5

ADVANCED ANTIALIASING

STEVEN WORLEY

It's very tempting to ignore the problem of antialiasing when writing textures. Certainly, the first textures anyone writes are very quick tools that make you happy to get any kind of pattern at all on your objects. But as you mature, you learn that even the simplest textures behave very poorly because of the problem of aliasing. This is why the topic of antialiasing is discussed so avidly in so many places in this book.

Aliasing has different definitions depending on context, but it ultimately resolves to the problem that you want to show the *average* effect of a texture over an area, yet it's a lot easier to just return a *sample* of the texture at just one infinitely small point. If you ignore the problem of antialiasing, you'll start making imagery that simply looks bad. You'll find artifacts like stippling or stairstepping in your images. You'll have especially visible problems when you make animations, since the motion of objects tends to highlight any aliasing problem by causing buzzing in the image—or worse.

It's very tempting to try to ignore the problem anyway, especially since most renderers have options to perform supersampling of the image for you to automatically reduce the aliasing problems. But don't depend on this! Your users may not need the whole image supersampled if the only problem is your texture. Image supersampling is expensive. *Supersampling is not an answer to the problem of texture aliasing.* It's just a final brute-force attempt to hide the problem.

The obvious objection to adding antialiasing abilities to textures is efficiency. Textures tend to be slow, and adding any more baggage (such as built-in integration for antialiasing) is bound to slow them even more, as well as increase the complexity of the code. This seems to argue for shorter, dumber textures, but in practice this is not true; a texture that can antialias itself can do so with a single (albeit slower) evaluation. Supersampling might take a dozen samples and still be less accurate. Correct antialiasing can therefore be an efficiency issue—and an important one.

Antialiasing is unfortunately a lot of work for the texture designer, since it often requires careful thought. It often takes a different design method to do proper

texture area integration, and the new method is never easier than simple point sampling.

This chapter covers only some aspects of antialiasing and in particular does not discuss band limiting, covered by Darwyn Peachey in Chapter 2.

INDEX ALIASING

It can be useful to look at the sources of aliasing in order to identify any components that we might be able to improve in behavior. In particular, nearly all textures can be summarized as being some formula or procedure (call it the "pattern" function) that returns a scalar defined over space. This value is transformed into a color or other attribute using a second function. This final transformation function might be something as simple as a linear gradient, but it can be abstracted to be a general lookup table that can characterize any behavior at all.[1] This design paradigm is convenient because it's not difficult to implement and is very versatile for the user.[2]

If we have a texture that follows this design method, we can see that aliasing is not caused from a single source, but from two. Imagine a point color sample being computed. The scalar pattern function is called, which returns a number. This number is used to index into the color transformation (lookup table), which identifies the color to return.

There is a source of aliasing in both steps of this process. Point-sampling the scalar pattern function obviously cannot characterize the average behavior of the pattern over the area. Less obviously, aliasing is caused in the color transformation step. A single value is used to determine the output color. This causes aliasing too, although it's more difficult to identify.

To illustrate a worst-case example, imagine our pattern function is a single scale of Perlin-style noise, and it varies between roughly -1 and 1. We choose a color map that is green for all input values except a very narrow band of bright red centered at the value corresponding to 0. This would make a pattern that is primarily green everywhere except for tiny spots of red where the noise value happens to be right at

1. A caveat here: since a table has only a finite number of entries, of course it can't perfectly represent any transformation. Luckily, in practice most of these transformations are simple enough that using just a few hundred entries does an excellent job of "summarizing" the transformation. Using a few thousand table entries is very likely to be adequate for any task. This is just a couple K of storage, so it's not a big overhead.

2. Note that there is more discussion of this methodology on pages 181 and 182.

0.0. Now, if we try to antialias this texture, we run into a large problem if our integration spot size is very large compared to the variation scale of the noise function. The average value of the noise over the large integration spot will converge to 0. Yet if we feed this perfectly integrated value into our color lookup table, we get a solid red color even though the true average color is a mostly green shade! This is a worst-case demonstration of aliasing in the final color transformation stage, sometimes termed *index aliasing*.

This aliasing is caused by rapid changes of the function used to transform scalar pattern values into output colors. Another example can help show how this aliasing might occur. Figure 5.1 shows a nontrivial color map that might be used to transform a fractal noise value into one component of the surface color. If we evaluate the fractal noise function several times, we get several corresponding color point evaluations. It can be seen that in this example, the samples lie in one region of the color map; does the average of the point samples accurately reflect the average of the region? We can convince ourselves that other noise samples are likely to occur through

