

Computer Graphics

Dr. Norman I. Badler
Professor, Computer & Information Science
Director, SIG Center for Computer Graphics

University of Pennsylvania

CIS 460 / CIS 560

Fall 2013

Part A

© Norman I. Badler

(except where otherwise noted)

What do (nearly all) these 50 Top Worldwide Grossing Movies have in Common?

1. Avatar (2009)
2. Titanic (1997)
3. Marvel's The Avengers (2012)
4. Harry Potter and the Deathly Hallows, Part 2 (2011)
5. Iron Man 3 (2013)
6. Transformers: Dark of the Moon (2011)
7. The Lord of the Rings: The Return of the King (2003)
8. Skyfall (2012)
9. The Dark Knight Rises (2012)
10. Pirates of the Caribbean: Dead Man's Chest (2006)
11. Toy Story 3 (2010)
12. Pirates of the Caribbean: On Stranger Tides (2011)
13. Star Wars: Episode I - The Phantom Menace (1999)
14. Alice in Wonderland (2010)
15. The Hobbit: An Unexpected Journey (2012)
16. The Dark Knight (2008)
17. Harry Potter and the Sorcerer's Stone (2001)
18. Jurassic Park (1993)
19. Pirates of the Caribbean: At World's End (2007)
20. Harry Potter and the Deathly Hallows, Part 1 (2010)
21. The Lion King (1994)
22. Harry Potter and the Order of the Phoenix (2007)
23. Harry Potter and the Half-Blood Prince (2009)
24. The Lord of the Rings: The Two Towers (2002)
25. Finding Nemo (2003)
26. Shrek 2 (2004)
27. Harry Potter and the Goblet of Fire (2005)
28. Spider-Man 3 (2007)
29. Ice Age: Dawn of the Dinosaurs (2009)
30. Harry Potter and the Chamber of Secrets (2002)
31. Ice Age: Continental Drift (2012)
32. The Lord of the Rings: The Fellowship of the Ring (2001)
33. Star Wars: Episode III - Revenge of the Sith (2005)
34. Transformers: Revenge of the Fallen (2009)
35. The Twilight Saga: Breaking Dawn, Part 2 (2012)
36. Inception (2010)
37. Spider-Man (2002)
38. Independence Day (1996)
39. Shrek the Third (2007)
40. Harry Potter and the Prisoner of Azkaban (2004)
41. E. T. The Extra-Terrestrial (1982)
42. Indiana Jones and the Kingdom of the Crystal Skull (2008)
43. Spider-Man 2 (2004)
44. Star Wars: Episode IV - A New Hope (1977)
45. 2012 (2009)
46. The Da Vinci Code (2006)
47. Shrek Forever After (2010)
48. The Amazing Spider-Man (2012)
49. The Chronicles of Narnia: The Lion, the Witch and the Wardrobe (2005)
50. The Matrix Reloaded (2003)

What do all of these Top Video Games have in Common?

Duh

<http://www.gamerankings.com/browse.html>

Great Ideas in Computer Graphics

- Computers (with suitable output devices) can draw geometric stuff, not just manipulate numbers.
- Computers can draw images of 3D worlds with realistic shapes and light and animate them as well.
- People can create 2D and 3D models.
- People can interact with them in 2D and 3D through innate visual and kinesthetic senses.
- Computers can be fun (games).
- Computers can make the virtual appear real (special effects).
- Computer graphics can sell computers.
- All that can fit on a low cost PC graphics board.
- All that can fit into a small mobile platform.

Challenge and Opportunity...

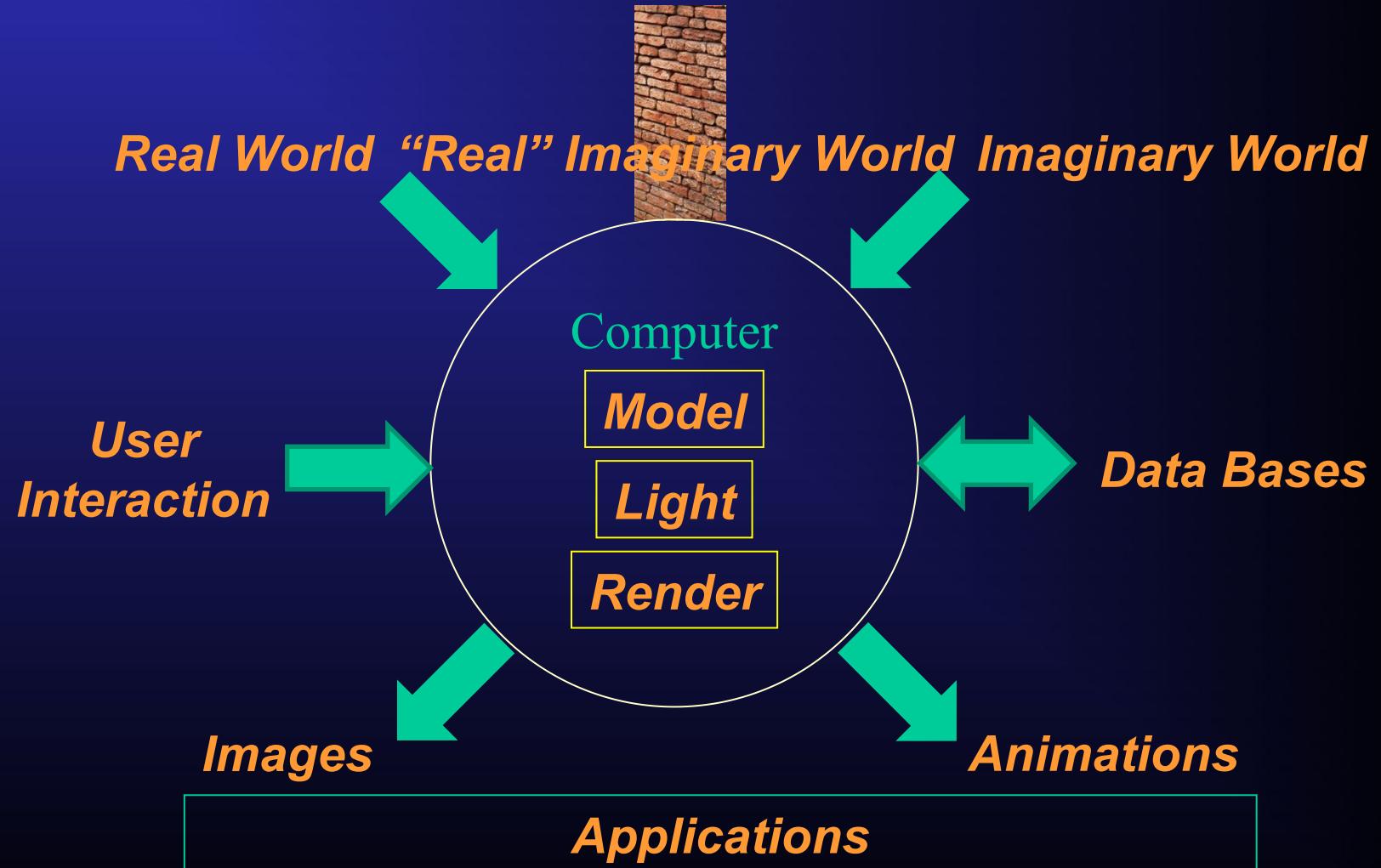
News Item, USA Today, Sept. 23, 2000:

“The angry woman with children in tow had some choice words for the theme park marketing director. Where were the live dinosaurs she and her kids saw in the park's TV commercials? She had paid good money, only to view fake ones.”

What will the Next 10 Years Bring?

- *Ours may be the last generation that can see a difference between real and virtual things.*

Computer Graphics



Computer Graphics Timeline (Very Brief)

- 1960 Phrase “Computer Graphics” (Fetter)
- 1963 First CG film (Bell Labs)
- 1972 Hand and face computer animations (Catmull, Parke)
- 1976 3D in movies (Futureworld)
- 1977 First home computers with graphics (Apple, TRS-80)
- 1982 Silicon Graphics workstation with VLSI graphics chips
- 1984 Global illumination algorithm
- 1984 Human motion capture (Brilliance ad for Superbowl)
- 1993 Jurassic Park (first CG effects movie that made a profit)
- 1995 First full 3D animated movie (Toy Story)
- 1999 nVidia’s GPU

See also: <http://design.osu.edu/carlson/history/timeline.html>

What's this Course About?

- How Computer Graphics “works”.
 - Not a “user” course.
 - Algorithms, mathematics, techniques, examples, and taxonomies.
- Simulation of “reality”.
 - Computer graphics is a field that cannot be neatly packaged in a few main theorems or algorithms.
 - But there are important fundamental principles.
 - There are many techniques (algorithms, data representations).
 - Techniques are combined in a combinatorial explosion of options.
 - ***Reality isn't simple, either!*** (Even physics needs 100+ elements)

What are You Programming in CIS460/560?

Programming assignments and exams:

1. Cloud rendering: Fill a voxel grid with procedural clouds, then render it with a volume renderer.
2. Scene graph: Create and edit a scene graph populated by geometric primitives.

Midterm Exam

3. Meshes and textures: Generate and color meshes from extrusions and surface revolutions.
4. Simple shaders and vertex buffer objects.
5. Raytracing: Render the scene graph with a raytracer capable of reflection, diffuse shading, and shadows.

Final Exam

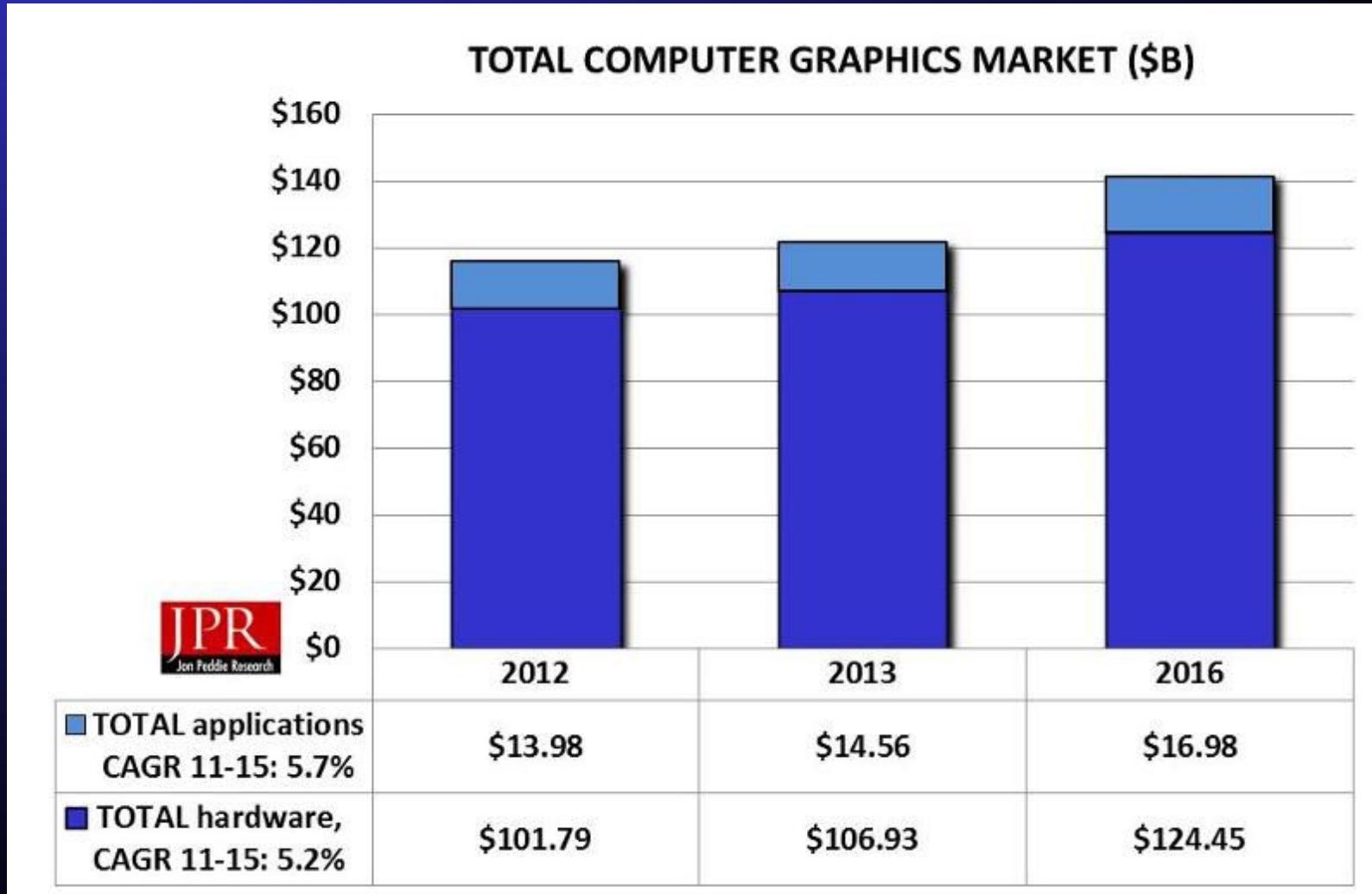
Graphics Roots: Applications and Influences

- Graphic Design
 - User interfaces, multimedia, image processing.
- Engineering and Architecture
 - 2D and 3D modeling (CAD), CAE, CAM, database exchange.
- Medicine and Science
 - Scientific visualization, interactive data exploration, networks.
- Simulation and Training
 - VR, real-time, very large databases, distributed simulations.
- Education
 - Usability, GUI, experimentation.
- Entertainment
 - Games, realism, characters, image quality, special effects, art.

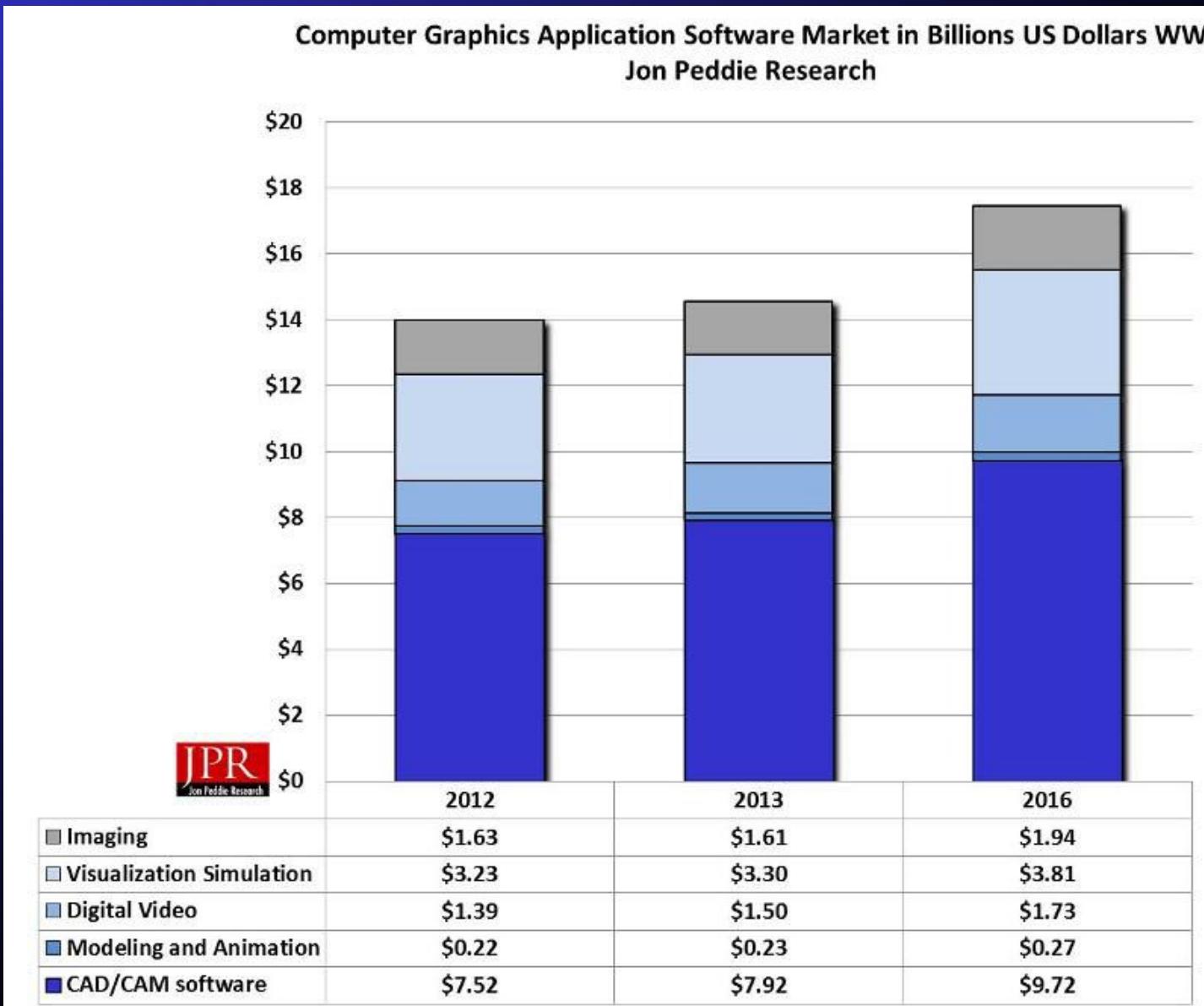
What are the Current Drivers?

- Technology:
 - Off-line animation (CGI/VFX: Visual Special Effects)
 - Real-time animation (Games)
 - Mobile platforms (Smart phones)
- Economics:
 - Games
 - Special effects
 - Consumer products
 - Digital photography
 - Web applications
 - Mobile applications
 - User sensing and interaction
 - VR and AR

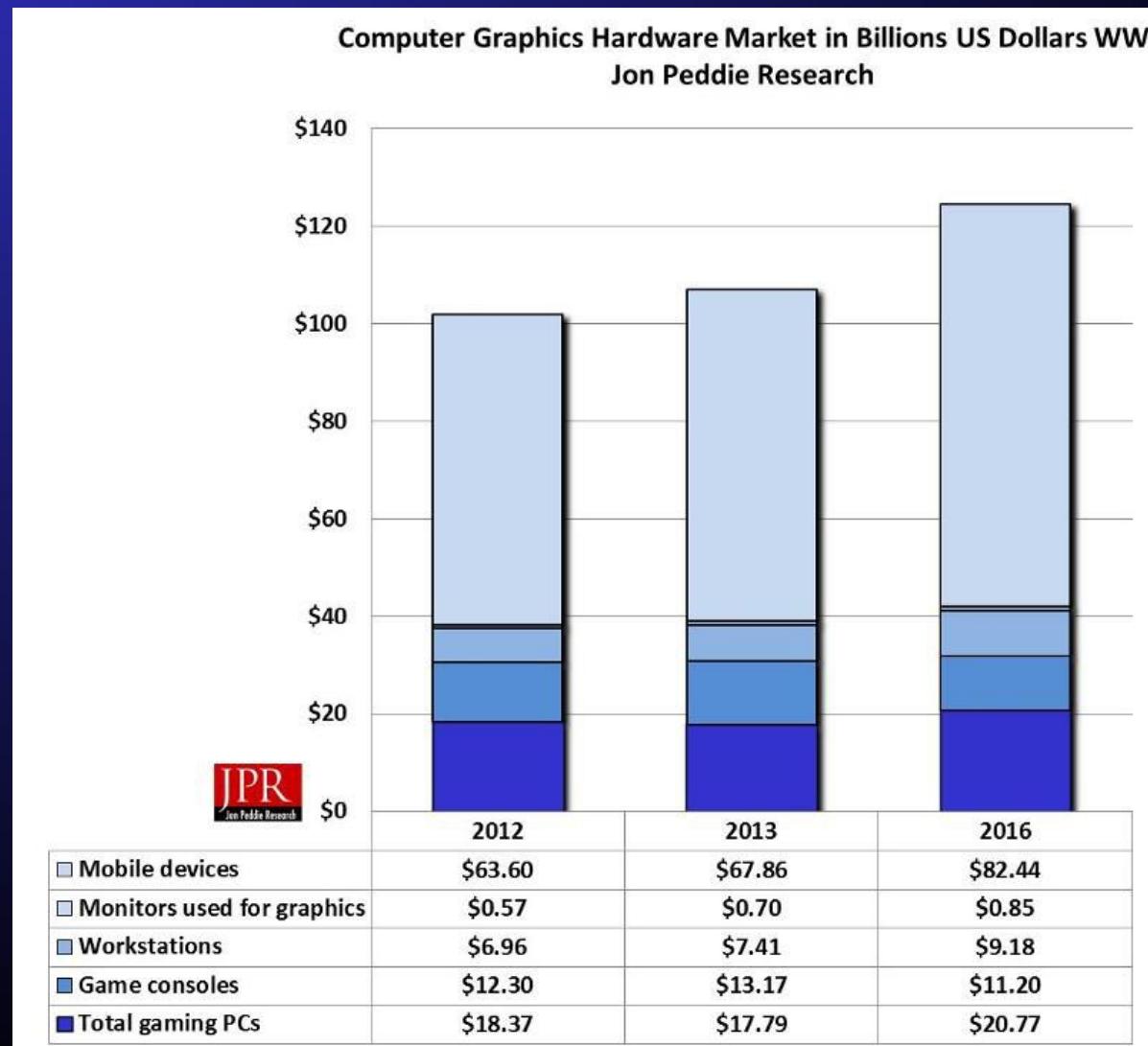
Computer Graphics Market



Computer Graphics Application Software Market



Computer Graphics Hardware Market



Outline

- Great Ideas ✓
- Virtual Environments & Devices
- 3D Perception
- Architecture, Color & Rasterization
- Vector Geometry & Transformations
- 3D Modeling
- Embedding, Hierarchies & Contours
- Model Generation & Deformation
- Visible Surface Algorithms
- Polygon Algorithms
- Image Synthesis & Shading
- Mapping
- 3D Viewing Transformations
- Anti-aliasing & Compositing
- Global Illumination
- Light Fields & HDR
- Path Tracing/Photon Mapping
- Futures

Virtual Environments –Virtual Reality – Games Empowered by Interactive Computer Graphics

- Three-dimensional
- Virtually substantive
- New displays and wearable devices
- Immersive / non-immersive
- Interactive / real-time
- Multi-modal
- Networked
- Agents, characters, and avatars
- Applications

Three-Dimensional

- Visualize the world as we know it.
- Explore higher dimensional worlds.
- Builds from Computer Graphics research and technology of the past 60 years.
- Enabled by high performance, low cost workstations and especially graphics cards.
- Continues to drive computer hardware (board- and chip-level) improvements.

Virtually Substantive

- Gives the appearance of solidity and form.
- Objects can be based on real things or imaginary ones.
- Collision detection simulates solidity.
- Collisions can influence object behavior or feedback to user.
- Collisions can result in geometric object changes (deformation, explosion, trigger other animation).
- Collisions can be used to synthesize sound.

New Displays and Wearable/Interactive Devices

- Mobile phones/computing platforms/apps
- Cameras (still and video)
- Unobtrusive wearable displays
- Motion capture suits (and less obtrusive methods)
- Image and depth sensors
- Body state sensing devices and feedback
- Flexible displays

Wearable “Smart” Cameras

- Microsoft Cambridge SenseCam.
- Takes low-res wide angle photo every 30 sec. or when triggered by various on-board sensors (sound, temperature, motion).
- Visual record of experience.



<http://research.microsoft.com/en-us/um/cambridge/projects/sensecam/video.htm>

Wearable Displays

Google Glass

- Camera
- Microphone
- Wireless communications
- Tiny display screen



Motion Capture

- Active Suit, e.g., Xsens Movens (accelerometers and magnetometers).
- Passive, e.g. Vicon (Cameras and LED markers).
- Video only, e.g. OrganicMotion.
- Wearable cameras and computer vision (CMU).



<http://www.youtube.com/watch?v=DVQ9MPjk5dI&feature=related>

Motion Capture Uses

- Enables data-driven animations (games)
- 3D motion data is re-usable, modifiable, and adaptable to other characters.
- User inputs for interactivity and virtual worlds.

Consumer Products: Nintendo Wii; Microsoft Kinect; LEAP

- Accelerometers
- Optical depth and motion sensing
- Inexpensive!



Flexible Displays: Based on Organic Light Emitting Diodes (OLED)



Psychophysical Experience

- Non-immersive or desk-top “VR”: The 2D computer screen.
- Immersive “Through the window” display (airplane cockpit).
- Immersive head-mounted displays (HMD) (helmets, goggles).
- Whole body immersive: CAVE® or other multi-projection rooms with imagery on walls, ceiling, floor.
- Augmented (Mixed) Reality: See the virtual superimposed on and registered with the real world.

Virtual Reality (VR) Individual Displays

- VR goggles or helmets
 - Complete replacement of visual field with synthetic imagery
 - Limited field of view
 - Cybersickness issues
 - Cautious navigation

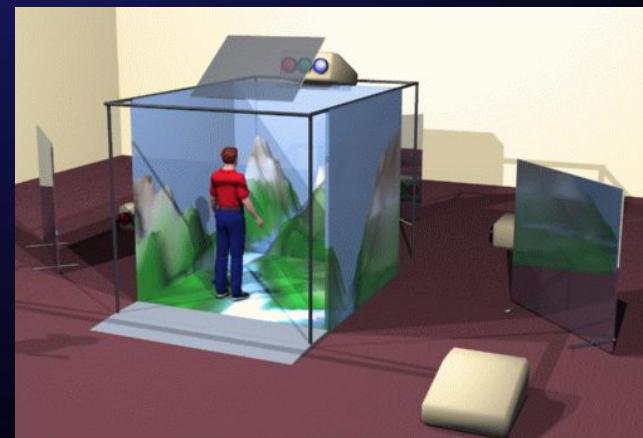


Virtual Reality (VR) Displays (Multiple Screens)

- One or more people can see and experience
- Single POV issue

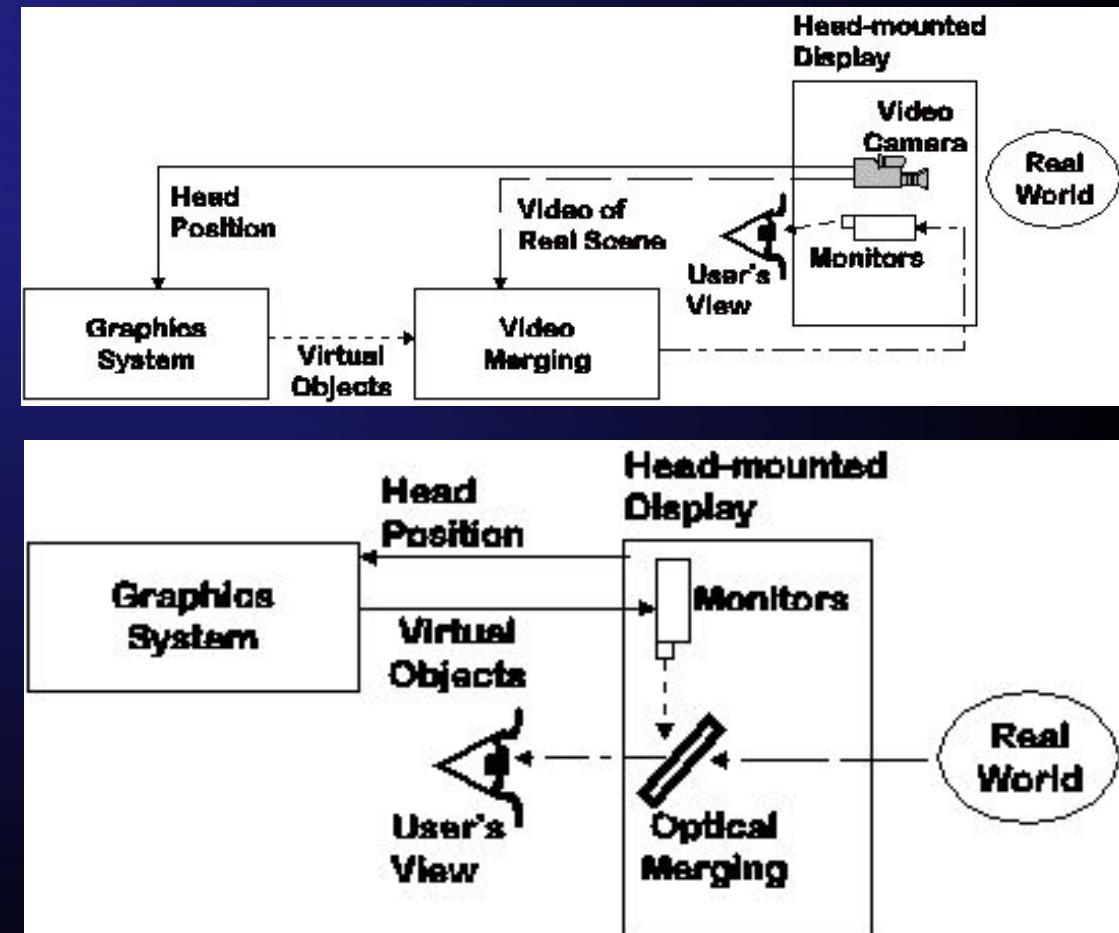


“CAVE”®



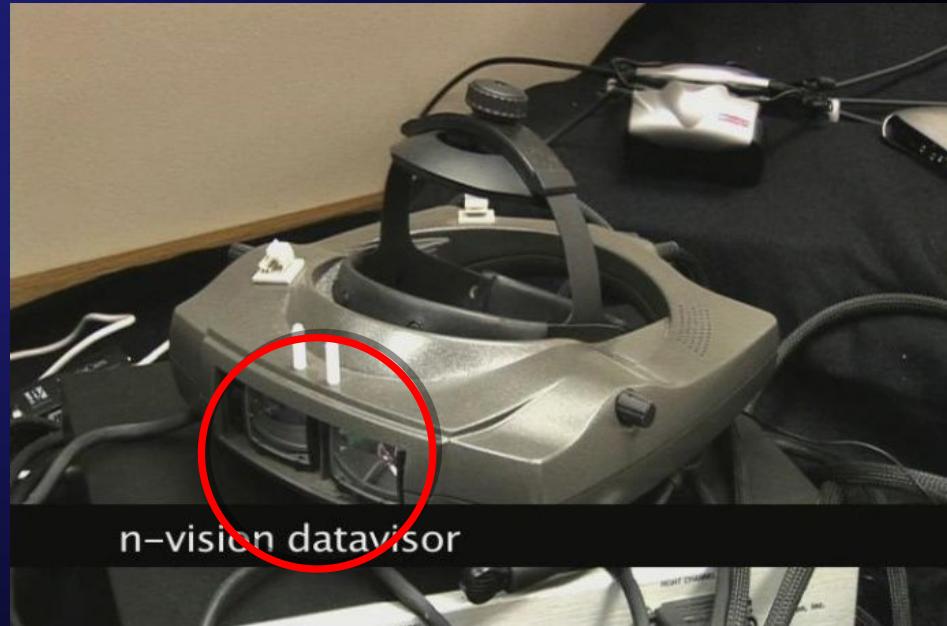
Augmented Reality (AR) Displays

- AR with video cameras
- AR with optical merging



Augmented Reality Wearable Display

- Video version
- Scene limited to video resolution (not so good).
- Primary issue is real/virtual image registration.



Augmented Reality with Portable Devices

- Use the embedded camera and overlap synthesized images and animation.
- Need real-time feature tracking for registration.

“Invisible Train”:

Schmalstieg and
Reitmayr, 2004

- Many AR apps



http://www.youtube.com/watch?v=zOS5Mb_k_Iuc

Overlaid
directions

Mixed and Augmented Reality in the Large: ICT Flatworld

- Modular physical environment with rear-projection screens as doors, windows, or backgrounds.
- Uses visual depth discontinuity to create visual experience realism.



Interactive / Real-Time

- Real-Time \equiv Motion at time of creation (as opposed to off-line development and re-display).
- System responds at human rate of effort or communication.
- System moves the environment at natural rates.
- Minimum ~ 30 Hz (frames per second); ≥ 60 Hz best.
- Worlds contain moving and changing objects and actors.

Multi-Modal

- Feedback to human senses (sight, sound, touch, force, smell,...).
- Various challenges to do this effectively (cost, latency, technology, encumbrances,...)
- Sight and sound “easiest”; tactile and force (haptics) possible but more expensive.
- Human factors issues: Stereo/monocular vision, localized sound, “simulator sickness”, eye strain, other impairments.

Networked

- Individual experience versus group experience.
- Enhanced reality when virtual world controlled by or populated with other players.
- Team training without having the team together.
- Provide remote access to virtual world (Distributed Interactive Simulation, multi-player online gaming, etc.).
- Provide shared experience for separated players (multi-player world games, military simulations, training).
- Issues include network bandwidth, database updates, localized detail, client-server vs. peer-to-peer architectures, etc.

Agents, Characters, and Avatars

- Agents are objects that effect changes in the world.
- Characters are (articulated/jointed/deformable) objects that resemble humans, animals, monsters, etc.
- Avatars are characters “inhabited” by a live participant.
- Active research areas: advanced “smarts”, game “AI”, group/crowd behaviors, learning.

Virtual Environment Applications

- Visualization and immersion.
- Created reality experiences (science, theme park rides, virtual navigation,...).
- Cockpit-like training (through the window): aircraft, cars, trucks, tractors, other vehicles.
- Combat or exploration training (whole body but limited actions): games, war simulations, team coordination (search and rescue, police, fire, ...)
- Physical skill acquisition: Assembly, disassembly, repair, machinery operation, factory operations, virtual surgery.
- Design and evaluation.

Outline

- Great Ideas ✓
- Virtual Environments & Devices ✓
- 3D Perception
- Architecture, Color & Rasterization
- Vector Geometry & Transformations
- 3D Modeling
- Embedding, Hierarchies & Contours
- Model Generation & Deformation
- Visible Surface Algorithms
- Polygon Algorithms
- Image Synthesis & Shading
- Mapping
- 3D Viewing Transformations
- Anti-aliasing & Compositing
- Global Illumination
- Light Fields & HDR
- Path Tracing/Photon Mapping
- Futures

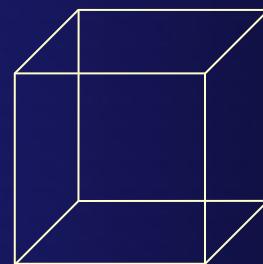
Visual Perception of 3D and Depth Cues

Issues:

- Displays almost always 2 dimensional.
- Depth cues (psychophysical factors) needed to visually restore the third dimension.
- Need to portray planar, curved, textured, translucent, etc. surfaces.
- Role of light and shadow.

1. Eliminate Hidden Parts (lines or surfaces)

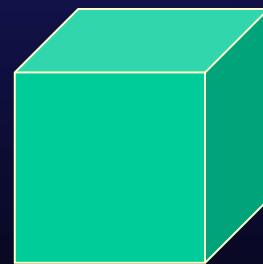
“Wire-frame”



Front?

Back?

“Opaque Object”



Convex?

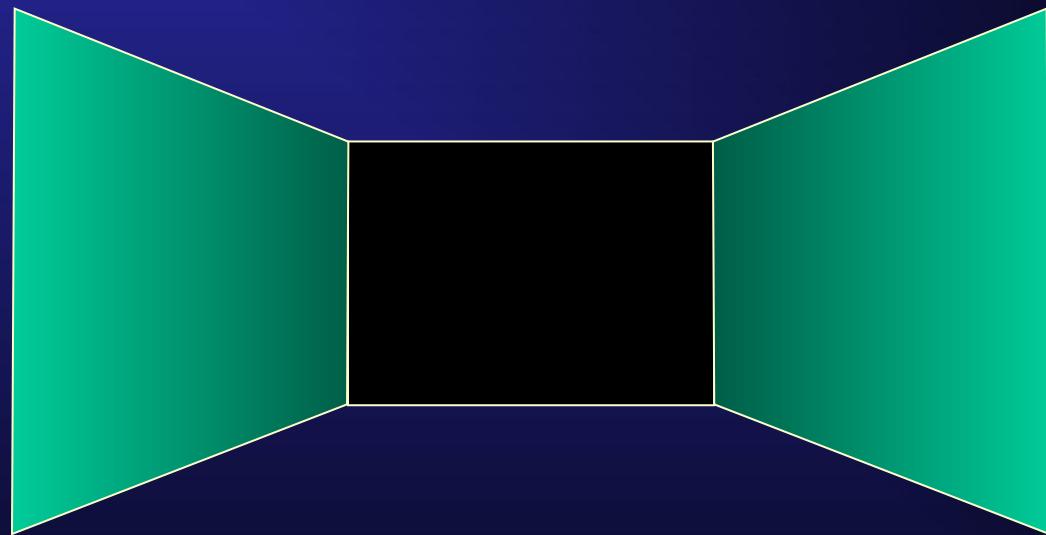
Concave?

2. Perspective Foreshortening



Objects of *same shape but different sizes* tend to be perceived as *same size but different depth!*

3. Texture and Brightness Gradients



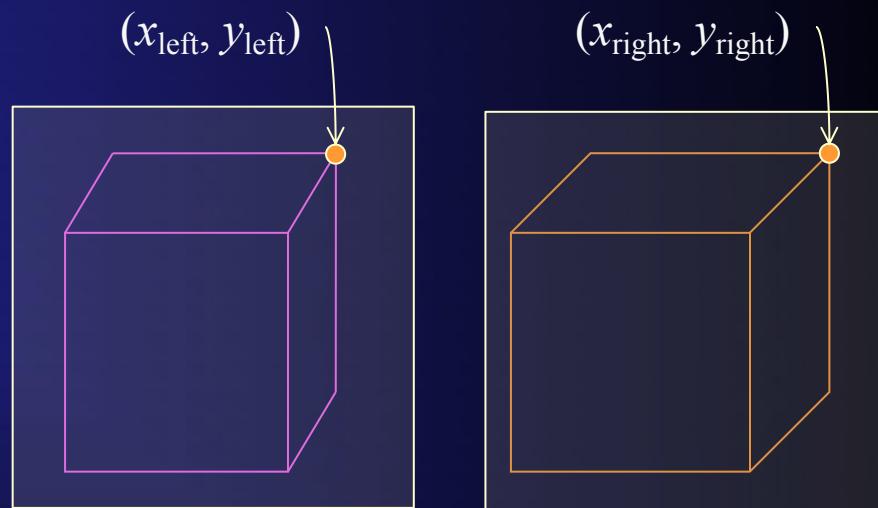
4. Shading implies Curvature



5. Stereo

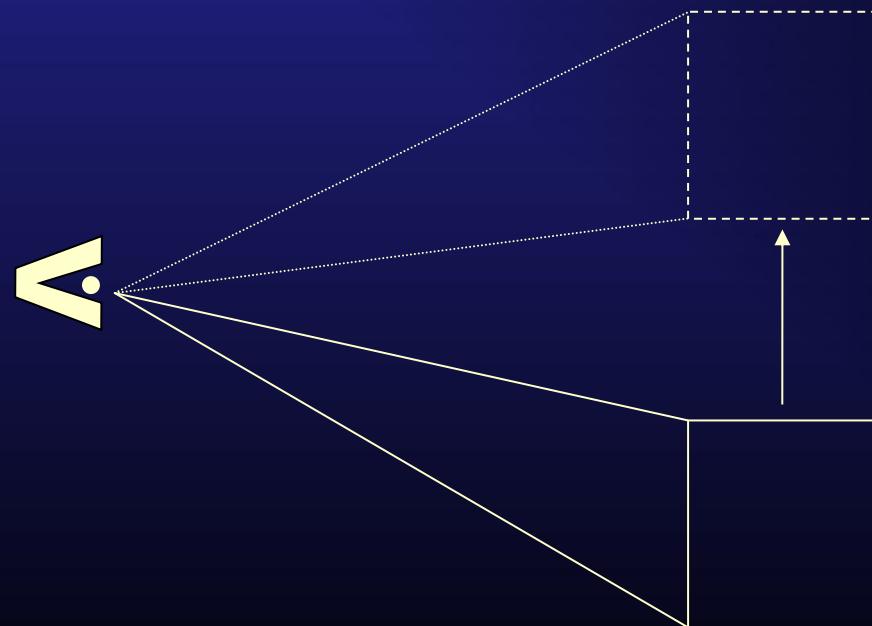
- Disparity in position of the same feature on two images allows computation of the (approximate) intersection of the two eye-feature rays in 3D space.
- Range limited to about 20-30 feet

Eye views are
different



6. Motion Parallax

Similar to stereo, except one eye (monocular) and a moving baseline or a moving object (or both).



Computer Graphics Techniques for Depth Cues

(\checkmark = can do)

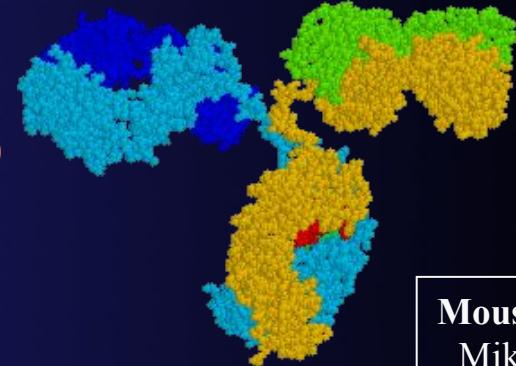
1. Visible surface synthesis. \checkmark
2. Perspective viewing transformation. \checkmark
3. Shading computation, for example, simple intensity cueing. \checkmark



Suppose Object-a and Object-b have same intensity or color, then display intensity divided by relative depth:

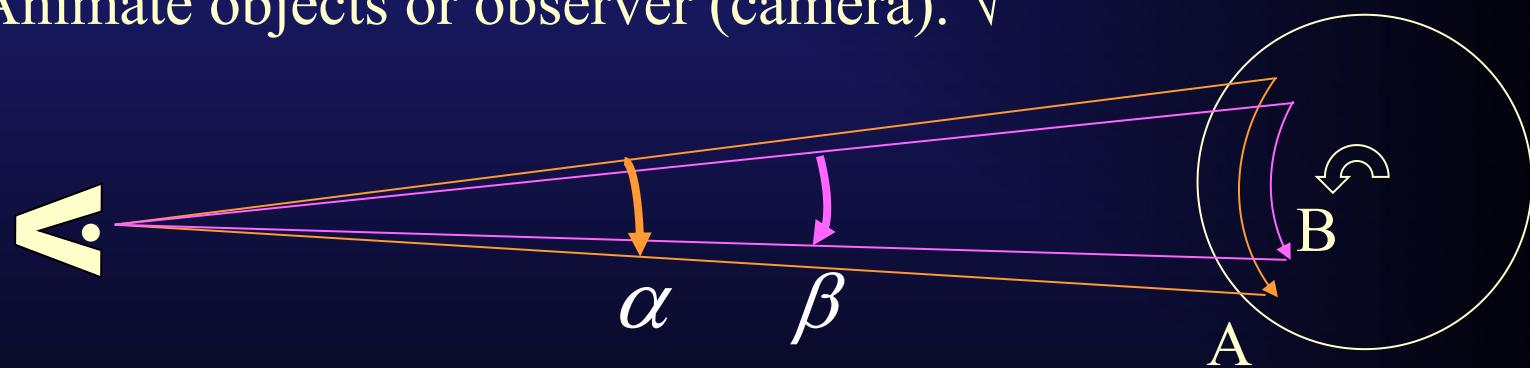
$$\frac{I}{Z_a} \gtreqless \frac{I}{Z_b} \quad \text{for} \quad Z_b \gtreqless Z_a$$

Depth Techniques (Cont'd)



Mouse IgG2a
Mike Clark

4. More important shading computation based on surface normal direction. ✓
5. Stereo view: just apply perspective viewing transformation once for each eye. ✓
6. Animate objects or observer (camera). ✓



$\alpha > \beta \Rightarrow$ A moves faster per unit time than B

faster implies closer!

Outline

- Great Ideas ✓
- Virtual Environments & Devices ✓
- 3D Perception ✓
- Architecture, Color & Rasterization
- Vector Geometry & Transformations
- 3D Modeling
- Embedding, Hierarchies & Contours
- Model Generation & Deformation
- Visible Surface Algorithms
- Polygon Algorithms
- Image Synthesis & Shading
- Mapping
- 3D Viewing Transformations
- Anti-aliasing & Compositing
- Global Illumination
- Light Fields & HDR
- Path Tracing/Photon Mapping
- Futures

Computer Graphics Hardware: A Very Simple Architectural Overview

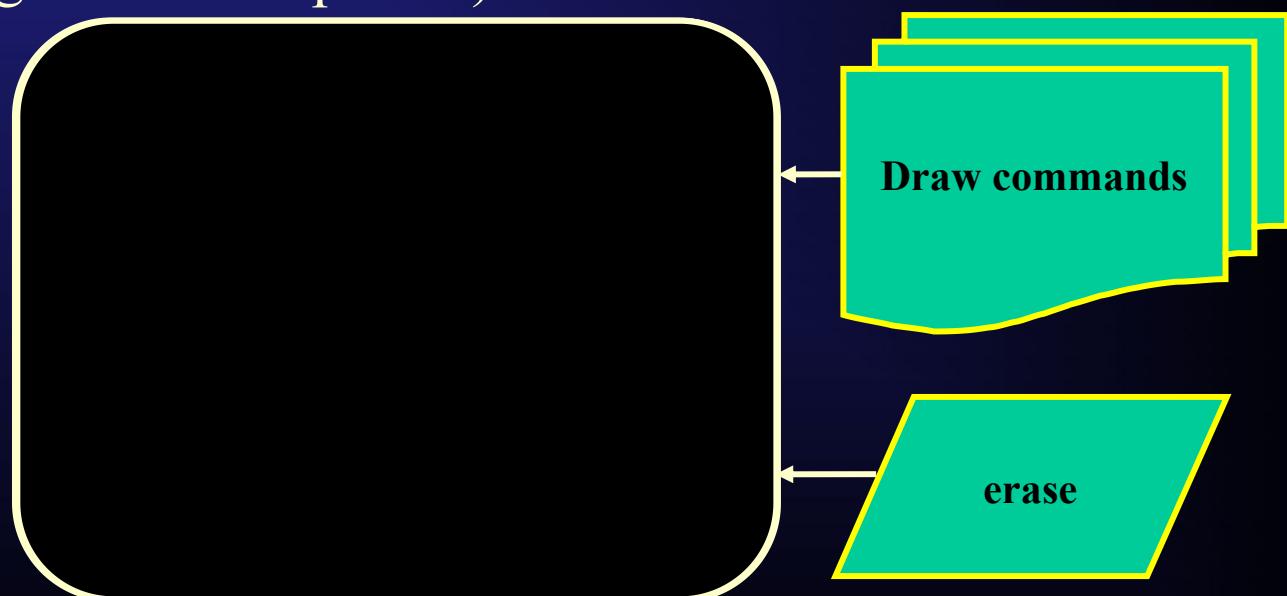
- History
- Devices
- Raster graphics fundamentals
- Color systems
- Raster hardware (conceptual) architectures
- GPU high-level architecture

Historical Progression

- Pen plotters (60's)
- Microfilm writers
- Storage tubes (early 70's)
- Vector displays (late 70's)
- Emergence of graphics workstations (80's); Silicon Graphics (SGI) dominance
- PC transition, graphics boards, game consoles (90's)
- Convergence of digital graphics, (HD) television, and the Internet (00's)
- Mobile (wireless) platforms

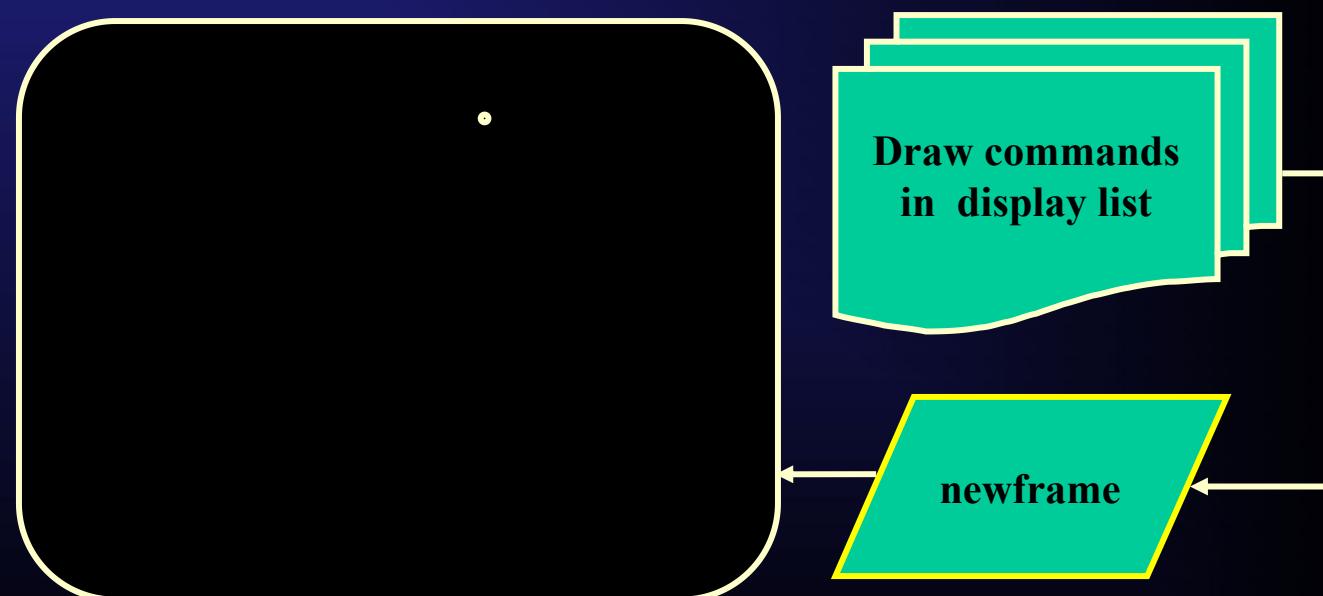
Direct Program Writes using a Storage Display

- ‘70’s graphics dominated by (inexpensive) storage tubes.
- Program draws graphics directly in CRT (continuous lines, no pixels).
- Global screen (flash) erase.
- New technology life in 2000+ as “Electronic Ink” eBooks (raster images and low power).



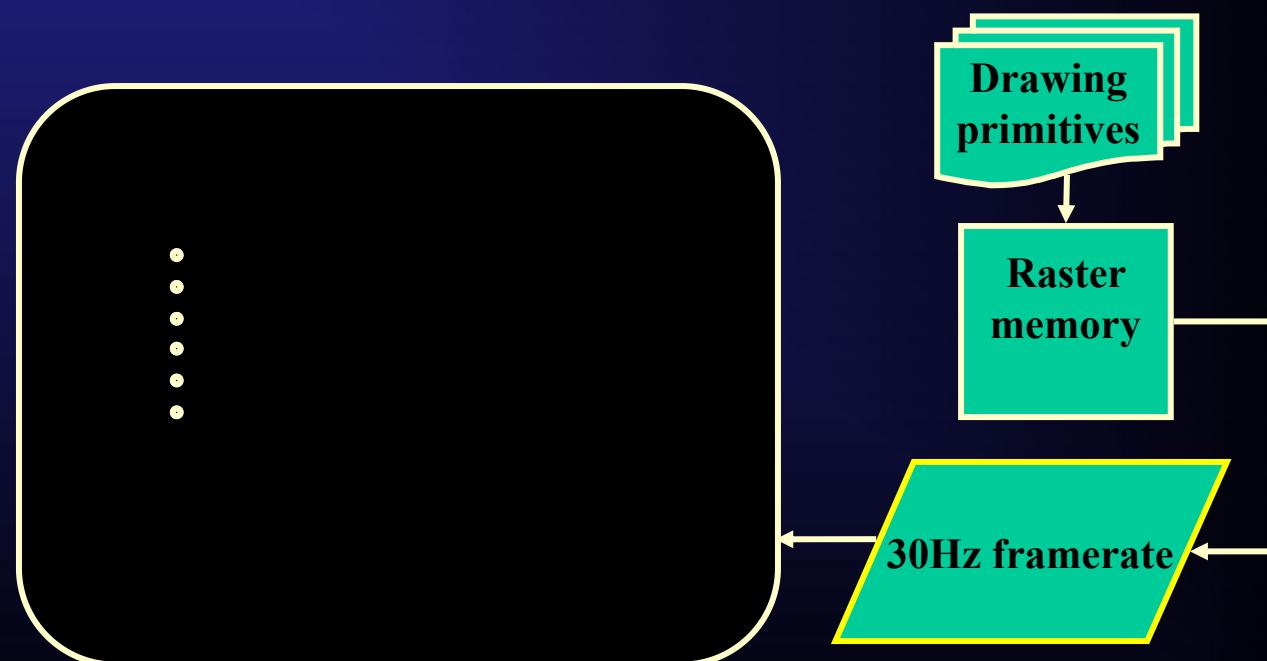
Refreshed Vector Graphics

- Random positioning refreshed **vector** CRT developed from established oscilloscope technology (early 70's).
- Store graphics elements to be drawn into a **display list**.
- Beam trace on CRT persists only briefly so display list must be continuously redrawn (“newframe” action).
- Allows some animation.



Writing Graphics into a Raster Memory

- Raster graphics CRT became feasible as memory cost dropped (mid 70's).
- Program creates drawing elements (saved in a display list) that are rasterized into the discrete rectangular array memory.
- Memory is scanned to display at a fixed rate in a fixed raster pattern.

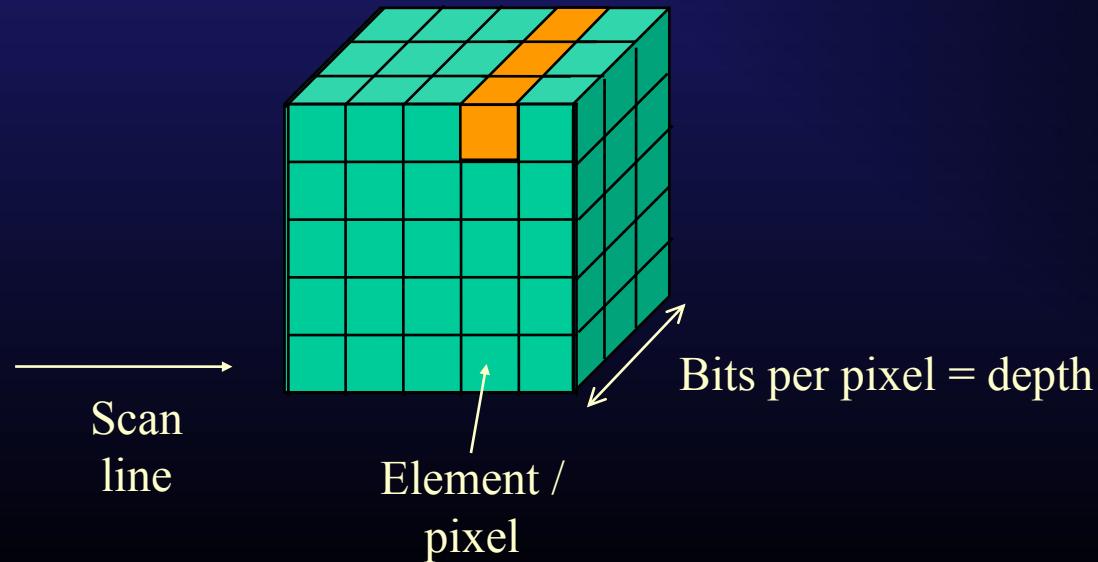


Raster Frame Buffer Definitions

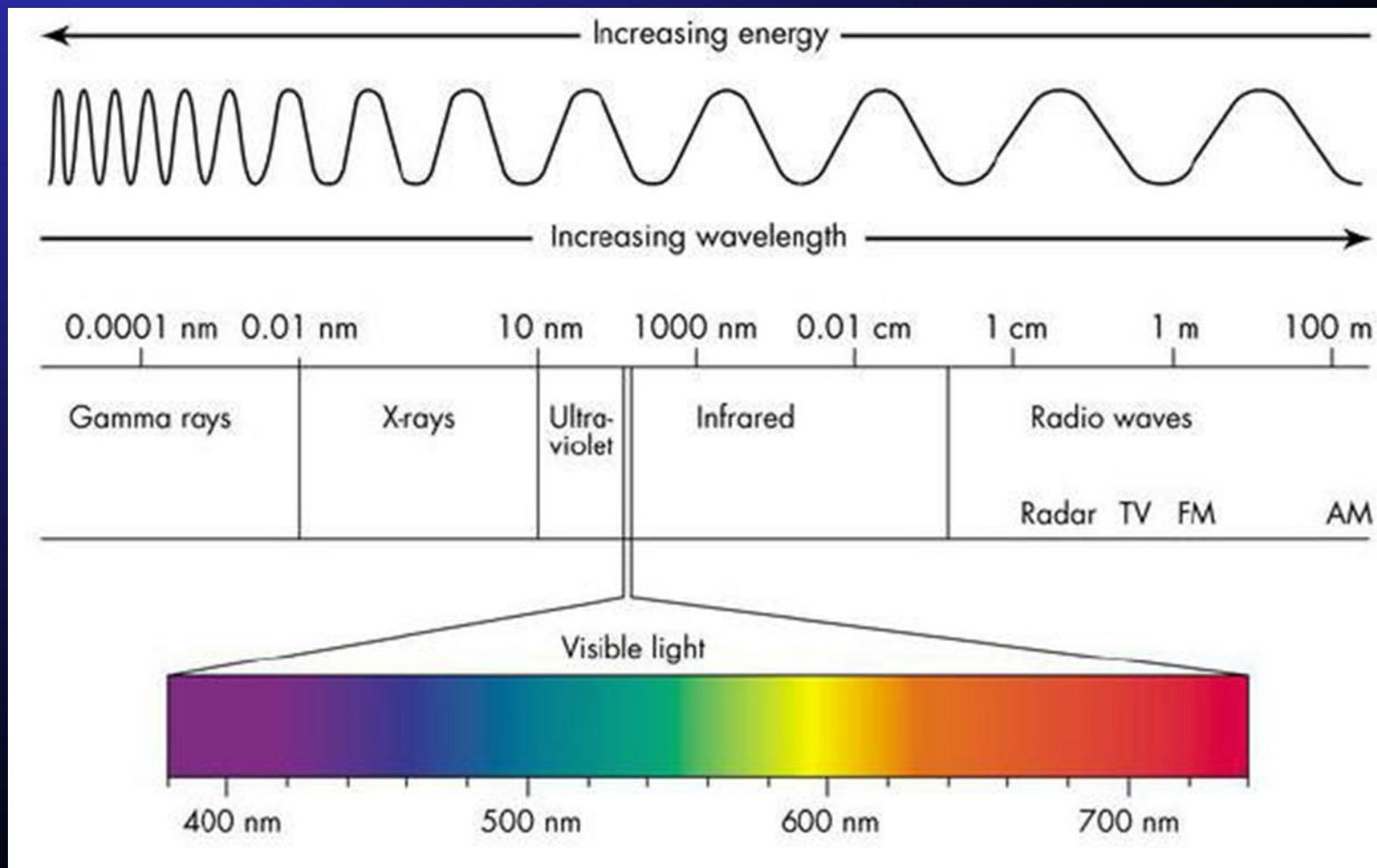
- A **frame buffer** is a rectangular array of display elements called **pixels**. A row of pixels is called a **scan line**.
- Raster **graphics** images are displayed by drawing the pixels sequentially in a fixed order.
- **Colors** are created by separately exciting red, green, and blue color generators within pixels on the imaging device.

Frame Buffer Graphics Memory

- Spatial **resolution** = elements \times scan lines
 - e.g. 5×5 (below)
- Color precision or depth = 2^b (bits per pixel)
- In this [trivially-sized] example each pixel can have any of $2^3 = 8$ possible values.

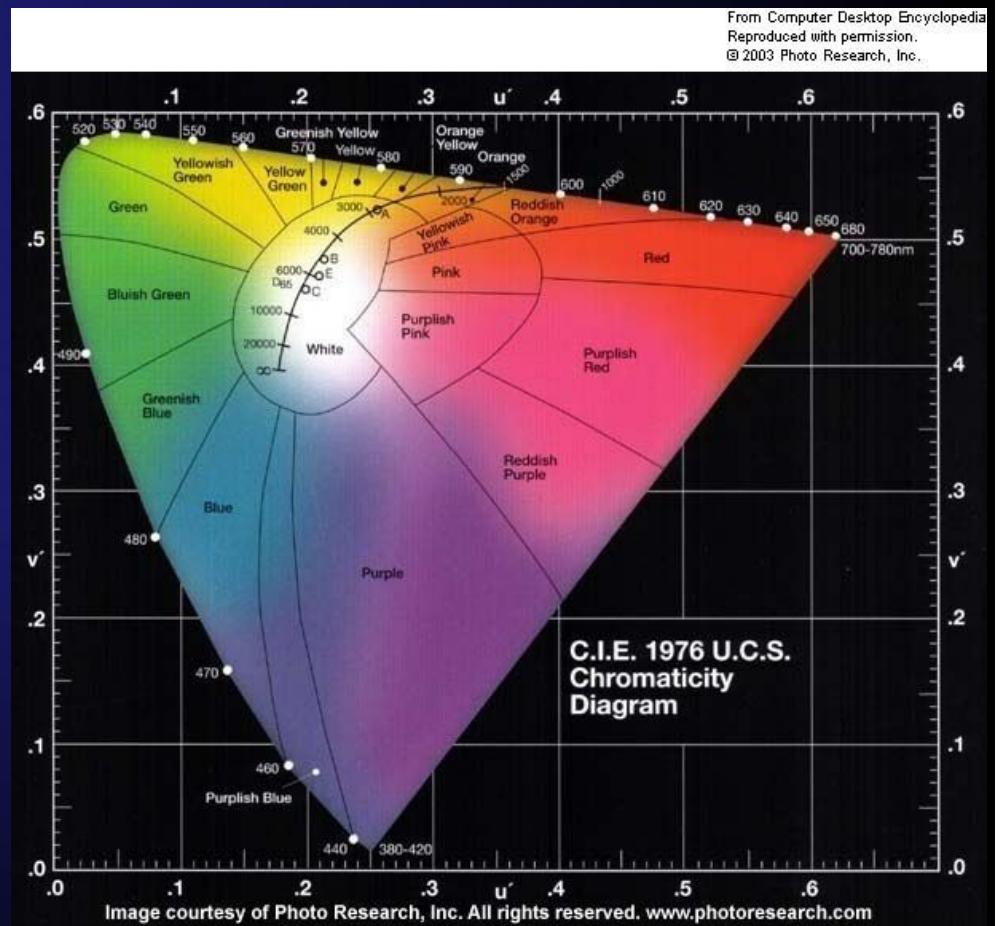


Real Light has Physical Properties: Wavelength (or Frequency), Energy



Color Perception

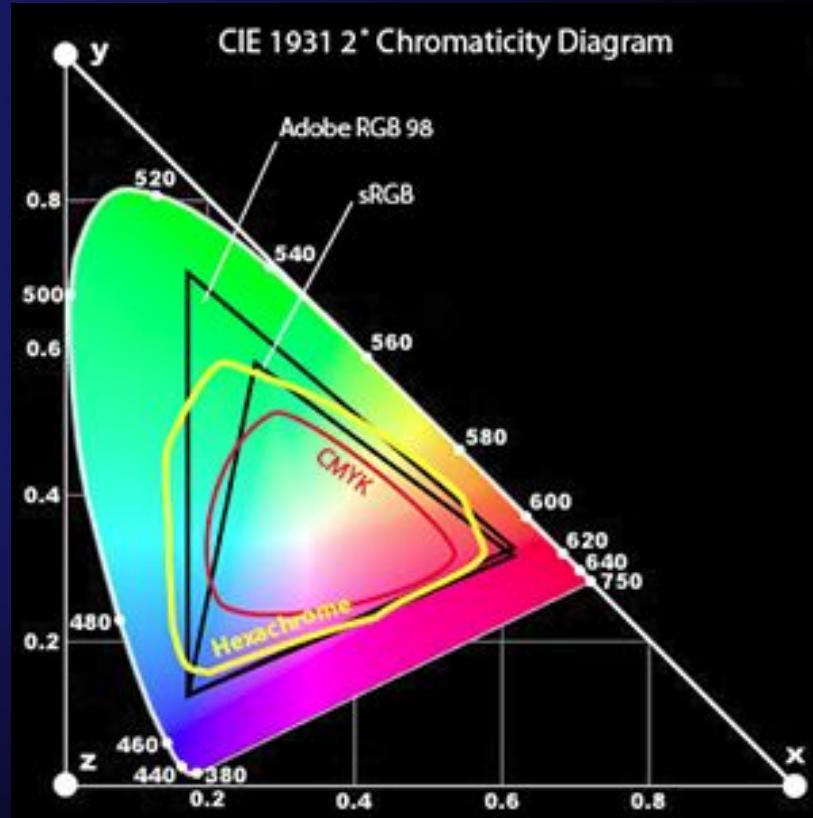
- Minimum number of levels for 2% just noticeable difference (JND) in intensity is 100, or about 7 bits.
- Thus, use at least 8 bits per color (R, G, B) (≥ 12 for photographic quality).
- 24 bit color gives around 4 million color choices.
- Often scale R, G, B each to [0..1] range.



- CIE = Commission Internationale de l'Eclairage = International Commission on Illumination.
- Graphs all humanly perceivable colors.
- L^* = luminance; u^* is the red-green axis; v^* is the blue-yellow axis.

CIE Color Standard

- Any physical device (display or printer) reproduces only a subset of the complete color space, i.e., its *gamut* is a subset of the CIE space.
- Linear combinations of **Red**, **Green**, and **Blue** are used to create colors: we usually work with digital quantities of these.
- But we could use sets of wavelengths to approximate continuous spectra. These are used in advanced rendering engines.



RGB Light Approximation

- R, G and B each in range [0..1].
- White light is (1, 1, 1); no light (black) is (0, 0, 0).
- Visual perception of surface color is component-wise product:
 - Let the emitted light source color be (R_{light} , G_{light} , B_{light})
 - Let the intrinsic surface color be (R_{surface} , G_{surface} , B_{surface}).
 - Then the reflected (=perceived) color is just the component-wise product “ \odot ”:

$$\begin{aligned} & (R_{\text{light}}, G_{\text{light}}, B_{\text{light}}) \odot (R_{\text{surface}}, G_{\text{surface}}, B_{\text{surface}}) \\ & = (R_{\text{light}} \times R_{\text{surface}}, G_{\text{light}} \times G_{\text{surface}}, B_{\text{light}} \times B_{\text{surface}}) \end{aligned}$$

Try it:

- White light on red surface:

$$(1, 1, 1) \odot (1, 0, 0) = (1, 0, 0) = \text{red!}$$

- Red light on white surface:

$$(1, 0, 0) \odot (1, 1, 1) = (1, 0, 0) = \text{red!}$$

- Red light on green surface:

$$(1, 0, 0) \odot (0, 1, 0) = (0, 0, 0) = \text{black!}$$

- Red light on yellow surface:

$$(1, 0, 0) \odot (1, 1, 0) = (1, 0, 0) = \text{red!}$$

- Yellow light on blue surface:

$$(1, 1, 0) \odot (0, 0, 1) = (0, 0, 0) = \text{black!}$$

(Note: If you try this at home, it will *almost* work. Why not?

⇒ Because of the discretization of continuous wavelengths into non-physically correct R, G, B: the spectra are not so “*smooth*”.)

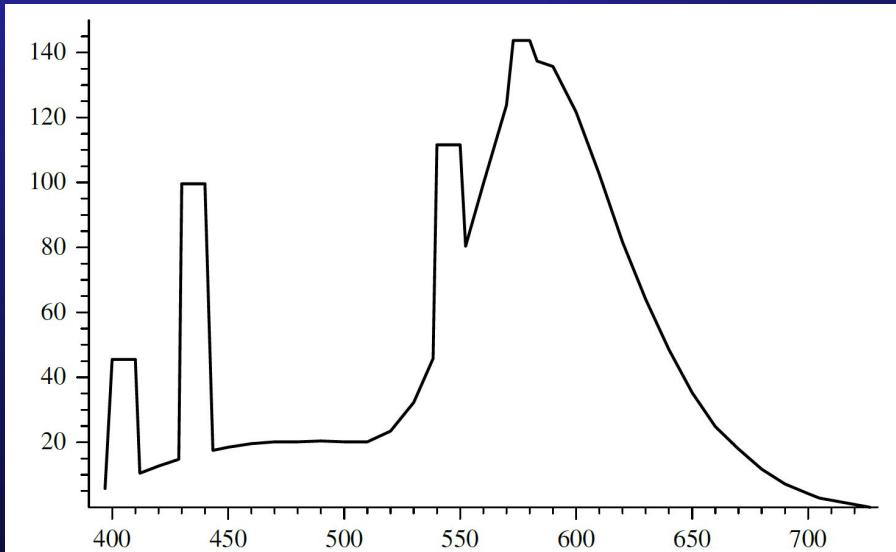
Digital Color Systems – Depend on Application

- RGB (Red, Green, Blue) : additive color primaries; easy to build in emissive hardware. [Add color to no light (black).]
- HLS (Hue, Lightness, Saturation) and HSV (Hue, Saturation, Value) : conform more to human naming conventions and easier color selection.
- CMY(K) (Cyan, Magenta, Yellow, (Black)) : subtractive color primaries (complements) used in ink printing.
 - Subtract \equiv absorb and therefore don't reflect complementary color from white (paper):
 - $C=G+B$, $M=R+B$, $Y=R+G$, $K=R+G+B$.
 - So, e.g., R is reflected off white paper by M+Y: M absorbs G (killing G from Y) and Y absorbs B (killing B from M), leaving just R.
 - K saves ink; gives more uniform black.

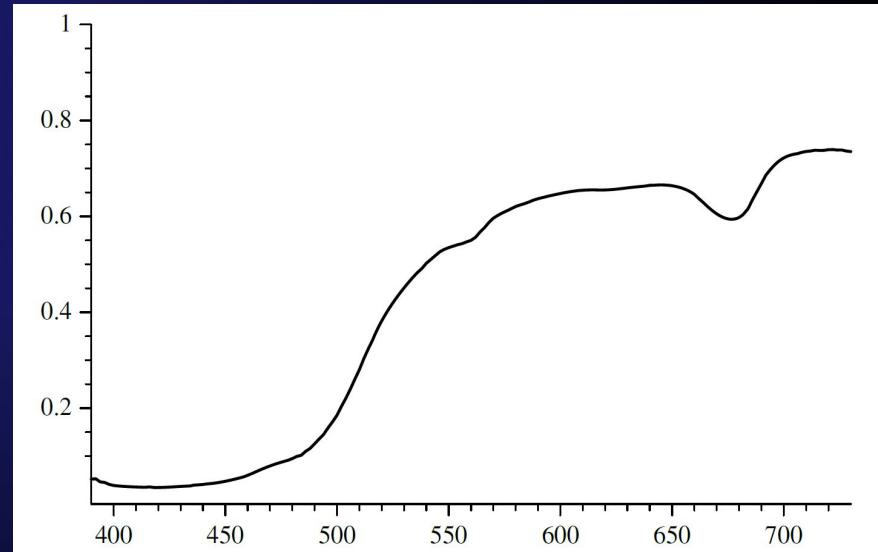


Beyond RGB to Physically-Based Illumination: Spectral Power Distribution (SPD)

- How much “light” is at each wavelength?



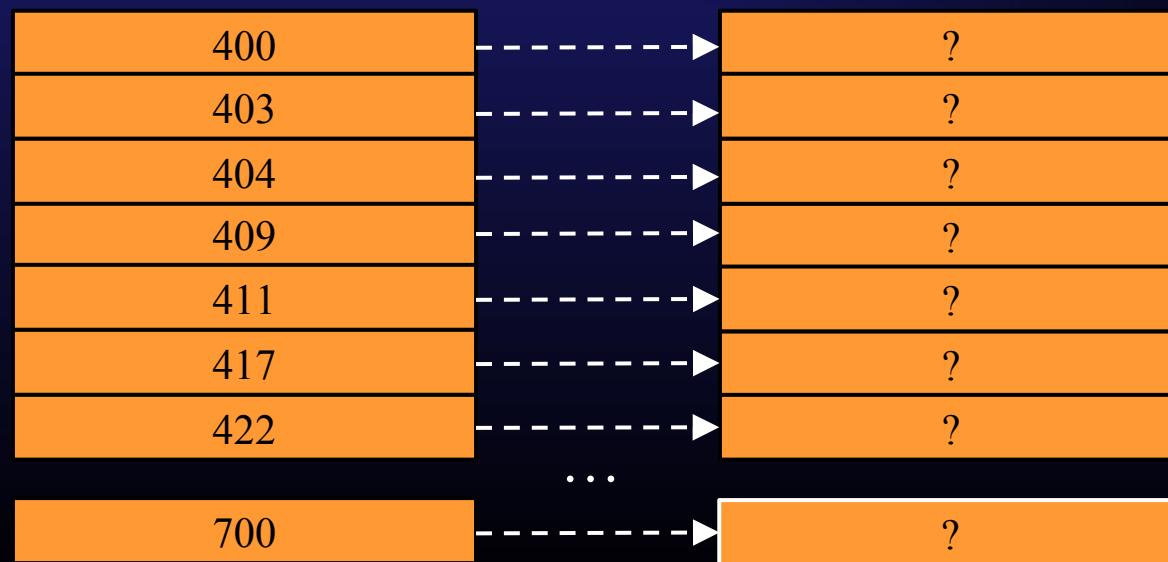
Fluorescent bulb
(emissive)
(approximations)



Lemon
(reflective)

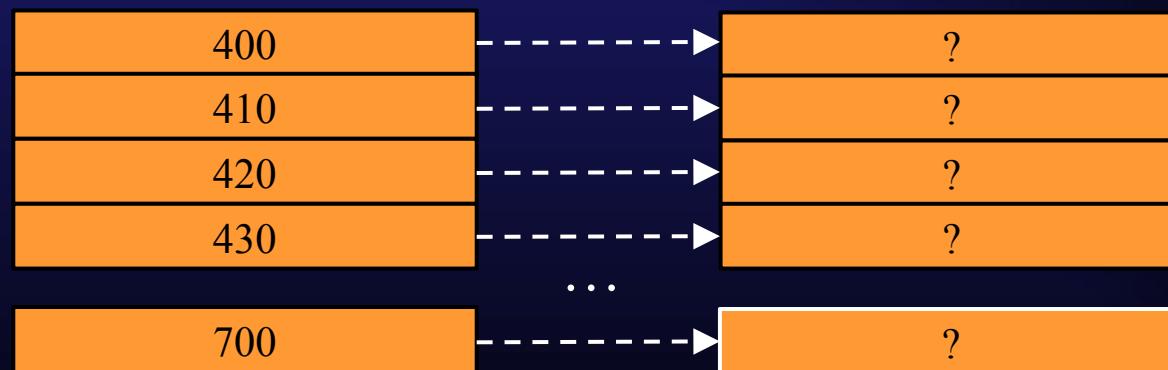
1D Sampling and Reconstruction

- We can take an SPD that is *arbitrarily sampled* and reconstruct it as an SPD with uniformly spaced samples.
- Visible spectrum is about 400 nm to 700 nm.
- Raw spectral data may look like this: non-uniform sample spacing, thus arbitrary array size (hundreds or thousands).



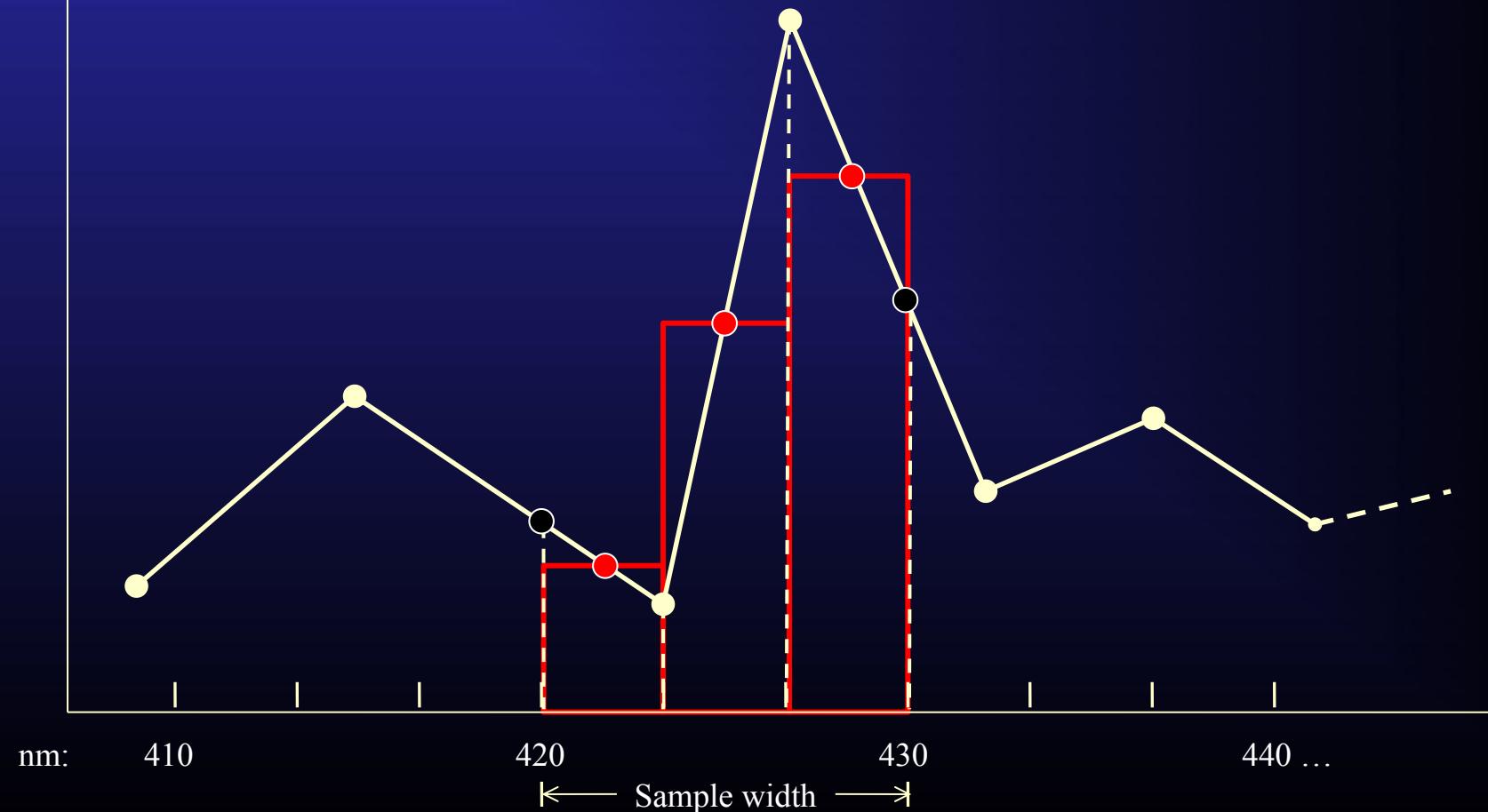
1D Sampling and Reconstruction

- Want an SPD with *uniformly spaced* samples.
- Rule of thumb: need 30 samples across visible spectrum, so $(700 \text{ nm} - 400 \text{ nm})/(30 \text{ samples}) = 10 \text{ nm/sample}$.



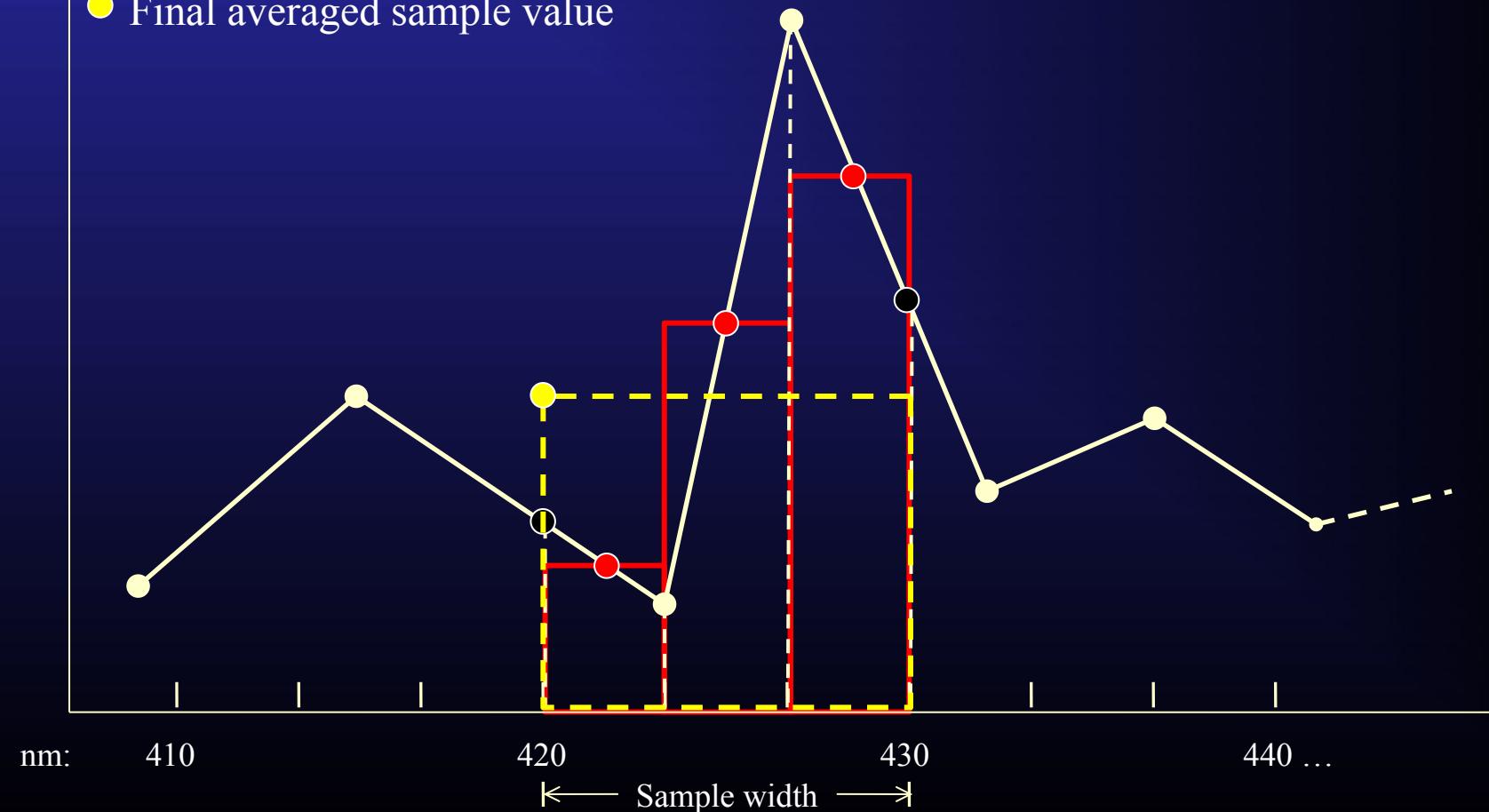
General Idea: Uniformly Re-sample over each Interval

- Original sample
- New uniform sample
- Lerp'ed midpoint



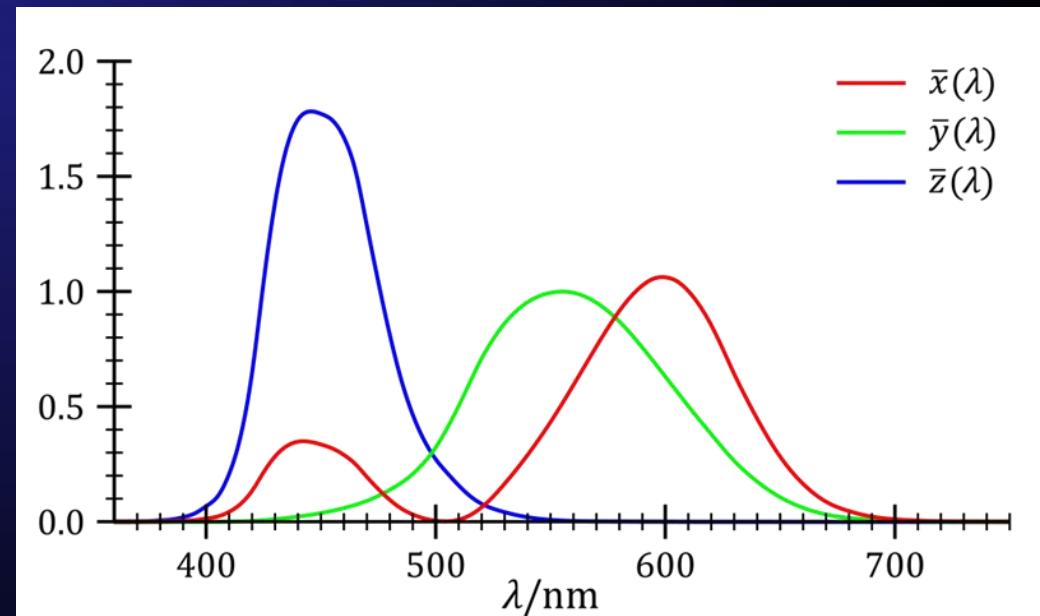
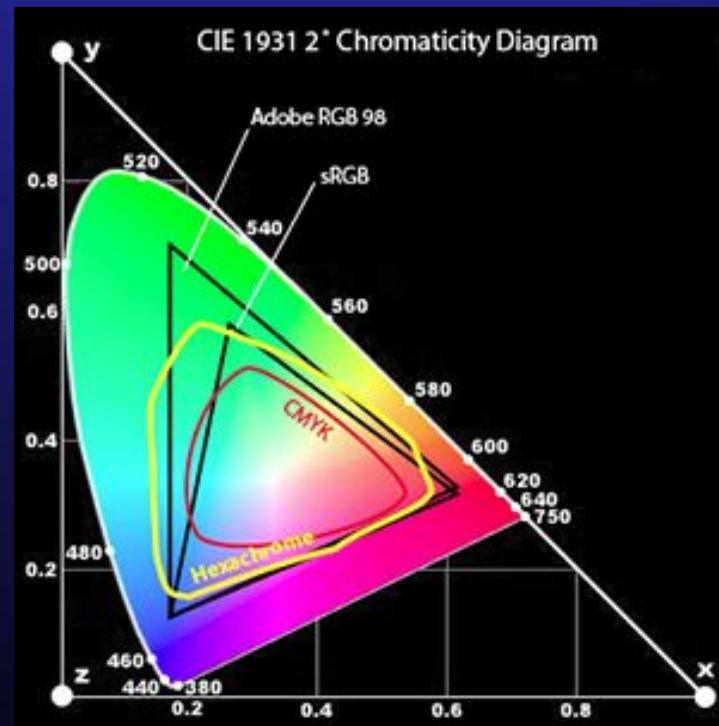
Average all Sub-sampled Values for each Sample Interval

- Original sample
- New uniform sample
- Lerp'ed midpoint
- Final averaged sample value



But our Displays use RGB!

- Convert SPD to CIE *tristimulus* perceptually-based color system called XYZ, then to RGB.
- Note XYZ axes in the diagram.



- (The y value is related to *luminance*, or the perceived brightness of a SPD to a human observer.)

Convolve SPD to get XYZ

- Continuous

$$x_\lambda = \frac{1}{\int Y(\lambda) d\lambda} \int_\lambda S(\lambda) X(\lambda) d\lambda$$

$$y_\lambda = \frac{1}{\int Y(\lambda) d\lambda} \int_\lambda S(\lambda) Y(\lambda) d\lambda$$

$$z_\lambda = \frac{1}{\int Y(\lambda) d\lambda} \int_\lambda S(\lambda) Z(\lambda) d\lambda$$

- Discrete

$$x_\lambda \approx \frac{1}{\int Y(\lambda) d\lambda} \sum_i X_i c_i$$

$$y_\lambda \approx \frac{1}{\int Y(\lambda) d\lambda} \sum_i Y_i c_i$$

$$z_\lambda \approx \frac{1}{\int Y(\lambda) d\lambda} \sum_i Z_i c_i$$

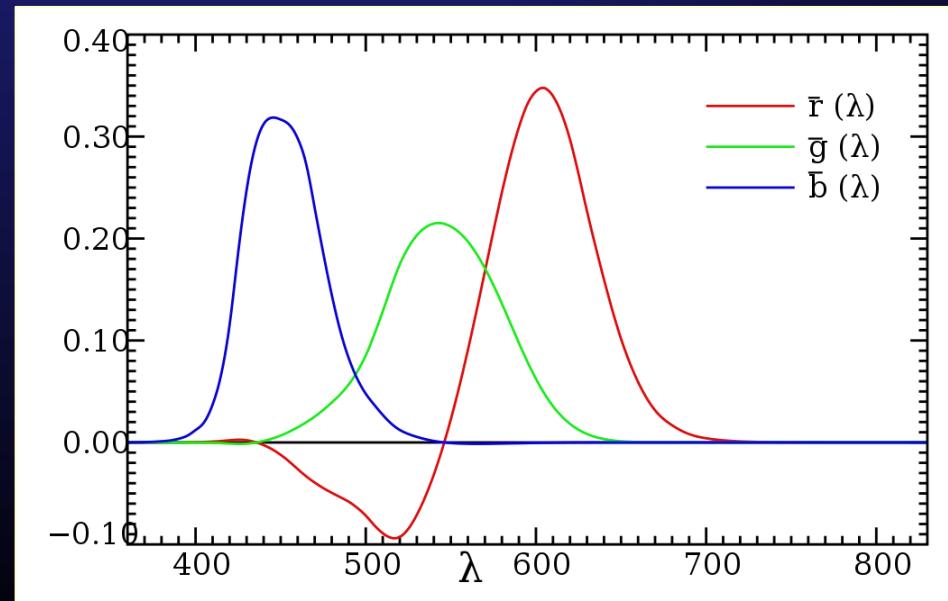
$$\int Y(\lambda) d\lambda \approx \sum_i Y_i$$

c_i values are from the uniformly sampled SPD.

X_i, Y_i, Z_i are uniformly sampled over the CIE definition, using the same averaging algorithm.

Can Directly Convert SPD to RGB

Exactly the same as the previous slide, but convolve using the RGB color matching curves (below) rather than the XYZ color matching curves.



But if we already have XYZ, then can get RGB via Matrix Multiplication

- Each matrix column is the XYZ basis in RGB space, where the spectral response curves of the display are $R(\lambda)$, $B(\lambda)$, and $G(\lambda)$

$$\begin{aligned} r &= \int R(\lambda)S(\lambda)d\lambda = \int R(\lambda)(x_\lambda X(\lambda) + y_\lambda Y(\lambda) + z_\lambda Z(\lambda))d\lambda \\ &= x_\lambda \int R(\lambda)X(\lambda)d\lambda + y_\lambda \int R(\lambda)Y(\lambda)d\lambda + z_\lambda \int R(\lambda)Z(\lambda)d\lambda \end{aligned}$$

$$g = \dots$$

$$b = \dots$$

$$\begin{bmatrix} r \\ g \\ b \end{bmatrix} = \begin{bmatrix} \int R(\lambda)X(\lambda)d\lambda & \int R(\lambda)Y(\lambda)d\lambda & \int R(\lambda)Z(\lambda)d\lambda \\ \int G(\lambda)X(\lambda)d\lambda & \int G(\lambda)Y(\lambda)d\lambda & \int G(\lambda)Z(\lambda)d\lambda \\ \int B(\lambda)X(\lambda)d\lambda & \int B(\lambda)Y(\lambda)d\lambda & \int B(\lambda)Z(\lambda)d\lambda \end{bmatrix} \begin{bmatrix} x_\lambda \\ y_\lambda \\ z_\lambda \end{bmatrix}$$

- So these values can be pre-computed, but they do depend on the display device characteristics!

E.g., for HDTV:

- The conversion matrix is:

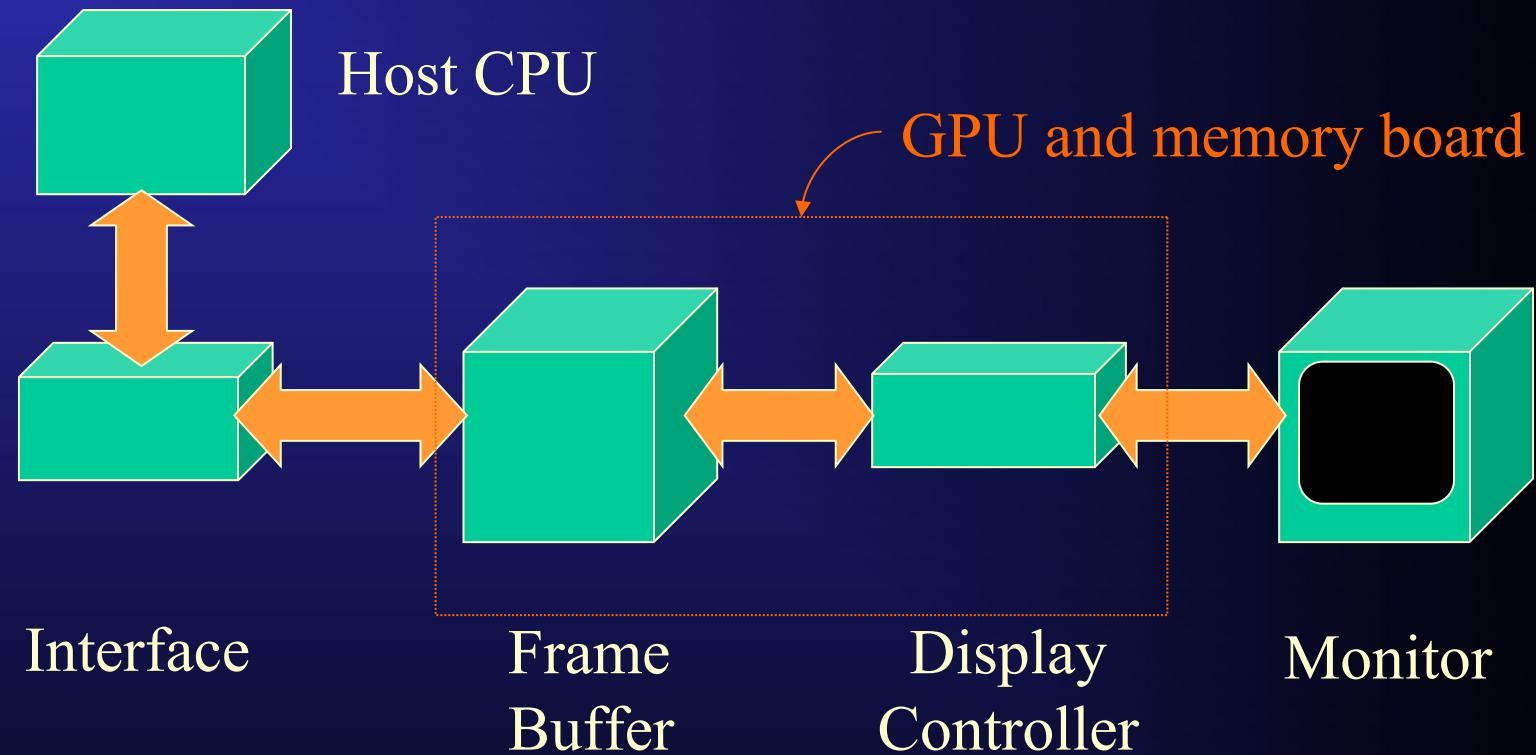
$$\begin{bmatrix} 3.240479 & -1.537150 & -0.498535 \\ -0.969256 & 1.875991 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{bmatrix}$$

- Beware: This matrix is different for LCD or for LED displays!
- And it doesn't take into account any local display settings.

Frame Buffer and GPU

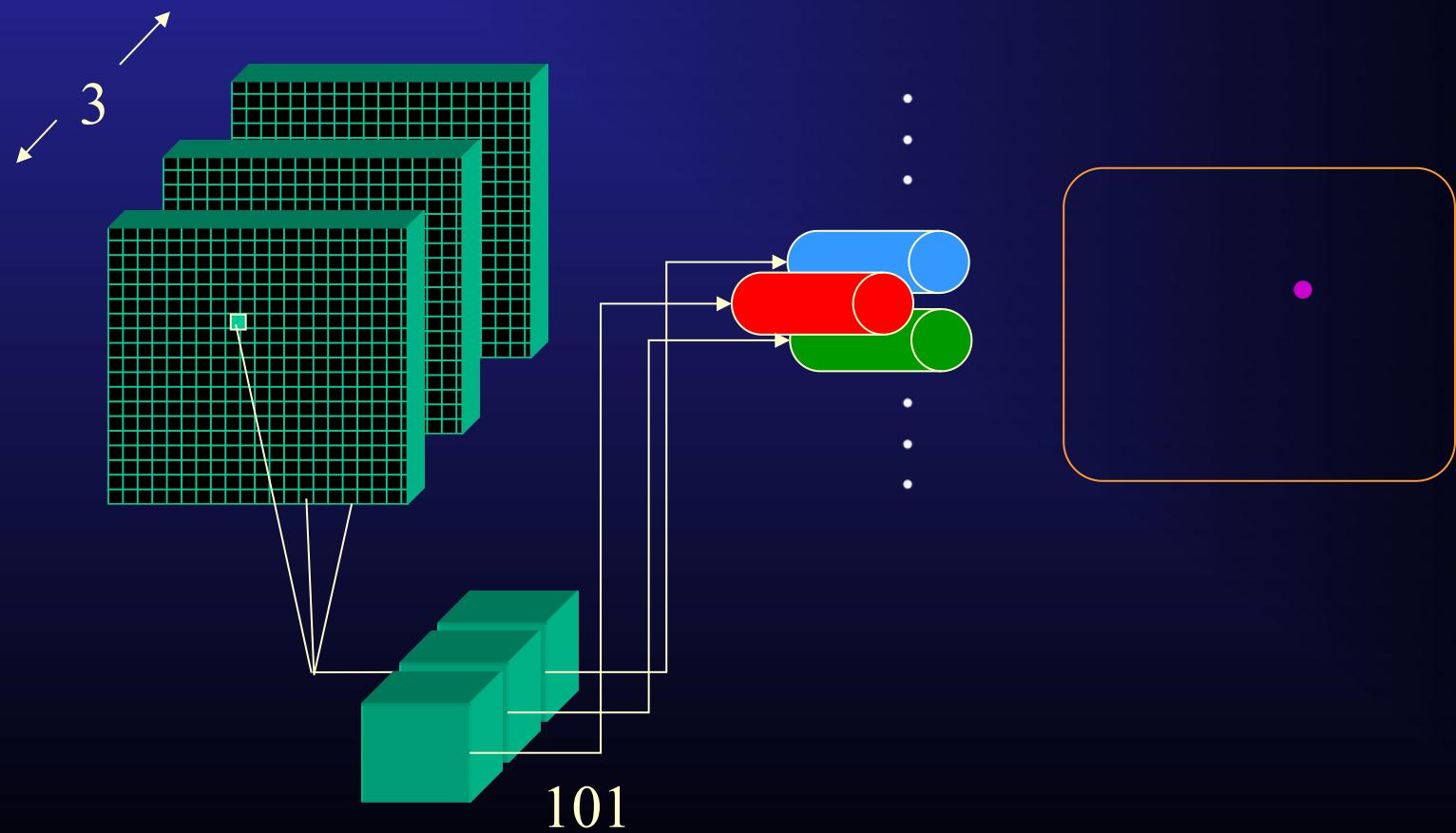
- Host computer transmits commands and data across an interface.
- The raster image described by this data is stored in a “frame buffer” memory.
- The frame buffer is built from one or more “bit planes” –(conceptually) two-dimensional arrays of bits. This memory is peripheral to the host on a separate board. (Of course, the actual memory is linear, so the 2D values are mapped to the linear addresses.)
- Increasing compute power on the board has led to rapid evolution of the **Graphics Processing Unit (GPU)**.

Simplified Frame Buffer Architecture



Modern graphics board architectures are best understood after exploring graphics image synthesis techniques.

8 Color Frame Buffer (3 Bit Planes)



3 Bit Planes = 8 Colors

0 0 0 == BLACK



1 0 0 == RED



0 1 0 == GREEN



0 0 1 == BLUE



1 1 0 == YELLOW



0 1 1 == CYAN



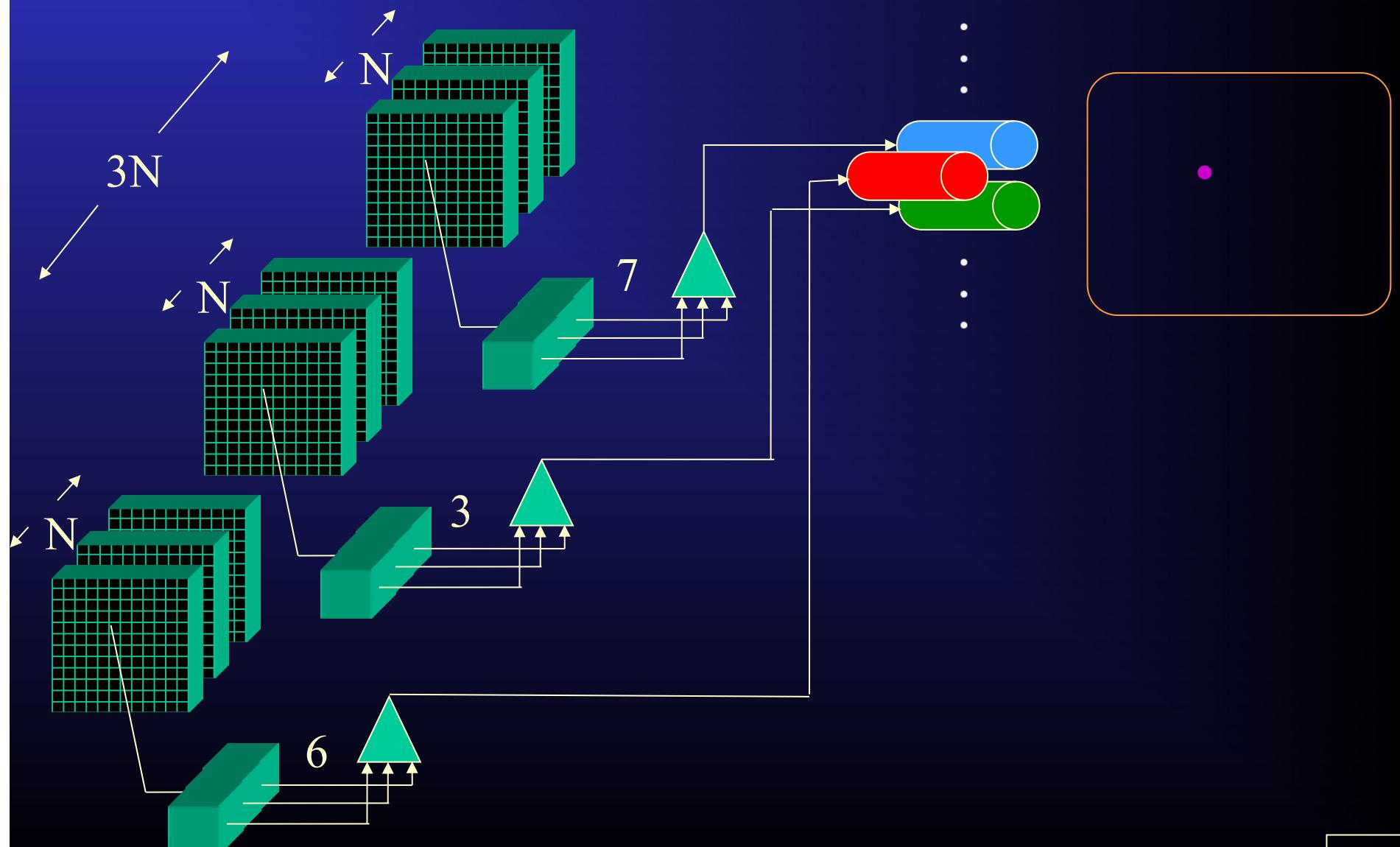
1 0 1 == MAGENTA



1 1 1 == WHITE

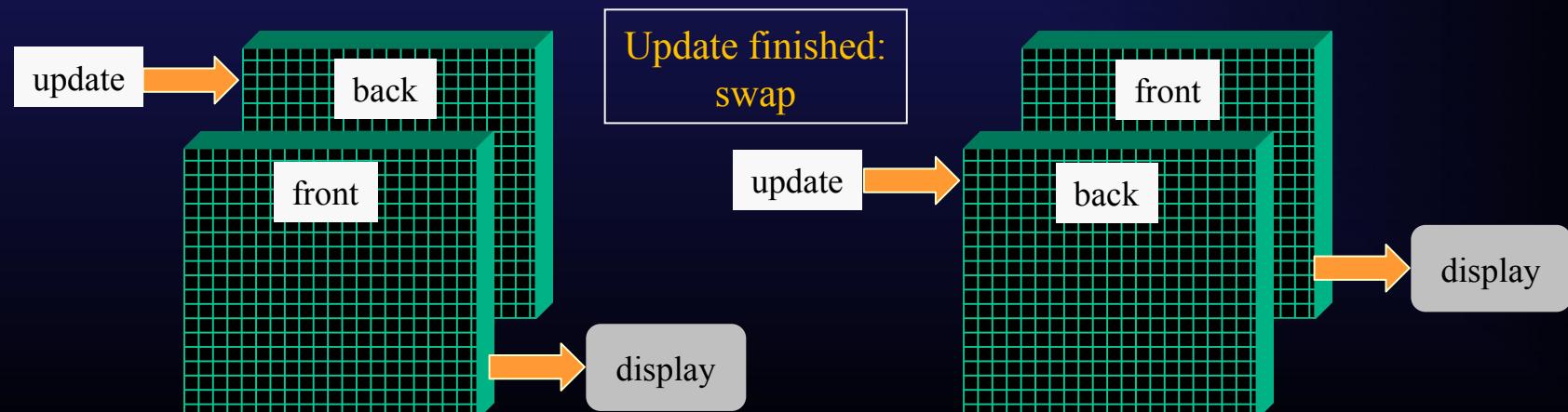


Color Frame Buffer with $3N$ Bit Planes

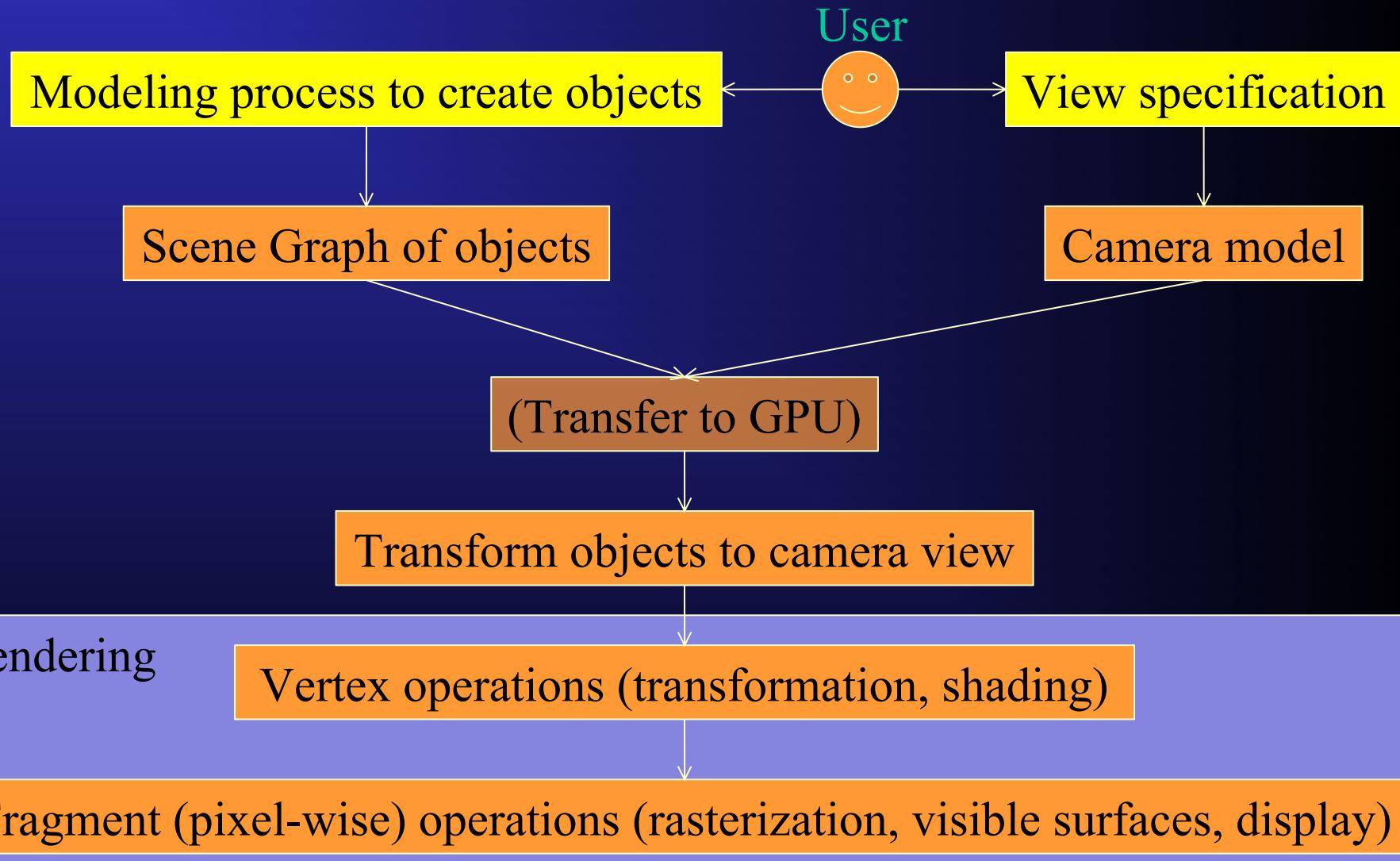


Double Buffering

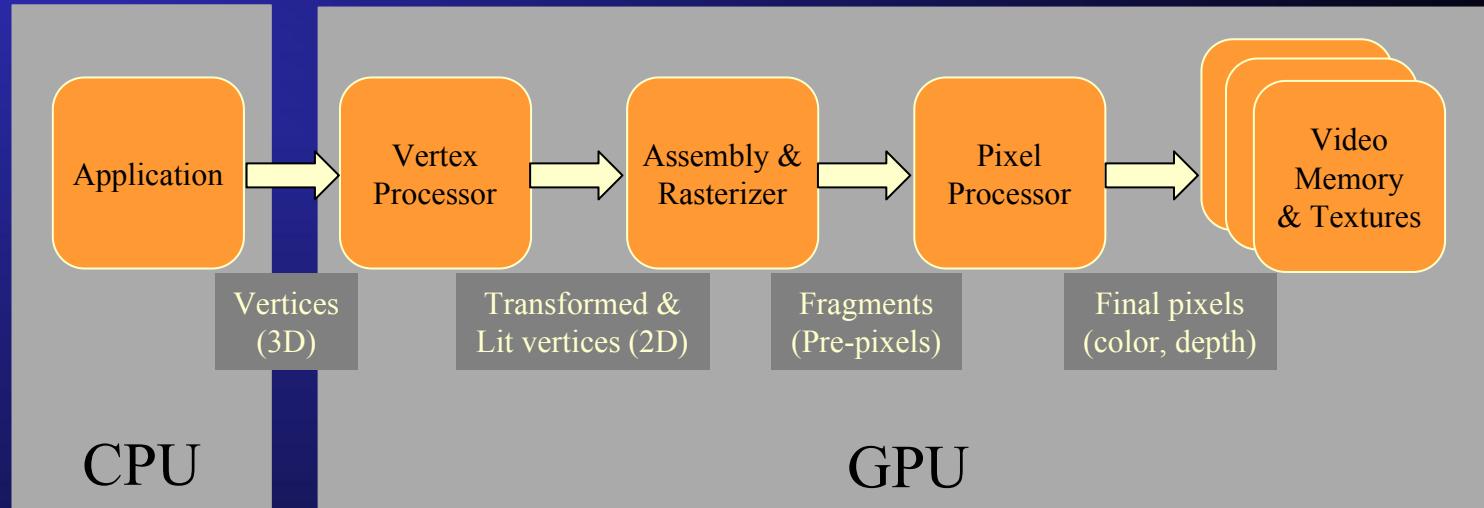
- Rasterize drawing elements into one frame buffer (“**back**”) while displaying another (“**front**”).
- Avoids having user “see” the image creation process (drawing order).
- When image is done, “swap” front and **back** buffers.
- Start drawing next image in new back buffer.



Overview of Graphics Processing



Overview of Modern Graphics Pipeline



- The modern GPU pipeline processes graphical elements from vertex data in 3D through spatial transformation, lighting (shading) and rasterization of pixel-wise (“fragment”) data.
- The final stage combines fragments into final colors and depths for image and depth memories.
- We’ll see all these algorithmic steps in detail.

Raster Graphics “Primitives”

- Points 
- Line segments 
- Rectangles 
- Circles 
or
- Conics 
- Triangles 
- Polygons 
- Characters or Special symbols  
- Bit maps, sprites, or patterns 

Rasterization

Rasterization is the process of transforming geometric shapes (given by vertices or primitives) into discrete raster fragments.



Rasterization Methods

- Depend upon:
 - Starting geometry (2D)
 - Data structure of the geometry
 - Smoothness desired (i.e., aliasing)
 - Assume integer grid of pixels
- Simple primitives: points, lines
- Polygons and triangles
- Characters, symbols, patterns, sprites

Points

- Given as floats or integer coordinates: (x,y)
- Assume procedure Write_pixel (round (x), round (y)).
- Do not display (clip) if either coordinate is outside raster boundary.
- (For real coordinate points and accurate display, should distribute point's intensity/color over neighboring pixels: see anti-aliasing section.)

Line Segments

- Given coordinates of line segment: from (x_1, y_1) to (x_2, y_2) .
- Line segment is generated as a series of pixels in an incremental algorithm: i.e., each point is generated from the previous point in a simple fashion.

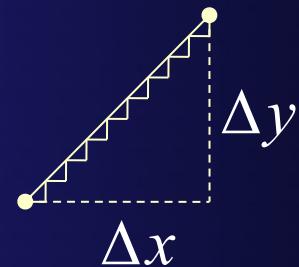


- Demo: 
- Sometimes called “DDA” --- Digital Differential Analyzer

Simple DDA (Digital Differential Analyzer)

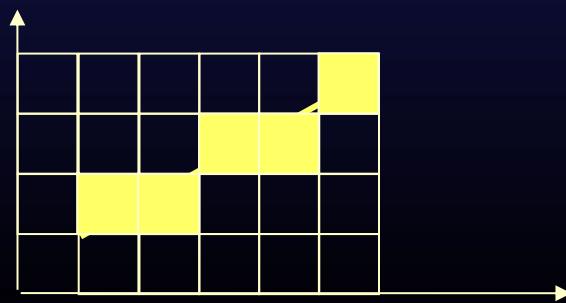
- Slope of line to be drawn:

$$\frac{dy}{dx} = \frac{\Delta y}{\Delta x}$$



- Method:
 1. Go to starting endpoint
 2. Increment x and y values by constants proportional to Δx and Δy

such that one of them is 1. Round to closest raster position.



Simple DDA Code (1)

```
void SIMPLE_DDA (float X1, float Y1, /* first endpoint */
                  float X2, float Y2) /* second endpoint */
{
    float X, Y, Xinc, Yinc, LENGTH; int i;
    Xinc := X2 - X1;
    Yinc := Y2 - Y1; /* if both 0, then just draw a point */
    if ((Xinc == 0.0) && (Yinc == 0.0)) draw_point (X1, Y1);
    else /* this is a line segment */
    {
        /* LENGTH is the greater of Xinc, Yinc */
        if (abs(Yinc) > abs(Xinc))
            {LENGTH := abs(Yinc);
             Xinc := Xinc / LENGTH; /* Note float division */
             Yinc := 1.0} /* Step along Y by pixels */
```

Simple DDA Code (2)

```
else
{LENGTH := abs(Xinc);
 Yinc := Yinc / LENGTH; /* Note float division */
 Xinc := 1.0} /* Step along X by pixels */
X := X1; Y := Y1; /* Start drawing from X1, Y1 */
for ( i := 0; i <= round(LENGTH); i++) /* Do this at */
    {draw_point (round(X), round(Y)); /* least once */
     X := X + Xinc;
     Y := Y + Yinc}
} /* end else `this is a line segment' */

return;
} /* end SIMPLE_DDA */
```

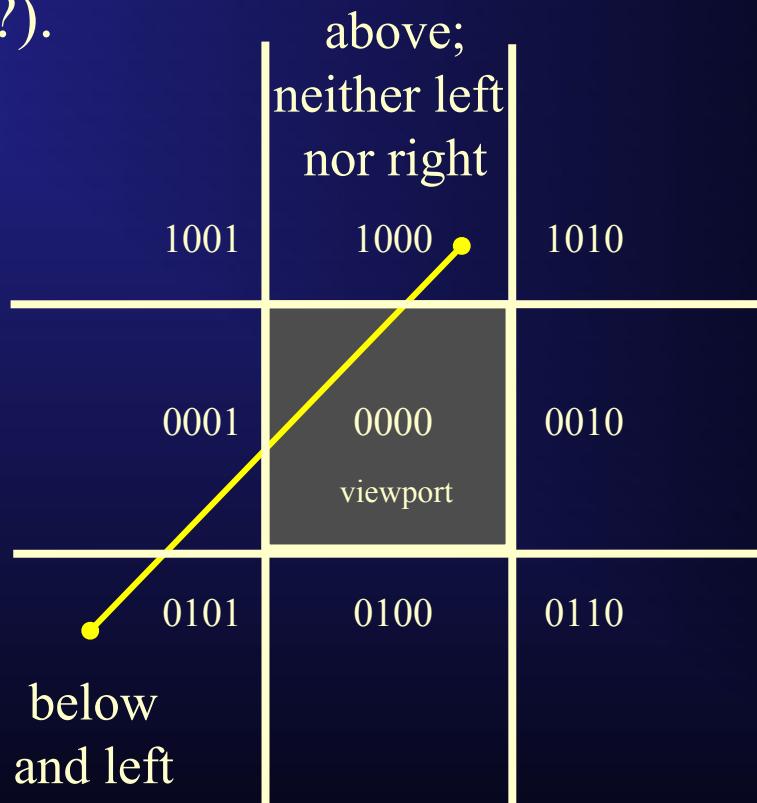
Line Clipping

Remove portion of line outside viewport or screen boundaries before rasterization (can't just truncate endpoint coordinates to boundaries: why?).

Endpoint “Code”

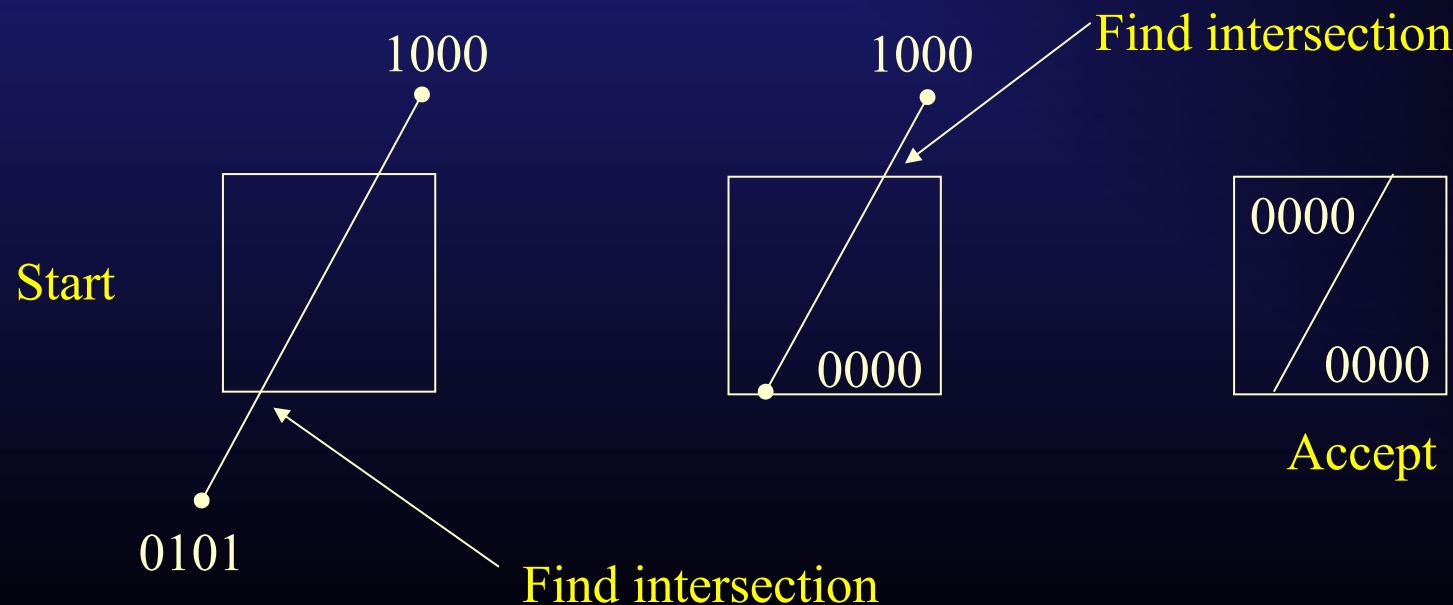
Key:

| | | | |
|---|---|---|---|
| a | b | r | l |
| b | e | i | e |
| o | l | g | f |
| v | o | h | t |
| e | w | t | |



Clipping in Action

1. Line segment is entirely in viewport if both endpoints = 0000
2. Line segment is entirely outside of viewport if logical AND of codes $\neq 0000$
3. Otherwise divide line at one viewport boundary, throw away outside part. Then test again (up to 4 times)

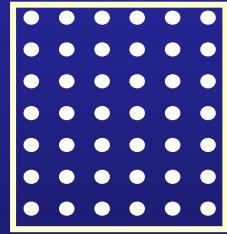


Recursive Clipping without Float Divisions

Recursively divide line segment in half until midpoint falls on boundary. (Avoids real division needed to compute boundary intersection.)



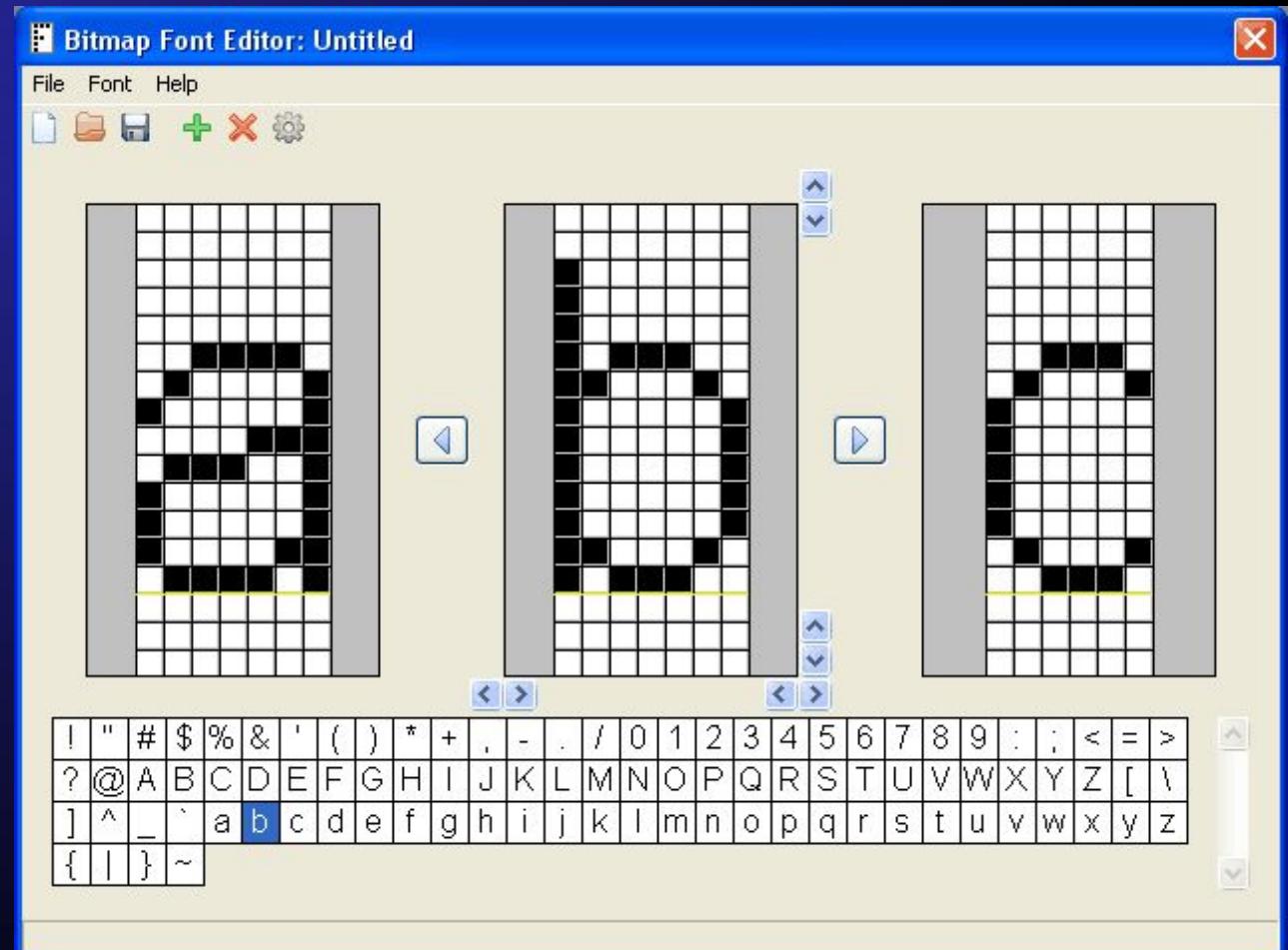
Axis Aligned Rectangles



Given opposite corners,
filling a rectangle is a
simple nested loop.

Characters, Fonts and Symbols (1)

- 2D bitmaps.
- Copy or map into frame buffer: **bitblt** – *pixel block move* machine instruction.
- Easy to design.
- Don't scale or rotate well.

