9.3

Real-Time Cellular Texturing

Andrew Griffiths, Deep Red Games

Introduction

Cellular texturing describes a type of procedural texture generation based on dividing space up into regions (cells). In 1996 Steve Worley presented a procedural cellular 3D texturing algorithm based on Voronoi cells [Worley96]. His algorithm, often called Worley Noise, is the most well-known implementation of cellular texturing. It has become very popular in the CG industry and is now a cornerstone of procedural basis functions, joining the ranks of Perlin [Perlin02] and Musgrave. One of the main features of the algorithm is its flexibility. It is a basis function with loads of room for experimentation and tweaking, resulting in a huge variety of possible outcomes. This type of texturing is very powerful, but it is also computationally expensive.

This article is the product of wanting to see what Worley noise would look like animated. It turned out that the traditional implementation of Worley noise wasn't suited for animation and was too slow. This article presents two methods for generating 2D Worley-based cellular imagery in real-time using the GPU. The first method is a general solution; the second is more compatible with older hardware and is slightly limited.

Background

Classic Worley noise is a general-purpose texture-basis function. It can be queried for any point in 3D space and will deterministically return a valid texture value. This article will concentrate on generating 2D cellular imagery based on the fundamentals of Worley noise. Luckily, the fundamentals are very simple.

The algorithm works with a set of randomly positioned 2D "feature" points as shown in Figure 9.3.1. For each pixel, the closest of these points is found; the distance to this point is called F_1. Next, the distance to the second closest point is found; this distance is called F_2 as shown in Figure 9.3.2. This can continue on to F_n, the distance to the nth closest point. This is the only data needed to generate cellular textures.

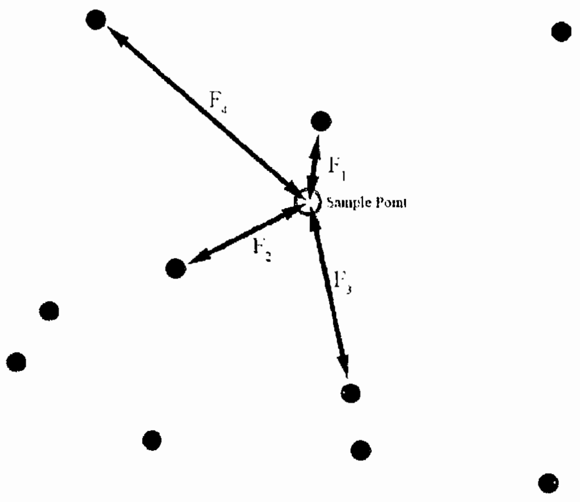

FIGURE 9.3.1 Randomly positioned 2D feature points. The first four feature points closest to the sample point are shown.

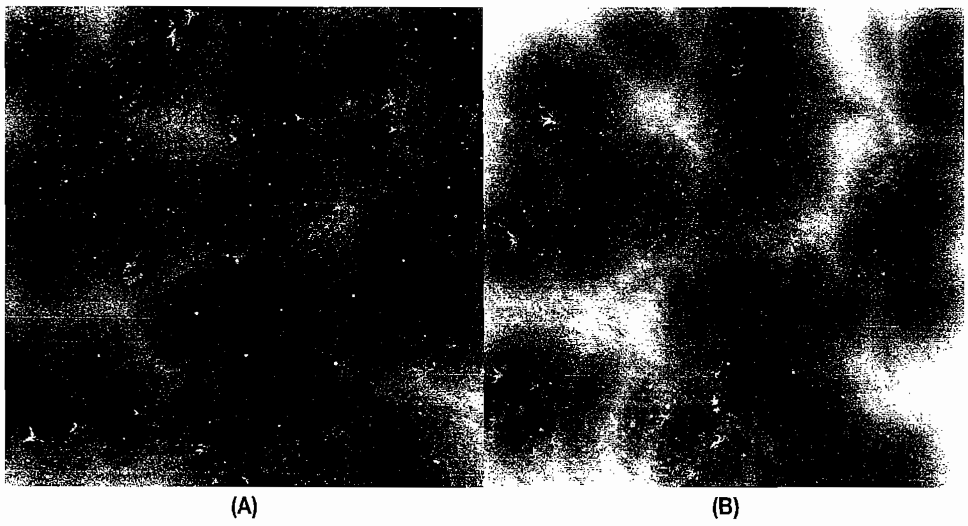

FIGURE 9.3.2 (A) Final F_1 image. (B) Final F_2 image.

