

# CSE 564: Scientific Visualization

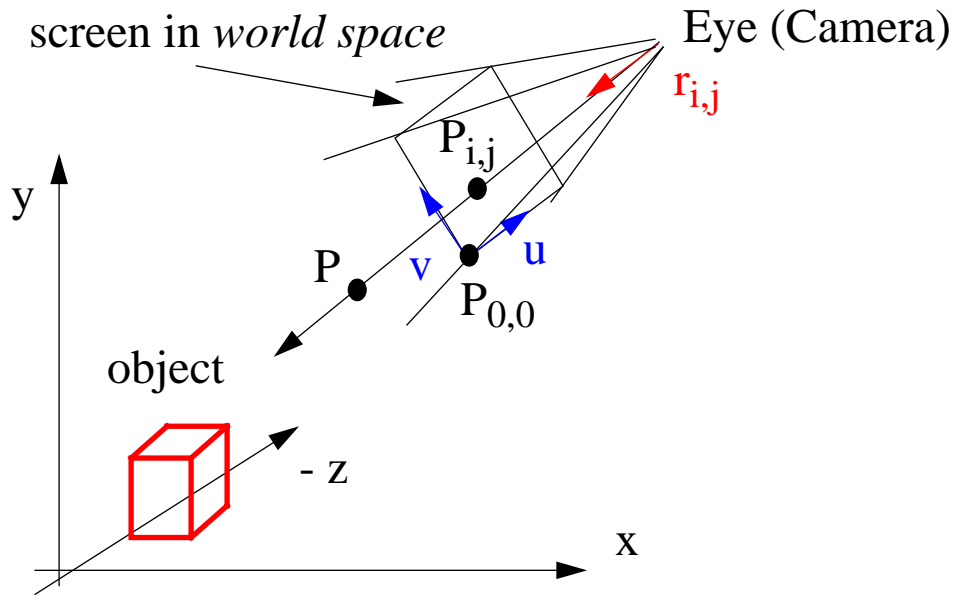
## Lecture 11: Volume Rendering With Raycasting

Klaus Mueller

Sony Brook University

Computer Science Department

# Image-Order Viewing (Raycasting) - Perspective Projection



A ray is specified by:

- eye position (Eye)
- screen pixel location  $P_{i,j}$

→ ray direction vector ( $r_{i,j}$ ) of unit length

$$r_{i,j} = \frac{P_{i,j} - Eye}{|P_{i,j} - Eye|}$$

A point  $P$  on a ray is given by:

$$P = Eye + t \cdot r_{i,j}$$

$t$ : parametric variable

Spacing of pixels on image plane:

$$\Delta i = \frac{W}{N_i - 1} \quad \Delta j = \frac{H}{N_j - 1}$$

$N_i, N_j$ : image dims. in pixels

Image-order projection:

- scan the image row by row, column by column:

$$P_{i,j} = P_{0,0} + i \cdot v \cdot \Delta j + j \cdot u \cdot \Delta i$$

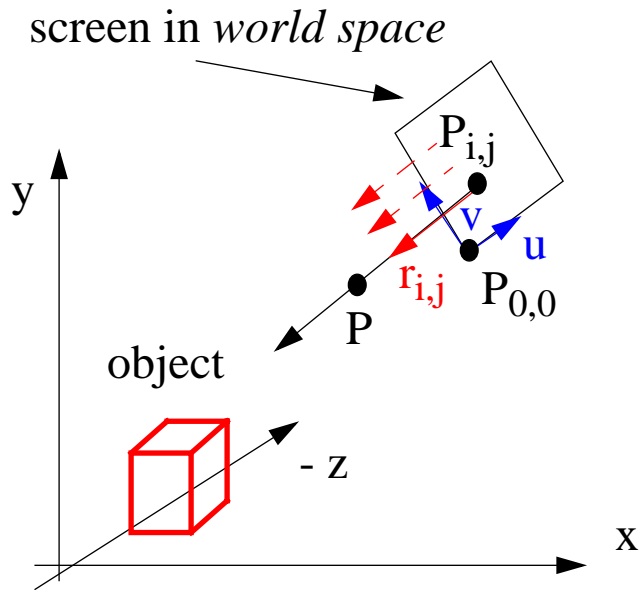
- $P_{i,j}$ : Location of image pixel ( $i, j$ ) in world space

$$0 \leq i < N_i \quad 0 \leq j < N_j$$

- $P_{0,0}$ : image (=screen) origin in world space

- $u, v, n$ : orthonormal image plane vectors ( $n = v \times u$ )

# Image-Order Viewing (Raycasting) - Orthographic Projection



All rays are parallel

A ray is specified as:

$r_{i,j} = n$ , the view direction vector

$n = v \times u$  (cross product of  $v$ ,  $u$ )

A point  $P$  on a ray is given by:

$$P = P_{i,j} + t \cdot r_{i,j}$$

$t$ : parametric variable

Spacing of pixels on image plane:

$$\Delta i = \frac{W}{N_i - 1} \quad \Delta j = \frac{H}{N_j - 1}$$

$N_i, N_j$ : image dims. in pixels

Image-order projection:

- scan the image row by row, column by column:

$$P_{i,j} = P_{0,0} + i \cdot v \cdot \Delta j + j \cdot u \cdot \Delta i$$

-  $P_{i,j}$ : Location of image pixel  $(i, j)$  in world space

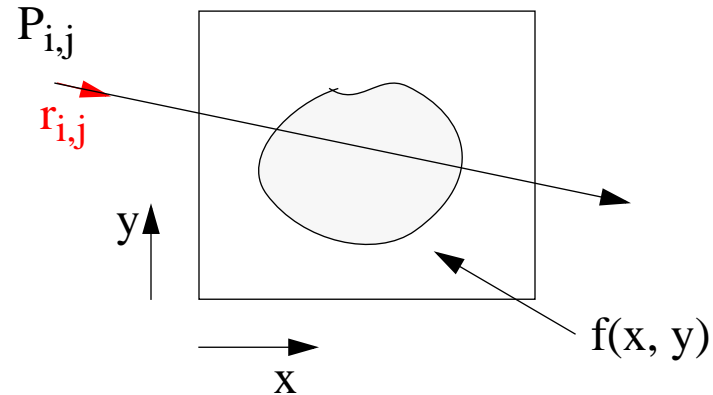
$$0 \leq i < N_i \quad 0 \leq j < N_j$$

-  $P_{0,0}$ : image (=screen) origin in world space

-  $u, v, n$ : orthonormal image plane vectors ( $n = v \times u$ )

# Volumetric Raycasting - Continuous Form

2D Example:



- X-Ray projection:

$$I_{i, j} = \int_0^L f(P_{i, j} + t \cdot r_{i, j}) dt$$

- accumulate (integrate) the volumetric function  $f(x, y)$  along the ray
- write the resulting intensity  $I_{i,j}$  to the image at location  $(i, j)$