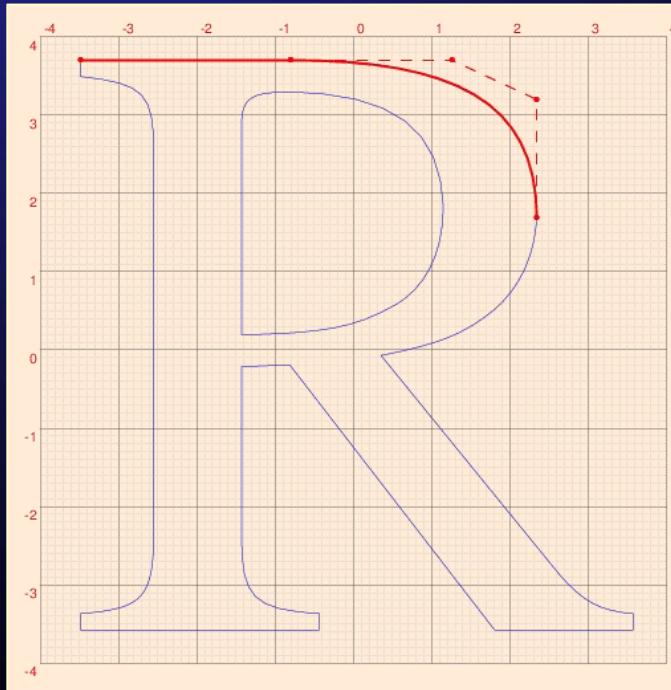
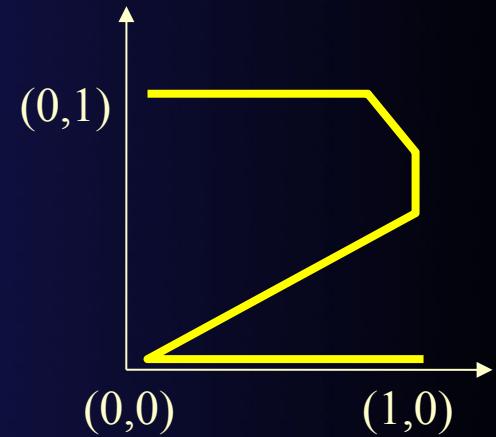


http://mobilefonts.sourceforge.net/images/bfe_screenshot.png

Characters, Fonts and Symbols (2)

- Stroke tables \Rightarrow
- Generally scale and rotate nicely
- For nicer results, store curve control vertices (e.g., Bezier curves in Postscript) \Rightarrow
- If closed polygons, can fill or texture as well \Rightarrow

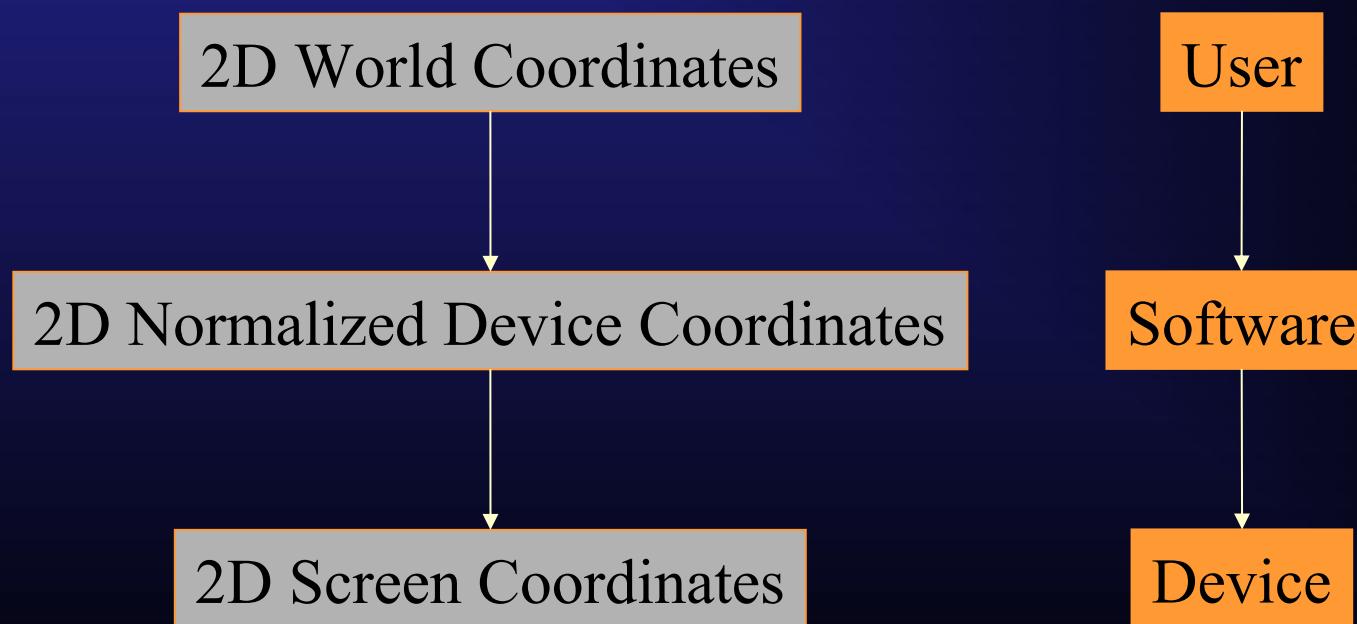
| | | |
|------|-----|-----|
| Move | 0.0 | 1.0 |
| Draw | 0.8 | 1.0 |
| Draw | 1.0 | 0.8 |
| Draw | 1.0 | 0.5 |
| Draw | 0.0 | 0.0 |
| Draw | 1.0 | 0.0 |



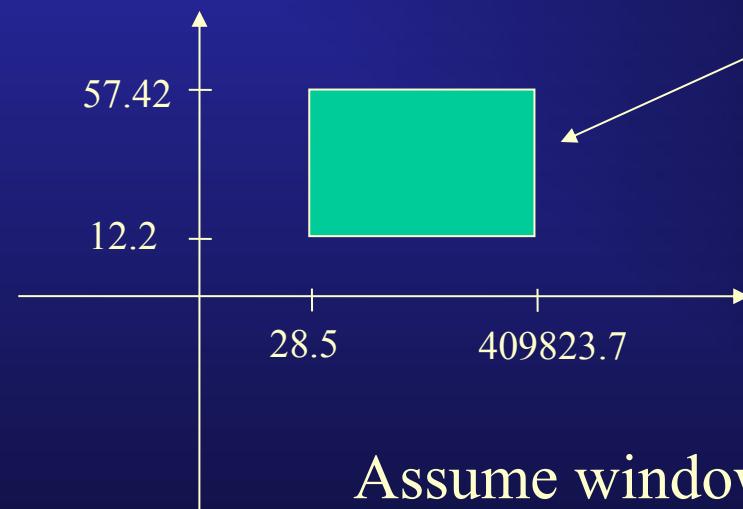
<http://i.stack.imgur.com/tL7r6.png>

2D Viewing Transformation: Convert 2D model coordinates to a physical display device

- 2D coordinate world
- 2D screen space
- Allow for different device resolutions



Window: That portion of the user's world coordinate space which is to be viewed

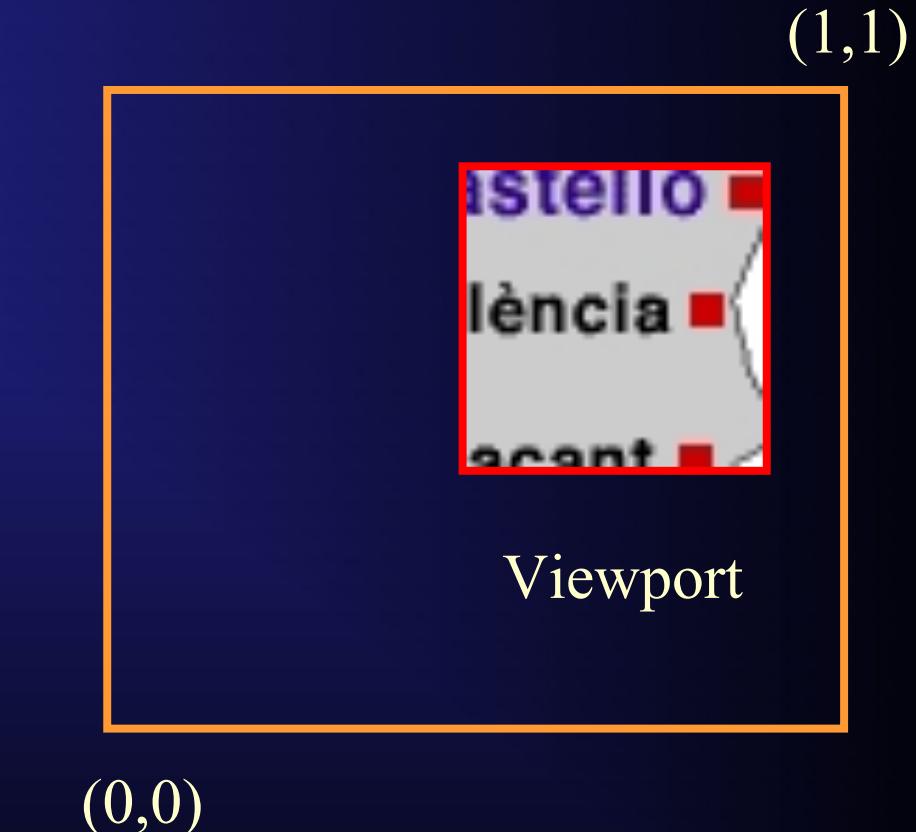


Window = area of interest within world

Assume window is rectangular

World coordinates AND units are chosen at the convenience of the application or user

Viewing Transformation: World to Normalized Device Coordinates (NDC)



Changing the viewport POSITION does NOT change the window data being displayed.

Normalized Device Coordinates

Viewing Transformation: World to Normalized Device Coordinates (NDC)



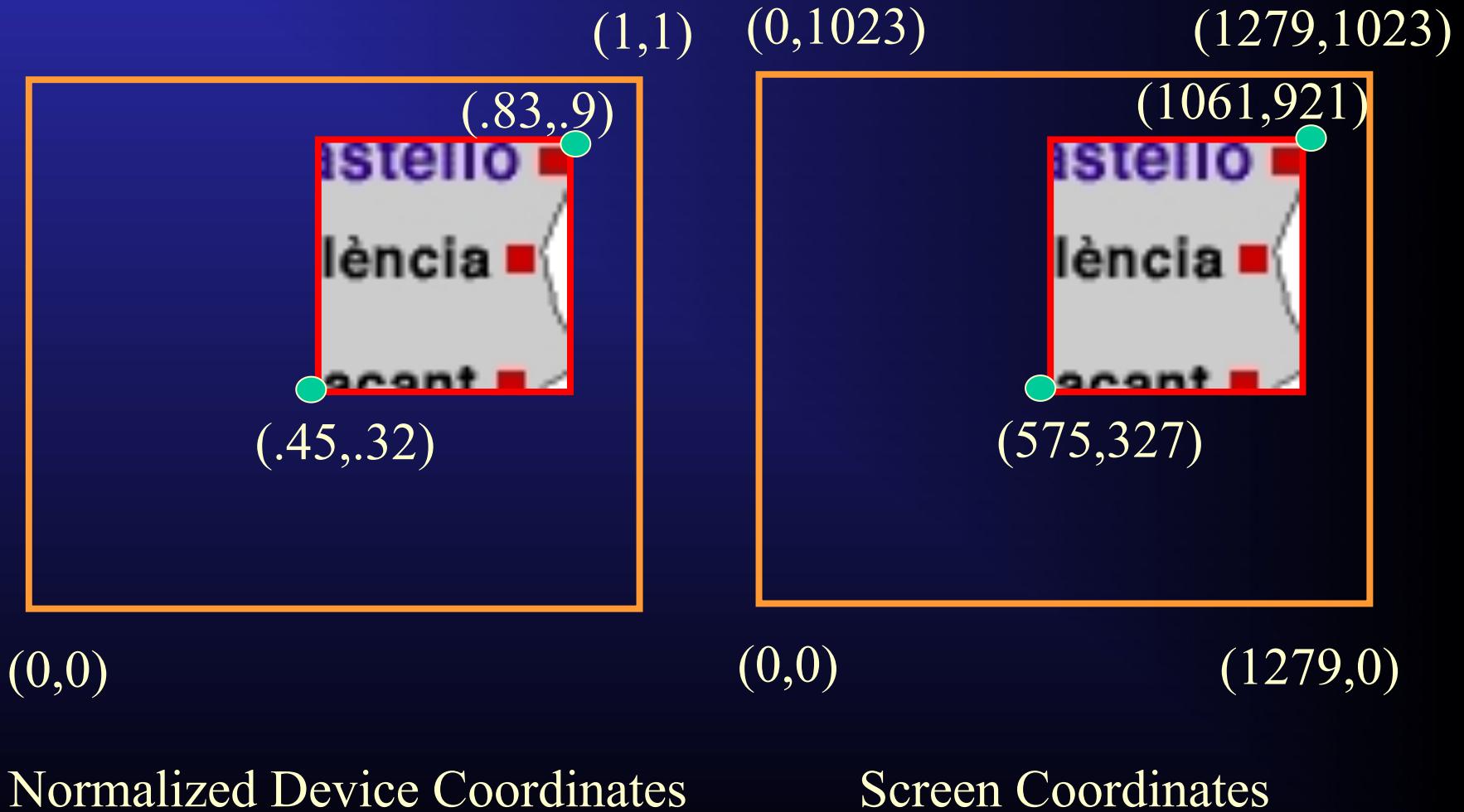
World Coordinates



Changing the WINDOW changes the data being displayed in the viewport.

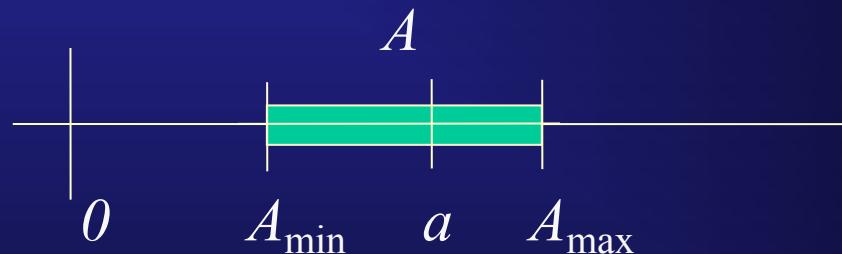
Normalized Device Coordinates

NDC to Screen

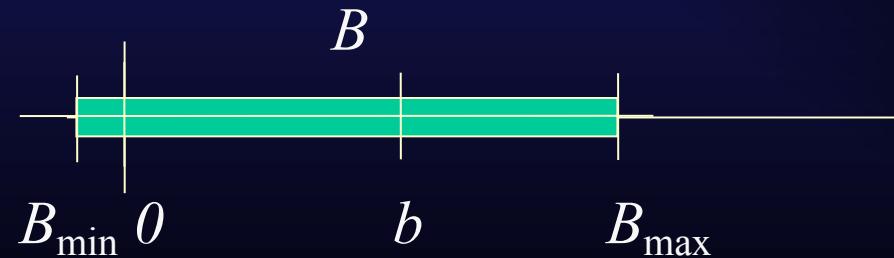


Range Mapping

- Given values in a range A , map them linearly into a (different) range of values B .
- Consider some arbitrary point a in A



- Find the image b of a in B



Solving for the Range Mapping

- Using simple proportions:

$$\frac{b - B_{\min}}{a - A_{\min}} = \frac{B_{\max} - B_{\min}}{A_{\max} - A_{\min}}$$

- Solving for b :

$$b = \frac{B_{\max} - B_{\min}}{A_{\max} - A_{\min}}(a - A_{\min}) + B_{\min}$$

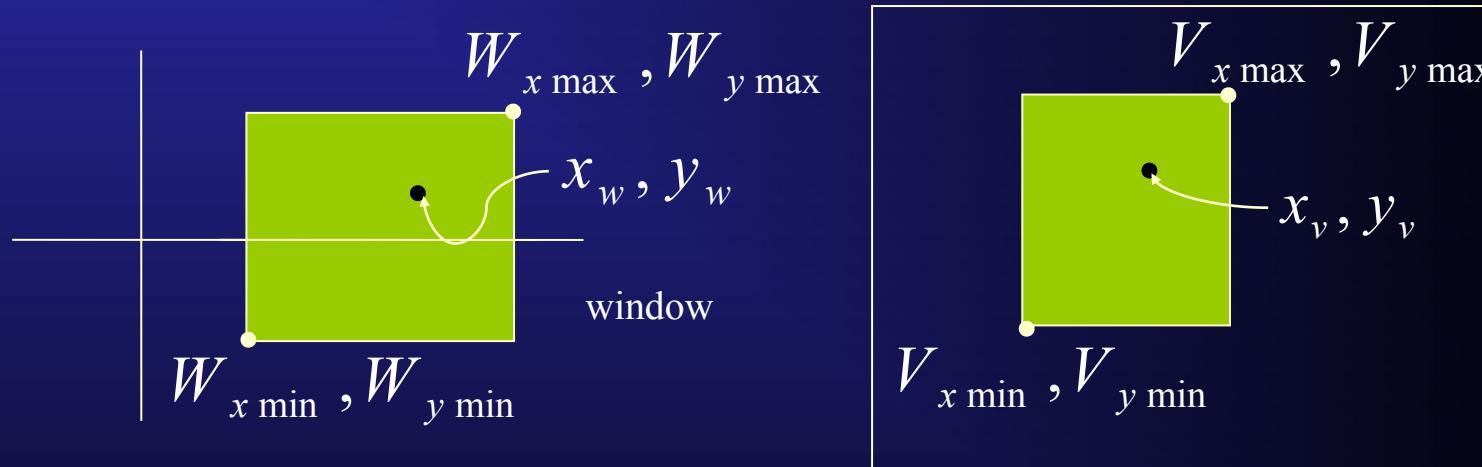
- In terms of transformations, the distance from a to A_{\min} is scaled by the ratio of the two ranges B and A :

$$\frac{B_{\max} - B_{\min}}{A_{\max} - A_{\min}}$$

- then translated from the end B_{\min} of B .

The Window to NDC Viewport Transformation

- The transformation is just two applications of range mapping (one for each coordinate axis):



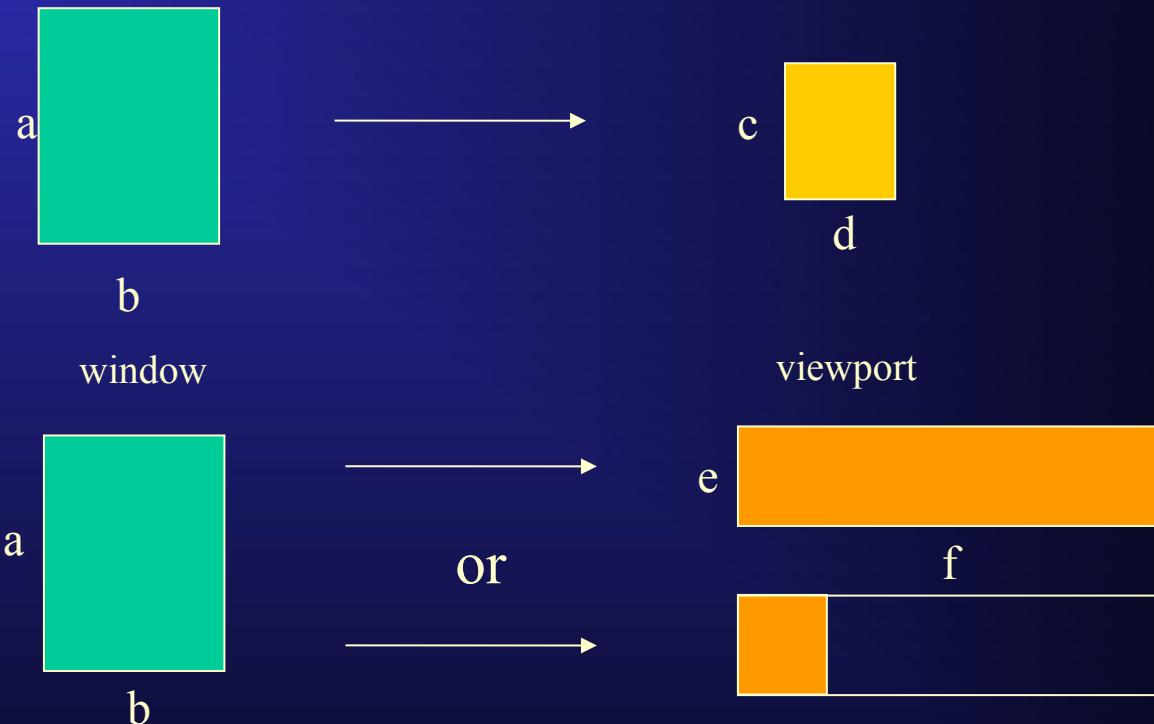
WORLD
COORDINATES

NORMALIZED DEVICE
COORDINATES

$$x_v = \frac{V_{x \max} - V_{x \min}}{W_{x \max} - W_{x \min}} (x_w - W_{x \min}) + V_{x \min}$$

$$y_v = \frac{V_{y \max} - V_{y \min}}{W_{y \max} - W_{y \min}} (y_w - W_{y \min}) + V_{y \min}$$

Usually must Account for Window and Viewport Aspect Ratio -- May be Different!

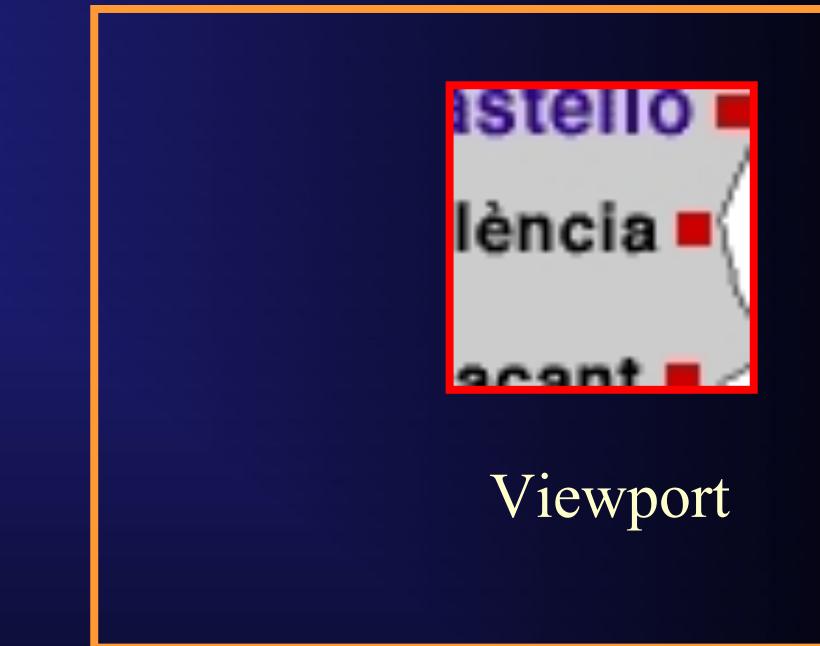


- If $\frac{a}{b} = \frac{c}{d}$ (= aspect ratio) then map causes no distortion
- If $\frac{a}{b} \neq \frac{e}{f}$ then stretching distortion or partial viewport fill can occur
- To avoid distortion, use $\min\left(\frac{e}{a}, \frac{f}{b}\right)$ as single scale factor in both x and y

Aspect Ratio Deformation (May or may not be what you want!)



Window



Changing the viewport aspect ratio deforms any window data displayed.

Viewport

Mapping the Viewport Back into the Window

- Note that the window-to-viewport transformation can be inverted (up to numerical resolution)

$$x_w = \frac{W_{x \max} - W_{x \min}}{V_{x \max} - V_{x \min}} (x_v - V_{x \min}) + W_{x \min}$$

$$y_w = \frac{W_{y \max} - W_{y \min}}{V_{y \max} - V_{y \min}} (y_v - V_{y \min}) + W_{y \min}$$

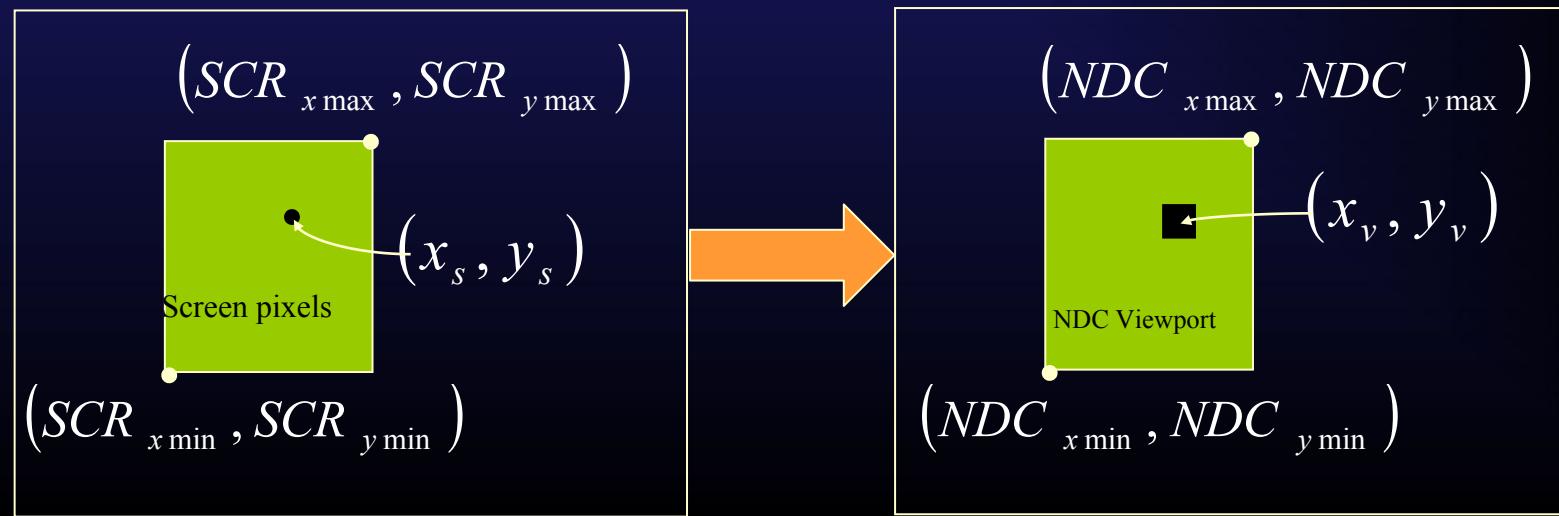
- by just solving the range equations for (x_w, y_w) in terms of the given (x_v, y_v) and the known window and viewport locations.

Inverting Integer 2D Screen Coordinates (SCR) to Floating Point Normalized Device Coordinates (NDC)

$$x_v = \frac{NDC_{x \max} - NDC_{x \min}}{SCR_{x \max} - SCR_{x \min}} (x_s - SCR_{x \min}) + NDC_{x \min}$$

$$y_v = \frac{NDC_{y \max} - NDC_{y \min}}{SCR_{y \max} - SCR_{y \min}} (y_s - SCR_{y \min}) + NDC_{y \min}$$

- Note that (x_s, y_s) are *integers* and (x_v, y_v) will be *floats*, so the mapping is **not one-to-one**; rather each pixel maps back to a **region** in the NDC viewport coordinate space:
- (x_v, y_v) will be the **center** of this region. (*It is left as an exercise to compute its actual size!*)

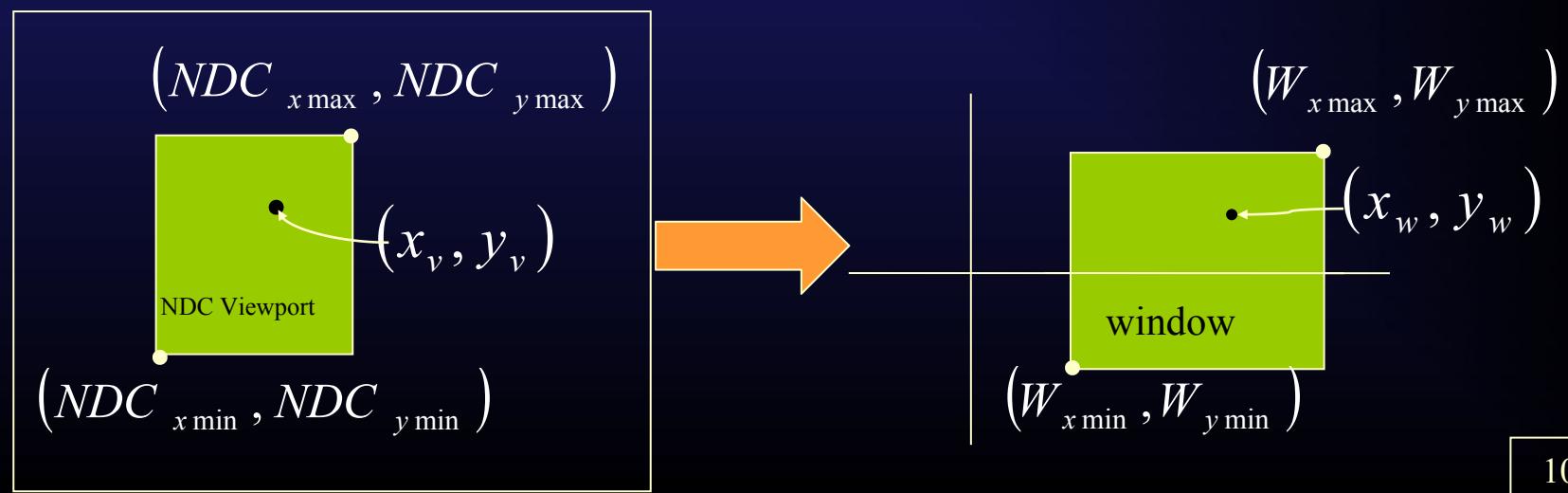


Inverting Floating Point NDC to Floating Point Window (World) Coordinates

$$x_w = \frac{W_{x \max} - W_{x \min}}{NDC_{x \max} - NDC_{x \min}} (x_v - NDC_{x \min}) + W_{x \min}$$

$$y_w = \frac{W_{y \max} - W_{y \min}}{NDC_{y \max} - NDC_{y \min}} (y_v - NDC_{y \min}) + W_{y \min}$$

- Note that (x_v, y_v) and (x_w, y_w) are *floats*, so *this* mapping **is** one-to-one.

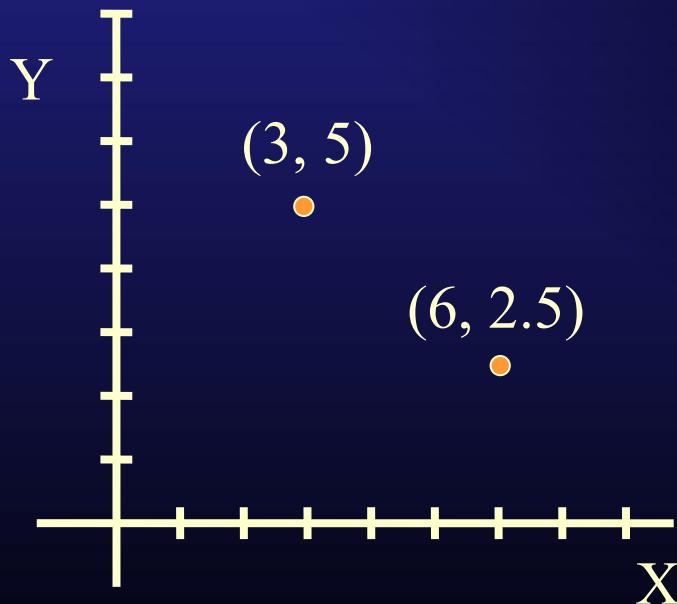


Outline

- Great Ideas ✓
- Virtual Environments & Devices ✓
- 3D Perception ✓
- Architecture, Color & Rasterization ✓
- **Vector Geometry & Transformations**
- 3D Modeling
- Embedding, Hierarchies & Contours
- Model Generation & Deformation
- Visible Surface Algorithms
- Polygon Algorithms
- Image Synthesis & Shading
- Mapping
- 3D Viewing Transformations
- Anti-aliasing & Compositing
- Global Illumination
- Light Fields & HDR
- Path Tracing/Photon Mapping
- Futures

Points

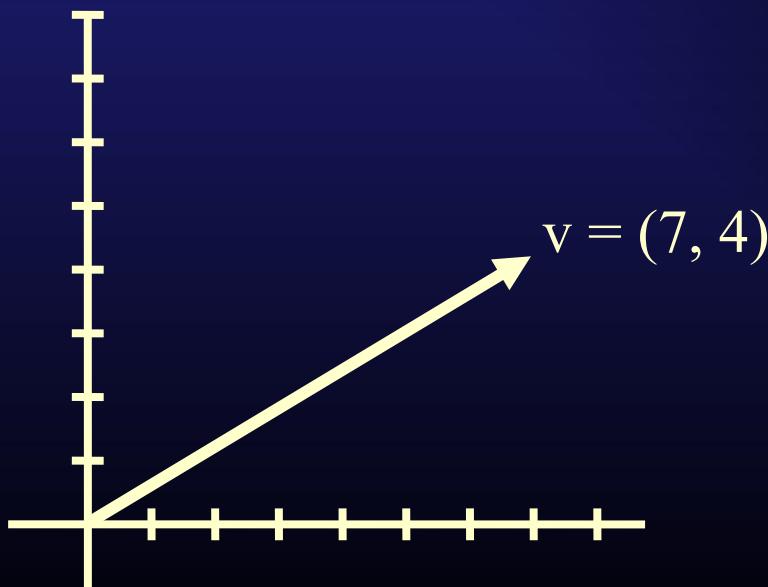
- A *point* is a “place” in space.
- In 2D a point is a 2-tuple of floats (x, y) .
- In 3D a point is a 3-tuple of floats (x, y, z) .



Thanks to David Brogan – from whom I adapted some of these mathematics slides.

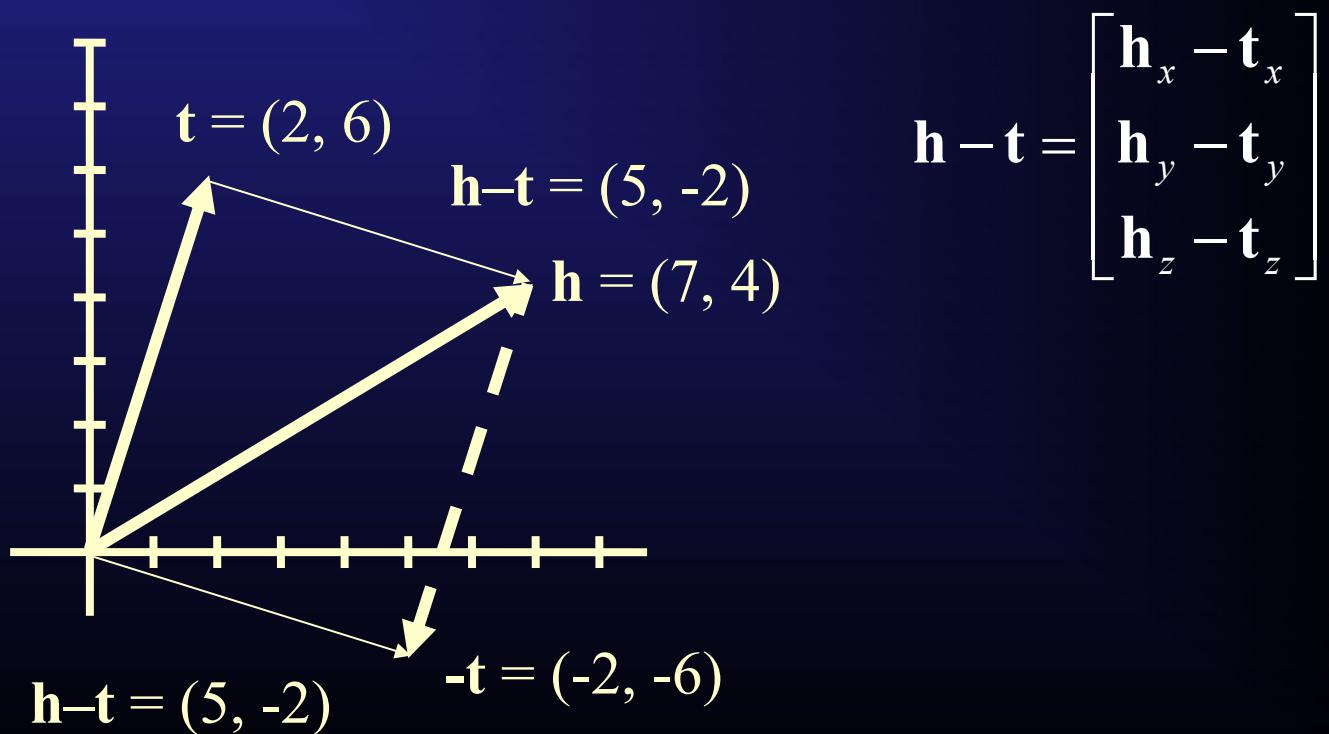
Vectors

- A *vector* is a **direction**.
- A vector **v** can be thought of as having a head at a point **v** = (**vx**, **vy**, **vz**) and a tail at the origin (0, 0, 0) that define its length and direction.
- If we translate (“slide”) a vector around it is still the same vector.



Vectors

- A *ordered* pair of points defines a vector; one is the tail and the other is the head. If **h** is the head point and **t** is the tail point, then the vector pointing to **h** starting at **t** is exactly **h-t** where the subtraction is done component-wise:



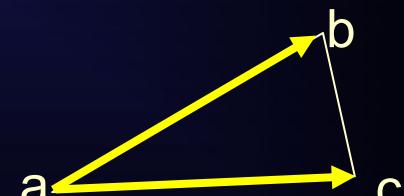
Triangles, Points, Vectors, and Arithmetic

- Consider a triangle having three points (vertices): (a, b, c) .
Each of a, b, c is a 3-tuple of scalars (x, y, z) .
- The *directions* between the triangle vertices are vectors, such as $(b - a)$ and $(c - a)$. ■■■
- Perpendicular (normal) to plane of triangle: $P = (b - a) \times (c - a)$ where \times is the *cross-product* operation*.
- $\| \|$ represents the *norm* or *distance* function:

$$\|P_1 - P_2\| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

for $P_i = (x_i, y_i, z_i)$, $i = 1, 2$

- Unit (length=1) normal is $\frac{P}{\|P\|}$



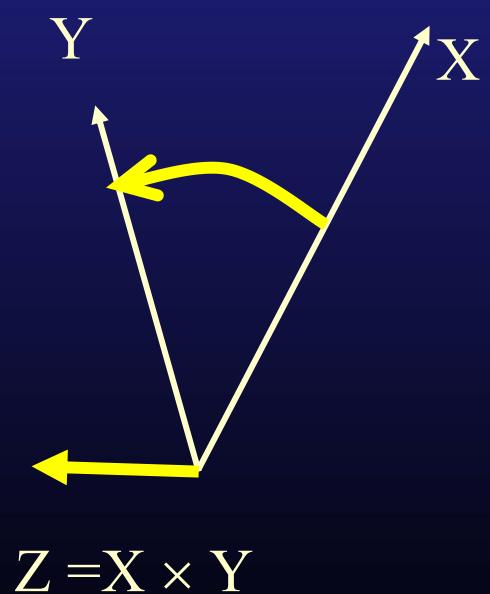
* (You don't remember how to do this, right?)

Vector Cross Product

Vector cross product has several uses:

- To find the perpendicular (normal) to the plane formed by two vectors.
- To get rotation axis to rotate one vector into alignment with another.
- To get area of parallelogram defined by two vectors.

“Right Hand Rule”



Assuming we're in
a right handed
coordinate system:
that is, $Z = X \times Y$.



Vector Cross Product

- Definition of cross product:

$$\mathbf{w} = \mathbf{u} \times \mathbf{v} \quad \text{for} \quad \mathbf{u} = (x_u, y_u, z_u) \\ \mathbf{v} = (x_v, y_v, z_v)$$

- Compute determinant of this matrix:

$$\mathbf{w} = \mathbf{u} \times \mathbf{v} = \begin{vmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} \\ x_u & y_u & z_u \\ x_v & y_v & z_v \end{vmatrix}; \quad \mathbf{x} = (1,0,0); \quad \mathbf{y} = (0,1,0); \quad \mathbf{z} = (0,0,1)$$

$$\mathbf{w} = (y_U z_V - y_V z_U) \mathbf{x} - (x_U z_V - x_V z_U) \mathbf{y} + (x_U y_V - x_V y_U) \mathbf{z}$$

- \mathbf{w} is perpendicular to the plane formed by \mathbf{u} and \mathbf{v} , and moreover, points in the direction defined by the right-hand rule for rotating the first vector towards the second.

Vector Spaces

- A *linear combination* of vectors results in a new vector:

$$\mathbf{v} = \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2 + \dots + \alpha_n \mathbf{v}_n$$

- If the only set of scalars such that

$$\alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2 + \dots + \alpha_n \mathbf{v}_n = \mathbf{0}$$

is $\alpha_1 = \alpha_2 = \dots = \alpha_n = 0$

then we say the vectors are *linearly independent*.

- The *dimension* of a space is the greatest number of linearly independent vectors possible in a vector set.
- For a vector space of dimension n , any set of n linearly independent vectors form a *basis*.

Vector Spaces: Basis Vectors

Given a basis for a vector space:

- Each vector in the space is a *unique* linear combination of the basis vectors.
- The *coordinates* of a vector are the scalars from this linear combination.
- If basis vectors are *orthogonal* (perpendicular) and unit length (1), then the vectors comprise an *orthonormal basis*.
- Best-known example: Cartesian coordinates with basis $(1,0,0)$, $(0,1,0)$, $(0,0,1)$. Other 3D basis vector sets are possible; in fact, an infinite number are possible.
- Note that a given vector \mathbf{v} will have different coordinates for different bases.

Matrices

Matrices can represent *collections* efficiently:

- Tuples (points)
- Vectors
- Transformations of points or vectors
- Collections of expressions (for solving multi-variable formulas)
- Sets of points or vectors

Matrices

- By mathematics convention, matrix element M_{rc} is located at row r and column c .

$$M = \begin{bmatrix} M_{11} & M_{12} & \cdots & M_{1n} \\ M_{21} & M_{22} & \cdots & M_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ M_{m1} & M_{m2} & \cdots & M_{mn} \end{bmatrix}$$

$$\mathbf{v} = \begin{bmatrix} \mathbf{v}_x \\ \mathbf{v}_y \\ \mathbf{v}_z \end{bmatrix}$$

- By computer graphics (OpenGL) convention, vectors are columns.
- Sometimes for convenience we write the *transpose*:

$$\mathbf{v} = [\mathbf{v}_x \quad \mathbf{v}_y \quad \mathbf{v}_z]^T$$

Transpose Operator

- The *transpose* $[]^T$ “flips” the matrix rows and columns:

$$\mathbf{M} = \begin{bmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} & \cdots & \mathbf{M}_{1n} \\ \mathbf{M}_{21} & \mathbf{M}_{22} & \cdots & \mathbf{M}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{M}_{m1} & \mathbf{M}_{m2} & \cdots & \mathbf{M}_{mn} \end{bmatrix}$$

$$\mathbf{M}_{ij}^T = \mathbf{M}_{ji}$$

Matrices

- Matrix-vector multiplication • applies a linear transformation to a vector:

$$\mathbf{M} \bullet \mathbf{v} = \begin{bmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} & \mathbf{M}_{13} \\ \mathbf{M}_{21} & \mathbf{M}_{22} & \mathbf{M}_{23} \\ \mathbf{M}_{31} & \mathbf{M}_{32} & \mathbf{M}_{33} \end{bmatrix} \begin{bmatrix} \mathbf{v}_x \\ \mathbf{v}_y \\ \mathbf{v}_z \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

- Why do we call this a linear transformation?

Matrices

- Because this

$$\mathbf{M} \bullet \mathbf{v} = \begin{bmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} & \mathbf{M}_{13} \\ \mathbf{M}_{21} & \mathbf{M}_{22} & \mathbf{M}_{23} \\ \mathbf{M}_{31} & \mathbf{M}_{32} & \mathbf{M}_{33} \end{bmatrix} \begin{bmatrix} \mathbf{v}_x \\ \mathbf{v}_y \\ \mathbf{v}_z \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

is just mathematical shorthand for the collection of
THREE simultaneous linear equations:

$$M_{11} \cdot v_x + M_{12} \cdot v_y + M_{13} \cdot v_z = a$$

$$M_{21} \cdot v_x + M_{22} \cdot v_y + M_{23} \cdot v_z = b$$

$$M_{31} \cdot v_x + M_{32} \cdot v_y + M_{33} \cdot v_z = c$$

Matrices

- So this tells us how to do matrix multiplication:

$$\mathbf{M} \bullet \mathbf{K} = \begin{bmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} & \mathbf{M}_{13} \\ \mathbf{M}_{21} & \mathbf{M}_{22} & \mathbf{M}_{23} \\ \mathbf{M}_{31} & \mathbf{M}_{32} & \mathbf{M}_{33} \end{bmatrix} \begin{bmatrix} \mathbf{K}_{11} & \mathbf{K}_{12} & \mathbf{K}_{13} \\ \mathbf{K}_{21} & \mathbf{K}_{22} & \mathbf{K}_{23} \\ \mathbf{K}_{31} & \mathbf{K}_{32} & \mathbf{K}_{33} \end{bmatrix}$$
$$= \begin{bmatrix} \mathbf{M}_{11} \cdot \mathbf{K}_{11} + \mathbf{M}_{12} \cdot \mathbf{K}_{12} + \mathbf{M}_{13} \cdot \mathbf{K}_{13} & \mathbf{M}_{11} \cdot \mathbf{K}_{12} + \mathbf{M}_{12} \cdot \mathbf{K}_{22} + \mathbf{M}_{13} \cdot \mathbf{K}_{32} & \mathbf{M}_{11} \cdot \mathbf{K}_{13} + \mathbf{M}_{12} \cdot \mathbf{K}_{23} + \mathbf{M}_{13} \cdot \mathbf{K}_{33} \\ \mathbf{M}_{21} \cdot \mathbf{K}_{11} + \mathbf{M}_{22} \cdot \mathbf{K}_{12} + \mathbf{M}_{23} \cdot \mathbf{K}_{13} & \mathbf{M}_{21} \cdot \mathbf{K}_{12} + \mathbf{M}_{22} \cdot \mathbf{K}_{22} + \mathbf{M}_{23} \cdot \mathbf{K}_{32} & \mathbf{M}_{21} \cdot \mathbf{K}_{13} + \mathbf{M}_{22} \cdot \mathbf{K}_{23} + \mathbf{M}_{23} \cdot \mathbf{K}_{33} \\ \mathbf{M}_{31} \cdot \mathbf{K}_{11} + \mathbf{M}_{32} \cdot \mathbf{K}_{12} + \mathbf{M}_{33} \cdot \mathbf{K}_{13} & \mathbf{M}_{31} \cdot \mathbf{K}_{12} + \mathbf{M}_{32} \cdot \mathbf{K}_{22} + \mathbf{M}_{33} \cdot \mathbf{K}_{32} & \mathbf{M}_{31} \cdot \mathbf{K}_{13} + \mathbf{M}_{32} \cdot \mathbf{K}_{23} + \mathbf{M}_{33} \cdot \mathbf{K}_{33} \end{bmatrix}$$

- For example:

$$\begin{bmatrix} -1 & 2 & 4 \\ 3 & 1 & -2 \\ 0 & 5 & 2 \end{bmatrix} \begin{bmatrix} 6 & 0 & 3 \\ -1 & 2 & 0 \\ 5 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 12 & 8 & 1 \\ 7 & 0 & 7 \\ 5 & 12 & 2 \end{bmatrix}$$

Matrices

$$\mathbf{M} \bullet \mathbf{K} = \begin{bmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} & \mathbf{M}_{13} \\ \mathbf{M}_{21} & \mathbf{M}_{22} & \mathbf{M}_{23} \\ \mathbf{M}_{31} & \mathbf{M}_{32} & \mathbf{M}_{33} \end{bmatrix} \begin{bmatrix} \mathbf{K}_{11} & \mathbf{K}_{12} & \mathbf{K}_{13} \\ \mathbf{K}_{21} & \mathbf{K}_{22} & \mathbf{K}_{23} \\ \mathbf{K}_{31} & \mathbf{K}_{32} & \mathbf{K}_{33} \end{bmatrix}$$
$$= \begin{bmatrix} \sum_{i=1}^3 \mathbf{M}_{1i} \mathbf{K}_{i1} & \sum_{i=1}^3 \mathbf{M}_{1i} \mathbf{K}_{i2} & \sum_{i=1}^3 \mathbf{M}_{1i} \mathbf{K}_{i3} \\ \sum_{i=1}^3 \mathbf{M}_{2i} \mathbf{K}_{i1} & \sum_{i=1}^3 \mathbf{M}_{2i} \mathbf{K}_{i2} & \sum_{i=1}^3 \mathbf{M}_{2i} \mathbf{K}_{i3} \\ \sum_{i=1}^3 \mathbf{M}_{3i} \mathbf{K}_{i1} & \sum_{i=1}^3 \mathbf{M}_{3i} \mathbf{K}_{i2} & \sum_{i=1}^3 \mathbf{M}_{3i} \mathbf{K}_{i3} \end{bmatrix}$$

- We can even make this look simpler...

Matrices

- Very compactly:

$$(\mathbf{M} \bullet \mathbf{K})_{rc} = \sum_{i=1}^n \mathbf{M}_{ri} \cdot \mathbf{K}_{ic} \quad r = 1, \dots, r_{\mathbf{M}}; c = 1, \dots, c_{\mathbf{K}}$$

where \mathbf{M} is $r_{\mathbf{M}} \times c_{\mathbf{M}}$ and \mathbf{K} is $r_{\mathbf{K}} \times c_{\mathbf{K}}$.

- There's nothing special about 3×3 matrices or even that they are *square* (#rows need not be the same as #columns):
- As long as the **number of COLUMNS of the left side = the number of ROWS of the right side ($=n$)**, the matrices can be multiplied.
- When it is clear, we will omit writing the \bullet operator explicitly.

Matrix Transformations

- A *sequence* or *concatenation* of linear transformations corresponds to the product of the corresponding matrices

$$\mathbf{M}_1 \mathbf{M}_2 \mathbf{M}_3 \mathbf{v}_{old} = \mathbf{v}_{new}$$

- Note: the matrices to the *right* affect the vector first!

$$= \mathbf{M}_1 (\mathbf{M}_2 (\mathbf{M}_3 \mathbf{v}_{old})) = \mathbf{v}_{new}$$

- Note: order of matrices matters since matrix multiplication is not commutative!

$$\mathbf{M}_1 \mathbf{M}_2 \neq \mathbf{M}_2 \mathbf{M}_1$$

Matrix Properties

- Matrix multiplication *is* associative, however:

$$(\mathbf{M}_1 \mathbf{M}_2) \mathbf{M}_3 = \mathbf{M}_1 (\mathbf{M}_2 \mathbf{M}_3)$$

- So, in particular:

$$\mathbf{M}_1 \mathbf{M}_2 \mathbf{v}_{old} = \mathbf{M}_1 (\mathbf{M}_2 \mathbf{v}_{old}) = (\mathbf{M}_1 \mathbf{M}_2) \mathbf{v}_{old} = \mathbf{v}_{new}$$

- This will be particularly useful soon.
- The ***identity matrix*** \mathbf{I} has no effect in multiplication.

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{M} \bullet \mathbf{I} = \mathbf{I} \bullet \mathbf{M} = \mathbf{M}$$

- Some (but not all) matrices have an inverse \mathbf{M}^{-1} :

$$\mathbf{M}^{-1}(\mathbf{M}(\mathbf{v})) = (\mathbf{M}^{-1}\mathbf{M})\mathbf{v} = \mathbf{I} \bullet \mathbf{v} = \mathbf{v}$$

A Note on Float Equality Comparisons

- Due to *rounding errors* and *finite bit representation*, most floating-point numbers are numerically imprecise. This means that numbers expected to be equal (e.g., when calculating the same result through different correct methods or making equality comparisons) often differ slightly, and a simple equality test fails. For example:

```
float a = 0.15 + 0.15
```

```
float b = 0.1 + 0.2
```

```
if (a == b) // can be false!
```

```
if (a >= b) // can also be false!
```

Absolute Error Margins – Not So Good

- The first (naïve) approach is to check whether the difference between two floats is very small. The difference error allowed is often called *epsilon* (ϵ). The simplest form is, e.g., with $\epsilon = 0.00001$:

```
#include<cmath>
if (fabs(a-b) < 0.00001) { // wrong - don't do this
```

- This is bad because a fixed ϵ may “look small” but might actually be way too *large* when the numbers being compared are *very small as well*. Then this comparison would return **TRUE** for numbers that are actually quite different.
- And when the numbers are very large, ϵ could be smaller than the smallest rounding error, so that the comparison always returns **FALSE**. Therefore, it is necessary at least to see whether the *relative* error is smaller than ϵ , but...:

```
if (fabs((a-b)/b) < 0.00001 ) { // this is still not right!
```

The Problem Lies in the Special Cases

- Some important special cases cause comparison failure:
 - When **b** is float zero, the IEEE standard gives a well-defined behavior of “infinity” or “not a number” depending on **a**, either of which is not what you want.
 - It returns **FALSE** when both **a** and **b** are very small but on opposite sides of zero, even when they’re the smallest possible non-zero numbers.
 - Finally, the result is not commutative (**nearlyEquals(a,b)** is not always the same as **nearlyEquals(b,a)**)! To fix these problems, the code has to get a lot more complex, so we really need to put it into a function of its own:

∴ A nearlyEqual function that works:

```
bool nearlyEqual(float a, float b, float epsilon){  
    {  
        const float absA = fabs(a);  
        const float absB = fabs(b);  
        const float diff = fabs(a - b);  
  
        if (a == b) { // shortcut  
            return true;  
        } else if (a * b == 0) { // a or b or both are zero  
            // relative error is not meaningful here  
            return diff < (epsilon * epsilon);  
        } else { // use relative error  
            return diff / (absA + absB) < epsilon;  
        }  
    }  
}
```

Spatial Transformations

- A *linear transformation*:
 - Maps one vector into another
 - Preserves linear combinations
- Thus behavior of a linear transformation is completely determined by what it does to a basis.
- Any linear transform can be represented by a *matrix* .

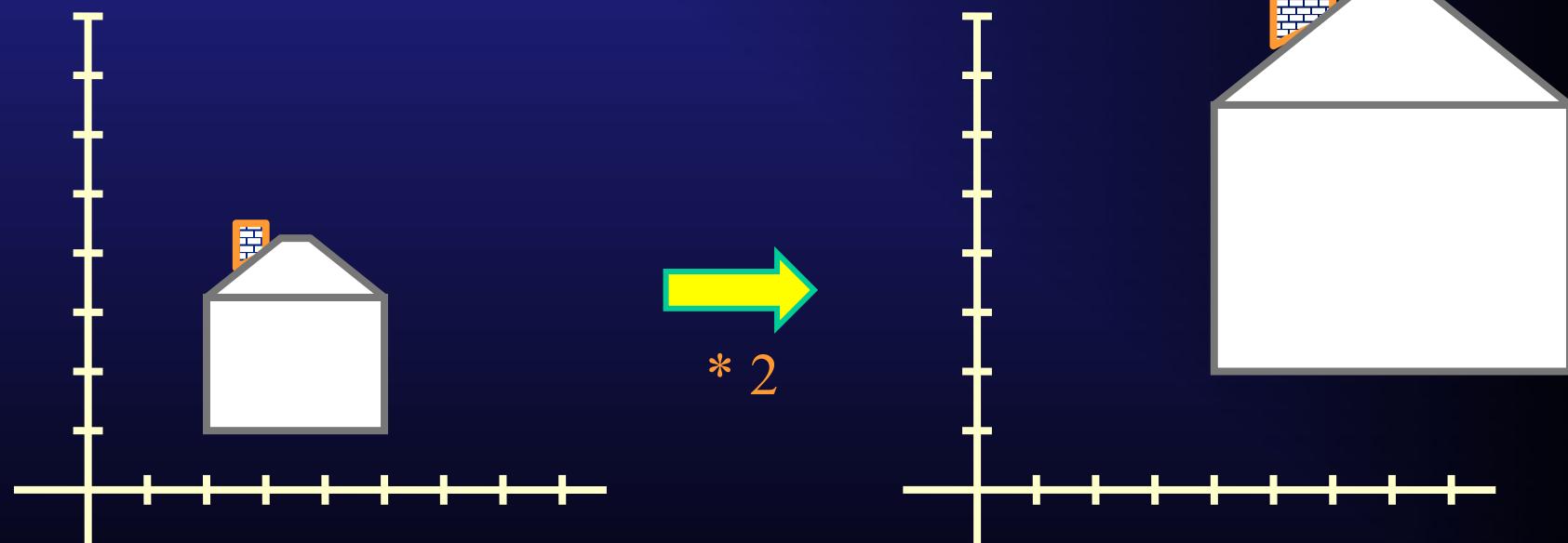
Matrix Transformations

We hypothesize that *all necessary linear transformations can be accomplished with matrix multiplication:*

- Scaling
- Rotation
- Translation (but we'll need to do something special)

Scaling

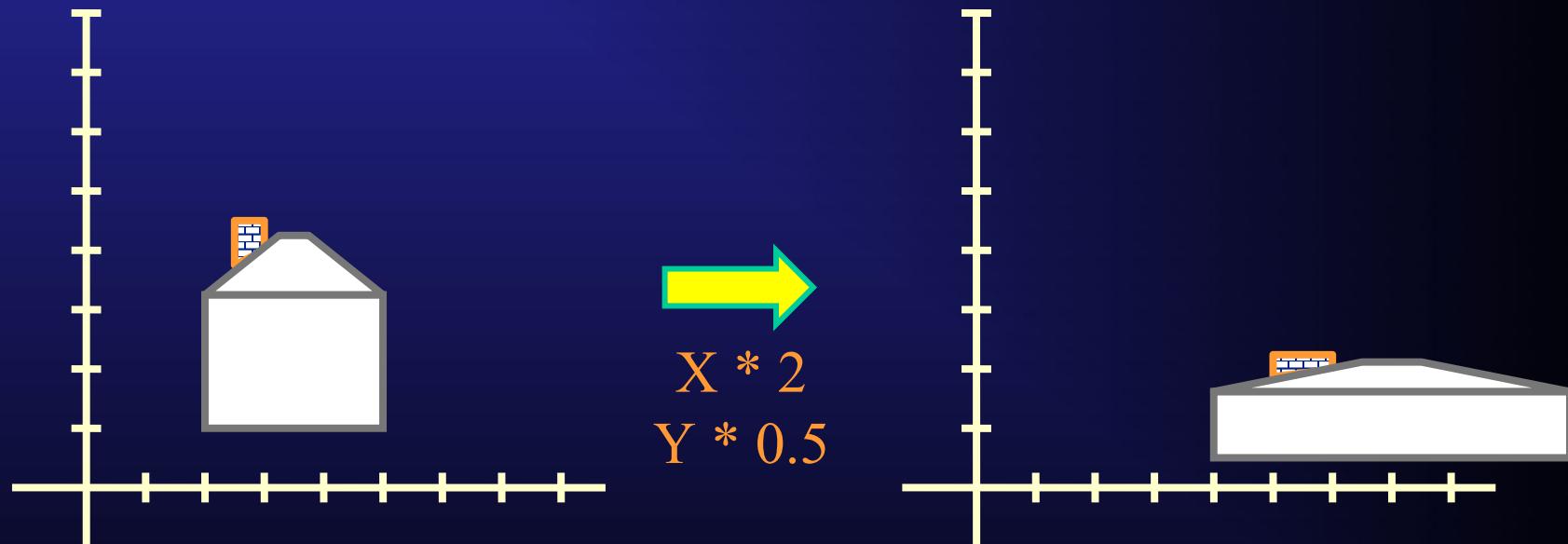
- *Scaling* a coordinate means multiplying each of its components by a scalar.
- *Uniform scaling* means this scalar is the same for all components:



- Note that the scale is relative to the origin $(0, 0)$.

Scaling

- *Non-uniform scaling:* different scalars per component:



- *How can we represent this in matrix form?*

Scaling

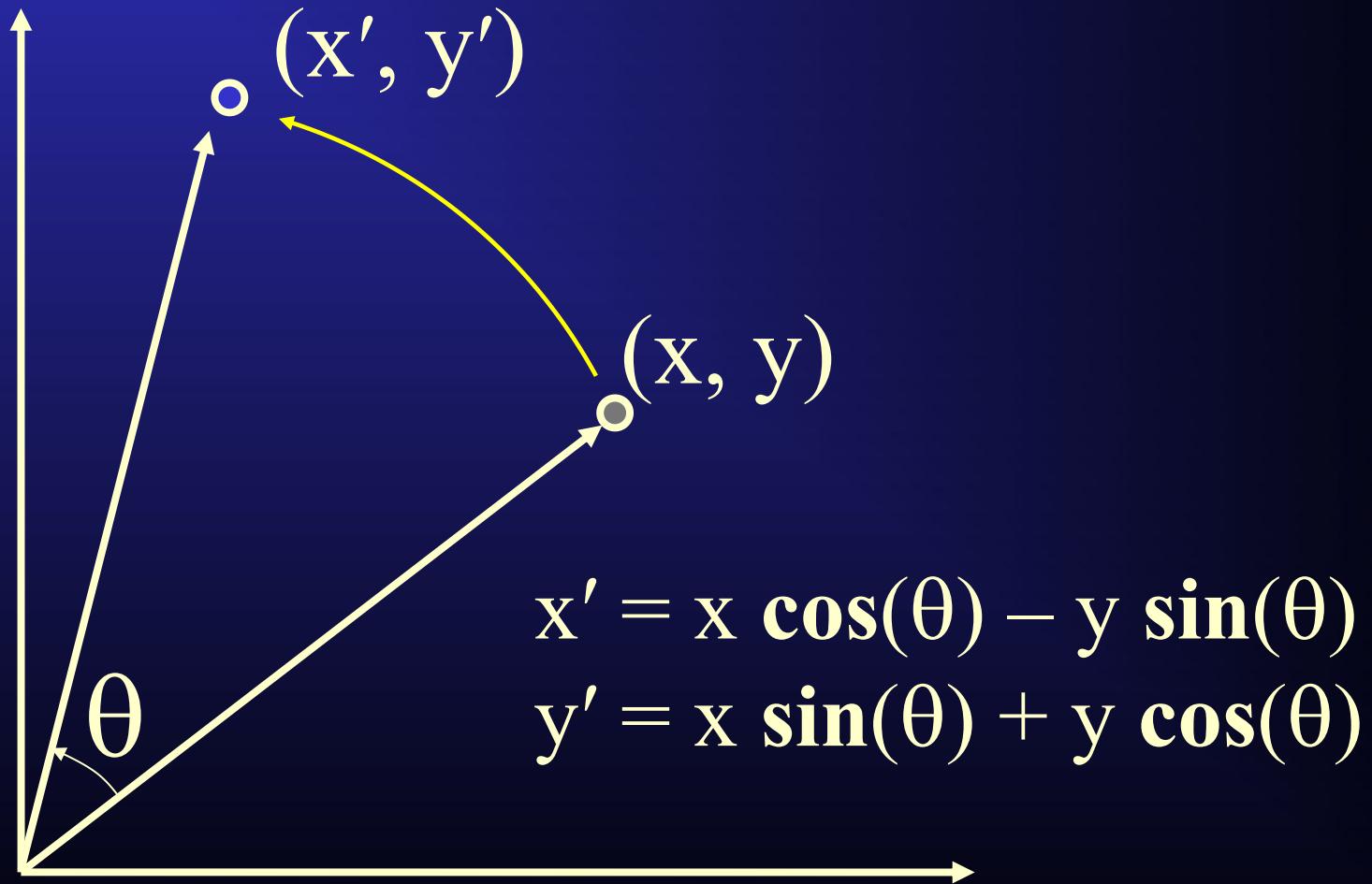
- Scaling formulas: $x' = ax$ and $y' = by$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} ax \\ by \end{bmatrix}$$

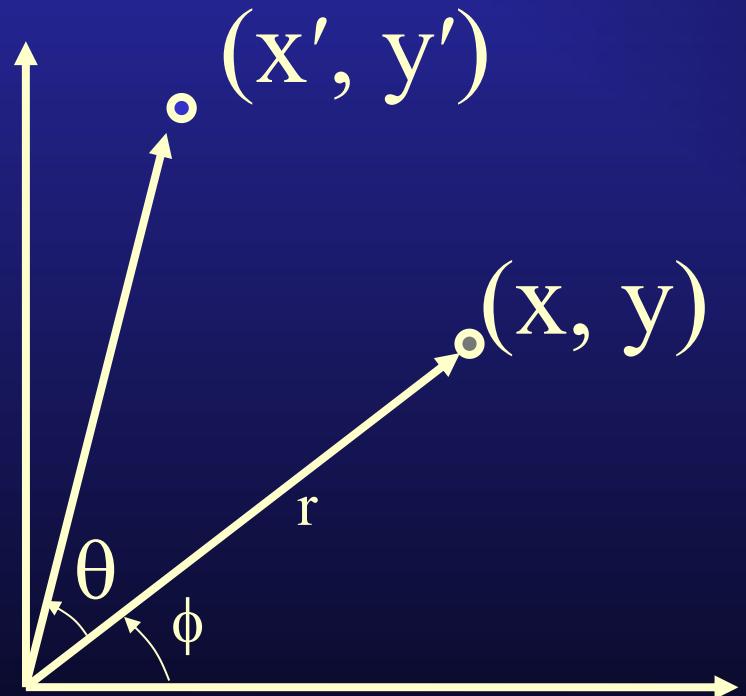
- Scaling operation:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \underbrace{\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}}_{\text{scaling matrix}} \begin{bmatrix} x \\ y \end{bmatrix}$$

2D Rotation



2D Rotation



$$\left. \begin{array}{l} x = r \cos (\phi) \\ y = r \sin (\phi) \\ x' = r \cos (\phi + \theta) \\ y' = r \sin (\phi + \theta) \end{array} \right\}$$

Trig Identity...

$$\begin{aligned} x' &= r \cos(\phi) \cos(\theta) - r \sin(\phi) \sin(\theta) \\ y' &= r \sin(\phi) \cos(\theta) + r \cos(\phi) \sin(\theta) \end{aligned}$$

Substitute...

$$\begin{aligned} x' &= x \cos(\theta) - y \sin(\theta) \\ y' &= x \sin(\theta) + y \cos(\theta) \end{aligned}$$

2D Rotation

- This is easy to capture in matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- Even though $\sin(\theta)$ and $\cos(\theta)$ are nonlinear functions of θ ,
 - x' is a linear combination of x and y
 - y' is a linear combination of x and y

Translation

- What can you do with a 2×2 matrix transformation?

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

- Scaling and rotation, but nothing corresponds to translation, since that involves the addition of a constant to a coordinate value, so...

Homogeneous Coordinates

- So we use

Homogeneous coordinates

to represent coordinates in 2 dimensions with a 3-vector (likewise we'll use a 4-vector for 3 dimensions).

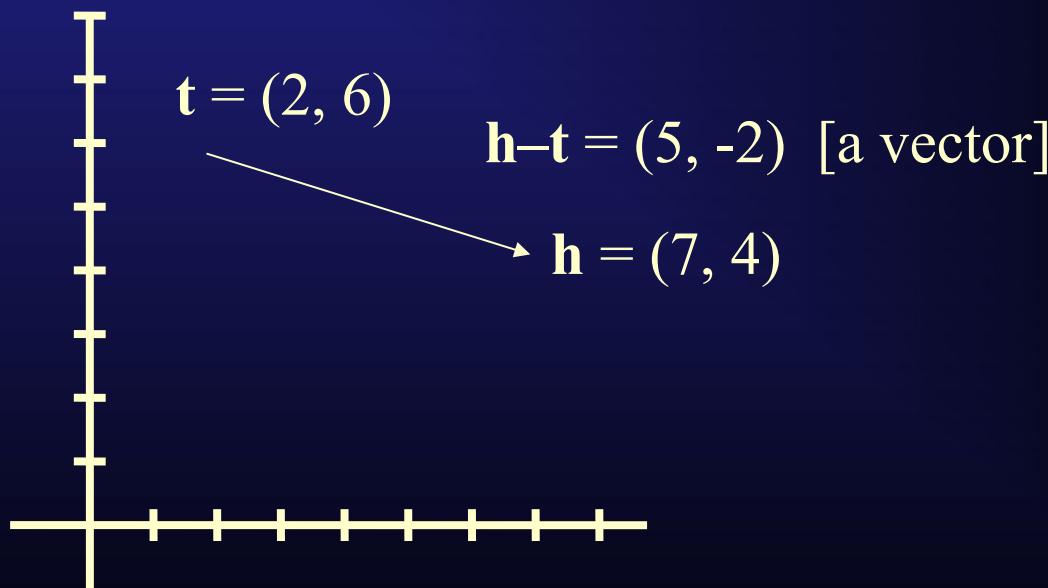
- Homogeneous coordinates seem unintuitive, but they make graphics (transformation) operations much easier.
- BONUS: Distinguishes points and directions (vectors).

$$\begin{bmatrix} x \\ y \end{bmatrix} \xrightarrow{\text{Homogeneous point coordinates}} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} \xrightarrow{\text{Homogeneous vector coordinates}} \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}$$

Subtracting Homogeneous Points Yields the Vector Between Them!

$$\mathbf{h} - \mathbf{t} = \begin{bmatrix} \mathbf{h}_x - \mathbf{t}_x \\ \mathbf{h}_y - \mathbf{t}_y \\ \mathbf{h}_z - \mathbf{t}_z \end{bmatrix}, \text{e.g.: } \begin{bmatrix} 7 \\ 4 \\ 1 \end{bmatrix} - \begin{bmatrix} 2 \\ 6 \\ 1 \end{bmatrix} = \begin{bmatrix} 7-2 \\ 4-6 \\ 1-1 \end{bmatrix} = \begin{bmatrix} 5 \\ -2 \\ 0 \end{bmatrix}$$



Homogeneous Coordinates

- Our 2D transformation matrices are now 3×3 :

$$\text{Rotation} \equiv \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{Scale} \equiv \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Homogeneous Coordinates

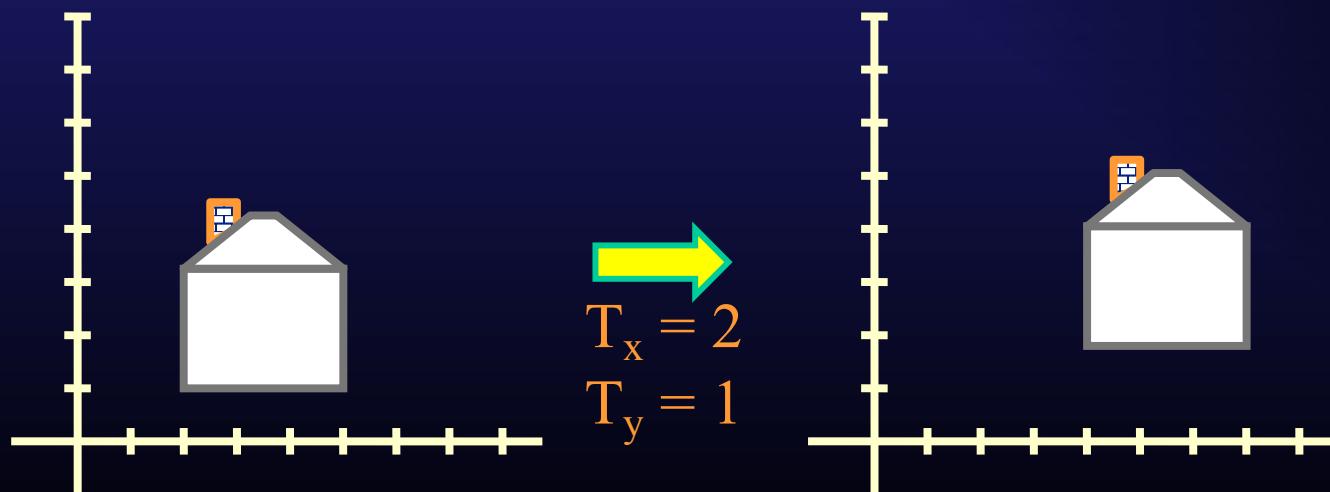
- **Q: How can we represent translation as a 3x3 matrix?**
- A: By using the rightmost column:

$$\text{Translation} = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix}$$

Translation

- 2D translation example

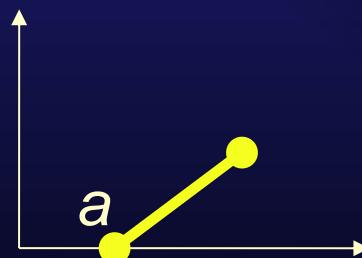
$$\begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x+T_x \\ y+T_y \\ 1 \end{bmatrix}$$



Concatenating Transformations

What if we want to *scale* then *rotate* then *translate*?

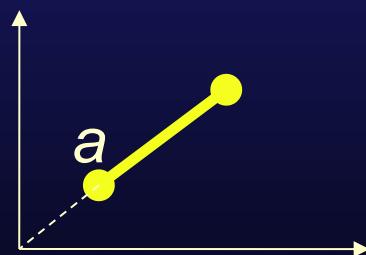
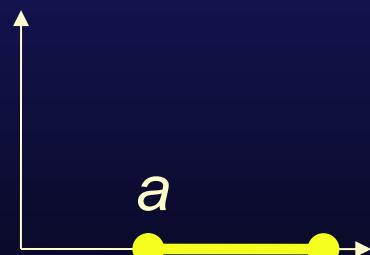
- We can execute transformations one after the other, i.e., multiply them in sequence.
- Ex: Rotate line segment by 45° about endpoint $a = (3, 0)$.



Multiplication Order – Must Get it Right!

Our line is defined by two endpoints, so:

- Applying a rotation of 45° via $R(45^\circ)$ to the segment affects both endpoints. ■
- We could first translate the endpoint we want to rotate about (point a) to the origin $T(-a_x, -a_y)$, then rotate about the origin (which we know how to do), then finally return the rotation center endpoint a' back to its original position $T(a_x, a_y)$. ■



Wrong
 $R(45^\circ)$

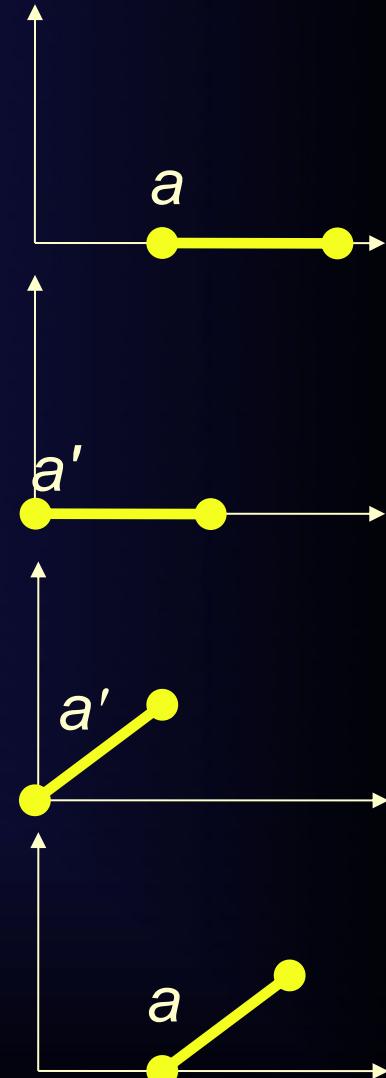


Correct
 $T(3,0) R(45^\circ) T(-3,0)$

Multiplication Order - Correct

Isolate endpoint a from rotation effects by moving it (and all the rest of the relevant geometry) to the origin:

- First translate line so a moves to origin:
 $T(-3,0)$
- Then rotate line (both endpoints) by 45° :
 $R(45^\circ)$
- Then translate back so a is where it was:
 $T(3,0)$



Concatenating Matrices

- This is the sequence of transformations $T(3,0) R(45^\circ) T(-3,0)$:

$$\begin{bmatrix} a'_x \\ a'_y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(45^\circ) & -\sin(45^\circ) & 0 \\ \sin(45^\circ) & \cos(45^\circ) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_x \\ a_y \\ 1 \end{bmatrix}$$

- Note the first to be applied is on the right!
- Don't be tempted to commute the matrices to "cancel" the left and right matrices! That's not allowed – and is just wrong besides.

Concatenating Matrices – Efficiency

- Since we need to use this matrix product to transform both segment endpoints, multiply matrices together FIRST and save, then multiply the saved product by each point.
- All vertices are easily transformed with one matrix multiply, e.g., let:

$$\mathbf{M} = T(3,0)R(45^\circ)T(-3,0)$$

- Collect multiple vertices of a more complex shape into a matrix of points, and transform via matrix multiplication:

$$\mathbf{M} \begin{bmatrix} v_0 & v_1 & v_2 & \dots & v_{n-1} \end{bmatrix} = \begin{bmatrix} v'_0 & v'_1 & v'_2 & \dots & v'_{n-1} \end{bmatrix}$$

- This works, by the definition of matrix multiplication, for any n .

Order of Transformations

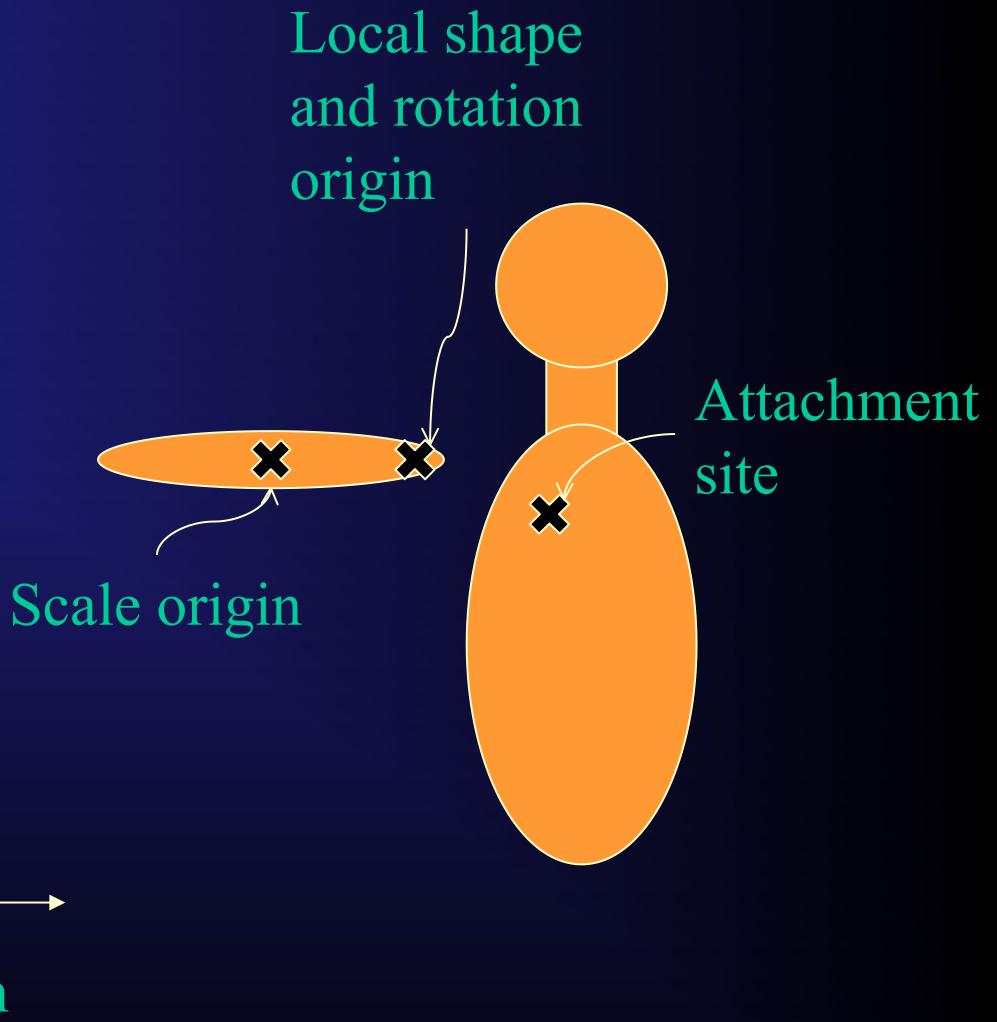
- Since Scale and Rotate involve an implicit origin, order of transformation is crucial to producing the result expected.
- Normally:
 - [Translate to desired scale origin]
 - Scale by desired factor(s)
 - [Translate to desired rotation origin if different from scale origin]
 - Rotate by desired angle
 - [Translate to undo rotation origin, if necessary]
 - [Translate to undo scale origin, if necessary (i.e., if different from scale origin)]
 - Translate to desired location

Example (with gratuitous animation)

- Scale just upper arm to thicken it:
 $T_S S(1, 1.5) T^{-1}_S$
- Rotate by 45° CW:
 $T_R R(-45^\circ) T^{-1}_R$
- Translate it to shoulder:
 $T_{\text{attachment}}$



World origin



Nice Side Effects: Parameterization!

- Scale in y can be independently set via S:

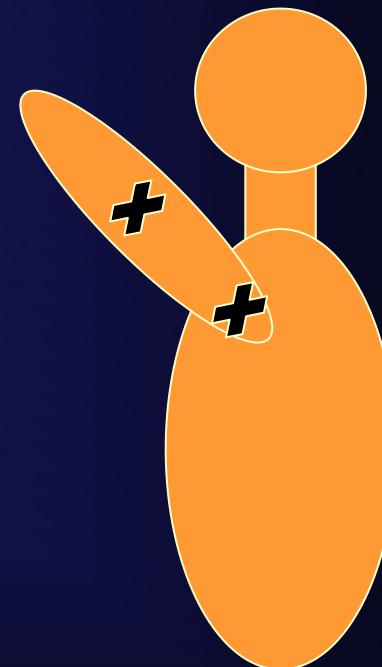
$$T_S S(1,y) T^{-1}_S$$

- And Rotation likewise:

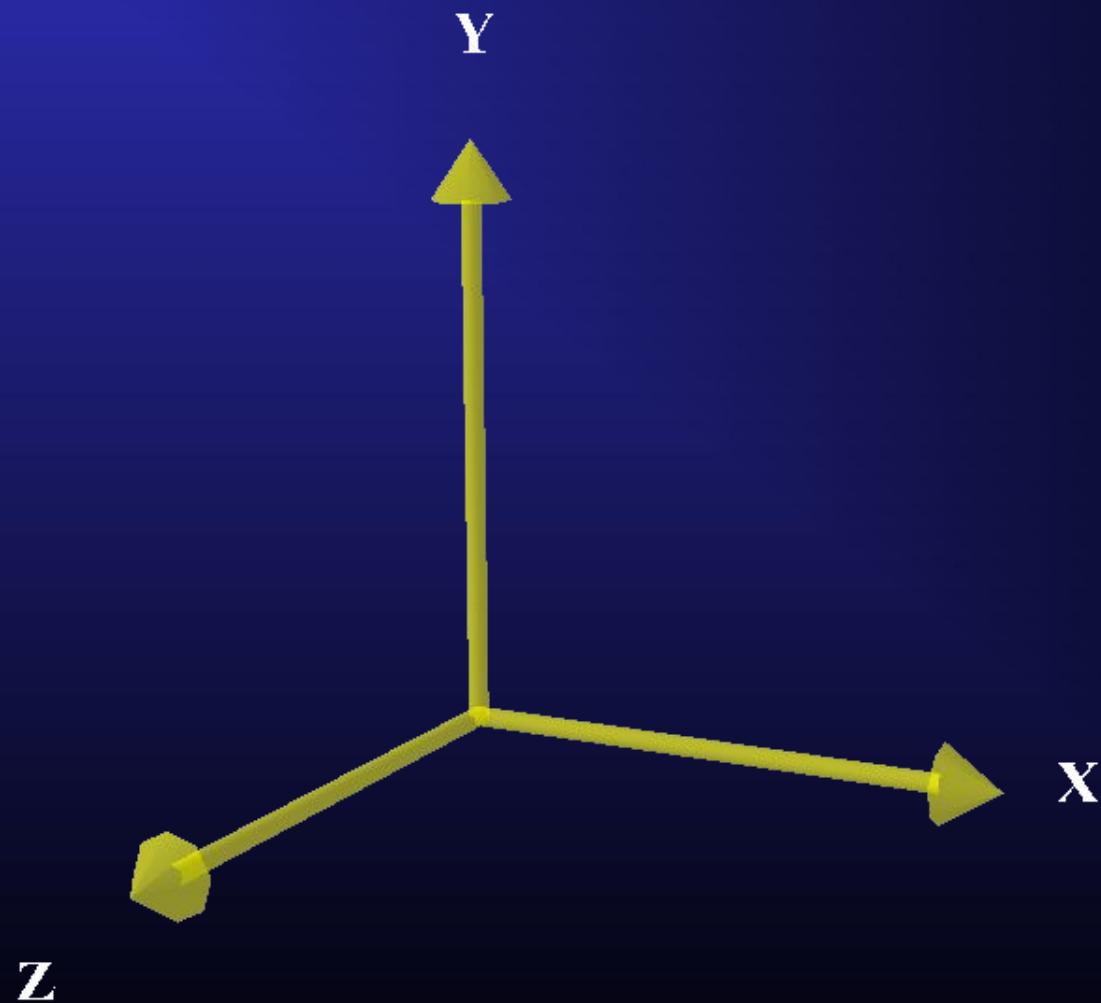
$$T_R R(\theta) T^{-1}_R$$

- And Translation can be to a point on the (moving) body

$T_{\text{attachment}}$



3D Transformations



Right handed
coordinate system



3D Transformations with Homogeneous Coordinates

$$\begin{bmatrix} X' \\ Y' \\ Z' \\ W' \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}$$

$$x = X' / W'$$

$$y = Y' / W'$$

$$z = Z' / W'$$

- If (x, y, z) is a real 3D point, $W' \neq 0$.
- 3D vector has homogeneous coordinate $(W) = 0$. (Can also think of this as the “point” in this direction infinitely far from the origin.)
- Uniform representation for all common transformations.
- Easy to manipulate with matrix algebra.

Scaling and Translation Transformation Matrices

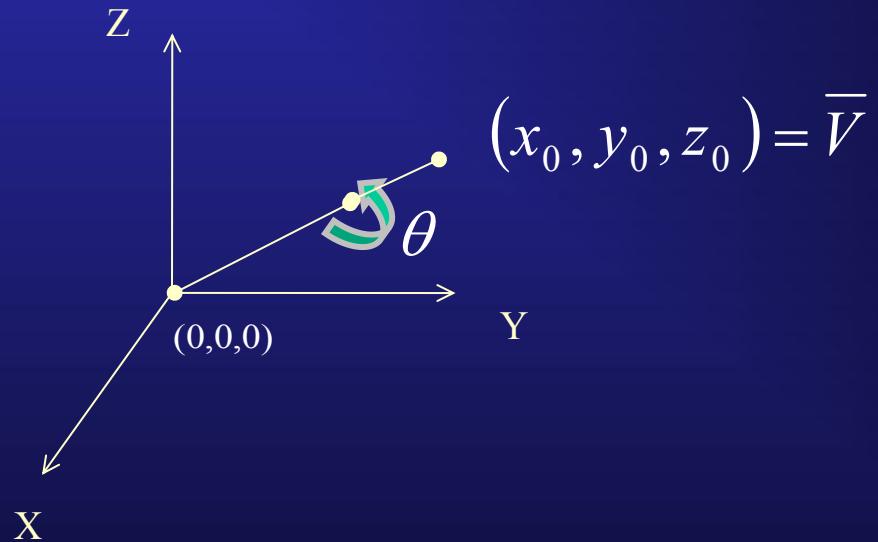
Scaling

$$\begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation about an Axis Through the Origin



Length of $\bar{V} =$

$$\|\bar{V}\| = \sqrt{x_0^2 + y_0^2 + z_0^2}$$

a, b, c are direction cosines: $a = \frac{x_0}{\|\bar{V}\|}$ $b = \frac{y_0}{\|\bar{V}\|}$ $c = \frac{z_0}{\|\bar{V}\|}$

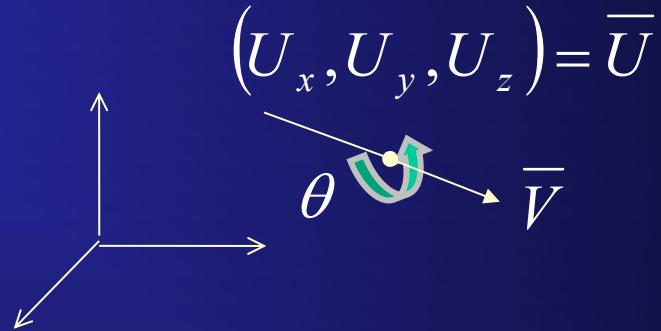
General Rotation Transformation Matrix

$$\begin{bmatrix} a^2 + (1 - a^2)\cos\theta & ab(1 - \cos\theta) - c\sin\theta & ac(1 - \cos\theta) + b\sin\theta & 0 \\ ab(1 - \cos\theta) + c\sin\theta & b^2 + (1 - b^2)\cos\theta & bc(1 - \cos\theta) - a\sin\theta & 0 \\ ac(1 - \cos\theta) - b\sin\theta & bc(1 - \cos\theta) + a\sin\theta & c^2 + (1 - c^2)\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where does this come from?

It is derived from a *quaternion* representation of 3D rotation.

What if Axis does not go through Origin?

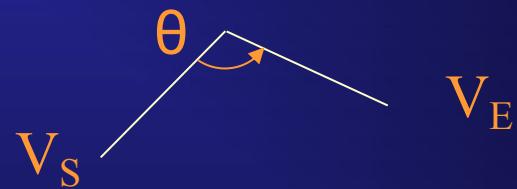


1. Translate to origin: $T(-U_x, -U_y, -U_z) = T(-\bar{U})$
2. Do rotation: $R(\theta)$
3. Translate back: $T(U_x, U_y, U_z) = T(\bar{U})$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = T(\bar{U}) R(\theta) T(-\bar{U}) \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

A Useful Role for the Cross Product in 3D Rotations

- Often we know the starting \mathbf{V}_S and ending \mathbf{V}_E position for some vector, but not the rotation axis \mathbf{V}_R nor the angle of rotation θ .



- Use the cross product $\mathbf{V}_S \times \mathbf{V}_E$ to find the axis of rotation \mathbf{V}_R perpendicular to the plane containing the starting and ending vectors. (If this cross product is 0, then either no rotation is needed ($\mathbf{V}_S = \mathbf{V}_E$), or else the rotation is by π . In the latter case, there are an infinite number of possible rotation axes.)
- Use the dot product of the vectors to find the rotation angle: $\theta = \arccos[(\mathbf{V}_S \cdot \mathbf{V}_E) / (\|\mathbf{V}_S\| \|\mathbf{V}_E\|)]$.

A Note on Repeating Transformations

- During animation, the same transformation may be applied repeatedly to hit rotation keyframes, e.g.
- In general, to avoid any chance of floating point error accumulation (and to end up where you expect to be), it is best to re-compute the transformation matrix rather than multiply by the incremental change.
- Let D = desired rotation angle; n = number of rotation steps; and $\Delta = D/n$
- Mathematically (but not computationally!)
$$\text{Rotation}(D) = \text{Rotation}(\Delta) \text{ Rotation}(\Delta) \dots \text{Rotation}(\Delta)$$

(n times)

- But this magnifies any round-off error.

Visualizing this...

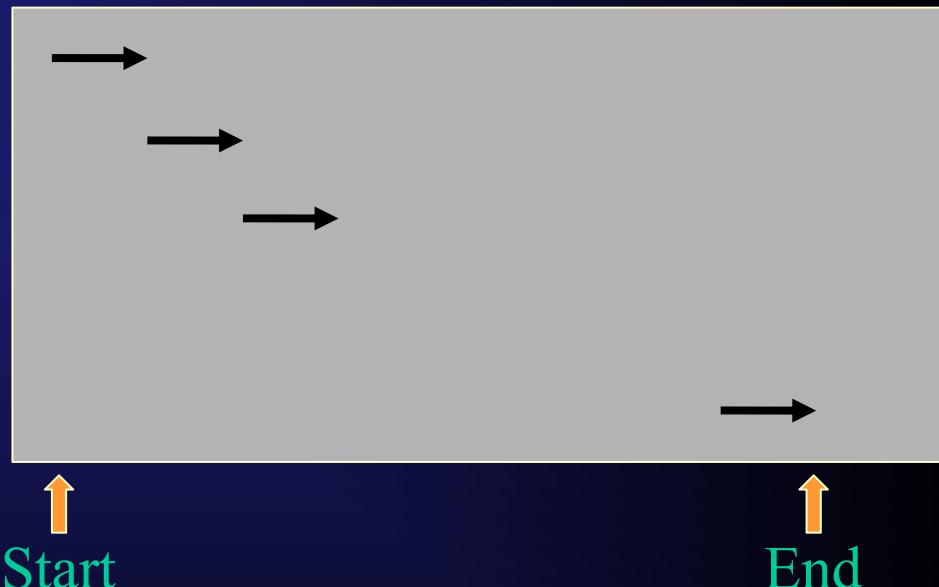
Rotation(Δ)

Rotation(Δ)

Rotation(Δ)

...

Rotation(Δ)



Better Solution

- To avoid this, compute instead the sequence of transformations:

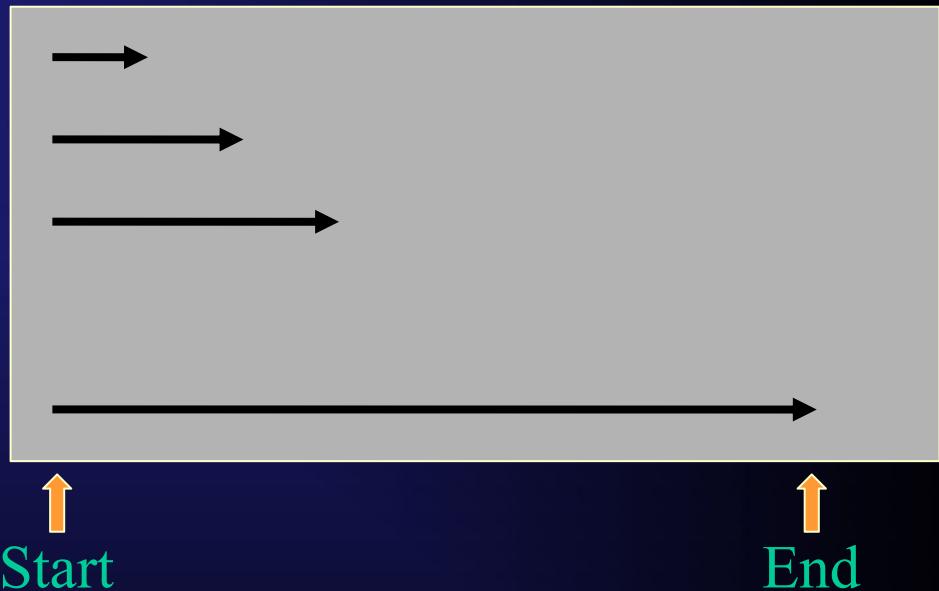
Rotation(1Δ)

Rotation(2Δ)

Rotation(3Δ)

...

Rotation($n\Delta$) = Rotation(D)

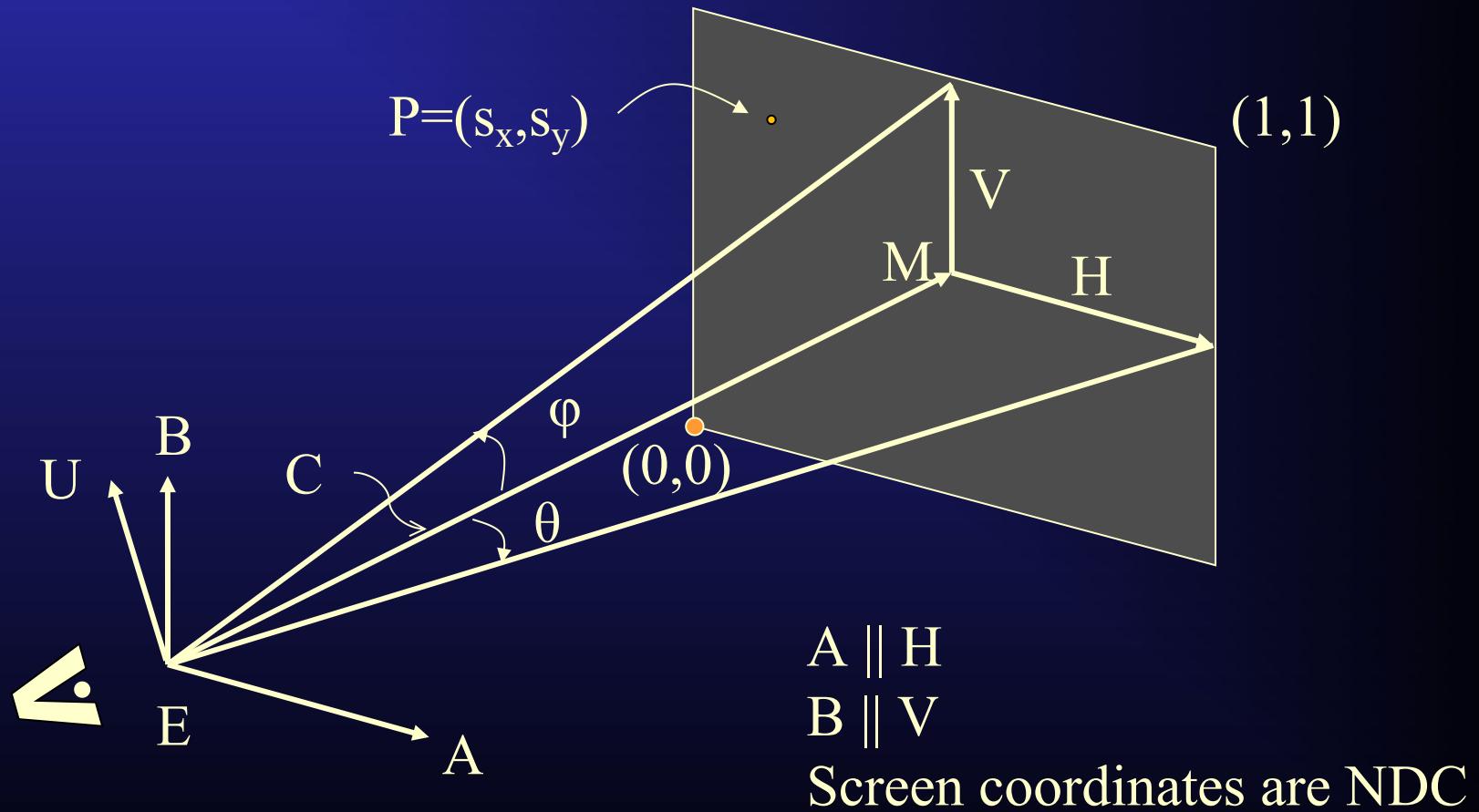


Yes, it takes more computation to compute the additional matrices and this has to be balanced against the need to be exactly where you want to be at the end. If it doesn't matter (e.g., just a turntable move), then use the single matrix approach.

Viewing Geometry for a Simple Ray Generator from a 3D Eye Point

- Avoids OpenGL projection setup.
- (Thanks to Andrew Glassner)
- Geometric Situation:
- Know:
 - Eye world coordinates: E
 - Viewing vector direction: C (implies viewing distance is $|C|$)
 - Up vector: U
 - Field of view half angles: θ and φ
- Output is a screen midpoint and two vectors used to iterate over the raster positions on the target NDC space.
- We'll compute the world coordinates of any point P in the NDC space.

Simple Viewing Geometry



Algorithm for Required Vectors and P

- $A \leftarrow C \times U$ (be sure this is well defined)
- $B \leftarrow A \times C$ (B is now in the correct plane)
- $M \leftarrow E + C$ (now have midpoint of screen)
- $H \leftarrow (A |C| \tan \theta) / |A|$ (rescale A to H)
- $V \leftarrow (B |C| \tan \varphi) / |B|$ (rescale B to V)

Given the origin M as shown, any point in NDC is (s_x, s_y)
[$0 \leq s_x \leq 1; 0 \leq s_y \leq 1$])

E.g., if frame buffer is 320 pixels wide by 240 pixels high,
then (50, 75) would map to $(s_x, s_y) = (50/319, 75/239)$.

$$P \leftarrow M + (2s_x - 1) H + (2s_y - 1) V$$

The ray equation is thus:

$$R \leftarrow E + t(P - E) / |P - E|$$

- (ray direction D is normalized $P - E$)
- So just iterate this for each pixel (integer) coordinate pair.

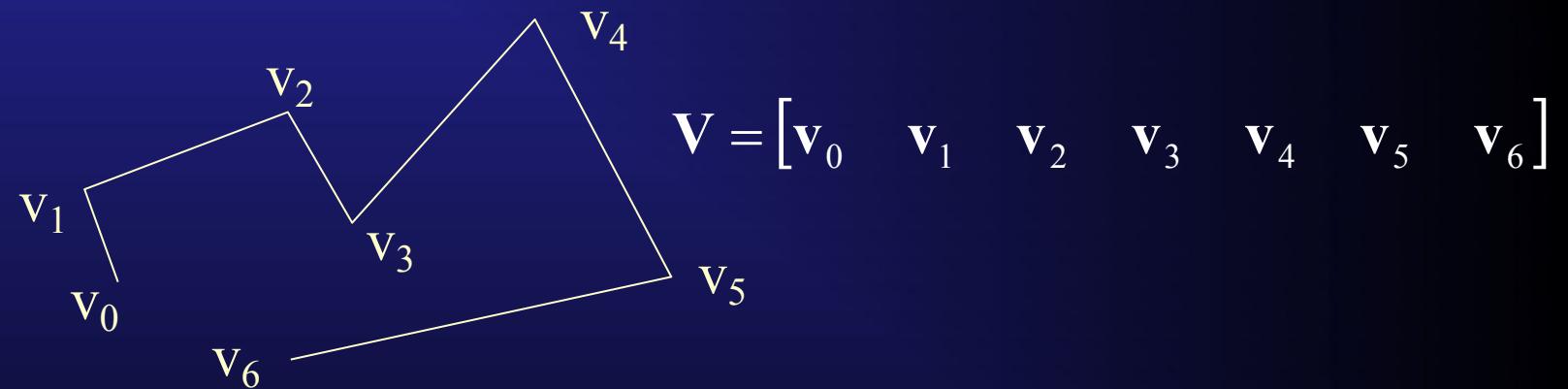
Defining “Objects”

- We need to model 2D and 3D objects via geometry.
- Will look at a variety of modeling methods later.
- Assume objects are built from polygons, so that they are defined by ordered lists of vertices.
- We can define transformations that change an object’s shape or position in space: e.g., translation, rotation, scaling.*
- The transformations are effected by simple operations on the vertex 2D or 3D homogeneous coordinates.

* These transformations are sometimes referred to as **affine**: transformations that preserve **collinearity** and **proportions**. There are interesting non-affine transformations such as **deformations** and **perspective**, that we will see later.

Beginnings of a Geometric Data Structure

- We can use our mathematical formulation of transformations and points to build a simple data structure for piecewise linear shapes (called *polylines*):



- By defining transformation matrices, concatenating them, and multiplying MV, we can use transformation matrices to manipulate geometric shapes composed of line segments.

Simple Polyline Representation

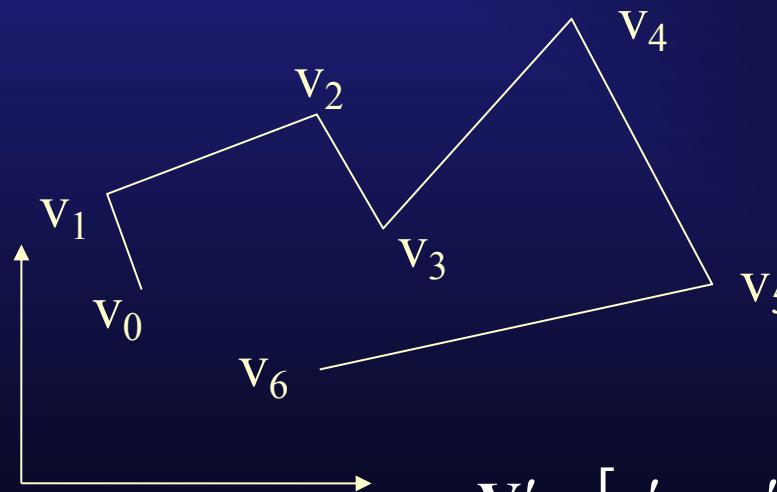
- For 3D data:

$$\mathbf{V} = \begin{bmatrix} \mathbf{v}_0 & \mathbf{v}_1 & \mathbf{v}_2 & \mathbf{v}_3 & \mathbf{v}_4 & \mathbf{v}_5 & \mathbf{v}_6 \end{bmatrix}$$
$$= \begin{bmatrix} x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\ y_0 & y_1 & y_2 & y_3 & y_4 & y_5 & y_6 \\ z_0 & z_1 & z_2 & z_3 & z_4 & z_5 & z_6 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{matrix} \leftarrow \text{Row 0} \\ \leftarrow \text{Row 1} \\ \leftarrow \text{Row 2} \\ \leftarrow \text{Row 3} \end{matrix}$$

- Note that array indices in C++ start at 0.
- The homogeneous coordinate in the last row is always 1 since these are points.

Transformations of Polyline Geometric Data

- We can transform the entire polyline V by matrix multiplication: e.g. suppose we want to rotate V (in xy plane) by 30° about point v_3 . The necessary transformation is $\mathbf{M} = T(v_3)R(30^\circ)T(-v_3)$.


$$\mathbf{V}' = \begin{bmatrix} v'_0 & v'_1 & v'_2 & v'_3 & v'_4 & v'_5 & v'_6 \end{bmatrix}$$
$$= \mathbf{M} \begin{bmatrix} v_0 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \end{bmatrix}$$

Polygons

- The points defining a polyline will be called *vertices*.
- One such point is called a *vertex*.
- A polyline is thus represented as an ordered list of vertices. A polyline whose starting vertex and ending vertex are identical is a *polygon*. (Graphically, if not mathematically; the difference will come out shortly.)
- In order to avoid representational ambiguities, if we use this matrix notation for a polygon, we will explicitly list the **first** and **last** vertex. [My convention!] That way we can check if a polyline is a polygon by checking if the first and last vertices are the same (equal coordinate tuples).

Simple Polygon Representation

- For an n vertex polygon \mathbf{P} we have an $n + 1$ column by 3 (if 2D) or 4 (if 3D) row array indexed $[0 .. 2 \text{ (or } 3\text{)}, 0 .. n+1]$:

$$\begin{aligned}\mathbf{P} &= [\mathbf{v}_0 \quad \mathbf{v}_1 \quad \mathbf{v}_2 \quad \cdots \quad \mathbf{v}_{n-2} \quad \mathbf{v}_{n-1} \quad \mathbf{v}_0] \\ &= \begin{bmatrix} x_0 & x_1 & x_2 & \cdots & x_{n-2} & x_{n-1} & x_0 \\ y_0 & y_1 & y_2 & \cdots & y_{n-2} & y_{n-1} & y_0 \\ z_0 & z_1 & z_2 & \cdots & z_{n-2} & z_{n-1} & z_0 \\ 1 & 1 & 1 & \cdots & 1 & 1 & 1 \end{bmatrix}\end{aligned}$$

- For example, a triangle is:

$$\begin{aligned}\Delta &= [\mathbf{v}_0 \quad \mathbf{v}_1 \quad \mathbf{v}_2 \quad \mathbf{v}_0] \\ &= \begin{bmatrix} x_0 & x_1 & x_2 & x_0 \\ y_0 & y_1 & y_2 & y_0 \\ z_0 & z_1 & z_2 & z_0 \\ 1 & 1 & 1 & 1 \end{bmatrix}\end{aligned}$$

Polygon Edges

- Note that polylines and polygons are defined by **a finite set of vertices, implicitly connected in order by straight line segments**. The polylines and polygons are NOT defined as “images”, pixels, or “dots inbetween” the vertices.
- The line segments inbetween the vertices are called *edges*.
- Q: So where do the edges come from?
- A: We must specifically *compute* the line segments that represent the edges if we want to draw or otherwise manipulate them. The matrix representation of the polyline or polygon only explicitly lists the ordered vertices, e.g.:

$$\mathbf{V} = \begin{bmatrix} \mathbf{v}_0 & \mathbf{v}_1 & \mathbf{v}_2 & \mathbf{v}_3 & \mathbf{v}_4 & \mathbf{v}_5 & \mathbf{v}_6 \end{bmatrix}$$

Drawing Polylines and Polygons

- Assume we can draw a single line segment on a display screen: let's just call it

DrawLineSegment(xstart, ystart, xend, yend)

where we'll limit ourselves to 2D for now.



- We can draw 2D polyline or polygon P with a simple loop:

```
for (i=0; i<P.cols()-1; i++)  
    DrawLineSegment(P[0,i], P[1,i], P[0,i+1], P[1,i+1])
```

(assuming P has at least two vertices)

Graphics Software (An instant history)

- Standards (motivated by device independence)
 - (Core: first 3D proposal; mostly vector-oriented)
 - (GKS: 2D, but included raster graphics workstations)
 - (PHIGS: added hierarchical display list)
 - VRML: 3D text-based graphics markup language
 - OpenGL: SGI's GL function library spec is opened
 - DirectX: Microsoft uses direct buffer writes
 - Cg: graphics board language
 - CUDA: Parallel C-like threads for GPU

Outline

- Great Ideas ✓
- Virtual Environments & Devices ✓
- 3D Perception ✓
- Architecture, Color & Rasterization ✓
- Vector Geometry & Transformations ✓
- **3D Modeling**
- Embedding, Hierarchies & Contours
- Model Generation & Deformation
- Visible Surface Algorithms
- Polygon Algorithms
- Image Synthesis & Shading
- Mapping
- 3D Viewing Transformations
- Anti-aliasing & Compositing
- Global Illumination
- Light Fields & HDR
- Path Tracing/Photon Mapping
- Futures

3D Object Modeling

- 3D modeling means defining or designing a geometric representation of some shape.
- Wide variety of 3D modeling techniques.
- 3D models apparently originated with Larry Roberts at MIT in early ‘60s:
 - 3D convex object models
 - 3D transformations in homogeneous coordinates
 - Visible line drawing

3D Modeling

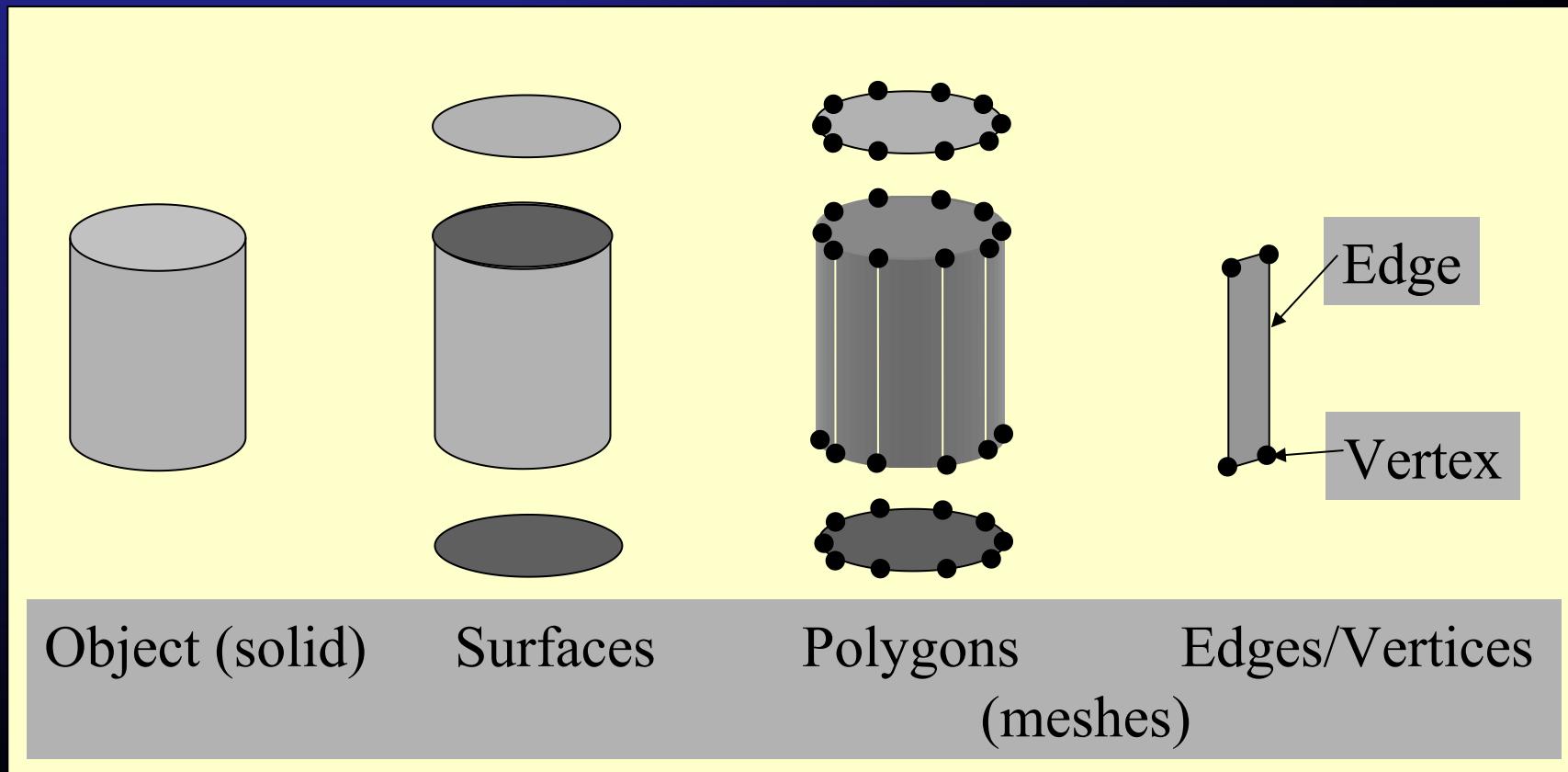
- What is 3D modeling?
- How are models used?
- What are the major issues?
- What's the difference between models and rendering?
- What are the operations appropriate to models?
- Representation structures
- Model classification
- Model design, authoring and synthesis

What is 3D Modeling?

- Computational models of real or imagined physical objects.
- Decomposition into primitives (the elements understood and used by a display or measurement algorithm).
- Organization in a computer (data structures or procedures).
- Model creation or synthesis (build it).
- Model manipulation (move it or modify its geometry or attributes).
- Model visualization and use (examine it).

Conceptualizing Models

- Geometric models can be viewed from several mathematical perspectives:



What are the Some Modeling Issues?

- Cost of the model
- Effectiveness
- Complexity
- Accuracy
- Conversion from one data representation to another

Cost of the Model

- In computer storage space
 - Number of primitives
 - Space per primitive
- In construction time
 - Data capture cost
 - Modification cost (digital “back lot”)
 - Availability of interactive and/or automated tools
- In display time
 - Display algorithm keyed to representation choice
 - Image quality parameters impact desired attributes of model

Cost of the Model

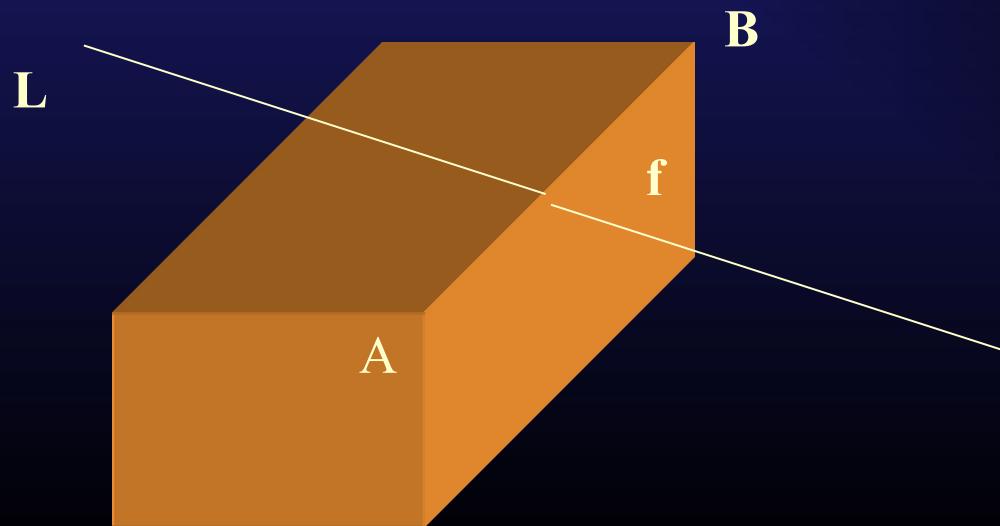
- In use (transformation, measurement) time
 - What is the principal use(s) of the model?
 - Complexity of geometric algorithms
 - Known limits (theoretically costlier algorithm may be more efficient)
- Implementation issues
 - Programmer time
 - Availability of existing software
 - Production (animation) vs. one-shot use

Complexity

- Multiplicity of simple forms
- Inherently complex form
 - Understand the nature of such complexity
 - Not usually chaotic, so must find underlying structures or regularities

Accuracy

- Numerical issues when doing real coordinate geometry with finite precision arithmetic.
- Use epsilon tests not “ $=0$ ” tests (as previously noted).
- Can have incompatible results to otherwise equivalent tests; e.g., does line L intersect edge AB and/or interior of face f ?
- Significant issue in rendering since a ray intersection may be *beneath* the surface due to floating point imprecision.



Conversions between Representations

- Required by given input data
- Required by certain operations
- Required by display algorithm
- For example:
 - Converting curved surfaces to planar polygons.
 - Tracking the surface of volume or implicit surface models (e.g., to obtain polygonal surfaces).
 - Filling interior of a surface model with volume elements.
 - Some almost impossible (e.g., boundary representation to CSG).

What's the Difference between Models and Rendering?

- Models describe object and its attributes.
- Rendering transforms the model to a viewable (2D) image.

Models Describe Object and its Attributes

- Shape / geometry
- Color
- Reflectivity
- Transmittance
- Surface smoothness
- Texture
- Hierarchy
- Shaders

Rendering Transforms the Model to a Viewable Image

- A center of projection or camera position.
- One or more light sources and atmospheric effects.
- A background or world projection.
- Algorithms to interpret the values and interactions of all the attributes.
- Image quality parameters.
 - Anti-aliasing.
 - Compositing.
 - Available color resolution.
 - Non-photorealistic (NPR) rendering
- Animation concerns.

What are the Operations Appropriate to Models?

- 3D Transformations
- Change in amount of detail
- Measurement
- Combination
- Deformation

Level of Detail

- The modeler's dilemma: Either have too little data, too much data, or limited time to author data.
- Interpolate to augment detail.
- Subdivide to improve detail from simpler input
- Decimate to reduce detail.
- Hierarchy of models of same object at different levels of detail: multi-resolution models.

Measurement

- Topology
 - Connectivity
 - Components
 - Holes
 - Surface consistency
- Point-to-point or surface-to-surface distance
- Surface area
- Volume
- Surface normals
 - Needed to display shaded surfaces
 - Blend two model parts smoothly

Combination

- Boolean operations
 - Union ($A \cup B$)
 - Intersection ($A \cap B$)
 - Difference ($A - B$)
 - Symmetric difference ($(A - B) \cup (B - A)$)
- Need regularized geometry: replace object by closure of its interior.
- Cut/slice (with plane or other object to remove parts, revealing interior)
- Cover (priority during display; e.g. picture on wall)

Deformation

- Shear
- Taper
- Stretch
- Bend
- Twist
- Free-form deformation (FFD)
- Cages
- Perturb (e.g., stochastically)
- Deform via physical models of materials and forces (see animation course!)

Representation Structures

- Data structures
 - The model is embedded in a conventional data structure
 - Linked lists
 - Arrays
 - Directed graphs (hierarchies)
 - (Out of core files)
- Procedural methods
 - The model is embedded into any convenient computational procedure, such as a formula, implicit equation, or other code.

Model Classification

- Volumetric Representations
- Boundary Representations
- Combinations and Hybrids

Volumetric Representations

- The 3D volume of the object is decomposed into (possibly overlapping) volumes
 - Voxels
 - Tetrahedral meshes
 - Constructive Solid Geometry (CSG)

Boundary Representations

- The surface of the object is approximated by or partitioned into (non-overlapping) 0-, 1-, or 2-dimensional primitives
 - Height fields
 - Polygon meshes
 - Fractals
 - Curved surfaces
 - Implicit surfaces
 - Superellipsoids
 - Potential functions, Metaballs

Combinations and Hybrids

- Particle systems
- Point set (meshless) geometry
- Embeddings and hierarchies
- Procedural deformation
- Free-Form Deformation (FFD)
- Model blending
- Multi-resolution surfaces

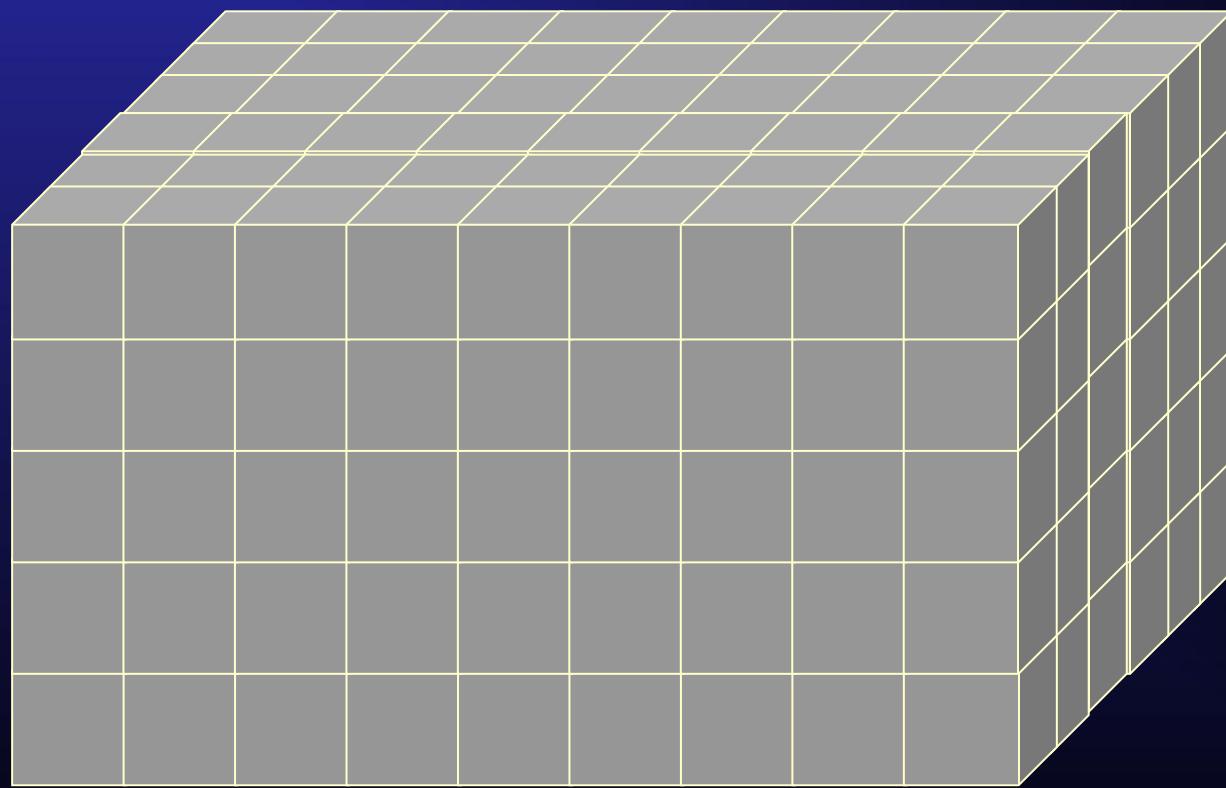
Volume and CSG Models

- Voxels
- Tetrahedral meshes
- Constructive Solid Geometry (CSG)

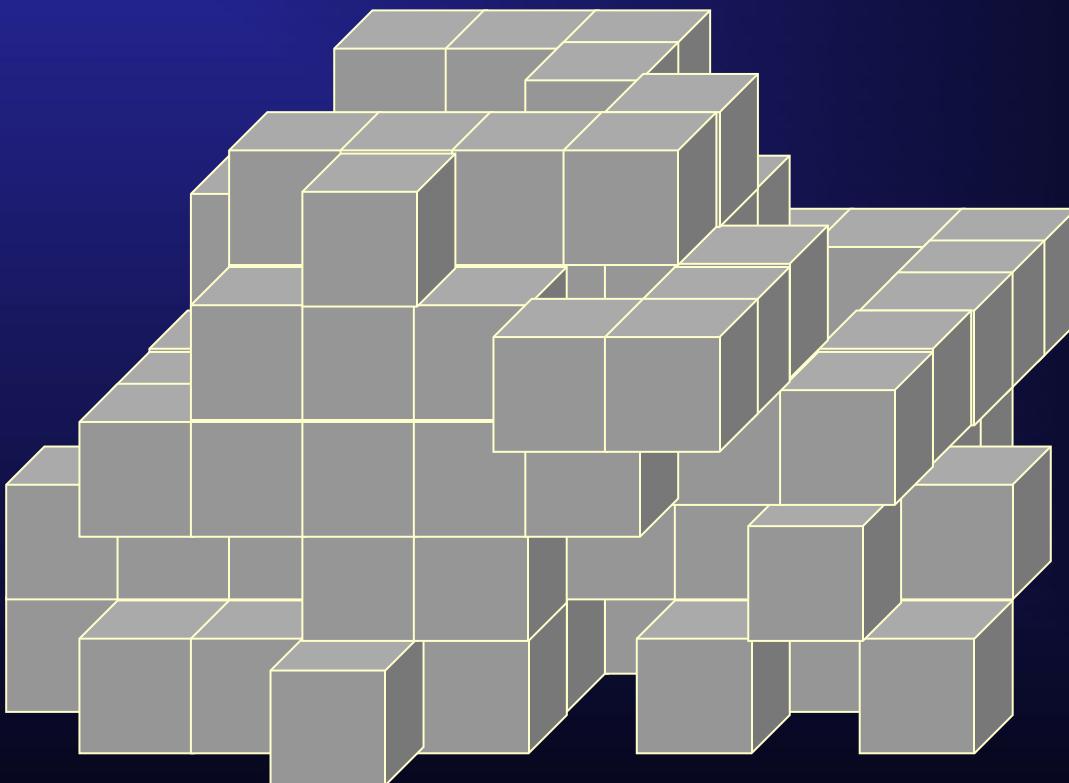
Voxels

- Also called **spatial subdivision**.
- Space-filling subdivision, usually with cubes or hexahedral (*parallelopiped*) elements.
- Density, value, or set of values associated with each voxel.
- High storage requirements but simple data structure.
- Special shading techniques needed to compute surface normals, gradients, and smoothing.
- Popular as volume visualization for scientific and medical applications.
- Hardware support attempted in early ‘80s; in late 90’s reappeared in single board volume rendering. Now pretty much replaced by faster CPUs and GPU methods.

