Midterm Review

Course web page:

http://goo.gl/EB3aA

Midterm Notes

- Thursday, Mar. 22
- Worth 20% of your grade (like a homework+)
- Closed book, no calculators, no notes
- Focus on lecture material (up to & including z buffers from today)
  - Use readings for depth and understanding, but I won't go there for new topics (some topics, like clipping algorithms, are NOT IN TEXTBOOK)
    - Readings include everything in "Readings" column of Course Page schedule
- Question types: Mostly definitions, explanations; some calculations (e.g., transformations)
  - Sample exams from 2003, 2004
  - A few OpenGL-related questions will be on it. For example, "What is `gluLookAt()`'s place and function in the geometry pipeline?"

Midterm topics

- Basic OpenGL/GLUT
- Rasterization
- 2-D texturing, blending
- Simulation/particle systems
- Geometry
  - Transformations
  - Projections
- Clipping
- Hidden surface elimination

Simple OpenGL program

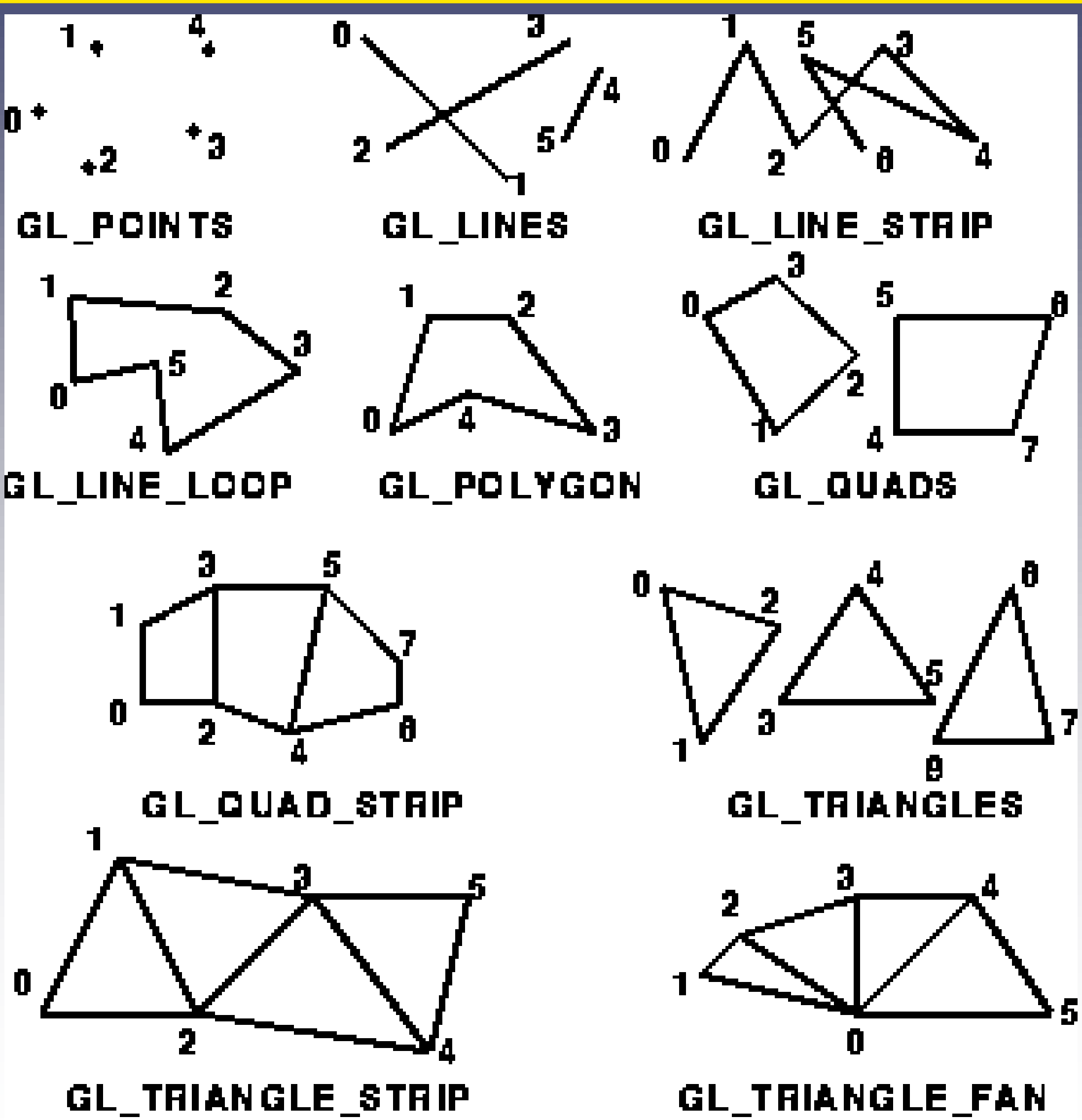
```
#include <stdio.h>
#include <GL/glut.h>

void main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
    glutInitWindowSize(100, 100);
    glutCreateWindow("hello");
    init();                      // set OpenGL states, variables

    glutDisplayFunc(display);      // register callback routines
    glutKeyboardFunc(keyboard);
    glutIdleFunc(idle);

    glutMainLoop();                // enter event-driven loop
}
```

OpenGL Geometric Primitives

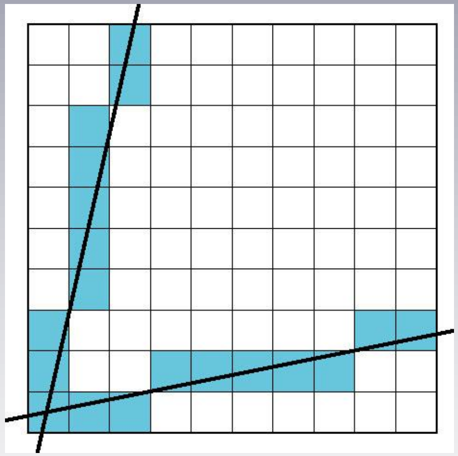


2-D Transformations: OpenGL

- 2-D transformation functions
  - **`glTranslate(x, y, 0)`**
  - **`glScale(sx, sy, 0)`**
  - **`glRotate(theta, 0, 0, 1)`** (angle in degrees; direction is counterclockwise)
- No shear, no reflection built into OpenGL
- Using **`glPushMatrix()`**/**`glPopMatrix()`** to isolate transformations

Rasterization: What is it?

- How to go from real numbers of geometric primitives' vertices to integer coordinates of pixels on screen
- Geometric primitives
  - Points: Round vertex location in coordinates
  - Lines: Can do this for endpoints, what about in between?
  - Polygons: How to fill area bounded by edges?

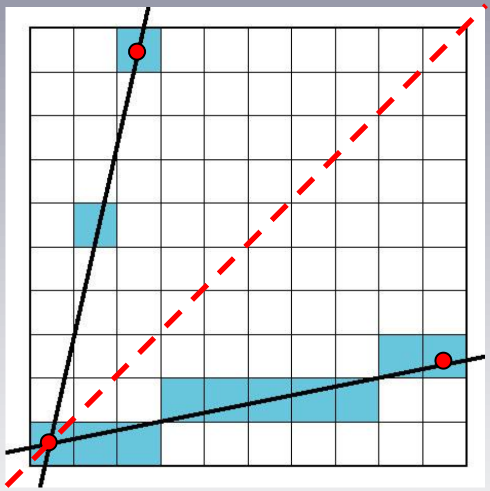


from Angel

UNIVERSITY OF DELAWARE

DDA/Parametric Line Drawing

- DDA stands for Digital Differential Analyzer, the name of a class of old machines used for plotting functions
- Slope-intercept form of a line: **`y = mx + b`**
  - **`m = dy/dx`**
  - **`b`** is where the line intersects the Y axis
- DDA's basic idea: If we increment the **`x`** coordinate by 1 pixel at each step, the slope of the line tells us how to much increment **`y`** per step



from Angel

  - I.e., **`m = dy/dx`**, so for **`dx = 1`**, **`dy = m`**
  - This only works if **`m <= 1`**—otherwise there are gaps
    - Solution: Reverse axes and step in Y direction. Since now **`dy = 1`**, we get **`dx = 1/m`**

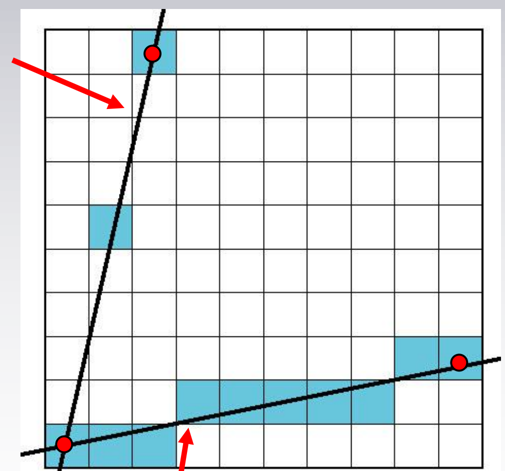
Midpoint line drawing: Line equation

- Recall that the slope-intercept form of the line is
  $y = (dy/dx)x + b$
- Multiplying through by dx, we can rearrange this in **implicit form**:

$$F(x, y) = dy\ x - dx\ y + dx\ b = 0$$

- F is:
  - Zero for points on the line
  - Positive for points **below** the line (**right** if slope > 1)
  - Negative for points **above** the line (**left** if slope > 1)
    - Examples: (0, 1), (1, 0), etc.

