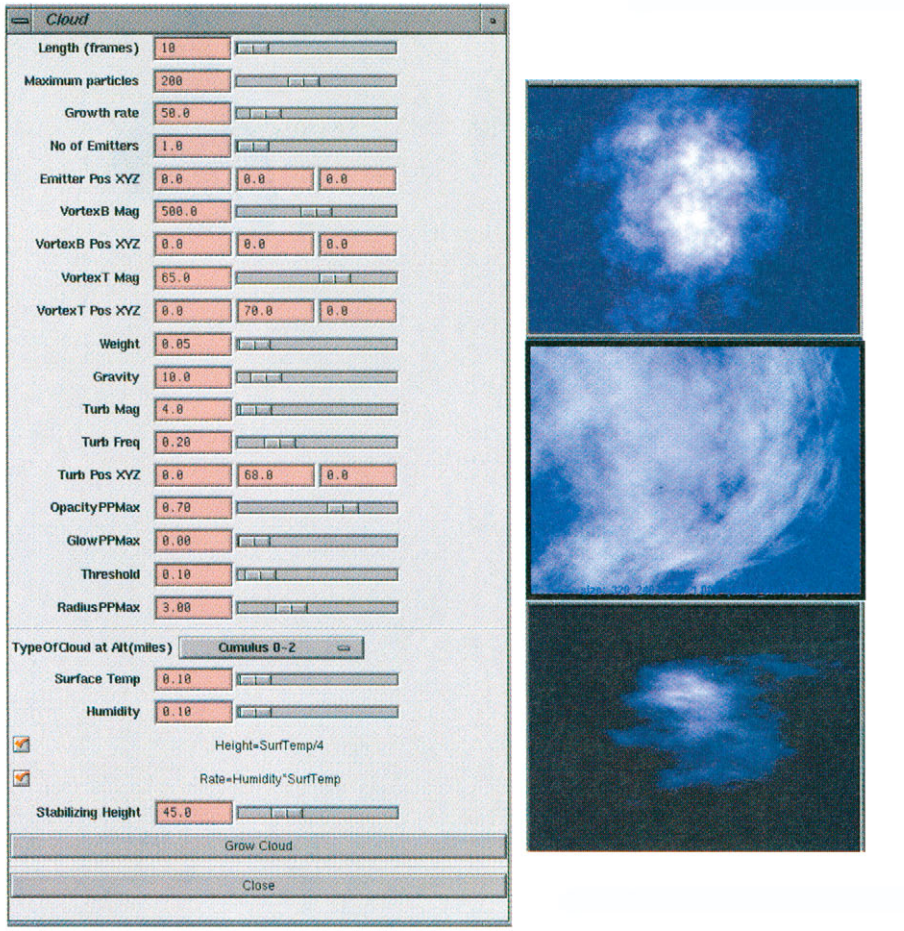


FIGURE 9.11 MEL GUI and sample images from Maya-based cloud particle system generator.

FIGURE 9.12. An example cloud rendering from a Maya plug-in. Image created by Marlin Rowley.

ISSUES AND STRATEGIES FOR HARDWARE ACCELERATION OF PROCEDURAL TECHNIQUES

DAVID S. EBERT

With contributions from Joe M. Kniss and Nikolai Svakhine

INTRODUCTION

As mentioned in Chapter 1, graphics hardware has reached a point in terms of speed and programmability that we can now start utilizing it for procedural texturing and modeling. This chapter will discuss general issues relating to hardware acceleration of procedural techniques, some common approaches to utilizing graphics hardware, and several examples showing how we can utilize the current generation of graphics hardware (as of 2002). With the rapid change in graphics technology, we should see more programmability and power for procedural techniques every year. One thing to remember in designing your application is that procedural techniques will continue to be a great solution to solve the data transfer bottleneck within your computer for years to come.

To clarify the presentation in the rest of the chapter, it is useful to review some basic terminology, which was introduced in Chapter 3. I'll refer to the graphics processing unit on the graphics board as the GPU. In most 2002-generation GPUs, there are two basic levels of processing: vertex processing and fragment processing. The vertex processing performs operations for each vertex (e.g., matrix and space transformations, lighting, etc.), and the fragment processing refers to processing that is performed per pixel-sized geometric primitive fragment. You are encouraged to read this chapter in conjunction with, or after, the discussion of real-time programmable shading by Bill Mark in Chapter 3. These chapters cover several similar issues, but each has a different perspective and different emphasis.

