SIGRAD 2013, pp. 41-46
T. Ropinski and J. Unger (Editors)

No more texels, no more facets: Emerging trends in GPU procedural shading

Stefan Gustavson, Linköping University, ITN
stefan.gustavson@liu.se

Abstract

Procedural textures have long been a staple of off-line rendering, and impressive creative results have been accomplished by using procedural methods to their advantage. As GPU speeds and computational capabilities continue to increase, procedural texturing will likely become a useful tool also for real time rendering. In fact, it is already possible to generate procedural patterns of considerable complexity at real time frame rates on a modern GPU. Even on current (2013) low-end and mobile GPUs, the programmable shading system offers considerable processing power that often remains largely unused. Such untapped resources could be used for procedural patterns and procedural geometry computed entirely on the GPU.

This article presents the state of the art in the field. Most of this is yet to be implemented in commercial projects like games, VR and visualization applications. Code for the shader examples in this article is available under permissive open source licenses or as public domain software and is collected on the address:
`http://www.itn.liu.se/~stegu76/sigrad13`

1 History of procedural shading

The concept of a *surface shader*, a small program to compute the color of a surface point using the view direction, surface normal, material properties and information on the scene illumination, has been around for a large portion of the history of computer graphics. The concept was formalized in the commercial product Photorealistic RenderMan from Pixar, first released to the public in 1989 [11, 8]. While the RenderMan Interface was made famous by its software implementations, the original intent was to make a hardware renderer, and a prototype device named RM-1 was manufactured and sold by Pixar for a few years in the 1980's. The Academy award winning Pixar short film "Luxo Jr." was rendered on that custom hardware.

After Pixar abandoned their hardware renderer project, hardware graphics rendering took off in a different direction with Silicon Graphics' IrisGL and OpenGL, where real time performance took priority over both image quality and flexibility. For a long time there was a harsh disconnect in terms of image quality between real time graphics, where a frame needs to be rendered in fractions of a second, and cinematic pre-rendered graphics, where rendering times are allowed to extend to several hours per frame.

2 GPU procedural shading

Graphics hardware has now come to a point where the image quality can match off-line rendering from about a decade ago. Around 2004, programmable shading was introduced into the hardware rendering pipeline of mainstream GPUs, which added considerable flexibility. The line between real time graphics and off-line rendered graphics is blurring, and GPU rendering is now successfully used not only for real time rendering, but also for acceleration of off-line rendering. In a way, we have now come full circle from Pixar's RM-1 hardware in the 1980's through a long tradition of software rendering, and back to a focus on hardware.

Instead of traditional fixed-function lights and materials, with a predetermined, rigid computational structure and a small set of parameters to control both the surface appearance and the illumination, programmable shading has finally set real time graphics free of the simplistic illumination and surface reflection models from the 1980's, and made it possible to explore more modern concepts for real time lighting, shading and texturing.

To date, this new-found freedom has been used mainly to explore better illumination and reflection models, which was indeed an area in great need of im-

provement. However, another fascinating area remains largely unexplored: procedural patterns. A good review and a long standing reference work in the field is [1]. This article is an attempt to point to the many fascinating possibilities in that area for real time applications.

3 Procedural patterns

A procedural pattern is a function over spatial coordinates that for each point (u, v) on a surface describes its color, transparency, normal direction or other surface property as $f(u, v)$. This function is defined over a continuous domain (u, v), not a discrete set of sample points, and contrary to a texture image it is computed directly for each surface point for each frame, rather than sampled and stored for repeated use. While it might seem wasteful and contrary to common sense in computing to throw away previous hard work and compute each point anew for each surface point, it offers many clear advantages:

- No memory is required, and no memory accesses are made. This saves bandwidth on the often congested memory buses of today's massively parallel GPU architectures.
- Patterns can be computed at arbitrary resolution.
- Patterns can be changed by simple parameters.
- Animated patterns come at no extra cost.
- 3D and 4D functions can be used for texturing, removing the need for 2D texture coordinates.
- The shader program can perform its own analytic anti-aliasing.

Of course, there are also disadvantages:

- Not all patterns can be described by simple functions.
- Creating a good procedural pattern is hard work.
- The pattern must be programmed, which takes a very different skill than painting it or editing a digital photo. Such skill is hard to find.
- Production pipelines for real time content are firmly set in their ways of using texture images as assets.
- Current GPU hardware is designed to provide huge bandwidth for texture images, making it somewhat counter-productive not to use them.

