CS 563 Advanced Topics in Computer Graphics
Noise

by Dmitriy Janaliyev

Outline

- Introduction
- Noise functions
- Perlin noise
- Random Polka Dots
- Spectral synthesis
- Fractional Brownian Motion function
- Turbulence function
- Bumpy and Wrinkled textures
- Windy waves
- Marble
- Worley noise

Introduction

- It is often desirable to introduce controlled variation to a process
- Difficulty with procedural textures: we can’t just use functions such as RandomFloat()
- The problem is addressed with: **Noise functions**

Noise functions

- General representation:
  - $R^n \to [-1,1]$

  for n = 1, 2, 3,... without obvious repetition

- Basic properties:
  - Should be band-limited (avoid higher frequencies that are not allowed by Nyquist limit)
  - Exclude obvious repetition of returned values

Noise functions

- Implementation
  - Based on the idea of an integer lattice over 3D space
  - A value is associated with each integer (x,y,z) position of the lattice
  - Given an arbitrary point in that space the 8 adjoining lattice values are found and interpolated
  - Result is a noise value for that particular point
- Example
  - *Value noise* function

Noise functions

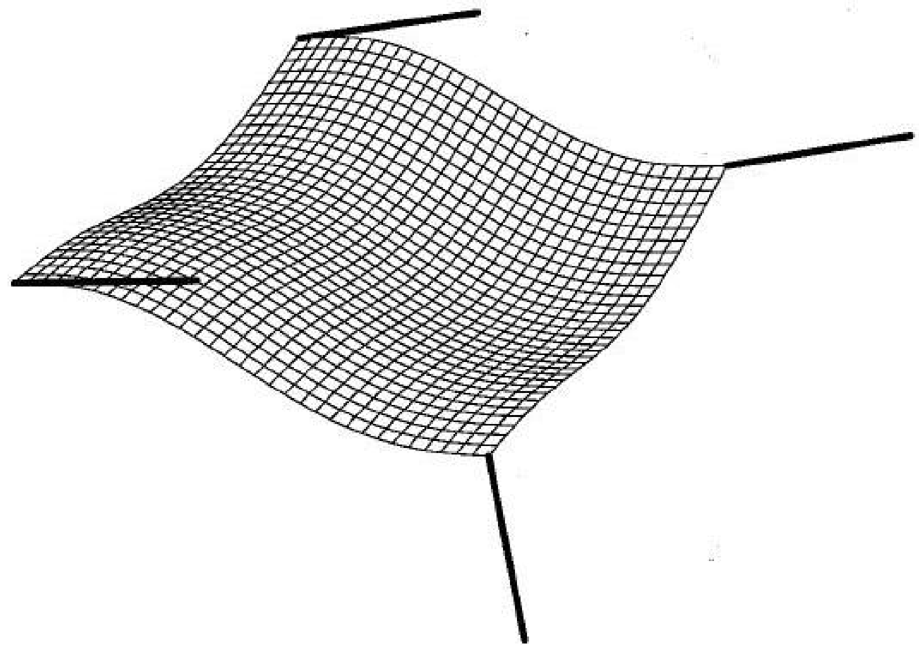
- Implementation issues
  - Noise function should associate an integer lattice point with the same value every time it is called
  - It is not practical to store values for all lattice points – some mapping mechanism is required
  - Hash function can be used with lattice points to look up parameters from a fixed-size table with precomputed pseudorandom values

  - The idea of the lattice can be generalized to more or less than 3 dimensions

Perlin Noise

- Basic description:
  - A noise function introduced by Ken Perlin in 1985
  - Has value of 0 at all (x, y, z) integer lattice points
  - Variation comes from gradient vectors that are associated with each point
- Advantages:
  - Computationally efficient
  - Easy to implement

Perlin Noise



2D slice of noise function with four gradient vectors (scanned from the pbrt book)

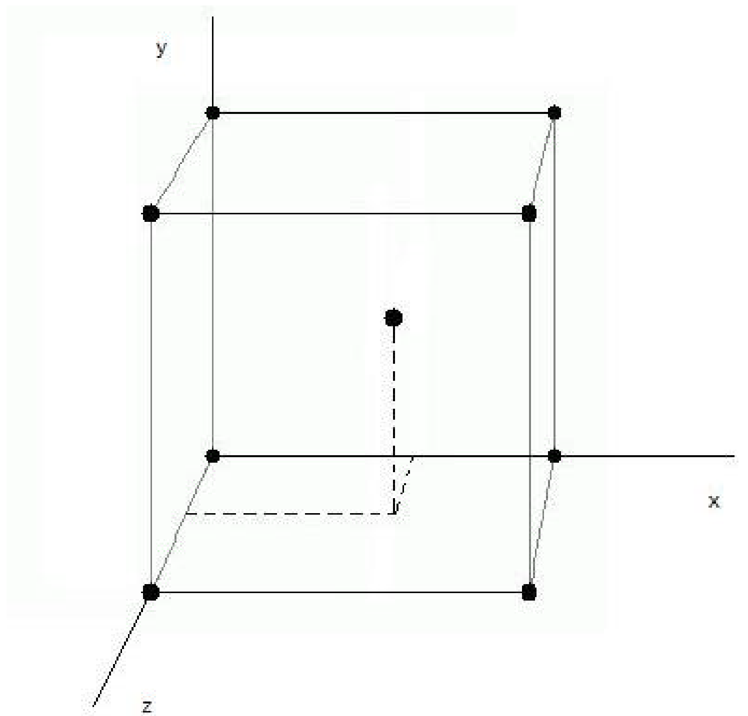
Perlin Noise

- Implementation overview

```
float Noise(float x, float y, float z){
    <Compute noise cell coordinates and offsets>
    <Compute gradient weights>
    <Compute trilinear interpolation of weights>
}
```

```
<Compute noise cell coordinates and offsets>
    int ix = Floor2Int(x)
    ...
    float dx = x – ix, dy = y – iy, dz = z - iz
```

