Noise is Beautiful

Part 1: Procedural textures

by Stefan Gustavson

Draft version 11/1/24, 18:51:47

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.

No facets, no texels, no jaggies!

Foreword

This book was written to fill a void. For decades, my own 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 Admittedly, some chapters had become outdated and would have needed a rewrite, but most of the book was still good, and it was our only option.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inexplicably chose to resurrect the *first* edition of the book, which was published in 1991 and is both badly outdated and considerably less well written.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

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, is an attempt at not setting a boring title to a book about a fun subject. A more formal, academic title would have been “Procedural Methods for Computer Graphics”, because that’s what the book is 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, “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.

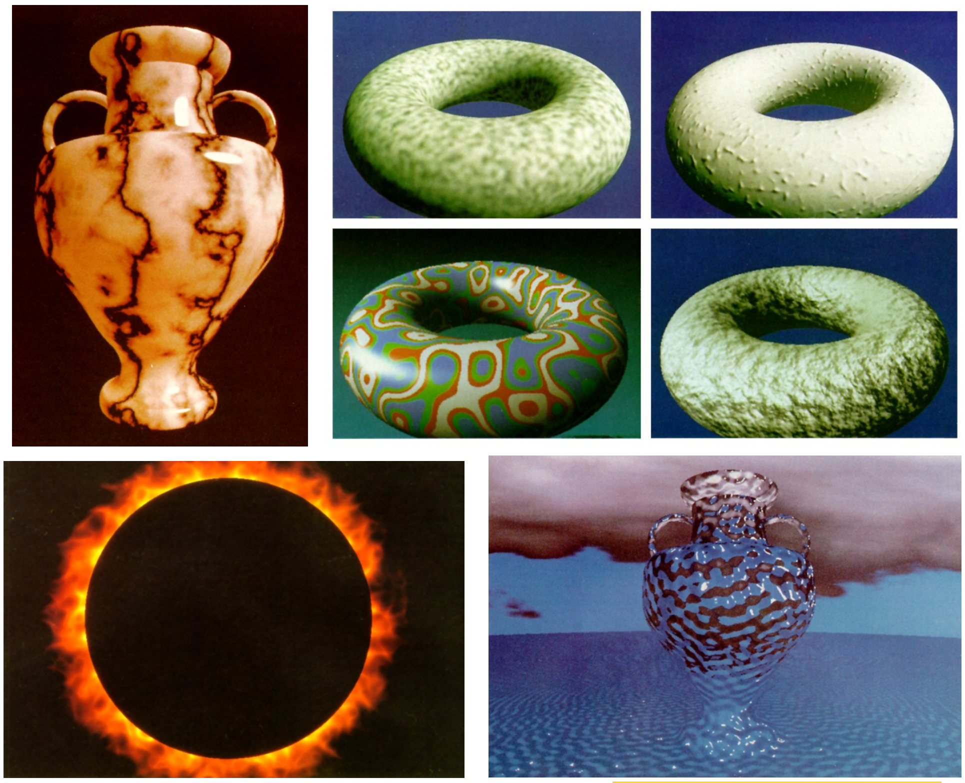
"Let us consult The Writ Of Common Wisdom!"

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" 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.***

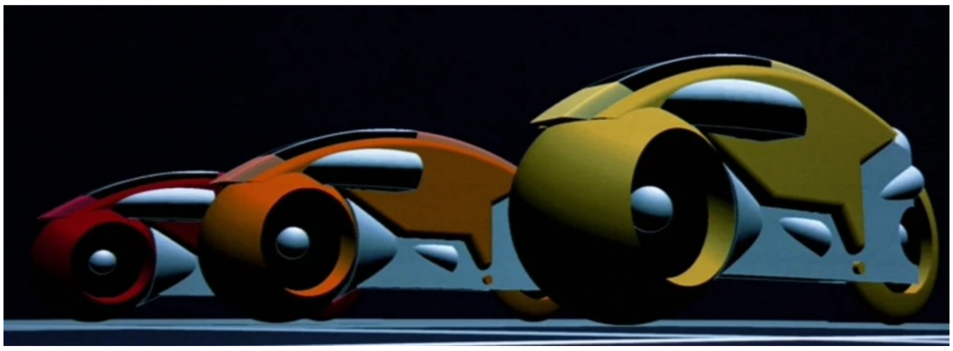
Around the same time, procedural texturing was picked up and used commercially by the Computer Graphics Group at Lucasfilm, which was later to become Pixar. The public release of Pixar's rendering software *Renderman* in 1989 made procedural texturing and modeling available to a rapidly growing community 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 geometric 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".
Permission will be difficult to get – Disney is notoriously restrictive. 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.

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 designed by NVIDIA with a somewhat C-like syntax. Because Cg programs compiled into standardized ARB assembly instructions, Cg could be used also to program other manufacturers’ hardware. Microsoft created 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.) 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 necessarily need 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 now reached a point in technical development where the visual quality of 3-D graphics is more dependent on the production budget than 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 to make 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 software 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. 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 image 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 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 mathematical 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 quite a lot of mathematical operations between them, image-based textures can take a lot of time to process, and 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 be an obvious or 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 severely 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 some chip area was 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 would change the playing field quite dramatically in favor of procedural textures.

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 with only 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 you run it.

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”.

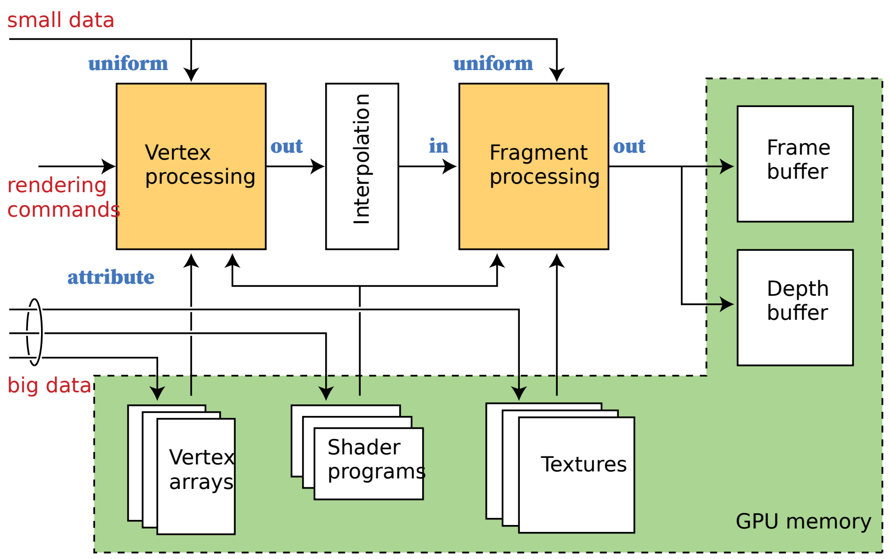
The art of not actually drawing

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 $F(x, y)$ that tells the renderer what color (or other local property) a point at the position (x, y) 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) mental 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 fun and 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:

$$x = x_0 + R\cos\phi$$
$$y = y_0 + R\sin\phi$$

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 that in pseudo-code would look something like this:

```
Set starting point to $(x_0+R, y_0)$
For angles $\phi$ from just above $0$ to $2\pi$:
    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 amount 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.

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’s also 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 smart and efficient in 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, inline assembly code could actually 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 level 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: use descriptive variable names, split long expressions into readable parts, stick to a commonly accepted coding style. 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 ) {               // Extra multiplication to save one square root
                return 1.0;
        } 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 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:

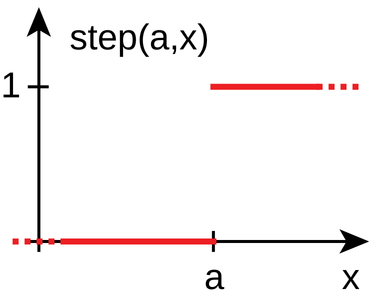
```
float inside;
if (r <= R) inside = 1.0;
else inside = 0.0;
```

```
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 (x, y) coordinate space of the shader. In most cases, this is what we want. The only situation where we would want one pixel wide lines is when we do schematic line drawings on a display with reasonably large pixels. 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 transform to screen space and use those coordinates for (x, y) 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*.

