Perlin Noise Pixel Shaders

John C. Hart

University of Illinois, Urbana-Champaign

Abstract

While working on a method for supporting real-time procedural solid texturing, we developed a general purpose multipass pixel shader to generate the Perlin noise function. We implemented this algorithm on SGI workstations using accelerated OpenGL PixelMap and PixelTransfer operations, achieving a rate of 2.5 Hz for a 256x256 image. We also implemented the noise algorithm on the NVidia GeForce2 using register combiners. Our register combiner implementation required 375 passes, but ran at 1.3 Hz. This exercise illustrated a variety of abilities and shortcomings of current graphics hardware. The paper concludes with an exploration of directions for expanding pixel shading hardware to further support iterative multipass pixel-shader applications.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation – *Bitmap and framebuffer operations*. I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism – *Color, shading, shadowing and texture.*

Keywords: Pixel shaders, Perlin noise function, hardware shading, register combiners.

1. INTRODUCTION

The concept of procedural shading is well known [17][19], and has found widespread use in graphics [3]. Procedural shading computes arbitrary lighting and texture models on demand. Procedural textures efficiently support high resolution, non-repeating features indexed by three-dimensional solid texture coordinates. These features were quickly adopted for production-quality rendering by the entertainment industry, and became a core component of the Renderman Shading Language [5].

With the acceleration of graphics processors outpacing the exponential growth of general processors, there have been several recent calls for real-time implementations of procedural shaders, e.g. [6][20]. Real-time procedural shading makes videogames richer, virtual environments more realistic and modeling software more faithful to its final result. Real-time procedural texturing, in particular, allows modelers to use solid textures to seamlessly simulate sculptures of wood and stone. It yields complex animated environments with billowing clouds and flickering fires. Designers and users can interactively synthesize and investigate new procedural worlds that seem vaguely familiar to our own but with features unique to themselves.

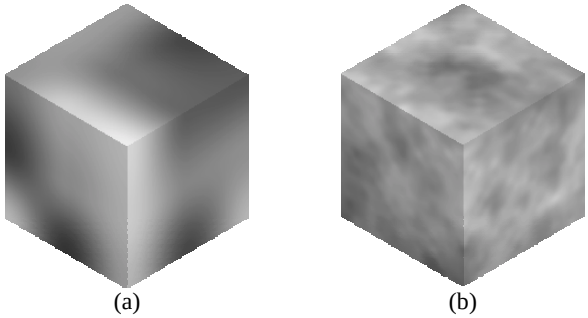
Contact info: Dept. of Computer Science, 1304 W. Springfield Ave., Urbana, IL 61801, (217) 333-8740, jch@cs.uiuc.edu.

HWWS '01 Los Angeles, CA USA

Several have researched techniques for supporting procedural shading with real-time graphics hardware [15][18][21][22]. These shading methods reorganize the architecture of the graphics API to suit the needs of procedural shading, applying API components to tasks for which they were not originally designed [8][11].

One such technique supports real-time procedural solid texturing [2] by using the texture map to store the shading of an object [1]. The technique maintains a texture atlas that maps triangles from a surface mesh into a non-overlapping array in texture memory. The triangles are plotted in texture memory using their solid texture coordinates as vertex colors. Rasterization then interpolates solid texture coordinates across their faces in the texture map. A procedural texturing pass replaces the solid texture coordinates in the texture map with the procedural texture color. Finally, this color is reapplied to the object surface via standard texture mapping. The result is a view-independent procedural solid texturing of the object.

One of the most common components of a procedural shading system is the Perlin noise function [19], a correlated three-dimensional field of uniform random values. This versatile function provides a deterministic random function whose bandwidth can be controlled to inhibit aliasing. Moreover, $1/f^{\beta}$ sums of noise functions can be used to form turbulence and other fractal structures whose statistics can be set to match those of various kinds of natural phenomena.



(a) (b)

Figure 1. Perlin noise function (a) and a $1/f$ sum (b).

We integrated the Perlin noise function into our real-time procedural solid texturing system in a variety of different ways, both as a CPU process and as a GPU process. This paper describes an algorithm for implementing the Perlin noise function as a multipass pixel shader. It also analyzes this noise implementation on a variety of systems. We used the available accelerated implementations of the OpenGL API and its device-dependent extensions on two SGI systems and an NVidia GeForce2. The paper concludes with suggestions for further hardware accelerator development that would facilitate faster

implementations of the Perlin noise function as well as a broader variety of texturing procedures.

2. PREVIOUS WORK

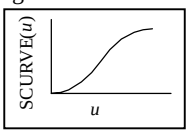
Because the Perlin noise function has become a ubiquitous but expensive tool in texture synthesis, it has been implemented in highly optimized forms on a variety of general and special purpose platforms.

Several fast host-processor methods exist for synthesizing Perlin noise. Goehring *et al.* [4] implemented a smooth noise function in Intel MMX assembly language, evaluating the function on a sparse grid and using quadratic interpolation for the rest of the values. Kameya *et al.* [10] used streaming SIMD instructions that forward differenced a linearly interpolated noise function for fast rasterization of procedurally textured triangles.

