

# CSE 564: Scientific Visualization

## Lecture 15: Isosurface Rendering

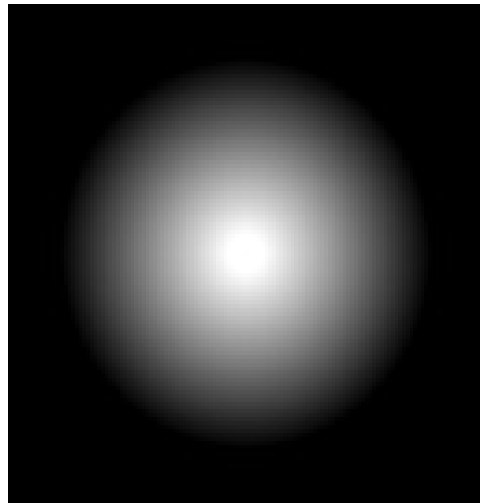
Klaus Mueller

Stony Brook University

Computer Science Department

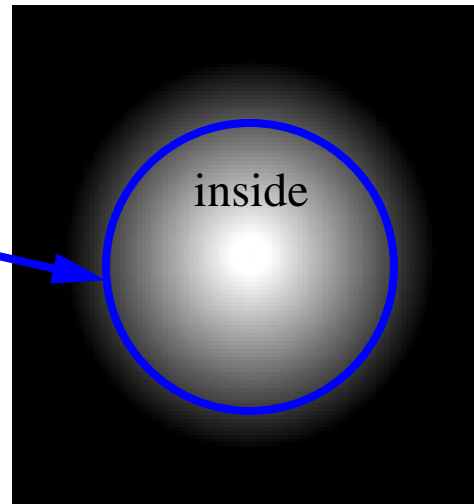
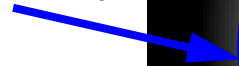
# Iso-Surface Rendering

- A closed surface separates ‘outside’ from ‘inside’ (Jordan theorem)
- In iso-surface rendering we say that all voxels with values  $>$  some threshold are ‘inside’, and the others are ‘outside’
- The boundary between ‘outside’ and ‘inside’ is the *iso-surface*
- All voxels near the iso-surface have a value close to the *iso-threshold* or *iso-value*
- Example:

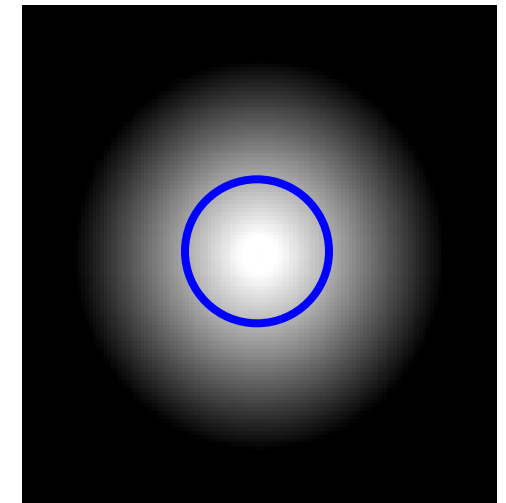


cross-section of a smooth sphere

iso-boundary



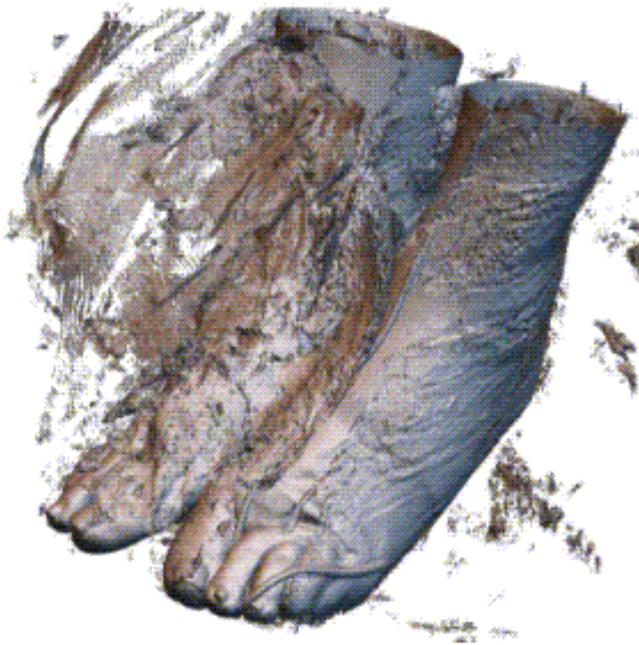
iso-value = 50  
will render a large sphere



