Contents

Procedural Textures in GLSL

Stefan Gustavson

1.1 Introduction

Procedural textures are textures that are computed on the fly during rendering, as opposed to pre-computed image-based textures. At first glance, computing a texture from scratch for each frame may seem like a stupid idea, but procedural textures have been a staple of software rendering for decades, and for good reasons. With the ever increasing levels of performance for programmable shading in GPU architectures, hardware-accelerated procedural texturing in GLSL is now becoming quite useful, and it deserves more consideration than what is current practice. An example of what can be done is shown in Figure 1.1.

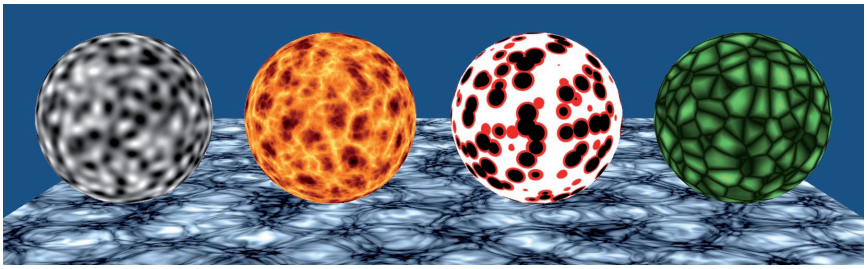

Figure 1.1. Examples of procedural textures. A modern GPU renders this image at full screen resolution in a few milliseconds.

Writing a good procedural shader is more complicated than using image editing software to paint a texture or edit a photographic image to suit our needs, but with procedural shaders, the pattern and the colors can be varied with a simple change of parameters. This allows extensive re-use for many different purposes, as well as fine tuning or even complete overhauls of surface appearance very late in a production process. A procedural pattern allows for analytic derivatives, which makes it less complicated to generate corresponding bump or normal maps and enables analytic anisotropic antialiasing. Procedural patterns require very little storage, and they can be rendered at an arbitrary resolution without jagged edges or blurring, which is particularly useful for rendering of close-up details in real time applications where the viewpoint is often unrestricted. A procedural tex-

ture can be designed to avoid problems with seams and periodic artifacts when applied to a large area, and random-looking detail patterns can be generated automatically instead of having artists paint them. Procedural shading also removes the memory restrictions for 3D textures and animated patterns. 3D procedural textures, *solid textures*, can be applied to objects of any shape without requiring 2D texture coordinates.

While all these advantages have made procedural shading popular for offline rendering, real-time applications have been slow to adopt the practice. One obvious reason is that the GPU is a limited resource, and quality often has to be sacrificed for performance. However, recent developments have given us lots of computing power even on typical consumer level GPUs, and given their massively parallel architectures, memory access is becoming a major bottleneck. A modern GPU has an abundance of texture units and uses caching strategies to reduce the number of accesses to global memory, but many real-time applications now have an imbalance between texture bandwidth and processing bandwidth. ALU instructions can essentially be "free" and cause no slowdown at all when executed in parallel to memory reads, and image-based textures can be augmented with procedural elements. Somewhat surprisingly, procedural texturing is also useful at the opposite end of the performance scale. GPU hardware for mobile devices can incur a considerable penalty for texture download and texture access, and this can sometimes be alleviated by procedural texturing. A procedural shader does not necessarily have to be complex, as demonstrated by some of the examples in this chapter.

Procedural methods are not limited to fragment shading. With the ever increasing complexity of real time geometry and the recent introduction of GPU-hosted tessellation, as discussed in Chapter **??**, tasks like surface displacements and secondary animations are best performed on the GPU. The tight interaction between procedural displacement shaders and procedural surface shaders has proven very fruitful for creating complex and impressive visuals in off-line shading environments, and there is no reason to assume that real-time shading would be fundamentally different in that respect.

This chapter is meant as an introduction to procedural shader programming in GLSL. First, we present some fundamentals of procedural patterns, including anti-aliasing. A significant portion of the chapter presents recently developed, efficient methods for generating Perlin noise and other noise-like patterns entirely on the GPU, along with some benchmarks to demonstrate their performance. The code repository for the book, available from `www.openglinsights.com`, contains a cross-platform demo program and a library of useful GLSL functions for procedural texturing.

1.2 Simple Functions

Procedural textures are a different animal than image-based textures. The concept of designing a function to efficiently compute a value at an arbitrary point without knowledge of any surrounding points takes some getting used to. A good book on the subject, in fact *the* book on the subject, is [Ebert et al. 03]. Its sections on hardware acceleration have become outdated, but the rest is good. Another classic text on software procedural shaders well worth reading is [Apodaca and Gritz 99].

Figure 1.2 presents a varied selection of regular procedural patterns and the GLSL expression that generates them. The examples are monochrome but, of course, black and white could be substituted with any color or texture by using the resulting pattern as the last parameter to the `mix()` function.