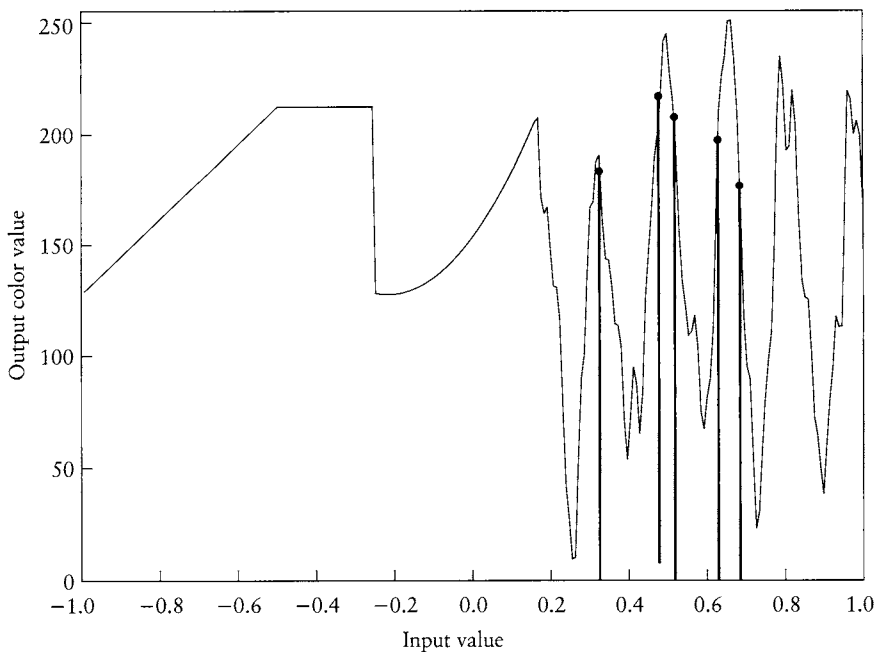


FIGURE 5.1 Point samplings of a complex color map.

the region from 0.3 to 0.7 and that the set of five current samples, by simple bad luck, probably shows a color value that is higher than the true mean color over this region.

With some thought, we can design a strategy to significantly reduce this source of aliasing. To do this, we need to think about what the true output should be and how it is created. Then we can examine our approximations and see how we might modify our strategy. We'll consider the scalar "pattern" function as a black-box function so the method will be applicable to any texture that uses a color map in this way.

The true output of an antialiased texture cannot be described by a single sample of our color spline. This is quickly seen by our red/green color example, since the average integrated color doesn't even appear in the color spline. If we consider what the average of an infinite number of point texture and color samples would converge to, we'll recognize the fact that the final average color is a weighted sum of the color spline. These weights are determined by the distribution of the pattern function's samples over the area in question. We can immediately see that since the *distribution* is what is important, a single sample value (like one evaluated at the mean of the distribution) is inadequate for determining the final average color.

What is the distribution defined by the pattern function over the integration area? *We don't know,* since we're treating the pattern function as a black box. We can make at least one assumption: the distribution is probably roughly continuous. We can make this assumption because most useful pattern functions (think of fractal noise, or a field that changes linearly with distance from a point or line, or a perturbed sine wave value) do not have discontinuities. Sharp changes *tend* to occur in the color transformation step, not the pattern function. If the pattern function is continuous, the samples taken over a local area will tend to cover a local range of values. This does not mean that all of these covered values will be equally likely, just that it's unlikely to have multiple separate peaks in the distribution with no samples in between.

Our only method of exploring the pattern function's distribution is to take point samples. We could implement naive supersampling by just indexing each of these samples into the color lookup table and averaging the output colors. But we can do better by thinking about the problem a bit more, perhaps with a simple example. Imagine we take just two samples of the pattern function. What is our best estimate for the final surface color?

Here we use our assumption of continuity. We can make an admittedly vague guess that the samples we took imply that future samples will occur between the two samples we have. Without any more information, a valid argument is that we can

make a best guess of the distribution of the pattern value by simply assuming that all values between the two samples are equally likely.

If we have a guess like this for the distribution of the pattern values, how can we turn this into a best guess for the final color output? For a certain input distribution, we can compute the final color distribution and therefore the mean of this output. Since we're modeling our input distribution as a box of equally likely values between the samples we've taken, we can simply integrate the color lookup table between these values to find the mean. The trick to try to eliminate the index aliasing is to try to *model the input distribution as best we can,* since then we can integrate the (explicitly known) color map using that distribution as a weighting to get as accurate an estimate of the average color as possible.

Two important questions remain: How do we best model that input distribution? And how do we compute the weighted integration of the color map efficiently? The first problem must be solved by sampling the black-box pattern function. For two samples, an argument can be made for using a uniform distribution between the two sample values. How about for three samples? If values of 1.0, 1.1, and 1.5 are returned, what is our best guess for the distribution? With such limited information, we can somewhat visualize a range of values between 1.0 and 1.5, with a lopsided distribution toward the low values. Yes, this might not be what the true distribution looks like, but it is our best guess.

