

Course Information

Welcome to CS 355! This is a course on the basic mathematics, geometry, algorithms, and other concepts underlying the fields of computer graphics, image processing, computer vision, interaction. We will cover not only these underlying concepts but see how they are used in these various areas.

CS 355 is a prerequisite to the following follow-on courses:

- CS 450: Introduction to Digital Signal and Image Processing (soon to be renamed to Computer Vision)
- CS 455: Computer Graphics

This class taken alone can provide a broad introduction to these areas and a set of tools that are useful in any context where computers render or interact with images. Taking either of the follow-on courses allows you to go deeper into one of these areas. Taking both will lay a solid foundation for any career in digital media, animation, computer vision, robotics, etc.

You may also find this course useful for the following other electives:

- CS 456: Human-Computer Interaction
- CS 478: Machine Learning
- CS 401R: Statistical Machine Learning
- CS 501R: Deep Neural Networks

Class Meetings

9:30 - 10:45 T Th

3106 JKB

Instructor

Parris Egbert (<http://morse.cs.byu.edu>)

egbert@cs.byu.edu

3318 TMCB

(801) 422-4029

Office Hours:

9:00-10:00 a.m., MWF

or [by appointment](http://morse.cs.byu.edu/schedule) (<http://morse.cs.byu.edu/schedule>)

Teaching Assistants

Christian Arnold

chrisarnold27@gmail.com

1058 TMCB, Cubicle #14

Office hours:

Monday: 9 a.m-5 p.m

Wednesday 9-10:30 a.m 12-5 p.m

Lance James

thelancejames@gmail.com

1058 TMCB, Cubicle #14

Office hours:

Tuesday: 12:00 - 1:30, 3:00 - 9:00

Thursday: 11:00 - 1:30, 3:00 - 9:00

Friday: 10:00 - 2:00

Prerequisites

- CS 240 or equivalent (including prerequisites CS 235 and 236).
- Math 313 (Linear Algebra) or equivalent.

Class Policies

Prerequisites

You should have taken CS 240 and all of its prerequisites. We will write a number of non-trivial programs, and now isn't the time to learn programming.

This course also requires Linear Algebra (Math 313). We'll be using linear algebra extensively, but don't be afraid of the math. The concepts in this class are pretty simple, and I'll review what we need for this course.

Note: for this semester, Math 313 is listed as a prerequisite. We are changing this to be a co-requisite, so if you know anyone who wants to take the class but couldn't get in because they're currently enrolled in Math 313, please invite them to contact me for an add code.

Text and Readings

There is no required textbook for this class. All of the slides used in class will be distributed here through Canvas so that you can reference them. Other electronic readings may also be assigned.

Class Communications

I will generally try to minimize administrative overhead at the beginning of class, so we will rely heavily on these Canvas, discussion groups, and [e-mail \(<mailto:morse@cs.byu.edu>\)](mailto:morse@cs.byu.edu) to handle administrative issues. *Information distributed through any of these means carries the same importance and validity as in-class announcements.*

Announcements

I will communicate with you all using the Announcements facility in Canvas, and any announcements posted there carry the same weight as if they were made in class. For things that are time-critical, I'll make sure to also send a copy by e-mail. **Please make sure the e-mail address you have on file with the university is the one that you want to receive such announcements at.**

Discussions / Q&A

We will also use Canvas's discussions facility for class discussions and questions. If you have a question, look to see if it's been asked there, and if not, please feel free to post your own question. If you can contribute to helping answer someone else's question, please also do so.

Homework and Programming Assignments

There will be programming assignments due approximately weekly throughout the semester (except for midterms).

There will also be paper-and-pencil assignments associated with many of the programming labs, which will generally be due a few days before the respective lab. The purposes of these homework assignments are threefold: 1) to walk you through on paper-and-pencil the math you'll be coding for that lab, 2) to apply some of the material covered in class but not actually used for a specific lab, and 3) to give you practice with the types of questions you might see on exams.

Late Policy

All written homework must be submitted by 5:00 p.m. on the day it is due. Late homework will not be accepted. Answer keys will be published right after the deadline so that you may make use of these keys while working on the corresponding programming labs. You are on your honor not to distribute the homework answer keys.

All programming labs must be submitted by midnight at the end of the date due. Late labs will be penalized 10% (of the maximum score) per day up to a maximum penalty of 50%. Weekends and university holidays are excepted, so any work turned in by the following day of university classes will incur only one day's penalty. No extra credit, if any, will be given for work that is turned in late.

All late work must be turned in by the last day of university classes at the end of the semester.



Introduction

CS 355: Introduction to Graphics and Image Processing

Pixels Everywhere!

Drawing

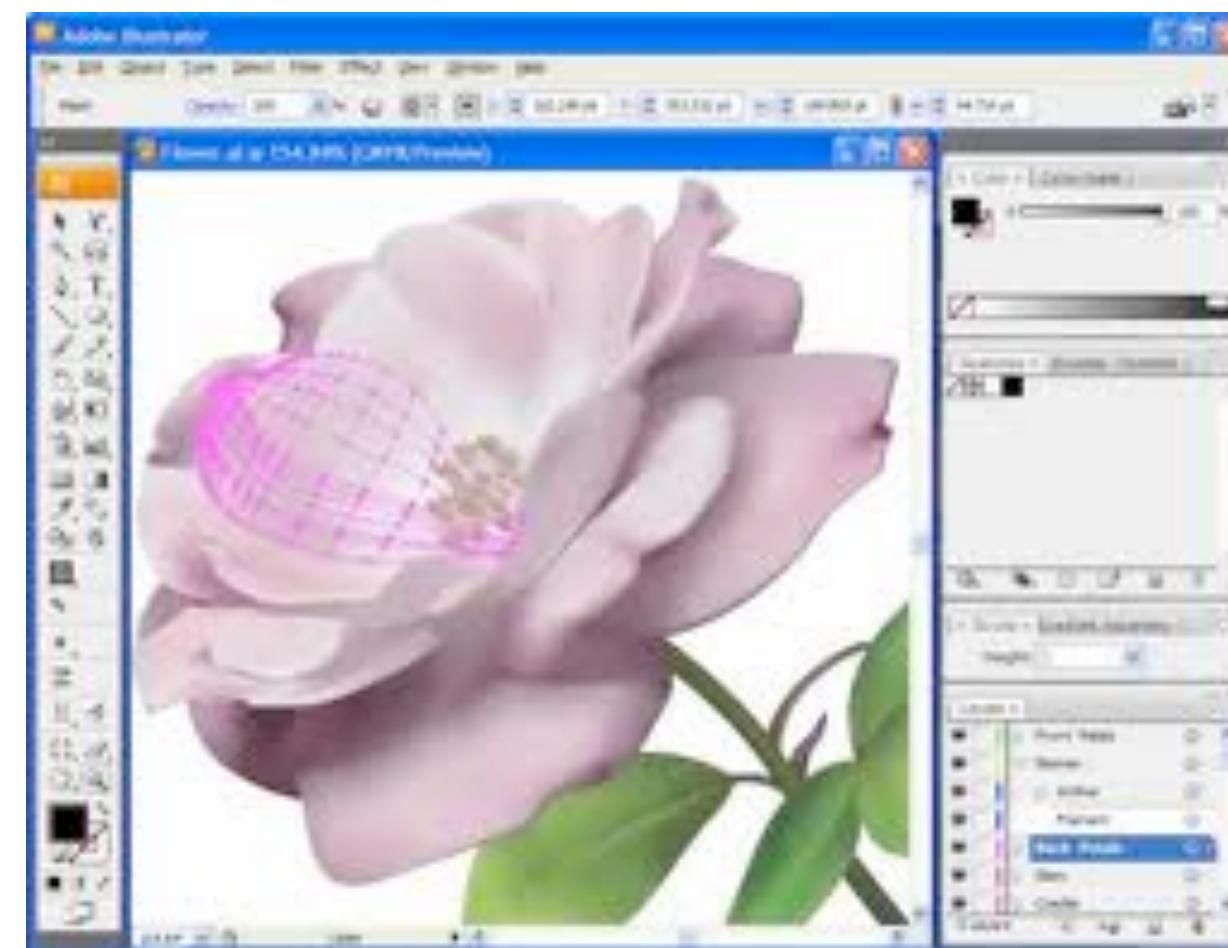
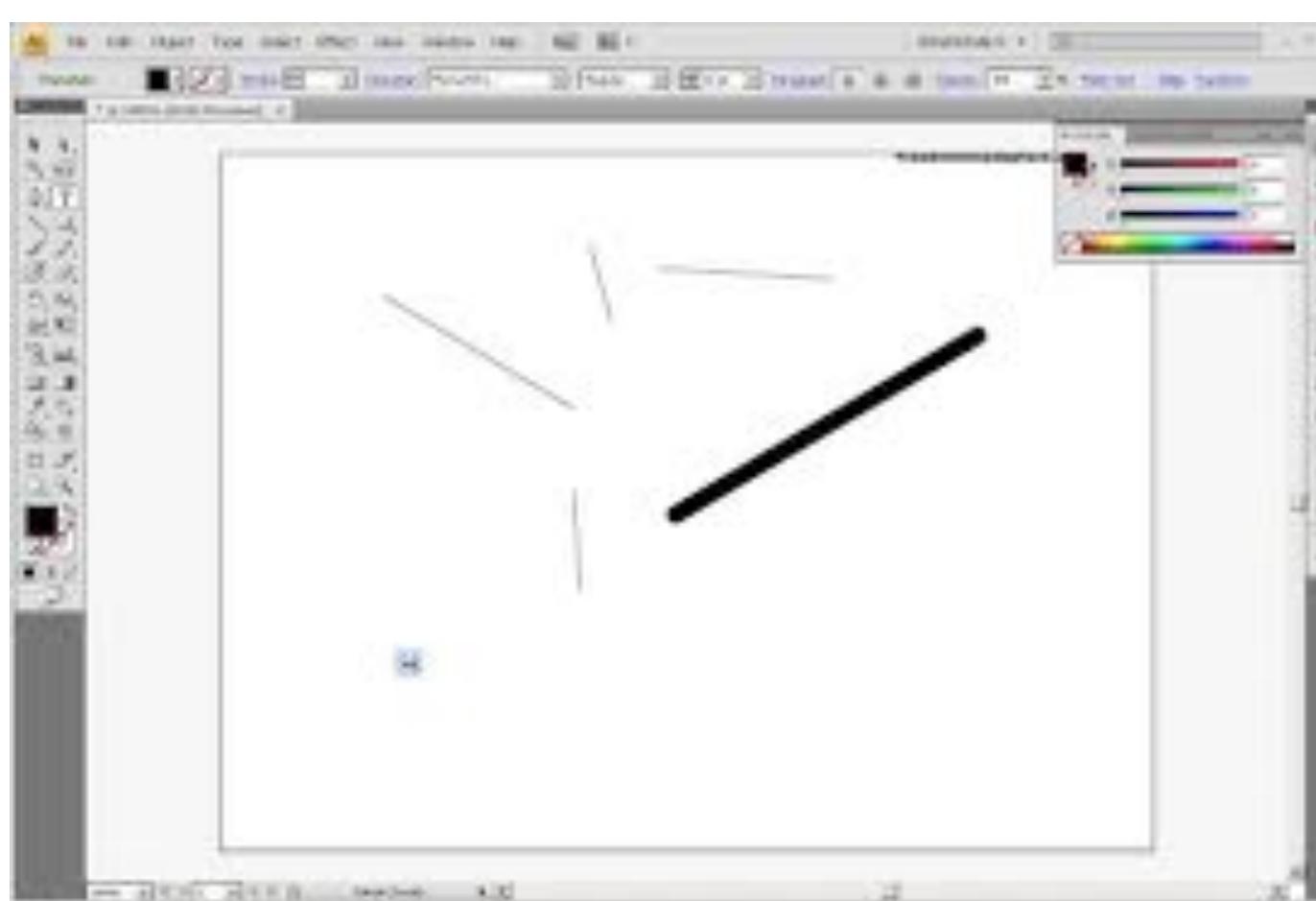
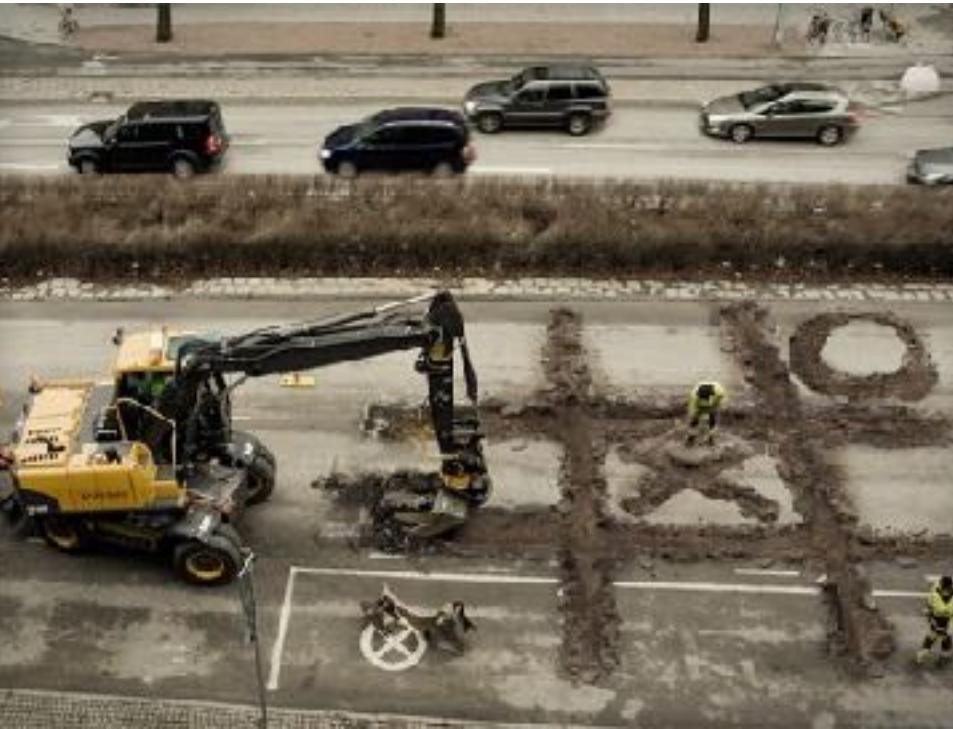
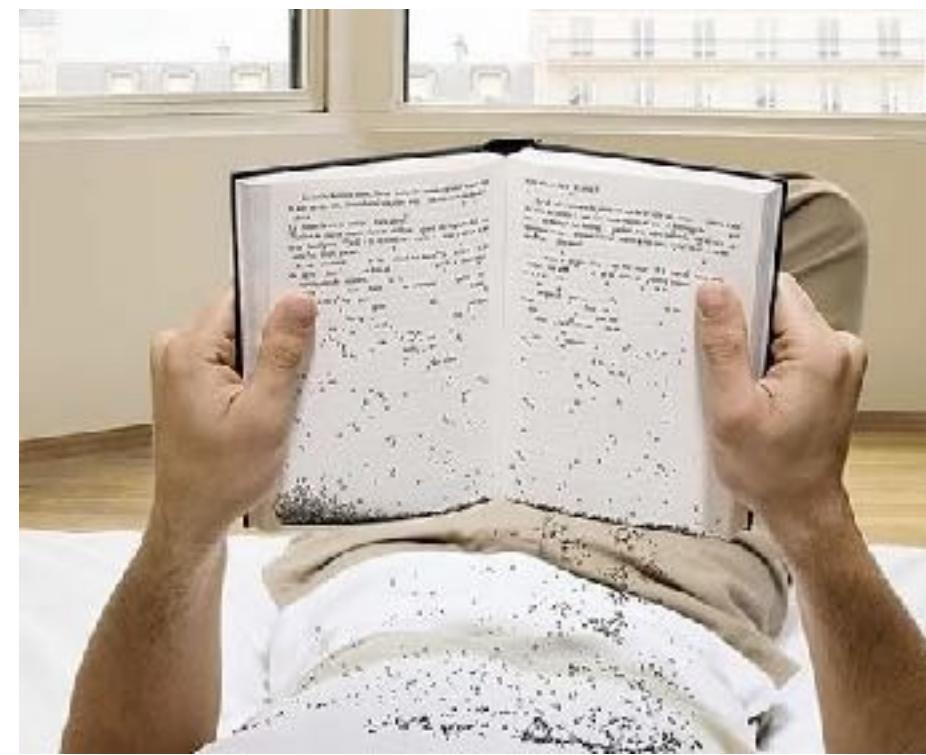
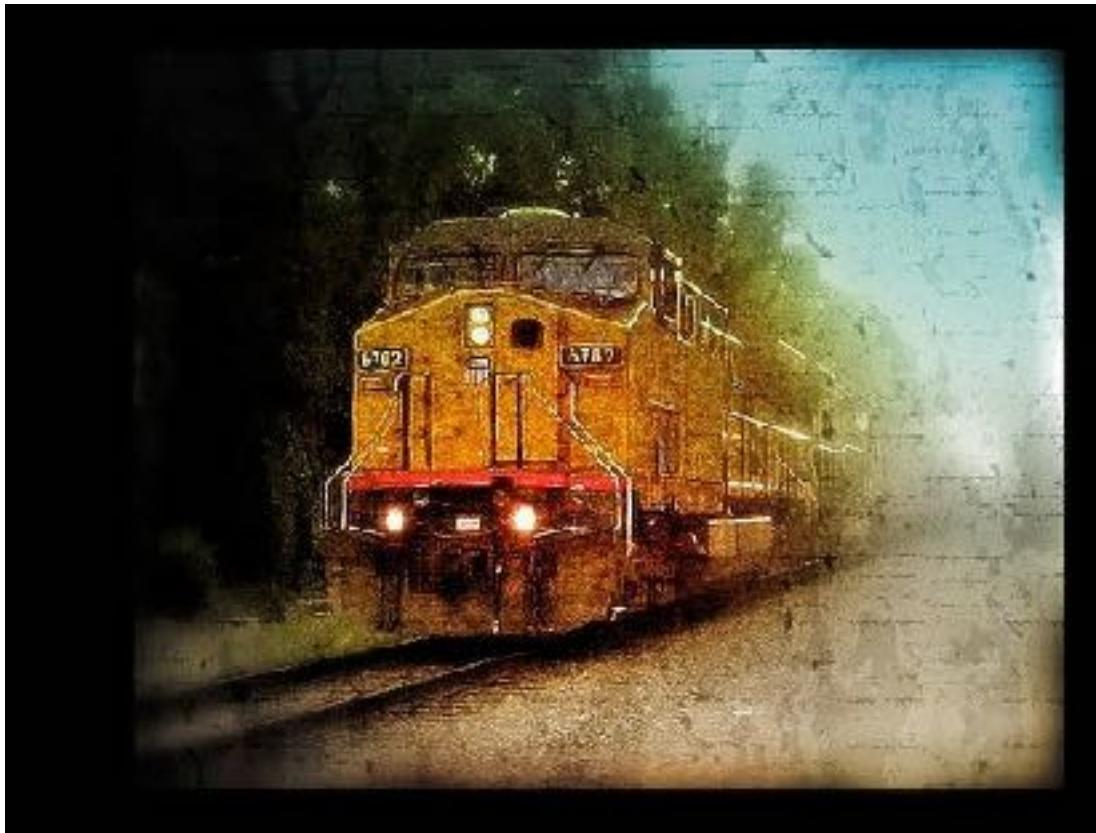


Image Editing



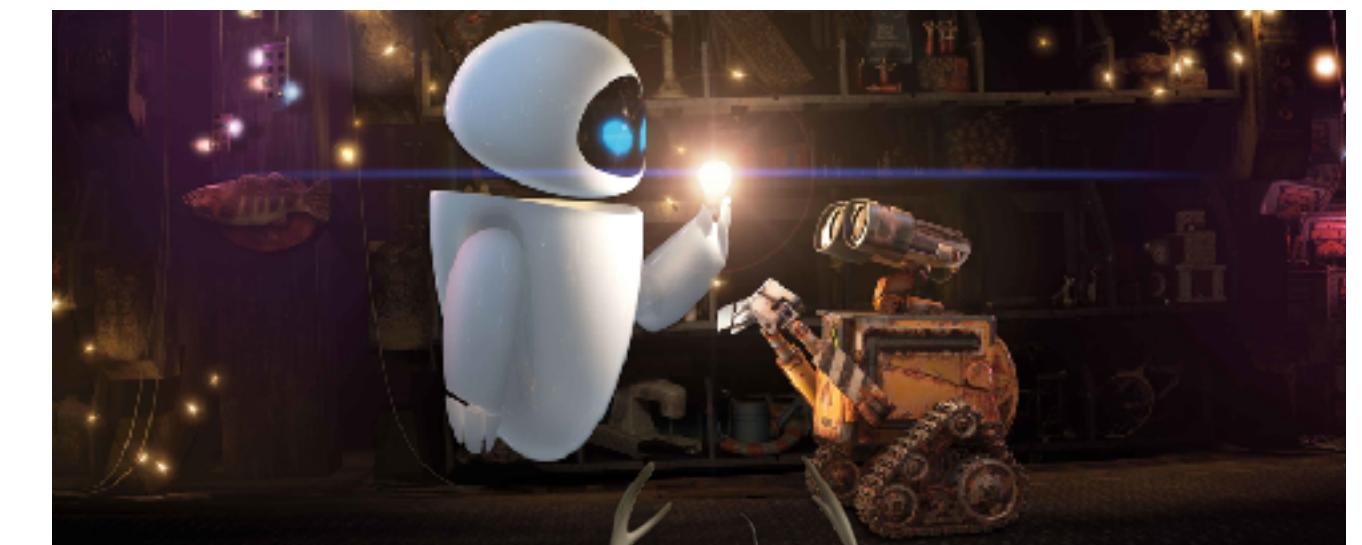
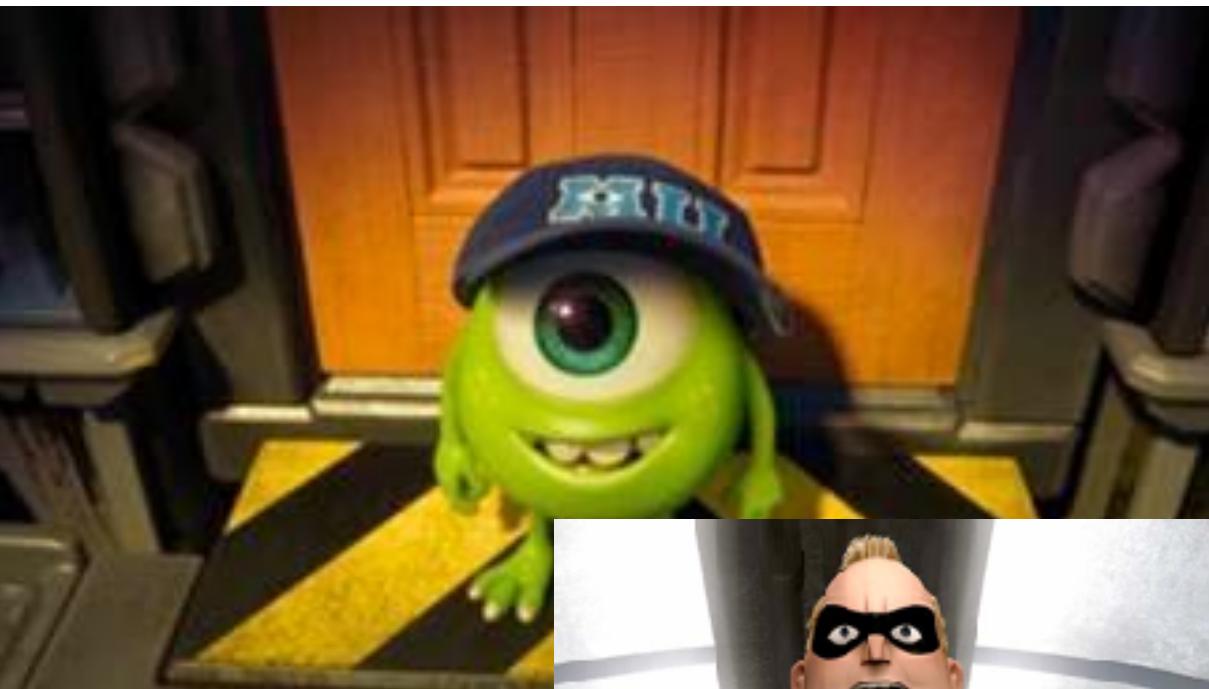
Graphical Interaction



Games



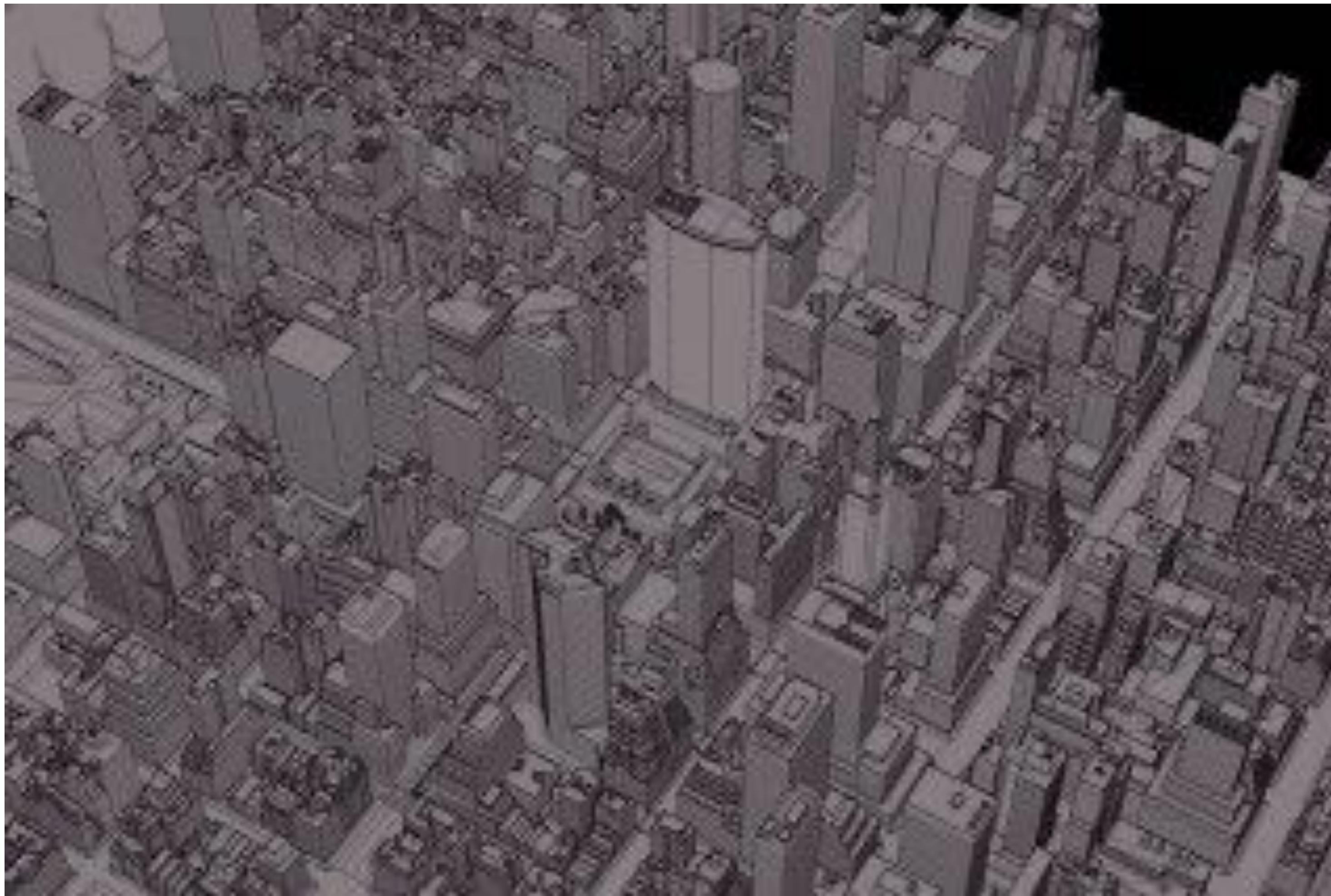
Animated Films



Special Effects



Special Effects



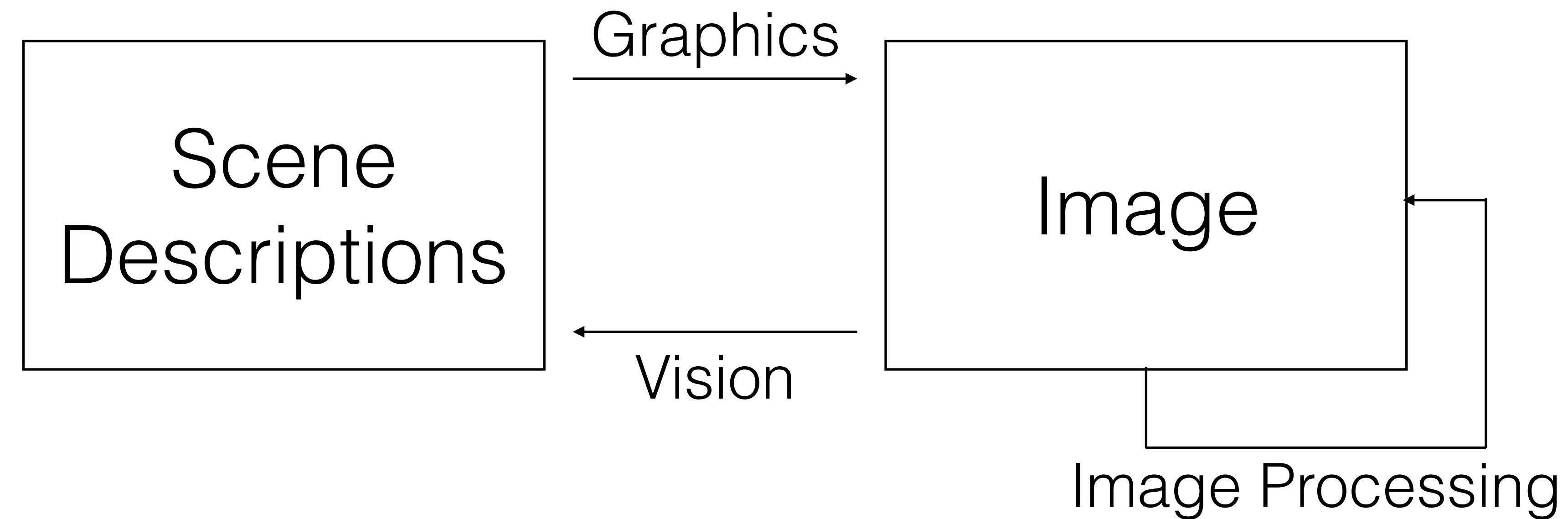
Math

Geometry

~~Pixels Everywhere!~~

Algorithms

Graphics, Image Processing, and Vision



Image/Vision

CS 456

CS 470

CS 478

CS 401R

CS 501R

CS 750

CS 650

CS 450

Graphics

CS 655

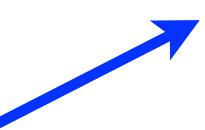
CS 455

CS ANIM
450

CS ANIM
459

CS 495R

CS 497R



You Are Here

What You're Going to Build

- Working with Images
 - Brightness, contrast adjustment (2)
 - Color image processing (3)
 - Blending and compositing (3)
 - Neighborhood operations: (3)
 - noise removal
 - sharpening, edge detection
- 2D Transformations
 - Simple rotation, translation, scale (4)
 - Homographies (9)
- 3D Graphics
 - Wireframe 3D rendering w/ OpenGL (5)
 - Hierarchies of transformations (6)
 - Wireframe 3D rendering (7)
 - Visibility, lighting, shading (8)
 - Geometric tests (9)
 - Image warping and texture mapping (9)
 - Frequency-domain processing (10)

Syllabus

- Instructor / TAs
- Prerequisites
- Policies
- Schedule
- Programming Labs
- Written Homework
- Exams

Next up...

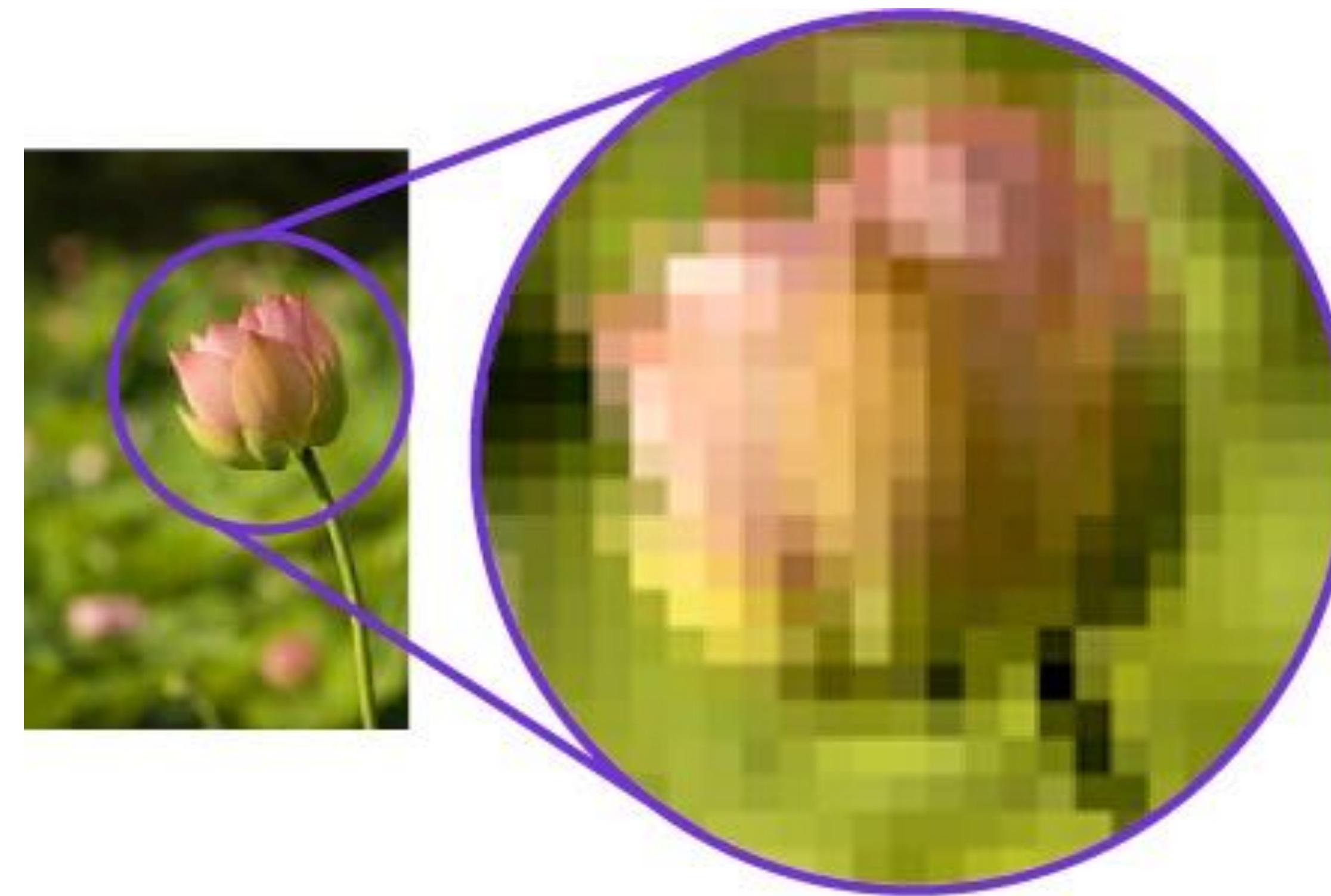
- Q&A on the syllabus, policies, etc. — read them before next time
- Introduction to Python and Jupyter
- Introduction to Lab #1



Raster Graphics and Displays

CS 355: Introduction to Graphics and Image Processing

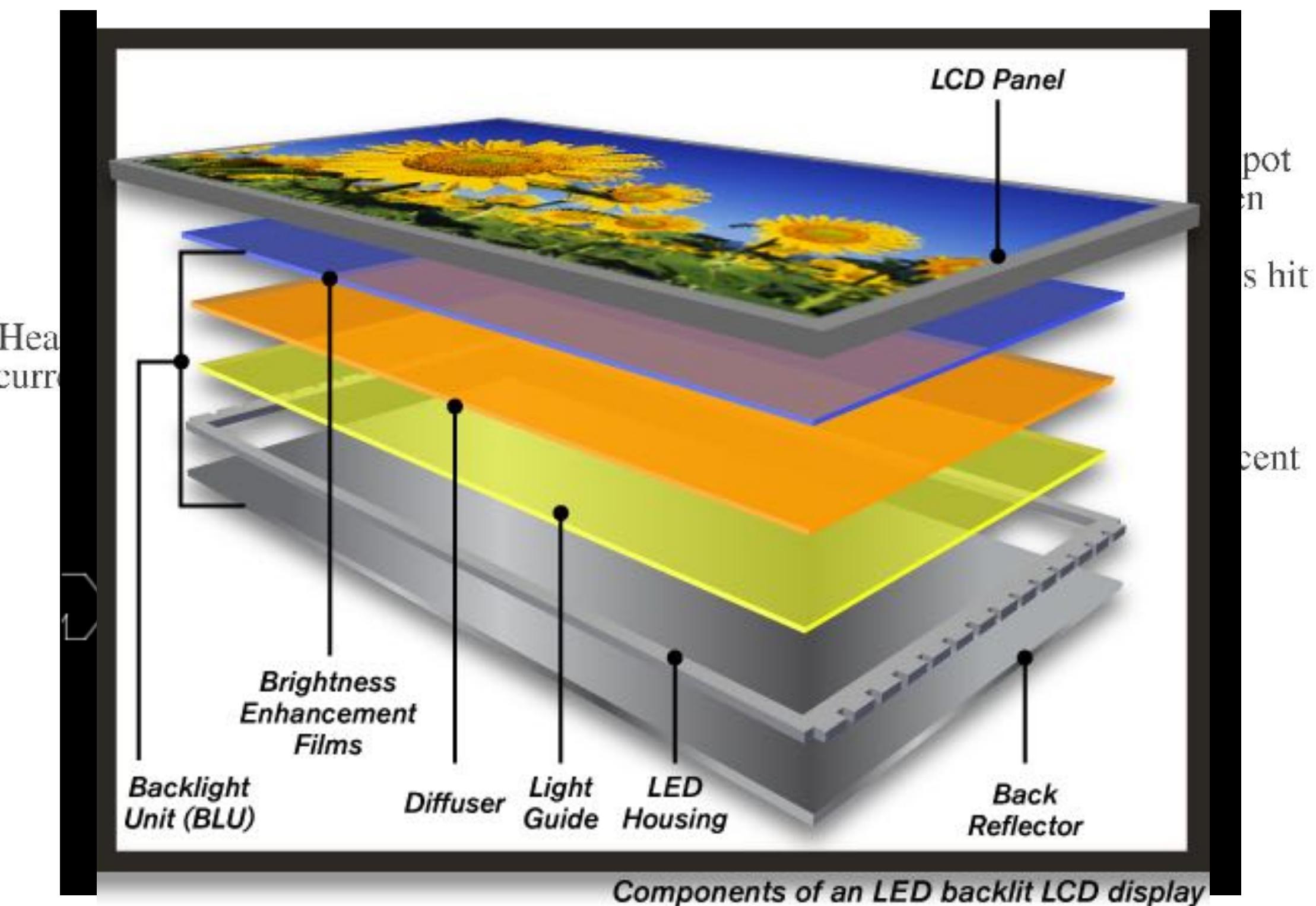
Raster Images



Most digital displays are made up of discrete dots called “pixels” (short for picture elements)

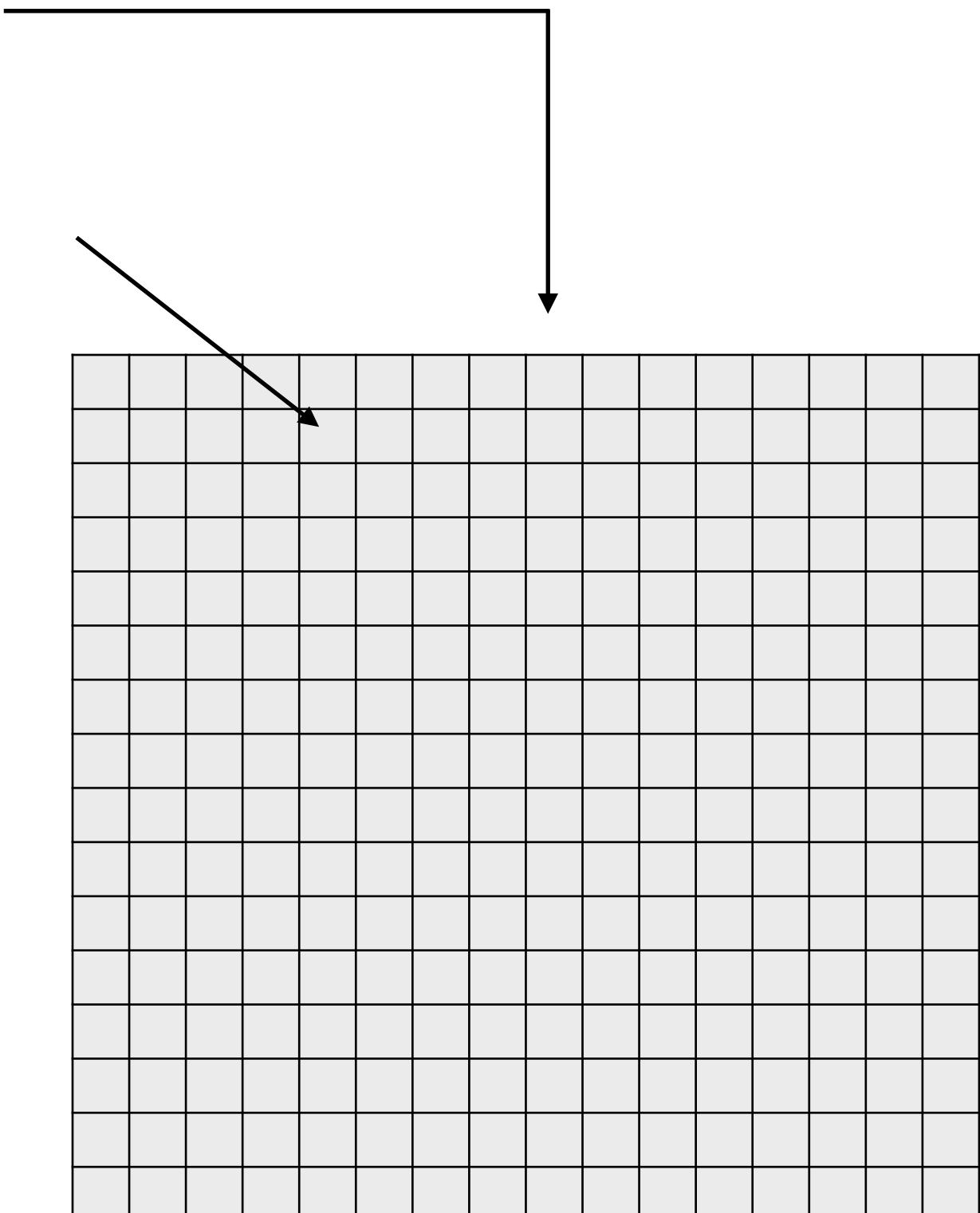
Quick History

- Vector graphics
- Raster (CRT)
- Raster (Digital)

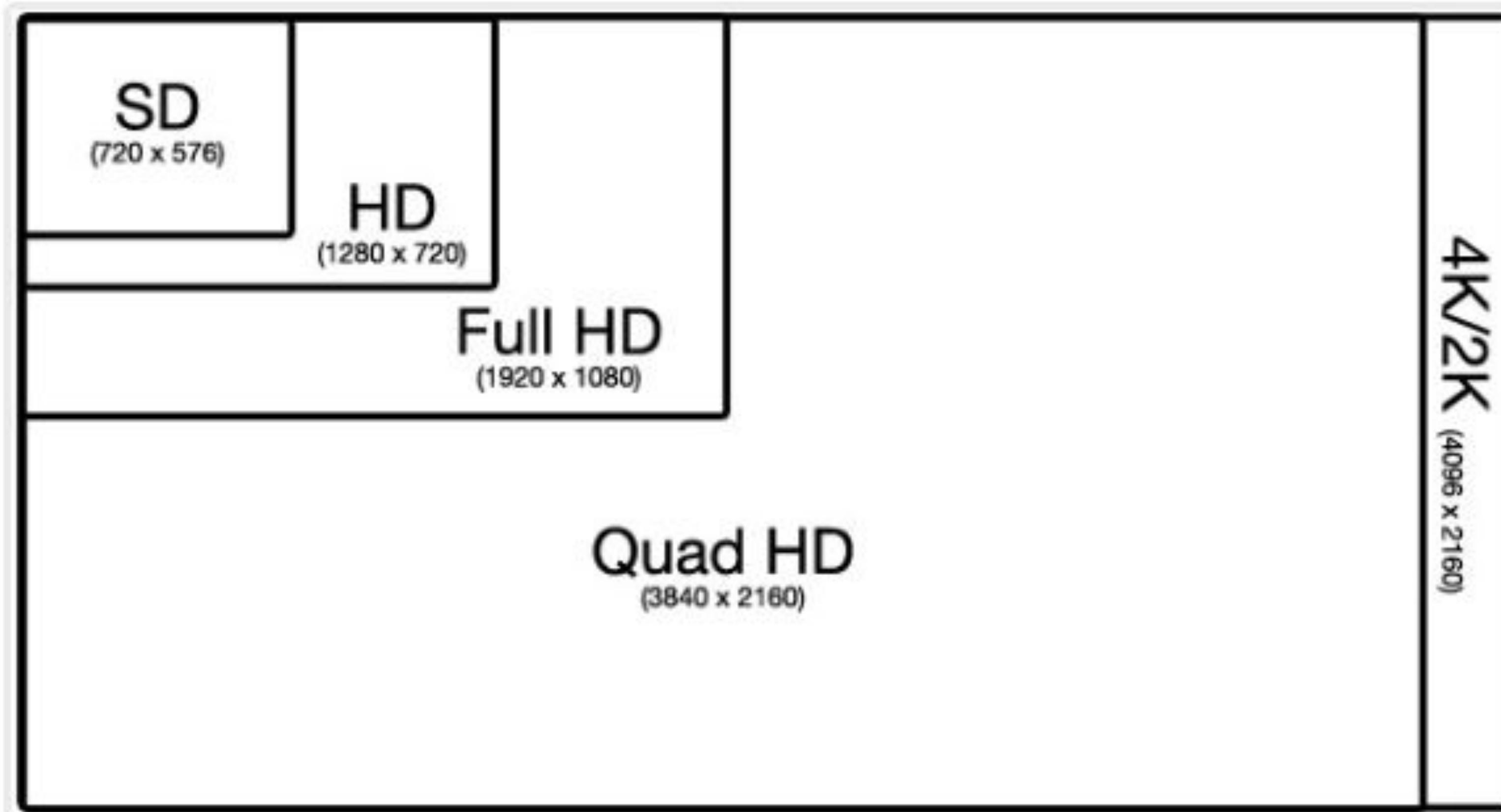


Raster Images

- Size: number of pixels (height x width)
- Bit depth: bits of precision for each pixel
 - Binary (true black and white)
 - 2-bit gray (4 shades)
 - 8-bit gray (256 shades)
 - 12-bit gray (X-rays / CT)
 - 24-bit color (8 bits each of R, G, B)

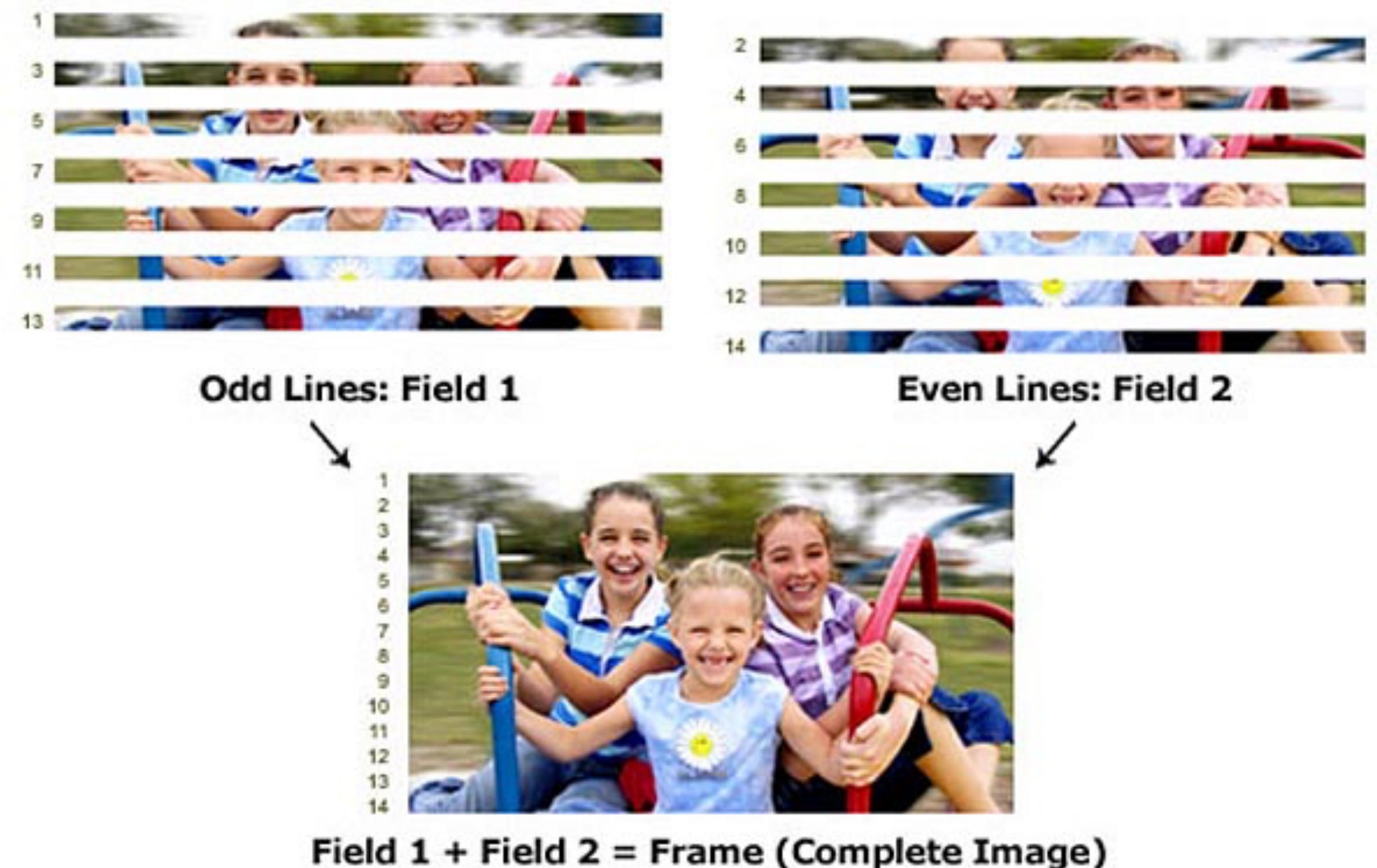


Common Display Sizes



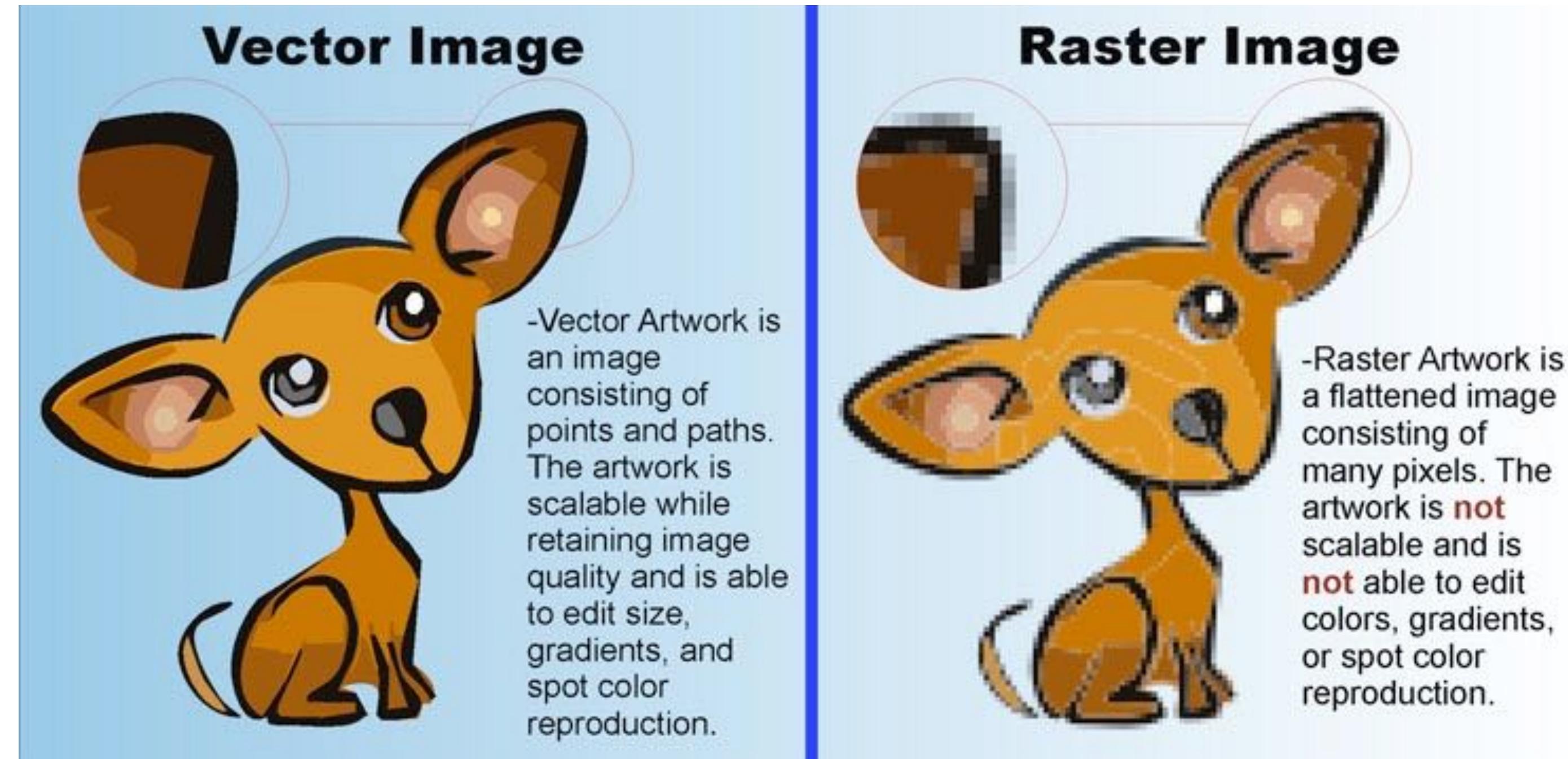
Interlacing

- Progressive
 - Full screen 60 frames / second
- Interlaced
 - Half screen *fields* 60 times / second
 - Odd lines, then even lines...
 - Equivalent of 30 frames/second



How you display pictures isn't
how you have to store them!

Raster vs. Vector Graphics



Vector:
Continuous Curves

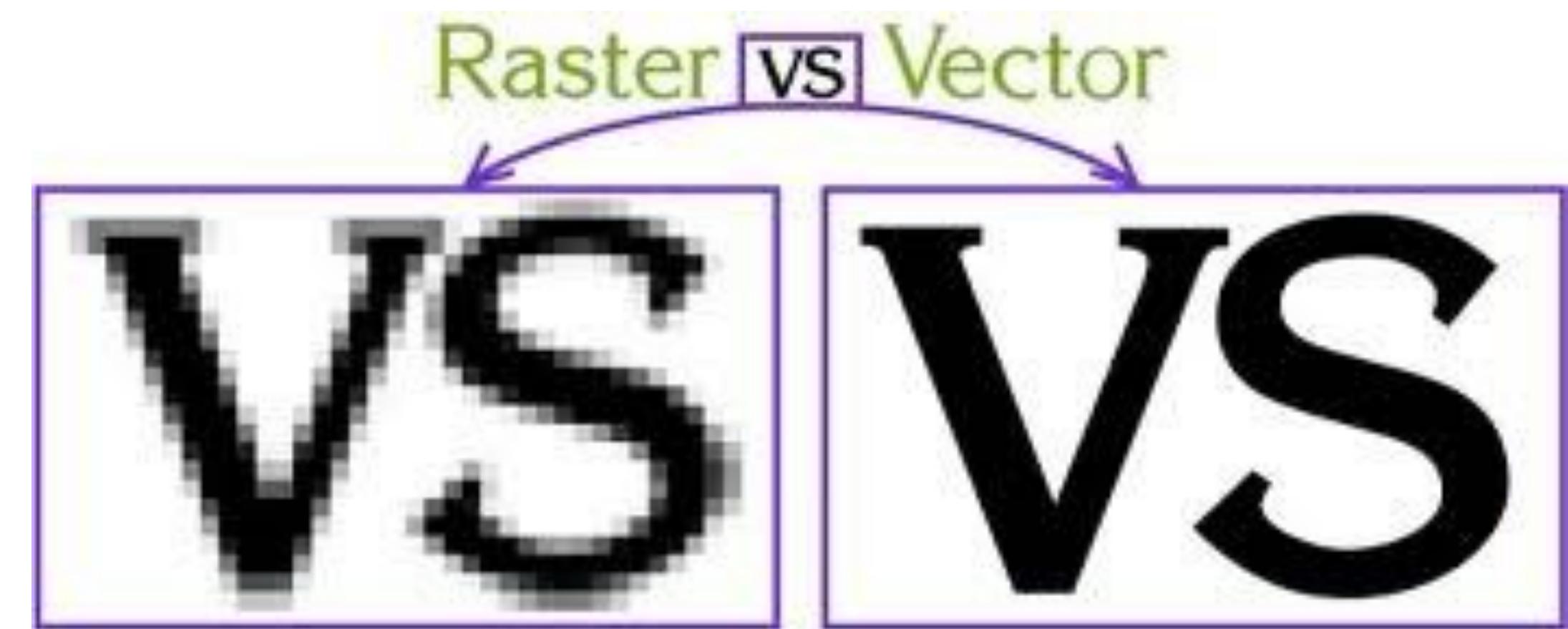
Raster:
Lots of dots

Rasterization

- Vector graphics are higher quality
- But everybody uses raster displays
- Have to convert to a raster image first
(But get to target resolution to device!)
- This process is called rasterization
(sometimes scan conversion)

Example: Text

- Most fonts aren't stored as bitmaps (images)
- Stored instead as curves representing the outline of the characters
- Converted to whatever resolution device supports



Example: Postscript

- When you send a Postscript file to a printer,
 - You aren't sending a picture of what to print (usually)
 - You are sending a program for how to print it



```
%!PS-Adobe-3.0
%%Title: QR Barcode 2-H, mask=6
%%Creator: JpGraph Barcode http://www.aditus.nu/jpgraph/
%%CreationDate: Mon 6 Jul 16:08:12 2009
%%DocumentPaperSizes: A4
%%EndComments
%%BeginProlog
%%EndProlog
%%Page: 1 1

%Module width: 3 pt

%Datum: ABCDEFGH01234567
%Each line represents one row and the x-position for black modules: [xpos]

3.05 setlinewidth
[[12][15][18][21][24][27][30][45][51][66][69][72][75][78][81][84]] {{}} forall 87 moveto 0 -3.05 rlineto stroke} forall
[[12][30][45][66][84]] {{}} forall 84 moveto 0 -3.05 rlineto stroke} forall
[[12][18][21][24][30][36][39][42][45][51][66][72][75][78][84]] {{}} forall 81 moveto 0 -3.05 rlineto stroke} forall
[[12][18][21][24][30][36][39][45][48][51][54][66][72][75][78][84]] {{}} forall 78 moveto 0 -3.05 rlineto stroke} forall
[[12][18][21][24][30][45][66][72][75][78][84]] {{}} forall 75 moveto 0 -3.05 rlineto stroke} forall
[[12][30][39][45][48][51][54][60][66][84]] {{}} forall 72 moveto 0 -3.05 rlineto stroke} forall
[[12][15][18][21][24][27][30][36][42][48][54][60][66][69][72][75][78][81][84]] {{}} forall 69 moveto 0 -3.05 rlineto stroke} forall
[[39][42][51][60]] {{}} forall 66 moveto 0 -3.05 rlineto stroke} forall
[[21][24][30][33][39][48][51][54][60][75][78]] {{}} forall 63 moveto 0 -3.05 rlineto stroke} forall
[[15][18][27][39][42][45][48][51][54][57][69][78][81][84]] {{}} forall 60 moveto 0 -3.05 rlineto stroke} forall
[[12][21][27][30][33][36][42][51][54][57][60][66][75][78][84]] {{}} forall 57 moveto 0 -3.05 rlineto stroke} forall
[[18][21][27][39][45][51][57][60][63][66][81][84]] {{}} forall 54 moveto 0 -3.05 rlineto stroke} forall
[[12][15][18][24][27][30][33][39][51][57][63][69][75][84]] {{}} forall 51 moveto 0 -3.05 rlineto stroke} forall
[[12][15][21][48][66][69][81][84]] {{}} forall 48 moveto 0 -3.05 rlineto stroke} forall
[[12][15][18][21][24][30][36][51][54][57][72][78][81]] {{}} forall 45 moveto 0 -3.05 rlineto stroke} forall
[[12][18][24][27][39][45][51][60][75][78][84]] {{}} forall 42 moveto 0 -3.05 rlineto stroke} forall
[[12][24][27][30][33][42][48][57][60][63][66][69][72]] {{}} forall 39 moveto 0 -3.05 rlineto stroke} forall
[[36][42][45][60][72][75][78][84]] {{}} forall 36 moveto 0 -3.05 rlineto stroke} forall
[[12][15][18][21][24][27][30][36][39][48][51][54][57][60][66][72][81][84]] {{}} forall 33 moveto 0 -3.05 rlineto stroke} forall
[[12][30][39][42][45][51][54][57][60][72][84]] {{}} forall 30 moveto 0 -3.05 rlineto stroke} forall
[[12][18][21][24][30][36][42][45][51][54][60][63][66][69][72][75][78]] {{}} forall 27 moveto 0 -3.05 rlineto stroke} forall
[[12][18][21][24][30][36][39][42][54][72][84]] {{}} forall 24 moveto 0 -3.05 rlineto stroke} forall
[[12][18][21][24][30][39][45][51][60][63][66][72][75][81][84]] {{}} forall 21 moveto 0 -3.05 rlineto stroke} forall
[[12][30][39][42][45][48][51][60][63][66][69][78][81]] {{}} forall 18 moveto 0 -3.05 rlineto stroke} forall
[[12][15][18][21][24][27][30][42][48][54][57][60][63][66][69][72][75][78][81][84]] {{}} forall 15 moveto 0 -3.05 rlineto stroke} forall
```

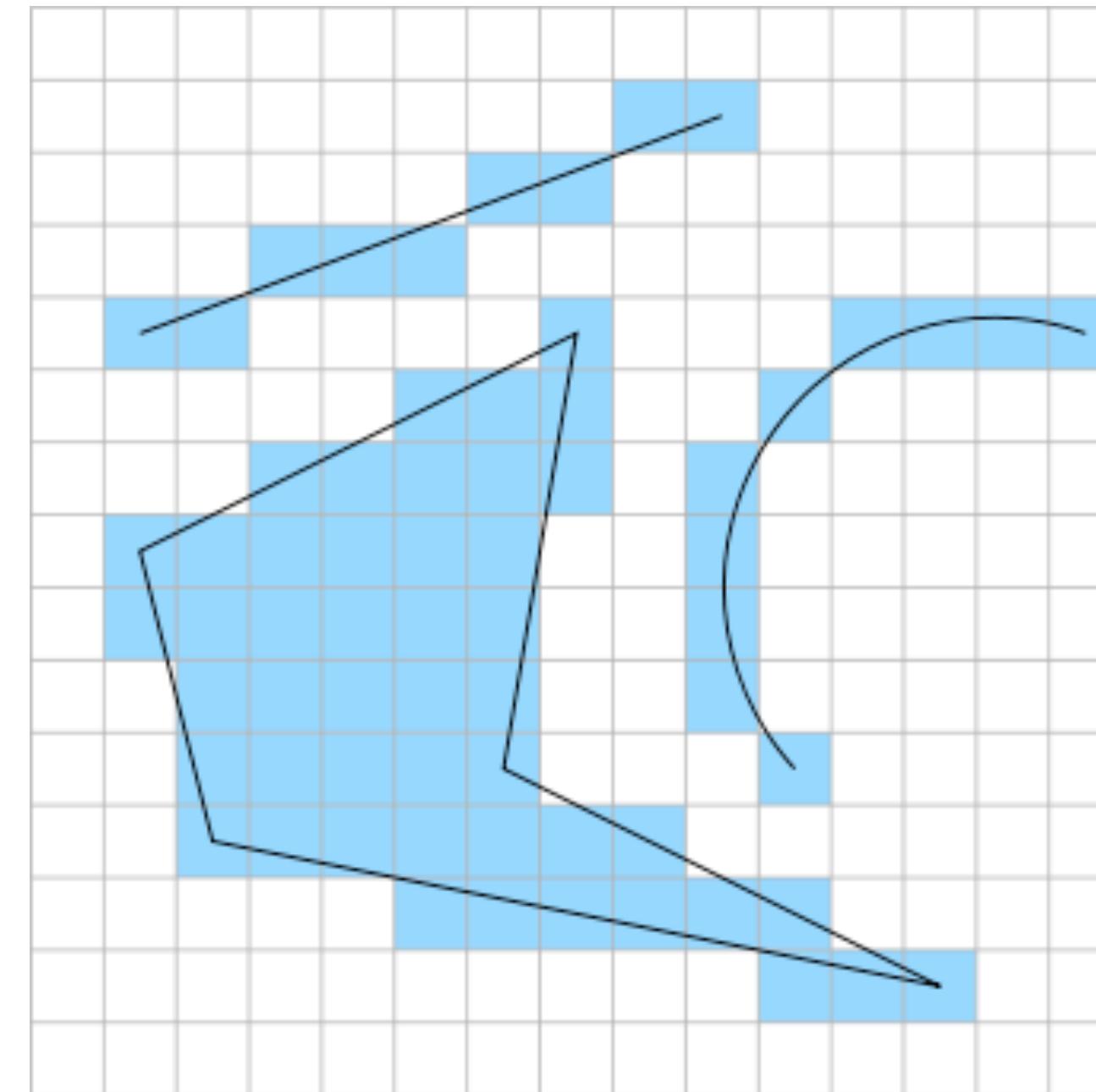
%End of QR Barcode

showpage

%%Trailer

What Could Possibly Go Wrong?

Which are in?