The final step is completely open. It is up to the individual to decide how to use this distance data to create interesting images. Two of the most commonly used combinations are *flagstone* and *plated*. A flagstone effect (see Figure 9.3.3a) can be achieved by drawing the value of $F_2 - F_1$ per pixel. A plated effect (see Figure 9.3.3b) can be created by using the index of the closest (F_1) feature point to color each pixel. We will only focus on generating grayscale textures, but there is nothing to stop somebody from coming up with colorful variations. The value computed for each pixel can be scaled to bring it between the 0 and 1 range. Scaling and biasing the value is usually needed to bring values into a nice range, and these extra parameters are great for achieving different looks.

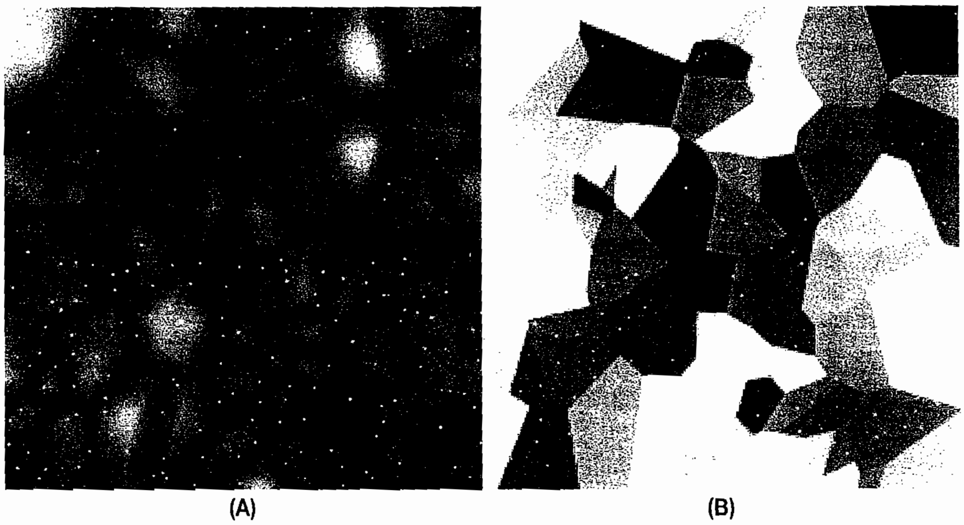

FIGURE 9.3.3 Distance data displayed in various ways. (A) Classic flagstone $F_2 - F_1$. (B) Classic plated texture.

The great thing is that it's easy to understand and implement this type of cellular imagery. The not-so-great thing is that it's very power hungry.

In a naive implementation each pixel that is sampled would have to do distance calculations to each feature point to find the *n*-closest point. A 640 × 480 image with 100 features would need to do 30 million distance calculations. Of course, most implementations use some sort of space segmentation, so the number of distance calculations is much lower than this, but it is still a sizable problem.

GPU Implementation

Our goal is to accelerate the generation of cellular texturing using the GPU. GPUs are massively parallel processors and have huge memory bandwidth compared to CPUs. Let's leverage these features to create some real-time cellular noise.

Generating F_1

For each pixel, F_1 is the distance to its closest feature point. Looking at an example F_1 image (Figure 9.3.4a), we can describe what appears to be happening.

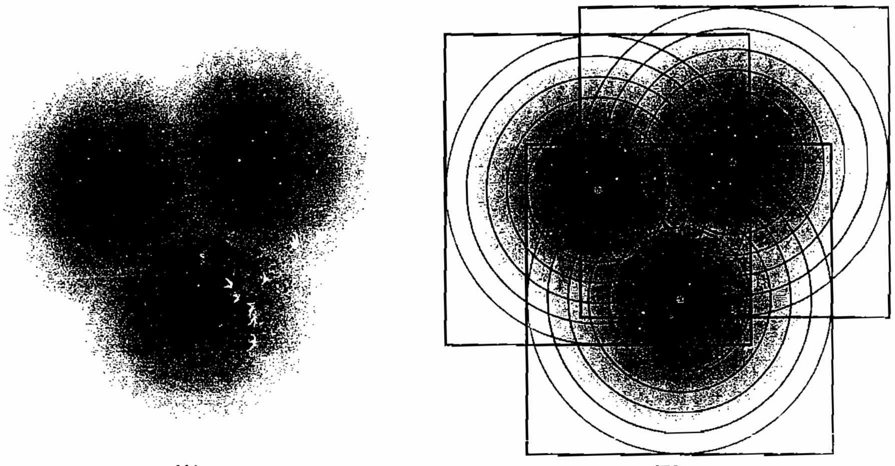

FIGURE 9.3.4 Three feature points. (**A**) The resulting F_1 image; the pixels are colored by distance to the closest point. (**B**) This can be represented as overlapping circles on each feature point.

