# Fundamentals of Scientific Visualization and Computer Graphics Techniques

# What is Scientific Visualization?

- Transformation of data or information into pictures (visual outputs)
- Note this does not necessarily imply the use of computers
- Classical visualization used hand-drawn figures and illustrations (2D means for visualization)
- Modern visualization is primarily 3D (digital images for 3D visualization)
- In both cases, the ultimate goal is to understand important insights about the data through *visual means*
- We really don't care *how* we get the picture in visualization – *what* picture we get is most important
- The technical ways to arrive at visual outputs are mainly depending on computer graphics techniques

# Why is Visualization Useful & Important?

- Which is more helpful: A or B?

16 million 3D points:

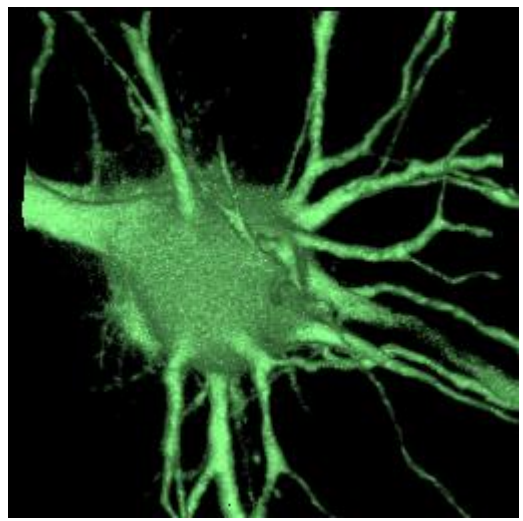
5, 34, 22, 56, 114, …

A

B

# Visualization Terminology

- Different sub-fields of visualization
- *Scientific visualization*
  - discipline of computer science
  - visualization of scientific and engineering data-sets
- Scientific visualization touches on a number of areas:
  - data representations
  - data processing algorithms
  - visual representations
  - user interfaces



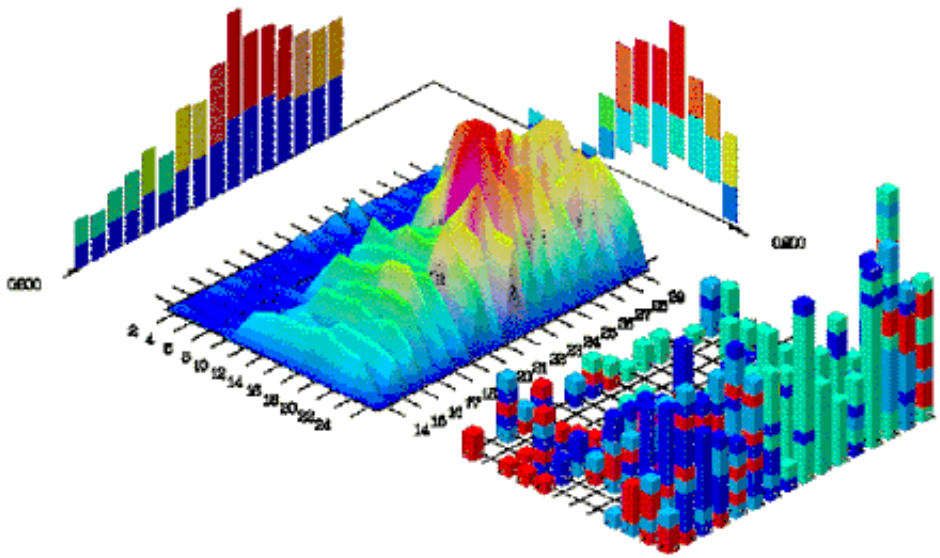
# Visualization Terminology

- *Data visualization* – includes data from other sources, such as financial, marketing, business
- Sometimes involves statistical analysis and other analysis techniques not employed in scientific visualization
- Can you think of an example of financial information we might want to visualize?
- So we might say that scientific visualization is a type or subset of data visualization
- We will be studying scientific visualization primarily, but look at more general data visualization occasionally

# Visualization Terminology

- *Information visualization* – abstract data sources, like WWW pages and databases
- No natural mapping to spatial domain (2D, 3D or n-D)
- How/what would we visualize in Amazon.com's book database?

- Visual analysis of customer call center performance at British Telecommunications:

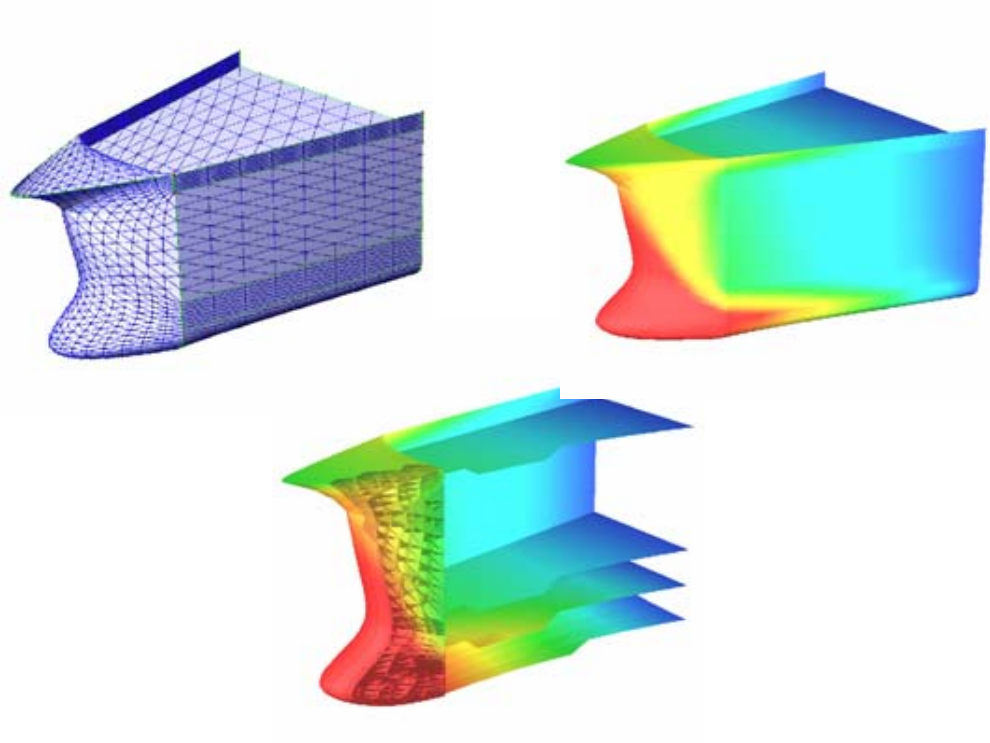


# Motivations of Visualization

- Make sense of huge data-sets
  - NYSE makes hundreds of millions of transactions per day
  - RHIC at BNL produces terabytes ($2^{40}$) of data with each experiment
- Uncover insights hidden in the data
- Extract important features and meaningful knowledge of the data to assist in the decision-making process
- But why use visual means?

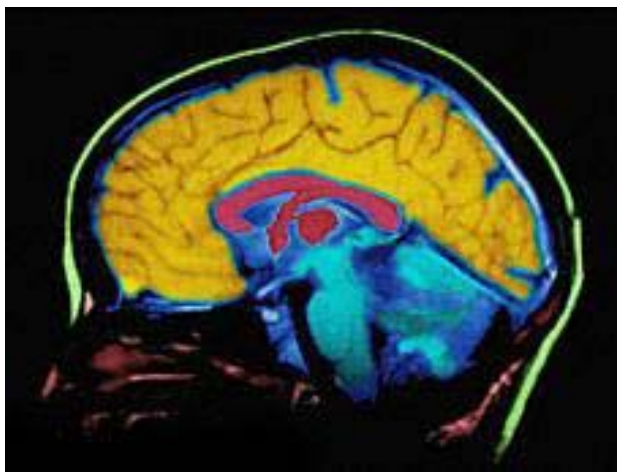
# Motivations of Visualization

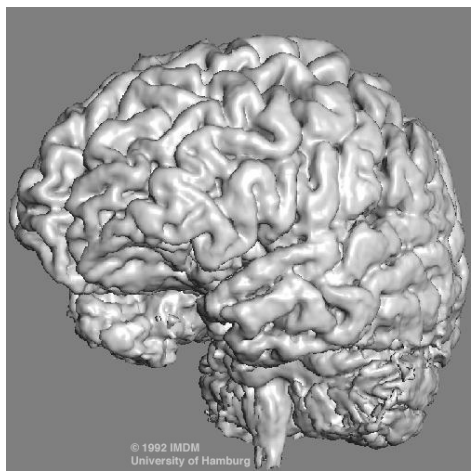
- Reduce time and save money
- Digital prototyping
  - Design model in virtual reality (VR)
  - Test model in VR
  - Refine and re-test
- Flight simulation
  - Why?
- Virtual training
  - Why?



# Examples of Visualization

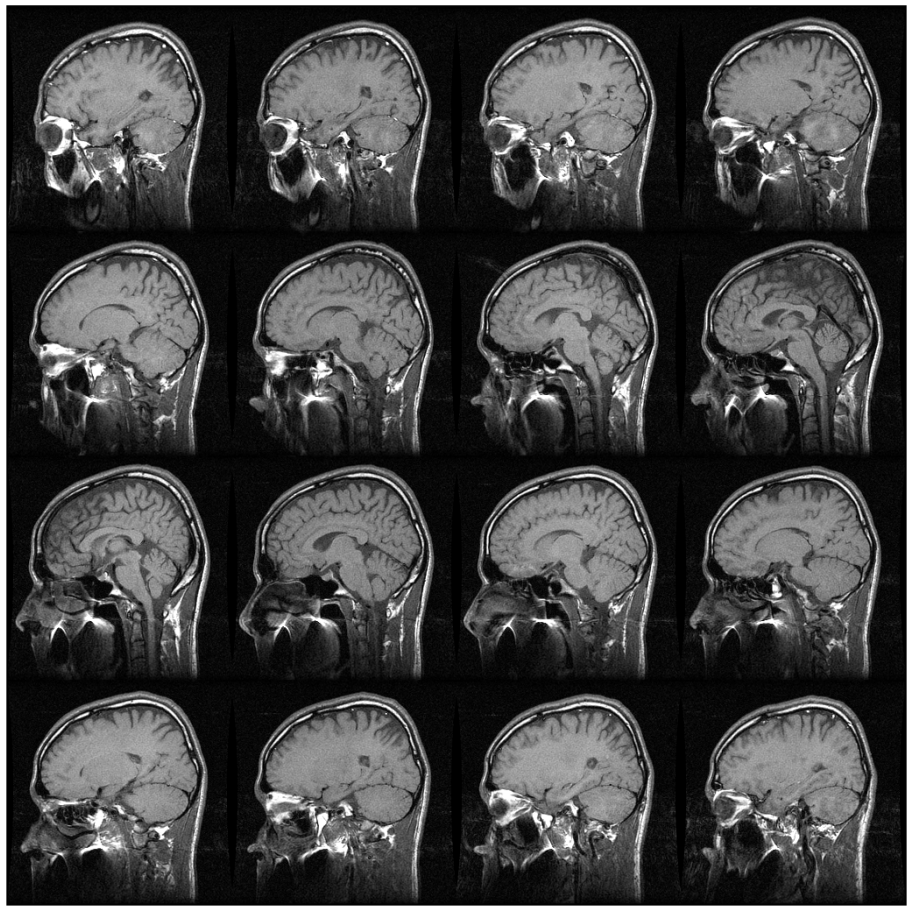
- Medical imaging
- X-ray Computed Tomography (CT)
  - pronounced as both “cat” or “see-tee”
- Magnetic Resonance Imaging (MRI)
  - uses very powerful magnetic fields





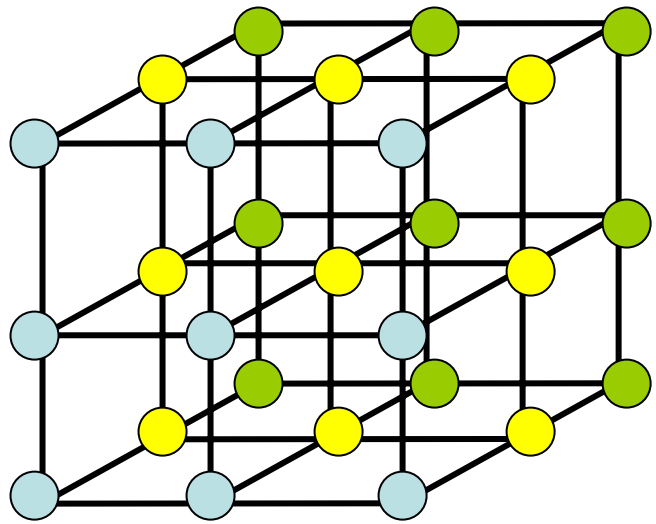
# Examples of Visualization

- CT and MRI produce *slice planes*
- Cross-sections of the patient
- Slices are combined to produce a *volumetric representation*
- But CT and MRI machines just output numbers – where do the gray values come from?

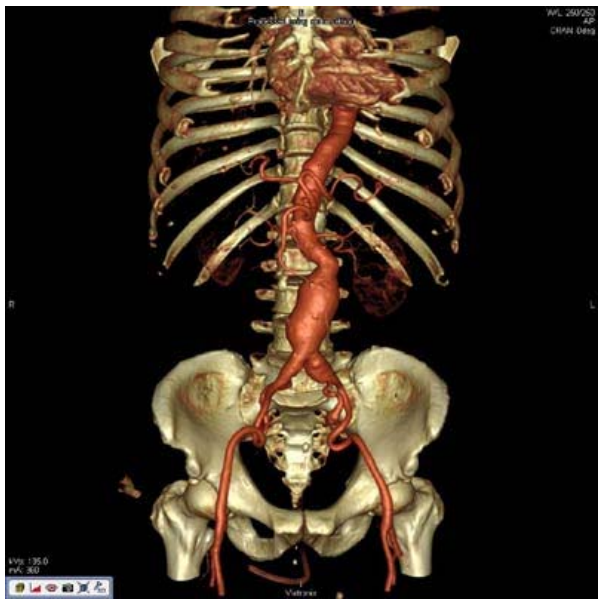


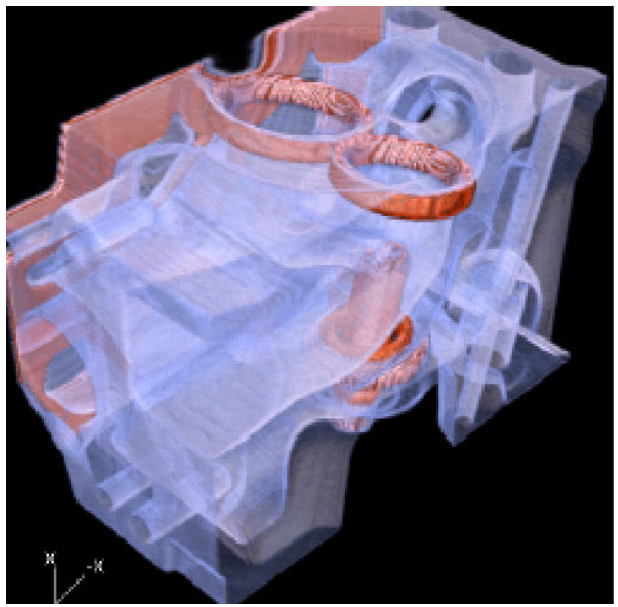
# Examples of Visualization

- A volumetric data-set is a 3D regular grid, or *3D raster*, of numbers that we map to a gray scale or gray level
- An 8-bit volume could represent 256 values [0,255]
- Human visual system naturally groups like-colored points, or *voxels*, into regions

# Volume Visualization Examples



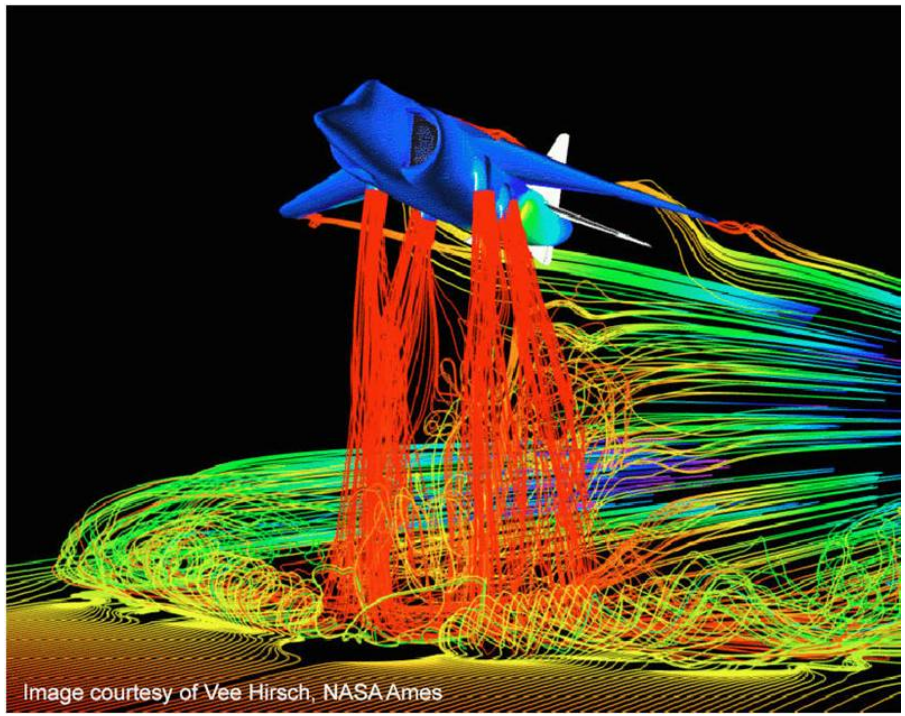


# Examples of Visualization

- Terrain visualization
- What are some applications?
- Satellite imaging
- x, y, elevation
- Terrain texture (photographs)
- Cloud cover
- What are some others?

# Examples of Visualization

- Scientific simulations
- Visualize the results of very sophisticated super-computer simulations
- Computational fluid dynamics example:
- What quantities are being visualized?
- Why bother?

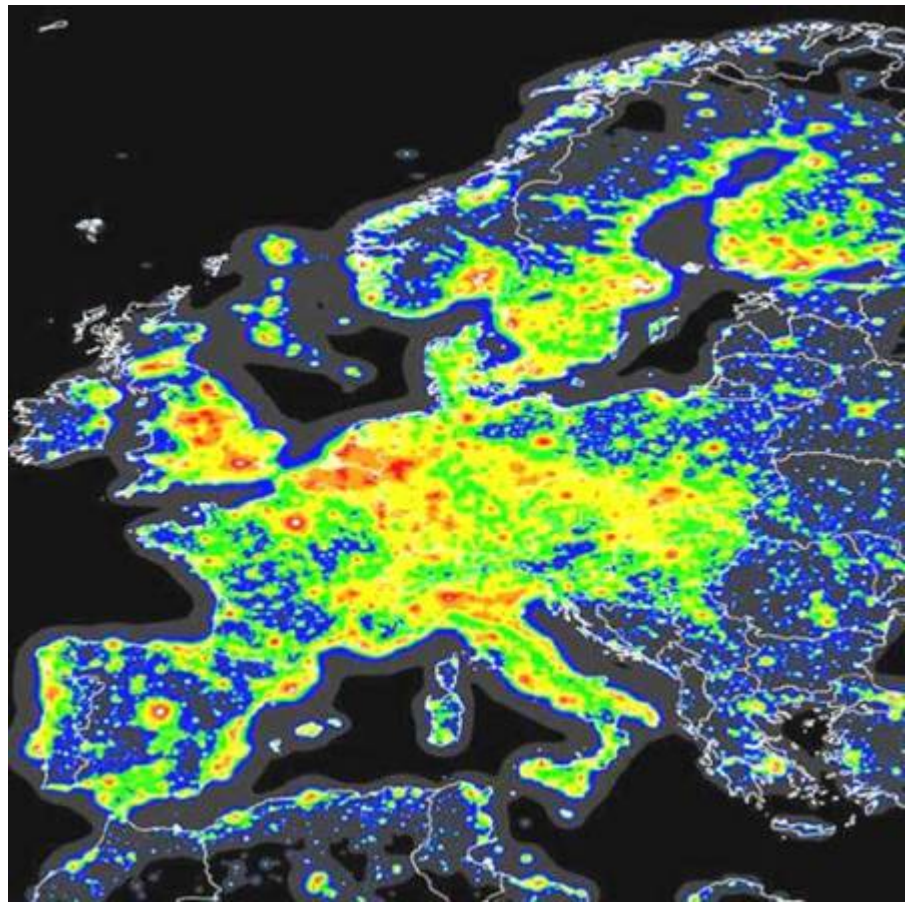


# Examples of Visualization

- Virtual archaeology
- How is a mummy examined?  A fossilized dinosaur egg?
- What's wrong with those methods?


# Examples of Visualization

- Map of artificial sky brightness over Europe. This is an effective tool for measuring “light pollution:” brightness of lights on ground affect ability to see starlight. Black: many stars visible. Red: few stars visible.



# Image Processing, Computer Graphics & Visualization

- *Image processing*
  - study and analysis of 2D pictures or images
- *Computer graphics*
  - process of creating images with a computer
- *Visualization*
  - process of exploring, transforming and viewing data as images
- What's in common?
- How do these three fields overlap?

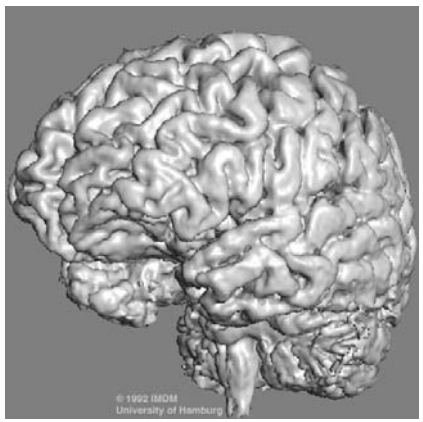
# Image Processing, Computer Graphics & Visualization

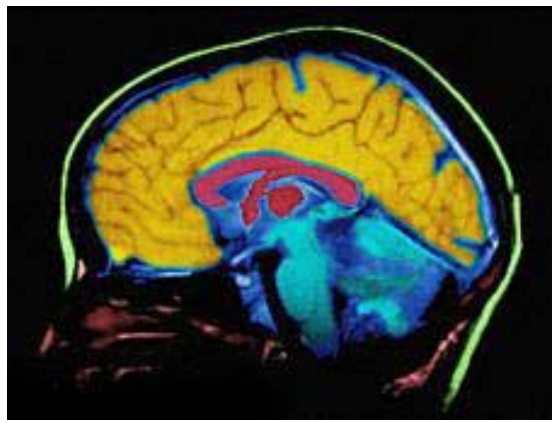
- Computer graphics outputs an image
- Visualization may employ graphics to generate images
- Visualization may employ image processing to study images
- Visualization
  - usually works with 3D or n-D data, for n >= 3
  - employs data transformation to enhance meaning of the data
  - is usually interactive and required human intervention

# The Visualization Process

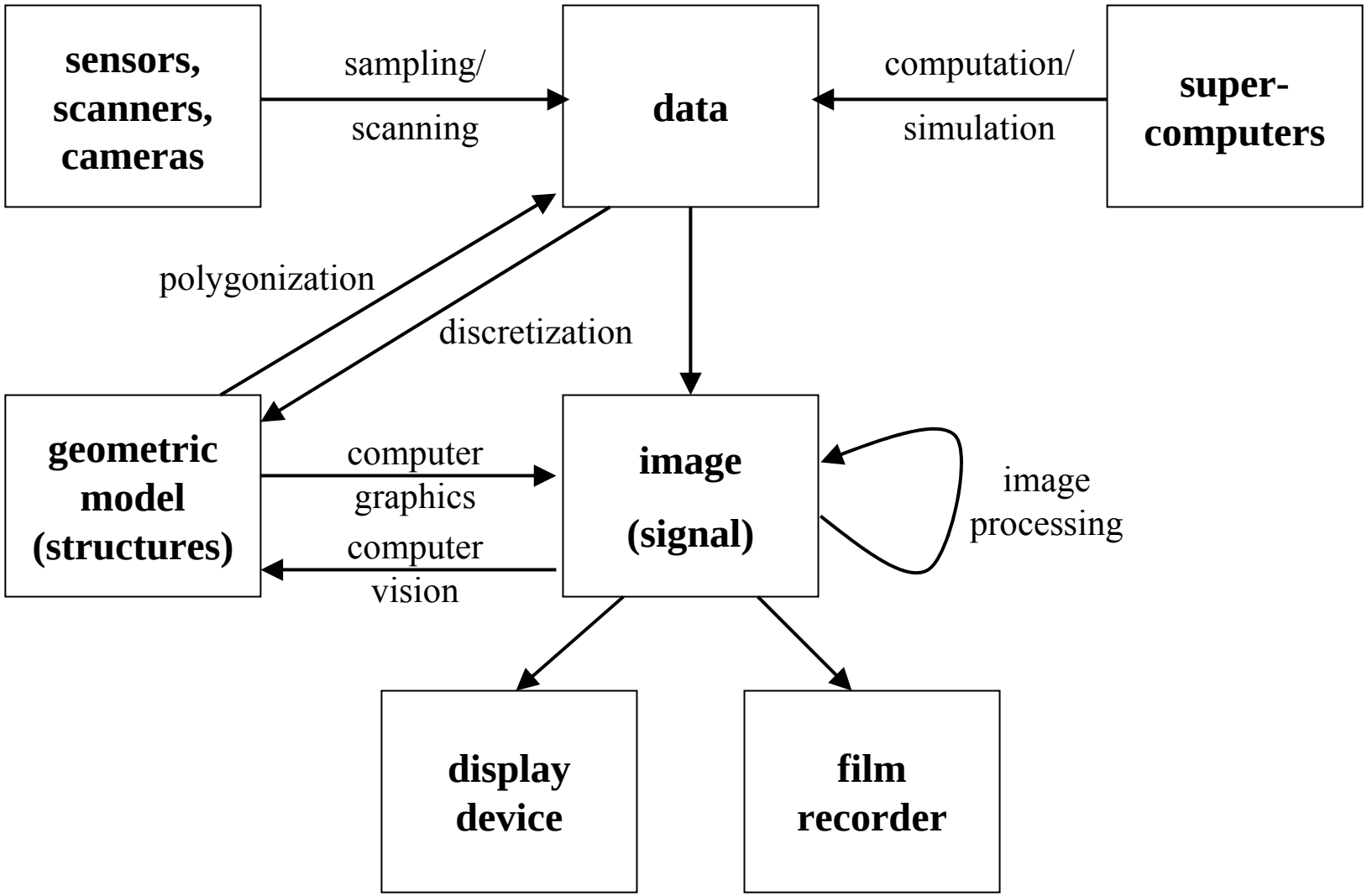
1. Data acquisition
2. Data transformation
3. Data mapping (e.g., to shapes & color)
4. Display (via computer graphics)

- Steps 2-4 are repeated as necessary to generate multiple visualizations



# An Alternate Visualization Pipeline



# Other Important Issues

- Accuracy
  - Safety, time, money, efficiency
- Ethics
  - What are some ethical concerns in visualization? (consider medical visualization)
- Psychological
  - Human visual/perception system
  - What makes an effective visualization?

# Current Trends in Visualization

- Scanning technologies (esp. MRI, CT) continue to improve
- New applications (virtual medical exam) for an aging population
- Multidimensional data (vector fields)
- Information visualization is very hot
- Homeland security generating new applications (threat planning in NYC)

# Summary & Questions

- Visualization overview
- Important visualization terminology
- Applications of visualization
- Connection with other fields
- What's the difference between computer graphics and visualization?

# Traditional Visualization: Historical Perspectives

# Traditional Visualization

- What are the origins of visualization?
- What are some of the troubles inherent in trying to visualize data?
- What makes a visual representation of some data faithful, helpful, accurate, etc?
- Surprisingly, we can learn a lot about 3D computer-driven visualization by looking at early attempts at effective 2D hand-drawn visualizations

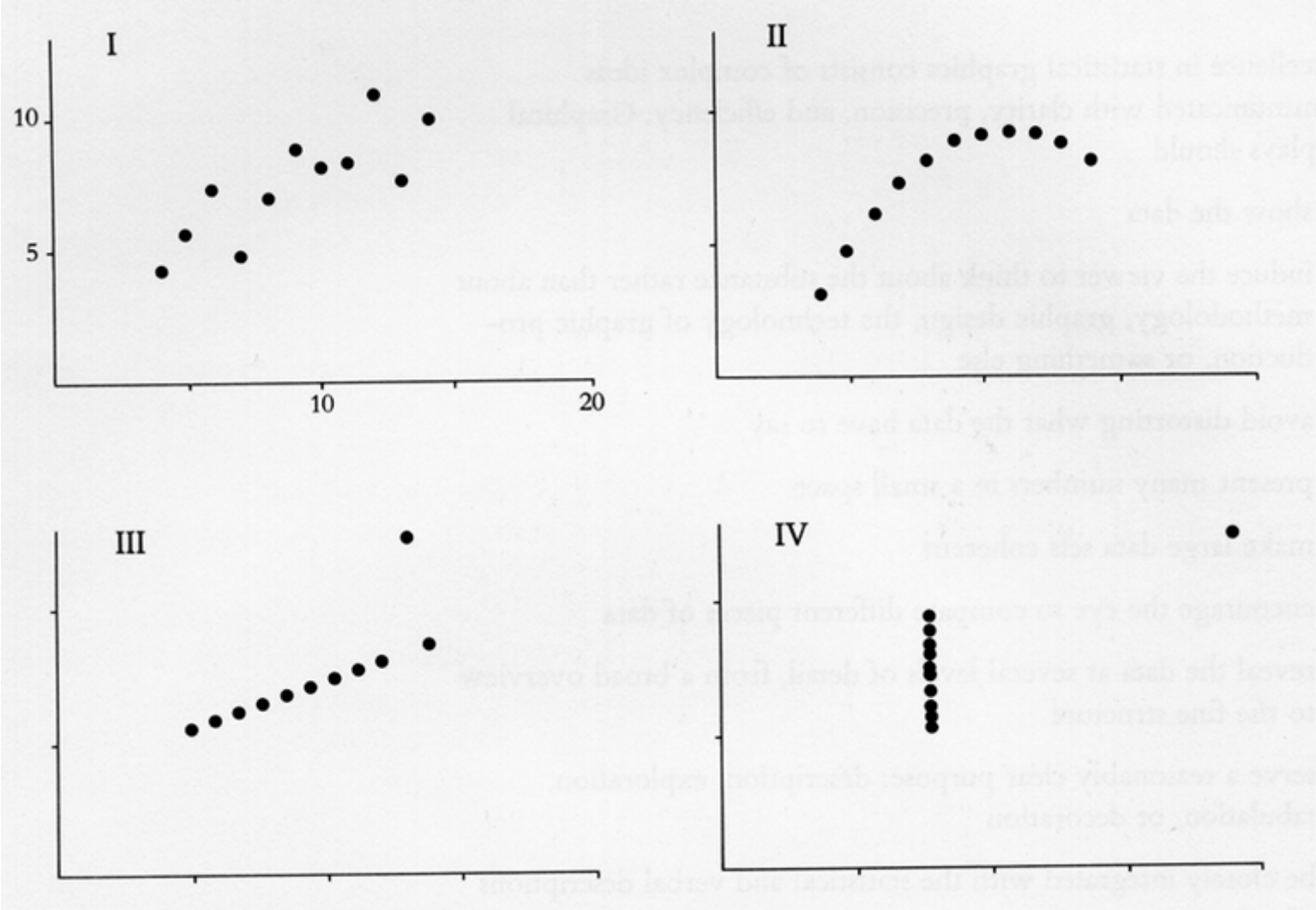
# Graphical Display

- Fundamental question: why bother with visualization? What do we gain? Why aren't words and numbers enough?
- Graphics (i.e., pictures) can be more precise and revealing than numerical display

| I | | II | | III | | IV | |
|---|---|---|---|---|---|---|---|
| X | Y | X | Y | X | Y | X | Y |
| 10.0 | 8.04 | 10.0 | 9.14 | 10.0 | 7.46 | 8.0 | 6.58 |
| 8.0 | 6.95 | 8.0 | 8.14 | 8.0 | 6.77 | 8.0 | 5.76 |
| 13.0 | 7.58 | 13.0 | 8.74 | 13.0 | 12.74 | 8.0 | 7.71 |
| 9.0 | 8.81 | 9.0 | 8.77 | 9.0 | 7.11 | 8.0 | 8.84 |
| 11.0 | 8.33 | 11.0 | 9.26 | 11.0 | 7.81 | 8.0 | 8.47 |
| 14.0 | 9.96 | 14.0 | 8.10 | 14.0 | 8.84 | 8.0 | 7.04 |
| 6.0 | 7.24 | 6.0 | 6.13 | 6.0 | 6.08 | 8.0 | 5.25 |
| 4.0 | 4.26 | 4.0 | 3.10 | 4.0 | 5.39 | 19.0 | 12.50 |
| 12.0 | 10.84 | 12.0 | 9.13 | 12.0 | 8.15 | 8.0 | 5.56 |
| 7.0 | 4.82 | 7.0 | 7.26 | 7.0 | 6.42 | 8.0 | 7.91 |
| 5.0 | 5.68 | 5.0 | 4.74 | 5.0 | 5.73 | 8.0 | 6.89 |

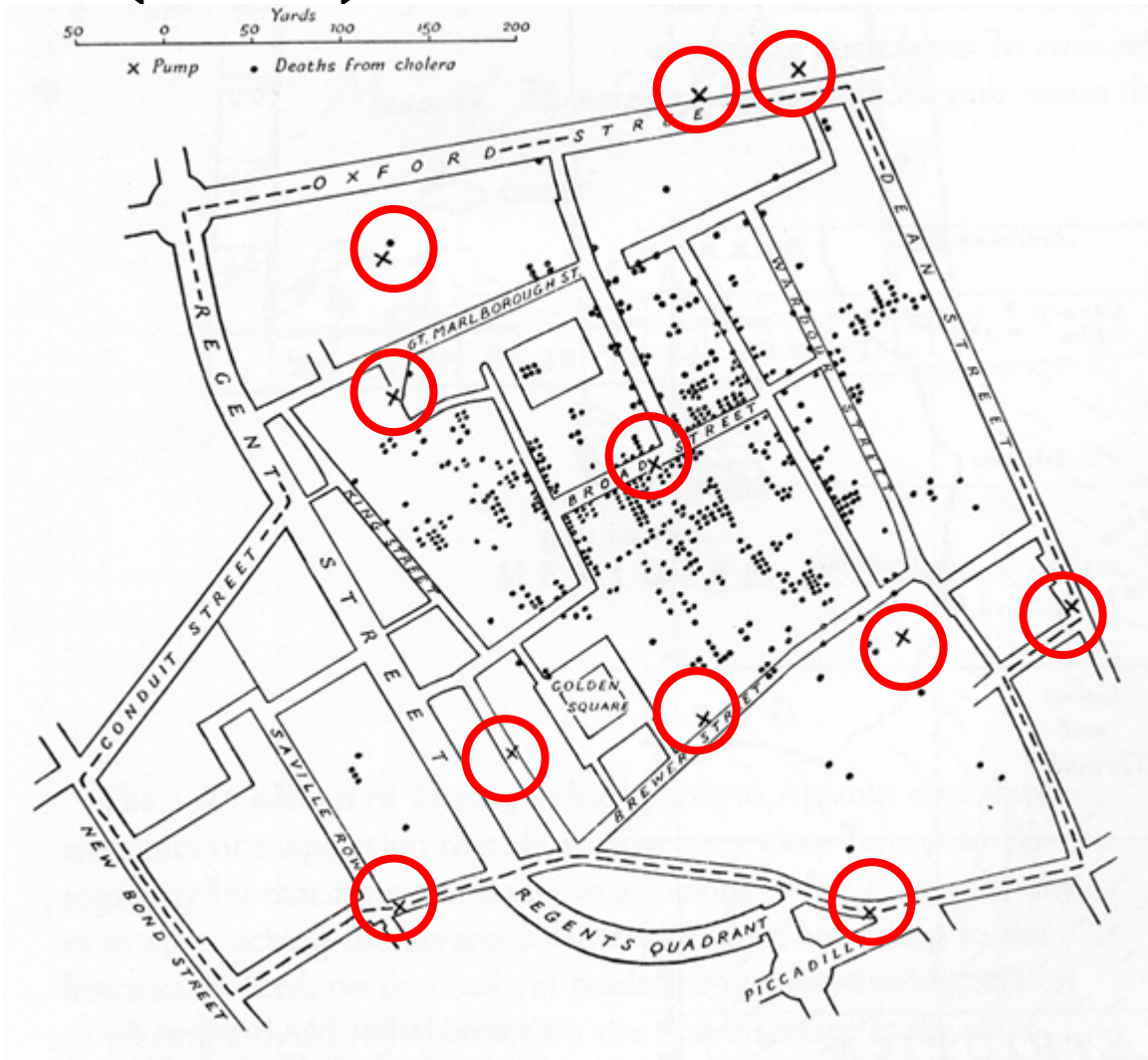
$N = 11$
mean of X's $= 9.0$
mean of Y's $= 7.5$
equation of regression line: $Y = 3 + 0.5X$
standard error of estimate of slope $= 0.118$
$t = 4.24$
sum of squares $X - \overline{X} = 110.0$
regression sum of squares $= 27.50$
residual sum of squares of $Y = 13.75$
correlation coefficient $= .82$
$r^2 = .67$

# Surprise Hidden in the Data



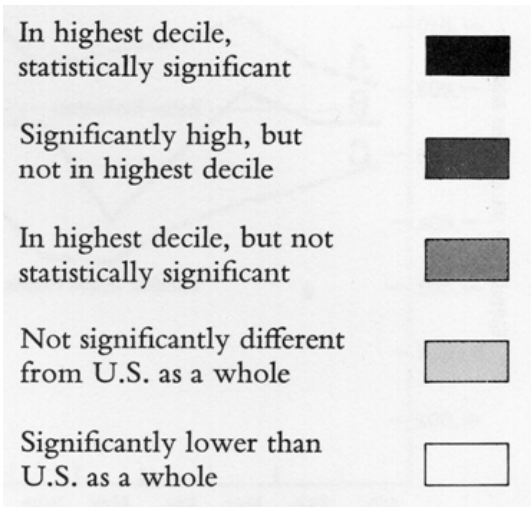
# Dr. John Snow's Cholera Map of London (1854)

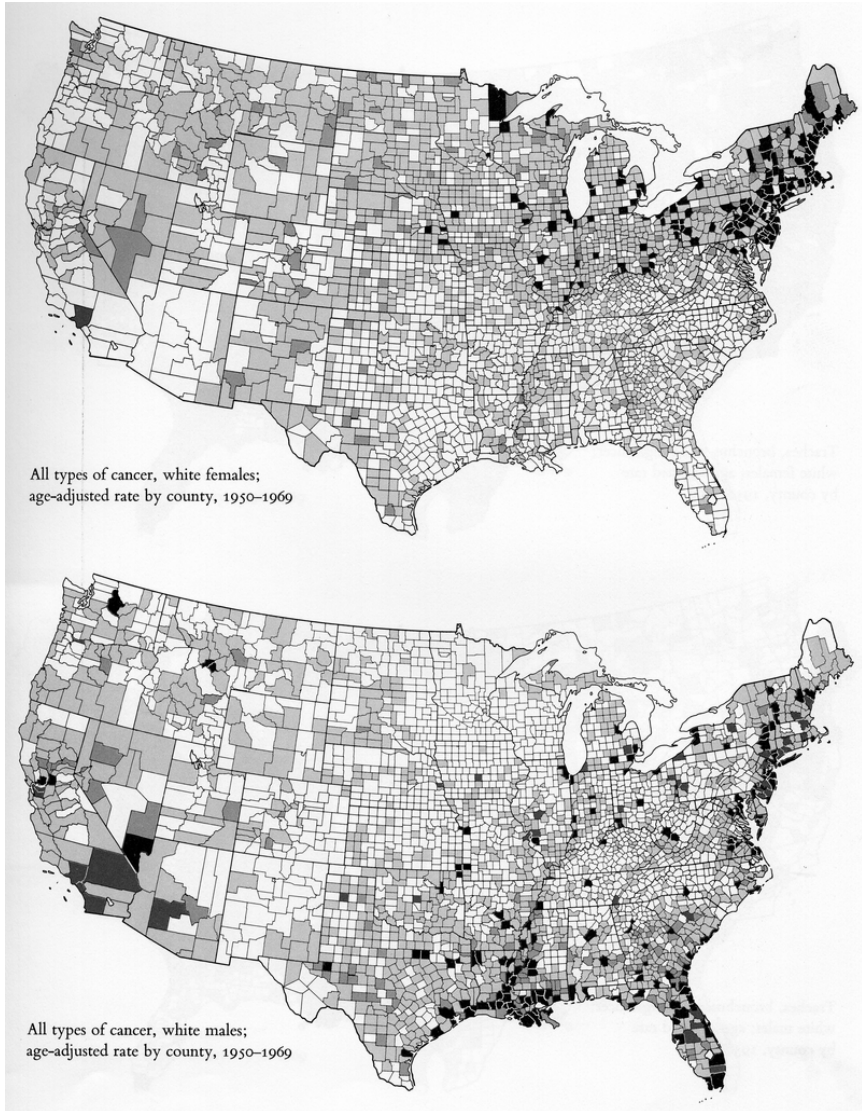
- Dot indicates Cholera death
- X indicates water pump (circled)
- What does this visualization tell us?



# Graphical Display – Large Datasets

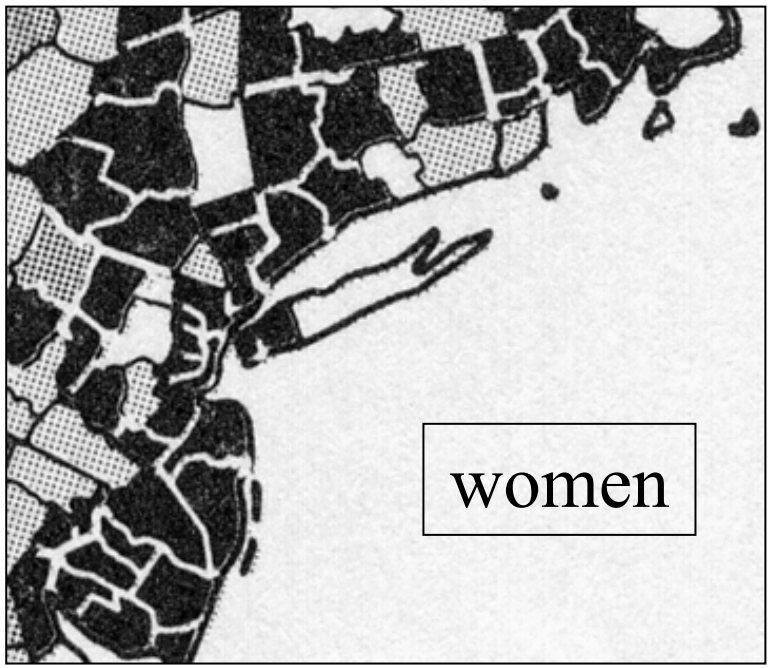
- Can capture a large amount of information in a very small space
- Total cancer deaths, 1950-1969 (top: white women; bottom: white men)

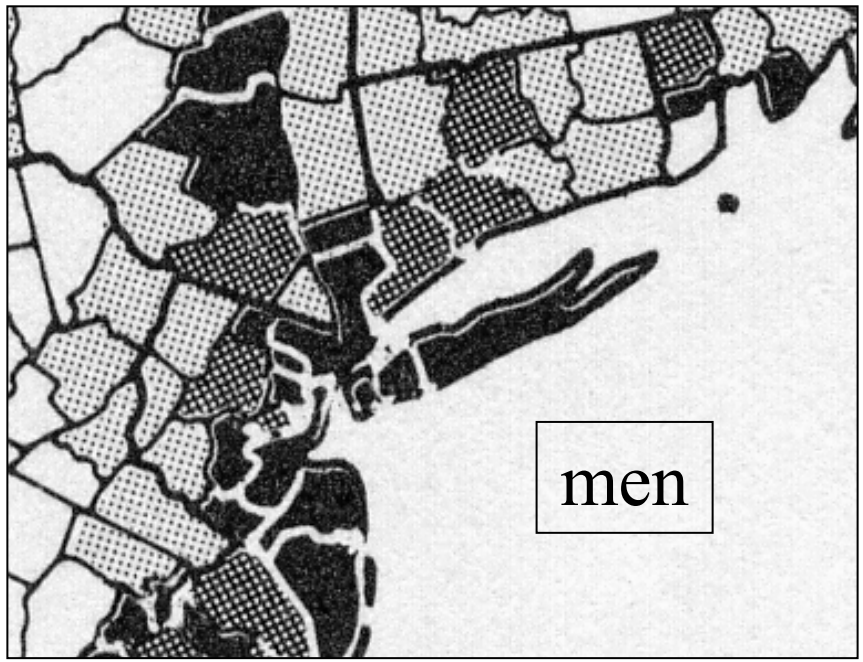


# Graphical Display – Large Datasets

- The local situation

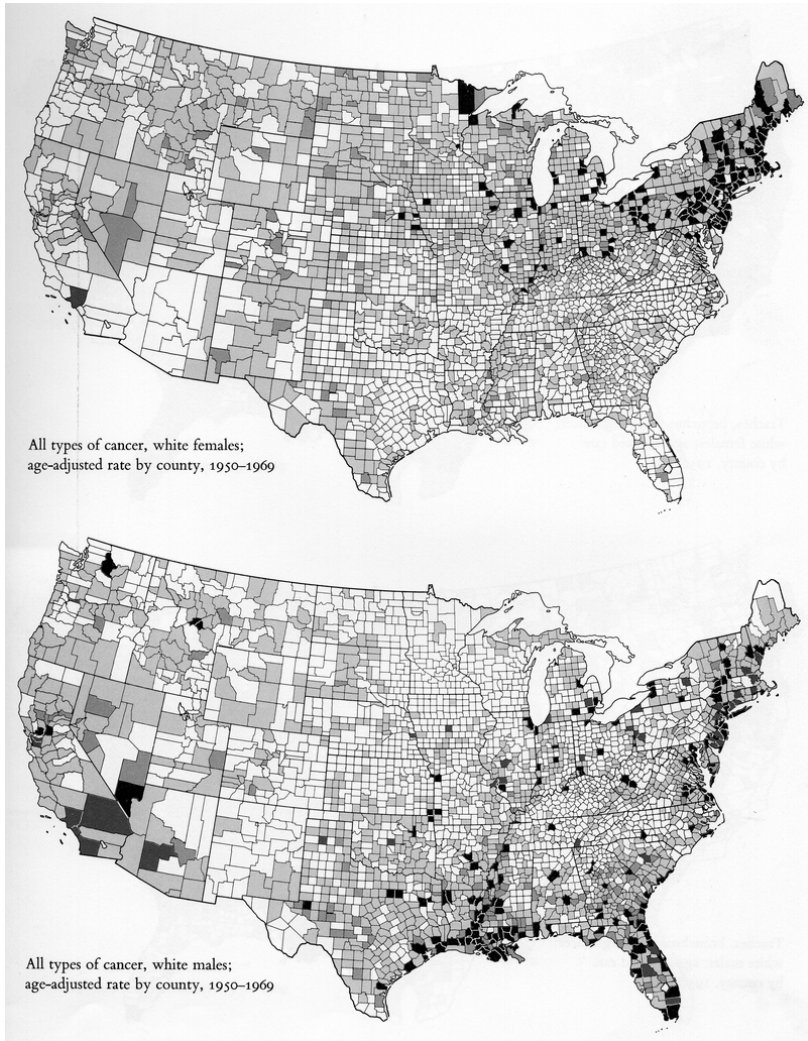



- Hopefully things have improved!

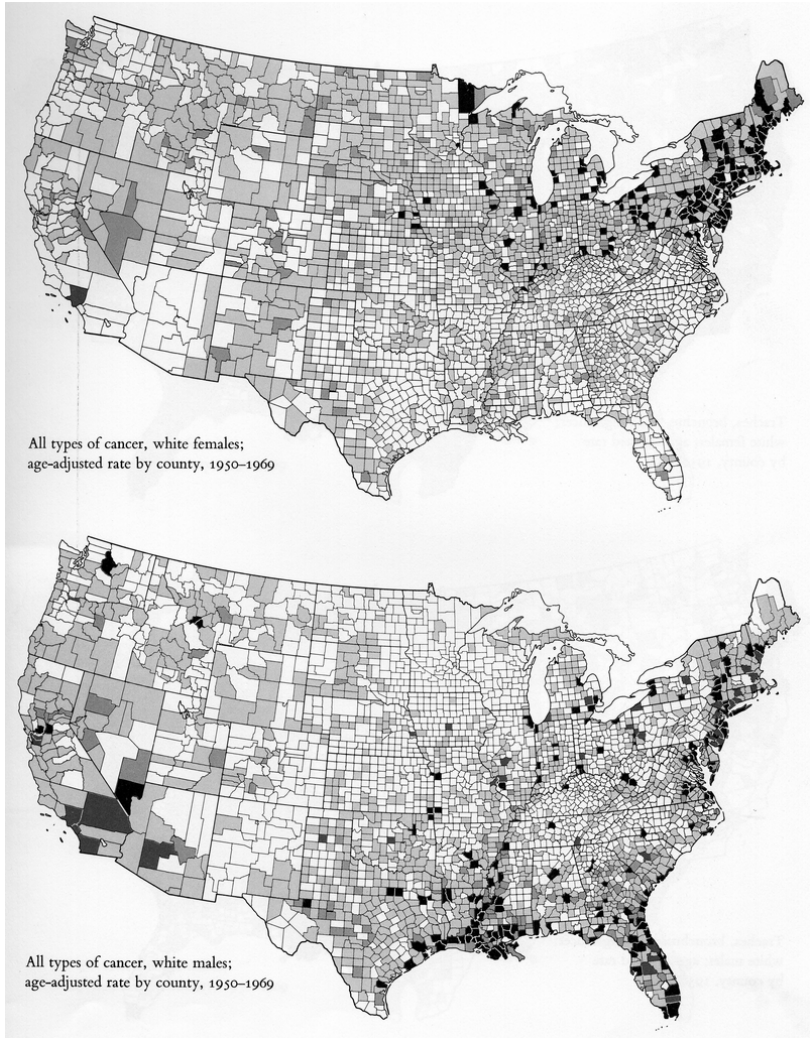
# Graphical Display – Large Datasets

- Important questions:
  - Where are the highest death rates?
  - Lowest rates?
  - Rate for men vs. women
  - Any anomalies?
  - What to do with the knowledge?



# Graphical Display – Large Datasets

- Do you see any possible problems with visualizing the data in this manner? (hint: consider land area)
- Focus is incorrectly drawn to land area rather than number of people actually living in county
- A large county may have only a few people living in it

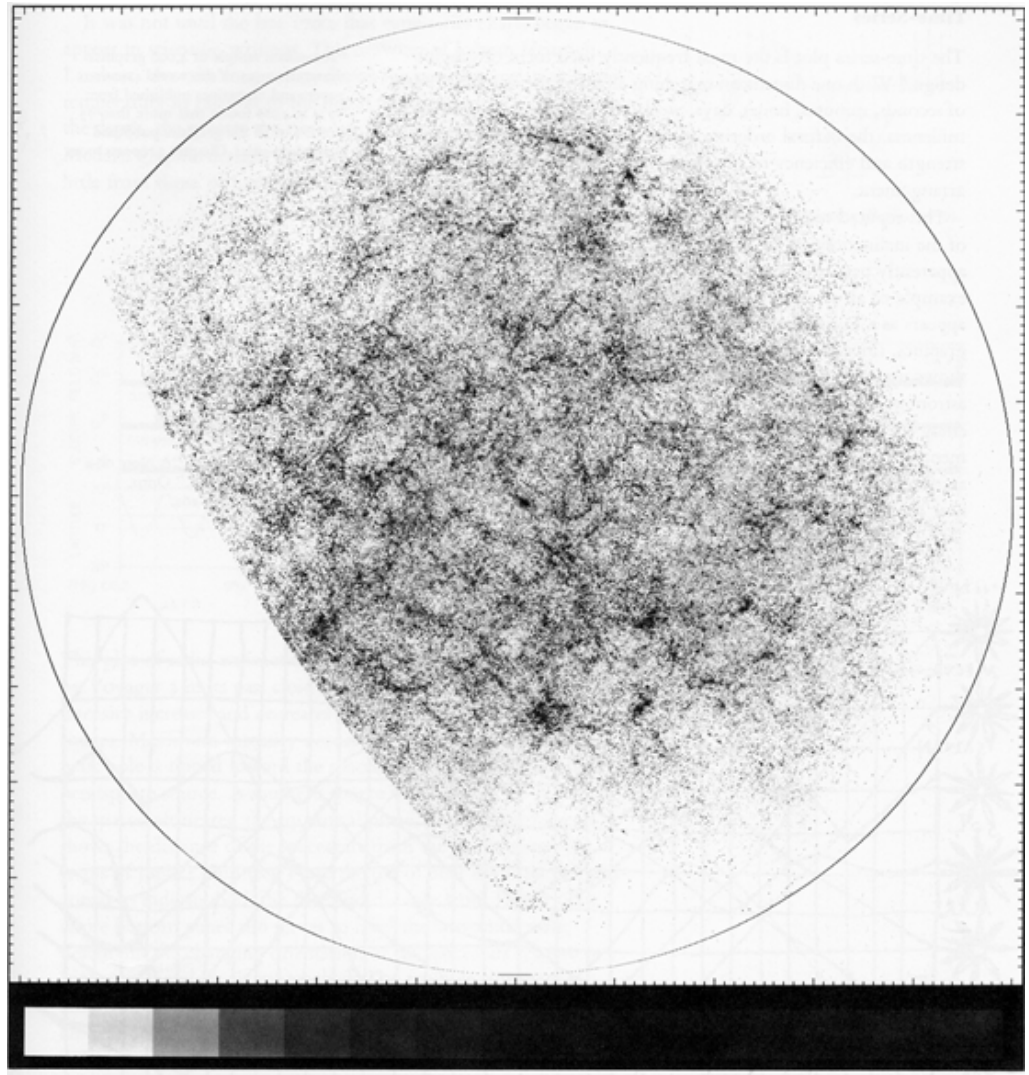


# Traditional Visualization ("Information Graphics")

- The fundamental goal of visualization is to reveal the substance of the data – i.e., what we can learn from the raw data and what we should do with our knowledge
- Our concern is with the data and not so much the techniques, algorithms or methodologies used to draw the image
- We also have to make reasonable assumptions that the data itself is not corrupt or skewed in some manner
- Visualization has moved from using hand-drawn illustration to Computer-Generated Imagery (CGI)

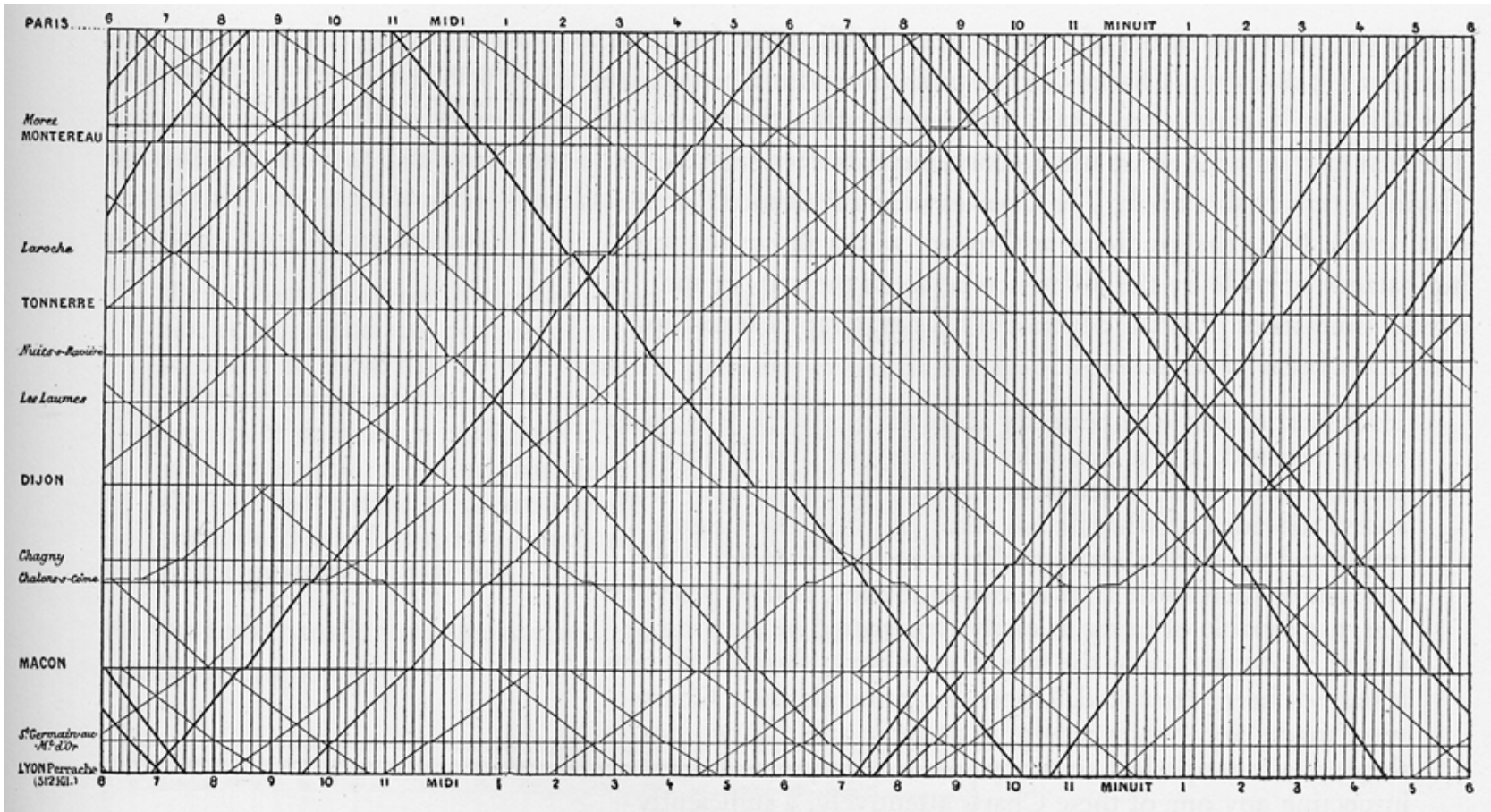
# Graphical Display – Large Datasets

- Galaxy map
- Each dot represents a collection of galaxies
- 1.3 million total
- 1024x2022 grid
- Can you see any structure in the data?
- What might these observations tell us?



# Time-Series Display

- Paris-Lyon train schedule from 1880s

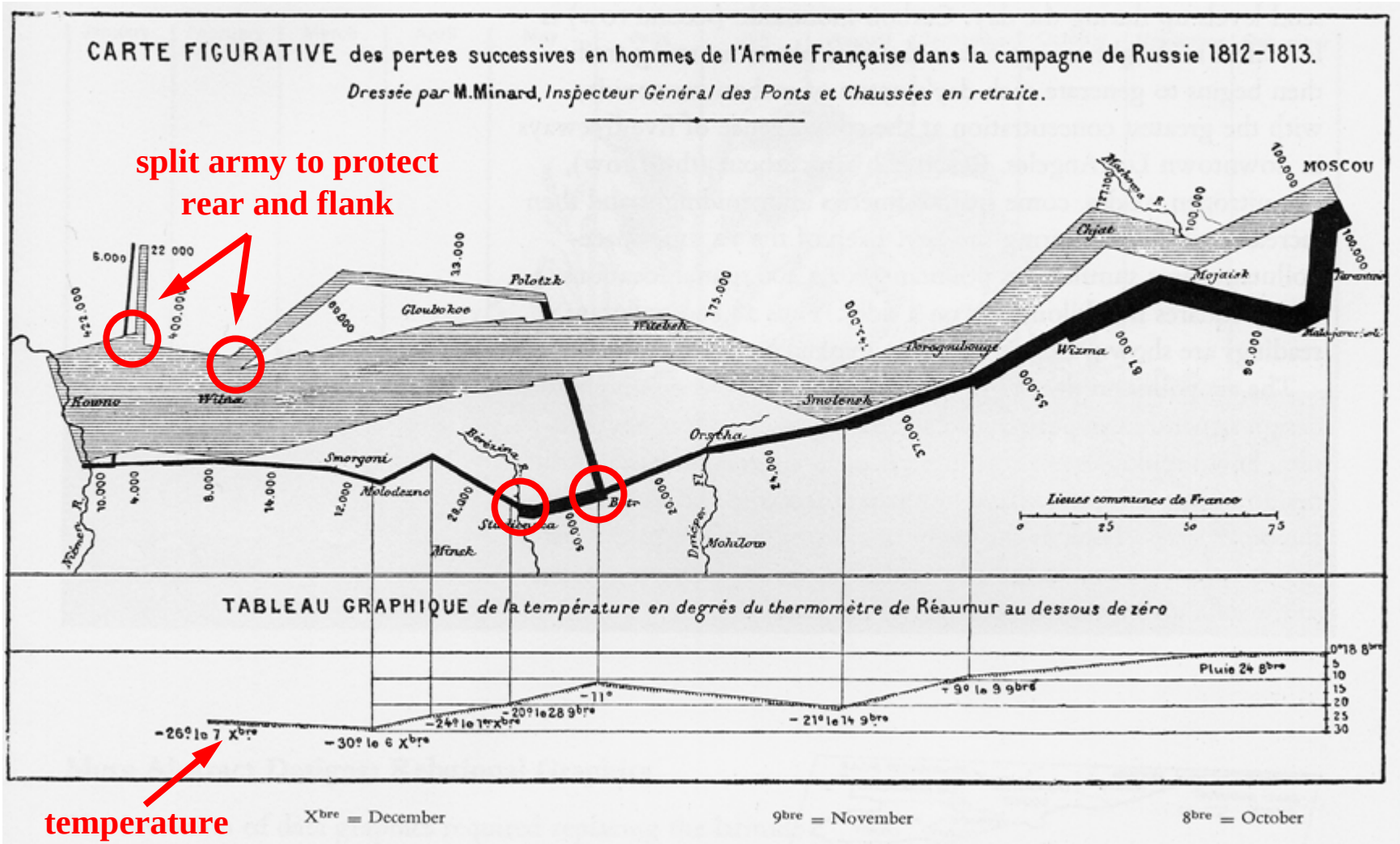


# Compare with this way…

- What are the advantages and disadvantages of each method?
- What do we learn about train routes from each?
- What does each visualization tell us that the other doesn't or can't?
- Consider issues like these in developing your own visualization algorithms and systems

| To Long Island<br>Monday to Friday Except Holidays | | | | |
|---|---|---|---|---|
| | Leave | | | Arrive |
| Notes | Penn Station | Flatbush Avenue | Jamaica | Stony Brook |
| | J1:44 am | J1:44 am | 2:11 am | 3:29 am |
| | 5:47 am | J5:51 am | 6:12 am | AT7:46 am |
| | J7:49 am | J7:50 am | 8:15 am | A9:28 am |
| | 9:15 am | J9:09 am | 9:37 am | E11:00 am |
| | 10:41 am | J10:37 am | 11:04 am | T12:30 pm |
| | 12:15 pm | J12:10 pm | 12:37 pm | E2:00 pm |
| | 1:42 pm | J1:37 pm | 2:04 pm | T3:30 pm |
| | 2:52 pm | J2:33 pm | 3:13 pm | T4:42 pm |
| | J3:34 pm | J3:32 pm | 4:00 pm | 5:15 pm |
| Peak | 4:19 pm | J4:20 pm | 4:40 pm | 5:54 pm |
| Peak | 4:49 pm | J4:47 pm | 5:12 pm | 6:24 pm |
| Peak | J5:04 pm | J5:06 pm | 5:30 pm | 6:49 pm |
| Peak | J5:38 pm | J5:45 pm | 6:07 pm | 7:19 pm |
| Peak | 6:01 pm | ..... | ..... | E7:49 pm |
| Peak | ..... | J6:08 pm | 6:28 pm | 7:49 pm |
| Peak | J6:27 pm | J6:27 pm | 6:49 pm | 8:04 pm |
| Peak | 7:22 pm | J7:23 pm | 7:44 pm | T9:07 pm |
| | 8:42 pm | J8:42 pm | 9:03 pm | T10:30 pm |
| | 9:42 pm | J9:38 pm | 10:03 pm | T11:30 pm |
| Notes | Penn Station | Flatbush Avenue | Jamaica | Stony Brook |
| | 10:42 pm | J10:30 pm | 11:03 pm | T12:22 am |
| | 11:42 pm | J11:30 pm | 12:03 am | T1:29 am |

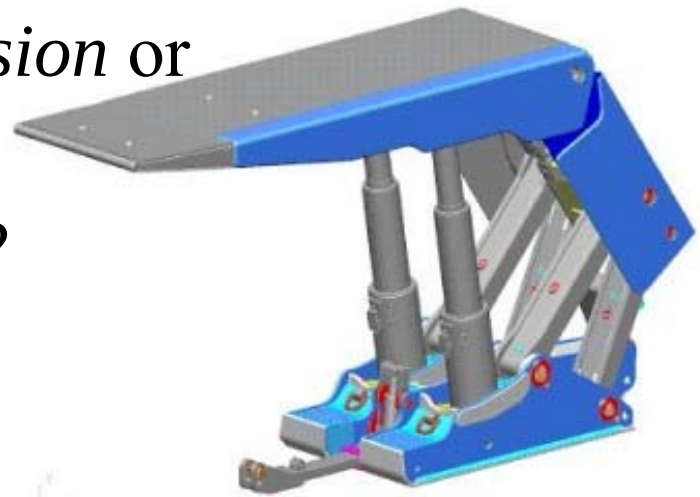
# Visualization in Narrative Form



# Computer Graphics vs. Scientific Visualization

# What is Computer Graphics?

- Process of generating images using computers
- This is called *rendering (computer graphics was traditionally considered as a rendering method)*
- A rendering algorithm converts a geometric model into a picture
- This process is called *scan conversion* or *rasterization*
- How does visualization fit in here?



# Computer Graphics

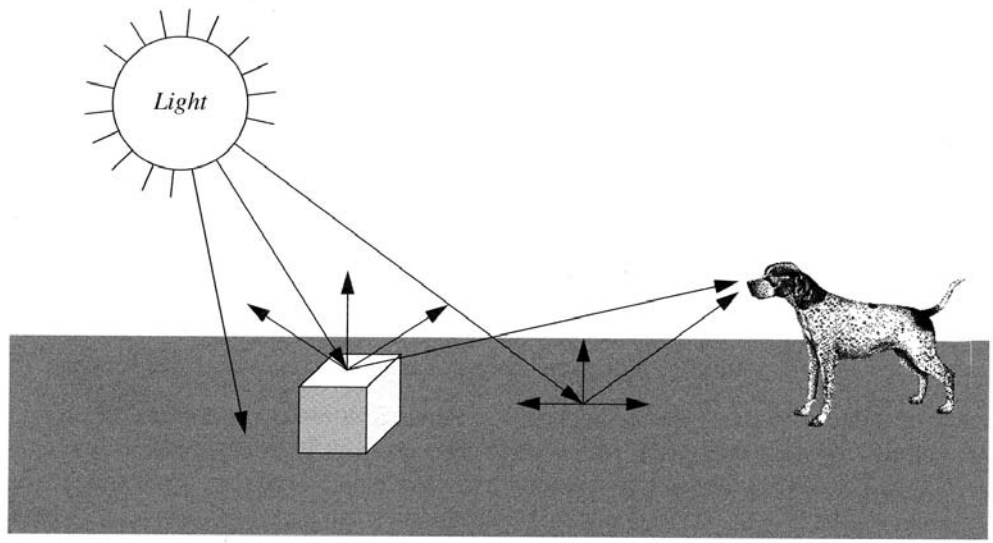
- Computer graphics consists of :
  1. Modeling (representations)
  2. Rendering (display)
  3. Interaction (user interfaces)
  4. Animation (combination of 1-3)
- Usually “computer graphics” refers to rendering

# Lights, Cameras and Objects

- How are we able to see things in the real world?
- What's the process that occurs?
- I'll get you started:
  1. Open eyes
  2. Photons from light source strike object
  3. Bounce off object and enter eye
  4. Brain interprets image you see

# Lights, Cameras and Objects

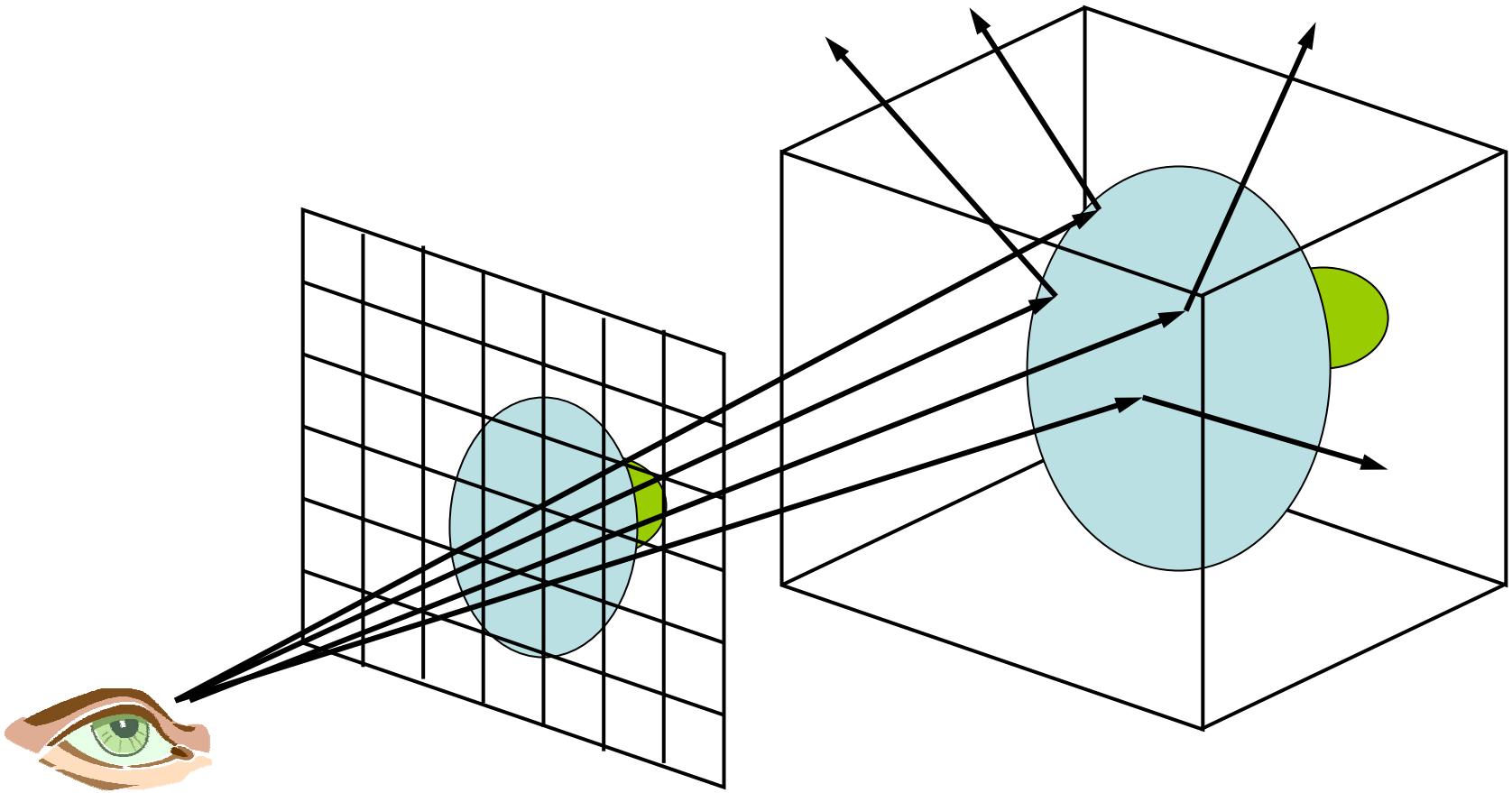
- Rays of light emitted by light source
- Some light strikes object we are viewing
  - Some light absorbed
  - Rest is reflected
  - Some reflected light enters our eyes



# Lights, Cameras and Objects

- How do we simulate light transport in a computer?
- Several ways
- *Ray-tracing* is one
- Start at eye and trace rays the scene
- If ray strikes object, bounces, hits light source → we see something at that pixel
- Most computer applications don't use it. Why?
- With many objects very computationally expensive

# Surface Ray-Tracing



# Rendering Processes: Image-Order and Object-Order

- Ray-tracing is an *image-order* process: operates on *per-pixel basis*
- Determine for each ray which objects and light sources ray intersects
- Stop when all pixels processed
- Once all rays are processed, final image is complete
- *Object-order* rendering algorithm determines for each object in scene how that object affects final image
- Stop when all objects processed

# Rendering Processes: Image-Order and Object-Order

- Image-order approach: start at upper left corner of picture and draw a dot of appropriate color
- Repeat for all *pixels* in a left-to-right, top-to-bottom manner
- Object-order approach: paint the sky, ground, trees, barn, etc. back-to-front order, or front-to-back
- Image-order: very strict order in which we place pigment
- Object-order: we jump around from one part of the regions to another

# Rendering Processes: Image-Order and Object-Order

- Advantages and disadvantages of each
- Ray-tracing can produce very realistic looking images, but is very computationally expensive
- Object-order algorithms more popular because hardware implementations of them exist
- Not as realistic as ray-tracing

# Surface Rendering

- We have considered interaction between light rays and object boundaries
- This is called *surface rendering* and is part of *surface graphics*
- Computations take place on boundaries of objects
- Surface graphics employs *surface rendering* to generate images of *surface' mathematical and geometric representations*

# Surface Graphics

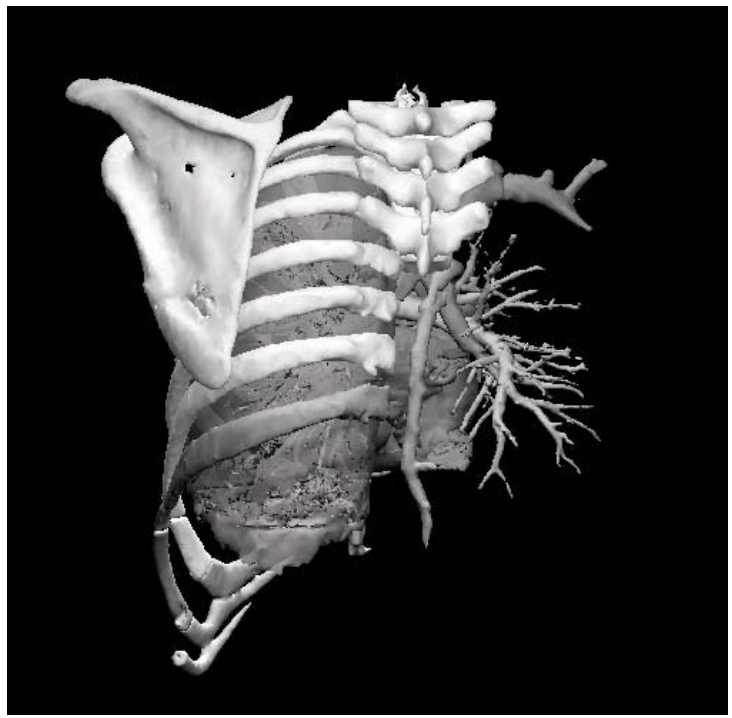
- Surface representations are good for objects that have *homogeneous* material distributions and/or are not *translucent* or *transparent*
- Such representations are good when only object boundaries are important
- Examples: furniture, mechanical objects, plant life
- Applications: video games, virtual reality, computer-aided design

# Surface Graphics – Pros and Cons

- Good: explicit distinction between inside and outside makes rendering calculations easy and efficient
- Good: hardware implementations are inexpensive
- Good: can use tricks like texture mapping to improve realism
- Bad: an approximation of reality
- Bad: does not let us peer into and through objects

# Surface Graphics

- Can you think of objects or phenomena for which this approach to rendering will fail?
- When is a surface representation not good enough?
- Would a surface representation suffice to represent the internal structure of the human body?

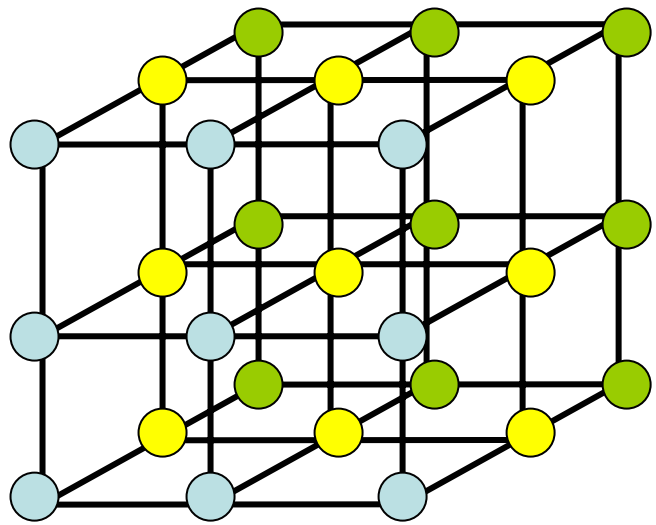


# Volume Graphics

- Surface graphics doesn't work so well for clouds, fog, gas, water, smoke and other *amorphous phenomena*
- "amorphous" = "without shape"
- Surface graphics won't help us if we want to explore objects with very complex internal structures
- Volume graphics provides a solution to these shortcomings of surface graphics
- Volume graphics includes *volume representations* and *volume rendering* algorithms to display such representations

# Volumetric Representations

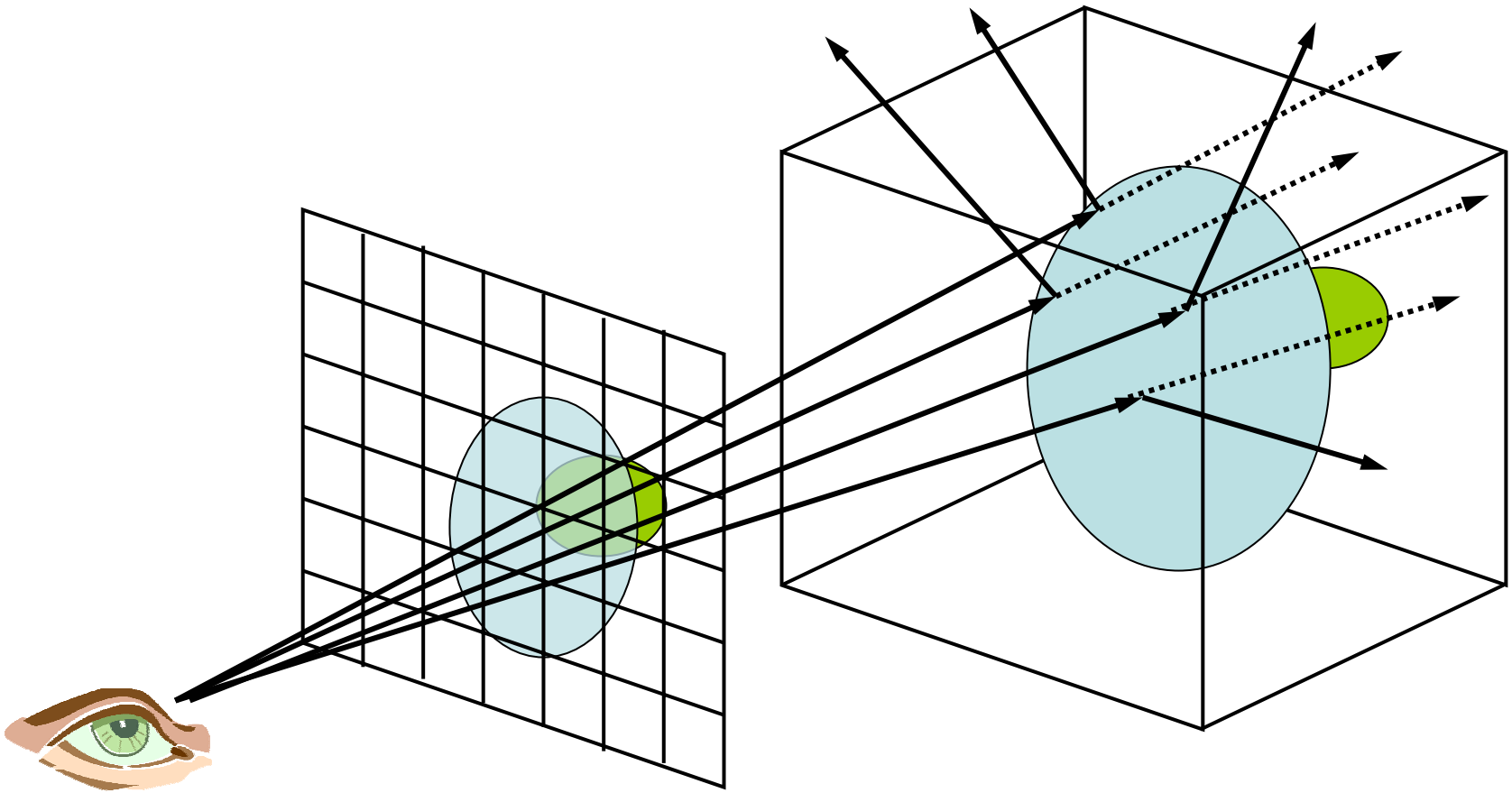
- A volumetric data-set is a 3D regular grid, or *3D raster*, of numbers that we map to a gray scale or gray level
- Where else have you heard the term *raster*?
- An 8-bit volume could represent 256 values [0, 255]
- Typically volumes are at least $200^3$ in size, usually larger
- How much storage is needed for an 8-bit, $256^3$ volume?