Perlin Noise



Offsets of the real valued point from the origin of the cell

Perlin Noise

<Compute gradient weights>

```
ix &= (NOISE_PERM_SIZE – 1)
...
float w000 = Grad(ix, iy, iz, dx, dy, dz);
float w100 = Grad(ix+1, iy, iz, dx-1, dy, dz);
float w010 = Grad(ix, iy+1, iz, dx, dy-1, dz);
...
```

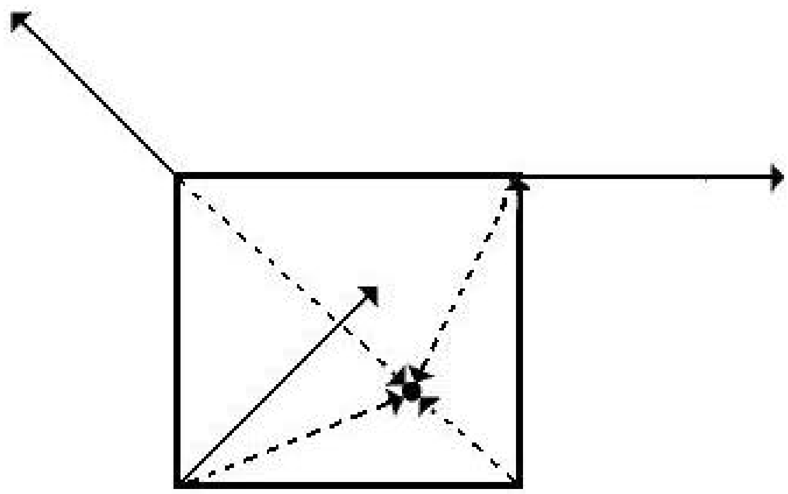
- NoisePerm[NOISE_PERM_SIZE*2] - a fixed-size table with precomputed values

- Indexing into NoisePerm:

  NoisePerm[NoisePerm[NoisePerm[ix] + iy] + iz]

  (rather than NoisePerm[ix + iy + iz] for instance)

Perlin Noise



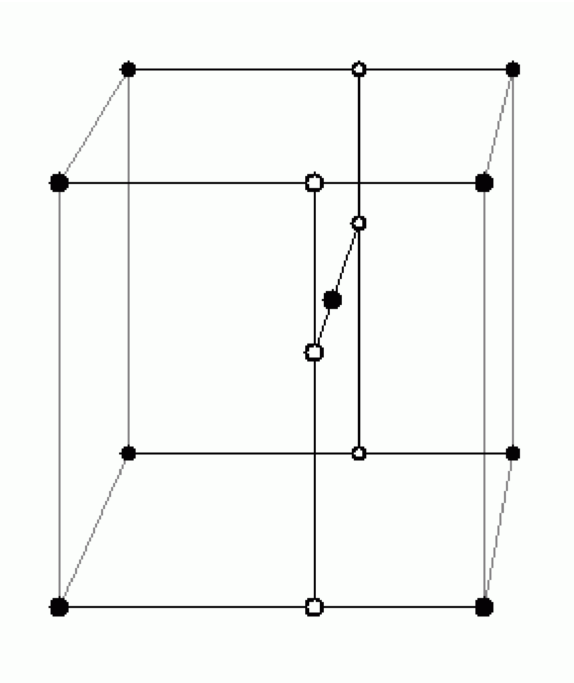
Dot product of vectors from the corners of the cell to the lookup point with gradient vectors gives the influence of each gradient to the noise value at that point (from the pbrt book)

Perlin Noise

```
<Compute trilinear interpolation of weights>
    float wx = NoiseWeight(dx);
    ...
    float x00 = Lerp(wx, w000, w100);
    float x10 = Lerp(wx, w010, w110);
    ...
    float y0 = Lerp(wy, x00, x10);
    float y1 = Lerp(wy, x01, x11);
    return Lerp(wz, y0, y1);
```

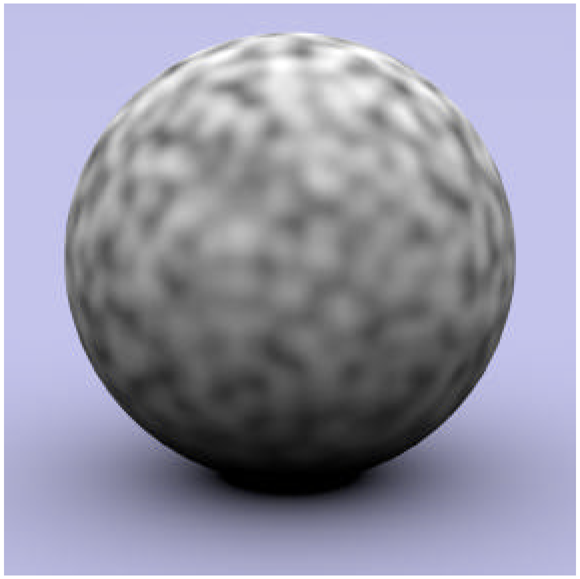
float NoiseWeight(float t) – smoothing function

Perlin Noise



Trilinear interpolation of adjacent points using Lerp() in 7 steps

Perlin Noise



companion CD-ROM of the pbrt book

http://freespace.virgin.net/hugo.elias/models/m_perlin.htm

Perlin Noise

http://escience.anu.edu.au

[Ken Perlin's demo]

[some more noise clouds]

