13

REAL-TIME PROCEDURAL SOLID TEXTURING

JOHN C. HART

As much of this book has demonstrated, procedural solid texturing is a powerful tool for production-quality image synthesis. Procedural solid textures can allow users to explore worlds flush with nonrepeating mountains, coastlines, and clouds. Dynamic animated textures like fire and explosions can be represented efficiently as procedural textures. Objects sculpted from a textured medium, like wood or stone, can be textured using solid texturing. Solid texturing is also easier than surface texturing because it does not require a surface parameterization.

However, much of the benefit of procedural solid texturing has been limited to the offline rendering of production-quality scenes. With the advent of programmable shading hardware in modern graphics accelerators, procedural solid texturing is becoming a useful tool for real-time graphics elements in video games and virtual environments. Procedural solid textures are compact and can be synthesized dynamically on demand. They can provide video games and virtual environments with a vast variety of textures that require little additional storage, at a resolution limited only by machine precision.

This chapter describes how to integrate procedural solid texturing into real-time systems using features already available in current graphics programming libraries. These techniques can also be used to create an interactive procedural solid texturing design system with real-time feedback.

A REAL-TIME PROCEDURAL SOLID TEXTURING ALGORITHM

The real-time procedural solid texturing algorithm is based on a technique from RenderMan that allows the texture map to hold the shading of a surface (Apodaca 1999). We assume each vertex in our model is assigned three kinds of coordinates. The *spatial coordinates* x, y, z of the vertex describe the location of the vertex in model space. The *parameterization* u, v of the vertex describes the location of the vertex in a 2D texture map. The *solid texture coordinates* s, t, r of the vertex describe from where in a 3D procedural texture space the vertex should get its color or other shading information. Given the spatial coordinates and the solid texture coordinates, we will construct a parameterization automatically. Often the solid texture coordinates are simply set equal to the spatial coordinates, so this algorithm can texture an object given only its spatial texture coordinates. The algorithm will run in three phases: rasterization, procedural evaluation, and texture mapping (see Figure 13.1). There is also a preprocessing step that we will call atlas construction that assigns the texture coordinates u, v to each vertex. This step will be described in the next section.

The rasterization phase plots the object's polygons in the texture map. The parameterization u, v serves as the coordinates of the vertices, and the solid texture coordinates serve as the color ($R = s$, $G = t$, $B = r$) of the vertices. Graphics hardware has long supported the linear interpolation of attributers across a polygonal face,[1] needed, for example, for smooth Gouraud shading. This linear interpolation automatically calculates the solid texture coordinates across the face of the polygon as it is rasterized in the texture map.

Rasterization

```
For each polygon $p$
        Begin polygon
        For each vertex $i$
                Color($p[i].str$)
                Vertex($p[i].uv$)
        End polygon
Save image as texture map $tex$
```

The procedural evaluation phase marches through all of the pixels in the texture map and evaluates the texturing procedure on the solid texture coordinates (s, t, r) stored as the pixel's RGB color. This evaluation will result in a new RGB color that represents the texture at the point (s, t, r) in the solid texture space. This color is stored at the current pixel in the texture map, overwriting the solid texture coordinate with its corresponding procedural texture color.

Procedural Evaluation

```
For each pixel $(x, y)$ in the texture map $tex$
        $tex[x, y]$ = proc($tex[x, y]$)
```

