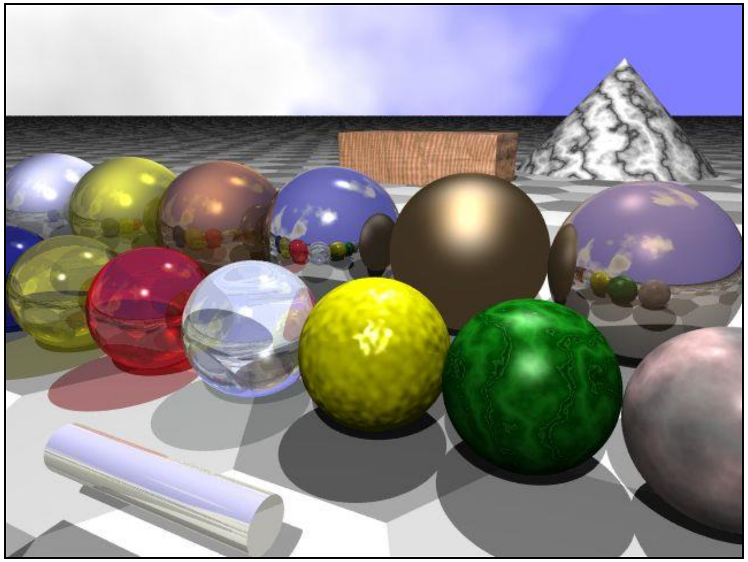
Traditional Computer Graphics

- Computer graphics deals primarily with *surface representations and rendering*
- Objects are defined by a surface or boundary representation
- Explicit distinction between the inside and the outside
- But the inside is empty – it has no substance



- A surface is infinitesimally thin – zero thickness
- This is just an approximation of reality – even a sheet of paper or a human hair has a thickness, however small
- Volume graphics includes a set of techniques for rendering and visualizing *volumetric data*, data that have interior information

Traditional Computer Graphics

Surface graphics advantages

+ fast rendering algorithms are available
+ acceleration in special hardware is relatively easy and cheap (~$100)
+ rich programming libraries like OpenGL and Microsoft's Direct3D make it easy to develop surface graphics applications
+ surface realism can be added via *texture mapping*

Traditional Computer Graphics

Surface graphics disadvantages

– Discards the object’s interior and just maintains a thin shell

– Does not facilitate real-world operations such as cutting, slicing, dissection

– Does not enable artificial viewing modes such as semi-transparency, X-ray to let us peer *into* and *through* objects

– *Amorphous phenomena* like clouds, fog, and gas are hard to represent – amorphous = “without shape”

- Why would it be hard to represent such phenomena using only thin shells as our core representation?
- Such objects have no definite boundaries

Volume Graphics Definitions

- A volume is sometimes called a *3D image*
  - Compare: image = 2D grid of intensities, volume = 3D grid of densities (also sometimes called intensities)
  - A 3D array of point samples called *voxels* (volume elements)
  - A 2D image on the computer screen is basically a 2D array of point samples
  - 2D image: each pixel has a position on the screen (*x*,*y*) and an intensity (*d*)
  - 3D image: each voxel has a position in space (*x*,*y*,*z*) and a density (*d*)
  - *Volume rendering* techniques basically perform computations over these 3D densities to produce an image

Real-World Volume Rendering Example

- An X-ray machine is actually a type of volume rendering architecture
- X-rays pass through the body (a volumetric data-set!) and produce a 2D image
- Some rays pass through the body and strike X-ray film
- Others are absorbed (d’oh!)
- Others are deflected
- This idea of firing rays through a volumetric (3D) space and collecting information along the way is the basic idea and process used in volume rendering

Volume Rendering

- In volume rendering, imaginary rays are passed through a 3D object that has been discretized (e.g., via CT or MRI)
- As these *viewing rays* travel through the data, they take into account of the *intensity* or *density* of each datum, and each ray keeps an accumulated value
- As the rays leave the data, they comprise a sheet of accumulated values
- These values represent the volumetric data *projected* onto a two-dimensional space (or image)
- Using transfer functions (mappings) we can highlight certain important features of the data-set using RGBα
- How was opacity (α) used in the image on the right? Hint: Is this what we look like from the outside? From the inside?
- Let's see a movie of stellar data (1)

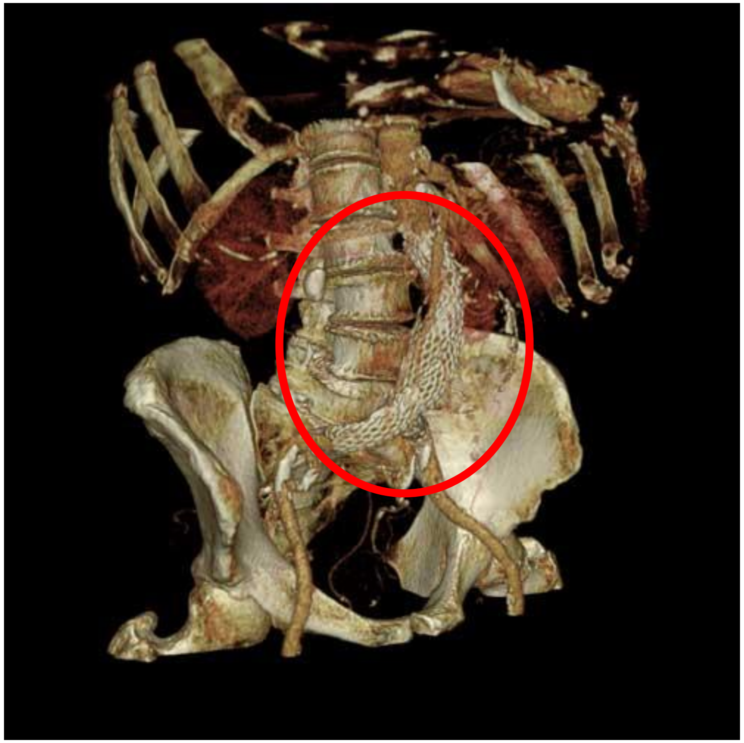
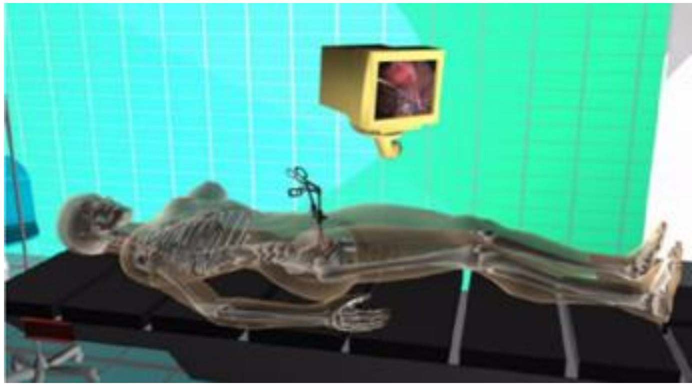


Image courtesy Viatronix, Inc.

Volume Rendering Applications – Medical Imaging

- Classic application for volume rendering since datasets are inherently volumetric
- Modalities are: CT, MRI, Ultrasound, others
- Doctors use volume rendering to visualize organs, structures, and tissue of interest
- Can render unimportant structures (semi-) transparently and emphasize important ones
- For example: render a brain tumor opaque and the surrounding brain tissue as a faint hull
- The medical check-up of the future:



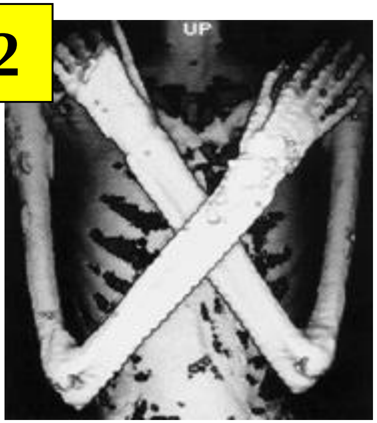
1. Get a full body scan with CT and MRI
2. Specialist doctors use volume visualization to investigate the state of the discretized patient:
3. A cardiologist checks coronary arteries for arteriosclerotic plaque
4. A radiologist flies through the virtual colon and checks for cancer, etc.
5. Simulate and plan a surgery or procedure on the digital patient if necessary
6. Keep the scan as a digital record of the patient for future reference

Volume Rendering Applications – Paleontology

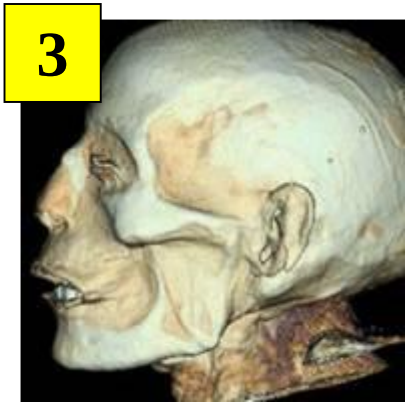
- Non-destructive exploration and dissection of:
- Prehistoric artifacts (dinosaur eggs, fossils embedded in soil)
- Artifacts from ancient cultures

1

2



3

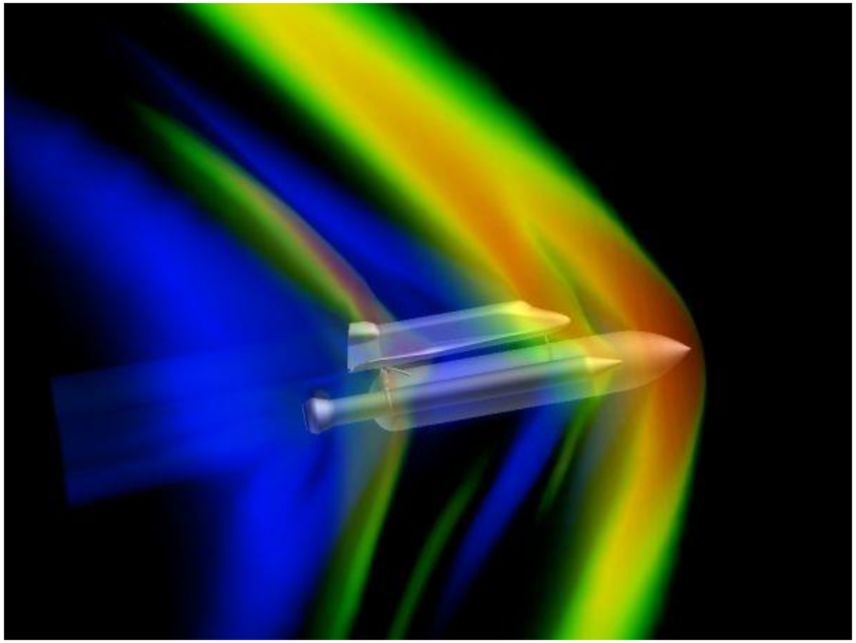


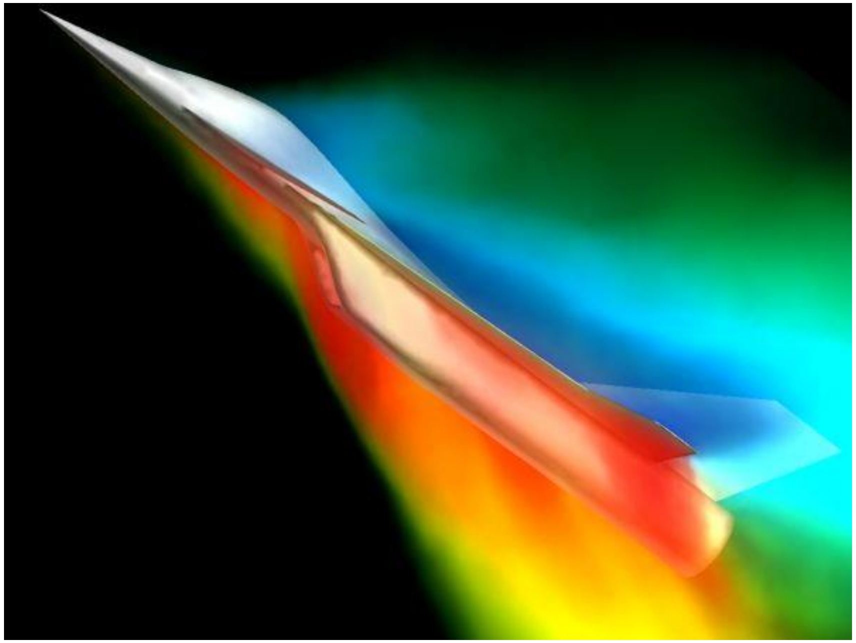
The new way

The old way

Volume Rendering Applications – CFD

- Computational fluid dynamics
- Fluid flow is governed by the Navier-Stokes system of simultaneous differential equations
- Velocity, pressure, temperature, viscosity
- These systems are solved using iterative methods on discrete grids
- The results are visualized with volume visualization



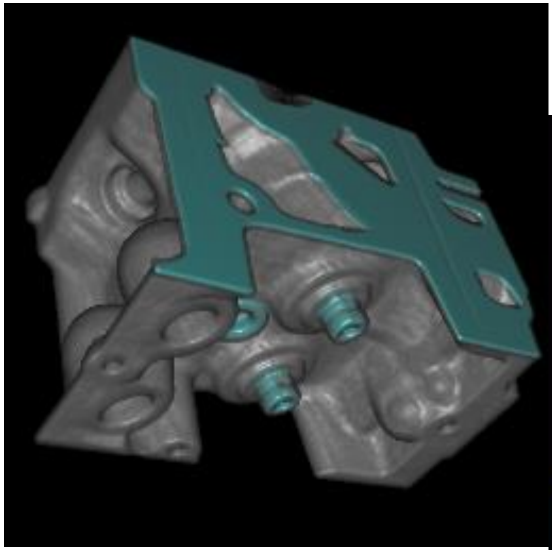


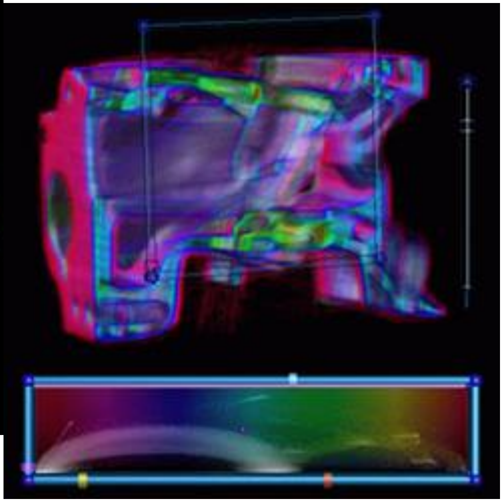
Volume Rendering Applications – Simulation

- CFD is an example of a simulation frequently performed using high-performance computers, often supercomputers
- Simulation is often extremely useful for understanding and predicting reality
- Sometimes it's impossible or too dangerous to setup a real-world experiment of that which you wish to view
- Let's look at two movies to see examples of these
- Tornado vortex simulation (11)
- Heptane fire simulation (3)

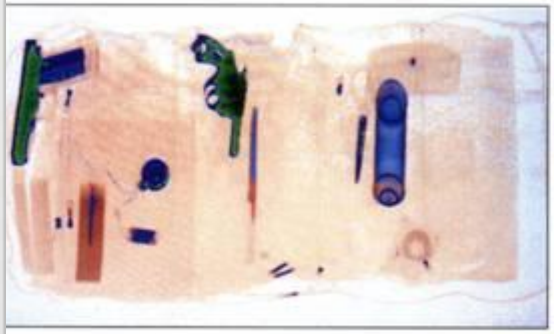
Volume Rendering Applications – Others

- Industrial CT:
- Reverse engineering
- Inspection for structural failures





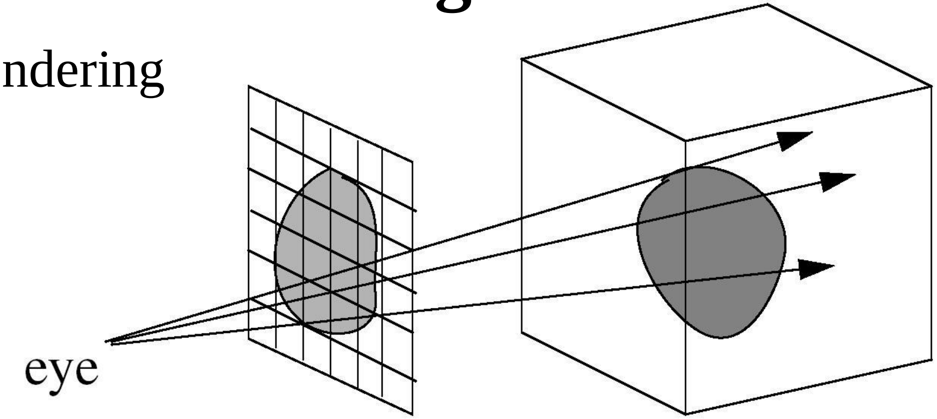
- Security:
- Airport luggage CT
- Search for drugs, weapons, etc.

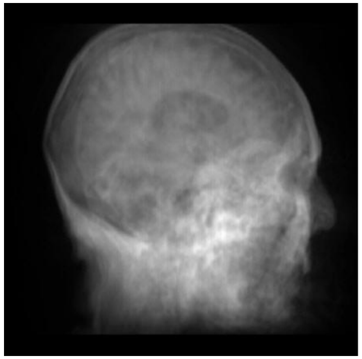


Typical Threat Objects
Dual Energy Organic/Inorganic Identification

Modes of Volume Rendering

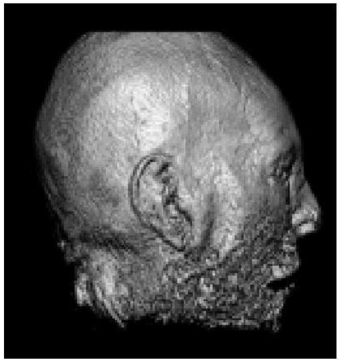
- We will explore a particular volume rendering algorithm called *ray-casting*
- We will explore other algorithms later in the semester
- Four main volume rendering *modes* exist:





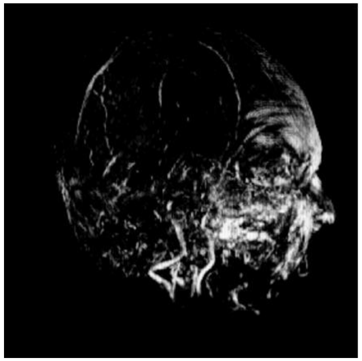
X-ray:

rays sum volume contributions along their linear paths



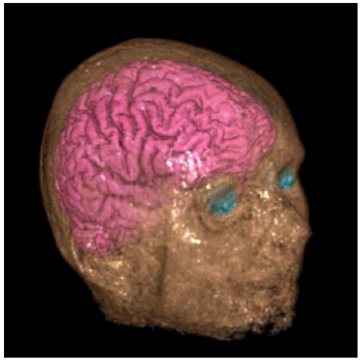
Iso-surface:

rays look for the object surfaces, defined by a certain volume value



Maximum Intensity Projection (MIP):

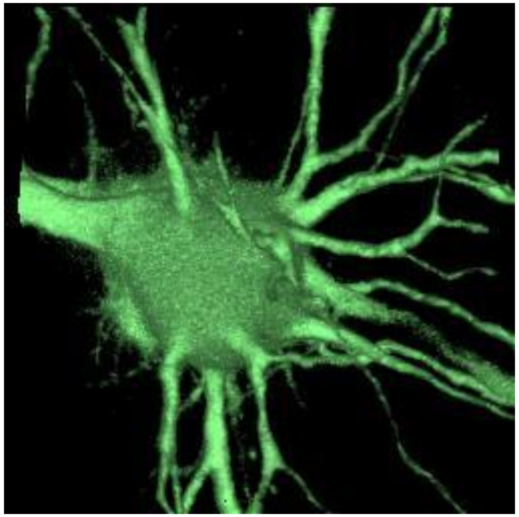
a pixel value stores the largest volume value along its ray

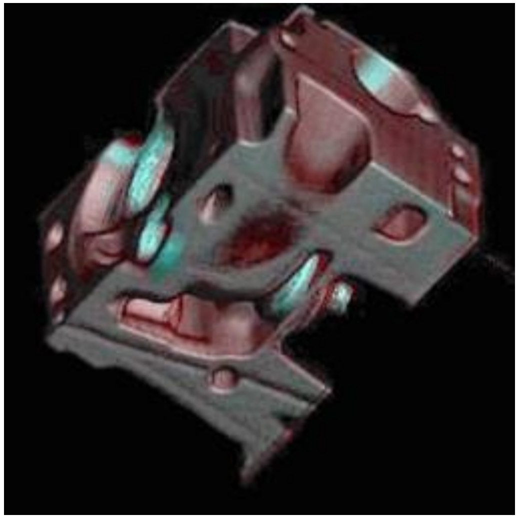


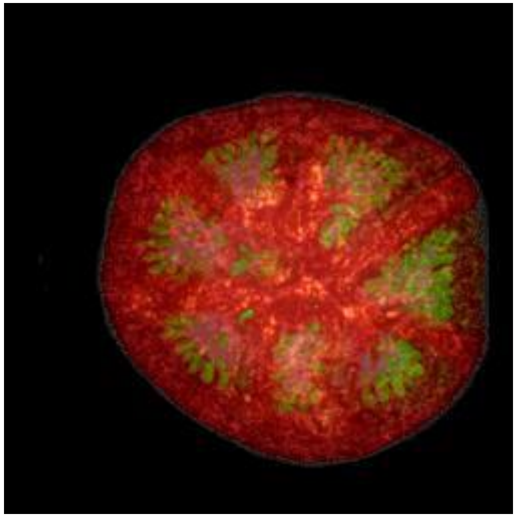
Full volume rendering:

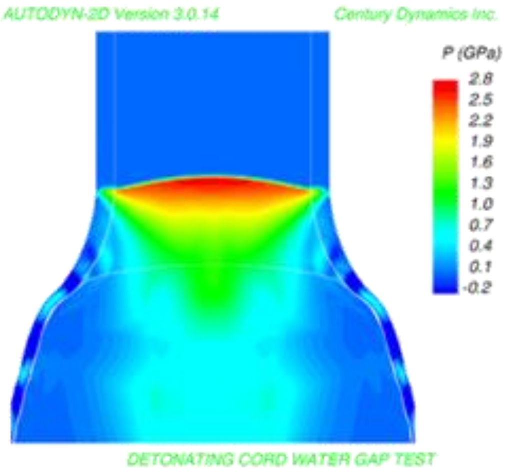
rays *composite* volume contributions along their linear paths

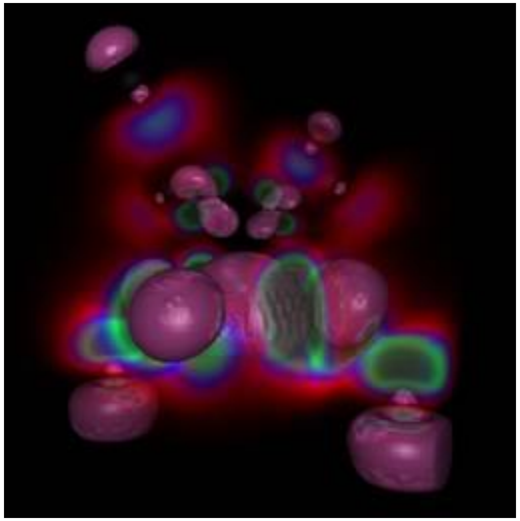
Volume Rendering Examples





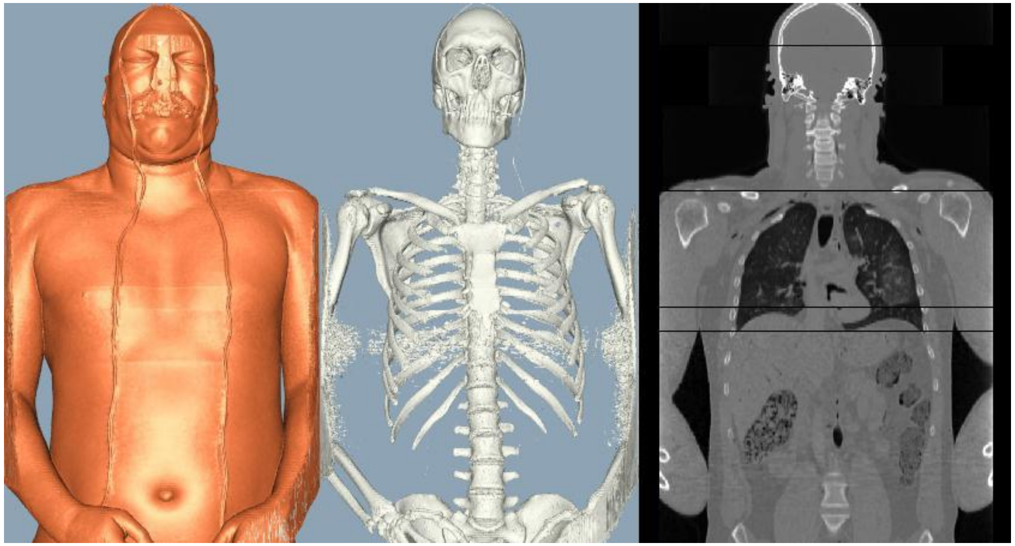






Famous Volumetric Data-Sets

- NIH (National Institutes of Health) Visible Human Project
- Visible Male and Visible Female
- http://www.nlm.nih.gov/research/visible/visible_human.html
- Visible Male:
- MRI and CT scans of a deceased prison who dedicated his body to science
- MRI slices taken at 4 mm intervals, 256 x 256 images, 12-bit grayscale
- CT slices taken at 1 mm intervals, 512 x 512 images, 12-bit grayscale
- Also, anatomical slices
- Cadaver frozen and physically sliced (!) in 1871 slices and photographed at 24-bit resolution color
- Used widely in research and education
- Movies of a few others (10)



Volume Graphics

- Advantages

+ Maintains a representation that is close to the underlying fully-3D object (but discrete) (What might be some problems associated with converting from a continuous representation to a discrete one?)
+ Can achieve a level of realism (and ‘hyper-realism’) that is unmatched by surface graphics
+ Allows easy and natural exploration of volumetric datasets

- Disadvantages

– High rendering complexity

– Hardware acceleration is complex and expensive (~$3000)

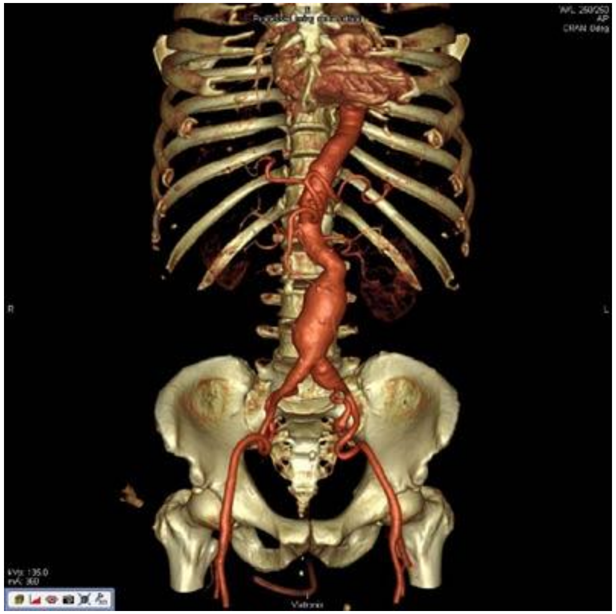
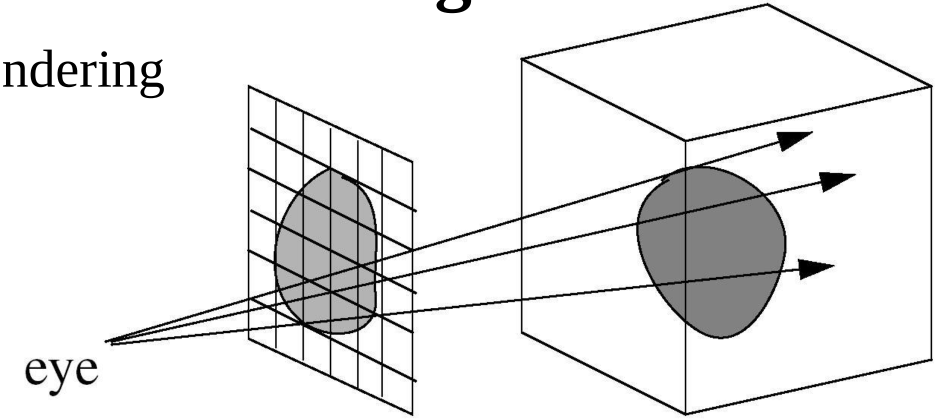
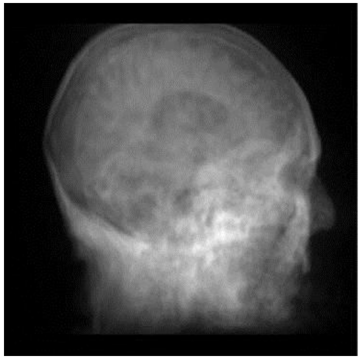


Image courtesy Viatronix, Inc.

