Fast High-Quality Noise

Jeppe Revall Frisvad
Technical University of Denmark

Geoff Wyvill
University of Otago

Abstract

At the moment the noise functions available in a graphics programmer's toolbox are either slow to compute or they involve grid-line artifacts making them of lower quality. In this paper we present a real-time noise computation with no grid-line artifacts or other regularity problems. In other words, we put a new tool in the box that computes fast high-quality noise. In addition to being free of artifacts, the noise we present does not rely on tabulated data (everything is computed on the fly) and it is easy to adjust quality vs. quantity for the noise. The noise is based on point rendering (like spot noise), but it extends to more than two dimensions. The fact that it is based on point rendering makes art direction of the noise much easier.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

Keywords: Noise, point rendering, GPU.

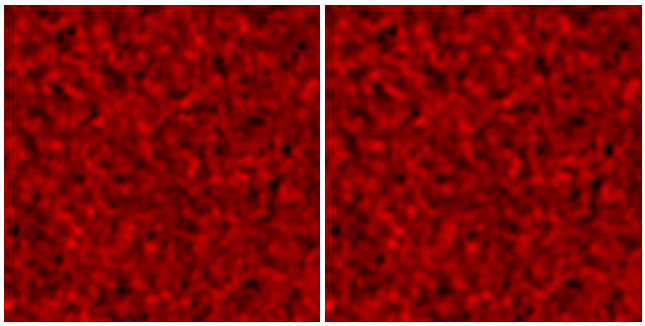
1 Introduction

We present a fast, high-quality noise computation based on rasterization. The noise function we implement is the sparse convolution noise advocated by J. P. Lewis [1984; 1989]. We have chosen sparse convolution noise since it has several qualitative advantages as compared to more commonly used noise functions such as improved Perlin noise. Sparse convolution noise is generally believed to be expensive to compute, but in this paper we challenge that consensus by exploiting the programmability of modern graphics hardware. Our approach has strong relations to van Wijk's [1991] spot noise, but we take the concept to the next level by computing solid noise using three-dimensional spots.

With a few tweaks, the Perlin noise function seems to be the most efficient way of getting individual noise values of decent quality. It is, however, not the only way to go. By our implementation of sparse convolution noise for the GPU, we intend to broaden the range of noise functions available in a graphics programmer's toolbox. The new tool in the box is a way of computing better quality noise without a heavy efficiency penalty.

Why do we need better noise? If you happen to pick a grid-aligned 2D slice from the standard 3D implementation of improved Perlin noise [Perlin 2002], gridline artifacts will be relatively obvious, since the noise function is forced to return zero at every grid node. We might be able to fix this problem, but visually the gradient noise still seems to show a recognizable pattern, especially when grid aligned. This is perhaps due to the fact that improved Perlin noise chooses from only a small set of different gradients at each node.

©ACM, 2007. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Proceedings of GRAPHITE 2007, ACM, pp. 243–248+315, December 2007.



(a) 628.1 s (b) 1.2 s

Figure 1: *Reference noise (and CPU rendering time in seconds). From left to right: (a) Pseudo-white noise filtered with a Gaussian kernel. (b) The same noise filtered with a cubic.*

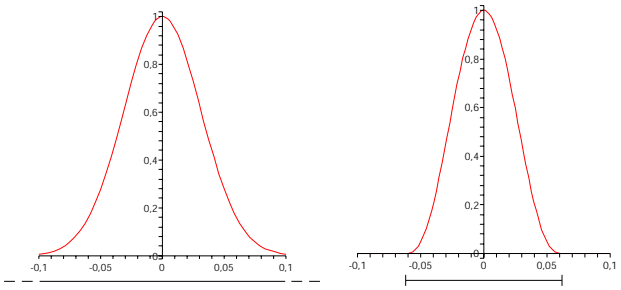


Figure 2: *Filter kernels. From left to right: (a) The Gaussian kernel used in Figure 1a. It never falls off to zero. (b) The cubic kernel used in Figure 1b. It has compact support.*

This problem can also be remedied, but we have already proved our point; it is worth considering different approaches. At the moment Perlin's very fast noise function of decent quality seems to be the de facto standard which everybody uses. There should be an option of at least one better quality noise function which is still fast.

