# Solid Procedural Texturing

## Solid texturing: What?

To explain solid texturing, I'll quote a rather good book on ray tracing.

While texture mapping is a powerful technique, it does have its limitations. The most obvious limitation is mapping a two-dimensional image onto a three-dimensional object without excessively distorting the texture-mapped image. Flat surfaces are no problem, but surfaces with complex shapes can be difficult to map. Even for a relatively simple surface such as a sphere, the texture map must be distorted to get it to fit onto the three-dimensional surface. For spheres, this results in the compression of the texture at the poles.

Peachey and Perlin simultaneously developed the idea of solid texturing to solve this problem. The underlying principle of solid texturing is to create a three-dimensional texture map from which the textured object appears to be carved. This texture map either may be defined explicitly as a three-dimensional array of values (which consumes a huge amount of memory) or be defined by a procedural function. The procedural function takes an (x,y,z) point and returns the surface characteristics at that point. Perlin introduced the noise() function to generate many of his procedural textures. To this day, the images he produced for his SIGGRAPH paper in 1985 are considered some of the best in computer graphics. [Watkins 1992]

## Solid texturing: Why?

### *Warping of textures*

As is explained above, even if you have a cool 2D bitmap to use as a texture, it tends to get squashed and stretched. Some mappings are pretty simple, but imagine trying to map a 2D bitmap onto, say, a car-shaped mass of primitives for your cool ray-traced Ferrari Testarossa. It might well be possible, but better you try than I.

### *Difficulty of finding appropriate 2D textures for wood, marble, etc.*

Something else to consider is the problem you'll have in finding a 2D bitmap for a marble staircase or a wooden bureau. You might be able to find a decently convincing texture for it, but those things are easier in 3D, as we'll see later. Also, finding a texture that looks good on one marble object doesn't guarantee it will look good on all marble objects. If the object folds back on itself, you might want veins in the marble or grain in the wood to match up or otherwise generally have the texture stay continuous. Also, if all your objects use the same texture, they'll look pretty boring at best, and weirdly unnatural at worst.

## Solid texturing: How?

### *A 3D function, and what to use it on*

This lecture will focus on solid procedural texturing. The alternative would be to use bitmaps, as is mentioned above. Nobody actually does this, and the memory cost is hideous, so we'll ignore it.

The easiest way to think of solid procedural texturing is to imagine that all your objects are being carved out of some material. You're basically describing to the ray tracer what color the material is everywhere inside it. Then, when the ray tracer wants to know what color a point is, it just calls your texture function to find out what color the material is at that point.

Let's consider how this fits into texturing as you're doing it now. Lighting is going to stay the same, because no matter what the surface is made of (as long as it doesn't cast its own light), it'll be darker where the light doesn't hit it. What you're really doing here is making a function to figure out what color the object is, and calling it every time you get an intersection. So, in your shading code you won't multiply the amount of light by the color in the material structure. Instead, you'll call your texture function to find the color, and multiply by that.

Now let's think about what a texture function looks like. It needs a point to work with. That will be the point where the ray intersects the surface, since that's where we want to know the color. So, your texture function will look something like:

Color really_cool_texture (Point p) {...}

My texture functions look like this:

void tex_whatever (Point p, Color *c) {...}

This isn't any better, or really any different, but I point it out so you won't be confused when the example code doesn't look exactly like the first prototype.

You're now equipped to try out neat texture stuff in your ray caster. I recommend starting it early, because while it isn't very difficult, it's lots of fun.

## *Simple checkerboard example*

To show you how texturing works, let's look at a simple example: a 3D checkerboard pattern.

To make life easier, we'll make the checkerboard alternate from black to white on integer boundaries. For a point (x,y,z), you can take the numbers xi, yi, and zi which are the integer parts of x, y, and z, and ((xi + yi + zi) mod 2) will alternate the way we want. An easy way to convince yourself of this is to just think what will happen whenever we add or subtract 1 from xi, yi, or zi.

