A Brief OpenGL Shading Tutorial

Marc Olano

University of Maryland, Baltimore County

1 Overview

There are two main pieces to using the OpenGL Shading language in an application, the language itself, and the portions of the OpenGL API that control that language. The language is similar in many respects to the other real-time shading languages presented here, Cg and HLSL, though we will highlight some of the similarities and differences. It is in the API that the difference is most evident.

2 The OpenGL API

As of OpenGL 2.0, the shading language features are a required part of the core API. If you have OpenGL 2.0, you need not worry about whether your particular card and driver support the shading language — they must. In OpenGL 1.5, the shading language features were an ARB-approved optional extension. The OpenGL ARB is the standards body that determines what is *officially* part of OpenGL, as compared to vendor-defined extensions that don't need ARB approval. As an ARB-approved extension, you know that anyone who does support it will support the same interface and language [Lichtenbelt and Rost 2004]. You can check if an OpenGL 1.5 driver supports the shading language by checking for the `GL_ARB_shader_objects` extension string in the `glGetString()` results.

We will present the OpenGL 2.0 versions of the calls. In most cases, the `ARB_shader_object` versions just add an ARB extension, so functions like `glCreateShaderObject()` become `glCreateShaderObjectARB()` and symbols like `GL_VERTEX_SHADER` become `GL_VERTEX_SHADER_ARB`. The definitive source for the OpenGL 2.0 API is the specification [Segal and Akeley 2004]. The OpenGL specification used to be a rather difficult document to use for all but the most determined, but as a searchable PDF, it is actually not too difficult to use it directly as a reference.

The main difference between OpenGL and the other languages is that the OpenGL shading language and compiler are built directly into the OpenGL API and provided by the graphics card vendor as part of their driver. There is no intermediate low-level code to get from the compiler and hand to the API. This has some potential hidden advantages as well. The instruction sets actually executed by the graphics hardware already different from the standard instructions sets. For example, even if the hardware can execute up to four independent arithmetic operations in a single 'instruction', the low-level instruction sets only support executing

the same operation on the same data. A final optimization on the low-level code can find and fix these, but the high-level language compiler is left to try to match a pattern that the low-level optimizer will recognize and fix. There's some chance this will work if both come from your hardware vendor, but it's easy to get non-optimal code. By hiding any low-level translation, the shading compiler built into OpenGL is free to target whatever the hardware really does, and the hardware vendors are more free to change it later to improve their hardware without worrying about matching that hardware to the language. Note that Direct3D is moving to this model as well [Blythe 2004].

2.1 The Interface

The OpenGL shading interface has two important concepts, *shader objects* and *program objects*. Shader objects hold a single high-level (vertex or fragment) shader, while program objects contain the collection of shader objects used to render an object. Shader types for other future kinds of shaders have not yet been defined (e.g. for geometry shaders [Blythe 2004]), but would fit cleanly into this interface.

To create, load from a single C-style string, and compile a vertex and fragment shader object, you could use code like this:

```
vert = glCreateShaderObject(GL_VERTEX_SHADER);
glShaderSource(vert, 1,(const GLcharARB**)&vString,NULL);
glCompileShader(vert);

frag = glCreateShaderObject(GL_FRAGMENT_SHADER);
glShaderSource(frag, 1,(const GLcharARB**)&fString,NULL);
glCompileShader(frag);
```

These two can be joined into a single program object like this:

```
prog = glCreateProgramObject();
glAttachObject(prog,vert);
glAttachObject(prog,frag);
glLinkProgramObject(prog);
```

Whenever you want to use this set of shaders on an object, you just tell OpenGL that you want to start using it.

```
glUseProgramObject(prog);
```

2.2 Alternatives for Loading

The interface for loading shader source is quite flexible, allowing many variations for how your program handles its shaders. The actual parameters are

```
glShaderSource(object,segments,stringArray,lengthArray)
```

The arrays (containing `segments` entries; one in the example above) allow shaders to be stored as a set of lines or code segments, which need not be null-terminated. If any length is -1, OpenGL assumes that segment is a null-terminated string and computes the length. If the pointer to the length array is NULL, OpenGL assumes every segment in the array is null-terminated.

Having an array of lengths means the segments need **not** be null-terminated if that isn't convenient. In particular, if you have the ability to map a file directly to memory, you could use code similar to this:

```
fd = open(vsname.c_str(), O_RDONLY, 0);
fstat(fd,&sb);
sh = (char*)mmap(0,sb.st_size,PROT_READ,MAP_FILE,fd,0);
glShaderSource(vert, 1, (const GLcharARB**)&sh,
               (const GLint*)&sb.st_size);
```

This code relies on `fstat()`, which reports information on a file, including its size, and `mmap()`, which maps a file directly into memory, potentially more efficiently than reading it in.

2.3 Compile Errors

The above code may be fine for production use, once the shaders are known to be error-free, but no one is capable of writing error-free code every time. After the `glCompileShader()`, you can find out if the shader compiled successfully with

```
glGetShaderiv(vert, GL_COMPILE_STATUS, &result);
```

`result` will be true (non-zero) if the shader compiled successfully. To find out what is wrong with a shader that didn't compile, check the info log:

```
glGetShaderInfoLog(vert, bufsize, NULL, buffer);
```

Similarly, `glGetProgramiv()` and `glGetProgramInfoLog()` can tell you about the success (or failure) of the progam linking step.