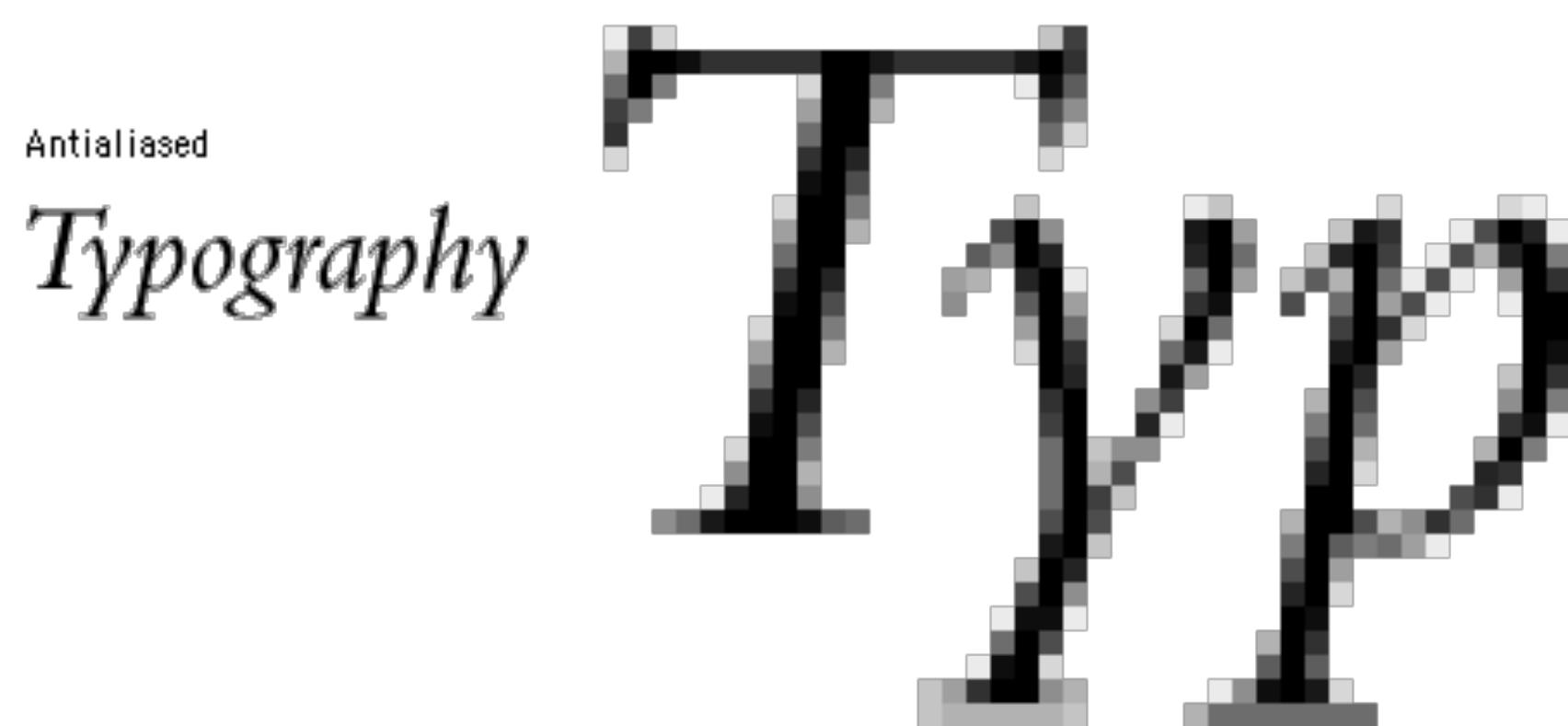
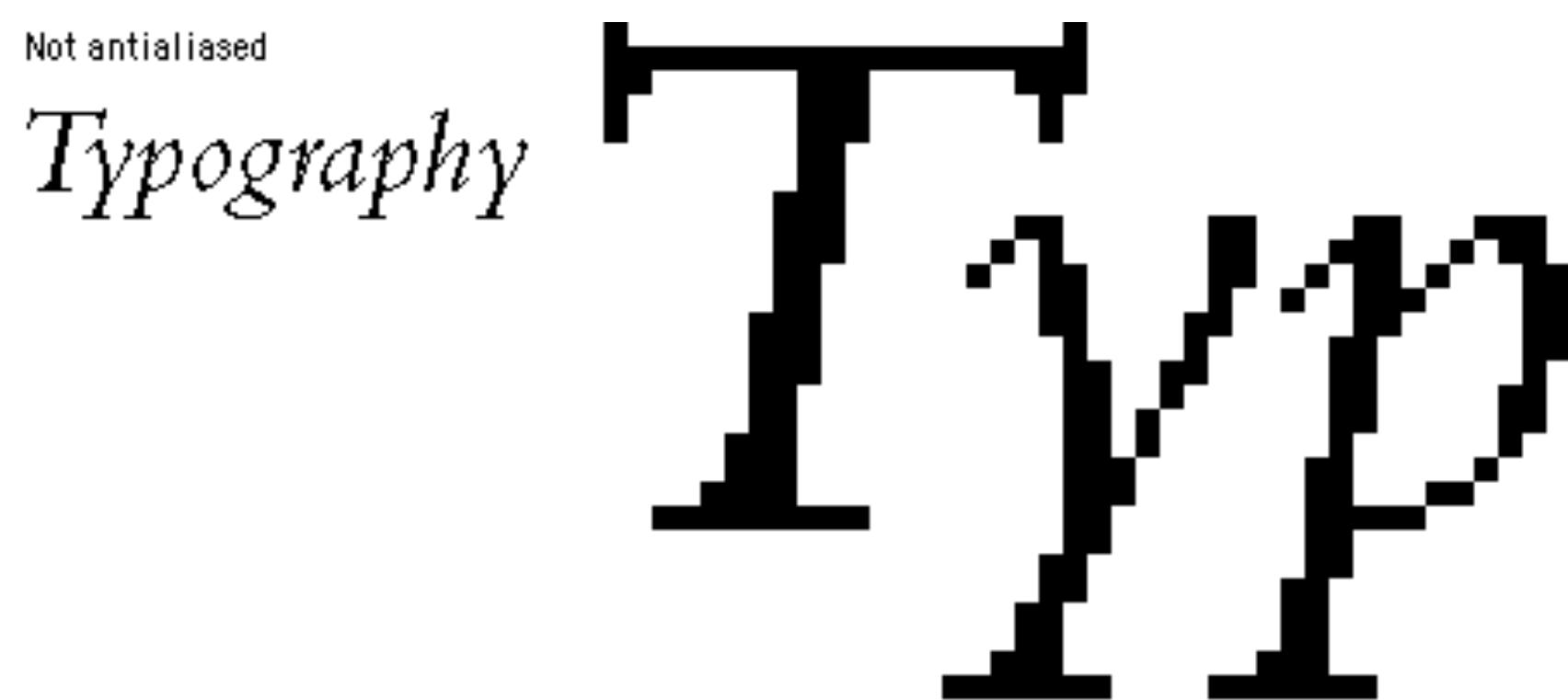
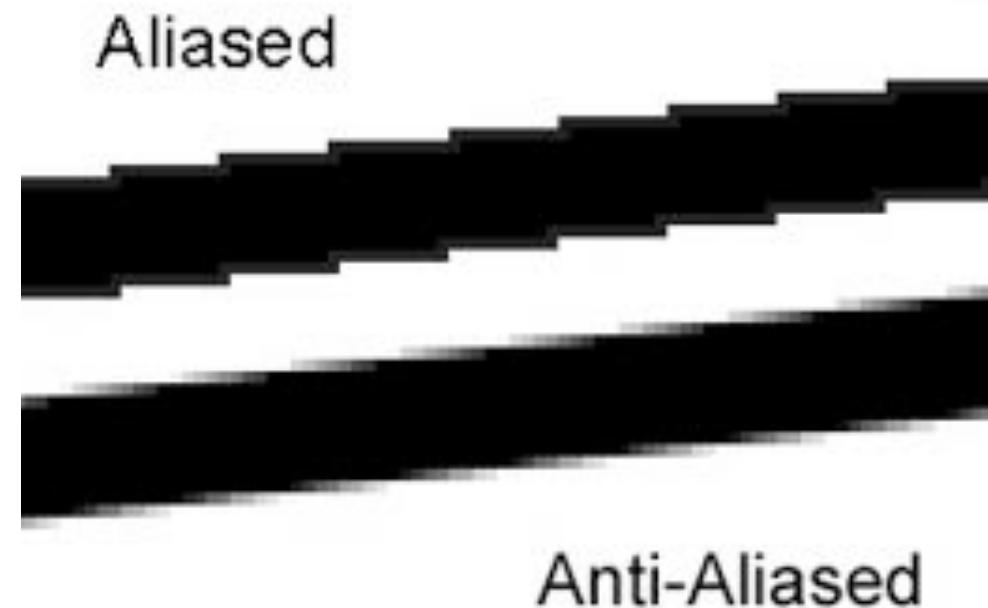


Which are out?

Aliasing
“Jaggies”

Antialiasing

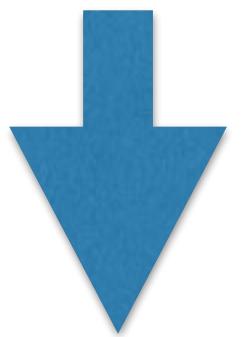
- Can make it look much sharper by blurring a bit (no, really!)
- Key: partially fill in pixels during rasterization



Screen Buffers

- To drive a raster display the computer must store a raster image of what goes on it (usually in graphics card)
- Called a ***screen buffer or video memory***
- Display hardware regularly sends the contents of the screen buffer to the screen
- You draw by writing into the buffer

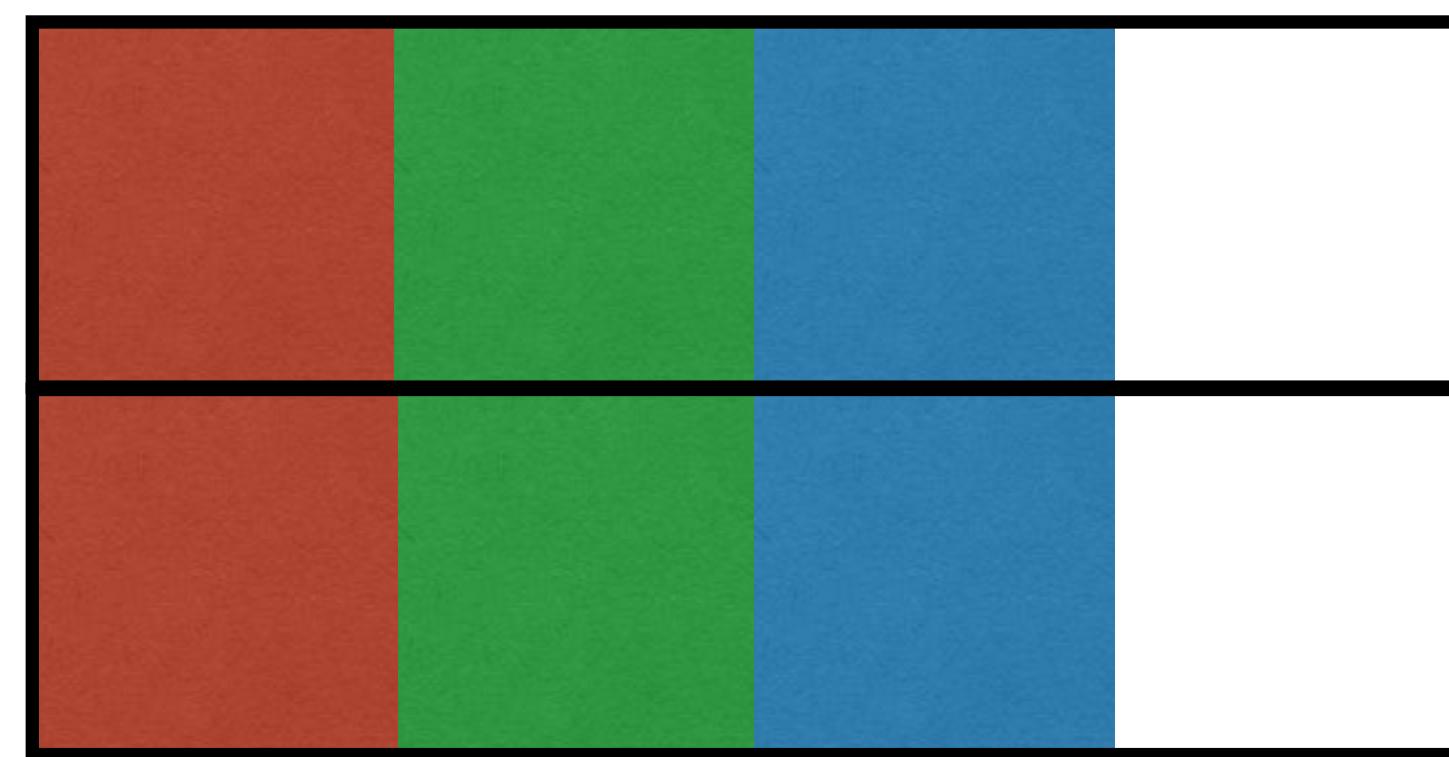
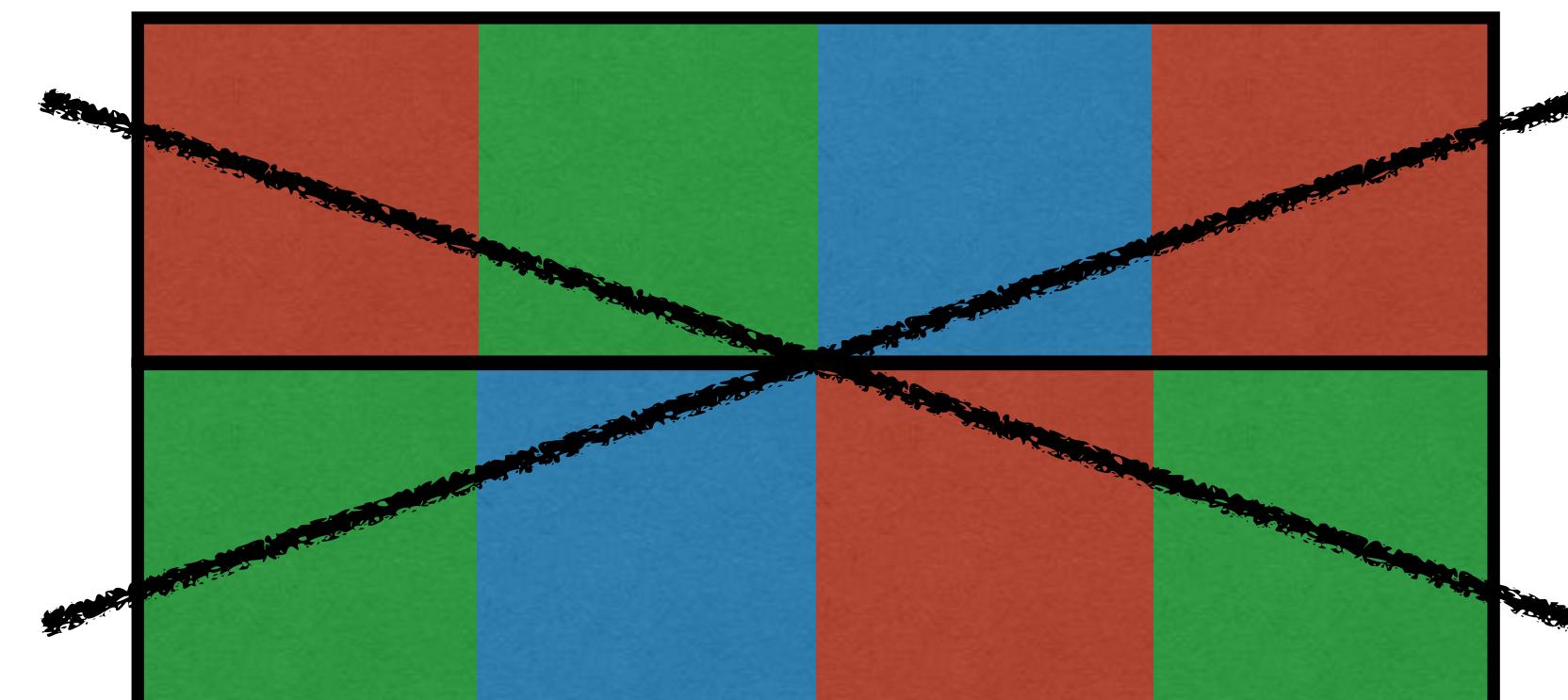
Screen Buffer



Display

Note about Color Buffers

- Color images are usually 24 bits
(one byte each red, green, blue)
- Memory words are usually 32 bits
(or some multiple of this)
- For speed, word-align your pixels
(works for other data as well!)
- But, but...



Alpha Channels

- But what about the wasted byte per pixel?
- Usually used for the alpha channel
 - Controls transparency
 - Useful for compositing
 - 0 = transparent, 255 = opaque
- More on this later...

Handling Buffers

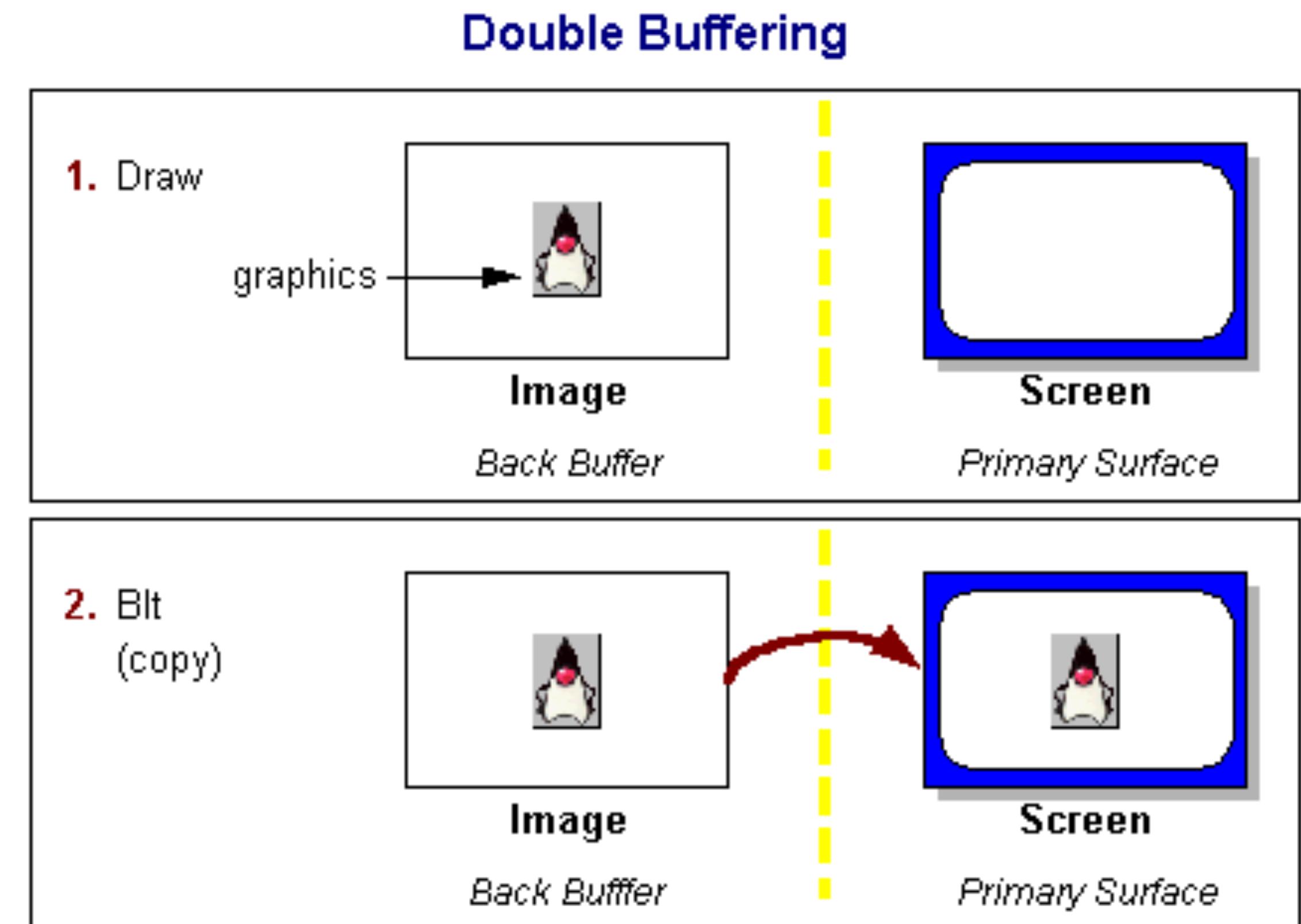
- What if it refreshes from the buffer while you're making changes?
- Two strategies:
 - Single buffer:
erase / redraw only what's changed
 - Double buffer

Single Buffer

- Erase / redraw only what's changed
 - Use clipping
 - Determining what's in that area is usually not worth it
- Might lead to flicker in these areas, but only these areas
- Used mainly in things that don't update a lot

Double Buffering

- Don't really draw to the real screen buffer
- Draw to ***offscreen buffer***
- Copy buffers (fast)
 - Some systems support switching with just pointers
- Most common for games, animation, etc.



Layers upon layers...

- In most systems there are lots of layers:
 - Drawing API
 - GUI
 - OS
 - Graphics drivers
 - Graphics cards
 - Display

Coming up...

- Level operations on images
 - Brightness
 - Contrast
 - ...



Level (Point) Operations

CS 355: Introduction to Graphics and Image Processing

Level Operations

- Simplest enhancement:
process each point independent of others
- Output value is a function of the input value *only*
- “Point operations” or “level operations”



Level Operations

- Simple idea with lots of applications:
 - Brightness
 - Contrast
 - Scaling
 - Clipping
 - Negatives
 - Thresholding
 - Quantization
 - Logarithmic encoding
 - Gamma correction
 - Windowing
 - Equalization
 - and many more...



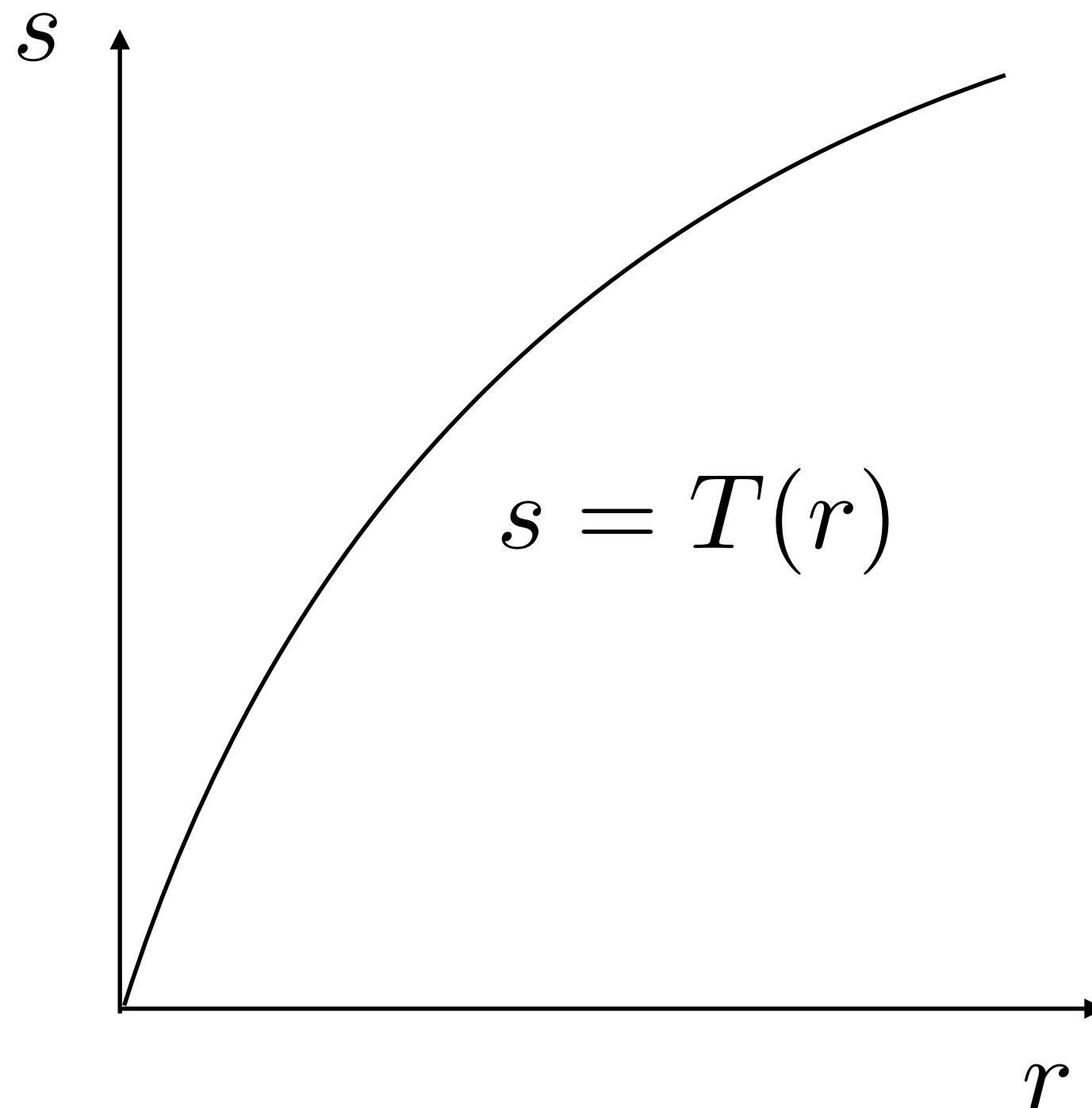
Level Operations

- Output value is a function of the input value

r = input value

s = output value

T = a greylevel transformation



```
for all pixel positions x, y:  
out[x,y] = func(in[x,y])
```

Brightness

$$s = r + c$$

$c > 0$ brighter

$c < 0$ darker



Contrast

$$s = a \cdot r$$

$a > 1$ more contrast

$a < 1$ less contrast



Linear Operations

$$s = a \ r + c$$

a gain

c bias / offset



Clipping

Clipping to a limited range:

$$s = \begin{cases} s_{\min} & \text{if } r < s_{\min} \\ s_{\max} & \text{if } r > s_{\max} \\ r & \text{otherwise} \end{cases}$$

Very common to clip to [min,max] of the range
to avoid unsigned wrap-around

Scaling

Scaling linearly from one range to another:

$$s = (r - r_{\min}) \frac{s_{\max} - s_{\min}}{r_{\max} - r_{\min}} + s_{\min}$$

Negative

$$s = r_{\max} - r$$

or

$$s = r_{\max} - r + r_{\min}$$

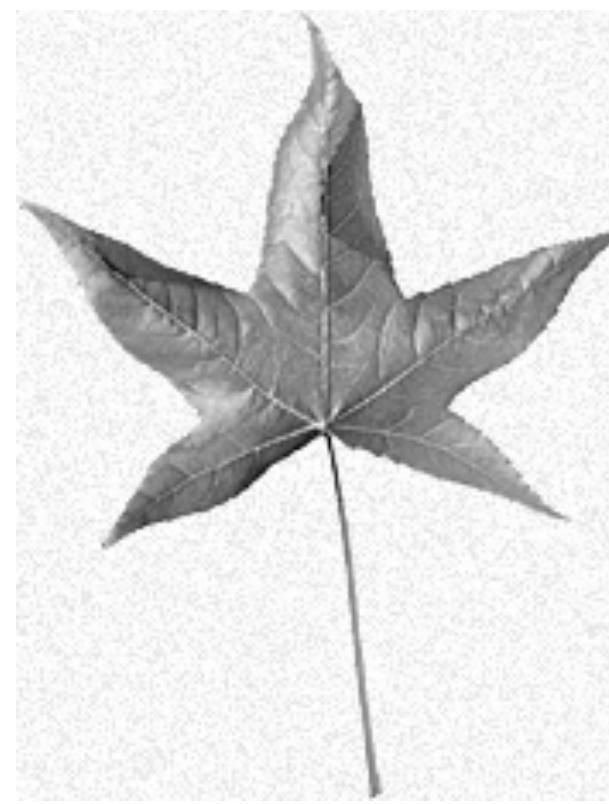


Thresholding

Binarization based on a threshold

$$s = \begin{cases} 1 & \text{if } r > r_0 \\ 0 & \text{otherwise} \end{cases}$$

r_0 = selected threshold



Quantization

$$s = \begin{cases} s_0 & \text{if } r_{\min} \leq r < r_0 \\ s_1 & \text{if } r_1 \leq r < r_2 \\ s_2 & \text{if } r_2 \leq r < r_3 \\ \vdots & \\ s_n & \text{if } r_n \leq r \leq r_{\max} \end{cases}$$



Logarithm / Exponent

- Sometimes care more about *relative changes* than absolute ones
- Lots of things use logarithmic scales
 - Decibel (dB) units
 - Apparent brightness
 - Richter scale
 - Human Vision
- Can “undo” with exponentiation

$$s = \log(r)$$

$$s = e^r$$

Power Functions

Can also raise to a desired power:

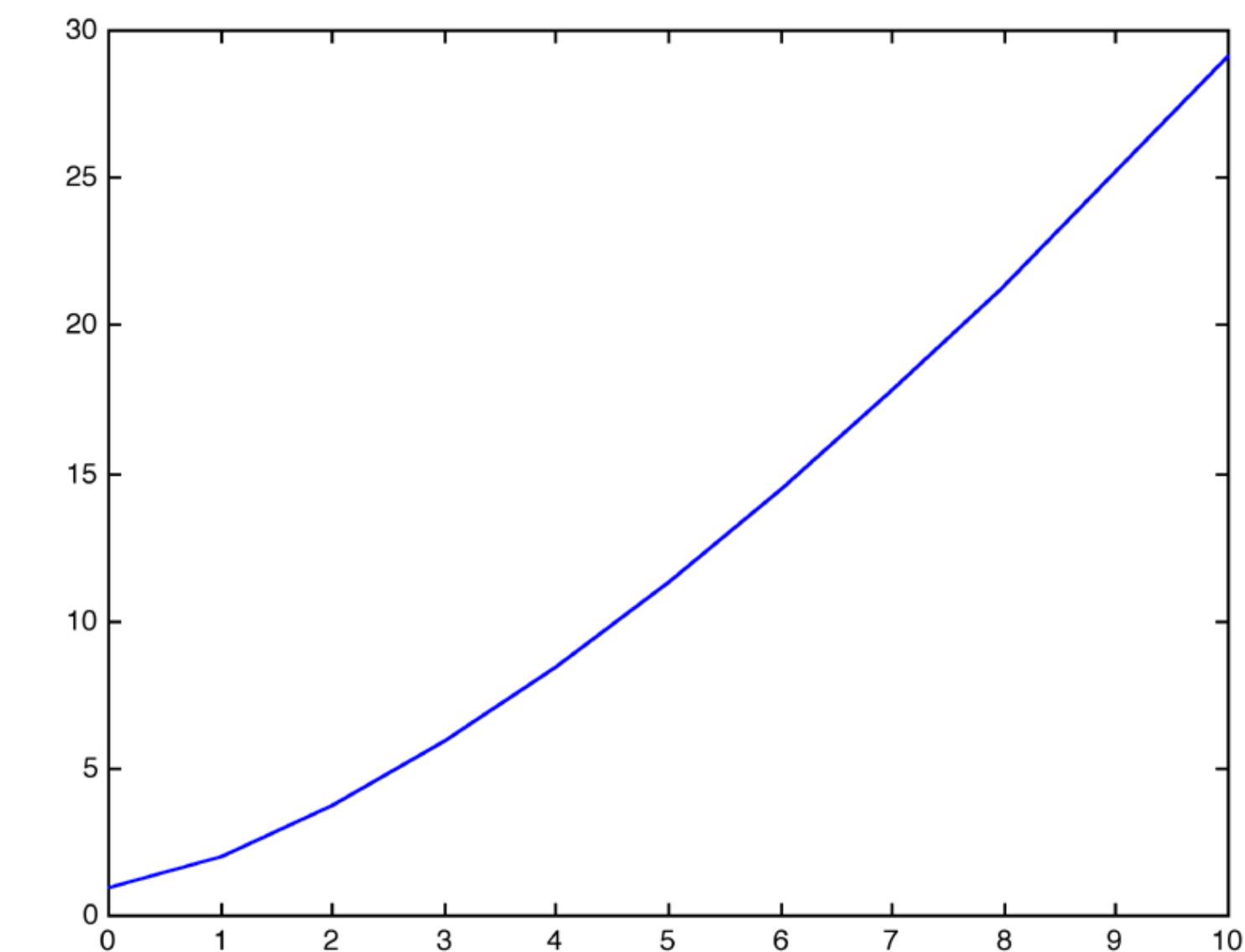
$$s = r^p$$

Gamma Responses

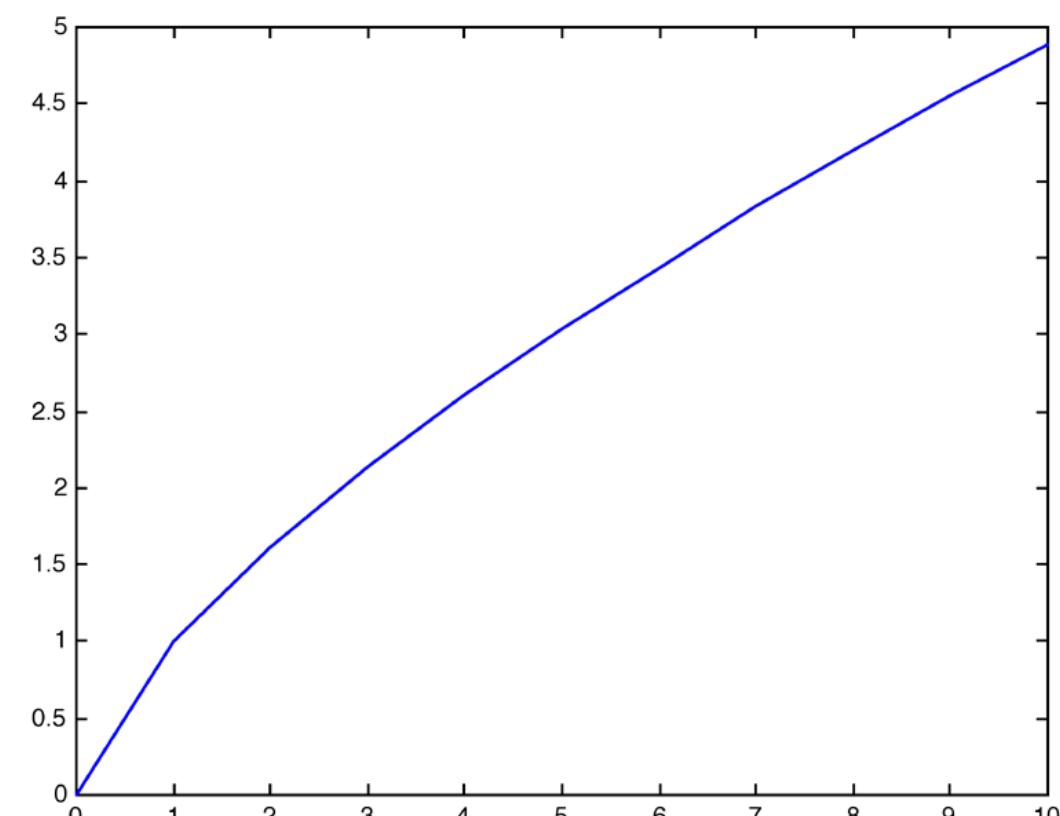
- Many devices have a nonlinear response:
- For a CRT, the intensity is related to the voltage by

$$I = V^\gamma + c$$

- The exponent is often called the “gamma” of the device

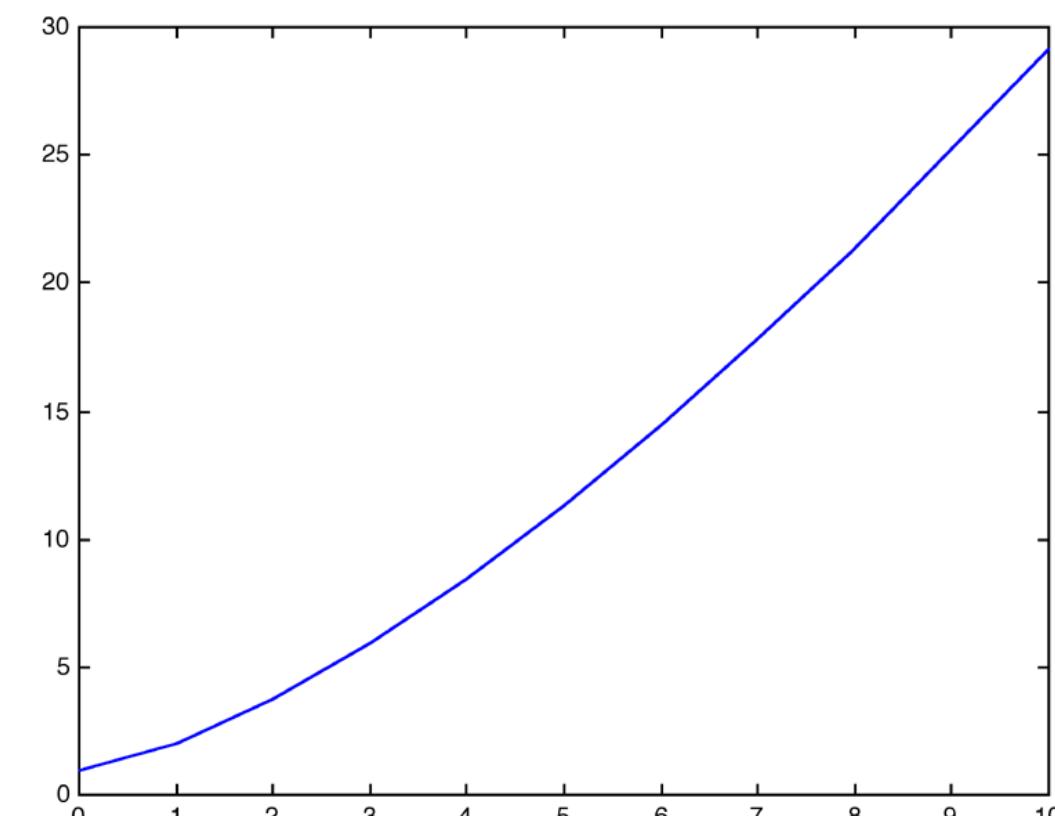


Gamma Correction



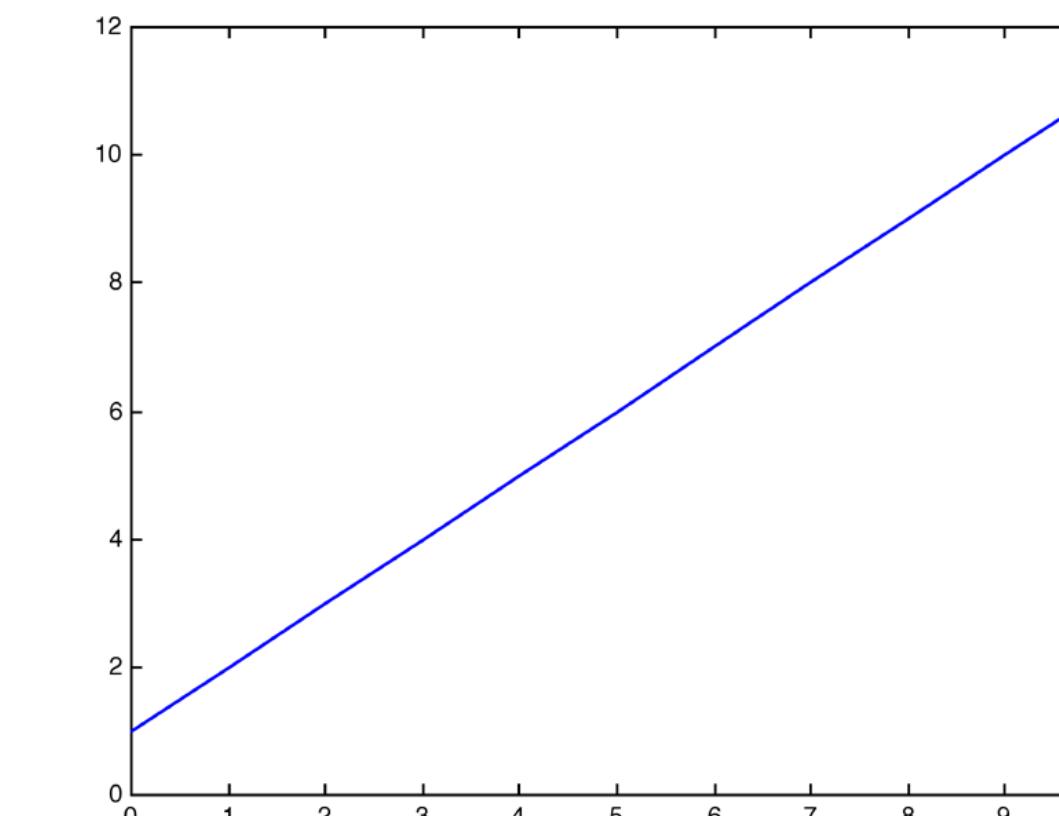
Preprocess

$$s = r^{1/\gamma}$$



Device

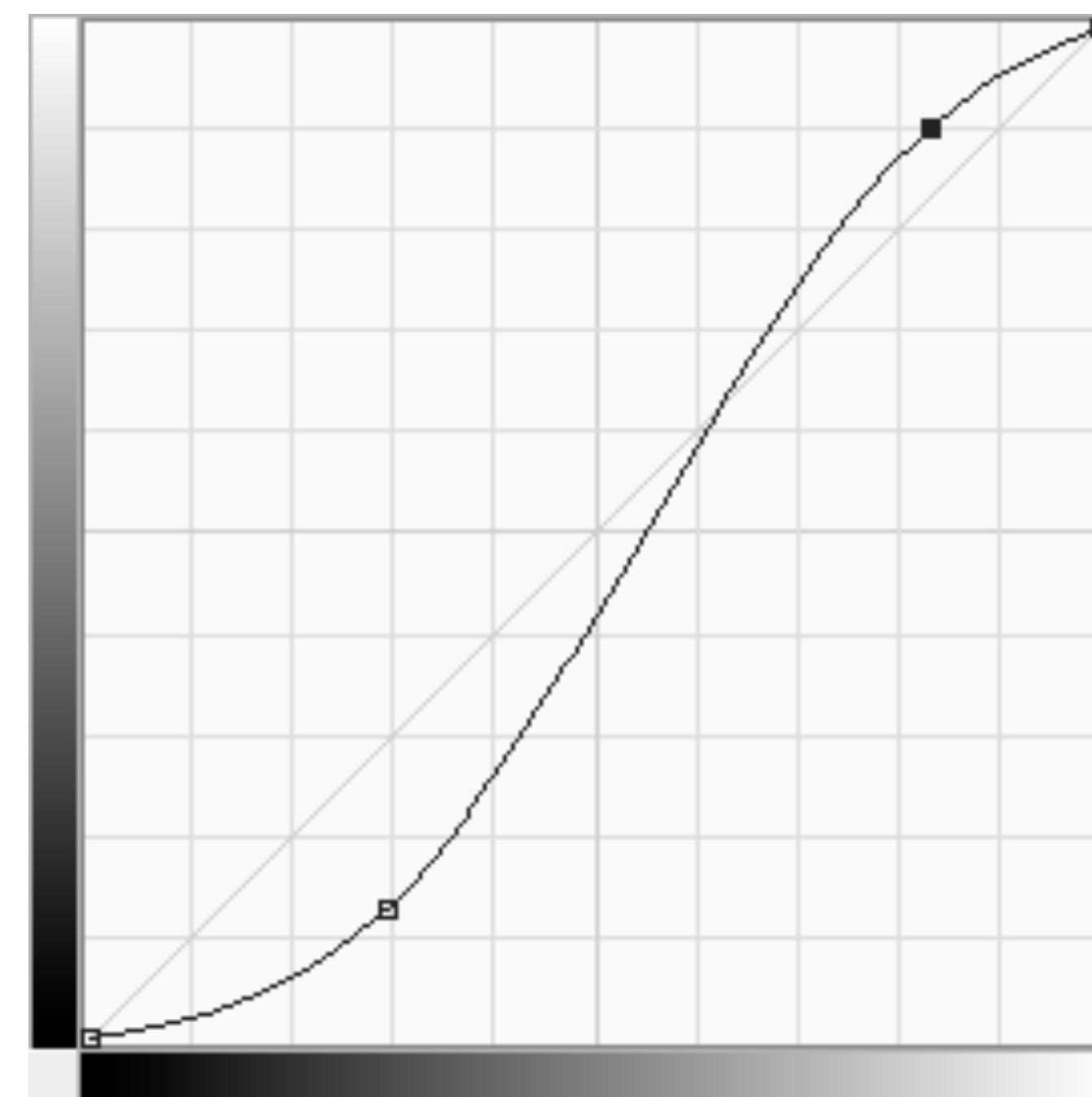
$$I = V^\gamma + c$$



Result

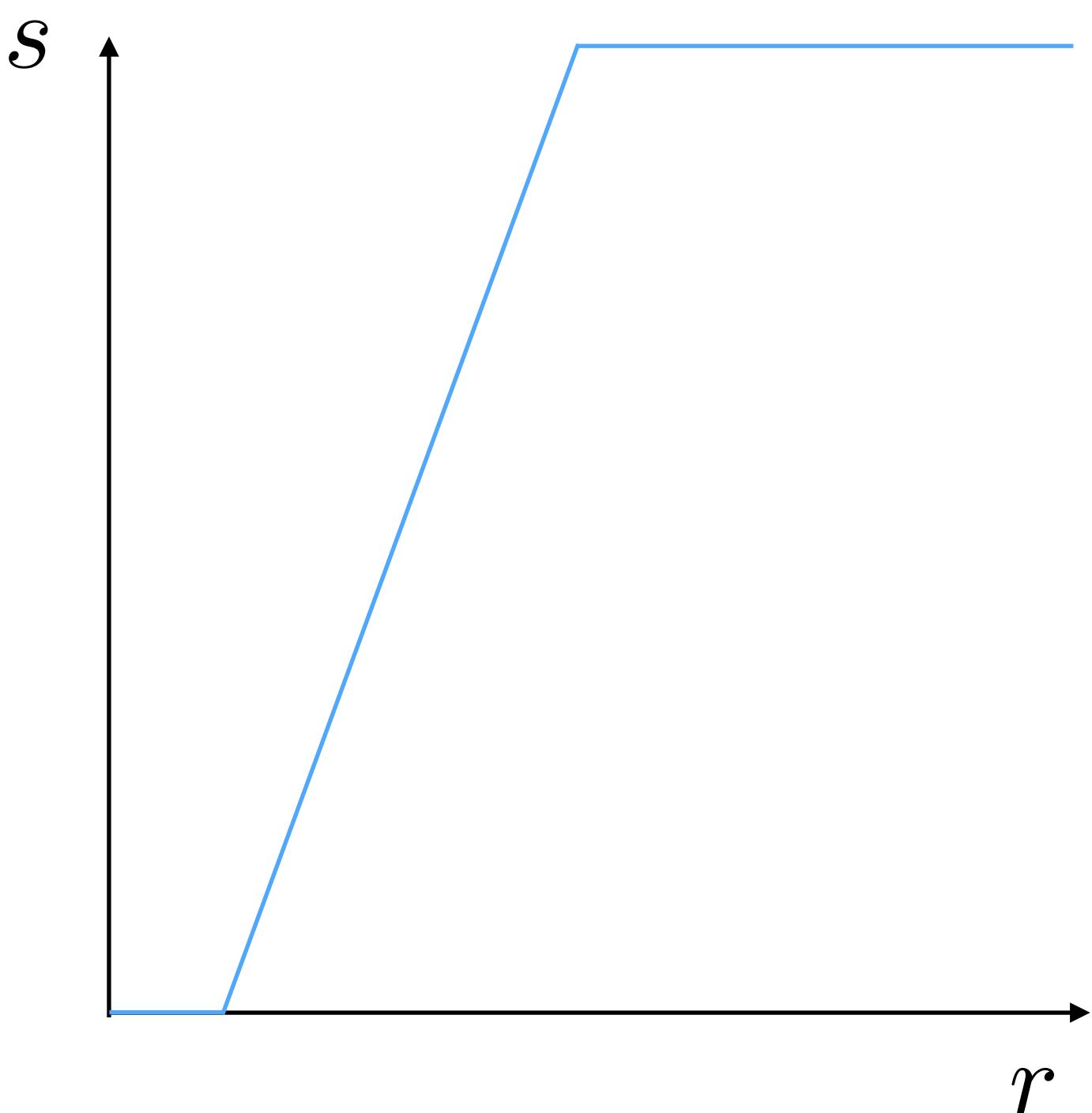
Contrast Enhancement

- *Contrast enhancement* makes differences more distinguishable
- Trades off *decreased contrast* in some part(s) of the range for *increased contrast* in the range we're interested in
- If we plot the function,
 - Slope > 1 means enhancement
 - Slope < 1 means reduction



Windowing

- *Windowing* is enhancement of *one part* of the image range
- Example:
Displaying 12-bit X-rays on an 8-bit screen
 - Simple: scale [0,4095] to [0,255] by dividing by 16
 - Better: if you know that what you're interested in is in the range [500,2000], *enhance that part of the range*



$$s = \begin{cases} 0 & \text{if } r < 500 \\ 255(r - 500)/(2000 - 500) & \text{if } 500 \leq r < 2000 \\ 255 & \text{if } r \geq 2000 \end{cases}$$

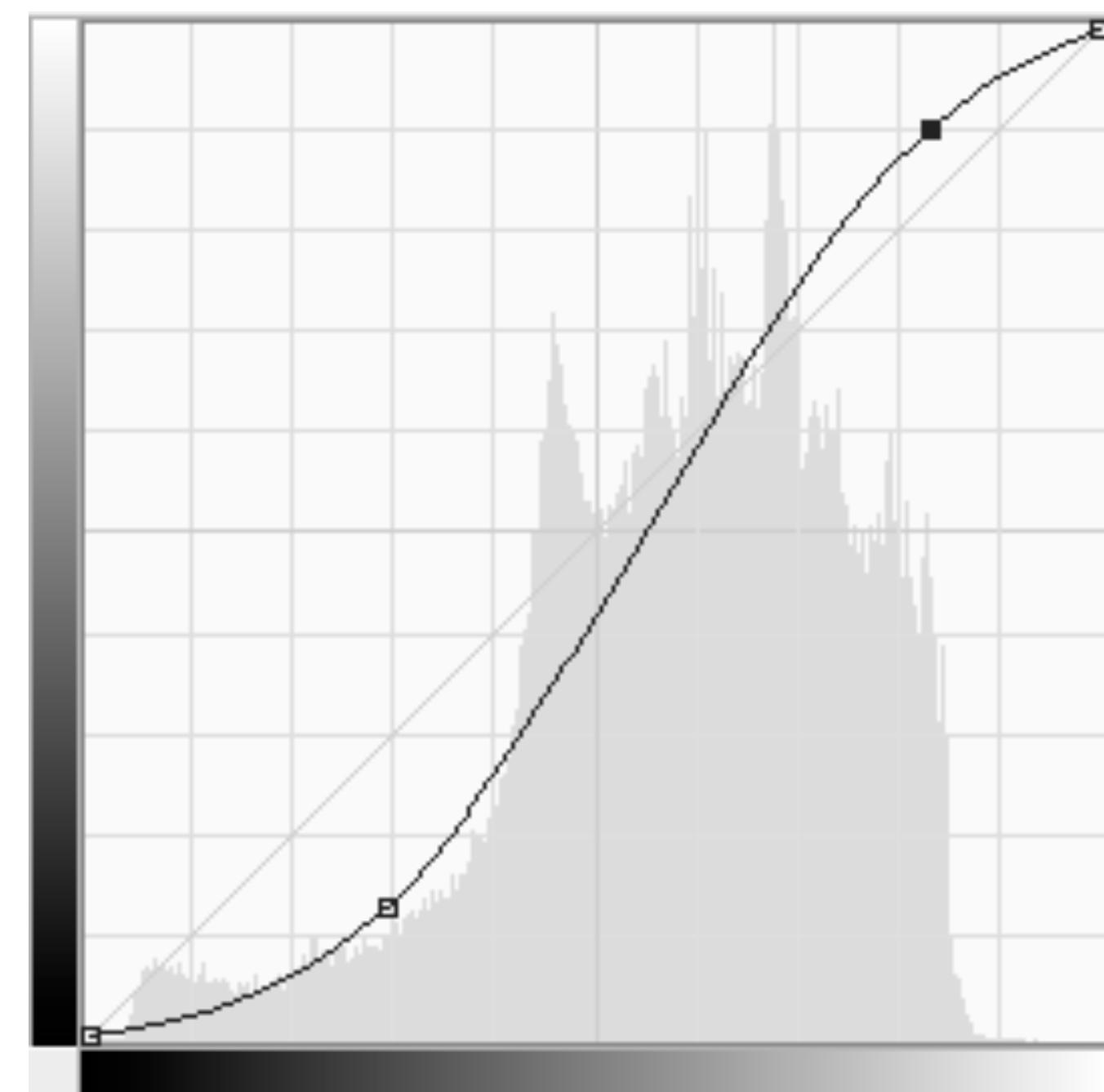
Windowing

- Another example:
 - Want to better see the reflection of the keys
 - Enhance this range at expense of contrast in brighter or darker areas
- If the full range is already used, contrast enhancement is a *zero-sum game*



Histogram Equalization

- Can try to automatically optimize contrast by allocating according to brightness distribution (histogram)
- This is called *histogram equalization* because it tries to spread contrast evenly
- Strong discrimination of detail, but not always good “real looking” result



Histogram Equalization



Coming up...

- Color image processing
- Interimage: blending, masking, differencing, compositing
- Neighborhood operations:
 - noise reduction
 - sharpening
 - edge detection



Color Concepts

CS 355: Introduction to Graphics and Image Processing

Light and Wavelengths

- Visible light is in the range 400 nm (blue) to 700 nm (red)

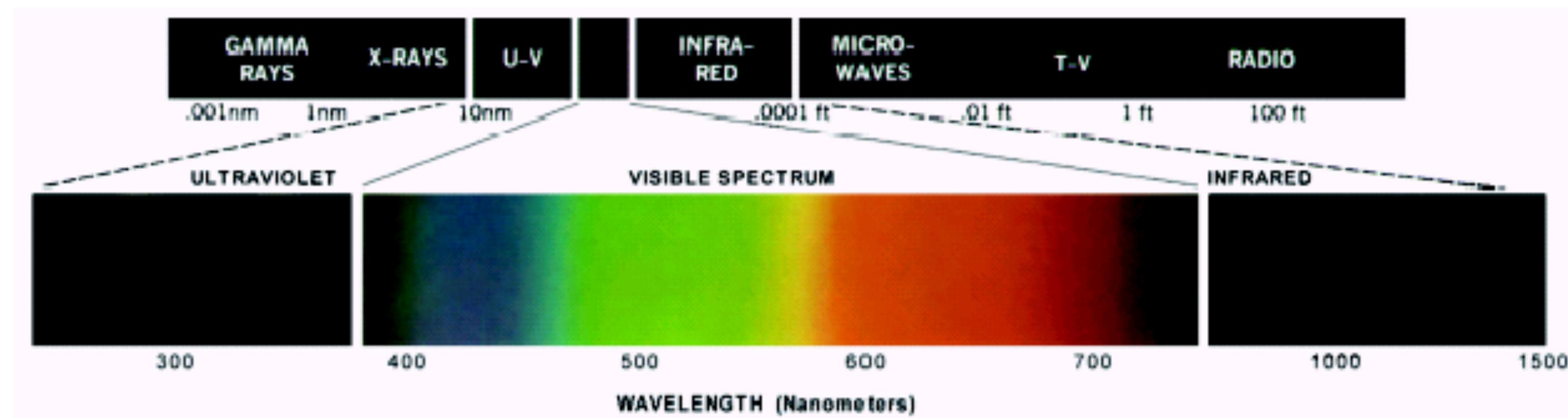


FIGURE 6.2 Wavelengths comprising the visible range of the electromagnetic spectrum.
(Courtesy of the General Electric Co., Lamp Business Division.)

Perception of Light

- Cones have three different kinds of color-sensitive pigments, each responding to a different range of wavelengths
- These are roughly “red”, “green”, and “blue” in their peak response *but each responds to a wide range of wavelengths*
- The *combination* of the responses of these different receptors gives us our color perception
- *Multiple combinations of wavelengths can produce the same response*
- This is called the *tristimulus model* of color

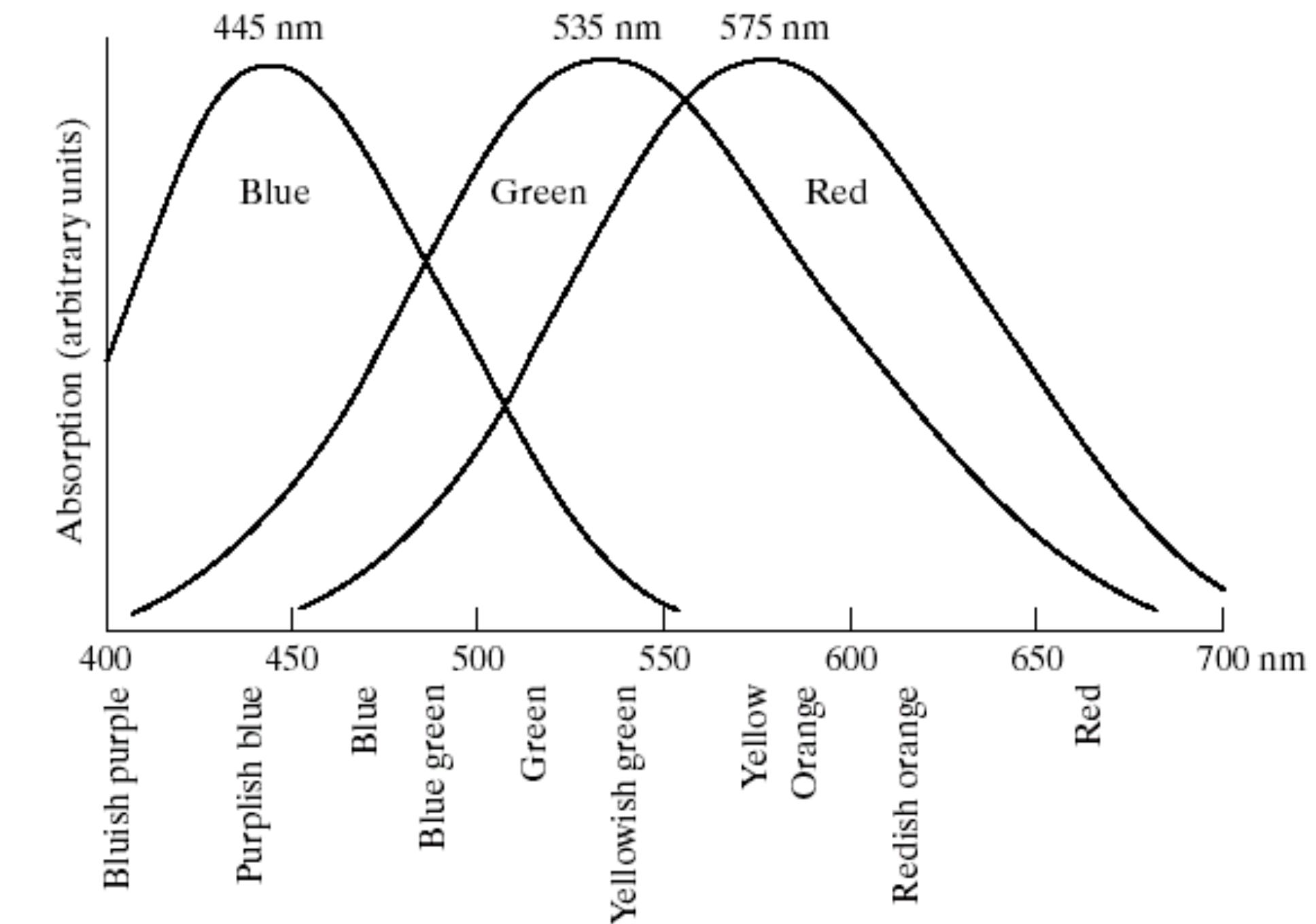
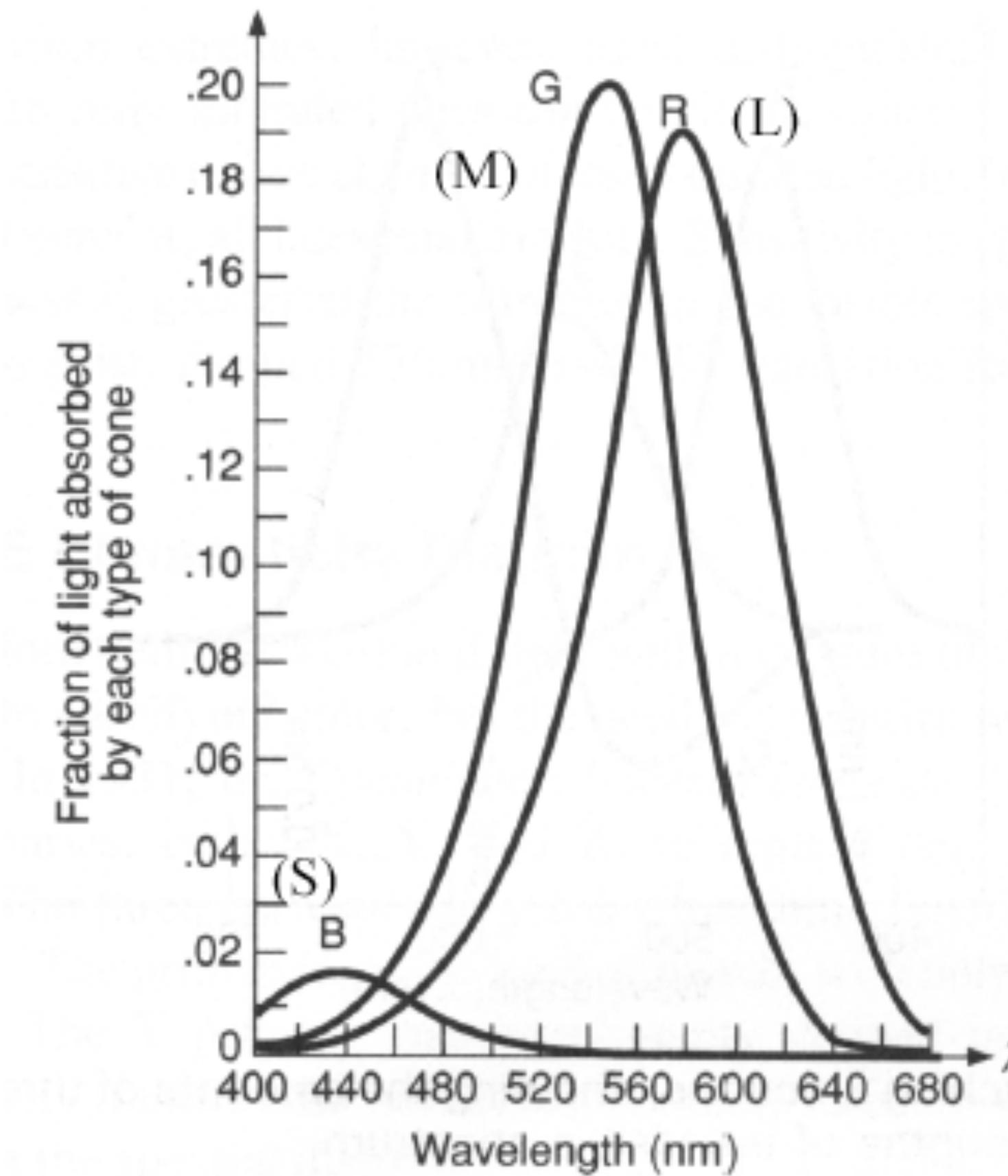


FIGURE 6.3 Absorption of light by the red, green, and blue cones in the human eye as a function of wavelength.