9x - 2y = 0

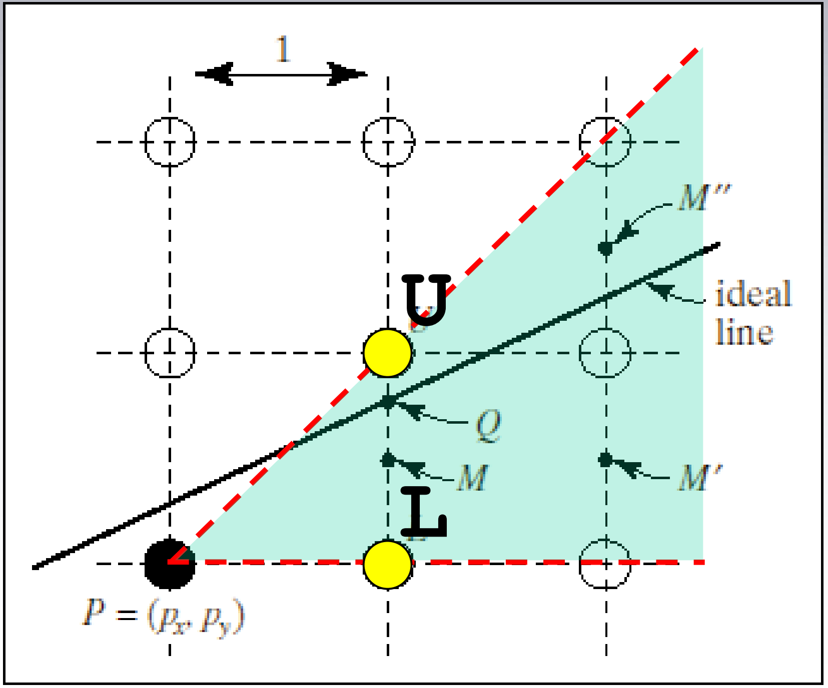


from Angel

2x - 9y = 0

Midpoint line drawing: The Decision

- Given our assumptions about the slope, after drawing **`(x, y)`** the only choice for the next pixel is between the upper pixel **`U = (x+1, y+1)`** and the lower one **`L = (x+1, y)`**
- We want to draw the one closest to the line



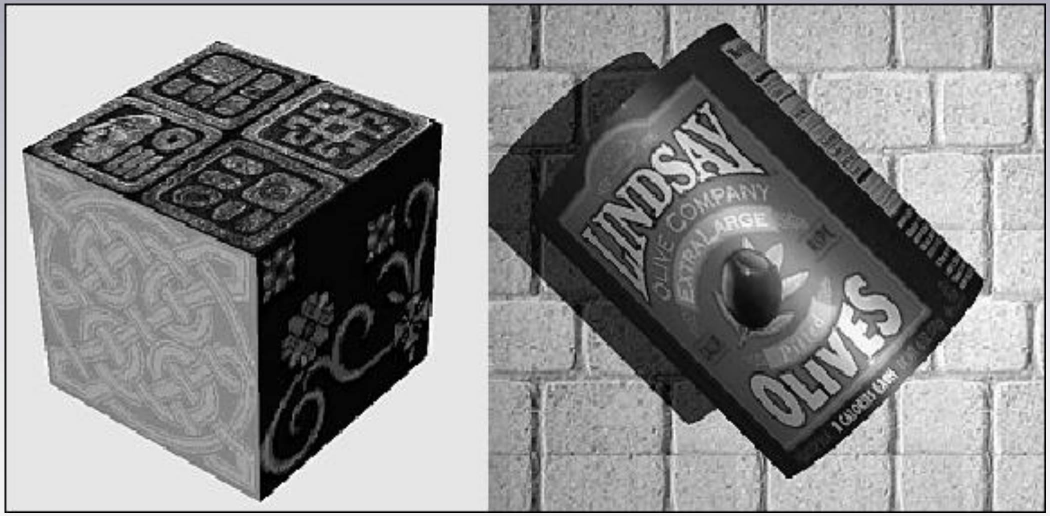
from Hill

Rasterizing triangles

- Special case of polygon rasterization
  - Exactly two **active** edges at all times
- One method:
  - Fill scanline table between top and bottom vertex with leftmost and rightmost side by using DDA or midpoint algorithm to follow edges
  - Traverse table scanline by scanline, fill run from left to right

What is Texture Mapping?

- Spatially-varying modification of surface appearance at the pixel level
- Characteristics
  - Color
  - Shininess
  - Transparency
  - ***Bumpiness***
  - Etc.



from Hill

- **“Sprite”** when on polygon with no 3-D

Texture mapping applications: Billboards

from Akenine-Moller & Haines

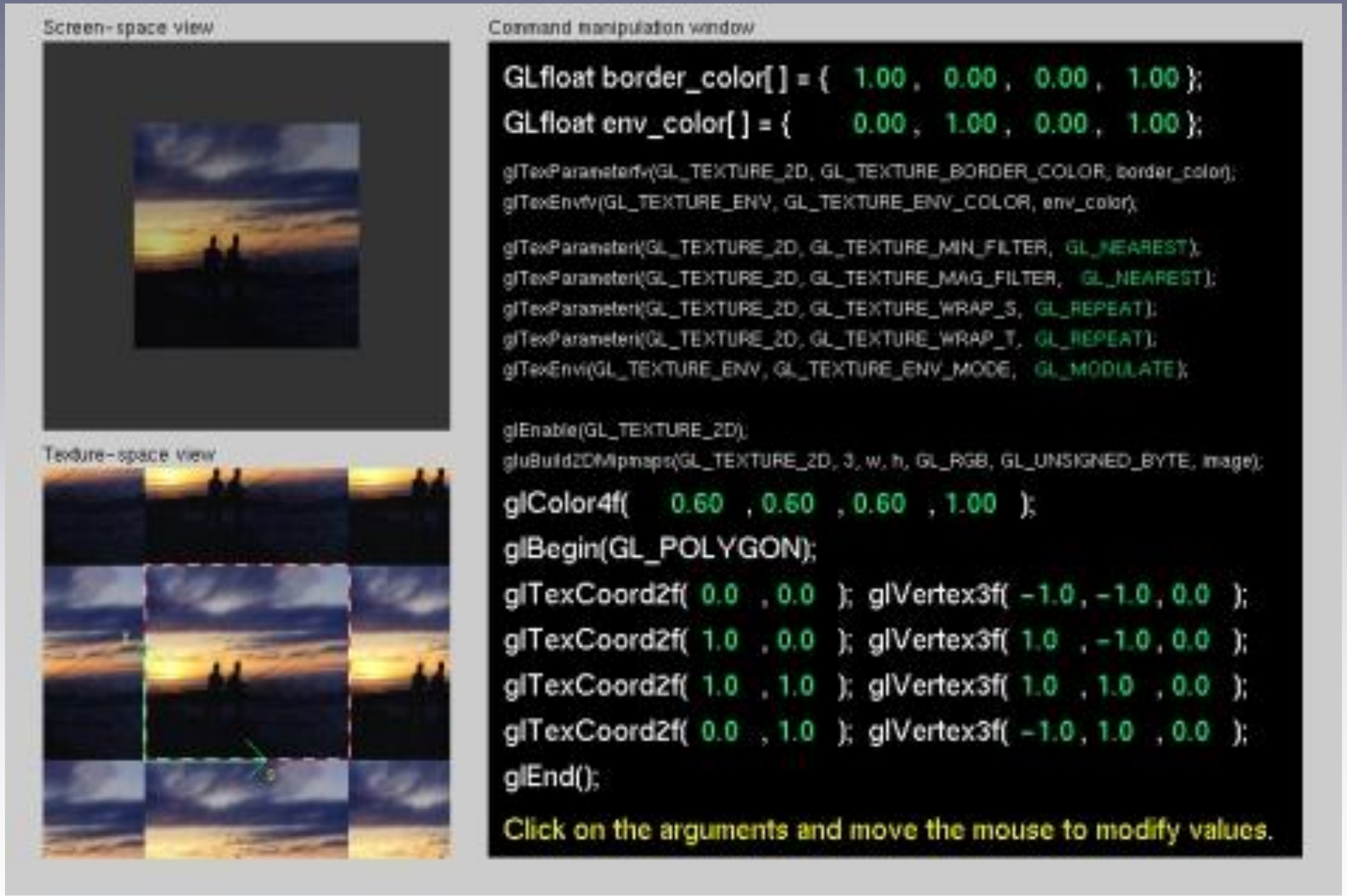
from www.massal.net/projects

Also called "impostors": Image aligned polygons in 3-D

OpenGL texturing steps (Red book)

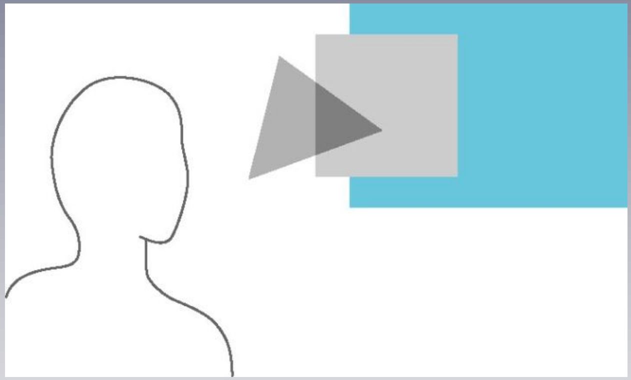
1. Create a texture object and specify a texture for that object
2. Indicate how the texture is to be applied to each pixel
3. Enable texture mapping with **`glEnable(GL_TEXTURE_2D)`**
4. Draw the scene, supplying both texture and geometric coordinates

