5.8

Operations for Hardware-Accelerated Procedural Texture Animation

Greg *James, NVIDIA Corporation*

gjames@nvidia.com

Consumer-level 3D accelerators have come a long way in recent years. Today's most advanced chips sample four textures per pass and offer powerful texture addressing and pixel processing operations. Among these are dependent texture reads, where the color value sampled from one texture is used to perturb the coordinates of a subsequent texture read. Dependent texture reads combined with the ability to render to a color buffer and use that buffer as a source texture in later rendering make it possible to generate interesting textures and texture animations entirely on the graphics processor. With texel sampling rates close to one billion per second, these procedural texture effects run very fast and are practical for real-time 3D-accelerated scenes.

Techniques for procedural texture generation have been with us from the early days of computer graphics. There are various algorithms for mimicking natural phenomena and generating complex emergent patterns [Ebert98]. Procedural texture animation is excellent for a wide range of effects while using a tiny fraction of the memory and storage that a prerendered or "canned" animation would require. Memory to hold two or three full frames is often all that is required to generate an endless non-repeating animation. User input can be applied to these animations on the fly, and this interactivity enables richer and more engaging virtual scenes.

This gem covers a few fundamental operations for generating procedural animations, and puts these to use in specific examples that simulate fire, smoke, water, or perform image processing. With today's consumer hardware, we can even run complex cellular automata programs entirely within the rendering of a 3D accelerator and put the resulting animations to use in various effects.

Hardware Operations

We will focus on two rendering operations that give rise to interesting procedural texture effects. The first is four-sample texture sampling from adjacent texels of an image, and the second is 2D dependent green-blue texture addressing. These operations are supported in the two most common APIs: OpenGL and Microsoft's

DirectXS. For the four-way multisampling, we will use a vertex program that is loaded into the graphics processor and operates on the incoming vertex position and texture coordinates. These vertex programs are DirectXS's "Vertex Shaders" [DXS'OO] and what is currently the NVIDIA NV_vertex_program extension to OpenGL [NVExt2001]. For our examples, this vertex program establishes the appropriate coordinates for neighbor texel sampling in each of the four texture stages that feed the pixel engine. This pixel engine will combine the samples for various effects, and can also further manipulate the samples with the dependent texture addressing operations. The pixel engine is also programmable as exposed through DirectX 8's "Pixel Shader" programs and NVIDIA's NV_texture_shader extension. First, we examine the vertex processing operations.

Neighbor Sampling for Blur, Convolution, and Physics

Many procedural texture algorithms rely on sampling a texel's neighbors and filtering or blurring these samples to create a new color value. We can accomplish this neighbor sampling for all the texels of a source texture by rendering it into a color buffer of the same resolution as the texture while using four-texture multisampling. The source texture is selected into all texturing units, and a vertex program generates four sets of independent texture coordinates, each offset by the distance to one of the texels neighbors. By rendering a single quad with texture coordinates from 0.0 to 1.0, which exactly covers the render target, and setting each texture coordinate offset to zero, each texel of the destination would sample from its single source texel four times. We would end up with an exact copy of the source texture. By offsetting each texture coordinate by a vector to a neighboring texel, each pixel of the destination samples from four of its neighbors in the source. We can convolve the samples (combine them with various scale factors) in the pixel engine however we like.

If we use point sampling, four-sample multitexturing hardware gives us up to four neighbor samples per pass. If we enable bilinear filtering, each offset sample draws upon four texels, so we could potentially sample 16 neighbors per pass. The weighting of texels within each 2x2 bilinear sample is determined by the precise texture coordinate placement. For example, we could grab and average all eight of a texels nearest neighbors by placing four bilinear samples exactly between pairs of neighboring texels. Figure 5.8.1b shows this, and Listing 5.8.1 presents code for neighbor sampling on four-sample multitexturing hardware. The 'X' of Figure 5.8.1 denotes the texel being rendered to the destination, and the dots indicate the sample locations from the source texture used to render it. We can select between various sets of sample locations (A or B in this case) by using the vertex shader indexing variable aO. x. The value of the indexing variable is loaded from a shader constant value that is set before rendering.