iso-value = 200  
will render a small sphere

# Iso-Surface Rendering - Example

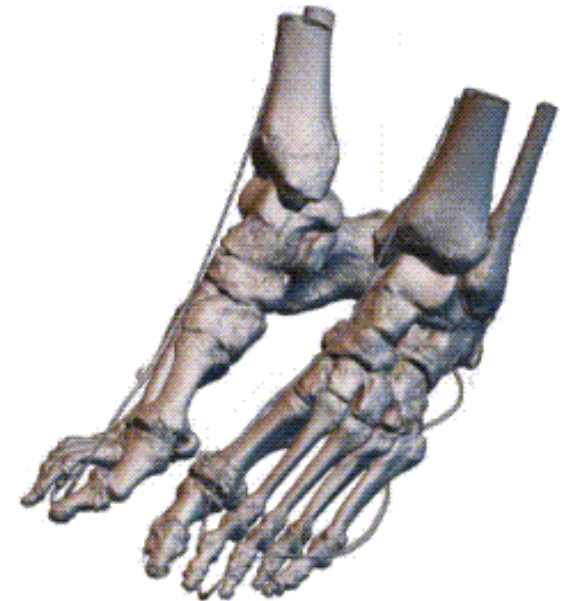
## Foot of the Visible Woman



iso-value = 30



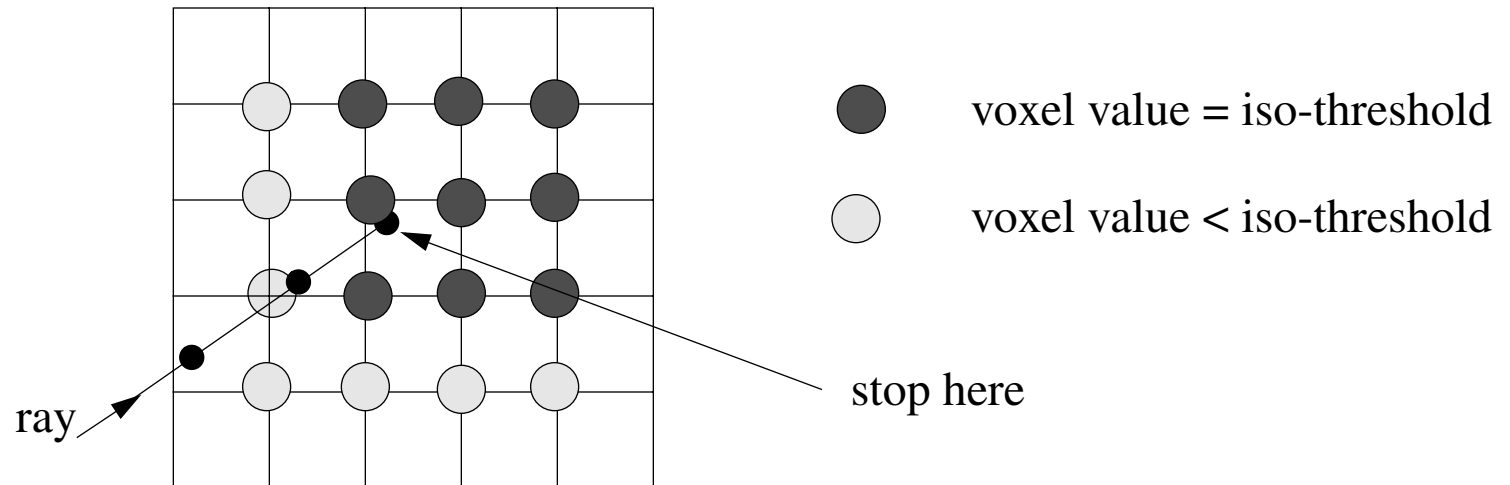
iso-value = 80



iso-value = 200

# Iso-Surface Rendering - Details

- To render an iso-surface we cast the rays as usual...  
but we stop, once we have interpolated a value  $\geq$  iso-threshold



- We would like to illuminate (shade) the iso-surface based on its orientation to the light source
- Recall that we need a normal vector for shading
- The normal vector  $N$  is the local gradient, normalized