Robins' texture tutor (aka details of Sprite class)



Compositing

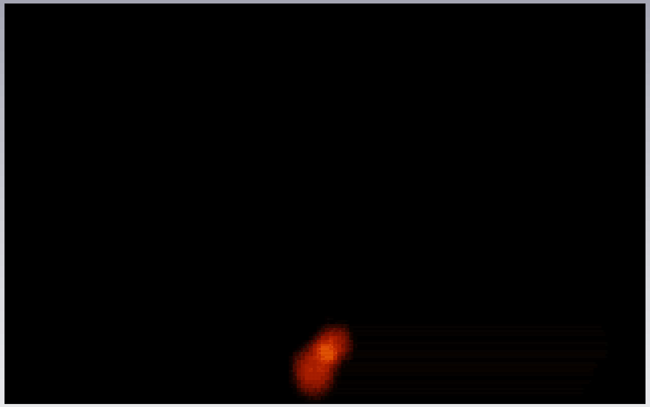
- When a pixel is drawn to a buffer, what happens to what's already there?
- Normally, we just overwrite… but there are more options
- Different operations
  - Blending: Use alpha channel to transparency vs. opacity
    - alpha = 1 -> Perfect opacity (default)
    - alpha = 0 -> Perfect transparency
    - In between, pixel is a mix of source and destination colors
- We did not cover details of **`glBlendFunc()`**, so it won't be on exam



from Angel

Particle Systems

- Definition: Simulation of a set of similar, moving agents in a larger environment
- Scale usually such that aggregate motion of "swarm" is more apparent than internal agent motion
- Applications
  - Water, snow
  - Smoke, fire
  - Cloth
  - "Creatures"
- Basic loop:
  1. Create, kill particles
  2. Update positions based on:
     - Previous positions, velocities, accelerations
     - Exterior and interior forces
  3. Render particles

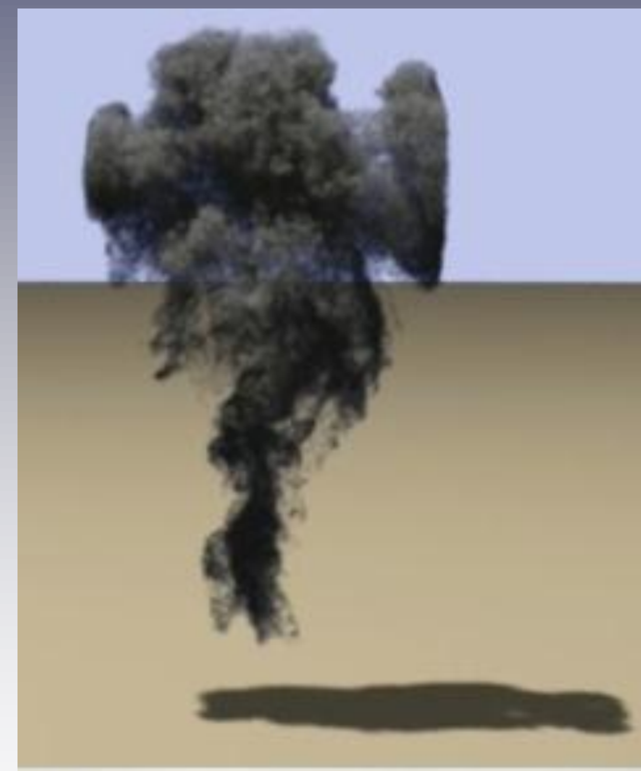


courtesy of S. Dunn

Fire: Each particle is a blended sprite

Particle/Agent Motion Factors

- Global exterior forces such as gravity, wind, pre-determined path, target position, etc.
- Interactions with fixed environment
  - Collisions
  - Friction
- Physical interactions with each other
  - Gravity, electrical attraction/repulsion
  - Spring connections
  - Collisions
- Interior "self determination"
  - Randomness
  - AI-like perception-action feedback
    - Flocking, seeking with collision avoidance, etc.



Smoke movie (turbulence…): Convection + invisible container = smoke in a bottle

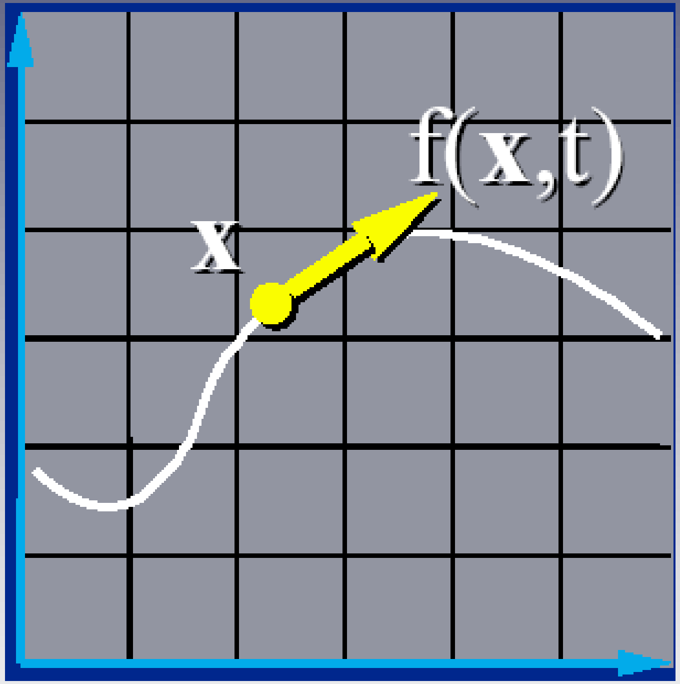
Initial upward and outward velocity + gravity = water fountain

Particle Update

- Given particle state at time t consisting of position, velocity, etc., how do we compute new values at time $\mathbf{x}(t + \Delta t)$?
- Typically, we don't have an explicit parametric function $\mathbf{x}(t)$ that we can just evaluate for any t
  - E.g., a spline curve
- Rather, we have a **set of forces** and an **initial value** for the particle state
- We have to simulate the action of the forces on the particle to "see what happens"!
- In practice, this means numerical methods for solving ordinary differential equations (ODEs)

Ordinary Differential Equations

- Consider points on **unknown** parametric curve **x**(t) with **known** derivatives (i.e., tangents) f(**x**, t)



from A. Witkin's SIGGRAPH course notes

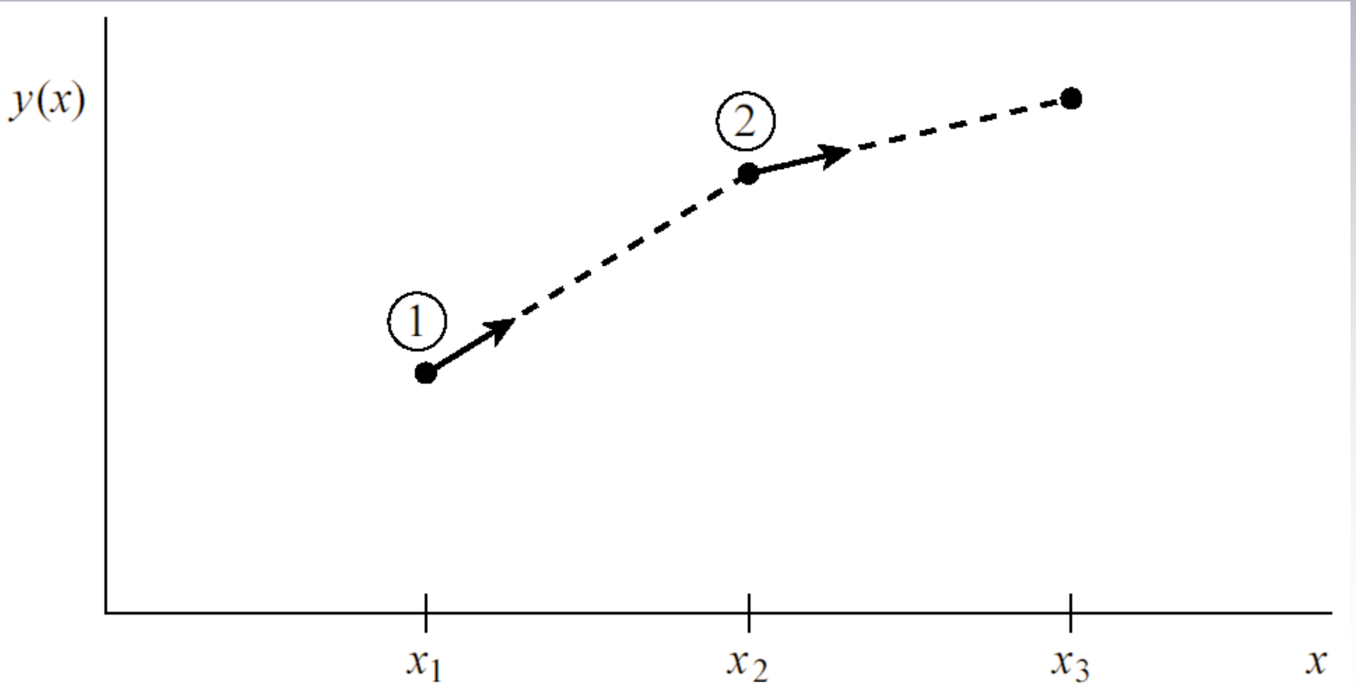
$$\frac{\partial \mathbf{x}(t)}{\partial t} = \dot{\mathbf{x}}(t) = f(\mathbf{x}, t)$$

different ways of writing derivative

Euler Integration

- First order (linear) approximation using a **step size** of Δt :