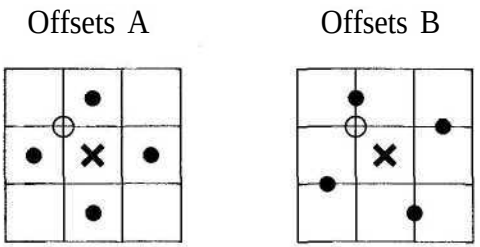


FIGURE 5.8.1 *Texel sampling for the code of Listing 5.8.1, showing the sample pattern when the texel marked by the X is being rendered. The vertex program's texture coordinates as iterated in each texture unit T[0-3] are marked with dots. The hollow circle marks where a sample of texture coordinate offset (0.0, 0.0) would fall, illustrating the need for the* s_of f *and* t_of f *half-texel offsets. Each texel is of dimension (si, tl) according to Listing 5.8.1.*

Listing 5.8.1. Code and vertex program for sampling each texel's neighbors

RenderFullCoverageQuadO renders a single quad with input texture coordinates from 0.0 to 1.0 in each axis, which exactly covers the render target. These coordinates are offset four ways into the four output oT[0-3] texture coordinates so that as each pixel is rendered, it draws upon neighboring texels for its result. All-caps variables are #defines to appropriate indices into vertex shader constant memory.

```
float s1 = 1.0f / texture_resolution_x;  // one texel width
float t1 = 1.0f / texture_resolution_y;  // one texel height
float s_off =  s1 / 2.Of;     //to sample texel center
float t_off =  t1 / 2.Of;

// s,t,r,q offsets for 4 nearest neighbors (bilinear or point)
float offset_al[4] = { -si  + s_off, O.Of  + t_off, O.Of,
                       O.Of};
float offset_a2[4] = {  s1  + s_off, O.Of  + t off, O.Of,
                       O.Of};
float offset_a3[4] = { O.Of + s_off,  t1   + t off, O.Of,
                       O.Of};
float offset_a4[4] = { O.Of + s_off, -t1   + t_off, O.Of,
                       O.Of};

// s,t,r,q offsets for 8 surrounding neighbors (use bilinear)
float offset_bl[4] = {  s1/2.0f + s_off,  tl     + t_off, 0.0f,
                        0.0f};
float offset_b2[4] = { -s1      + s_off,  t1/2.0f + t_off, 0.0f,
                        0.0f};
float offset_b3[4] = { -s1/2.0f + s_off, -t1     + t_off, 0.0f,
                        0.0f};
```

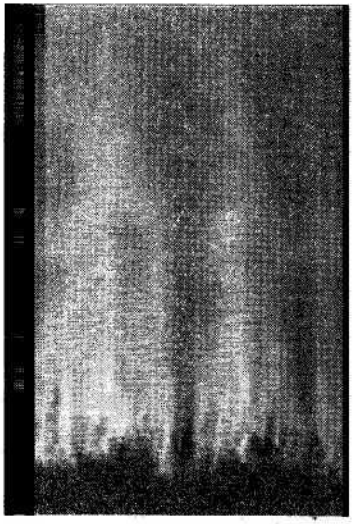
```
float offset_b4[4] = { s1      + s_off, -tl/2.0f + t_off, O.Of,
                       O.Of};

SetVShaderConstants((TO_BASE .. T3_BASE) + SET_A, offset_a1 ..
                    offset_a4);
SetVShaderConstants((TO_BASE .. T3_BASE) + SET_B, offset_b1 ..
                    offset_b4);
SetVShaderConstants( OFFSET_TO_USE,  use_a ? SET_A : SET_B );
RenderFullCoverageQuad();
```

Vertex Program

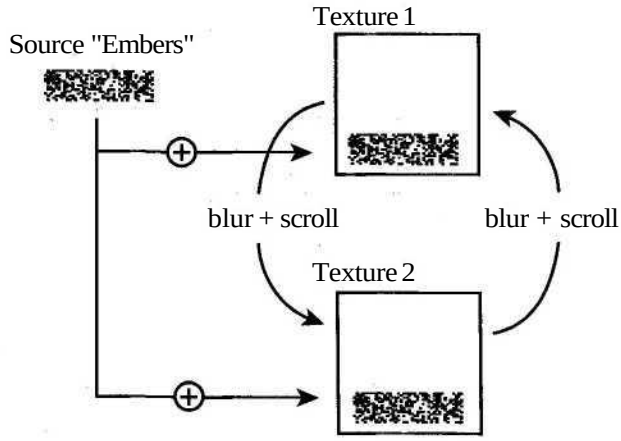
```
; vO = vertex position
; v1 = vertex texture coordinate

; Transform vertex position to clip space.  4-vec * 4x4-matrix
dp4  OPOS.X,  vO,  C[ WORLDVIEWPROJJ) ]
dp4  oPos.y,  vO,  C[ WORLDVIEWPROJ_1 j
dp4  OPos.z,  VO,  C[ WORLDVIEWPROJ_2 ]
dp4  OPOS.W,  vO,  C[ WORLDVIEWPROJ_3 ]

; Read which set of offsets to use - set A or B
mov  aO.x,  c[OFFSET_TO_USE ].x

; Write S,T,R,Q coordinates to all four texture stages, offsetting
;  each by either offset_a(1-4) or offset_b(1-4)
add  oTO,  v1,  c[  aO.x  +  TO_BASE  ]
add  oT1,  v1,  c[  aO.x  +  T1_BASE  ]
add  oT2,  vl,  c[  aO.x  +  T2_BASE  ]
add  oT3,  v1,  c[  aO.x  +  T3_BASE  ]
```

