# Procedural Texture Mapping on FPGAs

Andy G. Ye and David M. Lewis [*]

Department of Electrical and Computer Engineering
University of Toronto
{yeandy, lewis}@eecg.utoronto.ca

## Abstract

Procedural textures can be effectively used to enhance the visual realism of computer rendered images. Procedural textures can provide higher realism for 3-D objects than traditional hardware texture mapping methods which use memory to store 2-D texture images. This paper proposes a new method of hardware texture mapping in which texture images are synthesized using FPGAs. This method is very efficient for texture mapping procedural textures of more than two input variables. By synthesizing these textures on the fly, the large amount of memory required to store their multidimensional texture images is eliminated, making texture mapping of 3-D textures and parameterized textures feasible in hardware. This paper shows that using FPGAs, procedural textures can be synthesized at high speed, with a small hardware cost. Data on the performance and the hardware cost of synthesizing procedural textures in FPGAs are presented. This paper also presents, the FPGA implementations of two Perlin noise based 3-D procedural textures.

## 1 Introduction

In many computer graphic applications, polygon meshes are used to model geometrical surfaces. Texture mapping increases the level of surface detail of polygon meshes by mapping two-dimensional texture images on to the meshes. In common graphic cards, the 2-D texture images are pre-computed and stored in memory on the cards. Procedural texture mapping extends the concept of texture mapping by determining the surface coloring of polygon meshes using computer algorithms. These procedural texture algorithms typically model the structures of materials like concrete, wood and marble. They can be defined in 3-D space and be parameterized using input variables defining additional attributes other than the texture coordinates.

Procedural texture mapping has become an important method of generating visually realistic images in many graphic applications. The computation, however, is often time-consuming. Procedural texture algorithms, when executed in software, often cannot achieve the real time performance demanded by many computer animation applications. While 2-D textures can be stored in RAM, 3-D textures require excessive memory. There are no efficient methods of performing texture mapping using three-dimensional or parameterized procedural textures using fixed hardware. The primary reason for this is the variety of procedural textures, which makes it difficult to design a single, efficient hardwired implementation for synthesizing all textures. Other reasons include the complexity of many procedural texture algorithms, and the ongoing development of new algorithms. A hardwired accelerator not only would be difficult to design to support all the exiting procedural texture algorithms, but also difficult to modify to support new algorithms in the future.

This paper describes a new approach to synthesizing procedural textures in hardware in which FPGA hardware is used to provide high performance implementations of procedural texture algorithms. The primary technique used is to compile the procedural algorithms into hardware structures that can be programmed into FPGAs. This approach is more memory efficient than storing pre-generated textures in memory, since only the algorithms are stored. The use of FPGAs also results in the ability to exploit the parallelism presented in each individual algorithm.

A procedural texture generator was designed using FPGAs. It is flexible enough to synthesize a variety of procedural textures in high speed, and is small enough to be implemented on one modern FPGA chip. The procedural texture generator was implemented using the Transmogrifier-2 (TM-2) rapid prototype system [11], as a part of a 3-D computer graphic rendering system design. The performance and hardware cost of synthesizing procedural textures in FPGAs are estimated using the data collected on the TM-2 system.

## 2 3-D Rendering System

A 3-D computer graphic rendering system was designed to evaluate the implementation issues of synthesizing procedural textures in FPGA hardware. The architecture of this rendering system is briefly described here. The input to the rendering system is a list of triangles. Each vertex of these triangles is specified by two triplets. The first triple, $(x, y, z)$, specifies the position of the vertex in a 3-D world space. The second triple, $(u, v, w)$, specifies the position of the vertex in a 3-D texture space. The rendering system performs four major operations on each triangle. First, the system transforms the 3-D world coordinates of the vertices into the 2-D screen coordinates. Second, all pixels inside the triangle are determined using the 2-D screen coordinates of the vertices. The texture coordinates of these pixels are then calculated. Third, the system uses the texture coordinates to calculate the color of each pixel. Finally the image is stored in a frame buffer and displayed on a screen.

Figure 1 shows the overall architecture of the rendering system. It consists of four major components:

[*]This research was supported by Micronet, Altera Corporation, Cypress Semiconductor, I-Cube, NSERC, and ATI Technologies.

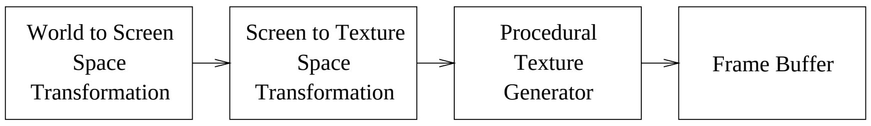

World to Screen Space Transformation
Screen to Texture Space Transformation
Procedural Texture Generator
Frame Buffer