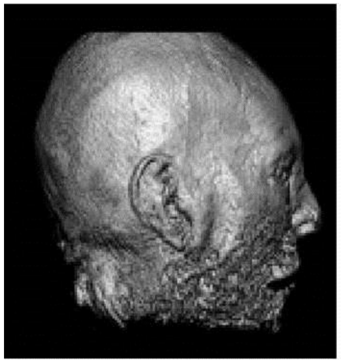
Modes of Volume Rendering

- We will explore a particular volume rendering algorithm called *ray-casting*
- We will explore other algorithms later in the semester
- Four main volume rendering *modes* exist:

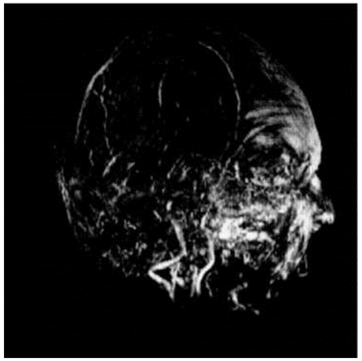




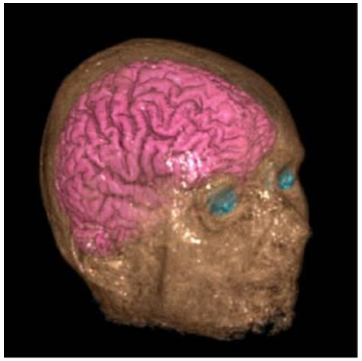
X-ray:
rays sum volume contributions along their linear paths



Iso-surface:
rays look for the object surfaces, defined by a certain volume value



Maximum Intensity Projection (MIP):
a pixel value stores the largest volume value along its ray

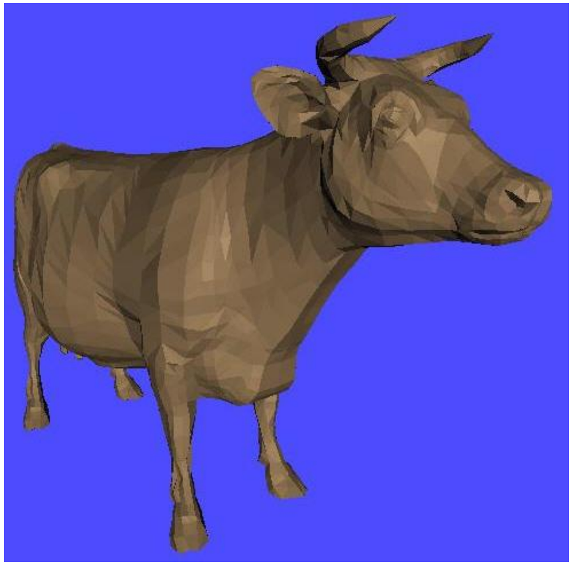


Full volume rendering:
rays *composite* volume contributions along their linear paths

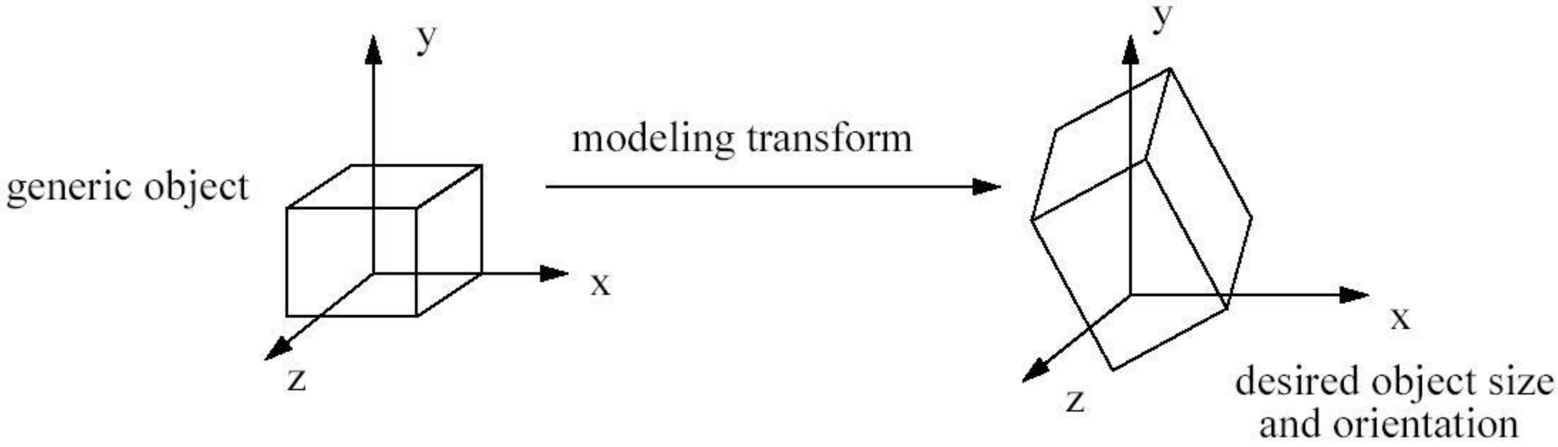
Volume Rendering Framework

- Before we study how to perform volume rendering via ray-casting, we need to review some 3D geometry and its relationship with linear algebra
- In particular, we need to have a concise mathematical framework for indicating the position (x,y,z) and orientation (via rotation) of a volumetric data-set so that we can volume render it from all sides
- We also have to examine how to specify the viewing position, direction and orientation of the camera or eye
- For instance, if we took two X-rays of your body, one from the front and one from the side, the resulting images would be very different
- We need to be able to perform similar *spatial transformations* on volumes so that we can visualize the data from different viewing angles
- These transformations take place with respect to some coordinate system (e.g., translate an object by 3 units in the x direction; rotate an object 45° around the z-axis)

Object Representations



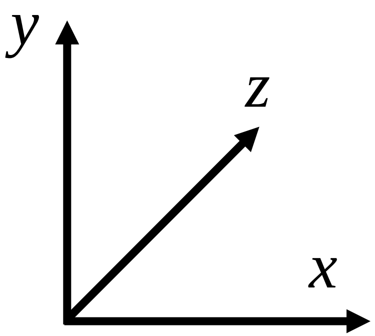
- An object may be represented in one of many ways:
- A vertex and polygon list
- A list of voxels (usually regular grids)
- A mesh structure (usually irregular grids)
- Many others…take CSE 530 ☺
- These generic objects can now be scaled, rotated, translated to fit the desired shape (*model space*)
- This is called the *modeling transformation*:



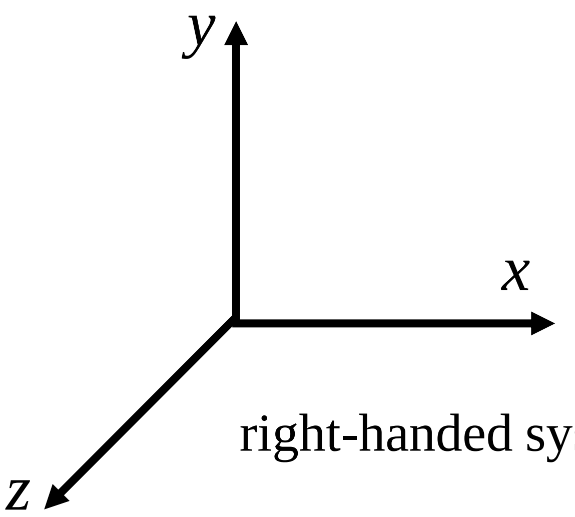
- We then place the object in the world using the *object-to-world transformation*

Coordinate Systems

- 3D objects we wish to display on a computer screen must have an (x,y,z) position in Euclidean space (usually called the *scene* or the *world space*)
- Two kinds of coordinate systems: right-handed and left-handed
- We will use whichever coordinate system seems most natural in the given context
- In the absence of transformations, it is very easy to convert between the two
- If the object has been rotated or otherwise transformed, however, a little extra work is necessary to make sure the transformations are also converted properly from one coordinate system to the other



left-handed system



right-handed system

Spatial Transformations

- Let’s assume for the moment we are using a right-handed system (this is what OpenGL uses by default)
- We need ways to *translate*, *rotate* and *scale* 3D objects
- Such transformations can be concisely defined using 4x4 matrices:

$$\begin{bmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{bmatrix}$$

- The values for *a*-*p* depend on the particular transformation
- Usually, the elements *d*, *h* and *l* are 0, and *p* is 1

Transforming a Spatial Location

- Now, if we want to transform a point at position (*x*,*y*,*z*) using the transformation matrix, it's just a simple matter of multiplying the current position by the matrix to get the new position (*x'*,*y'*,*z'*):

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$x' = ax + ey + iz + m$$

$$y' = bx + fy + jz + n$$

$$z' = cx + gy + kz + o$$

Identity Transformation

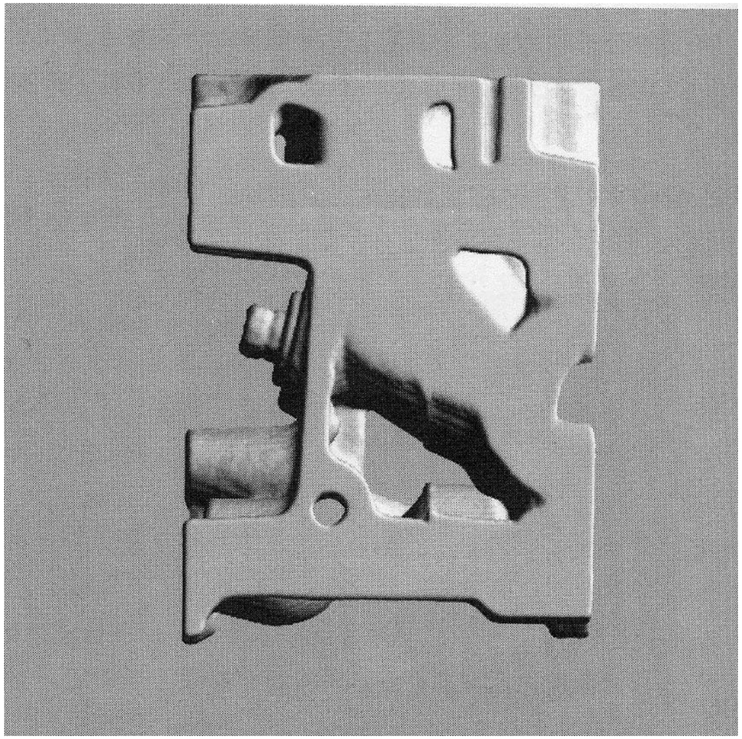
- Simplest transformation is the identity transformation
- Causes no change in position or orientation of object

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$x' = x$$

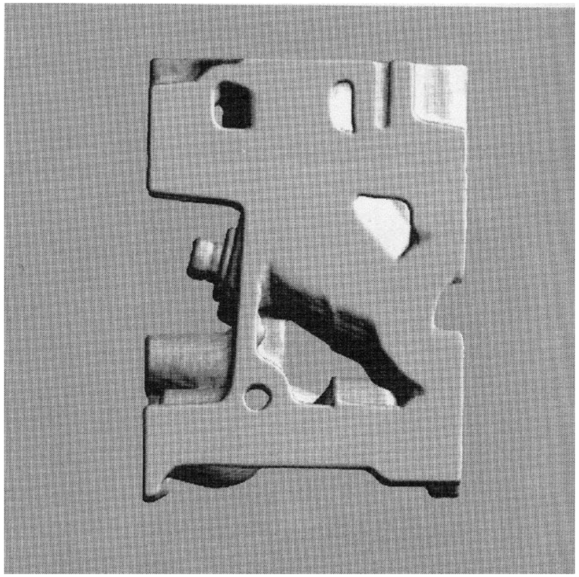
$$y' = y$$

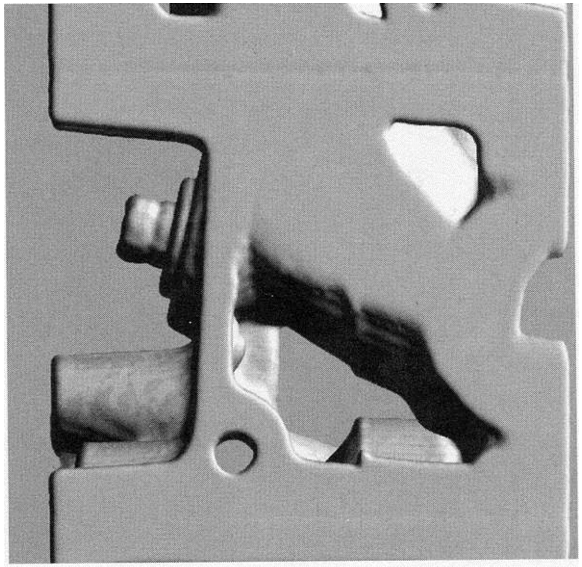
$$z' = z$$



$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling Transformation





$$x' = s_x x$$

$$y' = s_y y$$

$$z' = s_z z$$

- When $s_x = s_y = s_z$, we call it *uniform scaling*
- If $s > 1$, the object looks bigger; if $0 < s < 1$, object looks smaller
- What if $s = 0$? $s < 0$?

Rotation Transformations

about *x* axis

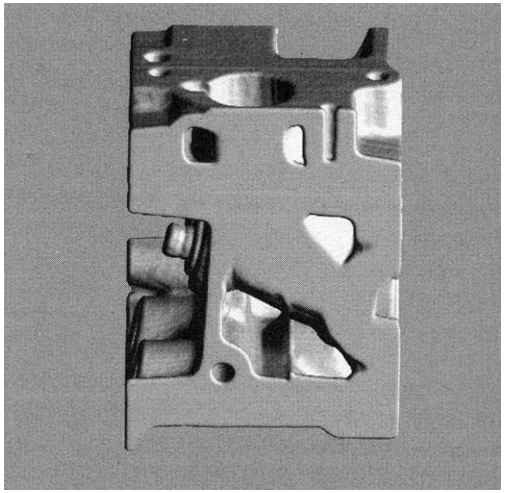
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

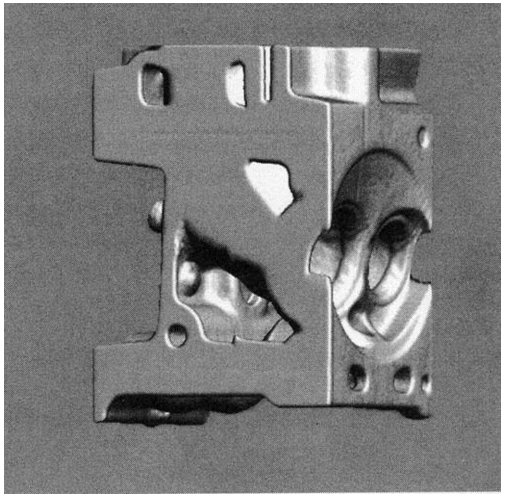
about *y* axis

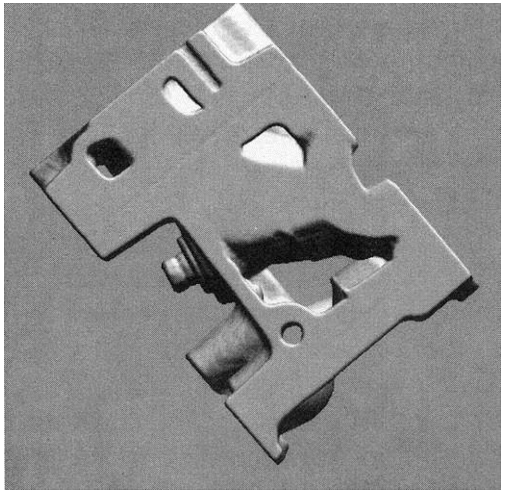
$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

about *z* axis

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



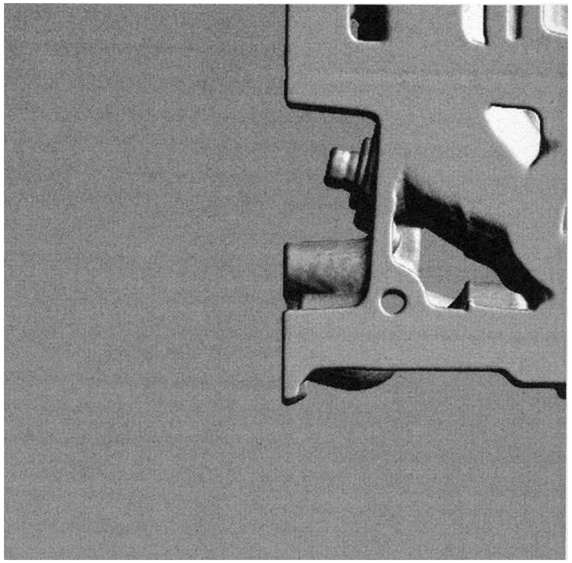




Translation Transformation

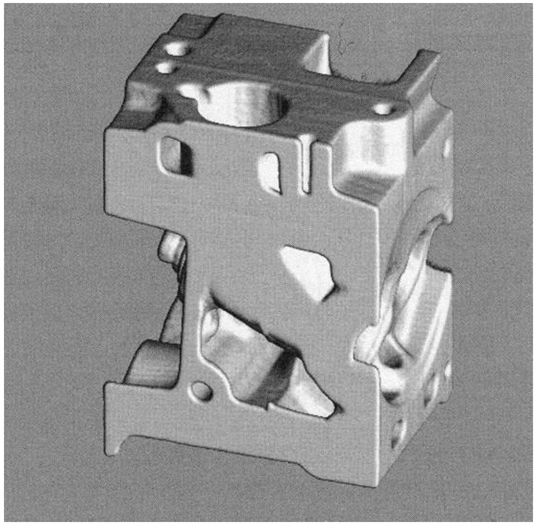
- You might be wondering why we don’t just use 3x3 matrices since the last column and row are [0 0 0 1]
- The reason is that translation of a point cannot be expressed as a 3x3 vector-matrix multiplication
- Translation really just adds a displacement: $x+t_x$, $y+t_y$, $z+t_z$
- This problem can be solved using *homogeneous coordinates*
- The idea is to express our point in 4D and then project it back to 3D

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



Composite Transformations

- One very convenient aspect of using 4x4 matrices to encode transformations is that they can be *composed*
- Given transformations $\mathbf{M}_1$ and $\mathbf{M}_2$, we can express the result of performing transformation $\mathbf{M}_2$ followed by $\mathbf{M}_1$ as $\mathbf{M}_1(\mathbf{M}_2\mathbf{p})$ where $\mathbf{p}$ is the point to transform
- Note that we must *right-multiply*, not left-multiply! For this reason, in computer code we program the transformations in the reverse order of what we want
- Also note that matrix multiplication is not commutative, so $\mathbf{M}_1\mathbf{M}_2\mathbf{p} \neq \mathbf{M}_2\mathbf{M}_1\mathbf{p}$
- Example: rotation & translation



Viewing Transformations

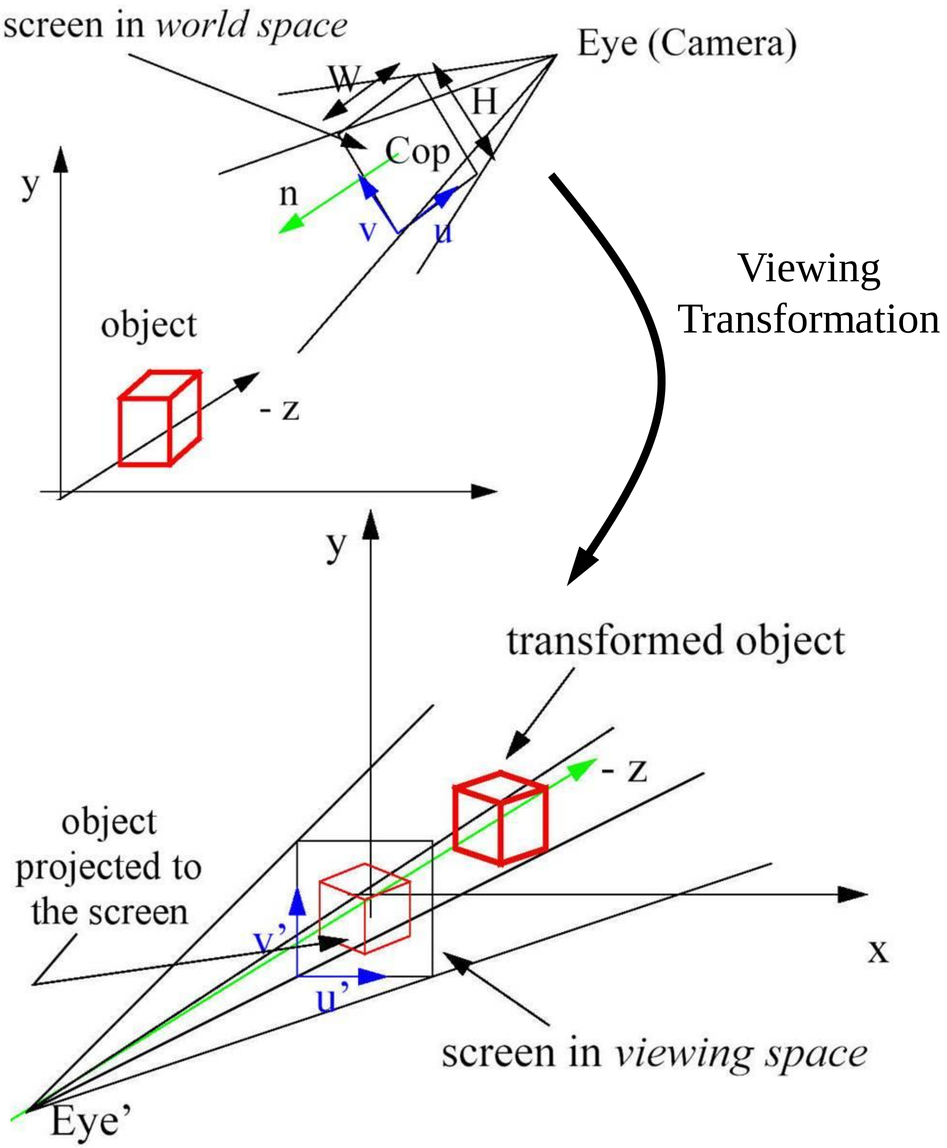
- So far we have been transforming the position and orientation of objects
- Suppose instead we transform the position and orientation of the *camera* or *eye*? (i.e., the *viewing* position and orientation)
- Compare:

1. Grabbing an object, turning it around in your hand, moving it around in space
2. Walking around a stationary sculpture, turning your head, looking up and down

- Both achieve the same effect: we can examine some 3D object from different vantage points, distances, etc.
- We can transform the object or the view – the effect is the same
- Models (objects) are transformed by *modeling transformations*, the camera is transformed by *viewing transformations*
- In OpenGL, modeling and viewing transformations are combined into the notion of a *modelview* matrix. In other words, use whichever paradigm seems more natural in a given circumstance and application

3D Viewing: Camera Metaphor

- A view is specified by:
  - eye position (Eye)
  - view direction vector (n)
  - screen center position (Cop)
  - screen orientation (u, v)
  - screen width and height (W, H)
- u, v, n are orthogonal to each other
- All objects are transformed by the viewing transform
- After the viewing transformation:
  - the screen center is at the coordinate system origin
  - the screen is aligned with the x, y axis
  - the viewing vector points down the negative z-axis
  - the eye is on the positive z-axis



Viewing Transformation Steps

- The sequence of transformations is:

1. Translate the screen Center of Projection (Cop) to the coordinate system origin (T_{view})
2. Rotate the translated screen such that the view direction vector n points down the negative z-axis and the screen vectors u, v are aligned with the x, y-axis (R_{view})

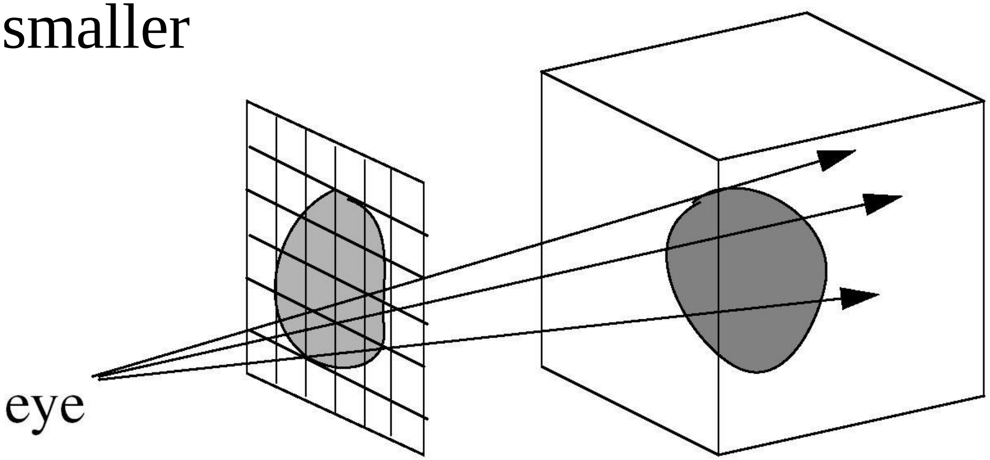
- We get $M_{view} = R_{view} \cdot T_{view}$
- We transform all object (points, vertices) by M_{view}:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -Cop_x \\ 0 & 1 & 0 & -Cop_y \\ 0 & 0 & 1 & -Cop_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Now the objects are easy to project since the screen is in a convenient position
- But first we have to account for perspective distortion…
- Perspective *what*?

Projection Transformations

- Now we need to see how to *project* a transformed object onto the screen so we can view it
- This process is conducted using a *projection transformation*, which is also defined as a 4x4 matrix
- We will look at two types: *perspective projection* and *orthographic projection* (also known as *parallel projection*)
- Perspective projection is like our own vision: distant objects appear smaller
- In parallel projection, this perspective effect doesn't occur – distance does not affect perceived object size

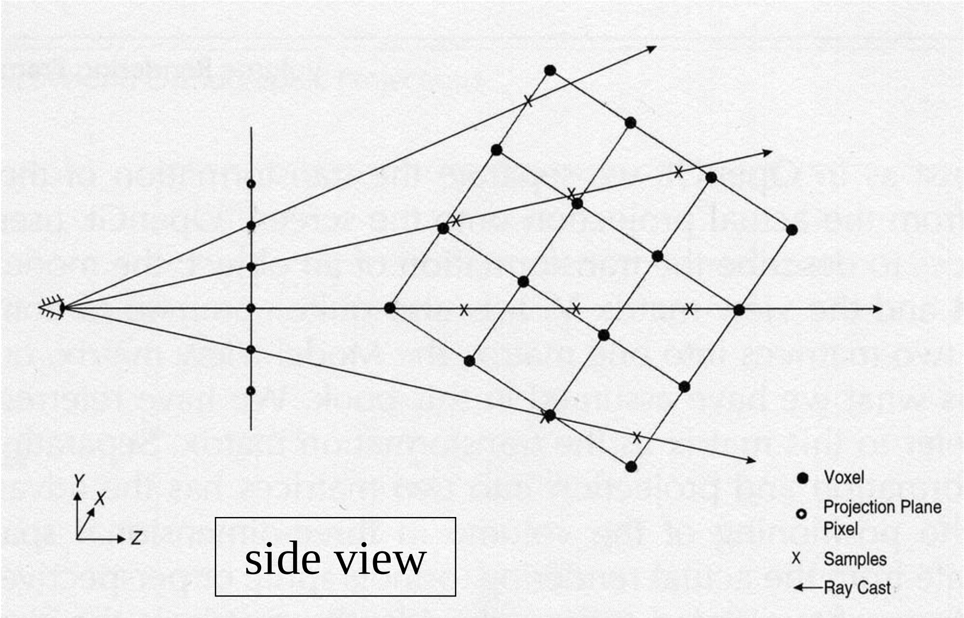


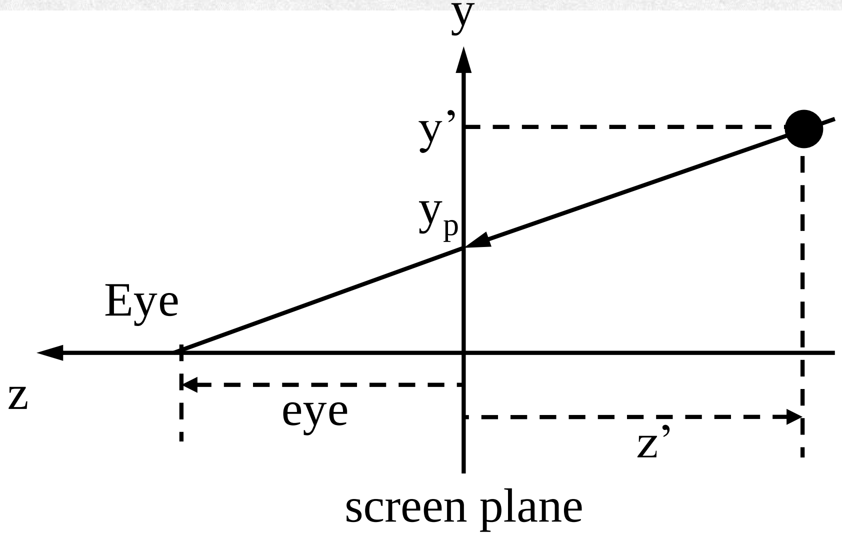
Perspective Projection

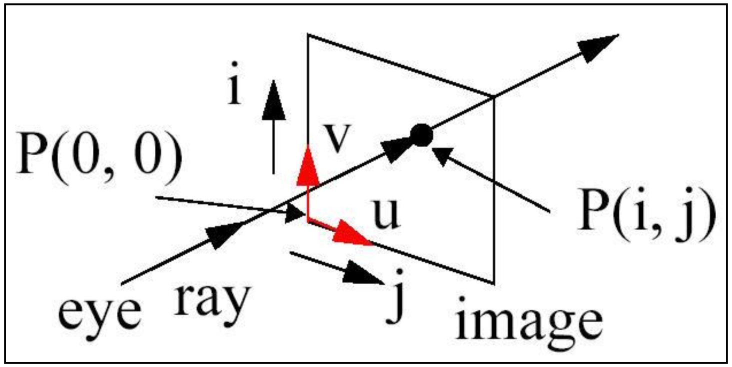
- A view-transformed vertex with coordinates (x', y', z') projects onto the screen as follows:

$$y_p = y' \cdot \frac{eye}{eye - z'} \qquad x_p = x' \cdot \frac{eye}{eye - z'}$$

- Note: $z' < 0$
- x_p and y_p can then be used to determine the *screen coordinates* of the object point (i.e., where to plot the point on the screen)







Perspective Projection

- Perspective projection can also be captured in a matrix M_{proj} with a subsequent *perspective divide* by the homogeneous coordinate *w*:

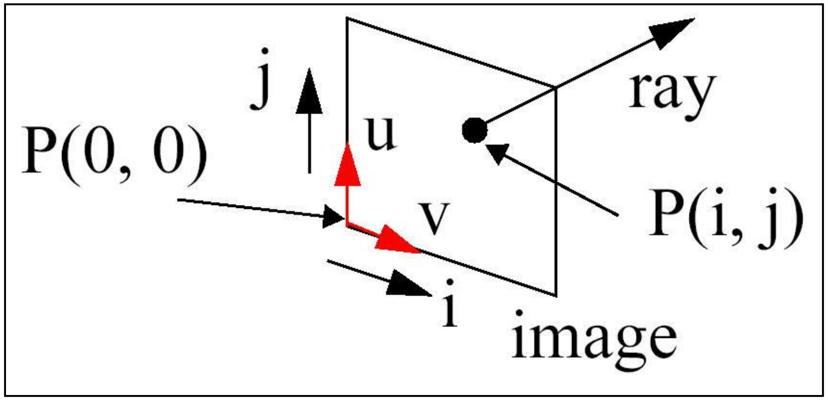
$$\begin{bmatrix} x_h \\ y_h \\ z_h \\ w \end{bmatrix} = \begin{bmatrix} eye & 0 & 0 & 0 \\ 0 & eye & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & eye \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} \qquad x_p = \frac{x_h}{w} \quad y_p = \frac{y_h}{w}$$

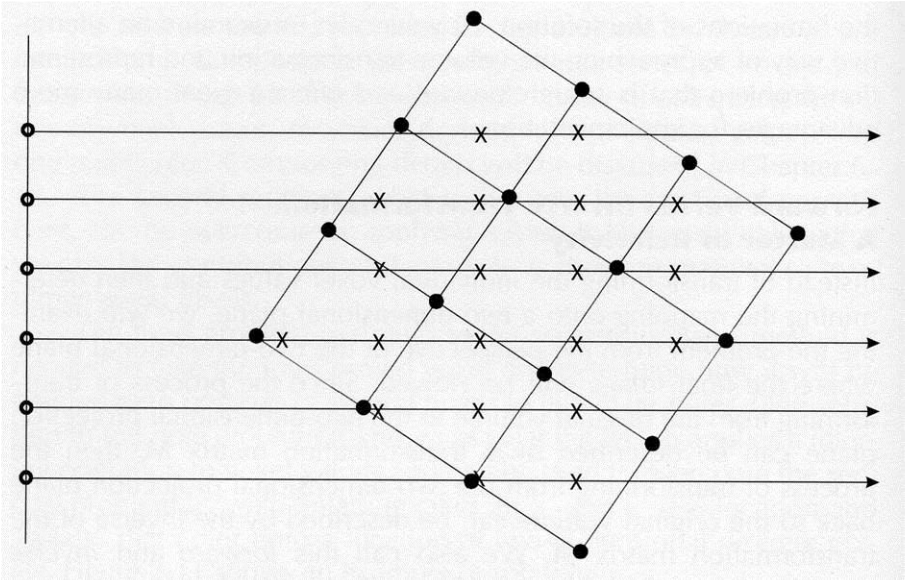
$$y_p = y' \cdot \frac{eye}{eye - z'} \qquad x_p = x' \cdot \frac{eye}{eye - z'}$$

- In volume rendering, the diverging rays cause the perspective effect and make distant objects look smaller than nearer objects
- So the entire *world-to-screen* transform is:
  $M_{trans} = M_{proj} \cdot M_{view} =$
  $M_{proj} \cdot R_{view} \cdot T_{view}$
  with a subsequent divide by the homogenous coordinate
- M_{trans} is composed only once per view and all object points (vertices) are multiplied by it

Parallel (Orthographic) Projection

- Sometimes we don’t want to use perspective projection because we want to preserve perceived relative object size (e.g., for comparison)
- In this case we can use *orthographic* or *parallel projection*
- X-ray machine analog
- No separate projection matrix, as with perspective projection. Just use the modelview matrix (i.e., $x_p = x'$, $y_p = y'$)
- But, like perspective, we need to apply the *window transformation*, which we’ll look at next

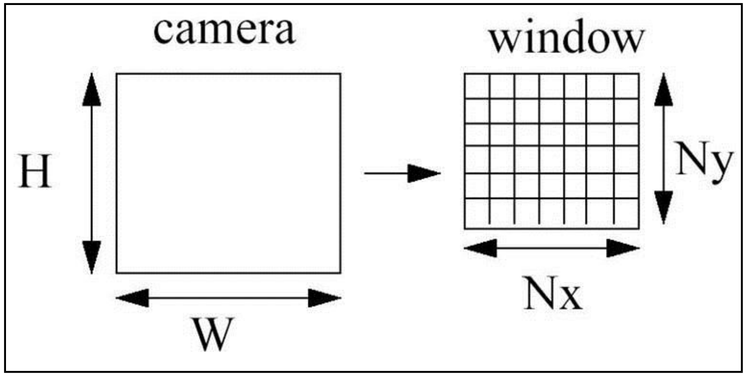




Window Transform

- We have our desired pixel location (x_p, y_p), but it's described in world coordinates, not screen coordinates
- Our display monitor is described on a pixel raster of size (Nx, Ny)
- Assume:
  - we want to display the rendered screen image in a window of size (Nx, Ny) pixels
  - the width and height of the camera screen in world coordinates are (W, H)
  - the center of the camera is at the center of the screen coordinate system
- Then:
  - the valid range of object coordinates is (-W/2 ... +W/2, -H/2 ... +H/2)
  - these have to be mapped into (0 ... Nx-1, 0 ... Ny-1) using the following transformation:

$$x_s = \left(x_p + \frac{W}{2}\right) \cdot \frac{Nx - 1}{W} \qquad y_s = \left(y_p + \frac{H}{2}\right) \cdot \frac{Ny - 1}{H}$$



Window Transform

- The window transform can be written as the matrix M_{window}:

$$\begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{Nx-1}{W} & 0 & \frac{Nx-1}{2} \\ 0 & \frac{Ny-1}{H} & \frac{Ny-1}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix}$$

- After the perspective divide, all object points (vertices) are multiplied by M_{window}
- Note: we could incorporate the window transform into M_{trans}
- In that case, there is only one matrix multiply per object point (vertex) with a subsequent perspective divide
- The OpenGL graphics pipeline does this

Rotating Object or Camera?

