# 2. In the beginning: the pixel stream editor

## Procedural texture

Combining controlled noise into various mathematical expressions produces *procedural texture* [EBERT98],[FOLEY96],[PERLIN85].

Unlike traditional texture mapping, procedural texture doesn't require a source texture image. As a result, the bandwidth requirements for transmitting or storing procedural textures are essentially zero.

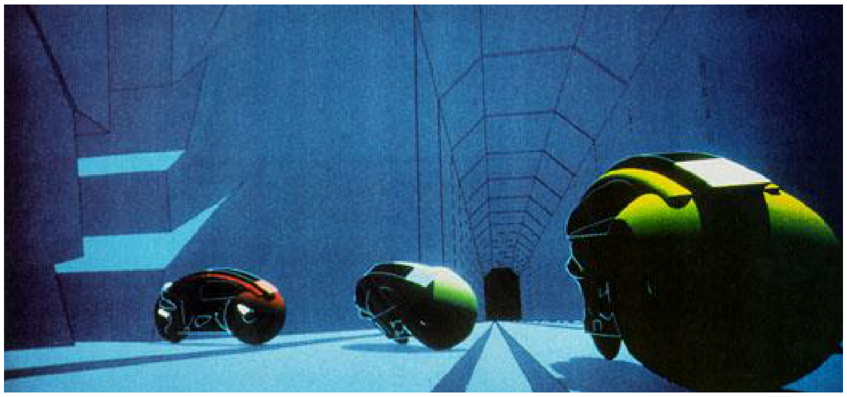
Also, procedural texture can be applied directly onto a three dimensional object. This avoids the "mapping problem" of traditional texture mapping. Instead of trying to figure out how to wrap a two dimensional texture image around a complex object, you can just dip the object into a soup of procedural texture material, defined as a function over a volumetric domain. Essentially, the virtual object is carved out of a virtual solid material defined by the procedural texture. For this reason, procedural texture is sometimes called *solid texture*.

## TRON

I first started to think seriously about procedural textures when I was working on TRON at MAGI in Elmsford, NY, in 1981. TRON was the first movie with a large amount of solid shaded computer graphics. This made it revolutionary. On the other hand, the look designed for it by its creator Steven Lisberger was based around the known limitations of the technology.

Lisberger had gotten the idea for TRON after seeing the MAGI demo reel in 1978. He then approached the Walt Disney Company with his concept. Disney's Feature Film division was then under the visionary guidance of Tom Wilhite, who arranged for contributions from the various computer graphics companies of the day, including III, MAGI, and Digital Effects.

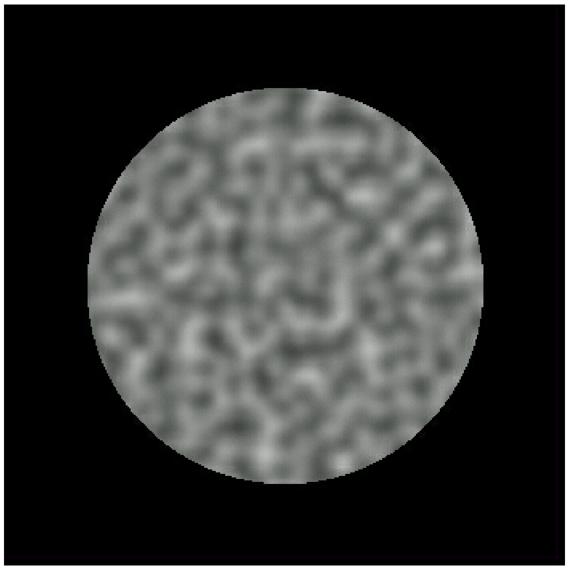
Working on TRON was a blast, but on some level I was frustrated by the fact that everything looked machine-like (a typical scene is shown below). In fact, that machine-like aesthetic became the "look" associated with MAGI in the wake of TRON. So I got to work trying to help us break out of the "machine-look ghetto."