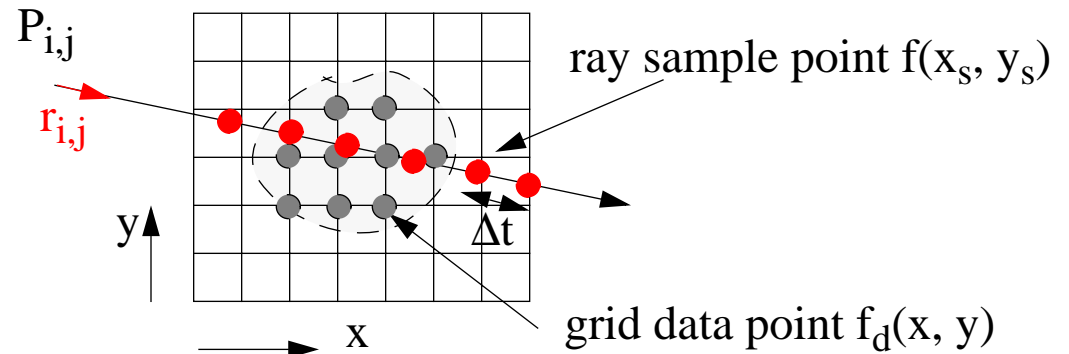
- Maximum Intensity Projection (MIP)

$$I_{i, j} = \text{Max}(f(P_{i, j} + t \cdot r_{i, j}))$$

- find the maximum of  $f(x, y)$  along the ray and write it to image location  $(i, j)$

# Volumetric Raycasting - Discrete Form

- In a practical volume rendering situation we have:
  - discretized data on a grid (obtained by some imaging or simulation modality)
  - volumetric integrals that cannot be solved in closed form (i.e. analytically) and therefore must be approximated by discrete rays



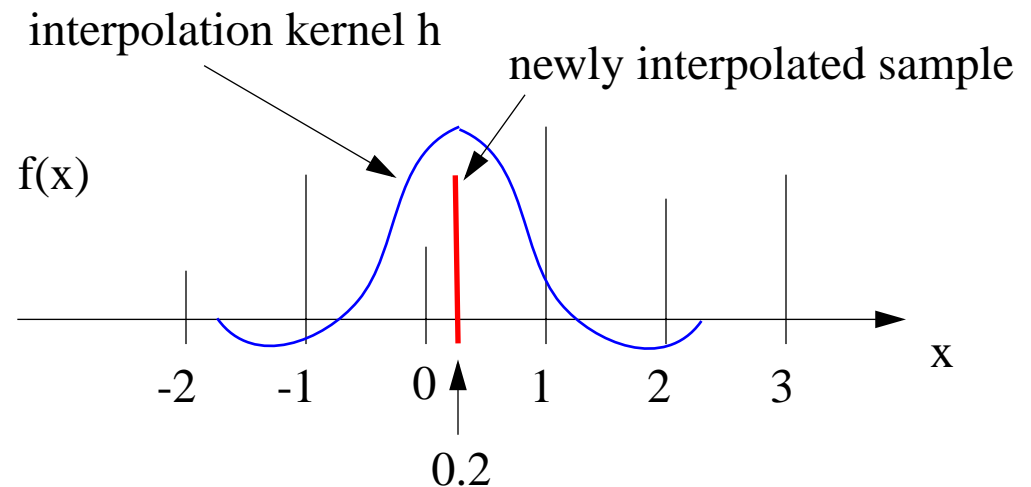
- Discrete X-Ray projection:

$$I_{i,j} = \sum_{k=0}^{L/\Delta t} f(P_{i,j} + k \cdot \Delta t \cdot r_{i,j}) \cdot \Delta t$$

- sample the volumetric function  $f(x, y)$  at equi-spaced distances  $\Delta t$  along the ray
  - accumulate the samples and write the resulting intensity  $I_{i,j}$  to the image at location  $(i, j)$
- Discrete Maximum Intensity Projection (MIP):  $I_{i,j} = \text{Max}(f(P_{i,j} + k \cdot \Delta t \cdot r_{i,j}))_{0 \leq k \leq L/\Delta t}$

# Interpolation

- The ray sample points do not always fall onto the known grid points
  - most often they fall somewhere inbetween
- We need to estimate the values of the ray samples from the existing grid points
- This is done via interpolation (as discussed earlier)
- Let's start with the 1D case:

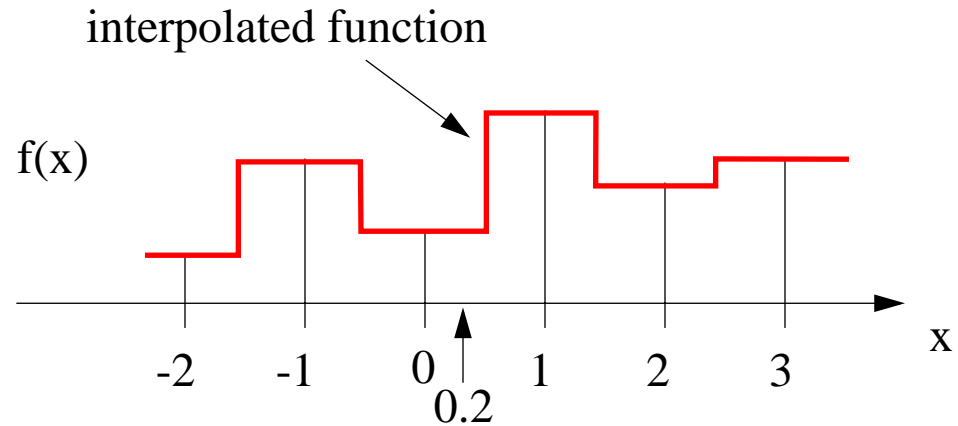
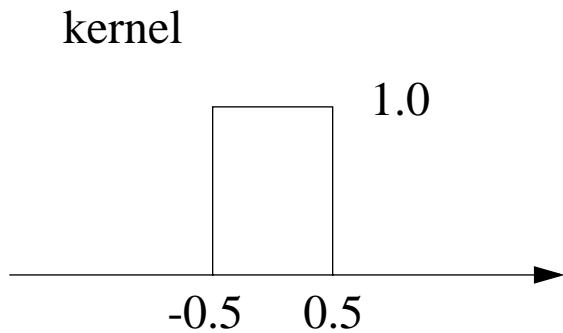


- center the interpolation kernel at the sample position and superimpose it onto the grid
- multiply the values of the grid samples with the kernel value at the superimposed position
- add all the products, this gives the value of the newly interpolated sample

$$\text{in the shown case: } f(0.2) = h(-0.2)f(0) + h(-1.2)f(-1) + h(0.8)f(1) + h(1.8)f(2)$$