One can also generate solid noise with a 3-D texture array of random values [13], using hardware trilinear interpolation to correlate the random lattice values stored in the volumetric texture. Fractal turbulence functions can be created using multitexture/multipass modulate and sum operations. A texture atlas of solid texture coordinates would then be replaced with noise samples using the OpenGL pixel texture extension, ala [9].

The vertex-shader programming model found in Direct3D 8.0 [12] and the recent NVIDIA OpenGL vertex shader extension [16] can support procedural solid texturing. A Perlin noise function has been implemented as a vertex program [14]. But a per-vertex procedural texture produces vertex colors that are Gouraud interpolated across faces, such that the frequency of the noise function must be at, or less than half, the frequency of the mesh vertices. This would severely restrict the use of turbulence resulting from $1/f$ sums of noise. Hence the Perlin noise vertex shader is limited to low-frequency displacement mapping or other noise effects that can be mesh frequency bound.

Our favorite implementation of the Perlin noise function is from the Rayshade ray tracer [24]. This implementation created its own pseudorandom numbers by hashing integer solid texture coordinates with a scalar function Hash3d(i,j,k), then interpolated these random values with a simple smooth cubic interpolant $\text{SCURVE}(u) = 3u^2 - 2u^3$ to yield the final result.



Given solid texture coordinates s,t,r, the Rayshade noise function effectively returned noise as the value

$$\sum_{k=0}^{1}\sum_{j=0}^{1}\sum_{i=0}^{1}\text{Hash3d}(\lfloor s\rfloor+i,\lfloor t\rfloor+j,\lfloor r\rfloor+k)w(s,i)w(t,j)w(r,k)$$

where

$$w(s,i) = \text{SCURVE}(s-\lfloor s\rfloor)^{i}\,(1-\text{SCURVE}(s-\lfloor s\rfloor))^{1-i}$$

is a weighting function. Hence, the noise function returns a weighted sum of the random values at the eight corners of the integer lattice cube containing s,t,r.

Figure 2 demonstrates the result of the Rayshade implementation of the Perlin noise function. The random values are indexed from a table generated by the drand48() function. Noise is defined on an integer coordinate lattice, which results in the strong horizontal and vertical correlation.

We will use this sample as a reference to compare our pixel-shader implementations of the Perlin noise function. The average brightness of the (s,t) slice of the noise is due to the fixed r coordinate. This average intensity will differ across

Cubic interpolation Linear interpolation

Figure 2. Result of the Rayshade implementation of the Perlin noise function, using cubic (left) and linear (right) interpolation of corner lattice random values.

implementations, resulting in variations in brightness for a given (s,t) slice of the three-dimensional noise field.

3. A MULTIPASS NOISE ALGORITHM

We based our real-time implementation of the Perlin noise function on the concise Rayshade implementation. We implemented a per-pixel noise function using multipass rendering onto a texture atlas initialized with solid texture coordinates stored as pixel colors.

The Perlin noise function is defined on a field of real values, where the integer subset of its domain defines the base frequency of the noise. Implementation of the noise function requires coordinates s,t,r to range over multiple integers, though color components only range over [0,1]. Hence, given three channels (R,G,B) each with a depth of b bits[1], we use a fixed-point representation with b_i integer bits and b_f fractional bits, $b = b_i + b_f$.

Following the form of the Rayshade noise implementation, the algorithm in Figure 3 computes a random value in [0,1] at the integer lattice points, and linearly interpolates these random values across the cells of the lattice.