Figure 1: 3-D Rendering System Using Procedural Textures

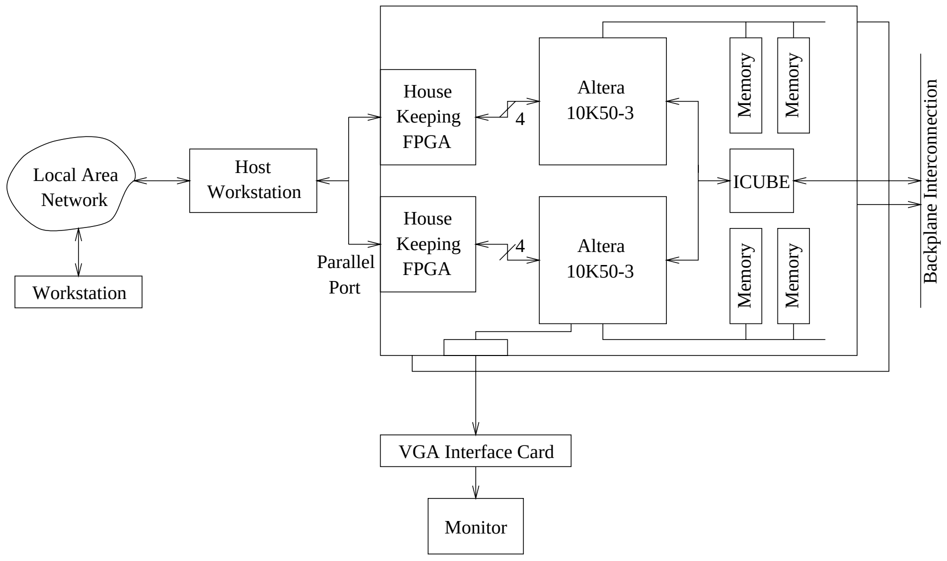


Figure 2: Experimental Setup

1. a world to screen space transformation (WSST) unit
2. a screen to texture space transformation (STST) unit
3. a procedural texture generator
4. a frame buffer

Each component performs one of the operations listed in the previous paragraph. Conventionally, WSST functions are usually implemented in software; STST and the frame buffer are implemented in hardware; and textures are implemented using a RAM. We propose to implement textures in FPGAs as a procedural texture generator. A set of textures can be implemented by loading their algorithms into the FPGA based procedural texture generator. Although STST and the frame buffer should ideally be implemented in ASIC, we also constructed them in FPGAs on our prototype.

The 3-D rendering system is implemented on the TM-2. As shown in Figure 2, the TM-2 consists of two boards. Each board contains two Altera 10K50 FPGAs and four banks of 64-bit wide SRAM. The TM-2 system can be connected to a local area network through a host workstation. Using the host, any workstation on the network can communicate with the TM-2.

The resources used in the implementation include one workstation, all four FPGAs on the TM-2, one bank of TM-2 SRAM, a VGA card, and a monitor. The workstation is connected to the TM-2 via the local area network. The partitioning of the rendering system among all hardware resources is shown in detail in Figure 3. Since there are only four FPGAs available, the entire rendering system cannot be implemented on the TM-2 system. The WSST calculations are performed once per triangle, while other units perform calculations once per pixel. Therefore, the WSST unit is implemented on the workstation, as commonly done in many graphic cards. Two FPGAs are allocated to the STST unit. One and half FPGAs are allocated for the procedural texture generator. The frame buffer is implemented using the remaining resources. It uses one bank of TM-2 SRAM as a double frame buffer. It also controls the VGA card and the monitor.

All software is written in the C programming language. All hardware designs are done in the Altera Hardware Description Language (AHDL). The rendering system uses a screen space resolution of $512 \times 512$. The texture space resolution is $512 \times 512 \times 512$. Colors are eight bits.

## 3 FPGA Implementations of Procedural Texture Algorithms

Six procedural texture algorithms have been implemented in FPGAs. Each of these algorithms takes three inputs, $u$, $v$, $w$. These three inputs specify a set of coordinates in a 3-D texture space. The substances that these textures model can be classified into two categories, solid and gaseous. Three textures model the coloring of solids including marble, brick, and wood. Another three model the coloring of gaseous substances including fog, fire, and cloud. Despite the difference in appearances, all six textures are fractal in nature — they all use the Perlin noise function to create fractal effects. In software, these algorithms are implemented in IEEE floating point arithmetic. Floating point hardware, however, is expensive to implement in FPGAs. Fixed point hardware is used, instead, for minimum precision implementations. Extensive pipelining is used to maximize the throughput of the algorithms.