One of the factors that influenced me was the fact that MAGI's *SynthaVision* system did not use polygons. Rather, everything was built from boolean combinations of mathematical primitives, such as ellipsoids, cylinders, truncated cones. As you can see in the illustration, the lightcycles are created by adding and subtracting simple solid mathematical shapes.

This encouraged me to think of texturing in terms not of surfaces, but of volumes. First I developed what are now called "projection textures," which were also independently developed by a quite a few folks. Unfortunately (or fortunately, depending on how you look at it) our Perkin-Elmer and Gould SEL computers, while extremely fast for the time, had very little RAM, so we couldn't fit detailed texture images into memory. I started to look for other approaches.

## Noise



The first thing I did in 1983 was to create a primitive space-filling signal that would give an impression of randomness. It needed to have variation that looked random, and yet it needed to be controllable, so it could be used to design various looks. I set about designing a primitive that would be "random" but with all its visual features roughly the same size (no high or low spatial frequencies).

I ended up developing a simple pseudo-random "noise" function that fills all of three dimensional space. A slice out of this stuff is pictured. In order to make it controllable, the important thing is that all the apparently random variations be the same size and roughly isotropic. Ideally, you want to be able to do arbitrary translations and rotations without changing its appearance too much. You can find a C version of my original 1983 code for the first version in Appendix A (actually my first implementation was in FORTRAN).

My goal was to be able to use this function in functional expressions to make natural looking textures. I gave it a range of -1 to +1 (like sine and cosine) so that it would have a dc component of zero. This would make it easy to use noise to perturb things, and simply "fuzz out" to zero when scaled to be small.

Noise itself doesn't do much except make a simple pseudo-random pattern. But it provides seasoning to help you make things irregular enough so that you can make them look more interesting.

The fact that noise doesn't repeat makes it useful the way a paint brush is useful when painting. You use a particular paint brush because the bristles have a particular statistical quality - because of the size and spacing and stiffness of the bristles. You don't know, or want to know, about the arrangement of each particular bristle. In effect, oil painters use a controlled random process (centuries before John Cage used the concept to make post-modern art).

Noise allowed me to do that with mathematical expressions to make textures.

## Pixel stream editing

In late 1983 I wrote a language to allow me to execute arbitrary shading and texturing programs. For each pixel of an image, the language took in surface position and normal as well as material ID, ran a shading, lighting and texturing program, and output color. As far as I've been able to determine, this was the first shader language in existence (as my grandmother would have said, who knew?).

Rob Cook at Pixar had, independently, developed an editable expression parser to parse user-defined arithmetic expressions at each surface sample. He called this technique "Shade Trees." But Shade Trees had no notion of flow-of-control (conditionals, variably iterated loops, procedures).

Pat Hanrahan has told me that he got the inspiration to make a full procedural shading language after he visited MAGI and I showed him what you could do by having access to a user-defined language at every pixel. Pat then designed and implemented the "RenderMan" shading language at Pixar (for which he received a well-deserved Technical Academy Award).

The key leap of faith I made (odd then, obvious now) is that you should just be able to go ahead and run whatever program you feel like at each surface sample, and that it should be easy to keep quickly modifying this program to refine your results. In order to make things run fast, I modified MAGI's existing SynthaVision renderer to create an intermediate file, after the visible surface and normal calculations have been done. The file just contained a stream of pixel samples, each consisting of *{ Point , Normal , SurfaceId }*. I would stream these samples into my procedural shader, which would then spit out a final RGB for each sample. The big advantage of this is that I could keep running the shader over and over, without having to redo the (in 1983) very expensive point/normal calculations.

By far the oddest thing about the environment at MAGI, in retrospect, was the fact that they ran everything in FORTRAN 66. This was a legacy issue - the SynthaVision ray tracer was originally written by Bob Goldstein, one of the founders of MAGI, sometime prior to 1968 (Bob's first paper on doing volumetric booleans by ray tracing, was published in the journal *Simulation* in 1968). Since then it had simply grown in the same language. FORTRAN 66 was very limiting - lacking recursion, insensitive to case, limited in variable name length to six characters or less, and a host of other qualities that reflected the era it came out of - the engineering culture up to the mid-sixties, which was very unlike the more countercultural aesthetic that nurtured UNIX and C at Bell Labs.