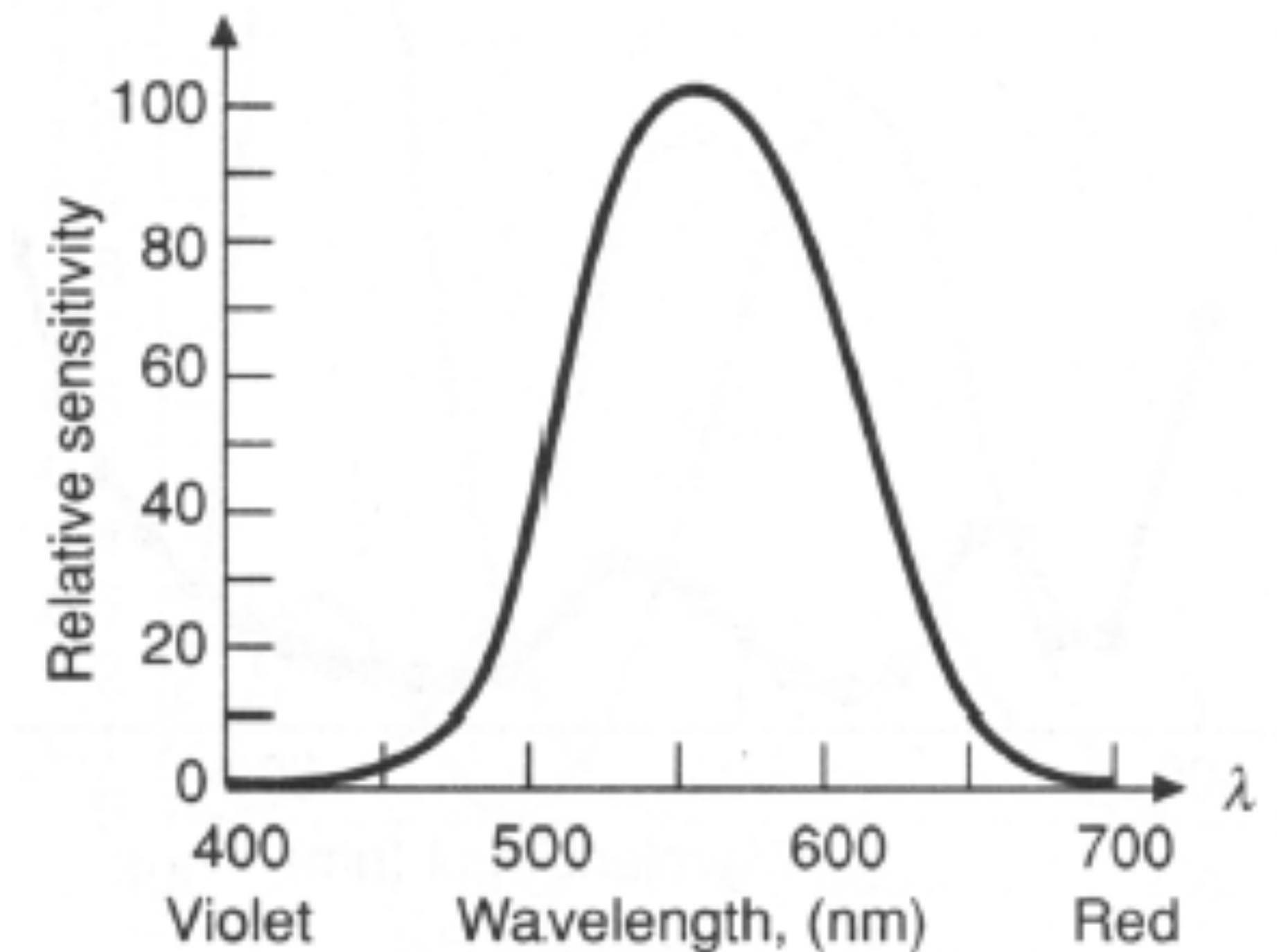
Perception of Light

- The sensitivity and number of the three types of cones are different
- More sensitive overall to green and red than to blue



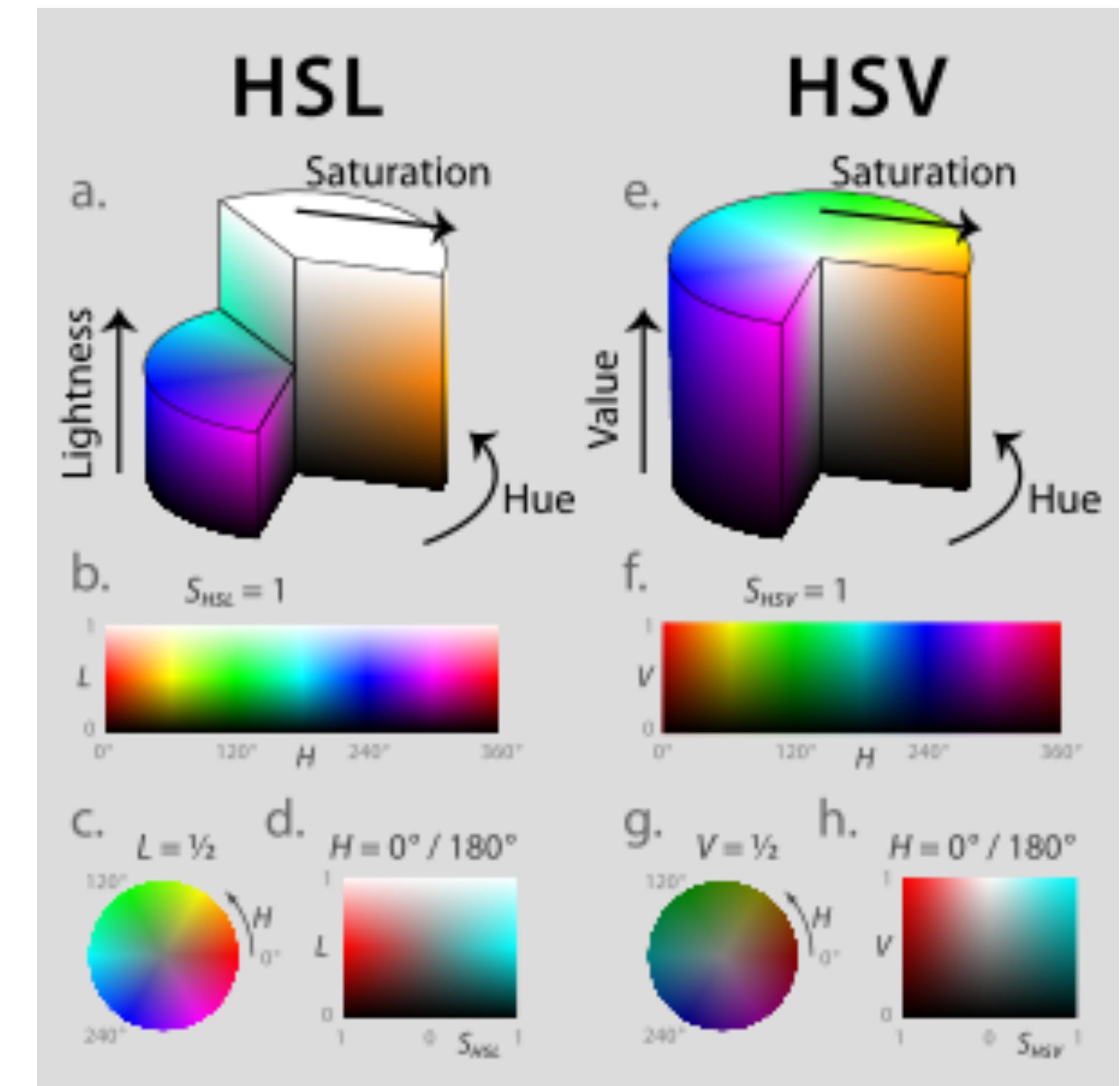
Intensity vs. Brightness

- Technically, there is a difference between
 - physical intensity
 - perceptual brightness
- Caused by differences in sensitivity of our eyes to different wavelengths
- *Luminous efficiency function*



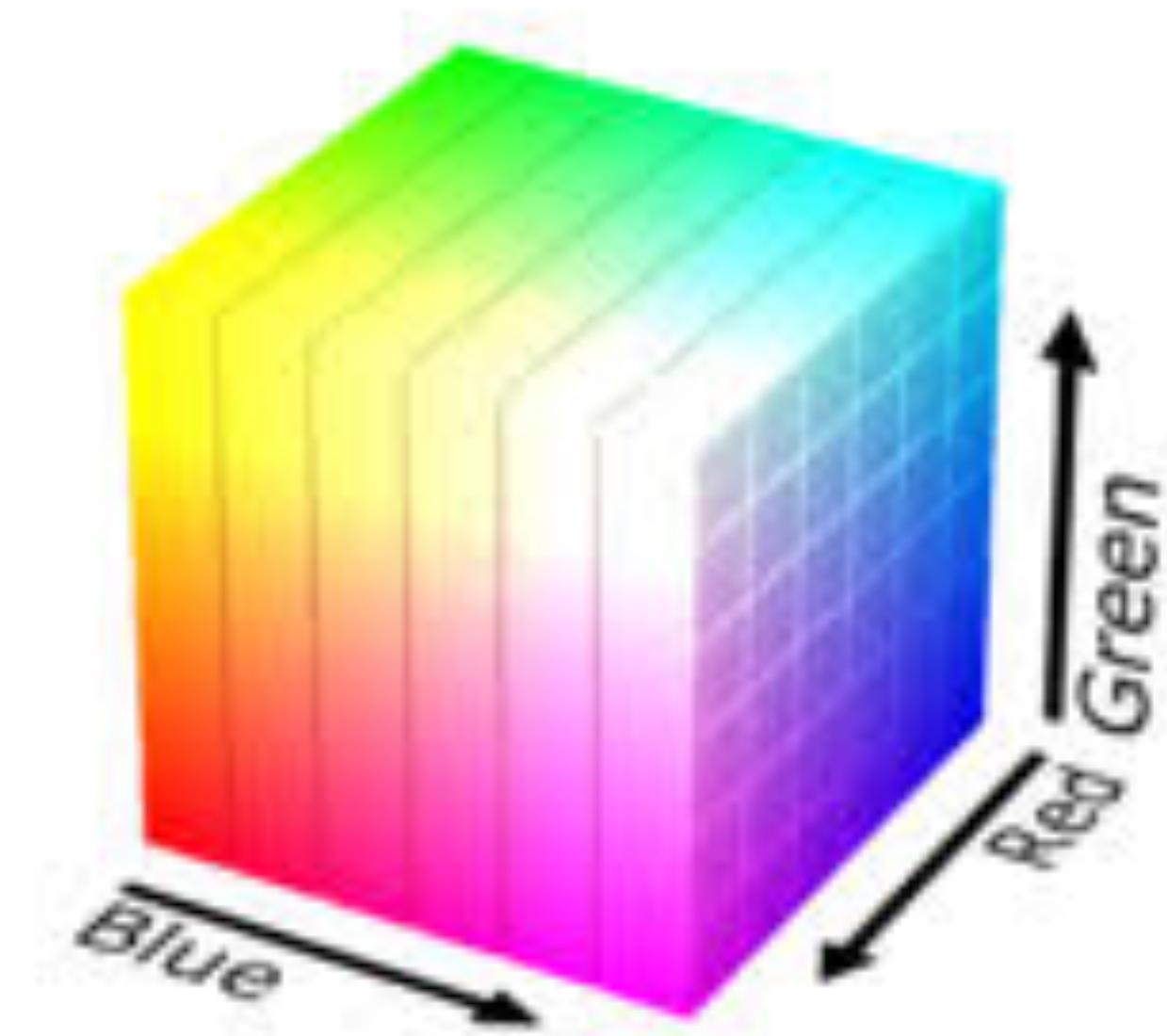
Color Models

- Color is a natural phenomenon, but for graphics and imaging we need to represent colors numerically
- There are many ways to do this
- Numerical representations of color are called *color models*



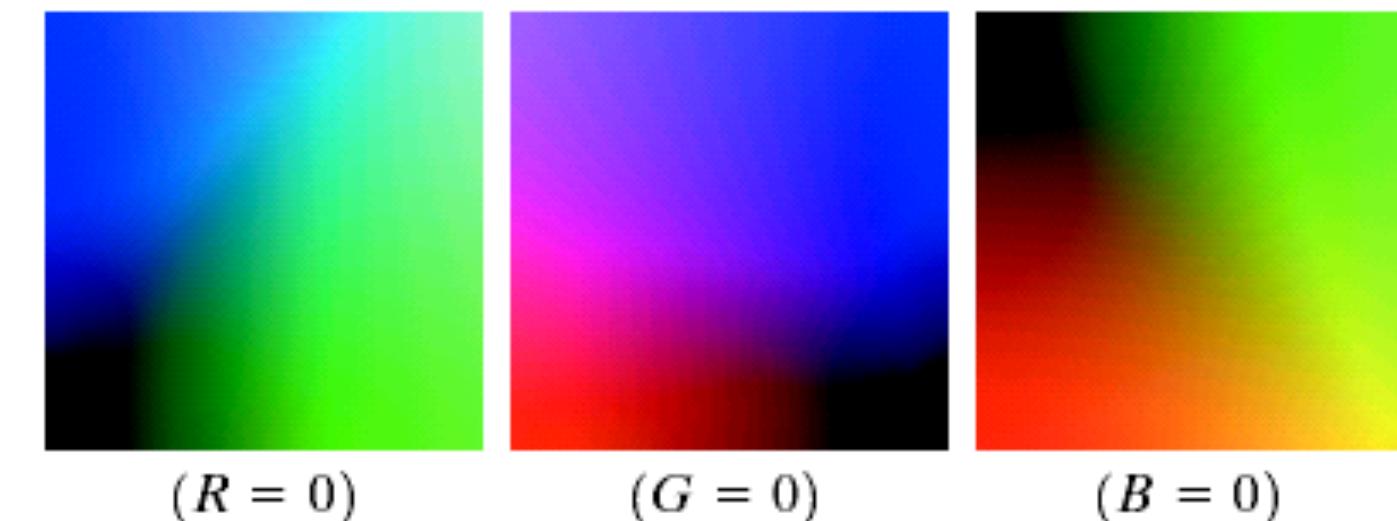
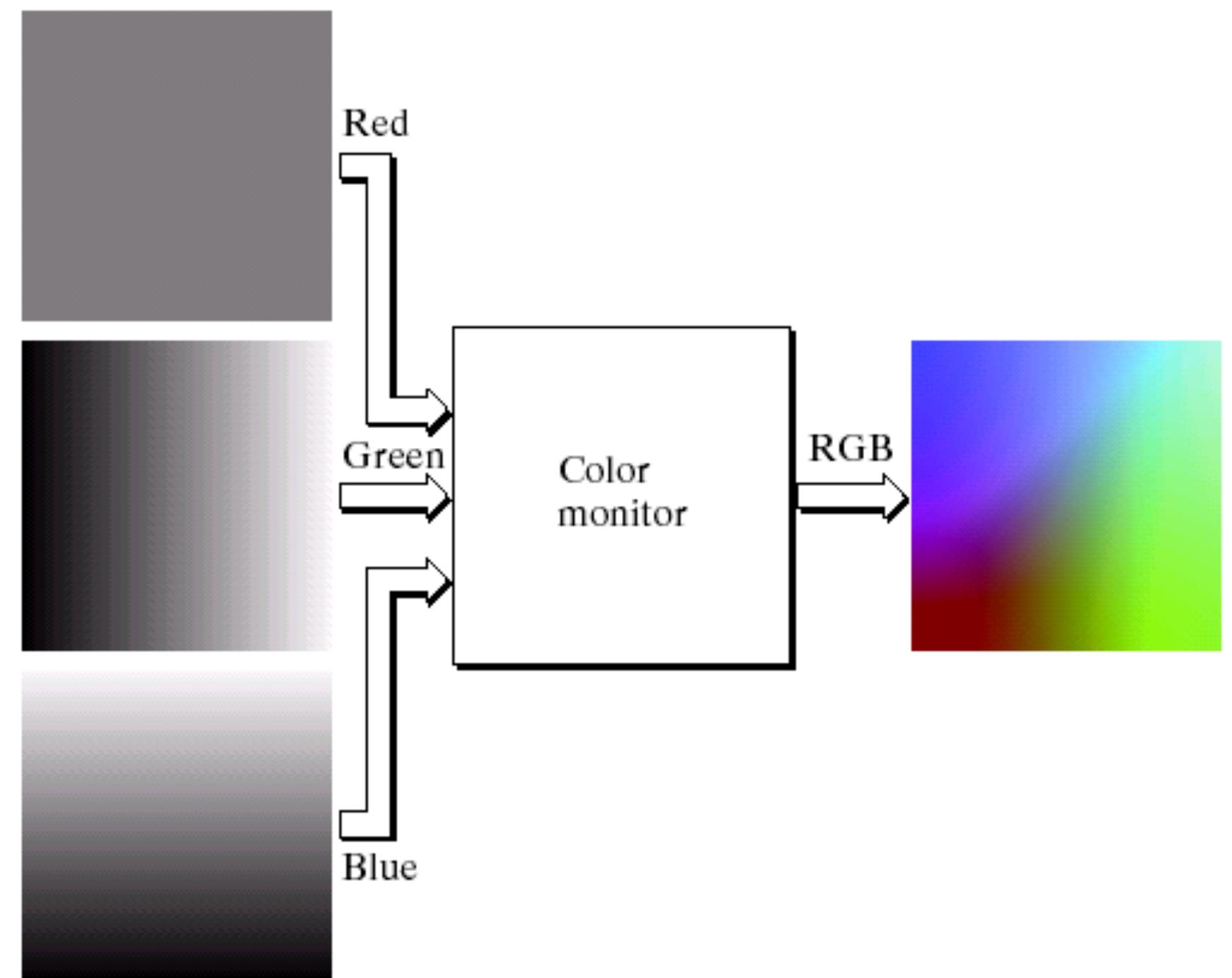
RGB Color Model

- Simplest model is just to store red, green, and blue values
- Colors can be thought of as points in a RGB cube



Color Channels

- Can think of an RGB image as distinct *color channels* or *color planes*
- Doesn't matter how you store it:
 - An RGB triplet for each pixel
 - Three separate images



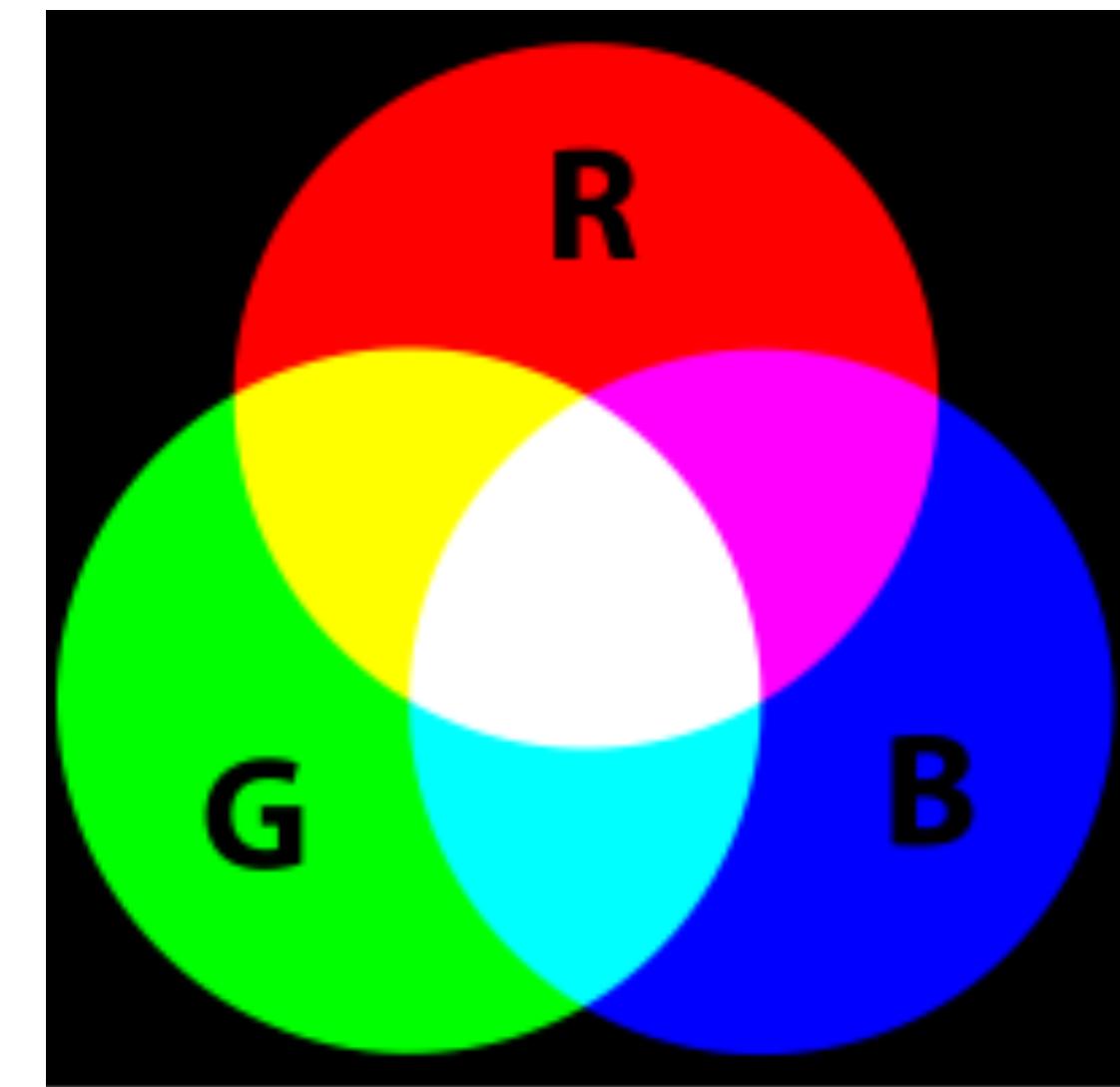
Primaries and Secondaries

- Primary colors:
ones mixed to make other colors
- Secondary colors:
pairwise combinations of primaries
- Primaries and secondaries are *complementary*
- Can be *additive* or *subtractive*

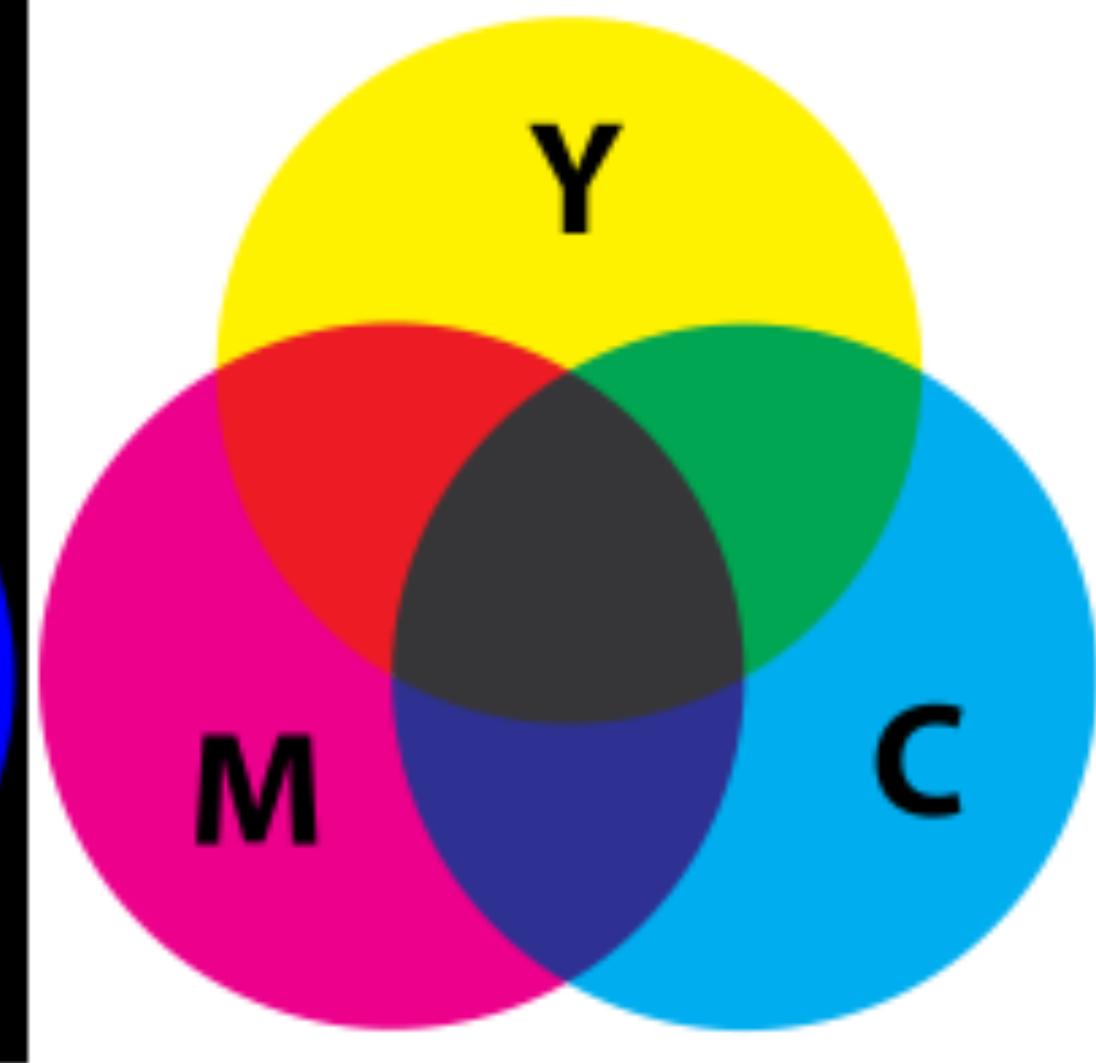


CMY Model

- Subtractive media absorb light
 - Real-world objects
 - Paint
 - Printing ink
- The perceived color is what is *not* absorbed
- CMY uses subtractive primaries: cyan, magenta, yellow



Additive RGB Color



Subtractive CMY(K) Color

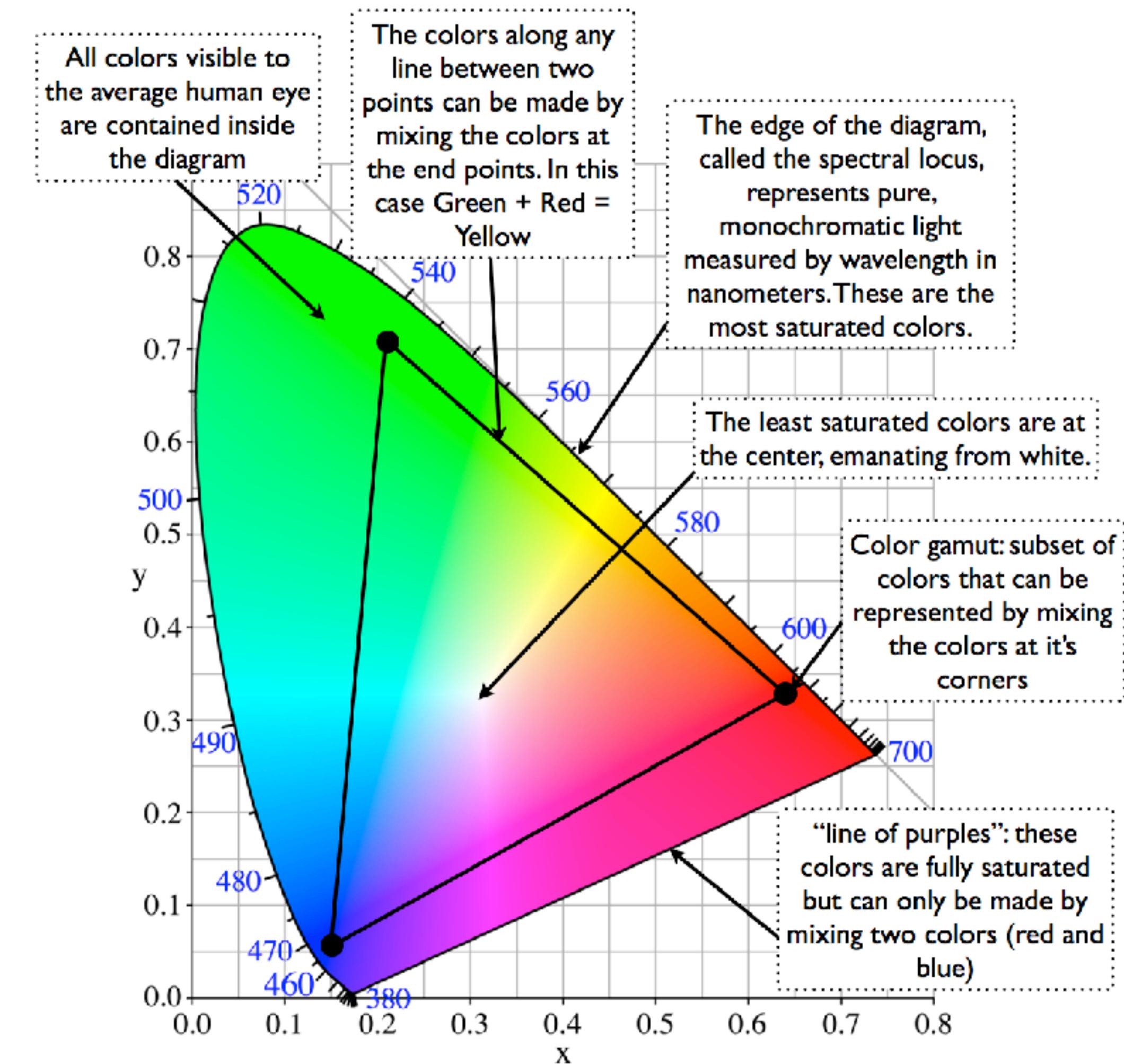
CMYK Model

- It's difficult and expensive to make pure subtractive primaries
- Many systems introduce pure black as a fourth primary (K)



Color Gamuts

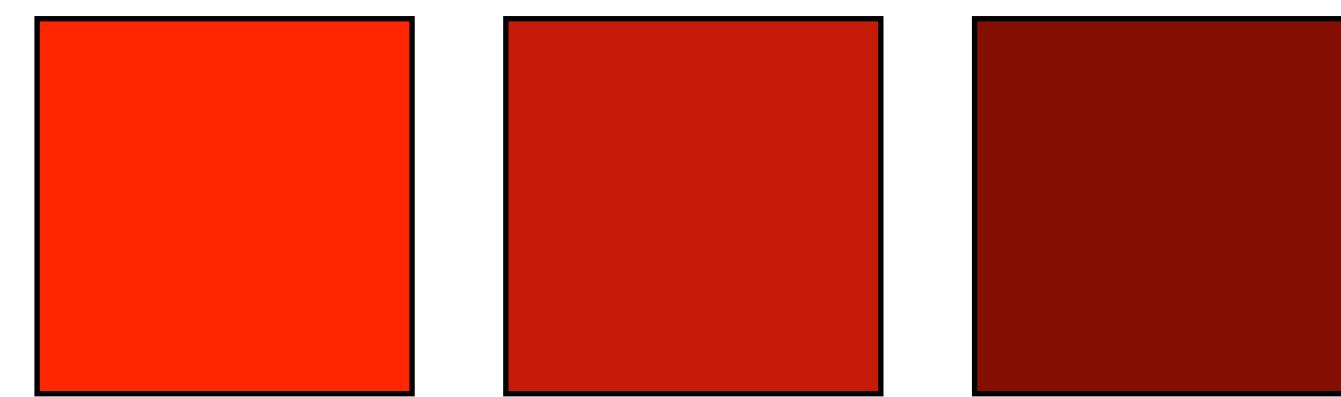
- Can visualize the space of all visible colors using a *chromaticity diagram*
- Important points:
 - No three primaries can span the space of visible colors
 - Different primaries will cover different parts of the space
 - *Colors producable on one device may not be producable on another*
 - *Even if they can be reproduced, the mix of primaries might be different—requires calibration*



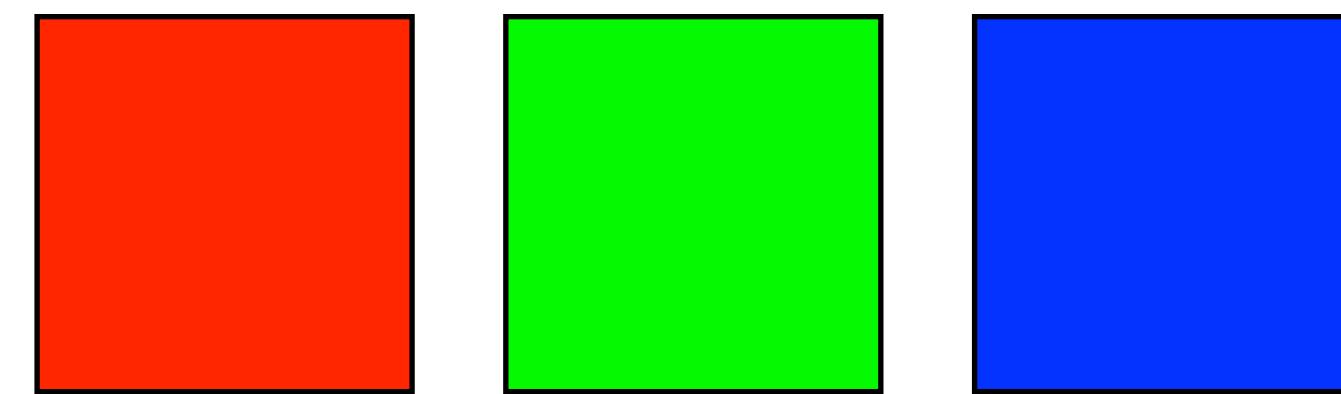
Anatomy of a CIE Chromaticity Diagram

Luminance and Chromaticity

- RGB model is common but not all that intuitive to use
 - Artists tend to think more in terms of luminance separate from chromaticity
 - Much more intuitive:
 - Luminance - how bright
 - Hue - the “color” (wavelength)
 - Saturation - how pure
- (lots of variations on this idea)



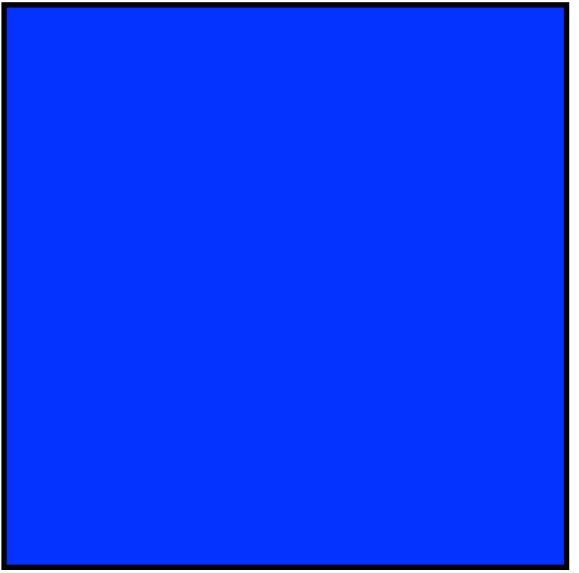
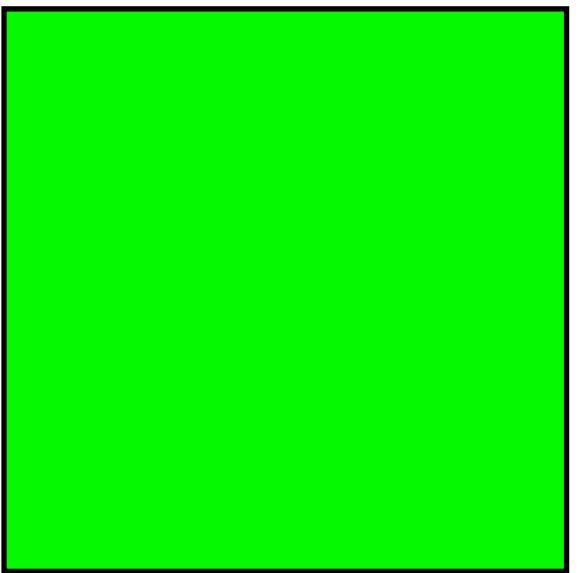
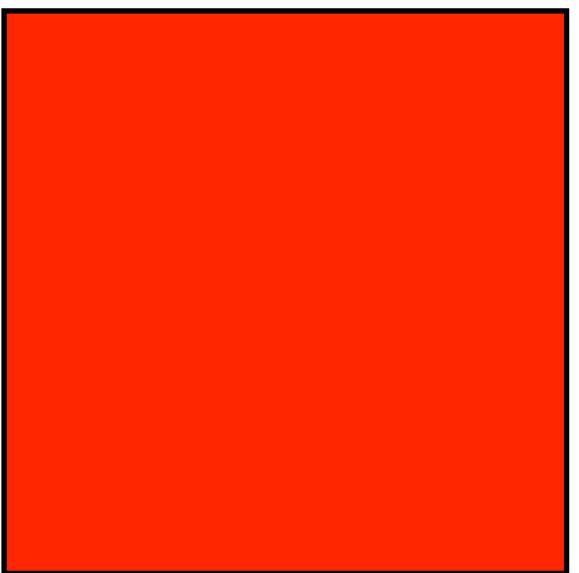
Different brightnesses



Different hues

Hue

- What we first think of as “color”
- Pure wavelength of light
- Artistically most important
- Example:
red vs. green vs. blue



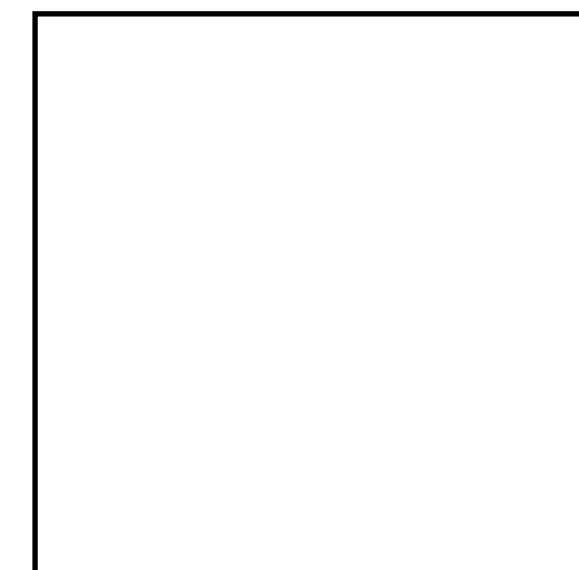
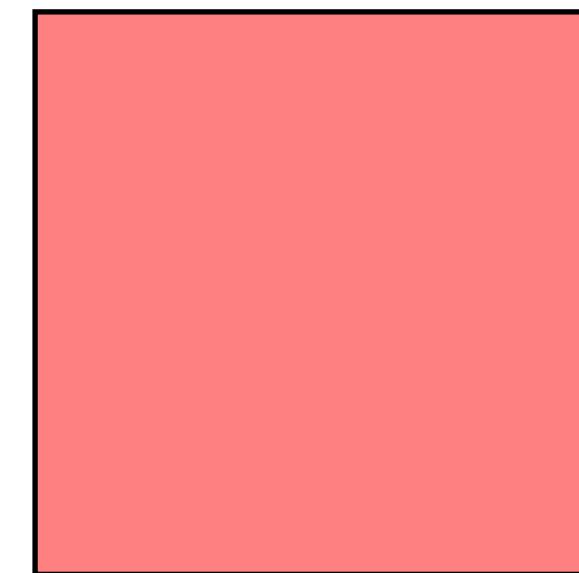
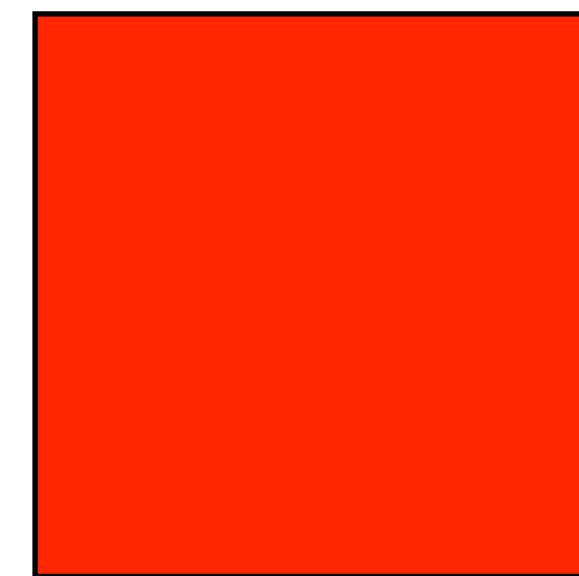


OH THE
HUE-MANATEE

A cartoon illustration of a manatee. The manatee's body is a vibrant, multi-colored gradient transitioning through yellow, orange, red, green, blue, and purple. It has a small, dark eye, a pink nose, and a gentle smile. Its front flippers are visible, and its tail is a bright pink color.

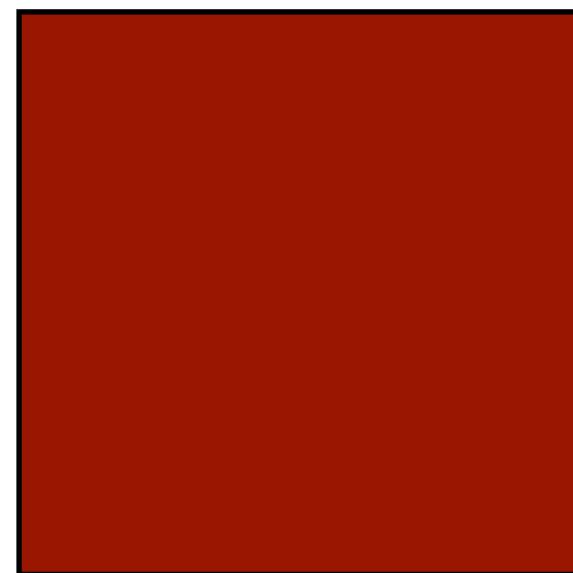
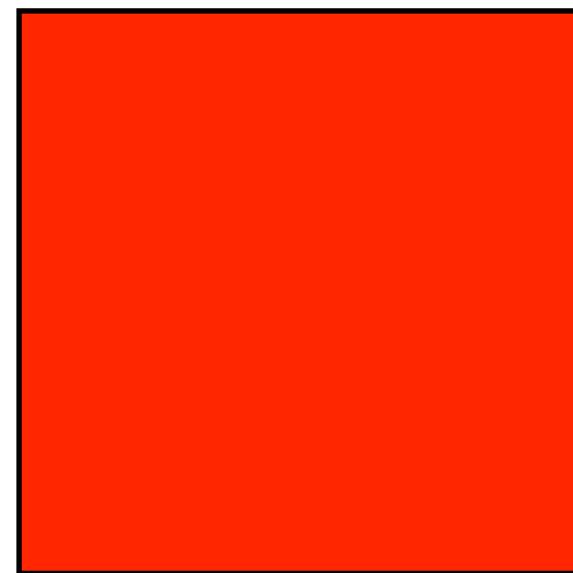
Saturation

- How pure a color is
- A single pure color is 100% saturated
- White, black, and gray are 0% saturated
- Examples:
 - red vs. pink
 - strong vibrant colors - high saturation
 - pastels - low saturation



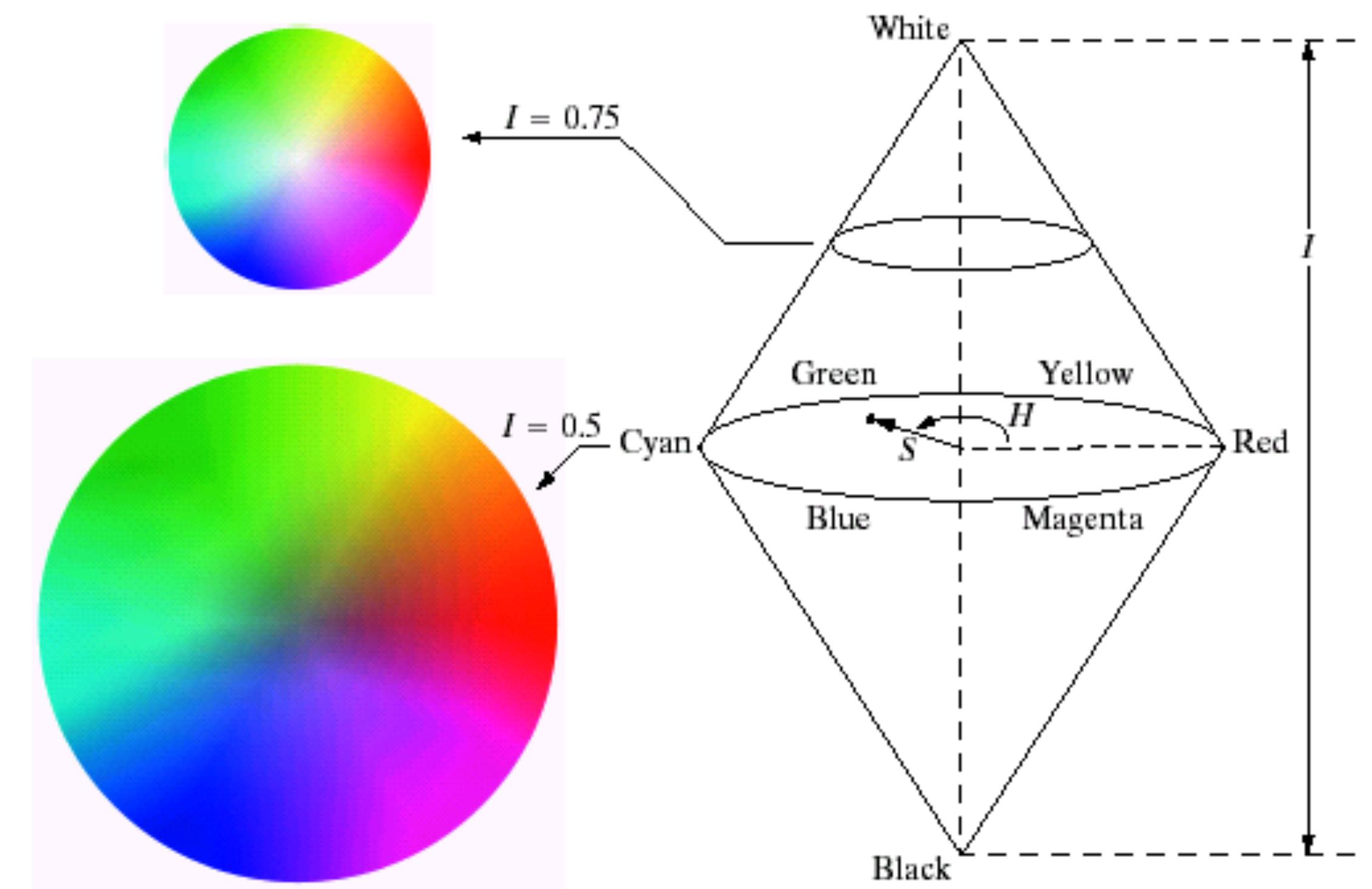
Luminance

- How bright /strong a color is
- Equivalent to measuring quantity of light independent of wavelength
- Most contributes to human perception of shape and form
- Lots of synonyms
 - Luminance
 - Intensity
 - Brightness (remember the difference)
 - Lightness
 - Luma
 - Value



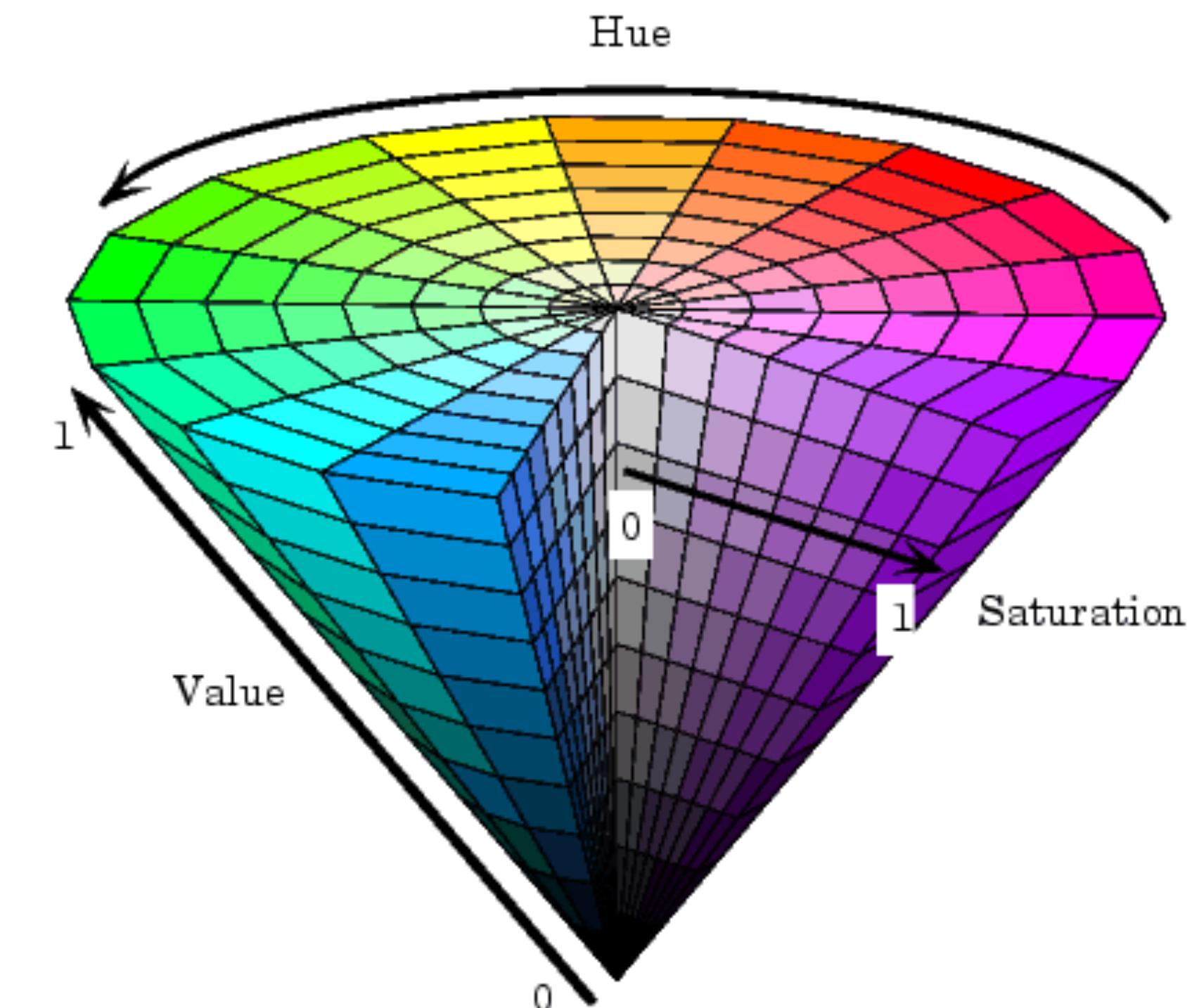
HSI Color Model

- One axis is intensity (luminance)
- The plane perpendicular to this axis represents chromaticity
 - Angle = hue
 - Distance from center = saturation
- Can map this plane using a triangle, hexagon, or circle
- The axis down the middle is the “line of grays”



Variations

- Hue-Saturation-Value (HSV)
 - Like HSI but with only one cone
- Hue-Lightness-Saturation (HLS)
- Hue-Value-Chroma (HVC)



NTSC YIQ Model

- Chromaticity requires two parameters,
but these don't have to be hue and saturation
- Lots of other variations
- The NTSC model was used in analog broadcast television
 - Y = luminance
 - I and Q = chromaticity

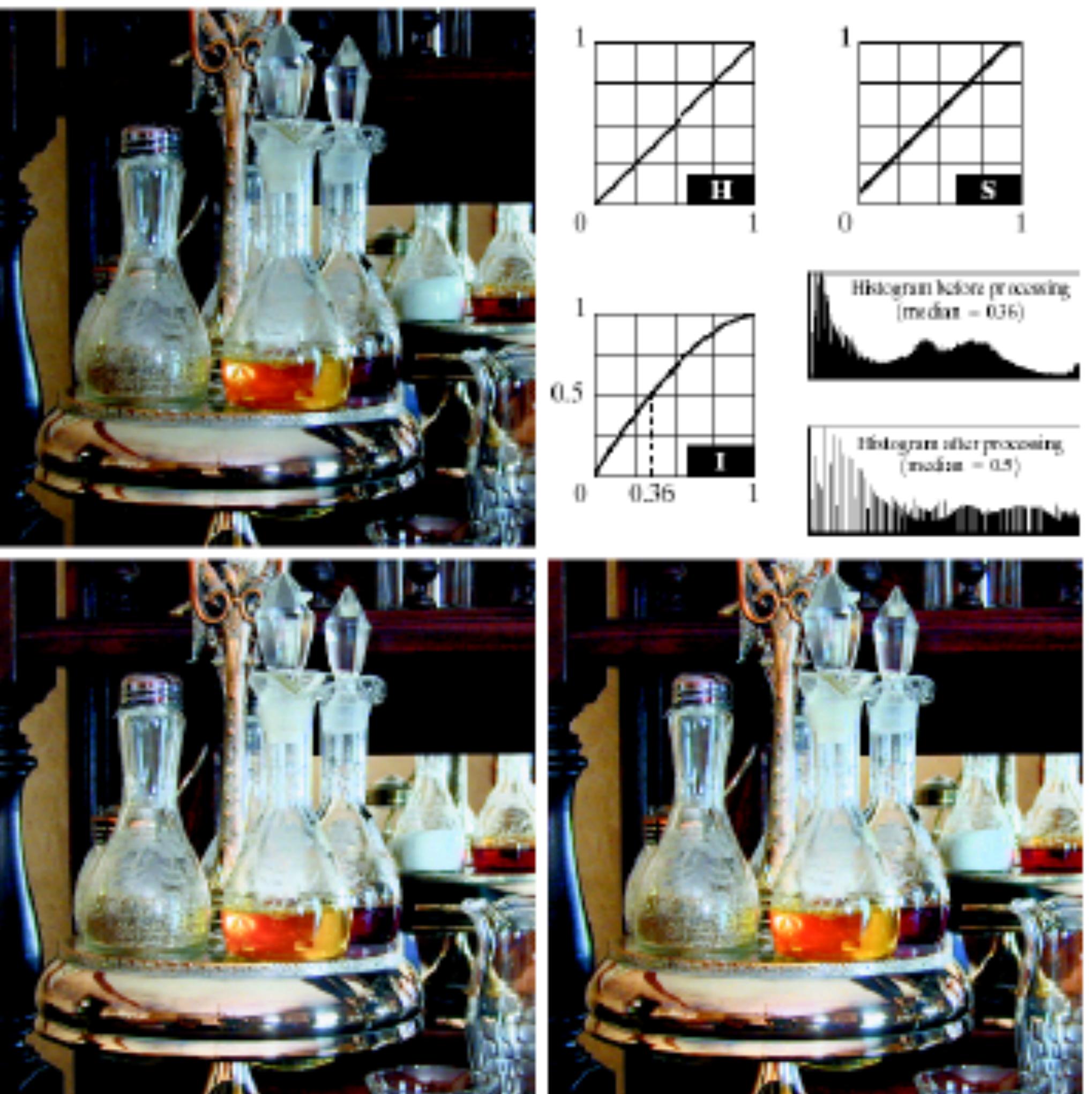
$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.532 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Other Chromaticity Representations

- CIE LUV
- CIE La^{*}b^{*} - attempts to be *perceptually linear*
- YCrCb - used in the JPEG standard

Color Image Processing

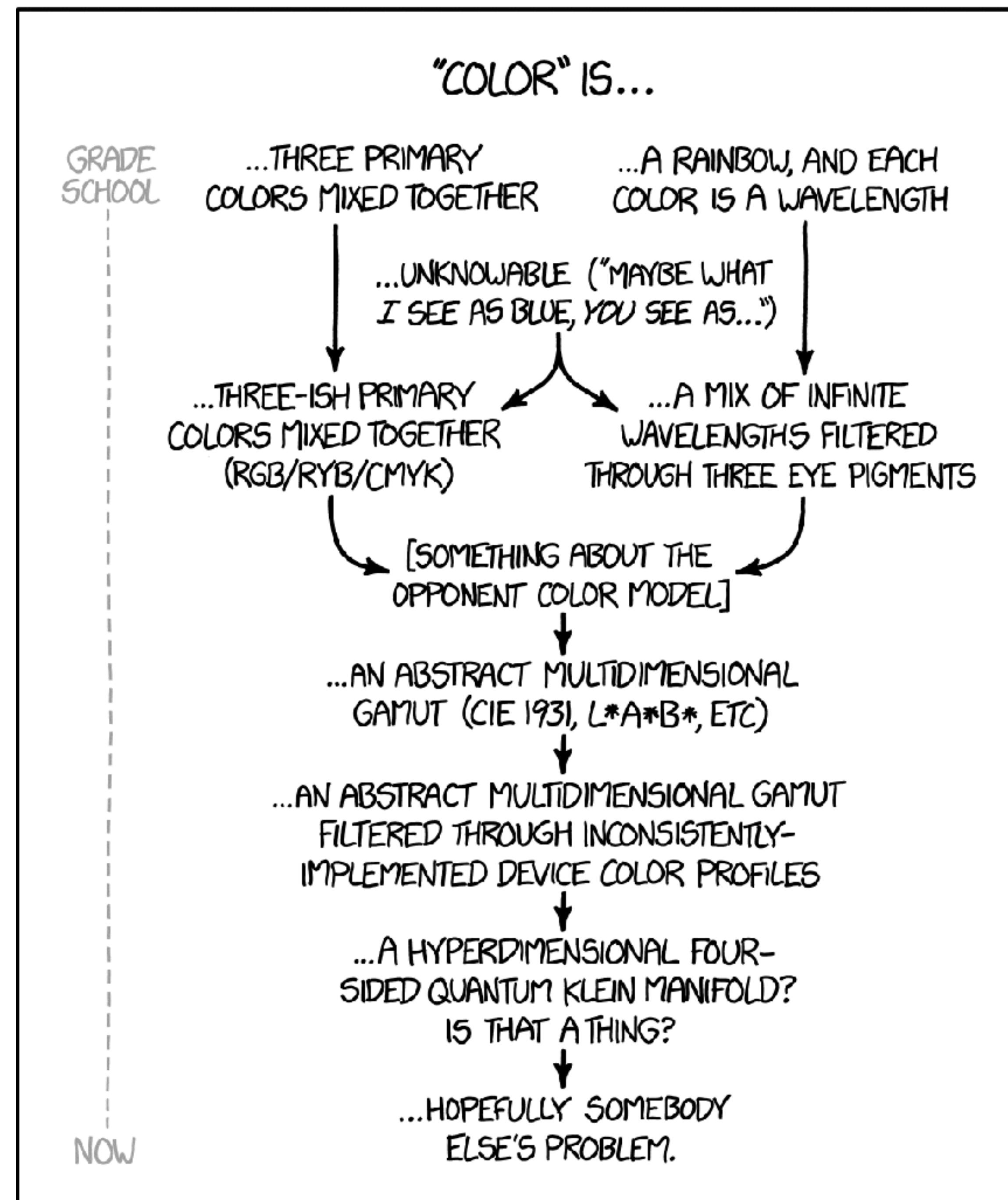
- Common approach:
 - Convert to an appropriate model such as HSI, etc.
 - Process
 - Brightness/contrast adjustments
 - Preserving or shifting the hues
 - Adjusting the saturation
 - Convert back to RGB if needed



Summary

- Pick a color space that makes sense for your application:
 - If the application is interactive, consider using HSI or other color space with explicit intensity/chromaticity
 - If you want to process intensities but keep hues the same, work (or at least think) in that kind of color space
- Keep physical limitations in mind:
 - Primaries, different color gamuts, etc.
- There's a lot more to color than RGB triplets!

EVOLUTION OF MY UNDERSTANDING OF COLOR OVER TIME:



Coming up...

- Interimage: blending, masking, differencing, compositing
- Neighborhood operations:
 - noise reduction
 - sharpening
 - edge detection

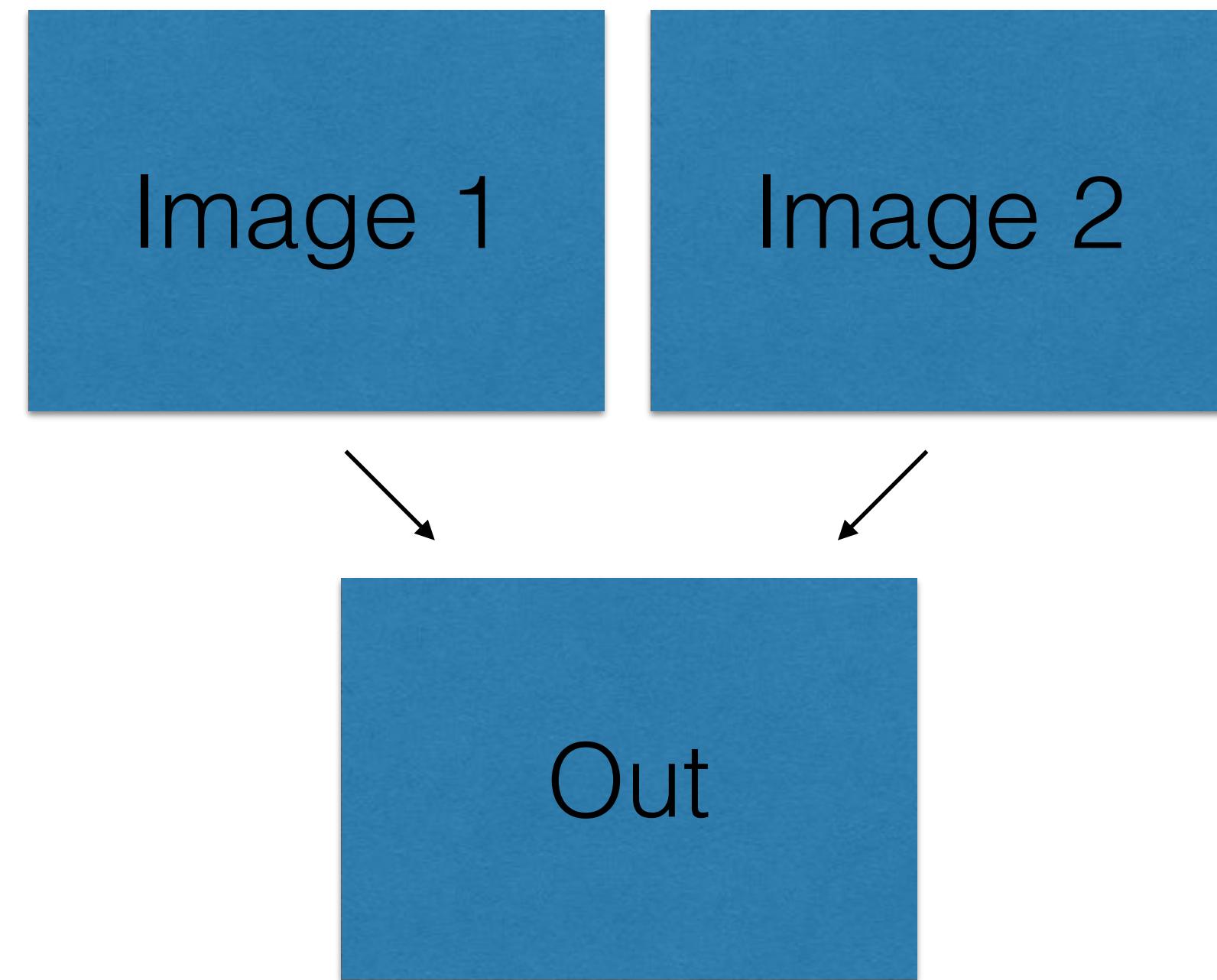


Blending, Differencing, and Masking

CS 355: Introduction to Graphics and Image Processing

Image Arithmetic

- Involve multiple images
- Apply a function pairwise (or more) to the pixels in the input images
- Possibilities:
add, subtract, and, or, min, max, ...



for all pixel positions x, y :
$$\text{out}[x,y] = \text{func}(\text{in1}[x,y], \text{in2}[x,y], \dots)$$

Addition

- Can be used for double exposures or composites
- Often a weighted blend



$$\text{out}(x, y) = \text{in}_1(x, y) + \text{in}_2(x, y)$$

$$\text{out}(x, y) = \alpha_1 \text{in}_1(x, y) + \alpha_2 \text{in}_2(x, y)$$

Subtraction

- Useful for finding changes between images

$$\text{out}(x, y) = \text{in}_1(x, y) - \text{in}_2(x, y)$$

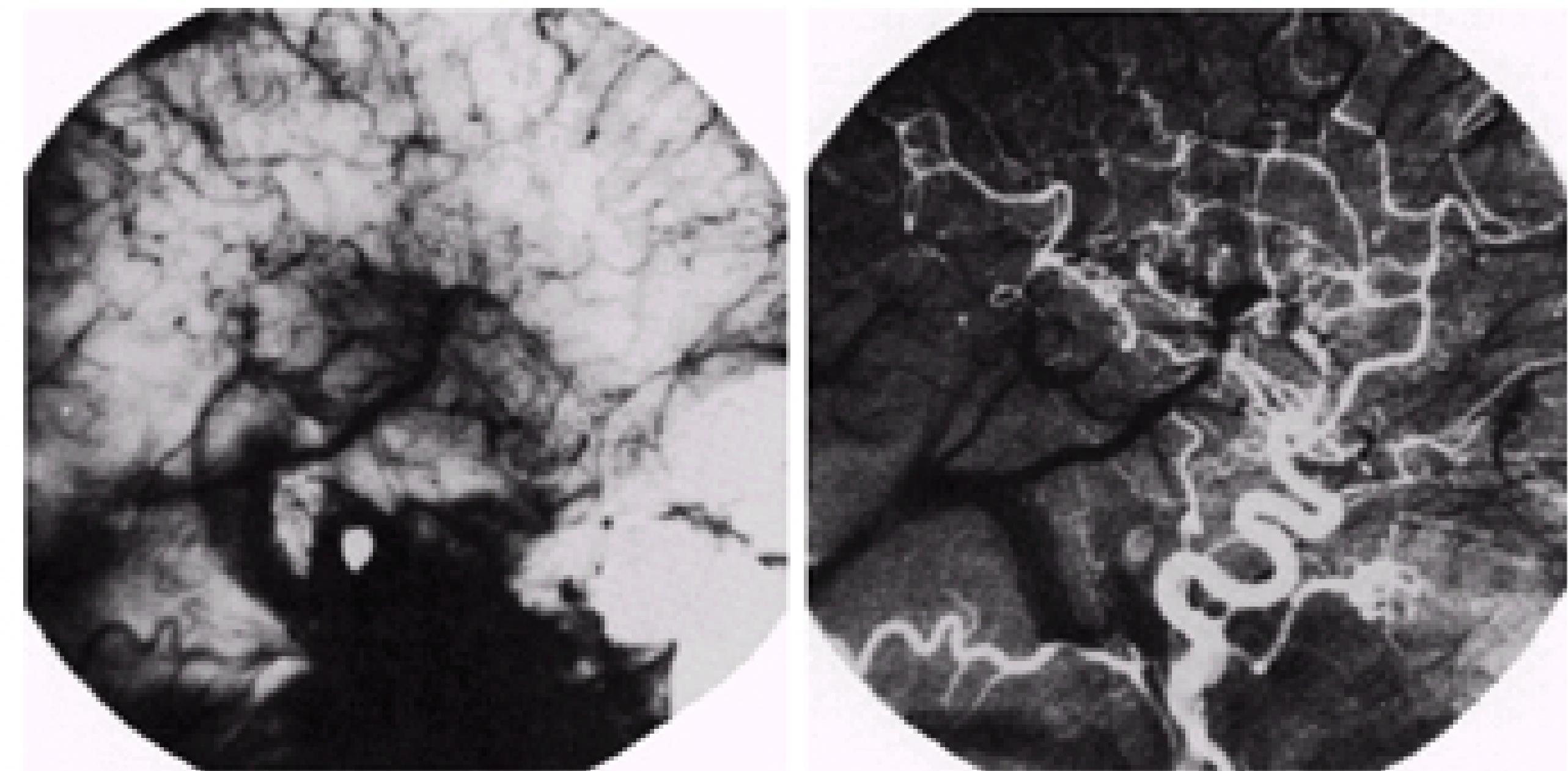
- Often more useful to use *absolute difference*

$$\text{out}(x, y) = |\text{in}_1(x, y) - \text{in}_2(x, y)|$$



Digital Subtraction Angiography

1. Take an x-ray
2. Inject patient with radio-opaque dye (“don’t move!”)
3. Take another x-ray
4. Subtract the two

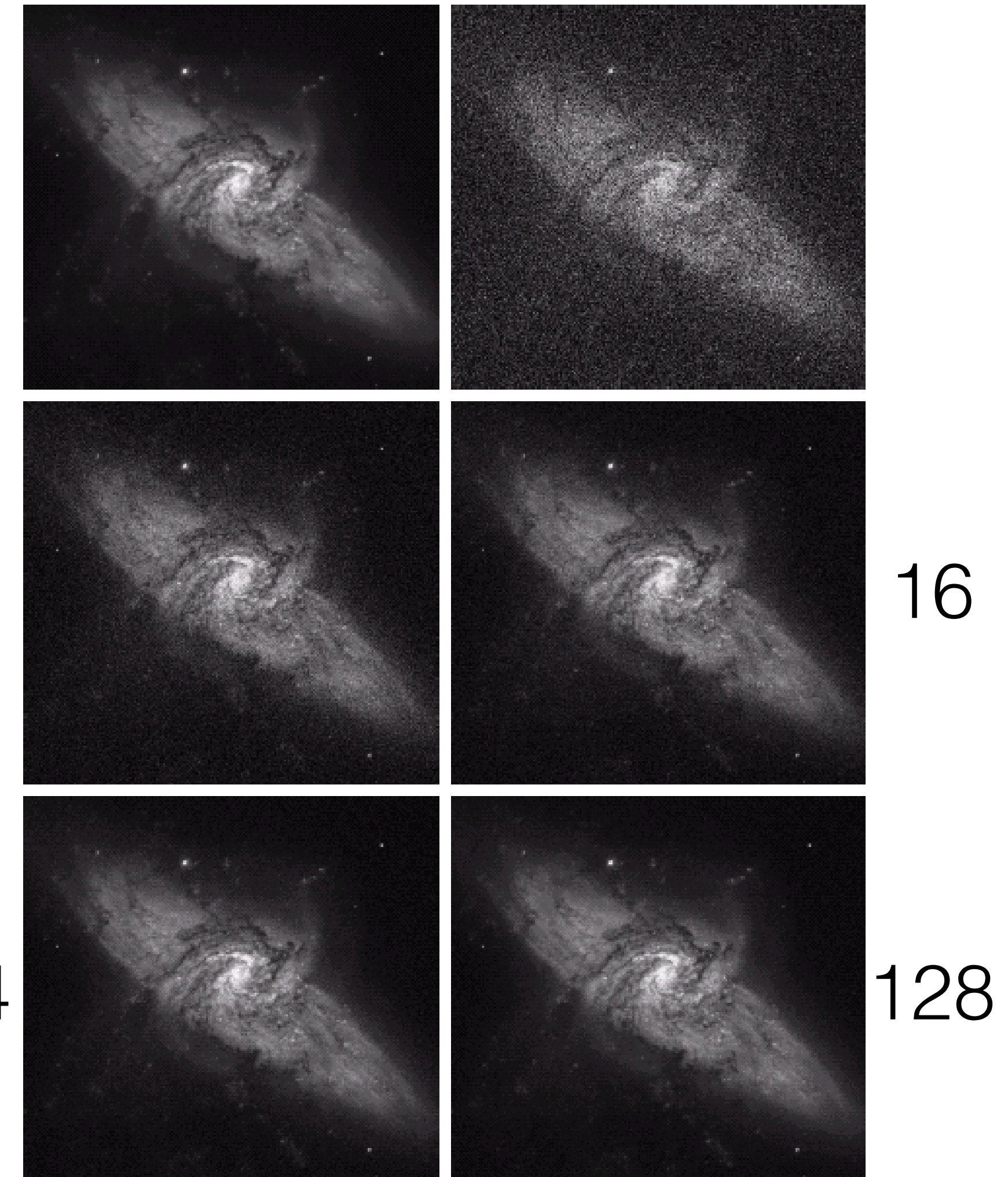


Motion

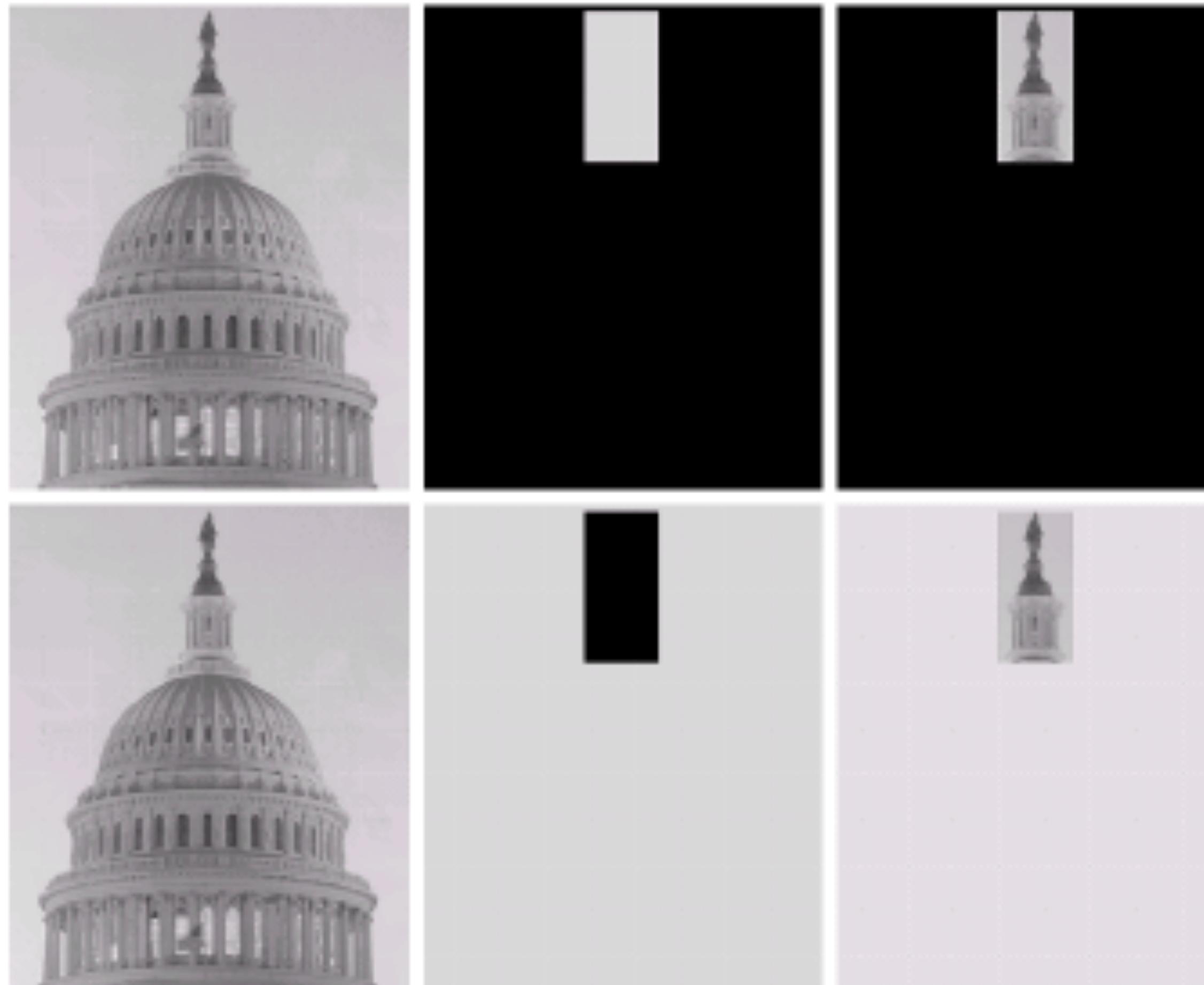
- Use differencing to identify motion in an otherwise unchanging scene (object motion, not camera motion)
 - Basis for motion tracking techniques in computer vision
- Use overall shift (minimum difference) for tracking camera motion
 - Part of a larger process called a “match move” in film making
 - Essential for inserting CGI into a real scene with a moving camera (the virtual camera has to move the same way the physical camera did)
- Useful for video compression
 - Only encode the difference between frames
 - Motion detection/prediction used in video compression (MPEG, etc.)