2 Reference Noise

What noise functions in general try to achieve is a good approximation of white noise filtered with some Gaussian kernel [Lewis 1989; Ebert et al. 2002]. But as Perlin points out "this approach would necessitate building a volume of white noise and then blurring it all at once. This is quite impractical" [Ebert et al. 2002, p. 340]. To get a feeling for what good noise looks like, it is nevertheless interesting to compute this reference noise. A sufficient number of pseudo-randomly placed impulses with a pseudo-random value in $[-1, 1]$ gives a good imitation of white noise [Lewis 1989]. Suppose we are looking at a noise image with the size of 512 by 512 units (pixels) and that we use a Gaussian filter with standard deviation $\sigma = 16/3$. This gives approximately 16 Gaussian blobs across the width of the image. The resulting noise image is shown in Figure 1a.

This Gaussian reference noise is quite expensive to compute, since the filter never falls off to zero, see Figure 2a. This means that

every impulse contributes to every pixel. With hardly any impact on quality, we choose a cubic filter instead, see Figure 2b. Let r denote the filter radius ($r = 16$ in this case to get the same number of filter kernels across the image as with the Gaussian) and let d denote the distance from the impulse to the point where we wish to compute the noise value. Then the cubic filter (which has continuous first and second derivatives) replacing the Gaussian is

$$w(d) = \begin{cases} \left(1 - d^2/r^2\right)^3 & , \quad d^2 < r^2 \\ 0 & , \quad d^2 \geq r^2 \end{cases} .$$

Now, for each noise value, we throw away all the impulses outside the filter radius. This greatly improves the efficiency. The result is shown in Figure 1b and is hardly distinguishable from the reference.

To have noise in different frequency ranges, we simply change the filter size r. We refer to the number of filter diameters across the width of a 2D noise image as the scale of the noise. Using this terminology, the noise in Figure 1 is of scale $s = 16$. A good quality measure for the sparse convolution noise is the average number n of sources under each filter kernel. For example $n = 15$ gives noise of decent quality while $n = 30$ gives noise of excellent quality. However, as n increases so does the cost of computing noise values. The two adjustable parameters s and n are convenient to have as input, and from those we find that the number of pseudo-randomly placed sources needed for a D-dimensional noise image is ns^D.

3 Point Rendering Approach

In the reference approach described above, a filter kernel is centered at each pixel, and we need to find the distance to each source in order to know whether it should be included in the noise calculation or rejected. By calculating the noise the other way around, using a rasterization approach, we are able to render each source as a point and obtain exactly the same result.

The advantage of a point rendering approach is that the graphics pipeline (almost for free) finds the pixels influenced by the filter kernel around a source. This is much faster than using some kind of grid or data structure to find the sources under the kernel around a pixel. At this stage the approach is completely analogous to van Wijk's [1991] spot noise.

Each source i has an associated impulse value $v_i \in [-1, 1]$. An average of $n^{1/D}$ sources will influence each pixel and, in this context, a pixel corresponds to a noise value that we wish to compute. The influence of a source on a pixel is a contribution of the value $v_i w(d)$, where d is the distance between the source and the location of the pixel in the noise space. Approximately half of the sources influencing a pixel will have a positive value, the other half will have a negative value. Since we are dealing with pseudo-randomly placed sources of pseudo-random impulse values, it is reasonable to assume that we will never end up with more than the average number of sources contributing one of the extreme values -1 or 1 to the same pixel. Therefore it is also reasonable to assume that the noise function will attain values in the interval $[-n^{1/D}, n^{1/D}]$.

Knowing an approximate interval for the noise values beforehand, enables us to compute noise values by the usual clamped alpha blending. First we split the sources into those with positive values $v_i \in [0, 1]$ and those with negative values $v_i \in [-1, 0)$. We then give each vertex (a vertex corresponds to a source) the color $|v_i|/(2n^{1/D})$. This is done to make sure that the final noise values fit the clamped color buffer. Vertices are rendered as textured point sprites with size $2r$ and an alpha texture given by the cubic filter $w(d)$. Before streaming the vertices to the GPU, we clear all four bands (RGBA) of the color buffer to the value 0.5.

