

# Real-Time Procedural Textures

John Rhoades, Greg Turk, Andrew Bell, Andrei State,  
Ulrich Neumann and Amitabh Varshney

Department of Computer Science  
University of North Carolina at Chapel Hill

## Abstract

We describe a software system on the Pixel-Planes 5 graphics engine that displays user-defined antialiased procedural textures at rates of about 30 frames per second for use in real-time graphics applications. Our system allows a user to create textures that can modulate both diffuse and specular color, the sharpness of specular highlights, the amount of transparency and the surface normals of an object. We describe a texture editor that allows a user to interactively create and edit procedural textures. Antialiasing is essential for real-time textures, and in this paper we present some techniques for antialiasing procedural textures. Another direction we are exploring is the use of dynamic textures, which are functions of time or orientation. Examples of textures we have generated include a translucent fire texture that waves and flickers and an animated water texture that shows the use of both environment mapping and normal perturbation (bump mapping).

## Introduction

The current trend in graphics libraries is to give users complete control of an object's surface properties by providing a language specifically for shading [Hanrahan & Lawson 90]. There are two lines of research that have come together to form modern shading languages. One line of research is the notion of programmable shaders, which has its roots in the flexibility of the shader dispatcher [Whitted & Weimer 82] and which was expanded to fully programmable shaders in [Cook 84]. The other research track is the use of mathematical function composition to create textures [Schachter 80] [Gardner 84]. These two lines of research were dramatically brought together to produce a mature shading language in the work of Ken Perlin [Perlin 85]. There are now several graphics machines fast enough to bring some of this flexibility to real-time graphics applications [Apgar 88] [Potmesil & Hoffert 89] [Fuchs 89]. This is the point of departure for our research.

The organization of this paper is as follows: a discussion of the pros and cons of procedural textures; an overview of the Pixel-Planes 5 hardware and software; a brief description of our

language for composing textures; an outline of the algorithms involved in displaying such textures on Pixel-Planes 5; a description of an interactive texture editor that dynamically displays a texture as the user changes its parameters; examples of dynamic textures; examples of applications that make use of the texture capabilities of our system; and future directions for this research.

## Why Use Procedural Textures?

Procedural textures provide an alternative to the choice of image-based textures. The central tradeoff between image and procedural textures is between memory cost and execution time.

Graphics architectures that are well-suited for displaying image textures typically have large amounts of memory associated with a handful of fast processors. Each processor retains a copy of every image texture for a given scene so that any processor can perform the texture look-up at any given pixel in the scene. Texture evaluation thus has a small, fixed computational cost, at the expense of using large amounts of memory to store the texture copies. The Silicon Graphics Skywriter and the Star Graphicon 2000 are two commercial graphics engines that use this approach with impressive results.

Our implementation of procedural textures on Pixel-Planes 5 provides a look at the opposite end of this spectrum. Each pixel processor has only 208 bits of memory, but the graphics machine may be configured to have on the order of 256,000 pixel processors, giving the ability to perform several billion instructions per second. Their very small memory makes the pixel processors poor for rendering image-based textures but their computational power makes them ideal for generating procedural textures on-the-fly.

It is clear that any procedural texture can be computed once, saved as an image, and used in a scene like any other image texture. In this sense, it can be argued that image-based textures offer everything that procedural textures can provide, with the only additional cost being the use of more memory. Also, it is clear that procedural textures are a poor choice when the scene requires a picture hanging on the wall or an image on the cover of a book. Nevertheless, procedural textures do have benefits of their own. One benefit is that the texture can be arbitrarily detailed, provided that the texture coordinates are represented with enough bits. Each additional bit added to computation of a function of two variables is reflected by a

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1992 ACM 0-89791-471-6/92/0003/0095...\$1.50

factor of four in memory cost to mimic the texture with a stored image. A more dramatic benefit is the ability to define textures which are functions of many variables, such as animated textures and solid textures. The memory capacity of graphics systems that we are familiar with is not large enough to explicitly store such textures. Pixel-Planes 5 offers us the alternative of evaluating on demand the values from textures of several variables.