GENERAL ISSUES

In general, when migrating a procedural technique from CPU rendering to GPU rendering, there are several factors to consider, including the following:

- Language interface
- Precision
- Flexibility and capabilities
- Storage
- Levels of operation

The language interface is a basic issue that must be confronted, since most current standards don't provide all of the features of the graphics hardware at a high-level language (C, C++) interface. Several hardware manufacturers are currently developing better tools to simplify the task of programming the graphics hardware, but, currently, the only high-level language interface available is the Stanford real-time shading language (RTSL). This is a very basic issue that will, hopefully, be solved in the near term and adopted into the next generation of graphics standards (e.g., proposals for OpenGL 2.0).

The issue of computational precision will probably still exist for several years, and it must be considered when designing and implementing algorithms that will run on a variety of graphics platforms and game consoles. Therefore, when designing an algorithm, the precision at each stage in the pipeline must be considered. For instance, in many 2002-generation graphics cards and game consoles, vertex processing has higher precision than fragment processing, while the final texture combiners in the fragment-processing hardware have higher accuracy than earlier operations. These precision issues are most important for approximating detailed functions, when performing a large number of computations and iterative computations (e.g., multipass computations). Accumulation of error may have a significant effect on the results. Another issue that needs to be considered is the precision of the operands. For instance, if you are limited to 8-bit operands because the operands are coming from a texture map access, having 16-bit precision for an operation is not much of an advantage. Again, these precision issues have a cumulative effect and increase as the complexity of the algorithm increases. Finally, precision is also a factor in hardware interpolation and must be considered when using hardware blending and texture mapping interpolation to perform some computations.

Flexibility is a longer-term concern for hardware-accelerated procedural techniques. Most graphics hardware has limits on the operations that can be performed at each stage, and it may be necessary to reorder the operations in your software procedural technique to implement it using graphics hardware. For instance, with 2002-generation GPUs, you cannot create geometry on the fly within the GPU for

performance reasons, and vertex programs cannot access texture maps. The available operations are also a determining factor in designing GPU-accelerated procedural techniques. What mathematical functions are required? Are these available at the desired stage in the hardware pipeline? If not, can they be approximated using texture maps, tables, or registers? How many texture maps can be accessed in a single hardware pass? What is the performance trade-off between two-dimensional texture map accesses and three-dimensional texture map accesses?

The recent trend in the flexibility of graphics hardware operations is that, in general, vertex processing offers more program flexibility than fragment processing. The GPU program flexibility has a significant impact on the ease and effectiveness of real-time procedural texturing and modeling. As this flexibility increases, it will be possible to implement more complex and elegant procedural models and textures entirely within the GPU. Hopefully, within the next several years, we will be able to simply download an algorithm to the GPU and completely generate the model, as well as the textures, within the GPU.

As with hardware flexibility, available storage can significantly impact the design of procedural techniques. When decomposing an algorithm into vertex and fragment components, is there enough storage at each stage? The type of storage and the access penalty of the storage are important to consider. Having the ability to write to a texture map provides a large amount of storage for the algorithm; however, this will be much slower than writing and reading from a small set of registers. Access to texture map storage also allows information that doesn't change per frame to be precomputed and then accessed from the texture map, increasing the performance of the algorithm.

As can be seen from the above discussion, the various levels of operation in the graphics pipeline, in most cases, must be considered to develop efficient real-time procedural techniques. Some high-level language compilers (e.g., RTSL) can hide some of the architecture specifics from the user, but the understanding of at least the separation of vertex and fragment processing must be considered to develop effective interactive procedural techniques.

COMMON ACCELERATION TECHNIQUES

This section describes a number of common approaches to accelerate procedural techniques using graphics hardware. Many of these techniques are straightforward extensions of common software acceleration techniques and common hardware rendering acceleration techniques. Precomputation is probably the most common acceleration technique in graphics and can be used to some extent with procedural

techniques. However, the trade-off of performance from precomputation with the flexibility and detail provided by on-the-fly computation must be considered. Precomputing a marble texture and storing it in a 2D texture map would not be considered a procedural texture since the detail on demand and other benefits of procedural techniques would be lost. However, precomputing functional approximations (e.g., sine, cosine, limited power functions) for functions not available in the GPU program is commonly used to allow hardware implementation of procedural techniques.