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t\, f(\mathbf{x}, t)$$



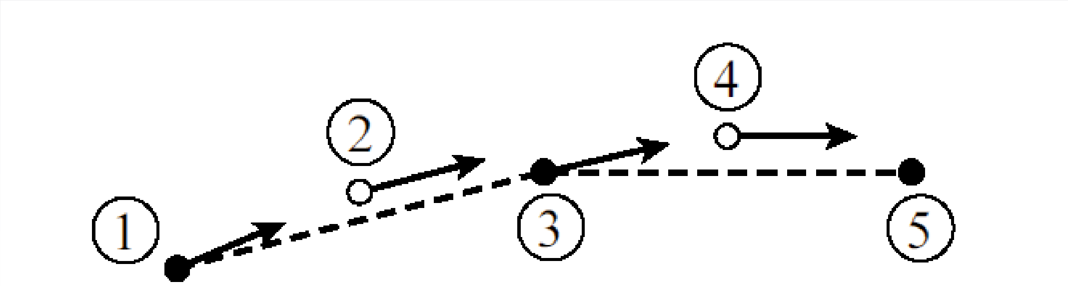
Midpoint/RK2 method: Steps

1. Compute Euler step $\Delta\mathbf{x} = \Delta t\, f(\mathbf{x}, t)$
2. Evaluate first derivative at midpoint (half step)

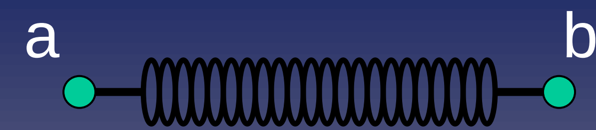
$$f_{\text{mid}} = f(\mathbf{x}(t) + \frac{\Delta \mathbf{x}}{2}, t + \frac{\Delta t}{2})$$

3. Take full step using midpoint derivative

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t\, f_{\text{mid}}$$



n-ary Forces: Springs



- 2 connected particles *a* and *b* exert force on one another proportional to displacement from **resting length** r of spring
- Assuming time t, let $\Delta\mathbf{x} = \mathbf{x}_a - \mathbf{x}_b$, $\mathbf{d} = \Delta\mathbf{x}/|\Delta\mathbf{x}|$, and $\Delta\mathbf{v} = \mathbf{v}_a - \mathbf{v}_b$. Then the force on *a* is (where $\mathbf{f}_b = -\mathbf{f}_a$):

$$\mathbf{f}_a = -\left[k_s(||\Delta\mathbf{x}|| - r) + k_d \Delta\mathbf{v}\cdot\mathbf{d}\right]\mathbf{d}$$

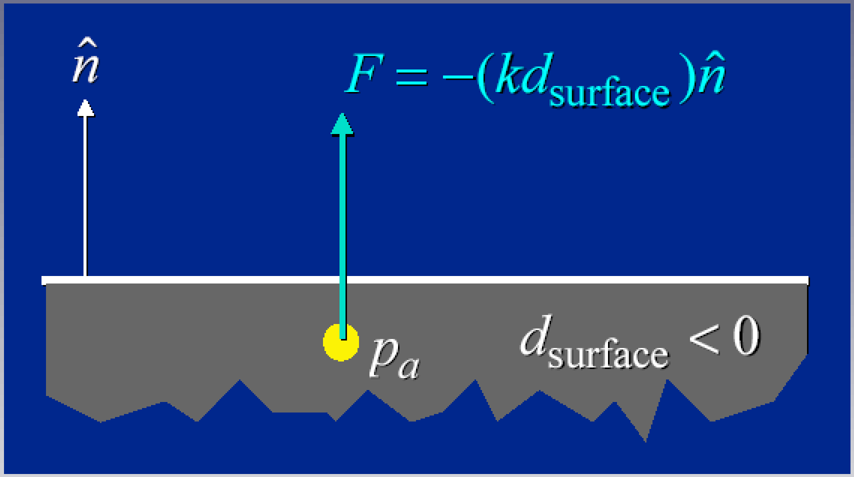
spring constant ("stiffness")

damping constant (like "spring drag")

See molecule examples at http://www.myphysicslab.com

Collisions

- Penalty method
  - Spring-like force proportional to penetration distance pushes particle out object interior



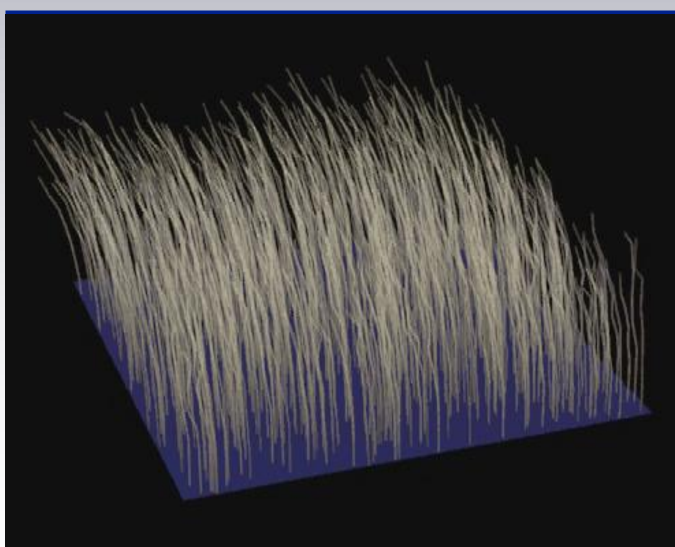
- Hard collisions
  - Detect intersection point explicitly, treat like a reflection (note that initial velocity vector points toward surface):

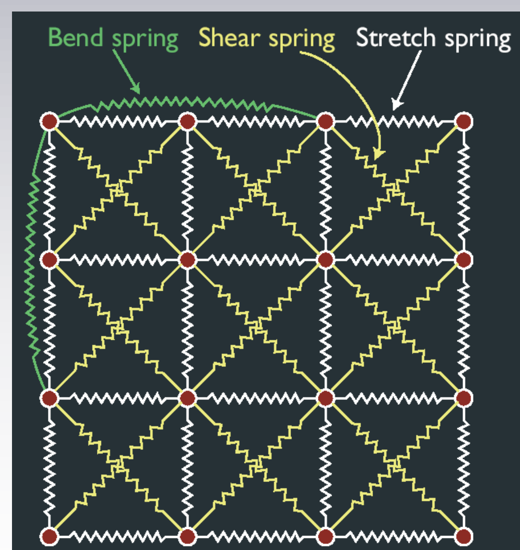
$$\mathbf{v}' = \mathbf{v} - 2(\mathbf{n} \cdot \mathbf{v})\mathbf{n}$$

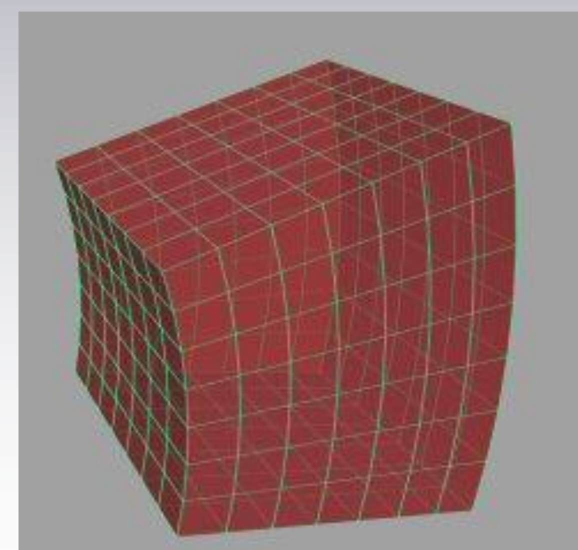
See collision examples at http://www.myphysicslab.com

Spring systems

- Networks of particles connected by springs can be used to simulate objects with elastic properties
  - 1-D: Rope, hair, grass
  - 2-D: Cloth
  - 3-D: Deformable (aka rubber) objects

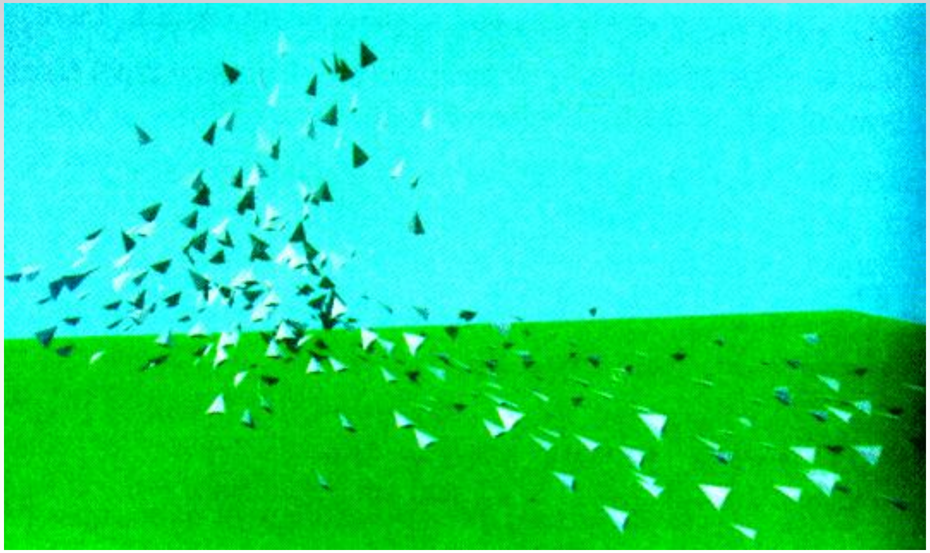






Flocking (C. Reynolds, SIGGRAPH 1987)