## Pixel-Planes 5 Overview

**Hardware** - The Pixel-Planes 5 machine has multiple Intel i860-based Graphics Processors (GPs) and multiple SIMD pixel processor arrays called Renderers. A Renderer is a 128x128 array of bit-serial pixel processors, each with 208 bits of local memory, called *pixel memory*, and 128x32 bits of off-chip backing store memory. Each Renderer can be mapped to any 128x128 pixel region of an image. The Renderer processors are capable of general arithmetic and logical operations and operate in SIMD mode. Each processor has an enable bit that regulates its participation in instructions. Graphics Processors, Renderers, Frame Buffers, and workstation host communicate over a shared 640 Mb/sec ring network.

**Software** - Generating images with textured polygons on Pixel-Planes 5 is a multi-stage process which can be viewed as a graphics pipeline [Fuchs 89] as shown in Figure 1. Transparent polygons are handled by making multiple passes through the pipeline. In the first stage of the graphics pipeline, the Graphics Processors transform the polygon vertices from model space to perspective screen space and create SIMD instruction streams (*Image Generation Controller* or *IGC commands*) for the Renderers to rasterize the polygons. A Z-buffer algorithm is executed in parallel for all pixels within a polygon. During rasterization, intrinsic color components, surface normals, texture u,v coordinates, texture scale factor (used for antialiasing), texture-id, etc., are stored in the pixels. After rasterization of all polygons, each pixel processor has the parameters of its front-most polygon. These parameters are then used in the next two stages of the pipeline: texture program interpretation and lighting model computation. At the beginning of texture program interpretation, some initialization is performed. The rasterization phase actually

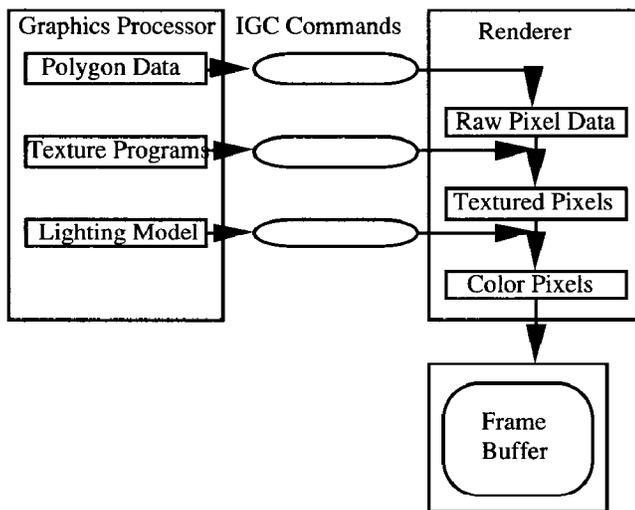


Figure 1: Pixel-Planes 5 Graphics Pipeline

stores  $u \cdot z$  and  $v \cdot z$  rather than  $u$  and  $v$  in pixel memory (since  $u \cdot z$  and  $v \cdot z$  are linear in screen space), so a  $z$  division is needed. Also a time value is stored in pixel memory for use in animated textures. The lighting model currently used is Phong shading. Since all pixels are handled concurrently after all rasterization, we call this approach *deferred shading*. Because of the high degree of parallelism achieved during deferred shading, we can afford to have quite elaborate procedural textures and lighting models while maintaining high frame rates.

## Texture Programs

**Programming Model** - Procedural textures are implemented via a simple virtual machine. This texture machine comprises an assembly language-like instruction set called *T-codes*, a set of registers in pixel memory, and a set of parameters in the Graphics Processor memory. The pixel parameters, such as intrinsic color,  $u,v$  coordinates, etc., are accessible to the texture machine via its pixel memory registers. The Graphics Processors execute the T-codes interpretively, modifying the pixel variables that affect shading. More exactly, interpretation of a T-code program produces an IGC command instruction stream, which is routed to the appropriate Renderers for SIMD execution.

**T-Codes** - There are three kinds of T-codes: generators, which produce several basic texture patterns, operators, which perform simple arithmetic operations on texture patterns, and conditionals which permit selected pixels to be included or excluded in a computation. Generators include Perlin's band-limited noise function [Perlin 85], Gardner's sum-of-sines [Gardner 84], antialiased square waves, and a Julia set. Examples of operators include add, scale, max, square root, splines, and color table lookup. These operators can be cascaded to implement arbitrary functional composition. There are T-codes for conditional execution (by having selected pixel processors conditionally disable themselves), but no T-codes for looping. Adding a new T-code to our system is a straightforward task. Besides coding and testing of the T-code subroutine in C, the programmer needs only to update the T-code assembler parse table and the T-code subroutine dispatch table.