It is important to note that for this case of rendering a full coverage quad to a buffer with the same resolution as the source texture, a texture coordinate offset of (0,0) would sample from the upper left (lowest coordinate point) of each texel. To sample from the exact texel center, we must either add half a texel width and height to the offset or move the quad by half a pixel in each axis. Listing 5.8.1 chooses to add half a texel width and height. It is essential to understand these half-texel offsets when using bilinear filtering. Without this, the bilinear sample will grab potentially undesired neighbors and produce very different results. It is essential to test and know exactly where texture samples are placed for procedural texture algorithms to work as expected. Conway's "Game of Life" described later in this gem makes a good test case.

The four resulting texture samples can be combined in the programmable pixel engine for various effects. By using the resulting image as a new source texture and applying the neighbor sampling again and again to create subsequent frames, a wide variety of interesting texture animations is possible. If the four samples are averaged, the result is a blurring of the source image. By introducing an additional *(s,t)* scrolling amount to the offsets presented previously, we can blur and scroll a source image over successive frames. As we blur and scroll, we can jitter the scrolling vector used in each frame and multiply each sample color by various RGBA values to fade or alter the color. If we supply a steady input to the blur and upward scroll by first rendering



A.



B.

FIGURE 5.8.2 *Fire and smoke animation using Listing 5.8.1 s offiets A with a scroll offset. The bright "embers" at the bottom are rendered to the texture each frame before blurring and scrolling upward. B) shows the progression of rendering operations.*

bright source pixels or "embers" in the bottom of the texture for each frame, the result is the fire and smoke effect of Figure 5.8.2. Using just two 128x128 32-bit textures, this effect runs endlessly without repetition at over 500 frames per second on a modern graphics card. Because we cannot simultaneously render to a texture and use it as a source, we must use two textures and ping-pong back and forth between them. One is used as the previous frame source, and the other as the current frame destination.

Rather than averaging neighbor samples to blur, we can compute the differences between samples and raise this to a power for high contrast. Using Listing 5.8.1's offsets A and reducing the magnitude of the offsets to partially sample the center texel, we can perform edge detection in a hardware pixel shader program, as implemented by Matthias Wloka [Wloka2000] and shown in Figure 5.8.3. Another curious "frosted glass" effect is produced by blurring the source and differencing this blur from the original image. Figure 5.8.3d shows the result of (src DIFF (src DIFF(BLUR(src))), where DIFF is the absolute value of the difference in RGB color.

Neighbor sampling and differencing can also be used to implement the physics of height-field-based water in successive render-to-texture operations. An algorithm for height-field water with vertex geometry on the CPU is presented in *Game Programming Gems* [GomezOO]. We can develop this into a series of rendering operations that use textures to represent the water height, velocity, and force. Each texel takes the place of a vertex in Gomez's implementation. Instead of sampling neighboring vertices on the CPU, we sample neighboring texels on the graphics processor. Our implementation involves six textures. Two textures represent the height of the water as