The next chapter is all about anti-aliasing. It covers other kinds of aliasing besides jagged edges, and explains what aliasing actually *is*, but let's at least introduce the subject here, because aliasing really has no place in modern computer graphics.

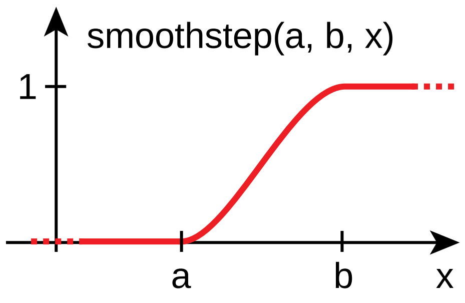
In traditional applications of 3-D computer graphics, anti-aliasing is the responsibility of the renderer, and it usually involves quite a lot of work. However, shaders are *functions* that can be programmed to change their behavior based on rendering resolution and viewing distance. This means that 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 shader to a nice, anti-aliased version. The function we will use in place of ***step*** is called ***smoothstep***:

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

where *curve* is a third degree interpolating polynomial:

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



*The **smoothstep** function*

Our circle shader doesn't need to change much at all 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 width ***w*** 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*. Fortunately, the transformations between shader space and device space are known at render time, and this can be used to compute a proper step width. However, that's not the way it's usually done. Shading languages have built-in mechanisms to make it 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 this 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 the next chapter.
        return 1.0 – smoothstep( d – w, d + w, R ); // It’s not actually magic.
}
```

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( d – w, d + w, R );
}
```

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 in a shader 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 something like:

```
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.

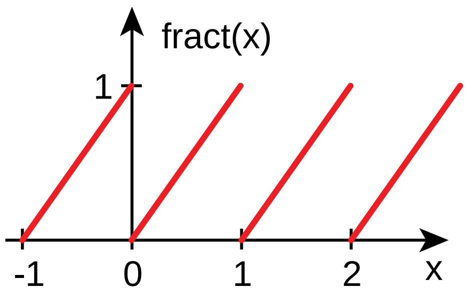
However, there are several disadvantages with using the ***sin*** function for making simple stripes. First, its period is 2π, a rather inconvenient irrational number, while we would typically want to create patterns where the period is some round number, like 1.0 or 0.25. Second, the relation between 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.)

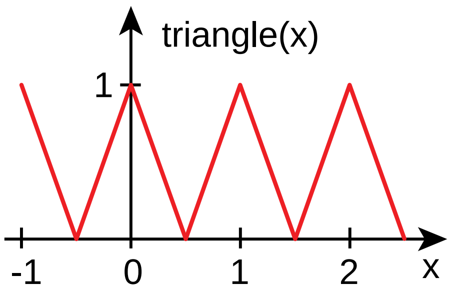
One flaw remains, however, and it concerns anti-aliasing. Where the ***step*** function acts on the upward ramp of the ***fract*** function, anti-aliasing is easy, but the ***fract*** function has a discontinuity at each integer, and discontinuities are notoriously prone to creating jaggies. A ***smoothstep*** 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.

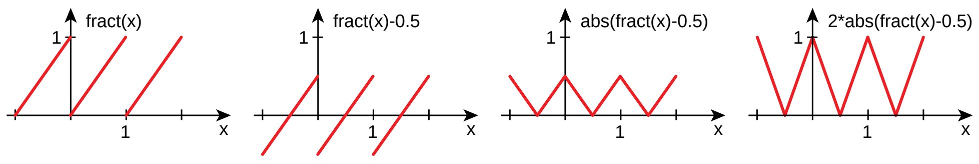
The most efficient way of creating a triangle wave is by the trick shown below. It's not completely obvious at first glance how it works, but it becomes clear if you plot the steps along the way.

$$triangle(x) = 2\left|\left(fract(x - 0.5)\right)\right|$$

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

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

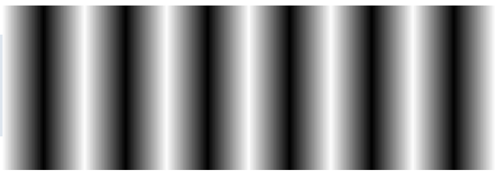




*A triangle wave function created from **fract** (and each step along the way)*

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 ) );
}
```

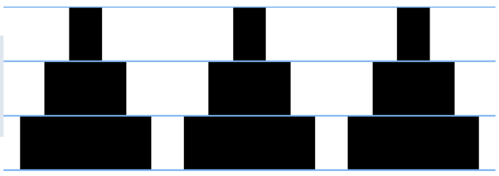


The triangle-wave function, as code and rendered to a grayscale image

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:

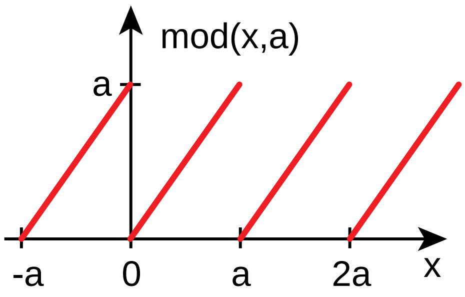
```
float stripes1 = step(0.2, triangle( x ) );
float stripes2 = step(0.5, triangle( x ) );
float stripes3 = step(0.8, triangle( x ) );
```



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 correct mathematical 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.

Optimization

Note that the modulo operation involves a division, and it takes some more work to compute compared to ***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*:

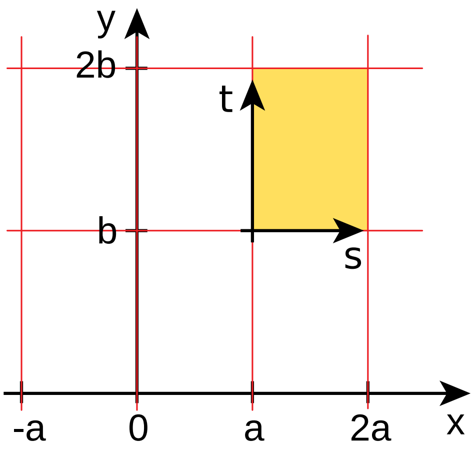
$$s = mod(x, a)$$
$$t = mod(y, b)$$

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.

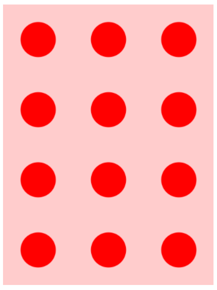
Polka-dots

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) (dots completely
// cover the plane). The dots will overlap if R>0.5.
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

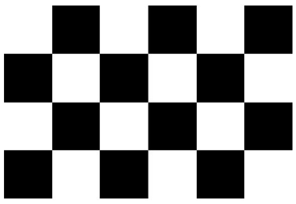
Note how the split into several lines made it possible to write detailed comments, and that the code is now a lot easier to read and understand, even without the comments. Try to resist the temptation to write “clever” one-liners. You will have a hard time understanding even your own code if you program like that, and it 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.