**Sample Texture Program** - The following T-code fragment computes an antialiased black and white checkerboard pattern. The  $U$  and  $V$  registers contain the texture coordinates, and the  $D$  register contains the texture scale factor. Output is to the diffuse color components  $D\_Red$ ,  $D\_Green$  and  $D\_Blue$ . The swave generator produces antialiased square waves in one dimension. Note how the outputs of the generators are combined by continuous operators for antialiasing, rather than using bitwise exclusive-OR.

```

# make antialiased square wave in U direction
  swave R,U,D; swave_params
# make antialiased square wave in V direction
  swave S,V,D; swave_params
# R and S registers now contain stripes
  mul   T,R,S
  add   W,R,S
  sub   W,W,T
  sub   W,W,T
# W := R+S-2*R*S, continuous exclusive OR
# set diffuse colors from W
  copy  D_Red,W
  copy  D_Green,W
  copy  D_Blue,W
  
```

Certainly, our texture programming language is hardly state-of-the-art with respect to programming ease. This is compensated to some extent by the fact that texture programs tend to be rather short - typically 20-40 instructions. The programs are short because the built-in generators and some of the operators (such as spline and color table lookup) are fairly powerful. The main job of the texture programmer is producing the appropriate "glue" code to tie these together. In addition, as discussed later, programming is facilitated by an interactive texture editor program which allows the use of macros.

## Texture Procedure Evaluation Details

**Pixel Memory Management** - The Pixel-Planes 5 Renderers contain 208 bits of on-chip memory and 4096 bits of off-chip backing store memory per pixel. Backing store memory cannot be directly addressed by IGC instructions, but must be swapped in and out by special instructions. Because texture programs usually require the use of scratch memory space and because a rather large number of pixel variables are needed to support deferred shading, there is not enough pixel memory to statically allocate it for the worst case. Therefore, a pixel memory manager keeps track of the locations of the variables and to perform memory movement and backing store swapping to make available required amounts of scratch memory space.

**Caching of IGC Commands** - For static texture programs, the IGC commands do not change from frame to frame, and thus the T-code translation step need occur only once. Note that static texture programs do not imply static textures; the result of executing a texture program may vary with time, if time is an input variable. During texture parameter editing, the T-code program must be reinterpreted each time it is changed. The Graphics Processors cache the IGC commands resulting from texture interpretation to avoid generating them repeatedly.

**Region-Hit Flags** - Since each Renderer covers a small (128x128 pixel) region of the screen, it is likely that only a small subset of the textures will be represented in a given region. The Graphics Processors flag each region that any textured polygon intersects as needing that particular texture. The Graphics Processor that creates the texturing commands for a particular region checks the OR'ed flags from all Graphics Processors for that region, and creates and sends the texture programs for only those textures that might be visible.

## Obtaining Real-Time Performance

Our goal for real-time procedural textures was to deliver at least 15 frames-per-second to real applications in research projects at UNC. This goal has been met, and these applications are described in a later section.

There are two crucial issues for rapid texture evaluation. The first issue is to maximize utilization of the pixel processors. This is achieved by waiting to execute the texture programs until all polygons have been rasterized, so parallelism of the texture programs can be maximized. In addition, by use of region-hit flags, we avoid processing texture programs for screen regions that don't have the texture. The second issue is enabling the Graphics Processors to keep up with the Renderers. This is accomplished by the IGC instruction caching. We increased the performance of the Walkthrough application from 2 to 20 frames/sec by the use of region-hit flags and the IGC instruction caching.

## Antialiasing Techniques

Antialiasing of procedural textures is a difficult problem to which we have not found a general solution; instead we have developed a few techniques which work fairly well for many texture programs. The theoretically proper method is to convolve the texture with a filter kernel of an appropriate shape, centered at the pixel. In principle, this is possible since each pixel processor knows the entire texture, but in practice, this can be done only for the simplest textures, because integrating arbitrary functions of two variables is difficult.