Listing 1: *The fragment shader replacing an alpha texture with precomputed weights $w(d)$.*

```
uniform float filter_size_sqr;
varying vec2 winspace_vert_pos;

float weight(float t) { return t*t*t; }

void main()
{
  vec2 v = gl_FragCoord.xy - winspace_vert_pos;
  float dist_sqr = dot(v, v);
  float w = dist < filter_size_sqr
            ? weight(1.0 - dist_sqr/filter_size_sqr)
            : 0.0;

  gl_FragColor = vec4(gl_Color.rgb, w);
}
```

The blending function is set such that the value

$$\frac{v_i}{(2n^{1/D})} w(d)$$

is added to every pixel covered by point i. To take into account that some sources are positive and some are negative, we do as follows: The vertices corresponding to sources with positive values are rendered with standard additive alpha blending, while the ones with negative values are rendered with reversed subtractive alpha blending.

The range of the resulting noise image is $[0, 1]$ (this interval is obtained subsequently for the reference noises described in the previous section). The d in $w(d)$ may be off by up to half a pixel width because we use a texture. Otherwise we obtain the same result as presented in Figure 1b. Perhaps with lower precision if we do not have a high precision color buffer available, but now we have the result in real-time (0.0083 seconds for the image in Figure 1b) even on old graphics hardware that does not support programmable shaders.

On newer hardware a better option is to render conventional points (instead of textured point sprites) and make a simple fragment program computing the weights $w(d)$. This removes the minor problem that d could be slightly off target in the computation of $w(d)$. The fragment shader is given in Listing 1 and a simple vertex shader to pass the window space position `winspace_vert_pos` of each vertex on to the fragment shader is necessary. It is important to notice that a varying parameter, such as this window space position, is *not* interpolated across the fragments covered by a point. This is always the case in point rendering (except for the texture coordinates in the case of textured point sprites). Therefore it is important that we use points rather than triangles or quads.

4 Introducing a Grid

In the 2D sparse convolution noise, the xy part of the fragment coordinates is used as input for the noise function. This means that sources must be placed either inside or in proximity of the view frustum, otherwise they will have no influence on the noise values we compute. The resulting picture could be referred to as a window into the image of the noise function. If we feel like moving our window to have a look at some arbitrary part of the noise image, we need a quick way of finding the sources to be rendered for this part of the image. Since we could move our window anywhere, it is not a good idea to compute locations and values of the sources in advance.

To compute locations and values of the sources on the fly, we use a linear congruential pseudo-random number generator:

$$x_i = (ax_{i-1} + c) \bmod m \ , \tag{1}$$

where we use $a = 3125$, $c = 49$, m equal to the maximum unsigned integer, and seed $x_0 = 1$. Another way to write this is

$$x_i = \left(a^i x_0 + (a^{i-1} + \ldots + a^0)c\right) \bmod m \tag{2}$$

which means that we are able to find the pseudo-random numbers for some index by implementing a simple function that finds integral powers of unsigned integers. By choosing m to be the largest unsigned integer, we do not need to use the modulo operator. This way of computing pseudo-random numbers was also done for the 4D noise function by Wyvill and Novins [1999].

The exact choice of pseudo-random number generator is not important. However, it needs to be fast and it should take an index as argument. Why an index as argument? Because it makes it easy to impose a grid on the noise domain (the space from which we get arguments for the noise function) and let each grid cell have an index for the generator. Using the index into the generator, we first get one pseudo-random number (2), but with that in hand, each step through the generator (1) finds another pseudo-random number, enabling us to have any number of pseudo-randomly placed sources in each grid cell. This means that no matter where we move the frustum (i.e. the window into the noise image), we are able to look up the sources influencing that particular part of the image by means of grid cell indices and the pseudo-random number generator.