Another common technique is the use of dummy geometry and depth culling. Most graphics hardware does not allow a vertex or fragment program to create new geometry. Therefore, in order to create a hardware procedural model, the common trick is to create a large list of dummy polygons that are initially created with a depth value that allows fast hardware culling of the geometry to reduce computations for unused dummy geometry. The algorithm that is creating "new" geometry simply transforms the geometry from the dummy list to make it visible and part of the procedural model. This can easily be done for a particle system that runs completely on the graphics hardware and could also be used to create L-system or grammar-based plant models that grow and evolve based on algorithms running on the graphics hardware.

A good algorithm designer can also harness the power of the components of the graphics hardware to perform tasks for which they weren't originally designed. Most GPUs have hardware that performs interpolation, dot products, blending, and filtering. By storing values appropriately into textures, registers, operands of different combiner stages, and so on, you can use the hardware to perform these operations as they are needed in your procedure. For instance, Pallister (2002) has used multitexturing, texture blending, and built-in hardware filtering and interpolation for quickly computing clouds for sky domes. The hardware dot product capability can also be used for space transformation to compute warps and procedures that use different texture/model spaces.

The use of multilevel procedural models can be easily adapted to hardware to achieve higher-resolution models at interactive rates. The most flexible way to implement this approach takes advantage of the dependent texture read facility in many GPUs. This feature is vital for many procedural techniques and allows texture coordinates for accessing a texture to be computed based, at least partially, on the results of a previous texture lookup result. To create the multilevel model, a coarse model is created from a low-resolution texture map, and a separate detail texture map is combined to produce higher-frequency details, as illustrated with the interactive procedural cloud example below. Detail and noise textures can also create more natural

and artistic effects by using them to adjust the texture coordinates for accessing another texture map. These texture coordinates can also be animated over time by varying the amount of the detail/noise texture that is used in generating the new texture lookup.

Animation of procedural techniques in real time is a challenge. Updating large texture maps per frame is not currently feasible because of memory bandwidth limits. However, animating texture coordinates and coefficients of procedures can be performed in real time. If this approach is used in the design of the procedural model/texture, it is very easy to create animated real-time effects.

EXAMPLE ACCELERATED/REAL-TIME PROCEDURAL TEXTURES AND MODELS

Several examples of real-time procedural techniques are presented below to illustrate how to take the procedural techniques described throughout this book and adapt them to interactive graphics hardware.

Noise and Turbulence

One of the most common procedural functions used is the noise function, so this is a natural place to start describing how to implement procedural techniques in graphics hardware. Several different approaches have been taken to implement hardware noise and turbulence. Hart (2001) has used the programmability of the graphics hardware to actually calculate noise and turbulence functions by making many passes through the graphics hardware. The more common approach is to use 2D or 3D texture mapping hardware to generate a lattice-based noise function. With a 2D texture map, the third texture coordinate is used as a scaling factor for accessing different components of the 2D texture map. A 3D texture map implementation is very straightforward for implementation: simply compute a 3D texture map of random numbers. If you then set the texture mapping mode to repeat the texture throughout space, the texture mapping hardware will do the interpolation within the lattice for free.

The main difficulty that arises is in implementing turbulence and more complex noise functions (e.g., gradient noise). A turbulence function can easily be implemented by making several texture lookups into the 3D noise texture, scaled and summed appropriately. However, there may be a penalty with this method, since it requires several texture accesses (e.g., four or five) for the turbulence functions alone. Depending on your hardware, this may significantly limit the number of

texture accesses available in a single pass. The good news is that all of the texture accesses are to the same texture, and this does not introduce the overhead of multiple 3D texture maps. Of course, multiple levels of turbulence values can be precomputed and the result stored in the three-dimensional texture, but the replication of the turbulence function throughout space without seams becomes trickier. Also, this introduces smooth interpolation of the turbulence values within the lattice cells and decreases the fine detail of the turbulence function that can be produced for the same-size three-dimensional table. For instance, with a 64^3 noise table and four octaves of noise values summed (four texture accesses), the resulting resolution of the turbulence function is much greater than using a 64^3 turbulence texture map.

