Noise is Beautiful

Part 1: Procedural textures

by Stefan Gustavson

Draft version 11/18/24, 23:09:45

This is a draft version, but if you find errors, omissions,
bad explanations or figures that could be made more
clear, feel free to send me comments by mail:
stefan.gustavson (at) gmail (dot) com

About the cover

The cover image is a demo GLSL shader from my Shadertoy account. It showcases many of the techniques covered in this book, among them several variants of noise, bump and displacement mapping, scatterings, hybrid methods (a precomputed distance field for the text) and anti-aliasing for magnification and minification.

The object is a perfectly smooth sphere rendered with raycasting, using no ordinary texture images, and the shader runs fine in real time even on a mobile phone.

The shader can be viewed live on: https://www.shadertoy.com/view/sdfyWX

No facets, no texels, no jaggies!

Foreword

This book was written to fill a void. For decades, the textbook of choice for my M Sc course on procedural methods, and in fact the only textbook available which put a specific and detailed technical focus on the subject, was “Texturing and Modeling: a Procedural Approach”, in its third edition from 2003, edited by David Ebert and written by him and an impressive list of co-authors who all helped shape the field: Ken Musgrave, Darwin Peachey, Ken Perlin and Steven Worley. It was affectionately called “the procedural Bible”, and for good reason.

However, that book was taken out of print several years ago by the publishers due to low demand. Textbooks on academic subjects outside of the mainstream tend to struggle with sales and, sadly, this one was retired when their already printed stock ran out. Admittedly, some chapters had become outdated and would have needed a rewrite, but most of the book was still good, and it was our only option.

In a failed attempt to replace the printed version with an e-book, the publishers inexplicably chose to resurrect the *first* edition of the book, which was published in 1991 and is both badly outdated and considerably less well written. Not even the revised second edition, which replaced the first edition in 1998, made it into the e-book format, and the much improved and critically acclaimed third edition from 2003 is now impossible to obtain other than from libraries, in single copies on the second hand market, and illegally as a pirated PDF.

After the third edition of the book was taken out of production and replaced with an embarrassingly low quality scan of an ancient edition selling at a ridiculously high price, I started giving out more detailed lecture notes and short write-ups of my own which became more extensive with every year. This book is basically a rewrite of those notes, with lots of additions and revisions.

If you get your hands on a copy of the third edition of the classic textbook (legally, of course), I still recommend it. You can skip the chapter on hardware shading, but the rest is good reading. I wouldn’t bother with the first edition, though, other than as a historic curiosity. Paying for the e-book version of that is a waste of money.

Contents

Foreword....3
1 Introduction....7
2 Scope....9
3 Background....11
The beginnings....11
The Dark Ages....14
The Renaissance....15
The new dawn....16
The state of the art....17
Pros and Cons....20
Disclaimer....23
4 Fundamentals....25
The concept of a shader....25
The art of not actually drawing....26
GPU shaders....26
Software shaders....27
Drawing a circle....28
Shading a circle....29
5 Patterns....39
1-D repetitive patterns....39
2-D repetitive patterns....44
6 Anti-aliasing....59
Types of aliasing....59
Reasons for aliasing....60
Classic remedies....61
Procedural remedies....63
7 Randomness....83
Pseudo-random numbers....83
Hash functions....83
Permutation tables....85
Integer hash....85
Floating point hashes....86
8 Fractals....93
What is a fractal?....93
Faking a fractal....99
9 Noise....105
The need for noise....106
Making noise....108
10 Noises....117
Creases....117
Zero crossings and scaling artifacts....118
Anisotropy....120
Complexity....120

Differentiability..... 122
2-D simplex noise..... 124
3-D simplex noise..... 129
Derivatives of simplex noise..... 132
Mistreating time as space..... 133
A modern, flexible noise function..... 133
11 Scatterings..... 134
12 Amalgamation..... 135
13 Animation..... 136
14 Displacement..... 137
15 Scale..... 138
16 Hybrids..... 139

1 Introduction

Ever since the advent of computer graphics, procedural generation of content has been part of its diverse toolbox. For a long time, its only application area was in off-line, software rendering, because CPUs didn't have enough processing power to render procedural content in real time, and hardware acceleration had its development focus put elsewhere. In recent years, however, the introduction of massively parallel and programmable graphics hardware with absolutely astounding processing power means that procedural methods can now be quite useful even in real time rendering, and there is renewed interest in the field.

The subject is deeply fascinating to me: challenging but rewarding, visually creative in a hands-on manner, and *fun*. Alongside my academic work in the field, it has been an enjoyable hobby over the years.

It's not a universal tool, far from it, but it deserves to be considered, and like with all tools, it's useful to know about its capabilities as well as its limitations so that you can make an informed decision on what to use for a particular task.

This book aims at teaching you how to create procedural content, to give you the clues you need in order to decide when to use procedural methods, and also to know when you are better off leaving this particular creative tool in the drawer.

"Use the right tool for the job!" − *Bob the Builder*

2 Scope

The title “Noise is Beautiful”, suggested by Ingemar Ragnemalm and immediately accepted by my not always very serious brain, is an attempt at not setting a boring title to a book about a fun subject. A more formal title for the book would have been “Procedural Methods for Computer Graphics”, because that’s what it’s about.

The subtitle for Part 1 is “Procedural Textures”. “Procedural” in this context means “governed by algorithms”, more specifically “generated by computations”. In most cases it’s also taken to mean “computed on the fly as needed”, which is one of its great strengths. Instead of storing a lot of data in advance in case you need it at render time, you compute exactly what you need, when you need it. The “pure-bred” version of procedural methods implies also that you store nothing for later, but instead rely on being able to recompute whatever you might need again. This might seem like a waste, but as we shall see at the end of the next chapter, “Background”, it can actually be an advantage.

The second word of the subtitle, “textures” means that we will restrict ourselves to computer graphics methods for creating *patterns*, with a strong focus on two-dimensional patterns, *surfaces*, but make use of higher-dimensional functions to facilitate surface mapping and animation. Methods for procedural generation of geometry will be touched upon, in the form of *height fields* for putting not only visual texture, but also geometrical *structure* to surfaces. Procedural generation of *objects* in a more general general sense will be the subject of Part 2 of the book.

The selection of subject matter for Part 1 was in part a personal choice: these are the procedural methods that I know best and enjoy teaching to people. However, I hope the selection makes sense. Computer graphics is a fascinatingly cross-disciplinary field which involves optics, physics, mechanics, mathematics and programming, and dips its toes rather deeply into several related subjects such as visual perception, aesthetics and creativity. That’s why it’s so much fun, but it also why a book like this one needs to omit a lot of potentially relevant things.

”When I use a word, it means just what I choose it to mean – neither more nor less.”

Humpty Dumpty in “Alice Through the Looking Glass” by Lewis Carroll

3 Background

Having been with the field more or less since its inception, I cannot avoid presenting some brief historic background. Procedural methods have been around for decades, and we shouldn't ignore their history, because it has quite a few lessons to teach us. Of course, historic "truths" are not always true *now*, and sometimes a disruptive technical invention turns "common wisdom" into plain falsehoods. This is a double-edged sword: procedural methods come into play in applications that were previously unsuitable for them, but they might also lose their advantage in applications where they used to shine. Both of these changes are natural and inevitable results of technical development, but that makes it all the more useful to know not just where these methods have been used before, but also where they have *not* been considered, and most importantly *why*.

Towards the end of this chapter, I will present the current state of the art and point to strengths and weaknesses of procedural methods in current applications of computer graphics. Knowing history is often a great help to understand the present, and a look at the current situation makes a suitable conclusion to the chapter.

"Let us consult The Writ Of Common Wisdom!"

Theodoric of York, Medieval Judge, played by
Steve Martin on "Saturday Night Live" in 1978.

The beginnings

Procedural generation of content was in fact central to many of the earliest experiments in computer graphics, because back then memory was in extremely short supply and texture images were cumbersome to create, unwieldy to store and slow to access. Computer graphics took its baby steps in the 1970's, when computer memory was measures in *kilobytes* rather than gigabytes, and merely storing the output image posed a problem. In that context, texture images were simply not affordable.

Perlin noise – by far the most famous algorithm in the field – is described in detail in the chapter "Noise", but its history deserves mentioning here. It emanated from a desire to add some visual detail to the otherwise plain, single-color, plastic-looking surfaces that were the signature of computer graphics in the 1970's and 1980's. It

was invented for the movie “Tron” from 1982, which had ground-breaking computer graphics sequences made by the pioneer company MAGI, and Perlin noise was formally presented to academic circles as a component for the Image Synthesizer system and its KPL pixel-processing language developed by Ken Perlin during his Ph D work at New York University. The imagery in his seminal 1985 article, “An Image Synthesizer” [https://doi.org/10.1145/325334.325247] took the entire graphics community by storm, showcasing a surprisingly wide range of natural-looking and visually interesting patterns that were generated by a fairly simple algorithm.

Images from Perlin’s original paper, © ACM 1985. ***Get permission or re-make.***

Around the same time, procedural texturing was picked up and used commercially by the Computer Graphics Group at Lucasfilm, which was later to become Pixar. The public release of Pixar’s rendering software *Renderman* in 1989 made procedural texturing and modeling available to a rapidly growing community of computer graphics professionals. While many creators used Renderman’s procedural approach to produce impressive images, Pixar themselves remained among the most creative and technically skilled users of their own software and

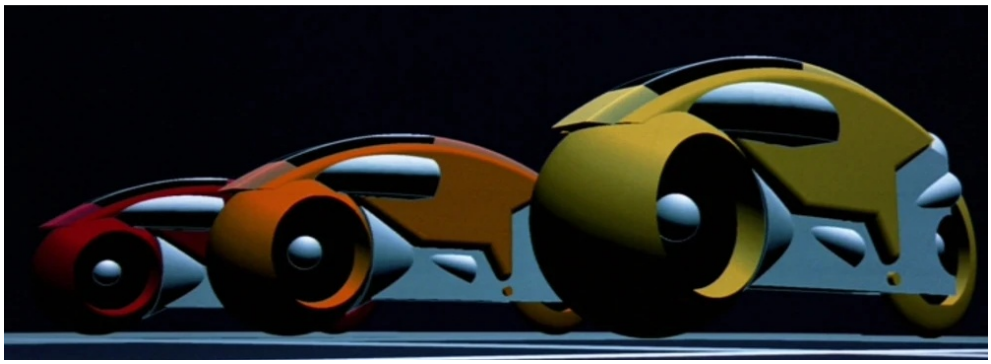
used it to render several groundbreaking productions of their own, including the first entirely computer animated feature length film, “Toy Story” from 1996.

Renderman is still a commercial product. Its internal rendering algorithms are now completely different from what they were in the 1980’s, but it maintains a procedural framework which is made easily available to its end users, and its modern Open Shading Language, OSL, is very similar to the original Renderman Shading Language, RSL. (In fact, RSL was simply called SL back in the day, because it was the only shading language around.)

In this context, it’s worth mentioning that the Renderman framework started out as a hardware project. From 1983 to 1990, Pixar developed and built several versions of a custom hardware renderer, the Pixar Image Computer. Due to its very high price it was never a commercial success, and only a few hundred units were sold before the project was terminated. One version of the Image Computer was used to render part of the award-winning short “Red’s Dream” which was released in 1987, but most of it was rendered in software, using general purpose computers to run what was to become Renderman.

A decommissioned Pixar Image Computer at a museum. (Public domain image)

Another curiosity that deserves mention is that the visual effects made by MAGI for the movie “Tron” back in 1982 were rendered using ray tracing – a comparably expensive approach that wasn’t used routinely in movie production until some 20 years later. The design of the “light cycles” in the famous racing scene is indicative of the limited set of primitives available to the artists. There were no polygonal models, only combinations of simple solid shapes: boxes, spheres, cylinders and toruses, and set operations between them: unions, intersections and subtractions.



Light cycles from the movie "Tron". ***© Disney – get permission or remake?***

The Dark Ages

Please note that the heading of this sub-section, "The Dark Ages", refers only to the narrow field of procedural methods. Computer graphics in general saw an absolutely incredible improvement around the turn of the millennium, and 3-D computer games aiming for photo-realism were the strongest reason for the development of reasonably low cost GPUs, which got us where we are today.

During the 1990's and early 2000's, the increased availability of memory and secondary storage, along with a rapid development of digital image capture and digital image editing, made it a simple matter to use texture images for surface detail, and this proved to be a quick and easy way to create images with visually interesting complexity. When low-cost 3-D graphics hardware for personal computers was developed, it was with a heavy emphasis on efficient handling of texture images.

Image-based texturing methods rapidly became the tool of choice for all hardware-assisted rendering. Procedural methods were used in software rendering only, and even there they fell out of use when texture images became easy to use. The creative community around Renderman remained active in their use and support of procedural textures, and procedural methods were still favored in some niche applications because of their flexibility and their small memory footprint, but from the mid-1990's most practitioners of computer graphics went all-in on texture images, and the developers of computer graphics hardware designed their chips for efficient handling of large amounts of high resolution texture images. For a while, most of the chip area of a graphics accelerator was dedicated to texture image handling. This is no longer the case, but a lot of the processing power is still spent on texture handling, and the amount of local memory on a modern graphics card for personal computers is comparable to the amount of memory in the host

computer system, in part because a *lot* of memory is required to store uncompressed texture images. The high performance of traditional, texture-intensive rendering methods on today's GPUs owes a lot to local memory with a wide and fast bus, along with clever strategies for memory management and caching.

Image-based textures are still a big deal, and they are definitely very useful, but for almost a decade, the alternative, procedural methods remained completely unsupported by hardware. We are still in a situation where most of the "surfacing" in computer games and computer graphics for visual effects is heavily lopsided towards using large amounts of high resolution textures, often created by small armies of texture artists who work exclusively in 2-D image editing software, and procedural methods are often dismissed out of hand, without even being considered. However, there *are* disadvantages with the image-based approach, alternatives *do* exist, and they are attracting well-deserved renewed interest.

The Renaissance

As development continued, the image-based methods for texturing made ever heavier use of "multi-texturing", where several texture images were combined to create the final appearance of a surface. This required considerable flexibility in how the textures were combined ("blended"), and the traditional fixed-function rendering paradigm became unwieldy, with lots of states to set and many stages to enable or disable. As a result, GPU cores were designed as programmable units to accommodate all the options, because it was the most flexible solution. At first, this programmability was hidden from the application programmer, but after a while it was made available as an option. The standardized "OpenGL ARB assembly language" ("ARB" is an acronym for "Architecture Review Board") was released as an extension to OpenGL in 2002. Because there were so few actors in the business of making GPU chips, and they all took reasonably similar approaches to their hardware design, the jointly developed standard was adopted by all.

The assembly language shader programs were crude by today's standards: they were severely restricted in length and complexity, loops were not supported, and they allowed only limited use of conditionals. Despite this, they made multi-texturing a lot simpler and more flexible, they allowed for novel non-traditional uses of texture images, and they even made it possible to use some very simple procedural texturing methods. GPU shader programming caught on quickly, and extensions were created to make the shader programs more capable. Soon, shaders could be much longer and have branches and loops, and the low level assembly language became inadequate.

The first high-level shader programming language for GPUs was Cg ("C for graphics"), a compiled language with a C-like syntax designed by NVIDIA. Because Cg

programs compiled into standardized ARB assembly instructions, Cg could be used also to program other manufacturers' hardware. Microsoft created its own HLSL ("High-Level Shading Language") with a very similar syntax and the same capabilities, and in 2004 OpenGL released its GLSL ("OpenGL Shading Language"). HLSL and GLSL both borrowed heavily from Cg, which in turn was based on the Renderman Shading Language, RSL – and for good reason, since RSL had proven to be a very useful tool.

The new dawn

Modern computer architectures are facing a huge problem with *memory bandwidth*, and this is especially true for GPUs. Many parallel processing cores can be put on the same chip and work together to achieve astronomically high theoretical processing speeds. However, in practice, those theoretical speeds are obtainable only for a narrow class of problems: those requiring comparably small amounts of memory and little or no communication between processing units. Many traditional computation problems don't belong to this class, and don't lend themselves well to massively parallel execution. However, the kind of algorithms that have traditionally been used in real time rendering *do* fall into that category, and that's why GPUs with hundreds or even thousands of processing units have been so successful in improving the quality and speed of real time 3-D computer graphics.

While real time rendering algorithms are a good fit for parallel processing, basically allowing for each pixel to be computed independently of all others, the handling of large amounts of *texture images* poses a problem. Texture data needs to be accessed by multiple processing units at once, with those units probably wanting *different* pieces of data, and memory access has become a bottleneck. There are workarounds involving local cache strategies, and even custom memory architectures that allow several simultaneous reads of different addresses, but memory access remains a fundamental choke point.

For this reason, massively parallel processing often involves considerable idle time while execution units are waiting for data and have nothing much to do until that data arrives. If such idle time could be filled with useful work that didn't require any memory accesses, a lot of untapped power could be used almost for free. (It wouldn't be *completely* free, because a busy unit uses more power than an idle one, but it wouldn't take more *time*.) This would make it a lot easier easier to come close to the theoretical maximum performance of massively parallel processing.

Enter procedural methods! Mixing texture images with computed procedural textures has the potential of increasing the complexity and quality of the rendered images without requiring more processing time. In the common case where memory access is the primary limiting factor for rendering speed, replacing some

texture images with procedural textures can even *increase* the rendering speed, even if the procedural textures take somewhat *longer* to compute. It sounds counter-intuitive that using a pre-computed lookup table of values is *less* efficient than to compute each value anew for each execution and throw it away after use, but it can happen, and *does* happen, in a massively parallel execution environment.

Another contributing factor is that memory access of a large RAM bank is quite slow (relatively speaking – it's still unbelievably fast from a human perspective), and has a high latency if you don't access data in sequential order. With clock speeds for operations on internal GPU registers now in the GHz range, one access to RAM could take several dozen clock cycles, and you can actually compute quite a lot in that time. In a modern GPU, it's definitely a good idea to make use of simple procedural patterns where you can. "Simple" in this context means "having low computational complexity", patterns that are easy to describe in algorithmic form and require small to moderate amounts of work to compute. As we shall see in the following chapters, such patterns don't have to be *visually* simple.

So, to summarize, if you can compute something with only a small amount of work, it's sometimes a better idea to compute it on the fly and discard it after use, rather than to store it in memory. In reality, if an algorithm is hampered by memory latency, modern compilers can play tricks with rearranging the order of low-level instructions so that the time to access memory is hidden and execution doesn't stall while waiting for data, but it's not always an option. Hardware-assisted computer graphics rendering is often memory bandwidth limited. In those situations, untapped processing power can be available "for free".

The state of the art

At the time of this writing (2024), shader-programmable graphics hardware is the default even on low-end personal computing devices, including low cost laptops and budget smartphones. This has been the case for quite some time – enough to completely phase out older devices which lack that capability. Performance varies wildly between high end and low end GPUs, but they can all be programmed using the same shading languages. The JavaScript version of OpenGL called WebGL uses GLSL shaders. HLSL, which is still being used in some applications, is very similar to GLSL. A still emerging standard for web-based 3-D graphics, WebGPU, has a shading language, WGSL, which is also very similar to GLSL. The 3-D graphics API Vulkan, which is expected to replace OpenGL in the long term, currently uses GLSL as its language for shader programming.

For software procedural shaders, which are still a thing despite all the current excitement around GPU rendering, the open standard "Open Shading Language", OSL for short, has now completely replaced Pixar's licensed product RSL, even in

Renderman. However, the similarities are striking, and both the old RSL and the modern OSL are quite similar to GLSL both in terms of syntax and the set of functions available to shader writers. OSL and GLSL were both based on RSL.

Speed

In terms of hardware rendering speed and processing power, there's a huge difference between a high-end personal computer built for gaming and a low cost laptop computer or a smartphone. Today, almost every graphics-capable device has some kind of dedicated GPU, but the best ones have several thousand processing units running at a speed of around 2 GHz, while budget GPUs for laptops can have less than one hundred cores that are less capable, operating at a slower speed and with less memory bandwidth. Smartphones usually have GPUs with only a few cores, because they must be small and operate on low power.

As development continues, all categories of GPUs are still improving with time, but the raw speed of two devices that are available on the market at any one time can differ by one hundred or even one thousand times, and that needs to be considered by application programmers.

In concrete numbers, a current gaming GPU, which is allowed to dissipate up to 500W power and is often the most expensive component in the computer, can process tens of *billions* of pixels per second. This should be compared to the needs for straightforward rendering of animated interactive content at 4k resolution and 60 frames per second, which translates to an output of "only" 500 million pixels per second. There is considerable extra capacity in a high end GPU which can be used to improve the image quality in various ways.

Realism

Software rendering achieved photo-realism quite some time ago. Except for some applications like synthetic human actors, where viewers are extremely picky with details, it's no longer possible to tell the difference between reality and fake in visual effects that are well done. Hardware rendering isn't quite as good yet, but it's rapidly getting there.

The gap in visual quality between real time graphics and pre-rendered content is closing, and it's not always possible to tell the difference. Some modern computer games look better than a lot of run-of-the-mill pre-rendered content, and some pre-rendered content take a deliberately simple approach, either for budget reasons or to achieve a "retro" look. Game engines are now being used to render cinematic content with a GPU, trading higher quality for reduced speed. We have reached a point in technical development where the quality of 3-D graphics is no longer as

heavily restricted by technology limitations or the output format. Budget, creativity and skill are now the main limiting factors for quality, and before long they will be the *only* limiting factors.

Trends

Ever since GPUs were introduced, they have been used for general computations as well. At first, this required workarounds and hacks to represent input and output as graphics data, because that's the only kind of data that was handled by the available programming interfaces. Gradually, however, GPU manufacturers embraced the dual role of their hardware, and GPUs are now routinely being used for pure computations, with neither input nor output having anything to do with graphics. New programming languages have been developed specifically for GPU-assisted computing, and GPUs have evolved to support higher numerical precision than what graphics computations would require.

Certain non-graphics algorithms lend themselves well to a massively parallel implementation, like the recent trend of "deep learning" algorithms for "AI", and large clusters of GPU hardware are used to process the massive amounts of data required for the training of those algorithms. (Speaking of AI, the current breed of algorithms falls far short of being actual artificial *intelligence*. It's just statistics and a layer of semantics to process words, images, sound, music or whatever the algorithm is meant to handle. At the heart of it, the algorithms are still derivative and dumb, *not* creative and smart. But let's get back to the point.)

There is currently a clear trend towards making GPUs even more general and flexible in their architecture, partly because they are being used for general computations, but also because real time computer graphics is moving into territory that was previously reserved for off-line rendering. The most notable change is that GPUs can now render scenes with ray tracing, albeit still in a much more restricted manner than traditional off-line ray tracing performed entirely in software. Traditional hardware rendering required only a small amount of local data from a scene to render one pixel. Ray tracing, however, is a *global illumination* rendering method which requires access to a lot of information about the scene for the processing of a single pixel. It's also notoriously difficult to predict exactly *which* data will be needed by each processing unit, which makes is harder to prepare the data for quick and efficient access. For this reason, manufacturers have had to rethink many of the traditional choices concerning the internal architecture of a GPU. Without going into detail, let's just say that a modern, high-end GPU is now a much less restricted and more thoroughly programmable machine than what used to be the case a decade ago, and the architectural changes are trickling down to the lower end designs as well.

Some computer graphics methods which were previously only possible to use in off-line rendering have now become practical to employ for real time rendering. Ray tracing is one such method, and in recent years it has received enormous attention. Another example is procedural texturing, but that has received very little attention so far. While this author is certainly biased, I think it deserves to be explored and used more in real time applications.

Pros and Cons

Procedural texturing is a specialized tool, not a universally applicable method. The obvious alternative is image-based texturing, but both methods have strengths and weaknesses. They both deserve to be considered, also in combination.

Generality

A digital image can depict any 2-D pattern, while not everything can be described in algorithmic form – at least not easily. Image-based textures can always do the job, even if they're not perfectly suited for the task. This is not the case for procedural textures, and this is probably one of the reasons why they aren't used more. You can always get by with slapping an image texture onto a 3-D model, but you can't use procedural texturing for everything.

Flexibility

A digital image may seem easy to change using modern image editing software, but it does take time and resources, and the editing is an off-line operation that needs to be performed in advance. A well designed procedural texture, on the other hand, can be made extremely flexible by clever use of parameters, and its pattern can be changed to fit a wide range of situations without any image editing. Such changes can be made very late in the production process without causing any delays or extra costs. For interactive content, changes can even happen dynamically at runtime. Having a quick and easy way to make changes to the appearance of surfaces can make for a more creative and exploratory production environment.

Quality

It would seem that photo-realism is easier to achieve if the textures are digital photos of the real world, but digital images have inherent limitations. The most prominent of these is *resolution*. A digital image consists of sampled pixels, and it has no means for representing details that are smaller than one pixel. A view up close of a surface that has an image texture on it will result in the pattern becoming either blocky or blurred. In off-line productions, extreme close-ups can be either

avoided or planned for, but real time interactive rendering, like in a computer game, can't always prepare for where the user wants to go and take a closer look.

Procedural patterns can have an effectively "infinite resolution", because they are computed at the resolution required at render time. This is similar to how *object graphics* work in 2-D. In object graphics, you describe contours of objects as ideal curves, and the shapes are rendered into pixels at the resolution required by the current view on the current output device. This technique is being used routinely in several applications of 2-D graphics, perhaps most commonly for rendering text, and nobody thinks twice about it. It's simply the best tool for some jobs.

Resolution is one of the most important aspects of visual quality. Digital images can be of very high resolution, but their resolution is always *limited*. Procedural patterns don't have that problem. Edges can remain crisp as you zoom in, and more detail can be added dynamically as required by the viewing situation. The quality can also be dynamically adapted to what the output device can handle.

Speed

Looking up the value of a pixel in a texture image which was stored in RAM doesn't come across as a complicated operation, but it *is* a memory access, and RAM speeds are lagging behind processor speeds. One clock cycle in a CPU or GPU is less than a nanosecond, but a RAM readout can take tens of nanoseconds or more, depending on how the data is accessed. Furthermore, in a massively parallel execution environment, memory reads can cause congestion when many processing units want different data from the same memory. Contradictary to "common wisdom", it can sometimes be faster to compute a simple procedural pattern than to perform one memory lookup. Given that most image-based textures involve more than one image and some mathematical operations between them, image-based textures can take a lot of time to process. A procedural pattern *could* execute faster.

A modern GPU is heavily tailored to handle image-based textures efficiently, so in most cases, procedural patterns won't cause a dramatic speedup. However, they aren't always a drag on performance either, and deserve to be considered.

Efficiency

A modern game, or a modern movie production involving 3-D effects, requires a lot of "assets", the bulk of which is usually comprised of texture images. Procedural textures are short snippets of code, an extremely compact representation of a pattern even compared to a low resolution digital image. In applications where the bulk of raw data is a factor to be considered, procedural methods can really shine. A modern premium quality game is seldom shipped on physical media, but requires

a download of up to tens of gigabytes, which is inconvenient. Situations where this can be even more of an issue include content that needs to be transferred across wireless networks, possibly at a low speed and at a considerable cost, and sometimes in a streaming situation where the bandwidth is limited.

Considering the efficiency of graphics hardware, it should be noted that a considerable portion of the chip area in a modern GPU is dedicated to handling of texture images. If even a fraction of that chip area was instead spent on dedicated functions to speed up procedural textures, perhaps by including a hash function (See chapter 7 , “Randomness”) and maybe even some dedicated “noise units” (chapter 9 , “Noise”), it could make a considerable difference.

Viability

When promoting procedural texturing as a method (which is essentially what we’re doing here), a common objection is that it’s difficult to do. Arguably, 3-D graphics is difficult to do no matter what, but it’s a valid point. Procedural textures are *programs,* and compared to digital images, a different kind of talent is required to create them. People who can use programming and mathematics as creative tools for visual art are not nearly as easy to find as people who can use 2-D image editing software to paint and edit images, and a procedural pattern can take considerable time and effort to create even by an experienced shader writer. Furthermore, parameterized procedural shaders are not always an easy fit for existing production environments. Change is often difficult, and old habits die hard.

As stated above, programmers with mathematical skills and a desire and talent for visual creativity are not obviously easy to find. However, experience has shown that such people do exist (readers of this book being great examples), and in some circumstances they can work wonders. Shader programming can be difficult, but it’s fun and rewarding work for a person with the aptitude for it, and while texture images are usually throwaway products that are created for one occasion only, procedural shaders – like all program code – can be re-used, perhaps with some minor changes, to be useful in another, completely different situation.

In situations where performance is still an issue and procedural textures would be non-viable, they can be rendered to ordinary texture images on the client side, as needed. Because hardware shading adheres to a universally adopted standard, platforms that don’t have enough processing power to render procedural textures in real time still have the capability to render them offline in advance. Thus, procedural methods can be used for maximum quality where the output device allows it, but still scale well to low performance devices that can’t quite handle them at full speed.

Disclaimer

The two preceding sections, “The state of the art” and “Pros and Cons”, contain observations and figures that will change over time, and some assumptions about upcoming technical development which, like all predictions about the future, may prove to be wrong. The general discussion should still be valid years from now, but the data and the conclusions might change. However, if I were to make an educated guess, further development will not make procedural methods *less* useful. I would personally consider them likely to become *more* useful over time.

Of course, that’s just my opinion, but I wrote this book because I think I’m right.

4 Fundamentals

Shader programming started out as a discipline in software rendering, but it has since become useful for hardware rendering as well. We will try to cover both aspects in this presentation. Most of the code examples will be in GLSL, which is a shading language for GPU shader programming. GLSL is compatible with several widely available platforms for your own experiments. However, most of the examples are applicable to other shading languages, because they all have a common ancestry and a C-like syntax.

If you don't know GLSL, you might want to learn it. You don't necessarily *need* to know GLSL to read this book, but it's a quite simple language that's easy to learn and fun to use, and the presentation has lots of code examples throughout which you might want to try. Conducting your own experiments makes the reading a lot more fun, and it's a great way of learning.

The concept of a shader

Let's start by defining what we mean by a "shader".

A *shader*, at the technical heart of it, is really nothing more than a *function* taking multiple values as input and yielding possibly multiple values as output. For purposes of repeatability and portability, that function needs to be *deterministic*. Any apparent "randomness" should be illusory, predictably yielding the same output for repeated calls with the same input, regardless of where and when it's being run.

A shader also needs to have a structure that allows it to be executed for any point independently of others, assuming nothing about which other points might be computed, or in what order. When using hardware acceleration, a shader typically executes in a massively parallel fashion, computing up to thousands of output values simultaneously, using the same program but different input values. This is called SIMD execution, which is an acronym for "Single Instruction, Multiple Data".

A more general technical name for a function with these properties is "compute kernel". Languages for SIMD programming (like OpenCL and CUDA) use that term, but for the purpose of this presentation, we stick to the name "shader".

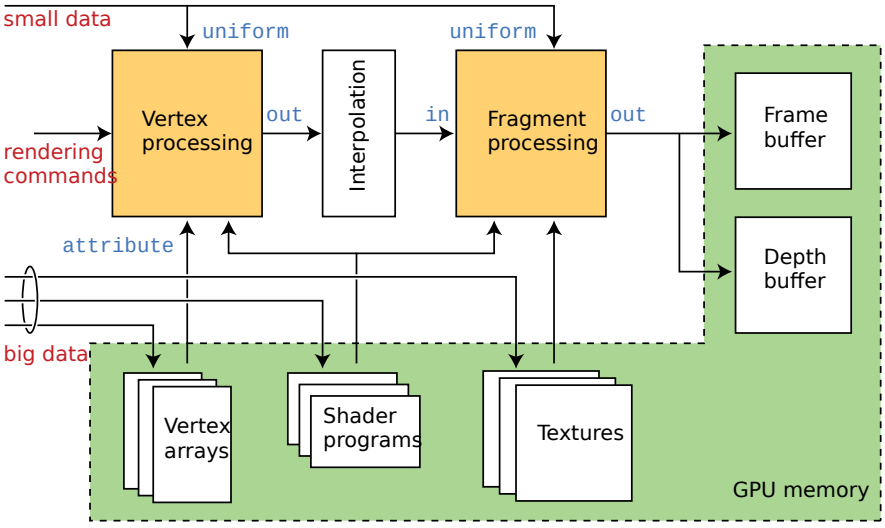
The art of not actually drawing

To get the hang of procedural texturing, you need to re-think some of the fundamentals of classic computer graphics. The most difficult threshold to get over is to get used to the fact that you aren't actually *drawing* anything in your shader program. You are designing a function $F(\bar{p})$ that tells the renderer what color (or other local property) a point at the position $\bar{p}$ should have, but you have no control over which pixels are drawn, in what order or in what resolution. It all needs to be computed at each pixel independently, using nothing but local data.

Such local data could include the position of the surface point, its surface normal, lighting information and texture mapping coordinates, but also any additional parameters you want, as long as they can be explicitly supplied to the shader.

GPU shaders

In traditional polygon-based hardware-assisted rendering, which is still the norm for real time content, you specify *vertex attributes* which are then interpolated, in a perspective-correct manner, between the corners of each triangle. The most basic GPU shader structure has two shaders: a *vertex shader* and a *fragment shader,* working together to compute the final output. A somewhat simplified model of the data flow and processing is in the figure below. The orange boxes are the shader execution units. (All types of shaders are now actually executed by the same set of general-purpose cores – the separation here is conceptual, not physical.)



A useful, simplified model of the data flow in a shader-capable GPU. The terminology used for the labels is from OpenGL and GLSL.

The vertex shader operates on triangle vertex data, taking as its input the position of a vertex and any additional attributes like surface normal and texture mapping coordinates. Its responsibility is to compute the final screen-space position and normal of the vertex after transformations, and possibly to modify some other attributes as well.

The fragment shader operates on the pixel level. The reason it isn't called "pixel shader" is that if multisampling is employed, the shader executes several times for each pixel, with the results being averaged to create the final output. The terminology isn't crystal clear, and while GLSL uses the name "fragment shader", HLSL actually calls it "pixel shader".

Potentially, a need for speedup could have the fragment shader being run more sparsely than once for each pixel and have its output interpolated, but that's not really an option with current GPU architectures. However, it *is* an option in procedural software shading, where the "shading rate" can be varied from several averaged shader evaluations per pixel to one shared and interpolated evaluation for every few pixels.

Software shaders

In software rendering, the original Renderman interface defined several types of RSL shaders, the most relevant here being *displacement shaders* and *surface shaders*. They are roughly equivalent to GPU-based vertex shaders and fragment shaders, with a few differences. One particularly important difference is that displacement shaders actually operated at a fragment level and could create fine detail on a surface, regardless of the vertex structure of the underlying geometry. This was possible because in the original software rendering algorithm used by Renderman, fine-grained surface tessellation down to roughly pixel-sized "micro-polygons" was inherent to the process. The split to micro-polygons happened immediately before surface shader execution, and the entire process was performed in software, which means that the tessellation could be dynamically refined as needed after the displacement, and the displacement shader could be re-run on the added micro-polygons.