I was one of a group of young upstarts at MAGI who were into UNIX and C, but were not permitted to use it, for legacy reasons. So my solution was to build an entire language on top of FORTRAN. I implemented in FORTRAN only those core "kernel" functions that needed to be computed quickly, such as Noise, or that needed to use built-in math libraries, such as Sin and Cos. For everything else, I used my homegrown shading language. For this reason, I called it "kpl", for "Kernel Programming Language". It has been claimed that the letters "kpl" could also be interpreted in other ways, but frankly I just don't see it.

Kpl was a special purpose language - the only thing I really cared about was being able to manipulate floating point vectors verey easily. This led to a number of language design decisions which made everything easier. In the next section I'll briefly describe the language.

The important thing about reducing everything to floating point vectors was that I could treat normal perturbation, local variations in specularity, nonisotropic reflection models, shading, lighting, etc. as just different forms of procedural texture - the environment does not make any a priori assumption about these things, so it was easy to mix it up and try different models.

## The language for Pixel Stream Editing

The language was very simple, but it got the job done. The important thing was that it compiled immediately into an intermediate P-code, which executed very fast. That allowed me to do fast repeated design iterations, with visual feedback at each iteration.

Perhaps the oddest feature of the language was that every variable maintained a separate stack - so scoping for any variable was based on run-time execution, not lexical. This turned out to be *extremely* useful for procedural texturing, since it provided an easy and flexible way to create nested data environments. The basic features of the language are outlined below:

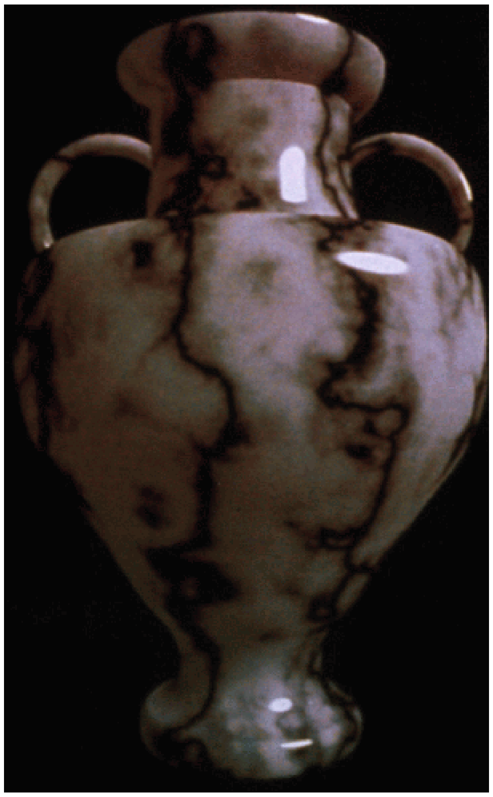
- Stack language
- Post-process to visible surface algorithm
- Intermediate "point/normal/id list" data-structure
- Evaluated at every surface sample
- Variables set at each sample:
  - Point
  - Normal
  - Id
- All values are vectors of floating point
- Transform matrices are vectors of length 12 or 16
- Values are TRUE iff at least one component is non-zero
- Every variable is a stack of values
- Flow of control:
  - IF THEN ELSE
  - LOOP with CONDITIONAL-BREAK
  - PROCEDURE with ARGS
- Scoping
  - ASSIGN (var is global to this proc)
  - PUSH-ASSIGN (var is local to this proc)
  - POP ON PROCEDURE EXIT
- Library of kernel functions, including:
  - + - * /
  - Index
  - Noise
  - Bias
  - Gain
  - Sin
  - Cos
  - Pow
  - Mul (matrix)
- Library implemented in the language, including:
  - Abs

  - Dot
  - Cross
  - ...