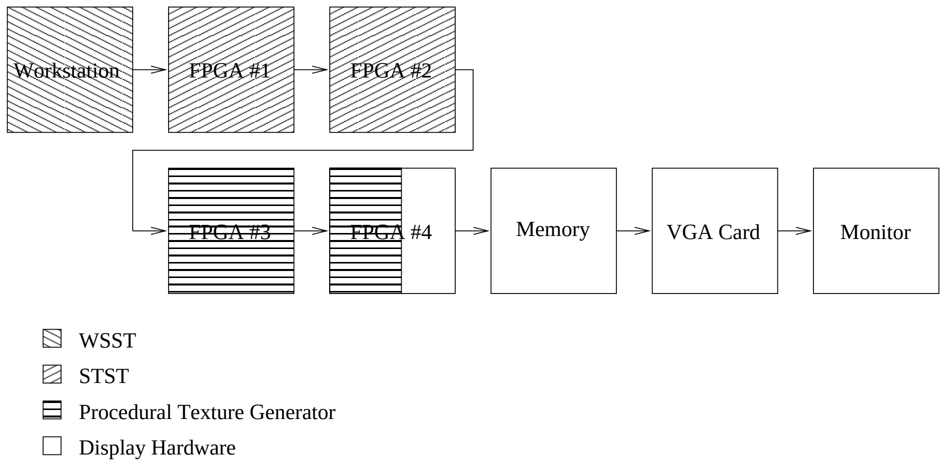
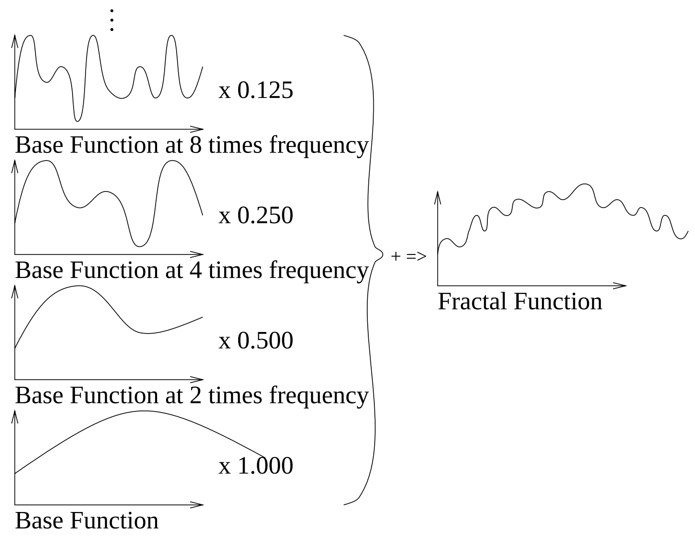


Figure 3: Partition of the Hardware Resources



x 0.125
Base Function at 8 times frequency
x 0.250
Base Function at 4 times frequency
x 0.500
Base Function at 2 times frequency
x 1.000
Base Function
+ =>
Fractal Function

Figure 4: One Dimensional Fractal Function

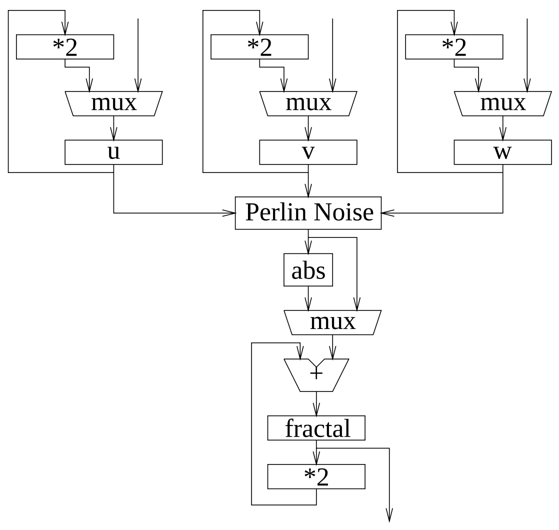


Figure 5: Fractal Function Hardware

### 3.1 Fractals and the Perlin Noise Function

This section describes the FPGA implementations of fractals and the Perlin noise function.

#### 3.1.1 Fractals

In computer graphics, fractal functions are often implemented by summing several versions of a base function at different scales and frequencies. Figure 4 shows this process in one dimension. There are a series of functions at the left side of the figure. They are derived from the same base function by varying the frequency and the amplitude. More formally, if the base function is represented by the equation $y = P(u)$, then the base function at $m$ times the frequency can be represented by the equation $y = P(m \times u)$. To create the fractal function, each version of the base function is scaled inversely proportional to its frequency; then all versions are summed together. Therefore, the fractal function becomes:

$$y = P(u) + \frac{1}{2} \times P(2 \times u) + \ldots + \frac{1}{m} \times P(m \times u)$$

For every new version of the base function created, the frequency is usually doubled and the scale factor is usually halved from the previous version. $m$ is usually set to be between eight and sixty-four. A 3-D fractal function uses a base function of three variables, $P(u, v, w)$. All input variables of the 3-D base function are scaled.

Figure 5 shows the architecture of the fractal function in detail. In the figure, blocks $u$, $v$, $w$, and $fractal$ are all registers. The multiplexers and the registers are controlled by a control unit not shown in the figure. The hardware is used to implement two fractal functions, turbulence and fractalsum. These functions are defined by the following formula:

$$turbulence = \sum_{i=0}^{3} 2^{-i} P(2^i u, 2^i v, 2^i w)$$
$$fractalsum = \sum_{i=0}^{3} \left| 2^{-i} P(2^i u, 2^i v, 2^i w) \right|$$

where the function $P(u, v, w)$ represents the Perlin noise function, the actual 3-D base function used. The hardware implements the above two equations by scaling and accumulating either the value of the Perlin noise function or the absolute value of the Perlin noise function into the register labeled fractal. When the absolute value