Marble

A simple, classic procedural texture is the following marble function:

```
marble(pnt) = color_spline( sin(pnt.x + turbulence(pnt)) +1)*.5, white, blue)
```

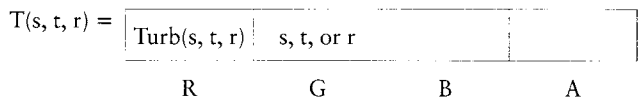
With flexible dependent texture reads, this can be implemented in hardware using the following algorithm:

```
R1 = texture3D(turbulence, pnt) using 3D texture access (1 to n accesses)
R2 = sin_texture(pnt.x+R1) => offset dependent read
     1D texture contains (sin(value)+1)*.5
R3 = color_spline_texture(R2) => value dependent read
```

This implementation requires at least three texture accesses with two dependent texture reads. Additionally, the register combiners need to support the offset dependent read and replace dependent read operations.

An example implementation of this in NVIDIA's Cg language (version 1.01), with restrictions in vertex and fragment programmability based on the capabilities in the Nvidia GeForce3 graphics processor, follows. This illustrates how these techniques can be applied to programmable graphics hardware and also the importance of the flexibility of operations at the different levels of the graphics pipeline: restrictions in the dependent texture read operations make this example more complex than the previous example. Therefore, the marble formula is simplified and several values are precomputed into textures. This procedure allows the real-time change of the period of the sine function and the amount of turbulence/noise added. To make this example more robust and eliminate texture seams, the single 3D turbulence texture call should be replaced with several scaled 3D noise texture calls to implement the turbulence function in the graphics hardware. The procedure has a vertex program and a fragment program component that use two textures: T(s, t, r) and sine01(s, t).

The noise and coordinate texture, T(s, t, r), has elements of the following form:



where

R-component – Turb(s, t, r) – turbulence at point (s, t, r).
G-component – s, t, or r, depending on desired marble orientation.
B, A – unused

The sine01(s, t) texture contains the values of the following function:

$$\text{sine01}(s, t) = (\sin((s+t)*2\text{pi})+1)*0.5$$

The color of the marble texture is computed to produce a blue-white marble using the following procedure:

```
VOID Sine(D3DXVECTOR4* pOut, D3DXVECTOR2* pTexCoord, D3DXVECTOR2* pTexelSize,
LPVOID pData)
{
        double sine;
        double R_, G_, B_;
        double t;
        sine = (sin((pTexCoord->x+pTexCoord->y)*2*MY_PI)+1)*0.5;

        if (sine < 0.2)
        {
          R_ = G_ = 255;
          B_ = 255;
        }
        else if (sine >= 0.2 && sine < 0.45)
        {
          t = (sine - 0.2)/0.25;
          B_ = 255;
          R_ = (1-t)*255;
          G_ = (1-t)*255;
        }
        else if (sine >= 0.45 && sine < 0.6)
        {
          R_ = G_ = 0;
          B_ = 255;
        }
        else if (sine >= 0.6 && sine < 0.8)
        {
          t = (sine - 0.6)/0.2;
```

```
            R_ = (t)*255;
            G_ = (t)*255;
            B_ = 255;
          }
          else if (sine >= 0.8)
          {
            R_ = G_ = 255;
            B_ = 255;
          }

  pOut->x = R_/255.0;
  pOut->y = G_/255.0;
  pOut->z = B_/255.0;
  pOut->w = 1.0;

return;

}
```

The following Cg vertex and marble shaders are based on NVIDIA's Direct3D implementation of DetailNormalMaps() that is part of the Cg Demo Suite.