# Volume Graphics

- Volumetric objects have interiors that are important to the rendering process (what does that mean?)
- Interior affects final image
- Imagine that our rays now don't merely bounce off objects, but now can penetrate and pass through
- This is known as *volumetric ray-casting* and works in a similar manner to surface ray-tracing

# Volumetric Ray-Tracing

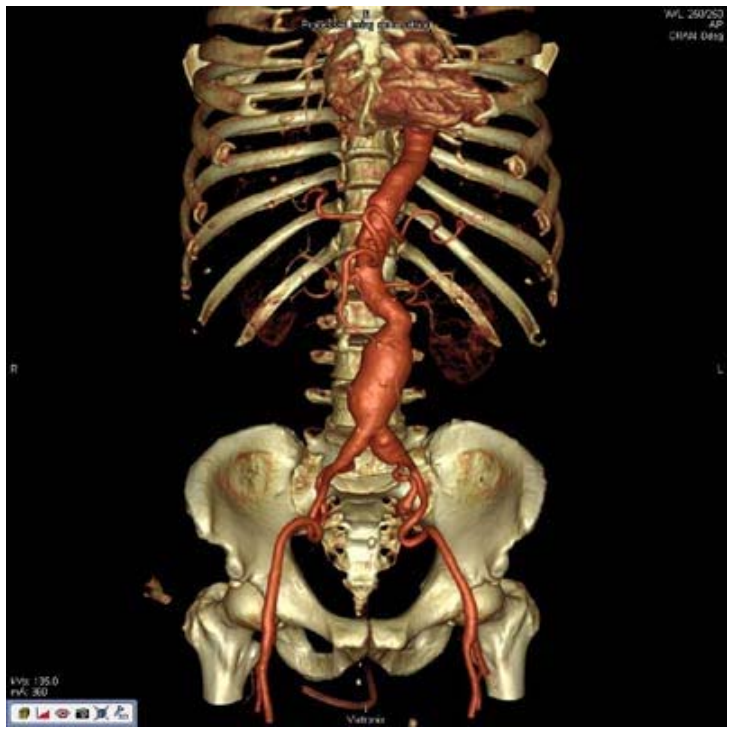


# Volume Rendering

- 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

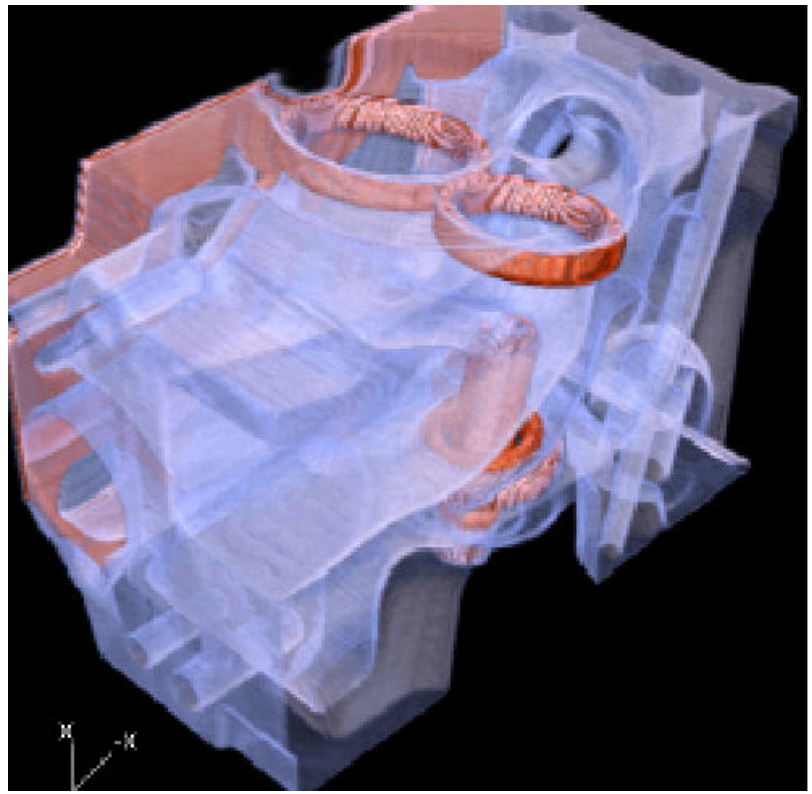
# Volume Rendering

- As the rays leave the data, they comprise a sheet of accumulated values
- These values represent the volumetric data *projected* onto a two-dimensional image (the screen)
- Special mapping functions convert the grayscale values from the CT/MRI into color



# Volume Rendering

- *Semi-transparent rendering*



# Volume Graphics

- Good: maintains a representation that is close to the underlying fully-3D object (but discrete)
- Good: can achieve a level of realism (and “hyper-realism”) that is unmatched by surface graphics
- Good: allows easy and natural exploration of volumetric datasets
- Bad: extremely computationally expensive!
- Bad: hardware acceleration is very costly ($3000+ vs $200+ for surface rendering)

# Surface Graphics vs. Volume Graphics

- Suppose we wish to animate a cartoon character on the screen
- Should we use surface rendering or volume rendering?
- Suppose we want to visualize the inside of a person's body?
- Now what should approach we use? Why?
- Could we use the other approach as well? How?
- We could visualize body as collection of surfaces

# Human Visual System and Color Theory

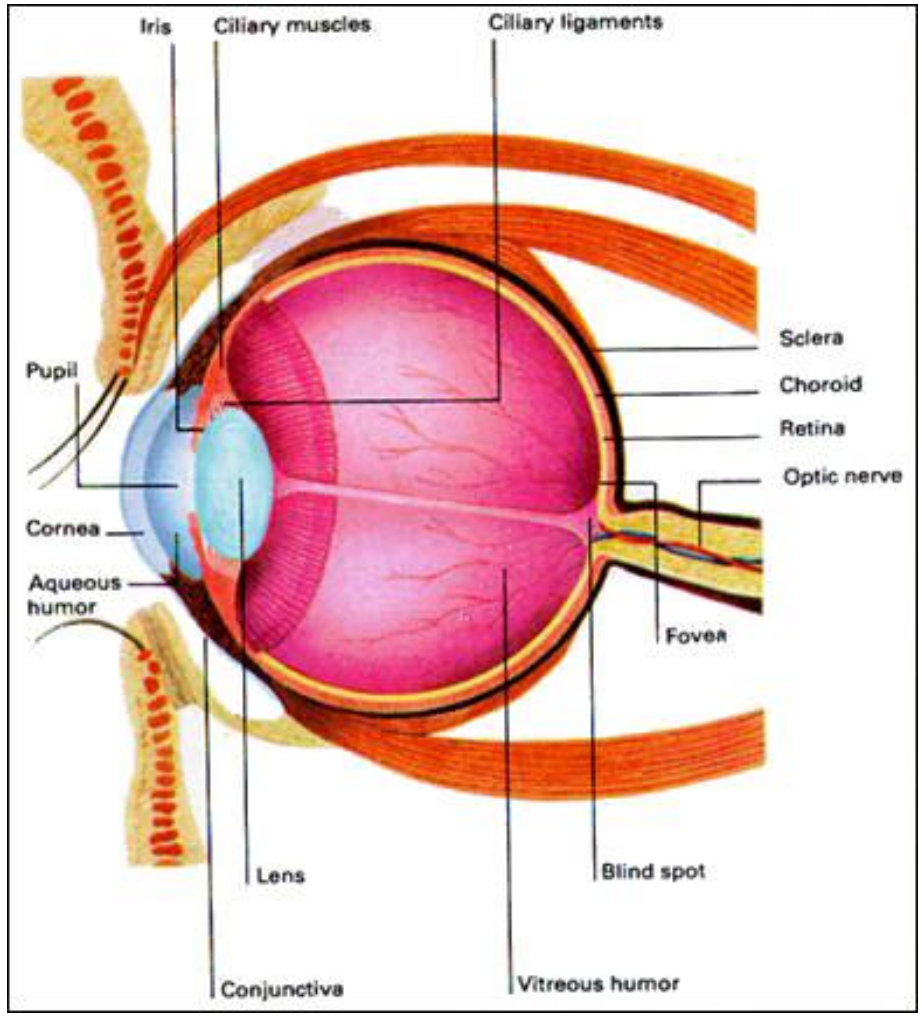
# Human Visual System and Color Theory

- Today: human visual system
- Human eye
- Color models
- Color perception

# Human Visual System

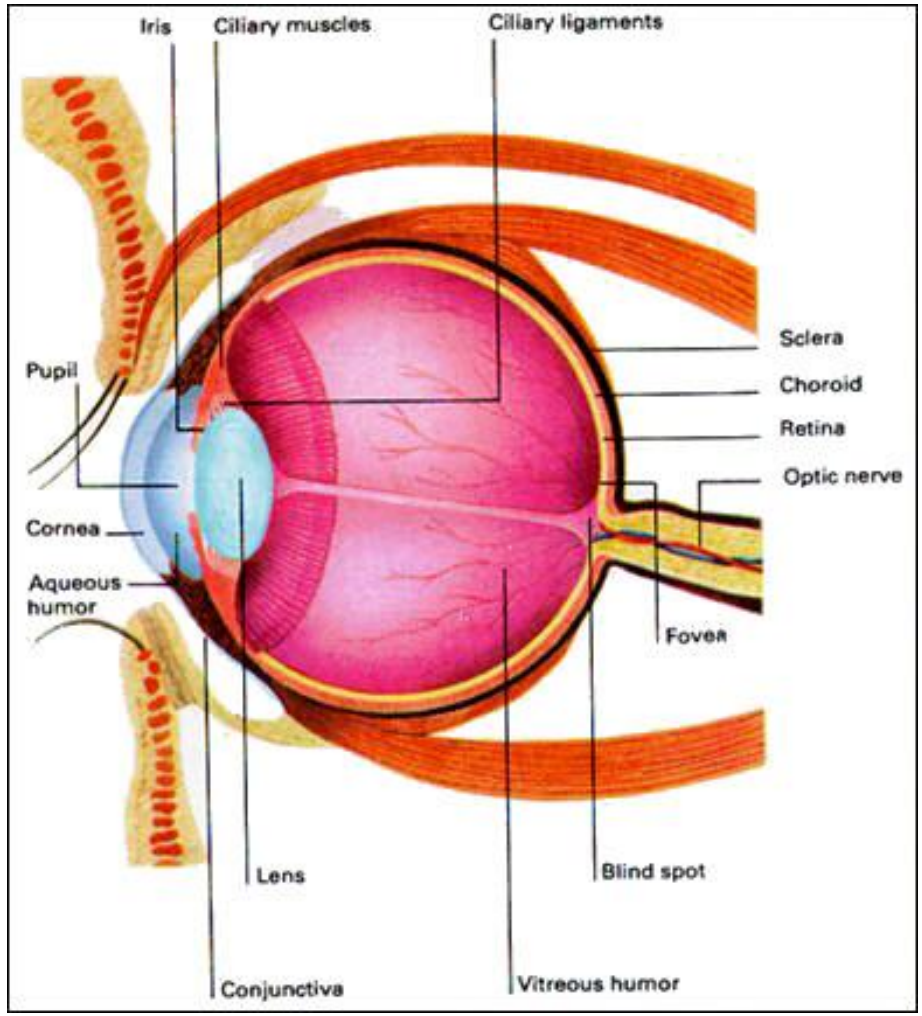
- How do we perceive the visible world?

1. Light enters eye and strikes lens
2. Muscles expand and contract to focus light on retina



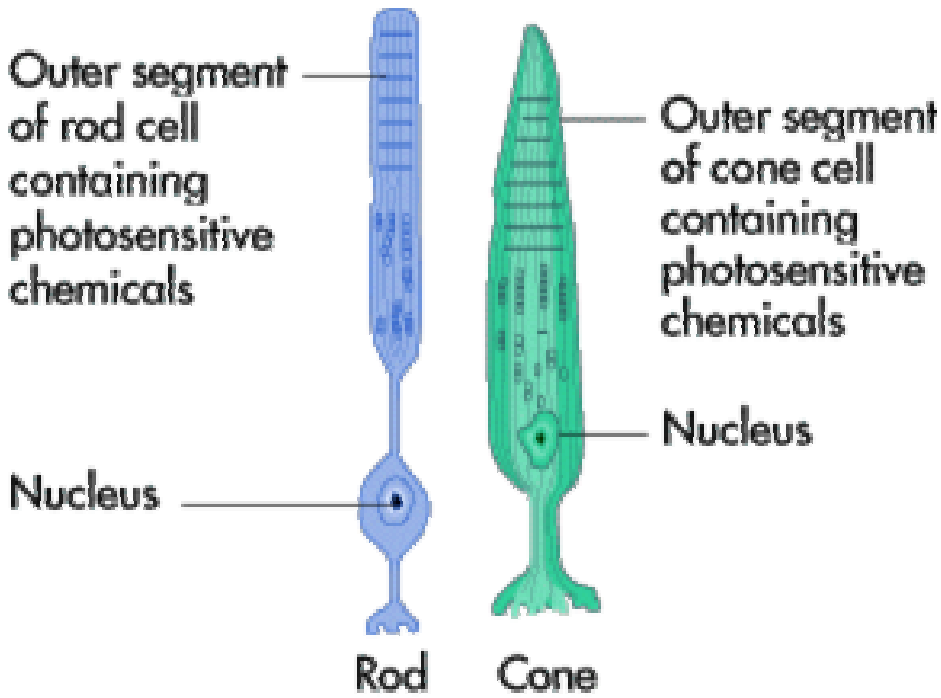
# Human Visual System

3. Retina senses light and contains **cone cells** and **rod cells**
4. Retinal nerve fibers connect to optic nerve, which carries signals to brain, where they are interpreted as an image



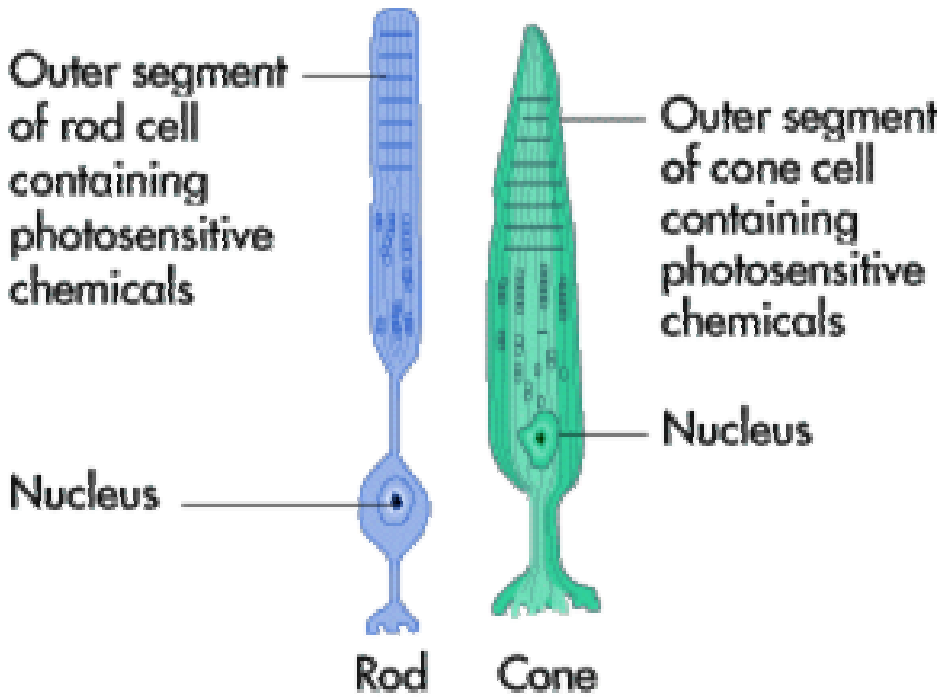
# Rods

- Spread all over retina
- 75-150 million
- Low resolution
- Don’t detect color
- Very sensitive to low light
- This is called **scotopic vision**



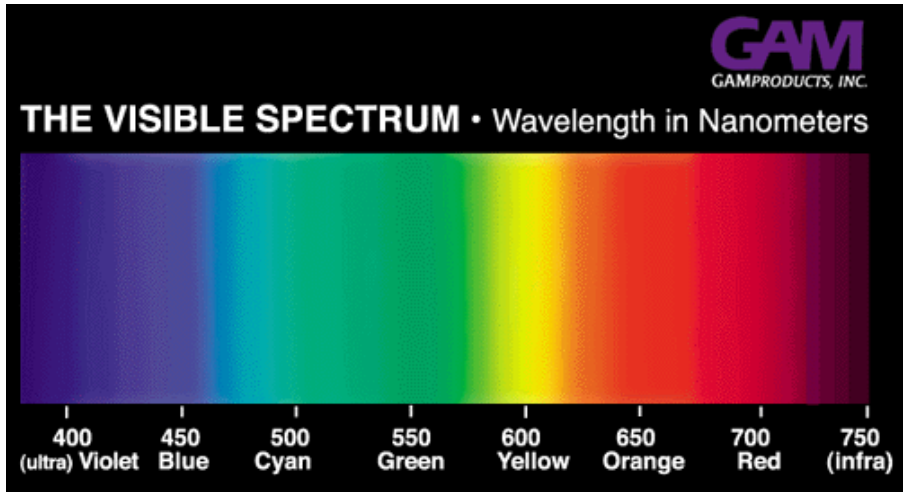
# Cones

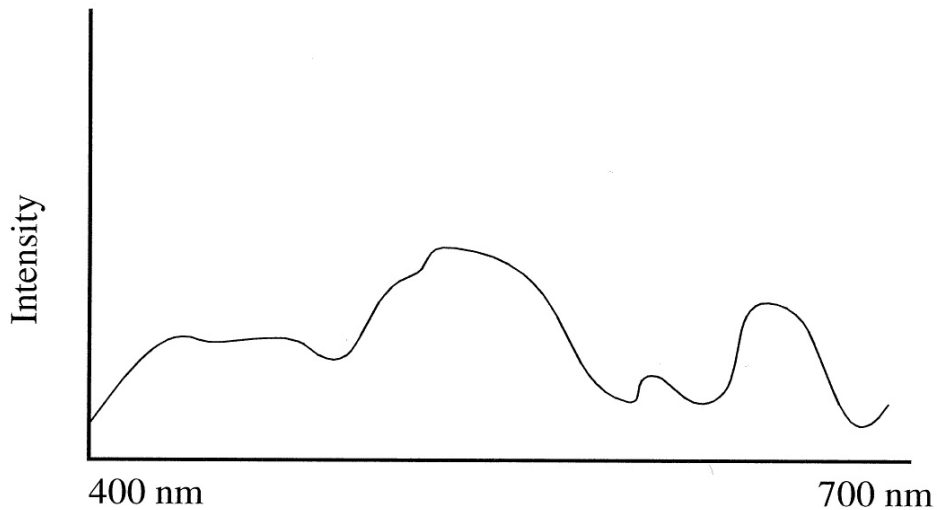
- Dense array of cells at retina center
- 6-7 million total
- High-resolution, detect color
- This is called **photopic vision**



# Color

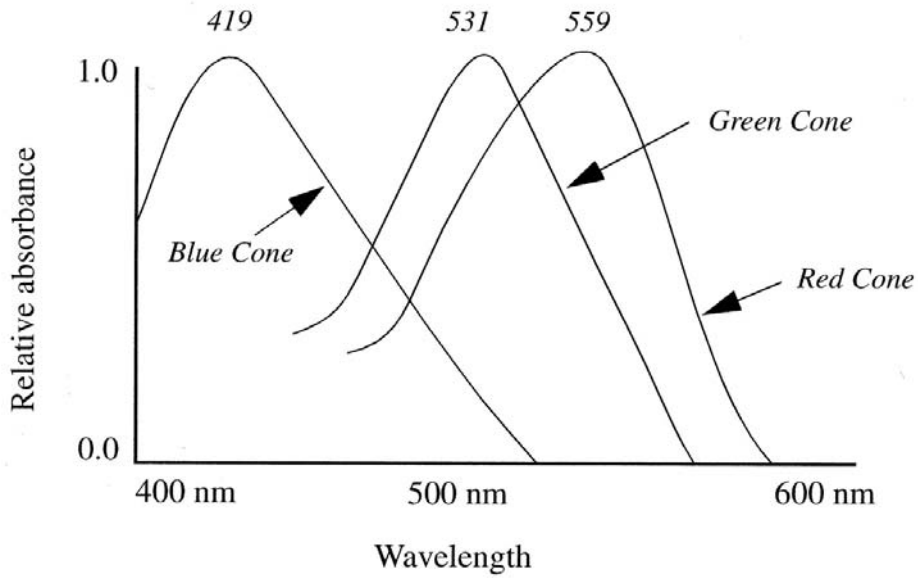
- Visible spectrum wavelengths 400-700 nm
- A given color has some distribution of these wavelengths
- Intensity of each wavelength determines contribution to color



# Color Receptors

- *Tristimulus theory*: "Red" cones (~60%), "green" cones (~30%) and "blue" cones (~10%)
- Mixing process takes place inside brain
- Graph of visible spectrum each type of cone is sensitive to:



# Human Eye Color Perception

- Human eye differentiates about 300 hues and 100-150 luminance variations. What does that mean?
- If red, green and blue cones are 60%, 30% and 10%, which colors can we perceive best?
- What does this mean for visualization?

# Human Eye Color Response

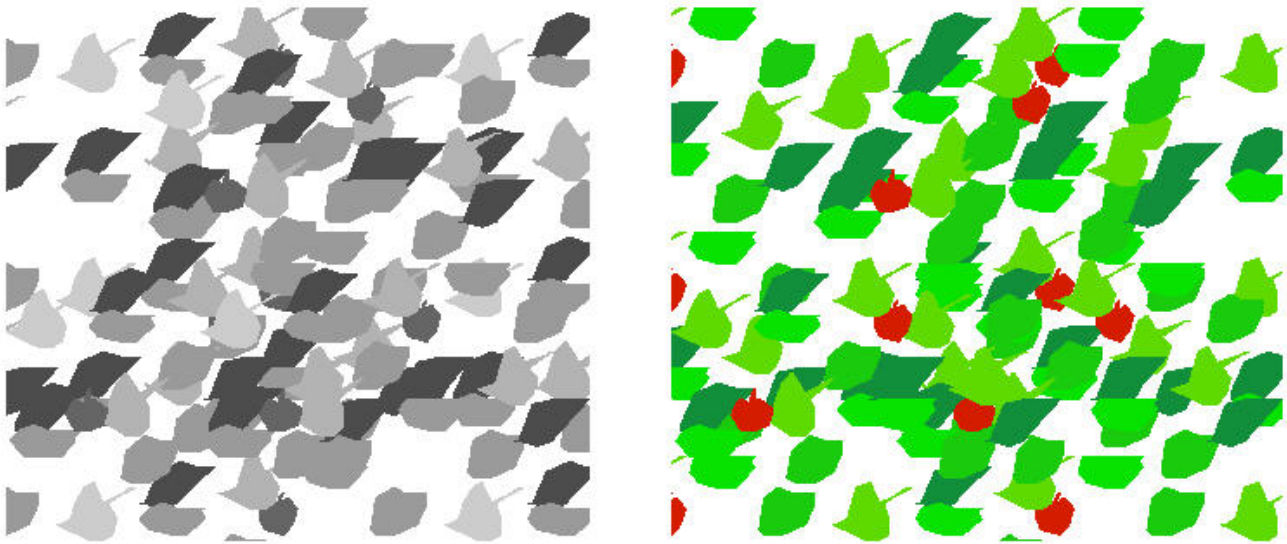
- Human eye responds to certain colors faster than others
- Color ranking (from best to worst):
  yellow > white > red > green > blue
- What colors should be used to highlight important features?
- Let's test your color response

# Human Eye Color Perception

- We are sensitive to small color differences
- Good at making side-by-side comparisons
- Not as good at identifying colors in isolation
- Hard: “Is that red, or orange-red, or red-orange, or maroon or…?”
- Easier: “Color A is redder than color B”

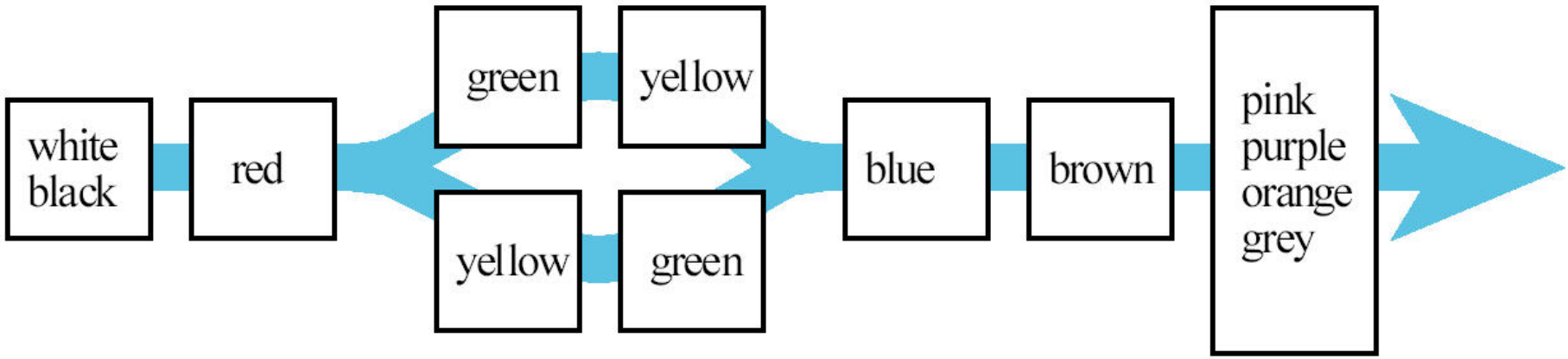
# Information Coding with Color

- Color good for *classification* – separation of data into classes
- In practice, only about six categories can be distinguished using color alone



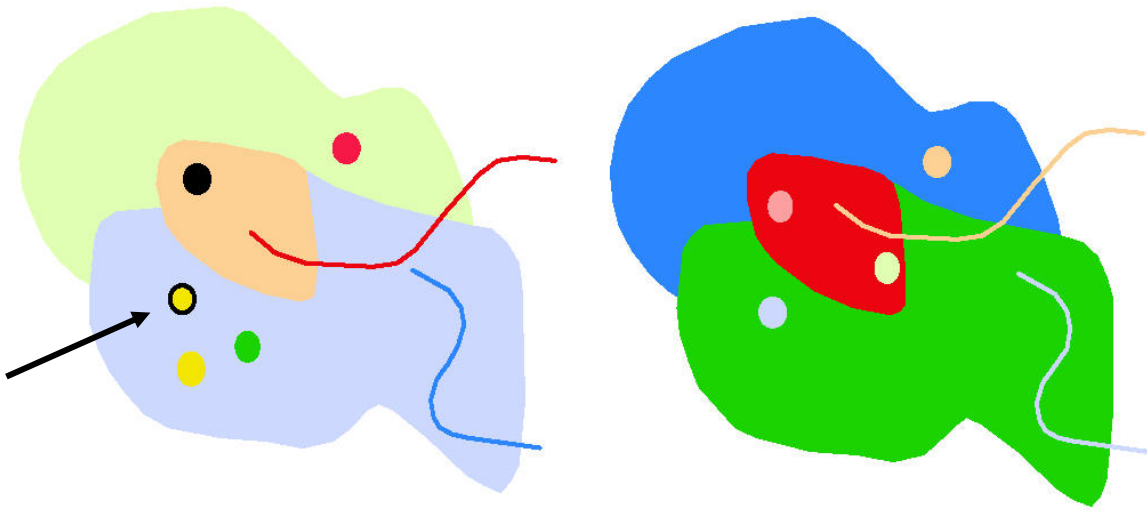
# Information Coding with Color

- A suggested order for adding color to visualizations



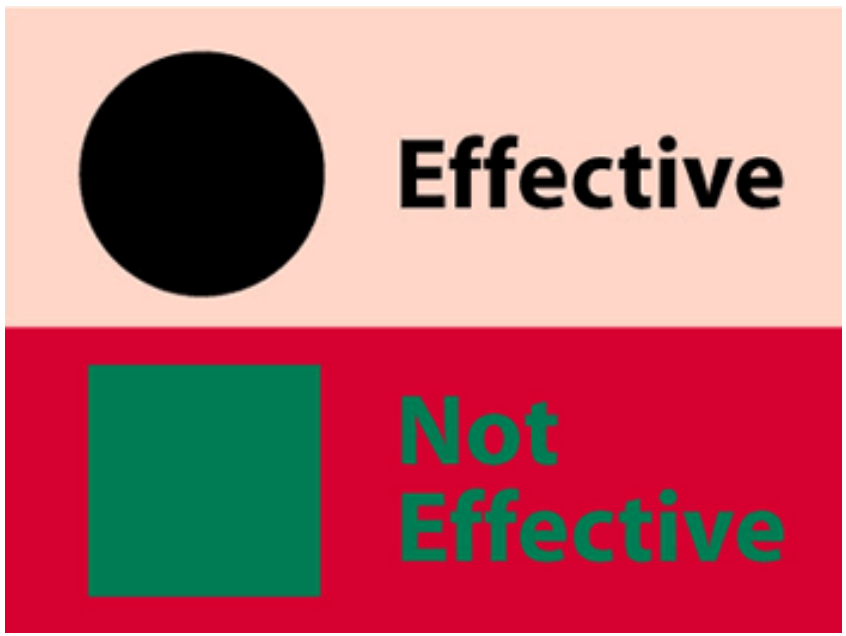
# Information Coding with Color – Helpful Tips

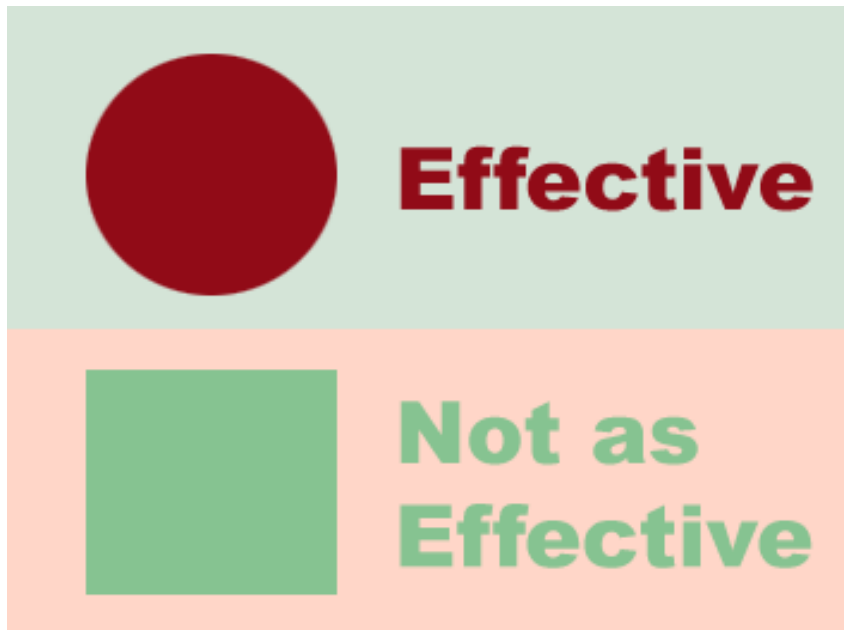
- Color coding
  - large areas: low saturation
  - small areas: high saturation
  - maintain luminance contrast
  - break iso-luminances with borders (?)



# Color Response and Perception Summary

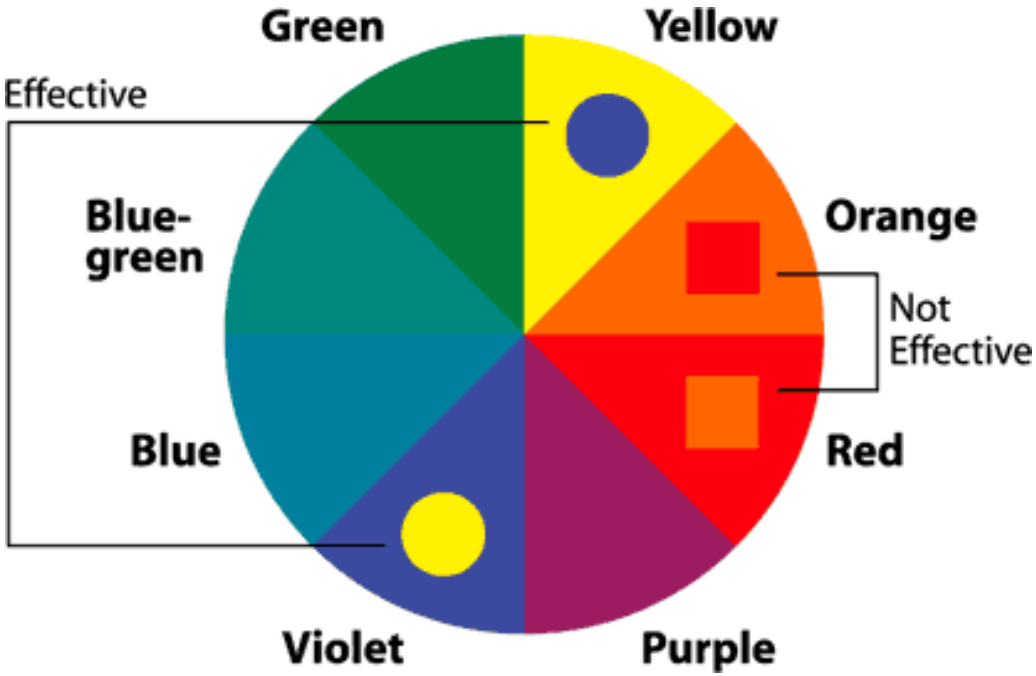
- Use bright colors to highlight important features
- Yellows, oranges, reds will be picked up first by the eye
- Make effective use of light/dark contrast:



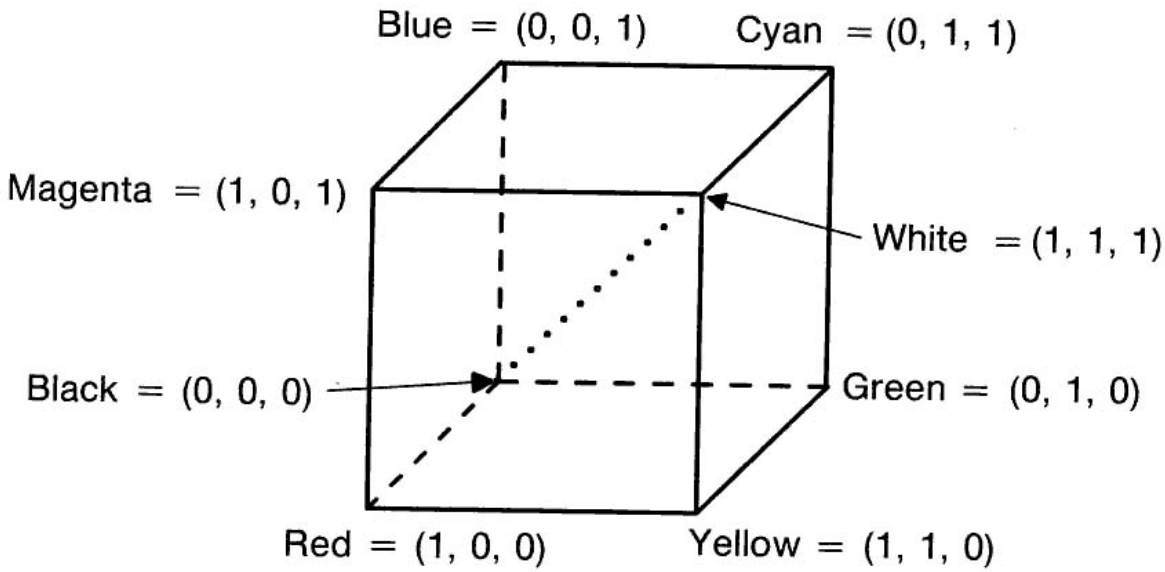
# Color Response and Perception Summary

- Also make effective use of color contrast to highlight important and interesting characteristics of data
- The human eye is very good at making side-by-side comparisons



# Computer Representation of Color

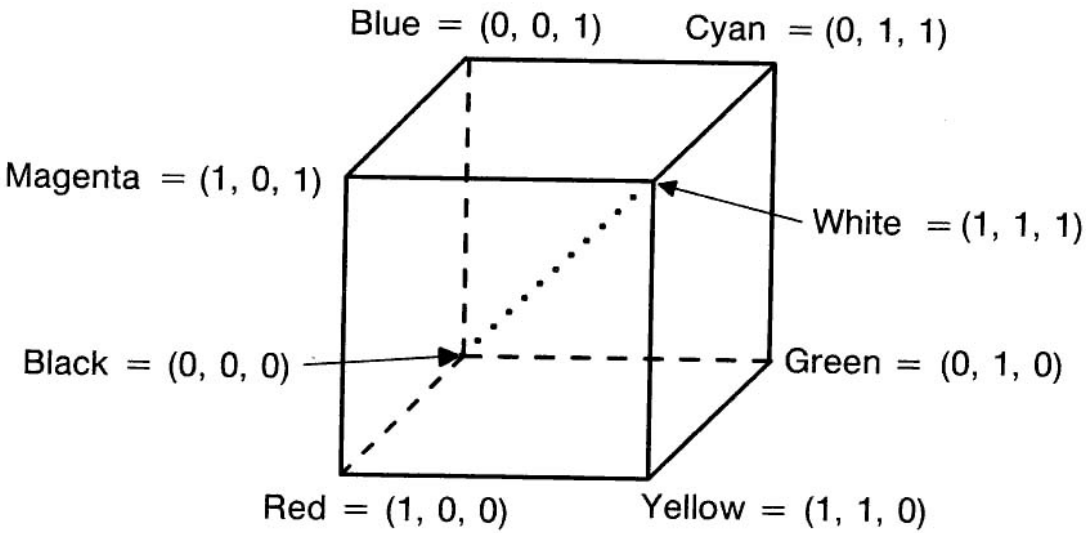
- Each screen pixel is combination of R, G and B light
- Three *color components* determine perceived color
- *RGB color model*
- R, G, and B in range [0.0, 1.0]



# RGB Color Arithmetic

- R + G = ?
- B + G = ?
- R + Y = ?

- R + C = ?
- G + M = ?
- B + Y = ?
- B + W = ?



# RGB Color Model

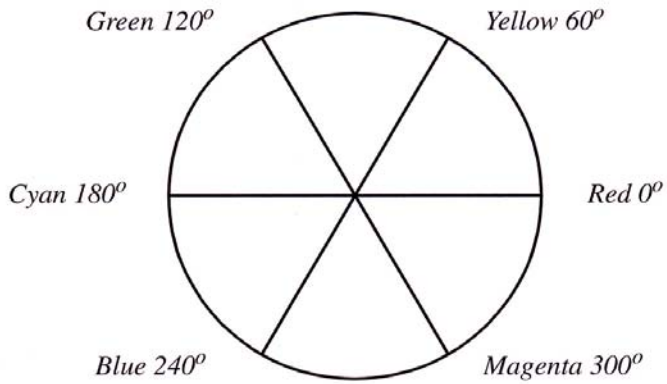
- Good: simple, easy hardware implementation
- Is it intuitive?
- How would you make a washed-out green?
- How would you make a bright blue?

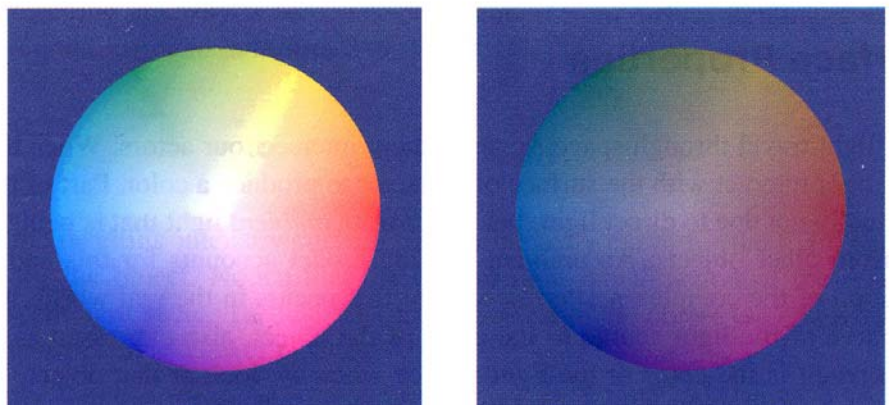
# HSV Color Model

- RGB: good for hardware, bad for human use
- Hard to change saturation and brightness
- *HSV color model* – more intuitive
- H = hue
- S = saturation
- V = value (brightness)

# HSV Color Model

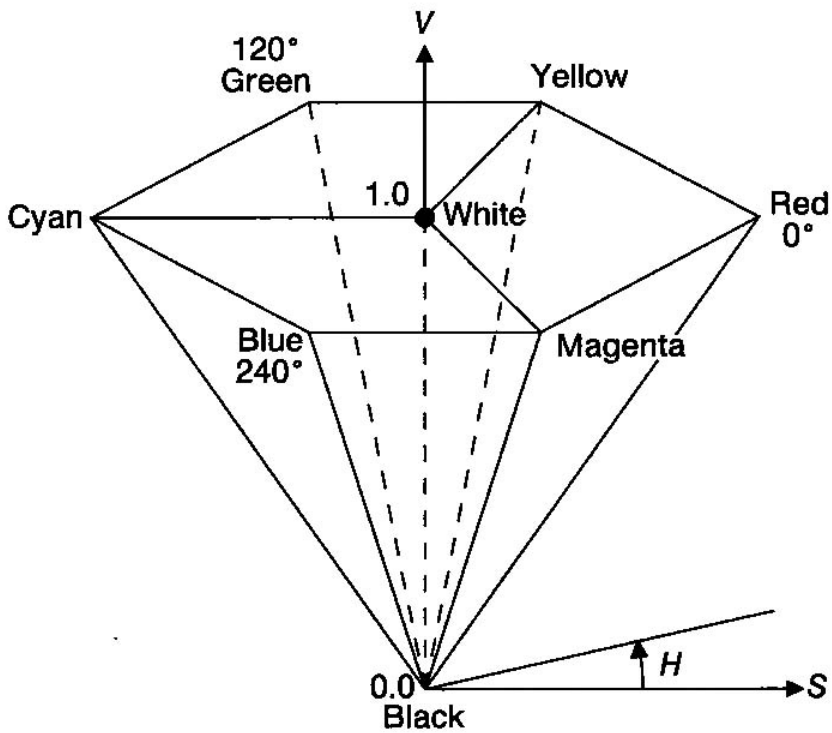
- The *hue* is the color
- Specified as angle around the *HSV (hex) cone*



# HSV Color Model

- *Saturation* measures vividness of color
- Distance from central axis
- *Value* measures brightness
- S, V in range [0.0, 1.0]
- H in range [$0^o$, $360^o$]



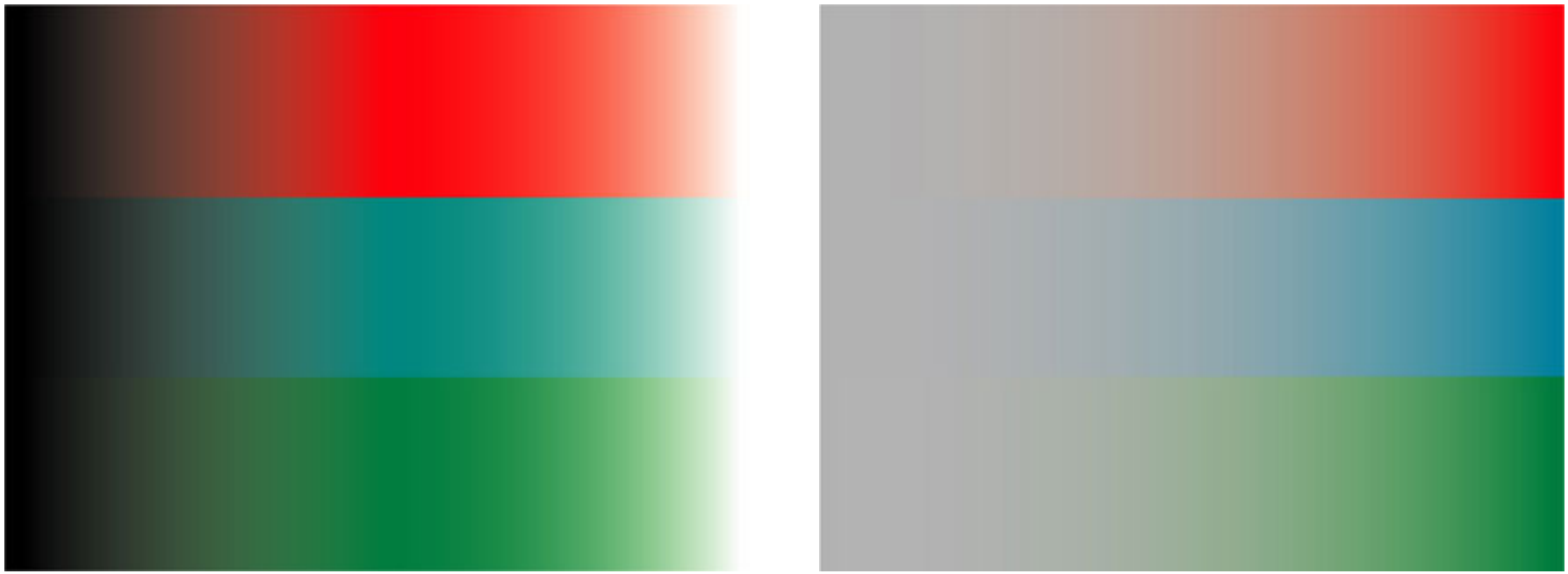
# HSV Color Model

- HSV very intuitive
- e.g., to brighten a color, increase V
- e.g., to wash it out (make grayer), decrease S
- How do we change H?
- Change angle
- Sequence of colors around cone:

$$R \rightarrow Y \rightarrow G \rightarrow C \rightarrow B \rightarrow M \rightarrow R$$

# HSV Examples

- Which component of HSV are we increasing in the left image? Right image?



# RGB vs. HSV

- Can use HSV in software, but convert to RGB for display
- Easy to convert between RGB and HSV
- Code on Web

| Color | RGB | HSV |
|---|---|---|
| Black | | |
| White | | |
| Red | | |
| Green | | |
| Blue | | |
| Yellow | | |
| Cyan | | |
| Magenta | | |
| Sky Blue | | |

# Modeling Methods and Techniques for Illumination and Shading

# Illumination and Shading

- *Illumination* and *shading* are two complementary aspects in Computer Graphics that add realism to rendered scenes
- Illumination refers to use of lights in virtual world
- Shading refers to effects that lights have on 3D objects in the scene
- Many kinds of *illumination models* and *shading models* in 3D computer graphics and visualization

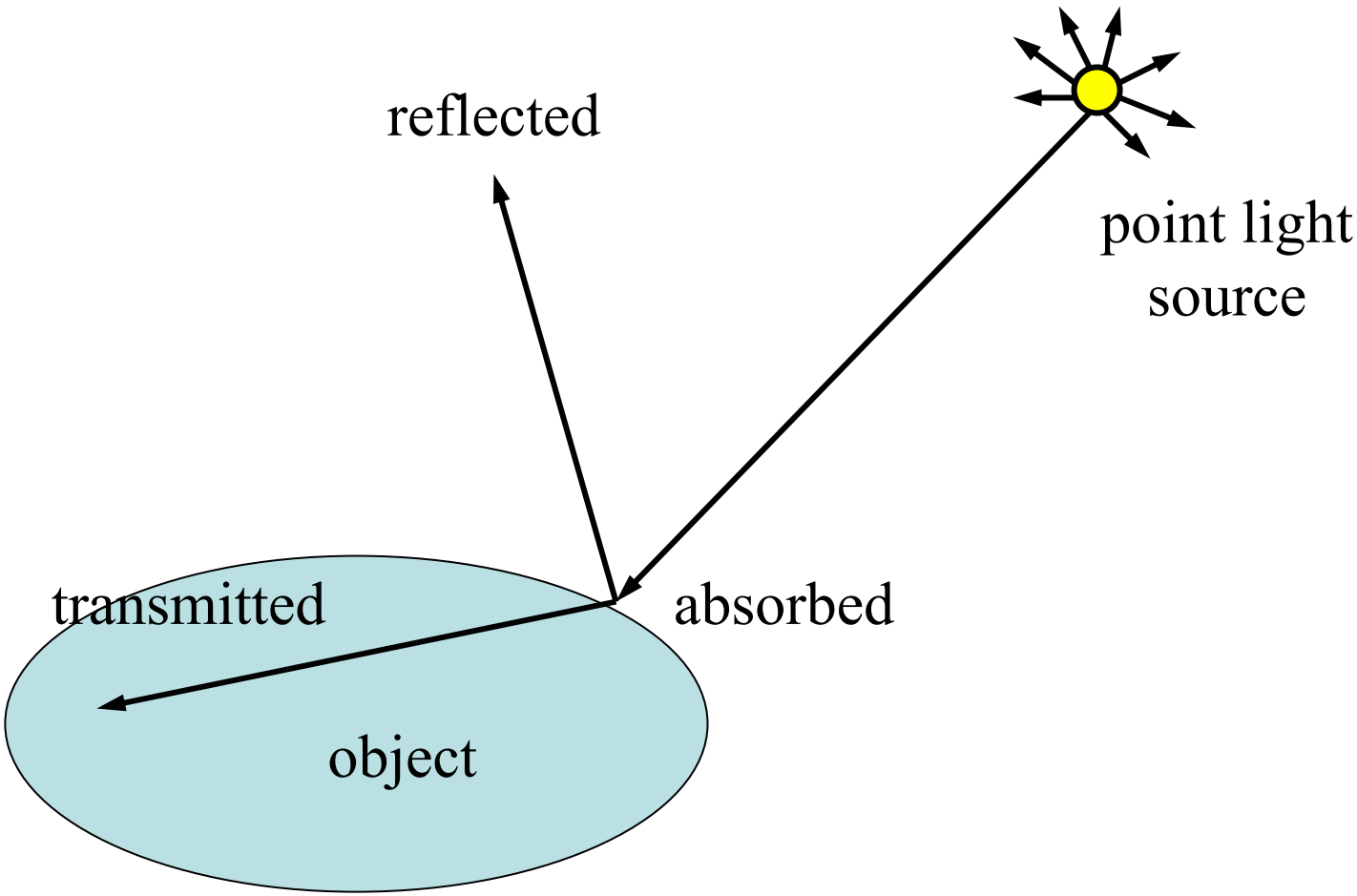
# Illumination

- Without lights a 3D scene is totally black
- Seek to simulate effects of light
- Simplest type of light is *point light source*
- Light is infinitely far away
- Light rays are parallel
- Is this a good approximation of a light bulb? Flash light? The sun?

# Shading Model

- A shading model checks lighting conditions and figures out what surface should look like based on lighting conditions and surface parameters:
  - Amount of light reflected (and which color(s))
  - Amount of light absorbed
  - Amount of light transmitted (passed through)
- Shading model tells us how much incoming light that strikes a surface is 1. reflected to the eye, 2. absorbed by the object, and 3. transmitted through the object

# Total Light Decomposition

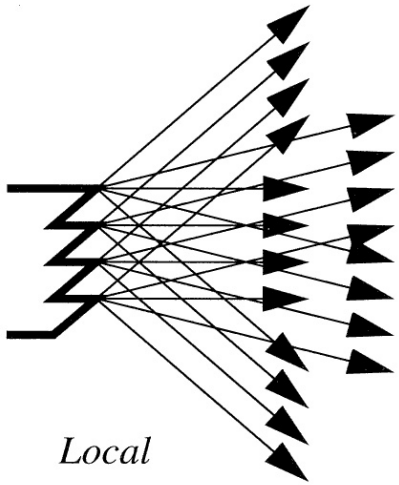


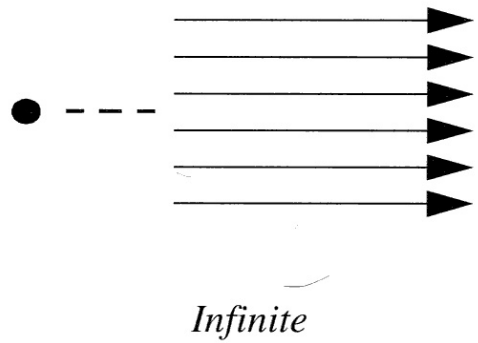
# Shading Model

- Typically in Computer Graphics, we are concerned with the reflected light – light which bounces off object and enters eye
- Other effects like refraction and translucency require more sophisticated shading models

# Local vs. Infinite Light Sources

**Local**




**Infinite**

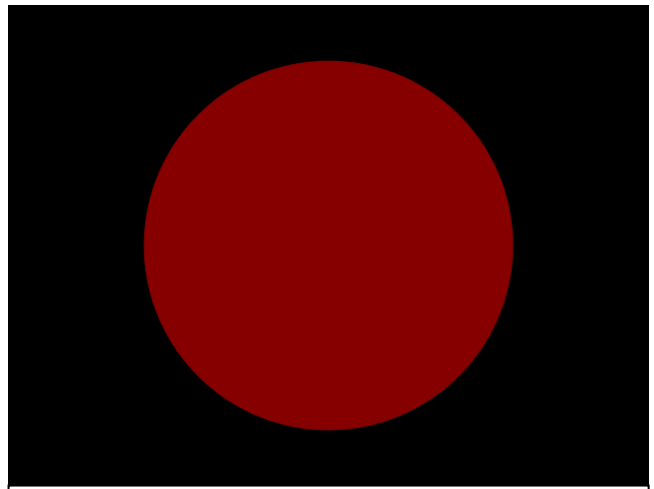
- Rays from a local light source emanate in different directions
- Rays from the infinite light source travel in the same direction

# Surface Properties – Ambient Lighting

- Rays of light strike objects or *actors* in the scene
- Illumination model determines how light and surface properties interact to generate a color image
- *Ambient lighting* is simplest illumination model
- It accounts for *indirect* light
- Models general level of brightness in the scene
- Accounts for light effects that are difficult to compute (secondary reflections, etc.)
- Constant for all surfaces and view directions (?)

# Surface Properties – Ambient Lighting

- Imagine yourself in room with curtains drawn
- Some sunlight will still get in, but it will have bounced off many objects before entering room
- When an object reflects this kind of light, we call it *ambient reflection*



Ambient-lit sphere

# Surface Properties – Ambient Lighting

- Ambient reflection can be expressed by this equation:

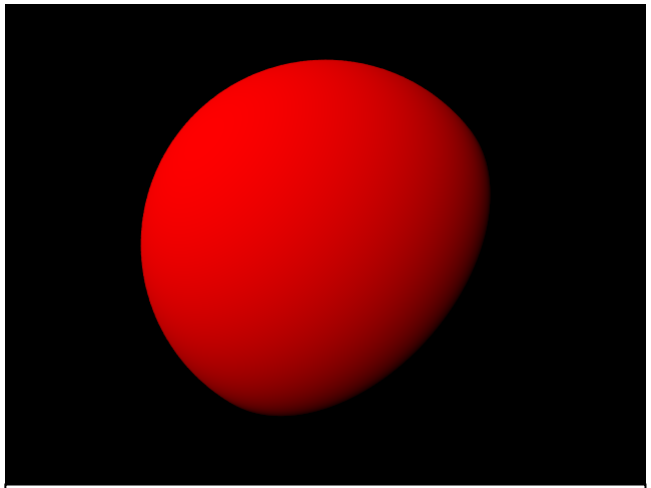
$$R_c = L_c \cdot O_c$$

- $R_c$ is color of reflected light
- $L_c$ is color of light source
- $O_c$ is color of object
- Shine white light on red sphere vs. shine red light on white sphere?
- Not the best notation really…

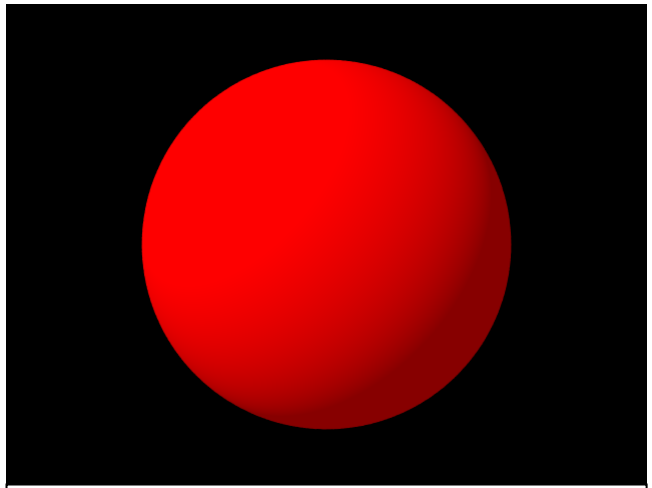
# Surface Properties – Diffuse Lighting

- Ambient lighting is a crude approximation of *secondary reflections*
- *Diffuse lighting* takes us one step closer to reality
- Direction of rays taken into consideration
- Unlike ambient reflection, diffuse reflection is dependent on location of light source relative to the object
- This is a type of *direct lighting*
- Models dullness, roughness of a surface
- Also called *Lambertian reflection*

# Surface Properties – Diffuse Lighting



Diffuse lighting



Ambient & diffuse

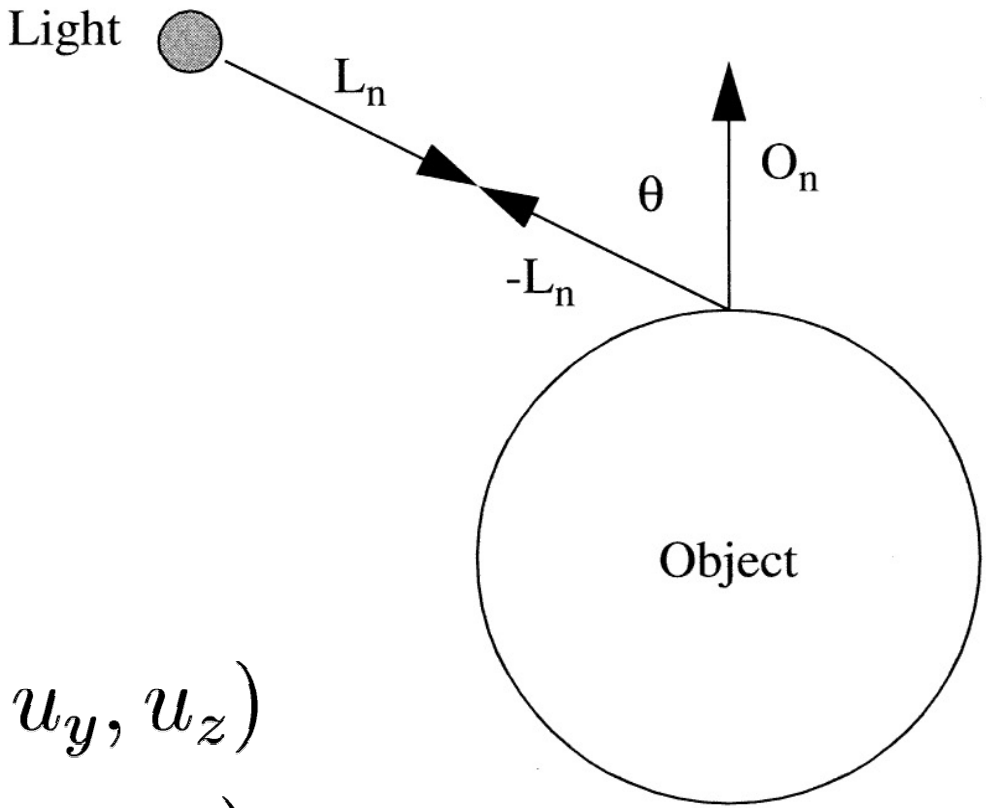
- Note difference between diffuse alone and diffuse with ambient lighting
- Suppose we moved light to around back of sphere – remind us: why would the sphere get darker?

# Surface Properties – Diffuse Lighting

Light Color = $L_c$

$\cos\theta = (\vec{O}_n \cdot -\vec{L}_n)$

Object Color = $O_c$




$$\vec{u} = (u_x, u_y, u_z)$$

$$\vec{v} = (v_x, v_y, v_z)$$

$$\vec{u} \cdot \vec{v} = u_x v_x + u_y v_y + u_z v_z$$

# Surface Properties – Diffuse Lighting

$$R_c = L_c O_c [\vec{O}_n \cdot (-\vec{L}_n)]$$

- $R_c$ is color of reflected light
- $L_c$ is color of light source
- $O_c$ is color of object
- $\vec{O}_n$ is object's *normal vector* – direction surface is pointing at that position
- $\vec{L}_n$ is *light vector* – direction of the light ray
- What's the relationship between $\vec{L}_n$ and the type of light source (local or infinite)?

# Surface Properties – Diffuse Lighting

$$R_c = L_c O_c [\vec{O}_n \cdot (-\vec{L}_n)]$$

- We assume that $O_n$ and $L_n$ are unit vectors (length = 1)
- We can *normalize* a vector by dividing each component by vector's length:

$$\vec{v} = (v_x, v_y, v_z)$$

$$|\vec{v}| = \sqrt{(v_x^2 + v_y^2 + v_z^2)}$$

$$\text{Normalized: } \frac{\vec{v}}{|\vec{v}|} = \left( \frac{v_x}{|\vec{v}|}, \frac{v_y}{|\vec{v}|}, \frac{v_z}{|\vec{v}|} \right)$$

# Surface Properties – Diffuse Lighting

$$R_c = L_c O_c [\vec{O}_n \cdot (-\vec{L}_n)]$$

- Dot product between -1 and +1
  - What does it mean when it is > 0?
  - What does it mean when it is < 0?
  - What does it mean when it is = 0?
- Given this knowledge, can we avoid computing the diffuse lighting equation entirely in some situations?

# Surface Properties – Diffuse Lighting

- Key points for diffuse lighting:
  - position of light w.r.t. object is important
  - models a rough surface
  - does not model shiny objects
  - contribution of a light source to the diffuse shading of an object computed with a dot product
- Compare: ambient vs. diffuse:

$$R_c = L_c \cdot O_c \qquad R_c = L_c O_c [\vec{O}_n \cdot (-\vec{L}_n)]$$

# Surface Properties – Diffuse Lighting

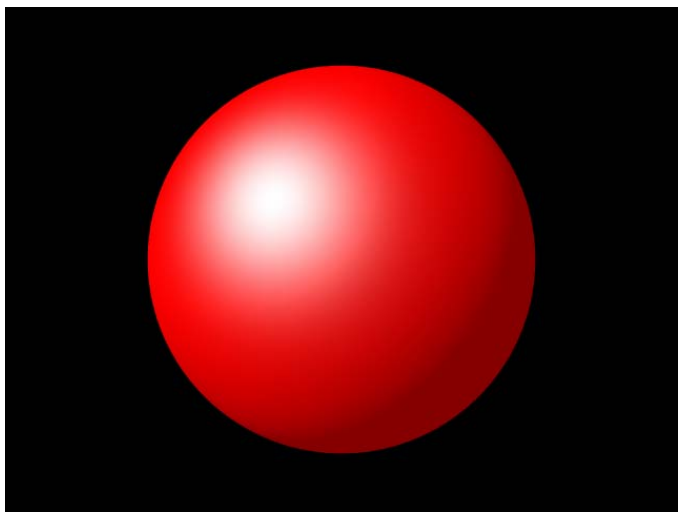
- Ambient vs. diffuse
- Ambient reflection of an object independent of light position, unlike diffuse lighting
- In this sense, does an ambient light source have a position?

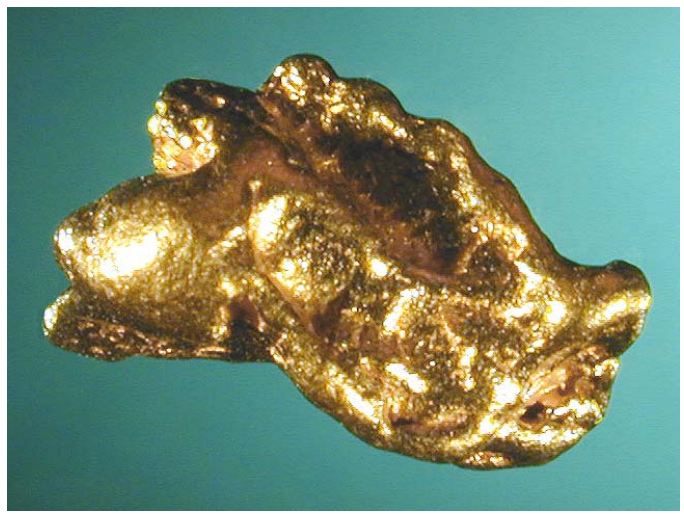
# Surface Properties – Specular Lighting

- Models reflections on shiny surfaces (polished metal, chrome, plastics, etc.)
- Specular reflection is *view-dependent* –specular *highlight* changes as camera's position changes
- Diffuse reflection is *view-independent* – reflection model is a function of light source direction and surface direction (normal)
- Specular reflection is a function of the light source direction, the surface direction, ***and the view direction***

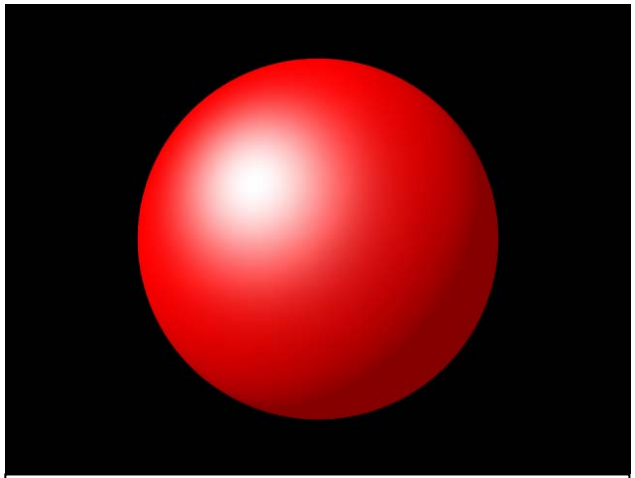
# Surface Properties – Specular Lighting

- Need angle light source makes with surface, and angle viewing ray makes with surface
- Example: chrome on your car shines in different ways depending on where you stand when looking at it

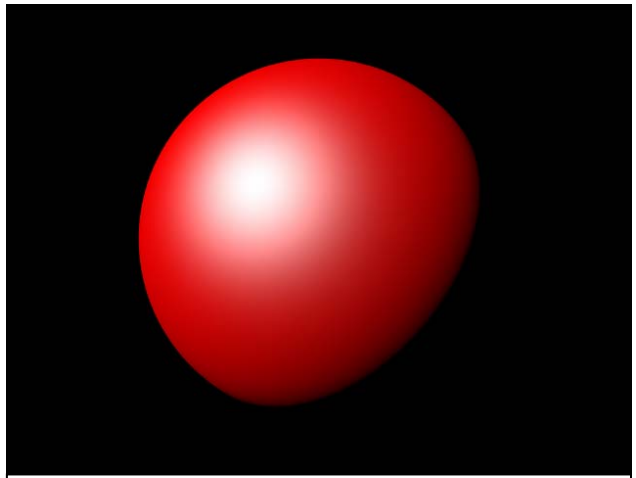




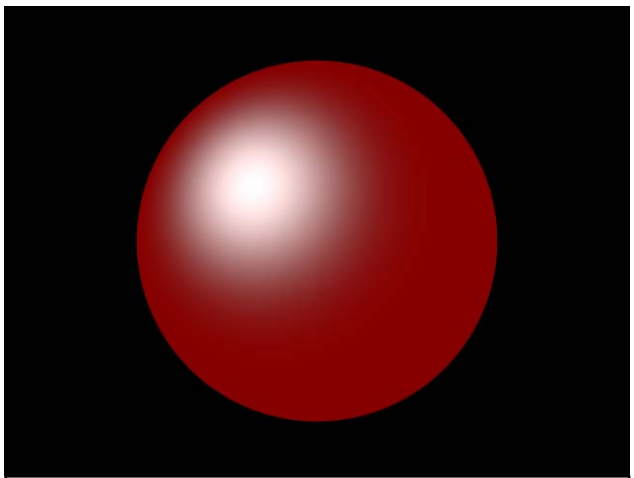
# Surface Properties – Specular Lighting



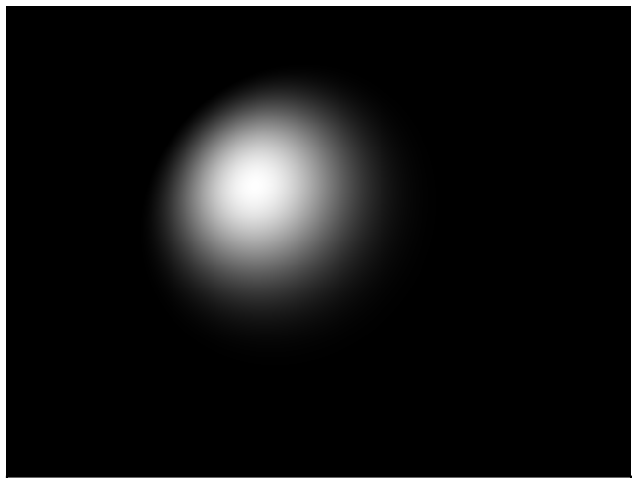
Specular & diffuse & ambient



Specular & diffuse

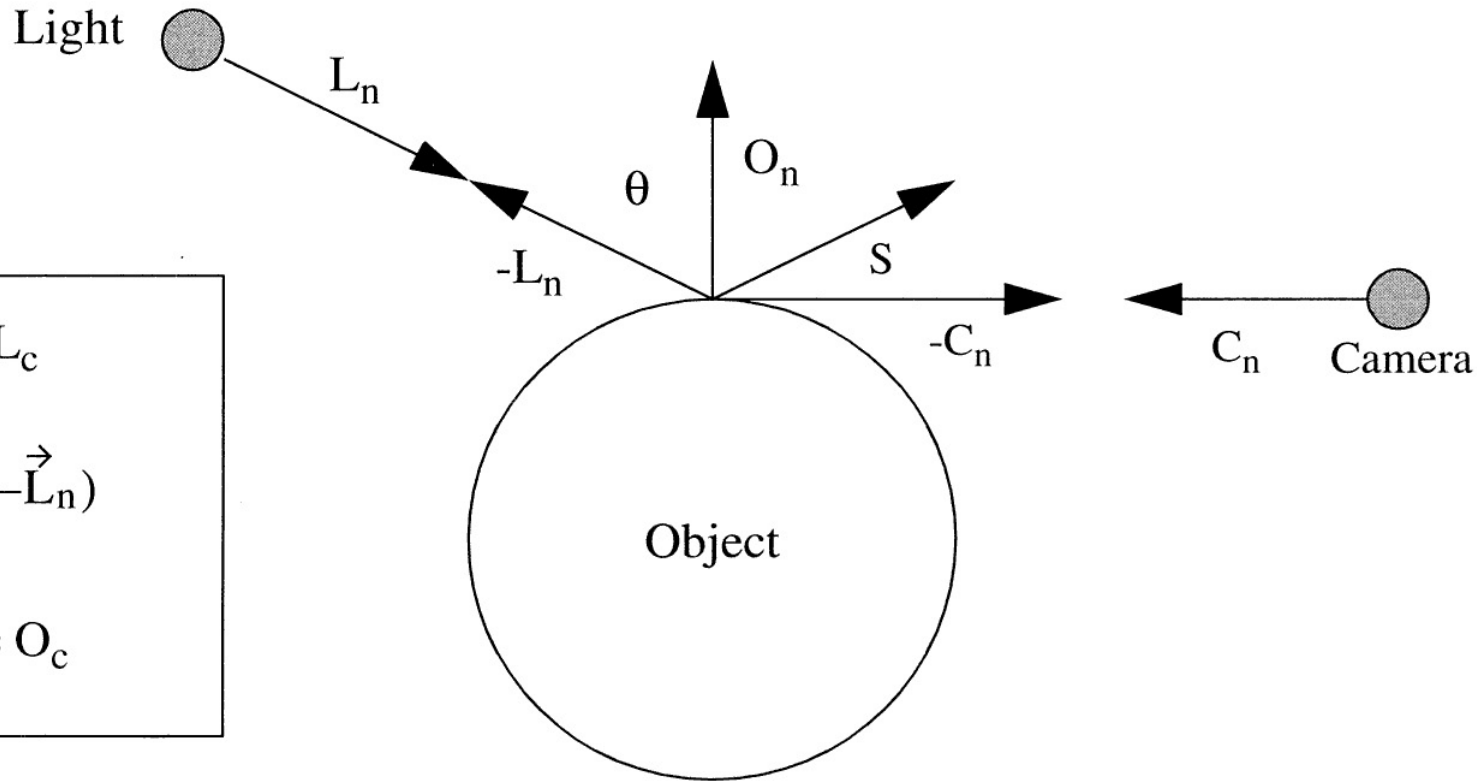


Specular & ambient



Specular only

# Surface Properties – Specular Lighting




Light Color = $L_c$

$\cos\theta = (\vec{O}_n \cdot -\vec{L}_n)$

Object Color = $O_c$

# Surface Properties – Specular Lighting

$$R_c = L_c O_c [\vec{S} \cdot (-\vec{C}_n)]^{O_{sp}}$$

$$\vec{S} = 2[\vec{O}_n \cdot (-\vec{L}_n)]\vec{O}_n + \vec{L}_n$$

- S is the direction of specular reflection
- The angle S makes with $O_n$ is the same angle $-L_n$ makes with $O_n$: $\theta$
- $C_n$ is the viewing direction
- $O_{sp}$ is the *specular power* and indicates how shiny the object is

# Surface Properties – Specular Lighting

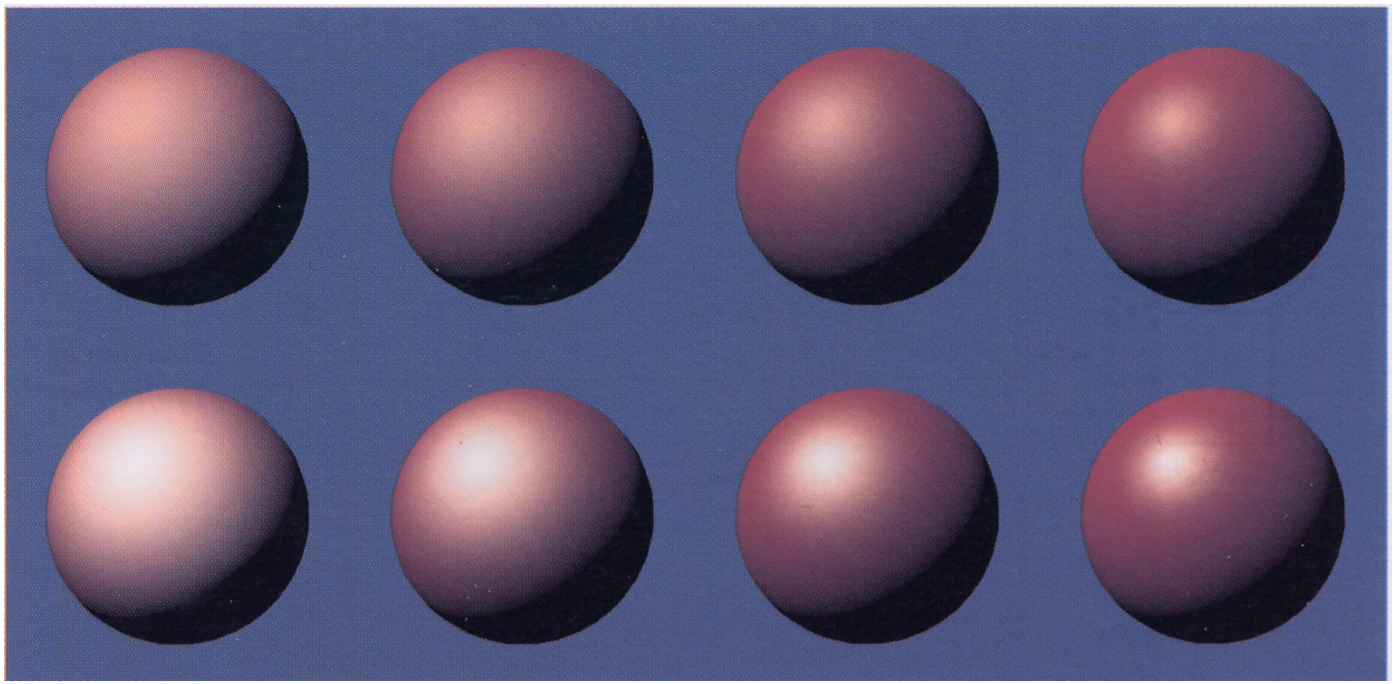
- Specular power indicates how quickly the specular reflection *diminishes* as direction of specular reflection deviates from view direction
- Specular power controls the size of specular highlight
- Inverse relationship:

$$R_c = L_c O_c [\vec{S} \cdot (-\vec{C}_n)]^{O_{sp}}$$

$$\vec{S} = 2[\vec{O}_n \cdot (-\vec{L}_n)]\vec{O}_n + \vec{L}_n$$

# Surface Properties – Specular Lighting

- Top row: specular *intensity* = 0.5 ($O_c$, essentially)
- Bottom row: specular intensity = 1.0
- Left to right: specular power = 5, 10, 20, 40



# Surface Properties – Total Illumination

- Ambient, diffuse and specular reflection are usually combined into a single equation:

$$R_c = O_{ai}O_{ac}L_c - O_{di}O_{dc}L_c(\vec{O}_n \cdot \vec{L}_n) + O_{si}O_{sc}L_c[\vec{S} \cdot (-\vec{C}_n)]^{O_{sp}}$$

- $O_{ai}$, $O_{di}$ and $O_{si}$ control the amounts of ambient, diffuse and specular lighting, with values in [0.0, 1.0] (these three values are called *reflection coefficients*)
- $O_{ac}$, $O_{dc}$ and $O_{sc}$ indicate the colors to be used with each type of lighting (specular color, $O_{sc}$, is usually white)

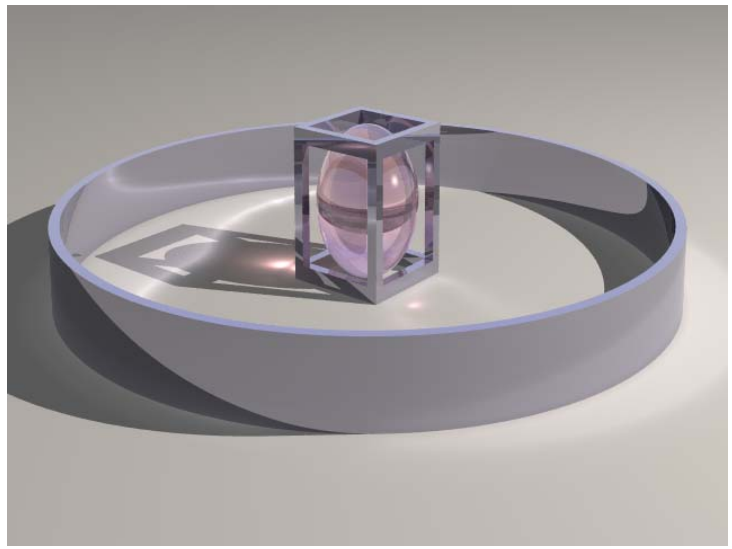
# Surface Properties – Total Illumination

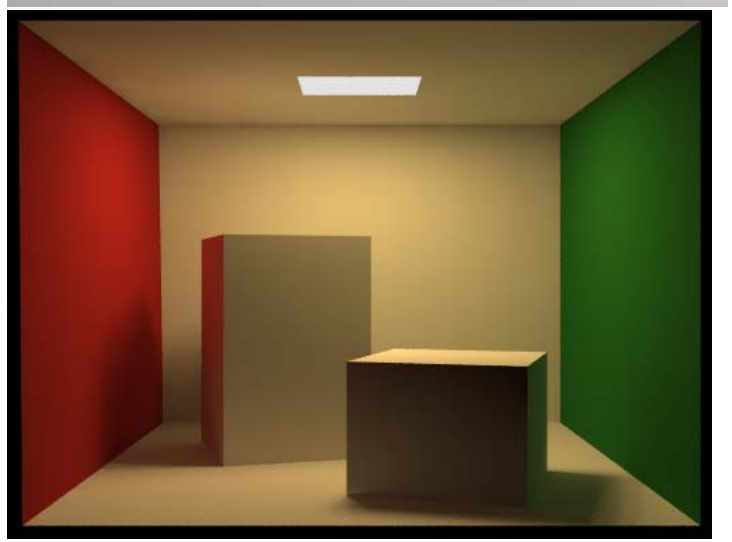
$$R_c = O_{ai}O_{ac}L_c - O_{di}O_{dc}L_c(\vec{O}_n \cdot \vec{L}_n) + O_{si}O_{sc}L_c[\vec{S} \cdot (-\vec{C}_n)]^{O_{sp}}$$

- What if $O_{sp} = 0$?
- What if $O_{sp}$ = infinity?
- What if some vectors are not normalized?
- How would we disable ambient reflection?
- What if some dot product is negative? What does this indicate? How should it be handled by the illumination equation?