Procedural patterns are not a universal tool, and they are not going to replace sampled texture images altogether. However, many animation and special effects studios have seen an advantage in using procedural patterns for many tasks, not least because of the ability to easily tweak and change the appearance of any surface in the scene late in the production process.

A modern GPU has a massively parallel architecture with lots of distributed computing power, but with thousands of computing units sharing a common memory of limited bandwidth. To counter the imbalance, local memory is used to implement cache strategies, but fact remains that memory-less procedural texturing is becoming ever more attractive for real time use, at least as a complement to traditional texturing. In a texture intensive shader, the execution speed is often limited by memory access bandwidth, and the arithmetic capabilites of the GPU processing units are often not fully utilized. Sometimes the computing power to generate a procedural pattern is available, but remains unused. Moreover, computations in local registers may be more power efficient than accesses to global memory.

4 Simple patterns

Some patterns are very suitable for description as functions. Stripes, spots, tiles and other periodic functions can be described by a combination of modulo functions and thresholding operations on the surface parameters (u, v) or directly on the object coordinates (x, y, z). Some examples with GLSL code are shown in Figure 1.

5 Noise

The world around us is not clean and perfect. The natural world is largely a result of stochastic processes, the same sort of random variation is seen in manufactured objects due to process variations, dirt and wear. Recognizing this, Ken Perlin created *Perlin noise* in the 1980's [10] as a tool for adding complexity and variation to surfaces and models. Variations on his function have seen heavy use ever since in most off-line rendering. Perlin noise has several nice and useful properties:

- It is fairly easy to compute.
- It is band limited and reasonably isotropic, which makes it suitable for spectral synthesis of more complex patterns.
- It has a well defined derivative everywhere.

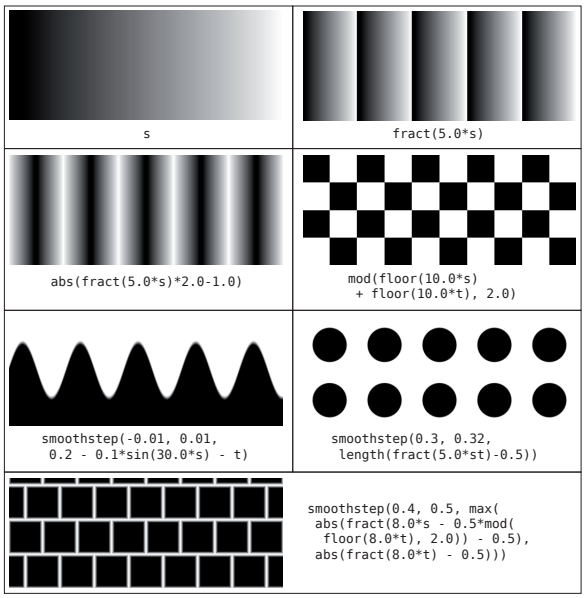

Figure 1: Some simple procedural patterns in GLSL.

Noise by itself does not make a very interesting pattern, although it can be used to generate waves and bumps. More interesting patterns are generated by performing spectral synthesis (combining noise of several spatial frequencies), and by using color tables or thresholding the result. Noise really shines when it is used to add variation to other, more regular patterns. Some patterns based on Perlin noise are shown in Figure 2.

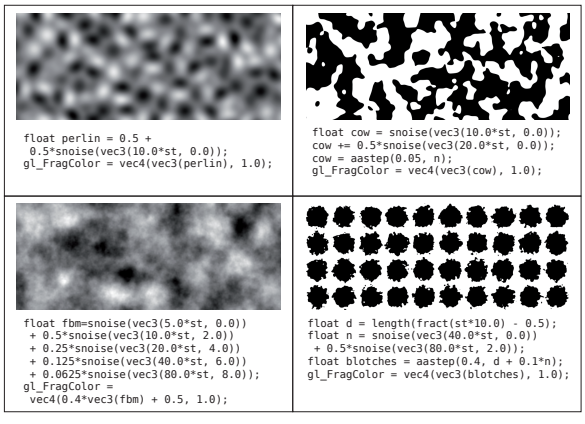

Figure 2: Some Perlin noise patterns in GLSL.