For anti-aliasing purposes, a good design choice is to first create a continuous distance function of some sort, and then threshold it to get the features we want. The last three of the patterns in Figure 1.2 follow this advice. None of the examples implement proper anti-aliasing, but we will cover this in a moment.

As an example, consider the circular spots pattern. First, we create a periodic repeat of the texture coordinates by scaling `st` by 5.0 and taking the fractional part of the result. Subtracting 0.5 from this creates cells with 2D coordinates in the range -0.5 to 0.5. The distance to the cell-local origin as computed by `length()` is a continuous function everywhere in the plane, and thresholding it by `smoothstep()` yields circular spots of any desired size.

There is a knack to designing patterns like this from scratch, and it takes practice to do it well, but experimenting is a fun learning experience. However, take warning from the last example in Figure 1.2: writing this kind of functions as one-liners will quickly make them unreadable even to their author. Use intermediate variables with relevant names, and comment all code. One of the advantages of procedural textures is that they can be reused for different purposes, but that point is largely moot if the shader code is impossible to understand. GLSL compilers are reasonably good at simple optimizations like removing temporary variables. Some spoon-feeding of GLSL compilers is still in order to create optimal shader code, but readability does not have to be sacrificed for compactness.

1.3 Anti-Aliasing

Beginners' experiments with procedural patterns often result in patterns that alias terribly, but that problem can be solved. The field of software

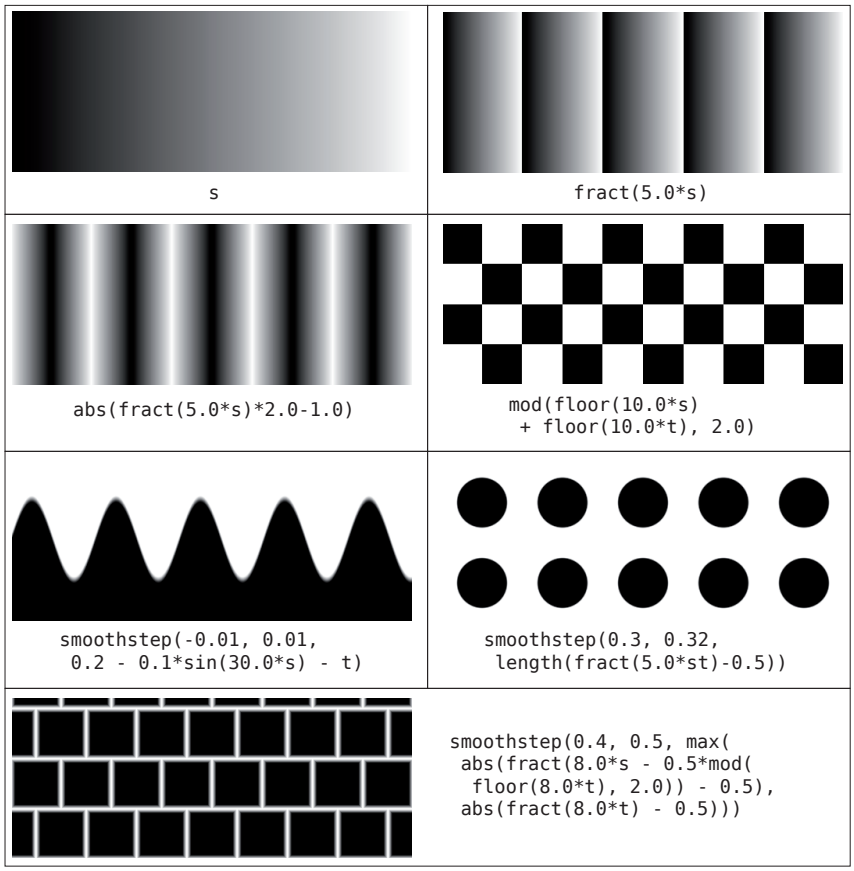

Figure 1.2. Examples of regular procedural patterns. Texture coordinates are either `float s,t` or `vec2 st`, $0 \leq s \leq 1$ and $0 \leq t \leq 0.4$.

shader programming has methods of eliminating or reducing aliasing, and those methods translate directly to hardware shading. Anti-aliasing is even more important for real-time content, because the camera view is often unrestricted and unpredictable. Supersampling can always reduce aliasing, but it is not a suitable routine remedy, because a well written procedural shader can perform its own anti-aliasing with considerably less work than what a brute force supersampling would require.

Many useful patterns can be generated by thresholding a smoothly varying function. For such thresholding, using conditionals (`if-else`) or the all-or-nothing `step()` function will alias badly and should be avoided. In-

stead, use the `smoothstep()` and `mix()` functions to create a blend region between the two extremes, and take care to make the width of the blend region as close as possible to the size of one fragment. To relate shader space (texture coordinates or object coordinates) to fragment space in GLSL, we use the automatic derivative functions `dFdx()` and `dFdy()`. There have been some teething problems with these functions, but now they can be expected to be implemented correctly and efficiently on all GLSL-capable platforms. The local partial derivatives are approximated by differences between neighboring fragments, and they require very little extra effort to compute. See Figure 1.3. The partial derivative functions break the rule that a fragment shader has no access to information from other fragments in the same rendering pass, but it is a very local special case handled behind the scenes by the OpenGL implementation. Mipmapping and anisotropic filtering of image-based textures use this feature as well, and proper anti-aliasing of textures would be near impossible without it.