Marble_Vertex.cg

```
struct a2v : application2vertex {
                                    float4 position;
                                    float3 normal;
                                    float2 texture;
                                    float3 S;
                                    float3 T;
                                    float3 SxT;
                                };

struct v2f : vertex2fragment {
                                    float4 HPOS;
                                    float4 COL0; //lightvector in tangent space
                                    float4 COL1; //normal in tangent space
                                    float4 TEX0;
                                    float4 TEX1;
                                    float4 TEX2;
                                };

v2f main(a2v I,
                                    uniform float4x4 obj2proj,
                                    uniform float3 lightVector, //in object space
                                    uniform float3 eyePosition, //in object space
                                    uniform float2 AB,
                                    uniform float4 offset,
```

```
                                    uniform float4 tile
                                    )
{
        v2f O;

/* Transform Position coordinates */

        // transform position to projection space

        float4 outPosition = mul(obj2proj, I.position);

        O.HPOS = outPosition;

/* Compute lightVector used for lighting */
        // compute the 3 x 3 transform from tangent space to object space
        float3x3 obj2tangent;
        obj2tangent[0] = I.S;
        obj2tangent[1] = I.T;
        obj2tangent[2] = I.SxT;

              // transform light vector from object space to tangent space and
              // pass it as a color
        O.COL0.xyz = 0.5 * mul(obj2tangent, lightVector) + 0.5.xxx;

              //Normal is (0, 0, 1) in tangent space
        O.COL1.xyz = float3(0, 0, 1);

/* Setting up noise and marble parameters */

        float A = AB.x; //A and B coefficients of marble formula, passed as
                        //uniform parameters to a shader
        float B = AB.y;

              //3D texture coordinate is just scaled and biased point coordinate
        O.TEX0 = tile.x*0.2*(I.position+offset);
        O.TEX0.w = 1.0f;

        //Those are the coordinates used for Sine calc in marble,
        //TEX1 and TEX2 are used as coordinates for dependent texture lookup
        //in pixel shader (Marble_Pixel.cg)

        O.TEX1.xyz = float3(A*B, 0, 0);
        O.TEX2.xyz = float3(0, A, 0);
      return O;
}
```

Marble_Pixel.cg

```
struct v2f : vertex2fragment {
              float4 HPOS;
              float4 COL0; //light vector in tangent space
              float4 COL1; //normal vector in tangent space
              float4 TEX0;
              float4 TEX1;
              float4 TEX2;
};

//fragout is standard connector fragment2framebuffer, outputting float4 color.

fragout main(v2f I,
             uniform sampler3D noiseTexture,
             uniform sampler2D sineTexture,
              )
{
             fragout O;
             //Texture coordinate interpolants, TEX0, TEX1, and TEX2 are set in
             //the vertex program (Marble_Vertex.cg)

             float4 noiseVal = tex3D(noiseTexture); //TEX0 is used here, getting
                                                    //turbulence/x value

             float4 marbleValue = tex2D_dp3x2(sineTexture, I.TEX1, noiseVal);

                  //TEX1 and TEX2 are used here.
                  //The coordinates for the sine lookup are
                  //S = A*B*Turb(x);
                  //T = A*P.x;
                  //So the result of lookup is sin(A*(P.x+B*Turb(x)).

             //Lighting part calculates parameter diffIntensity
             float ambient = 0.2f;
             float4 sNormal = (I.COL1);
             float4 lightVector = expand(I.COL0); //Calculated in vertex program
             float diffIntensity = uclamp(dot3(sNormal.xyz, lightVector.xyz))
               +ambient;

             //Final result
             O.col = marbleValue*diffIntensity;
             return O;
}
```

Results from this marble function, with varying periods of the sine function, can be seen in Figure 10.1.