- In volume rendering, most of the time we rotate the camera rather than the object
- Why?
- Much too expensive to apply the transformations to every voxel
- However, it’s more intuitive usually to think of the camera being fixed and the object being rotated (and/or translated)
- We can still formulate the rotation as though we would apply it to the object, but instead we will apply the inverse of the rotation matrix to the camera
- The rotation matrix is orthogonal (i.e., its inverse is its transpose, $RR^T=1$)
- Then you rotate the camera by that inverted matrix (this entails rotating the u, v, n vectors and the image origin)
- The effect is the same as if you had rotated the object
- Note that we are assuming the camera has already been translated some distance away from the center of the volume. If we rotate the camera and then translate it from the volume center, the camera would no longer be pointing at the volume in all likelihood!

Summary on Viewing & Projection

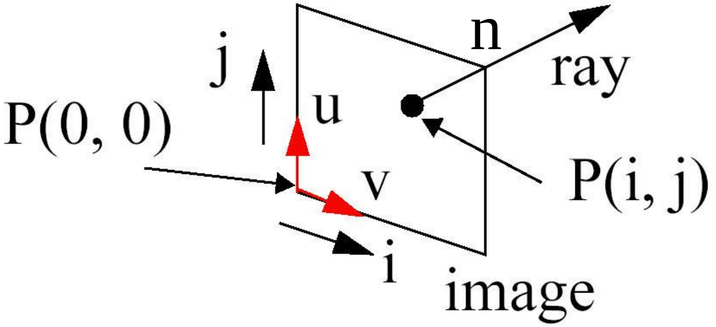
- The viewing and projection transformations are critical in computer graphics (and hence, visualization) applications
- We have really only scratched the surface. It would take several classes to cover all the details of the modeling, viewing, world-to-screen, and projection transformations
- The good news is that you will not have to do very sophisticated transformations in the assignments, but you will have to implement the ones we have covered (except for perspective projection)
- Come talk to me if you have trouble, or see Yiping
- The Foley/van Dam book or Lichtenbelt are good references, but pretty much any other introductory computer graphics book will have the matrices and their derivations

Transformations Recap

- We have looked at spatial transformations
- Modeling transformations, viewing transformations
- Projection transformations – perspective and parallel
- We defined the viewing transformations as happening to the camera, and modeling transformations as happening to objects
- This is a natural way of thinking of how to view an object
- In volumetric ray-casting practice, however, this is not precisely how things are done
- Too computationally expensive to rotate volumes
- Hence, transformations happen to the camera, or image plane
- We pretend we are rotating the volume and assemble the appropriate *modeling* transformation matrix
- BUT, then we instead apply the inverse of this matrix (transpose if rotation only) to the image plane, thereby moving the camera instead of the object
- So, you can think of it as converting a modeling transformation into a viewing transformation

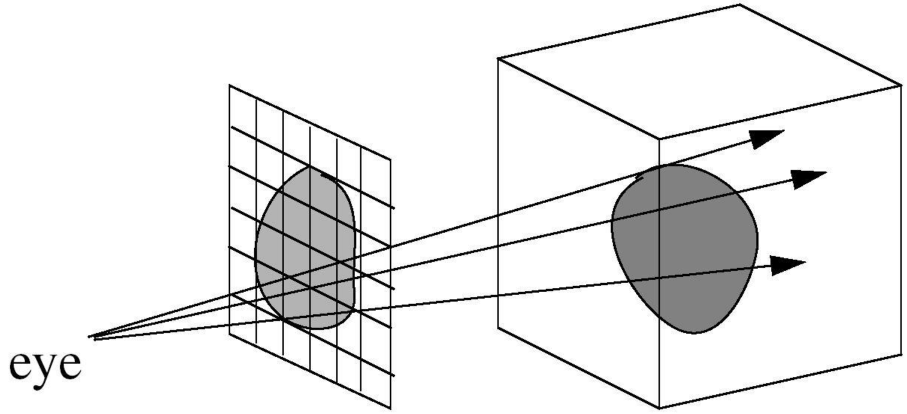
Transformation Recap

- Implementing these transformations for ray-casting is actually not so hard
- Assume the volume is fixed in space and the camera is initially on the z-axis looking in the negative z direction (and we use parallel projection)
- The viewing rays point in the same direction, but start at different (x,y)
- Viewing rays begin at the pixels and shoot into the scene towards the volume
- Pixels and rays described in same coordinate system as the volume
- To change the *perceived* position and orientation of the voxels, translate and rotate the camera to achieve the same effect as though you had transformed the voxels – this is the opposite of what we usually do in traditional graphics
- This means that the (parallel) rays can be pointed in a direction different from the default (along negative z) so that we can see the volume from different views
- To rotate the view, we just rotate the u, v and n vectors around the center of the volume

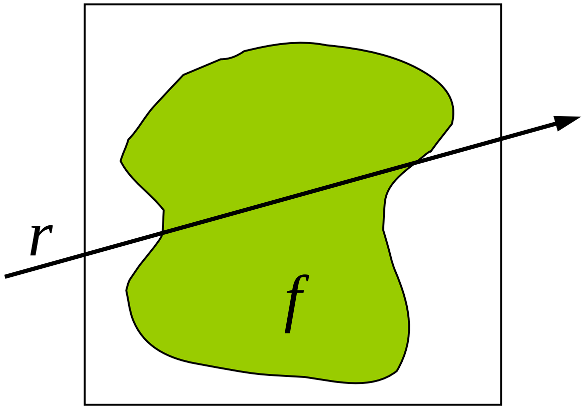


Volume Rendering – Continuous Case

- Let's return to volume rendering and see how transformations are used in *ray-casting*
- We want to shoot a ray through the volume and *accumulate* or *composite* information about the volume as we travel along it



- In the continuous domain, this ray traversal is best expressed as a definite integral.



X-ray equation:

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

$I_{i,j}$ is the intensity of pixel *P* located at position *i,j* on the screen. $r_{i,j}$ is the ray's direction, *L* its length, and *t* is the distance along *r*.
f(x,y,z) gives the value of the data-set.

Volume Rendering – Discrete Case (X-ray)

- In practice, we very rarely use “continuous” data
- Most of the time our data is discrete, i.e., sampled via CT, MRI or some similar scanning modality
- We can discretize the ray-casting integral and arrive at the notion of *discrete ray-tracing*

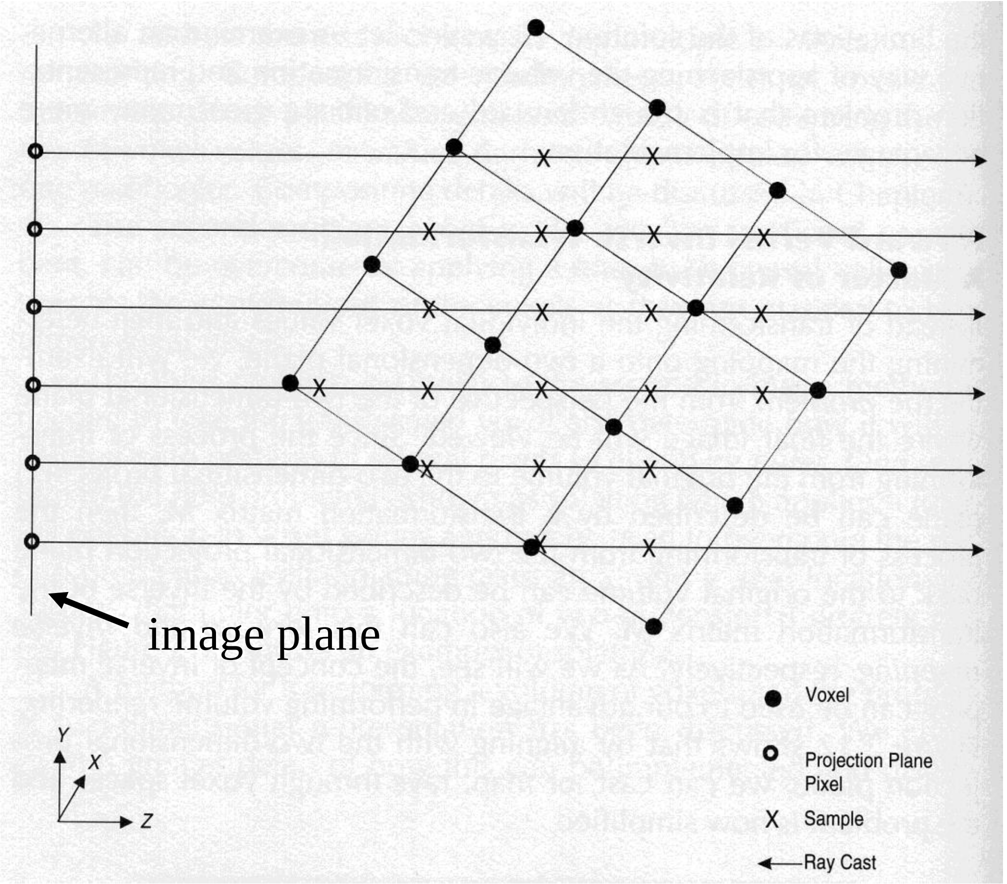


image plane

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

Remember: these equations are only for X-ray rendering!

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

Volume Rendering – Discrete Case (X-ray)

- X-ray rendering adds the densities as we step along the ray
- Some important things to note:
- Usually we take uniform steps along the ray (Δt = 1)
- Each sample we take from the volume (indicated by X) is given equal weight
- After ray-tracing the volume for each pixel, the intensity stored at each pixel may be greater than 255 (indicating pure white)
- Or, the maximum pixel intensity may be less than 255
- If so, the entire image should be normalized by dividing each pixel intensity by the maximum that appears in the image and multiplying by 255

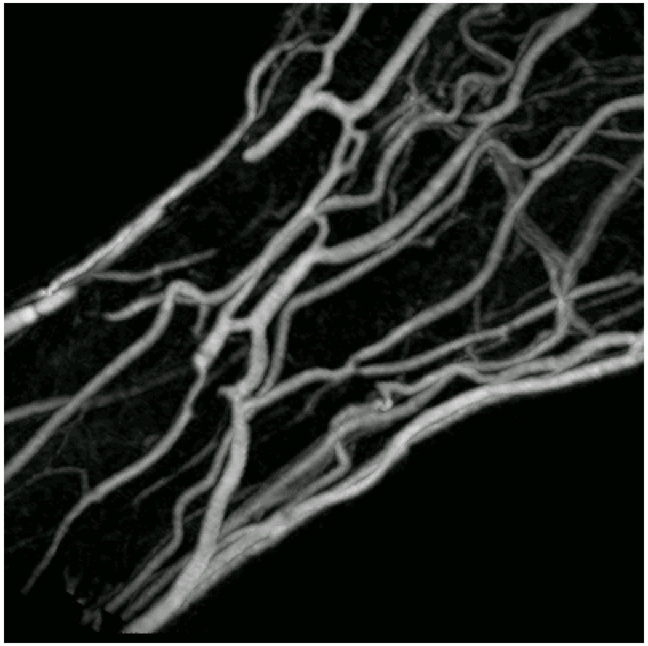
Volume Rendering – Discrete Case (MIP)

- Another important mode of volumetric ray-tracing is called Maximum Intensity Projection or MIP (note, this is not the same as mip-mapping)
- In this case we find the maximum value of *f* along the ray and store that value in the pixel

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

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

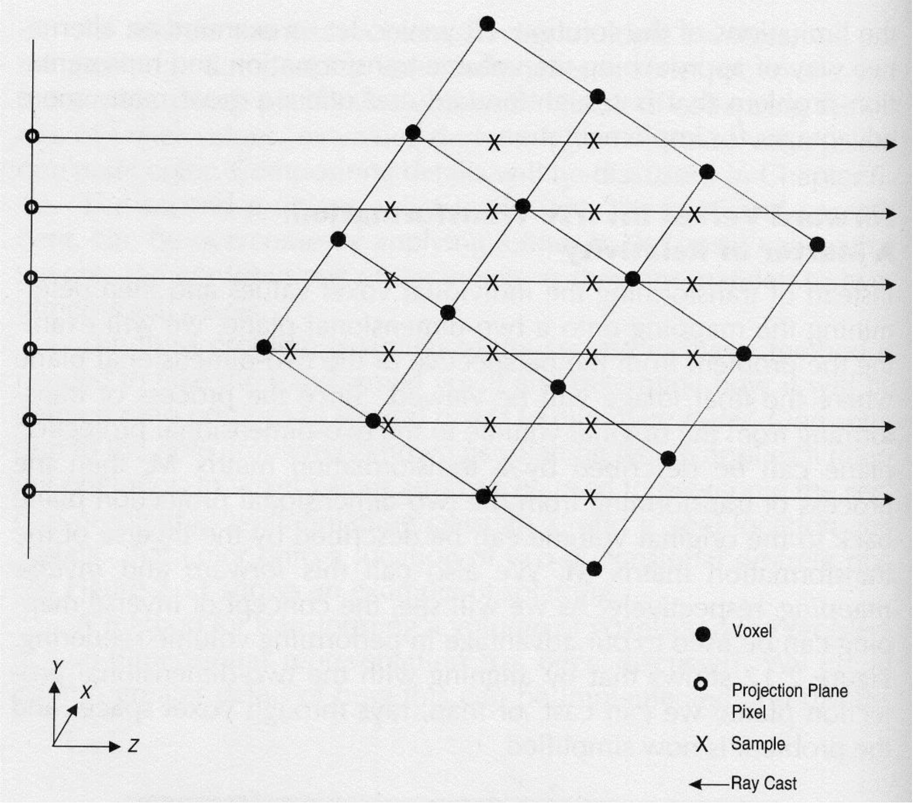
- MIP is great for visualizing bright structures in a data-set
- Bright, distant structures are *not* occluded by nearer, dimmer objects
- Let’s watch a movie! (2)



Courtesy Technische Universität Wien

Interpolation

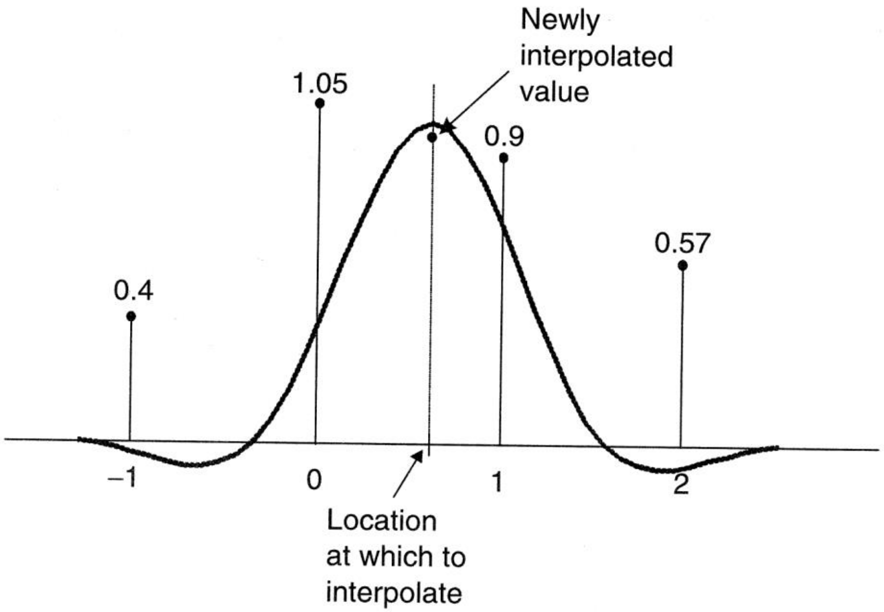
- Usually, the samples we take along the ray do not exactly coincide with the voxels on the grid
- They fall someplace between the grid points, whose values we know at the outset
- We need to estimate or approximate somehow what these in-between values probably were in the original, continuous object that was scanned
- This estimation process is called *interpolation*, which we saw earlier when we studied the human perceptive system
- We'll see how interpolation can be expressed mathematically in 1D, then generalize it to 2D and finally, 3D



- Finally, we can incorporate interpolation into the volume rendering pipeline to generate both X-ray and MIP visualizations

Interpolation – 1D

- The goal of interpolation is to approximate the true value of a function at some location by using nearby information to infer or approximate what the value should be or was in the original
- An *interpolation function* or *interpolation kernel* assigns weights to the neighbors to determine how much each neighbor should contribute to the interpolated value
- The function is centered over the position we want to interpolate
- For function on right: f [-1] = 0.4, f [0]=1.05, f [1] = 0.9, f [2] = 0.57 (*samples* of the function *f*)
- Interpolation kernel values are -0.02, 0.38, 0.66, -0.07



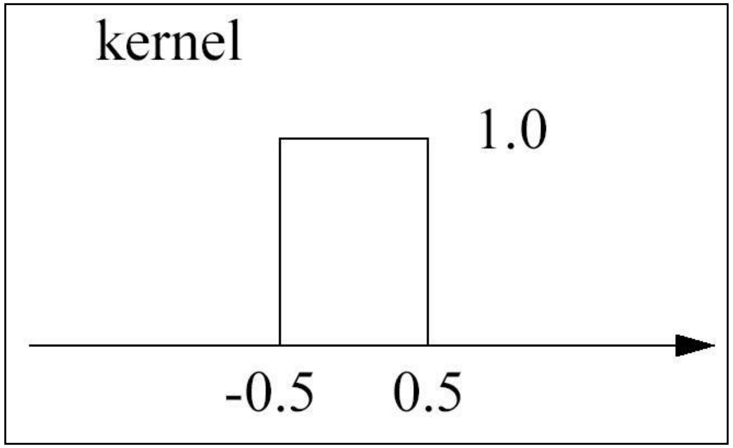
- The interpolated value is therefore: (0.4 * -0.02) + (1.05 * 0.38) + (0.9 * 0.66) + (0.57 * -0.07) = 0.945

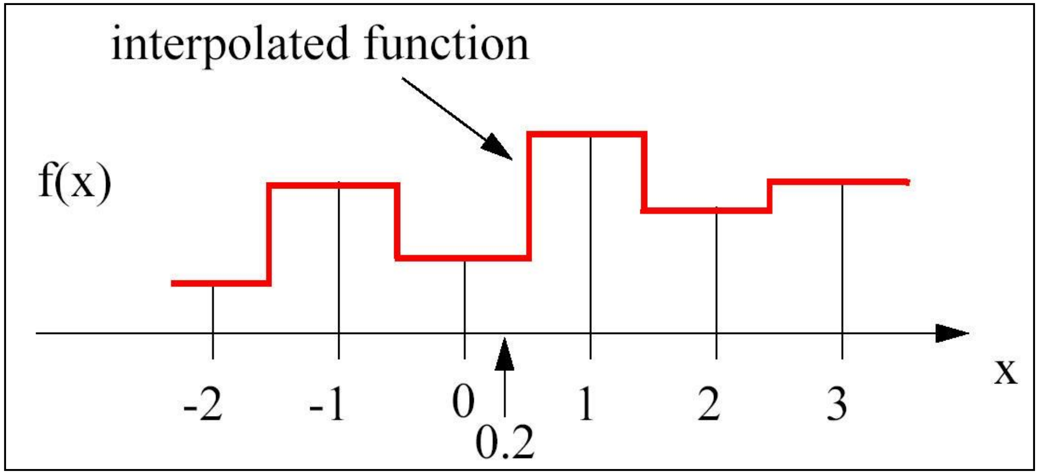
- Typically, the interpolation weights sum to 1.0 (but not always)

STONY BROOK
STATE UNIVERSITY OF NEW YORK

Interpolation Functions

- There are many choices for interpolation functions
- The simplest (and crudest) method is called *nearest neighbor interpolation*
- Basically, all you do is look for the nearest point to your selected position and take that value
- In the following example, we have applied nearest neighbor interpolation across the entire domain in an attempt to recover the original function, which we sampled

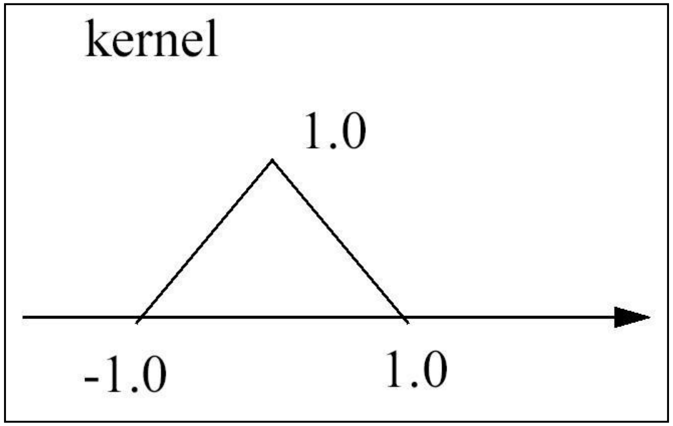


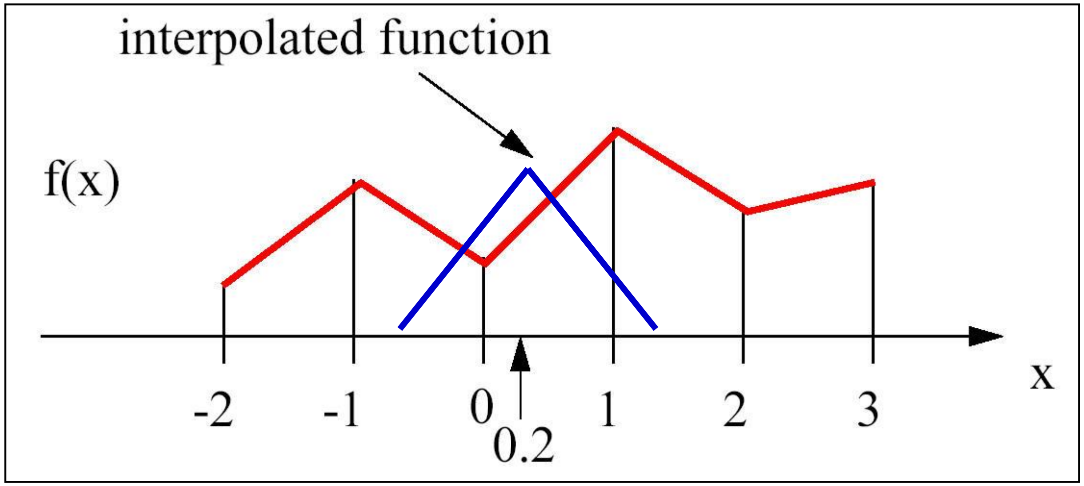


- f(0.2) = f(trunc(0.2+0.5)) = f(round(0.2)) = f(0)

Interpolation Functions

- A linear filter is another simple interpolation function



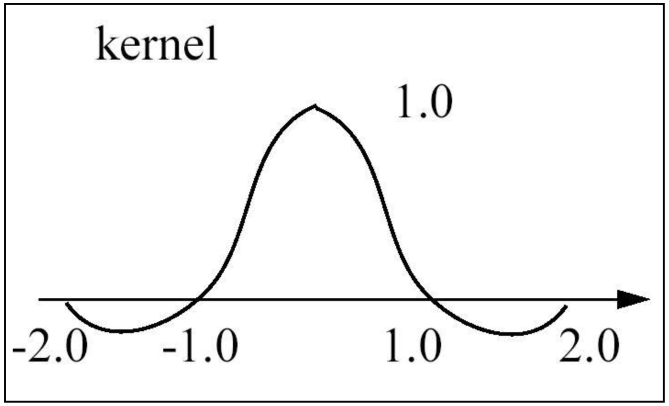


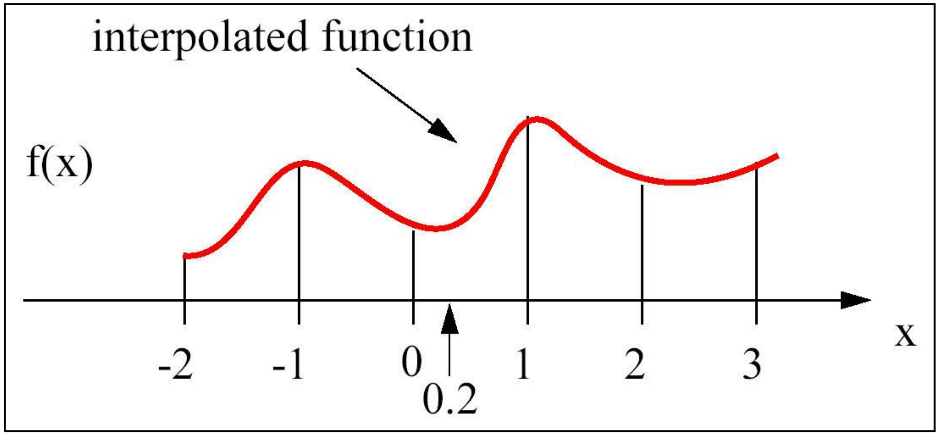
- We take a linear combination of the two neighboring grid values
- $f(\underline{0.2}) = \underline{0.2} * f(1) + \underline{0.8} * f(0)$

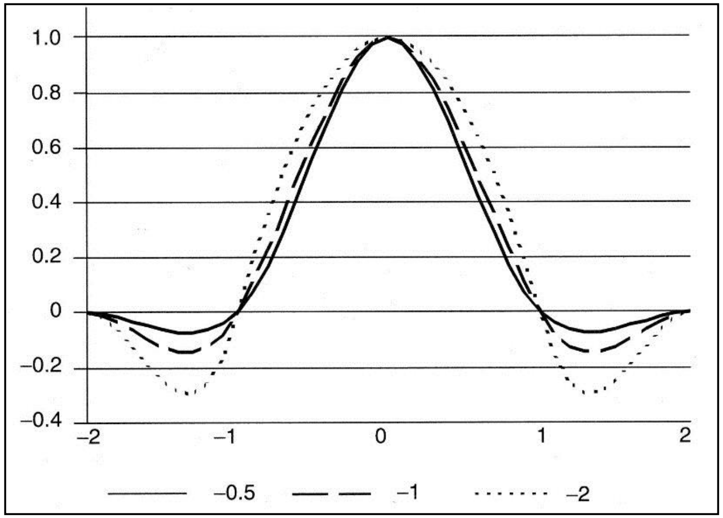
- The fact that these three values are related is no coincidence, and we’ll see that this makes linear interpolation very attractive

Interpolation Functions

- Cubic filters can give very good results, but are computationally expensive







$$f(x) = \begin{cases} (a+2)|x|^3 - (a+3)|x|^2 + 1 & 0 \leqslant |x| \leqslant 1 \\ a|x|^3 - 5a|x|^2 + 8a|x| - 4a & 1 \leqslant |x| \leqslant 2 \\ 0 & 2 \leqslant |x| \end{cases}$$

Back to Linear Interpolation

- Linear interpolation is by far the most popular interpolation method
- It's very fast and usually gives adequate results

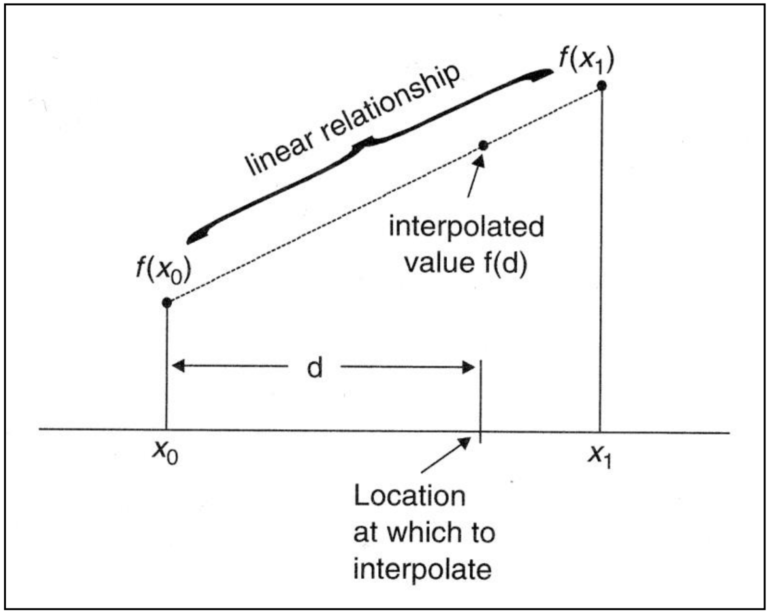
$$f(d) = \frac{f(x_1) - f(x_0)}{x_1 - x_0} \cdot d + f(x_0)$$