Imagine if we render circles centered on each feature point, and if each pixel of a circle is colored based on the distance to its feature point. Each pixel on the resulting image is being assigned a distance to one of the feature points, but we need to make sure that only the distance to the closest (F_1) feature point is left. This can be achieved through a simple comparison before writing the pixel: "Is my value less than the value already at this pixel?"

The GPU's depth buffer is custom built for this type of rejection test and can do it extremely quickly. For each pixel of the circle, write the distance between that pixel and its circle's center pixel position to the depth buffer. With the depth buffer set to the traditional LESSTHAN operation mode, the pixel will only be allowed to pass if the depth value is less than what already exists in the depth buffer for that pixel. This method of using the depth buffer to compute F_1 has been used before [Hoff97] [Warden05].

As shown in Figure 9.3.5, this implementation uses simple rectangles (quads) instead of circles for each feature point, but any 2D shape would do.

The vertex shader outputs the position of the center of the quad and the position of the corner vertex. These values are interpolated across the triangles and are sent to the pixel shader, where they are used to calculate the distance of that pixel from the center of the quad. This process is shown in Figure 9.3.6.

LISTING 9.3.1 HLSL Code for the Vertex and Pixel Shader

```
void
CellularVS_Rect(in float4 center : POSITION,
                in float3 offset : TEXCOORD0,
                out float4 oPos : POSITION,
                    out float4 oDist : TEXCOORD0)
{
    oPos = center + ( float4(offset.xy,0,0) * rectSize );
    oDist = float4( oPos.xy, center.xy );
}

void
CellularPS_Linear(in float4 dist : TEXCOORD0,
                  out float oDepth : DEPTH,
                  out float4 oColor : COLOR)
{
    float d = distance( dist.xy, dist.zw );
    oDepth=d;
    oColor=d;
}
```

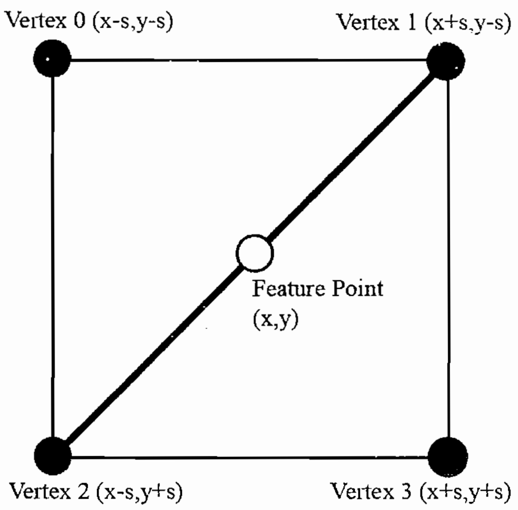

FIGURE 9.3.5 Quad generated around a feature point.

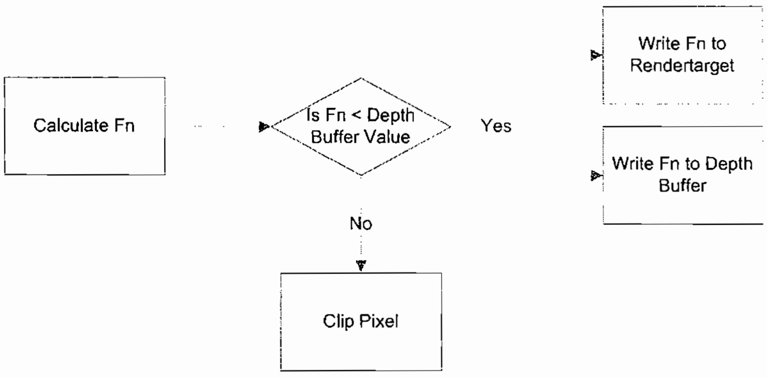

FIGURE 9.3.6 Flow of pixel operations to generate F_1.

If we do this for each feature point, and make sure the quads are large enough to overlap, we end up with an image of the closest feature points (Figure 9.3.2a).