1. This interpolation is actually projective such that texture is interpolated "perspective correct."

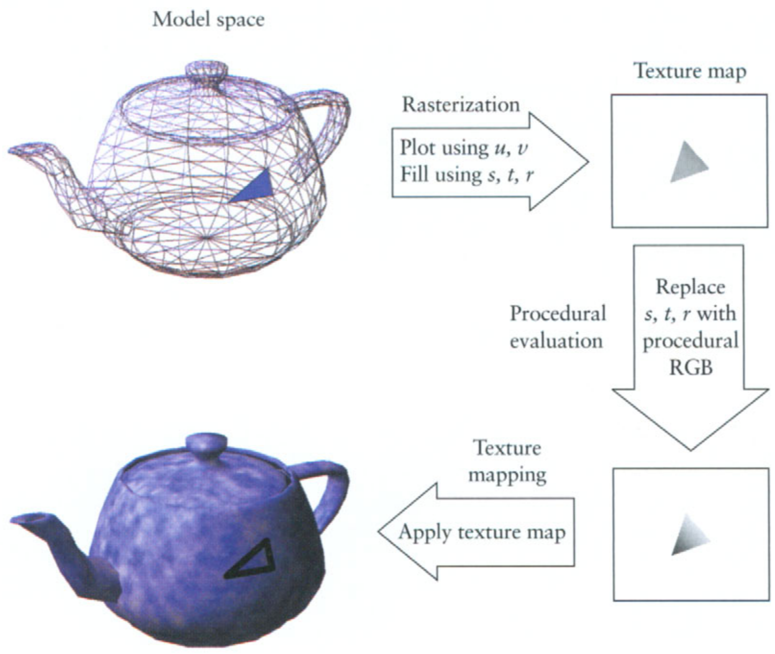

FIGURE 13.1 An algorithm for procedural solid texturing.

The texture mapping phase places the texture back on the object via standard texture mapping using the object's u, v parameterization. When the object is drawn, the spatial coordinates of its polygon vertices are passed through the graphics pipeline. The polygon is rasterized, which interpolates its u, v parameterization and textures the polygon using the corresponding pixel from the procedural solid texture stored in the texture map.

```
Texture Mapping
Set texture map to tex
For each polygon p
    Begin polygon
    For each vertex i
        TexCoord(p[i].uv)
        Vertex(p[i].xyz)
    End polygon
```

For this technique to work, the polygons on the object surface need to be laid out in the texture map without overlap. Such a texture mapping is called an *atlas*. The atlas construction step is performed as a preprocess to the previous algorithm and only needs to be computed once for an object. Techniques for performing this layout are described in the next section.

Models usually need one or more cuts in order to be laid flat into a texture map. These cuts can result in seams, which appear as discontinuities in the texture. Some texture layout techniques have focused on reducing the number and length of these seams. The section "Avoiding Seam Artifacts" shows how seams can be avoided for real-time procedural solid texturing, which allows the layout methods described in the next section to ignore seam length altogether and pack triangles individually into the texture atlas.

Both the rasterization and the texture mapping steps are hardware accelerated. The procedural evaluation step remains the bottleneck. Later in this chapter we will describe some techniques for efficient implementation of procedural textures, in particular those based on the Perlin *noise* function.

CREATING AN ATLAS FOR PROCEDURAL SOLID TEXTURING

A variety of techniques have been developed for creating texture atlases. Some of these techniques have been developed to automatically parameterize an object in order to place a 2D texture on its surface. These techniques try to minimize distortion, such that the proportions of the texture map image are reasonably reproduced on the textured surface. Because the real-time procedural solid texturing algorithm computed the texture from solid texture coordinates stored per pixel in the texture map, the distortion of the atlas does not affect the proportions of the procedural solid texture. The scaling component of the distortion, quantified in various forms elsewhere as the *stretch* (Sander et al. 2001) or the *relative scale* (Carr and Hart 2002), can affect the distribution of the samples across the object surface.

Because rasterization fills in the pixels in the texture map with solid texture coordinates, the location of triangles relative to each other in the texture map can be made irrelevant. Hence neighboring triangles in the object need not be neighbors in the texture map. This greatly simplifies the process of laying out the triangles in the texture atlas.

In order to use as many of the available texture pixels as possible, we lay out the triangles in a mesh across the entire texture map. This mesh consists of rows of isosceles axis-aligned right triangles that pack in a very straightforward manner, as shown in Figure 13.2.

