# Procedural Content Generation

Author: Bjarki Guðlaugsson

## Index

Introduction .......... 3
The Origin of the Theory .......... 3
Nature Described With Mathematics .......... 4
The Methods of Creation .......... 5
PCG Introduced to Computer Games .......... 7
The Need .......... 8
The Limitations .......... 9
Where Are We Today? .......... 10
Hardware Designed to Aid .......... 13
Conclusion .......... 14
References .......... 15
Images .......... 15
Appendix A – Blue Marble Stone Texture by Pixar .......... 17

## Introduction

The exact data needed to reconstruct the cratered surface of the Moon in precise detail requires absolutely vast amounts of computer-memory: reasonable enough for a catalogue of lunar geography, but pointless if the purpose is to produce a convincing background for a TV science-fiction drama or a level in a computer game. How can we achieve the effect of having a cratered moon-like planet without actually storing the detailed surface? The answer is procedural content generation. Procedural content generation uses computer algorithms and the processing power of a CPU to create an object on-the fly. As opposed to having somebody, like an artist, create it beforehand and load it from a storage medium when it is needed.

In this paper we will go into what makes this technology "tick", where it began, where it is headed and what its limitations are. To stress the need for this technology, we will focus mainly on the gaming industry and why that industry's mainstream methods for creating content will eventually prohibit progress in the field.

## The Origin of the Theory

According to Professor Stewart [1], in 1958 a young mathematician named Benoît Mandelbrot joined IBM. He began working on a variety of apparently unrelated problems: word-frequencies in linguistics, error-bursts in the transmission of messages, turbulence, galaxy clusters, fluctuations in the stock market and the level of the river Nile. By the early 1960s he began to realize that all of his work was somehow interrelated. It was all about the geometric structure of irregular phenomena and the ability to procedurally reconstruct them. Mandelbrot encapsulated his ideas in a single word "fractal" in 1975. He used it in the title of a book called The Fractal Geometry of Nature, published the same year. A fractal is a basic shape, like a triangle or a circle, which can be repeated e.g. recursively to create a more complex object."

"Fractal forgeries", as Benoît called them, are artificial computer-generated representations of e.g. mountains, coastlines, lunar landscapes, and even music, that all bear an uncanny resemblance to the real thing.

These methods mimic the desired forms without having to worry about precise details. There are many mathematical "works of art" that have been created with fractal forgery and probably the two most famous are the Menger sponge (see image 1) and the Sierpinski Arrowhead (see image 2).

**Image 1: The Menger Sponge**

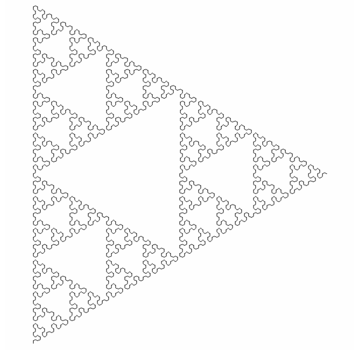
**Image 2: The Sierpinski Arrowhead**

The work of Benoît is the fundamental basis for procedural content generation as it allows us to define complex geometrical structures in the terms of a relatively simple fractals, in other words describe natural phenomena with numbers.

## Nature Described With Mathematics

Procedural content generation is possible because many things around us, both natural and man-made, have symmetrical properties that can be described mathematically. Professor Stewart explains this in [2] in the following way: "The simplest mathematical objects are numbers, and the simplest of nature's patterns are numerical. The phases of the moon make a complete cycle from new moon to full moon and back again every twenty-eight days. The year is three hundred and sixty-five days long. People have two legs, cats have four, insects have six and arachnids have eight. In nearly all flowers, the number of petals is one of the numbers that occurs in the strange sequence 3, 5, 8, 13, 21, 34, 55, 89 also known as the Fibonacci sequence."

Another mathematical method used to describe nature is geometry. Geometry allows us to easily describe for example the leaves of a fern (see image 3) in terms of a recursive occurrence of fractal properties. These properties can be seen at every scale, in everything from mountain ranges and cloud formations down to the details of cell structures in plants and animals. Fortunately as it turns out, many of the algorithms that can be used to simulate these patterns of real-life objects are relatively simple.