Using Alpha-Blending

As well as using the depth buffer comparison, alpha-blending can also be used to pick the smaller of two values. Setting alpha-blend mode to `MIN` and source and dest to `ONE` results in the following pixel blending operation:

```
FinalColor = Min(SourceColor * (1,1,1,1), PixelColor * (1,1,1,1))
```

This post–pixel shader operation is slower than the depth buffer comparison, as the frame buffer has to be read, and the pipeline isn't as highly optimized as depth-buffer reading. We'll come back to this later to show where it can be used in cellular texturing.

F_2 and Beyond

F_n is the distance to the nth closest point and is trickier to calculate than F_1, but luckily not too much. For F_n above F_1, the resulting distances in the image still have to be the smallest distances to a feature point, but they must always be larger than F_{n-1} (since $F_1 < F_2 < F_3 \ldots F_n < F_{n+1}$). If we have already computed F_{n-1}, we can evaluate each pixel with our computed distance values to make sure it is larger than the distance computed for that pixel at F_{n-1}. If it isn't greater than the distance value in the $Fn - 1$ texture, this pixel can be rejected. Otherwise, it is passed down to the depth buffer test as in the previous section.

As shown in Figure 9.3.7, this can be repeated to create images of F_2, F_3, F_4, and so on, but after F_3 the images tend to look very similar and aren't that useful for creating different types of textures.

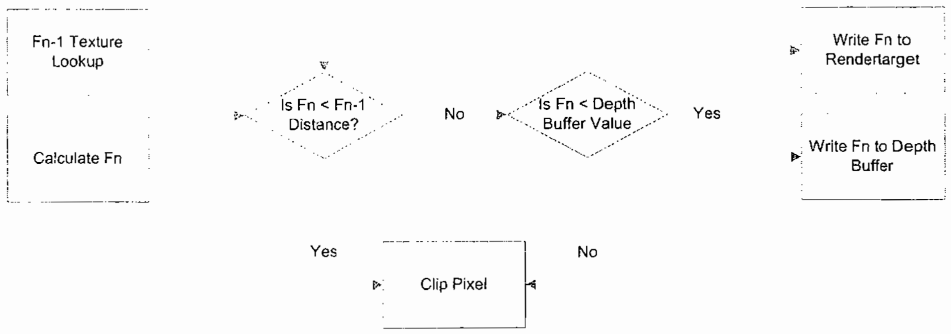

FIGURE 9.3.7 Flow of pixel operations to generate F_n above 1.

Notes

A summary of the algorithm follows:

- Pass for initial F_1:
  1. Set rendertarget to F_1 texture.
  2. Clear depth buffer to 1.0.
  3. Clear rendertarget to 1,1,1,1.
  4. Render rectangles for each feature point.
  5. For each pixel, compute the distance between it and the center of its quad.
  6. Use the depth buffer to test that this distance is less than the depth-buffer value. If it passes, write this distance to the depth buffer and to the render target.
- Passes for F_n where $n > 1$:
  1. Set rendertarget to F_n texture.
  2. Clear depth buffer to 1.0.
  3. Render rectangles for each feature point.
  4. For each pixel, compute the distance between it and the center of its quad.
  5. If this value is less than the corresponding distance value from the F_{n-1} texture, reject this pixel.
  6. Use the depth buffer to test that this distance is less than the depth-buffer value. If it passes, write this distance to the depth buffer and to the render target.
- Combiner pass:
  1. Set rendertarget to frame buffer.
  2. Draw full-screen quad.
  3. For each pixel, blend the previous F_n images in some creative manner.

Because the distance values have a wide range, a lot of precision is needed to avoid artifacts. A 24-bit depth buffer and floating-point textures are ideal.

The size of the quads is also very important; if the quads are not large enough, then discontinuity artifacts will become visible. However, the larger the quads, the more fill rate is used, so it's important to get the balance just right.

A handy note is that for randomly distributed points, the more points you have, the closer they will be, and thus the smaller the rectangles can be. With 10,000 points on screen, the rectangle only needs to be a few pixels wide to find its n closest feature points, so this technique scales well with a large number of points. Also, since $F_n < F_{n+1}$, larger quads are needed for each iteration. Supporting different-sized quads for each iteration can further reduce fill-rate consumption.

Different shapes such as circles could be used instead of quads to cover the surface area more optimally (depending on the distance function used). This technique puts a massive strain on the end of the graphics pipeline and burns a lot of the GPU's fill rate, as it relies on overlapping quads, resulting in massive overdraw. The distance calculation could be moved to the vertex shader, thus alleviating some of the per-pixel burden, but this would impact the quality of the results.