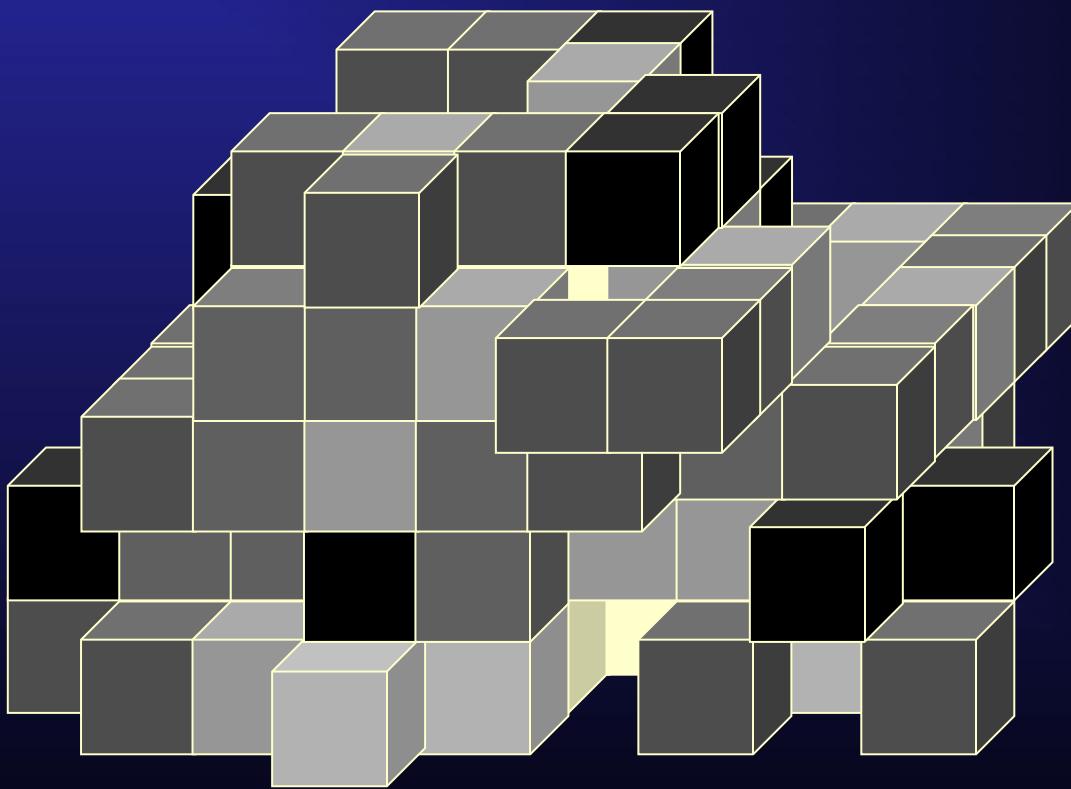
Voxel Structure



Voxel Structure (Binary Values)



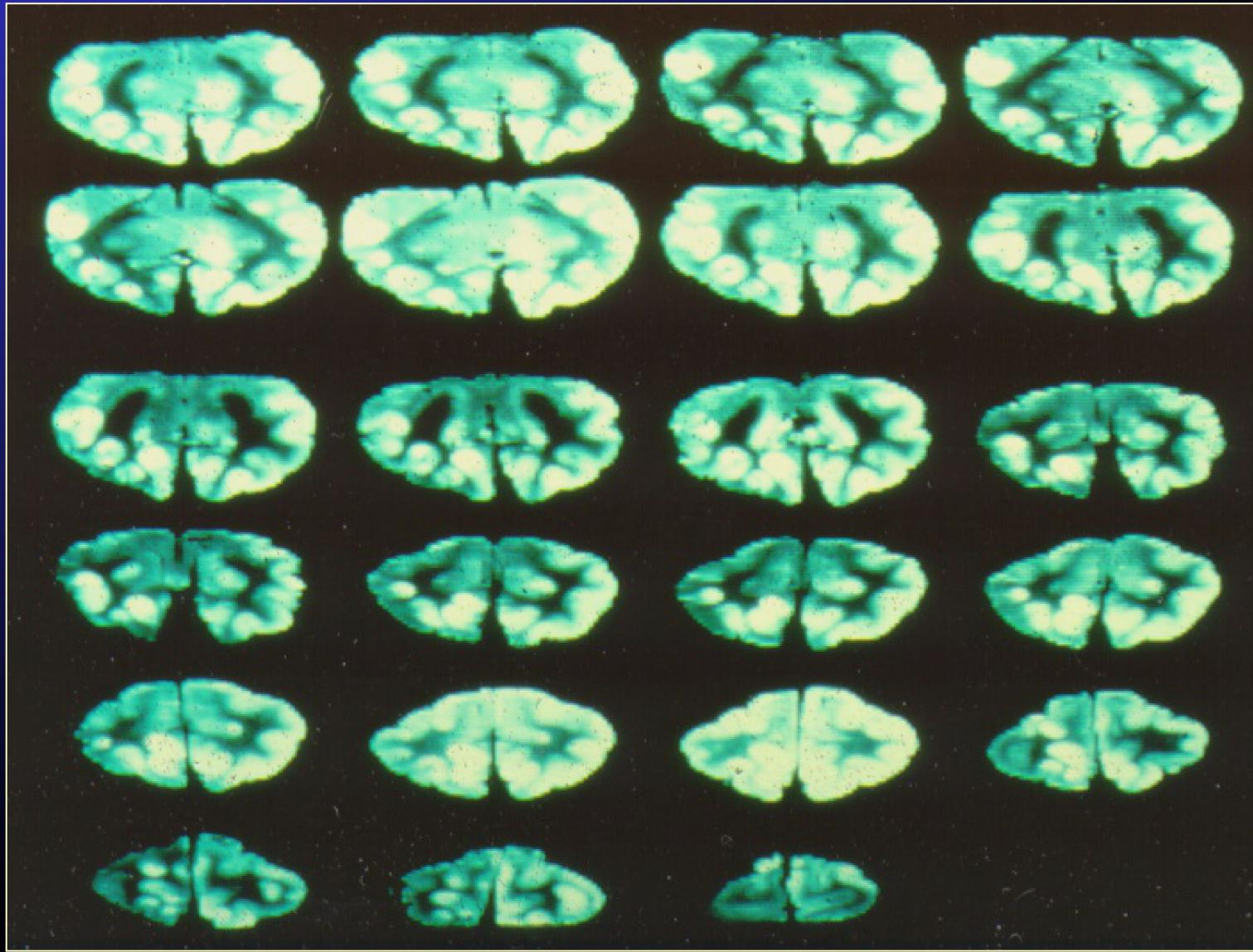
Voxel Structure (Scalar Density Values)



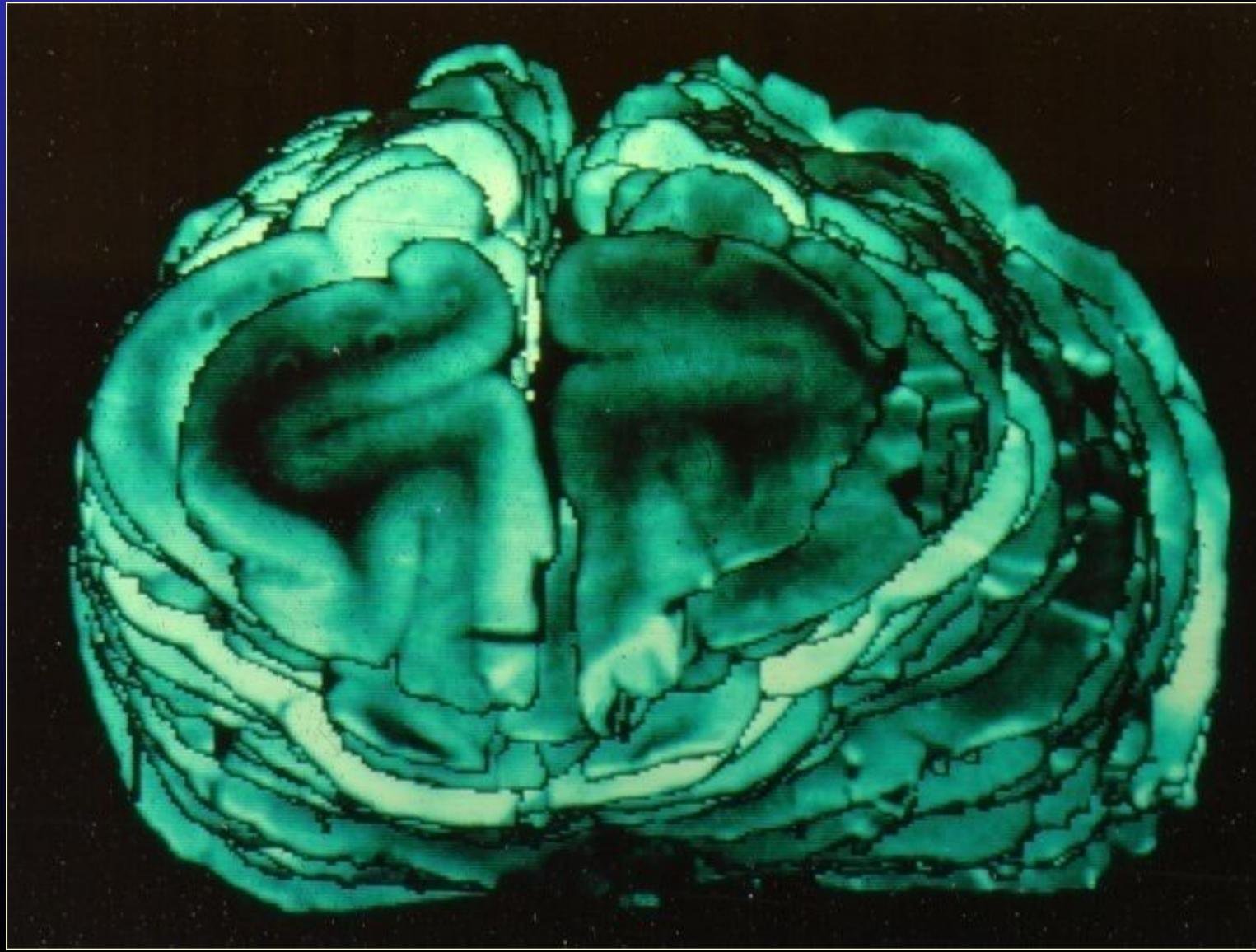
Generating Volumetric Data

- Image data (slices, e.g., medical data)
- Geometric “phantoms”
- Random functions (but not very meaningful)
- Controllable randomness: Perlin noise

Slices from Cat Brain



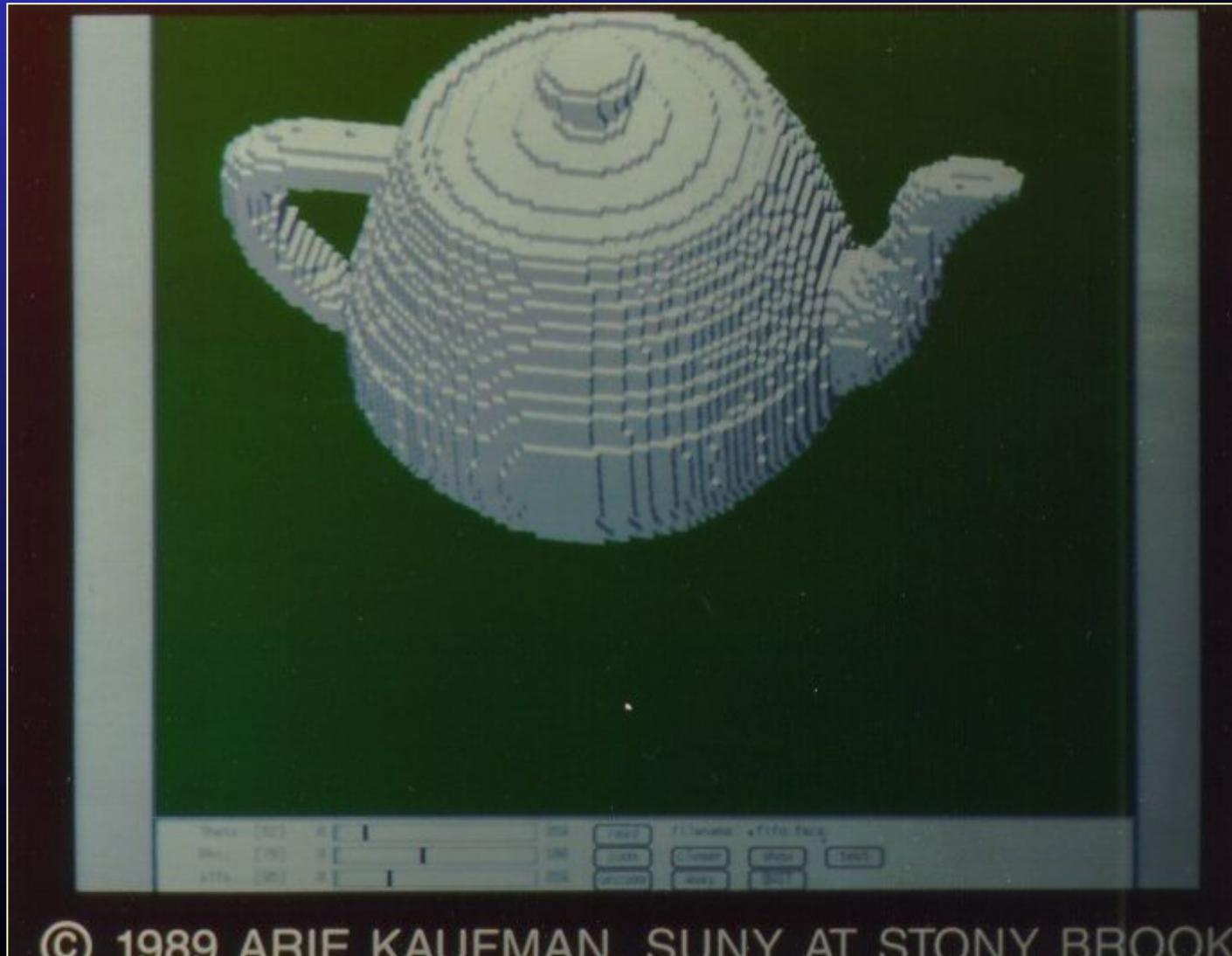
Boundary-Clipped and Stacked Slices



Surface Interpolation plus Depth Shading: Trying to Show Details – But Not Good Enough

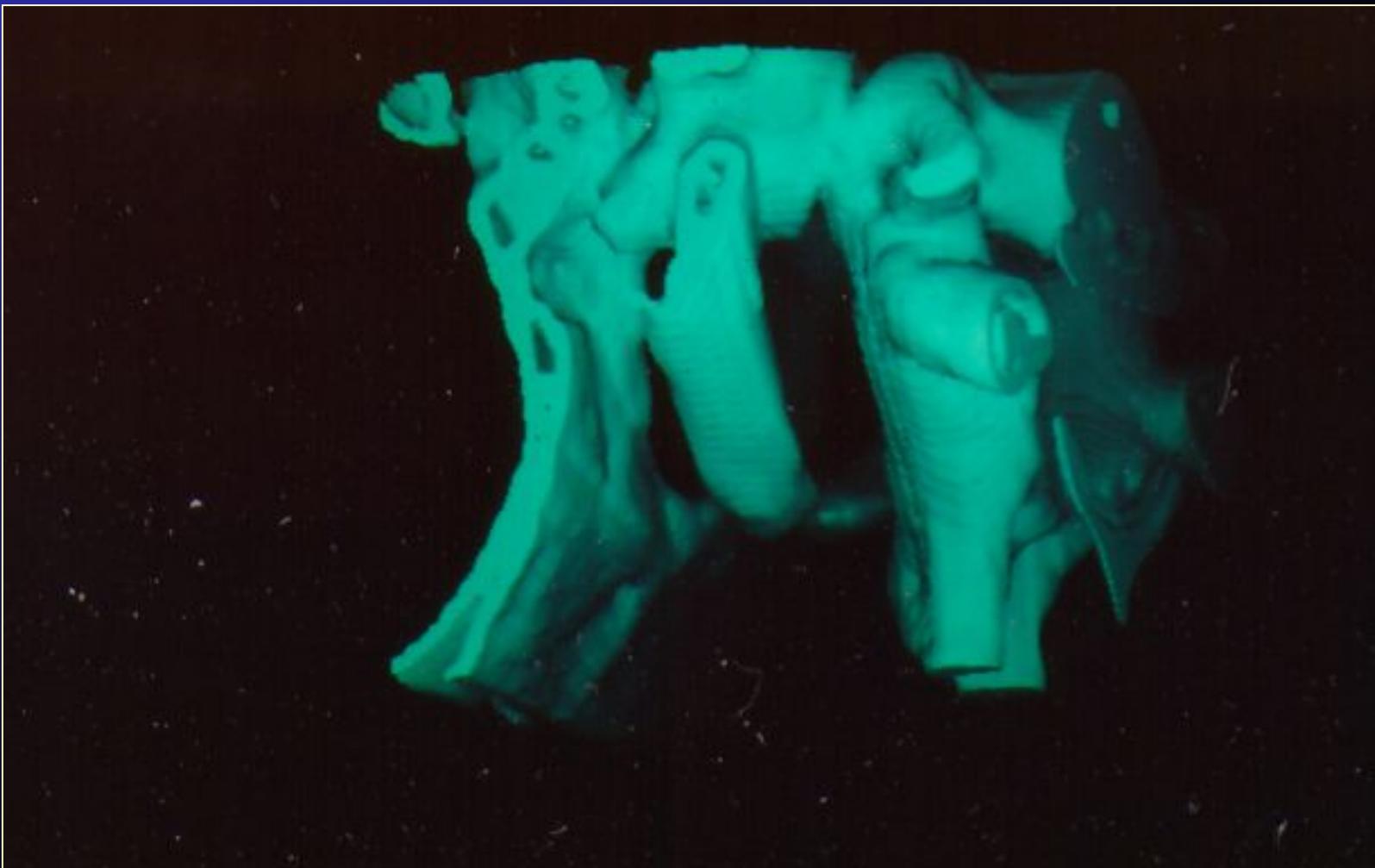


Geometric “Phantom”: E.g., a Teapot



© 1989 ARIE KAUFMAN, SUNY AT STONY BROOK

128^3 Voxel Space



DYNAMIC DIGITAL DISPLAYS INC. D. TALTON

If Voxels Represent Surfaces, Need to Approximate Surface Normal for Shading

- Let voxel widths be h_x , h_y , and h_z .
- Voxel value (e.g., density scalar) at $P_{i,j,k} = v(P_{ijk})$.
- Compute normal via local gradient in each dimension by central differences: at voxel $P_{i,j,k}$ it is:

$$Normal(P_{i,j,k}) = \nabla v = \left(\frac{\partial v}{\partial x}, \frac{\partial v}{\partial y}, \frac{\partial v}{\partial z} \right)$$

$$\frac{\partial v}{\partial x} = g_x(P_{i,j,k}) \approx \frac{v(P_{i+1,j,k}) - v(P_{i-1,j,k})}{2h_x}$$

$$\frac{\partial v}{\partial y} = g_y(P_{i,j,k}) \approx \frac{v(P_{i,j+1,k}) - v(P_{i,j-1,k})}{2h_y}$$

$$\frac{\partial v}{\partial z} = g_z(P_{i,j,k}) \approx \frac{v(P_{i,j,k+1}) - v(P_{i,j,k-1})}{2h_z}$$

+Z

| | | |
|----------------|----------------|----------------|
| 0.1 (2,0,2) | 0.3 (2,1,2) | 0.6 (2,2,2) |
| 0.4 (2,0,1) | 0.6 (2,1,1) | 0.5 (2,2,1) |
| 0.4 (2,0,0) | 0.8 (2,1,0) | 0.9 (2,2,0) |

+Y

+X

Example

- Assume $h_x=h_y=h_z=1$
- Assume [unseen] $v(P_{1,1,1}) = 0.6$
- Assume all other $v(P_{i,j,k}) = 0$ for $i<0, i>2; j<0, j>2; k<0, k>2$

$$Normal(P_{i,j,k}) = \nabla v = \left(\frac{\partial v}{\partial x}, \frac{\partial v}{\partial y}, \frac{\partial v}{\partial z} \right)$$

$$\frac{\partial v}{\partial x} = g_x(P_{2,1,1}) \approx \frac{v(P_{3,1,1}) - v(P_{1,1,1})}{2} = \frac{0 - 0.6}{2} = -0.3$$

$$\frac{\partial v}{\partial y} = g_y(P_{2,1,1}) \approx \frac{v(P_{2,2,1}) - v(P_{2,0,1})}{2} = \frac{0.5 - 0.4}{2} = 0.05$$

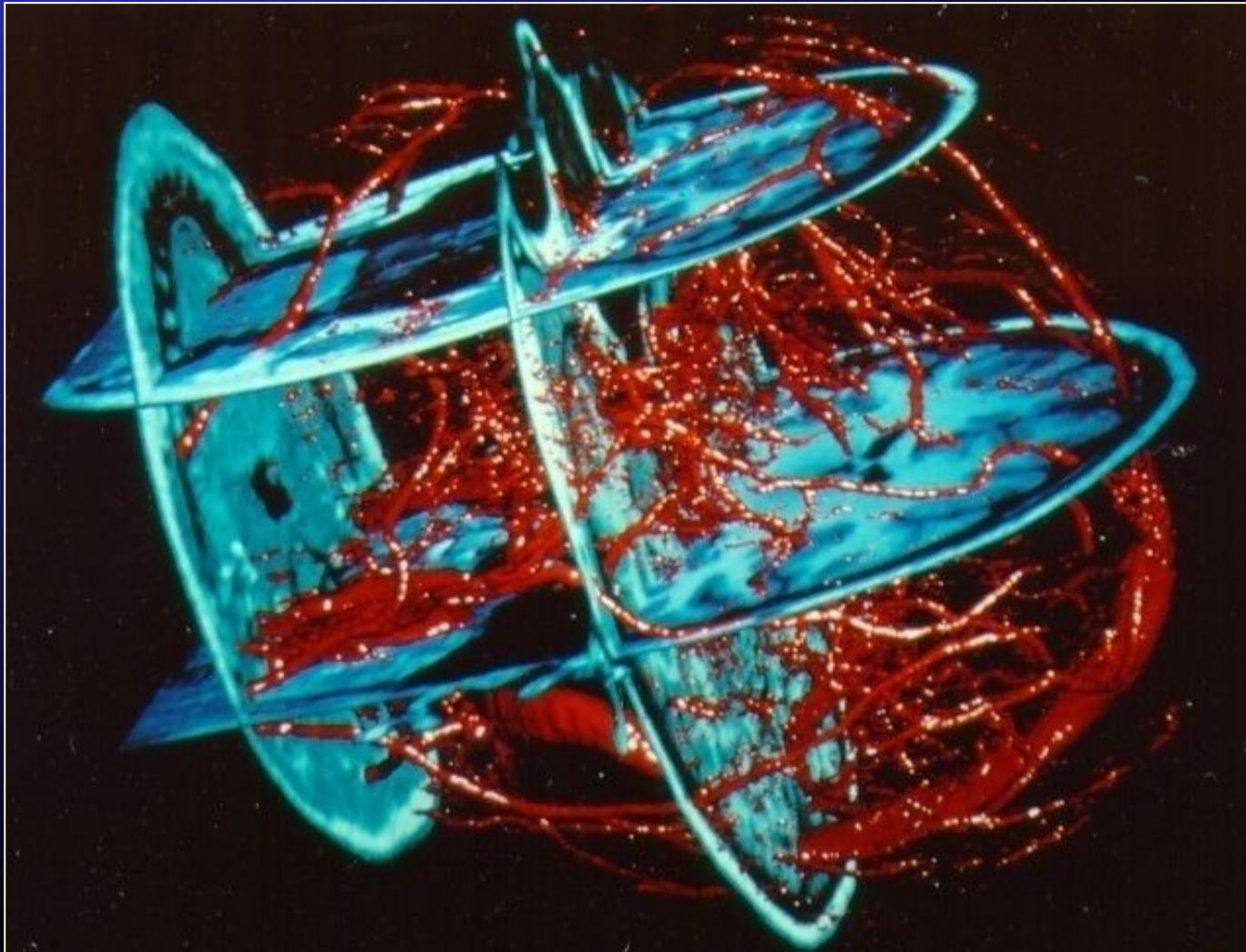
$$\frac{\partial v}{\partial z} = g_z(P_{2,1,1}) \approx \frac{v(P_{2,1,2}) - v(P_{2,1,0})}{2} = \frac{0.3 - 0.8}{2} = -0.25$$

Visible Man and Woman



©1998, CleMed, NUS, & Intel Corp.

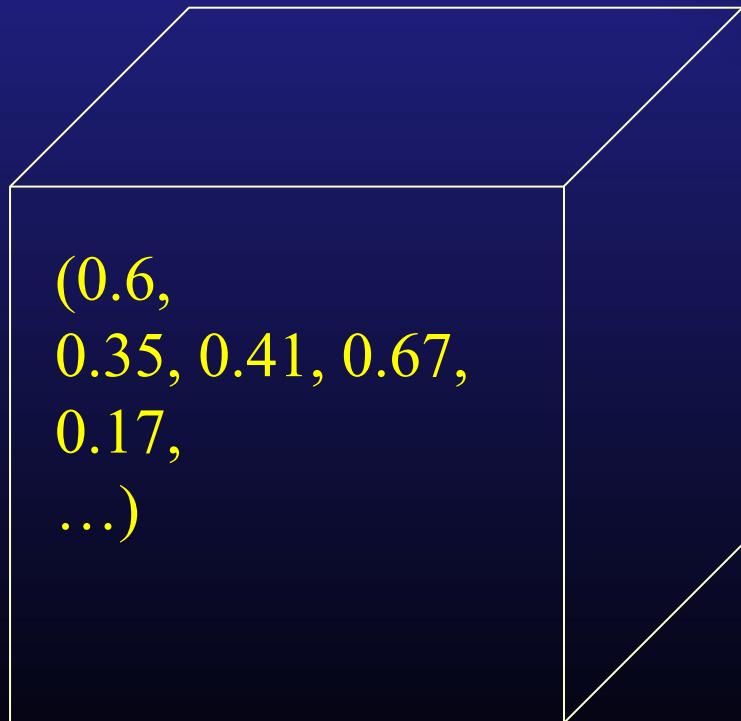
Continuity Troubles



© 1992 IMDM UNIVERSITY OF HAMBURG

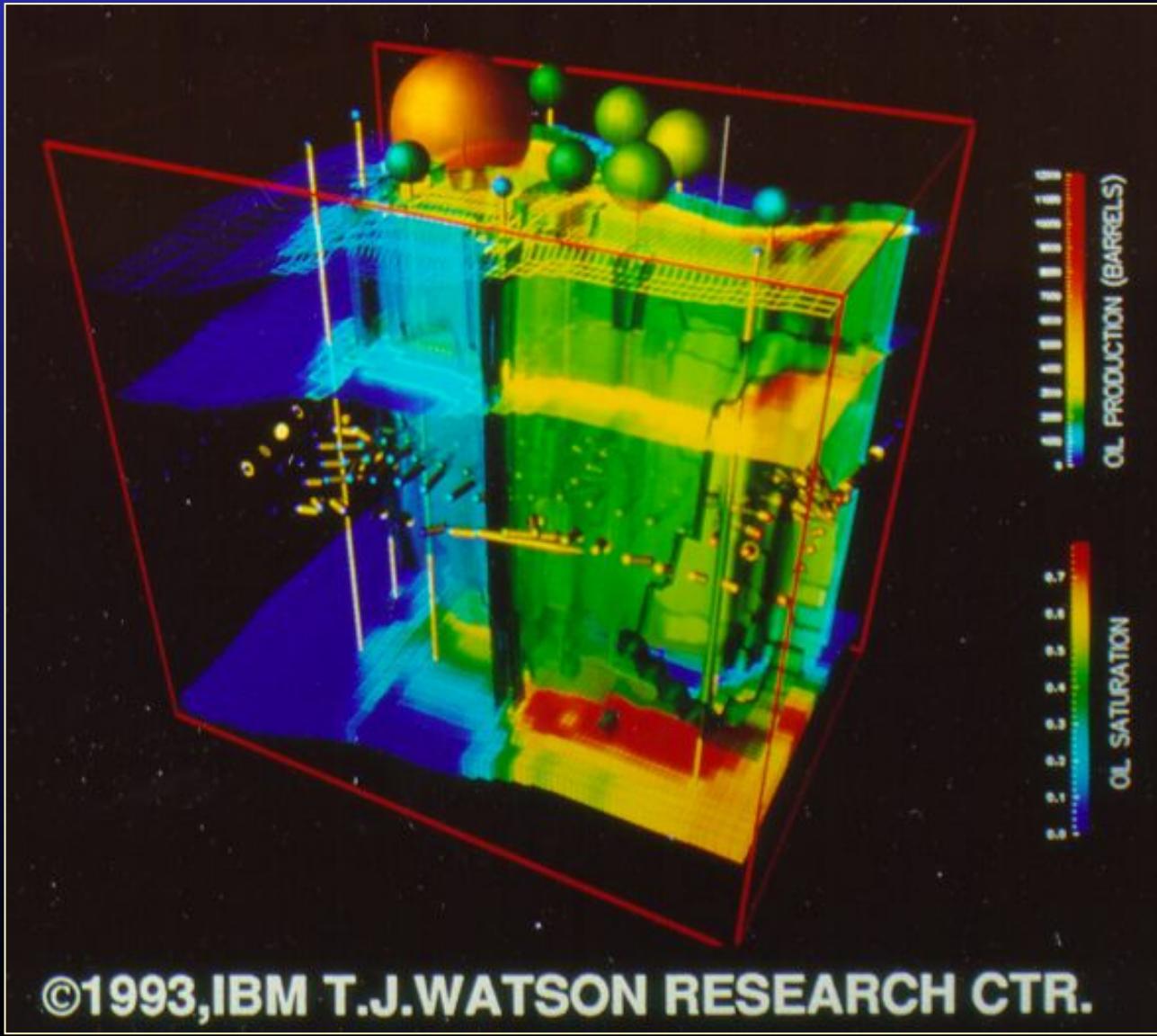
Voxel Contents

Each voxel can even have a vector of values...

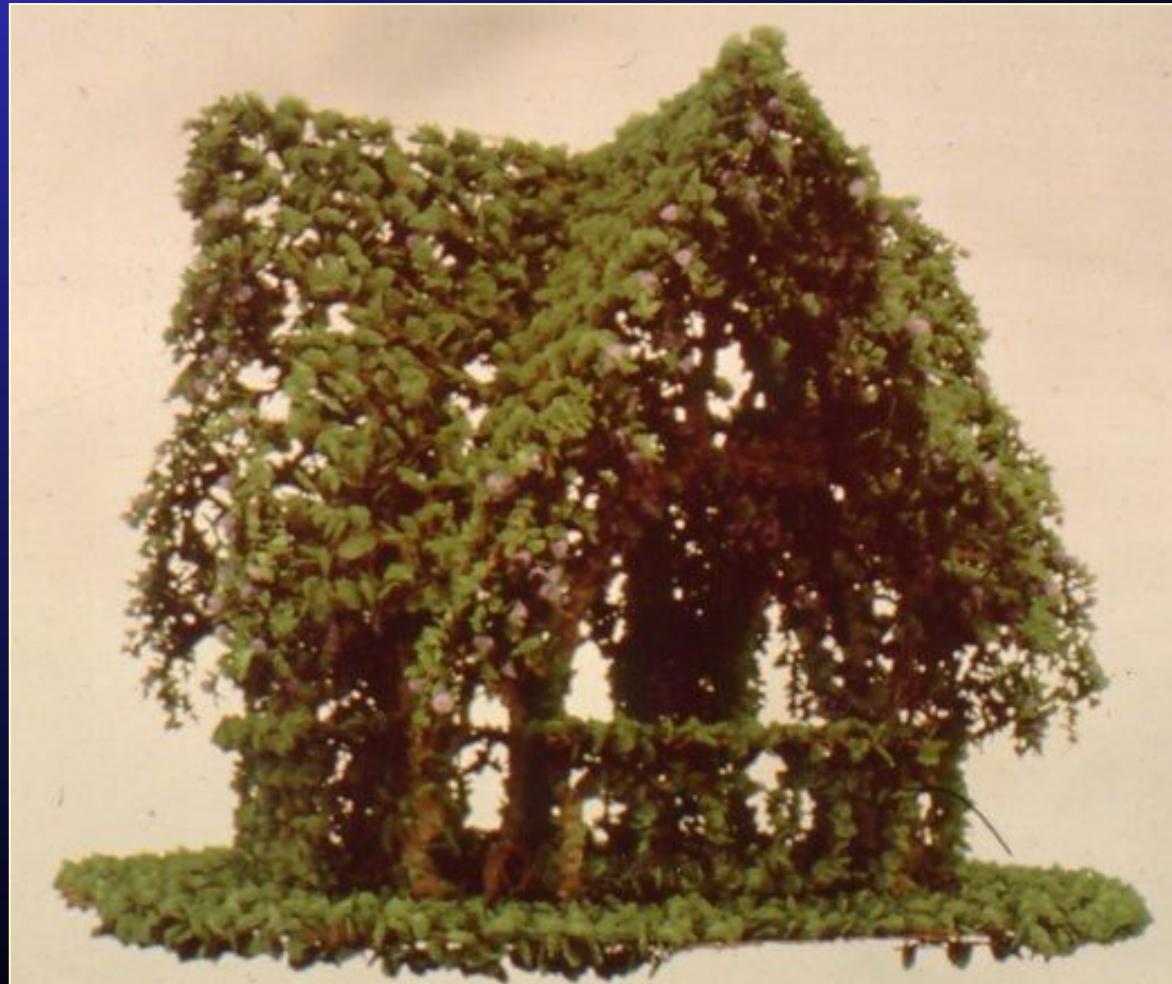


(Density,
Direction (x,y,z),
Velocity,
Curvature,
...)

3D Geological Data



Spatial Subdivision as Guide/Constraint for Other Geometry (Where it can or cannot be)



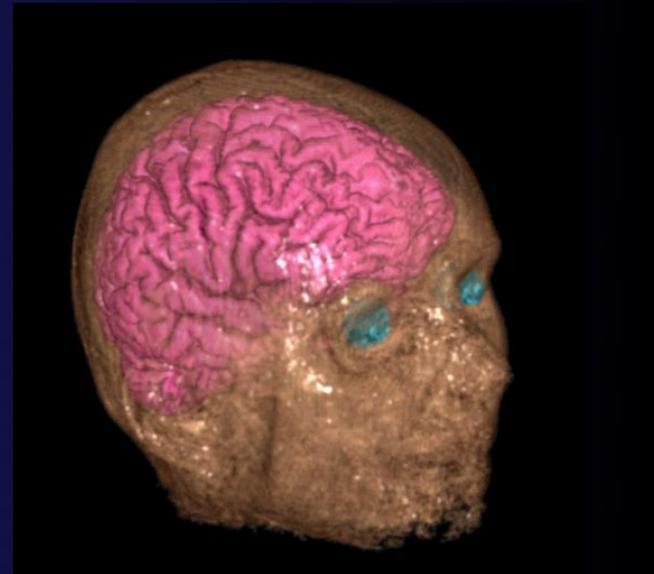
A Diversion into Volume Rendering (So you can do some Homework!)



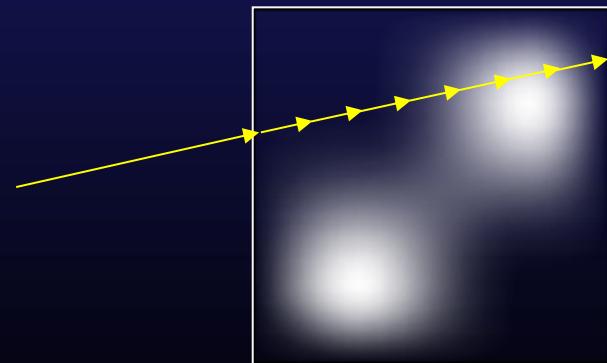
Zakiuddin Mohammed, CIS 560 Fall 2011

General Case: Volume Rendering

- Let's see how we might render the contents of a voxel space as a graphics image.
- IDEA: Cast rays from eye into volume and integrate density along intersected cells.
- Simple method: just sample at regular intervals in the volume and use a transmittance formula.

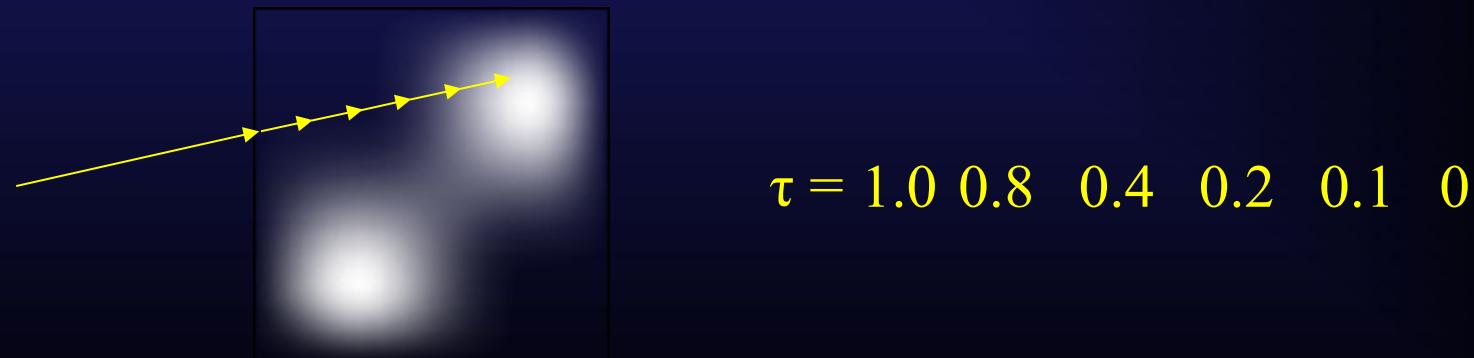


Christof Rezk-Salama



Transmittance Along the Ray

- *Transmittance* τ is 1 if clear and 0 if opaque.
- Transmittance of samples can be computed back-to-front or front-to-back as the ray is traversed in the volume.
- Advantage of the front-to-back scheme is that it can stop sampling when the decremented transmittance becomes within some tiny threshold of 0 (the volume is essentially opaque as far as the ray has penetrated).



Volume Rendering

- Notation: $\mathbf{x} = (x, y, z)$; $c = (r, g, b)$
- Eye at \mathbf{x}_e looking in direction \mathbf{n}
- Pixel color C based on color accumulated along ray $\mathbf{r} = \mathbf{x}_e + \mathbf{n}s$
- Accumulated color has form

$$C(\mathbf{x}_e, \mathbf{n}) = \int_0^\infty ds \bar{G}(\mathbf{x}_e, \mathbf{n}, s)$$

- where

$$\bar{G}(\mathbf{x}_e, \mathbf{n}, s) = c(\mathbf{x}_e + \mathbf{n}s) \rho(\mathbf{x}_e + \mathbf{n}s) T(\mathbf{x}_e, \mathbf{n}, s)$$

color density accum. transmittance

- and

$$T(\mathbf{x}_e, \mathbf{n}, s) = \exp\left\{-\kappa \int_0^s ds' \rho(\mathbf{x}_e + \mathbf{n}s')\right\}$$

[Beer's Law]

Density Rendering Summary (no external lights)

- Combined:

$$C(\mathbf{x}_e, \mathbf{n}) = \int_0^\infty ds \rho(\mathbf{x}_e + \mathbf{n}s) c(\mathbf{x}_e + \mathbf{n}s) \exp\left\{-\kappa \int_0^s ds' \rho(\mathbf{x}_e + \mathbf{n}s')\right\}$$

- Note that since we will choose a finite step size Δs , these integrals are all discrete sample sums.

$$\mathbf{x}_i = \mathbf{x}_e + \mathbf{n} i \Delta s, \quad i = 0, \dots, \infty$$

- So: $C(\mathbf{x}_e, \mathbf{n}) = \sum_{i=0}^{\infty} \Delta s \rho(\mathbf{x}_i) c(\mathbf{x}_i) T(\mathbf{x}_e, \mathbf{n}, i \Delta s)$
- We really don't march to ∞ ; we only need to iterate until this suitably converges or we exit the represented volume.
- κ is an “artistic” variable; acts like ambient “fog”.

Ray Marching

- March along the ray, voxel-by-voxel, from the eye into the volume.
- Initialize $C = (0,0,0)$; $\mathbf{x}_i = \mathbf{x}_e$; $T = 1$
- Iterate for $i=1,\dots$

$$\mathbf{x}_i + = \mathbf{n} \Delta s$$

$$T^* = \exp\{-\kappa \Delta s \rho(\mathbf{x}_i)\}$$

$$C+ = \Delta s \rho(\mathbf{x}_i) c(\mathbf{x}_i) T$$

Note that ρ is 0 “outside” the voxel grid; thus if the eye is placed outside the voxel grid, $T = T^* 1 = 1$ from the eye position until the ray enters the voxel space. This helps optimize code.

Opacity Issues at Edges

- Sometimes see a black fringe around volume edge when composited (blended) into another image:
- Problem lies in how transmittance is integrated into the ray march.



Image from: Antoine Bouthors, Interactive multiple anisotropic multiple scattering in clouds,
ACM Symposium on Interactive 3D Graphics and Games (I3D), 2008

Reformulate Solution

- Instead of

$$C+ = \Delta s \rho(\mathbf{x}_i) c(\mathbf{x}_i) T$$

- Use

$$C+ = \frac{\left(1 - e^{-\kappa \Delta s \rho(\mathbf{x}_i)}\right)}{\kappa} c(\mathbf{x}_i) T$$

Yielding

$$\mathbf{x}_i^+ = \mathbf{n} \Delta s$$

$$\Delta T = \exp\{-\kappa \Delta s \rho(\mathbf{x}_i)\}$$

$$T^* = \Delta T$$

$$C_+ = \frac{1 - \Delta T}{\kappa} c(\mathbf{x}_i) T$$

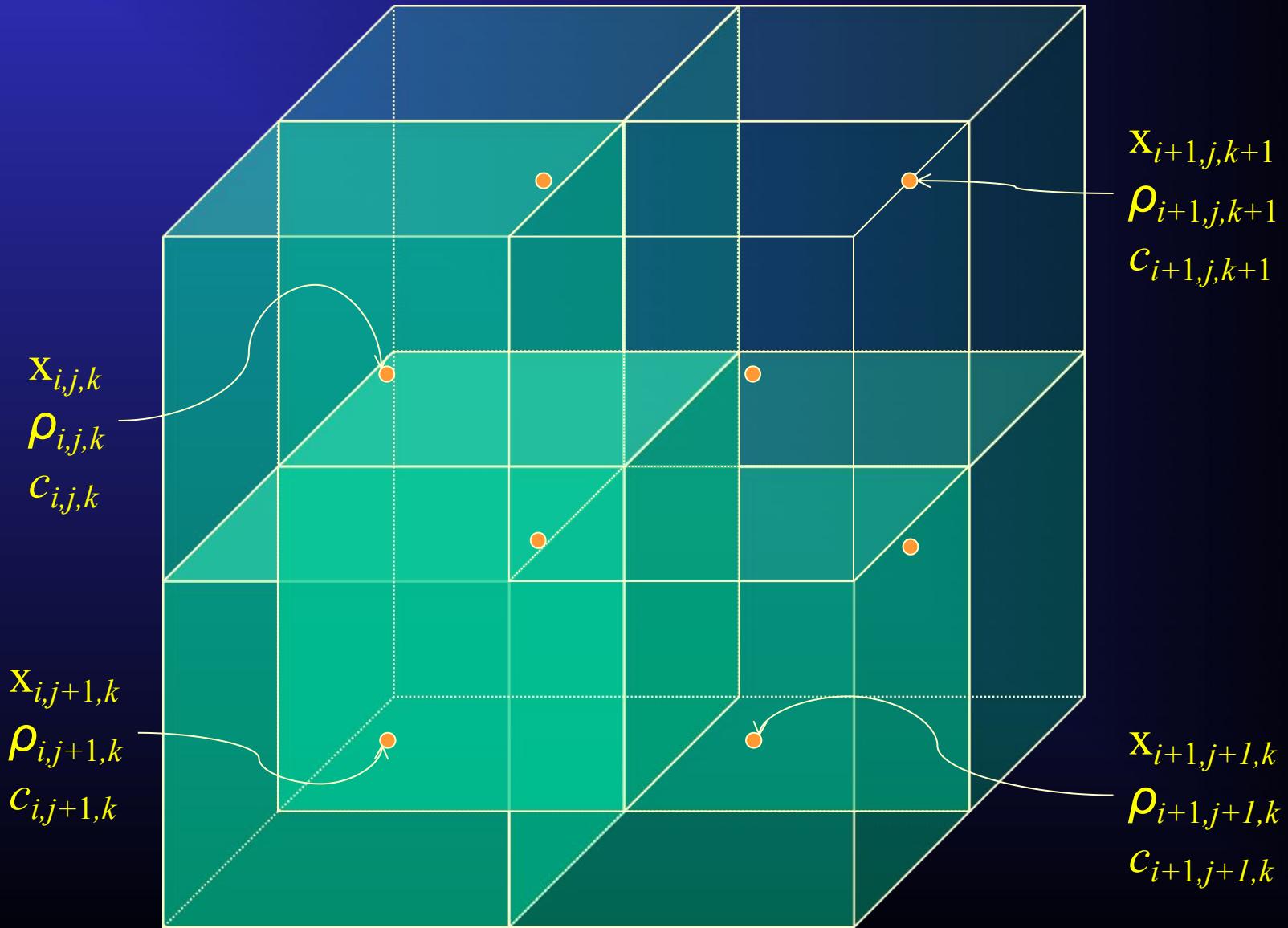
Voxel Gridded Density Fields

- Use bounding boxes with continuous density fields or voxel gridded volumes.
- Rectangular mesh of points \mathbf{x}_{ijk} at the center of voxels labeled ijk , $i,j,k = 1,\dots,M$.
- At each voxel center, density and color have values:

$$\rho(\mathbf{x}_{ijk}) = \rho_{ijk}$$

$$c(\mathbf{x}_{ijk}) = c_{ijk}$$

What this looks like (not all voxels labeled):



Tri-linear Interpolation in Voxel Space

- Can express ray march position as a weighted sum of 8 adjacent voxel positions

$$\mathbf{x}_i = \sum_{a=i}^{i+1} \sum_{b=j}^{j+1} \sum_{c=k}^{k+1} \mathbf{x}_{abc} \omega_{abc}$$

- Weights are positive and normalized

$$\sum_{a=i}^{i+1} \sum_{b=j}^{j+1} \sum_{c=k}^{k+1} \omega_{abc} = 1$$

- Use weights for density & color interpolation

$$\rho(\mathbf{x}_i) = \sum_{a=i}^{i+1} \sum_{b=j}^{j+1} \sum_{c=k}^{k+1} \rho_{abc} \omega_{abc}; \quad c(\mathbf{x}_i) = \sum_{a=i}^{i+1} \sum_{b=j}^{j+1} \sum_{c=k}^{k+1} c_{abc} \omega_{abc}$$

Interpolation Weights

$$\mathbf{x}_i = (x_i, y_i, z_i); \quad \mathbf{x}_{abc} = (x_{abc}, y_{abc}, z_{abc})$$

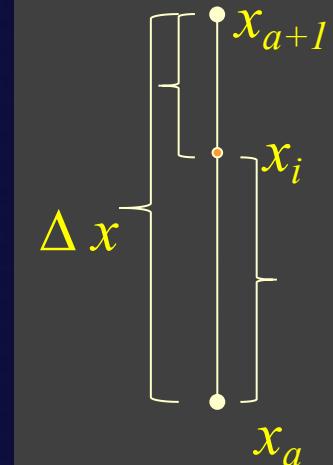
$$\omega_{abc}^x = \frac{\Delta x - |x_i - x_{abc}|}{\Delta x}$$

$$\omega_{abc}^y = \frac{\Delta y - |y_i - y_{abc}|}{\Delta y}$$

$$\omega_{abc}^z = \frac{\Delta z - |z_i - z_{abc}|}{\Delta z}$$

$$\omega_{abc} = \omega_{abc}^x \omega_{abc}^y \omega_{abc}^z$$

e.g., in 1D:

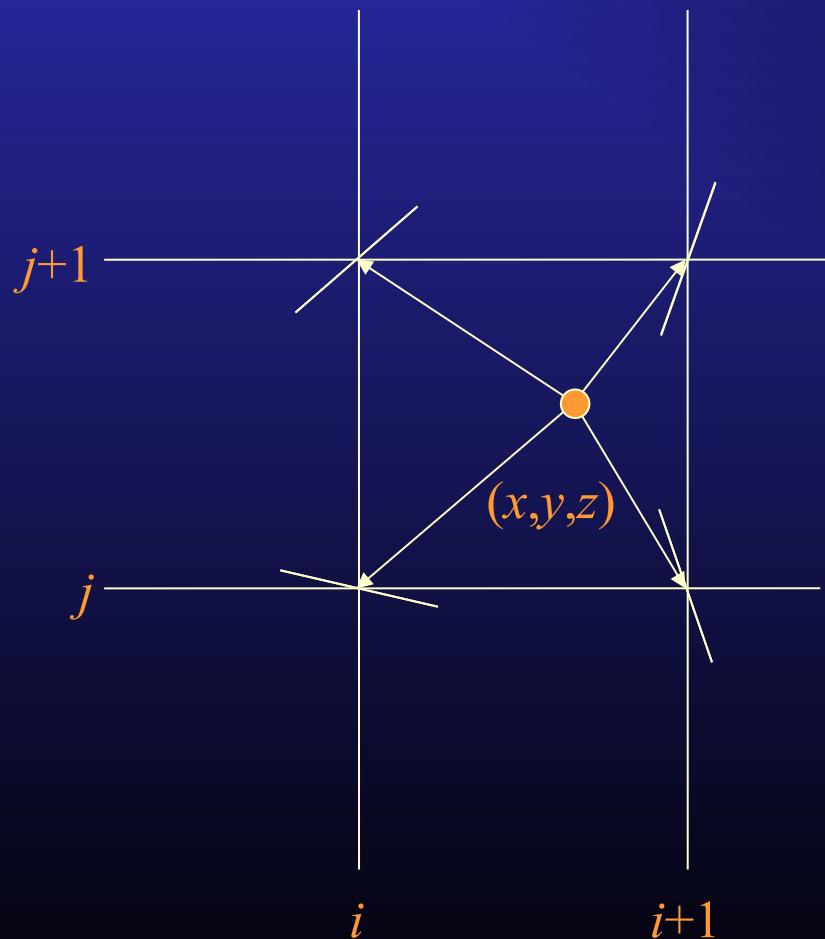


Perlin Noise Function (2D/3D) Generator

- Not just random “noise”: want some correlation for macroscopic scale texture.
 - Also want a function so that it can be repeatably evaluated with given input parameters.
 - Resembles fractal generation process but bandlimited (by using a finite resolution texture lattice).
-
- Divide space to a 3D regular lattice.
 - Assign a pseudo-random “wavelet” to each lattice point: a gradient than drops off to zero and integrates to 0.
 - The noise value for any arbitrary 3D point is computed by distance-weighted interpolation of the eight nearest lattice points...

General Idea (in 2D) with Weight Computation (in 3D)

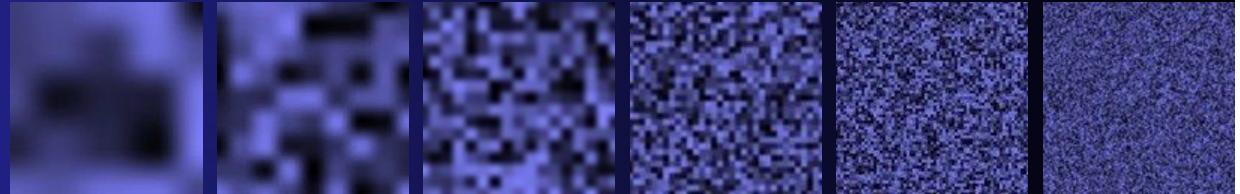
Pre-stored gradients $G[i, j, k]$
in range (-1.0 .. 1.0)



- For any point (x, y, z) , the wavelet from lattice $[i, j, k]$ is:
 - $[u, v, w] = [x-i, y-j, z-k]$
 - Weight =
 $\text{dropoff}(u)*\text{dropoff}(v)*\text{dropoff}(w)$
where $\text{dropoff}(t) = 6|t|^5 - 15|t|^4 + 10|t|^3$
 - $F = G[i, j, k] \cdot [u, v, w]$
 - Wavelet = Weight * F
- The noise value for (x, y, z) is then the sum of eight wavelets from the point's 3D lattice corners (though only 4 are shown in the figure at left).

Perlin Turbulence

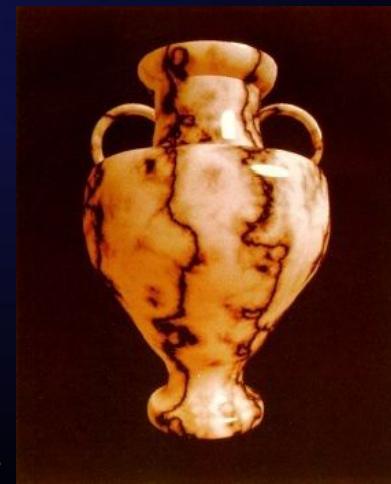
```
function turbulence(p)
    t = 0
    scale = 1
    while (scale > pixelsize)
        t += abs(Noise(p / scale) * scale)
        scale /= 2
    return t
```



=sum

For example:

```
function marble(point)
    x = point[1] + turbulence(point)
    return marble_color(sin(x))
```



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

Lighting Volumes

- Lights modify the ray march procedure
- The color of the material is “multiplied” by the color of the light.
- The color of the light is attenuated by the volume material between the light and the ray march points.

Recall: Color Triplet Product

- Color triplet $c = (r, g, b)$
- Color triplet $F = (F_r, F_g, F_b)$
- Component-wise product is a color triplet

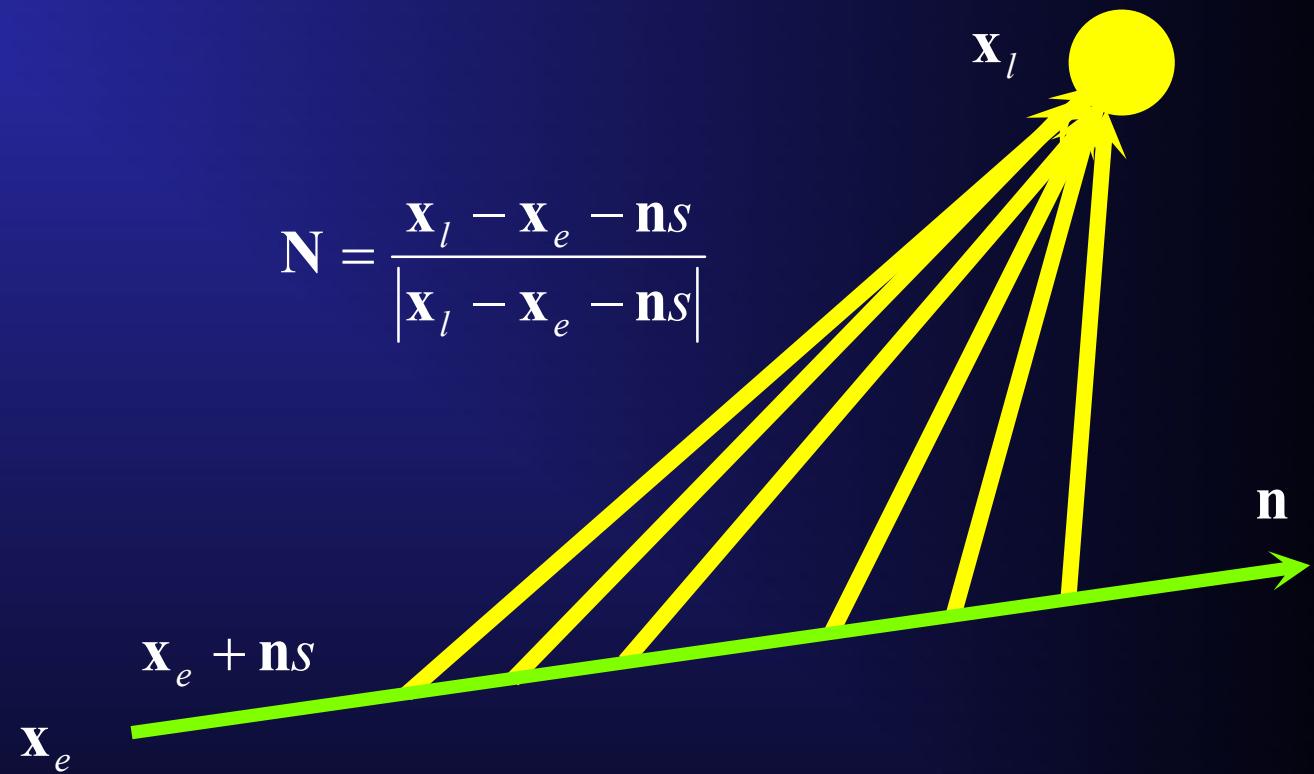
$$c \odot F = (rF_r, gF_g, bF_b)$$

When it is clear, we will omit the \odot operator.

Point Light Source(s)

- Position of light: \mathbf{x}_l
- Light intensity (color) in a vacuum: $F = (F_r, F_g, F_b)$
- In volumetric medium, intensity depends on how much material density exists between the light and the ray march point.
- Same form of transmittance as for ray march, but in the direction between the light and the ray march point.
- (Note that the following formulas only account for real-valued density in the volume, not the *colors* of any material between the ray march and the light. It is easy enough to accommodate this by separately considering the RGB colors as three independent densities.)
- It is also possible to extend this to multiple light sources by summing over each source's contribution.

Viewed Along the Ray



$$Q(\mathbf{x}_e \mathbf{n}, s, \mathbf{x}_l) = \exp \left\{ -\kappa \int_0^D ds' \rho(\mathbf{x}_e + \mathbf{n}s + \mathbf{N}s') \right\}$$

$$D = |\mathbf{x}_l - \mathbf{x}_e - \mathbf{n}s|$$

Rendering Form with Lights

$$C(\mathbf{x}_e, \mathbf{n}) = \int_0^\infty ds \rho(\mathbf{x}_e + \mathbf{n}s)(c(\mathbf{x}_e + \mathbf{n}s)F) \exp\left\{-\kappa \int_0^s ds' \rho(\mathbf{x}_e + \mathbf{n}s')\right\} Q(\mathbf{x}_e, \mathbf{n}, s, \mathbf{x}_l)$$

- Ray march with lights (for i=1,...):

$$\mathbf{x}_i+ = \mathbf{n}\Delta s$$

$$\Delta T = \exp\left\{-\kappa \Delta s \rho(\mathbf{x}_i)\right\}$$

$$T^* = \Delta T$$

$$C+ = \frac{1 - \Delta T}{\kappa} (c(\mathbf{x}_i)F) T Q(\mathbf{x}_e, \mathbf{n}, i \Delta s, \mathbf{x}_l)$$

Can Precompute Light Transmittance (assuming voxel densities stay constant and light is fixed)

- Compute Q to each voxel center
- Store Q_{ijk} at each voxel
- Use tri-linear interpolation for points off voxel centers

$$Q_{ijk} = \exp \left\{ -\kappa \int_0^D ds' \rho(\mathbf{x}_{ijk} + \mathbf{N}_{ijk} s') \right\}$$

$$\mathbf{N}_{ijk} = \frac{\mathbf{x}_l - \mathbf{x}_{ijk}}{\|\mathbf{x}_l - \mathbf{x}_{ijk}\|}$$

Full Ray March with Lights

$$\mathbf{x}_i + = \mathbf{n} \Delta s$$

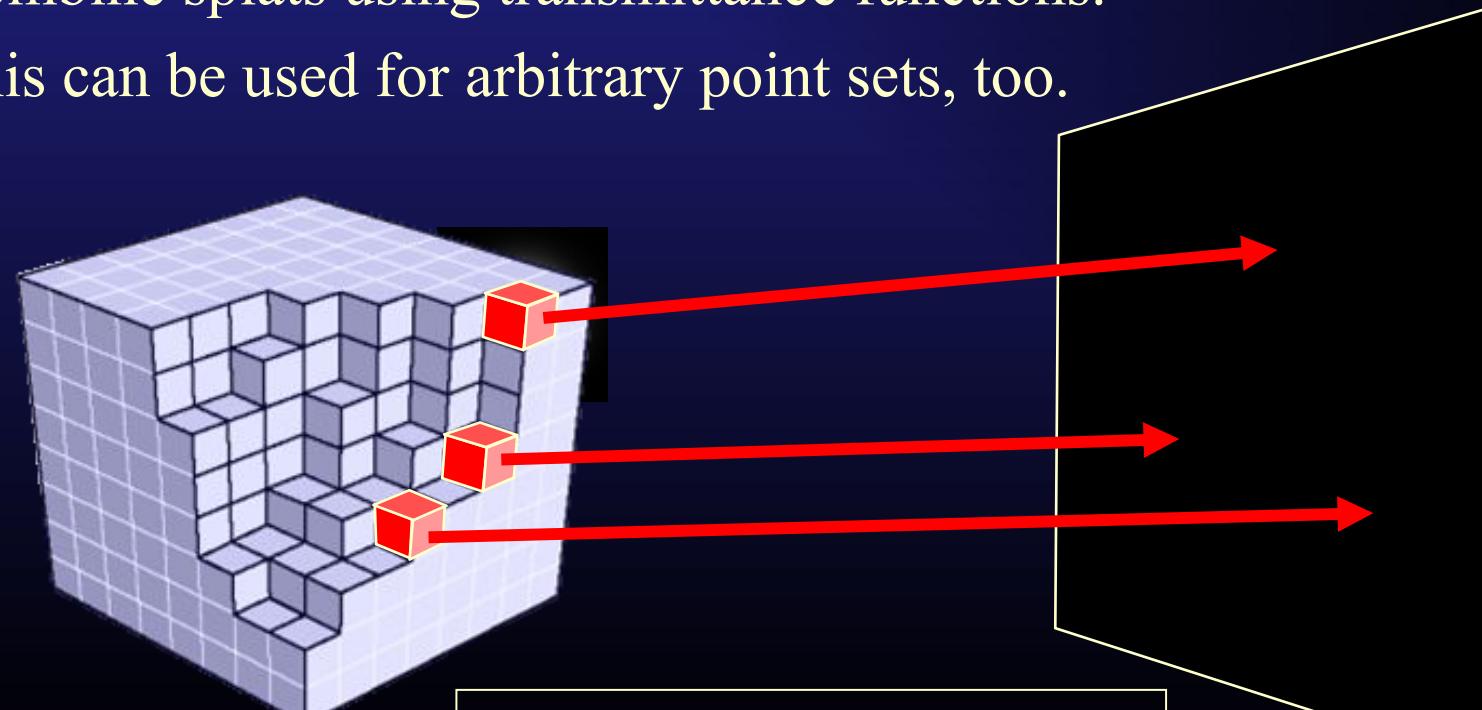
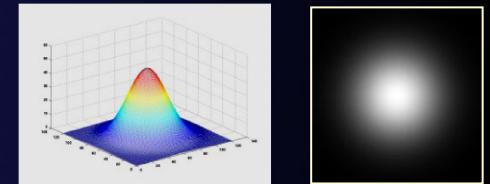
$$\Delta T = \exp\{-\kappa \Delta s \rho(\mathbf{x}_i)\}$$

$$T^* = \Delta T$$

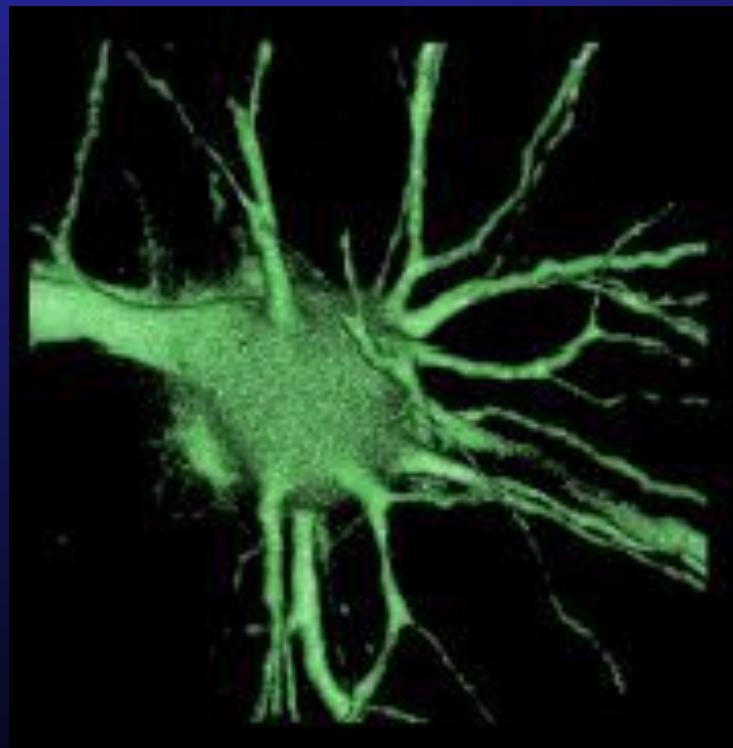
$$C+ = \frac{1 - \Delta T}{\kappa} (c(\mathbf{x}_i) F) T Q(\mathbf{x}_i)$$

Alternative to Ray Casting in Volumes: Splatting

- Instead of shooting rays from each pixel into the volume, start with the voxels and for each draw a fixed pattern – a “splat” – into the frame buffer.
- Splat is often a Gaussian: $f(x, y) = \int e^{-0.5(x^2 + y^2 + z^2)} dz$
- Can shoot front-to-back or back-to-front.
- Combine splats using transmittance functions.
- This can be used for arbitrary point sets, too.

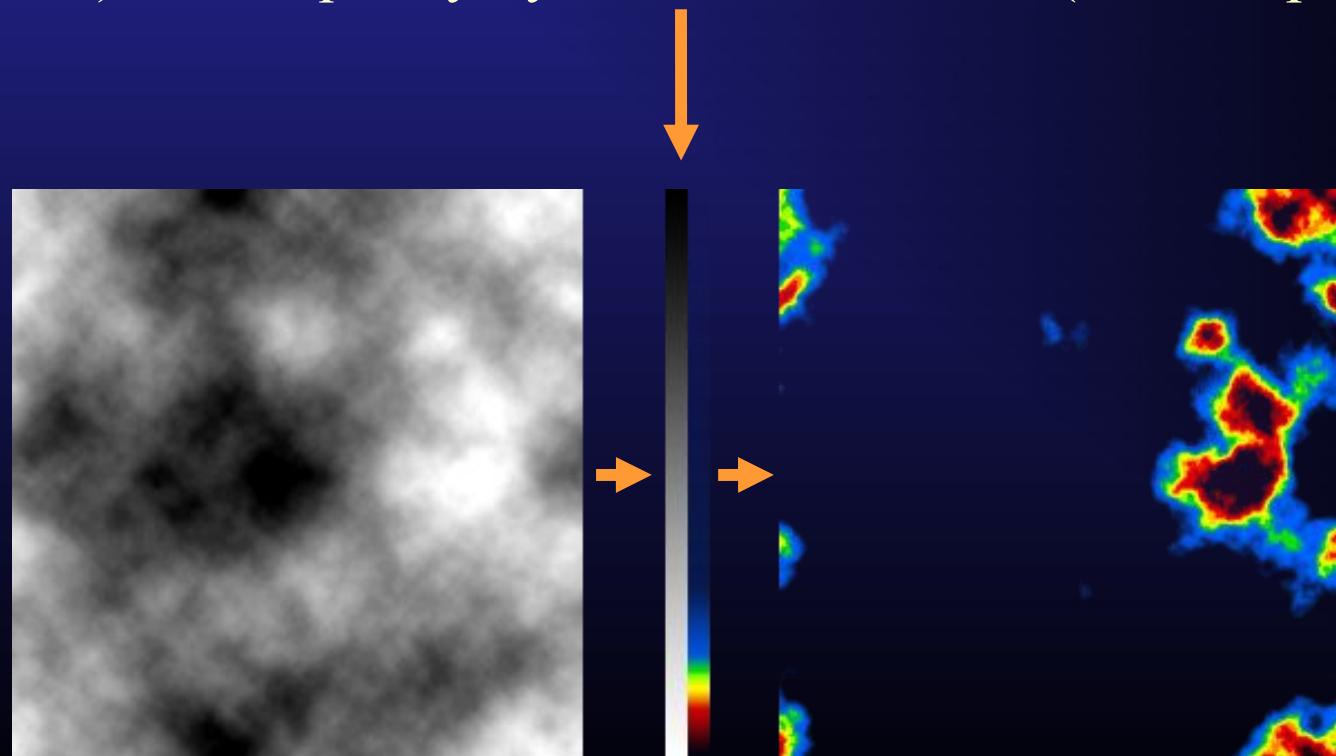


Splatting Examples



Volume Rendering

- Rays may be classified by the surfaces they are meant to represent (may be manual or automatic, say, depending on density).
- Transmittance scalar values can be further mapped to another (artificial) visual quality by a **transfer function** (a look-up table).



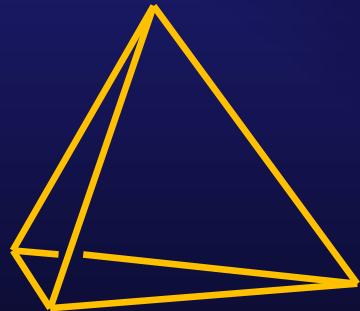
Different Transfer Functions Yield Different Images from Same Dataset



©1991 VINCENT ARGIRO VITAL IMAGES, INC.

Tetrahedral Mesh

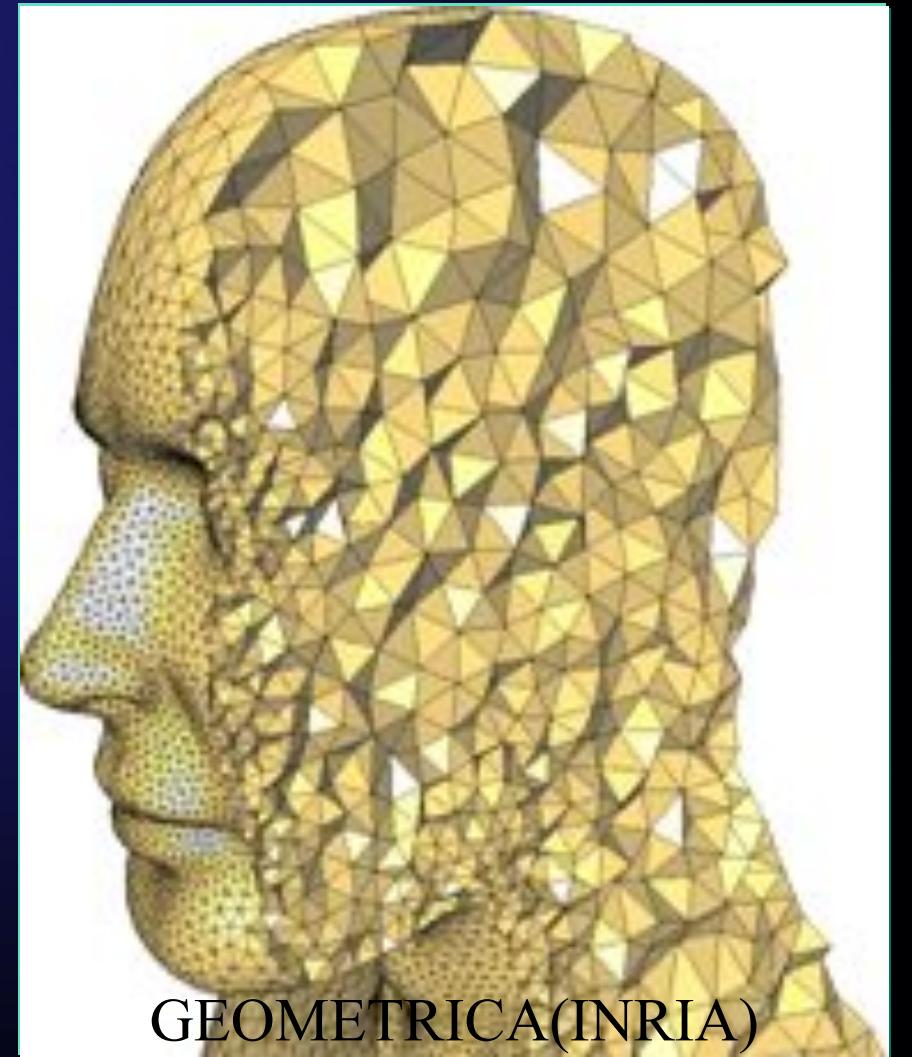
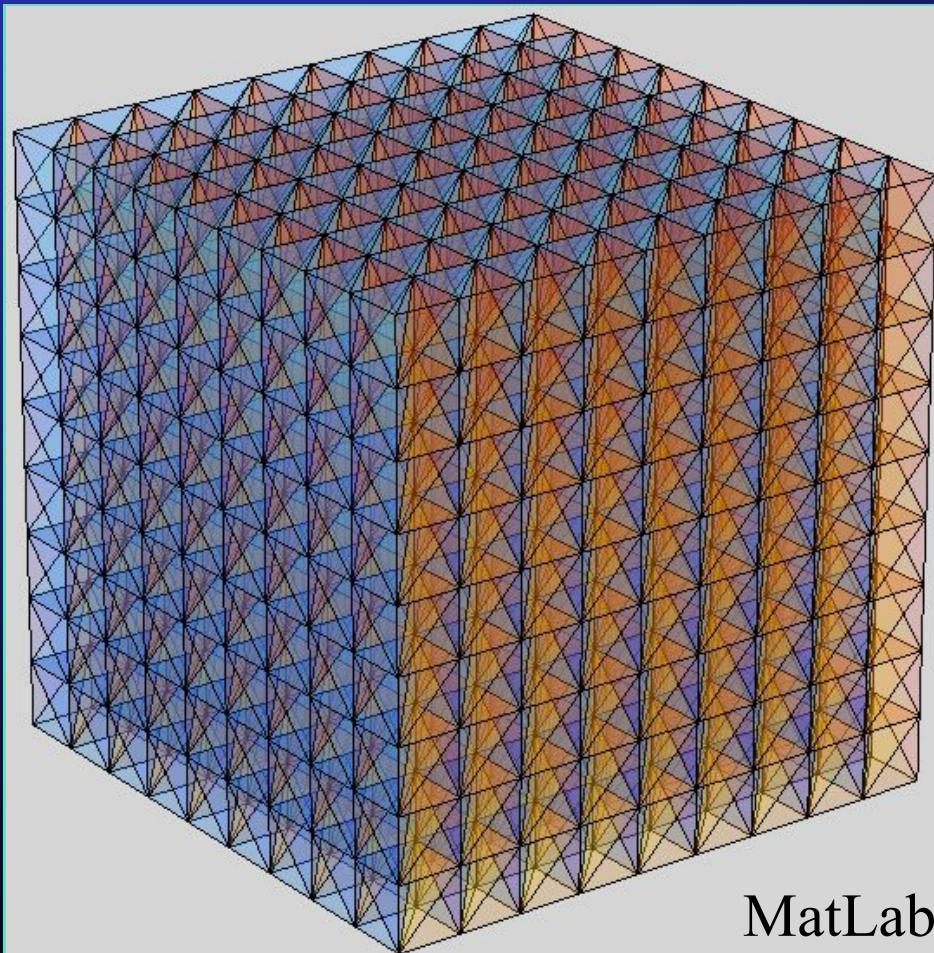
- A tetrahedron forms the basic element (a 3-simplex*).
- Each element in the mesh is a tetrahedral solid.
- Edges need not be equal lengths.
- Allows for much smoother surface shapes than voxels as the surface is triangulated.
- Bridge to physics-based models (e.g., spring-mesh systems, finite element models).



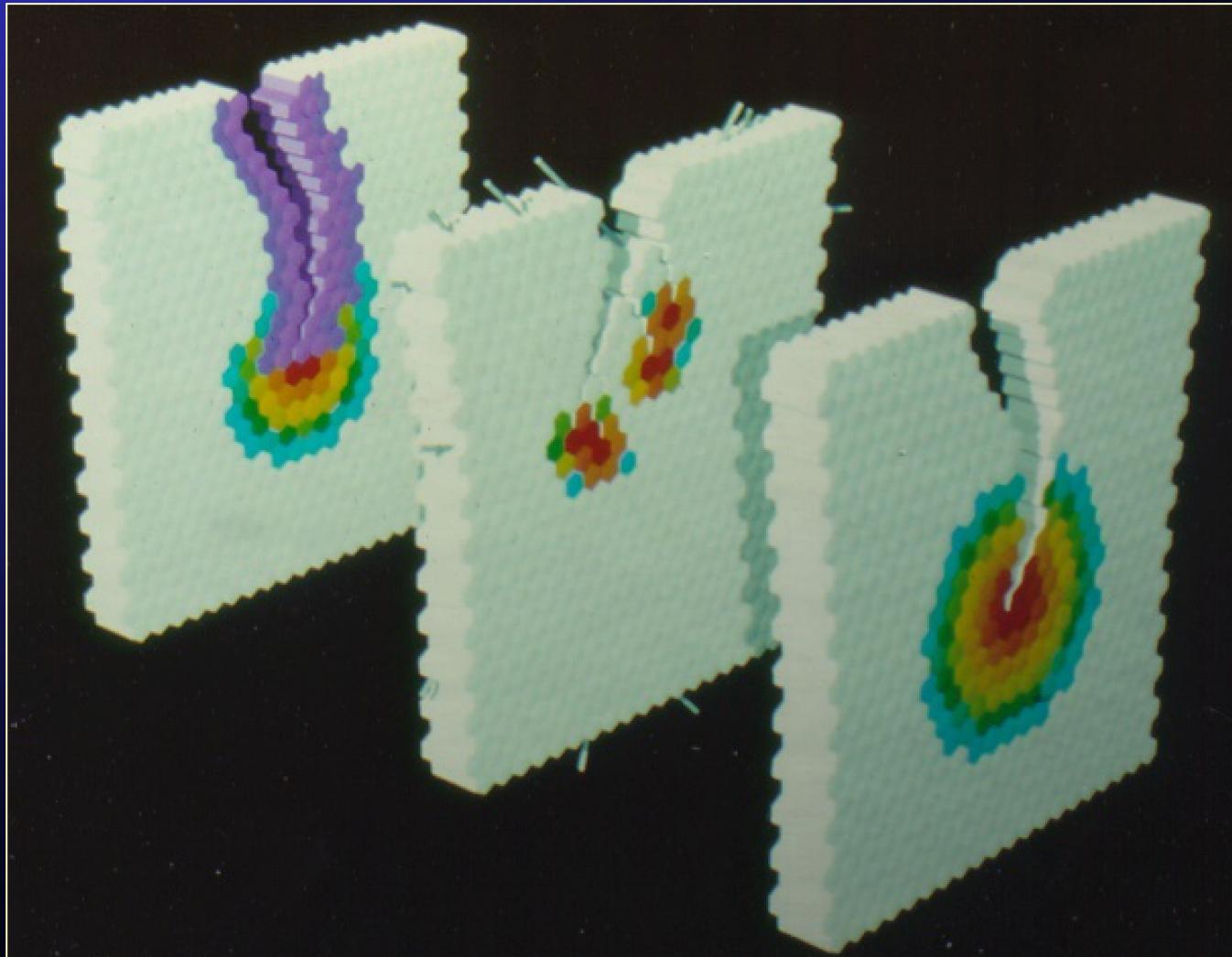
*An n -simplex is the convex hull of $n+1$ affinely independent points a_0, a_1, \dots, a_n , i.e., the vectors $\{a_0a_1, a_0a_2, \dots, a_0a_n\}$ are linearly independent.

For $n=3$ space consider $a_0=(0,0,0)$, $a_1=(1,0,0)$, $a_2=(0,1,0)$, $a_3=(0,0,1)$; which thus define a tetrahedron. In general, the points a_i need not lie along the coordinate axes.

Tetrahedral Mesh Examples



Hexagonal Finite Elements



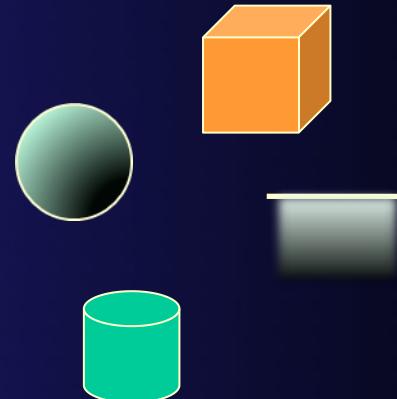
© 1992 MELVIN L.PRUETT,LOS ALAMOS NL.

Constructive Solid Geometry

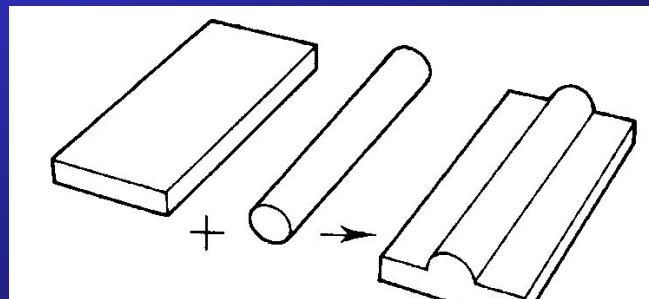
- Appel 1969; Goldstein & Nagel 1971.
- Simple solid primitives, e.g.
 - Cube, sphere, wedge, cylinder, cone, half-space, etc.
- Objects may be created by boolean combinations of other objects or primitives via
 - Union, intersection, and difference
- Object “exists” as a CSG tree of solids and operations:
 - Primitives at leaves
 - Transformations and attributes on tree edges
 - Boolean operations at non-terminal nodes
- Tree structure is “evaluated” during rendering via ray-casting.

CSG Primitives

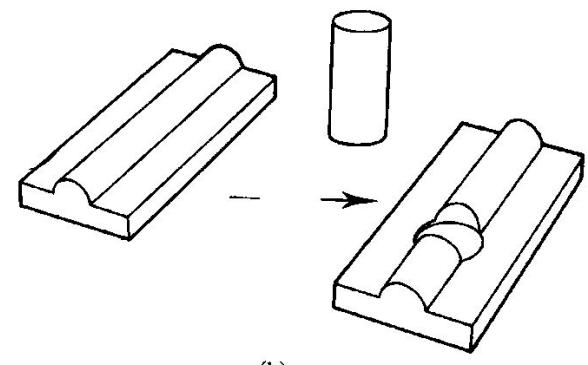
- Defined “mathematically”, e.g. all points (x, y, z) such that:
 - Unit cube: $0 \leq \{x, y, z\} \leq 1$
 - Unit sphere: $x^2 + y^2 + z^2 \leq 1$
 - Half space (e.g.): $z \leq 0$
 - Cylinder: $x^2 + y^2 \leq 1$ and $0 \leq z \leq 1$
 - etc.
- And transformed (scaled, rotated, translated, etc.) as needed:
 - Slabs and sticks are non-uniformly scaled unit cubes
 - Ellipsoids are non-uniformly scaled spheres
 - Half spaces can be defined by any plane equation
 - etc.



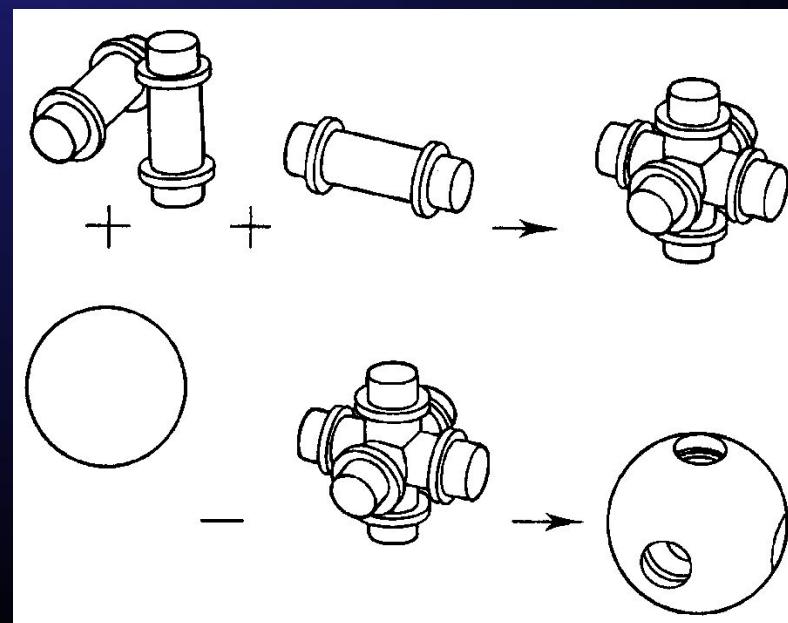
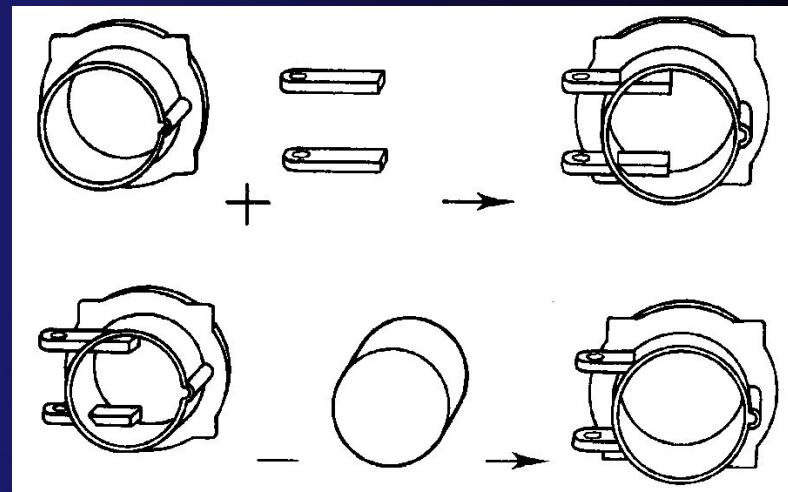
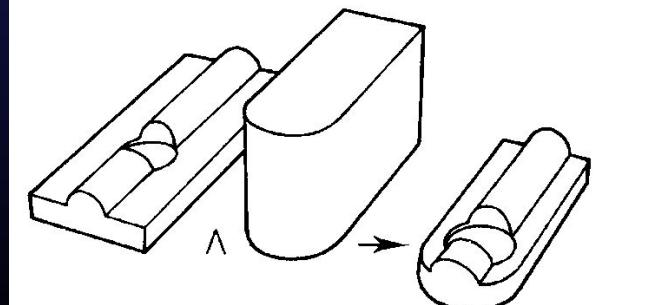
CSG Examples: Sequential Boolean Operations



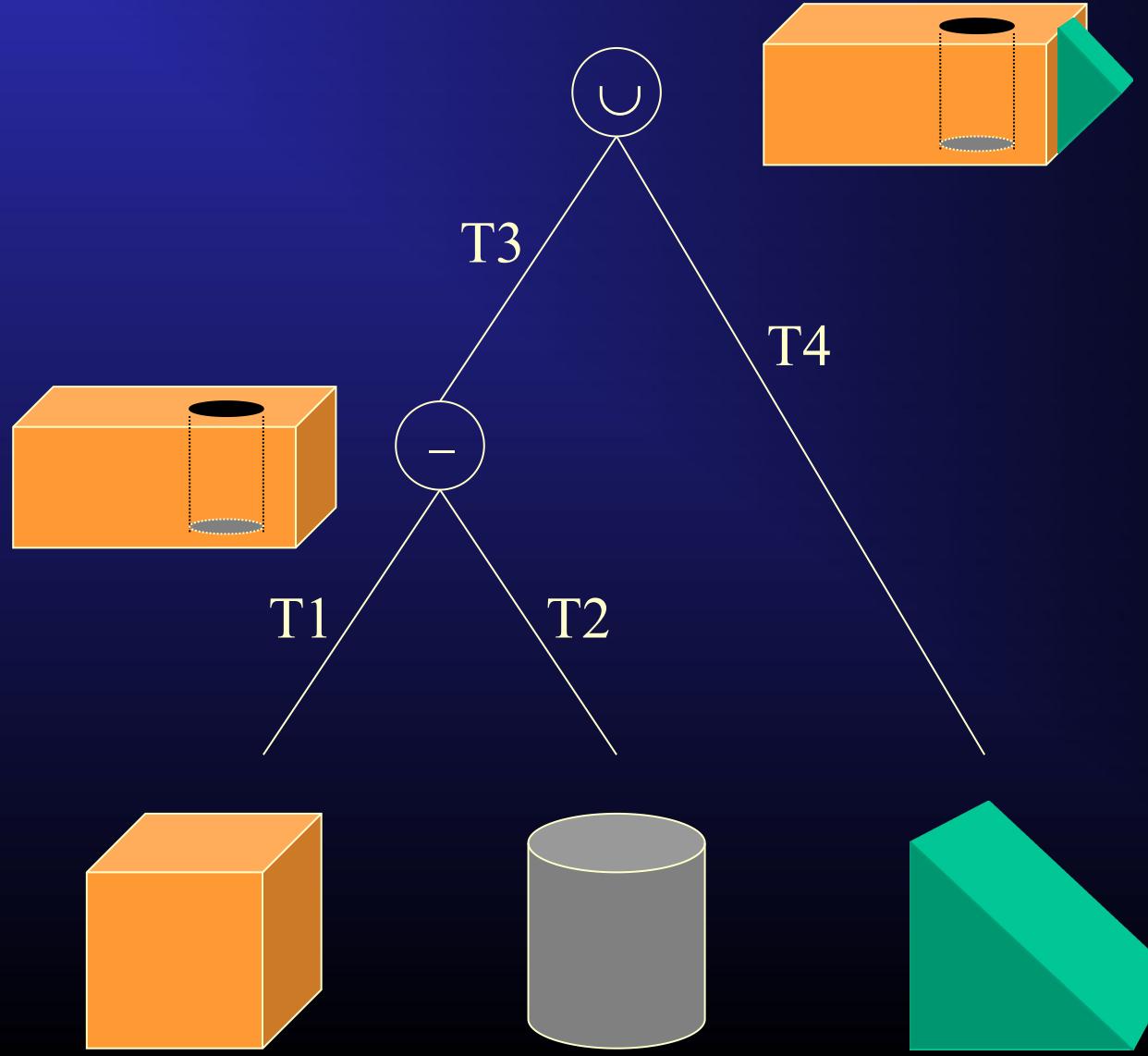
(a)



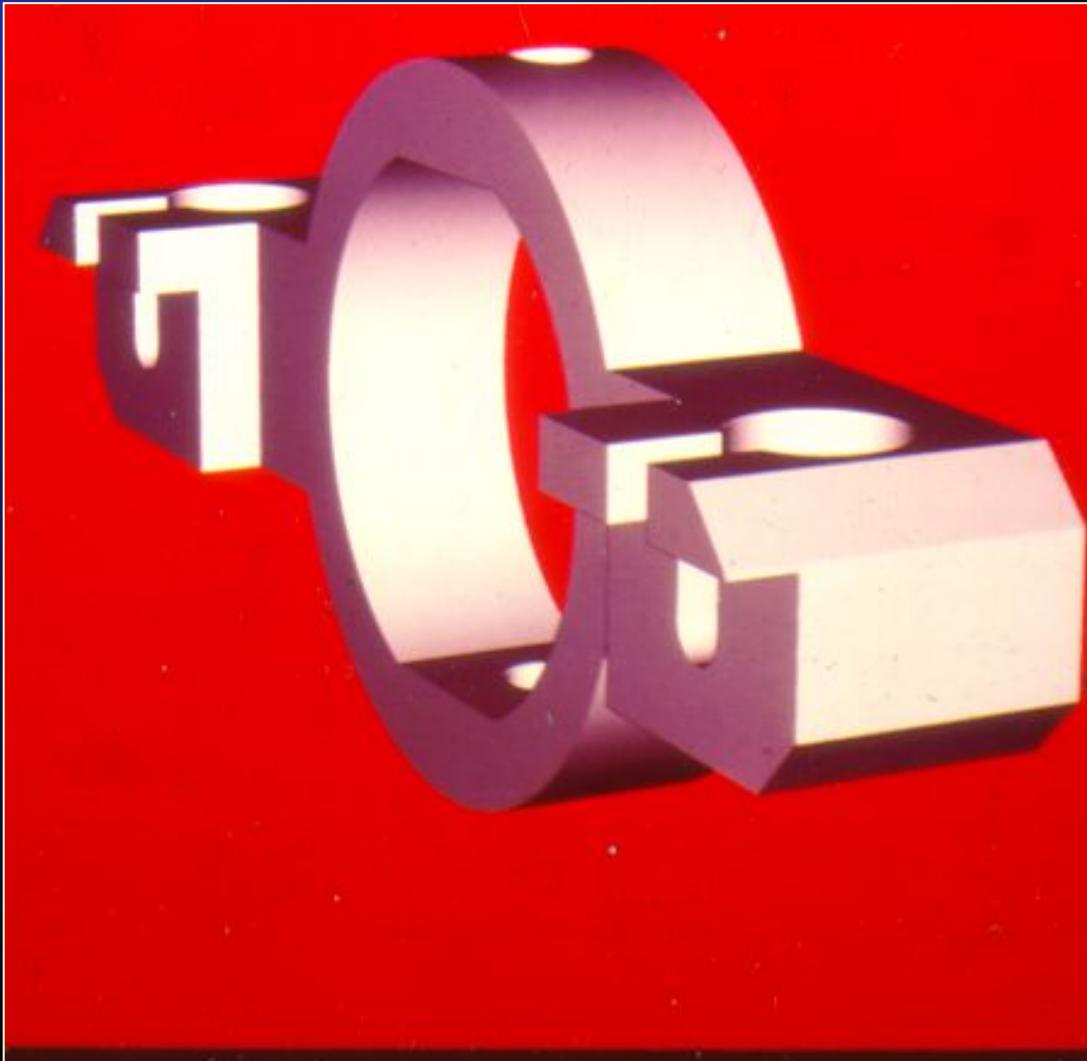
(b)



CSG Example Structure

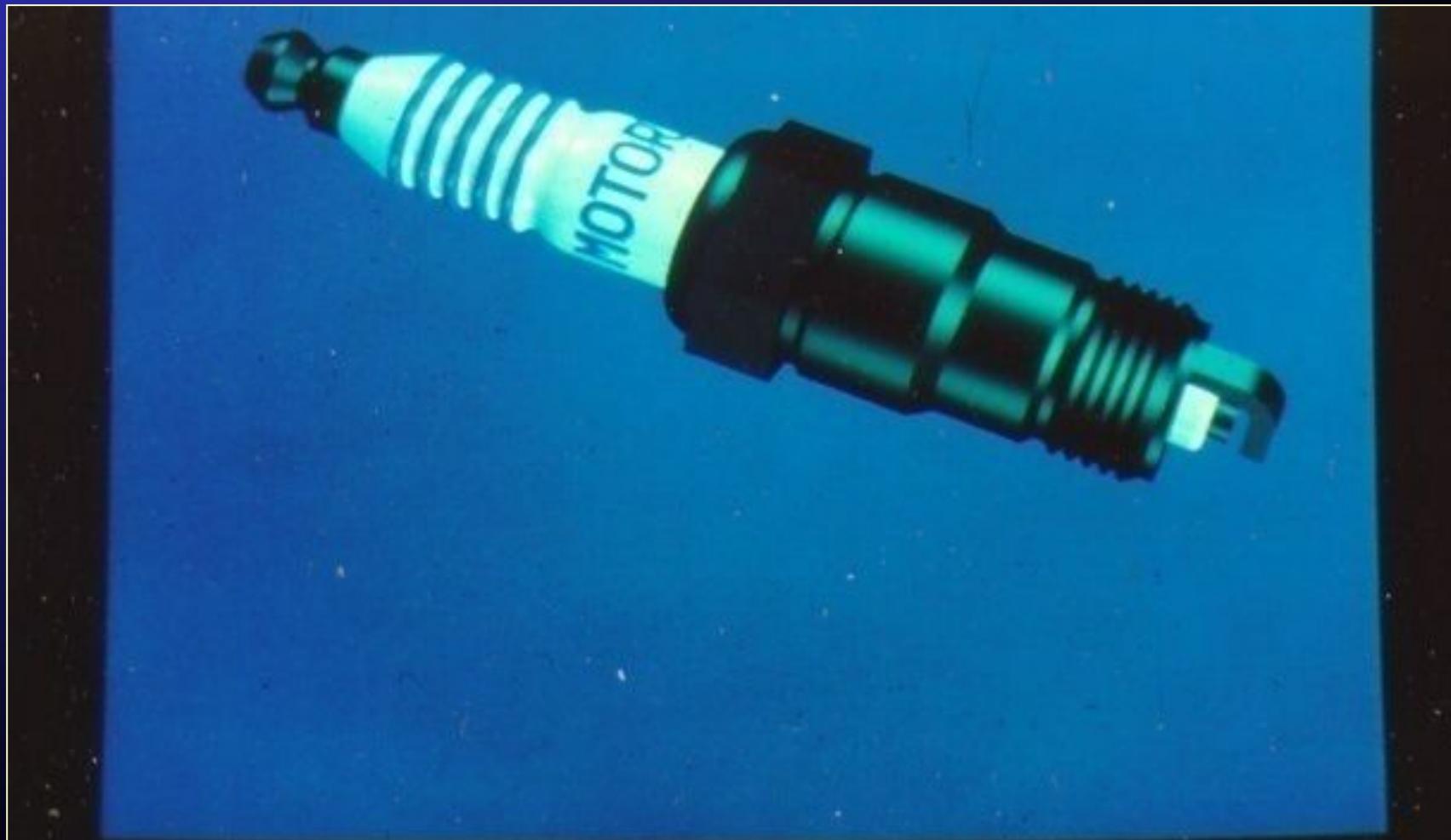


CSG Bracket

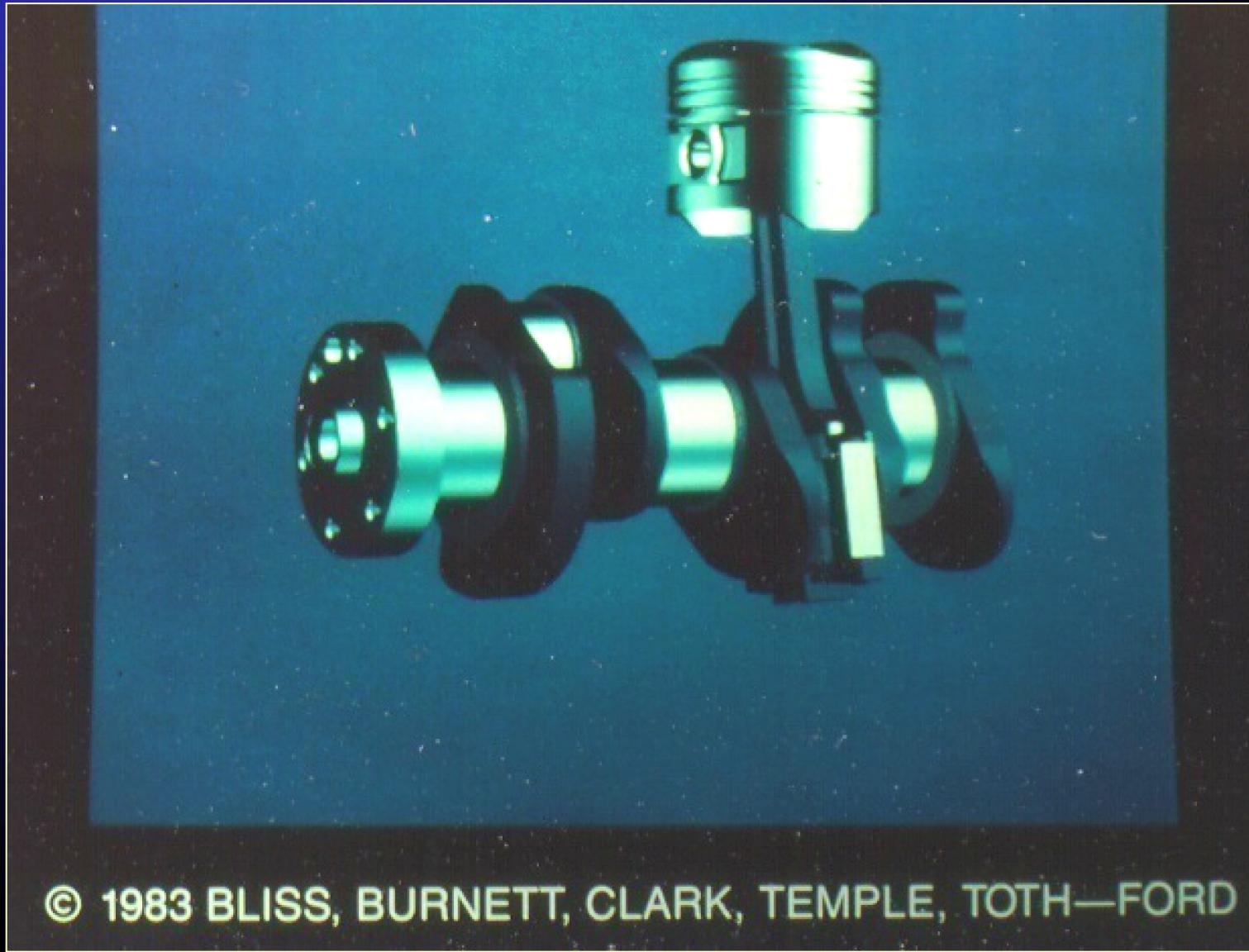


TAYLOR, L.—LLNL

Beautiful (but Uninsertable)

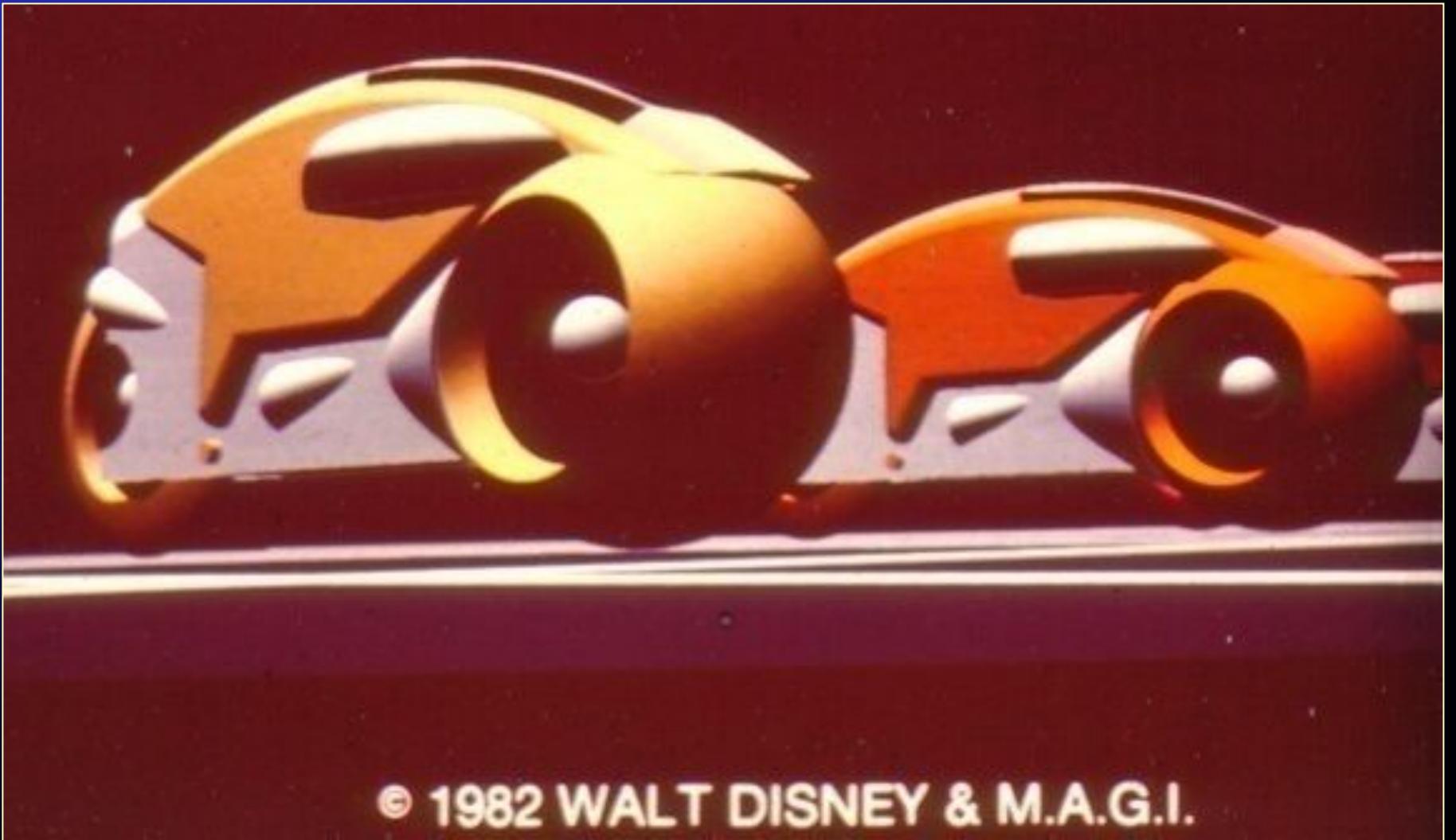


CSG Can Include Parametric Surfaces



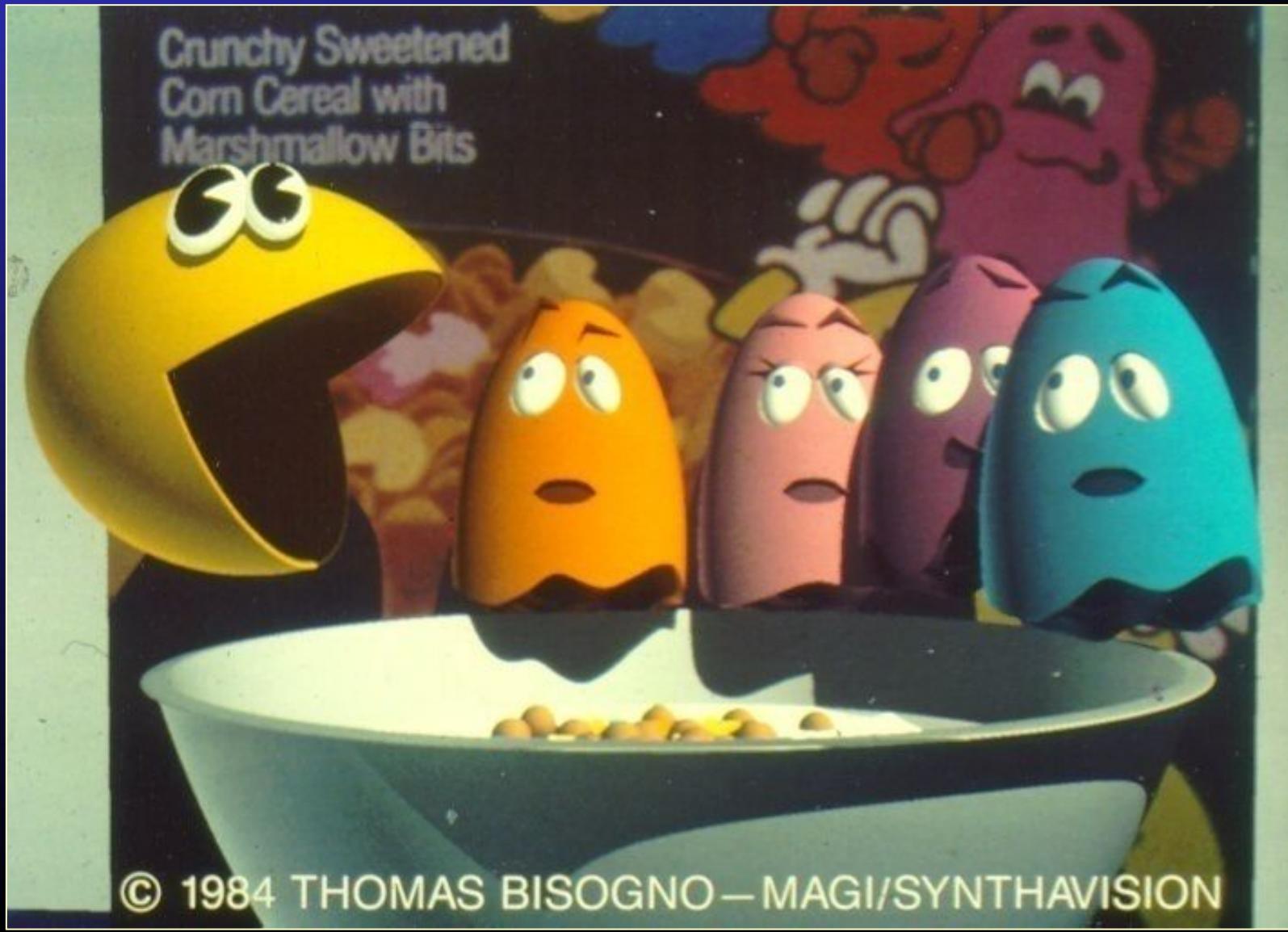
© 1983 BLISS, BURNETT, CLARK, TEMPLE, TOTH—FORD

Lightcycles from TRON



© 1982 WALT DISNEY & M.A.G.I.

CSG Animated Cereal Ad



Surface and Boundary Models

- Height fields
- Polygon meshes
- Fractals
- Curved surfaces
- Implicit surfaces
- Superellipsoids
- Potential functions, Metaballs

Height Fields

- List of coordinate triples
 - Height field is a function $z = F(x, y)$ for a regular grid of (x, y) values.
 - Can be created from photographs by interpreting luminance or “gray-value” as a height.
 - Can be created by procedural methods (functions, obviously), random processes, fractals,...
 - Requires fairly dense distribution of points for accurate modeling.

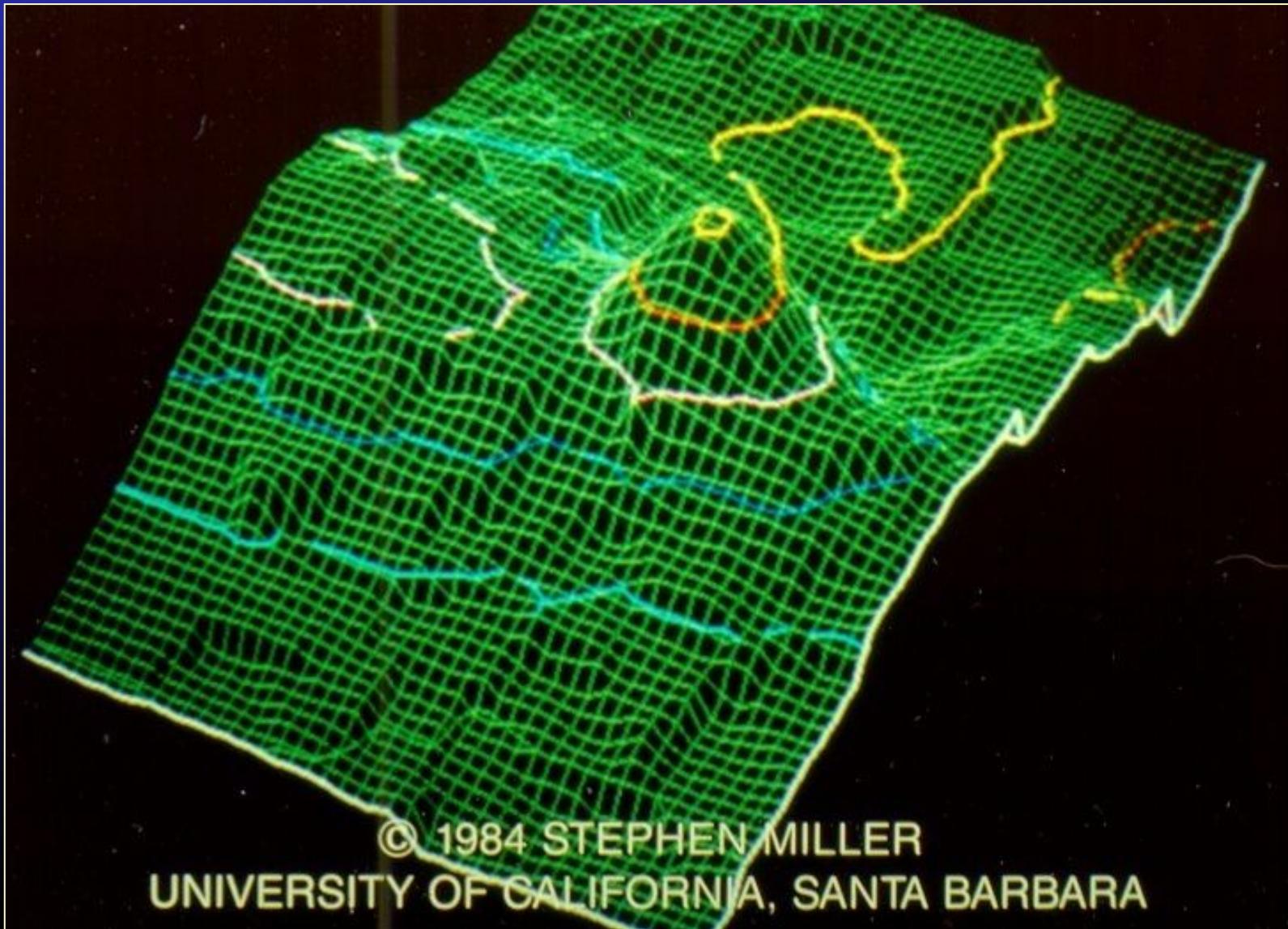


Google Earth
(visit the Grand Canyon)

3D Graph Defines a Surface as a Height Field:
This one is just drawn with line segments parallel to x- and y-axes



Gridded Map from Scattered (x, y) Data Points:
All Regular Grid Points are Computed by Interpolation

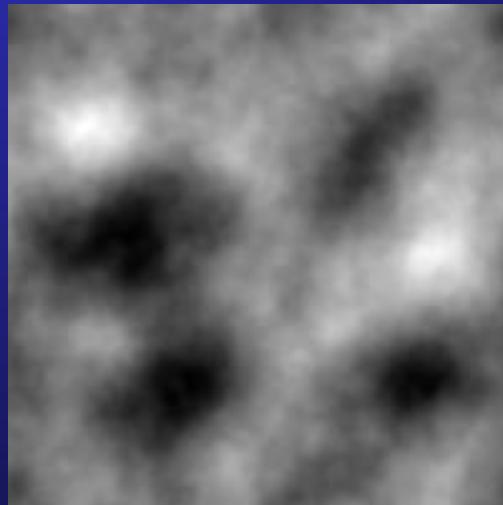


© 1984 STEPHEN MILLER
UNIVERSITY OF CALIFORNIA, SANTA BARBARA

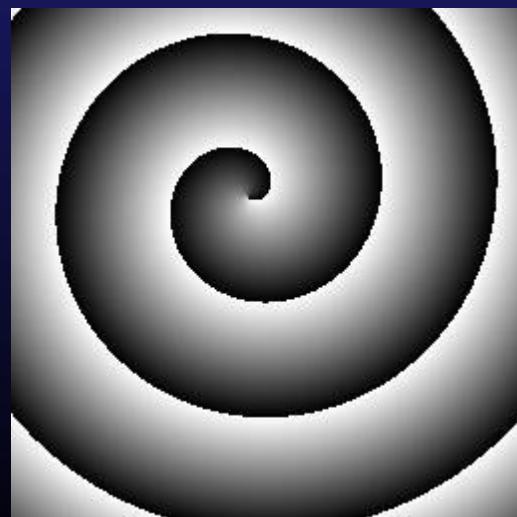
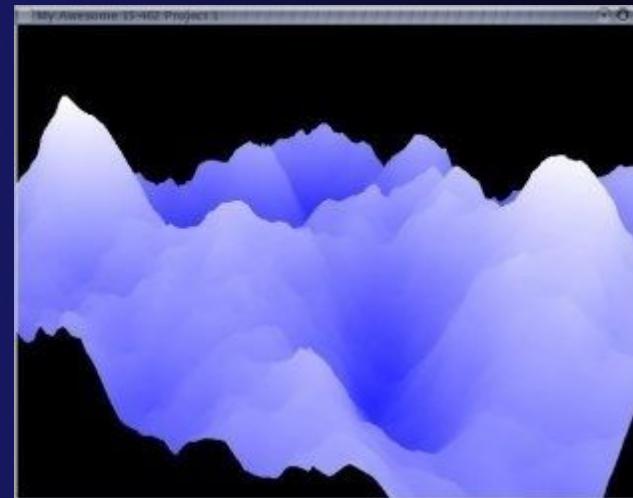
Drawing Surfaces

- The height field can be drawn as a solid-looking surface.
- But to do this, we'll look at how to draw general surfaces composed of 3D polygons. (soon)

Images to Height Field Examples



S H
o e
u i
r g
c h
e t



i f
m i
a e
g l
e d



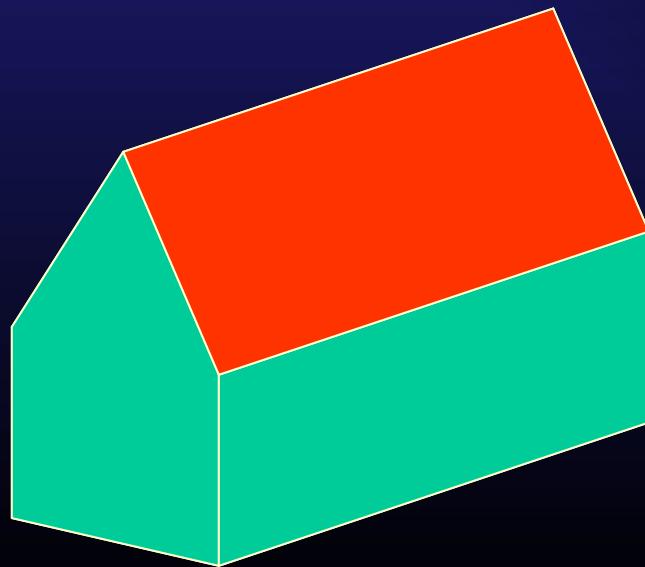
Height Fields from Continuous Functions: Fourier Series Water and Clouds



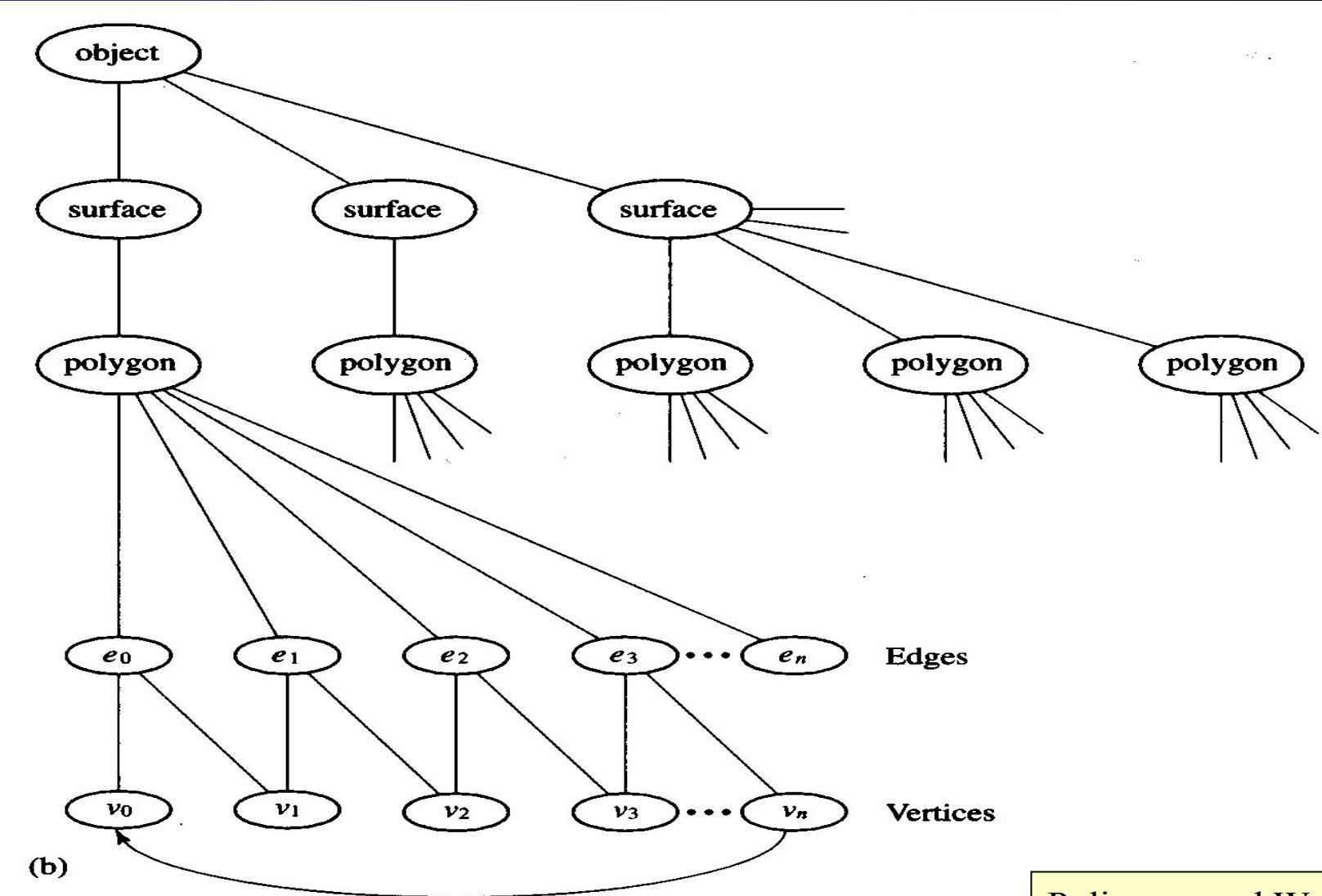
© 1986 D. PEACHEY, UNIV. OF SASKATCHEWAN

Polygons and Meshes

- Vertex, edge, face structure
- Variety of data structures possible
- Completely tile surface at some resolution
- Relatively simple to define, manipulate, and display
- Commonly used by GPU architectures



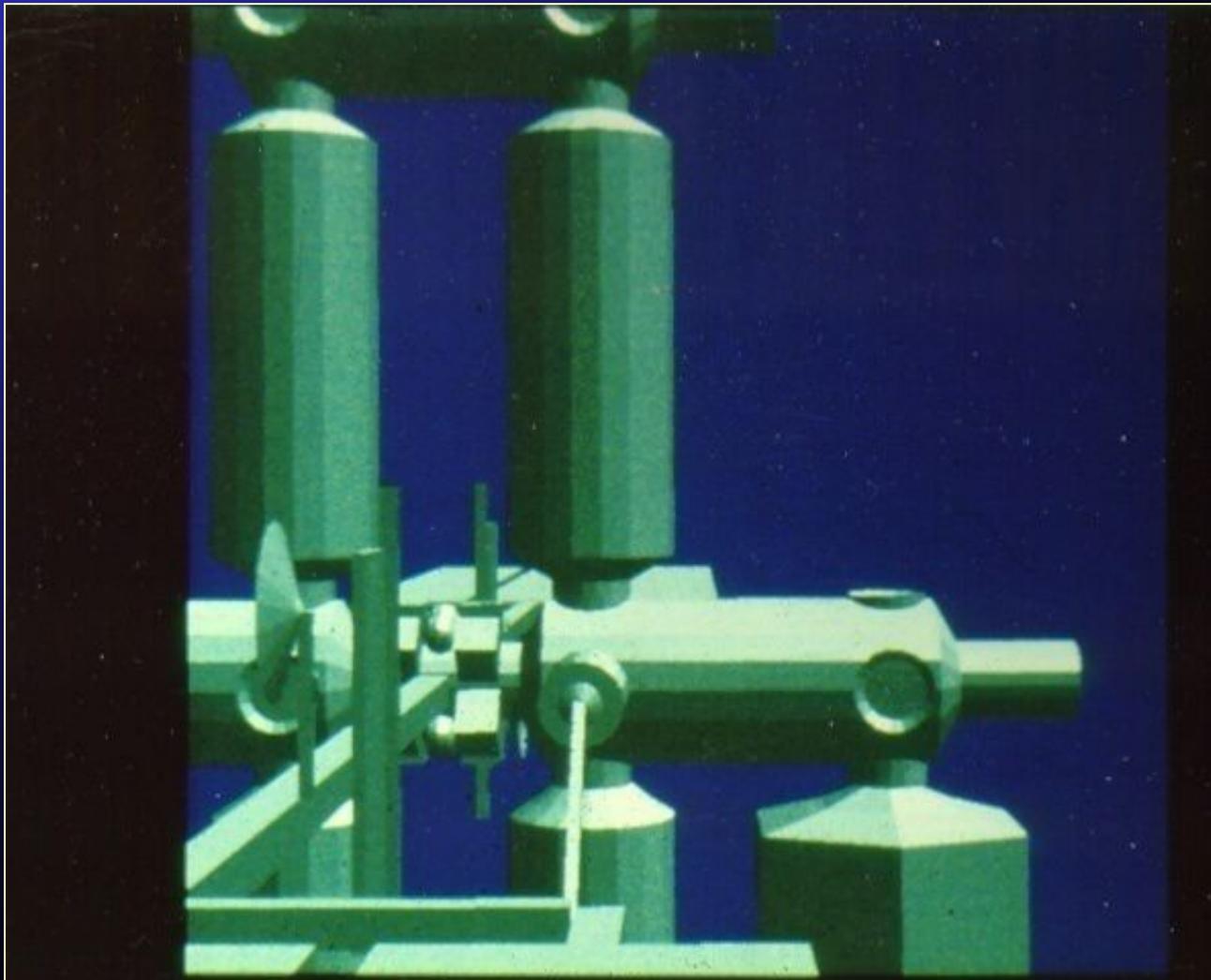
Polygon Mesh Conceptualization



Polygonal Buildings

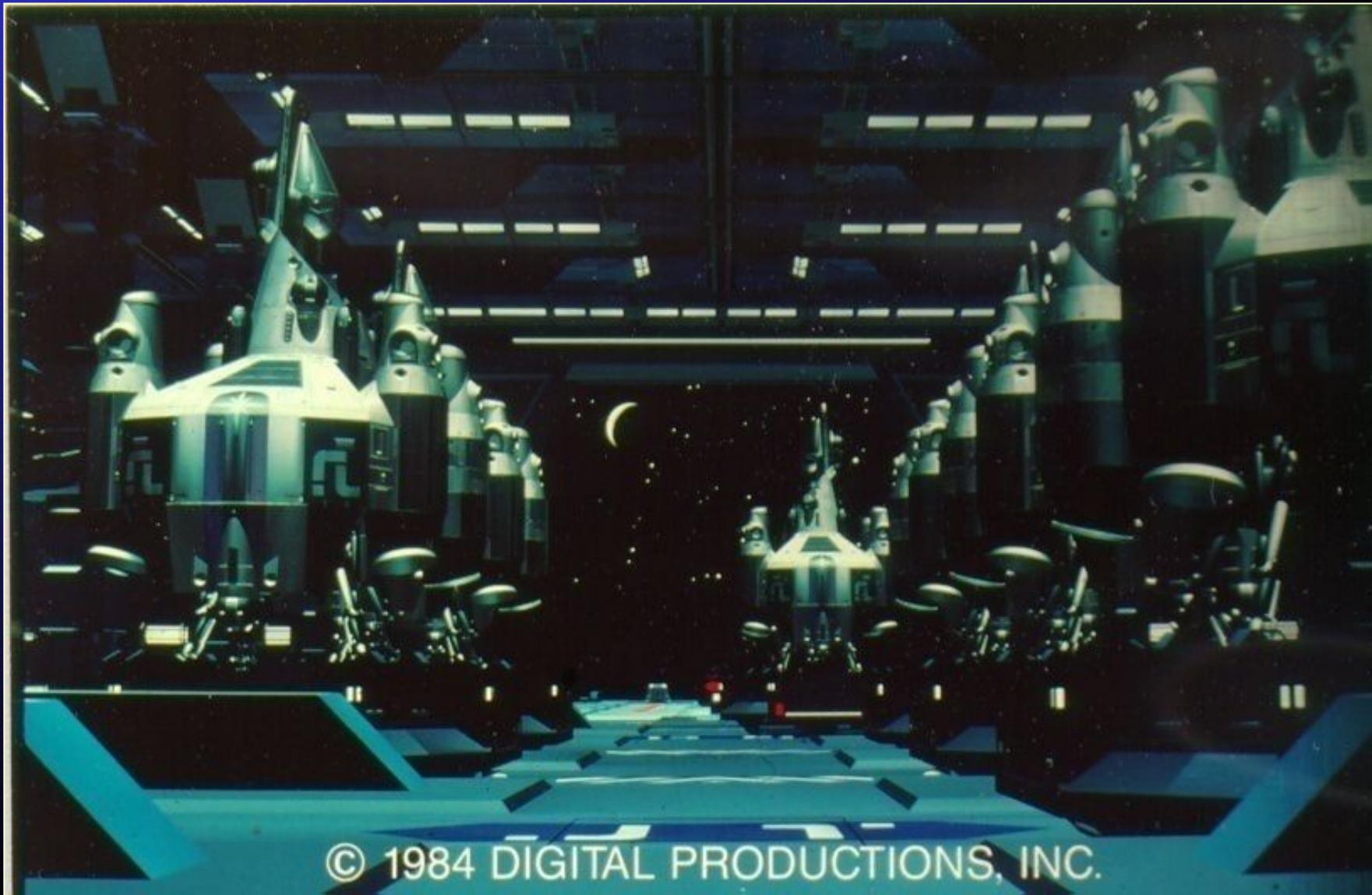


Round or Faceted?



© 1983 GATES, VON OFENHEIM—COMPUTER SCIENCES

8 Million Polygons (Note Smoothness!)



© 1984 DIGITAL PRODUCTIONS, INC.

Polygon Mesh Modeling Becomes an Industry

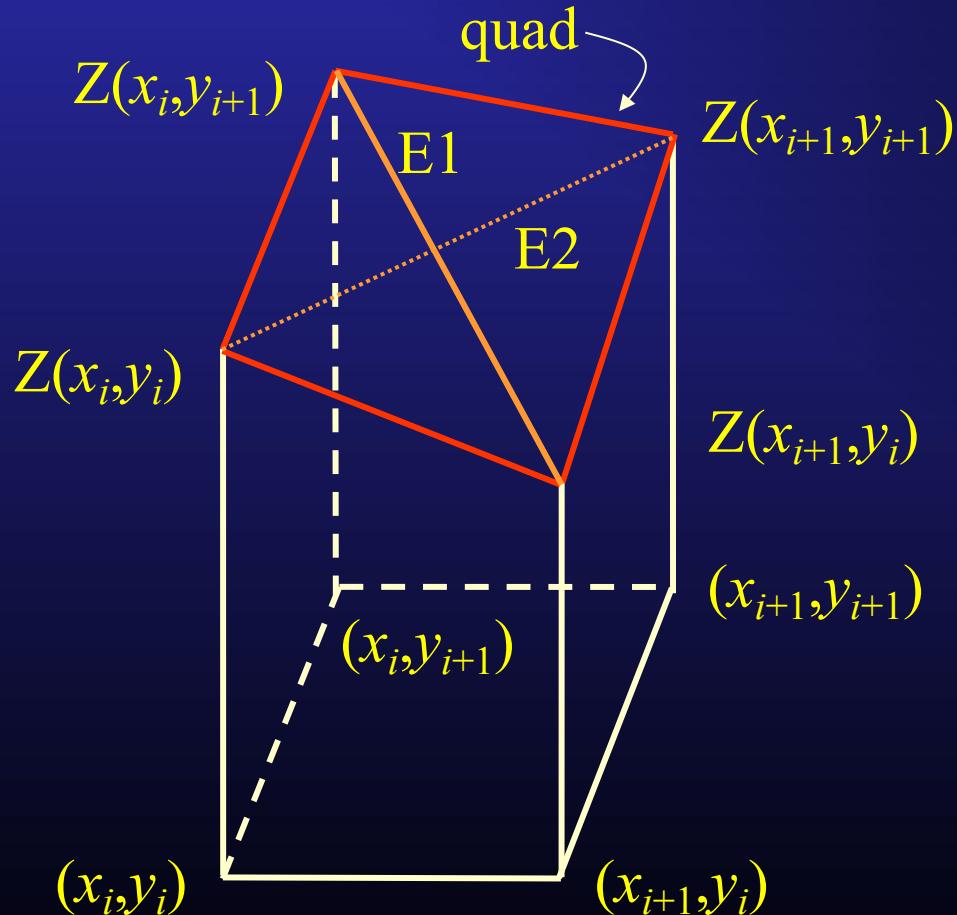


©1993, SUN MICROSYSTEMS, VIEWPOINT

Convert Height Fields to Polygon Meshes (Triangles) for Display



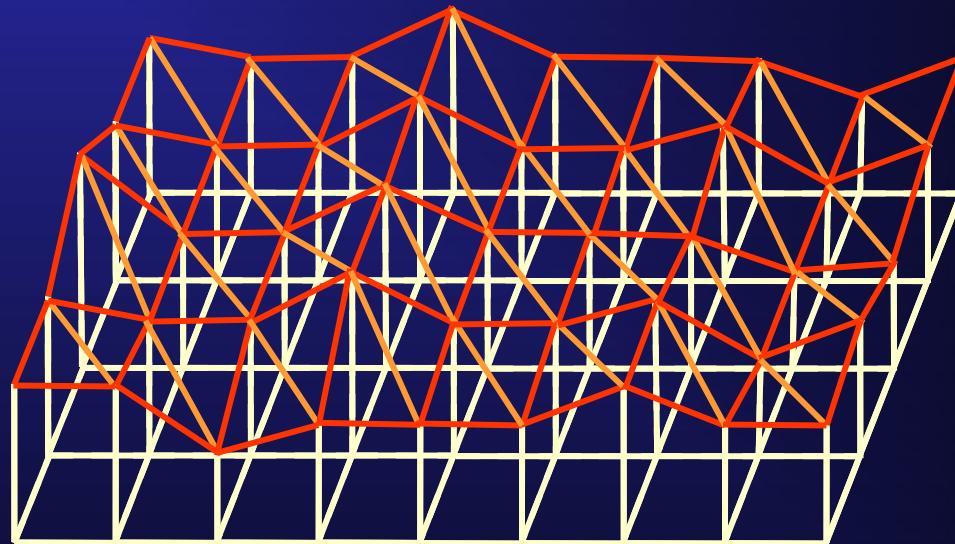
Triangulating Height Field Quadrilateral – Which is Very Likely to be Non-Planar



If quad is not planar, then split quad into 2 triangles, either:

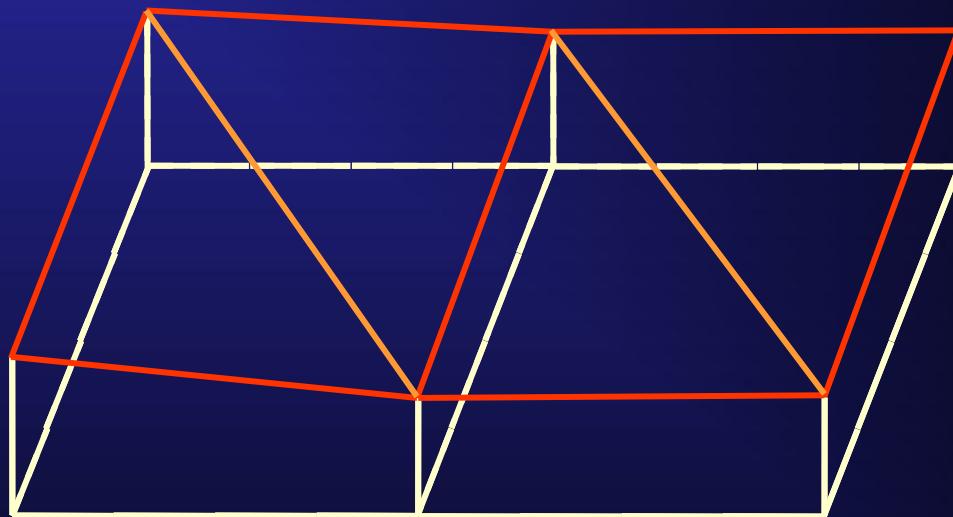
- a) Always the same way, (e.g., E1), or
- b) To make greatest dihedral angle (i.e., as flat as possible): dot product closer to 1 between triangle pair normals formed by splitting along E1 vs. along E2)

Polygon (Triangle) Mesh from Height Field



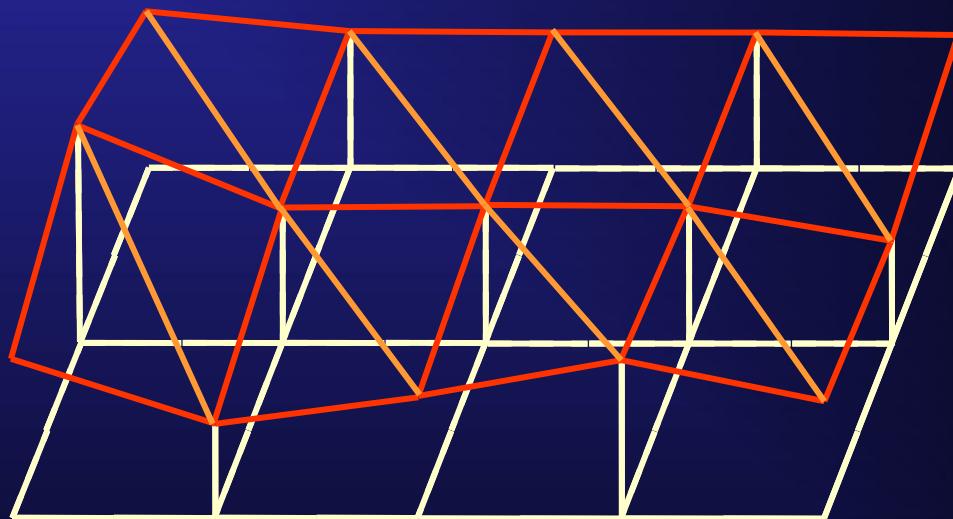
Quads split same direction, but not necessary.