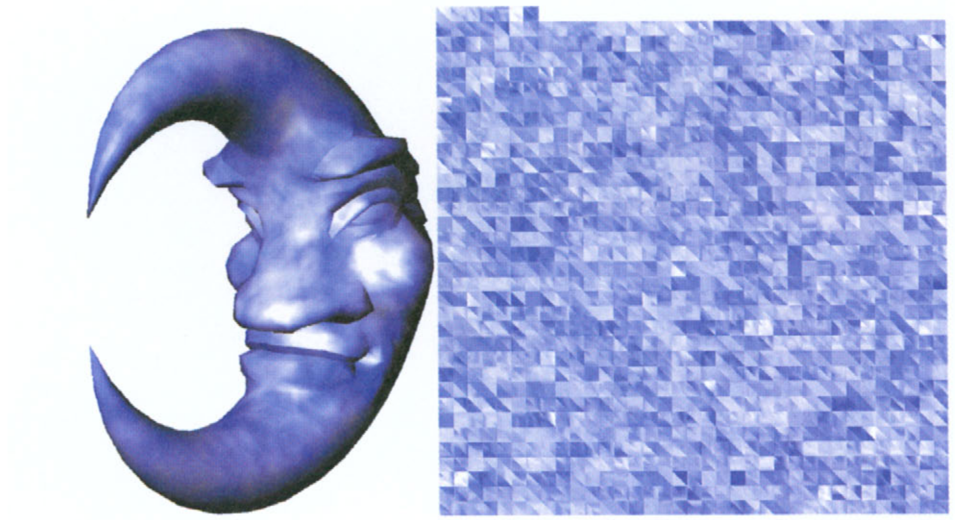

FIGURE 13.2 A uniform meshed atlas.

Laying out all object triangles into uniformly sized texture map triangles does not distribute texture samples well. Large object triangles should have more procedural solid texture samples than smaller object triangles, but the uniform mesh atlas assigns them the same number of samples. This can result in blocky texture artifacts on larger triangles and in general wastes texture space by giving smaller triangles too many texture samples.

We can also adjust the uniform mesh atlas to distribute the available texture samples more evenly across the object surface by varying the size of triangles per strip, as shown in Figure 13.3. We strip-pack the triangles into the texture map in order of nonincreasing area. We estimate a uniform scale factor as the ratio of the surface area to the texture area and use this scale factor to set the size of triangles in each horizontal strip. All of the texture map triangles in each horizontal strip are set to the same size to avoid wasting texture samples.

Other techniques have been developed to pack triangles of different sizes into a texture atlas. Maruya (1995) treated the triangles of a uniform mesh atlas as blocks and packed these blocks with triangles of a variety of sizes quantized to the nearest power of two. Rioux, Soucy, and Godin (1996) paired triangles of similar sizes into square blocks (quantized to the nearest power of two) and packed these blocks into the texture map. Battke, Stalling, and Aege (1996) rigidly strip-packed the triangles into the texture map, scaling them all uniformly so they would fit. Cignoni et al.

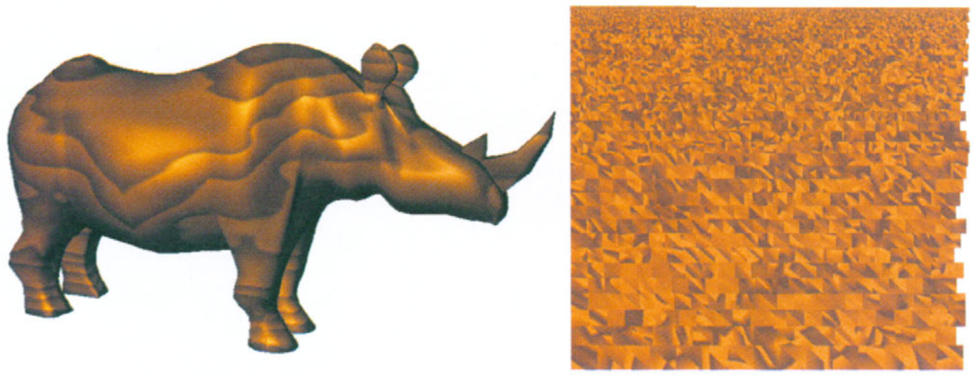

FIGURE 13.3 Rhino sculpted from wood and its area-weighted mesh atlas.

(1998) performed a similar strip packing, but sheared the triangles to better fit together.

Figure 13.3 shows an example of a shape shaded with a procedural solid texture using an area-weighted mesh atlas. The per-strip sizing of the triangles provides a more even distribution of texture samples across the surface than the other area-weighted layouts that quantize to the nearest power of two. Those techniques tend to more easily fill the texture map, whereas the area-weighted mesh has a blocky edge of wasted texture samples.

Skinny triangles can confuse the area-weighted atlas. A skinny triangle has two long edges that should receive more samples, but its surface area can be arbitrarily small. Meshes with a significant number of skinny triangles can still exhibit some texture blockiness using the area-weighted atlas.