- Particles for simulating simple creatures: **boids**
  - Birds in flock
  - Fish in school
  - Etc.
- Not passive—forces are internally generated
  - Can be combined with external forces
- "Intentions" of each boid depend on characteristics of local environment



2-D & 3-D Transformations

- Types
  - Translation
  - Scaling
  - Rotation
  - Shear, reflection
- Mathematical representation as matrices when points are in homogeneous coordinates

Homogeneous Coordinates

- Note: write vectors vertically instead of horizontally
- Let $\mathbf{x} = (x_1, \ldots, x_n)^T$ be a point in Euclidean space
- Change to *homogeneous* coordinates:

$$\mathbf{x} \dashrightarrow (\mathbf{x}^T, 1)^T$$

- Defined up to scale (think of as all points on a ray and w as how far along the ray):

$$(\mathbf{x}^T, 1)^T = (w\mathbf{x}^T, w)^T$$

- Can go back to non-homogeneous representation by **normalizing** as follows:

$$(\mathbf{x}^T, w)^T \dashrightarrow \mathbf{x}/w$$

3-D Rotation Matrices

- Similar form to 2-D rotation matrices, but with coordinate corresponding to rotation axis held constant
- E.g., a rotation about the X axis of θ radians:

$$\mathbf{R}_X(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3-D Rigid Transformations

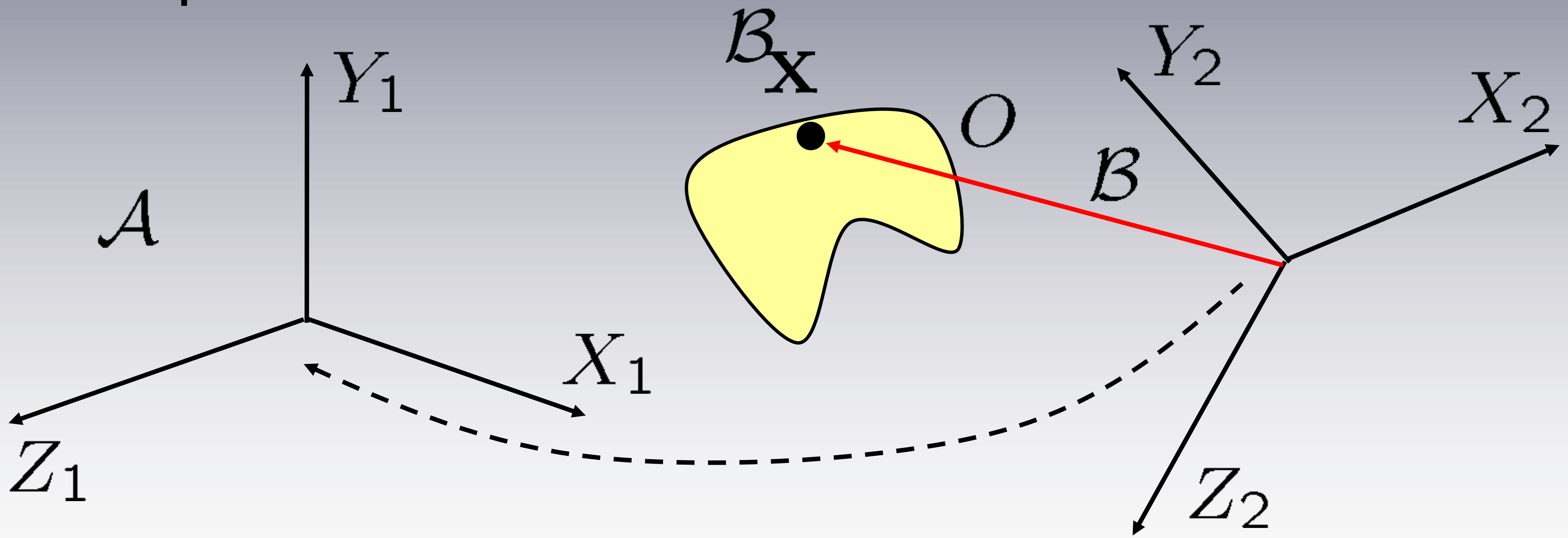
- Combination of rotation followed by translation, without scaling, etc.
- "Moves" an object from one 3-D pose to another

$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & \Delta x \\ r_{21} & r_{22} & r_{23} & \Delta y \\ r_{31} & r_{32} & r_{33} & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

T **R** **M**

3-D Transformations: Arbitrary Change of Coordinates

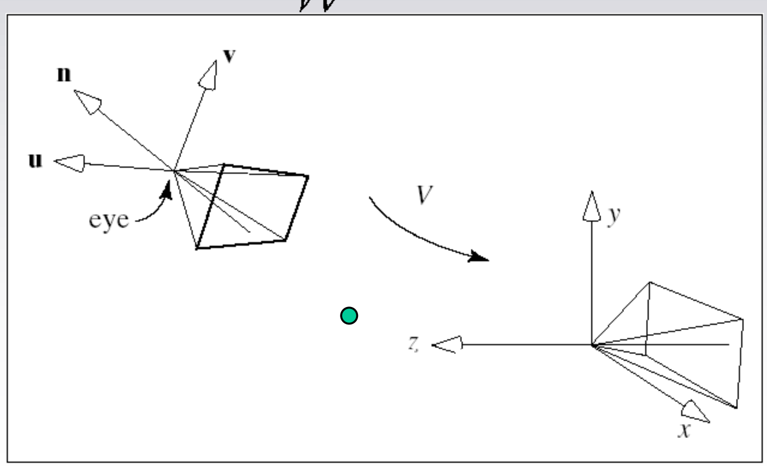
- A rigid transformation can be used to represent a change in the coordinate system that "expresses" a point's location



$$\boxed{{}^{\mathcal{B}}\mathbf{x} = {}^{\mathcal{B}}_{\mathcal{A}}\mathbf{R}\,{}^{\mathcal{A}}\mathbf{x} + {}^{\mathcal{B}}\mathbf{o}_{\mathcal{A}}}$$

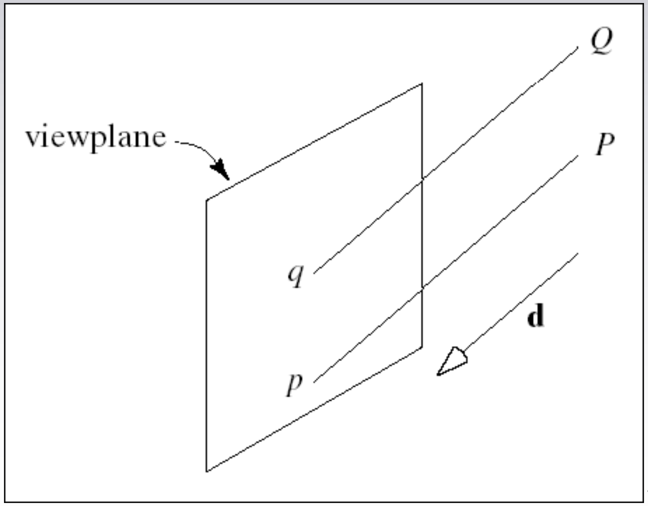
`gluLookAt()`: Details

- Moves scene points so that camera is at origin, "look at" point is on -Z axis, and camera +Y axis is aligned with **up** vector
  – Create and execute rigid transformation ${}^{\mathcal{C}}_{\mathcal{W}}\mathbf{M}$ making a change from world to camera coordinates
- Steps
  1. Compute vectors **u**, **v**, **n** defining new **camera axes** in **world coordinates**
     - "Old" axes are $\mathbf{u} = (1, 0, 0)^T$, $\mathbf{v} = (0, 1, 0)^T$, $\mathbf{n} = (0, 0, -1)^T$
  2. Compute location ${}^{\mathcal{C}}\mathbf{o}_{\mathcal{W}}$ of old camera position in terms of new location's coordinate system
  3. Fill in rigid transform matrix ${}^{\mathcal{C}}_{\mathcal{W}}\mathbf{M}$



Transformations vs. Projections

- **Transformation**: Mapping within n-D space
  - Moves points around, effectively warping space
- **Projection**: Mapping from n-D space down to lower-dimensional subspace
  - E.g., point in 3-D space to point on plane (a 2-D entity) in that space
  - We will be interested in such 3-D to 2-D projections where the plane is the **image**