Another useful and popular pseudo-random pattern function is cellular noise, introduced by Stephen Worley [12]. It is a different kind of function than Perlin noise, and it generates a different class of patterns: tiles, cells, spots or other distinct features distributed across a surface with a seemingly random placement. Some patterns based on cellular noise are shown in Figure 3.

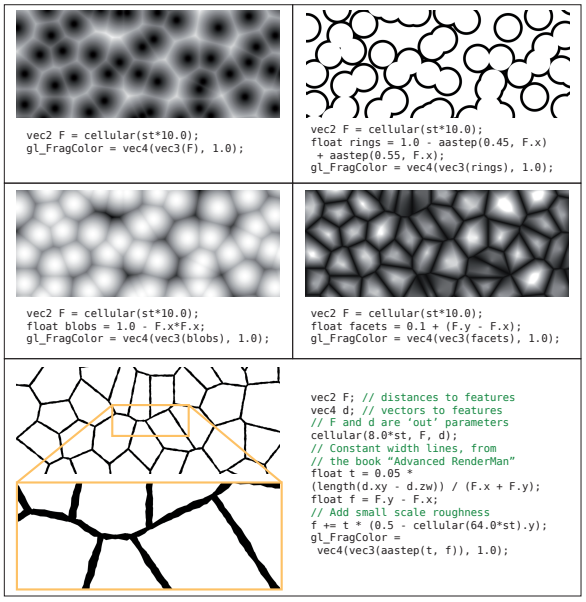

Figure 3: Some simple procedural patterns in GLSL.

Noise is ubiquitous in off-line rendering, in particular when rendering scenes from nature, but it has not been a good fit for real time rendering because of its relative complexity. Noise is not a very complicated function, but a typical noise-based texture needs several noise values per surface point, and each value takes more work to compute than a simple texture image lookup. When GLSL was designed, noise functions were included in the language, but to date (2013) they remain unimplemented. The desire to implement hardware support for Perlin noise has to compete for silicon area with other design goals of a modern GPU. Recent research has provided some very hardware friendly noise functions, and it might be time to reconsider. However, what GPU manufacturers choose to implement is largely beyond our control. In the meantime, we can get by with shader-implemented software versions of noise. Recent advances in GPU capabilites in combination with new variations on the kind of noise functions introduced by Perlin and Worley has provided hardware-friendly noise functions implemented in GLSL [9, 6]. The functions are licensed with a permissive MIT li-

cense, they are very easy to use and show good real time performance on modern GPUs. A visual example of the kind of patterns that can be generated with these functions, at fully interactive frame rates even on a mid-range GPU, is in Figure 5.

6 General contours

Another problem with traditional image-based texturing is crisp edges. Formally, an edge is not a feature that is suitable for sampling, because it is not band limited. Other representations have been proposed over the years to render edges and contours on 2D surfaces, but none have prevailed. Recently, NVIDIA presented an extension to OpenGL named `NV_path_rendering`, which is their take on rendering 2D graphics directly from contour descriptions of the kind that are used in PostScript, PDF, SVG and other modern standards for 2D object graphics. While successful, it taxes even a top performing GPU considerably to render contours in this manner, and it is not useful on lower end hardware or hardware from other manufacturers.

Another method, providing a significant new take on existing ideas from 3D shape rendering [3], was proposed in 2007 by Chris Green of Valve Software [4]: a distance transform representation of the contour is computed and stored as a texture, and a shader is used to generate the crisp edge. This allows a fully general, anisotropic analytic anti-aliasing of the edge, and the shader can be kept simple and fast. Combined with recent development in distance transforms [7], this method is very suitable for certain important kinds of surface textures, like alpha-masked outlines, text, decals and printed patterns. Until now, real time graphics has had an annoying tendency of making such patterns blurry or pixelated in close-up views, but contour rendering based on a distance transform can eliminate that and make edges crisp and clear in a very wide range of scales from minification to extreme magnifications. Moreover, the edges can be anti-aliased in an anisotropic, analytic manner. An example of this is presented in [5]. A pattern processed and rendered in that manner is shown in Figure 4.