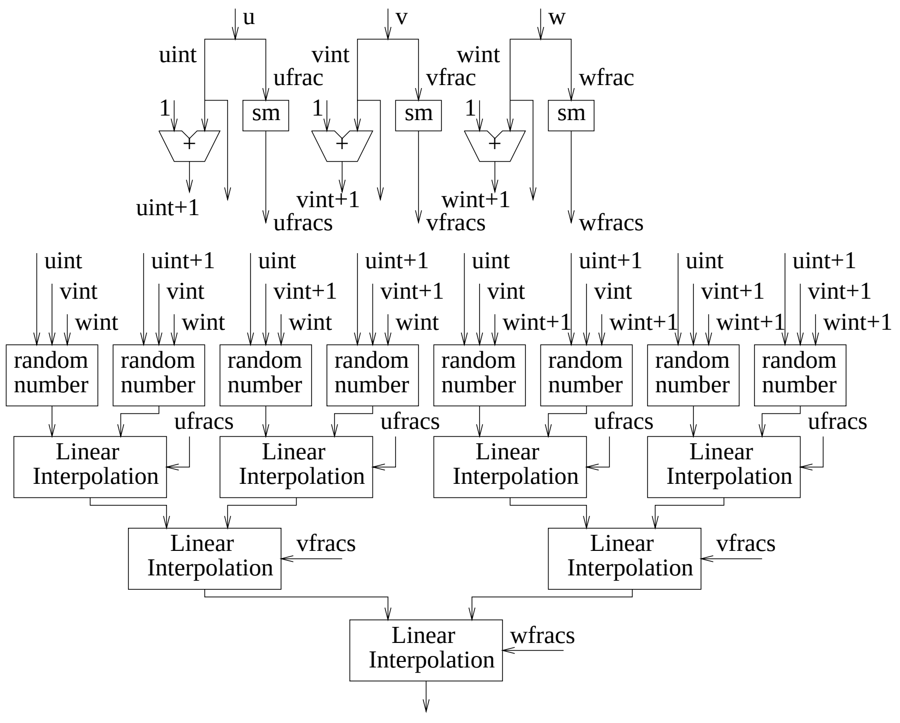


Figure 6: Perlin Noise Function Hardware

is used, the resulting fractal value is the turbulence. As the name implies, the turbulence function simulates the turbulence characteristics found in many fluids and solidified solids [6]. When the value of the Perlin noise function is directly used, the resulting function is the fractalsum function, which is often used to simulate gas formations [6]. In both cases, four cycles are needed to create one fractal value.

### 3.1.2 Perlin Noise Function

The Perlin noise function is one of the most computationally efficient base functions. In our applications, we use a Perlin noise function of three-dimensional space. It can be implemented using the following equation:

$$\begin{gathered} P(u,v,w) = \\ I(R(\lfloor u\rfloor,\lfloor v\rfloor,\lfloor w\rfloor), R(\lfloor u\rfloor,\lfloor v\rfloor,\lceil w\rceil), \\ R(\lfloor u\rfloor,\lceil v\rceil,\lfloor w\rfloor), R(\lfloor u\rfloor,\lceil v\rceil,\lceil w\rceil), \\ R(\lceil u\rceil,\lfloor v\rfloor,\lfloor w\rfloor), R(\lceil u\rceil,\lfloor v\rfloor,\lceil w\rceil), \\ R(\lceil u\rceil,\lceil v\rceil,\lfloor w\rfloor), R(\lceil u\rceil,\lceil v\rceil,\lceil w\rceil), \\ (u-\lfloor u\rfloor),(v-\lfloor v\rfloor),(w-\lfloor w\rfloor)) \end{gathered}$$

where $R(x_1, x_2, x_3)$ is a pseudo random function of its inputs; and $I(x_{000}, x_{001}, \ldots, x_{111}, x_u, x_v, x_w)$ is an interpolation function in three dimensions. This calculates the function value on the 8 corners of a grid cell, and performs interpolation based on the associated values of the eight and the distance between the point in question and each of these grid points [13].

The original Perlin noise function, as actually proposed by Ken Perlin, implements the function, $R(x_1, x_2, x_3)$, as three tables of 256 pre-generated pseudo random numbers stored in memory and two adders [6]. This method can consume quite large amounts of memory, since multiple copies of $R(x_1, x_2, x_3)$ are needed to fully exploit the parallelism available. A more efficient hardware method of generating pseudo random function values using $xor$ tables [15] is used in this study. This method provides significant saving in hardware.

The second improvement that we made to the original Perlin noise function for hardware implementation is to the interpolation method, $I(x_{000}, x_{001}, \ldots, x_{111}, x_u, x_v, x_w)$. The original function uses an computationally expensive wavelet interpolation method [6]. This method has some superior statistical properties than the ordinary 3-D linear interpolation method; however, it is much more computationally expensive. In this study, we use a smoothing function, $sm(x) = 3x^2 - 2x^3$, to remove any second order discontinuities that might result from the linear interpolation process. The interpolation function $I(x_{000}, x_{001}, \ldots, x111, x_u, x_v, x_w)$ becomes $L(x_{000}, x_{001}, \ldots, x_{111}, sm(x_u), sm(x_v), sm(x_w))$, where $L(\ldots)$ is the linear interpolation function. By adding this smoothing function, the image quality of the 3-D linear interpolation is much improved. The hardware consumption is still much lower than the wavelet method.