Another good reason to introduce a grid is that it reduces the cost of computing a single noise value. If we use a regular grid and let the width of the grid cells be the same as the diameter of the filter radius, then any noise value will only be influenced by sources in four grid cells in the 2D case (2^D cells for D dimensions). With this construction we easily find the cells influencing the window (the xy-coordinates in the view frustum). A grid cell is rendered as shown in Listing 2. The listed code could be simplified. In the simplest 2D case, we can make do with only $3n$ pseudo-random numbers to render one grid cell, i.e. three numbers for each source in the cell. Two for the position and one for the value. The listed code is slightly more general and finds both a three-dimensional value and position for the sources. This means that the code in Listing 2 is also useful for solid noise generation, and this is the subject of the next section.

Listing 2: *Pseudo-code for drawing the sources in a grid cell.*

n is the number of sources in a grid cell;
$a = 1/(2n^{1/3})$ is the impulse scale;
b is the size of a grid cell;
i is the index of the current grid cell;

seed the pseudo-random number generator by $6ni$;
find $6n$ pseudo-random numbers $v_{jk} \in [0, 1]$,
  where $j = 1, \ldots, n$ and $k = 1, \ldots, 6$;

find the position (x, y, z) of the lower left far corner using i;

draw $n/2$ points using additive alpha blending:
  use $(av_{j1}, av_{j2}, av_{j3})$ as the color of point j;
  use $(x + bv_{j4}, y + bv_{j5}, z + bv_{j6})$ as the position of point j;

draw $n/2$ points using reverse subtractive alpha blending:
  use $(av_{j1}, av_{j2}, av_{j3})$ as the color of point j;
  use $(x + bv_{j4}, y + bv_{j5}, z + bv_{j6})$ as the position of point j;

Listing 3: *Fragment shader for rendering of sources in three dimensions (including interval depth test).*

```
uniform sampler2DRect pos_tex;
uniform float filter_size_sqr;
varying vec3 eyespace_vert_pos;

float weight(float t) { return t*t*t; }

void main()
{
  vec3 position
    = texture2DRect(pos_tex, gl_FragCoord.xy).rgb;

  float depth_test = position.z - eyespace_vert_pos.z;
  if(depth_test*depth_test > filter_size_sqr) discard;

  vec3 v = vec3(position.xy - eyespace_vert_pos.xy,
                depth_test);
  float dist = dot(v, v);
  float w = dist < filter_size_sqr
          ? weight(1.0 - dist/filter_size_sqr) : 0.0;

  gl_FragColor = vec4(gl_Color.rgb, w);
}
```

5 Solid Sparse Convolution Noise

Noise functions taking two-dimensional arguments do not suffice if we, for example, wish to do noise-based procedural solid texturing. To achieve this, we need three dimensional arguments for the noise function. We cannot render all the noise values enclosed in the view frustum in one image as we could in the 2D case where the depth coordinate had no meaning. Instead we use the depth coordinates of some rendered geometry to pick a slice of the solid noise enclosed in the view frustum. That is, we render a model and get the positions on the surface which are seen by the camera as input for the noise function.

A depth texture could be used to pass the depth coordinate information on to the fragment shader. But to get distance calculations right for the filter, we would have to transform the depth coordinates back to eye space. This is not practical. It just necessitates a lot of extra computations in the fragment shader. Instead we render a texture of eye space positions and pass all the xyz information on to the fragment shader. To avoid scaling, which often results in a loss of precision, a floating point texture (e.g. GL_RGB16F_ARB) is employed.

When the arguments for the noise function are ready, the computation is almost the same as in the 2D case. However, now a texture look-up replaces the use of the fragment coordinates and the positions have three coordinates, see Listing 3.

There is one more problem to be considered. The sources needed for the point rendering of the noise must now be three dimensional. Hence, if we use perspective projection, the points change size depending on their distance from the camera. This is taken care of in a vertex program. A simple calculation leads to the following formula finding the width w_P of a point:

$$w_P = \frac{rw}{|z| \tan(\theta_{\text{fov}}/2)} \ , \tag{3}$$

where r is the filter radius, w is the width of the view port, z is the depth in eye space, and θ_{fov} is the field of view. The vertex shader is shown in Listing 4.