For these cases, a length-weighted atlas is a better approach (Rioux, Soucy, and Godin 1996). Instead of using the triangle's surface area to set the size of its image in the texture map, the length-weighted approach uses the length of the triangle's longest edge. This technique ensures that the longest edges in the mesh get the most samples, but the technique tends to oversample skinny triangles, and this waste reduces the number of samples available to other areas of the model.

Recent atlases have also been designed to support MIP mapping. An atlas by Sander et al. (2001) clusters regions of triangles and packs these clusters into the atlas. The regions between clusters are filled with "reasonable" colors that allow it to be MIP mapped. An atlas by Carr and Hart (2002) uses a subset property of MIP mapping to pack the triangles into a hierarchy of proximate, although not necessarily neighboring, regions. Figure 13.4 shows an example of the proximate MIP map,

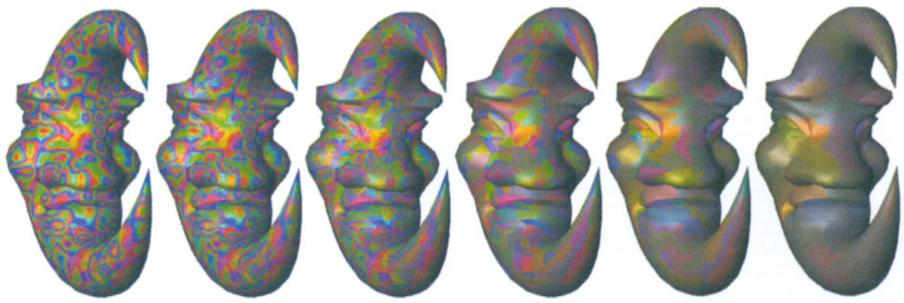

FIGURE 13.4 MIP mapping based on clusters of proximate triangles.

which maps proximate clusters of mesh triangles into the same quadrant at a given MIP map level.

AVOIDING SEAM ARTIFACTS

Texture filtering can cause some samples near the boundary of the polygon to be drawn from the wrong polygon image in the texture map. If the triangle is a neighbor, then this error is not at all serious. But since we are ignoring polygonal neighborhoods, we cannot depend on this situation.

Consider the example in Figure 13.5. In this example, the texture is a 4×4 pixel square. We will use integer coordinates for the pixel centers, with the lower-left pixel at (0,0). Two triangles are rasterized. The darker triangle has coordinates $(-0.5, -0.5)$, $(4.5, -0.5)$, and $(-0.5, 4.5)$, and the lighter triangle has coordinates $(-0.5, 4.5)$, $(4.5, -0.5)$, and $(4.5, 4.5)$.

The pixels in Figure 13.5(a) are assigned according to the rules of rasterization. These rules were designed to avoid competition over pixels that might be shared by multiple polygons. Pixels that fall in the interior of the polygon are assigned to the polygon. Pixels that fall on the shared edge of a pair of polygons are assigned to the color of the polygon on the right. If the shared edge is horizontal, they are assigned to the color of the polygon above. Hence the pixels that fall along the hypotenuse are assigned to the lighter triangle.

The two triangles in Figure 13.5(a) delineate two sampling regions. During texture-mapped rasterization, interpolated u, v coordinates may fall within the bounds of either of these two texture map triangles. These coordinates will likely not lie precisely at pixel centers. They instead need to sample a continuous function reconstructed from the discrete pixel samples by a reconstruction filter.

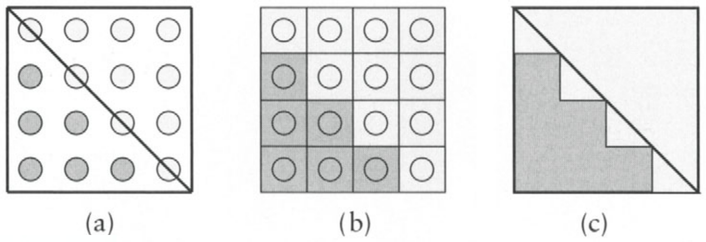

FIGURE 13.5 (a) Textures are stored as an array of pixels. (b) The nearest-neighbor filter returns the color of the closest pixel to the sampled location. (c) This can result in seam artifacts, shown here as the light triangle color bleeding into the darker triangle's sampling region.