Image Averaging

- Average multiple pictures of the same static scene to reduce noise
- Similar in principle to acquiring the image for a longer duration



Bitwise AND and OR



Useful for masking

Alpha Blending

- Use *per-pixel weights* to blend two images:

$$\text{out}(x, y) = \alpha_1(x, y) \text{ in}_1(x, y) + \alpha_2(x, y) \text{ in}_2(x, y)$$

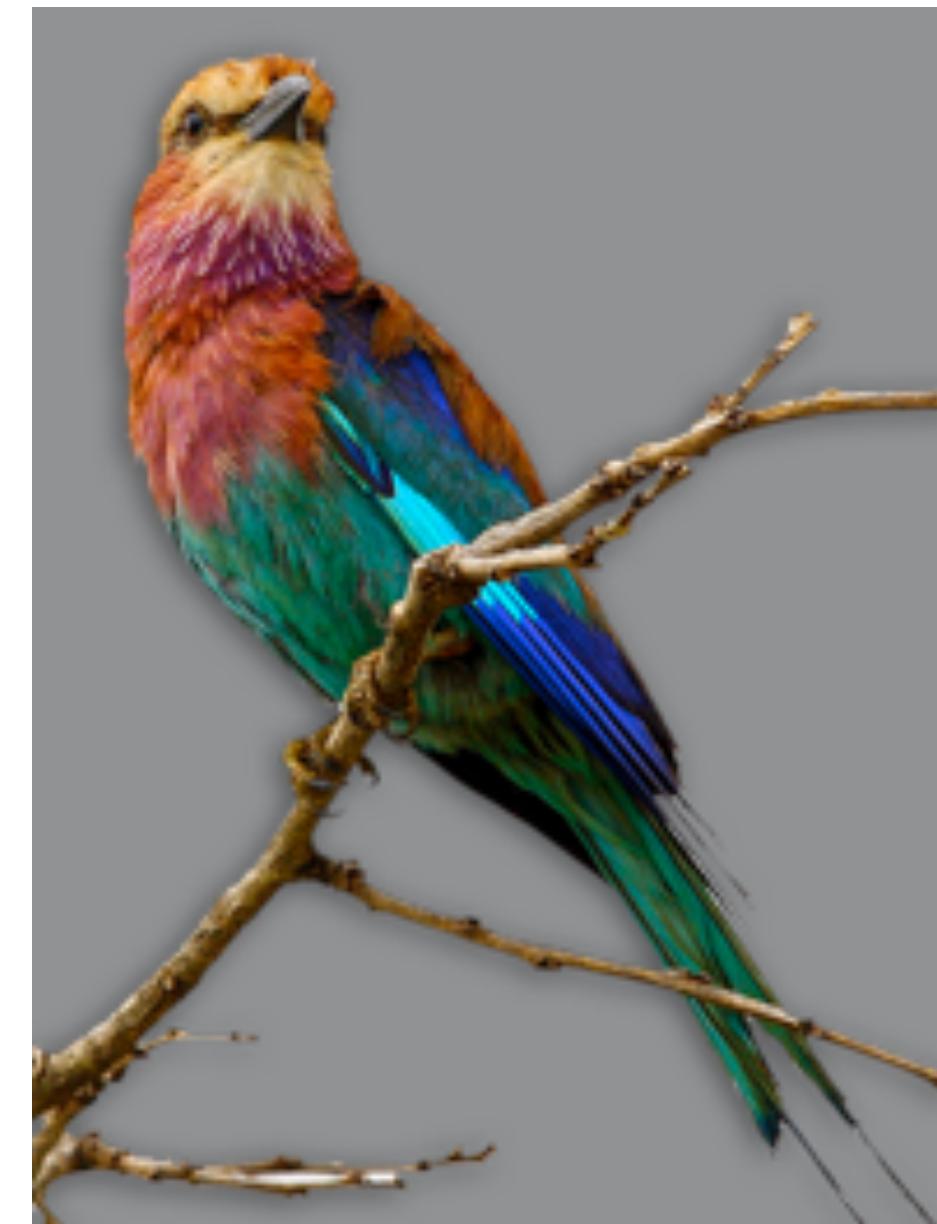
- Or most commonly:

$$\text{out}(x, y) = \alpha(x, y) \text{ in}_1(x, y) + (1 - \alpha(x, y)) \text{ in}_2(x, y)$$

- Useful for transparency, compositing, etc.

Alpha Masks

- Blending often uses an *alpha mask*
- Sometimes also called a *matte*
- Often stored with image as an extra *alpha channel*
- 0 = transparent,
1 = opaque



Source Image



Alpha Mask

$$\text{out}(x, y) = \alpha(x, y) \text{ in}_1(x, y) + (1 - \alpha(x, y)) \text{ in}_2(x, y)$$

Application: Blue Screening



Application: Blue Screening

- Film against blue (or green) background
- Mask out the blue parts
- Use fractional alpha values for partial-pixel effect
- “Decontaminate” the blue (or green) halo
- Store in RGBA format
- Composite onto background using alpha blending

Coming up...

- Neighborhood operations:
 - noise reduction
 - sharpening
 - edge detection



Neighborhood Operations

CS 355: Introduction to Graphics and Image Processing

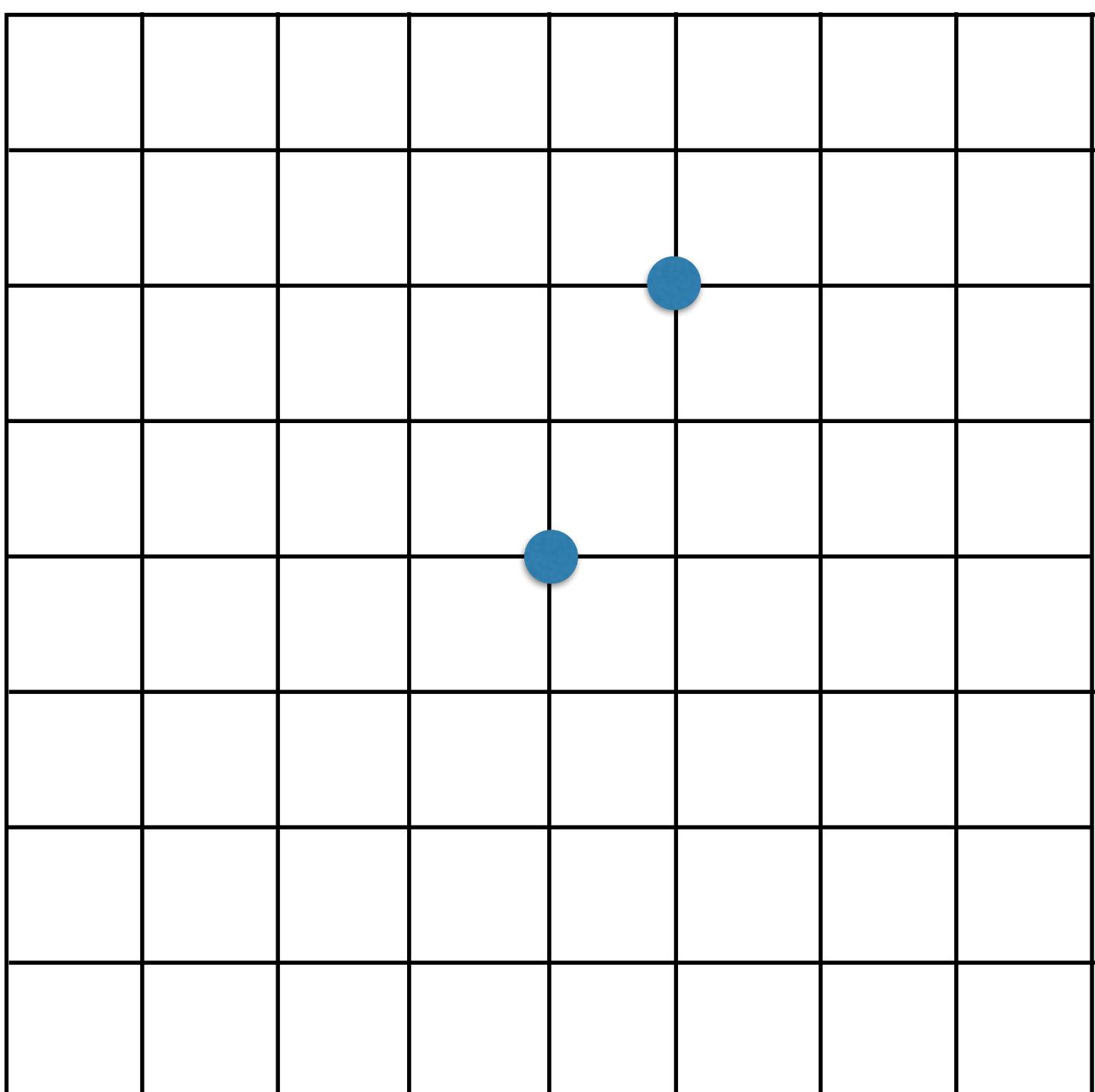
Neighborhood Operations

- Output pixel value is a function of that pixel and its neighbors
- Possible operations: sum, weighted sum, average, weighted average, min, max, median, ...
- Most common workhorse in image processing

$$I'(x, y) = f \begin{pmatrix} I(x - 1, y - 1) & , & I(x, y - 1) & , & I(x + 1, y - 1) & , \\ I(x - 1, y) & , & I(x, y) & , & I(x + 1, y) & , \\ I(x - 1, y + 1) & , & I(x, y + 1) & , & I(x + 1, y + 1) & \end{pmatrix}$$

The Pixel Grid

- Many of the things we do involve using “neighboring” pixels
- Common approaches:
 - 4-connected (N, S, E, W)
 - 8-connected (add NE, SE, SW, NW)
- Distance?
 - Euclidean (as the crow flies)
 - 4-connected (“city block”, “Manhattan”)
 - 8-connected (“chessboard”)



Spatial Filtering

- Most common is to multiply each of the pixels in the neighborhood by a respective weight and add them together
- The local weights are called a *mask* or *kernel*

$I(x-1, y-1)$	$I(x, y-1)$	$I(x+1, y-1)$
$I(x-1, y)$	$I(x, y)$	$I(x+1, y)$
$I(x-1, y+1)$	$I(x, y+1)$	$I(x+1, y+1)$

$w(-1, -1)$	$w(0, -1)$	$w(1, -1)$
$w(-1, 0)$	$w(0, 0)$	$w(1, 0)$
$w(-1, 1)$	$w(0, 1)$	$w(1, 1)$

Spatial Filtering

$I(x-1, y-1)$	$I(x, y-1)$	$I(x+1, y-1)$
$I(x-1, y)$	$I(x, y)$	$I(x+1, y)$
$I(x-1, y+1)$	$I(x, y+1)$	$I(x+1, y+1)$

$w(-1, -1)$	$w(0, -1)$	$w(1, -1)$
$w(-1, 0)$	$w(0, 0)$	$w(1, 0)$
$w(-1, 1)$	$w(0, 1)$	$w(1, 1)$

$$I'(x, y) = \sum_{s=-1}^1 \sum_{t=-1}^1 w(s, t) I(x + s, y + t)$$

Convolution

$I(x-1, y-1)$	$I(x, y-1)$	$I(x+1, y-1)$
$I(x-1, y)$	$I(x, y)$	$I(x+1, y)$
$I(x-1, y+1)$	$I(x, y+1)$	$I(x+1, y+1)$

$w(1, 1)$	$w(0, 1)$	$w(-1, 1)$
$w(1, 0)$	$w(0, 0)$	$w(-1, 0)$
$w(1, -1)$	$w(0, -1)$	$w(-1, -1)$

$$I'(x, y) = \sum_{s=-1}^1 \sum_{t=-1}^1 w(s, t) I(x - s, y - t)$$

Convolution is the same thing with the mask flipped

Correlation vs. Convolution

- Technically, spatial filtering is *correlation*, different from *convolution*
- They are the same up to flipping the mask/kernel
- Many casually use them interchangeably (be careful with the details)

$w(-1,-1)$	$w(0,-1)$	$w(1,-1)$
$w(-1,0)$	$w(0,0)$	$w(1,0)$
$w(-1,1)$	$w(0,1)$	$w(1,1)$

$w(1,1)$	$w(0,1)$	$w(-1,1)$
$w(1,0)$	$w(0,0)$	$w(-1,0)$
$w(1,-1)$	$w(0,-1)$	$w(-1,-1)$

Spatial Filtering

45	60	98	127	132	133	137	133
46	65	98	123	126	128	131	133
47	65	96	115	119	123	135	137
47	63	91	107	113	122	138	134
50	59	80	97	110	123	133	134
49	53	68	83	97	113	128	133
50	50	58	70	84	102	116	126
50	50	52	58	69	86	101	120

*

0.1	0.1	0.1
0.1	0.2	0.1
0.1	0.1	0.1

=

69	95	116	125	129	132
68	92	110	120	126	132
66	86	104	114	124	132
62	78	94	108	120	129
57	69	83	98	112	124
53	60	71	85	100	114

notation for convolution operator

$$I' = I * w$$

Spatial Filtering

- What do you do outside the image boundaries?
 - Assume zero (tends to darken borders if blurring)
 - Assume other constant value (perhaps average of entire image)
 - Wrap around
 - Assume same as closest pixel still in image
 - Or just don't go there

Spatial Filtering

- Applications:
 - Blurring
 - Sharpening
 - Edge detection
 - and many more...



Smoothing

- If we can average multiple images together to remove noise, why not average multiple pixels?
- What does this assume?
- Effects:
 - Reduces noise
 - Causes blurring



Smoothing

- Any kernel with all positive weights does smoothing / blurring
- To average rather than add, divide by the sum of the weights
- Can be any size (larger means more blurring)

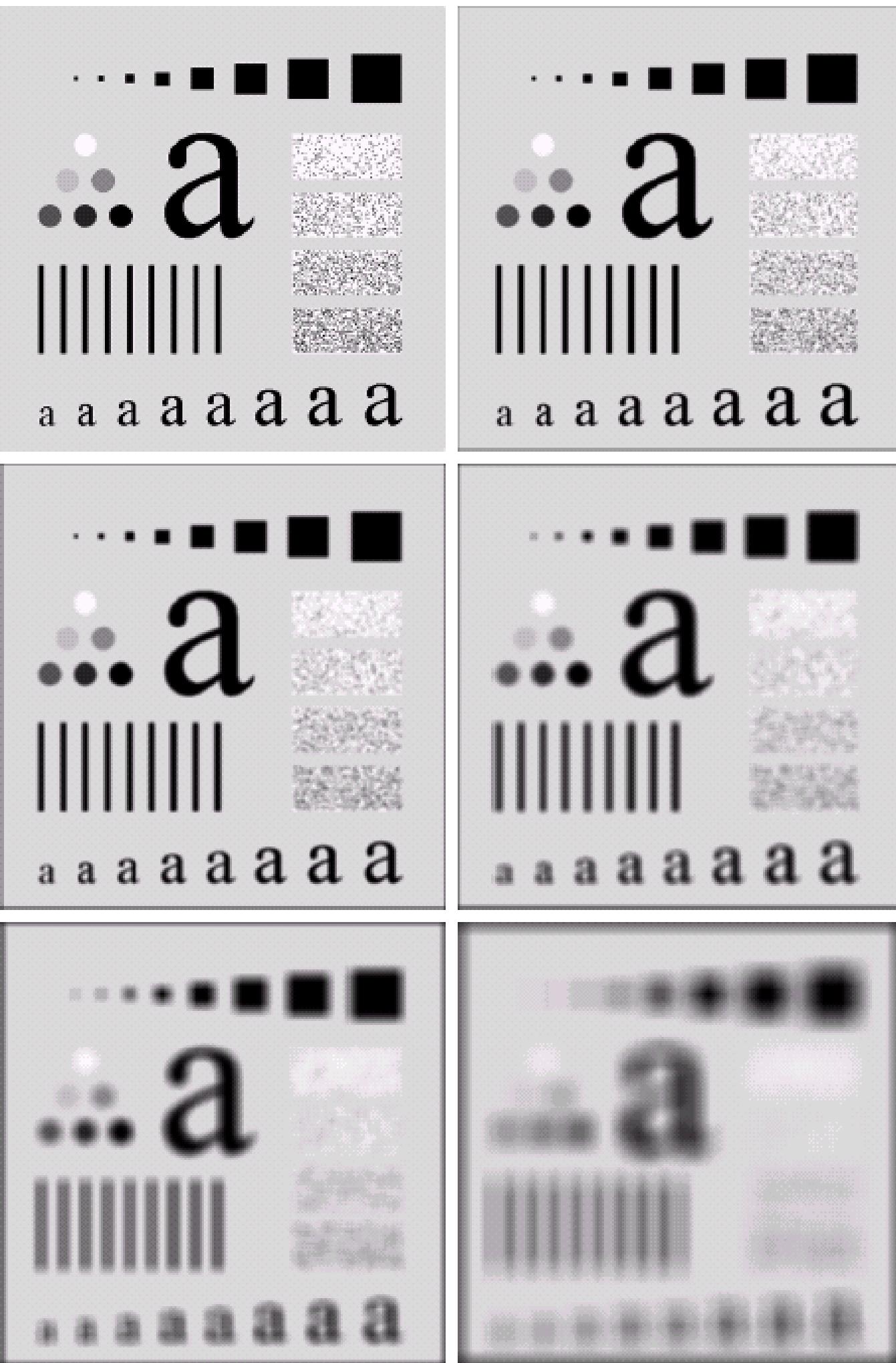
1	1	1
1	1	1
1	1	1

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

$$I'(x, y) = \frac{\sum_s \sum_t w(s, t) I(x + s, y + t)}{\sum_s \sum_t w(s, t)}$$

Smoothing



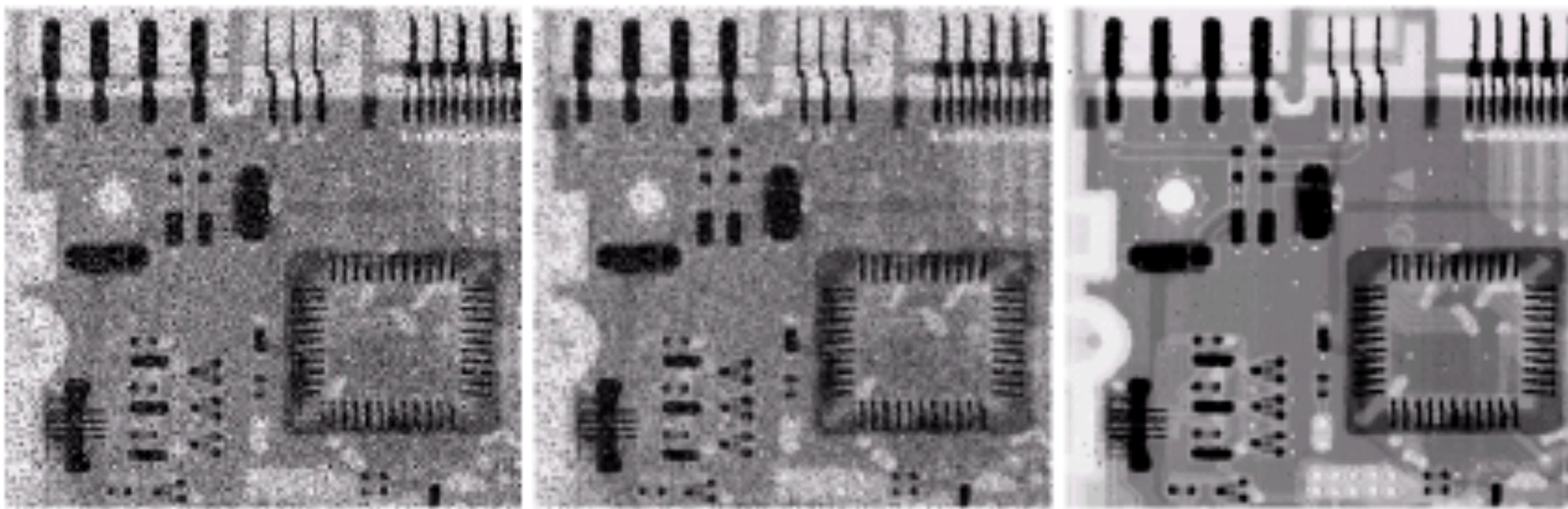
Nonlinear Smoothing

- Spatial filtering is linear, but many neighborhood operators are not
- Some do noise reduction:
 - Trimmed mean
 - Median filter
 - Bilateral filtering (or other adaptive weights)
- These try to be less sensitive to outliers and/or respect edges

Median Filtering

- Output is the median (not the mean) of the neighborhood pixels
 - More robust to outliers
(great for “salt and pepper” noise)
 - Tries to respect edges
(goes with local majority)
 - But often rounds corners or
loses very small/thin things

Median Filtering



Original

Mean

Median

Bilateral Filtering

- Spatially adapt the weights of the mask
 - Closer neighbors get more weight
 - Similar neighbors get more weight
- Many similar approaches use this idea, but this is most popular now
- Computationally expensive, but there are efficient approximations

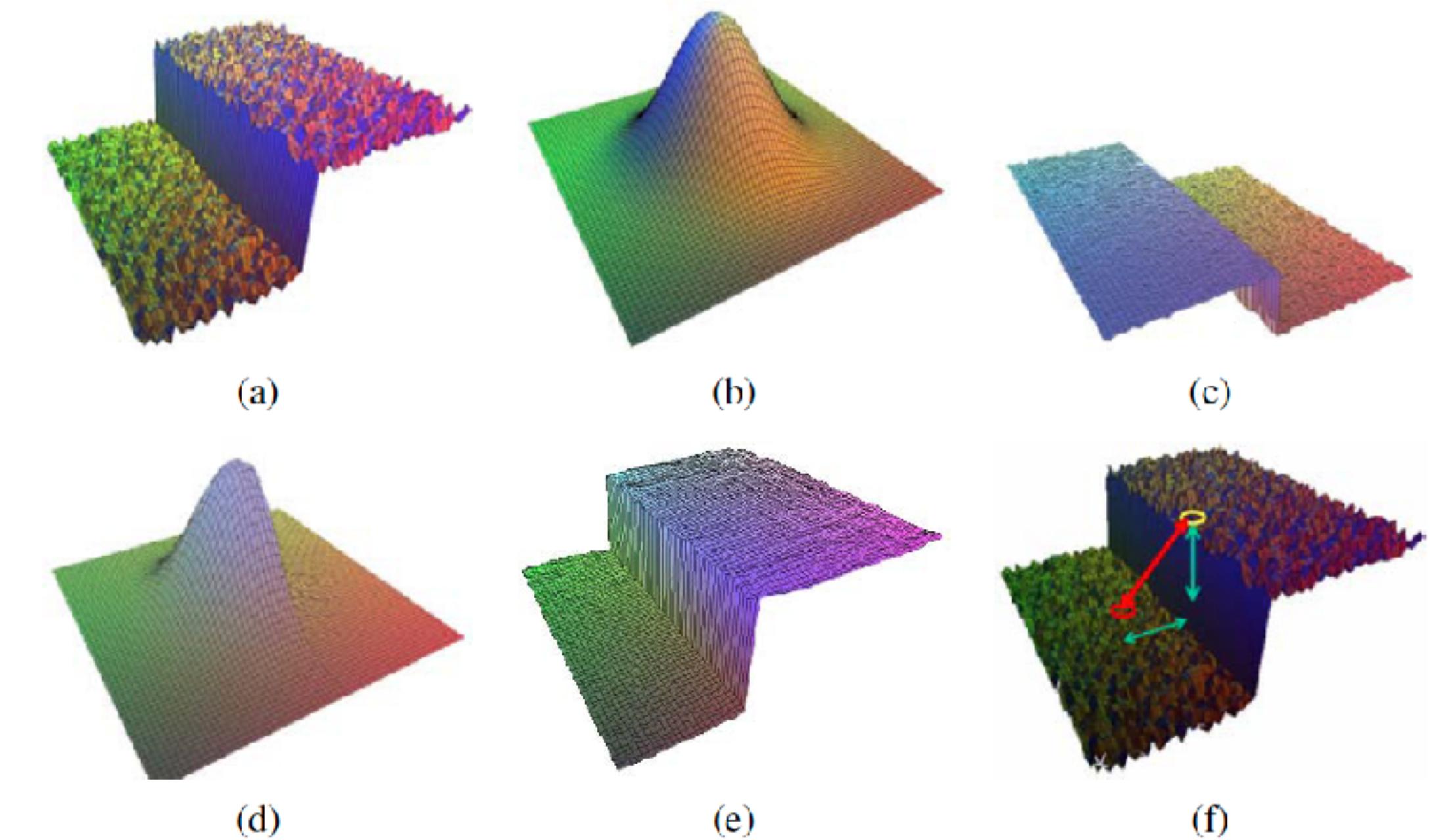
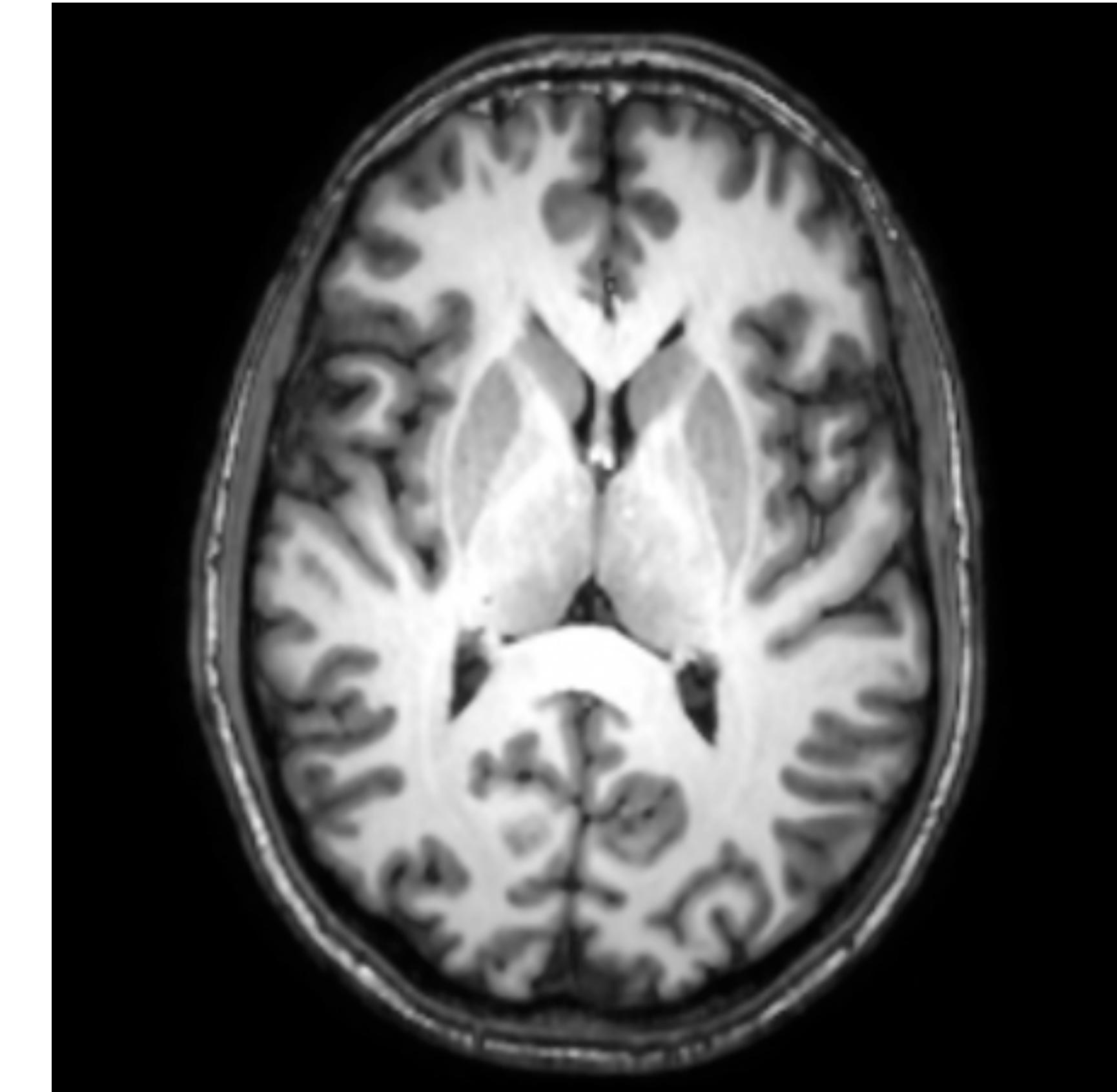
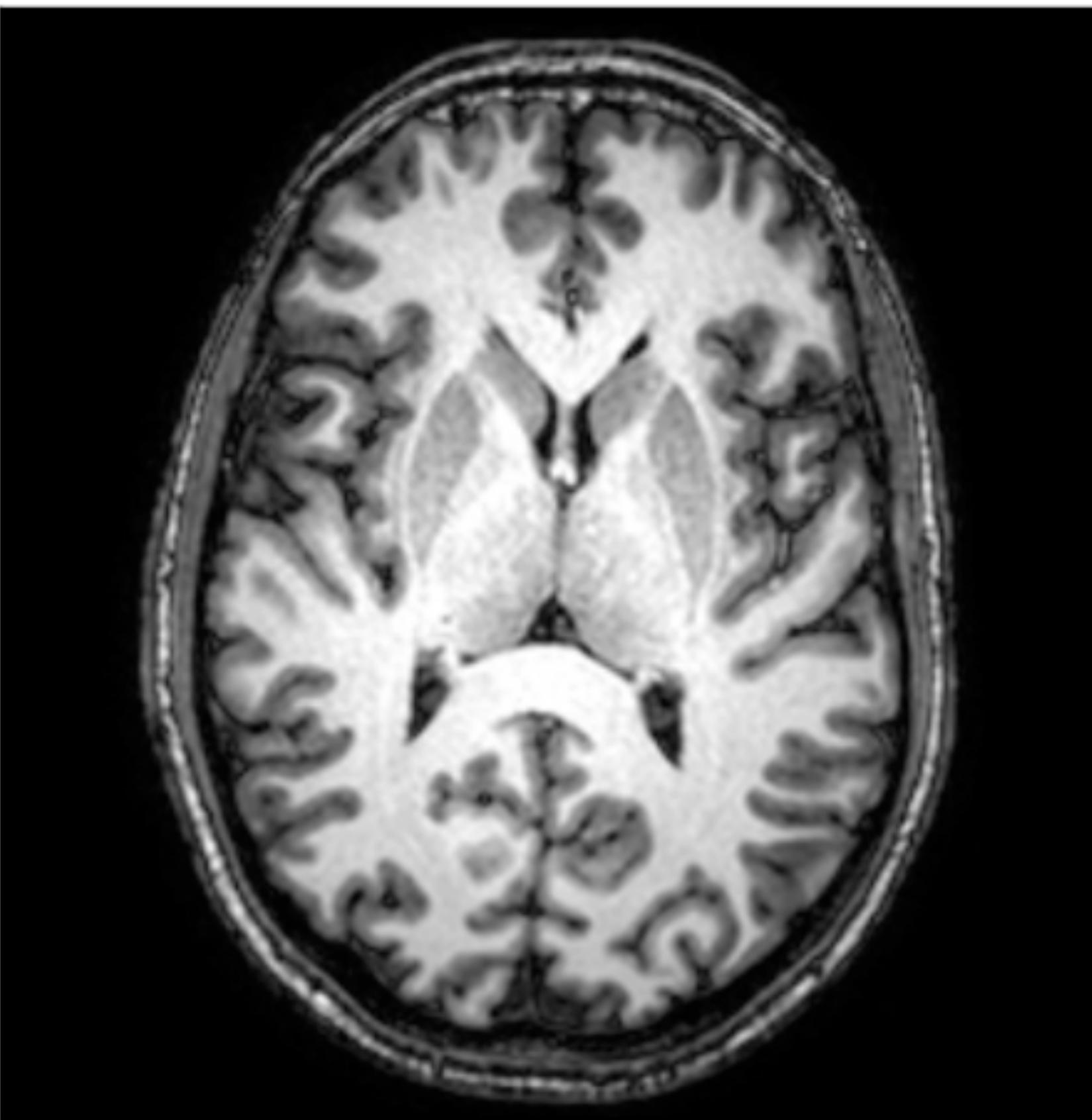


Figure 3.20 Bilateral filtering (Durand and Dorsey 2002) © 2002 ACM: (a) noisy step edge input; (b) domain filter (Gaussian); (c) range filter (similarity to center pixel value); (d) bilateral filter; (e) filtered step edge output; (f) 3D distance between pixels.

Anisotropic Diffusion



Iteratively diffuse (blur) based on neighbor similarity

Coming up...

- More neighborhood operations:
 - sharpening
 - edge detection



Neighborhood Operations (cont'd)

CS 355: Introduction to Graphics and Image Processing

Spatial Filtering

45	60	98	127	132	133	137	133
46	65	98	123	126	128	131	133
47	65	96	115	119	123	135	137
47	63	91	107	113	122	138	134
50	59	80	97	110	123	133	134
49	53	68	83	97	113	128	133
50	50	58	70	84	102	116	126
50	50	52	58	69	86	101	120

*

0.1	0.1	0.1
0.1	0.2	0.1
0.1	0.1	0.1

=

69	95	116	125	129	132
68	92	110	120	126	132
66	86	104	114	124	132
62	78	94	108	120	129
57	69	83	98	112	124
53	60	71	85	100	114

notation for convolution operator

$$I' = I * w$$

Negative Weights

- Detecting edges or sharpening images involve finding or accentuating differences
- Requires mix of *positive and negative weights*

-1	0	1
-1	0	1
-1	0	1

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

1	-2	1
1	-2	1
1	-2	1

0	-1	0
-1	5	-1
0	-1	0

Unsharp Masking

- Originated in analog darkrooms
- Key idea: mask (subtract) out the blur
- Procedure:
 - Blur more (yes, really!)
 - Subtract from original
 - Multiply by some fraction
 - Add back to the original

Unsharp Masking

- Mathematically:

$$I' = I + \alpha(I - \bar{I})$$

- Input image

$$I$$

- Blurred input image

$$\bar{I}$$

- Weighting (controls sharpening)

$$\alpha$$

- Output image

$$I'$$

Unsharp Masking

Let $\alpha = \frac{5}{A}$, then

$$I' = I + \alpha(I - \bar{I}) = \frac{1}{A} (AI + 5(I - \bar{I}))$$

$$\begin{aligned} & \frac{1}{A} \left[\begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & A & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \right] + \left(\begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 5 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array} - \begin{array}{|c|c|c|} \hline 0 & 1 & 0 \\ \hline 1 & 1 & 1 \\ \hline 0 & 1 & 0 \\ \hline \end{array} \right) \\ &= \frac{1}{A} \begin{array}{|c|c|c|} \hline 0 & -1 & 0 \\ \hline -1 & A+4 & -1 \\ \hline 0 & -1 & 0 \\ \hline \end{array} \end{aligned}$$

Unsharp Masking

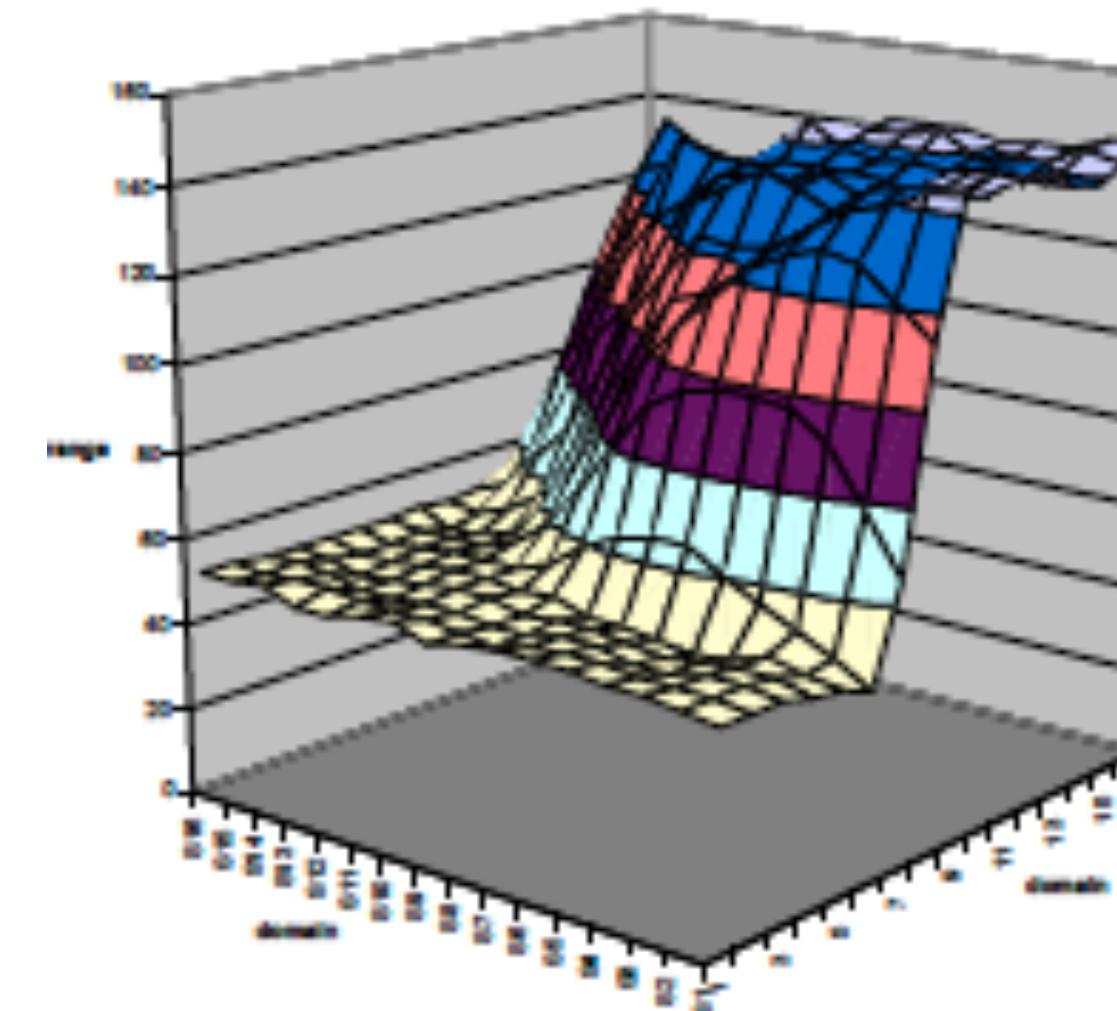
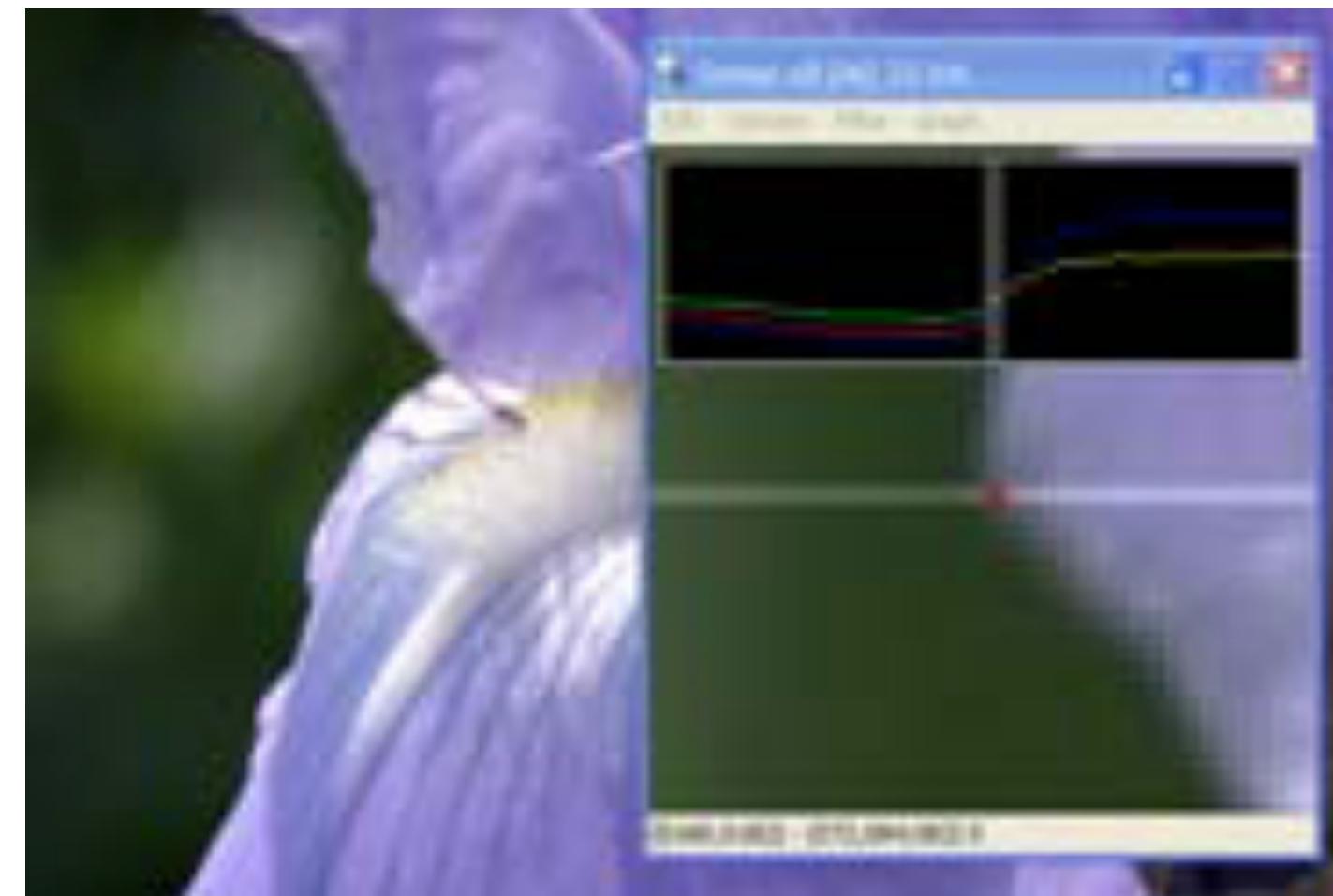


Tradeoff

- Blurring:
 - Reduces noise
 - Causes blur
- Sharpening:
 - Reduces blur
 - Strengthens noise

Edge Detection

- Edges between objects in images are often places where there are strong changes
- Find these using (approximations to) image derivatives



Approximating Derivatives

- Can approximate derivatives with finite differences

$$\frac{d}{dt} f(t) = \lim_{h \rightarrow 0} \frac{f(t + h) - f(t)}{h}$$

- Can choose

- Forward (right)

$$\frac{d}{dt} f(t) \approx \frac{f(t + 1) - f(t)}{1}$$

- Backward (left)

$$\frac{d}{dt} f(t) \approx \frac{f(t) - f(t - 1)}{1}$$

- Central

$$\frac{d}{dt} f(t) \approx \frac{f(t + 1) - f(t - 1)}{2}$$

Approximating Derivatives

- Simplest:
Just take central differences
horizontally and vertically
- Approximates partial derivatives

0	0	0
-1	0	1
0	0	0

0	-1	0
0	0	0
0	1	0

$$\frac{\partial}{\partial x}$$

$$\frac{\partial}{\partial y}$$

(Divide by 2, the separation between the pixels you're taking the difference between)

Prewitt Kernels

- Better still:
Average in other direction
- More robust since you're reducing
noise in one direction while taking
derivative in the other

-1	0	1
-1	0	1
-1	0	1

$$\frac{\partial}{\partial x}$$

-1	-1	-1
0	0	0
1	1	1

$$\frac{\partial}{\partial y}$$

(Divide by 3, the sum of the averaging weights)

(Divide by 2, the separation between the pixels you're taking the difference between)

Sobel Kernels

- More common:
Use a center-weighted average
- More robust to noise

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

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

$$\frac{\partial}{\partial x}$$

$$\frac{\partial}{\partial y}$$

(Divide by 4, the sum of the averaging weights)

(Divide by 2, the separation between the pixels you're taking the difference between)

2nd Derivatives

- Can also do second derivatives
- Differences of differences
- Can still combined with smoothing in other direction

0	0	0
1	-2	1
0	0	0

0	1	0
0	-2	0
0	1	0

1	-2	1
1	-2	1
1	-2	1

1	1	1
-2	-2	-2
1	1	1

Gradients

- The *gradient* is a vector of partial derivatives with respect to each of a function's inputs
- The direction of the gradient is the *direction of greatest increase*
- The magnitude of the gradient is the *amount of increase in that direction*

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \vdots \end{bmatrix}$$

$$\|\nabla f\| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2 + \dots}$$

Gradient Magnitude

- The magnitude of the local gradient is the most common form of edge detector

$$\nabla I = \begin{bmatrix} \frac{\partial I}{\partial x} \\ \frac{\partial I}{\partial y} \end{bmatrix}$$

$$\|\nabla I\| = \sqrt{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2}$$



Lab #3

- Level operations with color:
 - Convert to grey (1)
 - Adjust brightness (2)
 - Adjust contrast (3)
- Image blending (4)
- Alpha blending (5)
- Uniform averaging (6)
- Median filtering (7)
- General convolution (8)
- Sharpening (9)
- Gradient magnitude (10)

Coming up...

- Points, vectors, matrices and other concepts from linear algebra
- Coordinate systems
- Transformations



Points and Vectors

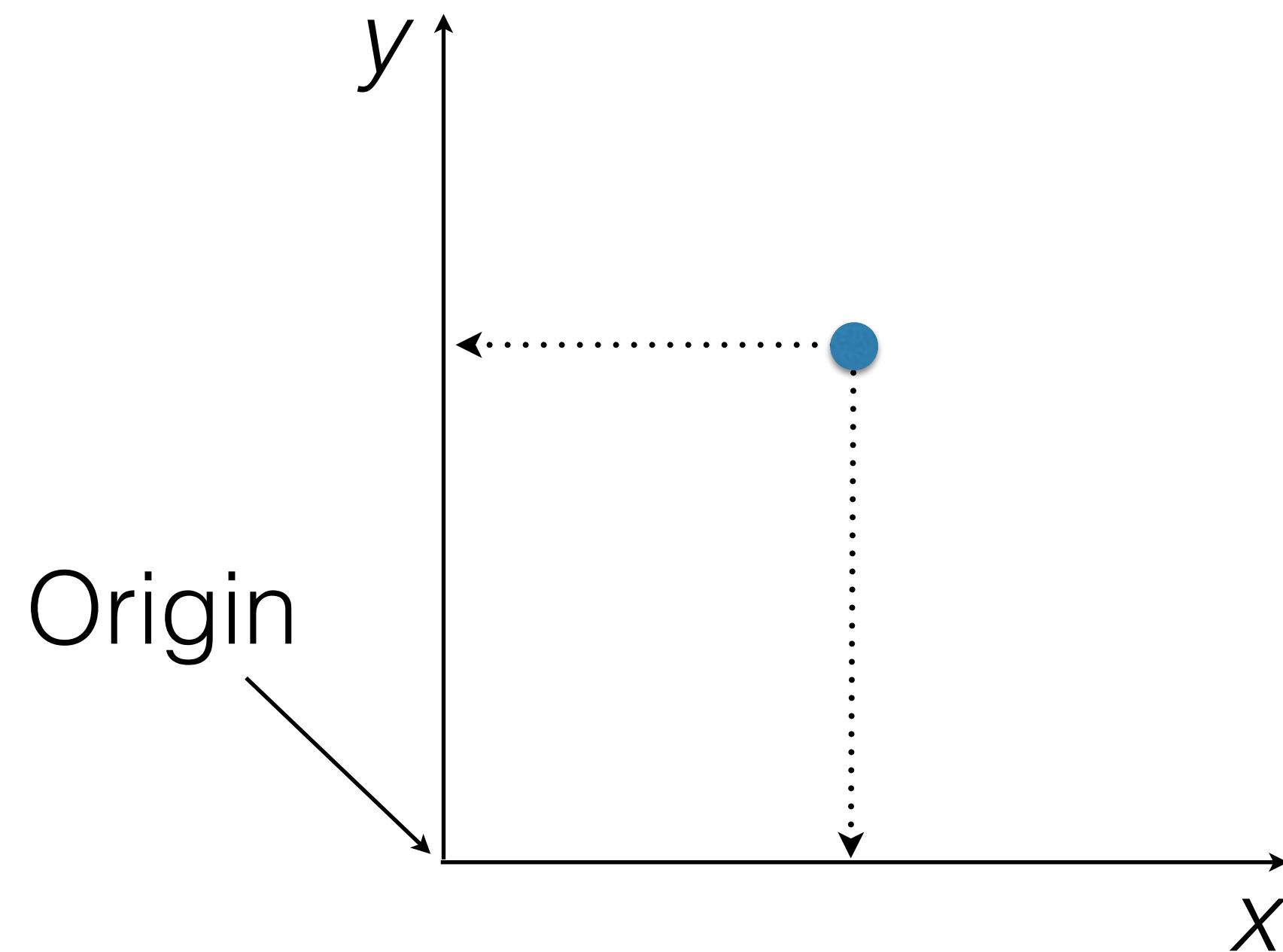
CS 355: Introduction to Graphics and Image Processing

Describing Points



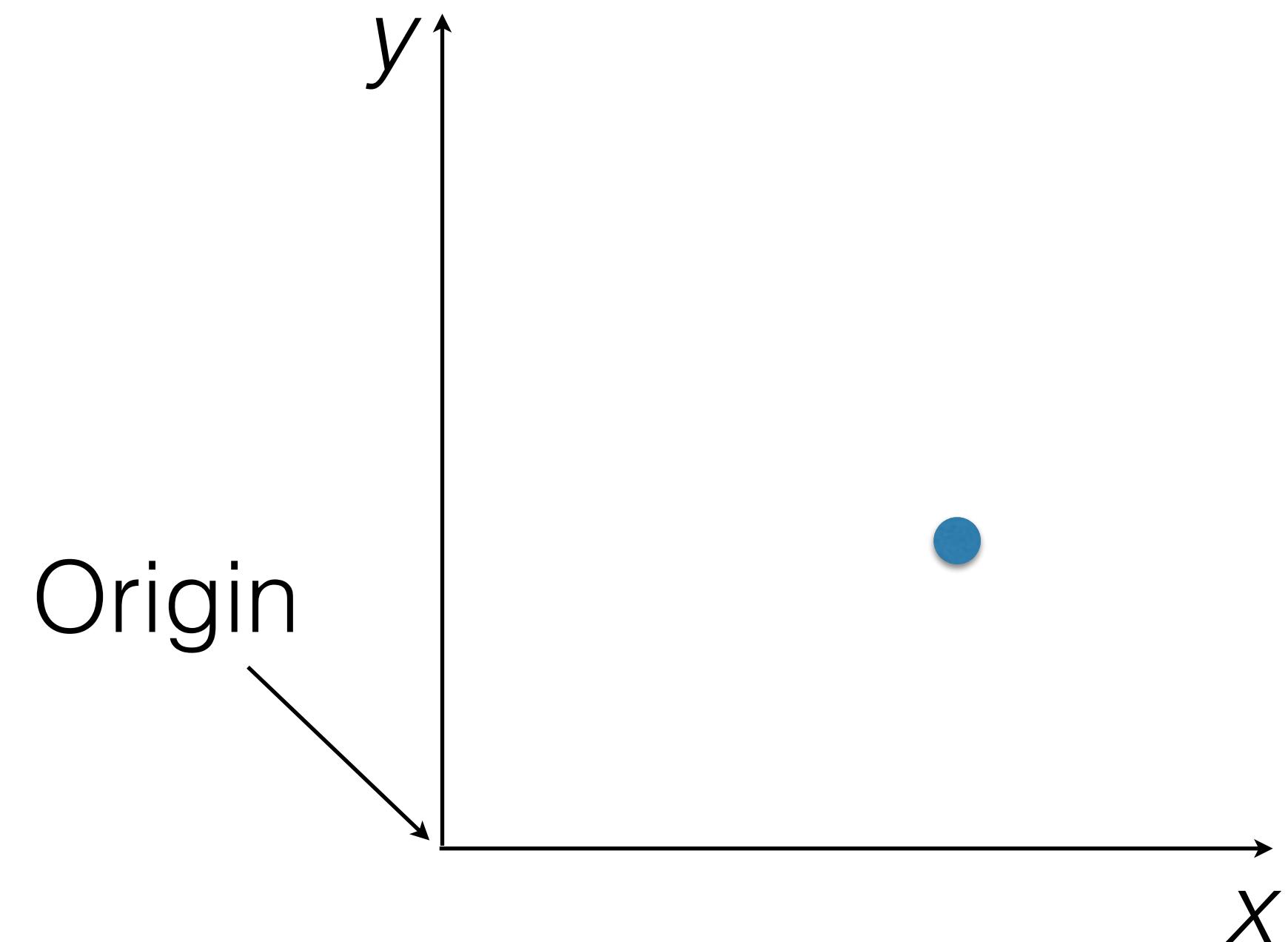
How do you describe this point numerically?

Coordinate Systems



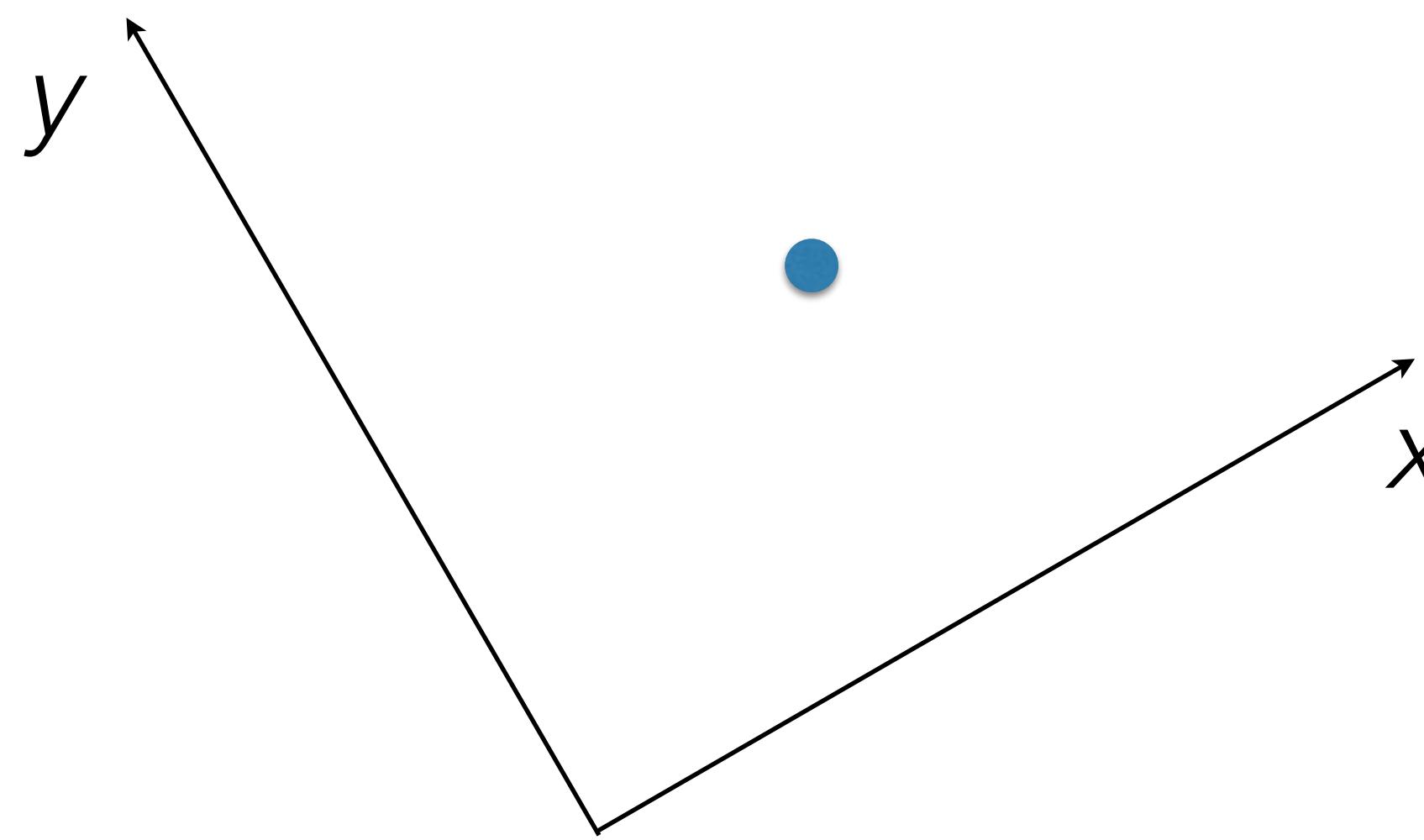
How do you describe this point numerically?

Coordinate Systems



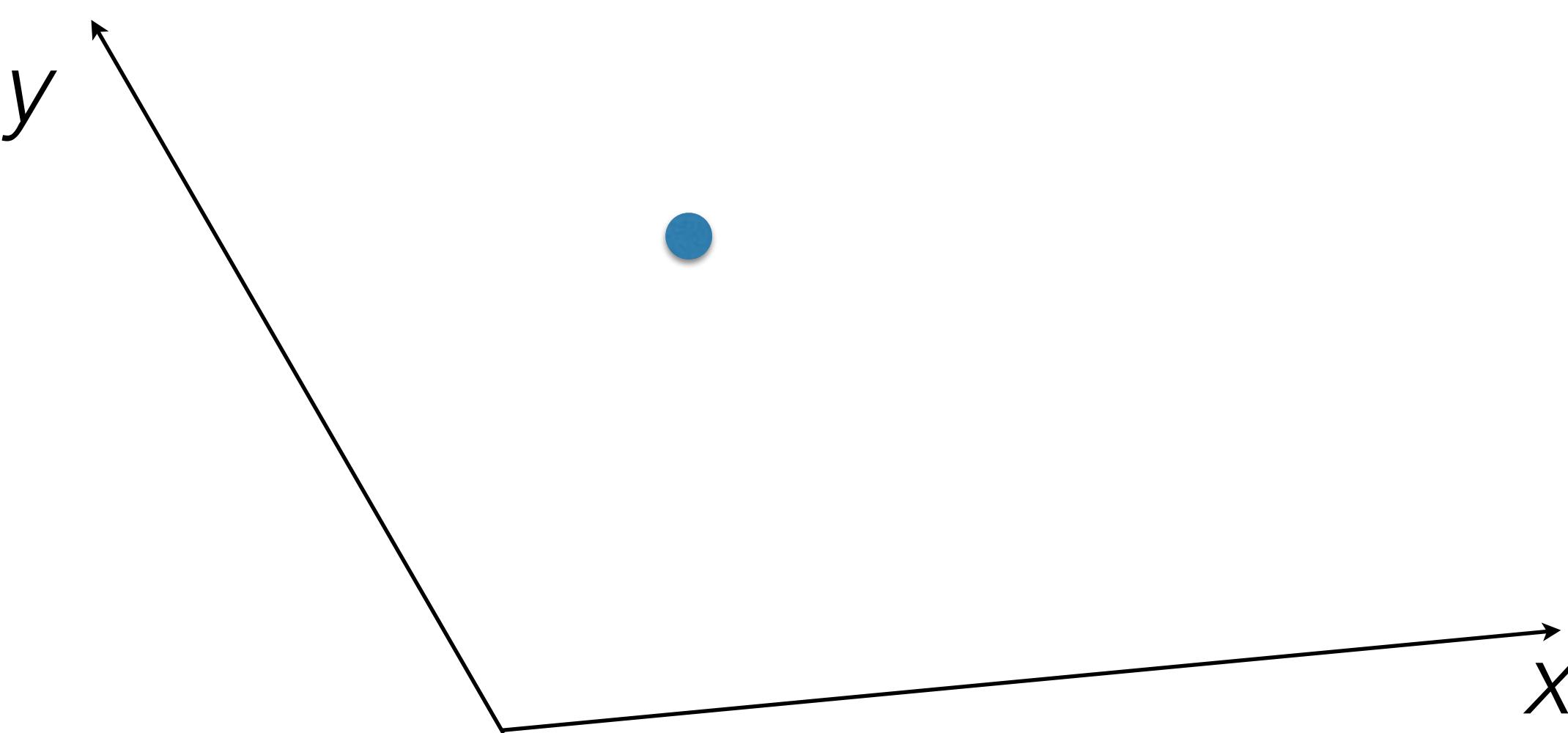
How about this coordinate system?

Coordinate Systems



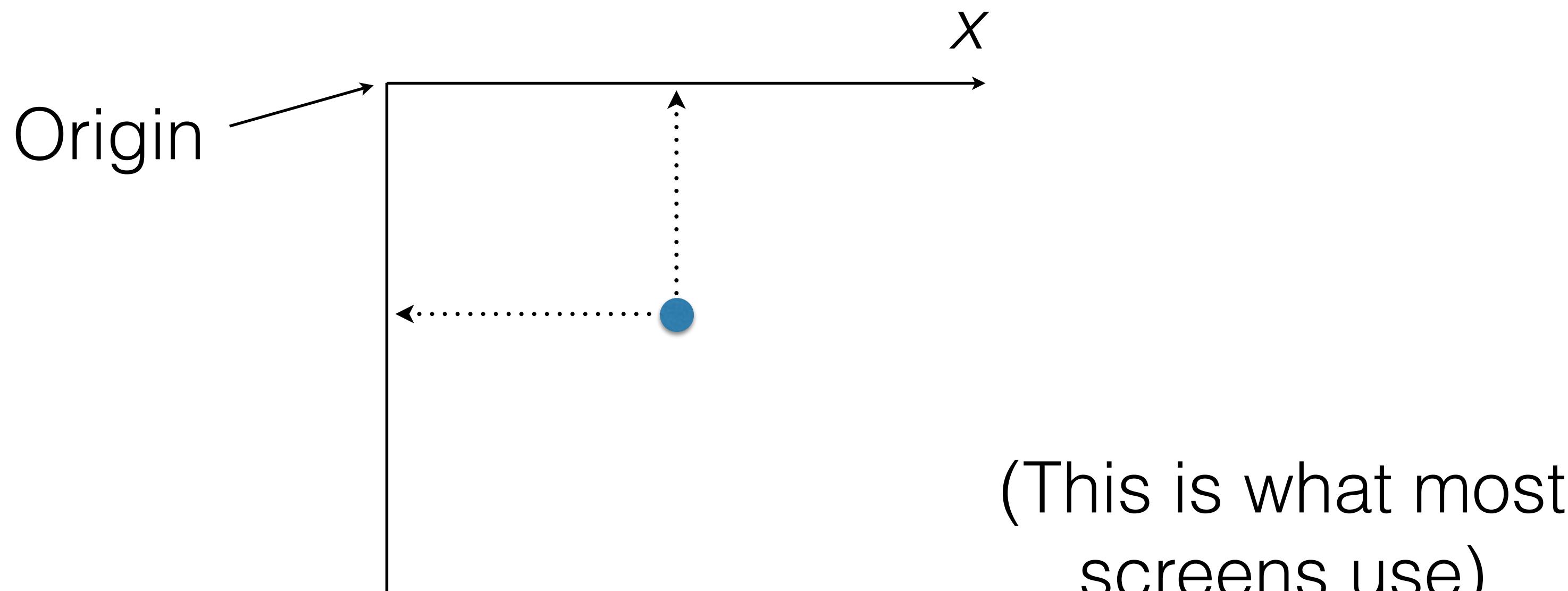
Or this one?

Coordinate Systems



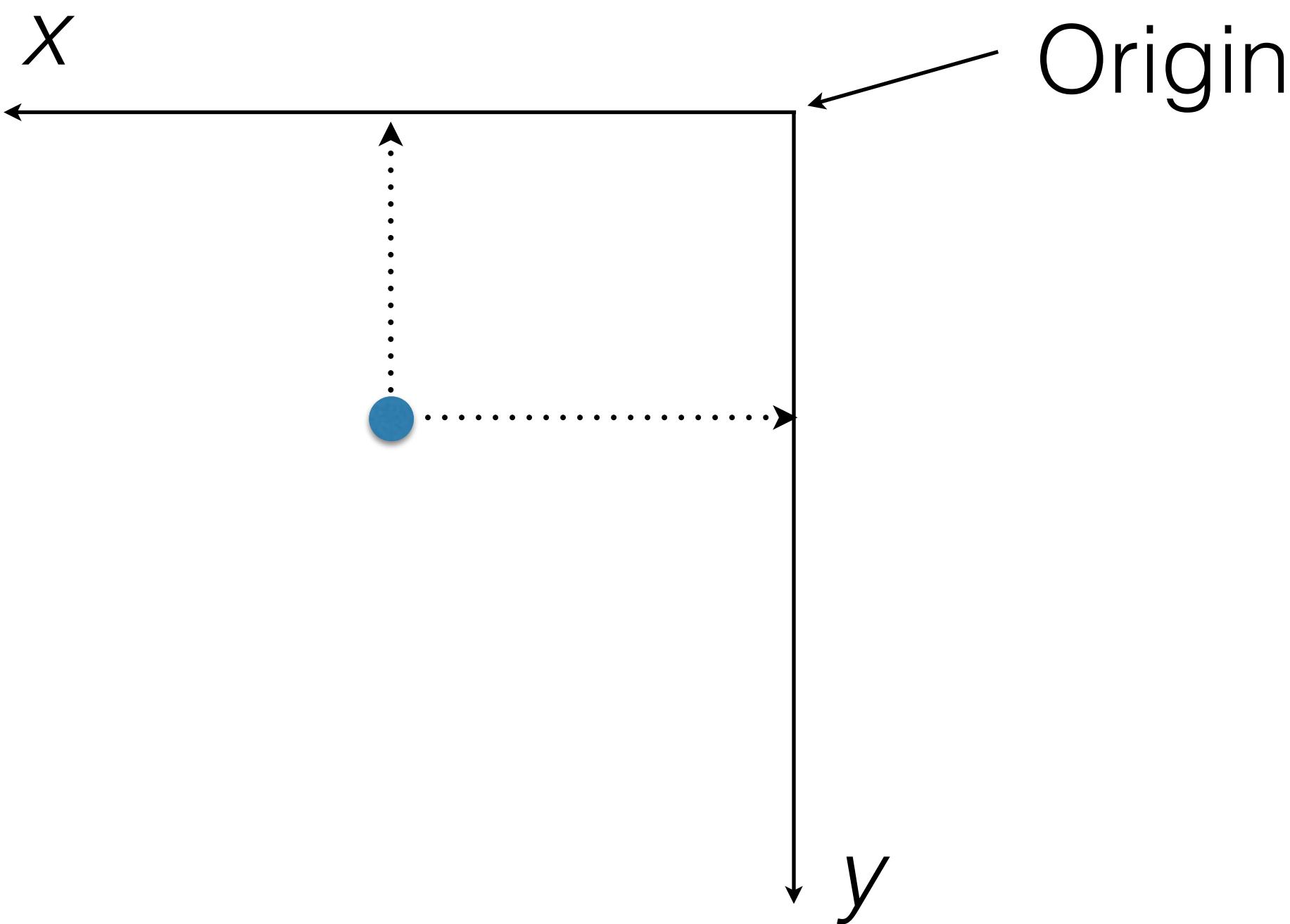
What about this one?