Even more speed can be achieved in some cases by rendering each feature point as a 3D cone [Hoff97]. Creating the cone geometry on the CPU or in the vertex shader tells the GPU that the depth values aren't going to be changing in the pixel shader and can operate much faster. The expensive distance calculations are also not done per-pixel. This can be a massive speed boost; however, it works best for linear or squared distance calculations and makes the technique quite cumbersome for using different distance formulae.

F_1 and F_2 Without Floating Point Buffers

The above method requires high-precision floating-point render targets to produce a useable and artifact-free image. Floating-point surfaces are great, but they do have some limitations. First, they are not as widely supported as integer buffers. Some cards only support lower-precision 16-bit buffers, and the support for different numbers of channels varies too. Second, they tend to be slower than integer buffers and take up much more memory. Taking all this into account, it would be useful to find a way to render cellular textures on the GPU without relying on floating point buffers.

The following techniques achieve the rendering of F_1 and F_2 using 8-bit surfaces, and do not suffer from artifacts, as they use the 24-bit depth buffer to do all of the comparisons. The main limitation is that F_3 can't be generated elegantly after F_2; however there is plenty of room for interesting effect combinations using just these two images.

The F_1 Pass

The F_1 pass is generated the same way as before, but to an integer buffer (e.g., R8G8B8A8). We render the rectangles for each point, calculating the distance from the point per-pixel and writing this distance to the frame and depth buffers if the depth test passes. After this pass our depth buffer is left filled with the F_1 shortest distance values.

1. Set 8-bit rendertarget.
2. Clear rendertarget (clearing to white is good to hide artifacts).
3. Clear depth buffer to 1.0.
4. Render quads for each point, using render states:
5. `ZEnable = TRUE`
6. `ZWriteEnable = TRUE`
7. `ZFunc = LESS`
8. `AlphaBlendEnable = FALSE`

The F_2 Pass

For F_2 we need to compute the second shortest distance. We need to make sure F_2 is larger than F_1 but also that it's the smallest of all the values tested.

1. Set 8-bit rendertarget.
2. Clear rendertarget.
3. Render quads for each point, using render states:
4. `ZEnable = TRUE`
5. `ZWriteEnable = FALSE`
6. `ZFunc = GREATER`
7. `AlphaBlendEnable = TRUE`
8. `BlendOp = MIN`
9. `SrcBlend=ONE`
10. `DestBlend=ONE`

The depth-buffer test ensures that the pixel that is written has a larger distance value than what is already in the depth buffer (F_1). Values that are higher will pass the depth test and will have their distance value written into the color buffer. This makes sure our distance is greater than F_1, but we will need to make sure it is the smallest remaining distance.

The alpha-blend picks the smallest value of the frame buffer and the pixel that has just passed the depth-buffer test. This leaves us with the second-smallest distance in the color buffer.

Depth writing is turned off on the F_2 pass. If it had been turned on, a depth value greater than F_1 could have been written to the depth buffer, but it wouldn't necessarily be the second-shortest distance (F_2), and this would stop smaller distances from passing.

Increased precision can also be achieved by encoding a float across the 8-bit RGBA channels. This requires some extra computation for packing and unpacking the value on reads and writes [NVFog04] [Baker01].

Tiling

The ability to generate seamless tileable textures is one of the most popular features of procedural texture generation, and cellular texturing is no exception. In the classic algorithms the distance function is modified to wrap around the image area [Scott01]:

LISTING 9.3.2 How the Distance Function Would Usually Be Modified to Generate Wrapped Distance Values

```
float
DistanceWrap(float2 a,float2 b)
{
    float dx = abs(a.x-b.x);
    float dy = abs(a.y-b.y);
    if (dx > width/2)
        dx = width-dx;
    if (dy > height/2)
        dy = height-dy;
    return sqrt( dx*dx + dy*dy );
}
```

Our image-based rendering approach can't use a wrapped distance function.

We create a seamless texture by detecting on the CPU which quads are overlapping the image boundary and render them again at the opposite location. This means we have to do four render calls. It's not very elegant but it works.

If GPUs were able to wrap geometry instead of clipping it, this type of seamless texturing could be achieved very easily and quickly.