Modern software renderers with support for programmable OSL shaders are better and more advanced in almost every respect compared to the original Renderman algorithm, but they still have considerable problems with replicating the flexibility and quality of classic displacement shaders in a strictly local illumination renderer.

Back in the "old days" (note that I am deliberately not using the term "*good* old days"), RSL displacement shaders could be used and abused to create complex geometry from simple primitives. This is still technically possible with OSL, but

it's no longer the best option, because a global illumination renderer needs to know the bounds of an object early in the rendering process. This requires early processing of all displacements for all objects, at least in an approximate fashion. When creating displacement shaders in a modern software shading framework, it's wise to restrict your displacements to small scale modifications of approximately correct geometry, rather than, say, deforming a sphere to make a banana. In software rendering, large deformations are now better done in advance than during rendering. RSL shaders performing extreme displacements were a fun and useful tool, and you can still find impressive examples of such shaders floating around on the Internet, but in retrospect they invited to abuse and were too strongly dependent on the rendering algorithm.

Drawing a circle

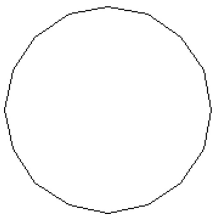
Procedural textures are commonly associated with visually complex, "busy" patterns with an apparent randomness to them. We will get to those patterns in chapter 7 and onwards, but let's start simple.

As an illustrative example, let's consider how we would draw a circle if we were to think in a sequential manner and set individual pixels in an image. Suitable math to use for this purpose is the *parametric equation* for a circle:

$$\begin{cases} x = x_0 + R\cos\phi \\ y = y_0 + R\sin\phi \end{cases}$$

In this equation, (x_0, y_0) is the midpoint of the circle, R is its radius, and the angle ϕ would be the *free parameter* for drawing. Varying ϕ between 0 and 2π sweeps out a full circle. To draw the circle, we could write a program like this:

Set starting point to $(x_0 + R, y_0)$
For angles ϕ *from just above* 0 *to* 2π*:*
    Compute the end point as $(x_0 + R\cos\phi, y_0 + R\sin\phi)$
    Draw a line from the starting point to the end point
    Set the end point as the next starting point
End



Now, there are several problems with this algorithm. First, it approximates a circle with a sequence of short line segments, which requires a decision on how small the steps in the angle ϕ should be to yield a good enough approximation. It's not even obvious what "good enough" would mean in any given situation. Second, the algorithm requires non-trivial computations with two trigonometric functions for

each iteration of the loop. Third, if the desired thickness of the line is not one pixel, it typically needs to be specified in screen space. Fourth, if anti-aliasing is desired (which is often an absolute requirement these days), it's not an easy task to do that right in a sequential line drawing algorithm. And finally, fifth, if we wanted to fill the circle with some color, it would be an additional and different problem.

There are other circle drawing algorithms, some not requiring an approximation with line segments, and some not even requiring any mathematical operations beyond integer addition and subtraction (rather surprisingly), but some fundamental problems remain: the drawing is explicitly sequential, and it's performed in screen space by setting individual pixels in a manner that doesn't lend itself easily to drawing thicker lines, performing anti-aliasing, or painting the interior of the circle. All these problems have of course been solved in classic computer graphics, but procedural methods can actually make it a whole lot easier.

Shading a circle

A shader function that specifies a circle would not use the parametric form, but instead use the *implicit equation*:

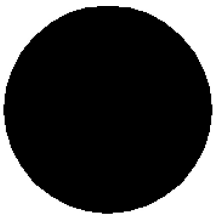
$$(x-x_0)^2+(y-y_0)^2=R^2$$

This specifies a condition for points on the circle perimeter. In a digital pixel image, the number of pixels which would have coordinates *exactly* on that perimeter would typically be zero, so let's start by specifying a condition for pixels that fall *inside* the perimeter, and render a filled circle instead:

$$(x-x_0)^2+(y-y_0)^2\leq R^2$$

This is, in fact, the entirety of our circle shader. In a naive GLSL implementation, the code would look something like this:

```
float circle( vec2 p, vec2 p0, float R ) {
	float r = sqrt( (p.x – p0.x) * (p.x – p0.x)
		+ (p.y – p0.y) * (p.y – p0.y) );
	if( r <= R ) return 1.0;
		else return 0.0;
}
```



The vector ***p*** is the point to be shaded, ***p0*** is the midpoint of the circle, and ***R*** is its radius. The return value is ***1.0*** for points inside the circle, and ***0.0*** for points outside.

It might seem wasteful to evaluate an equation for *every pixel* in the image just to fill some of them with color, but keep in mind that shaders are typically meant to draw *patterns*. Usually, every pixel on the surface should have *some* color, we just need to decide *which* color. We are "painting" both the background *and* the foreground at the same time, with the *same* shader. Instead of first setting all pixels to a background color and then asking "where do we put the circle?", we turn the problem inside out and ask "is this point part of the circle or the background?".

Please take the time to let the paragraph above sink in. If you didn't understand it completely, read it again. Getting the hang of this inside-out way of creating a pattern is central to the very concept of procedural texturing. It might seem awkward and roundabout, and sometimes it is, but *this is how shaders work*.

Bad code and stupid compilers

The code above is "naive", meaning it's bad. This is deliberate, because it serves a pedagogical purpose to point out exactly *how* it's bad, and what to do about it. It's bad in several ways, some far from obvious.

For starters, the fiddling with individual components of the vectors ***p*** and ***p0*** and taking the square root is merely computing the distance between two points, and there is built-in vector math and a built-in function for that. We should use that, because it's more likely to be compiled and executed in an efficient manner. It also makes the code more readable.

It would seem like an easy enough task for the shader compiler to find this pattern and map it to either of the appropriate built-in functions ***length*** or ***distance***, but keep in mind that shader compilers are not nearly as efficient with optimizing code as compilers for regular programming languages. Shaders need to be compiled quickly, often on the fly during execution of the application that uses them, and shader compilers sometimes take the easy way out and create inefficient code.

Before optimizing compilers were good at what they do, which was not all that long ago (we're talking about the 1990's), compilers often needed to be "spoon fed" with high level code that had a hardware-friendly structure, sometimes almost to the point of looking like assembly code. In some cases, snippets of actual assembly code could even be the best way to go. These days are largely behind us, because a human programmer is usually unable to find all the clever tricks that can be played with machine code to speed it up, make good use of all the capabilities of a modern CPU and avoid stalls in the execution. For shader compilers, however, *some* amount of spoon feeding can still be useful, and make a big difference for shader execution speed.

Optimization

Let's rewrite the code to use vector math and a built-in function:

```
// Determine whether p is inside a circle with radius R and midpoint p0
float circle( vec2 p, vec2 p0, float R ) {
	float r = distance( p0, p ); // or length( p – p0 )
	if( r <= R ) return 1.0; // inside the circle
		else return 0.0; // outside the circle
}
```

This is at least potentially faster code, and it's a lot more readable, not only because of the comments. Shader programmers are notorious for creating unreadable code, but it doesn't have to be that way. Comments are allowed and should be used, and most other recommendations on how to write readable and maintainable code for any other programming language are perfectly applicable to shader programming as well: comment your code, use reasonably descriptive variable names, comment your code, split long expressions into readable parts, stick to a commonly accepted coding style, and *comment your code*. (Did we say "comment your code"? Okay.) You can use intermediate variables and named constants freely, because those will definitely be optimized away. Shader compilers are not *terminally* stupid.

Over-optimization

If we want absolutely maximum performance from this shader, we *could* consider rewriting it to avoid computing a square root (which is implicit in the functions ***length*** and ***distance***) by instead using the square of the radius in the comparison:

```
float circle( vec2 p, vec2 p0, float R ) {
	vec2 v = p – p0;
	float r2 = dot( v, v );  // Compute v.x*v.x + v.y*v.y by a scalar product
	if( r2 <= R*R ) return 1.0;  // Extra multiplication to save one square root
		else return 0.0;
}
```

That, however, would cause some problems for our next step, where we want to draw the outline of the circle rather than painting its interior. There is seldom any need to hunt for things that could save single clock cycles in a shader, except for when you write library functions that are meant to be used and re-used a lot. In those cases, clarity could certainly be sacrificed for speed. However, try to avoid unnecessary convoluted code, and always use comments to explain what the code does if it's not immediately obvious to a reader who isn't also the author.

In this case, it's not even certain that a multiplication executes faster than a square root. Computing the of length of a vector and performing normalization are very common operations in computer graphics, and seeing how that requires square roots, there would typically be pipelined hardware inside the GPU for performing it about as quickly as a multiplication. Compilers for different platforms are also likely to behave differently when you try to speed up shader code. GPUs of different models and from different manufacturers can be very differently implemented at the lowest architectural level, and even minor updates to the graphics driver – of which the shader compiler is a part – could eliminate a slight speedup, or even turn it into a slight slowdown.

Use common sense when writing shader code, but keep in mind that refactoring for speed *can* be useful. A shader compiler is required to be quick, and is therefore less competent at performing automatic optimizations than a regular compiler, which can spend a lot more time to analyze your source code.

A peculiar quirk: the "step" function

There is still bad code in the example. The ***if-else*** statement is a common sight in most programming languages, and its structure is familiar to most programmers, but when used in a shader program it has several drawbacks.

This particular statement does the same kind of thing in both branches: it returns a value. At the hardware level, this maps to a very efficient "selection" instruction, where a logic condition (true or false) selects one of two values. (This, by the way, was the *only* conditional instruction in the original ARB shader assembly language.) However, the ***if-else*** statement is not required to do that. It's perfectly legal to write code that does completely different things in the two branches, and the compiler may or may not recognize that this is, in effect, a simple conditional assignment.

In C and C++, and many of their descendants, there is an infamous "ternary operator" which performs exactly this kind of conditional selection of either of two values. The two code snippets below yield exactly the same result, and are valid code in C as well as in GLSL:

```
float inside;
if (r <= R) inside = 1.0;
else inside = 0.0;
```

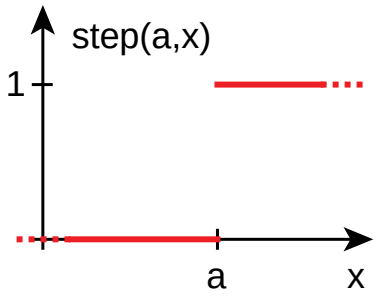
```
float inside = (r <= R) ? 1.0 : 0.0;
```

The code to the right is more compact, but it could be argued – successfully – that it's also less readable for a human, unless said human is very used to reading that kind of code. However, the *structure* of the more compact statement is different,

and more hardware-friendly, because it explicitly tells the compiler that we are assigning either of two values to one variable, and not making a conditional choice between two completely different, possibly long sequences of statements.

Shaders frequently need to perform this kind of conditional assignment, and that's why GLSL has the ternary operator. However, for reasons that will become apparent in the section "Anti-aliasing" below, as well as in the next chapter, the preferred way of making an "if-else" decision on what color to use is by instead using a built-in function: ***step***. The definition and its graph are as follows:

$$step(a,x) = \begin{cases} 0 & for\ x < a \\ 1 & otherwise \end{cases}$$



*The **step** function*

This function, returning the fixed values 0.0 and 1.0 as the result of a numerical comparison between two values, is likely to be hardware accelerated in a GPU, and execute faster than a general ternary operator or an ***if-else*** statement. The shader compiler might catch this special case and optimize it, but it also might not.

Now, we can rewrite our circle shader function using ***step***:

```
float circle(vec2 p, vec2 p0, float R) {
	float d = distance(p0, p);
	return 1.0 – step(R, d); // equivalent to (d < R) ? 1.0 : 0.0;
}
```

Note that we needed to subtract the return value from 1.0 to flip it from being 1.0 *outside* the circle to instead being 1.0 *inside*. This flip is a very frequent code pattern in shader programming. Another way of performing it would have been to reverse the order of the two arguments and write ***step(d, R)***, but that is *not* a great idea, not just because it's confusing and wrecks code readability, but also for reasons that will become apparent when we move on to performing anti-aliasing just after the next section. But first, let's modify the shader to do what we actually wanted: draw the *outline* of a circle.

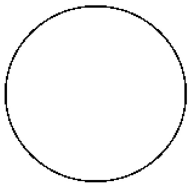
Drawing the outline

To draw a line, we need to specify a condition for when a pixel falls between the two edges of that line, and use it to select the color for the pixel. For the outline of a circle, we need to determine when the distance from the shaded point to the center of the circle is within a small margin from the radius. This requires two comparisons, which could be written as two logic conditions. If our desired line width is ***w***, the conditions would be:

```
return ((r >= R – w/2.0) && (r <= R + w/2.0)) ? 1.0 : 0.0;
```

with the same definitions for ***R*** and ***r*** as in the code in the preceding section. Now, the ***&&*** operator does exist in GLSL, and the code would work, but it's not the best structure. Using the ***step*** function instead, this becomes:

```
return step(R – w*0.5, d) - step(R + w*0.5, d);
```



If you are used to writing efficient code in C or C++, replacing simple comparisons with function calls might not seem like a bright idea, but keep in mind that ***step*** is a built-in, accelerated function, and there are currently no *actual* function calls in GLSL – everything is inlined by the compiler, and functions are just a means for structuring your code. Even if actual function calls were an option, the ***step*** function maps to only a few hardware instructions in total, and it would typically be inlined anyway.

Note that we also changed the division by 2.0 to a multiplication by 0.5. Mathematically, they are exactly the same, but a division is considerably more expensive to compute than a multiplication. Now, the compiler is almost certainly going to recognize this obvious division by a constant and replace it by a multiplication with its own pre-computed inverse of that constant, effectively doing the job for us, but it doesn't hurt to avoid a division even in the source. Either way, because of how the internal floating point representation of numbers works, a scaling by any constant power of two will map to a simple and very quick addition or subtraction in the exponent, and this entire paragraph might just be the ramblings of an old person who grew up with bad compilers. But still. For general variables ***a*** and ***b***, it takes considerably longer to compute ***a / b*** than ***a * b*** in most existing computer architectures, and a typical SIMD core has fewer units to perform division than multiplication. This *can* matter.

Our line width ***w*** is specified not in terms of device pixels, but in the coordinate space of the shader. In most cases, this is what we want. Any photo-realistic surface pattern would need the line width to be specified in object space rather than in screen space.

Of course, if we actually *want* to state the line width in screen space, it's perfectly possible to take a width specified in pixel units, transform it to object space and use that in the shader. Shading is a lot more flexible than sequential pixel drawing.

Anti-aliasing

We are still only drawing a binary pattern, "on" or "off", inside or outside the area we want to paint. Modern computer graphics is way past that. For a long time we have been enjoying shades of gray on our display devices, and even simple line drawings should utilize that for better rendering of edges. The jagged edges that appear when drawing sloped lines and curved contours with only two colors, foreground and background, are ugly and can be avoided. These "jaggies" are an inherent side effect of *point sampling*, where we just look at the center of a pixel and use that to decide what color to paint inside the entire pixel area. For reasons we won't go into here, but which you may know of if you have learned about signal processing, the "jaggies" are formally referred to as *aliasing*, and the process of removing them, or at least making them less obvious, is called *anti-aliasing*.

Chapter 6 is all about anti-aliasing. It will cover more than just smoothing out jagged edges, explain what aliasing actually *is* and why it's called "aliasing", but let's at least introduce the subject here, because aliasing really has no place in modern computer graphics. Graphics with jagged edges were acceptable twenty years ago, and in the early days of computer graphics jaggies were the norm, but not today. Procedural patterns do not get a pass, but need to keep up with the times.

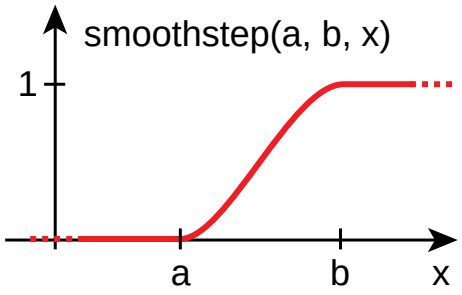
In traditional applications of 3-D computer graphics, anti-aliasing is the responsibility of the renderer, and it usually involves quite a lot of work. However, shaders are *functions* that can be programmed to change their behavior based on rendering resolution and viewing distance. This means that procedural patterns can perform *their own* anti-aliasing, making the renderer's job a whole lot easier.

Now is the time to reveal why we took all that trouble to use the ***step*** function to draw a circle: by replacing the hard stair-step with a gradual slope from 0.0 to 1.0, we can easily change our badly aliasing circle shader to a nice, anti-aliased version. The function we will use in place of ***step*** is called ***smoothstep***:

$$smoothstep(a,x)=\begin{cases}0 & for\ x\le a\\ curve & for\ a<x<b\\ 1 & for\ x\ge b\end{cases}$$

where *curve* is a third degree interpolating polynomial:

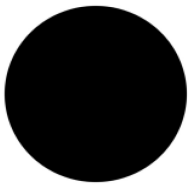
$$t=(x-a)/(b-a)$$
$$curve=3t^2-2t^3$$



*The **smoothstep** function*

Our shader doesn't need to change much to perform anti-aliasing of a filled circle. We can just replace the ***step*** with a ***smoothstep***:

```
float aacircle( vec2 p, vec2 p0, float R ) {
        float d = distance( p0, p );
        float w = …
        return 1.0 – smoothstep( d – w, d + w, R );
}
```



Of course, the value for ***w*** which determines the width of the smooth ramp needs to be carefully considered, and we skipped that detail above. To make the smooth transition region one pixel wide, we want ***w*** to give us a step from ***d*** of about half the size of a device pixel to either side, but we need to specify that distance in *shader coordinates*. The transformations between shader space and device space are known at render time, and this can be used to compute a proper step width explicitly. However, that's not the way it's usually done. Shading languages have built-in mechanisms to compute the step width *implicitly*, which makes it a *lot* easier.

In OSL, anti-aliasing of ***step(a, x)*** can be performed very conveniently by instead using the built-in function ***aastep(a, x)***, which means you can just replace each call to ***step*** with a call to ***aastep*** and be done with it:

```
// Anti-aliasing of an edge in OSL
float aacircle( vec2 p, vec2 p0, float R ) {
        float d = distance( p0, p );
        return 1.0 – aastep( d, R );
}
```

This is the reason why it's a good idea to use ***step*** to create crisp edges in a shader, instead of conditional statements: it makes it *very* easy to perform anti-aliasing. A procedural pattern that uses an ***if-else*** or similar "all or nothing" statement to decide a color is very likely – almost certain – to create ugly, jagged edges that require a lot of extra work by the renderer to be eliminated, or at least reduced.

In GLSL and other GPU shading languages, we need to write our own function to do the equivalent of ***aastep*** in OSL, but it works in exactly the same manner by a built-in mechanism called *automatic derivatives*. The inner workings and proper use of automatic derivatives takes some effort to explain. Here, let's just present how the same anti-aliasing could be performed in GLSL. The explanation for why it works, and how, will be presented in chapter 6 , "Anti-aliasing".

```
// Anti-aliasing of an edge in GLSL
float aacircle( vec2 p, vec2 p0, float R ) {
        float d = distance( p0, p );
        float w = fwidth( d ) * 0.5; // This is explained in chapter 6
        return 1.0 – smoothstep( d – w, d + w, R );
}
```

To wrap up this somewhat premature detour into anti-aliasing, let's just present code for implementing ***aastep*** in GLSL, for now without a proper explanation:

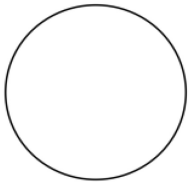
```
// Anti-aliased step function in GLSL
float aastep( float edge, float value ) {
        float w = fwidth( value ) * 0.5;
        return 1.0 – smoothstep( edge – w, edge + w, value );
}
```

Replacing the factor 0.5 with 0.75 often looks better, for reasons we will explain properly later (in, you guessed it, chapter 6).

Most people who use the built-in ***aastep*** in OSL don't need to care about how it works, and you don't need to be shy to copy this function for your own use in GLSL before you learn how it works.

Using our new toy twice, we can also create smooth, thin outlines.

```
// Anti-aliased circle outline
float aacirclerim( vec2 p, vec2 p0, float R, float linew ) {
    float d = distance( p, p0 );
    return aastep(R–linew*0.5, d) - aastep(R+linew*0.5, d);
}
```



---

5 Patterns

Most surfaces in the world around us have a pattern on them, either because of how they were made, or because they were decorated to look nice. This chapter deals with *repetitive* patterns, such as stripes, tilings, woven or knitted fabric, grids, polka-dot and checkered patterns, just to name a few. Random or random-like patterns that are not repetitive are common in the real world as well, but they will be covered later, starting with chapter 9 , “Noise”.

Image-based texturing often uses images that wrap around at the edges, so that they can be applied across a large area by being tiled seamlessly instead of stretched. Procedural patterns don’t *have* to be repetitive, but we might still want them to be. A formal word for repetitive is *periodic*, and periodic functions can be used to generate periodic patterns.

1-D repetitive patterns

When you think of periodic functions, your first thought is probably the sine and cosine functions. Both are built-in functions in all shading languages, because of their prominent role in computer graphics. They are likely to be highly accelerated in a modern GPU, and we can certainly use them for pattern generation.

A striped pattern could be created by a ***step*** function applied to a ***sin*** function. Assuming that ***x*** is one of the mapping coordinates for a surface, we can write:

```
float stripes = step( 0.0, sin( x ) );
```

This would create a pattern of stripes perpendicular to the ***x*** direction, with the variable ***stripes*** alternating between 0 and 1 with one stripe every 2π units. Changing the threshold value for the step function from ***0.0*** to anything between ***-1.0*** and ***1.0*** would change the relative width of the “0” stripes to the “1” stripes, and scaling the argument to the ***sin*** function would change the distance between stripes: scaling it by a factor larger than 1 would move the stripes tighter together, and scaling it by less than 1 would spread the stripes further apart.

However, there are several disadvantages with using the ***sin*** function for making simple stripes. First, its period is 2π, a rather inconvenient irrational number, while we would typically want to create patterns where the period is some round number,

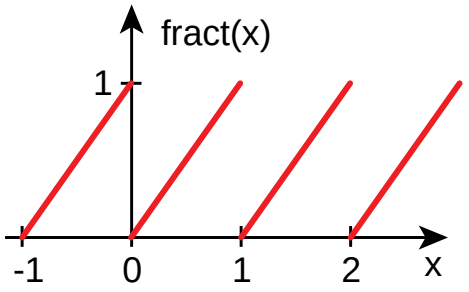
like 1.0 or 0.25. Second, the relation between the threshold value and the relative width of the “0” and “1” stripes is complicated. With the threshold at 0.0 we have a fifty-fifty proportion between stripes, but it takes some math to work out what the proportion would be if we changed the threshold to, say, 0.1, and also to determine the threshold value that would give us certain stripe proportions, like 30 % of “0” and 70 % of “1”. Third, the ***sin*** function is unnecessarily cumbersome to compute, even if it’s usually hardware accelerated. We can do better in all three respects.

A more straightforward periodic function is the built-in function ***fract (***or ***frac*** as it’s called in the two oldest GPU shading languages, HLSL and Cg). It takes the fractional part of a number, which of course repeats at integer intervals. Its definition and plot are shown below.

$$fract(x) = x - floor(x)$$

where $floor(x)$ is the largest integer that is smaller than or equal to x.

“Larger” and “smaller” are to be interpreted in terms of *signed* comparisons, not comparisons between absolute values.



*The **fract** function*

Note that the function has the same right-slanted saw-tooth shape for negative as well as positive values for the argument. The definition of the function in all current shading languages (but not necessarily in all *other* languages and libraries where a “fractional part” function is available) is precisely this, simply because it’s the most useful version for pattern generation. A periodic pattern should not change its appearance at an arbitrary sign change of the mapping coordinates – when crossing the origin, it should just keep repeating in the same manner.

A straightforward way to create a stripe pattern with the ***fract*** function would be:

```
float stripes = step( 0.5, fract( x ) );
```

A better “stripes” pattern, as code and rendered to a black and white image

This has none of the disadvantages of the ***sin*** stripes above: it has a period of 1, a linear relation between the threshold and the width of the “0” stripes, and a ***fract*** is

comparably easy to compute. (It's still not *trivial* to compute it in a floating point representation, but it's a quick process without iterations or approximations.)

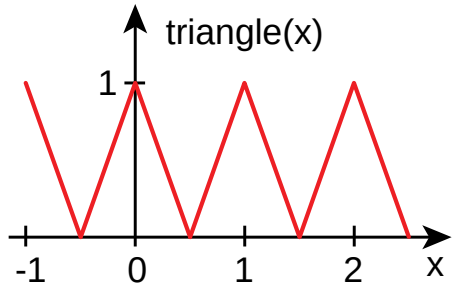
One flaw remains, however, and it concerns anti-aliasing. Where the ***step*** function acts on the upward ramp of the ***fract*** function, anti-aliasing is easy, but the ***fract*** function has a discontinuity at each integer, and discontinuities are notoriously prone to creating jaggies. A ***smoothstep*** can't smooth out the hard edges at integer positions, so the ***aastep*** we mentioned in chapter Error: Reference source not found won't work. We would much prefer what we send into the ***step*** function to lack any discontinuities, like the ***sin*** function, but to also have a nice linear ramp everywhere. What we want instead of a saw-tooth wave is a *triangle wave* with linear slopes both up and down.

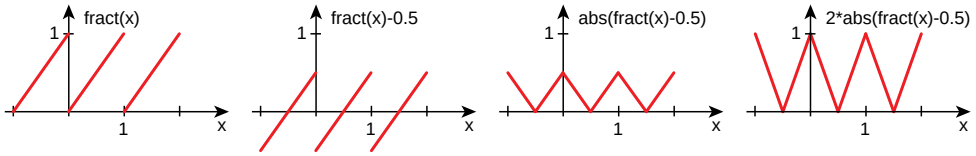
The most efficient way of creating a triangle wave is by the trick shown below. It's not completely obvious at first glance how it works, but it becomes clear if you plot the steps along the way.

$$triangle(x) = 2(|fract(x) - 0.5|)$$

where $|x|$ means "absolute value of x".

The scale factor 2 maps the result back to the range [0, 1], just for convenience.



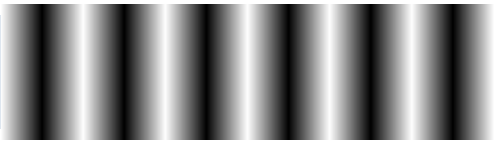


---

*A triangle wave function created from **fract** (and each step along the way)*

This triangle function requires only slightly more computations than the ***fract*** function, and putting it through an anti-aliased ***aastep*** function provides proper anti-aliasing of both edges of each stripe. Expressed in shader code, the function would be:

```
float triangle( float x ) {
        return 2.0 * abs( fract( x ) );
}
```

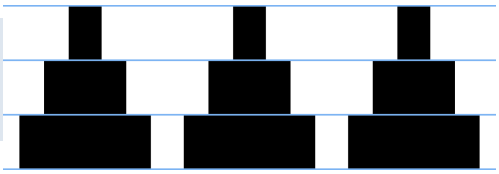


The triangle-wave function, as code and rendered to a grayscale image

This is not a built-in function in any current shading language, but it's simple to create it from ***fract*** in the manner shown here, and it's quite useful.

Creating stripes with the ***triangle*** function is just as easy as with the ***fract*** function:

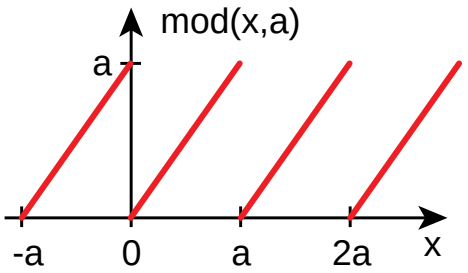
```
float stripes1 = step(0.2, triangle( x ) );
float stripes2 = step(0.5, triangle( x ) );
float stripes3 = step(0.8, triangle( x ) );
```



Well-behaved stripes (suitable for anti-aliasing) created with the triangle function. Note that both edges of the stripes move when the threshold is changed.

Another function that can be used to create periodic patterns is the *modulo* function, which is named ***mod*** in both GLSL and OSL.

$$mod(x,a) = a \cdot fract(x/a)$$



The modulo function, in its correct interpretation

The function shown here is the mathematically correct definition of a modulo operation, and it's the version that is useful for creating periodic patterns. Unfortunately, many programming languages, including some shading languages, have *incorrect* implementations. There seems to be great confusion about how to perform a modulo operation correctly on negative numbers, and there are a lot of broken ***mod*** functions out there. The ***floor*** function is less prone to misinterpretation and is usually implemented correctly, and from that you can redefine both the ***fract*** and the ***mod*** functions to work like they should:

```
// The fract function correctly defined in terms of the floor function
float correct_fract( float x ) {
        return x – floor(x);
}
```

```
// The mod function correctly defined in terms of the fract function
float correct_mod( float x, float a ) {
	return a * correct_fract( x / a );
}
```

If you ever have doubts about how a certain language with these functions implements them, test them, and re-implement them if necessary. GLSL and OSL do it right, but WGSL does its ***modf*** function *wrong*. That is not a bug, it's by (bad) design, and it's not likely to change. The ***%*** operator for "remainder of division" is a sorry mess with different meanings in different languages, so be careful with that.

Optimization

Note that the modulo operation involves a division, and it takes some more work to compute than ***fract***. If ***a*** is a constant, the shader compiler should be able to find that in the code above, and replace the division ***(x / a)*** with a multiplication by the inverse of ***a*** computed at compile time, ***x * (1.0 / a)***. However, this is not necessarily the case if you use the built-in function. If you want a modulo operation with a constant period, it could still be a better idea to create one yourself with an explicit constant instead of a variable as the denominator in the division. That will most likely be caught and replaced by a multiplication, even by a fairly dumb compiler. If you want to make absolutely sure, you could replace, say, ***x / 4.0*** with ***x * (1.0 / 4.0)*** in your source code, or even do the math yourself and write ***x * 0.25***, but there's rarely any need for that rather extreme kind of spoon-feeding nowadays. Things were different in this respect just ten years ago, but now even on-the-fly compilers for GPU shaders have become better at making such speedups on their own. Specifically, with modern GLSL compilers, ***mod(x, a)*** where ***a*** is either a literal constant like ***43.0*** or a ***const float*** type seems to compile to faster code than a fully general ***mod***. (You might want to verify that rather than take our word for it, because these things tend to change rapidly, and vary a lot between platforms.)

To round off this little side track into compiler technology, we should mention that manufacturers of shader compilers usually don't provide end users with details about optimization, and there is no guarantee that an optimization that is performed by one version of the compiler will be performed in all future versions as well. Sometimes it's not at all clear why a certain code variant is faster than another.

If in doubt, don't focus on performance, but concentrate on writing code that *works* and is *readable*. GPUs will continue to get faster. Writing for reasonable speed is one important aspect of GPU shader programming, but writing for absolutely *optimal* speed often ends up being an exercise in futility.

2-D repetitive patterns

1-D periodic patterns are common in the real world, and therefore very useful as procedural patterns, but of course we want to create 2-D periodic patterns as well. At the heart of a 2-D periodic pattern is a *tiling of the plane*, where many identically shaped small regions of the plane have their own local mapping coordinates.

Rectangular tiling

The most common kind of 2-D tiling is *rectangular tiling*:

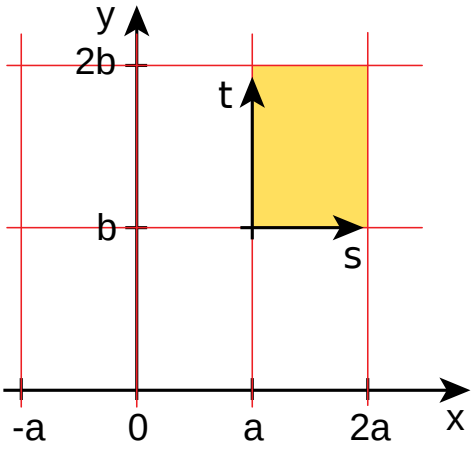
$$s = mod(x, a)$$
$$t = mod(y, b)$$

in GLSL vectorized code:

```
vec2 xy, ab, st;
…
st = mod(xy, ab);
```

or, if you want a grid of 1×1 squares:

```
st = fract(xy);
```



Rectangular tiling

In the example, the (s, t) local coordinates are in the range $[0, a]$ and $[0, b]$, respectively. Another, and often better, mapping is to scale and possibly translate the local coordinates to a normalized range like $[0,1]$ or $[-1,1]$. It depends on what you want to do. Whether you use ***mod*** to perform the coordinate wrapping, or ***fract*** with pre-scaling of the coordinates, is largely a matter of taste.

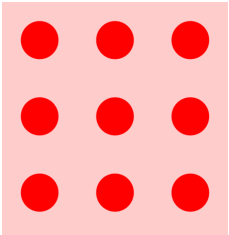
Polka-dots

Now, let's use our tiling coordinates to create some patterns! A polka-dot pattern could be created exactly like the circle we did previously, but using the local tiling coordinates to determine the distance from the center of the current grid cell:

```
// Polka-dot pattern in a square tiling.
// The range of R is 0.0 (no dots) to 1/sqrt(2) (no gaps).
float polkadots( vec2 p, float R ) {
        return 1.0 – step( R, length( fract( p ) – vec2 ( 0.5, 0.5) ) );
}
```

“Whoa, hold on”, you might say, and for good reason. This function does all its work in just *one line*. Shader programming is a creative, iterative process, usually involving lots of experimentation and quick edits, and it tends to make people write this kind of “one-liners”. However, leaving code in that state is a bad habit that makes it hard to understand, after some time has passed even for the person who wrote it, and therefore difficult to maintain. Let’s break it up for better readability by splitting the computation over several lines and using extra variables to store intermediate results. Even a very stupid compiler will optimize these away and create exactly the same code, because at the machine code level, variables simply don’t *have* names. It’s all just values stored in registers.

```
float polkadots( vec2 p, float R ) {
        vec2 q = fract( p ); // Local grid coordinates [0,1]
        q = q – vec2 ( 0.5, 0.5 ); // Coords are now [-0.5,0.5]
        float r = length( q );  // Distance from local origin
        return 1.0 – step( R, r ); // Circular dots of radius R
}
```



Polka-dot pattern in readable code, and a visual example with some color added

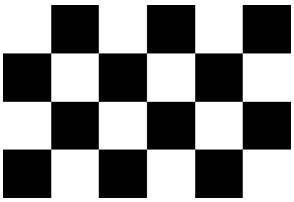
Note how the split into several lines made it possible to write detailed comments, and that the code is now a lot easier to read and understand, even without the comments. Try to resist the temptation to write “clever” one-liners. Your code will be unnecessarily hard to understand, even for yourself, and programming like that serves no useful purpose.

Checkerboard

A useful pattern which is often shown as an example of procedural texturing is the *checkerboard pattern*, with square regions in two alternating colors. This pattern is very common in human culture, because we like to make things pleasing to the eye by adding deliberate ornamentation, pretty much wherever we can. In tilework, much more visually interesting patterns are possible if the tiles are in at least two contrasting colors. At the extreme end of this is pixel-style image mosaics, but let’s keep things simple for now and just assign alternating colors to the grid cells.