Except for one important thing: anti-aliasing. The pattern above looks fine as long as you keep it oriented along the pixels of the output device, but any rotation, particularly small and slow rotations, any camera motion, or perspective effect, or almost any kind of animation, will make the pattern look absolutely awful. In its current form, it's a function that's basically useless function for direct pattern generation. We need to do better.

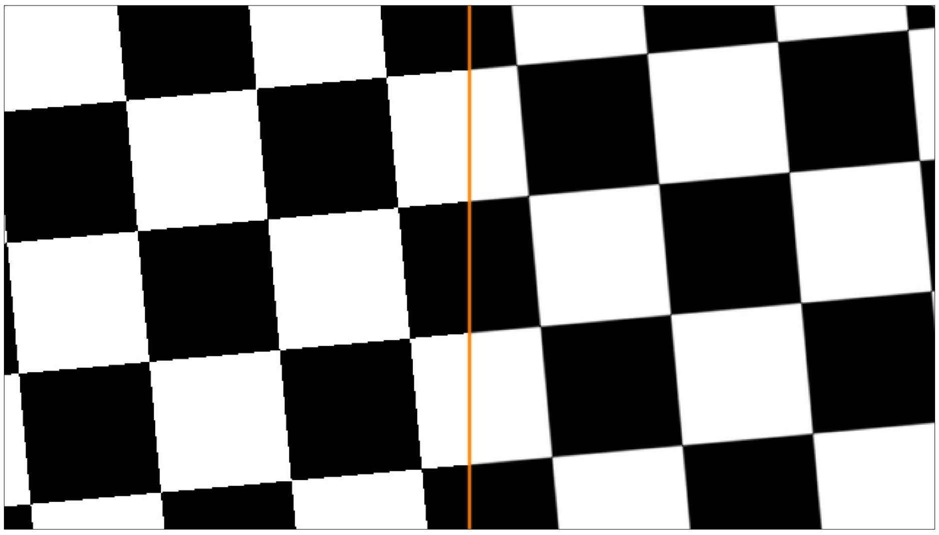
Anti-aliased checkerboard

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 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 just need a way to overlay the two sets of stripes, and perform a sort of "exclusive or" masking 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:

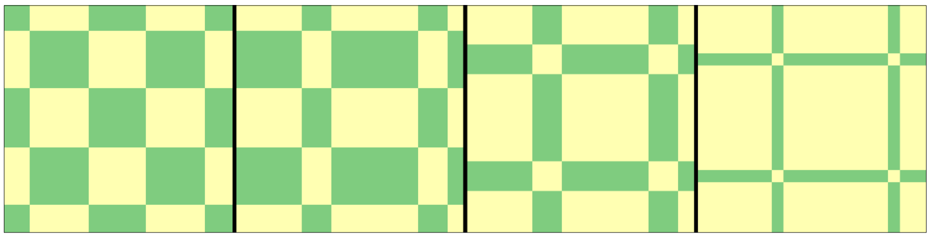
```
float aacheckers (vec2 p ) {
        vec2 ramps = 2.0 * abs( fract( p ) - 0.5); // "Triangle waves" in x & y
        vec2 stripes = step( vec2(0.5, 0.5), ramps ); // 50% stripes
        return abs( stripes.x - stripes.y ); // "XOR" at overlaps, preserving AA
}
```

This checkerboard pattern is created from step 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

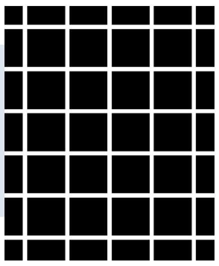
Now that you know how to do a checkerboard pattern that can be anti-aliased, *please* don't use the raw floor-mod version, except maybe for quick demos. It's simple code that's easy to remember, but apart from that it has no good qualities.

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***:

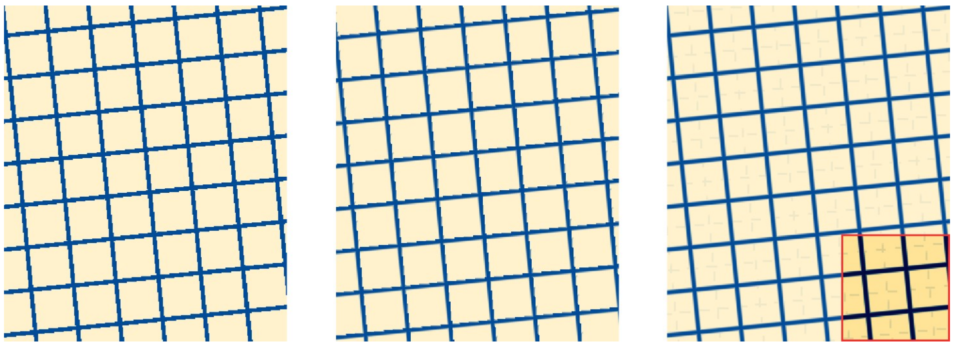
```
float gridlines( vec2 p ) {
        vec2 q = fract( p ); // Square grid of size 1x1
        vec2 grid = step( 0.9, q ); // Threshold x and y
        return max ( grid.x, grid.y ); // Overlay the two
}
```


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 1.0 to 0.0. 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.


Badly aliasing rotated grid (left), failed anti-aliasing of the naive code (center), and mostly successful anti-aliasing of the improved code (right).

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 wave” function from before:

```
// 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

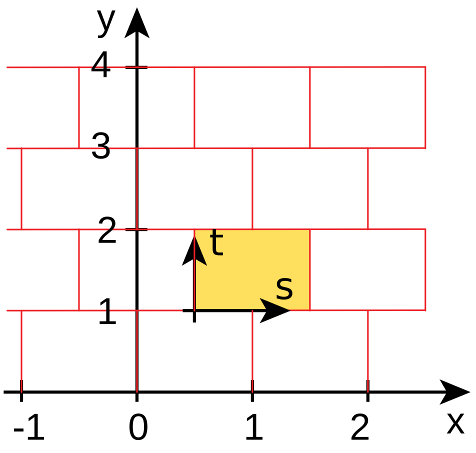
Variants of rectangular tiling are *skewed tiling*, where the tiles are not rectangles but parallelograms, and *staggered tiling* where successive rows or columns of rectangles are offset against each other. Skewed tiling will be treated extensively in the chapter "Noises", so let's save that for later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

The most common example of staggered tiling is brickwork, and a brick wall pattern is a classic example of a procedural shader, so let's just show that particular pattern here before we move on. Let's make the bricks have the value 1.0 and the mortar between them 0.0:

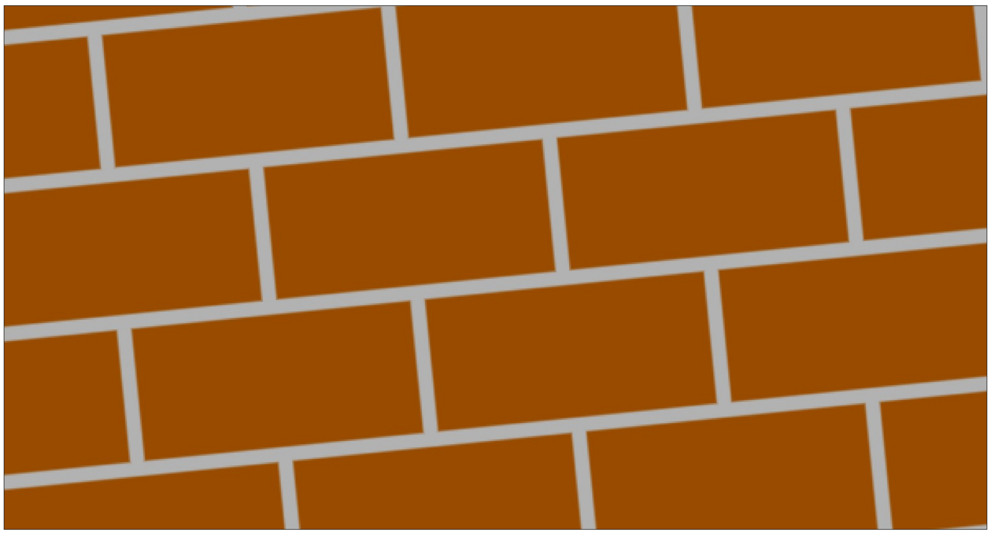
```
float bricks( vec2 p ) {
        vec2 q = fract( p + vec2( 0.5 * floor( p.y ), 0.0 ) ); // Staggered grid
        float brickx = step( 0.1, q.x ) - step( 0.9, q.x ); // vertical gaps
        float bricky = step( 0.1, q.y ) - step( 0.9, q.y ); // horizontal gaps
        return min( brickx, bricky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