2.4 Shading Parameters

Of course, once you can load and use a shader, you still need a means to control it. Since the OpenGL shading language is built into OpenGL, every vertex shader has access to all of the usual OpenGL vertex attributes (position, color, normal, etc.), and all shaders have access to the built-in OpenGL state (light positions, matrices, etc.). Shaders access all of these using special pre-defined names in the shading language. However, any shader can also define additional attributes and state that it would like to use. You can find out the *index* for a per-vertex attribute with

```
index = glGetAttribLocation(prog, "vertexAttribute");
```

and set a value using one of the `glVertexAttrib*()` functions. For example

```
glVertexAttrib3f(index, 0,.5,1);
```

All vertex attributes, whether built-in ones like `glColor` or user-defined must be set before the corresponding `glVertex` call.

A similar set of functions exist for any user-defined *uniform* state. The uniform state affects the current program, so switch programs before you start setting the state.

```
index = glGetUniformLocation(prog, "uniformVariable");
glUniform4f(index, .2,.4,.6,1);
```

The string name used to identify a variable can be more than the variable name alone. It can include array indexing and structure dereferencing operations to allow you to set a single element. For example,

```
index = glGetUniformLocation(prog,
    "structArray[2].element");
glUniformMatrix4fv(index, 1,0, matrix);
```

3 Shading Language

Many excellent references exist for the OpenGL Shading Language exist, so this document will not attempt to exhaustively list every feature. For more details, refer to one of the other sources [Kessenich et al. 2004; Rost 2004]. Many of the features of the OpenGL shading language, are similar if not identical to the other shading language options. All have you write vertex and fragment/pixel shaders (as opposed to the RenderMan model of displacement, surface, light, volume and imager). All inherit many syntactic features from C, including `if`, `while`, `for`, the

use of "{", "}", and ";", and the existence of structs and arrays for grouping data. They also share certain features of all shading languages (even non-real-time languages like RenderMan), in having small vectors and matrices as built-in types, and a common set of math and shading functions. Like other real-time languages, screen-space derivatives are available (through the `dFdx()` and `dFdy()` functions), a struct-like notation is used for swizzling vector components and writing to only specific vector components. For example

```
vec2.xzw = vec1.yyw;
```

assigns `vec1`'s y value to `vec2`'s x and z component and `vec1`'s w value to `vec2`'s w component. `vec2`'s z component keeps whatever value it had before the assignment.

3.1 Notable Differences

Probably the first difference you'd notice between the OpenGL Shading Language and either Cg or HLSL is the important part that *virtualization* plays in the OpenGL language philosophy. If all of the programs running on your CPU exceed the available physical memory, the operating system can make it seem as if you have a much larger pool of *virtual memory* by swapping some stuff off to disk. It's not as fast switching between applications as if you had a larger pool of physical memory, but in most instances it's **much** better than crashing or not letting you switch back and forth between two applications.

Similarly, the OpenGL shading language defines some minimum features (and some features like instruction count for which there are no defined limits). A working OpenGL Shading Language implementation is required to make it seem as if you're running on that ideal hardware. It may switch to running multiple passes, it may run some things in software, but they will always run. This means one code base may run in a two passes on one machine, in three on another, or in one on a future machine that didn't even exist when you wrote the shader. Splitting shaders into multiple passes is hard to do by hand, especially when you are writing high-level code, so it makes much more sense to let the shading compiler use one of the multi-passing compilation techniques [Foley et al. 2004; Riffel et al. 2004] than for you to try to do it twelve different ways by hand.

The second notable difference is that in the OpenGL Shading Language, vertex to fragment communication is determined by the vertex shader writing to a varying variable and the fragment shader using it. The compiler chooses the *interpolator* to use. From the language standpoint, it's just data.

There are other minor differences in the names of some of the data types (OpenGL's `vec4` vs. Cg or HLSL's `float4`). Those are usually pretty obvious and easy to translate from one to the other. One that may catch more users is that in OpenGL, `matrix*vector` is a matrix/vector product and `matrix*matrix` is a linear algebraic matrix product, as compared to Cg where you use the `mul()` function and `matrix*matrix` gives a component-wise multiply (there's a function for that in OpenGL). They're the same operations, just with syntax that differs in a way that may surprise the unsuspecting.

4 Example

This example shows a simple single-light diffuse shading computed per vertex and applied to a 3D noise fragment shader

Vertex shader:

```
// noise input to fragment shader
varying vec3 Nin;

void main()
{
    // transform vertices into projection space
    gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;

    // vertex in view space for lighting
     vec4 viewPos = gl_ModelViewMatrix*gl_Vertex;

     // normalized normal, also in view space
    vec3 nn = normalize(gl_NormalMatrix*gl_Normal);

    // handle directional and point lights together
    vec3 nl =
        normalize(gl_LightSource[0].position.xyz*viewPos.w
            - viewPos.xyz*gl_LightSource[0].position.w);

    // add ambient and diffuse lighting
    gl_FrontColor =
        gl_FrontMaterial.ambient*gl_LightSource[0].ambient
       +gl_FrontMaterial.diffuse*gl_LightSource[0].diffuse
            * max(0.,dot(nn,nl));

    // compute scaled noise input
    Nin = gl_Vertex.xyz*.1;
}
```