## Experience and interaction

My interactive process when working with this first shader was really simple. I had two interaction windows: a text editor and an rendered-image display. I'd make text modifications, hit a key that would save and run, and then look at the result. Then I'd make more text modifications, etc.

When playing with this interaction environment, I found that I could get big speedups by recomputing just a sub-window in the image where I really wanted to see an effect. I was able to get a good rhythm going of iterative shading/texturing as long as I could see the result within about 15 seconds of hitting the ENTER key (compilation took much less than a second; pretty much all of the time was taken up calculating the image). Of course, back in 1984 this didn't allow me to compute very high resolution images (around then we had only a few Mhz to play with), but any longer than 15 seconds of computation was too long to maintain a good interactive process. In any case, by looking at carefully selected subwindows, I could interactively steer the quality of the complete texture. Ultimately, it turned out to be fairly straightforward to interactively design subtle textures such as the marble vase below, which took about 20 minutes back then to render at high resolution (but would require only a matter of seconds on today's computers):

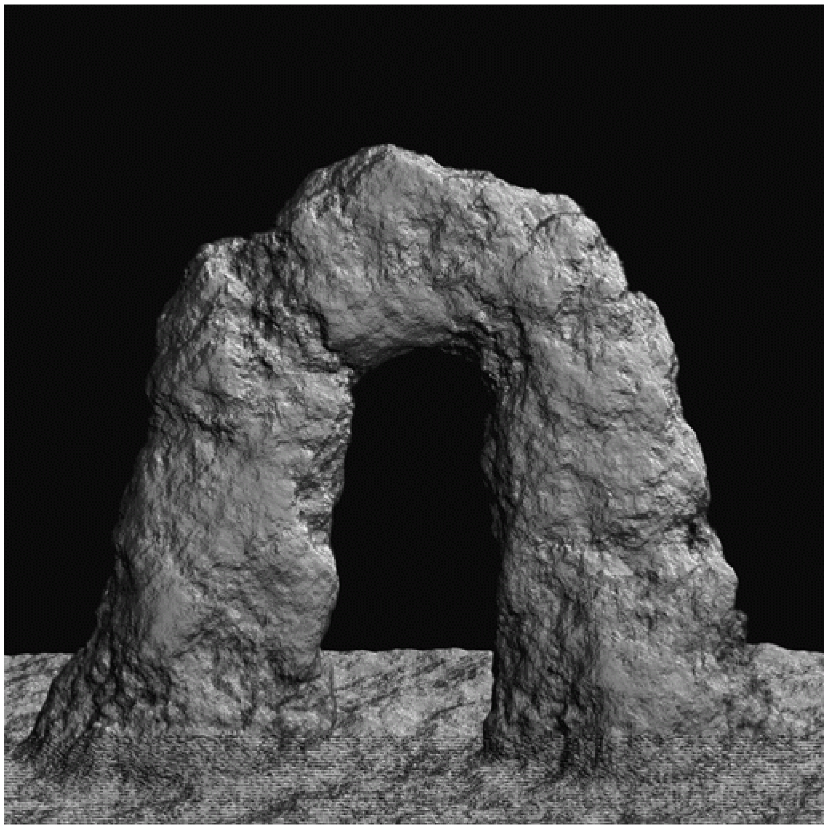


## Industry adoption

I presented this work first at a course in SIGGRAPH 84, and then as a paper in SIGGRAPH 85. Because the techniques were so simple, they quickly got adopted throughout the industry. The release of Pixar's commercial-strength RenderMan language helped a lot. By around 1988 noise-based shaders where *de rigeur* in commercial software.

I didn't patent. As my grandmother would have said...