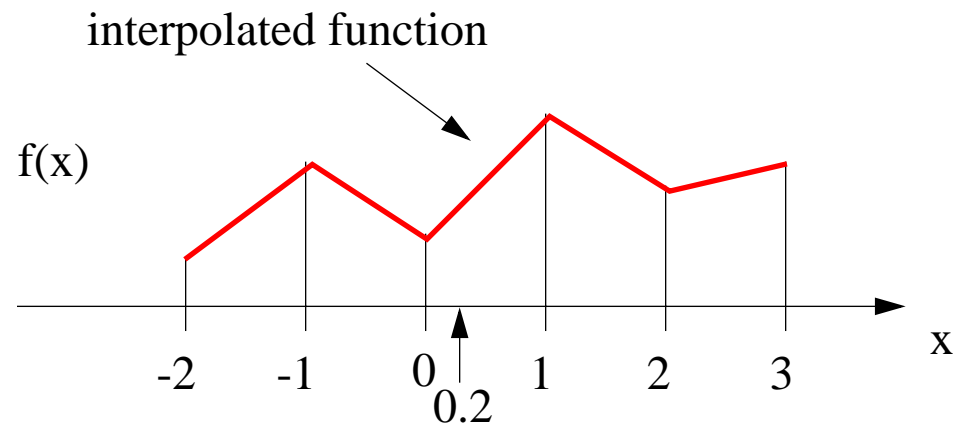
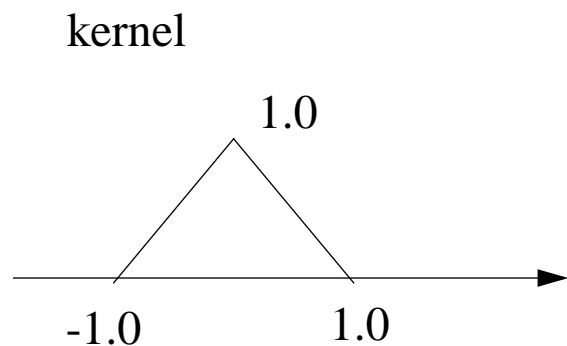
# Popular Interpolation Kernels (1)

- Nearest Neighbor:



- simply pick the value of the nearest grid point:  $f(0.2) = f(\text{trunc}(0.2+0.5)) = f(\text{round}(0.2))$

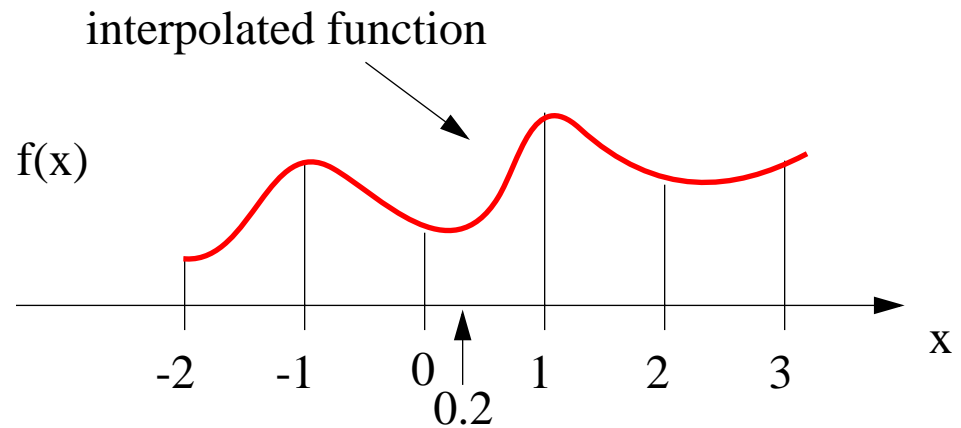
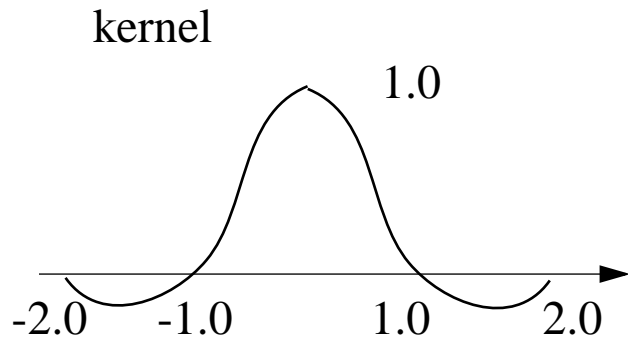
- Linear filter:



- use a linear combination of the two neighboring grid values:  $f(0.2) = 0.2 \cdot f(1) + 0.8 \cdot f(0)$

## Popular Interpolation Kernels (2)

- Cubic filter:



- see equation on pg. 114 in the Lichtenbelt book

- Discussion:

- nearest neighbor is fastest to compute (just one add) but gives a sharp edges
- linear interpolation takes 2 mults and 1 add and gives a piecewise smooth function
- cubic filter takes 4 mults and 3 adds but gives an overall smooth interpolated function
- linear interpolation is most popular in volume rendering application

# Fast Interpolation In Higher Dimensions (1)

- All interpolation kernels shown here are separable

$$h(x, y) = h(x) \cdot h(y) \quad \text{and} \quad h(x, y, z) = h(x) \cdot h(y) \cdot h(z)$$

- Linear interpolation

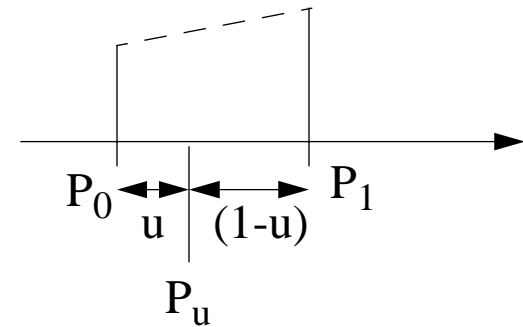
assume: grid distance = 1.0

$P_u$  is the location of the sample value

$P_0$  and  $P_1$  are neighboring grid points

then:  $u = P_u - P_0$

$$f(x) = f(P_u) = (1 - u) \cdot f(P_0) + u \cdot f(P_1)$$



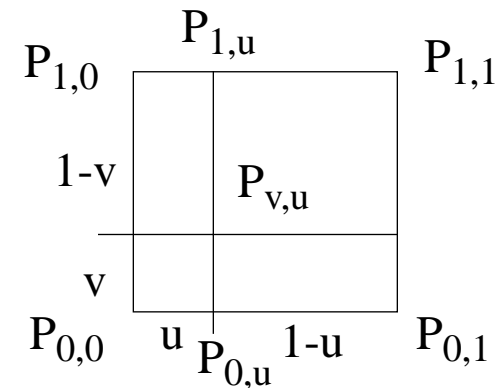
- Bilinear interpolation

$$f(P_{0,u}) = (1 - u) \cdot f(P_{0,0}) + u \cdot f(P_{0,1})$$

$$f(P_{1,u}) = (1 - u) \cdot f(P_{1,0}) + u \cdot f(P_{1,1})$$

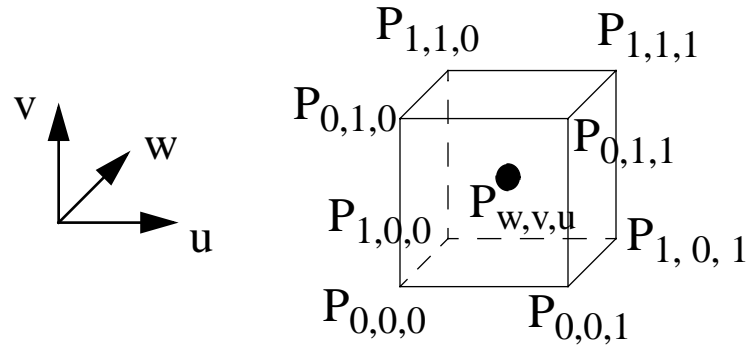
$$f(P_{v,u}) = (1 - v) \cdot f(P_{0,u}) + v \cdot f(P_{1,u})$$

$$\rightarrow f(x, y) = f(P_{v,u}) = (1-v) (1-u) f(P_{0,0}) + (1-v) u f(P_{0,1}) + v (1-u) f(P_{1,0}) + v u f(P_{1,1})$$