Fragment Shader:

```
// vertex to fragment communication for noise shaders
varying vec3 Nin;

// 2D noise texture
uniform sampler2D ntex;

// modulus for random hash
const float modulus = 61.;

void
main()
{
    // integer and fractional components of input
    float fracArg = fract(modulus*Nin.z);
    float intArg = floor(modulus*Nin.z);

    // hash z & z+1 to get offsets for noise slices
     // compared to the noise described in the Modified
     // Noise paper, I found that including a small
     // mutually prime multiplier improved the visual
     // quality of the hash near 0. I believe this is
     // because 0^2, 1^2, 2^2 are well under the hash
     // modulus and close enough to appear correlated
     // so… 3*(z+1) = 3*z + 3
    vec2 hash = mod(vec2(3.*intArg)+vec2(0.,3.),modulus);
     hash = mod(hash*hash,modulus)/modulus;

    // look up noise and blend slices
    vec2 g0, g1;
    g0 = texture2D(ntex, vec2(Nin.x,Nin.y+hash.x)).ra*2.-1.;
    g1 = texture2D(ntex, vec2(Nin.x,Nin.y+hash.y)).ra*2.-1.;
    float noise = mix( g0.x+g0.y*fracArg,
                       g1.x+g1.y*(fracArg-1.),
                       smoothstep(0.,1.,fracArg));

    // combine with lighting
    gl_FragColor = (noise*.5+.5)*gl_Color;
}
```

The results of this vertex and fragment shader is shown in Figure 1.

a b c

Figure 1. a) red component of texture. b) alpha component of texture. c) Teapot rendered with resulting appearance

References

[Blythe 2004] David Blythe, "Windows Graphics Foundation", WinHEC 2004 presentation, May 2004.

[Foley et al. 2004] Tim Foley, Mike Houston and Pat Hanrahan, "Efficient Partitioning of Fragment Shaders for Multiple-Output Hardware", Proceedings of Eurographics/SIGGRAPH Graphics Hardware 2004, July 2004.

[Kessenich et al. 2004] John Kessenich, Dave Baldwin and Randi Rost, *The OpenGL Shading Language*, Language Version 1.10, OpenGL ARB, April 2004.

[Lichtenbelt and Rost 2004] Barthold Lichtenbelt and Randi Rost, "ARB_shader_objects", OpenGL extension document, OpenGL ARB, June 2004.

[Riffel et al. 2004] Andrew T. Riffel, Aaron E. Lefohn, Kiril Vidimce, Mark Leone and John D. Owens, "Mio: Fast Multipass Partitioning via Priority-Based Instruction Scheduling", Proceedings of Eurographics/SIGGRAPH Graphics Hardware 2004, July 2004.

[Rost 2004] Randi Rost, *OpenGL Shading Language*, Addison Wesley, 2004.

[Segal and Akeley 2004] Mark Segal and Kurt Akeley, *The OpenGL® Graphics System: A Specification*, Version 2.0, Editors Jon Leech and Pat Brown, OpenGL ARB, October 2004.

Graphics Hardware (2005)
M. Meissner, B.- O. Schneider (Editors)

Modified Noise for Evaluation on Graphics Hardware

Marc Olano†

UMBC

Abstract

Perlin noise is one of the primary tools responsible for the success of procedural shading in production rendering. It breaks the crisp computer generated look by adding apparent randomness that is controllable and repeatable. Both Perlin's original noise algorithm and his later improved noise were designed to run efficiently on a CPU. These algorithms do not map as well to the design and resource limits of the typical GPU. We propose two modifications to Perlin's improved noise that make it much more suitable for GPU implementation, allowing faster direct computation. The modified noise can be totally evaluated on the GPU without resorting to texture accesses or "baked" into a texture with consistent appearance between textured and computed noise. However, it is most useful for 3D and 4D noise, which cannot easily be stored in reasonably-sized textures. We present one implementation of our modified noise using computation or direct texturing for 1D and 2D noise, and a procedural combination of 2D textures for the 3D noise.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image generation; Display algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism; Color, shading, shadowing and texture

1. INTRODUCTION

Procedural shading allows a programmer or technical artist to write short procedures called *shaders* in a *shading language* to define how surfaces will appear. This general technique has proved hugely successful in production rendering, appearing in almost every CGI film and effects shots, from Toy Story [Pix95] to Lord of the Rings [New01].

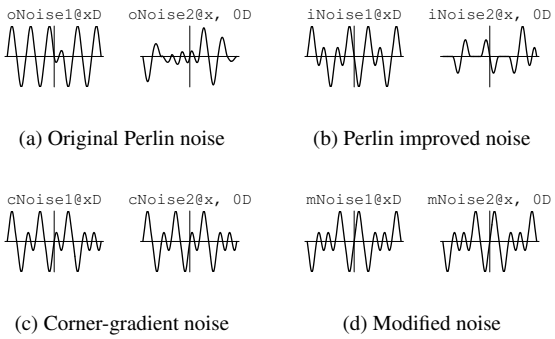
Arguably, one of the main reasons for the realism associated with procedural shading, and its popularity in production rendering, is the *Perlin noise* function. Noise was introduced as part of Perlin's *Image Synthesizer* [Per85]. In 1997 he received a Technical Achievement Academy Award for its development and the impact it has had on movie effects. The Perlin noise function adds randomness to procedural shaders, and is commonly used both as a surface appearance modeling tool and as a means to add dust, scuffs, or other imperfections to otherwise pristine surfaces [EMP*02].