Such a vague definition of a distribution is inadequate, and especially when higher numbers of samples are known, we must have a general method of computing an approximation to the input distribution. The best method seems to be the following: If we take N samples, we make the assumption that if we sort the values in ascending order, new samples are equally likely to occur between each adjacent pair of samples. This models a piecewise flat distribution, much like a bar graph. Figure 5.2 shows a collection of point samples from some (unknown) distribution, as well as a best-guess reconstruction of the distribution function. Each "bar" that forms the

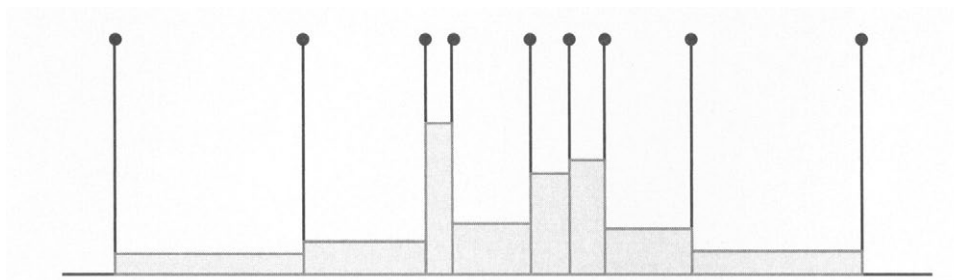


FIGURE 5.2 Reconstruction of an unknown distribution from point samples.

distribution denotes an equal probability, and therefore each has the same area. Note that this transforms a high density of samples (which are spaced closely together) into tall, high-probability regions.

This method for reconstructing a "best-guess" distribution has three advantages. First, it behaves as we would expect for small numbers of samples: one sample isn't enough to guess a distribution, two make a simple bar, and our example with three points does compensate for the lopsidedness of the sample points. Second, the reconstruction behaves better and better as more points are used. It will converge to the true distribution, even those with discontinuities or gaps. Third, we'll see that a piecewise constant distribution helps us perform the second required task for index antialiasing, which is to weight the color lookup table by the distribution and efficiently find the mean output color.

To do this integration, we can first develop a method of integrating a lookup table quickly over a range of values with uniform weight. This is equivalent to finding the sum of the table entries over the range, then dividing by the number of entries summed. There is an easy method for performing sums like this, called a *sum table.* These are in fact already used in computer graphics, primarily for 2D image map antialiasing. Here we wish to perform the sum on a mere 1D function, which is even easier.

A sum table is a simple concept. If we have an array of values and want to be able to sum those values over a certain interval, we can use a second, precomputed array called the sum table. Each entry of the sum table is equal to the sum of the entries of the original table up to that entry number. If our original table T were [1 3 4 2 3], our sum table S would be [1 4 8 10 13]. Now if we want to sum the values of entries a to b in T inclusively, we can simply evaluate $S(b) - S(a - 1)$.[3]

Thus, by two simple table lookups and one subtraction, it is an easy matter to sum any number of entries in the original table. Looking at our reconstructed input distribution, the piecewise uniform weights lend themselves to this summed table evaluation perfectly. Each interval of the distribution is assigned a weight (inversely proportional to its width). The mean value of the color table over the interval is found through the difference of two sum table entries divided by the number of entries spanned. This mean is weighted by the region's weight, and the complete weighted color sum is divided by the sum of all the weights to determine the final output color estimate.

Since the end of one interval is shared by the start of the next, we can even combine the terms containing the same indexing into the sum table. When we have N

3. A bit of care has to be taken when $a = 1$, where we assume $S(0)$ is 0.

samples of the pattern function, we only need N indexes into the sum table. This makes the evaluation extremely economical.

In some cases you can simplify the calculation further by calculating the mean and standard deviation of the samples. You can average the texture value over the color table range defined by the central mean value with total width equal to twice the standard deviation. This method simplifies the color map calculation to two lookups and also does not require sorting.

Index antialiasing is just one step of full texture antialiasing, but it's an extremely useful one. It's much more effective than blind supersampling, since (especially for low numbers of samples) at least a reasonable model of the variation of the texture is used to estimate the variation the color map undergoes. Implementation of index antialiasing is very easy, although it does assume a lookup table for your color map. This table will need to be filled, taking some preprocessing time and extra memory. This overhead is very small, however, even for a very large table, since the tables are only one-dimensional.

There are several extensions and adaptations of this algorithm, especially if you know something more about your pattern function. Sometimes you might be able to make assumptions about the shape of the sample distribution or be able to compute the true distribution analytically. For example, if the pattern function is defined by the distance of a point from a line, then the spot size defines a circle[4] of points. The true distribution of the sample values can also be determined algebraically or even geometrically. This can allow you to perform nearly perfect antialiasing! Obviously, the more you know about your pattern function, the better you'll be able to model its sample distribution over an area, and the better your antialiasing will become.

An Example: Antialiasing Planetary Rings

A great example of index aliasing is in generating planetary rings. The fine ring structure is a function of radius, but often the surface is viewed from far away and a single sample may range over a large range of radii, and many rings should be "averaged" together to provide a filtered version of the ring effect.