In order to do antialiasing, we need some estimate at each pixel of how an area element in screen space maps into texture space. Ideally, we would use the derivatives of  $u$  and  $v$  with respect to screen space  $x$  and  $y$ . However because of limited pixel memory, we decided to record this estimate using a single number, called the *texture scale factor*. This number is intended to represent the maximum magnification factor that can occur when a unit vector in screen space is mapped to texture space. The texture scale factor is available in a pixel memory register for use in T-code programs. The approximation we use is  $\max(|u_x|+|u_y|, |v_x|+|v_y|)$ , which is within a factor of 1.42 of the commonly used formula  $\max((u_x^2+u_y^2)^{1/2}, (v_x^2+v_y^2)^{1/2})$  for MIPmaps [Williams 83]. Because of this, our textures are over-blurred when viewed at certain angles, just like MIPmaps. Texture scale factor is computed for polygons as follows. When the polygon is rasterized, the  $u$  and  $v$  coordinates at the middle of the polygon,  $u_{mid}$  and  $v_{mid}$  are computed. The linear expression for  $uz = ax+by+c$ , is differentiated to give  $u_z z + u_z = a$ , which is solved for the constant  $u_z z = a - u_{mid} z_x$ . Similarly  $u_z z$ ,  $v_z z$ , and  $v_z z$  are computed. From these  $\max(|u_x z|+|u_y z|, |v_x z|+|v_y z|)$  is computed and stored in pixel memory. Finally, just before texture program evaluation, a parallel  $z$  divide is performed for all pixels. This is, of course, an approximation due to the substitution of  $u_{mid}$  for  $u$ . The approximation error manifests itself as a difference in the amount of blurring at the corners of a polygon that is being viewed at a very oblique angle (large  $z$ ). We found that the error is not noticeable in ordinary scenes, although it can be seen in contrived test cases.

The antialiased square wave generator produces an antialiased stripe pattern with a specified phase, frequency, and duty cycle. The generator analytically computes the convolution integral of a box filter kernel with a square wave function of its input parameter in one dimension. The width of the box filter is the texture scale factor. Initially we implemented a triangular filter kernel, but found that it required too much scratch pixel memory.

A method that works for some textures is to antialias the final color table lookup. The idea is to return a final color that is the integral over some finite interval in the color table, rather than a point sample. The width of the integration interval is proportional to the texture scale factor times the maximum gradient magnitude of color with respect to  $u$  and  $v$ . This integral is simple enough to be computed analytically in the pixel processors. If the gradient magnitude of the texture value input to the color table is reasonably smooth, this roughly approximates the correct convolution integral, and does a fairly good job in practice for many textures. It fails utterly for textures that are discontinuous functions of  $u$  and  $v$ . This kind of texture gradually loses contrast as the texture scale factor increases, but before the texture fades to a uniform color, there is severe aliasing.

Another method works in the frequency domain. Some of our texture programs “roll off” the amplitude of the band-limited noise based on the texture scale factor. The result is that the noise fades to a uniform value at scales where aliasing would be a problem.

### Interactive Editing of Texture Procedures

An interactive texture editor eliminates the need for an edit-compile-link-test cycle. Since T-code programs are executed interpretively at run time, texture procedures can be changed without recompilation. Furthermore, the interpretation phase is fast enough so that literal values (Graphics Processor parameters) in T-code instructions can be updated in a single frame time, at frame rates of more than 30 frames per second. The texture editor displays the T-code instructions of a selected procedural texture in a text window. The user can position a movable cursor on any literal value in a T-code instruction, and then smoothly vary this value via a joystick. The dynamically updated texture pattern is displayed on the graphics system with a two-frame lag (the graphics pipeline overlaps two frames). At over 30 frames per second this lag time is hardly noticeable. Hence the user can explore the parameter space of a texture procedure continuously in real time.

More drastic changes to texture programs can be made by interactively editing the text of the program in another window via a conventional text editor. T-code instructions can be added, rearranged, and deleted, producing a new program. Then with a couple of commands, the user can save the updated texture program and reload it into the texture editor for immediate display. This process takes from one to five seconds, which due to the more discrete nature of such changes, can still be viewed as interactive editing.

What the user sees on the graphics system is a complete scene with possibly many graphics primitives and texture procedures, not just a single isolated texture pattern. The texture editor provides a complete set of commands to access the facilities of our graphics library. Thus the user can change the viewpoint, move objects around, change the locations and parameters of light sources, etc. This is important, because the appearance of a texture is dependent on its visual context.