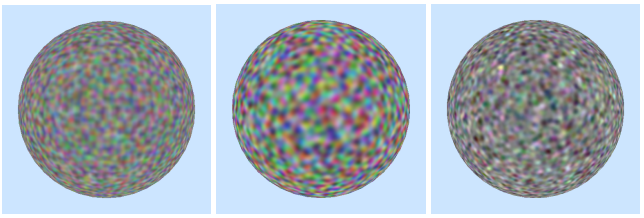


Figure 3: *Visual comparison of noise quality. From left to right: (a) Improved Perlin noise, (b) Perlin's simplex noise, and (c) our implementation of solid sparse convolution noise.*

Listing 4: *Vertex shader for rendering of sources in perspective.*

```
varying vec3 eyespace_vert_pos;

void main()
{
  float tan_fov_2 = 0.624869351909;

  eyespace_vert_pos = (gl_ModelViewMatrix*gl_Vertex).xyz;
  gl_FrontColor = gl_Color;
  gl_Position = ftransform();

  // With glPointSize(rw)
  gl_PointSize
    = gl_Point.size/(abs(eyespace_vert_pos.z)*tan_fov_2);
}
```

If we do not know, in advance, which grid cells that will influence the noise values that we want to compute, we have to render all the grid cells inside and in the proximity of the view frustum. Depending on the size of the view frustum, this could involve rendering of sources in a huge number of grid cells, and none of the fragments influenced by a source are eliminated by the traditional depth test since the fragments must be blended when inside an interval $[z - r, z + r]$ around the depth coordinate.

Since an interval depth test unfortunately is not available, we make our own implementation of it in the fragment shader (Listing 3). And in the next section we propose an additional optimization scheme for finding the indices of the grid cells that influence the noise values we want to compute. It is done at the cost of a small amount of CPU computation, but in return it eliminates the daunting need to cover the entire frustum with sources. Pseudo-code for rendering of solid sparse convolution noise is collected in Listing 5. Both the frame buffer object class by Lefohn et al. and the render texture class by Harris are employed[1]. The first for the simple rendering of eye space positions to a texture, the second for the point rendering to a texture with 16-bit precision alpha blending. The high-precision alpha blending seems to necessitate a second render context which the render texture class provides.

Figure 3 presents a visual comparison of our solid noise to improved Perlin noise [Perlin 2002] and Perlin's simplex noise [Perlin 2001] as implemented by Gustavson [2005]. Our noise does not have the grid line artifacts which are inherent in noises based on values or gradients placed on grid nodes.

6 Optimization

As mentioned in a previous section, if we choose grid cells with a width equal to the filter diameter, only $2^3 = 8$ grid cells can influence a noise value in 3D. Since we use a regular grid, the index

[1] Both classes are available at http://sourceforge.net/projects/gpgpu

Listing 5: *Pseudo-code for rendering of solid sparse convolution noise.*

```
draw world space positions to a texture;

find the IDs of the grid cells influencing the noise values;

switch to an off-screen render context;
    clear color buffer to (0.5, 0.5, 0.5, 0.0);
    enable vertex shader point size control;
    enable blending;
    enable the position texture;

    render the sources in the chosen grid cells:
        use Listing 2 for each cell;

    disable the position texture;
    disable blending
    disable vertex shader point size control;
switch back to standard render context
```

Listing 6: *A fragment shader drawing an ID buffer. The IDs point out grid cells influencing the noise values.*

```
uniform float grid_scale;
uniform float grid_extent;
varying vec3 vert_pos;

void main()
{
  // If we want to repeat the grid,
  // a modulo should be introduced
  vec3 cell_id = floor(vert_pos/grid_scale - 0.5);
  gl_FragColor = vec4(cell_id.zyx/grid_extent, 0.0);
}
```

of the lower left far grid cell is found easily by subtraction of $1/2$ from each coordinate and flooring the coordinates to integers.

If we do an extra pass of the geometry, we are able to find this lower left far grid cell index for all the desired noise values using a fragment shader. The shader is shown in Listing 6.