The planetary ring texture here uses the index aliasing reduction technique to provide a smoothly filtered version of the complex rings efficiently.

```
/* this is a 1D table, we can splurge for lots of entries and it still
   won't be too large to store. */
```

4. An ellipse, actually.

```
#define TABLESIZE 16384

int initialized=0;
float table[TABLESIZE];

static double RingTexture(double *pos, double spotsize,
  double inner, double outer, double transition,
  double varyscale, double ampratio,
  double low, double high)
{
  double R0, R1, weight;
  int I0, I1;

  /* We need a color table premade for us with the accumulated densities. */
  if (!initialized) MakeColorTable(inner, outer, transition,
  varyscale, ampratio, low, high);

  /* OK, let's find our range of radii. The range of radii are centered
     at the point's radius from the origin, and we split the spot radius
     between the two.
     To tweak the antialiasing, we could scale the spotsize up or down
     here before we use it.
     */

  R0=sqrt(pos[0]*pos[0]+pos[1]*pos[1]+pos[2]*pos[2])-0.5*spotsize;
  R1=R0+spotsize;

  /* R0 may be negative if we have huge spot sizes, which is obviously
     wrong and awkward. Let's make sure it's at least 0.0. */

  R0=MAX(R0, 0.0);

  /* OK, we know what range we want to average over. Let's transform those
     into index numbers in our color table. We know that at radius = [outer],
     we want to be at index [TABLESIZE-1]. So we multiply by a
     constant.

     Note that we lose a little resolution because we quantize the
     index into an integer. If the table were smaller, we could
     compensate by using a linear interpolation, but with a really big
     table the effect is negligible, and it definitely makes the code simpler.

  */

  I0=(int)(R0*(TABLESIZE-1)/outer);
  I1=(int)(R1*(TABLESIZE-1)/outer);
  /* We make a tiny change to make sure I0 and I1 are not coincident, which
     simplifies tests later. This case will rarely happen anyway. */
  if (I0==I1) I1++;
```

```
  /* These indexes may be out of range. If so, let's do the right thing. */
  if (I0>=TABLESIZE) return 0.0; /* We're completely outside the rings,
    no effect. */
  if (I1>=TABLESIZE)
    {
     /* Our outer range has run "off" of the end of the result. We have to
take this into account by figuring the ratio of what's fallen off
to what remains, and make sure our final output is properly
weighted. */

     weight=((double)(TABLESIZE-I0))/(I1-I0);
     /* Now we change I1 to start within the range. The weight parameter
will compensate for the range change. */
     I1=TABLESIZE-1;
    }
  else weight=1.0; /* We're fully within the rings, we'll use the full
     weight. */

  /* We now want the average value between I0 and I1. This is the
     easy part! */

  return weight*(table[I1]-table[I0])/(I1-I0);
}

/* Routine to build the summed color table itself. This includes the
   computation of the ring density tucked within a simple loop to
   build the sum table as it goes. This is a precomputation that only
   needs to be done once. */

static void MakeColorTable(double inner, double outer, double transition,
    double varyscale, double ampratio,
    double low, double high)
{
  double R, A, F;
  int i;

  /* Sweep the radii out to the outer radius. Accumulate samples in
     the summed color table. Point-sample the ring density at each sample.
     A radius of 0 is index 0. A radius of [outer] is equal to the table
     size-1.
  */

  table[0]=0.0; /* Start with a 0 table */

  for (i=0; i<TABLESIZE-1; i++)
    {
     R=outer*(i+0.5)/(TABLESIZE-1); /* R varies between 0 and [outer] */

     /* Compute the simple inner/outer transitions to form an alpha channel
 of a simple washerlike disk. This will be used to modulate the
```

```
 density of the fine rings and prevent any rings from being too
 close or too far from the center. */

     if (R<=inner) A=0.0;
     else /* In first transition zone? */
if (R<inner+transition)
  {
    A=(R-inner)/transition; /* Linear 0 to 1 ramp */
    A*=A*(3.0-2.0*A); /* Hermite curve smooth transition */
  }
else /* In outer transition zone? */
  if (R>outer-transition)
  {
    A=(outer-R)/transition; /* Linear 1 to 0 ramp */
    A*=A*(3.0-2.0*A); /* Hermite curve smooth transition */
  }
  else A=1.0; /* We're in the main body of the ring. */

    /* Now let's compute the ring density. We use a 1D version of
 fractal noise. We use a 3D noise routine but pass in (R, 0, 0)*/

    F=fractal3(R, 0.0, 0.0, varyscale, ampratio);

    /* F is now between -1 and 1. But we use the low and high values as
 a clipping range for the noise. */

    if (F<=low) F=0.0; /* F is too low, we set the ring density to 0.0. */
    else if (F>=high) F=1.0; /* Full ring density */
    else /* We're in a transition zone. */
{
 F=(F-low)/(high-low); /* Now a 0 to 1 range */
 F*=F*(3.0-2.0*F); /* Hermite it to make it smooth */
}

    /* OK, our ring density is F and our shaping alpha value is A. The
 net density is the PRODUCT of these two. */

    table[i+1]=table[i]+R*F;
    }
    initialized=1;
    return;
}
```

SPOT GEOMETRY

Your antialiasing goal is usually to find the *average* texture value over a small area. This area is known as the *spot size*, and usually the renderer will tell you what this size is. Some renderers like RenderMan are very careful in computing these spot

sizes, but others (especially ray tracers, it seems) are very careless and give no spot size or (perhaps worse) an incorrect spot size.