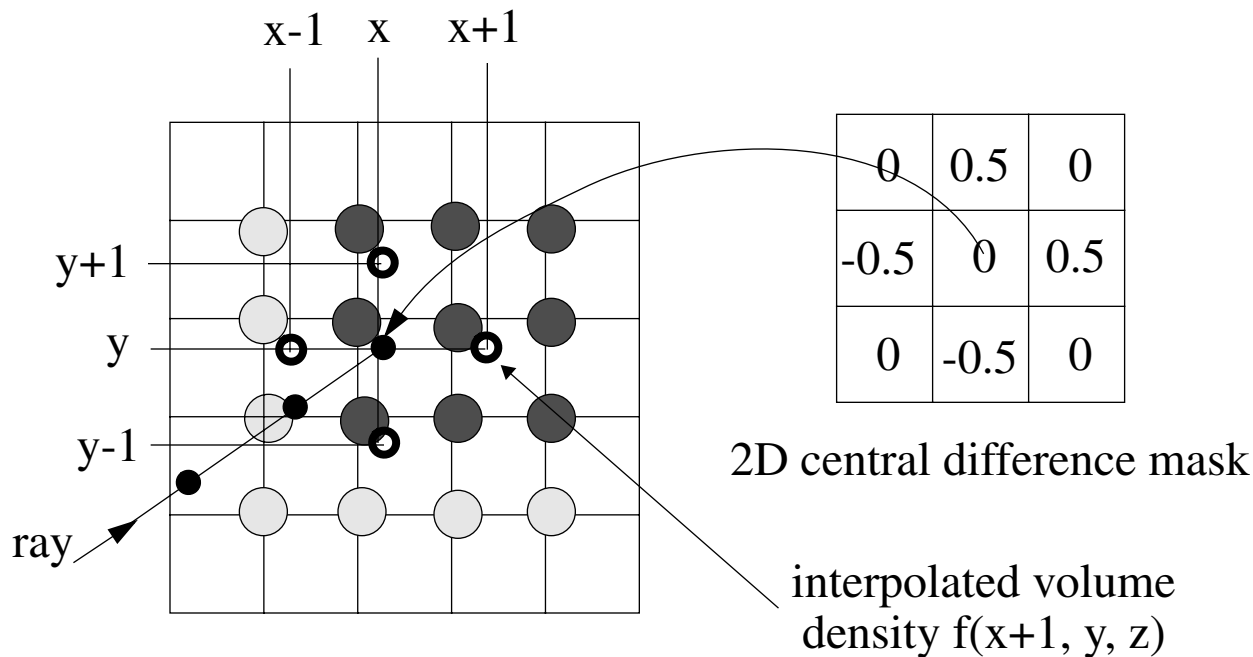
# The Gradient Vector

- The gradient vector  $\mathbf{g}=(g_x, g_y, g_z)^T$  at the sample position  $(x, y, z)$  is usually computed via central-differencing (for example,  $g_x$  is the volume density gradient in the x-direction):

$$g_x = \frac{f(x-1, y, z) - f(x+1, y, z)}{2}$$

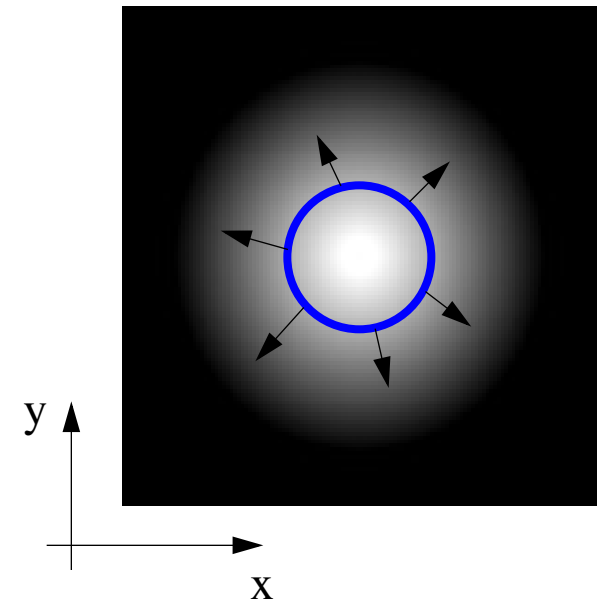
$$g_y = \frac{f(x, y-1, z) - f(x, y+1, z)}{2}$$

$$g_z = \frac{f(x, y, z-1) - f(x, y, z+1)}{2}$$



- voxel value = iso-threshold
- voxel value < iso-threshold
- ⊙ extra sample points interpolated to estimate gradient

the x and y component of the gradient vector for the smooth sphere



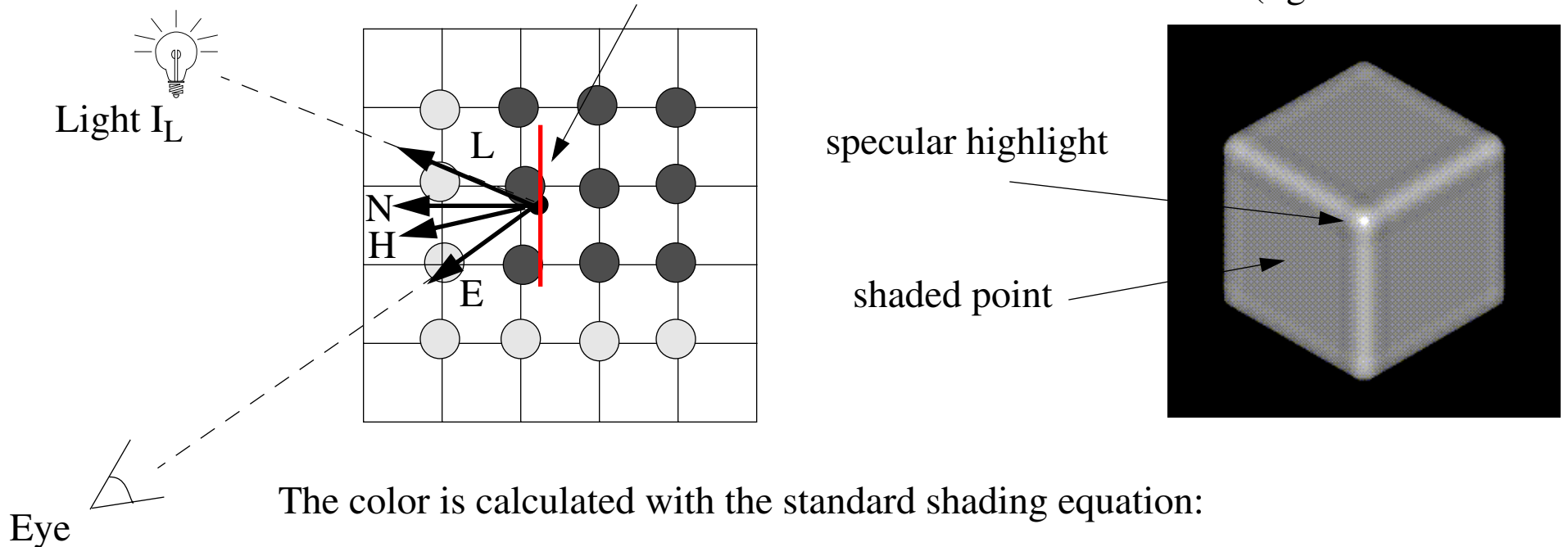
More on gradient filters later...