## Fast Interpolation In Higher Dimensions (2)

- Trilinear interpolation



$$f(P_{0,v,u}) = \text{BilinearInterpolation}(P_{0,0,0}, P_{0,0,1}, P_{0,1,0}, P_{0,1,1})$$

$$f(P_{1,v,u}) = \text{BilinearInterpolation}(P_{1,0,0}, P_{1,0,1}, P_{1,1,0}, P_{1,1,1})$$

$$f(x, y, z) = f(P_{w,v,u}) = \text{LinearInterpolation}(P_{0,v,u}, P_{1,v,u})$$

hence, a trilinear interpolation can be decomposed into 7 linear interpolations

$$\text{alternative form: } f(x, y, z) = \sum_{i=0}^1 \sum_{j=0}^1 \sum_{k=0}^1 u^i (1-u)^{1-i} v^j (1-v)^{1-j} w^k (1-w)^{1-k} P_{i,j,k}$$

- Nearest neighbor in three dimensions:

$$f(x, y, z) = f(P_{\text{trunc}(x+0.5), \text{trunc}(y+0.5), \text{trunc}(z+0.5)})$$

- Note, so far we have assumed that the image resolution = volume resolution (no lowpassing req.)

- many raycasters use these equations even when this is not true (more on this later)

## Trilinear Interpolation - Implementation Issues

- For fastest computation, precompute the interpolation weights once per sample point P:

$$u = P[0] - (\text{int})P[0]; \quad v = P[1] - (\text{int})P[1]; \quad w = P[2] - (\text{int})P[2];$$

- Now do the decomposition into 7 linear interpolations:

$$\text{val1} = u * \text{volData}[(\text{int})P[2] * \text{nx} * \text{ny} + (\text{int})P[1] * \text{nx} + (\text{int})P[0] + 1] + \\ (1-u) * \text{volData}[(\text{int})P[2] * \text{nx} * \text{ny} + (\text{int})P[1] * \text{nx} + (\text{int})P[0]];$$

$$\text{val2} = u * \text{volData}[(\text{int})P[2] * \text{nx} * \text{ny} + ((\text{int})P[1] + 1) * \text{nx} + ((\text{int})P[0]) + 1] + \\ (1-u) * \text{volData}[(\text{int})P[2] * \text{nx} * \text{ny} + ((\text{int})P[1] + 1) * \text{nx} + (\text{int})P[0]];$$

$$\text{val3} = (1-v) * \text{val1} + v * \text{val2}; \quad \dots$$

- The array indexing takes many operations and will not be very efficient, try instead:

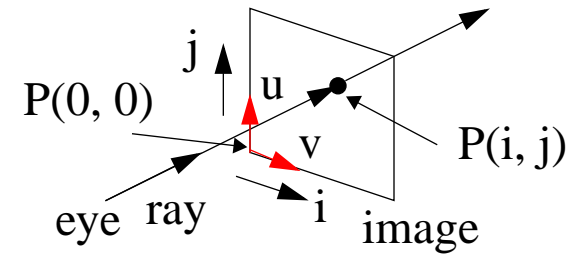
$$\text{cptr} = \&(\text{volume} \rightarrow \text{data}[(\text{int})P[2] * \text{nx} * \text{ny} + (\text{int})P[1] * \text{nx} + (\text{int})P[0]]);$$

$$P000 = * \text{cptr}; \quad P001 = *( \text{cptr} + 1 ); \quad P100 = *( \text{cptr} + \text{nx} * \text{ny} ); \quad P101 = *( \text{cptr} + \text{nx} * \text{ny} + 1 ); \quad P010 = *( \text{cptr} + \text{nx} );$$

$$P011 = *( \text{cptr} + \text{nx} + 1 ); \quad P110 = *( \text{cptr} + \text{nx} * \text{ny} + \text{nx} ); \quad P111 = *( \text{cptr} + \text{nx} * \text{ny} + \text{nx} + 1 );$$

$$\text{intVal} = (1-w) * ( (1-v) * (u * P001 + (1-u) * P000) + v * (u * P011 + (1-u) * P010) ) + \\ w * ( (1-v) * (u * P101 + (1-u) * P100) + v * (u * P111 + (1-u) * P110) );$$

# X-Ray Rendering Algorithm



```
RenderXRay(Volume V, int stepSize, Image I)
```

```
if (projectionMode == Orthographic)
```

```
    ray =  $u \times v / |u \times v|$  /* view direction vector is perpendicular to image plane */
```

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

```
     $P(i, j) = P(0, 0) + i \cdot v \cdot \Delta i + j \cdot u \cdot \Delta j$ ;
```

```
    sum = 0;
```

```
    if (projectionMode == Perspective)
```

```
        ray =  $(P(i, j) - \text{eye}) / |P(i, j) - \text{eye}|$  /* normalized view direction vector */
```

```
    IntersectRayWithVolumeBoundingBox(V, ray, t_front, t_back);
```

```
    for(t = t_front; t ≤ t_back; t += stepSize) /* traverse the volume front to back */
```

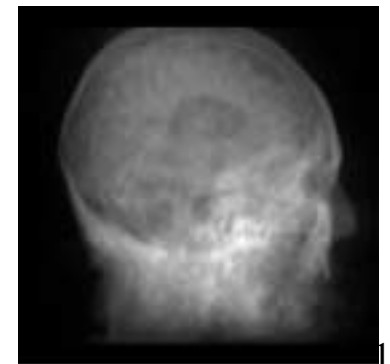
```
        sampleLoc =  $P(i, j) + t \cdot \text{ray}$  /* step along the ray */
```

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

```
        sum += intVal · stepSize; /* add interpolated value to X-ray sum */
```

```
    I(i, j) = sum;
```

```
NormalizeImage(I);
```



# MIP Rendering Algorithm

**RenderMIP(Volume V, int stepSize, Image I)**

if (projectionMode == Orthographic)

ray =  $u \times v / |u \times v|$  /\* view direction vector is perpendicular to image plane \*/

for each image pixel (i, j)

$P(i, j) = P(0, 0) + i \cdot v \cdot \Delta i + j \cdot u \cdot \Delta j$ ;

if (projectionMode == Perspective)

ray =  $(P(i, j) - \text{eye}) / |(P(i, j) - \text{eye})|$  /\* the ray direction vector, normalized \*/

max = 0;

IntersectRayWithVolumeBoundingBox(V, ray, t\_front, t\_back);

for(t = t\_front; t ≤ t\_back; t += stepSize) /\* traverse the volume front to back \*/