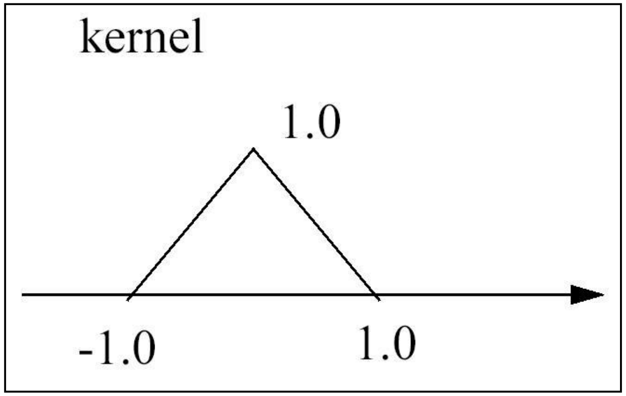
- If we assume unit spacing (i.e., $(x_1 - x_0) = 1$), then we can simplify the formula

$$f(d) = (f(x_1) - f(x_0)) \cdot d + f(x_0)$$

$$f(d) = (1 - d) \cdot f(x_0) + d \cdot f(x_1)$$

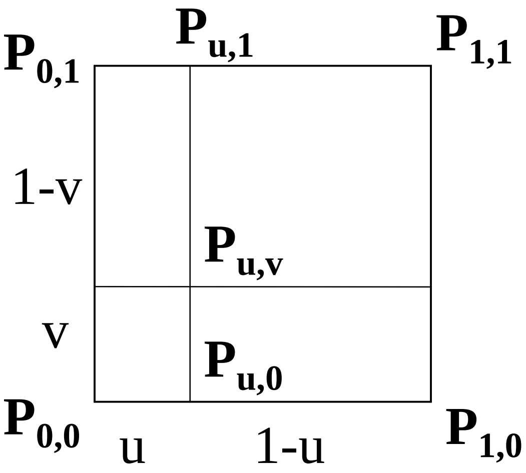
- The latter equation is how it's usually written, but the first form is potentially more computationally efficient (depends on what *f* is)





Bilinear Interpolation

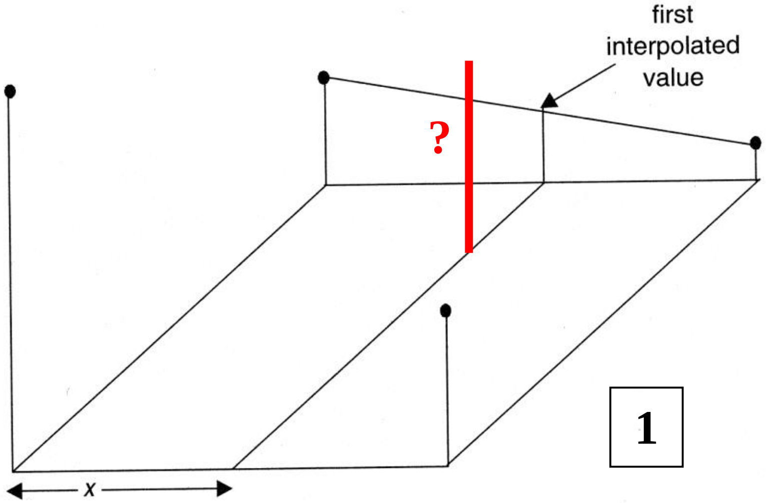
- Linear interpolation generalizes to 2D very easily
- In 2D it's called *bilinear interpolation* and works like this:
- $f\,(P_{u,0}) = (1 - u) \cdot f\,(P_{0,0}) + u \cdot f\,(P_{1,0})$
- $f\,(P_{u,1}) = (1 - u) \cdot f\,(P_{0,1}) + u \cdot f\,(P_{1,1})$
- $f\,(P_{u,v}) = (1 - v) \cdot \boxed{f\,(P_{u,0})} + v \cdot \boxed{f\,(P_{u,1})}$

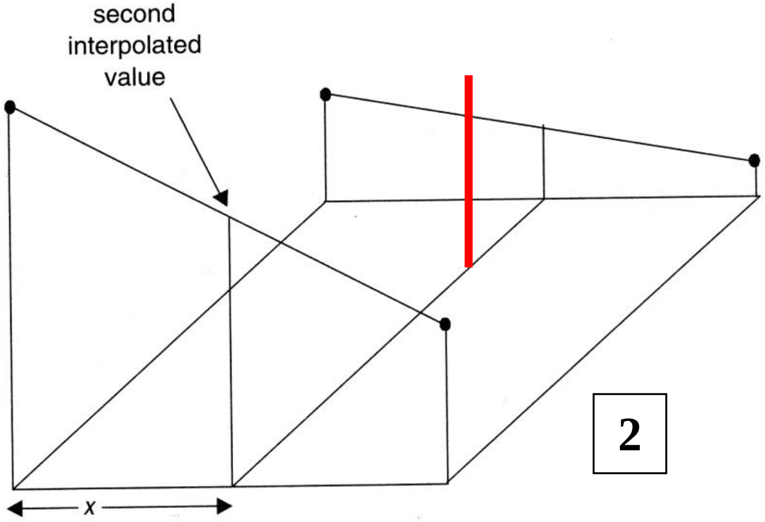


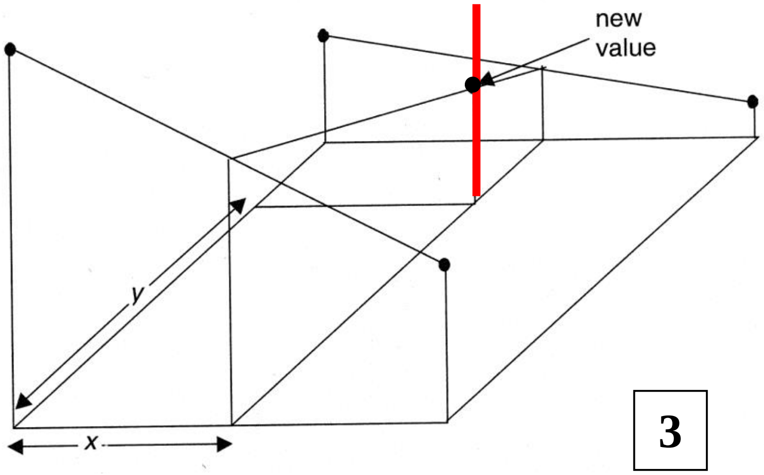
- So, a bilinear interpolation consists of 3 linear interpolations
- Alternate form:

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

A Second Look at Bilinear Interpolation







You should view bilinear interpolation as consisting of two 1D interpolations followed by a linear interpolation of the two interpolated values.

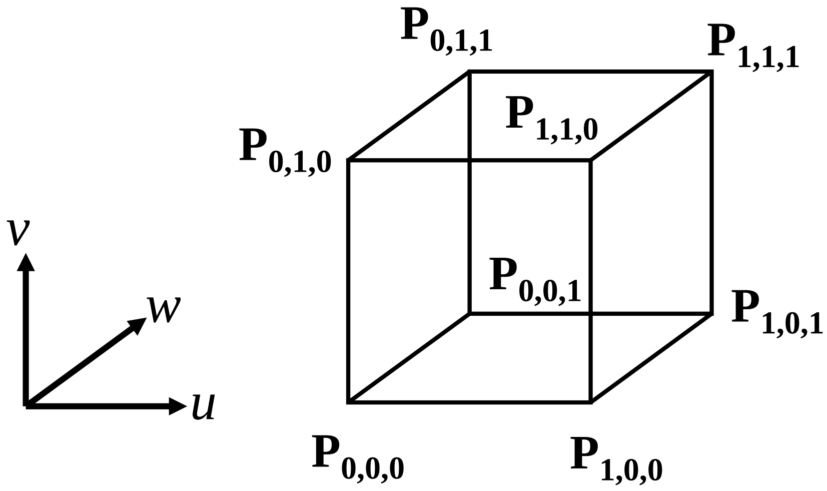
You can already see already how this can be generalized to 3D.

Trilinear Interpolation

- Linear interpolation: along a line
- Bilinear interpolation: inside a rectangle
- Trilinear interpolation: inside a…?
- Rectangular box, in general; a cube in particular
- Bilinear interpolation utilizes two 1D linear interpolations, followed by a linear interpolation of the two interpolated values
- Trilinear interpolation utilizes two (2D) bilinear interpolations, followed by a linear interpolation of the two interpolated values
- Hence, we should have 3 + 3 + 1 = 7 total 1D linear interpolations

Trilinear Interpolation

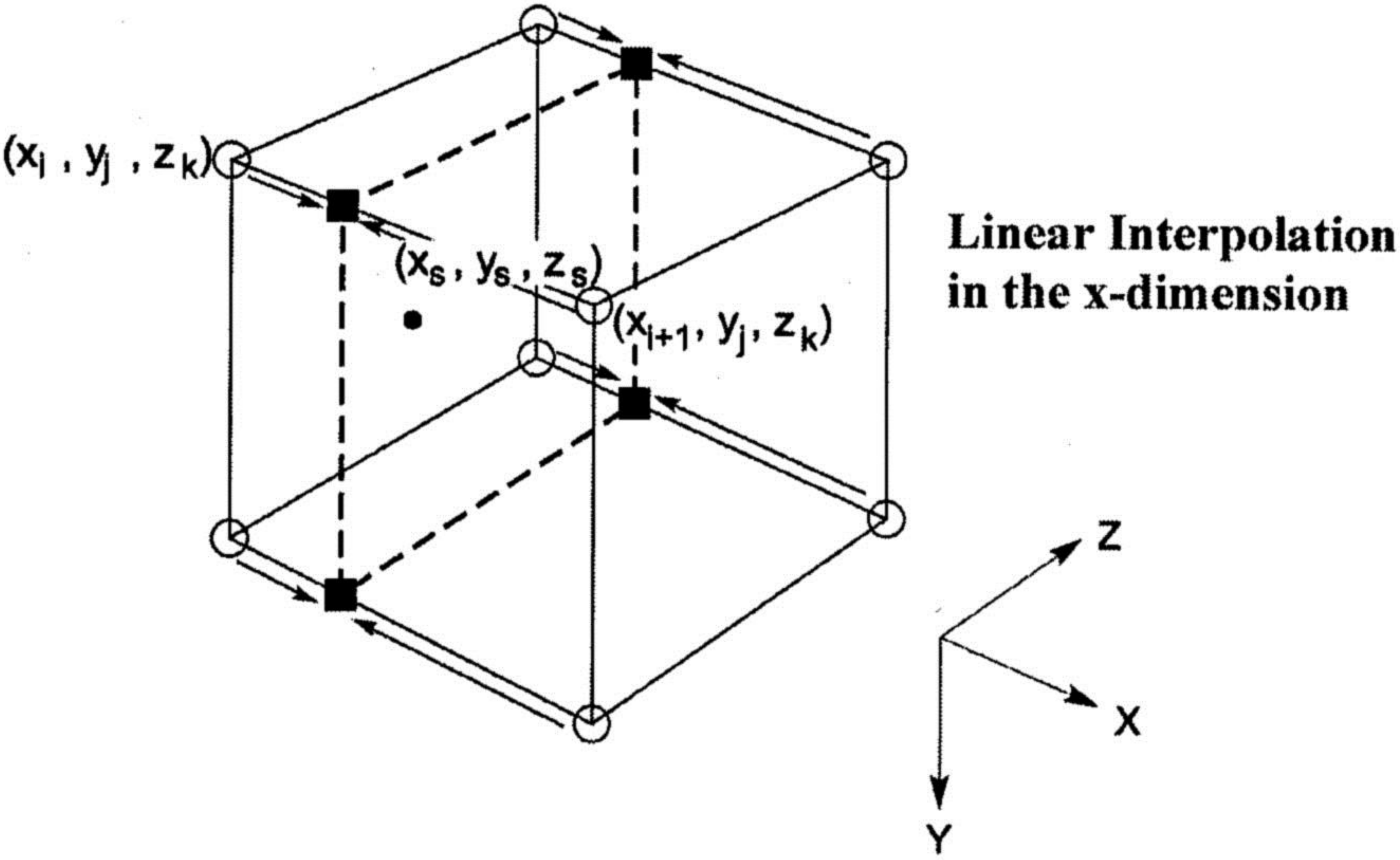
- $f(P_{u,v,0})$ = BilinearInterpolation($P_{0,0,0}$, $P_{1,0,0}$, $P_{0,1,0}$, $P_{1,1,0}$)
- $f(P_{u,v,1})$ = BilinearInterpolation($P_{0,0,1}$, $P_{1,0,1}$, $P_{0,1,1}$, $P_{1,1,1}$)
- $f(x,y,z) = f(P_{u,v,w})$ = LinearInterpolation($P_{u,v,0}$, $P_{u,v,1}$)
- Hence, a trilinear interpolation
  can be expressed as 7 linear interpolations
  (although simpler formulas exist)



- Alternate form:

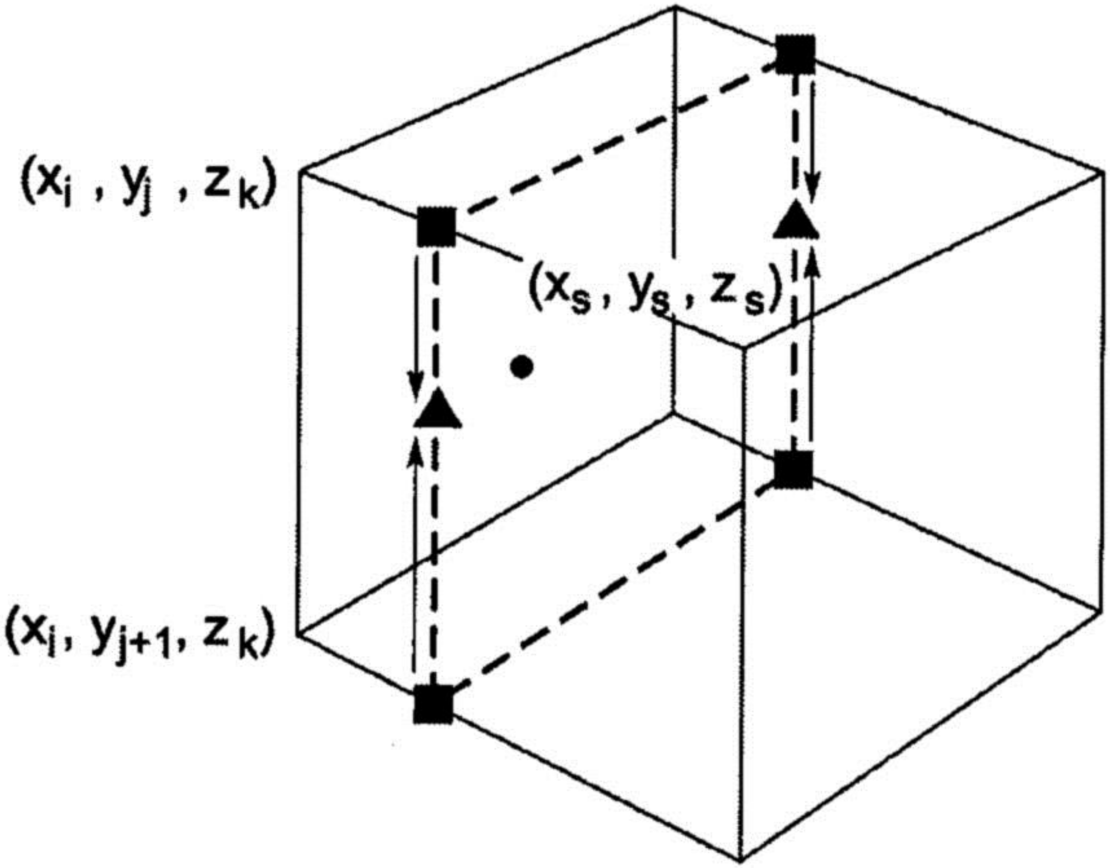
$$f(x,y,z) = \sum_{i=0}^{1}\sum_{j=0}^{1}\sum_{k=0}^{1} u^i(1-u)^{1-i} v^j (1-v)^{1-j} w^k (1-w)^{1-k} P_{i,j,k}$$

Trilinear Interpolation – Alternate Interpretation



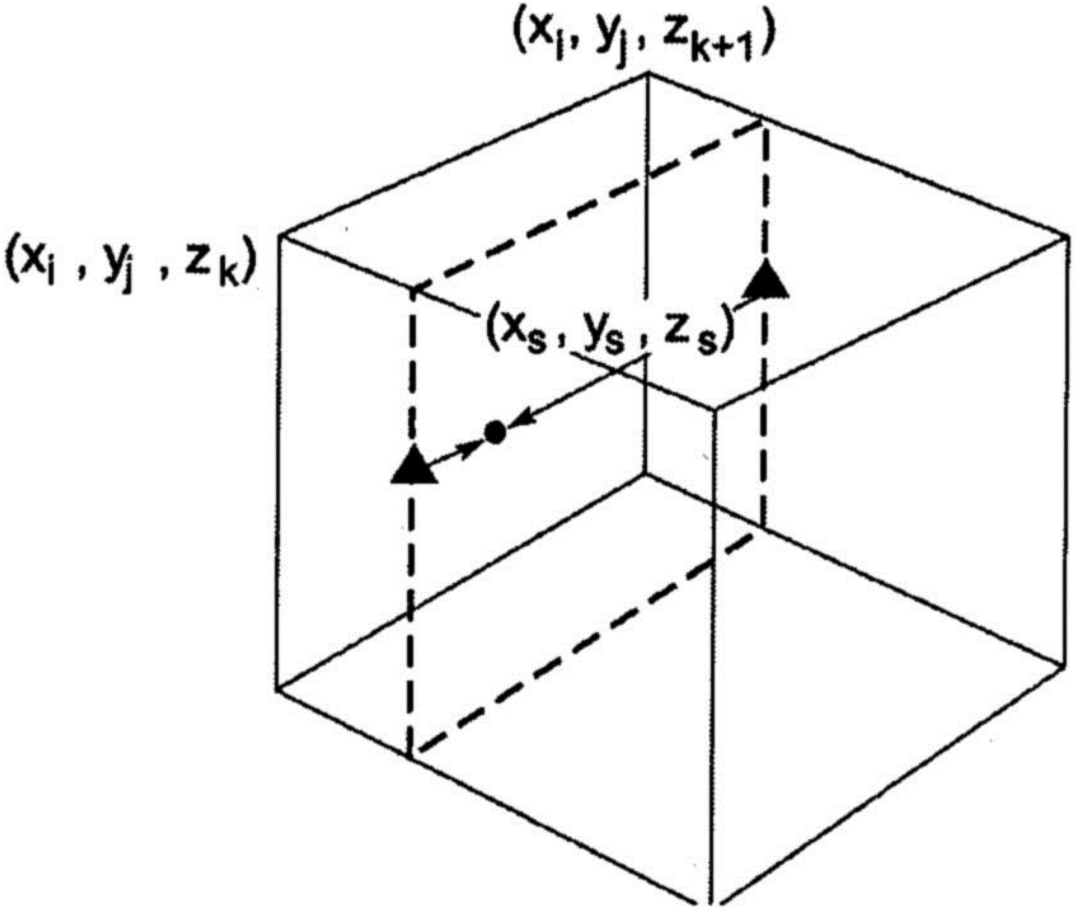
Linear Interpolation in the x-dimension

Trilinear Interpolation



Linear Interpolation in the y-dimension

Trilinear Interpolation



Linear Interpolation in the z-dimension

Trilinear Interpolation – Implementation Issues

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

```
        u = P[0] - (int)P[0]; v = P[1] - (int)P[1]; w = P[2] - (int)P[2];
```

- Volume stored in 1D array as x-y-z (x index changes faster than y index, y index changes faster than z index)
- Now do the decomposition into 7 linear interpolations:

```
        val1 = u * volData[(int)P[2] * nx*ny + (int)P[1] * nx + (int)P[0] +1 ] +
                (1-u) * volData[(int)P[2] * nx*ny + (int)P[1] * nx + (int)P[0] ]);
        val2 = u * volData[(int)P[2] * nx*ny + ((int)P[1] + 1) * nx + ((int)P[0]) +1 ] +
                (1-u) * volData[(int)P[2] * nx*ny + ((int)P[1] + 1) * nx + (int)P[0] ]);
        val3 = (1-v) * val1 + v * val2; ...
        val4, val5, val6, val7
```

(nx, ny = volume sizes in x and y, i.e., width and height)

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

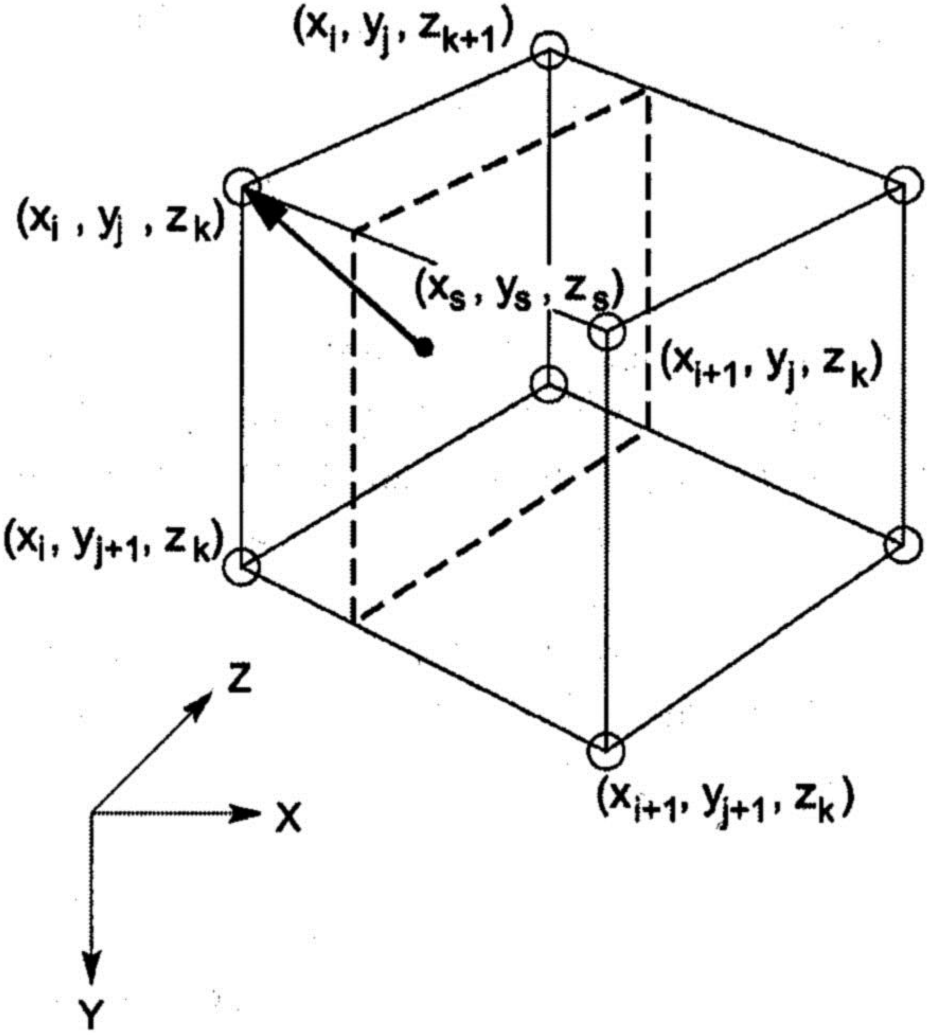
```
        cptr=&(volume->data[(int)P[2] * nx*ny + (int)P[1] * nx + (int)P[0]]);
        P000=*cptr; P100=*(cptr+1); P001=*(cptr+nx*ny); P101=*(cptr+nx*ny+1);
        P010=*(cptr+nx); P110=*(cptr+nx+1); P011=*(cptr+nx*ny+nx);
        P111=*(cptr+nx*ny+nx+1);
        intVal= (1-w) * ( (1-v) * (u * P100 + (1-u) * P000) + v * (u * P110 + (1-u) * P010))
                +w * ( (1-v) * (u * P101 + (1-u) * P001) + v * (u * P111 + (1-u) * P011));
```

Nearest-Neighbor Interpolation in 3D

- What one-line piece of code would implement nearest-neighbor interpolation?

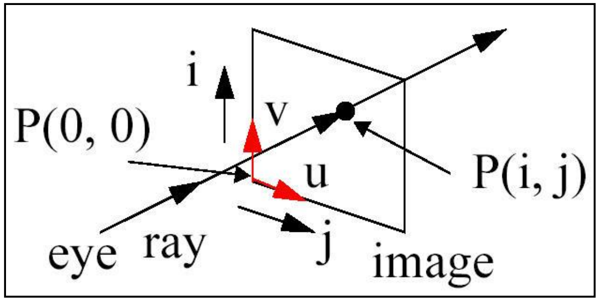
- Nearest neighbor interpolation in 3D:
  $f\,(x,y,z) = f\,(P_{round(x),\, round(y),\, round(z)})$

  where $(round(X) = trunc(X+0.5))$



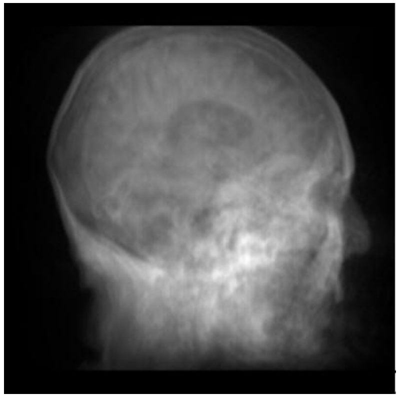
Putting It All Together: X-Ray Rendering Algorithm

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



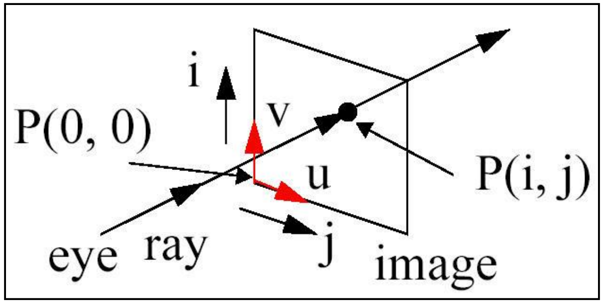
RenderXRay(Volume V, int stepSize, Image I)

```
if (projectionMode == Orthographic)
    ray = v × u / | v × u| // view direction vector is perpendicular to image plane
for each image pixel (i, j)
    P(i, j) = P(0, 0) + (i · v · Δi) + (j · u · Δj);
    sum = 0;
    if (projectionMode == Perspective)
        ray = (P(i, j) - eye) / | (P(i, j) - 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 · 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);
```



Putting It All Together: MIP Rendering Algorithm

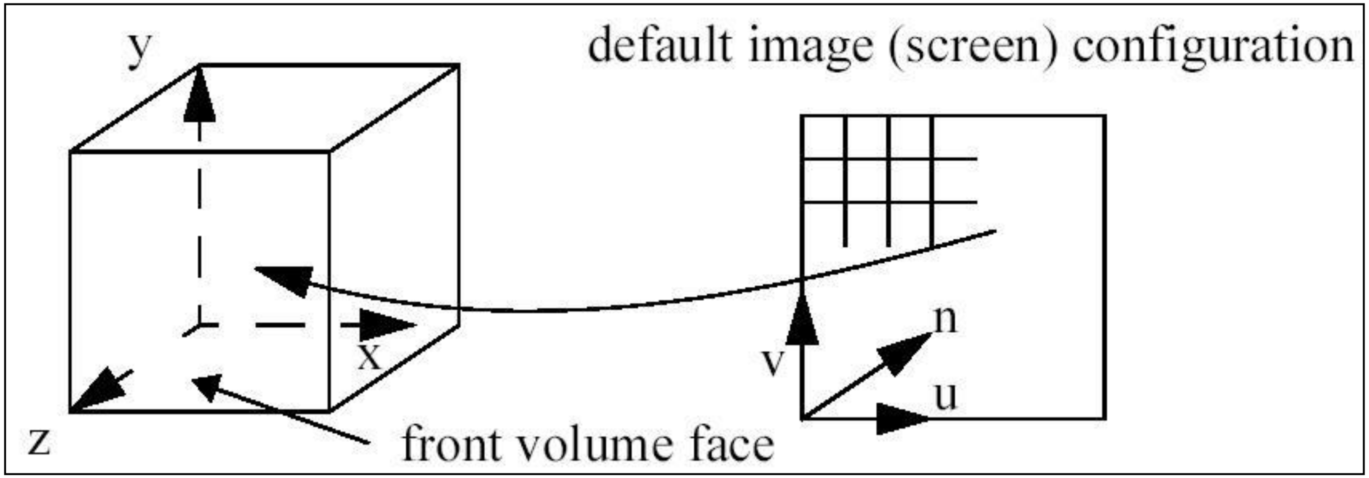
$$\boxed{I_{i,j} = \underset{0 \le k \le L/\Delta t}{Max} \left(f\left(P_{i,j} + k \cdot \Delta t \cdot r_{i,j}\right)\right)}$$



```
RenderMIP(Volume V, int stepSize, Image I)
if (projectionMode == Orthographic)
    ray = v × u / | v × u| // view direction vector is perpendicular to image plane
for each image pixel (i, j)
    P(i, j) = P(0, 0) + i · v · Δi + j · u · Δ j;
    if (projectionMode == Perspective)
        ray = (P(i, j) - eye) / | (P(i, j) - 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 · ray // step along the ray
        intVal = Interpolate(V, sampleLoc);
        if(intVal > max)
            max = intVal;
I(i, j) = max;
```

Image Plane and Projection

- *u* and *v* vectors
- Initially, set *u* = (1,0,0), *v* = (0,1,0)
- Viewing ray *n* = (0,0,-1)
- Center the volume over (0,0,0)
- Put the center of the image plane at (0,0,*k*) where *k* is at least ½ the depth of the volume



Cross-Product Confusion

- The ray direction, *n*, is, in general, given by the normalized cross-product of *v* and *u*:

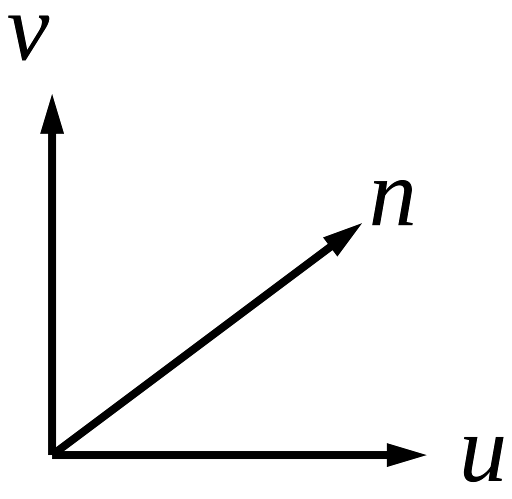
$$n = \frac{v \times u}{\left| v \times u \right|}$$

$$u = (u_x, u_y, u_z)$$

$$v = (v_x, v_y, v_z)$$

$$n = (n_x, n_y, n_z)$$

$$v \times u = (v_y u_z - v_z u_y,\ v_z u_x - v_x u_z,\ v_x u_y - v_y u_x)$$



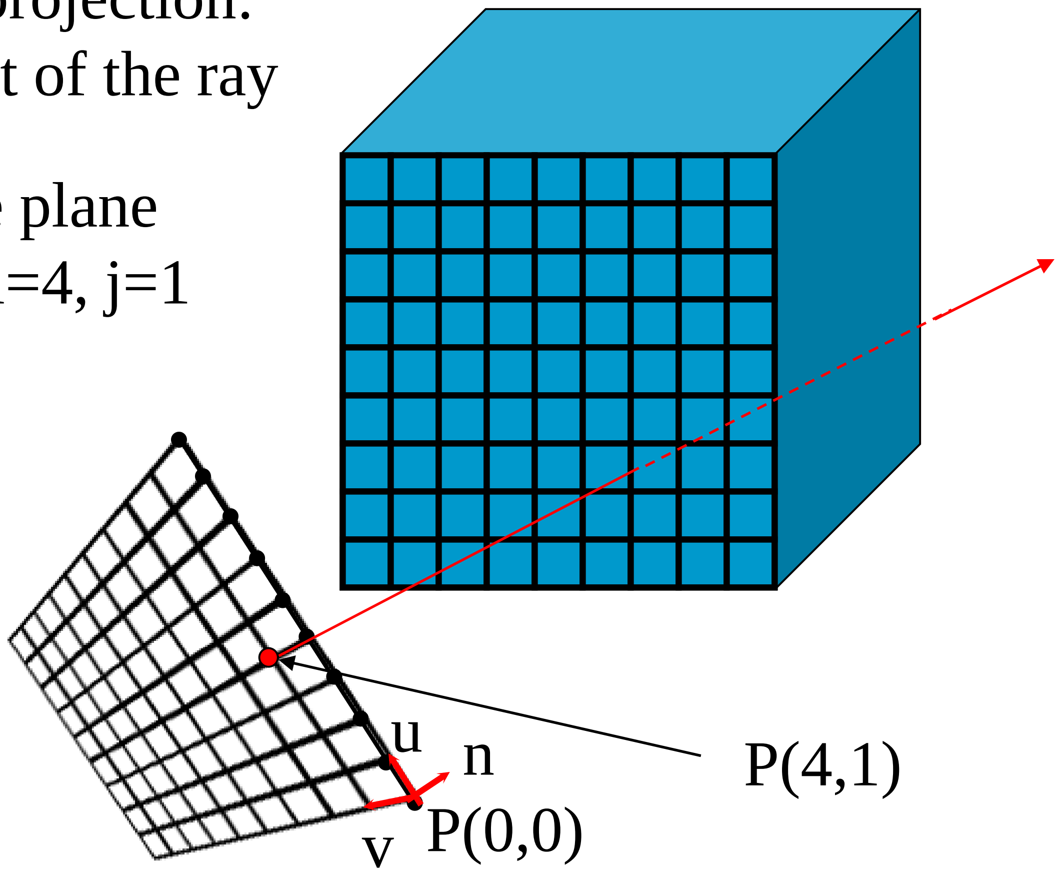
Ray-Casting & Interpolation Demo

- Let’s see an example of ray-casting and interpolation for a particular ray through a volumetric raster
- We’ll use parallel projection
- First we’ll use nearest-neighbor interpolation and then trilinear interpolation