This spot size is easy to understand if you think about a very simple renderer that takes one sample per pixel and renders a simple plane with a texture on it. Since it's rendering with just one sample per pixel, the texture algorithm will ideally return the average color of the texture pattern over that pixel. But textures don't know or care about pixels; they almost always want to be given a *location* in texture coordinates. What you usually want is a texture center location, in *XYZ* coordinates, and a radius (the spot size) measured in that same coordinate system.

If you're lucky, the renderer will give you this radius at the same time as the sample location. For example, RenderMan's rendering method chops objects into smaller and smaller parts until each bit is smaller than a pixel when it is projected onto the output image. During this slicing and dicing, it keeps track of the exact range of texture coordinates for each little bit, known as a micropolygon. It ultimately reduces surfaces to a tiny rectangular chunk of texture with exactly known texture coordinate ranges, which it passes to the texture to be evaluated. RenderMan is extremely texture-friendly because of this careful coordinate treatment.

More conventional renderers usually concern themselves with projecting the geometry onto the screen and then, grudgingly, determining the texture coordinates to pass to the texture code. These tend to be point locations computed by starting with a world coordinate value and transforming first to object and then to texture coordinates. The spot size is then ideally computed by analyzing the transformation between texture space and image space and using this to approximate a texture spot size.

If the previous sentence sounds like vague hand-waving, that's because *very few renderers compute spot sizes very well.* They all use different techniques and approximations. I've had personal experience with three different renderers, and each had inaccurate spot sizes. There are several ways of dealing with this inaccuracy, even if you have no control over the renderer itself. One method is to find correction factors to "massage" the spot size back to be more accurate by using different constants to scale the spot size.[5]

Another method is to compute the spot sizes yourself, if you have enough information. The derivation is straightforward but involves several steps and simplifying assumptions. It can be frustrating to locate all the information needed to complete each step, since renderers often don't give you all of the surface data you need.

5. This is straightforward to do with the image-based verification method described on page 174.

First, we must know what size the spot we are antialiasing is in the *output image*. This is often the size of one pixel, but in the case of image supersampling the size may be smaller.

We first need a conversion to change a world coordinate into an image coordinate I. (This is the formula that takes any 3D location in world coordinates and predicts where the point will project onto your output image.) This relation is usually computed by two transformations. The first is a rotation and offset transformation to translate the camera to a coordinate system where it is at 0, 0, 0 and looking forward. The second transformation is the perspective transform, which projects the (transformed) 3D point onto the screen. They are often combined into a single operation, although it may be easier to think of them as two sequential operations.

We then need a transformation from world coordinates to texture coordinates. This is usually a simple linear matrix equation with an offset. It is often the concatenation of two transformations, the first from world to object space and the second from object to texture space.

By concatenating the effects of each transformation in the sequence together, we arrive at a nonlinear transformation function, $I(T)$, which relates texture coordinates to screen coordinates. Its exact representation depends on your camera model and coordinate system definitions, so even its form tends to be unique to every renderer.

Next, we make an assumption that the texture area we are averaging over is so small that it can be well approximated by a planar sample through the texture. This is not always true (think of a sphere that is so distant that it is a single pixel in size), but the assumption holds well in almost every case.

If the antialiasing spot we are averaging over is a flat 2D area, the direction perpendicular to this region must be the same direction as the surface normal (in texture coordinates). The surface normal in fact defines the plane's normal. We can pick two vectors (call them R and S) that are mutually perpendicular to each other and also to the surface normal N. These three vectors define a new coordinate system. We know that the texture spot is defined only in the plane of R and S.

We can choose which R and S to use by thinking about the geometry of the texture spot. If the surface normal N directly faces the camera, the spot will be a flat circle, and we can use any R and S that are perpendicular to N since there's no preferred spot direction.

If the surface normal does not directly face the camera, the texture spot will be tilted slightly away from us. This will make the texture spot *elongated* in one direction. If the view direction toward the camera is V, the elongated texture direction will lie in the direction of V projected onto the surface.

We want to align our texture spot in this direction in order to simplify our antialiasing later. In practice, we often do this in world coordinates first. We start

with V_w, the world coordinate view direction from the texture location toward the camera, and N_w, the surface normal in world coordinates. We form S_w as the *non-elongated* direction by taking the cross product $V_w \times N_w$ and normalizing. R_w, the direction along the elongation, is computed by normalizing $S_w \times N_w$. We then transform S_w and R_w into texture coordinates by using the world-to-object and object-to-texture transformations.

If the texture sample position (which defines the center of the area we want to average over) is T, we can parameterize the region we want to average over as $T + rR + sS$, where r and s are scalars. But we don't know what range of r and s should be used—yet. To determine them, we can use our formula we computed earlier for transforming from texture coordinates to screen coordinates. If we substitute our parametric spot equation into the projected image formula, $I(T + rR + sS)$ tells us where on the screen a texture sample at parametric coordinate (r, s) falls.