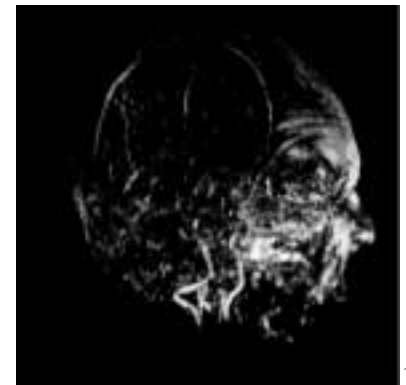
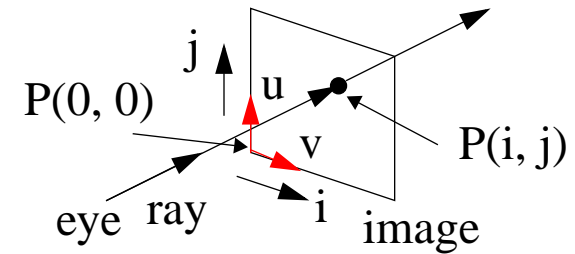
sampleLoc =  $P(i, j) + t \cdot \text{ray}$  /\* step along the ray \*/

intVal = Interpolate(V, sampleLoc);

if(intVal > max)

max = intVal;

I(i, j) = max;

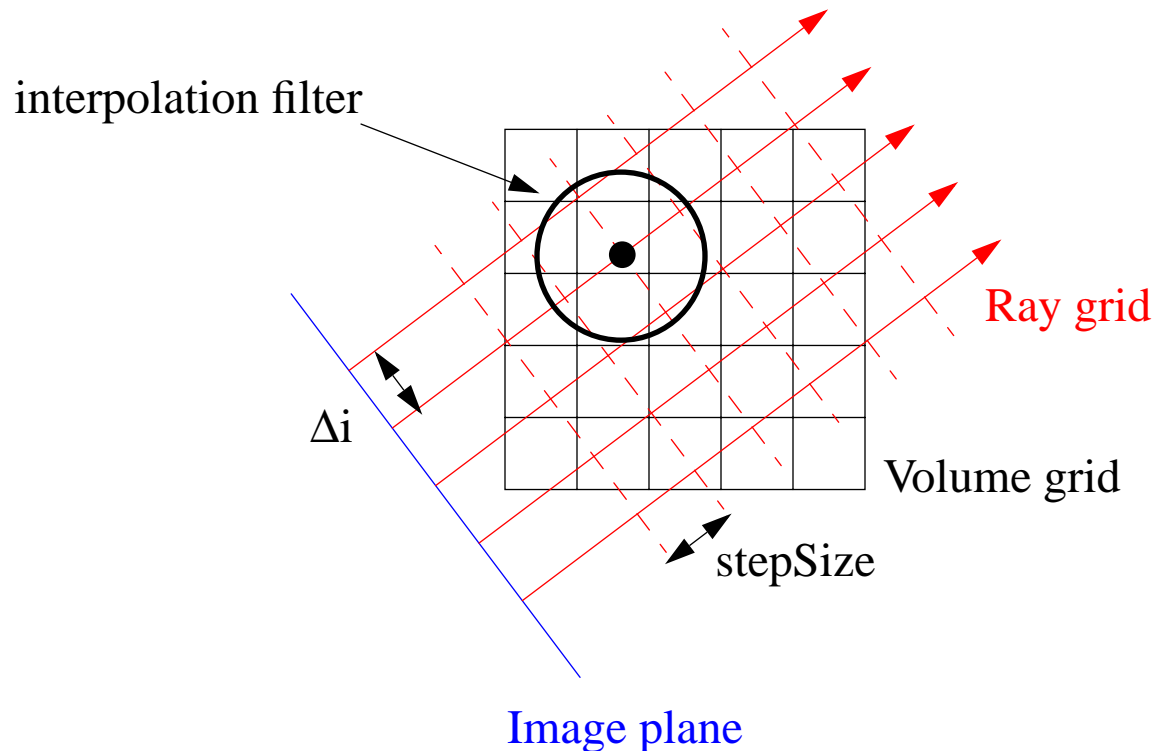


## Some Things To Think About....

- What is the minimum sampling rate (maximum sampling distance) along a ray (the ray stepsize)?
- What is the minimum sampling rate (maximum pixel distance) along the image axes ( $\Delta_i$ ,  $\Delta_j$ )?
- What should we do if either ray stepsize,  $\Delta_i$ , or  $\Delta_j$  fall above the permissible limits?
- What do we need to watch out for in perspective projection?
- What about errors resulting from the trapezoidal ray integration rule?
- How do we render isotropic rectilinear grids where the grid distances are not identical?

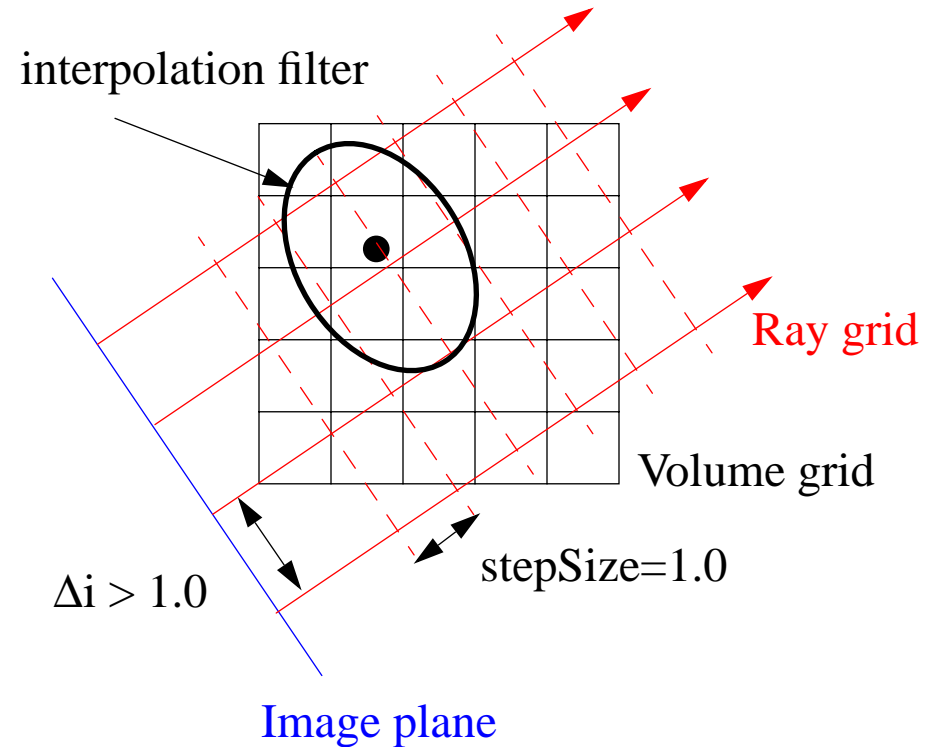
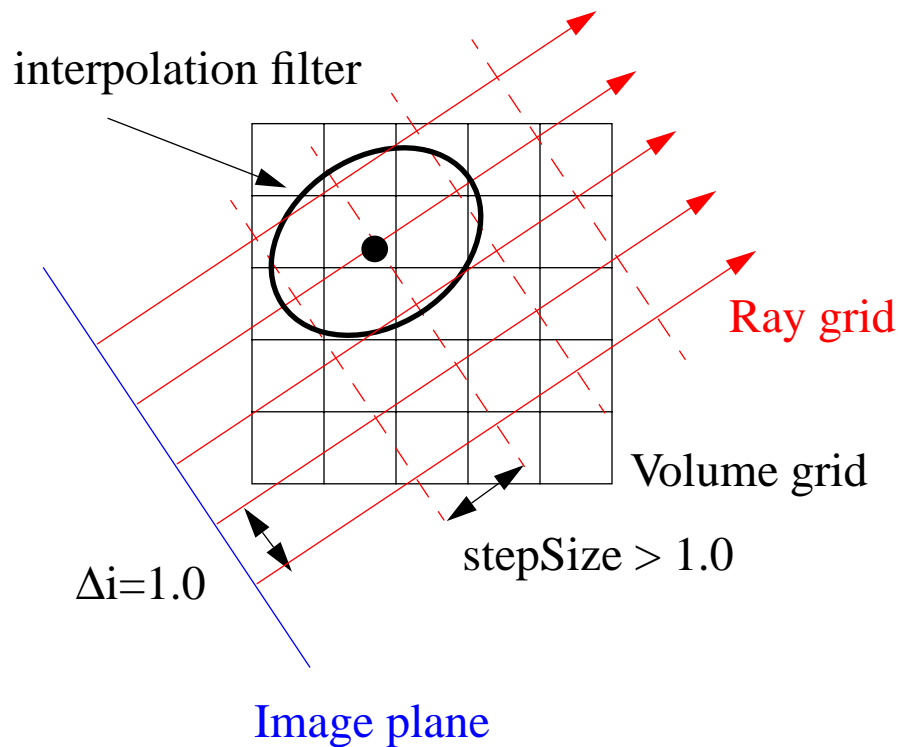
# Sampling Aspects Of The Image Formation Process