# Other Shading and Illumination Effects

- Area lights
- Shadows
- Refraction
- Reflection
- Caustics
- Color bleeding
- Radiosity
- How do we generate these effects?



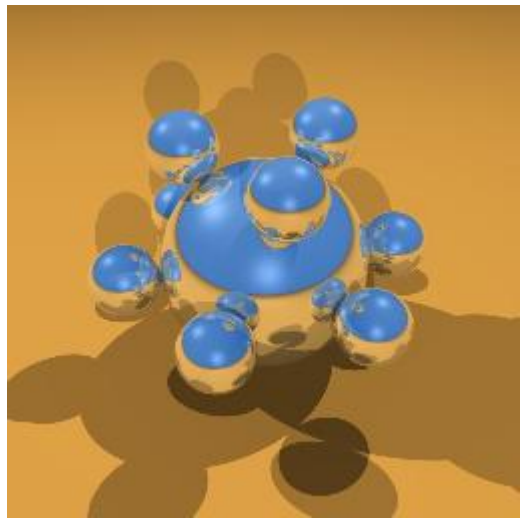


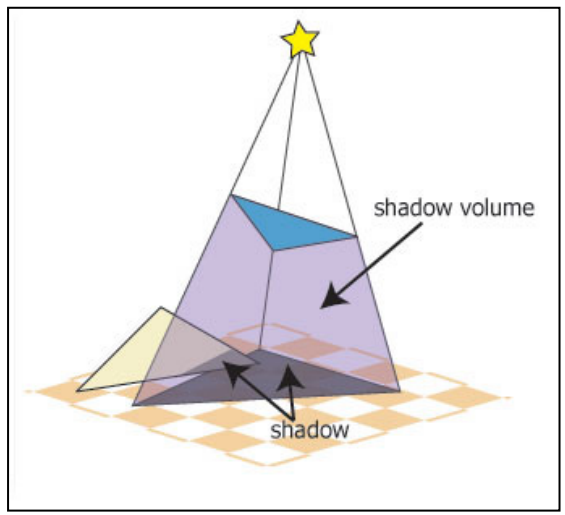
# Global Illumination

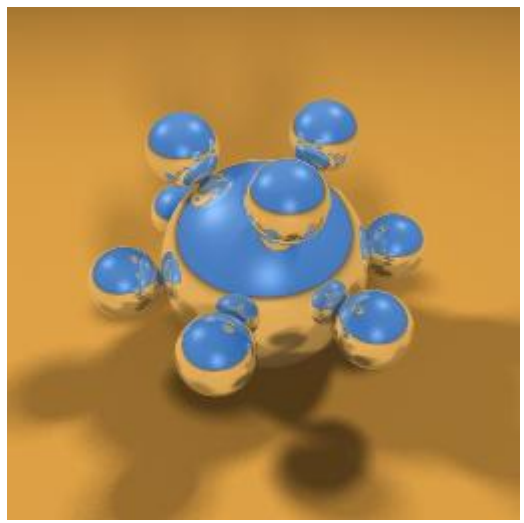
- These effects require *global illumination*, which is capable of generating all those *photorealistic* images you see in movies and special effects
- Most require the use of ray-tracing and radiosity, an $O(n^2)$ illumination technique
- Want to try it yourself? Go to *www.povray.org* and try out the free POV-Ray ray-tracing program

# Shadows

- Hard and soft shadows
- Hard shadows: caused by very distant light sources, like the sun
- Soft shadows: caused by close light sources, usually area light sources, like light bulbs
- Several techniques for generating shadows





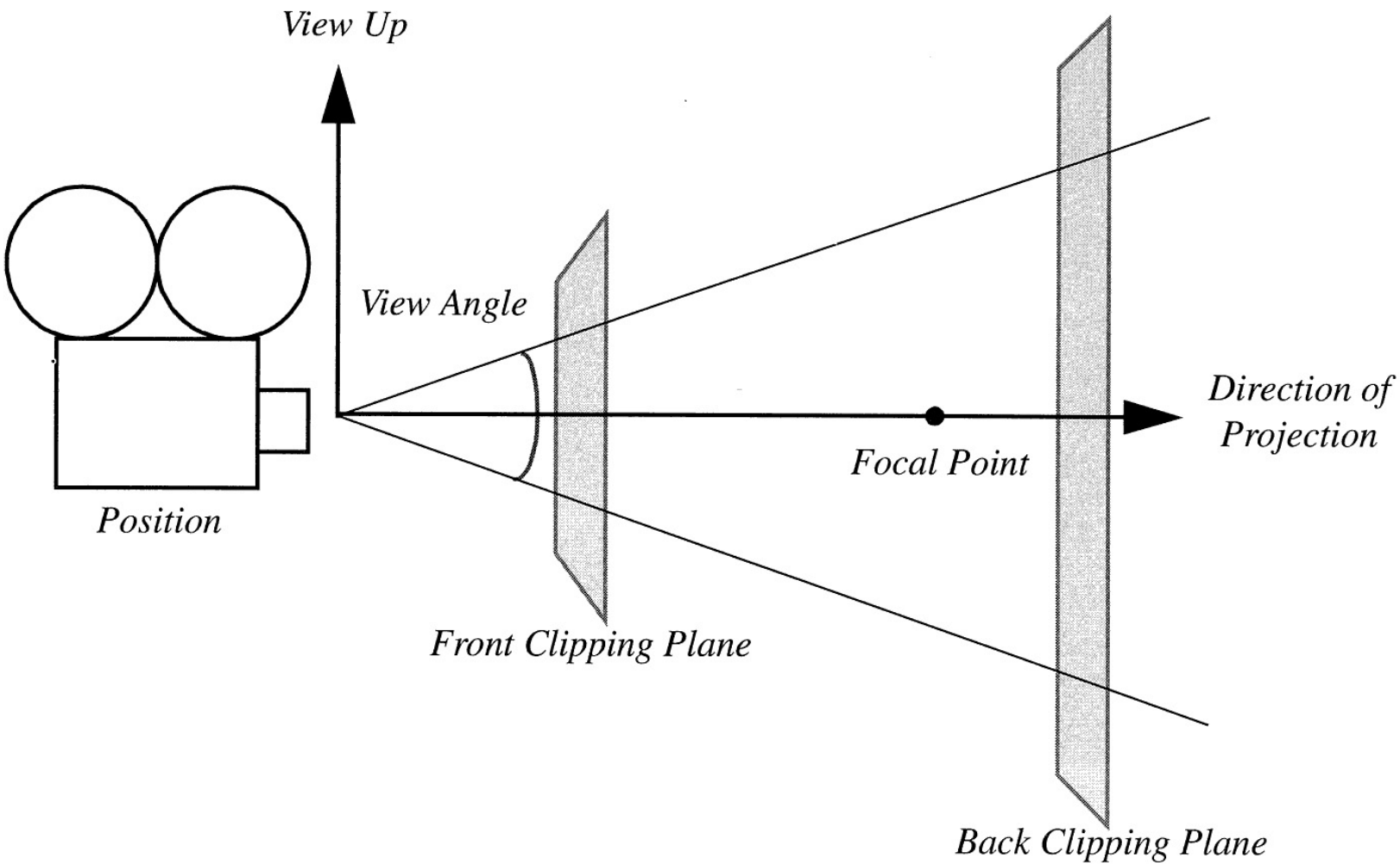



# Key Elements of Cameras and Geometric Coordinate Systems

# Cameras

- We have light sources that illuminate 3D objects (or actors) in our virtual scene
- Rays of light interact with surface properties and generate colors according to the illumination model
- But how do we view the scene, select the position and orientation of the viewpoint?
- This is where the virtual camera comes in

# Camera Attributes



# Camera Attributes

- *Position* – given in (x,y,z) coordinates
- *Up-vector* – orients the camera, given in (x,y,z)
- *Direction of projection* – points the camera in some (x,y,z) direction; also called *viewing direction*
- Why is the up-vector needed if we have a direction of projection?
- Why is the direction of projection needed if we have an up-vector?

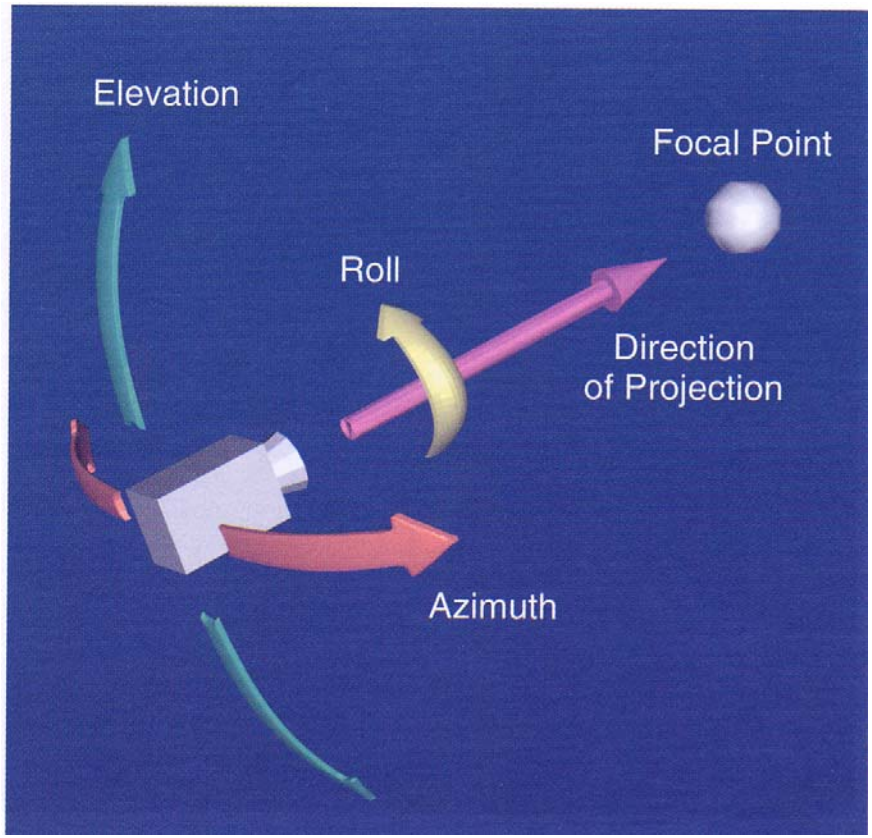
# Camera Attributes

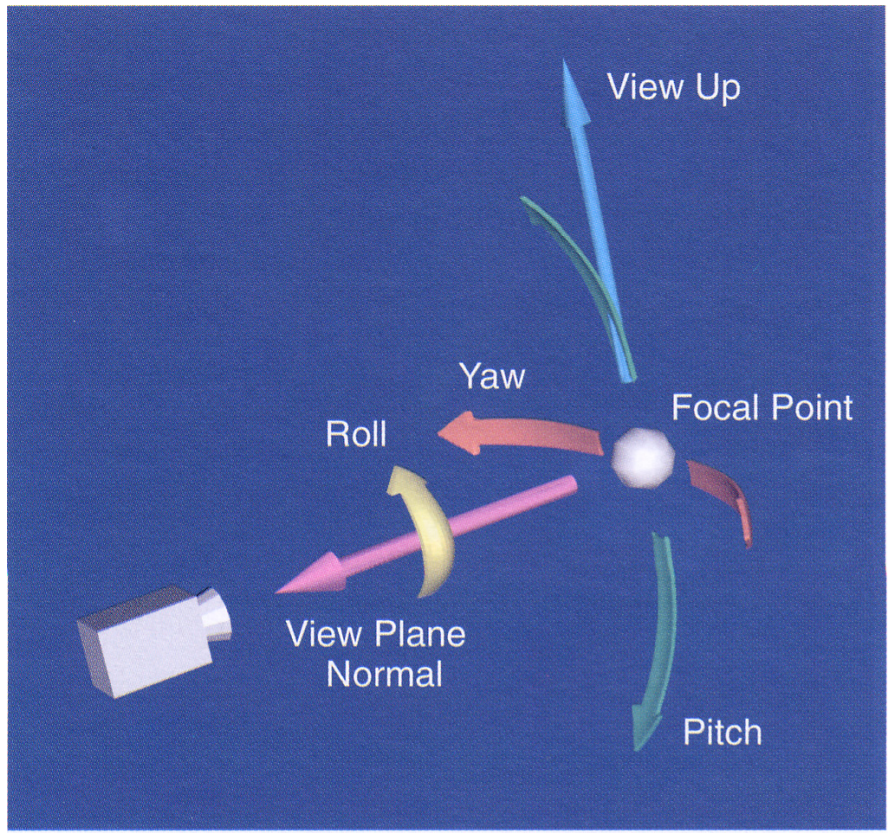
- *Front and back clipping planes* – determine which objects *might* be visible
- Planes perpendicular to viewing direction
- Specified as distances along viewing direction
- Also called *near and far clipping planes*
- Objects on near side of front clipping plane and on far side of back clipping plane are invisible
- Objects between the clipping planes may occlude each other and may be fully visible, partially visible, or invisible

# Camera Manipulation

- Nuisance to manipulate the camera by changing all those parameters
- Usually its easier to specify camera movements with respect to the camera's *focal point*, the position in space at which the camera is pointing
- Consider taking a portrait:
  - Move around the person
  - Move forward and backward w.r.t. to person
  - Move camera up and down
  - Rotate camera while standing still

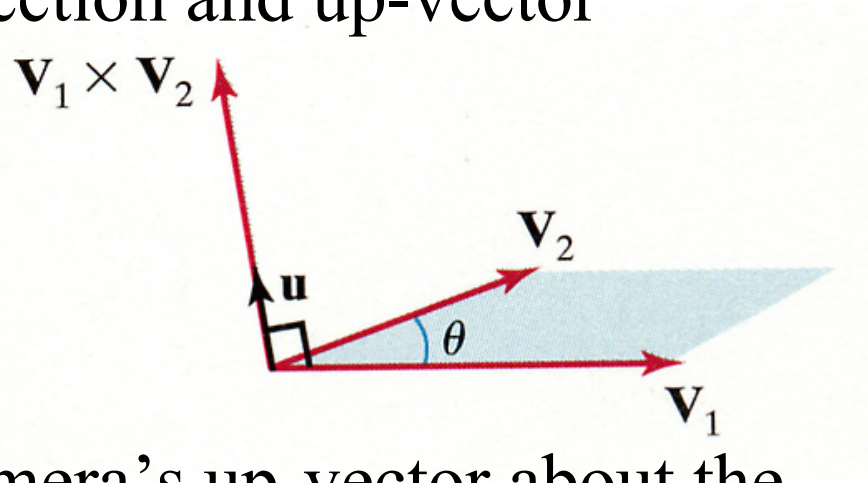
# Camera Manipulation



# Camera Manipulation

- Changing *azimuth* = rotating camera's position around its view vector w.r.t. focal point
- Changing *elevation* = rotating camera's position around cross-product of view direction and up-vector
- Cross-product of two vectors provides vector in dir. perpendicular to two original vectors




- Changing *roll* = rotate camera's up-vector about the viewing direction (*twisting* the camera)

# Camera Manipulation

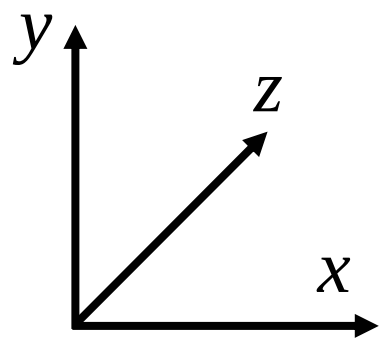
- Changing *yaw* = rotating focal point about the up-vector
- Changing *pitch* = rotating focal point about cross product of view vector and up vector
- *Dollying* – moves camera position along view vector (dollying in and out)
- Once camera attributes are set, objects are *projected* from 3D onto the 2D image plane
- Camera attributes determine which rays of light (that bounced off objects) will enter the camera and contribute to the rendered image

# Coordinate Systems

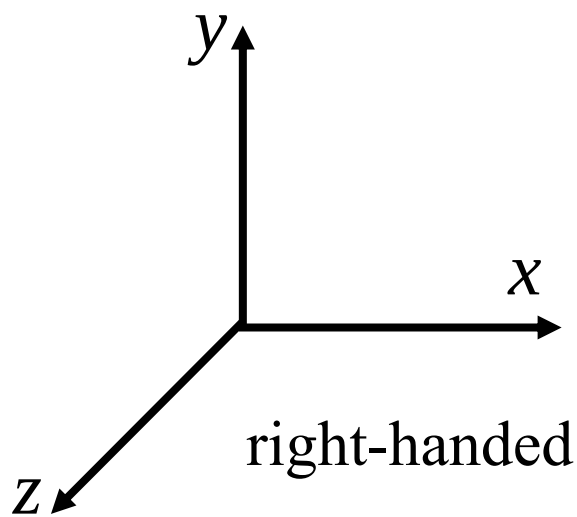
- You might be familiar with different types of coordinate systems:
  - Cartesian
  - Polar
  - Spherical
  - Cylindrical
- Computer graphics and visualization applications use several distinct coordinate systems: *model*, *world*, *view* and *display*
- Usually they use Cartesian coordinates

# Coordinate Systems

- Two kinds of Cartesian coordinate systems: right-handed and left-handed
- Use whichever coordinate system seems most natural in the given context




left-handed system




right-handed system

# Model Coordinate System

- Coordinate system used to define an object or actor
- Coordinate system will be a natural choice
  - Example: A football might be described using a cylindrical coordinate system
  - What coordinate system might we use for a planet?
- System choice of person who created the object
- Units are application-dependent: inches, meters, cubits, etc.

# World Coordinate System

- 3D space in which our actors are positioned
- Each actor's model coordinate system has some position and orientation inside the world space
- Many model coordinate systems, only one world coordinate system
- Each actor rotates, scales and translates itself into the world coordinate system
- Lights and cameras are specified with respect to the world coordinate system
- Does a camera have its own coordinate system?

# World Coordinate System

- Example:
  - Specify each of our bodies with a cylindrical coordinate system with the head as the origin
  - We position ourselves in the room (the world coordinate system) by giving the position of our heads w.r.t. the origin of the room (perhaps some corner)

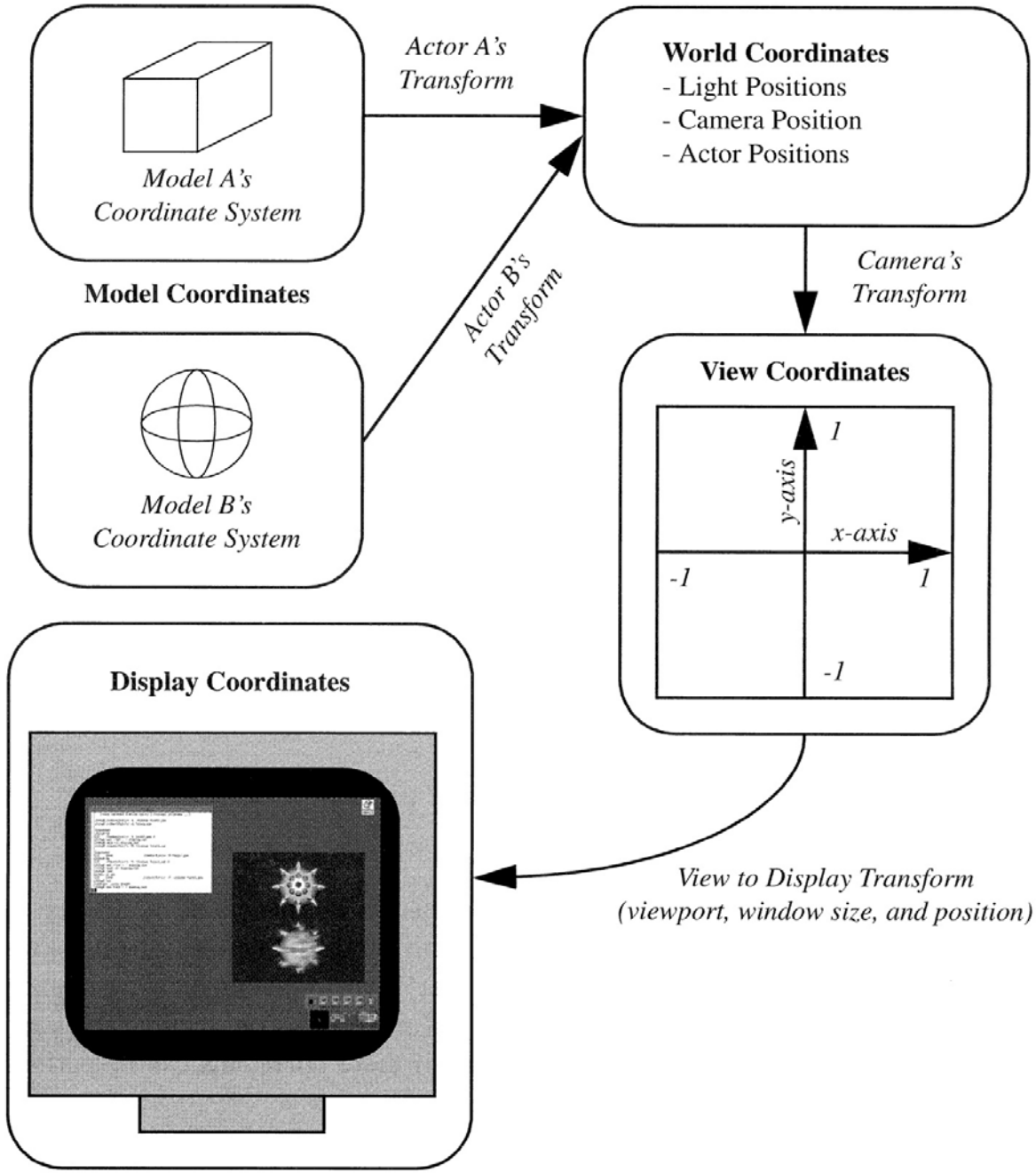
# View Coordinate System

- Represents what is visible to the camera
- Given by (x,y,z) values
- x, y in [-1, 1]
- z is some depth > 0
- x, y give location of some object in the image plane
- z give distance of object from camera
- A matrix is used to convert from world coordinates into view coordinates (i.e., projection!)
- Perspective effect can be accommodated by this matrix

# Display Coordinate System

- x, y are pixel values on screen
- z is still the depth
- What are restrictions on x and y?
- Window size helps determine valid range for x, y
- Display can be divided into multiple *viewports*, each of which has its own coordinate system
- Must select which viewport is used for rendering

# Coordinate Systems



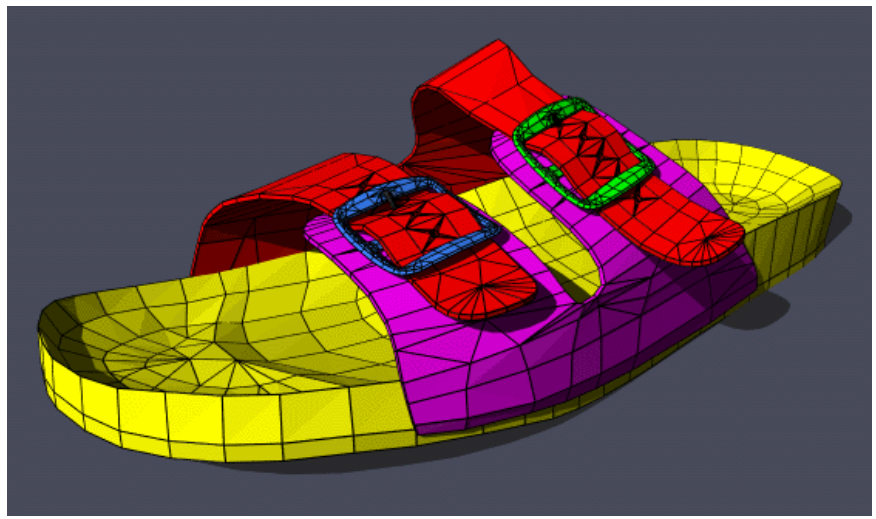
# Coordinate Systems

1. Model coordinates are  transformed into
2. World coordinates, which are transformed into
3. View coordinates, which are transformed into
4. Display coordinates, which correspond to pixel positions on the screen

- Transformations from one coordinate system to another take place via *coordinate transformations*, which we'll look at now

# Coordinate Transformations

- *Coordinate transformations* allow us to *translate*, *scale* and *rotate* our models in our virtual scene
- In Computer Graphics and Visualization, objects are often represented as meshes consisting of polygons, edges and vertices
- Two vertices define an edge
- Three or more edges define a polygon
- To transform an object, we apply the transformations to the vertices of the mesh



# Object Representations

- List of vertices: $v_1, v_2, \ldots, v_n$, each given as $(x_i, y_i, z_i)$
- List of edges: $(v_1, v_3), (v_4, v_7), \ldots, (v_i, v_j), \ldots$
- List of faces: $(e_1, e_3, e_4), (e_2, e_5, e_8), \ldots$OR
- List of faces: $(v_1, v_3, v_5), (v_6, v_7, v_9), \ldots$
- When a vertex's position is changed due to transformation, all edges and polygons that include the vertex are consequently changed
- If we apply the same transformations to all vertices, the entire polygonal mesh moves as a unit, which is what we want

# Coordinate Transformations

- Rather than represent 3D points using three coordinates (x,y,z), we will use four: (x,y,z,w)
- This approach is called *homogeneous coordinates*
- Transformations will be represented by (4 x 4) matrices
- Why not (3 x 3)?
- Because some transformations – including translation – cannot be represented by (3 x 3) matrices
- Most of the time w = 1, but there are special transformations for which w ≠ 1

# Coordinate Transformations: Translation

- Suppose we wish to translate the point (x,y,z) by the vector ($t_x$, $t_y$, $t_z$)
- This *translation transformation* can be described by the *translation matrix:*

$$T_T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Coordinate Transformations: Translation

- The new position is given by post-multiplying our point by the translation matrix:

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

- The new position of our point is (x', y', z')

# Coordinate Transformations: Translation

- We can see that the matrix-vector multiplication is equivalent to the following formulas:

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

$$x' = x + t_x$$

$$y' = y + t_y$$

$$z' = z + t_z$$

# Coordinate Transformations: Scaling

- We can scale a mesh by applying the *scaling transformation* to each of its vertices:

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

# Coordinate Transformations: Scaling

- When $s_x = s_y = s_z$, we call it *uniform scaling*
- Otherwise, we have *non-uniform scaling*

- Suppose someone said to you that it makes no sense to apply scaling to vertices
- After all, how do you scale a 3D point, which has no width, height or depth?

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

# Coordinate Transformations: Rotation

- We can rotate a vertex about one of the major axes by some angle θ using one of the *rotation matrices*:

$$T_{R_x} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad T_{R_y} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

# Coordinate Transformations

- Transformations can be *composed* by *right-multiplying* transformation matrices
- Example: a sequence ($S\ R_z\ T\ R_y$) would indicate:
  1. A rotation about the Y axis, followed by
  2. A translation, followed by
  3. A rotation about the Z axis, followed by
  4. A scaling
- So beware and remember: matrix multiplication is associative but it isn't commutative

# Coordinate Transformations

- The above transformations can be applied to objects in the scene – these are referred to as the *modeling transformations*
- The camera (viewpoint) can also be transformed by the *viewing transformation*
- What transformation(s) might not make sense to apply to the viewpoint?
- *Projection transformation* is applied after modeling transformations to project the 3D actors onto the screen
- We won't study projection transformations in this course

# Actor Geometry: Modeling

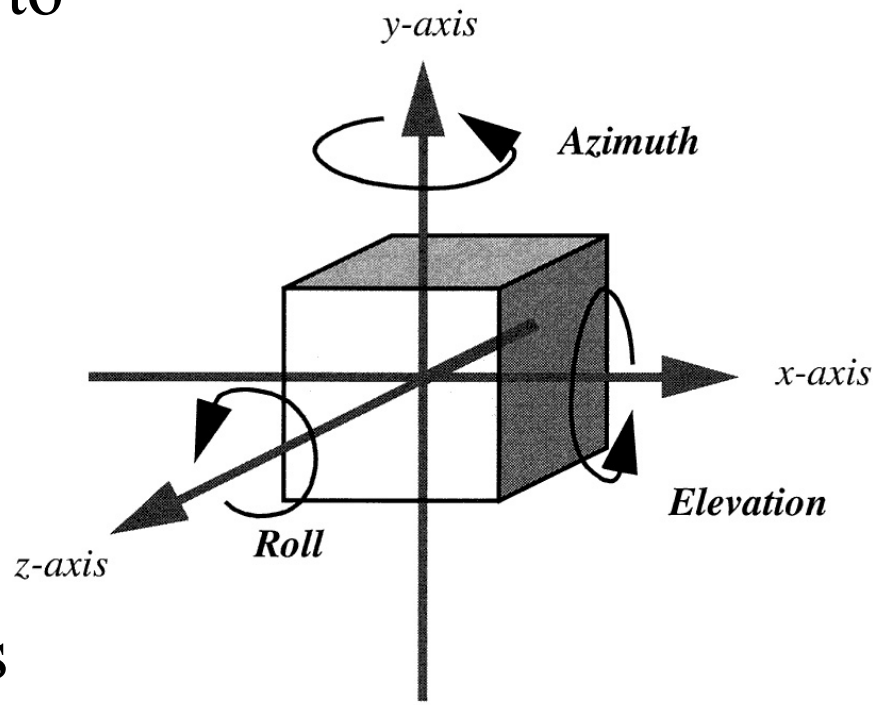
- In computer graphics, *modeling* refers to geometric representations of 3D objects
- Often these objects are manually constructed
- We looked at one type: polygonal meshes
- Many, many other representations exist
- Can you remember some? (consider some of the applications of visualization)
- In visualization, modeling means something slightly different

# Actor Geometry: Modeling

- In visualization, models are computed by some visualization algorithm
- Note the semantic distinction:
  - Computer graphics: object X is represented as a collection of triangles
  - Visualization: object X represents the surface of patient Y's skull and it just happens to be made of triangles
- The model (triangles) is simple, but complex visualization algorithms were used to obtain that model

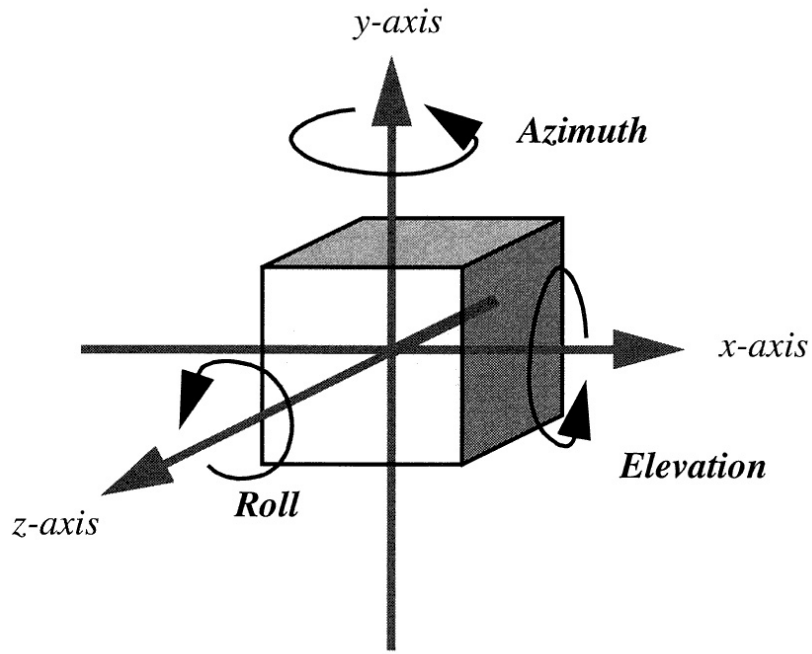
# Actor Geometry: Actor Location and Orientation

- The modeling transformations we looked at earlier allow us to change the location and orientation of objects
- It's often useful to associate (i.e., store) an *orientation vector* ($O_x$, $O_y$, $O_z$) for each actor
- This vector implicitly defines the three rotation matrices



# Actor Geometry: Actor Location and Orientation

- Rotations take place around the origin of the actor
- They are applied as a camera azimuth, elevation and roll, *in that order* – remember, order counts!
- VTK uses this orientation vector-based approach since it is very natural to manipulate objects in this fashion



# Camera Attributes

- *Projection* – method of projection determines how 3D objects are drawn on the *image plane*, or screen
- *Orthographic projection* – all rays of light are parallel to the projection vector
- 3D points are projected onto the screen along the same direction
- The perceived size of an object is not a function of its distance from the camera

# Camera Attributes

- *Perspective projection* – all light rays travel through a central point, such as the viewpoint
- Objects appear smaller as their distances increase from the viewpoint, and vice versa
- This is what happens in real life
- Simulating perspective projection requires a *view angle*
- View angle and clipping planes define a *view frustum*, a truncated pyramid; one type of *viewing volume*
- In orthographic projection, we have a rectangular view volume instead because the light rays are _____

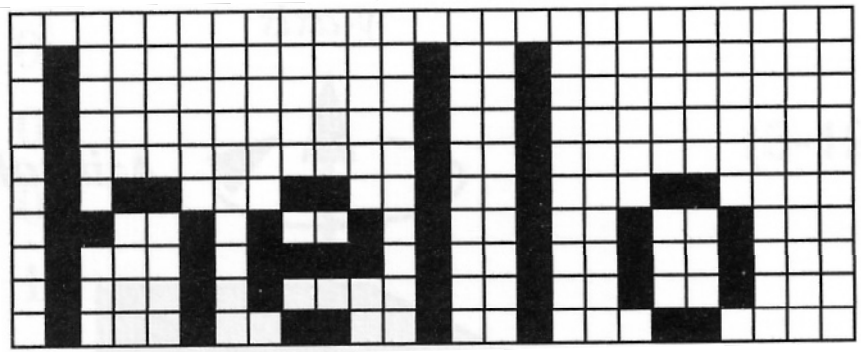
# Graphics Hardware and Display Devices

# Graphics Hardware

- Many graphics algorithms can be implemented efficiently and inexpensively in hardware
- Permits interactive graphics applications, including certain domains of visualization
- Topics today:
  - Raster devices
  - Video controllers & raster-scan display processors
  - Important rasterization and rendering algorithms
  - Pixels and images

# Raster Devices

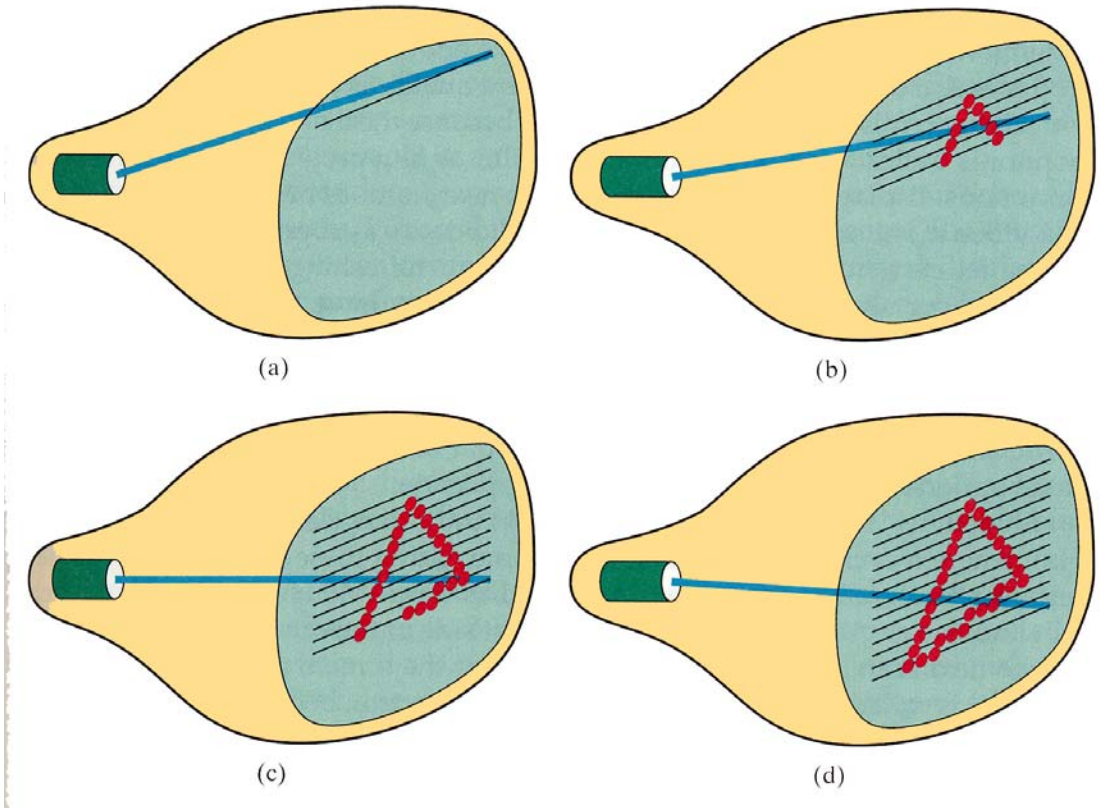
- Computer monitors (CRT, LCD, etc.), TVs
- These are *raster devices* because they display images on a *raster*, which is a regular n-D grid
- Each point on the grid is called a *pixel*, which stands for _____
- Raster dimension given in pixels: 25 x 10 in the example



- In a monochrome display, each pixel is black or white
- In a color display, each pixel has an RGB triple

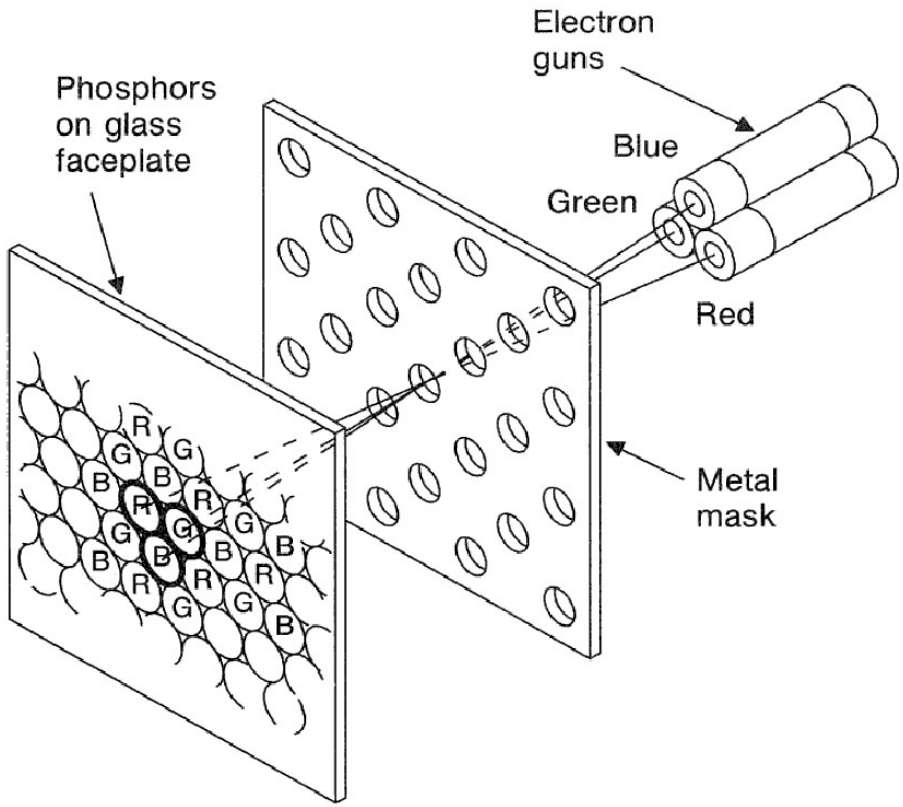
# Raster Devices

- Also called *raster-scan* displays or systems
- Pixels are drawn in a strict order, called *raster-scan order*
- Cathode ray tube (CRT) shown here
- Monochrome display



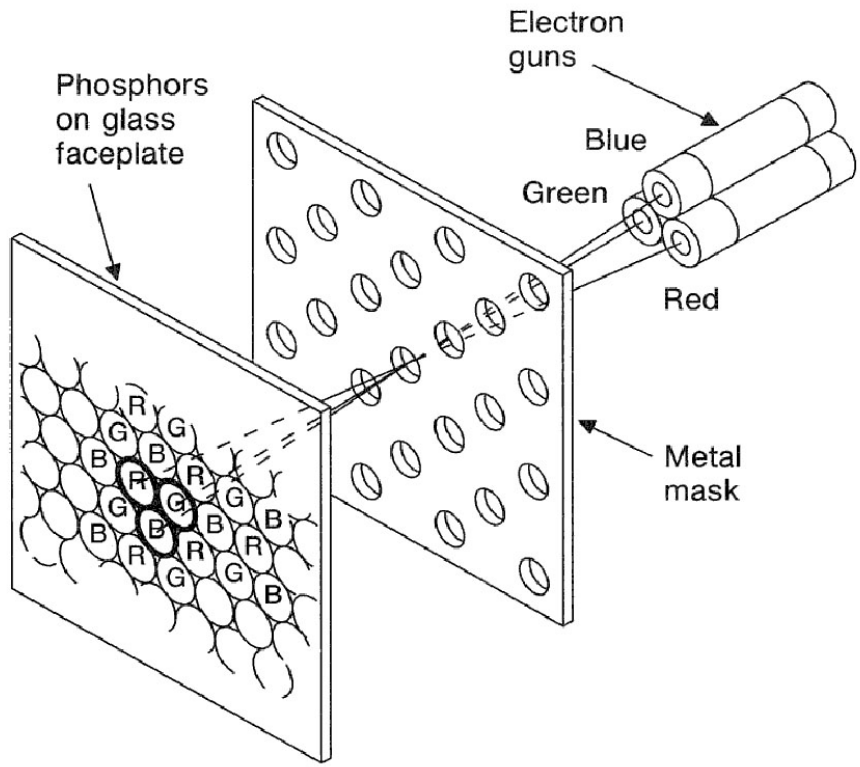
# Color Display Technology – CRT

- Cathode ray tube - used in TVs and computer monitors (the large, clunky type)
- A color CRT has three electron guns: one for red, one for green, and one for blue
- The beams scan screen in horizontal scanlines
- Metal mask steers beams



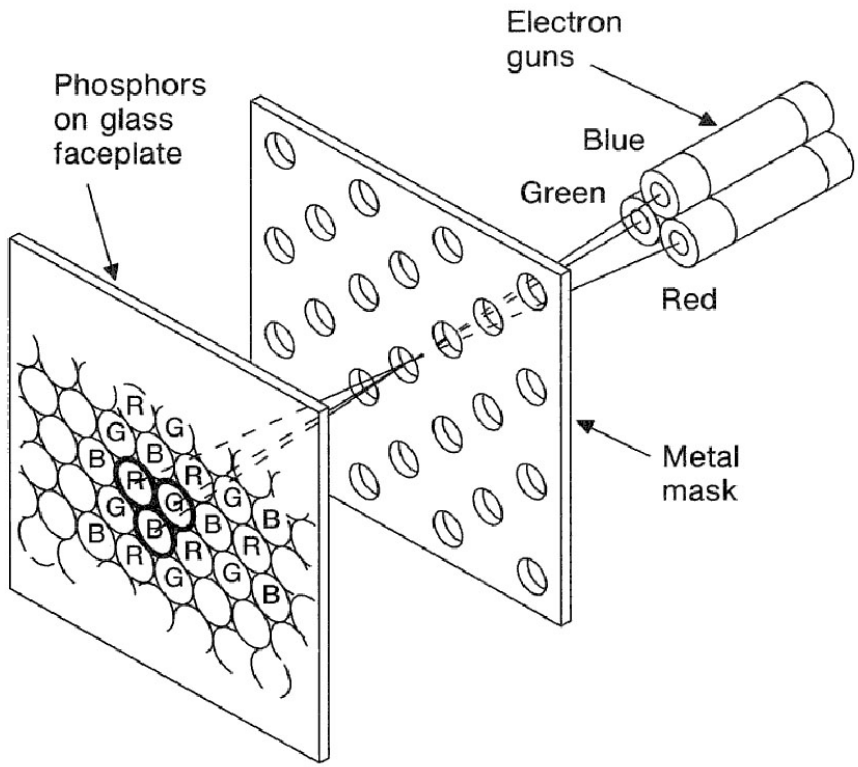
# Color Display Technology – CRT

- Each screen pixel consists of a *phosphor* triple: one glowing red, one green, and one blue
- A phosphor is a circular spot of *phosphorescent* material that glows when electrons strike it
- Red phosphors glow red
- RGB triad together form a single pixel on screen



# Color Display Technology – CRT

- Glowing phosphor triples blend together to form color encoded in RGB triple
- Amount of energy that electron guns deliver to each phosphor depends on RGB value of image pixel displayed there
- RGB values between 0 and 1 are mapped to voltages for the guns



# Color Display Technology – CRT

- True or false: A color image in a CRT is generated by blending the three colored beams of light that are fired from the back of the monitor and blended on the front surface of the screen.

# Color Display Technology – CRT

- The phosphors glow only for about 10-60 microseconds
- Image refreshed 30-60 times per second
- This rate is called the *refresh rate* and is given in Hz
- So if we redraw the image once every $1/60^{th}$ of a second, but the image lasts only a few millionths of a second, what about the gap?
- $1/60^{th}$ second is approximately 16667 microseconds
- (16667 - 10) microseconds = “long” delay between refreshes
- So why is there no visible flicker?

# Raster Devices: Display Resolution

- The raster is not 100% perfect –points of light corresponding to pixels can overlap slightly
- Same is true of raster printing technologies, like laser and injket printers
- Pixels are more like circles than squares in reality
- Raster devices also limited by resolution
  - Computer monitors 1600 x 1200 and higher
  - Laser printers 300 dpi, 600 dpi, 1200 dpi and higher
  - TV resolution? HDTV?

# Raster Devices: Color Depth

- Horizontal lines of pixels are called *scanlines*
- TV: 640  HDTV: 720 or 1080
- Monochrome monitor has 1 bits per pixel (bpp)
- Grayscale has 8 bpp (usually)
- Color monitors most often have 24 bpp: 8 bits each for red, green and blue color channels
- How many different levels of gray can we represent with 8 bits per pixel?
- How many different colors can 24-bit color represent?

# Image Resolution

**res = $300^2$ pixels** **res = $150^2$ pixels** **res = $75^2$ pixels** **res =$37^2$ pixels**

- Image resolution very important in visualization
- Why?
- When might we want to use a low resolution image?

# How Many Bits Do We Need?

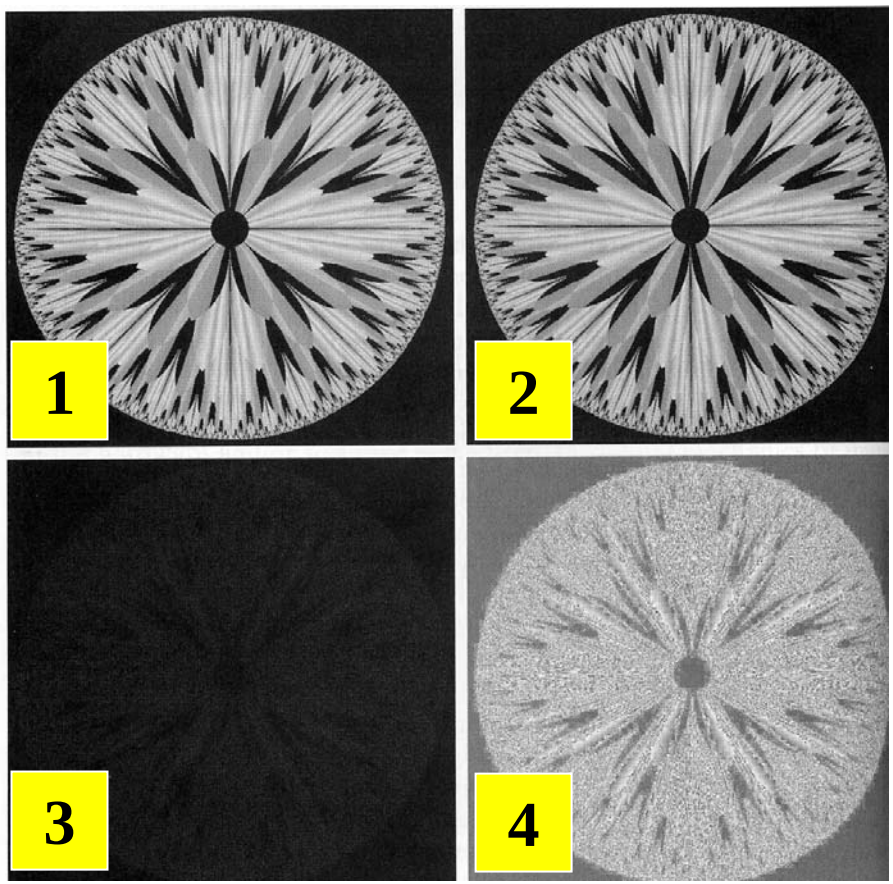
- Number of bits per pixel often called *bit depth*
- How many bits should we use in practice?

# 1: 8-bit original image

# 2: lower 4 bits dropped

# 3: (image #1 - image #2)

# 4: image #3 enhanced

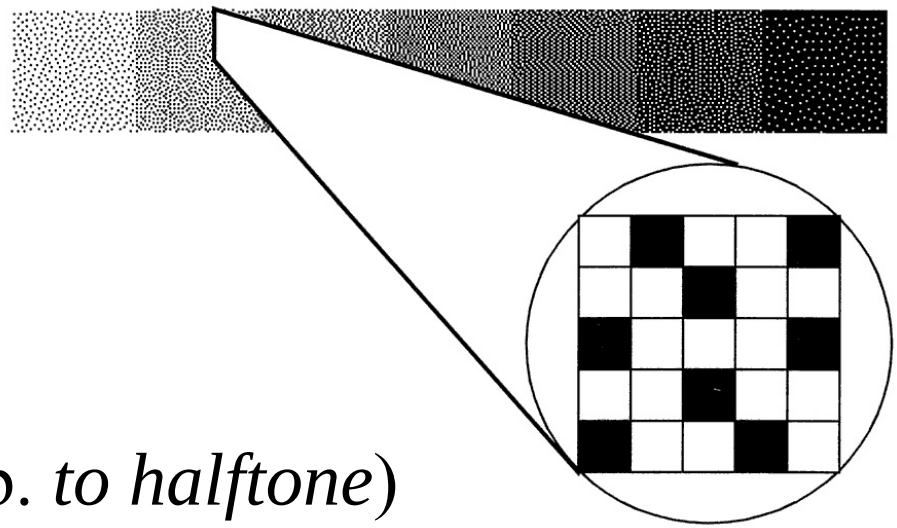


# Bit Depth

- Suppose we want to display 256 gray levels, but we have only 1-bit color.
- What colors *can* we display?
- How do we accommodate grayscale images?
- How do we accommodate color images?
- Suppose we want to display 16.7 million colors on our color monitor, but we have only 8-bit color. What can we do?

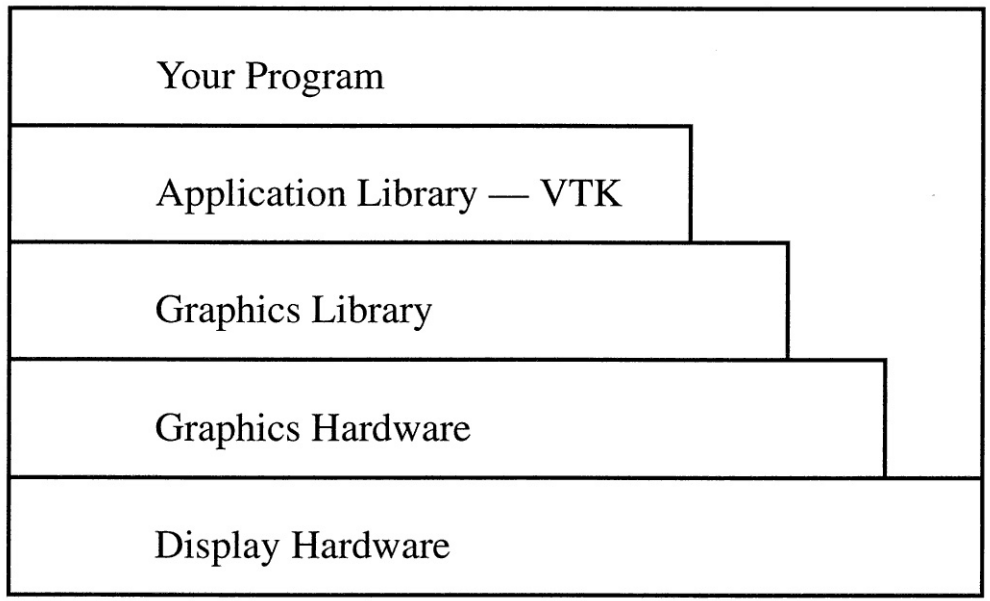
# Dithering

- *Dithering* is a way to use a mixture of colors to trick eye into seeing colors that cannot be actually represented by display device
- We can approximate gray by using a combination of black and white:
- The relative densities of black and white determine the “gray” value
- Also called *halftoning* (vb. *to halftone*)



# Interfacing to the Hardware

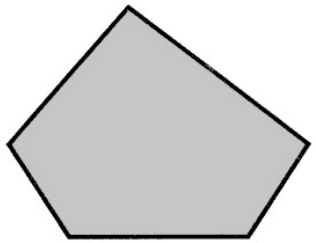
- A lot goes on "under the hood" in the graphics and display hardware
- Graphics hardware: converts geometry into pixels
- Display hardware: displays pixels
- Simplified hierarchy



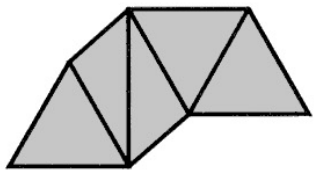
# Interfacing to the Hardware

- From perspective of visualization, mechanics of image display aren't too important
- We are more interested in what software can deliver
- Not even really interested in computer graphics!
- We just want to *visualize*!
- Why we use VTK and similar programming libraries
- We can treat everything under VTK as some nebulous "black box" that converts our 3D shapes into pixels
- Our building blocks are called *graphics primitives*

# Graphics Primitives



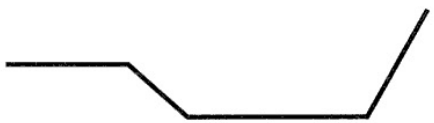
**Polygon** — a set of edges, usually in a plane, that define a closed region. Triangles and rectangles are examples of polygons.



**Triangle Strip** — a series of triangles where each triangle shares its edges with its neighbors.



**Line** — connects two points.

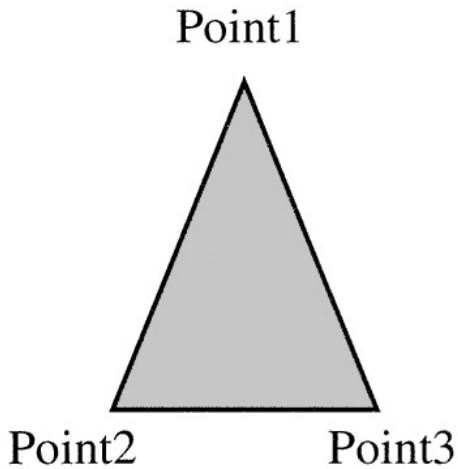


**Polyline** — a series of connected lines.

**Point** — a 3D position in space.

# Graphics Primitives

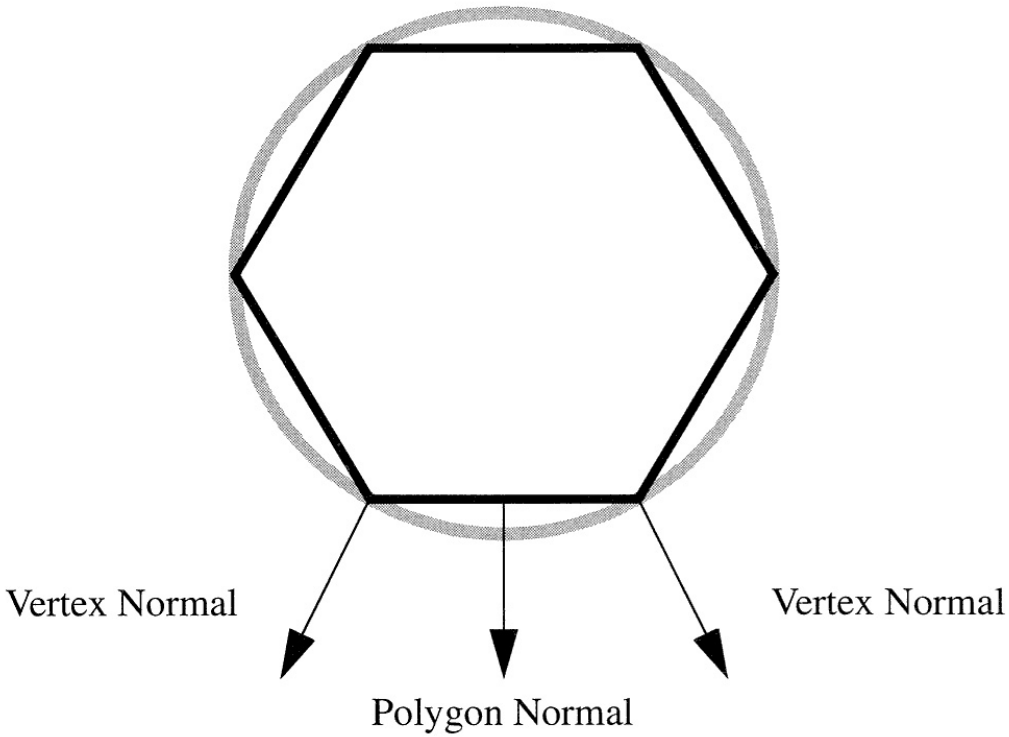
- Vertex: position, normal, color – how many values total?
- Polygon: series of connected vertices




```
Point1
  position=(1,3,0)
  normal=  (0,0,1)
  color=   (.8,.8,.8)
Point2
  position=(0,0,0)
  normal=  (0,0,1)
  color=   (.8,.8,.8)
Point3
  position=(2,0,0)
  normal=  (0,0,1)
  color=   (.8,.8,.8)

Polygon1
  points= (1,2,3)
```

# Graphics Primitives

- Normal vectors: why for vertices?
- If our polygonal object came from curved surface, vertex normals will not be same as polygonal normals



# Rasterization

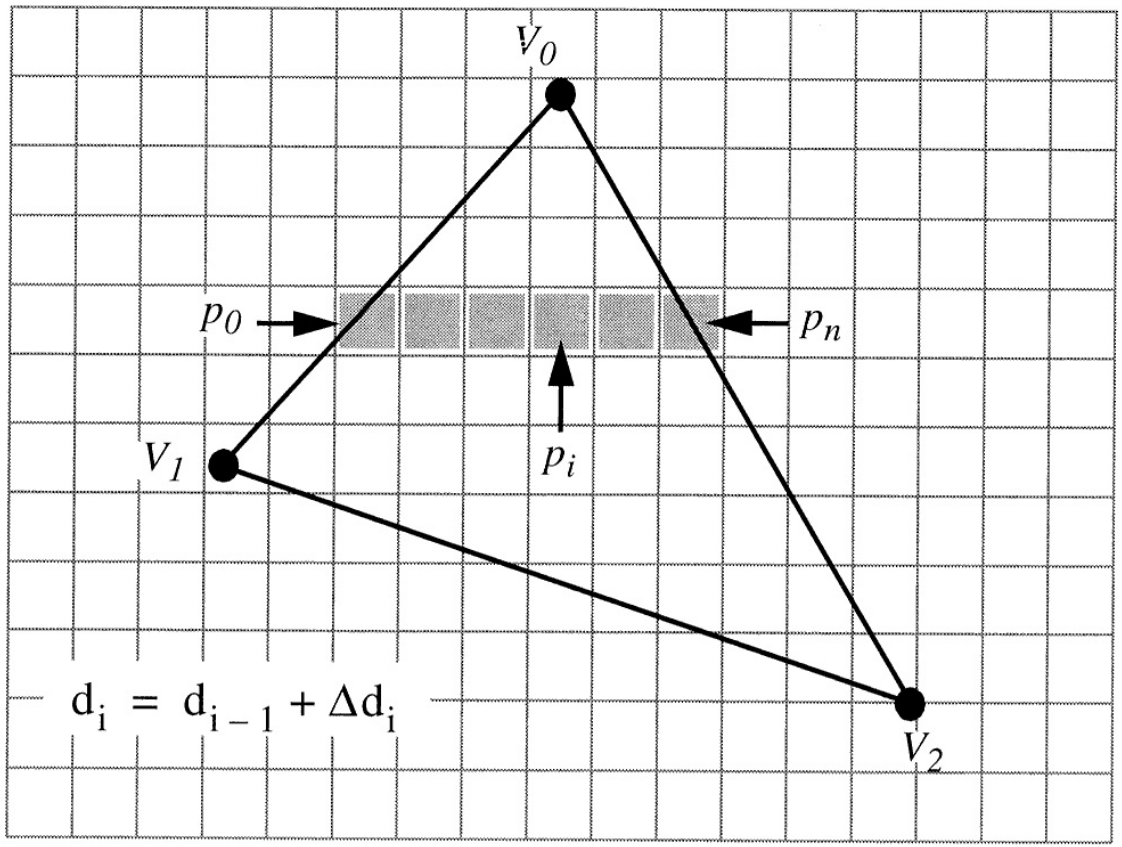
- We looked at raster devices and some different kinds of geometric objects we might wish to draw on the screen
- Process of converting geometry into pixels is called *rasterization* or *scan-conversion*
- Each triangle in our model is transformed (rotated, etc.) and projected by the transformation and projection matrices
- Next we *clip* each triangle to the image plane
- Each triangle is entirely inside, entirely outside, partially visible w.r.t the image plane

# Rasterization

- We will take an *object-order approach*
- Question: In contrast, ray-tracing is *what*-order?
- We process each triangle one by one
- After we transform and clip it, we rasterize it – we figure how what pixels on screen we need to update to draw the triangle on screen

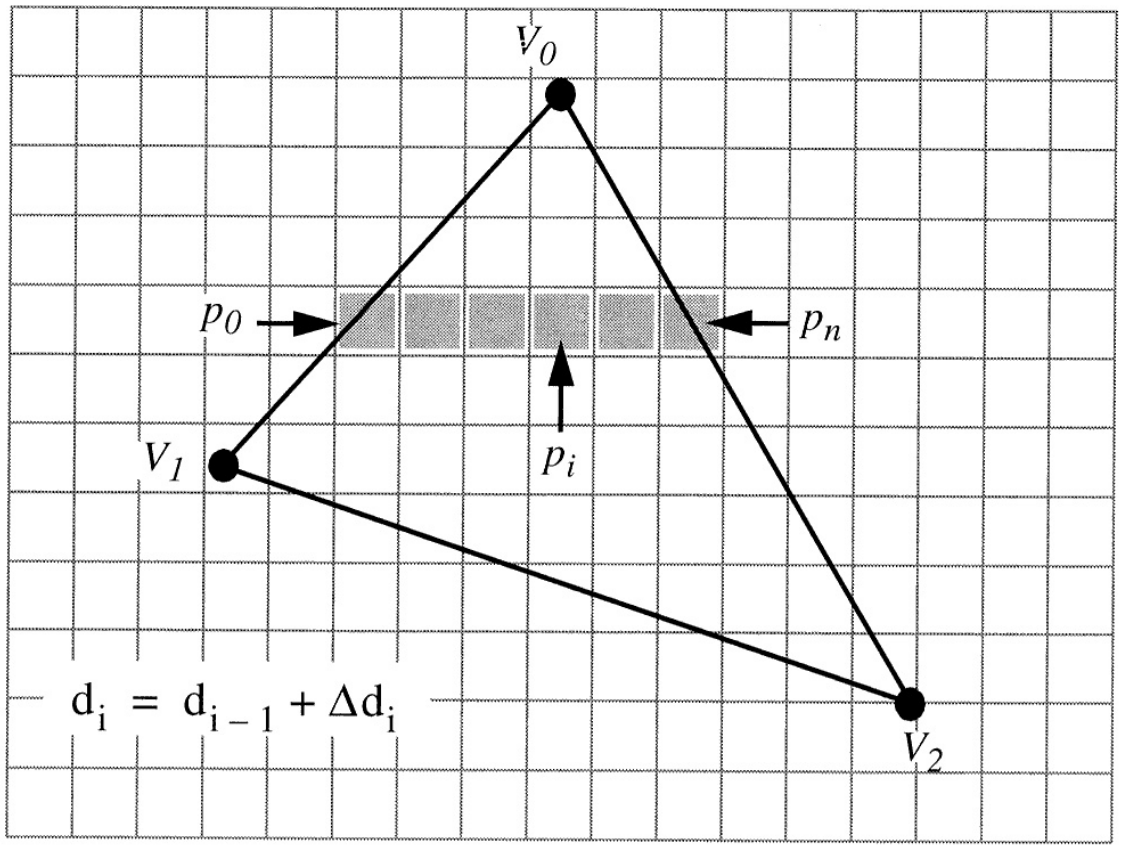
# Rasterization

- We will process the triangle in *scan-line order*: left-to-right starting at top left corner, moving right and down



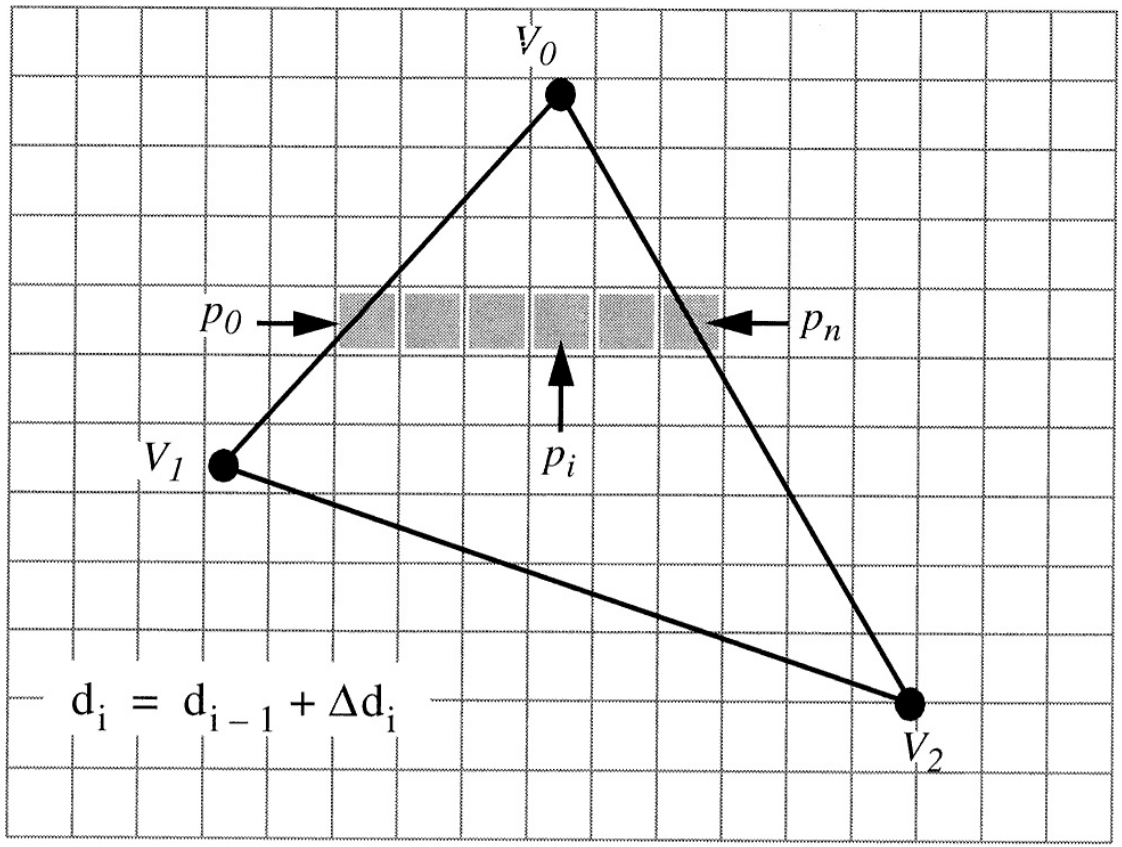
# Rasterization

- We sort the vertices by their *y* values and find the vertex with the maximal *y* value; call this vertex $v_0$



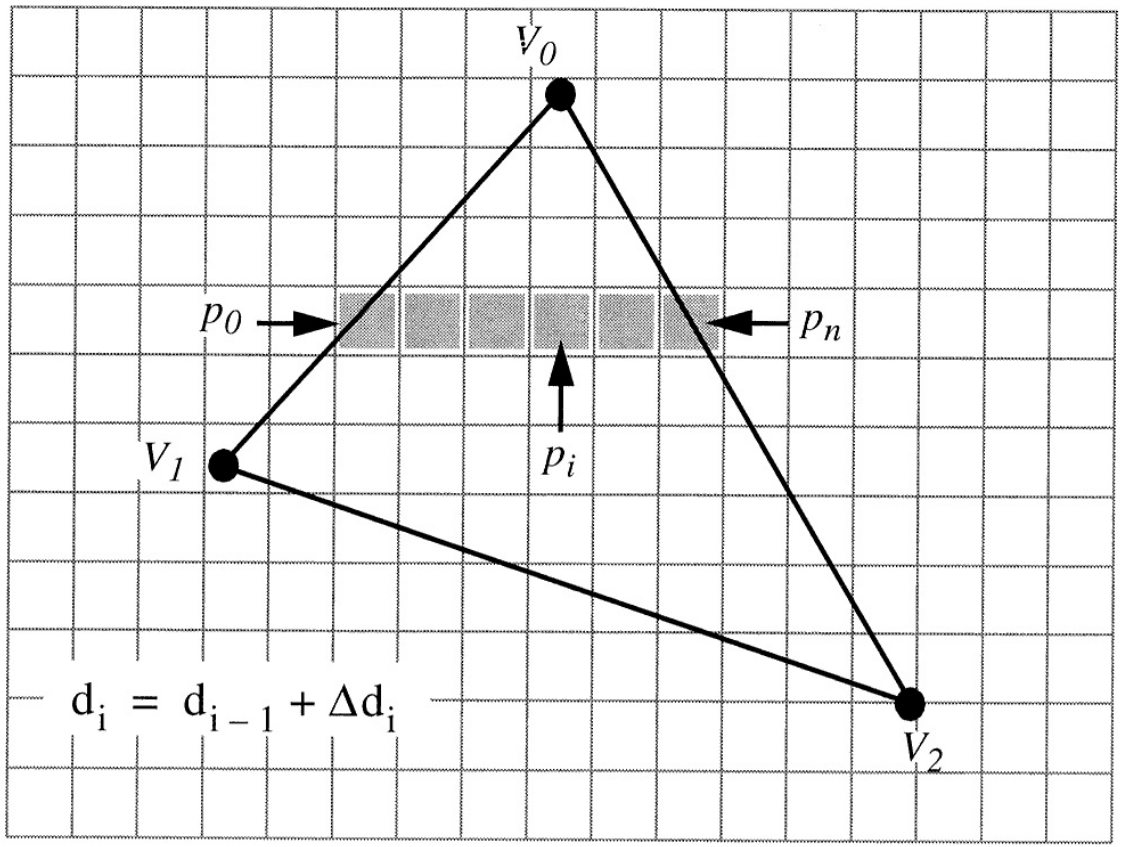
# Rasterization

- This sorting allows us to identify the other two vertices, $v_1$ and $v_2$



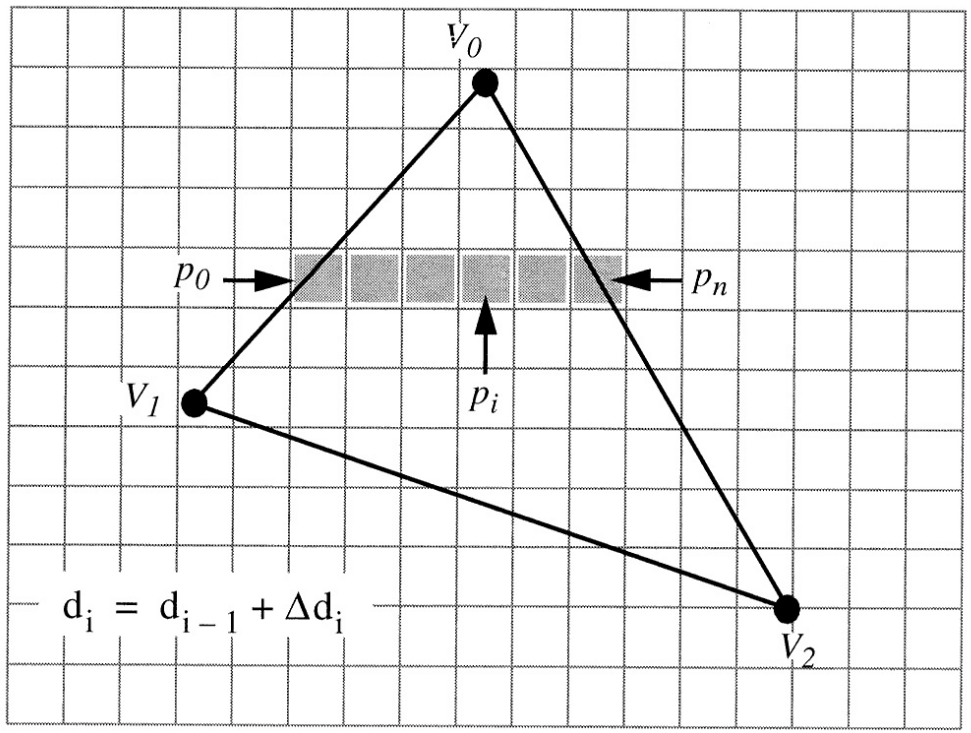
# Rasterization

- Using the slopes of the edges we can compute each row of pixels to process, called a *span* of pixels



# Rasterization

- Across each polygon we interpolate various data values $d_i$ for each pixel
- Example: RGB to assign colors to vertices

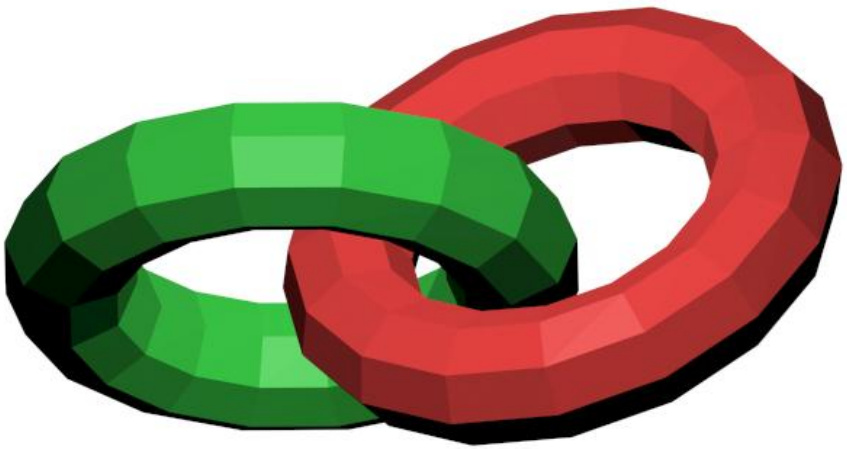


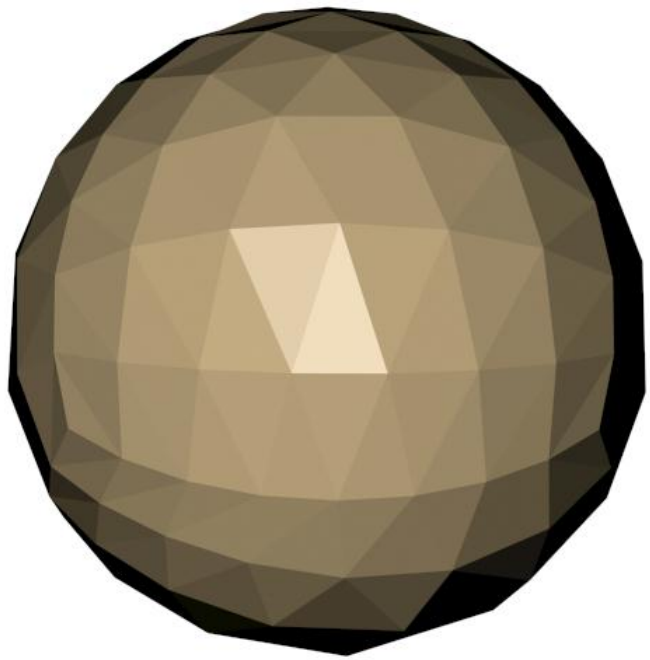
# Rasterization

- But where do we get the RGB values?
- A few classes ago we looked at shading and illumination
- Now we will see how the theory is put into practice
- We will look at three ways of implementing the illumination equations:
  - Flat surface rendering
  - Gouraud surface rendering
  - Phong surface rendering

# Flat Surface Rendering

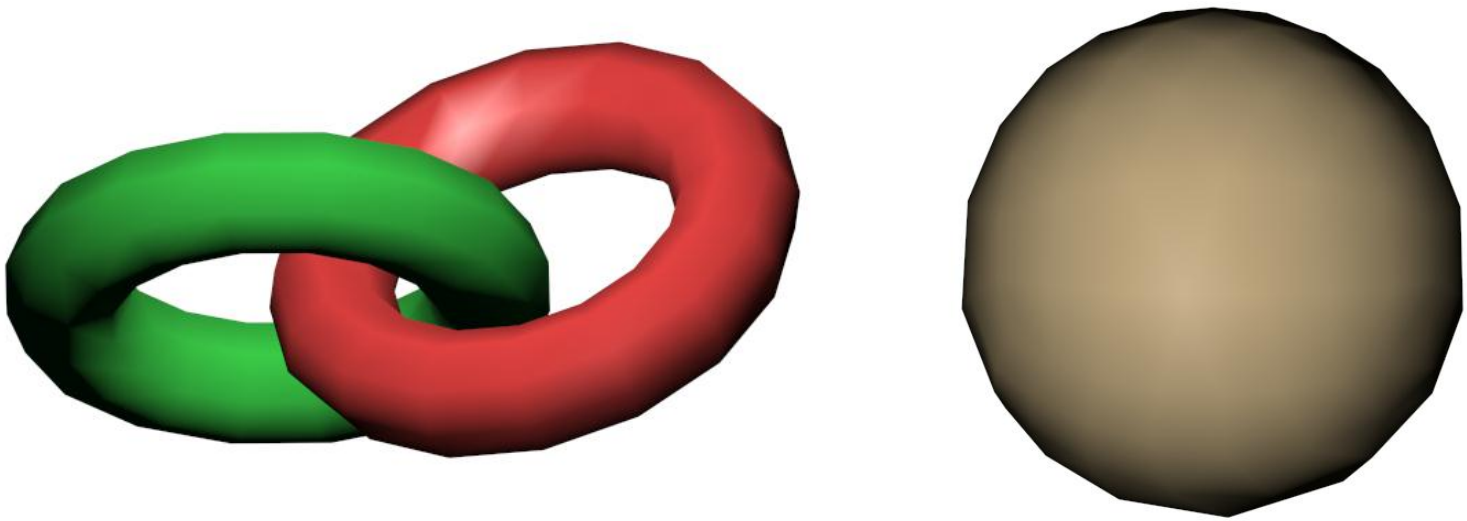
- Illumination equations applied to one normal vector of the polygon
- Result: all pixels for polygon have the same color





# Gouraud Surface Rendering

- Illumination equations calculated at all vertices of polygon using vertex normals
- Edges and interior of polygon colored by *interpolating* or smoothly blending the ***colors*** computed at vertices
- Result: color varies across the polygon

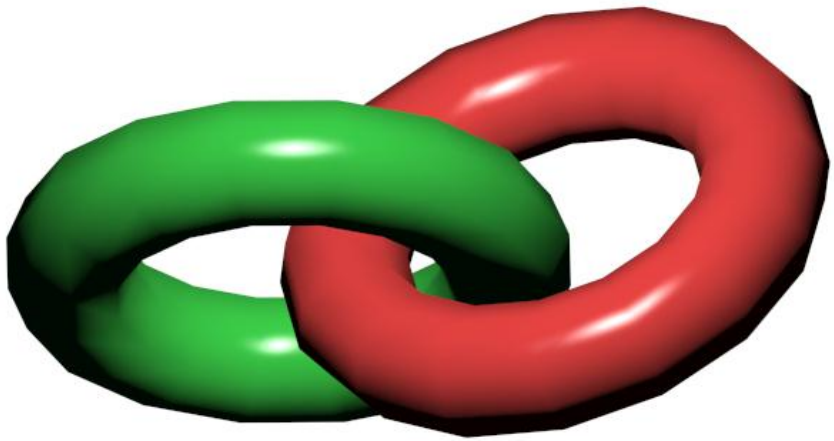


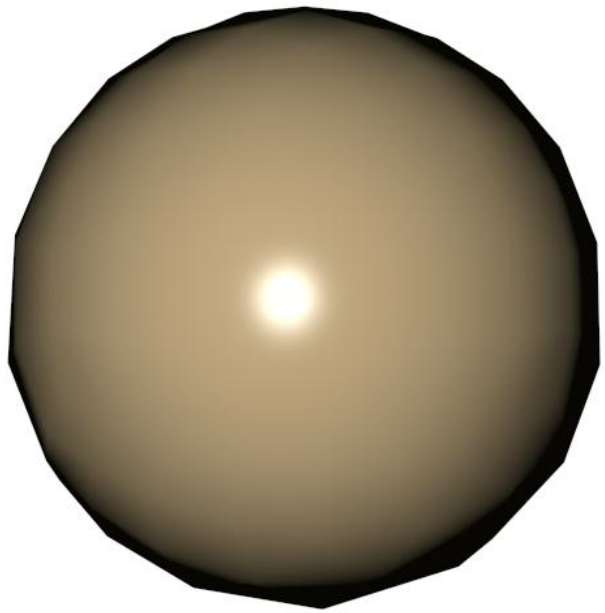
# Phong Surface Rendering

- **Normals** are first interpolated across edges
- Then interpolated across the polygon interiors
- Illumination equations are computed *for each pixel*
- Result: color varies across the polygon, plus we can generate specular highlights
- What do you think of the efficiency of Phong shading?