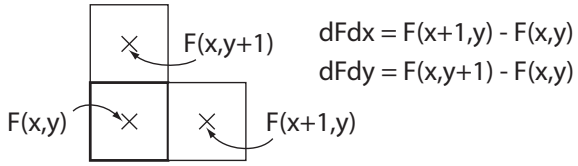

Figure 1.3. "Automatic derivatives" `dFdx()` and `dFdy()` in a fragment shader are simply differences between arbitrary computed values of two neighboring fragments. Derivatives in x and y in one fragment (bold square) are computed using one neighbor each (thin squares). If the right or top neighbors are not part of the same primitive, or for reasons of efficiency, the left or bottom neighbors may be used instead.

For smooth, anisotropic anti-aliasing of a thresholding operation on a smoothly varying function F, we need to compute the length of the gradient vector in fragment space and make the step width of the `smoothstep()` function dependent on it. The gradient in fragment space (x, y) of F is simply $(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y})$. The built-in function `fwidth()` computes the length of that vector as $|\frac{\partial F}{\partial x}| + |\frac{\partial F}{\partial y}|$ in a somewhat misguided attempt to be fast on older hardware. A better choice in most cases nowadays is to compute the true length of the gradient, $\sqrt{\frac{\partial F}{\partial x}^2 + \frac{\partial F}{\partial y}^2}$, according to Listing 1.1. Using ± 0.7 instead of ± 0.5 for the step width compensates for the fact that `smoothstep()` is smooth at its endpoints and has a steeper maximum slope than a linear ramp.

In some cases, the analytical derivative of a function is simple to compute, and it may be inefficient or inaccurate to approximate it using finite

```
// 'threshold' is constant, 'value' is smoothly varying
float aastep(float threshold, float value) {
  float afwidth = 0.7 * length(vec2(dFdx(value), dFdy(value)));
  // GLSL's fwidth(value) is abs(dFdx(value)) + abs(dFdy(value))
  return smoothstep(threshold-afwidth, threshold+afwidth, value);
}
```

Listing 1.1. Anisotropic anti-aliased step function.

differences. The analytical derivative is expressed in 2D or 3D texture coordinate space, but anti-aliasing requires knowledge of the length of the gradient vector in 2D screen space. Listing 1.2 shows how to transform or project vectors in texture coordinate space to fragment coordinate space. Note that we need two to three times as many values from `dFdx()` and `dFdy()` to project an analytical gradient to fragment space compared to computing an approximate gradient directly in fragment space, but automatic derivatives come fairly cheap.

```
  // st is a vec2 of texcoords, G2_st is a vec2 in texcoord space
  mat2 Jacobian2 = mat2(dFdx(st), dFdy(st));
  // G2_xy is G2_st transformed to fragment space
  vec2 G2_xy = Jacobian2 * G2_st;
  // stp is a vec3 of texcoords, G3_stp is a vec3 in texcoord space
  mat2x3 Jacobian3 = mat2x3(dFdx(stp), dFdy(stp));
  // G3_xy is G3_stp projected to fragment space
  vec2 G3_xy = Jacobian3 * G3_stp;
}
```

Listing 1.2. Transforming a vector in (s, t) or (s, t, p) texture space to fragment (x, y) space.

1.4 Perlin Noise

Perlin noise, introduced by Ken Perlin, is a very useful building block of procedural texturing [Perlin 85]. In fact, it revolutionized software rendering of natural-looking surfaces. Some patterns generated using Perlin noise are shown in Figure 1.4, along with the shader code that generates them. By itself, it is not a terribly exciting-looking function – it is just a blurry pattern of blotches within a certain range of sizes. However, noise

can be manipulated in many ways to create impressive visual effects. It can be thresholded and summed to mimic fractal patterns, and it has great potential also for introducing some randomness in an otherwise regular pattern. The natural world is largely built on or from stochastic processes, and manipulations of noise allows a large variety of natural materials and environments to be modeled procedurally.

The examples in Figure 1.4 are static 2D patterns, but some of the more striking uses of noise use 3D texture coordinates and/or time as an extra dimension for the noise function. The code repository for this chapter contains an animated demo displaying the scene in Figure 1.1. The left two spheres and the ground plane are examples of patterns generated by one or more instances of Perlin noise.