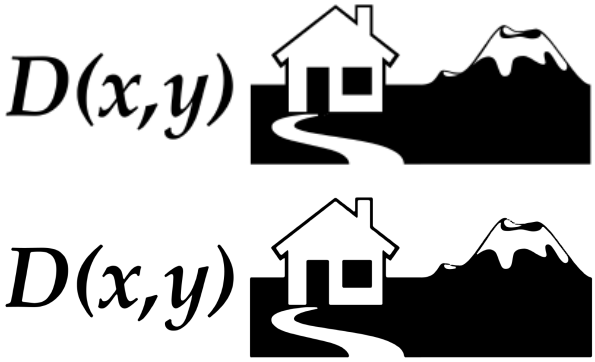

Figure 4: Shapes rendered by a distance transform. *Top:* bitmap texture used as input to the distance transform. *Bottom:* crisp, high resolution shapes rendered by the distance transform and a shader.

7 Anti-aliasing

The texture subsystem of a GPU has built-in mechanisms to handle anti-aliasing. Mipmapping has been part of OpenGL since its early days, and anisotropic mipmap filtering has been common for years. However, mipmapping only handles minification, and there is a limit to how much anisotropy it can handle. Procedural patterns can also fill in detail when the pattern is magnified. Edges remain crisp, and there is no pixelization. Finally, because the anti-aliasing is analytic in nature, a procedural surface pattern can be properly anti-aliased even in very oblique views with strong anisotropy in an arbitrary orientation. Anti-aliasing of procedural patterns is a problem that requires extra care in designing the shader, but it's not magic. Producers of off-line rendered graphics have a long tradition of anti-aliasing in shader programming, and the methods and concepts are directly applicable for real time use as well.

8 Displacement shaders

Procedural shaders in off-line production can not only set the properties of a surface point. Another important class of shaders is *displacement shaders*, which set the position of the surface point in relation to some original surface. Displacement mapping is the tangible concept that is simulated by bump mapping, normal mapping or parallax mapping. By performing a true displacement of the surface, a displacement shader can act as a modeling tool and add true geometric detail to a surface. Because the displacement shader is aware of the viewpoint and the view direction, it can adapt the level of detail to what is really needed, thus striking an important balance between performance and quality.

A powerful tool in off-line production is the combination of a displacement shader and a matching surface shader. By passing information about the position and orientation of the displaced surface from the displacement shader to the surface shader, very detailed and realistic surfaces can be created. Displacement shaders have been an important part of the toolbox for off-line production for a long time.

When GPU shading was introduced, vertex shaders did not allow for displacement of arbitrary surface points, only vertices. Thereby an important part of what displacement shaders can do was lost. To simulate a point by point displacement, dynamic tessellation to an appropriate resolution had to be performed on the CPU. While this has been done for things like real time terrain rendering, it is a process that is taxing for the CPU, and the tessellation needs to stop before the triangles become too small and too many for the geometry transfer from the CPU to the GPU to handle them at real time frame rates.

The recent introduction of hardware tessellation shaders changed that. Now, the GPU can execute a shader that dices the geometry into very small pieces in a view dependent manner, without even bothering the CPU. The GPU has a high internal bandwidth and can do this for many triangles in parallel. We have seen this being put to good use for adaptive tessellation of curved surfaces, finally making it possible to use bicubic patches, NURBS and subdivision surfaces as the native model description for real time content. This is a welcome thing by itself, as real time graphics has been somewhat hampered by the fixation on polygon modelling, while content for off-line rendering has been free to also use more flexible and adaptively tessellated curved surface primitives [2].

However, tessellation shaders can be used for more. They can bridge the gap between the comparably crude vertex shaders for hardware rendering and the much more versatile displacement shaders used in software rendering. GPUs are not quite yet at the point where they can easily dice all geometry in the scene to triangles the size of a single pixel, but we are rapidly getting there, and then the vertex shader stage will have the same capabilities as a traditional displacement shader for off-line rendering. The REYES rendering algorithm, which is the traditional scanline rendering method for Pixar's RenderMan, has a very hardware friendly structure because it originated as an algorithm for Pixar's hardware renderer RM-1. Simply put, it works by dicing all geometry to micropolygons that are about the projected size of a pixel, and executing shaders for each micropolygon. The time is now right to start doing similar things in GPU rendering.

9 Conclusion

We are used to dealing with real time graphics and off-line graphics in different ways, using different methods and different tricks to render the images we want. We have also been forced to accept a low image quality from real time graphics, incapable of matching even previous generations of off-line renderings. That gap is now closing. While some methods from software rendering, like the more advanced methods for global illumination, will probably remain out of reach for real time execution for quite some time longer, there are definitely possibilities to mimic in a real time setting what software renderers like RenderMan did ten years ago.