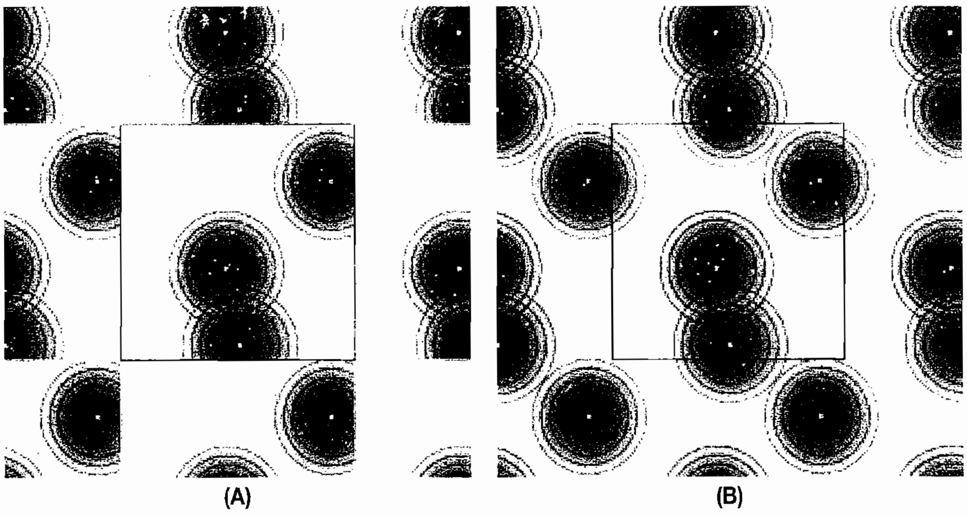

FIGURE 9.3.8 (**A**) A nontileable texture. (**B**) The same texture, but seamless.

Application

There are countless ways in which cellular texture can be used and combined to form interesting imagery. One of the great things about this type of texture generation is how much scope there is to customize and extend it.

Distance Functions

There is no need to only use the standard Euclidean distance function when plenty of others exist. Simply replacing the distance calculation can completely change the resulting style of texture. Some distance functions to try are:

Distance squared: *F*(*a*,*b*) = (*dx***dx*) + (*dy***dy*)
Manhattan distance: *F*(*a*,*b*) = *abs*(*dx*) + *abs*(*dy*)
Chessboard distance: *F*(*a*,*b*) = *max*(*dx*,*dy*)
Minkovsky distance: *F*(*a*,*b*) = (*dx*^*e* + *dy*^*e*)^(1/*e*)

You can also use a texture as a distance function lookup and create your own bizarre metrics. Usually though, the image won't look good unless the distance value roughly increases, the further you get from the feature point.

Combinations

Figures 9.3.9 through 9.3.12 show some of the many interesting ways to combine the distance images:

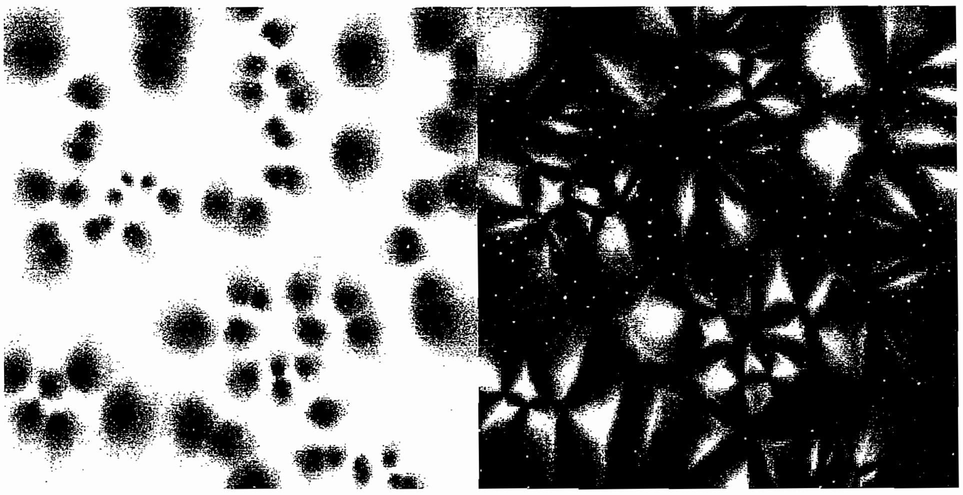

FIGURE 9.3.9 (**A**) (f1/f3). (**B**) smoothstep(f1,f3*2,f2).