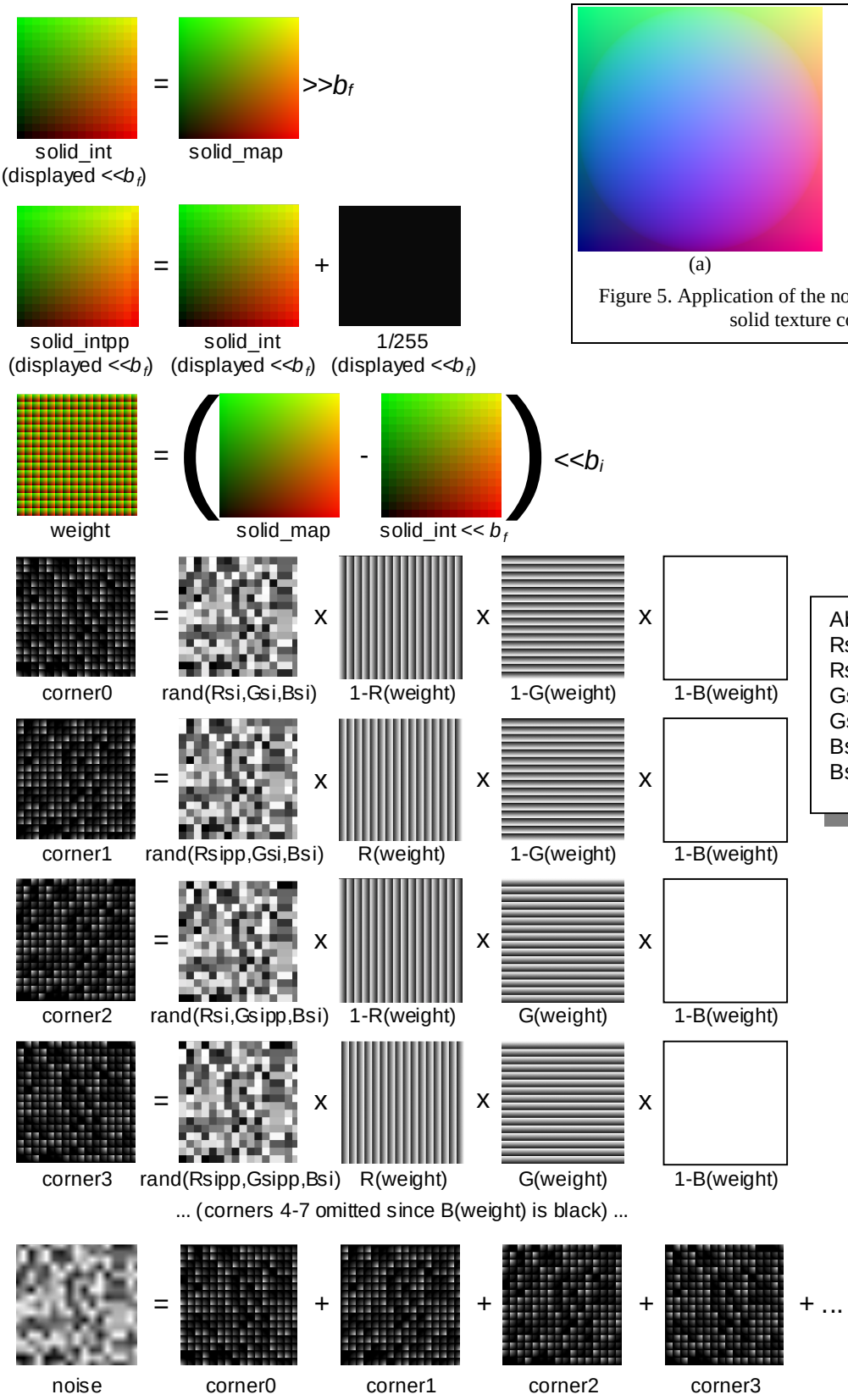
The input to the algorithm is an image solid_map whose R,G,B colors consist of solid texture coordinates. The first half of the algorithm decomposes solid_map into its integer part solid_int shifted right b_f times and a fractional part weight shifted left b_i times.

Input: 2-D texture solid_map with R,G,B containing s,t,r coordinates.
Initialize texture noise = black
texture solid_int = solid_map >> b_f
texture solid_intpp = solid_int + $1/(2^b-1)$
texture weight = (solid_map – (solid_int << b_f)) << b_i
for (k = 0; k < 8; k++) {
  texture corner = solid_int
  overwrite corner = solid_intpp with glColorMask(k&1,k&2,k&4)
  randomize corner
  corner *= if (k&1) then R(weight) else 1 – R(weight)[2]
  corner *= if (k&2) then G(weight) else 1 – G(weight)
  corner *= if (k&4) then B(weight) else 1 – B(weight)
  noise += corner
}
Output: solid noise texture map

Figure 3. Multipass Perlin noise algorithm.

[1] Framebuffers currently hold only 8 or 12 bits per channel though there is an extension that supports 32-bit floating point, and indications that floating point buffers may soon be supported by a larger variety of graphics hardware and drivers.

[2] The functions R(), G() and B() return a luminance image of the corresponding channel.



(a) (b)

Figure 5. Application of the noise function (b) on a sphere of solid texture coordinates (a).

Figure 4. Intermediate images in the multipass Perlin noise algorithm.

Figure 4 shows solid_map as a sample input texture map of solid texture coordinates as a plane of two-dimensional solid texture coordinates spanned by *s* and *t*, with *r* = 0 We set $b_f = 4$ bits. The solid texture coordinates *s*,*t*,*r* range from (0.0,0.0,0.0) to (15.9375,15.9375,0.0) and are represented in the texture map with RGB colors from (0,0,0) to (1,1,0). Internally in the 24bpp framebuffer, these RGB colors range from (0,0,0) to (255,255,0). These coordinates are shifted right by b_f to form solid_int, which is shown in Figure 4 shifted left by b_f to increase contrast and brightness. Subtracting (b) from (a) leaves solid_frac (not shown), which is shifted left by b_i to create a normalized weight function. This weight function forms a linear basis for the noise function.

The color (*R*,*G*,*B*) of each pixel (*x*,*y*) in solid_map corresponds to a solid texture point (*s*=*R*,*t*=*G*,*r*=*B*) that falls within some lattice cell. The corner of this cell is given by the coordinates in the corresponding pixel (*x*,*y*) stored in solid_int. The opposite corner of this cell is found in the corresponding pixel in solid_intpp (whose colors are increments of those in solid_int).

The integer solid texture coordinates of the eight corners of the cell can be found as a combination of the color channels *R*,*G*,*B* of the same pixel location in solid_int and solid_intpp. In Figure 4 these combinations are abbreviated Rsi, Rsipp, etc.

The second half of the algorithm iterates over these eight corners, creating a random value indexed by the integer value at that corner. This random number function is denoted rand() in Figure 4.

These random values are weighted by the fractional portion of the solid texture coordinates found in weight or its additive inverse. Summing the products of these weights for each of the eight corners corner0 … corner7 performs a trilinear interpolation of the random values at the corners, resulting in the image noise.