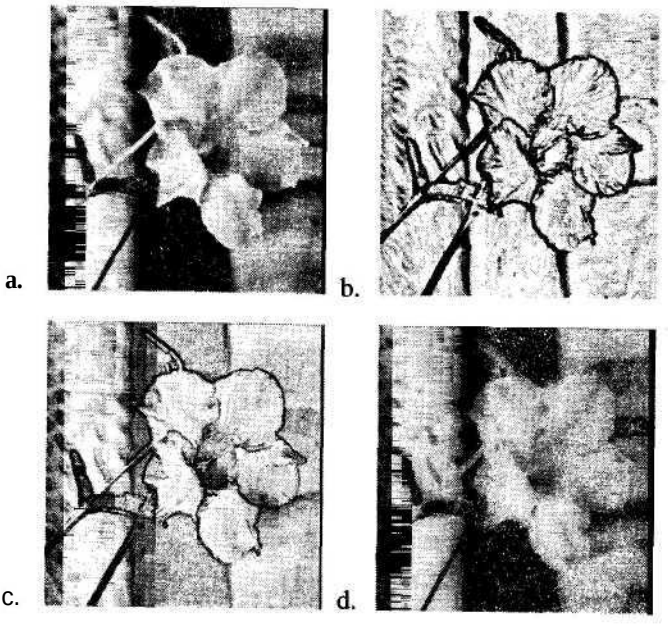


FIGURE 5.8.3 *Edge detection and image processing. A) is the original image. B) is the result of an edge detection in programmable pixel hardware. C) shows a 50-percent blend of a and b. D) is the original minus the difference between the original and a blur of the original.*

grayscale color values (height maps), with one texture for the heights at the current time step and one for the heights at the previous time step. Similarly, two textures represent the velocity of the water, and two are used to accumulate the nearest-neighbor forces acting on each texel. Figure 5.8.4 shows four frames in the progression of an animation using this technique, and despite the use of six 256x256 32-bit textures and four rendering passes per time step, the animation maintains a rate of over 450

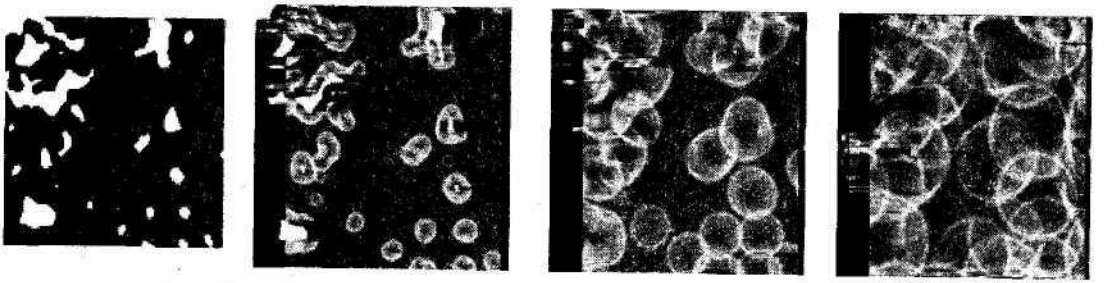


FIGURE 5.8.4 *Initial condition and three frames of height-based water animation in the pixel processing of a 3D accelerator. Six textures are used in generating subsequent time steps, although only the output height texture is shown here.*

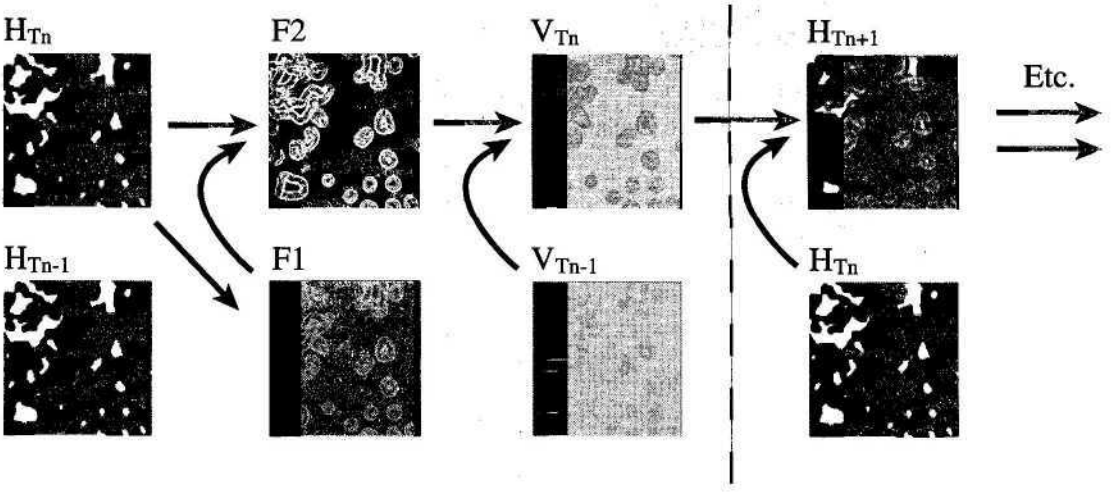


FIGURE 5.8.5 *Six state textures used for height-based water animation. H_m is height for the most recent time step. Fl and F2 are used to accumulate the force that will act on each texel height. The resulting F2 is applied to the previous time step's velocity $V^{\wedge}$,;, and the resulting velocity V_m is applied to the height to create the height field at the next time step $//_{w+}/$. Scale factors for "mass" and "time" that are multiplied into the texel values are not shown.*

time steps per second. Figure 5.8.5 shows the progression of textures used in generating one time step.