Orthographic (parallel) projection: Finding the starting point of the ray

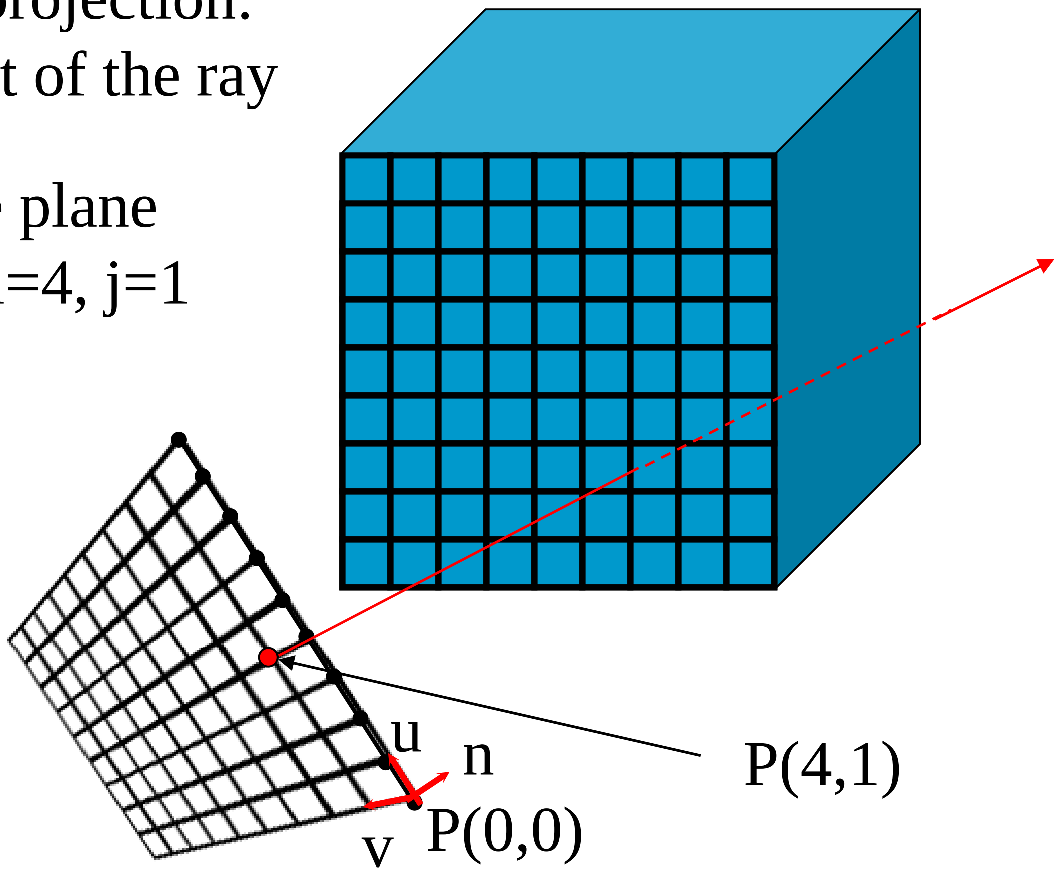
P(0,0): origin of image plane

P(4,1): image pixel at i=4, j=1



Orthographic (parallel) projection: Finding the starting point of the ray

P(0,0): origin of image plane
P(4,1): image pixel at i=4, j=1

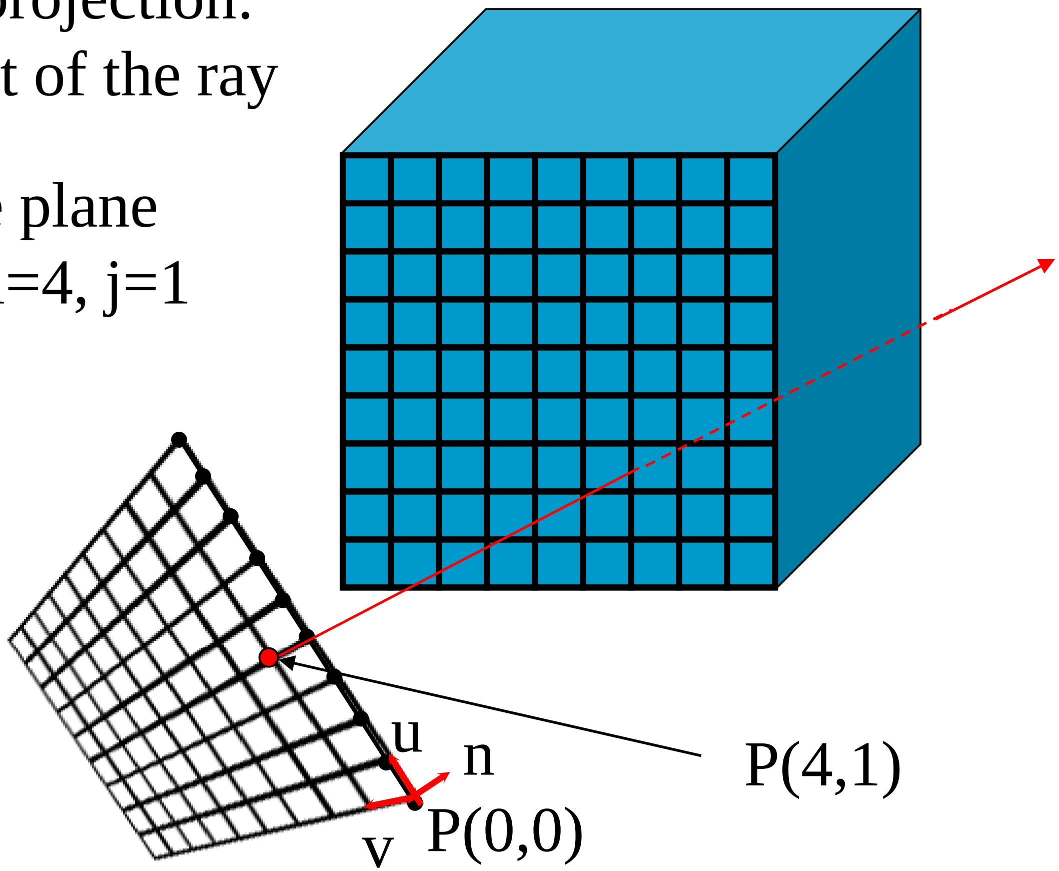


$$P(4,1) = P(0,0) + 4 \cdot u + 1 \cdot v$$

Orthographic (parallel) projection: Finding the starting point of the ray

P(0,0): origin of image plane

P(4,1): image pixel at i=4, j=1



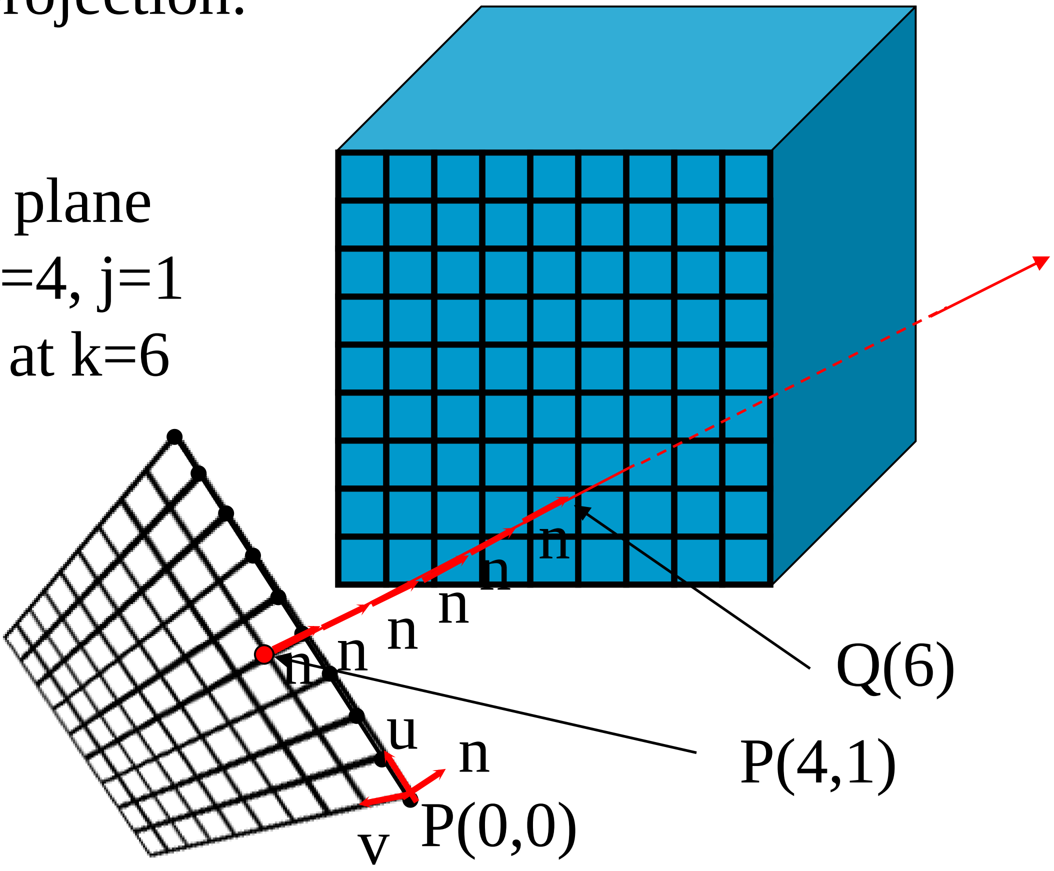
$$P(4,1)=P(0,0)+4\cdot u+1\cdot v$$

$$\begin{bmatrix} P(4,1)[0] \\ P(4,1)[1] \\ P(4,1)[2] \end{bmatrix} = \begin{bmatrix} P(0,0)[0] \\ P(0,0)[1] \\ P(0,0)[2] \end{bmatrix} + 4\cdot \begin{bmatrix} u[0] \\ u[1] \\ u[2] \end{bmatrix} + 1\cdot \begin{bmatrix} v[0] \\ v[1] \\ v[2] \end{bmatrix}$$

(places image plane into world space with the volume)

Orthographic (parallel) projection: Casting the ray

P(0,0): origin of image plane
P(4,1): image pixel at i=4, j=1
Q(6): ray sample point at k=6



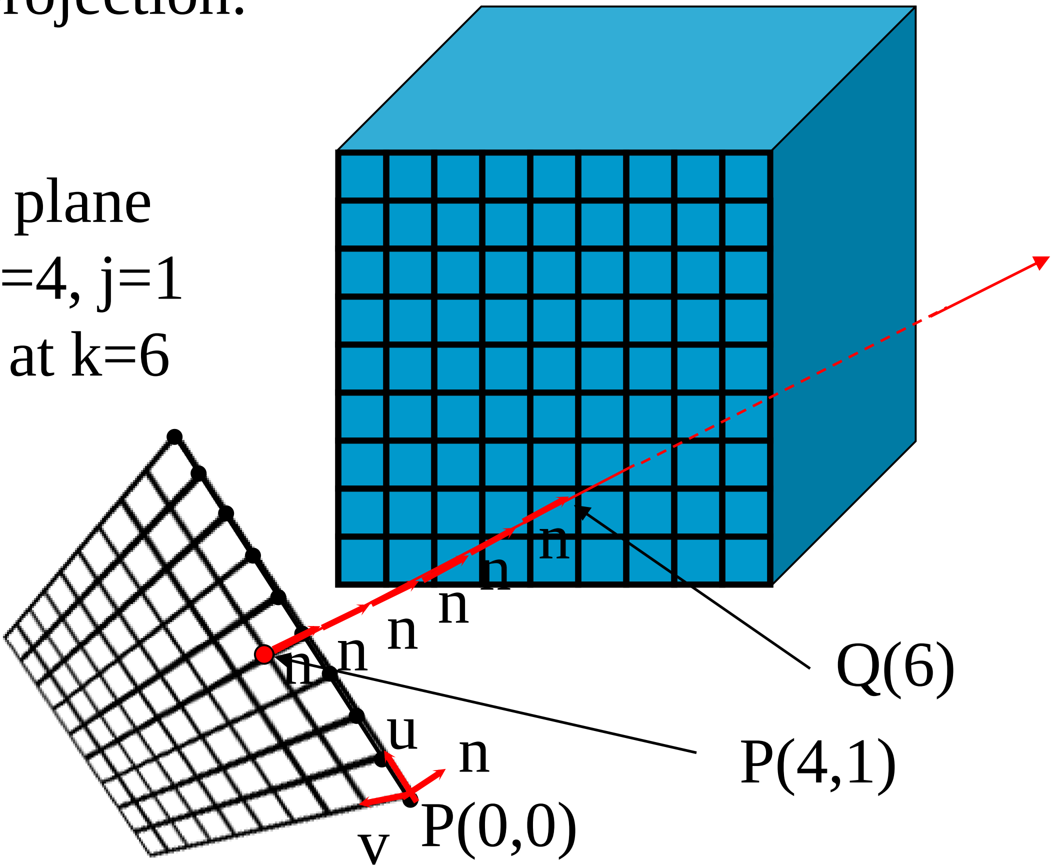
$$Q(6) = P(4,1) + 6 \cdot n$$

Orthographic (parallel) projection: Casting the ray

P(0,0): origin of image plane

P(4,1): image pixel at i=4, j=1

Q(6): ray sample point at k=6

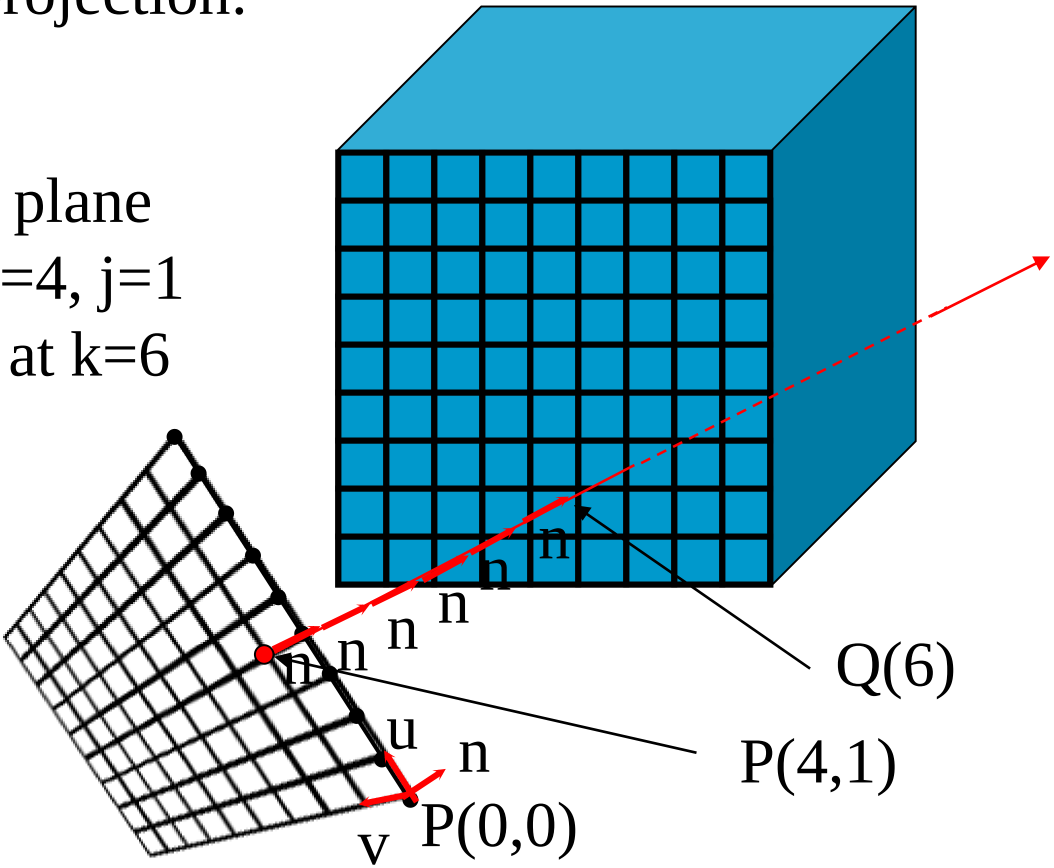


$$Q(6) = P(4,1) + 6 \cdot n$$

$$\begin{bmatrix} Q(6)[0] \\ Q(6)[1] \\ Q(6)[2] \end{bmatrix} = \begin{bmatrix} P(4,1)[0] \\ P(4,1)[1] \\ P(4,1)[2] \end{bmatrix} + 6 \cdot \begin{bmatrix} n[0] \\ n[1] \\ n[2] \end{bmatrix}$$

Orthographic (parallel) projection: Casting the ray

P(0,0): origin of image plane
P(4,1): image pixel at i=4, j=1
Q(6): ray sample point at k=6



$$Q(6) = P(4,1) + 6 \cdot n$$

$$\begin{bmatrix} Q(6)[0] \\ Q(6)[1] \\ Q(6)[2] \end{bmatrix} = \begin{bmatrix} P(4,1)[0] \\ P(4,1)[1] \\ P(4,1)[2] \end{bmatrix} + 6 \cdot \begin{bmatrix} n[0] \\ n[1] \\ n[2] \end{bmatrix} = \begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$$

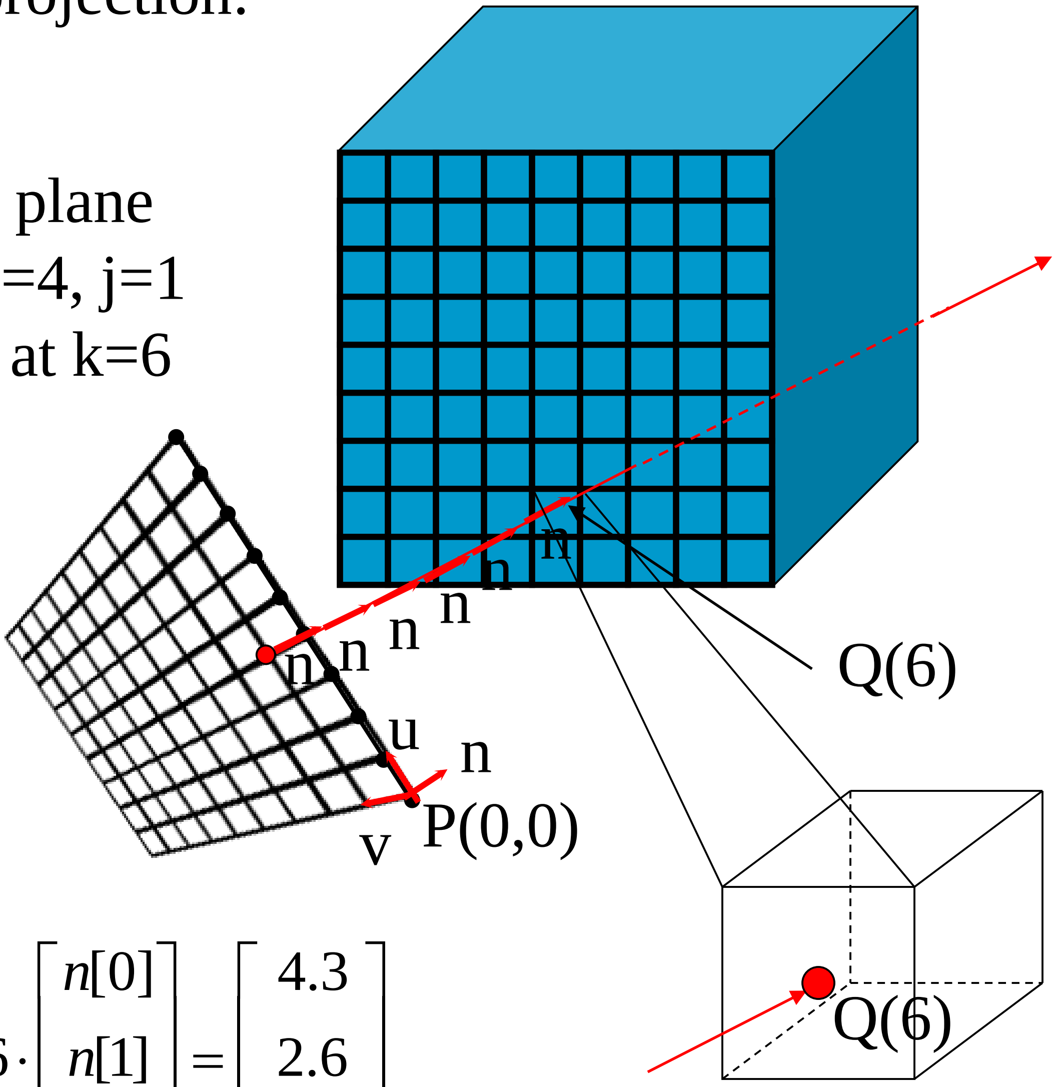
(assume)

Orthographic (parallel) projection: Interpolation

P(0,0): origin of image plane

P(4,1): image pixel at i=4, j=1

Q(6): ray sample point at k=6

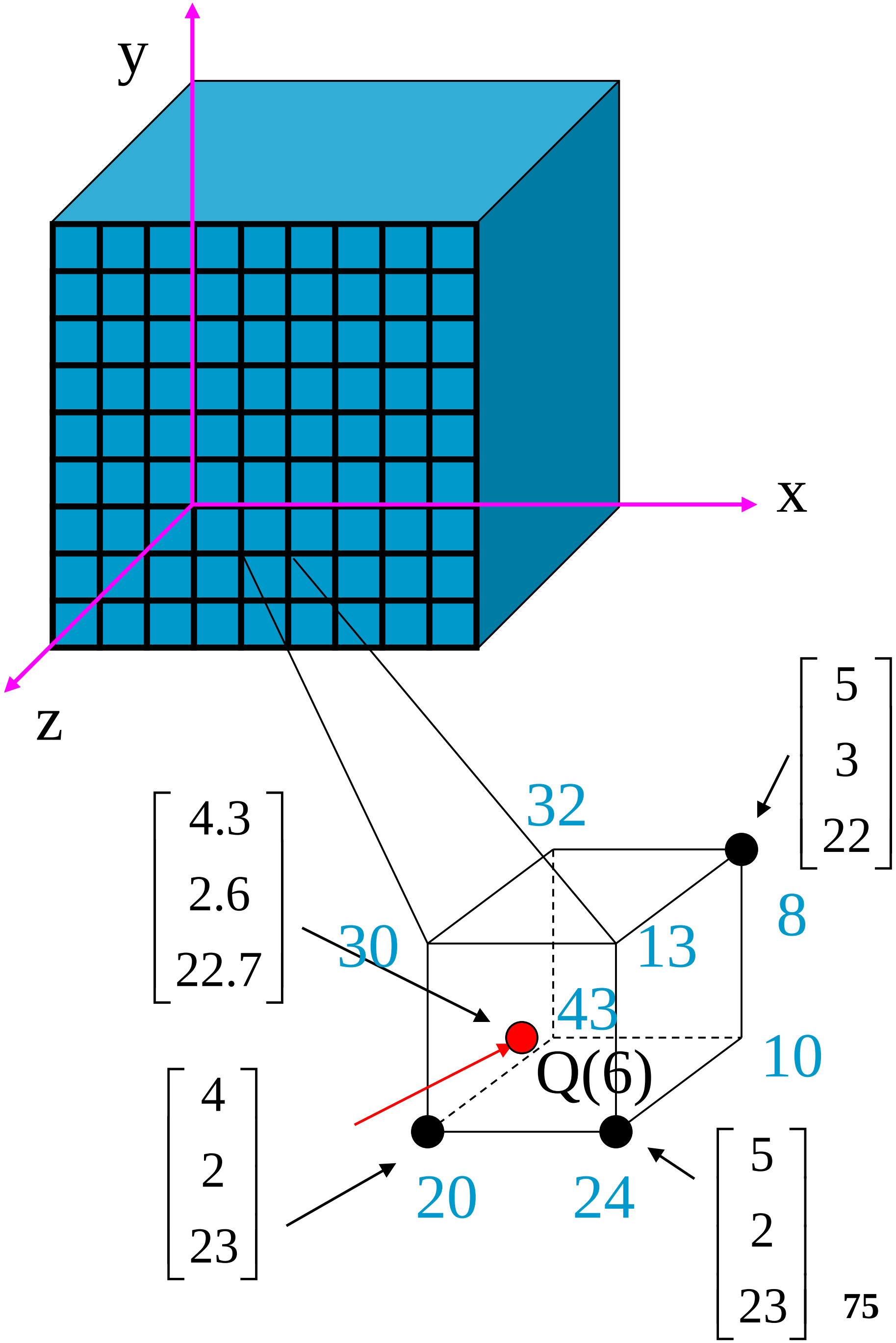


$$Q(6) = P(4,1) + 6 \cdot n$$

$$\begin{bmatrix} Q(6)[0] \\ Q(6)[1] \\ Q(6)[2] \end{bmatrix} = \begin{bmatrix} P(4,1)[0] \\ P(4,1)[1] \\ P(4,1)[2] \end{bmatrix} + 6 \cdot \begin{bmatrix} n[0] \\ n[1] \\ n[2] \end{bmatrix} = \begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$$

(assume)

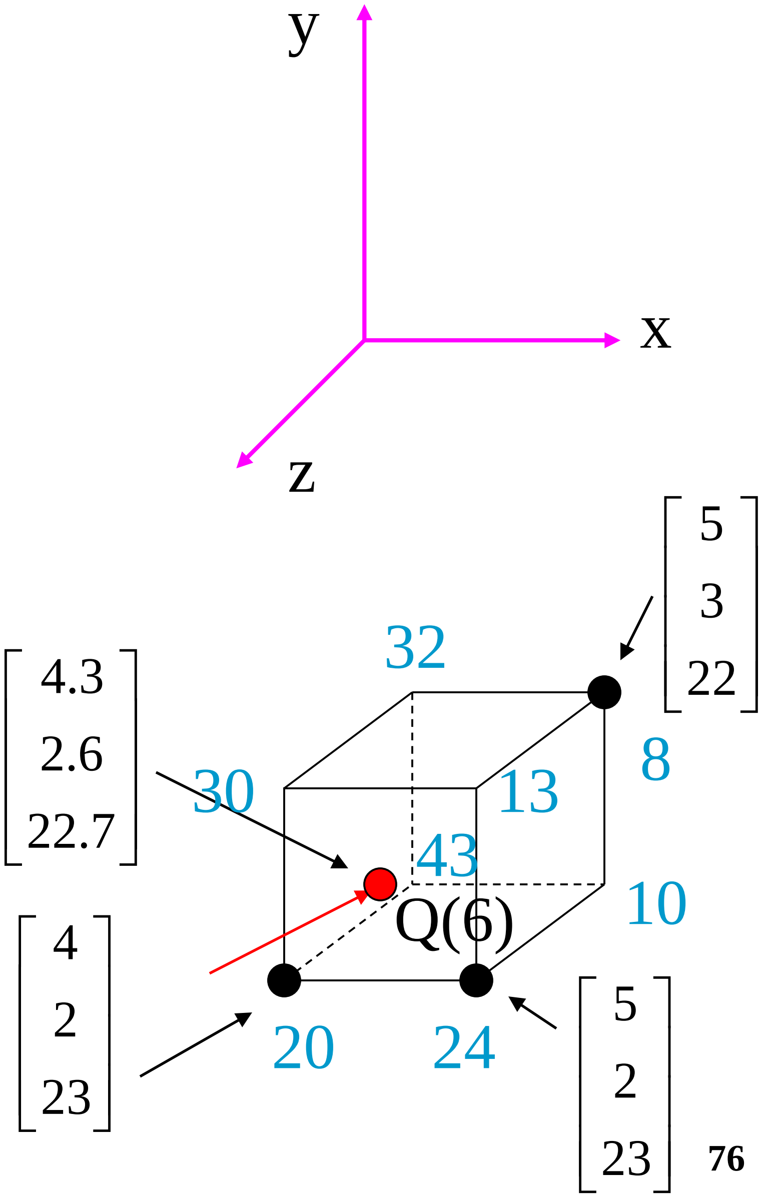
Orthographic (parallel) projection: Interpolation



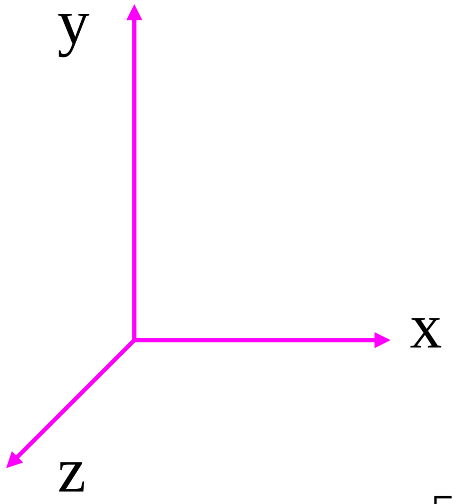
y
x
z
$\begin{bmatrix} 5 \\ 3 \\ 22 \end{bmatrix}$
32
$\begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$
30
13
8
43
Q(6)
10
$\begin{bmatrix} 4 \\ 2 \\ 23 \end{bmatrix}$
20
24
$\begin{bmatrix} 5 \\ 2 \\ 23 \end{bmatrix}$