We will spend the next two sections implementing this algorithm using the available accelerated features of two different graphics architectures. These implementations are each divided into two sections, on implementing the logical shift operations needed for the first half of the algorithm, and the random value synthesis needed for the second half.

4. SGI IMPLEMENTATION

The SGI graphics accelerators have focused on high-end real-time rendering for the scientific visualization and entertainment production communities. Hence accelerated features have included scientific imaging functions that support algebraic and lookup-table operations on pixels.

We focused our implementation on midline SGI workstations, which are commonly deployed for digital content creation and design in both the videogame and animation communities.

4.1 PixelTransfer and PixelMap

We implemented the noise function in multipass OpenGL on SGI workstations using accelerated PixelTransfer[3] and PixelMap functions. The PixelTransfer function performs a per-component scale and bias, whereas PixelMap performs a per-component lookup into a predefined table of values.

[3] Following the convention of the OpenGL ARB, we avoid the use of the "gl" prefix for functions and the "GL_" prefix for tokens when describing elements of the OpenGL API.

We defined an assembly language of useful PixelTransfer functions. Specifically, the function setPixelTransfer(*a*,*b*) sets OpenGL to perform an $ax + b$ operation during the next image transfer operation, where *x* represents each component of the RGBA color. The function setPixelMap(table) uses PixelMap to replace colors channels with their corresponding entries in a lookup table. We also defined a blendtex(*i*) operation that draws the texture image corresponding to texture index *i*. The instruction savetex(*i*) saves the current framebuffer as texture image *i*.

Unlike the basic algorithm described in the previous section, the SGI implementation begins with three luminance images tex_s, tex_t and tex_r instead of a single RGB image solid_map. Using the notation of Figure 4, tex_s = R(solid_map), tex_t = G(solid_map), and tex_r = B(solid_map).We could perform all of the decompositions on a single texture, but we would later need to break its red, green and blue channels into individual luminance textures, and we found it impossible to perform this efficiently with the OpenGL extension set available to low-end and midline SGI workstations that lacked the color_matrix extension.

4.2 Logical Shift Operations

The task of decomposing a texture map of fixed point solid texture coordinates into integer and fractional textures used PixelTransfer multiplication to achieve shifting operations. We defined an integer shift = $1 << b_f$. We modulated the texture by shift to perform a logical shift left by b_f, and by 1/shift to perform a logical shift right. (Some hardware required us to round instead of truncate, which was performed by a PixelTransfer bias of -0.5/255.0.) We also defined fracshift as $255.0/((1 << b_f) - 1)$. This allowed us to scale our fractional portions into normalized weights.

The following code fragment demonstrates the decomposition of the *s* coordinate. Similar decompositions need to be performed on tex_t and tex_r as well.

```
// shift s right to remove fractional part, save as si
blendtex(tex_s);
setPixelTransfer(1.0/shift, 0.0 /* or –0.5/255.0 */);
savetex(tex_si);
resetPixelTransfer();

// shift si back left
blendtex(tex_si);
setPixelTransfer(shift, 0.0);
CopyPixels(0,0,HRES,VRES,COLOR);
resetPixelTransfer();

// subtract si (floor of s) from s to get fractional part of s
Enable(BLEND);
BlendEquation(SUBTRACT);
BlendFunc(1, 1);
blendtex(tex_s);
Disable(BLEND);

// scale fractional part into normalized weight in [0,1]
setPixelTransfer(fracshift, 0.0);
savetex(tex_sf);
resetPixelTransfer();
```

4.3 Random Value Synthesis

We implemented randomization using a lookup table. This lookup table was accessed using the accelerated PixelMap OpenGL function. Recall the value k ranges from 0 to 7 denoting the current corner. The following code fragment synthesizes a random field based on the *s* coordinate.

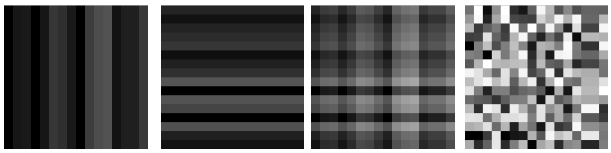
```
// tex_sin = random(si) or random(si++)
blendtex(tex_si);
setPixelTransferf(1.0, (k&1) ? 1.0/255.0 : 0.0);
setPixelMap(sran);
savetex(tex_sin);
```

Similar code fragments apply to the *t* and *r* coordinates, using (k&2) and (k&4) in the PixelTransfer, respectively. At this point tex_sin, tex_tin and tex_rin contain random values indexed by the *s*,*t*,*r* values at the *k*th corner of the cell. The following code fragment combines these three random values into a single random value.

```
// now tex_sin, tex_tin and tex_rin are random
// add them up into a single random number[4]
blendtex(tex_sin);
Enable(BLEND); BlendFunc(ONE,ONE);
blendtex(tex_tin);
blendtex(tex_rin);
Disable(BLEND);
```

This combination of random values is highly correlated due to the componentwise combination of random values. We reduce this correlation with an additional randomization pass.

```
// one more randomization (in place)
setPixelMap(nran);
CopyPixels(0,0,HRES,VRES,COLOR);
resetPixelTransfer();
```



(a) (b) (c) (d)

Figure 6. The sum of random numbers indexed by *s* (a). and *t* (b) is highly correlated (c). This correlation is reduced by indexing into a final randomization (d).