We can now differentiate I with respect to r and s. We do this twice, since I is really a coordinate pair (x, y). This gives us $\frac{dI_x}{dr}$, $\frac{dI_y}{dr}$, $\frac{dI_x}{ds}$, and $\frac{dI_y}{ds}$.

We can stop here and return an isotropic spot size that gives just a single radius to average over by noting that if we want to average over a pixel, we want to move a small delta of 0.5 pixels up, down, left, and right in the image. We can use something similar to

$$\Delta = \sqrt{\left(\frac{dI_x}{dr}\right)^2 + \left(\frac{dI_y}{dr}\right)^2 + \left(\frac{dI_x}{ds}\right)^2 + \left(\frac{dI_y}{ds}\right)^2}$$

to get a kind of "average spot radius." This result isn't bad, and most renderers stop here and give a value similar to this. We can stop here if we want and just integrate our texture over $(-\Delta \,..\, \Delta, -\Delta \,..\, \Delta)$.

But we've nearly derived a much higher-quality spot computation that accounts for stretched spots. We know how I changes with both s and r. We can set up two linear equations,

$$\left(\frac{dI_x}{dr}\right)\Delta r + \left(\frac{dI_x}{ds}\right)\Delta s = 0.5$$

$$\left(\frac{dI_y}{dr}\right)\Delta r + \left(\frac{dI_y}{ds}\right)\Delta s = 0.5$$

which states that we assume our image spot to vary half a pixel in both x and y, and we expect r and s to vary by $\pm\,\Delta r$, $\pm\,\Delta s$ to allow the texture spot to vary over the entire pixel. The solution of these equations is

$$\Delta s = \frac{\frac{dI_x}{dr} - \frac{dI_y}{dr}}{2\left(\frac{dI_x}{dr}\frac{dI_y}{ds} - \frac{dI_x}{ds}\frac{dI_y}{dr}\right)}$$

and

$$\Delta r = \frac{\frac{dI_x}{dr} - \frac{dI_y}{dr}}{2\left(\frac{dI_x}{dr}\frac{dI_y}{ds} - \frac{dI_x}{ds}\frac{dI_y}{dr}\right)}$$

These give us the ranges $(-\Delta r, \Delta r)$ and $(-\Delta s, \Delta s)$ that multiply the R and S direction vectors to define the texture antialiasing geometry. You can view this as an ellipse or rectangle in texture space. This is a great spot geometry because we can antialias over an elongated spot if we care to, and we can also change our spot size if we know more about the shape of the sample in image space.

This computation seems somewhat daunting, with multiple transformations to compute on the fly and terrible math equations to solve. In practice, it's not quite so bad because the transformation formula $I(T)$ doesn't change from texture sample to sample. The work that must be done does involve several multiplies and divides, but most of the overhead comes from the square roots used in normalizing the S and R vectors.

With a computation like this, it's easy to see how a renderer can give poor spot size estimates if it is careless.

SAMPLING AND BUMPING

Some textures can be analytically integrated, like step functions and thick lines. Other textures have certain behaviors that can give at least better approximations than point sampling, such as band-limiting fractal noise scales to match the sample size. However, really odd textures just aren't practical to modify or even understand because of their custom design or complexity.

Supersampling is a last resort to reduce aliasing artifacts. It always works. It's usually inefficient and crude, but if we're forced to do this supersampling, we can at least do it intelligently.

We can benefit from having the *texture* perform its own antialiasing (even by supersampling) instead of the renderer. A short header at the beginning of the texture does its own supersampling of the surface area and returns the mean result. This is useful for several reasons. First, this means that all textures are treated the same by the renderer; a position and spot radius are passed, and the texture returns an

integrated texture estimate. The renderer does not need to treat antialiasing textures differently than ones that can only be point-sampled. Second, since the evaluation loop for the samples is within the texture, not the renderer, overhead (in particular, function calls) are reduced. This is a small savings in general, but for simple textures like checkerboards it can be significant in proportion to the texture's overall speed.

But how should a texture antialias itself with point samples? How does it pick the locations? Luckily, we already know the answer! The previous section on "Spot Geometry" has told us exactly how to vary our samples to cover the range of the texture's spot.[6]

In the simplest case, we just need to evaluate the texture multiple times over the range of spots defined by Δr and Δs in our texture coordinates. The average response of these samples is an antialiased supersample of the texture. But if you've read the "Index Antialiasing" section, you'll realize that technique is designed for this kind of point supersampling.