Figure 6 shows the Perlin noise hardware. The inputs are $u$, $v$, $w$. The fraction, floor and ceiling values of each input are first calculated and are denoted by $ufrac$, $vfrac$, $wfrac$, $uint$, $vint$, $wint$, $uint + 1$, $vint + 1$, $wint + 1$, respectively. The function, $R(x_1, x_2, x_3)$, is implemented by blocks, labeled $random\ number$. The function, $I(x_{000}, x_{001}, \ldots, x_{111}, x_u, x_v, x_w)$, is implemented by blocks, labeled $sm$ and $Linear\ Interpolation$. $ufrac$, $vfrac$, and $wfrac$ are processed by the smoothing function, $sm$. The smoothing function implements the equation $sm(x) = 3x^2 - 2x^3$ in 10K50 EAB memory blocks [1]. The outputs of the smoothing function are denoted by $ufracs$, $vfracs$, and $wfracs$.

The internal structure of the $Linear\ Interpolation$ units is shown in Figure 7. Each unit implements the function $f(a, b, c) = a + c \times (b - a)$. This is a special case of the general linear interpolation formula, $g(x) = g(x_0) + \frac{g(x_1) - g(x_0)}{x_1 - x_0}(x - x_0)$, where $g(x_0) = a$, $g(x_1) = b$, $x_1 - x_0 = 1$, and $x - x_0 = c$. The input, $c$, must be a positive fraction value between 0 and 1. $a$ and $b$ are real numbers.

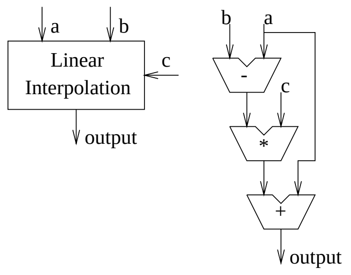


Figure 7: Linear Interpolation Unit

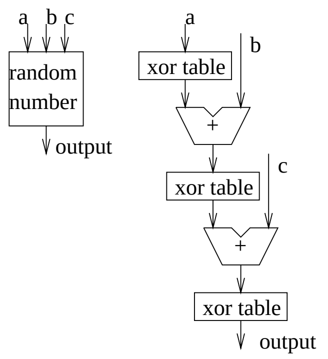


Figure 8: Random Number Generator

The internal structure of the $random\ number$ unit is shown in Figure 8. For a given set of inputs, the unit outputs a corresponding pseudo random number. The $xor$ tables shown in Figure 8 execute the function:

$$\begin{aligned} y_0 &= (x_0\ and\ r_{00})\ xor\ \ldots\ xor\ (x_n\ and\ r_{0n}) \\ y_1 &= (x_0\ and\ r_{10})\ xor\ \ldots\ xor\ (x_n\ and\ r_{1n}) \\ &\ldots \\ y_n &= (x_0\ and\ r_{n0})\ xor\ \ldots\ xor\ (x_n\ and\ r_{nn}) \end{aligned}$$

where $(y_n, y_{n-1}, \ldots, y_0)$ is the output bit vector, $(x_n, x_{n-1}, \ldots, x_0)$ is the input bit vector and

$$\begin{gathered} (r_{00}, r_{01}, \ldots, r_{0n}) \\ (r_{10}, r_{11}, \ldots, r_{1n}) \\ \ldots \\ (r_{n0}, r_{n1}, \ldots, r_{nn}) \end{gathered}$$

is a set of pre-generated constant bit vectors [15]. Since $r_{ij}$ is static, the entire $xor$ table can be implemented in around 8 LUTs. This is much less expensive than $256 \times 8$ RAM. The $xor$ table is used to scramble its input bits into a random value. This scrambling process is repeated three times to produce a random value for any point in space.

## 3.2 Perlin Noise Based 3-D Procedural Textures

This section discusses the implementation of marble and wood textures. Both use the turbulence fractal function.

### 3.2.1 Marble

The marble algorithm models the internal coloring of marble. As illustrated in Figure 9, marble is formed by layers of colored rock

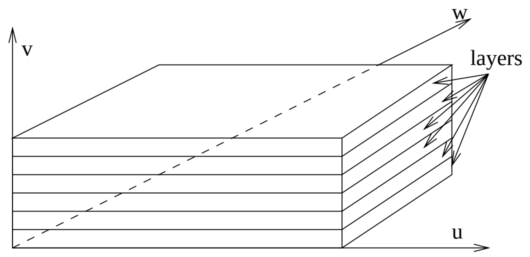


Figure 9: Marble Internal Structure

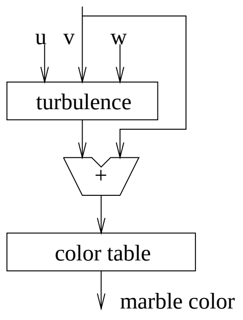


Figure 10: Procedural Texture Generator Configuration for the Marble Texture

deposits. Over time, different colored layers start to intermix with each other because of the extremely high pressure and the geological movements. This process generates the unique vein-like coloring inside the marble. This phenomenon is modeled by the function:

$$M(u, v, w) = (turbulence(u, v, w) + v)\ mod\ 128$$

$M(u, v, w)$ is used to index into a color table of 128 entries. The color table is configured to store the color of the various rock layers. Each color table entry represents the color of one layer; and the address of the entry corresponds to the layer position. When the color table is accessed according to $v$, the resulting 3-D texture image corresponds to the unmixed layers of marble. To simulate the intermixing of layers over time, the turbulence value is added to $v$.