The random number tables sran, tran and rran are uniform random number distributions over the range [0,1/3]. These three random values are added to form the final distribution, which is slightly non-uniform and heavily coordinate correlated, as shown in Figure 6(c). An additional randomization reduces this correlation as shown in Figure 6(d).

Figure 4 shows the random values generated at the four corners of the lattice as the result of the rand() function. Note that in this example in the *s*,*t* plane these are all translates of each other.

The random value is then weighted by the fractional part of the original texture coordinates s,t,r. Note that we have broken out the original RGB image weight from the previous section into three luminance images tex_sf, tex_tf and tex_rf.

```
// displayed texture now random value at corner k
// weight this contribution by fractional parts of s,t,r
Enable(BLEND);
BlendFunc(0, (k&1) ? SRC : 1 - SRC);
blendtex(tex_sf);
blendFunc(0, (k&2) ? SRC : 1 - SRC);
blendtex(tex_tf);
BlendFunc(0, (k&4) ? SRC : 1 - SRC);
blendtex(tex_rf);
```

[4] Note the addition of the component random values introduces a slight Gaussian bias to the resulting noise. This could be eliminated if an accelerated exclusive-or blending mode was available.

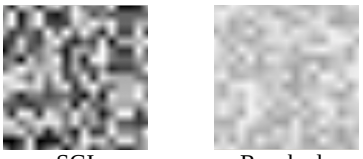
Figure 4 shows the random values at the corners corner0 … corner7 scaled by the product of weighting functions, in this implementation tex_sf, tex_tf and tex_rf. These weighting functions are luminance textures corresponding to the individual channels of weight, such that weight = (tex_sf, tex_tf, tex_rf).

The resulting weighted random value corresponding to the current corner is then added into a running total, as show in the following fragment.

```
// add noise component into noise sum
BlendFunc(1,1);
blendtex(tex_noise);
Disable(BLEND);

// keep track of sum
savetex(tex_noise);
```

The texture tex_noise is initialized to black. After all eight corners have been visited, tex_noise contains the final noise values corresponding to the solid texture coordinates in the input luminance images tex_s, tex_t and tex_r.



SGI Rayshade

Figure 7. Noise function resulting from the SGI implementation (left) compared to the reference Rayshade (linearly interpolated) implementation (right).

4.4 Results

Figure 7 shows the final noise function resulting from the SGI implementation compared to the Rayshade implementation. The correlation from Figure 6(c) was reduced by the randomization in Figure 6(d) but is still evident, particularly in the final interpolated version, as strong horizontal and vertical tendencies in the noise. However, this correlation is also found in the reference noise implementation, and is primarily due to the integer lattice of noise values.

The contrast of the SGI result is also higher than the contrast of the Rayshade result. The range of the noise function can be easily manipulated by scaling the image through a modulation pass.

We implemented this algorithm at a resolution of 256^2 on a SGI Solid Impact, a SGI Octane, and an NVidia GeForce2. The SGI workstations are designed for advanced imaging applications and have hardware accelerated PixelTransfer and PixelMap operations whereas the NVidia card designed for mainstream consumer applications does not. The execution times are given in Table 1.

Implementation	**Execution Time (Rate)**
SGI Octane	0.4 sec. (2.5 Hz)
SGI Solid Impact	0.75 sec. (1.3 Hz)
NVidia GeForce 256	5 sec. (0.2 Hz)

Table 1. Execution results for the multipass noise algorithm.

5. NVidia IMPLEMENTATION

We also implemented a noise function for consumer-level accelerators using the NVidia chipset. The NVidia products have been designed to accelerate commodity personal computer graphics, especially videogames. Hence the drivers did not

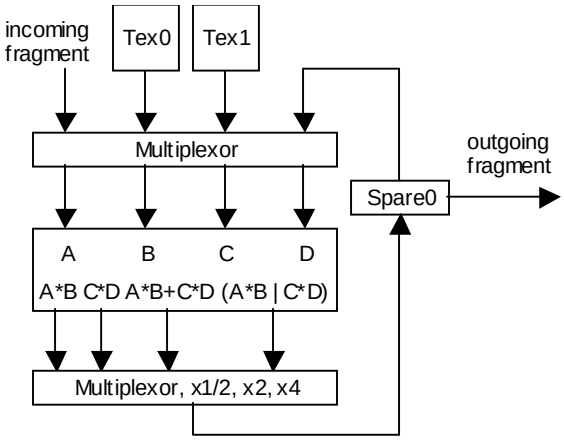


Figure 8. Partial block diagram of the register combiner functionality used in this paper.

accelerate PixelTransfer and PixelMap. We instead used register combiners to shift, randomize and isolate/combine components.

5.1 Register Combiners

Register combiners support very powerful per-pixel operations by combining multitextured lookups in a variety of manners. They support the addition, subtraction and component-wise multiplication (and even a dot product) of RGB vectors. They also support conditional operations based on the high-bit of the alpha channel of one of the inputs. They support signed byte arithmetic with a full 9 bits per channel, though can only store 8 bit results. They also provide several mapping functions for signed/unsigned conversion, and the ability to modulate output values by one-half, two and four.