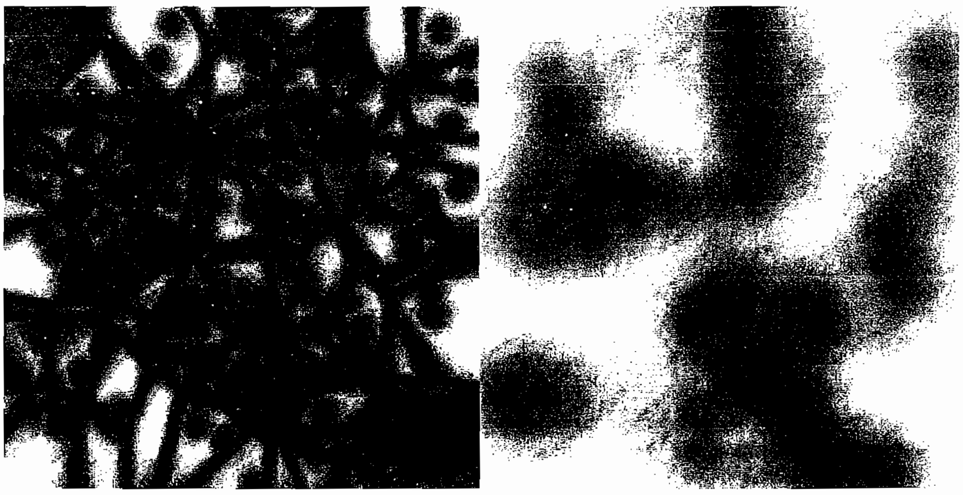

FIGURE 9.3.10 (**A**) min(f1,f3-f2). (**B**) length(float3(f1,f2,f3)).

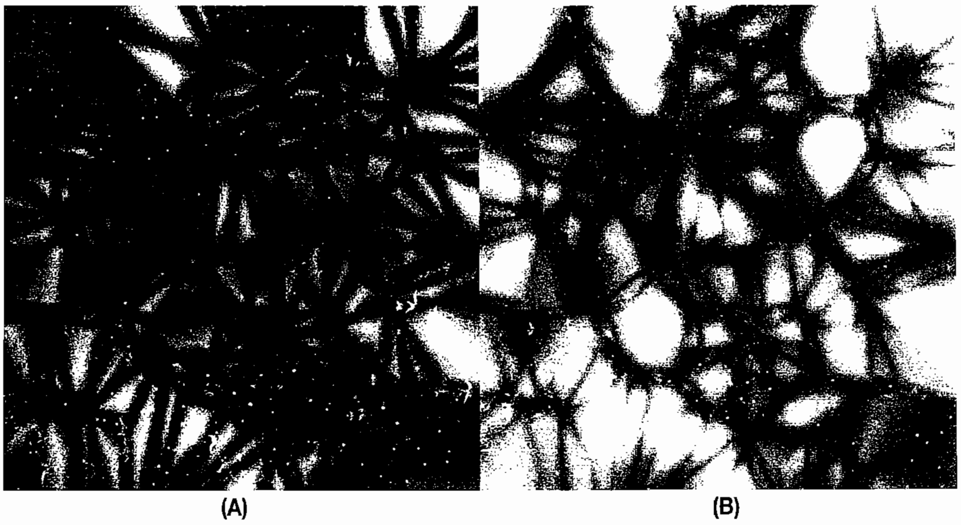

FIGURE 9.3.11 (A) min(f2-f1,f3-f2). (B) max(f2-f1,f3-f2).

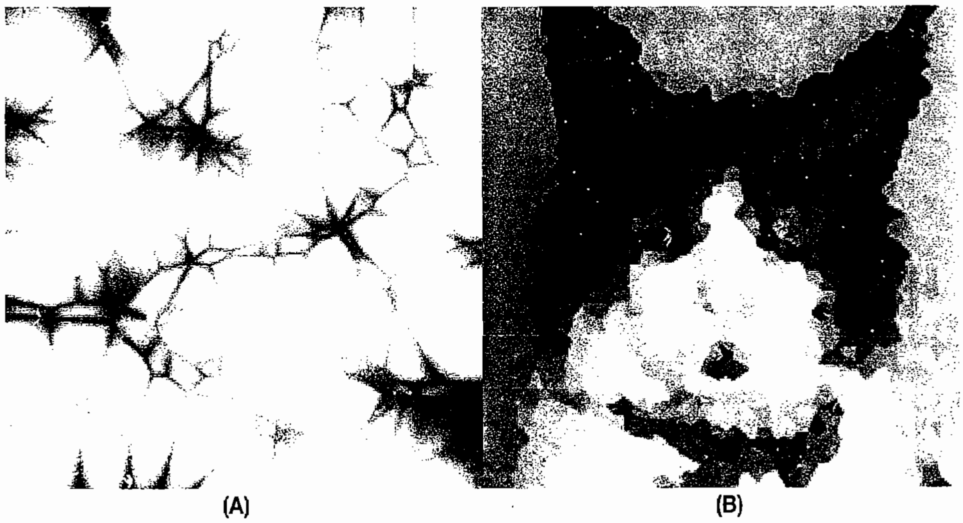

FIGURE 9.3.12 (A) abs(f1-f3/f2). (B) Mosaic of Pula.