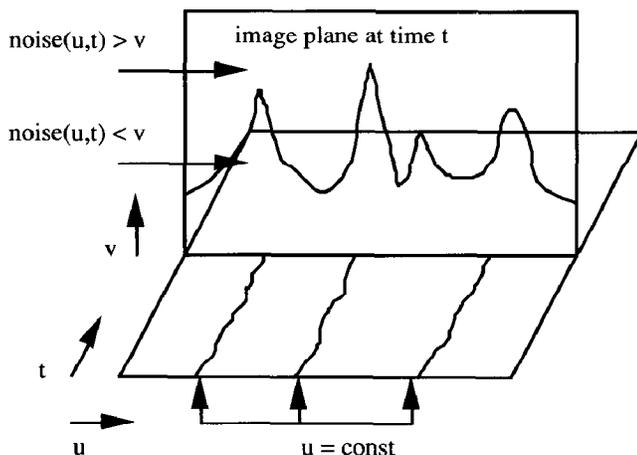


Figure 2: Generating Flames

### Dynamic Textures

Textures have been traditionally considered to be functions of spatial coordinates  $u$  and  $v$ . A generalized texture, however, need not be restricted to just mappings from the spatial coordinates. One could consider a texture to be a function of several other parameters as well - time and surface normal, to mention just a couple. Procedural textures permit us to create these generalized textures without the memory overheads that would be required with image textures. Since these textures change spatially based on input parameters that need not be restricted to just those that define the mapping, we prefer to call them *dynamic* textures.

If we consider a textures to be a function of  $u$ ,  $v$ , and  $t$ , where  $t$  is a time variable, we can produce time-varying *animated* procedural textures such as a fire texture that flickers and water waves that ripple. If we consider textures as functions of  $u$ ,  $v$  and  $n$  where  $n$  is the normal to the surface that has been textured, then it is possible to do environment mapping by defining an appropriate procedural texture. Dynamic textures implemented this way can still be precomputed because the program text for the texture doesn't change. Another way to produce dynamic textures is to edit the texture programs after each frame, but then there is some loss of performance since precomputation of IGC commands isn't possible. In the following sections we describe how we implemented several dynamic textures.

**Fire** - An example of an animated texture is a flickering flame. We implement a fire texture as follows (Figure 2): First perturb  $u$  by adding to it a 2D noise function of  $u$  and  $t$ . Then generate a height field  $h$  by applying a 2D noise generator to  $u$  and  $t$ . Compute flame intensity  $f = 1 - v/h$ . If  $f < 0$  set  $f$  to 0. This creates a moving outline of the flame. Because of the noise perturbation of  $u$ , the outline moves both vertically and horizontally. Finally we copy  $f$  to opacity and use a color table with input  $f$  to produce color. We use two layers of transparent fire texture to produce the fireplace shown in Photo 3.

**Environment Mapping** - The next example is a dynamic texture depending on object orientation instead of time. It implements environment mapping of a simple checkerboard pattern onto a teapot. The textured teapot appears to be located inside a room with checkerboard walls, as shown in Photo 5. Rotating the object lets the reflections move across the surface in a realistic way. We accomplish this by performing typical environment mapping computations [Blinn & Newell 76] (determine reflected eye vector, compute indices, compute procedural texture as function of indices) in a T-code program for each pixel.

Our current system has two limitations for environment mapping. First, because the normal vector is only available in eye space coordinates, the (infinitely distant) reflective environment appears to be attached to the camera. Thus, whenever the camera is rotated (panned, tilted or rolled), the reflections move across the object's surface in an erroneous way. If we had enough pixel memory to store world space normals this restriction could be removed. Second, we cannot perform antialiasing properly, since we do not have surface curvature information available in pixel memory.

**Water** - The final example, shown in Photo 6, is an animated texture approximating water waves by means of an animated procedural bump map. This dynamic texture is a function of both time and spatial orientation. The pixel normals are

perturbed on the basis of a height field whose value is computed at each pixel. The derivatives required for the normal perturbation are computed by finite differences. The height field consists of superimposed circular and parallel moving sinusoidal waves generated by a number of sources distributed across the water-textured surface, a common approach for this problem. The surface characteristics are such that the water surface appears highly specular. In addition, the normals are used to compute a simple one-dimensional color scale environment map, which is used to create a more natural appearance. The map has rotational symmetry about a vertical axis, so that the camera can be arbitrarily panned. However, tilting or rolling the camera would generate erroneous results, for the reasons mentioned in connection with the environment mapping texture. As mentioned, this restriction could be removed by storing world space normals at each pixel. We also have a problem with determining which way to perturb the surface normals, since we do not have the surface tangent vectors in the *u* and *v* directions available in pixel memory. We can circumvent this problem for horizontal polygons (like water surfaces) by broadcasting the current transformation matrix to the pixel processors during the texturing phase of each end-of-frame calculation. The scene in Photo 6 was rendered in 33 milliseconds, low resolution, with 24 GPs and 12 Renderers.