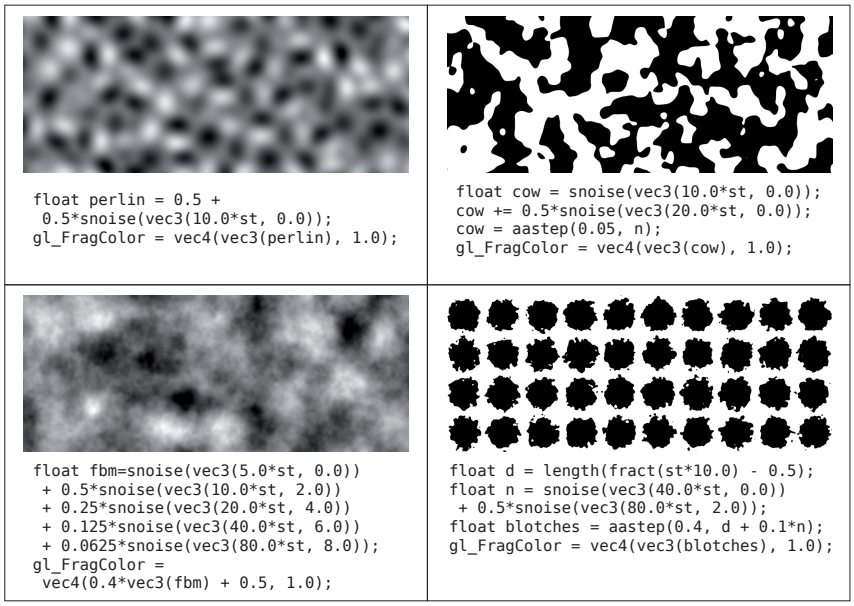

Figure 1.4. Examples of procedural patterns using Perlin noise. Texture coordinates are either `float s,t` or `vec2 st`.

When GLSL was designed, a set of noise functions was included among the built-in functions. Sadly, though, those functions have been left unimplemented in almost every OpenGL implementation to date, except for some obsolete GPUs by 3DLabs. Native hardware support for noise on mainstream GPUs may not appear for a good while yet, or indeed ever, but there are software workarounds. Recent research [McEwan et al. 12] has provided fast GLSL implementations of all common variants of Perlin

noise which are easy to use and compatible with all current GLSL implementations, including OpenGL ES and WebGL. Implementation details are in the article, and a short general presentation of Perlin noise in its classic and modern variants can be found in [Gustavson 05]. Here, we will just present a listing of 2D *simplex noise*, a modern variant of Perlin noise, to show how short it is. Listing 1.3 is a stand-alone implementation of 2D simplex noise ready to cut and paste into a shader – no setup or external resources are needed. The function can be used in vertex shaders and fragment shaders alike. Other variants of Perlin noise are in the code repository for this book.

The different incarnations of Perlin noise are not exactly simple functions, but they can still be evaluated at speeds of several billion fragments per second on a modern GPU. Hardware and software development have now reached a point where Perlin noise is very useful for real-time shading, and we encourage everyone to use it.

1.5 Worley Noise

Another useful function is the *cellular basis function* or *cellular noise* introduced by Steven Worley [Worley 96]. Often referred to as *Worley noise*, this function can be used to generate a different class of patterns than Perlin noise. The function is based on a set of irregularly positioned, but reasonably evenly spaced *feature points*. The basic version of the function returns the distance to the closest one of these feature points from a specified point in 2D or 3D. A more popular version returns the distances to the two closest points, which allows more variation in the pattern design. Worley's original implementation makes commendable efforts to be correct, isotropic, and statistically well-behaved, but simplified variants have been proposed over the years to cut some corners and make the function less cumbersome to compute in a shader. It is still more complicated to compute than Perlin noise, because it requires sorting of a number of candidates to determine which feature point is closest, but while Perlin noise often requires several evaluations to generate an interesting pattern, a single evaluation of Worley noise can be enough. Generally speaking, Worley noise can be just as useful as Perlin noise, but for a different class of problems. Perlin noise is blurry and smooth by default, while Worley noise is inherently spotty and jagged with distinct features.