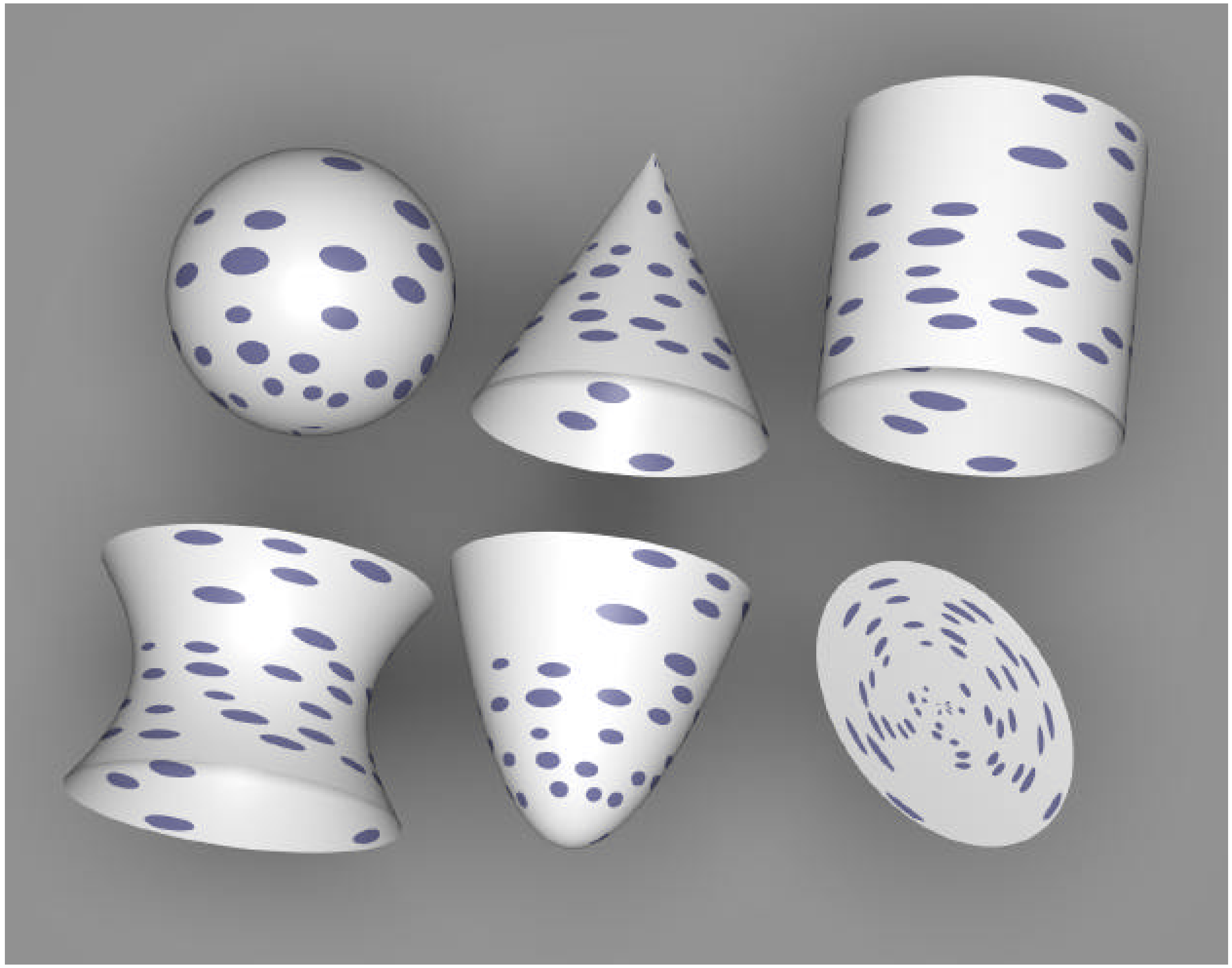
http://paintdotnet.12.forumer.com/viewtopic.php?t=1971

Random Polka Dots

- (s, t) texture space is divided into cells
- Each cell has a 50% chance of a dot inside of it
- Dots are randomly placed inside their cells
- Both dots and “empty space” are represented by textures

- Presence or absence of a dot at a particular cell is defined by Noise() function
- Noise() function is also used to specify the offset of the dot from the center of the cell

Random Polka Dots



Polka Dots texture applied to pbrt's Quadric Shapes
(from companion CD-ROM of the pbrt book)

Spectral Synthesis

- The fact that the noise function is a band-limited allows to create a noise function with a desired rate of variation
- Spectral synthesis – representation of a complex function $f_s(s)$ by a sum of weighted contributions from another function f(x):

$$f_s(x) = \sum_i w_i f(s_i x)$$

- Parameter scales s_i are generally chosen in a geometric progression, such that $s_i = 2s_{i-1}$ and weights $w_i = w_{i-1}/2$
- Each term in the summation is called *octave* of noise

Fractional Brownian Motion

- When spectral synthesis is used with Perlin noise the result is referred to as Fractional Brownian motion (FBm)
- Advantages:
  - Allows to vary level of noise returned by the function
  - Easy to compute and implement
  - Well-defined frequency content
- Implementation:

```
float FBm(const Point& P, const Vector& dpdx, const Vector& dpdy, float omega, int maxOctaves){
        <Compute number of octaves for antialiased FBm>
        <Compute sum of octaves of noise for FBm>
        return sum;
}
```