The Direct3D 8.0 specification includes a register-combiner based assembly language [12]. However, our implementation sought to squeeze the best possible performance out of the NVidia chipset. We chose instead to use the OpenGL register combiner extensions, which provide complete, though device dependent, access to the graphics accelerator.

Figure 8 illustrates the register combiner functionality used in this paper. The register combiner has four inputs A,B,C,D that can be any combination of the incoming fragment, a pixel from multitexture unit 0 or 1, and the contents of a scratch register called Spare0. The constants zero and one (via a special unsigned invert operation) can also be used as inputs, and other constant values can also be loaded via special registers.

The outputs of the register combiners include A*B, C*D, A*B + C*D and the special A*B | C*D. This latter output yields A*B if the alpha component of the register Spare0 is less than 0.5, otherwise the output yields C*D. These outputs can also be optionally scaled by ½, 2 or 4. For this paper, it is safe to assume the output is always contained in the register Spare0. The register combiner has separate but comparable functions for the RGB values and the alpha values of the inputs and registers.

There can be any number of register combiners that form a pipeline, using the temporary registers such as Spare0 to hold data between stages. The GeForce2 used to implement the pixel shaders in this paper contains two register combiners which allow two register combiner operations per pass. The GeForce3 is expected to have eight register combiners.

5.2 Logical Shift Operations

In order to perform the decomposition of the input solid texture coordinate image into integer and fractional components, we developed a logical shift left register routine. This routine used the x2 output mapping, but this causes values greater than one half to clamp to one. We avoided this overflow by using the conditional mode of the register combiners. The following example sets up the register combiners to perform such a logical shift left on a luminance value (R=G=B) in multitexture unit 0.

```
// first stage
// spare[α] = texture0[b]
A[α] = texture0[b]
B[α] = 1 (zero with unsigned_invert)
spare0[α] = A[α]*B[α]
// spare0 rgb = texture0 less its high bit (or zero if less than ½)
A[rgb] = texture0[rgb]
B[rgb] = white (zero with unsigned_invert)
spare0[rgb] = A[rgb]*B[rgb] - 0.5 // via bias_by_negative_one_half

// second stage
// spare0 rgb = (spare0[α] < 0.5 ? texture0[rgb] : spare0[rgb]) << 1
A[rgb] = texture0[rgb]
B[rgb] = white
C[rgb] = spare0[rgb]
D = white
spare0[rgb] = 2*(spare0[α]<0.5 ? A[rgb]*B[rgb] : C[rgb]*D[rgb])
```

We could also generate a register combiner to perform a logical shift right using the x½ output mode, but found it was much simpler to perform a multitextured modulate-mode blend with a texture consisting of the single pixel containing the RGB color (0.5,0.5,0.5).

5.3 Random Value Synthesis

Randomization on the NVidia controller was particularly difficult. The driver (and presumably the hardware) accelerated neither pixel transfer/mapping operations, nor logical operations like exclusive-or.

We instead implemented a register combiner random number generator by shifting each of the components of the integer values of the coordinates left one bit at a time. All four bits of each of the three components are at one point the high bit in multitexture unit 0. The register combiner's conditional mode will then display one of two colors depending on the high bit of the current texel of multitexture unit 0.

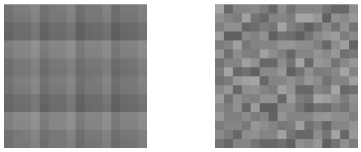
```
for (kk = 0; kk < 4; kk++) {
   for (comp = 0; comp < 3; comp++) {
      // display either tex_ranzero or tex_ranone
      // depending on hi bit of tex_comp
      setupblendhibit(ranzero[comp][kk],ranone[comp][kk]);
      blend2tex(tex_comp[comp],tex_corran); [5]
      savetex(tex_corran);
      if (kk < 3) {
         // shift tex_comp left one
         setupshift1();
         blendtex(tex_comp[comp]);
         savetex(tex_comp[comp]);
} } }
```

The arrays ranzero and ranone were initialized with random luminances. These random luminances were used as input to the function setupblendhibit(rgba0,rgba1). This function set up a

[5] The operation blend2tex(tex_a,tex_b) displays a multitextured image with tex_a as multitexture unit 0 and tex_b as multitexture unit 1.

register combiner that would display either constant color rgba0 or rgba1 depending on the high bit of texture0, and would blend the color (rgba0 or rgba1) with texture1.

We found that setting the alpha channel of rgba0 and rgba1 to 1/8 provided a reasonable balance of colors after twelve successive blending operations. These blends were accumulated in tex_corran (corner random). Note that this loop involves 12 randoms + 9 shifts = 21 passes, which expands to 168 passes for all eight corners.



(a) (b)

Figure 9. Heavily correlated random values generated by blending random colors depending on the bits of the integer lattice value (a). Using (a) to index into a random value reduces the correlation (b).

The resulting tex_corran still exhibited some coordinate correlation, which we reduced with an additional eight single-bit randomizations on tex_corran, yielding tex_corranran. This step resulted in an additional 8 randoms + 7 shifts = 15 passes per corner for a total of 120 passes.

Due to the successive blending, the register combiner noise function is Gaussian distributed. A normal distribution could be recovered through a histogram equalization step, though such operations are not yet accelerated on consumer-level hardware.