We have not found any recent publications of Worley noise algorithms for real-time use, but using concepts from our recent Perlin noise work and ideas from previous software implementations, we created original implementations of a few simplified variants and put them in the code repository for this chapter. Detailed notes on the implementation are presented

```
// Description : Array- and textureless GLSL 2D simplex noise.
// Author : Ian McEwan, Ashima Arts. Version: 20110822
// Copyright (C) 2011 Ashima Arts. All rights reserved.
// Distributed under the MIT License. See LICENSE file.
// https://github.com/ashima/webgl-noise

vec3 mod289(vec3 x) { return x - floor(x * (1.0 / 289.0)) * 289.0; }
vec2 mod289(vec2 x) { return x - floor(x * (1.0 / 289.0)) * 289.0; }
vec3 permute(vec3 x) { return mod289(((x*34.0)+1.0)*x); }

float snoise(vec2 v) {
  const vec4 C = vec4(0.211324865405187,  // (3.0-sqrt(3.0))/6.0
                      0.366025403784439,  // 0.5*(sqrt(3.0)-1.0)
                     -0.577350269189626,  // -1.0 + 2.0 * C.x
                      0.024390243902439); // 1.0 / 41.0
  // First corner
  vec2 i = floor(v + dot(v, C.yy) );
  vec2 x0 = v - i + dot(i, C.xx);
  // Other corners
  vec2 i1 = (x0.x > x0.y) ? vec2(1.0, 0.0) : vec2(0.0, 1.0);
  vec4 x12 = x0.xyxy + C.xxzz;
  x12.xy -= i1;
  // Permutations
  i = mod289(i); // Avoid truncation effects in permutation
  vec3 p = permute( permute( i.y + vec3(0.0, i1.y, 1.0 ))
                           + i.x + vec3(0.0, i1.x, 1.0 ));
  vec3 m = max(0.5 - vec3(dot(x0,x0), dot(x12.xy,x12.xy),
                          dot(x12.zw,x12.zw)), 0.0);
  m = m*m; m = m*m;
  // Gradients
  vec3 x = 2.0 * fract(p * C.www) - 1.0;
  vec3 h = abs(x) - 0.5;
  vec3 a0 = x - floor(x + 0.5);
  // Normalise gradients implicitly by scaling m
  m *= 1.79284291400159 - 0.85373472095314 * ( a0*a0 + h*h );
  // Compute final noise value at P
  vec3 g;
  g.x = a0.x * x0.x + h.x * x0.y;
  g.yz = a0.yz * x12.xz + h.yz * x12.yw;
  return 130.0 * dot(m, g);
}
```

Listing 1.3. Complete, self-contained GLSL implementation of Perlin simplex noise in 2D.

in [Gustavson 11]. Here, we just point to their existence and provide them for use. The simplest version is presented in Listing 1.4.

Some patterns generated using Worley noise are shown in Figure 1.5, along with the GLSL expressions that generate them. The right two spheres in Figure 1.1 are examples of patterns generated by a single invocation of Worley noise.

```
// Cellular noise ("Worley noise") in 2D in GLSL, simplified version.
// Copyright (c) Stefan Gustavson 2011-04-19. All rights reserved.
// This code is released under the conditions of the MIT license.
// See LICENSE file for details.

vec4 permute(vec4 x) { return mod((34.0 * x + 1.0) * x, 289.0); }

vec2 cellular2x2(vec2 P) {
const float K  = 1.0/7.0;
const float K2 = 0.5/7.0;
const float jitter = 0.8; // jitter 1.0 makes F1 wrong more often
  vec2 Pi = mod(floor(P), 289.0);
  vec2 Pf = fract(P);
  vec4 Pfx = Pf.x + vec4(-0.5, -1.5, -0.5, -1.5);
  vec4 Pfy = Pf.y + vec4(-0.5, -0.5, -1.5, -1.5);
  vec4 p = permute(Pi.x + vec4(0.0, 1.0, 0.0, 1.0));
  p = permute(p + Pi.y + vec4(0.0, 0.0, 1.0, 1.0));
  vec4 ox = mod(p, 7.0)*K+K2;
  vec4 oy = mod(floor(p*K),7.0)*K+K2;
  vec4 dx = Pfx + jitter*ox;
  vec4 dy = Pfy + jitter*oy;
  vec4 d = dx * dx + dy * dy; // distances squared
  // Cheat and pick only F1 for the return value
  d.xy = min(d.xy, d.zw);
  d.x = min(d.x, d.y);
  return d.xx; // F1 duplicated, F2 not computed
}
varying vec2 st; // Texture coordinates
void main(void) {
  vec2 F = cellular2x2(st);
  float n = 1.0-1.5*F.x;
  gl_FragColor = vec4(n.xxx, 1.0);
}
```

Listing 1.4. Complete, self-contained GLSL implementation of our simplified version of Worley noise in 2D.

1.6 Animation

For procedural patterns, all properties of a fragment are computed anew for each frame, which means that animation comes more or less for free. It is only a matter of supplying the shader with a concept of time through a uniform variable, and to make the pattern dependent on that variable in some manner. Animation speed is independent of frame rate, and animations do not need to loop, but can extend for arbitrary long periods of time without repeating (within the constraints of numerical precision if a floating-point value is used for timing). Animation literally adds a new dimension to patterns, and the unrestricted animation that is possible with procedural textures is a strong argument for using them. Perlin noise is available in a 4D version, and its main use is to create textures where