Progressive Transmission of Height Fields to Maximize Real-Time Appearance



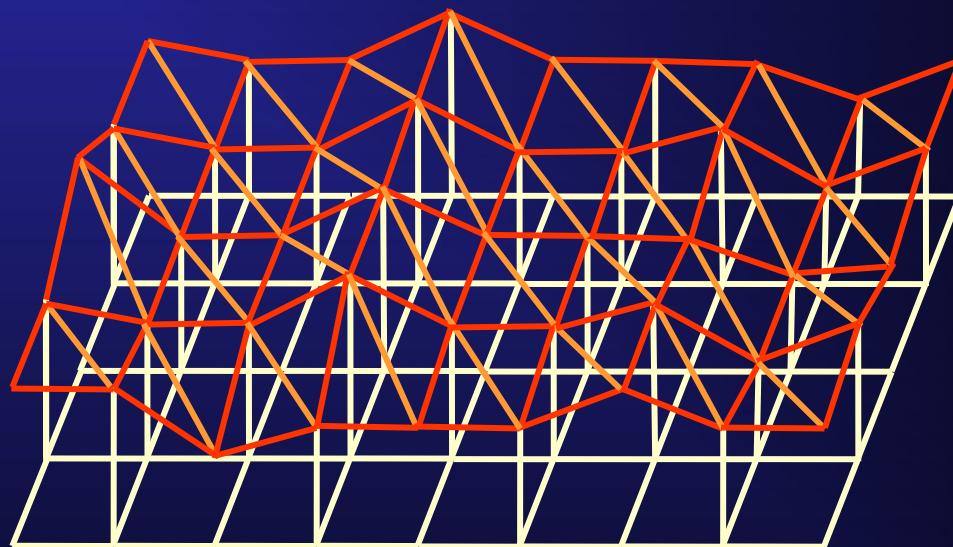
Increasing terrain level of detail, e.g., Google Earth.
E.g. 6 samples give general shape...

Progressive Transmission of Height Fields to Maximize Real-Time Appearance



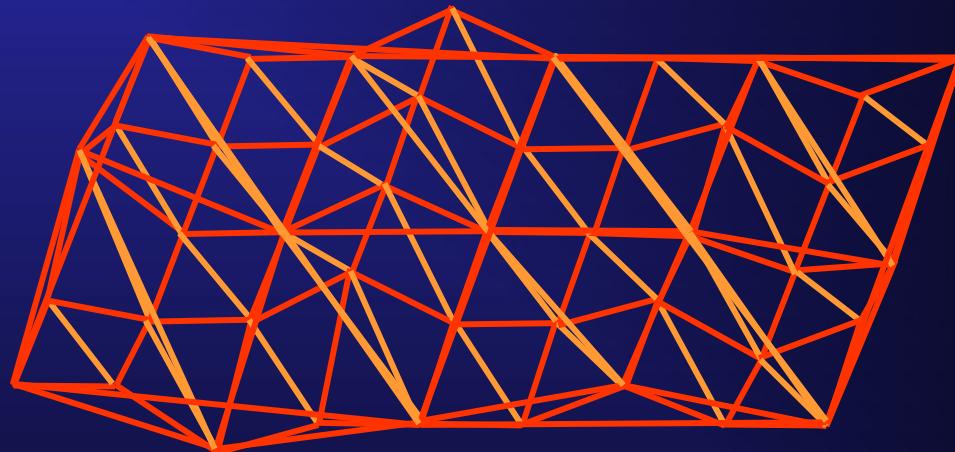
Increasing terrain level of detail, e.g., Google Earth.
9 more samples start to fill in details...

Progressive Transmission of Height Fields to Maximize Real-Time Appearance



Increasing terrain level of detail, e.g., Google Earth.
Remaining 30 samples provide highest resolution.

Progressive Transmission of Height Fields to Maximize Real-Time Appearance



Sent same 45 samples in all ($6 + 9 + 30$) “more uniformly” than if sent only at highest resolution.

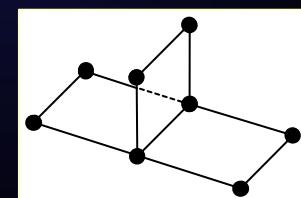
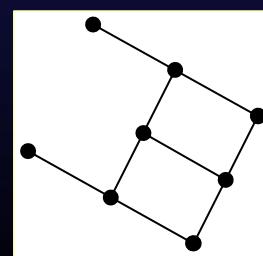
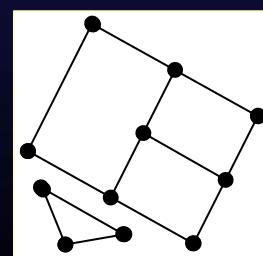
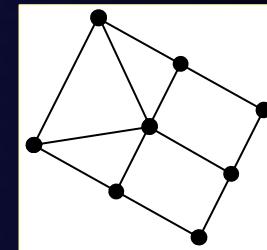
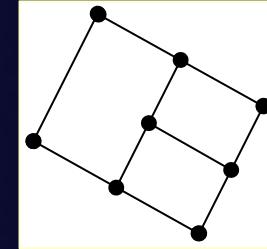
Representing General 3D Polygonal Meshes

- Definitions
- Simple Data Structures
- The HalfEdge Data Structure

Some slides in this section thanks to Justin Legakis

Well-Formed Surfaces

- Components intersect “properly”:
 - Faces are: disjoint, share single vertex, or share 2 vertices and the edge joining them
 - Every edge is incident to exactly 2 vertices
 - Every edge is incident to at most 2 faces
- Local topology is “proper”:
 - Neighborhood of a vertex is homeomorphic to a disk (permits stretching and bending, but not tearing)
 - (Called a 2-manifold)
- Global topology is “proper”:
 - Connected
 - Closed
 - Bounded



How might you make these 3 cases well-formed?

Computational Complexity Issues

- Support necessary geometric operations
 - Add, delete, modify: vertices, edges, faces in 3D
- Access Time
 - Linear, constant time average case, or constant time?
 - Requires loops/recursion/if ?
 - Avoid searching for stuff! (Who are my neighbors?)
- Memory
 - Variable size arrays or constant size?
- Maintenance
 - Ease of editing
 - Ensuring mesh topological consistency

Simple Data Structures

- List of Polygons
- List of Edges
- List of Unique Vertices & Indexed Faces
- Simple Adjacency Data Structure

List of Polygons (“Polygon Soup”)

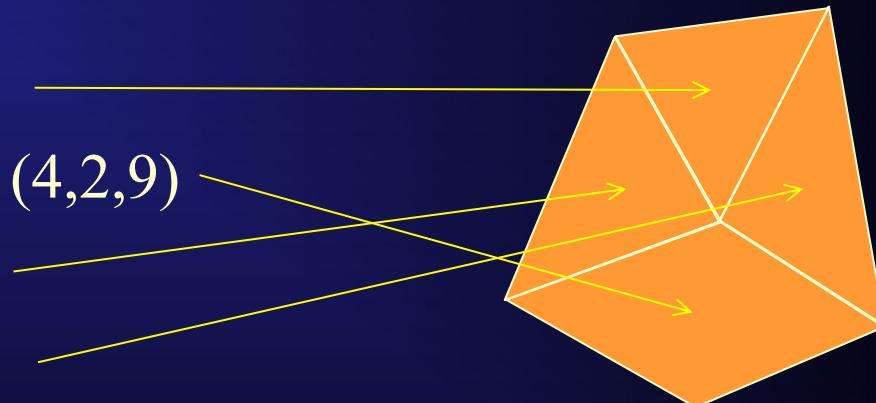
- Explicit vertex coordinates, listed in counter-clockwise order for each polygon face:

$(3,-2,5), (3,6,2), (-6,2,4)$

$(2,2,4), (-6,2,4), (9,4,0), (4,2,9)$

$(-6,2,4), (3,6,2), (9,4,0)$

$(2,2,4), (3,-2,5), (-6,2,4)$



No explicit edges, except by taking pairwise ordered vertices.

No explicit face adjacency.

List of Edges

- Explicit vertex coordinates, listed in pairs for each polygon edge:

(3,6,2), (-6,2,4)

(-6,2,4), (3,-2,5)

(3,-2,5), (3,6,2)

(9,4,0), (3,6,2)

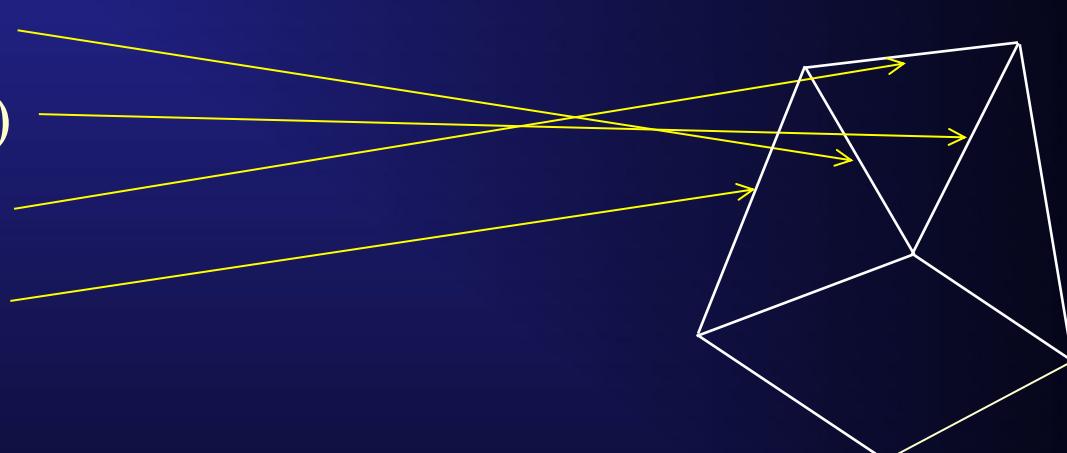
(9,4,0), (-6,2,4)

(9,4,0), (4,2,9)

(2,2,4), (4,2,9)

(2,2,4), (-6,2,4)

(2,2,4), (3,-2,5)



No explicit faces nor adjacency!

List of Unique Vertices & Indexed Faces

Vertices
(ordered in
array):

$(-1, -1, -1)$

$(-1, -1, 1)$

$(-1, 1, -1)$

$(-1, 1, 1)$

$(1, -1, -1)$

$(1, -1, 1)$

$(1, 1, -1)$

$(1, 1, 1)$

Faces (hold
vertex array
indices):

1 2 4 3

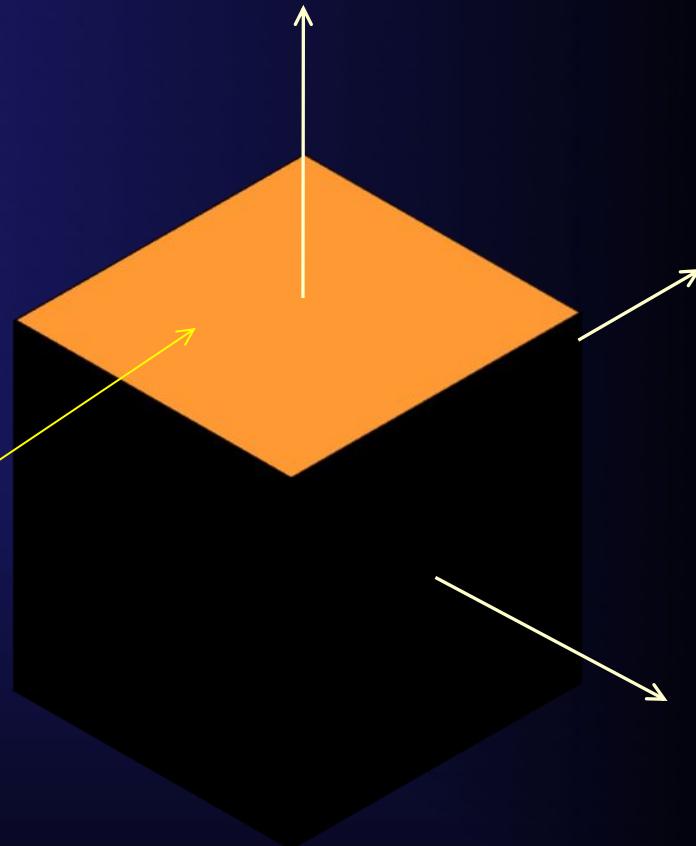
5 7 8 6

1 5 6 2

3 4 8 7

1 3 7 5

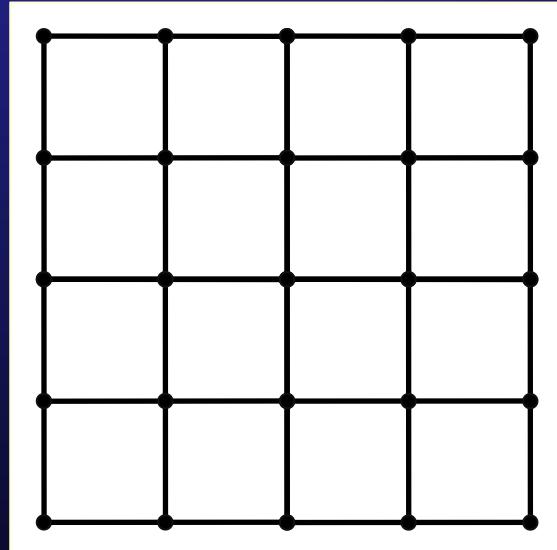
2 6 8 4



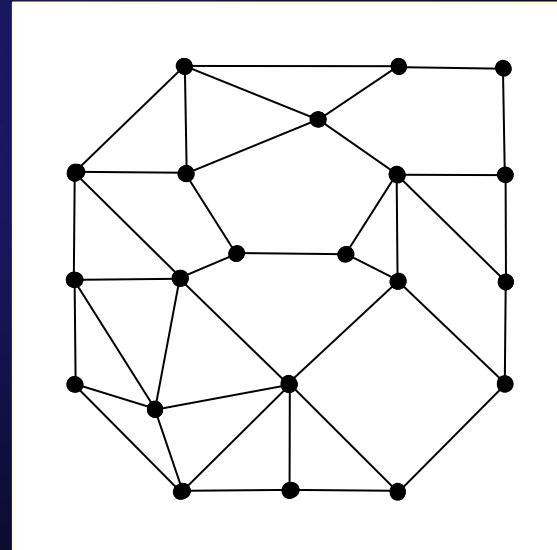
This is essentially the Maya OBJ file format.

Problems with Simple Data Structures

- No adjacency information.
- Linear-time searches for some things but not others.



Structured



Unstructured

- Adjacency is implicit (e.g., in array indices) for structured meshes, but what do we do for unstructured meshes?

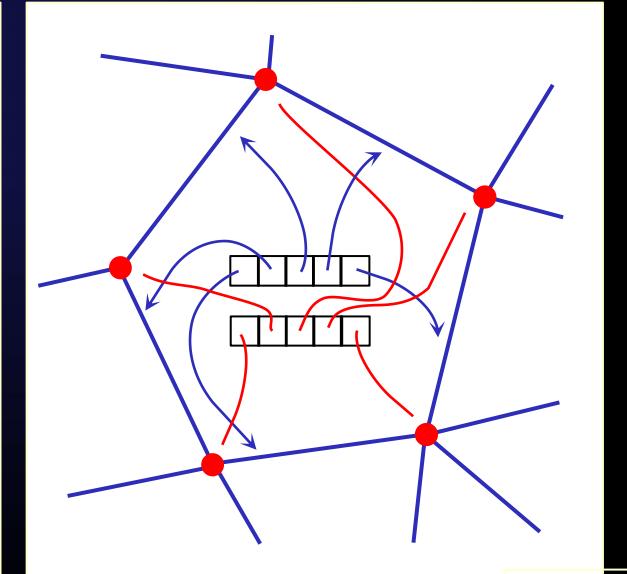
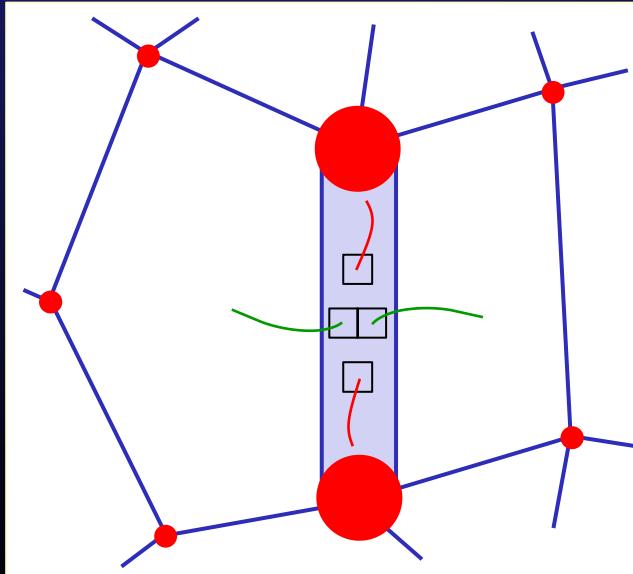
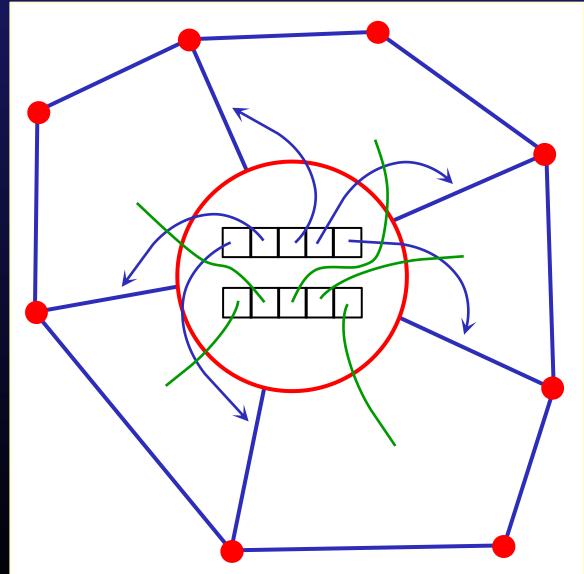
General Mesh Data

- So, in addition to
 - Geometric information (position coordinates) on vertices
 - Attribute information (color, texture, temperature, population density, etc.) on vertices or faces
- Let's store:
 - Topological information (adjacency, connectivity)

Simple (Naïve) Adjacency

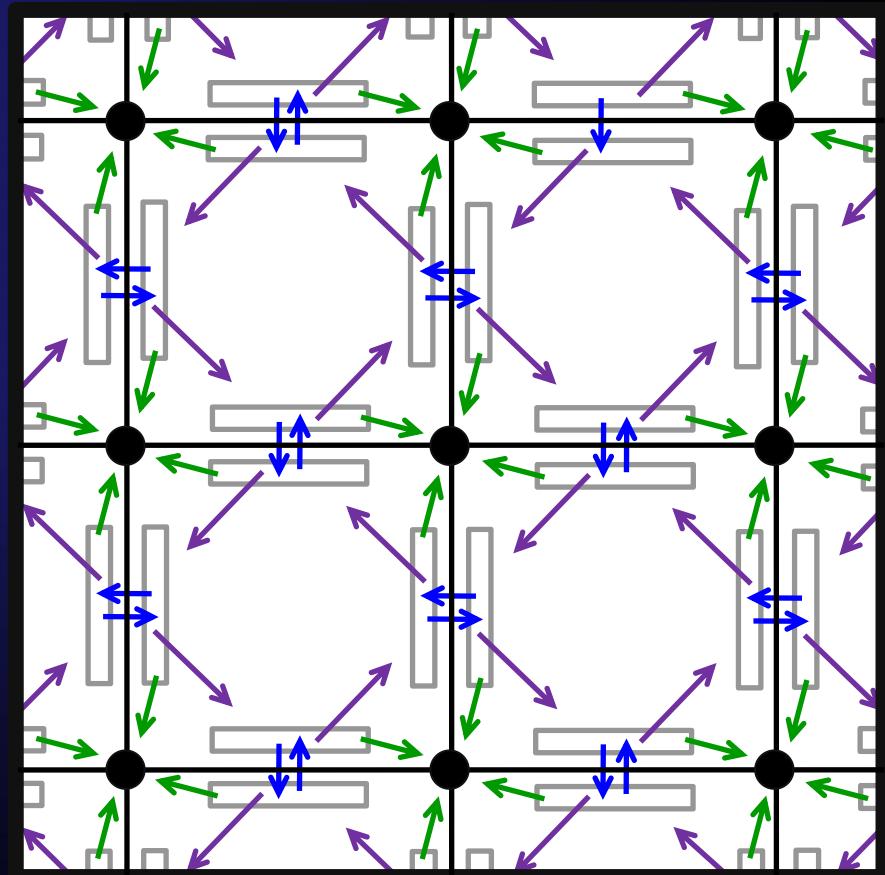
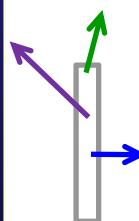
- Each element (vertex, edge, and face) has a list of pointers to all incident elements
 - Queries depend only on local complexity of mesh
 - Data structures do not have fixed size
 - Slow! Big! Too much work to maintain!

$V \rightarrow \{\text{adj } E\}$ and $V \rightarrow \{\text{adj } F\}$ $E \rightarrow \{F_l, F_r\}$ and $E \rightarrow \{V_a, V_b\}$ $F \rightarrow \{\text{adj } E\}$ and $F \rightarrow \{\text{adj } V\}$



Half_Edge Data Structure for General Polygonal Meshes*

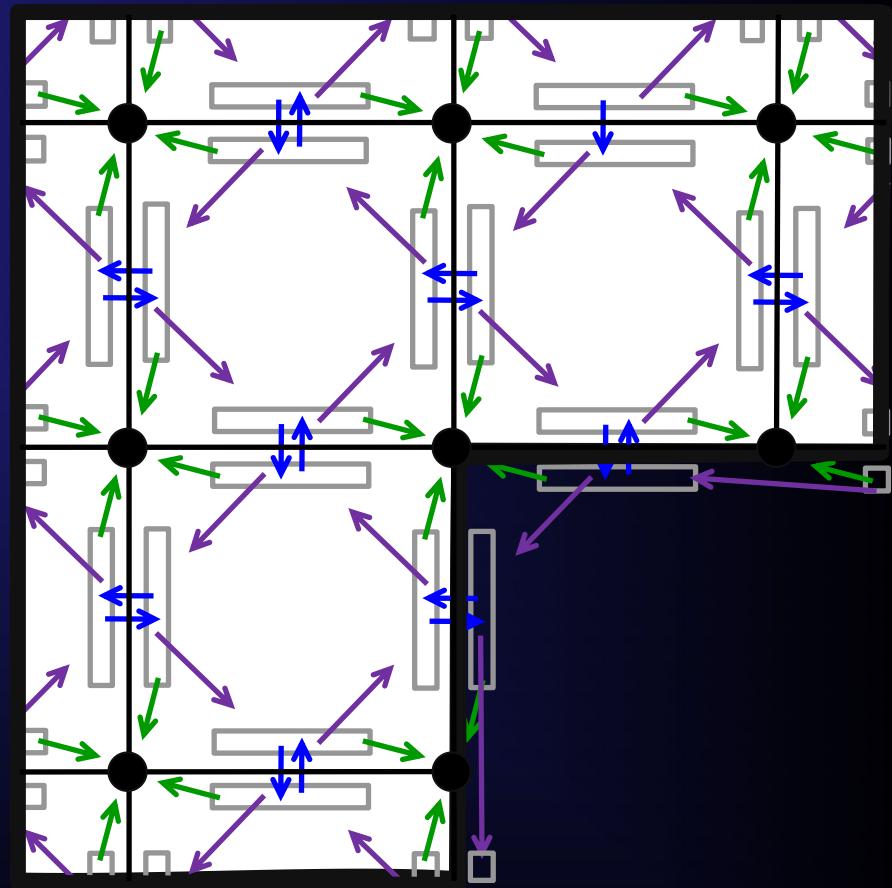
- Every **edge** is represented by two directed **Half_Edge** structures
- Each **Half_Edge** stores:
 - Face to left of edge
 - **Vertex** at end of directed edge
 - **Symmetric half edge**
 - **Next** points to the **Half_Edge** counter-clockwise around face on left
- Polygon orientation is essential, and must be done consistently.



*Eastman,C.M., Weiss,S.F. "Tree structures for high dimensionality nearest neighbor searching." Information Systems, vol. 7, no. 2 (1982)

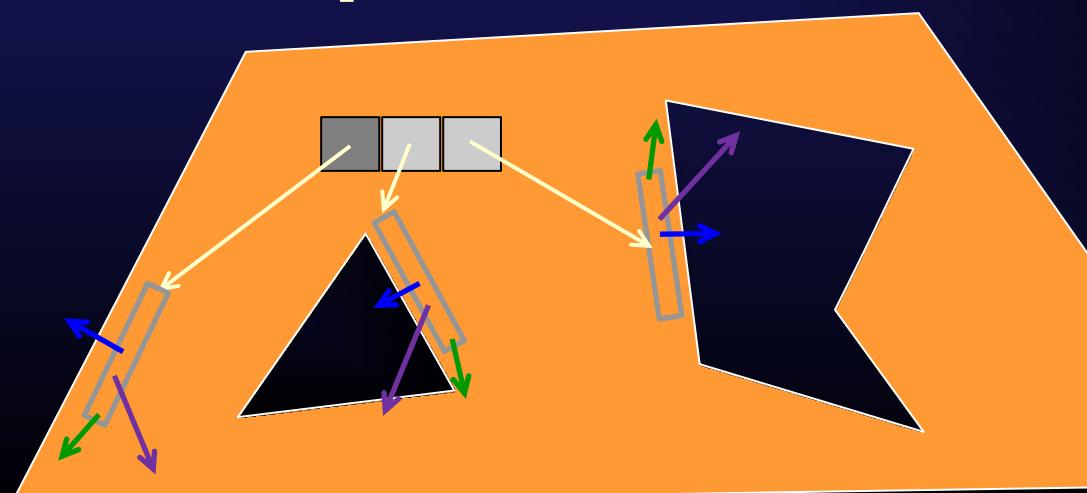
Want Happens at an External Boundary?

- If an edge lies on the mesh boundary, we still need BOTH Half_Edge blocks, but the outside one has a NIL Face pointer.
- Notice that the external boundary is fully linked via the Half_Edge Next pointers.



Completing the Half_Edge Data Structure

- Polygons can have **any number** of vertices (arbitrary mesh).
- Face pointer is **NIL** for edges on outside of boundary. (These Half_Edges are not drawn below, but they are there!)
- Each vertex object points to **any one** Half_Edge that surrounds it.
- Each face object points to **any one** Half_Edge on the face outer loop.
- Each face object stores **one** pointer to **any one** Half_Edge on **each** face inner loop (if it has any): i.e., a list of internal loops. [Though we won't use this here. Also note that inner loops are traced *opposite* to outer loop, i.e., clockwise.]



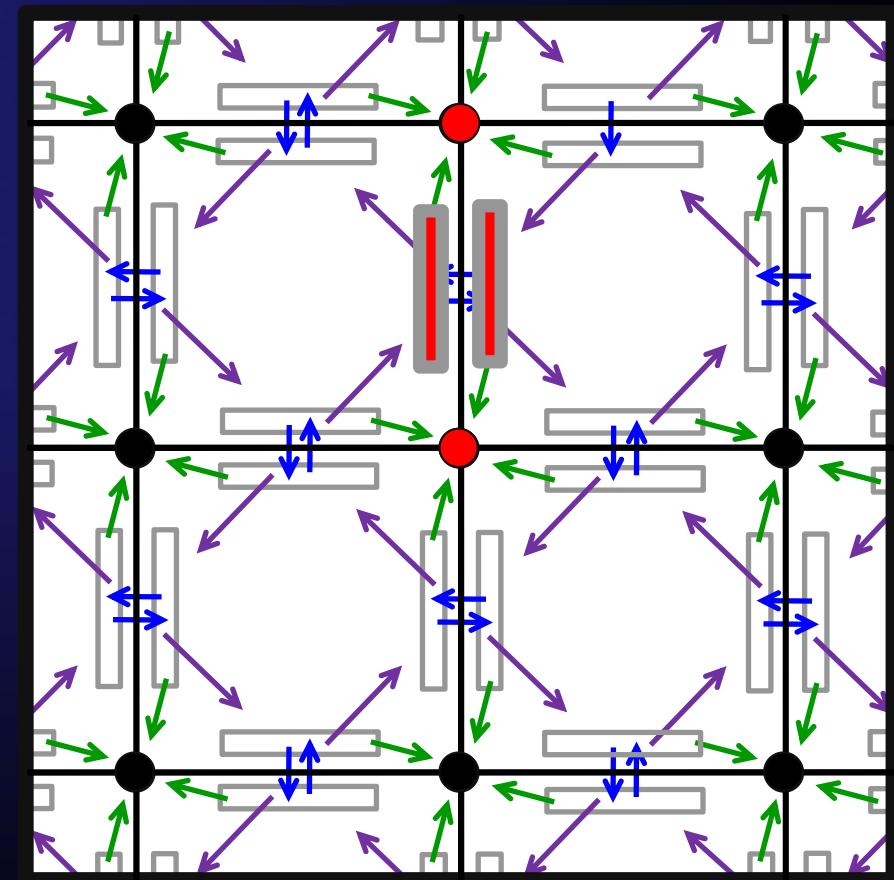
Half_Edge Data Structure: Sample Classes

Create classes for FACE, VERTEX, and Half_Edge:

- FACE is {Half_Edge, Color, ID}
 - Half_Edge is the (arbitrary) starting Half_Edge for this FACE
 - Color is {R, G, B}
 - ID is an identifying integer (for menus)
- VERTEX is {X, Y, Z, Half_Edge, ID}
 - X, Y, Z are float coordinates
 - Half_Edge is the (arbitrary) starting Half_Edge for this VERTEX
 - ID is an identifying integer (e.g., for menus)
- Half_Edge is {F, V, NEXT, SYM}
 - F points to a FACE
 - V points to a VERTEX
 - NEXT and SYM point to Half_Edge.

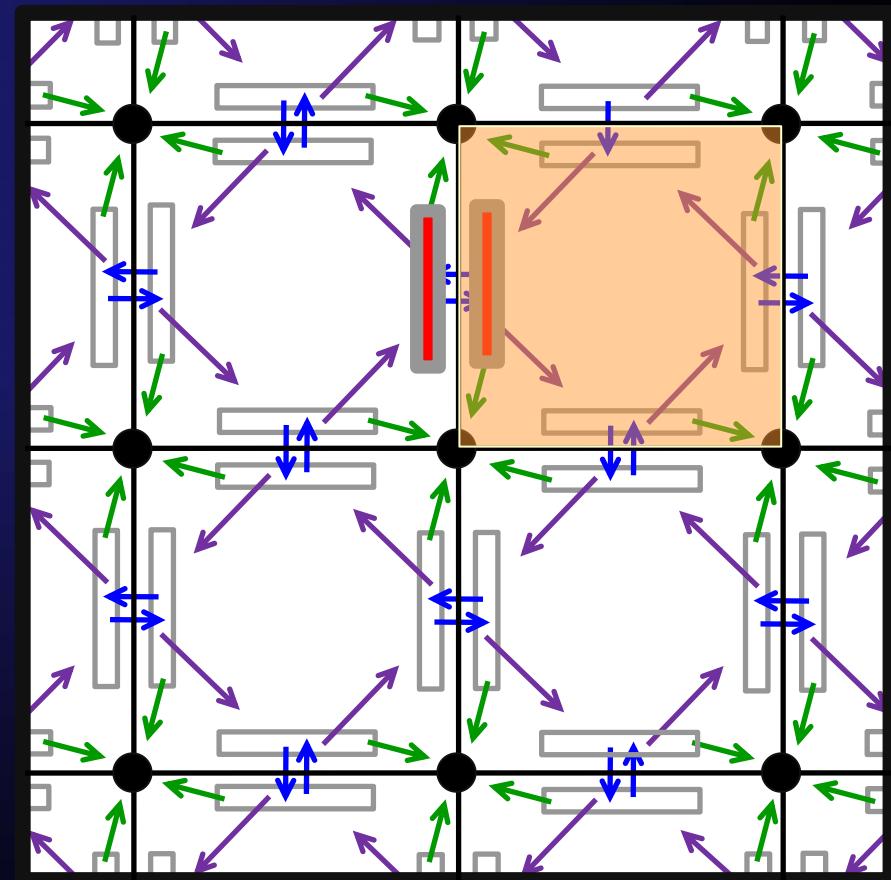
Half_Edge Access

- Starting at Half_Edge HE, how do we find:
 - the other vertex of the edge?
 - the other face of the edge?
 - the counter-clockwise edge around the face at the left?
 - all the edges surrounding the face at the left?
 - all the faces surrounding a vertex?



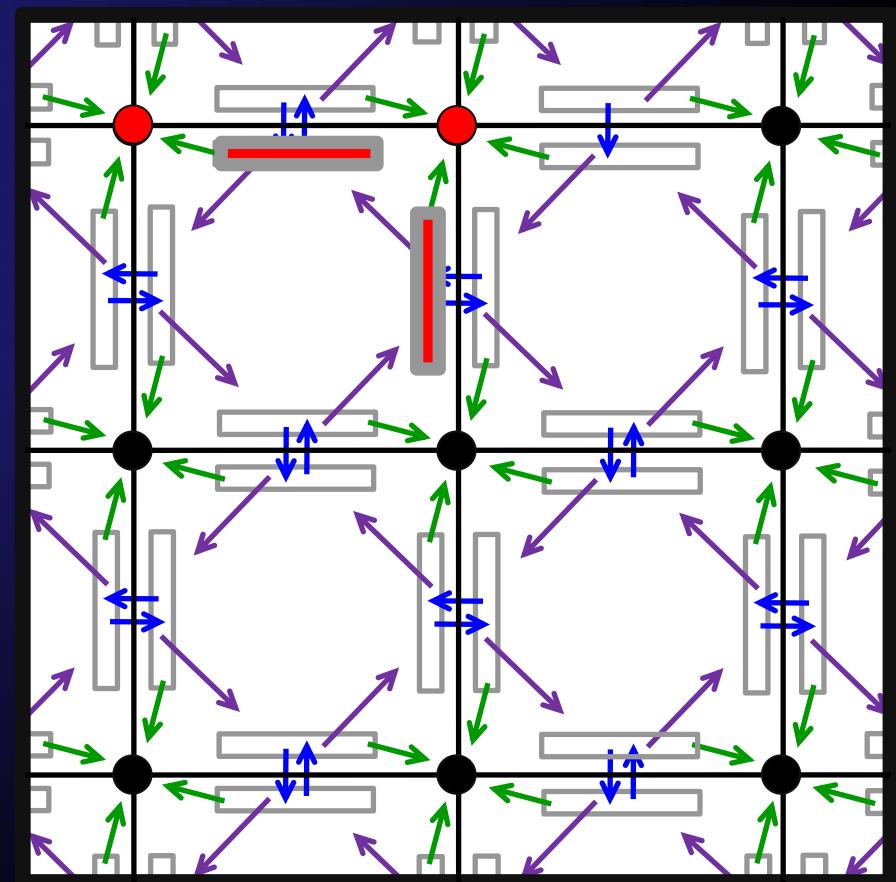
Half_Edge Access

- Starting at Half_Edge HE, how do we find:
 - the other vertex of the edge?
 - the other face of the edge?
 - the counter-clockwise edge around the face at the left?
 - all the edges surrounding the face at the left?
 - all the faces surrounding a vertex?



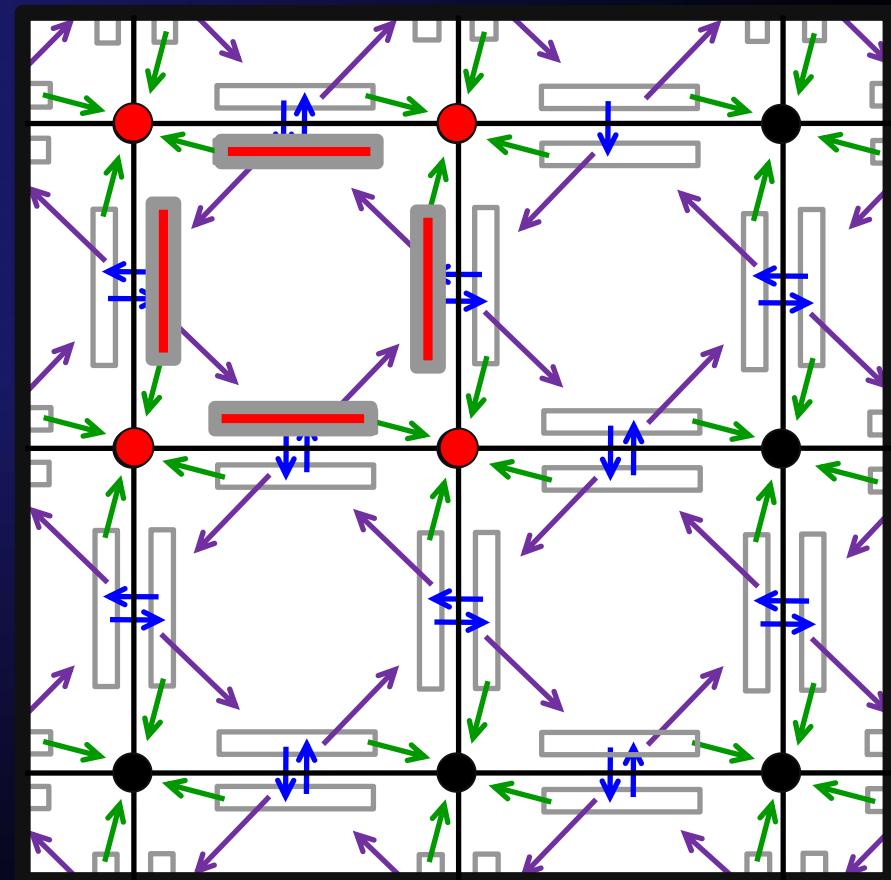
Half_Edge Access

- Starting at Half_Edge HE, how do we find:
 - the other vertex of the edge?
 - the other face of the edge?
 - the counter-clockwise edge around the face at the left?
 - all the edges surrounding the face at the left?
 - all the faces surrounding a vertex?



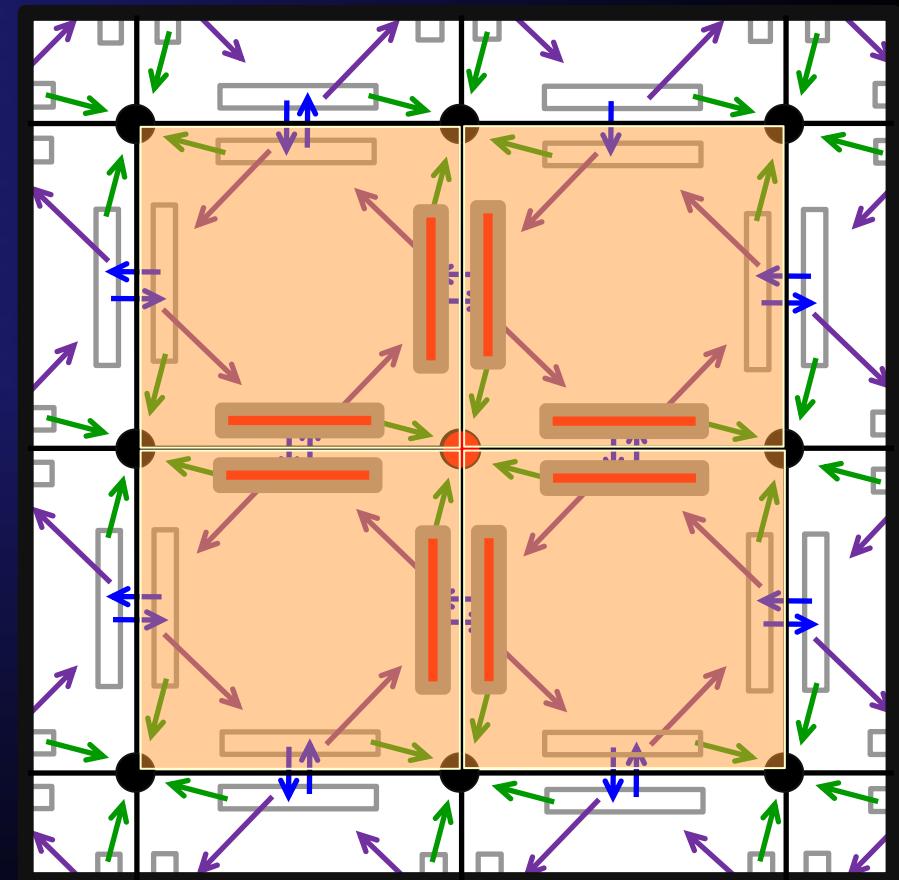
Half_Edge Access

- Starting at Half_Edge HE, how do we find:
 - the other vertex of the edge?
 - the other face of the edge?
 - the counter-clockwise edge around the face at the left?
 - all the edges surrounding the face at the left?
 - all the faces surrounding a vertex?



Half_Edge Access

- Starting at Half_Edge HE, how do we find:
 - the other vertex of the edge?
 - the other face of the edge?
 - the counter-clockwise edge around the face at the left?
 - all the edges surrounding the face at the left?
 - all the faces surrounding a vertex?



Sample Traversal Code

- Loop around a face (e.g., visit vertices of HE->F):

```
Half_Edge_Mesh::FaceLoop(Half_Edge *HE) {  
    Half_Edge *loop = HE;  
    do {    // e.g., do something with loop->V  
        loop = loop->NEXT;  
    } while (loop != HE); // note no IF statements //  
}
```

- Loop around a vertex (e.g., visit adjacent faces to HE->V):

```
Half_Edge_Mesh::VertexLoop(Half_Edge *HE) {  
    Half_Edge *loop = HE;  
    do {    // e.g., do something with loop->F  
        loop = loop->NEXT->SYM;  
    } while (loop != HE); // note no IF statements //  
}
```

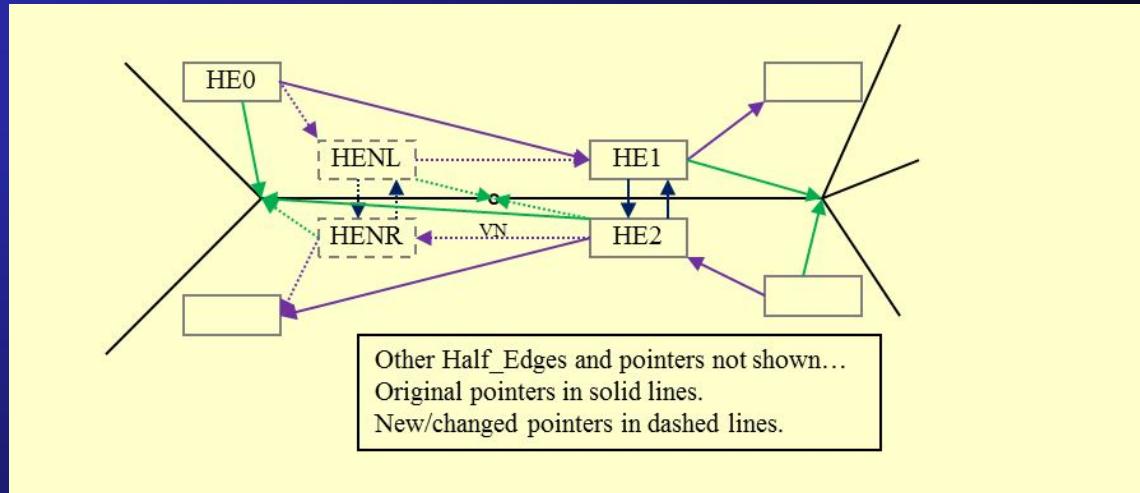
Half_Edge Features and Advantages

- Data Structure Size?
 - Fixed size elements
- Data:
 - Geometric information stored at vertices.
 - Attribute information in vertices, Half_Edges, and/or faces.
 - Topological information in Half_Edges only!
- Orientable surfaces only (no Möbius strips!)
- Local consistency everywhere implies global consistency.
- Time complexity?
 - Linear for all local information (gathering face, edge, or vertex lists).
 - Independent of global mesh complexity.

Half_Edge Operation Examples

- Add a vertex into an edge.
- Split a quad into two triangles.
- Add an edge between two faces that share a vertex.

Add a vertex into an edge



Input: edge starting at vertex HE0->V; new VERTEX VN

Goal: Insert VN on edge just after HE0->V

```

Half_Edge *HE1 = HE0->NEXT; //HE1 next edge
Half_Edge to           get next
vertex//               vertex
Half_Edge *HE2 = HE1->SYM; //HE2 other side of edge
//                   //

Half_Edge *HENL = new Half_Edge(); //HE New Left//
Half_Edge *HENR = new Half_Edge(); //HE New
Right//
```

```

HENL->NEXT = HE1;
HE0->NEXT = HENL;
HENR->NEXT = HE2->NEXT;
HE2->NEXT = HENR.
```

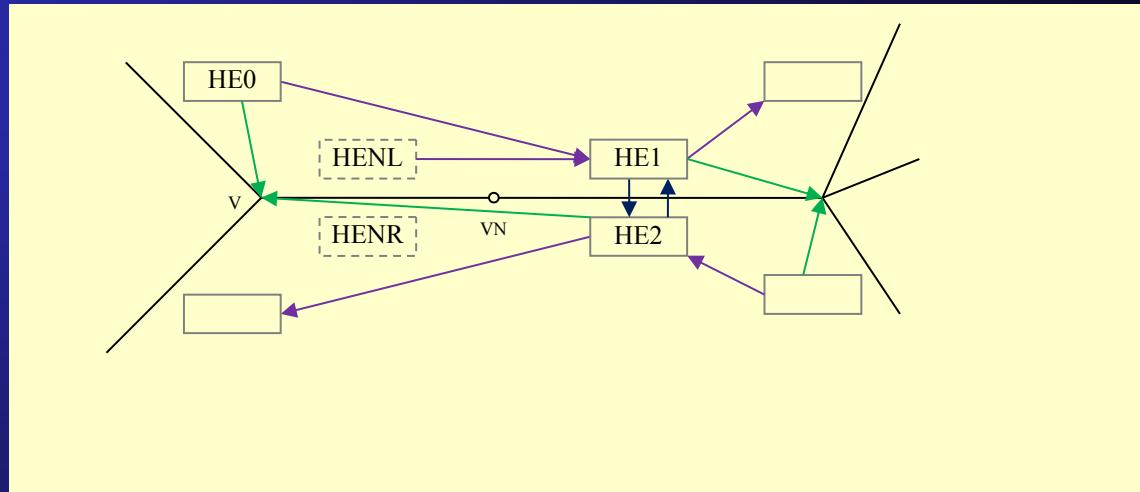
HENL->SYM = HENR;
HENR->SYM= HENL;

HENL->F = HE0->F;
HENR->F = HE2->F;

HENL->V = VN;
HENR-> V = HE2->V;
HE2->V = VN;

VN (X,Y,Z) = average(HE0->V, HE1->V);

Add a vertex into an edge



Input: edge starting at vertex HE0->V

Goal: Insert new vertex VN on edge just after HE0->V

```

Half_Edge *HE1 = HE0->NEXT; //HE1 next edge
Half_Edge to           get next
vertex//                vertex
Half_Edge *HE2 = HE1->SYM; //HE2 other side of edge
//  

VERTEX *VN = new VERTEX();
Half_Edge *HENL = new Half_Edge(); //HE New Left//
Half_Edge *HENR = new Half_Edge(); //HE New
Right//
```

```

HENL->NEXT = HE1;
HE0->NEXT = HENL;
HENR->NEXT = HE2->NEXT;
```

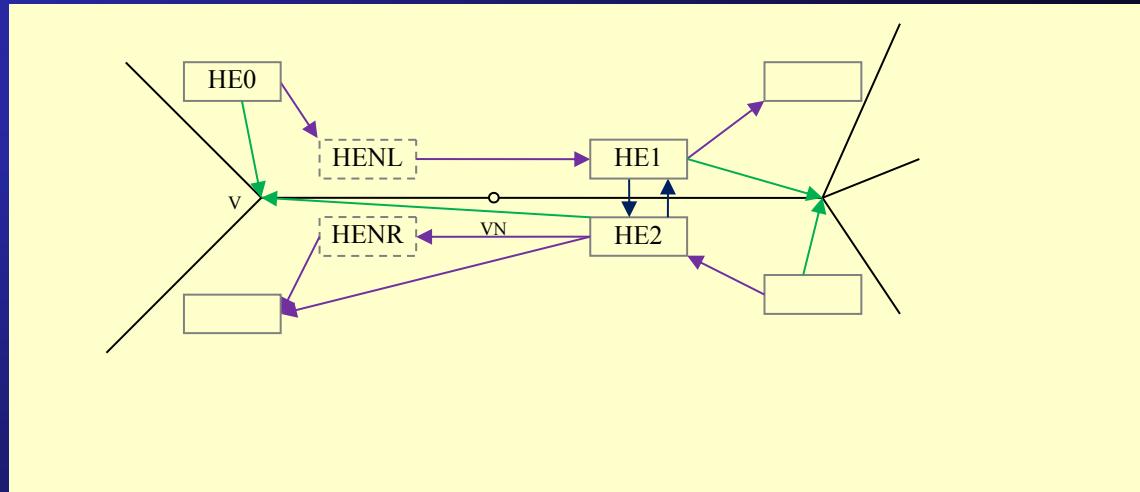
HENL->SYM = HENR;
HENR->SYM= HENL;

HENL->F = HE0->F;
HENR->F = HE2->F;

HENL->V = VN;
HENR-> V = HE2->V;
HE2->V = VN;

VN (X,Y,Z) = average(HE0->V, HE1->V);

Add a vertex into an edge



Input: edge starting at vertex HE0->V

Goal: Insert new vertex VN on edge just after HE0->V

```

Half_Edge *HE1 = HE0->NEXT; //HE1 next edge
Half_Edge to           get next
vertex//                vertex
Half_Edge *HE2 = HE1->SYM; //HE2 other side of edge
//  

VERTEX *VN = new VERTEX();
Half_Edge *HENL = new Half_Edge(); //HE New Left//
Half_Edge *HENR = new Half_Edge(); //HE New
Right//
```

```

HENL->NEXT = HE1;
HE0->NEXT = HENL;
HENR->NEXT = HE2->NEXT;
```

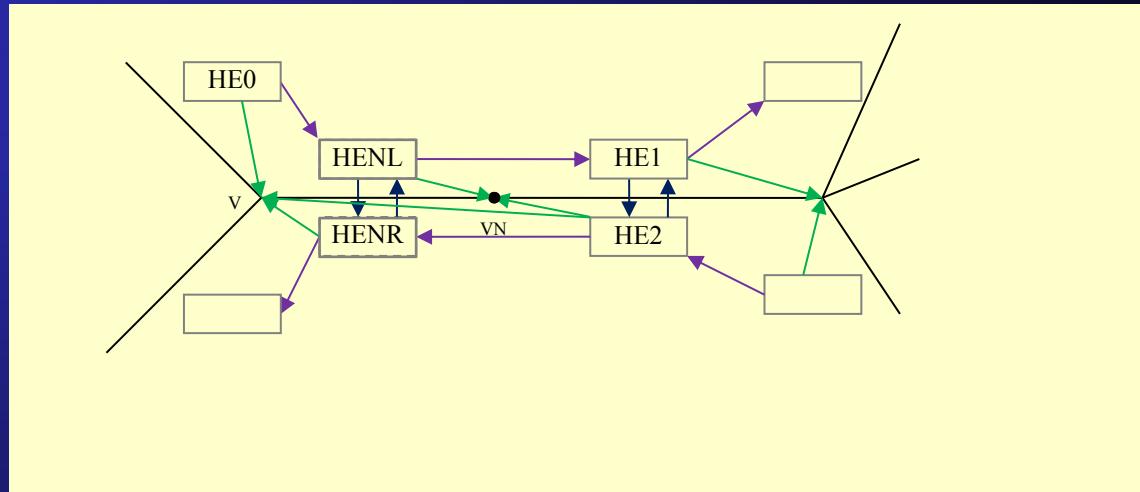
HENL->SYM = HENR;
HENR->SYM= HENL;

HENL->F = HE0->F;
HENR->F = HE2->F;

HENL->V = VN;
HENR-> V = HE2->V;
HE2->V = VN;

VN (X,Y,Z) = average(HE0->V, HE1->V);

Add a vertex into an edge



Input: edge starting at vertex HE0->V

Goal: Insert new vertex VN on edge just after HE0->V

```

Half_Edge *HE1 = HE0->NEXT; //HE1 next edge
Half_Edge to
vertex//  

Half_Edge *HE2 = HE1-> SYM; //HE2 other side of edge
//  

VERTEX *VN = new VERTEX();
Half_Edge *HENL = new Half_Edge(); //HE New Left//
Half_Edge *HENR = new Half_Edge(); //HE New
Right//
```

```

HENL->NEXT = HE1;
HE0->NEXT = HENL;
HENR->NEXT = HE2->NEXT;
```

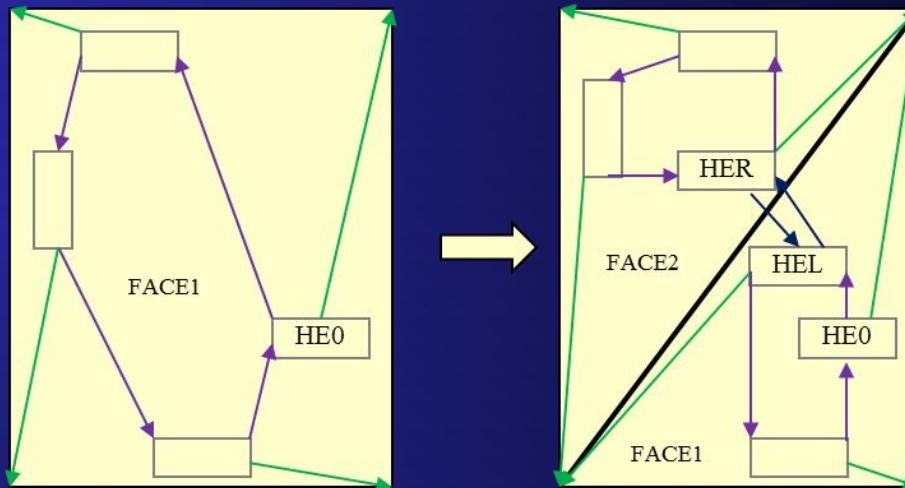
HENL->SYM = HENR;
HENR->SYM= HENL;

HENL->F = HE0->F;
HENR->F = HE2->F;

HENL->V = VN;
HENR-> V = HE2->V;
HE2->V = VN;

VN (X,Y,Z) = average(HE0->V, HE1->V);

Split a quad into two triangles



Input: FACE FACE1

Check that FACE1 has exactly 4 edges, else return;

```
HE0 = FACE1->HE; //starting edge//
```

```
FACE *FACE2 = new FACE();
```

```
FACE2->COLOR = FACE1->COLOR;
```

```
Half_Edge *HEL = new Half_Edge(); //HE New Left //
```

```
Half_Edge *HER = new Half_Edge(); //HE New Right//
```

```
HEL->F = FACE1;
```

```
HER->F = FACE2;
```

```
FACE2->HE = HER; //FACE1 still points to its starting edge HE0, //
//and will contain the new HEL in its loop; //
```

```
//new FACE2 points to HER as its new starting edge//
```

$\text{HEL} \rightarrow \text{SYM} = \text{HER};$
 $\text{HER} \rightarrow \text{SYM} = \text{HEL};$

$\text{HEL} \rightarrow \text{V} = \text{HE0} \rightarrow \text{NEXT} \rightarrow \text{NEXT} \rightarrow \text{V};$

$\text{HEL} \rightarrow \text{NEXT} = \text{HE0} \rightarrow \text{NEXT} \rightarrow \text{NEXT} \rightarrow \text{NEXT};$

$\text{HER} \rightarrow \text{NEXT} = \text{HE0} \rightarrow \text{NEXT};$

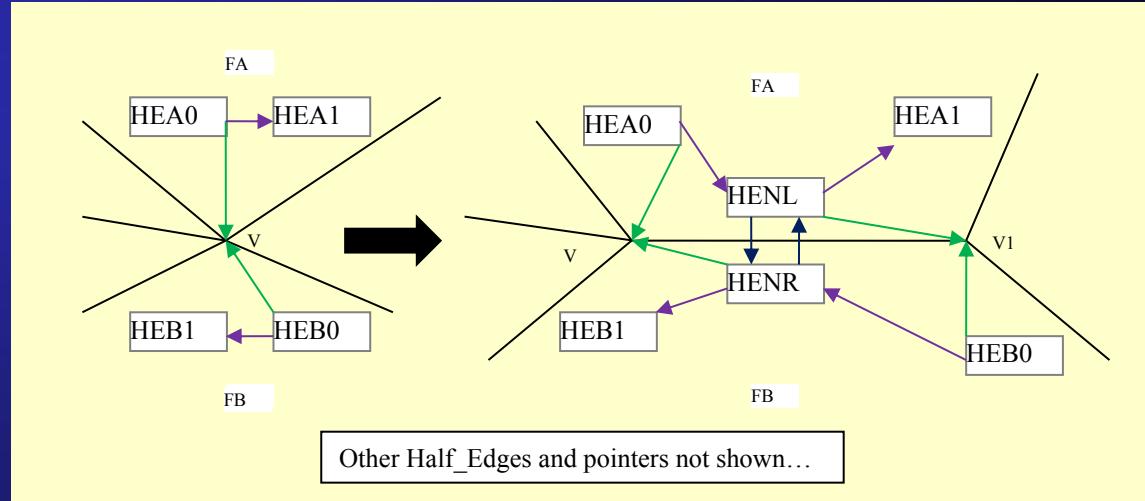
$\text{HER} \rightarrow \text{NEXT} \rightarrow \text{NEXT} \rightarrow \text{NEXT} = \text{HER};$

$\text{HE0} \rightarrow \text{NEXT} = \text{HEL};$

$\text{HER} \rightarrow \text{V} = \text{HE0} \rightarrow \text{V};$

// Note that the other faces adjacent to FACE1
that now abut FACE2 are automatically “OK”
because the SYM pointers on the original
Half_Edges of FACE1 are all still correct!

Add an edge between two faces that share a vertex



Input: FACES FA and FB, and VERTEX HEA0->V and HEB0->V.

If HEA0->V != HEB0->V error return;

VERTEX *V1 = New VERTEX();

// set X, Y, Z values for V1, say = V for a start //

// new edge will be of length 0, but user can change //

Half_Edge *HENL = new Half_Edge(); //HE New Left //

Half_Edge *HENR = new Half_Edge(); //HE New Right//

$HENL \rightarrow NEXT = HEA_0 \rightarrow NEXT;$
 $HEA_0 \rightarrow NEXT = HENL;$
 $HENL \rightarrow V = V_1;$
 $HENL \rightarrow F = FA;$
 $HENL \rightarrow SYM = HENR;$

$HENR \rightarrow NEXT = HEB_0 \rightarrow NEXT;$
 $HEB_0 \rightarrow NEXT = HENR;$
 $HENR \rightarrow V = HEA_0 \rightarrow V;$
 $HEB_0 \rightarrow V = V_1;$
 $HENR \rightarrow F = FB;$
 $HENR \rightarrow SYM = HENL;$

[Other HE vertex pointer adjustments will be needed on faces sharing new vertex V1]

Fractals

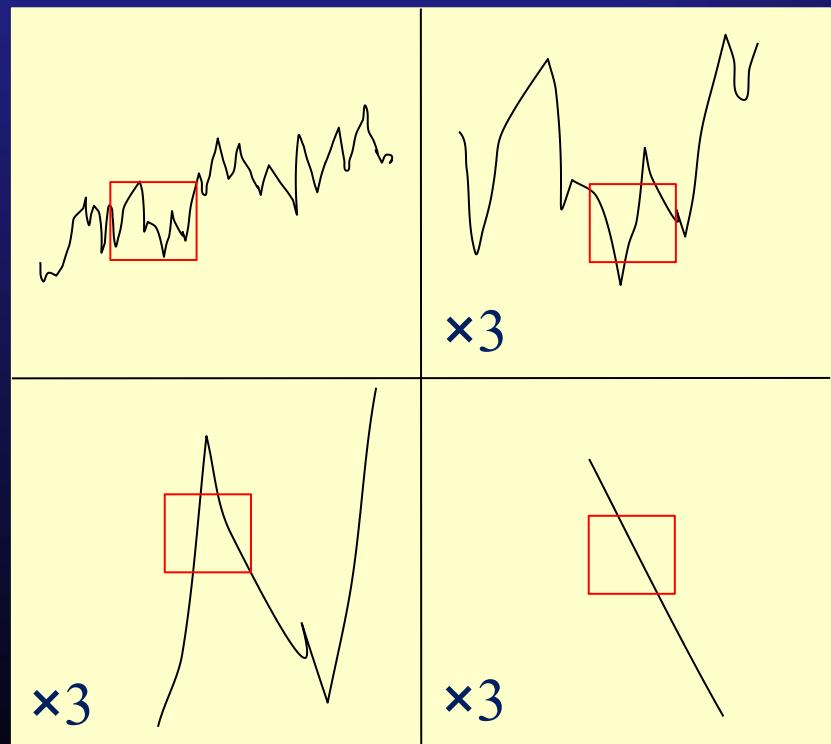
- Procedural method starting with a flat height field or one “seeded” with rough notion of the shapes desired.
- Decompose height field polygon models into finer details.
- Fractals use “random” process: “sub-divide and displace”.
- True fractals are statistically **self-similar**: this is, at whatever scale we observe the surface it has the **same statistical distribution of shapes or displacements**.
- In practice, surface is implicit and computed only when needed for rendering (e.g., when a ray intersects the surface the required height and normal is then computed).

Fractal Essence



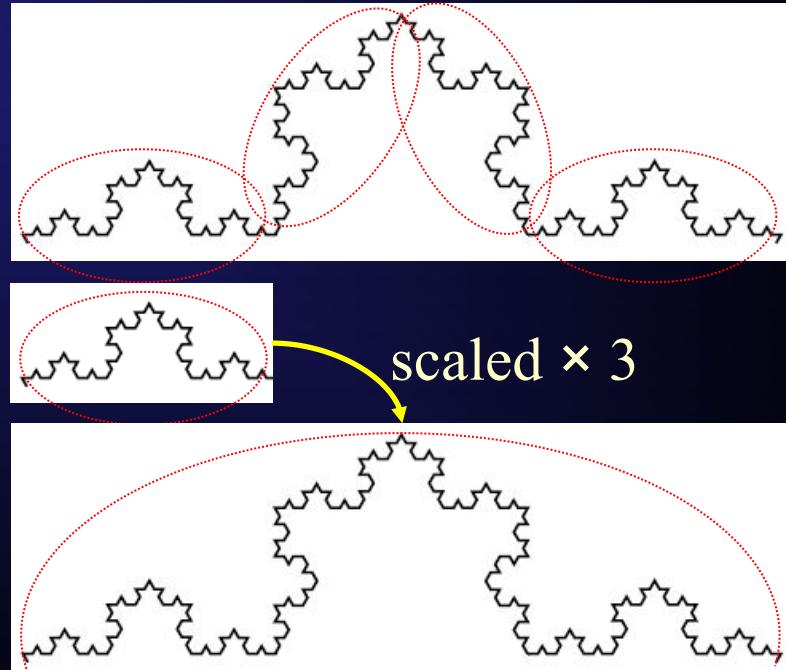
Euclidian Geometry

- Shapes change with scaling



Fractal Geometry

- Shapes remain similar under scaling (self-similar)

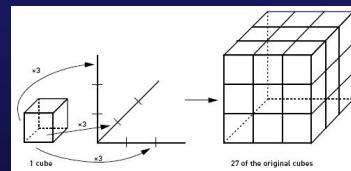
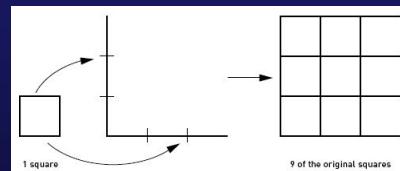


Adapted from Christoph Traxler

Fractal (Hausdorff) Dimension

Euclidean: Integer dimension

| | Scale factor (S) | N self-similar copies | Exponent D to get SS copies |
|-------------------|------------------|-----------------------|-----------------------------|
| Line segment (1D) | 1/3 | 3 | 1 |
| Square (2D) | 1/3 | 9 | 2 |
| Cube (3D) | 1/3 | 27 | 3 |



N = number of self-similar copies

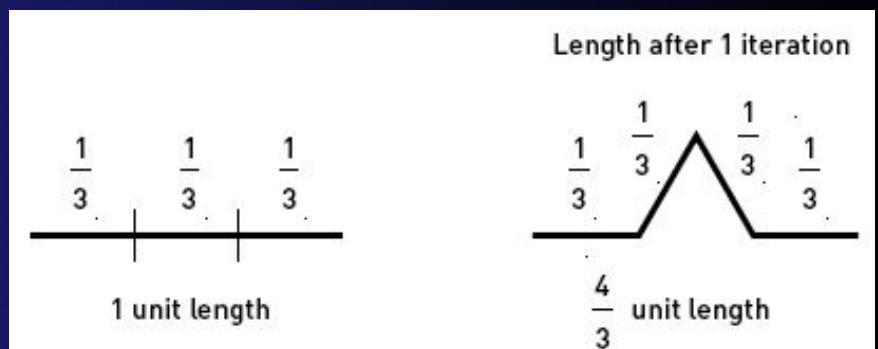
S = Scale factor

D = Dimension

$$N = 1/(S^D)$$

e.g., for cube, $S=1/3$, $D=3$, so $N=27$

Fractal: Real number dimension



$$\ln(N) = D \ln(1/S) \quad [\text{log base is irrelevant}]$$

thus:

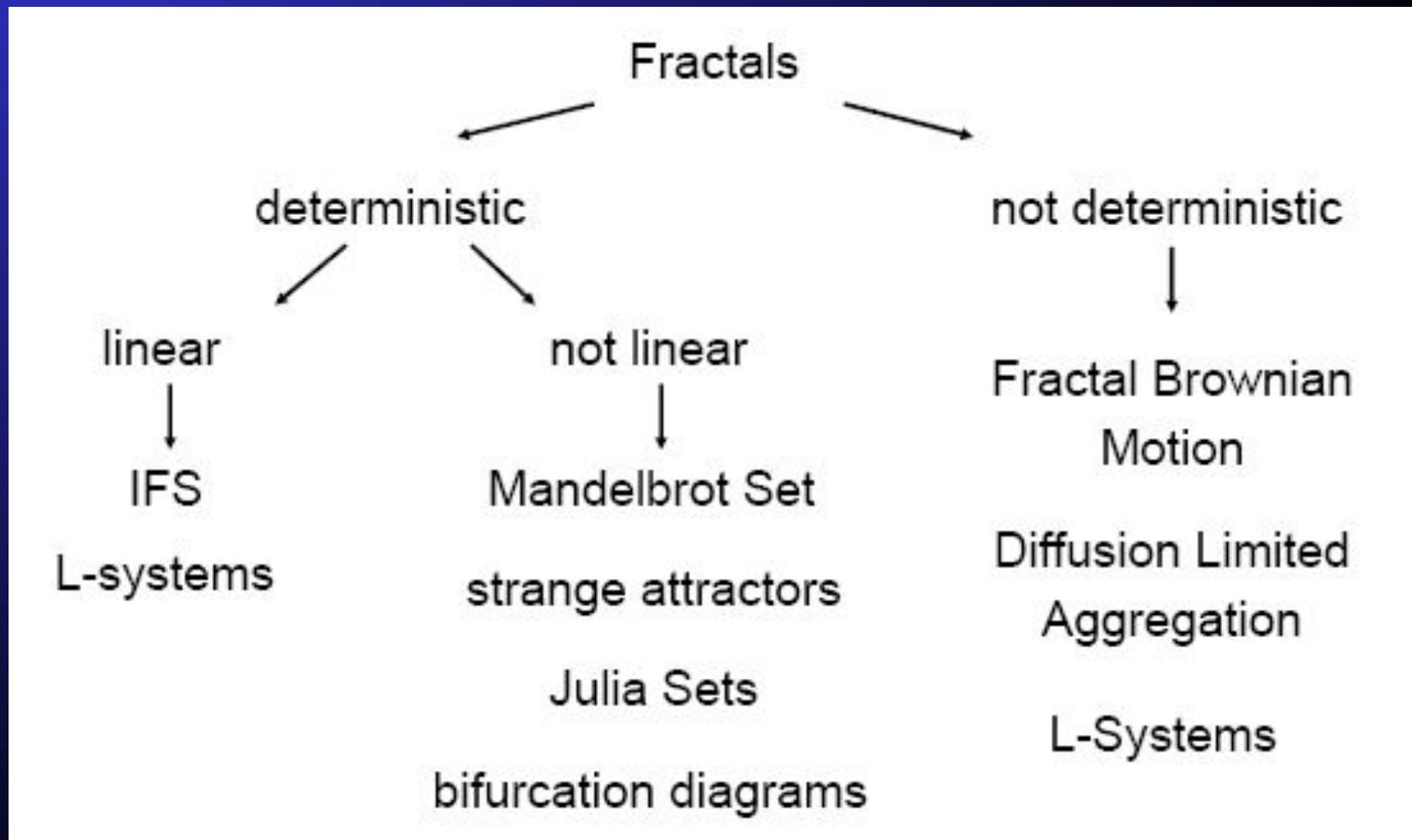
$$D = \ln(N) / \ln(1/S)$$

So Koch curve has dimension

$$\ln(4) / \ln(1/(1/3)) =$$

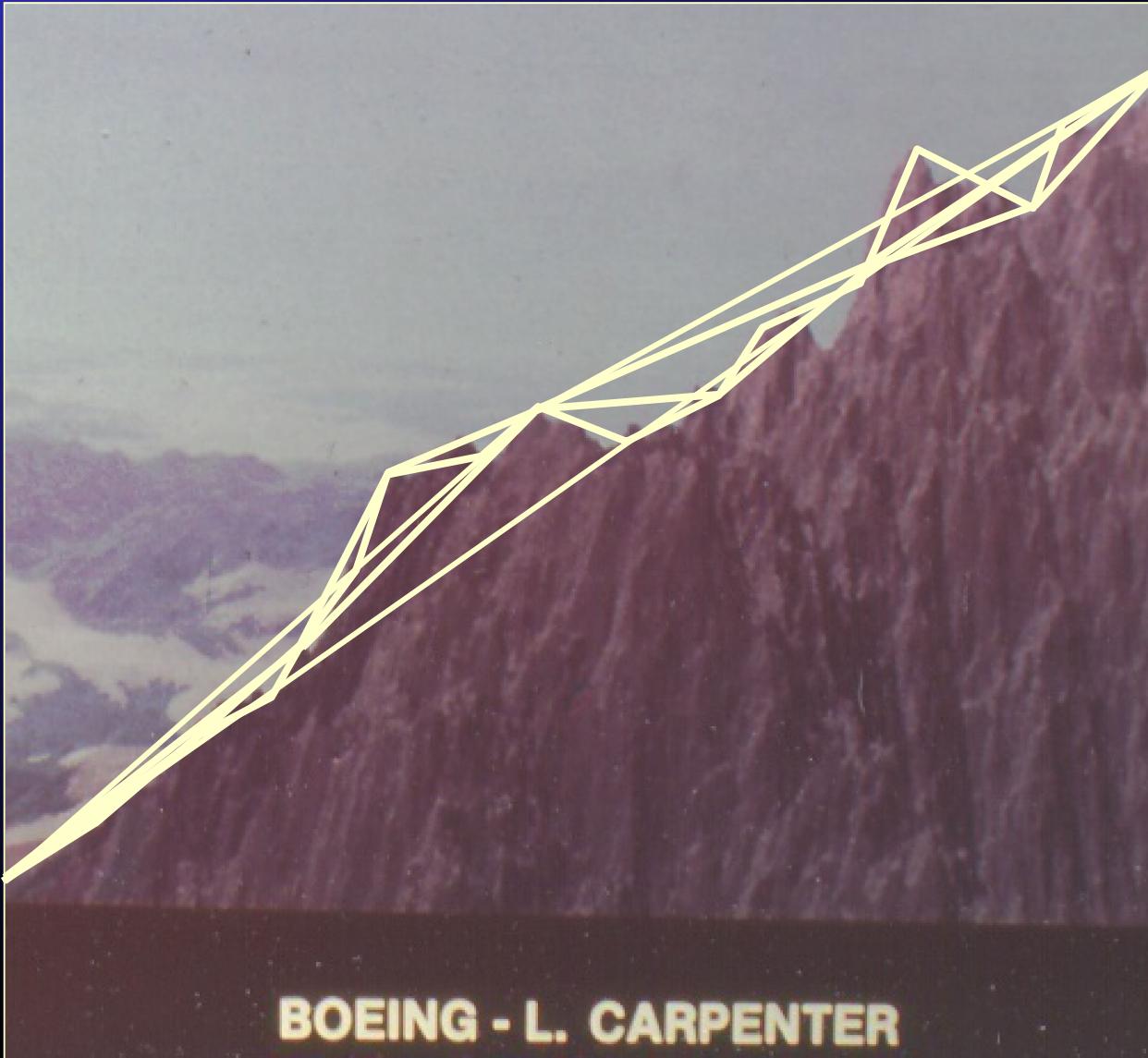
$$\ln(4) / \ln(3) = 1.2619\dots$$

A Whole Family of Fractals



For fun see: http://en.wikipedia.org/wiki/List_of_fractals_by_Hausdorff_dimension

Fractal Ridge: Repeatedly Divide Edge at Midpoint and Displace



BOEING - L. CARPENTER

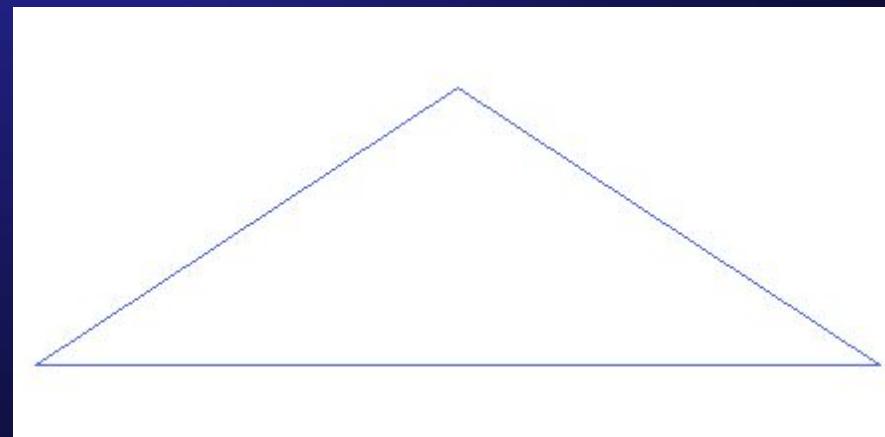
Fractal Mountain from Initial Polygon Pyramid



BOEING - L. CARPENTER

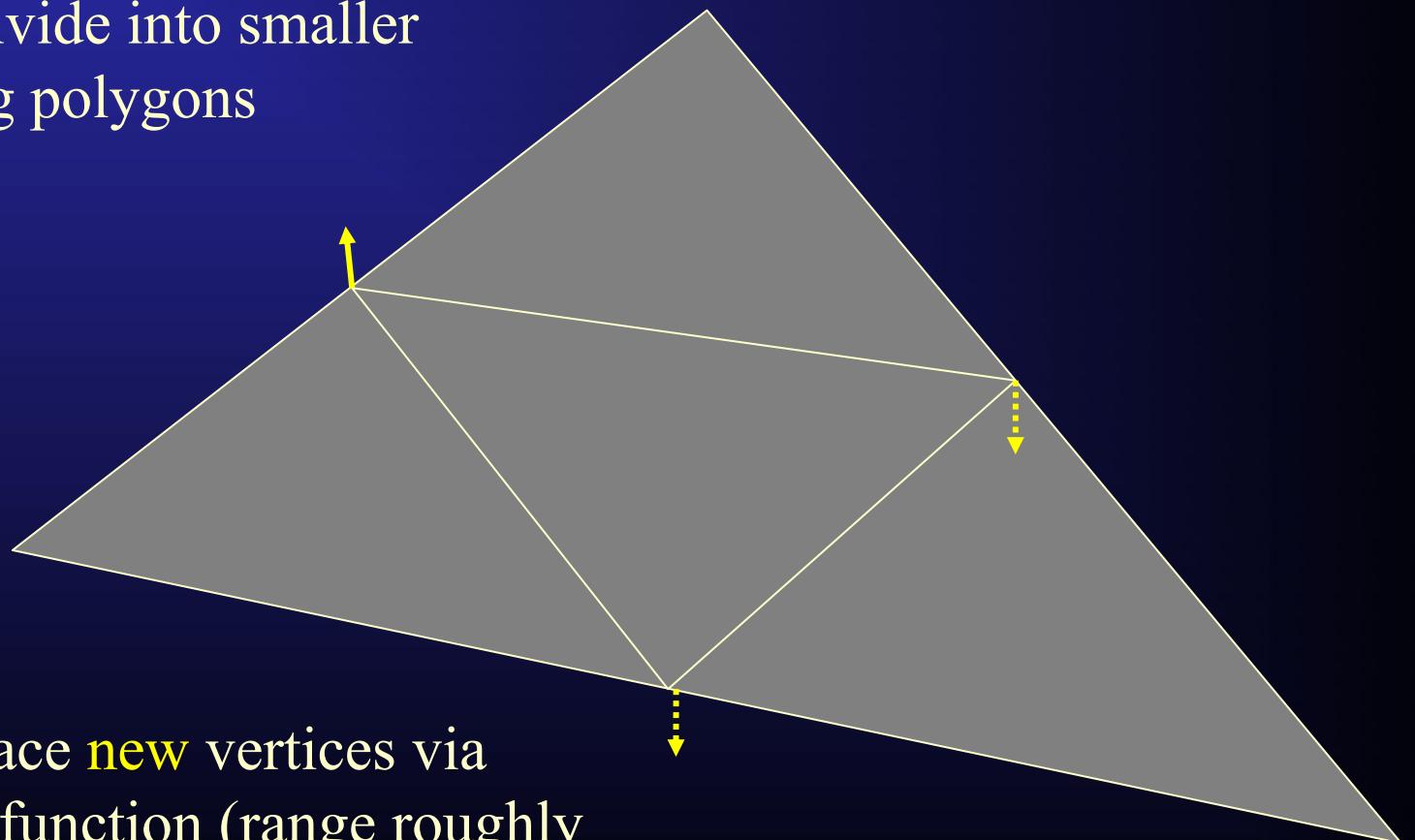
See Fractal Terrains on: <http://ibiblio.org/e-notes/3Dapp/Mount.htm>

Fractal Landscape



Fractal Surface Generation – Based on Height Field Perturbations

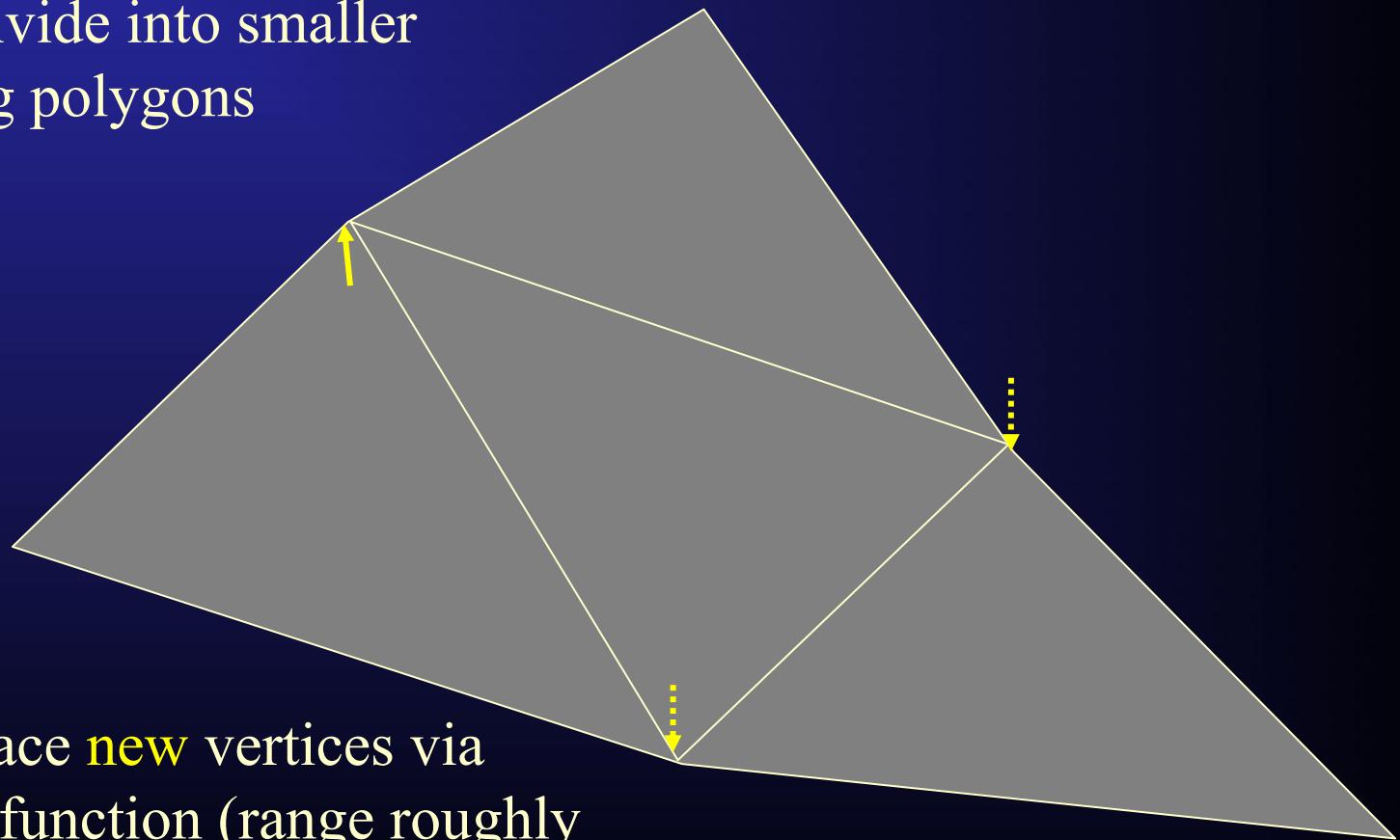
1. Subdivide into smaller covering polygons



2. Displace **new** vertices via random function (range roughly half of previous range depending on fractal dimension)

Fractal Surface Generation – Based on Height Field Perturbations

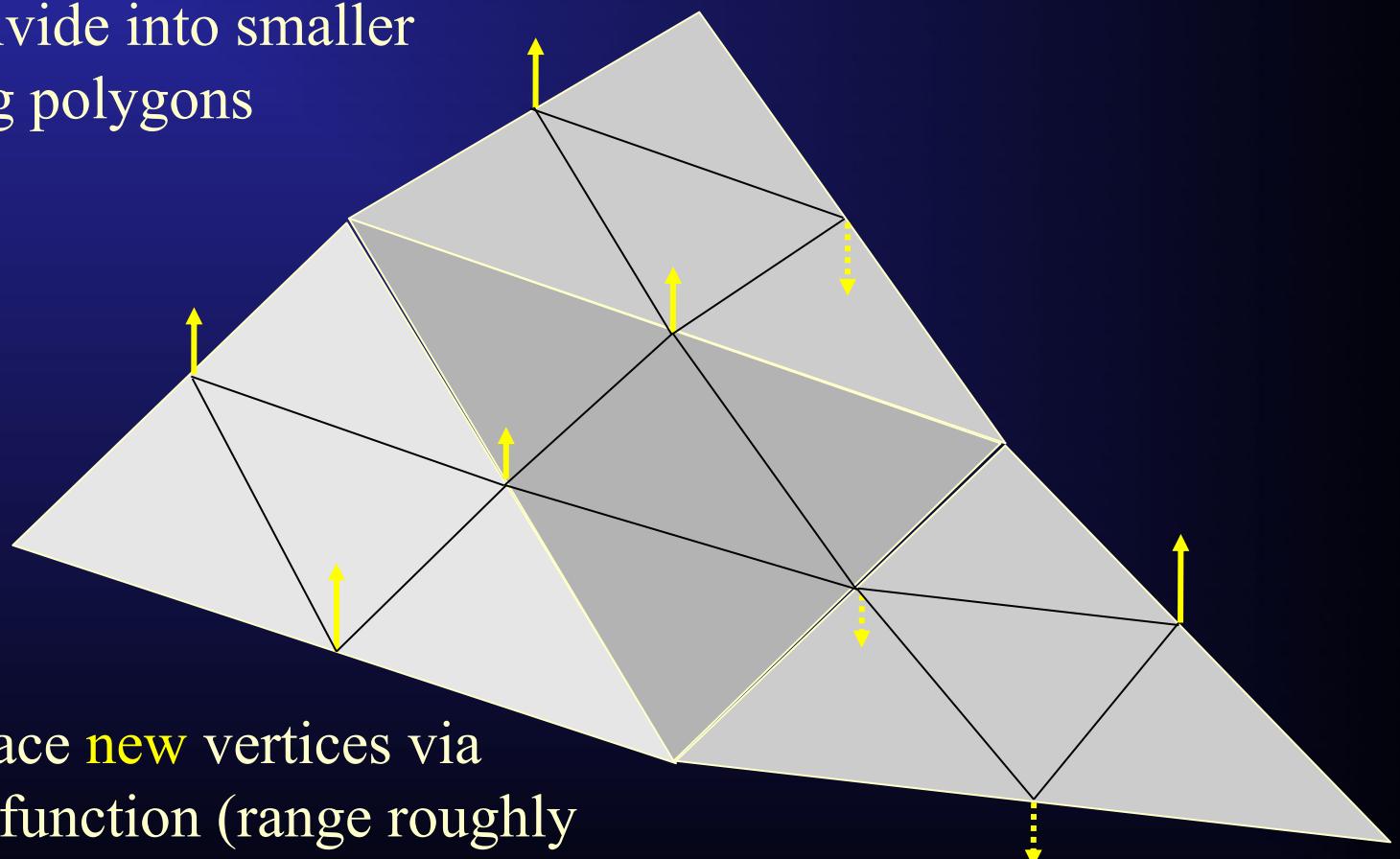
1. Subdivide into smaller covering polygons



2. Displace **new** vertices via random function (range roughly half of previous range depending on fractal dimension)

Fractal Surface Generation – Based on Height Field Perturbations

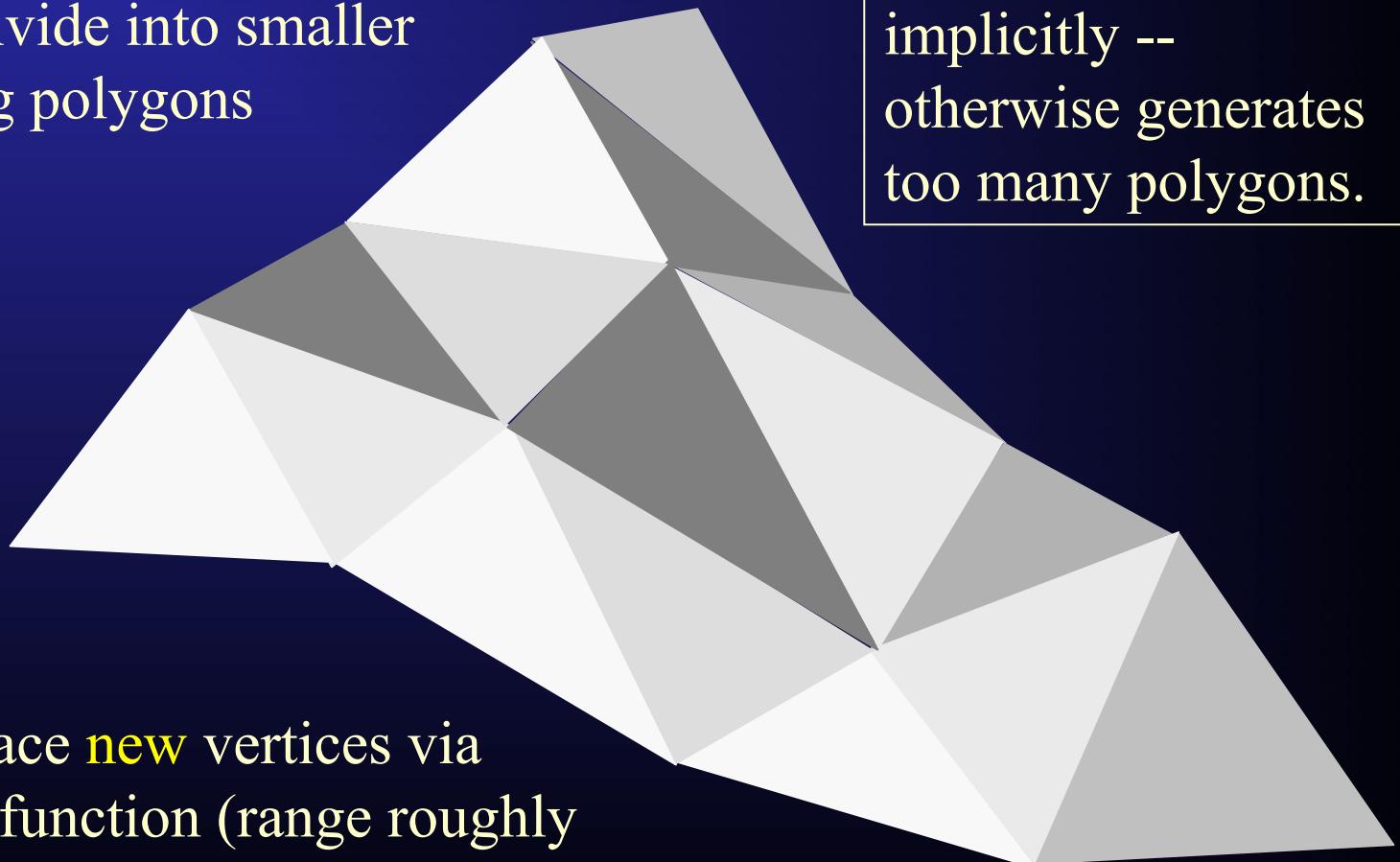
1. Subdivide into smaller covering polygons



2. Displace **new** vertices via random function (range roughly half of previous range depending on fractal dimension)

Fractal Surface Generation – Based on Height Field Perturbations

1. Subdivide into smaller covering polygons



In practice, do this implicitly -- otherwise generates too many polygons.

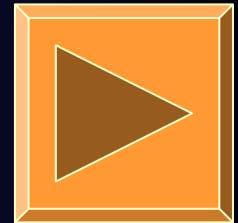
2. Displace **new** vertices via random function (range roughly half of previous range depending on fractal dimension)

Mountains, Valleys, and Lakes:
Water = level off height field and color it blue



RICHARD F. VOSS IBM © 1986

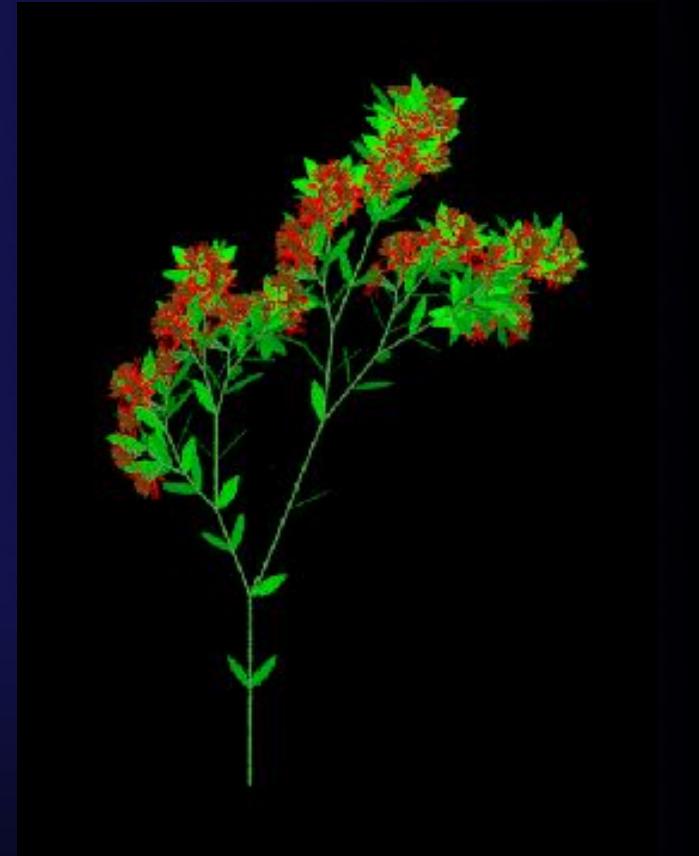
Lunar Landscape with Craters and Pseudo-Earth



© 1982 VOSS, R.—I.B.M.

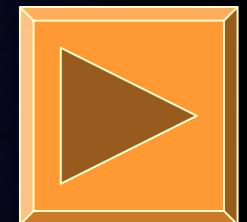
L-Systems, Iterated Functions, and Spatial Grammars

- L = Lindenmayer
- Motivated by plant growth patterns
- Use deterministic or rule-based generative process.
- Describe placement and relationships of object primitives via an explicit (usually context sensitive) grammar.



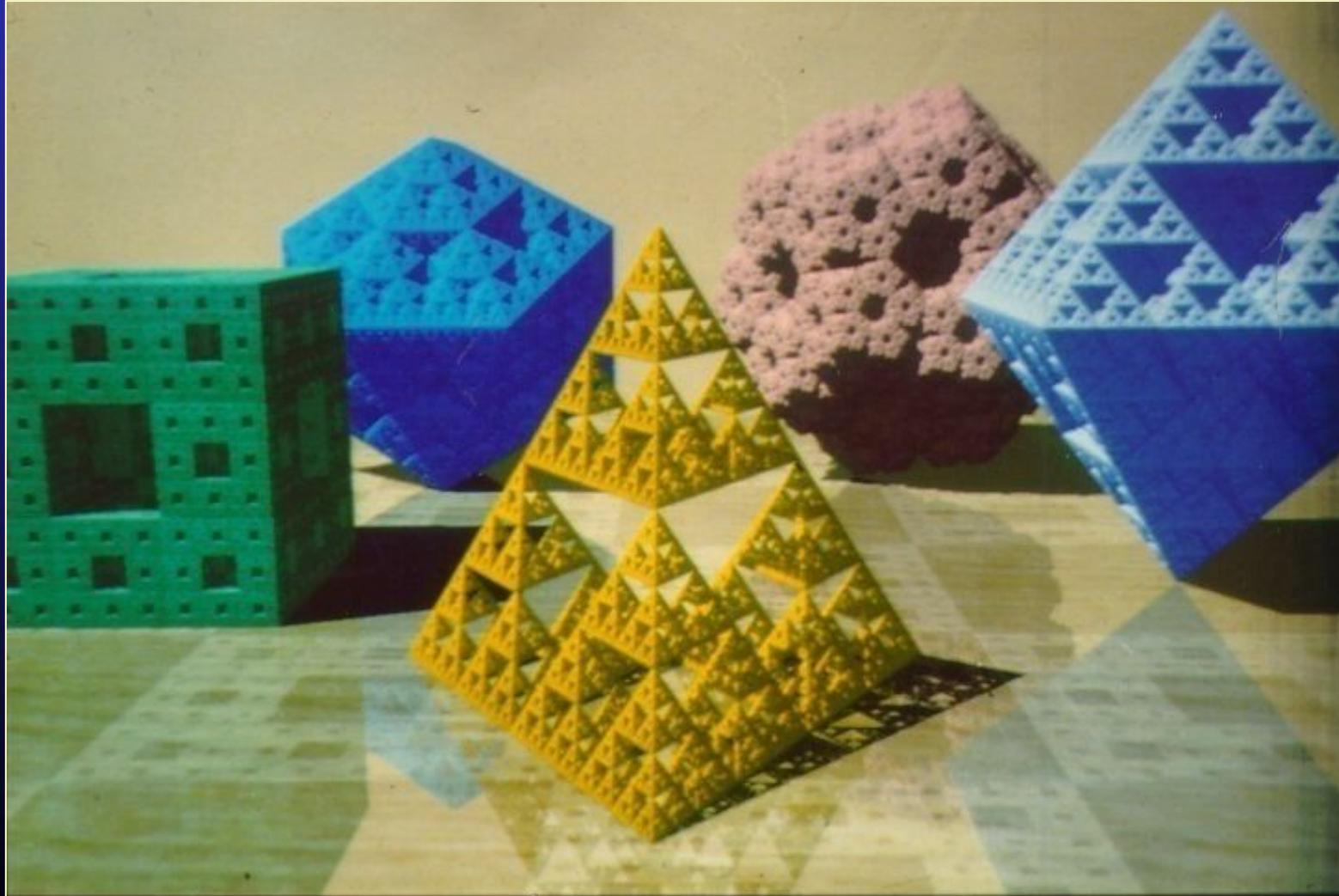
<http://www.csc.villanova.edu/~soong/vbl.html>

Iterated Function System: e.g., Fern, leaf, tree...



HODGES, BARNSLEY, NAYLOR — GEORGIA INST. OF TECH

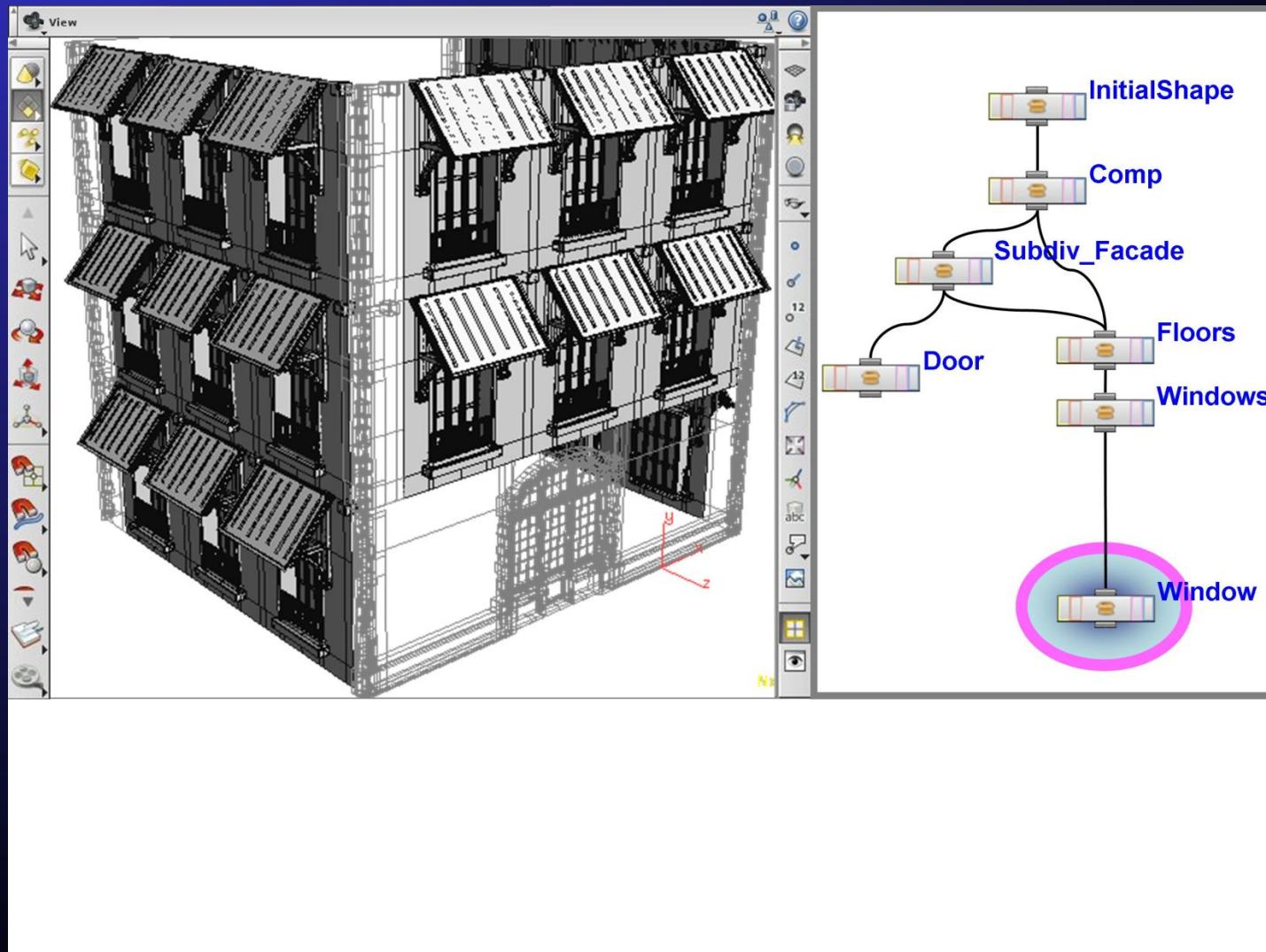
3D Sierpinski Gaskets



©1991 JOHN C HART, ELEC. VIS. LAB UIC

320

Spatial Grammar: Procedural Buildings



<http://ima.udg.edu/~dagush/papers/icons/geometryFlow.jpg>

And even Procedural Cities



http://lesterbanks.com/lxb_metal/wp-content/uploads/2010/02/rich_sun_productions_DemoCity.jpg

Curved Surfaces

- Parametric functions of two variables (bivariate).
- (s, t) parameters: $P(s, t) = (x(s, t), y(s, t), z(s, t))$
- True mathematical curvature.
- Simple surface tangents and normals.
- Adjacent patches may be constrained for continuity.
- Shape derived from control points and/or tangent vectors.
- Approximating and interpolating types.

Curved Surface Formulations

- Bezier ←
- Hermite
- B-spline ←
- Beta-spline
- Rational polynomial
- Non-Uniform Rational B-Splines (NURBS)
- Composite splines
- Subdivision curves and surfaces ←
- ...

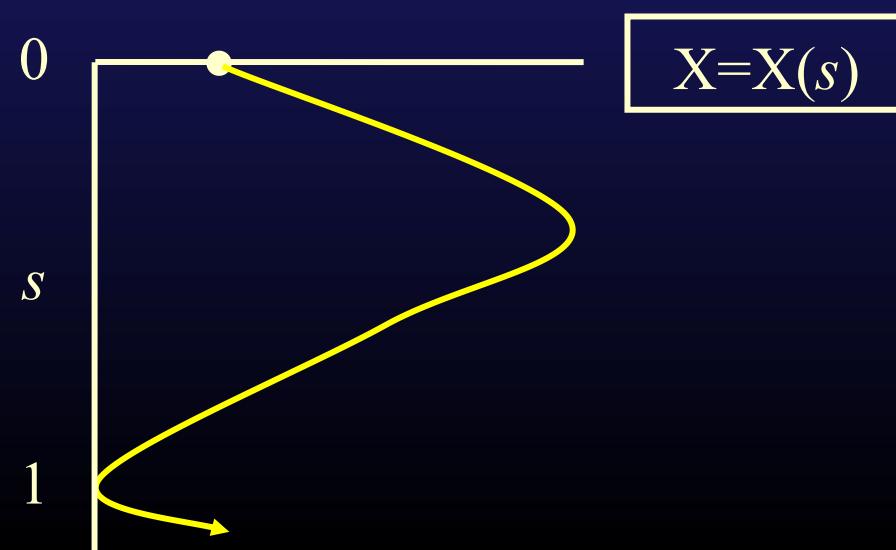
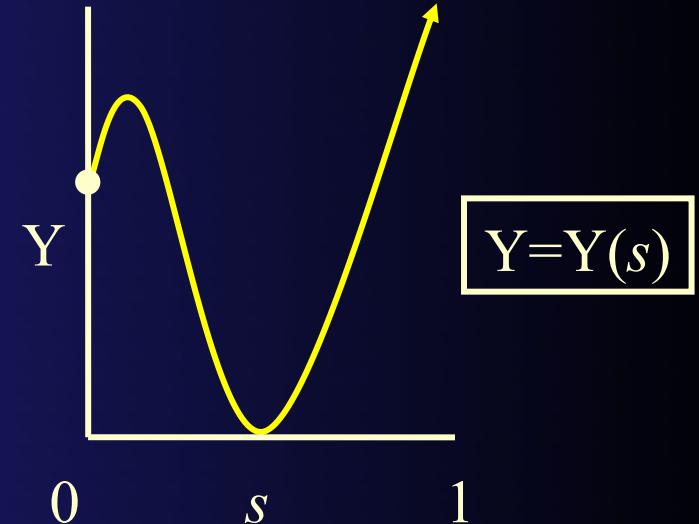
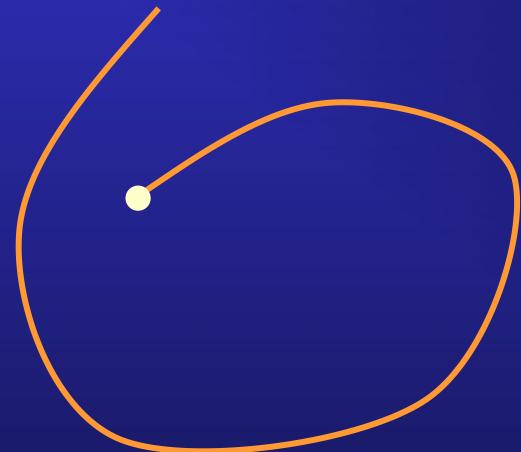
2D Curves and 3D Curved Surfaces

- Generalize interpolation over line segments.
(Recall $\mathbf{V} = (x', y') = \mathbf{V}_1 (1 - t) + \mathbf{V}_2 t$ for $t \in [0,1]$.)
- How to define curves/surfaces?
- How to manipulate them?
- How to draw them?
- What we intuitively know about curves and surfaces makes them appear to be difficult to represent with discrete data structures.
- But in fact, we need very little more than what we've already seen: vectors, polygons, and matrices.
- Points → polygon vertices; points → curve control points

2D Curves and 3D Curved Surfaces

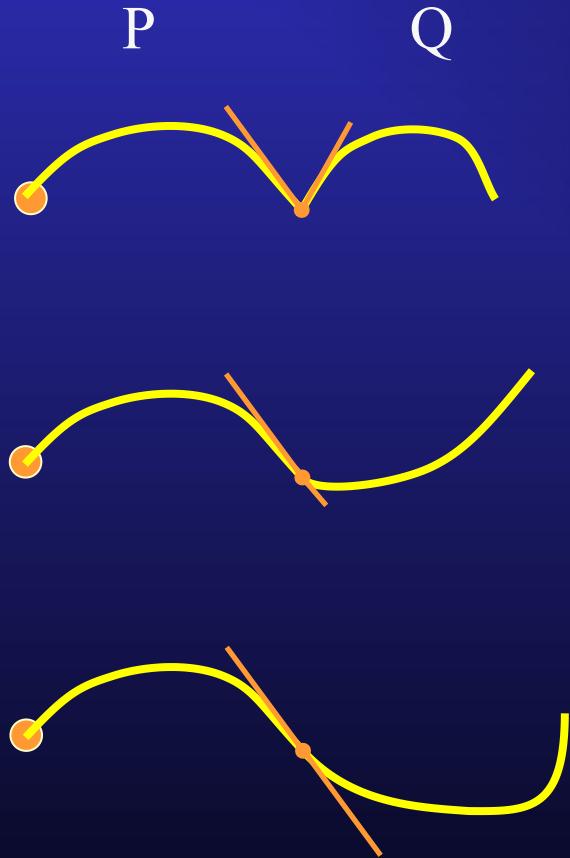
- Could define curves and surfaces implicitly with (arbitrary) functions, e.g., $F(x,y,z) = 0$ (we'll do that soon). Instead, use:
- Parametric functions of one (curve) or two (surface) variables:
 - Curves: Univariate: $P(s) = (x(s), y(s))$
 - Surfaces: Bivariate: $P(s, t) = (x(s, t), y(s, t), z(s, t))$
- For a 2D curve, s may be called the *arc length*, as it describes the position of a point along the curve.
- These have true mathematical curvature so we can use and easily compute surface tangents and normals.
- Note that a height field is just a special case of a 3D bivariate surface where $P(s, t) = (s, t, z(s, t))$.

Parametric Curves



$P=(X(s), Y(s))$
 $0 \leq s \leq 1$
 s is the arclength parameter

C^n Continuity



C^0 : 0th-order: Geometric

- (just “touch”)
- $P(X(1), Y(1)) = Q(X(0), Y(0))$

C^1 : 1st-order: Tangent

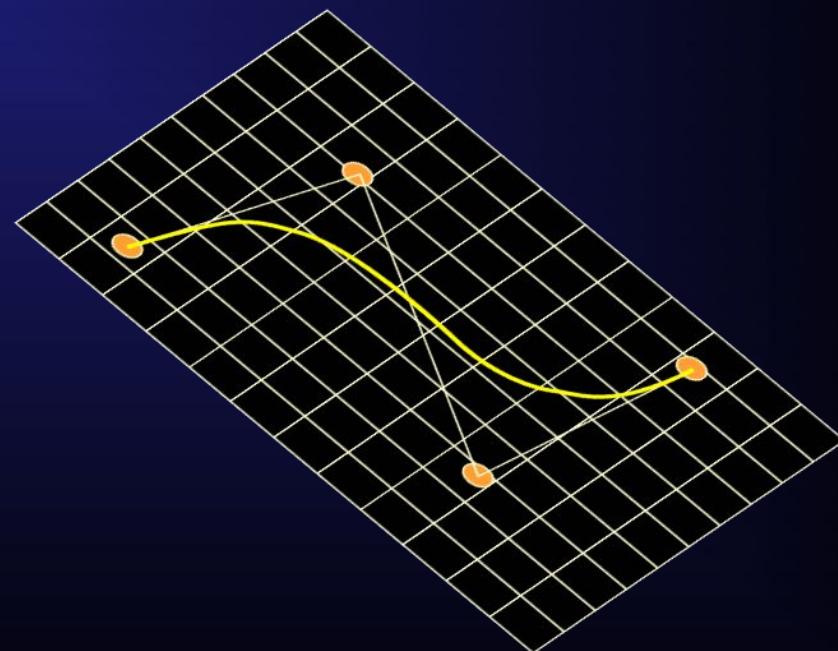
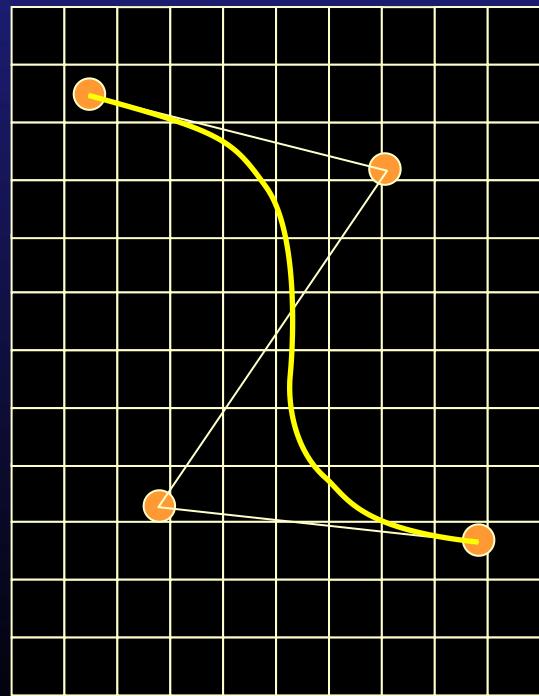
- (smooth direction)
- $P(X(1), Y(1)) = Q(X(0), Y(0))$ AND
- $P'(X(1), Y(1)) = Q'(X(0), Y(0))$

C^2 : 2nd-order: Curvature

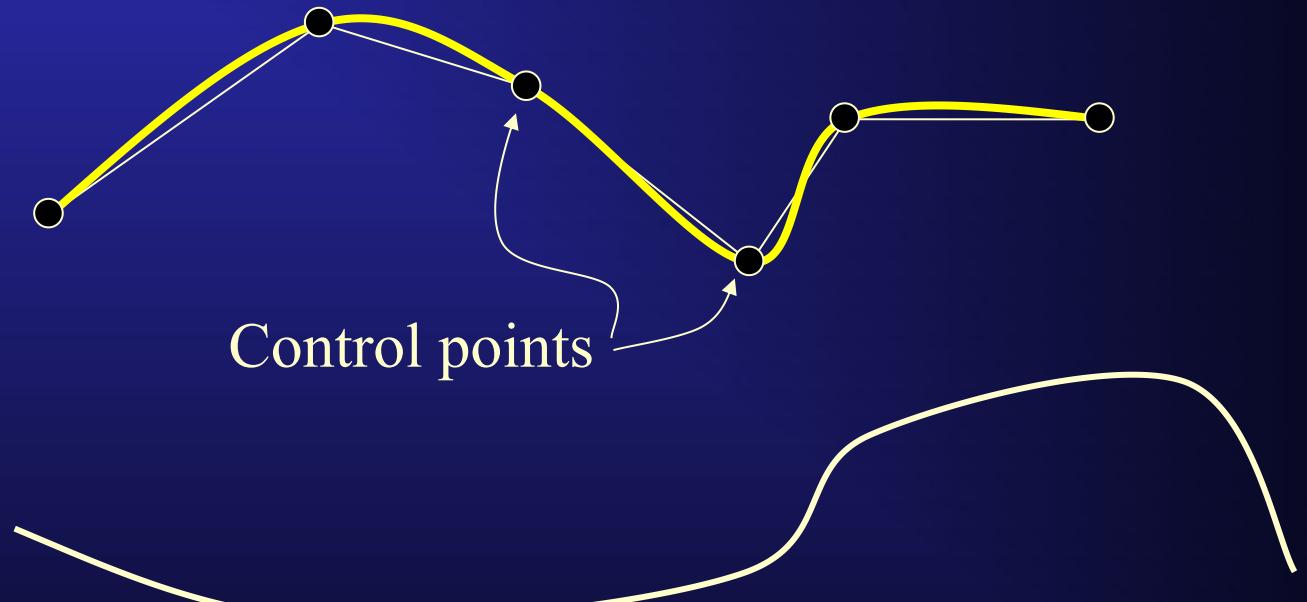
- (smooth “acceleration”)
- $P(X(1), Y(1)) = Q(X(0), Y(0))$ AND
- $P'(X(1), Y(1)) = Q'(X(0), Y(0))$ AND
- $P''(X(1), Y(1)) = Q''(X(0), Y(0))$

Affine Invariance

- Curves are represented by finite sets of *control points*.
- Transforming the control points gives the same result as transforming the curve.
- (Basis functions associated with curve type should always sum to 1.)

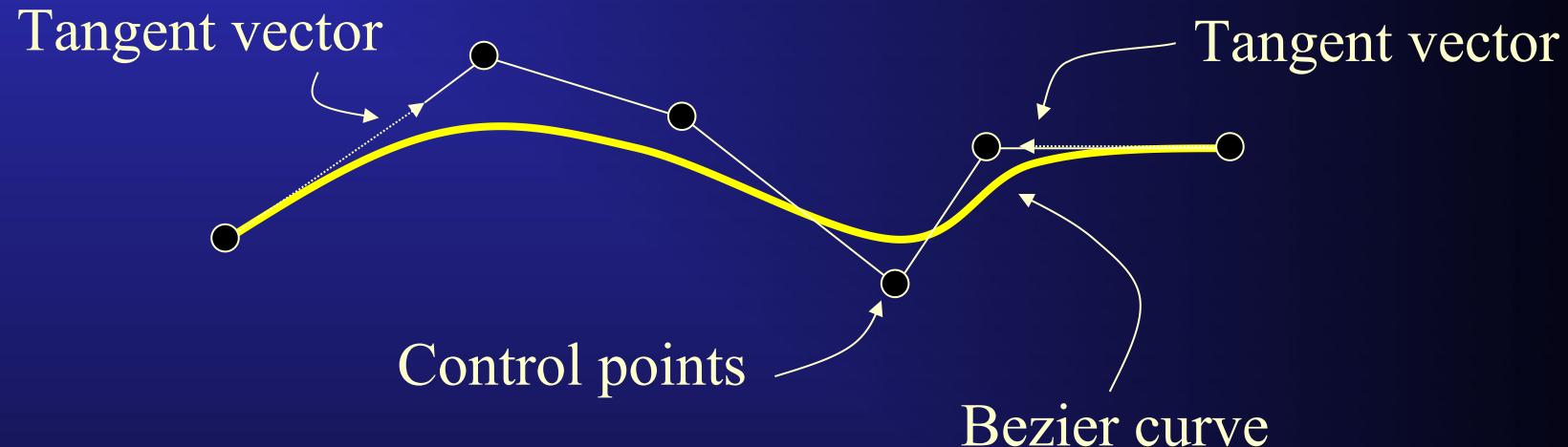


Basic Univariate Curve: B-spline



- *Interpolating* curve since it must pass through all the control points.
- User controls shape by control point position and two tangents at each control point.
- (Note affine invariance!)

Basic Univariate Curve: Bezier



- *Approximating* curve since it only usually passes through the first and last control points.
- User controls shape by positioning the control points.
- (Again, affine invariant.)



Bezier demo

<http://www2.mat.dtu.dk/people/J.Gravesen/cagd/decast.html>

The Math Behind Bezier Curves: Degree d Polynomials

- Use $d+1$ points as control points, called the control polygon.
- Need 2 of these to be the endpoints of the curve.
- 2 control endpoint tangents: $\mathbf{p}_1 \mathbf{p}_0$ and $\mathbf{p}_{d-1} \mathbf{p}_d$
- The other $d-1$ points \mathbf{p}_i (if there are any) geometrically control the shape of the rest of the curve.
- Write the curve as:

$$\mathbf{x}(t) = \sum_{i=0}^d p_i B_i^d(t) \quad \text{where} \quad B_i^d(t) = \binom{d}{i} t^i (1-t)^{d-i}$$

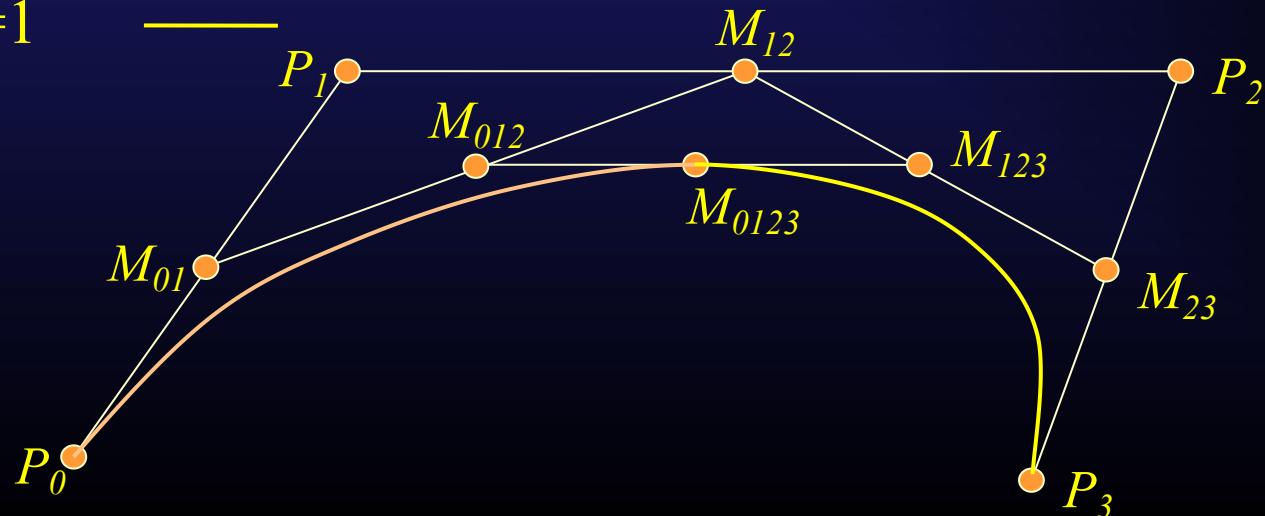
- Using the binomial coefficient:
$$\binom{d}{i} = \frac{d!}{(d-i)!i!}$$
- The polynomials $B_i^d(t)$ are Bernstein Basis functions – demo: 
- For $d=3$, the Bezier curve is a cubic (degree 3) polynomial.
- Pre-compute coefficients for fixed d , e.g., $\{1, 3, 3, 1\}$ for $d=3$.

Bezier Curve Evaluation: The *de Casteljau* Algorithm

- A Bezier curve lies entirely within the convex hull of its control polygon.
- If the control points are “nearly” collinear, then the curve is “nearly” a straight line.
- A cubic Bezier curve can be broken into two shorter cubic Bezier curves that exactly cover the original curve.
- These observations lead to *de Casteljau’s algorithm*:
 - Break the curve into sub-curves,
 - Stop when the control points of each sub-curve are nearly collinear,
 - Draw the polygon formed by the control points.
- Evaluates the curve without computing the cubics.

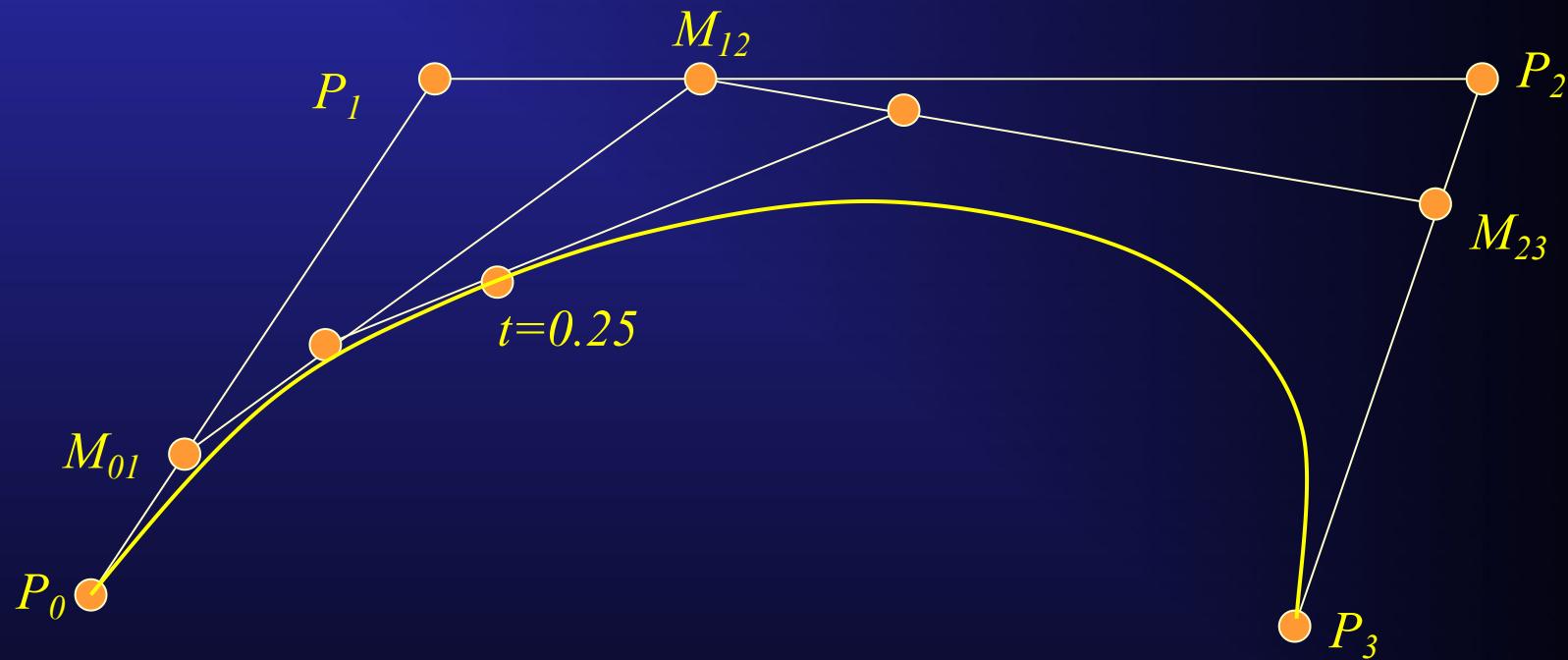
For Example: Break at midpoint $t = 0.5$:

- Step 1: Find the midpoints M_{01}, M_{12}, M_{23} of the lines joining the original control vertices.
- Step 2: Find the midpoints M_{012}, M_{123} of the lines joining M_{01}, M_{12} and M_{12}, M_{23} .
- Step 3: Find the midpoint M_{0123} of the line joining M_{012}, M_{123} .
- The curve with control points P_0, M_{01}, M_{012} and M_{0123} exactly follows the original curve from the point with $t=0$ to the point with $t=0.5$
- The curve with control points $M_{0123}, M_{123}, M_{23}$ and P_3 exactly follows the original curve from the point with $t=0.5$ to the point with $t=1$



And, Moreover...