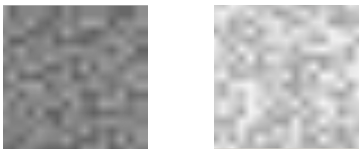
5.4 Results

The register combiner implementation resulted in 375 passes, but runs in .77 seconds at a resolution of 256^2 on a GeForce2 using version 12.0 of the "developer" driver. This results in a 1.3 Hz performance, which is suitable for interactive applications but is not yet real-time. A discussion of the reasons why the performance is slower than necessary is given later in Section 6.2.

The resulting noise is shown in Figure 10. As before, the brightness and contrast can be altered to create an even closer match. The NVidia implementation blended random colors, yielding Gaussian noise, whereas the reference and SGI implementations produced white noise. If desired, one could redistribute the Gaussian noise into white noise with a fixed histogram equalization step, though no such operation is currently accelerated on NVidia GPUs.

6. DISCUSSION

The implementation of the Perlin noise function on SGI and NVidia GPUs has been successful in that we found it was feasible, but disappointing in that subtle hardware limitations prevent truly efficient implementations. These limitations included the limited precision available in the 8 bit per component framebuffer, the delay in performing a CopyTexSubImage transfer from the framebuffer to the texture memory, and the lack of acceleration of logical operation blend modes such as exclusive-or. The process has also been illuminating, and has inspired us with several ideas for further advancement in hardware design to overcome these limitations and better support efficient multipass pixel shading.



NVidia Rayshade

Figure 10. Noise function resulting from register combiners (left) compared to the reference implementation (right).

6.1 Limited Precision

Most of the per-pixel operations need only a single channel, and set R=G=B since this is the most efficient mode of operation. The register combiners can be implemented to a higher precision, but their input and output precision is limited to the framebuffer precision.

The register combiners currently support a conversion between 8-bit unsigned external values and 9-bit signed internal values. These conversions perform the function $f(x) = 2x - 1$ on an input, and $f^{-1}(x) = 0.5x + 0.5$ on the output, where x is each of the components of an RGBA pixel.

We could likewise create a packed luminance conversion to the input and output of the register combiners. The input mapping would perform the function L = R << 16 | G << 8 | B yielding a 24-bit luminance value on which one could perform scalar register combiner operations. Internally, the register combiner could maintain a 16.8 fixed-point format, and support operations such as addition, subtraction, multiplication and division using the extended range and precision of the new format. Once the operation is completed, the result may then be unpacked into the 8-bit framebuffer with the output mapping R = L >> 16, G = (L>>8)&0xff and B =L & 0xff.

6.2 Swizzle-Blits

Given the number of passes required, the register combiner performance was astounding, currently 1.3 Hz on a GeForce2 graphics accelerator at a resolution of 256x256. Profiling the code revealed that the main bottleneck was the time it took to save the framebuffer to a texture, adding an average of 2 ms per pass for 354 of the passes. OpenGL currently does not support rendering directly to texture, and the register combiner does not allow the framebuffer to be used as an input.

Whereas framebuffer memory is organized in scanline order, modern texture memory is organized into blocks and other patterns to better capitalize on spatial coherence. This coherence allows texture pixels to be more effectively cached during texture mapping operation. However, in this case the layout of texture memory is counterproductive. The cost to "swizzle" the memory into the clustered arrangement when saving a framebuffer image to texture memory dominates the execution time of iterative multipass shaders.

We have verifies this delay with a profile of the code, revealing that our CopyTexSubImage operations were taking longer than any other component of our shader. We also experimented with various resolutions and found a direct 1:1 correspondence between the number of pixels and the execution time.

Perhaps a mode can be incorporated into the graphics accelerator state that optionally defeats the spatial-coherent clustering of texture memory. This mode could be enabled during multipass shader evaluation, to eliminate the shuffled memory delay incurred during the CopyTexSubImage operations.

Alternatively, upcoming modes that support rendering directly to texture may also ameliorate this problem.

6.3 Logical Blend Modes

Blending modes such as exclusive-or and logical shifts left and right are extremely valuable when generating random values. Unfortunately these operations are not accelerated under current graphics drivers. Such operations are of the simplest to implement in hardware, and we suspect they will become accelerated as demand for them increases.

7. CONCLUSION

We have investigated the implementation of the Perlin noise function as a multipass pixel shader, implementing a general algorithm using the accelerated features from two different manufacturers.

The SGI implementation based on PixelTransfer and PixelMap operations remains faster than the NVidia implementation based on register combiners. However, we expect the additional register combiner stages available in the upcoming GeForce3 will close this gap.

The process of implementing a general-purpose procedure using GPU accelerated operations has been illuminating. We are excited by the prospect of using the GPU as a SIMD-based supercomputer. However, this vision has been stifled by the low precision available in the buffers and processors, and the latency due to slow framebuffer-to-texture memory transfers. We believe both problems can be solved with moderate changes to existing graphics accelerator architectures, and have suggested possible solution implementations.

Our noise implementation uses linear interpolation of random values on an integer lattice. One can also implement cubic interpolation at the expense of four extra passes. The function SCURVE(u) = $3u^2 - 2u^3$ can also be expressed as $uu(3-2u)$. The function 1/4 SCURVE(u) can be implement by modulating the images u, u and $3/4 - 1/2\ u$. Note the latter is necessarily scaled by ¼ to fall within the legal [0,1] OpenGL range. This result can then be scaled by 4 (either through PixelTransfer or a register combiner) to yield SCURVE(u).