The nearest-neighbor filter is the simplest such reconstruction filter, so we will begin our analysis with it. Nearest neighbor returns the color of the pixel closest to the sample location. Because the texture is stored in a rectilinear grid of pixels, nearest-neighbor sampling surrounds each of the pixel centers with square Voronoi cells of constant color. Hence the nearest-neighbor filter reconstructs from the discrete pixels in Figure 13.5(a) a continuous function consisting of square regions of constant color as illustrated in Figure 13.5(b).

Seams appear in part because the rules of texture filtering are inconsistent with the rules of rasterization. Samples taken anywhere within the lighter triangle draw from an appropriate nearest-neighbor pixel. But some positions in the darker triangle near the hypotenuse have, as their nearest neighbor, pixels assigned by rasterization to the lighter triangle. This results in the staircased seam in the sampling region of the darker triangle. The two triangles are unrelated and could appear in completely different locations on the object surface, representing completely unrelated textures.

We need to lay out the triangles in the texture map so they can be sampled correctly. The solution to this problem is to offset the triangles sharing the hypotenuse by one pixel horizontally (Rioux, Soucy, and Godin 1996), as shown in Figure 13.6. We rasterize an enlarged darker triangle with vertices $(-0.5, -0.5)$, $(5.5, -0.5)$, and $(-0.5, 5.5)$ and a translated lighter triangle with vertices $(0.5, 4.5)$, $(5.5, -0.5)$, and $(5.5, 4.5)$ as shown in Figure 13.6(a). The rasterization of these new triangles shades the pixels as shown in Figure 13.6(b), giving an equal number of pixels to each triangle. The texture coordinates u, v of the darker triangle have not changed, and so the smaller triangle is sampled properly from the overscanned rasterization as shown in Figure 13.6(c). The lighter triangle is similarly sampled from its translated position.

Bilinear filtering can also be supported (Carr and Hart 2002) by using the sampling regions shown in Figure 13.6(d). In this case, triangles that share a hypotenuse

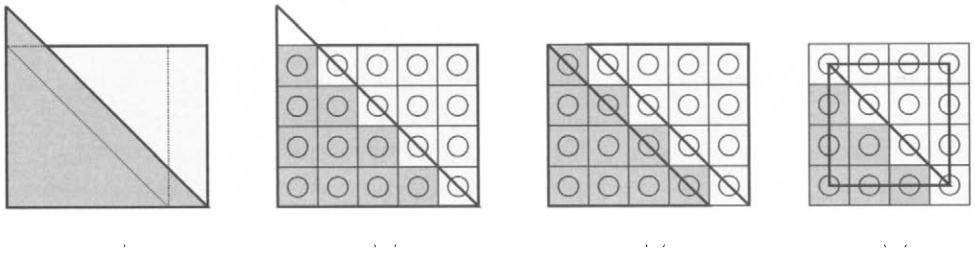

FIGURE 13.6 Avoiding seam artifacts.

in the atlas must share an edge in the mesh. The vertices of the sampling region have been inset one-half pixel, providing the buffer necessary to support bilinear filtering.

IMPLEMENTING REAL-TIME TEXTURING PROCEDURES

The procedural evaluation step of our real-time procedural solid texturing algorithm executes the texturing procedure on the texture atlas after the rasterization step has filled it with solid texture coordinates. The procedural evaluation step replaces the solid texture coordinates stored in the RGB values with a resulting texture color, which is applied to the object in the texture mapping step. This section explores various techniques for implementing fast texturing procedures. These techniques focus on implementations of the Perlin *noise* function (Perlin 1985), which is a common element found in many procedural textures.

One option is to implement the texturing procedure on the CPU. Several have implemented the Perlin *noise* function using special streamlined instructions available on Intel processors (Goehring and Gerlitz 1997; Hart et al. 1999). Our streaming SIMD implementation was able to run at 10 Hz on an 800 MHz Pentium III. The main drawback to CPU implementation is the asymmetry of the AGP graphics bus found in personal computers, which is designed for high-speed transmission from the host to the graphics card but not vice versa. In fact, we found that when we used the CPU to perform the procedure evaluation step, it was faster to also perform the atlas rasterization step on the CPU instead of the GPU because the result of the CPU implementation did not require a readback through the AGP bus.