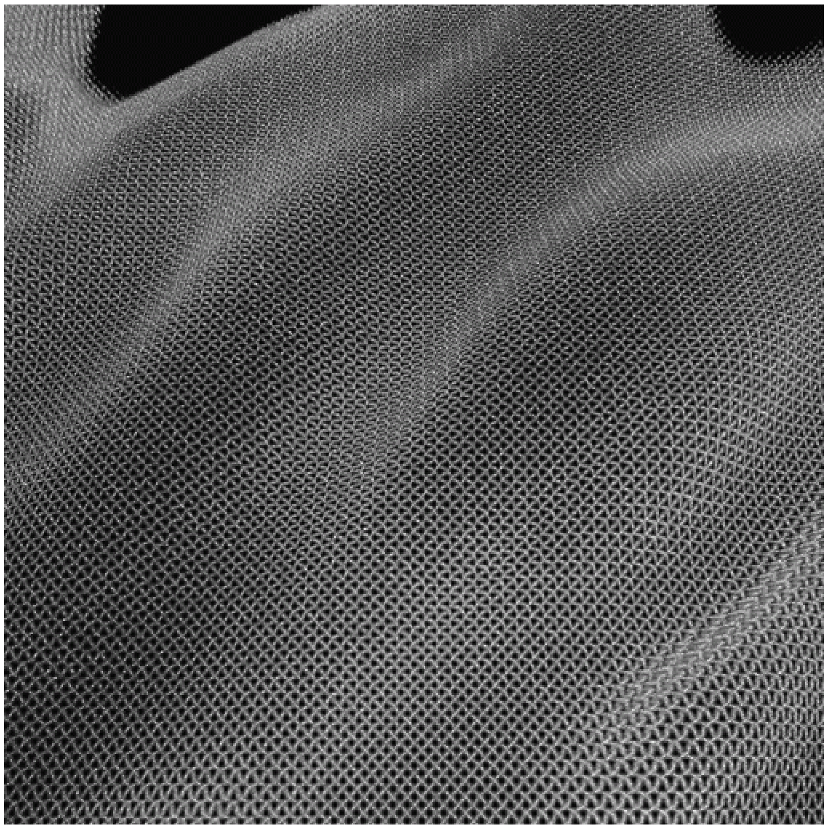
## Hypertexture



Meanwhile, I joined the faculty at NYU and did all sorts of research. One of the questions I was asking in 1988 and 1989 was whether you could use procedural textures to unify rendering and shape modeling. I started to define volume-filling procedural textures and render them by marching rays through the volumes, accumulating density along the way and using the density gradient to do lighting.

I worked with a student of mine, Eric Hoffert, to produce a SIGGRAPH paper in 1989 [PERLIN89]. The technique is called hypertexture, officially because it is texture in a higher dimension, but actually because the word sounds like "hypertext" and for some reason I thought this was funny at the time. I offer no redeeming excuse.