**Image 3: The frond of a fern**

"Fractals and computers are a marriage made in heaven. One of the most powerful techniques in programming is recursion, whereby a procedure is broken down into a sequence of repetitions of itself. A very simple example of this is a brick wall. The procedure *build a brick wall* is defined in the terms of itself, mainly laying one course of bricks, then *build a brick wall* on top of it. In practice you must also specify when the procedure stops. In our case a logical limitation to the brick wall would be to stop when it is high enough." [2] Because of the recursive property of programming languages it is relatively easy to mimic a real-life object, the only problem lies in finding the fractals that actually make up the object. To name a very simple example, consider the stem of a tree. The stem is made up of a cylinder with a repeating collection of cylinders inside. However, saying that a tree is made up of cylinders is by itself not enough we have to consider what makes a tree look like a tree. A cylinder with a bunch of cylinders inside is just that if nothing else is applied.

## The Methods of Creation

If things are arranged in this algorithmic fashion, why are no two snowflakes the same? Or two leaves, trees or mountains for that matter? The reason is the multitude of things that exert different influences over these objects during their creation and evolution. For example, two mountains may have received, over many thousands of years, different amounts of rainfall, different amounts of pressure from movement of tectonic plates, different quantities of vegetation preventing erosion, and other variations that produce different results in the end. Accurately modeling these elements can be incredibly complicated, involving huge amounts of data and calculations. Pallister and Macri [3] describe two main methods for procedural content generation i.e. "teleological modeling" and "ontogenetic modeling".

Teleological modeling was created by Alan H. Barr and is a technique for modeling phenomena using true physical rules. This is an incredibly data-intensive approach, and it's probably the correct one for scientific applications where the end result must accurately reflect reality. For example, if one were modeling how rainfall might affect a hillside over time, with the result dictating where a home could safely be built, the teleological modeling approach would be the best choice. This approach is generally overkill for real-time 3D-graphics applications such as games where realism plays second fiddle to cool and fast.

Ontogenetic modeling was created by F. Kenton Musgrave and is a technique for modeling phenomena based on properties of their appearance, similar to the theory developed by Benoît i.e. to describe a complex object as a result of its fractals. The ontogenetic modeling method focuses mainly on the end results rather than looking at how we actually got there.

To really understand the difference between the teleological and ontogenetic approaches, imagine the modeling of clouds. The teleological approach would model the properties of water, its evaporation, temperature of the environment and so on, to try and produce the desired end result from the bottom up. The ontogenetic approach would be to observe properties of the end (such as the small wispy bits change more frequently than the larger bits, the shape changes more as the wind picks up, low-lying clouds tends to be darker, and so on). Pallister and Macri [3] mention an analogous to the ontogenetic modeling approach namely the way that real-time 3D-graphics applications model lighting. It is too complex to model the true physical lighting of a scene in real-time, with every point on every object reflecting light onto every other point. So all indirect contribution (everything other than the direct light from light sources) is, instead, modeled with a single term called the "ambient light contribution".

The analogous to the ambient light contribution, used in the real-time light calculations, is the sum of different influences on an object's creation using a single random primitive called "Noise". Noise in itself is just a random number or a collection of random numbers which are by themselves not very interesting. However, if we take the noise and use it as an input to a function in order to change its output things become of a much greater value. To further illustrate on this, imagine a function that generates a recursively cylinder with a smaller cylinder inside it (the stem of a tree described earlier). If we now add some noise to this function we can change the output to actually produce something that resembles a stem of a tree (see image 4).