- Think of the image formation process as a resampling task followed by sample combination:
  - first, the rays resample the volume grid (image) into a new ray grid (image)
  - then, the values in the ray grid (image) are combined (e.g. integrated) to form the image
- Hence, we need to follow the rules imposed by the sampling theorem stated earlier
  - if  $\Delta i, \Delta j, \text{stepSize} \leq 1.0$ , then we can use a standard interpolation filter (no stretching needed)
  - for ease of illustration, assume a radially symmetric interpolation filter (e.g., a Gaussian)



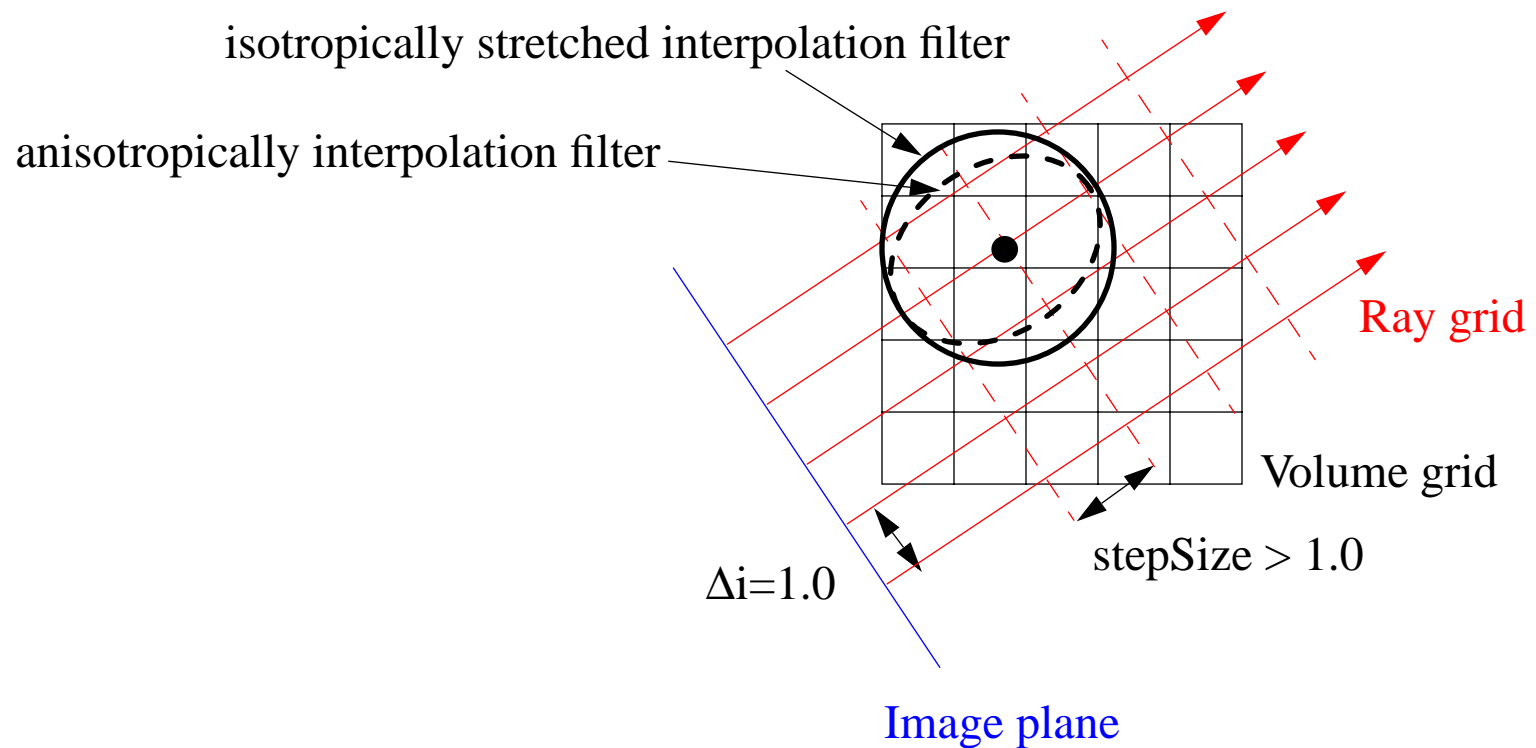
## $\Delta i, \Delta j, \text{ Or StepSize} \geq 1.0$ (1)

- We need to stretch the interpolation filter in the undersampled direction for lowpassing
- Stretching factor is determined by the increase in  $\Delta i, \Delta j,$  or  $\text{stepSize}$ 
  - recall, one needs to scale the filter amplitude accordingly (in the stretched direction)