A seemingly elegant, but also naive, code snippet for doing that is the following:

```
// A checkerboard pattern, in simple but naive code
float checkers (vec2 p ) {
        return mod( floor( p.x ) + floor( p.y ), 2.0 );
}
```



Checkerboard pattern done wrong (read on to learn why)

This is dangerously close to being a “clever one-liner”, so let’s break it down and add comments, at least for the purpose of this presentation:

```
float checkers (vec2 p ) {
        float steps = floor( p.x ) + floor( p.y ); // Stair-steps, integer-valued
        return mod( steps, 2.0 ); // Wraps all integers to only 0.0 and 1.0
}
```

Adding together the stair-step ***floor*** versions of x and y makes the sum increase or decrease by 1 when you move from one cell to the next along either x and y. Moving diagonally will either increase or decrease the sum by 2 or make it remain the same. Putting that sum through a modulo operation with 2 will make all even numbers wrap to 0, and odd numbers wrap to 1. Moving along x or y, the output value changes between ***0.0*** and ***1.0*** when we move across tile boundaries, but doesn’t change when we move diagonally across tile corners. This is exactly what we want. Right?

Except for one important thing: anti-aliasing. The pattern above looks fine as long as you keep it aligned with the pixel grid of the output device, but any rotation – particularly small rotations – any camera motion, perspective effect, or almost any kind of animation, will make the pattern look absolutely awful. In its current form, the function is basically useless for direct pattern generation. We need to do better.

Anti-aliased checkerboard

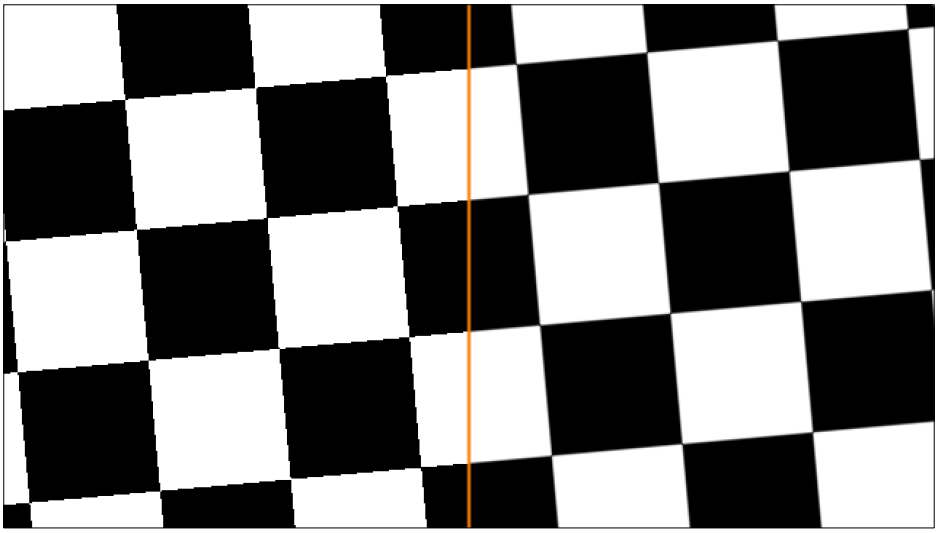
The fundamental problem with the checkerboard pattern above is with the discontinuities created by the ***floor*** function. We can’t use the output from that function directly to draw a pattern, because the edges will alias something terrible. Instead, we need to create a function with smooth ramps in both the x and y directions, and threshold those ramps with ***step*** to make the final edges.

Drawing from previous experience (yes, pun intended), we already have a way of doing anti-aliased stripes, and we can create one set of stripes running in the x direction and another in the y direction. To create a checkered pattern, we then need to overlay the two sets of stripes, and perform a sort of “exclusive or” mask-

ing operation between them to make the regions where two stripes cross take on the value 0. Without providing a full explanation of exactly why this is a good way of doing it, or how to come up with it in the first place, let's just say that taking the absolute value of the difference between the vertical and horizontal stripes achieves exactly that, and also preserves any smooth edges created by anti-aliasing:

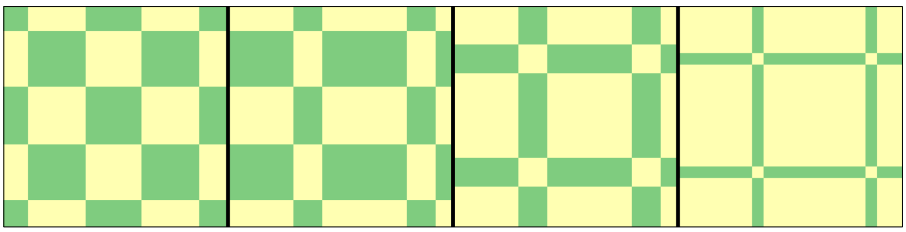
```
float aacheckers (vec2 p ) {
        vec2 ramps = 2.0 * abs( fract( p ) - 0.5); // "Triangle waves" in x & y
        vec2 stripes = step( vec2(0.5, 0.5), ramps ); // 50% stripes
        return abs( stripes.x - stripes.y ); // "XOR" at overlaps, preserving AA
}
```

This checkerboard pattern is created by thresholding of underlying functions without discontinuities, and will anti-alias nicely by changing the ***step*** to an anti-aliased variant of it, be it a built-in ***aastep*** or a function you supply yourself.



Checkerboard pattern, without antialiasing (left) and with anti-aliasing (right)

Note that either of the thresholds to the ***step*** function can be changed to give the pattern different looks with more variation.



Variations on the checkerboard pattern

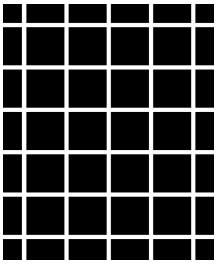
Now that you know how to do a checkerboard pattern that can be anti-aliased, *please* don't use the raw "floor-mod" version, except maybe for quick demos. It's simple code that's easy to remember, but apart from that it has no good qualities.

Anti-aliased grid lines

Grid lines is a very common pattern, not only in graphs and abstract drawings, but also in real world scenes, for example tilework. Plaid patterns, a very common decorative element in human culture, can be comprised of crossing stripes as well.

It's tempting to write a function for crossing (x, y) grid lines in the following manner, because it requires only one ***fract*** and one ***step***:

```
float gridlines( vec2 p ) {
        vec2 q = fract( p ); // Square grid of size 1x1
        vec2 grid = step( 0.9, q ); // Threshold x and y
        return max ( grid.x, grid.y ); // Overlay the two
}
```

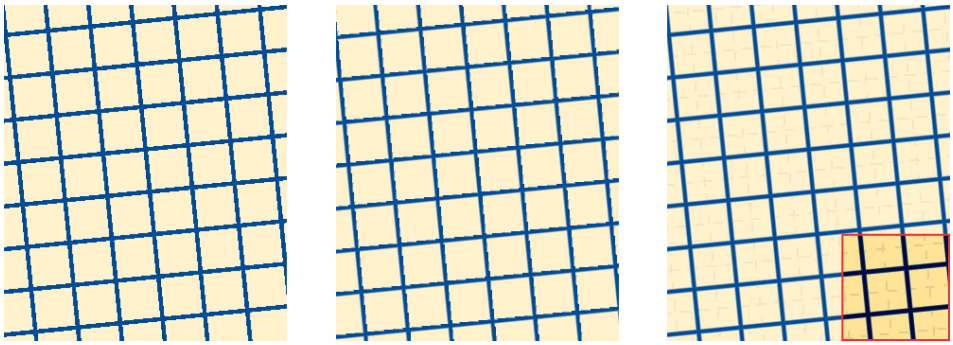


Grid lines coded in a naive manner, and a visual example

However, this creates particularly nasty aliasing at the discontinuities in ***q***, at the grid cell boundaries. Discontinuities in grid mappings are not necessarily *always* bad, but edges in the pattern should *not* line up with them. (Not *ever.* Seriously.) Instead, we should use two ***step*** functions for each line:

```
float gridlines( vec2 p ) {
        vec2 q = fract( p ); // Square grid of size 1x1
        vec2 grid = step( 0.45, q ) - step( 0.55, q ); // Both edges are explicit
        return max ( grid.x, grid.y ); // Overlay the two sets of crossing lines
}
```

The pattern is the same in both cases (except for a translation) but when you try to anti-alias the grid lines by replacing ***step*** with ***aastep***, the "quick and dirty" version won't let you smooth out the implicit edge where the grid coordinates make a jump from 1 back to 0. Making both edges explicit, by using one ***step*** function for each, is the better way by far to do it. Patterns that alias badly aren't nearly as useful as patterns that perform their own anti-aliasing.



Left: *Badly aliasing rotated grid.* ***Center:*** *Failed anti-aliasing by the naive code.* ***Right:*** *Mostly successful anti-aliasing by the improved code.*

Looking closely at the rightmost image above, the improved version has smooth edges for the lines, but faint cross-like artifacts can be seen in the squares between the lines, at the discontinuities of the coordinates we send to ***aastep***. A contrast-enhanced view is shown in the inset. This occurs because the ***aastep*** function fails to compute an appropriate step width at discontinuities. (A full explanation for this will be given in the next chapter.) To get rid of these artifacts, we could treat those parts of the pattern with extra care, or try to improve the performance of the ***aastep*** somehow. Both are perfectly possible solutions. However, we could also remember the lessons learned from our checkerboard pattern and use the "triangle" function:

```
// A pattern of (x,y) grid lines, done right
float aagridlines ( vec2 p ) {
	vec2 ramps = 2.0 * abs( fract( p ) - 0.5); // "Triangle waves" in x and y
	vec2 lines = aastep( 0.45, ramps ) - aastep( 0.55, ramps); // 0.1 wide
	return max( lines.x, lines.y ); // Overlap the two sets of lines
}
```

The result from this code looks the same as the rightmost image above, only without the faint artifacts. We fixed the problem, and at a very small computational cost. Because of how we made our triangle waves, the grid is now twice as dense with grid lines 0.5 units apart, but that can be adjusted by a simple scaling of ***p***.

Now, couldn't we get away with using a single ***step***, like for the checkerboard? Well, we *could*, but it wouldn't work well for thin lines. For the expected uses of this pattern, the two edges of the grid lines would usually be quite close in screen coordinates: within a single pixel of each other, or even passing through the *same* pixel. The threshold for the ***triangle*** function would be close to either ***0.0*** or ***1.0***, and anti-aliasing with ***aastep*** has problems near the pointy extremes of the ***triangle*** function. A single threshold works great for making reasonably wide

stripes, but the code above with two thresholds is better for making thin lines. The reason behind this will become clear in the next chapter, but until then, “trust me, I’m a doctor.”

Staggered tiling

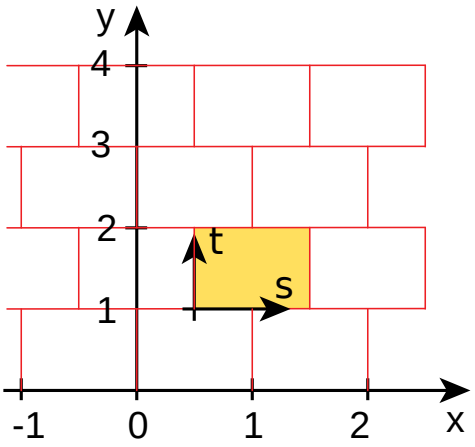
Variants of rectangular tiling are *skewed tiling*, where the tiles are not rectangles but parallelograms, and *staggered tiling* where successive rows or columns of rectangles are offset against each other. Skewed tiling will be treated extensively in the chapter “Noises”, so let’s save that for later and just do staggered tiling here.

There are many real world examples of staggered tilings, such as brickwork and floor tiles, and a common type has every second row of rectangles offset one half rectangle from the other rows:

$$s = fract(x + floor(y)/2)$$
$$t = fract(y)$$

or, in shader code:

```
vec2 xy, st;
...
st = fract( xy +
 vec2( 0.5 * floor( xy.y ), 0.0 ) );
```



Staggered tiling

Brickwork pattern

Staggered tiling is used in brickwork, and a brick wall pattern is a classic example of a procedural shader, so let’s just show that pattern here before we move on. Let’s make the bricks have the value 1.0 and the mortar between them 0.0:

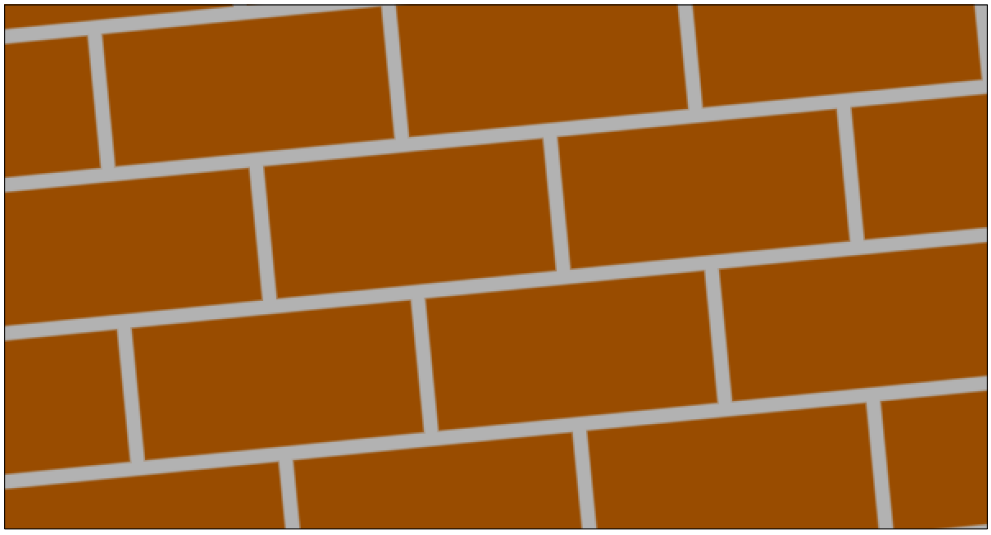
```
float bricks( vec2 p ) {
	vec2 q = fract( p + vec2( 0.5 * floor( p.y ), 0.0 ) ); // Staggered grid
	float brickx = step( 0.1, q.x ) - step( 0.9, q.x ); // vertical gaps
	float bricky = step( 0.1, q.y ) - step( 0.9, q.y ); // horizontal gaps
	return min( brickx, bricky ); // If either is 0, return 0
}
```

Again, this first attempt is not great for anti-aliasing, because there are discontinuities in the function we send through ***step***. We are not creating any edges exactly at the problematic positions where either of the local coordinates are close to 0 or 1, but like with the grid lines in the previous section, replacing the ***step*** functions with anti-aliased ***aastep*** versions will create faint but seemingly random artifacts at discontinuities in the ***floor*** function.

Let's fix that problem before we leave this example. Like before, we want symmetric ramps that meet without discontinuities at cell edges. It can be done like this:

```
float bricks( vec2 p ) {
	vec2 q = fract( p + vec2( 0.5 * floor( p.y ), 0.0 ) ); // Staggered grid
	q = 2.0 * abs( q – 0.5 ); // Change from sawtooth to triangle
	float brickx = 1.0 – step( 0.95, q.x ); // vertical gaps at both sides
	float bricky = 1.0 – step( 0.9, q.y ); // horizontal gaps at top and bottom
	return min( brickx, bricky ); // If either is 0, return 0
}
```

Because the ramps in x and y are now upwards slopes from the middle of the brick towards either edge, we can also manage with half the amount of ***step*** functions. Perhaps a bit surprisingly, we actually ended up making the code more efficient while also improving the anti-aliasing.



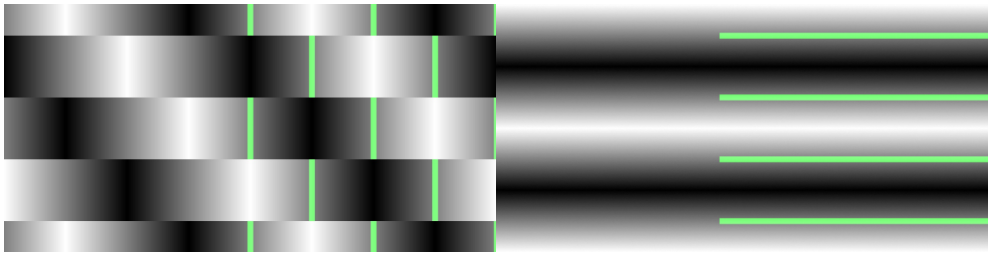
Anti-aliased brick pattern in color, rotated and scaled to make the bricks 2x wider

Now, we did advise against using this arrangement of thresholds in the case of grid lines in the previous section, and that warning still holds. There will be artifacts from ***aastep*** computing the wrong step width in zoomed-out views, where gaps

between bricks are around the size of one pixel. While grid lines are typically *meant* to be very thin, this is not necessarily the case for mortar between bricks, so this code might be good enough for many purposes. However, a little more thinking can remove this remaining flaw as well. Looking at the pattern we want, we can construct a triangle-wave mapping (s,t) such that all vertical lines are located at the center of the sloping ramps of the s coordinate, and all horizontal lines are at the center of the ramps of the t coordinate. The code for s is a horrible mess of a one-liner, but the image should make it clear what it does.

```
s = 2.0 * abs( fract( 0.5 * ( x +
    0.5 * floor( 2.0*y - 0.5) ) ) - 0.5 );
```

```
t = 2.0 * abs( fract( y ) - 0.5 );
```



Mapping without problematic discontinuities for the "bricks" pattern. The green lines show where the mapping coordinate is 0.5. This is where we place the gaps.

Using this mapping to create the brick pattern, the code would look like this:

```
float s = 2.0*abs(fract(0.5*(x+0.5*floor(2.0*y-0.5)))-0.5); // Finicky mapping
float t = 2.0*abs(fract(y)-0.5);
float bricksx = aastep(0.475, s) - aastep(0.525, s); // s is stretched 2x
float bricksy = aastep(0.45, t) - aastep(0.55, t);
float bricks = max(bricksx, bricksy);
```

There *is* a less cumbersome way to fix this – a solution that works for many other patterns with similar problems without requiring as much special attention. This, however, requires insight into how ***aastep*** really works, so let's move on and return to this pattern in the next chapter.

A note on precision

The shaders in this chapter are all using either ***fract*** or ***mod*** to compute the local coordinates of the tiling. This has an inherent potential problem which is not obvious: we are taking the fractional part of coordinates which are represented by floating point numbers. These numbers have a decreasing amount of bits of

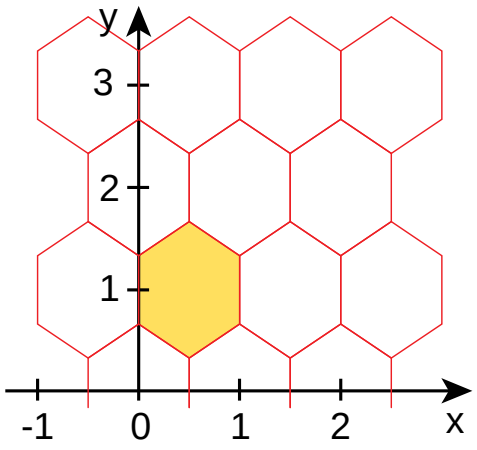
fractional precision as they become larger. Mathematically speaking, the tiling is infinite, but in a floating point representation, the local mapping coordinates lose their fractional precision as we move away from the origin in the coordinate system for the overall surface mapping. When coordinates reach one thousand, we have lost ten bits of fractional precision, which shouldn't be noticeable in most cases, but when we get to one million, there are almost no fractional bits left at all. Because of this, many procedural patterns start to look wonky when you get very far from the origin for the surface coordinates. A common cause of these errors is an animated translation that is left running for too long.

We will return to this and dig deeper into problems with floating point precision in chapter 15 , "Scale". For now, just keep in mind that we are not doing exact math. We are working with numbers of limited precision, and it can cause problems.

Hexagonal tiling

Another kind of tiling is *hexagonal tiling*, and we will cover that next, but first, a word of caution is in order.

Please don't be alarmed by the relative complexity of this compared to the other tilings! A hexagonal tiling is not ideally suited for mapping to a Cartesian (x, y) coordinate system, and it will take some work. The mapping shown below is not obvious. Quite a lot of effort went into creating it, and we are really just presenting the end result of that process. A hexagonal tiling can be quite useful, though, so please remember that it's an option, and that there's code for it here. There are several other useful algorithms for creating a hexagonal grid, and you might find another one to be your favorite. The one presented here was chosen because it's both efficient and reasonably easy to explain. Not easy, but *reasonably* easy.



One variant of hexagonal tiling

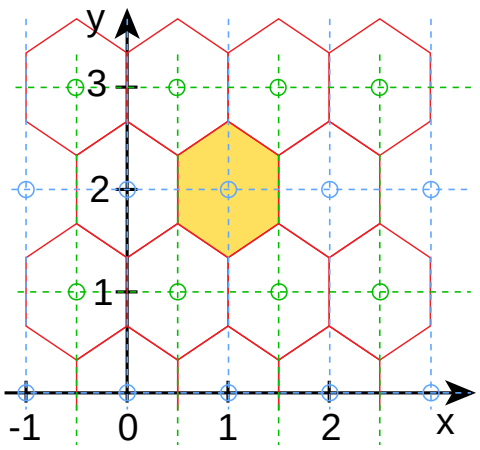
Expressing a hexagonal tiling in shader code is a bit tricky. Quite *literally* tricky, because there are certain tricks to doing it in an efficient manner. First, note the integer coordinates in the diagram above, and our choice of scaling. This tiling is not built from *regular* hexagons – they are slightly taller, by a factor of $2/\sqrt{3} \approx 1.1547$. This particular choice of grid makes the code for the tiling less complicated, and more readable. To create a tiling of regular hexagons from this slightly stretched-out version, the mapping coordinates can simply be scaled in the y direction by the factor $2/\sqrt{3}$, effectively squashing the hexagons back into equilateral proportions.

With the tiling in this orientation, consecutive horizontal rows of hexagons are offset from each other, and every second row is the same. It can be expressed as either a staggered tiling or a skewed tiling. The hexagons as such can be straightforwardly defined as the regions where one particular grid point is closest. Remember that we are not aiming for a function to locate the boundary lines of the tiling and draw them – we want to compute *local coordinates* within each hexagon that repeat periodically. We can achieve this by first finding, for an arbitrary point in the plane, the position of the closest hexagon midpoint.

We will choose to map the grid to a staggered tiling. (Treating it as a skewed tiling can be slighty more efficient, but it's also rather messy.) In fact, we will go one step further and treat the staggered tiling as two overlapping rectangular tilings:

The blue grid has its vertices at integer coordinates in x and at even integers in y.

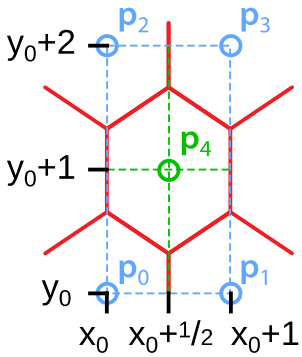
The green grid has its vertices half-way between integer coordinates in x and at odd integers in y.



A staggered grid regarded as two rectangular grids with an offset between them

Now, for an arbitrary point $\bar{p} = (x, y)$ in the plane, we want to determine which one of the hexagon midpoints is closest. It's enough if we work out how to do this for one cell of either the blue or the green grid, because the algorithm is the same for

all other cells. Looking at one cell of the blue grid, we see that there are five possible candidates for which grid point is closest to a point within the cell.

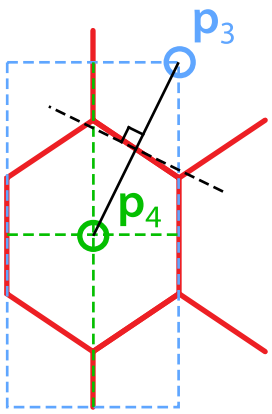


One cell of the blue grid, with numbered hexagon midpoints $\vec{p}_i$

For half of the area of the cell, $\bar{p}_4$ is closest, but towards each of the corners, either $\bar{p}_0$, $\bar{p}_1$, $\bar{p}_2$ or $\bar{p}_3$ is closer. We could simply compute the distances to all five candidates and pick the nearest one, but the most efficient way to sort this out is to first determine which corner we are in, and then compare distances only to $\bar{p}_4$ and *one* other candidate. For a point (x, y) inside the cell, the lower left vertex is at $(x_0, y_0) = (floor(x), 2\,floor(y/2))$, and the other vertices are at fixed offsets from it. If $x < x_0 + 0.5$, we are in the left half of the cell, and if $y < y_0 + 0.5$, we are in the lower half. Performing both these tests determines which of the four corners might be closer than $\bar{p}_4$. Sign tests are particularly efficient, because they don't need to perform a subtraction – they just look at the sign bit to determine if a number is negative. Because of this, we pick a corner based on the vector from $\bar{p}_4$ to the current point $\bar{p} = (x, y)$.

By performing two simple comparisons, we have now eliminated all but two candidates for the closest point. We could simply compute the distances to both and compare them, which would be a lot less work than doing it for all five points.

However, the boundary where the Euclidean distance switches from being closer to a corner than to $\bar{p}_4$ is not quite what we want to make a tiling of stretched-out regular hexagons. Looking closely at the situation, we see that the line of equal distances in our stretched-out grid is at a slight angle to our desired cell boundary, and our cells would be slightly too short and stout.



Our desired boundary (red) and the Euclidean distance boundary (dotted black)

This might not be a big issue – in fact, it could even be what we want in some circumstances. We could also fix it by simply computing the distance in a non-uniformly scaled coordinate system where we squash the y component by a factor of $\sqrt{(3)}/2 \approx 0.866$. This will yield the same metric as if we had computed the Euclidean distances in the regular, non-stretched grid, and make the cell boundaries match the red lines in our graphs. This would be a perfectly acceptable solution.

However, noting that the boundary is a line, we can use the implicit equation for that line to determine which side of the line our point is on. This can be done by determining the sign of a simple linear polynomial. It's a little bit of a hassle to work out the equation for the line, but it boils down to:

$$(\bar{p} - \bar{p}_4) \cdot (\pm 2/3, \pm 1) - 2/3 = 0$$

where the dot denotes a scalar product, and the plus/minus signs for the components of the second vector differ depending on which corner we are in. If this expression is negative, $\bar{p}$ is closer to $\bar{p}_4$ than to the corner at $\bar{p}_4 + (\pm 0.5, \pm 1)$.

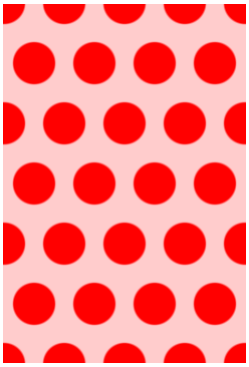
We're done! Let's express the entire algorithm as program code:

```
// Find the closest point from p in a hexagonal grid.
// The grid is not quite regular. Scale p.y by 2.0/sqrt(3.0) before
// the function call if you prefer a regular hex grid,
// with grid points at pesky irrational coordinates in y.

vec2 hextiling( vec2 p ) {
	// Lower left vertex p0 of local integer-aligned 1x2 rectangular cell
	vec2 p0 = vec2( floor( p.x ), 2.0 * floor( p.y * 0.5 ));
	// Midpoint p4 of that cell
	vec2 p4 = p0 + vec2(0.5, 1.0);
	// Vector from midpoint to p (local cell coordinates)
	vec2 v4 = p – p4;
	// Set px to the closest corner, based on signs of v4.x and v4.y
	vec2 dx = vec2( ( v4.x < 0.0 ? -0.5 : 0.5 ), ( v4.y < 0.0 ? -1.0 : 1.0 ) );
	vec2 px = p4 + dx;
	vec2 vx = p - px; // Vector from corner to p ( also: vx = v4 - dx )
	// Determine whether the corner px or the center p4 is closer.
	// The vector ex is the normal to the decision boundary.
	vec2 ex = vec2( ( v4.x < 0.0 ? -2.0/3.0 : 2.0/3.0 ), dx.y );
	// Use the line equation for points half-way between p4 and px
	float d = dot(v4, ex) - 2.0/3.0; // If d is negative, p4 is closer
	// Return the closest grid point
	return ( d < 0 ? p4 : px );
}
```

Now, this might seem like a lot of work just to figure out a tiling – we haven't even created a pattern with it yet! However, we really went overboard with the amount of comments, and the code doesn't contain a whole lot of computations. Counting multiplications with powers of 2 as additions, because that's what they end up being in floating point arithmetic, the entire function above amounts to fifteen additions/subtractions, two multiplications (the ***dot***), two ***floor*** operations and three conditional selections based on sign bits. (Incidentally, one of the multiplications is by 1 or −1, which is just a conditional sign flip, but we won't bother trying to speed that up.) Even a low-end GPU can easily handle much more than this in a fragment shader. The code was difficult to *write*, but it's easy for a GPU to *execute*.

Now, let's use the hexagonal tiling to create a pattern! A simple polka-dot pattern would be similar to what we did before with the rectangular grid, only with using the local coordinates for the hexagonal grid to compute the distance to the midpoint of the dot:

```
// Polka-dot pattern in a hexagonal tiling.
// The range of R is 0.0 (no dots) to 2.0/3.0 (dots cover
// the plane). The dots will start to overlap if R>0.5.

float hexpolkadots( vec2 p, float R ) {
        // Compute the distance to the nearest gridpoint of
        // a hexagonal grid, and create a circle around it
        return 1.0 – step( R, length( p - hextiling( p ) );
}
```



Polka-dots in a hexagonal grid arrangement

Yes, once we have the tiling worked out, it really is that simple. Again, apologies for a rather complicated algorithm here, but it *is* hairy to express hexagonal tiling in a Cartesian grid – at least if you want to do it in a reasonably efficient manner.

Hexagonal tiling will be revisited in the chapter “Noises”, because a variant of it is used for 2-D *simplex noise*. Fun times ahead.

6 Anti-aliasing

In the previous chapter, we mentioned briefly that a well written procedural shader can perform its own anti-aliasing. This deserves repeating, with an added strong recommendation to actually do it. It even deserves its own little framed box:

A well written procedural shader can,
and *should*, perform its own anti-aliasing.

(Apologies if this comes across as unnecessary lecturing. This is a textbook, and I *am* lecturing, but I don't want to annoy the reader. Well, not more than necessary.)

It cannot be overstated that in a procedural shader, aliasing *can* be avoided, and *should* be avoided in all situations where visual quality matters. Particularly when people write shaders without proper care, aliasing in surface patterns can be horrendous. This is acceptable for beginners and creative experimentation, but for serious production there is simply no excuse for it.

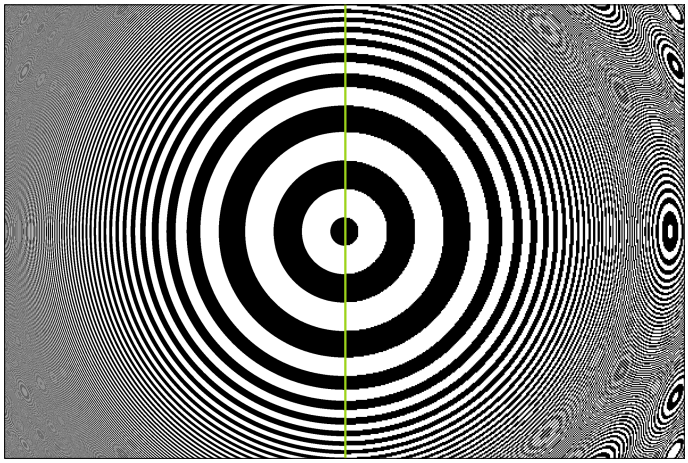
There's a fitting analogy from cooking: "There is no excuse for under-cooked rice." Rice can be easily checked if it's ready before you serve it. In the same manner, a procedural shader can be easily checked for aliasing problems before you release it. The analogy isn't perfect, because under-cooked rice can simply be left to boil until it's ready, while aliasing in a procedural shader can take a lot of hard work to fix. Nevertheless, it's a good analogy to keep in mind. Under-cooked rice can ruin an otherwise excellent meal, and aliasing can ruin an otherwise excellent image.

Types of aliasing

In the previous chapter, we mentioned jagged edges as one kind of aliasing. Another kind is interference between the sampling distance and the underlying pattern, referred to as *moiré*. Both of these are on prominent display in the test pattern on the next page.

As you can see, untamed Moiré remains strong even with a very high sampling frequency, even when individual pixels are too small to be seen by a human observer. This is because we are *undersampling* a pattern that is too detailed to be represented as pixels in the chosen resolution. High frequencies (small details) are

misinterpreted as lower frequencies (larger details). Sometimes the Moiré patterns have a very low frequency, even if the sampling frequency is high.



A pattern of concentric circles with the pattern growing denser with distance, at two different sampling frequencies (left: ~600 dpi, right: ~150 dpi).

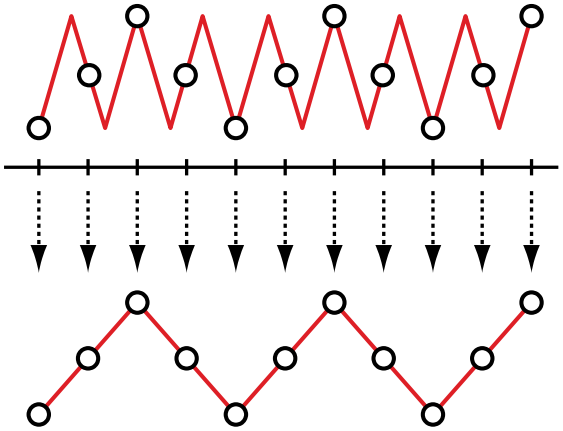
In animations, aliasing can also cause strong flickering and “stroboscope effects”, causing confusion about speed and direction of motion. Unfortunately, we have no good way of demonstrating this in print, but it can be thought of as the temporal sampling equivalent to Moiré: the frame rate is insufficient to properly reflect changes in the image that happen at a higher rate, and the sampled sequence can end up displaying grossly incorrect motion information.

Reasons for aliasing

Now, before we dive into exactly how anti-aliasing is performed, it’s time to take a step back and briefly explain what “aliasing” actually *is*, and where the name comes from. It’s always good to know exactly what the problem is before trying to find a solution.

The principle is more clearly explained in a 1-D plot. According to the famous *sampling theorem*, which is often named after Nyquist but should actually be attributed to Shannon (or, rather, to several pioneers in signal processing, including but not necessarily limited to Borel, Ogura, Whittaker, Kotelnikov, Raabe, Shannon, Weston and Someya, but I digress), we need to sample a periodic signal strictly more than twice per period, or else there is no way to reconstruct the original signal from the samples. Attempts at reconstruction will wrongly interpret the samples as coming from a signal with a lower frequency. In other words, the

insufficiently sampled high frequency signal shows up in the reconstruction as an *alias* with a different, lower frequency. This is the reason why it's called "aliasing".