Coordinate Systems



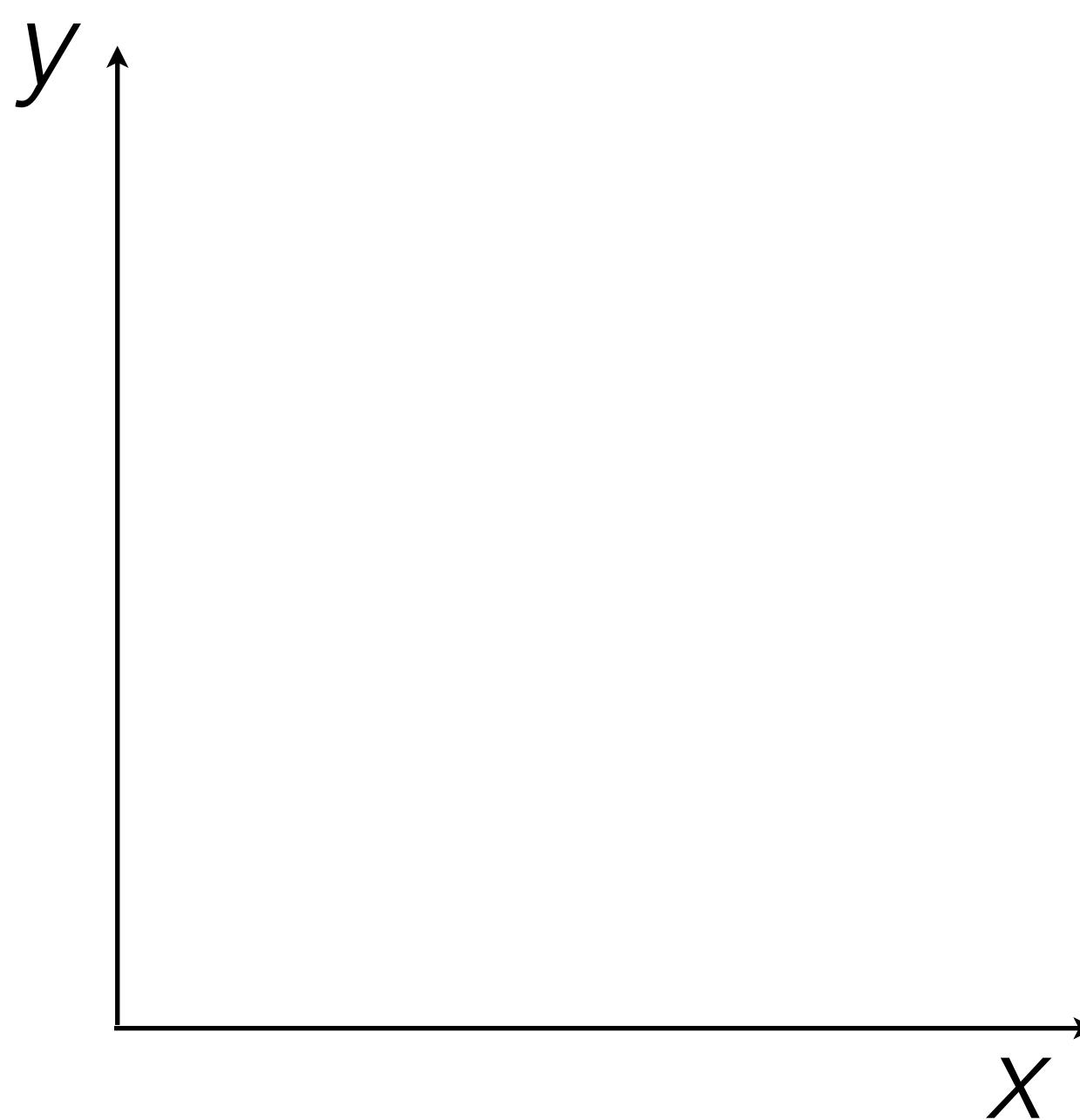
Why not this?

Coordinate Systems

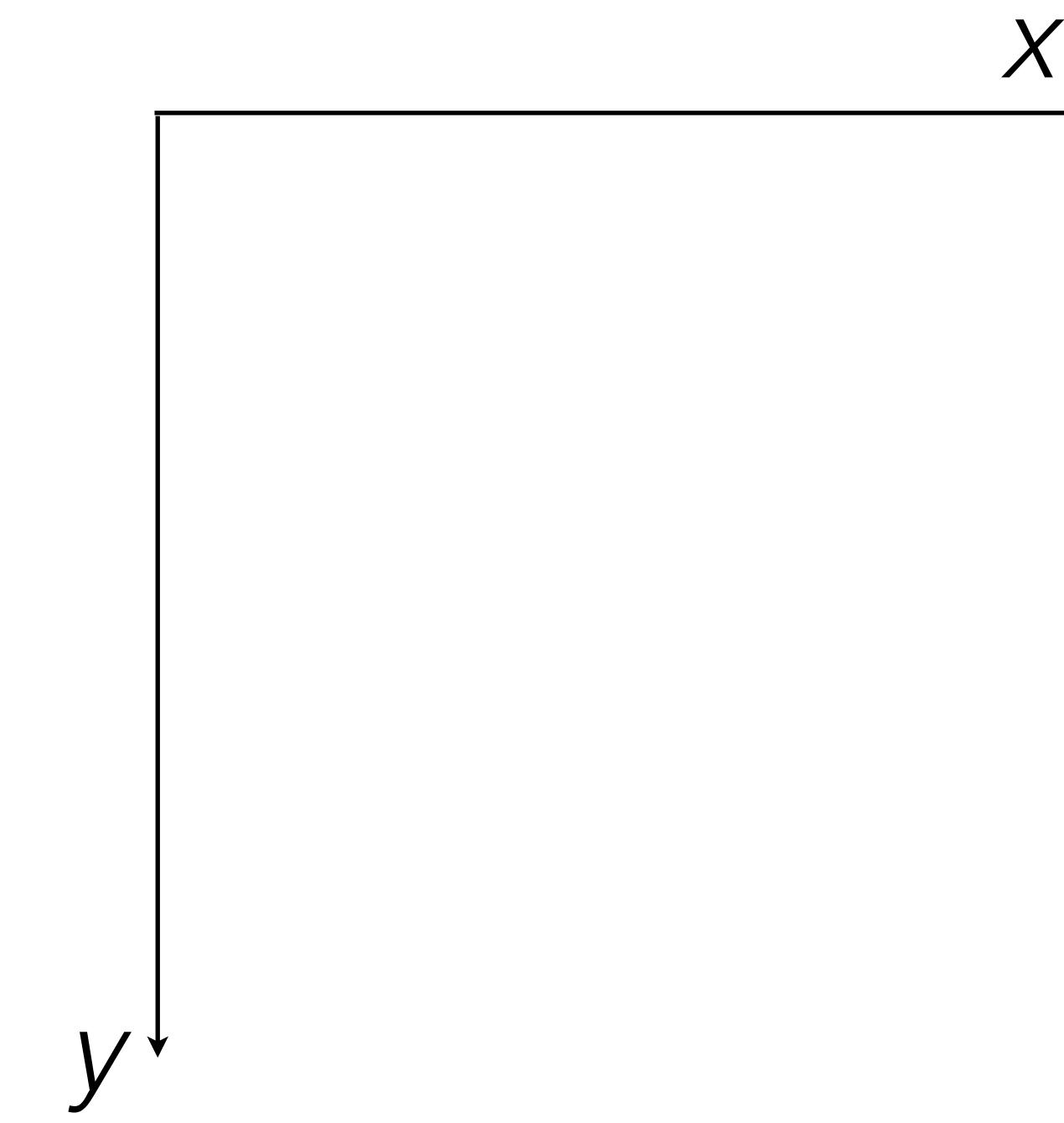


Or this?

Coordinate Systems

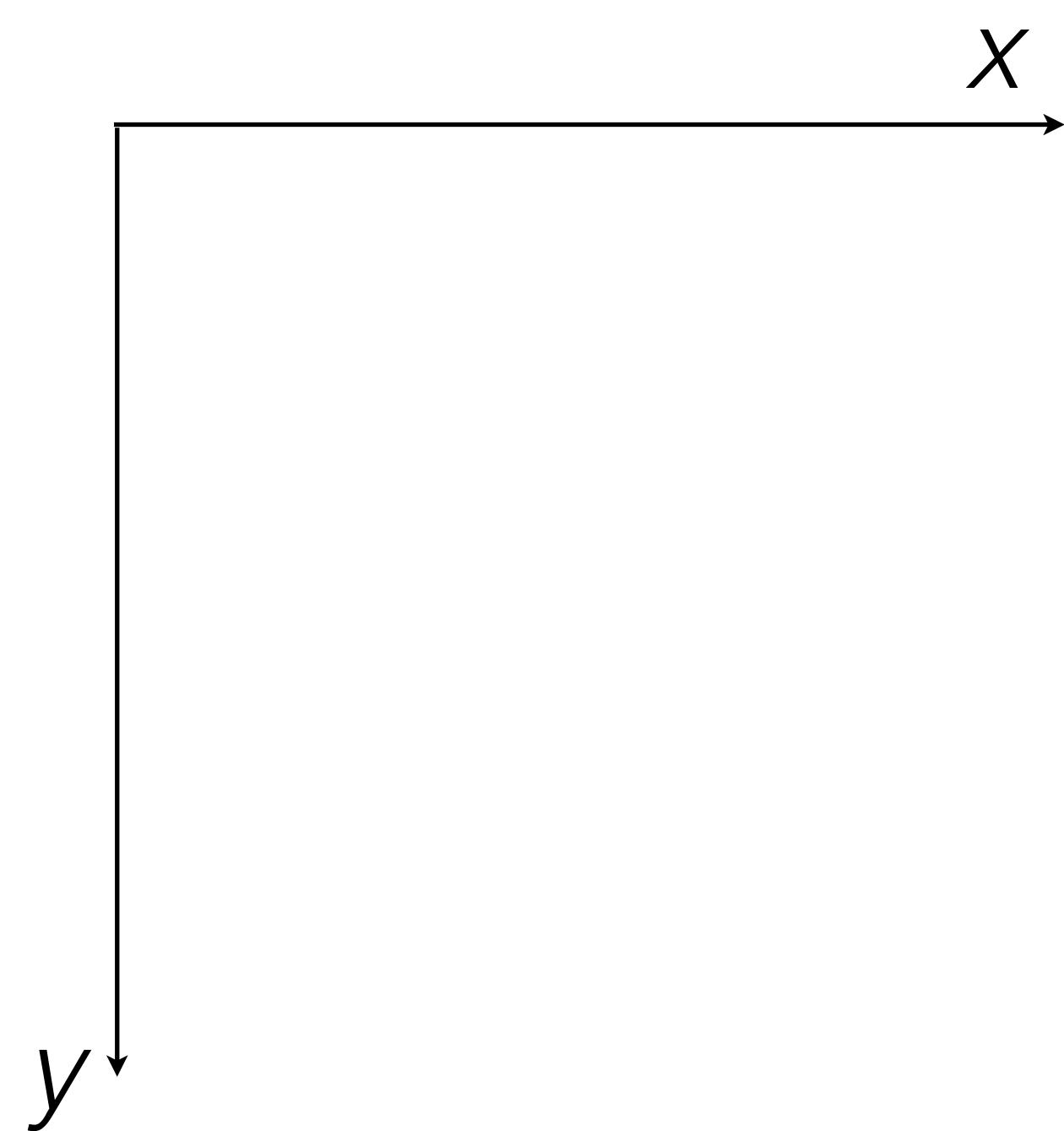


Math teachers
("right handed")

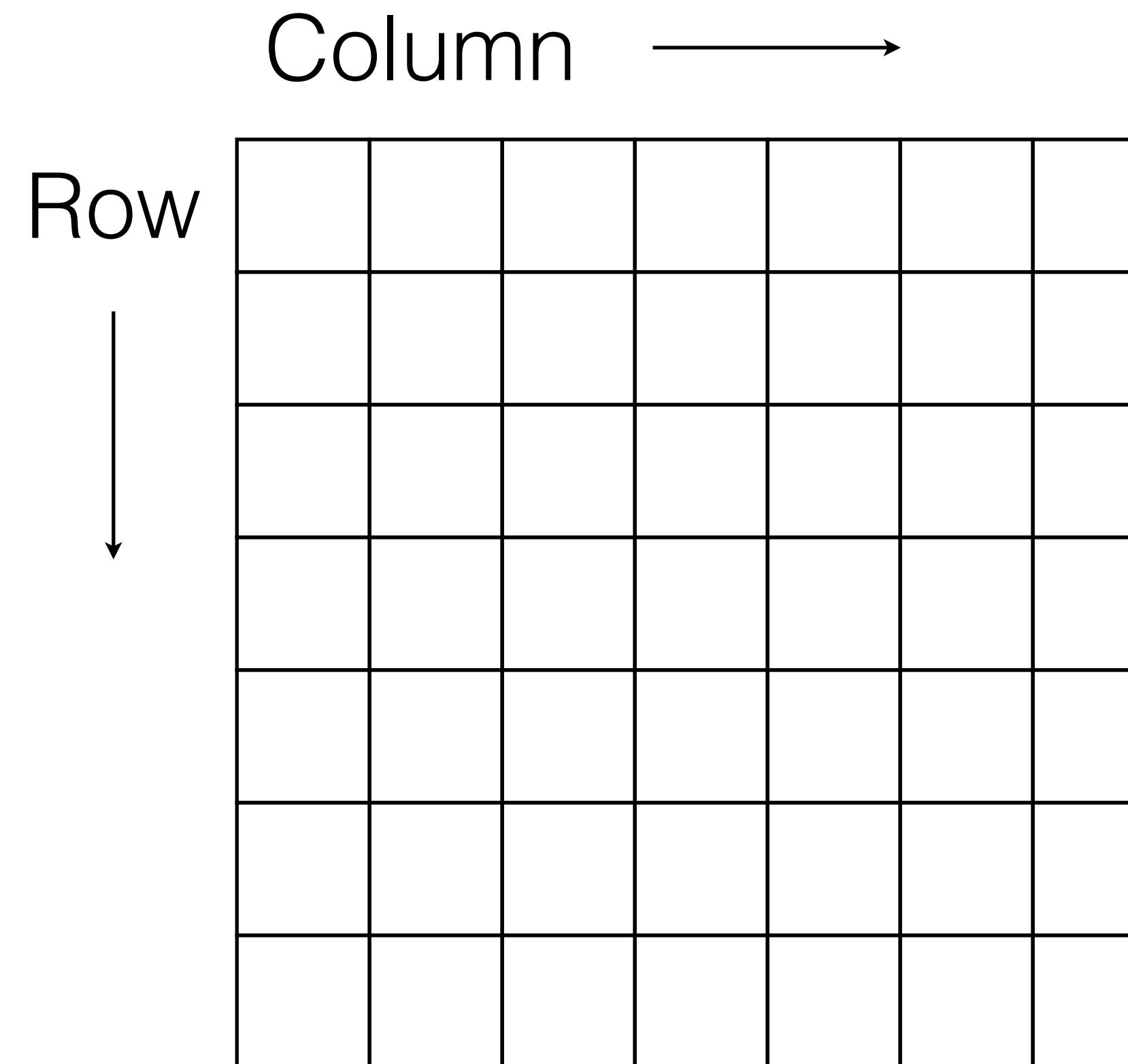


Computer screens
("left handed")

Math vs. Code

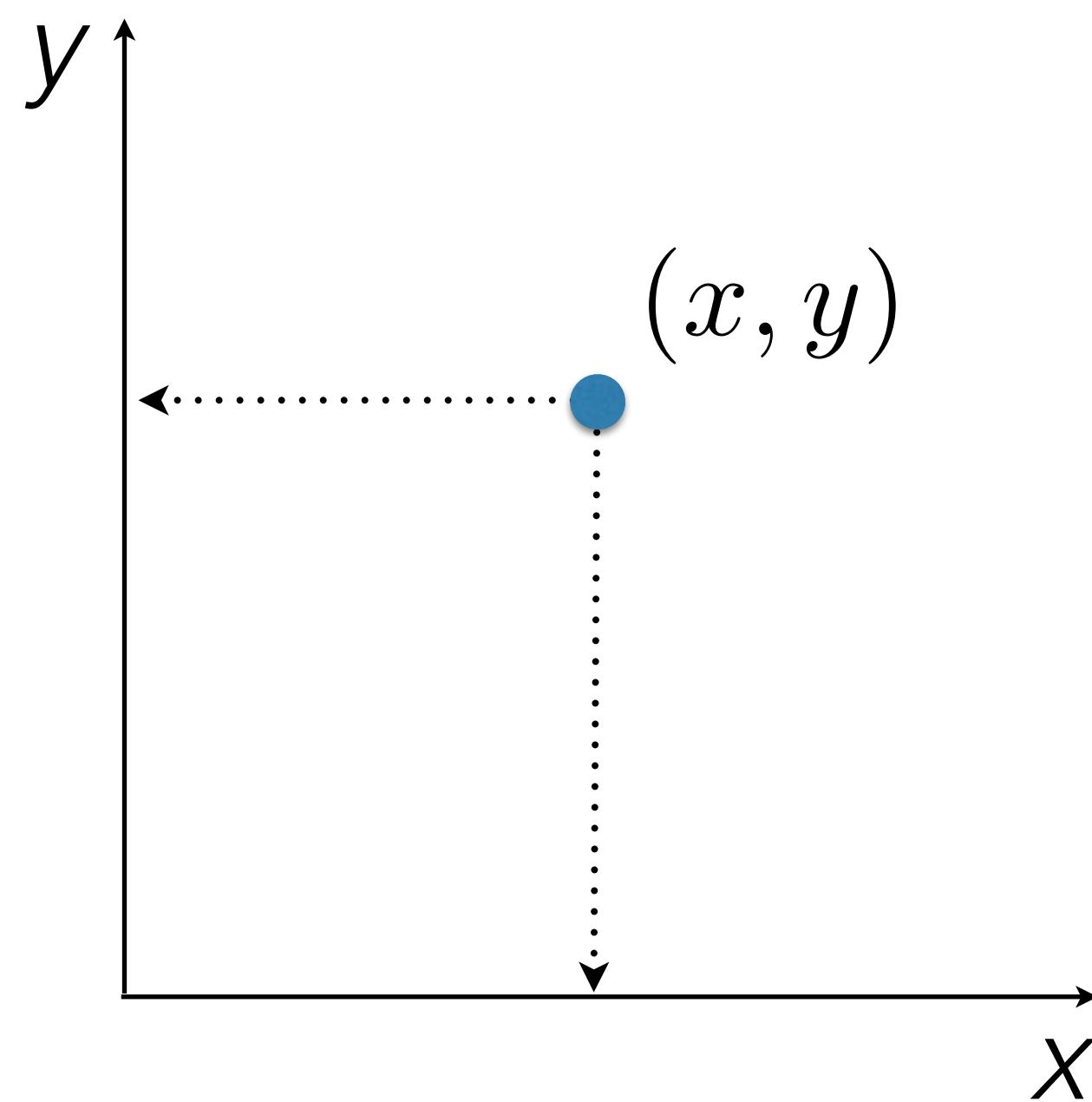


How we draw
(x,y)



How we store
(row,col)

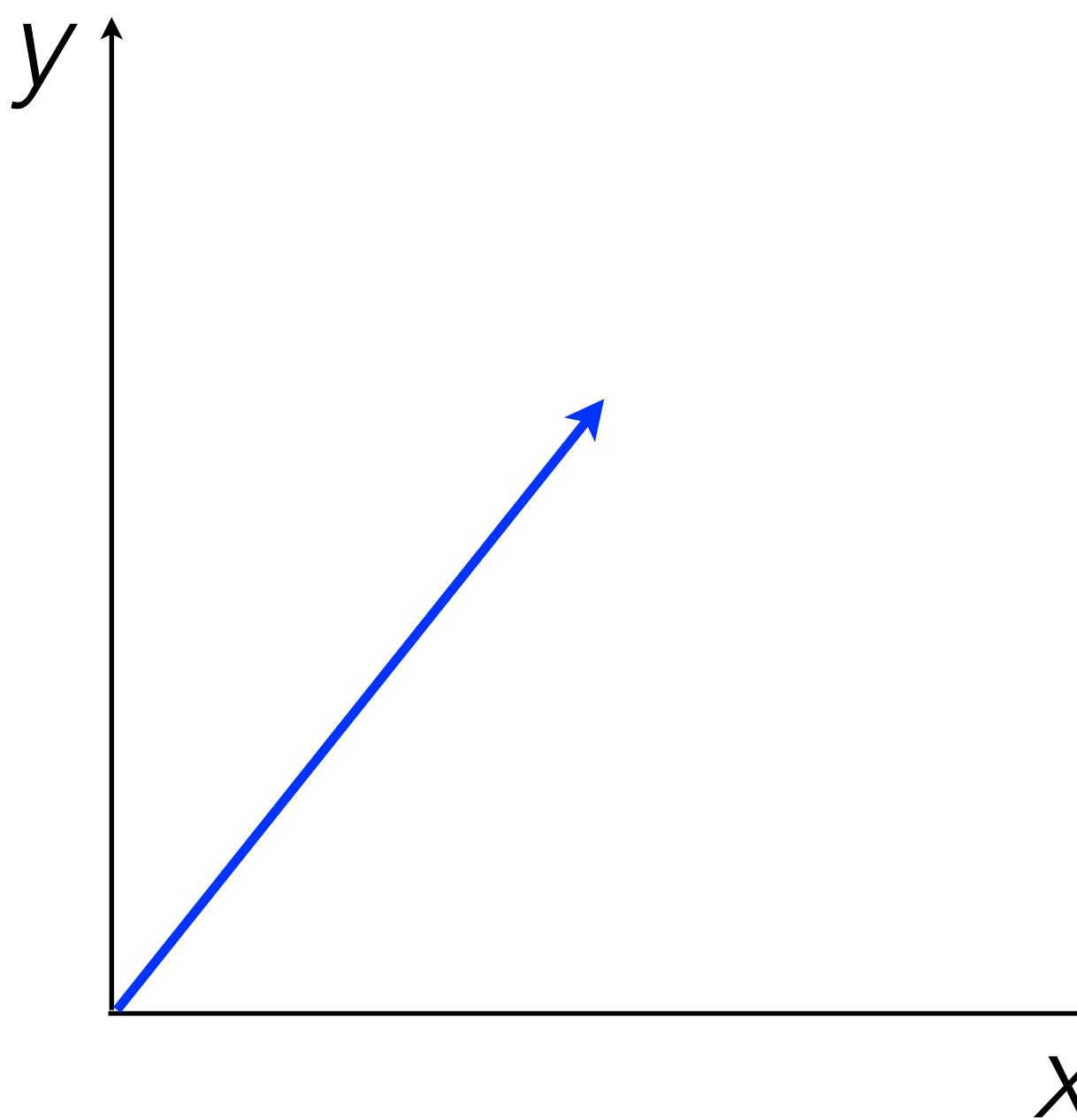
Points



Points can be described by their Cartesian coordinates.

Coding: use short arrays or n-tuples

Vectors



Vectors can also be thought of
in terms of Cartesian coordinates

Vectors

$$\mathbf{a} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Coding: just store in an array

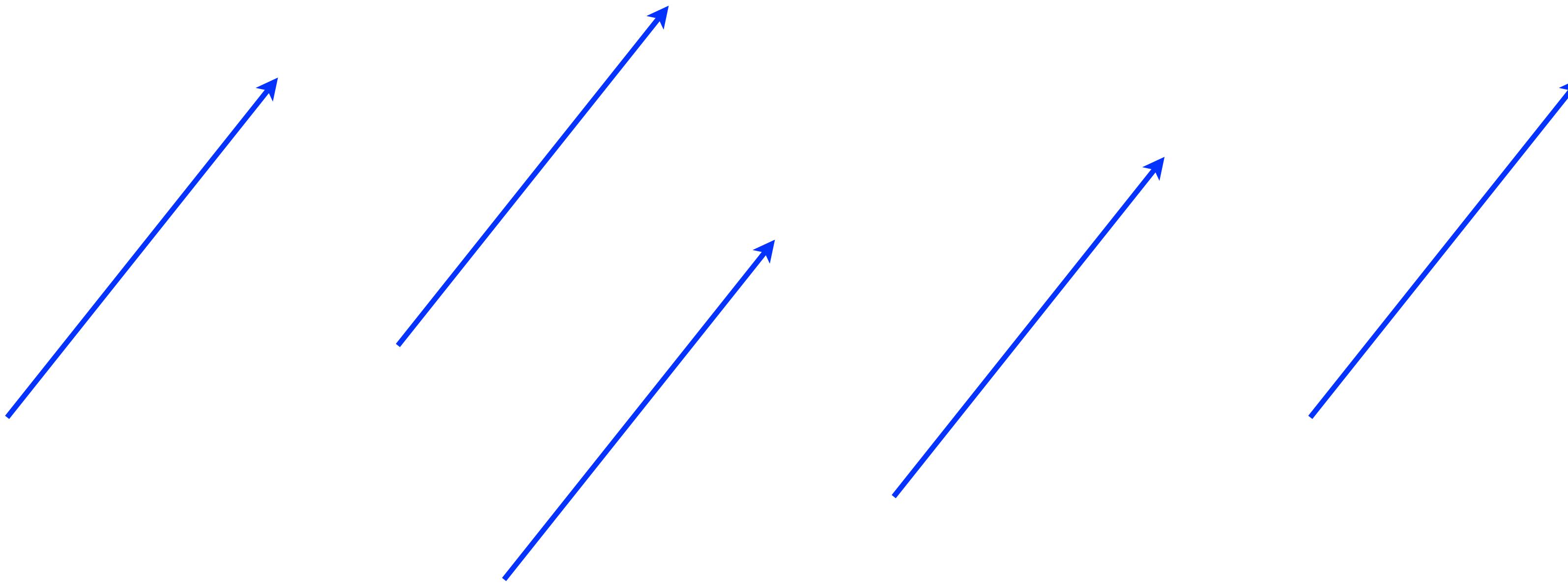
Vectors

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$$

Often use subscripts to denote elements of a vector.

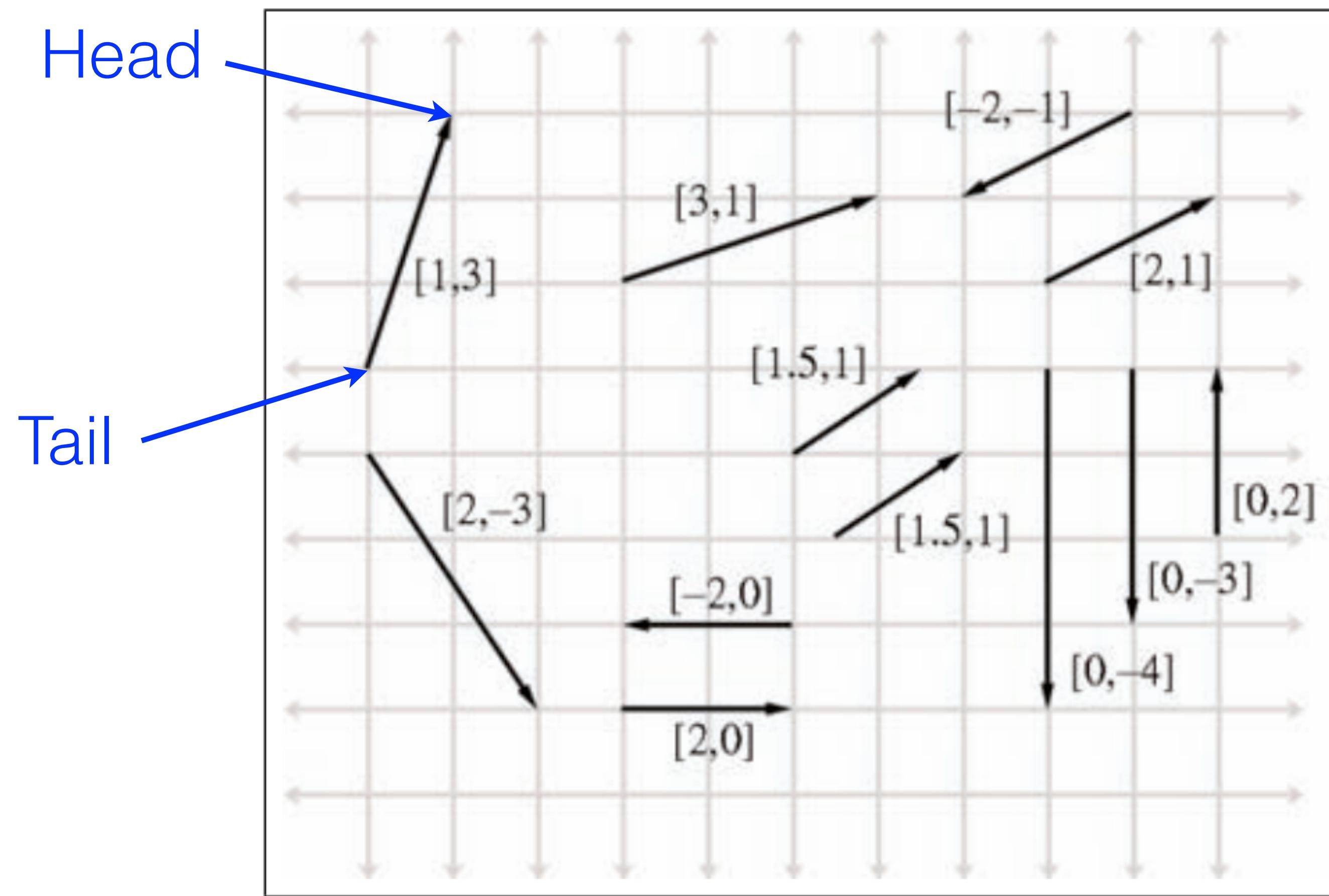
Coding: use `a[i]`

Vectors



Vectors are directional quantities
without a specific location

Vectors as Displacements

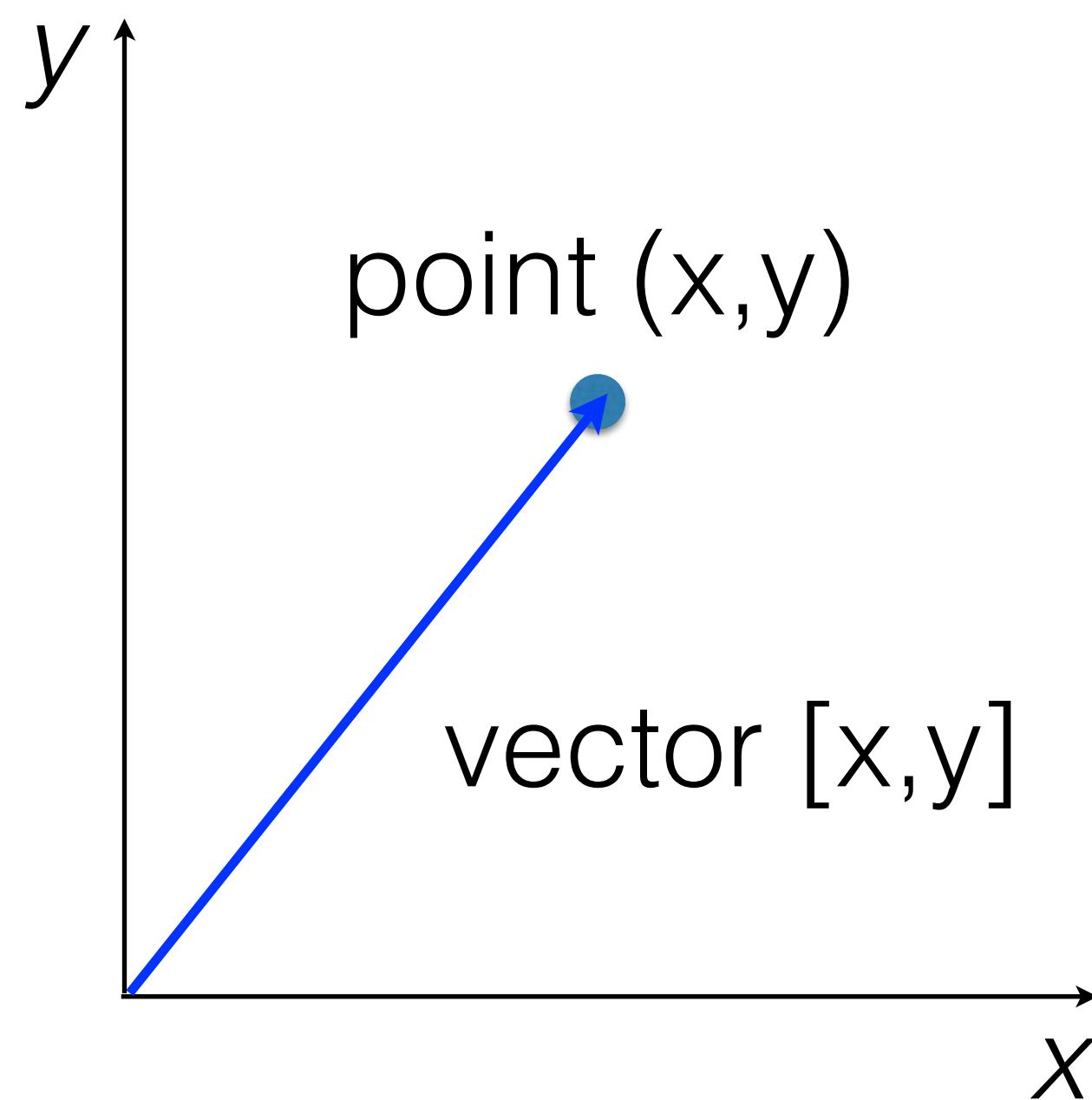


Zero Vector

- The *zero vector* is all zeroes
- No displacement
- Magnitude is 0
- Direction is undefined

$$\mathbf{0} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Points and Vectors

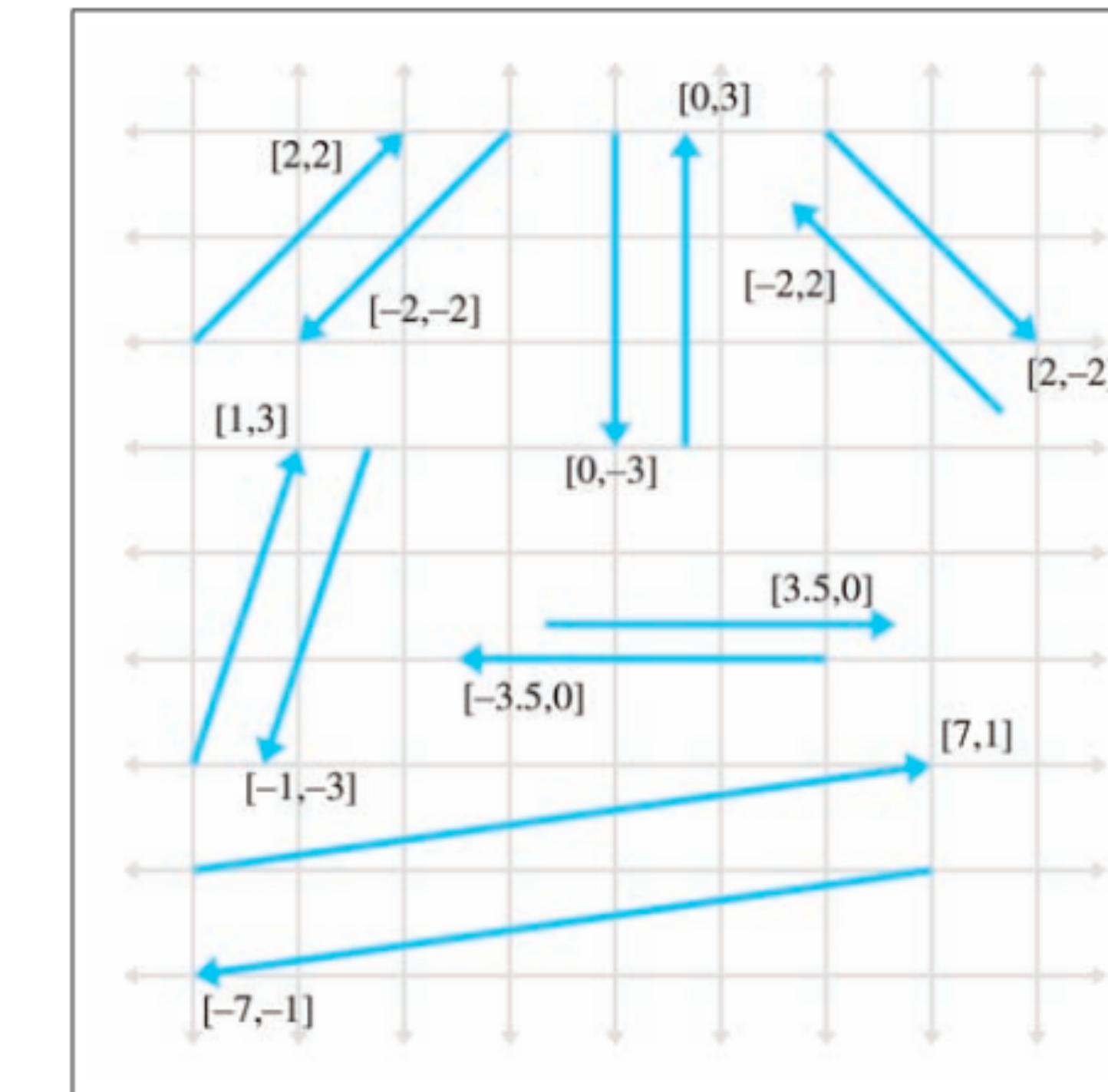


Points and vectors are different but related
Interchangeable (but be careful)

Negating Vectors

$$\mathbf{v} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

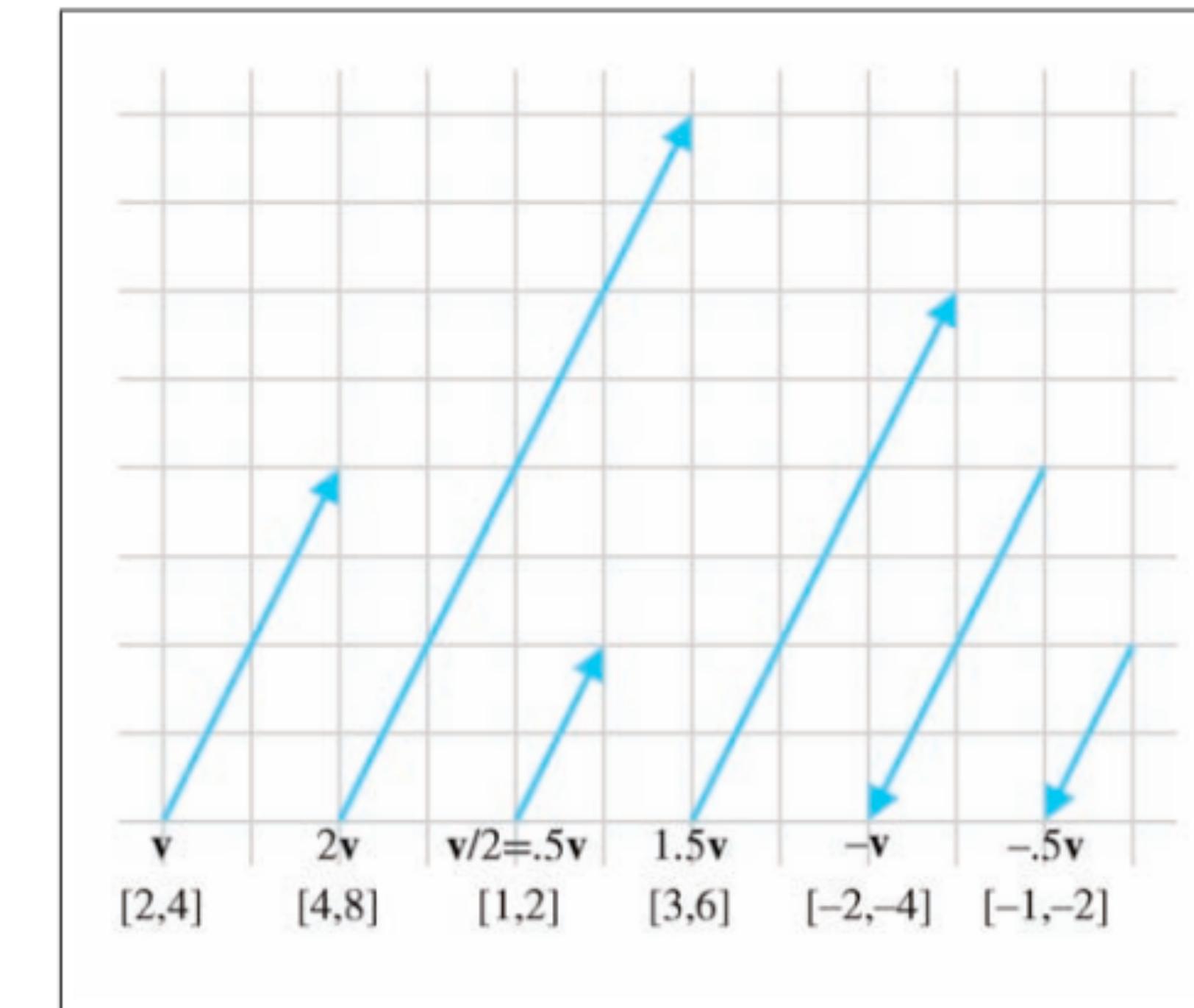
$$-\mathbf{v} = \begin{bmatrix} -x \\ -y \\ -z \end{bmatrix}$$



The negative of a vector has the same magnitude
in the opposite direction

Scaling Vectors

$$\mathbf{v} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$
$$k \mathbf{v} = \begin{bmatrix} kx \\ ky \\ kz \end{bmatrix}$$



Multiplying by a constant multiplies each element
-- multiplies magnitude, same (or opposite) direction

Adding Vectors

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

$$\mathbf{a} + \mathbf{b} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{bmatrix}$$

Point-wise add the elements

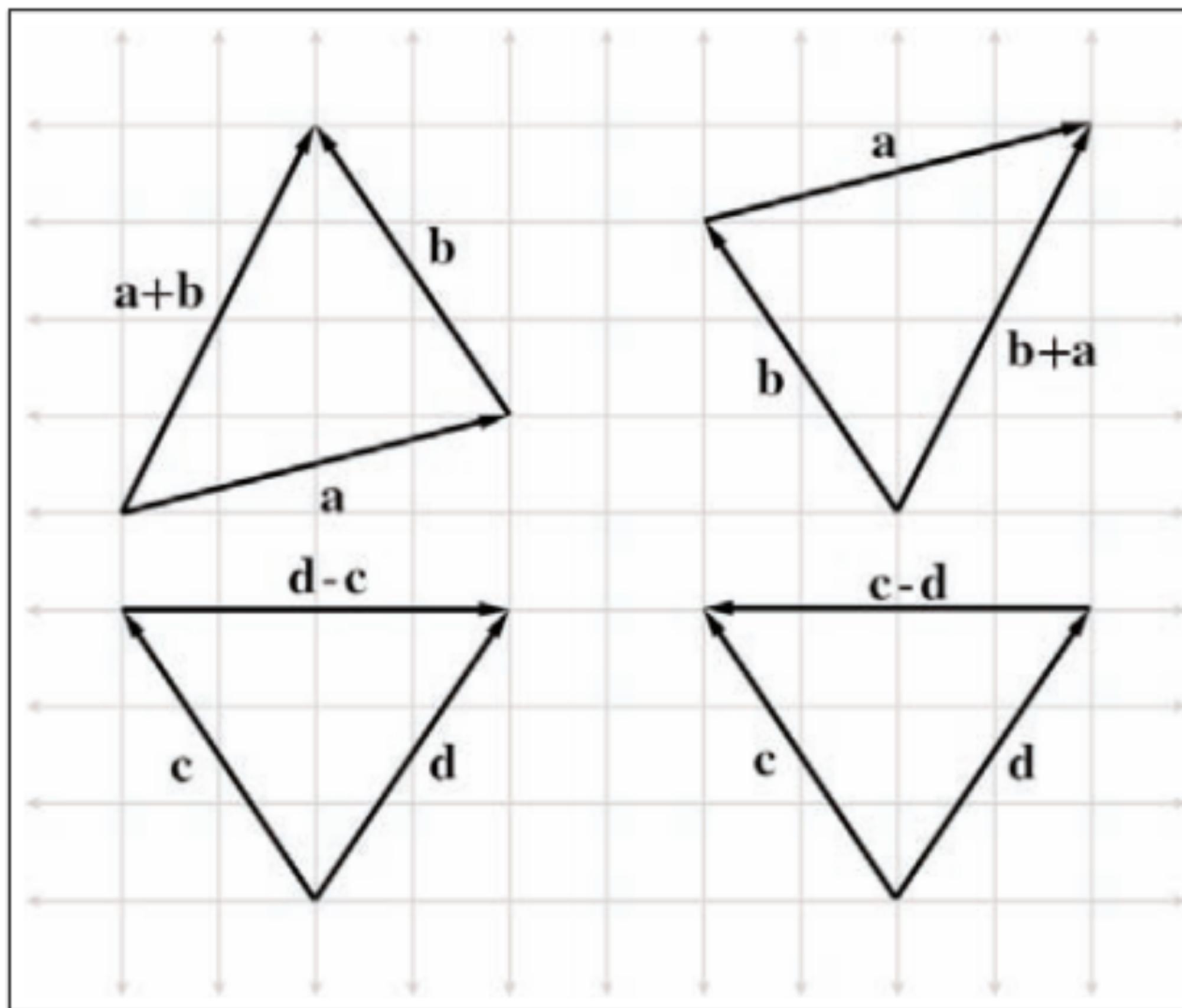
Subtracting Vectors

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

$$\mathbf{a} - \mathbf{b} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} - \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_1 - b_1 \\ a_2 - b_2 \\ a_3 - b_3 \end{bmatrix}$$

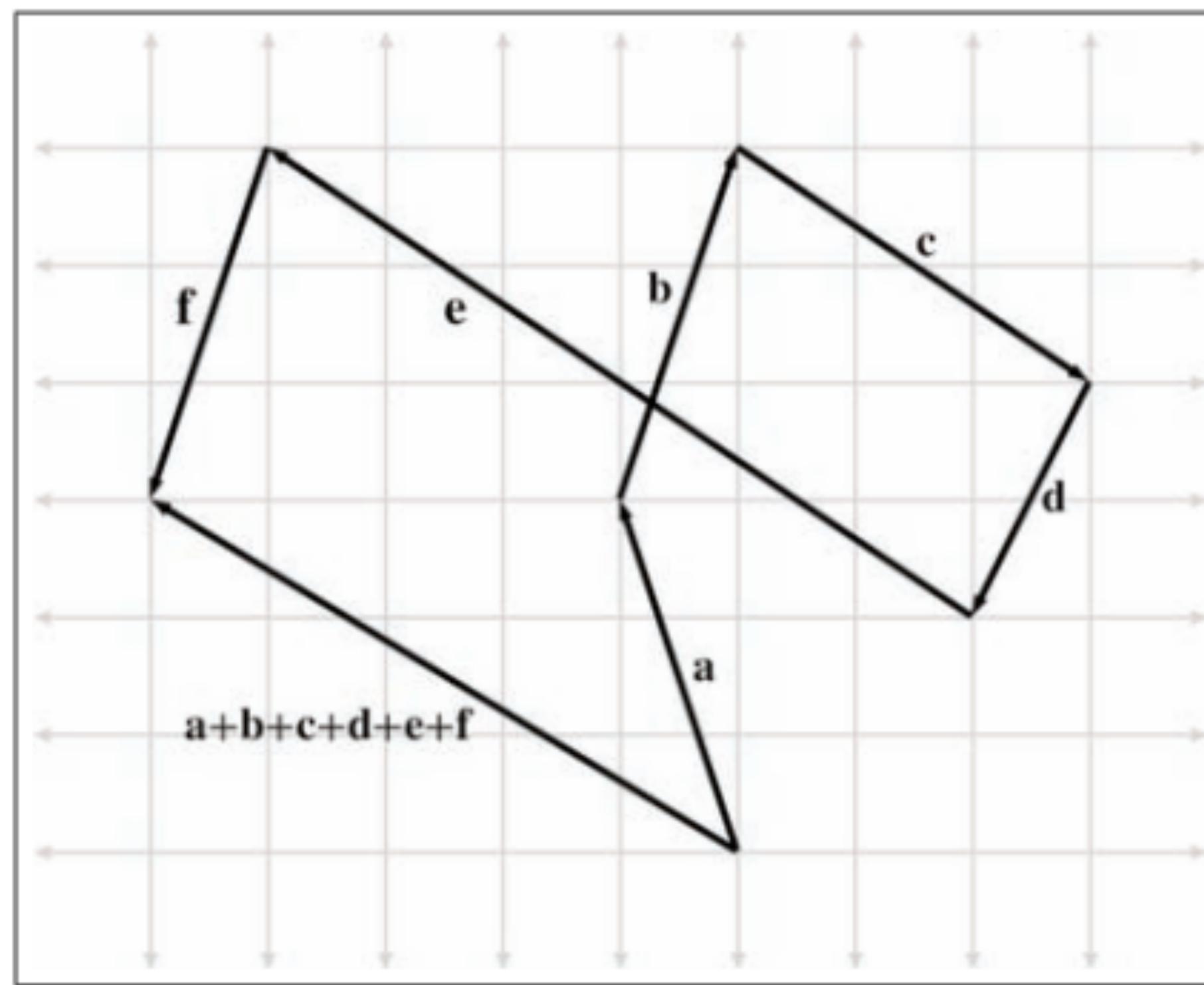
Point-wise subtract the elements

Geometric Interpretation

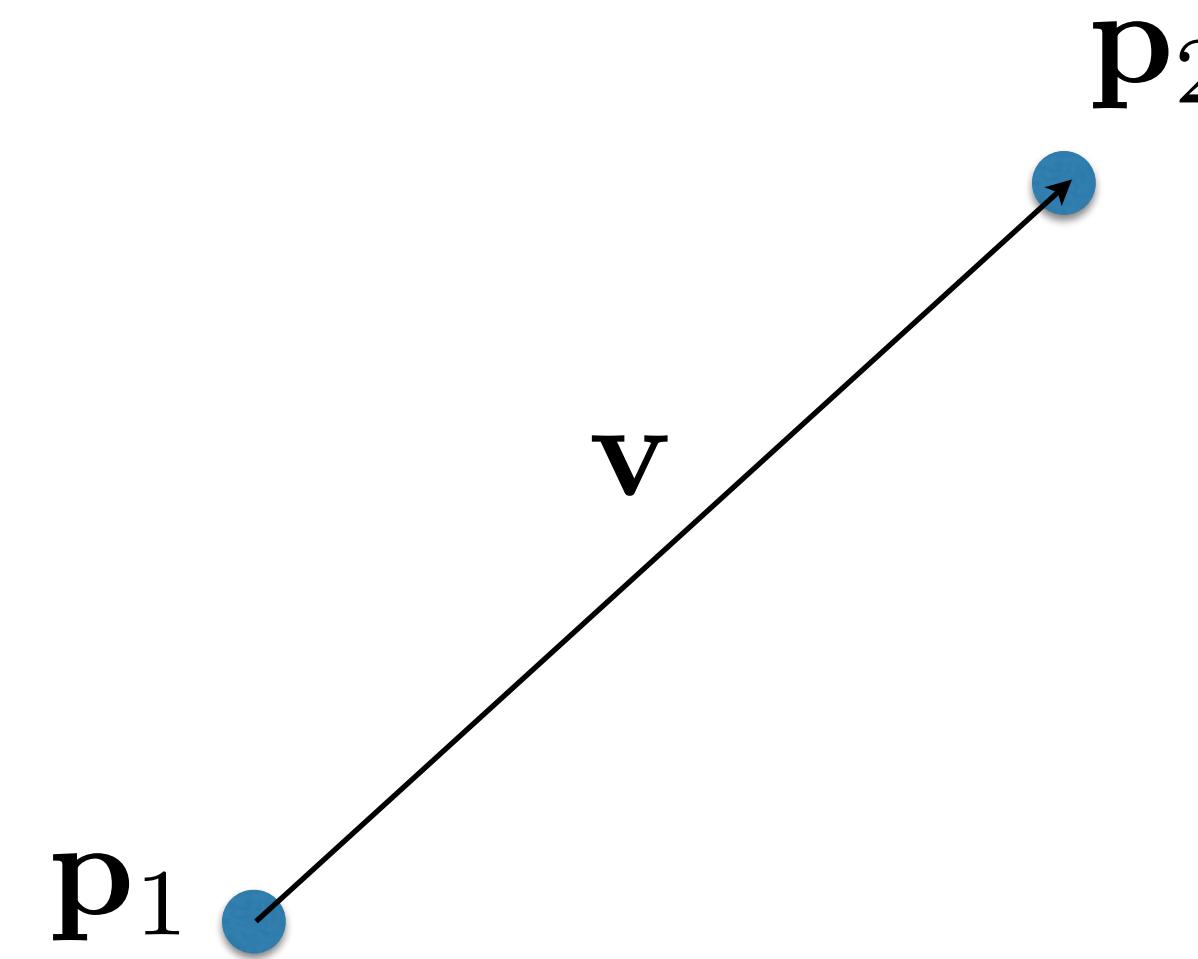


“Go this way, then go that way”

Geometric Interpretation



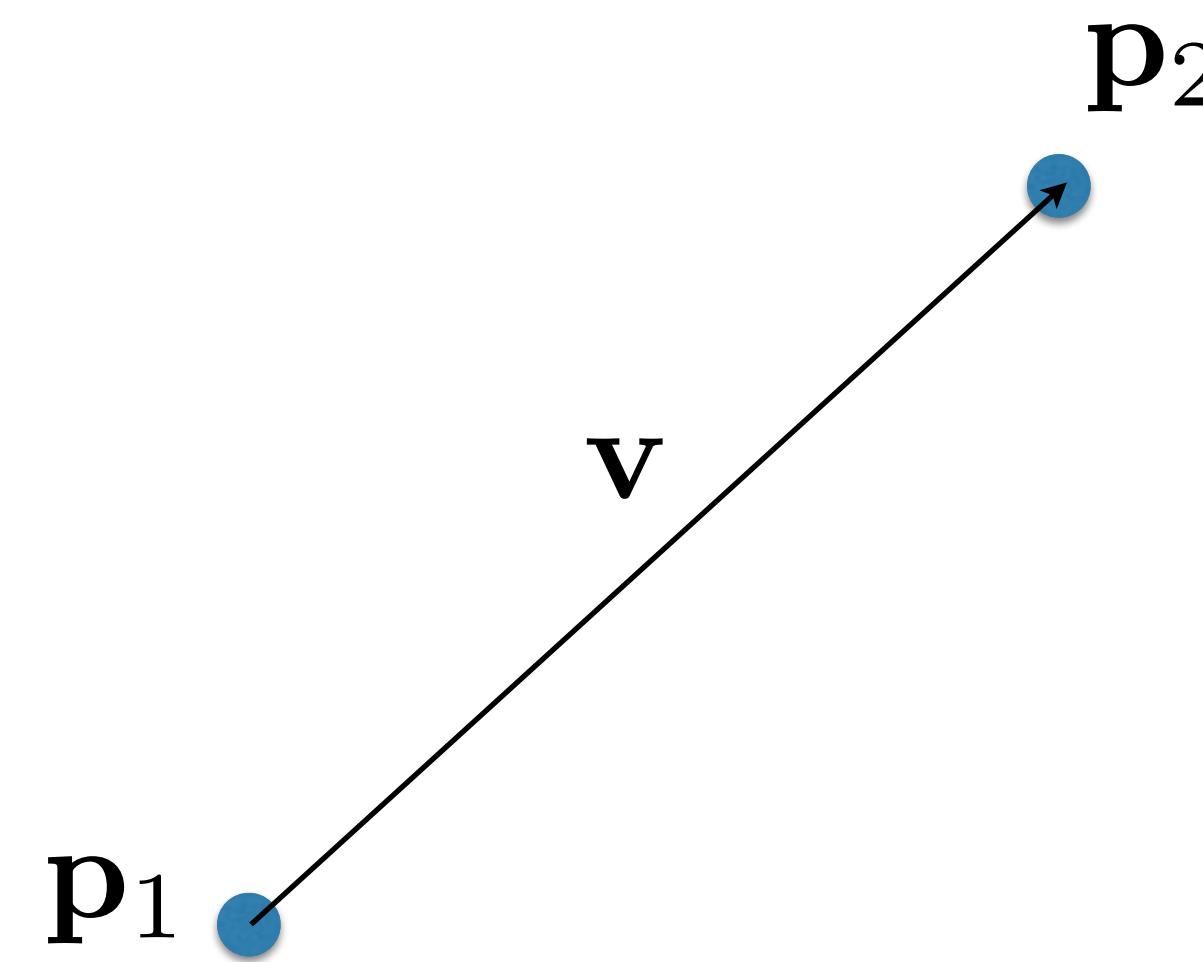
Vectors as Displacements



$$\mathbf{p}_1 + \mathbf{v} = \mathbf{p}_2$$

Type system: point + vector = point

Vectors as Displacements

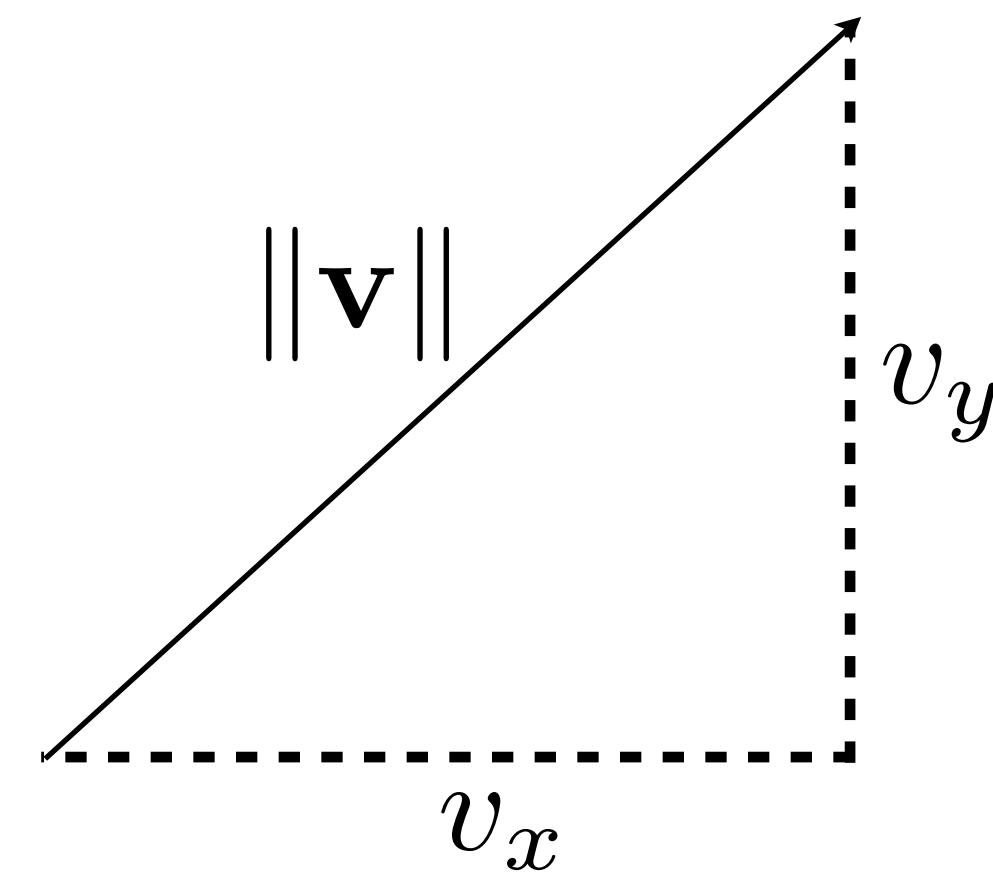


$$\mathbf{v} = \mathbf{p}_2 - \mathbf{p}_1$$

Type system: point - point = vector

Magnitude

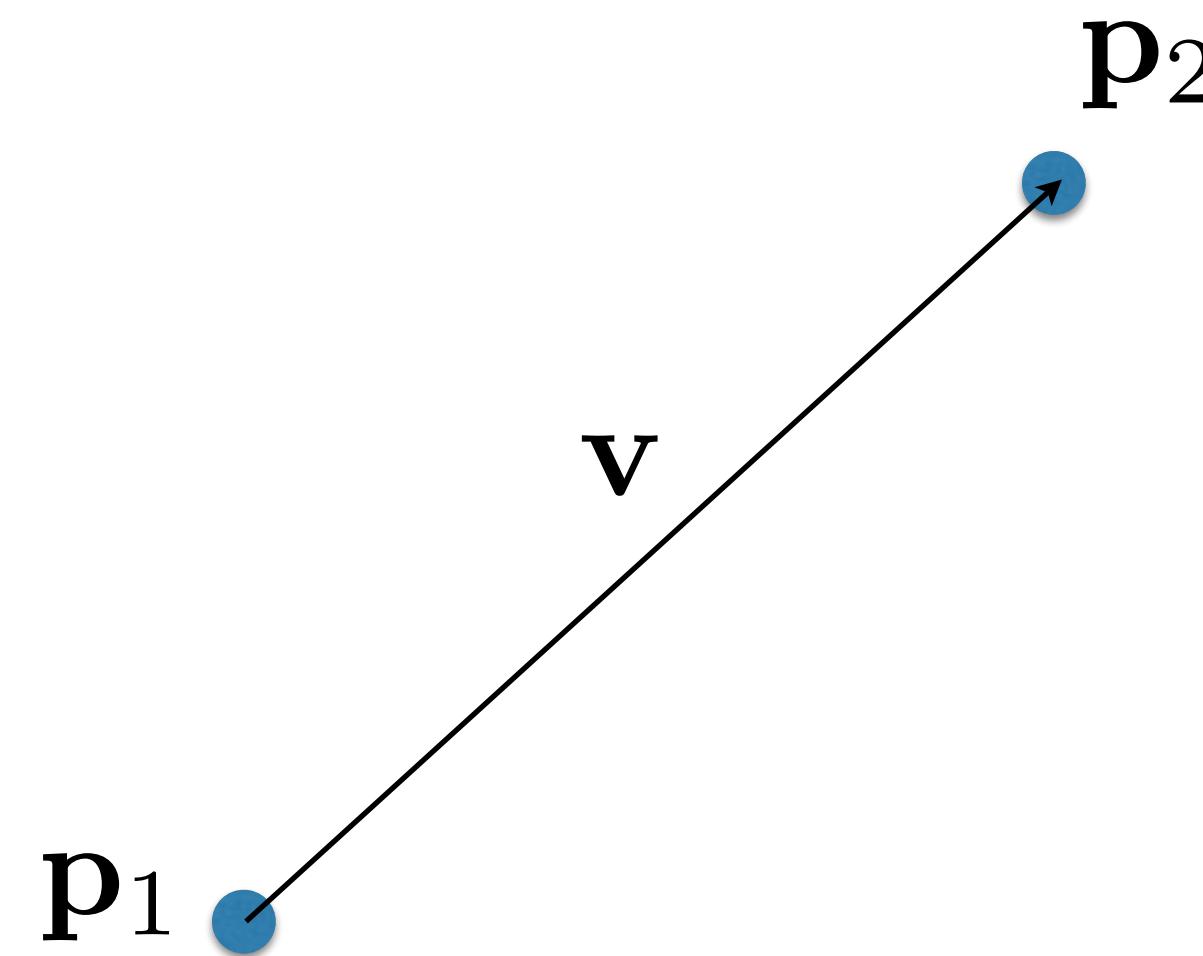
- The magnitude (length) of a vector can be calculated using Pythagorean theorem
- Sometimes called the *norm* of the vector
- Note: there are other vector norms, but assume this unless stated otherwise



$$\|\mathbf{v}\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

“magnitude”, “length”, or “norm” of \mathbf{v}

Distance



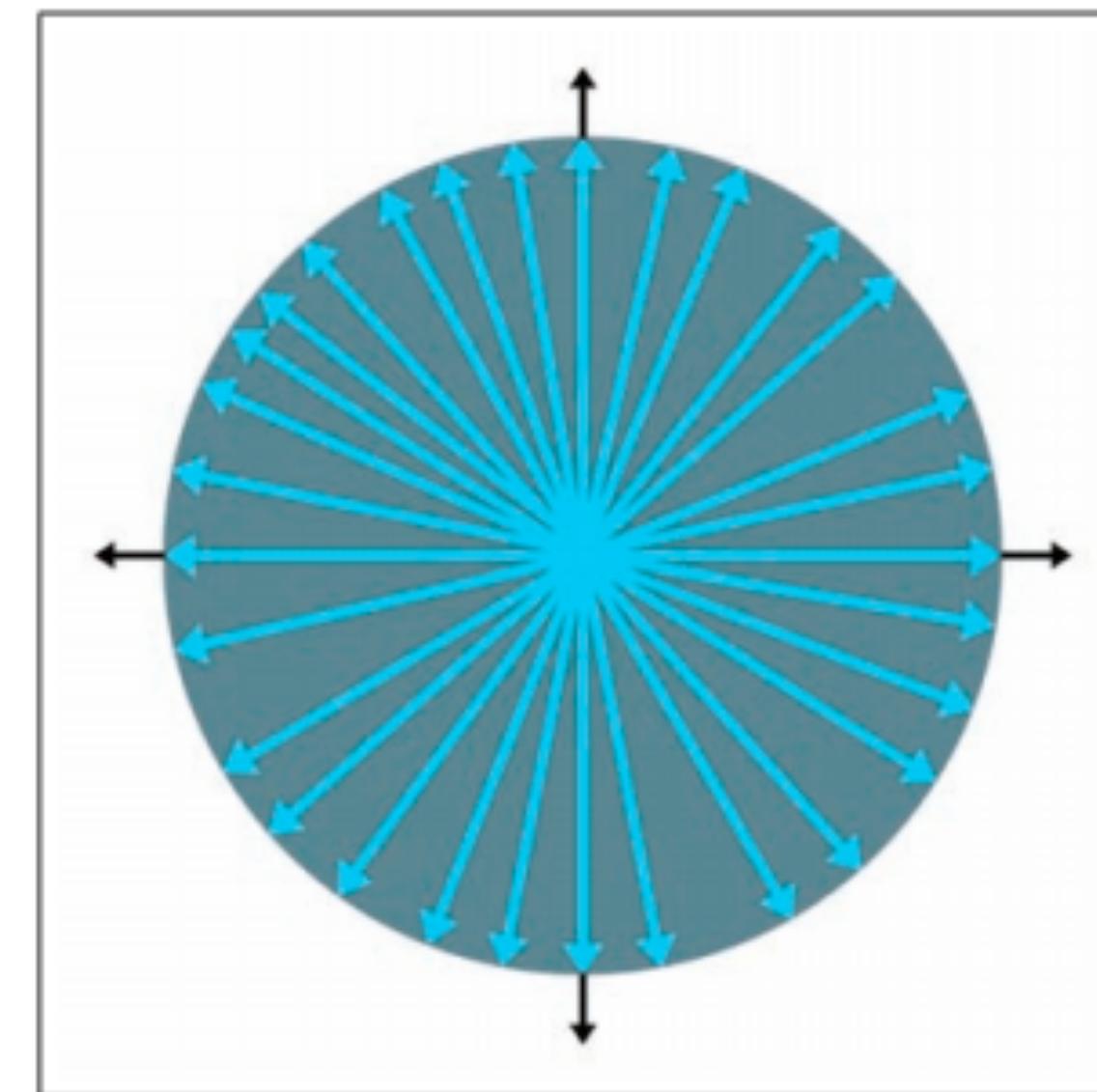
$$\|\mathbf{v}\| = \|\mathbf{p}_2 - \mathbf{p}_1\| = \sqrt{(p_2[x] - p_1[x])^2 + (p_2[y] - p_1[y])^2}$$

Convenient way to write / calculate distance between points

Unit Vectors

$$\|v\| = 1$$

- A “unit vector” has a **length of one**
- Useful to describe direction when we don’t care about magnitude



Normalizing

- Sometimes we want to **normalize** a vector to have the same direction but unit length
- Key: just divide it by its own length
- Can't do this for the zero vector

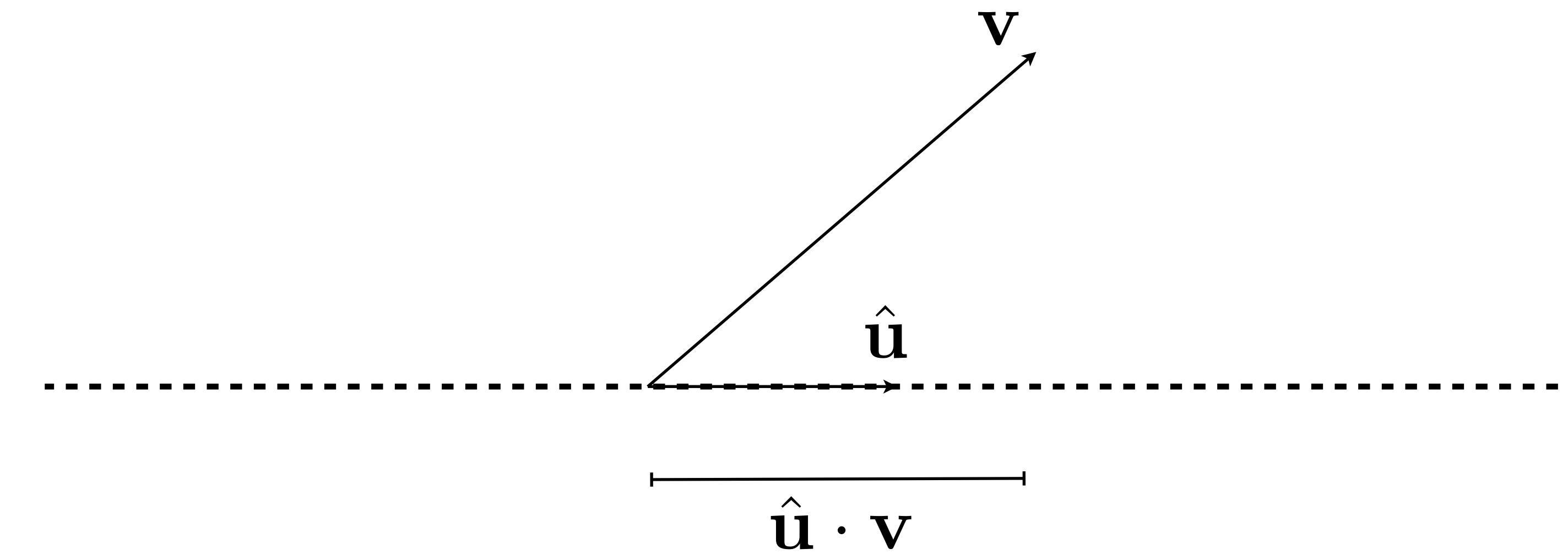
$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|} = \frac{\mathbf{v}}{\sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}}$$

Vector Dot Products

$$\mathbf{a} \cdot \mathbf{b} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \sum_{i=1}^n a_i b_i$$

One of the most commonly used
vector operations in graphics

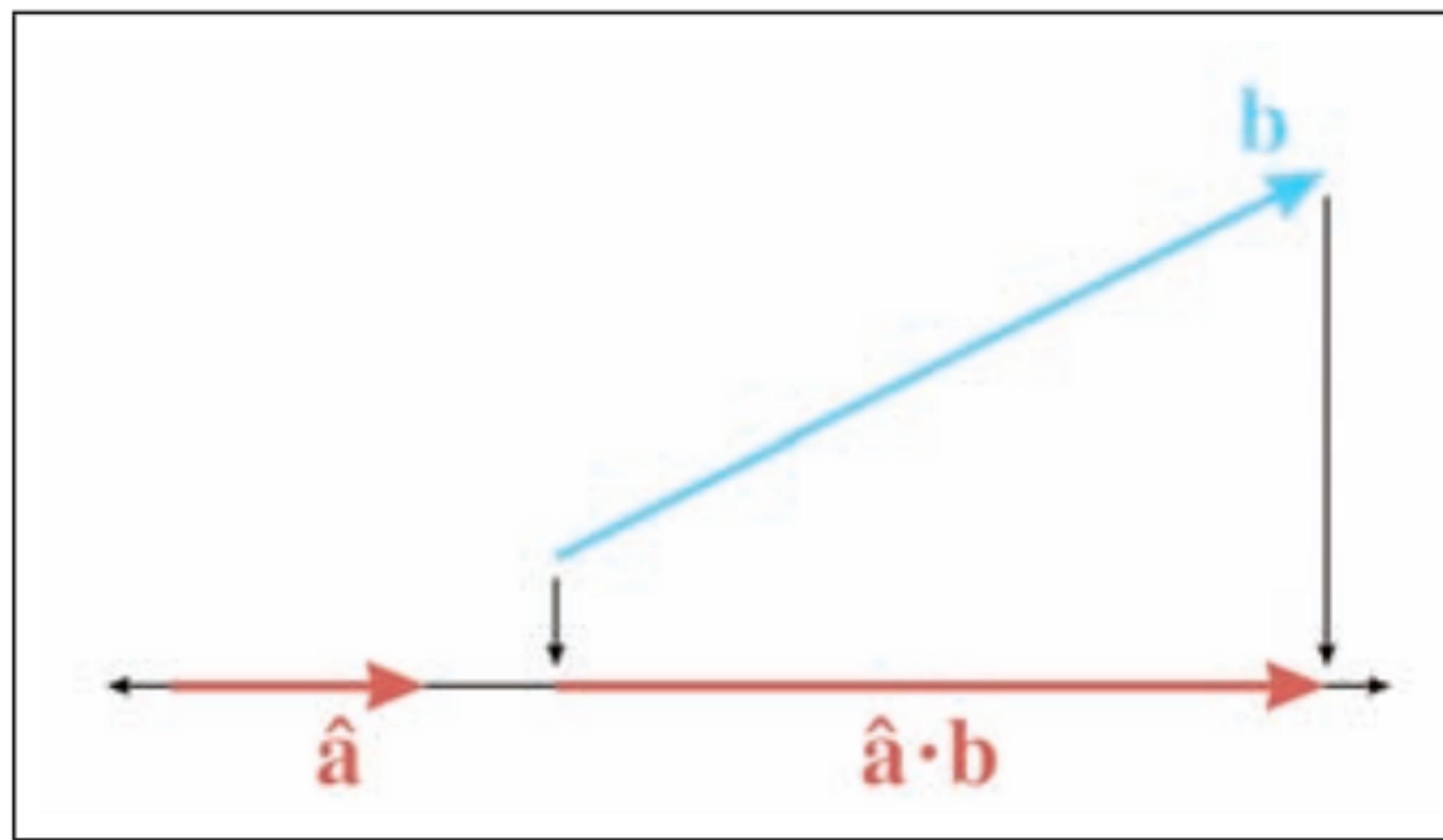
Geometric Interpretation



The dot product of a vector and a unit vector is the length of the projection onto that unit vector

“How much of this vector lies in that direction?”