# Phong Surface Rendering

- Phong rendering just too expensive to use in real-time
- Software ray tracers use it, where speed is already slow



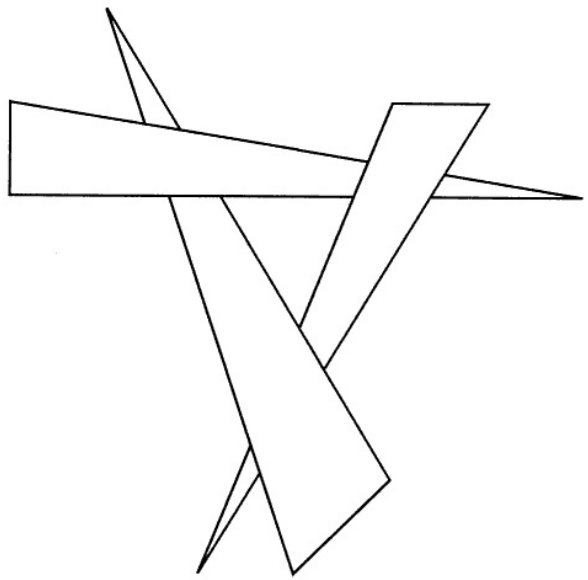


# Hidden Surface Removal

- We looked earlier at ray-casting
- We trace rays from the camera, through the images and into the scene
- We see whatever objects the rays strike
- Usually we don't use ray-casting and instead use the object-order approach we've been talking
- A complex scene could contain thousands or even millions of triangles that will overlap
- How do we know in which order to draw the triangles?

# Hidden Surface Removal: Painter's Algorithm

- One solution is called the *painter's algorithm*
- Sort the triangles
- Back-to-front or front-to-back?
- One major problem:



Can cut into smaller triangles, but the way we cut the triangles is *view-dependent*

What does that mean?

# Hidden Surface Removal: Z-Buffer Algorithm

- An easier and very efficient solution is the z-buffer algorithm
- We store a 2D array the same dimensions as the image
- Before we draw a pixel for a triangle, we compare its z value to what is stored in the z-buffer
- If the new pixel would be in front of the z-buffer's algorithm, we replace the current pixel with the new one
- Otherwise, we do not change the pixel
- How should we initialize the z-buffer?

# Visualization Toolkits: Overview

# VisualizationToolKits

- VTK is a C++ class library for developing visualization applications
- www.vtk.org → Manual 4.2 → Class Hierarchy
- Every non-trivial VTK program must contain the following seven elements:

1. vtkRenderWindow – the window on screen
2. vtkRenderer – C++ object for drawing shapes
3. vtkLight – light to illuminate scene
4. vtkCamera – camera (next class)

# VTK

5. vtkActor – an object in the scene
6. vtkProperty – set of properties for an actor (color, specular power, diffuse reflection coefficient, etc.)
7. vtkMapper – defines what is actually drawn on the screen for an actor

- If no light is specified, a default one is created
- Same goes for the camera

# Compiling Cone.cxx

- Download the **VTK Cone Example**
- Open the **VTK Setup Guide** on the home page
- Most of these steps have been performed for you
- Now open the source code and compile it
- May have to change some settings in Visual Studio .NET…

# Cone.cxx

- vtkConeSource *cone – represents a mathematical cone, and nothing more
- vtkPolyDataMapper *coneMapper – represents the cone as a set of triangles that the computer will render
- vtkActor *coneActor – the cone as VTK will deal with it; this actor can be moved around, its appearance changed, etc.
- vtkRenderer *ren1 – this C++ object will actually draw the cone

# Cone.cxx

- vtkRenderWindow *renWin – this is the window in which the renderer will draw the cone
- We can have multiple vtkRenderer's for a single vtkRenderWindow
- A **viewport** is given that tells the vtkRenderer at what position inside the vtkRenderWindow it should render its actors

# Events and Observers

- Open the file Cone2.cxx
- This program features a “callback function”
- This is a function that is invoked when a given **event** occurs
- In Cone2.cxx, the “event” is the drawing of the window

# Transformations

- Translation
- Rotation
- Scaling
- We will study these next class
- Translation just means you slide an object from one place to another
- Rotation can take place around X axis, Y axis, Z axis or an arbitrary axis
- Scaling means we increase/decrease the object's size

# Assemblies

- vtkAssembly lets us group shapes logically
- Example: robot arm: shoulder joint, upper arm, elbow, lower arm, wrist joint, hand
- If we rotate the arm at the shoulder, we expect all parts of the arm to rotate together
- If we bend the elbow, the lower arm, wrist and hand will rotate together with respect to the elbow

# Programs for You to Try

- expCos.cxx – Stavros
- Mace.cxx – Kostadin
- Model.cxx – Raymond
- Cone4.cxx – Rohit
- Cone5.cxx – Naval
- Follow the directions in the VTK Setup Guide, get your program to compile, zip it all up (delete **release** and **debug** folders first!), and it will count for credit
- I will post your zipfile on the Blackboard home page

# Data Visualization Pipeline in VTK

# The Visualization Pipeline

- Visualization: transformation of data into graphical form
- Object-Oriented-based approach: data are the objects, transformations are the methods

# Data Visualization Example

- A **quadric** is a special function with maximum degree 2:

$$F(x, y, z) = a_0x^2 + a_1y^2 + a_2z^2 + a_3xy + a_4yz + a_5xz + a_6x + a_7y + a_8z + a_9$$

- A **solid** sphere is an example of a quadric with $a_3$, $a_4$, $a_5$, $a_6$, a6, $a_7$ and $a_8$ all equal to zero
- If those values aren't zero, we get some pretty strange shapes
- Imagine squishing a **solid** rubber ball (i.e., not a hollow ball, like a tennis ball)

# Data Visualization Example

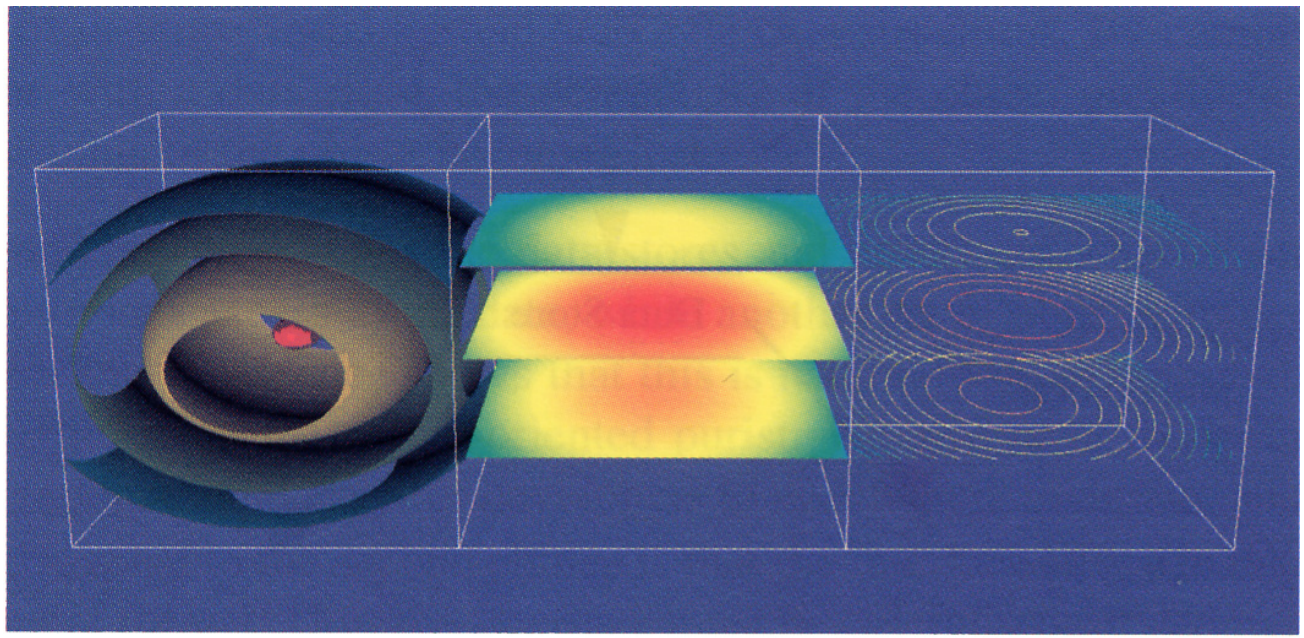
- Usually we **evaluate** the equation of a sphere for a particular radius, r: $x^2 + y^2 + z^2 = r^2$
- Suppose we evaluate it for different values of r?
- We get a solid sphere
- Now imagine we evaluate it for any value of *x*, *y*, *z* and *r*
- We get what's called a **field function**
- You plug in some values for *x*, *y*, *z*, *r* and get some number. That number is "located" at position (*x*, *y*, *z*)

$$F(x, y, z) = x^2 + y^2 + z^2 - r^2$$

# Data Visualization Example

$$F(x, y, z) = a_0x^2 + a_1y^2 + a_2z^2 + a_3xy + a_4yz + a_5xz + a_6x + a_7y + a_8z + a_9$$

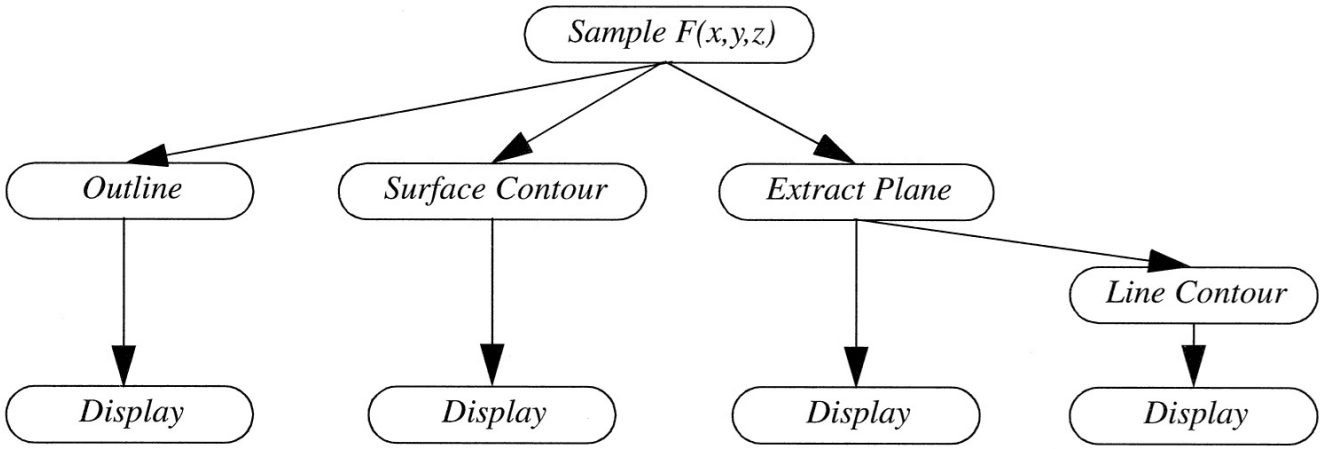
- If we plug in x, y, z, r for any quadric, we can get some very strange-looking **field functions**. Here's an example:



(a) Quadric visualization (`Sample.cxx`)

# Visualization Pipeline

- Depicts data flow through a visualization system

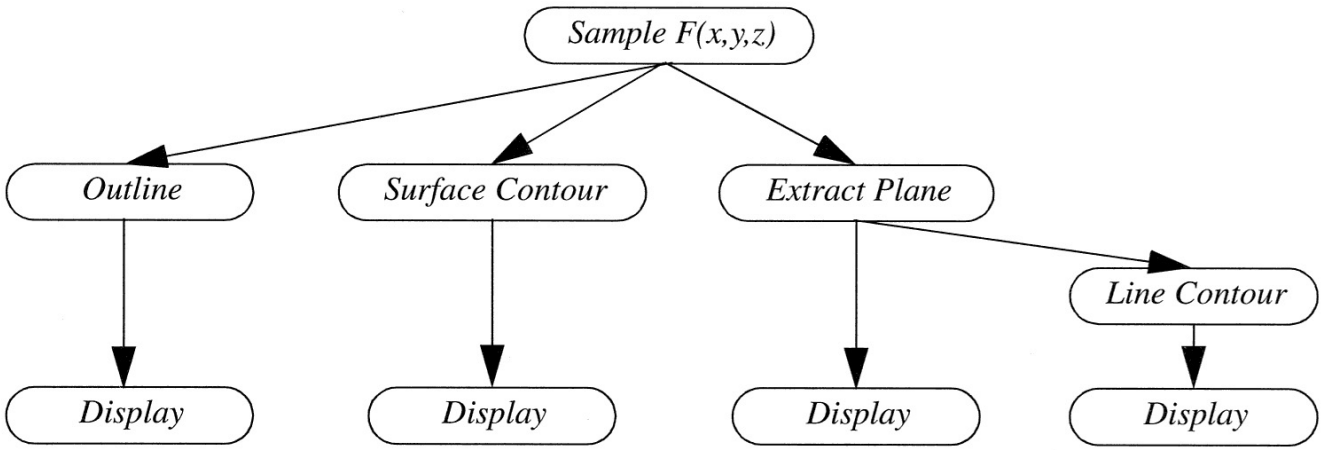



(c) Visualization network

- **Source** processes produce output (i.e., data)
- **Sink** processes consume data: no output
- **Filter** processes consume data and produce output

# Visualization Pipeline

- This figure depicts a particular **visualization pipeline**
- **Data objects** operated upon by **process objects**, as indicated by arrow directions (depicting the flow through the pipeline)




(c) Visualization network

# Data Objects

- Data objects represent information
- Also provide methods to create, access, delete this info
- Do not support modification of the data

# Process Objects

- Operate on input data to generate output data
- Derives new data from inputs, or transforms input into new form
- **Source objects** interface to external data sources or generate data from local parameters
- Former kind are called **reader objects**
- Latter kind are called **procedural objects**
- **Filter objects** take one ore more input objects and generate one or more output objects

# Process Objects

- **Mapper objects** are sinks, and terminate the visualization pipeline flow
- **Writer objects** are mapper objects that write data to disk

# VTK's Visualization Pipeline

- Strongly typed
- Demand-driven execution
  - Update() and Execute() methods
  - Update() called when rendering requested
  - Update() called recursively up network, until source object hit
  - Execute() method run if input has changed
  - Recursion unwinds as Execute() methods invoked in objects

# VTK's Visualization Pipeline

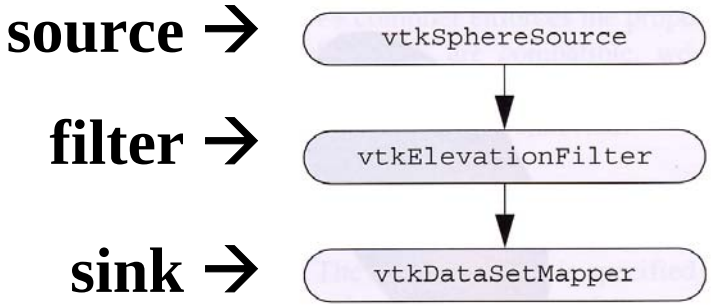
- Filters connect like thus:

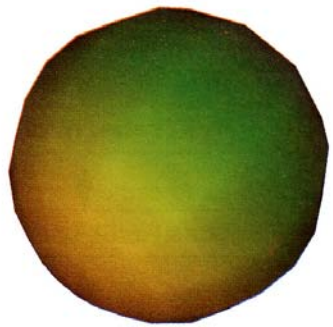
```
filter2→SetInput(filter1→GetOutput());
```

- Multiple output filter example: vtkExtractVectorComponents. Go to www.vtk.org
- Used to extract x, y and z components of a vector
- Map each component to a different geometric object of some kind
- Useful for vector visualization applications

# ColorSph.cxx

- Login to Blackboard and experiment with ColorSph.zip
- Note the visualization network on the left size:

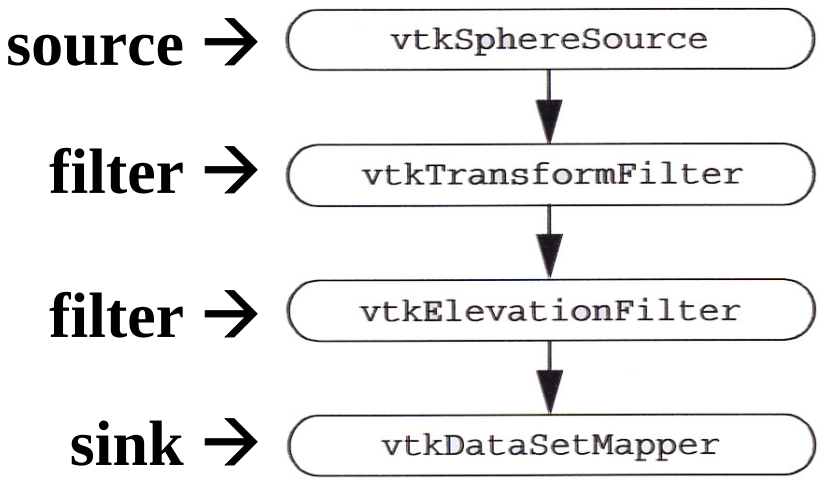


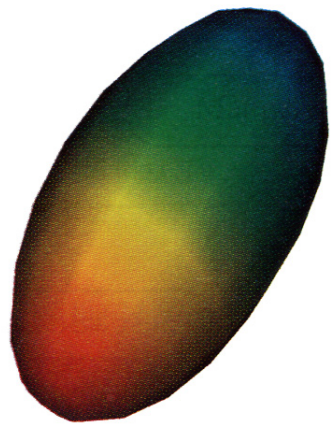



```
vtkSphereSource *sphere = vtkSphereSource::New();
  sphere->SetPhiResolution(12); sphere->SetThetaResolution(12);

vtkElevationFilter *colorIt = vtkElevationFilter::New();
  colorIt->SetInput(sphere->GetOutput());
  colorIt->SetLowPoint(0,0,-1);
  colorIt->SetHighPoint(0,0,1);

vtkDataSetMapper *mapper = vtkDataSetMapper::New();
  mapper->SetInput(colorIt->GetOutput());

vtkActor *actor = vtkActor::New();
  actor->SetMapper(mapper);
```

# StrSph.cxx

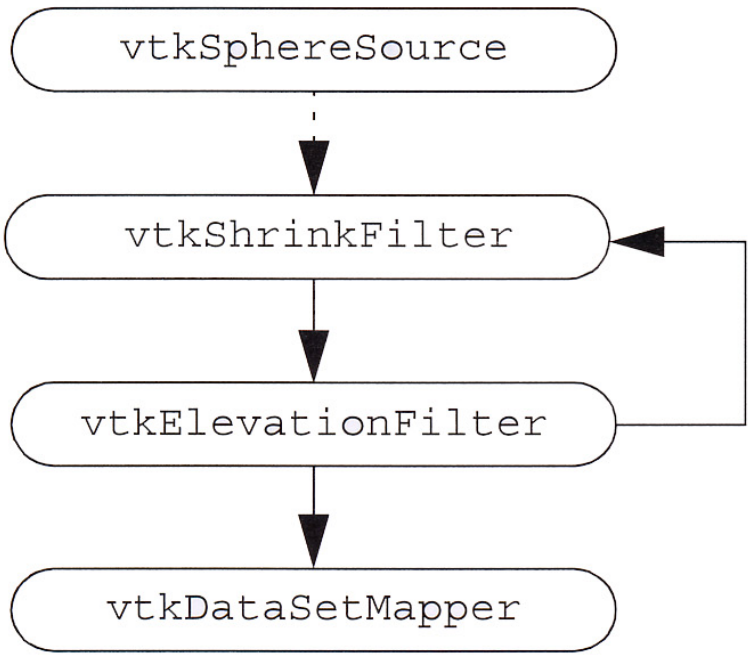
- Try this one next
- Play with the LUT: look-up table
- LUT is a kind of **transfer function**



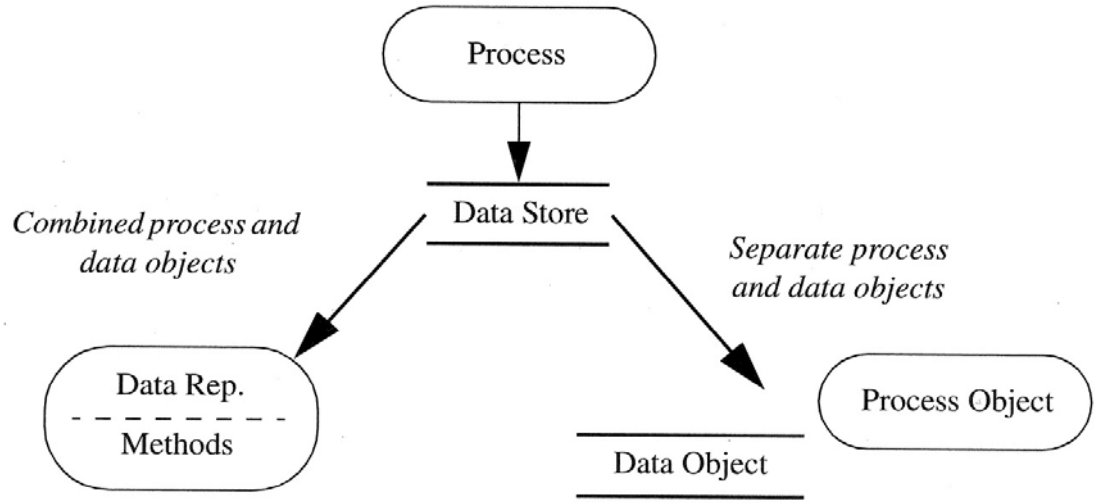



# LoopShrk.cxx

- The incredible disappearing sphere
- The feedback loop causes the shrinking filter to be applied each time the scene is rendered

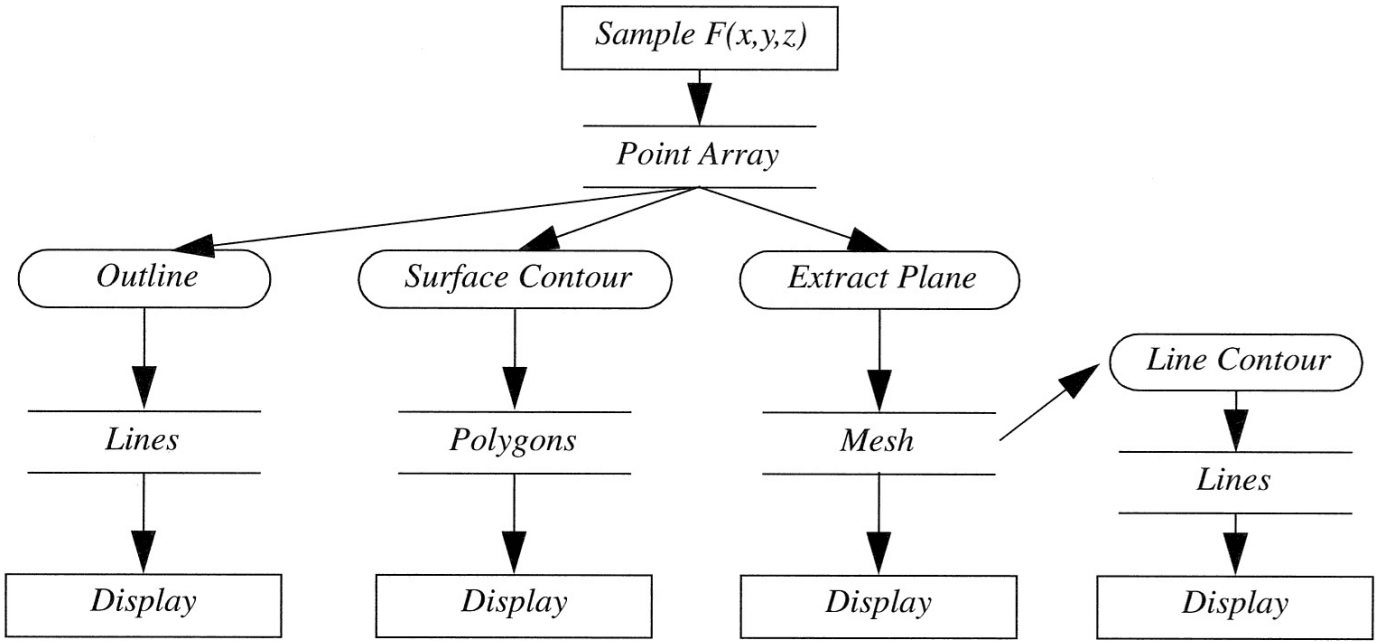



# The Object Model




1. Traditional OO approach: combine data and methods (processes)
2. Other option: separate data representations and processes
3. VTK: mostly like #2, with some small aspects of #1

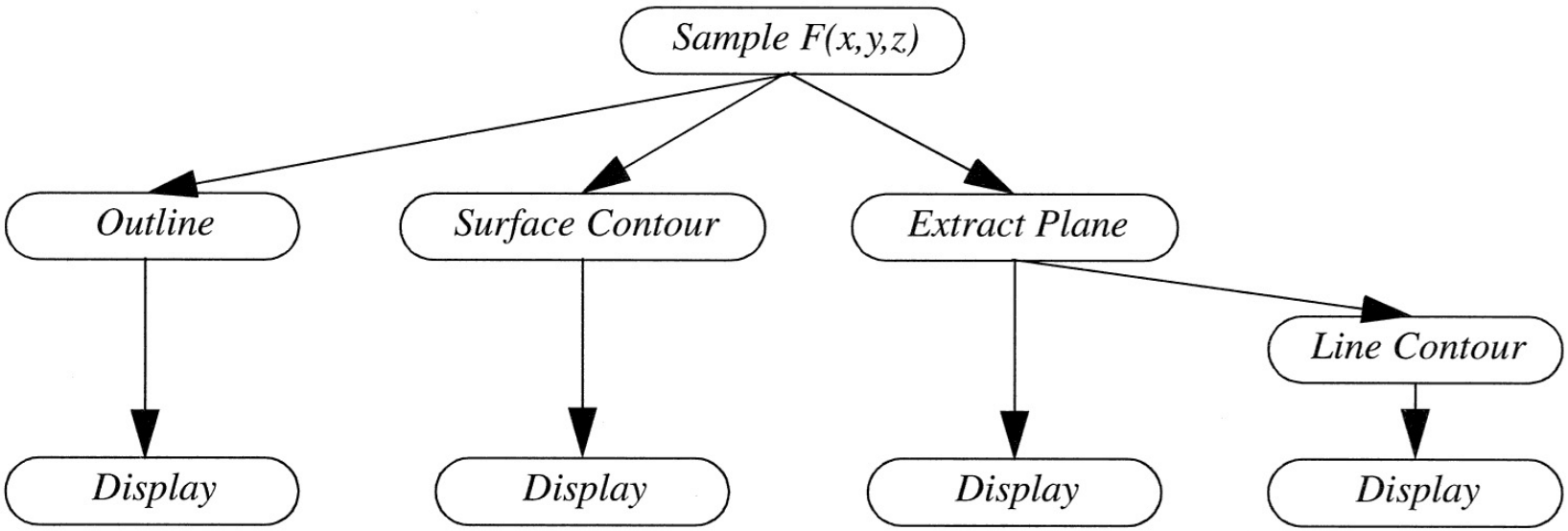
# The Functional Model: Example




(b) Functional model

# The Visualization Model

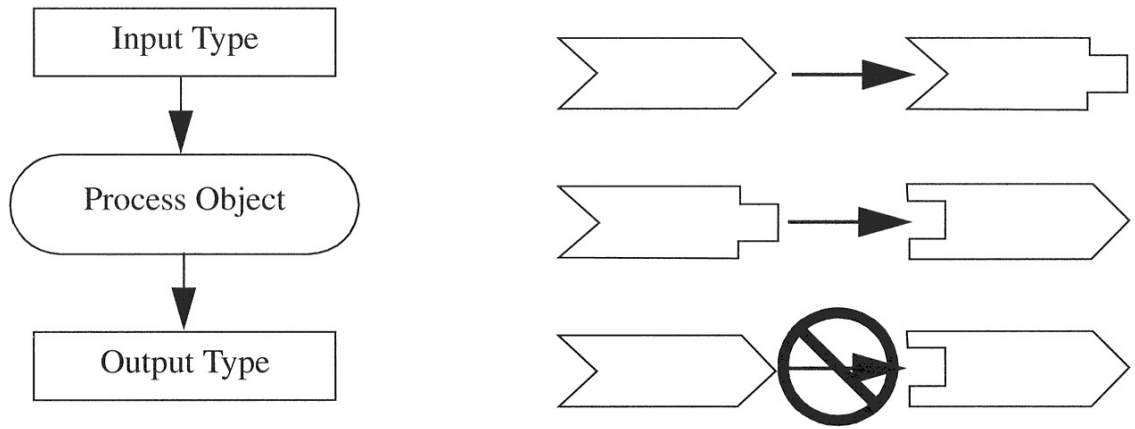
- Omits the graphical representations from the functional model




(c) Visualization network

# Pipeline Topology

- Sources, filter and mappers are **typed** objects
- Input and output have types that must be respected
- Example: sphere source object may generate polygons or some other representation as output
- VTK is strictly-typed

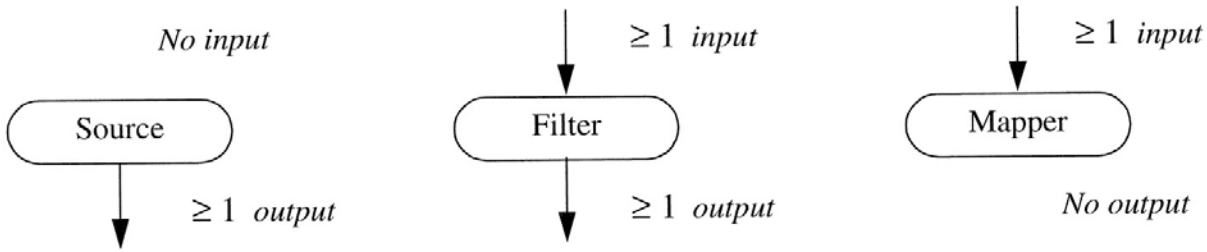



(a) Single-type system
*Input Type = Output Type*

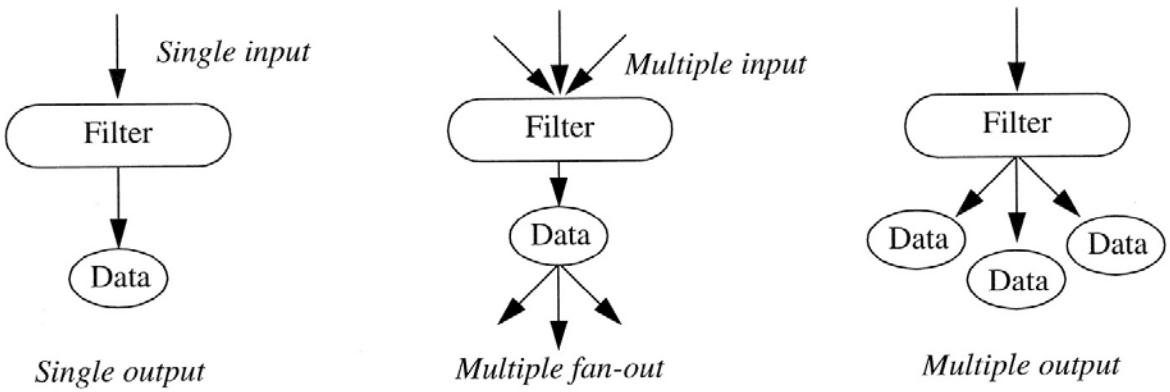
(b) Enforced type checking

# Multiplicity

- **Multiplicity** refers to the number of inputs and outputs of a process object



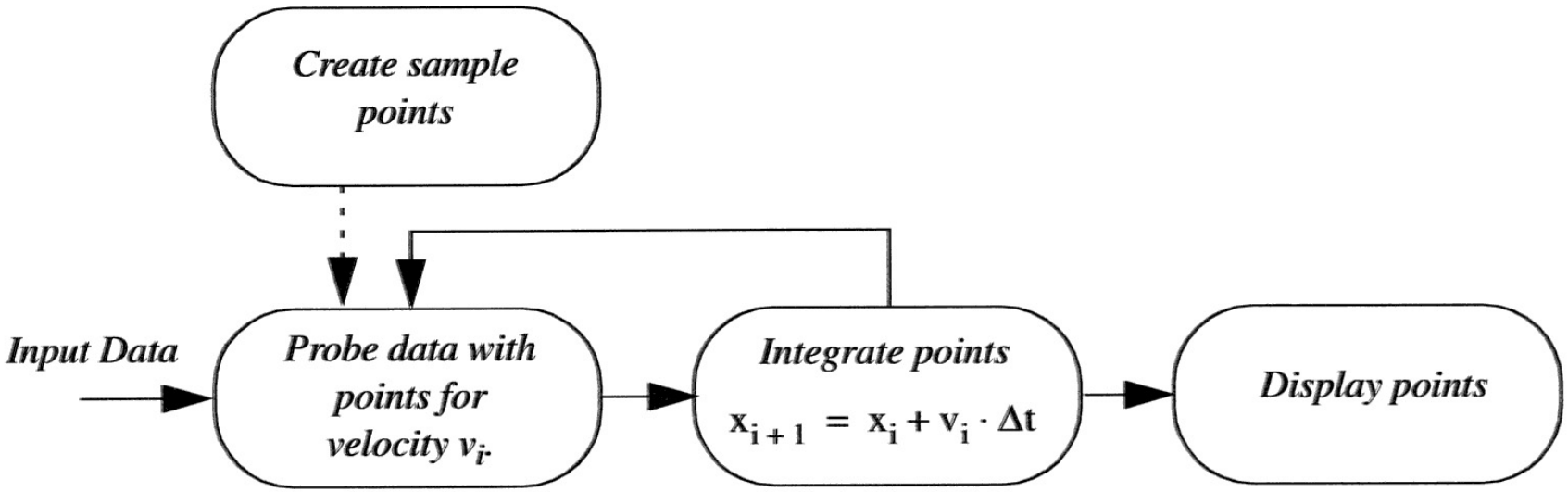

(a) Sources, filters, and mappers




(b) Multiplicity of input and output

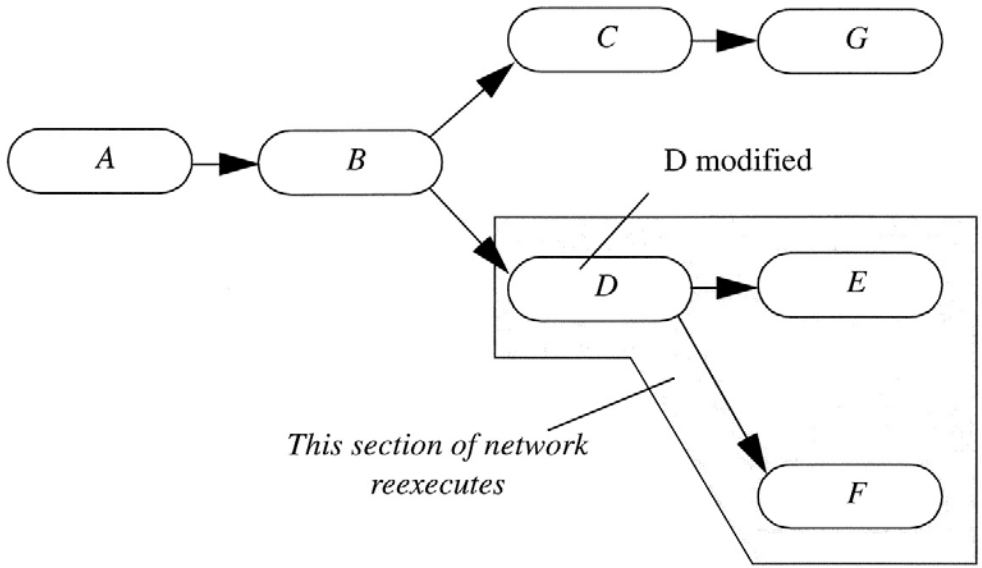
# Loops

- Most visualization networks are acyclic, but **feedback** is sometimes a useful option to have
- Output of a process object affects the input of a process object "upstream"



# Executing the Pipeline

- **Execution** refers the act of causing each process object to operate
- Pipeline re-executed as input data changed
- Ideally, a process will object will execute only its particular input is changed



# Execution Control

- **Demand-driven execution** vs. **event-driven execution**
- With demand-driven execution, we generate output upon request and execute only that portion of the pipeline affecting the output
- With event-driven execution, every change to a process object or its input causes a re-execution of the pipeline
- We should execute a process object only when its inputs have changed
- How do we know when this happens?

# Explicit Execution

- With an **explicit execution** approach, a special **executive object** monitors the process objects' parameters and inputs
- Orders re-execution of pipeline when necessary
- Can be demand-driven or event-driven
- Demand-driven: executive keeps track of changes and executes pipeline on request
- Event-driven: executive is notified when a changes occurs, who then re-executes the network

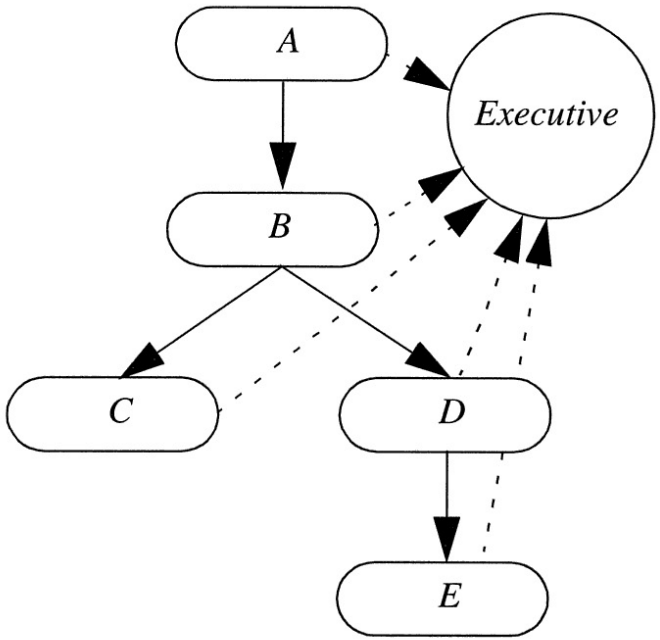
# Implicit Execution

- With an **implicit execution** approach, a process object executes itself only if its input or parameters change
- When object's output is requested, that object requests input from its input objects
- Repeats recursively up the pipeline to the sources
- Source objects check their parameters and external inputs, and re-execute if necessary (**update** pass)
- Recursion unwinds as downstream processes re-execute as necessary (**execution** pass)

# Implicit Execution

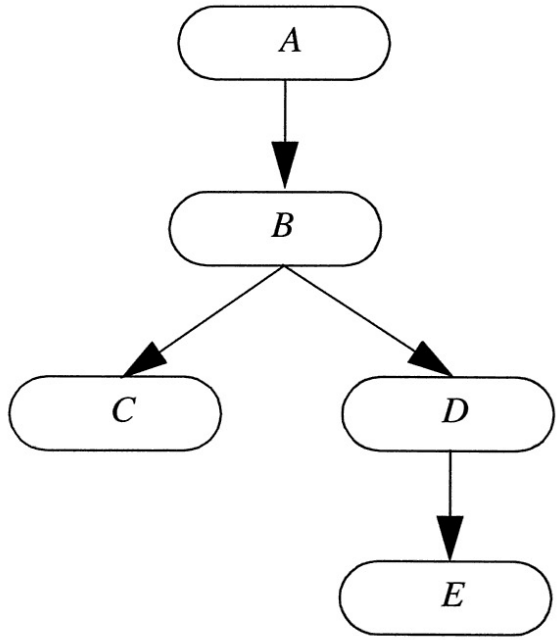
- Implicit execution requires demand-driven control
- Execution occurs when output is requested
- Simple approach

# Explicit vs. Implicit Execution




1. *A* parameter modified
2. Executive performs dependency analysis
3. Executive executes necessary modules in order *A-B-D-E*

(a) Explicit

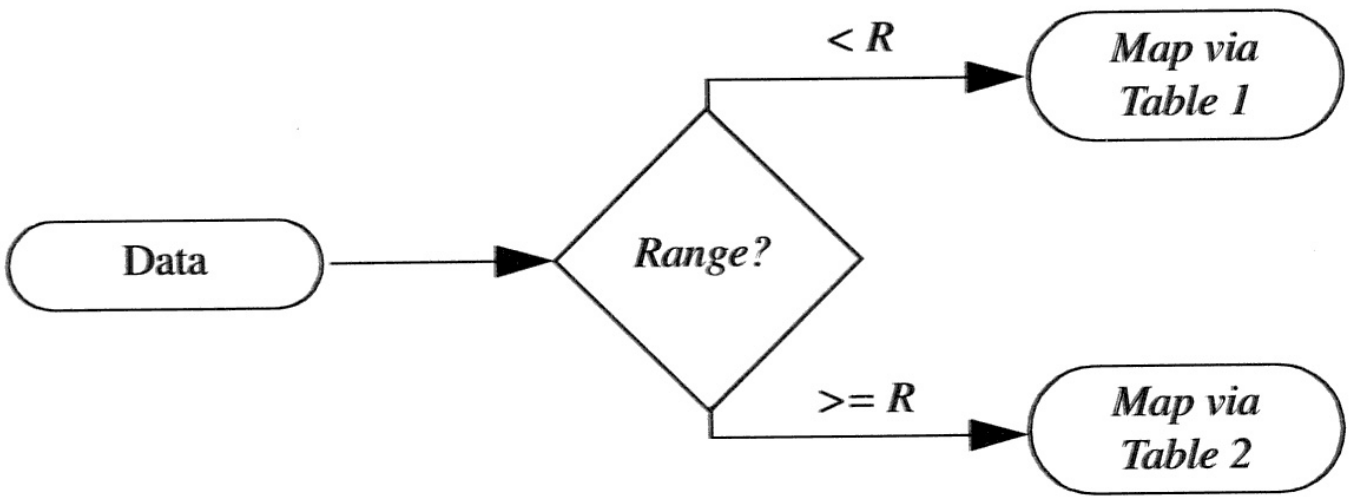



1. *A* parameter modified
2. *E* output requested
3. Chain *E-D-B-A* back propagates `Update()` method
4. Chain *A-B-D-E* executes via `Execute()` method

(b) Implicit

# Conditional Execution

- Execution performed only if a condition is met
- Example: map data through different color lookup tables depending on the range of the data



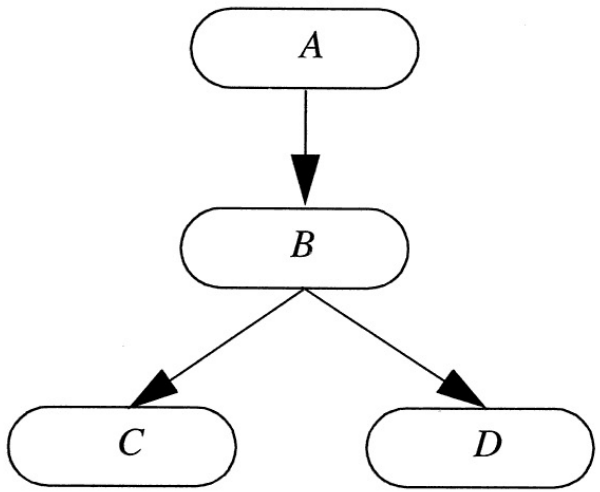
# Memory and Computation Tradeoff

- Visualization of non-trivial data is computationally expensive both in time and memory
- Static vs. dynamic memory allocation – what's the difference?
- In a visualization network, a **static memory model** maintains all the intermediate results in memory
- In a dynamic memory model, intermediate results are discarded as soon as they are no longer needed
- Static: less computation required later on
- Dynamic: more computation required later on

# Static and Dynamic Memory Models

- Use static when data size is small, vis. network traversed infrequently
- Use dynamic when data size is large, vis. network traversed frequently

A executes
B executes
C executes
D executes



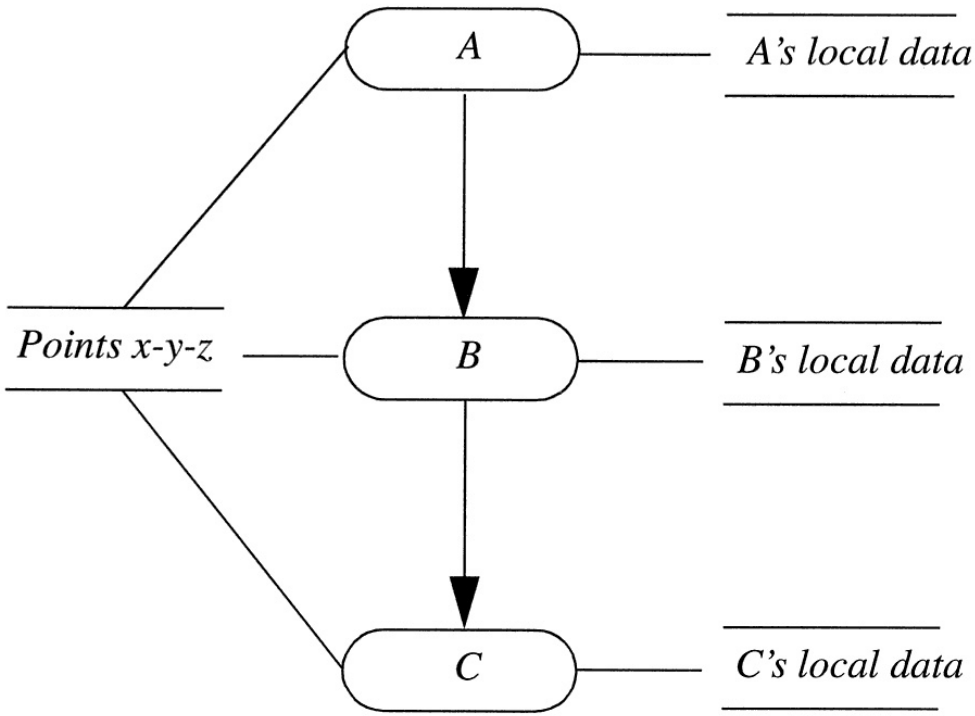

*Static model*

A executes
B executes
A releases memory
C executes
B releases memory
A re-executes
B re-executes
A releases memory
D executes
B releases memory
B releases memory.

*Dynamic model*

# Reference Counting

- **Reference counting:** each memory cell maintains a count of the number of other memory cells that point to it



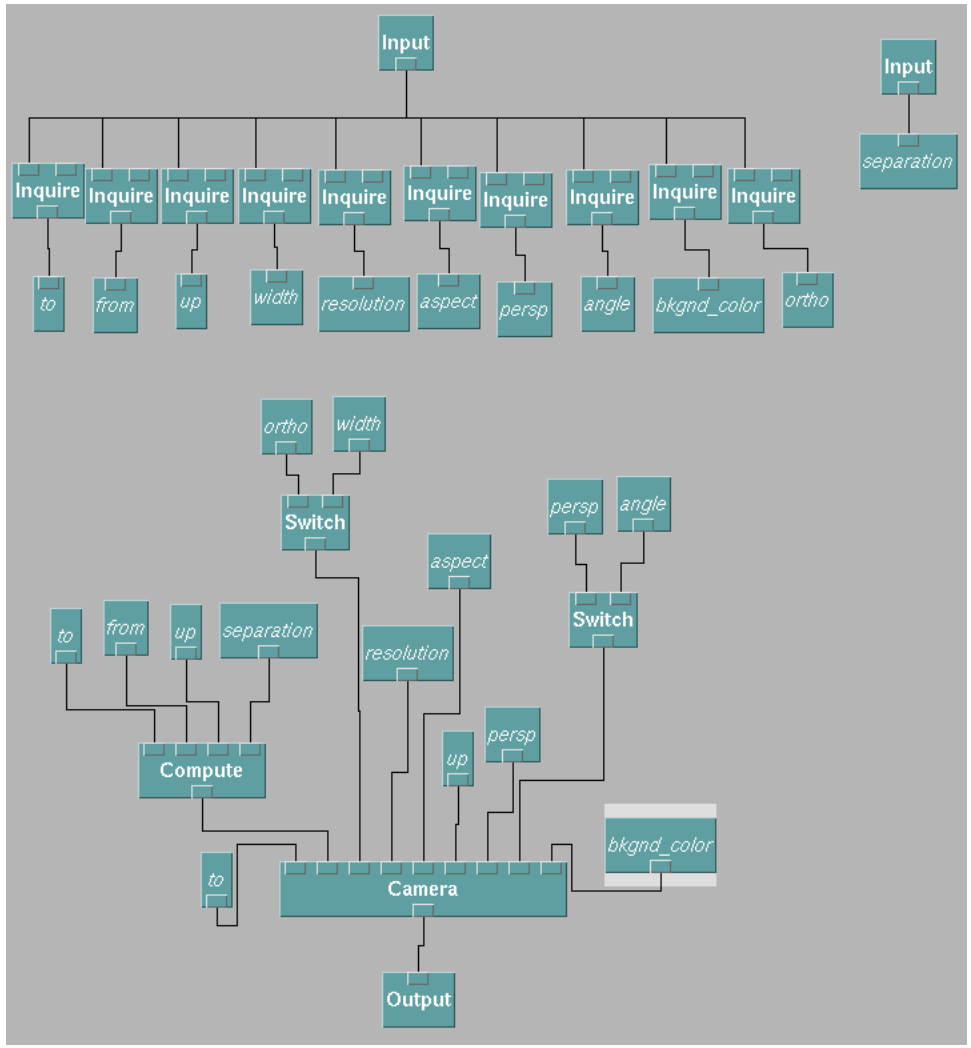
# Reference Counting

- Object freed once its reference count becomes zero
- This is how garbage collection works in languages like Java

# Programming Models

- **Visualization Models:** application software vs. programming libraries
- Example: fluid flow visualization system vs. library written in C++
- Third option: visual tool lets you build the network using a graphical interface, like IBM Data Explorer
- This is called a **visual programming** model

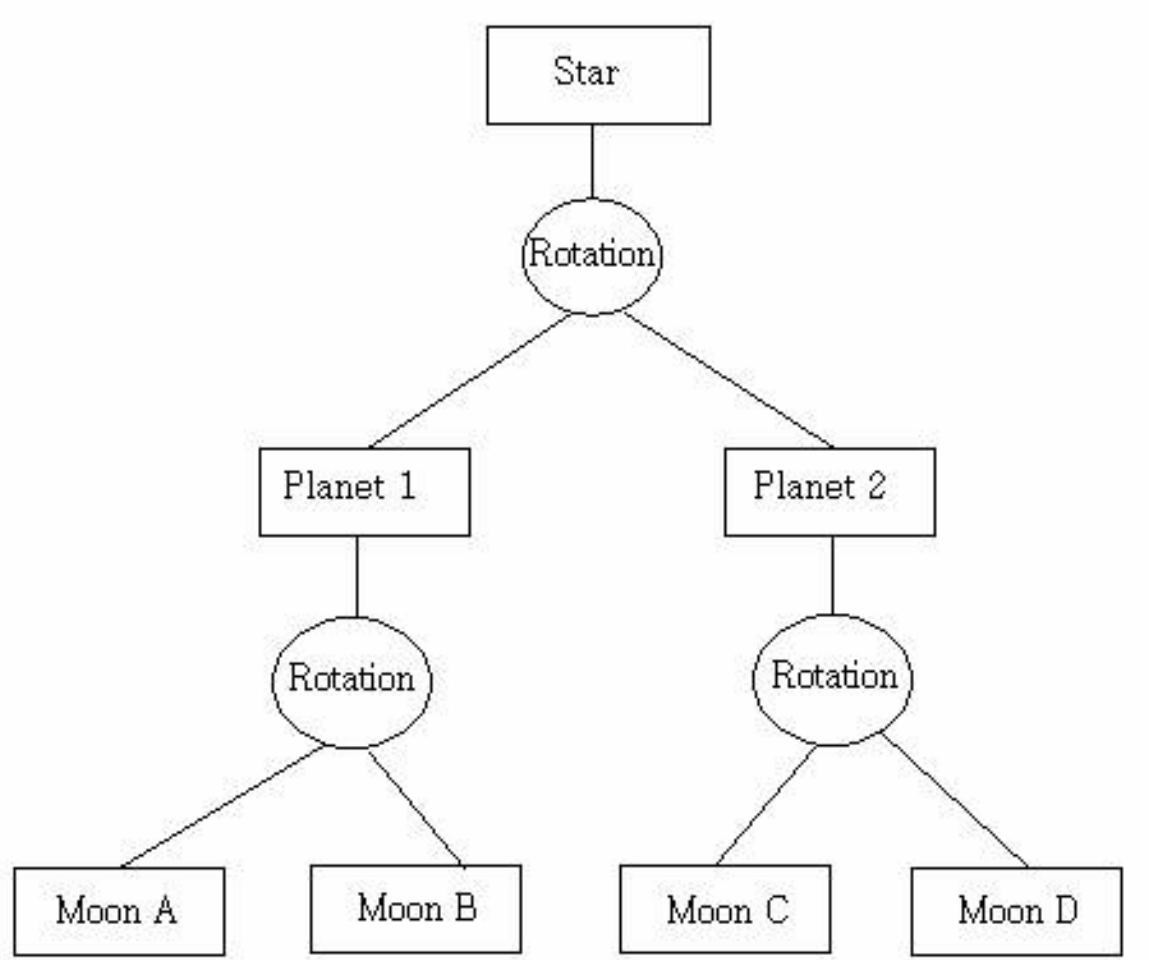
# Example IBM DX Visualization Network



# Programming Models

- **Scene graph** model
- A scene graph a tree-structure that represents objects in an order defined by the tree
- Not a visualization network! Rather, control rendering process
- Nodes contain 3D shapes and transformations
- Can work in conjunction with visualization networks
- Vis. network defines 3D shapes, scene graph draws and transforms them

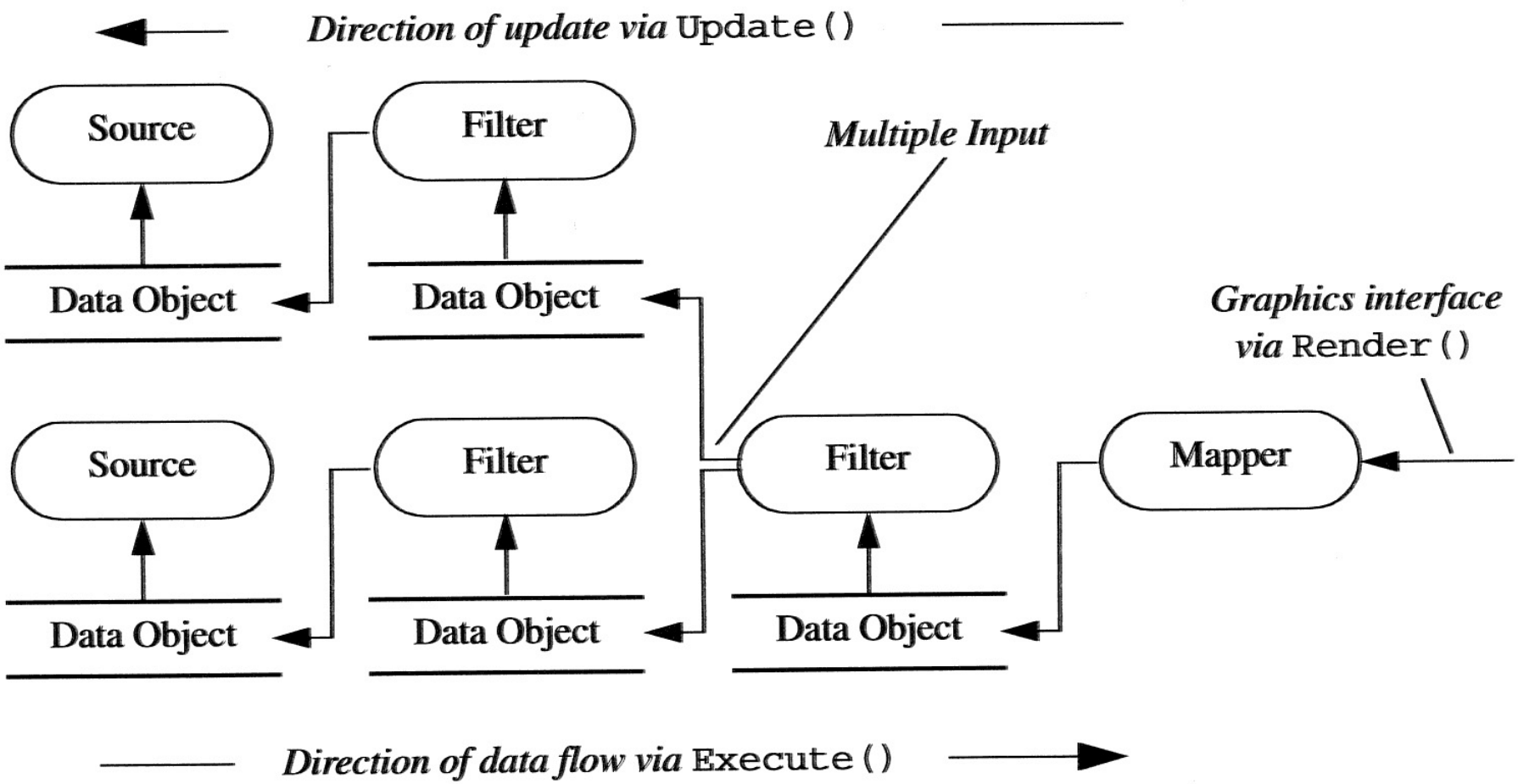
# Simple Scene Graph



# How VTK Fits In

- VTK borrows ideas from the above approaches
- It has aspects of visual programming systems as well as programming libraries
- Idea: be general enough to support many visualization applications, but not so general to require extremely extensive coding

# Example of VTK's Implicit Execution Framework with Multiple I/O Filters



# VTK's Visualization Pipeline

- Conditional execution implemented using C++ control structures (**if** statements, **while** loops, **for** loops, etc.)
- Computation/memory tradeoff can be controlled; by default, intermediate results saved to reduce computation

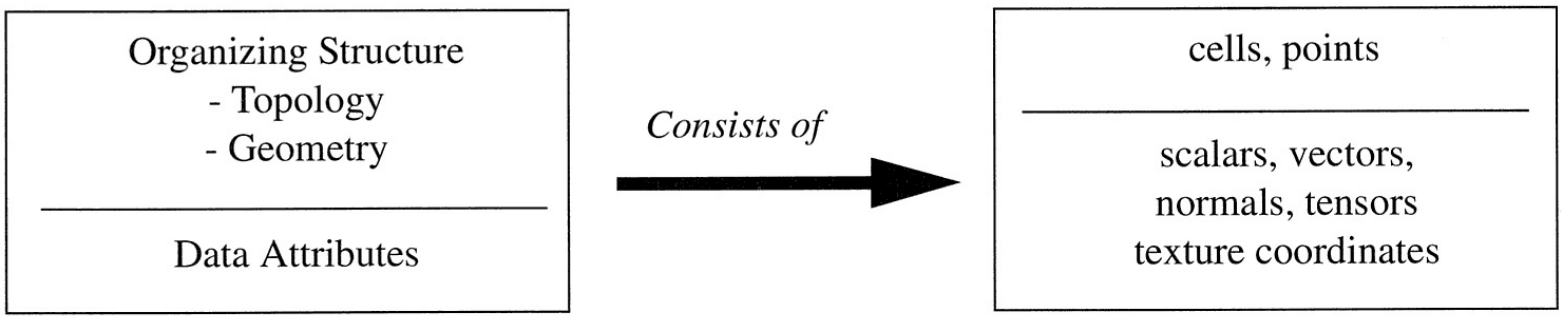
# Basic Data Representation

# Data Representations

- Many ways to represent data
- Points (e.g., 3D raster, point cloud)
- Lines
- Vectors
- These are all **discrete** data representations
- Data can be **regular** or **irregular**
- Regular = relationship exists between data points
- Compare: 3D raster vs. point cloud
- Data also has **dimension**: 1, 2, 3, …, n, …

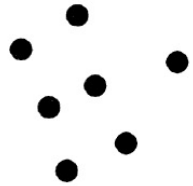
# Dataset = Structure + Attributes

- Structure = topology and geometry
- Topology refers to characteristics unchanged by transformations (holes, handles, branches)
- Geometry refers to (x,y,z) positions of data points

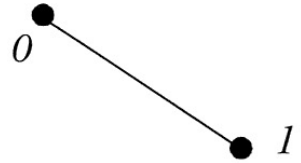



- In VTK, **cells** define topology, **points** define geometry
- See Figures 5-2 and 5-3 for examples of **cell types**
- Linear cell types and non-linear cell types

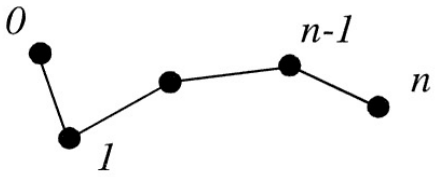
(a) Vertex



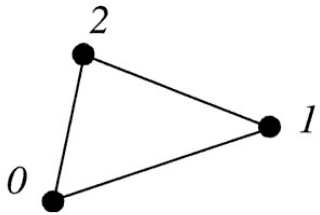
(b) Polyvertex



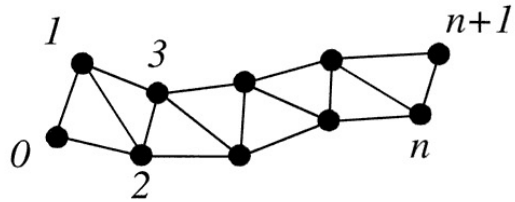

(c) Line



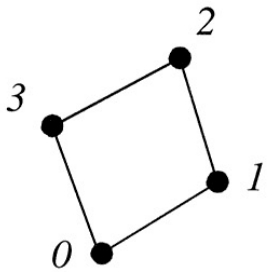

(d) Polyline (*n* lines)



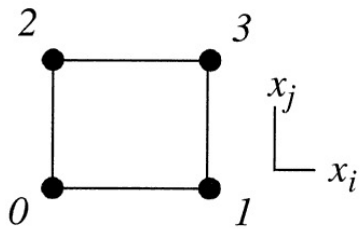

(e) Triangle



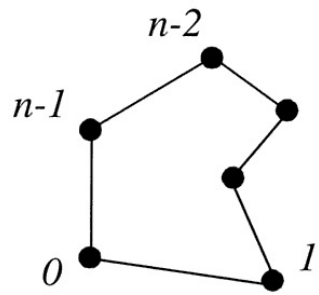

(f) Triangle strip (*n* triangles)




(g) Quadrilateral

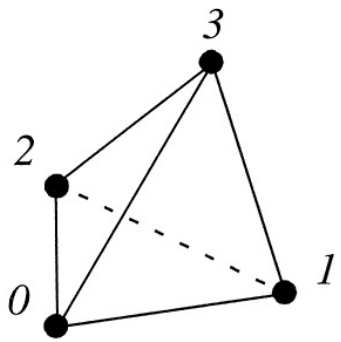



(h) Pixel

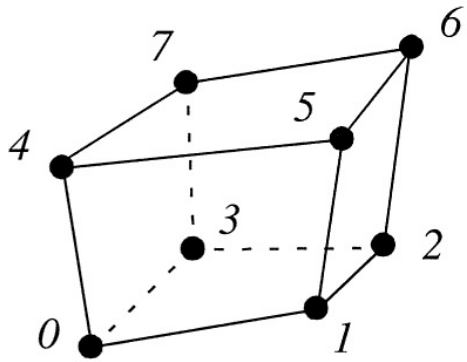



(i) Polygon (*n* points)

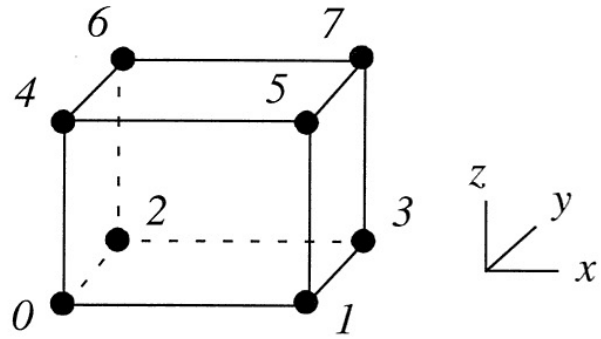
- Cell topology defined by **connectivity** of vertices



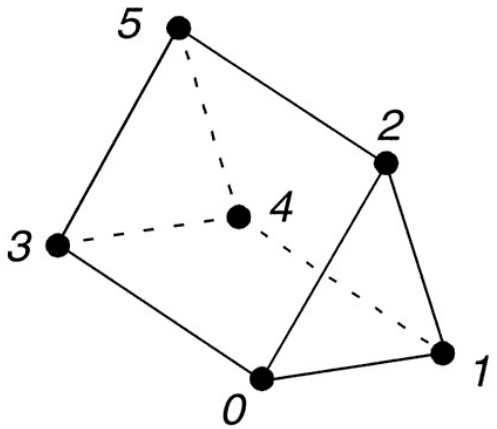

(j) Tetrahedron



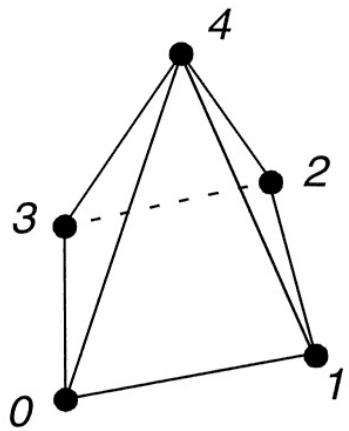

(k) Hexahedron



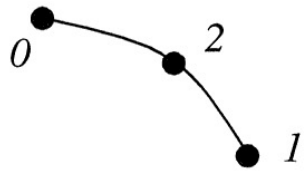

(l) Voxel



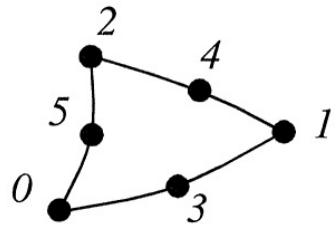

(m) Wedge



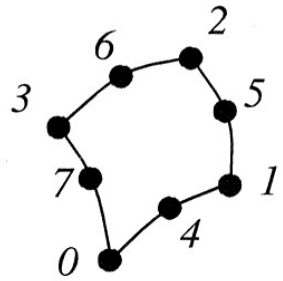

(n) Pyramid



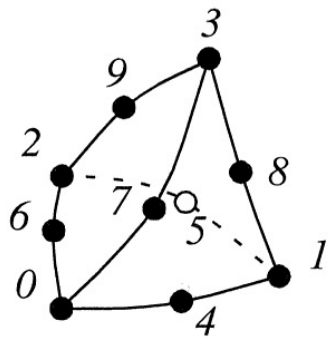

(a) Quadratic Edge



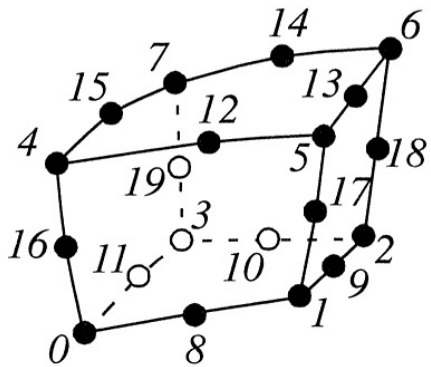

(b) Quadratic Triangle




(c) Quadratic Quadrilateral




(d) Quadratic Tetrahedron

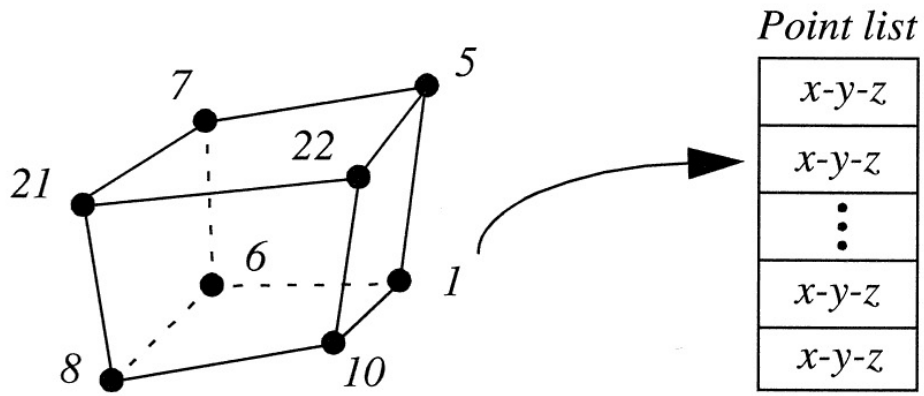



(e) Quadratic Hexahedron

# Cell Example: Hexahedron

- Vertices listed in special order define topology

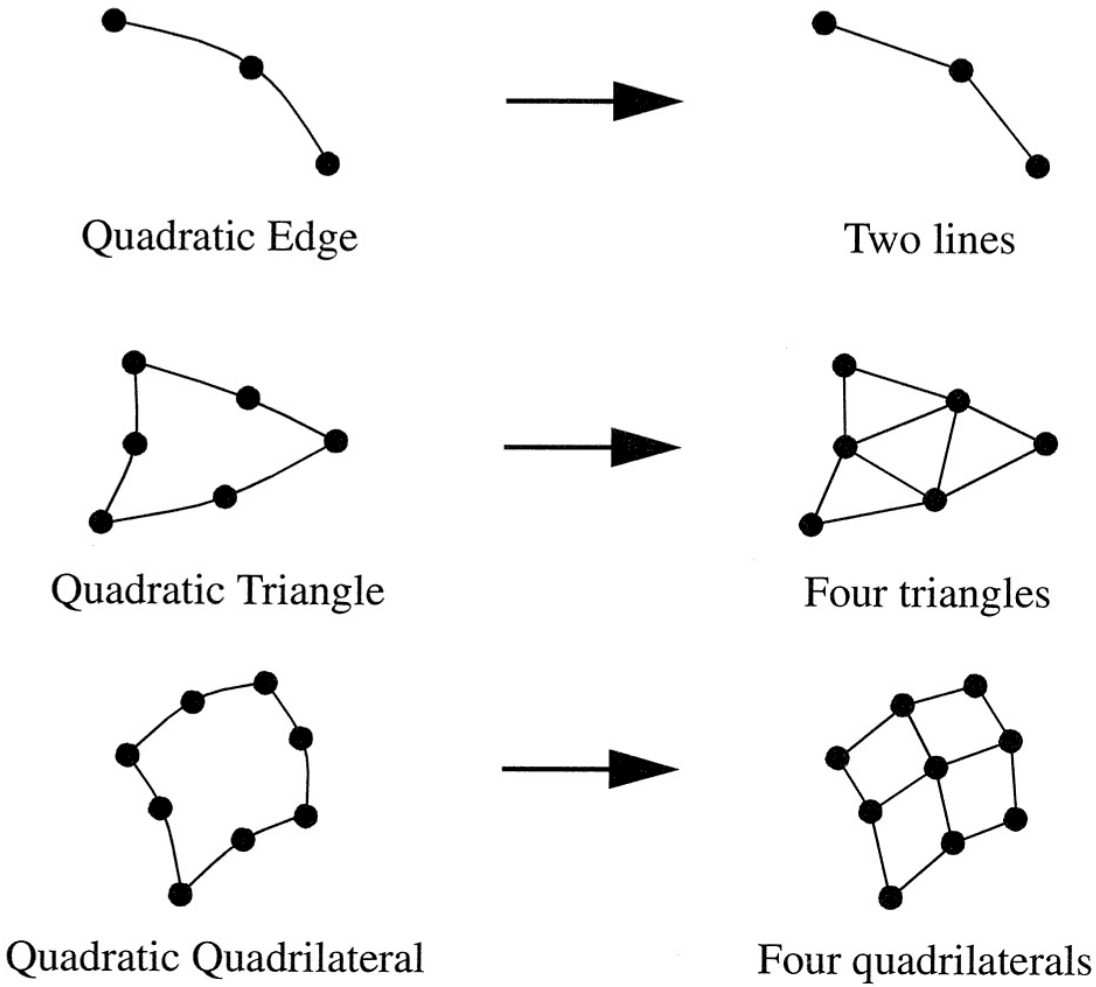
*Definition:*
*Type: hexahedron*
*Connectivity: (8,10,1,6,21,22,5,7)*



# Non-Linear Cell Decomposition

- Non-linear cells must be linearized for visualization
- Break non-linear cells into linear cells

# Non-Linear Cell Decomposition



# Attribute Data

- Data values (attributes) usually assigned to vertices, as opposed to edges or faces
- Why?
- Interpolation concept easy to apply across edges and faces
- Common attributes include:
  - Temperature, density, velocity, pressure, heat flux, chemical concentration, others
- Scalars, vectors, tensors

# Attribute Data

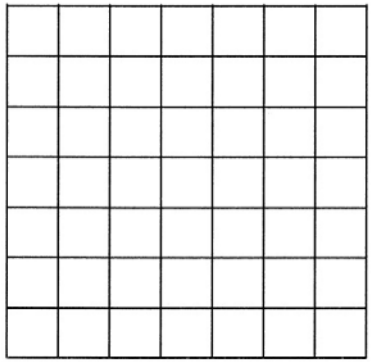
- **Scalar** data is data that is single-valued at all locations in a data-set
- Examples: temperature, stock price, elevation
- **Vector** data is data with magnitude and direction
- Examples: position, velocity, acceleration
- **Normals** (direction vectors) are vectors of magnitude 1
- **Texture coordinates** map a point from Cartesian space into a 1-D, 2-D or 3-D texture space
- Textures let us add color, transparency and other details to geometric shapes

# Attribute Data

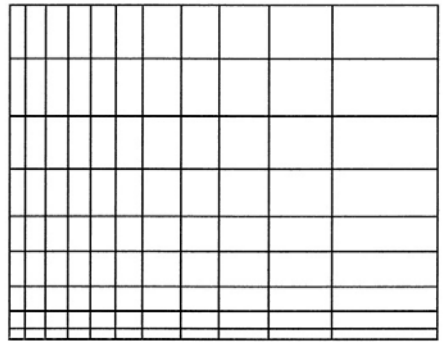
- **Tensors** are mathematical generalizations of vectors and scalars
- Usually written as matrices
- Tensor visualization is extremely difficult

# Types of Data-sets

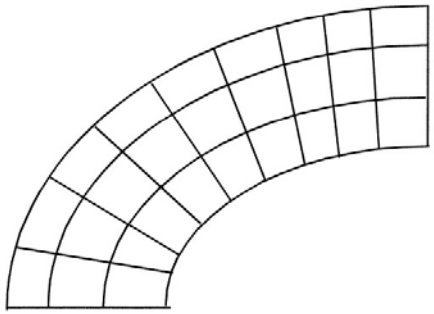
- Regular vs. irregular structure – refers to topology of data-set
- Data-sets with regular topology, we do not need to store connectivity information
- Points themselves can be regular or irregular
- If irregular, we need to store the positions
- Unstructured data must be explicitly represented
- High computational and storage costs usually



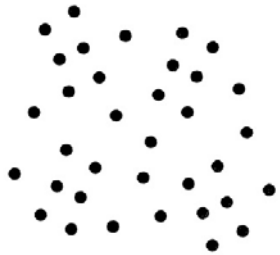
(a) Image Data



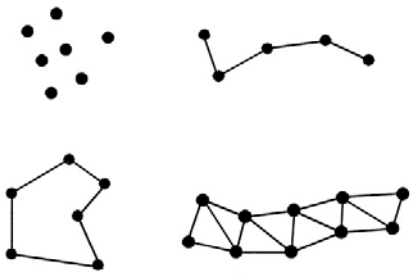
(b) Rectilinear Grid



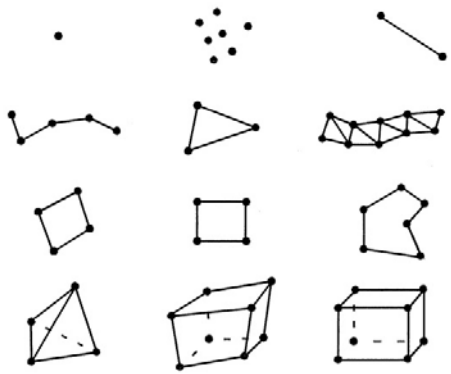
(c) Structured Grid



(d) Unstructured Points



(e) Polygonal Data



(f) Unstructured Grid

# Polygonal Data

- Vertices, edges, polygons, polylines, triangle strips, etc.
- Triangle strips can represent $n$ triangles using only $n+2$ points, vs. $3n$ points normally required

# Image Data

- Collection of points and cells on a regular, rectangular grid
- Also called a “raster”
- (Book uses word “lattice” – avoid!)
- 2D grid → image
- 3D grid → volume
- *i*-*j*-*k* coordinate system parallel to global *x*-*y*-*z* coordinate system
- Simple representation, but “curse of dimensionality”

# Rectilinear Grid

- Regular grid, but spacing along axes can vary
- Need to store 3 extra arrays of length $n_x$, $n_y$, $n_z$ – dimensions of the grid
- Each array stores spacing, basically

# Structured Grid

- Regular topology, irregular geometry
- Curvilinear grids most common type

# Unstructured Points

- No topology, irregular geometry
- Also called **point clouds**

# Unstructured Grid

- Irregular topology and geometry
- Any combination of cells permitted
- Encountered in relatively few applications
- e.g., computational geometry

# VTK Data Representations

- vtkFloat Array
- vtkImageData
- vtkRectilinearGrid
- vtkStructuredGrid
- vtkPolyData
  - vtkCellArray
- vtkUnstructuredGrid

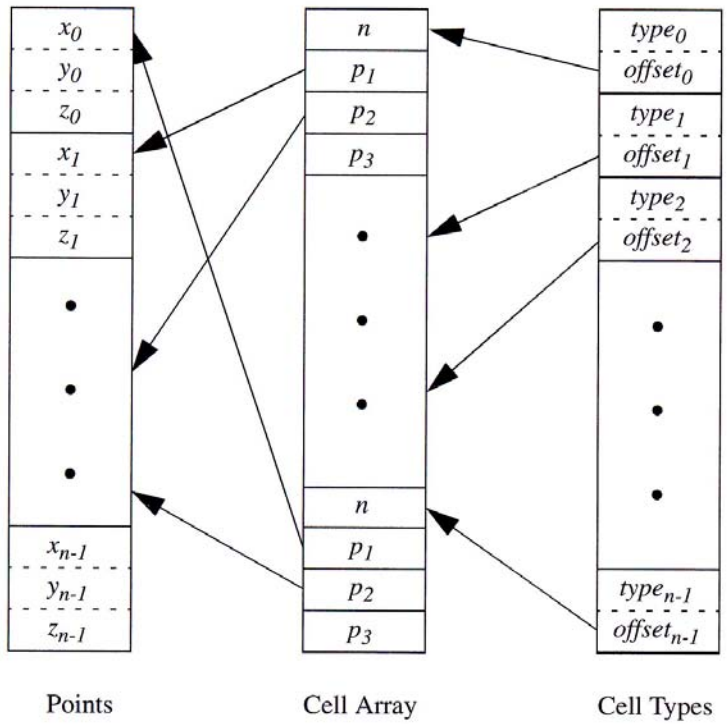



**Figure 5–13** The data structure of the class `vtkUnstructuredGrid`. (This is a subset of the complete structure. See Chapter 8 for complete details.)