The key aspects of Perlin noise are that it is deterministic and frequency band-limited. Determinism allows it to be reliably used in animation. Thus, the appearance of an object will not change unexpectedly from frame to frame. The frequency band limits allow it to be used in a controllable fashion. Perlin's noise function has most of its energy between .5 and 1 cycles per input unit. A shader-writer desiring higher or lower frequencies simply scales the noise inputs to produce the desired frequency profile. More complex frequency profiles can be synthesized by adding several octaves, as with the *fractional Brownian motion* (fBm) and *turbulence* functions [Per85].

A number of *noise* variants are used, with varying input dimension (usually 1D-4D), and varying output dimension (usually 1D or 3D). 3D-output noise can be built from three offset 1D-output noise functions. Consequently, as Perlin did, we concern ourselves with 1D (single float) output. In the remainder of this paper, *n-D noise* refers to a noise function with n-dimensional input and 1D output.

Perlin's formulation uses several chained table lookups, operations that are relatively fast on a CPU, but can be a bottleneck on a GPU [Wei04]. We present a modified noise

† email:olano@umbc.edu



(a) Original Perlin noise (b) Perlin improved noise

(c) Corner-gradient noise (d) Modified noise

Figure 1: *Plots of 1D noise functions and a 1D slice through their 2D counterparts. Note that the dimension reducible property, makes* `noise1[x]` *and* `noise2[x,0]` *match for (c) and (d)*

function with no table lookups. It may be accelerated with textures to hold evaluated results of 2D subexpressions, but unlike the original noise formulation, they are neither required, nor do they form a dependent chain.

Section 2 explores prior work in noise construction. Section 3 introduces our modifications and compare them to Perlin's noise functions. As is seen in Figure 1(b), Perlin's improved noise sometimes exhibits artifacts that the original did not, so we compare our modified noise to both (in fact, these artifacts were the original inspiration for this work).

2. PRIOR WORK

Peachy [EMP*02] created a useful taxonomy for noise functions. The most common choice in production rendering is *gradient noise*: a repeatable function is used to choose a random gradient at each point on an integer lattice. Between the lattice points, polynomial interpolation produces a smooth noise function with the desired gradient at each lattice point. Perlin's original noise was of this form [Per85], using a hash of the lattice point locations to choose random gradients from a pre-computed table.

Lewis [Lew89] instead used sparse convolution to construct a noise function. *Sparse convolution noise* avoids grid artifacts common with grid-based noise synthesis approaches, but is more computationally expensive to evaluate on the fly. Van Wijk's *Spot Noise* is similar, but modifies the noise by changing the spot shapes [vW91].

Perlin's simplex noise variant also reduces grid artifacts by using an n-simplex as the basic grid (i.e. triangle rather than square, tetrahedron rather than cube) [Per01]. There is a GPU implementation by Gustavson [Gus05]. Simplex noise has the advantage of only combining $n+1$ gradients rather than 2^n for n-D noise, but is otherwise independent of any modifications we propose here.

Most real-time systems have elected to use the inferior

Figure 2: *Comparison of computed (light) and sampled (dark) noise for different texture sizes: 2 samples/unit = texture size 2×period; 3 samples/unit = texture size 3×period; or 4 samples/unit = texture size 4×period.*

value noise [LMOW95, Har01]. Value noise interpolates between random noise values defined at each lattice point The maximum frequency of value noise is 1/2 Hz, but contains significant lower-frequency energy. Combined with the more prevalent grid artifacts, this makes value noise a poor visual choice. It is, however, exceptionally easy to compute.

Recently, Perlin [Per02] introduced an improvement to his original noise. The improved version is still a lattice *gradient noise*, with two changes. It uses a higher order interpolant for a smoother look, and rather than hashing lattice points into a table of gradients, one of a fixed set of gradients is computed from the hash value itself. There is an optimized GPU implementation by Green [Gre05].

One common optimization for hardware is baking the noise function into a texture. Close viewing requires a texture at least 3-4 times the noise period (Figure 2). Traditional Perlin noise has a period of 256, requiring 768^2 or 1024^2 textures to accurately represent the 2D noise or $768^3 - 1024^3$ for 3D noise!

Perlin has also presented a technique to greatly reduce these memory requirements for one 3D noise [Per04] (though the method could apply for 1D-3D noise). He generates a small 3D complex-number noise texture, then extends the period of this repeating texture by *rotating* the complex output according to a second, low-frequency, 3D noise (which can use the same texture). The final noise value is the real component of the output. His version uses a 25^3 texture with a base period of 8 to achieve noise with a period of 72. The noise created through this method does not match any of the previous computed noises, though its statistical character is similar. The complex rotation uses trigonometric functions, and computing a matching noise is problematic, but the memory compression is excellent.

3. MODIFYING NOISE

As our first modification, we would like the lower-dimensional noise functions to be a direct slice from their higher-dimensional counterpart. We will refer to noise with this property as *dimension reducible*. Textured implementations of a dimension-reducible noise can make particularly efficient use of the available texture memory. In our final implementation, we use single 256^2 texture for 2D noise with a period of 61. 1D noise is just one line from the same texture (see Figure 1).

Second, while the noise function derived here may be used directly from a texture, we feel it is important that it also be computable **without** textures or table lookups. We will refer to noise with this property as *purely computable*. In our implementation, the noise textures are generated by direct GPU computation into a texture. In addition, purely computable noise allows a partial precomputation into textures, as presented in Section 3.4.

Finally, even if textures are to be used in the implementation, we would like to avoid dependent texturing. Hardware limiting the number of dependent lookups does still exist, so it is good shader citizenship not to use all of them in implementing the noise function. Even on hardware that does not limit the number of dependent lookups, chains of lookups make it more difficult to hide the texture fetch latency.