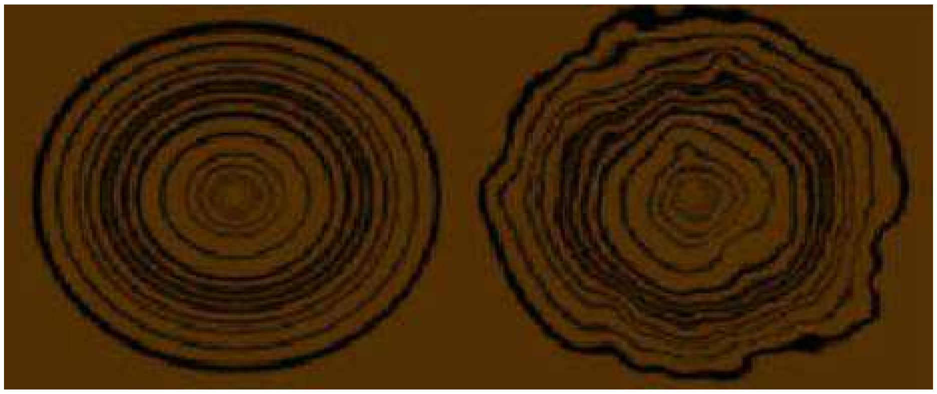

**Image 4: Tree without and with added noise to the generating function**

## PCG Introduced to Computer Games

According to [4] procedural generation for creating game contents was first introduced by a group known as the demoscene some 20 years ago. The demoscene is a computer subculture that came to prominence during the rise of the 16/32-bit micros (the Atari ST and the Amiga), but demos first appeared during the 8-bit era on computers such as the Commodore 64 and ZX Spectrum. Demos began as software crackers "signatures". When a cracked program (game) was started, the cracker or his team would take credit via an increasingly impressive-looking graphical introduction called a "cracktro". The first time this appeared was on the Apple II computers in the late 1970s and early 1980s. Later, these intros evolved into their own subculture independent of cracking software.

The demos are still a more or less an obscure form of art even in the traditionally active demoscene countries. But, nonetheless the demoscene subculture has had a tremendous affect on the gaming industry. A great deal of European game programmers, artists and musicians have come from the demoscene, often cultivating the learned techniques, practices and philosophies in their work. For example, the PC group Future Crew founded the Finnish company Remedy Entertainment, known for the Max Payne series of games, and most of its employees are former or active Finnish demosceners.

Apart from the content from the demosceners, the industry deserves some credit because the technology was actually a forced method in the early days of game development. Due to space restrictions a game had to be fitted, along with its content, into a very limited amount of disk space and only a small amount of memory was available. When deciding between the content, which usually takes a lot of unnecessary space, and an algorithm of a few bytes, that accomplishes the same idea, the choice is easy. Sadly though, with more disc space and more memory the industry somehow forgot the procedural generation.

## The Need

When the game industry and procedural content generation parted ways they did so for a good reason. They wanted more impressive environments and at the time it was cheaper to just have an artist create the content than spend time on trying to come up with a way to let the computer create it. Back then everyone was okay with low-definition artwork that could be created in relatively short amount of time. In those days the typical computer could also not have handled much more. It is safe to say that a lot has changed since then.

A typical computer game that you can buy in a store today has a large collection of very detailed environments known as levels. Levels are then typically occupied by non-player characters (NPC) and/or various structures, plants, etc. These environment do not materialize out of thin air, they have to be created by artists and shipped with the game. Because of this static content that ships with today's games we consider it normal that computer games each take up several CDs of storage. A part from the content, a game has to have a graphical engine to display the content and some kind of logical purpose. One scenario could be you character shooting his way through a Nazi infested dungeon in order to save a beautiful princess.

The idea of static content was fine and the work of the artists and the programmers in the development process was fairly balanced. The content took usually as long to make as it took to write the program (game) itself. That is until about 5 years ago. Today the work of the artists in the development process has become exponentially larger compared to that of the programmers. Games developed today typically require a warehouse full of artists and a budget larger than the budgets seen in the most expensive movie productions in Hollywood. Also the skyrocketing retail prices of games, that are a direct consequence of the prolonged development time and the high budget, again are making the games undesirable to make.

Perry and Roden [5] state that handcrafted game content currently has at least four drawbacks. First, advances in technology mean artists need, as mentioned before, increasingly more time to create content. Secondly, handcrafted content is often not easy to modify once created. In a typical game development environment, game content is created at the same time that programmers are working on the game engine (rendering, networking, code, etc.). Changes in specification or design of the game engine can dramatically alter technical requirements for content, making content already produced, obsolete. Thirdly, many widely used content creation authoring tools output content in a different format than that used by proprietary game engines. Developers typically build conversion utilities that can suffer from incompatibilities between what artists view in an authoring tool and how the content appears in the game engine. This difference can lead to repeated create-convert-test cycles, which can be costly when the conversion process is time intensive, such as creating a BSP tree from the output of a level editor. Interactive games have the potential to become much more than even the most epic film. If games are to reach this potential then humans can no longer continue to function as the predominant content creators. Also, a very good reason for the use of procedural content generation comes from game designer Will Wright [6]. He said during his introduction of his new game called Spore that during the development of The Sims game he realized that people love to make their own content. They love to customize their experience. "Owning" the content in this way means that all the stories that the gamer creates are much more meaningful. Allowing users to create their own content requires the use of some type of procedural generation in order to make the user-created content functional in the game.