The hardware used for this example operates internally on 9-bit per component signed color values, and this gives rise to inaccuracy. Low bit values are easily lost, producing rounding errors in the physics that cause the system to decay to zero or grow to saturation. The nearest-neighbor sampling can also produce high-frequency oscillation between texel values that appears as noise. These problems are lessened by damping the system with blurring and adding a restoring force that pulls all heights gently to a midrange value. The blurring is accomplished by using bilinear filtering with neighbor offsets slightly greater than one texel width or height. The system can be driven by occasionally rendering seed droplets to the texture at random locations. By experimenting with the scale factors at which blurring, force, and velocity are applied, a stable endless animation is not difficult to achieve.

Animated grayscale height maps are useful, but we can go a step farther and use nearest-neighbor sampling to create an RGB normal map from the grayscale map. This normal map can then be used for realistic dot product bump mapping [Moller99]. The normal map can be created in a single render-to-texture pass on the graphics processor, and this avoids costly CPU work, texture memory copies, and graphics pipeline stalls. Creating normal maps in hardware this way is an elegant means of updating and animating the surface detail they represent. Grayscale height features such as bullet holes, cracks, and so forth can be rendered into the height map

as the game is running, and this is then converted to a normal map on the fly. The conversion is comparable in speed to the fire and smoke effect discussed earlier. Listing 5.8.2 shows a pixel program for creating an RGB normal map from a grayscale height map in a single pass on four-sample multitexture hardware. It uses an approximation in normalizing the resulting RGB texels, and the result works very well for lighting and reflection calculations. Two passes and a dependent texture read operation could be used to produce exactly normalized RGB vectors.

Use texel offsets A from Listing 1.

Listing 5.8.2 Code for RGB normal map creation in hardware from a grayscale height texture. The grayscale image is selected into all four texture stages, and offsets A from Listing 5.8.1 are used.

```
Pixel Program;
    // Creates an RGB normal map from an input grayscale height map
    // Pairing of RGB and Alpha instructions is not used
    //    Normal map parameterization is [0,1] so 0.5 = zero component
    //   along that axis (value of 127 stored in the texture).
    //     Red   = positive S axis
    //     Green = positive T axis
    //     Blue  = positive R axis (up out of page)

    // Declare pixel shader version
    ps.1.1


    def c5, 1.0, 0.0, 0.0, 1.0    // red mask for s axis component
    def c6, 0.0, 1.0, 0.0, 1.0    // green mask for t axis component
    def c4, 0.0, 0.0, 1.0, 1.0    // blue mask for r axis component
                                  // (blue = up out of texture)
    def c2, 0.5, 0.5, 0.0, 0.0    // 0.5 bias for red & green
    def c1, 1.0, 1.0, 0.0, 0.0    // color mask for red & green


    ; get colors from all 4 texture stages
    ; to = -s,  0
    ; t1 = +s,  0
    ; t2 =  0, +t
    ; t3 =  0, -t


    // Select source grayscale texture into all 4 texture stages
    // Sample all 4 texture stages
    tex to              // to = RGBA texel at coordinate offset by
                        //      (-s, 0)
    tex t1             // t1 = (+s, 0)
    tex t2             // t2 = ( 0,+t)
    tex t3             // t3 = ( 0,-t)
```

```
sub_x4 rO, to, t1      // rO = (tO-t1)*4 = s axis height slope
                       // Use _x4 to increase low contrast grayscale
                       // input
mul    to, rO, c5      // to = rO * red mask = red component only
                       // Use to as temp storage

sub_x4 M, t3, t2       // r-1 = (t3-t2)*4 = t axis height slope
mad    rO, r-1, c6, to     // rO = M.green + to  = s and t result in
red,green
mul    t1, rO, rO      // t1 = square s and t components
                       // Use t1 as temporary storage

dp3_d2 n, 1-tI, c1   // rI.rgb = (1 - s"2 + 1 - t^2 )/2
                       // (1-s"2) is approx sqrt(1-s*2) for small s
add rO, rO, c2         // rO = rO + 0.5 red +0.5 green
                       // shifts signed values to [0,1]

mad rO, M, c4, rO    // RGB = (r+0, g+0, 0+blue )
                       // output = rO
```