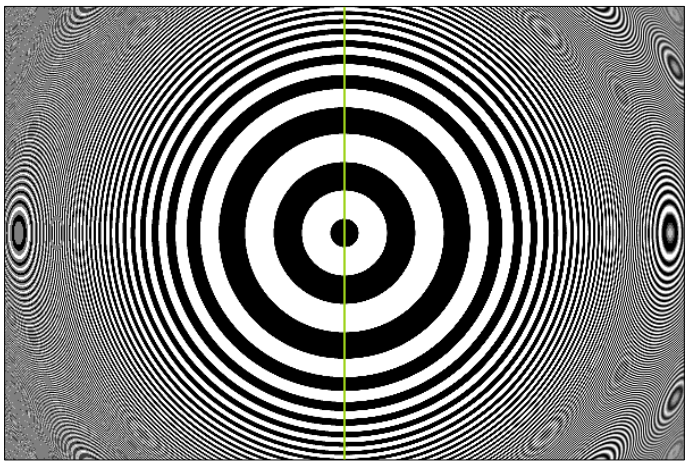
Undersampling of a signal, with frequency aliasing as a result

Jaggies from crisp edges have the same cause. A sharp transition in color is a discontinuity, but signal processing theory assumes that all signals being sampled are continuous (formally *band-limited*, but continuity is a required condition for that). Therefore, a pattern with razor-sharp edges can't be point sampled without creating erroneous artifacts – the "jaggies" – which are not present in the underlying signal.

Classic remedies

To reduce aliasing, shaders can be *multi-sampled* or *supersampled*. The exact definitions of those terms vary somewhat depending on the context, but they both amount to *oversampling* the pattern: taking samples at more than one point for each pixel and averaging the result. This is an approximation of *area sampling*, which is one way of handling a signal that isn't band limited. It can also be thought of as a kind of pre-filtering to blur the pattern before point sampling.

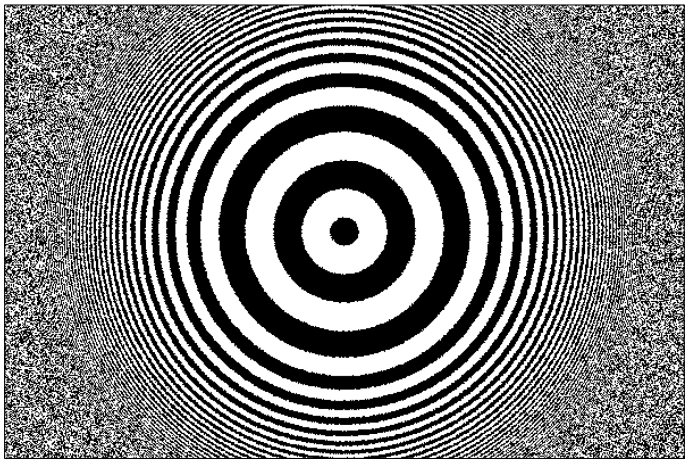
Oversampling, however, is usually not the best option for a procedural pattern. If the shader performs a lot of computations, executing it more than once for each pixel is not a great idea. Multi-sampling also doesn't eliminate aliasing – it just reduces it, and the quality improvement (expressed as standard deviation from the desired value) scales only with the square root of the number of samples. It *is* a remedy, but not a great one, and it's costly.



Multisampling. ***Left:*** 2×2 *samples per pixel.* ***Right:*** 4×4 *samples per pixel.*

As you can see in the image above, jagged edges are definitely improved by multisampling. Moiré, however, not so much. Some defects get pushed to higher frequencies, but the moiré pattern remains strong where the frequency of the pattern interferes with the sampling frequency in an unpredictable manner.

Another way of hiding aliasing is to sample each pixel not at its exact center, but in a pseudo-random point inside the pixel area. This is called *jittered sampling*, and while it doesn't actually *remove* the aliasing, it breaks up regular interference patterns, which changes regular stair-step jaggies and moiré patterns into fine-grained noise. To a human observer, noise is usually more tolerable than distinct, repetitive pattern artifacts. Not always, but usually.

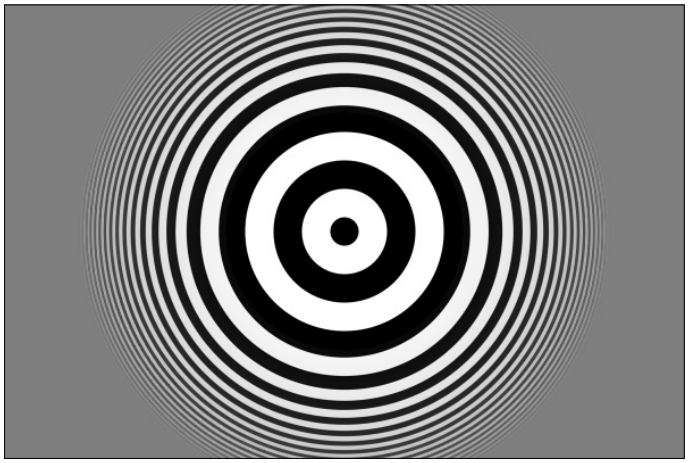


Jittered sampling

Software renderers commonly employ some combination of jittered sampling and multi-sampling, usually performed in an adaptive manner around object edges and at regions of sharp contrast. In animations, the samples can also be spread over time to simulate motion blur, to reduce flickering and stroboscope artifacts. This is referred to as *distributed sampling*. GPU rendering can do similar things with multi-sampling and averaging across frames, but it comes at a great computational cost, and in an interactive rendering situation, the options for spending more work on particularly problematic pixels in an adaptive manner are limited.

Procedural remedies

Fortunately, none of these workarounds are needed for procedural patterns, because a well written procedural shader (as you should know by now, right?) can *perform its own anti-aliasing.* The image below is from a shader that does that.



A self-anti-aliasing procedural pattern

The jagged edges from the hard transitions between white and black are now explicitly smoothed out by the shader code, and where moiré would start to kick in, the pattern is gradually faded to its average color, in this case medium gray. This anti-aliasing requires only one sample per pixel, and it's performed at a very small additional computational cost compared to the badly aliasing shader in the first image of this chapter.

This kind of "smart" anti-aliasing internal to a shader is often referred to as *analytic anti-aliasing*. The term is used broadly to refer to any kind of anti-aliasing performed by the shader. The name suggest that it would involve a lot of math, but in many cases it can be quite simple. In this case, it's all performed by means of the

“magic” functions ***aastep*** and ***fwidth*** which were briefly mentioned in the previous chapter, with some premature examples of to use them.

Now it’s time to explain how those functions actually *work*.

Auto-derivatives

One key reason why anti-aliasing is reasonably easy in shaders is the existence of a subsystem in the renderer called *automatic derivatives*, or *auto-derivatives* for short. They work more or less the same for both software and hardware rendering, but our explanation will focus on GPU rendering.

There are two functions in GLSL called ***dFdx*** and ***dFdy***. Each will try its best to compute a partial derivative of its argument, regardless of that argument’s complexity. As you can probably guess from their names, ***dFdx*** computes the partial derivative in the x direction, and ***dFdy*** does the same in the y direction.

Now, how can a shader function compute a derivative of an arbitrary expression? Well, it’s not actually taking the true, analytical derivative of anything – it’s computing an approximation by a finite difference, and the difference is computed between the value of the expression at the current pixel and at neighboring pixels in the x and y directions, respectively. The “derivative” is really only a subtraction.

But wait a minute! A shader program shouldn’t even be aware of what neighboring pixels are doing, much less have access to their results? Well, that’s what causes this feature to be hidden inside built-in functions. Access to the result of evaluating the same expression in neighboring pixels (fragments) is *implicit*. The rendering algorithm computes several fragments in parallel, and when a “derivative” function is encountered in the instruction stream, the numerical value of the expression in the argument is shared between two threads, and the finite difference is computed. What looks like magic is in fact just a few subtractions performed behind the scenes on the programmer’s indirect request.

Are these steps for the finite difference taken in the positive or negative directions in pixel coordinates x and y? What about pixels that don’t have a neighbor in that direction that runs the same shader program? Or worse, pixels that have no neighbors at all executing the same shader? The short answers to those questions are “it depends” and “it’s complicated”.

Slightly longer answers are that some GPUs have options to set preferences for accurate or fast auto derivatives, and “fast” can mean that the same value for the derivative is shared between two neighboring fragments, which effectively makes the derivative functions sub-sampled to half the fragment resolution. Some GPUs do it only like that, while some have the option to use a consistent direction for

derivatives, with some reduction in performance. As for what a GPU does when no neighbor is available to compute a derivative, the auto-derivative functions are not actually guaranteed to return sensible results when the shader operates on isolated pixels or long, narrow objects that are rendered as only one pixel wide. In such cases, the results are “undefined” and most likely useless.

Because of how the auto-derivatives work, making a simple subtraction between values from only two neighboring fragments, they can only compute first order derivatives. Taking the derivative of a derivative doesn’t give you the second derivative. A call to ***dFdx(dFdx(…))*** or even ***dFdx(dFdy(…))*** is likely to return either garbage or zero. The syntax is formally allowed, but the result is “undefined”, meaning you shouldn’t do it. If you need second order derivatives, you need to compute the analytical derivative of your function and implement it in shader code. It’s actually not as difficult as it might sound, and it can be very useful, but let’s not dwell on that here. We will return to it in chapter 10 ,“Noises”.

Now that we have the auto-derivatives explained, what are they used for? The most common use is indirect, and contained in the built-in function ***fwidth***. In the language specification, ***fwidth*** is defined in terms of its equivalent GLSL code:

```
float fwidth(float value) {
	return abs( dFdx( value ) ) + abs( dFdy( value ) );
}
```

That’s all there is to it. This is an approximation of the length of the gradient vector of the argument value, in screen space (pixel) coordinates. The short-cut using two absolute values instead of two multiplications and a square root to compute the length of a 2-D vector saves on computations. (In floating point arithmetic, adding two absolute values is basically no work at all except for the addition – you just ignore the sign bits for both terms.)

The “quick and dirty” length computation made by ***fwidth*** is correct for a gradient along the x or y direction, but over-estimates its length with up to a factor of $\sqrt{2}$ in other directions. In many cases, this error doesn’t matter much. However, if you want better accuracy, you should consider implementing your own version of ***fwidth***, like this:

```
float better_fwidth(float value) {
	return length( vec2( dFdx( value ), ( dFdy( value ) ) );
}
```

A reasonably competent modern GPU has no problems coping with the extra work, and in a non-trivial procedural shader it probably won’t add much to the total

anyway. However, if you are desperate for clock cycles and want absolute maximum performance, the built-in ***fwidth*** is useful and often good enough.

Soft edges

So much for how ***fwidth*** is computed. But how can we *use* it? The name of the function hints at the most straightforward use: to compute a suitable "filter width" for a transition ramp to smooth out edges. For an arbitrary expression F, the length of its gradient, $\|\nabla F\| = \sqrt{(\partial F/\partial x)^2 + (\partial F/\partial y)^2}$ tells us the rate of change, in screen space, of the expression. Moving a distance of one pixel in screen space along the direction of the gradient will make the argument change by that much. Therefore, to create a one-pixel wide blend across a crisp edge created by a step function, we should replace the step with a smooth transition happening across an interval of length $\|\nabla F\|$. This is exactly what the ***aastep*** function does in OSL.

The most common use case of ***fwidth*** in GLSL is to create the equivalent of the ***aastep*** function, even if it's not always referred to by that name. In Renderman RSL it was called ***filterstep***, and some authors choose to name it something else. The name ***aastep*** seems to have stuck, though, and it's a good name.

The aastep function

We showed an implementation of ***aastep*** in GLSL in the previous chapter, but let's repeat it here.

```
// Anti-aliased step function in GLSL
float aastep( float edge, float value ) {
        float w = fwidth( value ) * 0.5;
        return smoothstep( edge – w, edge + w, value );
}
```

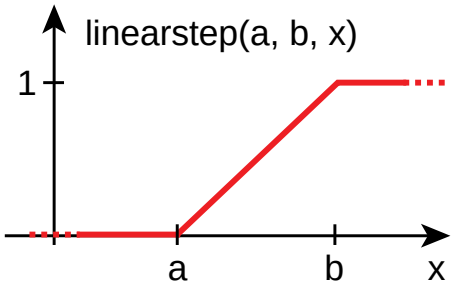
This is a straightforward implementation, but some tweaking can make it perform a little better. First, our use of ***fwidth*** makes the value for ***w*** anisotropic – it will depend on the angle of the gradient. This can be fixed by implementing our own version without the "speed cheat". Second, our step width is a bit too short when using ***smoothstep*** for the blending. As you may recall, $smoothstep(a, b, x)$ has zero slope at $x = a$ and $x = b$. This means that not much happens towards either end, and half-way between a and b the function has a steeper slope than a linear ramp. To fix that, we can either use a linear ramp instead, or make our ***w*** somewhat larger.

In OSL, there is a function ***linearstep(a, b, x)*** which works similarly to ***smoothstep*** but has a linear ramp from 0 to 1 between a and b.

$$linearstep(a,x)=\begin{cases} 0 & for\ x \le a \\ ramp & for\ a<x<b \\ 1 & for\ x \ge b \end{cases}$$

where *ramp* is a linear ramp:

$$ramp=(x-a)/(b-a)$$



*The **linearstep** function*

Other shading languages don't have a built-in ***linearstep***, but it's easy to make one. In GLSL, it would look like this:

```
float linearstep( float a, float b, float x ) {
        float ramp = (x – a) / (b – a);
        return clamp( ramp, 0.0, 1.0 );
}
```

The ***clamp*** function is used a lot for other purposes in GPU shading, in particular in this role where it clamps a value to the range $[0,1]$. Its function can be described by this equivalent code:

```
float clamp( float x, float minval, float maxval ) {
        if( x < minval )
                return minval;
        else if ( x > maxval )
                return maxval;
        else return x;
}
```

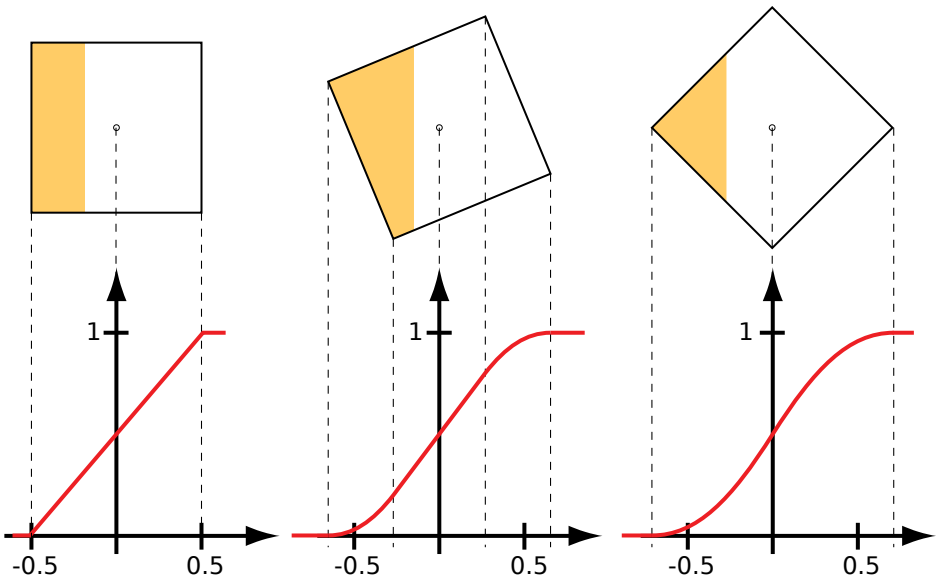
The ***clamp*** function in GLSL is most likely hardware accelerated, and easy for the compiler to optimize, so you are better off using the built-in version for speed rather than rolling your own version of it. The built-in version also has several overloaded versions for vectors and integer types, which is convenient.

However, a linear ramp is not obviously the right choice for our anti-aliasing. What we usually strive to do in anti-aliasing is to make our *point sampling* mimic *area sampling*. Instead of deciding the value of a pixel based only on whether the pixel center is inside or outside an edge, we want the pixel value to be determined by *how much* of the pixel area that falls inside the edge, as a function of the position and orientation of the edge.

An approximation of area sampling can be made in several ways. One rather rough approximation is to assume that pixels are circles with fuzzy edges. That is obviously not the case, but in signal processing terms, it's a low-pass filtering followed by point sampling, and it works reasonably well – at least it's better than point sampling. However, we can look at least a little closer at the problem.

With a regular, square sampling grid, which is the overwhelmingly most common choice, pixels areas cover a little square. The intersection between that square and an edge as a function of edge position is reasonably easy to compute. Not trivial, but reasonably easy. The illustration on the next page shows what it looks like for three different edge orientations.

For axis-aligned edges, the area coverage as a function of edge position is a linear ramp of width 1 pixel. For slanted edges, however, the area coverage starts out and ends with parabolas that taper off to zero slope at the endpoints. For most directions, this looks rather more like a ***smoothstep*** function.



Area coverage as a function of the position of an edge relative to the pixel center. Left to right: axis-aligned edge, 22.5 degree edge, 45 degree edge.

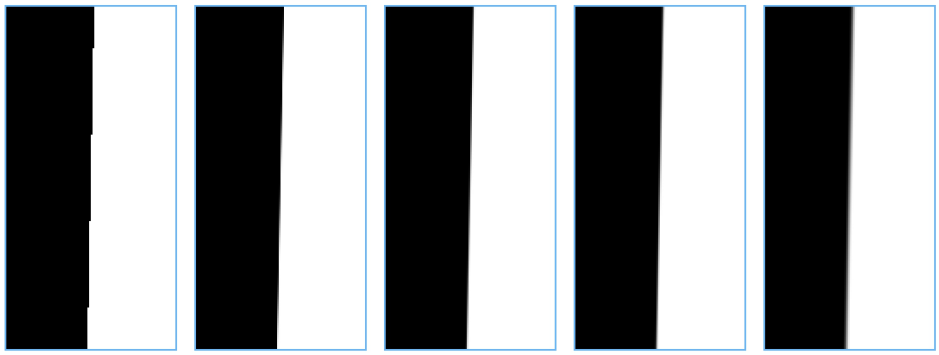
Whether to use ***linearstep*** or ***smoothstep*** is largely a matter of taste. Of course, we could compute a more accurate area coverage that depends on the angle of the gradient, but that wold be an extreme overkill in most cases. Our choice is to use ***smoothstep*** and make the width of the step somewhat larger than the length of the gradient. The precise width is not all that important, because we're not doing this in

an exact manner anyway. Formally, even if we were to compute the true area coverage, we would need to take into account both the colors to either side of the edge and the non-linear "gamma curve" of the display device to compute a photometrically correct blend at the edge. We will fudge it and simply use a step that is 50% wider than $\|\nabla F\|$:

```
// Better anti-aliased step function in GLSL
float aastep( float edge, float value ) {
        float w = 0.75 * length( vec2( ( dFdx( value ), dFdy( value ) ) ) );
        return smoothstep( edge – w, edge + w, value );
}
```

Note that we still opted for using the true, isotropic gradient length. The built-in ***fwidth*** is quite crude, and the execution time it saves is usually insignificant. However, unless you notice any visual differences, you might prefer to go with the built-in version for a slight speedup.

There's nothing magic about the factor 0.75 – it just makes a ramp for ***smoothstep*** that is 1.5 times wider than $\|\nabla F\|$. For a specific display and a specific pattern, you might want to make that factor slightly smaller to create more crisp edges, or slightly larger to make softer edges. The visual test case to determine what looks good is a slightly sloping edge with a high contrast, like this:



*Edges using **smoothstep** with different step widths in proportion to $\|\nabla F\|$.*
***From left to right:** 0.0, 1.0, 1.5, 2.0, 4.0.*

Neither the left nor the right extreme would be a good choice, but which one looks "best" of the three middle ones depends on both the slope and the contrast of the edge. It also differs between display devices, and it can even be a matter of taste and artistic expression. A factor of 1.5 seems to be adequate for most uses, but don't be afraid to change it. This is about eyeballing, it's not exact math.

A final note on the ***aastep*** function, whether it's the built-in version in OSL or the roll-your-own version we just did in GLSL: the argument for the threshold should be constant, or at least slowly varying compared to the value being compared to that threshold. If both vary at around the same scale, the auto-derivative inside ***aastep*** will fail to take into consideration the variation of the threshold and make edges either too crisp or too soft, seemingly at random, due to the non-zero gradient of the threshold. If both values vary rapidly, we can compute their difference and threshold against 0.0 instead:

```
edge = aastep( varyingvalue1, varyingvalue2); //  Wrong
edge = aastep( 0.0, varyingvalue2-varyingvalue1); // Right
```

Using the plain ***step*** function, the two lines above would do exactly the same thing, but our own implementation of ***aastep*** treats the two arguments differently.

We could, of course, change our implementation of ***aastep*** to instead compute the gradient of the difference between the arguments. All it takes is to move the extra subtraction above into the function:

```
// More robust anti-aliased step function in GLSL
float aastep( float edge, float value ) {
        float F = value – edge;
        float w = 0.75 * length( vec2( ( dFdx( F ), dFdy( F ) ) );
        return smoothstep( – w, w, F );
}
```

Looking closely at the code, this is actually slightly *less* work than the version above, because the extra subtraction removes two additions for computing the arguments to ***smoothstep***. In some cases you may want to use the same function to compute several ***aastep*** transitions with different thresholds, and in that case the compiler won't be able optimize your code by re-using the same auto-derivatives, but keep in mind that each auto-derivative is really just a single subtraction. There is usually no clear downside to using this modified version of ***aastep*** instead of one that requires the threshold to be constant.

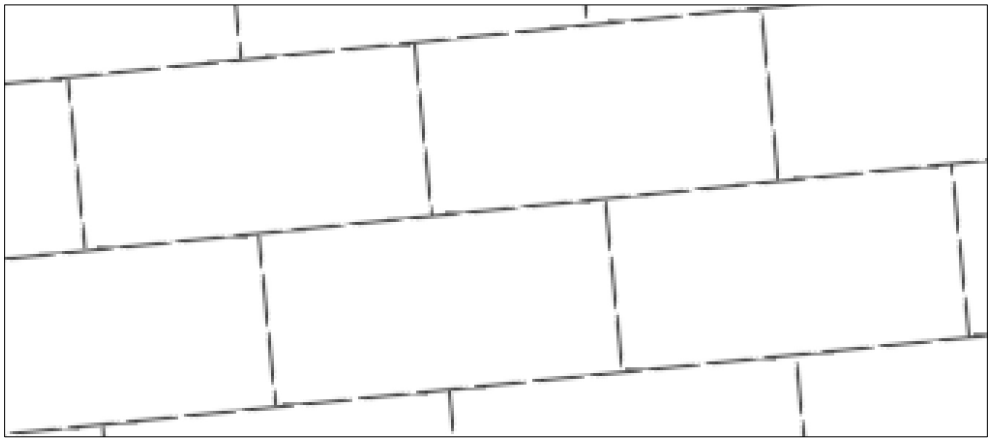
Brick pattern, revisited

In the previous chapter, a remaining problem with the brick pattern was fixed with a mapping that took quite a bit of thinking to create, and we mentioned that there is another, more general way to fix that kind of problems. Now you know enough to understand how.

Let's return to the penultimate version of the "bricks" pattern, the one where the gaps between the bricks fall on the *peaks* in the "triangle wave" mapping. The problem is with artifacts appearing at grid lines as they get very thin. This happens because the auto-derivatives are frequently wrong near sharp extremes, and the peaks of our triangle function are really pointy (the peaks have infinite curvature). The second-to-last shader we used to render the brickwork pattern was:

```
float bricks( vec2 p ) {
        vec2 q = fract( p + vec2( 0.5 * floor( p.y ), 0.0 ) ); // Staggered grid
        q = 2.0 * abs( q – 0.5 ); // Change from sawtooth to triangle
        float brickx = 1.0 – aastep( 0.99, q.x ); // vertical gaps
        float bricky = 1.0 – aastep( 0.98, q.y ); // horizontal gaps
        return min( brickx, bricky ); // If either is 0, return 0
}
```

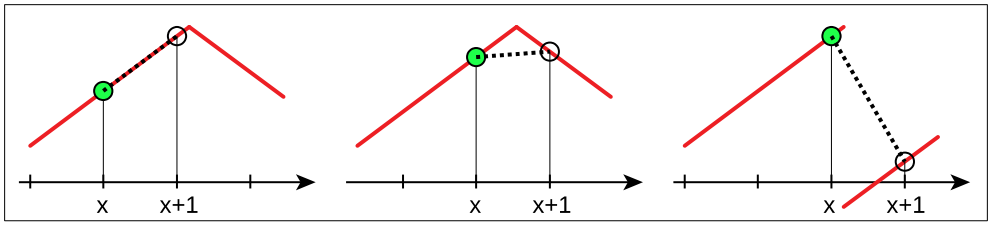
Artifacts appear when the gaps between bricks become narrow. Their exact appearance depends on how auto derivatives are computed, and that can differ somewhat between platforms, but it typically looks like this:



Anti-aliasing artifacts at thin lines, due to incorrect auto-derivatives

The dropouts in the lines look terrible, and they depend strongly on the exact position of the pattern relative to the pixel grid, meaning that they appear random and flicker badly in animations. The problem here is that we're asking ***aastep*** to compute auto-derivatives of a triangle wave very close to its peaks. Looking at it in one dimension, it's easy to see what goes wrong.

The figure on the next page shows the case for outright discontinuities as well, to demonstrate that we are already dodging some big errors by using a continuous mapping function.



Auto-derivative at x (green circle) computed as finite difference in fragment space.
***Left:** correct result. **Middle:** estimated slope too small, yielding incorrect AA.*
***Right:** at discontinuities, the estimate can be much too large.*

What we want for the anti-aliasing to work is the correct derivative even near the peaks. We only need the length of the gradient, which depends only on the absolute values of the partial derivatives, and those are in fact *constant* everywhere for this mapping. The ramps of the triangle wave have alternating positive and negative slope, but the absolute rate of change is the same everywhere. Furthermore, that constant derivative is the same as for the *original* mapping coordinates, the ones we had before creating our local coordinates for the grid. Well, *almost* the same – we scaled the function by 2.0, meaning that we scaled its derivative as well.

Using this information, we could insert the code of our anti-aliasing ***aastep*** function into the shader and modify it to compute the step width from the un-modified mapping coordinates. We could also implement a hacked version of the function where we supply a separate function for the derivative computations. It will be somewhat of a hack either way, but the "hacked aastep" approach makes the shader code less cluttered and gives us a re-usable helper function:

```
// Hacked aastep function to compute correct derivatives at problematic points
float aaxstep( float threshold, float value, float value_for_fwidth ) {
	float w = fwidth( value_for_fwidth );
	return smoothstep( threshold - w, threshold + w, value );
}
```

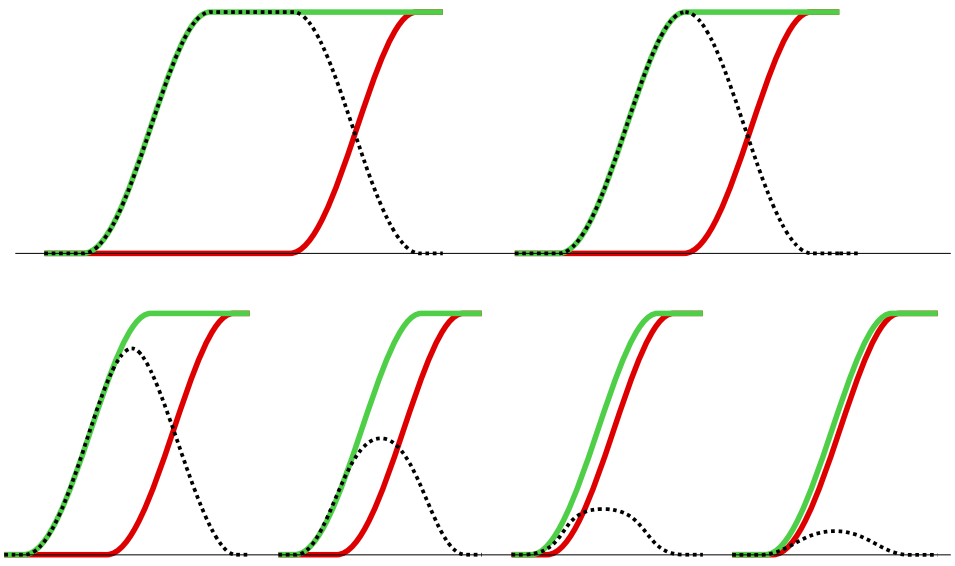
Using this modified function, the shader code would be:

```
// Brick pattern that antialiases correctly even when the gaps are very thin
float nicebricks( vec2 p ) {
	vec2 q = fract( p + vec2( 0.5 * floor( p.y ), 0.0 ) ); // Staggered grid
	q = 2.0 * abs( q - 0.5 ); // Change from sawtooth to triangle
	float brickx = 1.0 - aaxstep( 0.995, q.x, 2.0*p.x ); // vertical gaps
	float bricky = 1.0 - aaxstep( 0.99, q.y, 2.0*p.y ); // horizontal gaps
	return min( brickx, bricky ); // If either is 0, return 0
}
```

We could have eliminated the three multiplications by 2.0 by leaving ***q*** in the range $[0, 0.5]$ and adjusting the thresholds, but it's not a big deal. It's convenient to have the local mapping for each grid cell span the range $[0, 1]$. Remember, also, that a multiplication with 2 in floating point math just means adding 1 to the exponent.

Now, this shader performs reasonably well even when the gaps between bricks become smaller than one pixel. Unfortunately, the lines don't fade away when they become *significantly* less than one pixel wide, and that's definitely a remaining flaw of this shader. We *can* fix that, so let's do it!

Remember the "double edge" thresholding method from the grid line example? That's a good way to make very narrow lines fade away. What we are doing is to take the difference between two ***smoothstep*** ramps, both with the same width, but offset from each other. When they are far apart, the result is obvious: we get one ramp up and one ramp down, with a plateau in between. As they get close together, however, the result becomes different: we get a lower peak, but its extent in the lateral direction stays more or less the same.



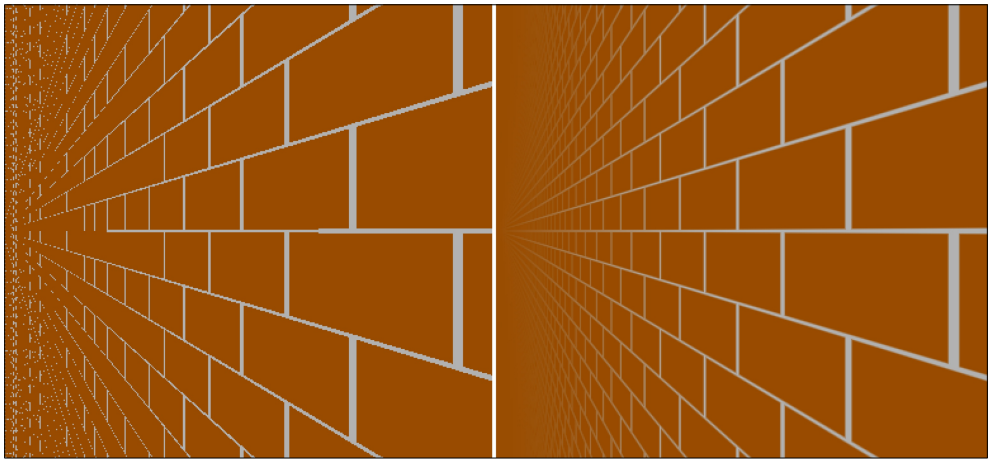
*Green and red lines: two identical **smoothstep** ramps with decreasing offset. Black line: the difference between them (green minus red).*

When the two smooth ramps move very close together, the spatial extent of their difference remains almost the same (their step width plus the small offset), but its amplitude drops linearly with the offset distance. This is similar to what we would get if we area sampled a very thin line passing through a pixel anywhere within its

bounds. It's not *exactly* right, but with a carefully tuned step width for the two ***smoothstep*** ramps, it's good enough for most uses. Using this technique, very thin lines will automatically fade away with distance and blend into the background. What's more, the fade with distance is achieved by *the same code* that renders the line in close-ups, which is convenient.

Considering that we have now also made a "glitch-proof" version of ***aastep*** where we can supply a continuous function for the derivative computations but use any function with the same local slope for the thresholding, we can actually move all the way back to our raw ***fract*** coordinates with unidirectional slope and perform the thresholding symmetrically around the midpoint 0.5 of those ramps:

```
float verynicebricks( vec2 p ) {
	vec2 q = fract( p + vec2( 0.5 * floor( p.y ), 0.5 ) ); // Staggered grid
	float gw2 = 0.005; // Gap width/2, relative to brick height
	float brickx = aaxstep( 0.5-gw2, q.x, p.x ) - aaxstep( 0.5+gw2, q.x, p.x );
	float bricky = aaxstep( 0.5-gw2, q.y, p.y ) - aaxstep( 0.5+gw2, q.y, p.y );
	return max( brickx, bricky ); // If either is 1, return 1
}
```



Left: *badly aliasing bricks.* ***Right:*** *Our now quite robustly anti-aliased bricks.*

The image above shows our now very nice brick pattern, with the naive first attempt next to it for comparison. The difference in quality is quite dramatic.

When experimenting with shaders, it often feels like not much is happening. In those cases, it's usually a rewarding experience to compare what you have currently to what you had a few hours ago, or yesterday. The steps along the way may be small, but they add up to a lot over time. It's a good idea in any case to save at least some of the many iterations you make along the way, because sometimes

you get lost, find yourself stuck in a dead end, or come to think of a better solution, and then it's nice to be able to backtrack easily.

Looking closely at the code, we are actually instructing the GPU to compute the auto-derivatives of ***2.0*p.x*** and ***2.0*p.y*** *twice* – once for every call to ***aaxstep*** having them as the last argument. Because functions are inlined in GLSL, we can reasonably assume that this repeated expression will be caught by the compiler and that each derivative is computed only once in the generated code. Should this not be the case, remember that an auto-derivative is really only a subtraction, and it involves very little extra processing. Most of the work here is with the four ***smoothstep*** ramps.

There's a morale to this section. Without any consideration whatsoever to anti-aliasing, a certain pattern can be easy to create. With *proper* and *robust* anti-aliasing, it's often considerably less easy to create. However, it's still not magic, it just takes more thought and some testing. As already mentioned, a procedural shader is a *lot* more useful if it has analytic anti-aliasing done *right*, so this is always preferred. Anti-aliasing by post-processing will require a lot more work by the renderer, and in some cases there's no fix to throw in afterwards. Jagged edges can be smoothed out reasonably well by multi-sampling, at a considerable cost, but moiré and flicker can be near impossible to eliminate in post-processing.

As a final note, even though most of our examples focus on GLSL and GPU shading, it deserves mention that the software shading language OSL has variants of its built-in ***aastep*** function that allow you to supply your own analytical derivatives for the arguments. That feature is there for a reason. The problems related to aliasing, and many good solutions to eliminate it, are well known in software shading, and it deserves equal attention in hardware shading. Software shader programmers have paved the way for decades, and it's wise to learn from their examples, when applicable, when writing GPU shaders.

Frequency clamping

Until now, we have been focusing our anti-aliasing efforts on edges, to make them smooth instead of jagged. Another kind of aliasing is the *moiré* that was utterly destructive to the images at the beginning of this chapter. Admittedly, that test pattern was made specifically to cause moiré, but it isn't unrepresentative for real scenes. A surface shader should allow viewing at any distance. At some distance, the period of a repetitive pattern will come close to the pixel resolution, and beyond that point something must be done to avoid moiré-type aliasing.