However, we should take advantage of the power of the graphics accelerator. This means we should take advantage of the programmable shading features available on modern GPUs in the implementation of the procedure evaluation step. Doing so also allows us to take advantage of the GPU during the rasterization step.

A variety of implementations exist using different components of modern graphics accelerators. The *noise* function can be implemented using a 3D texture of random values with a linear reconstruction filter (Mine and Neyret 1999). A texture atlas of solid texture coordinates can be replaced with these noise samples using the OpenGL pixel texture extension or dependent texturing. Others have implemented the Perlin *noise* function as a vertex program (NVIDIA 2001), but a per-vertex procedural texture produces vertex colors or other attributes that are Gouraud interpolated across faces. Hence the frequency of the noise is limited by the frequency of the tessellation.

The Perlin *noise* function can also be implemented as a multipass pixel shader (Hart 2001). This implementation is based on the formulation of the Perlin *noise* function as a 3D integer lattice of uniformly distributed random values from 0 to 1 (see Figure 13.7). These discrete lattice values are reconstructed into a continuous function through interpolation, which locally correlates the random values. Depending on the application, this reconstruction can be C^1 smooth, using cubic interpolation, or fast, using linear interpolation.

The multipass pixel shader implementation of the Perlin *noise* function is based on the Rayshade implementation (Skinner and Kolb 1991). That implementation of the *noise* function uses 3D reconstruction filter kernels at the integer lattice points, with amplitudes set to the lattice point random values

$$\sum_{k=0}^{1}\sum_{j=0}^{1}\sum_{i=0}^{1}\text{Hash3d}(\lfloor s\rfloor + i, \lfloor t\rfloor + j, \lfloor r\rfloor + k)w(s,i)w(t,j)w(r,k)$$

The summation iterates over all eight corners of the cube containing the point s, t, r. For each of these corners, the function Hash3d(x, y, z) constructs a random

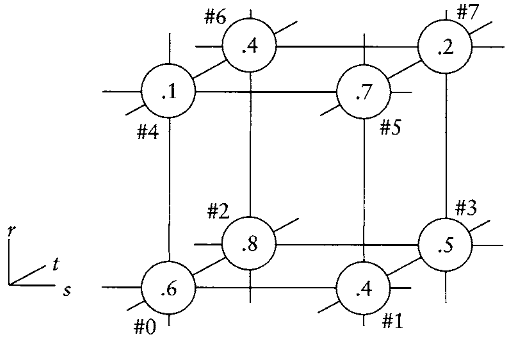

FIGURE 13.7 Noise based on an integer lattice of random values. Corner are labeled from #0 to #7 for later reference.

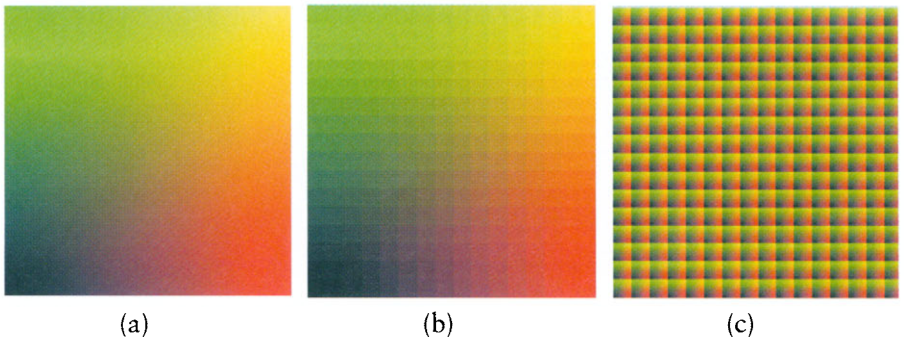

FIGURE 13.8 (a) A texture map of solid texture coordinates (s, t, r) ranging from (0, 0, 0) to (1, 1, 0), decomposed into (b) an integer part `atlas_int` and (c) a fractional part `weight`.

value t the integer lattice point x, y, z by performing an arbitrary set of bitwise operations on the integer coordinate values x, y, and z.