## The Limitations

Imagine a single algorithm that can be used to generate a realistic-looking tree; the algorithm could be called to generate random trees, and used as a subroutine in a larger algorithm used to create a whole forest. Storing all the vertices required by the various models would be unnecessary. This would save storage space, reduce the burden on the artists; while at the same time provide a richer and more unique gaming experience for the player.

The fairy tail vision given above is a very pleasant one but also a fairly naïve one as it actually describes a way to just hand a big problem over to somebody else without actually solving it. There is no reason to doubt that this will be possible in the future but it is fairly logical to assume that it will not be done on the von Neumann architecture any time soon.

A known fact in the world of computer scientists is that a computer does not have an imagination. Asking the computer to create an interesting puzzle is like asking the computer to tell a story, make up a joke or create a riddle. Of course it's not impossible, but the time spent programming a computer to do it could be better spent doing it yourself. To further bring things down to earth the result generated by the computer would probably be a repetition of other puzzles with slight variations. Now, try to program a computer to make puzzles that also fit the theme of the game you are writing. Add that the puzzle should not be too tough, but still challenging. Finally, add in the requirement that the puzzle should be fun for the user. We can see that it would probably be easier to try and program a computer to solve an NP-complete problem in linear time than to try and take on this problem head on.

This does not mean that we should scrap the whole idea of procedurally generated content; it means that procedural methods only work as a tool to help solve a problem, not as a complete solution to it. Let's look at the puzzle generation again. The best attack strategy would probably be divide-and-conquer. Take a small part of the puzzle generation and make the computer do that for you while you take care of the part that requires imagination. For instance you could make a list of riddles (with answers), the computer could then randomly place them in a game and decide rewards for solving them. This will work because the part played by the computer does not involve any use of imagination what so ever just be sure that the computer is "aware" of all the rules that we consider as common sense. [7] This is exactly how people are utilizing these methods today.

## Where Are We Today?

We have only talked about the technology being used in the old days of the gaming industry, what its limitations are and also explored that there is a need for it today. Has the technology been in stasis since the departure with the gaming industry? What has been accomplished since then and what is the status of the technology today?

Even though the gaming industry forgot about this technology it was not forgotten by all. Researchers and hobbyists have been working hard on this over the last two decades and the results that some individuals have achieved are truly remarkable. Probably the best examples of applications today are the computer game .kkrieger and the middleware SpeedTree.

The first person shooter game .kkrieger, developed by a demoscene group known as .theprodukkt, was first introduced for the PC in 2004. The game made use of procedural synthesis for textures, levels, audio, animations and character models with unbelievable results (see image 5). What really makes the game impressive is the fact that it takes only 96-kilobytes of disk space. The CPU is used more extensively instead to generate the game's impressive content. This is quite an achievement considering that most modern games are released on several CDs, often exceeding 2 gigabytes in size which is more than 20.000 times the space that .kkrieger requires. [8]