In image-based texture mapping, this problem is routinely handled by the method known as *mipmapping*, which involves down-sampling every texture image to

successively lower resolutions. This can be performed either in advance, before upload, or assisted by GPU hardware during texture uploads. However, mipmapping requires *filtering*, and filtering requires access to neighboring points in the pattern. The top level of the mipmap stack is the average color of all pixels in the image. A procedural shader can't take that approach, because the texture isn't computed in its entirety, and usually not at regularly spaced sample points. The samples that happen to be computed aren't even stored as a separate data set.

This is not necessarily a disadvantage. What we *can* do in a procedural shader is to completely *avoid generating* patterns with frequencies that are too high, and replace them with the average value of the pattern. Just like with edge anti-aliasing, this can be thought of as a simulation of the result of an area sampling. In fact, for periodic patterns it's even better, because an area sampling will still have problems with interference – causing moiré – when the sampling frequency is close to half the pattern frequency, near the "Nyquist limit".

The size of a pixel

To determine how small details we can resolve in the image that is being rendered, we need to know the distance between pixel samples. This can be done by tracking all transformations and taking the viewport resolution into account, but the auto derivatives makes it a lot easier. For a 2-D surface mapping (s,t), we want to compute four partial derivatives: $\partial s/\partial x$, $\partial s/\partial y$, $\partial t/\partial x$ and $\partial t/\partial y$. A first approximation of the "size of a pixel" in shader space could be the maximum absolute value of these four, or the length of a 4-D vector with them as components. Neither is correct, but both are good enough.

```
vec4 diff = vec4( dFdx(s), dFdx(t), dFdy(s), dFdy(t) );
float pixelsize1 = length( diff );
vec4 absdiff = abs(diff);
float pixelsize2 = max( max( absdiff.x, absdiff.y ), max( absdiff.z, absdiff.w ) );
```

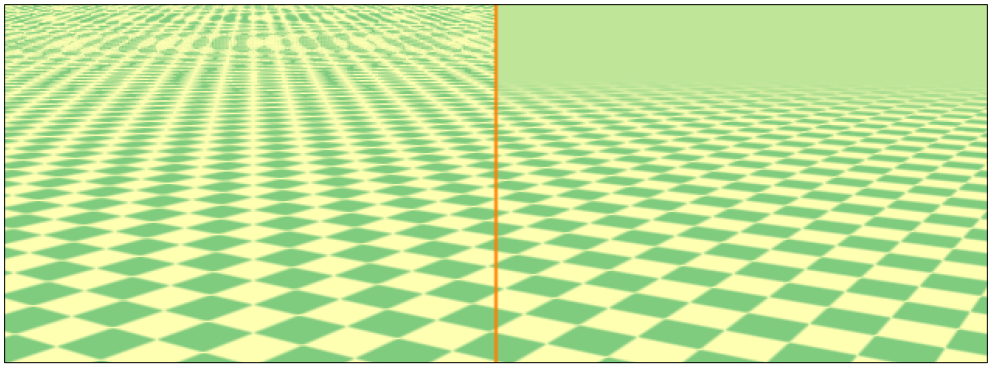
Note the nested ***max*** functions – GLSL can only select the maximum of two values at a time. OSL has a ***max*** function that takes an arbitrary number of arguments and returns the maximum in one go, but the built-in GLSL version maps to a hardware accelerated "compare-and-select" instruction and takes only two arguments.

Using this somewhat crude measure of the size of a pixel in shader space, we can check whether a periodic pattern we are about to generate would violate the sampling theorem and cause aliasing. That happens if the period of the pattern is smaller than twice the size of a pixel, with both measured in shader space. If that is the case, we just don't generate the pattern, but replace it with its average value.

Let's try this with our checkerboard pattern from the previous chapter!

```
// Anti-aliased checkerboard pattern with frequency clamping
float clampedcheckers( vec2 p ) {
	vec2 ramps = 2.0 * abs( fract( p ) - 0.5); // “Triangle waves”
	vec2 stripes = aastep( vec2(0.5, 0.5), ramps ); // 50% stripes
	float aacheckers = abs( stripes.x - stripes.y ); // “XOR” for overlaps
	// Frequency clamping to handle extreme minification
	vec4 diff = vec4( dFdx(p), dFdy(p) ); // Four derivatives in total
	float pitch = length(diff); // Roughly the pixel size in p space
	// A smoothstep with these bounds looks good in this particular case.
	return mix(aacheckers, 0.5, smoothstep(0.2, 0.35, pitch));
}
```

This shader removes moiré aliasing by making a gradual transition from a checkered pattern with values 0 and 1 to a constant value 0.5 when the pattern approaches the Nyquist limit. As you can see, it takes just a few lines of code, and it’s not much work for the GPU.



***Left:** checkered pattern with soft edges only. **Right:** Frequency clamping added.*

How to decide the extent and slope of the fade region isn’t an exact science. To determine what “looks OK” to a human observer, some trial and error is often required. What we want is to avoid visible and disturbing moiré while also keeping the pattern visible for as long as possible. The fade shouldn’t be too slow. With proper anti-aliasing of edges, moiré first appears at pattern frequencies only slightly lower than where it becomes absolutely disastrous to the rendering. It’s not a subtle effect, and it doesn’t sneak up on you gradually, it just pops into existence at a certain point. It may seem difficult to stomp it out in a discreet enough manner, but keep in mind that this is an inherent and unavoidable limitation with the pixel representation of the final image. There’s simply *no way* to make a pattern render nicely if it has details too small for the sampling to handle, and those details need to be taken out of the picture. (Pun intended – I have absolutely no shame.)

Efficient Level-of-Detail

The method of frequency clamping can be compared to level-of-detail (LOD) strategies for geometric models. If we know at render time that a certain content would not be visible, or even mess up the view, we choose another version of that content that renders better, and perhaps requires less work. Now, our version of the shader performs frequency clamping as a final step, fading the pattern to a constant color as we zoom out. There's nothing wrong with that as long as we are still seeing anything of the pattern, but once we reach the end of the fade curve, we are spending a lot of effort on computing a procedural pattern, which is then multiplied by zero and never shown. This is wasteful. It would be better if we could make an "early out" test and just return the constant value if the surface is too far away to have its pattern visible. A simple if-else conditional at the start could do that for us. Now, GPU shaders are executed in a massively parallel SIMD fashion, and the traditional "common wisdom" is to assume that *both* branches are being executed by all threads in a kernel. To put it more correctly, the instruction stream for both branches is broadcast to all threads, and they choose to ignore one or the other. However, a very handy but often overlooked feature of many modern GPUs is that if an entire kernel (a cluster of cores rendering a small region of pixels with the same shader) has *all* its cores decide on the *same* branch in an if-else fork, the instruction stream for the fork that isn't needed will not be sent. Not all GPUs have this feature (yet), but the more capable desktop GPU models do.

If we move some of the frequency clamping computations to the beginning of the shader, we can at least make it possible for it to execute a lot faster in cases where large contiguous regions with the pattern get frequency-clamped out of existence.

```
// Anti-aliased checkerboard pattern with early-out frequency clamping
float clampedcheckers( vec2 p ) {
	vec4 diff = vec4( dFdx(p), dFdy(p) ); // Four partial derivatives
	float pitch = length(diff); // Roughly the pixel size in p space
	if( pitch > 0.35 ) return 0.5; // Save some work if we're far, far away
	else {
		vec2 ramps = 2.0 * abs( fract( p ) - 0.5); // "Triangle waves"
		vec2 stripes = aastep( vec2(0.5, 0.5), ramps ); // 50% stripes
		float aacheckers = abs( stripes.x - stripes.y ); // "XOR" for overlaps
		return mix(aacheckers, 0.5, smoothstep(0.2, 0.35, pitch));
	}
}
```

This particular shader is simple, so it might not benefit greatly from the early-out test, but with a more complicated pattern it could save a lot of unnecessary work.

Anisotropic anti-aliasing

Periodic patterns are often one-dimensional, or have different periods in different directions. In such cases, we can do better than to make a rough estimate of the average size of a pixel and removing all frequencies that might possibly cause moiré. We are dealing with 3-D graphics, and we are routinely looking at a surfaces at oblique angles rather than head-on. This causes *foreshortening* in the view projection, in addition to the scaling caused by perspective, and 1-D patterns on the surface will appear more dense if they have their main axis of variation affected by that foreshortening.

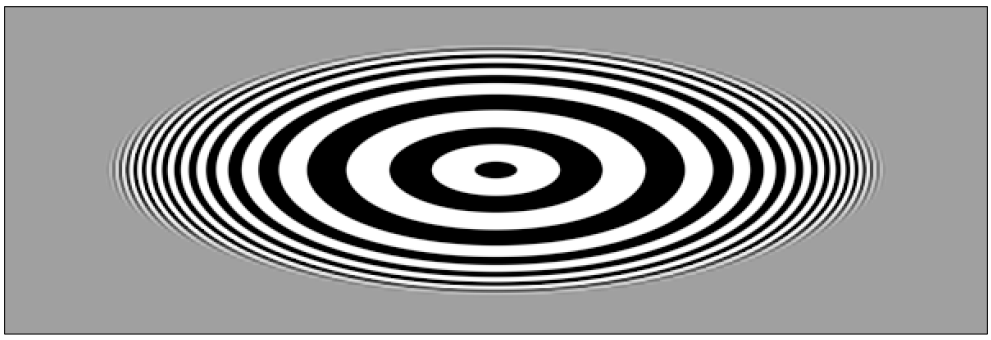
The key to not stomping out these patterns prematurely is to use the partial derivatives in exactly the way they were meant to: to compute the length of a vector on a surface in screen space. If the pattern is a wave along a certain direction, we can construct a vector in that direction with the length of the period of the pattern, transform it to device space and check whether it is too short to render correctly. Or, we can simply take the shortcut provided by auto-derivatives and just compute the gradient in device space of the argument to the periodic function, possibly adjusting for the wavelength of the function we are using. (This is why it's often a good idea to use functions with a period of 1.0 in local coordinate mappings.)

Frequency clamping of a locally 1-D pattern could be performed like this:

```
float clampedbullseye( vec2 p ) {
	p *= vec2(0.5, 1.2); // Scale differently in x and y
	float r = 4.0 * (length(p) – 0.5); // Radial distance
	// Linear increase near center, exponential increase further away
	float rf = (r < 0.0 ? r + 1.0 : pow( 2.81828, r ) );
	float circles = aastep(0.25, abs( fract( rf ) - 0.5 ) ); // Concentric circles
	float fw = length(vec2(dFdx(rf), dFdy(rf))); // isotropic fwidth(rf)
	// Frequency clamping
	float fade = smoothstep(0.2, 0.3, fw);
	return mix(circles, 0.5, fade); // Fade to constant 0.5 when clamping
}
```

We have used this test pattern before, but without presenting the code for it. It's shown again at the top of the next page. The pattern is 2-D, but locally it's a pattern of 1-D stripes in every direction. Note that the frequency clamping depends on the pattern frequency in *fragment space*, and therefore the anisotropic scaling causes more stripes to render in the stretched-out *x* direction.

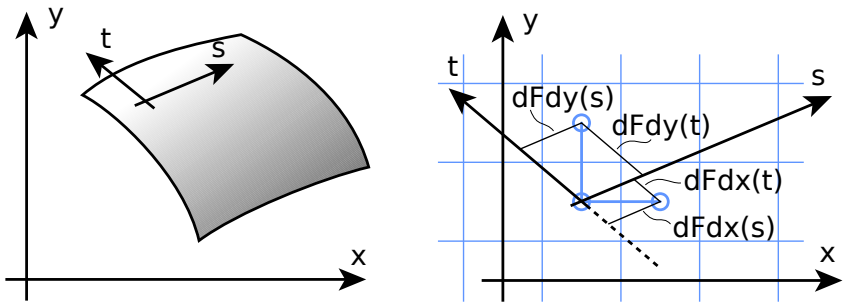
In image-based texturing, anisotropic anti-aliasing is a difficult problem that requires multisampling, and it hasn't been solved to full satisfaction even in modern GPUs. With procedural textures, anisotropic anti-aliasing can be *easy.*



Ellipses with direction-dependent (anisotropic) frequency clamping.

The Jacobian matrix

A more formal treatment of the relation between shader coordinates and the pixel size is to use the partial derivatives to create a finite difference approximation of the local Jacobian matrix of the transformation. This 2×2 matrix can then be used to transform a direction vector from shader space (texture mapping coordinates for the current surface) to device space (pixel coordinates in the current view).



Local surface coordinates (s,t) in relation to fragment coordinates (x,y)

```
// 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 ;
```

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

Usually, transformations using the full Jacobian matrix are not needed, because the auto-derivatives can be used to perform the relevant computations implicitly, often with less work. However, it's useful to know that the local Jacobian for the transformation to fragment space *can* be constructed and used for analytic anti-aliasing. In some cases, this can be the best solution.

7 Randomness

Truly random numbers are very difficult to generate. Since the invention of computers, software has been getting away with using functions that *appear* random, in the sense that they have the observable statistical properties of truly random numbers. However, “random number generators” implemented in software are actually *pseudo-random*. (“Pseudo” is Latin for “fake”. Yes, literally.) The algorithms output a deterministic sequence – it’s just very thoroughly jumbled. The sequence is also repetitive, although the repetition period can be very long indeed.

Pseudo-random numbers

A pseudo-random number generator has an internal state that uniquely determines the next number in the sequence, and that state is updated with the result to make the next number different. However, knowing the algorithm and the internal state, it’s possible for another computer to reproduce the exact same sequence. The “random” variation between different copies of the same generator is achieved by picking a different *seed*, which is literally the internal state. In some classic algorithms, the seed is simply the value of the previous pseudo-random number.

For something that approaches real randomness, a pseudo-random number generator can be given a seed that depends on some real world property that is effectively random for the purpose at hand. Computers mostly keep very accurate track of the time of day, and when a human user chooses to run a program, the number of microseconds on the clock at program startup is a truly random number (in that context) which can be used as a seed. Other strategies are to generate a seed from some mouse input, or the exact timing of a few key presses by a human operator.

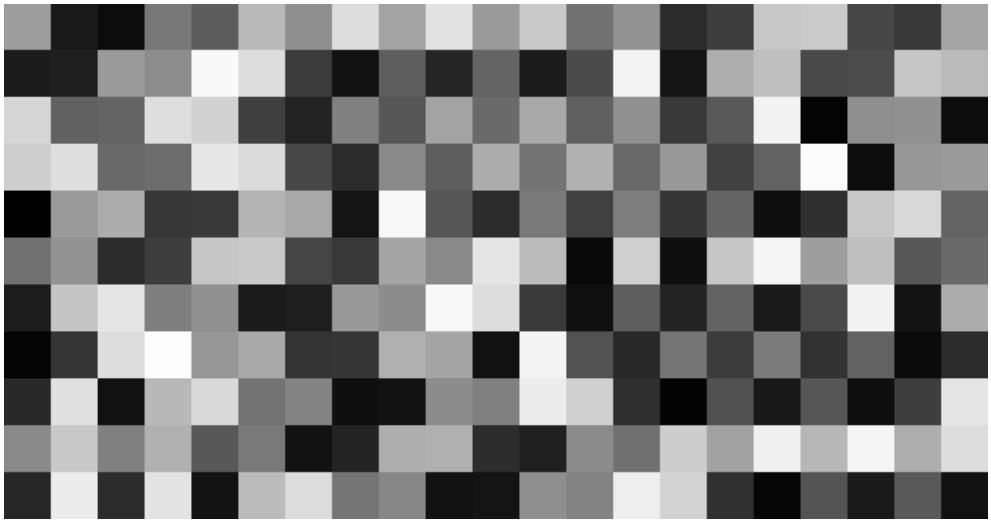
Hash functions

Pseudo-randomness in the usual sense is *not* what we want for procedural methods. Instead, we want *repeatability*: a shader executed again with the same input should yield the same result as last time. This is absolutely required for re-rendering, distributed rendering and parallel rendering, for patterns to remain consistent during animations, and not least importantly for artistic control over the detailed looks of a procedural pattern. Once you find a look you like for a pseudo-random

pattern in a certain situation, you want it to stay exactly that way. For this reason, pseudo-random numbers are not very useful as such in shaders, because of their "randomness". That randomness is just an illusion, though, and we can exploit that.

A shader function can use something akin to a random number generator, but supply a *deterministic seed*. The seed should be dependent only on deterministic parameters in the scene, such as a surface or object ID or a user-selectable constant. A decent algorithm should then be able to jumble the seed enough to hide the underlying deterministic nature of its generated "random" value. Functions like this are often called *hash functions*, or *hashes* for short.

To be useful in a point sampling situation, hash functions for procedural texturing usually operate on integer coordinates in a regular grid, and the hash value remains the same within a grid cell. There are no built-in hash functions of any kind in GLSL and other current GPU shading languages, but in RSL and OSL, there's a built-in hash function named ***cellnoise(x,y)***, which returns a *seemingly* random (but actually deterministic) value for each cell in a 1×1 square grid:



cellnoise(x,y) *from OSL: a "random" value in [0,1] for each integer pair* (x, y)

The language specification doesn't say exactly what algorithm to use for the ***cellnoise*** hash, and it's unwise to make a shader dependent on the exact values. It's guaranteed to be repeatable across renderings on the same platform, but not to match between platforms, such as two different renderers that both support OSL shaders. If you want that kind of repeatability, you had better create your own hash function in shader code, even in environments where a built-in function like ***cellnoise*** is available.

Permutation tables

Hash functions have applications in computer science far beyond the narrow field of procedural patterns, and we don't have to invent them for this particular purpose. However, what we need to keep in mind when borrowing hash functions from other fields is that we're not aiming for high quality, but for speed and ease of use. What constitutes enough quality depends on the situation, but we can often get by with surprisingly *bad* hash functions (by formal statistical measures) and still create visually pleasing random-looking patterns.

A trick which was commonly employed a couple of decades ago was to use a *permutation table*, which is simply an array with a jumbled sequence of numbers. Many authors used a table of 256 numbers between 0 and 255, rearranged by trial and error to remove undesirable regularity. A lookup table, however, requires a memory access to use it. In many modern environments for shader execution, even in software rendering, memory access can be quite expensive.

Fortunately, there are several means for generating both permutations and hashes on the fly. In fact, there are way too many to cover them all here, so we will restrict ourselves to four specific variants, chosen for having quite different approaches.

Integer hash

Classic random number generators, the kind that you usually find implemented in most programming languages as library functions with names like ***rand*** or ***random***, typically use an internal state that is just a single integer, and by supplying a seed you set the value of precisely that integer. The statistical quality of these ***rand*** functions varies wildly between implementations, but most favor speed over quality. If you want better apparent randomness, there is nothing magic about the built-in functions – it's just software, and you can implement your own using any algorithm you want.

As our only example of this class of functions, we present *PCG hash*, where PCG is short for "permuted congruential generator", a concept originally presented by Melissa O'Neill in 2014 [https://www.cs.hmc.edu/tr/hmc-cs-2014-0905.pdf] and now adopted as a hash for computer graphics by many authors. Despite its simplicity, it has excellent statistical properties for most purposes:

```
uint pcg_hash(uint input) {
    uint state = input * 747796405u + 2891336453u;
    uint word = ((state >> ((state >> 28u) + 4u)) ^ state) * 277803737u;
    return (word >> 22u) ^ word;
}
```

A 2-D hash can be made by repeated calls to the hash function, using the strategy $hash(x, y) = hash(hash(x) + y)$, but this function is "random" enough to allow mashing the x and y components together and computing the hash in one go:

```
// 2-D cellnoise by two nested calls to pcg_hash
float cellnoise_pcg2(vec2 p) {
	uvec2 pu = uvec2( ivec2( floor( p ) ) & ivec2( 0x7FFFFFFF ) );
	return float( pcg_hash( pu.x + pcg_hash( pu.y) ) & 0xFFFFu ) / 65536.0;
}

// 2-D cellnoise by a single call to pcg_hash and an ad-hoc combo of x and y
float cellnoise_pcg1(vec2 p) {
	uvec2 pu = uvec2( ivec2( floor(p) ) & ivec2( 0x7FFFFFFF ) );
	return float( pcg_hash( pu.x + pu.y * 289u) & 0xFFFFu) / 65536.0;
}
```

The PCG algorithm we chose uses unsigned integers, ***uint***, and the conversion from ***float*** to ***uint*** is performed via ***int*** and a bit-mask (a "binary modulo" operation) to stomp out the sign bit and wrap negative integers to positive. If we did the conversion from ***float*** to ***uint*** directly, this ***cellnoise*** would fail for negative integers. The same bit mask trick is used on the return value to mask out the high order bits of the ***uint***. A ***float*** can't hold the larger unsigned 32-bit integers without losing precision, and losing precision in the hash would get us fewer unique values.

The bit-twiddling and integer math in these functions can be *very* slow on some GPUs. Before you go all-in on these, you might want to test their performance.

Floating point hashes

Unfortunately, integers are still second-rate citizens with weak or even nonexistent support in some GPUs. A traditional CPU is often a lot faster with performing arithmetic on integers than on floating point values (the standard types named ***float*** and ***double*** in many programming languages), because the bit-level operations in hardware are considerably less complicated for integer arithmetic. Low-end CPUs for embedded systems still don't even have floating point support in hardware.

With shader-programmable GPUs, it was the other way around at first: floating point math is essential to computer graphics, but integer math isn't, so the shading languages had *only* floating point types, and only the 32-bit "single precision" variant (***float***) was available. A GPU is still designed for rendering images, even though the high-end models are now doubling as general purpose computing devices. Many low-end models have weak or nonexistent support for 64-bit "double precision" floating point (***double***), and even the best GPUs of today are

significantly faster with ***float*** than with ***double***. (Some mobile GPUs use even lower precision than 32 bits for what GLSL still refers to as the ***float*** type.) Integer math is still not supported on all platforms, and even when it is, it typically has *lower* performance than floating point math.

As a consequence of this, it's still useful to have a hash function which uses nothing but floating point math. Some of the operations that are ubiquitous in traditional integer hashes, like bit-shifts and the XOR operation, are simply not available in floating point, so we need a different strategy. What we want is an algorithm that takes as its input an integer-valued ***float***, and returns a reasonably random-like number. To compute multi-dimensional hashes, it's useful if the return value is also an integer-valued ***float***, because then we can compute a 2-D hash as a sequence of two calls to a 1-D hash: $hash(x, y) = hash(hash(x) + y)$.

Permutation polynomials

An algorithm that works reasonably well, if you take some care to tweak it, is to use a *permutation polynomial*. This is a class of polynomials that permute a sequence of consecutive integers modulo-N. If the input is the numbers $\{0, 1, 2, 3, \ldots, N-1\}$ the output is the same numbers, but in a different order.

We won't get into details on how or why this works, or exactly how to construct a permutation polynomial, but it can be useful to know that quadratic polynomials of the form $p(x) = (2Ax^2 + Bx) \bmod A^2$, where A is a prime number, are guaranteed to be permutation polynomials. A quadratic polynomial is easy to compute, and it's a good fit for float-only or float-favoring hardware. If the input values are evenly distributed, the output values are, too, and the statistical properties of this kind of hash can be good enough for many applications if you're not too picky. A function that shows decent "randomness" for our purposes is $p(x) = (34x^2 + 10x) \bmod 289$:

```
// Simple 1-D floating-point hash of adequate quality
float permute289( float x ) {
	float h = mod( x, 289.0 );
	return mod( ( 34.0 * h + 10.0 ) * h, 289.0); // 34x^2 + 10x mod 289
}

// 2-D cellnoise of adequate quality using only floating point math
float cellnoise_perm( vec2 p ) {
	return permute289( permute289( p.x ) + p.y ) / 289.0; // Range [0,1]
}
```

Trash bits hash

Finally, because hacks like this are floating around in all sorts of forums and refuse to die, we present this barely passable 2-D hash function:

```
// Horribly bad 2-D cellnoise
float cellnoise_sin( vec2 p ) {
    return fract( sin( dot( p, vec2(12.9898, 78.233) ) ) * 43758.5453 );
}
```

At first glance, it looks like an absolute mystery how this single call to the ***sin*** function with some weird scaling can yield anything that is even close to random-looking, but the devil is in the details. The scalar product by the vector $(12.9898, 78.233)$ is a way of combining the x and y components into one value that *probably* won't cause too strong directional artifacts, at least not if you don't look too closely. The up-scaling by 43758.5453 just before the ***fract*** is a trick that shaves off all the high-order bits from the ***sin*** value and jumbles the remaining low-order bits somewhat. In current GLSL implementations, the ***sin*** function is computed by a numerical approximation designed for speed rather than accuracy. Scaling the value up by a lot and then chopping off the integer part leaves only the "trash bits", the low order bits that are sort of random in nature, and *hopefully* don't have a clearly visible periodicity to them.

This is compact code that runs quite fast, but there are strong downsides. It abuses a numerical approximation of ***sin*** by deliberately using the low order "trash" bits to compute the main result. The details of the ***sin*** algorithm are undocumented, it can change without notice, and the result is platform-dependent. The distribution of output values is non-uniform, and there are directional and periodic artifacts in the output. This is an unpredictable, ugly hack, and you *really* shouldn't use it. Many people use it in their examples, but that doesn't make it right. It's not super fast, and it's not even good. When rendered as a pattern, it has streaks and periodic artifacts which stem from the attempt at saving one ***sin*** call by the initial scalar product. A better function for 2-D ***cellnoise*** would be to use the hash twice:

```
// The infamous "fract-sin hash", a hack that works only by accident
float sin_hash( float x ) {
    return fract( sin( floor(x) * 12.9898 ) * 43758.5453 );
}
// A 2-D cellnoise implemented with two nested calls to the bad hash above
float cellnoise_sin2( vec2 p ) {
    return sin_hash( sin_hash( p.x ) * 78.233 + floor( p.y ) );
}
```

However, when we do that, the speed advantage over the permutation polynomials disappears. Even with hardware acceleration for the ***sin*** function, a simple permutation polynomial requires only a few multiplications (in our case four, counting the inherent multiplications by $1/289$ in the mod calls), and is actually faster to compute on most GPU architectures, despite the hardware-accelerated ***sin***. Compactness on the source code level doesn't necessarily mean fast execution, and compact source code is *not* something to strive for as such. What matters in shader programming is the quality of the output, the execution speed on the targeted GPU platforms and, as with all kinds of programming, readability of the code.

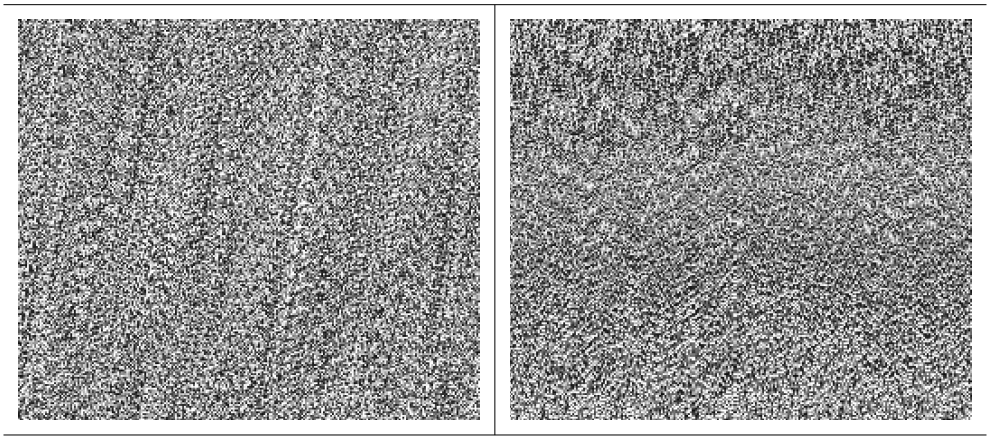
Ad-hoc float hash

Instead of using the trash bits of the ***sin*** function, we can use a ***fract*** function on the result of a non-linear but still reasonably simple sequence of arithmetic operations on the components (x, y). This particular one was suggested by Dave Hoskins on Shadertoy [https://www.shadertoy.com/view/4djSRW]:

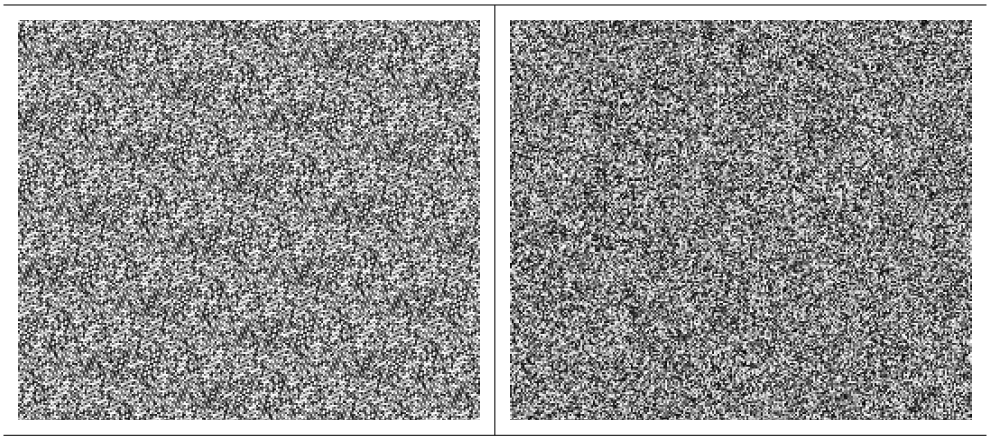
```
// One of Dave Hoskins' "sin-less" fract hashes from Shadertoy
float hash12(vec2 p) {
	vec3 p3  = fract(vec3(p.xyx) * 0.1031);
	p3 += dot(p3, p3.yzx + 33.33);
	return fract((p3.x + p3.y) * p3.z);
}
// A surprisingly good cellnoise function using the hash above
float cellnoise_hoskins( vec2 p ) {
   return hash12(floor(p));
}
```

This is sort of a mixed cubic polynomial in x and y, and it does a pretty good job. It requires a few more multiplications than the permutation polynomial hash, but it's still both faster *and* better than the ***sin*** hack. Don't use the ***sin*** hack. The ***sin*** hack is *bad*. Your friends, colleagues and pets will shun you for it.

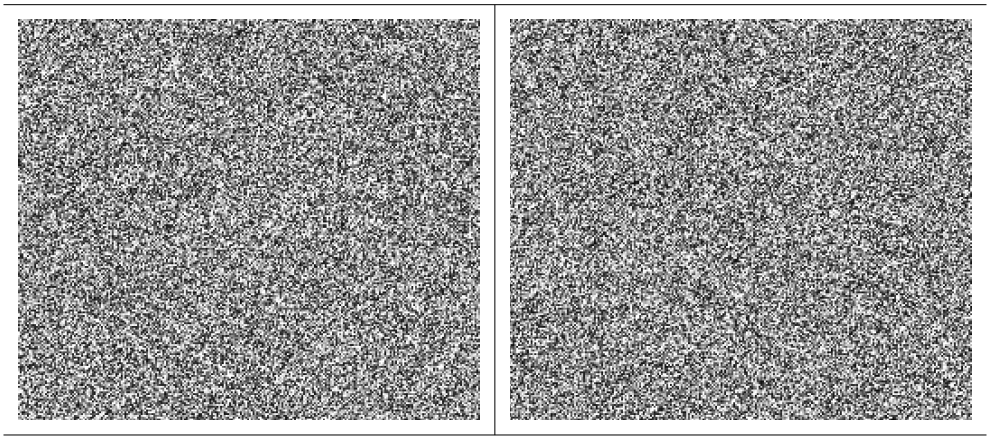
In conclusion, we present a visual comparison of the ***cellnoise*** variants presented above.



*2-D **cellnoise** implemented with "trash bit hash" using the **sin** function.*
***Left:** one combined call. **Right:** two nested calls. The quality is adequate only with two nested calls, but even there we can see some large scale pattern artifacts.*



***Left:** permutation polynomials. **Right:** Hoskins' ad-hoc (but good) hash.*
Permutation polynomials are adequate, but with potentially problematic artifacts.
Hoskins' ad-hoc hash is surprisingly good for this purpose.
On most modern GPUs, both of these execute faster than the "sin hack".



*2-D **cellnoise** implemented with he PCG hash function.*
***Left:** one call with ad-hoc merging of x and y. **Right:** two nested calls.*
The quality is excellent in both cases. On modern GPUs with good integer support, this could execute faster than the floating point hashes above. It's worth trying.

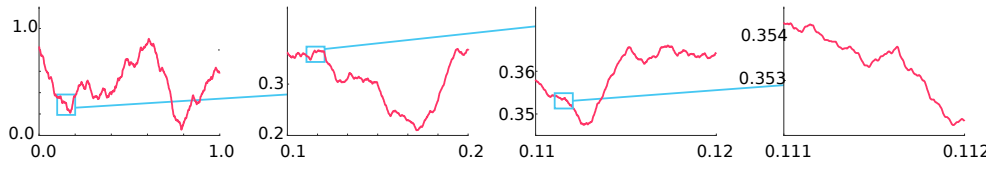
It's near impossible to say anything conclusive about execution speed, because there are huge differences between different GPUs. Some can't handle integers at all, some do it painstakingly slowly, and others do it well. The ***sin*** function has considerable variation as well, both in terms of speed and (in)accuracy.

8 Fractals

Entire books have been written about fractals, including the famous title from 1982 that started a trend: *"The Fractal Geometry of Nature"*, by Benoït Mandelbrot – the mathematician who actually coined the term "fractal" in the 1970's. This chapter will only touch briefly on the vast subject, in order to motivate some of the tricks presented in the following chapters. Part 2 of this book contains a more thorough treatment of fractals, because they are more useful for creating *objects*.

What is a fractal?

A fractal is a function that has "self-similarity across all scales", which is a rather useless definition if we don't explain it properly. Simply put, a fractal is a function that has the same general appearance in any local region, regardless of zoom level.



A fractal function sampled at increasing resolution. Note that the range of values shrinks at about the same rate as the size of the interval.

Fractals in real life

Many objects and phenomena in the world around us have a fractal appearance across a wide range of scales: coast lines, ocean waves, rivers, mountains, clouds, paper (when you look at it *really* close), trees and plants, even our lungs and the network of blood vessels in our own bodies. These things all came into being in a hierarchical fashion, with smaller features being spawned from larger features. Biological structures are grown, while inanimate fractal objects are often created by *turbulence* in a fluid medium. "Fluid" in this context means "not solid" – either a gas or a liquid. Somewhat crudely described, turbulence happens because some disturbance in a flow causes swirls, and the larger swirls create smaller swirls in the opposite direction at their perimeter. This process continues to spawn ever smaller

swirls, until they are too small and have too little energy for the process to continue. Solids can be shaped by turbulence as well, either when they are formed (like rock forming from lava) or because they are affected by wind and water through *erosion*, which is caused by turbulent flow across a solid surface.