Orthographic (parallel) projection: Nearest Neighbor Interpolation

$$Val(Q(6)) = Val(NearestNeighbor(Q(6))$$

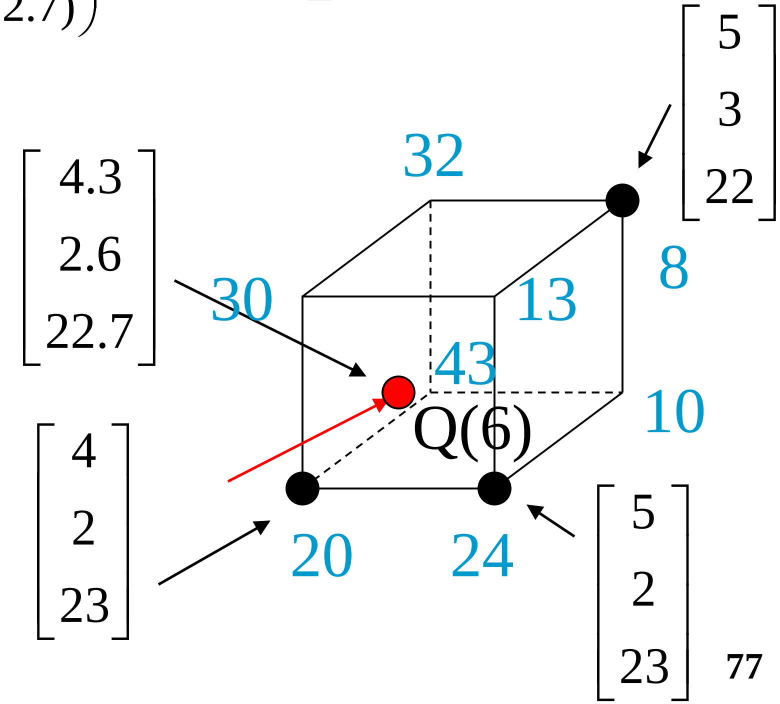


Orthographic (parallel) projection: Nearest Neighbor Interpolation



$$Val(Q(6)) = Val(NearestNeighbor(Q(6))$$

$$NearestNeighbor(Q(6)) = Round(Q(6)) = \begin{pmatrix} Round(4.3) \\ Round(2.6) \\ Round(22.7) \end{pmatrix}$$

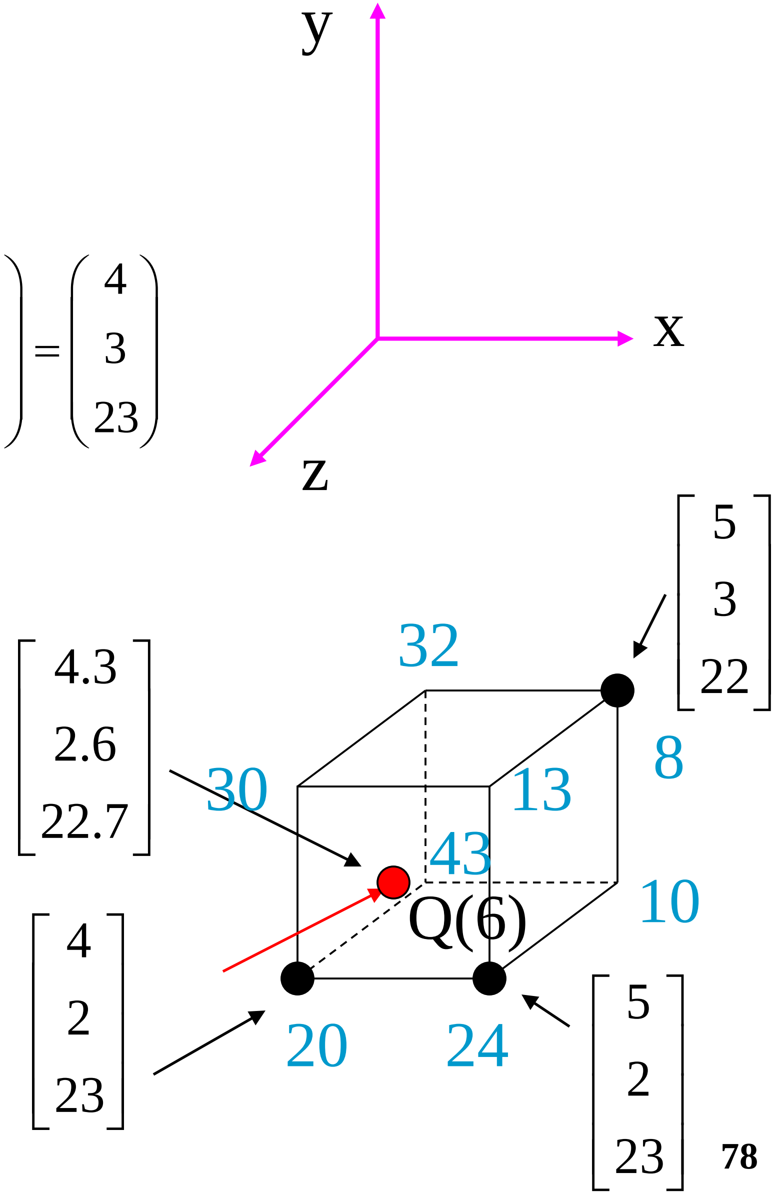

$\begin{bmatrix} 5 \\ 3 \\ 22 \end{bmatrix}$
32
$\begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$
8
30
13
43
10
Q(6)
$\begin{bmatrix} 4 \\ 2 \\ 23 \end{bmatrix}$
20
24
$\begin{bmatrix} 5 \\ 2 \\ 23 \end{bmatrix}$

Orthographic (parallel) projection: Nearest Neighbor Interpolation

$$Val(Q(6)) = Val(NearestNeighbor(Q(6))$$

$$NearestNeighbor(Q(6)) = Round(Q(6)) = \begin{pmatrix} Round(4.3) \\ Round(2.6) \\ Round(22.7) \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \\ 23 \end{pmatrix}$$

$$Val(Q(6)) = Val\begin{pmatrix} 4 \\ 3 \\ 23 \end{pmatrix}$$

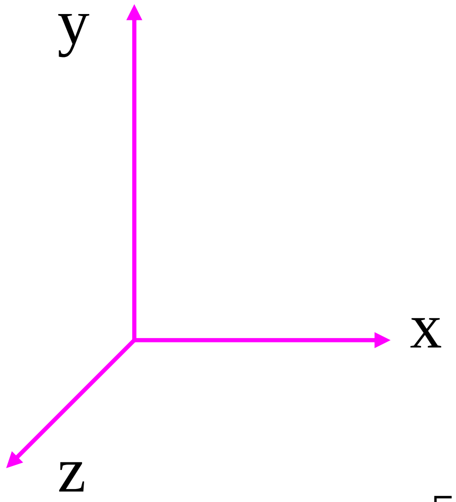


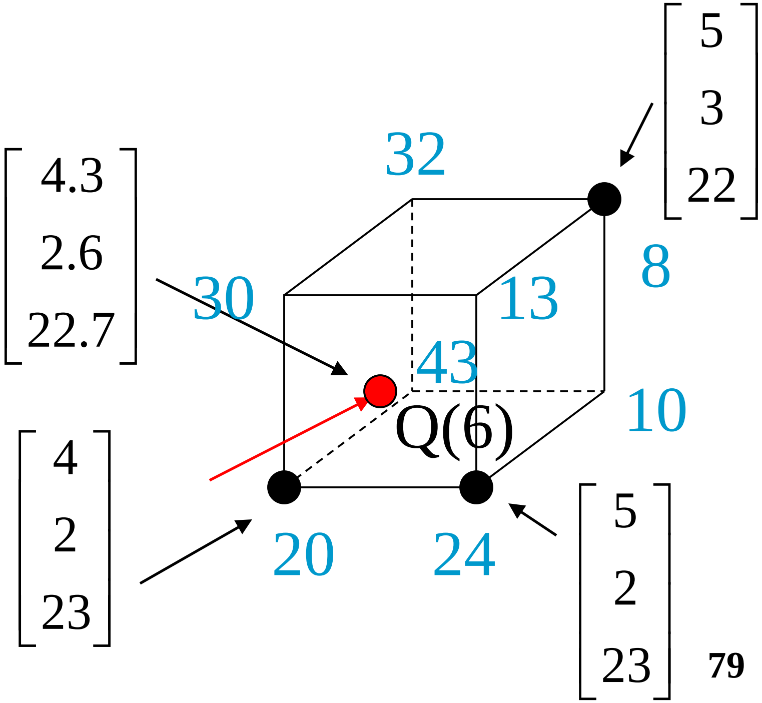
Orthographic (parallel) projection: Nearest Neighbor Interpolation

$$Val(Q(6)) = Val(NearestNeighbor(Q(6))$$

$$NearestNeighbor(Q(6)) = Round(Q(6)) = \begin{pmatrix} Round(4.3) \\ Round(2.6) \\ Round(22.7) \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \\ 23 \end{pmatrix}$$

$$Val(Q(6)) = Val\begin{pmatrix} 4 \\ 3 \\ 23 \end{pmatrix} = 30$$



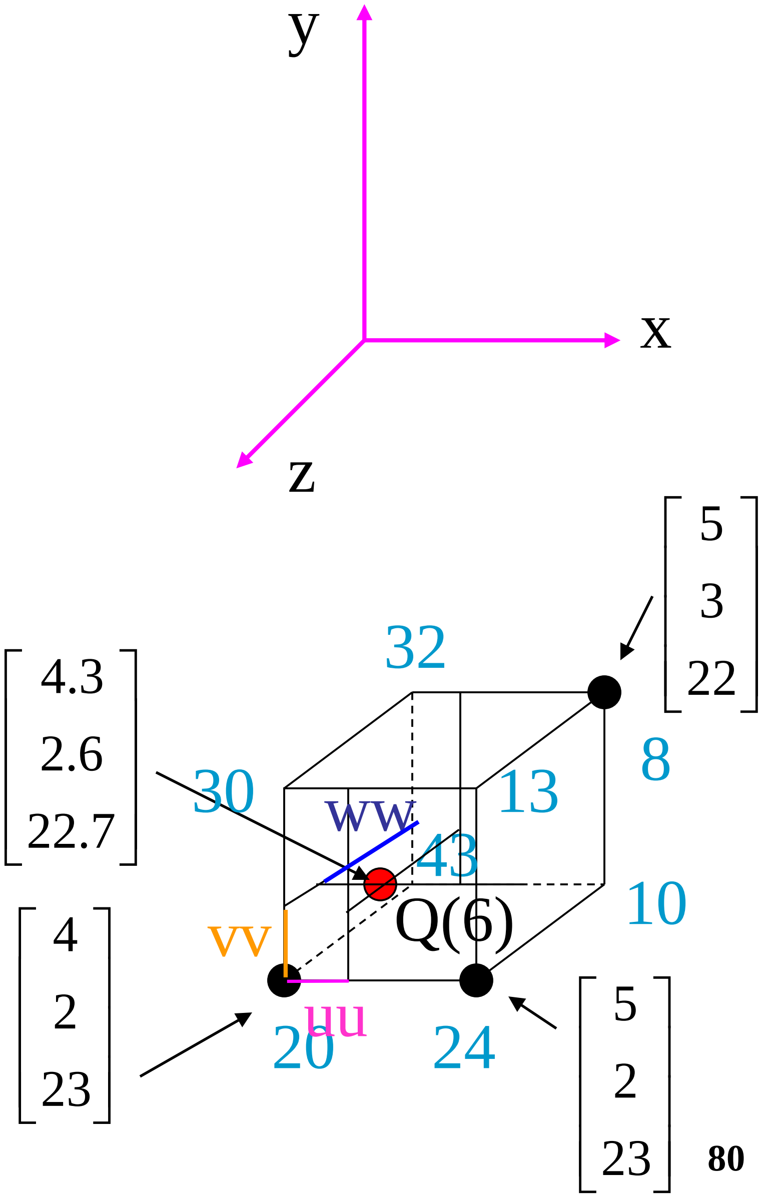


Orthographic (parallel) projection: Trilinear Interpolation

$$uu =$$

$$vv =$$

$$ww =$$

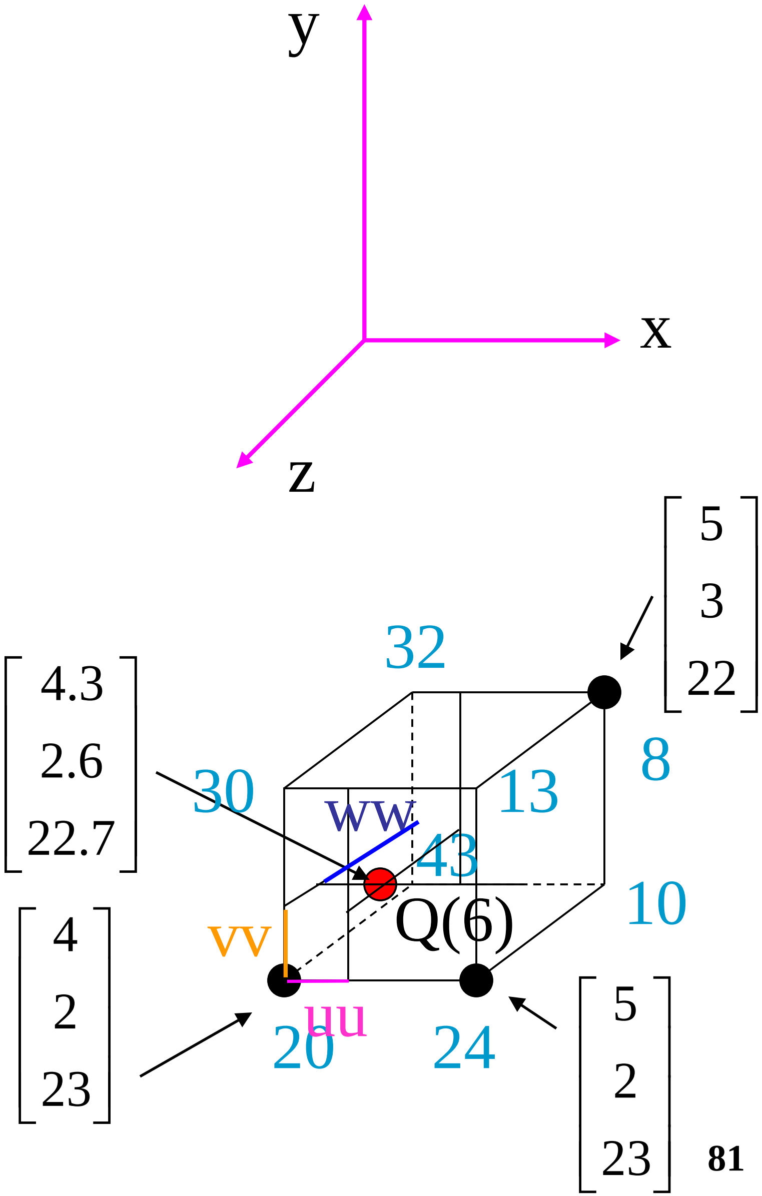


Orthographic (parallel) projection: Trilinear Interpolation

$$uu = Q(6)[0] - trunc(Q(6)[0]) =$$

$$vv =$$

$$ww =$$

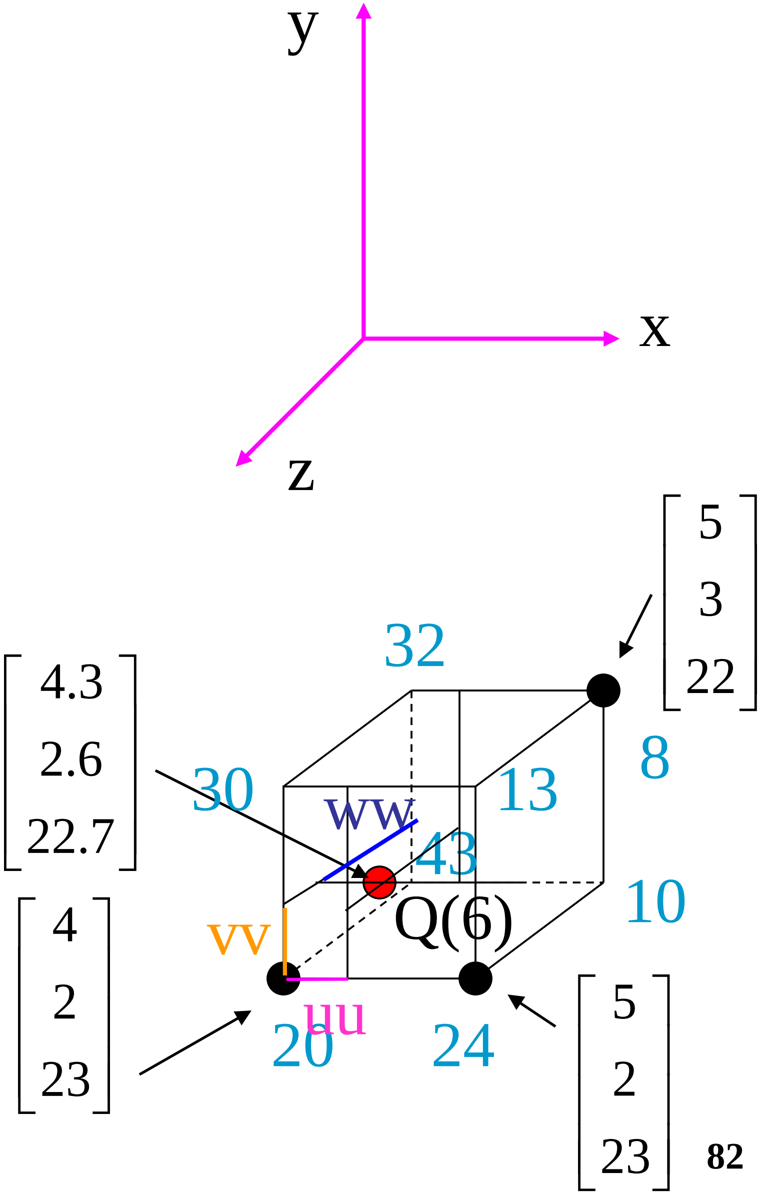


Orthographic (parallel) projection: Trilinear Interpolation

$$uu = Q(6)[0] - trunc(Q(6)[0]) = 4.3 - 4 =$$

$$vv =$$

$$ww =$$

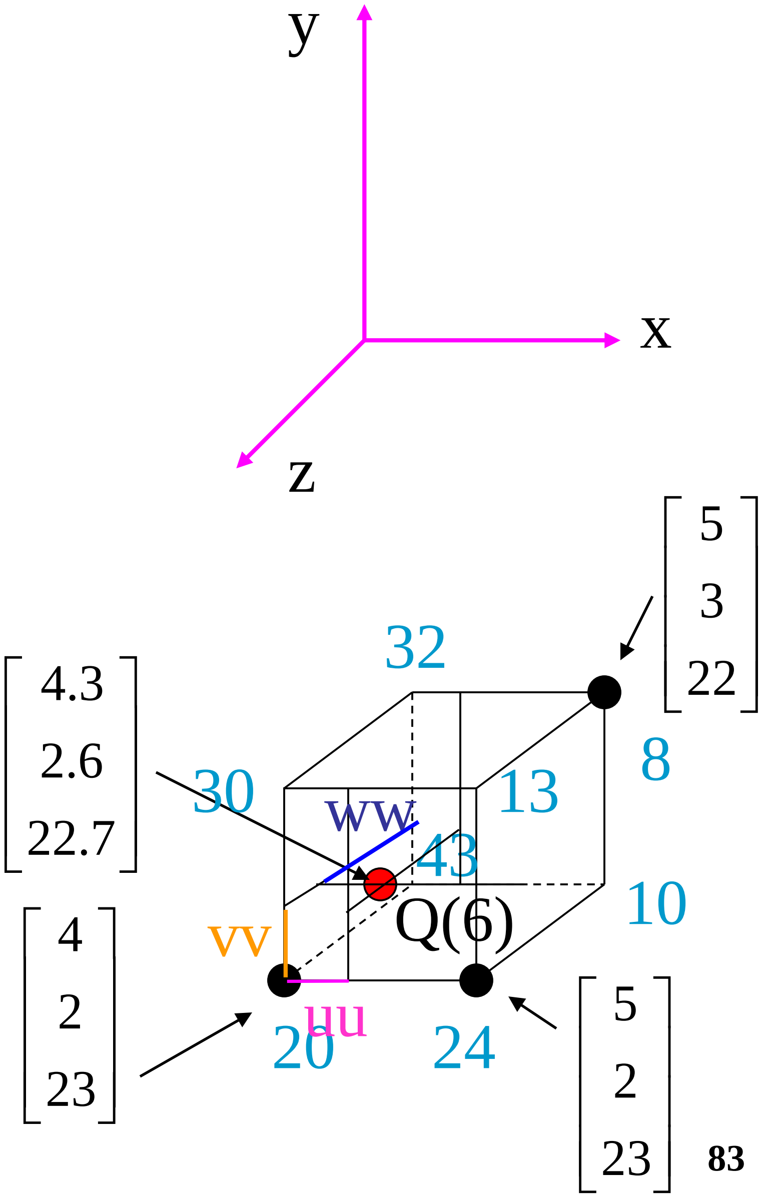


Orthographic (parallel) projection: Trilinear Interpolation

$$uu = Q(6)[0] - trunc(Q(6)[0]) = 4.3 - 4 = 0.3$$

$$vv =$$

$$ww =$$

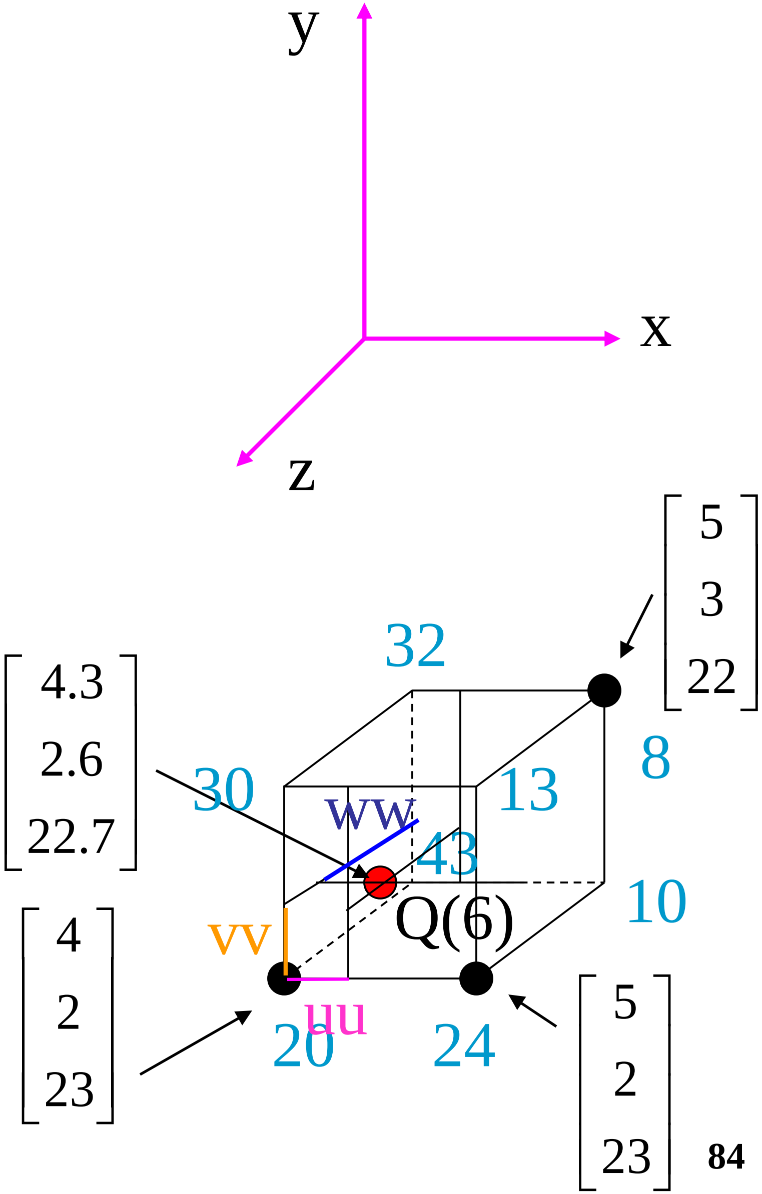


Orthographic (parallel) projection: Trilinear Interpolation

$$uu = Q(6)[0] - trunc(Q(6)[0]) = 4.3 - 4 = 0.3$$

$$vv = Q(6)[1] - trunc(Q(6)[1]) =$$

$$ww =$$

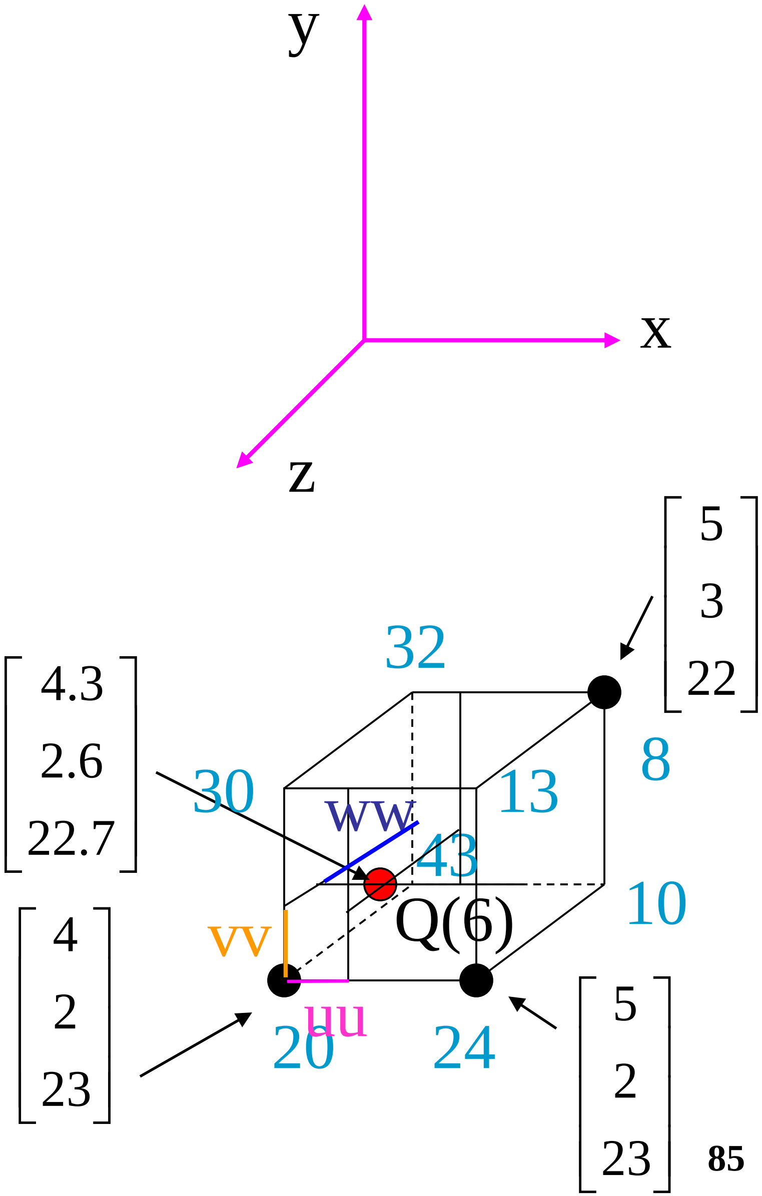

y
x
z
$\begin{bmatrix}5\\3\\22\end{bmatrix}$
32
$\begin{bmatrix}4.3\\2.6\\22.7\end{bmatrix}$
8
30
13
ww
43
10
Q(6)
$\begin{bmatrix}4\\2\\23\end{bmatrix}$
vv
uu
20
24
$\begin{bmatrix}5\\2\\23\end{bmatrix}$

Orthographic (parallel) projection: Trilinear Interpolation

$$uu = Q(6)[0] - trunc(Q(6)[0]) = 4.3 - 4 = 0.3$$

$$vv = Q(6)[1] - trunc(Q(6)[1]) = 2.6 - 2 =$$

$$ww =$$

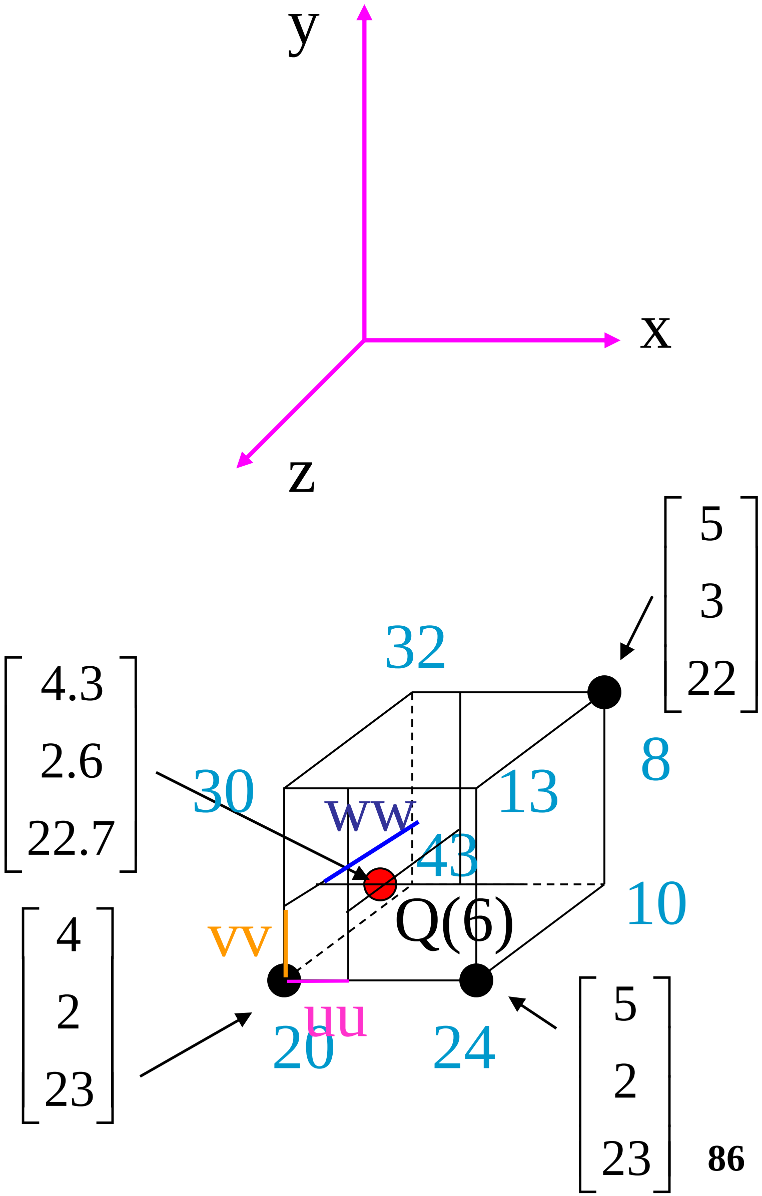

y
x
z
$\begin{bmatrix} 5 \\ 3 \\ 22 \end{bmatrix}$
32
$\begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$
8
30
13
ww
43
10
Q(6)
$\begin{bmatrix} 4 \\ 2 \\ 23 \end{bmatrix}$
vv
uu
20
24
$\begin{bmatrix} 5 \\ 2 \\ 23 \end{bmatrix}$