Current research and development in GPU rendering seems to be focusing on adding even more sophisticated illumination and surface reflectance models to the hardware rendering pipeline. The two concepts emphasized in this article, procedural patterns and displacement shaders, do not seem to attract an interest nearly as strong. That is a shame, as there are many fun and useful things to be done in that area. Looking further into this could have a significant impact on the overall perceived realism in real time graphics, possibly more so than current incremental improvements in illumination and reflectance. Real time illumination and surface reflectance models are now roughly at the point where off-line scanline rendered graphics was ten years ago. Geometry modelling and surface texturing, however, are still twenty years behind or more. To put it bluntly, real time graphics is basically still pasting pixels onto polyhedra like we did in the early 1990's, only at a much greater speed.

Imagine if in real time graphics applications we could eliminate pixelated textures in close-up views, add arbitrary surface variation programmatically to any texture, model geometry features down to a microscopic scale using displacement shaders, and stop worrying about polygon edges showing.

No more texels, no more facets!

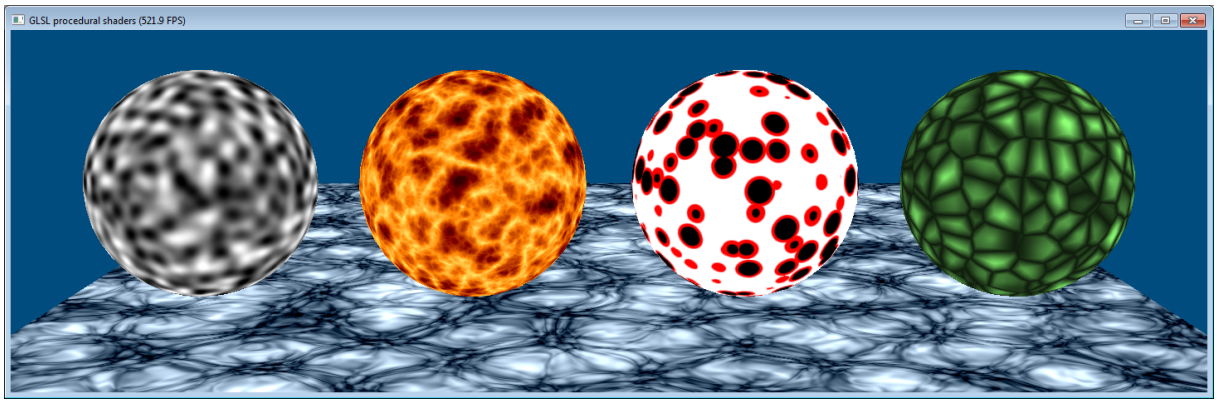

Figure 5: Some more complex noise-based procedural patterns in GLSL

References

[1] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2002.

[2] A. R. Fernandes and B. Oliveira. Gpu tessellation: We still have a LOD of terrain to cover. In P. Cozzi and C. Riccio, editors, *OpenGL Insights*, pages 145–162. CRC Press, 2012.

[3] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones. Adaptively sampled distance fields: a general representation of shape for computer graphics. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 249–254, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[4] C. Green. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, pages 9–18, New York, NY, USA, 2007. ACM.

[5] S. Gustavson. 2D shape rendering by distance fields. In P. Cozzi and C. Riccio, editors, *OpenGL Insights*, pages 173–181. CRC Press, 2012.

[6] S. Gustavson. Procedural textures in GLSL. In P. Cozzi and C. Riccio, editors, *OpenGL Insights*, pages 105–120. CRC Press, 2012.

[7] S. Gustavson and R. Strand. Anti-aliased euclidean distance transform. *Pattern Recogn. Lett.*, 32(2):252–257, Jan. 2011.

[8] P. Hanrahan and J. Lawson. A language for shading and lighting calculations. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '90, pages 289–298, New York, NY, USA, 1990. ACM.

[9] I. McEwan, D. Sheets, M. Richardson, and S. Gustavson. Efficient computational noise in GLSL. *Journal of Graphics Tools*, 16(2):85–94, 2012.

[10] K. Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, July 1985.

[11] S. Upstill. *RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[12] S. Worley. A cellular texture basis function. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 291–294, New York, NY, USA, 1996. ACM.