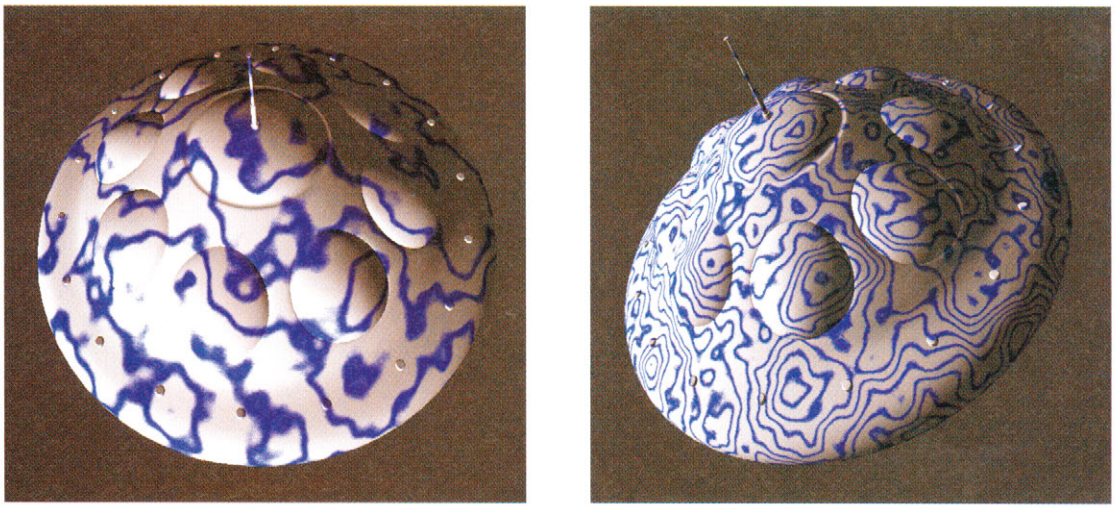


FIGURE 10.1 Example real-time marble textures created with varying periods of the sine function. Images created by Nikolai Svakhine.

Smoke and Fog

Three-dimensional space-filling smoke and fog can easily be created with 3D texture mapping hardware. The common approach is to implement volume rendering using a series of slicing planes that "slice" the volume and texture space. Evaluating shaders and 3D textures at each fragment in the slice plane samples the volume and performs the volume rendering. The shader should return the color and opacity of the given region in space. These slice plane polygons are then composited in a front-to-back or back-to-front order in the rendering process. For procedural smoke and fog, the actual opacity and color is determined by shaping a turbulence function generated from a 3D noise texture, as described in Chapter 7. A simple smoke function for determining opacity is the following:

```
Smoke(pnt) = (turbulence(pnt)*scalar)^power
```

This can be implemented with three octaves of noise using the following procedure for each volume slice fragment:

```
  R1 = texture3D(noise, pnt)—1st octave of noise using 3D texture
pnt2 = pnt * {2.0,2.0,2.0,1}
```

```
  R2 = texture3D(noise, pnt2)—2nd octave of noise using 3D texture
  R2*= R1
pnt2 = pnt2 * {2.0,2.0,2.0,1}
  R3 = texture3D(noise, pnt2)—3rd octave of noise using 3D texture
  R3*= R2
  R4 = scalar—for general density scaling
```

or

```
  R4 = texture3D (scalar, pnt)—use a varying scalar texture for more
                               variation
  R5 = texture1D (power_texture, R3*R4)—only need 1D texture if power is
                                        constant
```

or

```
  R5 = texture2D(power_texture,R3*R4, power)—for multifractal effect where
                                       the power is varied throughout space
```

This requires three 3D texture accesses plus a dependent texture read. Depending on your hardware capabilities, the above general idea may need to be reorganized due to restrictions on the order of operations. Of course, a precomputed turbulence texture could be used, but a high-resolution 3D texture would be needed to get good results. To animate the function, a downward helical path is very effective in creating the impression of the smoke rising. Therefore, the point is first swirled downward before the calls to the noise texture. The swirling can be calculated using a simple time-dependent helical path (as described in Chapter 8). This helical path is generated by a cosine texture and the following formula, which adds another texture read and changes the noise texture read to a dependent texture offset read:

```
p2.x = r1*cos(theta)—theta varies with frame_number, r1 is one ellipse
       radii
p2.z = r2*cos(theta)—r2 is 2nd ellipse radii
p2.y = -const*frame_number
  R1 = texture3D(noise, pnt+p2)—using 3D dependent offset texture
...
```

Real-Time Clouds and Procedural Detail

High-frequency detail cannot be represented effectively using reasonably sized models that will fit in 3D texture memory on current-generation graphics boards. These high-frequency details are essential for capturing the characteristics of many volumetric objects such as clouds, smoke, trees, hair, and fur. Procedural noise simulation is a very powerful tool to use with small volumes to produce visually compelling