This works for any value of t between 0 and 1, e.g. $t = 0.25$:



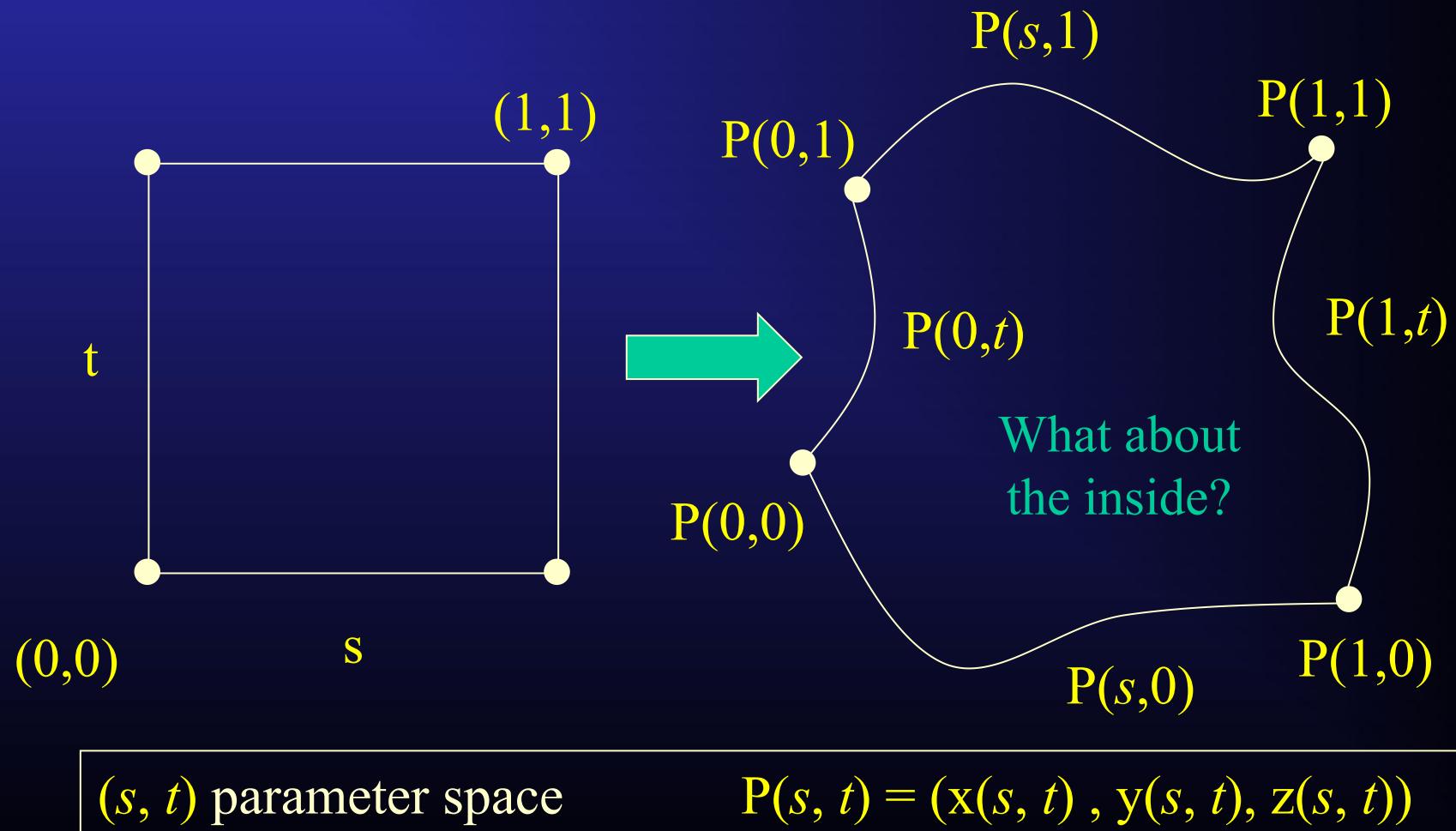
This leads naturally to a drawing method for Bezier curves:



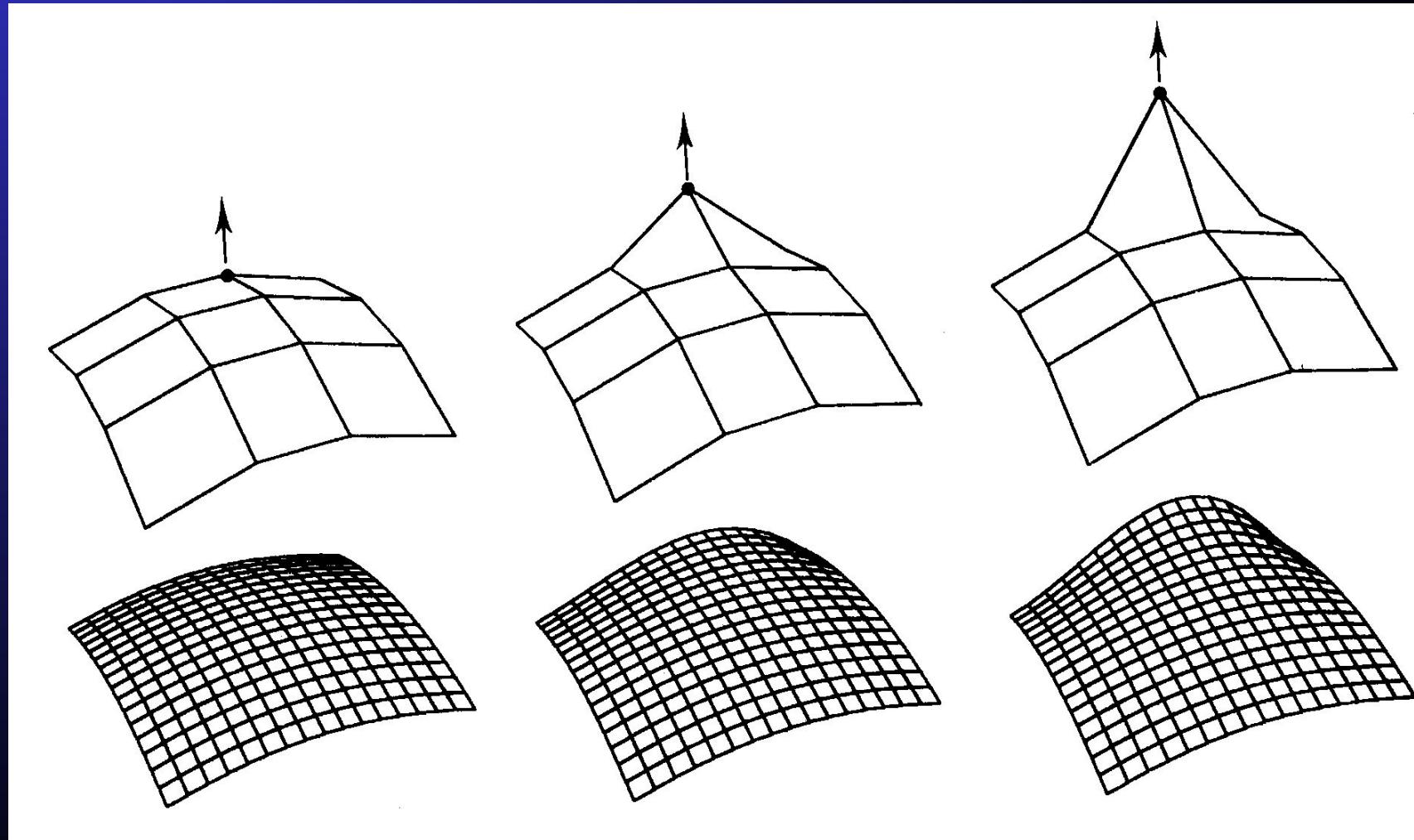
Bezier de Casteljau demo

<http://www2.mat.dtu.dk/people/J.Gravesen/cagd/decast.html>

Basic Bivariate Mapping



Simple Bezier Surface Wireframe Example

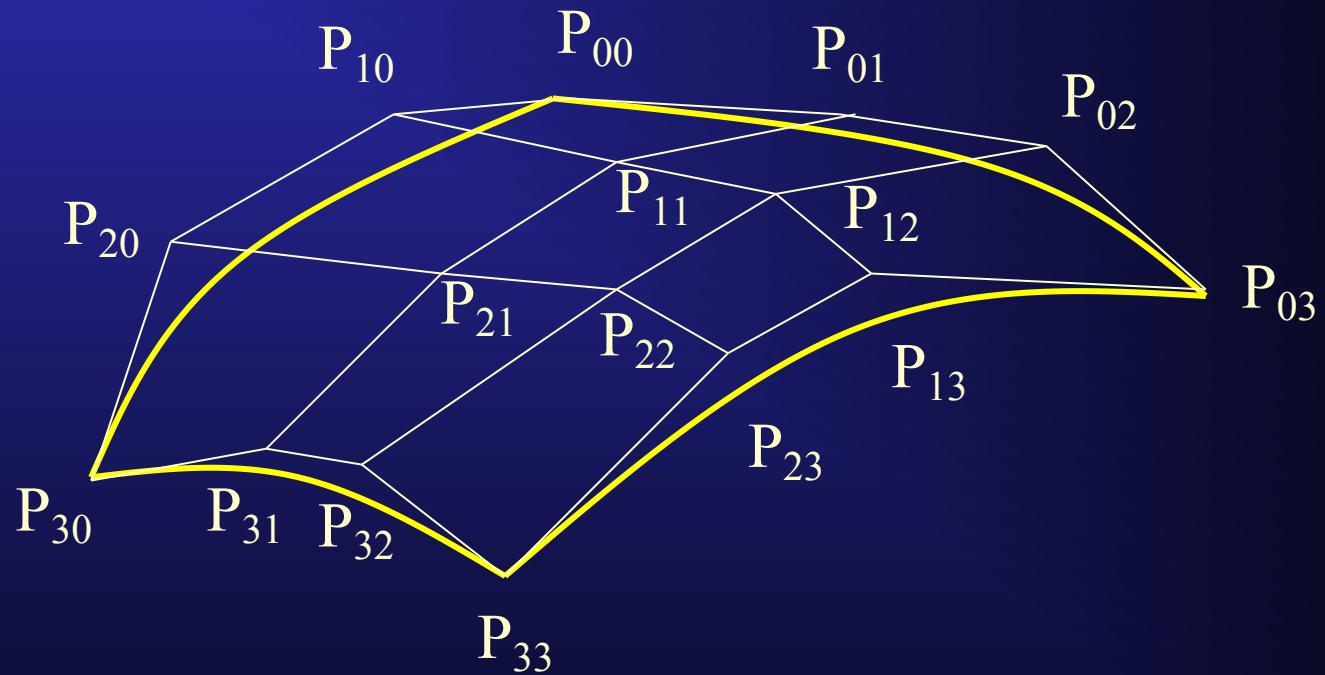


Bezier Surface demo

<http://www.nbb.cornell.edu/neurobio/land/OldStudentProjects/cs490-96to97/anson/BezierPatchApplet/index.html>



Bezier Surface Representation as Matrix



$$P(s, t) = S[M][G][M]^T T^T$$

Bezier Matrix (Tensor Product) Form

where

$$S = \begin{pmatrix} s^3 & s^2 & s & 1 \end{pmatrix}$$

$$T = \begin{pmatrix} t^3 & t^2 & t & 1 \end{pmatrix}$$

$$[M] = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} = [M]^T$$

$$[G] = \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix}$$

These are the variables.
(Note they form cubic polynomials -- this is the form for cubic Bezier surfaces; other degrees are possible)

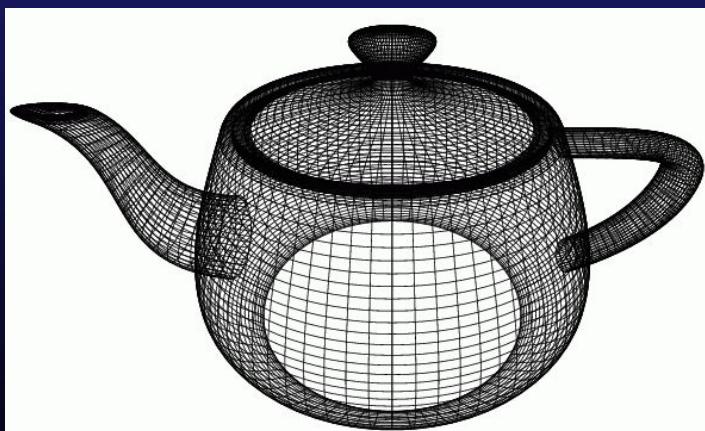
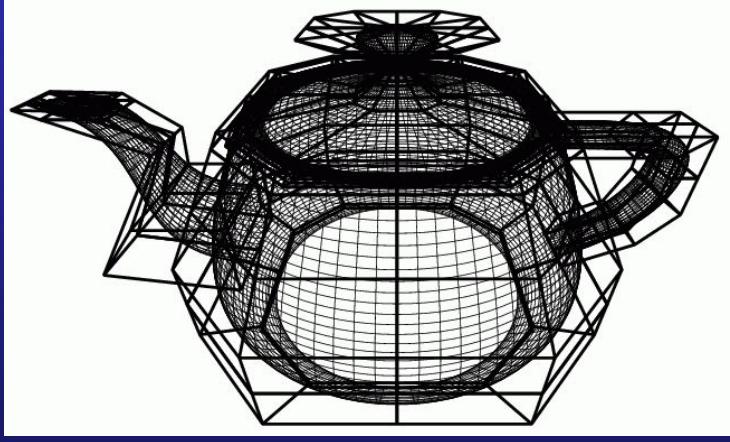
Notice how the control points appear explicitly in the G (geometry) matrix. (This makes G a tensor.)

Drawing the Wire Mesh

stepsize = 1.0/nsteps; {e.g., 0.05 for nsteps = 20 each
in s and t}

```
for (s = stepsize; s=1.0; s=s+stepsize){  
    for (t = stepsize; t=1.0; t=t+stepsize){  
        drawline (P(s-stepsize, t-stepsize), P(s,t) )} } }
```

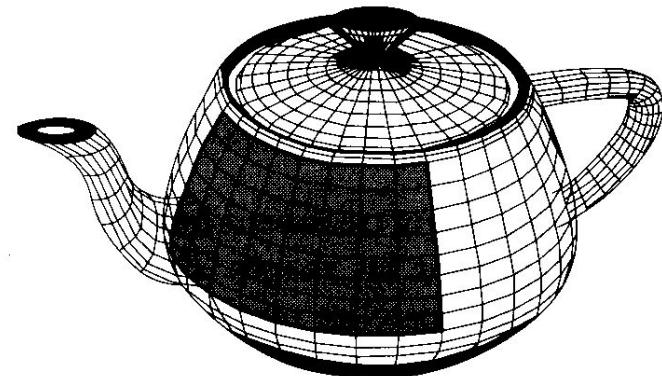
The Famous Graphics Teapot is Just 9 4×4 Bezier Patches



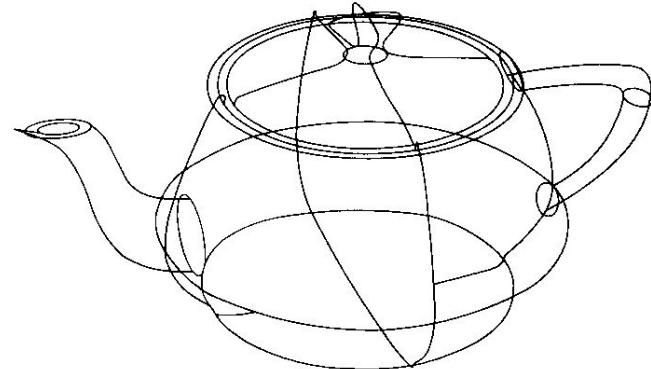
```
Rim:  
{ 102, 103, 104, 105, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 }  
Body:  
{ 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27 }  
{ 24, 25, 26, 27, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40 }  
Lid:  
{ 96, 96, 96, 96, 97, 98, 99, 100, 101, 101, 101, 101, 101, 101, 101, 101 }  
{ 0, 1, 2, 3, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117 }  
Handle:  
{ 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56 }  
{ 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67 }  
Spout:  
{ 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83 }  
{ 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95 }  
  
Vertices:  
  
{ 0.2000, 0.0000, 2.70000 }, { 0.2000, -0.1120, 2.70000 }, { 0.1120, -0.2000, 2.70000 },  
{ 0.0000, -0.2000, 2.70000 }, { 1.3375, 0.0000, 2.53125 }, { 1.3375, -0.7490, 2.53125 },  
{ 0.7490, -1.3375, 2.53125 }, { 0.0000, -1.3375, 2.53125 }, { 1.4375, 0.0000, 2.53125 },  
{ 1.4375, -0.8050, 2.53125 }, { 0.8050, -1.4375, 2.53125 }, { 0.0000, -1.4375, 2.53125 },  
{ 1.5000, 0.0000, 2.40000 }, { 1.5000, -0.8400, 2.40000 }, { 0.8400, -1.5000, 2.40000 },  
{ 0.0000, -1.5000, 2.40000 }, { 1.7500, 0.0000, 1.87500 }, { 1.7500, -0.9800, 1.87500 },  
{ 0.9800, -1.7500, 1.87500 }, { 0.0000, -1.7500, 1.87500 }, { 2.0000, 0.0000, 1.35000 },  
{ 2.0000, -1.1200, 1.35000 }, { 1.1200, -2.0000, 1.35000 }, { 0.0000, -2.0000, 1.35000 },  
{ 2.0000, 0.0000, 0.90000 }, { 2.0000, -1.1200, 0.90000 }, { 1.1200, -2.0000, 0.90000 },  
{ 0.0000, -2.0000, 0.90000 }, { -2.0000, 0.0000, 0.90000 }, { 2.0000, 0.0000, 0.45000 },  
{ 2.0000, -1.1200, 0.45000 }, { 1.1200, -2.0000, 0.45000 }, { 0.0000, -2.0000, 0.45000 },  
{ 1.5000, 0.0000, 0.22500 }, { 1.5000, -0.8400, 0.22500 }, { 0.8400, -1.5000, 0.22500 },  
{ 0.0000, -1.5000, 0.22500 }, { 1.5000, 0.0000, 0.15000 }, { 1.5000, -0.8400, 0.15000 },  
{ 0.8400, -1.5000, 0.15000 }, { 0.0000, -1.5000, 0.15000 }, { -1.6000, 0.0000, 2.02500 },  
{ -1.6000, -0.3000, 2.02500 }, { -1.5000, -0.3000, 2.25000 }, { -1.5000, 0.0000, 2.25000 },  
{ -2.3000, 0.0000, 2.02500 }, { -2.3000, -0.3000, 2.02500 }, { -2.5000, -0.3000, 2.25000 },  
{ -2.5000, 0.0000, 2.25000 }, { -2.7000, 0.0000, 2.02500 }, { -2.7000, -0.3000, 2.02500 },  
{ -3.0000, -0.3000, 2.25000 }, { -3.0000, 0.0000, 2.25000 }, { -2.7000, 0.0000, 1.80000 },  
{ -2.7000, -0.3000, 1.80000 }, { -3.0000, -0.3000, 1.80000 }, { -3.0000, 0.0000, 1.80000 },  
{ -2.7000, 0.0000, 1.57500 }, { -2.7000, -0.3000, 1.57500 }, { -3.0000, -0.3000, 1.35000 },  
{ -3.0000, 0.0000, 1.35000 }, { -2.5000, 0.0000, 1.12500 }, { -2.5000, -0.3000, 1.12500 },  
{ -2.6500, -0.3000, 0.93750 }, { -2.6500, 0.0000, 0.93750 }, { -2.0000, -0.3000, 0.90000 },  
{ -1.9000, -0.3000, 0.60000 }, { -1.9000, 0.0000, 0.60000 }, { 1.7000, 0.0000, 1.42500 },  
{ 1.7000, -0.6600, 1.42500 }, { 1.7000, -0.6600, 0.60000 }, { 1.7000, 0.0000, 0.60000 },  
{ 2.6000, 0.0000, 1.42500 }, { 2.6000, -0.6600, 1.42500 }, { 3.1000, 0.0000, 0.82500 },  
{ 3.1000, 0.0000, 0.82500 }, { 2.3000, 0.0000, 2.10000 }, { 2.3000, 0.0000, 2.10000 },  
{ 2.4000, -0.2500, 2.02500 }, { 2.4000, 0.0000, 2.02500 }, { 2.7000, 0.0000, 2.40000 },  
{ 2.7000, -0.2500, 2.40000 }, { 3.3000, -0.2500, 2.40000 }, { 3.3000, 0.0000, 2.40000 },  
{ 2.8000, 0.0000, 2.47500 }, { 2.8000, -0.2500, 2.47500 }, { 3.5250, -0.2500, 2.49375 },  
{ 3.5250, 0.0000, 2.49375 }, { 2.9000, 0.0000, 2.47500 }, { 2.9000, -0.1500, 2.47500 },  
{ 3.4500, -0.1500, 2.51250 }, { 3.4500, 0.0000, 2.51250 }, { 2.8000, -0.1500, 2.40000 },  
{ 2.8000, 0.0000, 2.40000 }, { 3.2000, -0.1500, 2.40000 }, { 3.2000, 0.0000, 2.40000 },  
{ 0.0000, 0.0000, 3.15000 }, { 0.8000, 0.0000, 3.15000 }, { 0.8000, -0.4500, 3.15000 },  
{ 0.4500, -0.8000, 3.15000 }, { 0.0000, -0.8000, 3.15000 }, { 0.0000, 0.0000, 2.85000 },  
{ 1.4000, 0.0000, 2.40000 }, { 1.4000, -0.7840, 2.40000 }, { 0.7840, -1.4000, 2.40000 },  
{ 0.0000, -1.4000, 2.40000 }, { 0.4000, 0.0000, 2.55000 }, { 0.4000, -0.2240, 2.55000 },  
{ 0.2240, -0.4000, 2.55000 }, { 0.0000, -0.4000, 2.55000 }, { 1.3000, 0.0000, 2.55000 },  
{ 1.3000, -0.7280, 2.55000 }, { 0.7280, -1.3000, 2.55000 }, { 0.0000, -1.3000, 2.55000 },  
{ 1.3000, 0.0000, 2.40000 }, { 1.3000, -0.7280, 2.40000 }, { 0.7280, -1.3000, 2.40000 },  
{ 0.0000, -1.3000, 2.40000 }, { 0.0000, 0.0000, 0.0000 }
```

For data sets see: <http://www.holmes3d.net/graphics/teapot/>

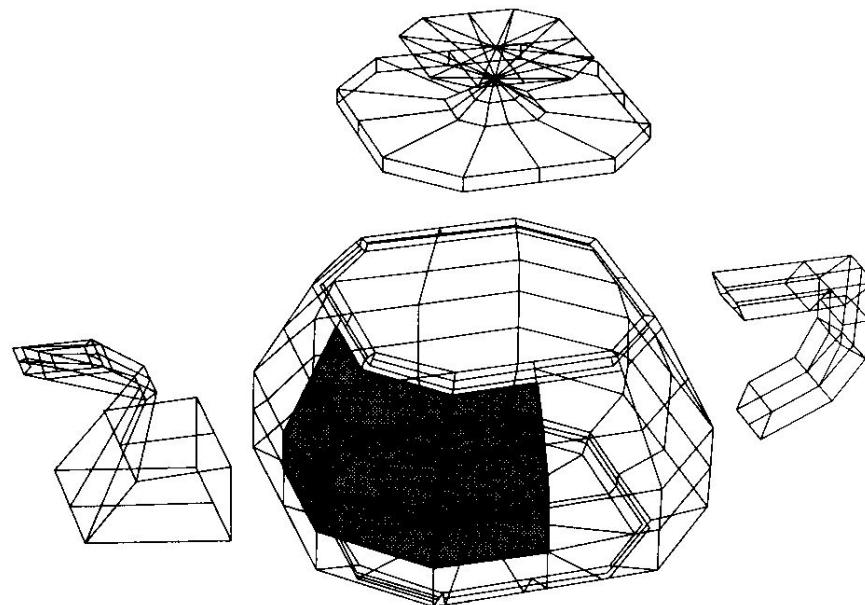
The Graphics Teapot (Martin Newell)



Lines of constant s and t



Bezier patch boundaries



Bezier control polygons

Bezier Patches for Designing a Goblet



Patch-patch seam

Note collinear
control points
across seam
guarantee at least
first order
continuity!

Parametric Shape Control: These are Beta Splines with Tension and Bias Parameters



© 1984 BRIAN A. BARSKY—BERKELEY COMPUTER
GRAPHICS LABORATORY, UNIVERSITY OF CALIFORNIA

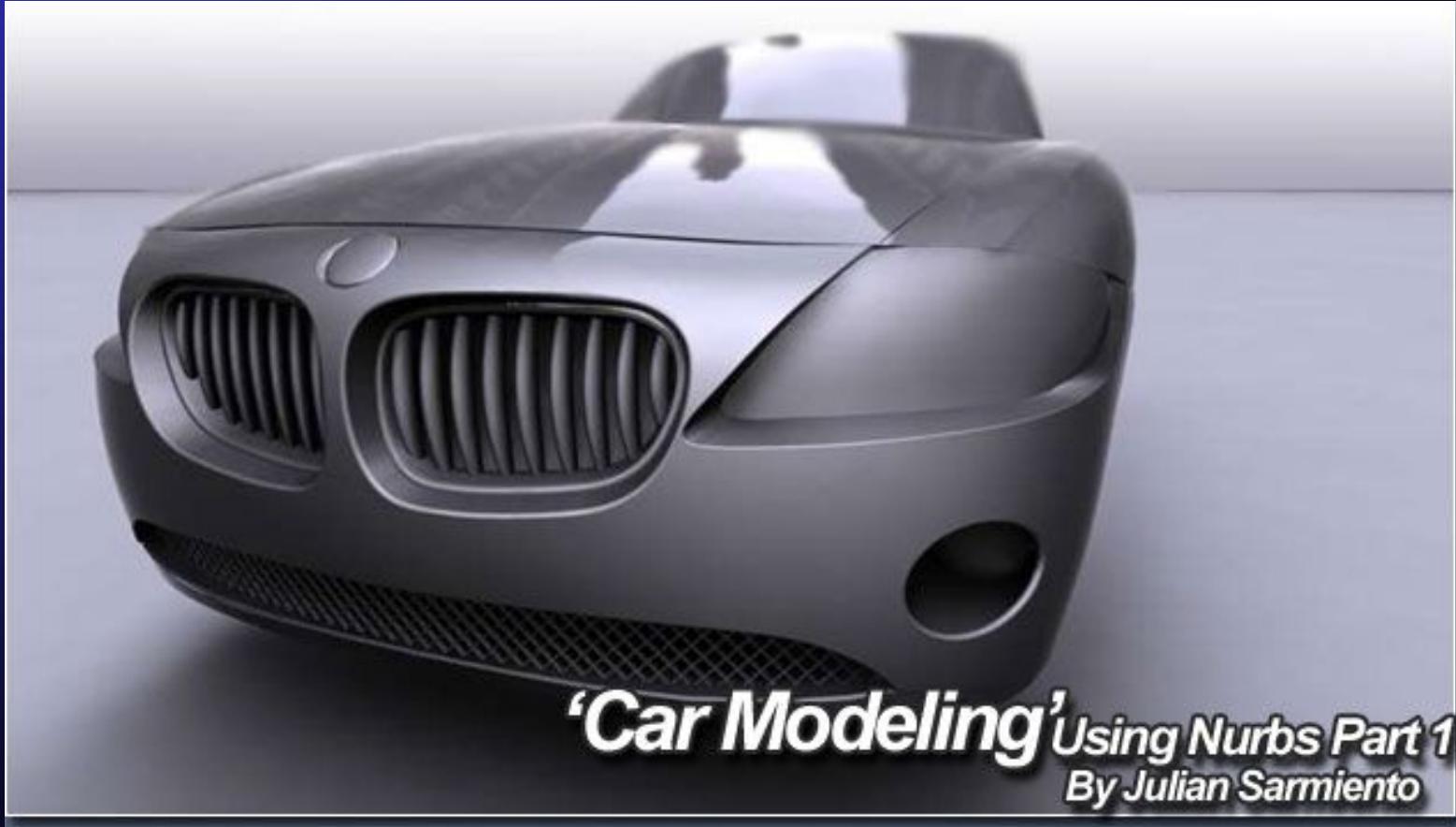
Smooth Curves are Common;
Another Software Industry Grows Up



c 1988

G. Mundell, A. Pearce, Alias Res.

Patch-to-Patch Continuity Important for Design



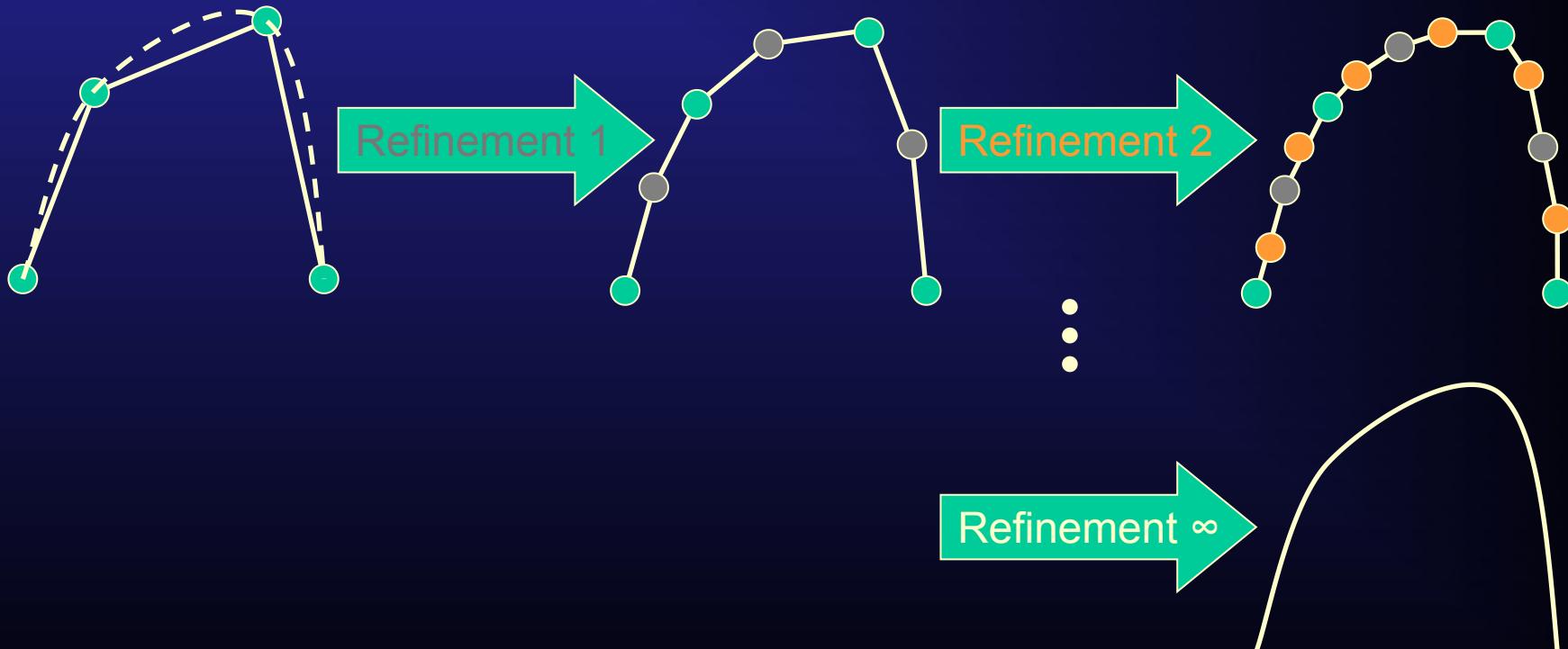
Subdivision Surfaces



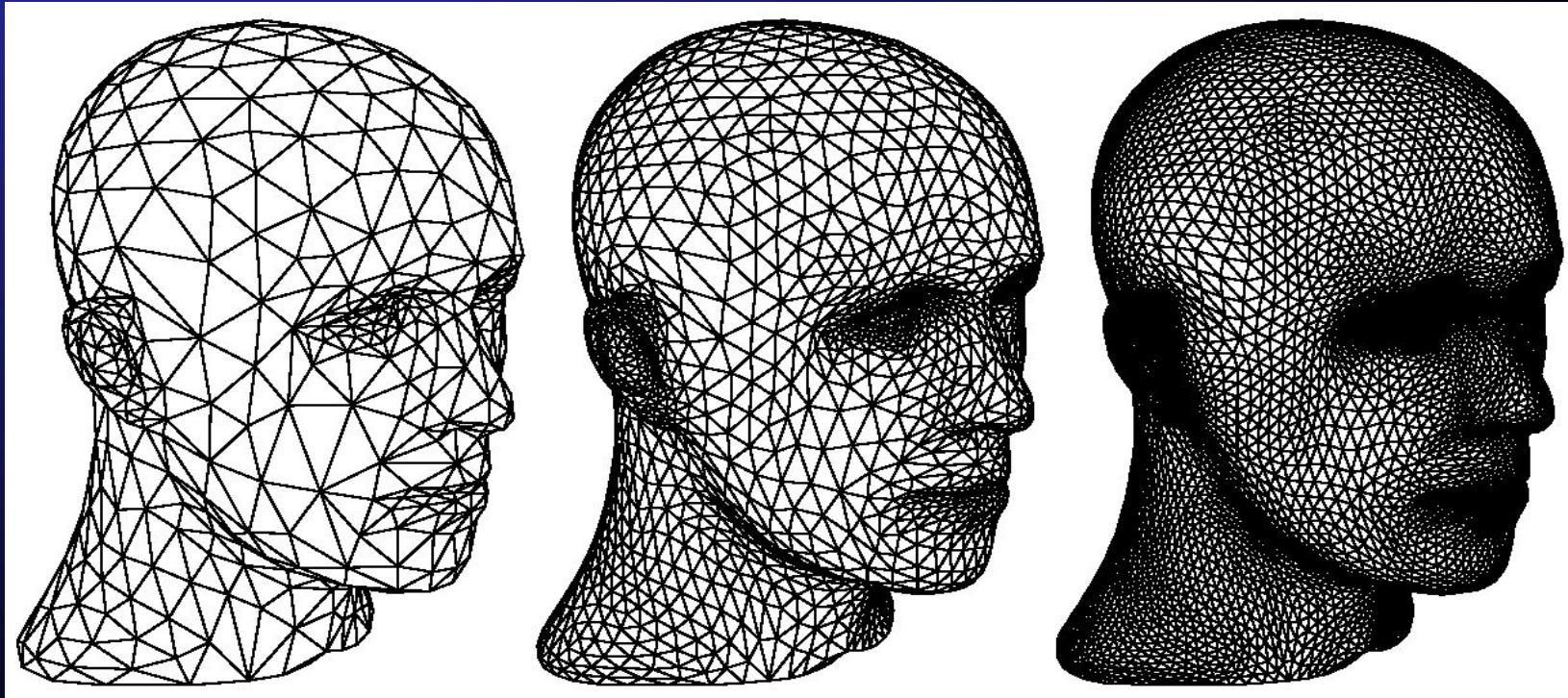
Geri's Game (1997) : Pixar Animation Studios

Subdivision Surfaces

- Approach limit curve or surface through an iterative refinement (subdivision) process that converges to a smooth spline.

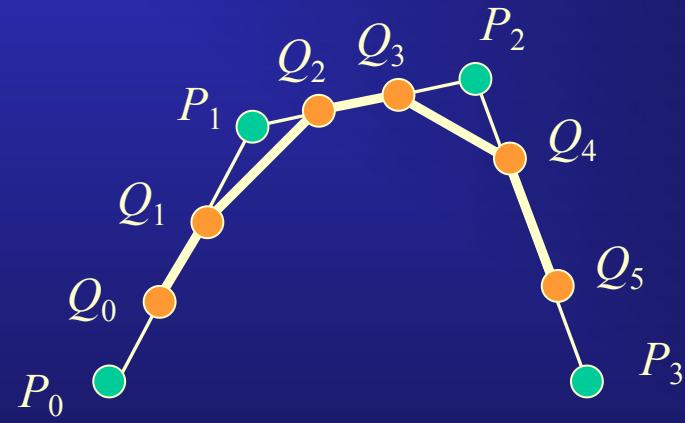


Subdivision in 3D



Refinement →

Chaiken's Algorithm for Iterative Subdivision



Subdivision “rule”:

$$Q_{2i} = \frac{3}{4}P_i + \frac{1}{4}P_{i+1}$$

$$Q_{2i+1} = \frac{1}{4}P_i + \frac{3}{4}P_{i+1}$$



Limit Curve or Surface as $i \rightarrow \infty$

$$\text{E.g.: } Q_0 = \frac{3}{4}P_0 + \frac{1}{4}P_1$$

$$Q_1 = \frac{1}{4}P_0 + \frac{3}{4}P_1$$

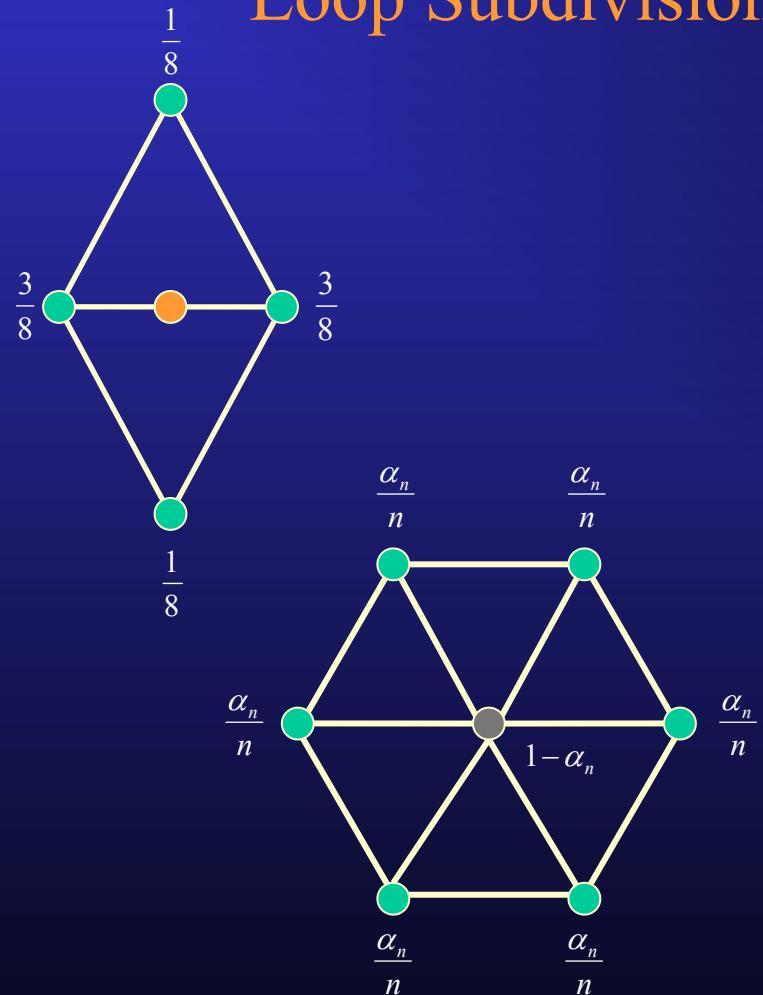
$$Q_2 = \frac{3}{4}P_1 + \frac{1}{4}P_2$$

$$Q_3 = \frac{1}{4}P_1 + \frac{3}{4}P_2$$

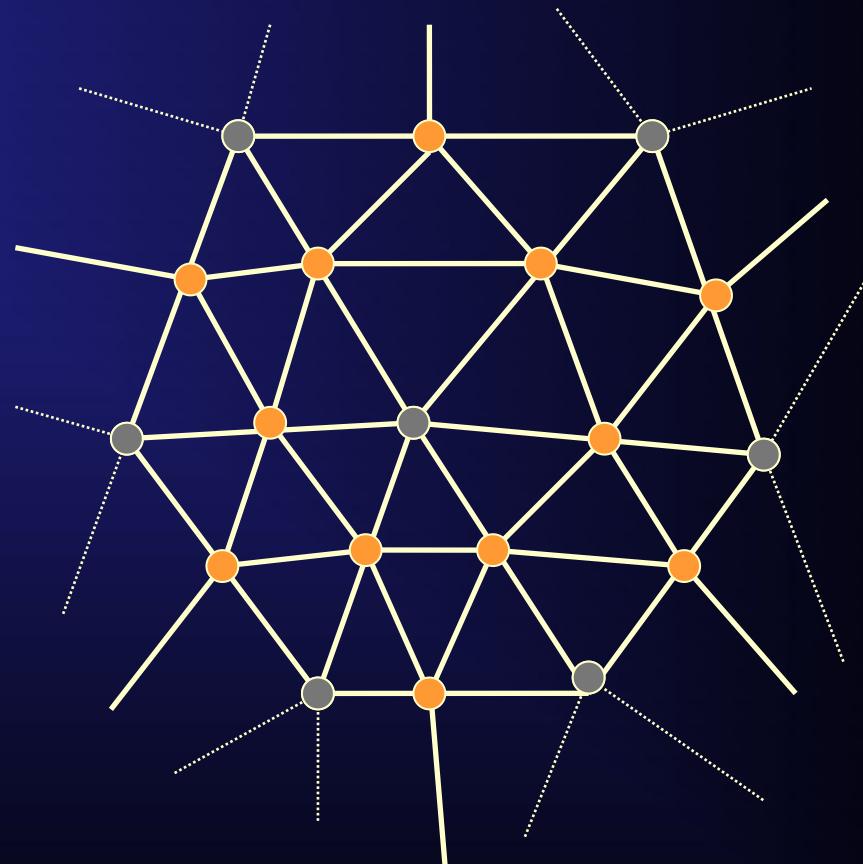
$$Q_4 = \frac{3}{4}P_2 + \frac{1}{4}P_3$$

$$Q_5 = \frac{1}{4}P_2 + \frac{3}{4}P_3$$

Loop Subdivision Masks (Triangle Meshes)

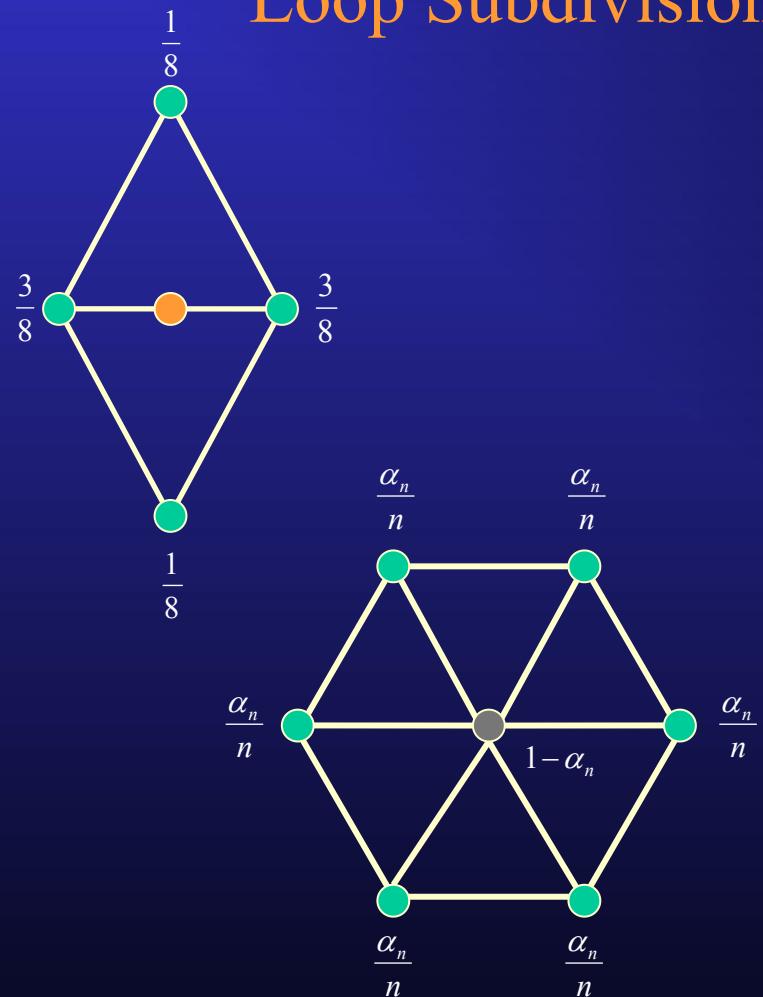


$$\alpha_n = \frac{1}{64} \left(40 - \left(3 + 2 \cos\left(\frac{2\pi}{n}\right) \right)^2 \right)$$

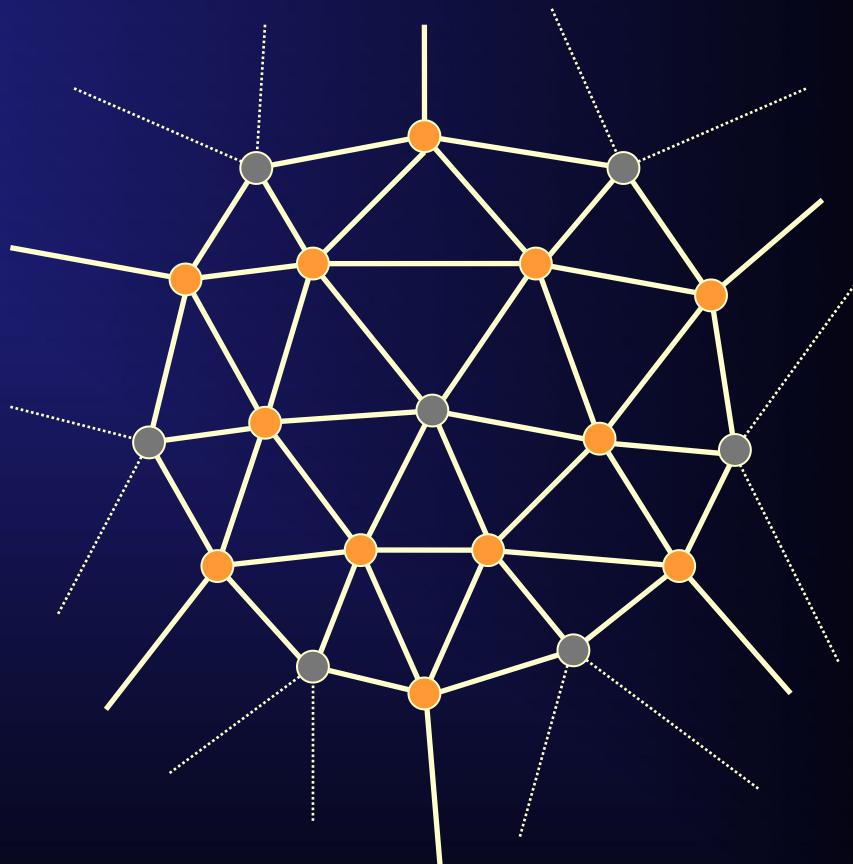


New coordinates for these vertices
($n = \#$ adjacent vertices)

Loop Subdivision Masks (Triangle Meshes)



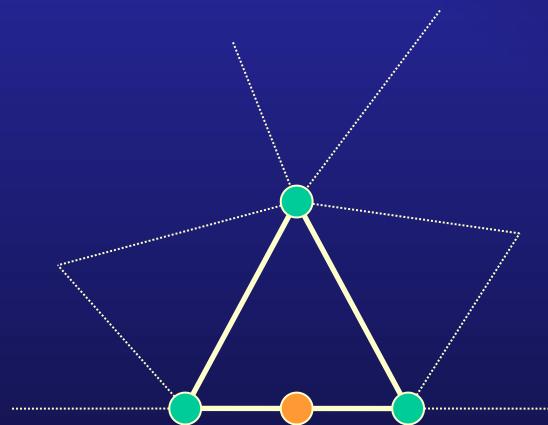
$$\alpha_n = \frac{1}{64} \left(40 - \left(3 + 2 \cos\left(\frac{2\pi}{n}\right) \right)^2 \right)$$



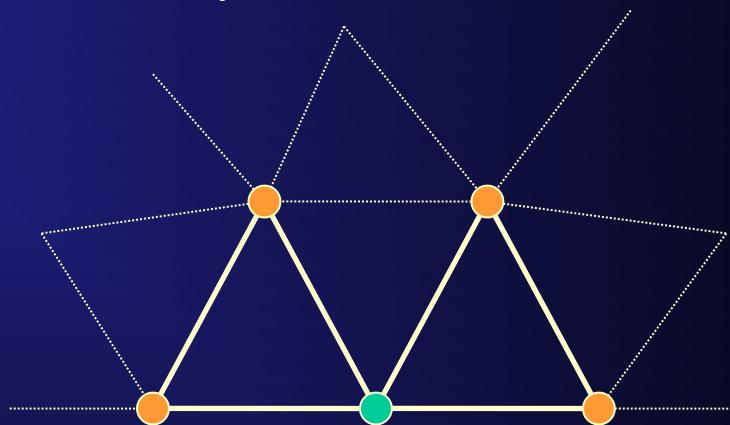
New coordinates for these \circlearrowright vertices
($n = \#$ adjacent vertices)

Loop Subdivision at Boundaries

- Subdivision mask for boundary conditions



Edge Rule



Vertex Rule

Catmull-Clark Subdivision (Originally Defined only for Quadrilateral Meshes) (1978)

Process all geometric elements for each iteration i starting from original mesh,
 $i \rightarrow 1, 2, 3, \dots$

- **FACE** (with m vertices; new vertex is at centroid)

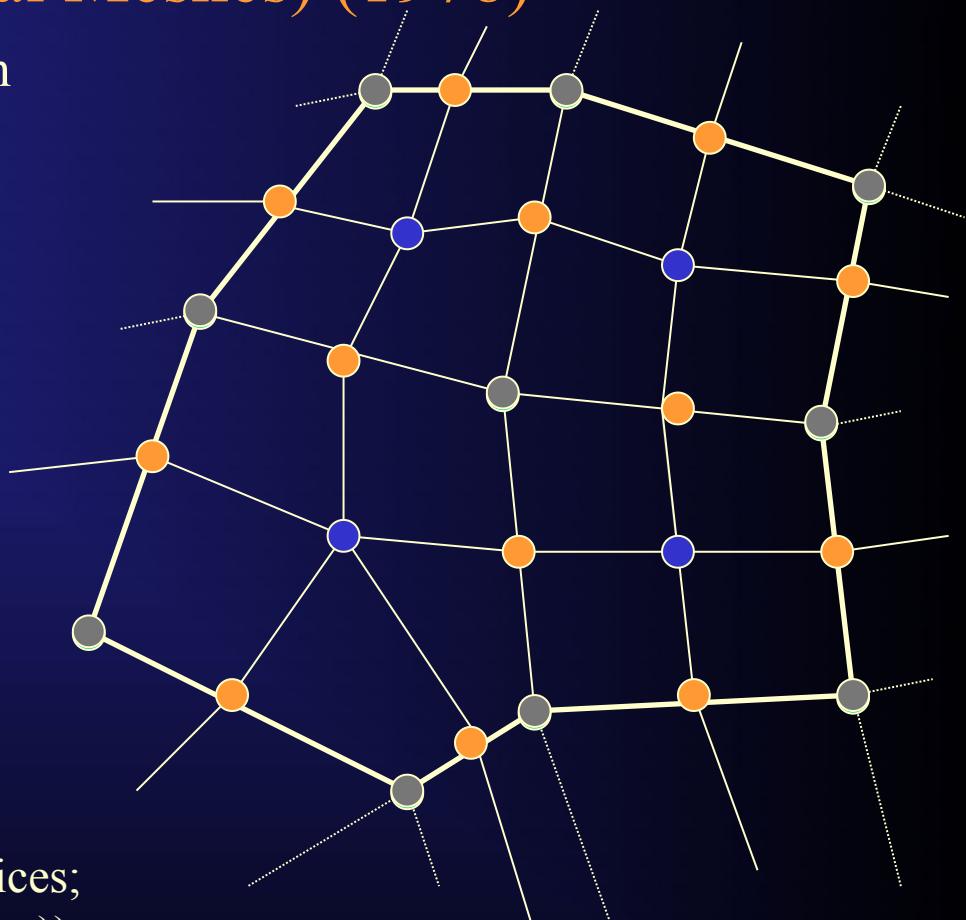
$$f = \frac{1}{m} \sum_1^m v_i$$

- **EDGE (new vertex)**

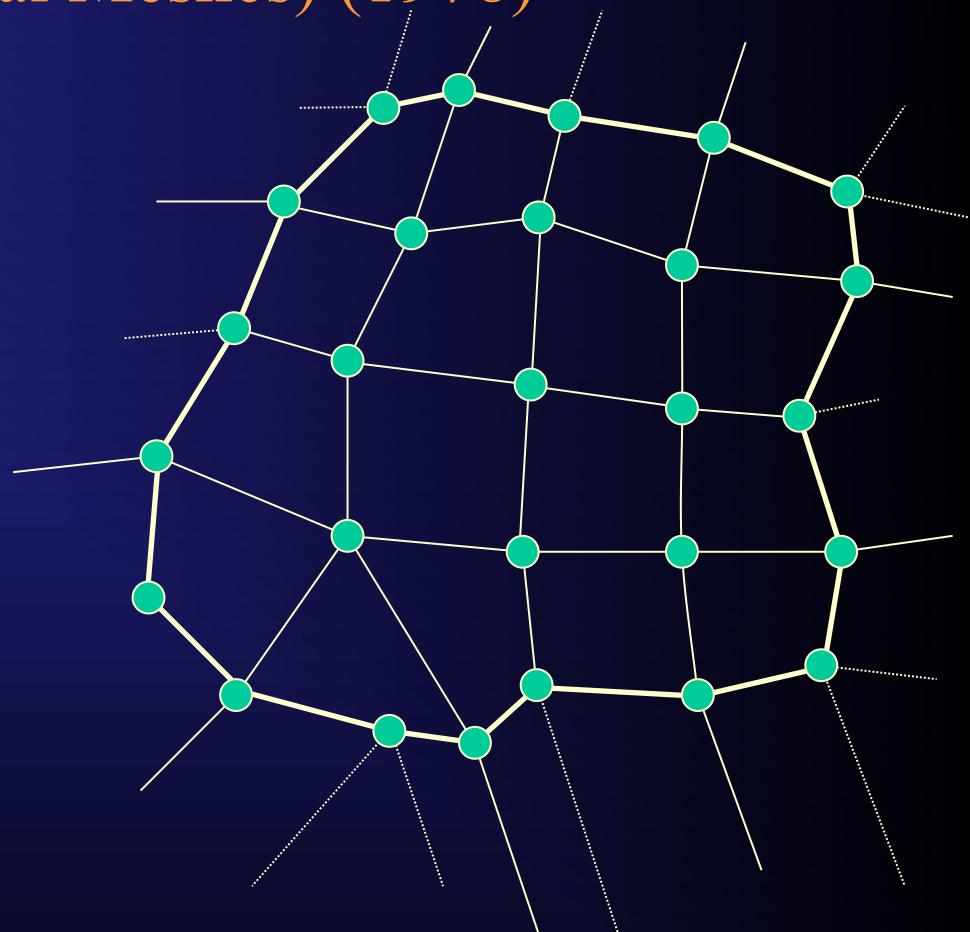
$$e = \frac{v_1 + v_2 + f_1 + f_2}{4}$$

- → ○ **VERTEX** ($n = \#$ of adjacent vertices;
 $j = \#$ of incident edges ($= \#$ of faces))

$$v_{i+1} = \frac{n-2}{n} v_i + \frac{1}{n^2} \sum_j e_j + \frac{1}{n^2} \sum_j f_j$$



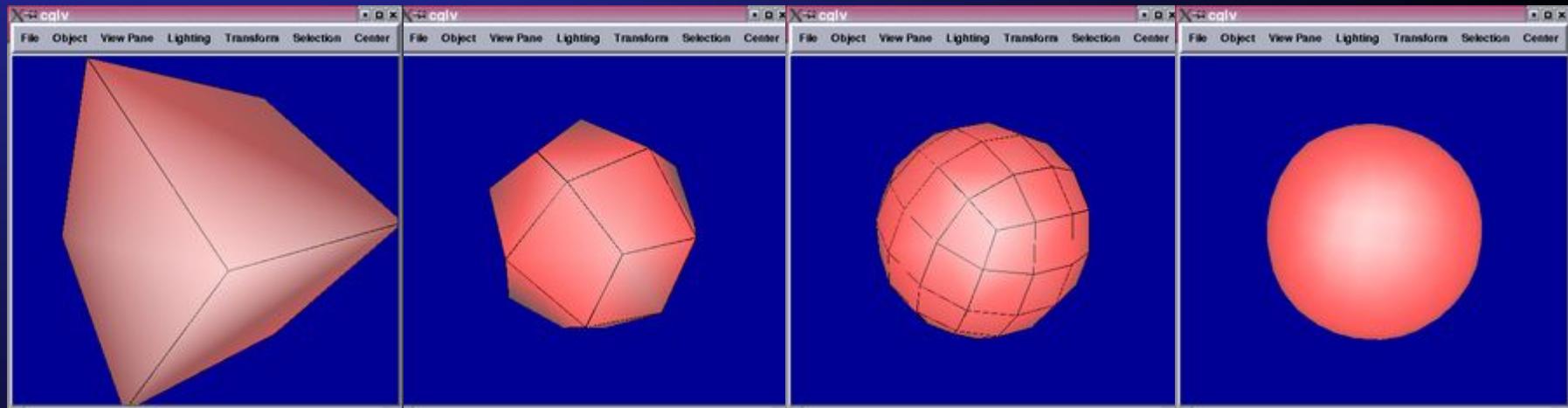
Catmull-Clark Subdivision (Originally Defined only for Quadrilateral Meshes) (1978)



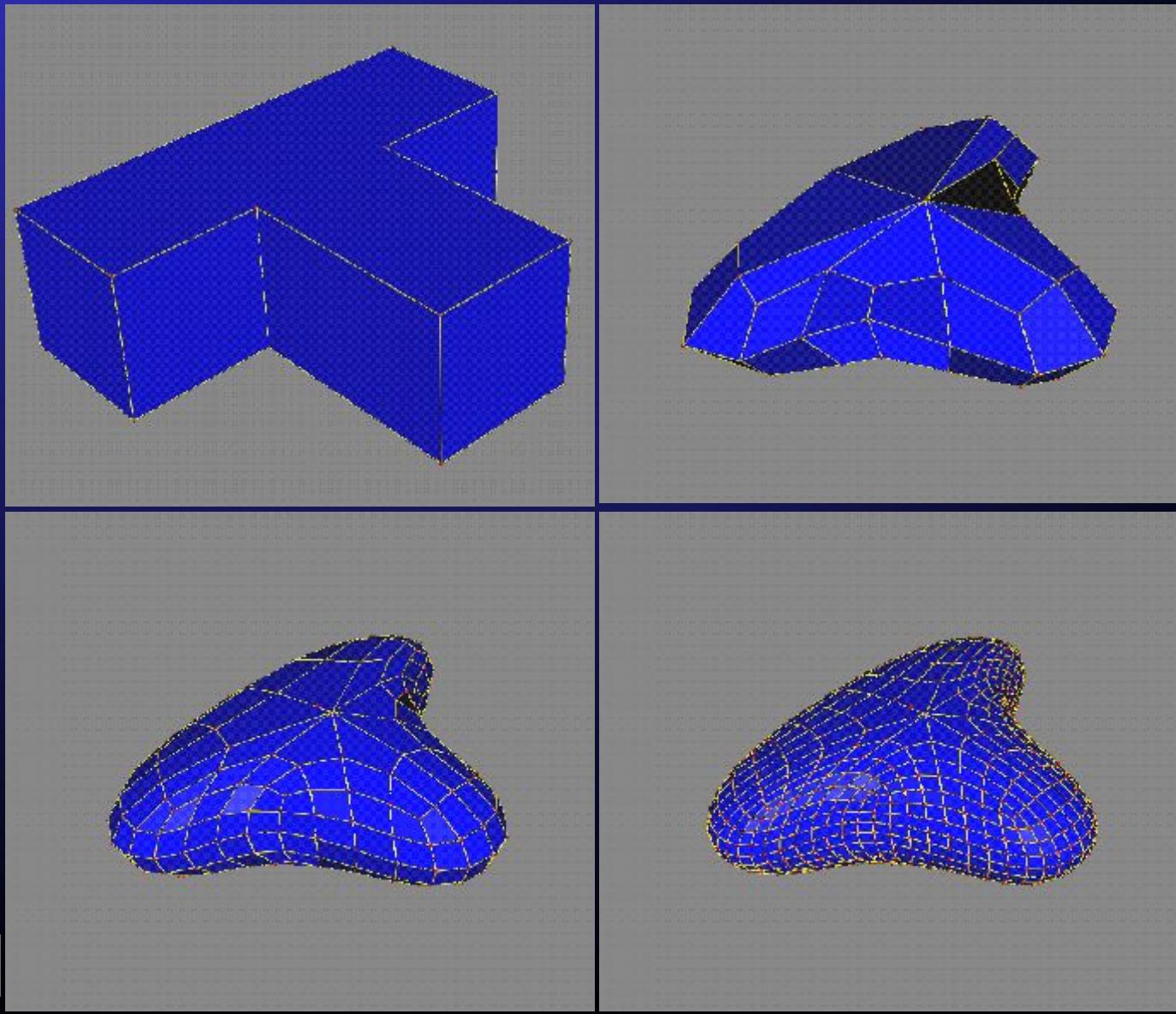
Ready for next iteration! $i = +1$

Catmull-Clark Example

- A cube converges to a sphere.



Catmull-Clark Example



Catmull-Clark Surfaces

- Iterative subdivision produces smooth continuous surfaces.
- But natural surfaces might require varying degrees of “sharpness” and creases.
- Create new subdivision rules for “creased” edges and vertices:
 - Tag “sharp” edges in mesh.
 - If an edge is sharp, apply new subdivision rules.
 - Others subdivide normally.



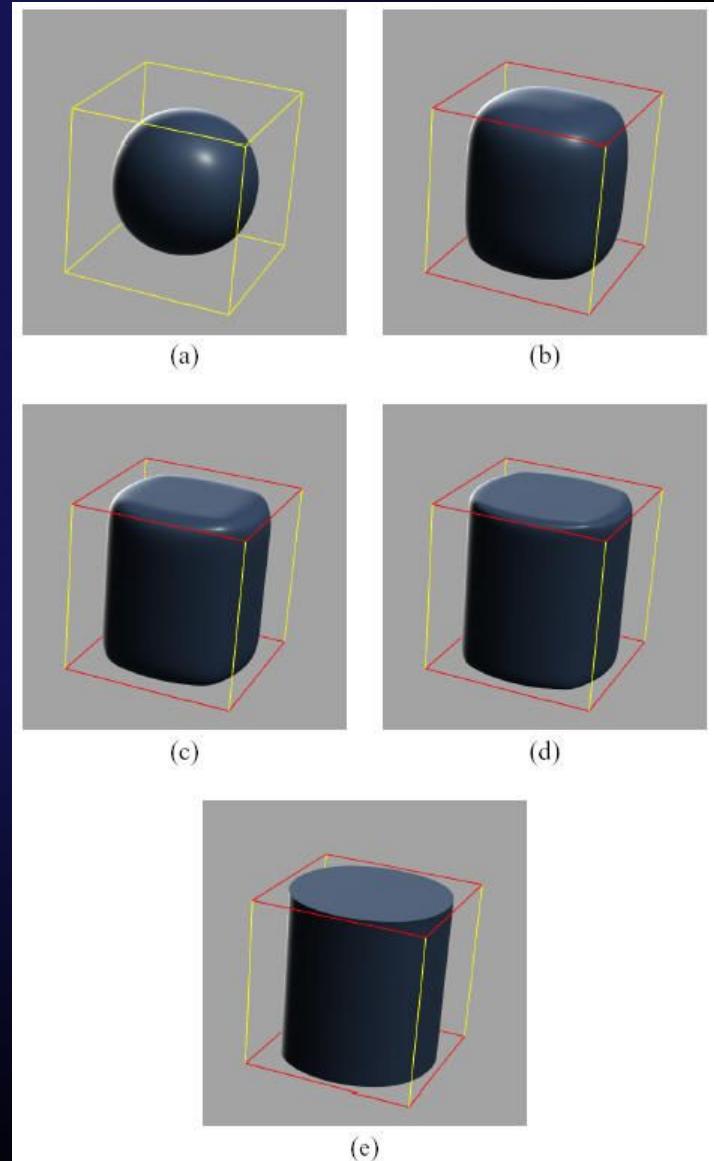
Subdividing to Obtain Sharp Edges

1. Tag Edges as **sharp** or **not-sharp**:
 - $n = 0 \Rightarrow \text{not sharp}$
 - $n > 0 \Rightarrow \text{sharp}$

During subdivision,

2. if an edge is **sharp**, use sharp subdivision rules. Newly created edges are assigned a sharpness of $n-1$.
3. If an edge is **not-sharp**, use normal smooth subdivision rules.

Result: Edges with a sharpness of “ n ” are not subdivided smoothly for “ n ” iterations of the algorithm.



Sharp Rules

- FACE (unchanged): $f = \frac{1}{n} \sum_1^n v_i$

- EDGE: $e = \frac{v_1 + v_2}{2}$

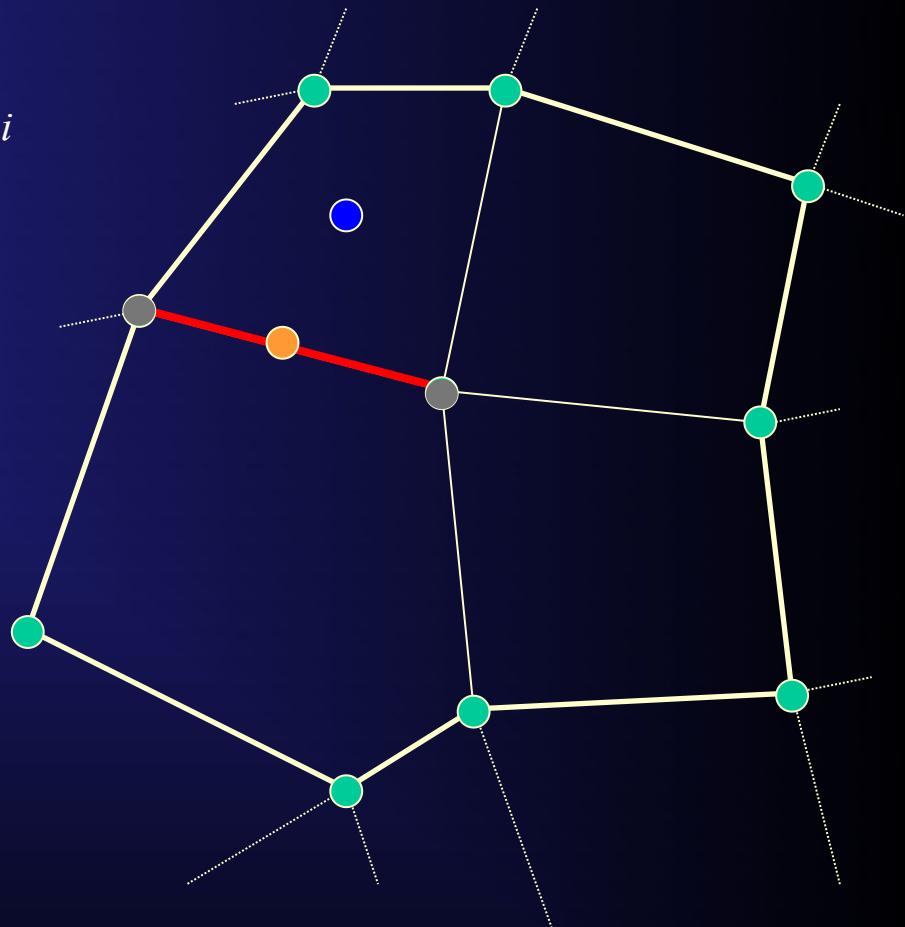
 →  VERTEX:

adj. Sharp edges

dart >2 $v_{i+1} = v_i$

crease 2 $v_{i+1} = \frac{e_1 + 6v_i + e_2}{8}$

**corner
(unchanged)** $0,1$ $v_{i+1} = \frac{n-2}{n} v_i + \frac{1}{n^2} \sum_j e_j + \frac{1}{n^2} \sum_j f_j$



Another Look at Curves

- Bezier curves use Bernstein polynomials as *blending* functions; these are nonzero over the entire interval (endpoint to endpoint).
- Each control point thus affects the shape of the entire curve between the endpoints.
- That's why low degree polynomials are used in Bezier curves; too many control points change the curve globally in possibly undesirable ways: there is no *local control*.
- Check the demo again with higher degree Bezier curves:



Bezier demo

<http://www2.mat.dtu.dk/people/J.Gravesen/cagd/decast.html>

Better Blending Functions

- These blending functions only affect the curve over a given interval, e.g.:

$R_0(t)$ over $[0, 0.2]$

$R_1(t)$ over $[0, 0.4]$

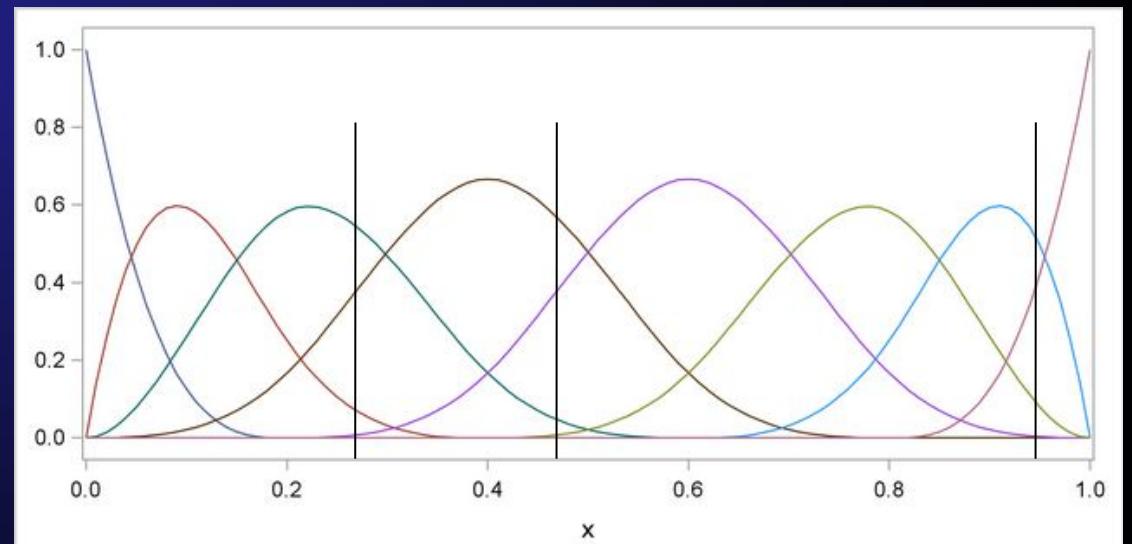
$R_2(t)$ over $[0, 0.6]$

$R_3(t)$ over $[0.2, 1]$

$R_4(t)$ over $[0.4, 1]$

$R_5(t)$ over $[0.6, 1]$

$R_6(t)$ over $[0.8, 1]$



- The interval over which a function is nonzero is called its *support*.
- At each point the curve depends on only four control points. ▶

Splines

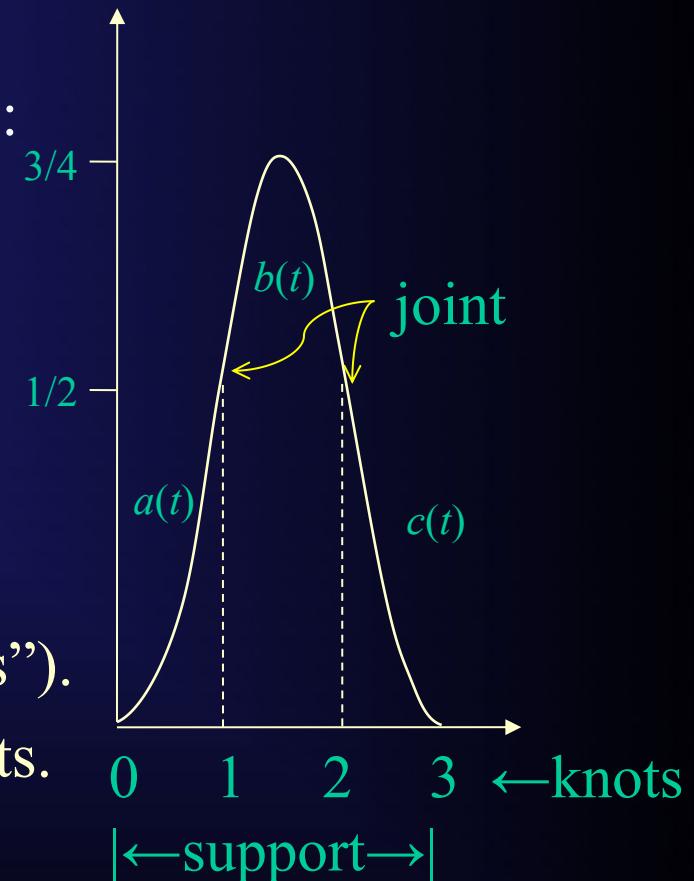
- A *spline* is a curve constructed by piecing together several other curves, e.g., low degree polynomials.
- E.g., $g(t)$ is a (quadratic) *spline function*:

$$a(t) = \frac{1}{2}t^2, \quad [0,1]; 0 \text{ elsewhere}$$

$$b(t) = \frac{3}{4} - \left(t - \frac{3}{2}\right)^2, \quad [1,2]; 0 \text{ elsewhere}$$

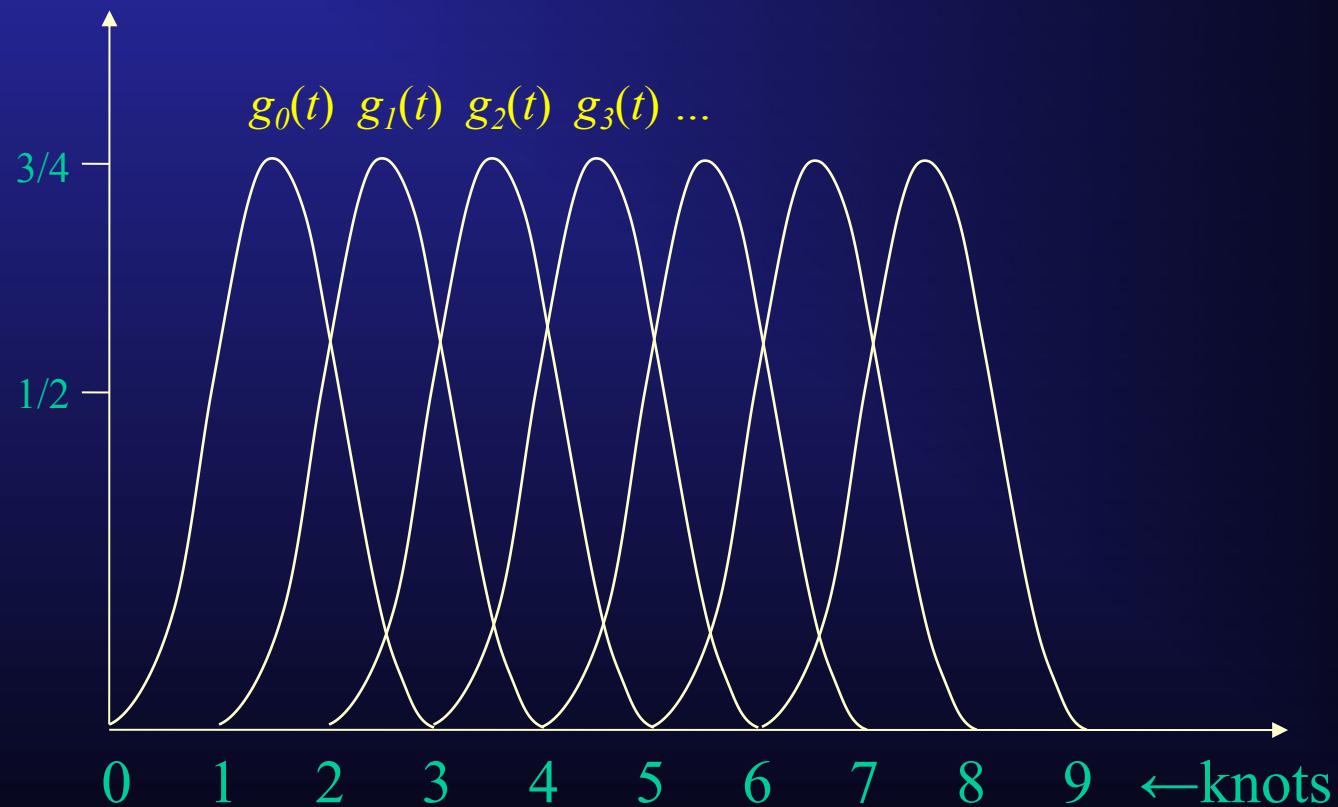
$$c(t) = \frac{1}{2}(3-t)^2, \quad [2,3]; 0 \text{ elsewhere}$$

- The polynomial segments must have:
 - The same values at the knots (“joints”).
 - The same first derivatives at the knots.
 - (Check that!)
- An m^{th} degree spline function is a piecewise polynomial of degree m that is $(m-1)$ -smooth at each knot.



Using Shifted Versions for Blending

$$g_k(t) = g(t - k) \text{ for integers } k = 0, 1, 2, \dots$$



Allows long sequences of control points to be blended
into a curve without numerical issues.

B-Splines

$$P(t) = \sum_{k=0}^6 P_k g(t - k)$$

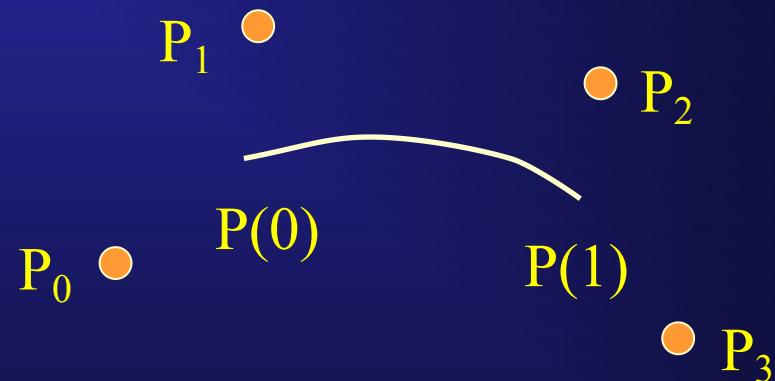
- Or, more generally: $P(t) = \sum_{k=0}^n P_k R_k(t)$

for $n+1$ control points P_k and $n+1$ blending functions $R_k(t)$.

- The $R_k(t)$ should be piecewise polynomials defined on a *knot vector* $T=(t_0, t_1, t_2, \dots)$, $t_i \leq t_{i+1}$, i.e., some knots may have the same value.
- B-splines (basic splines) have nice properties, including:
 - Interpolation through certain control points
 - Minimal support, and thus
 - Local shape control.
 - Repeated knots decrease order of continuity.

Cubic (4th Order) B-Splines

- Use 4 points but approximate curve with middle two:



- Draw curve with overlapping segments 0-1-2-3, 1-2-3-4, 2-3-4-5, etc.

Bezier Curve is a Special Case of B-Splines

- All the Bernstein polynomials are of degree n (e.g., 3).
- There is one span (interval of support between knots) from 0 to 1.
- There are $n+1$ knots at $t=0$ and $n+1$ more knots at $t=1$.
- So Bezier curves are B-splines, since they have continuous derivatives of order n .

B-Spline Definition

- Given:
 - The degree n (order -1) of the B-spline functions
 - m real valued knots $t_0 \leq t_1 \leq t_2 \leq \dots t_{m-1}$
 - $m-n+1$ control points P_i

- Define $P(t) = \sum_{i=0}^{m-n} P_i b_{i,n}(t) \quad t \in [t_{n-1}, t_{m-n}]$

- Where $b_{i,n}$ are recursively defined (Cox-de Boor) blending functions:

$$b_{j,0}(t) = \begin{cases} 1 & \text{if } t_j < t < t_{j+1} \\ 0 & \text{otherwise} \end{cases}$$

$$b_{j,n}(t) = \frac{t - t_j}{t_{j+n} - t_j} b_{j,n-1}(t) + \frac{t_{j+n+1} - t}{t_{j+n+1} - t_{j+1}} b_{j+1,n-1}(t)$$

For example, Quadratic B-splines

- With equally spaced knots:

$$b_{0,3}(t) = \frac{t}{2} b_{0,2}(t) + \frac{3-t}{2} b_{1,2}(t)$$

- Which is exactly the blending function $g(t)$ we saw earlier.
- $b_{0,3}$ depends on the 4 knots 0, 1, 2, 3, so its support is the interval $[0, 3]$.
- Exercise: Compute the values of the B-spline at $t=2, 2.5, 3, 3.5$.

Uniform and Non-Uniform B-Splines

- When the knots are equidistant, the B-spline is called *uniform*, otherwise it is *non-uniform*.
- (Note: if any knots are the same, then interpret 0/0 as 0 in the recursive definition.)
- Non-uniform B-splines:
 - Allow discontinuities to be arbitrarily spaced
 - Permit better control over continuity or interpolation effects at control points
 - Inherit local control
- *Non-uniform rational B-splines (NURBS)* are weighted ratios of non-uniform B-splines:
 - Are perspective as well as affine invariant.
 - Represent conic sections exactly!

NURBS

- Exploit homogeneous coordinates:

$$P_i = \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} \cong w_i \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix} = Q_i$$

- Where the w_i can be interpreted as weights on each control point.
- Renormalize:

$$P(u) = \frac{\sum_{i=0}^n b_i(u) w_i P_i}{\sum_{i=0}^n b_i(u) w_i}$$

Cubic B-Splines

$$b_{0,4}(t) = \begin{cases} u(1-t) & 0 \leq t \leq 1 \\ v(2-t) & 1 \leq t \leq 2 \\ v(t-2) & 2 \leq t \leq 3 \\ u(t-3) & 3 \leq t \leq 4 \end{cases}, 0 \text{ otherwise}$$

where

$$u(t) = \frac{1}{6}(1-t)^3$$

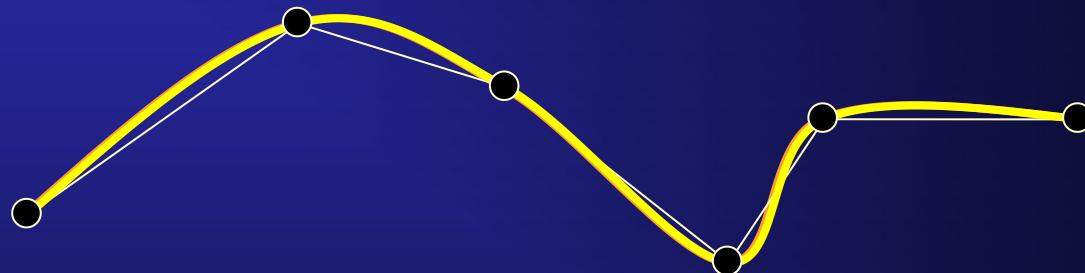
$$v(t) = \frac{1}{6}(3t^3 - 6t^2 + 4)$$

$$P_i(t) = [t^3 \quad t^2 \quad t \quad 1] \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} P_{i-1} \\ P_i \\ P_{i+1} \\ P_{i+2} \end{bmatrix}, \text{ for } t \in [0,1]$$

Multiple (Repeated) Knots

- The resulting B-spline is only m smooth at the multiple knot, i.e., cubic splines have a discontinuous first derivative (a “kink”) at a knot of multiplicity 3.
- As parameter t approaches a knot of multiplicity > 1 , there is a stronger “pull” toward the governing control point.
- A B-spline curve of order m interpolates (passes exactly through) the governing control point of a knot with multiplicity m .
- A typical B-spline of order m begins and ends with a knot of multiplicity m (so that the curve goes through the endpoints).
- Example: for 8 control points and cubic ($m=4$) B-splines, the uniform knot vector would be $(0,0,0,0,1,2,3,4,5,5,5,5)$.

PowerPoint's Curves are?



PowerPoint demo

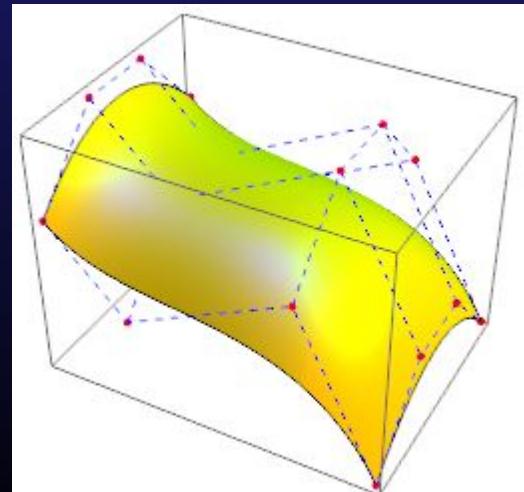


B-Spline Surface Patches

- Use B-splines instead of Bernstein polynomials:

$$P(s, t) = \sum_{i=0}^m \sum_{k=0}^n P_{i,k} b_{i,k}(s) b_{k,n}(t)$$

where the b 's are the B-spline basis functions.



Demo

Implicit Surfaces

- Solution to general equation $F(x,y,z) = 0$.
- Special cases:
 - Quadric (polynomial degree at most 2) surfaces (sphere, ellipsoid, cone, paraboloid, hyperboloid).
 - Superellipsoids.
 - Potential functions.
 - Requires **iterative** numerical techniques if F is not **analytic** (algebraically solvable).
 - **Marching squares** and **marching cubes**: “walk” along a discrete grid/volume looking for edges that contain pairs of vertices where one is inside and the other is outside. (Note similarity to volume rendering.)
 - We’ll see this technique used later.

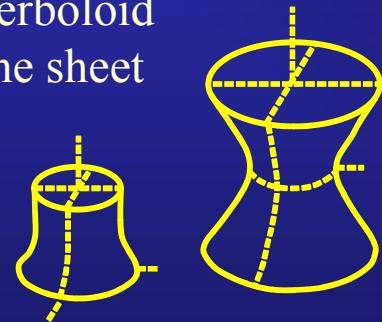
Implicit Surface



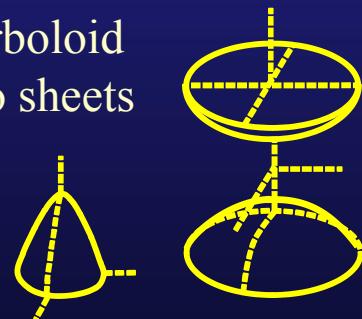
© 1982 PRUEITT, M.

Quadric Surfaces

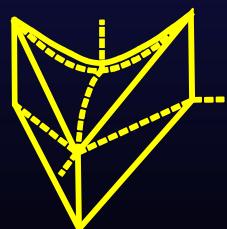
Hyperboloid
of one sheet



Hyperboloid
of two sheets



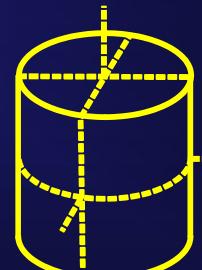
Hyperbolic
paraboloid



Parabolic
cylinder



Elliptic
cylinder



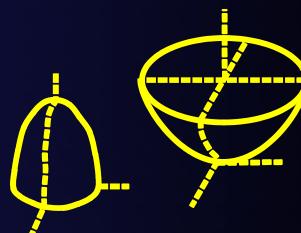
Hyperboloid
cylinder



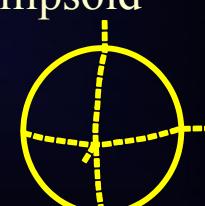
Cone



Elliptic paraboloid



Ellipsoid



Superellipsoids (Parametric form)

- One interesting class of implicit surfaces: Superellipsoids:

$$N_x(\phi, \beta) = \frac{1}{r_x} \cos^{2-n_1}(\phi) \cos^{2-n_2}(\beta)$$

$$N_y(\phi, \beta) = \frac{1}{r_y} \cos^{2-n_1}(\phi) \sin^{2-n_2}(\beta)$$

$$N_z(\phi, \beta) = \frac{1}{r_z} \sin^{2-n_1}(\phi)$$

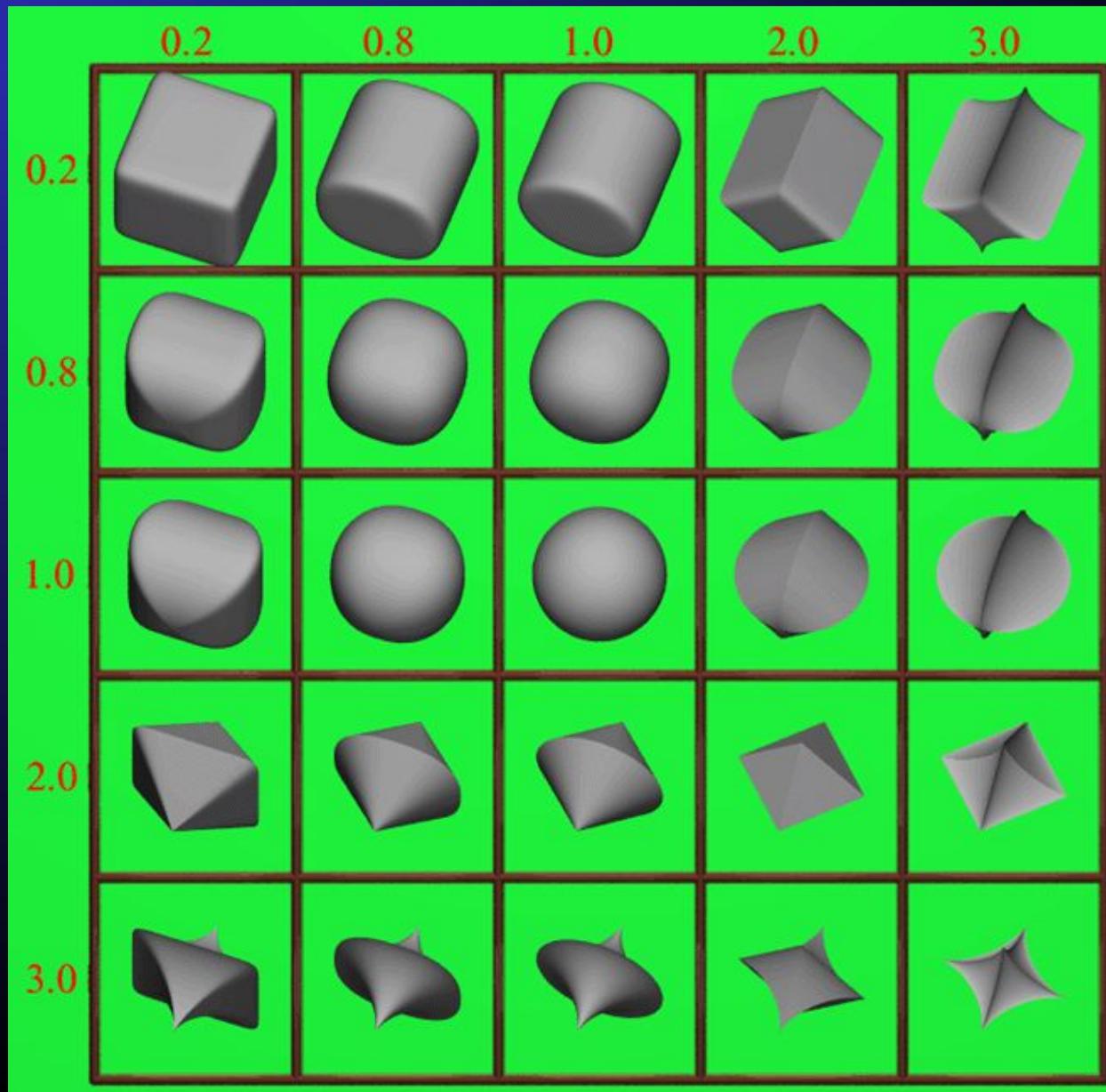
- Notice if $n_1=n_2=1$ this is just the polar coordinate formula for a sphere.

Superellipsoid:

(Implicit form) $f(x, y, z) = \left[\left(\frac{x}{r_x} \right)^{2/n_2} + \left(\frac{y}{r_y} \right)^{2/n_2} \right]^{n_2/n_1} + \left(\frac{z}{r_z} \right)^{2/n_1}$

| n_1 | n_2 | rx, ry, rz | form |
|--------------------|----------|--------------|---------------|
| 0 | 0 | | box |
| 0 | 0 | $rx=ry=rz$ | cube |
| <1 | <1 | | cuboid |
| <1 | 1 | | pillow shapes |
| 1 | <1 | | cylindrical |
| 1 | 1 | | ellipsoid |
| 1 | 1 | $rx=ry=rz$ | sphere |
| 2 | 2 | $rx=ry=rz$ | octahedron |
| n_1 or $n_2 > 2$ | | | pinched |
| n_1 or $n_2 = 2$ | | | bevelled |
| ∞ | ∞ | | 3D “jack” |

Varying the Superellipsoid Exponents

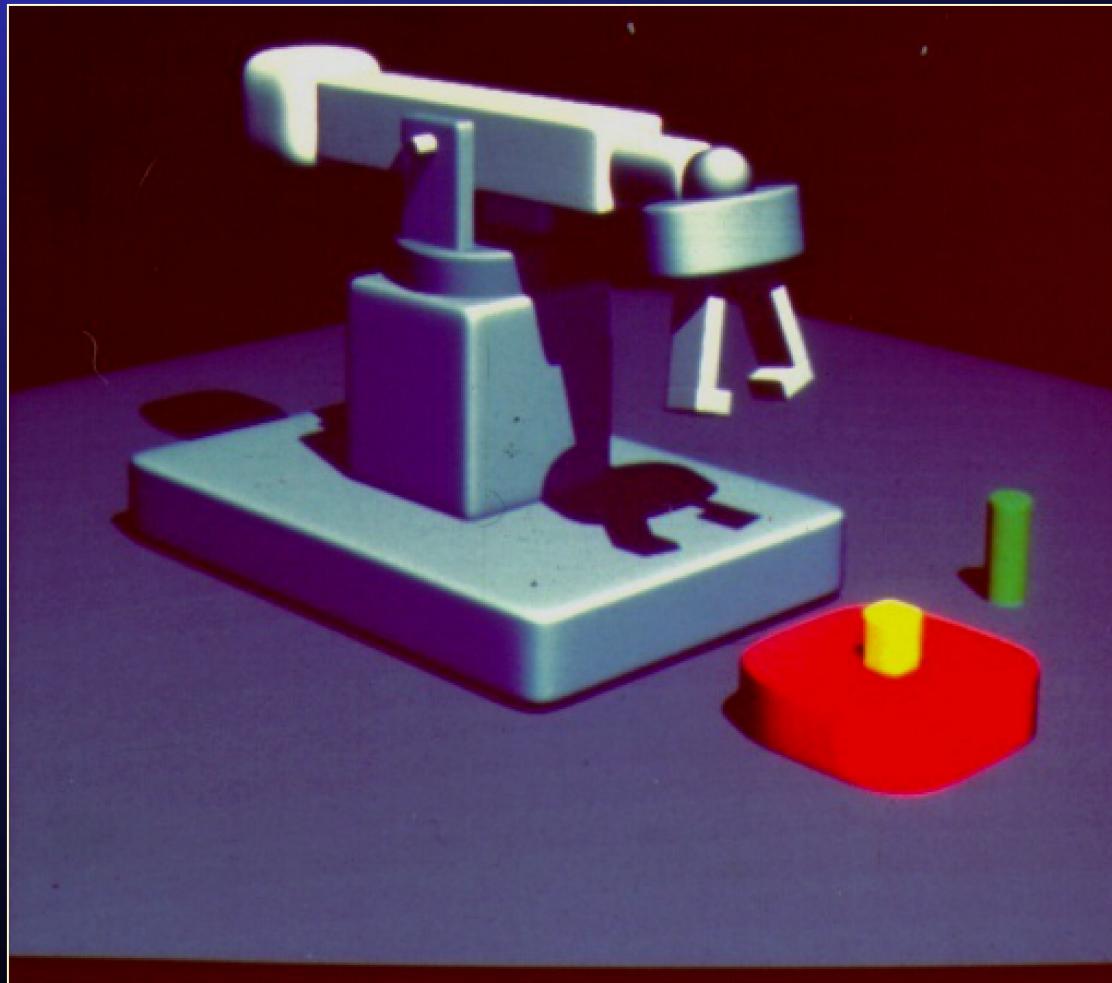


Some Superellipsoids



© 1985 R. KUCHKUDA — MEGATEK CORP.

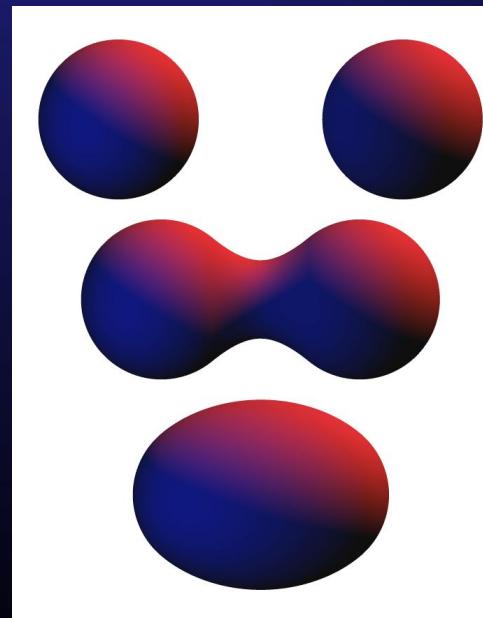
All Geometric Components Here are Superellipsoids!



BARR, A.—RASTER TECHNOLOGIES

Potential Functions (Based on Spherical Shapes)

- Center point.
- Radius-dependent decreasing value.
- Act like energy fields, summing when overlapping.
- Smooths intersections between potentials.

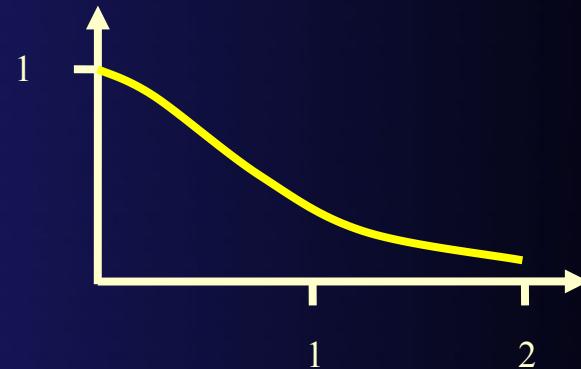


<http://commons.wikimedia.org/wiki/File:Metaballs.png>

Potential Functions: Different Formulations

- Blobs (Blinn 82):

$$e^{-ar^2}$$

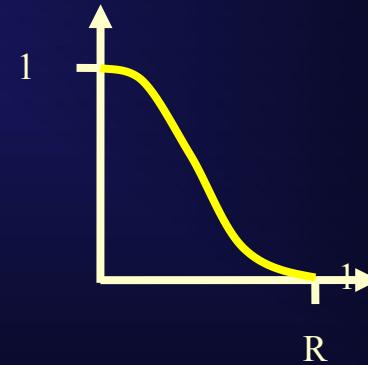


- Metaballs (Nishimura 83):

$$1 - 3\left(\frac{r}{R}\right)^2 \quad 0 \leq r \leq \frac{R}{3}$$

$$\frac{3}{2}\left(1 - \frac{r}{R}\right)^2 \quad \frac{R}{3} \leq r \leq R$$

$$0 \quad R < r$$



- Soft objects (Wyvill 86) –
polygonal approximation to
Blinn $\exp(-)$ blobs.

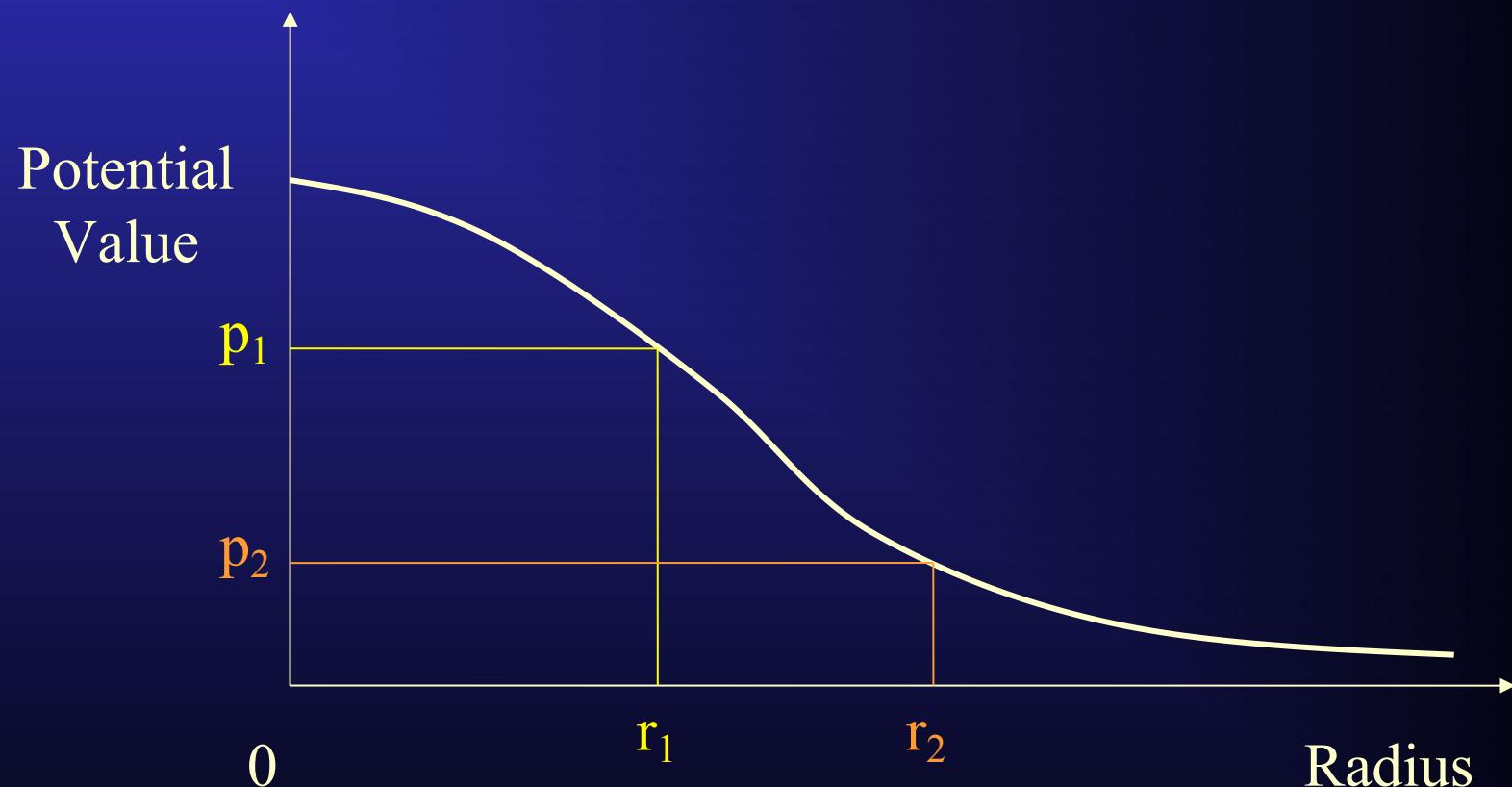
(See local.wasp.uwa.edu.au/~pbourke/miscellaneous/implicitsurf)

Organic Form



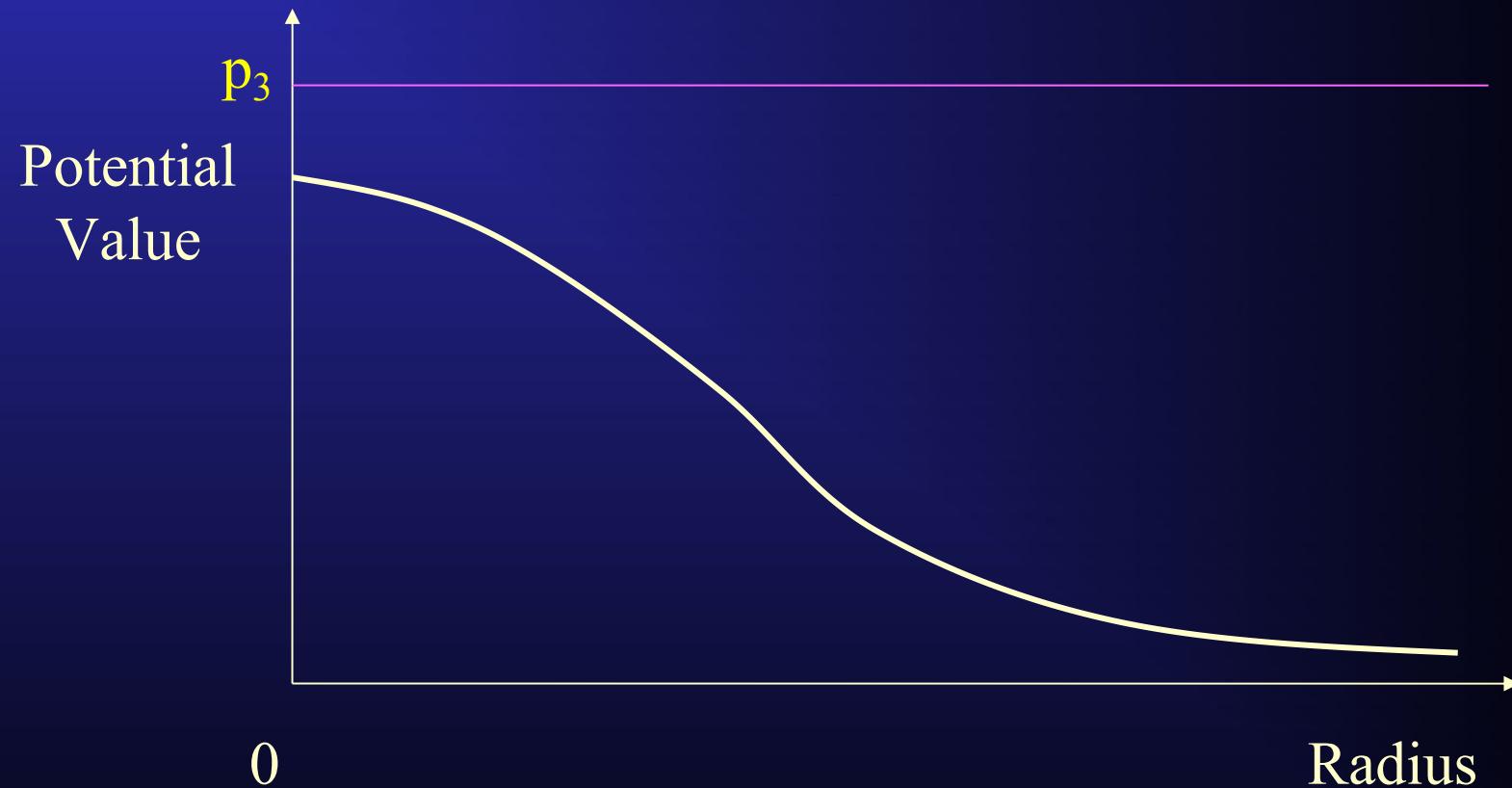
© 1984 FRANK DEITRICH—WEST COAST UNIVERSITY

Potential Function Shape; e.g., $\exp()$;
Needs a Threshold (Level) to Be Visible



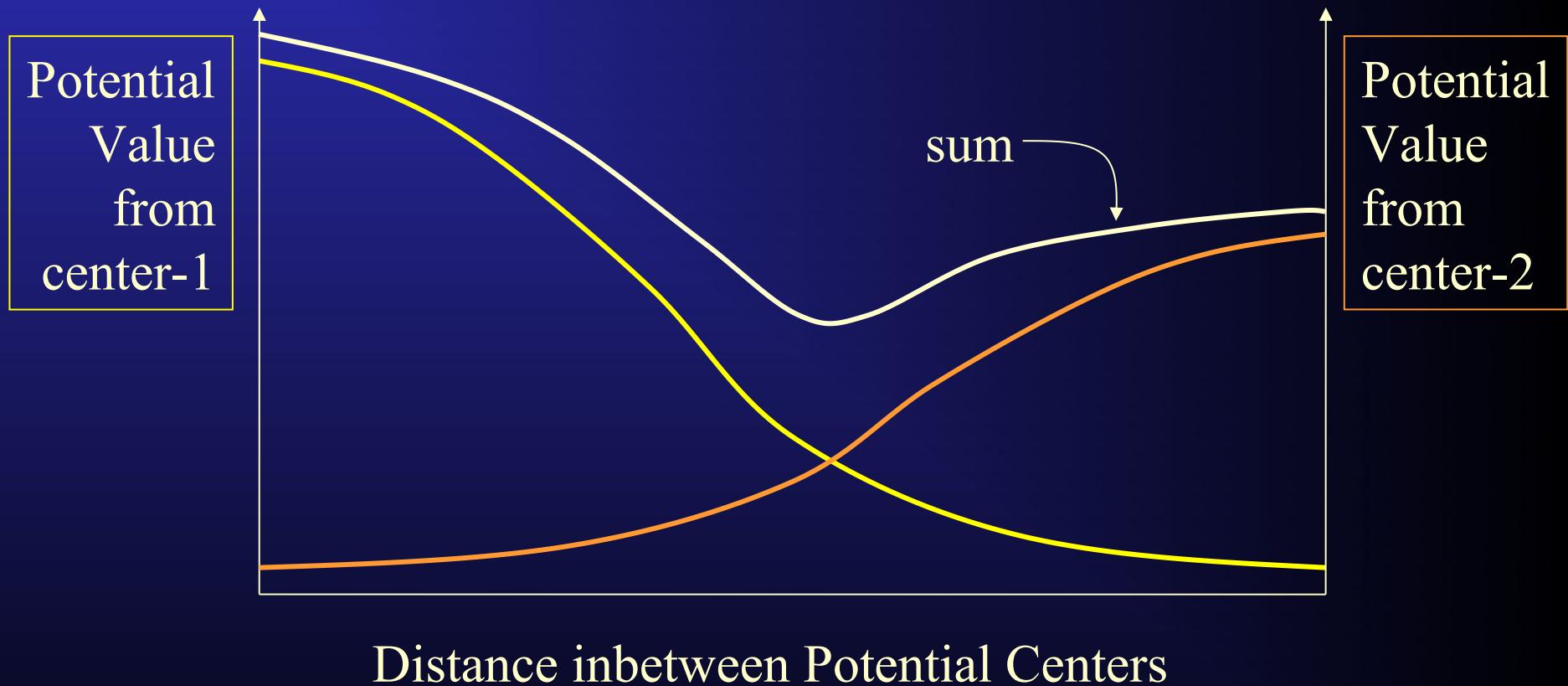
Choosing a threshold defines a sphere or iso-surface or *level set*:
The lower the potential, the larger the sphere radius

The Threshold Matters



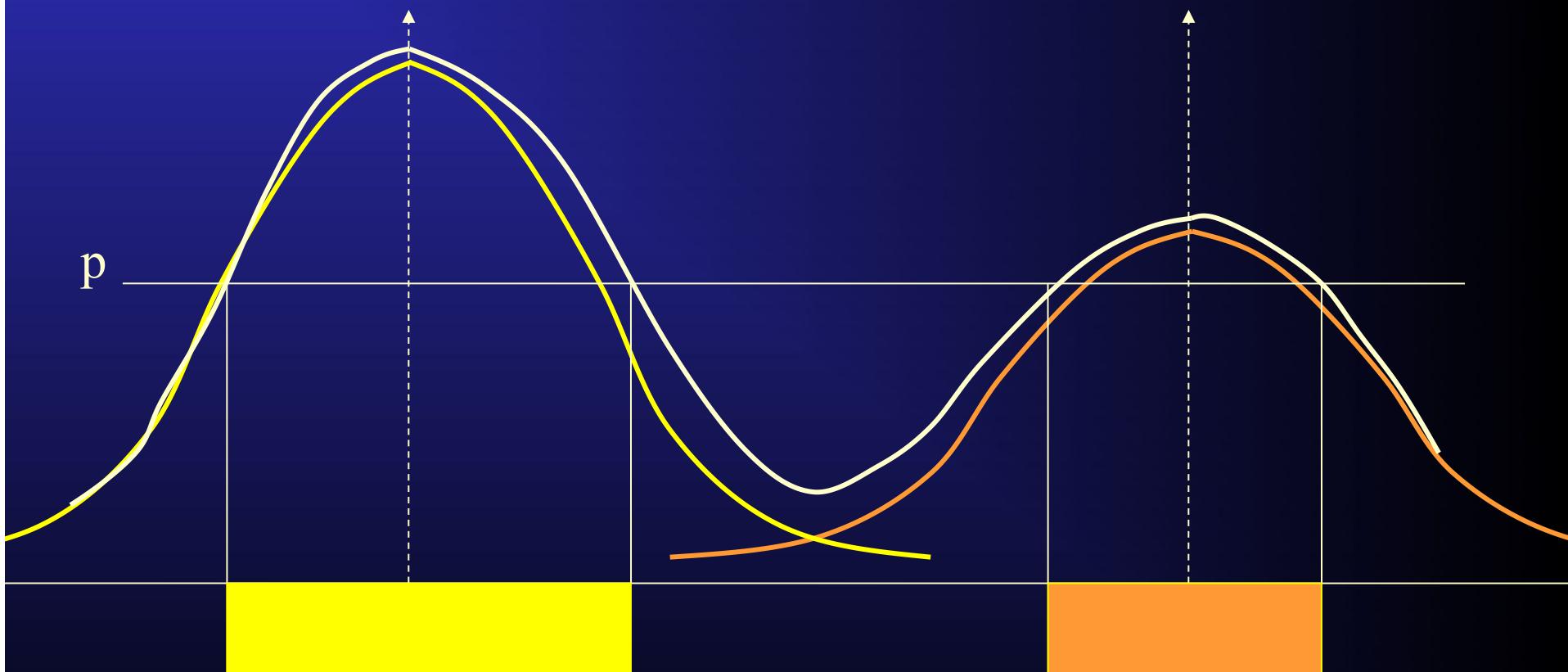
If the threshold is greater than the potential, we'll see nothing: there is no iso-surface of that potential field of that value.

Potential Function Overlaps Make Smooth Shapes



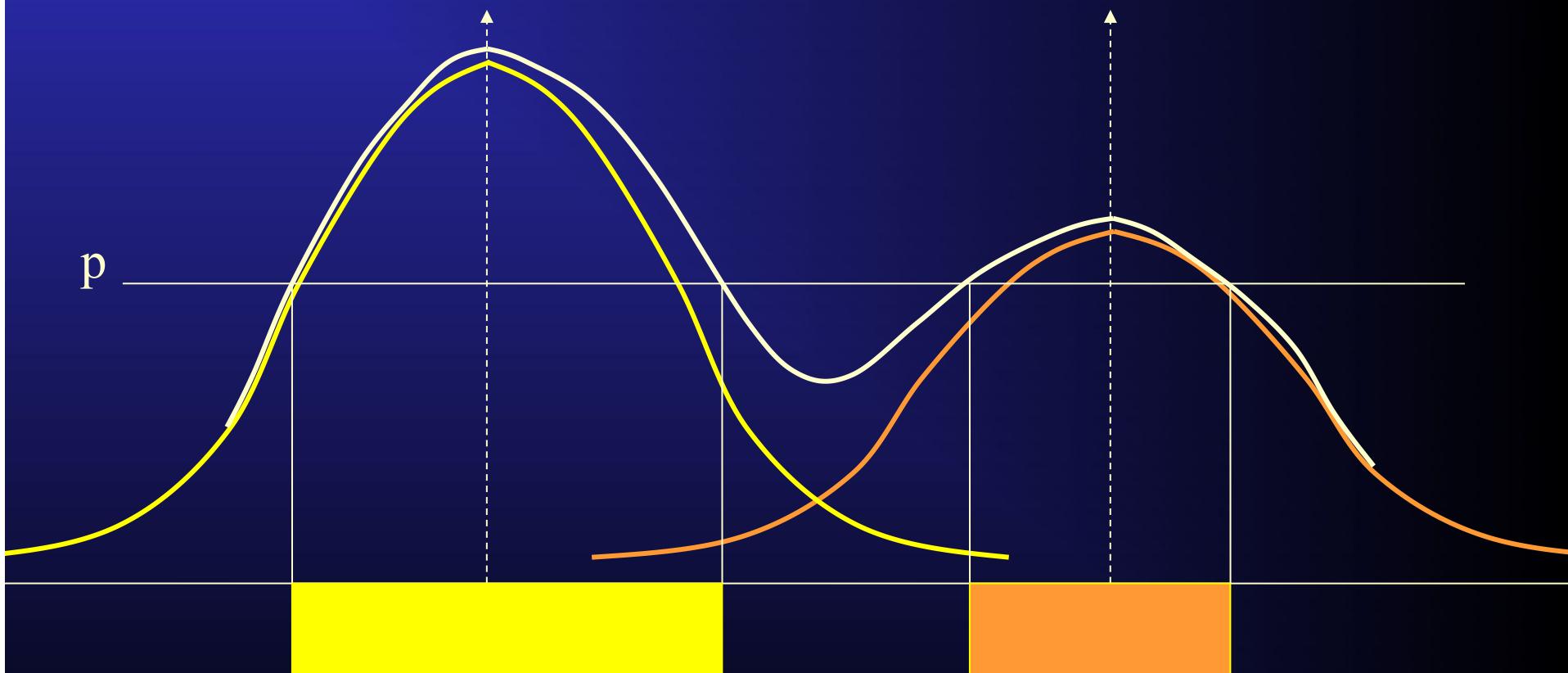
If two potential function are close, their potentials **SUM**.
Notice that the sum is **SMOOTH**, not creased!

For a Given Potential Threshold, Two Approaching Fields Merge



Two separate iso-surfaces

For a Given Potential Threshold, Two Approaching Fields Merge



Two separate iso-surfaces

For a Given Potential Threshold, Two Approaching Fields Merge



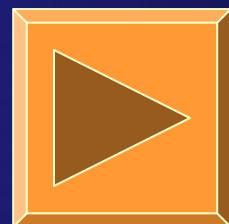
Two iso-surfaces just join (smoothly)

For a Given Potential Threshold, Two Approaching Fields Merge

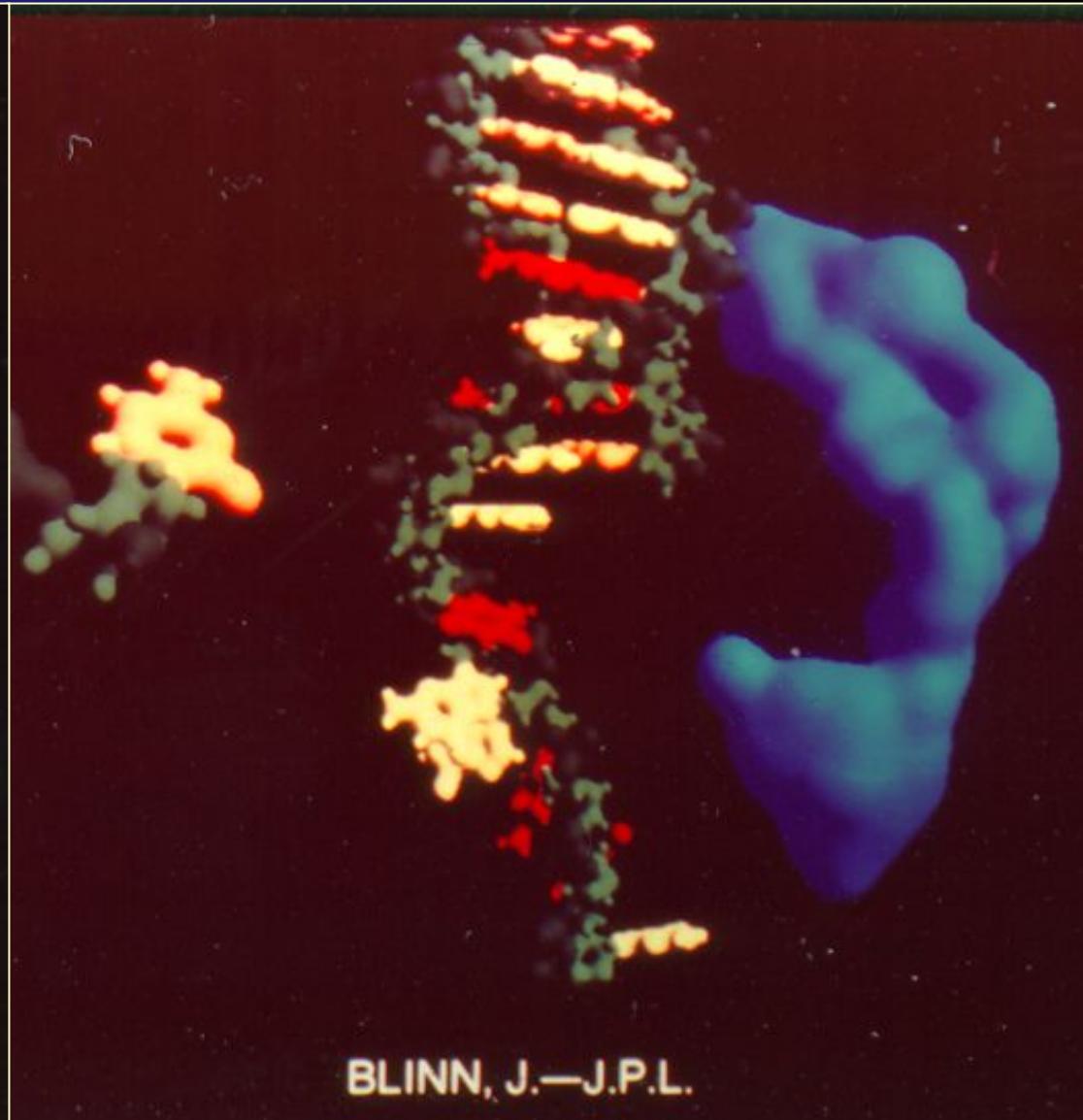
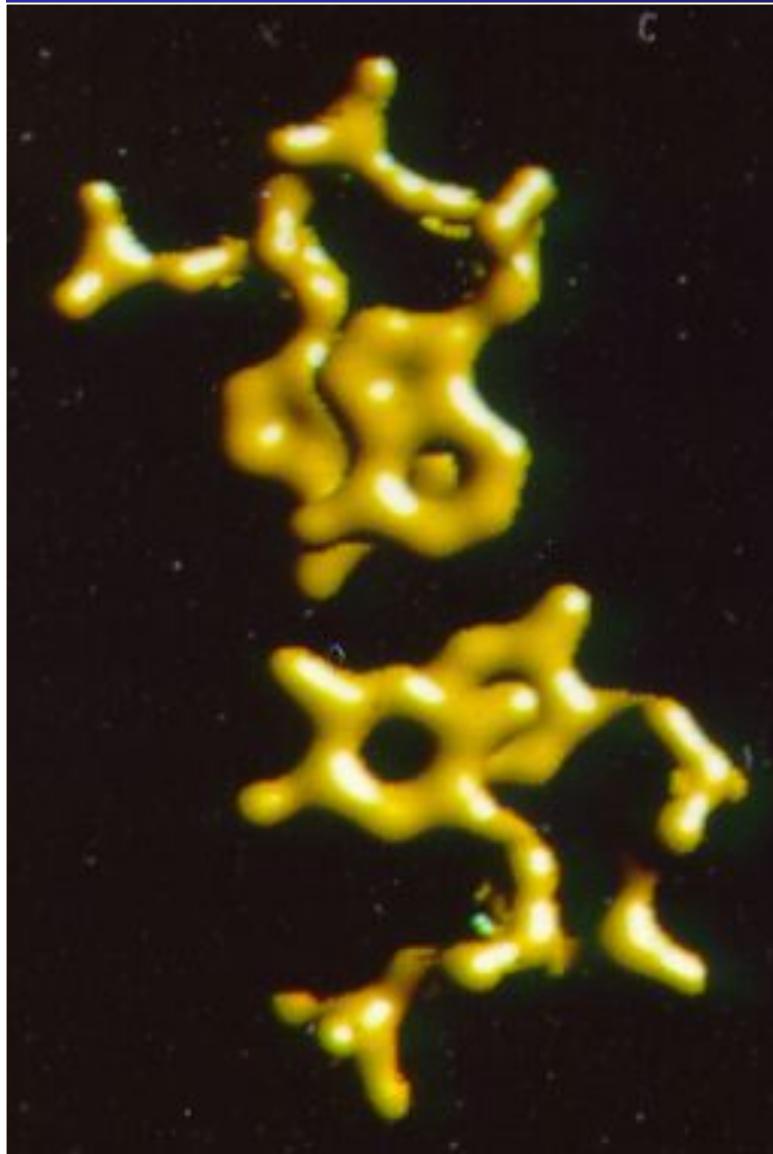


Merged iso-surface

Potentials can be Positive or Negative, and of
Varying Strengths; Note Level Sets!

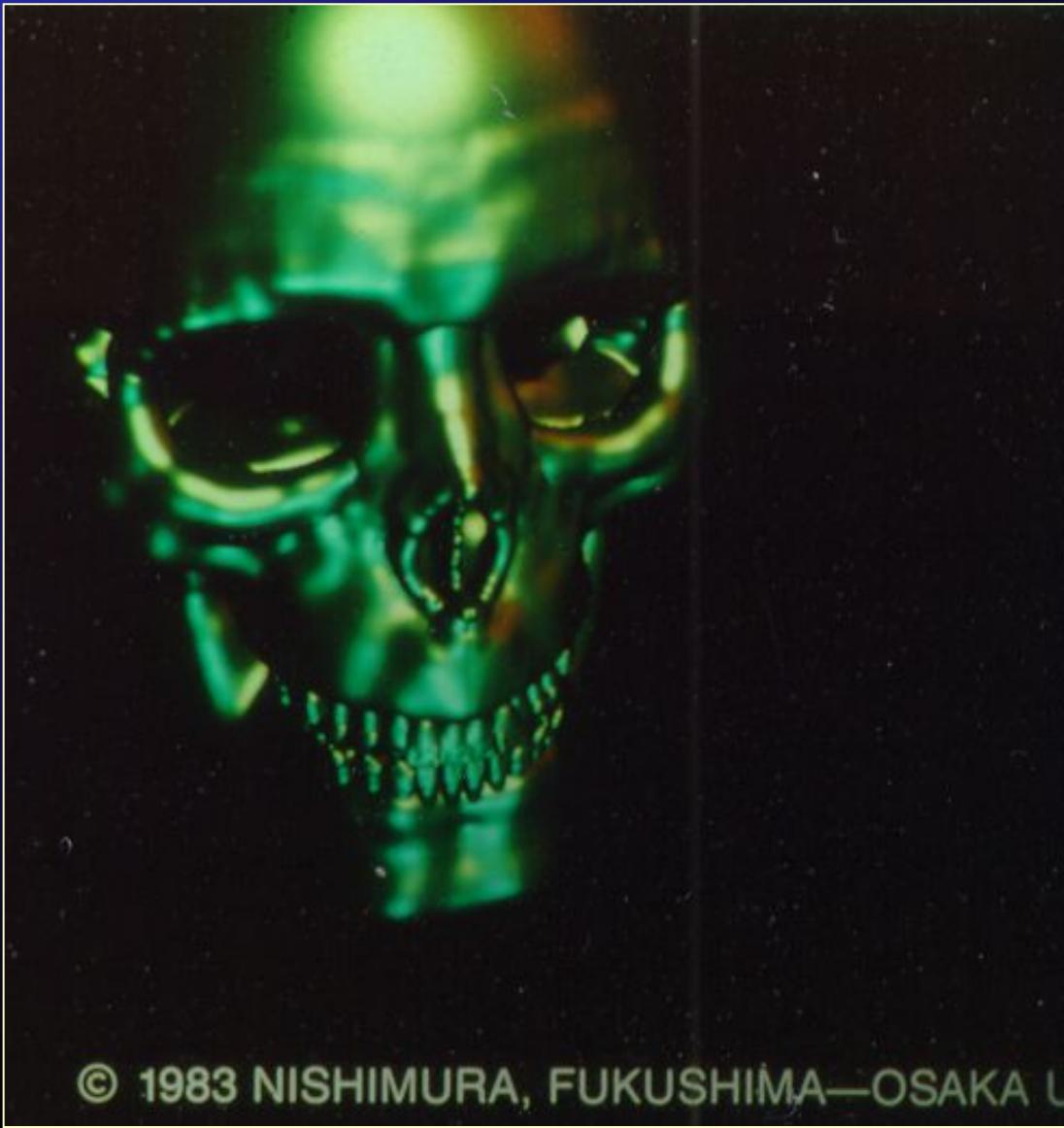


Now These Look More Like Molecules!



BLINN, J.—J.P.L.