# VTK Data Representations

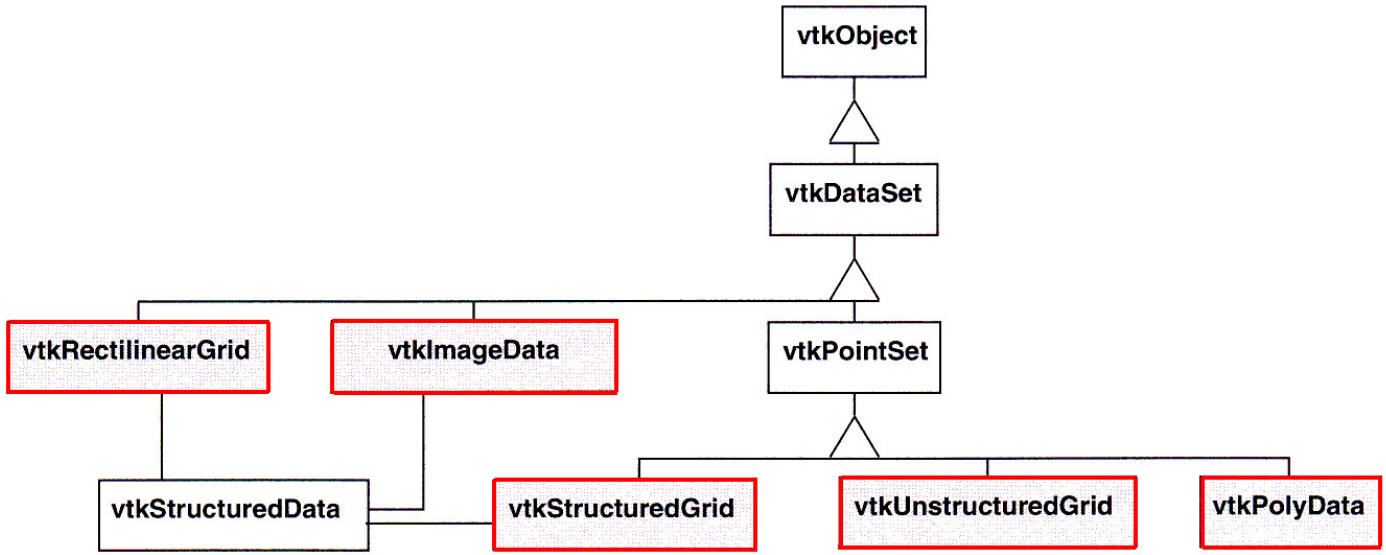



**Figure 5–14** Dataset object diagram. The five datasets (shaded) are implemented in VTK.

# VTK Cell Types

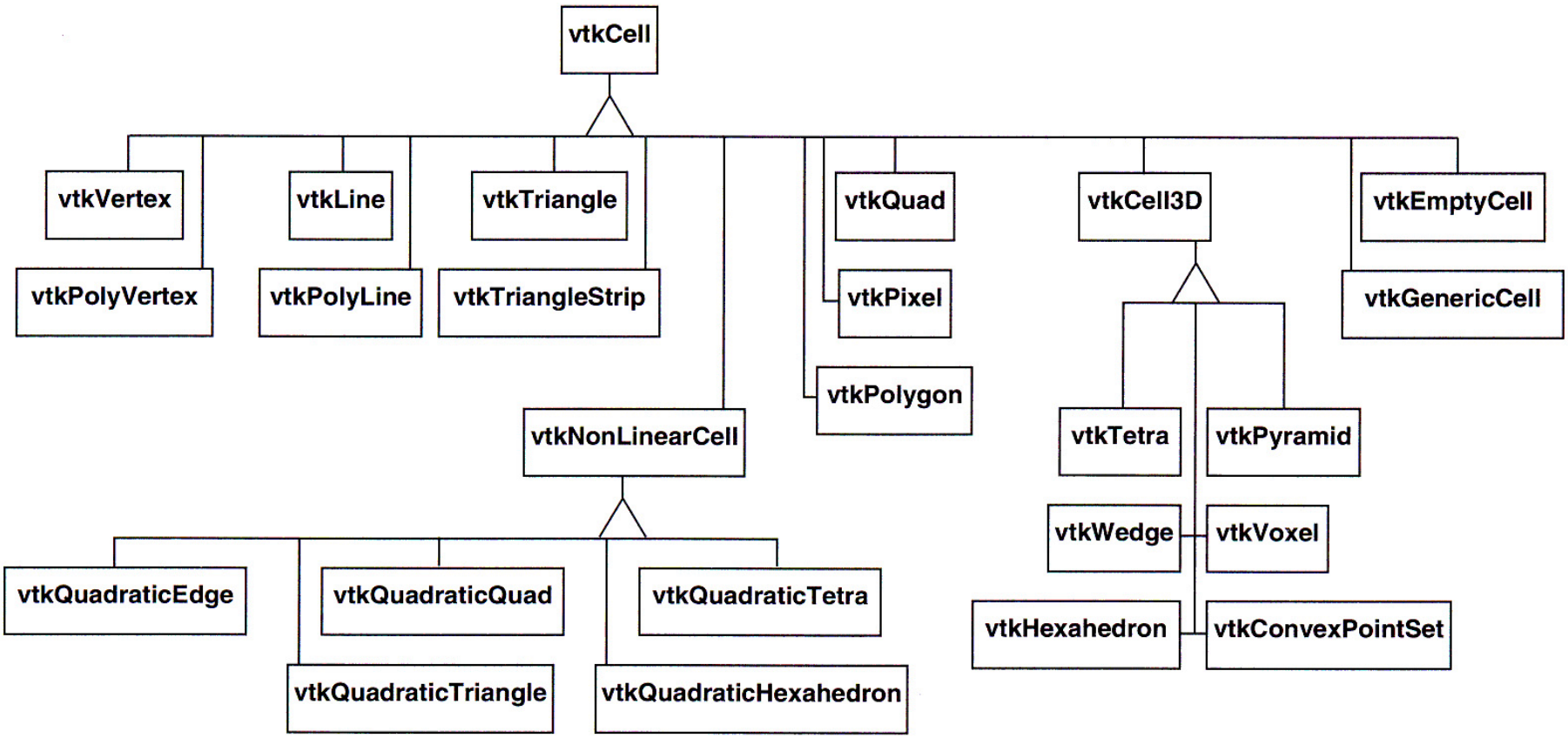
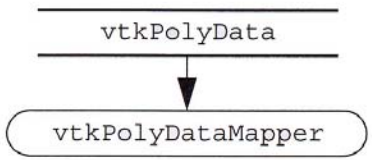
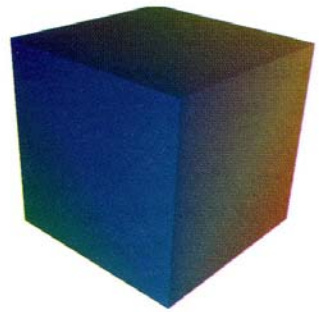



**Figure 5–15** Object diagram for twenty concrete cell types in VTK. vtkEmptyCell represents NULL cells. vtkGenericCell can represent any type of cell. Three-dimensional cells are subclasses of vtkCell3D. Higher order cells are subclasses of vtkNonLinearCell.

# Example: Cube.cxx

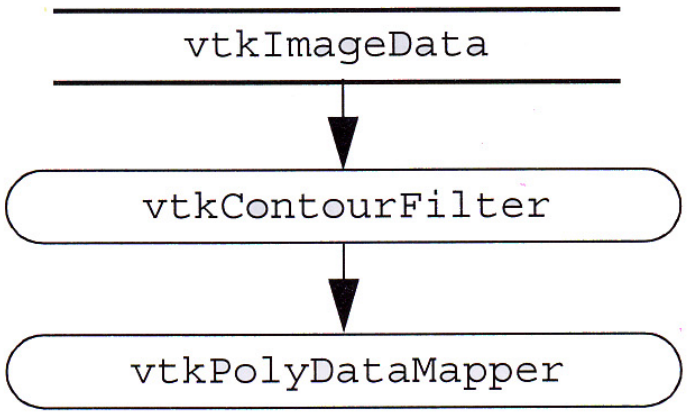


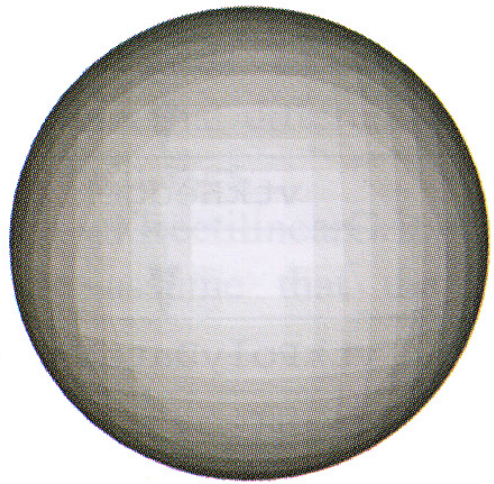



```
vtkPolyData *cube = vtkPolyData::New();
vtkPoints *points = vtkPoints::New();
vtkCellArray *polys = vtkCellArray::New();
vtkFloatArray *scalars = vtkFloatArray::New();

for (i=0; i<8; i++) points->InsertPoint(i,x[i]);
for (i=0; i<6; i++) polys->InsertNextCell(4,pts[i]);
for (i=0; i<8; i++) scalars->InsertTuple1(i,i);

cube->SetPoints(points);
points->Delete();
cube->SetPolys(polys);
polys->Delete();
cube->GetPointData()->SetScalars(scalars);
scalars->Delete();
```

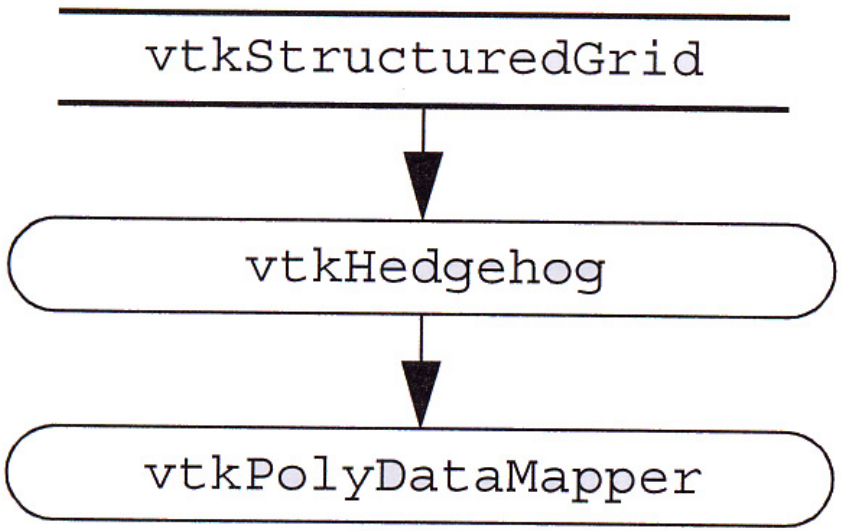
**Figure 5–17** Creation of polygonal cube (Cube.cxx).

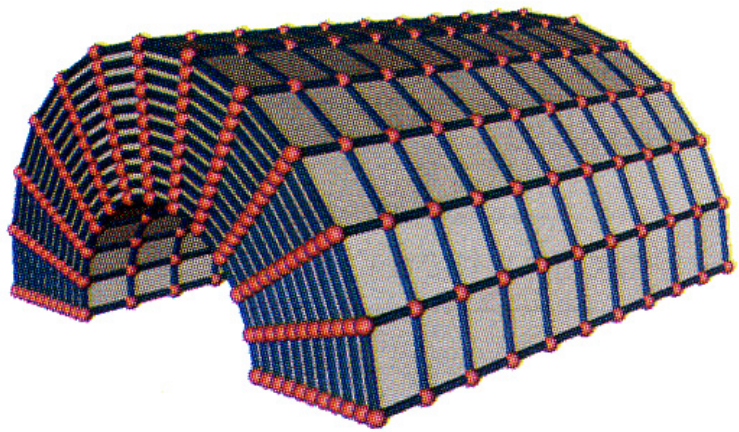
# Example: Vol.cxx



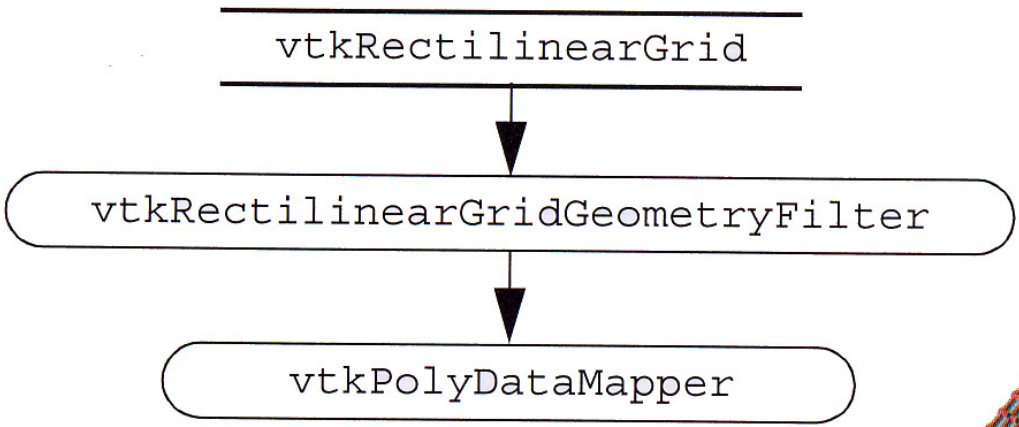



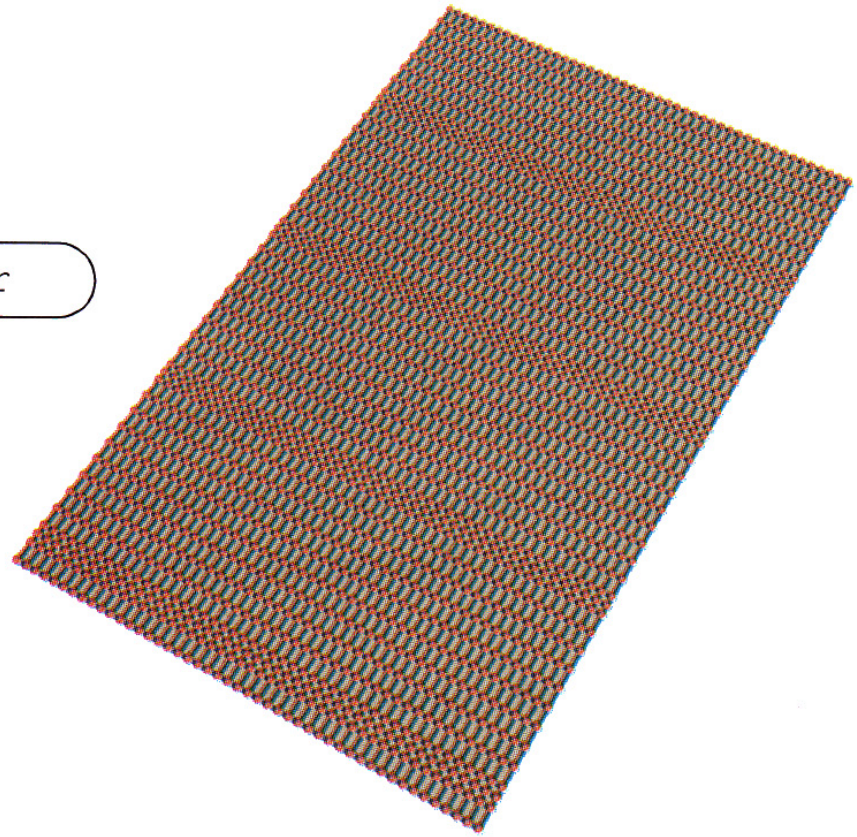
# Example: SGrid.cxx



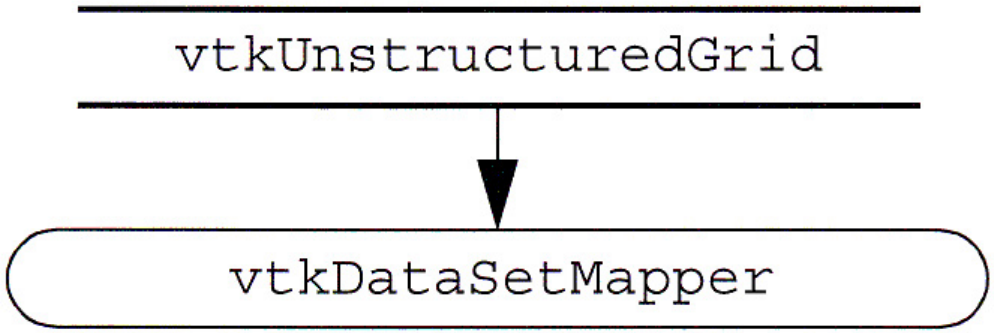



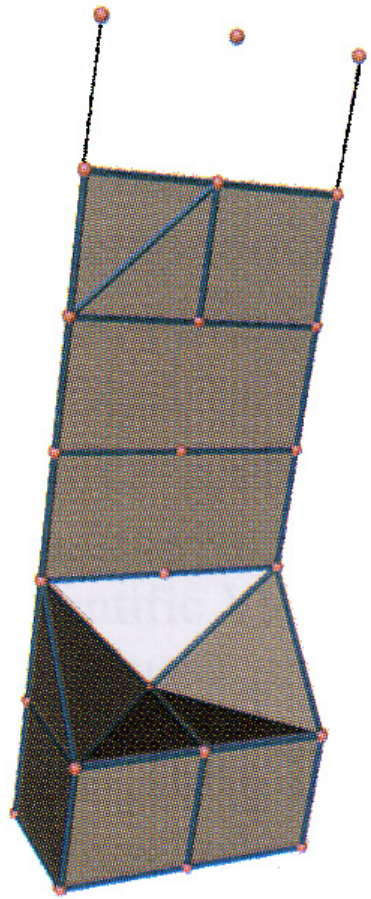
# Example: RGrid.cxx



# Example: UGrid.cxx



# Fundamental Visualization Algorithms

# Visualization Algorithms

- "Algorithms that transform data are the heart of visualization"
- Algorithms classified according to **structure** and **type** of data
- **Geometric transformations** change geometry but not topology
- Examples: translation, rotation, scaling
- **Topological transformations** change topology but not geometry
- Example: convert from regular to irregular grid

# Visualization Algorithms

- **Attribute transformations** convert or create attributes in data
- Example: convert vector to scalar
- **Combined transformations** change data structure and attributes
- Algorithms that change data type include **scalar algorithms, vector algorithms, tensor algorithms,** and **modeling algorithms**
- **Volume visualization** and **vector visualization** have their own special algorithms

# Scalar Algorithms

- **Color mapping** – map scalar data to colors
- Why scalars?
- How would you map a vector to a color?
- **Color lookup table (LUT)** – attributes inside particular range are mapped to color

$$s_i < \min,\ i = 0$$

$$s_i > \max,\ i = n - 1$$

$$i = n\left(\frac{s_i - \min}{\max - \min}\right)$$

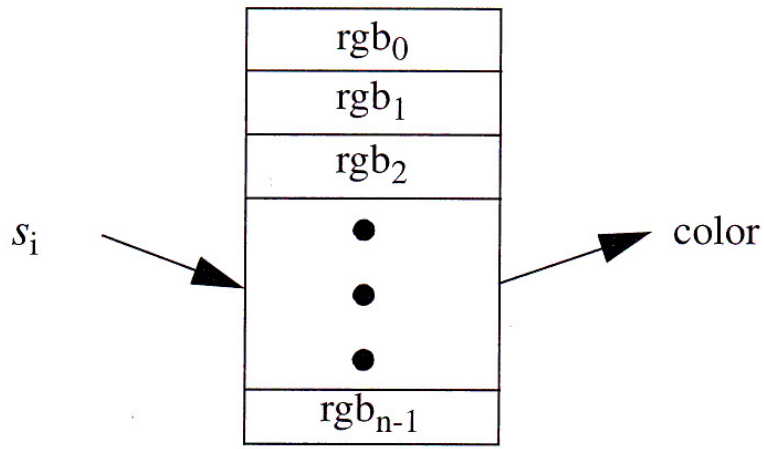



**Figure 6–1** Mapping scalars to colors via a lookup table.

# Transfer Functions

- More general form of lookup table
- Can map data to color as well as transparency
- Usually expressed as actual functions

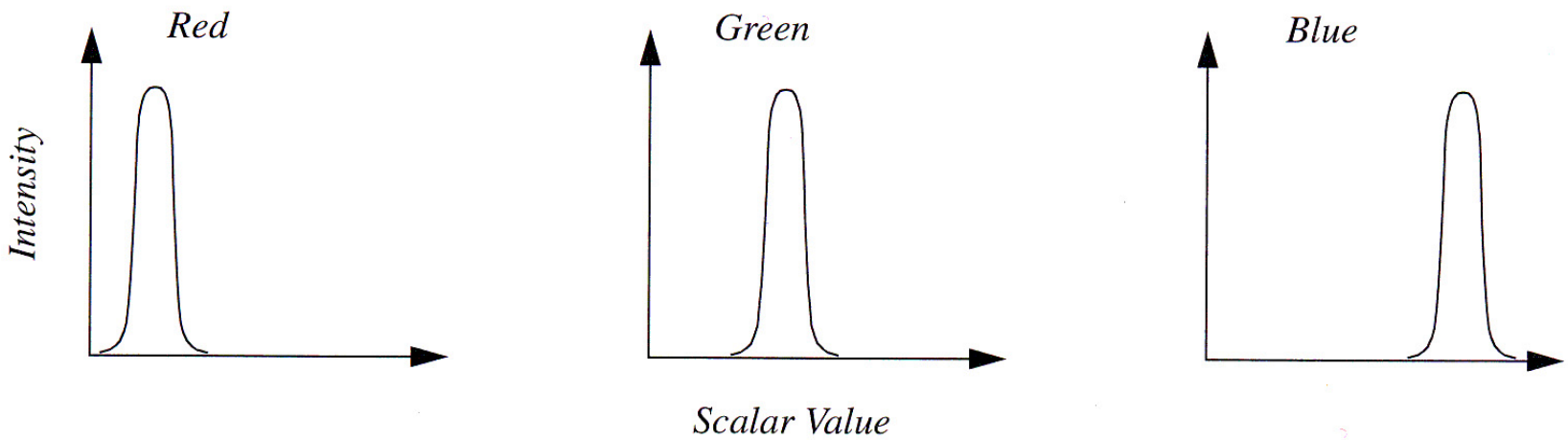
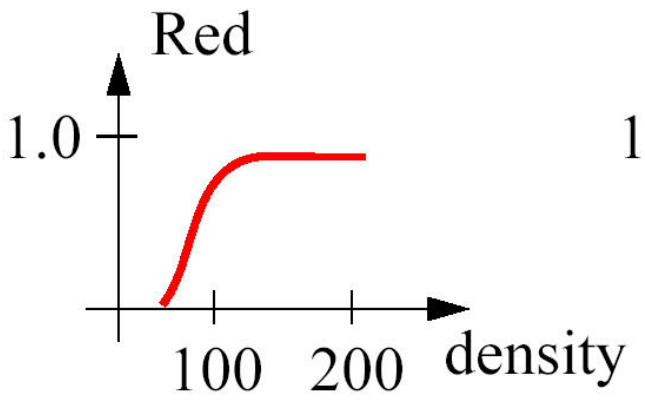
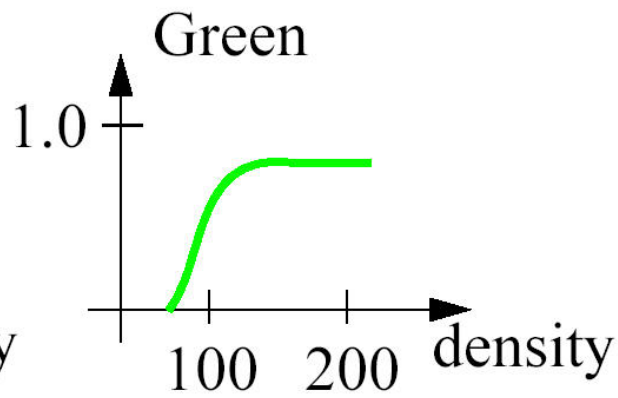
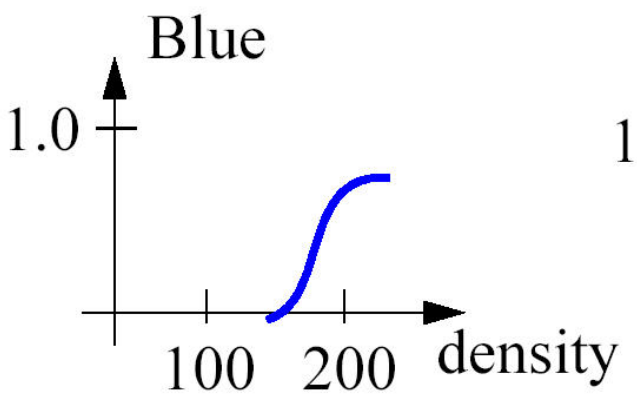


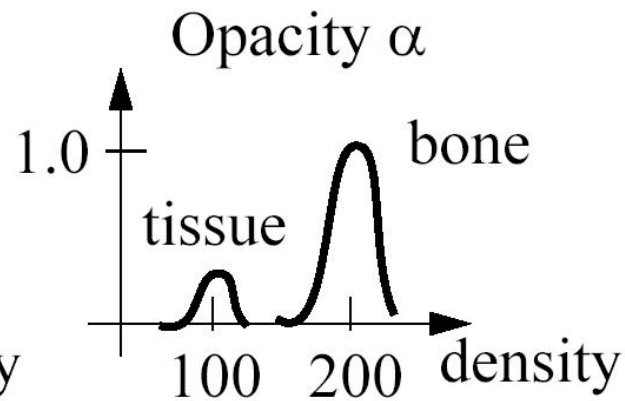

**Figure 6–2** Transfer function for color components red, green, and blue as a function of scalar value.

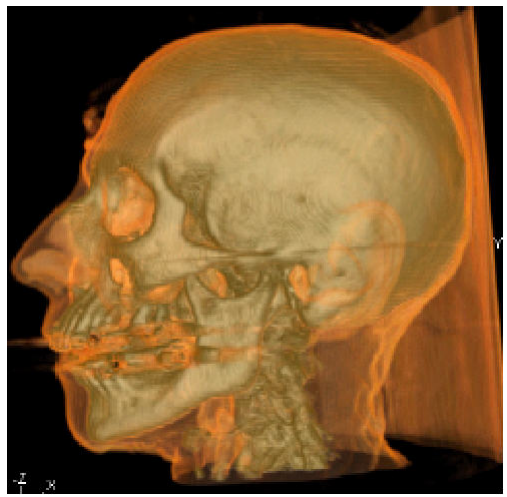
# Transfer Functions



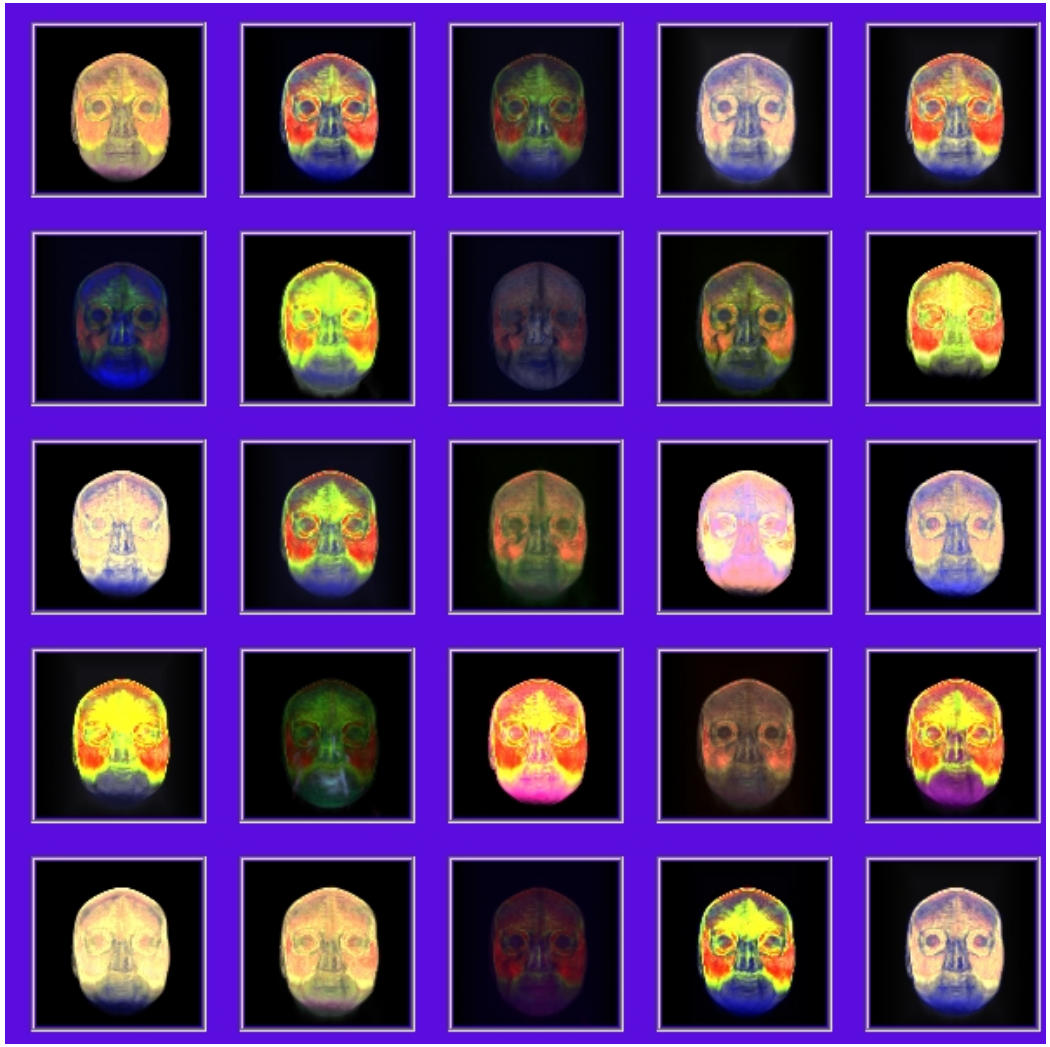



# Transfer Functions

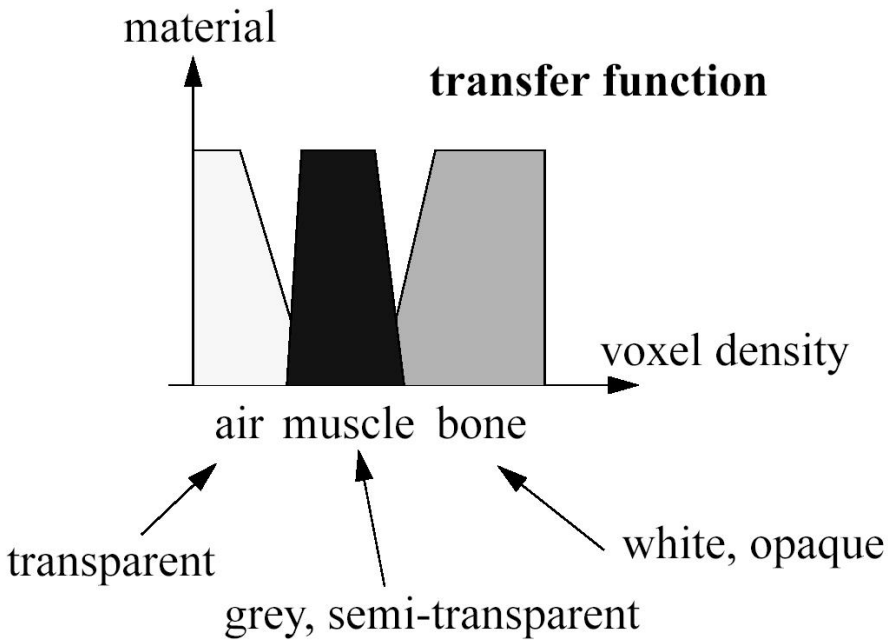
- Difficult to design
- Semi-automatic systems exist: **transfer function design galleries**
- Idea: generate random transfer functions, user selects ones he likes, system *mutates* them using a genetic algorithm to create new ones

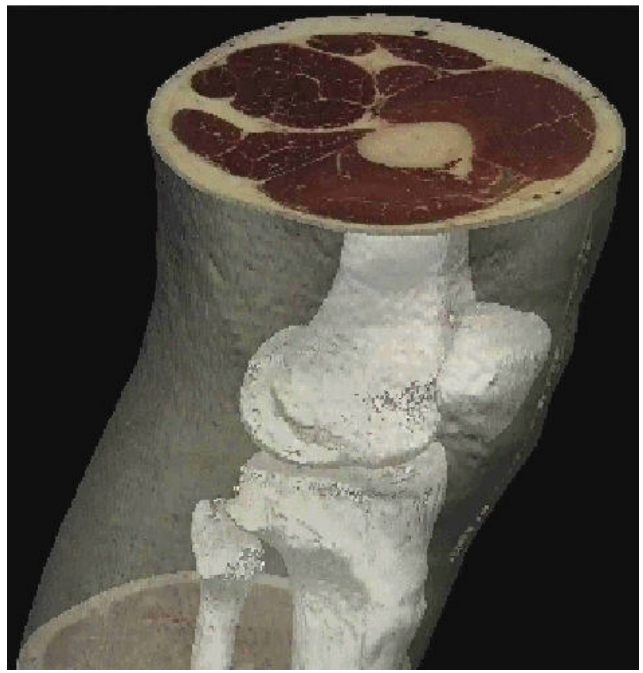
# Transfer Function Design Galleries



# Transfer Functions

- The assignment of color and transparency to density is also called **classification**



# Transfer Functions

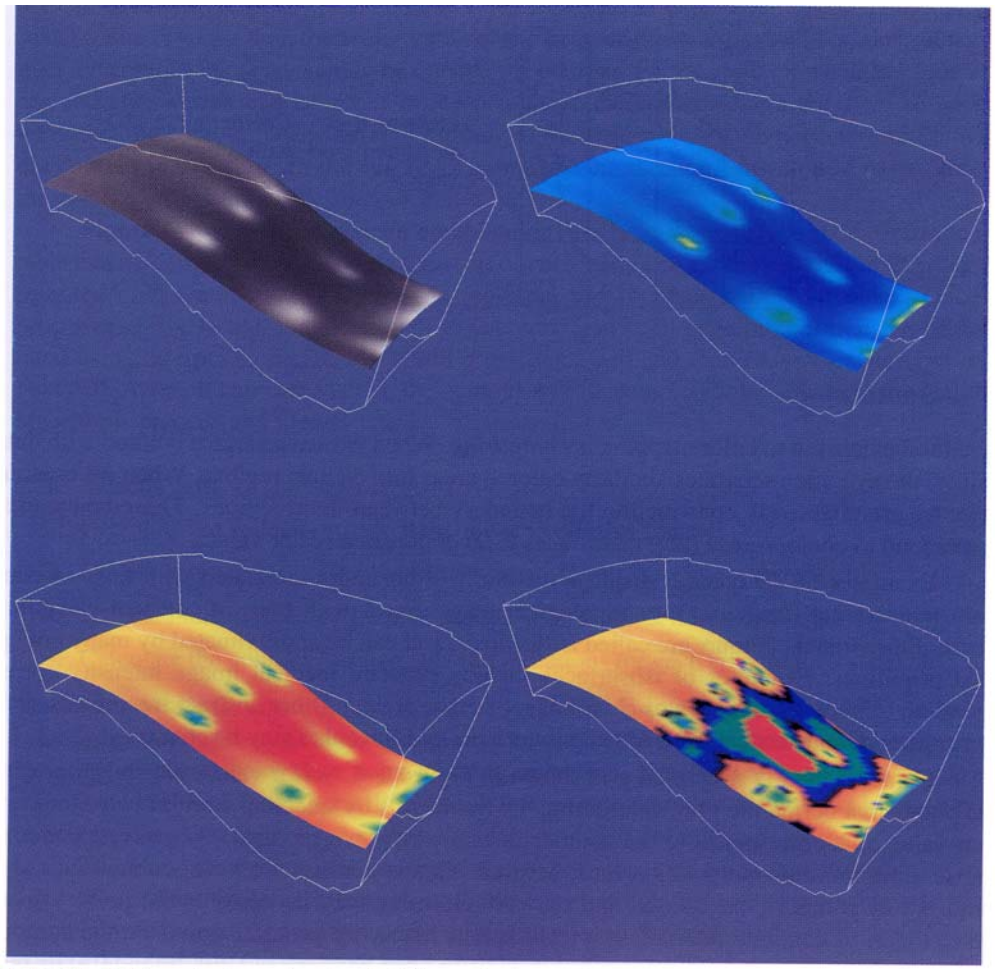


**Figure 6–3** Flow density colored with different lookup tables. Top-left: grayscale; Top-right rainbow (blue to red); lower-left rainbow (red to blue); lower-right large contrast (`rainbow.tcl`).

# Contouring

- **Isocontour** and **isosurface extraction** can reveal structure of data (e.g., isobars on weather maps)
- Separate data into regions
- Isocontours: connected line segments
- Isosurfaces: triangular meshes

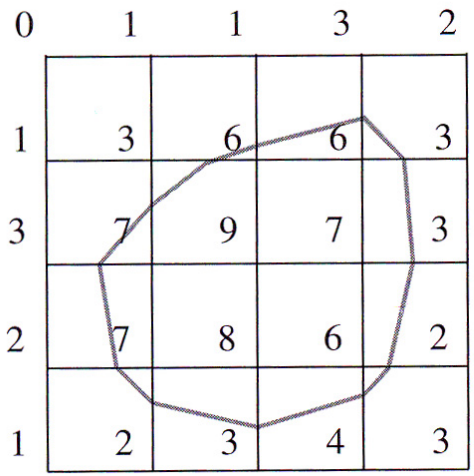



**Figure 6–4** Contouring a 2D structured grid with contour line value = 5.

# Contouring

- Isolines cross cell boundaries
- Use **interpolation** to compute crossing point
- **Marching squares** algorithm processes each quadrilateral cell independently
- Each vertex may be inside or outside (or on) contour
- How many cases must we consider?
- Ambiguous cases

# Marching Squares Cases

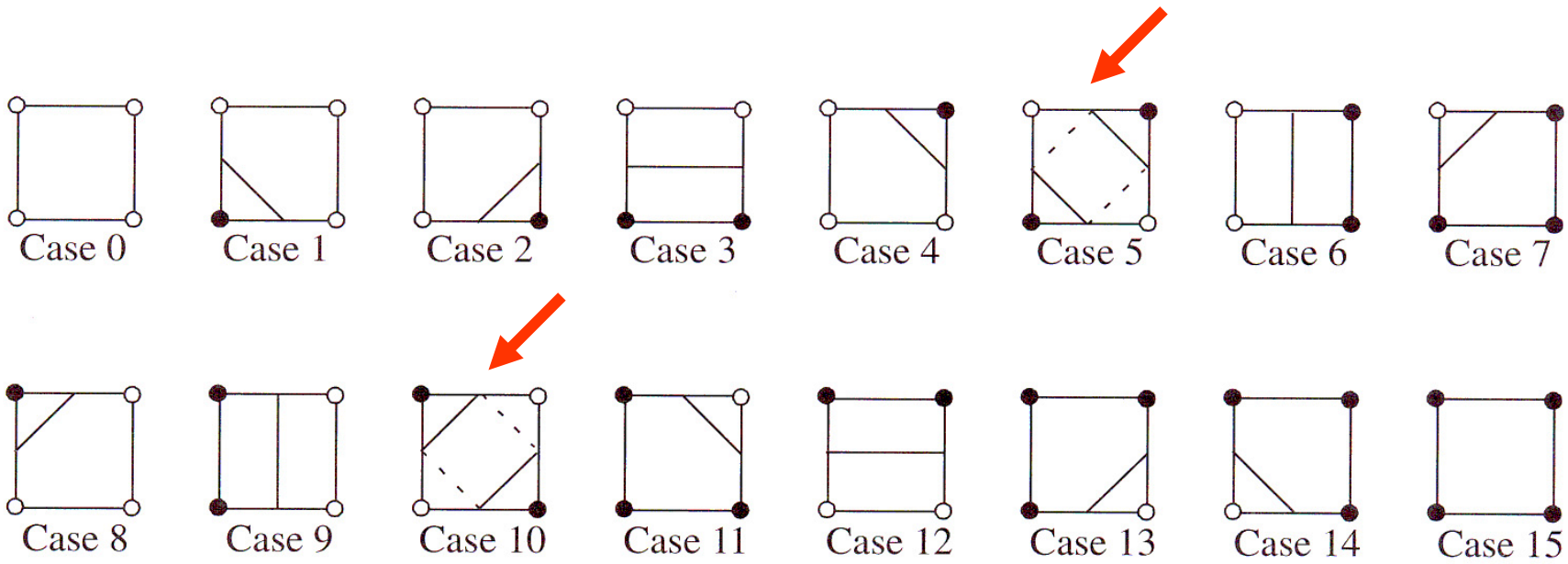
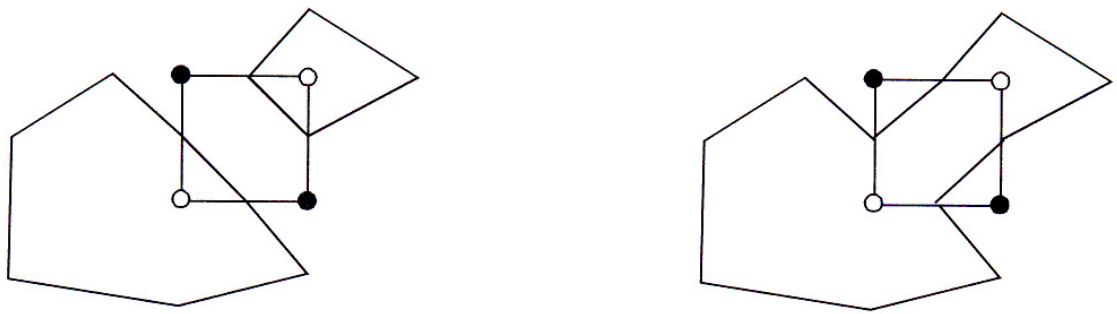



**Figure 6–5** Sixteen different marching squares cases. Dark vertices indicate scalar value is above contour value. Cases 5 and 10 are ambiguous.

# Marching Squares Ambiguous Case



(a) Break contour (b) Join contour

**Figure 6–8** Choosing a particular contour case will break (a) or join (b) the current contour. Case shown is marching squares case 10.

# Marching Cubes

- **Marching cubes** algorithm extracts isosurfaces from 3D rasters
- Very famous algorithm
- How many cases of hexahedral cells must we consider?
- Each of 8 vertices may be inside or outside
- $2^8 = 256$
- Lots of symmetry → really only 15 cases to consider

# Marching Cubes Cases

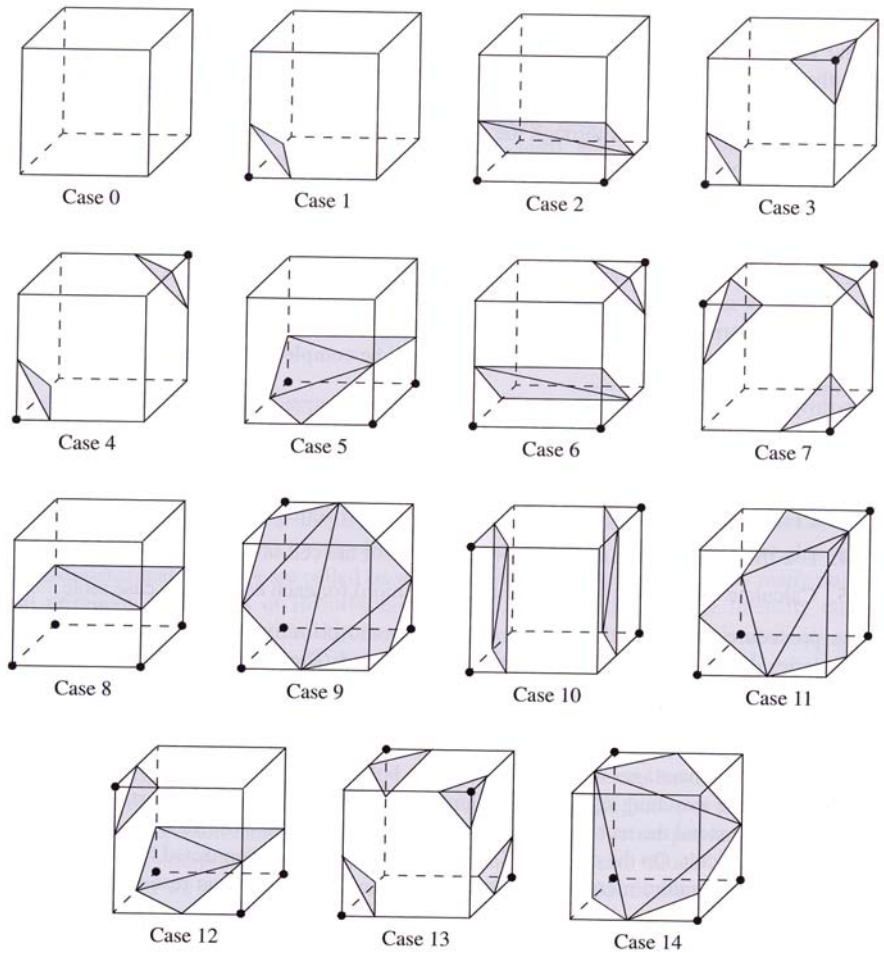



**Figure 6–6** Marching cubes cases for 3D isosurface generation. The 256 possible cases have been reduced to 15 cases using symmetry. Dark vertices are greater than the selected isosurface value.

# Marching Cubes Ambiguous Cases

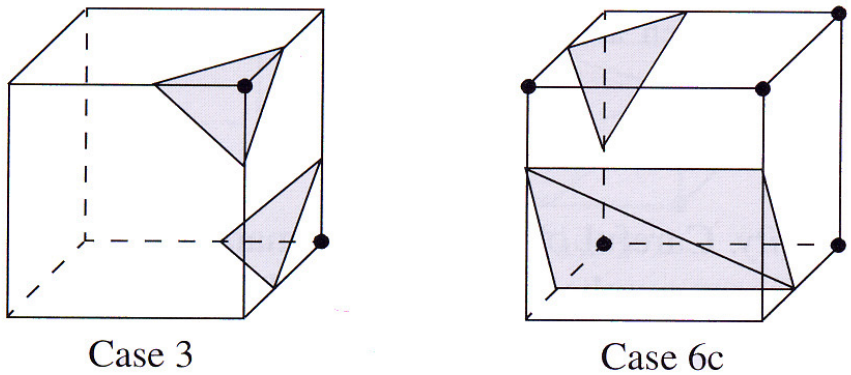



**Figure 6–9** Arbitrarily choosing marching cubes cases leads to holes in the isosurface.

# Marching Cubes Complementary Cases Used to Avoid Holes

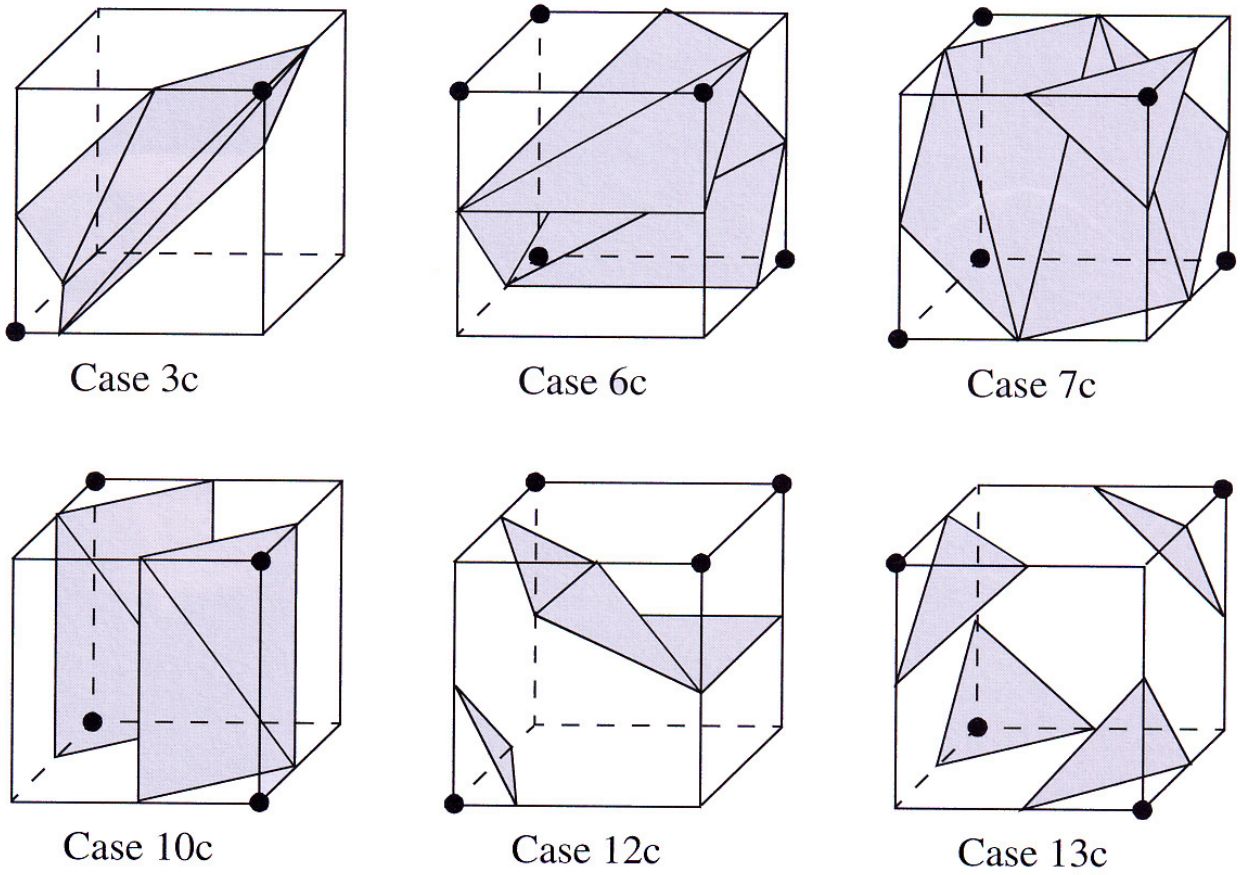



**Figure 6–10** Marching cubes complementary cases.

# Marching Triangles & Tetrahedra

- Can extend marching squares to *marching triangles*, and marching cubes to *marching tetrahedra*
- Divide squares into triangles, cubes into tetrahedra (how?) and then run different algorithms
- Tradeoff for both algorithms: simplicity vs. memory usage

# Contouring Examples

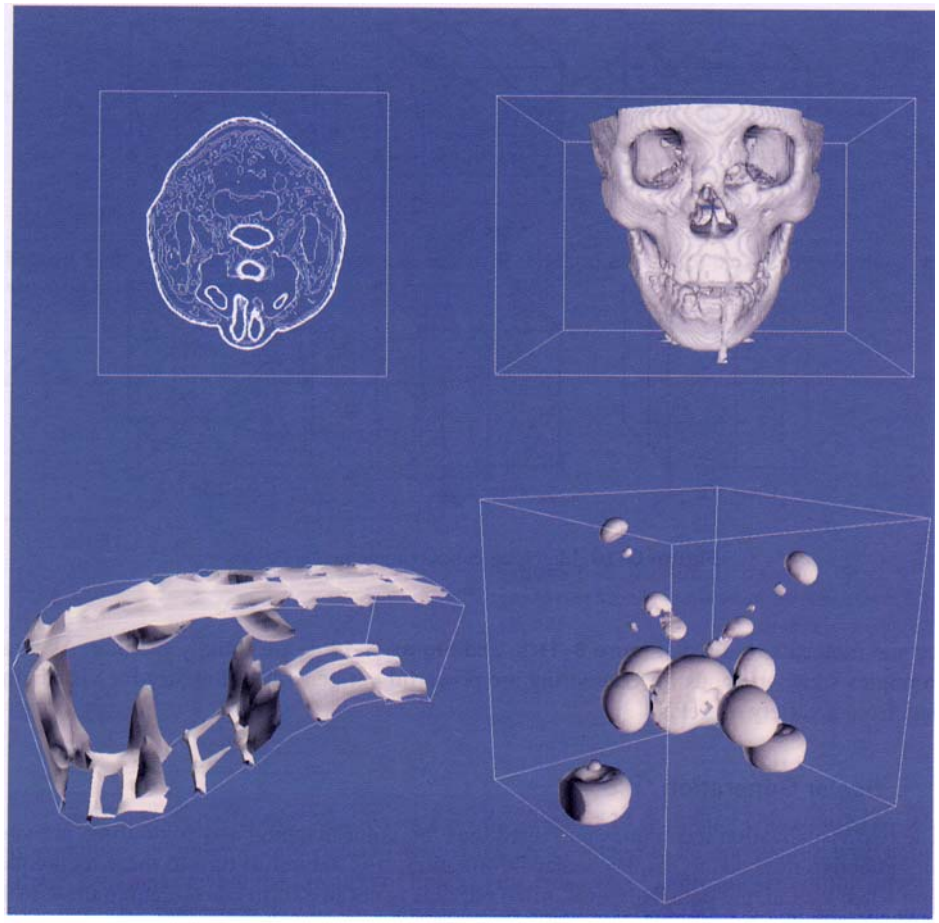


**Figure 6–11** Contouring examples. (a) Marching squares used to generate contour lines (`headSlic.tcl`); (b) Marching cubes surface of human bone (`headBone.tcl`); (c) Marching cubes surface of flow density (`combIso.tcl`); (d) Marching cubes surface of iron-protein (`ironPIso.tcl`).

# Scalar Generation

- Vectors and other n-D quantities can be turned into scalars
- Example: taking magnitude of vector
- Example: Hawaii terrain visualization created by projecting vector onto vertical
- Normalize vectors to give maximum magnitude of 1.0
- Steepest slope mapped to brightest color

# Scalar Generation

$$s_i = \frac{(p_i - p_l) \cdot (p_h - p_l)}{|p_h - p_l|^2}$$

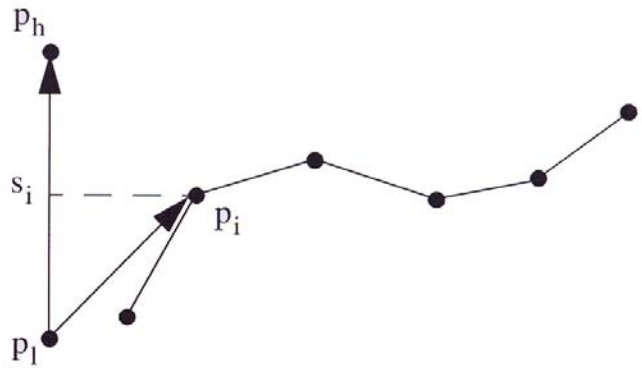
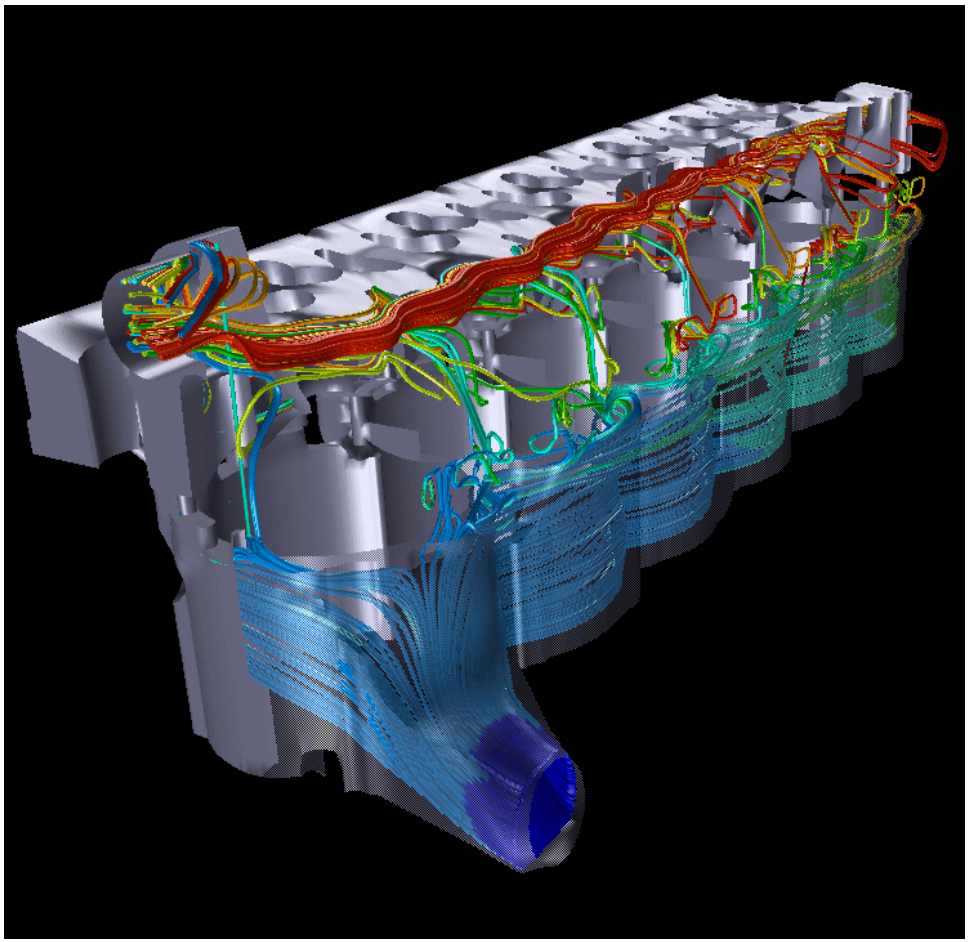



**Figure 6–12** Computing scalars using normalized dot product. Bottom half of figure illustrates technique applied to terrain data from Honolulu, Hawaii (`hawaii.tcl`).

# Vector Field Visualization

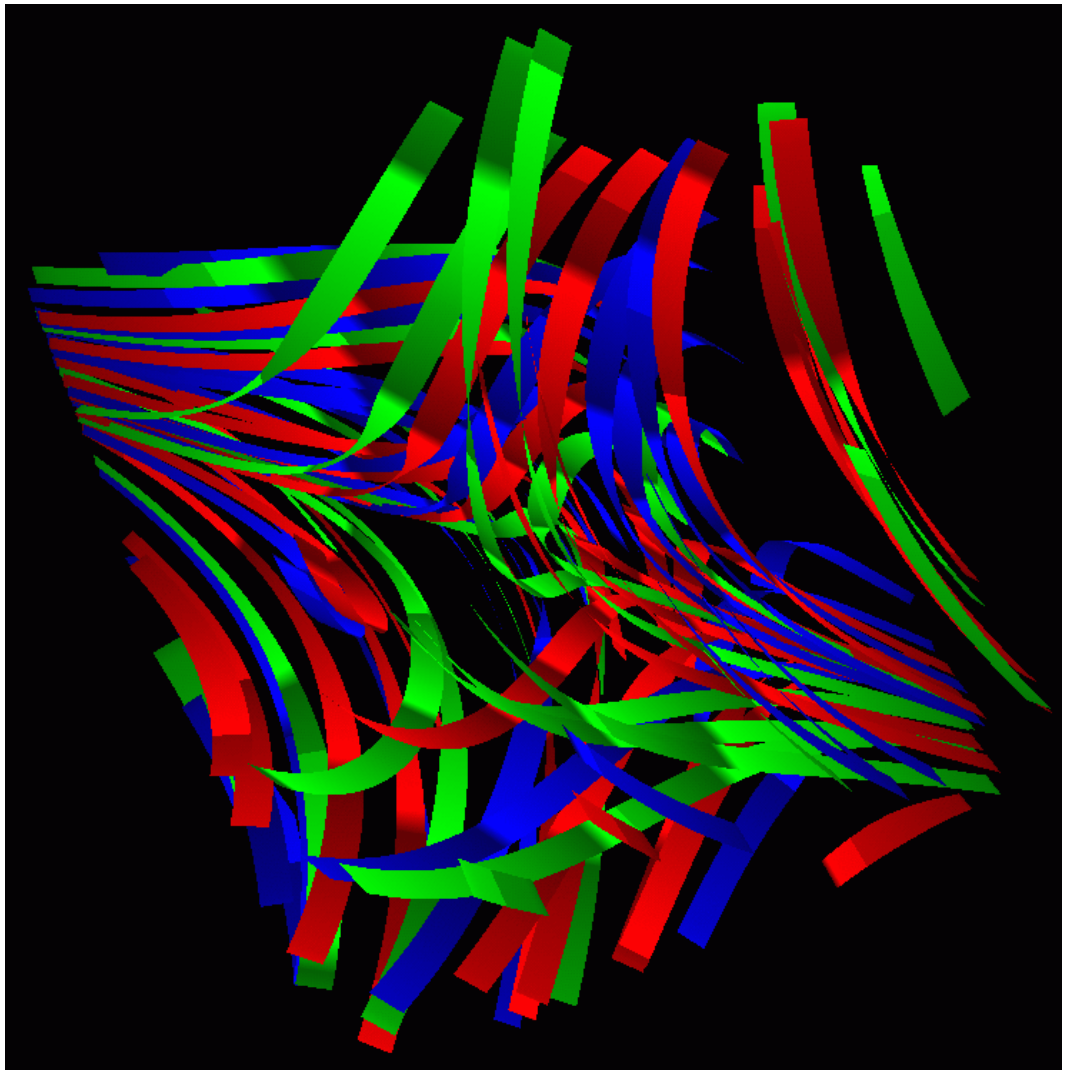
- Streamlines
  - Integration through vector field
- Stream ribbons
  - Connect two streamlines
- Streamtubes
  - Connect three or more streamlines
- Stream surfaces
  - Sweep line segment through vector field

# Streamlines Example

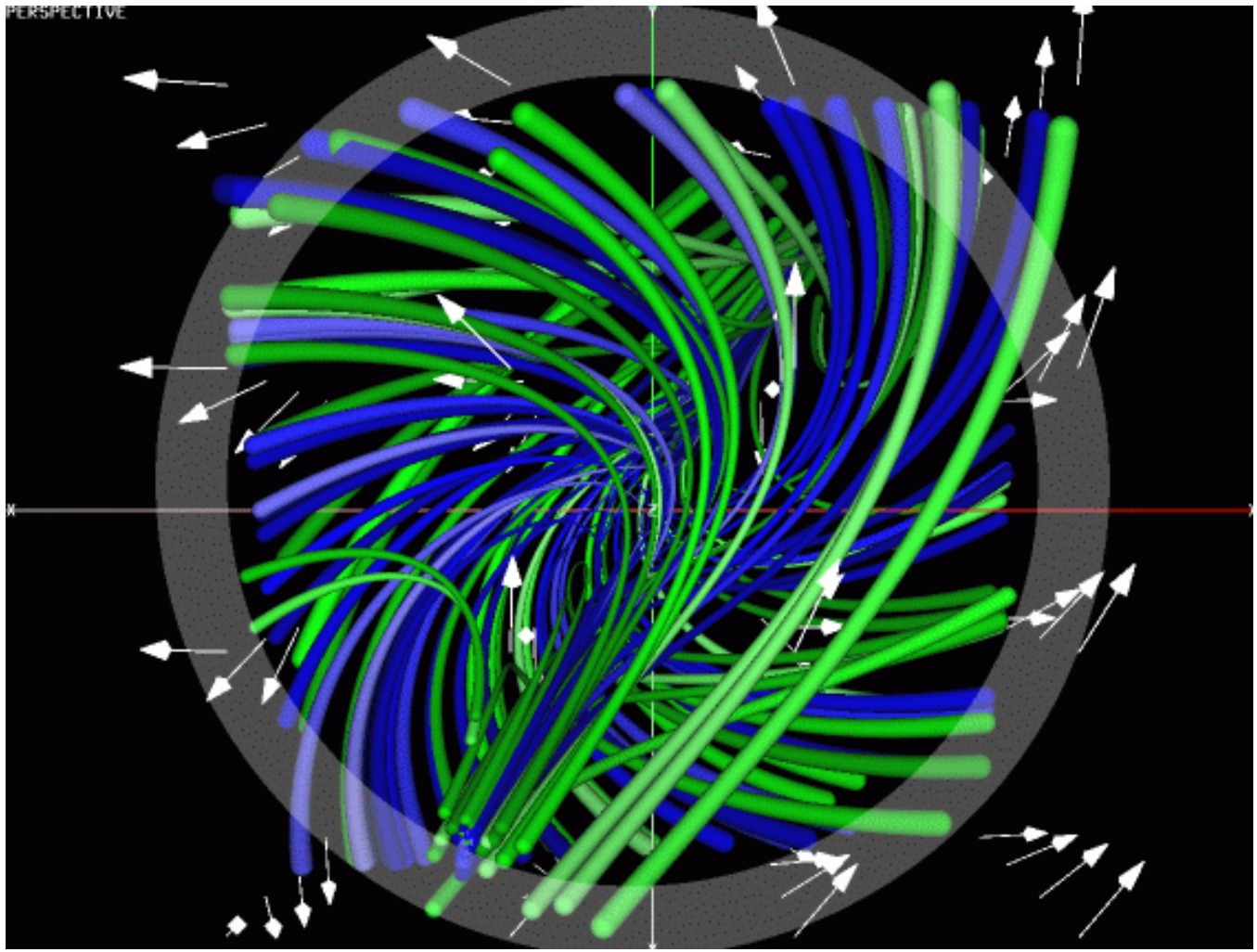


Color indicates temperature of air flowing through engine

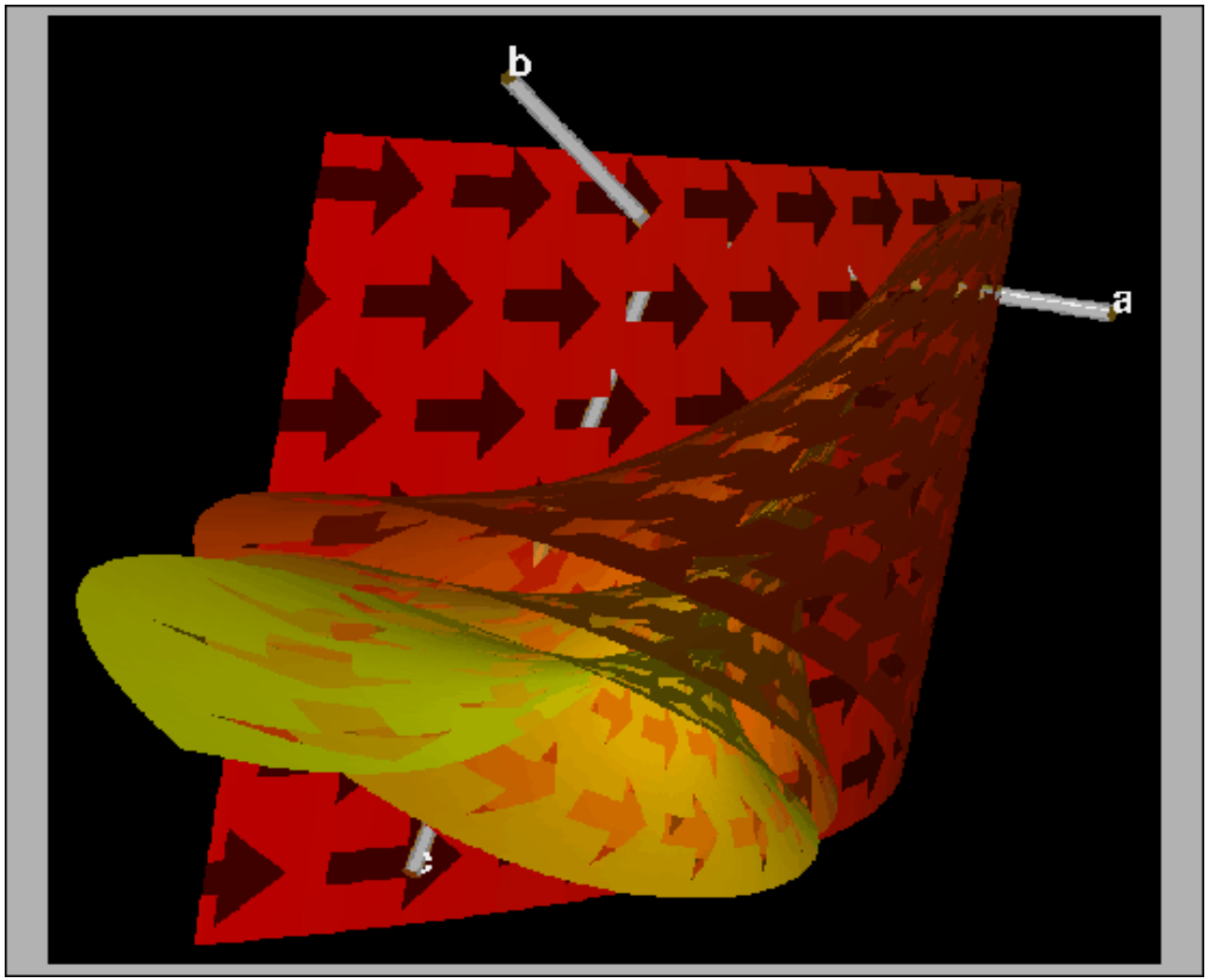
# Streamribbons Example



# Streamtubes Example



# Streamesurfaces Example

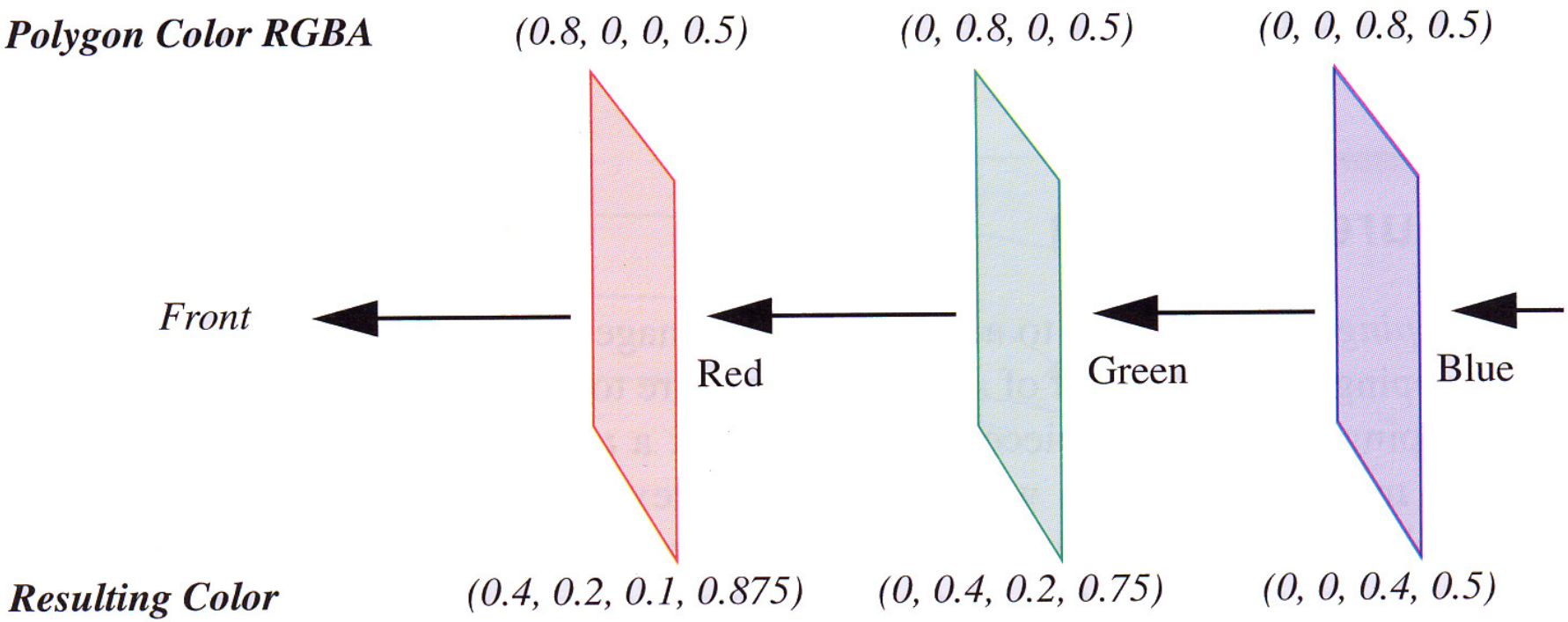


# Advanced Computer Graphics and Volume Rendering

# Transparency and Alpha Values

- Why is transparency useful?
- Peer into and through objects
- Volume visualization
- $\alpha$ = opacity
- $\alpha = 1 \rightarrow$ opaque
- Modern graphics hardware supports **alpha blending**
- Need to **composite** transparent actors
- Does order matter?
- Yes

# Alpha Compositing



# Alpha Compositing

$$R = A_s R_s + (1 - A_s) R_b$$

$$G = A_s G_s + (1 - A_s) G_b$$

$$B = A_s B_s + (1 - A_s) B_b$$

$$A = A_s + (1 - A_s) A_b$$

- *s* represents *surface* of actor
- *b* represents what is *behind* actor's surface
- Suppose $A_s = 0$? $A_s = 1$?