Here's a simple piece of C code to do just exactly that:

```
void tex_checkerboard(Point pt, Color *col) {
int x,y,z,c;

x = floor(pt.x);
y = floor(pt.y);
z = floor(pt.z);

c = (x+y+z) % 2;
if ( c ) {
      col->r = 0.0;
      col->g = 0.0;
      col->b = 0.0;
} else {
      col->r = 1.0;
      col->g = 1.0;
      col->b = 1.0;
  }
}
```

# Noise: What?

## *What does a noise function do?*

A noise function perturbs points before you find their color. Noise functions tend to be wavy and irregular looking, which is what gives them their name.

By "perturbs points before you find their color", I mean you find the 3D noise vector for a point, add it to that point, and then look up the color at that point, rather than finding the color directly. This has several consequences.

First, it defines what a noise function looks like. A noise function will take a 3D vector, and return a 3D vector that has been moved slightly. So, it takes an input point and returns a similar, but perturbed, output point.

This also means that a noise function affects an existing texture instead of being a new one. So you wouldn't replace the checkerboard texture above with a noise texture, but you might replace it with a noisy-checkerboard texture.

Finally, this means that the texture will look different. As a very simple example, imagine you used a noise function that always translated the input vector by (3,0,0). This would translate the texture by -3 in the x direction. As you'll see later, that's not a very good noise function, at least in the way most people use them, but it does illustrate that the texture is somewhat different with noise.

## *Why use noise?*

A neat thing about noise is that it makes a regular texture look wavy, but still recognizably like itself. A great example of this is wood grain. Because trees grow in rings, which is what causes the wood grain, a tree looks like a whole bunch of concentric cylinders. So, if you cut it crosswise like a stump, you see a lot of rings. If you cut it lengthwise, like a board, you see a lot of lines running along the board.

Now it's no surprise to any of you that trees aren't made of perfect cylinders, but are instead rather wavy and irregular. Imagine if you made the cylinders wander just a bit, so they were more-or-less still cylinders, but with a bit of curviness to them. That is exactly what a well-chosen noise function can do for you. That's why people use it for marble, wood grain, and other wavy, irregular textures.

## *Characteristics of a good noise function*

To understand what goes into a good noise function, let's look at the wood grain example again.

For one thing, the noise function should be continuous and pretty smooth. If it weren't, we'd see jumps in the texture. We want something that's wavy, but we don't want any skips or jumps.

For another thing, the noise function should vary pretty randomly when you look at points that are far apart. That's one problem with the noise function above that just adds 3 to the x component of the vector. It's too predictable, even though it's smooth and continuous.

Finally, the noise function should avoid repeating too often. If there is visible repetition in the noise function, you'll see identical sections of marble or wood or whatever, and that's a problem, as was discussed back in the problems with 2D texturing.

## *One conceptually simple noise function*

One way to make a noise function is to make an NxNxN block of points for some N, and to find the amount of noise at a point in the block, interpolate between them. To find the original vectors, just randomize the three components. The vectors shouldn't be normalized, although making sure that they have a maximum length of 1 might be a good idea. To ensure that, just re-randomize any vector longer than that.

This method has some problems. The first and biggest problem is the memory use. It isn't as big and space-hungry as a 3D bitmap, but it's similar. A second problem is that if N is too small, it visibly repeats. Unfortunately, the bigger N is, the bigger the memory problem is. Still, this is a good example noise function, and is good enough for many simple applications, especially with a few modifications to keep it from visibly repeating.

The remainder of this document will assume you're using a noise function that looks a lot like this. There are many other types of noise functions, but fewer are commonly used, and most act in a very similar way.

### *The supplied noise function*

The noise function that you're being given is a little more complicated, but takes O(N) space instead of O(N^3). Basically, it defines the NxNxN grid in terms of an N-vector, and uses a simple hashing function to find what element of the vector a particular grid point maps to. It uses quadratic interpolation between points, which is a little rough looking. As you can probably tell from the description, it's designed for speed, and sacrifices a lot to gain it. Many other noise functions use cubic or higher-order interpolation for additional smoothness, and most use a better hashing function than the supplied noise function.