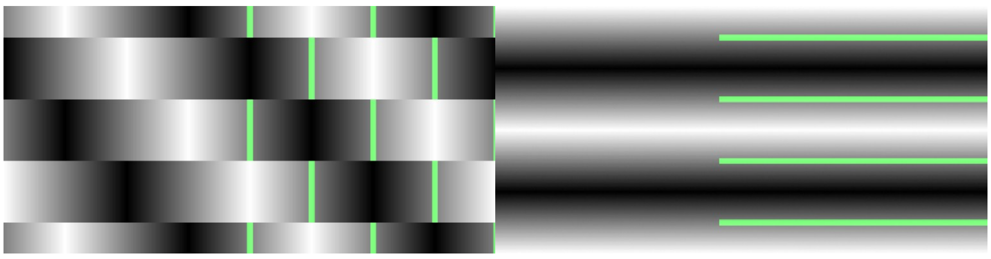


Anti-aliased brick pattern in color, rotated and scaled to make the bricks 2x wider

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 the chapter "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.

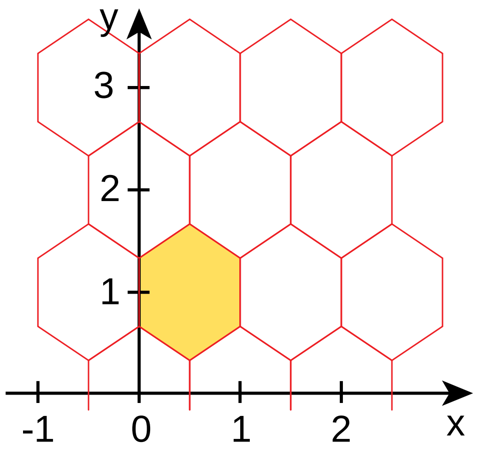
Hexagonal tiling

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. Before you have seen the solution laid out for you, it's far from obvious that what we show here is a convenient method of mapping to a hexagonal grid in a shader. Quite a lot of effort went into creating this mapping, 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 is not the absolutely most efficient one, but it's quite fast, and also reasonably easy to explain. Not easy, but *reasonably* easy.

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



One variant of hexagonal tiling

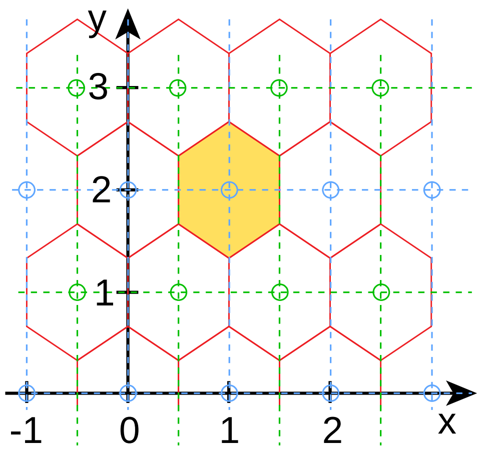
Expressing a hexagonal tiling in shader code is a bit tricky. Quite *literally* tricky, because there are certain tricks to doing it in an efficient manner. First, note the integer coordinates in the diagram above, and our choice of scaling. This tiling is not built from regular hexagons – they are slightly 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. That, in turn, can be used to compute the nearest midpoint of a hexagon from any point in the plane. The hexagons as such can then 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 messy.) In fact, we will go one step further and treat the staggered tiling as two overlapping rectangular tilings:

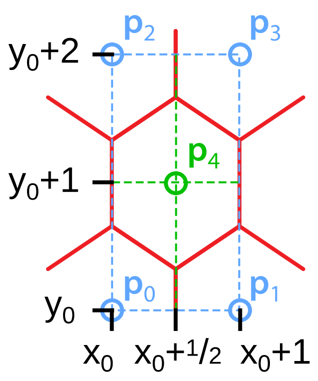
The blue grid has its vertices at integer coordinates in x and at even integers in y.

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 $\vec{p}=(x,y)$ 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 the closest midpoint of the hexagonal grid from any point in the cell.

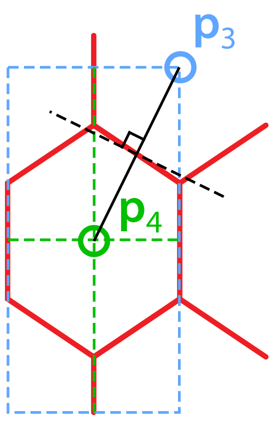


One cell of the blue grid, with numbered hexagon midpoints $\vec{p}_i$

For half of the area of the cell, $\vec{p}_4$ is closest, but towards each of the corners, either $\vec{p}_0$, $\vec{p}_1$, $\vec{p}_2$ or $\vec{p}_3$ 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 $\vec{p}_4$ and *one* other candidate. For a point (x, y) inside the cell, the lower left vertex is at $(x_0, y_0) = (2\,floor(x/2), floor(y))$, 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 $\vec{p}_4$. 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 $\vec{p}_4$ to the current point $\vec{p} = (x, y)$.

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 $\vec{p}_4$ 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 comes down to:

$(\vec{p}-\vec{p_4}) \bullet (\pm\frac{2}{3}, \pm 1) - \frac{2}{3} = 0$ ***The dot for the scalar product is way too big!***

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, $\vec{p}$ is closer to $\vec{p_4}$ than to the corner at $\vec{p_4} + (\pm 0.5, \pm 1)$.