### Applications Using Procedural Textures

Pixel-Planes 5, besides being a research project in its own right, is also an important resource for several other research projects at UNC. Two of those for which textures are important are the Building Walkthrough project and the Head-Mounted Display project. Both of these use a stereo head-mounted display and head tracking, so high frame rates are necessary to maintain the illusion of the virtual environment.

**Walkthrough** - The UNC Walkthrough Project aims at the development of a system for creating virtual building environments [Brooks 86]. This is intended to help architects and their clients explore a proposed building design prior to its construction, correcting problems on the computer instead of in concrete. Texturing plays an important role in enhancing image realism. Having textures for bricks, wood, ceiling tiles, etc., adds to the richness of the virtual building environment and gives an illusion of greater scene complexity. The radiosity illumination model is used in the Walkthrough project. We can display a model of a house that contains about 34,000 polygons and 20 procedural textures at 15-20 frames/sec on 24 Graphics Processors and 12 Renderers at 640×512 resolution. Photo 1 shows a view of the living room of the house, and Photo 2 shows a view of the kitchen.

For enhanced realism, textures have been integrated with radiosity in Walkthrough. There are two stages in this integration. The first stage is to calculate radiosity values for a textured polygon, such that the radiosity effects such as color bleeding are correctly simulated for the polygons near this textured polygon. The second stage is to shade the textured polygon itself by the radiosity values at its vertices. To effect the first step, the color of a textured polygon is assigned to be the average color of its texture. This color is then used in the radiosity process as usual. After the radiosity values at the vertices of a polygon have been computed, they are passed as input parameters to the procedural texture for this polygon along with other input parameters such as the *u* and *v* coordinates. These shading values are linearly interpolated across the polygon. The procedural texture is computed as before and a post-multiplication of the interpolated radiosity

shading values with the computed texture colors at each pixel gives a smooth shading effect over the textured polygon.

Another application which textures find in the Walkthrough project is that they offer one way to switch lights in a virtual building. The total radiosity illumination of a polygon is determined by the dot-product of the vector of light values and the radiosity vector specifying the contribution of each light source to the illumination of the polygon. This then means that given the latter, the user can vary the intensity of a light source and observe the same building model under different light scales (but same light positions), by just computing the dot product as described before [Airey90]. This however takes roughly 3 - 5 seconds for a dataset of roughly 30,000 polygons and 20 light sources if done sequentially on the host workstation and fails to provide the effect of instantaneous light switching. One possibility to do this fast enough to provide an instantaneous effect (under a tenth of a second) is to do this in parallel by using T-codes. The idea is to pass the intensity value of a light source as an input parameter to a T-code program (along with the polygon colors) which computes the dot-product of the input parameter with the value of the interpolated radiosity (as described in the preceding paragraph) and uses the resulting value to shade the polygon. Changing the intensity of a light source can then be done by editing the T-code program and changing this input parameter. This is essentially using the T-code commands as a shading language.

**Head-Mounted Display** - In the Head-Mounted Display project, the primary use of textures has so far been in a mountain bike simulation, where the user rides a stationary bicycle and views simulated terrain through the head-mounted display. Textures are used to increase the apparent scene complexity and to improve the user's perception of motion through the environment. This application features relatively few textures (grass, road, and cloudy sky), each of which covers a fairly large area of the images. A scene from this application is shown in Photo 4. The cloudy sky texture makes use of the Gardner texture generator. The grass and road texture make use of band-limited 2-D noise, and are antialiased by decreasing the noise amplitude as the texture scale factor increases. Several frequencies of noise are used, each with its own threshold for rolloff. This simulation runs at 20-25 frames per second in low resolution (640×512) stereo mode using 32 Graphics Processors and 20 Renderers.