We present our modifications along with an overview of Perlin's original and improved noise. We will refer to these as *oNoise* and *iNoise* respectively. This comparison will serve to highlight the areas of flexibility in defining a noise with the same general characteristics as *oNoise* and *iNoise*.

3.1. Commonalities

Given an input point, both algorithms locate the 2^n surrounding lattice points. For 2D, the points surrounding $\vec{p}$ are

$$\vec{p}_{i;jk} = \begin{pmatrix} \lfloor \vec{p}^x \rfloor + j \\ \lfloor \vec{p}^y \rfloor + k \end{pmatrix}; j,k \in \{0,1\}$$

Through some method (differing for each noise), a gradient vector is computed at each $\vec{p}_i$, and a function with the desired gradient constructed

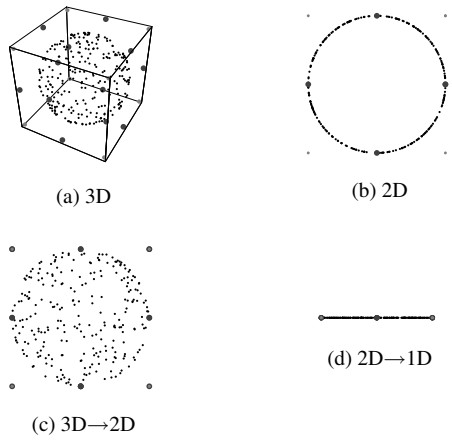
$$\vec{p}_f = \vec{p} - \vec{p}_i$$
$$grad(\vec{p}_i, \vec{p}_f) = gradient(\vec{p}_i) \bullet \vec{p}_f$$

The nearest lattice gradient functions are blended using a smooth interpolation. The most common method factors a smooth fade function into the interpolation parameter of a linear, bilinear or trilinear interpolation.

$$\begin{aligned} fade(t) &= \begin{cases} 3t^2 - 2t^3 & \text{for } oNoise \\ 10t^3 - 15t^4 + 6t^5 & \text{for } iNoise \end{cases} \\ flerp(t,a,b) &= (1 - fade(t))\, a + fade(t)\, b \end{aligned}$$

We can choose either fade function (or even a linear fade) to balance computation and appearance. Except where indicated, images here were produced using the cubic fade function. *oNoise*, *iNoise* and our modifications all differ **only** in the choice of fade function and gradient vectors. Thus, the 2D noise equation **for all** is

$$\begin{aligned} noise2(\vec{p}) = flerp(\vec{p}^y_{f;00}, & flerp(\vec{p}^x_{f;00}, grad(\vec{p}_{i;00}, \vec{p}_{f;00}), \\ & grad(\vec{p}_{i;10}, \vec{p}_{f;10})) \\ & flerp(\vec{p}^x_{f;00}, grad(\vec{p}_{i;01}, \vec{p}_{f;01}), \\ & grad(\vec{p}_{i;11}, \vec{p}_{f;11})) \end{aligned}$$



(a) 3D (b) 2D (c) 3D→2D (d) 2D→1D

Figure 3: *Noise gradients. a,b) direct; c,d) from a slice of higher-dimensional noise. Small/Black: oNoise gradients on a unit n-sphere; Large/Medium: iNoise gradients at the edge centers of a unit n-cube; Small/Light: cNoise and mNoise gradients at the corners of a unit n-cube*

	iNoise	*cNoise*	
3D:	$(\pm x, \pm y, 0), (\pm x, 0, \pm z), (0, \pm y, \pm z)$	$(\pm x, \pm y, \pm z)$	
3D→2D:	$(\pm x, \pm y), (\pm x, 0), (0, \pm y)$	$(\pm x, \pm y)$	]
2D:	$(\pm x, 0), (0, \pm y)$	$(\pm x, \pm y)$	]
2D→1D:	$(\pm x), (0)$	$(\pm x)$	]
1D:	$(\pm x)$	$(\pm x)$	]

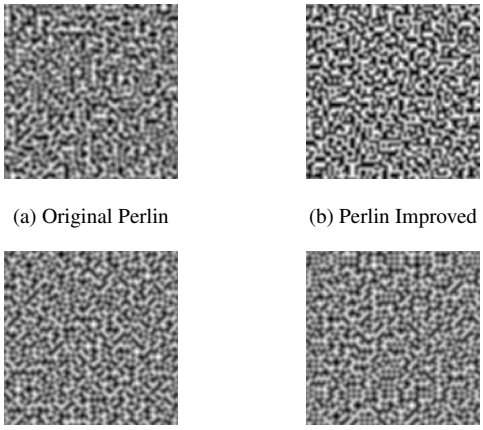
Table 1: *grad functions for Perlin's improved noise (iNoise) and our cNoise. Notice that the projection to a lower dimension produces spurious gradients in iNoise, but projects cleanly in cNoise*

3.2. Gradients

The noise functions begin to differ with their selection of random gradients. *oNoise* chooses lattice gradient values from a unit n-sphere, while *iNoise* instead chooses gradients at the center of the edges of a unit n-cube. The values of these gradients are shown in Figure 3 and (for *iNoise*) Table 1. Figures 3(c) and 3(d) show the effective gradient distribution for a 2D or 1D slice of a higher dimensional noise.