Metaballs do not Blend Outside a Given Radius:
Easier to Control Adjacent Shapes



© 1983 NISHIMURA, FUKUSHIMA—OSAKA U

Inspiring a Life's Work



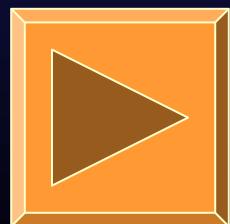
© 1984 YOICHIRO KAWAGUCHI
NIPPON ELECTRONICS COLLEGE, TOKYO, JAPAN

Combinations and Hybrids

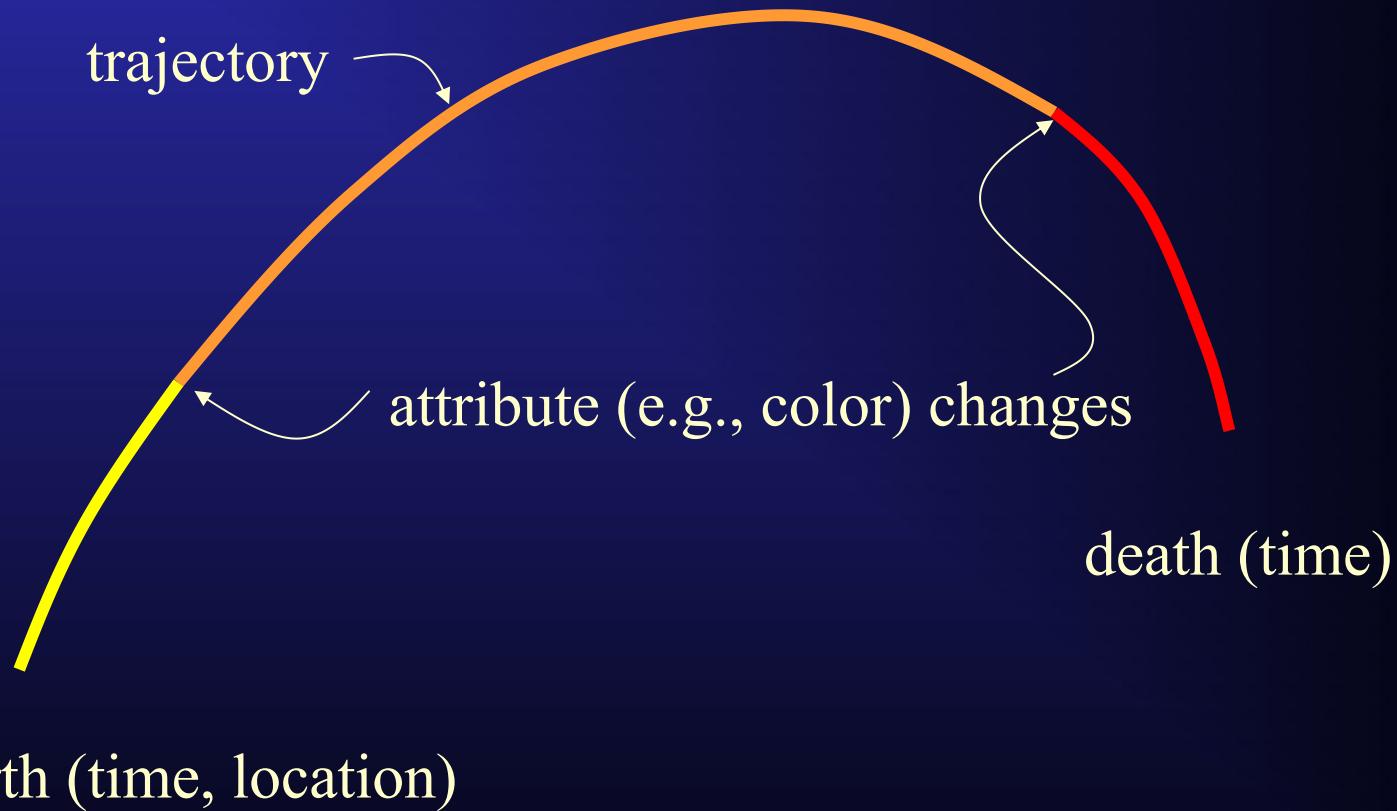
- Particle systems
- Point set (Meshless) Geometry
- Embeddings
- Procedural deformation
- Free-Form Deformation (FFD)
- Model blending
- Multi-resolution surfaces

Particle Systems

- Sets of (many) points moving in spatial paths.
- Each has a history.
- Usually controlled by probabilistic algorithms.
- Define a volume implicitly by (statistical) distribution.
- The points may be used to position other geometric objects, e.g., spheres, cattle, birds, etc.

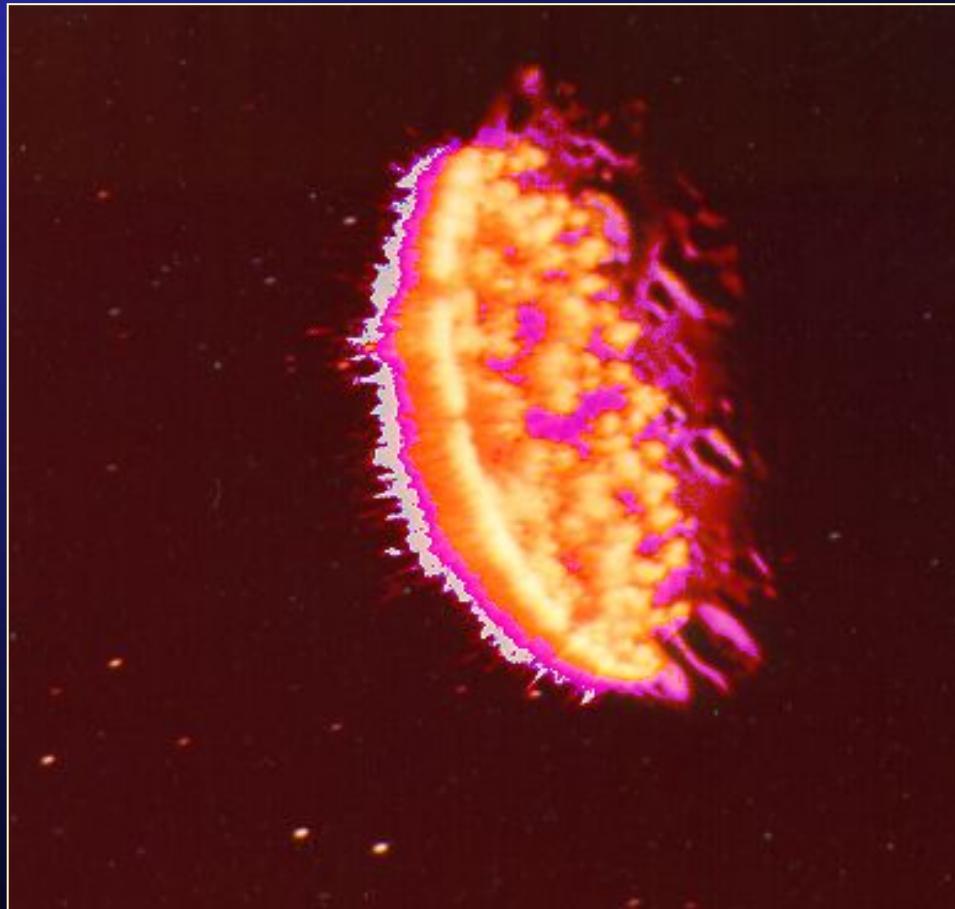


Particle Example



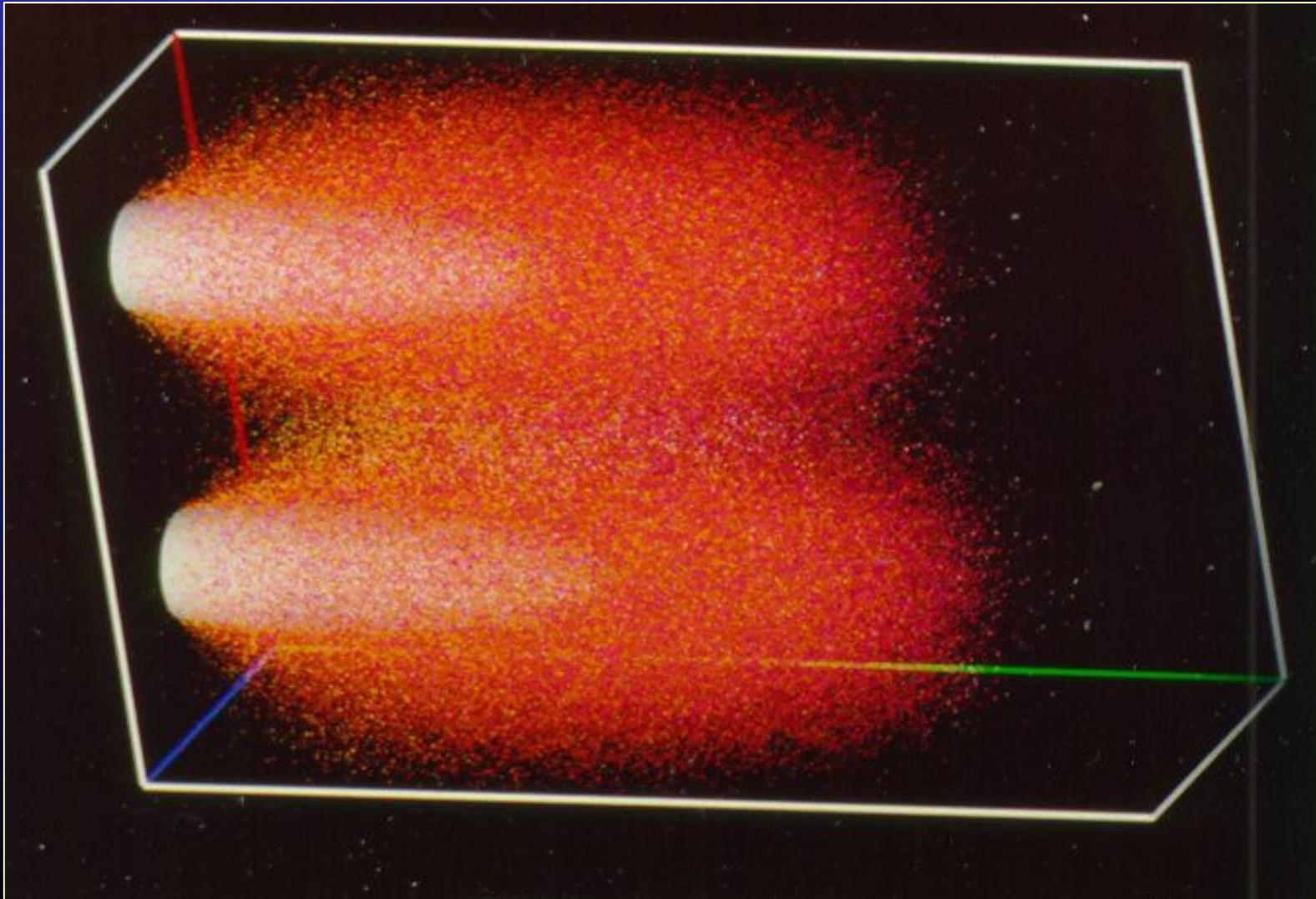
This generally happens over several frames, creating animation.

Genesis II Fire Sequence



STAR TREK® II: The Wrath of Khan
© 1982 Paramount Pictures
All rts. res.

Particles as Volumes



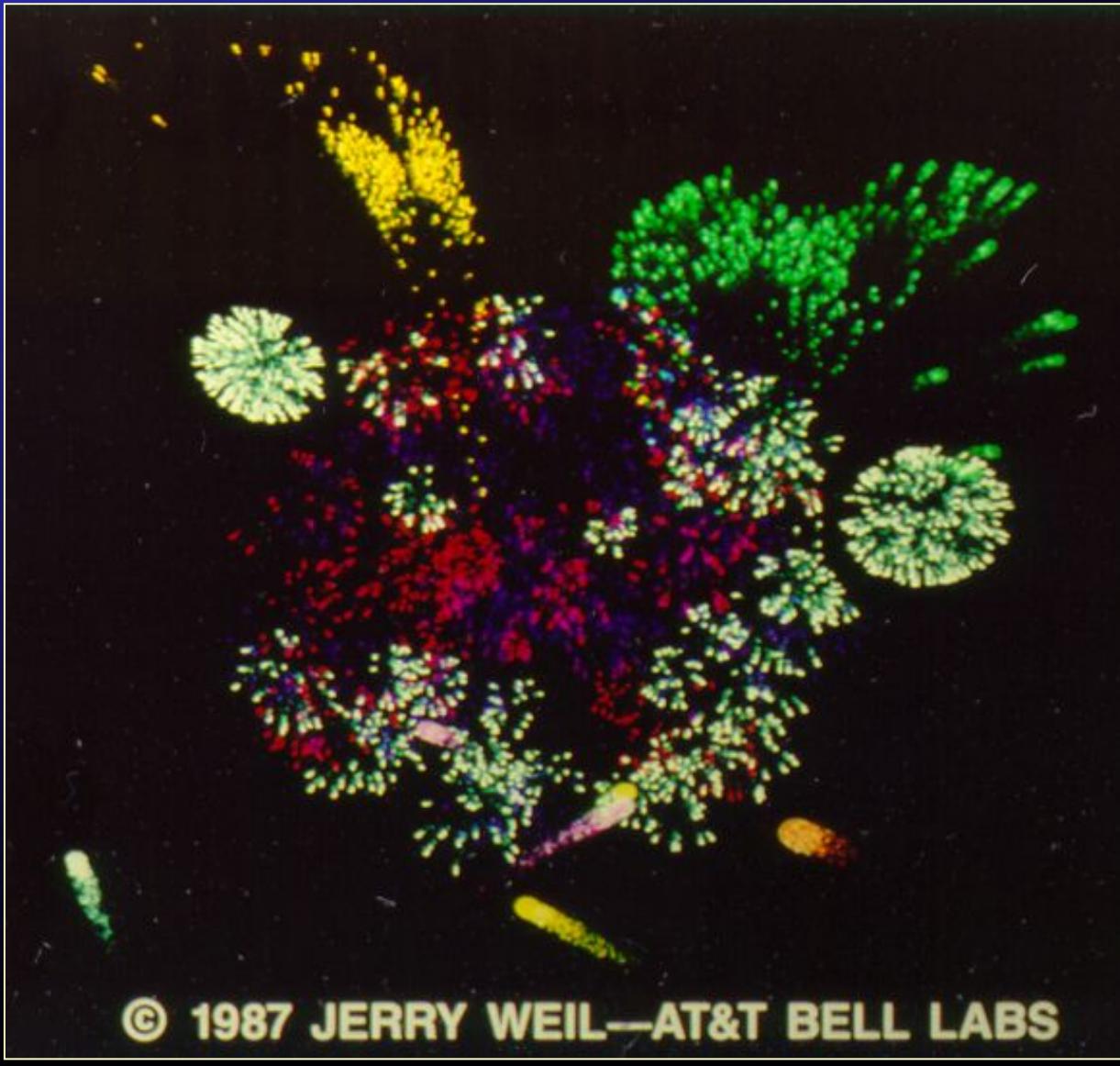
©1994, ICASE/USRA

Particle Fountains



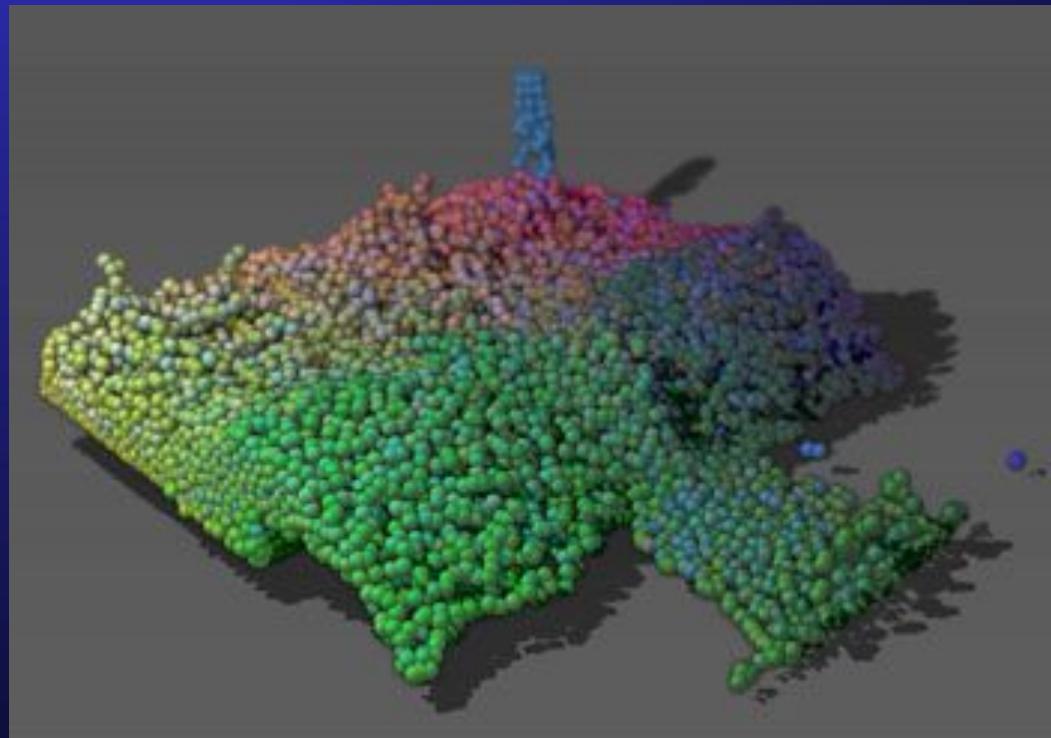
©1994, Miller/Greene, Apple Computer

Safe Fireworks



© 1987 JERRY WEIL—AT&T BELL LABS

Particles with Mass are the Basis for Lagrangian Fluid Simulations



<http://www.rchoetzlein.com/eng/graphics/fluids.htm>



<http://disney.go.com/wheresmywater/index.html#about>

Point Set (Meshless) Geometry

- Particle systems can give the appearance of solid form.
- Voxels led to new rendering.
- Voxels are rigidly structured, so...
- Represent objects by points without any explicit ordering or mesh (edge) structure: **Point Set Geometry**.
- Points that have normal information as well are called **surfels** (surface elements).
- Approximate surface normals by computing local gradients (extension of method to compute voxel gradients)



Point Set and Surfel Examples



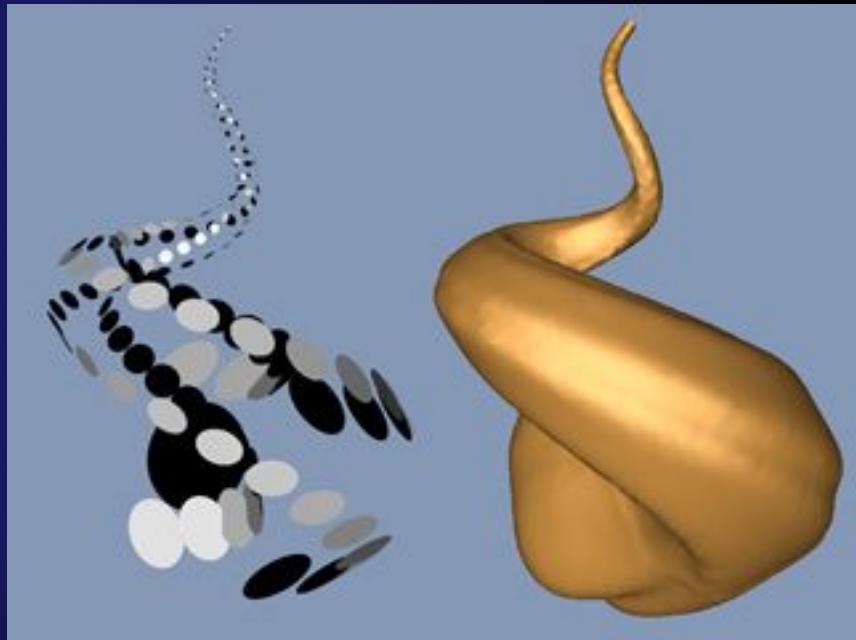
(a)



(b)



(c)



Surfels and resulting surface.
(Amenta and Kil)

Point set (a); resampled (b); rendered (c). (Alexa et al. 2004)

Outline

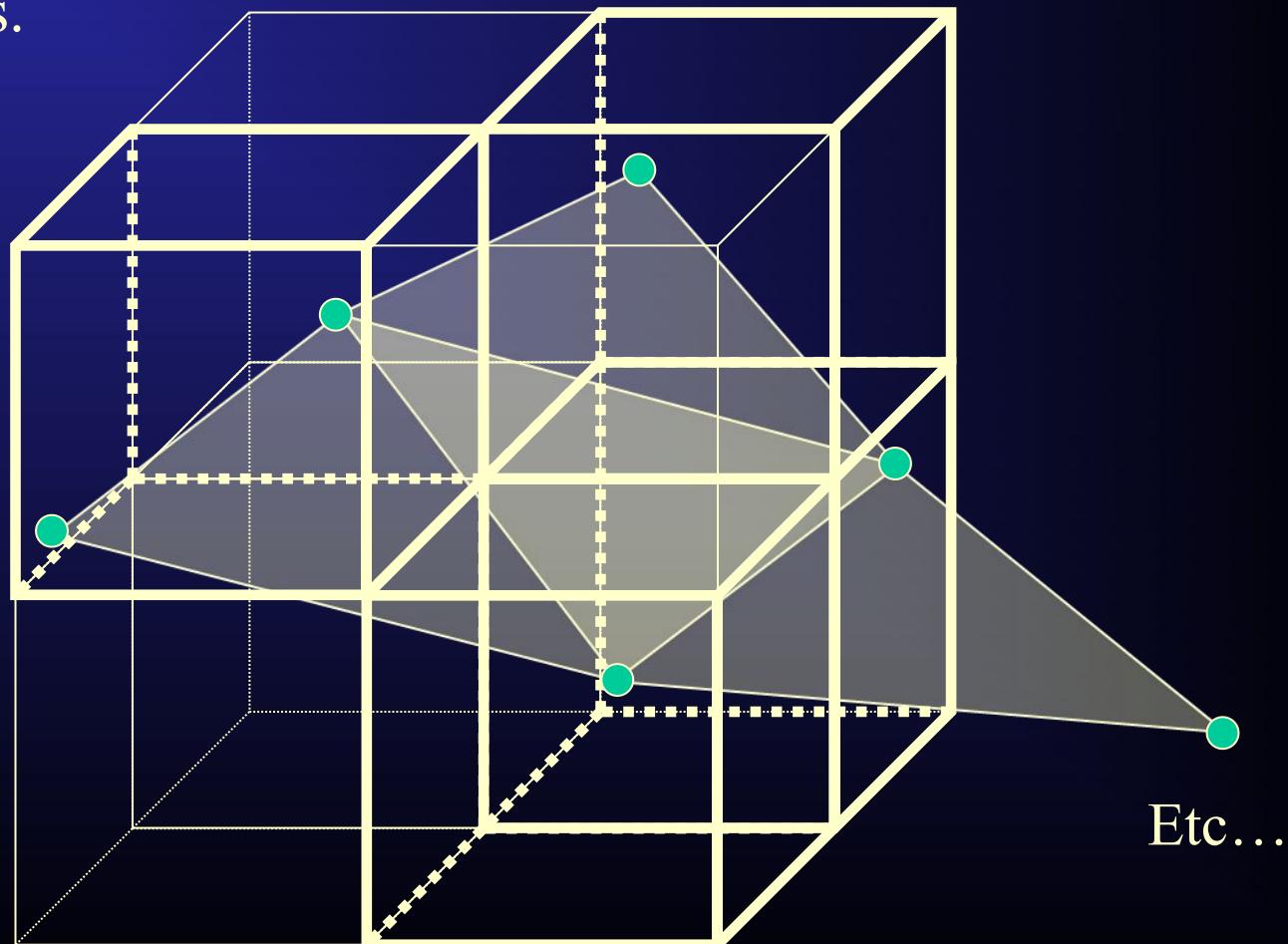
- Great Ideas ✓
- Virtual Environments & Devices ✓
- 3D Perception ✓
- Architecture, Color & Rasterization ✓
- Vector Geometry & Transformations ✓
- 3D Modeling ✓
- Embedding, Hierarchies & Contours
- Model Generation & Deformation
- Visible Surface Algorithms
- Polygon Algorithms
- Image Synthesis & Shading
- Mapping
- 3D Viewing Transformations
- Anti-aliasing & Compositing
- Global Illumination
- Light Fields & HDR
- Path Tracing/Photon Mapping
- Futures

Embeddings and Hierarchies

- An **embedding** or a **Bounding Volume Hierarchy (BVH)** is typically used to increase efficiency of search cost or access time in a complex model. Examples:
 - A polygonal surface may be embedded into a voxel space.
 - A polygonal surface may be embedded into a tetrahedral tessellation, because this embedding is probably a tighter fit to the actual surface and the tetrahedra can vary in size.
 - A point set may be embedded into a polygonal mesh or implicit model.
 - These embeddings add efficiency to, e.g., *ray-surface intersection*, *collision detection* or *geometric neighbor queries*.

Example of Embedding

- A polygonal mesh is embedded in a voxel space: each mesh vertex is linked to its containing voxel, and each voxel lists its contained mesh vertices.

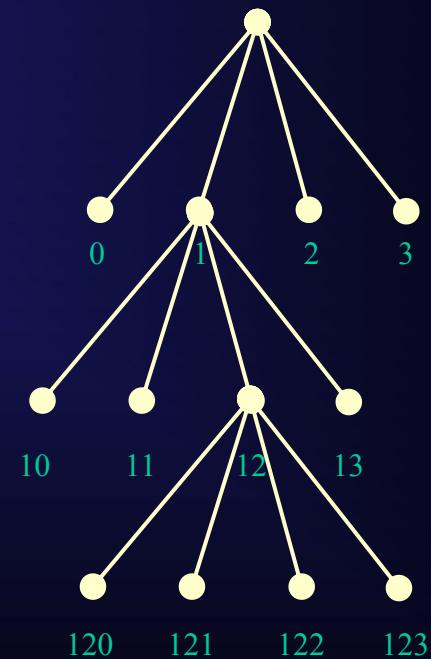
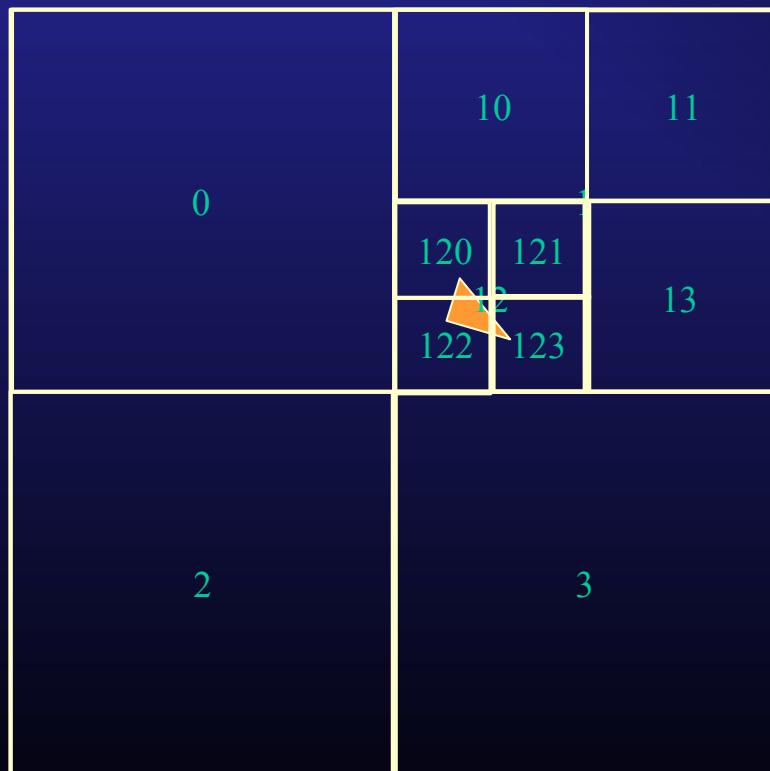


Spatial Data Structures

- Quad trees
- Oct-trees
- K-d trees
- Bounding volume hierarchy
- Binary Space Partitioning trees (later)

Quad-Trees (2D)

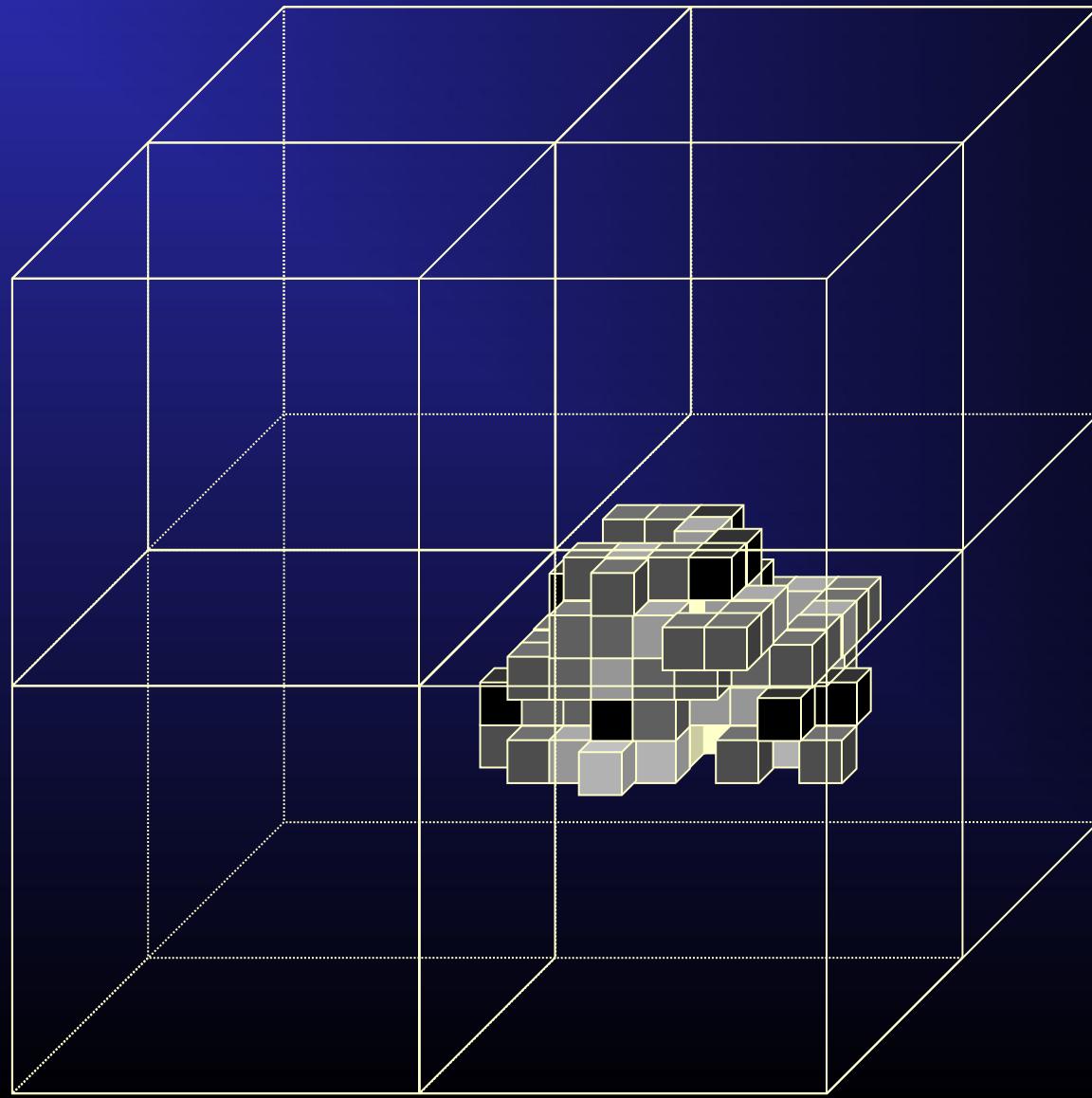
- If a cell (quadrant) is completely full or completely empty, done.
- Else split quadrant into 4 quadrants and recurse on each.
- Quads have 4 children or none.



Oct-Trees (3D)

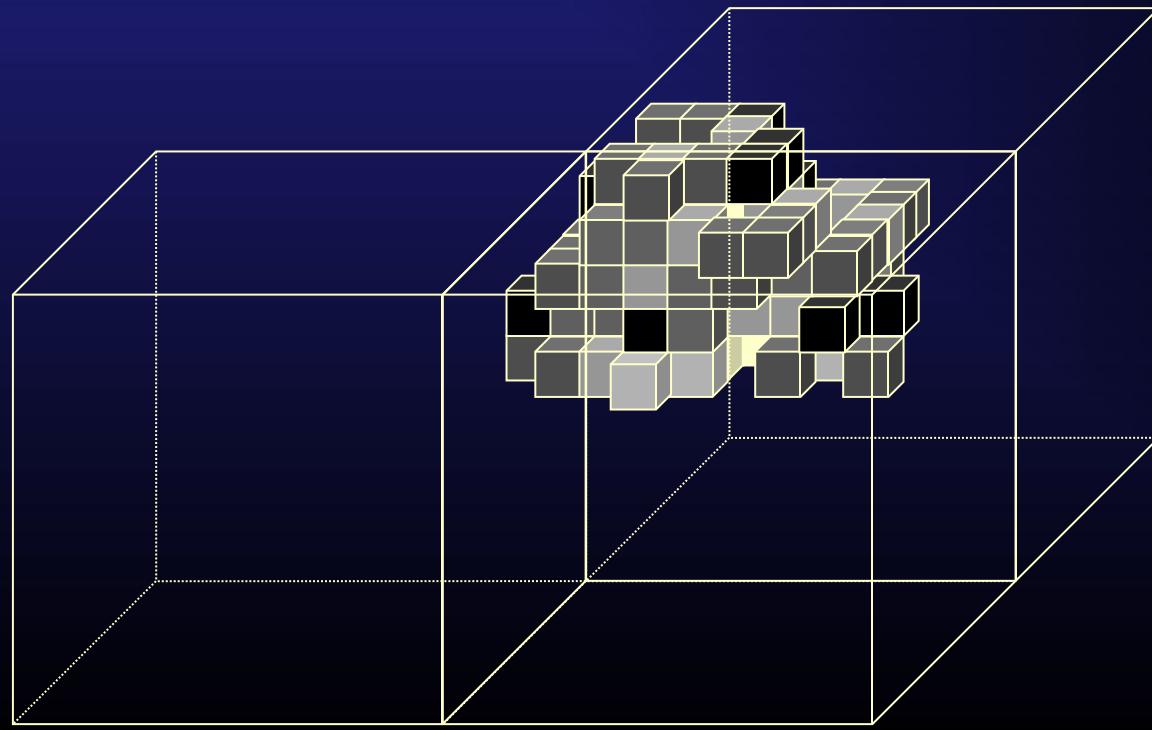
- Same idea as quad-trees but in 3D
- Partition space into 8 cubical octants, recursively: is content simple? if not, subdivide.
- Increase space and search efficiency of spatial subdivisions.
- Similar encoding: 0-7 (2^3 combinations) for each new finer level.

Oct-Tree Construction



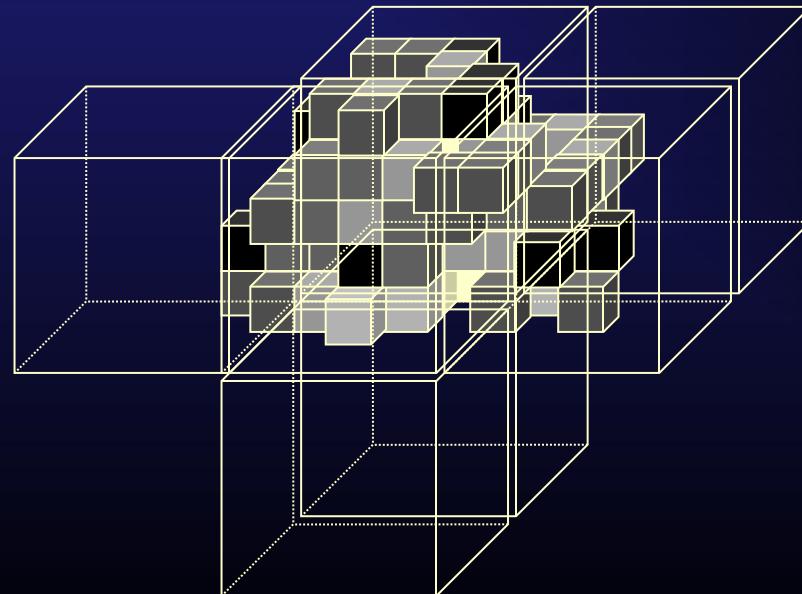
Oct-Tree Construction

Only these 3 sub-octants
have material

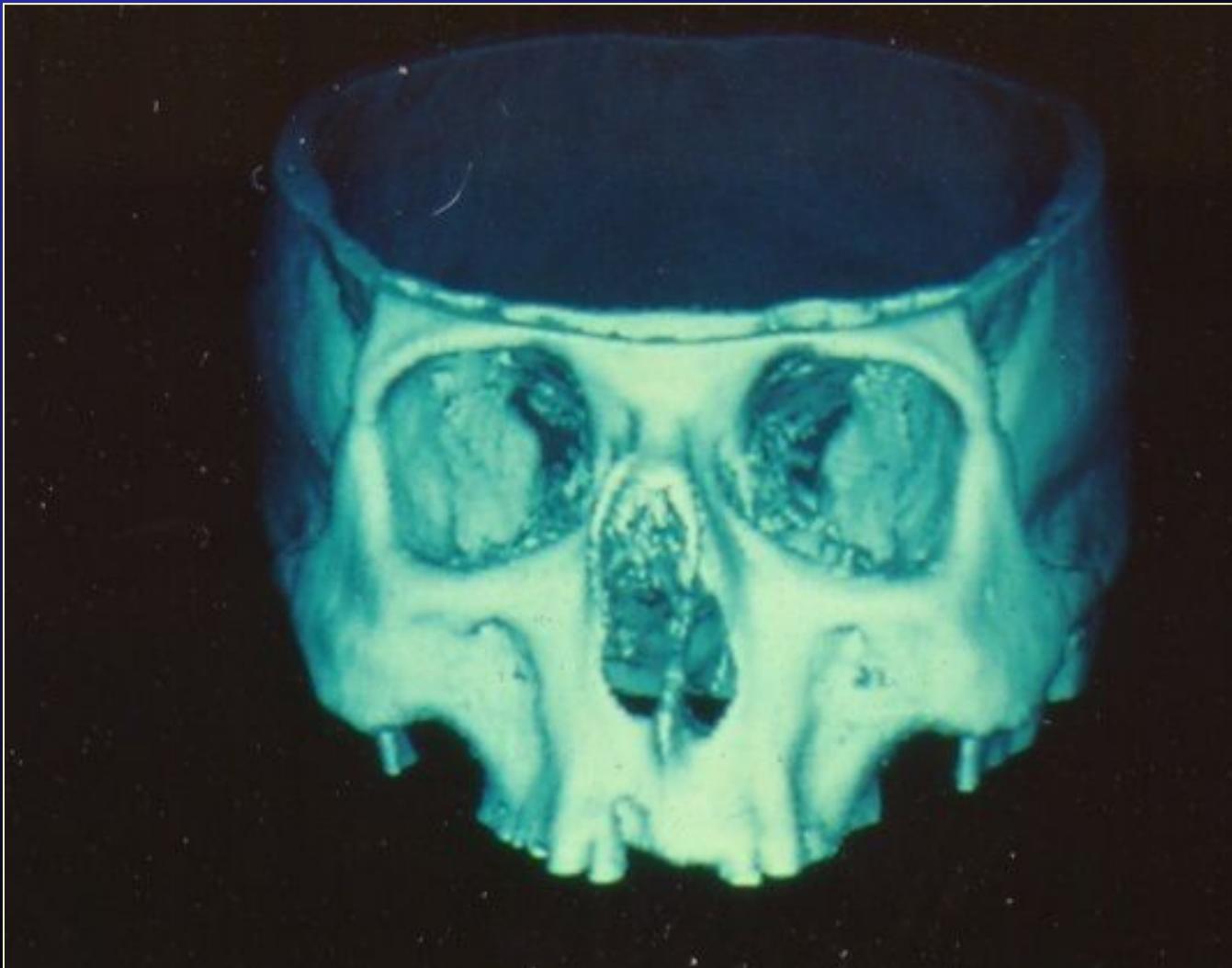


Oct-Tree Construction

Only these 7 sub-sub-octants
have material, and so on ...



Oct-Tree Skull from Voxel Data



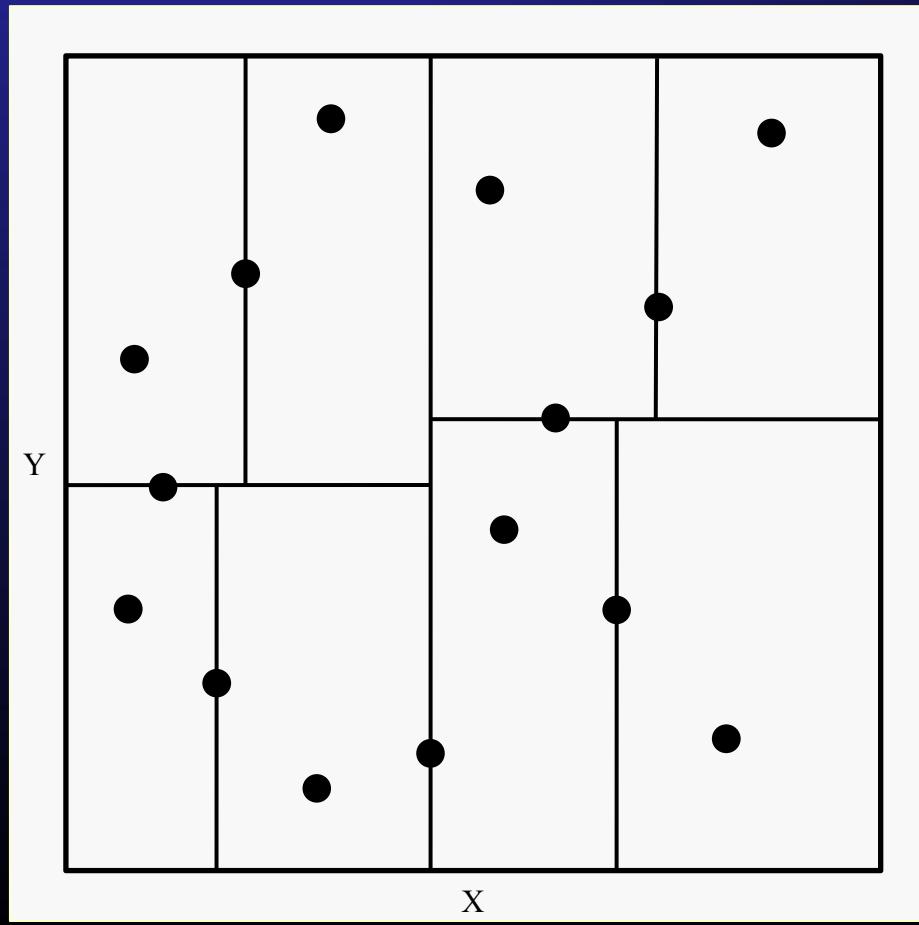
© 1985 W.E. SCHMIDT — PHOENIX DATA SYSTEMS,

K-dimensional (K-d) Trees

- Quad- and oct-trees can be very unbalanced: they do not try to divide space so that roughly similar numbers of points appear in each branch of the subdivision.
- Solution: Divide one dimension “in half” at each level, making the “halfway” point the *median* of the data in the partition cell so far. Cycle through the K spatial dimensions one at a time (e.g., X then Y then Z then X, etc.), dividing at the median in each dimension in turn, repeating until no more divisions are possible.
- For a finite point set this must terminate.
- Divisions are always axis-aligned.
- Best to create a list of sorted points for each dimension in advance to make median finding simple.
- Search for points in a rectangular window in $O(\sqrt{n} + k)$
- $O(n)$ storage and $O(n \log n)$ construction time.

2-d Simple Example

- Each partition passes through the median in that dimension.
- The cut is at the \geq node.
- X, Y, X, Y... (showing the depth first construction)



Build K-d Tree Algorithm (K=2)

P a set of points, $depth$ the current depth in the tree

BuildKdTree($P, depth$)

if P contains only one point

then return a leaf storing the point

else if $depth$ is even

then

Split P into two subsets with a vertical line l through the median x-coordinate of the points in P . Let A be the set of points to the left of l , and B be the set of points to the right of l or on l .

else

Split P into two subsets with a horizontal line l through the median y-coordinate of the points in P . Let A be the set of points below l , and B be the set of points above l or on l .

Create a node v storing l with children $left$ and $right$

$left \leftarrow \text{BuildKdTree}(A, depth + 1)$

$right \leftarrow \text{BuildKdTree}(B, depth + 1)$

return v

Query K-d Tree

v a node, R a range (an axis-aligned rectangle or other shape, e.g., a disk)

SearchKdTree(v, R)

if v is a leaf

 then Report the point stored at v if it lies in R

else

 if region($v->left$) is fully contained in R

 then ReportSubtree($v->left$)

 else

 if region($v->left$) intersects R

 then **SearchKdTree($v->left, R$)**

 if region($v->right$) is fully contained in R

 then ReportSubtree($v->right$)

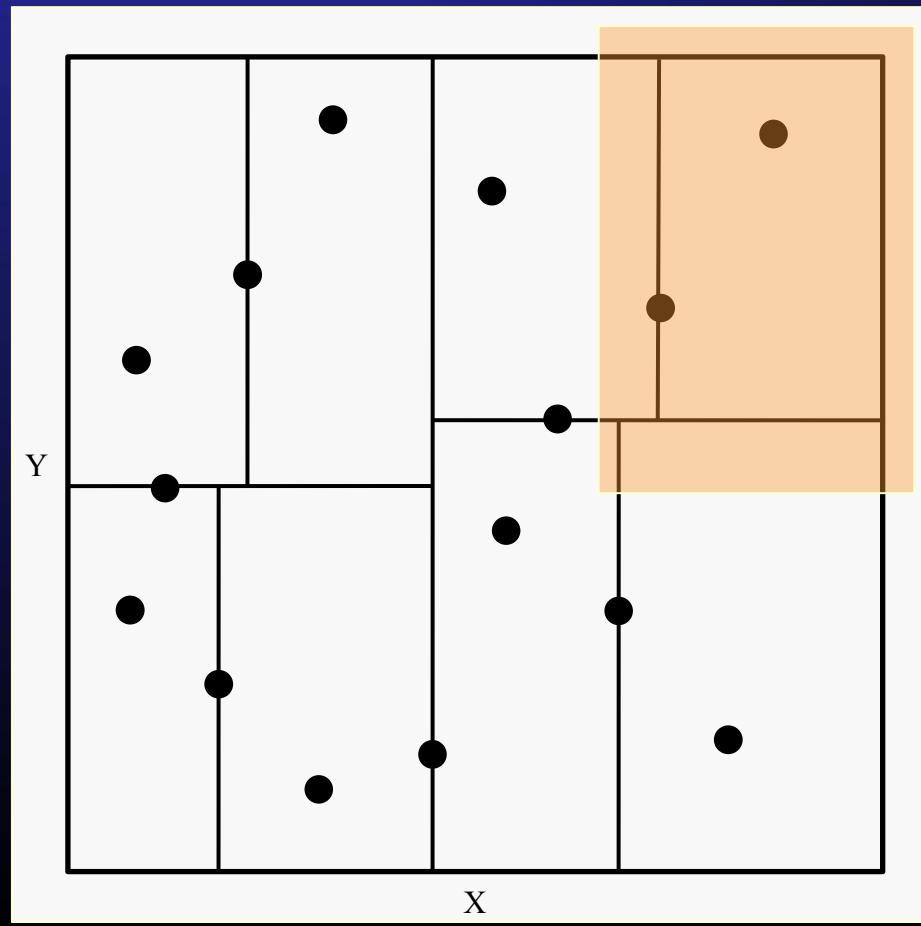
 else

 if region($v->right$) intersects R

 then **SearchKdTree($v->right, R$)**

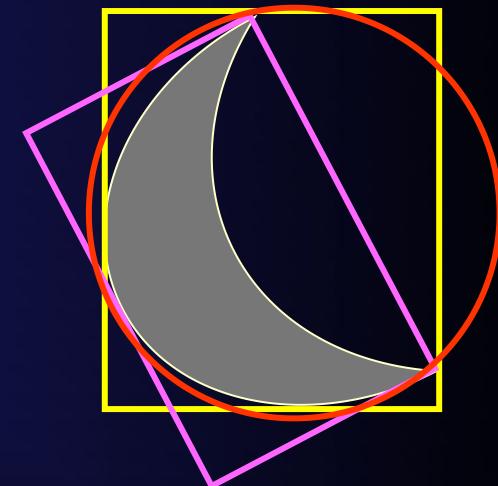
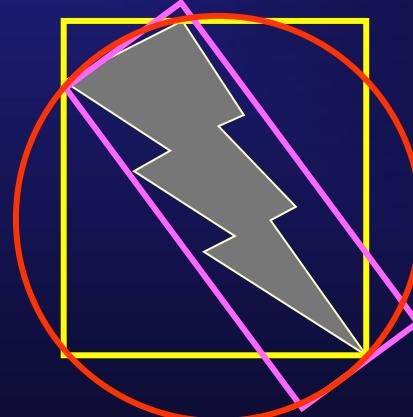
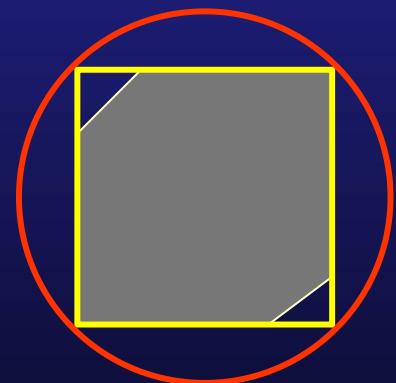
2-d Query

- Finding the set of points within a region.
- If a node is completely contained in a region, then all of its children are.



Bounding Boxes/Volumes

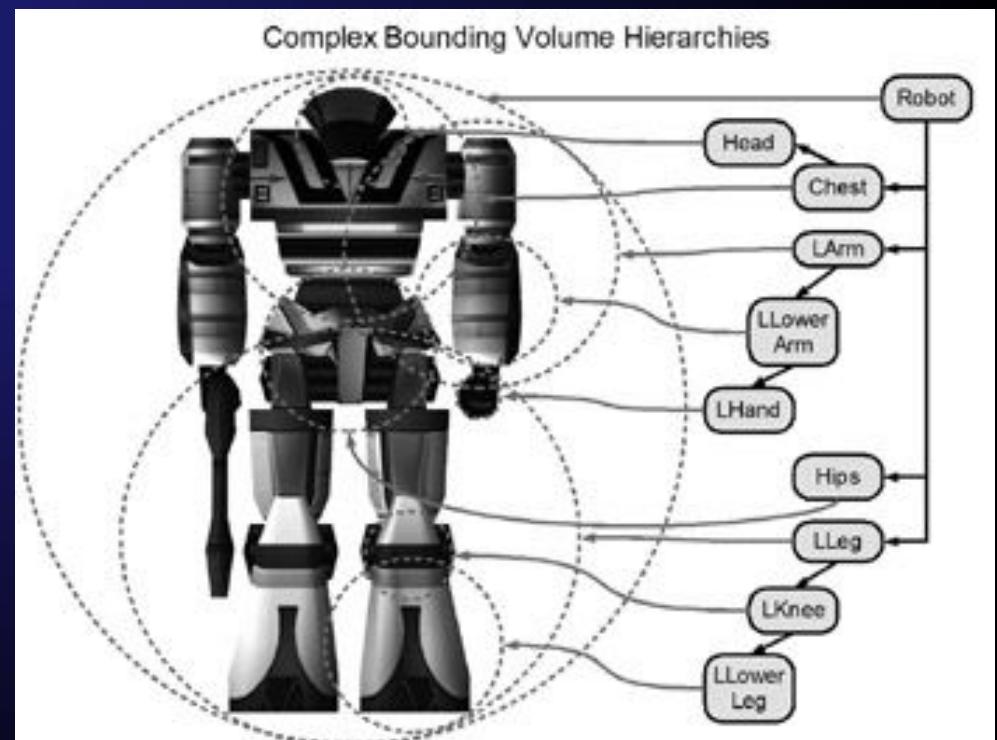
- Spheres
- Axis aligned bounding boxes (AABB)
- Object aligned bounding boxes (OABB)



- In general, OABB fit tighter than AABB, which fit tighter than spheres.
- But AABB is the easiest to compute: just *max* and *min* each coordinate.

Bounding Volume Hierarchy (BVH)

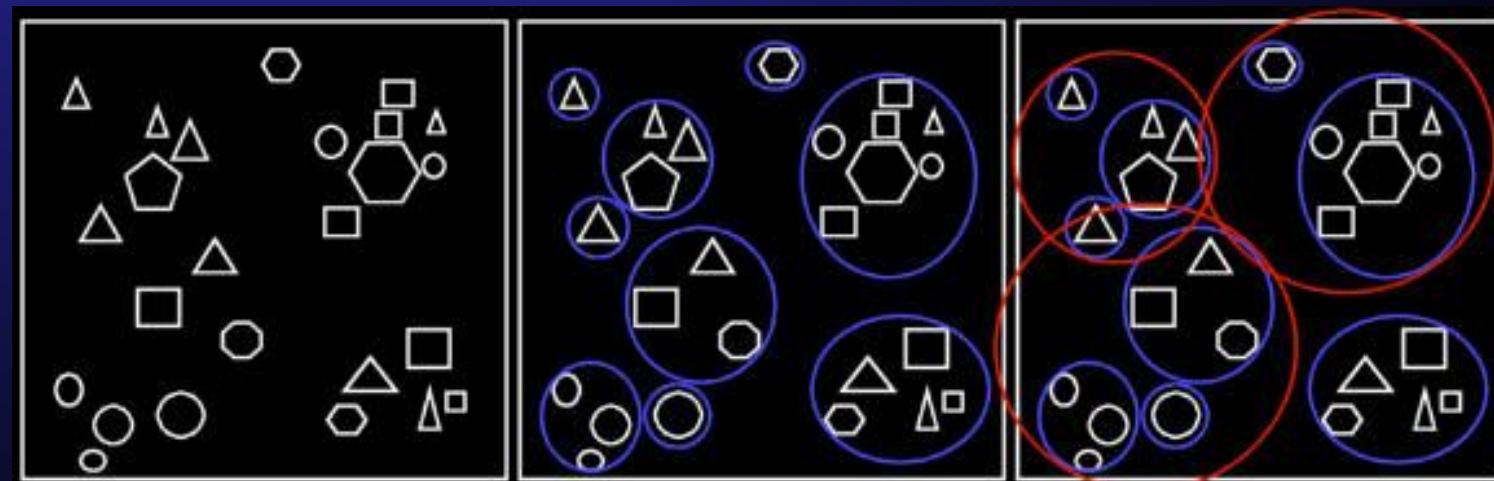
- Spatial sets are not required to be a partition, i.e., they can overlap.
- Can use various volume sets:
 - Spheres (e.g. →)
 - Axis-aligned bounding volumes
 - ...



<http://flylib.com/books/en/2.124.1.133/1/>

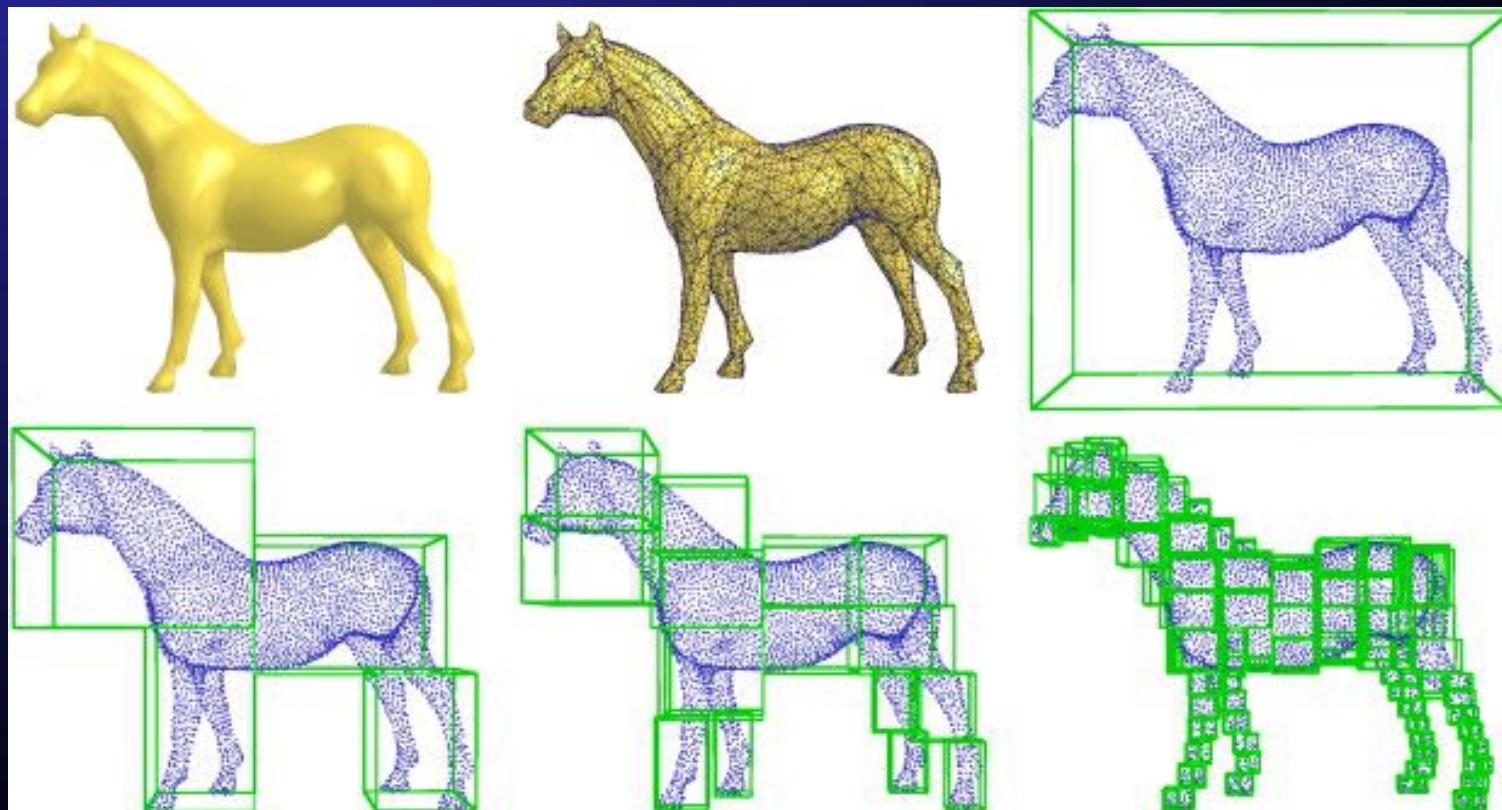
Bounding Volume Hierarchy

- Group objects any reasonable way: by complex areas, by natural parts, by proximity, by scene graph node, by spatial density, etc.



Benefit: Search efficiency for membership, ray-intersection or collision queries

- Advantage: Simple reject of complex geometry within a volume if the volume itself is not intersected.



<http://www.sciencedirect.com/science/article/pii/S0921889011001515>

Computing Surfaces

- Level Sets
- Contouring
- Marching Cubes

How to Make a Surface from Point Sets or Surfels?

- One way is to use point set to generate a convex polygon mesh by the **Voronoi algorithm**. (Convex polygons are of course easy to triangulate.)
- Demo: 
- Need to be mindful of surface topology (since its not obvious what points ought to be neighbors of each other – hence on the surface).
- Let's look at the general issue of creating a polygonal mesh surface from volumetric or implicit function data.

Polygonization of Implicit Surfaces

- We can use marching cubes to convert an implicit surface to a polygonal mesh.
- Can display implicit surface using polygons and thus use fast polygon display methods.
- Polygonization is a special case of finding and displaying a level set, contour line or contour surface in a 2D or 3D dataset: this is a fundamental operation in fluid simulation and in scientific visualization of volume datasets.
- Major steps
 - Partition space into 2D or 3D cells
 - Fit a line or polygon to surface in each cell

Slides adapted from John C. Hart

Contouring or Level Set Applications

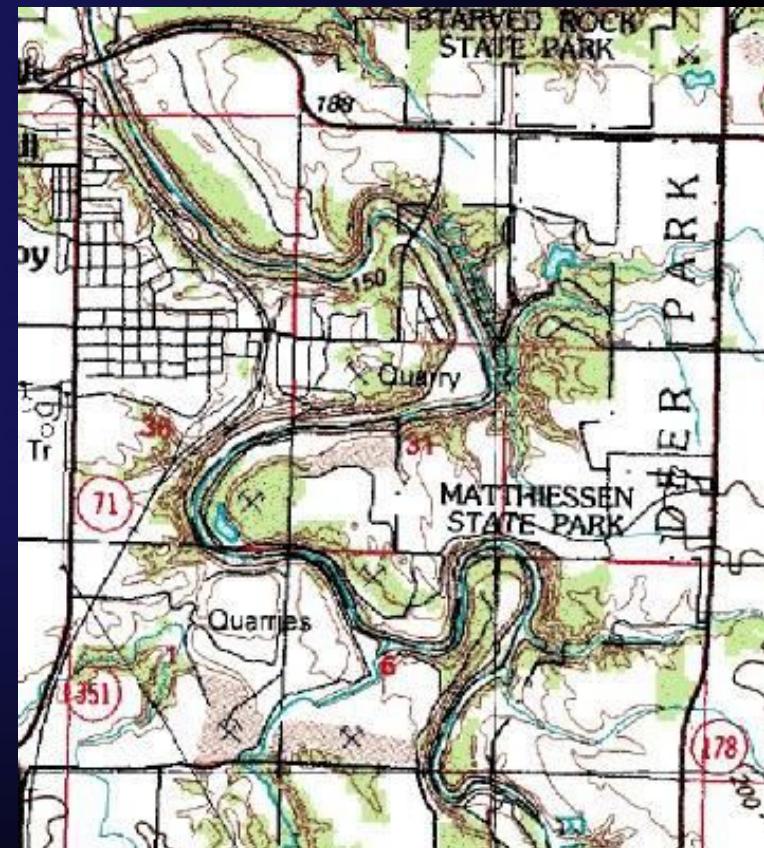
- Conversion of a scalar field into iso-valued level sets
- Applications
 - Elevation contours from topography



US Geological Survey

Contouring or Level Set Applications

- Conversion of a scalar field into iso-valued level sets
- Applications
 - Elevation contours from topography



US Geological Survey

Contouring or Level Set Applications

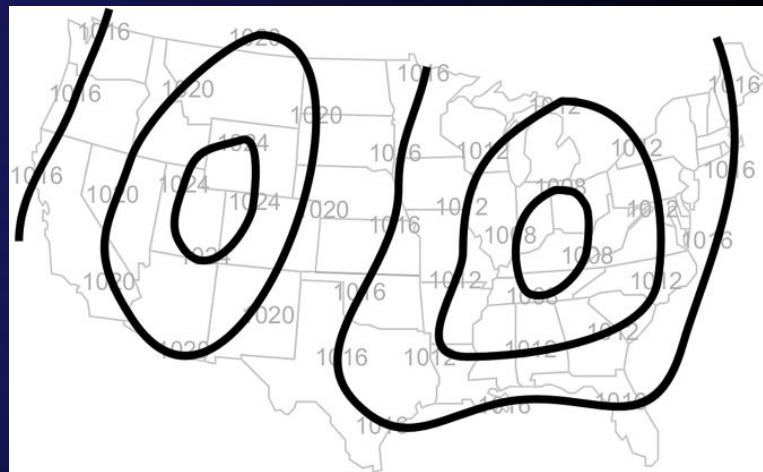
- Conversion of a scalar field into iso-valued level sets
- Applications
 - Elevation contours from topography
 - Pressure contours from meteorology



NOAA JetStream Weather School

Contouring or Level Set Applications

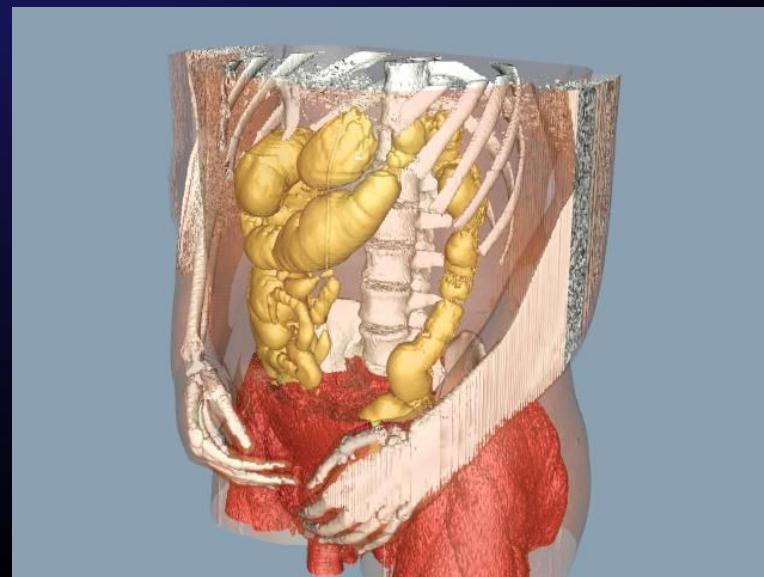
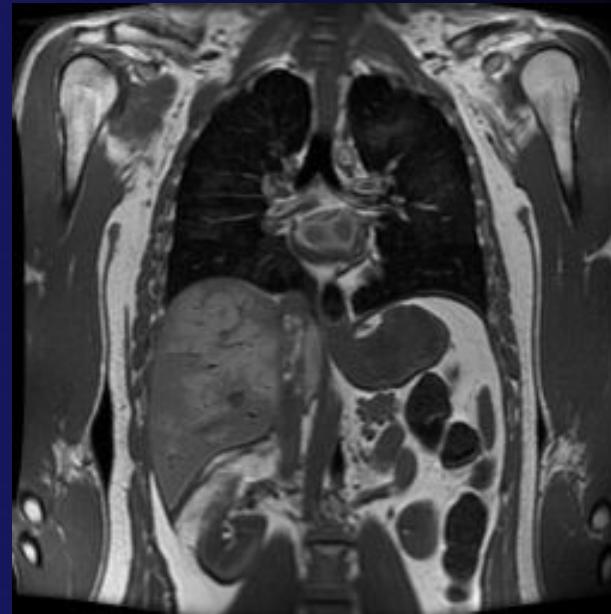
- Conversion of a scalar field into iso-valued level sets
- Applications
 - Elevation contours from topography
 - Pressure contours from meteorology



NOAA JetStream Weather School

Contouring or Level Set Applications

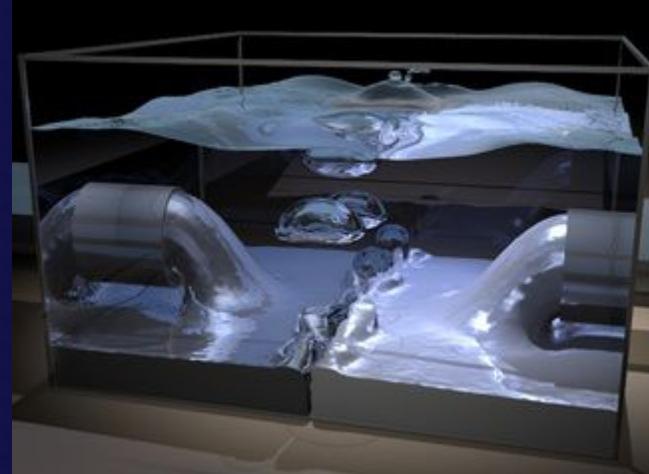
- Conversion of a scalar field into iso-valued level sets
- Applications
 - Elevation contours from topography
 - Pressure contours from meteorology
 - Tissue surfaces from tomography



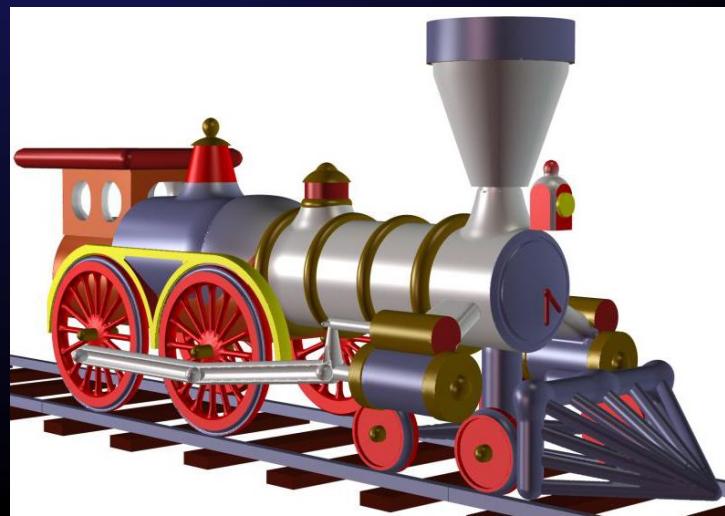
Lorensen, Marching Through the Visible Man

Contouring or Level Set Applications

- Conversion of a scalar field into iso-valued level sets
- Applications
 - Elevation contours from topography
 - Pressure contours from meteorology
 - Tissue surfaces from tomography
 - Implicit surfaces from math, CAD, or fluids



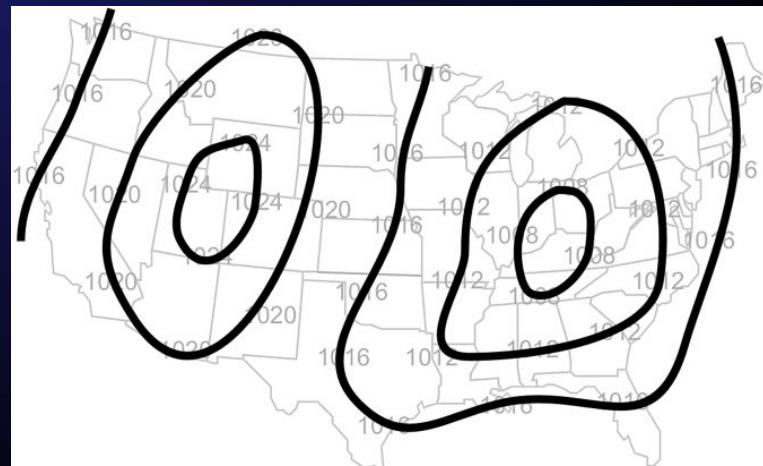
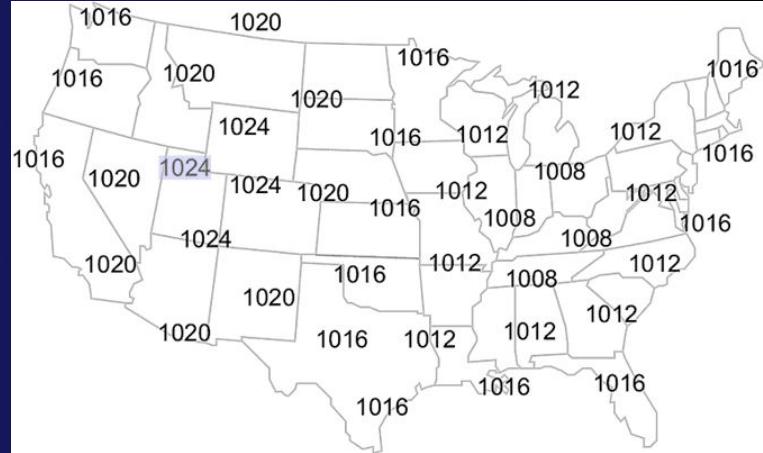
Multiple Interacting Liquids:
Fedkiw, Losasso, Shinar and Selle



Brian Wyvill's implicit train

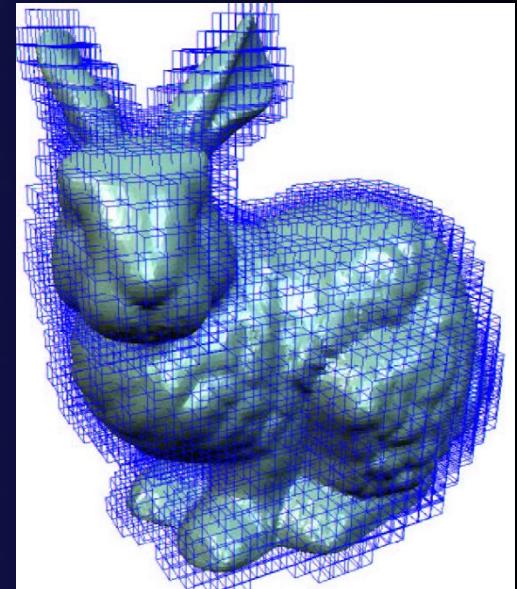
Contouring

- Two step process
 1. Explore space to find points near contour
 2. Connect points into contour (or surface)



Spatial Partitioning

- Divide space into lattice of cells
 - Cubes
 - Tetrahedra (simplicial)
 - Adaptive (smaller near detail)
 - Hierarchical (cells within cells)
- Three techniques
 - Subdivision (oct-tree, K-d tree)
 - Enumeration (all cells in space)
 - Continuation (surface tracking from seed point)

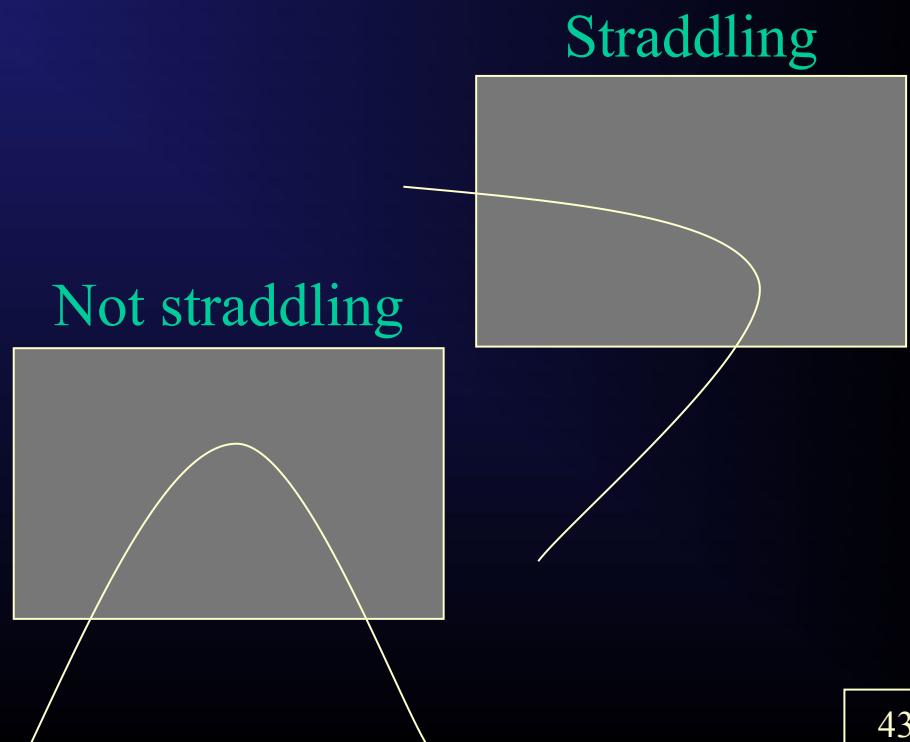
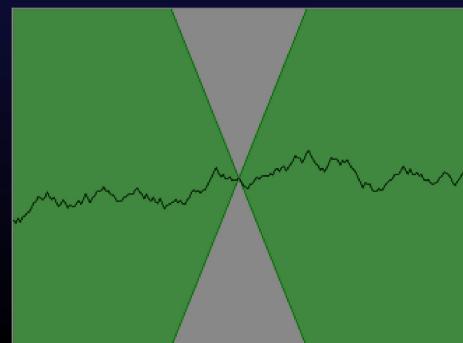


Kobbelt et al. SIGGRAPH '02

Subdivision Criteria

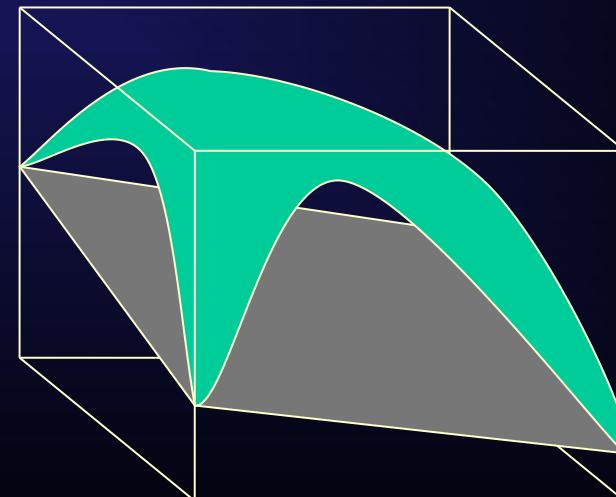
How do we know a cell contains the surface?

- Straddling Cells
 - At least one cell vertex inside and at least one cell vertex outside surface.
 - Non-straddling cells can still contain surface.
- Need guarantees
 - Interval analysis
 - Lipschitz condition:
i.e., bounded |slope|



Cell Polygonization

- Cell known to contain surface
- Need to approximate surface within cell
- Use piecewise-linear approximation (polygon)
- Often used for fluid surface modeling, but notice that this approximation can change fluid volume!



Enumeration: Use “Marching Cubes” Method

Pseudo-Code:

Read in volume in two slices at a time
(here in 2D, rows; in 3D, planes)

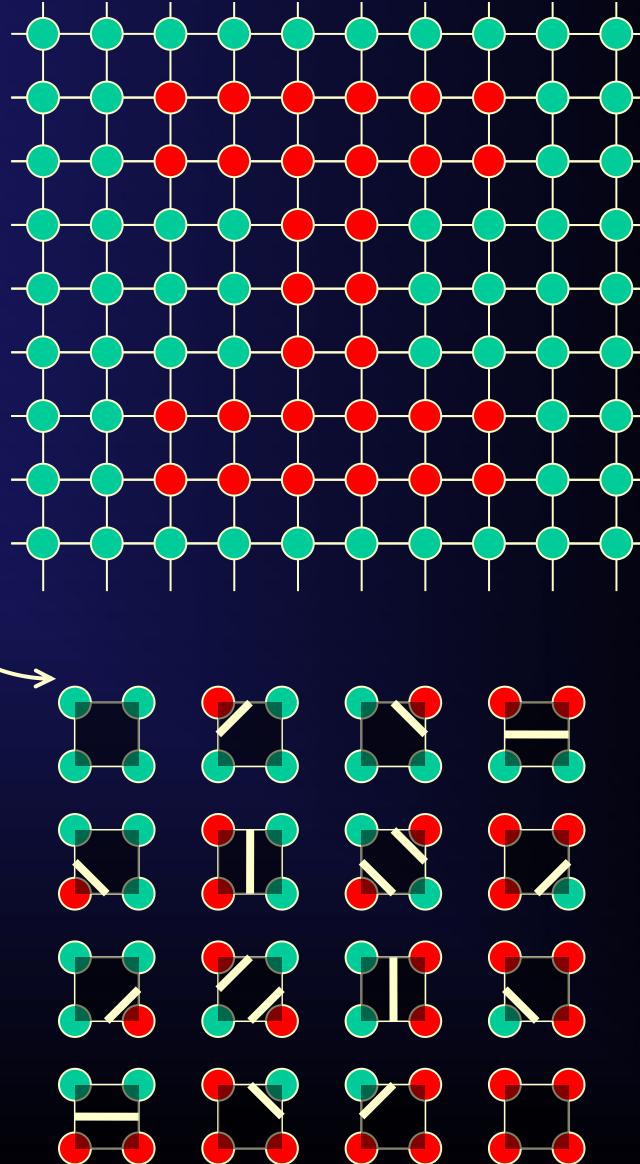
For each cubic cell in this slice

 Compute Marching Cubes (MC)
 index using bitmask of vertices

 Output polygon(s) stored at MC
 index translated to cell position

End for

Remove last slice, add new slice,
repeat



Enumeration: Use Marching Cubes

Pseudo-Code:

Read in volume in two slices at a time
(here in 2D, rows; in 3D, planes)

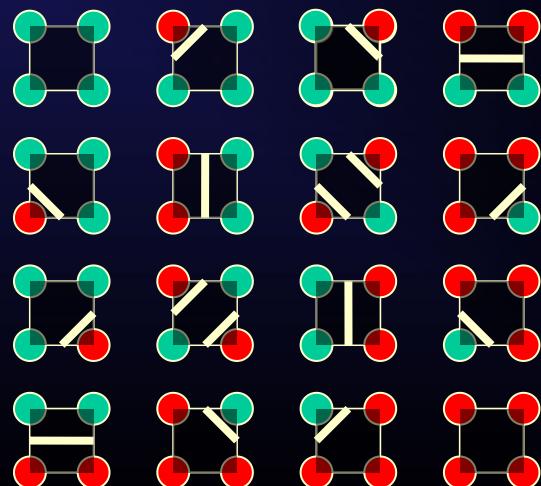
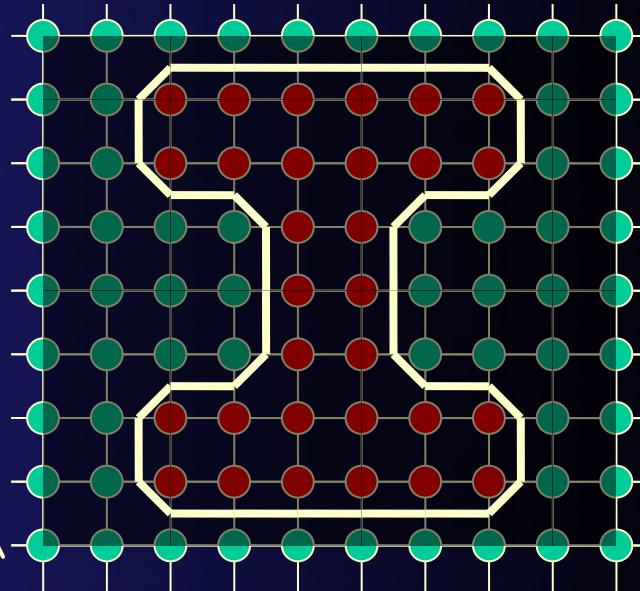
For each cubic cell in this slice

 Compute Marching Cubes (MC)
 index using bitmask of vertices

 Output polygon(s) stored at MC
 index translated to cell position

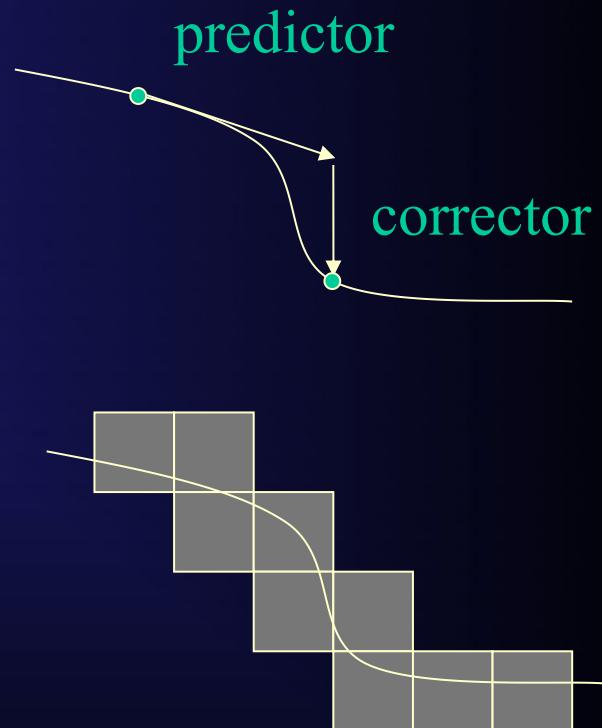
End for

Remove last slice, add new slice,
repeat



Continuation Techniques

- Predictor-corrector
 - Extrapolate in tangent.
 - Solve for surface location.
- Piecewise-linear
 - Flood-fill along cube faces,
 - I.e., check cells adjacent to current cell to find next straddling vertex set.



Continuation Algorithm

Pseudo-Code:

Find a straddling cell and push it onto stack

While stack not empty

 Pop cell

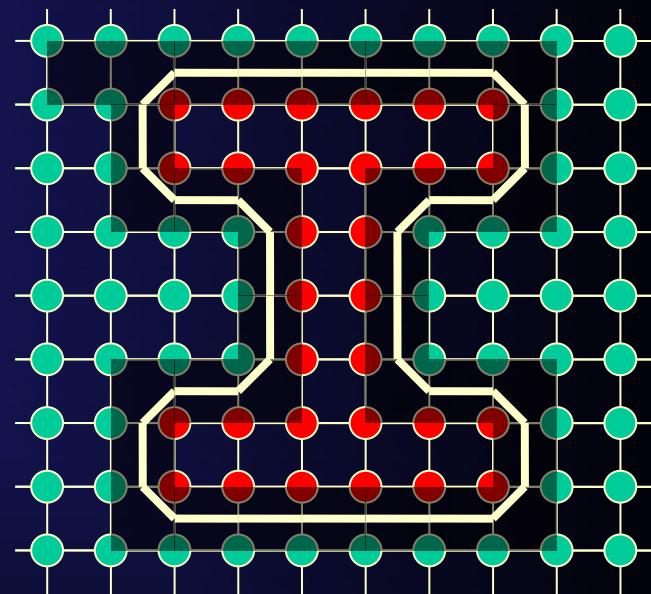
 If (!straddling) continue

 Output cell polygon

 Push unvisited neighbors onto stack

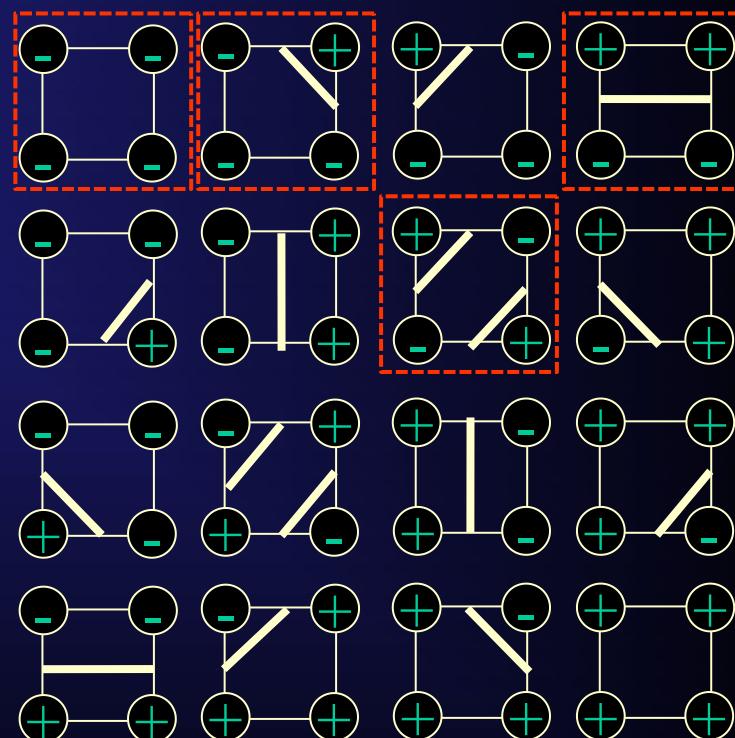
End while

- Flood-fill along cube faces
- Stack more cache coherent but queue more efficient
- Can use surface gradient to predict which neighbors will straddle



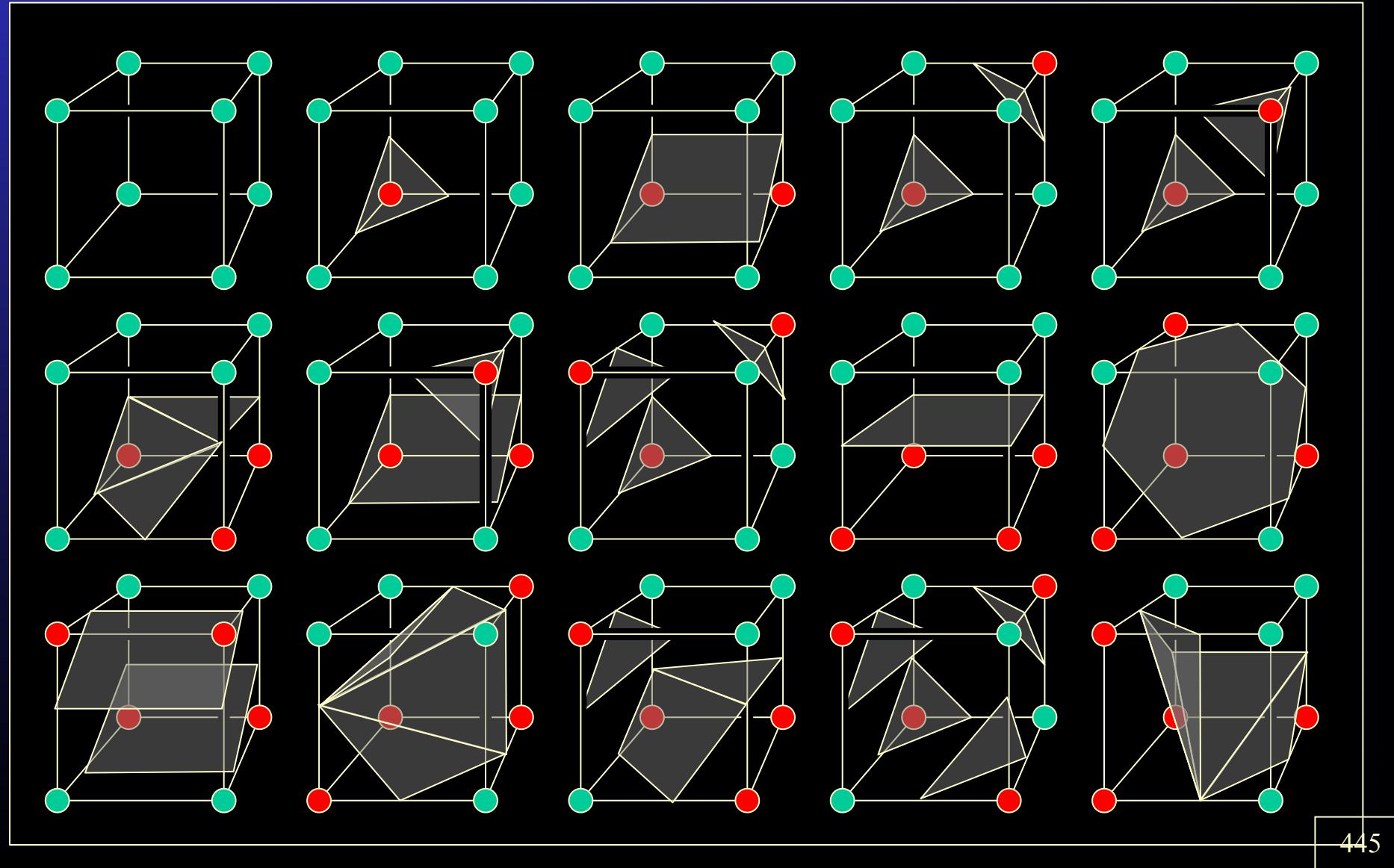
Vertex Bitmask → Polygon Shape

- Use table indexed by vertex signs
- Let + be 1, - be 0
- Table size
 - For 2D (contour line) case: 16 square cells; 4 modulo symmetry. 
 - 3D tetrahedral cells: 16 entries; 3 modulo symmetry.
 - 3D cubic cells: 256 entries; 15 modulo symmetry



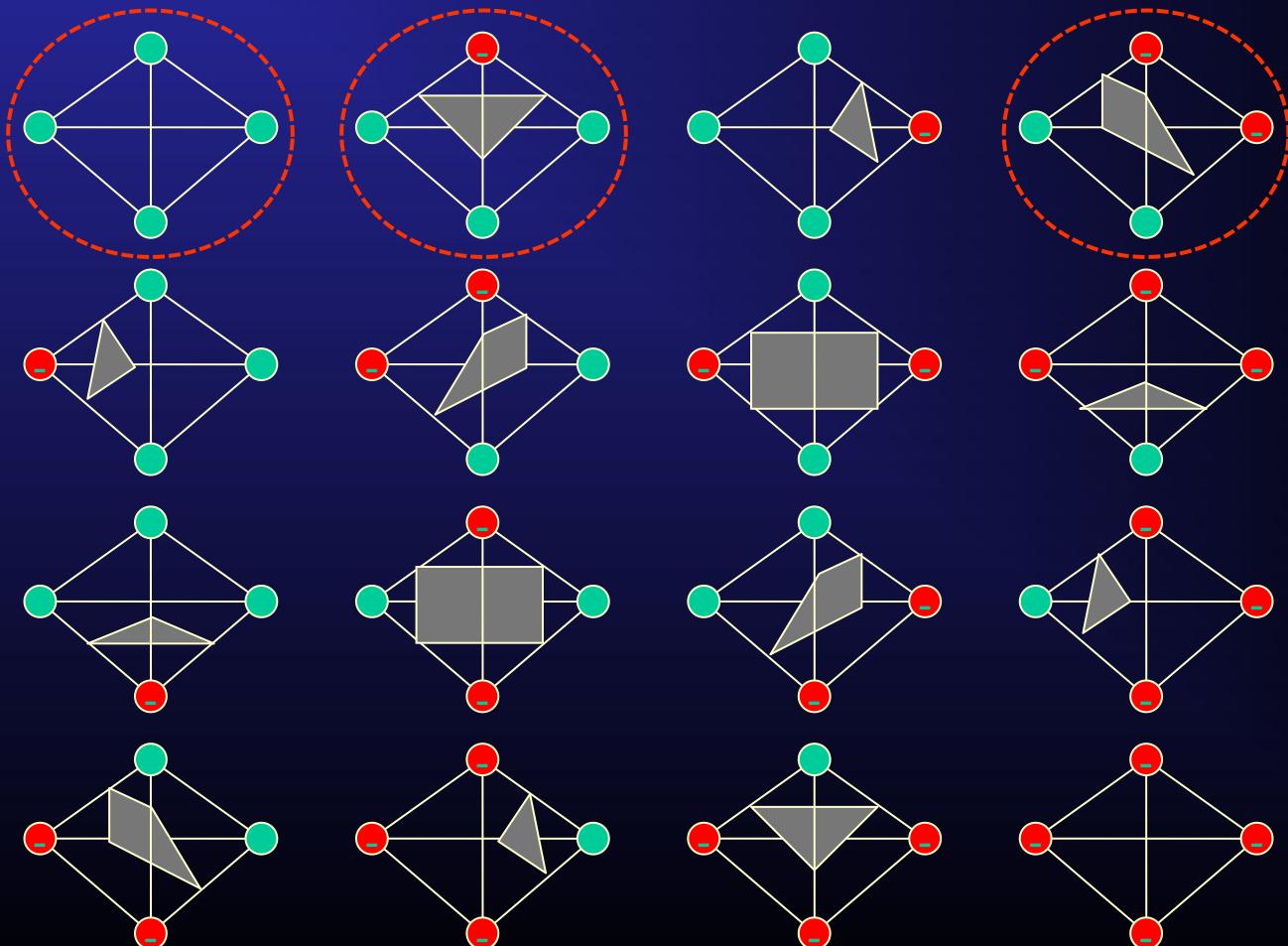
Marching Cube Cases

256 cases; 15 modulo symmetry. Note multiple facets.



Marching Tetrahedrons (Simplices) Cases

16 cases; 3 modulo symmetry; note maximum 1 polygonal facet each: this is an advantage to tetrahedral meshes!



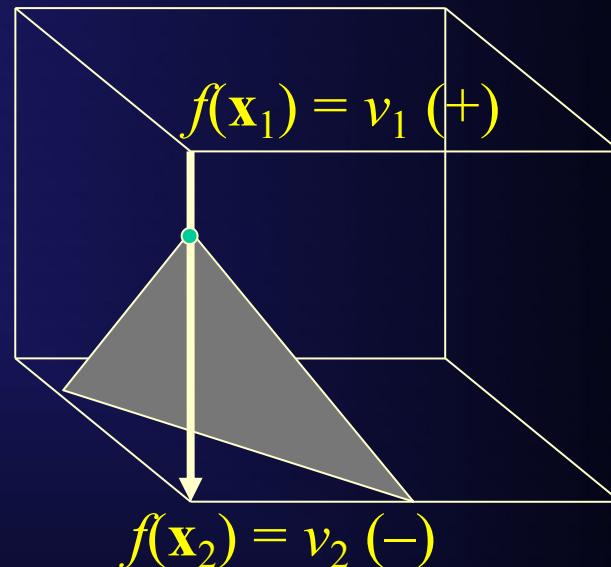
Surface Vertex Computations

- Determine where implicit surface intersects cell edges
- EITHER numerically find zero of $f(\mathbf{r}(t))$:

$$\mathbf{r}(t) = \mathbf{x}_1 + t(\mathbf{x}_2 - \mathbf{x}_1)$$

$$0 \leq t \leq 1$$

- OR just linearly interpolate function values to approximate:

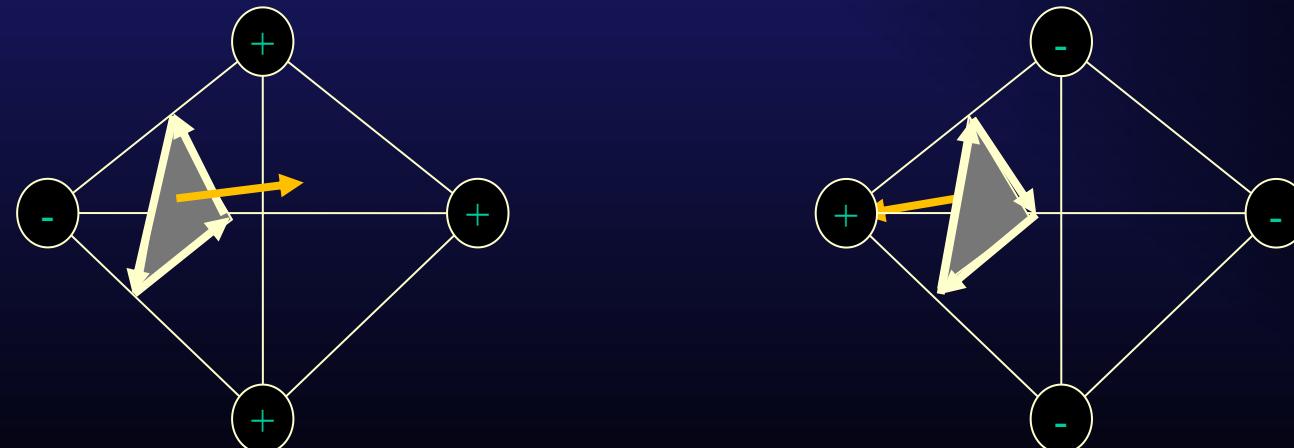


$$\mathbf{x} = \frac{v_1}{v_1 - v_2} \mathbf{x}_1 + \frac{v_2}{v_1 - v_2} \mathbf{x}_2 \quad (\text{assuming } v_1 > 0 \text{ and } v_2 < 0)$$

Facet Orientation

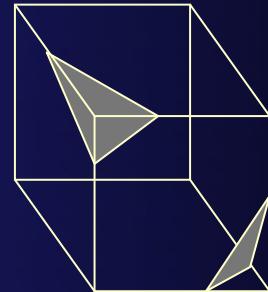


- Consistency allows polygons to be drawn with correct orientation: e.g., normal points to the “outside” (+):

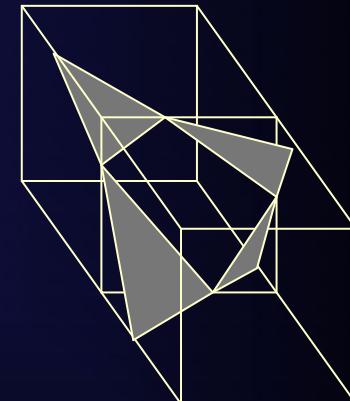
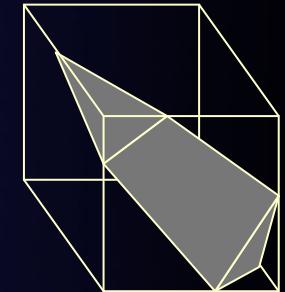


Problem: Ambiguity

- Some cell corner value configurations yield more than one consistent polygon
- Only for cubes, not tetrahedra (why?)
- In 3D can yield holes in surface!
- How can we resolve these ambiguities?

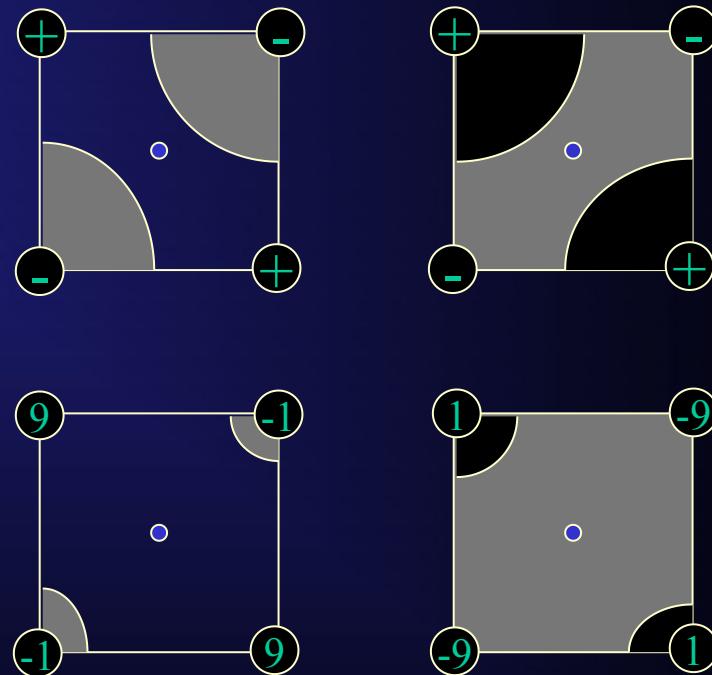


or



Topological Inference

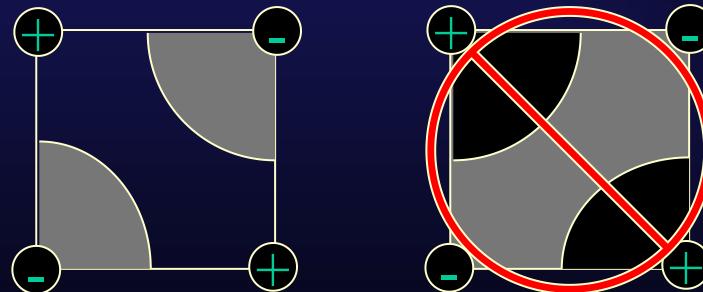
- Sample a point in the center of the ambiguous face.
- If data is discretely sampled, bilinearly interpolate samples.



$$p(s,t) = (1-s)(1-t) a + s (1-t) b + (1-s) t c + s t d$$

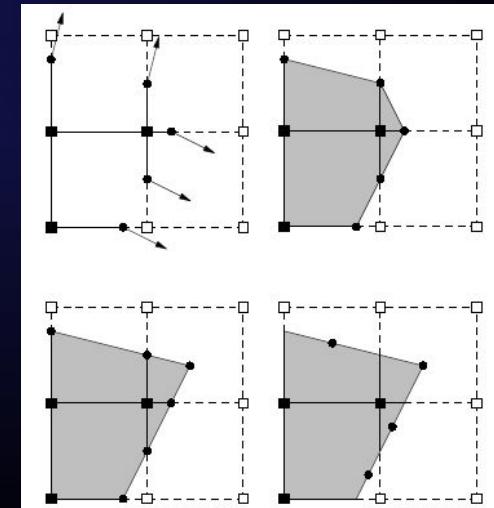
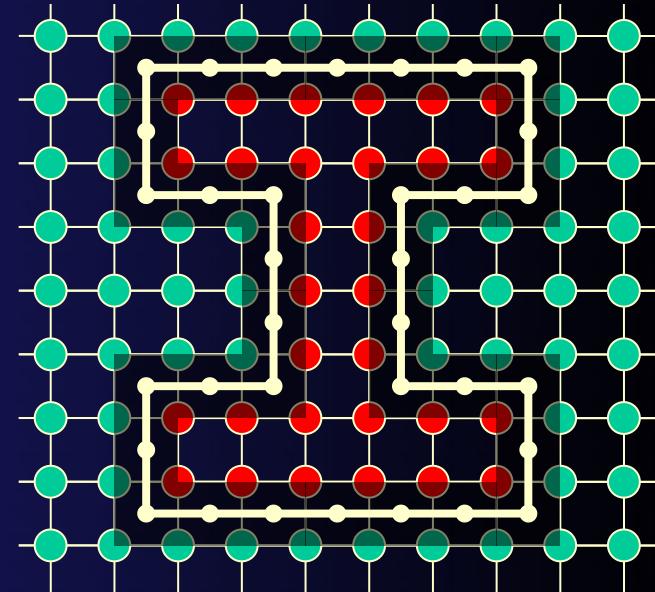
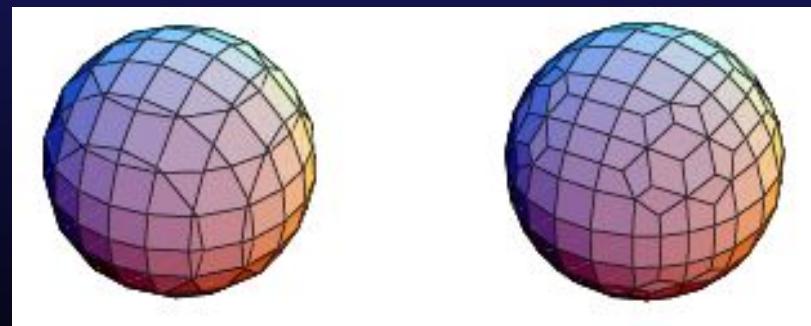
Preferred Polarity

- Assume ambiguous face centers always +
- (or always -)
- Preference can be encoded into table



Dual Contouring

- Surface Nets, Volume Visualization '98.
- Vertices **in cells**, not along edges, of cubes.
- Greater freedom in placement leads to better triangle aspect ratios.
- Works well with hierarchy, subdivision.



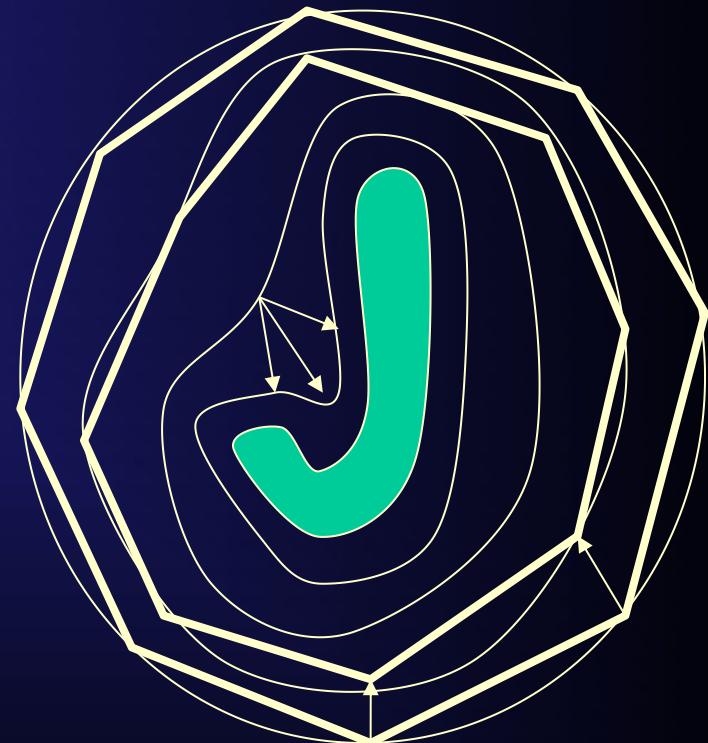
Another Polygonization Method: Use Particle (Surfel) Systems

- Witkin & Heckbert SIGGRAPH 1994.
- Constrain sets of oriented particles (surfels) to implicit surface: implicit surface $f = 0$ becomes constraint surface $C = 0$.
- Particles exert repulsive forces on each other to spread out across surface.
- Particles subdivide to fill open gaps.
- Particles commit suicide if overcrowded.
- Display particle as oriented disk.
- Constrain implicit surface to particles for interactive design!

Using Particles
to
Sample and Control
Implicit Surfaces

Shrinkwrapping

- Look at family of surfaces $f^{-1}(s)$ for $s > 0$
- For s large, $f^{-1}(s)$ is spherical
- Polygonize sphere
 - Choose sphere mesh in advance
- Reduce s to zero
 - Allow vertices to track surface
 - Subdivide polygons as necessary when curvature increases
- E.g., Zhao, Osher, Fedkiw: Fast Surface Reconstruction Using the Level Set Method, Proc. IEEE Workshop on Variational and Level Set Methods, 2001.



Other Geometric Issues

- Model blending
- Multi-resolution methods
- Polygon decimation
- Model generation

Model Blending

- Have a set of example models $\{\mathbf{x}_k\}$ in different shapes or poses.
- Have a correspondence between comparable vertices in each model.
- New model = vertex-wise weighted blend of example models.
- Can be smarter choosing exemplars: a basis set.
- Interpolate (blend) via **Radial Basis Functions**.

Radial Basis Functions

- Basically think “potential functions” but use them to **interpolate example models** rather than **be** a model.
- E.g., use Gaussian distribution around each sample (σ sets range of influence):

$$\psi(r) = e^{(-r^2/2\sigma^2)}$$

- If each sample or basis model is called \mathbf{x}_k , then the “influence” of sample \mathbf{x}_k elsewhere in the model space is just:

$$\psi(\|\mathbf{x} - \mathbf{x}_k\|)$$

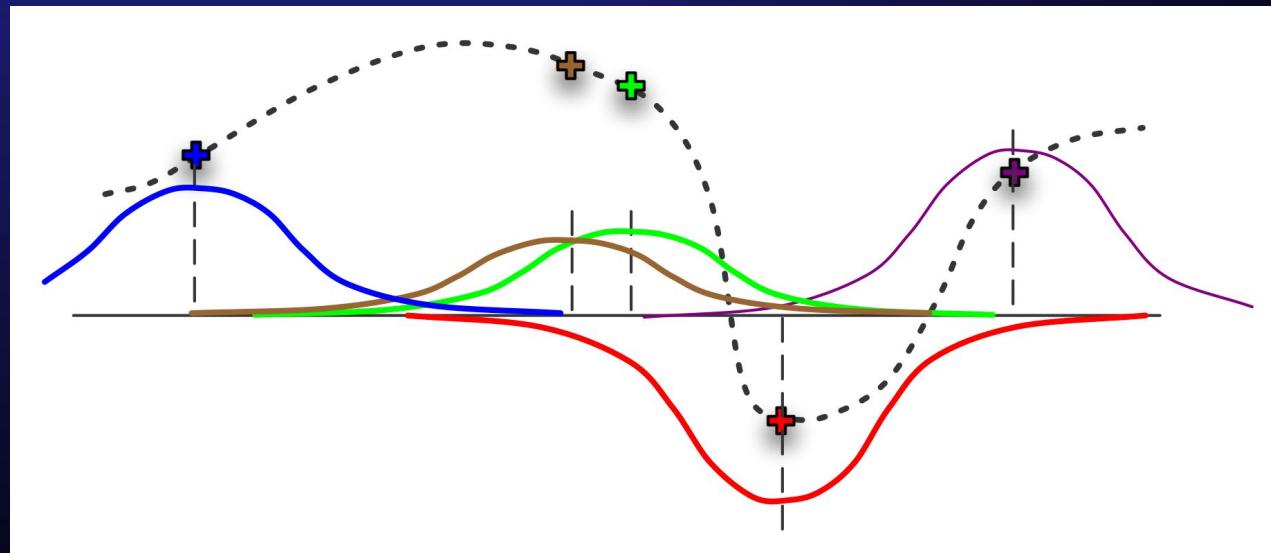
- Note that the influence of a sample \mathbf{x}_k is $e^0=1$ at “itself”.

Radial Interpolation

- The interpolated model is a weighted sum of the samples \mathbf{x}_k :

$$d(\mathbf{x}) = \sum_k^N w_k \psi(\|\mathbf{x} - \mathbf{x}_k\|)$$

- Below: 1D example with 5 \mathbf{x}_k (“+” show interpolated value sums where all the weights w_k are 1):



- What if we want $d(\mathbf{x})$ to pass through the samples (the basis functions)? Have to solve for the weights w_k .

Radial Basis Functions – Solving for the Weights (Using Linear Least Squares)

$$d(\mathbf{x}) = \sum_k^N w_k \psi(\|\mathbf{x} - \mathbf{x}_k\|) \quad (\text{Assume } \sigma \text{ is the same for each } \psi.)$$

Write ψ_{ij} as shorthand for $\psi(\|\mathbf{x}_i - \mathbf{x}_j\|)$. Note that $\psi_{ii} = 1$.

$$\text{Then } d_i = d(\mathbf{x}_i) = w_i + \sum_{k \neq i}^N w_k \psi(\|\mathbf{x}_i - \mathbf{x}_k\|).$$

Let $\mathbf{d} = [d_1 \dots d_N]^T$; $\mathbf{W} = [w_1 \dots w_N]^T$ and

$$\Psi = \begin{bmatrix} \psi_{11} & \dots & \psi_{1N} \\ \vdots & \ddots & \vdots \\ \psi_{N1} & \dots & \psi_{NN} \end{bmatrix} = \begin{bmatrix} 1 & \dots & \psi_{1N} \\ \vdots & \ddots & \vdots \\ \psi_{N1} & \dots & 1 \end{bmatrix}. \quad (\text{Note that } \Psi \text{ is symmetric.})$$

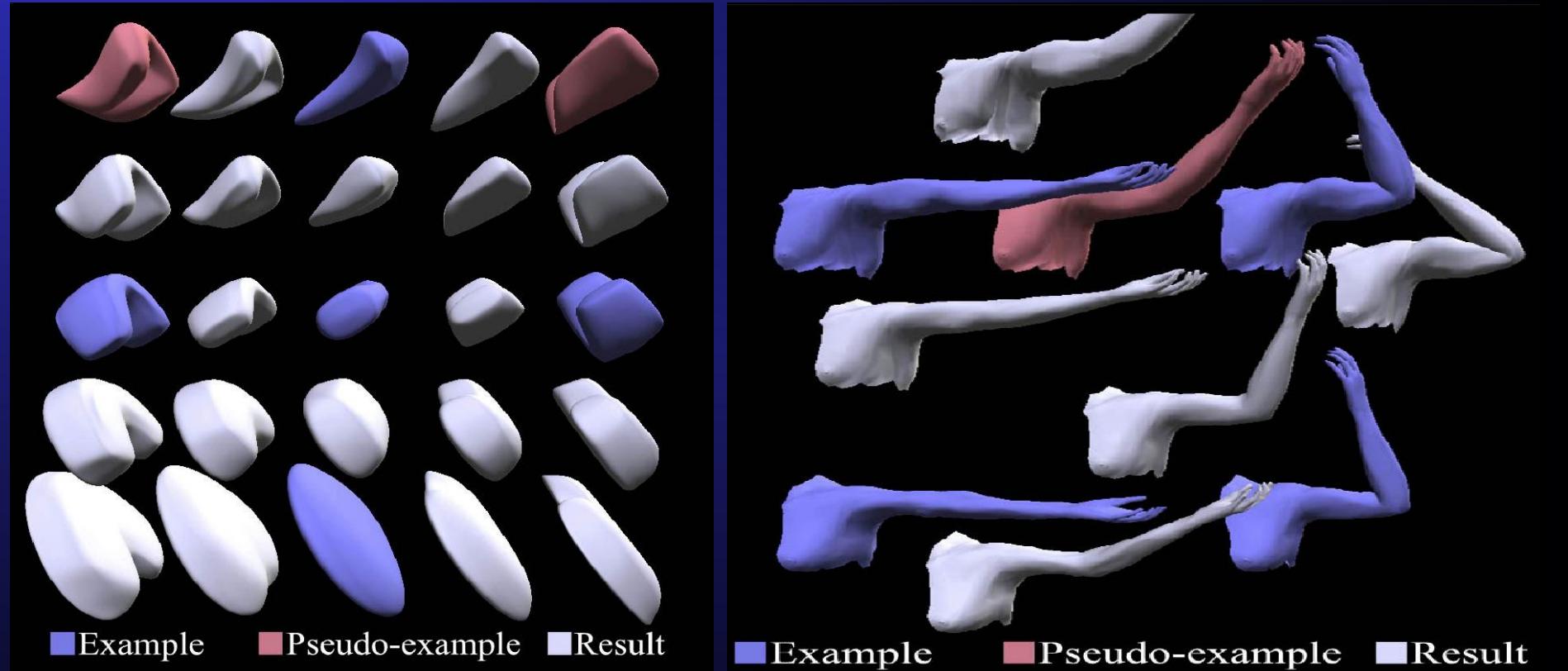
Then

$$e = \|(\mathbf{d} - \Psi \mathbf{W})\|^2 = (\mathbf{d} - \Psi \mathbf{W})^T (\mathbf{d} - \Psi \mathbf{W})$$

$$\frac{de}{d \mathbf{W}} = 0 = -\Psi^T (\mathbf{d} - \Psi \mathbf{W}) \quad \text{Thus } \Psi^T \mathbf{d} = \Psi^T \Psi \mathbf{W}$$

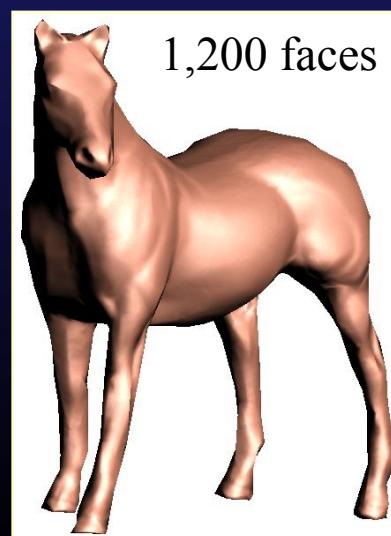
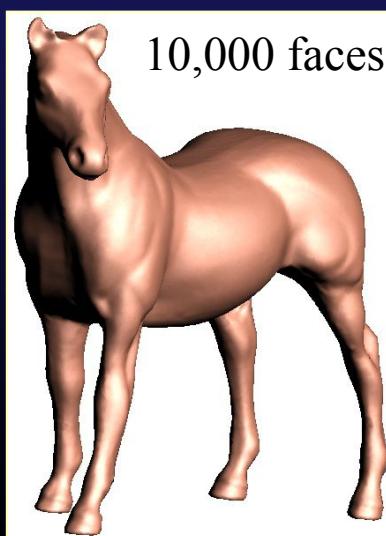
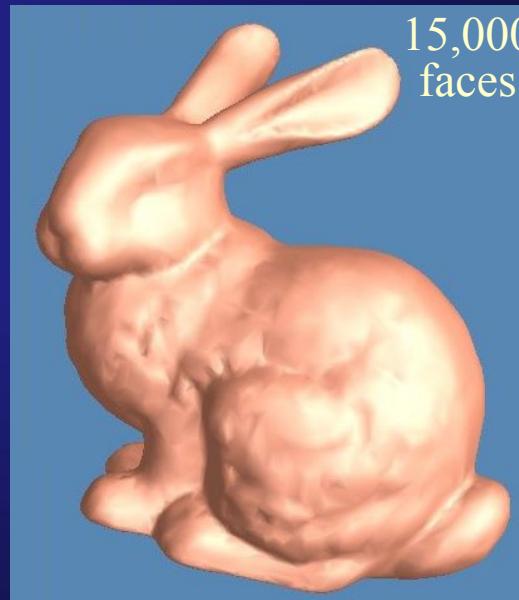
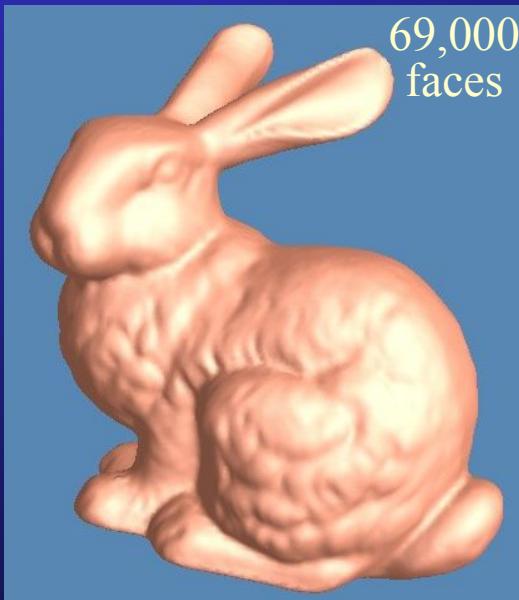
$$\therefore \mathbf{W} = (\Psi^T \Psi)^{-1} \Psi^T \mathbf{d} = \Psi^{-1} \mathbf{d} \quad \text{So just need to invert matrix } \Psi.$$

Examples



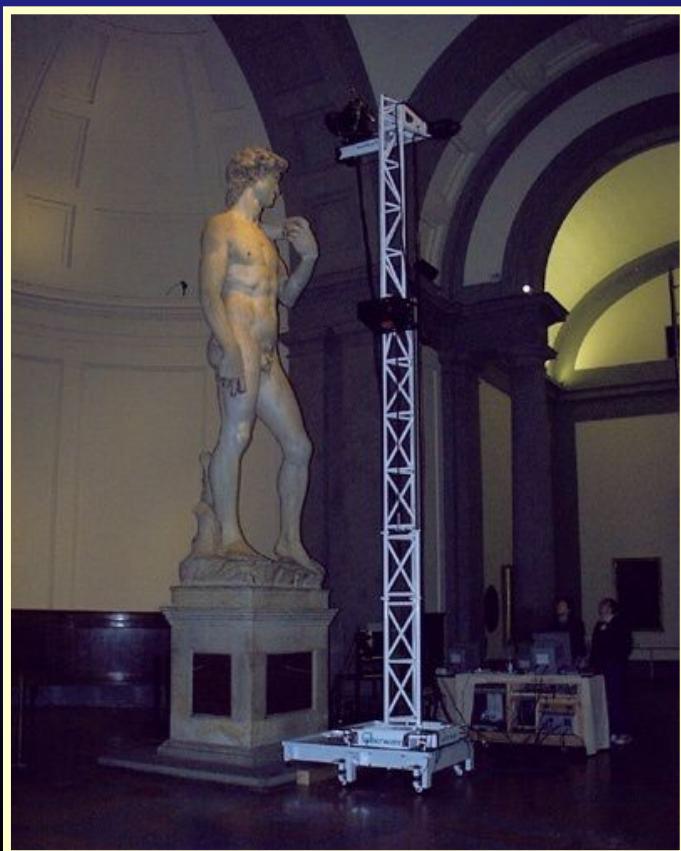
- (A **pseudo-example** is a manually specified example that differs from the interpolated shape. Usually do this to avoid some mesh nastiness, such as an unwanted fold.)

Multi-Resolution Methods (Level of Detail): Decimate (remove detail) or Subdivide (smooth)

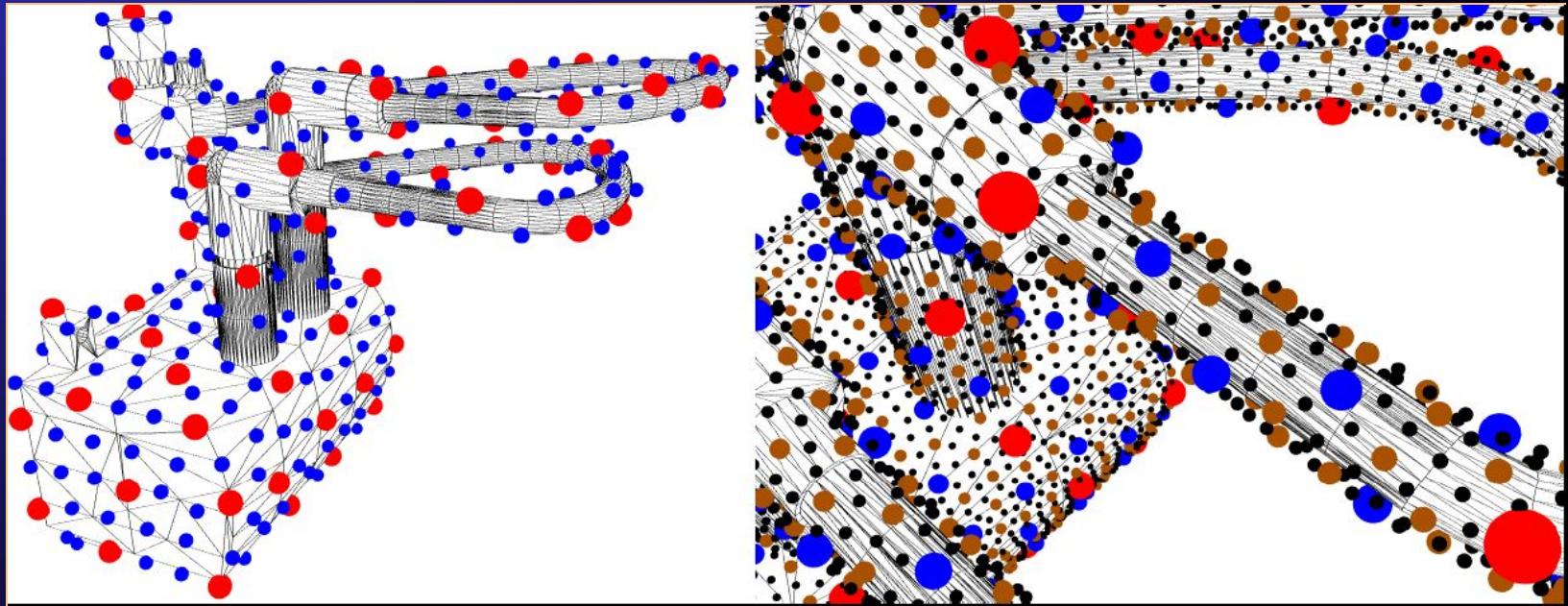


Example: Digital Michelangelo Project: Stanford and University of Washington

- Michelangelo's David sculpture (left) scanned at 0.29 mm resolution creating 2 billion triangles.
- The renderings (right) are based on 2mm 8 million polygon data.



(Multi-Resolution) Point Shells



Two levels

Four levels

(These point shells are defined by surfels as they also include inward pointing normals.)

Outline

- Great Ideas ✓
- Virtual Environments & Devices ✓
- 3D Perception ✓
- Architecture, Color & Rasterization ✓
- Vector Geometry & Transformations ✓
- 3D Modeling ✓
- Embedding, Hierarchies & Contours ✓
- Model Generation & Deformation
- Visible Surface Algorithms
- Polygon Algorithms
- Image Synthesis & Shading
- Mapping
- 3D Viewing Transformations
- Anti-aliasing & Compositing
- Global Illumination
- Light Fields & HDR
- Path Tracing/Photon Mapping
- Futures

Model Generation

- Scene graphs
- Manual digitization
- Semi-automated data acquisition
- Algorithmic methods
- Procedural models

Grouping Transformations in a *Scene Graph*

- Motivation:
 - Often an object has a part or sub-structure.
 - Repeated instances of an object need not be separately constructed.
 - Organize transformations.
 - Organize geometry (shapes).

Grouping Transformations in a *Scene Graph*

- Create a directed **tree** of transformations:
 - Geometry at terminal nodes.
 - Non-terminal nodes become transformed instances.
- The same geometry primitive may be referenced by leaf nodes and transformed several times resulting in a **scene graph**.
- Pictorially...