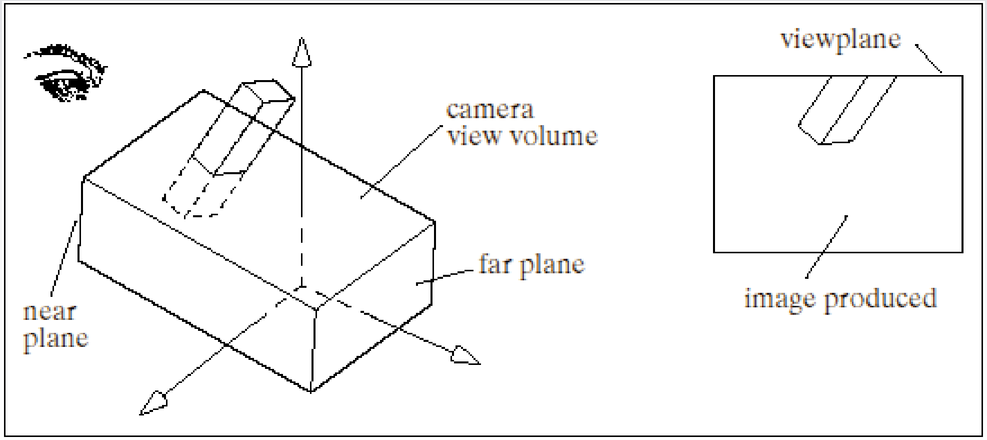


from Hill

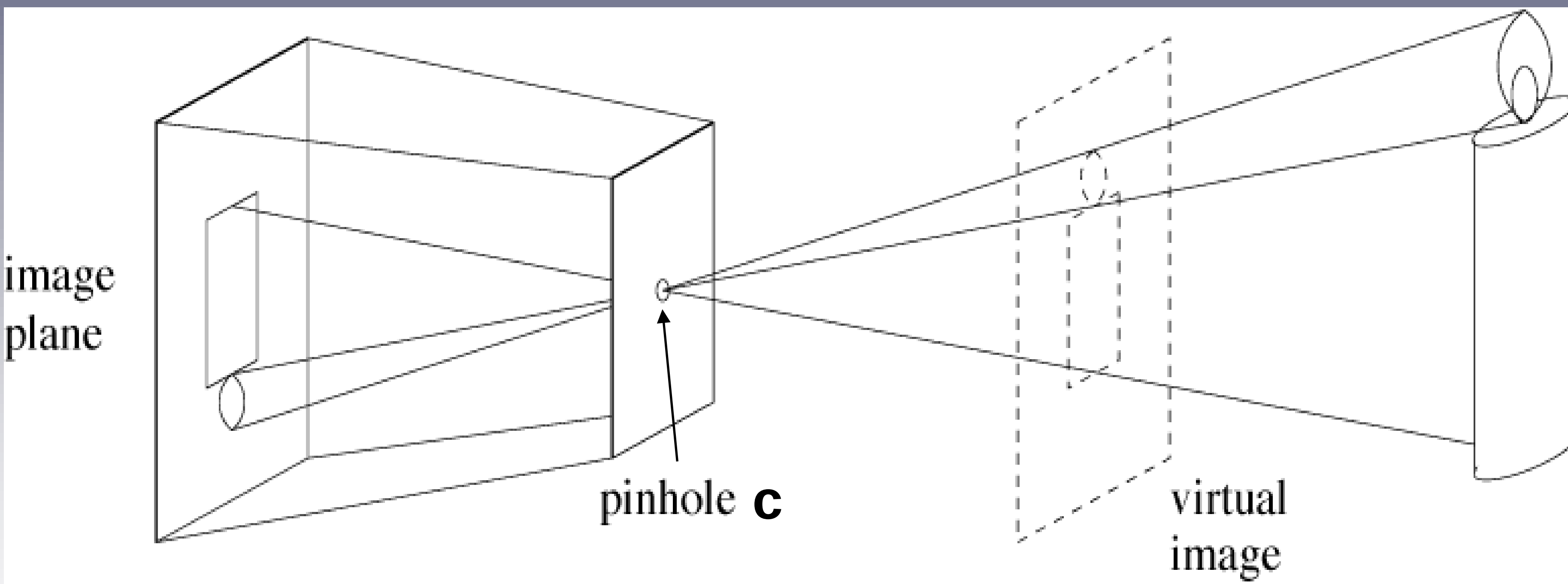
Parallel projection along direction **d** onto a plane

Orthographic Projection

- Projection direction **d** is aligned with Z axis
- Viewing volume is "brick"-shaped region in space
  - Not the same as image size
- No perspective effects—distant objects look same as near ones, so camera (x, y, z) $\Rightarrow$ image (x, y)



Perspective with a Pinhole Camera (i.e., no lens)



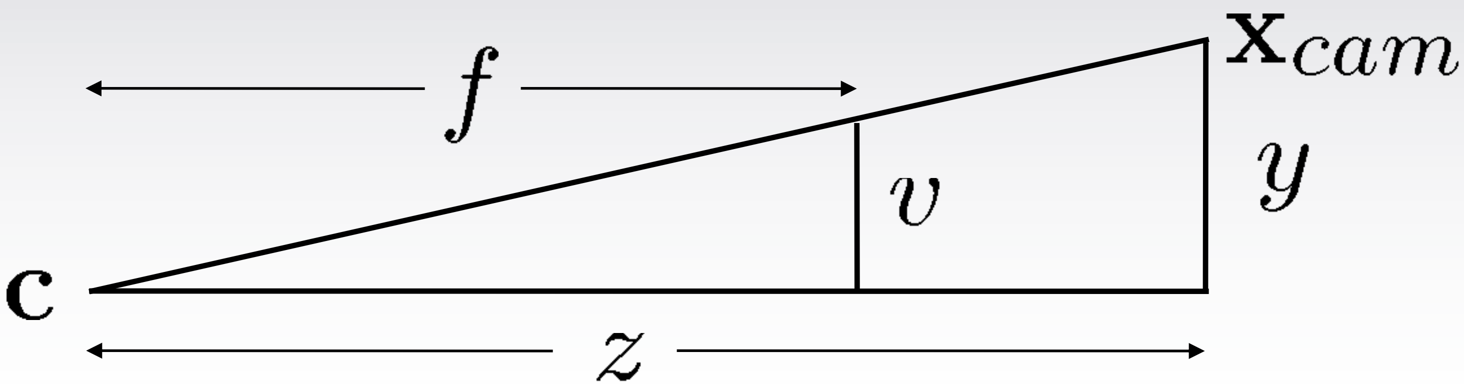
from Forsyth & Ponce

Instead of single direction **d** characteristic of parallel projections, rays emanating from single point **c** define perspective projection

Perspective Projection

- Letting the camera coordinates of the projected point be $\mathbf{x}_{cam} = (x, y, z)^T$ leads by similar triangles to:

$$\mathbf{x}_{im} = \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} fx/z \\ fy/z \end{pmatrix}$$



Perspective Projection Matrix

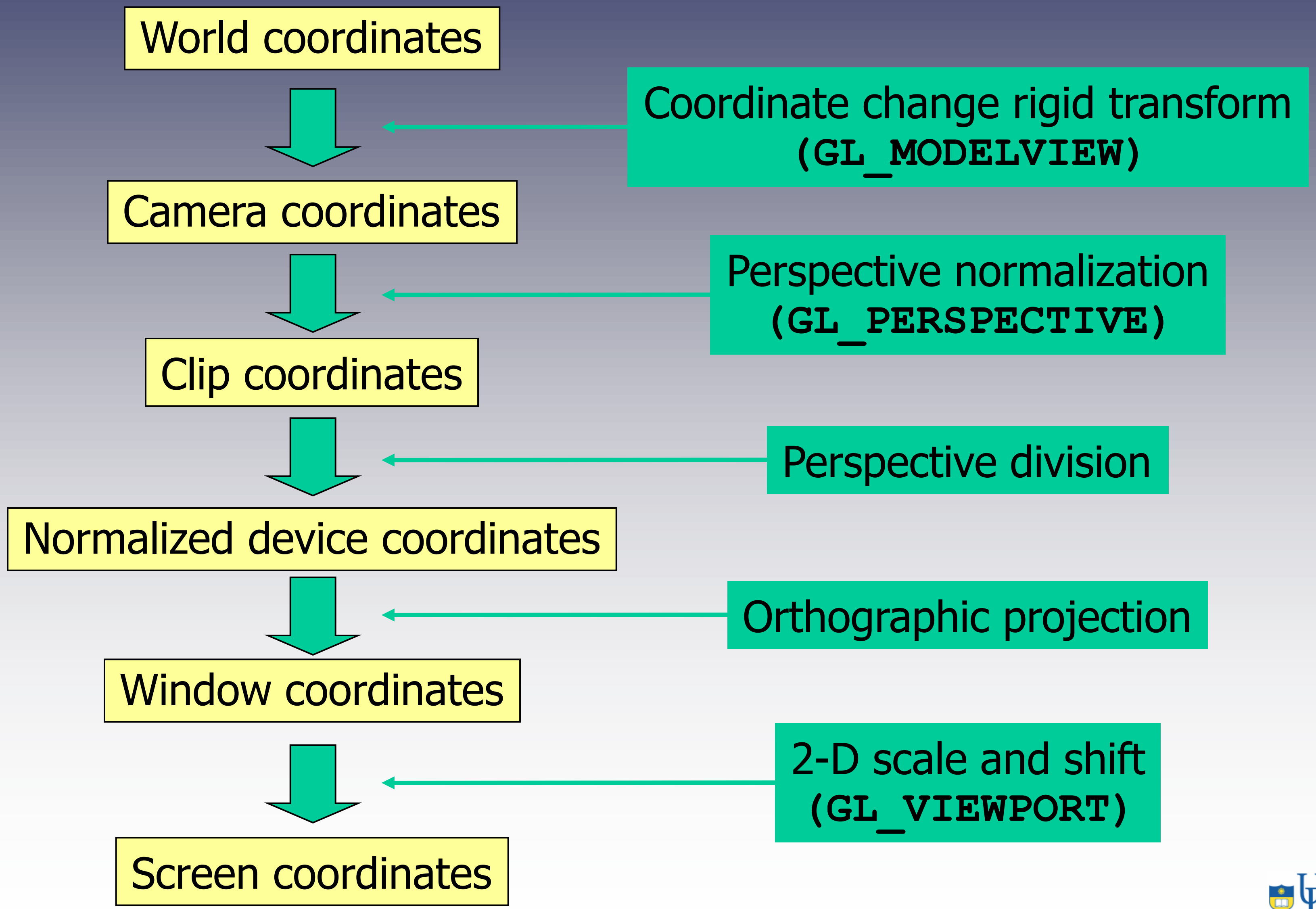
- Using homogeneous coordinates, we can describe a perspective transformation with the image plane at *z = -f* (because *f > 0* but *z < 0*) via a 4 x 4 matrix multiplication:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/f & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ -z/f \end{pmatrix} \rightarrow \begin{pmatrix} -fx/z \\ -fy/z \\ -f \\ 1 \end{pmatrix}$$