Figure 1 shows that these gradient distributions result in noise functions that look similar at the intended dimensionality, but appear quite different for lower-dimensional slices. From Figure 3 and Table 1, we observe that the problem arises when a gradient vector in one noise does not project down to the gradient that would be chosen in the lower-D noise. *iNoise* is closer to resolving the problem, with only a few problem gradients preventing degree reducibility. These problem gradients result in the flat stretches in Figure 1(b) (which will average one out of every four unit intervals!).

We solve the problem by changing the gradient selection rules. Rather than choose gradients on the edge centers as *iNoise* does, we choose gradients on at the cube corners.



(a) Original Perlin (b) Perlin Improved

(c) cNoise (d) mNoise

Figure 4: *Comparison of 2D noise functions*

This new noise will be referred to as *cNoise*. Perlin notes the possibility of axis-aligned clumping with this distribution [Per02], but as can be seen in Figure 3, *iNoise* still ends up using these same corner gradients in lower-dimensional slices of the noise. Comparing the noise results (Figure 4), we see that the clumping artifacts are not severe. We feel the dimension reducibility far outweighs any minor artifacts.

Note that the gradient **vectors** will differ for 3D, 2D, and 1D, but the *grad* function will produce identical results. For example, gradient vectors $(1,1,1)$, $(1,1)$, 1 are not even the same dimensonality. But the 3D *grad* function is $\vec{p}_f^x + \vec{p}_f^y + \vec{p}_f^z$. In any integer-z plane (including z=0), this becomes $\vec{p}_f^x + \vec{p}_f^y + 0$, which is exactly equal to the 2D *grad* function, $\vec{p}_f^x + \vec{p}_f^y$. Similarly, for integer-z **and** integer-y, the 3D *grad* is $\vec{p}_f^x + 0 + 0$.

3.3. Hashing

Gradients for *oNoise* and *iNoise* are both chosen using the same hashing function; *oNoise* from a table of unit gradient vectors, and *iNoise* from bits of the hash. Each lattice point $\vec{p}_i$ is hashed using repeated applications of a small permutation table. This table is precomputed to map each integer from 0-255 to a unique new integer between 0 and 255. The final single hash value is

$$hash3(\vec{p}_i) = permute(permute(permute(\vec{p}_i^x) + \vec{p}_i^y) + \vec{p}_i^z)$$

Note that the lower-dimensional hash functions are just a slice out of the higher dimensional hash, offset by $permute^{-1}(0)$:

$$\begin{aligned} hash2(\vec{p}_i) &= permute(permute(\vec{p}_i^x) + \vec{p}_i^y) \\ &= hash3(permute^{-1}(0), \vec{p}_i^x, \vec{p}_i^y) \\ hash1(\vec{p}^x) &= permute(\vec{p}^x) \\ &= hash2(permute^{-1}(0), \vec{p}^x) \end{aligned}$$

cNoise uses this same hash function as well — the only difference from *iNoise* is in the gradient vector selection. Computed gradients eliminate one possible lookup into precomputed values, but the hash itself is a major bottleneck for GPU computation [Wei04]. The permutation table must be computed in advance and stored, either into large 2D and 3D textures, or needing multiple chained lookups.

If we instead had a hash function that could be computed on the fly, we could achieve the goal of a purely computable noise. The requirements for a computed hash function are that it take few instructions, be repeatable, and that there be no noticeable correlation between nearby values. Note that, while the permutation table hash is one-to-one for values between 0 and 255, this is not a necessity.

Quasi-random number generators, such as Sobel or Halton sequences, are common for jittered sampling in Monte-Carlo methods [MC95]. They are easy to compute, but prove far too correlated for our purposes. Pseudo-random number generators have many of the features we desire, but tend to be designed to have little correlation between **successive values** [Cov60], so random number n is uncorrelated to random number $n+1$. To use as a good hash function, we need a way to jump to step 50 without having to call the random number generator 50 times, or we need it to be uncorrelated relative to **successive seeds**. These constraints rule out many of the common generators.

Linear congruential generators [Knu81] appear too correlated when using successive seeds. The high-order bits of successive values would work, and a closed form to jump to value n does exist, but is expensive to compute robustly due to the large exponents involved. Lagged Fibonicci generators [Knu81] have similar characteristics, but the closed form to jump to value n involves a matrix exponentiation.

We observed better success with a modification of the Blum Blum Shub (BBS) pseudo-random generator [BBS86]. The BBS generator computes

$$x_{i+1} = x_i^2 \bmod M$$

Where $M = pq$, for large primes p and q. The low order bit or bits form the random output, and are all we need to generate gradients by the same method as *cNoise*. The period of the resulting noise would be M. In CPU applications, BBS is generally regarded as cryptographically secure, but too expensive for non-cryptographic use [Jun99]. We observe excellent results in CPU simulation when using the lattice location as the seed and performing only one or two steps of the generator, achievable in 4-5 low-level shading instructions per step. Note that GPU color-channel parallelism allows computation of hash values for four lattice points simultaneously.

The five instruction version implements $i \bmod M$ as $i - floor(i/M) * M$. The four instruction version implements it

as $frac(i/M)*M$, but suffers from precision problems. The latter version produces visually acceptable noise results, but numerical inaccuracies result in a noise that does not repeat at M as it should. We can use the first, exact, mod to compute 2D noise (where matching along texture seams is desirable), but the latter, inexact, mod for 3D noise, where the noise repeat factor is less of an issue.