But there's a further bonus that is very easy to miss. A very common technique of bump mapping is to compute a vector derivative of a function and "perturb the surface normal" with it. But many times, this perturbation is done incorrectly because it often looks good anyway! In particular, if you take the derivative of a function (for example, fractal noise) and simply add it to the shading normal and renormalize, you'll get good-looking bumps, as shown in Figure 5.3(a). This is exactly what Ken Perlin did in his 1985 paper with its fabulous images. It's also what I did for dozens of commercial procedural textures. *But it's wrong!*

The right answer is to use the bump-mapping formula first shown by Blinn (1978), which Darwyn Peachey discusses in Chapter 2.[7] It's easy to skip this because the math is annoying. But, if you look at Figure 5.3, you'll see why understanding the texture spot geometry is so important.[8]

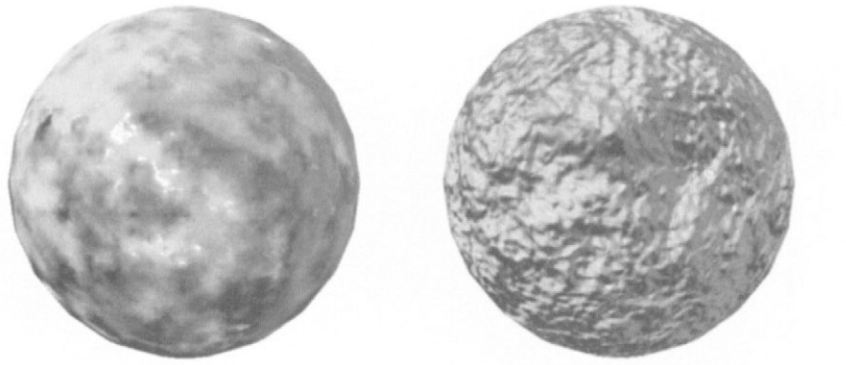
Why does the "simple" method work? Because it looks good and especially for fractal noise patterns you don't have any "correct" sample for your eyes to compare it to. There are clues, however, that indicate that it's wrong, especially if you animate

---

6. I know you skipped that section because it didn't seem exciting and had some ugly-looking math formulas. You're reading *this* section first because of the bumpy sphere pictures. You can improve your textures too, once you understand your spot geometry, though . . .

7. And again, a very good renderer will do all of this for you. This is one of the simple secrets of why RenderMan procedural textures tend to look so good! But many other renderers, even high-end commercial ones, do *not* do this for you!

8. I wish I had understood this for my 1992 commercial texture collections!



(a) (b)

FIGURE 5.3 Incorrect bumping of many textures: (a) "classic" bumping versus (b) correct.

your object rotating; the lighting will simply be inconsistent because the normal is incorrect.

When we know the spot geometry, we have all of the information we need to do "correct" bumping. If we're doing a small supersampling of our texture (to help eliminate aliasing), we can use the same engine to compute the bumping for us simultaneously. We don't even need to evaluate the derivative of our texture, since our point samples can tell us the average derivative over our spot, which is all that matters.

If we're sampling over our texture spot, we might have part of our texture return a single scalar value for the bump height of the surface. If we evaluate the height over the whole spot, we can fit an average "tilt" to the texture spot. We then just tilt our surface normal to match the tilt of the texture spot, and we get the correct surface bumping.

Luckily, computing this tilt is easy! It's just a least-squares fit of the arbitrary texture height function $H(r, s)$ to a line, repeated for the R and S directions. If we take samples symmetrically around the center of the spot, the slopes of the tilts are easy to compute:[9]

$$\frac{dH}{dr} = -\frac{\sum rH(r,s)}{\sum r^2}$$

9. Using a 1D least-squares line fitting method, which isn't perfect for super-steep slopes but is more than good enough.

and

$$\frac{dH}{ds} = -\frac{\sum sH(r,s)}{\sum s^2}$$

We then add these slopes to the surface normal. We need to do this in *world coordinates* since that's probably what the renderer expects. But this is just

$$N_{\text{bumped}} = \frac{N_{\text{original}} + \frac{dH}{dr} R_w + \frac{dH}{ds} S_w}{\sqrt{\left(1 + \left(\frac{dH}{dr}\right)^2 + \left(\frac{dH}{ds}\right)^2\right)}}$$

The √ term is just to renormalize the surface normal. Note that we are using the world coordinates R_w, S_w instead of texture coordinate R, S.

As a practical example, imagine sampling our texture at four texture locations, at the corners of our texture spot. We have some height function (maybe a fractal noise value) $H(r, s)$. So we can expand out the equations and find

$$\frac{dH}{dr} = \frac{\Delta r(H(-\Delta r, \Delta s) + H(-\Delta r, -\Delta s) - H(\Delta r, \Delta s) - H(\Delta r, -\Delta s)}{4(\Delta r)^2}$$

and similarly for S. If you look closely, you can even see how this is a finite-difference approximation to the derivative. But it's better than knowing even the exact true derivative at the center of the spot, because this is an *average* derivative over the whole spot and it will not suffer nearly as much aliasing.

This entire section on spot geometry is really a topic that authors of renderers should understand and implement, but unfortunately they often don't. Luckily, we texture authors can sometimes do the work ourselves to get good results. The proper texture spot definitions can give better antialiasing, proper sampling, and great bump mapping.