Fractional Brownian Motion

- Antialiasing the FBm function is based on *clamping* – ignoring those components of the summation that have frequencies beyond Nyquist limit and using average values instead
- Maximum frequency content for the Noise() function is $w = 1$
- Given a sampling rate *s* we need to find number of terms *n* such that:

$$2^n s = 2w = 2$$

$$2^{n-1} = \frac{1}{s}$$

$$n - 1 = \log\left(\frac{1}{s}\right)$$

$$n = 1 - \frac{1}{2}\log(s^2)$$

Fractional Brownian Motion

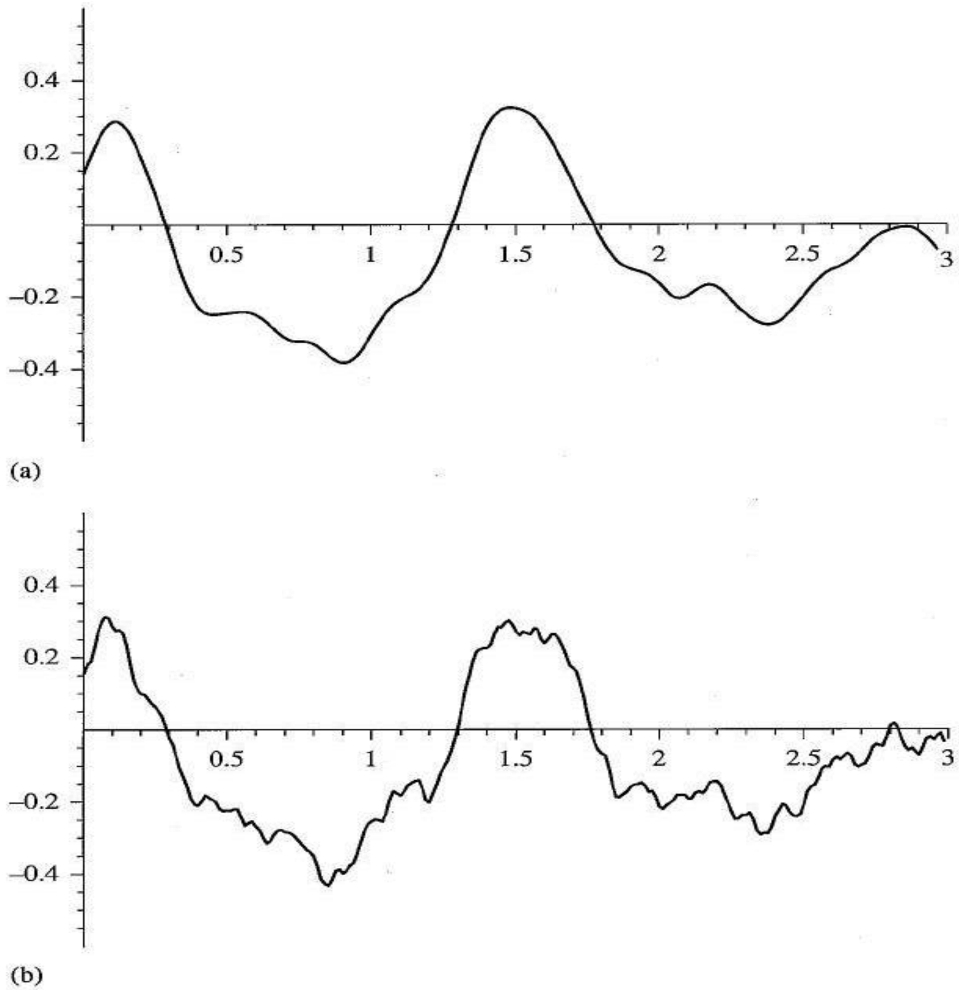
<Compute number of octaves for antialiased FBm>

```
float foctaves = min(maxOctaves, 1 – 0.5*Log2(s*s));
int octaves = Floor2Int(foctaves);
```

<Compute sum of octaves of noise for FBm>

```
float sum = 0., lambda = 1., o = 1.;
for( int i = 0; i < octaves; i++ ){
        sum += o*Noise(lambda*P);
        lambda *= 1.99;
        o *= omega;
}
float partialOct = foctaves – octaves;
sum += o*SmoothStep(0.3, 0.7, partialOct)*Noise(lambda*P);
```

Fractional Brownian Motion



Graphs of the FBm functions with 2 and 6 octaves of noise respectively (from the pbrt book)

Turbulence function

- Similar to FBm, but uses absolute values of the Noise function:

$$f_s(x) = \sum_i w_i \left| f(s_i x) \right|$$

- Taking absolute values introduces first-derivative discontinuities in the resulting function which leads to infinitely high frequency content and makes antialiasing techniques not that effective