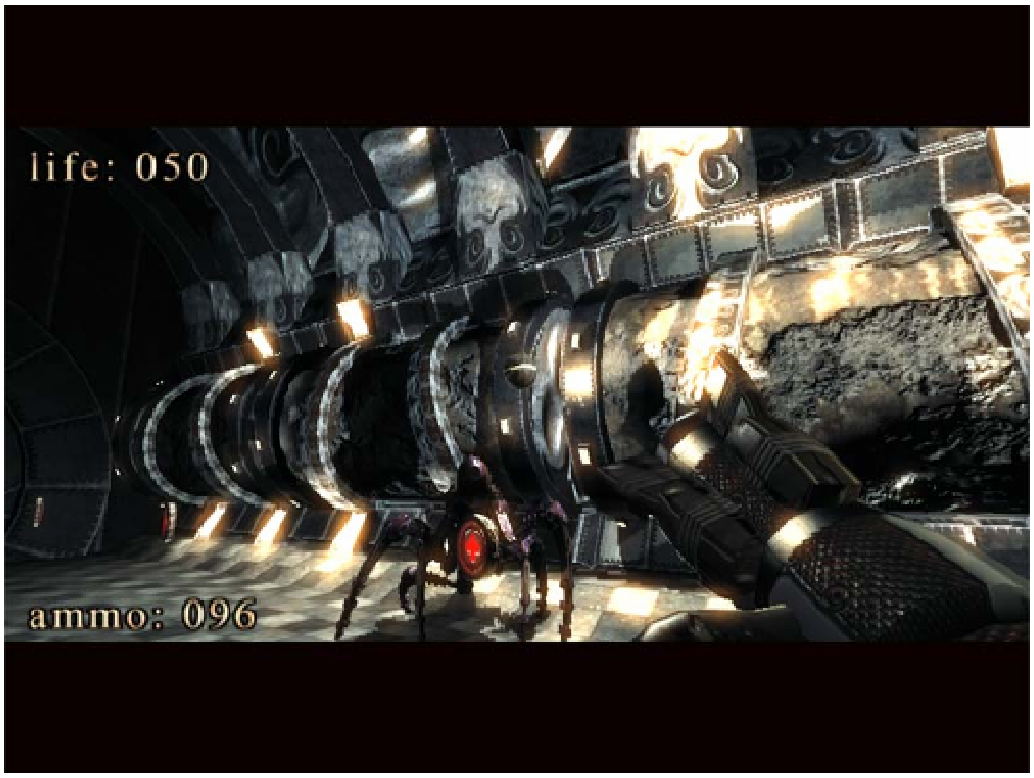

**Image 5: .kkrieger a FPS game in 96-kilobytes**

Since the release the developers of .kkrieger have almost all become employees of the gaming company Maxis and are currently working on a revolutionary game called Spore. Spore [9] is a game envisioned by the creator of the hit game The Sims, Will Wright, and will be released to the public in the fall of 2006. The game uses a variation of procedural content generation called "procedural verbs" which allow small fragments of animations, textures etc. to be procedurally combined. This allows for something that has never been seen before in the gaming industry i.e the player is now completely in control of the content of the game.

Another interesting product is SpeedTree, which is a middleware designed to generate trees and is mentioned here because it is probably the first product from a company focusing entirely on procedural content generation. Their software can be licensed and used as a part of a larger software product and is currently available for the PC, Xbox360 and PlayStation 3 platforms. SpeedTree has been used in Dark Age of Camelot and Unreal Engine 2 and will be used in the upcoming Unreal Engine 3. (see image 6)

POWERED BY
UNREAL
TECHNOLOGY
speedtree

**Image 6: SpeedTree middleware used in Unreal Engine 3**

It should be noted at this point that the gaming industry is not the only industry realizing the potential of this technology. The entertainment business seems to also have realized that this technology is going to save them a great deal of money and time. Rich and detailed environments, used for special effects in motion pictures, can all be created with less amount of work by artists and therefore less amount of money. Like it is in the gaming industry, this technology is not exactly new to the entertainment business either. In fact as early as 1989 the company Pixar patented an algorithm for creating a floor grate texture (see image 7) among others, for use in their motion pictures. [10] (Computer code for an actual texture algorithm can be seen in Appendix A)

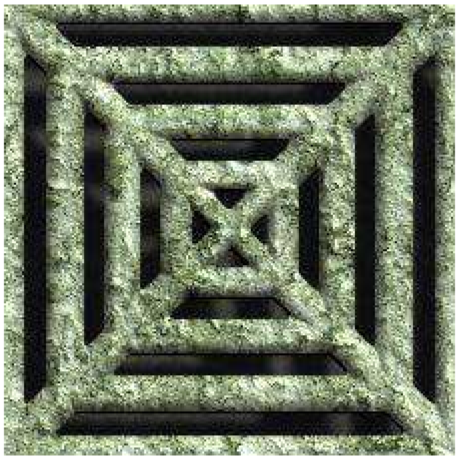

**Image 7: Generated floor grate texture patented by Pixar**

## Hardware Designed to Aid

One of the reasons this technology is not being used more widely today is that during a computer game the CPU, of a typical PC, is not exactly just sitting there and waiting for something to do.