OPTIMIZATION AND VERIFICATION

After you've added antialiasing support or new spot geometry computations, it can be difficult to determine exactly how well it's working. The most common method is to render an image, zoom into the detail, see that it looks kind of blurry, and congratulate yourself.

This obviously isn't very scientific! In fact, it's hard to judge by eye whether your antialiasing is adequate or not. You can argue that if your eye can't tell, it doesn't matter, but in practice it's good to minimize it even below visible levels, since a

different application (perhaps with more extreme conditions) may amplify even a small amount of aliasing artifacts into a real problem.

There's a surprisingly straightforward and useful method for tweaking antialiasing for optimum results. It is important to reduce your measure of antialiasing "goodness" into a measurable number that you can actually try to optimize.

This can be done in nearly any situation by forming a controlled scientific experiment. It's usually very easy to adapt the following procedure to optimize antialiasing in every case.

First, you need to design a scene to render that exercises your texture. Strive to include situations that will typically cause problems. You want to have a single image that shows your texture at different scale sizes (infinite planes work well for this) as well as different viewing angles (spheres work well for this). I often use four infinite planes forming a box, receding to infinity, with about 10 spheres at different depths.

You need to generate a "reference" image of this scene, one that is as accurate as possible, including texture antialiasing. This can be done by rendering your scene at very high resolution, such as 4K by 4K, then filtering the image down in size to something more manageable like 256 by 256. This "manual supersampling" is nearly foolproof in making a good antialiased image since so many samples are used per pixel. Be wary of letting your renderer do supersampling for you when building a reference image! Its own supersampling may be biased or have subtle errors of its own.

This reference image provides a measure that we can compare our antialiasing against. When we render with no image supersampling, we can detect errors due to aliasing by simply comparing the rendered image with the reference image, pixel by pixel. The match is far from perfect because the reference image also includes *geometric* antialiasing, but this doesn't upset our texture antialiasing comparison.

The comparison between our test render image and the reference image should be done numerically. The most obvious error metric is to use a simple sum of the squared differences for each pixel. If the images are identical, this value will be 0. You can write a small application that takes the difference between two images and returns the *numeric* error value. It can also be interesting to output an image that highlights the pixels that have the most error.

With a tool for determining the antialiasing error, it becomes very easy (and, surprisingly, a little fun if you're a math geek) to optimize your texture antialiasing to minimize it. In particular, the simplest yet most useful variation to explore is a "tweak" value that scales your texture spot size.

Even if you've been careful in determining spot size, it's very easy for the spot size to "drift" from its optimal value. It is not uncommon for the spot size to be off significantly! One renderer I deal with returns a spot size that is at least four times larger than it should be, and I have to scale it down appropriately to reduce the error.

With an error metric in hand, you can vary any part of your antialiasing code to determine whether it has a positive effect. Are there strange constants in your antialiasing code for dealing with spot size or sample rates? How about estimates of texture variability for use with index antialiasing? Cutoff frequencies for fractal noise? You can tweak all of these to optimize your algorithm's output.

You can also use the reference comparison to investigate the efficiency and quality of different algorithms. You can test a supersampling method to determine its speed and error as compared to a more intelligent but slower band-limiting method.

This method does not do well at catching *temporal* aliasing problems since it considers only a single image at a time. You could make a similar test that uses multiple images to determine the antialiasing error metric. Even this is not perfect, but it does help in better characterizing the final error.

EMERGENCY ALTERNATIVES

Sometimes with deadlines looming, you may still be tearing out your hair due to aliasing problems. While you can always throw more supersampling onto the problem, this often still isn't acceptable because of time or CPU limits. The ideas that follow may be technically dubious and even repugnantly crude, but elegance tends not to matter when you have a shot deadline in three hours.

The most common and obvious texture "tweak" is to scale the texture spot size. Even without a reference image comparison as discussed in the previous section, you can usually reduce aliasing artifacts by using an exaggerated spot size. Often this may eliminate aliasing at the expense of a softened look to your surface.

If your texture is aliasing and it uses a color map, you can try applying a blur to the color map. This will soften the transition zones between colors, which can hide a lot of terrible artifacts. Since the blur only has to be done once to the simple 1D color map, the computational overhead is inconsequential and does not slow final rendering. I first used this method to solve an aliasing problem, but I later found that it was a useful control for users to use at their own discretion.

A painful alternative that should only be used in true emergencies is a simple image blur effect. Make a mask that isolates the pixels that show the texture, and apply a 2D image blur to the final rendered image using that mask. This mask may be an

alpha channel that the renderer can output, or even a hand-painted one. By applying a small blur of just one or two pixels radius, aliasing artifacts in your texture are usually hidden very quickly. It has the unfortunate side effect of softening your surface features and even geometry details.

A final desperate alternative is effective with simple geometry. If you render your texture out as a 2D image, you can use the rendered image as a map on your surface. Most renderers have decent image map antialiasing using a MIP map or summed area table. Two-dimensional antialiasing with these methods tends to do especially well in high-compression areas, where a large amount of detail gets crammed into just a few pixels.