Dependent Texture Addressing

We have seen that a simple multisample operation can go a long way to produce various effects. There are other more sophisticated and powerful texture operations at our disposal. Dependent texture address operations enable us to fetch texture samples from one texture based on the colors of another texture, or to fetch based on the dot products of iterated texture coordinates and texture colors. This gem covers only one of the more simple dependent texture operations: the dependent green-blue texture addressing operation expressed as DXS's texreg2gb instruction, or the GL_DEPENDENT_GB_TEXTURE_2D_NV extension to OpenGL.

Dependent green-blue addressing is a straightforward process. For the pixel being rendered, the hardware fetches the source texture's color at that pixel. The green component of this color is used as the S (horizontal) coordinate of a fetch into another texture. The blue component is used as the T (vertical) coordinate. Figure 5.8.6 illustrates this with a 3x3 source texture.

Texture 2 can have any size relative to Texture 1; however, if Texture 1 holds 8-bit green and blue values, then any resolution of Texture 2 greater than 256 is pointless, as the fine resolution cannot be accessed by the coarse 8-bit values. Texture 2 provides an arbitrary lookup table for the Texture 1 input, and this can be used with render-to-texture operations to run some very sophisticated programs entirely within in the pixel hardware. We'll refer to Texture 1 as the "input" texture, and Texture 2 as the "rules" texture, because Texture 2 determines how an input color maps to a result.

With render-to-texture operations, we can use one or several texture maps to store intermediate logic results. We can also generate the input and rules maps on the fly based on previous results or user input. As you can imagine, there's quite a lot of flexibility here, and we haven't even considered dependent alpha-red lookups or the dot product address operations! Since the green-blue addressing operation relies on

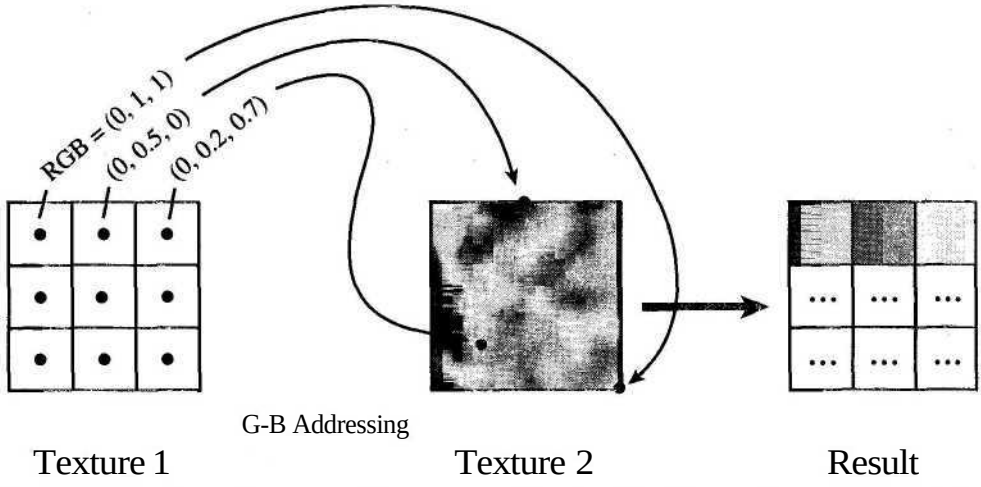


FIGURE 5.8.6 *Dependent green-blue texture addressing. The Texture 1 source is sampled at the points indicated. Each texel of Texture 1 determines the coordinate at which to sample from Texture 2, the "rules" texture. The resulting color from Texture 2 is output as the result.*

specific texture colors that are probably not the best colors for displaying in a scene, it is a good idea (and trivial) to add a second dependent lookup into a color table. This second operation maps the results of the procedural texture to any RGBA values for display in the scene. It creates a final output texture, so the user never sees the underlying green-blue textures driving the calculations. Our final example involves combining neighbor sampling and dependent lookups to run Conway's "Game of Life" on a graphics processor.