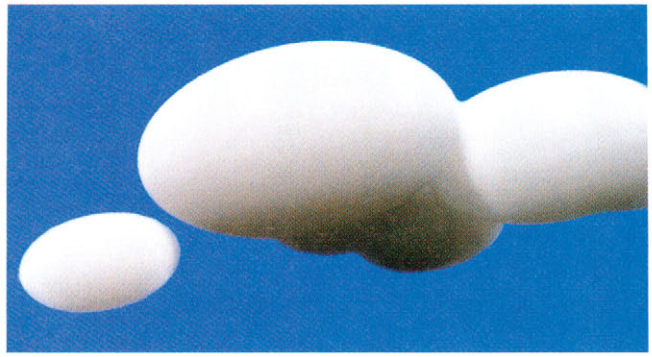
simulations of these types of volumetric objects. My approach for modeling these details is similar to my approach for modeling clouds, described in Chapter 9: use a coarse technique for modeling the macrostructure and use procedural noise-based simulations for the microstructure. These techniques can be adapted to interactive volume rendering through two volume perturbation approaches that are efficient on modern graphics hardware. The first approach is used to perturb optical properties in the shading stage, while the second approach is used to perturb the volume itself.

Both volume perturbation approaches employ a small 3D perturbation volume, 32^3. Each texel is initialized with four random 8-bit numbers, stored as RGBA components and blurred slightly to hide the artifacts caused by trilinear interpolation. Texel access is then set to repeat. Again, depending on the graphics hardware, an additional rendering pass may be needed for both approaches because of limitations imposed on the number of textures that can be simultaneously applied to a polygon and the number of sequential dependent texture reads permitted. The additional pass occurs before the shading volume-rendering pass.

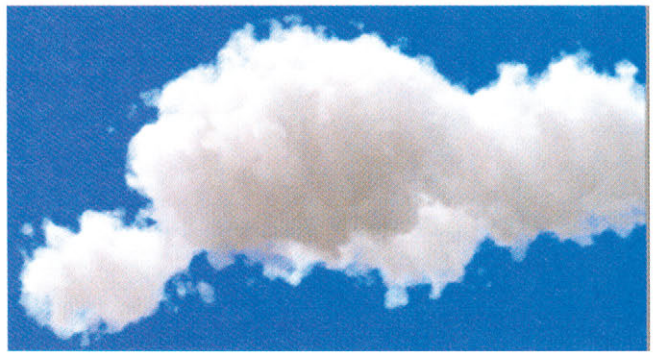
With a volume object, multiple copies of the 3D noise texture are applied to each volume slice at different scales. They are then weighted and summed per pixel. To animate the perturbation, we add a different offset to each noise texture's coordinates and update it each frame.

The first approach uses my lattice-based noise described in Chapter 7 to modify the optical properties of the volume-rendered object using four per-pixel noise components. This approach makes the materials appear to have inhomogeneities. The user may select which optical properties are modified, and this technique can produce the subtle iridescent effects seen in Figure 10.2(c).

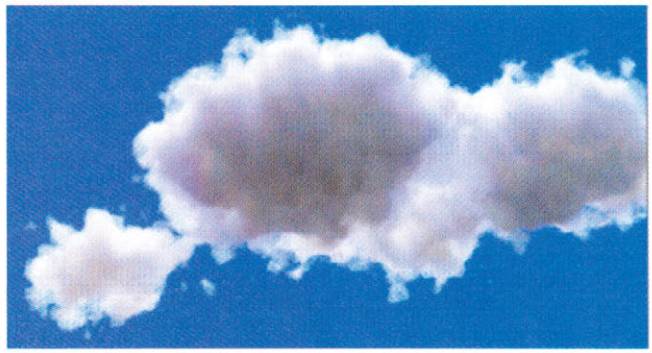
The second approach is closely related to Peachey's vector-based noise simulation described in Chapter 2. It uses the noise to modify the location of the data access for the volume. In this case, three components of the noise texture form a vector, which is added to the texture coordinates for the volume data per pixel. The data is then read using a dependent texture read. The perturbed data is rendered to a pixel buffer that is used instead of the original volume data. The perturbation texture is applied to the polygon twice, once to achieve low frequency with high-amplitude perturbations and again to achieve high frequency with low-amplitude perturbations. Allowing the texture to repeat creates the high-frequency content. Figure 10.2 shows how a coarse volume model can be combined with the volume perturbation technique to produce an extremely detailed interactively rendered cloud. The original 64^3 voxel data set is generated from a simple combination of volumetric blended implicit ellipses and defines the cloud macrostructure, as described in Chapter 9. The final rendered image in Figure 10.2(c), produced with the volume perturbation



(a)



(b)



(c)

FIGURE 10.2 Procedural clouds generated at interactive rates by Joe Kniss: (a) the underlying data (64^3); (b) the perturbed volume; (c) the perturbed volume lit from behind with low-frequency noise added to the indirect attenuation to achieve subtle iridescent effects.