Procedural content generation algorithms fall under the category of being CPU heavy which would not be a problem if the typical computer game did not include things like artificial intelligence and other CPU intensive game elements as well. Some game genres that could actually benefit a lot from this technology are already using the CPU to a great extent. Consider for example real-time strategy games. These games, if created by ambitious developers, have a very sophisticated artificial intelligence opponent to give the player a run for his money. Often during a typical game play scenario an average 2 GHz processor will be at 40-70% load a great deal of the time. This does not allow for much processing power to be utilized by content generation algorithms.

Fortunately hardware developers are realizing this problem and have already come up with a solution. The Xbox 360's CPU known as Xenon has a special built in property that allows it to handle procedural synthesis in a separated thread inside the CPU and pipelining it directly to the graphics processing unit of the CPU. This allows for procedural synthesis algorithms to be run very fast because the RAM has been cut out as an intermediate step. The Cell processor, developed by IBM and Sony, which will be used in the upcoming PlayStation 3, has a similar internal mechanism but instead of having 3 symmetric cores like the Xenon, it has 8 processing units called SPEs. These processing units can stream data to each other with each SPE performing a different operation on a large set of data, thus creating a highly efficient chain which performs a sequence of operations on a data set of arbitrary size. To put things into a more simple perspective this allows the tasks of an algorithm, used for production of game objects, to be divided between parts of the CPU. There is also a 35 GB/s link between the Cell and the PlayStation 3’s Graphical Processing Unit allowing objects to be rendered directly without storing them in RAM first. [8]

## Conclusion

With every new generation of games, the customer is expecting a more detailed world, more intelligent enemies, more levels etc. With more detail, more work is required from artists to produce content for the games. Soon content will be expected to be as detailed as real life objects. If the gaming industry continues to develop games with static content it is obvious that in the coming years, development costs will go through the roof and it will become impossible to create the high-definition games the customer expects at an affordable budget.

Motion pictures are pretty much the same as they were a hundred years ago apart from improved sound and picture quality. The game industry faces the same stagnant reality if static content continues to dominate.

Something has to change and the way forward for game developers, is to use the procedural content generation technology. The technology allows content to be created fast and cheap and has the included benefit of being flexible and reusable. It allows game developers and artists to focus more on the task of creating a better game and providing a gaming experience that is more interactive and unique for the player.

Fortunately as we look at the gaming products scheduled for release this fall the technology is becoming more main-stream and hopefully this is a preview of things to come.

## References

1. Stewart, Ian (1995). **Nature's Numbers (discovering order and pattern in the universe)**. The Guernesey Press Co. Ltd. Guernsey, Channel Island (2001). *ISBN: 0-75380-530-8*
2. Stewart, Ian (1997). **Does God Play Dice? – The New Mathematics of Chaos, Second Edition.** Penguin Press, (1997). *ISBN: 0-14-025602-4*
3. Macri, Dean & Pallister, Kin (2004). **Procedural 3D Content Generation**. Retrieved March 1, 2006, from http://www.devx.com/Intel/Article/20182
4. **Demoscene**. Wikipedia. Retrieved February 24, 2006, from http://en.wikipedia.org/wiki/Demoscene
5. Perry, Ian & Rode, Timothy (2003?). **From Artistry to Automation: A structured Methodology for Procedural Content Creation**. University of Denton Texas. Retrieved March 1, 2006, from http://www.eng.unt.edu/~ian/pubs/SP1-roden-timothy.pdf
6. **GDC Report: Will Wright's "The Future of Content" Lecture.** Gamasutra. Retrieved March 1, 2006, from http://www.gamasutra.com/gdc2005/features/20050315/postcard-diamante.htm
7. Noah Gibbs (2004). **Procedural Content Generation**. Retrieved February 24, 2006, from http://www.skotos.net/articles/neo6.phtml
8. **Procedural Content Generation**. Wikipedia. Retrieved February 24, 2006, from http://en.wikipedia.org/wiki/Procedural_generation
9. **Spore (game)**. Wikipedia. Retrieved March 1, 2006, from http://en.wikipedia.org/wiki/Spore_%28game%29
10. **Procedural Texture**. Wikipedia. Retrieved February 26, 2006, from http://en.wikipedia.org/wiki/Procedural_texture