The hardware for generating the marble texture is shown in Figure 10. The final marble texture mapped onto a cube is shown in Figure 11.

### 3.2.2 Wood

As illustrated by Figure 12, the internal structure of wood can be approximated by a series of cones that are have random perturbations. A tree grows one layer every year. The color within each layer varies with the seasons. The range of color within each layer is roughly the same from one layer to another.

Wood is modeled by the following function:

$$\begin{gathered} W(u, v, w) = ((u^2 + v^2 + \alpha w) + \\ turbulence(u, v, w))\ mod\ 128 \end{gathered}$$

$W(u, v, w)$ is used to index into a color table of 128 entries. The range of color within a single layer is stored in the color table. The basic shape of each cone is modeled by function $u^2 + v^2 + \alpha w$, which is a hyperbolic function of $u$ and $v$. The exact equation for cones is $\sqrt{u^2 + v^2 + \alpha w}$. The hyperbolic function is less expensive to compute, and also models the non-uniform growth of trees,

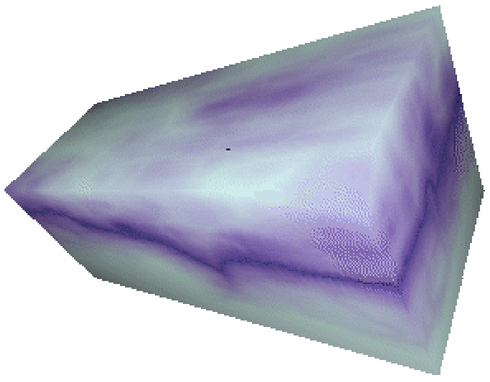


Figure 11: Marble Texture Mapped Cube

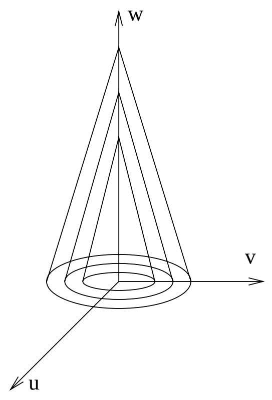


Figure 12: Wood Internal Structure

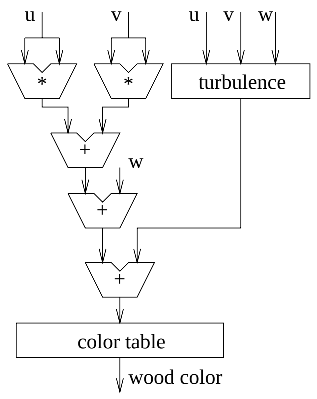


Figure 13: Procedural Texture Generator configuration for the Wood Texture

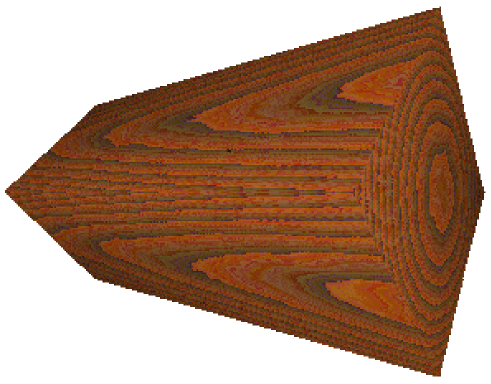


Figure 14: Wood Texture Mapped Cube

where young trees grow much faster than older ones. The $mod$ operation creates the layering effect of cones. Adding turbulence to $W(u, v, w)$ simulates the irregularity of tree growth.

The hardware for calculating the wood texture is shown in Figure 13. A wood texture mapped cube is shown in Figure 14. Notice that the pattern is realistic and consistent across all faces of the cube. This is more clearly shown in full color prints.

## 4 Performance and Hardware Cost

### 4.1 Performance

The portion of the rendering system implemented on the TM-2 uses two clock signals. The frame buffer uses a clock frequency of 25.0 MHz. This speed is mandated by the VGA monitor that the frame buffer controls. The rest of the system uses a clock frequency of 12.5 MHz. Under the 12.5 MHz clock, the system is able to produce one pixel for every four clock cycles. The WSST software is executed on a 296MHz UltraSPARC-II CPU. The software is able to keep up with the performance of the hardware.