Turbulence function

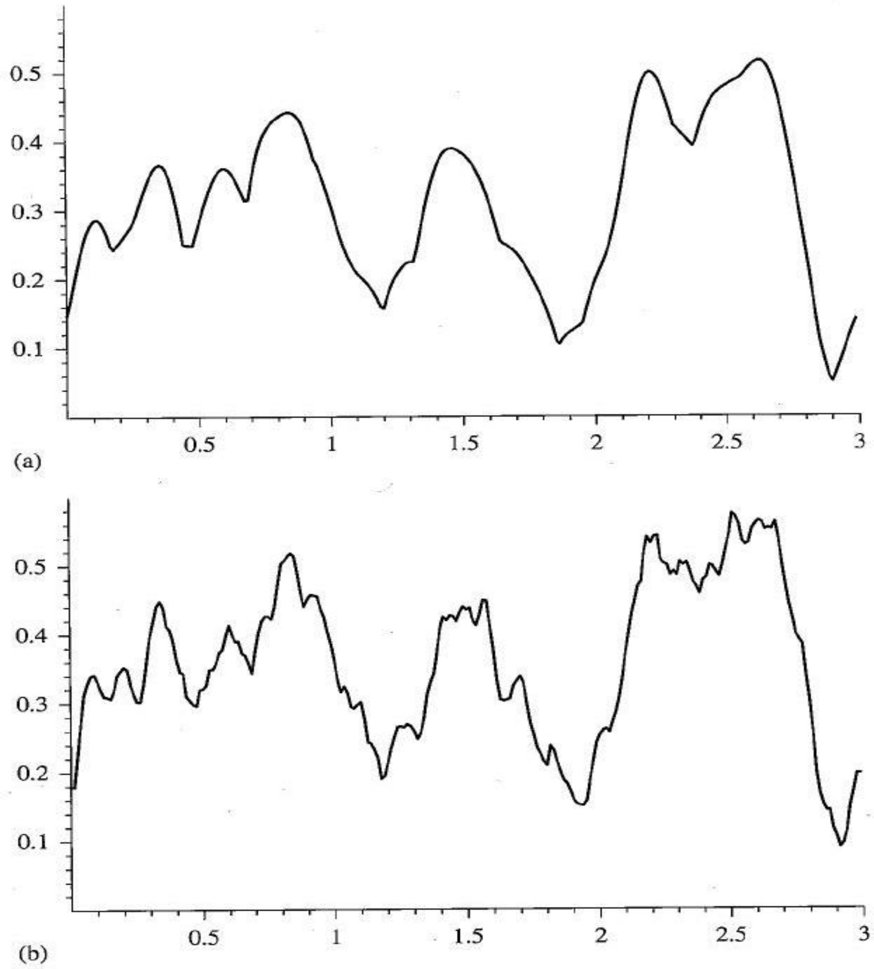
- Implementation

  same as FBm's but taking absolute values of Noise()

<Compute sum of octaves of noise for turbulence>

```
float sum = 0., lambda = 1., o = 1.;
for( int i = 0; i < octaves; i++ ){
    sum += o*fabs(Noise(lambda*P));
    lambda *= 1.99;
    o *= omega;
}
float partialOct = foctaves – octaves;
sum +=
o*SmoothStep(0.3,0.7,partialOct)*fabs(Noise(lambda*P));
```

Turbulence function

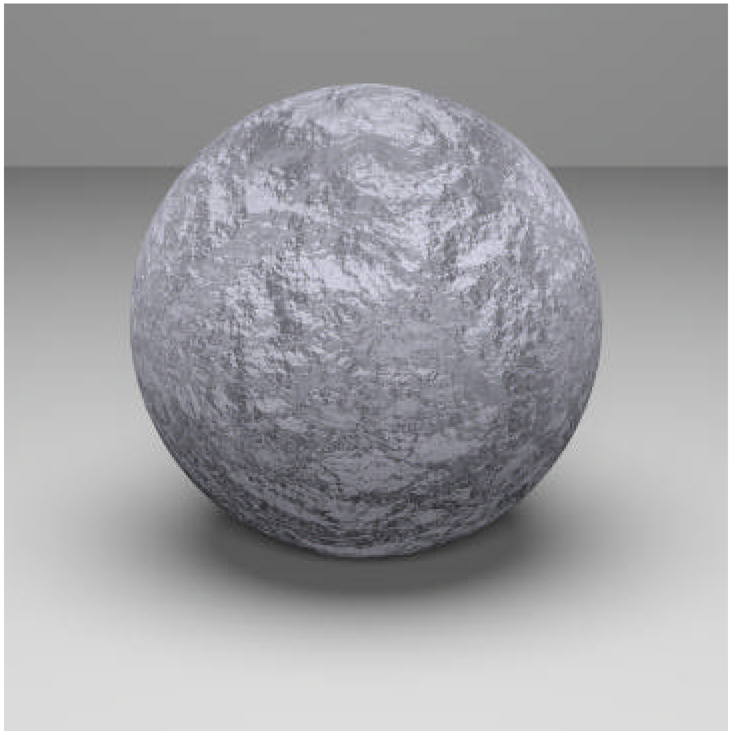


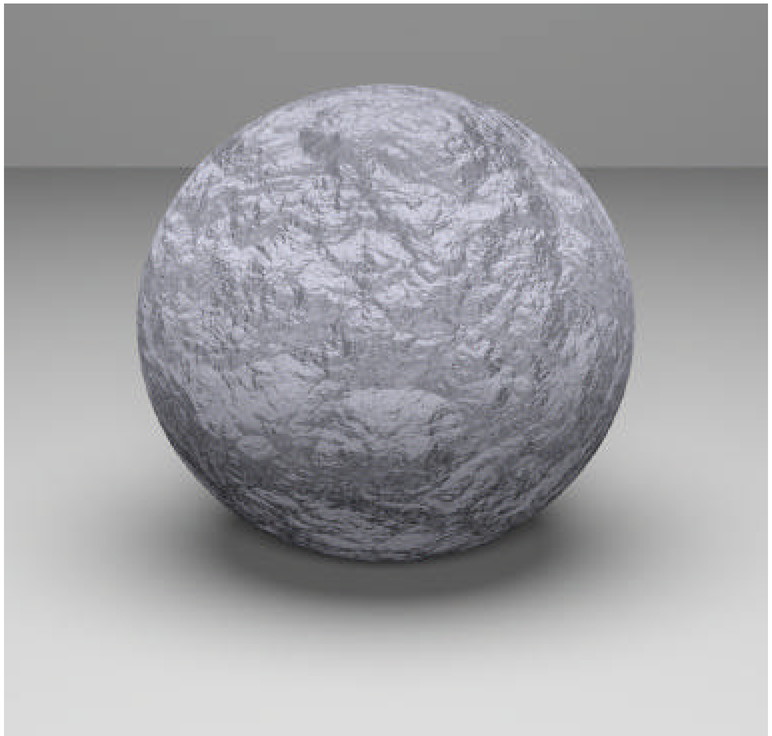
Graphs of the Turbulence functions with 2 and 6 octaves of noise respectively (from the pbrt book)

Bumpy and Wrinkled textures

- The FBm and Turbulence functions can be used to compute offsets for bump maps
- In PBRT FBmTexture uses FBm for bump mapping and WrinkledTexture uses Turbulence for the same purposes

Bumpy and Wrinkled textures





FBmTexture and WrinkledTexture used for bump mapping of a sphere (from companion CD-ROM for the pbrt book)