# Alpha Compositing Example

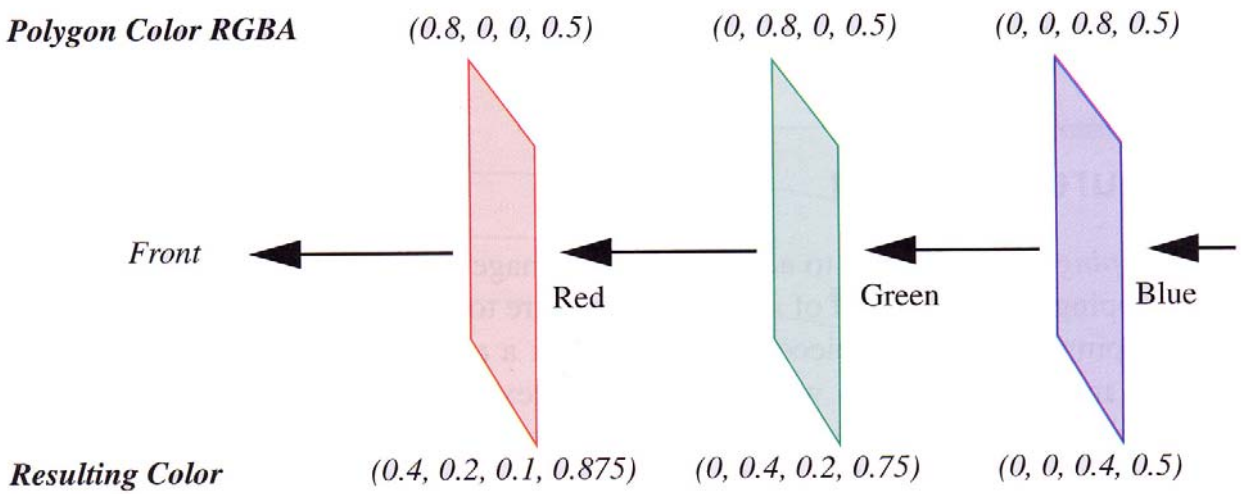
- Use $\alpha = 0.5$ for all 3 polygons and work through the calculations

$$R = A_s R_s + (1 - A_s) R_b$$

$$G = A_s G_s + (1 - A_s) G_b$$

$$B = A_s B_s + (1 - A_s) B_b$$

$$A = A_s + (1 - A_s) A_b$$

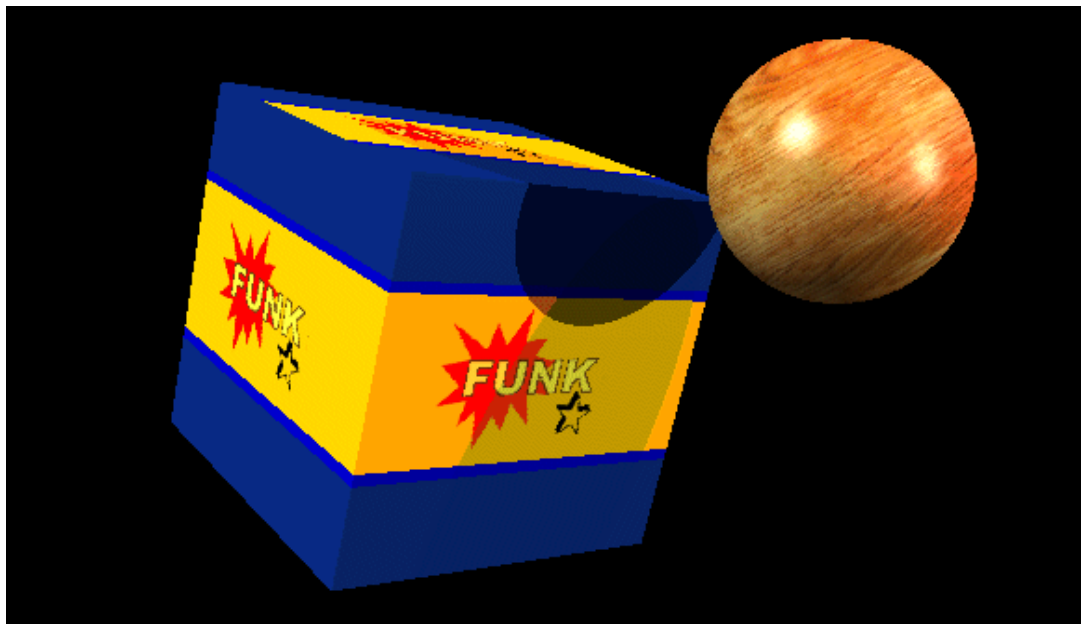


# Compositing Order Matters!

- Recall the *z buffer* algorithm, which is used for…?
- Will not necessarily composite polygons in right order
- Usually must use software to order actors by their increasing distance from camera

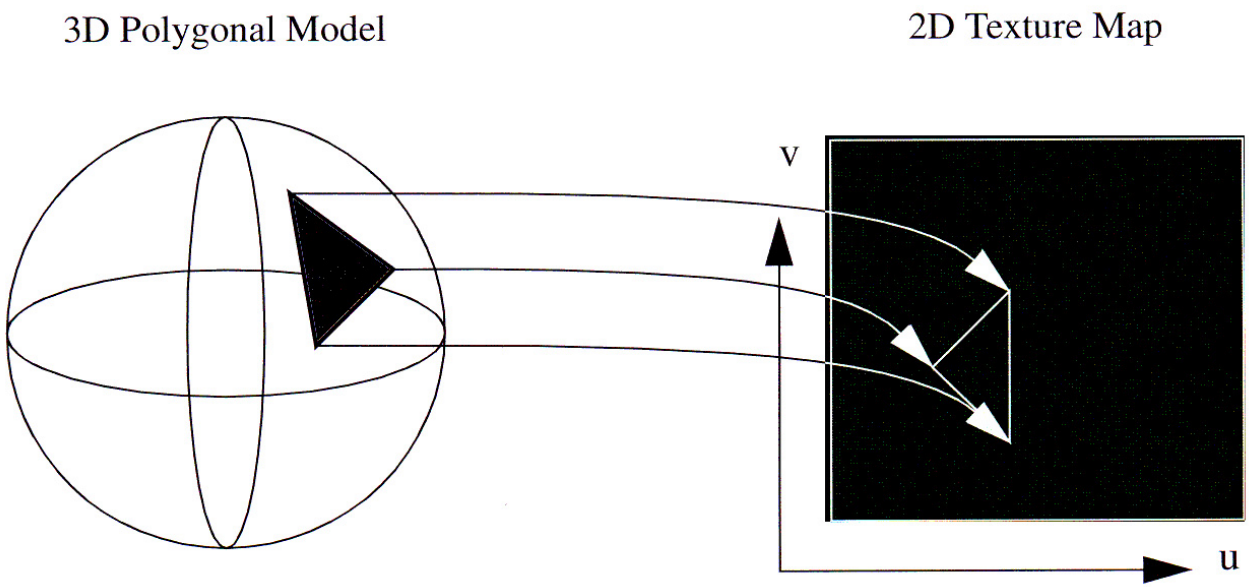
# Texture Mapping

- Idea: add detail to image without requiring modeling detail
- Map picture called a **texture map** onto object
- **Texture coordinates** tell you where on object to put picture



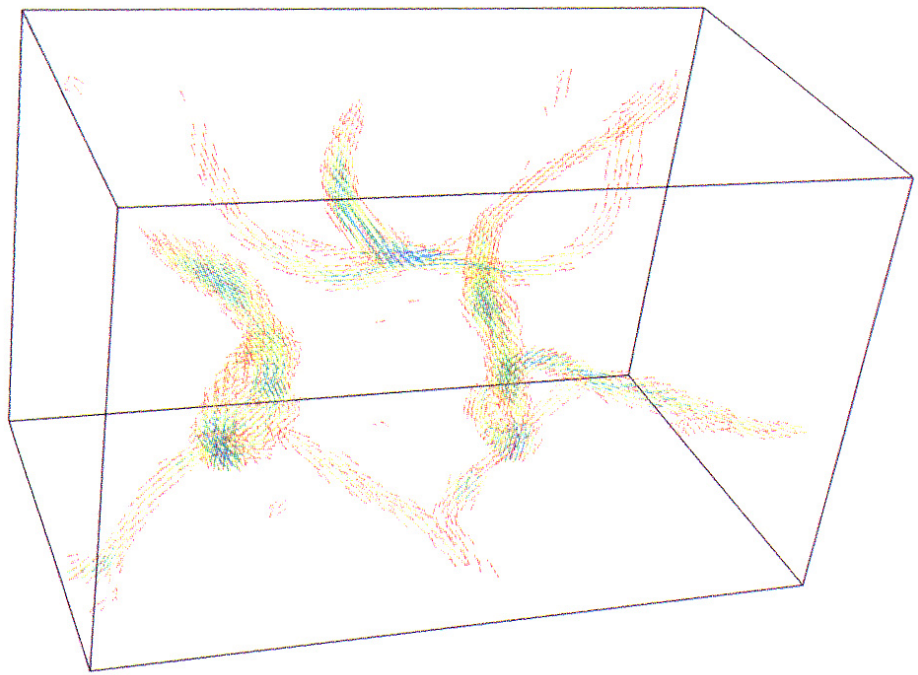
# Texture Mapping

- 2D texture mapped onto 3D geometry
- Each 3D vertex assigned 2D texture coordinates, usually written (*u*,*v*)
- Texture is an RGBA image made of **texels**, texture elements



# Texture Mapping in Visualization

- Animated texture maps
- Flow visualization
- Colors cycle in a loop to show direction of flow

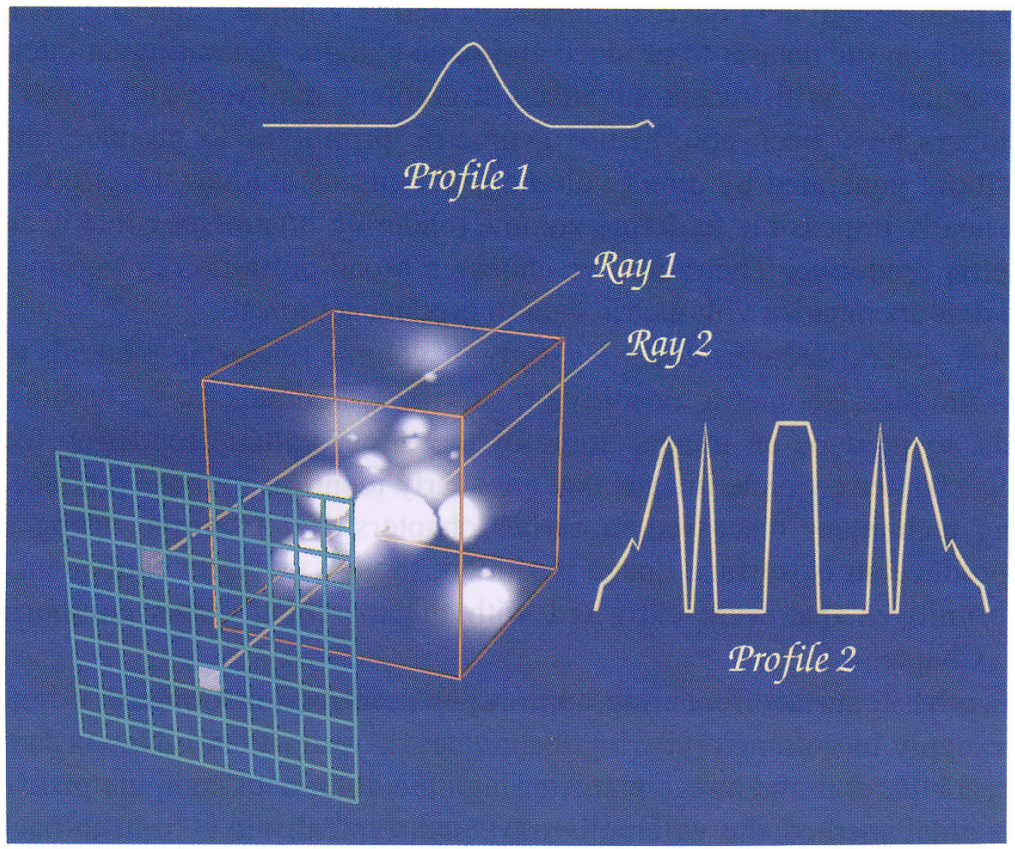


# Volume Rendering

- Image-order and object-order volume rendering
- Ray-casting vs. splatting

# Ray Casting

- Idea: send *viewing ray* into volume and examine data encountered to compute pixel's color

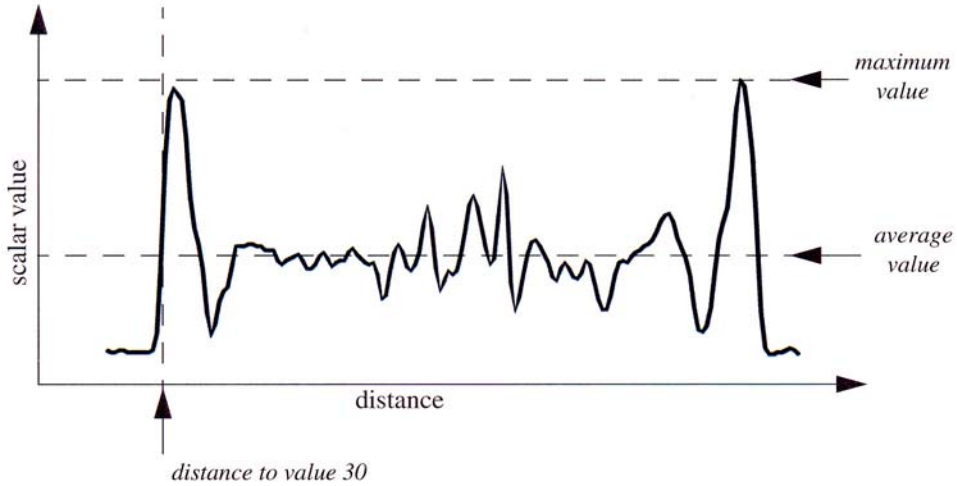


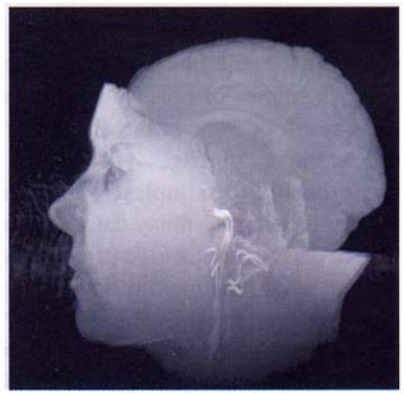
# Ray Casting

- Each ray has a different profile we can draw as a 2D curve
- Essentially we will numerically integrate (?) the curve
- Material density, illumination parameters, other attributes affect this integration

# Ray Profile Example

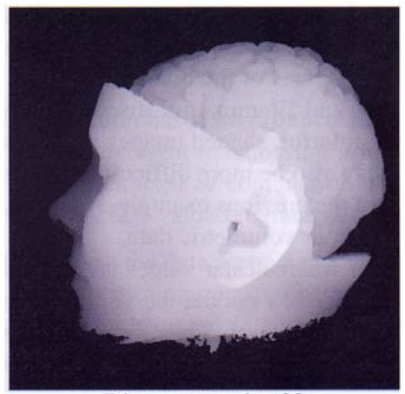
- 8-bit density volume
- Range: 0…255
- x-axis: distance from view plane
- y-axis: density
- image 3: distance to first voxel with 30+ density value
- image 4: alpha compositing



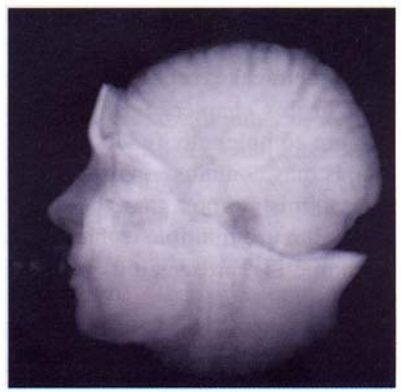



Maximum value

Average value



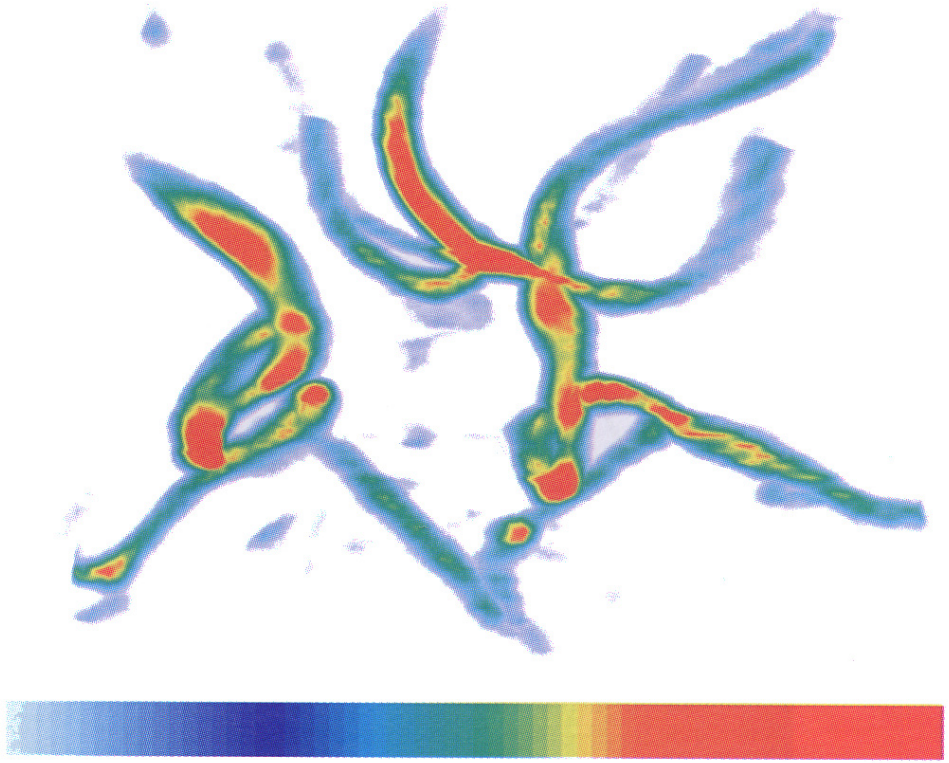
Distance to value 30



Composite

# Maximum Intensity Projection

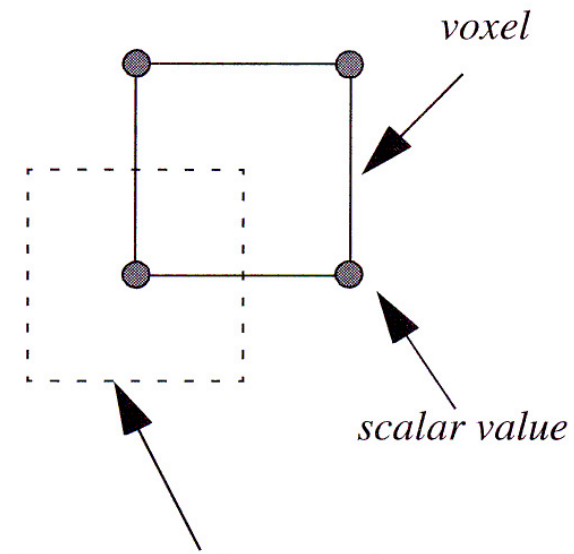
- MIP simple yet effective technique
- Depth perception lost, though
- Can do colored MIP also
- Which blood vessel is in front of the others?
- No compositing, so colors don't blend together



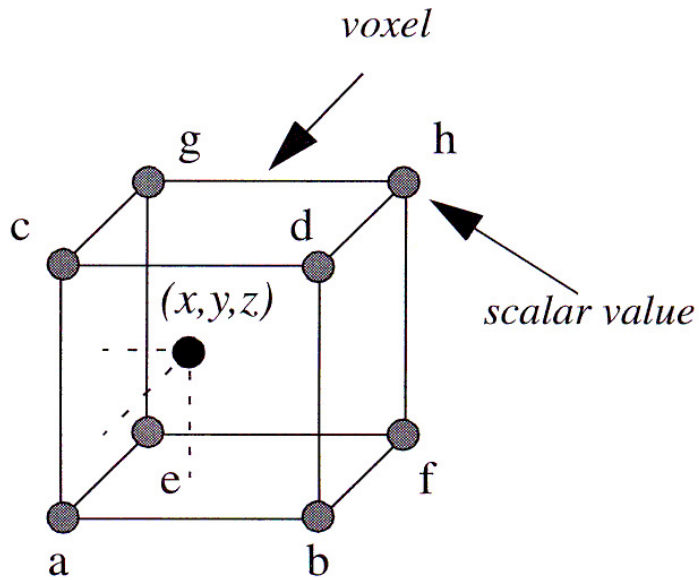
# Ray Traversal

- We take small steps along the ray
- Don’t always land on a voxel
- Need to estimate density somehow (?)
- Interpolation!
- **Nearest neighbor interpolation**: just find closest voxel and use its density
- **Trilinear interpolation**: take some weighted sum of 8 nearest voxels’ densities

# Interpolation Techniques



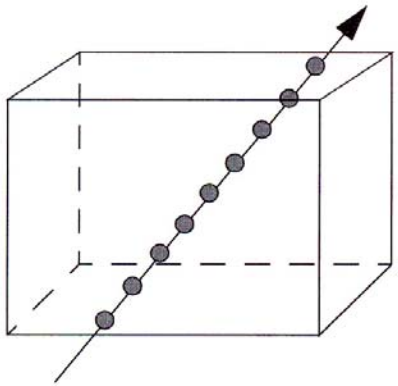

Nearest neighbor region
of uniform value



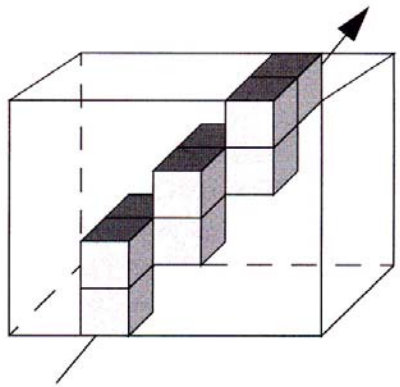

value at *(x,y,z)* derived from eight surrounding scalar values

# Ray Traversal

- Usually we traverse the ray at uniform intervals



Uniform sampling



Voxel by voxel traversal

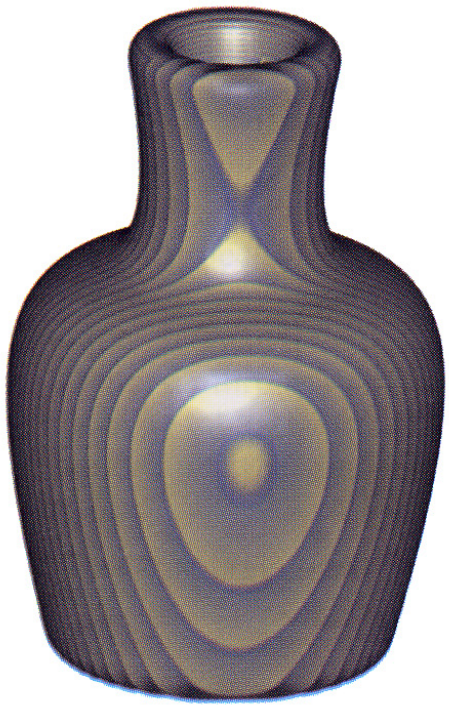
- Parametric form: $(x, y, z) = (x_0, y_0, z_0) + (a, b, c)\, t$
- $(x_0, y_0, z_0)$ is the origin of the ray
- $(a, b, c)$ is the normalized ray direction vector

# Ray Traversal Pseudocode

```
t = t1;
v = undefined;
while ( t < t2 )
   {
   x = x0 + a * t;
   y = y0 + b * t;
   z = z0 + c * t;
   v = EvaluateRayFunction( v, t );
   t = t + delta_t;
   }
```

- t1 and t2 are distances where ray enters and leaves volume, respectively

# Step Size Affects Image Quality



Step size = 2.0

Step size = 1.0

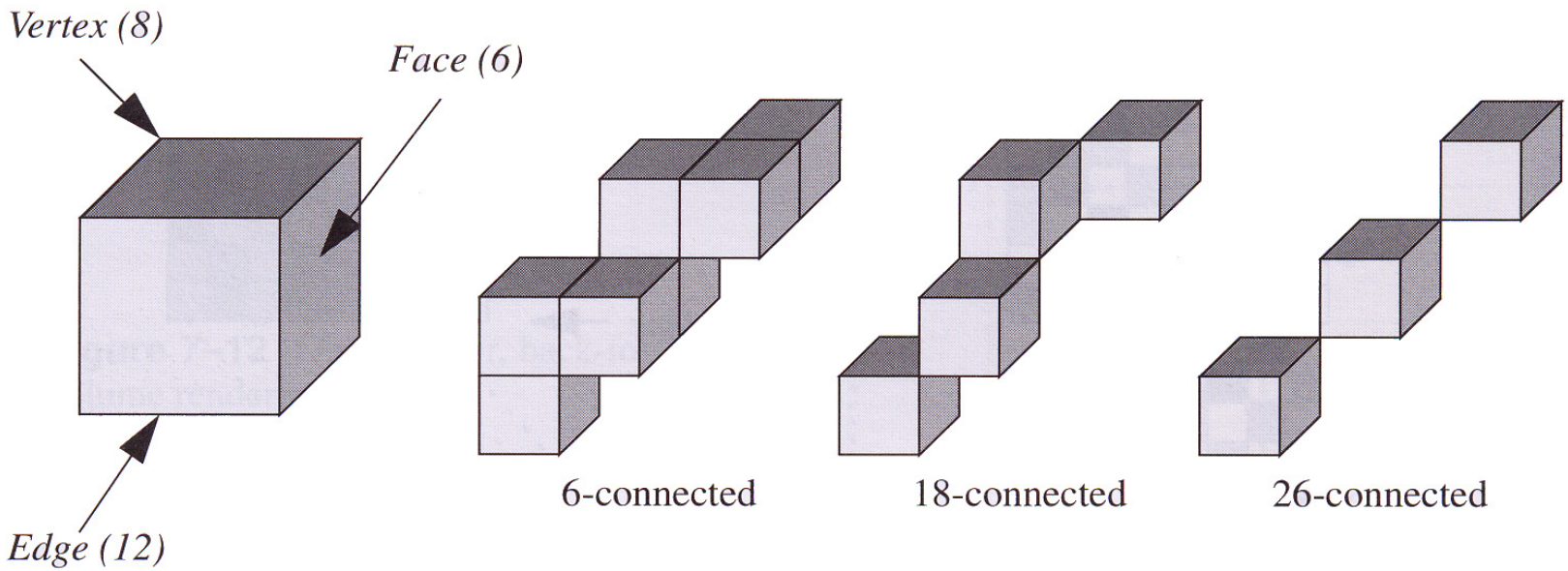
Step size = 0.1

# Step Size

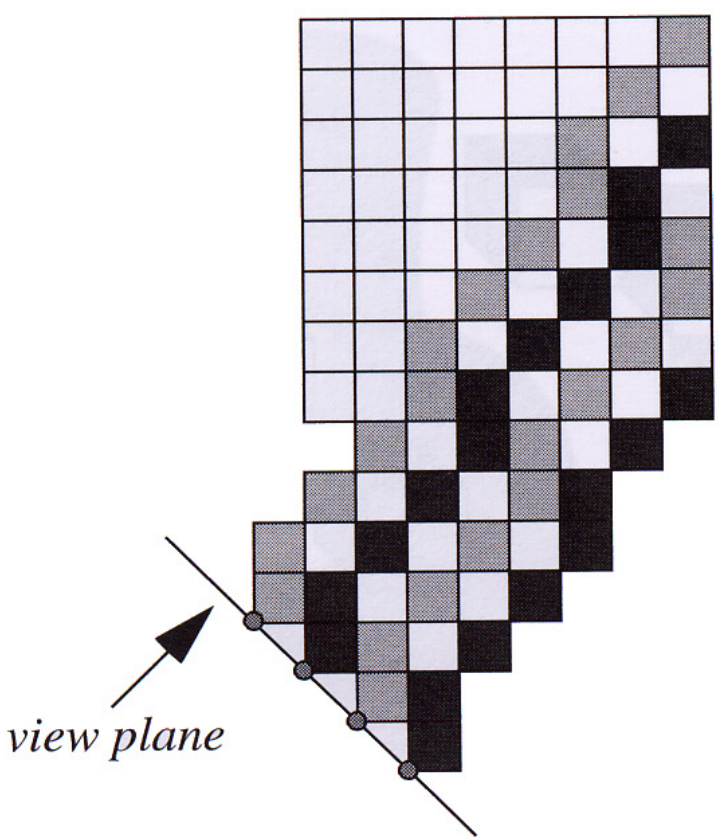
- Small step size = higher quality, slow speed
- Large step size = converse
- Large step size causes the banding effect

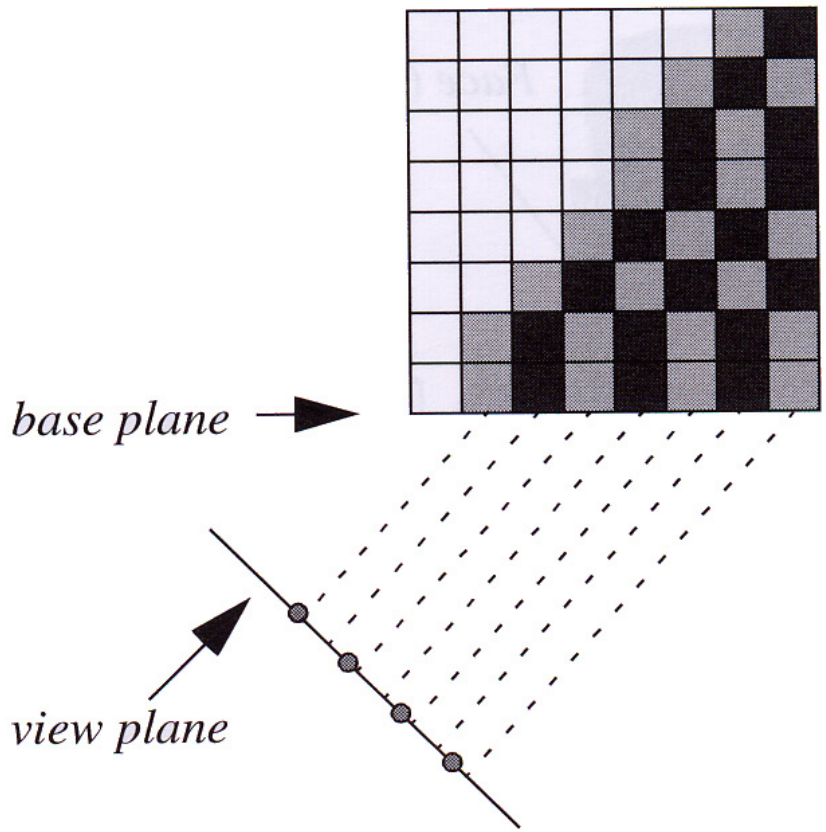
# Voxel-based Ray Traversal

- Jump from one voxel to another instead of along a continuous ray
- Related to concept of **connectedness**



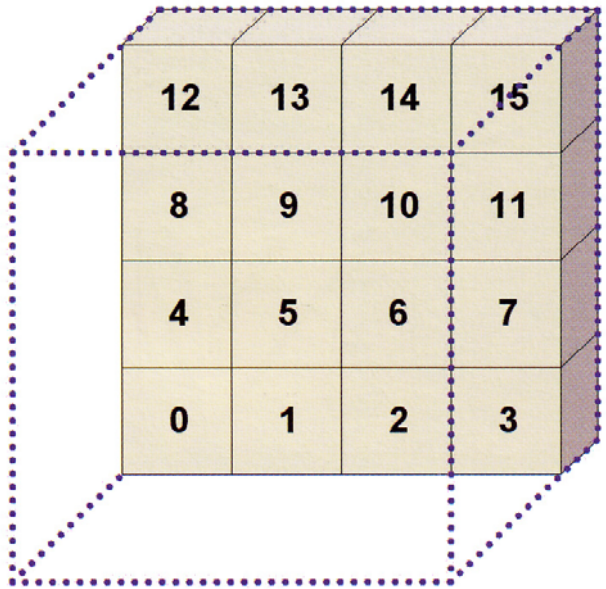
# Voxel-based Ray Traversal



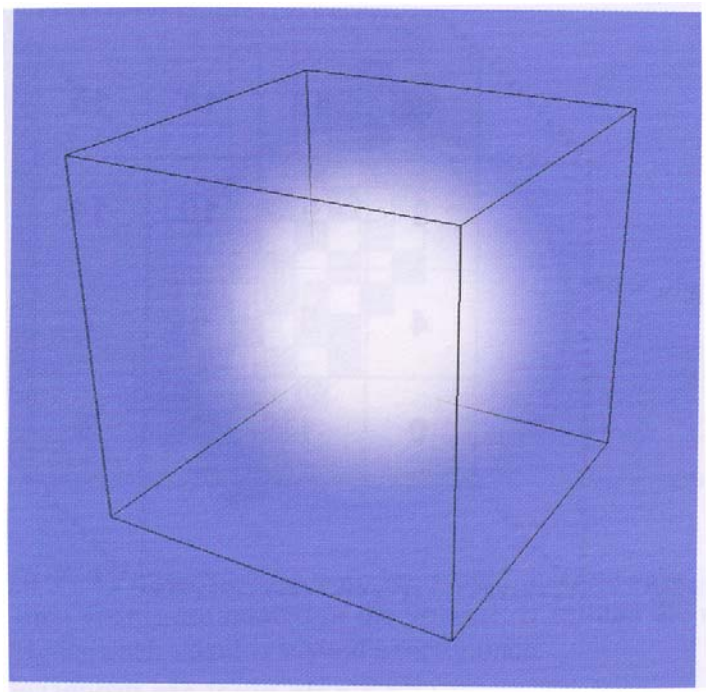
# Object-Order Volume Rendering

- Back-to-front or front-to-back processing of voxels
- Requires a triply nested loop
- **for** z = … {
  **for** y = … {
  **for** x = … {
  …
  }
  }
  }
- Select plane most parallel to image plane



# Splatting

- Fuzzy sphere (called the kernel) placed around each voxel
- Kernel projected onto viewing plane, producing a **footprint**
- Repeat for all voxels
- Kernel size affects image quality
- Footprint *discretized* to a resolution appropriate for image resolution



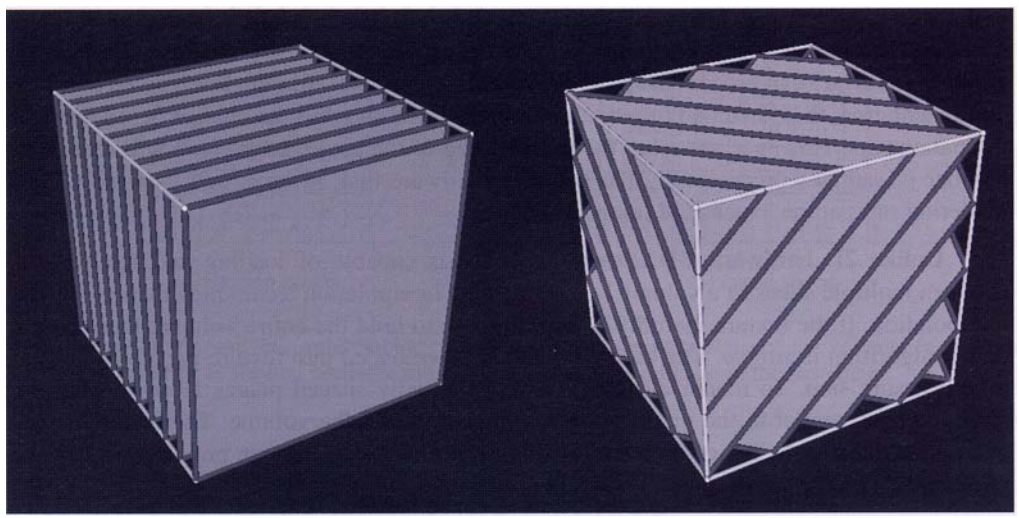
# Implementing Splatting

- Software-only vs. hardware-assisted
- Footprint table – slices generic kernel into image-aligned slabs

# Texture Mapping-based Volume Rendering

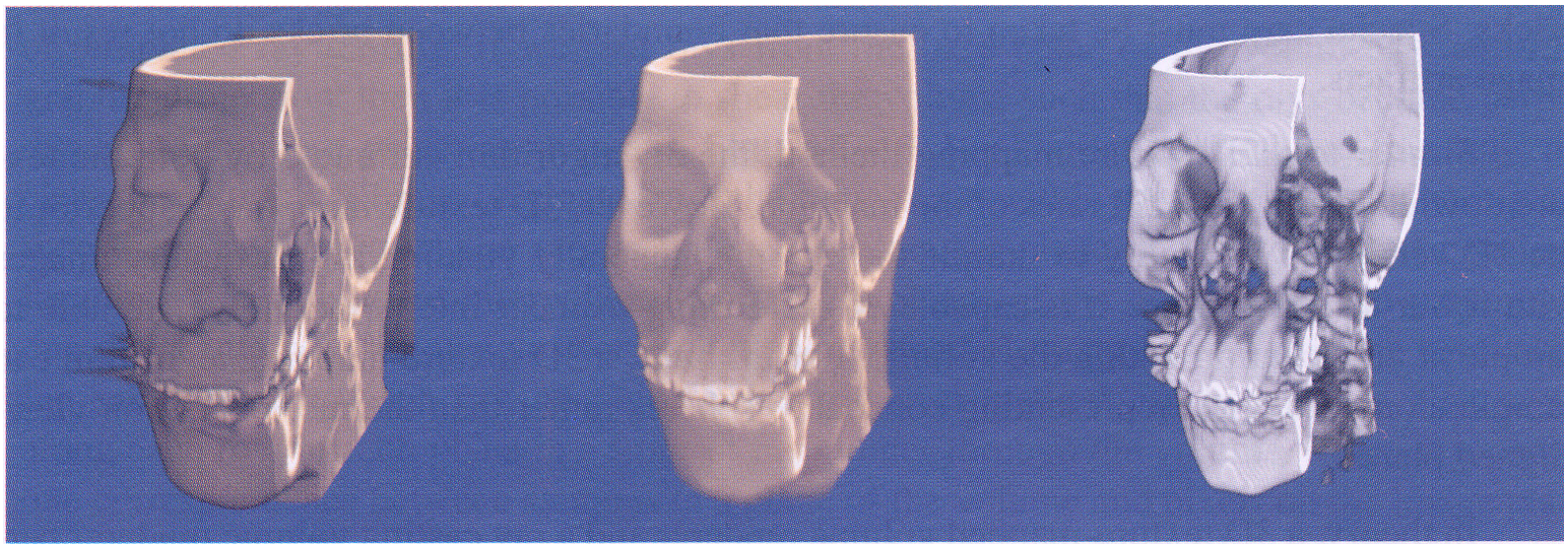
- In 2D: project and composite *axis-aligned* slices onto image plane
- In 3D: cut volume into slices that are *parallel* to the image plane ("image-aligned slices")
- Use interpolation and compositing in both cases

using 2D texture mapping hardware



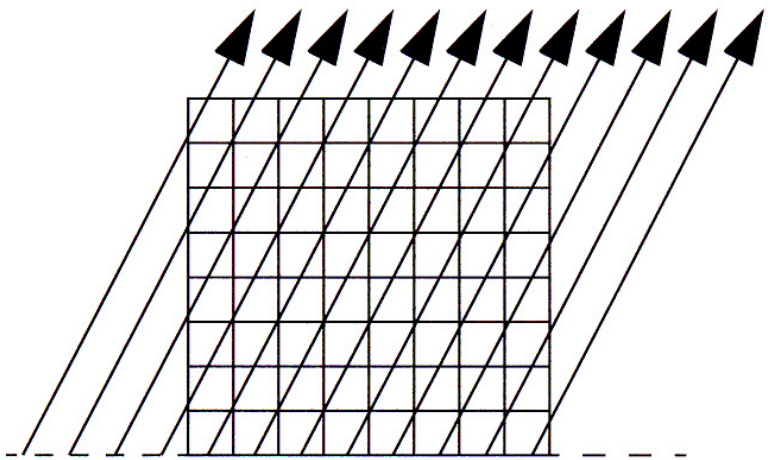
using 3D texture mapping hardware

# 2D Texture-Mapped Volume Rendering Example

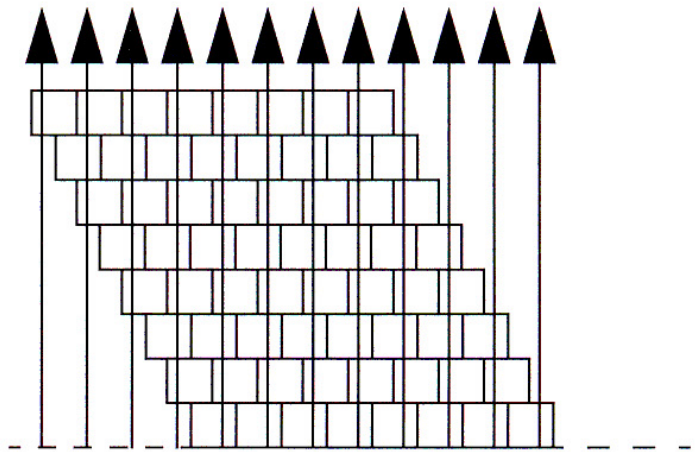


# Shear-Warp Volume Rendering

- Hybrid technique – aspects of object-order and image-order rendering
- Idea: convert a rotation of the camera into a *shearing* of the volume



Cartesian space



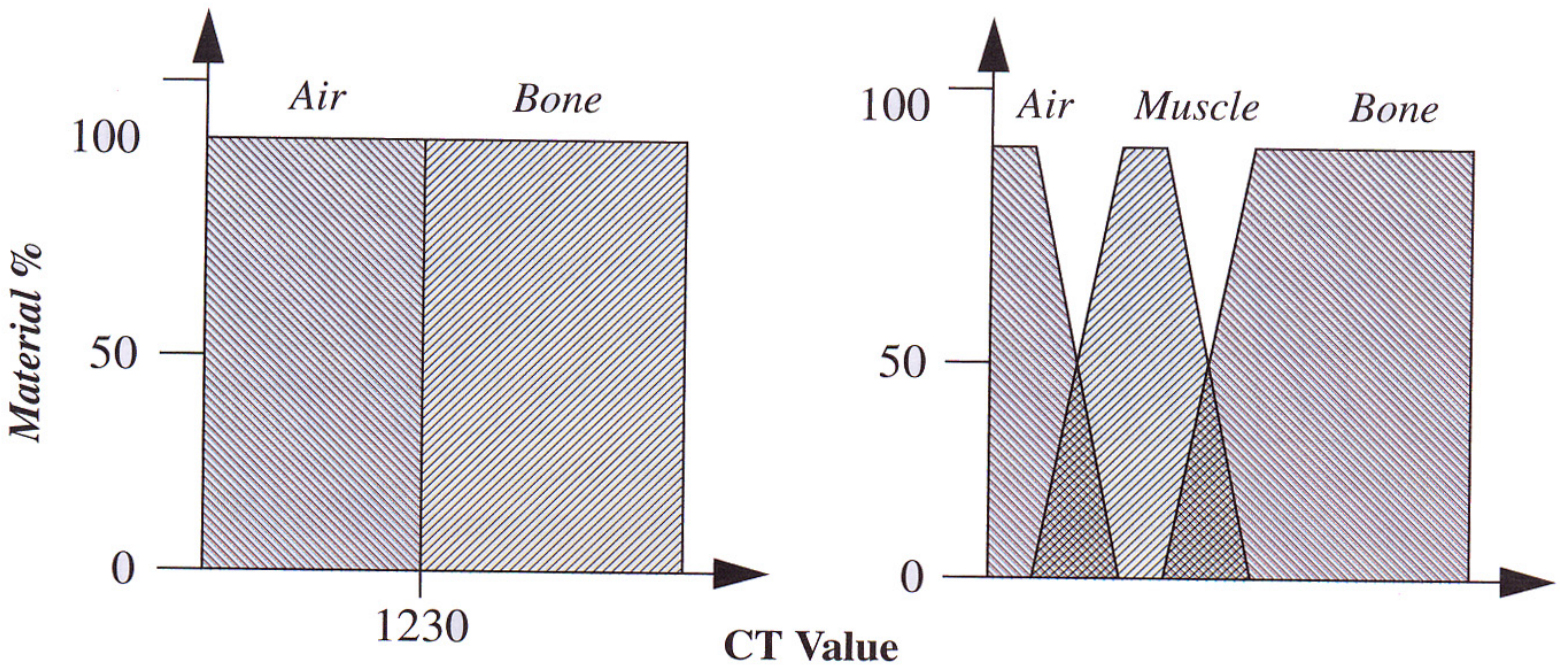
Sheared space

# Shear-Warp Volume Rendering

- Need to use **bilinear interpolation** to resample the slices
- Front-to-back ray traversal
- Essentially a very efficient form of ray-casting
- Downside?  Extra interpolations introduce error and hurt image quality
- Requires 3 copies of the volumes so we can shear volume along direction most parallel to image plane
- Shear in xy-plane, xz-plane or yz-plane
- Need to be able to process raster in any order

# Volume Classification

- Assignment of density ranges to categories
- Represented by transfer functions
- "Material percentage" transfer functions:



# Volume Classification

- Usually we classify a volume using red, green, blue and opacity transfer functions
- Two possibilities to apply classification during ray traversal:
  1. Interpolate voxel densities and then compute color
  2. Assign colors to voxels and then interpolate colors
- Option 1 tends to make nicer looking images

# Volume Classification

- We can also compute the **gradient** of the density field and use that to modulate the color
- Gradient is a vector that tells you how the material is changing at a position
- Vector of first partial derivatives in x, y and z:

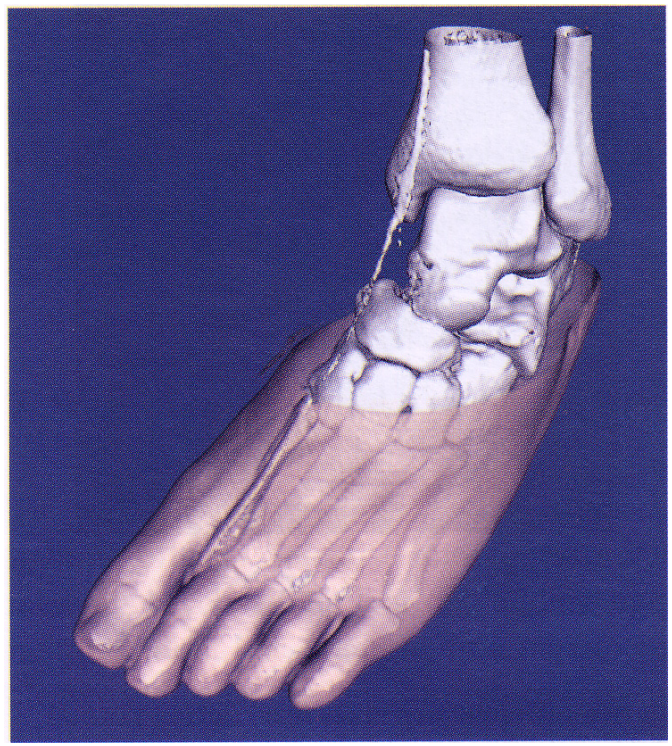
$$g_x = \frac{f(x + \Delta x, y, z) - f(x - \Delta x, y, z)}{2\Delta x}$$

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

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

# Volume Classification

- If the vector **g** is long, that means the material is changing quickly
- Example: boundary between bone and flesh
- Implies presence of a surface
- Modulate color based on gradient magnitude to ignore regions of **homogeneous** material distributions
- Small magnitude = little or no change of material



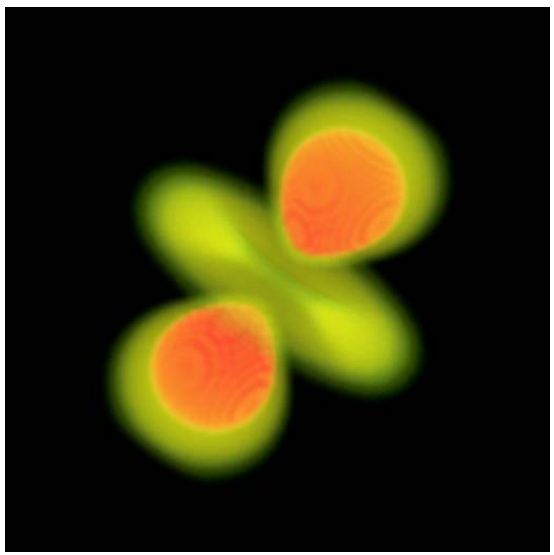
# Uses of the Gradient Vector

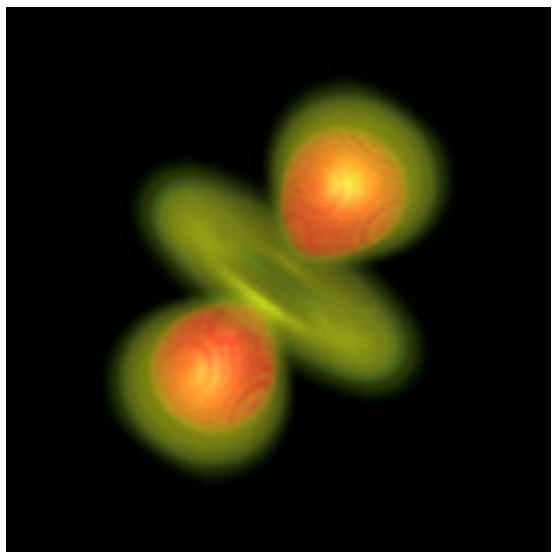
- We can treat the gradient as a normal vector and evaluate the lighting equation to shade and illuminate volumes

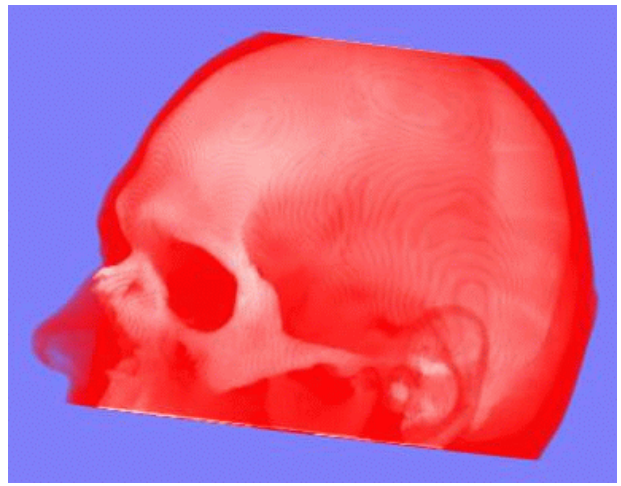
Maximum intensity | Composite (unshaded) | Composite (shaded)

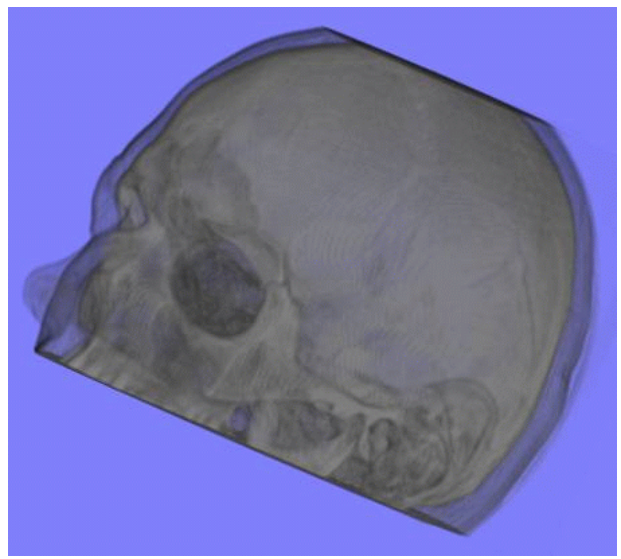
# Volumetric Shading

- Can reveal surfaces inside the data
- Compare:









# Volumetric Shading: How?

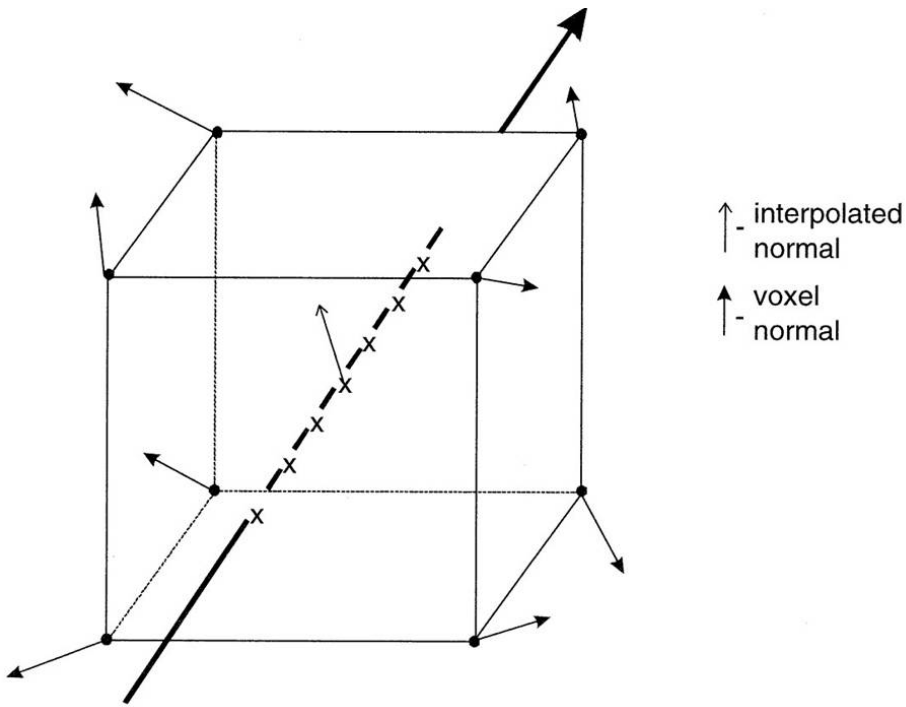
- Gradient allows us assigns a direction vector to each voxel
- This (normalized) vector is used just like the normal vector in surface graphics
- It will modulate the color we assign to samples and thereby allow us to create 3D effects
- Look at the skull on the right

# Volumetric Shading

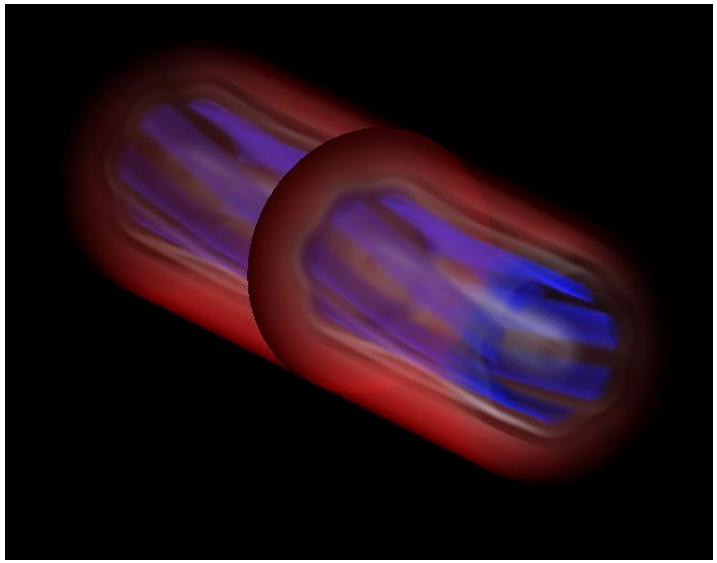
- But how do we incorporate color, opacity and shading information?
- First we interpolate the density at a given sample position
- Interpolate gradient at same position
- Then assign color and opacity to each sample, and shade using interpolated gradient
- When we shoot the rays through the volume, we have to composite all these samples together

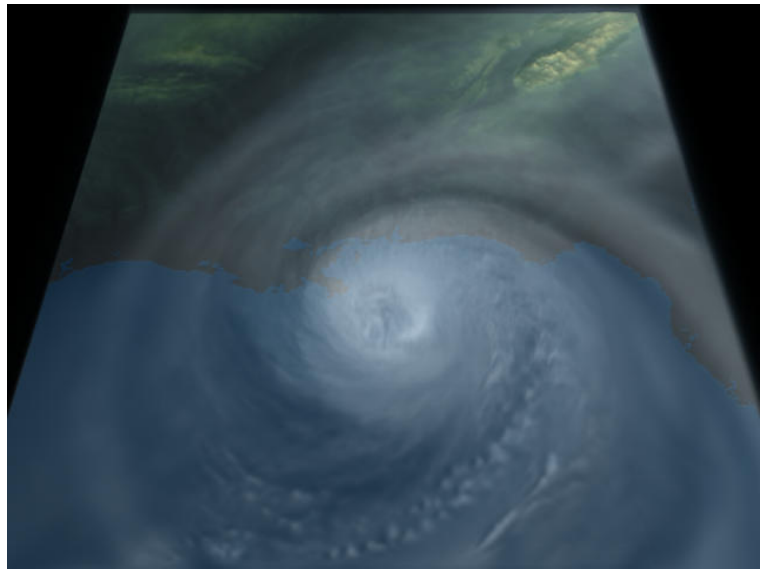
# Gradient Interpolation

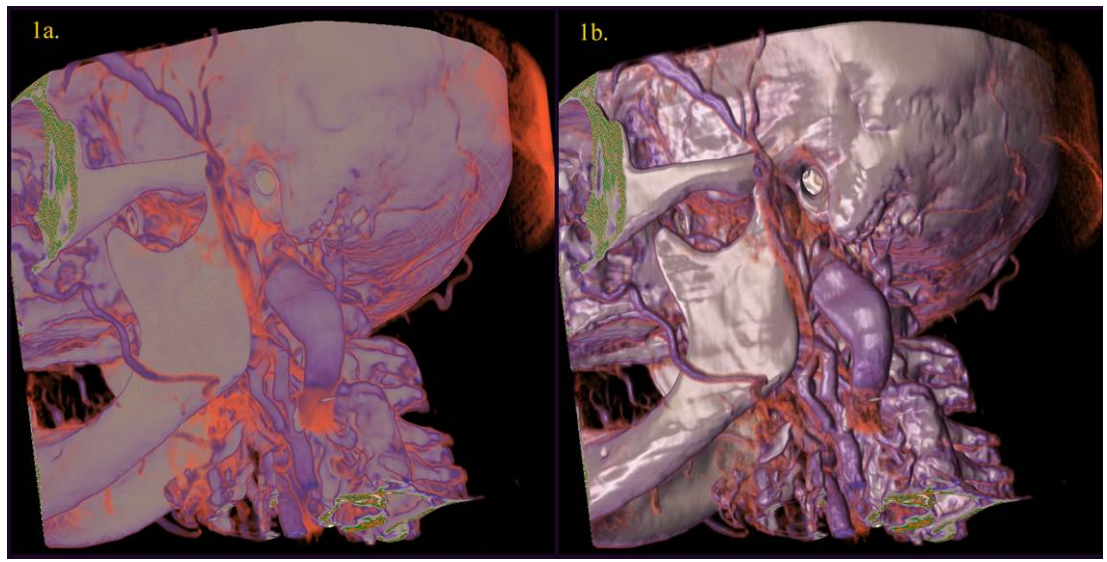
- At start of processing, compute gradient at each voxel
- During ray traversal, estimate gradient with trilinear interpolation
- Like densities (and unlike colors), gradients are **intrinsic** attributes of models



# More Volumetric Shading Examples

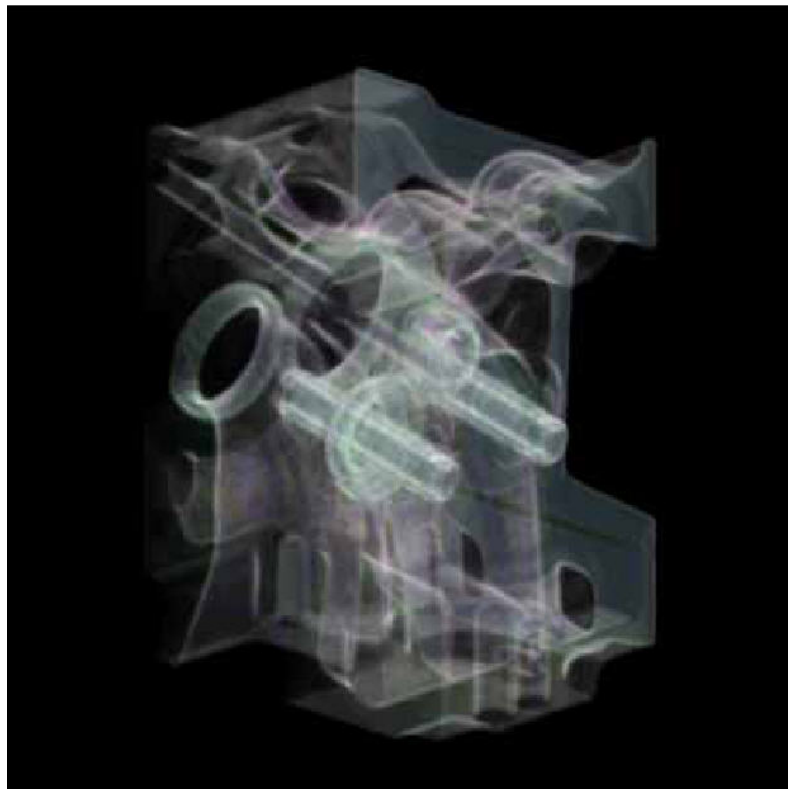






# Gradient Modulation

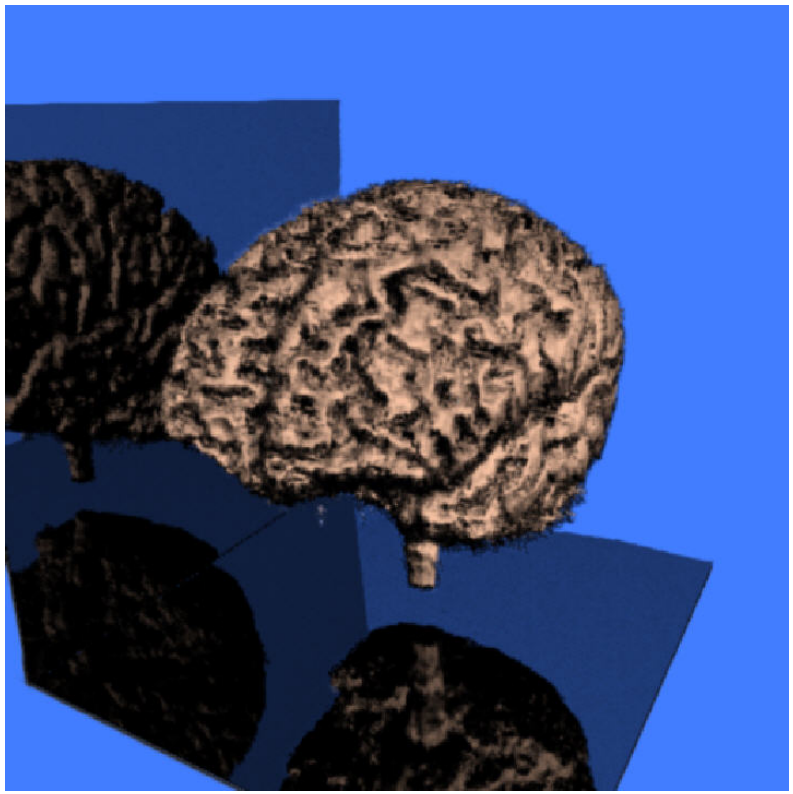
- With **gradient modulation** in we modulate opacity/color of a voxel by gradient
- We multiply opacity and color by some function of gradient magnitude (or given by a transfer function, #5)
- Regions of high gradient magnitude increase opacity; regions of low gradient magnitude decrease opacity

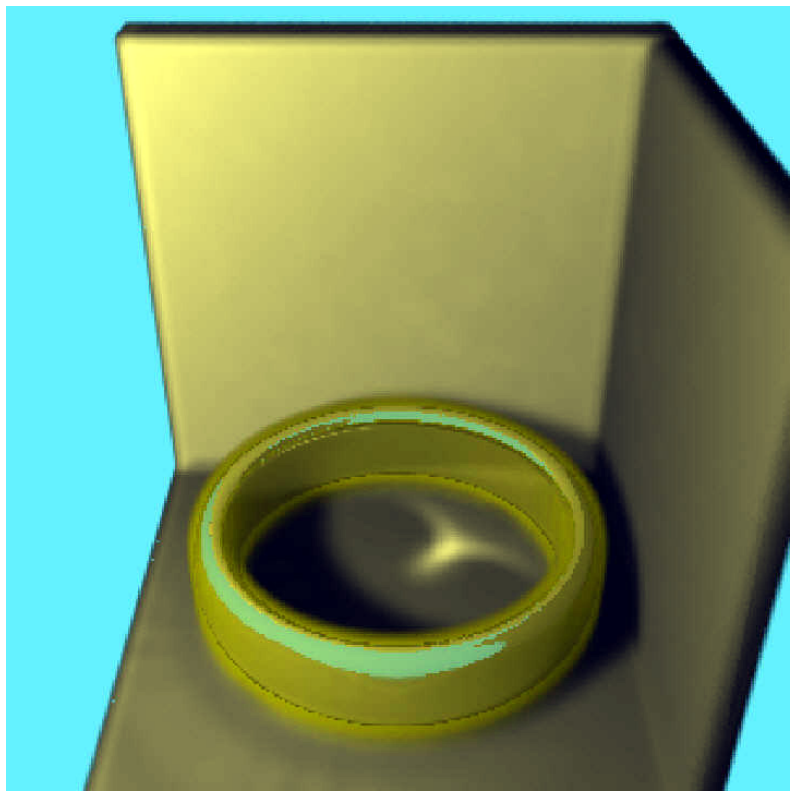


- Explain this image

# Volumetric Global Illumination

- Global illumination refers to reflections, shadows and other effects that cannot be computed locally





# Regions of Interest

- An ROI is simple a portion of the data-set of particular importance
- Use **cropping planes** to reveal interior
- Simple idea, but very, very useful
- Eliminates set of voxels from consideration

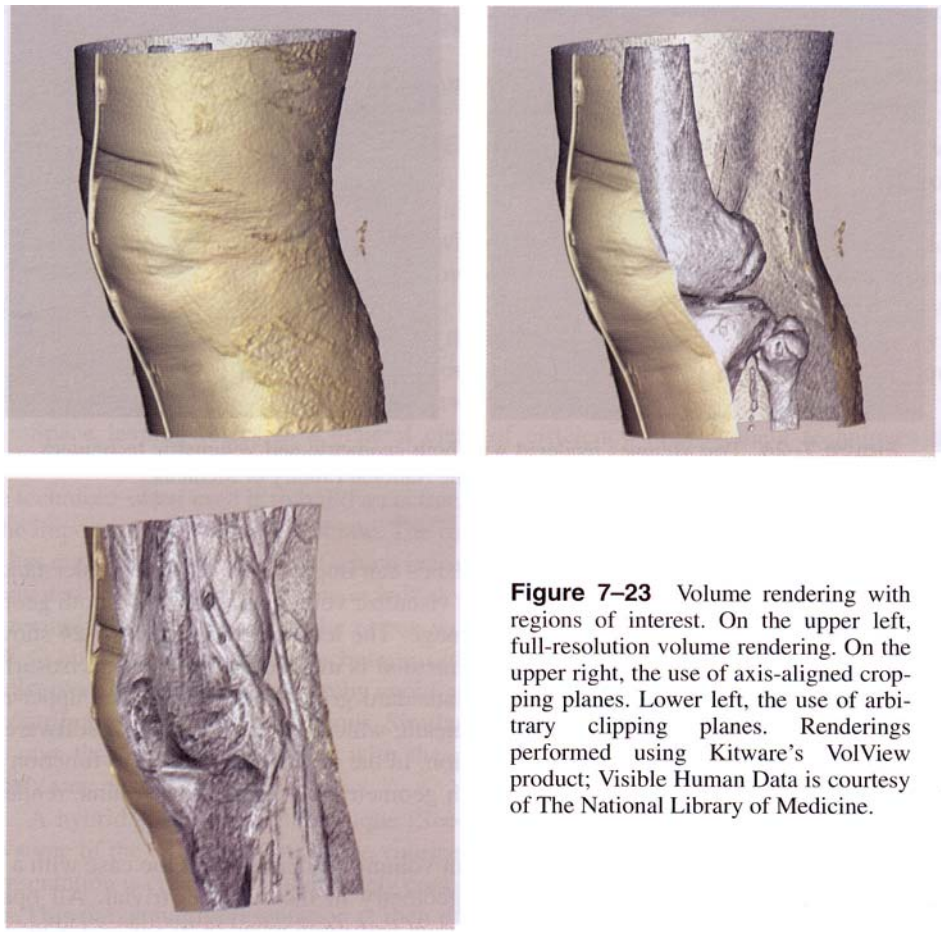


**Figure 7–23** Volume rendering with regions of interest. On the upper left, full-resolution volume rendering. On the upper right, the use of axis-aligned cropping planes. Lower left, the use of arbitrary clipping planes. Renderings performed using Kitware's VolView product; Visible Human Data is courtesy of The National Library of Medicine.

# Image Processing Primer

# Image Processing

- Operations performed over images (2D or 3D)
- Purpose:
  - enhance certain features
  - de-emphasize other features
- Implemented as *filters* or transformations:
  - some operate on the entire set of pixels at once (global operations)
  - examples: brightness and contrast enhancement

# Image Processing

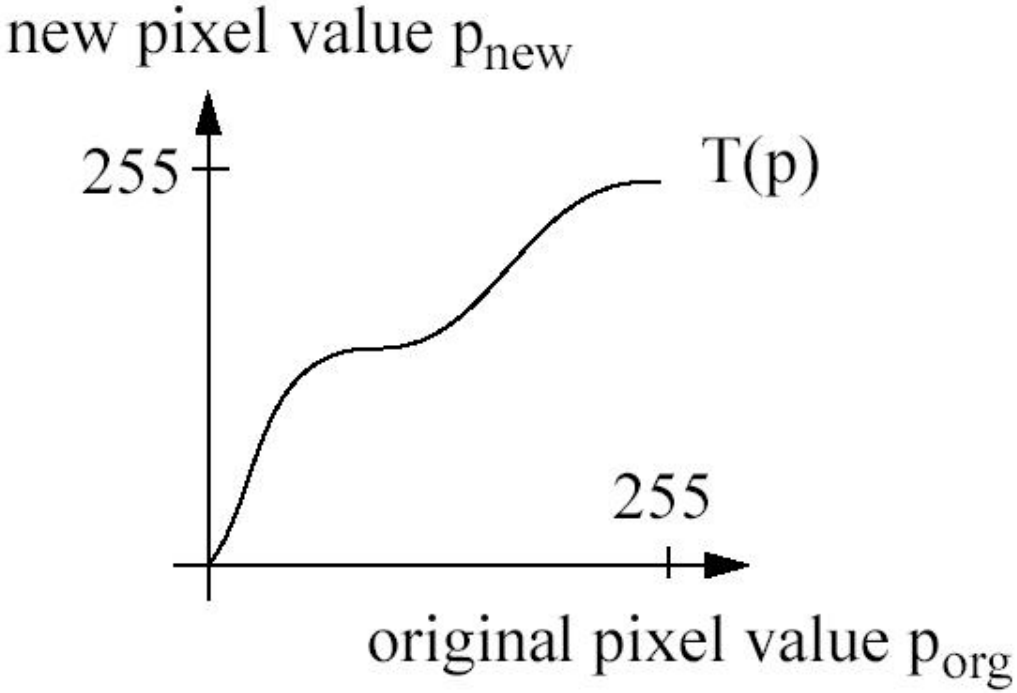
- Some operate only on a subset of pixels (local operations in a pixel neighborhood)
- Examples: edge detection, contouring, image sharpening, blurring, “noise” reduction

# Intensity Transformations

- Modify distribution of gray levels in an image
- Example: reduce number of grayscale levels used to represent images
- Reasons: memory, display/printing limitations, cost
- Reduce number of bpp (bits per pixel) (e.g., 24 → 8 bits)
- Usually intensity transformations used for **image enhancement**

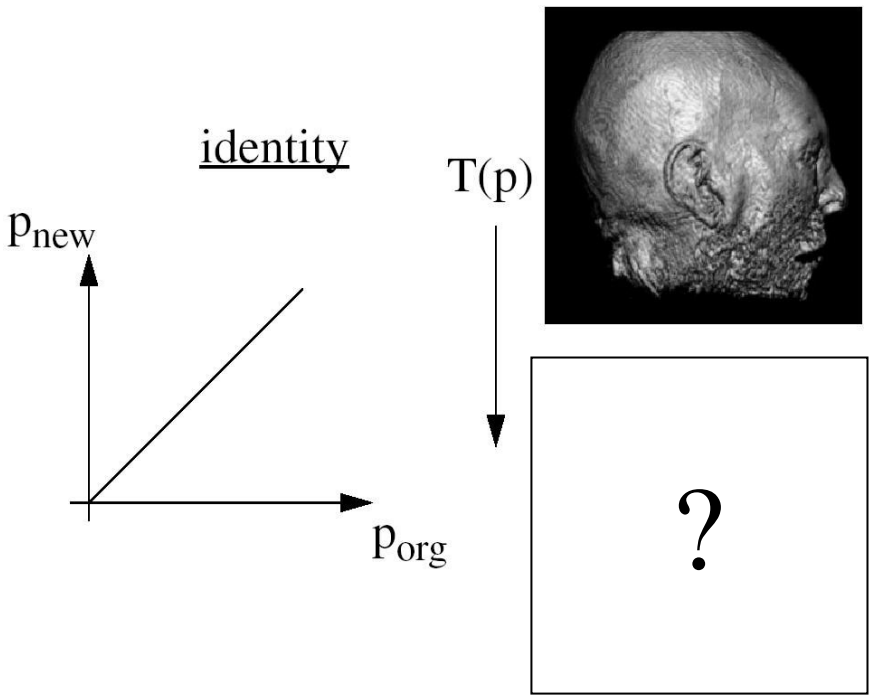
# Intensity Transformations

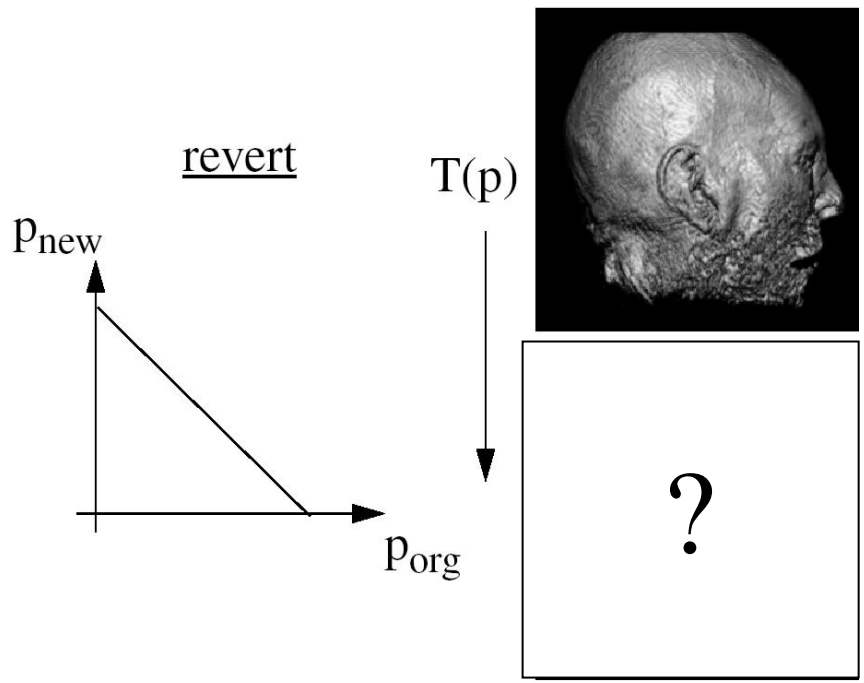
- An intensity transformation most easily expressed as function T(p) over domain of possible pixel intensities
- New pixel intensity given as height of function



# Intensity Transformation Examples

- What would happen to the image in each case?

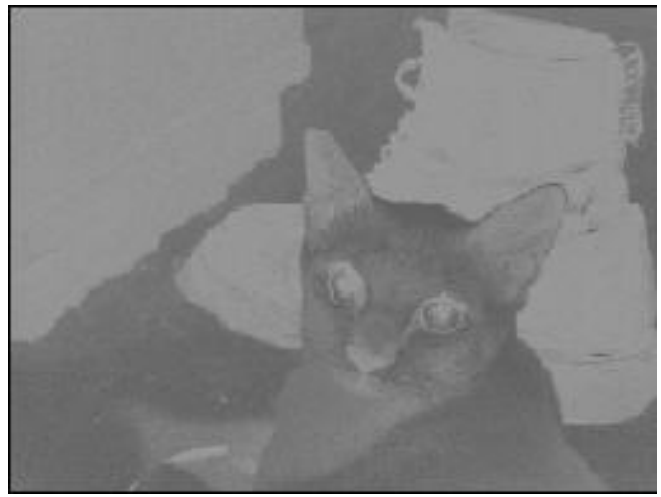


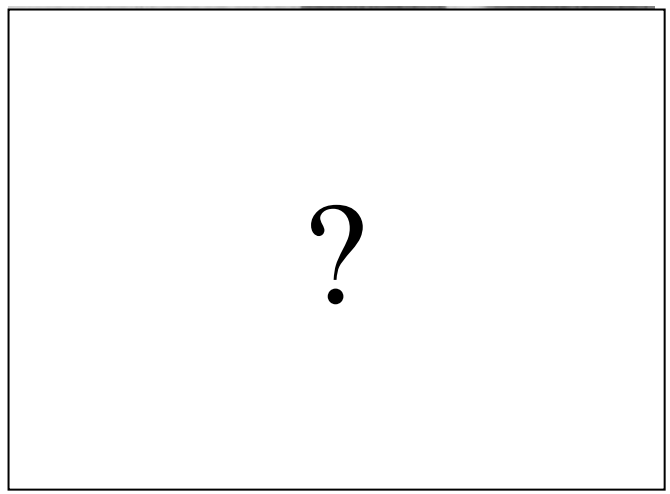




- What does the right-bottom image look like?

# Contrast Enhancement

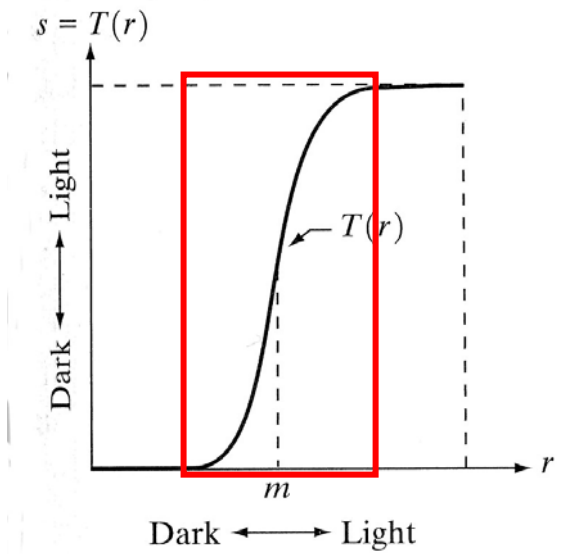
- Often one is given an image with poor contrast
- Image seems washed out and features are hard to see
- Need to enhance the contrast somehow





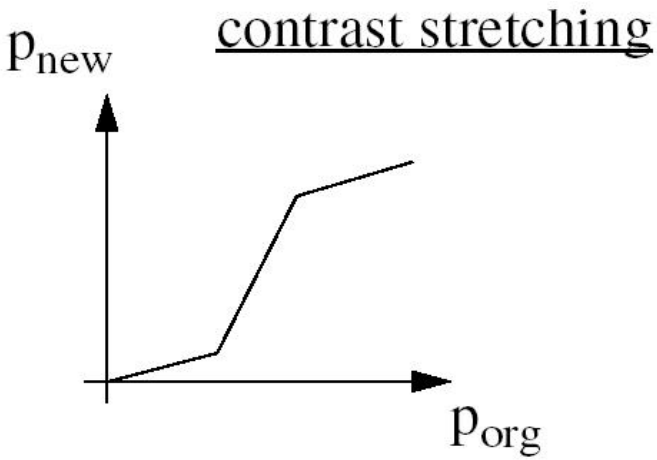
# Contrast Enhancement

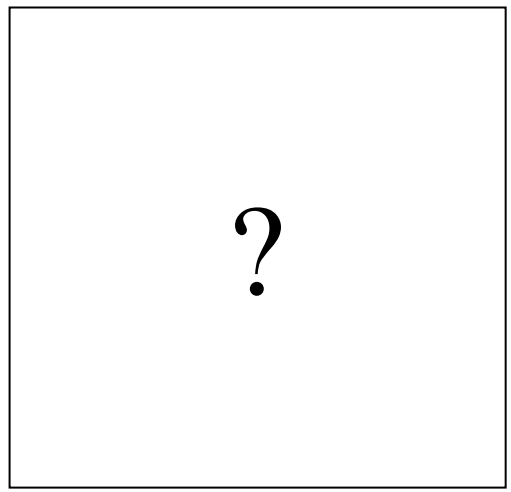
- One technique for fixing such images is process called *contrast stretching*
- Basic idea: perform an intensity transformation to cause darker shades to become darker, and lighter shades to become lighter



# Contrast Enhancement

- Piecewise linear functions are typically used to specify contrast stretching instead of continuous ones
- Give greater user control

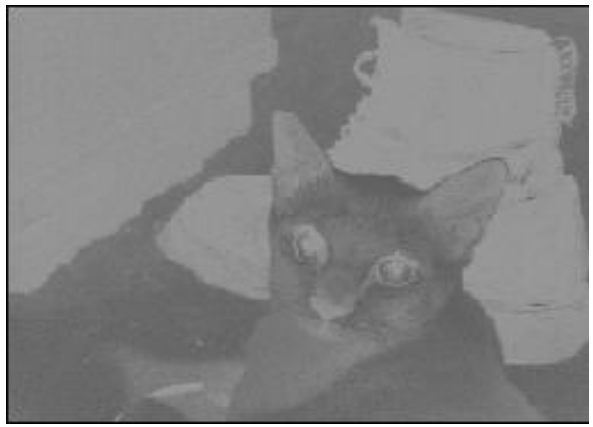


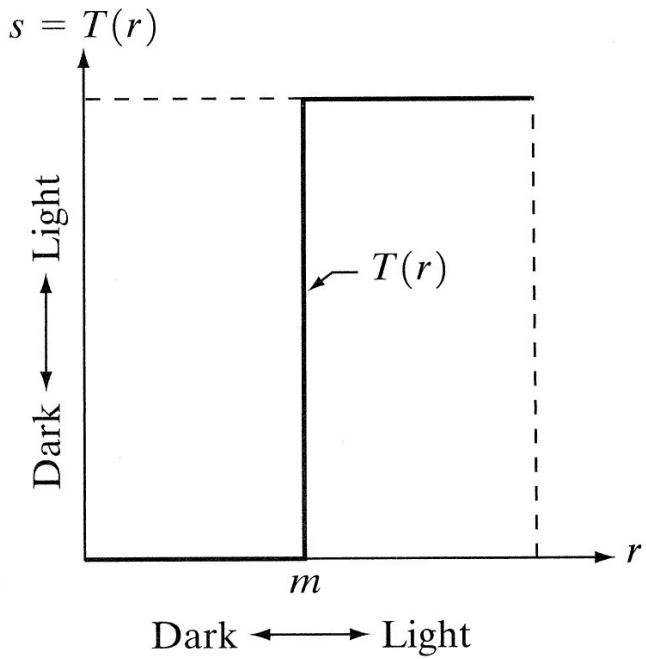



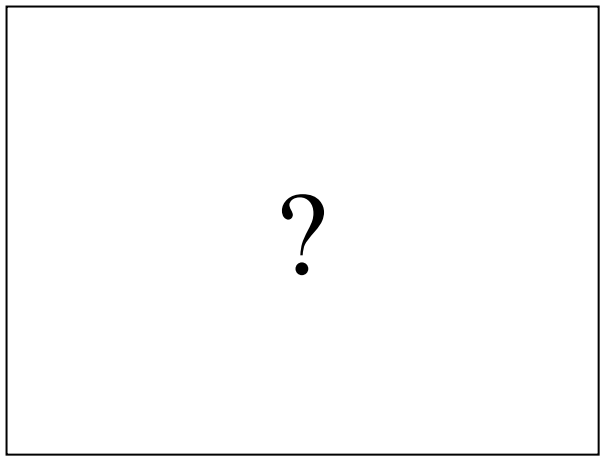

T(p)

# Thresholding

- Another way of manipulating contrast is called thresholding
- What's going to happen?