### Future Work

The logical next step to our simple texture language is to implement a full-fledged shading language that can be executed on-the-fly. Using the deferred shading paradigm on a high-end graphics machine, real-time execution of a shading language such as Renderman [Hanrahan & Lawson 90] seems to be a very real possibility. Unfortunately, this is impractical on the current Pixel-Planes system due to the small amount of memory available to the pixel processors. However, it is likely that the Pixel-Flow machine [Molnar 91], now being designed at UNC Chapel Hill, will have sufficient pixel memory to make this idea viable.

### Acknowledgements

We would like to acknowledge the support and cooperation of Pixel-Planes 5, Walkthrough, and Head-Mounted Display research project teams at UNC. In particular, we would like to thank the following individuals: David Ellsworth, Trey Greer, and Brice Tebbs for their help on the initial stages of the

design of texture algorithms, Eric Erikson for his work on the virtual bike, John Alspaugh for modeling the house model in Walkthrough and Carl Mueller for the Pixel-Planes frame-saving facility. We would also like to acknowledge Professors Henry Fuchs, Frederick Brooks, and John Poulton for their encouragement and support.

This work has been supported in part by the following grants: NSF MIP-9000894 DARPA order No 7510, DARPA Grant No: DAEA 18-90-C-0044, NSF Cooperative Agreement No ASC 8920219, NSF Walkthrough Grant No: CCR 8609588, ONR Grant No N00014-86-K-0680.

## References

[Airey 90] Airey, J.M., J.H. Rohlf and F.P. Brooks, Jr. "Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments" *ACM Computer Graphics* (Proceedings 1990 Symposium on Interactive 3D Graphics), Vol. 24, No. 2, pp 41-50.

[Apgar 88] Apgar, Brian, Bret Bersack and Abraham Mammen, "A Display System for the Stellar Graphics Supercomputer Model GS1000," *Computer Graphics*, Vol. 22, No. 4, (SIGGRAPH '88), pp. 255-262.

[Blinn & Newell 76] Blinn, J. F. and M. E. Newell, "Texture and Reflection for Computer Synthesized Pictures," *Communications of the ACM*, Vol. 19, No. 10, pp. 542-547.

[Brooks 86] Brooks, F. P., Jr., "Walkthrough- A Dynamic Graphics System for Simulating Virtual Buildings," *Proceedings of the 1986 Workshop on Interactive 3D Computer Graphics*, pp. 9-21.

[Cook 84] Cook, Robert L., "Shade Trees," *Computer Graphics*, Vol. 18, No. 3, (SIGGRAPH '84), pp. 223-231.

[Fuchs 89] Fuchs, Henry, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs and Laura Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *Computer Graphics*, Vol. 23, No. 3, (SIGGRAPH '89), pp. 79-88.

[Gardner 84] Gardner, Geoffery Y., "Simulation of Natural Scenes Using Textured Quadric Surfaces," *Computer Graphics*, Vol. 18, No. 3, (SIGGRAPH '84), pp. 11-20.

[Hanrahan & Lawson 90] Hanrahan, Pat and Jim Lawson, "A Language for Shading and Lighting Calculations," *Computer Graphics*, Vol. 24, No. 4, (SIGGRAPH '90), pp. 289-298.

[Molnar 91] Molnar, Steve, "Image Composition Architectures for Real-Time Image Generation," Ph.D. thesis, Computer Science Department, University of North Carolina at Chapel Hill, 1991.

[Perlin 85] Perlin, Ken, "An Image Synthesizer," *Computer Graphics*, Vol. 19, No. 3, (SIGGRAPH '85), pp. 287-296.

[Potmesil & Hoffert 89] Potmesil, Michael and Eric M. Hoffert, "The Pixel Machine: A Parallel Image Computer," *Computer Graphics*, Vol. 23, No. 3, (SIGGRAPH '89), pp. 69-78.

[Schachter 80] Schachter, B. J., "Long-crested Wave Models," *Computer Graphics and Image Processing*, Vol. 12, pp. 187-201.

[Whitted & Weimer 82] Whitted, Turner and David M. Weimer, "A Software Testbed for the Development of 3D Raster Graphics Systems," *ACM Transactions on Graphics*, Vol. 1, No. 1, pp. 44-58.

[Williams 83] Williams, Lance, "Pyramidal Parametrics," *Computer Graphics*, Vol. 17, No. 3, (SIGGRAPH '83), pp. 1-11.