The Rayshade implementation of the *noise* function can be adapted to a multipass pixel shader. The basic outline of the multipass pixel shader implementation of *noise* is as follows.

1. The algorithm begins with the input RGB texture named `atlas` whose colors contain the interpolated s, t, r coordinates from the rasterization step (Figure 13.8(a)).
2. Initialize an output luminance texture `noise` to black.
3. Let `atlas_int` be an RGB texture whose pixels are the integer part of the corresponding pixels in `atlas`. Each pixel of `atlas_int` contains the coordinates of the lower-left front corner of the noise lattice cell that contains the coordinates of the corresponding pixel in `atlas` (Figure 13.8(b)).
4. Add the value one to each of the RGB components of the pixels of texture `atlas_int` to get the texture `atlas_int++`. Each pixel of `atlas_int++` now contains the coordinates of the upper-right back corner of the noise lattice cell that contains the coordinates of that pixel in `atlas`. Note that now all eight corners of the noise lattice cell can be constructed as a combination of the components of `atlas_int` and `atlas_int++`.
5. Let `weight` be a texture whose pixels are the fractional parts of the corresponding pixels in `atlas` (Figure 13.8(c)).

6. For $k = 0..7$:

   a. Let corner be an RGB texture equal to texture atlas_int overwritten with the texture atlas_int++ using the color mask (k&1, k&2, k&4). The texture corner now contains the integer coordinates of corner #k of the cell.

   b. Let random be a luminance texture whose pixels are uniformly distributed random values indexed by the corresponding pixels of corner. The texture random now holds the noise value at corner #k (Figure 13.9(a)).

   c. Multiply random by the red channel of weight if (k&1), otherwise by one minus the red channel of weight (Figure 13.9(b)).

   d. Multiply random by the green channel of weight if (k&2), otherwise by one minus the green channel of weight (Figure 13.9(c)).

   e. Multiply random by the blue channel of weight if (k&4), otherwise by one minus the blue channel of weight. These three instructions have now computed the contribution of that corner's value in the trilinear interpolation (Figure 13.9(d)).

   f. Add random to noise.

7. At this point the pixels of the luminance texture noise will now contain values linearly interpolated from random values indexed by the coordinates of its eight surrounding cell corners (Figure 13.10).

Our first implementations of this algorithm were on pixel shaders that only allowed 8 bits of precision (Hart 2001). These implementations used fixed-point

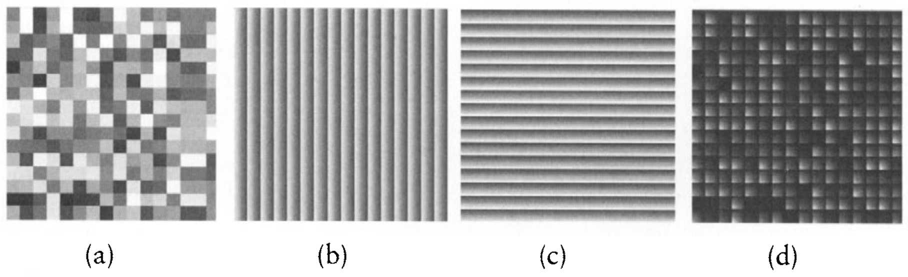

FIGURE 13.9 (a) Random values computed from atlas_int, weighted by (b) 1 − R(weight), and (c) 1 − G(weight) (and 1 − B(weight), which is uniformly equal to one), to produce (d) corner #0.

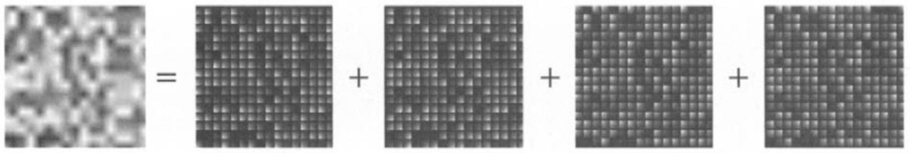

FIGURE 13.10 Noise is equal to the sum of corners #0 through #3. (Corners #4 though #7 are black since solid texture coordinate *r* is an integer, specifically zero, throughout this test.)

numbers with 4 bits of integer and 4 bits of fractional parts, and special pixel shader routines were developed for shifting values left or right to obtain the integer and fractional parts. Our original implementation also implemented a random number generator in the pixel shader, although modern graphics hardware now supports dependent texturing, which allows the `random` texture to be generated by using the components of the `corner` texture to index into a precomputed texture of random values.