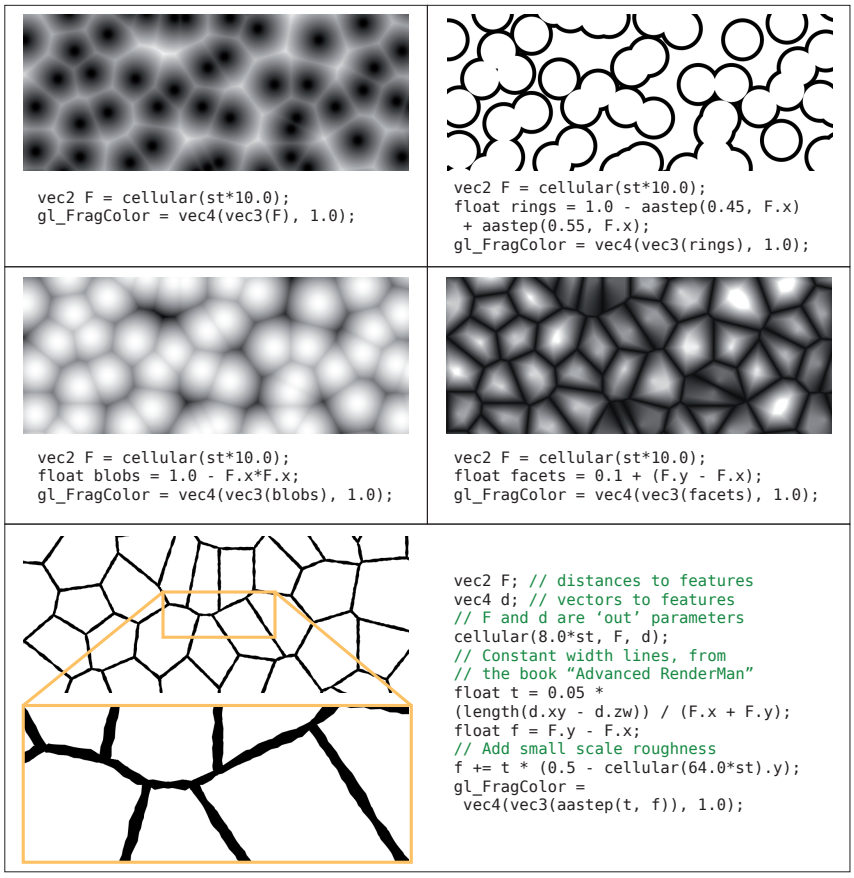

Figure 1.5. Examples of procedural patterns using Worley noise. Texture coordinates are `vec2 st`. For implementations of the `cellular()` functions, see the code repository.

3D spatial coordinates and time together provide the texture coordinates for an animated solid texture. The demo code that renders the scene in Figure 1.1 animates the shaders simply by supplying the current time as a uniform variable to GLSL and computing patterns that depend on it.

Unlike pre-rendered image sequences, procedural shader animation is not restricted to simple, linear time dependencies. View-dependent changes to a procedural texture can be used to affect the level of detail for the rendering, so that for example bump maps or small scale features are computed only in close-up views to save GPU resources. Procedural shading allows

arbitrary interactive and dynamic changes to a surface, including extremely complex computations like smoke and fluid simulations performed on the GPU. Animated shaders have been used in software rendering for a long time, but interactivity is unique to real-time shading, and a modern GPU has considerably more computing power than a CPU. There are many fun and wonderful things left to explore here.

1.7 Texture Images

Procedural texturing is all about removing the dependency on image based textures, but there are applications where a hybrid approach is useful. A texture image can be used for coarse detail to allow better artistic control, and a procedural pattern can fill in the details in close-up views. This includes not only surface properties in fragment shaders, but also displacement maps in vertex shaders. Texture images can also be used as data for further processing into a procedural pattern, like in the manner presented in Chapter **??**, or like in the halftoning example in Figure 1.6, rendered by the shader in Listing 1.5. The bilinear texture interpolation is performed explicitly in shader code. Hardware texture interpolation often has a limited fixed-point precision which is unsuitable for this kind of thresholding under extreme magnifications.

Of course, some procedural patterns that are too cumbersome to compute for each frame can be rendered to a texture and re-used between frames. This approach maintains several of the advantages with using procedural patterns (flexibility, compactness, dynamic resolution), and it can be a good compromise while we are waiting for complex procedural texturing to be easily manageable in true real-time. Some of the advantages are lost (memory bandwidth, analytic anisotropic anti-aliasing, rapid animations), but it does solve the problem of extreme minification. Minification can be tricky to handle analytically, but is solved well by mipmapping of an image-based texture.

1.8 Performance

Shader-capable hardware comes in many variations. An older laptop GPU or a low cost, low power mobile GPU can typically run the same shader as a brand new high end GPU for gaming entusiasts, but their raw performance might differ by as much as 100 times. The usefulness of a certain procedural approach is therefore highly dependent on the application. GPUs get faster all the time, and their internal architectures change between releases, sometimes radically so. For this reason, absolute benchmarking is a rather