## $\Delta i, \Delta j, \text{ Or StepSize} \geq 1.0$ (2)

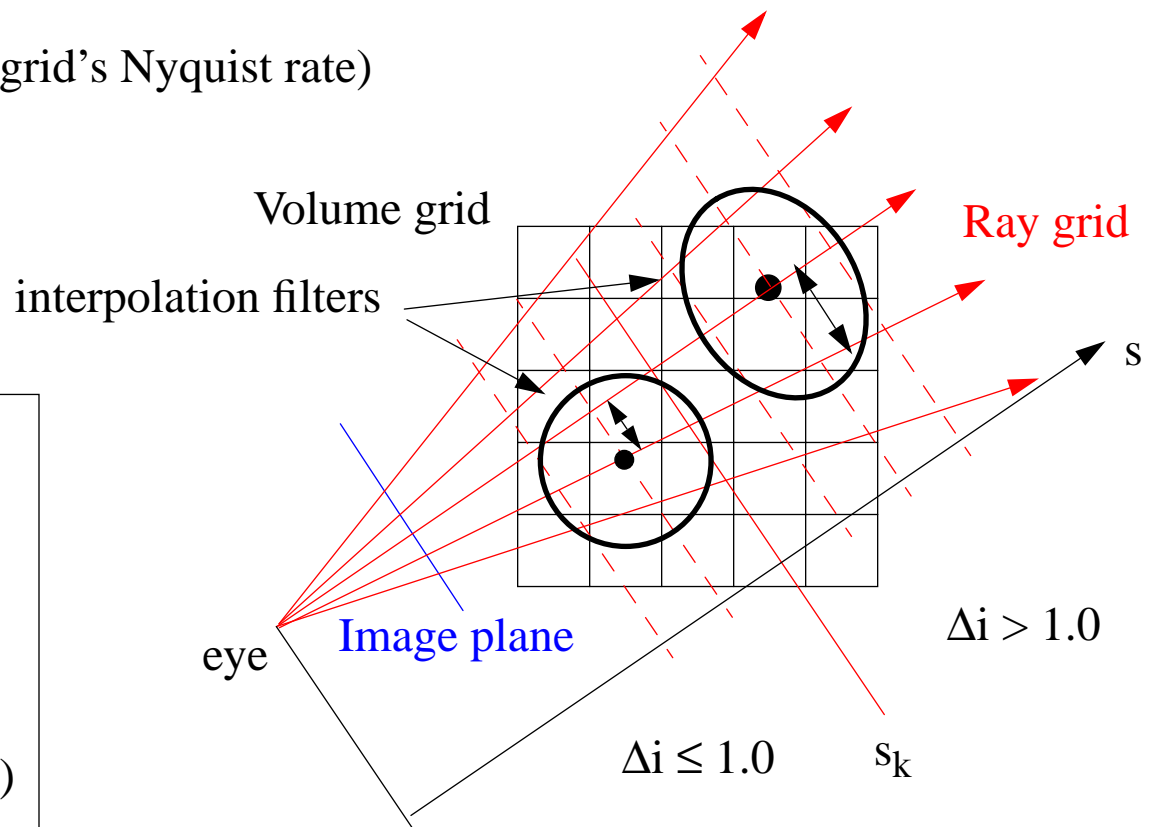
- Why not stretch in all directions according to  $\text{Max}(\Delta i, \Delta j, \text{ Or StepSize})$ ?



- This would cause too much lowpassing in the sufficiently sampled direction (but no aliasing)
- This may, however, be acceptable in many applications

# Perspective Projection

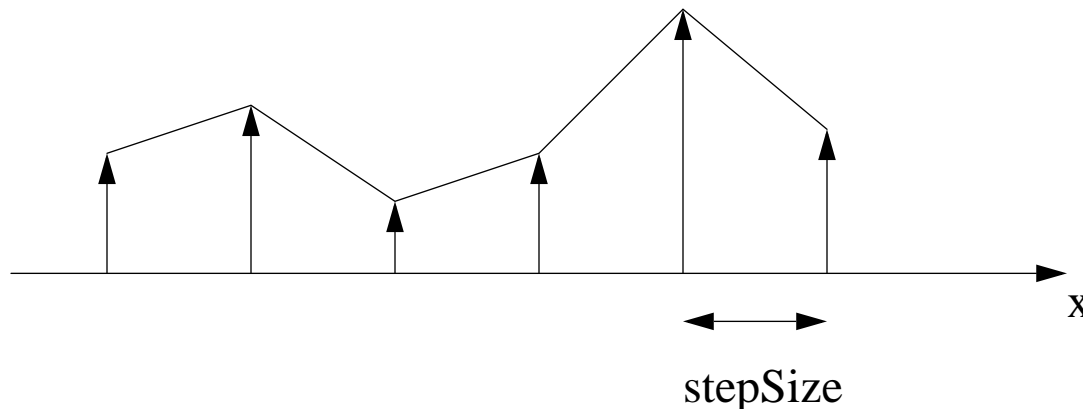
- In perspective projection,  $\Delta i$  and  $\Delta j$  grow as a function of distance from the eye point
- There is a distance  $z_k$  which partitions the required filter size:
  - for ray samples with  $s \leq s_k$ ,  $\Delta i$  and  $\Delta j \leq 1.0$  and we can use the standard interpolation filter  
(no aliasing since we sample above the grid's Nyquist rate)
  - for rays samples with  $s > s_k$ ,  $\Delta i$  and  $\Delta j > 1.0$  and we should use a stretched interpolation filter  
(we now sample below the grid's Nyquist rate)



- for  $s > s_k$ :  
kernel stretch factor =  $s / s_k$
- Simple approximation:  
stretch kernels along the axis  
directions most orthogonal to  
center ray (still use factor  $s / s_k$ )