Last step accomplishes distance-dependent scaling by the rule for converting between homogeneous and regular coordinates. This is called the **perspective division**

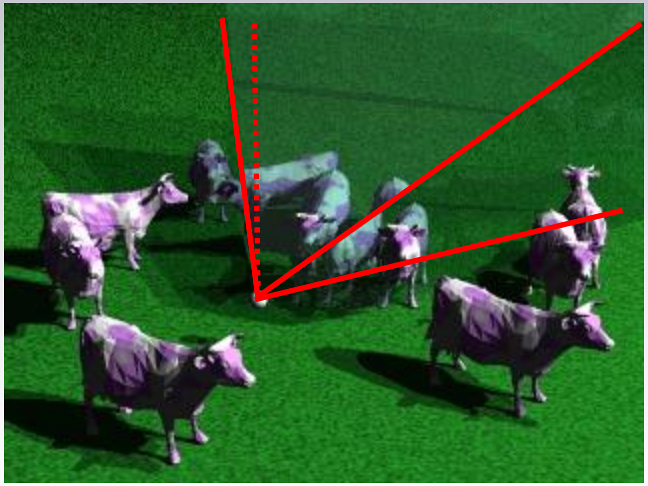
- Actual matrix has additional terms to scale everything to CVV
- Now just do orthographic projection to image coordinates
  - After any steps that require **depth information**

Geometry pipeline



Clipping

- Removal of portions of geometric primitives outside viewing volume (VV)
- Why?
  - Optimization that saves computation which would otherwise be wasted on lighting, texturing, etc.
- Cases
  - Trivial acceptance: Complete inside VV
  - Trivial rejection: Completely outside VV
  - Crossing **clip plane(s)**: Partially outside, so must trim to fit
- Different primitives require different methods
  - Points: Only trivial accept/reject
  - Lines: Chop at intersection with clip plane
  - Polygons: Must trim so as to maintain connectivity

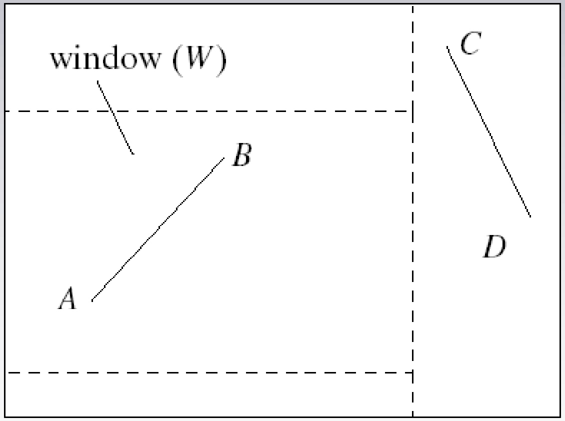


courtesy of L. McMillan

Cohen-Sutherland clipping

- Outcodes partition plane around the viewing area
- Trivial line clipping cases
  - **Accept** line ($\mathbf{p}_1$, $\mathbf{p}_2$): Both endpoints are inside the rectangle
    - In terms of outcodes, this means o($\mathbf{p}_1$) = **FFFF** and o($\mathbf{p}_2$) = **FFFF**
  - **Reject** line: Both endpoints outside rectangle **on same side**
    - This means both points' outcodes have a **T** at the same bit position—e.g., o($\mathbf{p}_1$) = **FTTF** and o($\mathbf{p}_2$) = **FFTF**

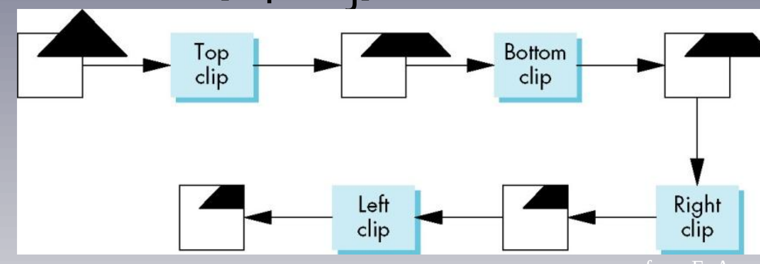
TTFF	FTFF	FTTF
TFFF	FFFF window	FFTF
TFFT	FFFT	FFTT



from Hill

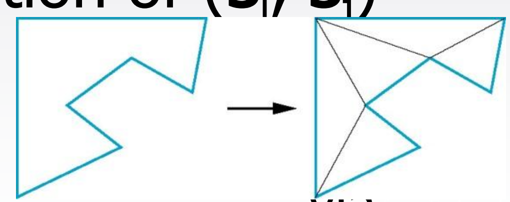
Sutherland-Hodgman 2-D polygon clipping

- Algorithm for clipping polygon S (convex or not) against a convex clip polygon C
- Basic approach: Clip S against each side ($\mathbf{c}_i$, $\mathbf{c}_j$) of C in sequence
  - Traverse vertex list of S clipped vertex list
  - For each old edge ($\mathbf{s}_i$, $\mathbf{s}_j$) of S:
    - If both inside ($\mathbf{c}_i$, $\mathbf{c}_j$): Output new $\mathbf{s}_j$
    - If both outside: Output nothing
    - If $\mathbf{s}_i$ inside, $\mathbf{s}_j$ outside: Output intersection of ($\mathbf{s}_i$, $\mathbf{s}_j$) with ($\mathbf{c}_i$, $\mathbf{c}_j$)
    - If $\mathbf{s}_i$ outside, $\mathbf{s}_j$ inside: Output intersection of ($\mathbf{s}_i$, $\mathbf{s}_j$) with ($\mathbf{c}_i$, $\mathbf{c}_j$), then $\mathbf{s}_j$



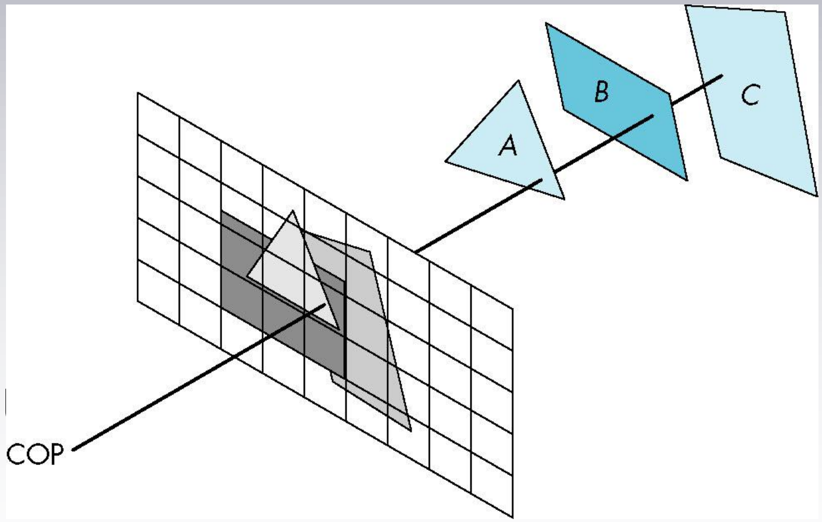
from E. Angel

- Simple case: S is a triangle, C a rectangle
  - Can convert arbitrary polygon to set of triangles via **tesselation** (e.g., `gluTess*()` functions)



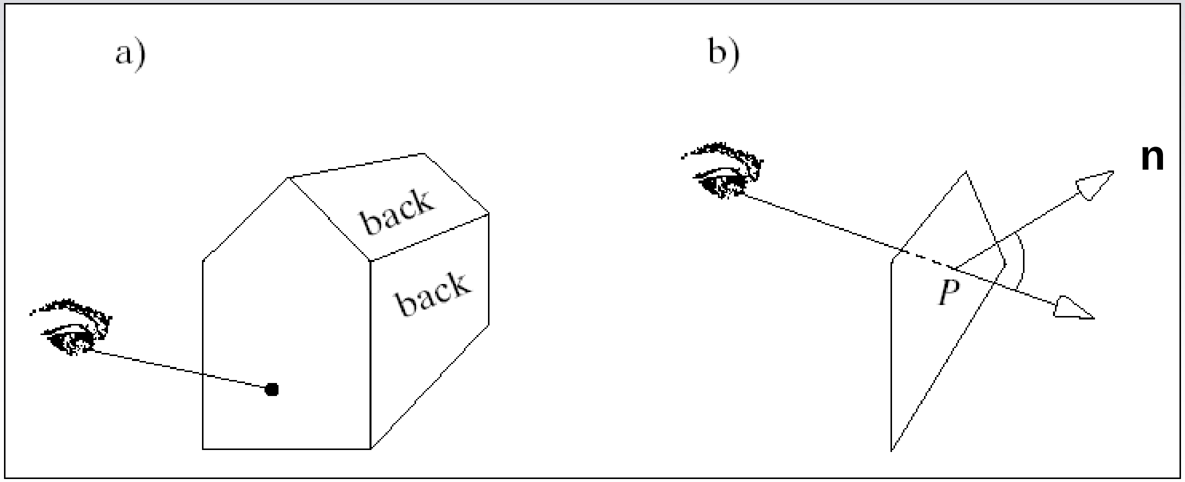
Hidden Surfaces: Why care?