The performance bottleneck for the rendering system is the STST unit. When implemented on its own, the procedural texture generator can be clocked at a much higher clock frequency. When measured in isolation from the rest of the system, the execution speed of all six textures is determined by the fractal function unit. On the TM-2, the generator can run at a maximum clock frequency of 28

| Textures | Look-Up Tables | Memory | Area | Area as % of 1 Gb of DRAM Area |
|---|---|---|---|---|
| Marble | 2839 | 1152 bits | $47mm^2$ | 4.1% |
| Wood | 3428 | 1152 bits | $57mm^2$ | 5.0% |
| Brick | 2870 | 1152 bits | $47mm^2$ | 4.1% |
| Fog | 2700 | 1152 bits | $45mm^2$ | 3.9% |
| Cloud | 3006 | 1152 bits | $50mm^2$ | 4.4% |
| Fire | 3152 | 5760 bits | $52mm^2$ | 4.5% |

Table 1: Area Cost of Implementing Procedural Texture Generator

MHz for all six textures, limited to 12.5 MHz by rest of the system. As designed, it can produce one pixel of texture for every four clock cycles. This performance is equivalent to 7 Million Pixels Per Second (MPPS). The system can fill 230K pixels per frame at 30 Hz frame rate.

### 4.2 Hardware Cost in Comparison to Memory Based Texture Mapping

In memory based texture mapping, large amounts of memory are required to store three-dimensional texture images. In this study, 3-D procedural textures are synthesized with a resolution of $512 \times 512 \times 512$. Eight bits are used to represent the color of each pixel. Since textures are accessed randomly by rendering engines, their can not be compressed by conventional compression techniques. Without any form of compression, each of these three-dimensional images requires 1 Gbit of storage memory. On the other hand, less than one and half 10K50 FPGAs are required to implement each texture. This section compares these two approaches to procedural texture mapping using silicon area as a yard stick.

Current state of the art technologies can package 256 Mbits of DRAM onto a $286mm^2$ die area using a $0.25\mu m$ process [17]. Using the same DRAM technologies, 1 Gbit of memory would require $1144mm^2$ of die. Altera 10K100 FPGAs are the latest implementation of the 10K50 architecture. Scaled to the same $0.25\mu m$ process, each logic array block of the 10K100 FPGAs consumes $132,000\mu m^2$ of silicon [3]. This area not only includes the area consumed by the look-up tables, but also the associated routing resources for each logic array block. Since each logic array block contains eight look-up tables, each look-up table consumes approximately $16,000\mu m^2$ of silicon. Besides logic array blocks, embedded memory blocks are also used in texture synthesis. Each embedded memory block contains 2048 memory bits; and one embedded memory block is approximately the same size as one logic array block.

The amount of FPGA resource consumed by each procedural texture is shown in column two and column three of Table 1. Two types of resources are consumed, the look-up tables and the embedded memory blocks. The total silicon areas consumed by these programmable logic resources are shown in column four. The fifth column of Table 1 shows the programmable logic area as a percentage of the area consumed by 1 Gbit of DRAM. For the texture algorithms investigated, the programmable logic implementations use 3.9% to 5.0% of the area required by the texture memory storing uncompressed textures of the same resolution. The FPGAs can achieve even higher area efficiency for algorithms with more input variables and larger texture spaces.

### 4.3 Single-Chip Graphic Accelerator with On-Chip Support for Perlin Noise based Procedural Texture Mapping

The experimental data and the wide spread use of Perlin noise function also suggest the possibility of synthesizing procedural textures in a mixture of ASIC and FPGA hardware. The combined ASIC+FPGA approach have the potential of synthesizing Perlin noise based textures at higher speed and with smaller silicon area cost. The ASIC+FPGA procedural texture generator might be small enough to fit on a single chip with the rest of the graphic accelerator. The possible floor plan for such an single-chip design is shown in Figure 15.

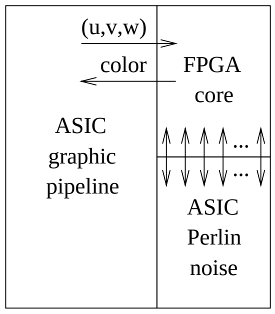


Figure 15: ASIC+FPGA Procedural Texture Mapping Organization

The difference between this approach and the pure FPGA approach is that the Perlin noise would be directly implemented in ASIC hardware, which has higher performance and higher logic density. Some other commonly used procedural texture functions might also be directly implemented in ASIC along with the Perlin noise. Only the remaining functions in procedural texture algorithms are required to be implemented in FPGAs. Table 2 shows the possible performance figure for the ASIC+FPGA implementation for six textures investigated. Table 3 shows the possible area consumption by the six textures. These data are measured by removing the Perlin noise function from these six textures and measuring the speed and hardware costs of the remaining FPGA circuits. It is assumed that the ASIC implementation of the Perlin noise function is able to keep up with the performance of the FPGA circuits.