The only unfortunate thing about this approach is that we have to read back the result to the CPU. This is done relatively efficiently with a pixel buffer object. After reading back the buffer, we sort the indices and remove redundant indices using the `unique` algorithm in the C++ standard library. After this the seven other grid cell indices must be added to the list of grid cells that will be rendered. Again the `sort` and `unique` algorithms must be used to remove redundant indices. Even though this optimization involves some CPU computation, it gives a good speed-up without using any precomputations or assumptions about the input for the noise function. The larger our grid is, the lower is the resolution that we need in this pass. Thus when the scale s is small this optimization is good. We use it for all the rendering times given in this paper (if we render all the sources in the view frustum, rendering times would be close to a second).

Alternatively, occlusion queries could be employed to determine whether a grid cell is close enough to the geometry to have an influence on the noise computation. If this approach is preferred, the advice of Wimmer and Bittner [2005] must be followed to achieve performance comparable to the ID buffer approach described above.

New GPU architectures supporting geometry shaders and integer arithmetics allow further optimizations. The rendering of all the sources in a grid cell is accomplished by submitting only the index of the cell to the GPU. A point for each source in the grid cell is then spawned in a geometry shader and the position of each of the points is determined by the pseudo-random number generator mentioned previously (here the integer arithmetic is needed). On the new GPUs it is also possible to implement the entire noise computation as a `noise` function in a shader. This approach is preferable (and faster) if we need single noise values rather than an entire slice of the noise image. The point rendering scheme only gives a speed up when we need a slice, but in our experience a noise slice is needed more often than separate values.

7 Several Octaves

Many noise-based applications require several octaves of noise. A standard example is the turbulence function. Different noise octaves are easily accomplished by rendering points of varying sizes. In other words, we combine noises of different scales. As an example, consider the turbulence function [Perlin 1985]:

$$\texttt{turbulence}(\boldsymbol{x}) = \sum_{f=f_{\mathrm{lo}}}^{f_{\mathrm{hi}}} \frac{1}{2^f} \left|\mathrm{noise}(2^f \boldsymbol{x})\right| \quad . \tag{4}$$

We evaluate the turbulence function by placing an appropriate number of sources of each scale $s_f = 2^f s_0$ in each grid cell. The value for a source of scale s_f should be scaled by $1/2^f$ as in the function above, and after each octave has been computed a very simple pass is made to take the absolute value. By additively blending each octave, we obtain the turbulence function. The only thing missing is to choose an appropriate number of sources for each scale. As previously, we use n sources for the first octave. When the scale is doubled, the size of a source is scaled by $1/2^D$. Then $n(2^D)^f$ sources are needed to have the noise in octave f at the same quality as the noise in the first octave. This means that the number of sources m to be rendered for each grid cell is

$$m = \sum_{f=f_{\mathrm{lo}}}^{f_{\mathrm{hi}}} n(2^D)^f \quad . \tag{5}$$

Of course, this quickly turns into a huge number of sources. But using the optimizations discussed above, we are again able to cull away most of them. Examples using turbulence are shown in Figure 4.

8 Discussion and Conclusion

Elephant slices of solid sparse convolution noise with $n = 30$ and $s = 20$ are shown in Figure 5. They are computed using the method presented here with a frame rate around 88 on a Pentium 4, 3.5 GHz, with a NVIDIA GeForce 8800 GTX. As the noise is computed in real-time, it is possible for us to have a look at arbitrary elephant slices of the noise image. Of course, we can also use the slice of a noise image to make a bump map, see Figure 4 and 6. This is done by another pass of the geometry in which the noise texture gives a noise value for each position on the surface of the model seen by the camera. Hence, by four texture look-ups we approximate the noise derivatives (using central differences) in the u and v directions on the surface and use these for perturbation of the normal as in traditional bump mapping.