All real life "fractals" are in fact *not* fractals in the mathematical sense, because there is a limit to how much you can zoom in on them before they lose their self-similarity. Trees have smaller branches sprouting from larger branches in several levels, but eventually the branching stops, and the smallest twigs sprout leaves instead. Leaves are at least somewhat fractal-like as well, but trees are not *infinitely* detailed. A coastline gets more jagged the closer you look, but when you zoom in *really* close, even on a particularly rocky portion, it stops looking rough at some scale and becomes more like a straight line. You could make a contrived example where you zoom in on individual grains of sand and count their surfaces as the "coastline" (although it strains the definition of the word), but grains of sand have a crystalline structure with mostly flat facets that makes the fractal character of terrain break down at least when you get to the microscopic level. A "fractal" in nature can show self-similarity across a *wide* range of scales, but not across an *infinite* range. Turbulent phenomena have a cut-off in size because of *viscosity*, which is a thing even for gases, although it's orders of magnitude stronger for liquids. The impact on turbulence from viscosity is non-linear – it has a proportionally stronger dampening effect on small swirls.

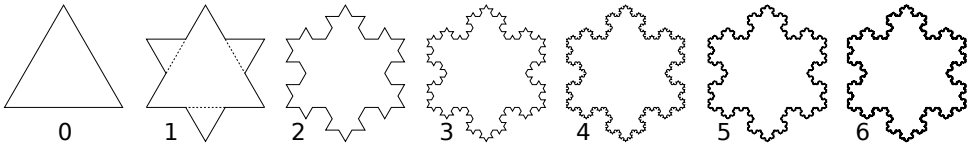
The fact that fractal-like objects in nature are not fractals in the mathematical sense means that they don't have infinite complexity, which is fortunate when we want to model their shape and appearance with computer graphics. Furthermore, in computer graphics we concern ourselves with what things *look like*, not necessarily how they are formed or built or grown, and if we don't zoom in really close on something, it doesn't need to be modeled down to the minute detail. There is seldom any need to concern ourselves with details that are significantly smaller than one pixel in the rendered output.

Fractal dimension

Fractals are weird mathematical curves and surfaces that can be characterized by their *fractal dimension*. We won't present the formal definition of fractal dimension here, but the concept of a curve somehow being more than one-dimensional, or a surface being more than two-dimensional, could use at least an intuitive explanation. A reasonably simple demonstration uses the *snowflake fractal*, also called the "Koch curve" after the Swedish mathematician Helge von Koch, who first presented it in 1904. It's defined in a recursive manner, with each step being a

Some real life objects with fractal properties. There are countless other examples.

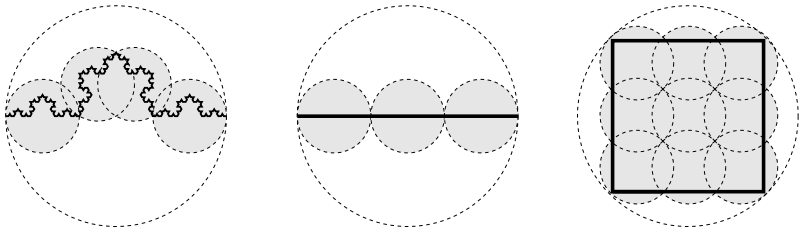
well defined subdivision of line segments into a sequence of shorter segments. The sequence starts with a triangle and ends in something that looks like a snowflake:



Constructing the Koch "snowflake" fractal. The last few iterations will look the same unless you zoom in close. A printout might fail to render the thin lines.

Now, if we make the mental leap of performing this transformation an infinite number of times, the limit curve ("after" an infinite amount of iterations) has some really weird properties. It is continuous, because all the steps along the way are continuous. However, it's not differentiable anywhere, because there are infinitely many sharp corners within any interval of the curve. It encloses a finite area, because it's bounded in extent, but it has infinite length, because with each iteration, the length of the curve increases by $4/3$. The length of the curve at iteration N is $(4/3)^N$ times the length of the original line segment, which tends to infinity as N tends to infinity.

The concept of fractal dimension can be intuitively explained (still correctly, just not *formally*) by covering the curve with circles of decreasing diameter and counting how many circles are needed as they get progressively smaller. For comparison, we do the same with a line and a square as well.



Covering objects with circles. ***Left:*** *Koch curve.* ***Center:*** *Line.* ***Right:*** *Square.*

When we scale the circles to 1/3 of their original size, we need 4 times as many to cover the Koch curve. A line requires only 3 times as many. For a circle covering a square we would need $3 \times 3 = 9$ times as many. The Koch curve somehow falls in between. It sort of covers some area, only not really. We won't go into the reasons why, but the fractal dimension of a shape can be characterized as the ratio of the logarithms of the number of circles and the inverse of their scaling. For the line, it's $\ln 3 / \ln 3 = 1$, and for the square it's $\ln 9 / \ln 3 = 2$, which both make sense. For the

Koch curve, it's $\ln 4 / \ln 3 \approx 1.26186$, which might not make a lot of *sense*, but it's consistent with the curve being "slightly more than 1-D, but not quite 2-D".

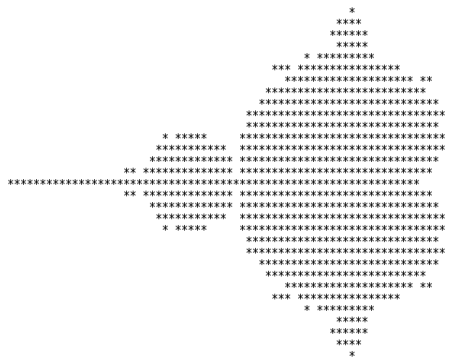
The Mandelbrot set

A chapter about fractals could not really skip mentioning the iconic Mandelbrot set, named after Benoït Mandelbrot. He didn't discover it, it was actually published by Robert W. Brooks and Peter Matelski in 1978, but Mandelbrot made it famous in 1980 by visualizing it with impressive (at that time) computer graphics.

The algorithm is defined in the complex plane, $z = x + i\,y$, by means of a deceptively simple recursive formula for a sequence of complex numbers:

$$z_i = z_{i-1}^2 + z_0$$

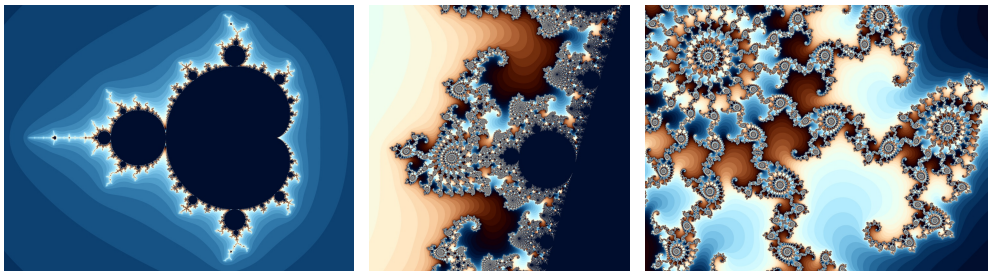
The Mandelbrot set is the area for z_0 in the complex plane where the magnitude of z_i remains bounded, meaning it doesn't diverge and escape to infinity. All points outside $\|z_0\| = 2$ diverge, but inside that circle some fun and weird stuff happens. For parts of the set, it's possible to formally prove the lack of divergence, but for visualizations it's usually enough if you run a reasonably large number of iterations and consider the initial values z_0 that still haven't caused $\|z_i\|$ to blow up to be potentially ("probably") part of the set. A remake of the first crude image of the Mandelbrot set, using 1978-style computer graphics, is shown below.



The Mandelbrot set, as it was visualized by Brooks and Matelski in 1978.

For modern visualization purposes, it's also useful to characterize points *outside* the set by counting the number of iterations it takes before $\|z_i\|$ exceeds 2, after which divergence is guaranteed. The algorithm is reasonably shader-friendly, because each point is computed independently of all other points with a simple algorithm. To evaluate the finer details of a zoomed-in view of the set, many iterations might be required, and extreme magnifications will require higher precision arithmetic than ***float***, even higher than ***double*** if you want to go wild, but as long

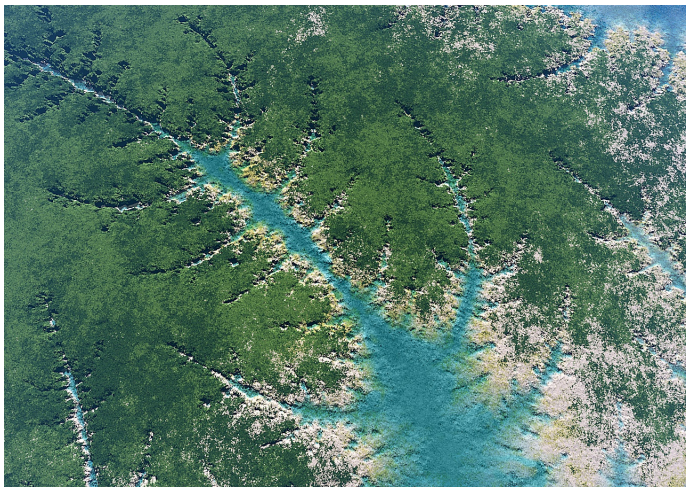
as you stay within the limits imposed by precision, a modern GPU can actually handle the computations in real time.



The Mandelbrot set, visualized by a GPU shader. Points in or near the set required hundreds of iterations, and 4×4 multisampling was used for antialiasing. Still, a modern desktop GPU can render this full-screen at an interactive frame rate.

Controlling chaos

Matematical fractals such as the Mandelbrot set (there are many others) are fascinating, but for the most part, they aren't terribly useful for creating textures. The self-similarity makes a fractal literally look very much like itself everywhere and at all scales. While there is infinite variation in the formal sense, there's not all that much variation in the *visual* sense. Any use of the Mandelbrot fractal, for example, is bound to reveal its very recognizable character. The image below, using it to create a river in a terrain-generating program, is a nerdy joke. This is absolutely *not* a bad thing, quite the contrary, but it's not a believable natural scene.



"Mandelbrot River" by Alexis Monnerot-Dumaine, username "Prokofiev" on Wikipedia. CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0/)

Trying to make a mathematical fractal do what you want is a bit like herding cats – it's frustrating, futile, and fundamentally the wrong approach. To make things worse, the infinite and crisp detail of a true fractal makes anti-aliasing a pain. It usually requires multisampling, which is computationally expensive.

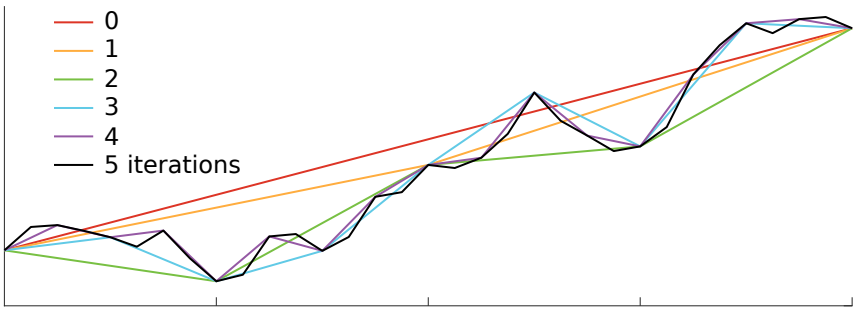
Actual, mathematical chaos from truly fractal functions isn't really what we want. We want at least *some* control over our patterns. Therefore, in the true spirit of computer graphics, we will instead fake it and create the *appearance* of chaos.

Faking a fractal

Even though the term "fractal" was coined in 1975, mathematical fractals are a much older concept, and fake fractals have been around for a long time as well, only under different names. An early example, which is still useful for procedural generation of geometry, is the *midpoint displacement algorithm*, described by Norbert Wiener already in the 1920's as part of his ground-breaking analysis of *Brownian motion*, the erratic-looking, jerky motion of microscopic solid particles in a gas or liquid. Their motion stems from collisions between the particles and individual atoms of the surrounding medium that are in constant (and rather vigorous) motion due to heat.

Because it's caused by discrete collision events, Brownian motion has a smallest scale where the particles can be seen traveling along straight lines with abrupt changes in speed and direction at each collision, so it's not an actual fractal with infinite complexity. However, from that smallest scale and up, the path of a particle *has* fractal properties, such as self-similarity and a non-integer fractal dimension. Modeling such paths can be performed either by simulating the *Wiener process* (named after Norbert Wiener) with its myriad of random collisions, but the same *kind* of curve can also be constructed by picking a starting point and an endpoint, drawing a line between them and then subdividing that line recursively, with a random displacement of the midpoint from the originally straight line. The recursive subdivisions need to be performed in a very specific manner to give a curve with *exactly* the same statistical properties as Brownian motion, but if we just want the general visual *appearance* of Brownian motion, we can be much less picky about those details.

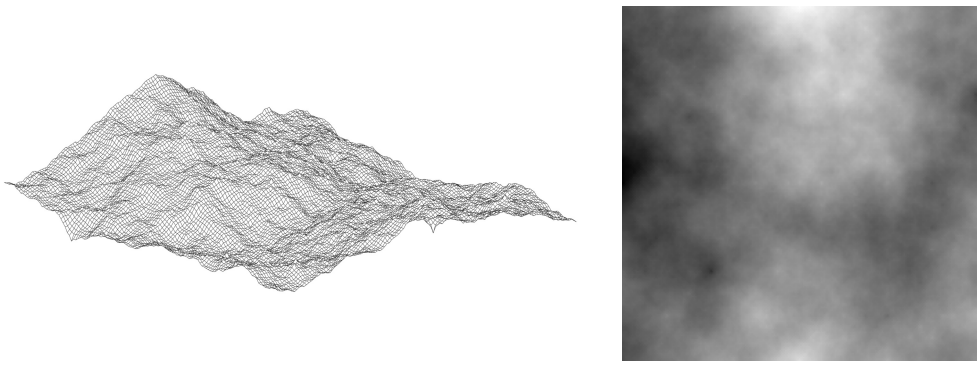
In fact, changing how the average amplitude of the displacement changes between subdivisions can yield functions that don't model physical Brownian motion, but can be very useful in their own right. This class of functions even have their own name: *fractional Brownian motion*, historically referred to as "fBm", but now mostly written in all-caps: FBM. Most people don't even care where the name comes from, but now you know. Please also note that the "F" is for *"fractional"*, not "fractal". Regular, non-fractional Brownian motion is also fractal in nature.



Random midpoint displacement of a line, in several recursive iterations.

For use in computer graphics, the midpoint displacement algorithm can be extended to 2-D to generate reasonably convincing terrain, and rather interesting images that were playfully named *plasma clouds* in computer graphics demos from the 1990's. Sometimes they are still called "plasma fractals". The recursive subdivision can be terminated whenever we have reached the desired level of detail. For image generation, that would be when we have generated all pixels in the chosen resolution, and for terrain generation, it could be when we have either generated details small enough that further subdivisions wouldn't have any noticeable impact on the rendered image, or when going further would take too much time to process. For interactive rendering, we also need to take into account how much data can be rendered at the desired frame rate by the current output device, and stop before we generate too much.

Without going into details on the algorithms, the figure below shows two results from 2-D random midpoint displacements, one which displaces a polygon mesh to create something that looks like terrain, and another which generates pixels in an image.

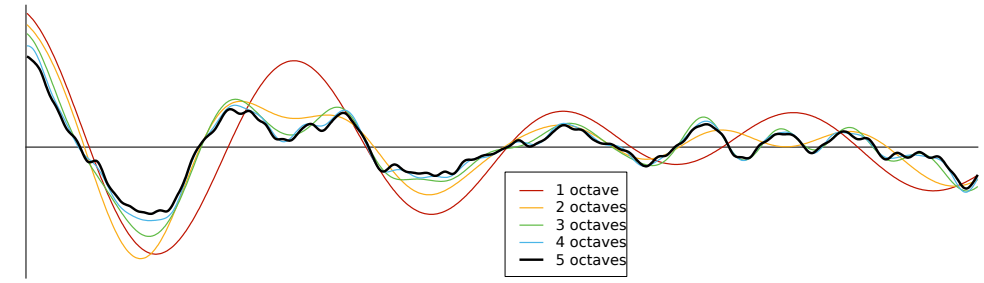


The result from a 2-D midpoint displacement algorithm. The mesh plot and the grayscale image show the same data, only displayed differently.

Because this presentation is about procedural *texturing*, we will focus on the image. Displacing a polygonal mesh with this technique will be covered in Part 2 of this book, which focuses on procedural *modeling*.

Unfortunately, the midpoint displacement algorithm isn't very shader-friendly. To compute one single data point in the final resolution, we often need to compute all the preceding iterations for a fairly large number of other points. We don't need to compute *all* the points to know *one* of the final results, but we usually need quite a few. It's *possible* to perform this in a shader without pre-computing and storing the end result, but it involves a lot of repeated work. The fundamental structure of the midpoint displacement algorithm makes it "want" to compute the entire data set in one go and store it for later use. While this is certainly an option (pre-computed but procedurally generated texture images *can* work just fine), we would also want a "pure" procedural way – some reasonably efficient way to create the same kind of pattern *without* pre-computing anything.

For this, we turn to *additive frequency synthesis*, which is a useful method to create signals with any desired frequency content, often referred to as *spectral properties*. For reasons of efficiency, this is often done by Fourier transform methods, but that, too, requires pre-computation of an entire image. Instead, we take a "back to basics" approach and sum up individual sinusoidal components within the desired range of frequencies. We start at some base frequency, representing the largest features we want to generate, and proceed to add more terms until we get the kind of signal we want. In one dimension, this can look like the plot below.

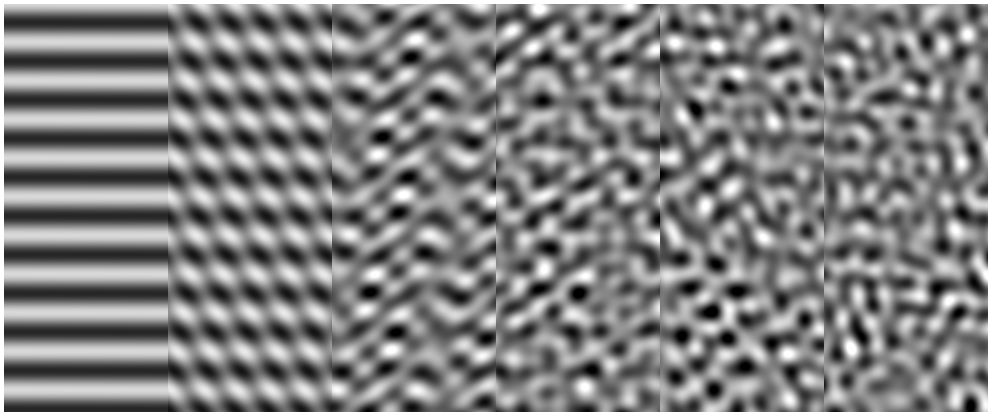


Direct frequency synthesis by adding many sinusoidal terms of random phase, with different frequencies covering 1 to 5 octaves

The term "one octave" is borrowed from music and denotes a range of frequencies from f to $2f$. When we include frequencies from enough octaves, the curve starts to look like a fractal. Mathematically, we can show that the limit curve, if we imagine that we were able to actually sum up an infinite number of terms spanning an infinite range of frequencies, *is* indeed a fractal. The advantage here is that, contrary to an actual fractal, we have full control over how much fine detail we

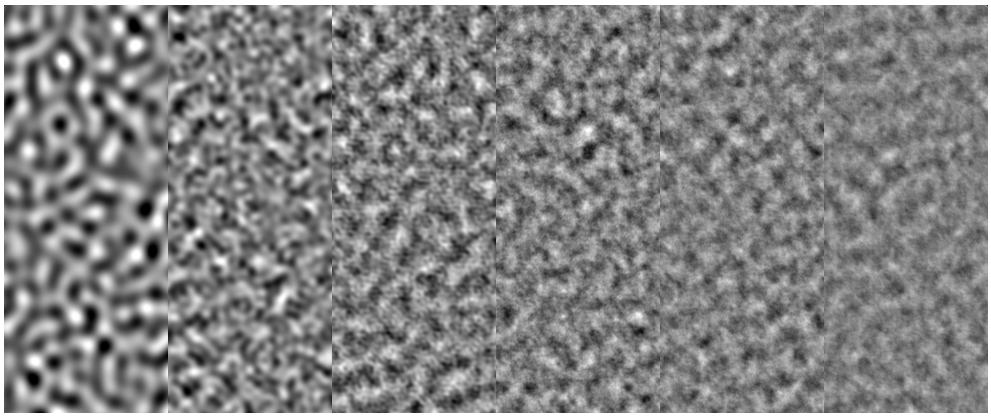
generate. If we are to sample the signal, we can stop adding terms when we hit the Nyquist limit – we can perform *frequency clamping*. This isn't possible in an actual fractal, but our fake fractal makes it concrete and outright *easy.*

This looks promising, so let's extend it to two dimensions! Sinusoidal waves in 2-D have frequency and phase just like in the 1-D case, but also a direction, so we need to make the terms have random phase *and* direction for the pattern to look random.



Direct frequency synthesis in 2-D by adding sinusoidal terms within one octave, having random phase and direction. Left to right: 1, 2, 5, 10, 20 and 50 terms.

With too few terms, the pattern is uneven and has artifacts stemming from the fact that we built it from simple sinusoidal stripes. The artifacts don't disappear until we reach several dozen terms. Adding some 50 terms for each rendered pixel, with each term using one call to the ***sin*** function, just to get a blobby, blurry image might sound silly, and for most practical purposes it *is* silly. However, it gives us a clue to what we want. The sum of terms for one octave might not look like much, indeed it doesn't really look like *anything*, but it's a very useful *band limited* pattern. The size of its largest features are determined by the lowest frequency in the sum, and the smallest features are determined by the highest frequency. What's more, we can control the amplitude of each component individually, making the higher frequencies have successively lower amplitude. This mimics the decay in amplitude of the random displacements of the recursive midpoint displacement method, and we can create an image that looks like the "plasma cloud" by adding terms across several octaves.

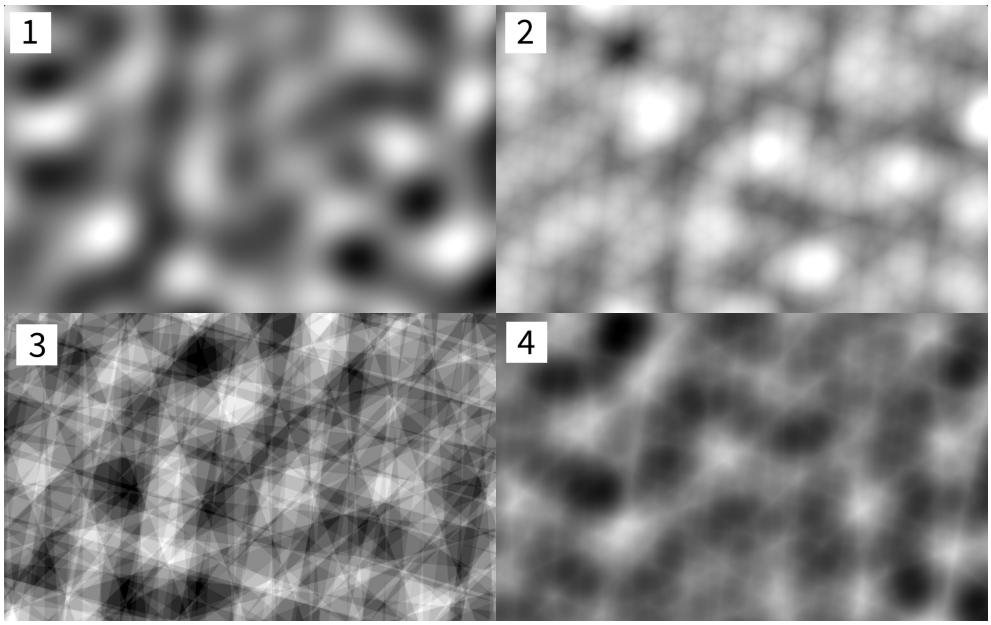


Adding sinusoidal terms within several octaves. Left to right: 1, 2, 3, 4, 6 and 10 octaves. The rightmost image required 500 terms in total – 50 for each octave.

The rightmost image in the figure above looks a lot like the result from the random midpoint displacement, with one important difference: it lacks low frequency content, meaning it has no large features. We could add them if we wanted to, by including more octaves at the low frequency end of the sum, but we don't have to. What we have here is a *controllable* fake fractal, with a distinct *feature size* (its lowest frequency) and with controllable "graininess" (its highest frequency and the decay in amplitude as the frequency increases). This provides a great foundation for creating procedural textures.

The blurry-blobby patterns above, the ones with components in one octave only, look very similar to "Perlin noise", which will be the subject of the two following chapters. Contrary to using hundreds of ***sin*** calls to render the pattern, Perlin noise is a lot easier to compute and can be used routinely as a tool for putting texture to surfaces, even in real time applications, and even on inexpensive low power GPUs.

However, first we have one more fun thing to try with our fake fractal sums. The ***sin*** function is not the only option for the base functions for each term. What if we tried some other periodic functions instead? Well, if we keep the number of terms in the sum small and stay within one or two octaves, the characteristics of the base functions show through in the end result.



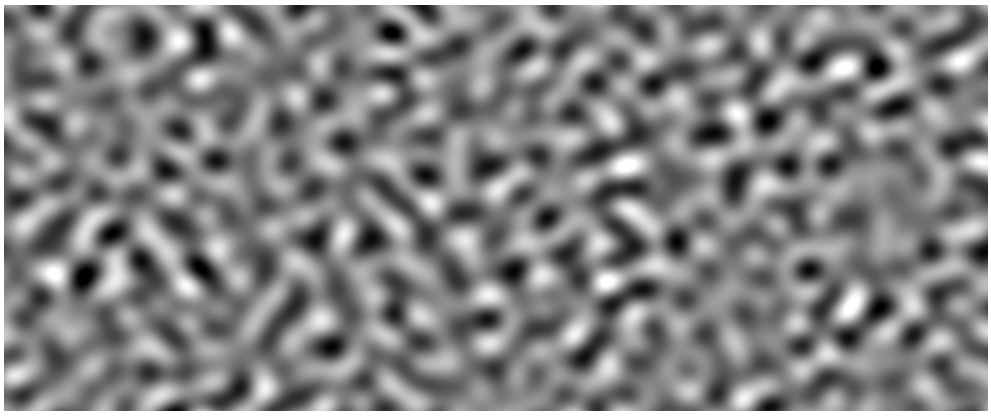
2-D frequency synthesis with different periodic functions for the individual terms. 1) ***sin****, 2)* ***abs*** *of* ***sin****, 3) square waves and 4) negative* ***abs*** *of* ***sin****. These examples span only one octave and have no fractal character.*

When we're not using sine functions, the sum is no longer formally band limited, but it still has a clear "scale" in terms of its visual feature size, and we have direct control over the detail if any of the components should prove problematic and cause aliasing. In fact, for the square wave example, ***aastep*** was used for each term to get a better looking pattern.

If we sum up lots of terms and let them span several octaves, these pattern variations lose their unique character and take on more or less the same look as if we had used the ***sin*** function. This *could* be utilized to save us some work, because a simple periodic function like a triangle wave can be computed with considerably less work than a trigonometric function. However, we would still need to sum up hundreds of terms to create a fractal-like pattern, and there are better ways by far to achieve the same goal – most notably the ones presented in the following chapters.

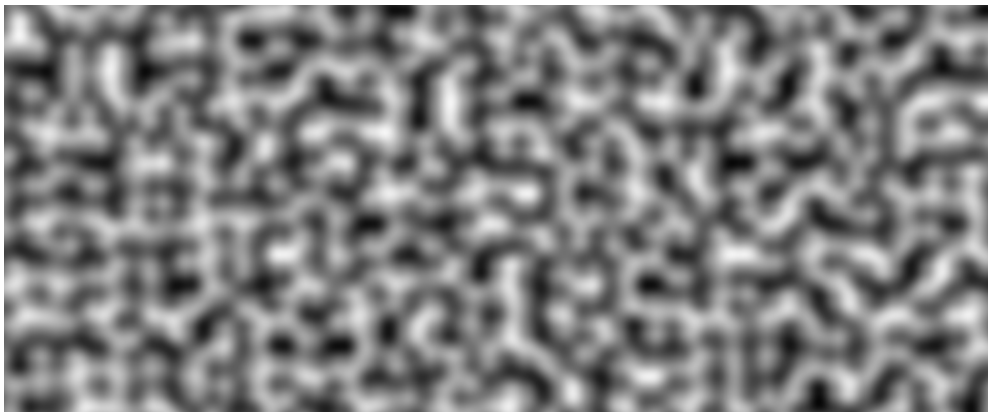
9 Noise

At the end of the previous chapter, we arrived at a pattern that looked like this:



Frequency synthesis using a sum of 50 random sinusoidal terms in one octave

In this chapter, we will present an algorithm to generate a similar pattern directly:

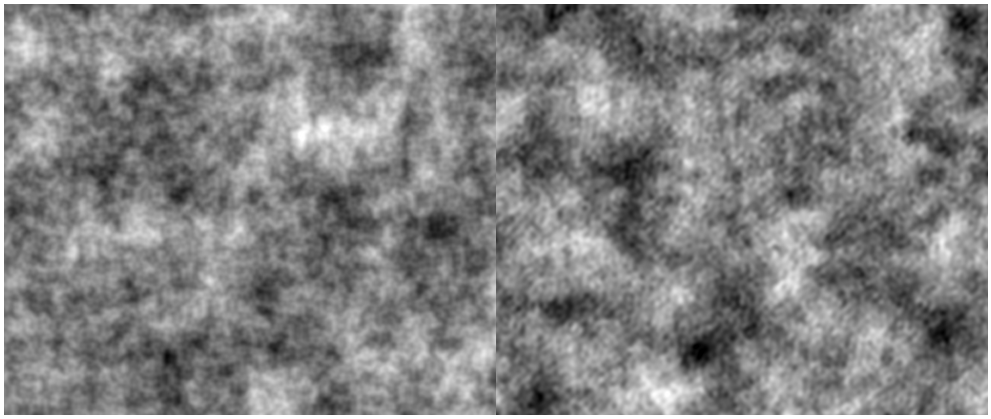


A classic 2-D "Perlin Noise" function, from https://github.com/stegu/webgl-noise

The algorithm and the pattern both go by the name *Perlin Noise*, named after its inventor Kenneth ("Ken") Perlin, who published it in 1985 in an article titled "An Image Synthesizer" [https://doi.org/10.1145/325165.325247].

The need for noise

Why do we even want this pattern? It doesn't look terribly interesting. Well, no, it doesn't look like much, but it's not supposed to. It's a *base function* that can be used as a building block for a very wide range of patterns. It's a band limited function with a bandwidth of one octave, meaning that it has features within a certain range of sizes, lacking larger as well as smaller details. Frequency synthesis of fractal patterns can be performed by summing up scaled versions of noise. Instead of summing up hundreds of sine waves, we can sum up only a few noise functions. We lose the detailed control over phase, direction and amplitude for every single term, but there are many applications where that level of control isn't needed. When Perlin noise is used in "FBM" style fake fractal sums, the final result can be visually indistinguishable from synthesis using individual sine waves.



Fake "FBM" fractal sums. ***Left:*** *five terms of Perlin noise.* ***Right:*** *500 sine waves.*

In fact, the "faked" band limited pattern from Perlin's algorithm is more well-behaved for pattern generation purposes. It's more restricted in amplitude, so it doesn't have the occasional strong local minima or maxima that can be seen in a "true" spectral synthesis. It doesn't look as random by itself, but it doesn't need to – it's random-looking *enough* and easy to compute. The Perlin noise function, or variants of it, has been a workhorse like no other for the computer graphics industry, and it's being used a *lot* in pretty much *any* computer graphics production. It actually earned Ken Perlin an Academy Award (an "Oscar", statuette and all) for technical achievement.

It is a rare mind indeed that can render the hitherto nonexistent blindingly obvious. The cry "I could have thought of that!" is a very popular and misleading one, for the fact is they didn't.
From "Dirk Gently's Holistic Detective Agency" by Douglas Adams

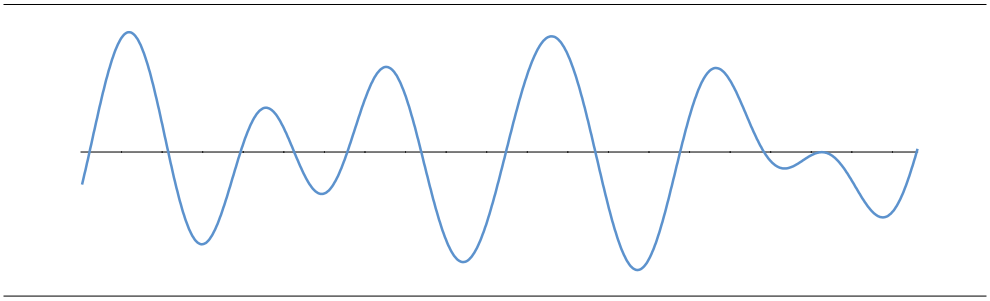
Below, we show a selection of patterns created by manipulating noise in various manners, along with the code that created the patterns. The examples are not taken from commercial productions, but self-made for the purpose of this presentation, so please pardon the slightly crude visuals. The intention is to show principles, not demonstrate photo-realism, but please know that photo-realistic rendering uses noise extensively to create dirt, wear, imperfections and roughness – in one word, to make the images *believable*.

Still not done.	Cow spots: thresholding two or three octaves of noise
These examples should have reasonably simple, readable code.	Blood spatter: extreme radial displacement of a circle
I will probably have to write them all from scratch for this purpose.	Marble ball: noise-perturbed 3-D layers
But hey, fun!	Wood: trim down the Shadertoy demo
	Fake water: pool with fake refraction

Examples of use and abuse of Perlin noise

Making noise

Like so many great ideas, the noise algorithm is deceptively simple once you've had it explained to you, but not at all obvious. We want to mimic a band-limited signal, so let's look at what a sum of sine waves in one octave looks like.

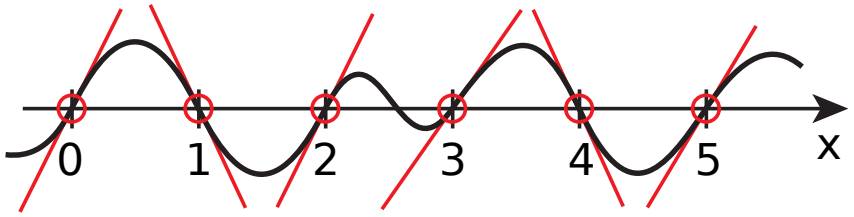


1-D signal with a bandwidth of one octave (sum of 100 random-phase sine waves)

The general appearance of the curve is that of a smooth sine function, but with varying width and height for the lobes. Because the function has no low frequency content, there are lots of zero crossings, and because it has no high frequency content, the lobes are smooth and vary in width only by a factor of 2.