Orthographic (parallel) projection: Trilinear Interpolation

$$uu = Q(6)[0] - trunc(Q(6)[0]) = 4.3 - 4 = 0.3$$

$$vv = Q(6)[1] - trunc(Q(6)[1]) = 2.6 - 2 = 0.6$$

$$ww =$$

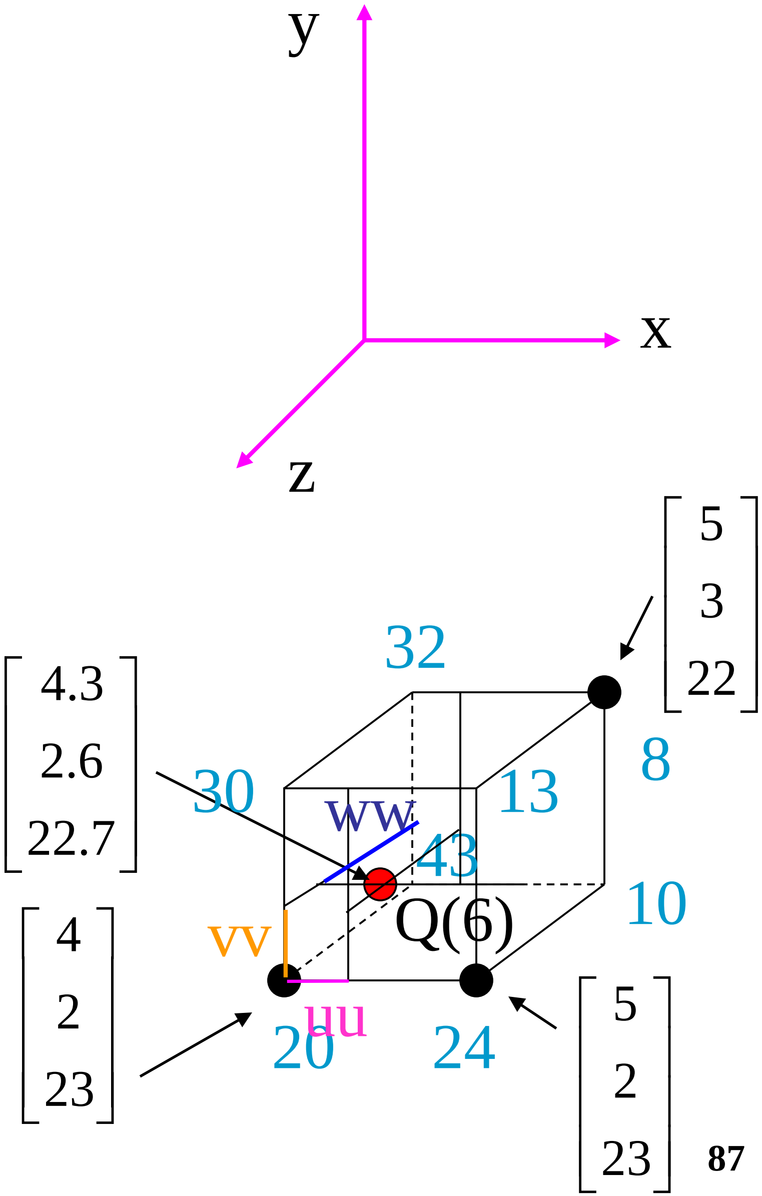


Orthographic (parallel) projection: Trilinear Interpolation

$$uu = Q(6)[0] - trunc(Q(6)[0]) = 4.3 - 4 = 0.3$$

$$vv = Q(6)[1] - trunc(Q(6)[1]) = 2.6 - 2 = 0.6$$

$$ww = Q(6)[2] - trunc(Q(6)[2]) =$$

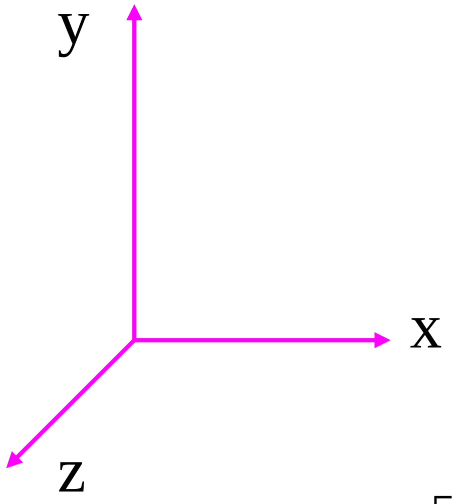


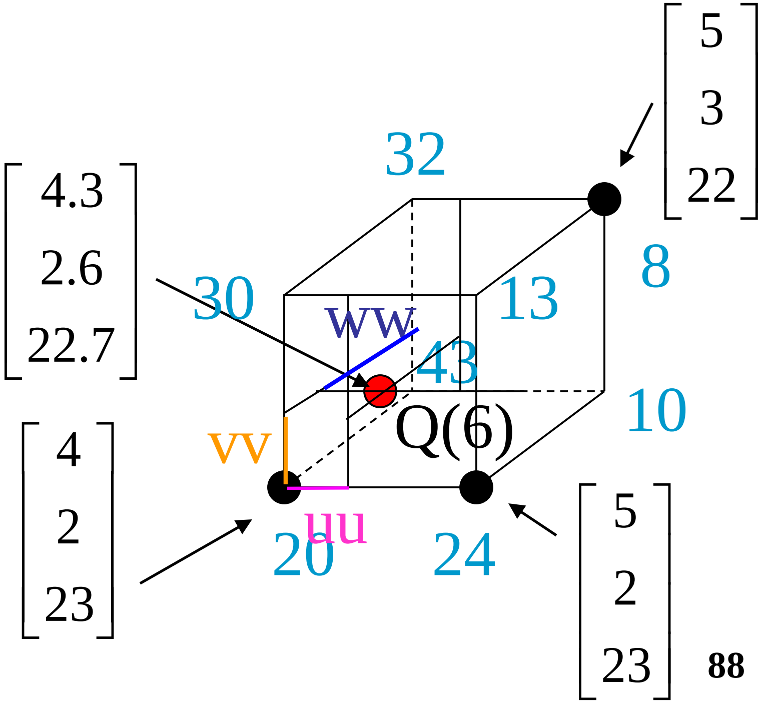
Orthographic (parallel) projection: Trilinear Interpolation

$$uu = Q(6)[0] - trunc(Q(6)[0]) = 4.3 - 4 = 0.3$$

$$vv = Q(6)[1] - trunc(Q(6)[1]) = 2.6 - 2 = 0.6$$

$$ww = Q(6)[2] - trunc(Q(6)[2]) = 22.7 - 22 =$$



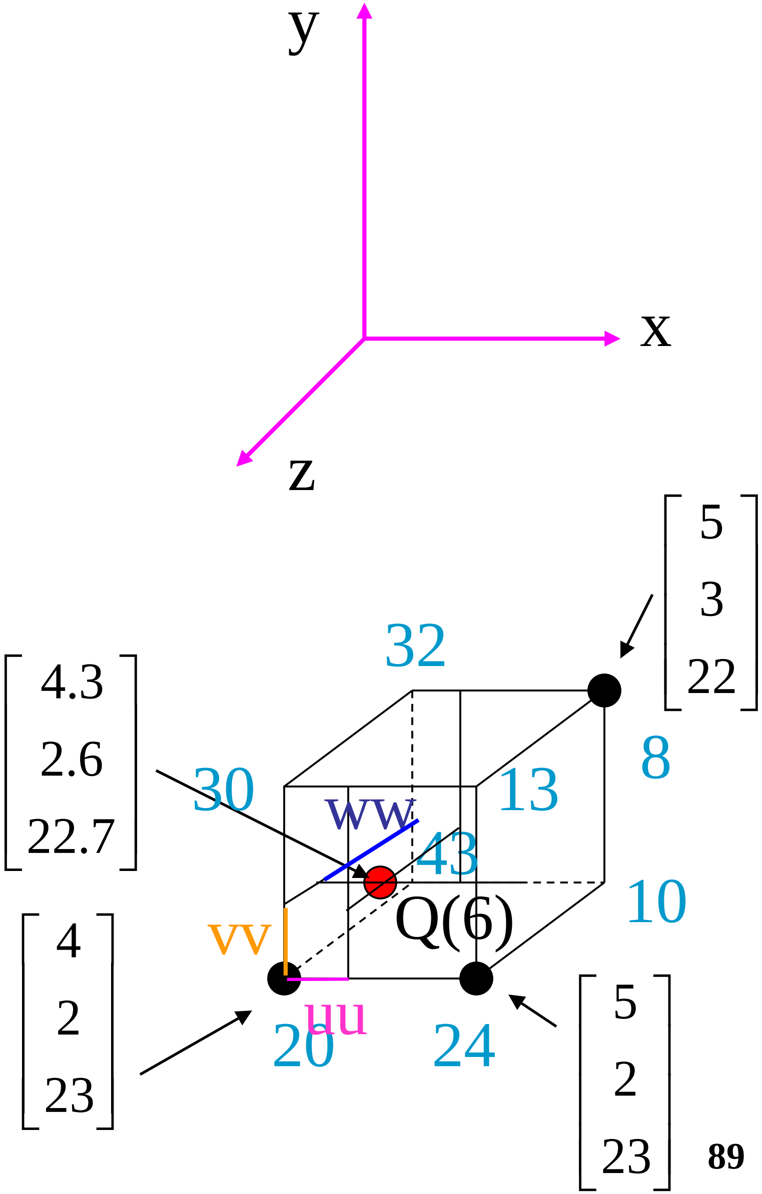


Orthographic (parallel) projection: Trilinear Interpolation

$$uu = Q(6)[0] - trunc(Q(6)[0]) = 4.3 - 4 = 0.3$$

$$vv = Q(6)[1] - trunc(Q(6)[1]) = 2.6 - 2 = 0.6$$

$$ww = Q(6)[2] - trunc(Q(6)[2]) = 22.7 - 22 = 0.7$$


y
x
z
32
30
13
8
43
10
ww
vv
uu
Q(6)
20
24
$\begin{bmatrix}5\\3\\22\end{bmatrix}$
$\begin{bmatrix}4.3\\2.6\\22.7\end{bmatrix}$
$\begin{bmatrix}4\\2\\23\end{bmatrix}$
$\begin{bmatrix}5\\2\\23\end{bmatrix}$

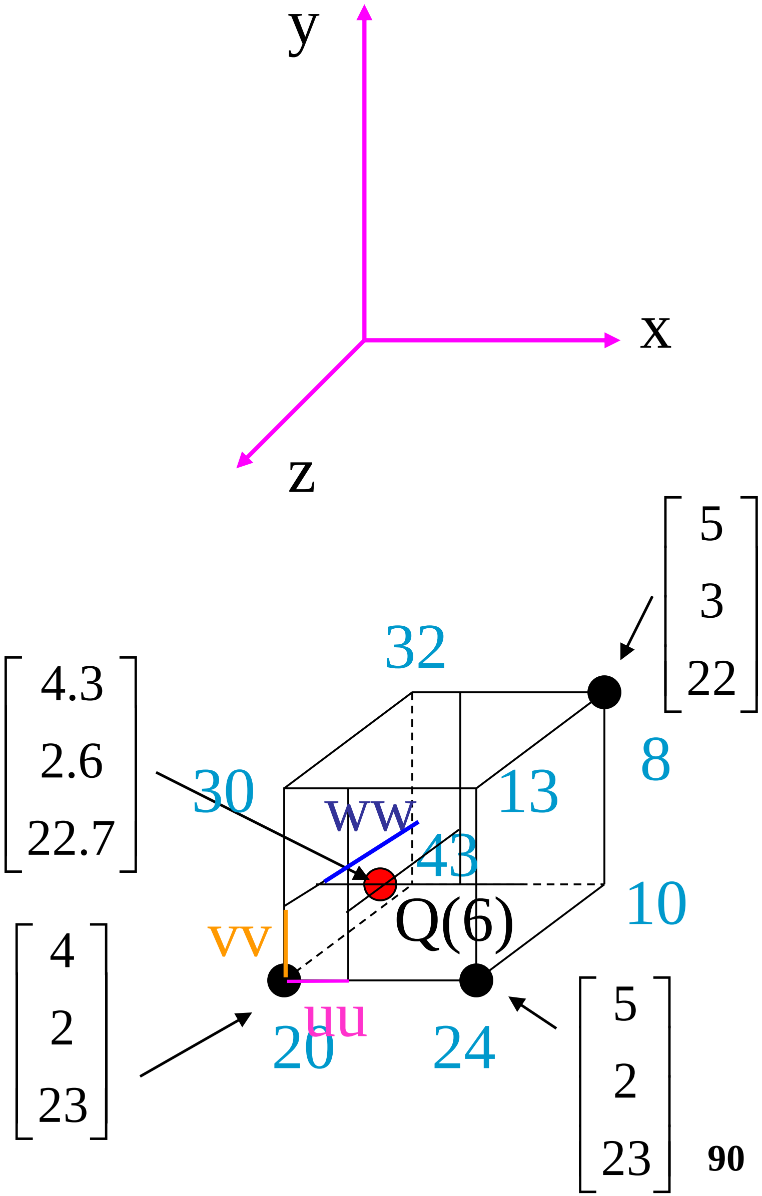
Orthographic (parallel) projection:
Trilinear Interpolation

$$uu = Q(6)[0] - trunc(Q(6)[0]) = 4.3 - 4 = 0.3$$

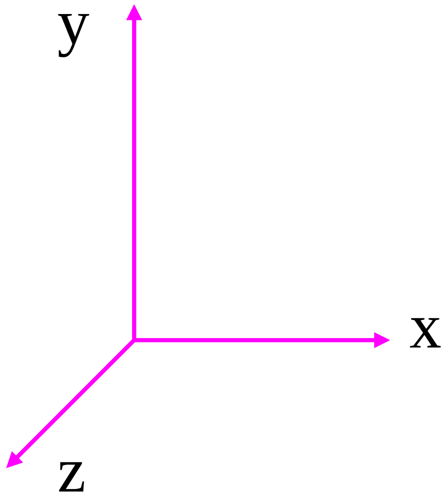
$$vv = Q(6)[1] - trunc(Q(6)[1]) = 2.6 - 2 = 0.6$$

$$ww = Q(6)[2] - trunc(Q(6)[2]) = 22.7 - 22 = 0.7$$

$$\begin{bmatrix} x0 \\ y0 \\ z0 \end{bmatrix} = trunc\begin{pmatrix} Q(6)[0] \\ Q(6)[1] \\ Q(6)[2] \end{pmatrix} = \begin{bmatrix} 4 \\ 2 \\ 22 \end{bmatrix}$$

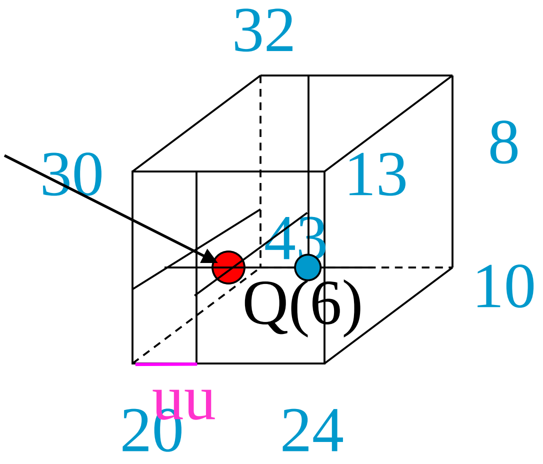


Orthographic (parallel) projection:
1st linear interpolation: along x

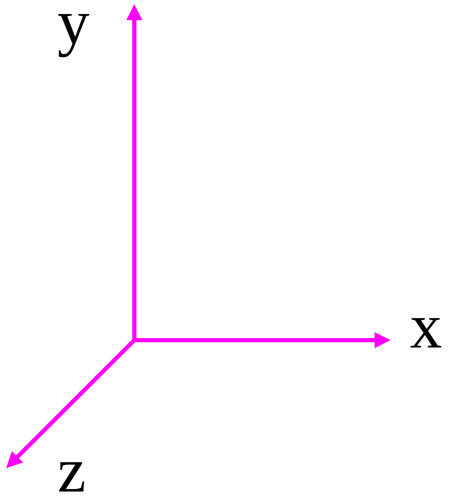


$P(uu, y0, z0) =$

$$\begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$$



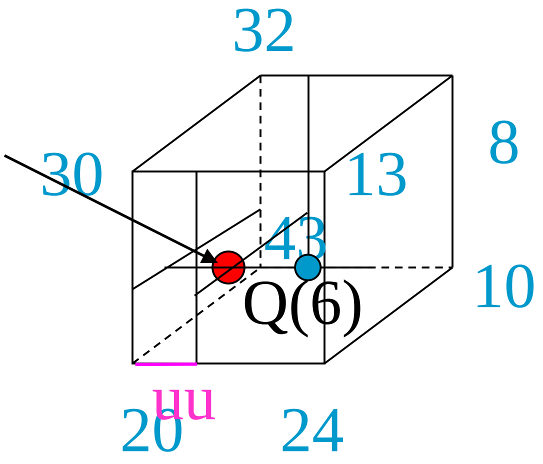
Orthographic (parallel) projection:
1st linear interpolation: along x



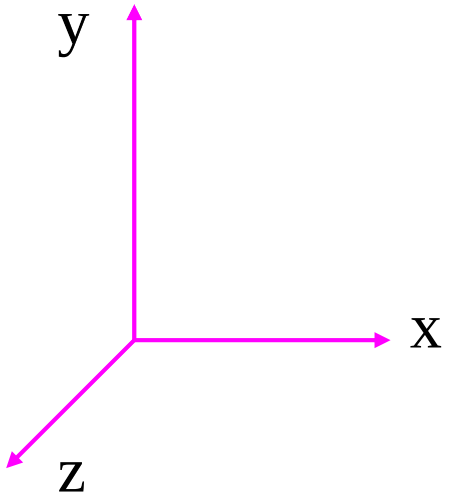
$$P(uu, y0, z0) = uu \cdot Val\left(\begin{bmatrix} x0+1 \\ y0 \\ z0 \end{bmatrix}\right) + (1-uu) \cdot Val\left(\begin{bmatrix} x0 \\ y0 \\ z0 \end{bmatrix}\right)$$

$$=$$

$$\begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$$

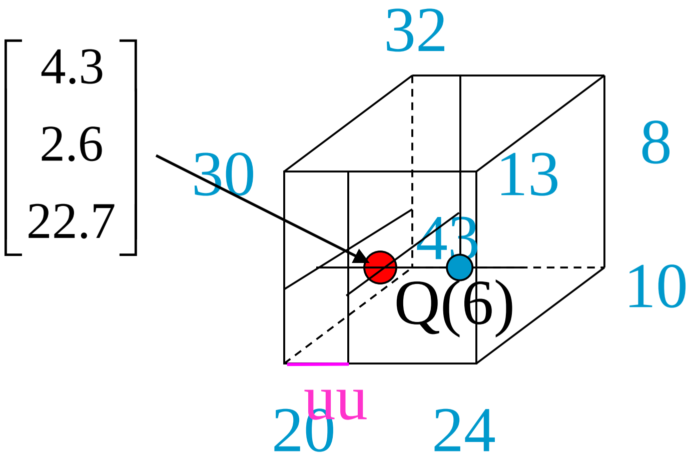


Orthographic (parallel) projection:
1st linear interpolation: along x

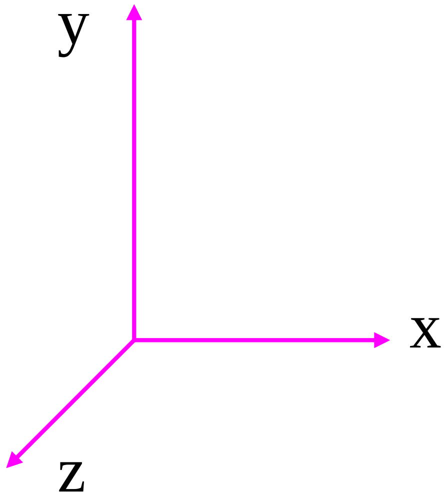


$$P(uu, y0, z0) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0 \\ z0 \end{bmatrix}) + (1-uu)\cdot Val(\begin{bmatrix} x0 \\ y0 \\ z0 \end{bmatrix})$$

$$= 0.3 \cdot 10 + 0.7 \cdot 43 =$$

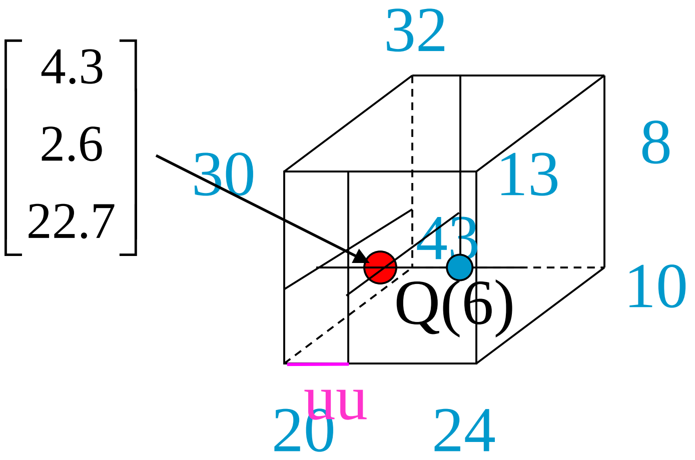


Orthographic (parallel) projection: 1st linear interpolation: along x

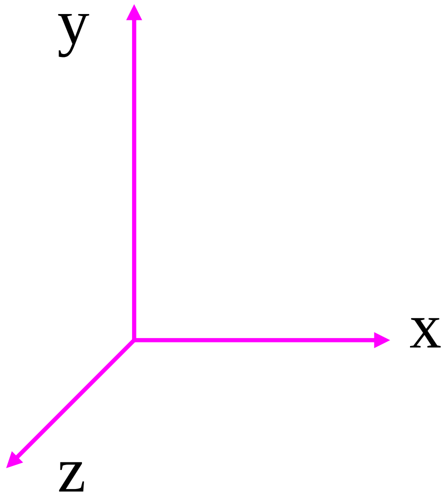


$$P(uu, y0, z0) = uu \cdot Val\left(\begin{bmatrix} x0+1 \\ y0 \\ z0 \end{bmatrix}\right) + (1-uu)\cdot Val\left(\begin{bmatrix} x0 \\ y0 \\ z0 \end{bmatrix}\right)$$

$$= 0.3 \cdot 10 + 0.7 \cdot 43 = 33.1$$


$$\begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$$

Orthographic (parallel) projection:
1st linear interpolation: along x

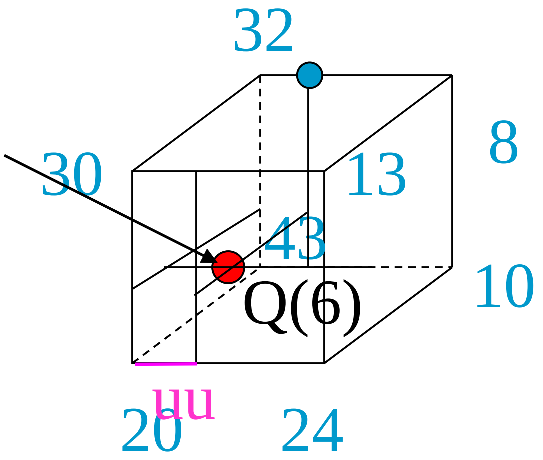


$$P(uu, y0, z0) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0 \\ z0 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0 \\ z0 \end{bmatrix})$$

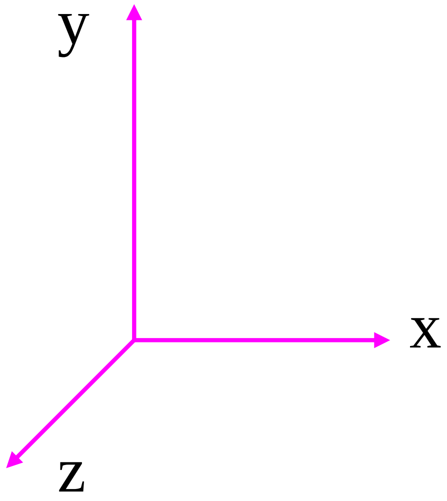
$$= 0.3 \cdot 10 + 0.7 \cdot 43 = 33.1$$

$$P(uu, y0+1, z0) =$$

$$\begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$$



Orthographic (parallel) projection:
1st linear interpolation: along x

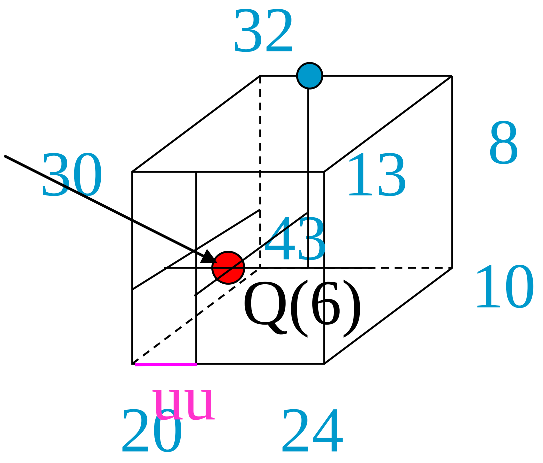


$$P(uu, y0, z0) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0 \\ z0 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0 \\ z0 \end{bmatrix})$$

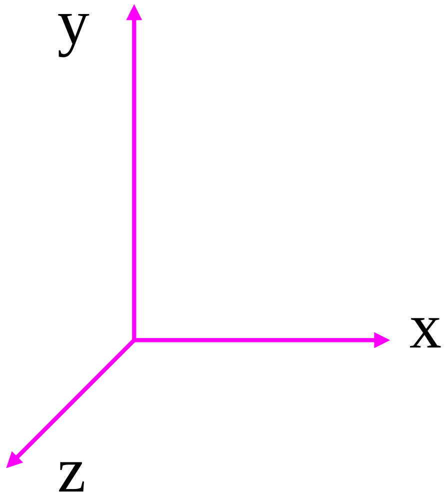
$$= 0.3 \cdot 10 + 0.7 \cdot 43 = 33.1$$

$$P(uu, y0+1, z0) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0+1 \\ z0 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0+1 \\ z0 \end{bmatrix})$$

$$\begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$$



Orthographic (parallel) projection:
1st linear interpolation: along x

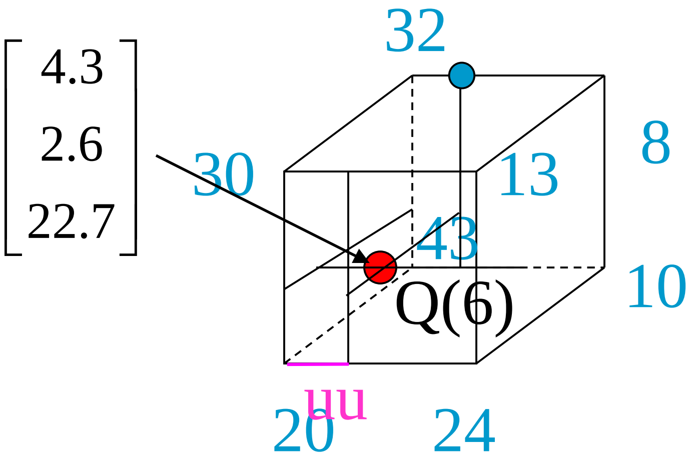


$$P(uu, y0, z0) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0 \\ z0 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0 \\ z0 \end{bmatrix})$$

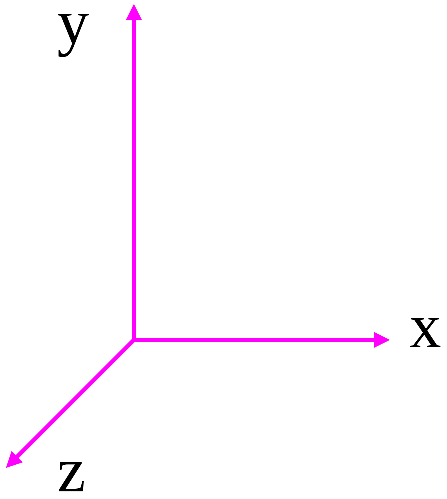
$$= 0.3 \cdot 10 + 0.7 \cdot 43 = 33.1$$

$$P(uu, y0+1, z0) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0+1 \\ z0 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0+1 \\ z0 \end{bmatrix})$$

$$= 0.3 \cdot 8 + 0.7 \cdot 32 = 24.8$$



Orthographic (parallel) projection:
1st linear interpolation: along x



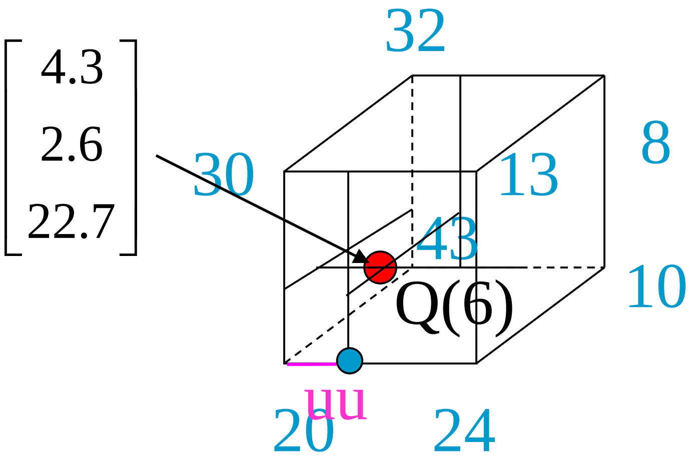
$$P(uu, y0, z0) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0 \\ z0 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0 \\ z0 \end{bmatrix})$$

$$= 0.3 \cdot 10 + 0.7 \cdot 43 = 33.1$$

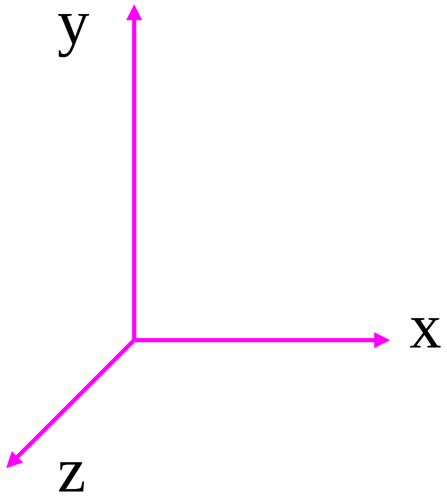
$$P(uu, y0+1, z0) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0+1 \\ z0 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0+1 \\ z0 \end{bmatrix})$$

$$= 0.3 \cdot 8 + 0.7 \cdot 32 = 24.8$$

$$P(uu, y0, z0+1) =$$



Orthographic (parallel) projection:
1st linear interpolation: along x



$$P(uu, y0, z0) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0 \\ z0 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0 \\ z0 \end{bmatrix})$$

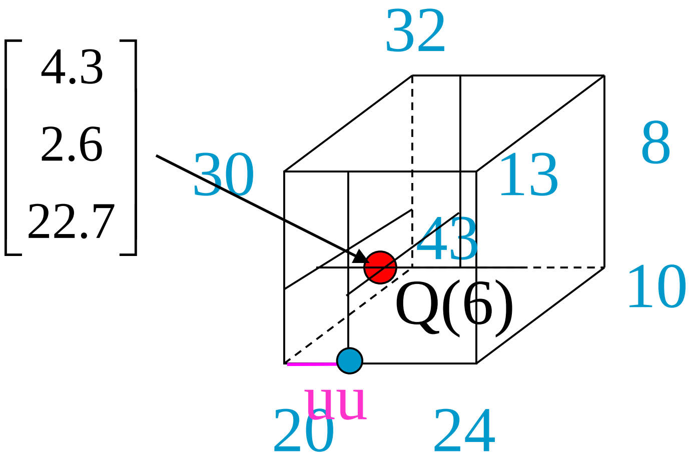
$$= 0.3 \cdot 10 + 0.7 \cdot 43 = 33.1$$

$$P(uu, y0+1, z0) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0+1 \\ z0 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0+1 \\ z0 \end{bmatrix})$$

$$= 0.3 \cdot 8 + 0.7 \cdot 32 = 24.8$$

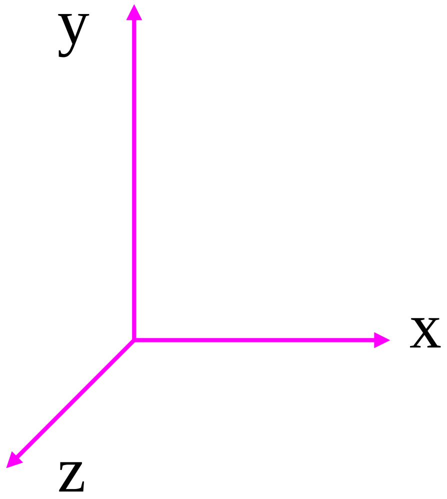
$$P(uu, y0, z0+1) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0 \\ z0+1 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0 \\ z0+1 \end{bmatrix})$$

$$=$$



$$\begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$$

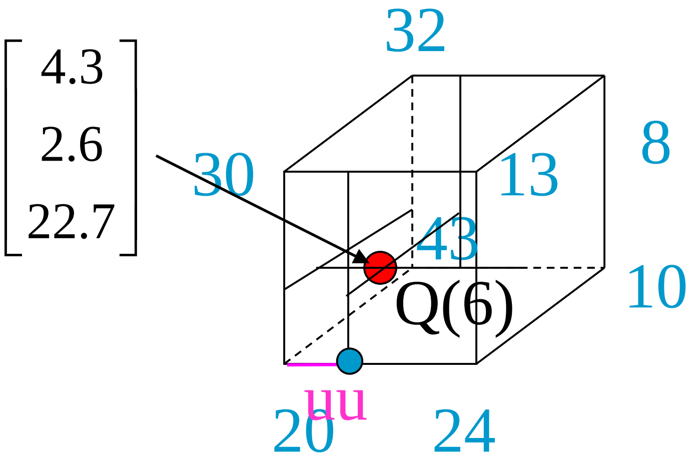
Orthographic (parallel) projection:
1st linear interpolation: along x



$$P(uu, y0, z0) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0 \\ z0 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0 \\ z0 \end{bmatrix})$$
$$= 0.3 \cdot 10 + 0.7 \cdot 43 = 33.1$$

$$P(uu, y0+1, z0) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0+1 \\ z0 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0+1 \\ z0 \end{bmatrix})$$
$$= 0.3 \cdot 8 + 0.7 \cdot 32 = 24.8$$

$$P(uu, y0, z0+1) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0 \\ z0+1 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0 \\ z0+1 \end{bmatrix})$$
$$= 0.3 \cdot 24 + 0.7 \cdot 20 = 21.2$$