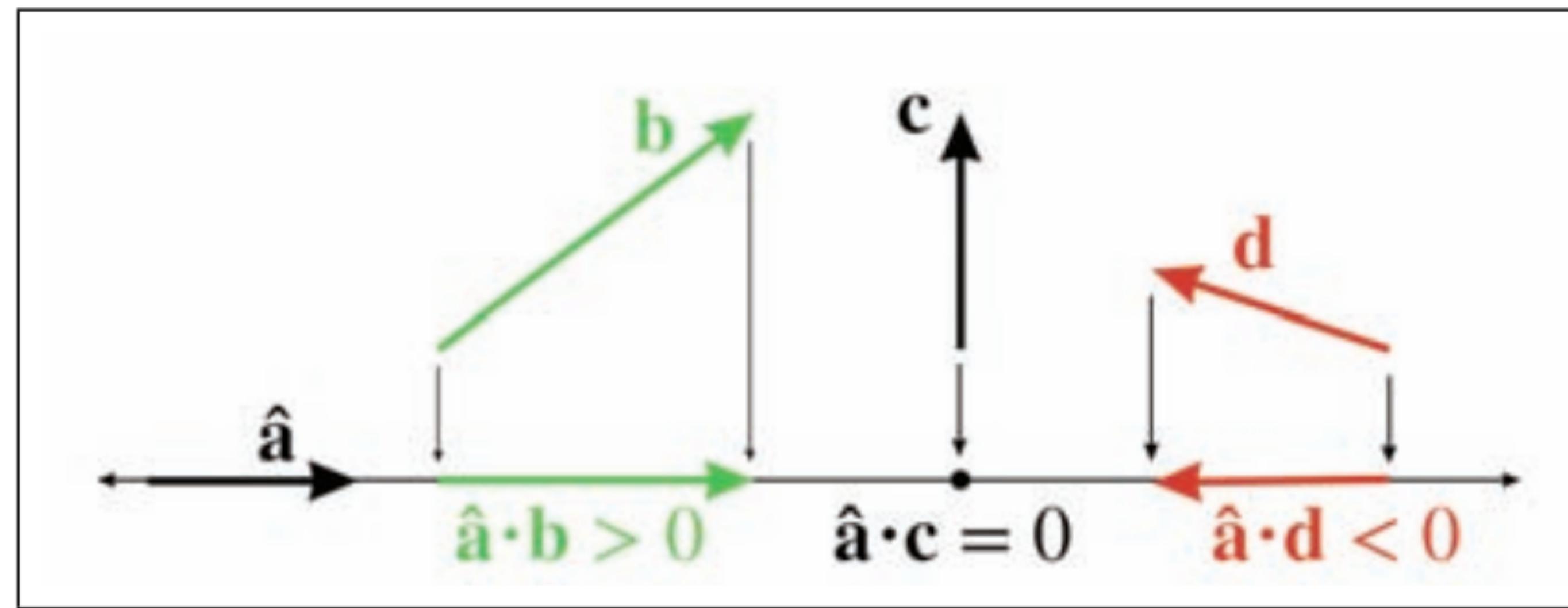
Geometric Interpretation



The dot product of a vector and a unit vector is the length of the projection onto that unit vector

“How much of this vector lies in that direction?”

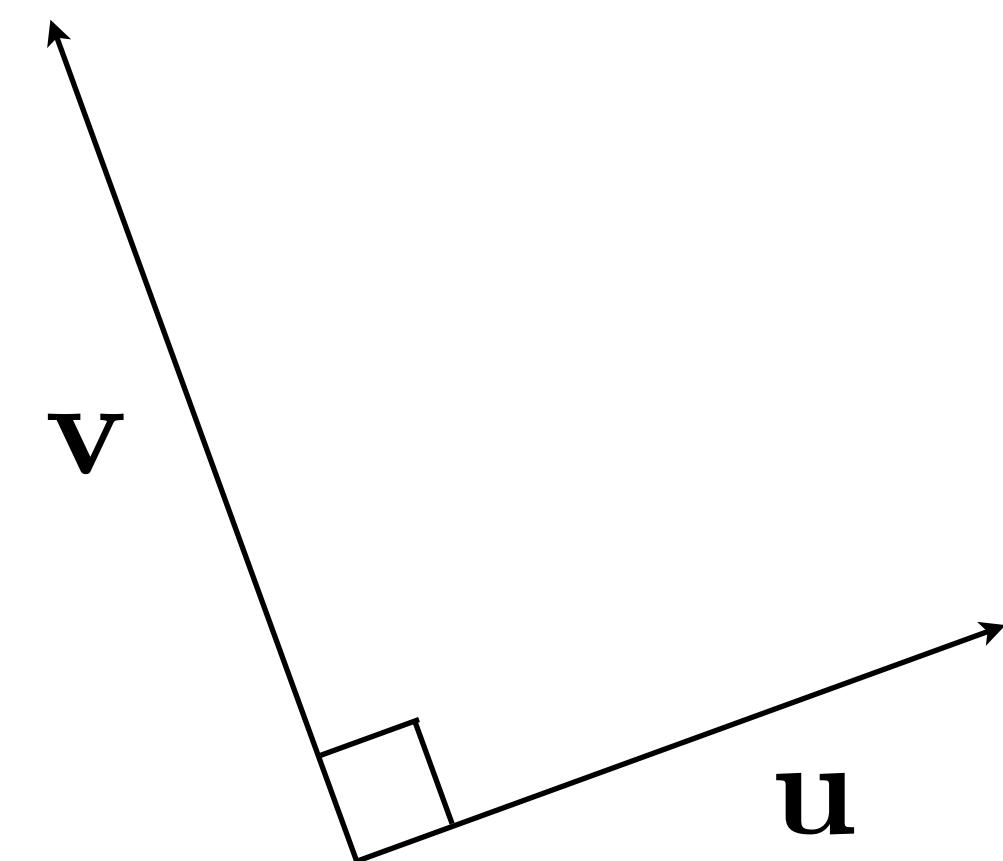
Sign of the Dot Product



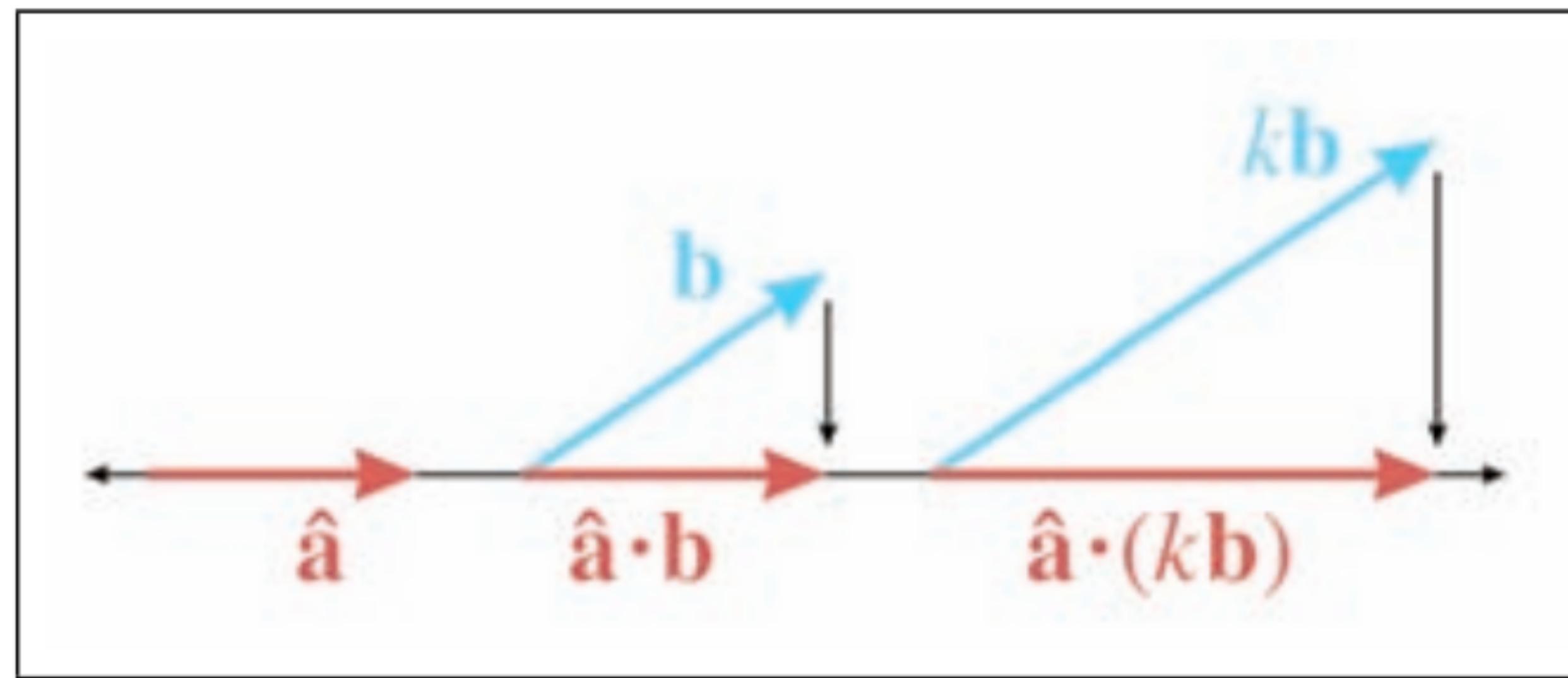
The sign of the dot product between two vectors tells whether the projection is in the same direction

Orthogonality

- Vectors whose dot product is zero are said to be “orthogonal”
- “Right angle” to each other (regardless of length)
- The zero vector is trivially orthogonal to everything else

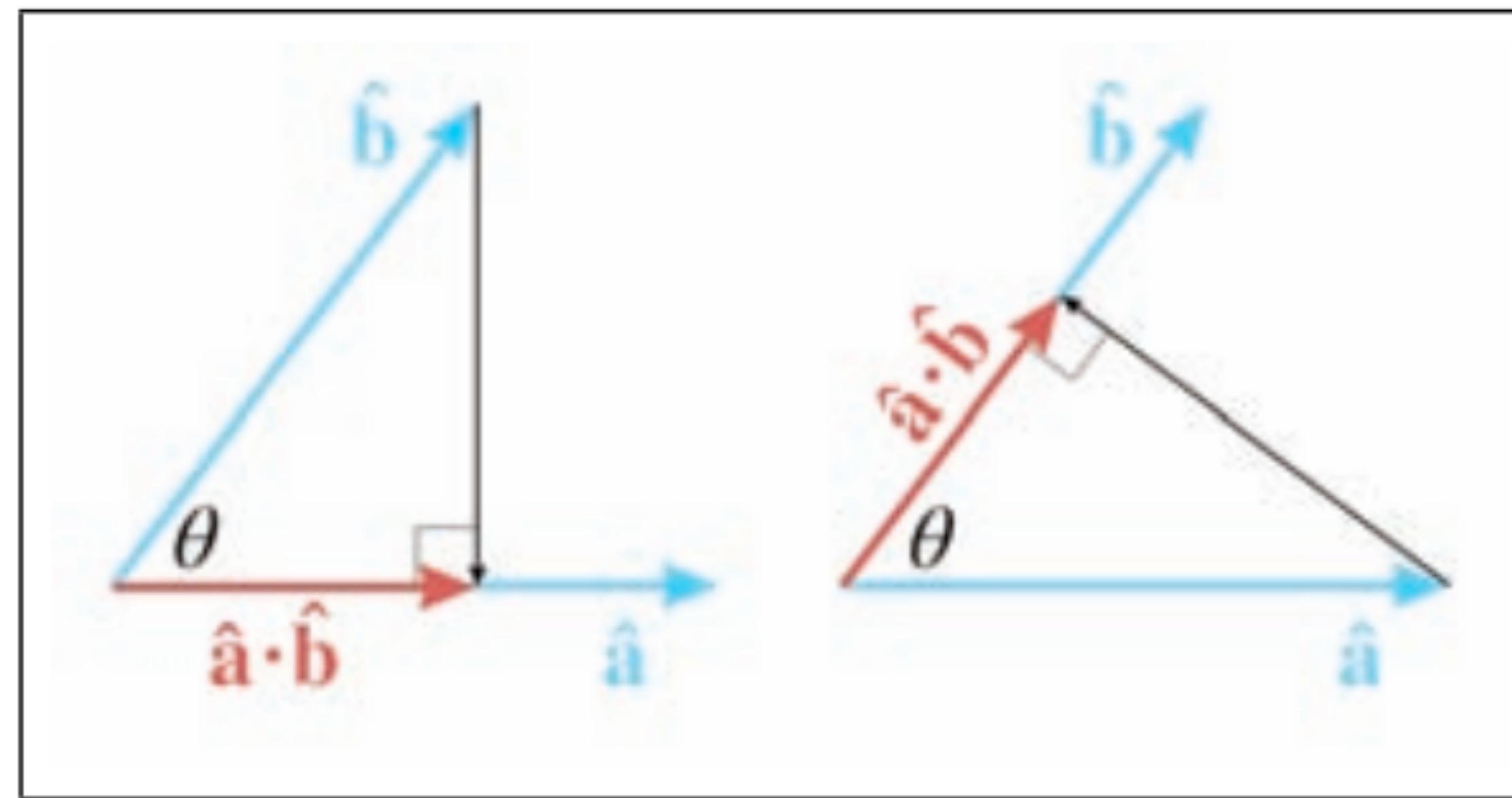


Scalar Multiplication



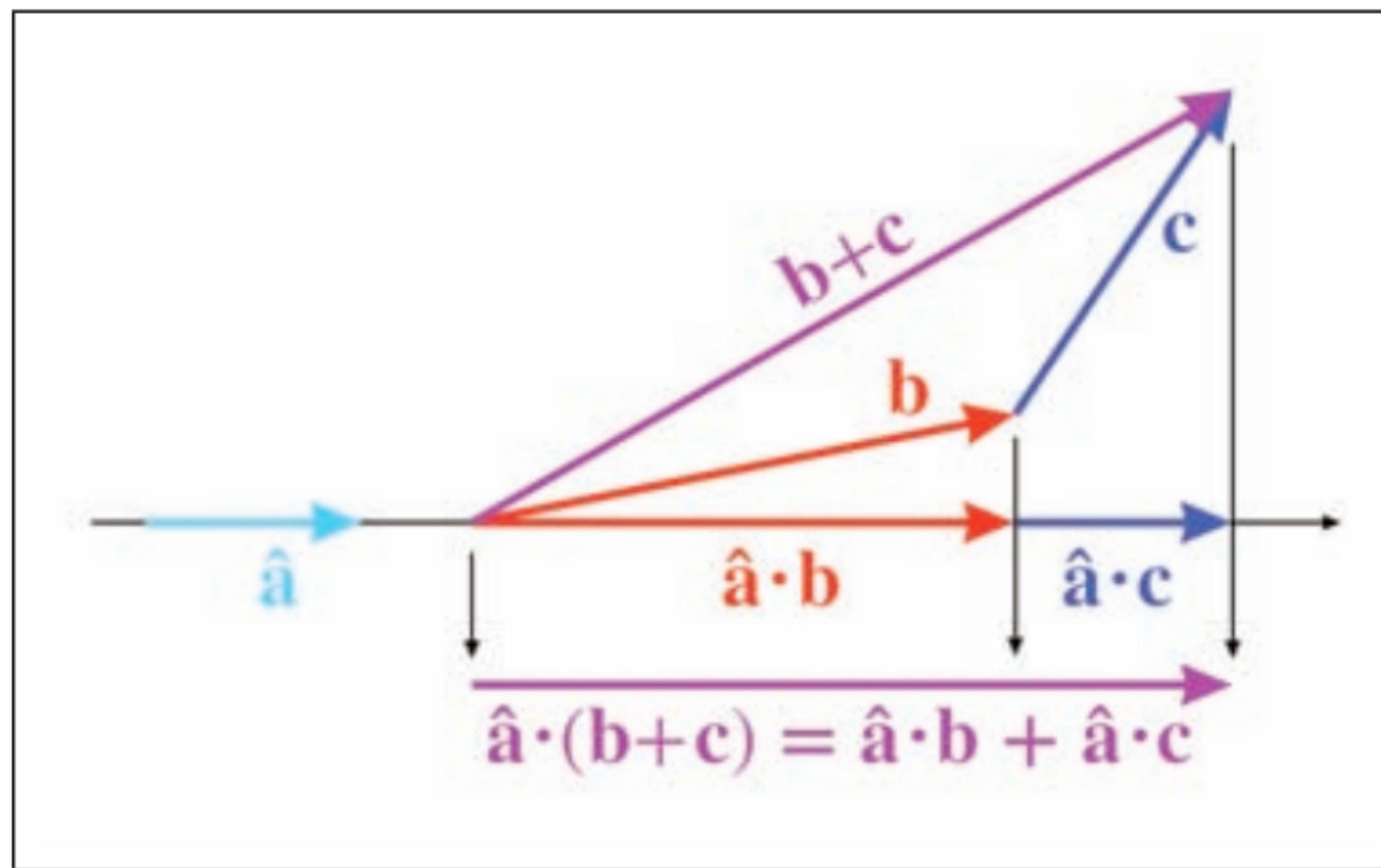
$$\mathbf{a} \cdot (k\mathbf{b}) = k(\mathbf{a} \cdot \mathbf{b}) = (k\mathbf{a}) \cdot \mathbf{b}$$

Commutative



$$\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$$

Distributes Over Addition



$$\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c}$$

Angles

$$\mathbf{u} = \|\mathbf{u}\| \hat{\mathbf{u}}$$

$$\mathbf{v} = \|\mathbf{v}\| \hat{\mathbf{v}}$$

$$\begin{aligned}\mathbf{u} \cdot \mathbf{v} &= \|\mathbf{u}\| \|\mathbf{v}\| (\hat{\mathbf{u}} \cdot \hat{\mathbf{v}}) \\ &= \|\mathbf{u}\| \|\mathbf{v}\| \cos \theta_{uv}\end{aligned}$$

The dot product of two vectors is the product of their lengths times the cosine of the angle between them

Lengths

$$\mathbf{v} \cdot \mathbf{v} = \|\mathbf{v}\|^2$$

The dot product of something with itself
is its own length squared

Tip: lots of “distance” tests only need squared distance

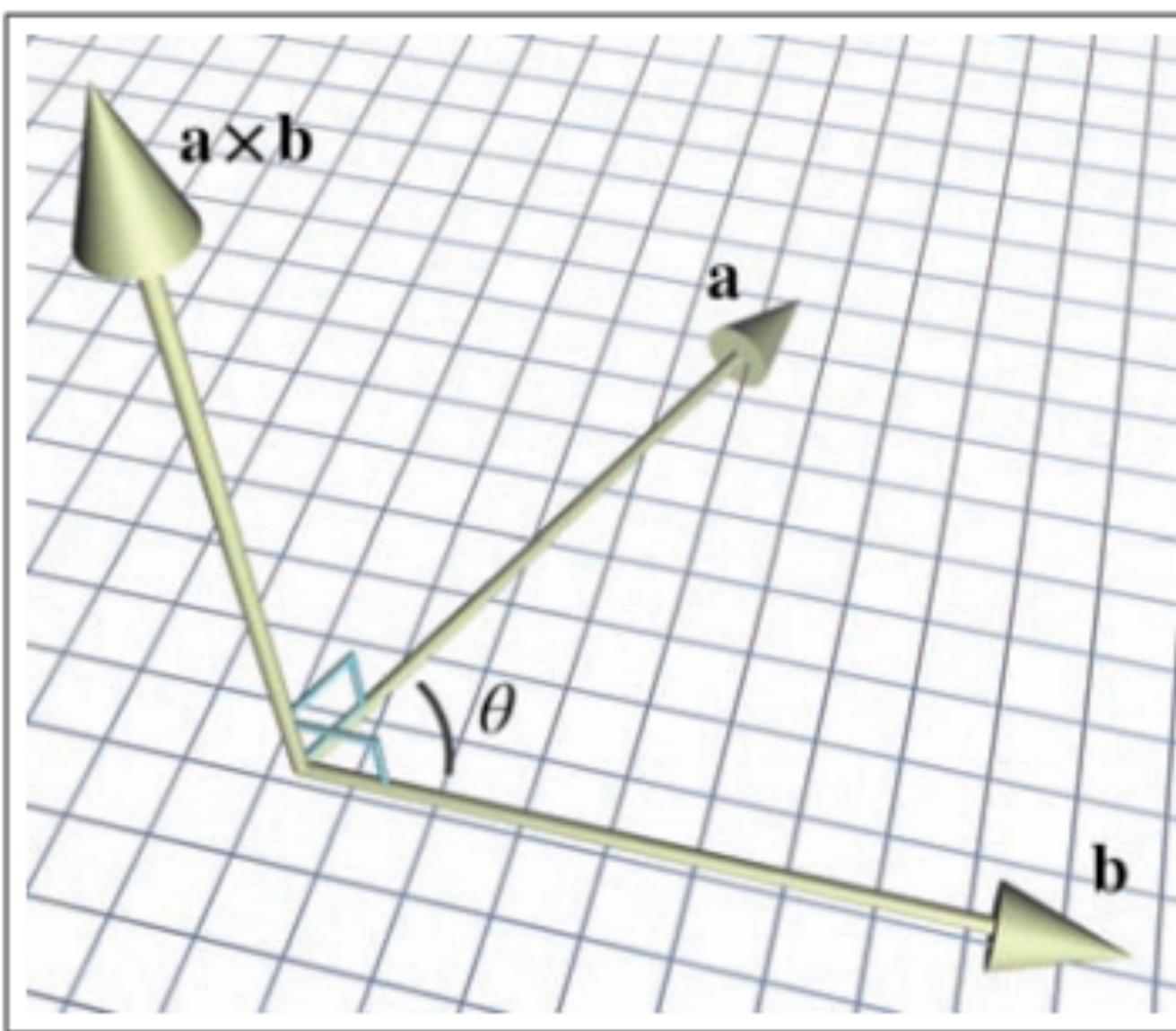
Cross Product

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \times \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} y_1 z_2 - z_1 y_2 \\ z_1 x_2 - x_1 z_2 \\ x_1 y_2 - y_1 x_2 \end{bmatrix}$$

Result is a vector

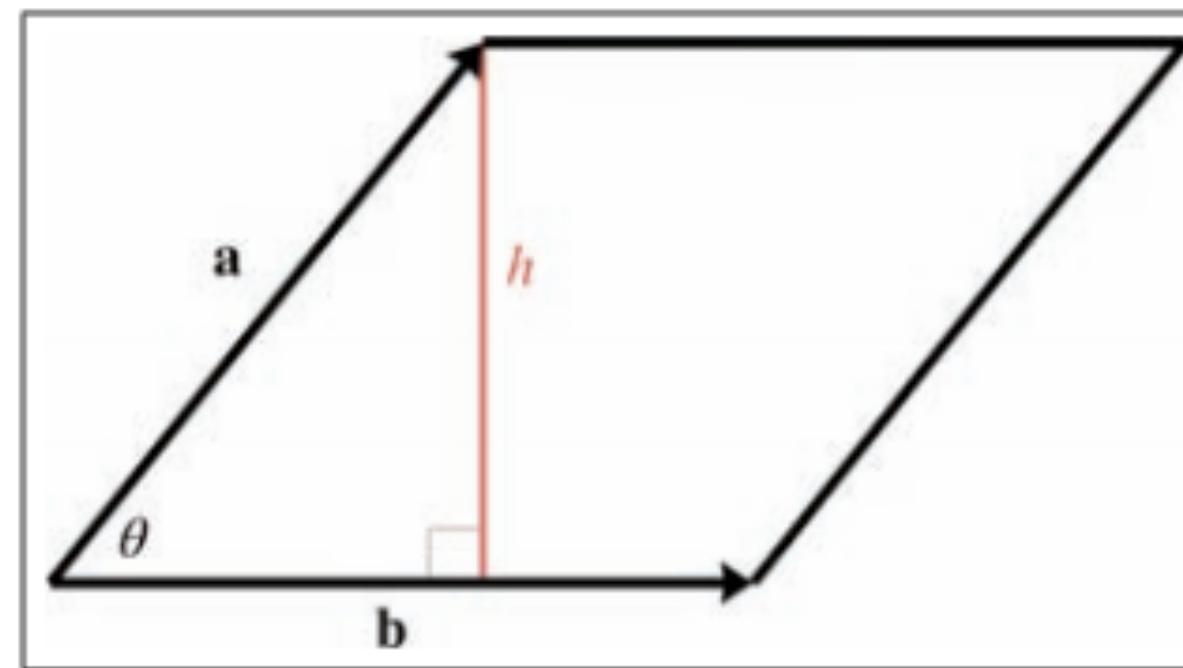
Only done in 3D

Geometric Interpretation



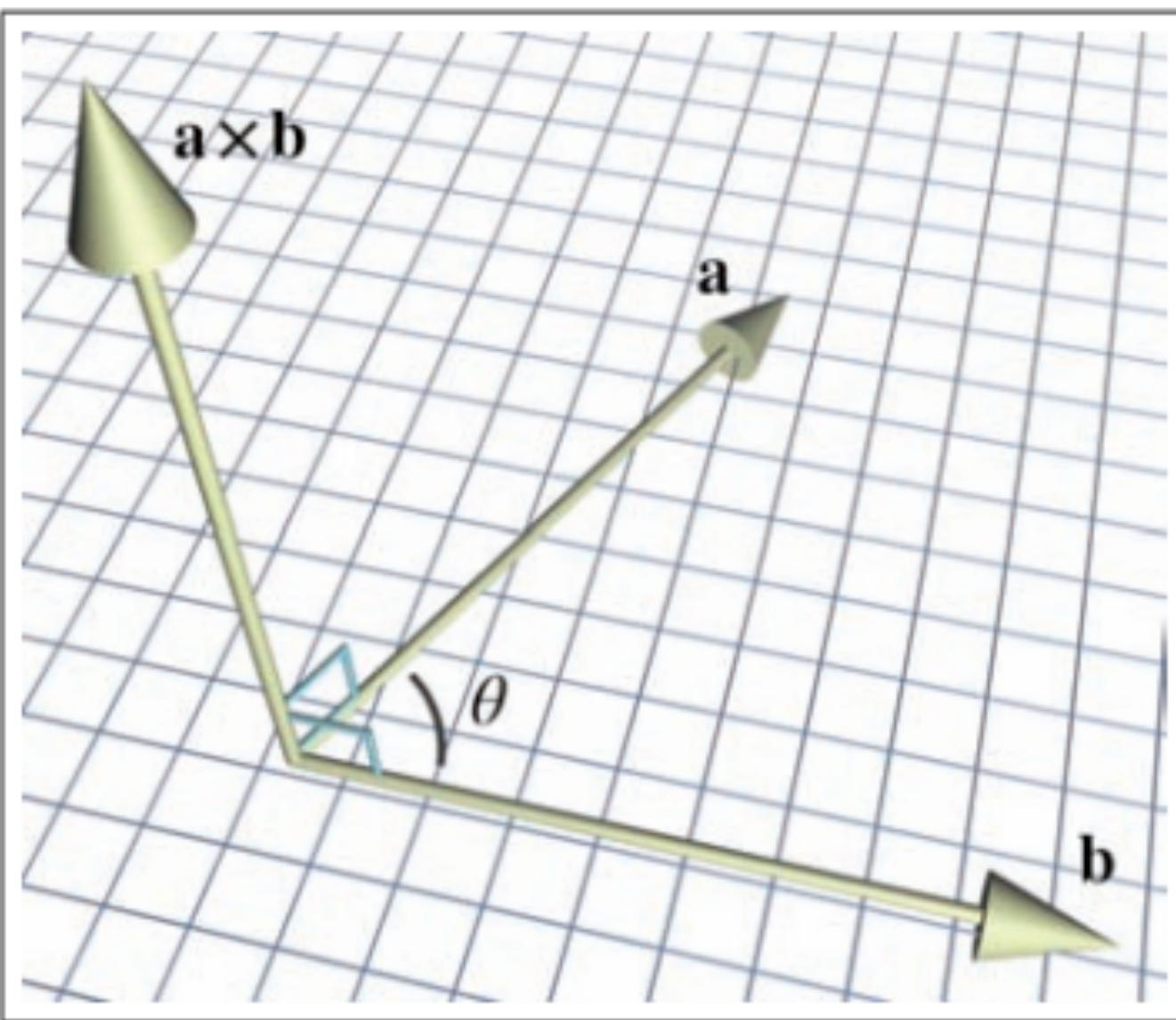
The cross product of two vectors is another vector orthogonal to the two (really useful property in 3D geometry!)

Geometric Interpretation



The length of the cross product of two vectors
is the area of the parallelogram spanned by the two

Geometric Interpretation



$$\|\mathbf{a} \times \mathbf{b}\| = \|\mathbf{a}\| \|\mathbf{b}\| \sin \theta_{ab}$$

Linear Algebra Identities

Identity	Comments
$\mathbf{a} + \mathbf{b} = \mathbf{b} + \mathbf{a}$	Commutative property of vector addition
$\mathbf{a} - \mathbf{b} = \mathbf{a} + (-\mathbf{b})$	Definition of vector subtraction
$(\mathbf{a} + \mathbf{b}) + \mathbf{c} = \mathbf{a} + (\mathbf{b} + \mathbf{c})$	Associative property of vector addition
$s(t\mathbf{a}) = (st)\mathbf{a}$	Associative property of scalar multiplication
$k(\mathbf{a} + \mathbf{b}) = k\mathbf{a} + k\mathbf{b}$	Scalar multiplication distributes over vector addition
$\ k\mathbf{a}\ = k \ \mathbf{a}\ $	Multiplying a vector by a scalar scales the magnitude by a factor equal to the absolute value of the scalar
$\ \mathbf{a}\ \geq 0$	The magnitude of a vector is nonnegative
$\ \mathbf{a}\ ^2 + \ \mathbf{b}\ ^2 = \ \mathbf{a} + \mathbf{b}\ ^2$	The Pythagorean theorem applied to vector addition.
$\ \mathbf{a}\ + \ \mathbf{b}\ \geq \ \mathbf{a} + \mathbf{b}\ $	Triangle rule of vector addition. (No side can be longer than the sum of the other two sides.)
$\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$	Commutative property of dot product
$\ \mathbf{a}\ = \sqrt{\mathbf{a} \cdot \mathbf{a}}$	Vector magnitude defined using dot product
$k(\mathbf{a} \cdot \mathbf{b}) = (k\mathbf{a}) \cdot \mathbf{b} = \mathbf{a} \cdot (k\mathbf{b})$	Associative property of scalar multiplication with dot product
$\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c}$	Dot product distributes over vector addition and subtraction
$\mathbf{a} \times \mathbf{a} = \mathbf{0}$	The cross product of any vector with itself is the zero vector. (Because any vector is parallel with itself.)
$\mathbf{a} \times \mathbf{b} = -(\mathbf{b} \times \mathbf{a})$	Cross product is anticommutative.
$\mathbf{a} \times \mathbf{b} = (-\mathbf{a}) \times (-\mathbf{b})$	Negating both operands to the cross product results in the same vector.
$k(\mathbf{a} \times \mathbf{b}) = (k\mathbf{a}) \times \mathbf{b} = \mathbf{a} \times (k\mathbf{b})$	Associative property of scalar multiplication with cross product.
$\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c}$	Cross product distributes over vector addition and subtraction.

Coming up...

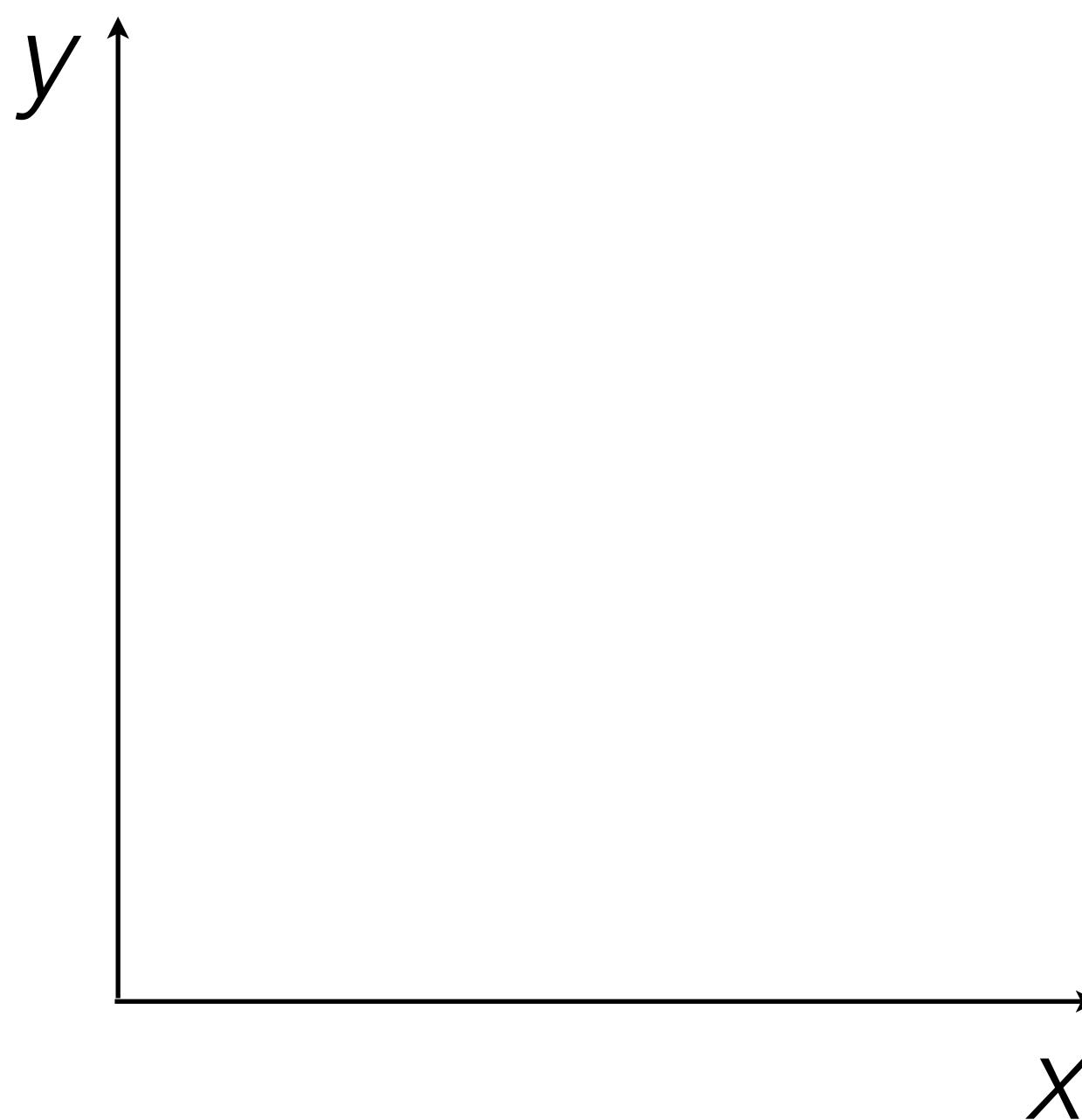
- Transformations



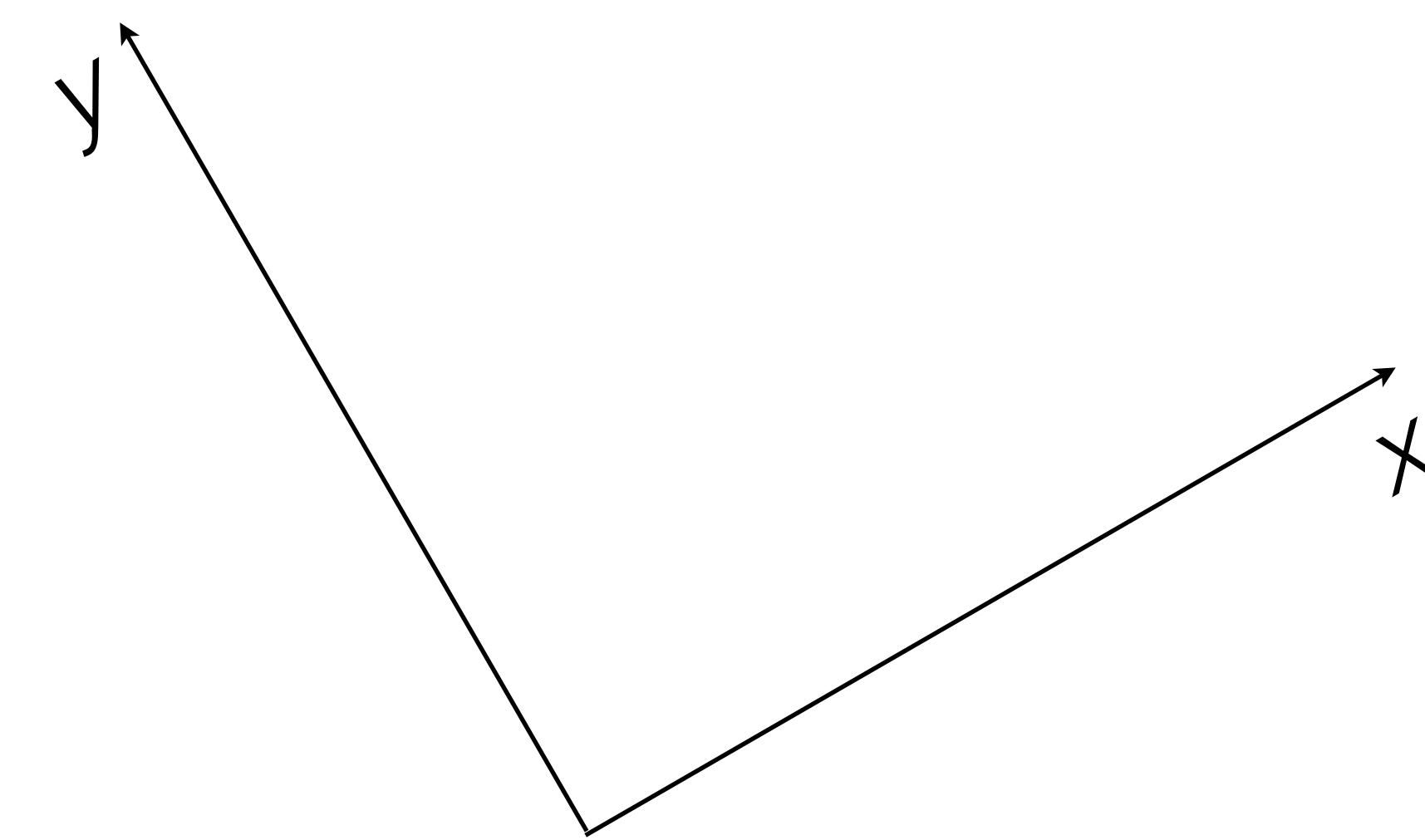
Introduction to Transformations & Review of Matrices

CS 355: Introduction to Graphics and Image Processing

Changes of Coordinates



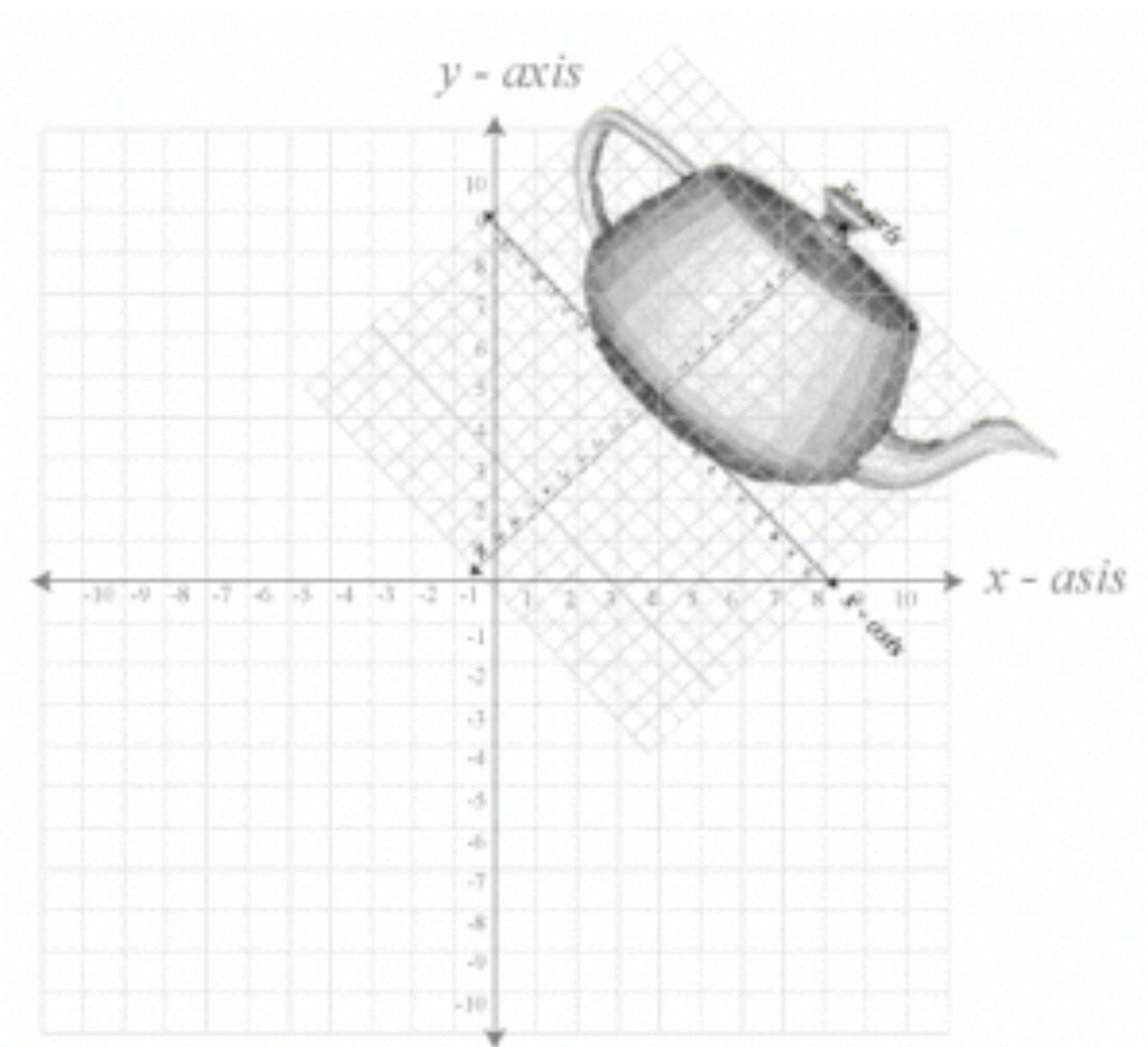
What if we have a point
described on one
coordinate system...



But we want to
describe it in another?

Why Change Coordinates?

- Moving stuff around (Lab 4)
- Model in one space, place in another (Lab 5)
- Modeling hierarchically (Lab 6)
- Where are 3D points *relative to the camera*?
(Lab 7, Lab 9)
- Geometric tests (Lab 9)
- Data transformations (Lab 10)



Translation

- *Translating* a coordinate system simply moves the origin
- Or can be thought of as moving the point
- Keeps the x and y directions the same
- Just add desired x, y offsets

$$(x', y') = (x + t_x, y + t_y)$$

OR

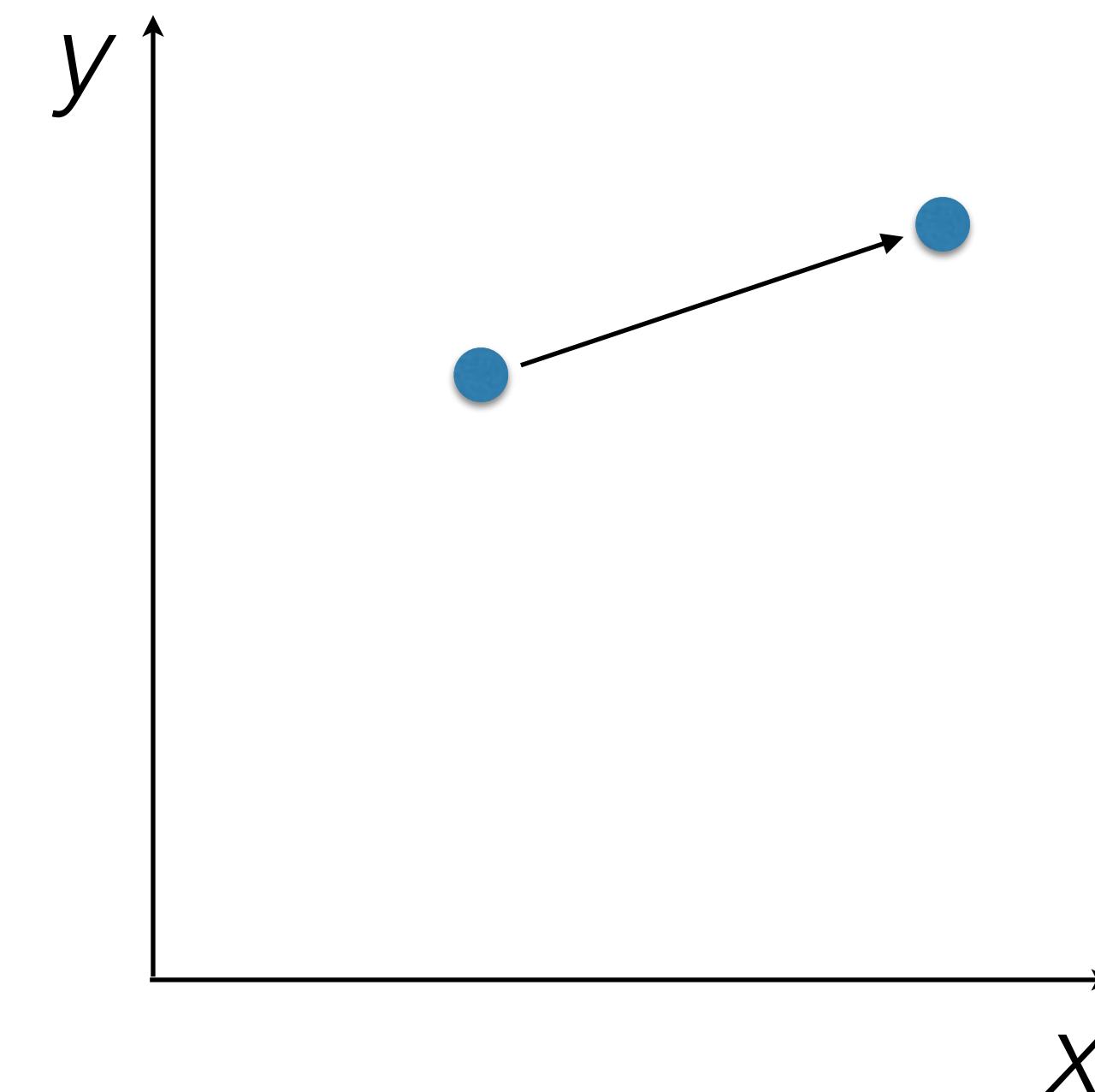
$$\mathbf{p}' = \mathbf{p} + \mathbf{t}$$

$$\mathbf{t} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

Example: Translation

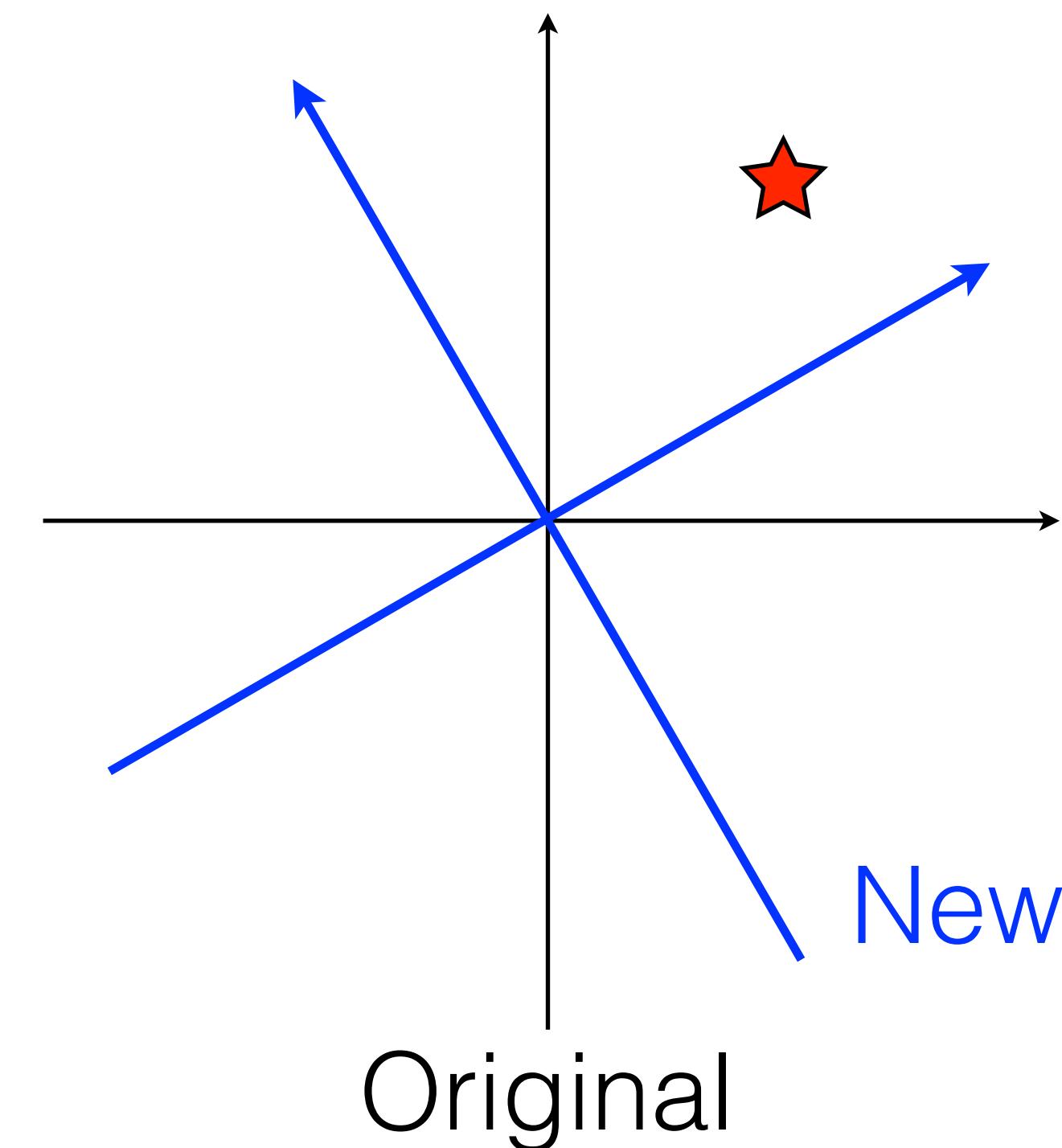
- Suppose that you have a point \mathbf{p} at $(100,200)$ and you want to translate it by $[160,50]$?

$$\begin{aligned}\mathbf{p}' &= \mathbf{p} + \mathbf{t} \\ &= (100, 200) + [160, 50] \\ &= (260, 250)\end{aligned}$$



Rotation

- Rotating a coordinate system keeps the origin, turns the axis directions
- Conceptually,
 - The point stays the same and the axes change
 - The axes stay the same and the point rotates around the origin

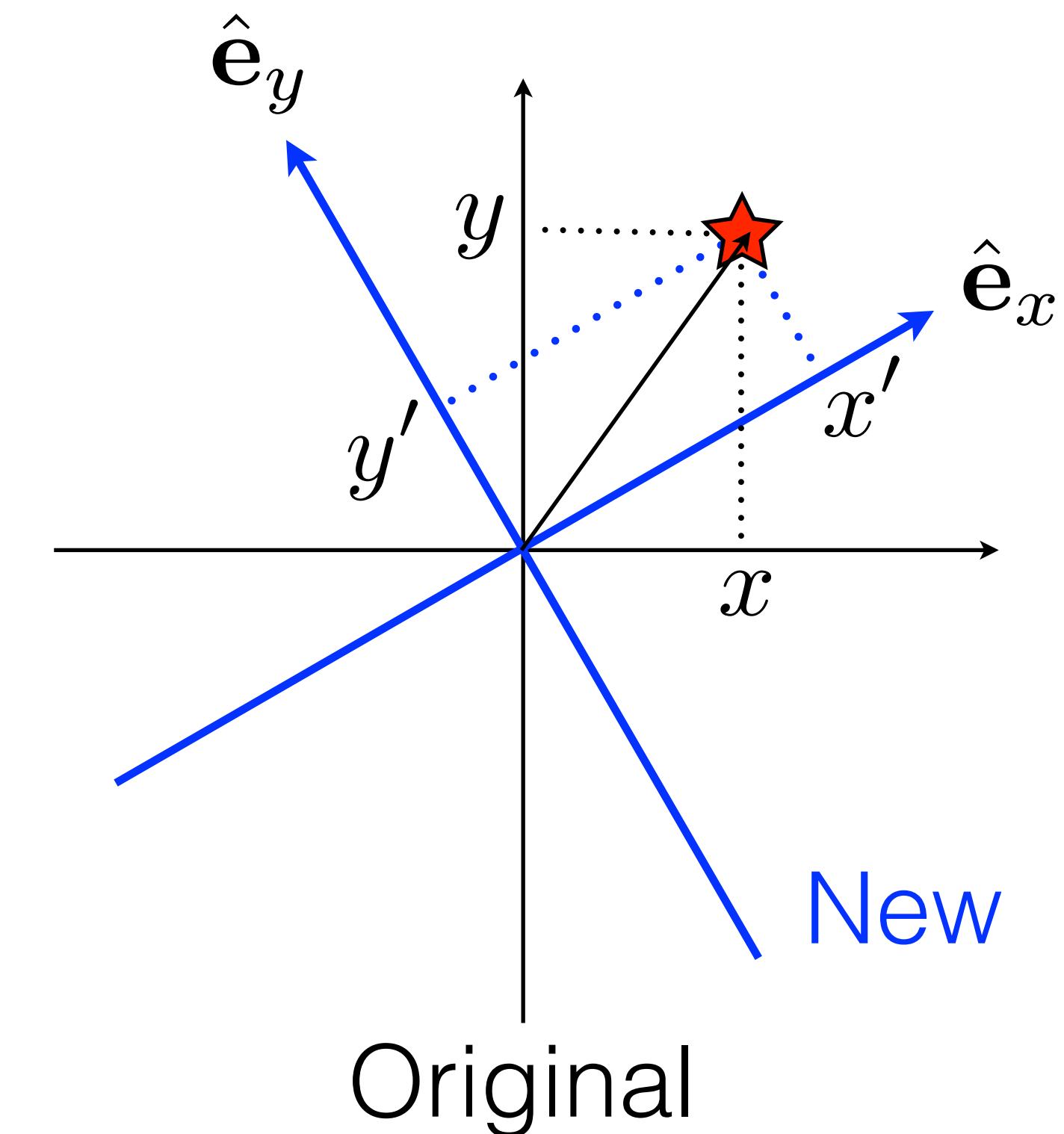


Computing Rotation

- To compute coordinates in the rotated system, just project to each of the new axis directions
- Use dot products!

$$p'_x = \mathbf{p} \cdot \hat{\mathbf{e}}_x$$

$$p'_y = \mathbf{p} \cdot \hat{\mathbf{e}}_y$$



More on transformations later,
but first let's review...

Matrices

$$\mathbf{M} = \begin{bmatrix} 3 & 1 & 8 & 5 \\ -1 & 4 & -3 & 3 \\ 2 & 0 & -1 & 4 \end{bmatrix}$$

A matrix is an n by m array of numbers

Notation

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

Index an array by row, then column

Square Matrices

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

A 3x3 matrix M is shown. The diagonal elements m₁₁, m₂₂, and m₃₃ are highlighted with a blue diagonal line. An arrow points from the word "Diagonal" to this line.

Square matrices have the same number
of rows as columns ($n = m$)

If everything off the diagonal is 0,
it is a diagonal matrix

Vectors as Matrices

$$\mathbf{v} = \begin{bmatrix} 2 \\ 3 \\ -1 \end{bmatrix}$$

A vector is simply an $n \times 1$ matrix

(technically, this is a *column vector*—some use *row vectors*)

Transposing

$$\mathbf{M} = \begin{bmatrix} 3 & 1 & 8 & 5 \\ -1 & 4 & -3 & 3 \\ 2 & 0 & -1 & 4 \end{bmatrix} \quad \mathbf{M}^T = \begin{bmatrix} 3 & -1 & 2 \\ 1 & 4 & 0 \\ 8 & -3 & -1 \\ 5 & 3 & 4 \end{bmatrix}$$

“M transpose”

The transposition of a matrix simply swaps
the rows for the columns

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

Stacks of Transposed Vectors

$$\mathbf{M} = \begin{bmatrix} 3 & 1 & 8 & 5 \\ -1 & 4 & -3 & 3 \\ 2 & 0 & -1 & 4 \end{bmatrix}$$

A matrix is an n by m array of numbers

OR a matrix is a stack of n transposed vectors,
each with m elements

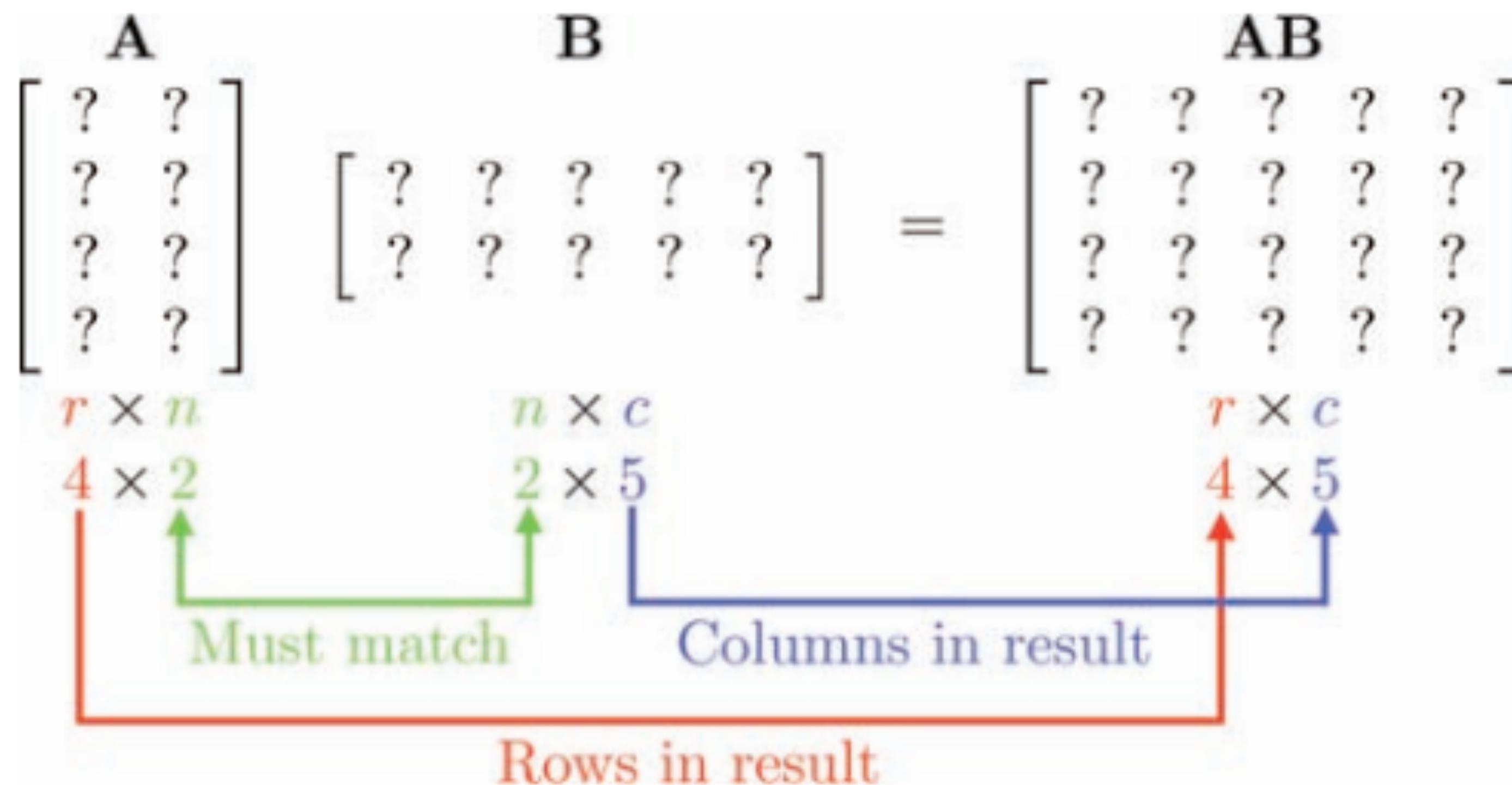
Multiplying by Scalar

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

$$k\mathbf{M} = k \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} = \begin{bmatrix} k m_{11} & k m_{12} & k m_{13} \\ k m_{21} & k m_{22} & k m_{23} \\ k m_{31} & k m_{32} & k m_{33} \end{bmatrix}$$

Multiply a matrix by a scalar
multiplies each element accordingly

Matrix Multiplication



Width of first must match height of second

Matrix Multiplication

$$\mathbf{C} = \mathbf{AB}$$

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

$$\begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} & c_{15} \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} \\ c_{41} & c_{42} & c_{43} & c_{44} & c_{45} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \end{bmatrix}$$
$$c_{24} = a_{21}b_{14} + a_{22}b_{24}$$



Look, a dot product!