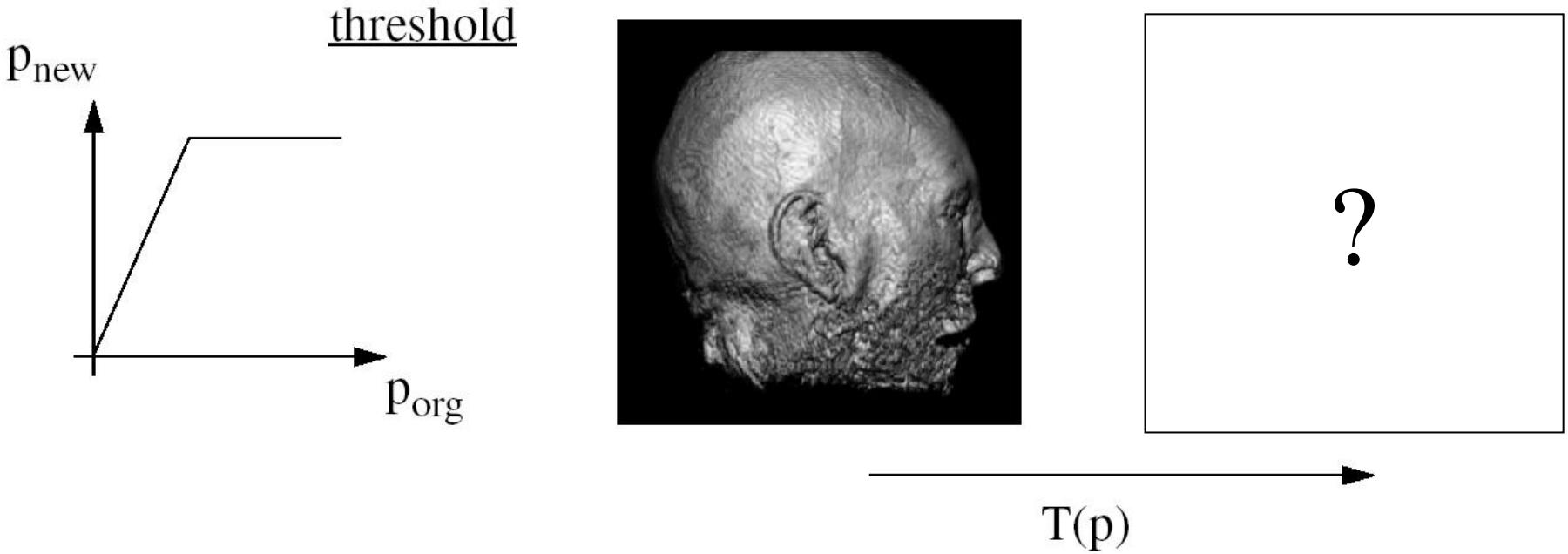




# Image Transformations

- Another example of thresholding using a linear ramp



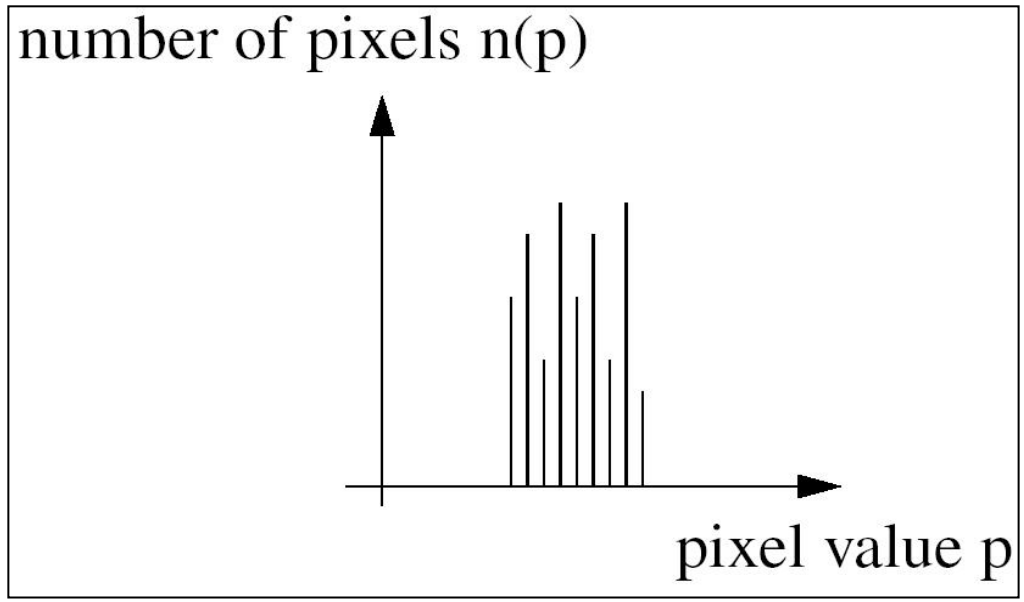

- Why were some of the graylevels preserved?
- Compare with cat image

# Histograms

- An important concept in image processing (and probability & statistics) is the **histogram**
- Suppose we can display 256 discrete gray level intensities, ranging from 0 to 255 (8-bit image)
- To generate a histogram of the image, we would first count the number of pixels having each intensity:
  - $p_0$: $n_0 = n(p_0)$
  - $p_i$: $n_i = n(p_i)$
  - etc.

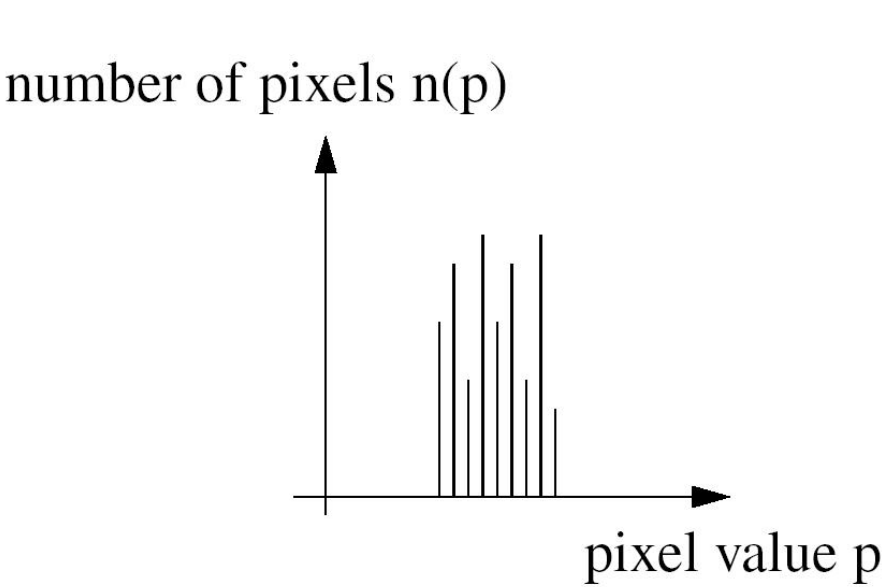
# Histograms

- Then we can plot the counts in a graph to view distribution of intensities across image
- Q: Given an array histogram[], AND array of pixels with associated intensities (pixels[i].intensity), how would you build the histogram?

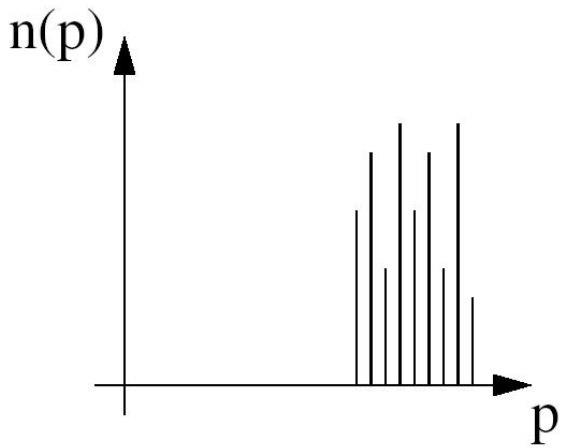



- A: histogram[pixels[i].intensity]++ in a **for** loop over pixels[]

# Example Histograms

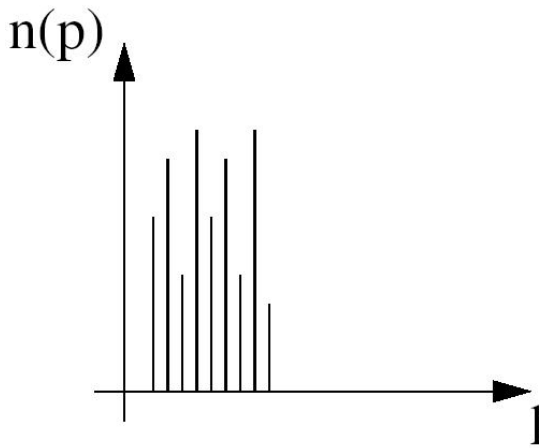


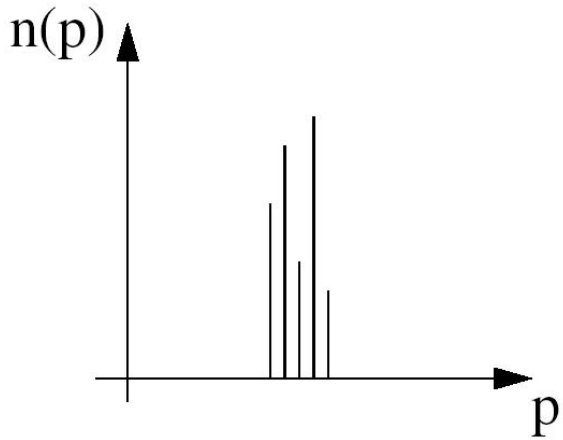

?



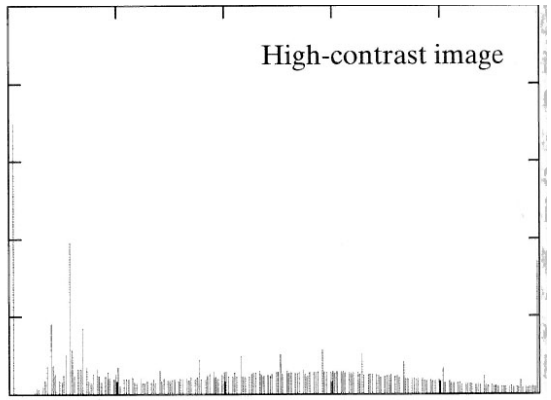

?

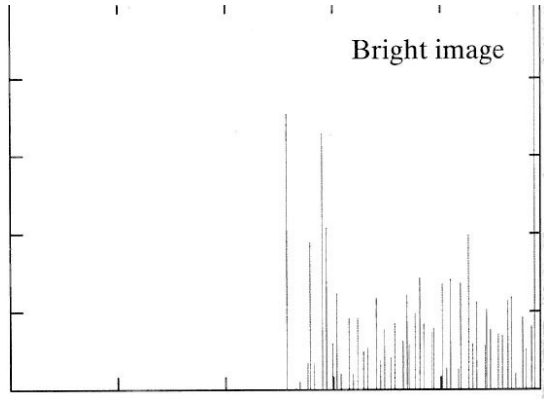
?

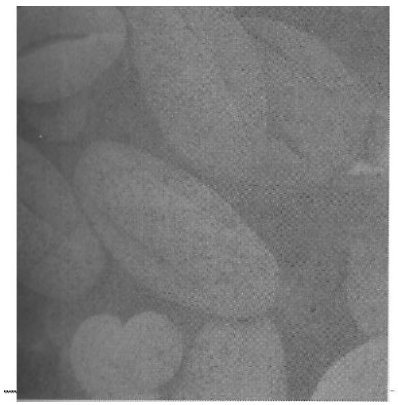


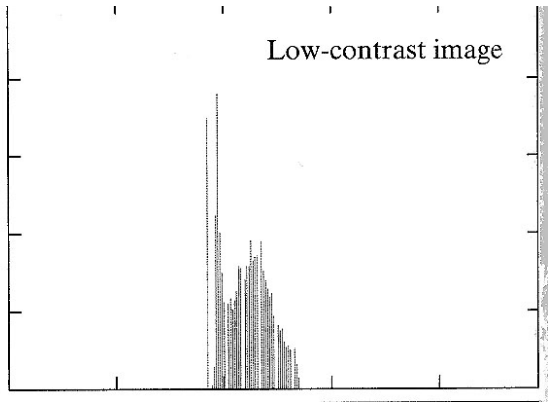



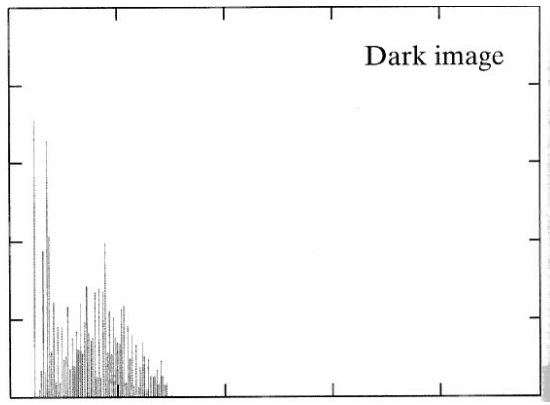
# Example Histograms



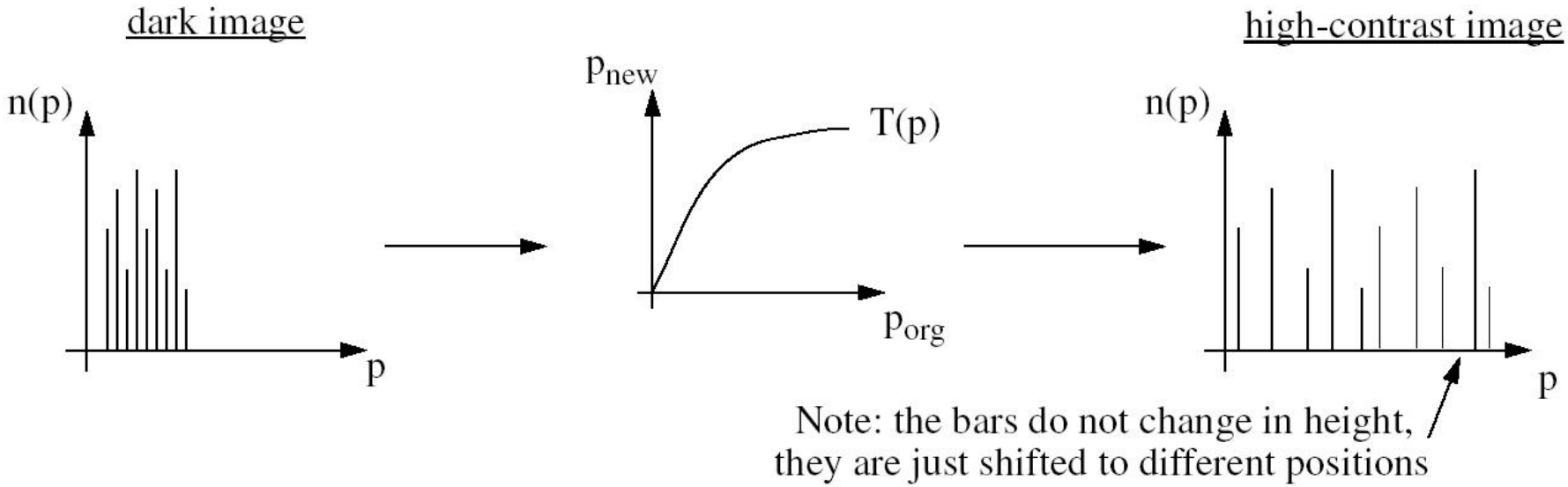





# Histogram Equalization

- One automated (i.e., algorithmic) technique for improving contrast is *histogram equalization*
- Basic idea: increase range of intensities displayed in an image by "stretching" the histogram
- Range of displayed intensities becomes more uniform

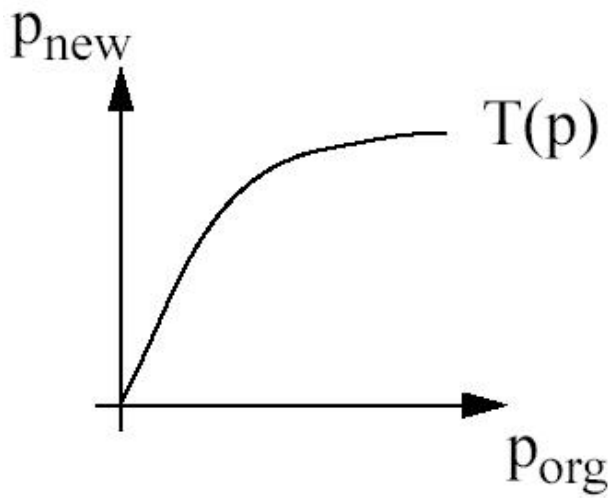


# Histogram Equalization

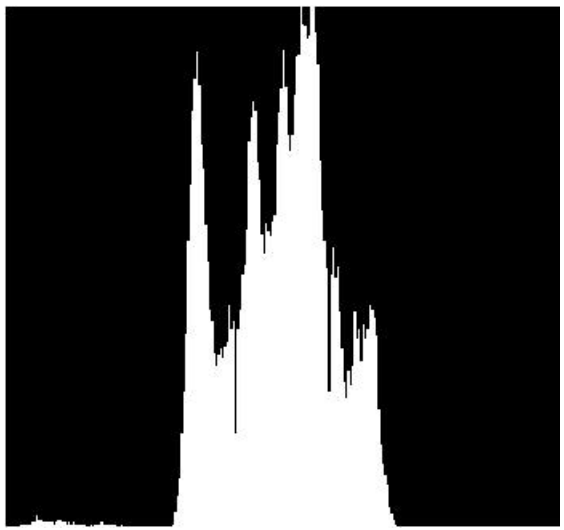
- The discrete histogram equalization equation is

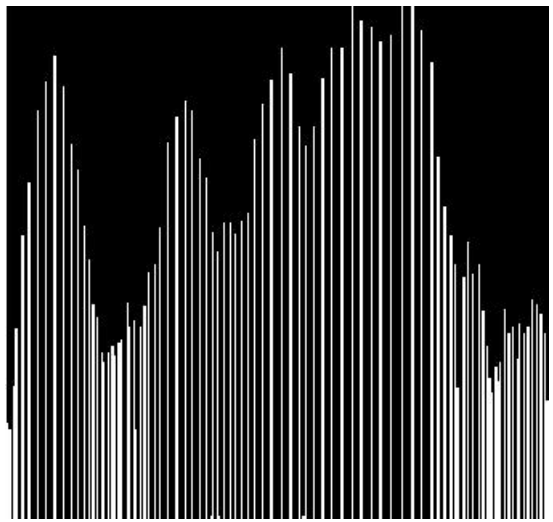
$$p_{new}(k) = \sum_{j=0}^{k} \frac{n(j)}{n_{total}} p_{\max}$$

- $p_{max}$ is maximum possible intensity (not necessarily maximum intensity that happens to appear in the image)
- We accumulate a running total
- This accumulation explains shape of function, which resembles a *cumulative distribution function*



# Histogram Equalization Example



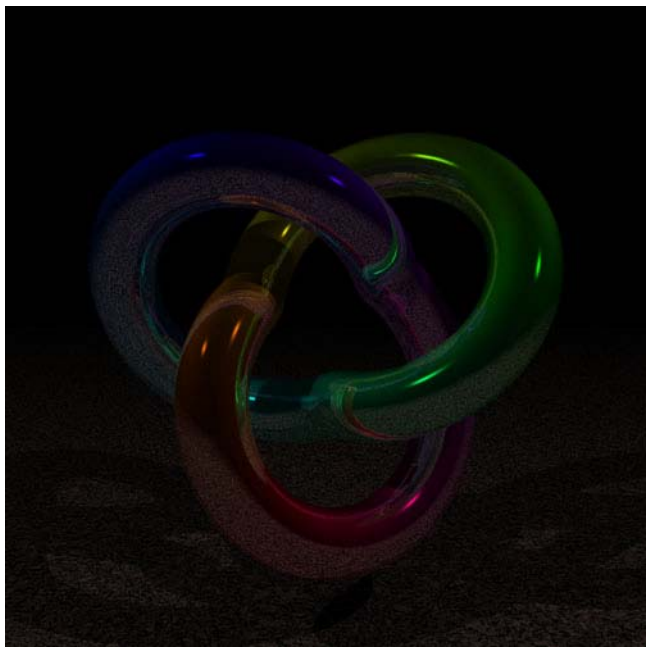


# Histogram Equalization Examples

# Histogram Equalization Example

# Can This Work for Color Images?

- How do we apply histogram equalization to color image?
- Convert RGB → HSV, then equalize histogram of V



- Could we equalize the H or S channels?

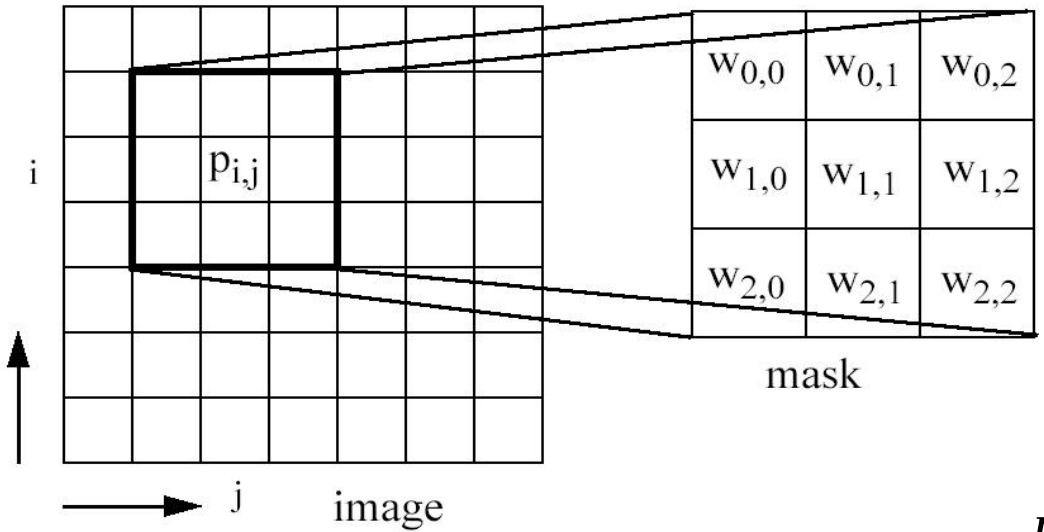
# Histograms Summary

- Histograms are a useful tool for studying images
- We can manipulate images to improve contrast
  - contrast stretching and thresholding
  - histogram equalization
- These are all *global* processes
- Suppose we localize computations and use only local information when processing an image?
- This brings us *discrete convolution* or *filtering*

# Discrete Convolution (Filtering)

- Examples of image processing based on local information include smoothing and edge enhancement
- We use discrete convolution for these operations
  - place a square matrix of weights called a *mask* over each pixel
  - mask takes a weighted sum of neighboring pixels according to weights in mask
  - the resulting intensity is the new output pixel
  - when done for all pixels, a new image is produced of same resolution as original

# Discrete Convolution (Filtering)




for each $i, j$

$temp = 0$

for each $k, l$

$$temp += p^{org}_{(i-1+k,\, j-1+l)} \cdot w_{(k,l)}$$

$$p^{new}_{i,j} = temp$$

$$p^{new}_{i,j} = \sum_{k=0}^{2} \sum_{l=0}^{2} p^{org}_{(i-1+k,\, j-1+l)} \cdot w_{(k,l)}$$

- **Very important note**: do not replace computed values into the original image, but write to an output image
- You need a second memory buffer (array) for this

# Image Smoothing

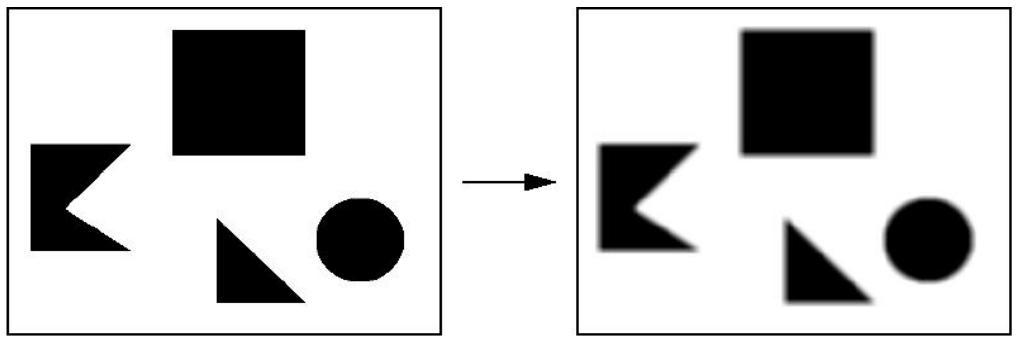
- A smoothing mask averages local pixel neighborhood
- Each pixel's value is replaced by its local average in the output image
- Can be used to remove noise, like speckling

$\frac{1}{9} \times$

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

$\frac{1}{16} \times$

| | | |
|---|---|---|
| 1 | 2 | 1 |
| 2 | 4 | 2 |
| 1 | 2 | 1 |



# Image Smoothing

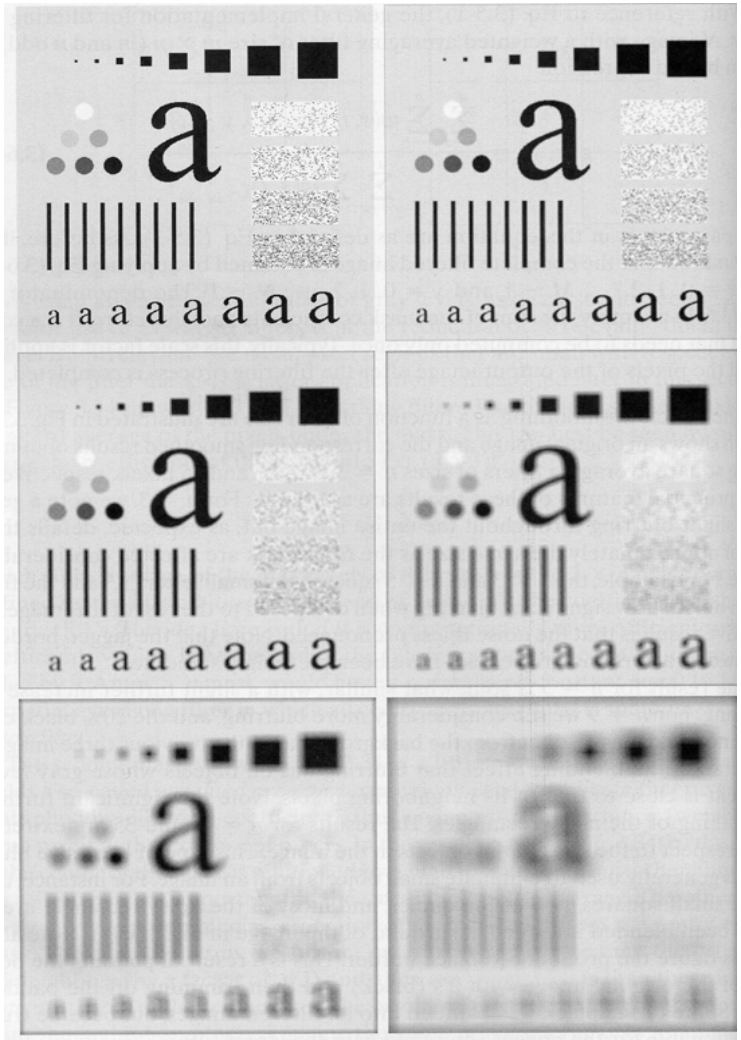
- Larger masks smooth more and cut more noise
- Always make sure that sum of all mask elements equals 1.0
- What would happen if the sum weren't 1.0?
- Image brightness would increase or decrease
- Smoothing the image blurs it – larger masks blur more
- Jagged edges are replaced by blur

# Image Smoothing

- Smoothing is often used in graphical applications
- Why diagonal lines (and fonts) on a screen look smooth, even though they are comprised of a sequence of pixels
- This kind of blurring is a special application of image smoothing known as *anti-aliasing*
- Eye is tricked into seeing a “continuous” line segment

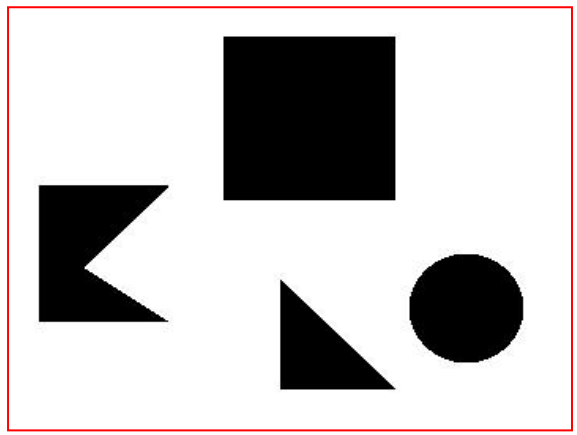
# Image Smoothing Example

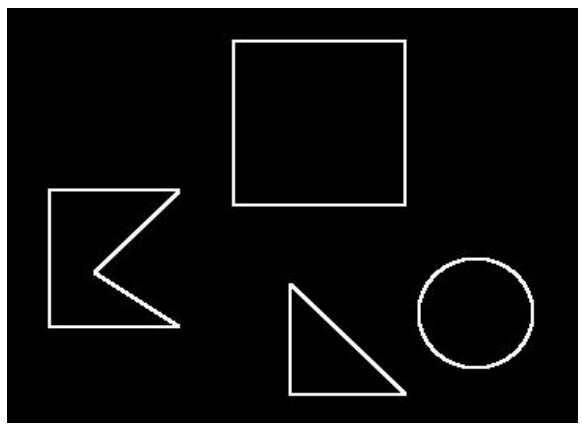
- Results of smoothing top-left image with masks of size 3, 5, 9, 15, and 25
- Notice how some of circles completely disappear
- Also notice how smoothing lessens or even eliminates noise in rectangles



# Image Sharpening

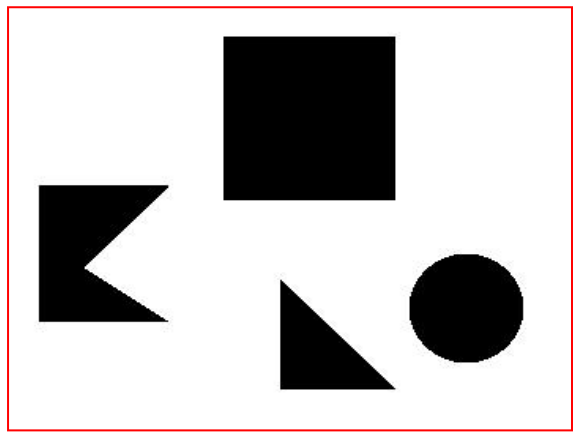
- This operation enhances the edges, rather than blurring the image
- **Edge enhancement**
- It has little effect in smoothly varying areas that have no edges
- Why do this?
- Extract boundaries of regions, perhaps

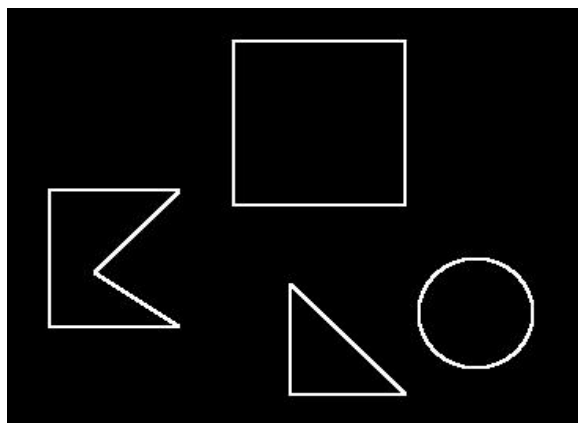




# Image Sharpening

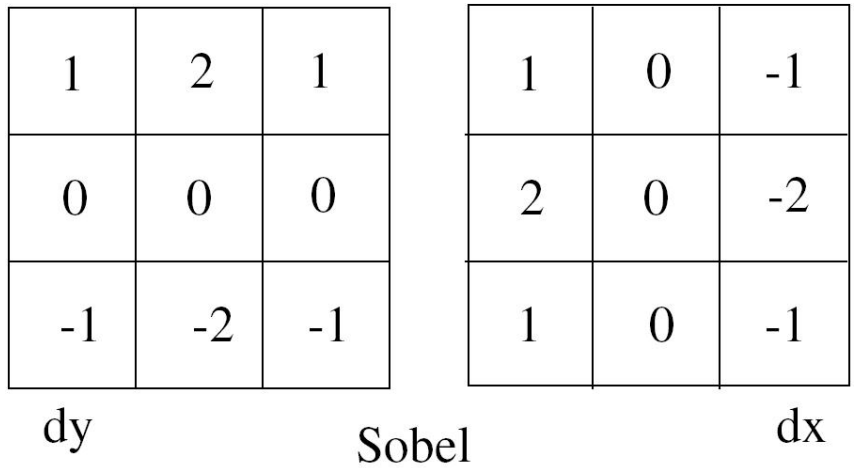
- An edge in image indicates that there is a high local first derivative or *gradient* at the given pixels
- Sharpening masks therefore implement some sort of differentiation
- Usually we are only interested in **gradient magnitude**





# Image Sharpening Mask Example: The Sobel Mask

- The Sobel filter comes in a pair of masks
- Each mask computes an image for x-derivative (dx), other for y-derivative (dy)




- Note that the dy-masks do some smoothing in x-direction (dx-mask smoothes in y)
- This decreases sensitivity to noise in one direction

# Image Sharpening Mask Example: The Sobel Mask

- But increases the sensitivity in the other direction, which is exactly what we want
- Pixel values below zero will occur at edges with negative gradients

| | | |
|---|---|---|
| 1 | 2 | 1 |
| 0 | 0 | 0 |
| -1 | -2 | -1 |

dy

Sobel

| | | |
|---|---|---|
| 1 | 0 | -1 |
| 2 | 0 | -2 |
| 1 | 0 | -1 |

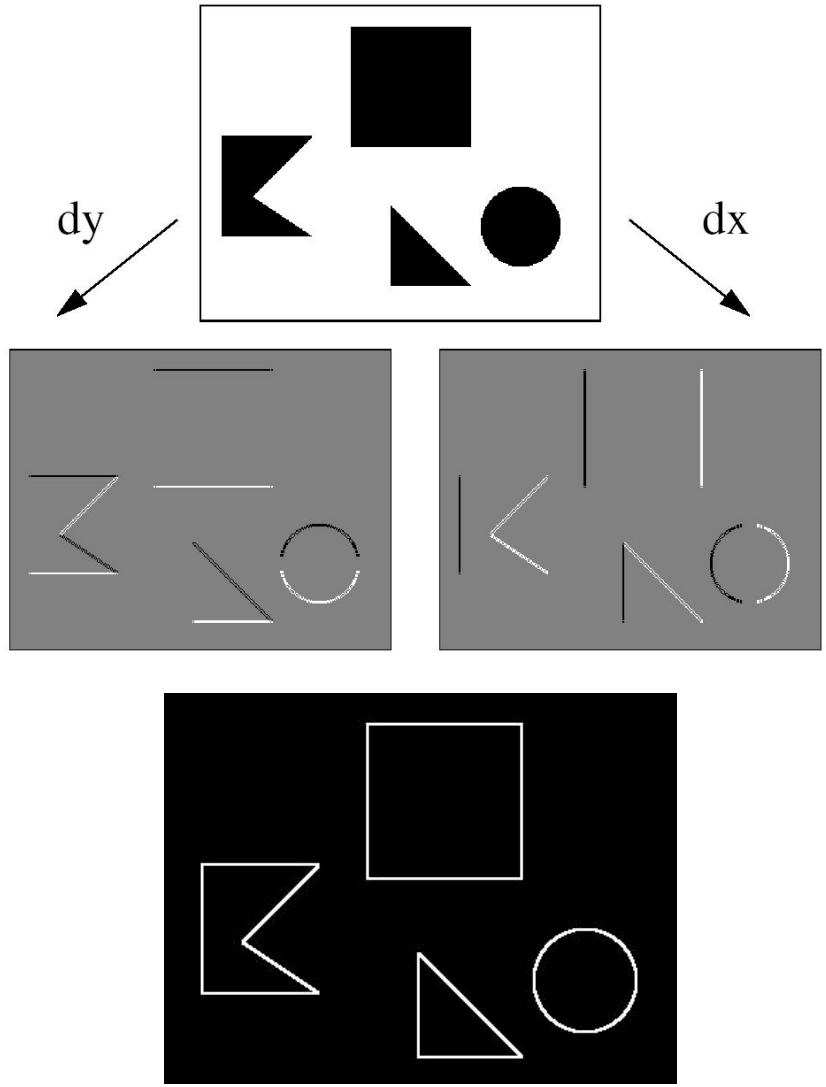
dx

- But this is OK because we are actually only interested in the magnitude, not the sign…why only the magnitude?
- High magnitude (positive or negative) indicates an edge!

# Sobel Mask

- We use the Sobel mask by applying the two masks separately, thereby generating two images, $img_{dx}$ and $img_{dy}$
- Their pixels are combined by

$$img_{new} = \left( img_{dx}^{2} + img_{dx}^{2} \right)^{\frac{1}{2}}$$

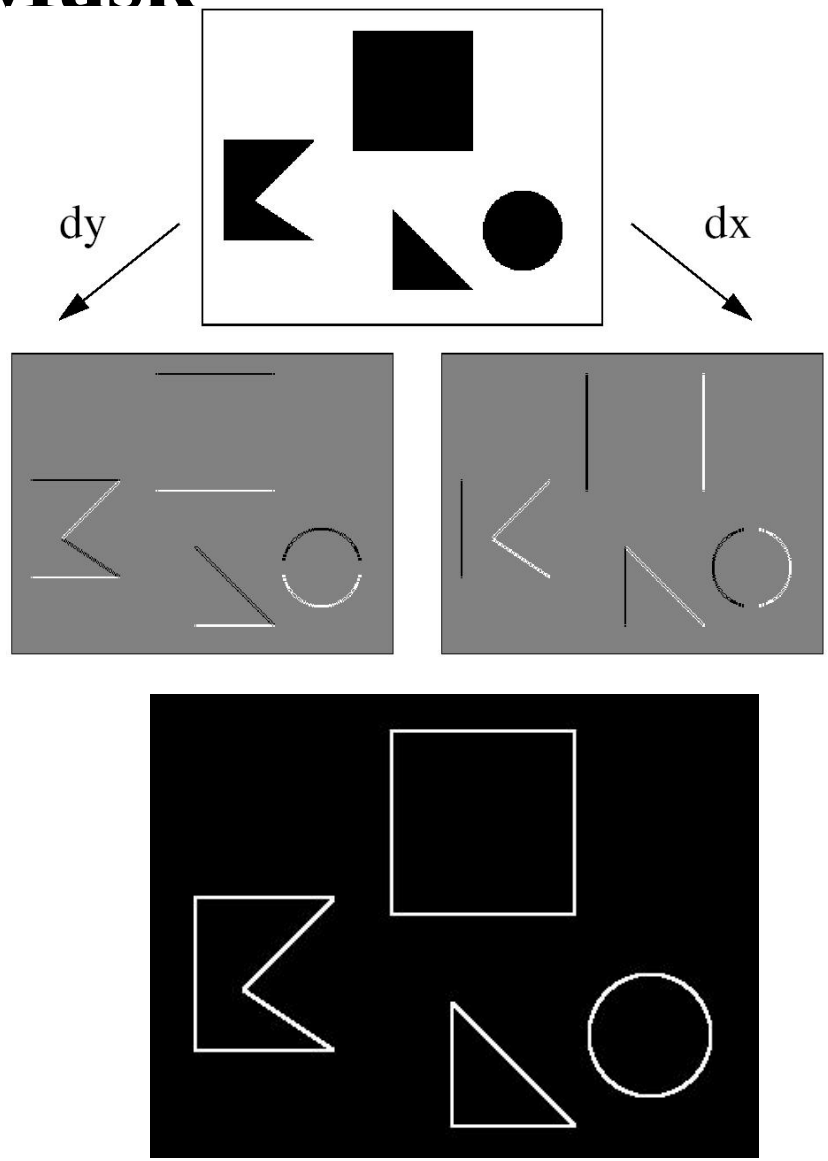


# Sobel Mask

- Since this formula is very computationally expensive, typically the following approximation is used instead:

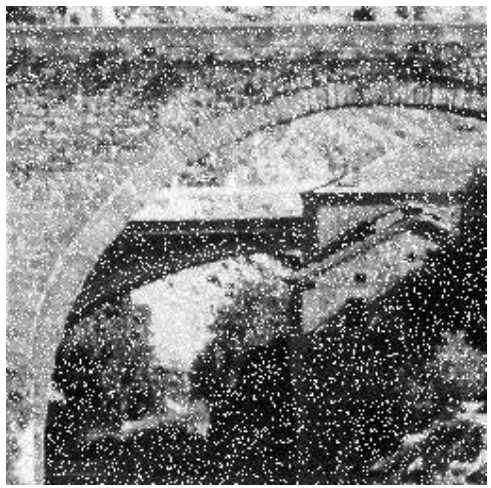
$$img_{new} = |img_{dx}| + |img_{dy}|$$

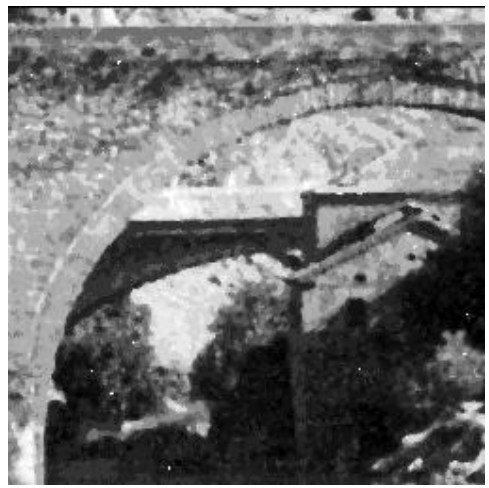
- Again, gradient magnitude is what we want, not direction



# Median Filter

- The median filter is example of an *order-statistics filter*
- Employs local statistical information about pixels to produce output pixel
- Note that we don't use a fixed mask for all pixels
- With a median filter, look at local neighbor-hood and take median value

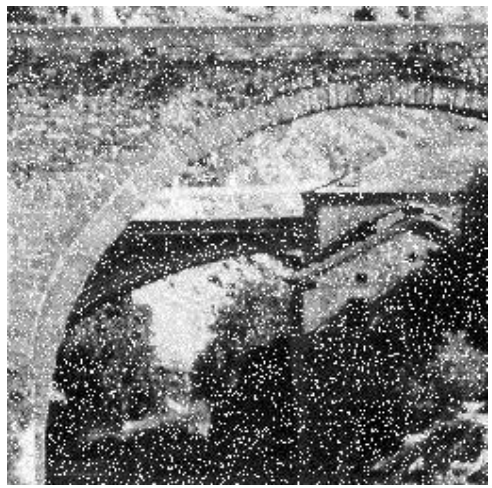


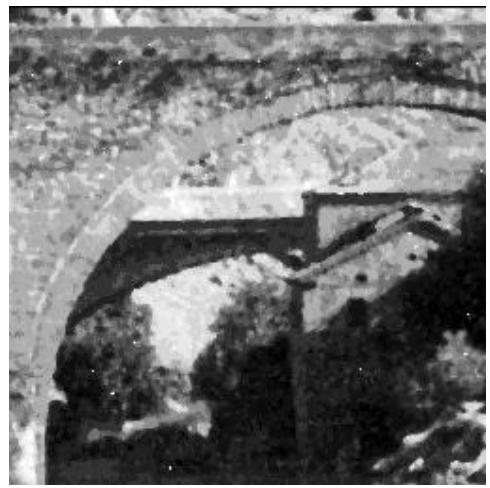


- Naturally, this requires some kind of sorting algorithm

# Median Filter

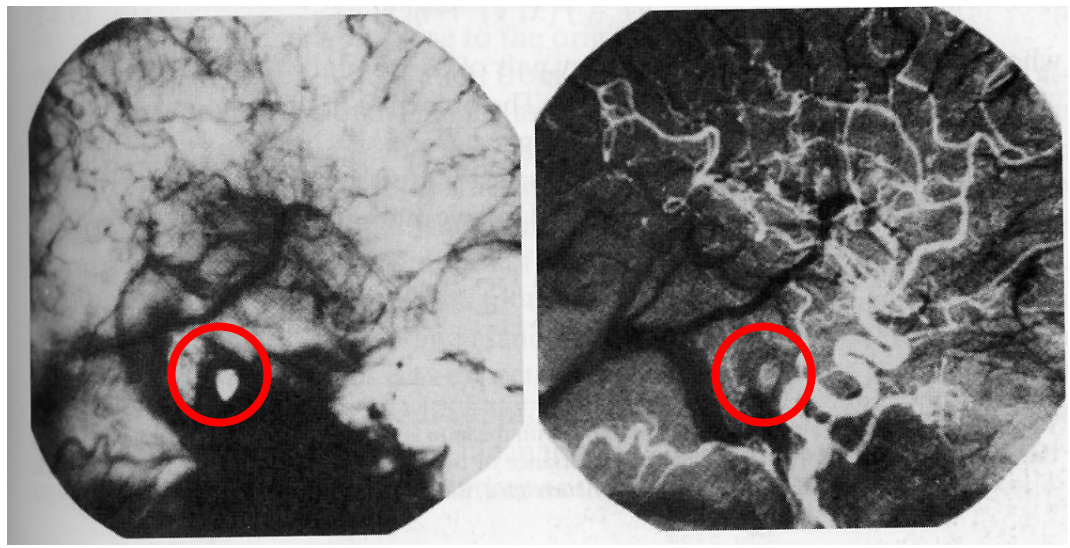
- Median filters are effective for removing **impulse noise**, also called **salt-and-pepper noise**
- Suppose we took the mean instead of the median?
- That's just image smoothing!
- Since median filters perform less blurring than smoothing masks, they end to preserve features like lines and edges





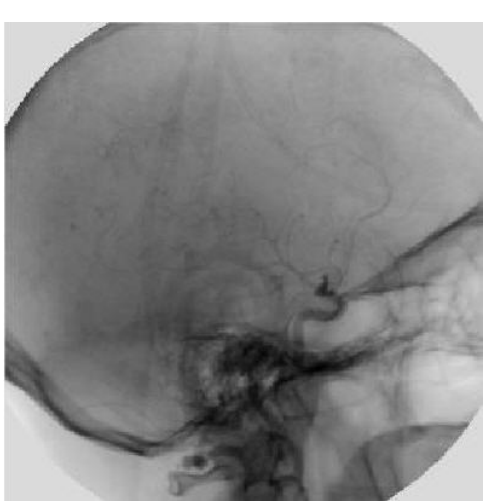
# Image Enhancement via Image Masking/Subtraction

- Say we want to visualize blood vessels in brain
- First, we take an image of brain (e.g., MRI)
  - this will be called the *mask*
- Then we inject a contrast agent and take another image
- Then we subtract first image from second
- The resulting image shows changes introduced by contrast agent

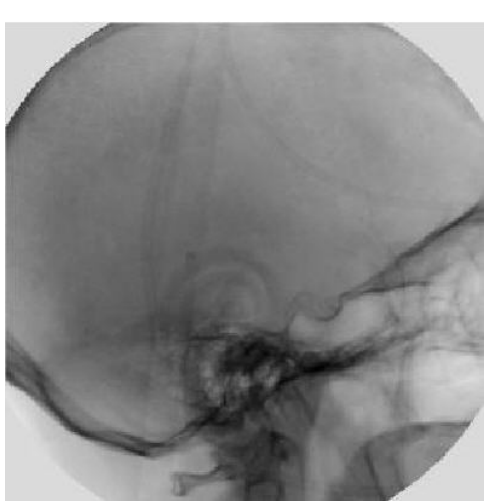


# Image Subtraction Example

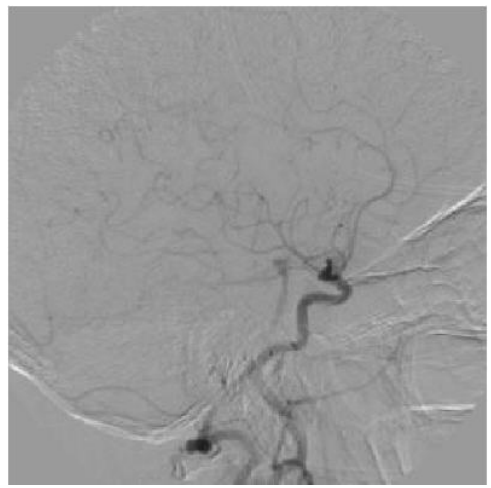
- X-ray angiography to enhance *perfused* vessels



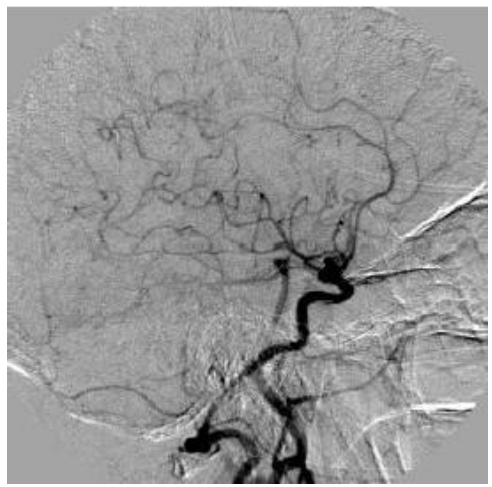
perfused



non-perfused
(mask)



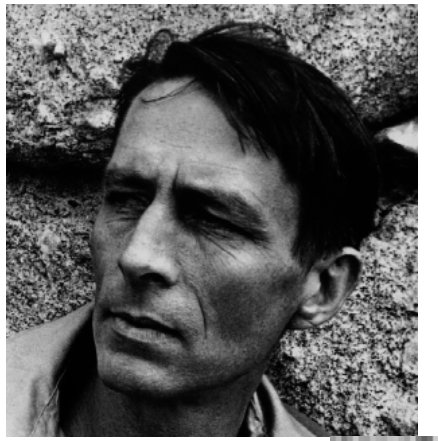
contrast-
enhanced

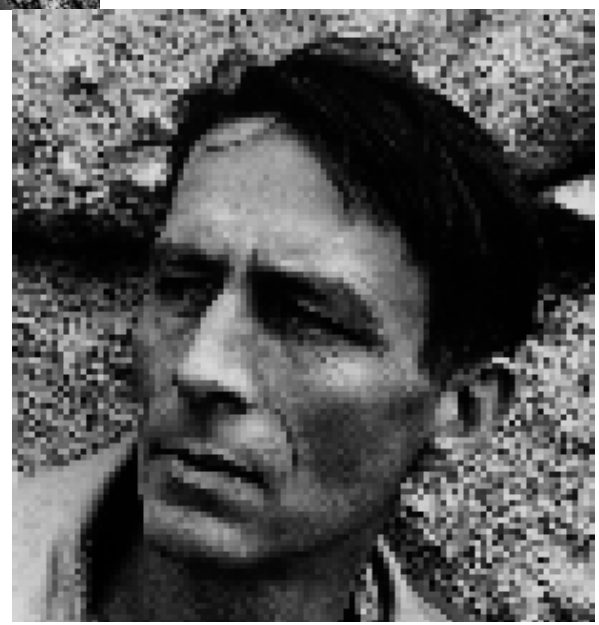


- Perfuse = to force fluid through something

# Subsampling

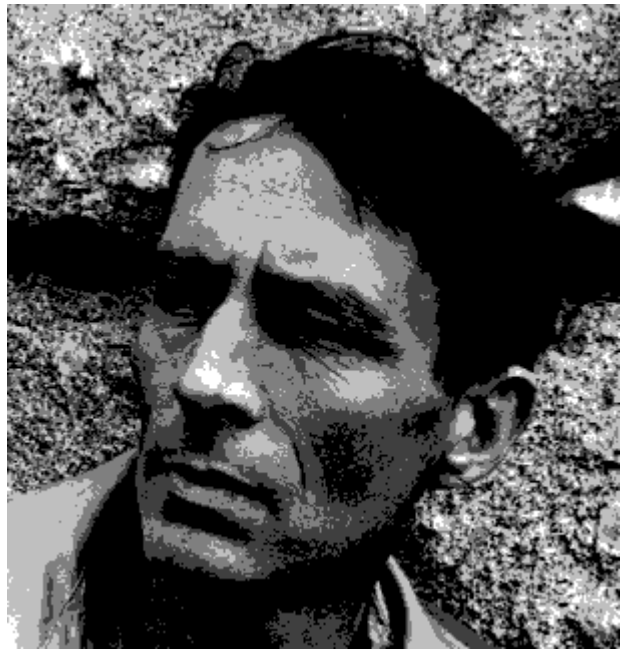
- Sometimes we need to change image resolution
- Subsampling used to decrease resolution
- Supersampling used to increase resolution
- How can we improve image quality in both cases?
- Interpolation





# Quantization

- Very common technique in all of computer science
- Basic idea: represent broad range of values using a much smaller set
- In image processing: reduce number of graylevels (bits) represented
- For normal vectors: store only a subset of the infinite number of possibilities (unit sphere)

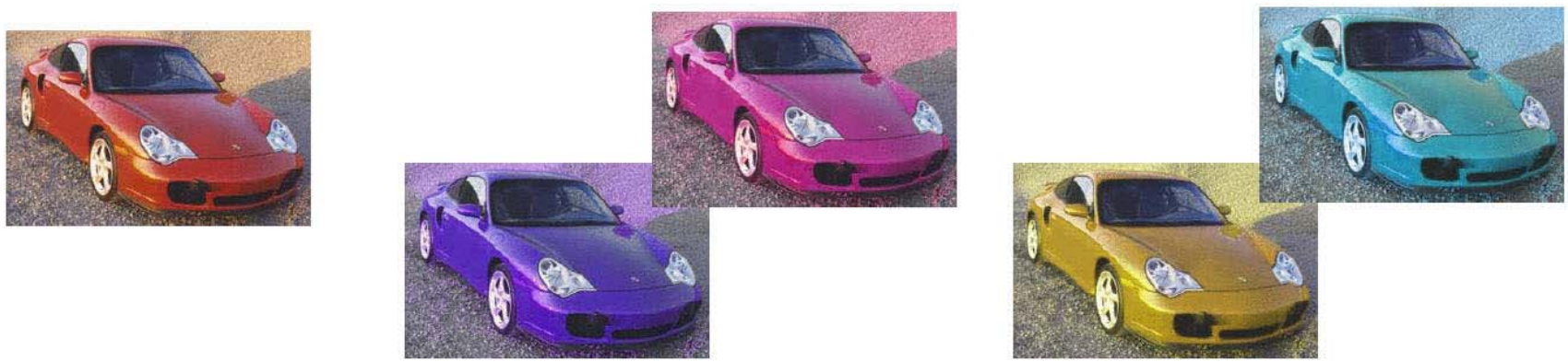


# Color Transformations

- Image transformations not limited to intensity trans.
- We can also transform the H and S channels

Original

Hue transformed

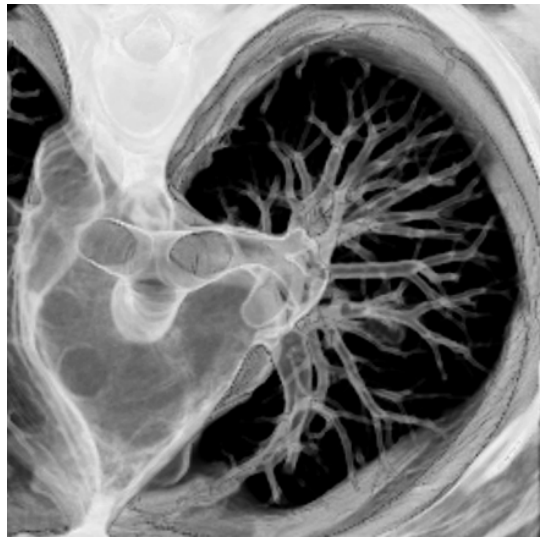


# Volume Rendering Hardware

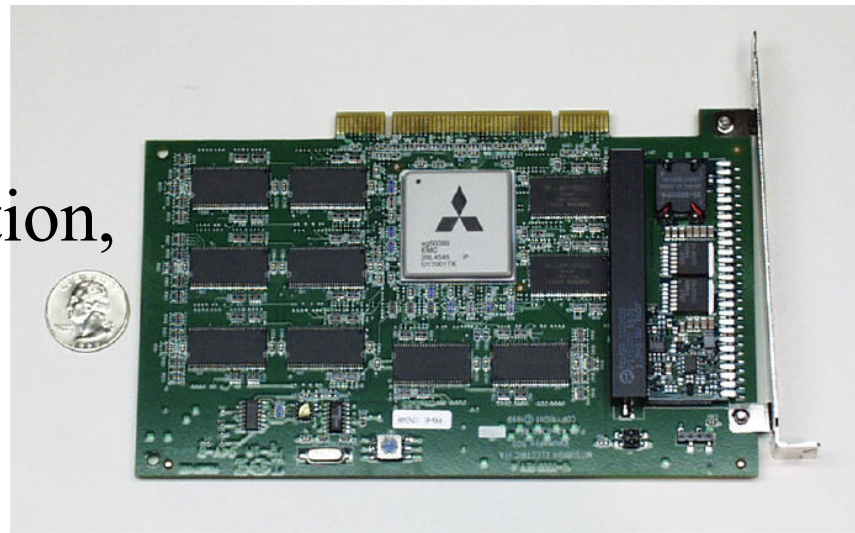
# Volume Rendering Hardware

- We have looked at several volume visualization algorithms
- Today: how to implement ray-casting in hardware
- Transformations, viewing, projection, interpolation, classification, shading, compositing
- Desirable: rendering large-ish volumes (~$256^3$) in real-time
- Limit pre-processing time and permit interactive changes to classification and shading parameters

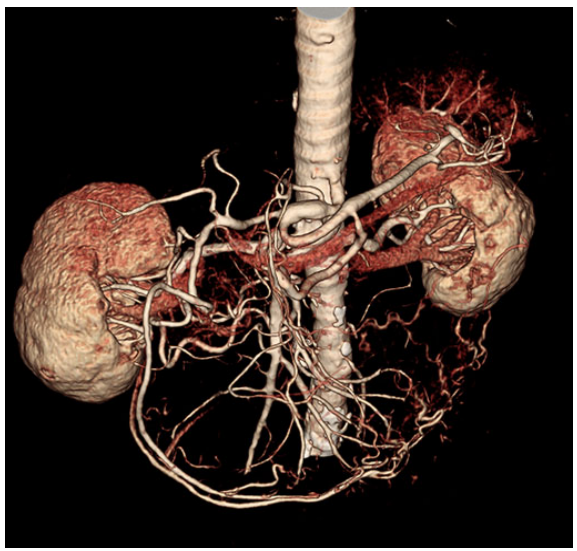
# VolumePro Rendering Hardware

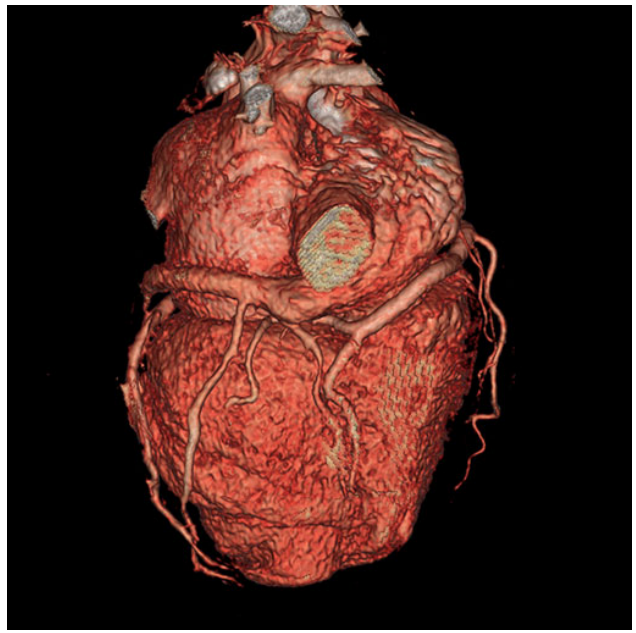


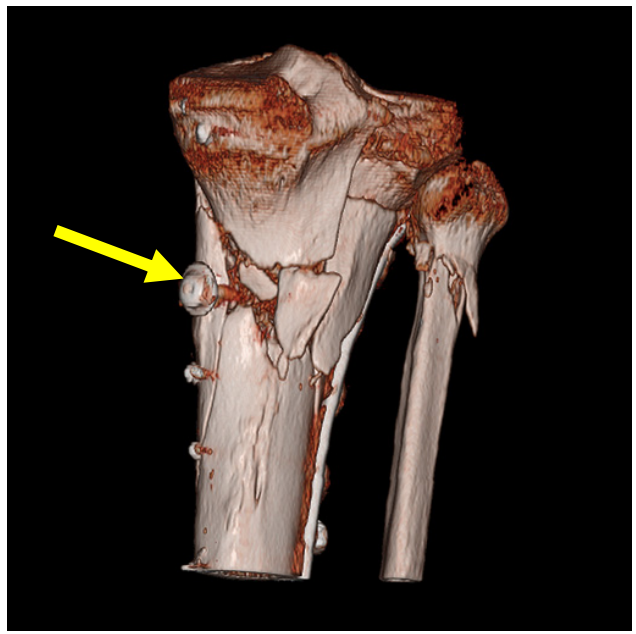
- VolumePro 500 – MERL/USB
- PCI hardware extension card
- Hybrid of shear-warp and ray-casting
- $256^3$ volumes at 30 fps
- Parallel projection
- Projection via ray-casting
- Gradient estimation, classification, Phong illumination
- Cropping and cutting planes to visualize portions of volumes



# VolumePro 1000

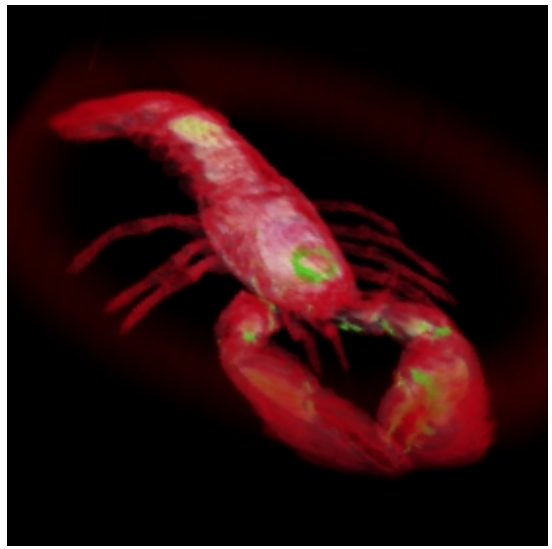


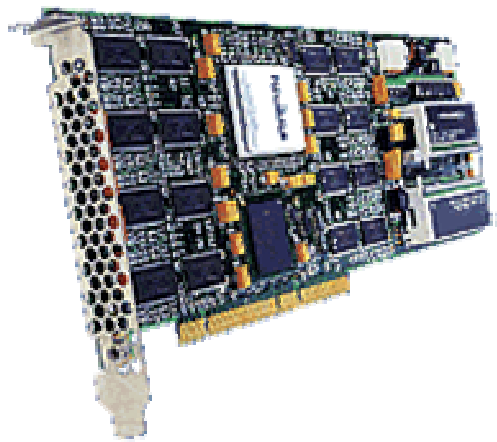




# VolumePro Rendering Hardware

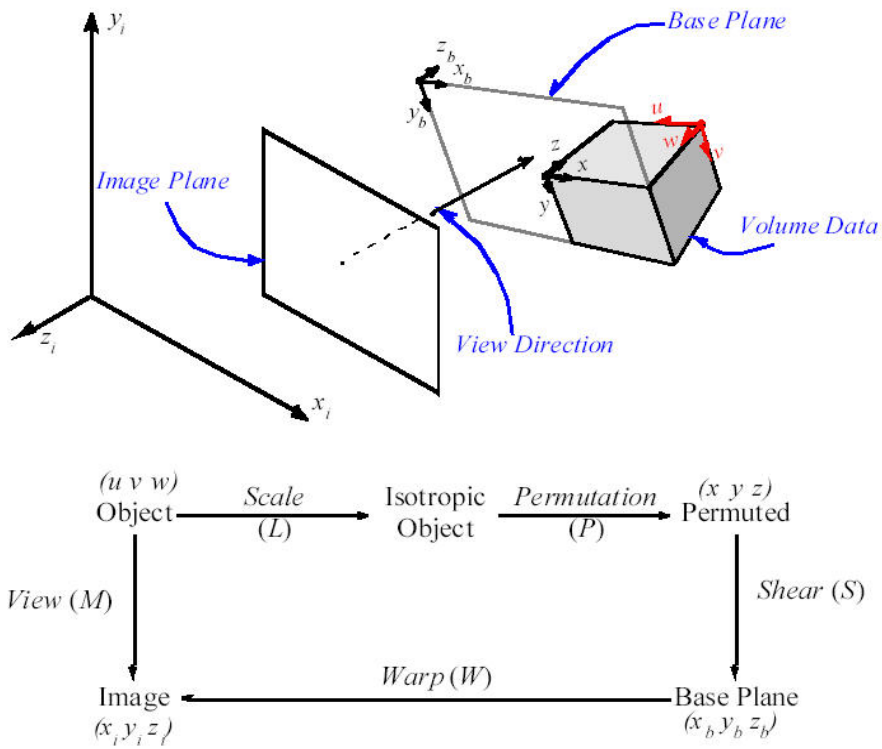
- Users can change classification and shading parameters at run-time
- Not scalable, fits entirely on a single chip
- Four parallel rendering pipelines
- Pipelines share information to reduce memory bandwidth requirements
- Supports only 8-bit and 12-bit volumes (CT & MRI often use 12)
- MERL → TeraRecon Corp





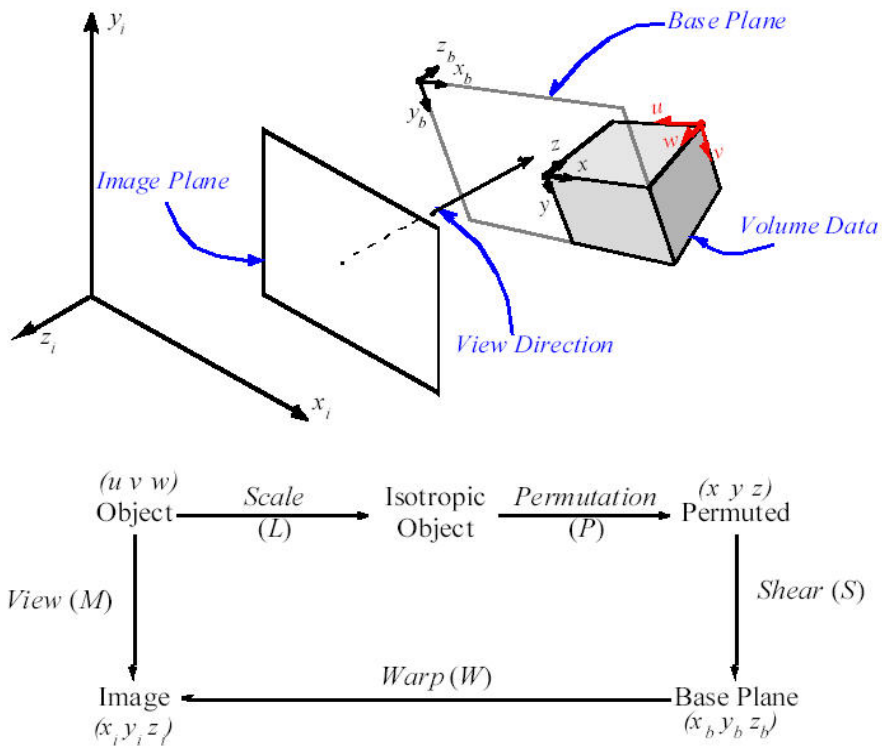
# VolumePro Rendering Algorithm

- Ray-casting of isotropic and anisotropic rectilinear volumes
- Shear-warp factorization of viewing matrix = step through volume along a major axis
- Scaling and shearing ($L$) transform anisotropic volume into isotropic



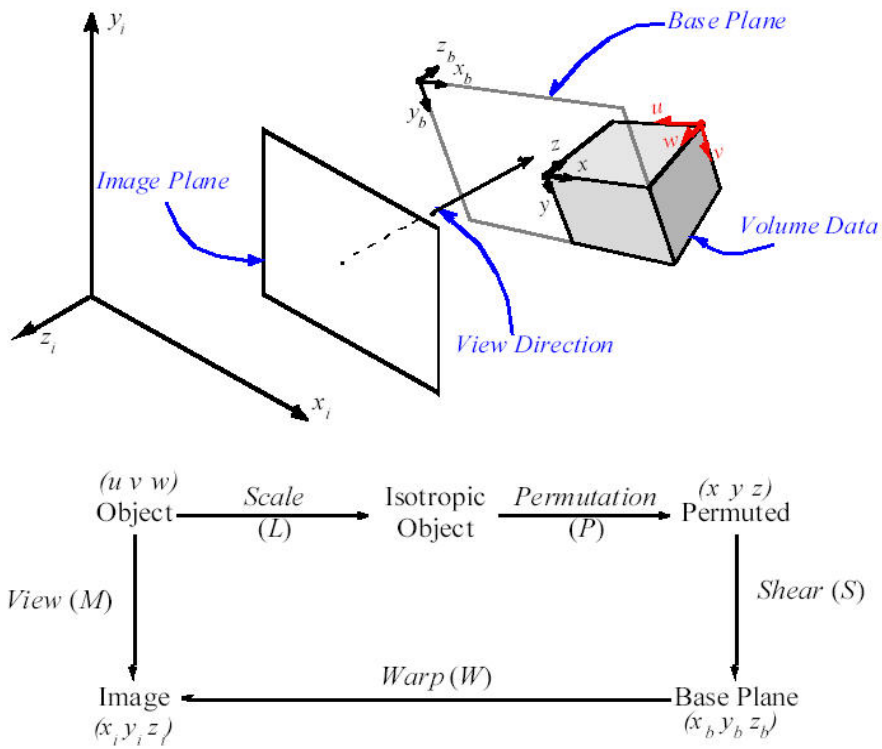
# VolumePro Rendering Algorithm

- Permutation matrix *P* makes axis most parallel to viewing direction the *z* axis
- Shear matrix *S* projects volume onto base plane via ray-casting
- Rays cast from base plane into the volume, rather than from image plane



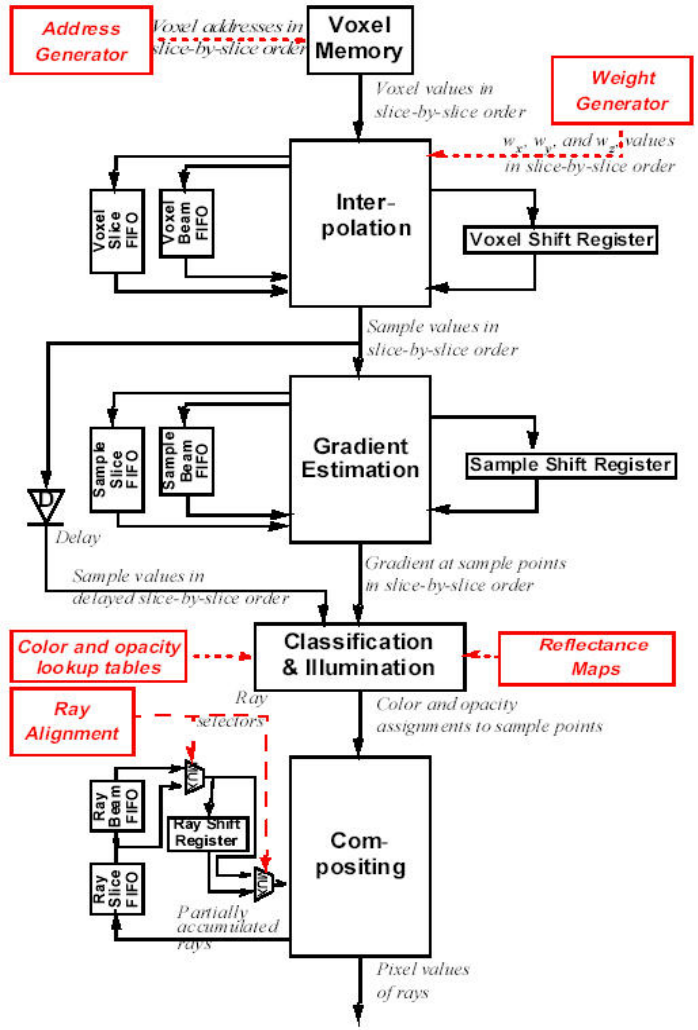
# VolumePro Rendering Algorithm

- Guarantees one-to-one mapping from sample points to voxels
- Base plane transformed to image plane by warp matrix
  $W = M \bullet L^{-1} \bullet P^{-1} \bullet S^{-1}$
- Bilinear interpolation done in external 3D graphics hardware



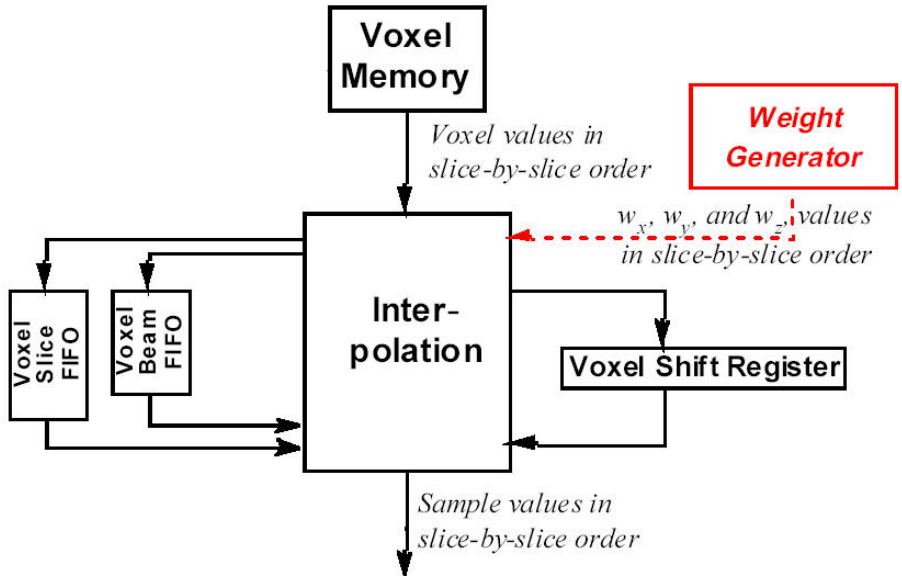
# VolumePro Ray-Casting Pipeline

- By design, each voxel read only once from memory
- Pipelined architecture means voxels processed as quickly as they can be read from memory
- Major phases: interpolation, gradient estimation, classification and shading, compositing



# Interpolating Voxel Values

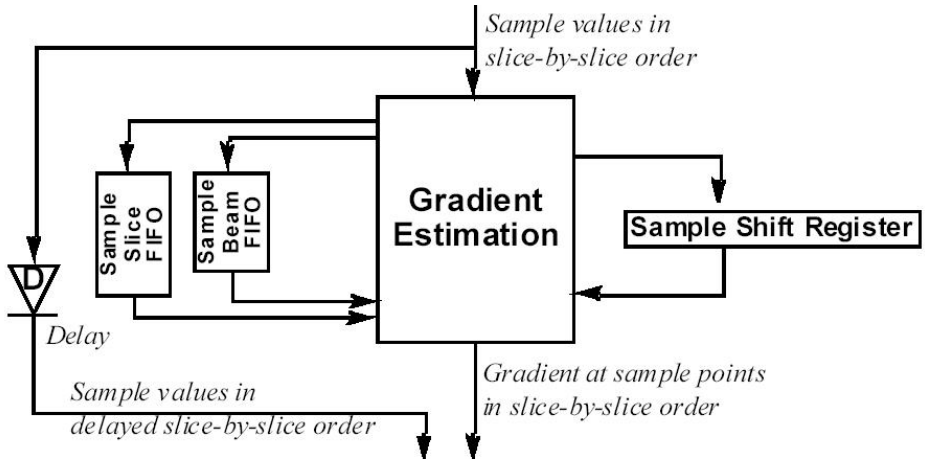
- Interpolation unit converts each *beam* of voxels in a scanline into a stream of samples
- Voxel Slice FIFO and Voxel Beam FIFO locally store voxels to maintain neighborhood of "current" voxel
- Trilinear interpolation requires weights




- Weight Generator calculates weights
- All rays are parallel → weights identical for all samples in a single slice

# Gradient Estimation

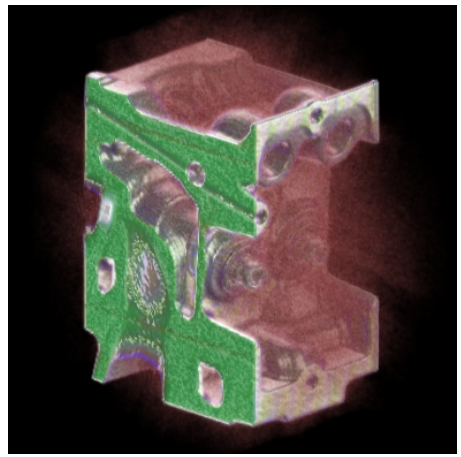
- Gradient Estimation unit uses central differences to estimate gradients
- We have *x* and *y* neighbors, but what about *z*?
- We maintain *previous* and *next* slices into two FIFO buffers



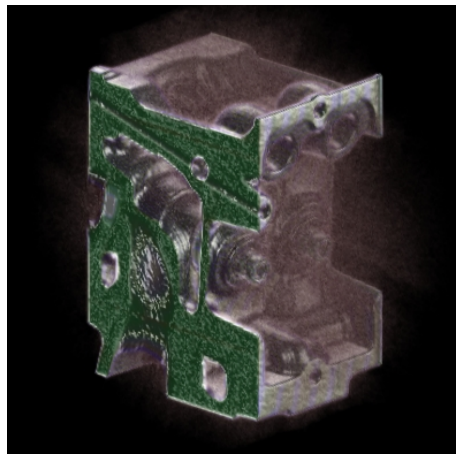

- Gradient estimation unit lags a little behind the interpolation unit

# Gradient Magnitude

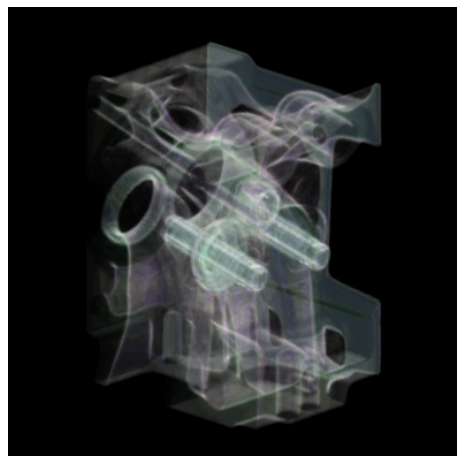
- Can multiply opacity or specular illumination (!) by gradient magnitude to create certain effects
- Gradient magnitude modulation of the specular illumination will highlight curved regions and attenuate flat regions



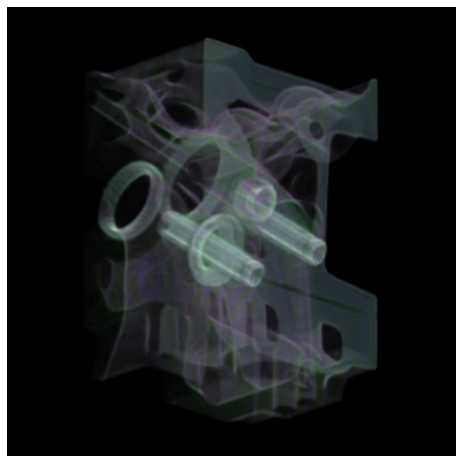
None



Illumination



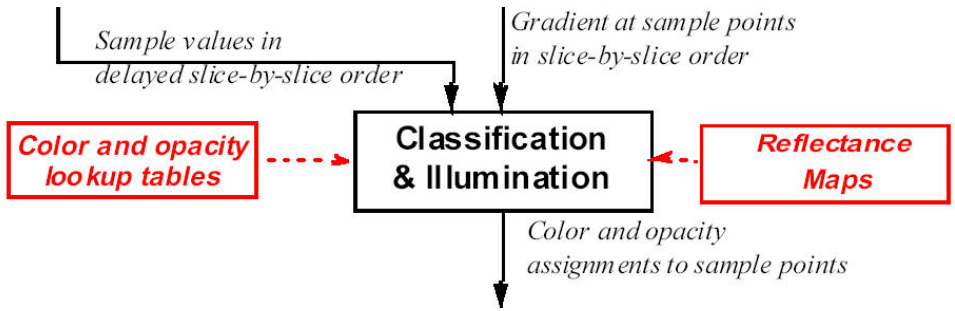
Opacity



Opacity & Illumination

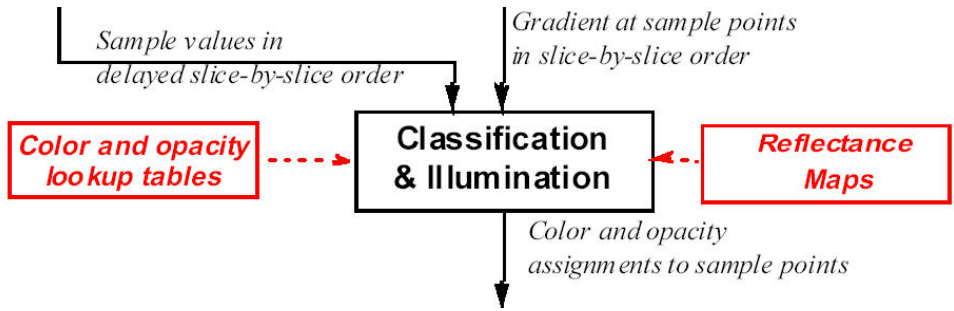
# Assigning Color and Opacity

- Post-shaded pipeline (we already did interpolation)
- Look-up tables can be changed at run-time easily
- LUTs make classification very rapid
- Transfer functions given in are converted by hardware driver in OS to LUT format



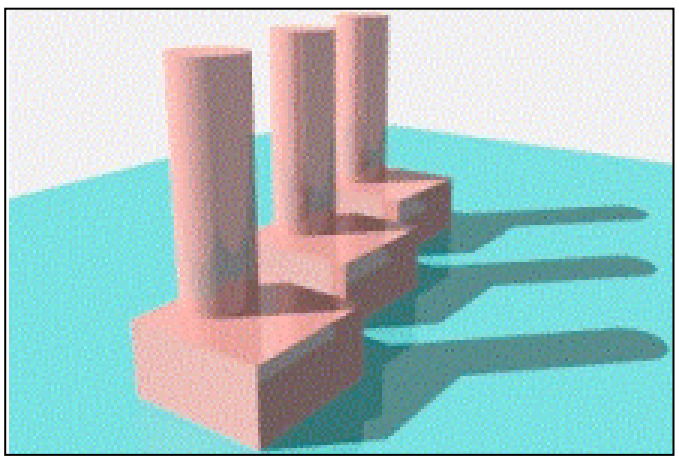
# Sample Illumination

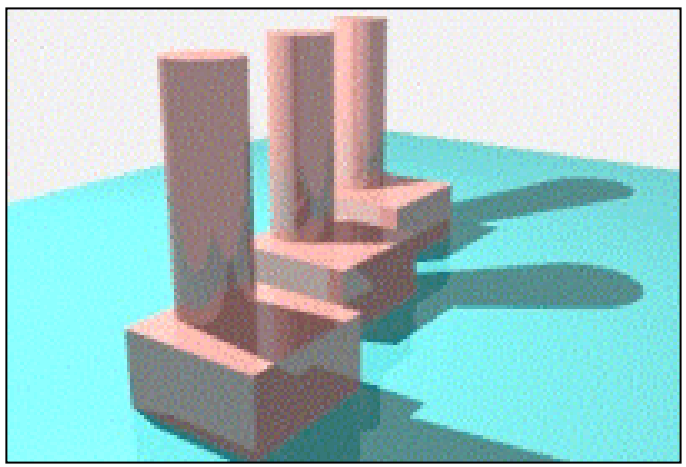
- Phong illumination
- User provides $k_d$ and $k_s$
- Diffuse and specular coefficients
- Reflectance maps convert gradient vector to illumination value



# Reflectance Map

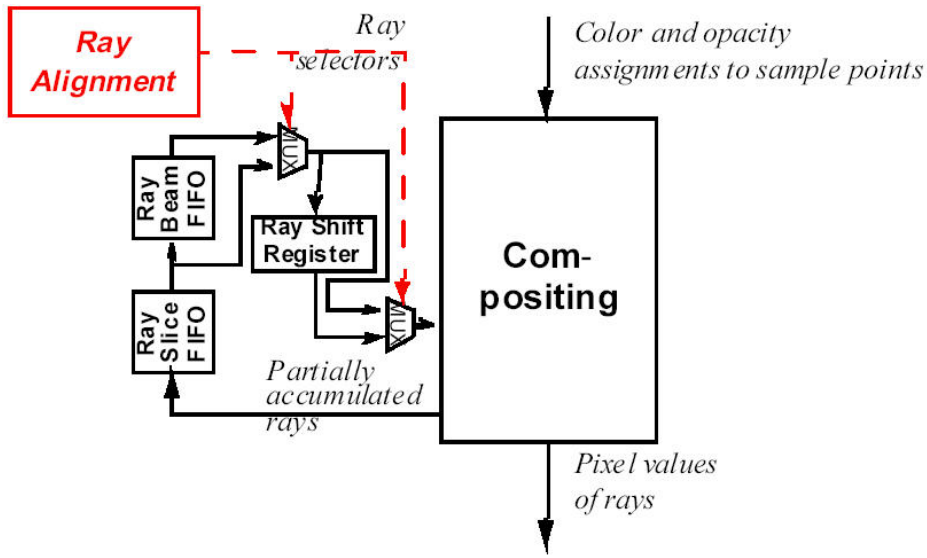
- Reflectance maps constructed by taking a subset of all possible gradient directions, and for each direction and given viewpoint, compute the illumination
- Changed when light source(s) changed
- Directional lights but no positional lights
- How does the type of light affect shading?