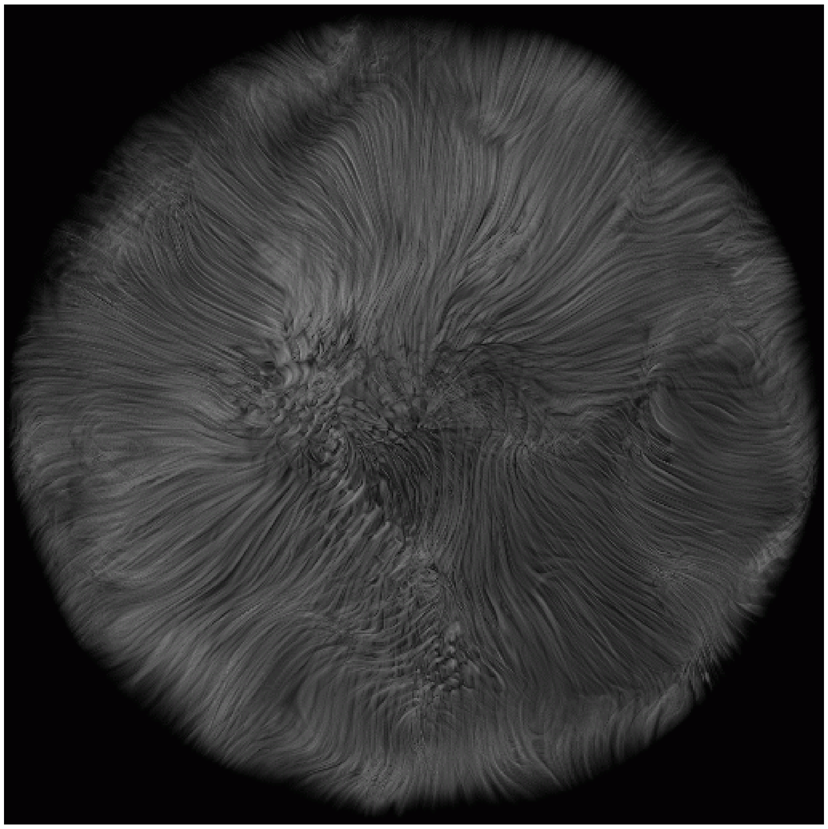
The image above is of a procedurally generated rock archway. Like all hypertextures, it's really a density cloud that's been "sharpened" to look like a solid object. I defined this hypertexture first by defining a space-filling function that has a smooth isosurface contour in the shape of an archway. Then I added to this function a fractal sum of noise functions: at each iteration of the sum I doubled the frequency and halved the amplitude. Finally, I applied a high gain to the density function, so that the transition from zero to one would be rapid (about two ray samples thick). When you march rays through this function, you get the image shown.



I tried to make as many different materials as possible. Above is one of a series of experiments in simulating woven fabric. To make this, I first defined a flat slab, in which density is one when y=0, and then drops off to zero when y wanders off its zero plane. More formally: f(x,y,z) = {if |y| > 1 then 0 else 1 - |y|}.

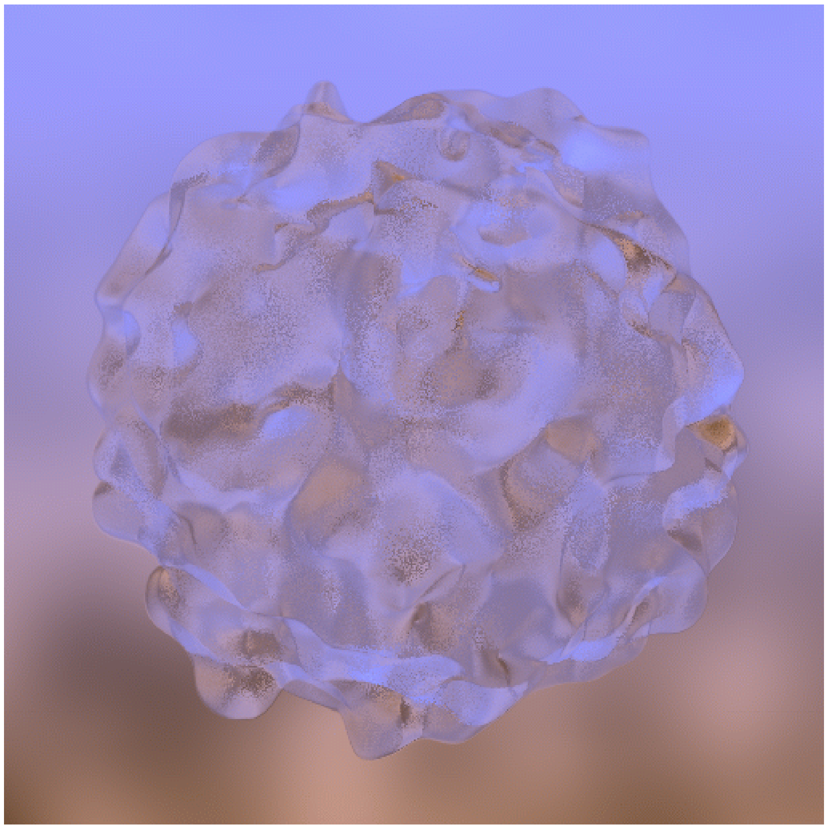
I made the plane ripple up and down by replacing y with y + sin(x)*sin(z) before evaluating the slab function. Then I cut the slab into fibers by multiplying the slab function by cos(x). This gave me the warp threads of a woven material. Finally, I rotated the whole thing by $90^o$ to get the weft threads. When you add the warp and the weft together, you get something like the material on the left.