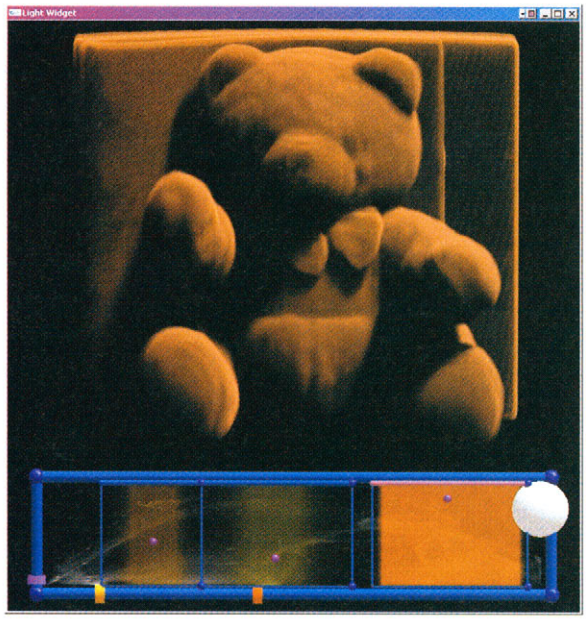
FIGURE 10.3 More complex cloud scene rendered at interactive rates using a two-level approach. Image created by Joe Kniss.

technique, shows detail that would be equivalent to an unperturbed voxel data set of at least one hundred times the resolution. Figure 10.3 contains another cloud image created with the same technique that shows more complex cloud structures with dramatic lighting. Figure 10.4 demonstrates this technique on another example. By perturbing the volume data set of a teddy bear with a high-frequency noise, a furlike surface on the teddy bear can be obtained.

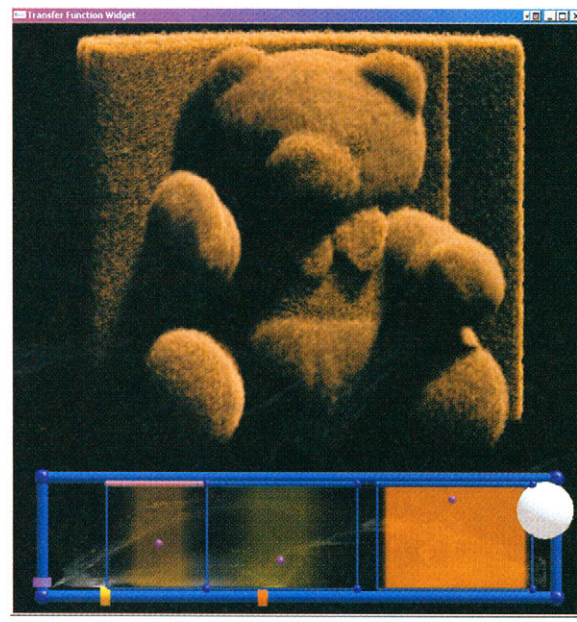
These volumetric procedural models of clouds and fur can be animated by updating the texture coordinates of the noise texture octaves each frame. Dynamically changing three-dimensional textures is too time consuming; however, varying perturbation amounts and animation of texture coordinates offers a fast, effective way to animate procedural techniques in real time.

CONCLUSION

This chapter discussed a number of important issues related to adapting procedural techniques to real-time graphics hardware. As graphics hardware and CPU hardware change, the trade-offs between computation in the CPU and on the GPU will vary. However, issues such as flexibility, storage, levels of computation, and language interfaces will persist. Designing effective real-time procedural techniques is a challenge that has started to become solvable. The examples in this chapter should give some insight into approaches that can be used to adapt procedural techniques to real-time rendering.



(a)



(b)

FIGURE 10.4 Volume renderings of a teddy bear volume data set by Joe Kniss: (a) the original volume; (b) procedural high-frequency fur added in the rendering process.