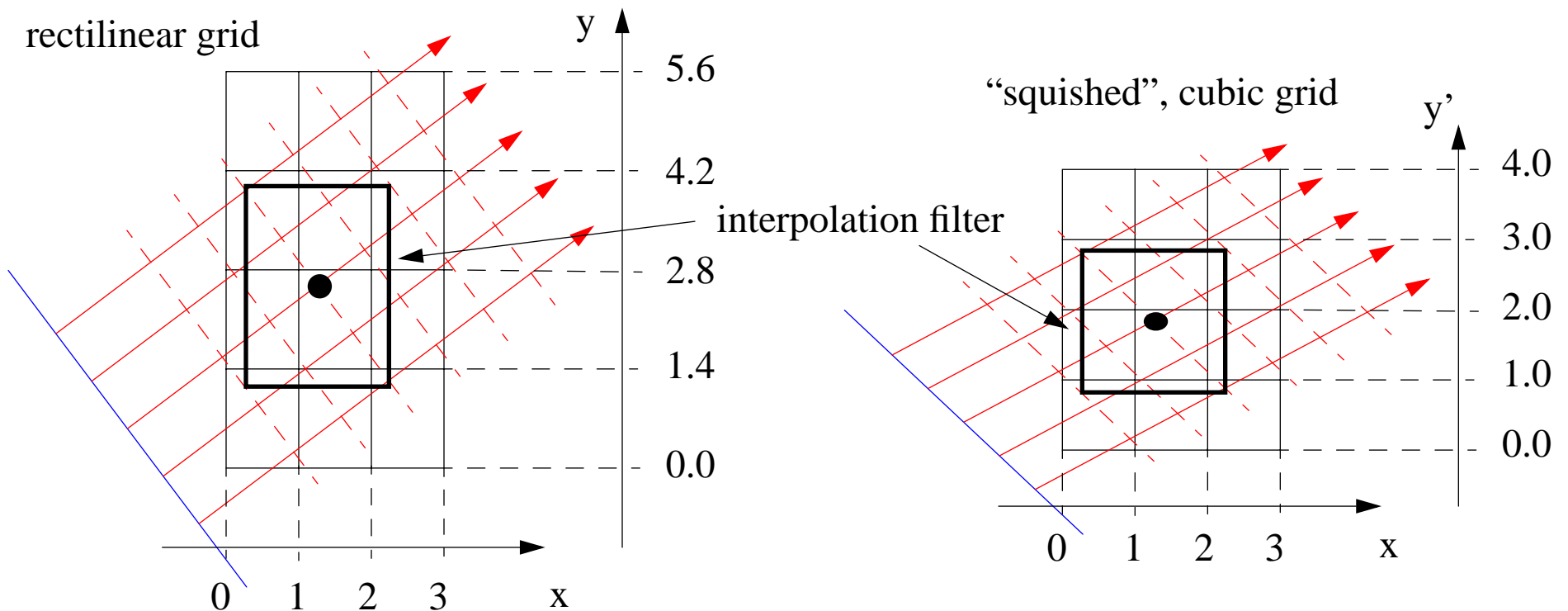
# Errors Due To Discrete Integration

- We approximate the continuous X-Ray integral by a discrete sum of point samples
- This is equivalent to numerical integration via the trapezoidal rule
- The error is  $\sim \frac{stepSize^3}{12} \cdot f''(\xi)$ 
  - so the smaller we choose the ray stepsize, the better the integration
  - there is no error if the underlying function is linear (at least between sample points)
- Higher order integration (e.g., Simpson's), involving larger sample neighborhoods, are possible
- We will see later, a rendering algorithm called splatting will provide the highest accuracy



# Rendering Isotropic Rectilinear Grids

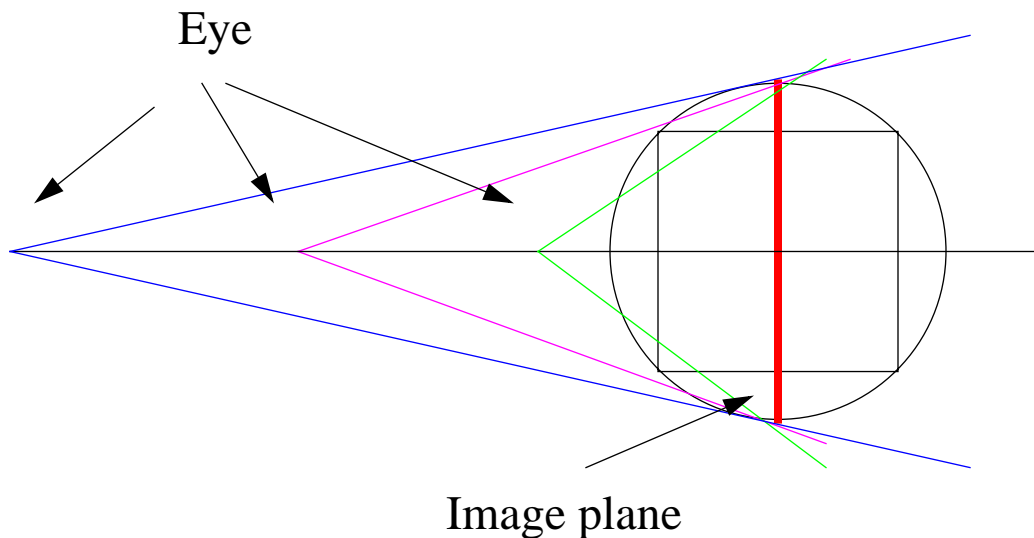
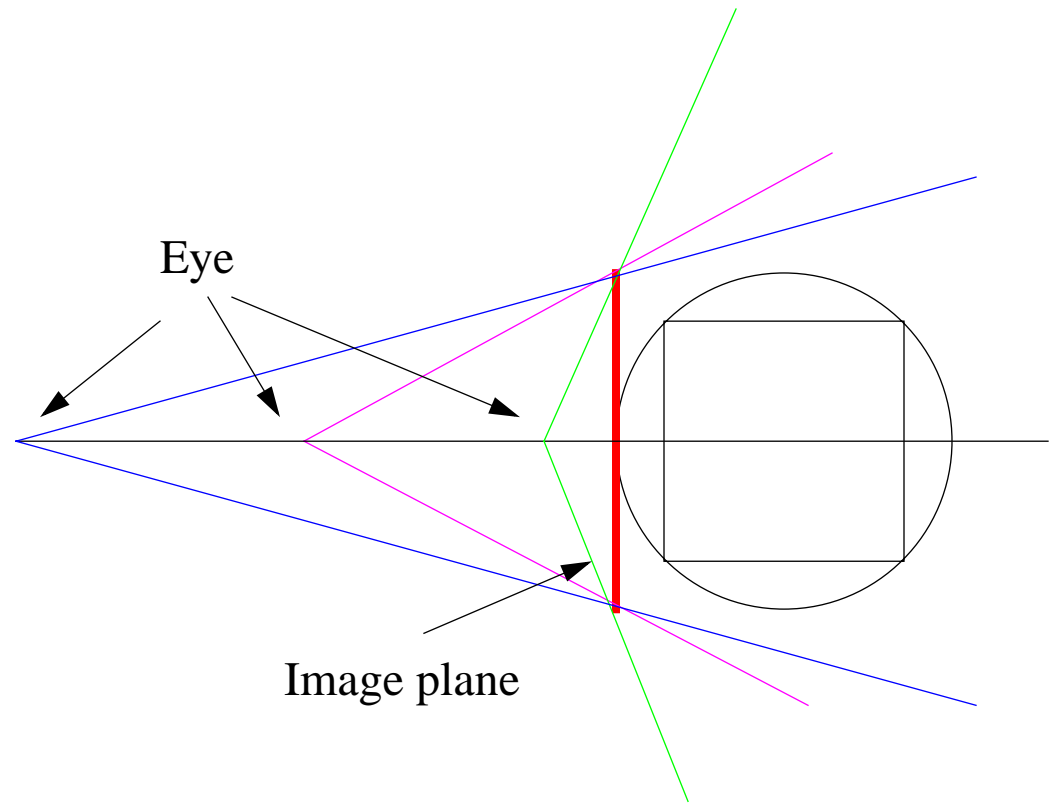
- Solution 1: Sample with anisotropic interpolation kernels in rectilinear space
  - note: NO amplitude scaling is required since now the kernel extent is determined by the underlying volume grid (not by the ray sampling grid)
- Solution 2: squish the grid into a cubic grid and use an isotropic interpolation kernel
  - note: need to squish the image plane and the rays as well
  - this changes the orientation of both the image plane and the rays, as well as stepSize,  $\Delta_i$ ,  $\Delta_j$



Possible setup for perspective:

The image plane touches the bounding sphere.

Problem: When the eye moves closer, the object appears smaller (which is somewhat counter-intuitive)



Better setup (easy fix, there are better ones):

Place the image plane in the center of the volume. Trace rays from the eye:

$$P = \text{Eye} + t \cdot r_{i, j}$$

This way the center slice will always have the same size on the screen.

Find  $t_{\text{front}}$  and  $t_{\text{back}}$  as usual.