### *Images*

1. *BlackMaiden Interceptor* (Cover Image). Wikipedia. Retrieved February 24, 2006, from http://en.wikipedia.org/wiki/Image:Demo_PC_BlackMaiden_Interceptor.jpg
2. *The Menger Sponge*. Google Images. Retrieved February 24, 2006, from http://astronomy.swin.edu.au/~pbourke/raytracing/angelo/light3.gif

3. *The Sierpinski Arrowhead*. Google Images. Retrieved February 24, 2006, from http://astronomy.swin.edu.au/~pbourke/fractals/sierpinski_arrowhead/
4. *Frond of a Fern*. Devx.com. Retrieved March 1, 2006, from http://www.devx.com/assets/intel/9051.jpg
5. *.kkrieger screenshot*. .theprodukkt.com. Retrieved March 1, 2006, from http://kk.kema.at/files/gfx/full1.jpg

## Appendix A – Blue Marble Stone Texture by Pixar

```
/* Copyrighted Pixar 1989 */
/* From the RenderMan Companion p.355 */
/* Listing 16.19  Blue marble surface shader*/

/*
 * blue_marble(): a marble stone texture in shades of blue
 * surface
 */

blue_marble(
         float   Ks    = .4,
                 Kd    = .6,
                 Ka    = .1,
                 roughness = .1,
                 txtscale = 1;
         color   specularcolor = 1)
{
   point PP;            /* scaled point in shader space */
   float csp;           /* color spline parameter */
   point Nf;            /* forward-facing normal */
   point V;             /* for specular() */
   float pixelsize, twice, scale, weight, turbulence;

   /* Obtain a forward-facing normal for lighting calculations. */
   Nf = faceforward( normalize(N), I);
   V = normalize(-I);

   /*
    * Compute "turbulence" a la [PERLIN85]. Turbulence is a sum of
    * "noise" components with a "fractal" 1/f power spectrum. It gives
the
    * visual impression of turbulent fluid flow (for example, as in the
    * formation of blue_marble from molten color splines!). Use the
    * surface element area in texture space to control the number of
    * noise components so that the frequency content is appropriate
    * to the scale. This prevents aliasing of the texture.
    */
   PP = transform("shader", P) * txtscale;
   pixelsize = sqrt(area(PP));
   twice = 2 * pixelsize;
   turbulence = 0;
   for (scale = 1; scale > twice; scale /= 2)
       turbulence += scale * noise(PP/scale);

   /* Gradual fade out of highest-frequency component near limit */
   if (scale > pixelsize) {
       weight = (scale / pixelsize) - 1;
       weight = clamp(weight, 0, 1);
       turbulence += weight * scale * noise(PP/scale);
   }

   /*
    * Magnify the upper part of the turbulence range 0.75:1
    * to fill the range 0:1 and use it as the parameter of
    * a color spline through various shades of blue.
    */
   csp = clamp(4 * turbulence - 3, 0, 1);
   Ci = color spline(csp,
   color (0.25, 0.25, 0.35),      /* pale blue         */
       color (0.25, 0.25, 0.35),  /* pale blue         */
       color (0.20, 0.20, 0.30),  /* medium blue       */
       color (0.20, 0.20, 0.30),  /* medium blue       */
       color (0.20, 0.20, 0.30),  /* medium blue       */
       color (0.25, 0.25, 0.35),  /* pale blue         */
       color (0.25, 0.25, 0.35),  /* pale blue         */
       color (0.15, 0.15, 0.26),  /* medium dark blue */
       color (0.15, 0.15, 0.26),  /* medium dark blue */
       color (0.10, 0.10, 0.20),  /* dark blue         */
       color (0.10, 0.10, 0.20),  /* dark blue         */
       color (0.25, 0.25, 0.35),  /* pale blue         */
       color (0.10, 0.10, 0.20)   /* dark blue         */
       );

   /* Multiply this color by the diffusely reflected light. */
   Ci *= Ka*ambient() + Kd*diffuse(Nf);

   /* Adjust for opacity. */
   Oi = Os;
   Ci = Ci * Oi;

   /* Add in specular highlights. */
   Ci += specularcolor * Ks * specular(Nf,V,roughness);
}
```