Scene Graph Elements

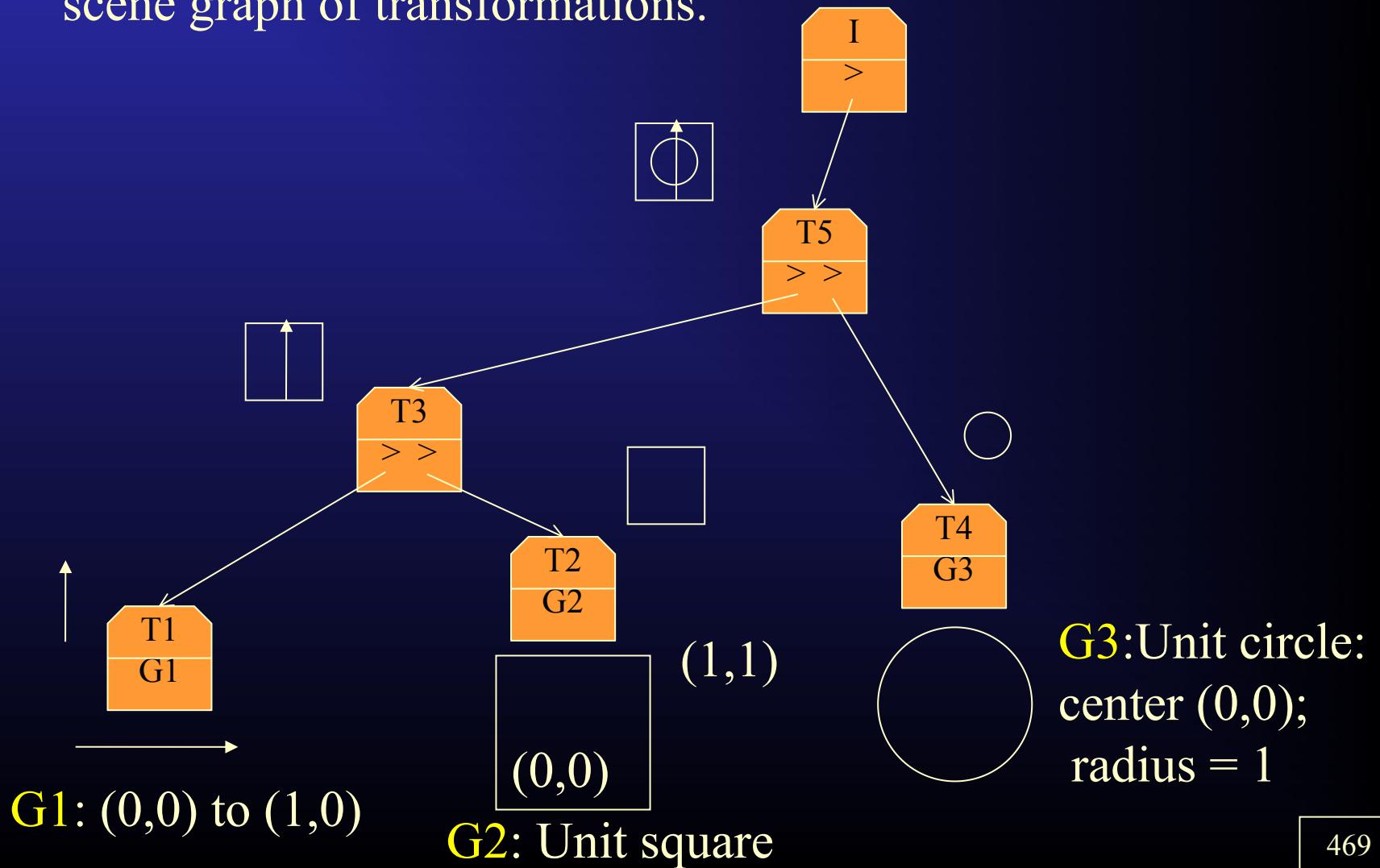
- Generic scene graph node:



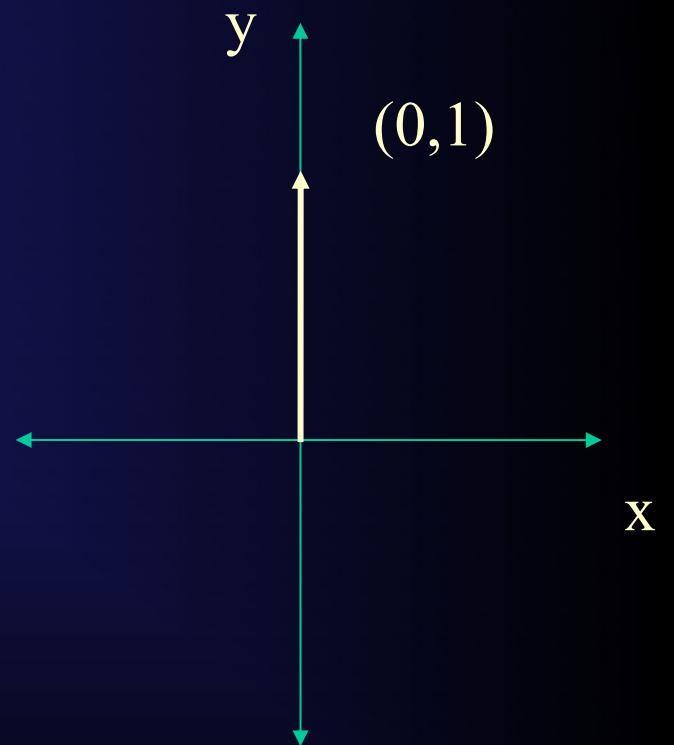
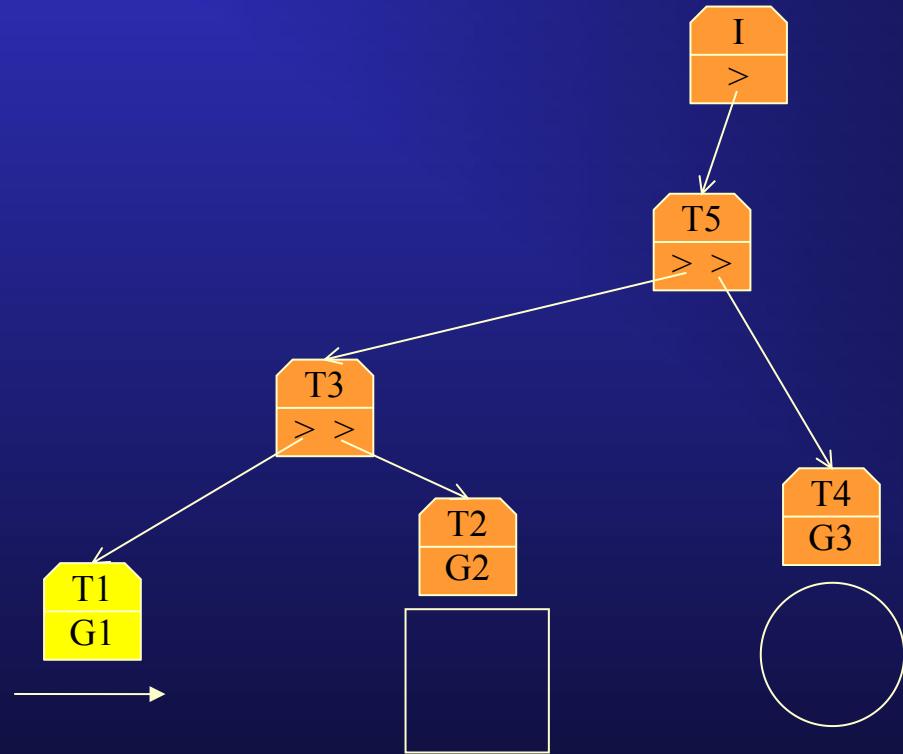
- Each child points to another (lower) scene tree node or to stored geometry.

Sample Scene Graph

Shown is a 2D shape built from simple geometric parts in a scene graph of transformations.

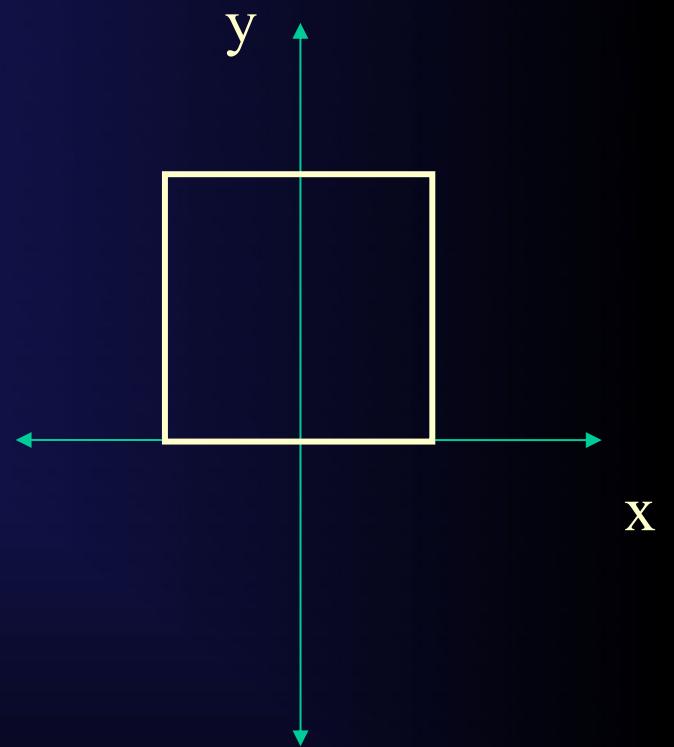
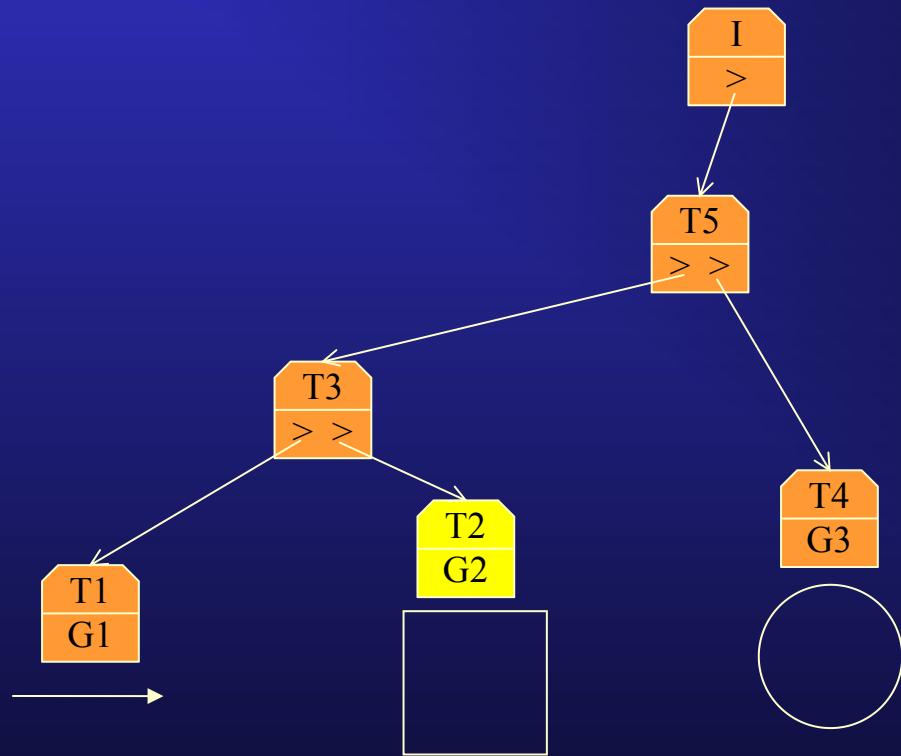


$T_1 = \text{Rotate by } 90^\circ$



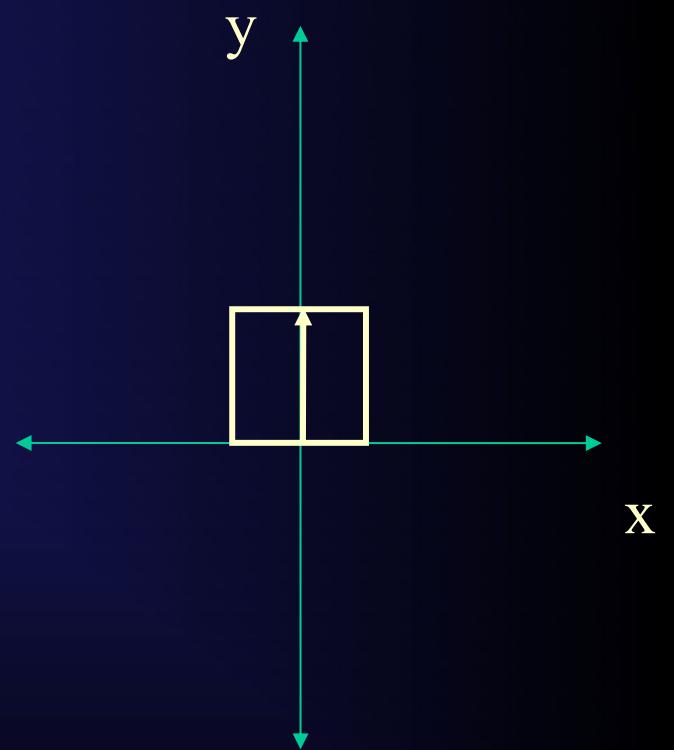
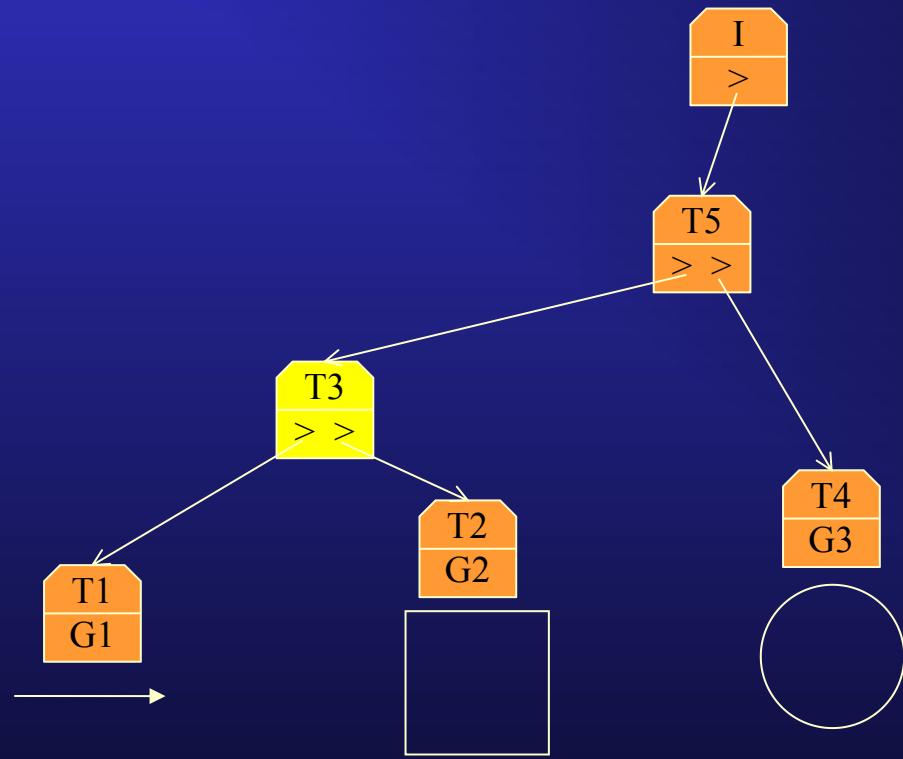
$$\begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$T_2 = \text{Translate by } (-0.5, 0)$



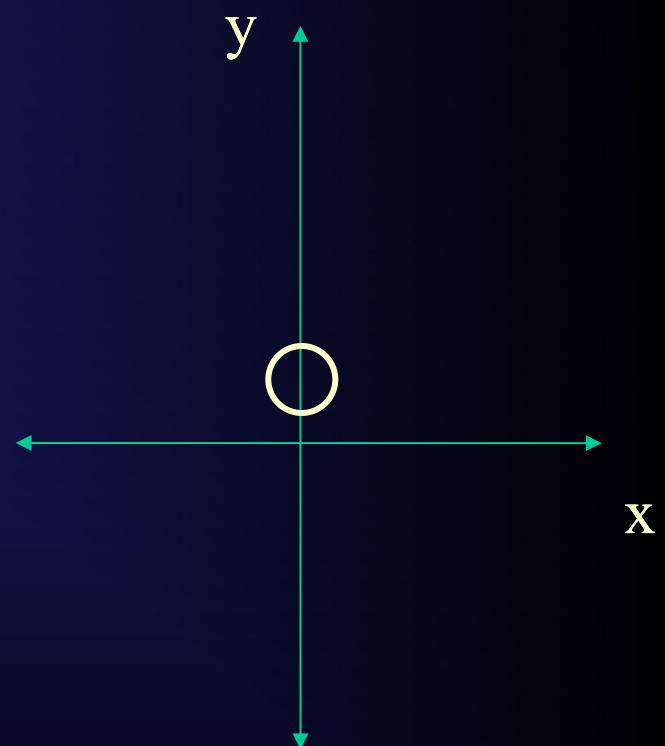
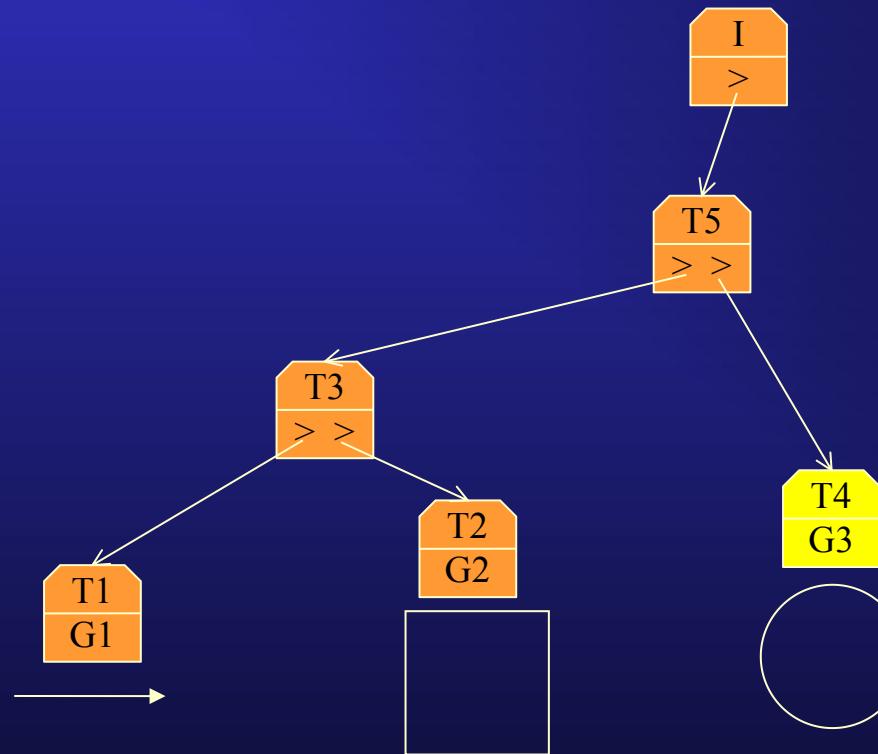
$$\begin{bmatrix} 1 & 0 & -0.5 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$T_3 = \text{Scale by } 0.5 \text{ in both } x \text{ and } y$



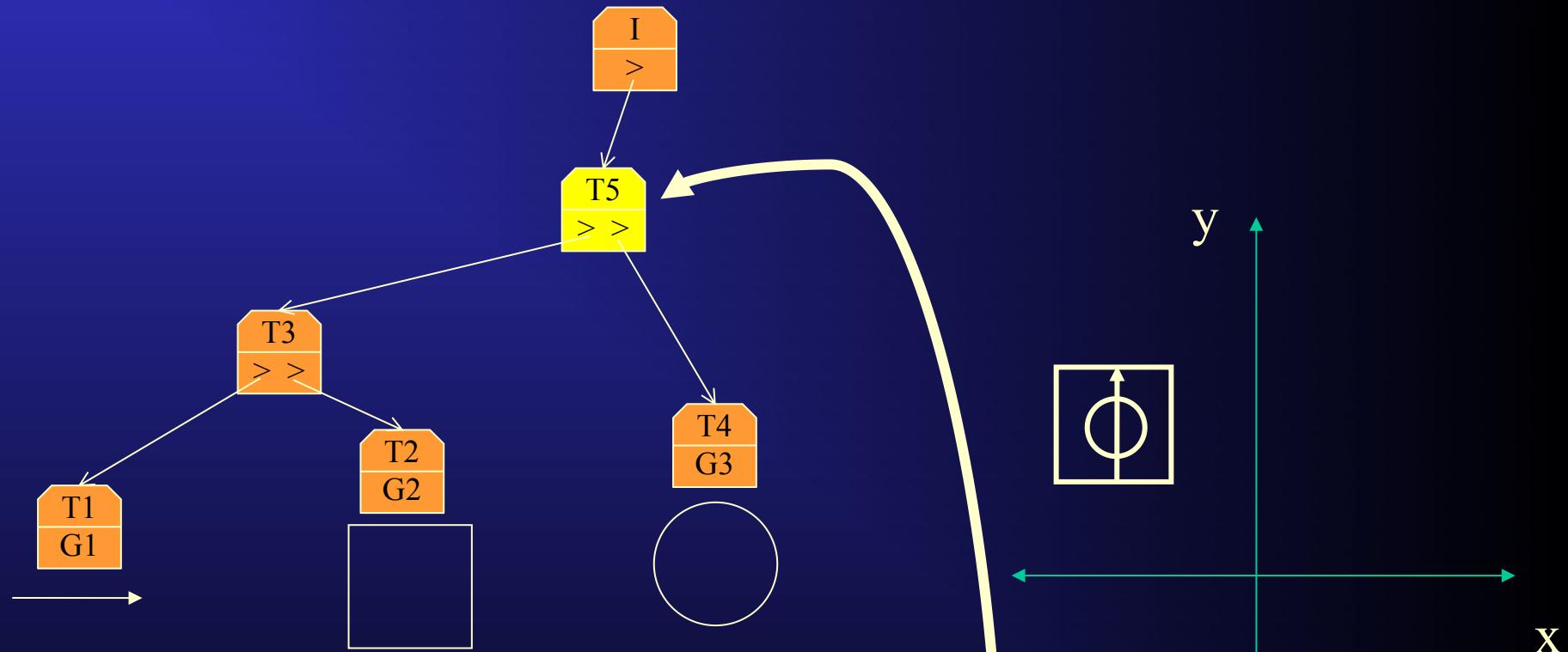
$$\begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$T_4 = \text{Scale by } 0.15 \text{ in } x \text{ and } y \text{ THEN Translate by } (0, 0.25)$



$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0.25 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.15 & 0 & 0 \\ 0 & 0.15 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$T5 = \text{Translate by } (-0.6, 0.4)$



$$\begin{bmatrix} 1 & 0 & -0.6 \\ 0 & 1 & 0.4 \\ 0 & 0 & 1 \end{bmatrix}$$

Think of this as a new object (group)
instance that can itself be
transformed.

“Executing” the Scene Graph: INORDER Traversal of the Transformation Tree

Traverse (root.child, I);

Traverse (N, T):

 T= T•N.transformation; //push (multiply) transformation on right

 for (i=0; i<N.child()-1; i++)

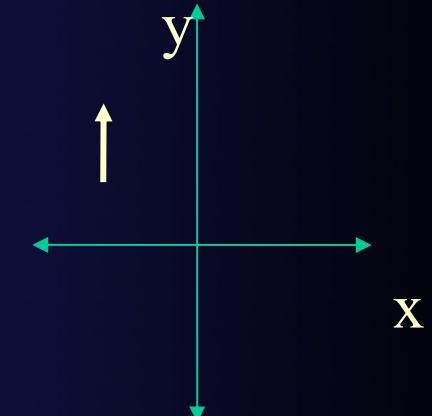
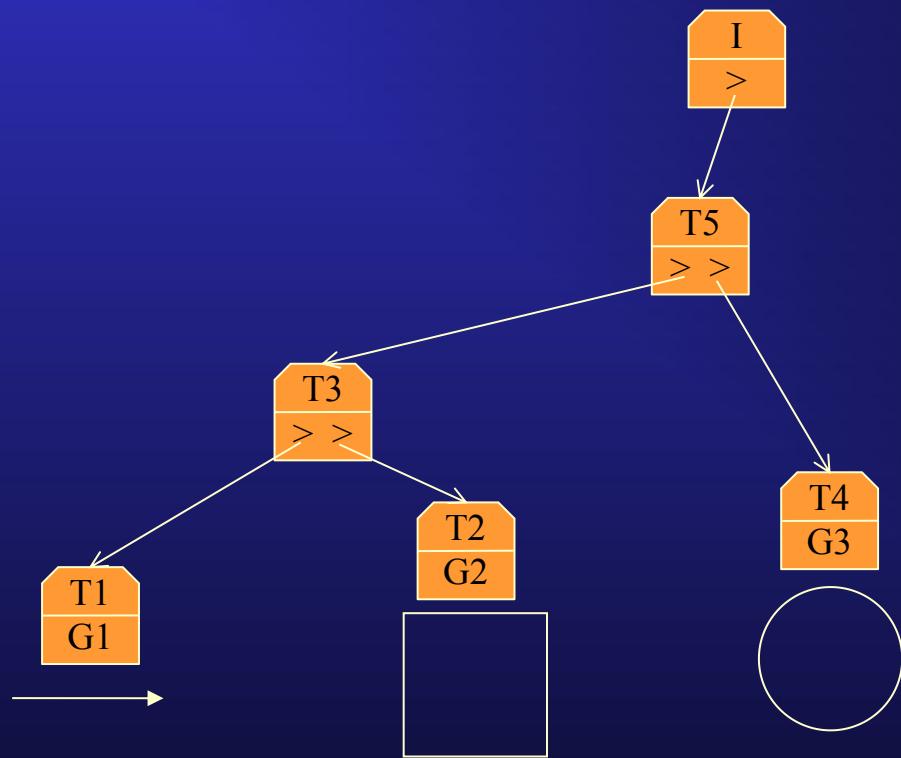
 If N.child[i] points to geometry

 then apply T to N.child[i] and draw it

 else Traverse(N.child[i], T)

- Recursion works for you: Go up \Rightarrow implicit POP transformation from right: [.] T \rightarrow [.]
- For our example, we just created all the transformation matrices.
- Though we'll draw the stack, it is really hidden in the procedure calls; you don't create it explicitly.

Scene Graph Example



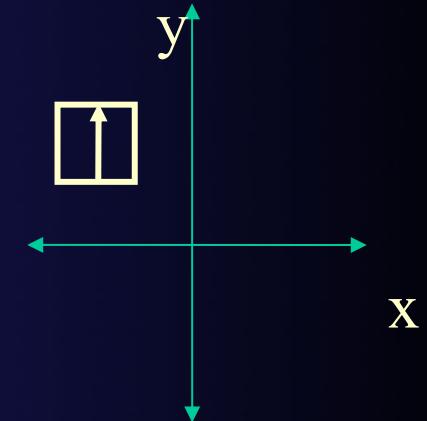
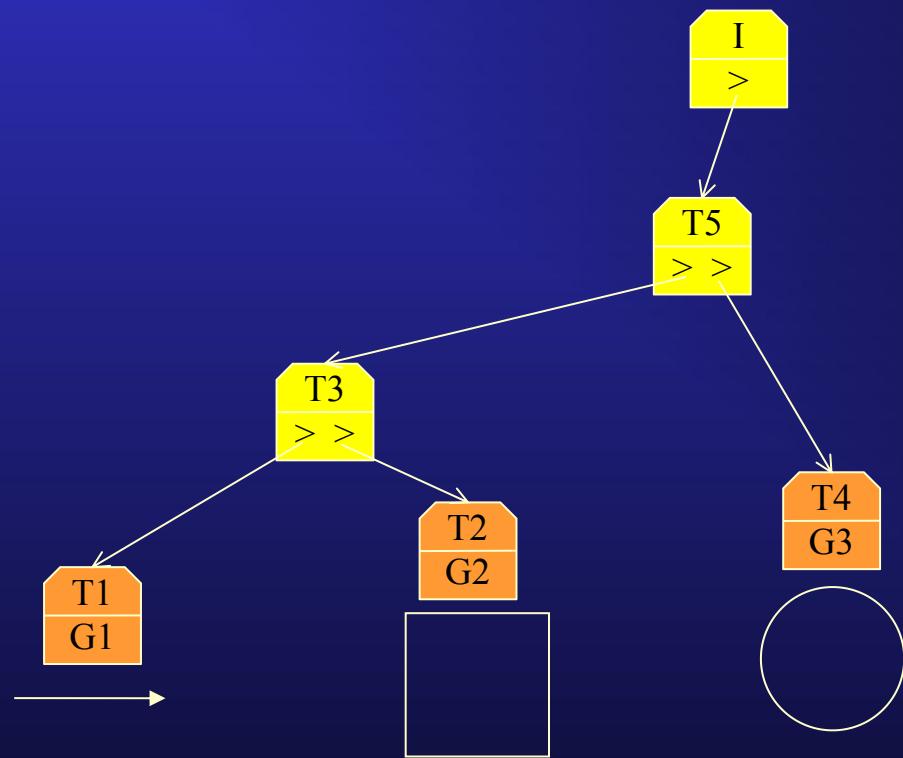
I T5 T3 T1

Then transformation of **G1** line segment is this matrix product:

$$\begin{bmatrix} -0.6 \\ 0.4 \\ 1 \end{bmatrix} = (T5 \quad T3 \quad T1) \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} -0.6 \\ 0.9 \\ 1 \end{bmatrix} = (T5 \quad T3 \quad T1) \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

Scene Graph Example

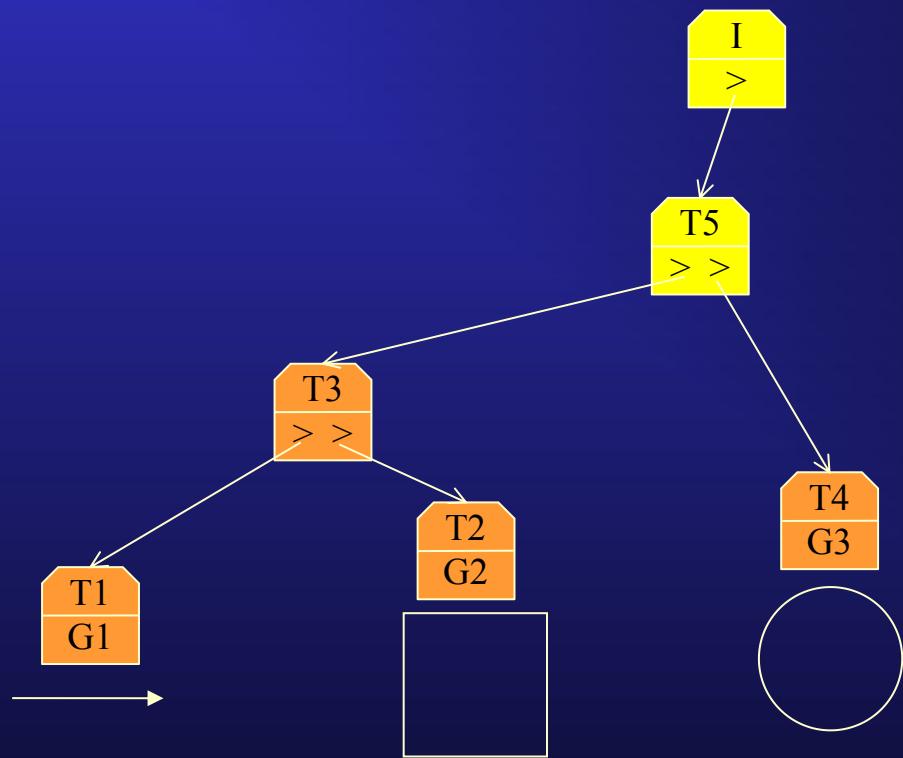


I T5 T3 T2

Then transformation of G2 square is this matrix product:

$$\begin{bmatrix} -0.85 \\ 0.4 \\ 1 \end{bmatrix} = (T5 \quad T3 \quad T2) \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad \begin{bmatrix} -0.35 \\ 0.9 \\ 1 \end{bmatrix} = (T5 \quad T3 \quad T2) \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \quad (+ \text{ two more } \dots)$$

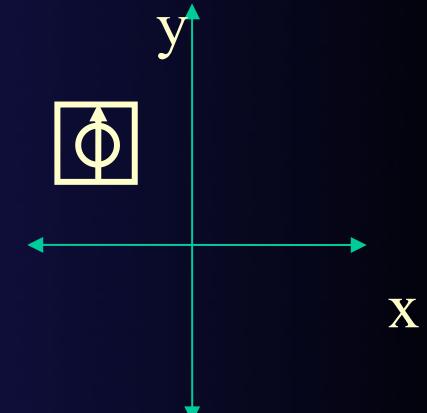
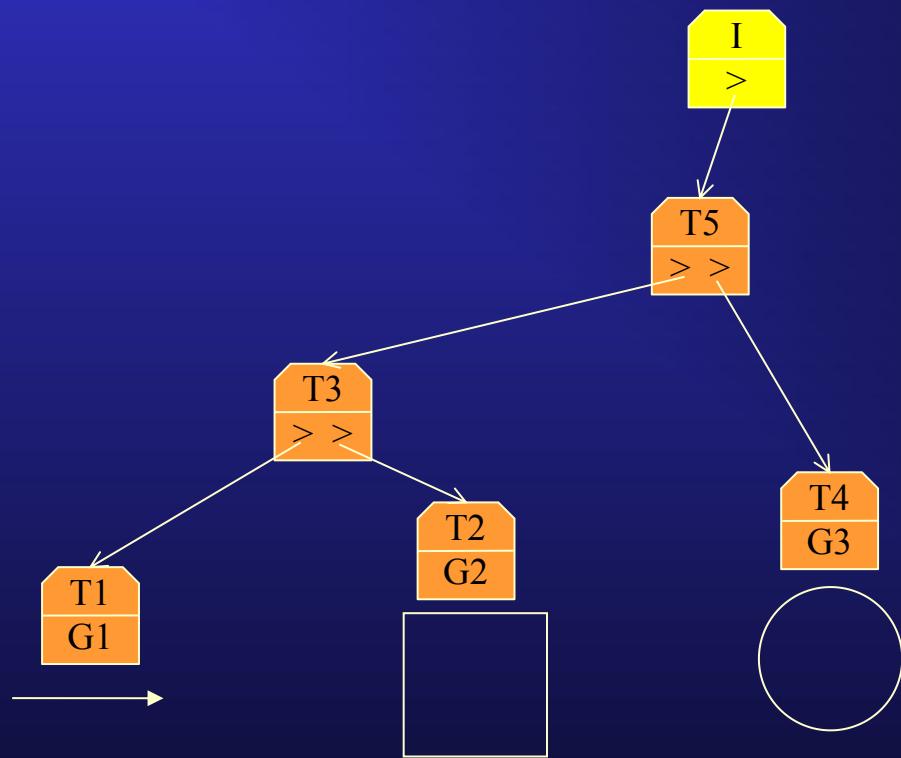
Scene Graph Example



Then transformation of G3 circle is this matrix product:

$$\begin{bmatrix} -0.6 \\ 0.65 \\ 1 \end{bmatrix} = (T5 \quad T4) \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad (\text{New radius} = 0.15)$$

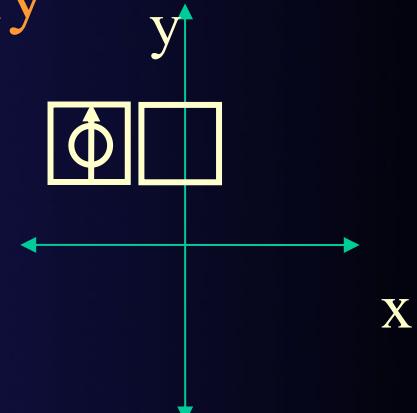
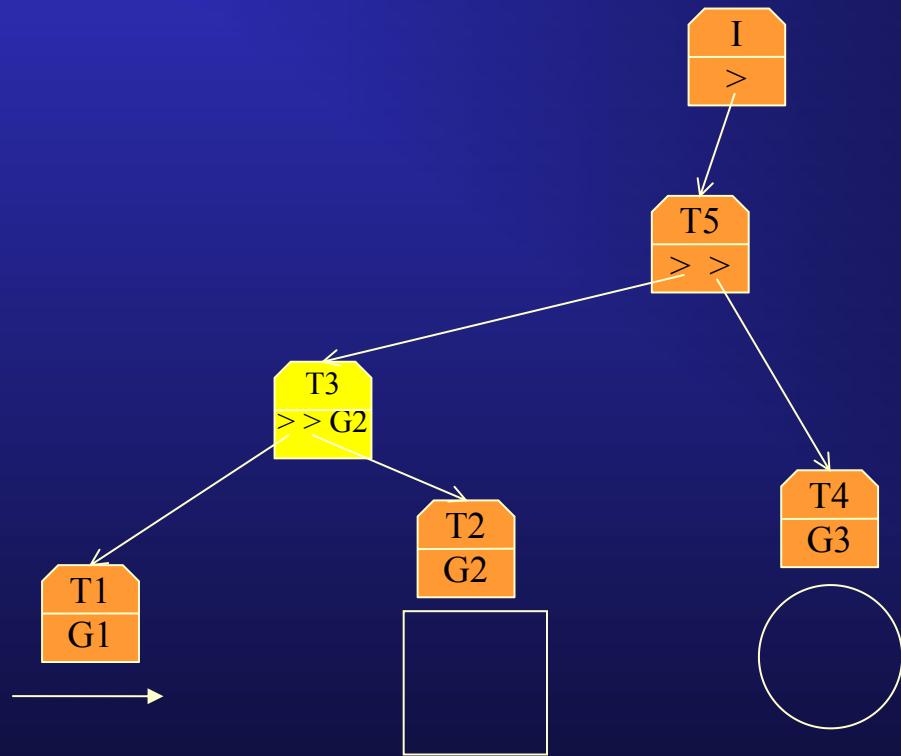
Scene Graph Example



I

The transformation stack is empty so we are done.

Scene Graph with “Internal” Geometry



- We added a third child to the node with T3, pointing to geometry G2 (the unit square). The result is that this instance of G2 is only transformed by (T5 T3): scaled by 0.5 in x and y and then translated by (-0.6,0.4).
- This feature can be used to create articulated parts (such as character arms and legs), or objects such as tables with supported (sub-)objects.

Scene Graph as Vector of Links

(Like a folder/file structure; see PowerPoint ‘Selection Pane’)

T; root

T; part-A

T; part-AA

T; part-AB

T; part-AC

T; part-B

T; part-C

T; part-CA

T; part-CB

T; part-CBA

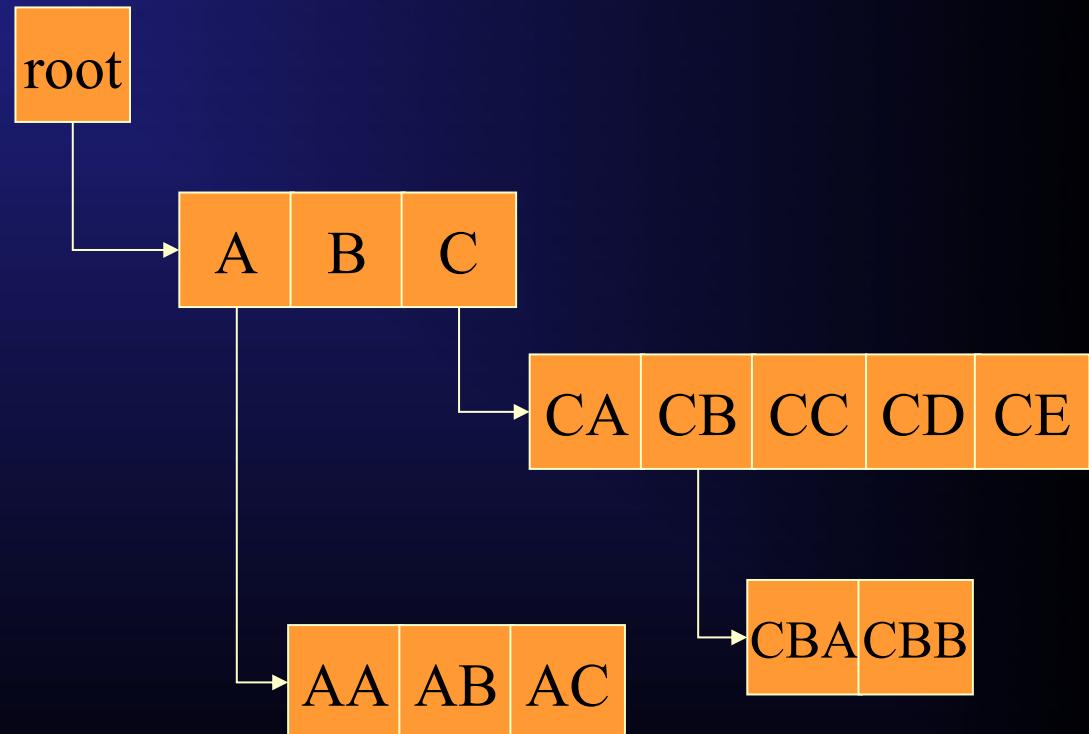
T; part-CBB

T; part-CC

T; part-CD

T; part-CE

(where the T’s are the transformations
and where the “parts” contain geometry
and/or sub-trees)



Manual Digitization

- From plans, models, or the real thing.
- Must have or build model or real thing.
- Encode coordinates with interactive tools.
 - use spatial digitizers.
- Photogrammetric (stereo or multiple view) techniques.
 - needs feature points, careful calibration, real objects.

Semi-Automated Data Acquisition

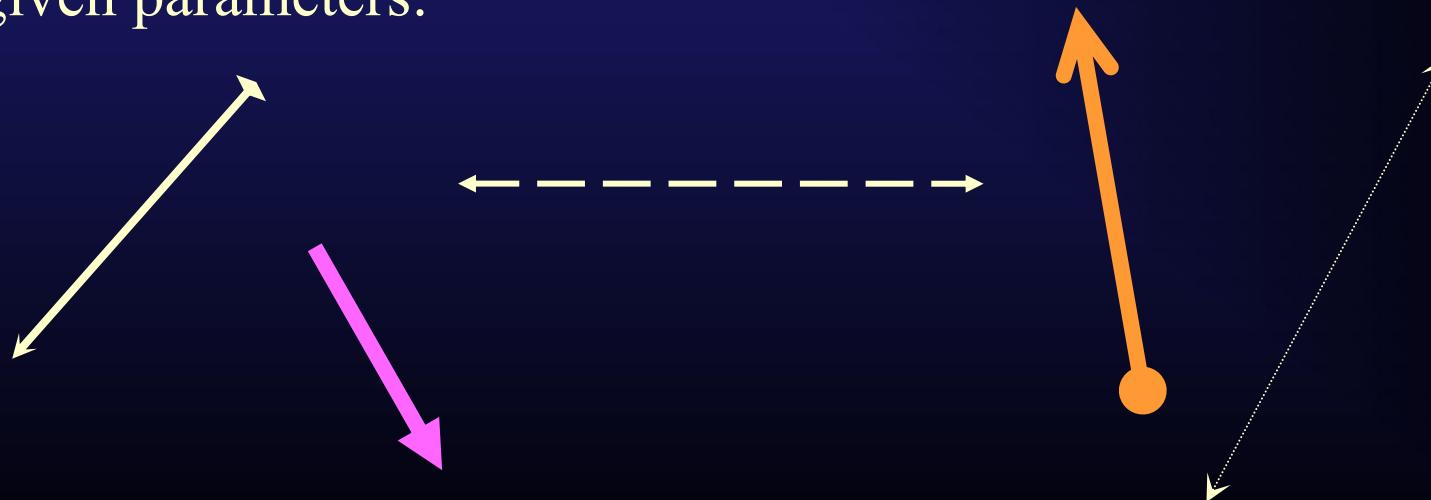
- Direct 3D coordinate input:
 - e.g. CT scanning and reconstruction.
- Laser scanning. (e.g., Digital Michelangelo):
 - e.g. light stripe or grid encoding.
 - intersection of a known light plane or ray and camera direction ray gives unique 3D point.
- Video image analysis (image processing and computer vision):
 - shape reconstruction problem.
 - “zippering” multiple views into a single topologically and geometrically coherent model.

Algorithmic Methods

- Interactive graphical editor (e.g., CAD systems, Maya):
 - Instantiate primitives.
 - Extrude, duct, surface of revolution, sweep, height field,...
 - Transform, join, combine, deform, create hierarchy.
 - Assign attributes.
- Enforced constraints (e.g., SketchUp and ProEngineer):
 - Use design constraints to enforce horizontal, vertical, parallel, touch, perpendicular, tangent, etc.
 - Work from 2 views, 2 points over-constrain 3 coordinates.
- Interpolation between parallel contours:
 - Triangulate surface between contours.
 - End cap and branching problems.

Procedural Models

- “Database amplification”: Parametric generation of complexity and variants in shapes.
- Code plus parameters to generate variants of a class:
e.g. `Arrow(tail_x, tail_y, head_x, head_y, tail_style,
head_style, line_width, line_style, angle, color)`
- Create a graph instance of the model with each call and given parameters:



Procedural Models of Many Things

- Buildings
- City layouts
- Vegetation
- Motion
- Crowds

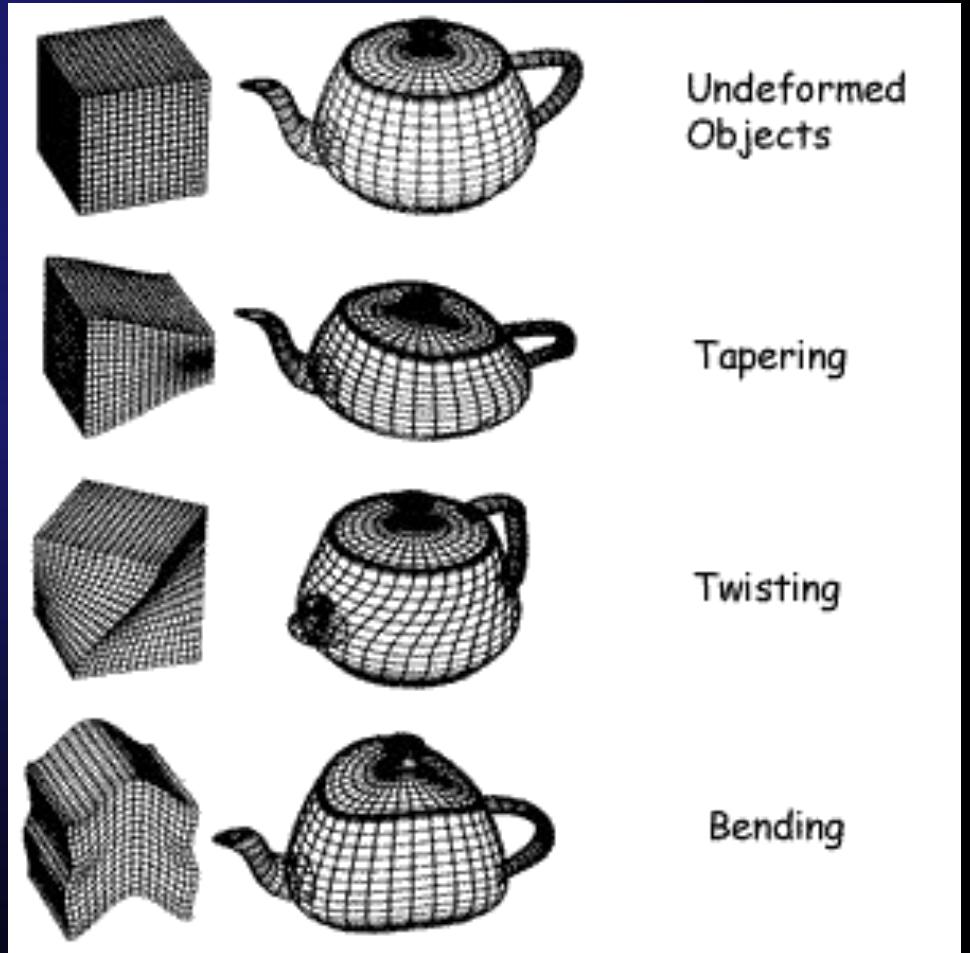
Procedural methods offer parametric control, flexibility and variety.

Simple Object Deformation Methods

- Global deformation (taper, twist, bend)
- Free-Form Deformation (FFD) in 2D and 3D

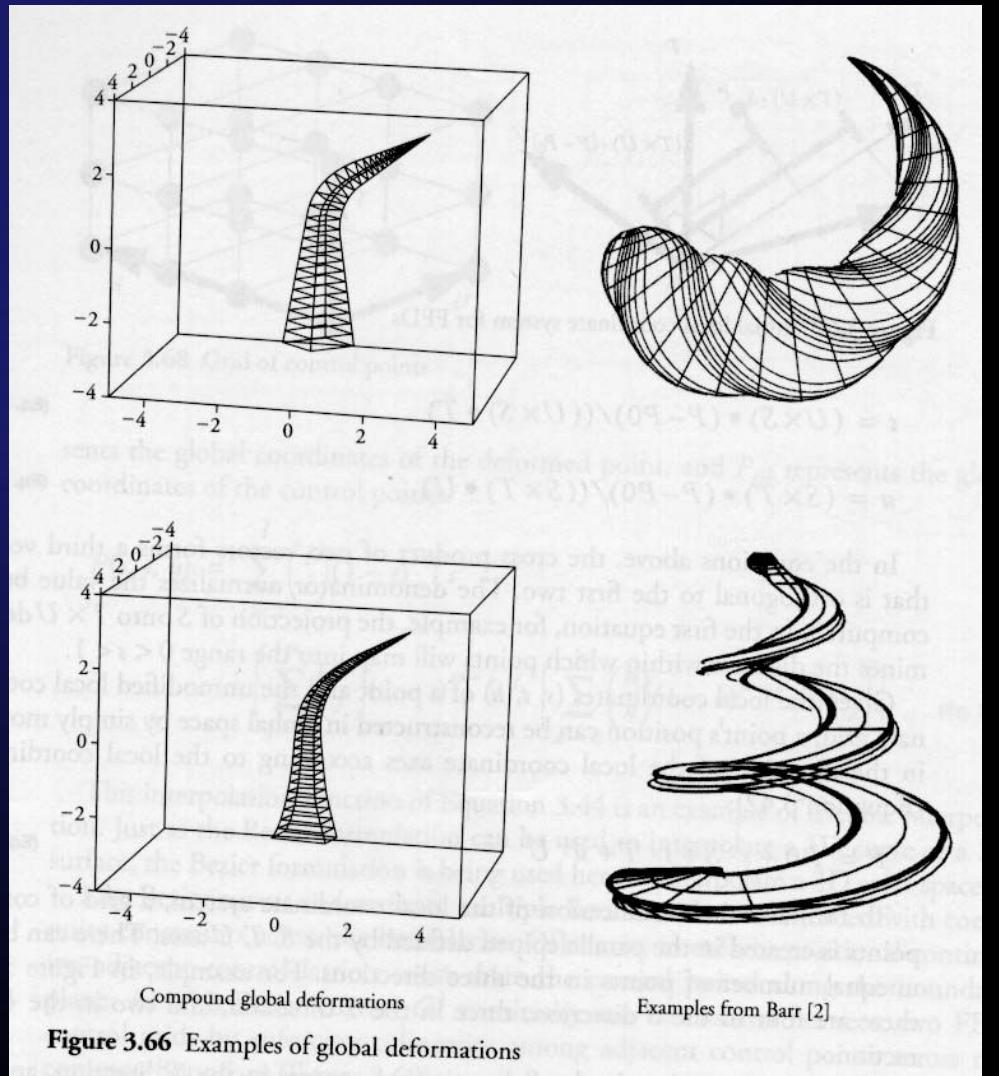
Global Deformations: Generalization of Basic 3D Transformation Matrix

- Alan Barr, SIGGRAPH '84.
- A 3×3 transformation matrix affects all vertices.
- Matrix entries are functions: $(x, y, z) = F(x, y, z)$.
- Obtain taper, twist, bend...
- E.g. taper:
 $(x, y, z) = (zx, zy, z)$ or $(x, y, z) = (rx, ry, z)$ for $r=f(z)$, where **z** is the tapering axis and **f** is a linear or non-linear function.



Twist Global Deformation

- Global axial twisting is just scaling the rotation angle by some function (along an axis).
- Recall that the 2D rotation around the **z** axis is just:
$$(x, y, z) = (x \cos\theta - y \sin\theta, x \sin\theta + y \cos\theta, z)$$
- Changing fixed θ to $\theta = f(z)$ will twist the object being deformed.
- $f(z)$ is the rate of twist per unit length along the **z** axis.



Bend Global Deformation

- Global linear bend along an axis (e.g., the y axis).
- Composite transformation consisting of 3 regions:
 - A region for $y \leq y_{\min}$; here the transformation is a rotation and a translation.
 - A bent region for $y_{\min} < y < y_{\max}$ with radius of curvature $1/k$ and the center of the bend at $y=y_0$
 - A region for $y_{\max} \geq y$; here the transformation is a rotation and a translation, opposite that of the other end.
- The bending angle is $\varnothing = k (y' - y_0)$ where:
 - $y' = y_{\min}$ if $y \leq y_{\min}$
 - $= y$ if $y_{\min} < y < y_{\max}$
 - $= y_{\max}$ if $y \geq y_{\max}$

Bend Global Deformation

- The bend deformation transformation (along y) is thus:

$$x' = x$$

$$y' = -\sin\theta (z - 1/k) + y_0 \quad y_{\min} \leq y \leq y_{\max}$$

$$= -\sin\theta (z - 1/k) + y_0 + \cos\theta (y - y_{\min}) \quad y < y_{\min}$$

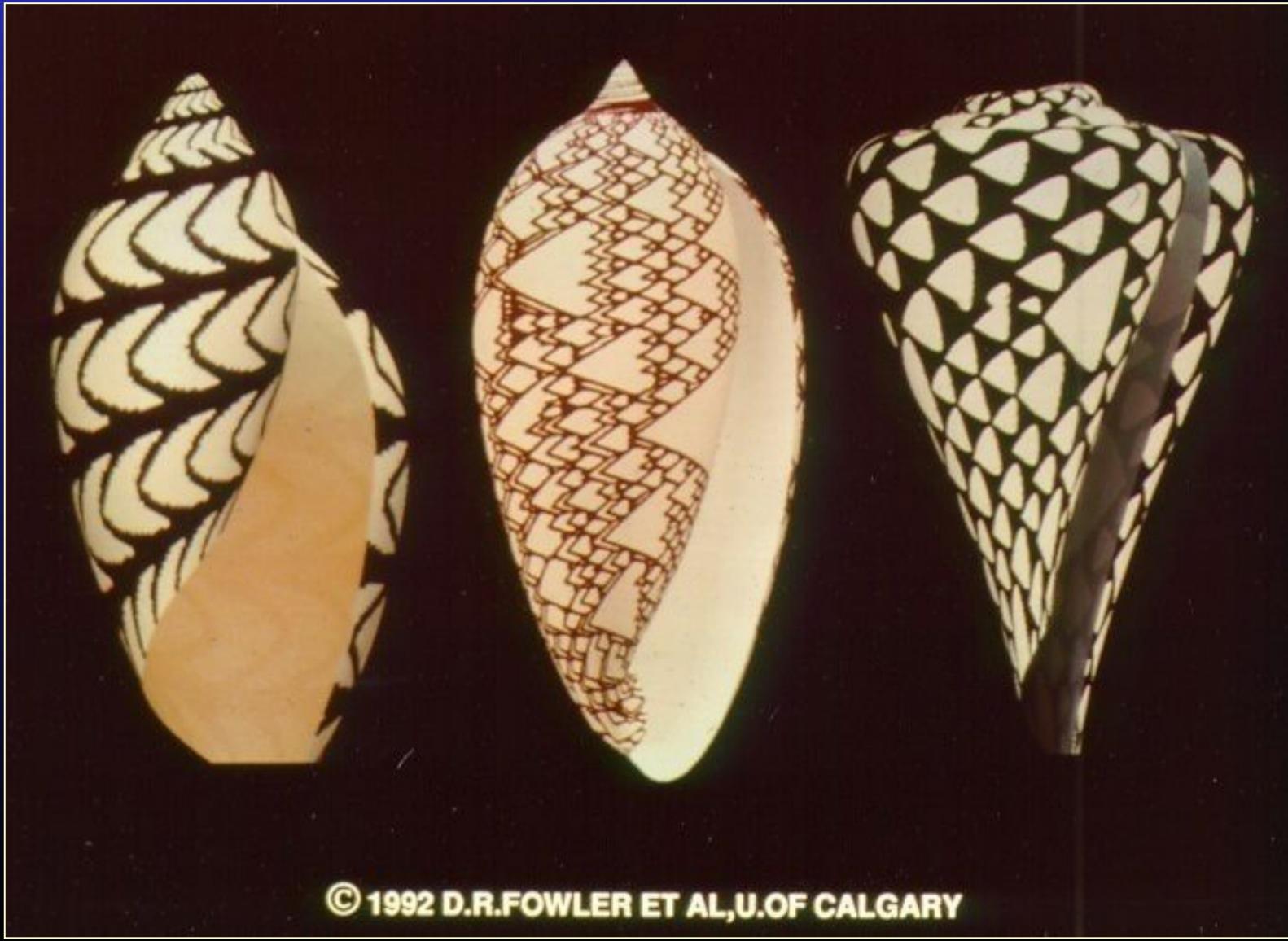
$$= -\sin\theta (z - 1/k) + y_0 + \cos\theta (y - y_{\max}) \quad y > y_{\max}$$

$$z' = \cos\theta (z - 1/k) + 1/k \quad y_{\min} \leq y \leq y_{\max}$$

$$= \cos\theta (z - 1/k) + 1/k + \sin\theta (y - y_{\min}) \quad y < y_{\min}$$

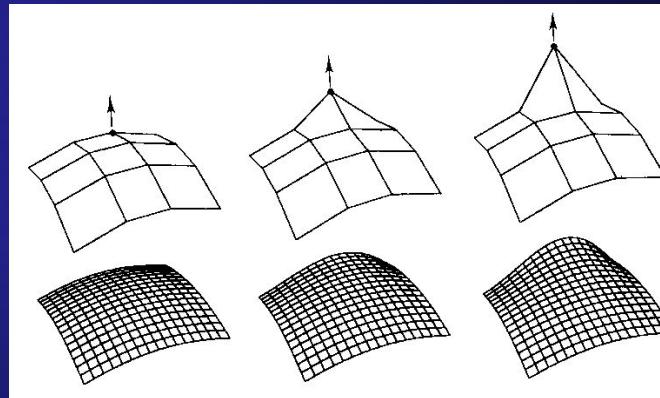
$$= \cos\theta (z - 1/k) + 1/k + \sin\theta (y - y_{\max}) \quad y > y_{\max}$$

L-System Seashells Plus Global Deformations



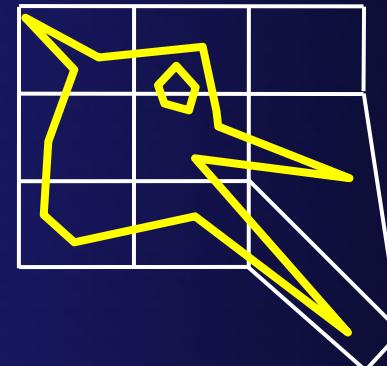
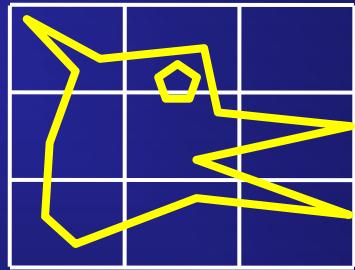
© 1992 D.R.FOWLER ET AL, U.OF CALGARY

Recall the Bezier Surface Wireframe Example



- Note that the surface mesh of lines is deformed by the curve (obviously!) since the curve's math is the defining shape.
- We can exploit this feature by **embedding** any 2D shape (e.g., polygons!) into the curve's coordinate system.
- This is the key idea behind **Free-Form Deformation** (FFD).

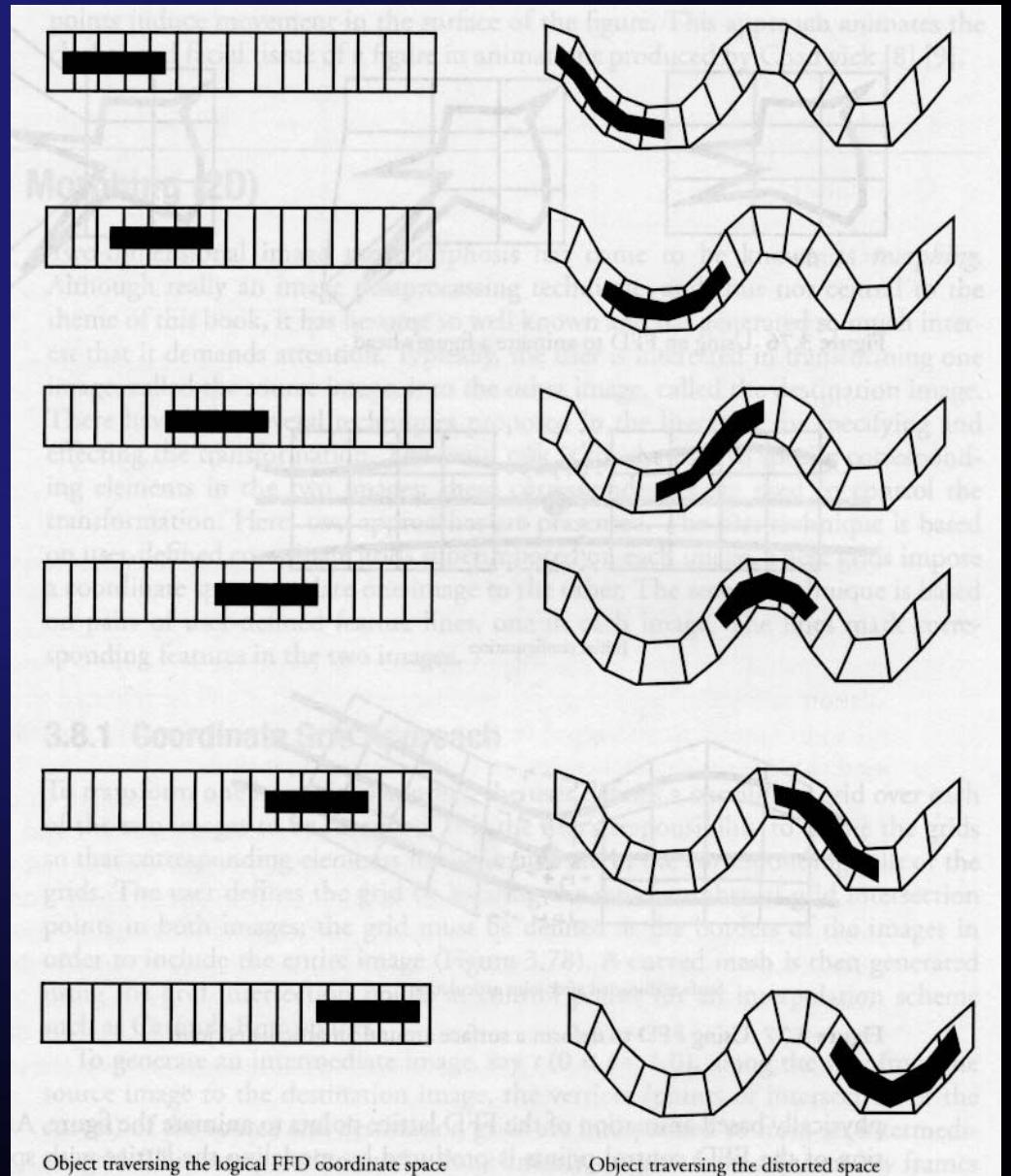
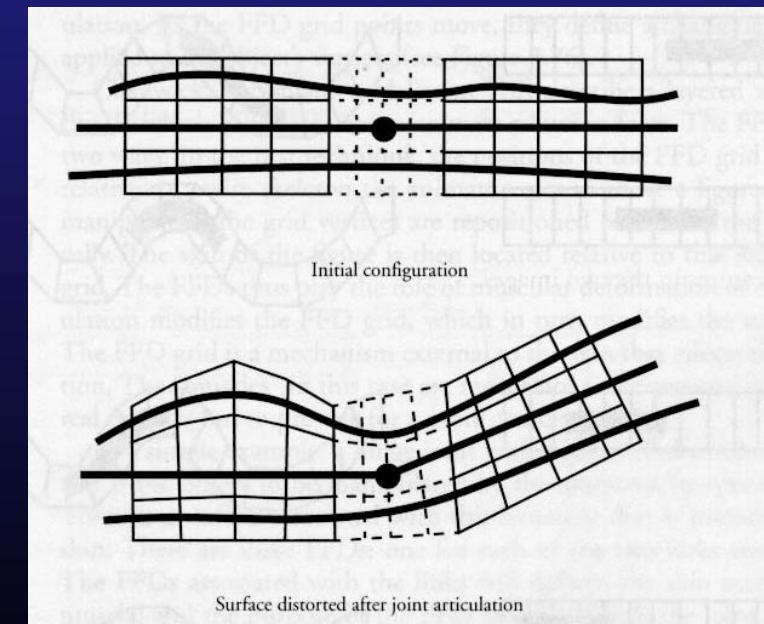
2D Free-Form Deformation



- Note the 4×4 Bezier control mesh (control points).
- Note the embedded polygon vertices.
- Note that changing the Bezier control points then changes the polygon vertices, RATHER THAN creating a curved surface.
- Note that this is NOT a “rubbersheet” kind of deformation, since ONLY the polygon vertex coordinates are updated!

FFD Animation

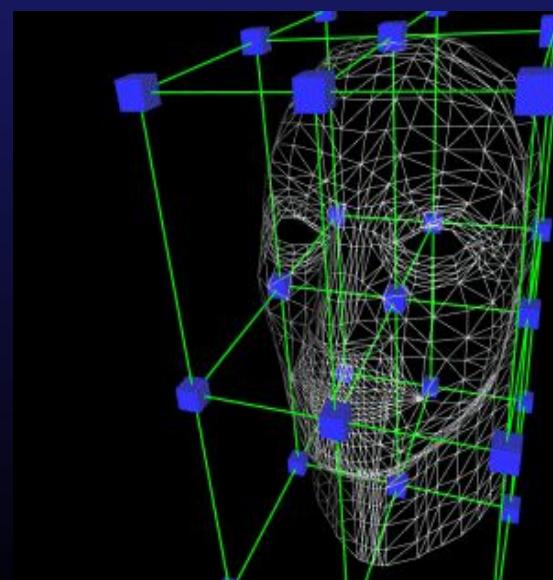
- Moving the 3D control points displaces the original vertices of the object.



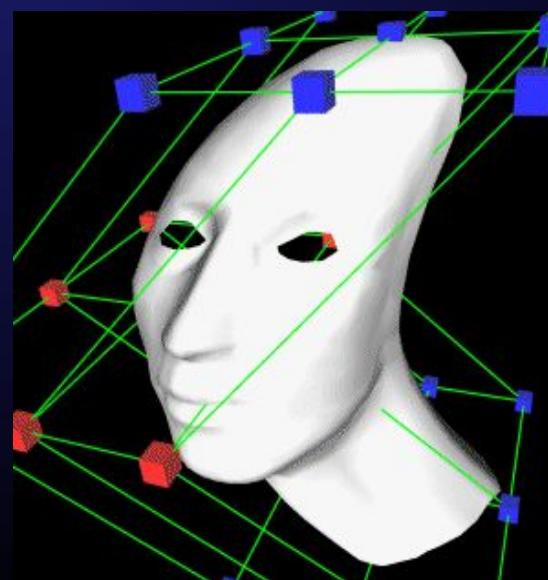
3D Free-Form Deformations (FFD)

- Might as well define this for 3D shapes.
- Sederberg and Parry, SIGGRAPH '86.
- Embed geometric object in local coordinate space (usually cubical but not necessarily).
- Deform local coordinate space and thus deform geometry:

Before

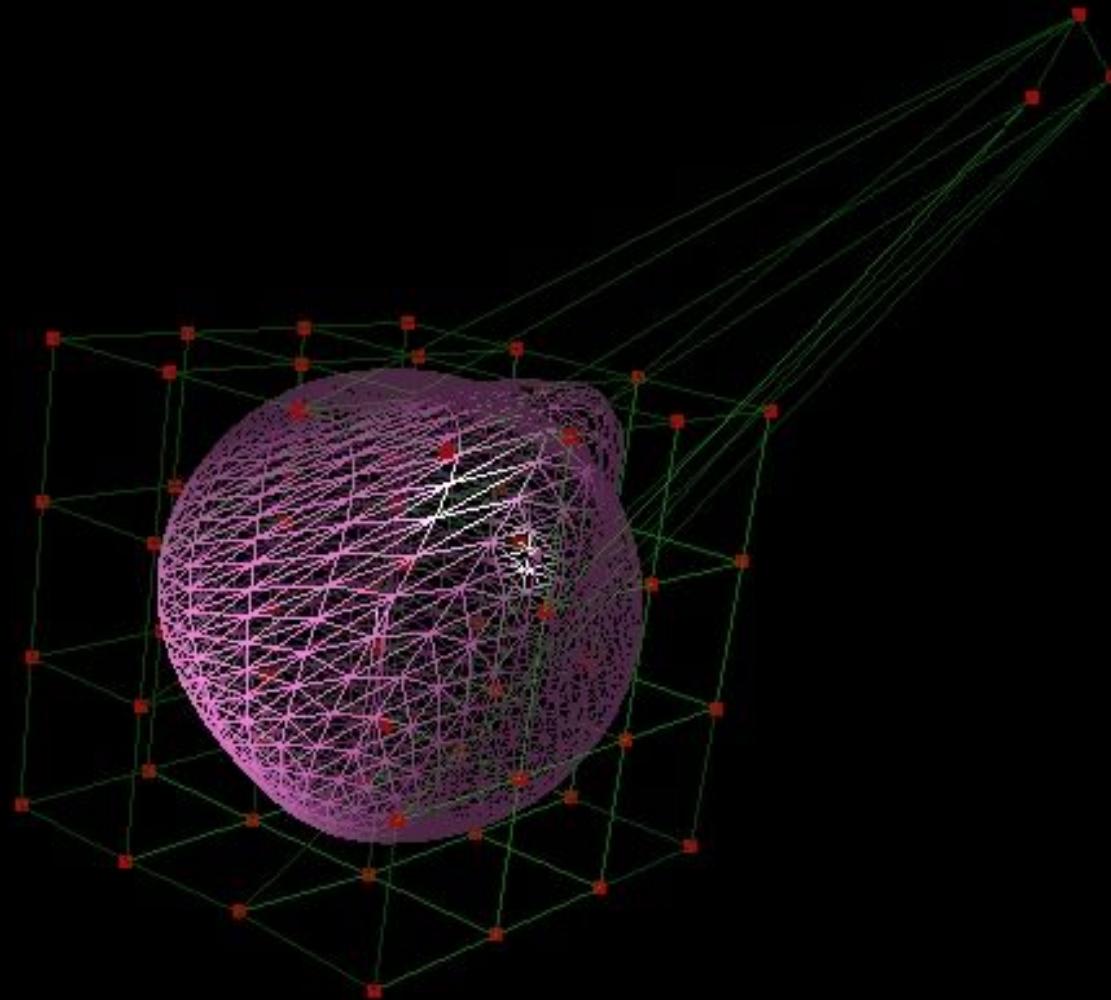


After



D. Brogan,
U. Va.

Move Control Points to Move Embedded Mesh



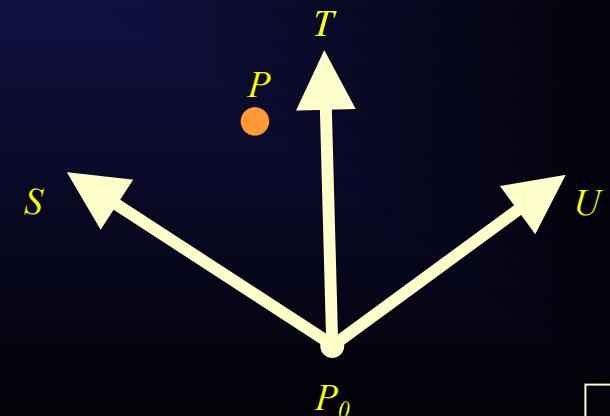
Tri-linear Interpolation (Axis-Aligned Bounding Box Lattice)

- Compute the 3D Axis-Aligned Bounding Box L for the polygonal mesh object M .
- Let P_0 be that corner of L having minimum x , y , and z values. P_0 is the “new” origin of the space L .
- Let S , T , and U define local coordinate axes of L . In this case: $S \parallel x\text{-axis}$; $T \parallel y\text{-axis}$; $U \parallel z\text{-axis}$.
- The lengths of S , T , and U are the side lengths of the bounding box (not unit vectors).
- A mesh vertex P when in this deformation space has “new” coordinates s , t , u :

$$s = (x(P) - x(P_0)) / \|S\|$$

$$t = (y(P) - y(P_0)) / \|T\|$$

$$u = (z(P) - z(P_0)) / \|U\|$$



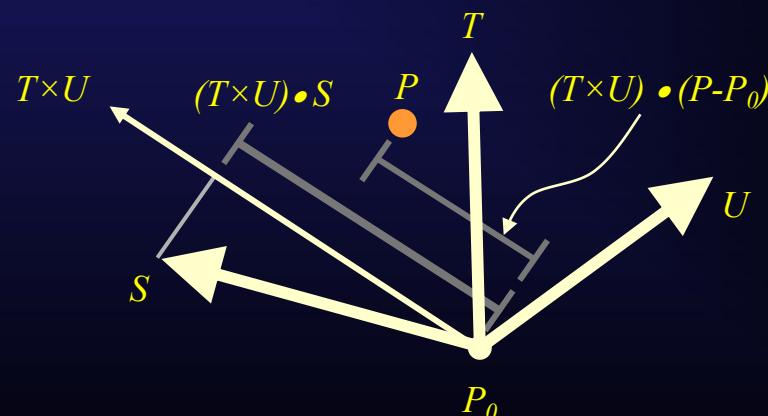
Tri-linear Interpolation (General Case)

- Embed mesh M inside a selected 3D volume L , where
- S , T , and U (with “new” origin P_0) define local coordinate axes of L . [S , T , U do not need to be orthogonal.]
- A mesh vertex P when in this deformation space has “new” coordinates s , t , u :

$$s = \frac{(T \times U) \cdot (P - P_0)}{(T \times U) \cdot S}$$

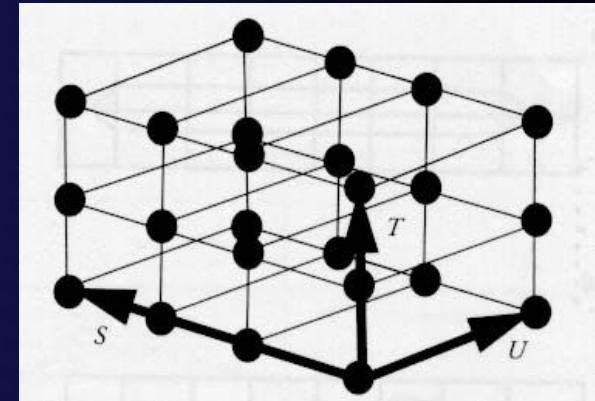
$$t = \frac{(U \times S) \cdot (P - P_0)}{(U \times S) \cdot T}$$

$$u = \frac{(S \times T) \cdot (P - P_0)}{(S \times T) \cdot U}$$



FFD 3D Control Points and Computation

- Then each of S , T , and U axes are subdivided by (l, m, n) into Bezier control points, respectively, P_{ijk} .
- Position P_{ijk} control points as desired.
- Bezier interpolate control points to define new vertex positions of embedded geometry.



E.g., $l=3, m=2, n=1$

$$P = P_0 + s \cdot S + t \cdot T + u \cdot U$$

$$P_{ijk} = P_0 + \frac{i}{l} \cdot S + \frac{j}{m} \cdot T + \frac{k}{n} \cdot U \quad 0 \leq i \leq l; \quad 0 \leq j \leq m; \quad 0 \leq k \leq n$$

$$P(s, t, u) = \sum_{i=0}^l \binom{l}{i} (1-s)^{l-i} s^i \cdot \left(\sum_{j=0}^m \binom{m}{j} (1-t)^{m-j} t^j \cdot \left(\sum_{k=0}^n \binom{n}{k} (1-u)^{n-k} u^k P_{ijk} \right) \right)$$

Notes on the FFD

- Note that the original mesh points are just scaled to the range [0,1] in (s,t,u) inside the bounding box. For an axis-aligned FFD mesh, $S=(1,0,0)$; $T=(0,1,0)$; $U=(0,0,1)$.
- But note that the P_{ijk} control points are in WORLD (x,y,z) coordinates. Check: $s=t=u=0$ yields $P(0,0,0)=P_{000}$; $s=t=u=1$ yields $P(1,1,1)=P_{111}$. So if we change the (x,y,z) values of any control point the formula will interpolate the *original* mesh points accordingly.
- The (s,t,u) values plugged in are for each mesh vertex in the bounding box scaled space. So if there were a mesh vertex at bounding box coordinates $(s,t,u)=(0,0,0)$, e.g., the formula would compute a value of P_{000} – that is, the original mesh point will be exactly at control point P_{000} – wherever it was moved in space.
- Try FFD deforming a cube. The 8 cube vertices – since they fall exactly at the corners of the bounding box – should transform exactly to new positions of the 8 corner control points as you move them. That's your sanity check.

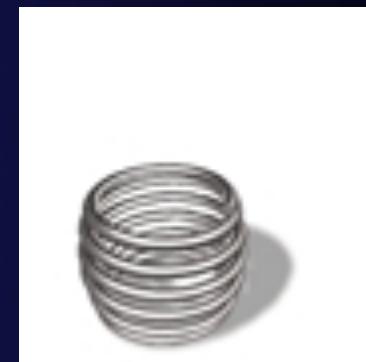
Examples



+



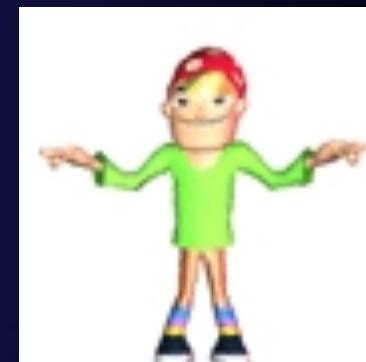
=



+



=



+



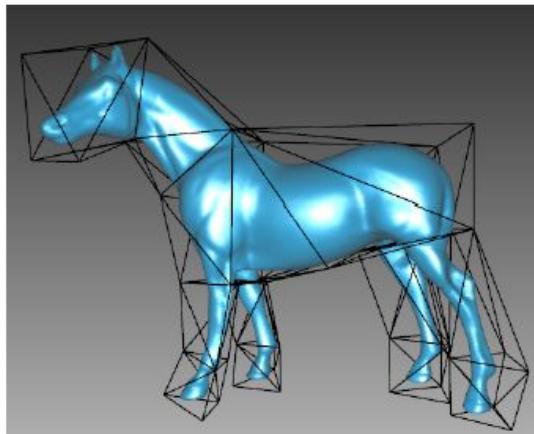
=



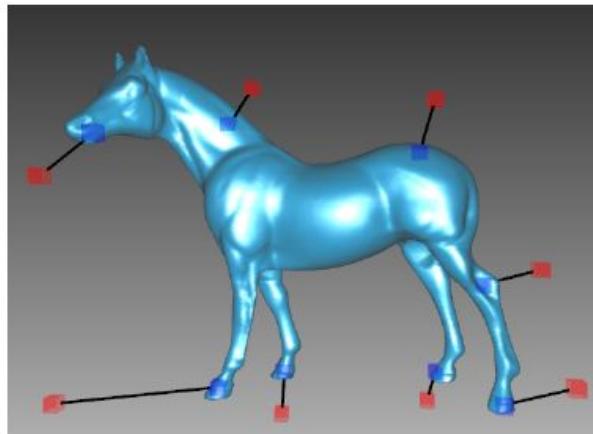
Extension to General Deformation Methods

- Surround model with a meshed deformation *cage*.

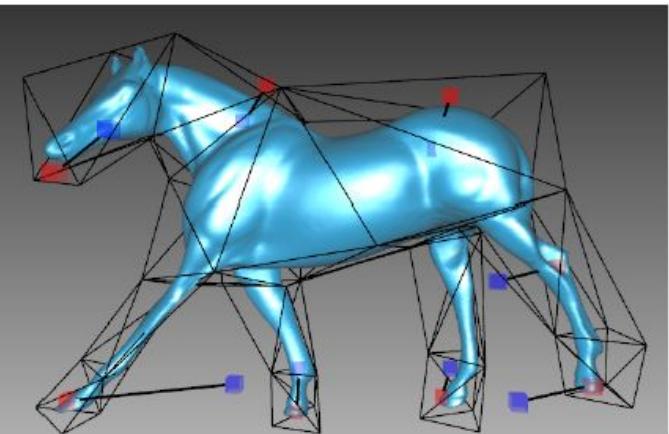
Cage+Model



Deformation handles



Resulting model



Outline

- Great Ideas ✓
- Virtual Environments & Devices ✓
- 3D Perception ✓
- Architecture, Color & Rasterization ✓
- Vector Geometry & Transformations ✓
- 3D Modeling ✓
- Embedding, Hierarchies & Contours ✓
- Model Generation & Deformation ✓
- Visible Surface Algorithms
- Polygon Algorithms
- Image Synthesis & Shading
- Mapping
- 3D Viewing Transformations
- Anti-aliasing & Compositing
- Global Illumination
- Light Fields & HDR
- Path Tracing/Photon Mapping
- Futures

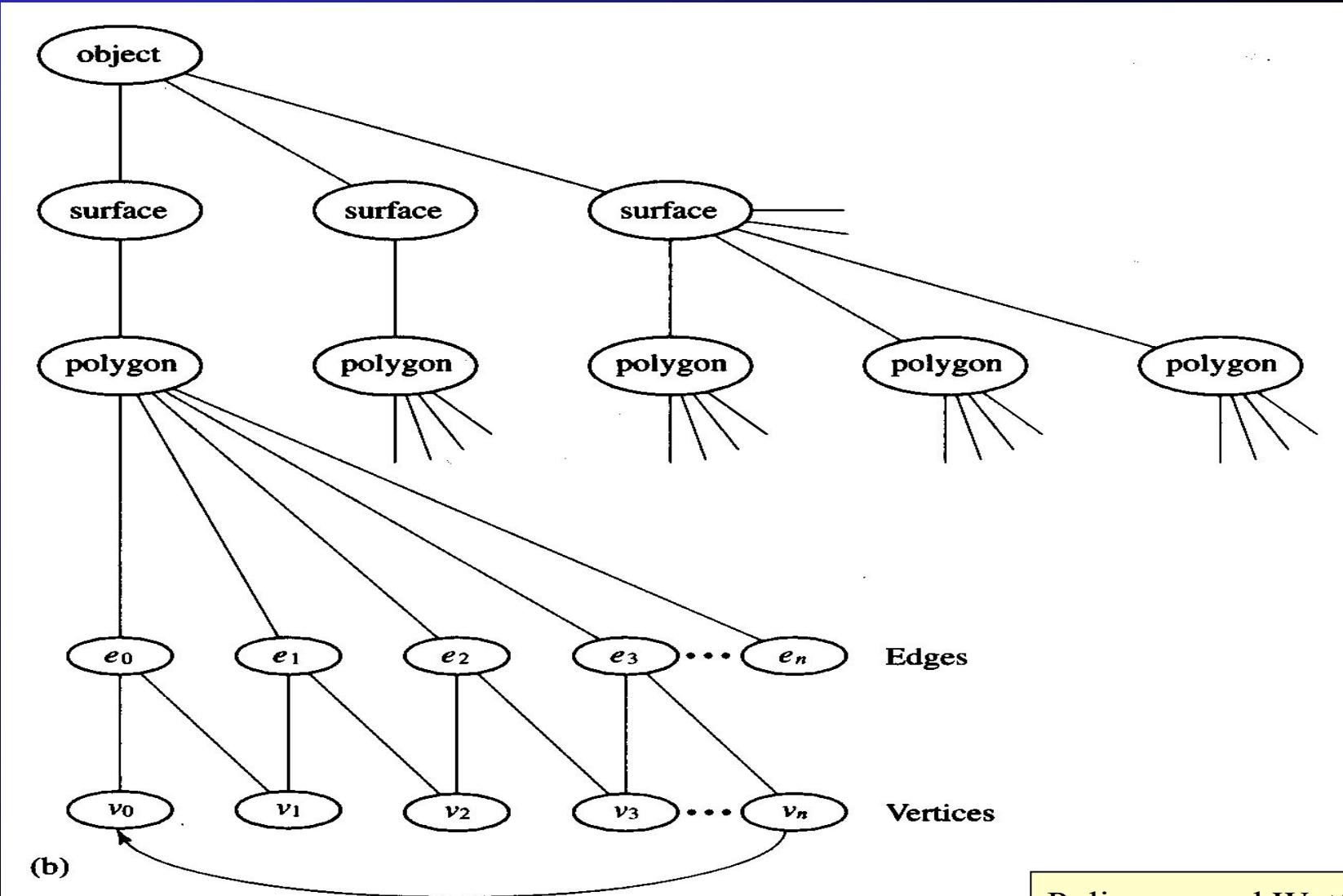
Visible Surface Algorithms

- For polygon meshes
 - Depth buffer
 - Scanline methods
 - Area methods
- For curved surfaces
- Ray casting
 - Triangles
 - Polygons
 - Meshes
 - CSG
 - Implicit surfaces
- Pre-visibility culling
- Binary space partitioning

Polygonal Mesh Model Assumptions (we will make) (for visible surface algorithms)

- Algorithms depend on 3D modeling representation.
- First known algorithm: Larry Roberts, 1963: Sets of convex, planar-faced objects.
- Start with polygonal meshes and cover more general model representations later.
- Clip geometry to view volume.
- Planar polygon faces (convex or concave).
- Consistent edge traversal order -- to establish uniform notion of **inside** and **outside**.
 - Surface normal points outward in a right-handed world modeling coordinate system.

Polygon Mesh Conceptualization



A Polygon Mesh Model

VERTICES

P1 = (1, 2, 0)

P2 = (1, 2, 3)

P3 = (0, 2, 5)

P4 = (-1, 2, 3)

P5 = (-1, 2, 0)

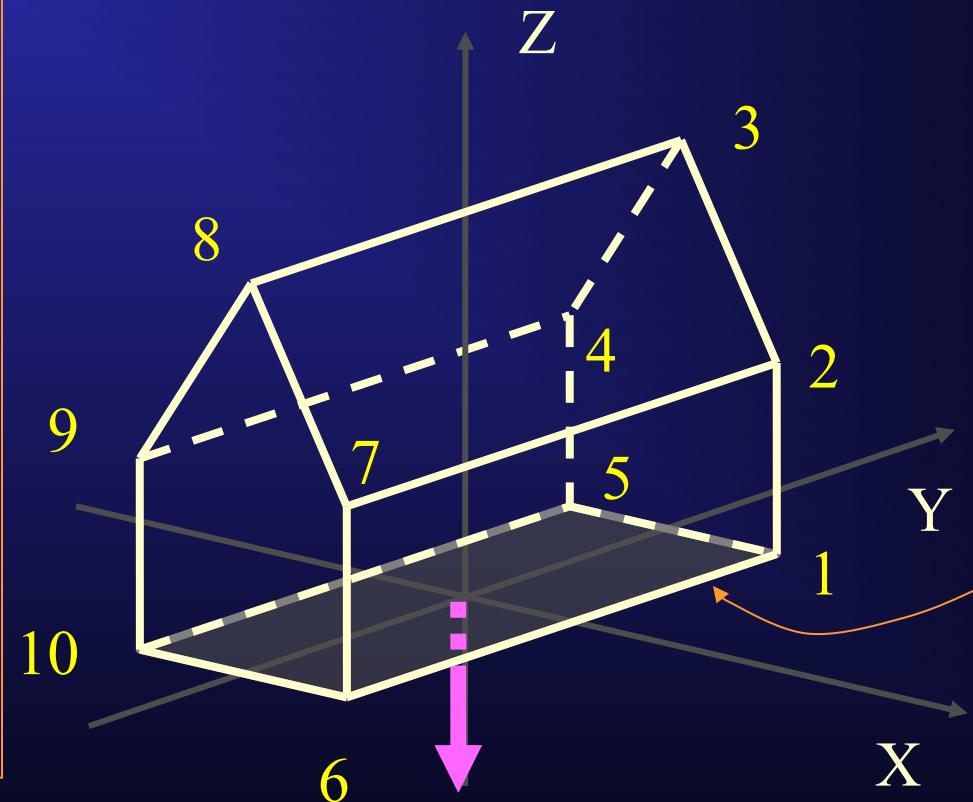
P6 = (1, -2, 0)

P7 = (1, -2, 3)

P8 = (0, -2, 5)

P9 = (-1, -2, 3)

P10 = (-1, -2, 0)



POLYGONS

P1 P5 P4 P3 P2

P6 P7 P8 P9 P10

P1 P2 P7 P6

P2 P3 P8 P7

P3 P4 P9 P8

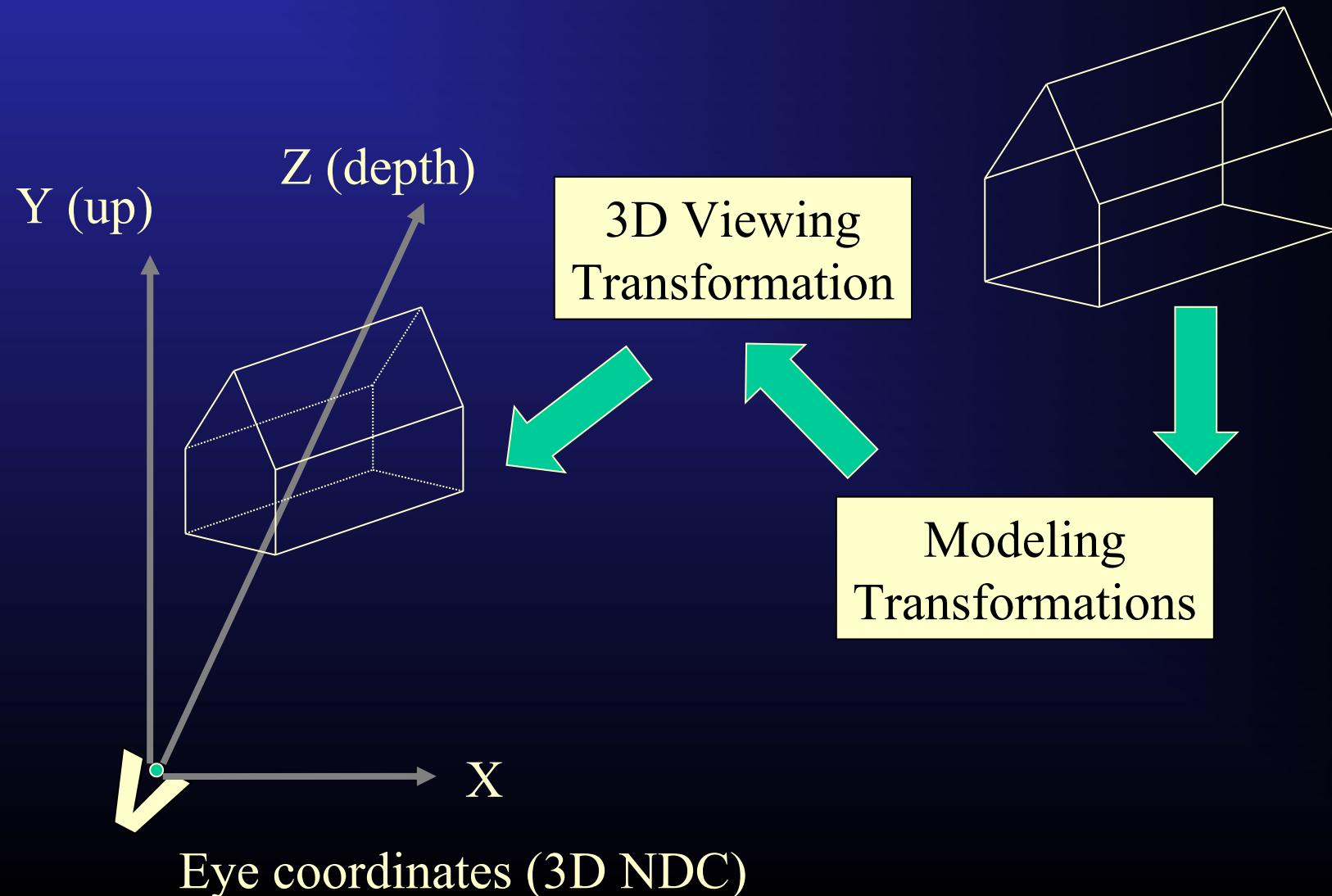
P4 P5 P10 P9

P1 P6 P10 P5

ATTRIBUTES:

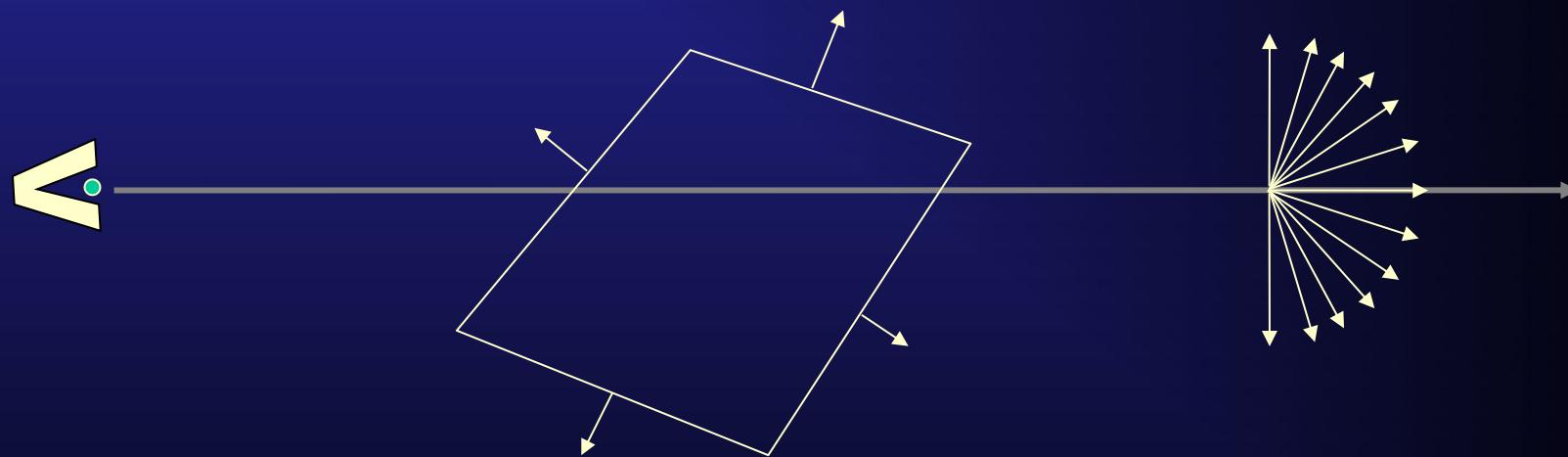
name = 'floor', normal = (0, 0, -1), color = (R=0.1, G=0.1, B=0.1),
fill = yes, edge-color = (R=1, G=1, B=1), ...

The Model Undergoes Other Transformations as Needed for Positioning, Scaling, and Viewing



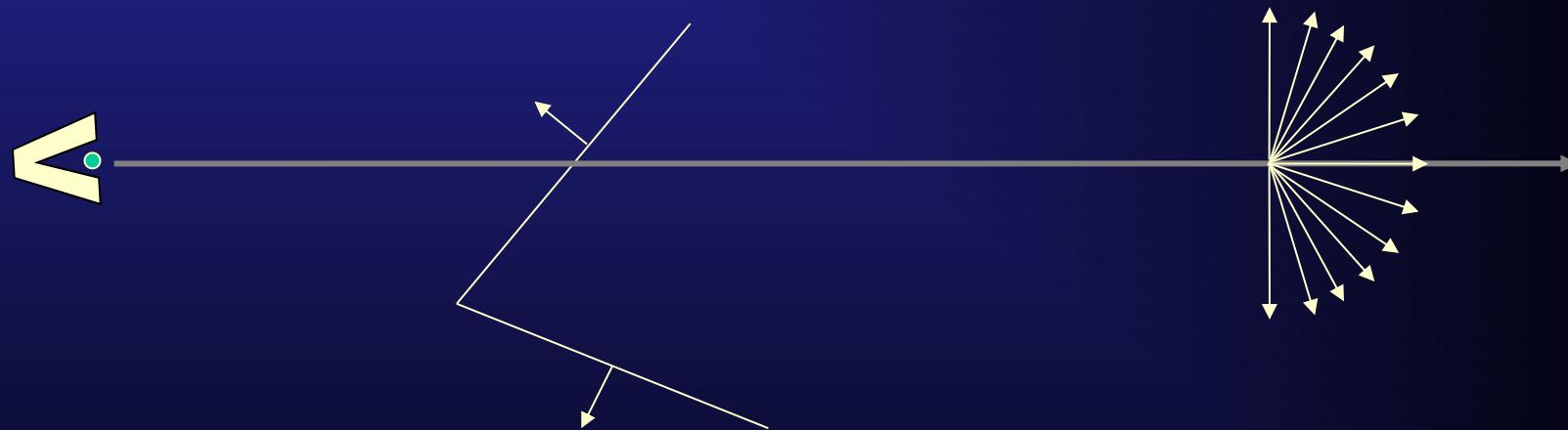
Back Face Cull

- Throw out polygons facing away from eye -- that is, any polygon with a **BACK-facing normal**:



Back Face Cull

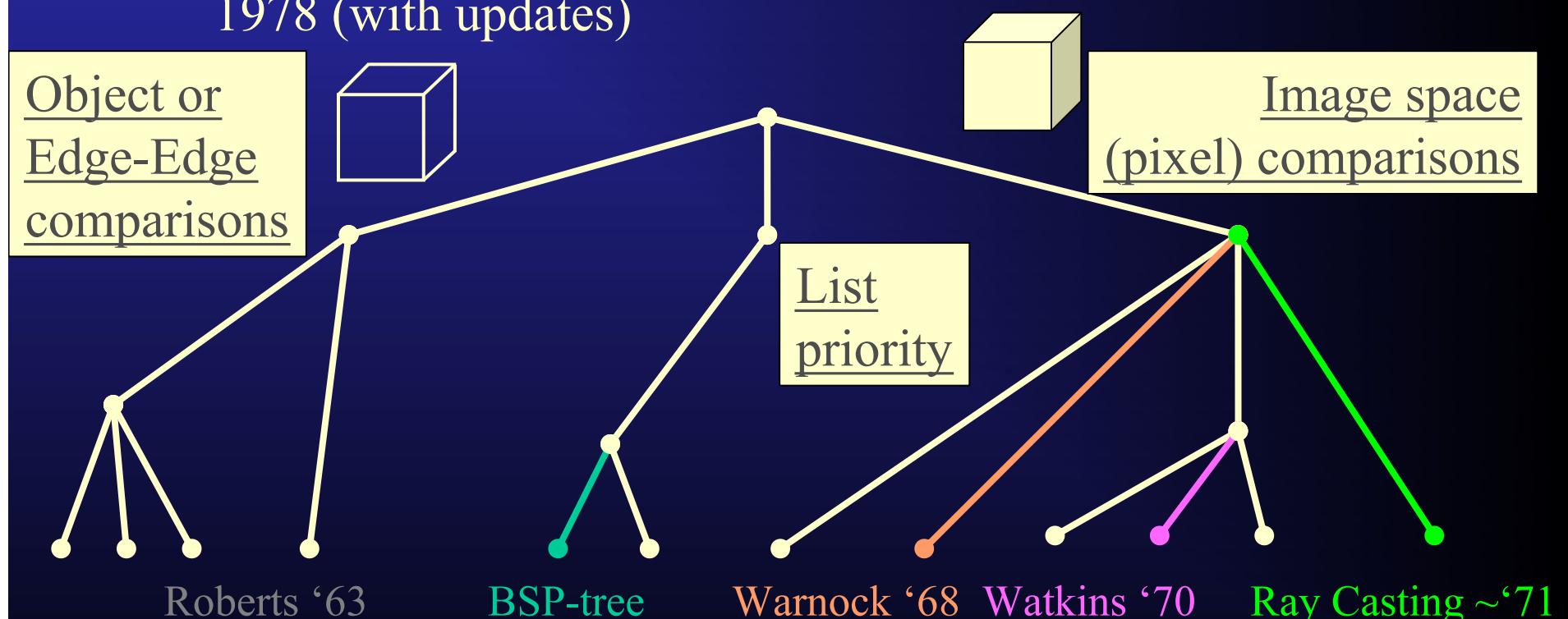
- Only FRONT-facing ones left to process further for potential visibility.



(Note that for more advanced rendering methods, these back faces might still be needed, useful, or visible in reflections.)

Visible Surface Algorithm Taxonomy

Sutherland, Sproull, Schumacker, ACM Computing Surveys,
1978 (with updates)

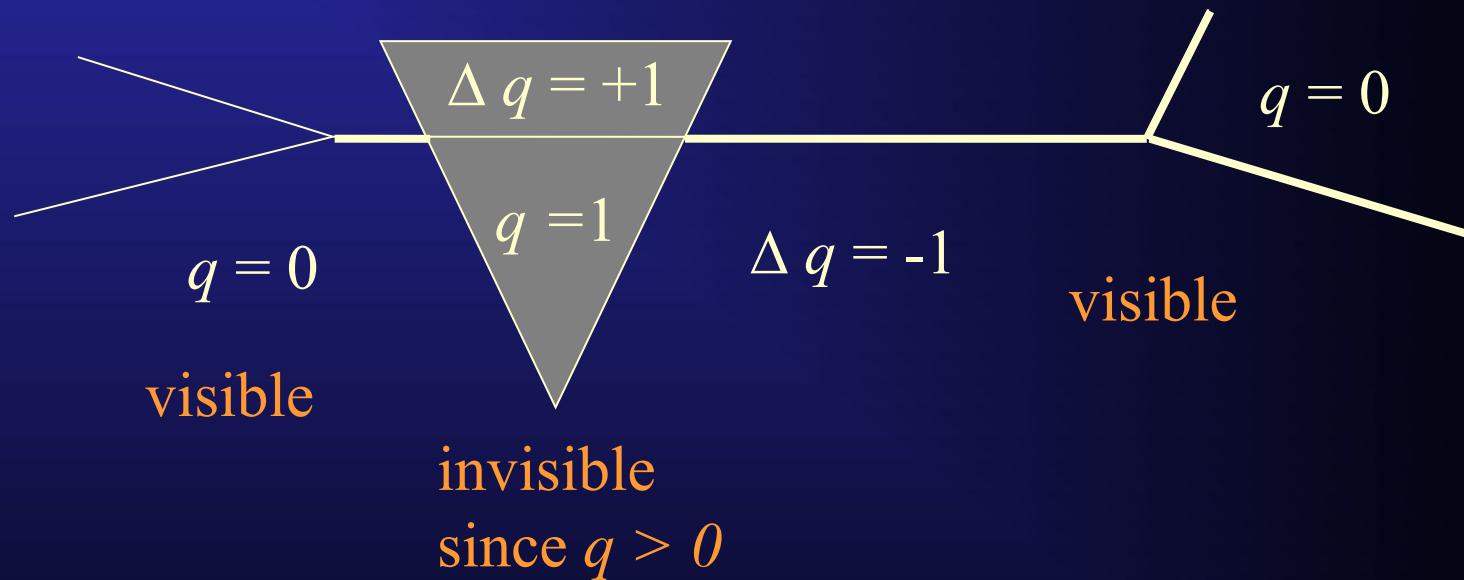


Complexity grows $O(n^2)$
(n =number of objects)

Complexity \sim visual complexity
Bounded by sorting cost $O(n \log n)$

Appel 1967 -- Visible LINE Algorithm

Traverse entire edge network to establish *Quantitative Invisibility* (q)



- Must start from known visible vertex, e.g., choose vertex closest to eye (lowest z).
- Suffers from vertex crossing (degenerate case) problems!

Visible Surface Algorithms for Polygon Meshes

- Depth or Z-buffer
- Screen space algorithms
- Scan-line methods
- Area methods
- Painter's algorithm
- “Exact” algorithm

Depth or Z-Buffer

- Each element corresponds to one pixel.
- Each element stores COLOR and DEPTH.
- Algorithm:

Initialize all elements of buffer (COLOR(row, col),
DEPTH(row, col)) to (background-color, maximum-depth);

for each polygon:

Rasterize polygon to frame;

for each pixel center (x, y) that is covered:

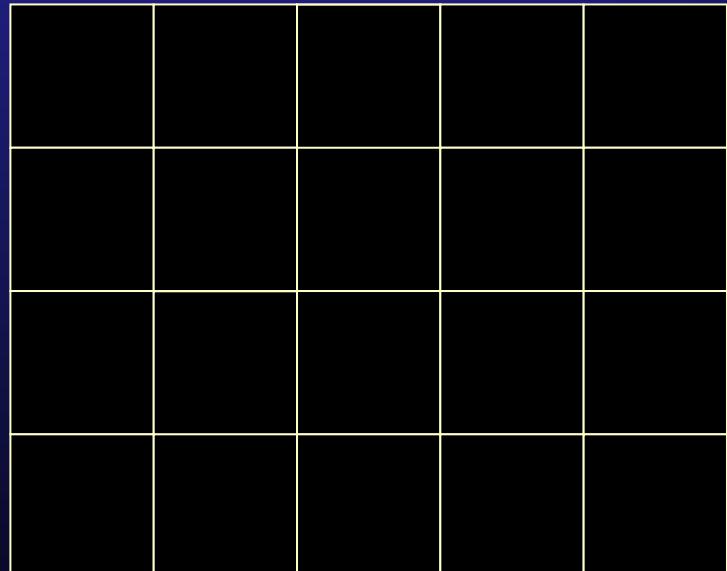
 if polygon depth at $(x, y) < \text{DEPTH}(x, y)$

 then COLOR(x, y) := polygon color at (x, y) ;

 DEPTH(x, y) := polygon depth at (x, y)

Depth Buffer Operation

Frame



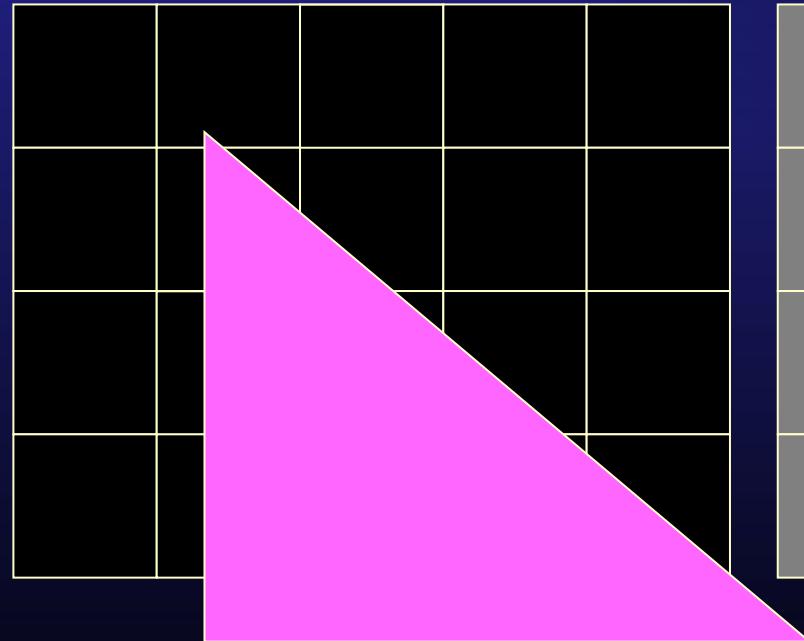
Depth

| | | | | |
|----------|----------|----------|----------|----------|
| ∞ | ∞ | ∞ | ∞ | ∞ |
| ∞ | ∞ | ∞ | ∞ | ∞ |
| ∞ | ∞ | ∞ | ∞ | ∞ |
| ∞ | ∞ | ∞ | ∞ | ∞ |

Initialize (“New Frame”)

Depth Buffer Operation -- First polygon element

Frame



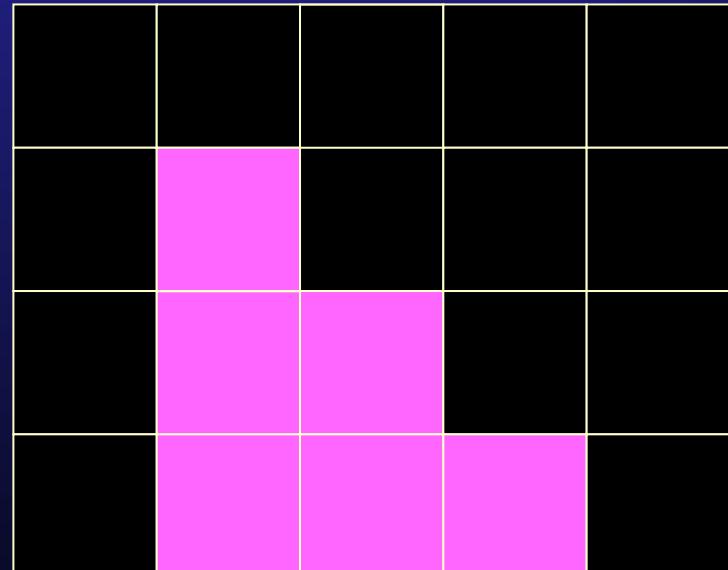
Depth

| | | | | |
|----------|----------|----------|----------|----------|
| ∞ | ∞ | ∞ | ∞ | ∞ |
| ∞ | 24 | ∞ | ∞ | ∞ |
| ∞ | 26 | 27 | ∞ | ∞ |
| ∞ | 28 | 29 | 30 | ∞ |
| | | | | |

Pink Triangle -- depths computed at pixel centers

Depth Buffer Operation -- First polygon element

Frame



Depth

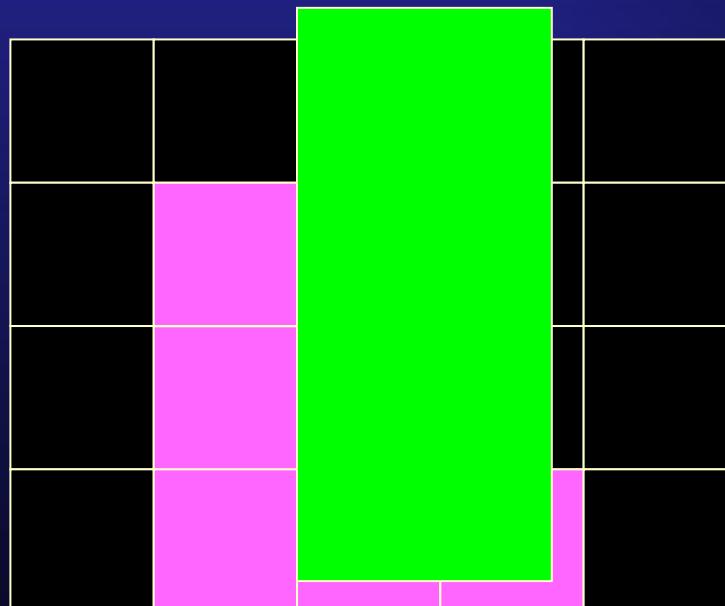
| | | | | |
|----------|----------|----------|----------|----------|
| ∞ | ∞ | ∞ | ∞ | ∞ |
| ∞ | 24 | ∞ | ∞ | ∞ |
| ∞ | 26 | 27 | ∞ | ∞ |
| ∞ | 28 | 29 | 30 | ∞ |
| | | | | |

A yellow triangle highlights the depth values assigned to the pink pixels in the frame buffer. The triangle's vertices are at (1,1), (3,1), and (1,3). The depth values are 24, 26, and 28 respectively.

Pink Triangle -- pixel values assigned

Depth Buffer Operation -- Second polygon element

Frame



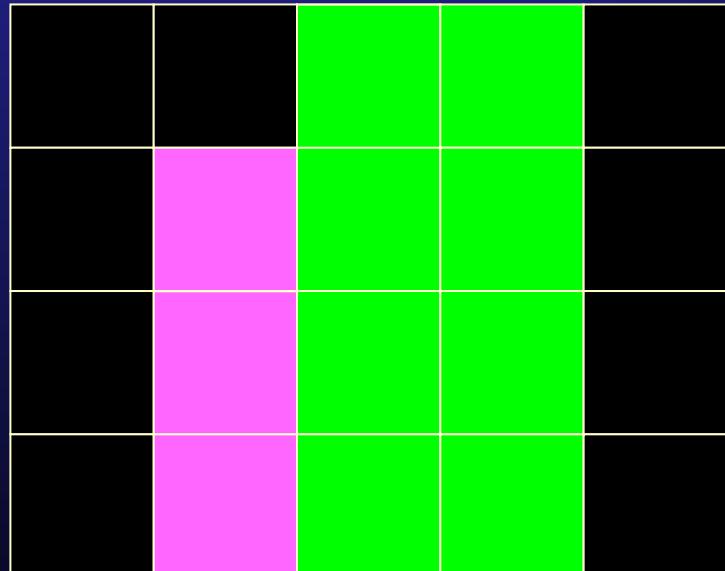
Depth

| | | | | |
|----------|----------|----|---|----------|
| ∞ | ∞ | 10 | 9 | ∞ |
| ∞ | 24 | 9 | 8 | ∞ |
| ∞ | 26 | 8 | 7 | ∞ |
| ∞ | 28 | 7 | 6 | ∞ |

Green Rectangle -- depths computed at pixel centers

Depth Buffer Operation -- Second polygon element

Frame

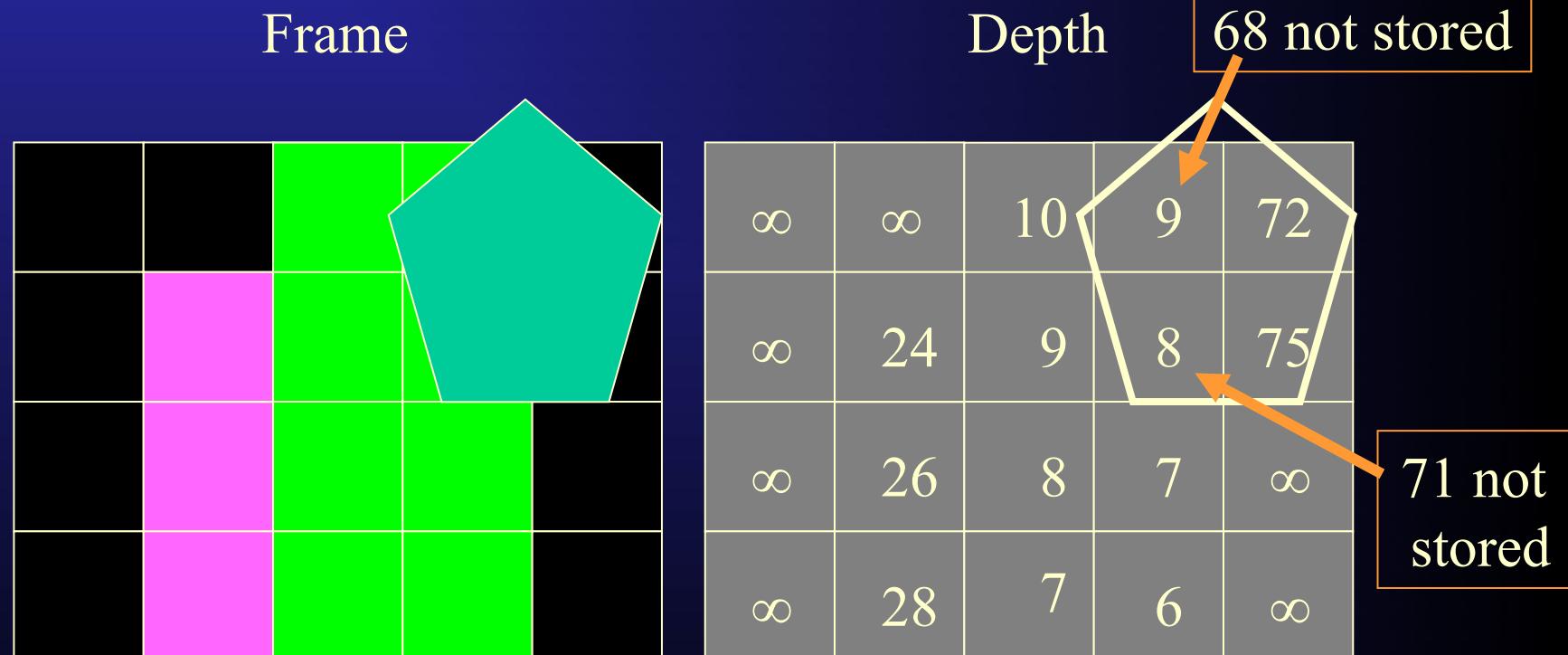


Depth

| | | | | |
|----------|----------|----|---|----------|
| ∞ | ∞ | 10 | 9 | ∞ |
| ∞ | 24 | 9 | 8 | ∞ |
| ∞ | 26 | 8 | 7 | ∞ |
| ∞ | 28 | 7 | 6 | ∞ |

Green Rectangle -- pixel values assigned: NOTE REPLACEMENTS!

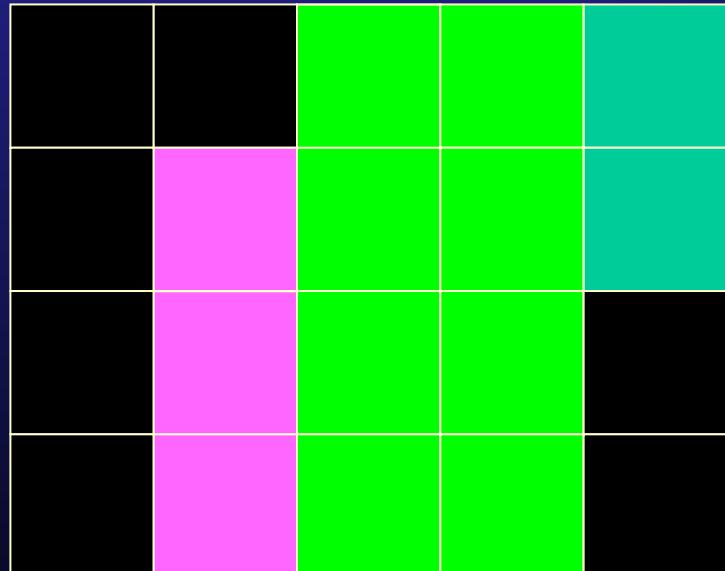
Depth Buffer Operation -- Third polygon element



Blue Pentagon -- depths computed at pixel centers

Depth Buffer Operation -- Third polygon element

Frame

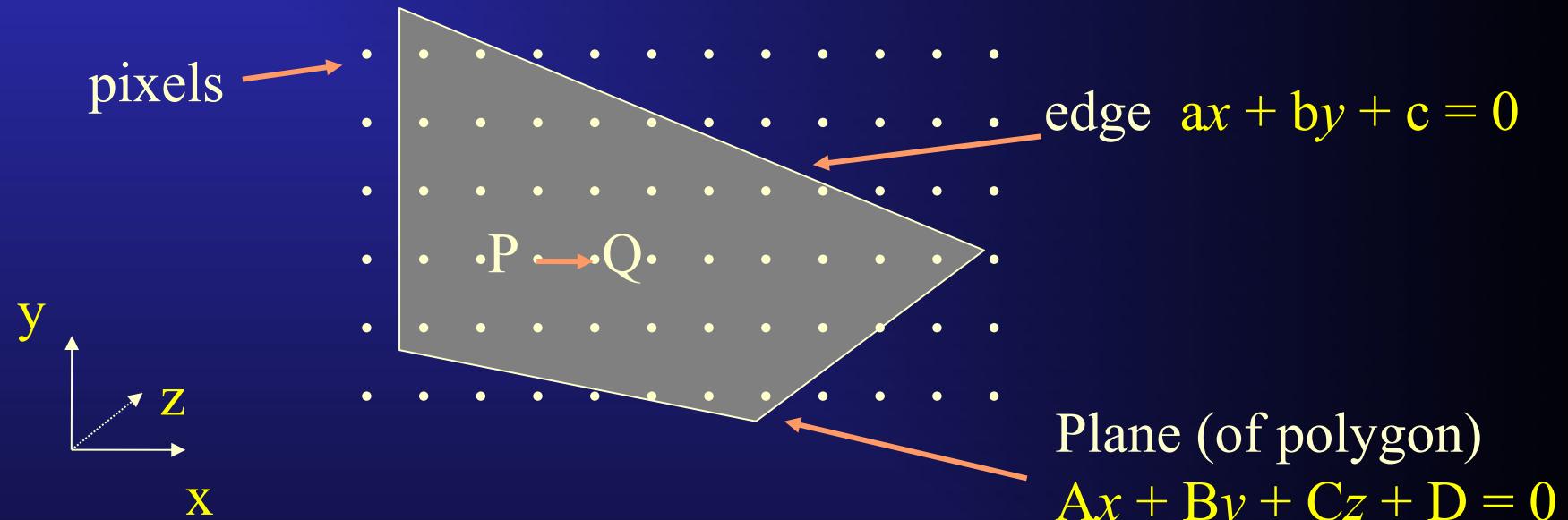


Depth

| | | | | |
|----------|----------|----|---|----------|
| ∞ | ∞ | 10 | 9 | 72 |
| ∞ | 24 | 9 | 8 | 75 |
| ∞ | 26 | 8 | 7 | ∞ |
| ∞ | 28 | 7 | 6 | ∞ |

Blue Pentagon -- pixel values assigned: NOTE 'GOES BEHIND'!

Depths at Pixels can be Computed very Efficiently



Move from P to Q ($x =+ 1$)

Distance from polygon edge: $d =+ a$

Depth: $z =- (A / C)$ [Q-P; solve for z]

Point is inside polygon if $d > 0$ for all edges.

Similarly when moving to next scan line: $y =+ 1$

Depth Buffer Example

Roof-1

Floor

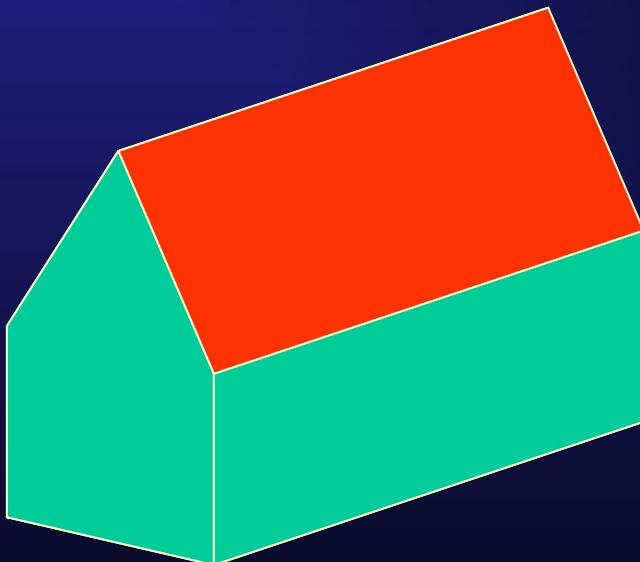
Roof-2

Wall-1

End-2

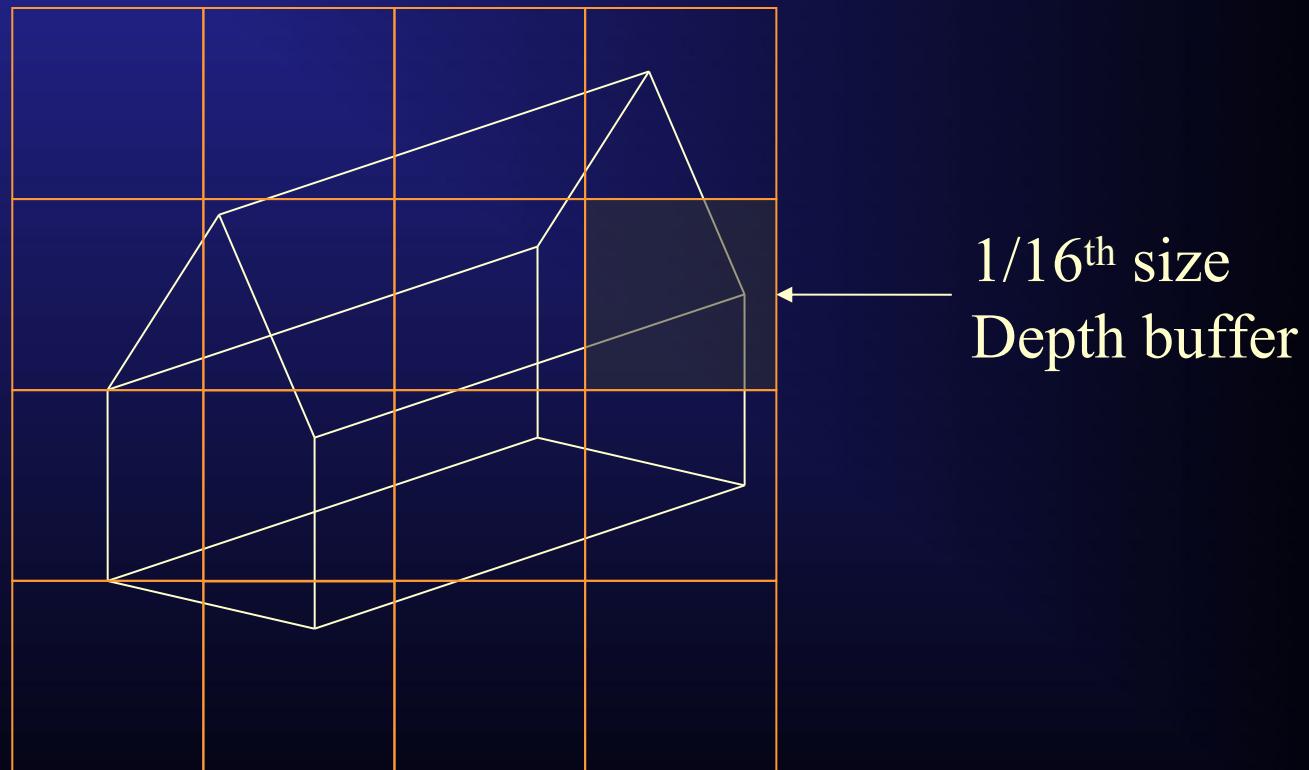
Wall-2

End-1



Static Screen Subdivision

- Use smaller depth buffer and repeat depth buffer algorithm multiple times (e.g. 16) -- process each polygon 16 times.

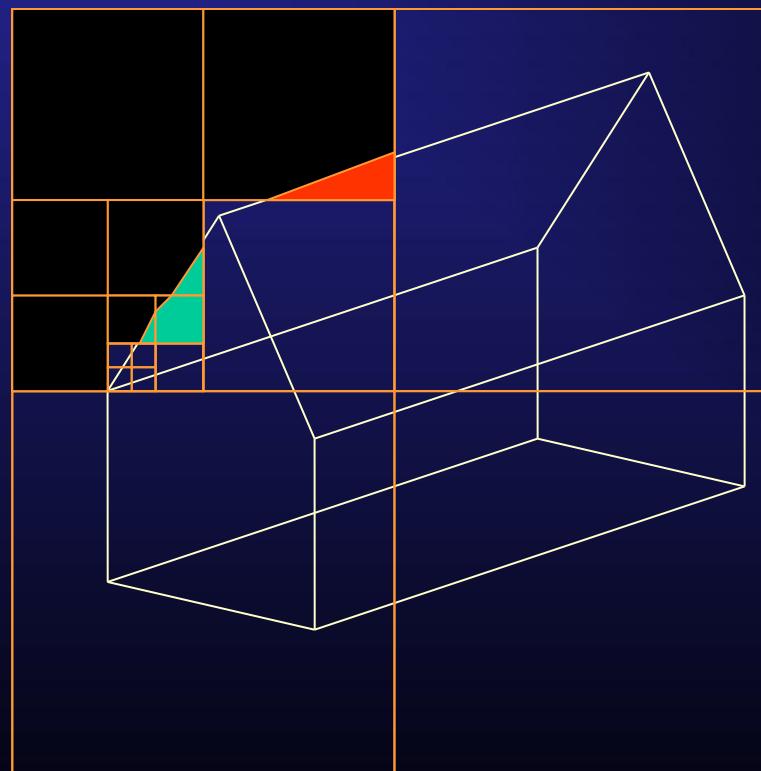


Adaptive Screen Subdivision

- Subdivide frame buffer into smaller chunks in detail areas.
- Implement as a recursive algorithm -- Warnock.
 - If frame area is **simple**, then just draw it.
 - If complex, then subdivide into quadrants and recurse.

simple = {all background;
area covered entirely by one polygon;
area split into two regions by one polygon edge;
if area is a single pixel, find front-most polygon
}

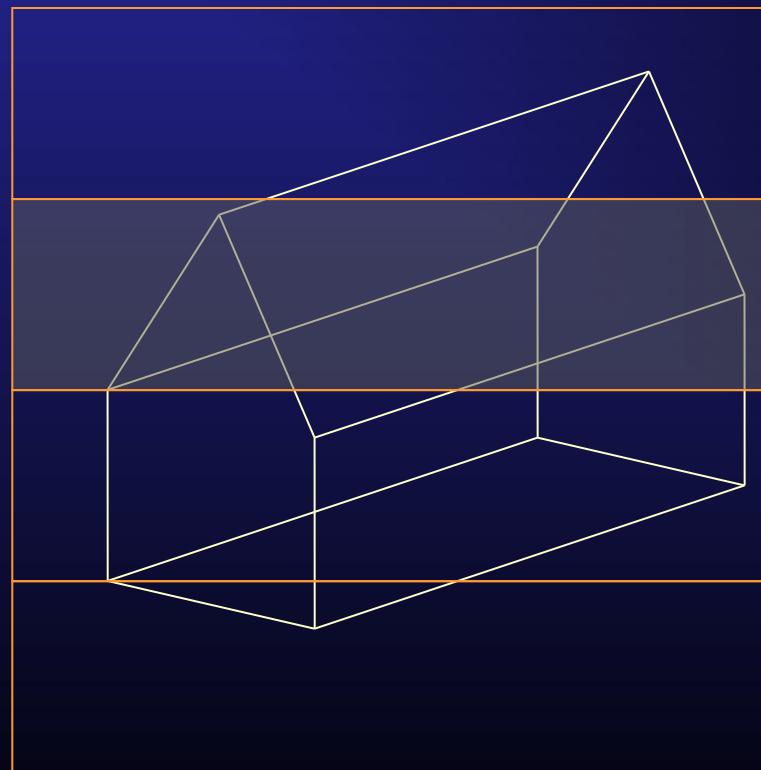
Warnock's Algorithm: Adaptive Screen Subdivision (Quad-Tree Screen Space Decomposition)



Neat, but slow because recursion gets deep at many edges.

Static Screen Subdivision: Strips

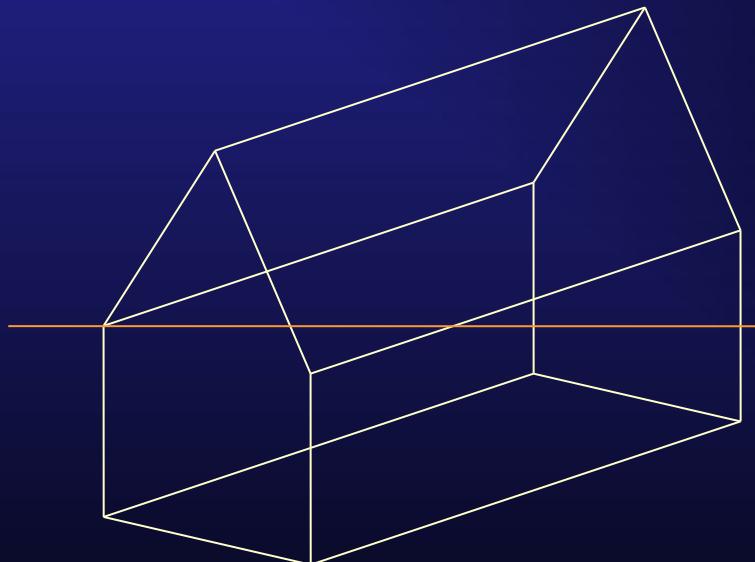
- Use smaller depth buffer consisting of a number of scan lines:



Advantage: image is created in full width strips, top to bottom.