## 5 Conclusions and Future Work

This paper has presented the architecture of a 3-D computer graphic rendering system which synthesizes 3-D procedural textures in FPGA hardware. The rendering system is implemented on the TM-2 digital prototype system. The prototype system executes at a speed of 12.5 MHz and can produce pixels at a rate of 3.125 MPPS. On the TM-2 system, only 3.9% to 5.0% of the silicon area that would be consumed by the texture memory is consumed by FPGAs implementing the procedural texture generator. The implementation also shown that the procedural texture generator can achieve high performance required by the animation applications.

| Textures | Max. Clock Freq. | MPPS | Frames Per Second |
|---|---|---|---|
| Marble | 125 MHz | 125 | 476 |
| Wood | 74 MHz | 74 | 282 |
| Brick | 47 MHz | 47 | 179 |
| Fog | wiring delay | Limited by ASIC | Limited by ASIC |
| Cloud | 43 MHz | 43 | 164 |
| Fire | 50 MHz | 50 | 190 |

Table 2: ASIC+FPGA Performance

| Textures | Look-Up Tables | Memory | FPGA Area |
|---|---|---|---|
| Marble | 147 | 0 bits | $2.4mm^2$ |
| Wood | 736 | 0 bits | $13mm^2$ |
| Brick | 178 | 0 bits | $2.9mm^2$ |
| Fog | 29 | 0 bits | $0.47mm^2$ |
| Cloud | 335 | 0 bits | $5.5mm^2$ |
| Fire | 481 | 4608 bits | $8.2mm^2$ |

Table 3: Area cost of FPGA Hardware in ASIC+FPGA Approach

There are three main areas of future work. First, it is a time-consuming job to manually translate procedural texture algorithms into hardware, especially when fixed point representation is used. CAD tools need to be developed to automate most of this translation process. Second, procedural texture algorithms contain many arithmetic computations. New programmable logic architectures can be developed to target at arithmetic applications, so procedural texture algorithms can be implemented in smaller and faster programmable hardware. Third, to make the concept of synthesizing procedural texture in FPGA hardware practical, more procedural texture algorithms need to be developed. More importantly these algorithms need to be efficiently implemented in programmable logic.

## 6 Acknowledgment

We would like to thank Dave Galloway for laying the ground work by designing a 2-D texture mapping system on the TM-2. We also like to thank him for all the TM-2 software and hardware support that he has provided.

We would also like to thank Marcus van Ierssel for designing the VGA interface card and maintaining and constantly improving the TM-2 hardware, Jonathan Rose for his technical input, and Vaughn Betz for providing area estimate on Altera 10K series FPGAs.

## References

[1] ALTERA. *Altera 10k FPGA Databook*.

[2] BERTIN, P., RONCIN, D., AND VUILLEMIN, J. Introduction to Programmable Active Memories. Tech. rep., Digital Equipment Corporation, June 1989.

[3] BETZ, V. *Architecture and CAD for Speed and Area Optimization of FPGAs*. PhD thesis, University of Toronto, 1998.

[4] BUELL, D. A., ARNOLD, J. M., AND WATER, J. *Splash 2: FPGAs in a custom computing machine*. IEEE Computer Society Press, Los Alamos, CA, 1996.

[5] CHEREPACHA, D., AND LEWIS, D. DP-FPGA: An FPGA Architecture Optimized for Datapaths. Tech. rep., University of Toronto, 1994.

[6] EBERT, DAVID S.AND MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., AND STEVEN, W. *Texturing and Modeling: A Procedural Approach*. AP Professional, Boston, 1994.

[7] FOLEY, J. D., HUGHES, J., VAN DAM, FEINER, AND HUGHS. *Computer Graphics: Principles and Practice*, second ed. Addison-Wesley, Reading, Mass, 1990.

[8] GALLOWAY, D. 2-D Texture Mapping on TM-2. Tech. rep., University of Toronto, 1996.

[9] GLEICK, J. *Chaos: Making a New Science*. Penguin Books, New York, 1987.

[10] KATZ, R. H. *Contemporary Logic Design*. Addison-Wesley Pub Co, 1990.

[11] LEWIS, D. M., GALLOWAY, D. R., IERSSEL, M. V., ROSE, J., AND CHOW, P. The Transmogrifier-2: A 1 Million Gate Rapid Prototyping System. *Transactions on VLSI* (1997).

[12] PEACHEY, D. Solid Texturing of Complex Surfaces. *Computer Graphics (SIGGRAPH '85 Proceedings)* (1985), 279–286.

[13] PERLIN, K. An Image Synthesizer. *Computer Graphics (SIGGRAPH '85 Proceedings) 19* (July 1985), 287–296.

[14] RAJAMANI, S., AND VISWANATH, P. V. Accelerating the RISC processor using Programmable Logic. Tech. rep., University of Berkely, 1992.

[15] RAU, B. R. Pseudo-Randomly Interleaved Memory. *ACM* (1991).

[16] RAZDAN, R. *PRISC: Programmable Reduced Instruction Set Computers*. PhD thesis, Harvard University, May 1994.

[17] WATANABE, Y., WONG, H., KIRIHATA, T., KATO, D., DEBROSSE, J. K., HARA, T., YOSHIDA, M., MUKAI, H., QUADER, K. N., NAGAI, T., POECHMUELLER, P., PFEFFERL, P., WORDEMAN, M. R., AND FUJII, S. A $286mm^2$ 256Mb DRAM with x 32 Both-Ends DQ. *IEEE Journal of Solid-State Circuits 31* (April 1996).

[18] WITTING, R. D. OneChip: an FPGA Processor with Reconfigurable Logic. Master's thesis, University of Toronto, 1995.