- Hidden surfaces are objects **inside** the viewing volume that should not be seen
- **Occlusion**: Closer (opaque) objects along same viewing ray obscure more distant ones
- Reasons to remove
  - Efficiency: As with clipping, wasting work on
  - Correctness: The image will we don't model occl



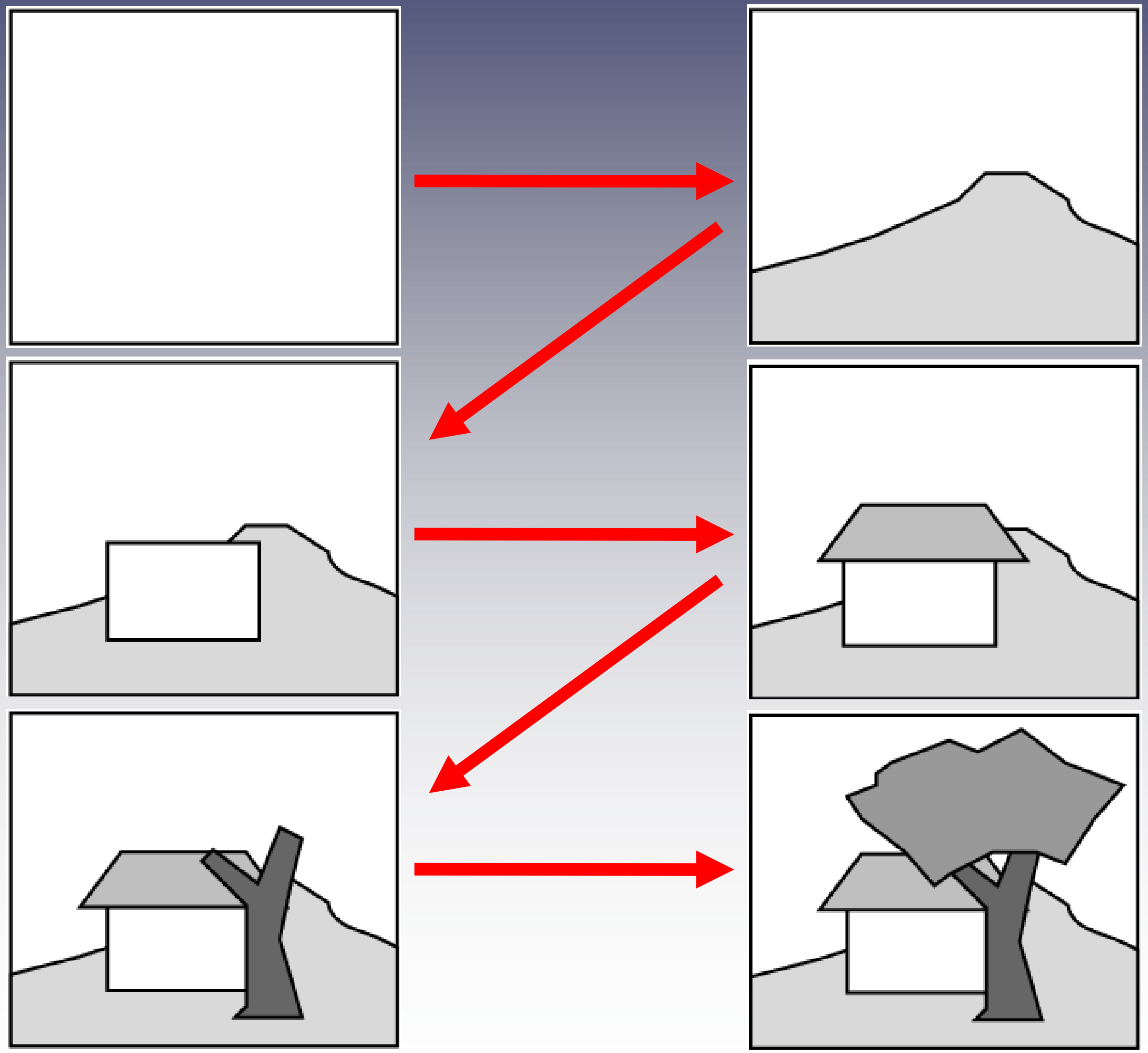
Backface Culling

- Basic idea: We don't have to draw polygons that face away from the viewer, since front-facing polygons will occlude them
- A back-facing polygon's **normal** forms an acute angle with the view vector



from Hill

Painter's algorithm

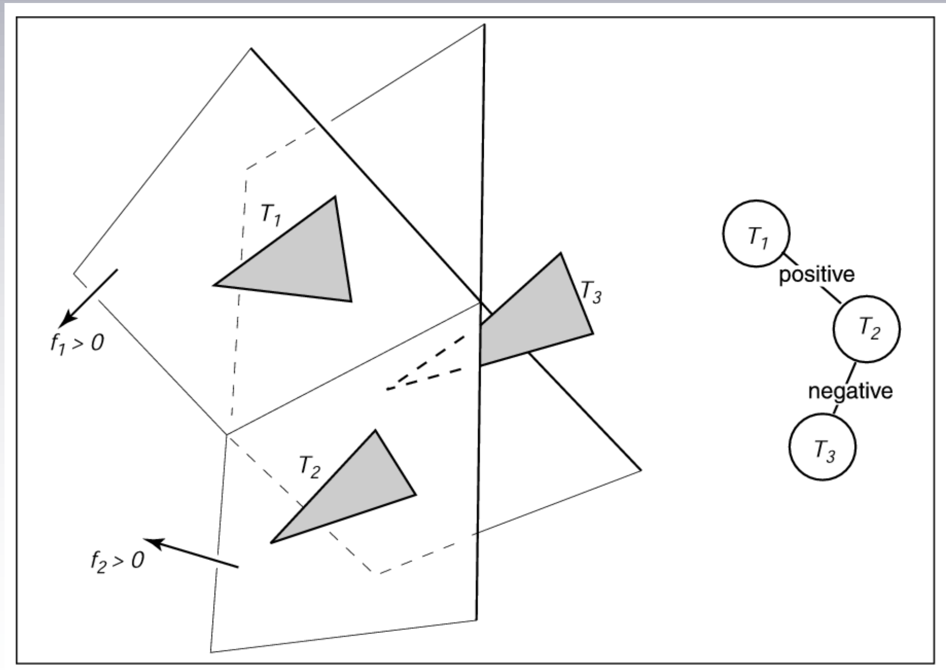


Draw primitives from back to front to avoid need for depth comparisons

from Shirley

Binary Space Partitioning (BSP) trees

- A preprocess to organize objects in a tree such that a traversal gives a depth sort relative to the eye
- Each internal node represents a "splitting plane"
- See applet at http://pauillac.inria.fr/~levy/bsp



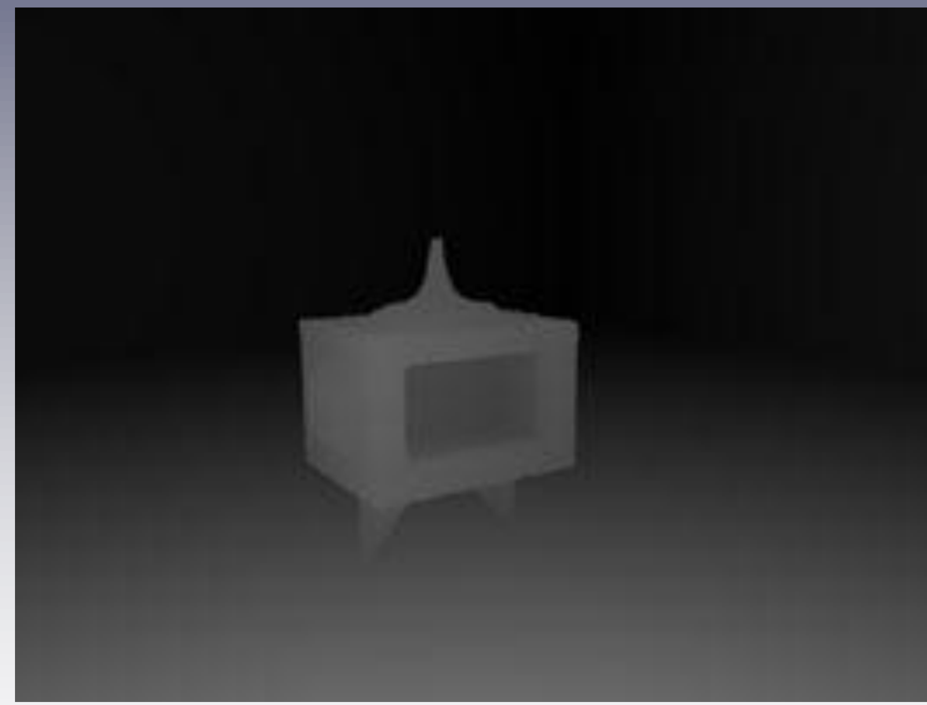
Z-Buffering

- Maintain an image-sized buffer of the depths of the closest pixels drawn so far
- Only draw a pixel if it's closer than what's been rendered already

```
for (each face F)
      for (each pixel (x,y) covering the face)
      {
          depth = depth of F at (x,y);
          if(depth < d[x][y])      //F is closest so far
           {
                c = color of F at (x, y);
                set the pixel color at (x, y) to c
                d[x][y] = depth; // update the depth buffer
           }
      }
```

from Hill

Z-buffer: Example



courtesy of DAM Entertainment

Color buffer

Depth buffer