Still, a fast, simple function can have its uses. Specifically, a fast, simple function can be called a lot more times per point to get a better appearance. To find out how, read the section on turbulence below.

## Turbulence

A neat trick you can do with noise functions to make them look nicer is turbulence. Turbulence just involves calling the noise function repeatedly and multiplying the amount of noise by smaller and smaller amplitudes to get a much visually busier pattern. A noise function will look quite smooth, since it will use interpolation between grid points. A turbulence function looks much less smooth since it sums the noise function at a number of points. The noise function you are given uses only quadratic interpolation, and as a result is very fast. So, using it for 3-level turbulence (and incidentally calling the noise function 3 times) will be faster than using a better noise function for 2-level turbulence, and calling the more complex and slower noise function twice. While this is something of a matter of taste, more levels of turbulence will usually look better than smoother interpolation.

For your convenience, a turbulence function is supplied in the same file as the noise function you have been given.

## Animation: Problems

### *Crawling textures*

A problem with noise the way we're doing it above is that it animates poorly. This won't be a problem with P3, because you're not doing animations, but if you were, you'd need to make some modifications to the texturing algorithm above. To understand why, let's go back to the checkerboard example.

Imagine you have a checkerboard-textured sphere that moves left at a constant speed. Now look at its leftmost point. That point will move, and will switch from white to black and back quite often, as the point moves left. One good way to solve that is to store a transformation to and from object space for each object.

An object's object space is a coordinate system that is constant for that object, so that when that object moves, it's object space coordinates don't change, but all other objects change in its object space.

Let's look at an example. Take the (uniformly scaled) sphere above. In its object space, we will define its center to be (0,0,0), and a particular point we choose on its surface to be (1,0,0). We will find a transformation that takes a point in world space to this coordinate system, and store it with the sphere. In an object format like the one we use in Graphics II, it's easy, since we store transformations in the graphics

state to get from world space to the current object's space anyway. We would simply store this matrix with the object in the tree, and use that transformation on the world-space intersection point to get its location in object space. That way, when the sphere is rotated, for example, its texture rotates with it.

# Supplied code and resources:

## *Images*

A number of images that I created using a modified version of the solution ray caster are in pub/pix/texture under the main 463 directory. I recommend looking at them if you enjoyed this lecture, because, being ray traced images, they look much better in color.

## *Code*

I put a lot of the code I used to create the example images in pub/src/p3/texture under the main 463 directory. For your 'dirty' tiff for P3, you'll probably want a noise function, and this code contains a perfectly serviceable one, as well as a turbulence function that uses it.

# Credit where credit is due

## *Peachey and Perlin*

Darwin Peachey and Ken Perlin were the inventors of solid texturing. While they did not work together, they discovered it simultaneously in 1985. The noise function that I use is descended from Ken Perlin's original 1985 noise function. Without their work, or something very like it, you would not be hearing this lecture today.

A really wonderful book (which I've loaned to a friend and can't find for the life of me) that also is the definitive reference on this topic is "Texturing and Modelling: A Procedural Approach". It's an excellent reference, and is written by Peachey, Perlin, Worley, and Musgrave. Most of it is written in RenderMan shading language, but it converts to C in a really straightforward way. I highly recommend it if you decide you like solid texturing.

## *Watkins, Coy and Finlay*

Much of the material I'm using is taken from the book "Photorealism and Ray Tracing in C", by Christopher Watkins, Stephen Coy, and Mark Finlay. It was my first book on ray tracing, and I can recommend it in good conscience to anyone with a good tolerance for DOS and VGA programming. It goes into a great deal of detail and assumes you have no system tools to work with, because that is the case in DOS.

From their book, I adapted the noise function given to you. Also, the introductory paragraphs on exactly what solid texture is are quoted almost verbatim from it.