Alternate View

$$\mathbf{C} = \mathbf{AB}$$

$$\begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \end{bmatrix}$$
$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} \quad \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} & c_{15} \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} \\ c_{41} & c_{42} & c_{43} & c_{44} & c_{45} \end{bmatrix}$$

$$c_{43} = a_{41}b_{13} + a_{42}b_{23}$$

$$c_{ij} = \mathbf{A}.\text{row}[i] \cdot \mathbf{B}.\text{col}[j]$$

Identity Matrix

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

$$\mathbf{M}\mathbf{I} = \mathbf{I}\mathbf{M} = \mathbf{M}$$

Matrix Inversion

The inverse of a matrix is the matrix such that

$$\mathbf{M}\mathbf{M}^{-1} = \mathbf{M}^{-1}\mathbf{M} = \mathbf{I}$$

↑
inverse

There are multiple ways to compute the inverse of a matrix, but we won't cover that here

Matrix Multiplication

- Matrix multiplication is associative $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$
- And distributes over addition $\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC}$
- Is NOT commutative, but... $(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T$
- And... $(\mathbf{AB})^{-1} = \mathbf{B}^{-1} \mathbf{A}^{-1}$

Multiplying by Vector

$$\mathbf{b} = \mathbf{M} \mathbf{a}$$

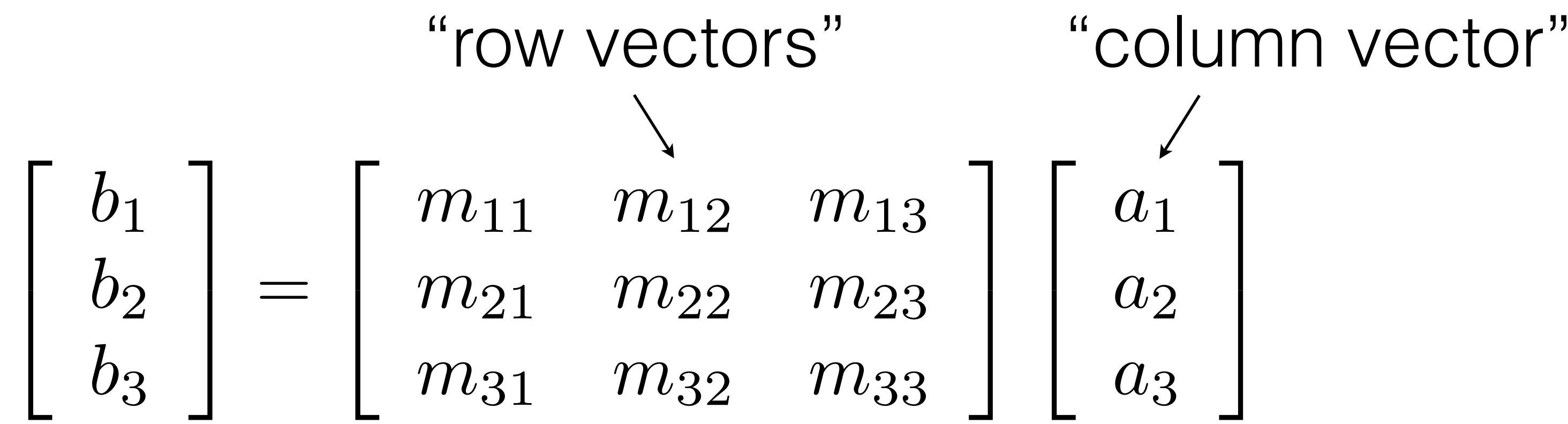
$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Multiplying a vector by a matrix is just a compact way of writing a bunch of dot products

Row vs. Column

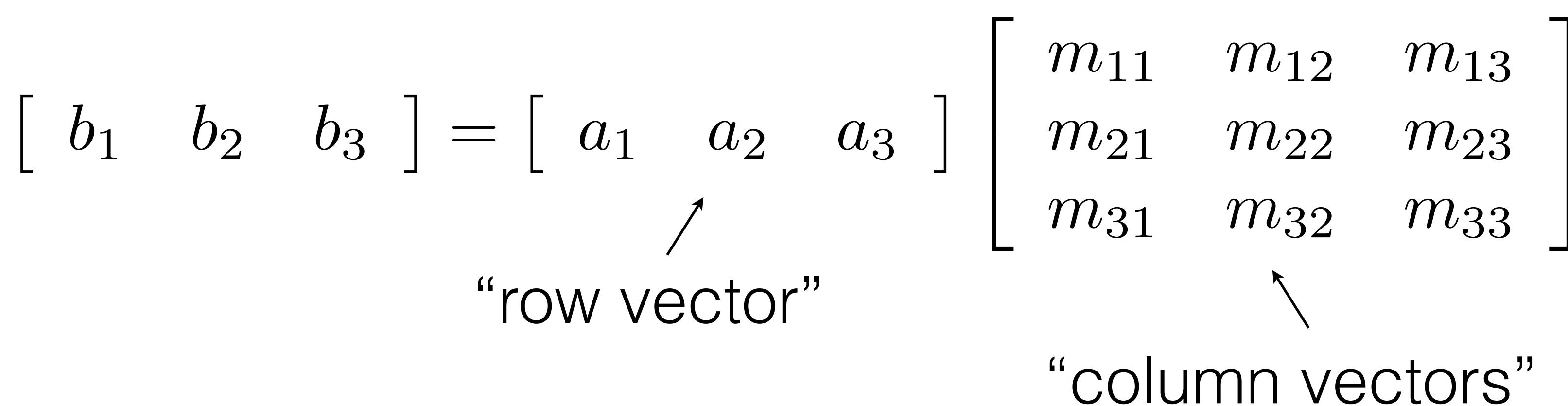
$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

“row vectors” “column vector”



$$\begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix} \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

“row vector” “column vectors”



Row vs. Column

- Column vectors:
 - Most common in math and science
 - Used in most scientific computing code
 - Used in many graphics libraries
 - Writes a little more compactly
 - Read right-to-left:
- Row vectors:
 - Used many programmers and graphics books
 - Used in many graphics libraries
 - Much less compact to write
 - Read left-to-right:

$$\mathbf{C}\mathbf{B}\mathbf{A}\mathbf{v} = \mathbf{C}(\mathbf{B}(\mathbf{A}\mathbf{v}))$$

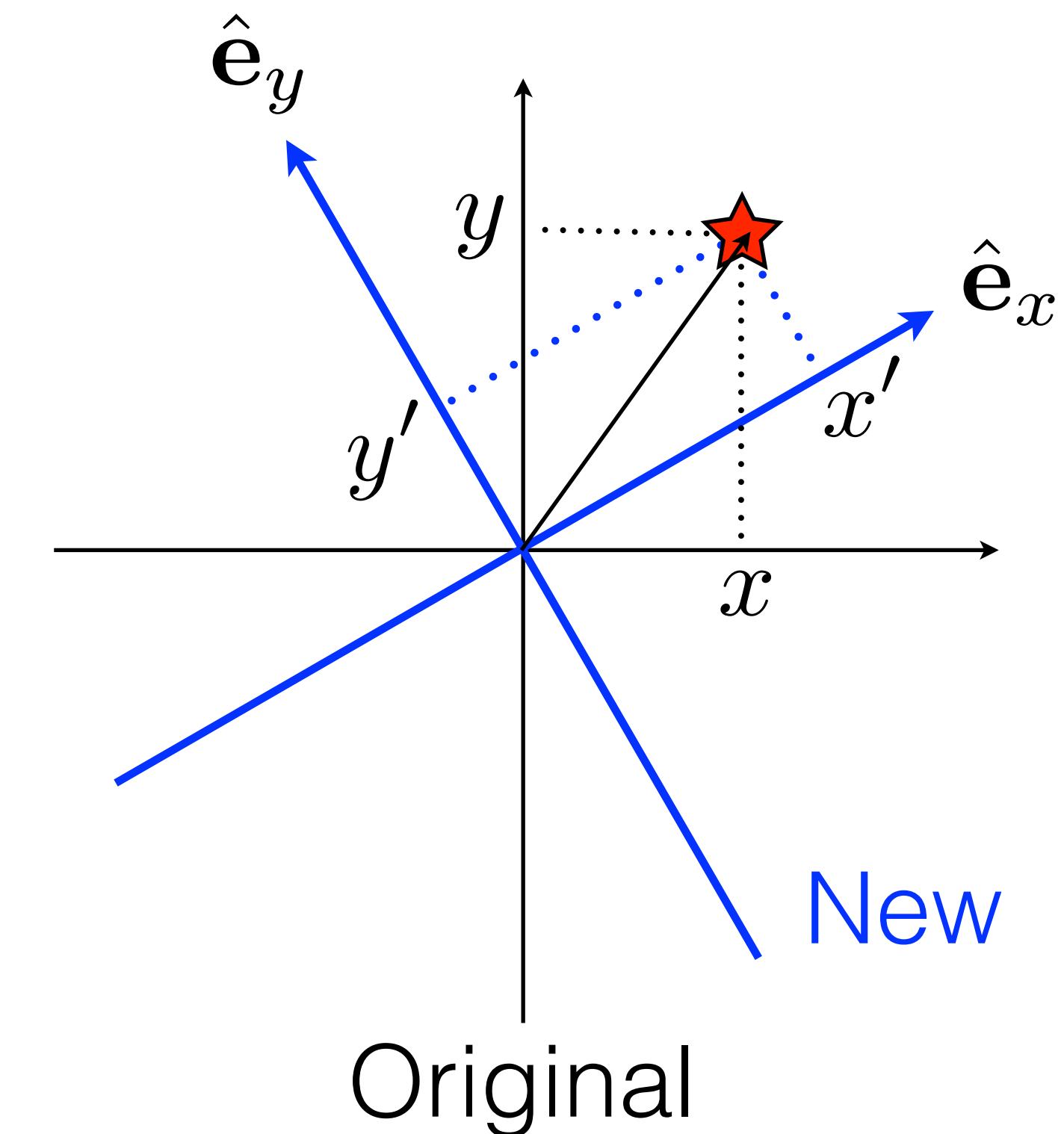
$$\mathbf{v}\mathbf{A}\mathbf{B}\mathbf{C} = (((\mathbf{v}\mathbf{A})\mathbf{B})\mathbf{C})$$

Computing Rotation

- To compute coordinates in the rotated system, just project to each of the new axis directions
- Use dot products!

$$p'_x = \mathbf{p} \cdot \hat{\mathbf{e}}_x$$

$$p'_y = \mathbf{p} \cdot \hat{\mathbf{e}}_y$$

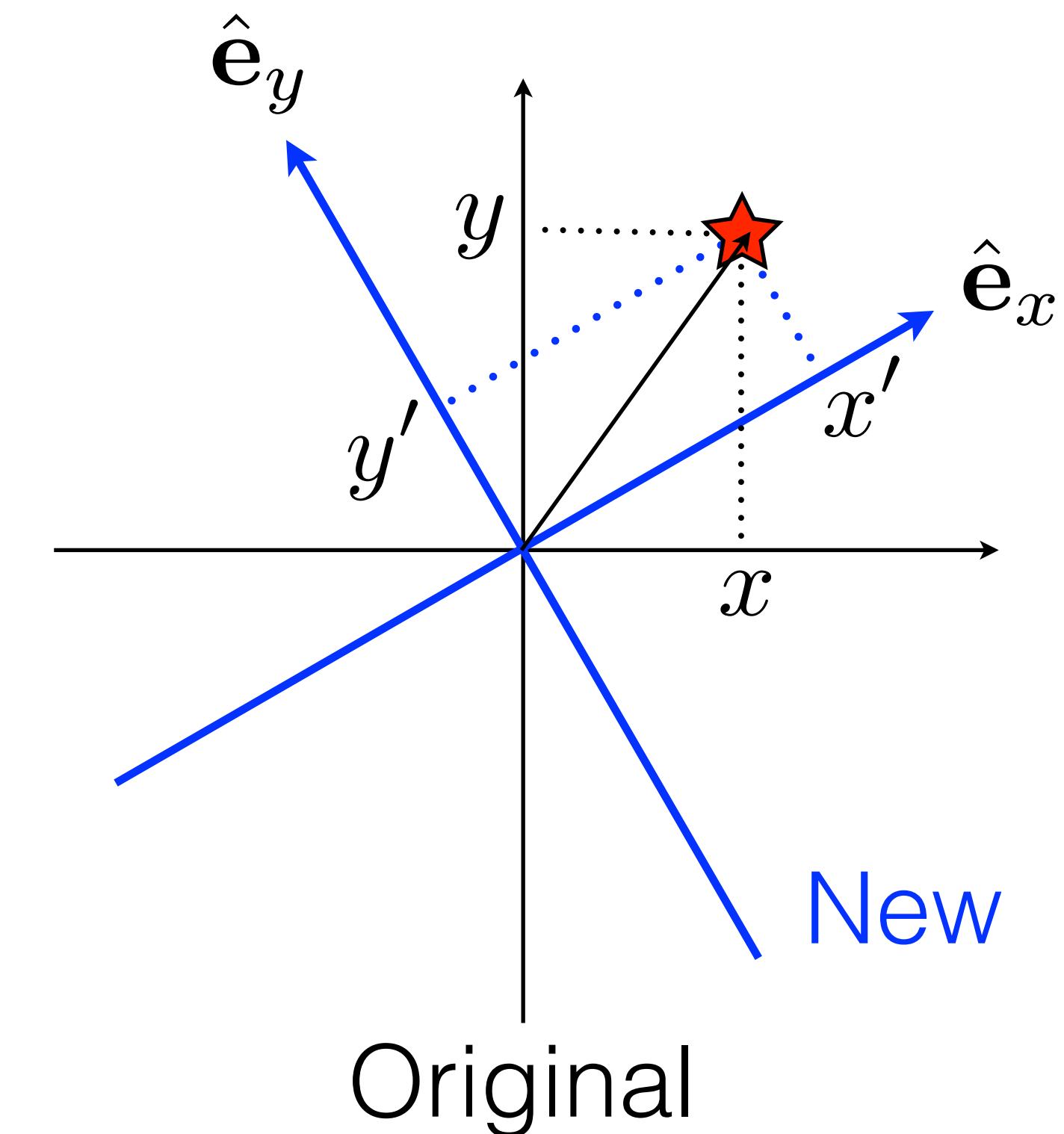


Computing Rotation

- To compute coordinates in the rotated system, just project to each of the new axis directions
- Use dot products!

$$\mathbf{p}' = \begin{bmatrix} e_{x1} & e_{x2} \\ e_{y1} & e_{y2} \end{bmatrix} \mathbf{p}$$

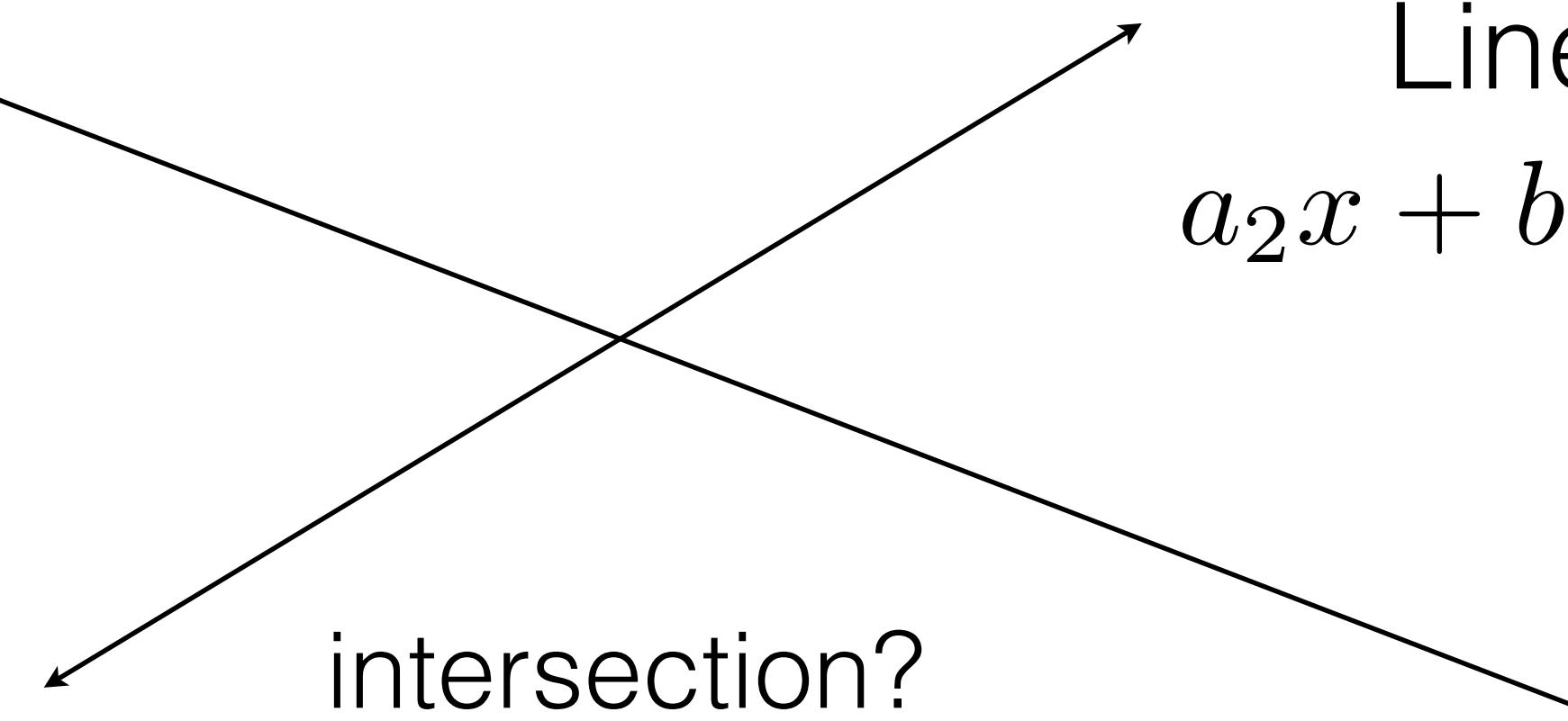
But do it with a matrix!!



More Applications

Line 1:
 $a_1x + b_1y = d_1$

Line 2:
 $a_2x + b_2y = d_2$



$$\begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \end{bmatrix}$$

This is a *system of linear equations*

Ways to solve these are covered in Math 313,
but we'll use code in Python when we have to

Coming up...

- Linear (matrix) transformations
- Homogeneous coordinates



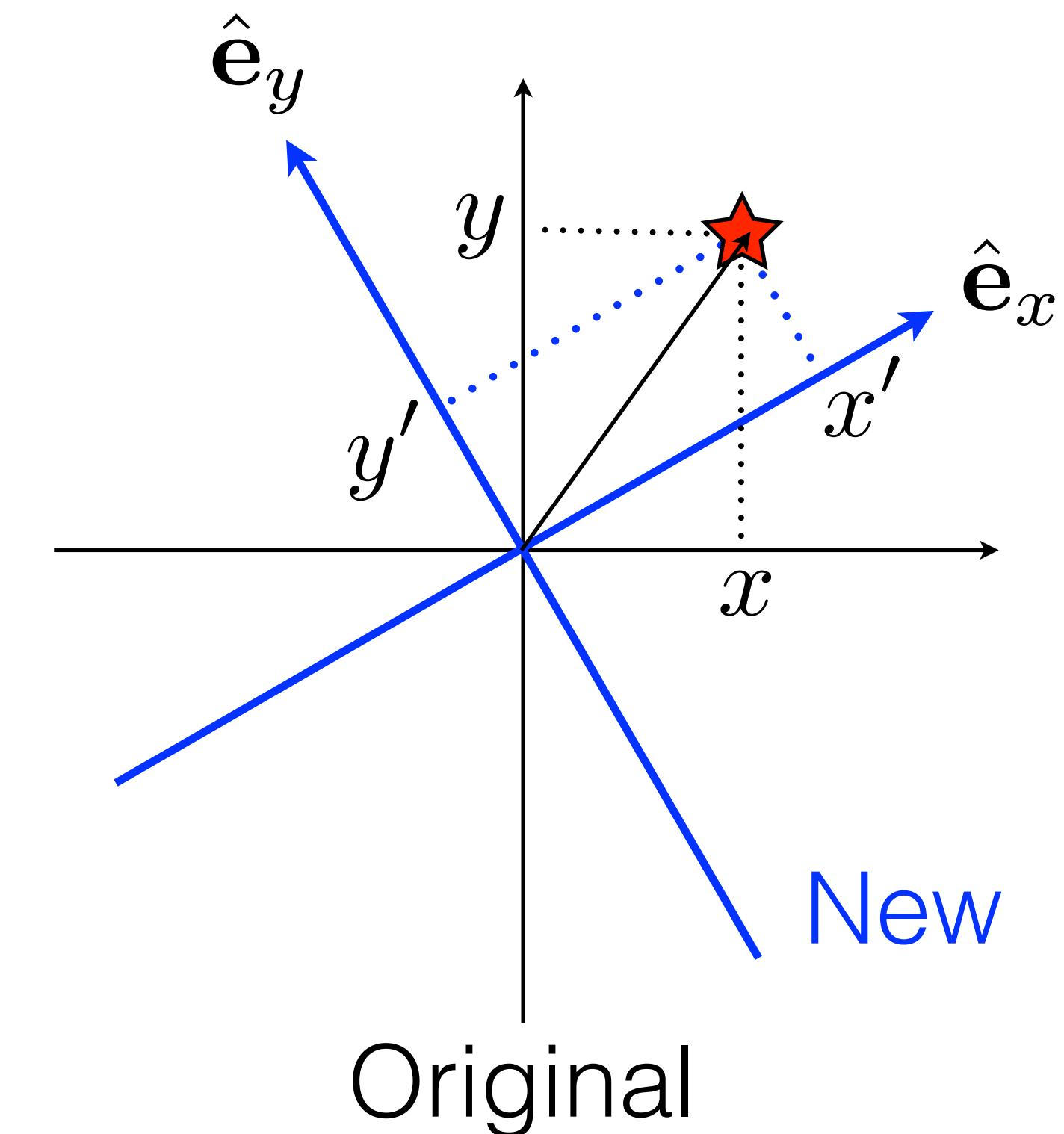
Linear Transformations

CS 355: Introduction to Graphics and Image Processing

Change of Coordinates

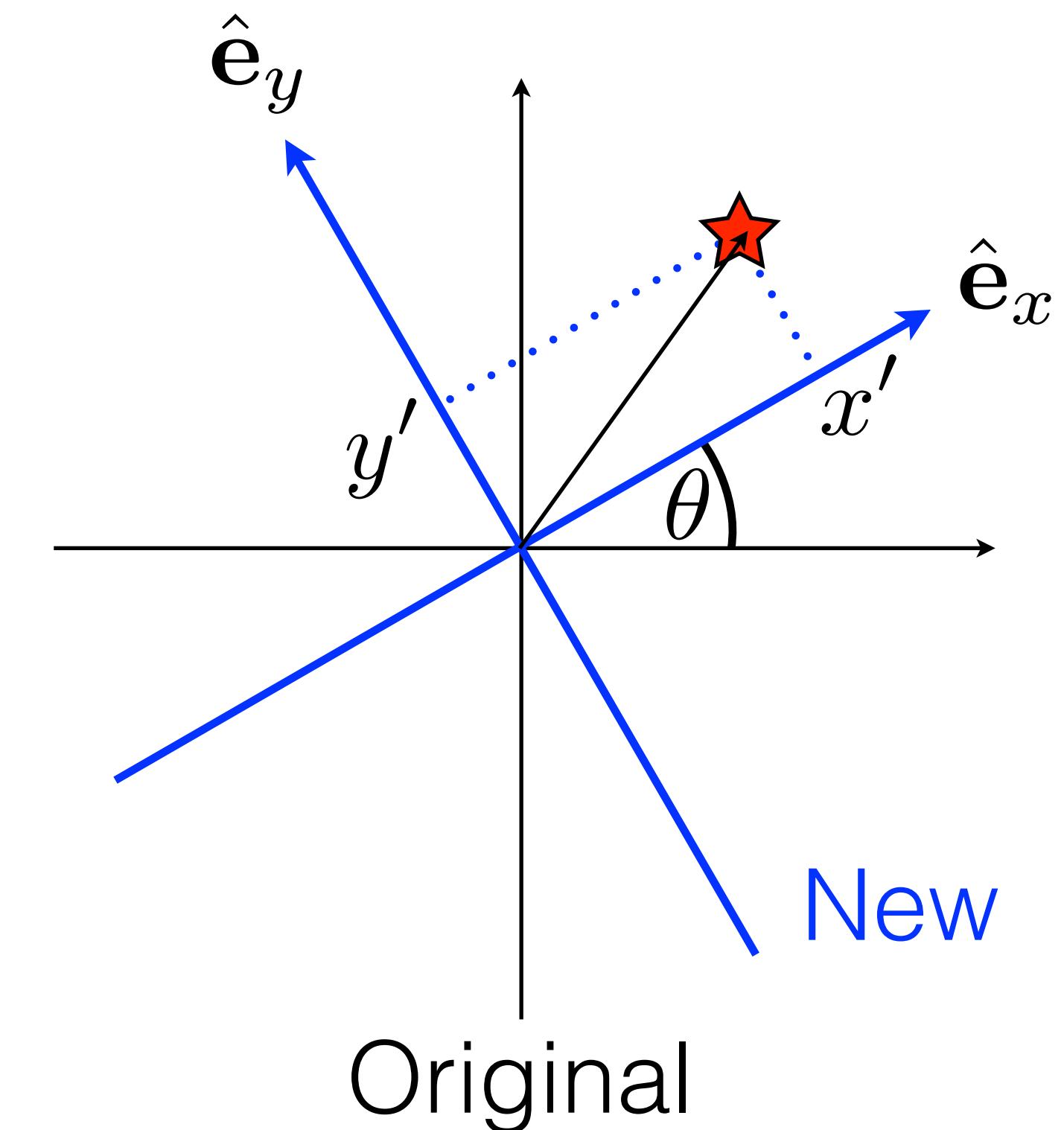
- To compute coordinates in the rotated system, just project to each of the new axis directions
- Use a matrix to do the dot products

$$\mathbf{p}' = \begin{bmatrix} e_{x1} & e_{x2} \\ e_{y1} & e_{y2} \end{bmatrix} \mathbf{p}$$



Rotation

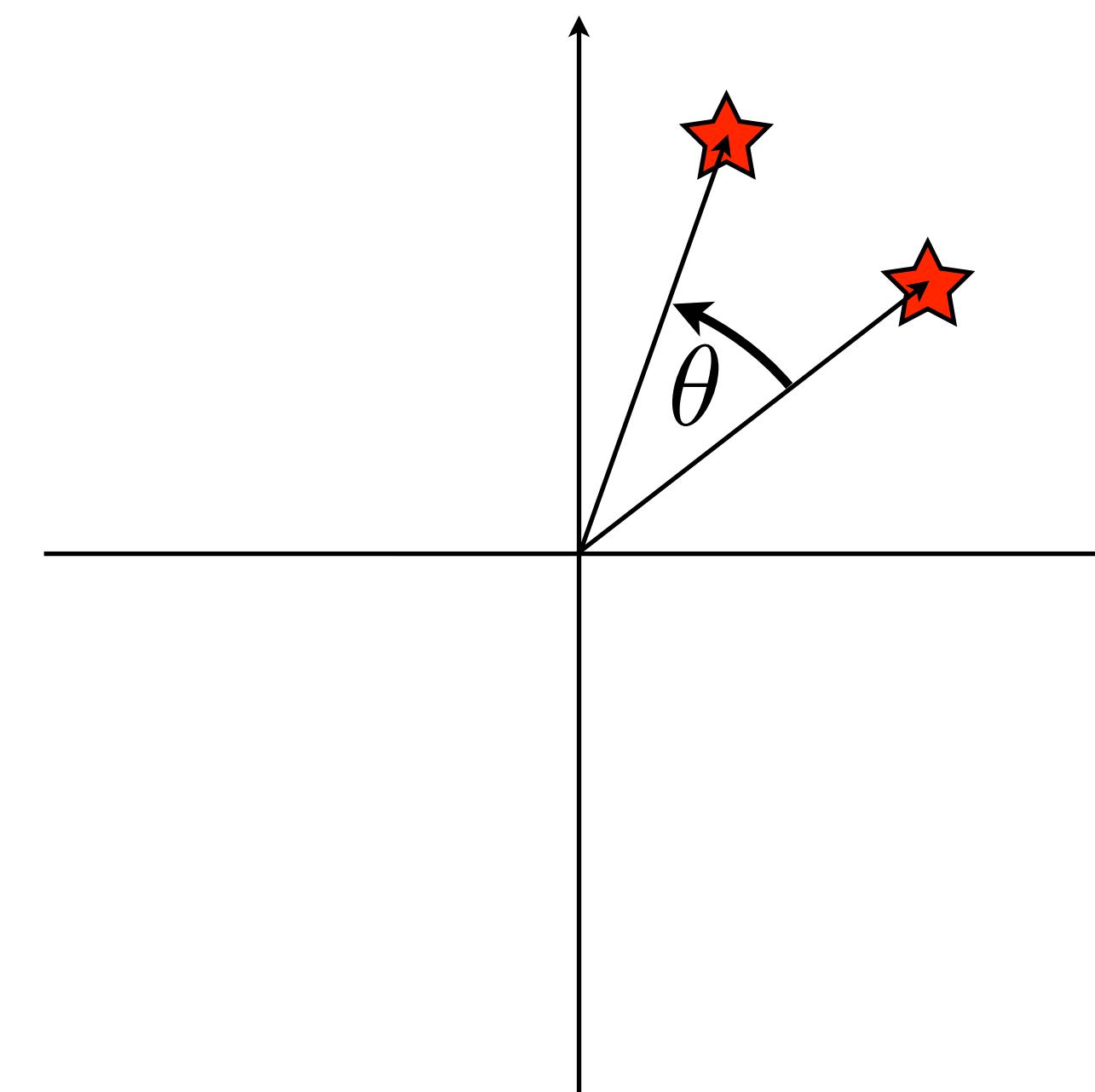
- Rotating the coordinate system one way rotates the point/object the other way
- From here on we'll talk in terms of rotating the point or object



Rotation

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

$$\mathbf{R}^{-1}(\theta) = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix}$$

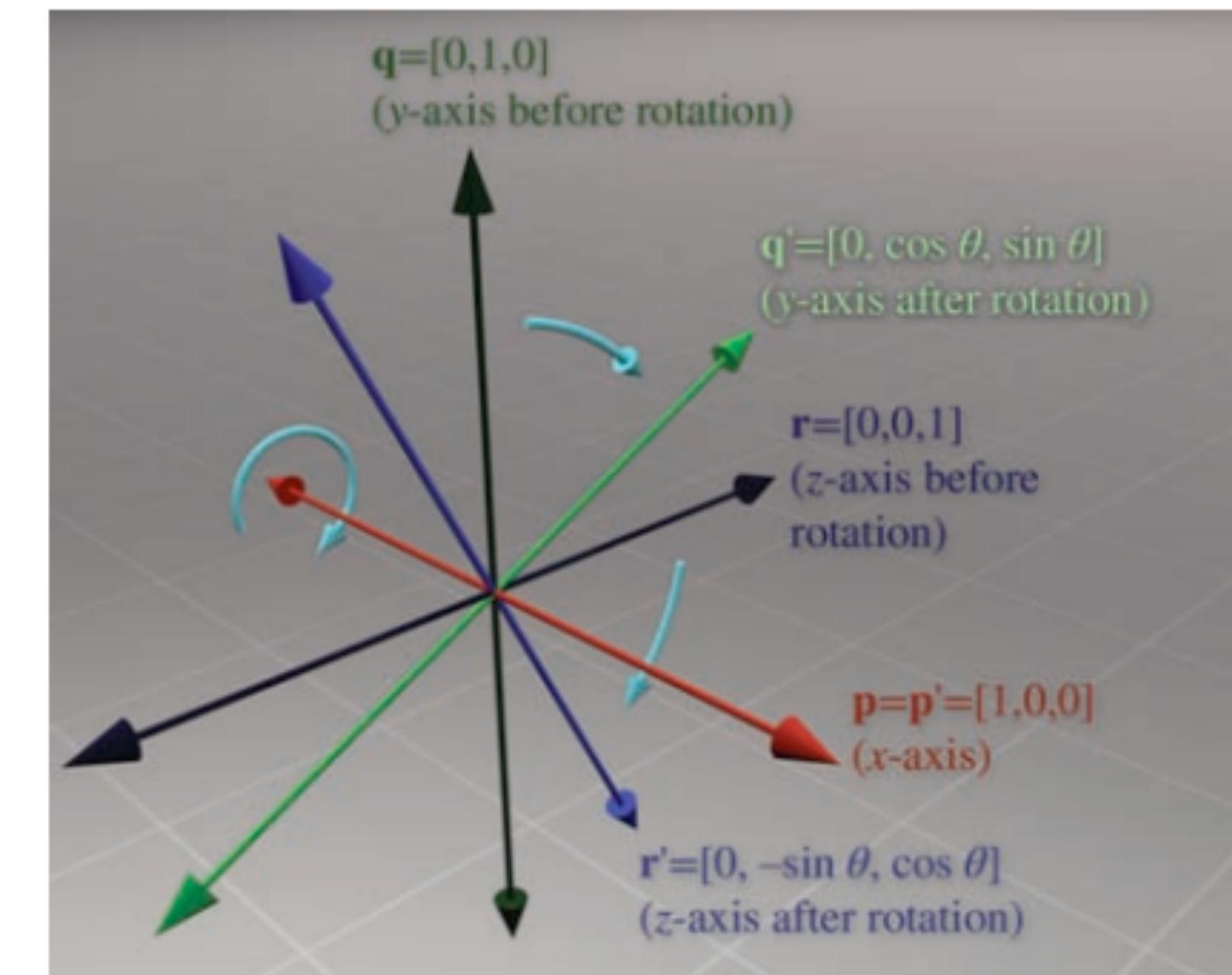


Rotation in 3D

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

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

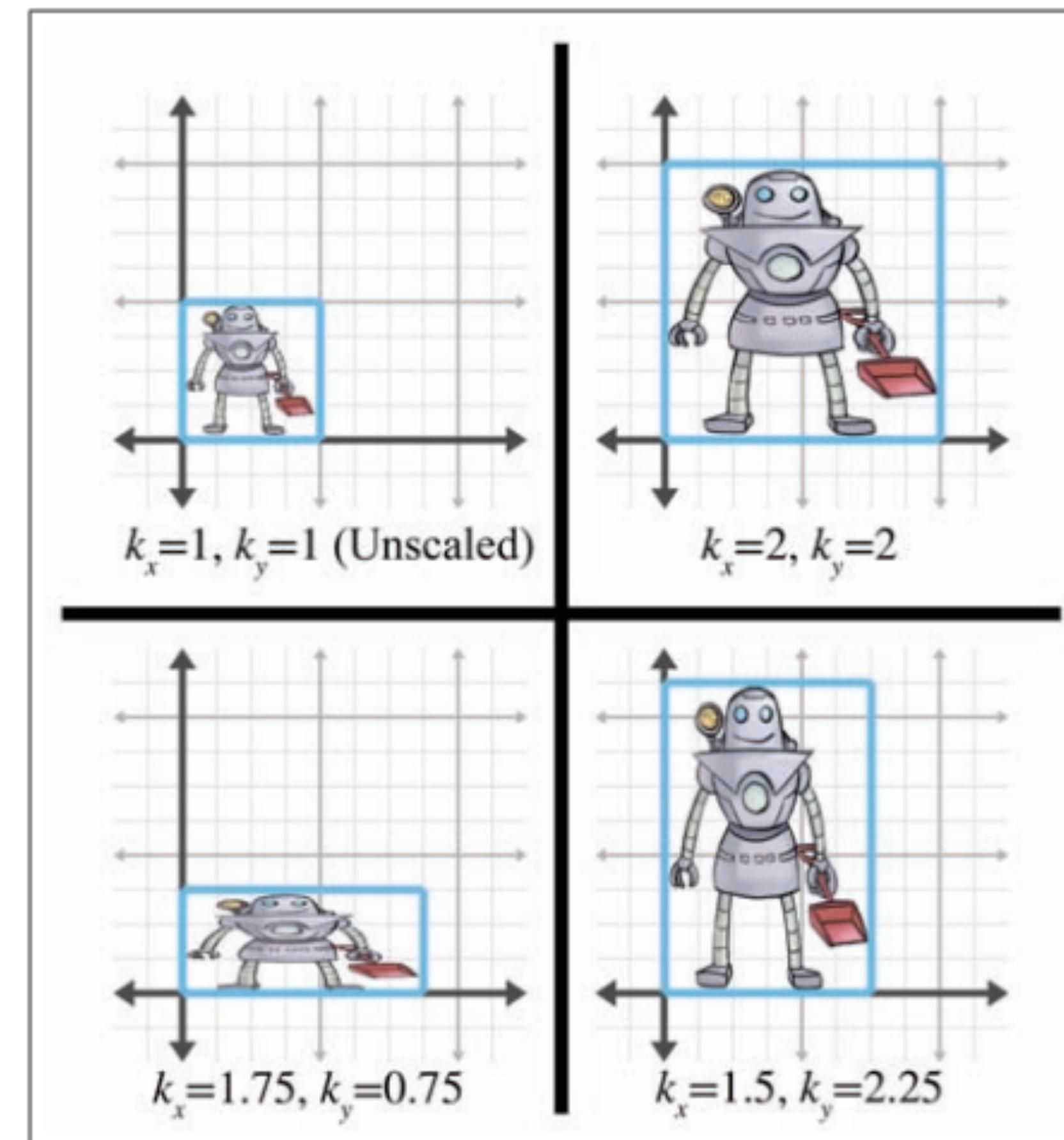
$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



Scaling

$$\begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} sx \\ sy \end{bmatrix}$$

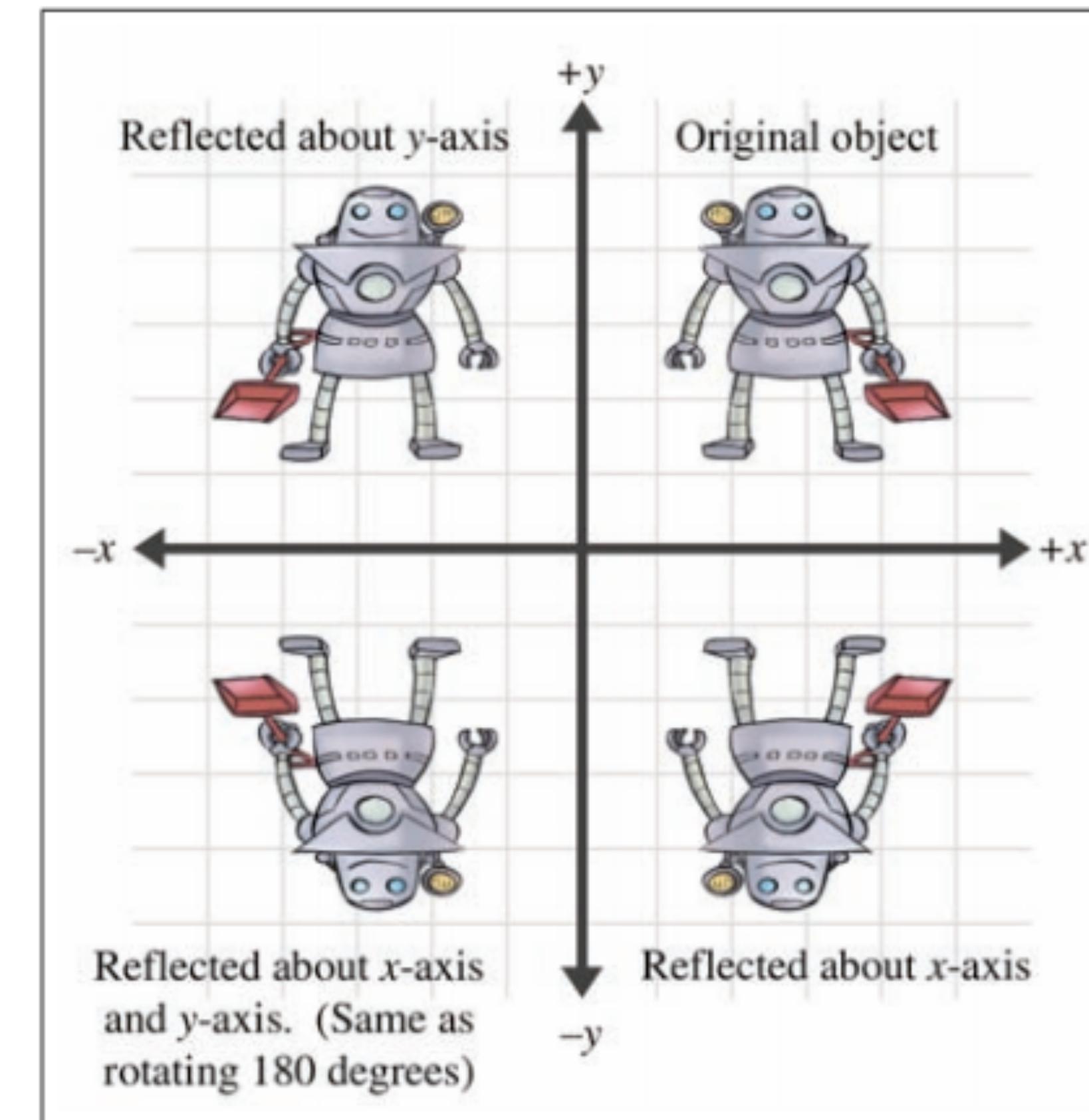
$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \end{bmatrix}$$



Reflection

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

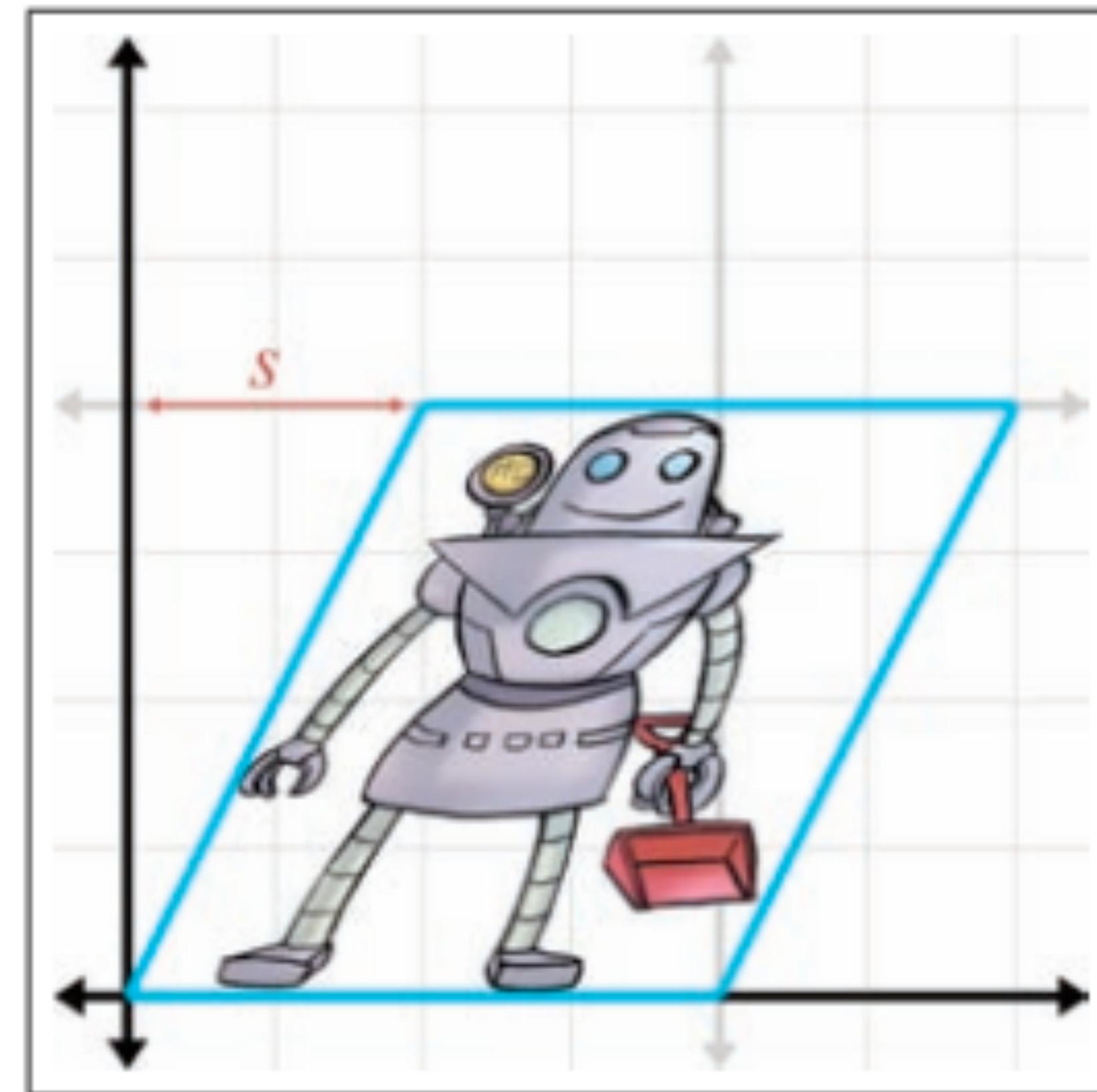
$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ -y \end{bmatrix}$$



Shearing

$$\begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + sy \\ y \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ sx + y \end{bmatrix}$$



Sometimes called a skew transform

Composition

$$\begin{aligned} \mathbf{M}_2 &= \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} & \mathbf{M}_1 &= \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \\ &\searrow &&\downarrow \\ \mathbf{M}_{12} &= \mathbf{M}_2 \mathbf{M}_1 &&\longleftarrow \end{aligned}$$

$$\mathbf{M}_{12}\mathbf{p} = (\mathbf{M}_2\mathbf{M}_1)\mathbf{p} = \mathbf{M}_2(\mathbf{M}_1\mathbf{p})$$

But...

- Composition of transformations (matrix multiplication) is really useful
- But translation must be done as a separate operation
- *Can you include translation as a linear transformation?*

Homogeneous Coordinates

- Can include translation as a linear transformation using *homogeneous coordinates*
- Also useful for perspective (covered later)

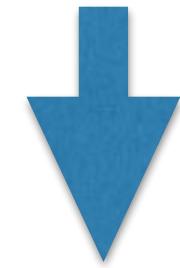
$$\mathbf{p} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Add additional “homogeneous” element

Homogeneous Matrices

- To make a matrix work with homogeneous coordinates, add an extra row and column
- To do nothing else, make these $[0 \ 0 \ 1]$

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix}$$



$$\begin{bmatrix} m_{11} & m_{12} & 0 \\ m_{21} & m_{22} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Translation

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ 1 \end{bmatrix}$$

(General note: for row vectors, transpose this)

Combining With Translation

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} t_x \\ t_y \\ 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11}x + m_{12}y + t_x \\ m_{21}x + m_{22}y + t_y \\ 1 \end{bmatrix}$$

Always $[0 \dots 0 1]$
(for now)

Other operation

Translation

The diagram illustrates the combination of a linear transformation and a translation. It shows a matrix multiplication where the first two columns of the matrix and the first two entries of the vector are grouped by a blue bracket labeled "Other operation". The third column of the matrix and the third entry of the vector are grouped by a red bracket labeled "Translation". Below the equation, it is noted that the third column of the matrix is always [0 ... 0 1] (for now).

Other operation *then* translation

Rotation then Translation

Rotation Translation

$$\begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \\ 0 & 0 \end{bmatrix} \begin{bmatrix} t_x \\ t_y \\ 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$p' = \mathbf{R}(\theta) \mathbf{p} + \mathbf{t}$$

Rotation then Translation

Rotation Translation



$$\begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix}$$

$$R = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \quad t = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

Points vs. Vectors

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

allows translation

$$\mathbf{v} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}$$

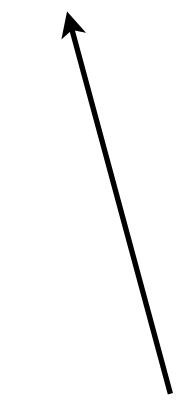
ignores translation

Remember: points have a position and *can* be translated,
but vectors are only a displacement and *cannot* be translated

Classes of Transformations

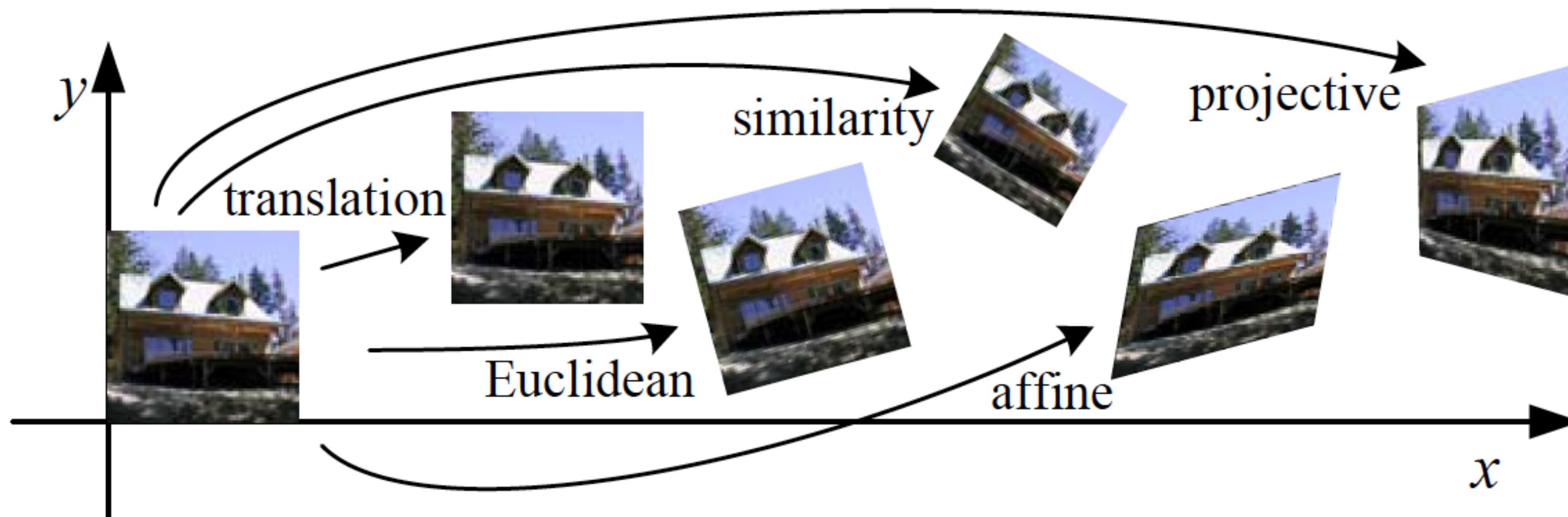
- Euclidean / Rigid body (preserves lengths):
Rotation and translation
- Similarity (preserves angles):
Rotation, translation, scale
- Affine (preserves straight lines):
Arbitrary linear combinations

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{bmatrix}$$

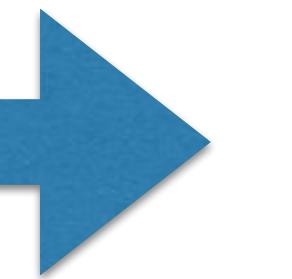
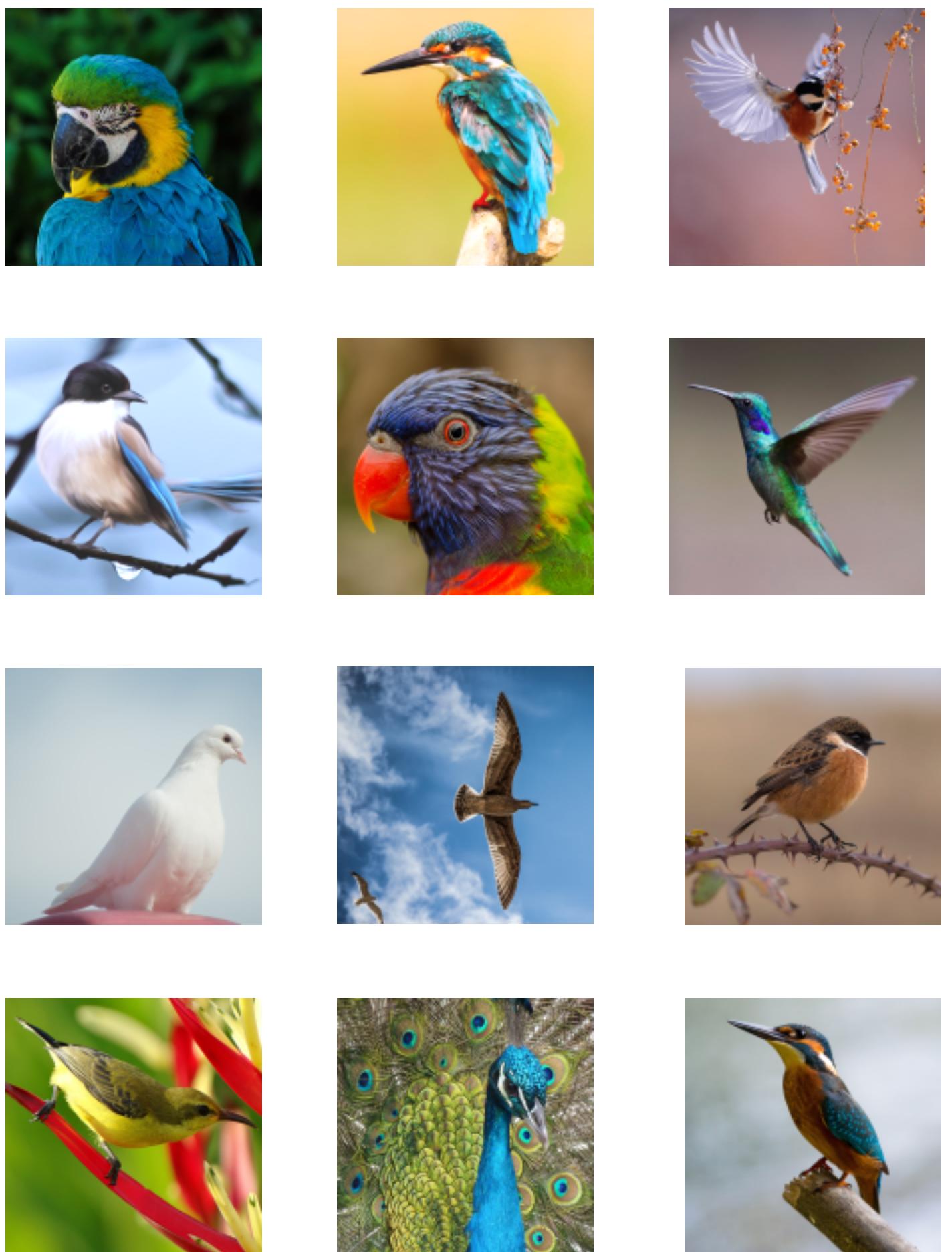


General form of an
affine transformation

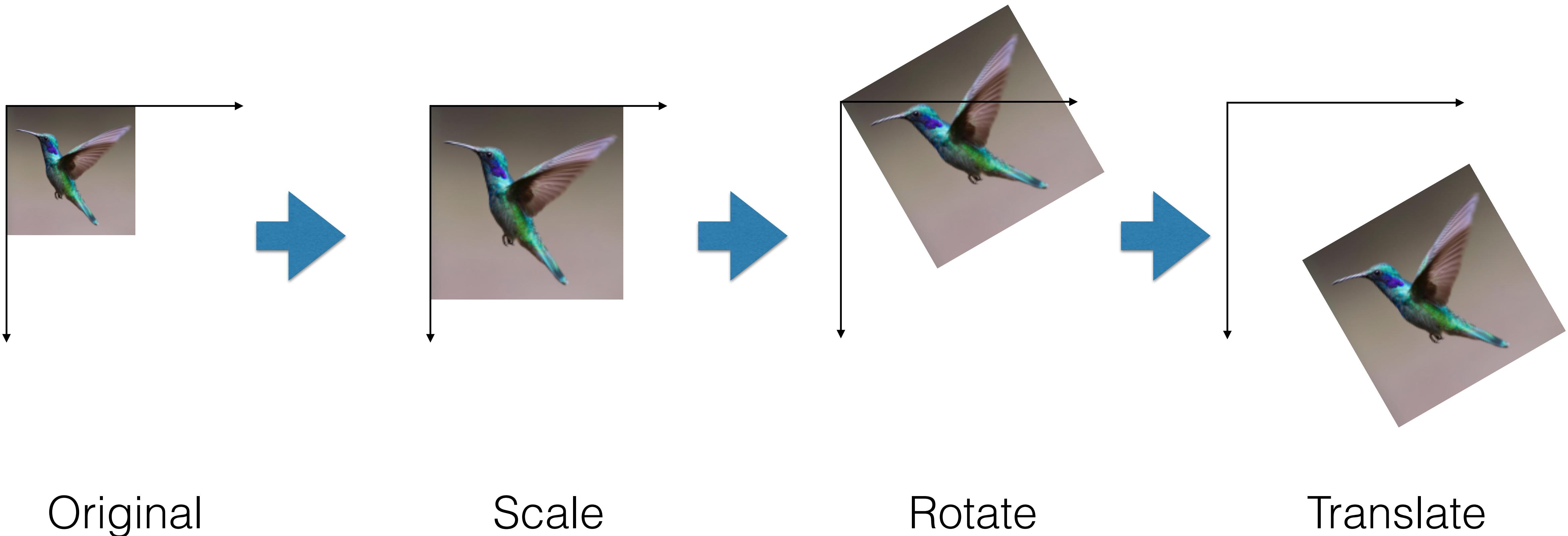
Classes of Transformations



Lab 4



Lab 4



Coming up...

- A bit more on matrices as transformations, then...
- 3D Graphics!



More on Matrices as Transformations

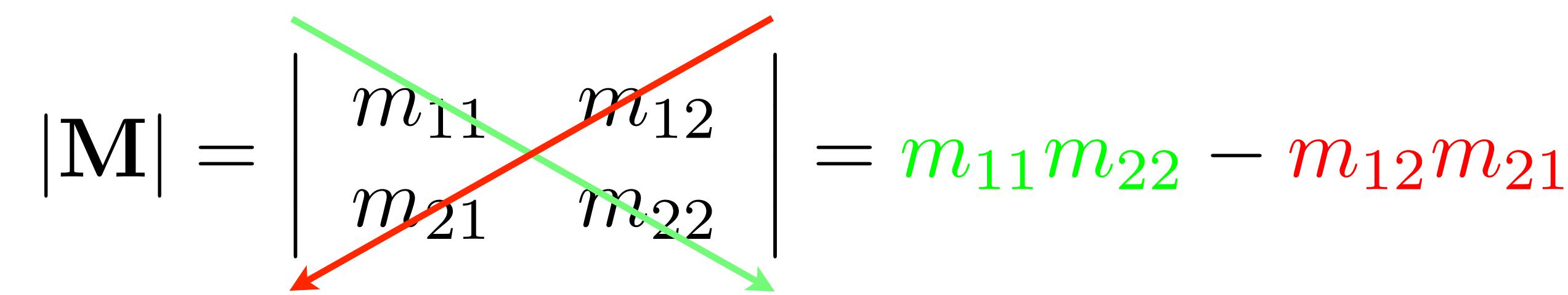
CS 355: Introduction to Graphics and Image Processing

A little more on matrices as
coordinate transformations...

Determinant

$$|\mathbf{M}| = \begin{vmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{vmatrix} = m_{11}m_{22} - m_{12}m_{21}$$

Determinant

$$|\mathbf{M}| = \begin{vmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{vmatrix} = m_{11}m_{22} - m_{12}m_{21}$$


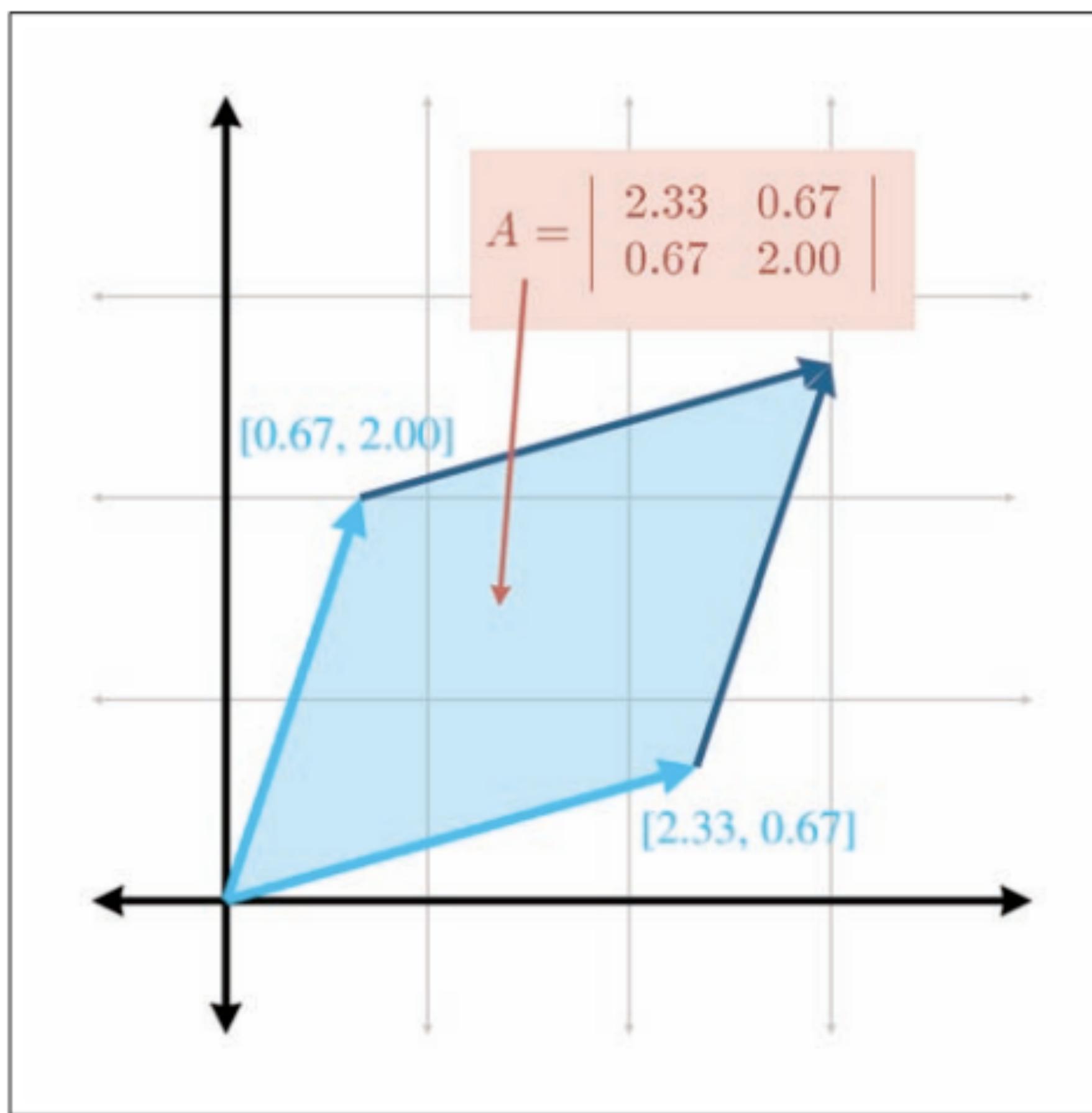
Determinant

$$|\mathbf{M}| = \begin{vmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{vmatrix}$$



$$\begin{aligned} & m_{11}m_{22}m_{33} + m_{12}m_{23}m_{31} + m_{13}m_{21}m_{32} \\ & - m_{13}m_{22}m_{31} - m_{12}m_{21}m_{33} - m_{11}m_{23}m_{32} \end{aligned}$$

Geometric Interpretation



Properties of Determinants

$$|\mathbf{I}| = 1$$

$$|\mathbf{AB}| = |\mathbf{A}| |\mathbf{B}|$$

$$|\mathbf{M}^T| = |\mathbf{M}|$$

$$|\mathbf{M}^{-1}| = \frac{1}{|\mathbf{M}|}$$

Linear Independence

A set of vectors is said to be *linearly dependent* if at least one of them can be expressed as a linear combination (weighted sum) of the others:

$$\mathbf{v}_j = \sum_{i \neq j} w_i \mathbf{v}_i$$

If not linearly *dependent*, then linearly *independent*

Singular Matrices

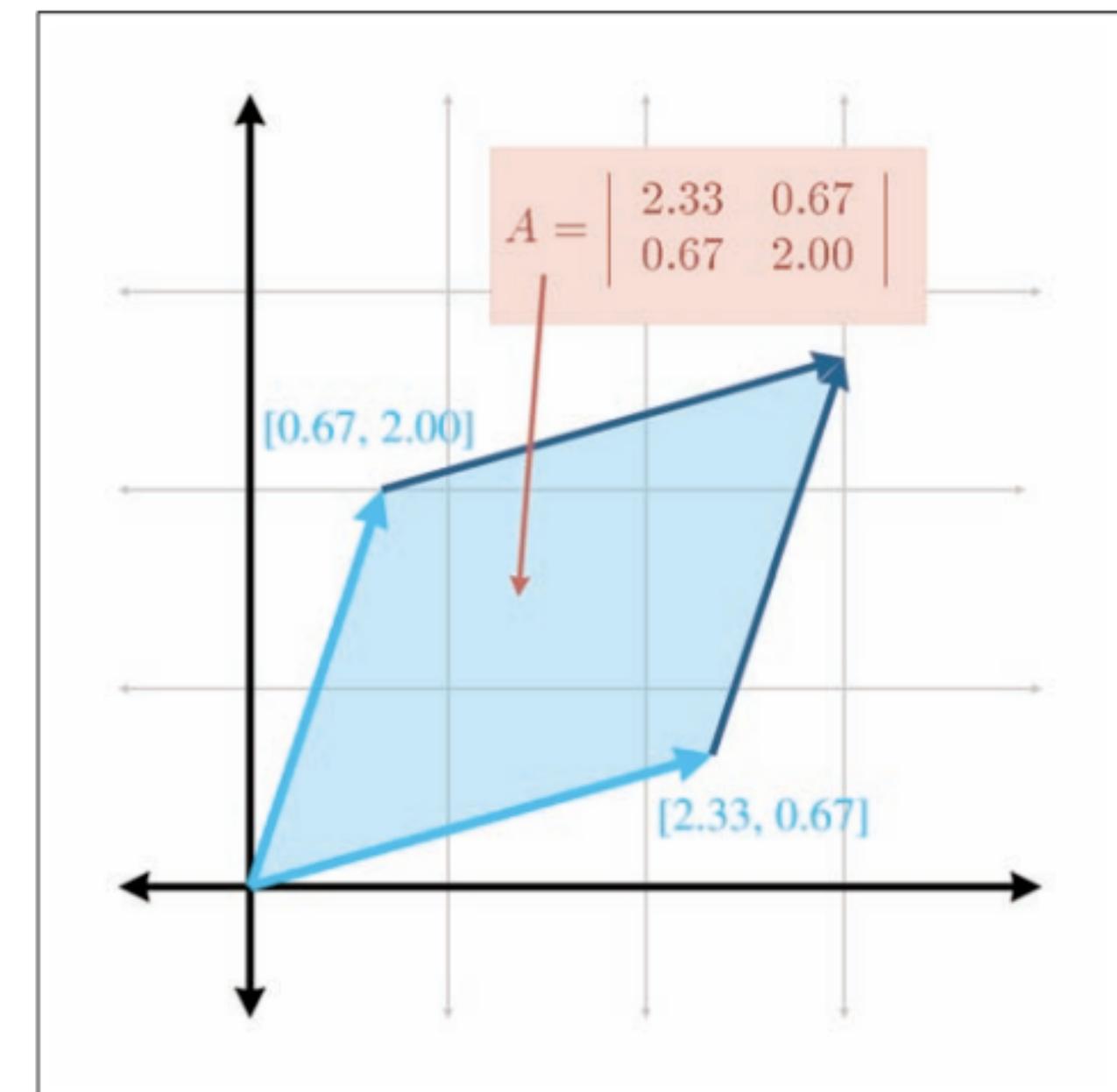
$$|\mathbf{M}^{-1}| = \frac{1}{|\mathbf{M}|}$$

But what if $|\mathbf{M}| = 0$?