# Shading the Iso-Surface

- The normal vector is the *normalized* gradient vector  $g$   
 $N = g / |g|$  (normal vector always has unit length)
- Once the normal vector has been calculated we shade the iso-surface at the sample point
- The color so obtained is then written to the pixel that is due to the ray

detected iso-surface portion

rendered cube (light from the front)

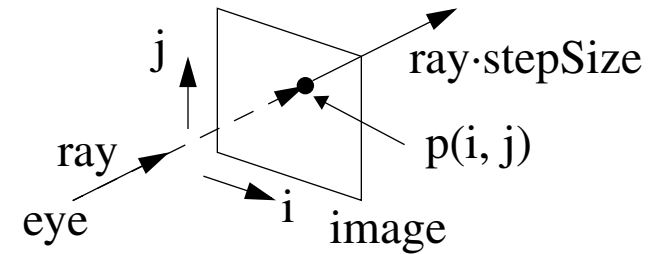


The color is calculated with the standard shading equation:

$$C = C_{obj} (k_a I_A + k_d I_L N \cdot L) + k_s I_L (H \cdot N)^{ns}$$

$C_{obj}$  is obtained by indexing the color transfer function with the interpolated sample value

## Iso-Surface Rendering - Algorithm



```
RenderIsoSurface(Volume V, int stepSize)
```

```
for each image pixel p(i, j)
```

```
ray = (p(i, j) - eye) / | (p(i, j) - eye) | /* the ray direction vector, normalized */
```

```
t = 0 /* start at the eye point */
```

```
do forever
```

```
sampleLoc = eye + t · stepSize · ray /* step along the ray */
```

```
intVal = Interpolate(V, sampleLoc)
```

```
if opacityTransferFunction(intVal) > isoThreshold /* found the iso-surface */
```

```
/* interpolate 6 samples around sampleLoc and compute the gradient */
```

```
gradVec = ComputeGradientVector(V, sampleLoc)
```

```
/* shade the surface using standard illumination model and color transfer functions */
```

```
{r, g, b} = Shade(gradVec, lightSource, eye, sampleLoc, {R, G, B}TransFunc(intVal))
```

```
value(p(i, j)) = {r, g, b} /* write color into image pixel p(i, j) */
```

```
break /* terminate this ray and go to next image pixel */
```

```
t = t + 1 /* iso-surface not found yet, get ready to step to next sample point */
```

# Gradient Filter Theory (1)

- The gradient  $g$  is the derivative of the signal  $f$

$$g = f_c' = \begin{cases} (h \otimes f_s)' \\ h \otimes f_s' \\ h' \otimes f_s \end{cases} \quad \text{where}$$

$f_c$ : continuous signal

$f_s$ : sampled signal

$h$ : interpolation filter

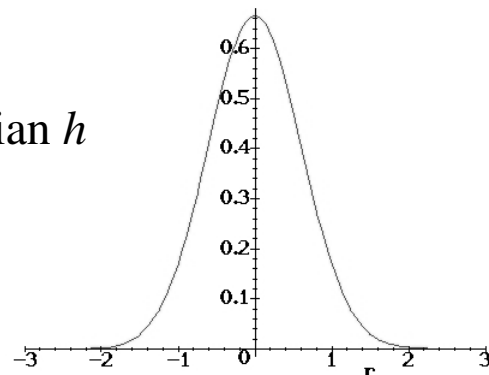
$'$ : the derivative operator

$\otimes$ : convolution operator

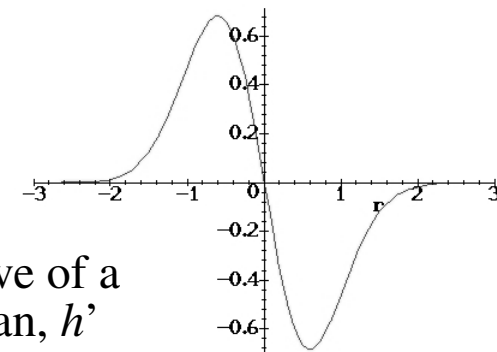
- So we can either:

- interpolate the signal  $f_s$  first and then take the derivative (that is what we did before)
- take the derivative at the grid positions ( $\rightarrow \mathbf{g}_s = (g_x, g_y, g_z)^T$ ) and then interpolate  $\mathbf{g}_s$
- take the derivative of  $h$  ( $\rightarrow h'$ ) and interpolate  $f_s$  with  $h'$

Gaussian  $h$

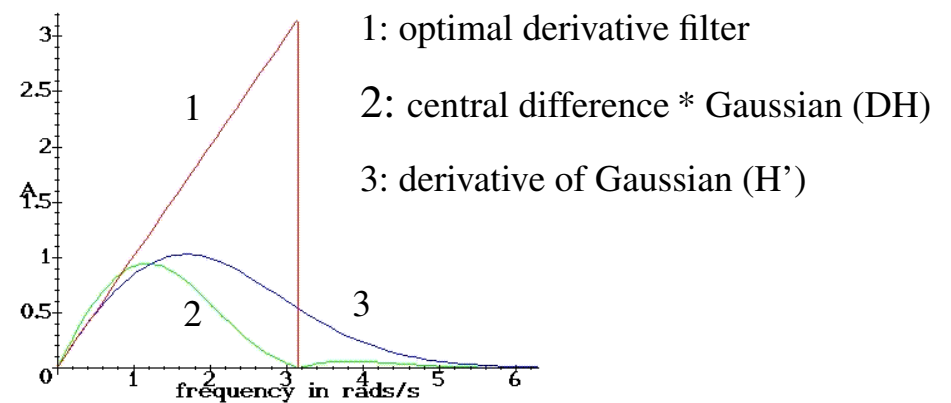
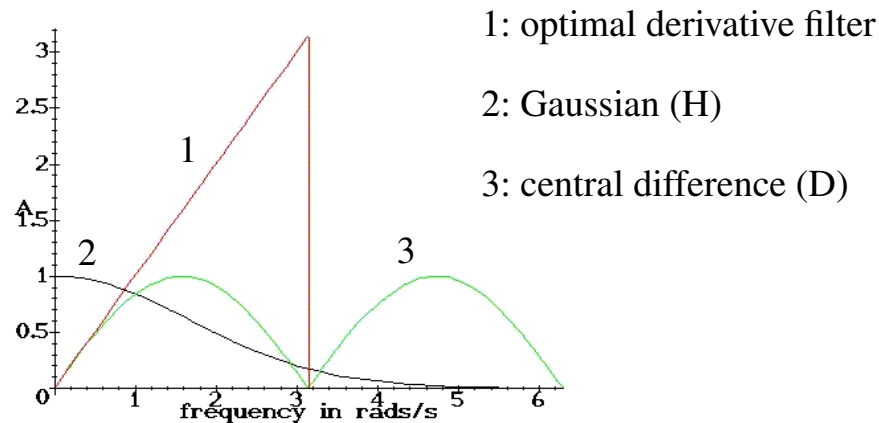


Derivative of a Gaussian,  $h'$



## Gradient Filter Theory (2)

- Quality evaluation is best done in the frequency domain:



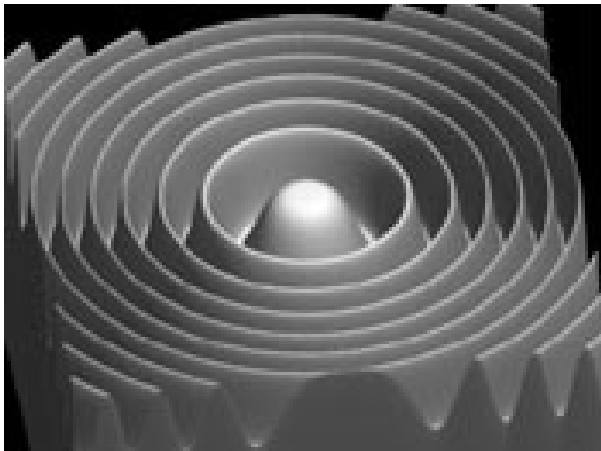
- The optimal gradient filter is a ramp that ends at the cut-off frequency
  - gradients (edges, etc.) are the high frequencies that we want to emphasize
  - homogenous (uniform) regions have low frequencies and need to be suppressed
- The derivative filter  $H'$  passes the higher frequencies better than  $DH$  but it also passes some aliases
  - more sensitive to noise
  - but promises to estimate crisper edges