1-D noise

Perlin noise mimics this general behavior by a rather brutal but successful approximation: it (1) forces zero crossings at regular intervals, (2) assigns a pseudo-random positive or negative slope to the curve at the zero crossings, and (3) interpolates the function smoothly to all intermediary points. To make the algorithm simple, the zero crossings are placed at integer points. When two consecutive zero crossings have slopes of opposite sign, we get one blob between them. When they have slopes of the same sign, we get an extra zero crossing and two blobs between them. This is pretty much the kind of function we want.



The principle behind Perlin noise in one dimension

For the portion between two integers i and $i+1$, the classic interpolation scheme proposed by Perlin in 1985 was to use the following formula. Keep in mind that the value of the function is set to be zero at either endpoint of the interval.

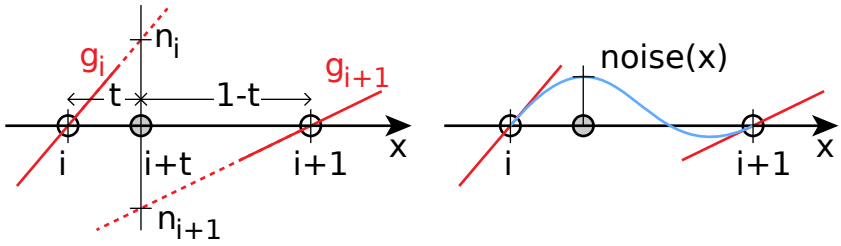
First, we assign pseudo-random slopes g_i and g_{i+1} at the endpoints. Then we *extrapolate*, extend those slopes to values n_i and n_{i+1} as if the curve would have continued straight from either point. Expressing the current point as $x=i+t$, where $0 \le t < 1$, we can write:

$$n_i = t\, g_i$$
$$n_{i+1} = (t-1)\, g_{i+1}$$

Note that the factor $(t-1)$ for n_{i+1} is negative, because we are extrapolating in the negative direction, from $i+1$ to x. Then, we blend the two values together by a cubic (third degree) interpolation polynomial that has zero slope at $t=0$ and $t=1$:

$$f = (3t^2 - 2t^3) = (3-2t)t^2$$
$$noise(i+t) = (1-f)\, n_i + f\, n_{i+1}$$

Remember the ***smoothstep*** function from chapter 4 ? This is the same curve.



The steps in computing 1-D Perlin noise

Counting the total amount of arithmetic operations we needed to perform to compute the value of $noise(x)$, we see that we only did one ***floor*** operation to compute $i = floor(x)$, eight multiplications and a few subtractions. To that total we need to add the operations required to select pseudo-random slopes for the two endpoints. If we use a second degree modulo-N permutation polynomial (see chapter 7) as our hash function, that requires four multiplications, 1 addition and 2 ***floor*** operations for each endpoint. Then we need to map a random integer in the range $[0, N-1]$ to a slope in the range, say, $[-1, 1]$, which requires at least another multiplication and a subtraction, so the gradient selection is actually more work than the interpolations.

The point here is not to count the exact number of operations, but to point out that noise can be computed with a very reasonable number of operations – less than

what would have been required to compute even one decent approximation of the ***sin*** function. This is a strong advantage of the noise function: it's *easy* to compute!

A synonym for "Perlin noise" is *gradient noise*, which is a good name because it says something about the algorithm instead of just naming its author. There's a lot of confusion among well-meaning but badly informed authors about what "Perlin noise" actually means (it means what we have just told you, nothing else), so we will try to use the term "gradient noise" from here on. Try to, but often fail.

Even one-dimensional Perlin noise (sorry, *gradient noise*) is useful, because it can be applied to animations. Position and many other parameters that need to be animated in a pseudo-random fashion often benefit from having a quasi-periodic character to them: not too regular, but not too random either, and smoothly varying with no jerks or jumps. Gradient noise in 1-D fits that bill quite nicely, and 1-D noise has saved countless animators many hours of tedious keyframing for secondary animations. However, for it to be of much use in pattern generation, we need to extend the algorithm to 2-D.

2-D noise

For 2-D noise, we stay with the general idea and assign pseudo-random gradients and a function value of zero to the square grid of integer points in the (x, y) plane. Gradients have a magnitude and a direction, and we could vary both. However, to maintain local contrast in the pattern, we choose to keep the magnitude of the gradient reasonably large, thereby avoiding large flat portions with values near zero. The interpolation is now performed in two dimensions, and we choose to do it one dimension at a time. The order is arbitrary, but let's do it first along x, then along y.

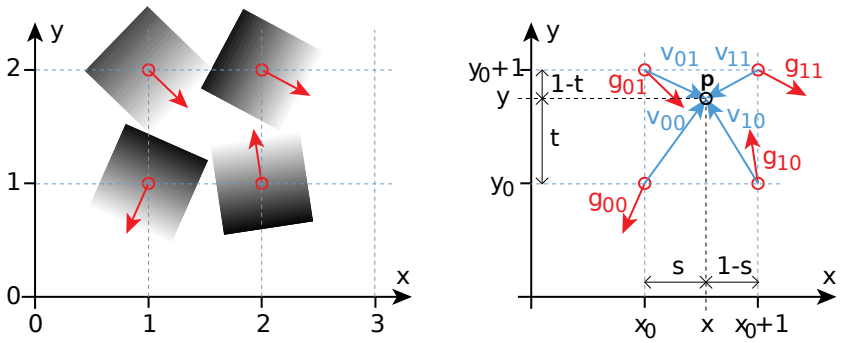
Using general indices for everything in the equations gets in the way of the explanation, so let's look only at one unit square of the (x, y) grid and give local names to the four corners:

$\bar{p}_{00}=(x_0, y_0)=(floor(x), floor(y))$
$\bar{p}_{10}=(x_0+1, y_0)$, $\bar{p}_{01}=(x_0, y_0+1)$, $\bar{p}_{11}=(x_0+1, y_0+1)$

Then, we need to assign pseudo-randomly oriented 2-D vectors as gradients, one for each of the corners. Let's name these $\bar{g}_{00}$, $\bar{g}_{10}$, $\bar{g}_{01}$, $\bar{g}_{11}$. Different authors have suggested different strategies for picking these gradients. Perlin's original version suggested using random components in the range $[-1,1]$ to make a 2-D vector, but that generates a noise field that has an uncomfortably strong variation in local contrast. This introduces low frequency spotty detail and causes the function as a whole to be insufficiently band limited. Another variant is to randomly select one vector from a small set of vectors with evenly distributed directions and only a

small variation in magnitude. A particularly simple method that works well is to pick gradients that are all of the same length, differing only in direction. Regardless of how the gradients are picked, the subsequent extrapolation and interpolation stages proceed similarly to the 1-D case, but with the added complication that the interpolants are now 2-D vectors (s,t) and that the linear extrapolation should be performed along the direction of the gradient. This is performed by one scalar product for each of the four corners.

$$
\begin{aligned}
s &= x - floor(x) \\
t &= y - floor(y) \\
\bar{v}_{00} &= (s,t) \\
\bar{v}_{10} &= (s-1,t) \\
\bar{v}_{01} &= (s,t-1) \\
\bar{v}_{11} &= (s-1,t-1)
\end{aligned}
\qquad
\begin{aligned}
n_{00} &= \bar{g}_{00} \cdot \bar{v}_{10} \\
n_{10} &= \bar{g}_{10} \cdot \bar{v}_{10} \\
n_{01} &= \bar{g}_{00} \cdot \bar{v}_{00} \\
n_{11} &= \bar{g}_{00} \cdot \bar{v}_{00}
\end{aligned}
\qquad
\begin{aligned}
f_x &= (3s^2 - 2s^3) = (3-2s)s^2 \\
f_y &= (3t^2 - 2t^3) = (3-2t)t^2 \\
\\
n_0 &= (1-f_x)n_{00} + f_x n_{10} \\
n_1 &= (1-f_x)n_{01} + f_x n_{11} \\
noise(x,y) &= (1-f_y)n_0 + f_y n_1
\end{aligned}
$$



Gradient (Perlin) noise in 2-D: extrapolations from four corners to $\bar{p} = (x,y)$

This is obviously more work than for the 1-D case, but it's still nothing compared to direct frequency synthesis. Multiplications are the most cumbersome operation here, and there are about 20 scalar multiplications in the algorithm above. This does not include the operations needed to select four pseudo-random two-dimensional gradients, which would add about the same amount of work to the total. Performing maybe 40-50 multiplications to generate a 2-D noise value is considerably less work than evaluating 50 ***sin*** functions, with different arguments requiring not only a multiplication, but also the generation and addition of one pseudo-random phase – for *each*. Even if we rely on the ***sin*** function being hardware accelerated, the gradient noise algorithm has a *huge* speed advantage.

Pipe dream: hardware-accelerated noise

If we had at least *some* hardware support for generating noise, like a good built-in hash function similar to ***cellnoise*** from OSL, noise generation would be a breeze. A weird detail here is that the GLSL specification contains several functions which were intended to generate gradient noise, named ***noise1*** to ***noise4*** depending on how many noise components you want to generate, and each of them defined for 1-D to 4-D vectors as arguments. However, only one GPU manufacturer, 3DLabs, has ever chosen to implement those functions. They were never particularly fast, and 3DLabs stopped designing GPUs in 2006, shortly after GLSL was introduced. All current GPU manufacturers list these functions as "unimplemented". The language syntax allows calling the functions, but they all return zero.

In the early days of GLSL, serious efforts were made to agree on which version of noise to implement, but the discussions got stuck in indecision. Additional issues with patent claims to the algorithms that were considered effectively put a stop to any further efforts to make built-in GLSL noise happen.

However, there's no reason to weep over this, because these days a GPU is capable enough for us to implement our own noise algorithm of choice and use plenty of it in shaders. Having detailed control over the implementation makes the shader patterns portable, and it also presents the shader programmer with options to balance speed against quality.

Sample implementation

An old but still very much serviceable GLSL implementation of 2-D gradient noise is presented below. It's not a lot of code. Some of it is a nearly unreadable mess written for speed, but the general structure is precisely the algorithm presented in the equations above, and you should be able to follow it.

```
// GLSL classic 2D gradient noise ("Perlin noise")
// Copyright (c) 2011, 2024 Stefan Gustavson
// with thanks to Ian McEwan for several details.
// Distributed under the permissive MIT license:
// https://github.com/stegu/webgl-noise
// Please give credit, and please keep this header.

// Different permutation polynomials for x and y to avoid artifacts
vec4 hash(vec4 ix, vec4 iy) {
  vec4 h = mod((ix*51.0 + 2.0)*ix + iy, 289.0); // 51x^2 + 2x
  return mod((h*34.0 + 10.0)*h, 289.0); // 34x^2 + 10x
}
```

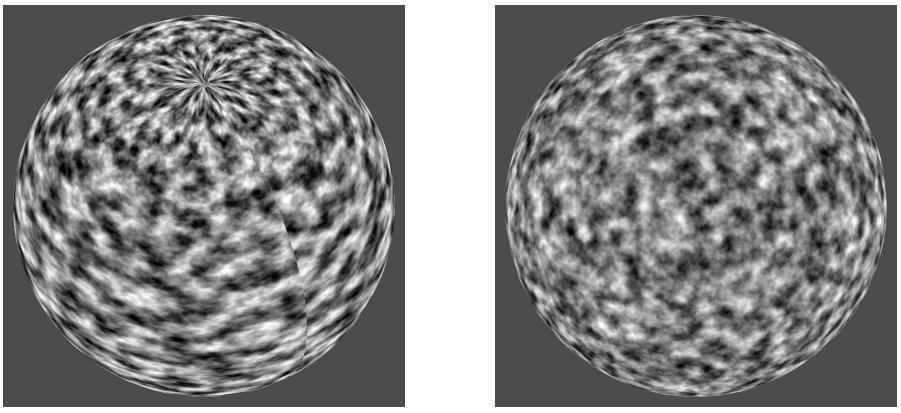
```
// A fifth degree interpolating function to replace the cubic interpolation
vec2 fade(vec2 t) {
  return ((6.0*t-15.0)*t+10.0)*t*t*t; // Better than (3.0-2.0*t)*t*t
}

// Classic 2-D "Perlin noise"
float cnoise(vec2 P) {
        vec4 Pi = floor(P.xyxy) + vec4(0.0, 0.0, 1.0, 1.0);
        vec4 Pf = fract(P.xyxy) - vec4(0.0, 0.0, 1.0, 1.0);
        Pi = mod(Pi, 289.0); // avoids truncation in the hash function
        vec4 ix = Pi.xzxz;
        vec4 iy = Pi.yyww;
        vec4 fx = Pf.xzxz;
        vec4 fy = Pf.yyww;
        // Generate hash values for the four corners
        vec4 i = hash(ix, iy);
        // Gradients are generated as points along a diamond shape
        vec4 gx = fract(i / 41.0) * 2.0 - 1.0 ;
        vec4 gy = abs(gx) - 0.5 ;
        vec4 tx = floor(gx + 0.5);
        gx = gx - tx;
        // Rearrange components to create the four gradients
        vec2 g00 = vec2(gx.x, gy.x);
        vec2 g10 = vec2(gx.y, gy.y);
        vec2 g01 = vec2(gx.z, gy.z);
        vec2 g11 = vec2(gx.w, gy.w);
        // Factors to scale gradients to equal lengths
        vec4 norm = inversesqrt(vec4(dot(g00, g00), dot(g10, g10),
                dot(g01, g01), dot(g11, g11)));
        // Extrapolated contributions from each corner
        float n00 = norm.x * dot(g00, vec2(fx.x, fy.x));
        float n10 = norm.y * dot(g10, vec2(fx.y, fy.y));
        float n01 = norm.z * dot(g01, vec2(fx.z, fy.z));
        float n11 = norm.w * dot(g11, vec2(fx.w, fy.w));
        // Successive interpolations, first along x, then along y
        vec2 fade_xy = fade(Pf.xy);
        vec2 n_x = mix(vec2(n00, n01), vec2(n10, n11), fade_xy.x);
        float n = mix(n_x.x, n_x.y, fade_xy.y);
        return 1.58 * n; // Empirical factor to scale output to [-1,1]
}
```

The implementation above is a modernized version of code published in 2012 [Journal of Graphics Tools, https://doi.org/10.1080/2151237X.2012.649621]. Some spoon-feeding for the less clever compilers that were prevalent at the time has been removed, but the code is still largely the same as in 2012, it *works*, and it's *fast*. GLSL has remained a remarkably stable platform for development for two decades. Many useful things have been added to the language, but old code keeps working.

Note that the interpolation is not using the cubic polynomial from the equations presented previously, but a fifth degree interpolation which yields a clearly noticeable improvement at a small extra cost. This is covered in detail in the next chapter – we just didn't want to provide bad code in an example. Feel free to use the noise function above for your own experiments. The "MIT license" mentioned in the header is a very permissive license that basically says "use freely for any purpose, but please give credit". Just don't remove the header, and you're fine.

3-D noise

The noise algorithm can be extended to 3-D as well. A 3-D function can be used for *hypertexture*, describing things like the local density of smoke, clouds or fog, or the structure of a porous sponge-like material, but its most common application is to eliminate the need for an explicit 2-D surface mapping on objects with a complicated shape, or objects that are literally carved out of a material that has a 3-D structure, like wood or marble. Instead of assigning 2-D coordinates to each point on the surface, we can simply use the 3-D spatial coordinates for the surface points and let a 3-D procedural texture $F(x, y, z)$ determine the appearance at each point. 3-D textures can save a lot of work with surface mapping of objects with a complicated shape. A 3-D texture can also avoid texture seams, uneven scaling and pinching problems. 3-D noise is a particularly versatile function for 3-D textures.

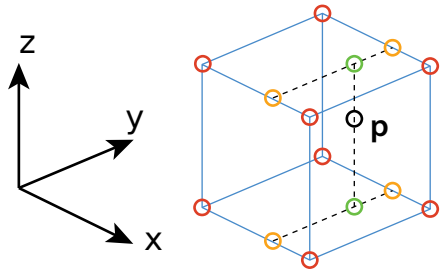


***Left:** 2-D texture mapping of a sphere, showing stretching, pinching and a texture seam. **Right:** 3-D mapping, showing none of those problems.*

The extension from 1-D to 2-D was a reasonably smooth ride, and the extension to 3-D is quite straightforward as well:

$$
\begin{aligned}
s &= x - floor(x) \\
t &= y - floor(y) \\
p &= z - floor(z) \\
\\
f_x &= (3s^2 - 2s^3) \\
f_y &= (3t^2 - 2t^3) \\
f_z &= (3u^2 - 2u^3)
\end{aligned}
\qquad
\begin{aligned}
n_{000} &= \bar{g}_{000} \cdot (s, t, u) \\
n_{100} &= \bar{g}_{100} \cdot (s-1, t, u) \\
n_{010} &= \bar{g}_{010} \cdot (s, t-1, u) \\
n_{110} &= \bar{g}_{110} \cdot (s-1, t-1, u) \\
n_{001} &= \bar{g}_{001} \cdot (s, t, u-1) \\
n_{101} &= \bar{g}_{101} \cdot (s-1, t, u-1) \\
n_{011} &= \bar{g}_{011} \cdot (s, t-1, u-1) \\
n_{111} &= \bar{g}_{111} \cdot (s-1, t-1, u-1)
\end{aligned}
\qquad
\begin{aligned}
n_{00} &= (1-f_x) n_{000} + f_x n_{100} \\
n_{10} &= (1-f_x) n_{010} + f_x n_{110} \\
n_{01} &= (1-f_x) n_{001} + f_x n_{101} \\
n_{11} &= (1-f_x) n_{011} + f_x n_{111} \\
\\
n_0 &= (1-f_y) n_{00} + f_y n_{10} \\
n_1 &= (1-f_y) n_{01} + f_y n_{11} \\
\\
noise(x, y, z) &= (1-f_z) n_0 + f_z n_1
\end{aligned}
$$

The interpolations are still performed one dimension at a time, four along x followed by two along y, and finally one along z to compute the final noise value.



Successive interpolations in 3-D by the classic Perlin noise algorithm: first along x (orange circles), then along y (green circles) and finally along z (black circle).

All computations are performed in the same manner as before, but there's quite a lot more of them now. A cube has $2^3 = 8$ corners, and the eight scalar products are performed with gradient vectors that have 3 components, each of which also needs to be generated. The complexity of 3-D noise is still not a big deal – the equations above specify about 45 scalar multiplications – but we might want to extend it further. A straightforward way to create an animated 3-D noise is to use a 4-D noise function and make the fourth coordinate depend on time: $noise(x, y, z, t)$. This isn't always a great idea, but it's one way of doing it. However, for each dimension we add, the classic noise algorithm requires more than twice as much work. This is not optimal, and there are better strategies for generating higher-dimensional noise as well as animated noise. More on this in the next chapter.

Things to consider adding to the presentation in this chapter:

~~List the desirable traits of an ideal noise function, with credit to Perlin. Or save that for the next chapter?~~ *On second thought, no.*

Additive iterative frequency synthesis with fake fractal sums of different scales ("FBM"), with **examples**. (Perhaps present the examples early as a teaser?)

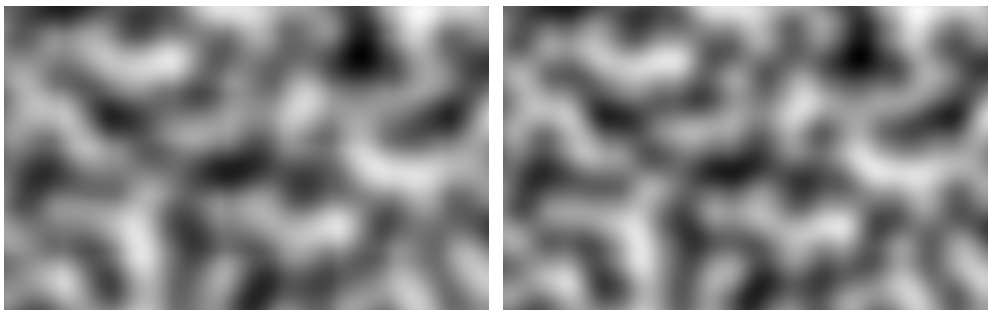
Anti-aliasing by frequency clamping of terms (revisit of the AA chapter).

10 Noises

Perlin's original gradient noise has been immensely useful. Considering how old it is, and how crude computer graphics was back in 1985, it's pretty amazing how much it does right. However, as even Ken Perlin himself has pointed out, it has several traits that, in retrospect, are undesirable. Over the years, several better versions of noise have been presented to fix those flaws. Perlin's original gradient noise function, as presented in the previous chapter, is still useful – and it's still being used a *lot* – but these days there are improved variants that should at least be considered, because they address some fundamental problems with the function.

Creases

The interpolation between adjacent gradients is performed in a manner that creates discontinuities in the second order derivatives of the noise. That might not seem like a big deal, but it's actually visible to the naked eye if you look at one single low-frequency component of noise. When noise is used to displace a surface or create bump maps, its derivative is what determines the intensity, and then these discontinuities show up clearly as creases along the boundaries of the grid cells.



Second order discontinuities in classic 2-D noise. ***Left:*** *problem.* ***Right:*** *solution.*

The solution to this is to use a "blend curve" for the interpolation that is smoother at the endpoints. The third degree curve $3t^2 - 2t^3$ used by ***smoothstep*** has zero slope at its endpoints, but its second derivative is non-zero. We can construct a polynomial $f(t)$ that has zero slope *and* zero second derivative at both endpoints. The constraints would be:

$f(0)=0, f(1)=1, f'(0)=0, f'(1)=0, f''(0)=0, f''(1)=0$

Applying these constraints to a general fifth-degree polynomial with six coefficients, $At^5+Bt^4+Ct^3+Dt^2+Et+F$, working out the first and second derivatives and solving an equation system with six unknowns gives us the solution:

$f(t)=6t^5-15t^4+10t^3=((6t-15)t+10)t^3$

This improved interpolation was suggested by Ken Perlin already in 2002 [“Improving Noise”, https://doi.org/10.1145/566654.566636], and it has made its way into most commercial implementations of Perlin noise. It’s slighty more work to evaluate the fifth degree polynomial (five multiplications instead of three), but computing the interpolation weights is only a minor part of the algorithm. Considering the clear visual improvement provided by this single change, there is no reason *not* to use it, except for rare cases where you might want to preserve the *exact* look of an old shader that depends on the details of some historic implementation of noise. (Such strong dependencies are best avoided.)

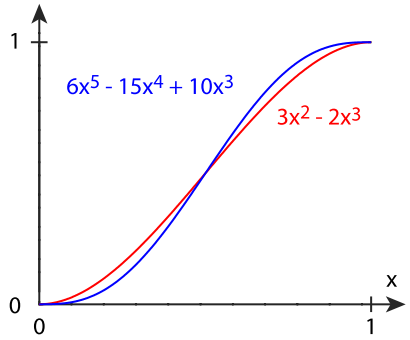
As a side note, we could define a function like ***smoothstep*** which uses the fifth degree polynomial for the transition. This function is sometimes given the playful name ***smootherstep***, and it’s not a bad name at all. A “smoother smoothstep” is quite useful in some cases, most notably when creating procedural bump maps.

$$smootherstep(a,b,x)=\begin{cases} 0 & for\ x\le a \\ curve & for\ a<x<b \\ 1 & for\ x\ge b \end{cases}$$

where *curve* is the polynomial:

$t=(x-a)/(b-a)$

$6t^5-15t^4+10t^3$

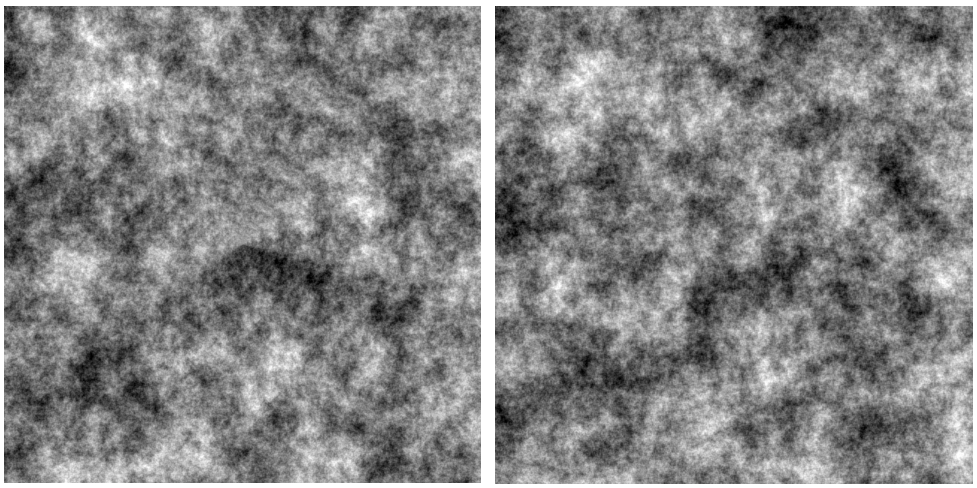


The “smootherstep” function, with second order continuity at its endpoints

Zero crossings and scaling artifacts

For gradient noise, the forced zero crossings at integer coordinates make the function “phase-locked”, to borrow a term from signal processing, and you need to take care to avoid correlation artifacts when adding several scaled versions of noise together. Consider not scaling with exactly 2.0 for each octave so that you don’t end up having all terms contribute nothing but zeroes to some points. You should also add a *translation* to each term to avoid having identical but scaled copies of

the noise pattern add up at the origin. It doesn't hurt to rotate each term as well, if you can afford the extra computations. You could even consider warping the grid somewhat, by adding noise to the texture coordinates before evaluating some terms. If done in a reasonably subtle manner, this will jitter the zero crossings away from the regular grid without significantly altering the spectral properties of the noise. Translations are the most important thing, though, and also easiest to do. It doesn't matter much what the translations are, as long as they are reasonably different non-integer numbers that are uncorrelated with the scaling and make the noise pattern different between terms. If you scale the terms around a common point, and that point is within view, there will be artifacts.



Left: *"Zoom" artifacts at the origin (center) from adding scaled but otherwise identical copies of noise.* ***Right:*** *Translation of each term removes the artifacts.*

These remedies are fixes – workarounds for inherent issues with the noise algorithm. More fundamental solutions have been proposed, like using an irregular grid for the synthesis. Most of the alternative algorithms that have been proposed over the years are not in common use, because gradient noise on a regular grid is fast and *good enough* for most purposes. However, as the performance of GPUs continues to increase, it's becoming possible to focus more on quality than speed. There are quite a few algorithms waiting in the wings to find their way into mainstream use. Gradient noise is, quite frankly, a cheat. Admittedly, computer graphics is all about cheating and hoping that it won't be noticed, but it can be good to know that there are *better* cheats out there if you need them. Two of them, *Gabor noise* and s*parse convolution noise*, will be described briefly in chapter 12 .

Anisotropy

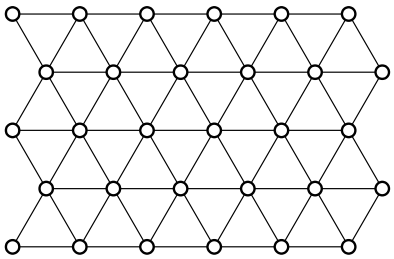
When looking at a single instance of noise, the square grid causes anisotropy in the final noise pattern. The grid is not exactly *visible*, but you can tell the direction of the (x, y) principal axes. What we would like is for the grid to be more uniform and less visible to the naked eye. A hexagonal grid is a better fit in both respects.

Before we present the solution to this problem, let's first describe two more problems that are addressed by the same solution: *simplex noise*.

Complexity

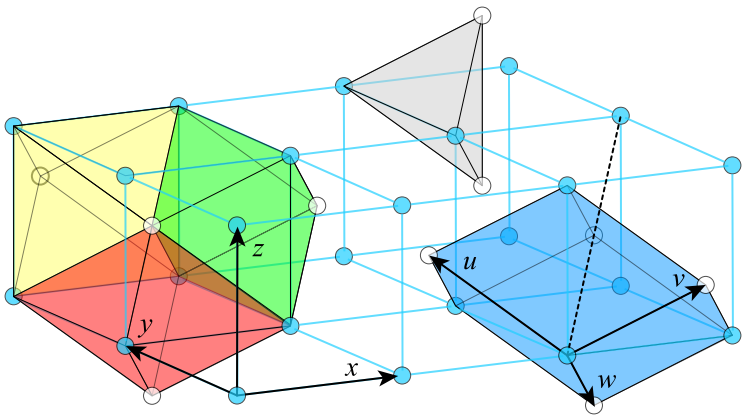
The grid cells of classic Perlin noise are squares, cubes or hypercubes, and the number of neighbors that need to be evaluated increases exponentially as 2^N with the number of dimensions N: 1-D nose requires two neighbors, 2-D requires four, 3-D requires eight and 4-D requires sixteen. Adding a dimension doubles the complexity. This makes 4-D and even 3-D noise rather expensive to compute.

There are tilings that have cells with fewer corners. The shape with the fewest corners that tiles N-D space is called an N-simplex. For 1-D, it's still a line, but for 2-D, it's a *triangle* instead of a square. Equilateral triangles can tile the 2-D plane, and the centerpoints of a hexagonal tiling (or a "staggered rectangular" tiling) can be connected to make a triangular *simplex grid* in 2-D.



Simplex tiling of equilateral triangles in 2-D

In 3-D, the simplex shape is the polyhedron with the least amount of vertices that encloses a volume: a *tetrahedron*, with 4 vertices. A regular tetrahedron won't tile 3-D space, but a tetrahedron with slightly different lengths for its sides will. Recall from the hexagonal tiling in chapter 5 that we can regard the vertices of the tiling of triangles in 2-D as two interleaved rectangular grids. The simplex tiling of tetrahedra in 3-D can be drawn on two interleaved cubical grids 3-D grid, each with grid spacing 1.0, and offset 0.5 from each other in all three dimensions. This is difficult to visualize in a 2-D still image, but here's an attempt at doing it anyway:



Simplex tiling of tetrahedra in 3-D. For explanation, see the text.

In this image, the two cubical grids are the blue circles and the white circles, offset one half grid spacing from each other. Each tetrahedron has two vertices from each grid (gray shape at the top center). To tile 3-D space without gaps, we need to use the tetrahedral cell in different rotations. One way of constructing the tiling is to first create octahedra from four tetrahedra. Three different orientations of those octahedra can then be used to tile 3-D space (yellow, red, green shapes to the left).

A much less obvious way to construct the tiling, but a way that is also more useful, is to instead join *six* tetrahedra that share an edge along one of the diagonals that connect one point from either grid (blue shape on the right). Their combined shape is a “squashed cube”, a parallel epiped with all sides the same length, which tiles 3-D space neatly with only translations of itself. (The formal name for this shape is *trigonal trapezohedron*, but very few people know that. Even I keep forgetting it.) We will make use of this alternate formulation of the tiling shortly.

The tetrahedra are not uniform, but the distance between vertices is at least more uniform than the distances between vertices in a cube. A cube with side length 1 has a side of length 1 and a diagonal of length $\sqrt{(3)}$, a relative difference of 1.732, while the sides of our tiling tetrahedra are 1 and $\sqrt{(3)}/2$, a relative difference of only 1.155. The 3-D simplex grid has a more uniform spacing than a cubical grid.

Without even trying to visualize it, let’s just say that there is a similar simplex tiling also for 4-D space, using the four-dimensional 4-simplex with 5 vertices, *and* for any higher dimension as well, although the simplices (yes, that’s the awkward plural form) get more irregular, more squashed, as we move to higher dimensions.

A simplex in N dimensions has $N+1$ corners. Compared to a regular Cartesian grid which has cells with 2^N corners, this is a big improvement. Already for 2-D we have one less corner to care about, 3 compared to 4. For 3-D, we have 4 neighbors

instead of 8, which is only half as many, and for 4-D we have 5 neighbors instead of 16. If we were to compute our noise in a simplex grid, higher dimensional noise functions could be made significantly more efficient.

For 2-D and 3-D, the simplex tilings are very useful indeed as the foundations for a noise function. The 4-D simplex grid has some issues with non-uniformity, but it's still useful. The uniformity gets progressively worse as we move to higher dimensions, but they can be solved by using several staggered simplex grids, or using more than a single simplex cell to evaluate noise. Noise in higher than 4-D is very rarely needed, and those rare needs can often be worked around, so this is not really a problem, but it deserves mentioning that simplex grids are not *universally* better than Cartesian grids. In higher dimensions, it also gets progressively more difficult to work out which vertices make up the simplex containing a certain point, whereas the cell corners for a Cartesian grid cell can be enumerated very easily by simply rounding the coordinates up and down in each dimension separately.

Differentiability

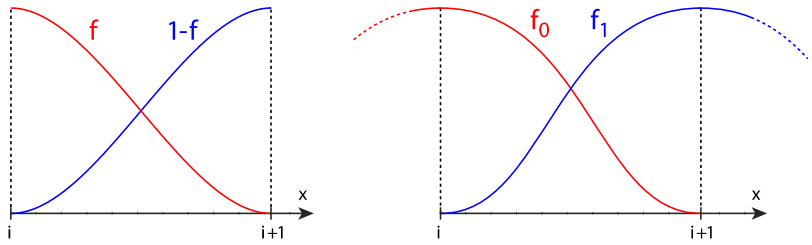
In many situations, it's useful to know not only the value, but also the *derivative* of the noise function. You can always compute several noise values and approximate the derivative with a finite difference, but that requires computing noise for several points – at least one extra point per dimension. Looking at how noise is computed, it's perfectly possible to work out the analytical derivative and compute it *exactly* (to within the available numerical accuracy) with only a modest amount of extra work. Most of the computations that are required can be identified as sub-expressions in the noise function and re-used, and computing the exact partial derivatives along each of the dimensions amounts to a lot less work than doing an approximation with one finite difference for each dimension. Finite differences are required only if you don't have any access to the inner workings of the noise function. We are implementing our own noise in shader code, so we can do it right.

Unfortunately, for classic noise, the successive interpolations with polynomials end up making the closed expression for the final noise value a polynomial expression of uncomfortably high degree with lots of mixed terms, and computing its analytical derivative requires quite a lot of multiplications. It's still less work than faking it with finite differences, but it's not exactly convenient. The culprit is the successive interpolation. 3-D noise is a 16^{th} degree polynomial in x, y and z with rather messy partial derivatives.

An alternative to successive interpolation is *sparse convolution*. This is a method borrowed from signal processing, where interpolation between sample points is replaced by simple summation of *kernels of influence*, one around each point. In signal processing, this is also called *impulse summation*. Instead of mixing two

values according to a fade curve, we multiply each of the values by a decay function that decreases with the distance from the point, and then we just add the two results. In 1-D, this is simply a matter of replacing the interpolation weights with two independent factors f_0 and f_1 instead of using f and $1-f$. The final step in the 1-D noise generation in chapter 9 (page 108) could be changed like this:

Interpolation ("mix")	*Sparse convolution (impulse summation)*
$f=(3t^2-2t^3)=(3-2t)t^2$ (or similar) $noise(i+t)=(1-f)n_i+f\,n_{i+1}$	$f_0=(1-t^2)^2$, $f_1=(1-(1-t)^2)^2$ noise(i+t) = $f_0 n_i + f_1 n_{i+1}$



***Left:** interpolation weights. **Right:** impulse summation weights.*

The decay functions here were chosen somewhat arbitrarily, but they are simple expressions that drop from 1 to 0 when t goes from 0 to 1 or vice versa, and they have zero slope at both ends. Note that while trivially $f+(1-f)=1$, the sum of weights f_0+f_1 is not necessarily constant. In this case, at $t=0.5$ we have $f_0+f_1=9/8$, which means that we could end up making the midpoint value slightly larger than either of the two base values, which is improper behavior for an interpolation. However, keep in mind that we are *generating noise*, not carefully reconstructing a sampled signal, so this is rarely an issue. The advantage here is that the two points don't have to be at distance 1 from each other for the blending to work. We could, for example, imagine jittering the points a little to make the zero crossings appear at less regular positions. Interpolation could be modified to support that as well, but impulse summation works *without* modification.