APPLICATIONS

The real-time procedural solid texturing method is view independent. Hence, once the procedural solid texture has been computed on the atlas of an object, then the object can be viewed in real time. Each new view of the procedural solid texture on the object requires only a simple texture-mapped rendering, which is supported by modern graphics processors.

The most expensive operation of the real-time procedural solid texturing process is the generation of the atlas. Fortunately, the atlas need only be generated once per object. The atlas needs to be regenerated when the object changes shape. However, if the object deforms by changing only its vertex positions, then the atlas can remain unchanged (although the relative sizes of triangles on the object surface may have changed, which can result in a poor distribution of texture samples). Furthermore, if the object deforms but retains the same mesh structure, then the procedural solid texture will adhere to the object, overcoming the problem where the object swims through a solid texture, as shown in Figure 13.11.

The atlas encapsulates a procedural solid texturing of an object. The textured atlas can be attached to the object model as a simple 2D texture map, which is already supported by numerous object file formats.

The next most expensive operation is the procedural texture evaluation step mentioned earlier. The rasterization step can be performed on the graphics hardware, but we found it was sometimes useful to perform this step on the CPU given

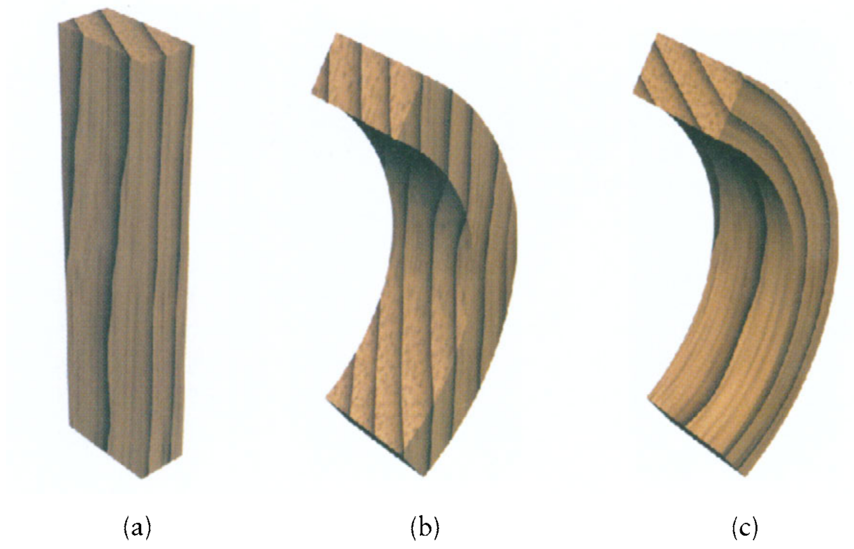

FIGURE 13.11 (a) A plank of wood. (b) A curved plank carved out of wood versus (c) a warped plank of wood. Image courtesy of Jerome Maillot, Alias|Wavefront.

the constraints of current graphics hardware. The first reason was that it allowed the CPU to apply the procedural texture without performing an expensive readback of the rasterized pixels. The second reason was that it gave us full control over the rules of rasterization, which can sometimes vary between hardware implementations. As readback rates improve and graphics hardware becomes more programmable, implementation of all three steps on the GPU will certainly be the better practical choice.

We have used these techniques to construct an interactive procedural solid texturing design system. This system allows the procedural solid texturing to be manipulated via parameter sliders. As the sliders move, the resulting procedural solid texture is reapplied to the object. Since the shape of the object is not changing, the atlas does not need to be recomputed, but the procedural texture does need to be recomputed on the atlas. Our implementations were able to support rates of 10 Hz for an atlas of resolution 256^2 using the host processor for rasterization and texture evaluation. These speeds will improve as graphics hardware performance continues to accelerate.

ACKNOWLEDGMENTS

Nate Carr was responsible for most of the work described in this chapter. Our work on this topic was supported in part by the Evans & Sutherland Computer Corp. Jerome Maillot was instrumental in developing the initial ideas behind this work.