# Interlude - The Marschner-Lobb Test Function for Filters

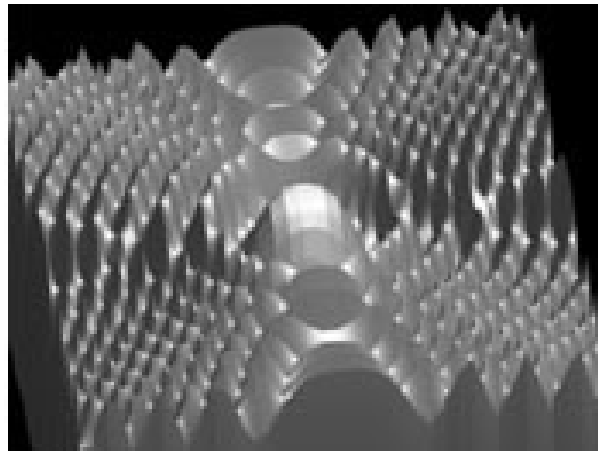
- A common test function to evaluate rendering filters is the Marschner-Lobb function

$$\rho(x, y, z) = 255 \cdot \frac{(1 - \sin(\pi z/2)) + \alpha \left(1 + \rho_r \left(\sqrt{x^2 + y^2}\right)\right)}{2(1 + \alpha)} \quad \rho_r(r) = \cos\left(2\pi f_M \cos\left(\frac{\pi r}{2}\right)\right) \quad \begin{array}{l} f_M = 6 \\ \alpha = 0.25 \end{array}$$

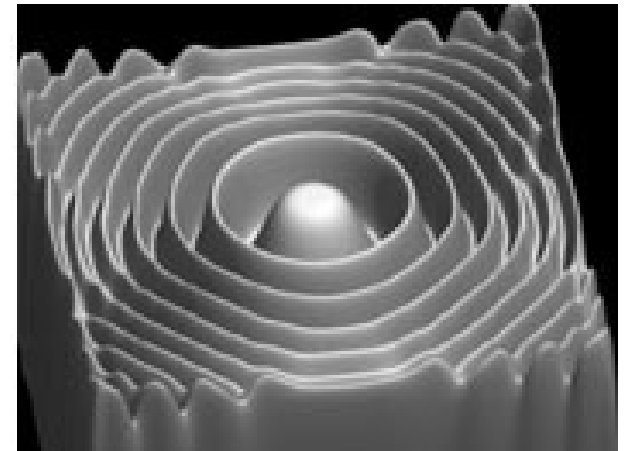
- when sampled into a  $40^3$  grid, 99.8% of the frequencies are below the Nyquist rate,
  - the function is interesting since a significant portion of the spectrum is close to Nyquist
  - this makes this function a demanding test for interpolation and gradient filters
- Usually the iso-value is set to 128
  - Some images of the ML-function obtained with different interpolation filters:



original function



tri-linear filter



windowed sinc

## Gradient Filter Theory (3)

- It turns out that inaccurate estimation of gradients is much more noticeable than inaccurate interpolation
  - this is because the lighting depends greatly on the normal vectors
  - especially specular highlights (due to the exponentiation of  $H \cdot N$ ) are rather sensitive
- To show this, let us take a good filter (family) and vary its quality
- We know that the cubic function is a reasonably good interpolation filter
- To study the sensitivity, we can just vary the coefficient  $\alpha$  to sub-optimal values
  - use the cubic filter for interpolation
  - use the derivative of the cubic function as a gradient filter
  - theory has shown that  $\alpha = -0.5$  is a good choice for interpolation
  - let us now deviate from that value and see what happens for both interpolation filter and gradient filter

(We will see that the choice of  $\alpha$  affects the derivative filter much more than the interpolation)

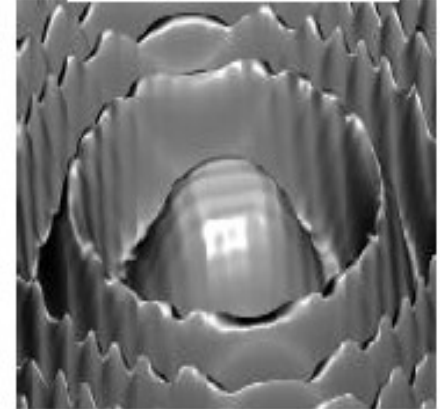
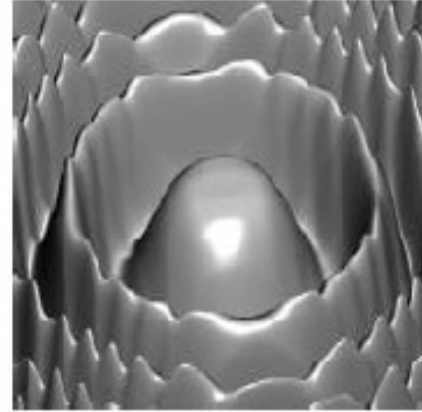
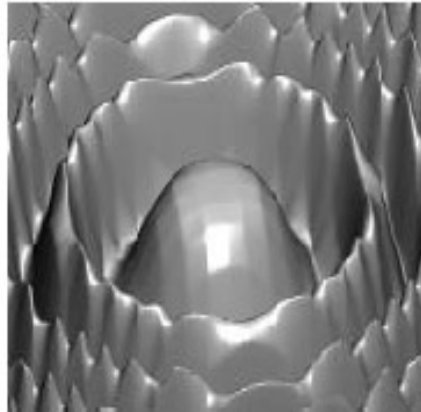
# Gradient Filter Theory (4)