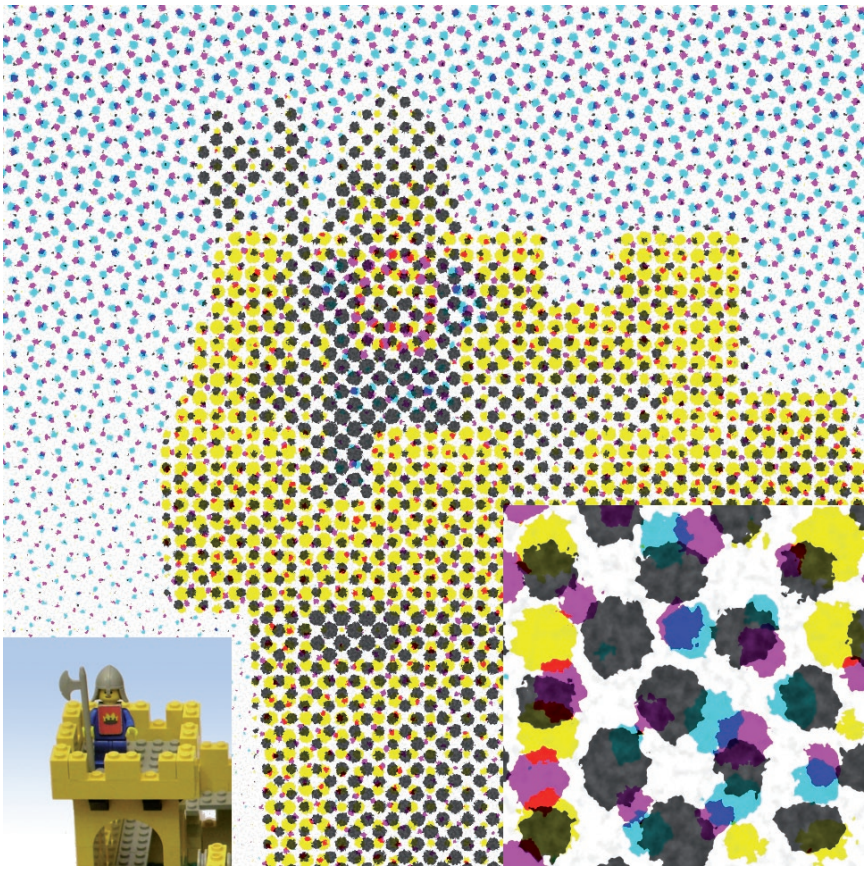

Figure 1.6. A halftone shader using a texture image as input. The shader is listed in Listing 1.5. Small random details become visible in close-up views (inset, lower right). For distance views, the shader avoids aliasing by gradually blending out the halftone pattern and blending in the plain RGB image (inset, lower left).

futile exercise in a general presentation such as this one. Instead, we have measured the performance of a few of the example shaders from this chapter on a selection of hardware. The results are summarized in Table 1.1. The list should not be considered a representative or carefully picked selection – it is just a few random GPUs of different models, neither top performing nor particularly new, and some of the shaders we have presented in this chapter. The program to run this benchmark is included in the code repository. The absolute figures depend on operating system and driver version and should only be taken as a general indication of performance.

The most useful information in the table is the relative performance within one column: it is instructive to compare a constant color shader or a single texture lookup with various procedural shaders on the same GPU. As is apparent from the benchmarks, it is very hard to beat a single texture lookup for raw speed, not least because most current GPUs are specifically designed to have a high texture bandwidth. However, reasonably complex procedural textures can run at perfectly useful speeds, and they become more competitive when the limiting factor for GPU performance is memory bandwidth. Procedural methods can execute in parallel to memory reads, and add to the visual complexity of a textured surface without necessarily slowing things down. For the foreseeable future, GPUs will continue to have a problem with memory bandwidth, and their computational power will keep increasing. There is certainly lots of room to experiment here.

Shader	NVIDIA 9600M	AMD HD6310	AMD HD4850	NVIDIA GTX260
Constant color	422	430	2,721	3,610
Single texture	412	414	2,718	3,610
Dots (Fig 1.2, lower right)	360	355	2,720	3,420
Perlin noise (Fig 1.4, top left)	63	97	1,042	697
5x Perlin (Fig 1.4, bottom left)	11	23	271	146
Worley noise (Fig 1.5, top left)	82	116	1,192	787
Worley tiles (Fig 1.5, bottom)	26	51	580	345
Halftone (Fig 1.6)	34	52	597	373

Table 1.1. Benchmarks for a few example shaders. Numbers are in millions of fragments per second. NVIDIA 9600M is an old laptop GPU, AMD HD6310 is a budget laptop GPU. AMD HD4850 and NVIDIA GTX260 were mid-range desktop GPUs in 2011. High-end GPUs of 2011 perform several times better.

1.9 Conclusion

The aim of this chapter was to demonstrate that modern shader-capable GPUs are mature enough to render procedural patterns at fully interactive speeds, and that GLSL is a good language to write procedural shaders very similar to the ones that have become standard tools in off-line rendering over the past two decades. In a content production process that includes procedural textures, some of the visuals need to be created using math and a programming language as tools for creative visual expression, and this requires a slightly different kind of talent than what it takes to be a good visual artist with traditional image editing tools. Also, the GPU is still

a limited resource, and care needs to be taken not to overwhelm it with overly complex shaders. Procedural texturing is not yet a wise choice in every situation. However, there are situations where a procedural pattern simply does the job better than a traditional, image-based texture, and the tools and the required processing power are now available to do it in real-time. Now is a good time to start writing procedural shaders in GLSL.