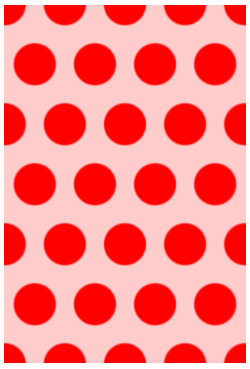
Let's express the entire algorithm as program code:

```
// 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 which of the two candidate points 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, it's not really all that many computations. Counting multiplications with powers of 2 as additions, because that's what they end up being in floating point arithmetic, the entire program code 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* it.

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
completely
// 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 the rather complicated algorithm, 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.

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 we *are* lecturing, but we don't want to annoy the reader. Not more than necessary.)

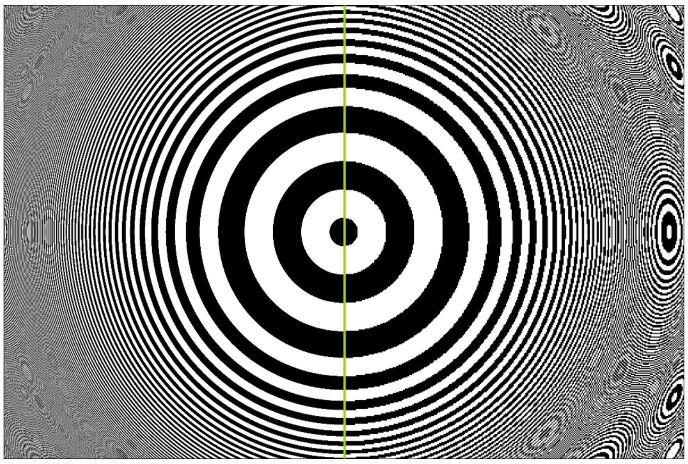
It cannot be overstated that in a procedural shader, aliasing *can* be avoided, and *should* be avoided in all situations where visual quality matters. Particularly when people write shaders without proper care, aliasing in surface patterns can be horrendous. This is acceptable for beginners and creative 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 undercooked 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.



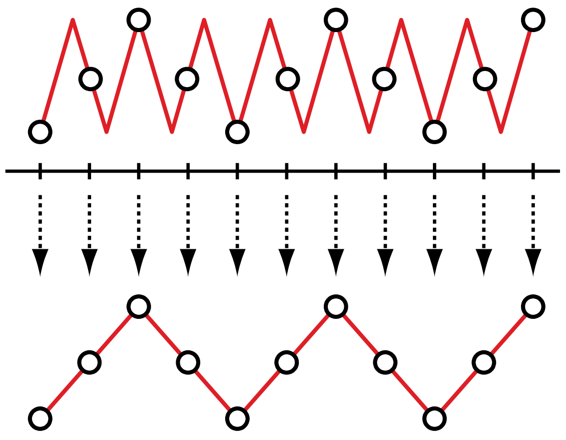
A pattern of concentric circles with the pattern growing denser with distance, at two different sampling frequencies (left: ~600 dpi, right: ~150 dpi).

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".

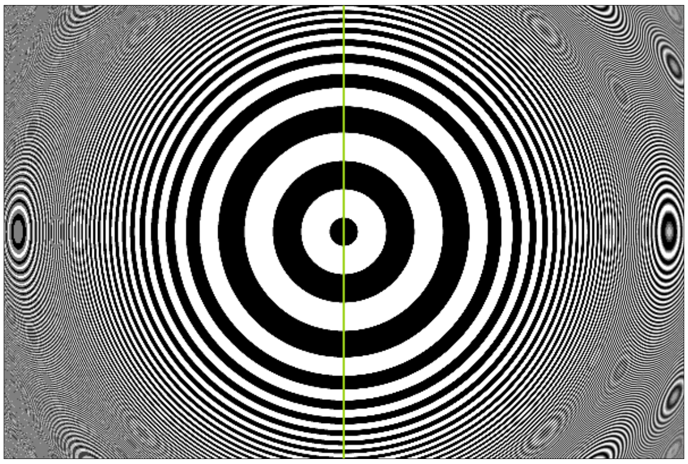

Undersampling of a signal, with aliasing as a result

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 crisp 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 pre-filtering to blur the pattern before point sampling.

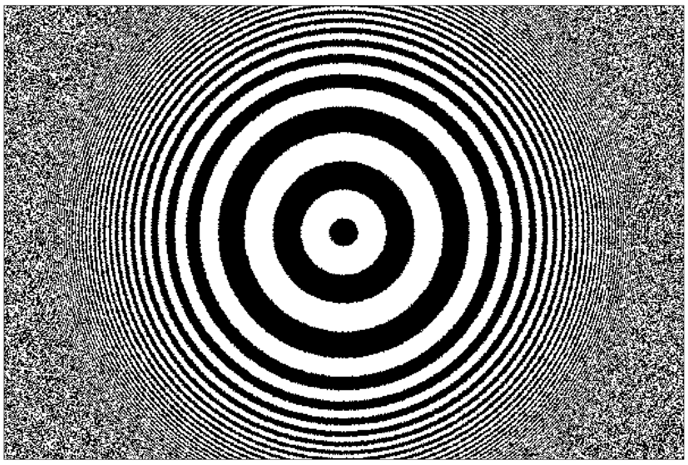
Oversampling, however, is usually not the best option for a procedural pattern. If the shader performs a lot of computations, executing it more than once for each pixel is not a great idea. Multi-sampling also doesn't eliminate aliasing – it just reduces it, and the quality improvement (expressed as 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 with 2x2 samples (left) and 4x4 samples per pixel (right)

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.

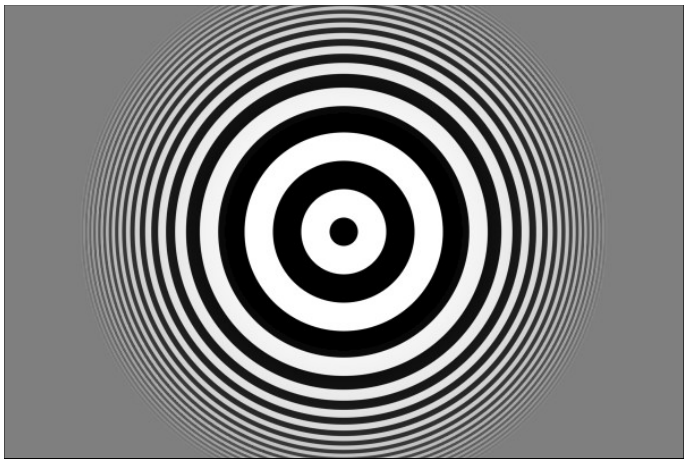


Jittered sampling

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 $\|\nabla F\|$. 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, an 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 );
}
```

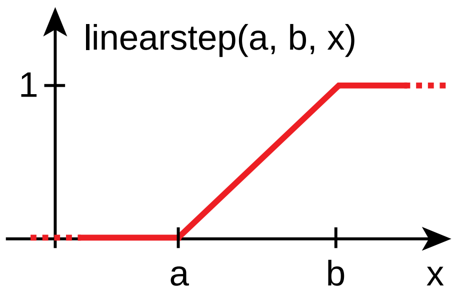
This is a straightforward implementation, but some tweaking can make it perform a little better. First, our use of ***fwidth*** makes the value for ***w*** anisotropic – it will depend on the angle of the gradient. This can be fixed by implementing our own version without the "speed cheat". Second, our 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.