The noise computation presented differs from previous methods (such as Green's [2005] implementation of improved Perlin noise)

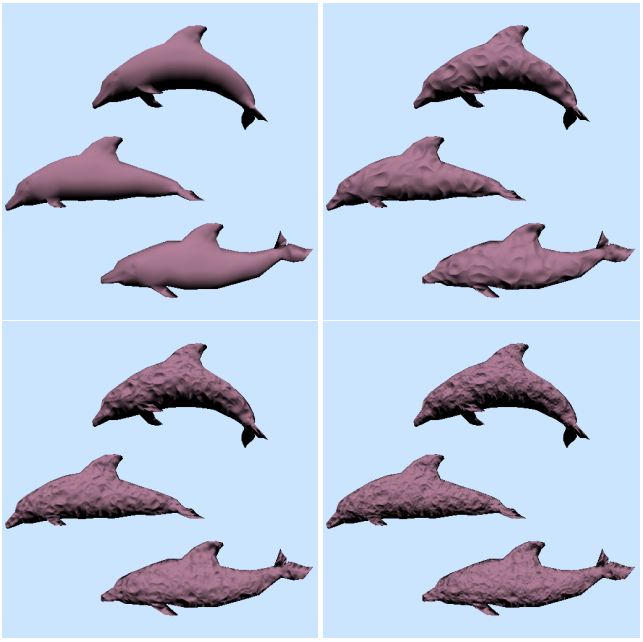


Figure 4: *Examples using the turbulence function. From top left to bottom right the dolphins are rendered using: No noise, one octave of turbulence, two octaves, and three octaves.*

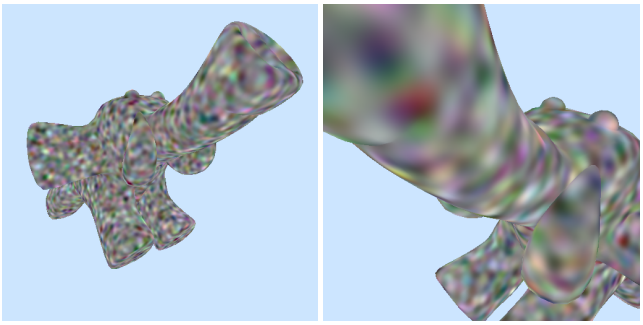


Figure 5: *Elephant slices of the solid sparse convolution noise image (rendering time 0.011 seconds).*

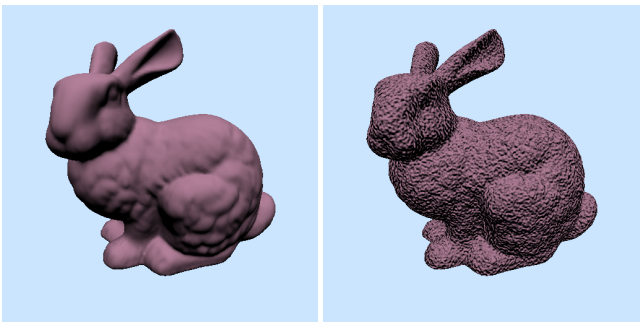
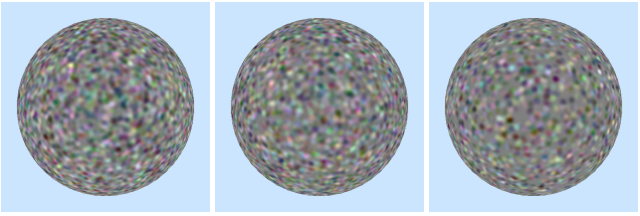


Figure 6: *Procedural bump mapping using solid sparse convolution noise.*



(a) 0.012 s (b) 0.009 s (c) 0.006 s

Figure 7: *Quality/quantity tradeoff. From left to right: Solid sparse convolution noise using (a) 30 sources per grid cell, (b) 20 sources per grid cell, and (c) 10 sources per grid cell.*

because the desired noise values are rendered to a texture. The rendering of the texture is, however, sufficiently fast to allow for real-time changes. In other words, we compute a new slice of the noise image for every frame. For most applications this is just as good as having a noise method which is called in the fragment or vertex shader. The limitation is that xy-coordinates in the position texture with arguments for the noise function must correspond to xy-values in the view frustum. This means that we only have a window into the noise image. The advantage is that everything is computed on the fly. No texture data or look-up table is necessary. If we want to evaluate the noise function for independent arguments, it would correspond to taking a one-by-one picture of the noise image for each evaluation. This is a less efficient option, since the same sources will be evaluated over and over again. It is possible to implement our noise as a function in a shader. This renders the elephant slices in around 50 frames per second (fps) as compared to the 88 fps using the point rendering approach.