To make the fiber more coarse (ie: wooley) or conversely more fine, I modulated the bias and gain of the resulting function. To make the surface undulate, I added low frequency noise to y before evaluating anything. To give a nice irregular quality to the cloth, I added high frequency noise into the function.



I also made a Tribble, as shown here, as well as other experiments in "furrier synthesis". Here I shaped the density cloud into long fibers, by defining a high frequency spot function (via noise) onto an inner surface, and then, from any point P in the volume, projecting down onto this surface, and using the density on the surface to define the density at P. This tends to make long fibrous shapes, since it results in equal densities all along the line above any given point on the inner surface.

I made the hairs curl by adding low frequency noise into the domain of the density function. This was the first example in computer graphics of long and curly fur. Around the same time Jim Kajiya made some really cool fur models, although his techniques produced only short and straight fur. Jim had the good sense to use earthly plushy toys for his shape models, instead of alien ones. The earthly ones are more easily recognized by the academy...



It has always seemed to me that there would be advantages in having optical materials with continually varying density, within which light travels in curved paths. The image on the left is a hypertexture experiment in continuous refraction. The object is transparent, and every point on its interior has a different index of refraction. I implemented a volumetric version of Snell's law, to trace the curved paths made by light as it traveled through the object's interior.

The background is not really an out-of-focus scene; it's just low frequency noise added to a color grad. This is a situation in which noise is really convenient - to give that look of "there's something in background, and I don't know what it is, but it looks reasonable and it sure is out of focus."

## Meanwhile, back at the Ranch...

Meanwhile, back at the Ranch (if you're reading this you presumably know *which* ranch I'm talking about) the use of noise spread like wildfire. All the James Cameron, Schwartzenegger, Star Trek, Batman, etc. movies started relying on it.

Procedural texture benefits from Moore's law: as computer CPU time becomes cheaper, production companies increasingly have turned away from physical models, and toward computer graphics. Noise-based procedural shading is one of the main techniques production companies use to fool audiences into thinking that computer graphic models have the subtle irregularities of real objects. For example, Disney put it into their *CAPS* system - you can see it in the mists and layered atmosphere in high end animated features like The Lion King. In fact, after around 1990 or so, *every* Hollywood effects film has used it, since they all make use of software shaders, and software shaders depend heavily on noise. Eventually, they even gave me a Technical Academy Award for it [SCITECH97].

One problem with all this is that as audience expectations improve, the size and computational complexity of shaders has been increasing steadily. For example, "The Perfect Storm" averaged about 200 evaluations of Noise per shading sample. Even with the current impressive performance of computers, each frame took a long time to compute.

Recently I've been working on addressing this problem. In my next chapter, I'll show work I've been doing more recently on making Noise better, faster, and more "hardware friendly".

## References:

[EBERT98] *Texturing and Modeling; A Procedural Approach,* Second Edition; Ebert D. et al, AP Professional; Cambridge 1998c;

[PERLIN89] Perlin, K., and Hoffert, E., *Hypertexture,* 1989 Computer Graphics (proceedings of ACM SIGGRAPH Conference); Vol. 23 No. 3.

[PERLIN85] Perlin, K., *An Image Synthesizer,* Computer Graphics; Vol. 19 No. 3.

[SCITECH97] Technical Aphievemegt Award from the Academy of Motion Picture Arts and Sciences, *``for the development of Perlin Noise, a technique used to produce natural appearing textures on computer generated surfaces for motion picture visual effects."*