Now, in 1-D this is not obviously better, because even with the mildly unsuitable but easy to compute decay function $(1-t^2)^2$, it's slightly more work to compute two independent decay curves f_0 and f_1 instead of just computing f and $1-f$. The new blending is not obviously better, or smoother, or easier to understand, but for 2-D and up we start seeing benefits: the decay can be made isotropic (circularly symmetric), and we can allow for any number of kernels to influence each point. In particular, we can use a simplex grid with *fewer* corners for each grid cell and cover N-D space reasonably well with kernels of influence from the corners of a simplex.

This leads us to another, more recent idea from Ken Perlin which is obvious only in hindsight: *simplex noise*.

2-D simplex noise

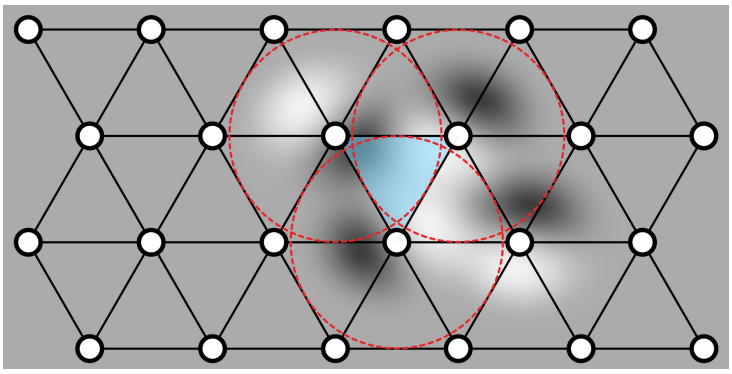
Simplex noise is an invention by Ken Perlin, but it's impossible to cite a formal publication for it. It was presented informally as part of the notes for a course on hardware shading at the Siggraph 2001 conference, but it's not part of the conference proceedings, and it was never peer reviewed. The description can still (2024) be found online on the university web address of the course organiser, Marc Olano [https://www.csee.umbc.edu/~olano/s2002c36/ch02.pdf], but it's not "published" in the scientific meaning of the word, and the description isn't as clear as one would have wished for. Simplex noise was patented, which ideally *should* be a kind of formal public disclosure, but the patent document is deliberately unreadable to obscure the fact that the patented invention is nothing but a mathematical algorithm implemented by a program for a general computer, neither of which are patentable "as such" according to the US Patent Office guidelines. Patent lawyers have routinely worked around that for decades, and the patent situation in the software field is a huge mess. The patent for simplex noise ended up in the hands of a so-called "patent holding company" (a more accurate name for it is "patent troll") which caused people not to use it. This is the exact *opposite* of what a patent is supposed to do. However, in 2022 the patent finally expired, and there is no longer any potential legal implications for implementing it or using it.

Simplex noise is a great idea – it was just presented in a manner that made it tank with would-be implementers. An unencumbered version called *OpenSimplex noise* was created to work around the patent, but it was slower and never caught on.

In retrospect, the patent was most likely never valid anyway, and wouldn't have survived a challenge in court – not because it's a patent on software and mathematics (there are tens of thousands of those, and patent offices all over the world keep granting them), but because its primary claim is demonstrably false. The claimed subject matter is "a method for generating images … producing the images with texture that do not have visible grid artifacts" *(sic)*. Simplex noise has clearly *visible* grid artifacts, they're just not as *objectionable* as classic noise, and the coordinate system is tilted out of axis alignment to typically hide them better.

Anyway, the patent debacle around simplex noise is now history, and we can present the algorithm without reservations. It's a *gradient noise* using the tools presented above: a *simplex grid* instead of a Cartesian one, and *impulse summation* instead of interpolation. Perlin's original version (and the patent) covered only 3-D noise, and the supposedly standard reference implementation had a bug, so we are not actually presenting his algorithm here, just using the same general idea.

The 2-D simplex grid is a tiling of triangles. We assign a circular region of influence around each grid point that covers as much as possible of the neighboring triangles without spilling over into more distant simplex cells:



Kernels for 2-D simplex noise (red circles). Most of the area of one simplex (blue shade) is influenced by three kernels. For clarity, only a few kernels are shown.

The kernels (the "wiggles" inside each circle) consist of two parts: one gradient extrapolation like for classic noise, and one *circularly symmetric* decay function. Assuming that the vector from simplex corner i to the point being evaluated is $\bar{v}_i$, that the associated pseudo-random gradient is $\bar{g}_i$, and that the grid is composed of equilateral triangles with side length 1, we have:

For corners $i=\{0,1,2\}$:

$r^2=\|\bar{v}_i\|^2=\bar{v}_i\cdot\bar{v}_i$ *(squared distance from grid point)*

$w=max(0.75-r^2,0)$ *(radial decay, drops to zero at opposite triangle edge)*

$n_i=w^4(\bar{g}_i\cdot\bar{v}_i)$ *(smoother radial decay times linear ramp)*

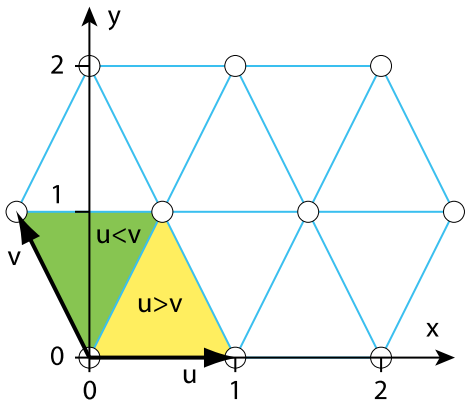
$noise=\sum n_i=n_0+n_1+n_2$ *(final value is just the sum of the three contributions)*

The decay function w^4 has zero derivative and zero second derivative both at the center $r=0$ and where it drops to 0 at $r=\sqrt{3}/2$. It's also simple to compute: $w^4=(w^2)^2$ which requires only two multiplications.

All variants of gradient noise, of which simplex noise is one, look good and are nicely band limited when the gradients have unit length but pseudo-random direction. This isn't the *only* option, but it's the most common choice. We can do this in some smart manner like in the code shown previously for classic 2-D noise, but we can also assume that the ***sin*** and ***cos*** functions are hardware accelerated and generate a unit vector as $(\cos\phi,\sin\phi)$ for a pseudo-random angle $0\le\phi<2\pi$.

We have skipped one part of the algorithm, and it's an important one: we need to find the three corners for the simplex that contains our point. There are several ways to do this, but we will present one that is simple and easy to understand.

The hexagonal tiling from chapter 5 had the right kind of grid, but there we were only concerned with finding the closest point of the grid. Now we want to find the *three* closest points. The most efficient manner is to do it in two steps. First, we transform our point from Cartesian (x, y) coordinates to a skewed coordinate system (u, v) where u and x are parallel but v is rotated 30 degrees counter-clockwise from y. While we're at it, we cheat and scale the grid a bit in the y direction so that our grid points fall on integer positions in y. The grid will be slightly anisotropic for it, but the transformations are easier, both for humans to understand and for computers to compute. The kernels of support around each point will still be circularly symmetric – they will only be slightly wider apart in the y direction. The impact on the noise pattern is unnoticeable to human eyes.



A modified simplex grid, and skewed coordinates (u,v) to describe it. The shaded region has $(floor(u), floor(v))=(0,0)$. Yellow is u>v, green is u<v.

The skewed coordinate system can be thought of as a tiling of parallel-epiped cells built from two triangles each. A point with coordinates (u, v) is inside the cell with its lower left vertex at $(u_0, v_0)=(floor(u), floor(v))$. The local coordinates for the point within this cell are $(u_f, v_f)=(u-u_0, v-v_0)$. If $u_f>v_f$, the point is in the right triangle towards positive u, and if $u_f<v_f$, it's in the left triangle towards positive v. Thus, after the transformation, we can enumerate the three corners of the simplex either as (u_0, v_0), (u_0, v_0+1), (u_0+1, v_0+1) or (u_0, v_0), (u_0+1, v_0), (u_0+1, v_0+1) in the (u, v) system, depending on which of u_f and v_f is larger. We generate the gradients based on hash values computed from the (u, v) integer coordinates of the three corners. Then we can transform the corners back to the (x, y) system to compute the noise contributions from each.

Transforming back and forth might sound like a lot of work just to find the closest neighbors, but for our slightly non-uniform grid, both the forward and inverse transformations are very simple:

$$\begin{cases} u = x + y/2 \\ v = y \end{cases} \qquad \begin{cases} x = u - v/2 \\ y = v \end{cases}$$

And with that, we have the entire algorithm figured out. The steps are, in order:

1) Transform the input point to skewed space.
2) Find which cell we are in. (This gives us 4 corners.)
3) Determine which triangle we are in. (This gives us the 3 closest corners.)
4) Generate a pseudo-random gradient for each of the 3 corners.
5) Transform the 3 corners back to unskewed space.
6) Compute the radial falloff from each corner.
7) Extrapolate the gradient ramp from each corner.
8) Compute and sum up contributions from all 3 corners.
9) Scale the output to the desired range of values.

Implemented in GLSL, a 2-D simplex noise function is quite compact:

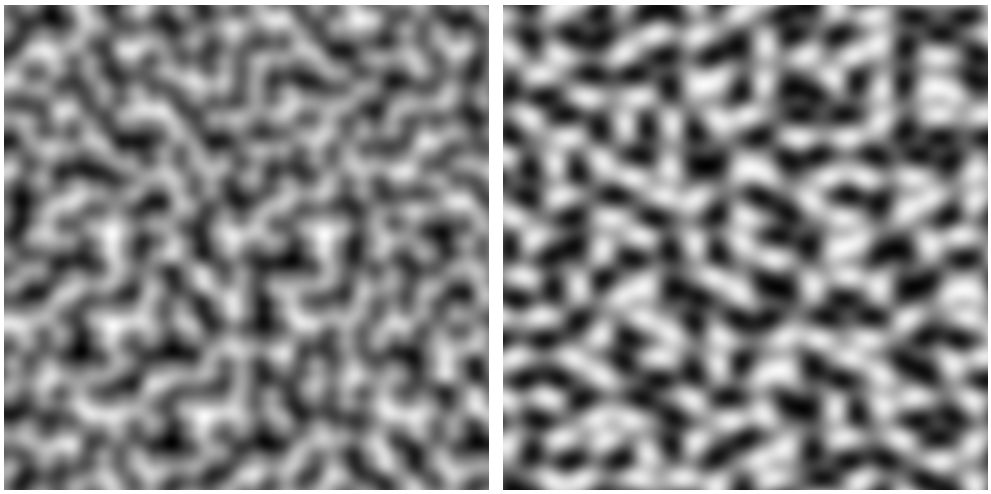
```
//
// simple simplex noise (c) Stefan Gustavson,
// version 2024-11-12, published under CC-BY-SA 4.0
// https://creativecommons.org/licenses/by-sa/4.0/
//

float ssnoise(vec2 p) {
   // Staggered grid, points at integer y and integer/half x
   // (Yields a slightly non-uniform triangular/hex grid.)
   vec2 uv = vec2(p.x+p.y*0.5, p.y + 0.001); // Transform to skewed space
   vec2 i0 = floor(uv); // Tells us which grid quadrilateral contains p
   vec2 f0 = fract(uv); // Tells us which of the two triangles contains p
   float cmp = step(f0.y, f0.x); // Equivalent to "(f0.y>f0.x)?1.0:0.0"
   vec2 o1 = vec2(cmp, 1.0-cmp); // o1 is the offset to the second corner
   vec2 i1 = i0 + o1, i2 = i0 + 1.0; // Grid coordinates, used for hash
   vec2 p0 = vec2(i0.x - i0.y*0.5, i0.y); // Transform back to p space
   vec2 p1 = vec2(p0.x + o1.x - o1.y*0.5, p0.y + o1.y);
   vec2 p2 = vec2(p0.x + 0.5, p0.y + 1.0);
   vec2 v0 = p - p0, v1 = p - p1, v2 = p - p2; // Vectors from corners to p
```

```
  // Compute a simple hash for each of the three corners
  vec3 iu = vec3(i0.x, i1.x, i2.x); // Hash coords
  vec3 iv = vec3(i0.y, i1.y, i2.y);
  vec3 hash = mod(iu, 289.0); // Mod to avoid truncations below
  hash = mod((hash*51.0 + 2.0)*hash + iv, 289.0); // These are not great,
  hash = mod((hash*34.0 + 10.0)*hash, 289.0);     // but "good enough"
  vec3 phi = hash*0.07482; // Scale the hash (and jumble it some more)

  // Generate three gradients
  vec3 gx = cos(phi); // sin and cos are usually fast these days
  vec3 gy = sin(phi);
  vec2 g0 = vec2(gx.x, gy.x);
  vec2 g1 = vec2(gx.y, gy.y);
  vec2 g2 = vec2(gx.z, gy.z);

  // Compute the kernels of influence from each corner
  vec3 w = 0.8 - vec3(dot(v0, v0), dot(v1, v1), dot(v2, v2)); // radial decay
  w = max(w, 0.0); // Cut off the negative part
  vec3 w2 = w*w; vec3 w4 = w2*w2; // w4 is our final radial weight
  vec3 gdotv = vec3(dot(g0, v0), dot(g1, v1), dot(g2, v2)); // extrapolated ramps

  // Compute and sum up the three contributions in one go
  float n = dot(w4, gdotv);
  return 10.9*n; // Scale the noise value to span the range [1,1]
}
```

In total, not counting multiplications with 0.5 or 2.0 as actual multiplications, this implementation performs 46 multiplications plus three ***sin*** and ***cos*** calls (sine and cosine for the same angle can usually be computed in parallel in a modern GPU) to compute one 2-D noise value, including the work to generate gradients. It's less work than the implementation of classic noise shown in the previous chapter, but not by a lot. Having only one gradient less to generate doesn't make a huge difference. However, 2-D simplex noise is at least *somewhat* faster to compute than classic noise, and it's better-looking in several respects.

Note that the radial decay is computed slightly differently from the equations we presented previously, with 0.8 instead of 0.75 for the maximum. This is because the stretched grid can handle a slightly larger circular region of support for the kernels.

After all these equations and code, it's high time we showed a picture of what simplex noise actually looks like. It should be noted that because simplex noise is a completely different algorithm from classic noise, it also has a different look to it:



Left: *classic 2-D Perlin noise.* ***Right:*** *2-D simplex noise.*

Simplex noise uses impulse summation of small round wiggles, giving it more of a wobbly-blobby character than the crawly-wormy pattern of classic noise (yes, those are proper technical terms, trust me, I'm a doctor), and the pattern is therefore more narrowly band limited than classic noise. In many circumstances, this is actually an advantage, but the different look needs to be taken into account when creating patterns with it. Simplex noise is not a direct drop-in replacement for classic noise, and most patterns designed with classic noise will need some work to look similar when using simplex noise as the random-looking band limited base function for "controlled randomness".

Classic, interpolated noise on a square grid is by no means obsolete. Sometimes it's the better option for purely aesthetic reasons, and it's still a very useful function.

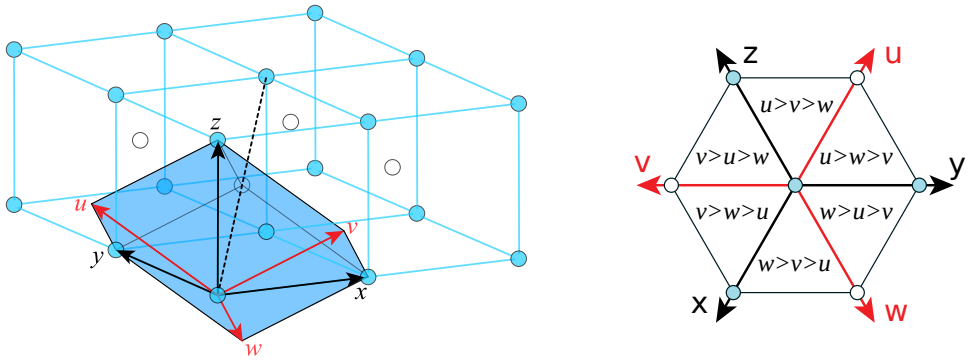
3-D simplex noise

The algorithm for simplex noise in 3-D is basically the same as for 2-D:

1) Transform the input point to skewed space.
2) Find which cell we are in. (This gives us 8 corners.)
3) Determine which simplex we are in. (This gives us the 4 closest corners.)
4) Generate a pseudo-random gradient for each of the 4 corners.
5) Transform the 4 corners back to unskewed space.
6) Compute the radial falloff from each corner.
7) Extrapolate the gradient ramp from each corner.
8) Compute and sum up contributions from all 4 corners.
9) Scale the output to the desired range of values.

We have one more vertex to consider and three components for the gradients and all coordinates, but the extension is reasonably straightforward.

Let's revisit and revise part of our previous figure explaining the 3-D simplex grid:



Determining which simplex contains a point in 3-D. ***Left:*** *skewed coordinates* (u,v,w) *and a "squashed cube" (blue shape) containing six simplices.* ***Right:*** *view of those six simplices from along the diagonal x=y=z (dotted line on left).*

We use a skewed coordinate system (u,v,w) to quickly determine which cluster of six tetrahedra we are in by simple floor operations of each coordinate, and then we decide which tetrahedron contains the point by comparing the magnitudes of the local coordinates within that "squashed cube" cell. The corners of the simplex are traced out by moving from the local origin and then taking one step along each of the coordinates (u,v,w), in order from largest to smallest magnitude. For example, if $u>v>w$, the simplex corners are (u_0,v_0,w_0), (u_0+1,v_0,w_0), (u_0+1,v_0+1,w_0) and finally (u_0+1,v_0+1,w_0+1). The first and last corners are common to all six simplices in the same cell, but the other two need to be determined by comparisons.

The transformations between the two spaces are almost as simple as in 2-D:

$$\begin{cases} u=y+z \\ v=x+z \\ w=x+y \end{cases} \qquad \begin{cases} x=(-u+v+w)/2 \\ y=(u-v+w)/2 \\ z=(u+v-w)/2 \end{cases}$$

To determine which simplex we are in, we transform the position of the input point (x,y,z) to the skewed system, giving us new coordinates (u,v,w). The top and bottom vertices of the 6-simplex cell are common to all six simplices:

$(u_0,v_0,w_0)=(floor(u),floor(v),floor(w))$
$(u_3,v_3,w_3)=(u_0+1,v_0+1,w_0+1)$

To find the two other vertices, we look at the position of the input point relative to the cell origin:

$$(u_f, v_f, w_f) = (u - u_0, v - v_0, w - w_0) = (fract(u), fract(v), fract(w))$$

To iterate through each of the vertices from (u_0, v_0, w_0) to (u_3, v_3, w_3), we first add 1 to the component with the largest coordinate value, which gives us (u_1, v_1, w_1), and then add 1 to the coordinate with the second largest coordinate value, yielding (u_2, v_2, w_2). This requires components to be compared to each other and the two increments being computed accordingly. Doing that in a shader-friendly manner without ***if-else*** statements gets a bit weird, but it *can* be done with ***step*** functions.

Once we have all four vertices (u_i, v_i, w_i), we transform them back to the original space, giving us (x_i, y_i, z_i), compute the vectors $\bar{v}_i = (x - x_i, y - y_i, z - z_i)$ and select pseudo-random gradients $\bar{g}_i$ for each.

The equations to then compute the noise value are nearly identical to the 2-D case, except for the number of components and the number of terms:

For corners $i = 0,1,2,3$:
$r^2 = \|\bar{v}_i\|^2 = \bar{v}_i \cdot \bar{v}_i$
$w = max(0.5 - r^2, 0)$
$n_i = w^3 (\bar{g}_i \cdot \bar{v}_i)$
$noise = \sum n_i = n_0 + n_1 + n_2 + n_3$

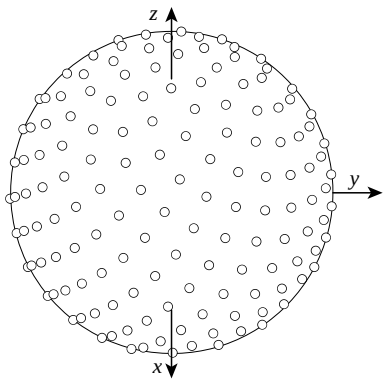
The radial decay for 3-D is chosen as w^3, because w^4 would drop off a bit too rapidly. w^3 looks better and is more than enough to make the second derivative zero at the perimeter $r = 1/\sqrt{2}$. Perlin's original algorithm used $(0.6 - r^2)^4$, which looks similar but is non-zero at the simplex boundaries. (This is the bug we mentioned.)

Gradients in 3-D can be chosen in different ways, but if we want unit length vectors in a uniform distribution, one good way is to pick points on a *Fibonacci sphere*. This allows us to map from a single hash value to a pseudo-random point on a sphere. A Fibonacci sphere is a Fibonacci spiral on a spherical surface, which doesn't say much unless you know what a Fibonacci spiral is. Both of these concepts will be treated in the next chapter. For now, we just present the equation for mapping an integer k to one of N vectors on a sphere, without explanation:

$$k \in \{0, \ldots, N-1\} \qquad w = \sin\phi = \sqrt{1 - z^2}$$
$$\theta = 4\pi k / (1 + \sqrt{5}) \qquad x = w \cos\theta$$
$$z = \cos\phi = 1 - 2(k + 1/2)/N \qquad y = w \sin\theta$$

Note that only one sin/cos pair is being computed explicitly, for x and y. The angle ϕ is implicit, expressed in terms of its cosine which is equal to z, and $w = \sin\phi$ is generated by knowing that $(\sin\phi)^2 + (\cos\phi)^2 = 1$ for any ϕ. To generate gradients on a sphere in this manner is only slightly more work than to generate points on a circle in 2-D.

The points end up being very evenly distributed across the surface of the sphere:



Fibonacci sphere with 289 points (only the front-facing points are shown).

Another useful method is to map points in a square to points on an octahedron, and normalize the length of the generated vectors. This was the method used in several speed-optimized gradient noise functions published by McEwan et al in 2012 [https://doi.org/10.1080/2151237X.2012.649621], which are still useful and *very* fast on modern hardware. However, given the speed of trigonometric functions in most modern GPUs, a Fibonacci sphere is efficient enough, it's more flexible, and it yields a more uniform distribution.

Derivatives of simplex noise

We claimed that the derivative of simplex noise was easier to compute than for classic noise. The main reason for that is that the differentiation is distributive on addition: the derivative of a sum is the sum of the derivatives of each of the terms. The partial derivatives for 2-D and 3-D simplex noise are in fact easy to work out, and they contain partial expressions that can be re-used from the computation of the noise value. The analytical gradients of our particular versions of noise are:

For each term of 2-D simplex noise	*For each term of 3-D simplex noise*
$n_i = w^4(\bar{g}_i \cdot \bar{v}_i)$	$n_i = w^3(\bar{g}_i \cdot \bar{v}_i)$
$\nabla n_i = w^4 \bar{g}_i - 8w^3(\bar{g}_i \cdot \bar{v}_i)\bar{v}_i$	$\nabla n_i = w^3 \bar{g}_i - 6w^2(\bar{g}_i \cdot \bar{v}_i)\bar{v}_i$

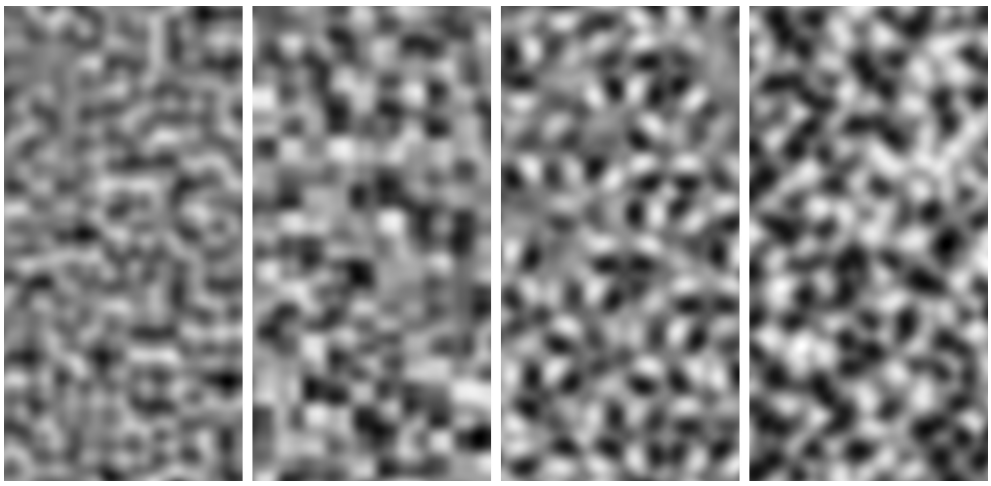
Slices through higher-dimensional space

(Note to self: that would be a great name for a trance album.)

3-D texturing of arbitrary curved surfaces is performed by defining a pattern in 3-D and letting the surface cut through it. While this is a very convenient method in some cases, it has a potential problem of its own. There is a theorem from multi-dimensional signal processing, the *projection-slice theorem*, explaining in detail what the problem is, but it requires reasoning in the Fourier domain, and we don't really need to go into that.

It's enough if you realize that cutting through a 3-D noise function with a 2-D slice occasionally ends up cutting through the center of a noise wiggle at near right angle with its gradient, and then the wiggle is just a blurry gray blob. That part of the pattern will show up as featureless region in the 2-D pattern. Even if we use some other function instead of gradient noise, a slice in an arbitrary direction could end up cutting through the pattern at any angle with the local gradient of the pattern. Making a cut parallel to the gradient will make the pattern in the cut have strong contrast, but making a cut perpendicular to the gradient will result in a low contrast. The same thing happens if we make a 3-D "slice" through a 4-D noise function, or a 1-D slice through a 2-D function. While not necessarily objectionable, it should be kept in mind that lower-dimensional slices through a band limited pattern defined in higher-dimensional space introduces low frequency content (larger features), even if the higher-dimensional signal is strictly band limited.

Just to make it clear that a 2-D slice through 3-D noise looks different from 2-D noise, and that it can present some nasty surprises, consider the images below.



Planar cuts through 3-D noise at constant z. ***From left to right:*** *classic noise at z=0, classic noise at z=0.5, simplex noise at z=0, simplex noise at z=0.25.*

The classic interpolated noise is literally interpolating between two different noise patterns at $z=0.5$, so the second panel above shows the average of two different pseudo-random patterns looking like panel 1. The result is more blurry, has less contrast, and has considerably stronger grid artifacts because the zero crossings of the two patterns are exactly aligned. Simplex noise has a denser and more uniform grid and is less sensitive to the exact position of the slice, but the pattern at $z=0$ (and $z=0.5$, and $z=1$, and so on) cuts directly through the grid points, making the wiggles appear in a square grid which is too regular for human eyes to accept it as "random-looking". The contrast of each wiggle varies with the angle between its generating gradient and the cut plane, but it's not enough. At $z=0.25$, the plane passes between grid points and has a much better look. Slices that are not axis-aligned end up having a bit of both of these patterns, and that usually looks good as well. Slices with nearly constant x (or y, or z) can be problematic, though.

These are worst case scenarios, but they can, and do, occur in practice. Try to avoid making axis-aligned planar cuts through 3-D noise. Try to tilt or rotate the slice if possible. Ken Perlin's original formulation of simplex noise used a coordinate mapping that rotated the simplex grid out of alignment with (x, y, z) and dodged the problem with axis-aligned planar cuts. It still has the same artifacts, but they occur in planes at odd angles that are less likely to be used in practice.

Also, if you want to make a 2-D pattern, don't forget about 2-D noise. 3-D texture mapping isn't always the best idea if there's an easy way to create 2-D texture mapping coordinates. It *could* be, but it's not guaranteed. 2-D shader functions are usually both faster to compute and easier to design.

A modern, flexible noise function

Present psrdnoise. Don't go into details on the algorithm, but list the code and point out that it's free for any use, with credit.

Multifractals

Present how earlier terms can influence later terms in a fake fractal sum.

~~List flaws with the classic "Perlin noise" (with citation of Perlin himself)~~

~~More modern variants of Perlin noise: Improved Noise, Simplex Noise (cite Perlin), once again with detailed explanations of the algorithms. Present also the equations and code for computing partial derivatives of simplex noise.~~

~~Hardware friendly versions of simplex noise (cite McEwan and Gustavson)~~

Present code for simplex noise ~~in 2-D~~ and 3-D ~~(not 4-D, the code is too messy)~~, and use them in a few simple but creative examples

Using previous terms (large scale) to influence later terms (small scale), creating "multifractals". (One quick hack: multiply noise components in [0,1] instead of adding noise components in [-1,1])

Flow noise as a variation of this. ~~Comment on the Fourier slice theorem in some manner that doesn't require a Fourier domain discussion.~~ Point to the difference between rotating gradients and moving a 2-D slice in the third dimension. Demonstrate the utility of analytic derivatives when creating pushy-swirly effects with flow noise. ("pushy-swirly" is a word)

11 Scatterings

Often lumped together with “noise”, but named and treated separately here, because they are a completely different class of patterns.

Cellular noise and (simpler, more hardware friendly) variations of it, with an overview explanation of the original (cite Worley) and detailed explanation of a more hardware friendly version (Cite old RSL code example? Cite myself?)

Drawing Voronoi boundary lines of (near) constant width

Fibonacci spirals on a plane and on a sphere (tricky inverse, but doable, cite Keinert et al and provide code)

Hybrids: Sparse convolution noise (cite Wyvill), Gabor noise (cite Lagae), “Voronoise” (cite Quilez)

12 Amalgamation

(Find a better choice of word for the chapter title? “Mash-ups” is too informal. I might have to reconsider my choice of setting single-word titles to all chapters.)

Combining all the tools presented so far

Concrete example: Noise to make wobbly lines, torn edges and irregularly shaped spots (halftone dots, blood spatter)

Concrete example: Cell-dependent “seed” (offset) for noise to differentiate adjacent patterned tiles in regular (square) and irregular (Worley) tilings.

Perhaps also craquelure: Voronoi cells within Voronoi cells, with top level cells setting the seed for the next level of cracks, like above.

13 Animation

Using time as a parameter: conceptually simple but requires care to get it right

“Mistreating time as space” (removed from chapter 10) is a relevant heading here.

Waves, single-frequency and in fractal sums, both directional (wind waves) and non-directional (residual ripples)

Turbulence (fire, smoke) by fractal sums

Rotating gradient noise with successive displacement of later terms in fractal sums (“flow noise”), cite Perlin and Gustavson/McEwan, present psrdnoise23 and give some examples. Motivate the options for periodicity and analytical derivatives.

For flow noise, present both “turbulence” (increasing speed with smaller sizes) and “foam” (the opposite, making the small features more persistent).

Cover “curl noise” here as well, not just for particle animations, but also for area-preserving (and volume-preserving) coordinate distorsions.

Mistreating time as space

Section cut from chapter "Noises", edit to fit in here

To make animated noise, the traditional way is to make a slice through a higher dimensional noise and move the position of that slice through the extra dimension over time. This introduces a lot of low frequency content into the noise pattern and ruins its band limited character. There is a theorem from multi-dimensional signal processing that states this, the *projection-slice theorem*, but explaining it in detail requires reasoning in the Fourier domain, and we don't really need to go into that.

It's enough if you realize that cutting through a 3-D noise function with a 2-D slice occasionally ends up cutting through the center of a noise wiggle at near right angle with its gradient, and then the wiggle is just a blurry gray blob. That part of the pattern will show up as featureless region in the 2-D pattern. The same thing happens if we make a 3-D "slice" through a 4-D noise function, or a 1-D slice through a 2-D function. While not necessarily objectionable, this is not really what we want. We want our noise to change while also remaining band limited.

One solution, again from the mind of Ken Perlin, is to animate the underlying gradients of a gradient noise algorithm. In 2-D, we can rotate them all with a common angle, and the visual result will be a "wiggly-swirly noise". The look is difficult to explain in words, and it's impossible to show it in still images, but the supplementary material to this book contains animated figures. ***(not done yet)***

The idea of rotating gradients can be extended to 3-D noise as well. Not obviously, and not easily, but an algorithm for it was published by Gustavson and McEwan in 2022 [https://jcgt.org/published/0011/01/02/]. (Yes, that – this – Gustavson. I feel no shame in promoting my own relevant peer-reviewed work.)

14 Displacement

Bump mapping (just modify the normals), cite Blinn

Displacement mapping (actually move the surface, possibly not along the normal), mention Renderman displacement shaders, compare to GLSL vertex shaders

Recomputation of normals: the procedural way (clean, quick, quite simple), cite Mikkelsen

Present an example of combined displacement and bump mapping

Point out that displacements can work on already displaced surfaces to create overhangs

Mention “smootherstep” (fifth degree blend, second order continuity at ends)

Relate hardware displacements to software: dynamic fine-grained tessellation at render time is considerably more complicated in a hardware-constrained context (given the inherent assumptions made by the current hardware architecture).

Mention what can be done by geometry and tessellation shaders, and what their limitations are, but don’t get into details. That’s a subject for Part 2.

15 Scale

Brief explanation of number representations: fixed point (scaled integers) and floating point (64 and 32 bits, 16 bits and less)

Numerical precision of mapping coordinates when zooming in on detail. (Some supposedly floating point properties, like linear interpolated texture lookups, are often computed in fixed point arithmetic, and it shows. Demonstrate that?)

Numerical precision of mapping and displacement when working on a planetary scale

64-bit double precision allows a planetary scale sphere with its local origin in the center to have a coordinate precision of about one nanometer, but a cheaper way to handle the problem is to use a local origin at the observer and stick to 32-bit floats. Surface features near the camera can then be represented accurately down to microscopic scales, while distant features have lower precision, in the order of one meter.

Mention galactic and universe scales? High precision (128 bit) coordinates, transformations to local systems

Caching the initial terms of fractal sums in extreme close-ups

16 Hybrids

Mixing procedural shaders with image texture data

Pre-computing procedural texture images at runtime (tiling patterns and cyclic animations). Other methods than point by point shaders can be used, including Fourier synthesis and other methods from digital image processing. Show tiling filtered Fourier noise as an example.

Scrolling textures, displacement maps, layered textures (Nintendo style)
(Look at https://www.youtube.com/watch?v=8rCRsOLiO7k and the recent diploma work for inspiration)

Detail textures (utilize untapped computational capacity in the hardware)

Using texture storage for procedural parameters (including painted blends)

Using texture data as a starting point (example: shapes from distance fields)