We have investigated numerous methods for enhancing the performance of these multipass pixel shaders. The 2-D s-t plane examples suggested that image processing applications such as translation and convolution could be applied, but such techniques would not work for arbitrarily shaped objects in the solid texture coordinate image, such as in Figure 5.

The source code and an executable for both implementations of the Perlin noise pixel shader can be found at:

http://graphics.cs.uiuc.edu/~jch/mpnoise.zip

ACKNOWLEDGMENTS

Conversations with Pat Hanrahan and Henry Moreton were helpful in determining the cause of the 2ms CopyTexSubImage delay. This research was supported in part by a grant from the Evans & Sutherland Computer Corp. George Francis provided access to a variety of SGI workstations via the Rennaisance Experimental Lab at the NCSA. Thanks also to Nate Carr for proofreading the paper.

REFERENCES

[1] Apodaca, A.A. Advanced Renderman: Creating CGI for Motion Pictures. Morgan Kaufmann 1999. See also: Renderman Tricks Everyone Should Know, in SIGGRAPH 98 or SIGGRAPH 99 Advanced Renderman Course Notes.

[2] Carr, N.A. and J.C. Hart. Real-Time Procedural Solid Texturing. Manuscript, 2001.

[3] Ebert, D., F.K. Musgrave, D. Peachey, K. Perlin and S. Worley. Texturing and Modeling: A Procedural Approach, Academic Press.1994.

[4] Goehring, D. and O. Gerlitz. Advanced procedural texturing using MMX technology. Intel MMX Technology Application Note, Oct. 1997. http://developer.intel.com/software/idap/resources/technical_collateral/mmx/proctex2.htm.

[5] Hanrahan, P. and J. Lawson. A language for shading and lighting calculations. *Computer Graphics* 24(4), (*Proc. SIGGRAPH 90*), Aug. 1990, pp. 289-298.

[6] Hanrahan, P. Procedural shading (keynote). Eurographics / SIGGRAPH Workshop on Graphics Hardware, Aug. 1999. http://graphics.standford.edu/hanrahan/talks/rts1/slides.

[7] Hart, J. C., N. Carr, M. Kameya, S. A. Tibbits, and T.J Colemen. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. *1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, Aug. 1999, pp. 45-53.

[8] Heidrich, W. and H.-P. Seidel. Realistic hardware-accelerated shading and lighting. *Proc. SIGGRAPH 99*, Aug. 1999, pp. 171-178.

[9] Heidrich, W., R. Westermann, H-P Seidel and T. Ertl. Applications of Pixel Textures in Visualization and Realistic Image Synthesis. Proc. ACM Sym. on Interactive 3D Graphics, Apr. 1999, pp. 127-134.

[10] Kameya, M. and J.C. Hart. Bresenham noise. *SIGGRAPH 2000 Conference Abstracts and Applications*, July 2000.

[11] McCool, M.C. and W. Heidrich. Texture Shaders. *1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, Aug. 1999, pp. 117-126.

[12] Microsoft Corp. Direct3D 8.0 specification. Available at: http://www.msdn.microsoft.com/directx.

[13] Mine, A. and F. Neyret. Perlin Textures in Real Time using OpenGL. Research Report #3713, INRIA, 1999. http://www-imagis.imag.fr/Membres/Fabrice.Neyret/publis/RR-3713-eng.html

[14] NVidia Corp. Noise, component of the NVEffectsBrowser. Available at: http://www.nvidia.com/developer.

[15] Olano, M. and A. Lastra. A shading language on graphics hardware: The PixelFlow shading system. Proc. SIGGRAPH 98, July 1998, pp. 159-168.

[16] OpenGL Architecture Review Board. OpenGL Extension Registry. Available at: http://oss.sgi.com/projects/ogl-sample/registry/

[17] Peachey, D.R. Solid texturing of complex surfaces. *Computer Graphics* 19(3), July 1985, pp. 279-286.

[18] Peercy, M.S., M. Olano, J. Airey and P.J. Ungar. Interactive multi-pass programmable shading, Proc. SIGGRAPH 2000, July 2000, pp. 425-432.

[19] Perlin, K. An image synthesizer. *Computer Graphics* 19(3). July 1985, pp. 287-296.

[20] Pixar Animation Studios. Future requirements for graphics hardware. Memo, 12 April 1999.

[21] Proudfoot, K., W.R. Mark, S. Tzvetkov and P. Hanrahan. A real-time programmable shading system for programmable graphics hardware. Proc. SIGGRAPH 2001, Aug. 2001, to appear.

[22] Rhoades, J., G. Turk, A. Bell, U. Neumann, and A. Varshney. Real-time procedural textures. 1992 *Symposium on Interactive 3D Graphics* 25(2), March 1992, pp 95-100.

[23] Segal, M. and K. Akeley. The OpenGL Graphics System: A Specification, Version 1.2.1. Available at: http://www.opengl.org/.

[24] Skinner, R. and C.E. Kolb. noise.c component of the Rayshade ray tracer, 1991.