Derivative  $\alpha = -0.2$

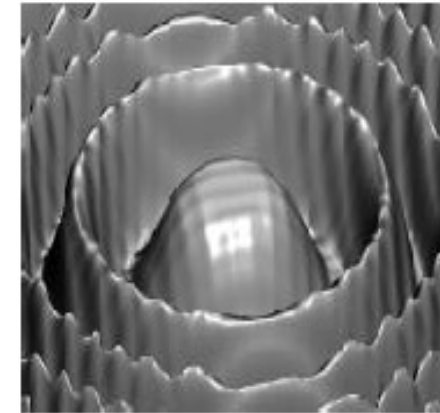
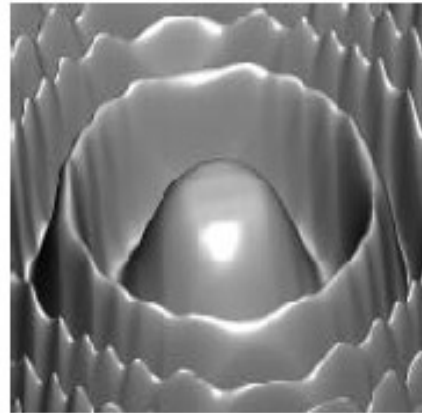
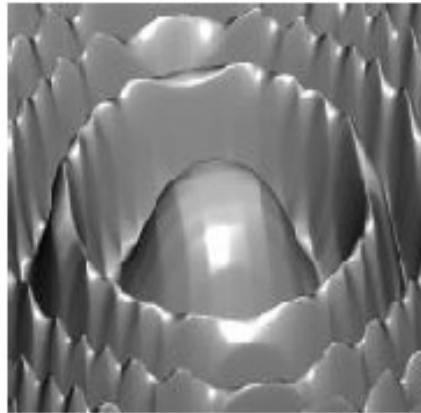
Derivative  $\alpha = -0.5$

Derivative  $\alpha = -1.0$

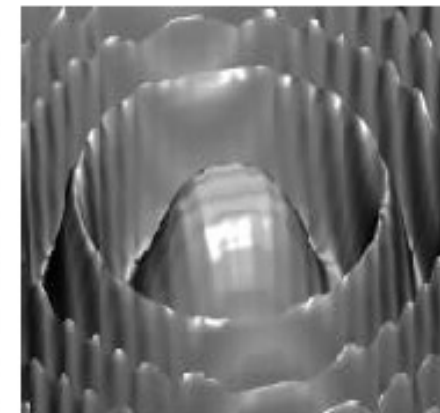
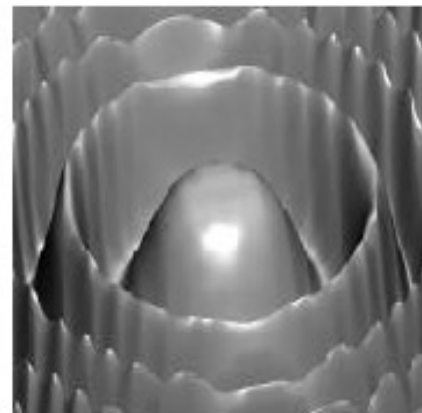
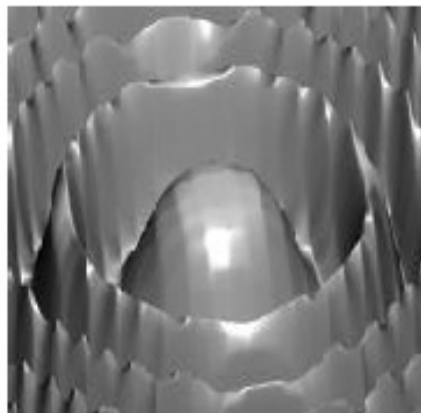
Interpolation  $\alpha = -0.2$



