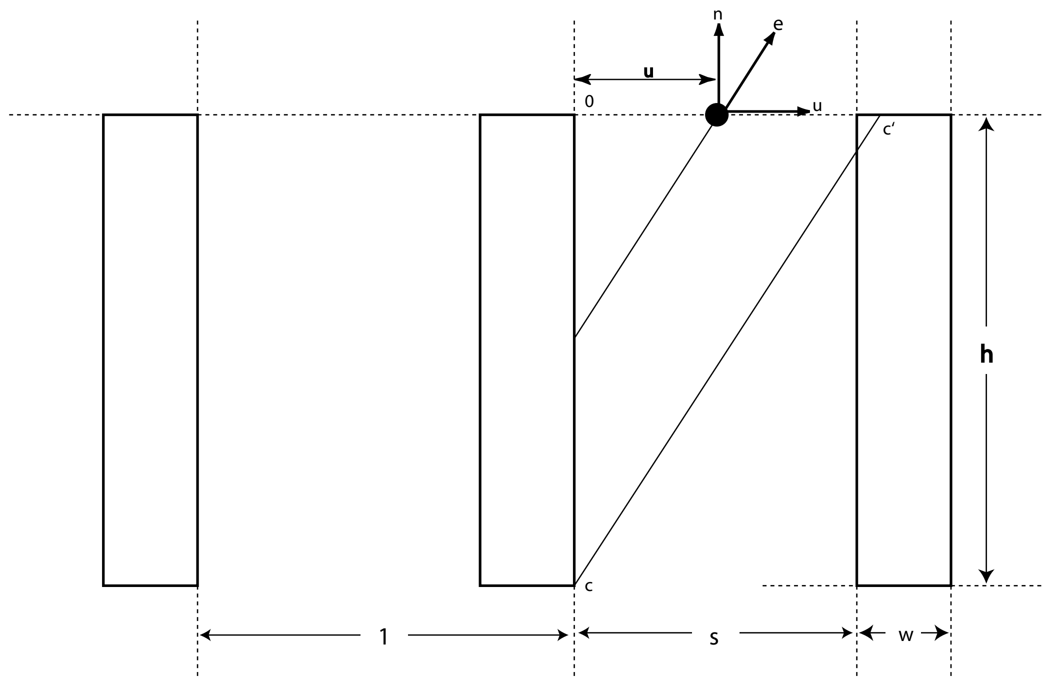

Figure 3: Geometry problem in 2D. The distance between two slats is 1. w is the slats' width, h their height. $\vec{e}$ points toward the eye-point. s is the space between two slats. The eye-beam hits the right side of the slat. c is the bottom right corner of the slat and c' is the projection of this corner onto the quad.

This equation is numerically instable for very flat viewing angles (i.e. $\langle \vec{e}, \vec{n} \rangle$ close to zero). By multiplying the equations by this factor one gets numerically more stable versions that should be used in the implementation.

The following pseudo code uses the results above to determine geometry. Top, right, left, and none indicate where and if the slats were hit by the eye-beam.

```
float3 geometry(float u, float h, float s, float3 n, float3 u, float3 e)
{
   if (u >= s)
      return top;

   float HdotEU = h * dot(e, u);
   float dotEN = dot(e, n);

   if (u * dotEN < HdotEU)
      return right;

   if (HdotEU < (u - s) * dotEN)
      return left;

   return none;
}
```