Index and Mosaic

By rendering the index number of the feature point that each pixel is closest to, you can create an interesting indexed image (see Figure 9.3.3b). This looks exactly like a Voronoi diagram and effectively achieves the same effect by segmenting the image into regions (cells), where every pixel in a region is closer to a particular feature point. Extending this to sample a color for each cell from a texture can create a mosaic effect (see Figure 9.3.12b).

Other

There are many other possible applications for this type of pattern generation:

- Repeatedly rendering smaller cells inside larger cells at higher frequencies gives some interesting fractal effects [Shirriff98].
- Cellular textures have been used to synthesize cracks [Mould05].
- Using an animated F_1 fractal image converted into a bump map is a popular way of simulating the surface of the ocean.
- Cellular texture can be used in NPR (non-photorealistic rendering).
- Careful distribution and coloring of the feature points can create interesting patterns [Kaplan00].

Examples

ON THE CD

On the companion CD-ROM you will find several programs (source and binaries) that demonstrate different aspects of this technique. There are also screenshots and a video for those who don't have compatible hardware.

The examples require a Windows PC with DirectX9.0c and a pixel shader 2.0 graphics card. They use the Direct3D sample framework, and the shaders have been written in HLSL.

You can also download the examples from *http://www.rawhed.com/shaderx5/*.

Conclusion

This article has presented a technique to produce images based on classic 2D Worley cellular texturing, using the GPU to accelerate the process to interactive rates. This can allow for rapid experimentation with immediate user feedback when working with this type of procedural texture. The image-based approach and use of a shading language allows for many different derivations from the typical results. A weakness of this technique is that artifacts appear when the rectangles are not large enough, but using large rectangles uses more bandwidth, making it run slower at higher resolutions.

There is still a huge amount of room for investigation and improvement. With the imminent arrival of DirectX 10 Shader Model 4.0 and geometry shaders, it will be interesting to see what can be done with cellular texturing on the GPU in the future.

References and Further Reading

[Baker01] Baker, Dan. "Volumetric Rendering in Real-time." Available online at *http://www.gamasutra.com/features/20011003/boyd_04.htm.*

[Ebert03] Ebert, David et al. "Texturing & Modelling: A Procedural Approach." Morgan Kaufmann Publishers, 2003 *(http://www.texturingandmodeling.com/)*.

[Hoff97] Hoff, Kenneth et al. "Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware." Available online at *http://www.cs.unc.edu/~geom/voronoi//.*

[Kaplan00] Kaplan, Chris. "Voronoi Art." Available online at *http://www.cgl.uwaterloo.ca/~csk/projects/voronoi/.*

[McCombs05] McCombs, Shea. "Introduction to Procedural Textures." Available online at *http://www.newcottage.com/index.php?section=tutorials&subsection=tutorials/3d_procedural4.*

[Mould05] Mould, David. "Image-Guided Fracture." Available online at *http://www.cs.usask.ca/~mould/papers/crack.pdf.*

[NVFog04] NVIDIA. "Fog Polygon Volumes." Available online at *http://download.nvidia.com/developer/SDK/Individual_Samples/DEMOS/Direct3D9/src/FogPolygonVolumes3/docs/FogPolygonVolumes3.pdf.*

[Scott01] Scott, Jim. "Making Cellular Textures." Available online at *http://www.blackpawn.com/texts/cellular/default.html.*

[Perlin02] Perlin, Ken. "Improving Noise." Available online at *http://mrl.nyu.edu/~perlin/noise/.*

[Shirriff98] Shirriff, Ken. "Fractals from Voronoi Diagrams." Available online at *http://www.righto.com/fractals/vor.html.*

[Warden05] Warden, Peter. "Animating Worley Noise." Available online at *http://petewarden.com/notes/archives/2005/05/testing.html.*

[Worley96] Worley, Steven. "A Cellular Texture Basis Function." *Proceedings of SIGGRAPH '96, Computer Graphics Proceedings*, Annual Conference Series, New Orleans, Louisiana, pp. 291–94.

Zaid Mian's and Bryan Chan's implementation of Worley noise using SH. Available online at *http://libsh.org/shaders/.*

Matt Pharr's implementation of Worley noise in C. Available online at *http://pharr.org/matt/code/wnoise.cpp.*