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

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 its easy to make. 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;
}
```

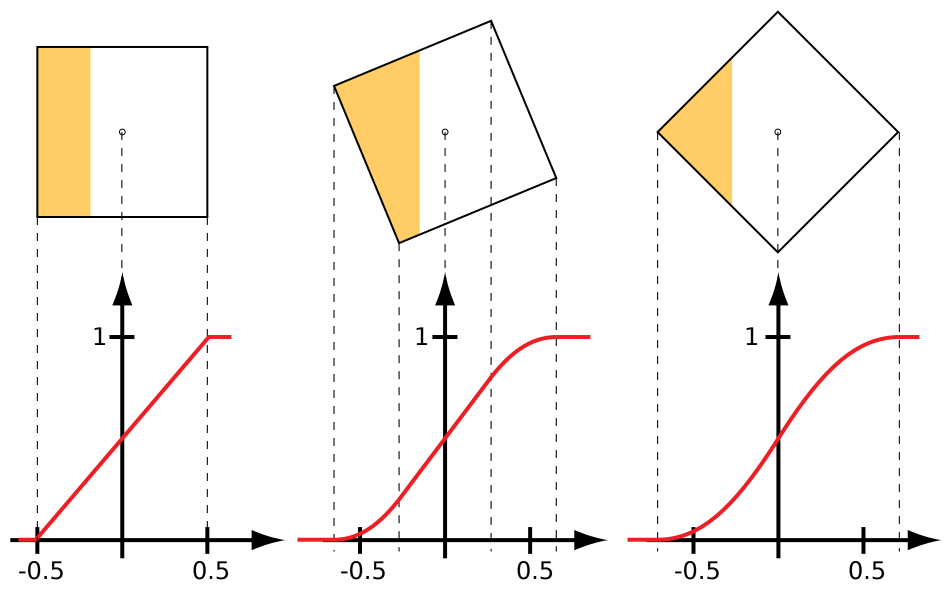
The ***clamp*** function in GLSL is most likely hardware accelerated, and easy for the compiler to optimize, so you are better off using the built-in version for speed rather than rolling your own version of it. The built-in version also has several overloaded versions for vectors and integer types,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



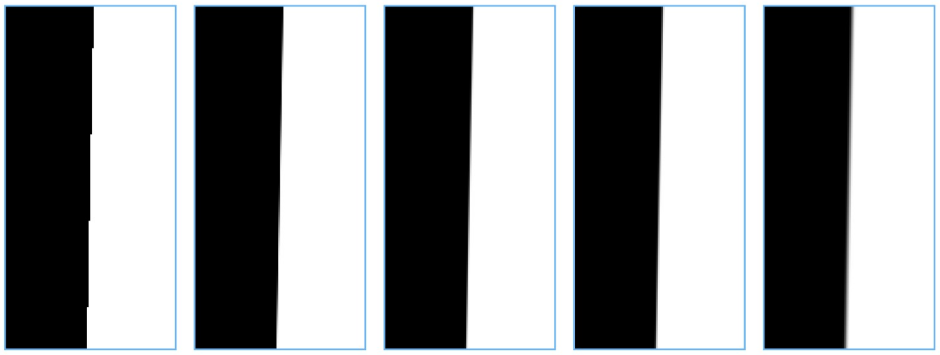
Area coverage as a function of the position of an edge relative to the pixel center. Left to right: axis-aligned edge, 22.5 degree edge, 45 degree edge.

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 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 its slight speedup 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 no clear downside to using this modified version of ***aastep*** instead of one that requires the threshold to be constant.

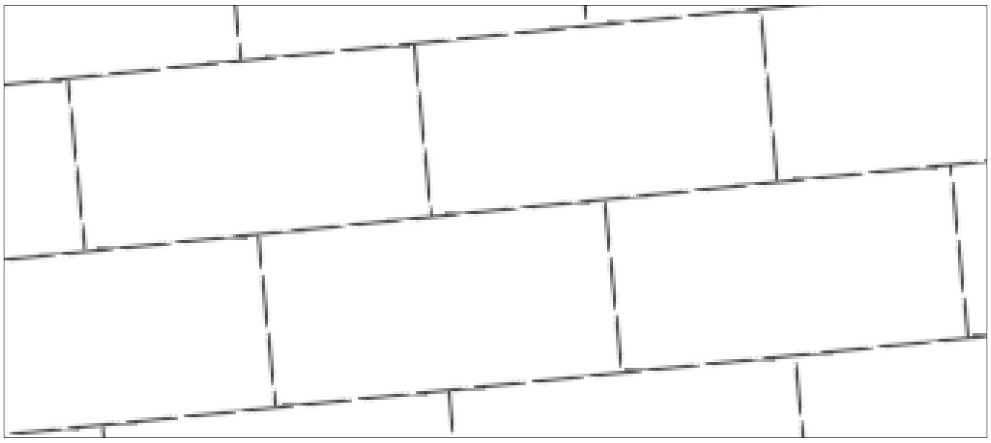
Brick pattern, revisited

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:

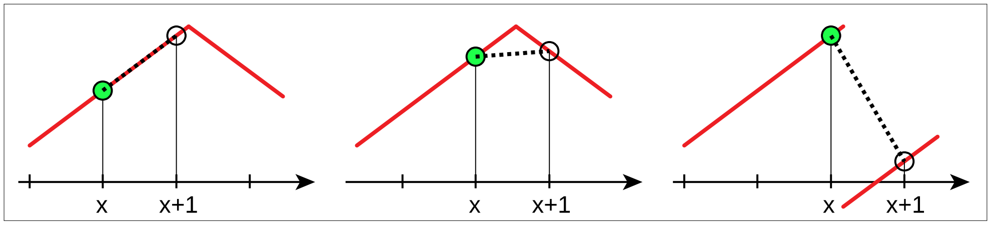
```
float bricks( vec2 p ) {
        vec2 q = fract( p + vec2( 0.5 * floor( p.y ), 0.0 ) ); // Staggered grid
        q = 2.0 * abs( q – 0.5 ); // Change from sawtooth to triangle
        float brickx = 1.0 – aastep( 0.99, q.x ); // vertical gaps
        float bricky = 1.0 – aastep( 0.98, q.y ); // horizontal 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 appropriate ***smoothstep***. We could also implement a hacked version of the function where we supply one function for the thresholding and another one 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 helper function that is at least somewhat re-useable:

```
// 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 );
}
```

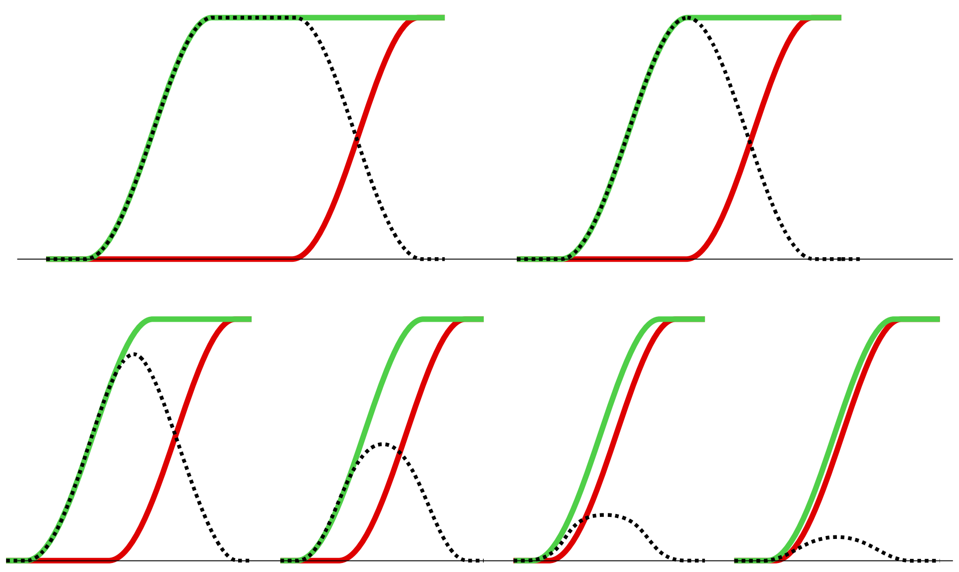
Using this modified function, the shader code would be:

```
// 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 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 $[0,1]$.

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? This is a good way to make very narrow lines fade away. What we are doing is take the difference between two ***smoothstep*** ramps, both with the same width and height, 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.



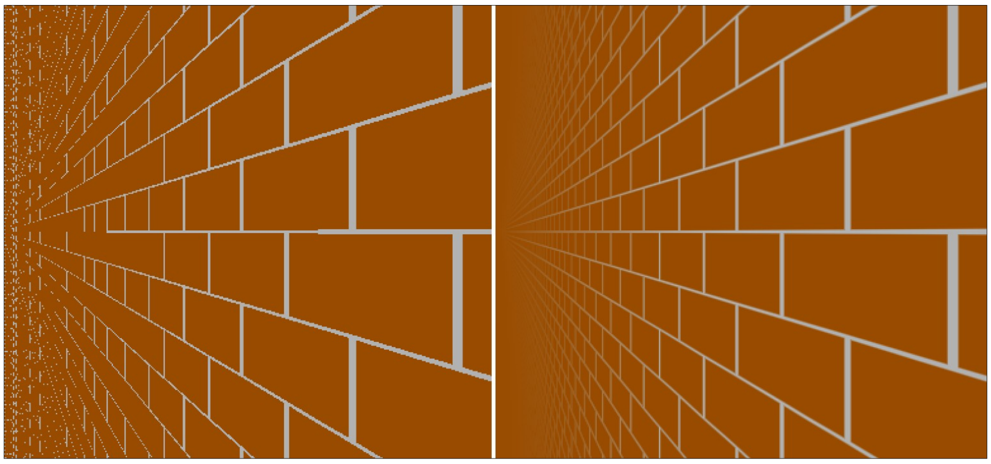
*Green and red lines: two identical **smoothstep** ramps with decreasing offset. Black line: the difference between them (green minus red).*

When the two smooth ramps move very close together, the spatial extent of their difference remains almost the same (their step width plus the 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 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: dFdx(s), dFdx(t), dFdy(s) and dFdy(t). 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 );
float adiff = abs(diff);
float pixelsize2 = max( max( absdiff.x, absdiff.y ), max( absdiff.z, absdiff.w ) );
```