Conway's "Game of Life" in Hardware

John Conway's "Game of Life" [Gardner70] is a popular cellular automata program. Though it does not at first seem relevant or of much use to the average computer game, it is familiar territory in which to work, and the basic operations of running the game on a graphics chip are applicable to other procedural techniques. The game can produce interesting patterns of correlated noise that are useful in driving other effects; for example, the embers in the fire and smoke effect shown previously. Implementing this game on your own platform is a good way to verify that the hardware is doing exactly what you expect. The game has easily recognizable cyclical patterns and depends sensitively on proper texel sample placement, making it easy to determine when you have your sampling correct.

In Conway's "Game of Life," each cell begins as either "on" or "off," which we represent by a texture map of white or black texels, respectively. The rules for creating the next generation are, for every cell on the map:

1. If a cell is on and has two or three neighbors on, the cell remains on in the next generation.
2. If a cell is on and has fewer than two or greater than three neighbors on, the cell is turned off in the next generation.
3. If a cell is off and has three neighbors on, the cell is turned on in the next generation.

The game requires that we sample eight neighbors and the center cell, and apply logic to the result of the sampling. The OpenGL "Red Book" [Neider93] outlines a technique for running the game using a hardware stencil buffer, but we can implement it in a more flexible manner using fewer passes with hardware capable of dependent texture addressing.

Our approach is to create a render target color buffer of the same resolution as the field of cells. Beginning with black at each texel of this buffer, we render colors additively from the source cell field for each of the eight neighbor texels and the center texel. Each neighbor texel of the source is multiplied by 1/8 green and added to the destination, and the center texel is multiplied by full blue and added in. In practice, we use bilinear sampling and the B offsets of Listing 5.8.1 to sample two neighbors at once, and multiply the sample by 1/4 green. A second pass samples, the center texel multiplied by blue. The result is a color that encodes the condition of each source texel in relation to the rules of the game. Each cell's neighbor count ranges from 0 to

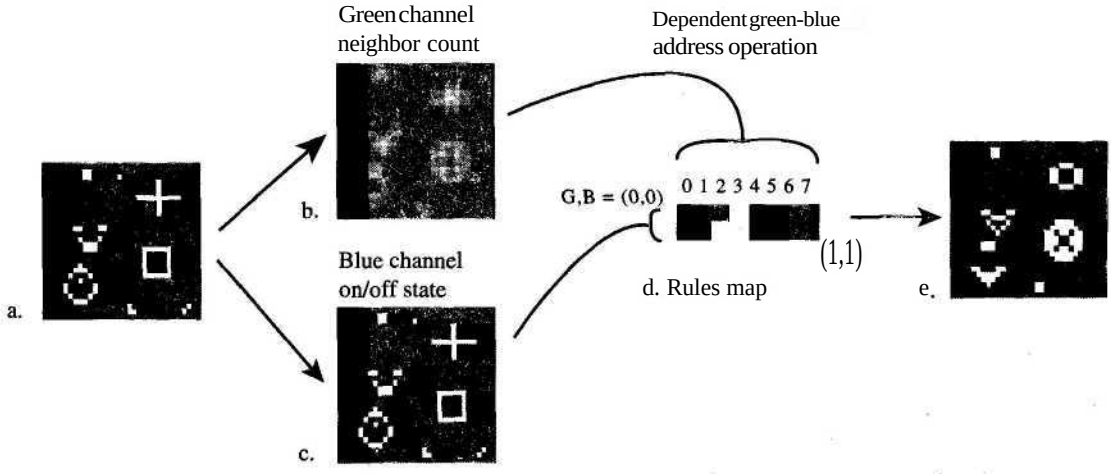


FIGURE 5.8.7 *Steps in the production of the next generation of cells in Conways "Game of Life. "A) is the initial cell field. B) is the green component of the condition texture that is each 'cell's number of active neighbors. C) is the blue component of the condition that reflects whether the cell is on or off. D) is an 8x2 pixel texture that encodes the rules of the game. E) is the resulting generation of cells that will be used again as the new input a.*

1.0 in green, and the cell's "on" or "off" state is 0 or 1 in blue. Each green-blue texel is then used as the input for a dependent green-blue address operation. The operation reads from an 8x2 pixel rules texture that determines the color (white or black) of the texel in the next generation. This rules texture is black at all texels except those at (2,1), (3,1), and (3,0), which are white. These white pixels are those that the rules of the game determine to be "on" in the next generation. Figure 5.8.7 shows an initial generation source texture, the green-blue condition rendered from it, the rules texture, and the subsequent generation produced.