Orthographic (parallel) projection:
1st linear interpolation: along x

$$P(uu, y0, z0) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0 \\ z0 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0 \\ z0 \end{bmatrix})$$

$$= 0.3 \cdot 10 + 0.7 \cdot 43 = 33.1$$

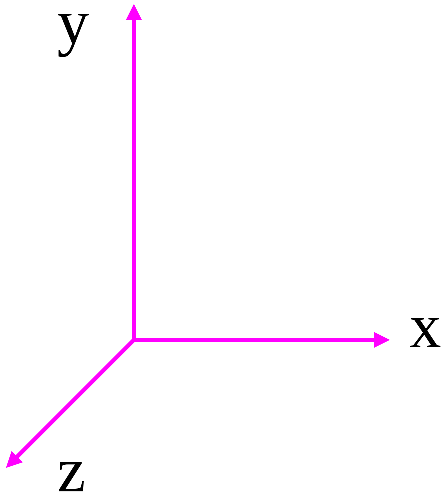
$$P(uu, y0+1, z0) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0+1 \\ z0 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0+1 \\ z0 \end{bmatrix})$$

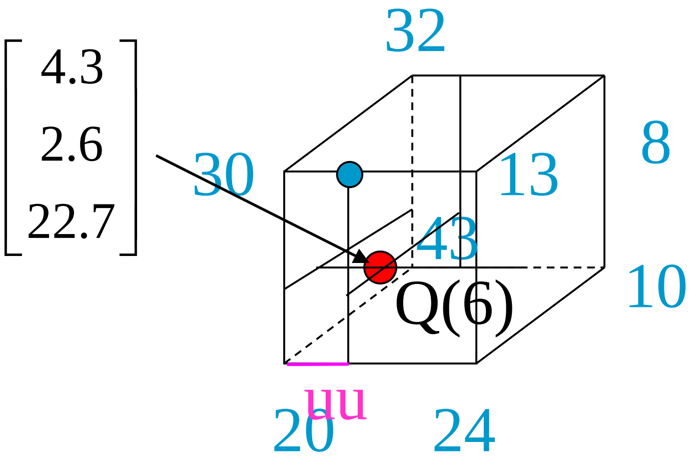
$$= 0.3 \cdot 8 + 0.7 \cdot 32 = 24.8$$

$$P(uu, y0, z0+1) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0 \\ z0+1 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0 \\ z0+1 \end{bmatrix})$$

$$= 0.3 \cdot 24 + 0.7 \cdot 20 = 21.2$$

$$P(uu, y0+1, z0+1) =$$





$$\begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$$

Orthographic (parallel) projection:
1st linear interpolation: along x

$$P(uu, y0, z0) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0 \\ z0 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0 \\ z0 \end{bmatrix})$$

$$= 0.3 \cdot 10 + 0.7 \cdot 43 = 33.1$$

$$P(uu, y0+1, z0) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0+1 \\ z0 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0+1 \\ z0 \end{bmatrix})$$

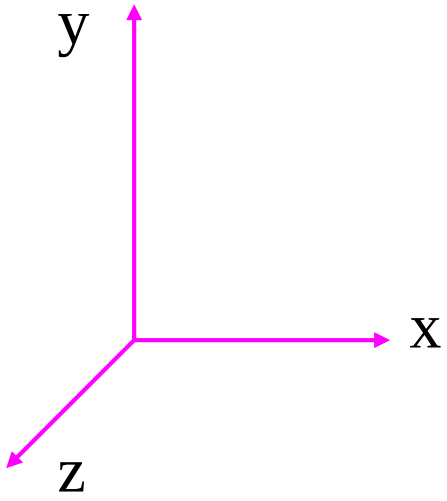
$$= 0.3 \cdot 8 + 0.7 \cdot 32 = 24.8$$

$$P(uu, y0, z0+1) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0 \\ z0+1 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0 \\ z0+1 \end{bmatrix})$$

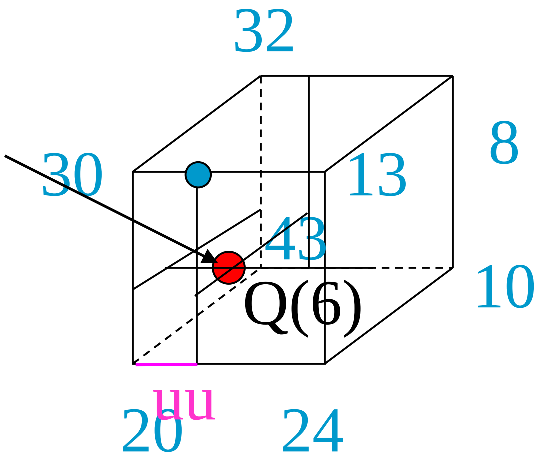
$$= 0.3 \cdot 24 + 0.7 \cdot 20 = 21.2$$

$$P(uu, y0+1, z0+1) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0+1 \\ z0+1 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0+1 \\ z0+1 \end{bmatrix})$$

$$=$$



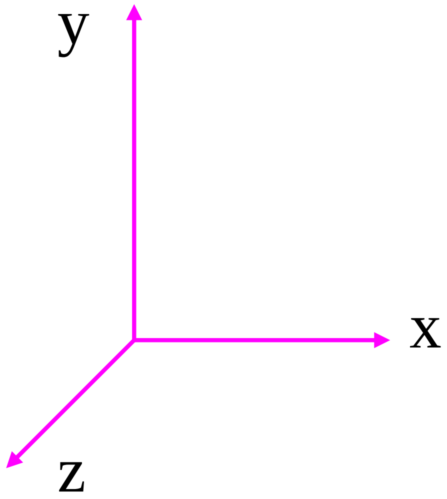
$$\begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$$



Orthographic (parallel) projection:
1st linear interpolation: along x

$$P(uu, y0, z0) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0 \\ z0 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0 \\ z0 \end{bmatrix})$$

$$= 0.3 \cdot 10 + 0.7 \cdot 43 = 33.1$$

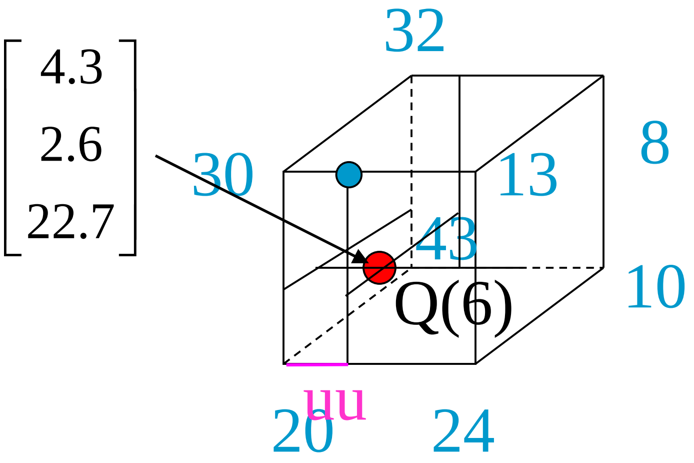


$$P(uu, y0+1, z0) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0+1 \\ z0 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0+1 \\ z0 \end{bmatrix})$$

$$= 0.3 \cdot 8 + 0.7 \cdot 32 = 24.8$$

$$P(uu, y0, z0+1) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0 \\ z0+1 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0 \\ z0+1 \end{bmatrix})$$

$$= 0.3 \cdot 24 + 0.7 \cdot 20 = 21.2$$

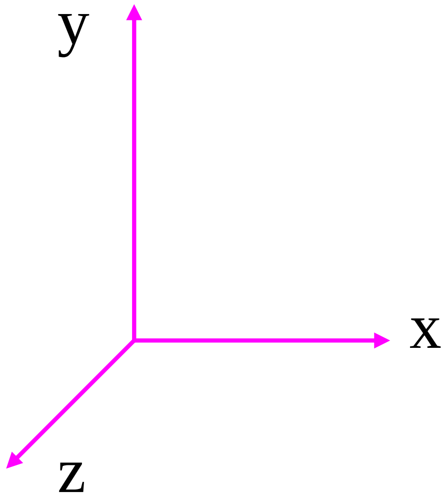


$$P(uu, y0+1, z0+1) = uu \cdot Val(\begin{bmatrix} x0+1 \\ y0+1 \\ z0+1 \end{bmatrix}) + (1-uu) \cdot Val(\begin{bmatrix} x0 \\ y0+1 \\ z0+1 \end{bmatrix})$$

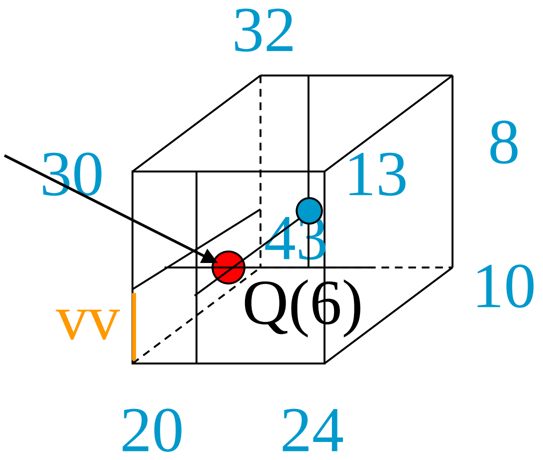
$$= 0.3 \cdot 13 + 0.7 \cdot 30 = 24.9$$

Orthographic (parallel) projection: 2nd linear interpolation: along y

$P(uu, vv, z0) =$



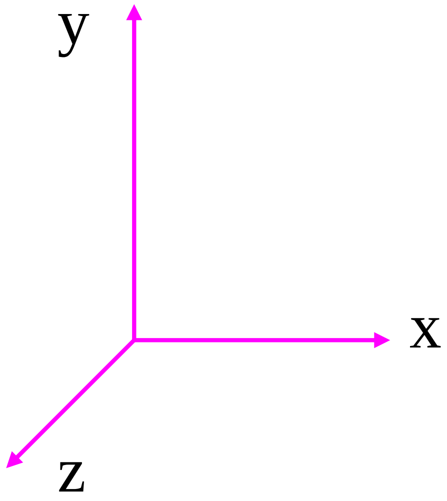
$$\begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$$

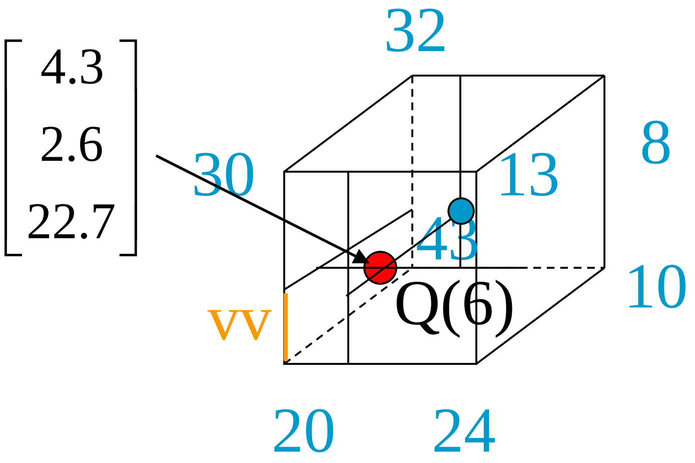


Orthographic (parallel) projection:
2nd linear interpolation: along y

$$P(uu,vv,z0) = vv \cdot Val(P(uu,y0+1,z0) + (1-vv) \cdot Val(uu,y0,z0)$$

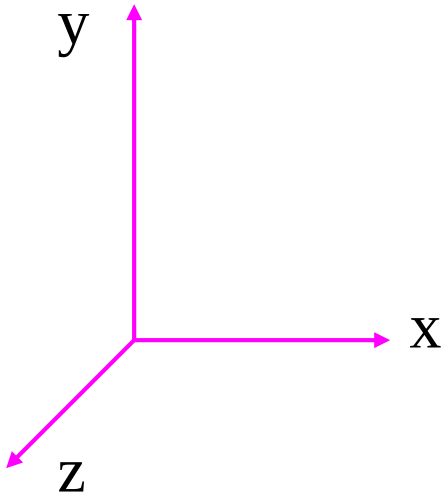
$$=$$





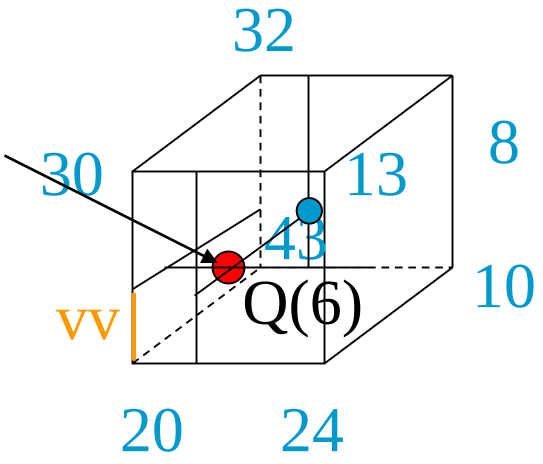
$$\begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$$

Orthographic (parallel) projection:
2nd linear interpolation: along y



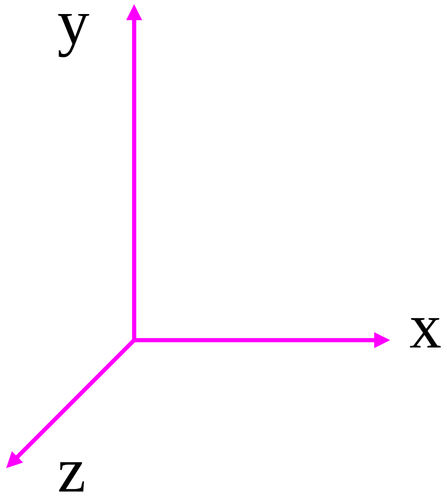
$$P(uu, vv, z0) = vv \cdot Val(P(uu, y0+1, z0) + (1-vv) \cdot Val(uu, y0, z0)$$
$$= 0.6 \cdot 24.8 + 0.4 \cdot 33.1 =$$

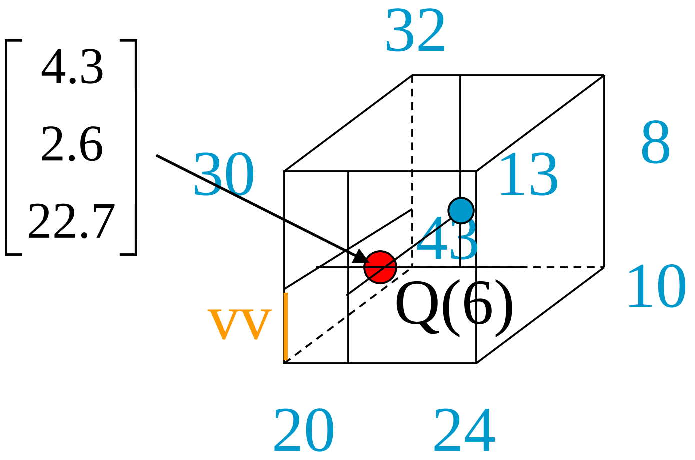
$$\begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$$



Orthographic (parallel) projection:
2nd linear interpolation: along y

$$
\begin{aligned}
P(uu, vv, z0) &= vv \cdot Val(P(uu, y0+1, z0) + (1-vv) \cdot Val(uu, y0, z0) \\
&= 0.6 \cdot 24.8 + 0.4 \cdot 33.1 = 28.12
\end{aligned}
$$

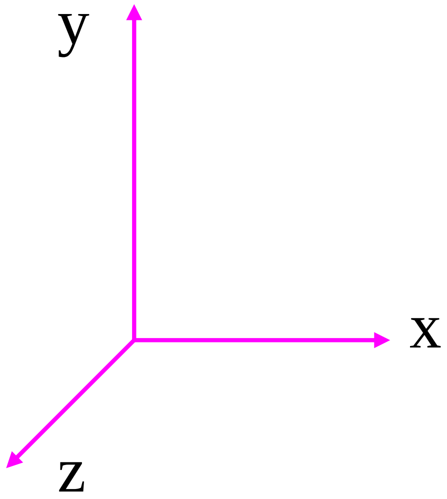



$$\begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$$

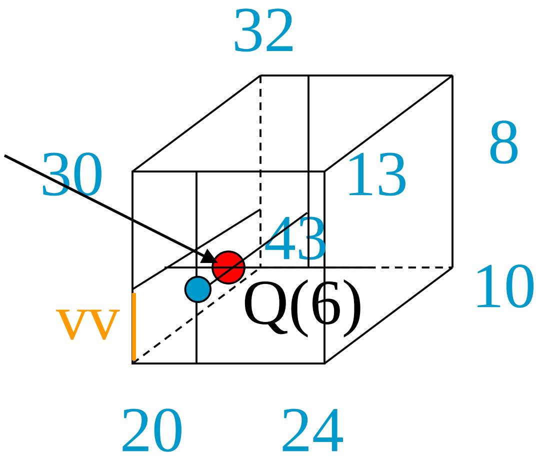
Orthographic (parallel) projection:
2nd linear interpolation: along y

$$P(uu, vv, z0) = vv \cdot Val(P(uu, y0+1, z0) + (1-vv) \cdot Val(uu, y0, z0)$$
$$= 0.6 \cdot 24.8 + 0.4 \cdot 33.1 = 28.12$$

$$P(uu, vv, z0+1) =$$



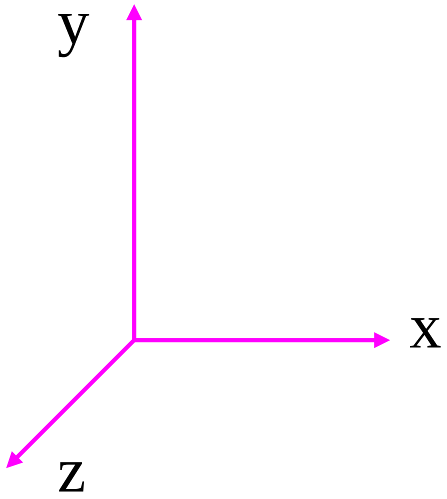
$$\begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$$

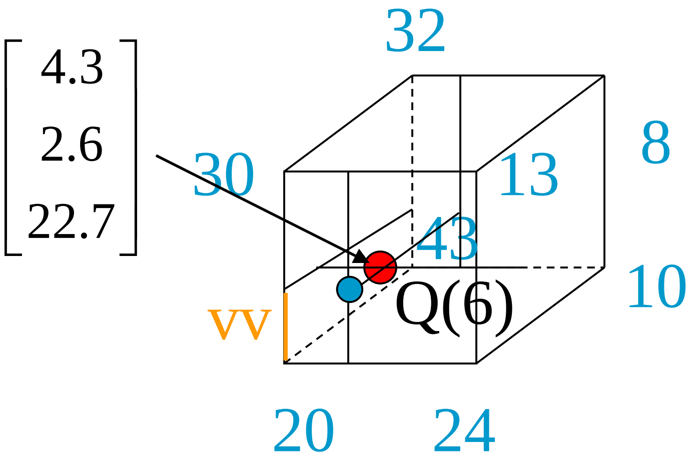


Orthographic (parallel) projection:
2nd linear interpolation: along y

$$P(uu,vv,z0) = vv \cdot Val(P(uu,y0+1,z0) + (1-vv)\cdot Val(uu,y0,z0)$$
$$= 0.6 \cdot 24.8 + 0.4 \cdot 33.1 = 28.12$$

$$P(uu,vv,z0+1) = vv \cdot Val(P(uu,y0+1,z0+1) + (1-vv)\cdot Val(uu,y0,z0+1)$$
$$=$$

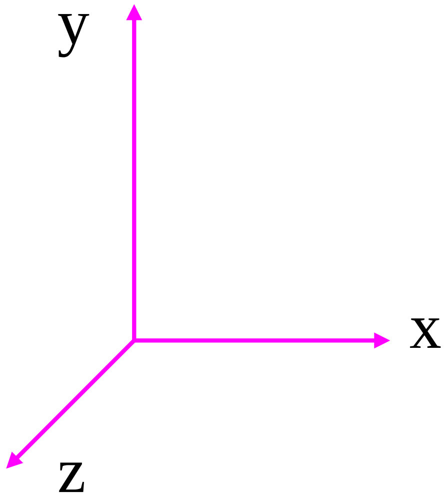


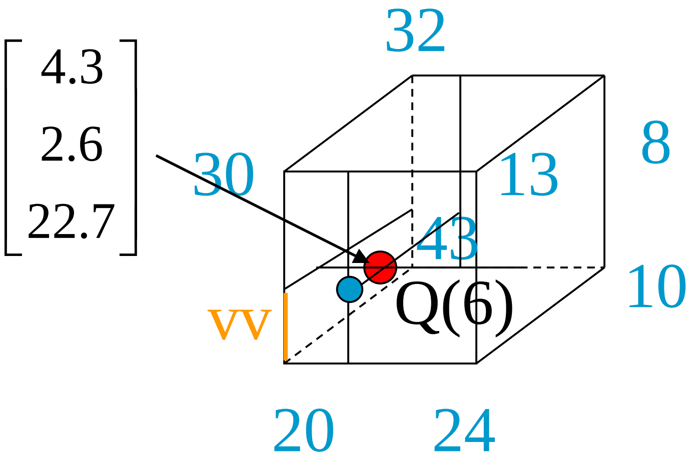

$$\begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$$

Orthographic (parallel) projection:
2^{nd} linear interpolation: along y

$$P(uu,vv,z0) = vv \cdot Val(P(uu,y0+1,z0) + (1-vv) \cdot Val(uu,y0,z0)$$
$$= 0.6 \cdot 24.8 + 0.4 \cdot 33.1 = 28.12$$

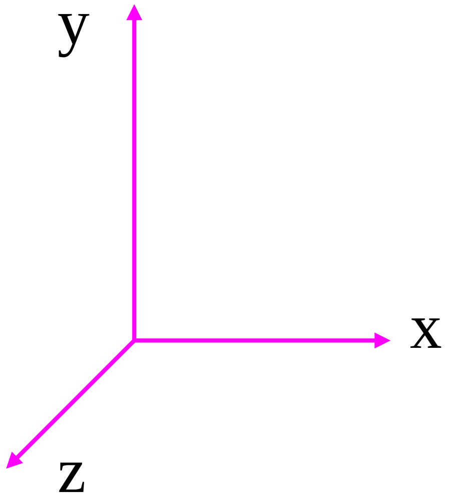
$$P(uu,vv,z0+1) = vv \cdot Val(P(uu,y0+1,z0+1) + (1-vv) \cdot Val(uu,y0,z0+1)$$
$$= 0.6 \cdot 24.9 + 0.4 \cdot 21.2 = 23.42$$

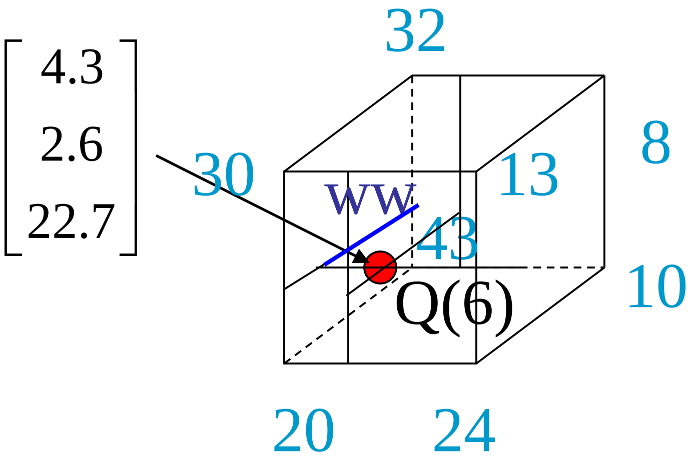




Orthographic (parallel) projection:
3rd linear interpolation: along z

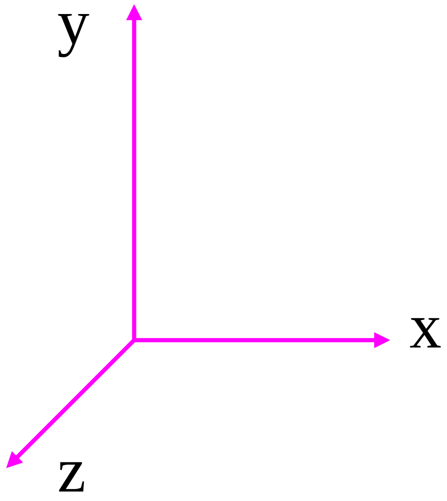
$$P(uu, vv, ww) =$$





$$\begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$$

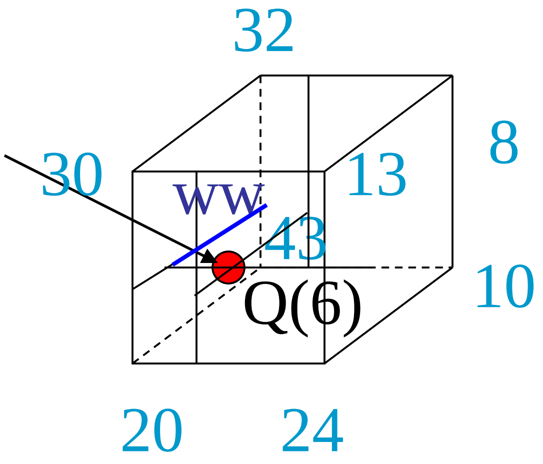
Orthographic (parallel) projection:
3rd linear interpolation: along z



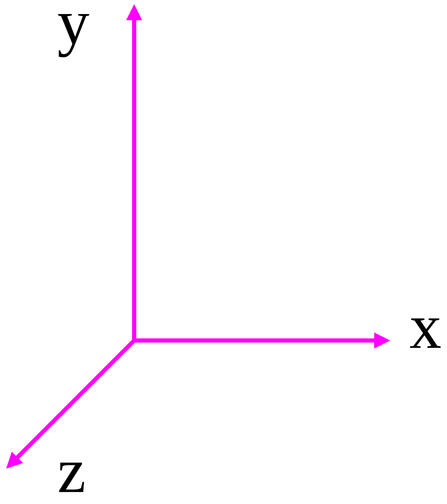
$$P(uu,vv,ww) = ww \cdot Val(P(uu,vv,z0+1) + (1-ww) \cdot Val(uu,vv,z0)$$

$$=$$

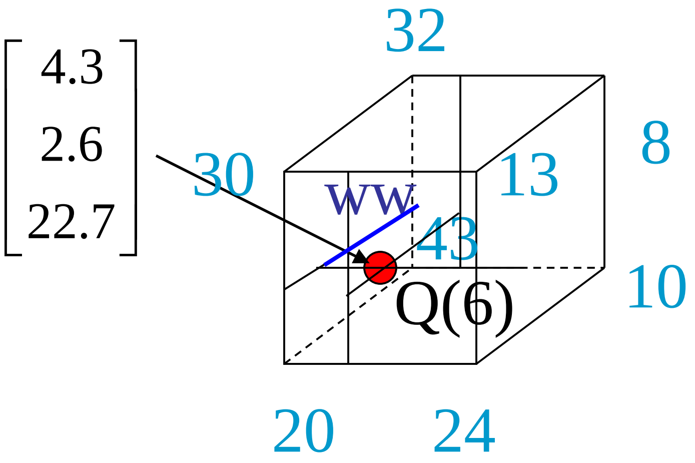
$$\begin{bmatrix} 4.3 \\ 2.6 \\ 22.7 \end{bmatrix}$$



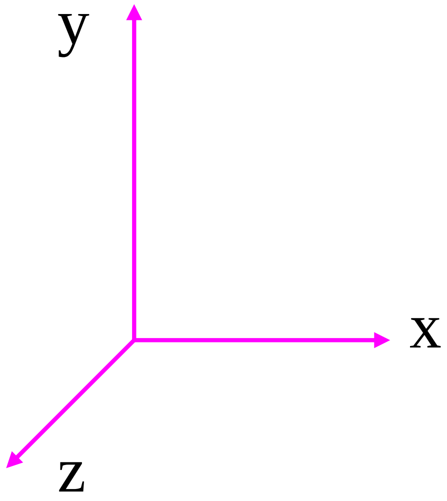
Orthographic (parallel) projection:
3rd linear interpolation: along z



$$
\begin{aligned}
P(uu,vv,ww) &= ww\cdot Val(P(uu,vv,z0+1)+(1-ww)\cdot Val(uu,vv,z0)\\
&= 0.7\cdot 23.42+0.3\cdot 28.12=
\end{aligned}
$$

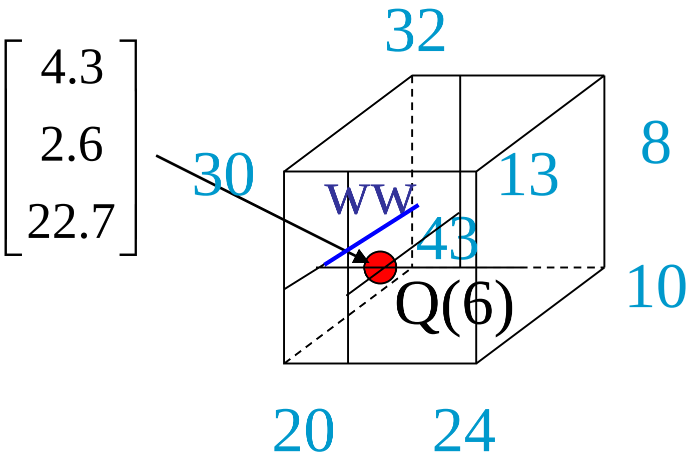


Orthographic (parallel) projection:
3rd linear interpolation: along z



$$P(uu, vv, ww) = ww \cdot Val(P(uu, vv, z0+1) + (1-ww) \cdot Val(uu, vv, z0)$$
$$= 0.7 \cdot 23.42 + 0.3 \cdot 28.12 = 24.83$$

Val(Q(6))


4.3
2.6
22.7
32
30
13
8
ww
43
10
Q(6)
20
24