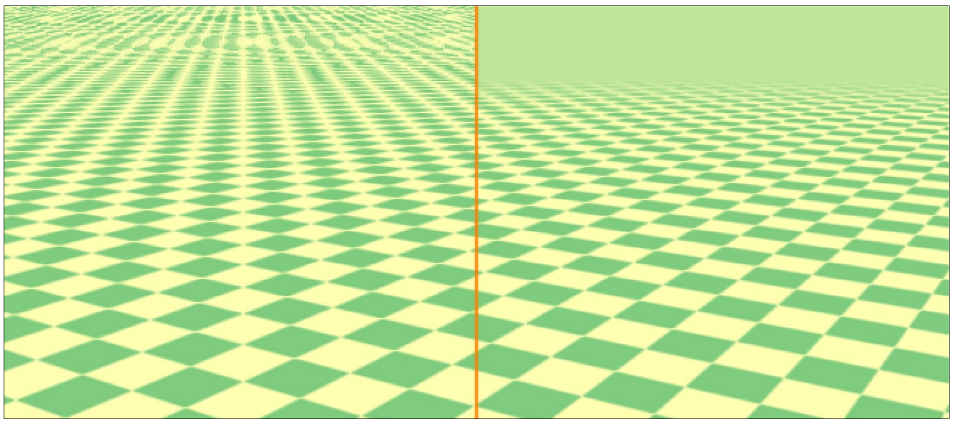
Note the nested ***max*** functions – GLSL can only select the maximum of two values at a time. OSL has a ***max*** function that takes an arbitrary number of arguments and returns the maximum in one go, but the built-in GLSL version maps to a hardware accelerated “compare-and-select” instruction and 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
        mat2 J = mat2( dFdx(p), dFdy(p) ); // From p space to device space
        float pitch = length(abs(vec4(J))); // 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 smooth transition from a checkered pattern with values 0 and 1 to a constant value 0.5 when the pattern approaches the Nyquist limit. 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 parts of the surface pattern is frequency-clamped out of existence.

```
// Anti-aliased checkerboard pattern with early-out frequency clamping
float clampedcheckers( vec2 p ) {
        mat2 J = mat2( dFdx(p), dFdy(p) ); // From p space to device space
        float pitch = length(abs(vec4(J))); // 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

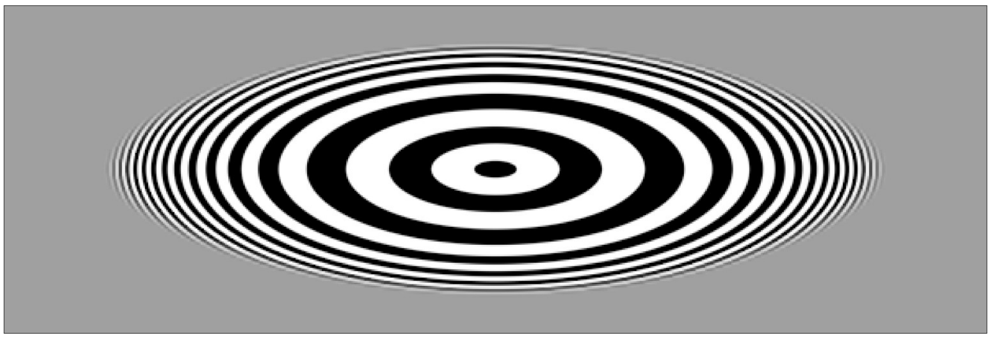
Periodic patterns are often one-dimensional, or have different periods in different directions. In such cases, we can do better than to make a rough estimate of the average size of a pixel and removing all frequencies that might possibly cause moiré. We are dealing with 3-D graphics, and we are routinely 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 gradient of a coordinate mapping 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 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:

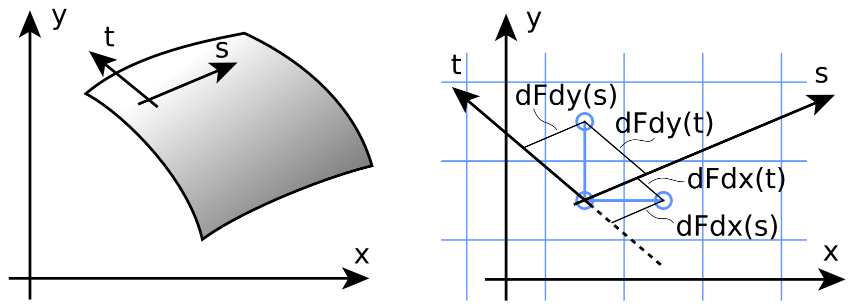
```
float clampedbullseye( vec2 p ) {
        p *= vec2(0.5, 1.2); // Scale differently in x and y
        float r = 4.0 * (length(p) – 0.5); // Radial distance
        // Linear increase near center, exponential increase further away
        float rf = (r < 0.0 ? r + 1.0 : pow( 2.81828, r ) );
        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
}
```

A rendering of this test pattern is shown 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 that the anisotropic scaling causes more stripes to render in the stretched-out *x* direction. Anisotropic anti-aliasing in image-based texturing is a difficult problem that requires multisampling, and it's a problem that 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 many classic algorithms, the seed is simply the value of the previous pseudo-random number.

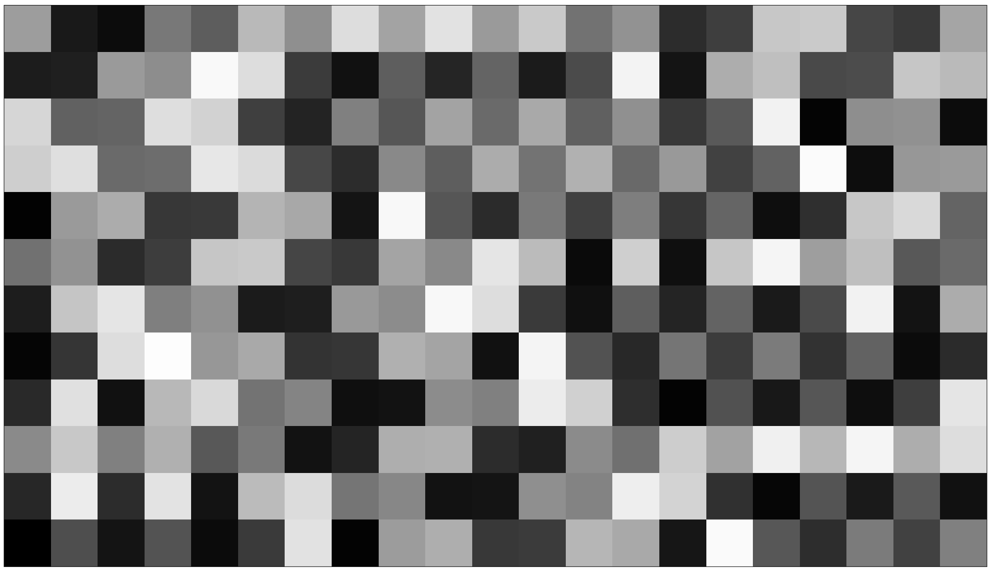
For something that approaches real randomness, a pseudo-random number generator can be given a seed that depends on some real world property that is effectively random for the purpose at hand. Computers mostly keep very accurate track of the time of day, and when a human user chooses to run a program, 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 unique hash 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.

Permutations

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 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 permutations on the fly. In fact, there are way too many to cover them all here, so we will restrict ourselves to three 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*. A surprisingly recent publication by Jarczynski and Olano [Marc Jarzynski and Marc Olano: https://jcgt.org/published/0009/03/02/] suggests this reasonably simple function for hashes in computer graphics, and 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, but this function is “random” enough to allow mashing the x and y components together into one and computing the hash in one go:

```
// 2-D hash by two nested calls to pcg_hash
float cellnoise_pcg(vec2 p) {
    vec2 pf = floor(p);
    return float(pcg_hash(uint(pf.x)+ pcg_hash(uint(pf.y))) % 256u) / 256.0;
}

// 2-D hash by a single call to pcg_hash with an ad-hoc combo of x and y
float cellnoise_pcg2(vec2 p) {
    vec2 pf = floor(p);
    return float(pcg_hash(uint(pf.x + pf.y*289.0)) % 256u) / 256.0;
}
```

Floating point hash

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 increasingly becoming more 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, but in those cases the programmer can usually ask for proper 32-bit precision, at the cost of a reduction in performance.) 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 using 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: $hash(x, y) = hash(hash(x) + y)$.

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 $\{1, 2, 3, \ldots, N\}$ 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 $p(x) = (2Ax^2 + Bx) \, mod \, A^2$, 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. Functions that work reasonably well are:

```
// 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 floating-point hash by nested calls to the 1-D hash
float permute289( vec2 p ) {
        return permute289( permute289( h.x ) + h.y );
}

// Cellnoise of adequate quality using only floating point math
float cellnoise_perm( vec2 p ) {
        return permute289( p ) / 289.0; // Scale output range to [0,1]
}
```

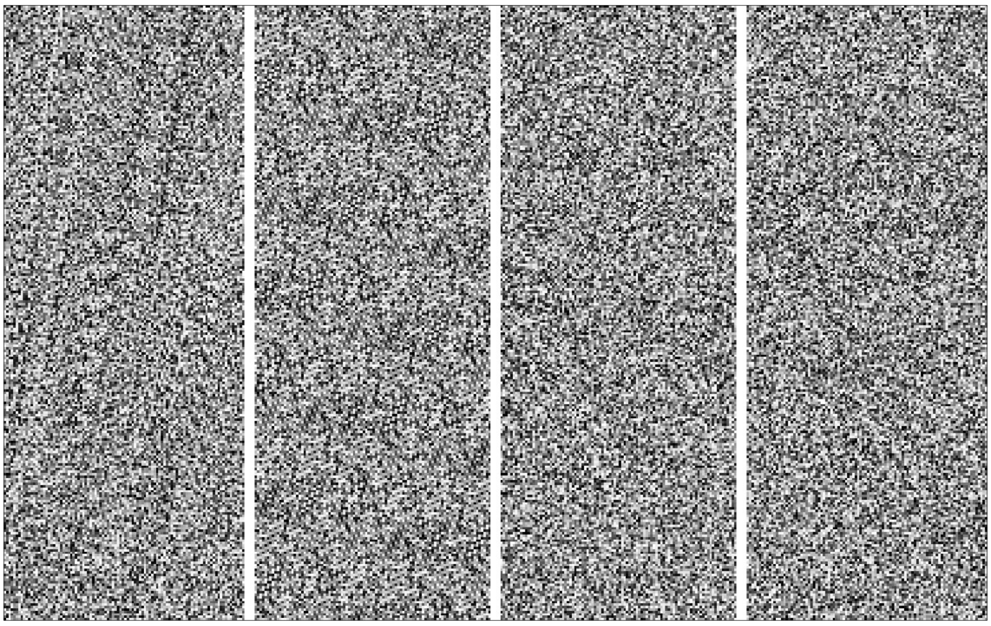
Sin hack

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

```
// Horrible hash that works mostly by accident and shouldn't really be used
float hash_hack( 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 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 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 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 algorithm is undocumented, it can change without notice, and the result is strongly platform-dependent. The distribution of output values can be 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 even good.



cellnoise(x,y) *implemented in GLSL with code from this chapter. Left to right: "sin hack", permutation polynomials, two calls to PCG hash, one call to PCG hash. The sin hack stands out in a bad way, and it's really unfit for pattern generation. Permutation polynomials are adequate. Both uses of PCG hash are excellent.*

8 Fractals

Theory (including some intuitive explanation of the property “fractal dimension”) and a visual gallery of natural objects. Present Mandelbrot as one fun and simple example with very limited practical use. No control over appearance, huge problems with anti-aliasing, iterative function that can require a lot of computations, execution time varies wildly depending on where you evaluate it. Still, a fascinating class of functions that are reasonably shader-friendly.

As a 2-D example of an iterative fractal, show midpoint displacement, but mention that it’s not a shader-friendly algorithm – it needs to be precomputed, and it’s a better fit for geometry generation than for procedural texturing.

Faking fractals by additive frequency synthesis. Again, requires pre-computation of large texture images. Use narrowband base functions instead, and sum up terms with increasing frequency and decreasing amplitude until you reach the desired level of detail. Termination criteria can be reaching sub-pixel feature size, negligible impact on the final result, and time constraints.

Concrete example: “fractal sum” of sine waves with random phase and orientation. Limiting the span of frequencies to create a narrow-band function leads up nicely to the next chapter.

9 Noise

Takeaway from the previous chapter: spectral synthesis with sine waves looks nice, but requires a lot of terms to look “random” enough. Seek a better base function which has the properties of many sine waves of similar frequency and random orientations already summed up. (List the desirable traits, with credit to Perlin.)

Perlin noise in its original form (cite Perlin), with the algorithm explained in detail. (This will require a substantial amount of exposition and illustrations. Borrow from “Simplex Noise Demystified” where appropriate, but rethink the presentation for this context.)

10 Noises

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

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

Present code for snoise in 2-D, 3-D and 4-D, and use them in a few simple examples

Additive iterative synthesis with “fractal” sums of different scales, with examples

Using previous terms (large scale) to influence later terms (small scale), creating “multifractals”.

Flow noise as a variation of this. Present the utility and simplicity of analytic derivatives of simplex noise.

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

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

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).

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

Show 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 tesselation at render time is considerably more complicated in a hardware-constrained context (given the inherent assumptions of the current hardware architecture).

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)