This simulation depends on the source texels being white and the intermediate being green-blue. We could easily perform an additional dependent texture read from the source or intermediate into an arbitrary color ramp to create a separate texture for use in rendering the scene. Also, there is no need to limit the rules texture to 8x2 pixels. Any size texture could be loaded, and this texture could encode highly complex rules, or rules that spawn pixels of any RGB value. See the online examples and source code mentioned below for a glimpse of the possibilities.

Future Work

As graphics processors draw more texture samples per rendering pass, the 8-bit color components and internal pixel processing operations of today's hardware are squeezed for precision. It is all too easy to end up with color banding or other rendering artifacts. Future generations of hardware will almost certainly need higher precision formats, and with 16 or 32 bits per color component, there are great opportunities for hardware-accelerated procedural texture effects.

Problems of limited precision were mentioned earlier in the water simulation. 16-bit floating point values would virtually eliminate these. Higher precision floating-point values in textures and pixel calculations could enable fluid flow simulations and lattice grid-based Navier-Stokes algorithms to run in graphics hardware [Witting99]. CPU implementations of these algorithms are already close to real time in special effects houses and graphics research. Increased precision of render target buffers will also enable hardware-accelerated advanced lighting calculations that involve simple Fresnel integrals, wave propagation, and diffraction effects [Feynman85][Stam99].

Cellular automata programs give rise to a wealth of interesting animations and rjatterns with various degrees of correlation. Cellular automata programs do have the disadvantage of being difficult to control and engineer to a specific pattern or effect, but collections of discovered programs and convenient browsing programs exist on the Internet. Work on lattice gas automata (generally 2D surface simulations) may also provide useful animated patterns.

The future of real-time procedural texturing and animation is filled with promise. There is a vast landscape of techniques and effects to explore, and these certainly have the power to bring added realism and variety to interactive 3D games.

Acknowledgments

Special thanks to Matthias Wloka for his work on multisampling and hardware image convolution, and to my co-workers at Nvidia for inspiring and enabling such amazing graphics hardware.

Code Samples

DirectX 8 code for the preceding examples is posted on Nvidia's public Developer Web site (www.nvidia.com/Developer/DX8) for you to download and experiment with. The samples can be run in the DX8 SDK's reference rasterizer or on hardware that supports DX8 Vertex Shaders v1.1 and Pixel Shaders v1.1.

References

[Ebert98] Ebert, David S., et al, *Texturing and Modeling: A Procedural Approach,* Academic Press, 1998 [ISBN 0-12-228739-4].

[DXS'OO] Microsoft Corporation, *DirectXS* SDK, available online at http://msdn .microsoft.com/directx/, November, 2000.

[NVExt2001] Nvidia Corporation, "Nvidia OpenGL Extensions Specifications" (nvOpenGLSpecs.pdf). available online at www.nvidia.com/opengl/OpenGLSpecs, March 2001.

[Wloka2000] Wloka, Matthias, "Filter Blitting," available online at www.nvidia.com\ Developer/DX8, November 2000.

[Gomez2000] Gomez, Miguel, "Interactive Simulation of Water Surfaces," *Game Programming Gems,* Charles River Media Inc., 2000: pp. 187-194.

[Moller99] Moller, Tomas, and Haines, Eric, *Real-Time Rendering,* A K Peters, Ltd., 1999.

[Gardner70] Gardner, Martin, "Mathematical Games," *Scientific American,* vol. 223, no. 4, October 1970, pp. 120-123.

[Neider93] Neider, Jackie, et al, *OpenGL Programming Guide,* Addison-Wesley Publishing Co., 1993: pp. 407^409.

[Witting99] Witting, Patrick, "Computational Fluid Dynamics in a Traditional Animation Environment," Computer Graphics Proceedings (SIGGRAPH 1999) pp. 129-136.

[Feynman85] Feynamn, Richard P., *QED: The Strange Theory ofLight and Matter,* Princeton University Press, 1985, pp. 37-76. Although he does not label it a "Fresnel integral," this is the name for the calculation he explains in Chapter 2. This powerful mathematics accounts for all phenomena in the propagation of light by a large sum of a few simple terms.

[Stam99] Stam, Joe, "Diffraction Shaders," Computer Graphics Proceedings (SIGGRAPH 1999) pp. 101-110.

Gomez, Miguel, "Implicit Euler Integration for Numerical Stability," *Game Programming Gems,* Charles River Media Inc., 2000: pp. 117-181.