Since current GPUs perform even integer computation in floating point, reasonable values for M are simply too large for small-format floating point numbers to hold without loss of precision in intermediate computation. While it removes any claim to cryptographic quality, we found a sufficient lack of correlation for several prime M of a size small enough to prevent the chance of overflow. All examples shown here use $M = 61$. The maximum intermediate value of $61^2 = 3721$ is well within the range exactly representable as an integer on these machines. This choice is a tradeoff between portability (to small floats) and quality.

Like the Perlin noise functions, we construct a multidimensional hash as follows.

$$\begin{aligned} hash3(\vec{p}_i) &= hash(\vec{p}_i^{\,x} + hash(\vec{p}_i^{\,y} + hash(\vec{p}_i^{\,z}))) \\ hash(x) &= x^2 \bmod M \end{aligned}$$

We call the resulting modified noise, combining corner-based gradients and the computed hash, *mNoise*. An example of *mNoise* is shown in Figure 5. *mNoise* is compared to several other noise functions in Table 2.

3.4. Separable Computation

While the modified noise can be computed totally in a GPU vertex or fragment shader, our low-level OpenGL shading code for 2D noise is 45 instructions. That is simply too many for many applications, especially for a function typically used several times in a single shader. Instead, we prefer to use texture lookup for 1D and 2D noise. Results for 2D *mNoise* for both a 128^2 and 256^2 texture are shown in Figure 2. Note that, while 128^2 is above the Nyquist limit, it is not sufficient to capture the character of the noise function.

The 3D noise is still large to store as a full texture, but even more expensive for pure computation. Instead, we factor the 3D noise into groups of x-y expressions. The resulting form involves two x-y terms for the integer z value below the noise argument, and two for the integer z above. Thanks to the dimension reducibility, one of these terms is exactly the 2D noise function! The other is the z component of the gradient (missing in the 2D noise) blended in x-y according to the *flerp* function.

We can store each of these x-y terms (2D noise and z-gradient) into a single 256^2 texture, accessed using:

$$\begin{aligned} \vec{c}_0 &= (\vec{p}^{\,x}, \vec{p}^{\,y} + hash(\vec{p}_i^{\,z})) \\ \vec{c}_1 &= (\vec{p}^{\,x}, \vec{p}^{\,y} + hash(\vec{p}_i^{\,z} + 1)) \end{aligned}$$

The final 3D noise using separate 2D noise and z-gradient textures is

$$\begin{aligned} flerp(\vec{p}_f^{\,z}, \quad & mNoise2(\vec{c}_0) + \vec{p}_f^{\,z} * zgrad(\vec{c}_0), \\ & mNoise2(\vec{c}_1) + \vec{p}_f^{\,z} * zgrad(\vec{c}_1)) \end{aligned}$$

Or, with a single combined 2-channel texture, we see computation similar to computed 1D noise (Figure 6)

$$flerp(\vec{p}_f^{\,z},\ (1, \vec{p}_f^{\,z}) \bullet tex(\vec{c}_0),\ (1, \vec{p}_f^{\,z}) \bullet tex(\vec{c}_1))$$

4D is similar to computed 2D noise, using a 3-channel 2D texture containing noise, z-gradient, and w-gradient.

$$\begin{aligned} flerp(\vec{p}_f^{\,w}, & flerp(\vec{p}_f^{\,z},\ (1,z,w) \bullet tex(\vec{c}_{00}),\ (1,z,w) \bullet tex(\vec{c}_{01})) \\ & flerp(\vec{p}_f^{\,z},\ (1,z,w) \bullet tex(\vec{c}_{10}),\ (1,z,w) \bullet tex(\vec{c}_{11}))) \end{aligned}$$

where

$$\begin{aligned} &\vec{c}_{jk} = (\vec{p}^{\,x}, \vec{p}^{\,y} + hash(\vec{p}_i^{\,z} + k + hash(\vec{p}_i^{\,w} + j))) \\ &j,k \in \{0,1\} \end{aligned}$$

4. Using *mNoise*

It is common to combine several octaves of noise in order to produce a random function with a more complex spectrum. One of the most common such compositions is *turbulence* [Per85], constructed as

$$\|noise(\vec{p})\| + \frac{1}{2}\|noise(\frac{1}{2}\vec{p})\| + \frac{1}{4}\|noise(\frac{1}{4}\vec{p})\| + ...$$

While we can perform independent calls to the noise function, the common computation allows for a more efficient single turbulence function. For a 3D turbulence built from 2D noise textures, we can compute the *hash* function for up to two octaves together, and the *flerp* and sum for up to four octaves together. Figure 7(a) shows 4-octave *turbulence*. Figures 7(b) and 7(c) show examples of 3D wood and marble shaders using the 3D turbulence function based on a 2D noise texture. All figures and video were generated with an ATI 9700-equipped laptop running OpenGL 1.5 using single-pass fragment shaders.

5. CONCLUSIONS AND FUTURE WORK

We have presented two modifications to Perlin's improved noise function that increase its potential for hardware-based implementation. The resulting noise is simple and inexpensive to evaluate and produces results comparable to the original CPU noise functions. In addition, our noise has the advantage that the same noise function can be computed without any texture, or efficiently evaluated with a single texture usable for all dimensions of noise. Sample implementations and more details are are online at www.umbc.edu/~olano/noise/.

