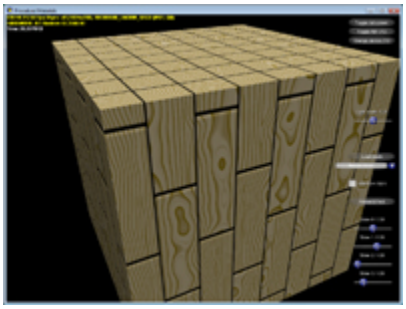
ProceduralMaterials Sample

Collapse All



Path

Source	*SDK root* \Samples\C++\Direct3D10\ProceduralMaterials
Executable	*SDK root* \Samples\C++\Direct3D10\Bin\ *x86 or x64* \ProceduralMaterials.exe

How the Sample Works

This sample uses simple repetitive functions as building blocks to create procedural materials. Procedural textures are resolution independent, exist everywhere in 3D, and do not require UV mapping coordinates. By combining octaves of noise, grid functions, and other specially crafted pieces of code, this sample aims to show that many different types of materials can be created without the use of textures.

Noise

Two types of noise are used in this sample to help construct the procedural textures. The first is simplex noise. This is a lattice-based noise function similar to Perlin noise. For more information see [3,5]. This noise does sample a small 256×256 texture of random values.

The second type of noise this sample uses is Voronoi noise. It is also called Worley or Cellular noise [1,2,4]. Depending on the parameters, this can give a very bubbly or even chip-like appearance. This noise does sample a small 1D texture of random vectors.

Other Functions

Noise functions aren't desirable for regular man-made patterns. Take, for example, a brick wall. There isn't a good noise function that produces the pattern of brick and mortar needed for such a surface. For that reason, we have added several other helper functions to the sample that simulate grids, staggered grids, regular arrays of spheres, and so on. These, combined with noise functions, can produce results that are resolution-independent, exist in 3D space, require no UV mapping, and simulate various real-world materials.

Using the Sample

The sample loads an effect file that contains various procedural effect techniques. By default, these are applied to a simple cube. The **Load Mesh** button can be used to load different pieces of geometry.

The technique drop-down control contains all of the techniques contained within the effect file.

Clicking the **Reload Effect** button reloads the effect file from disk. This button facilitates making changes to the effect file and viewing them immediately (after recompiling the effect) in the sample.

There are four generic sliders. These are passed in as shader constants to all techniques. The techniques can use these as they please. For example, most techniques will use Slider 3 to change the scaling of the mesh in the Y direction.

Getting Normals for Procedural Materials

Combining procedurals to create an albedo or diffuse texture is straightforward. Simple blending between

building blocks will give the right result. However, perturbing the normal using procedurals isn't as intuitive. Most real-time applications use a normal map to change the lighting across a surface. This normal map is a pre-perturbed surface that is fitted to the object through its UV coordinates. Procedural materials are meant to be dynamic. Therefore, there is no pre-perturbed normal surface to apply. Furthermore, since many of these techniques involve combining various procedurals in arbitrary ways, there is no analytic derivate that we can use to construct a normal for the surface.

One solution is to run the shader multiple times and compute the difference in height at each sample point. If we calculated the height one pixel to the right of the currently rasterized pixel and one pixel above the currently rasterized pixel, we could compute tangent and bitangent vectors to the central pixel. Doing a cross product on these would give us the normal for that point. Unfortunately, this is expensive, and it results in running the shader multiple times. Fortunately, the video hardware can do much of this operation for us. The ddx and ddy instructions compute the change of any value that is computed by the shader in screen space. For example, ddx computes the change of any value with respect to the horizontal. The code to handle this is as follows:

```
float3 FindNormal( float3 Value )
{
    float3 dWorldX = ddx(Value);
    float3 dWorldY = ddy(Value);
    float3 bumpNorm = normalize( cross( dWorldX, dWorldY ) );
    return bumpNorm;
}
```

While ddx and ddy are fast, they do have a limitation: They do not operate at the granularity of the screen pixel. Modern video hardware runs the pixel shader over groups of pixels called pixel quads. All pixels in a quad are computed at once. The ddx and ddy instructions compute the change in the variable over the pixel quad. Therefore, the normal is computed at a much lower resolution than the screen pixels. This can cause blocking artifacts. To test this, click the **Use DDX/DDY** button below the technique drop-down. The framerate will increase, but blocking artifacts will now be visible in techniques that use perturb the normal.

The solution that this sample uses by default is to render the perturbed heights of the objects in the scene into an off-screen render target. That render target is then read back in on another pass. For each pixel on the screen, its right and top neighbors are sampled. Tangent and bitangent vectors are created from the neighbors to the central pixel. A cross product between these will give the normal. This approach is a compromise between running the procedural shader multiple times at different sample points and using ddx/ddy. For each pixel on the screen, the procedural shader is only run once, and the normal calculation happens at the pixel level instead of the quad level. It does, however, require a separate render pass.

References

[1] Chan, Bryan, Michael McCool. "Worley Cellular Textures in Sh". *ACM SIGGRAPH 2004 Posters*, p.18, August 2004.

[2] Ebert, David S., F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley. *Texturing and Modeling: A Procedural Approach*, Academic Press Professional, Inc., San Diego, CA, 1994

[3] Gustavson, Stefan. "Simplex noise demystified". http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf, March 2005.

[4] Worley, Steven. "A cellular texture basis function," *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, p.291-294, August 1996.

[5] Tatarinov, Andrei. "Perlin Fire". http://developer.download.nvidia.com/whitepapers/2007/SDK10/PerlinFire.pdf, February 2007.