Static Screen Subdivision: Scan-Lines

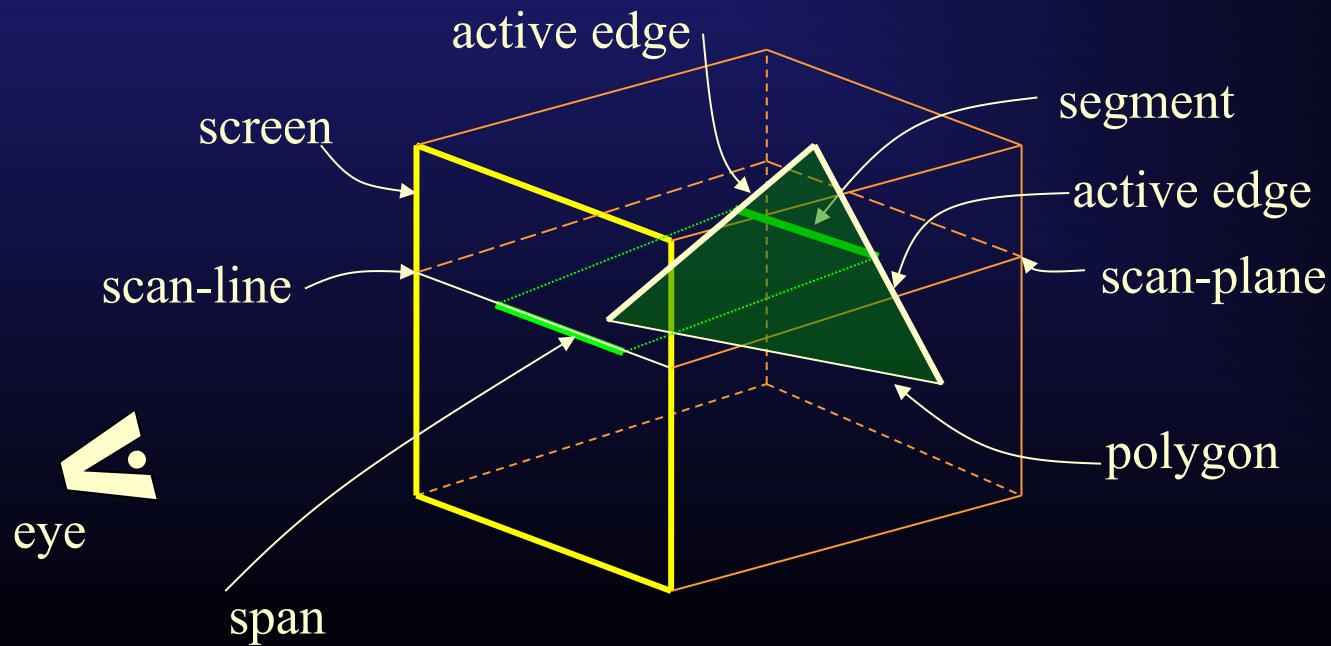
- In the limit, the strip can be a single scan-line.
- BUT: All polygons need be processed for each scan-line!



Watkins came up with a data structure that avoided this overhead.

Scan-Line Algorithm Definitions

- Scan-Plane : The projection of the scan line into the world.
- Edge : Line between two polygon vertices.
- Active Edge : An edge intersected by the scan-plane.
- Segment : Portion of a polygon between two active edges.
- Span : Projection of segment onto scan-line.



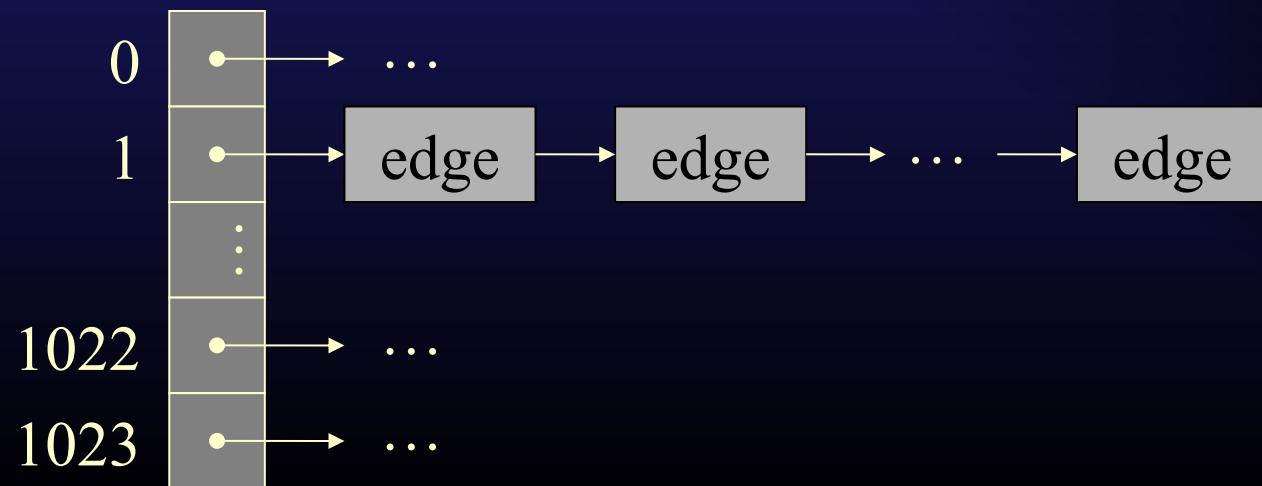
Edge Data Blocks

- Name of parent polygon.
- y_s - scan-line intersected by topmost edge vertex.
- n - number of scan-lines spanned by edge.
- x_s - x coordinate of topmost edge vertex.
- Δx - change in x to get edge to next scan-line.
- z_s - z coordinate of topmost edge vertex.
- Δz - change in z to get edge to next scan-line.
- I_s - beginning edge color.
- ΔI - change in color to next scan-line.
- priority – higher priority wins (is closer) in a depth tie

Scan-Line Algorithm Overview

Basis for general **plane-sweep** algorithms of computational geometry: reduce dimensionality of the problem by one -- make a 3D problem into a 2D problem.

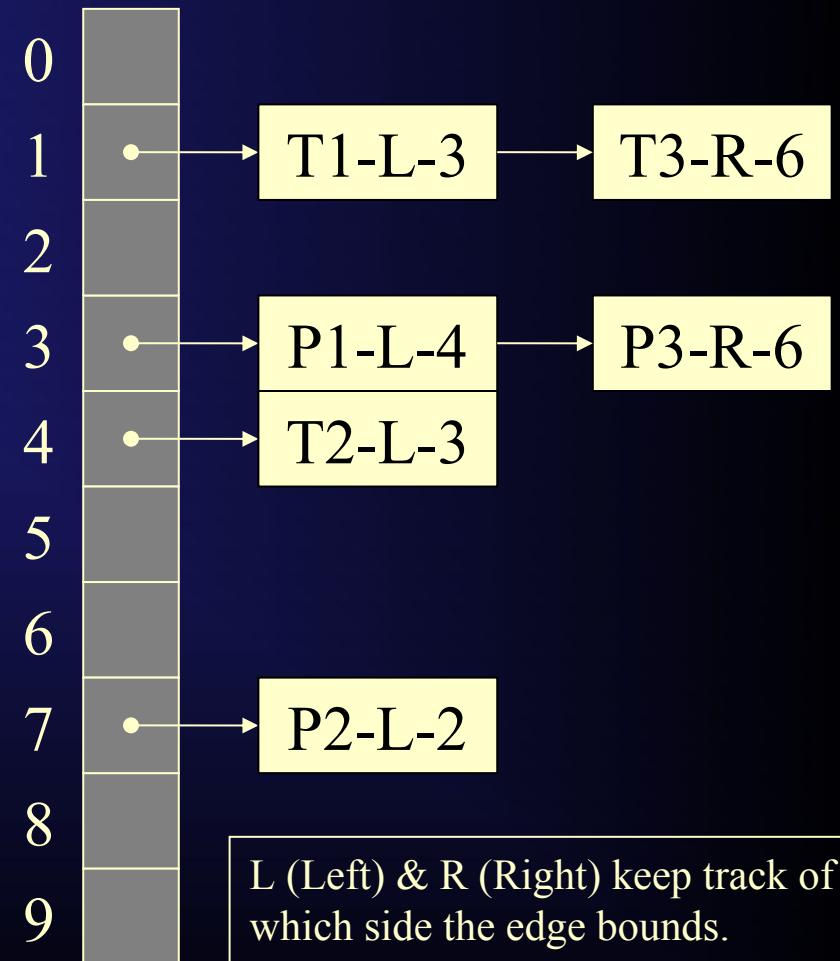
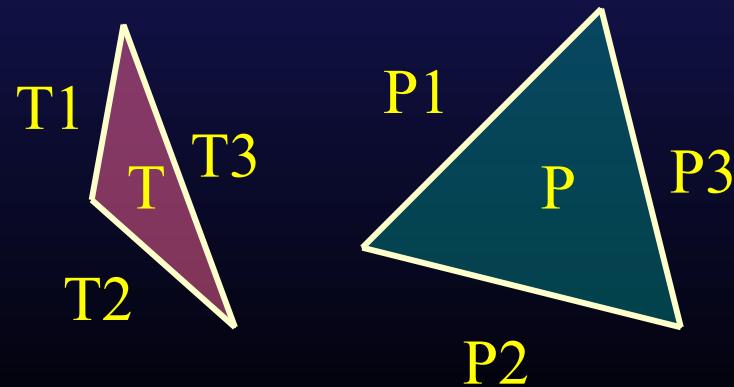
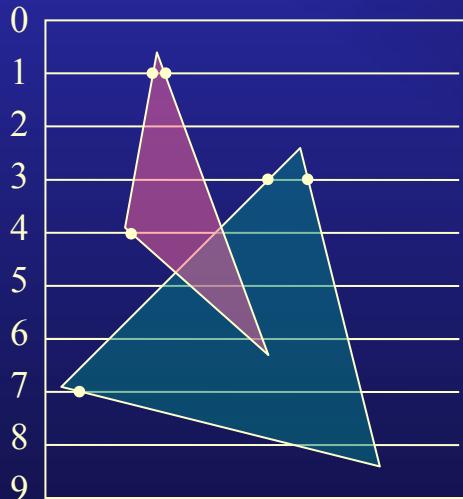
- For each new image: vertically sort all polygon edges by y_s coordinate. Use a bucket sort with one bucket per scan-line of vertical resolution. Within the edge list, sort by x_s :



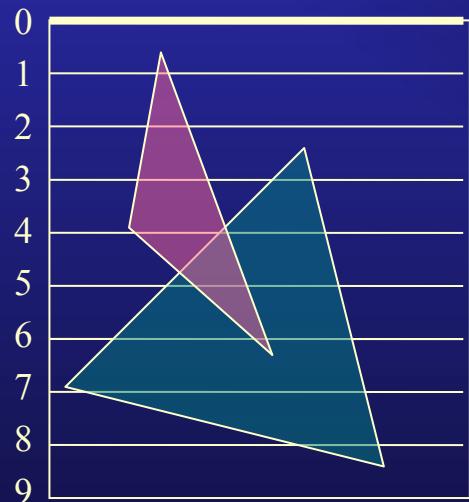
For each new image...

- For each new scan-line: Advance the active scan-line downward from the top. Decrement the scan-line crossing count per block. At each scan-line generate the active edge list based on additions, deletions (crossing count becomes 0), and (x position) modifications to the edge blocks already stored in that bucket.
- When the data structure for the scan line is complete either render it into a one scan line depth buffer or just draw straight line segments from one edge block to the next.
- Let's do an example with 10 scan-lines and just 2 triangle polygons called T and P:
 - T has three edges T₁, T₂, and T₃
 - P has three edges P₁, P₂, and P₃

Initial State (Pre-Process) of Scan-Line Buckets (y-x sort)



For each Scan-Line, Build the Active Edge List

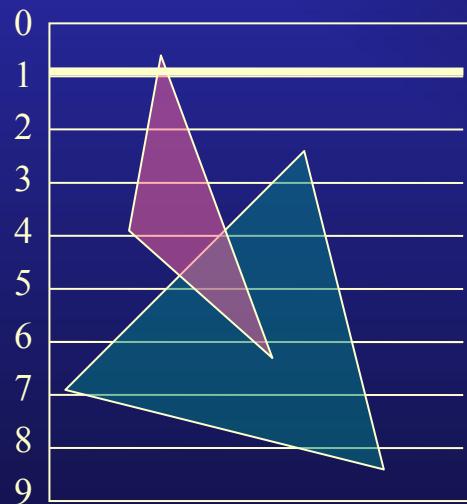


0 

Scan-line 0

- No additions
- No deletions
- No updates

For each Scan-Line, Build the Active Edge List



Scan-line 1

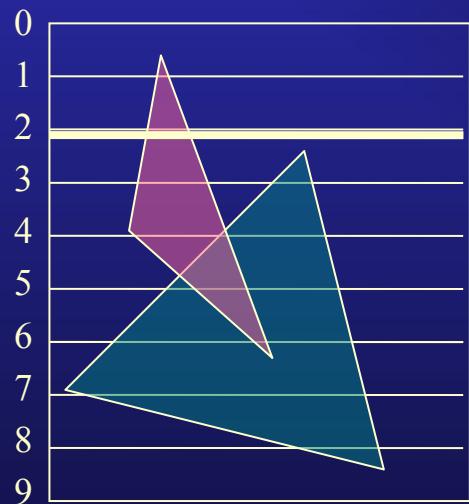
- 2 additions
- No deletions
- No updates



Add

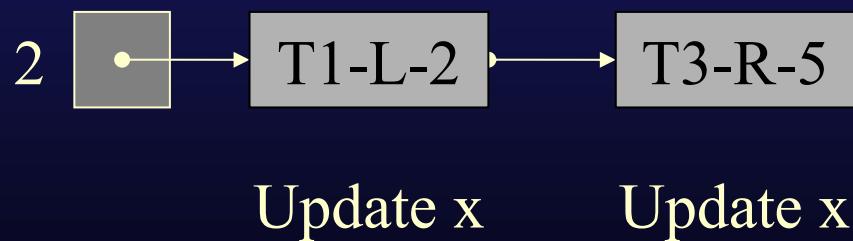
Add

For each Scan-Line, Build the Active Edge List

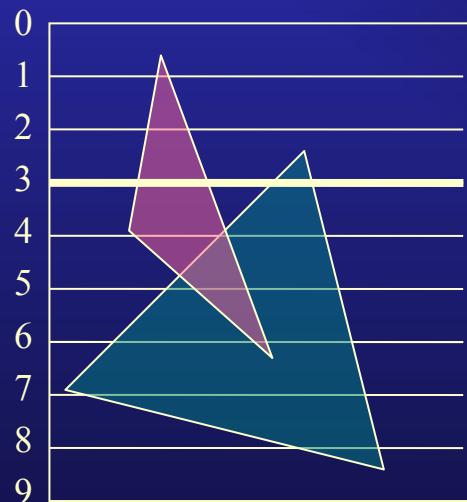


Scan-line 2

- No additions
- No deletions
- 2 updates



For each Scan-Line, Build the Active Edge List

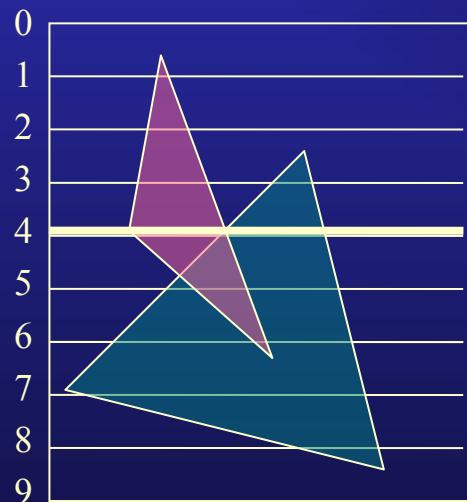


Scan-line 3

- 2 additions
- No deletions
- 2 updates



For each Scan-Line, Build the Active Edge List



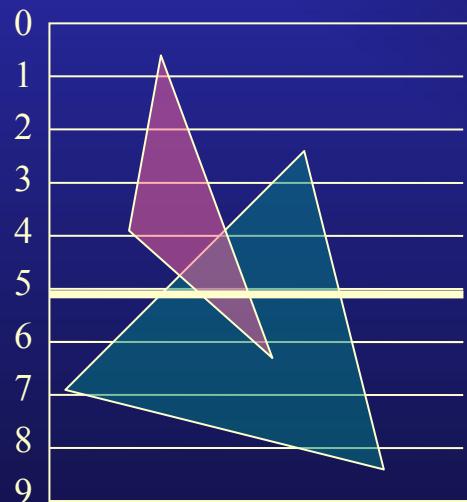
Scan-line 4

- 1 addition
- 1 deletion (T1-L-0)
- 3 updates



Note that these two blocks
are re-sorted to maintain x
order.

For each Scan-Line, Build the Active Edge List



Scan-line 5

- No additions
- No deletions
- 4 updates



Update x

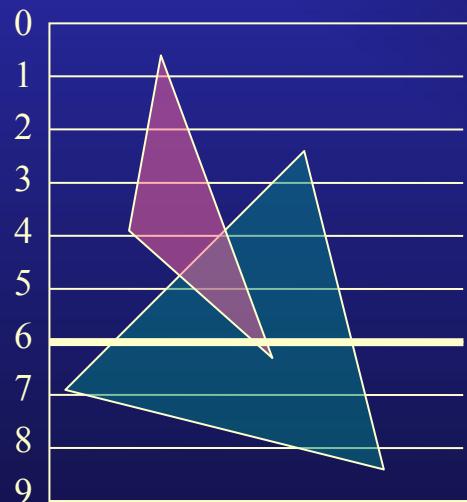
Update x

Update x

Update x

Note that these two blocks
are re-sorted to maintain x
order.

For each Scan-Line, Build the Active Edge List



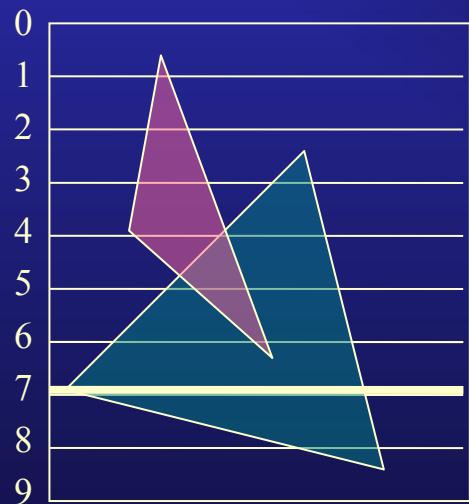
Scan-line 6

- No additions
- No deletions
- 4 updates



No re-sorting is needed.

For each Scan-Line, Build the Active Edge List



Scan-line 7

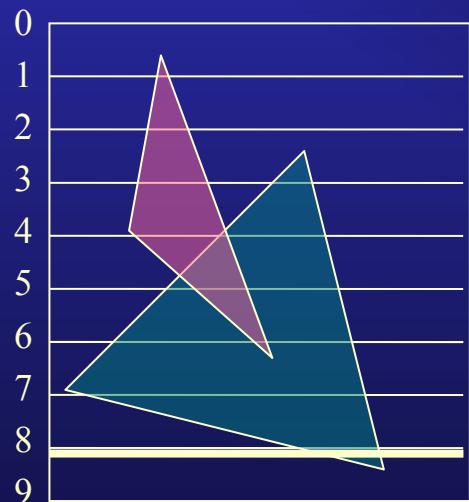
- 1 addition
- 3 deletions (P1-L-0, T2-L-0, T3-R-0)
- 1 update



Add

Update x

For each Scan-Line, Build the Active Edge List

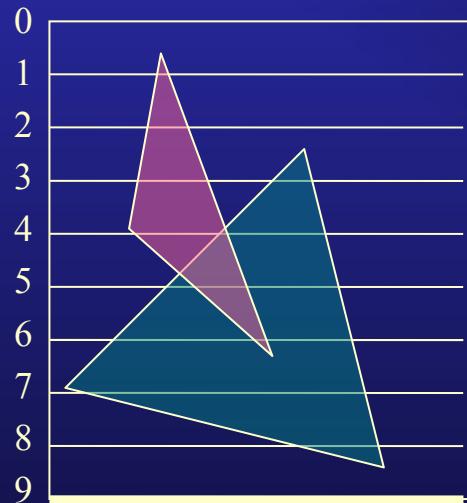


Scan-line 8

- No additions
- No deletions
- 2 updates



For each Scan-Line, Build the Active Edge List



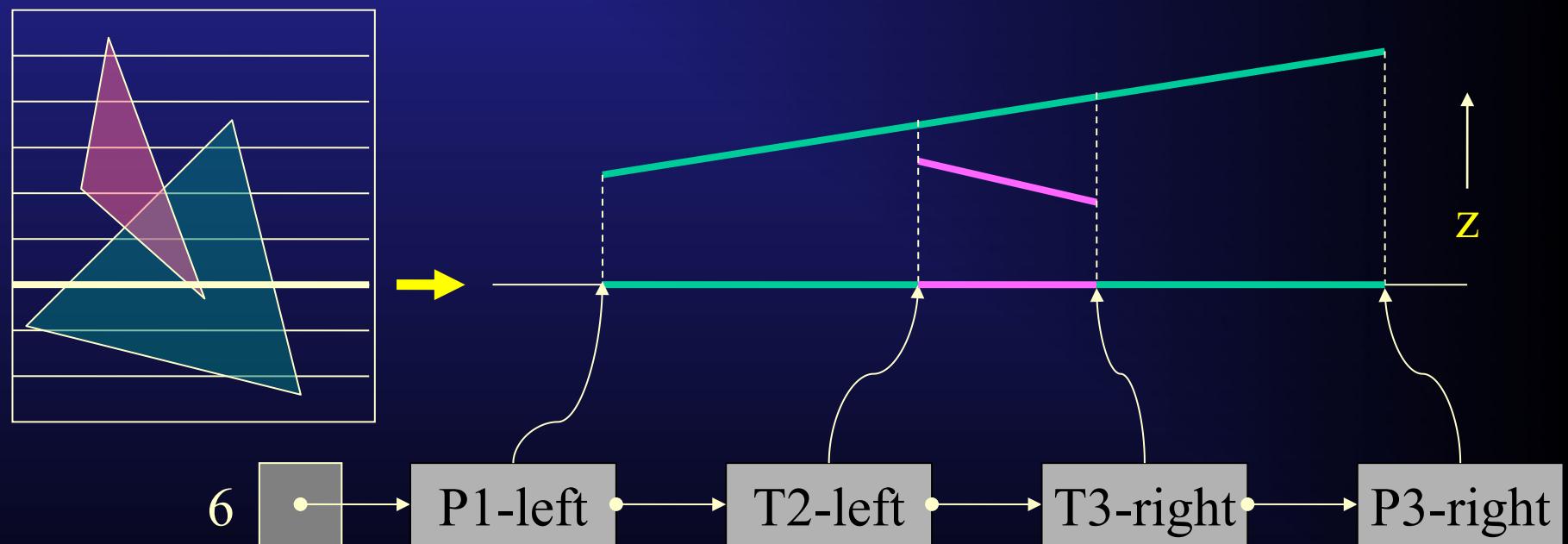
9 

Scan-line 9

- No additions
- 2 deletions (P2-L-0, P3-R-0)
- No updates

For each Scan-line, Generate Visible Segments

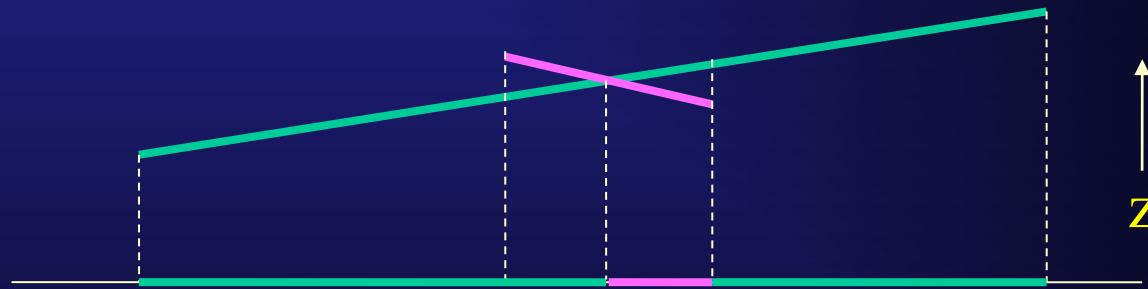
- Generate the segment list: Scan the active scan-line left to right to determine visible segments or segment fragments, hence spans, based on 1-D depth (smallest z) comparisons or priority, e.g.:



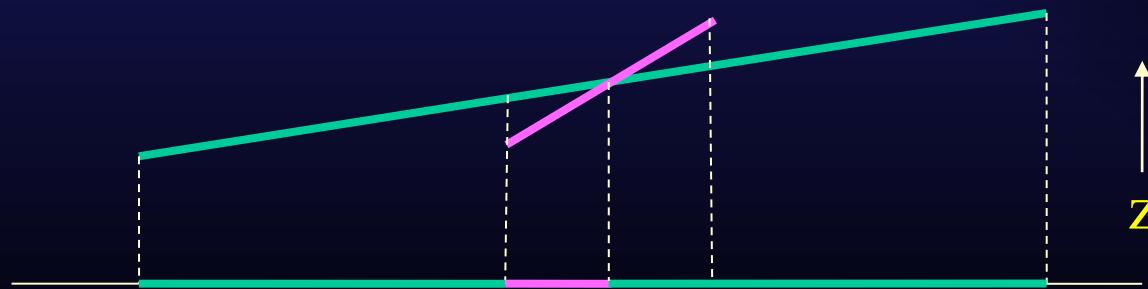
Need Additional Logic if Active Segments Intersect

- Reduces to a 1-D problem.
- If active segments cross, find intersection and solve for the closest segments in each interval.
- Two cases: (a) Segment emerges; (b) Segment “disappears”.

(a)



(b)



List or Priority-Based Algorithm

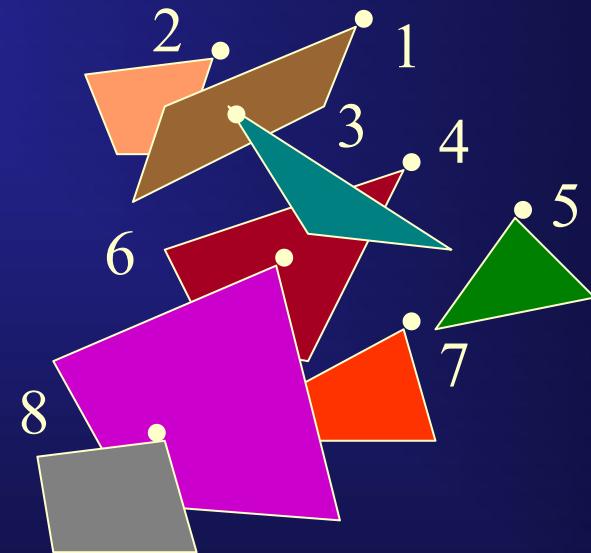
- One object has a lower priority than another if it cannot obscure the other.
- A priority ordering is a partial ordering on the set of polygons in the scene.
- In general, such orderings may not exist: overlapping polygons and improper sort orders can occur.
- But we'll press on anyway...

The Painter's Algorithm

- Newell, Newell, and Sancha.
- Assume priority order exists.
- Sort polygons on furthest vertex.
- Rasterize polygons into frame buffer in sorted order from furthest to closest.
- Note that no depth buffer is used!
- Since this doesn't always work, why do it?
 - Sorting is done prior to rendering.
 - No extra depth buffer memory or pixel depth checking (i.e., no special hardware needed -- this was before GPU cards...)
 - Allows for some transparency effects.

Painter's Example

Sort by
furthest
vertex

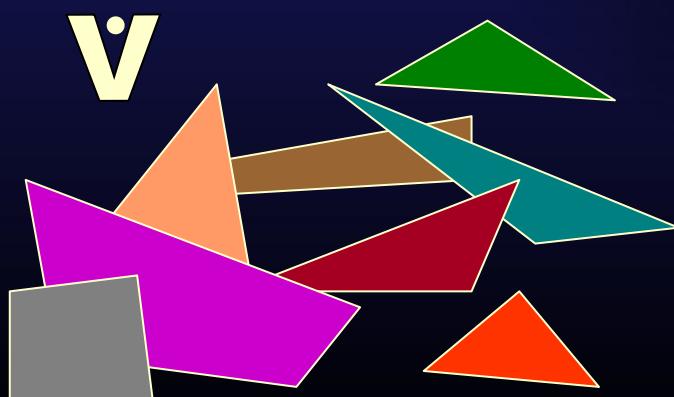


Scan-plane



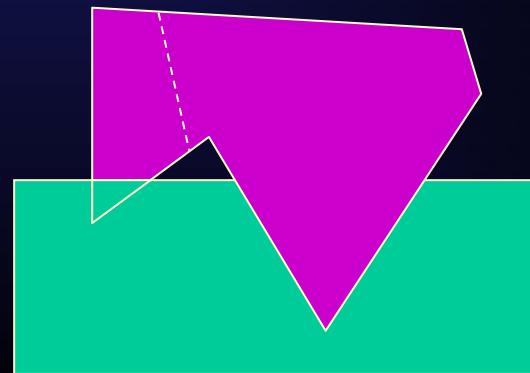
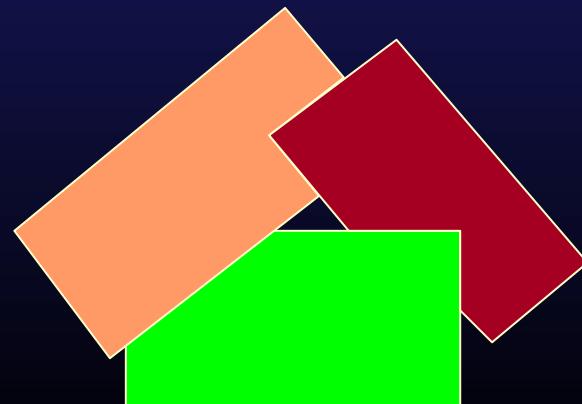
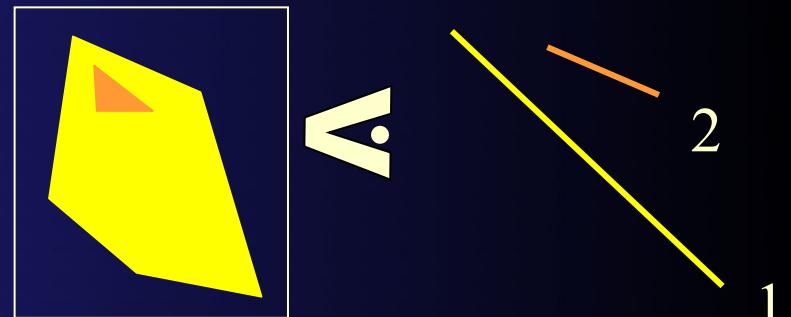
Scan-line

Then rasterize in order:
1, 2, 3, 4, 5, 6, 7, 8.



Painter's Problems

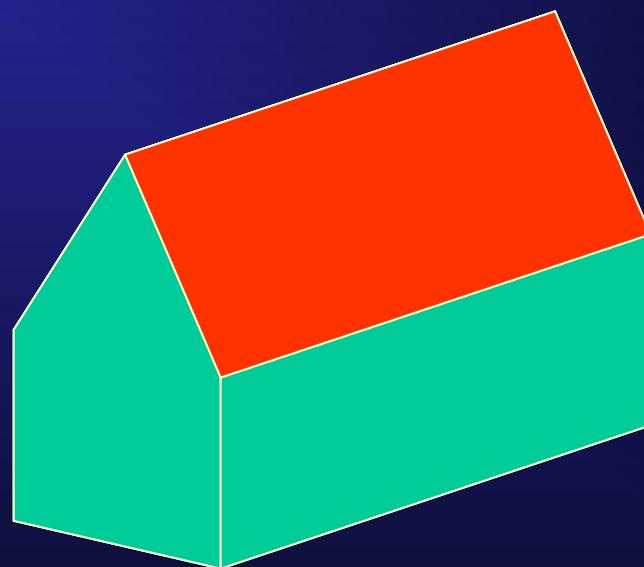
- Problems if priority order exists but not corresponding to sort on furthest vertex:
- Problem **not** solved by choosing another polygon feature, such as “closest vertex”.
- Need additional special case checks.
- Problem with cyclic overlaps: must find and split polygons:



Why Painter's Algorithm was Used

- The raw back-to-front sort can approximate a visible surface display reasonably well if
 - Objects are well separated.
 - All polygons are roughly the same size.
 - All polygons are small.
 - No polygons intersect.
 - No polygons create cyclic overlaps.
- Surprisingly, many scenes meet these criteria if carefully designed.
- This algorithm was used successfully at Ohio State for many years during the 80's.
- Used in PowerPoint! See house example! (But polygons aren't in 3D.) [See "Selection Pane" in PowerPoint 2007.]

Painter's Example -- House



Exact Algorithm -- Atherton and Weiler

- Screen subdivision by projected polygon outlines.
- “Exact” because it uses the actual polygon boundaries as the rendering primitive.
- Uses each polygon as a *cookie-cutter* on remainder of scene.
- Within each cookie-cutter polygon:
 - if remainder of scene is behind cookie-cutter polygon,
then draw the cookie-cutter polygon
 - else re-enter the algorithm with another polygon as
the cookie-cutter.
- Benefits from, but does not require, a back-to-front polygon ordering.
- Permits shadows and transparency effects.

Atherton and Weiler Algorithm Outline (1)

Draw (List):

while List NOT Nil **do**

 List \leftarrow DrawOne (head (List), tail (List))

DrawOne (ClipPoly, List) **returns** (OutList):

 InList \leftarrow OutList \leftarrow Nil

for each Poly **in** List **do**

 CookieCut (ClipPoly, Poly, InList, OutList)

if InList = Nil

then display (ClipPoly)

else append (InList, ClipPoly); Draw

 (InList)

Atherton and Weiler Algorithm Outline (2)

CookieCut (ClipPoly, Poly, InList, OutList):

clip Poly into inside fragments and outside fragments

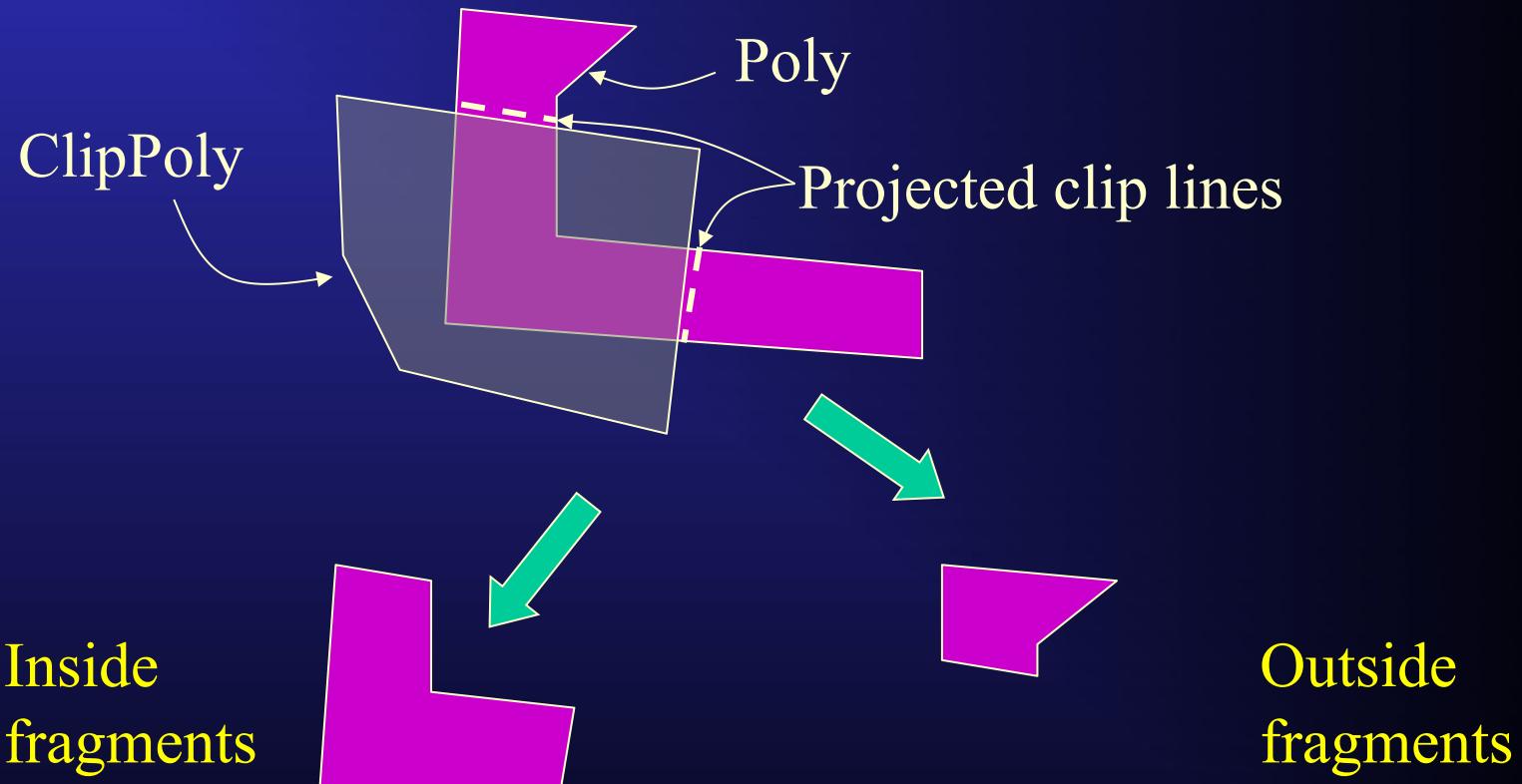
append (OutList, outside fragments)

depth clip inside fragments with plane of ClipPoly

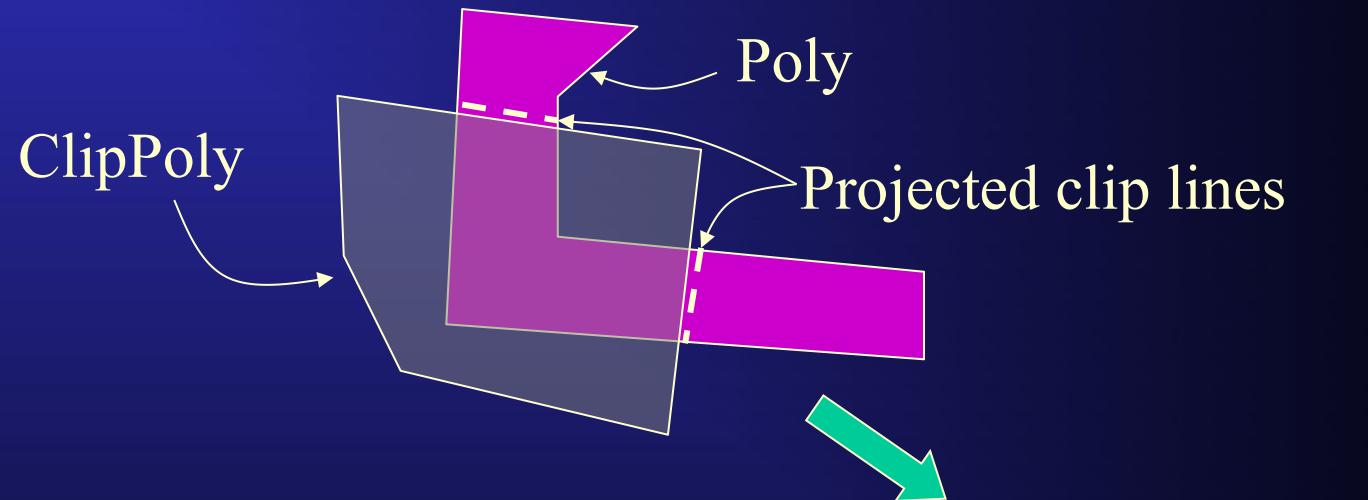
append (InList, remaining inside fragments)

- NOTICE that the clip operation is both exact and not cheap.
- It is also subject to degeneracies.
- Anyway, here's how it works...

Clipping a Polygon with ClipPoly



Clipping a Polygon with ClipPoly



Inside
fragments



Clip to plane of ClipPoly (yields Nil)

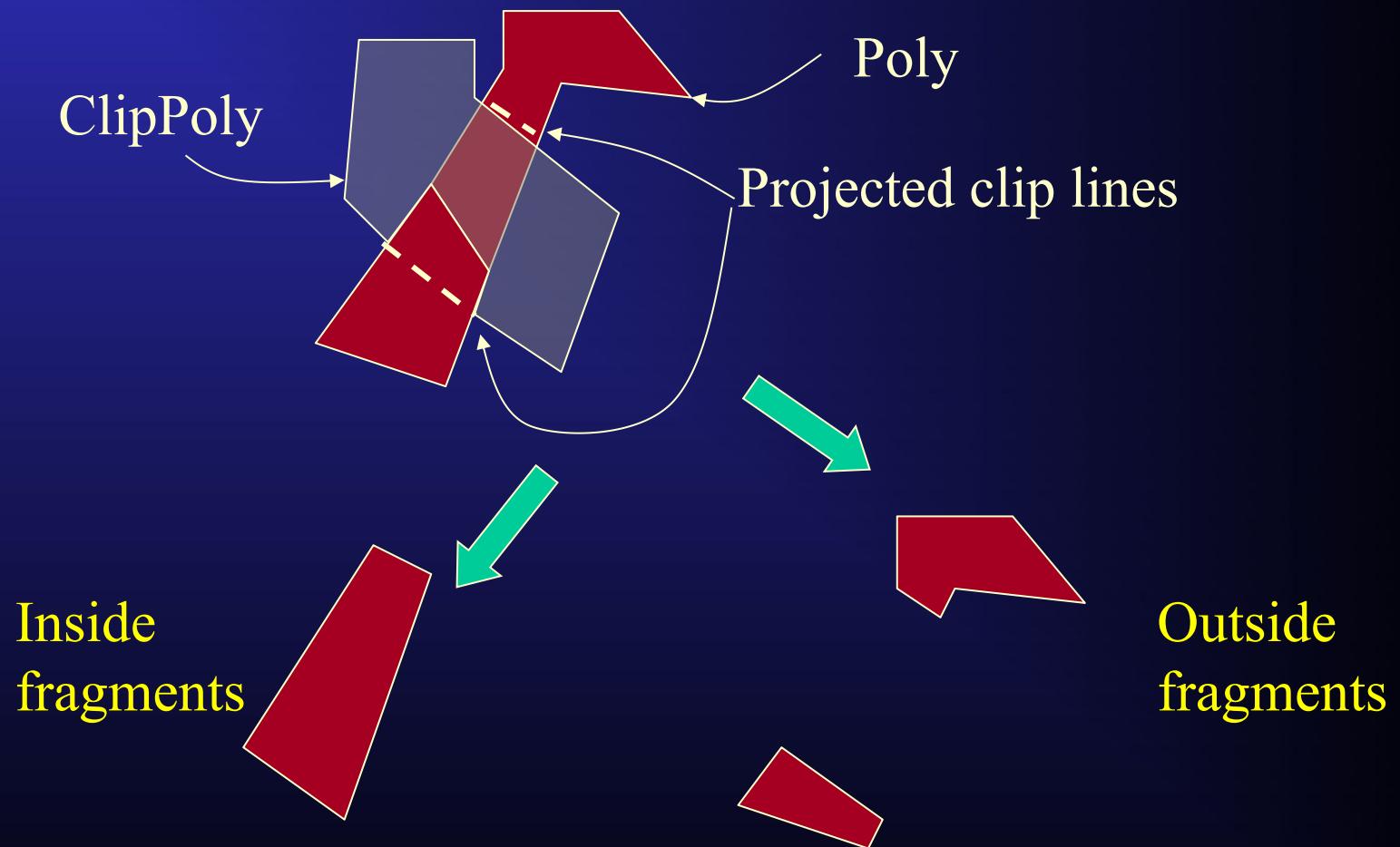
Append to InList (Nil)

Outside
fragments



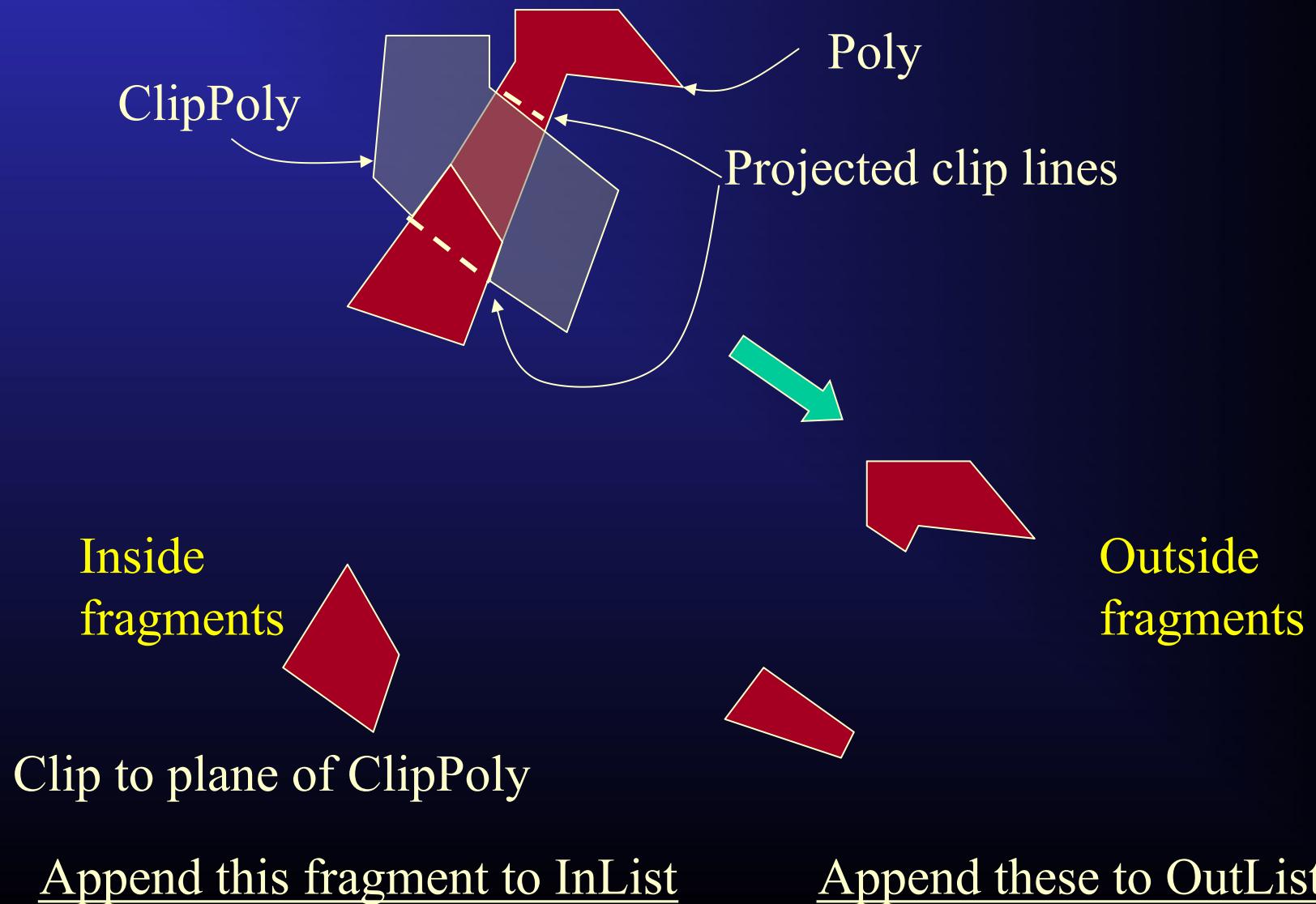
Append these to OutList

Clipping a Penetrating Polygon with ClipPoly



Append these to OutList

Clipping a Penetrating Polygon with ClipPoly

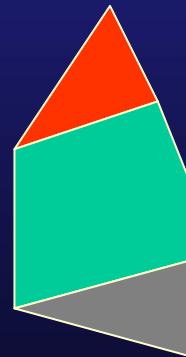


House Example

ClipPoly



Inside fragments

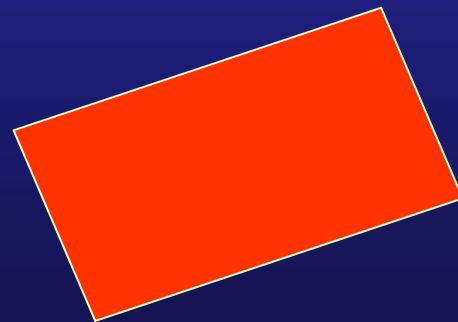


Draw result

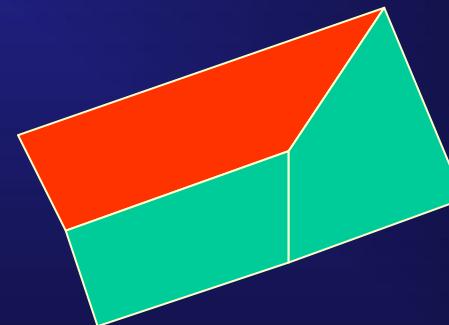


House Example

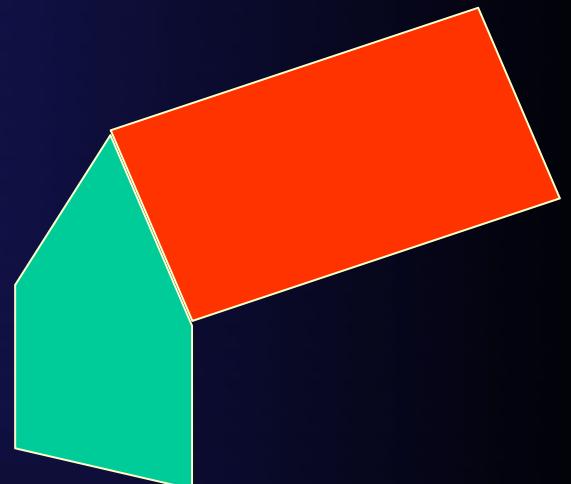
ClipPoly



Inside fragments

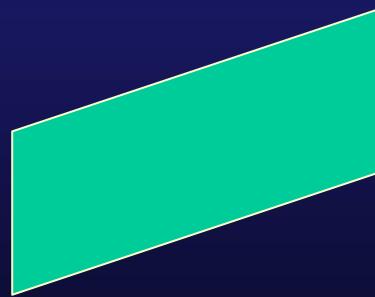


Draw result

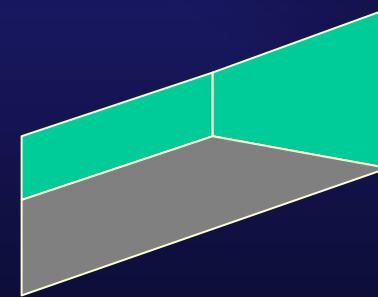


House Example

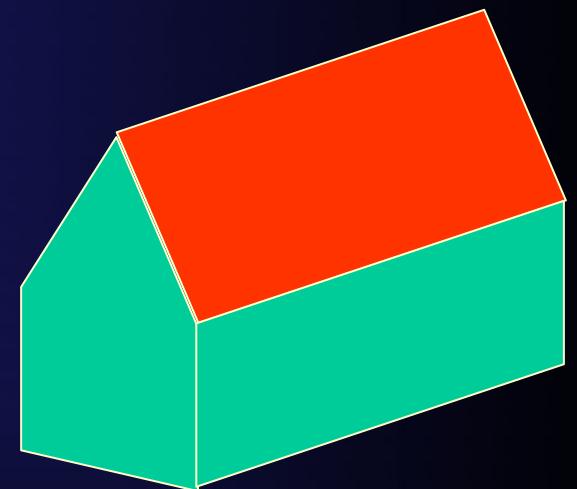
ClipPoly



Inside fragments



Draw result

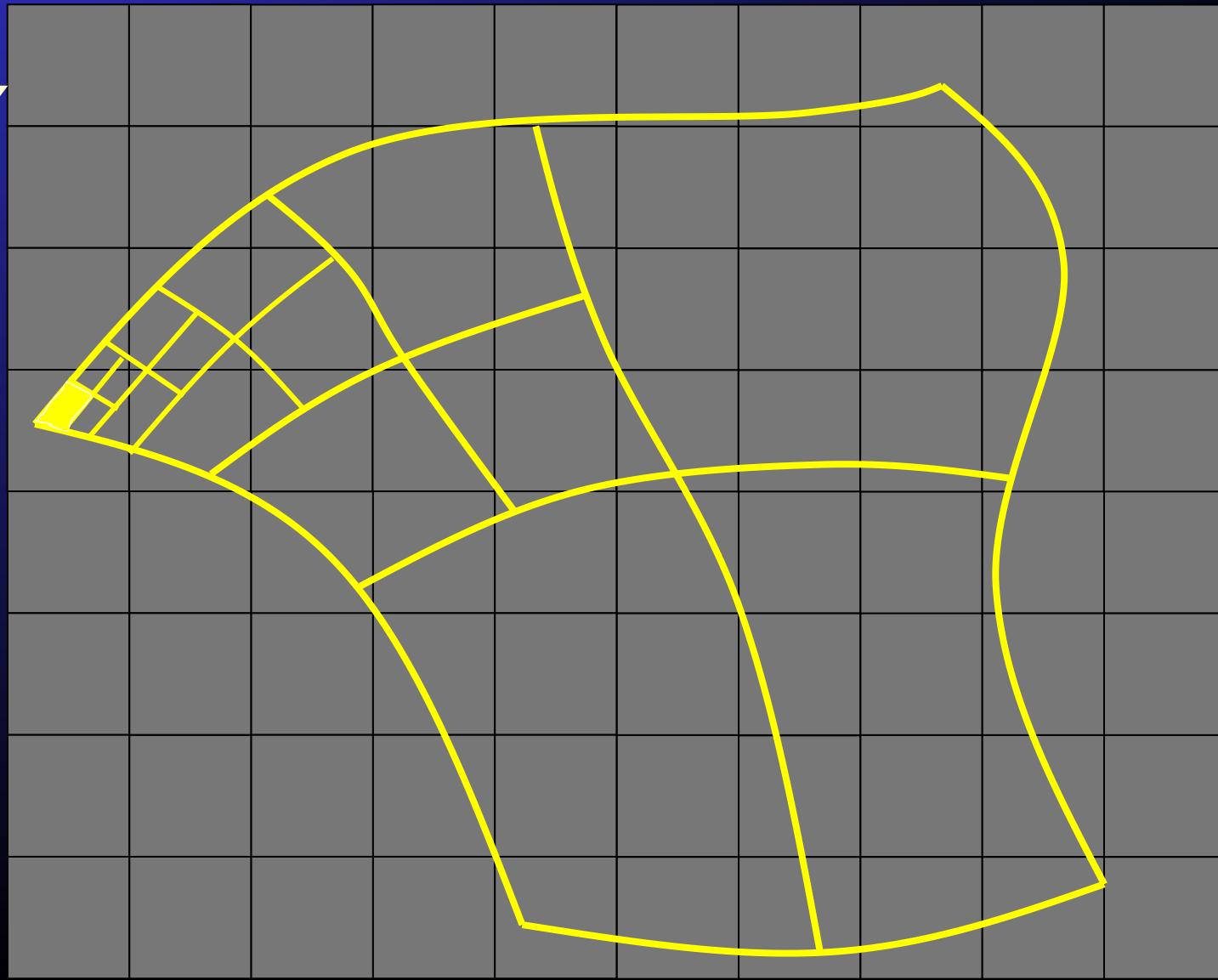


Curved Surface Patch Display

- Rendering curved patches is complicated by their non-planarity.
- Either subdivide or find ray-surface intersection.
- Subdivision method:
 - 1) See if patch projects entirely inside a pixel. If so, find its closest point.
 - 2) If not, subdivide patch into 4 (smoothly tangent) pieces and recurse with each from (1).
- Notice in the example that the subdivided 4-sided patches eventually become virtually planar -- so they can be split into two triangles and used in a conventional polygon visible surface display...
 - But, watch out for cracks!

Subdividing Curved Patches

pixel ↗



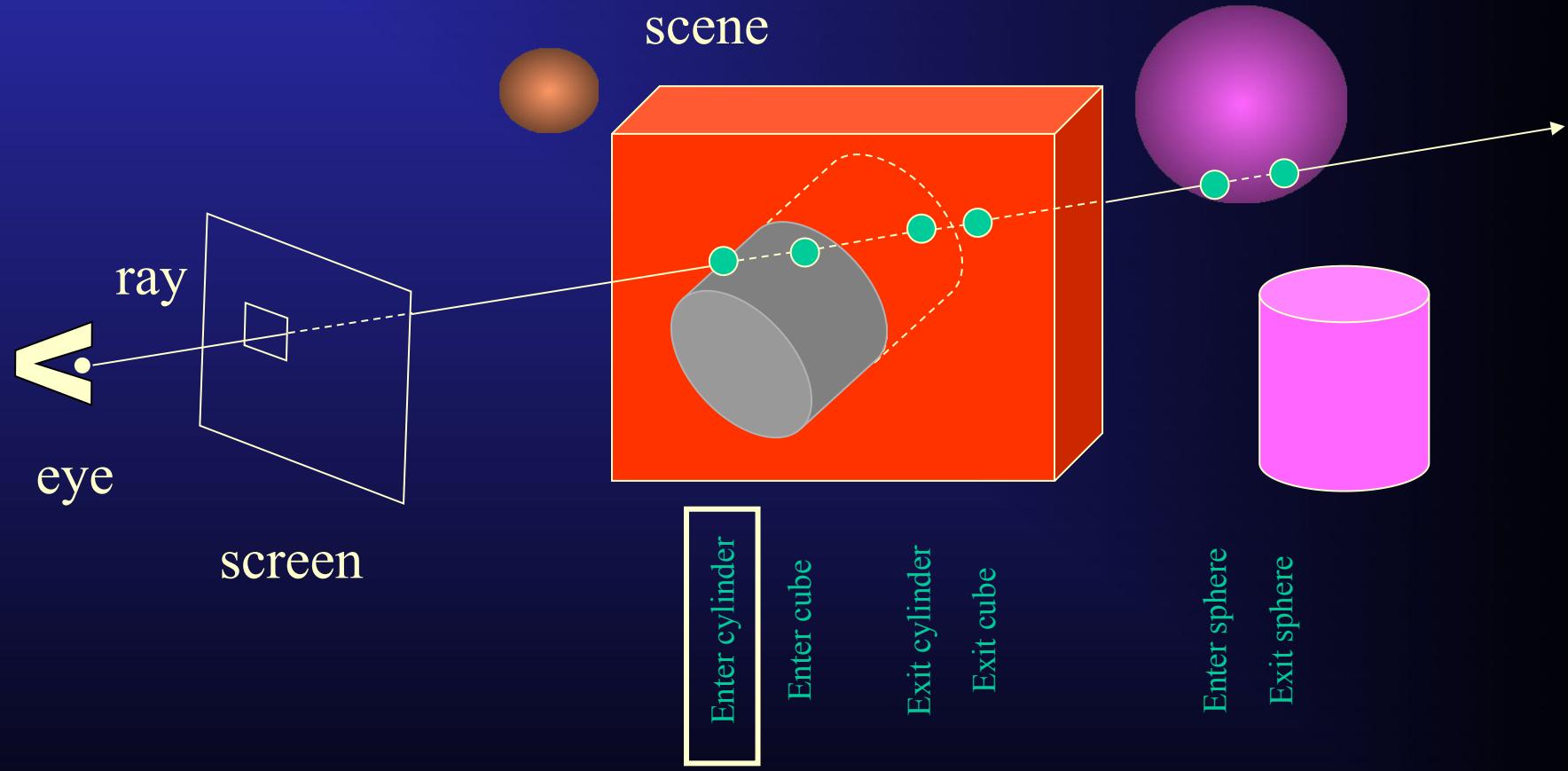
Ray Casting

- Based on ray-object intersection computation.
- The ray from the eye through the pixel center intersects transformed instances of the scene's geometry primitives.
- We'll ignore degeneracies (although they are important).
- Computing ray-object intersections:
 - Ray-plane, ray-triangle, ray-polygon, ray-box
 - Ray-sphere.
 - Ray-cylinder.
 - Ray-casting CSG-tree models.
 - Ray-casting voxels.
 - Ray-casting implicit surfaces (“Marching Cubes”).

Ray-CSG Intersection

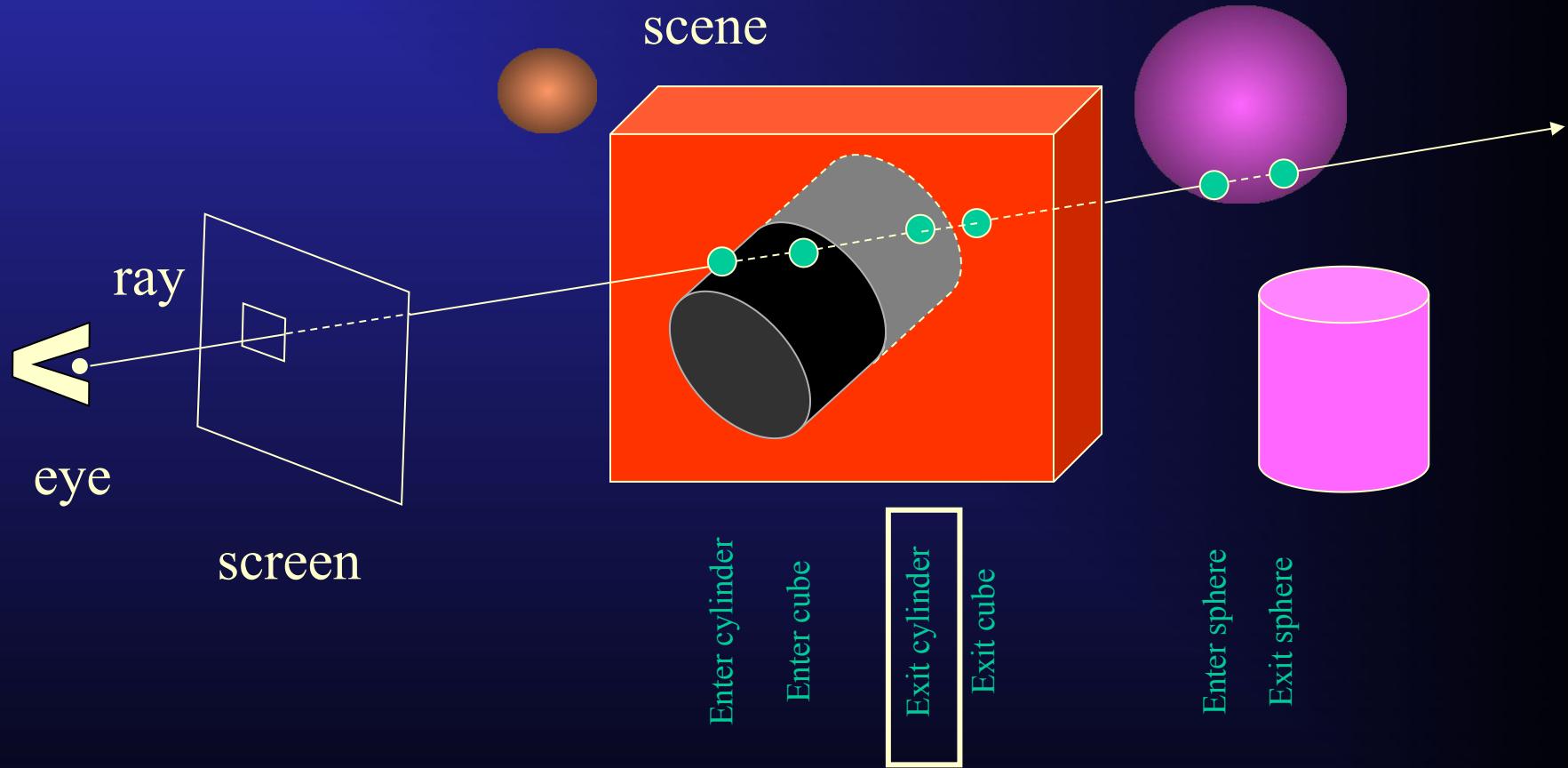
- Need algorithm that works with CSG trees ($- \cup \cap$ operators)
- Combinations imply surface normal may come from one primitive while surface color may come from another.
- Some single operator examples first, then the algorithm for the general ray-CSG intersection.

CSG Ray Casting: (CUBE \cup CYLINDER)



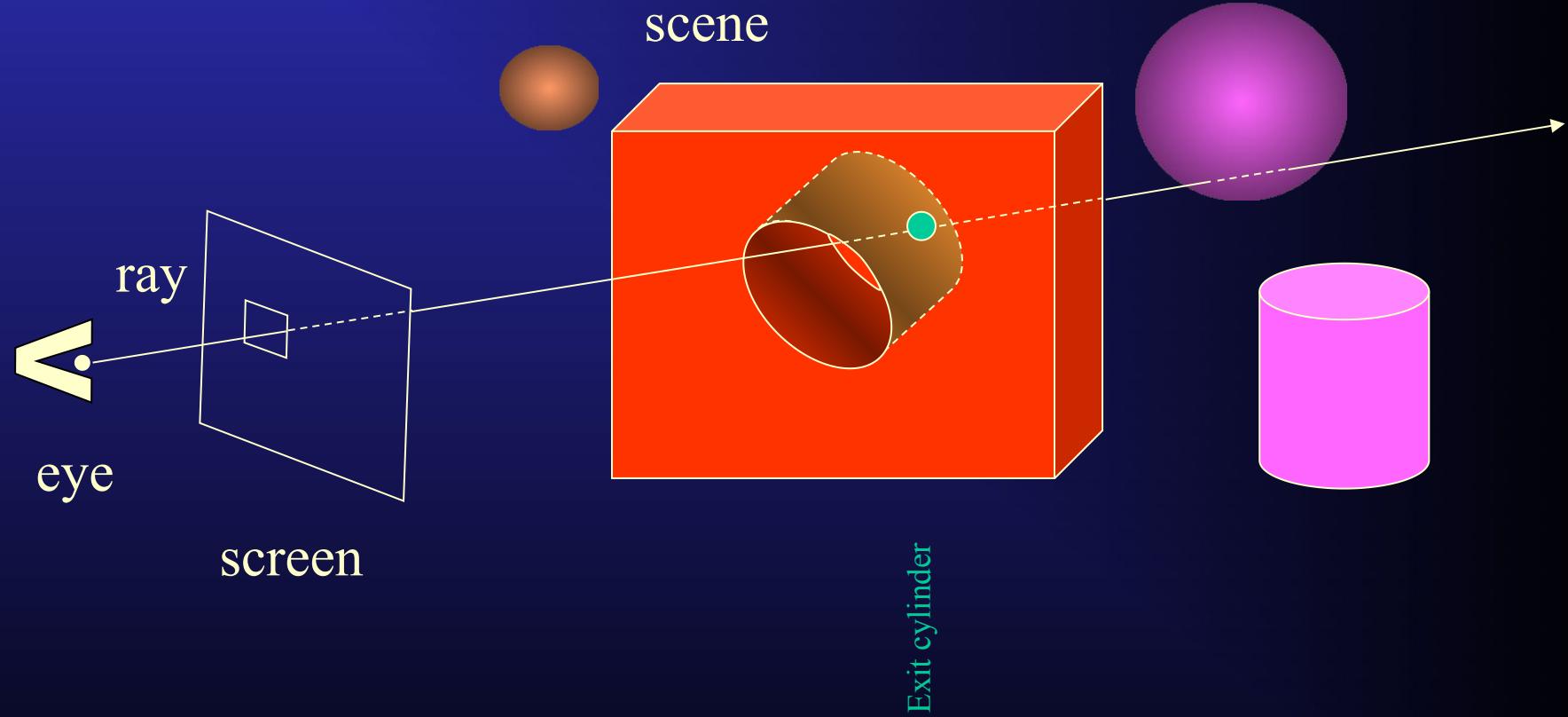
- Find intersections and sort by Z distance from eye
- Display first VISIBLE point in color of hit object

CSG Ray Casting: (CUBE – CYLINDER)



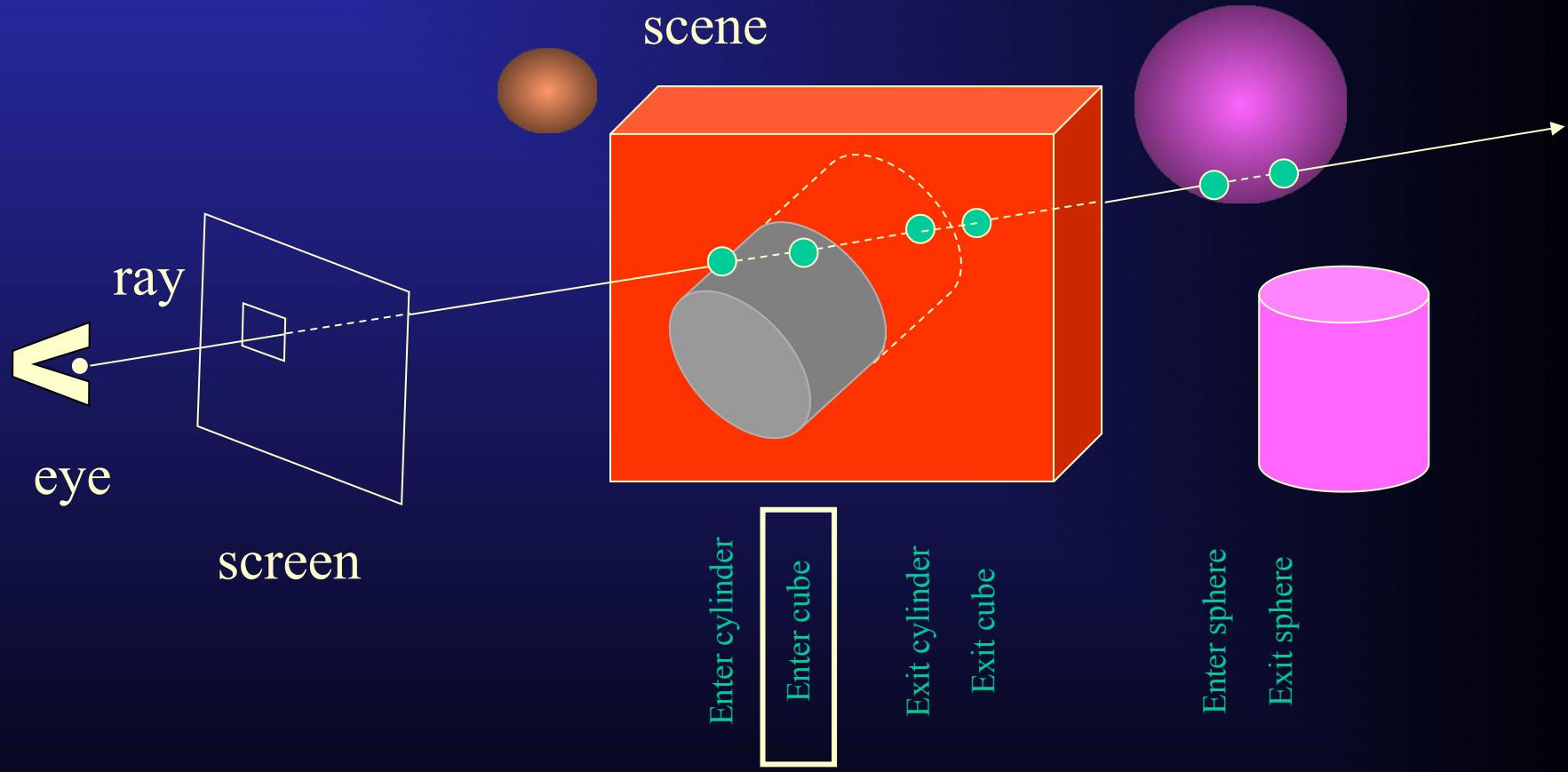
IN CUBE BUT OUTSIDE CYLINDER:
color from CUBE; normal from CYLINDER

CSG Ray Casting: (CUBE – CYLINDER)



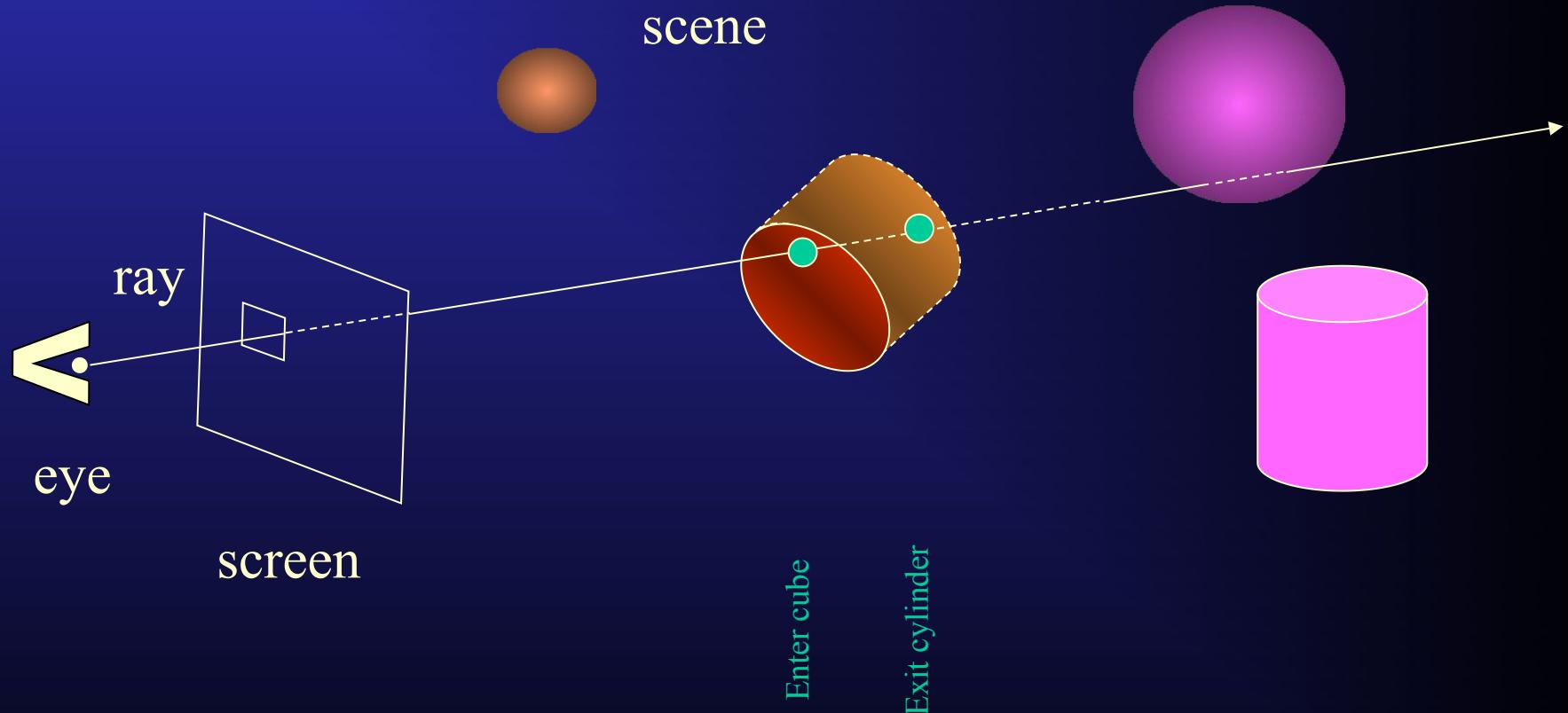
IN CUBE BUT OUTSIDE CYLINDER:
color from CUBE; normal from CYLINDER

CSG Ray Casting: (CUBE \cap CYLINDER)



IN CUBE AND INSIDE CYLINDER:
color from CUBE; normal from CUBE

CSG Ray Casting: (CUBE \cap CYLINDER)



IN CUBE AND INSIDE CYLINDER:
color from CUBE; normal from CUBE

Ray-CSG Algorithm (Roth)

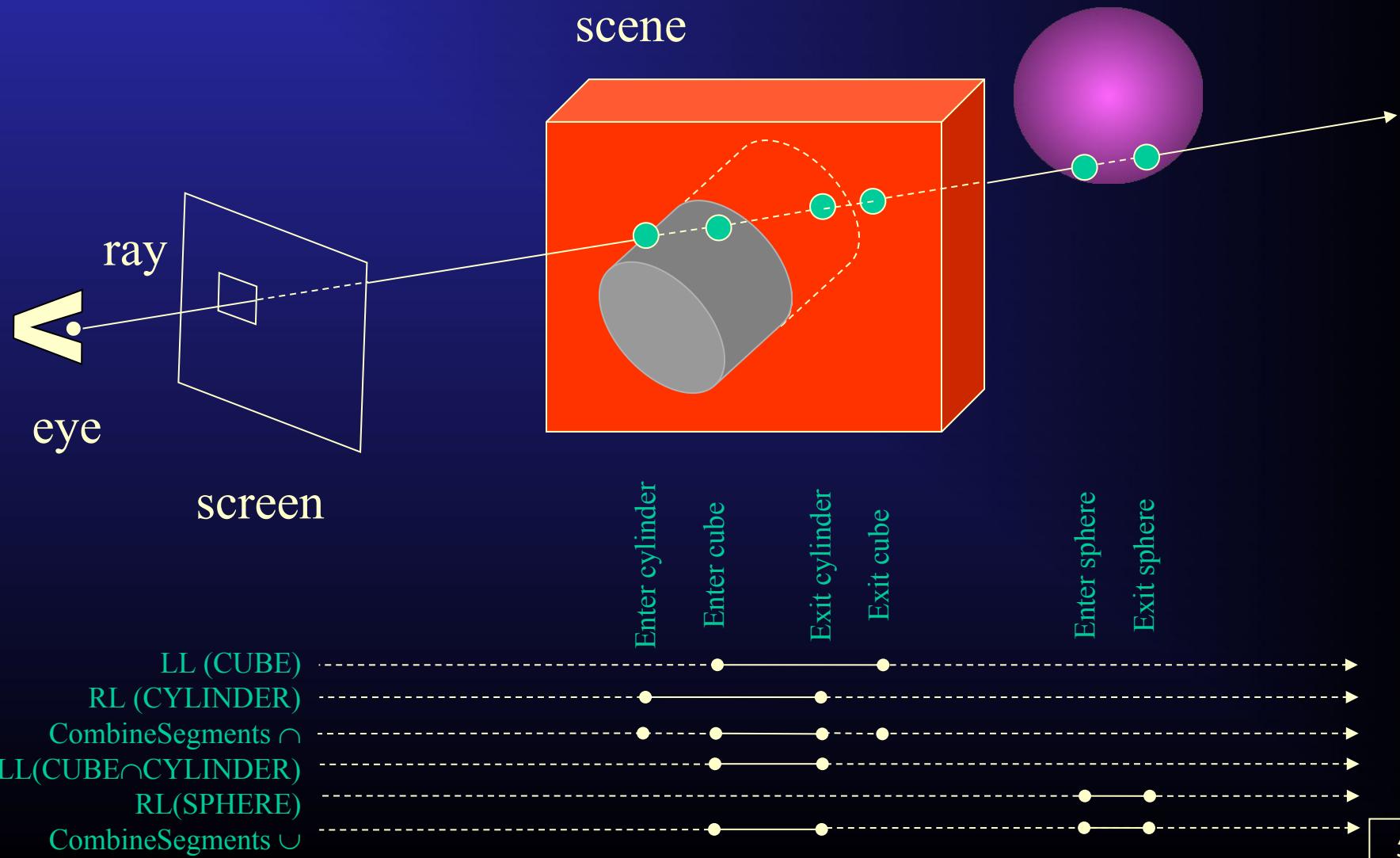
- Main process: Pre-order CSG-tree traversal.
- Intersect ray with left-subtree to obtain LL (Left List) of **in** and **out** segments; likewise, intersect ray with right-subtree to obtain RL (Right List) of **in** and **out** segments.
- Key function:
 - CombineSegments: merge LL and RL segments into a single sorted list, re-classify* segments as **in** or **out**, and merge like-labeled contiguous segments.
 - *Classification table:

| Left seg | Right seg | \cup | \cap | - |
|----------|-----------|--------|--------|-----|
| in | in | in | in | out |
| in | out | in | out | in |
| out | in | in | out | out |
| out | out | out | out | out |

CSG-Tree Ray Casting Traversal

```
function ray_CSG_intersection (CSGtree T, ray r) returns SegmentList
begin SegmentList LL, RL;
if T is a primitive // i.e., a leaf node
then return  $r \cap T$ 
else begin
    LL = ray_CSG_intersection (left_subtree(T), r)
    if (empty(LL) and Op(T) ≠ ∪))
        then return nil
    else begin
        RL = ray_CSG_intersection (right_subtree(T), r);
        return CombineSegments(LL, RL, Op(T))
    end
end
end
```

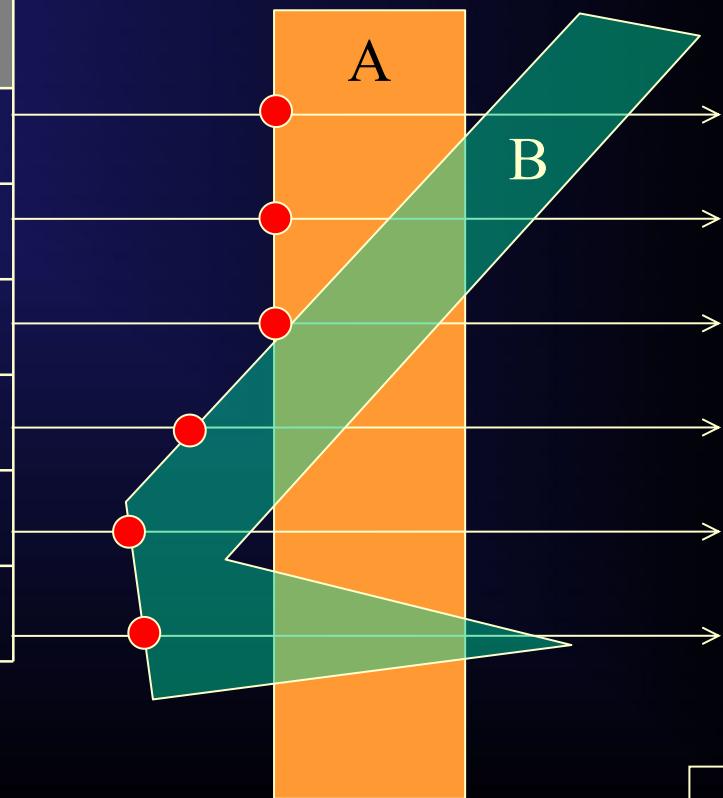
E.g. -- $(\text{CUBE} \cap \text{CYLINDER}) \cup \text{SPHERE}$



What is the Normal and Color?

- CombineSegments yields the visible point, but we still need the correct surface normal and color.
- Cases (+ means enter; – means exit; N is normal; C is color):

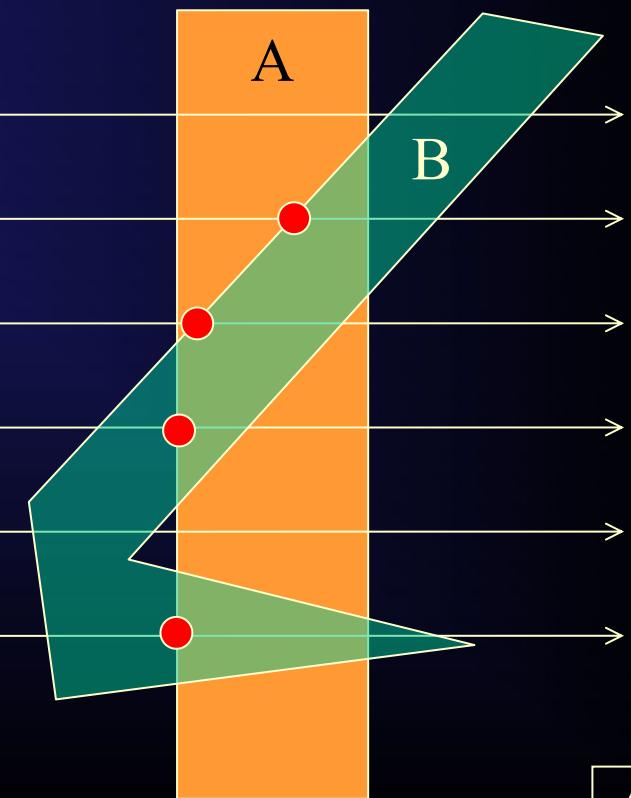
| CASE | $A \cup B$ |
|----------------|------------------------------------|
| +A, -A, +B, -B | $N \leftarrow +A; C \leftarrow +A$ |
| +A, +B, -A, -B | $N \leftarrow +A; C \leftarrow +A$ |
| +A, +B, -B, -A | $N \leftarrow +A; C \leftarrow +A$ |
| +B, +A, -B, -A | $N \leftarrow +B; C \leftarrow +B$ |
| +B, -B, +A, -A | $N \leftarrow +B; C \leftarrow +B$ |
| +B, +A, -A, -B | $N \leftarrow +B; C \leftarrow +B$ |



What is the Normal and Color?

- CombineSegments yields the visible point, but we still need the correct surface normal and color.
- Cases (+ means enter; – means exit; N is normal; C is color):

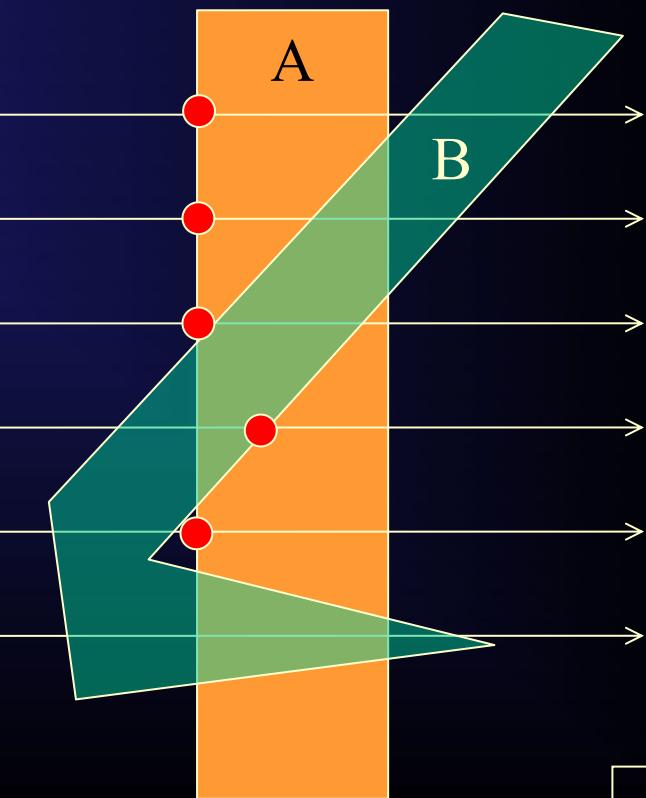
| CASE | $A \cap B$ |
|----------------|------------------------------------|
| +A, -A, +B, -B | null |
| +A, +B, -A, -B | $N \leftarrow +B; C \leftarrow +A$ |
| +A, +B, -B, -A | $N \leftarrow +B; C \leftarrow +A$ |
| +B, +A, -B, -A | $N \leftarrow +A; C \leftarrow +A$ |
| +B, -B, +A, -A | null |
| +B, +A, -A, -B | $N \leftarrow +A; C \leftarrow +A$ |



What is the Normal and Color?

- CombineSegments yields the visible point, but we still need the correct surface normal and color.
- Cases (+ means enter; – means exit; N is normal; C is color):

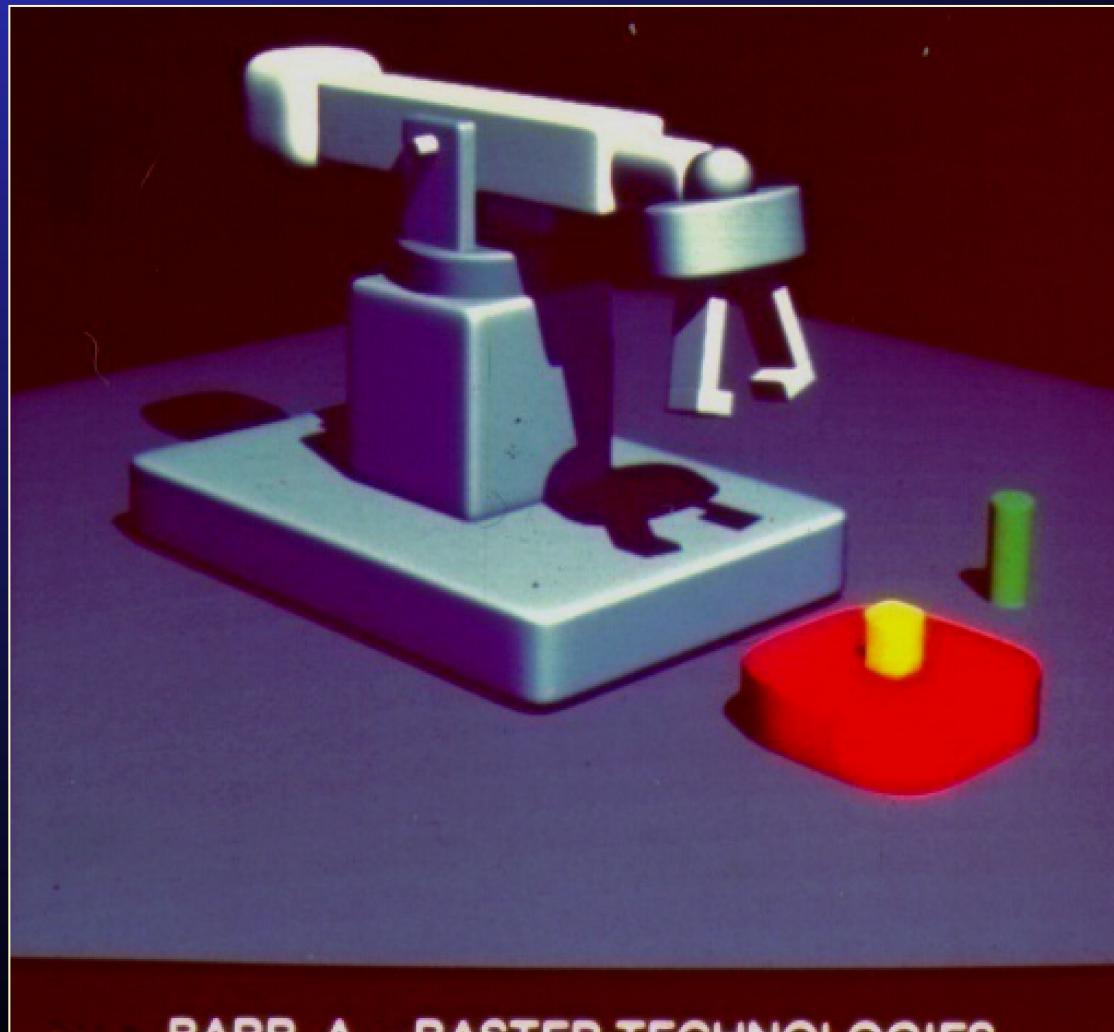
| CASE | A–B |
|----------------|------------------------------------|
| +A, -A, +B, -B | $N \leftarrow +A; C \leftarrow +A$ |
| +A, +B, -A, -B | $N \leftarrow +A; C \leftarrow +A$ |
| +A, +B, -B, -A | $N \leftarrow +A; C \leftarrow +A$ |
| +B, +A, -B, -A | $N \leftarrow -B; C \leftarrow +A$ |
| +B, -B, +A, -A | $N \leftarrow +A; C \leftarrow +A$ |
| +B, +A, -A, -B | null |



Ray-Casting Voxels

- Possible but rather inefficient as it is a **search problem**:
 - Cast ray into volume to find first occupied voxel.
- Also, most *interesting* voxel datasets are not binary, so “closest” point on volume dataset is relative to content.
- Thus the prevalent interpretation for **volume rendering** is to treat the voxels as a spatial density (translucency) function, and the ray cast is thus accumulating density as it passes through the volume, attenuated with increasing distance.

Ray Casting Needed to Directly Render Implicit Surfaces: E.g., Superellipsoids



BARR, A.—RASTER TECHNOLOGIES

Ray Intersection with a General Implicit Surface

- Defined by a function $F(\mathbf{p})$ and a level or threshold T , e.g., $F(\mathbf{p}) = 0$:

$$F(\mathbf{p}) > T \quad \mathbf{p} \text{ is inside the object}$$

$$F(\mathbf{p}) = T \quad \mathbf{p} \text{ is on the surface}$$

$$F(\mathbf{p}) < T \quad \mathbf{p} \text{ is outside the object}$$

- E.g., sphere of radius s centered at (O_x, O_y, O_z) :

$$F(\mathbf{p}) = s^2 - (\mathbf{p}_x - \mathbf{O}_x)^2 - (\mathbf{p}_y - \mathbf{O}_y)^2 - (\mathbf{p}_z - \mathbf{O}_z)^2$$

- In ray-implicit function intersection, must find a root of F for every ray (if one exists): hence can require much iterative numerical computation.
- Speed-ups without heavy-duty numerical methods: **ray marching** and **hierarchic spatial subdivision (oct-trees)**.

Finding the Ray-Implicit Surface Intersection

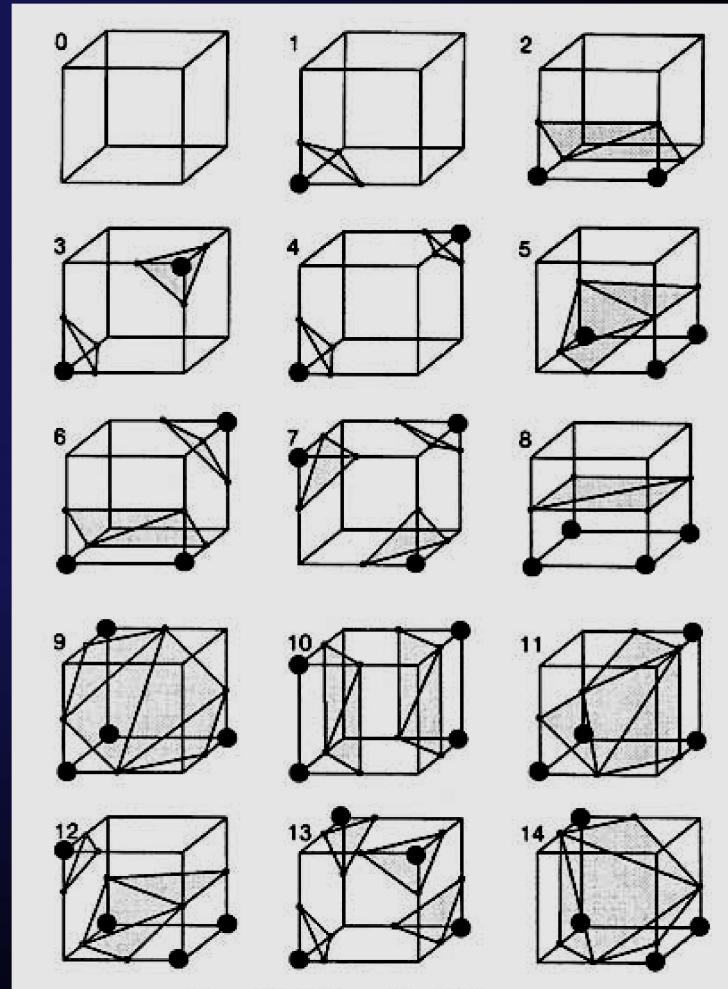
- Ray $\mathbf{r}(t) = \mathbf{p} + t\mathbf{V}$
- First intersection is $\min \{ t \mid F(\mathbf{r}(t)) = T \text{ and } t \geq 0 \}$
- Can use numerical root finder to isolate root – but convergence can be slow and there may be many regions with no roots.
- So use method that starts root finder with tighter bounds
- ... and saves information from ray to ray.

Hierarchical Spatial Subdivision Using Oct-Trees

- Idea: Embed F in an oct-tree.
- Create bounding box of the support of F : the root cell.
- Subdivide the root cell into 8 uniformly-sized subcells.
- Evaluate F at the 8 corners of each cell (note that values of F are shared by adjacent cells).
- Classify subcells as:
 - Empty:** if $F(p) < T \forall$ 8 corners (completely outside)
 - Full:** if $F(p) > T \forall$ 8 corners (completely inside)
 - Unknown:** if some $F(p) < T$, rest $F(p) > T$ (contains surface)
- Recursively subdivide **Unknown** subcells until max depth.
- Create local estimated surfaces within **Unknown** cells via Marching Cubes »

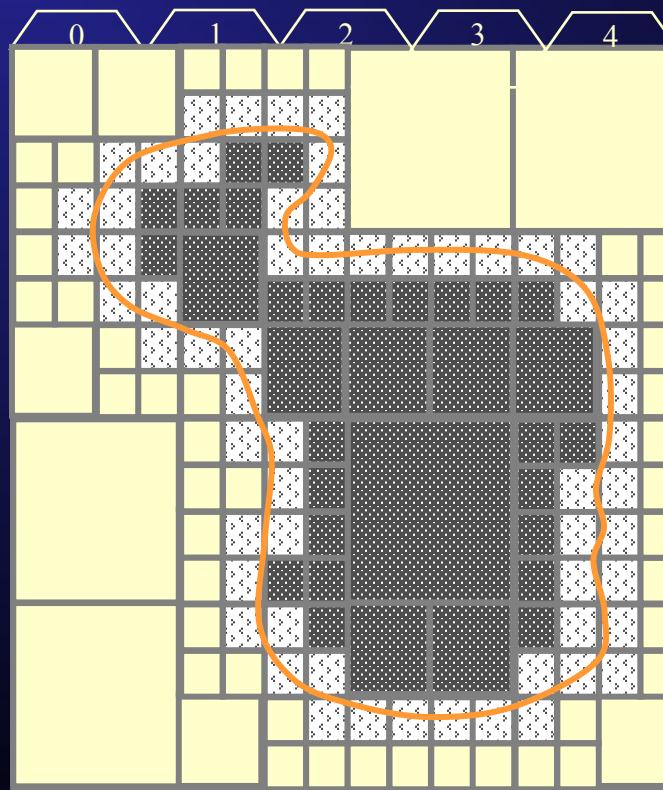
Marching Cubes (Reprise): 15 Cases

- Compute F at each cube corner.
Assume each result is $> T$ or $< T$.
(Full and Empty are case 0;
Unknown yields cases 1 through 14.)
- Where does $F = T$ isosurface pass through cube?
- Interpolate intersections between $> T$ and $< T$ values to estimate crossing.
- Can do more accurately, but polygonal approximation will suffice for tiny (think pixel resolution) cubes!
- Connect intersections to form surfaces; then can intersect with ray.
- (Cases do not account for single surface-only material incursions.)



Example Implicit Surface Subdivision

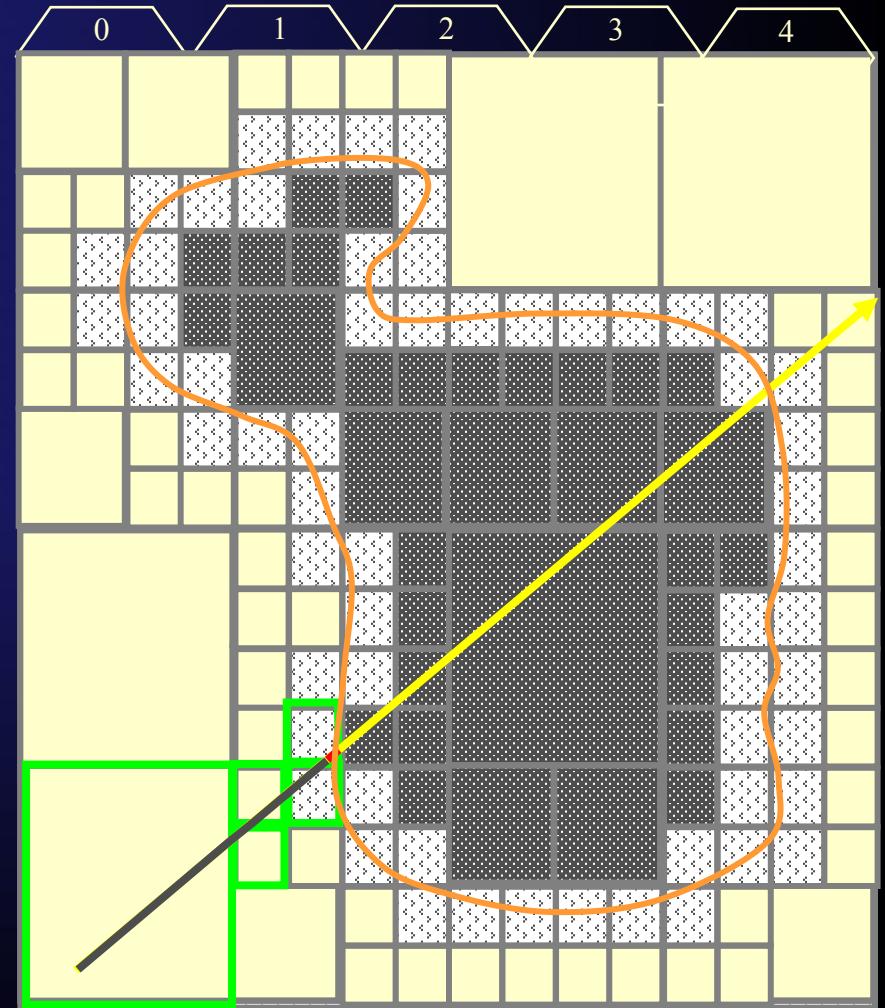
- **Full** (interior) cell
- **Unknown** cell containing $F(p)=T$
- **Empty** (outside) cell



(Showing 2D for simplicity)

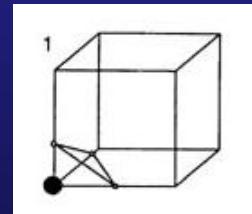
Ray Marching

- Consider each subcell that intersects ray, in order from start through ray exit from bounding box root cell:
 - Skip irrelevant **Empty**/**Full** cells.
 - Recurse on subdivided cells.
 - Use root solver or interpolation on intervals passing through leaf **Unknown** cells.
 - Find closest intersection in **Unknown** cell (if any exist); otherwise move along to next cell.

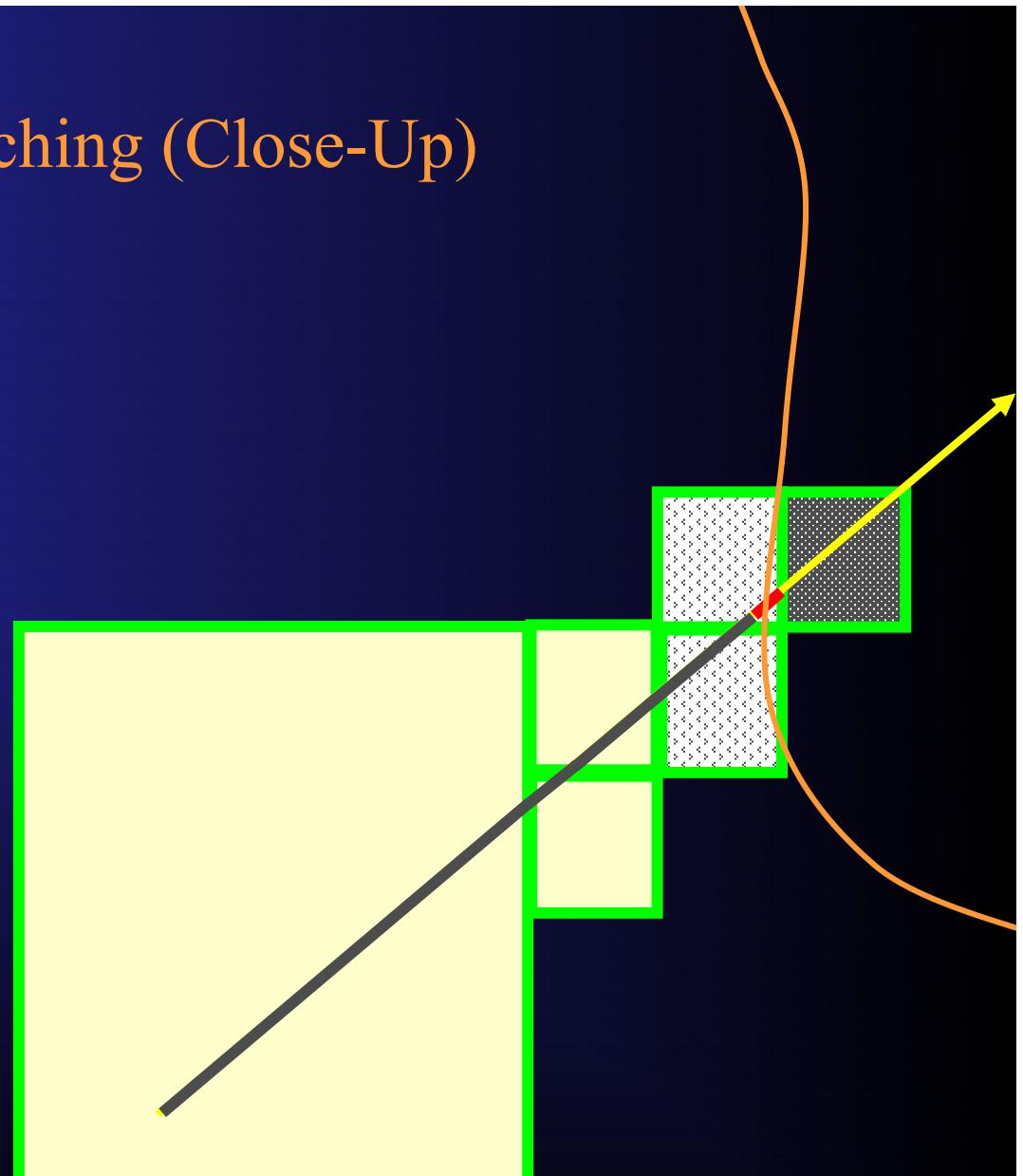
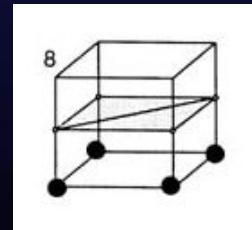


Ray Marching (Close-Up)

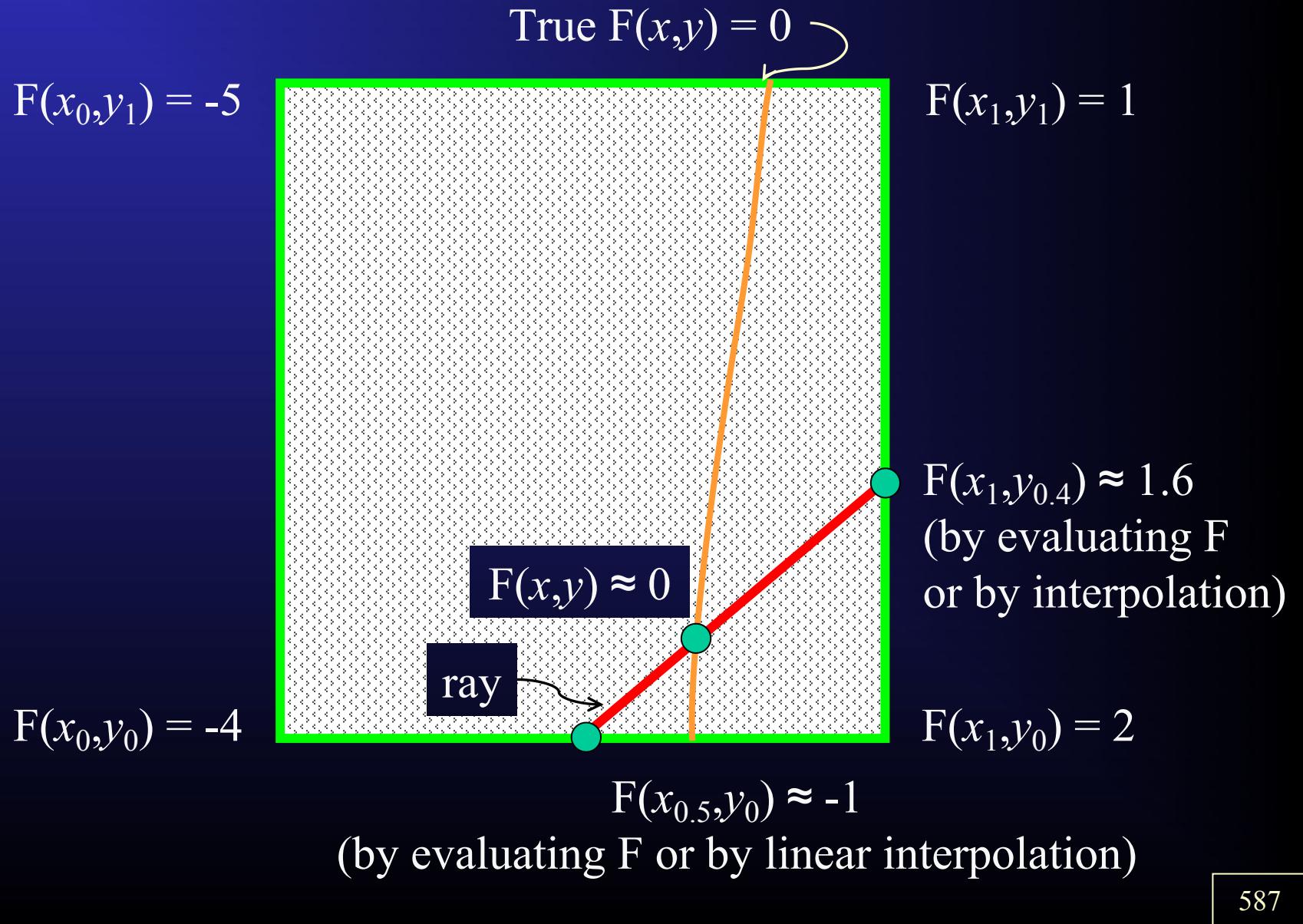
- Note that ray misses any intersection with the surface in the first **Unknown** cell; e.g.,
case 1:



- But crosses surface in the next **Unknown** cell;
e.g., case 8:



Extreme Close-Up Example (2D for clarity); Use Interpolation for Intersection Estimate



The Final Interpolation to Estimate Ray-Implicit Surface Intersection Point

- From the two edge F evaluations or interpolated values along the ray, we compute the approximate zero of the function F.
- Can do a few ways (binary search, numerical solve, etc.), but interpolation is simple.
- At one cell edge $F(x_a, y_a) = v_a$ is positive and the other cell edge $F(x_b, y_b) = v_b$ is negative.

$$F \cong 0 \quad \text{at} \quad (x_{interp}, y_{interp}) = \frac{v_a}{v_a - v_b} (x_a, y_a) - \frac{v_b}{v_a - v_b} (x_b, y_b)$$

- Check: $v_a = 1.6$; $v_b = -1$ so

$$\frac{1.6}{1.6 - (-1)} (x_{0.5}, y_0) - \frac{-1}{1.6 - (-1)} (x_1, y_{0.4}) = (x_{0.692..}, y_{0.230..})$$

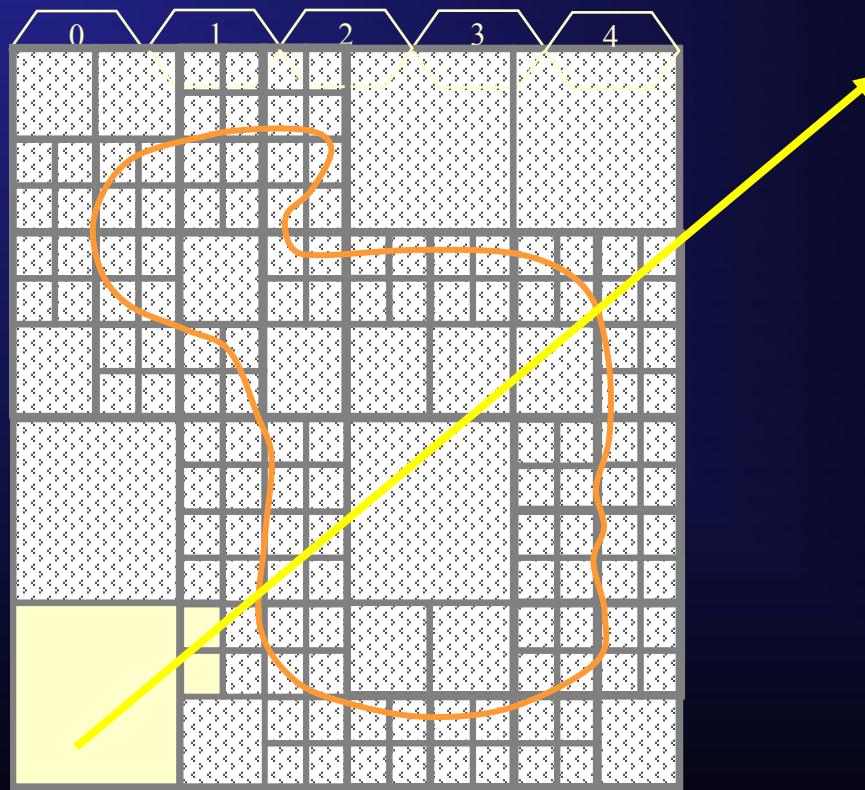
Adaptively Subdividing

- Hierarchical Spatial Subdivision is wasteful; it subdivides even in occluded regions of the domain.
- **Adaptive** Hierarchical Spatial Subdivision subdivides on demand:
 - Cells are marked **Unconsidered**
 - Ray marcher classifies an **Unconsidered** cell if and when it encounters one

Example Subdivision with **Unconsidered** Cells

-  **Full** (interior) cell
-  **Unknown** cell containing $F(p)=T$
-  **Empty** (outside) cell
-  **Unconsidered** cell

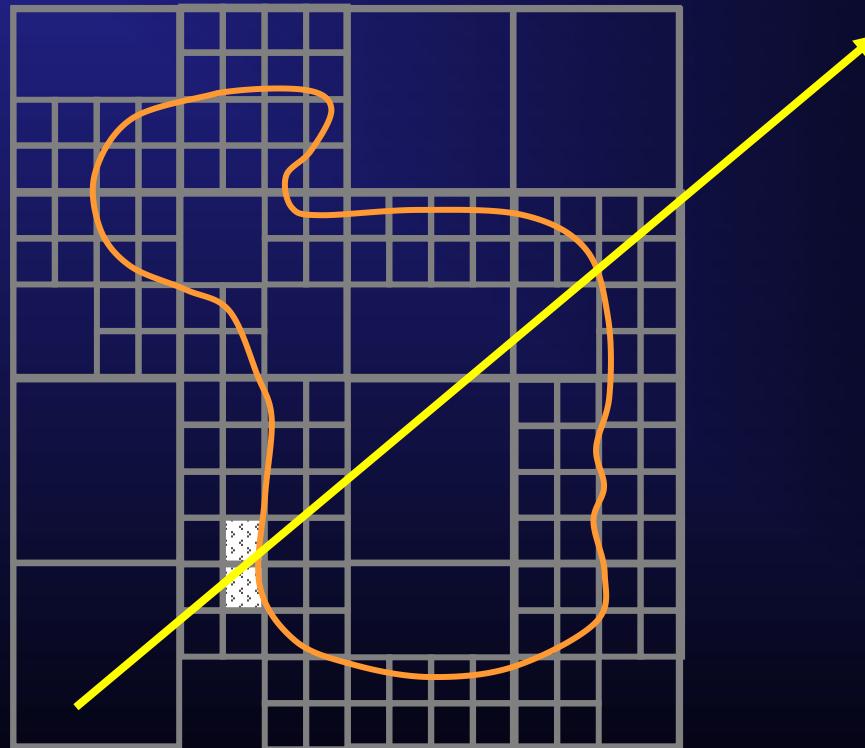
- Only expand **Unconsidered** cells at next level that intersect ray.



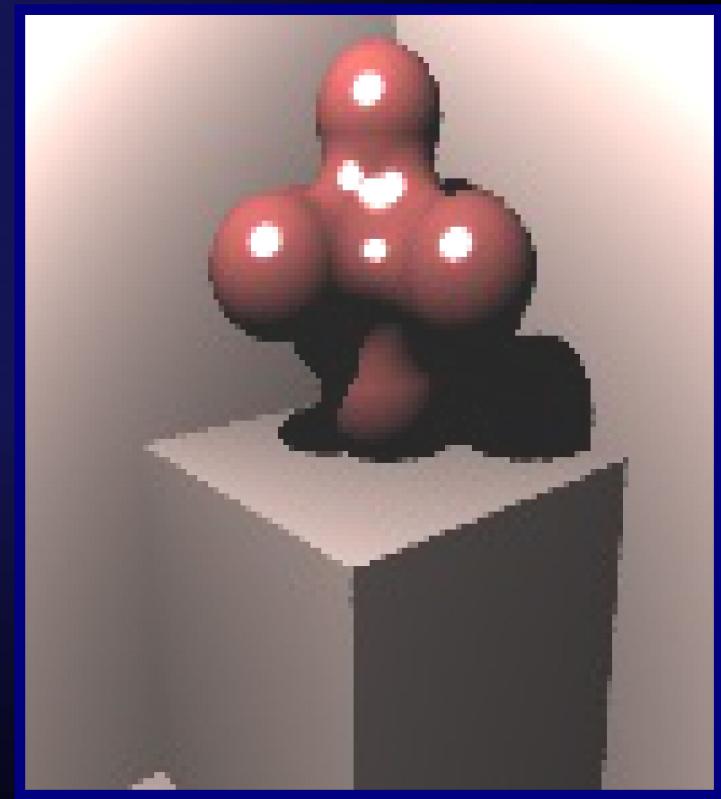
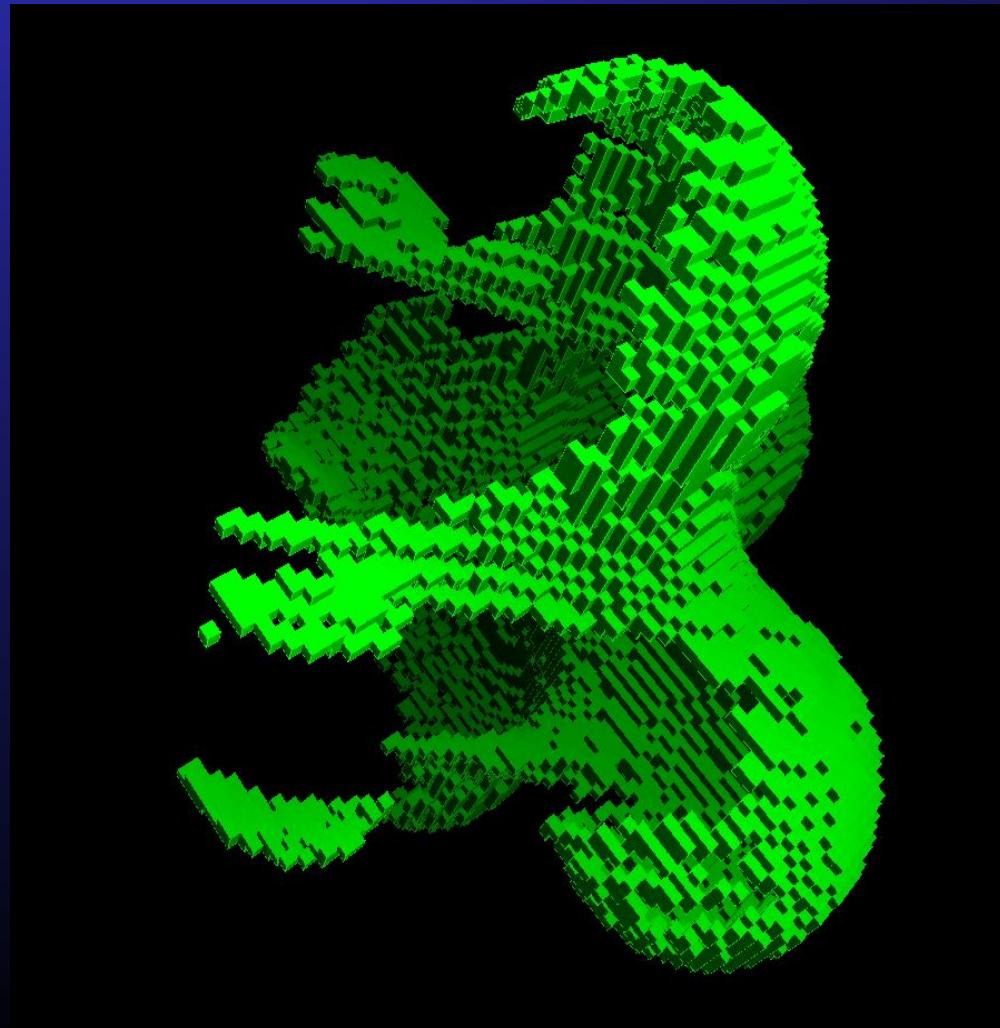
Example Subdivision with **Unconsidered** Cells

-  **Full** (interior) cell
-  **Unknown** cell containing $F(p)=T$
-  **Empty** (outside) cell
-  **Unconsidered** cell

- Only expand **Unconsidered** cells at next level that intersect ray.
- For this ray, that amounts to looking carefully at only about $2/256 \approx 1\%$ of the area.



Example On-Demand Adaptive Hierarchy



Ian Stewart, U. Waterloo

A General Rule in Computer Graphics

- *Throw away as much data as you can before you render a scene to avoid wasting expensive computation.*
- We just saw that in the adaptive cell subdivision.
- Rule used in many other situations. We'll look at visibility culling next.

Pre-Visibility Culling

- A family of techniques that attempt to cull as many invisible polygons BEFORE they are even sent into the rendering pipeline.
- Establish Potentially Visible Set (PVS) of polygons.
- Portal techniques for room-like structures.
- Pre-visibility from selected viewpoints.
- View frustum culling for general environments.
- Enhanced rendering performance, e.g., for games.
- Often combined with binary space partitioning...

Binary Space Partitioning

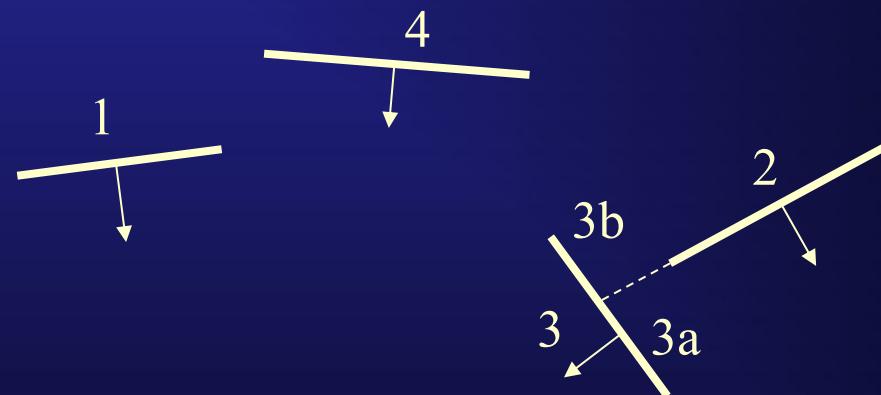
- There exist scenes in which visibility can be predetermined and is independent of view (camera) viewpoint.
- Basic idea: compute visibility in advance, then use this structure to pre-define the display ordering (back to front).
- Data structure built is called a *binary space partition* tree or *BSP-tree*.
- Construct separating planes iteratively as a preprocess, then traverse efficiently given an actual viewpoint.

Binary Space Partitioning -- Algorithm

- Select any polygon P from the set of polygons; place it at the root of a tree.
- If no polygons remain in list, done.
- Else for each polygon in the remaining polygon list, test which side of the plane of P it lies on:
 - If same side as viewpoint: put in the left (front) sub-tree list.
 - If opposite side as viewpoint: put in the right (back) sub-tree list.
 - If intersects the plane of P : split it into front and back pieces and insert pieces into their respective left and right sub-tree lists.
- Recurse on each sub-tree list.

BSP-Tree Generation

- Polygons shown edge-on for simplicity.
- Easily extends to 3D polygons.
- Arrow shows side toward viewpoint.



BSP – To Create View

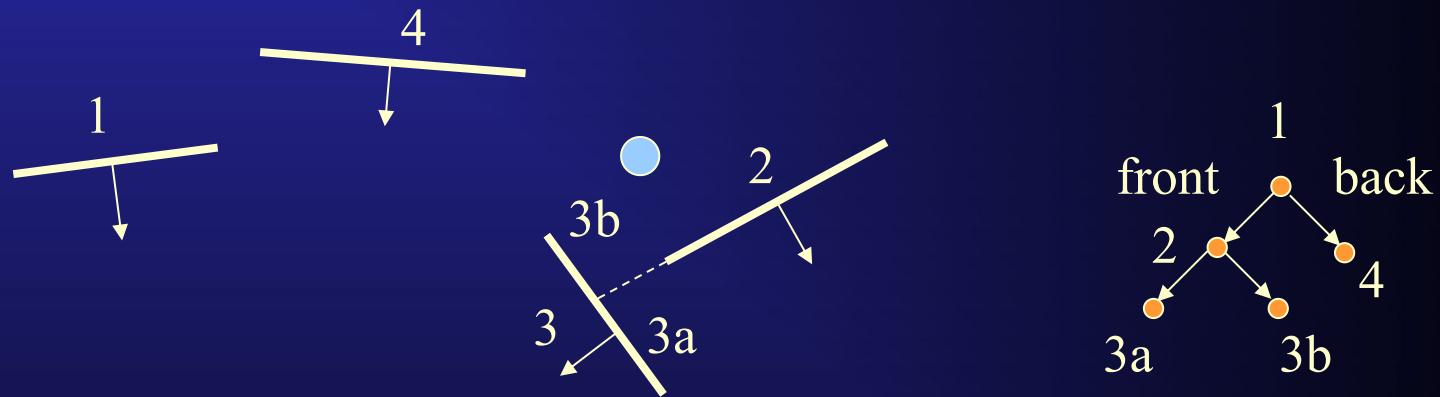
- To create view, traverse BSP-tree in in-order:

At each node of the BSP-(sub-)tree check which side of the root polygon the viewpoint is on:

- If the viewpoint is in **front** of the current root node polygon:
 - Traverse the **back** (right) sub-tree
 - **Output** the root polygon
 - Traverse the **front** (left) sub-tree
- If the viewpoint is in **back** of the current root node polygon:
 - Traverse the **front** (left) sub-tree
 - **Output** the root polygon
 - Traverse the **back** (right) sub-tree

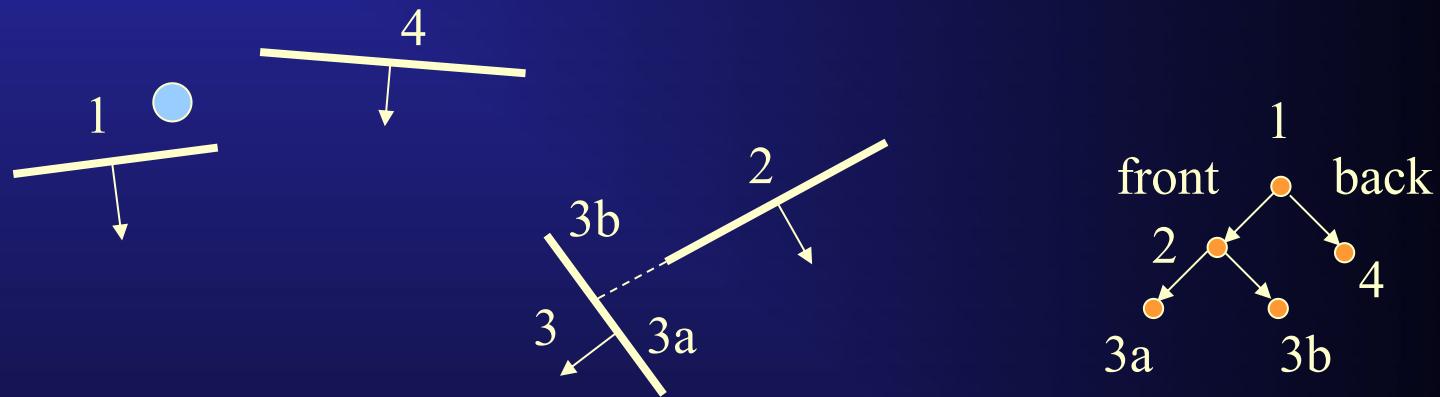
Examples

- Showing traversal for varying viewpoints:



Examples

- Showing traversal for varying viewpoints:



Output = 3a 2 3b 1 4

This is Basically the “DOOM” Graphics Engine that Revolutionized “First-Person” 3D Games!

- The BSP-tree is independent of viewpoint; its traversal is not; hence its display efficiency.
- Does not increase storage requirements of polygon dataset appreciably, if one can avoid excessive splits. (Notice that the polygons are just nodes in the BSP-tree.)
- So in practice, select a polygon that avoids splitting other polygons. (Remember, this is a preprocess, so time spent here is worth it.)
- Works best for static polygon environments, otherwise have to incrementally update BSP-tree dynamically.
- Try it yourself:

<http://symbolcraft.com/graphics/bsp/index.html>

Perspector data - do not edit