# Accumulating Color Values Along Rays

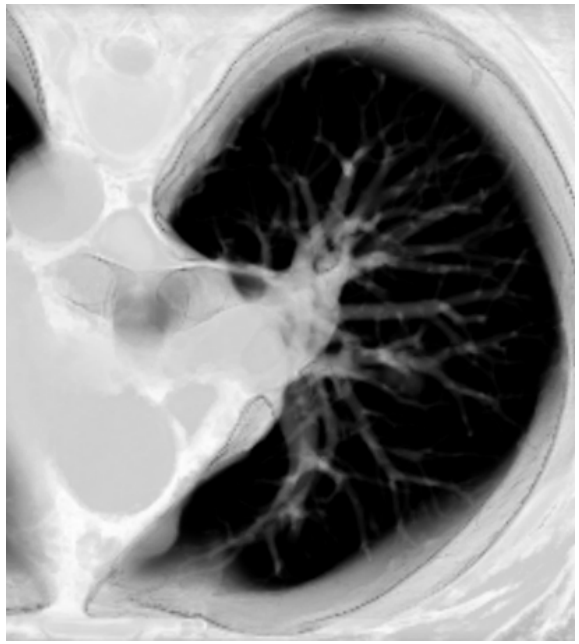
- Compositing happens in front-to-back order
- FIFOs store intermediate results and wait for the next sample to arrive in the stream
- When ray passes out of volume, pixel finished
- When all rays are finished, warp base plane to image plane




- Warping done by texture-mapping base plane onto a quadrilateral
- 3D graphics hardware rasterizes quadrilateral

# Example Visualizations

MIP

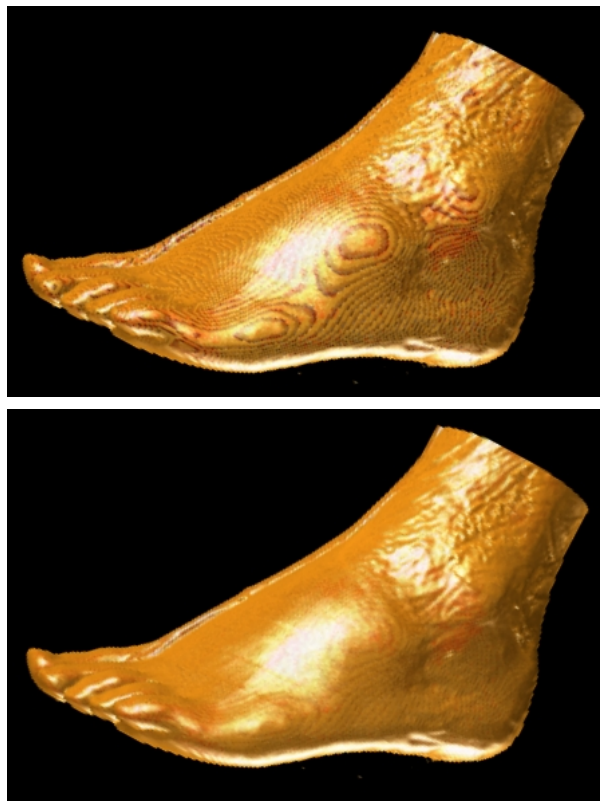


No illumination

Illumination, gradient magnitude modulation of opacity

# Advanced Features of VolumePro: Supersampling

- Sample data at a higher frequency than voxel spacing
- VolumePro does it only along *z* direction, i.e., along rays cast into the volume
- Minor changes to architecture
- A supersampling factor of *k* reduces frame rate by a factor of 1/*k*
- Compare top and bottom images
- Top = no supersampling
- Bottom = 3x supersampling

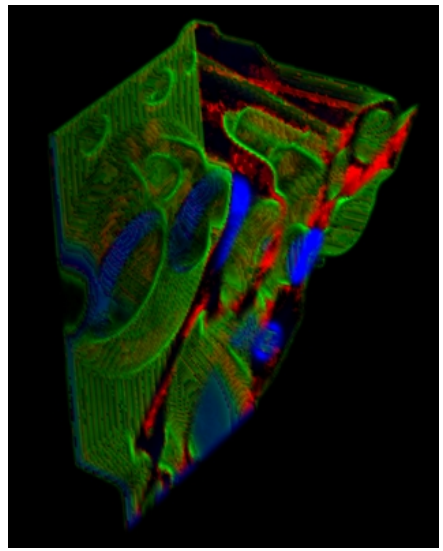


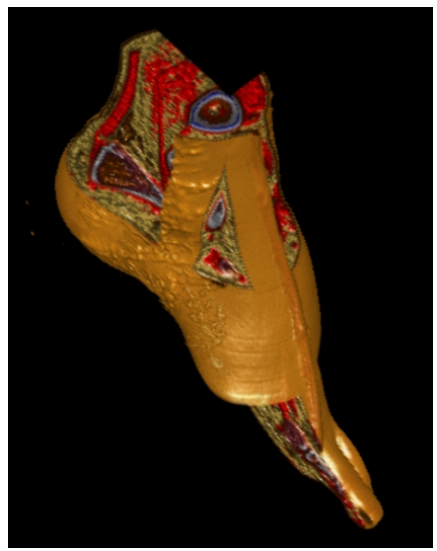
# Advanced Features of VolumePro: Supervolumes and Subvolumes

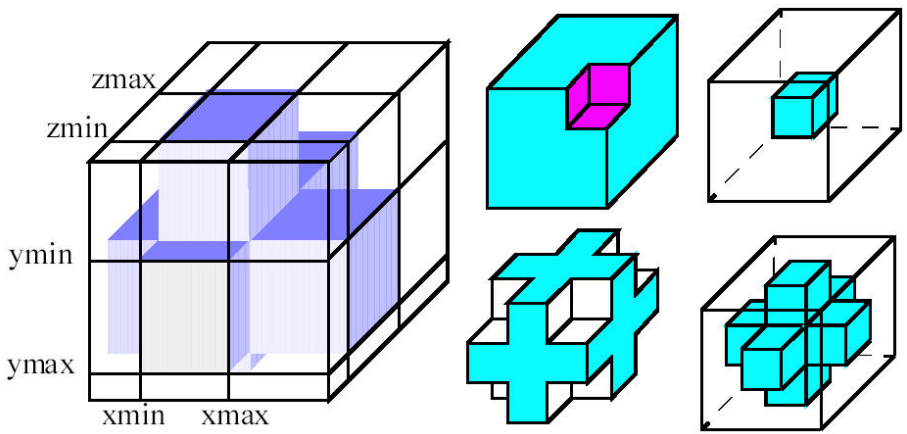
- Hardware PCI card can handle volumes of up to dimension 256 in its 128 MB on-board memory
- To render large volumes, partition volume into smaller blocks, render each, and combine resulting images in *software*
- Software driver partitions volumes
- Gradient estimation, trilinear interpolation
- Memory blocks swapped to and from main memory across PCI bus

# Advanced Features of VolumePro: Cropping and Cut Planes

- Hide portions of data using parallel planes
- Why do this?
- See cross-sections, interior of volume
- Cropping along volume planes only
- Why restrict it to only parallel planes?
- Union, intersection, difference

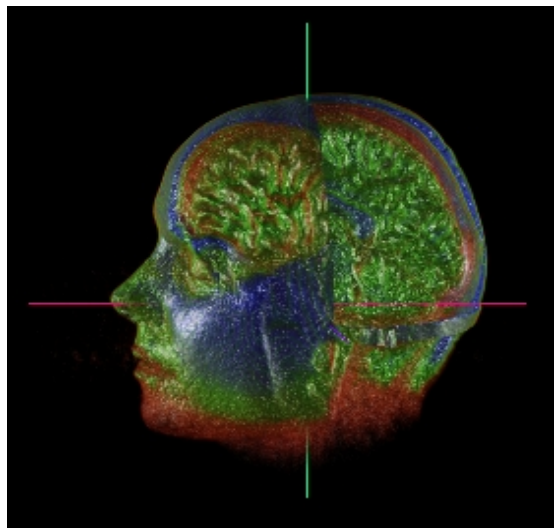






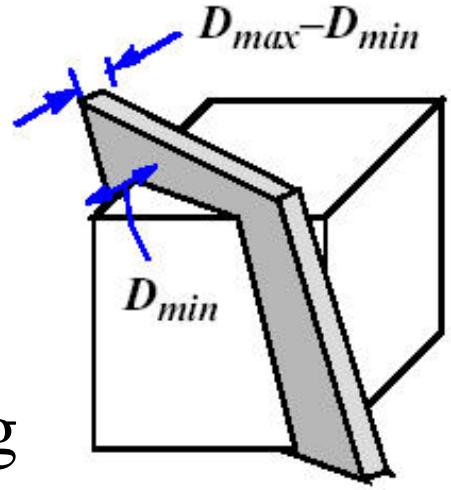
# Advanced Features of VolumePro: Hardware Cursor

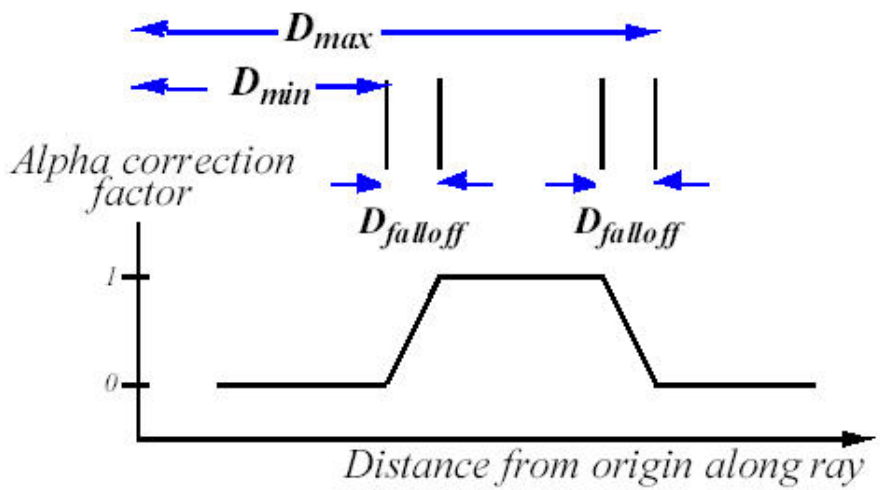
- Hardware features a built-in 3D cursor
- Generated by hardware, controlled in software
- Allows interrogation of volume
- How might we implement this cursor in hardware?
- Compositing unit generates extra voxels for cursor and composites them with volume



# Clipping Planes

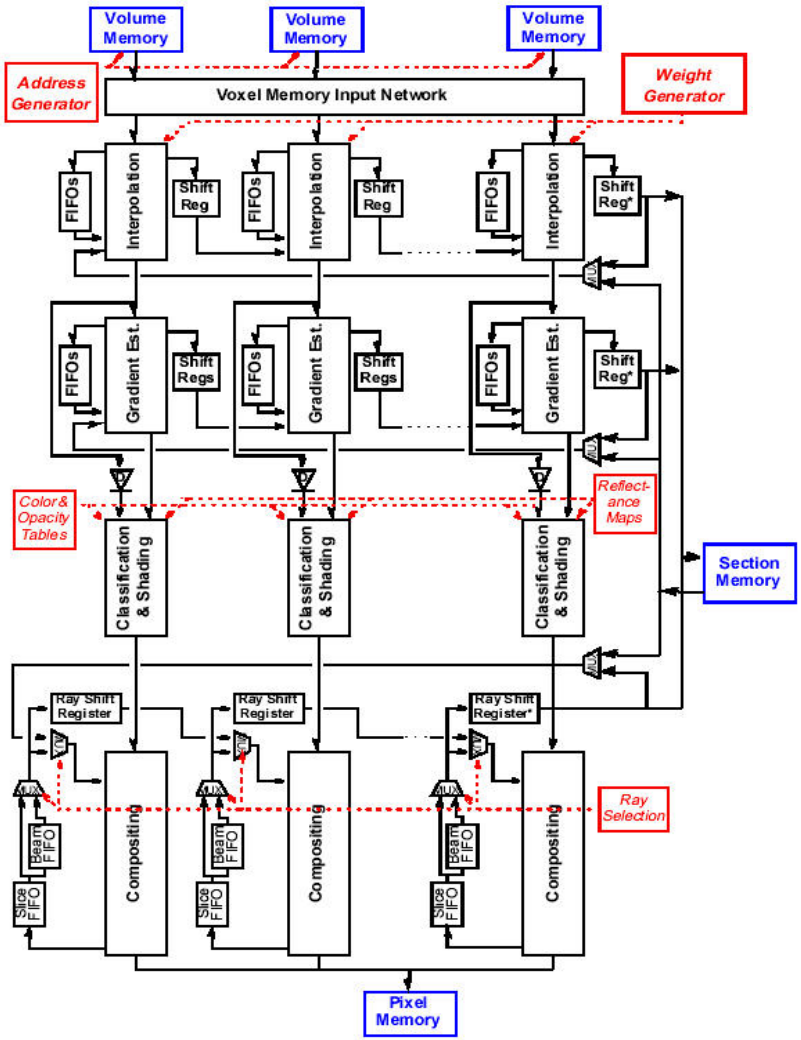
- VolumePro supports a single clipping "plane" with arbitrary thickness
- Samples visible if between two planes
- What problem might be caused by cutting data along an arbitrary plane?
- Aliasing! So how could we make "smooth cuts"?
- Create a smooth transition region between boundary of inside and outside



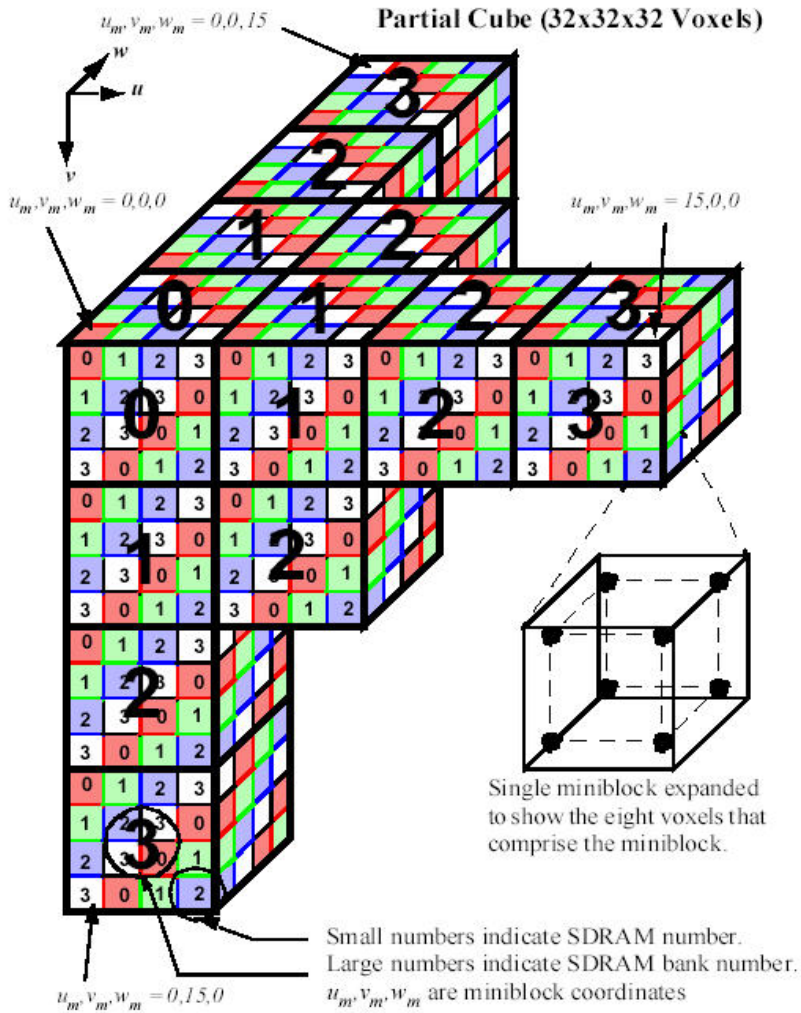
# vg500 Chip Architecture: Parallel Pipelines

- Chip implementing rendering is the *vg500*
- 3.2 million transistors, 125 Mhz clock frequency
- To render $256^3$ voxels 30 times per second, hardware processes over 500 million voxels per second
- Four parallel pipelines
- 125 million voxels/sec per pipeline



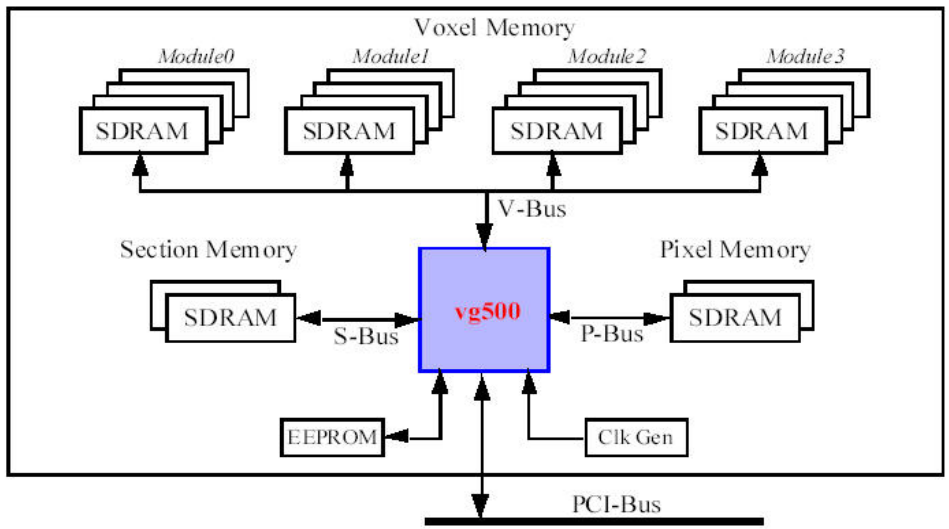
# vg500 Chip Architecture: Voxel Memory Organization

- Voxels are stored in a very intricate, *skewed* fashion to permit parallel reads and avoid bus contention
- Done to prevent delays for *any* viewpoint the user chooses
- View-independent memory layout



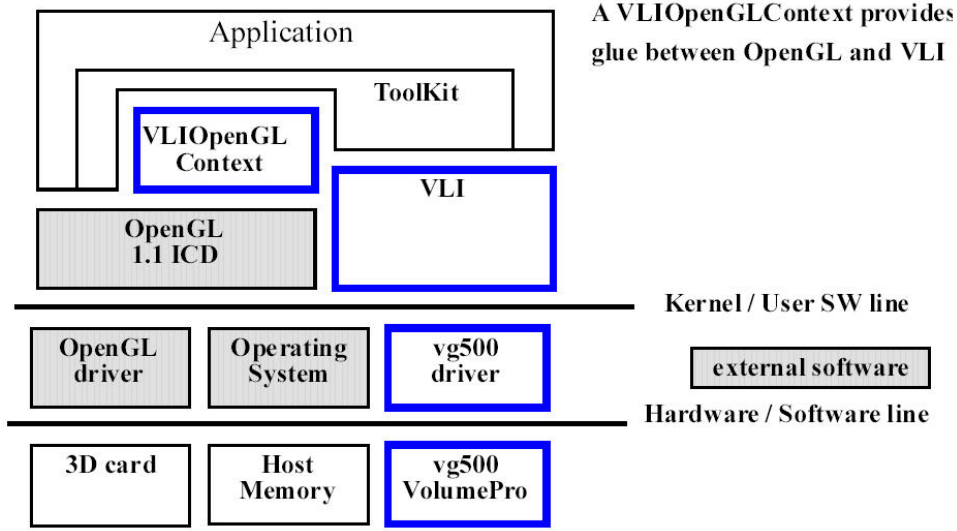
# VolumePro PCI Card

- First version released in 1999
- 66 MHz PCI bus interface
- Can connect multiple cards together via high-speed network for parallel rendering
- Separate volume across boards
- Or, can integrated multiple vg500 chips on a single, multi-processing board

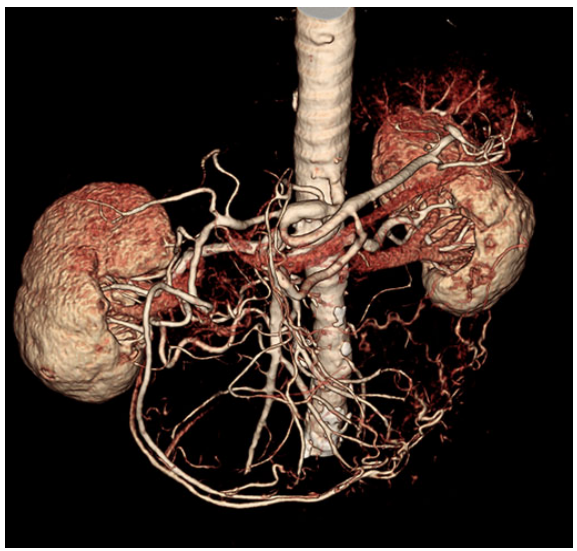


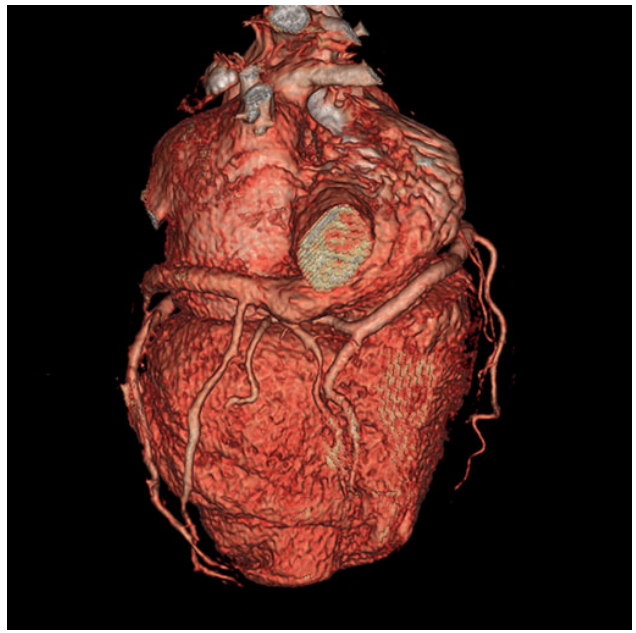
# VLI – The Volume Library Interface

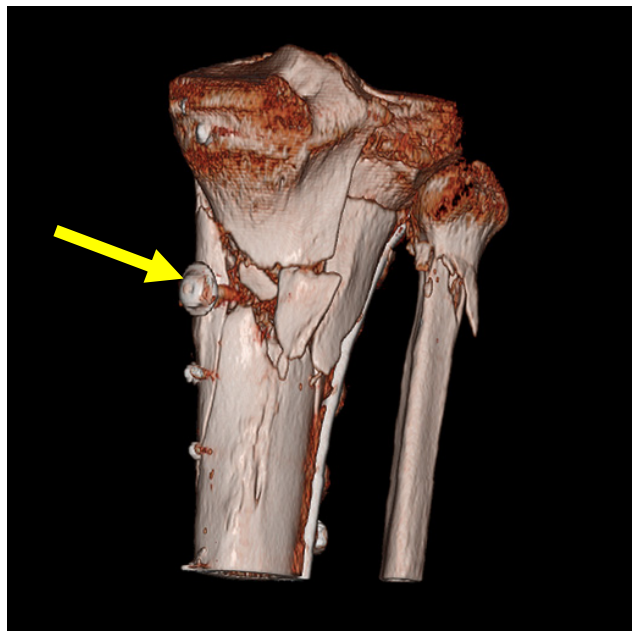
- VLI is C++ class that provide access to vg500 chip
- Works with 3D graphics library, like OpenGL
- Contains software hooks for all major functionality of hardware




# VolumePro 1000







# Vector Field Visualization

# Vector Field Visualization

- We have looked primarily at scalar field visualization
- Iso-surface extraction, volume rendering algorithms
- These algorithms do not extend to vector-valued quantities, which may have 2, 3 or more values per voxel
- What would it mean to volume-render a field of velocity vectors?
- How would we perform classification, shading, compositing, and the other stages of the pipeline?

# Vector Field Visualization

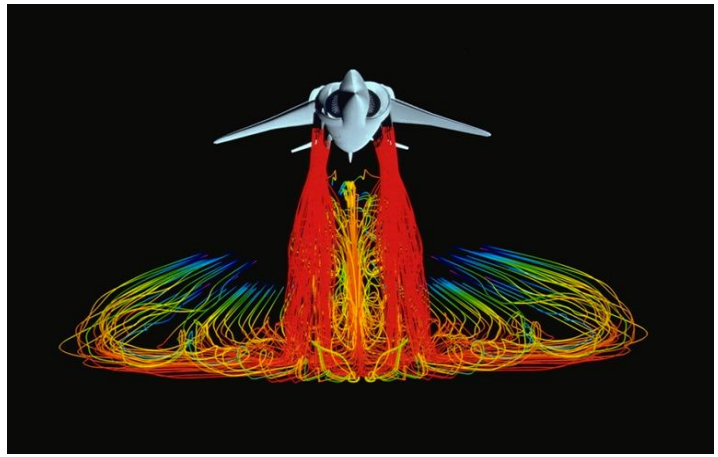
- Computational fluid dynamics (CFD) has been the classical application driving R&D in vector visualization
- Why?  Many components at a given (x,y,z) position: velocity, temperature, pressure, rotation, etc.
- Many vector field visualization techniques, some quite clever
- Remember goal of visualization: understand important aspects and features of complex data-sets

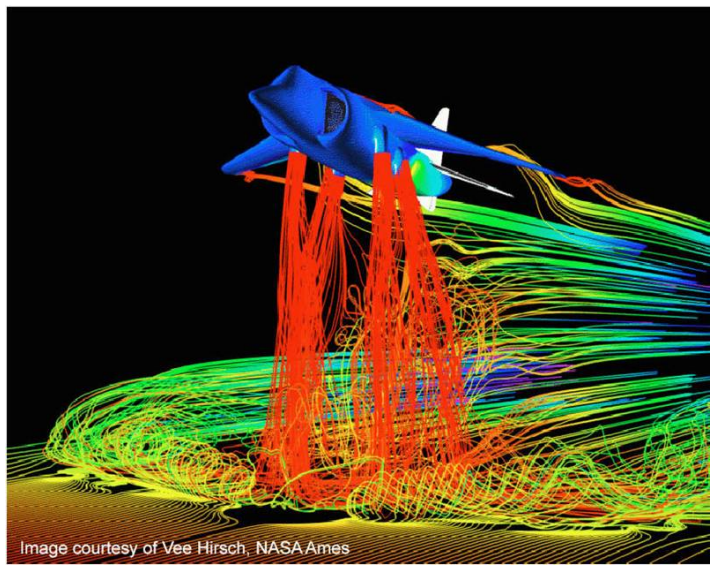
# Data Contraction

- Reduce vector-valued functions to scalar ones
- Vector magnitude
- Scalar product with a given direction vector
- Advantage: very simple technique and uses existing volume visualization
- Disadvantage: very simple technique that discards too much information

# Streamlines, Pathlines, Streaklines

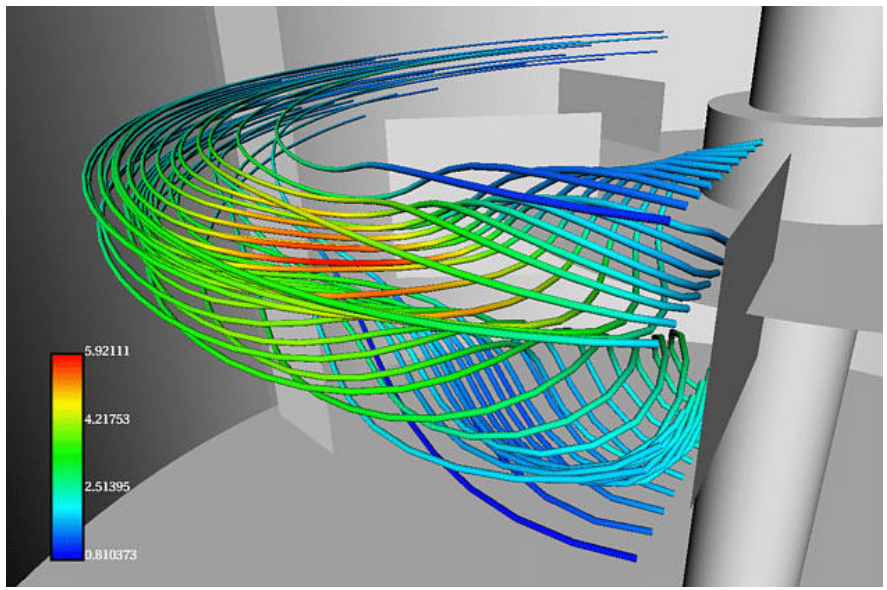
- Particle advection (line integration)
- Streamline – path always tangent to flow field
- Streamlines best used for stationary flows, flows that do not change as a function of time
- Color-coded





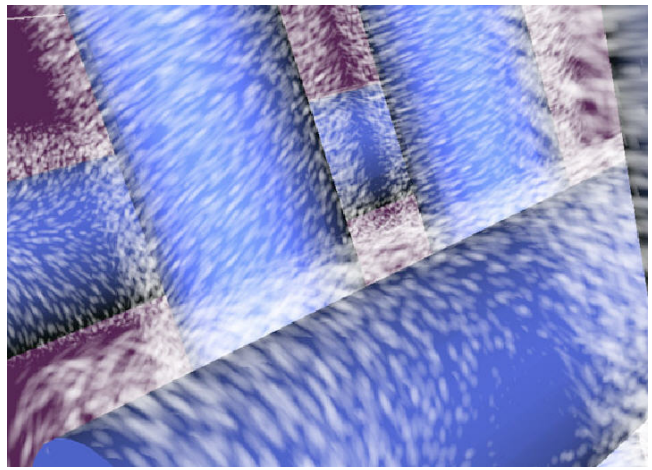
# Streamlines, Pathlines

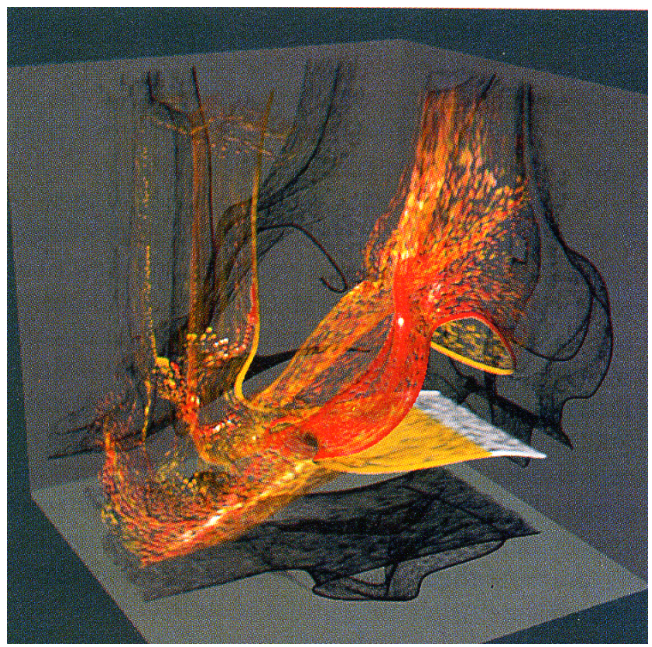
- Pathline – similar to streamline; trajectory that results if single particle is released and traced over time
- If flow is stationary (time invariant), pathline coincides exactly with the streamline at a given starting position



# Particle Systems

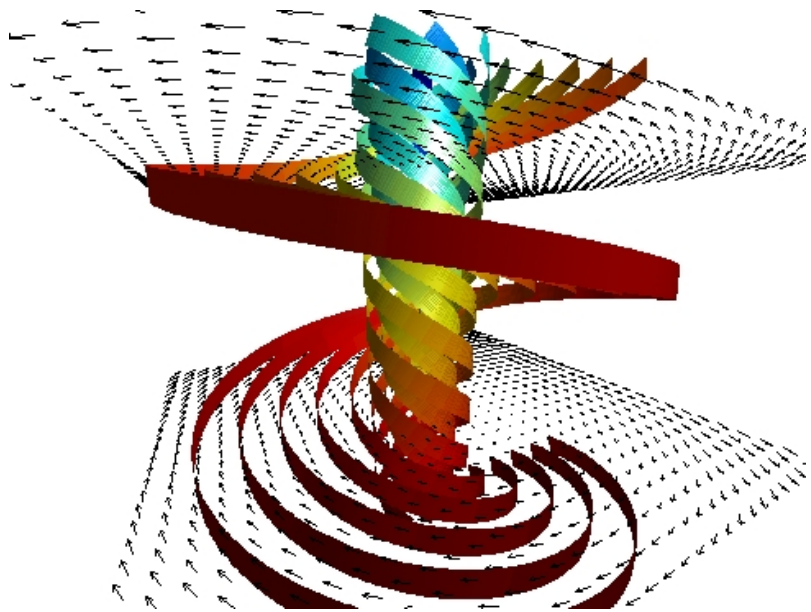
- Particles are injected into the flow field, which may be time-varying (turbulent)
- Enter, travel, leave
- Animated particles show direction and magnitude of velocity

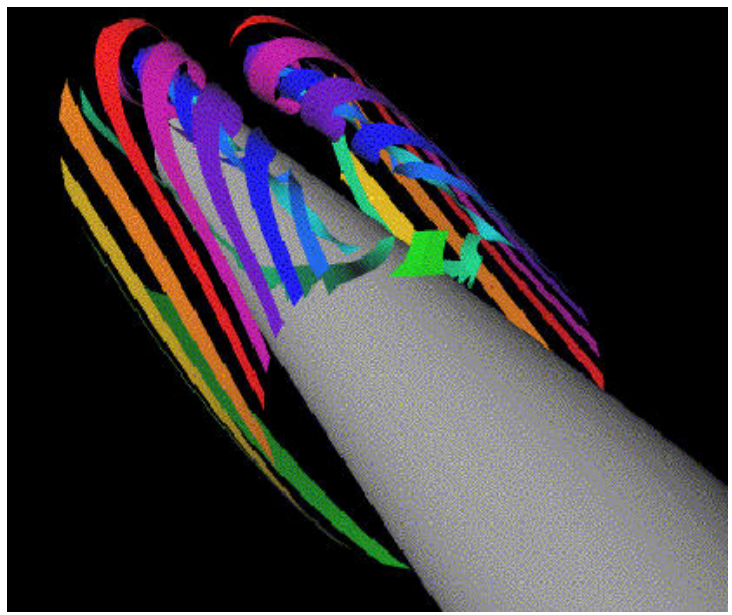




# Ribbons and Tubes

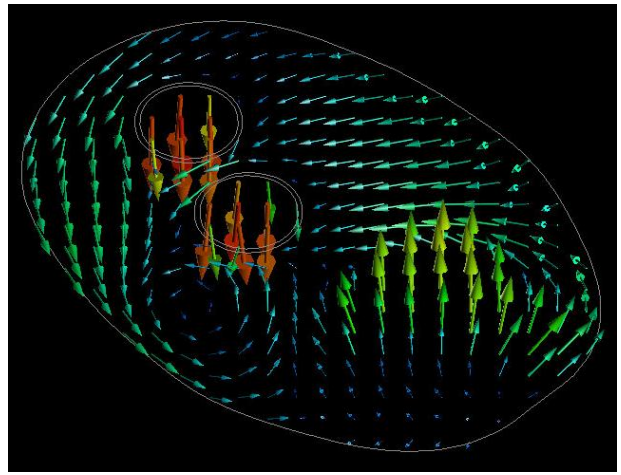
- Multiple particle advections per segment in the discretized line integration
- Connect two of them together to generate a ribbon, more to make a tube

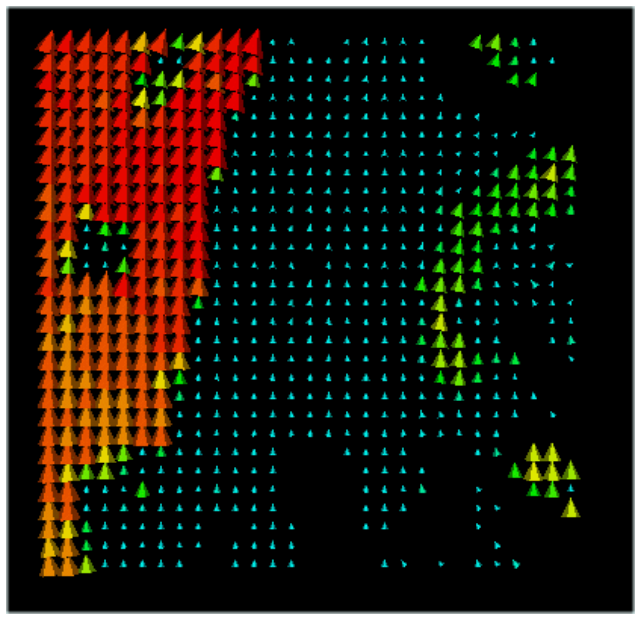




# Hedgehogs

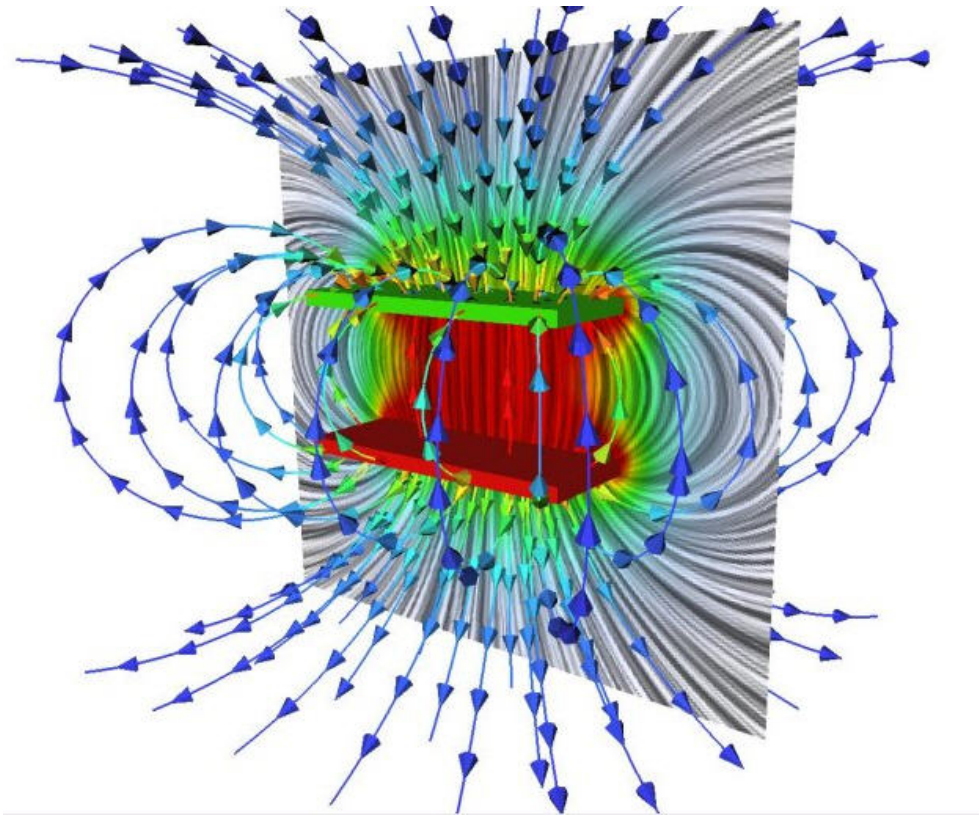
- Draw the vectors themselves
- Advantages: simple
- Disadvantages: many!
- Clutter
- Direction ambiguity
- Spatial ambiguity (start/end locations of arrow)





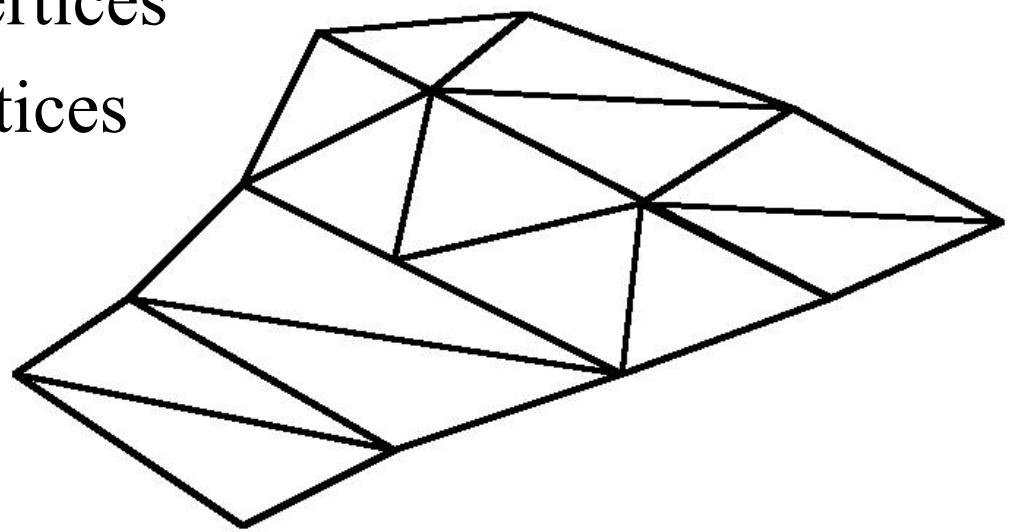
# Streamlines + Hedgehogs

- Can you identify the physical phenomenon being visualized here?

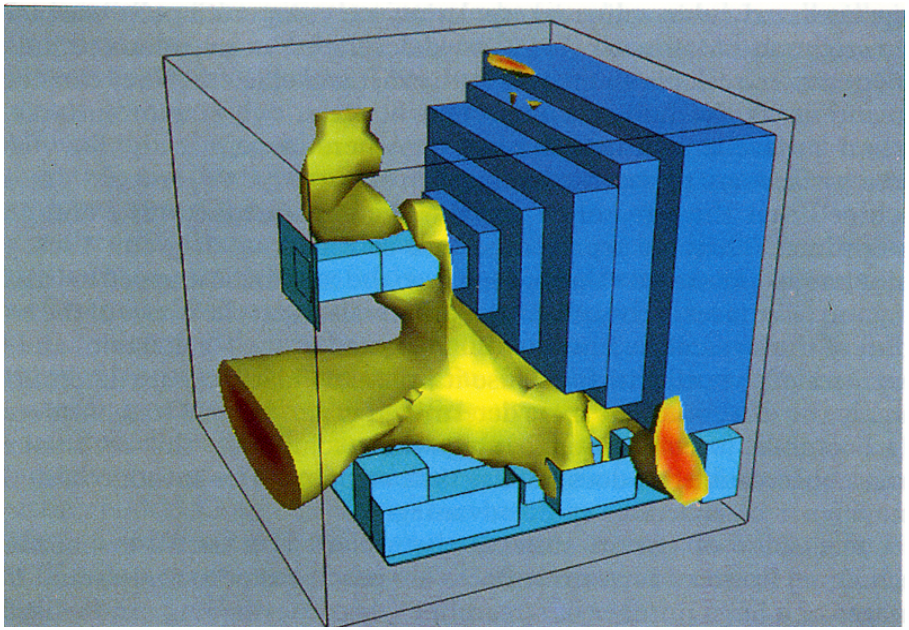


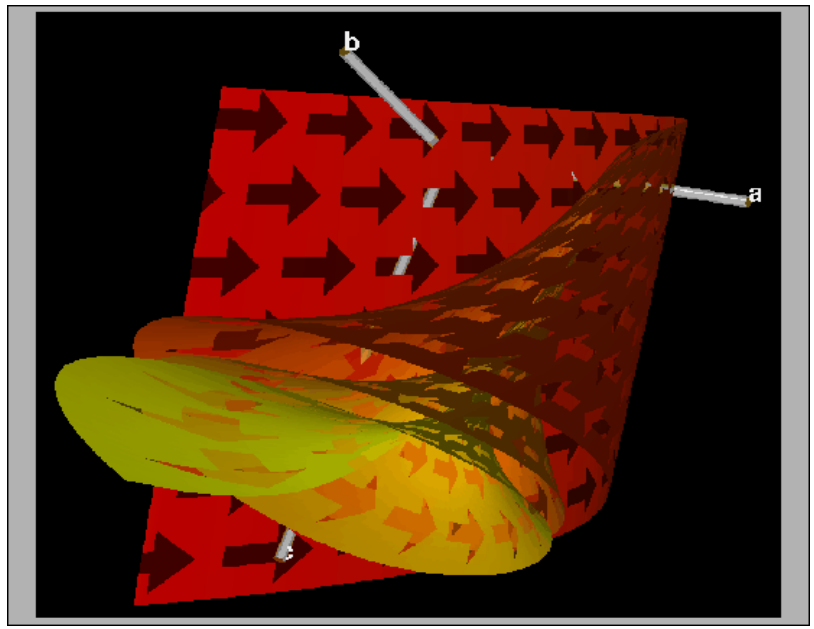
# Stream Surfaces

- Calculate multiple stream lines
- Discretize
- Connect points to form triangles
- Diverging and converging flow causes problems
- Divergence: add extra vertices
- Convergence: merge vertices



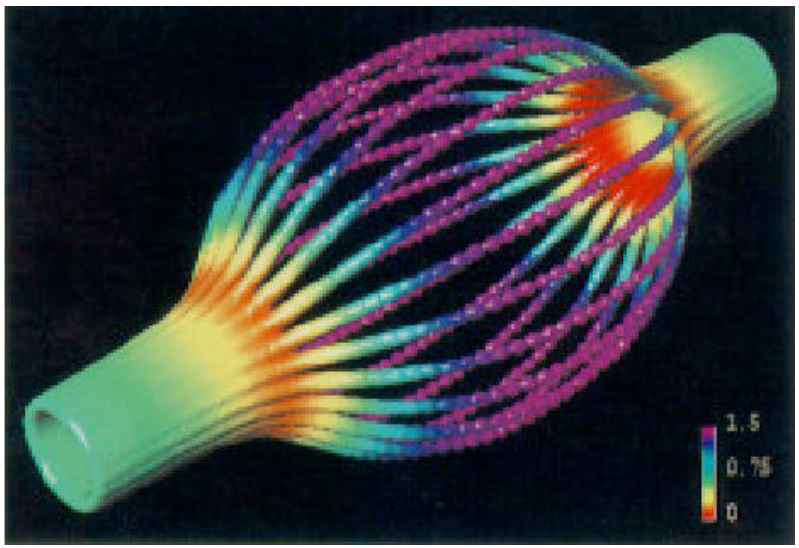
# Stream Surfaces

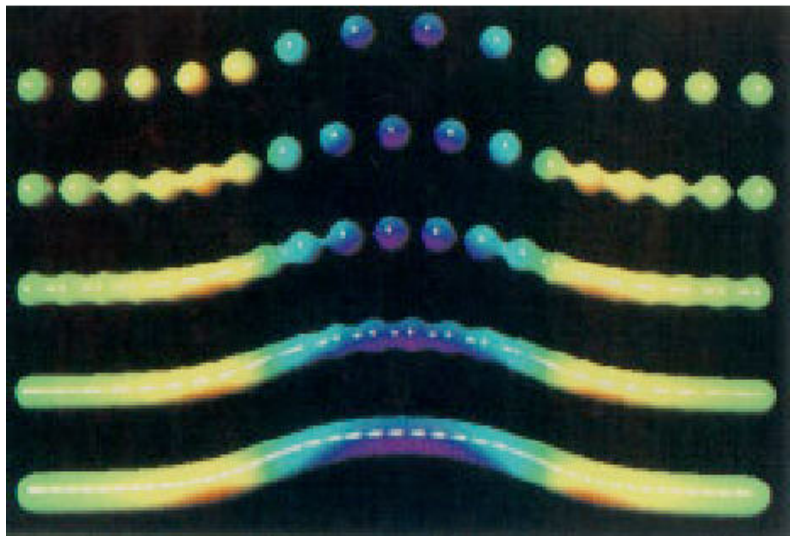




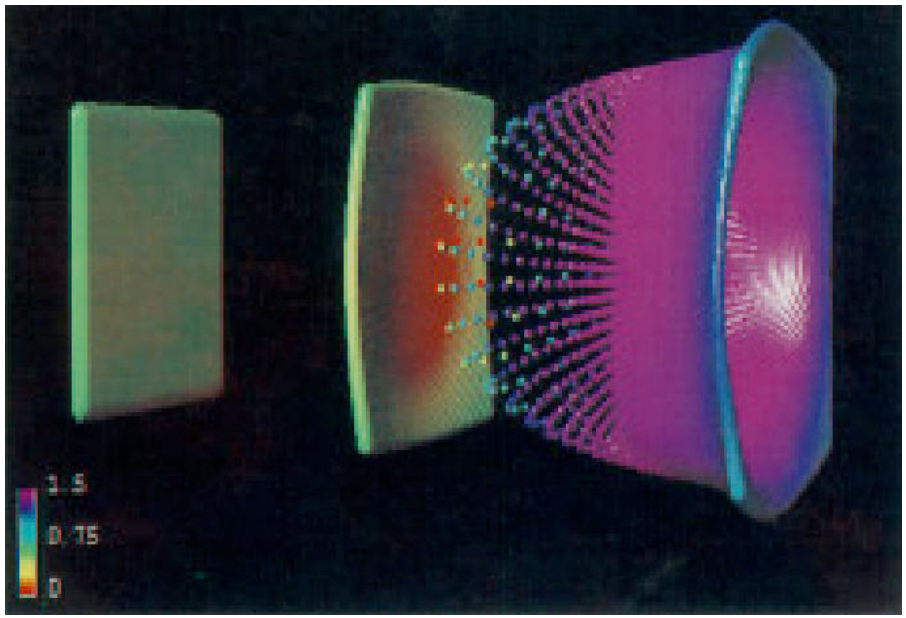
# Streamballs

- Basic idea is to create a continuous function f(x,y,z).
- Take isocontours of this function.
- Use meta-balls (not meatballs) to generate this function



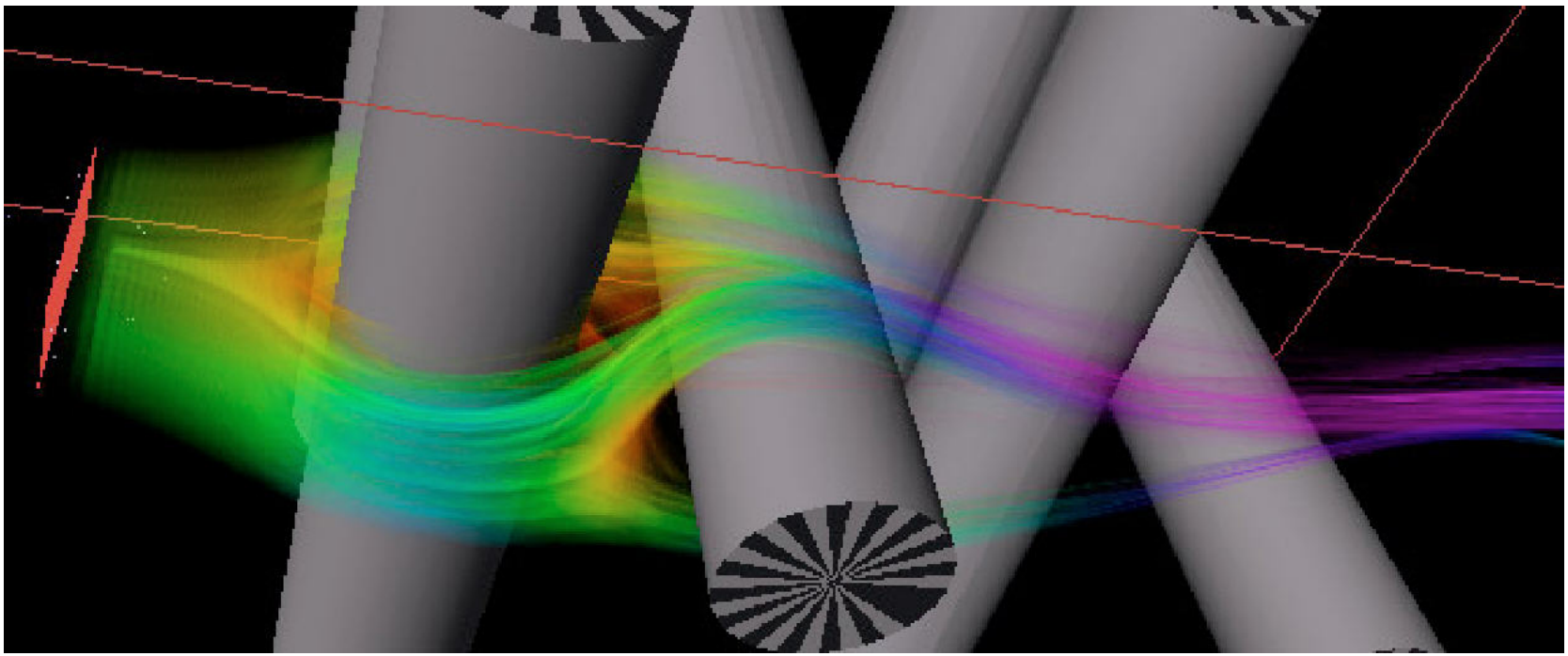



# Streamballs



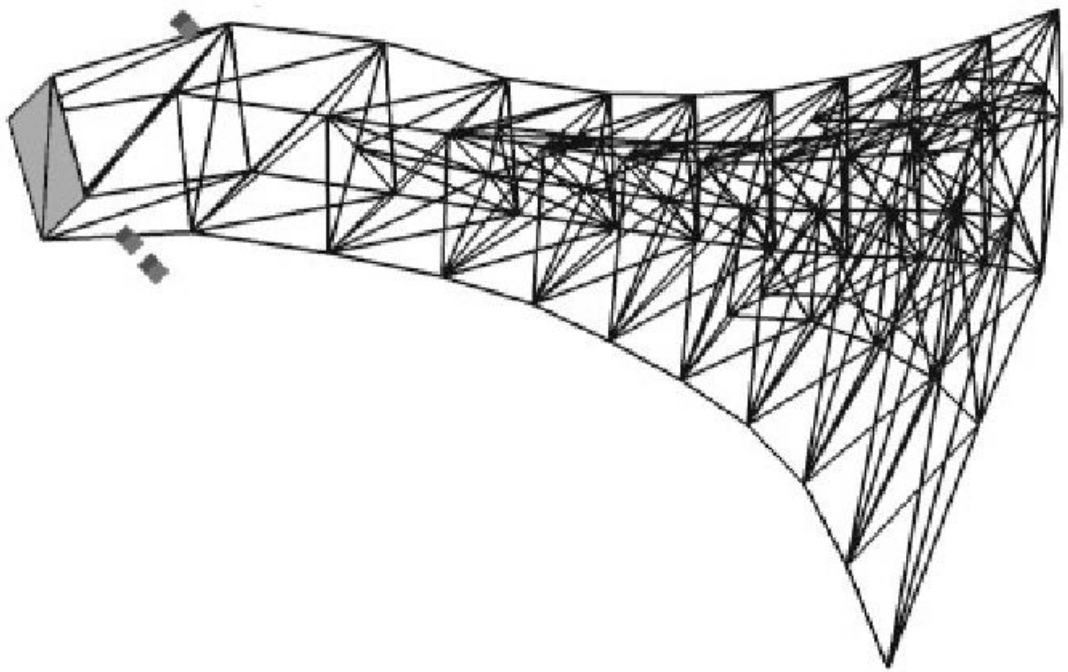

# Flow Volumes

- Imagine standing outside with a smoking flare in hand
- Smoke trail guided by wind field
- This is the basic idea of flow volumes



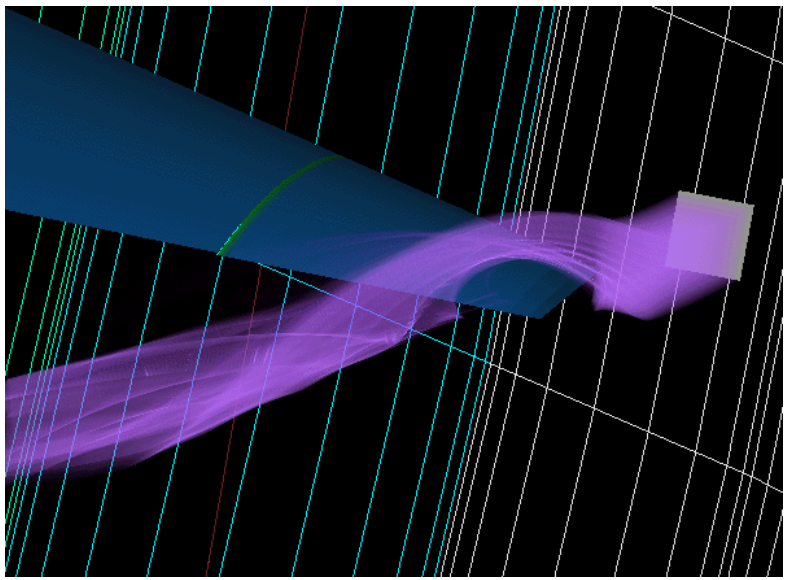
# Flow Volumes

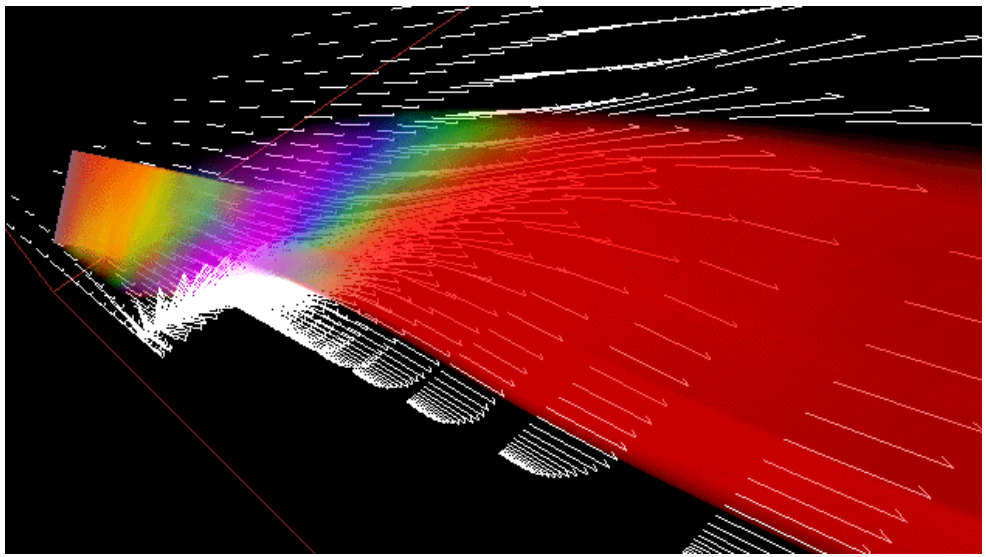
- Seed polygon (square) is used as smoke generator
- Constrained such that center is perpendicular to flow
- Square can be subdivided into a finer mesh



# Flow Volumes

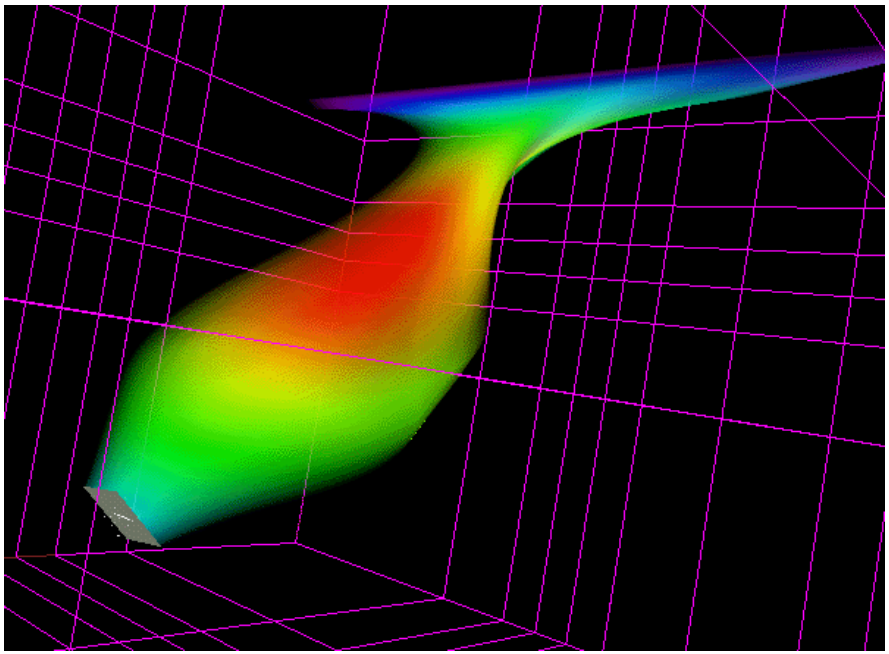
- Fast rendering on commodity hardware
- Can color the smoke to indicate other quantities

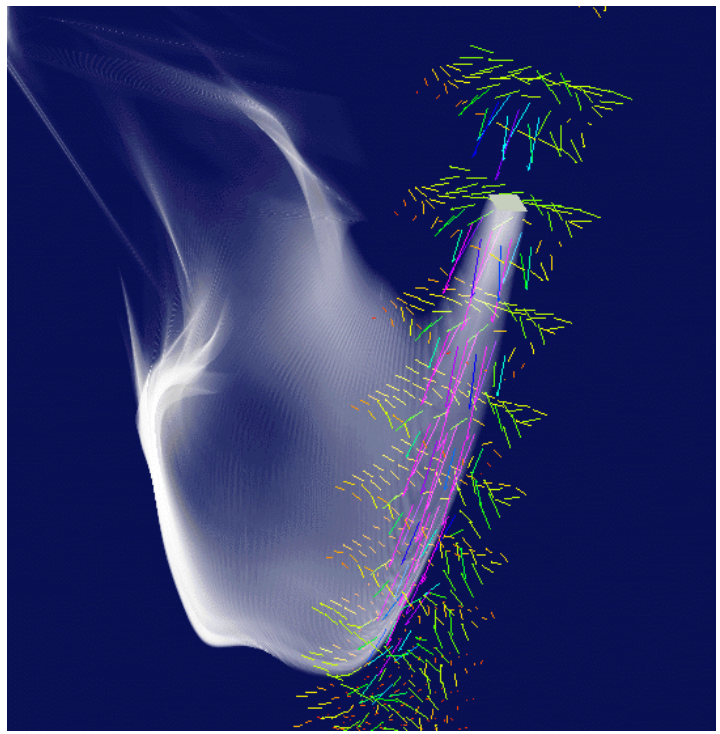




# Flow Volumes

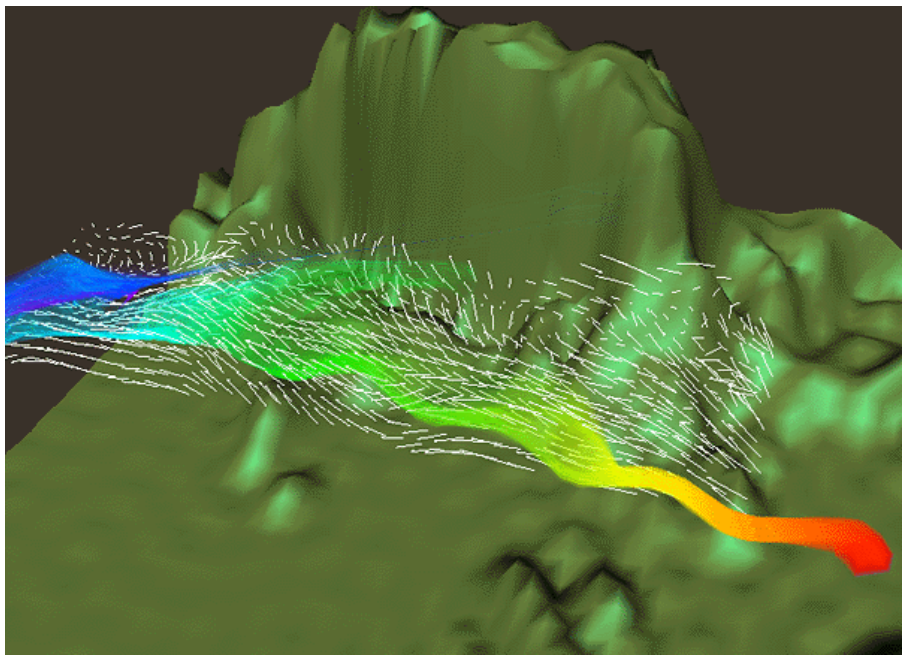
- Currently defined for regular, rectilinear, curvilinear, multigrid and unsteady meshes

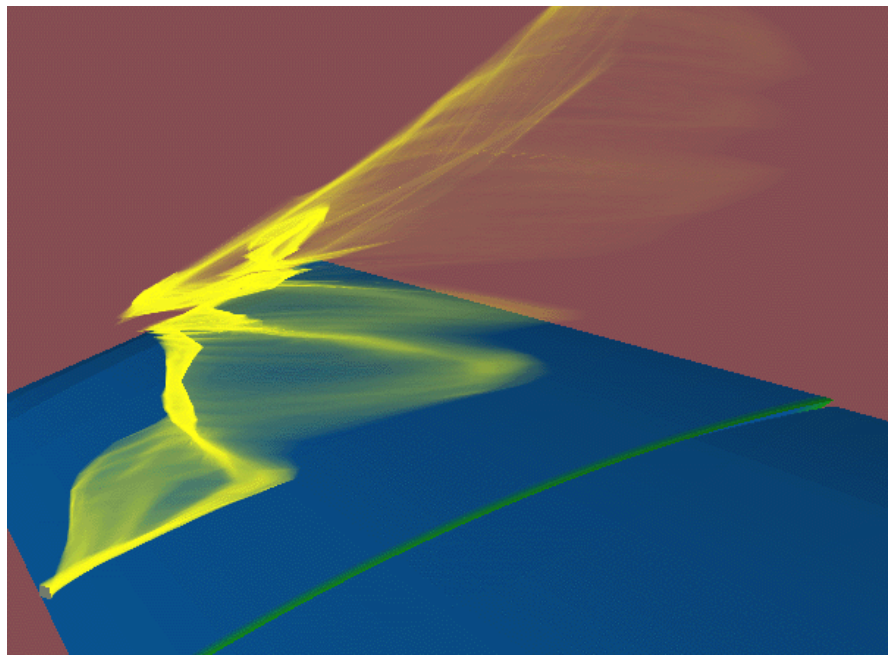




# Flow Volumes – Unsteady Flows

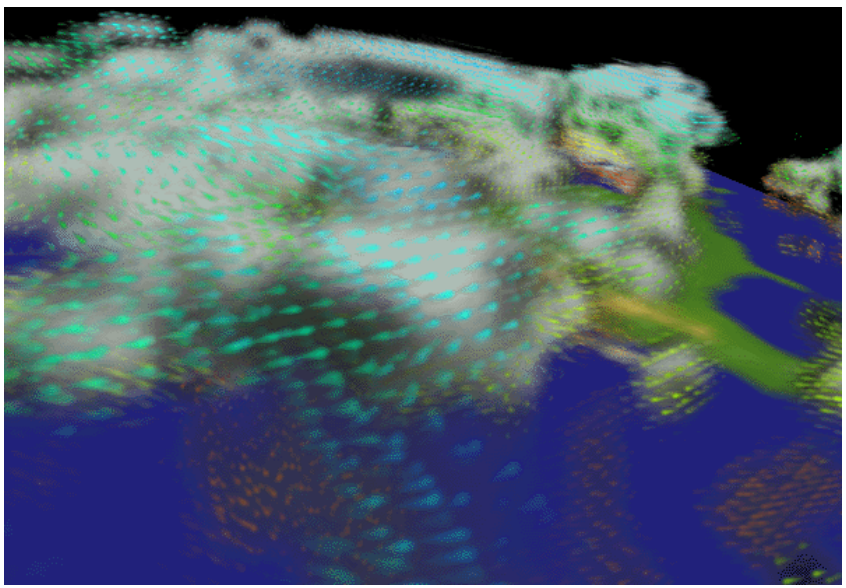
- Can work for unsteady flows for all mesh types (curvilinear, rectilinear, irregular, etc.)
- Complex twisting must be handled carefully

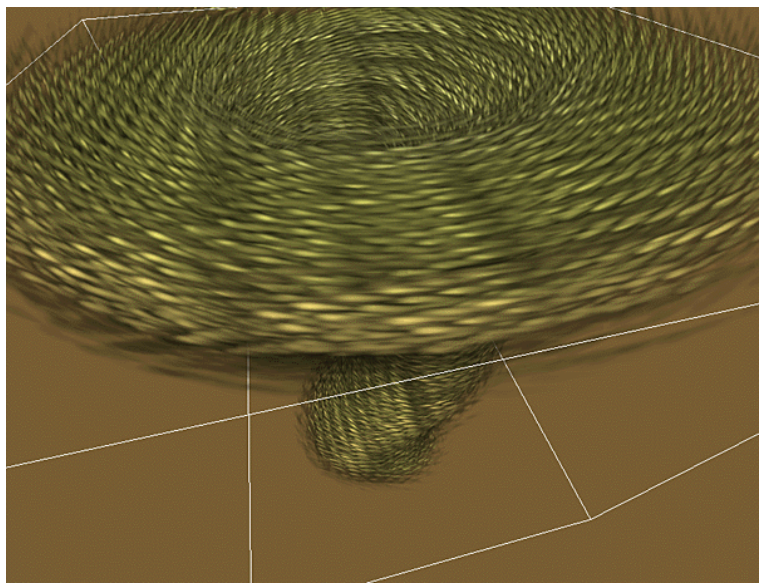




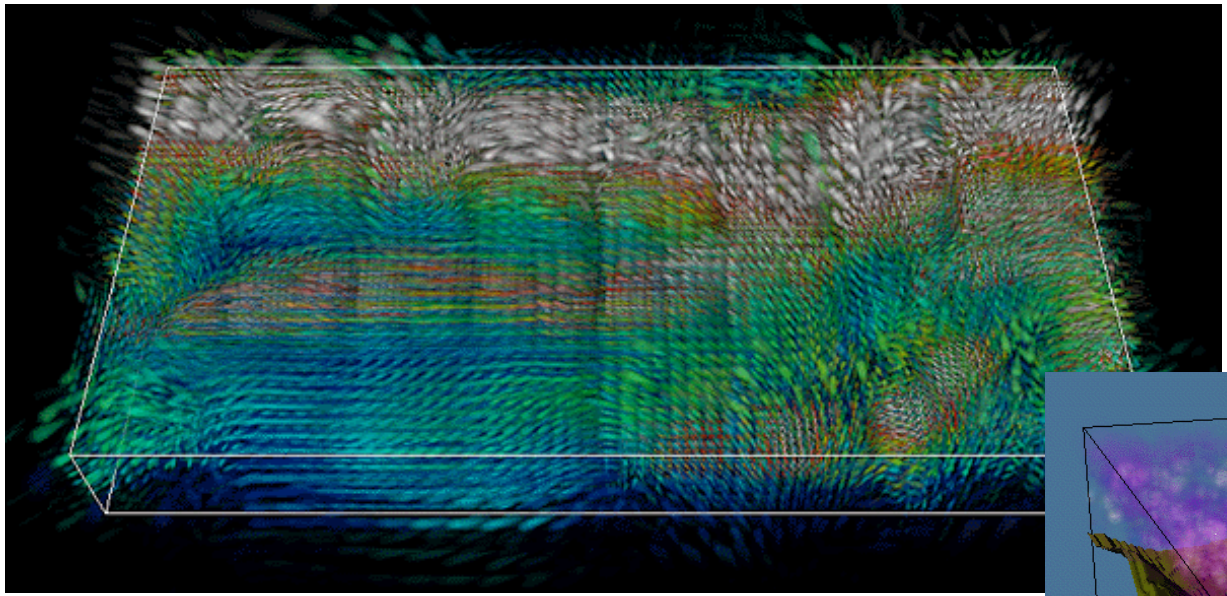
# Textured Splats

- Basic idea: map reconstruction footprint from splatting to a 2D textured square
- Splat textures oriented in projected direction of flow

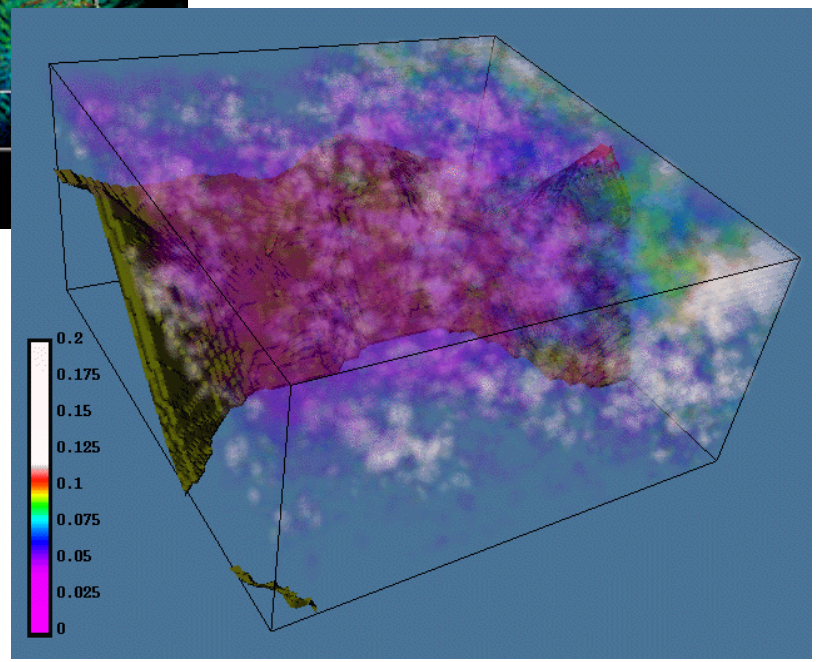




# Textured Splats



Wind direction and magnitude




Soil conductivity