Bibliography

[Apodaca and Gritz 99] Anthony Apodaca and Larry Gritz. *Advanced RenderMan: Creating GCI for Motion Pictures*. Morgan Kaufmann, 1999.

[Ebert et al. 03] David Ebert, Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steve Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann, 2003.

[Gustavson 05] Stefan Gustavson. "Simplex Noise Demystified." http://www.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf, March 22, 2005.

[Gustavson 11] Stefan Gustavson. "Cellular Noise in GLSL: Implementation Notes." http://www.itn.liu.se/~stegu/GLSL-cellular/GLSL-cellular-notes.pdf, April 19, 2011.

[McEwan et al. 12] Ian McEwan, David Sheets, Stefan Gustavson, and Mark Richardson. "Efficient computational noise in GLSL." *Journal of Graphics, GPU and Game Tools* 16:1 (2012), (to appear).

[Perlin 85] Ken Perlin. "An Image Synthesizer." *Proceedings of ACM Siggraph 85* 19:3 (1985), 287–296.

[Worley 96] Steven Worley. "A Cellular Texture Basis Function." In *SIGGRAPH '96 Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 291–293, 1996.

```
uniform sampler2D teximage;
uniform vec2 dims; // Texture dimensions (width and height)
varying vec2 one;  // 1.0/dims from vertex shader
varying vec2 st;   // 2D texture coordinates

// Explicit bilinear lookup to circumvent imprecise interpolation.
// In GLSL 1.30 and above, 'dims' can be fetched by textureSize().
vec4 texture2D_bilinear(sampler2D tex, vec2 st, vec2 dims, vec2 one) {
  vec2 uv = st * dims;
  vec2 uv00 = floor(uv - vec2(0.5)); // Lower left of lower left texel
  vec2 uvlerp = uv - uv00 - vec2(0.5); // Texel-local blends [0,1]
  vec2 st00 = (uv00 + vec2(0.5)) * one;
  vec4 texel00 = texture2D(tex, st00);
  vec4 texel10 = texture2D(tex, st00 + vec2(one.x, 0.0));
  vec4 texel01 = texture2D(tex, st00 + vec2(0.0, one.y));
  vec4 texel11 = texture2D(tex, st00 + one);
  vec4 texel0 = mix(texel00, texel01, uvlerp.y);
  vec4 texel1 = mix(texel10, texel11, uvlerp.y);
  return mix(texel0, texel1, uvlerp.x);
}

void main(void) {
  vec3 rgb = texture2D_bilinear(teximage, st, dims, one).rgb;
  float n = 0.1*snoise(st*200.0);
  n += 0.05*snoise(st*400.0);
  n += 0.025*snoise(st*800.0); // Fractal noise, 3 octaves
  vec4 cmyk;
  cmyk.xyz = 1.0 - rgb; // Rough CMY conversion
  cmyk.w = min(cmyk.x, min(cmyk.y, cmyk.z)); // Create K
  cmyk.xyz -= cmyk.w; // Subtract K amount from CMY

  // CMYK halftone screens, in angles 15/-15/0/45 degrees
  vec2 Cuv = 50.0*mat2(0.966, -0.259, 0.259, 0.966)*st;
  Cuv = fract(Cuv) - 0.5;
  float c = aastep(0.0, sqrt(cmyk.x) - 2.0*length(Cuv) + n);
  vec2 Muv = 50.0*mat2(0.966, 0.259, -0.259, 0.966)*st;
  Muv = fract(Muv) - 0.5;
  float m = aastep(0.0, sqrt(cmyk.y) - 2.0*length(Muv) + n);
  vec2 Yuv = 50.0*st; // 0 deg
  Yuv = fract(Yuv) - 0.5;
  float y = aastep(0.0, sqrt(cmyk.z)-2.0*length(Yuv)+n);
  vec2 Kuv = 50.0*mat2(0.707, -0.707, 0.707, 0.707)*st;
  Kuv = fract(Kuv) - 0.5;
  float k = aastep(0.0, sqrt(cmyk.w) - 2.0*length(Kuv) + n);

  vec3 rgbscreen = 1.0 - vec3(c, m, y);
  rgbscreen = mix(rgbscreen, vec3(0.0), 0.7*k + 0.5*n);
  vec2 fw = fwidth(st);
  float blend = smoothstep(0.7, 1.4, 200.0*max(fw.s, fw.t));
  gl_FragColor = vec4(mix(rgbscreen, rgb, blend), 1.0);
}
```

Listing 1.5. The fragment shader to generate the halftone pattern in Figure 1.6.

Index