The video accompanying this paper illustrates that our way of computing noise works in practice. The video shows a rendering of more than ten fireballs falling into the sea. Everything in the video (both fireballs, water ripples, and reflection distortion) is procedurally generated using our point-based noise computation. The video renders in 20-30 frames per second (frame rate depends on how close the fireballs are to the camera).

We would like to emphasize (as J. P. Lewis did in his analysis of sparse convolution noise [Lewis 1989]) that the efficiency vs. quality tradeoff is easily adjusted by changing the number of sources n in each grid cell (i.e. by making the white noise imitation sparser or denser), see Figure 7. This is a very attractive feature of sparse convolution noise. Moreover sparse convolution noise fulfils all the criteria for good noise which have been described repeatedly [Perlin 1985; Ebert et al. 2002; Green 2005], and which are required of a noise function implementation by the OpenGL Shading Language specification [Kessenich et al. 2003]. With respect to performance, our noise is not faster than Perlin noise. On a "modeler" card (NVIDIA Quadro FX Go 1400) Perlin noise is around five times faster than our noise with $n = 20$, on a "gamer" card (the 8800 GTX used previously) Perlin noise is around ten times faster.

Finally we would like to mention that, as opposed to gradient-based noise (like Perlin noise), it is very easy to do art direction of the noise we present. This nice feature is inherited from van Wijk's [1991] spot noise. We have not explored this feature, but by changing the simple cubic filter kernel to some other procedurally defined kernel or an arbitrary 3D texture constructed by an artist, we are able to modify the overall appearance of the noise. For future work it could be interesting to use the distance field of an arbitrary 3D model (perhaps convolved with a cubic filter) to generate solid noise with an artistic feel that depends on the shape of the model.

Acknowledgement

Thanks to Alexis Angelidis for the elephant model.

References

EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., AND WORLEY, S. 2002. *Texturing and Modeling: A Procedural Approach*, third ed. Computer Graphics and Geometrical Modeling. Morgan Kaufmann Publishers, San Francisco. With contributions from William R. Mark and John C. Hart.

GREEN, S. 2005. Implementing improved Perlin noise. In *GPU Gems 2*, M. Pharr, Ed. Addison Wesley, ch. 26.

GUSTAVSON, S., 2005. Simplex noise demystified. http://staffwww.itn.liu.se/∼stegu/simplexnoise/, March.

KESSENICH, J., BALDWIN, D., AND POST, R. 2003. *The OpenGL® Shading Language*. 3Dlabs, Inc. Ltd., February. Version 1.051.

LEWIS, J. P. 1984. Texture synthesis for digital painting. *Computer Graphics (Proceedings of ACM SIGGRAPH 84) 18*, 3 (July), 245–252.

LEWIS, J. P. 1989. Algorithms for solid noise synthesis. *Computer Graphics (Proceedings of ACM SIGGRAPH 89) 23*, 3 (July), 263–270.

PERLIN, K. 1985. An image synthesizer. *Computer Graphics (Proceedings of ACM SIGGRAPH 85) 19*, 3 (July), 287–296.

PERLIN, K. 2001. Noise hardware. In *Real-Time Shading. ACM SIGGRAPH 2001 Course Notes*, M. Olano, Ed. ACM Press, August.

PERLIN, K. 2002. Improving noise. In *Proceedings of ACM SIGGRAPH 2002*, ACM Press, 681–682.

VAN WIJK, J. J. 1991. Texture synthesis for data visualization. *Computer Graphics (Proceedings of ACM SIGGRAPH 91) 25*, 4 (July), 309–318.

WIMMER, M., AND BITTNER, J. 2005. Hardware occlusion queries made useful. In *GPU Gems 2*, M. Pharr, Ed. Addison Wesley, ch. 6.

WYVILL, G., AND NOVINS, K. 1999. Filtered noise and the fourth dimension. In *ACM SIGGRAPH 99 Conference Abstracts and Applications*, ACM Press, 242.