In the future, we would like to explore more fully other factorizations for the turbulence and fBm functions. We would also like to apply our computed hash function to

noise	period (p)	dimension (n)	texture sizes (size×components)	texture accesses	longest chain	computation notes
any, baked to texture	any	$1-3$	values: $\geq (3p)^n \times 1$	1	1	no computation
[Per85]	256	$1-4+$	hash: $p \times 1$; grad: $p \times 1$	$2^{n+1}-2$	n	expensive
[Per02],cNoise	256	$1-4+$	hash: $p \times 1$	$2^{n+1}-2$	n	similar to [Per85]
[Gre05]	256	3	hash: $p^2 \times 4$; grad: $p \times n$	8	2	merge lookups to 2D textures
[Per04]	72	$1-3$	complex values: $\geq 25^n \times 2$ $k^n \vert k = 3q+1; q \cdot (q+1) = p$	2	1	simple (w/ trig); to match without 3D texture, need 4 3D noise calls
computed mNoise	61	$1-4+$	—	—	—	similar to [Per85]; no lookups
mixed mNoise	61	3+	value+grad: $\geq (3p)^2 \times (n-1)$	2	1	like (n-2)-D iNoise; matches computed

Table 2: *Comparison of texture-based noise implementations*

other hash-based procedural primitives, including *cellnoise* [Ups90], Worley's n-th closest point cellular texture basis function [Wor96], or tiled texture mapping [Wei04].

References

[BBS86] BLUM L., BLUM M., SHUB M.: A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing 15*, 2 (May 1986), 364–383.

[Cov60] COVEYOU R. R.: Serial correlation in the generation of pseudo-random numbers. *J. ACM 7*, 1 (1960), 72–74.

[EMP*02] EBERT D. S., MUSGRAVE F. K., PEACHEY D., PERLIN K., WORLEY S.: *Texturing and Modeling: A Procedural Approach*, third ed. Morgan Kaufmann, 2002.

[Gre05] GREEN S.: Implementing improved perlin noise. In *GPU Gems 2*, Pharr M., (Ed.). Addison-Wesley, 2005, ch. 26.

[Gus05] GUSTAVSON S.: Simplex noise demystified, March 2005. http://www.itn.liu.se/~stegu/simplexnoise/.

[Har01] HART J. C.: Perlin noise pixel shaders. In *Graphics Hardware 2001* (Los Angeles, CA, August 2001), Akeley K., Neumann U., (Eds.), SIGGRAPH/EUROGRAPHICS, ACM, New York, pp. 87–94.

[Jun99] JUNOD P.: Cryptographic secure pseudo-random bits generation : the blum-blum-shub generator. http://crypto.junod.info/bbs.pdf, 1999.

[Knu81] KNUTH D.: *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA., 1981.

[Lew89] LEWIS J. P.: Algorithms for solid noise synthesis. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques* (1989), ACM Press, pp. 263–270.

[LMOW95] LASTRA A., MOLNAR S., OLANO M., WANG Y.: Real-time programmable shading. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics* (1995), ACM Press, pp. 59–ff.

[MC95] MOROKOFF W. J., CAFLISCH R. E.: Quasi-Monte Carlo integration. *J. Comp. Phys. 122* (1995), 218–230.

[New01] NEW LINE CINEMA: Lord of the rings. Motion Picture, 2001.

[Per85] PERLIN K.: An image synthesizer. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques* (1985), ACM Press, pp. 287–296.

[Per01] PERLIN K.: Noise hardware. In *Real-Time Shading SIGGRAPH Course Notes* (2001), Olano M., (Ed.).

[Per02] PERLIN K.: Improving noise. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (2002), ACM Press, pp. 681–682.

[Per04] PERLIN K.: Implementing improved perlin noise. In *GPU Gems*, Fernando R., (Ed.). Addison-Wesley, 2004, ch. 5.

[Pix95] PIXAR/DISNEY: Toy story. Motion Picture, 1995.

[Ups90] UPSTILL S.: *The RenderMan companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley, 1990.

[vW91] VAN WIJK J.: Spot noise - texture synthesis for data visualization. In *Computer Graphics (Proceedings of ACM SIGGRAPH 91)* (July 1991), Sederberg T. W., (Ed.), pp. 309–318. ISBN 0-201-56291-X.

[Wei04] WEI L.-Y.: Tile-based texture mapping on graphics hardware. In *Graphics Hardware* (August 2004), Akenine-Mo̲ller T., McCool M., (Eds.), Eurographics/ACM SIGGRAPH, ACM Press.

[Wor96] WORLEY S.: A cellular texture basis function. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (1996), ACM Press, pp. 291–294.

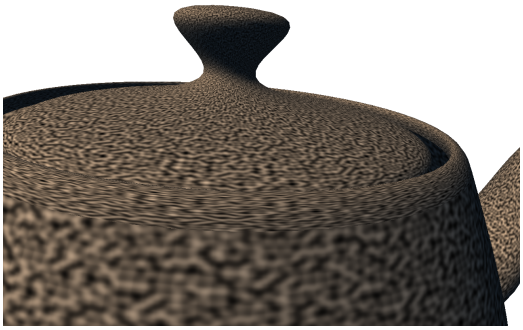


Figure 5: *2D mNoise, mapped onto a teapot. Note the changes in density with texture parameterization*

Figure 6: *3D mNoise built from two accesses to one* 256^2 *texture. Noise is uniform in size throughout and continuous even across the junction between pot and spout.*

(a) (b) (c)

Figure 7: *3D mNoise turbulence, and wood and marble shaders built using it*