Windy waves

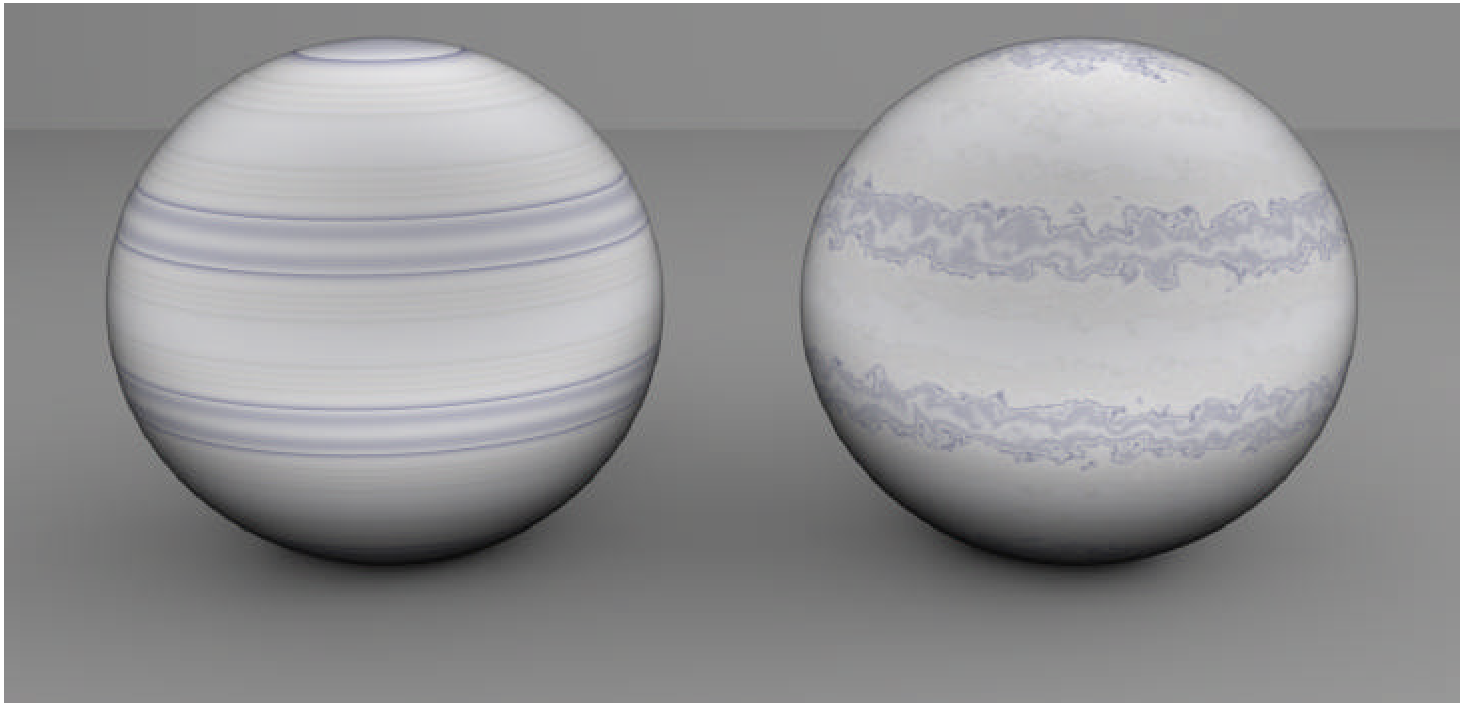
- FBm can be used to create textures of windy waves
- In PBRT WindyTexture class employs FBm function twice to generate texture of realistic water surface
- The first call to FBm is used to get local variation of wind strength
- The second call to FBm determines amplitude of the wave at the particular point
- The product of these two values is returned by Evaluate function as actual wave offset for a particular point

Windy waves

Waves created with WindyTexture
(from companion CD-ROM for the pbrt book)

Marble

- Marble material can be represented as a series of a layered strata
- Noise is then used to perturb coordinates that are used to look up color values among the strata



MarbleTexture perturbs the coordinate used to index table of colors with FBm function (from companion CD-ROM for the pbrt book)

Worley Noise

- In 1996 Steven Worley introduced "Cellular texture based function"
- Basic idea
  - In 3D space n points are randomly chosen – **feature points**
  - Given arbitrary point x the function $F_1(x)$ - distance between point x and the closest feature point. $F_2(x)$ - distance from x to the second closest feature point and so on
  - Values, returned by functions $F_n(x)$ are mapped to color or texture coordiantes

Worley Noise

- Implementation
  - 3D space is partitioned into cubes with faces at integers
  - Given a point ***p*** with real coordinates (x, y, z) the index of the cube that the point lies inside is floor of x, y and z
  - Index of the cube is used to seed a random number generator
  - The random number is then used for a number of feature points inside the cube
  - The random number generator is used again to find coordinates of those feature points
  - The distances from feature points to the lookup point are calculated and sorted
  - Neighboring cubes should also be checked for presence of feature points that are closer to ***p*** than those in the current cube