Interpolation  $\alpha = -0.5$

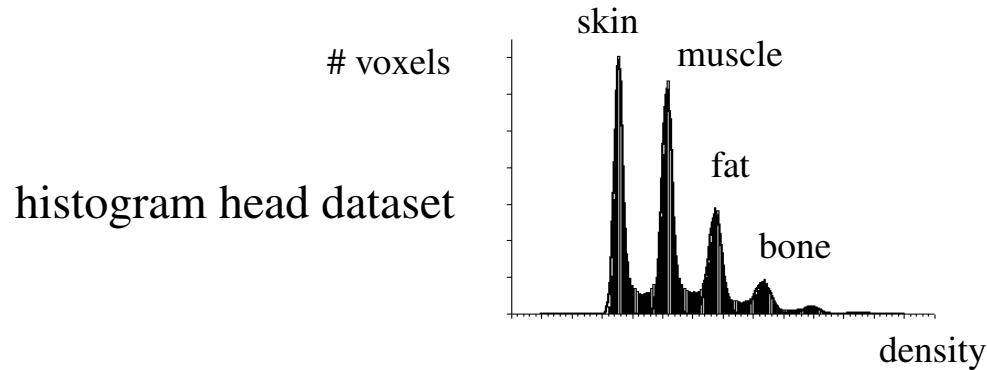


Interpolation  $\alpha = -1.0$

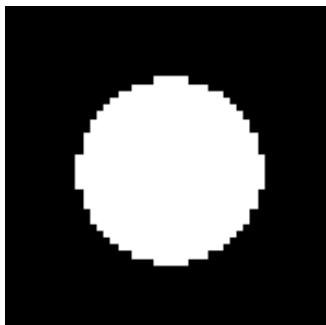


# Iso-Surface Rendering - Tips and Tricks (1)

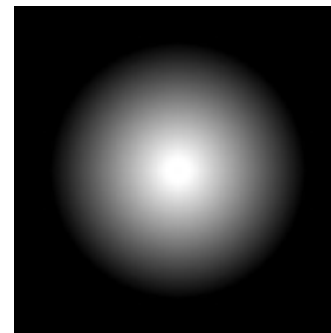
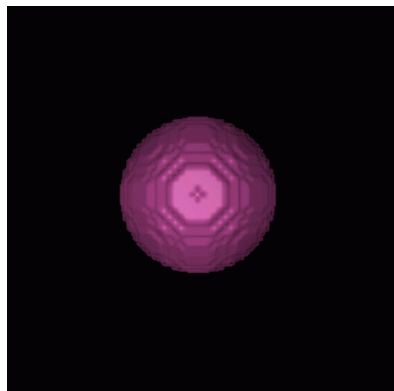
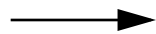
- Finding a good iso-value is not always easy
  - make a histogram of the volume densities and look for peaks (iso-value = onset of peak)



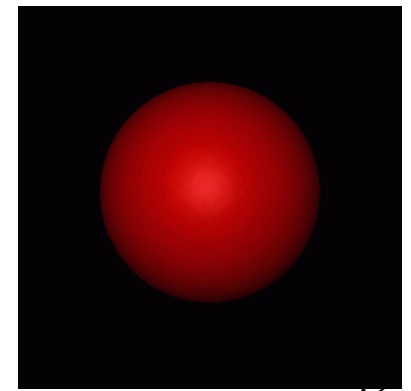
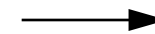
- Good shading requires good gradients around iso-surface
  - need smooth degradations at iso-surface for good gradient estimation
  - else get aliasing (example: volumetric binary sphere)



binary sphere

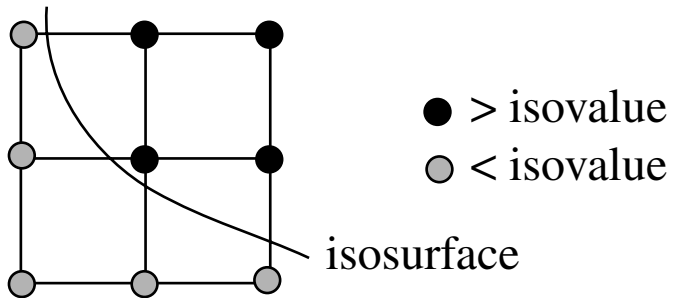


smooth sphere

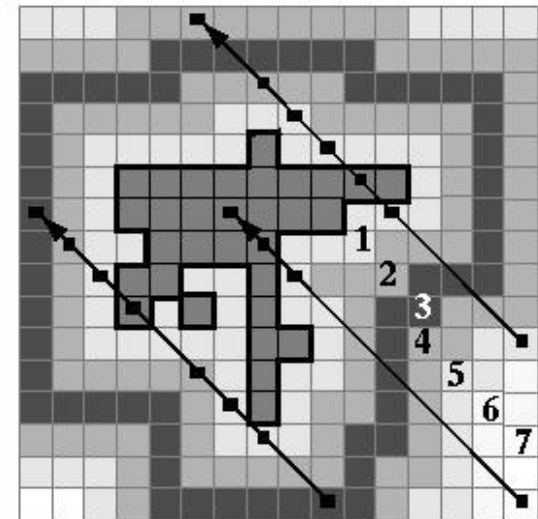


## Iso-Surface Rendering - Tips and Tricks (2)

- Ray stepsize must be chosen sufficiently small
  - choose stepsize of less or equal 1.0 voxel units (or we may get aliasing in the ray direction)
- But even for small stepsizes, we may never exactly hit the isosurface
  - isosurface goes through a cell when at least one vertex, but not all, has a density  $>$  isoValue
  - compute exact location of the iso-surface within a cell by solving a cubic function in  $t$



space leaping



- A variety of acceleration methods are possible:
  - enclose the object in a bounding box and start rays at the bounding box intersection
  - store distance values in voxels outside the object  $\rightarrow$  this enables quick *space leaping*
  - multi-resolution volume representation (octree)