Worley Noise

- For the distance calculation different distance metrics can be used:
  - Eucidean distance ("as-the-crow-flies"):

$$d = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

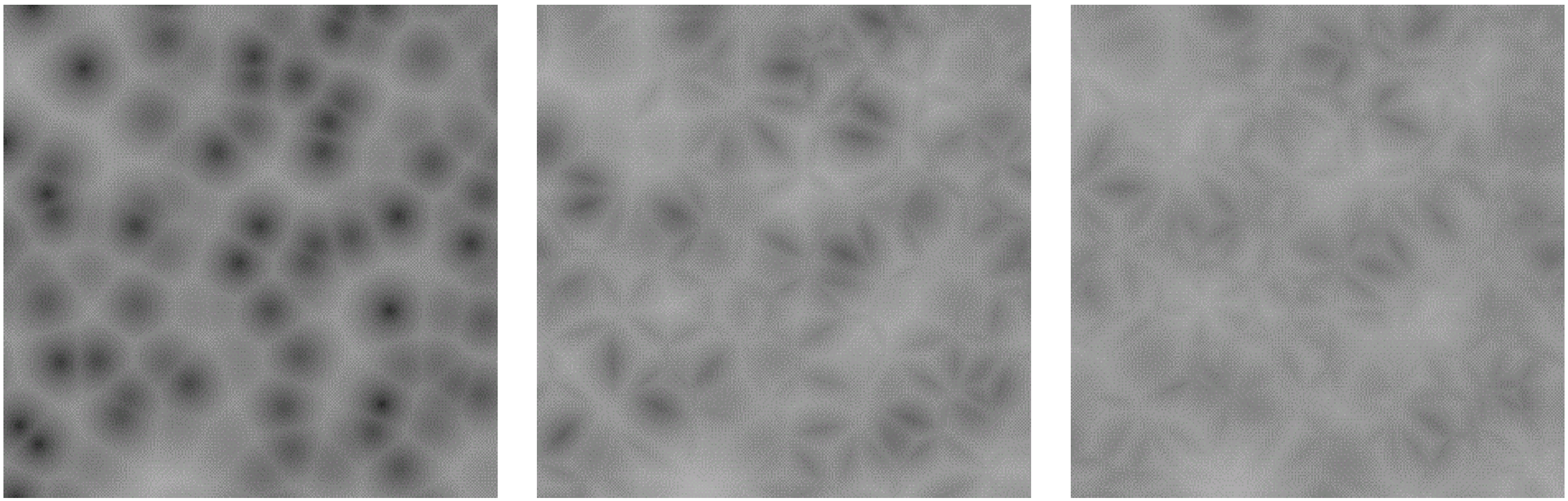
  - Manhattan distance ("city-block"):

$$d = \sum_{i=1}^{n} \left| x_i - y_i \right|$$

  - Chebychev distance:

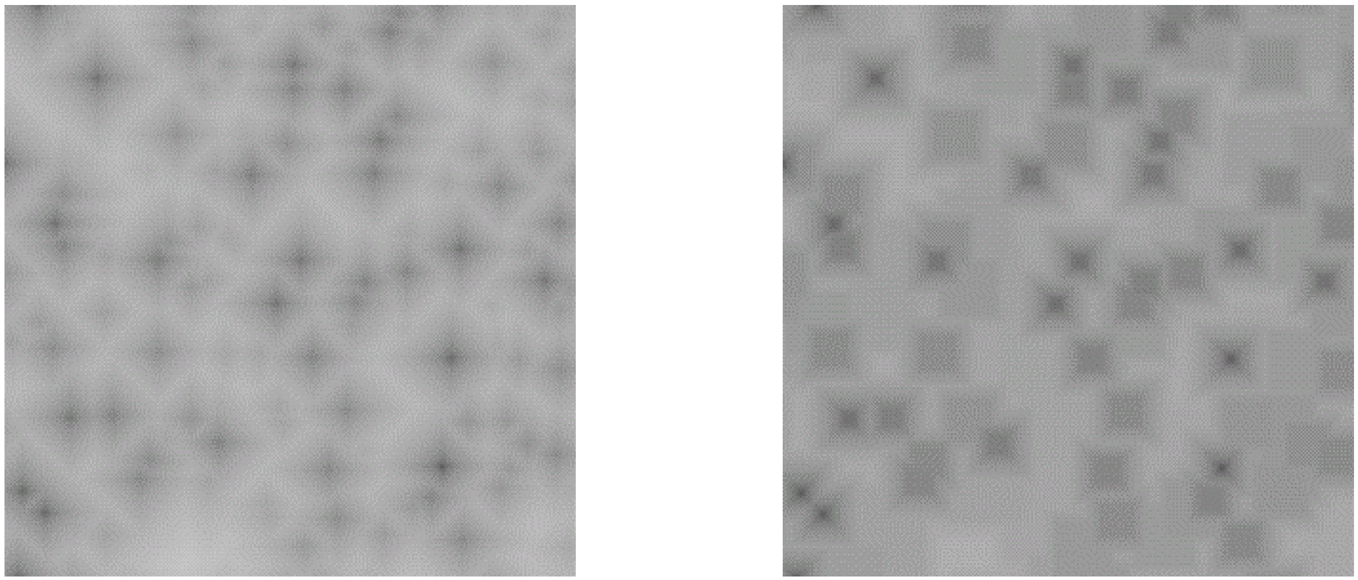
$$d = \max_i (x_i - y_i)$$

Worley Noise



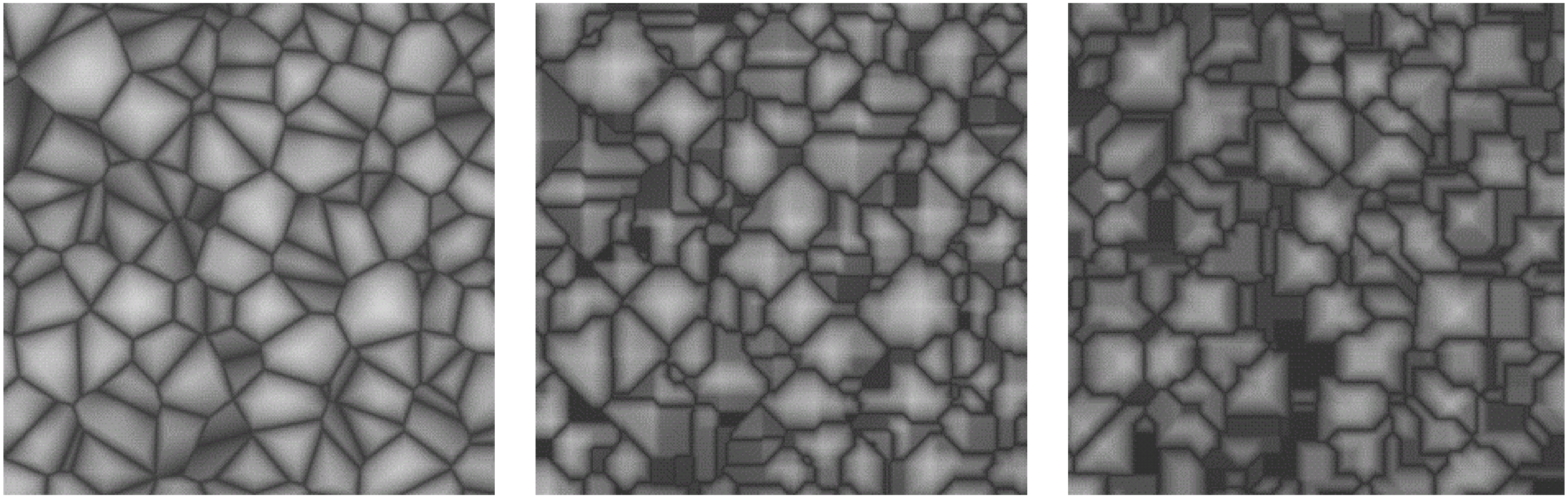
F1, F2 and F3 (from left to right) mapped to grayscale color. Euclidean distance was used as distance metrics

Worley Noise



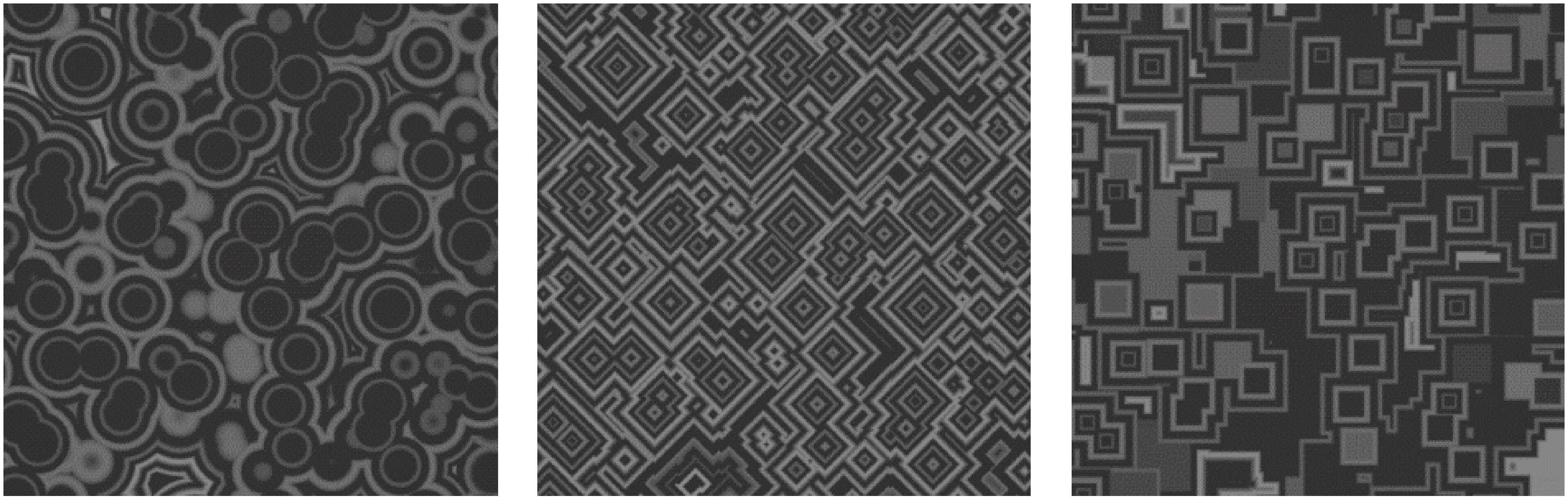
F1 mapped to grayscale color with Manhattan and Chebychev distance metrics

Worley Noise



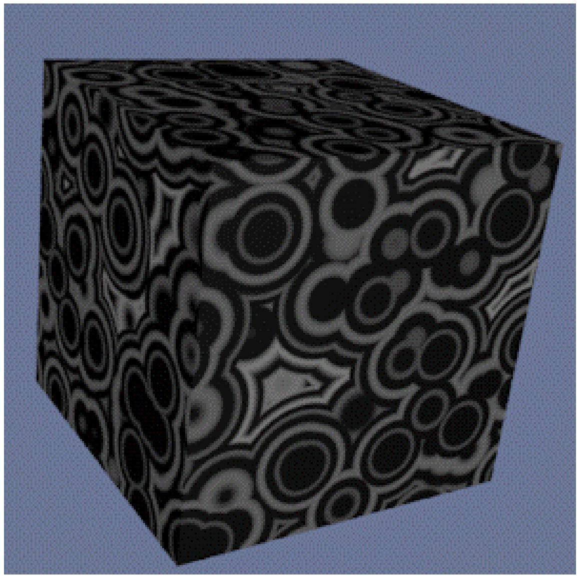
F2 – F1 is mapped to grayscale color with Euclidean, Manhattan and Chebychev distances (from left to right)

Worley Noise



F1 was used to find components of a point p (x, y, z) that was then passed to Perlin noise function. The result was mapped to grayscale color

Worley Noise



"Concentric rings" texture applied to different shapes

References

- Matt Pharr, Greg Humphreys

  “Physically Based Rendering from theory to implementation”
- Boonthanome Nouanesengsy

  CSE 782 Lab 4: http://www.cse.ohiostate.edu/~nouanese/782/lab4/
- Ken Perlin

  “Making noise”:

  http://www.noisemachine.com/talk1/index.html