A matrix whose determinant is zero
has no inverse and is said to be *singular*

Singular Matrices

- What does a singular matrix mean geometrically?
- *The rows are linearly dependent*



Matrix Rank

- The *rank* of a matrix is the number of linearly independent rows
- When used as transforms, matrices with *full rank* transform to the full space
- Singular matrices have *insufficient rank* and collapse to a corresponding subspace
- Geometric interpretation: the rank of a matrix is the dimensionality of the (sub)space that matrix maps to

Orthogonal Matrices

- Two (square) matrices are said to be orthogonal iff

$$\mathbf{M}\mathbf{M}^T = \mathbf{I}$$

- Implies rows are orthonormal vectors

Orthogonal Matrices

- Orthogonal matrices are also easily invertible:

$$\mathbf{M}^{-1} = \mathbf{M}^T$$

- Implies

$$|\mathbf{M}| = |\mathbf{M}^{-1}| = 1$$

Orthogonal Matrices

- All rotation matrices are orthogonal

AND

- All orthogonal matrices are rotations!

Coming up...

- 3D Graphics!



Introduction to 3D Graphics

CS 355: Introduction to Graphics and Image Processing

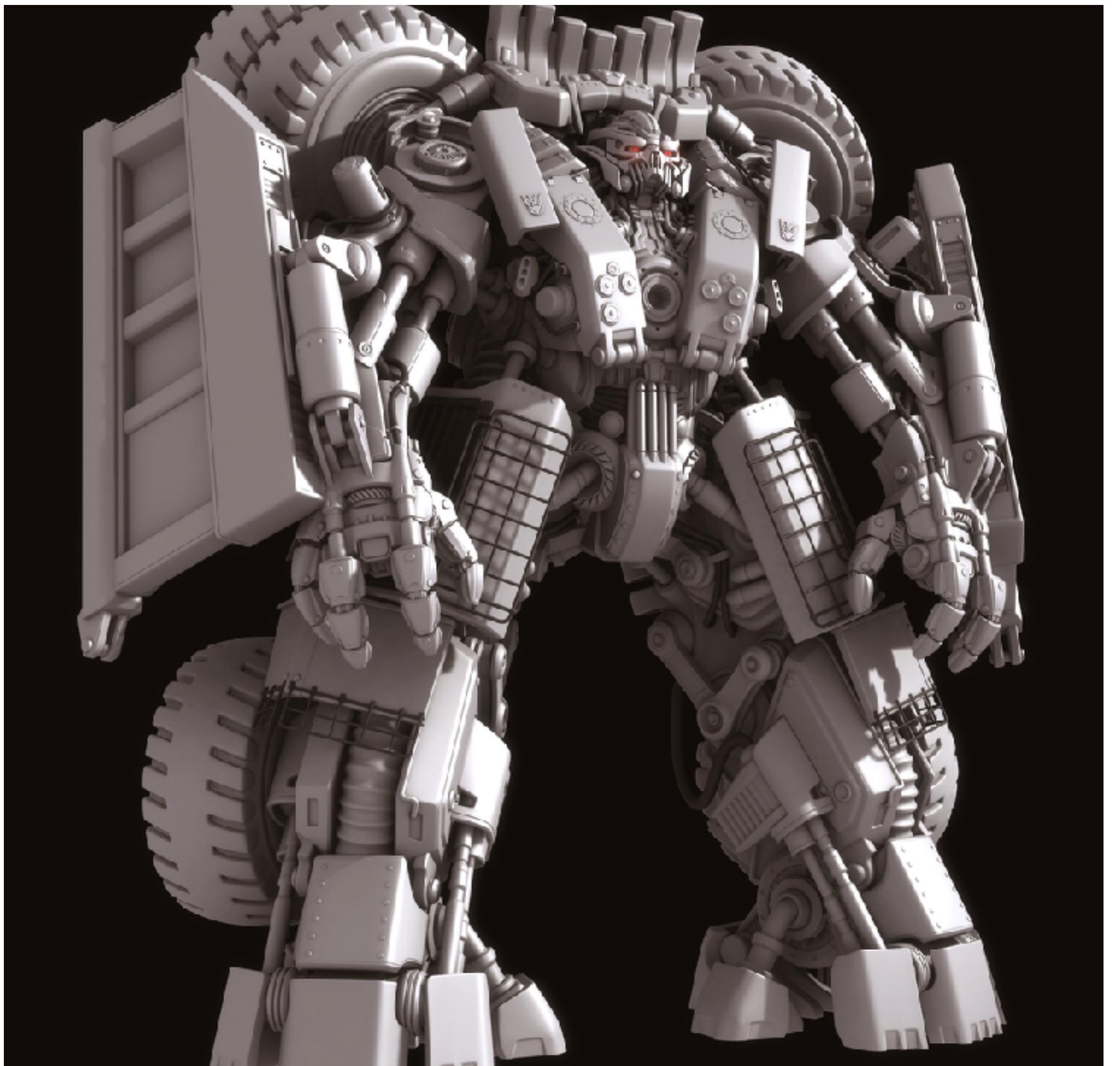
3D Rendering

- If I took a picture of a scene with a virtual camera:
 - What point in 3D is visible at each 2D point in the projected image?
 - What color is the light coming from that point as it reaches the camera?



It All Starts With Models

- Can be quite complex
- Ways to create:
 - By hand
(interactive software)
 - Scanning
(existing model, sculpting)
 - Image-based
(from photographs)
- Time intensive = \$\$\$



3D Animation

- Modeling
- Rigging
- Animation
- Physical Simulation
- Lighting
- Rendering
- Compositing

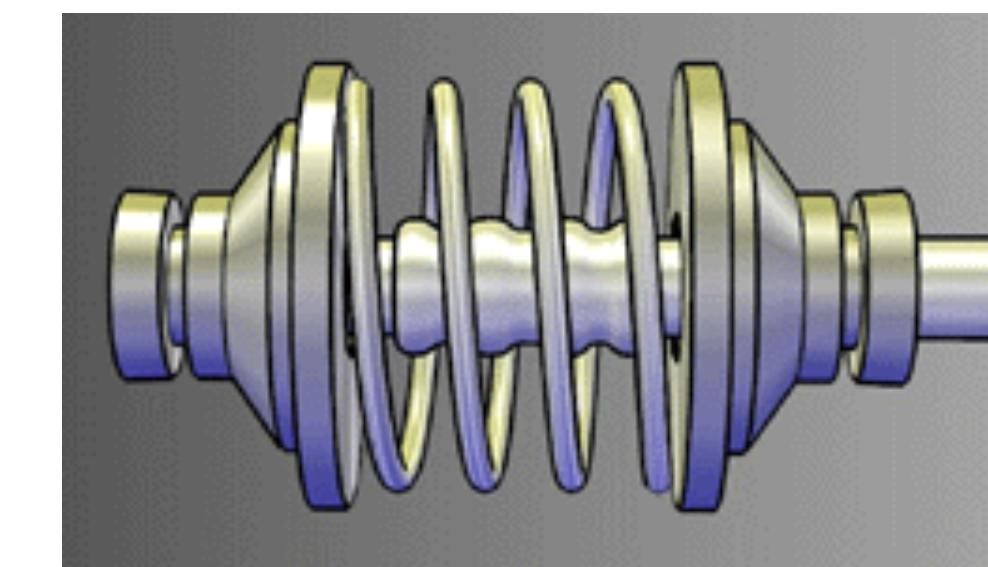
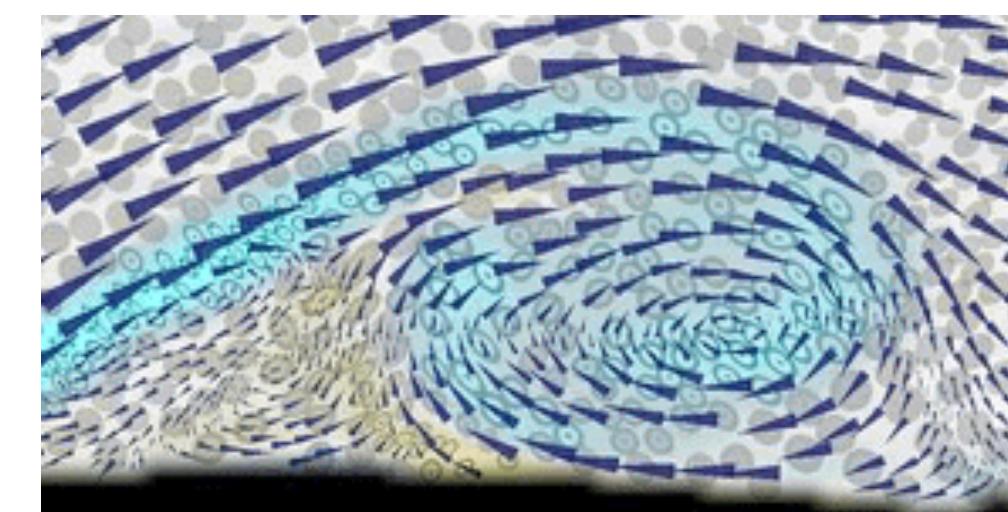
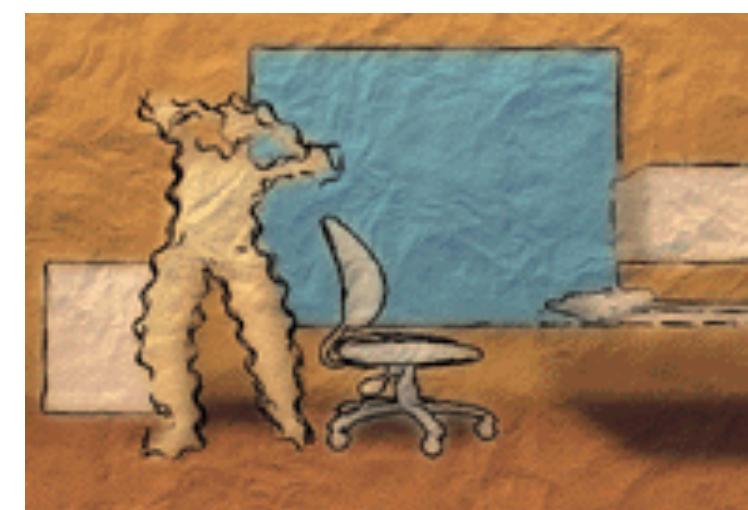
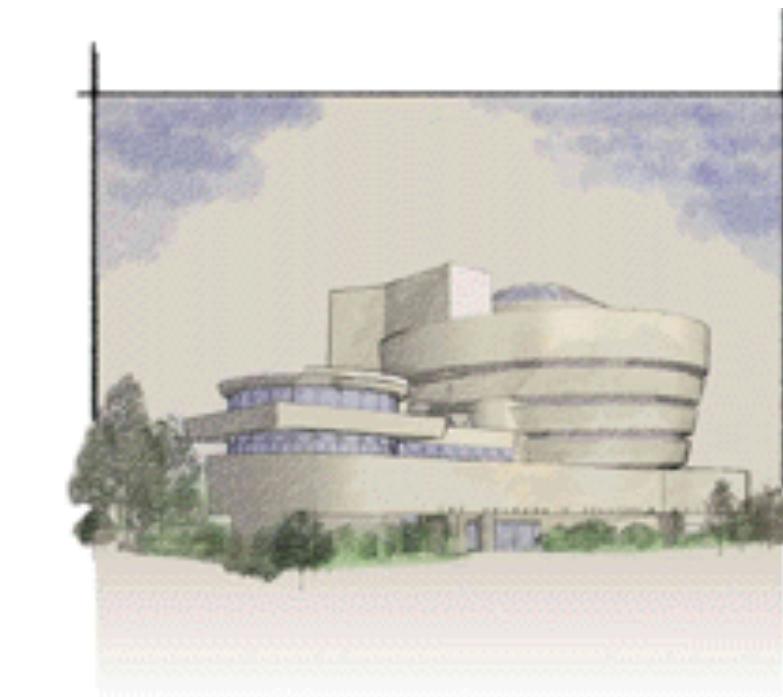


Special Effects

- Usually a combination of *real filmed elements* and *computer-generated*
- Virtual camera must be at *exactly position of the real one*
- Position of CGI elements must be right in “real” world
- May involve not only adding new elements but also removing ones in the real scene (green screen, wires, unwanted background, etc.)



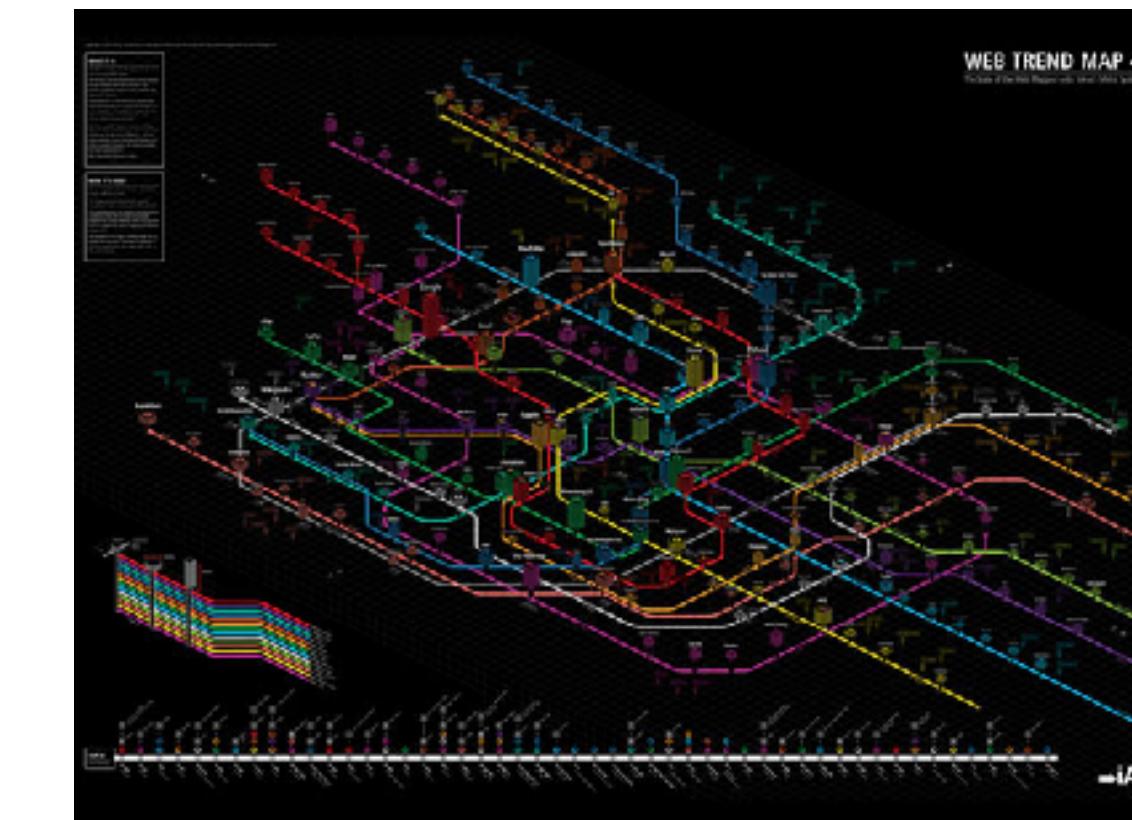
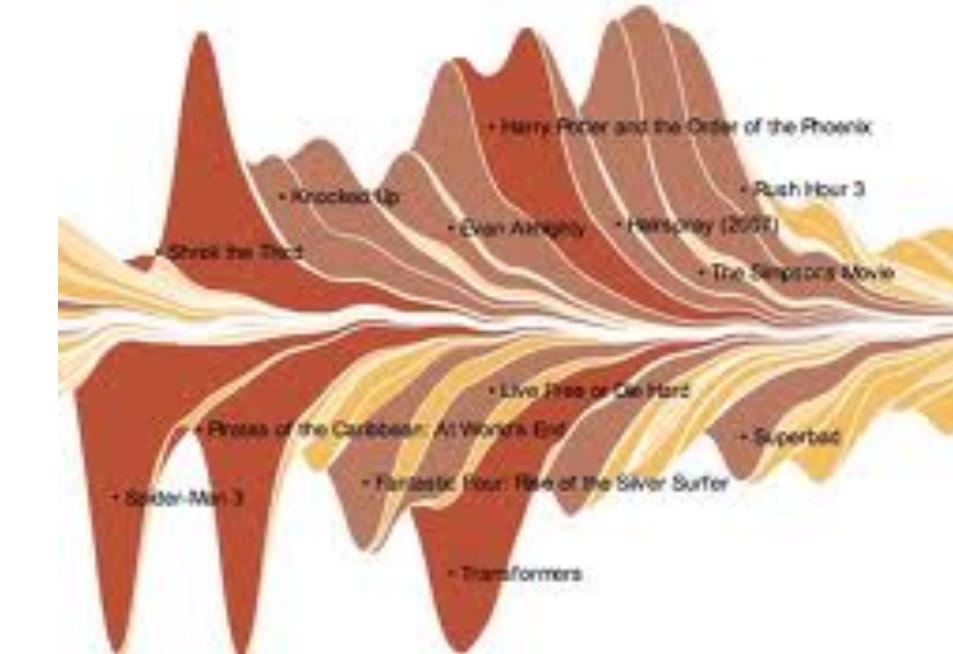
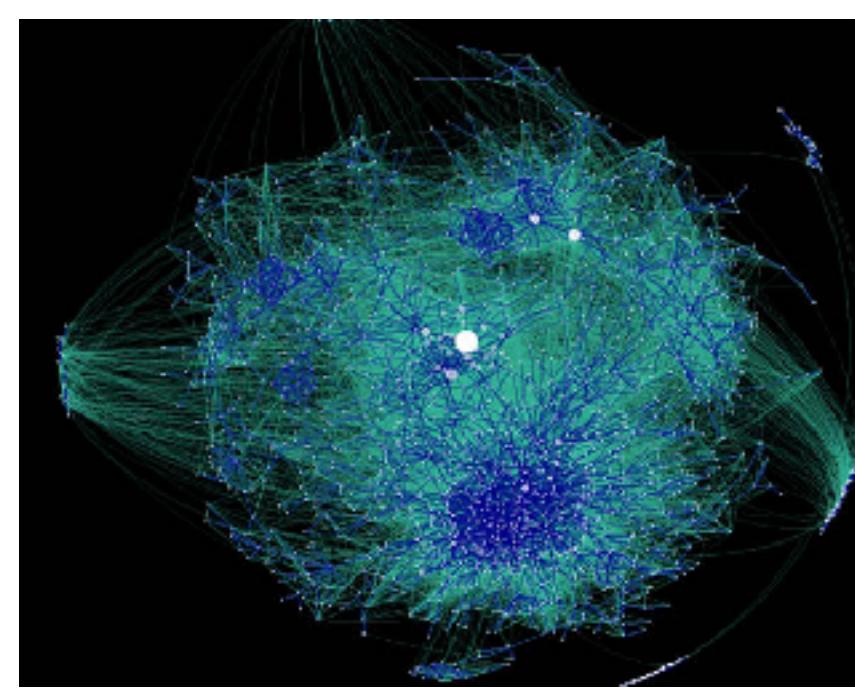
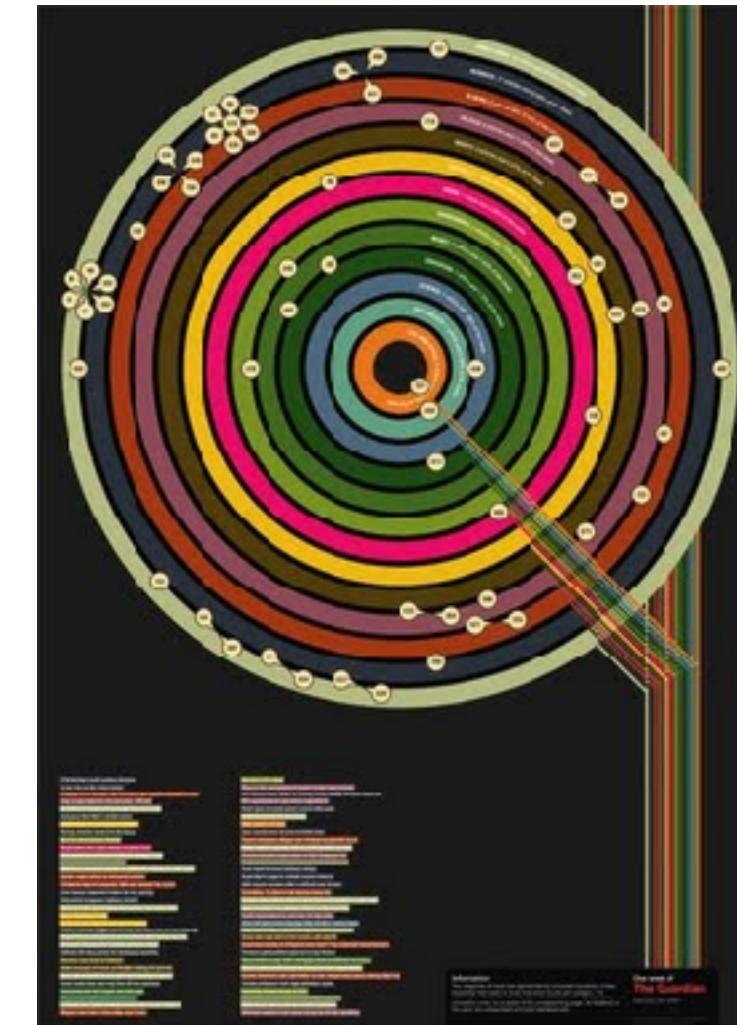
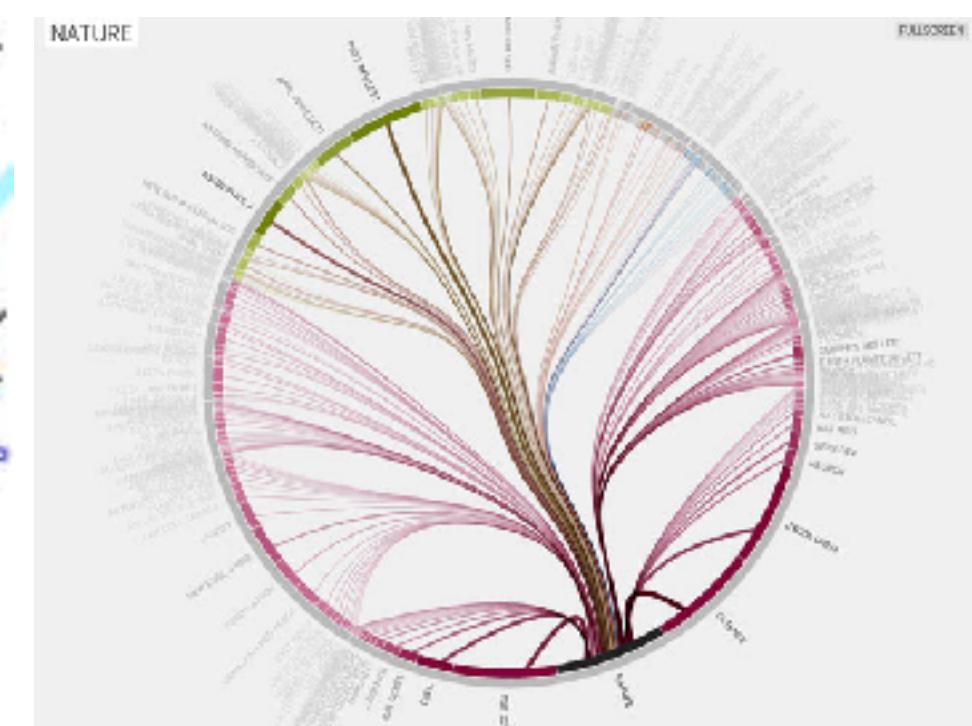
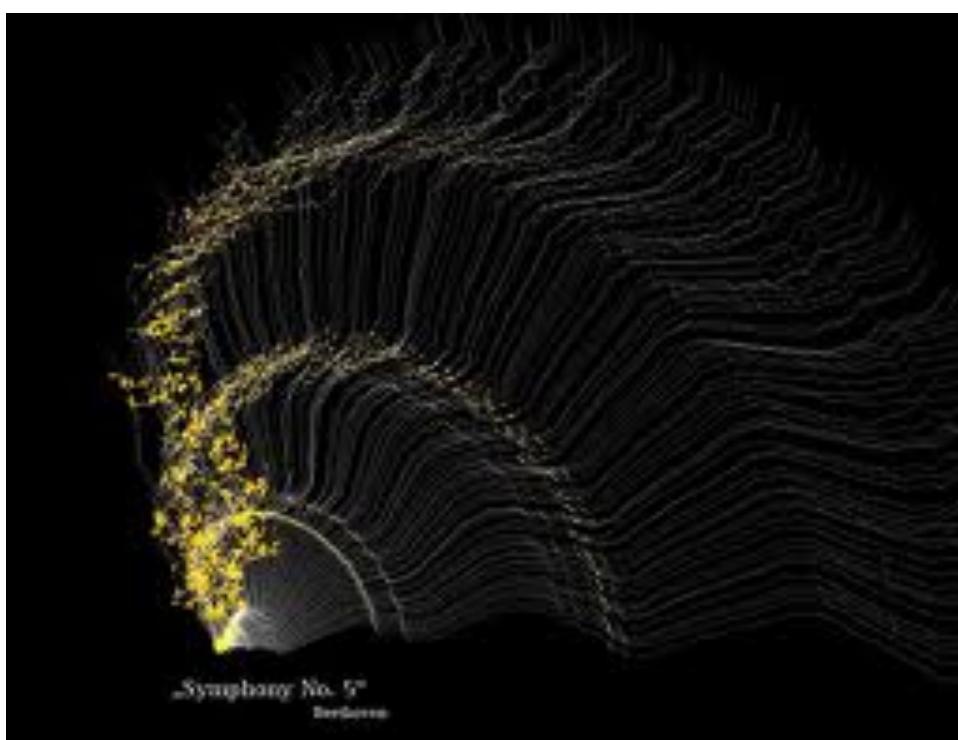
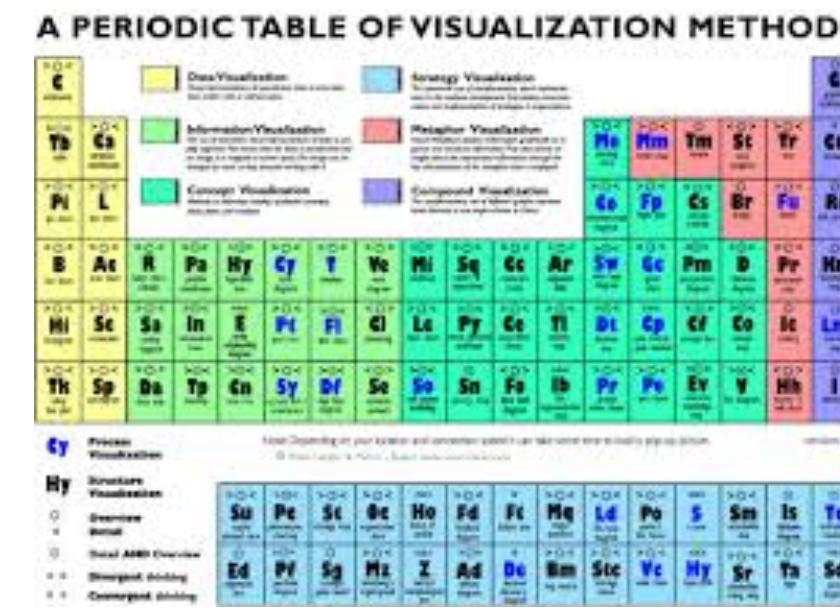
Nonphotorealistic Rendering



Nonphotorealistic Rendering

- 2D
 - Drawing packages
 - Layout
(circuits, music)
 - Illustration
 - Painterly rendering
 - “Toon” generation
- 3D
 - Video games
 - Technical illustration

Visualization



3D Rendering

- If I took a picture of a scene with a virtual camera:
 - What point in 3D is visible at each 2D point in the projected image?
 - What color is the light coming from that point as it reaches the camera?
- Geometry
Lighting



Coming up...

- Camera basics
(we'll come back
to this more later)
- Perspective projection
- Simple modeling primitives
(points, lines, polygons)
- 3D rendering geometry
- Introduction to OpenGL
- Hierarchical transformations
- Visibility
- Lighting

Labs 5–8

- Lab 5: 3D Geometric Rendering (with OpenGL)
- Lab 6: Hierarchical Transformations (with OpenGL)
- Lab 7: 3D Geometric Rendering (implement it yourself)
- Lab 8: Visibility, Lighting, and Shading (implement it yourself)



Cameras and Projection

CS 355: Introduction to Graphics and Image Processing

3D Rendering

- If I took a picture of a scene with a virtual camera:
 - What point in 3D is visible at each 2D point in the projected image?
 - What color is the light coming from that point as it reaches the camera?
- Geometry
Lighting

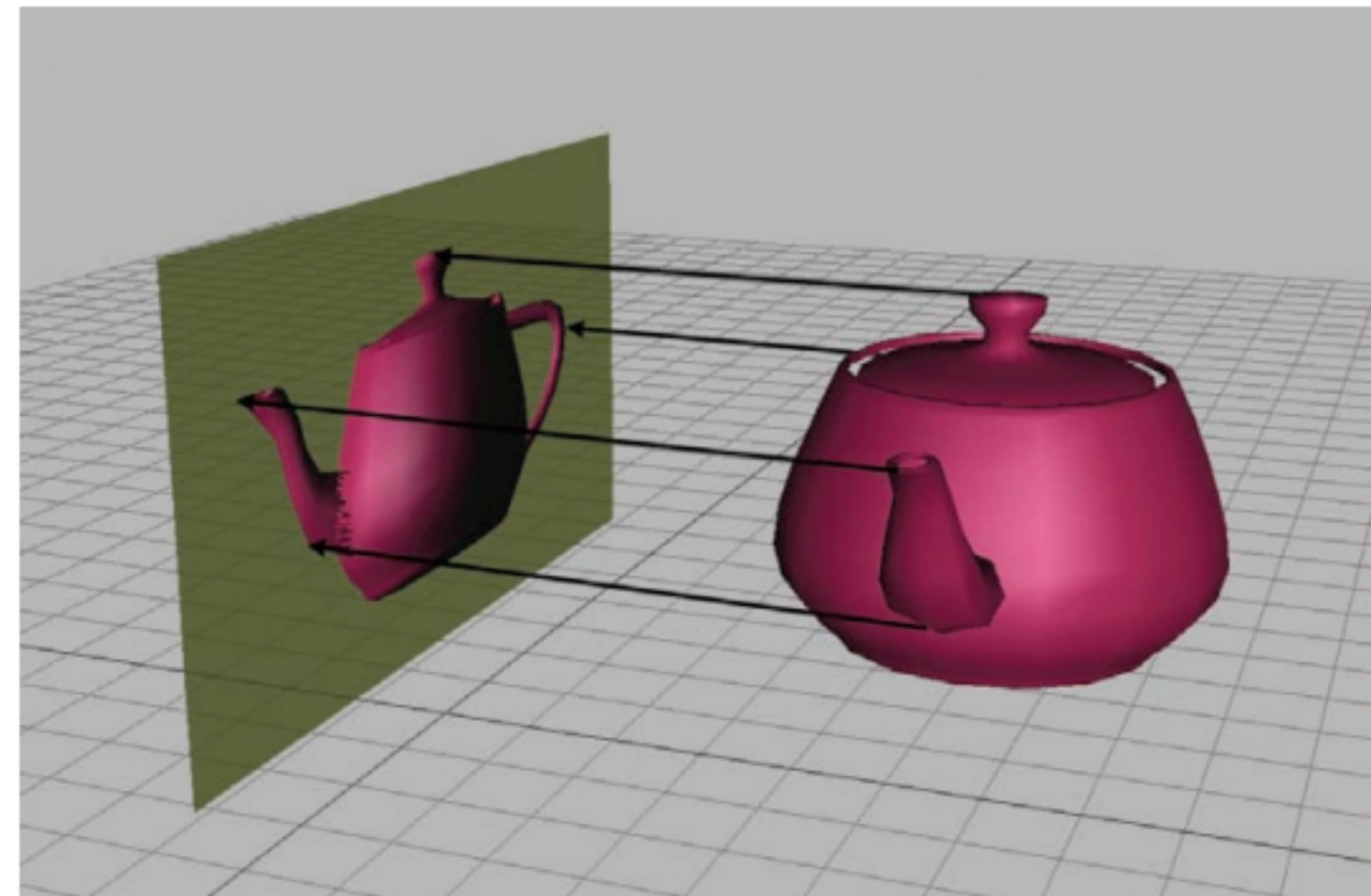


Projection

- To get 2D pictures of a 3D world,
you have to use *projection*
 - Orthographic
 - Perspective



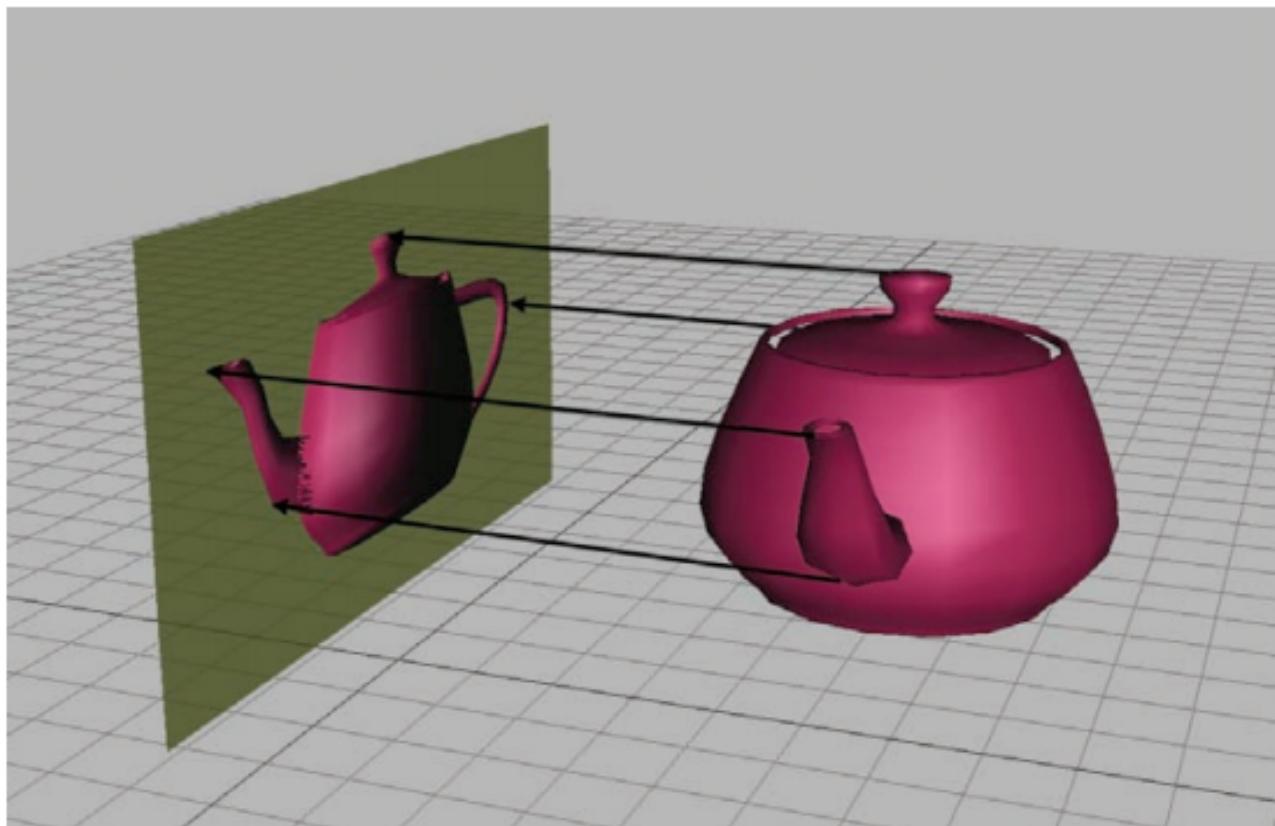
Orthographic Projection



Orthographic projection is just dropping a dimension.

Used in technical drawings, etc.

Orthographic Projection



3D point in
homogeneous
coordinates

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Orthographic projection is just dropping a dimension.

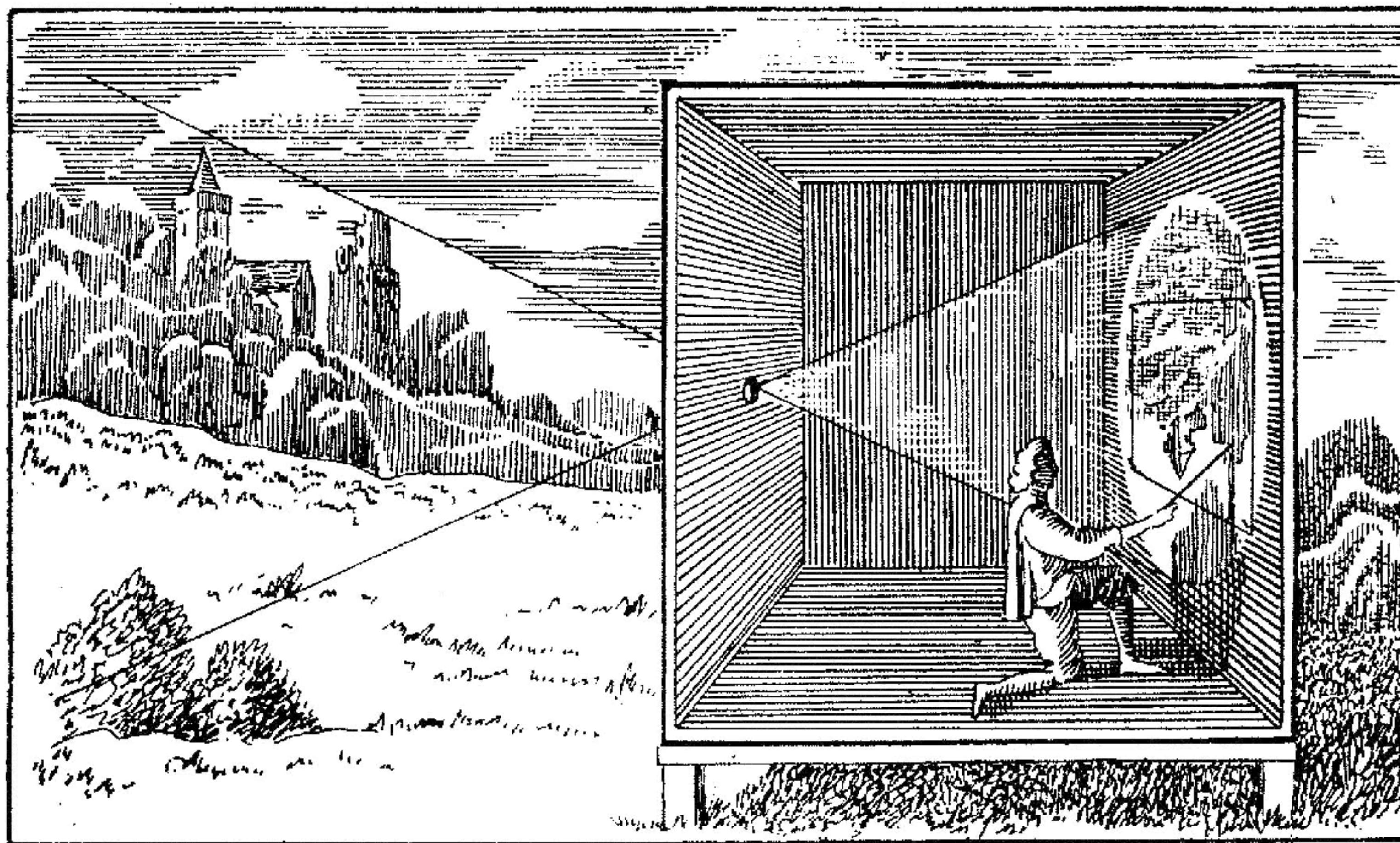
Lacking Perspective



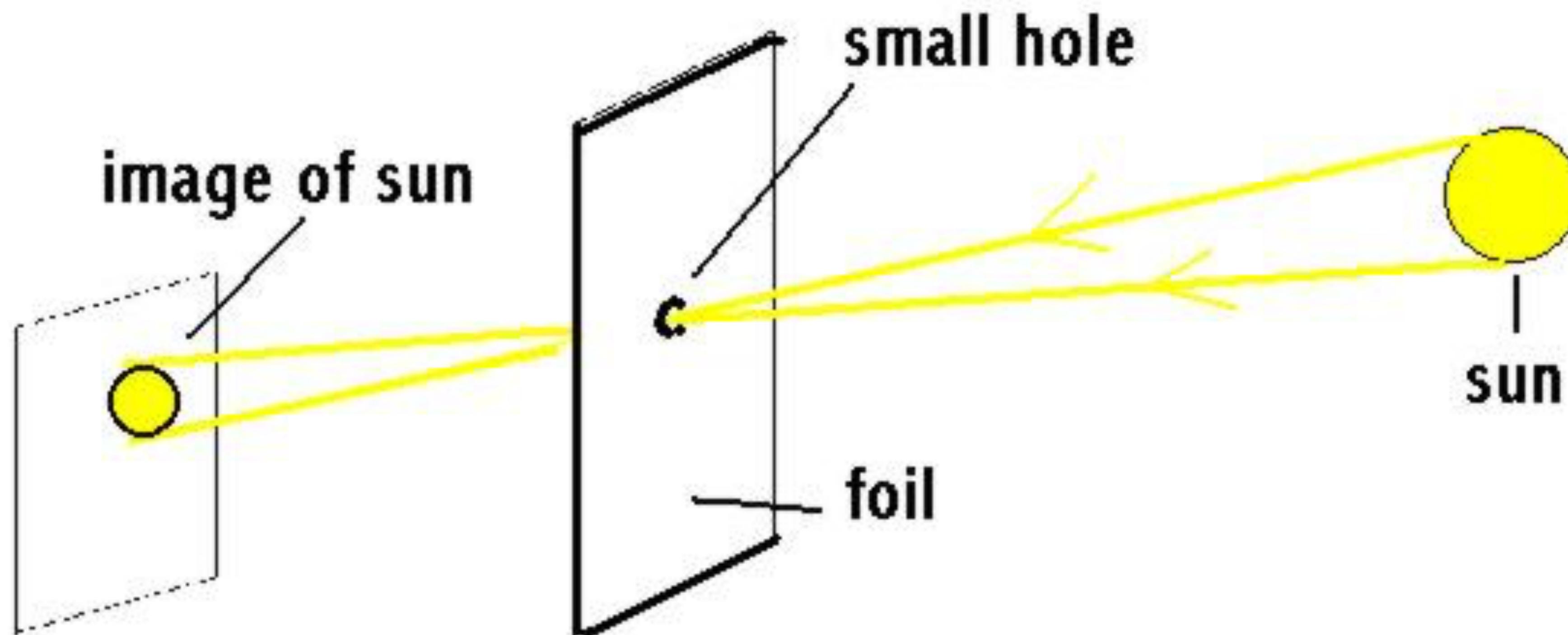
Perspective Projection



Camera Obscura



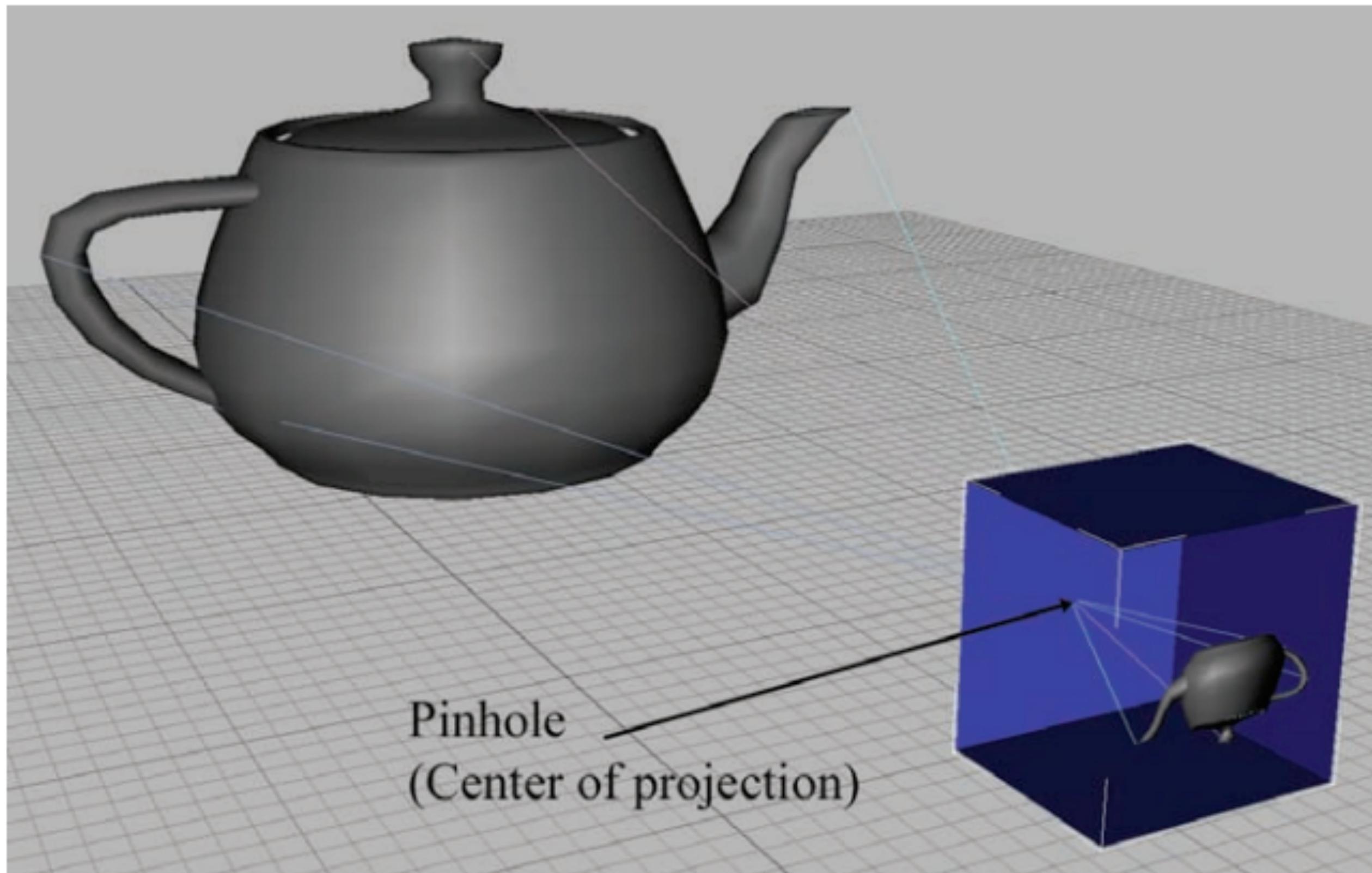
Eclipse Viewing



Eclipse Viewing

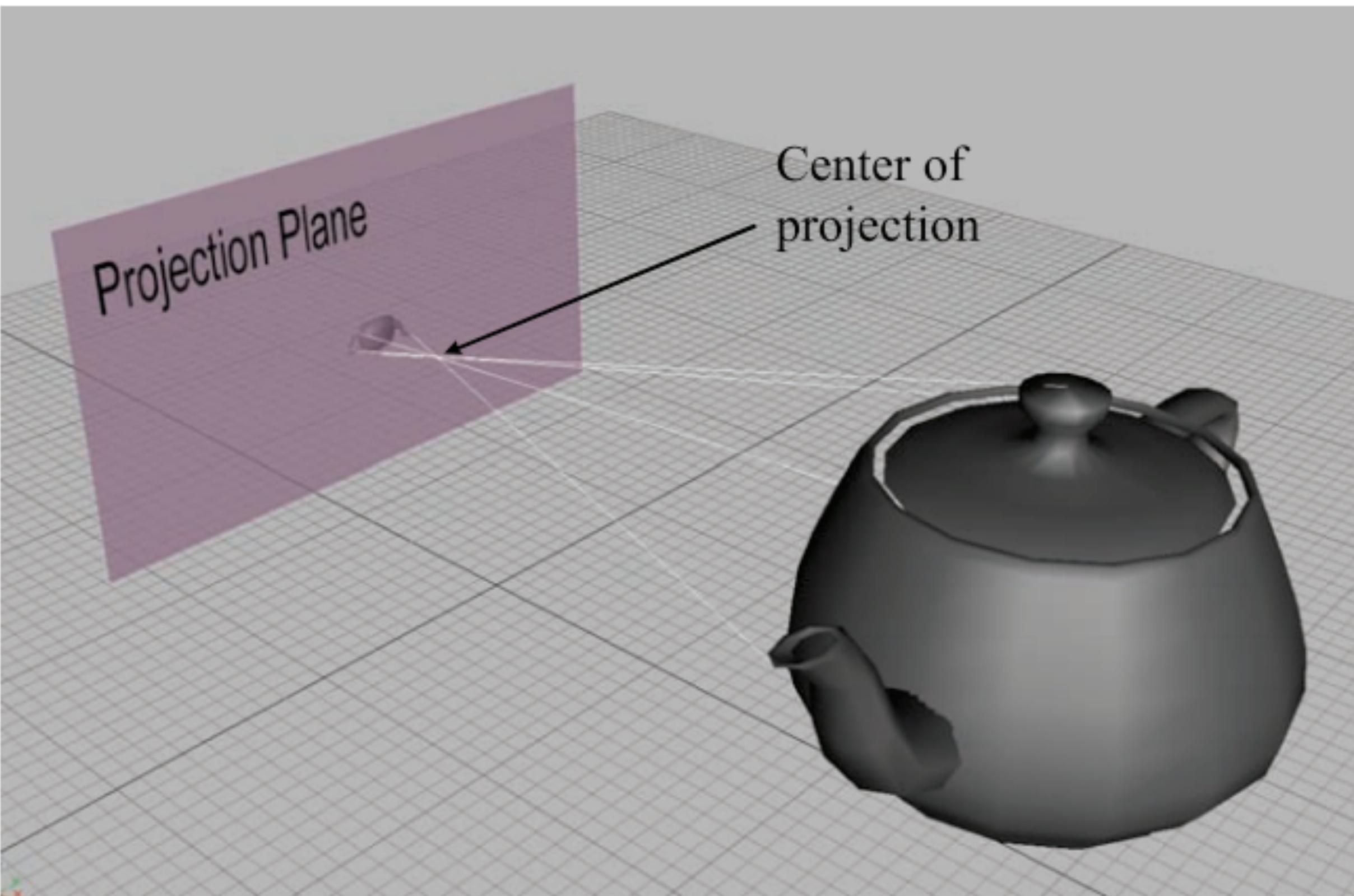


Perspective Projection



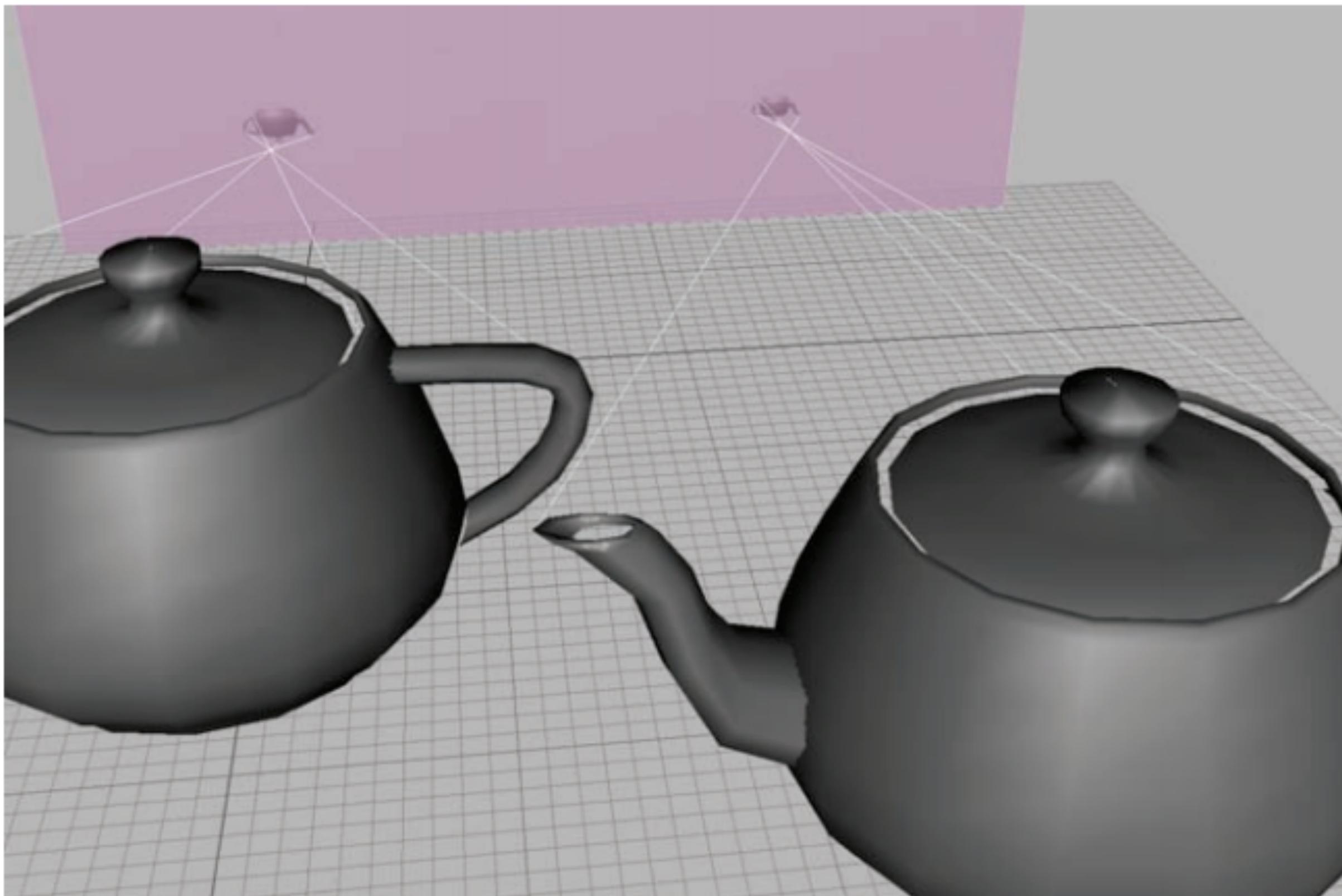
Many graphics systems assume a simple pinhole camera model

Perspective Projection

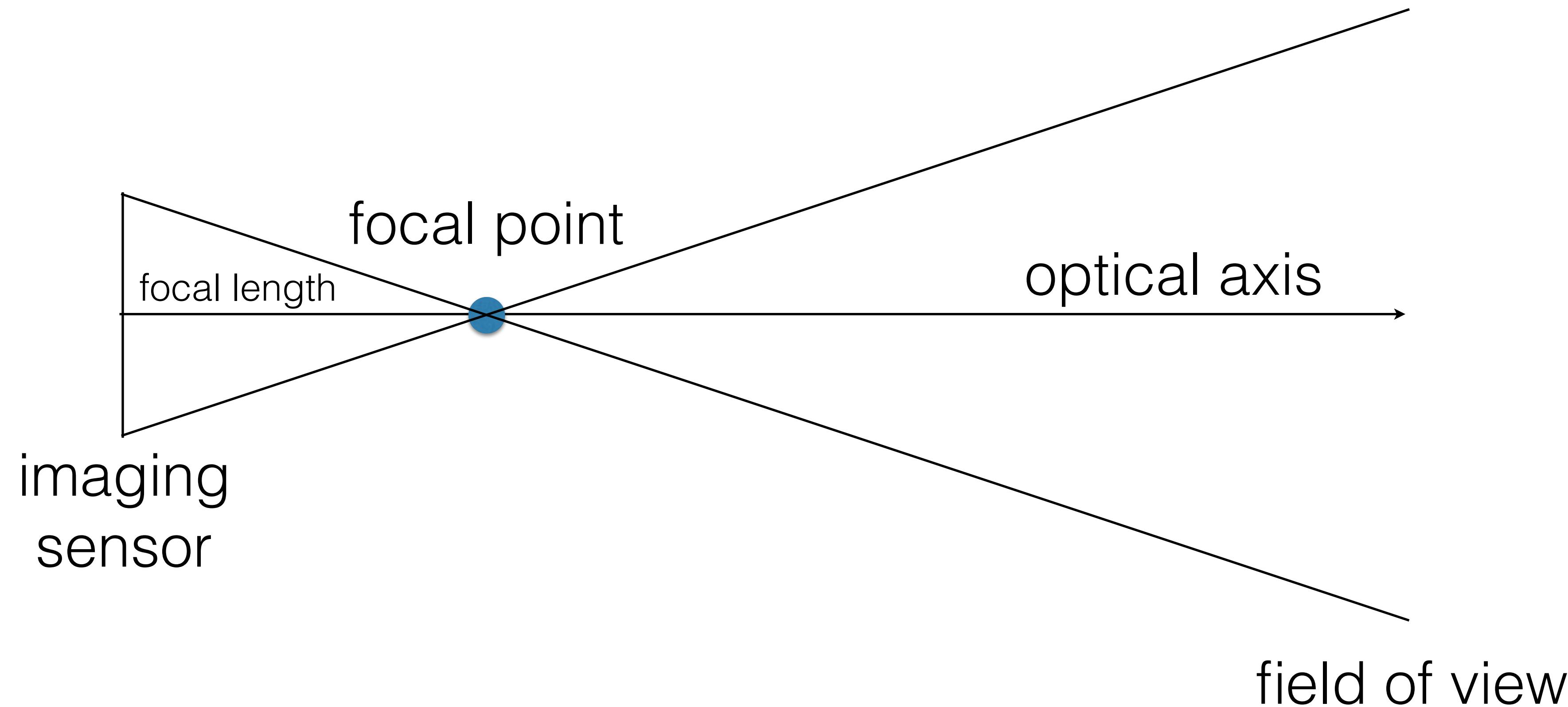


Pinhole camera model

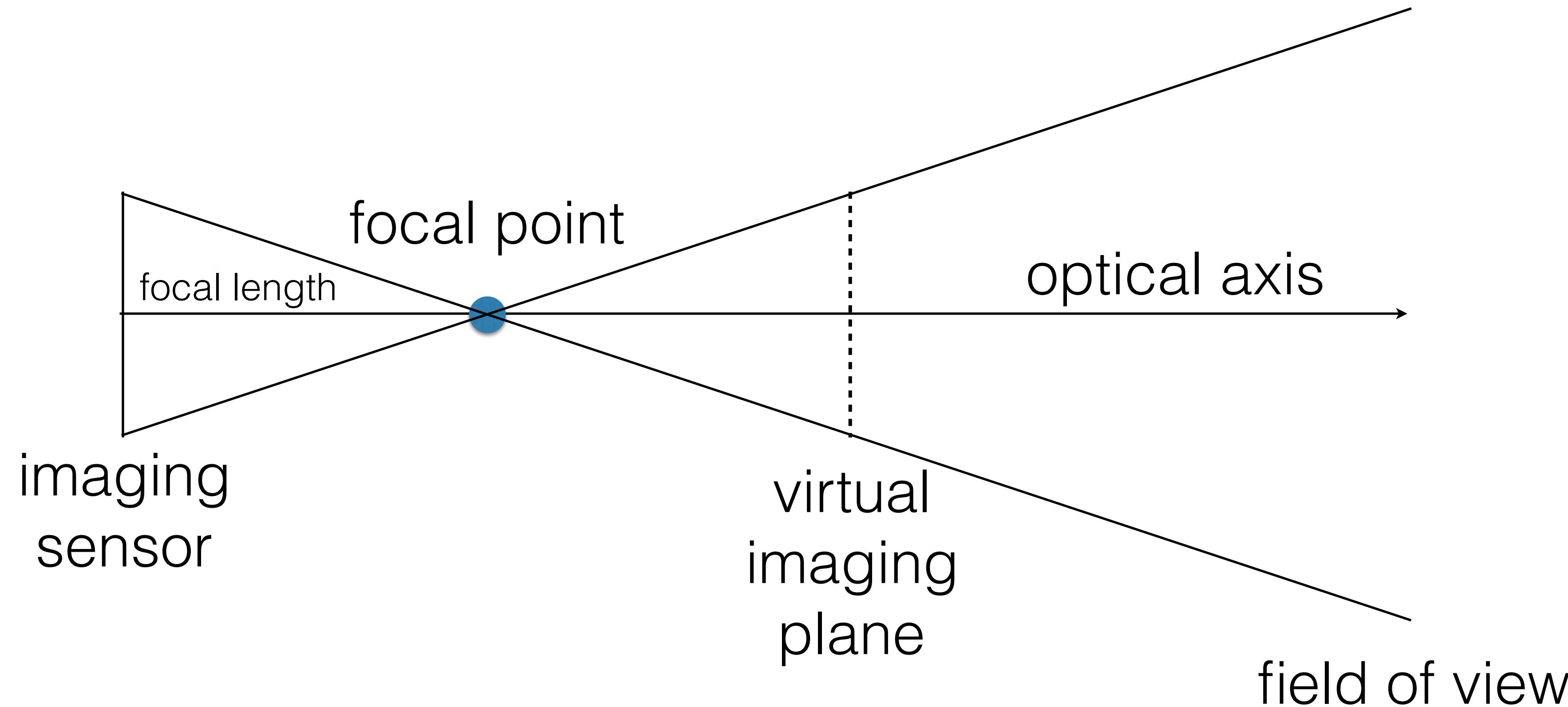
Perspective Projection



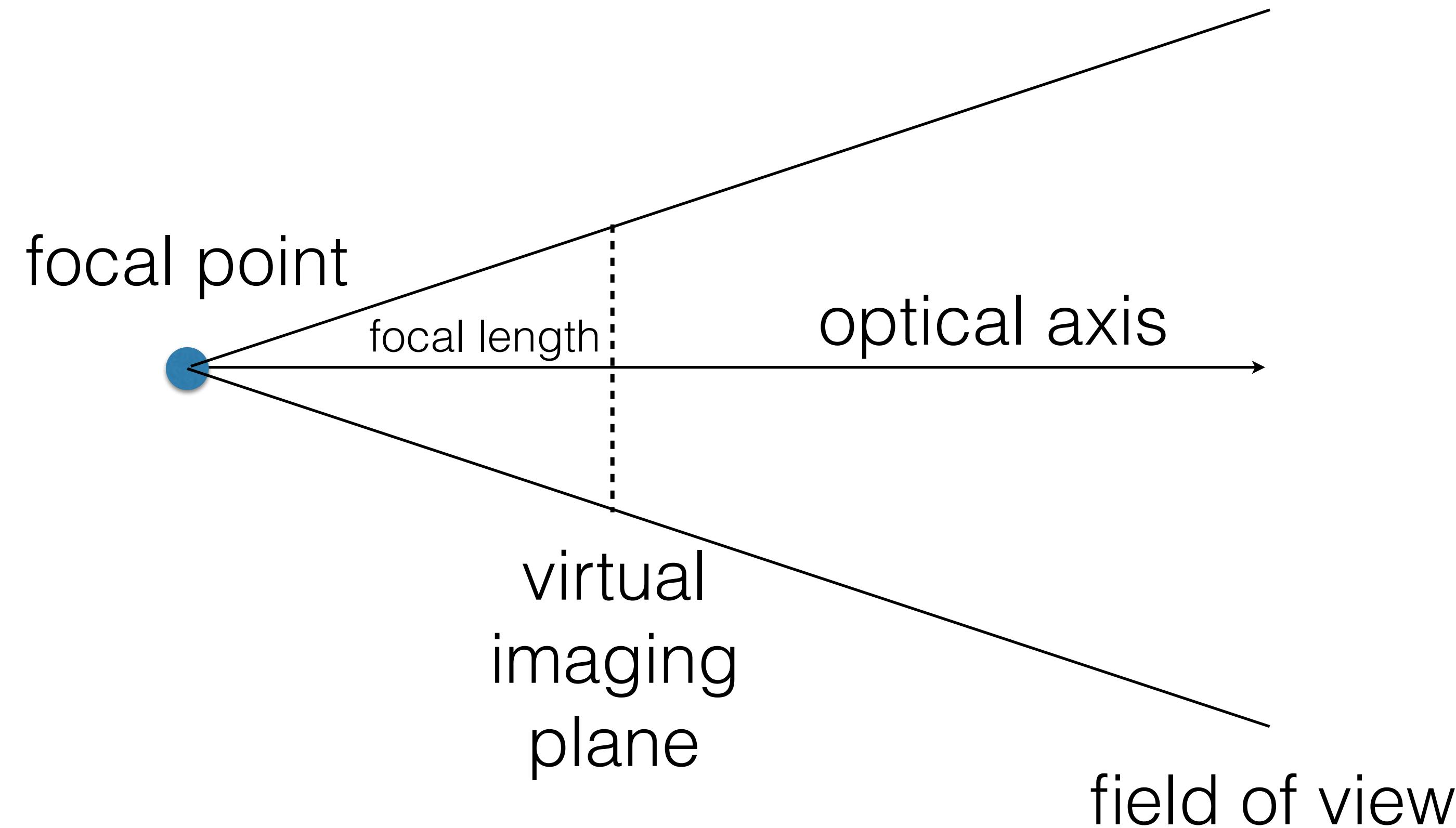
Pinhole Cameras



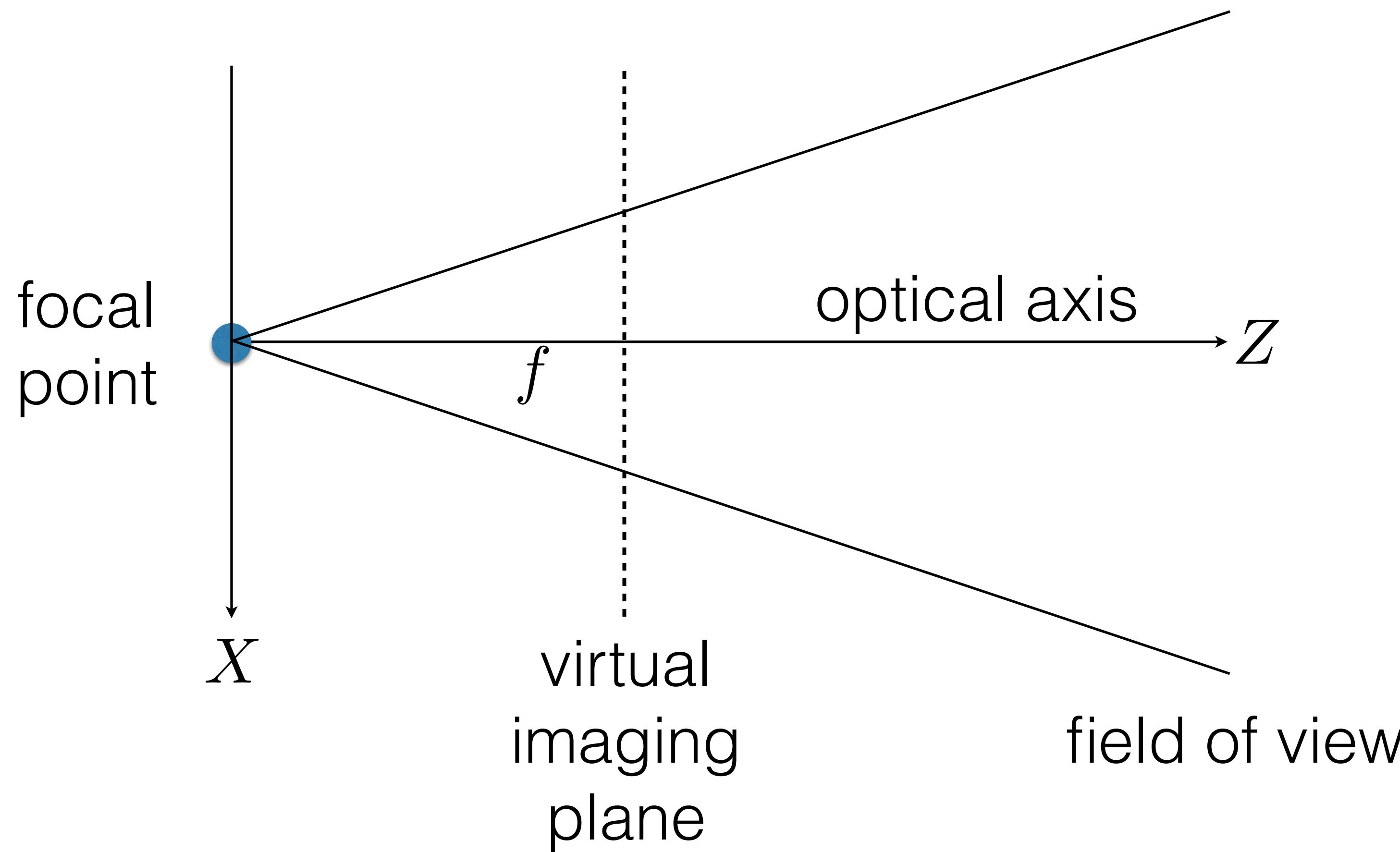
Geometric Model



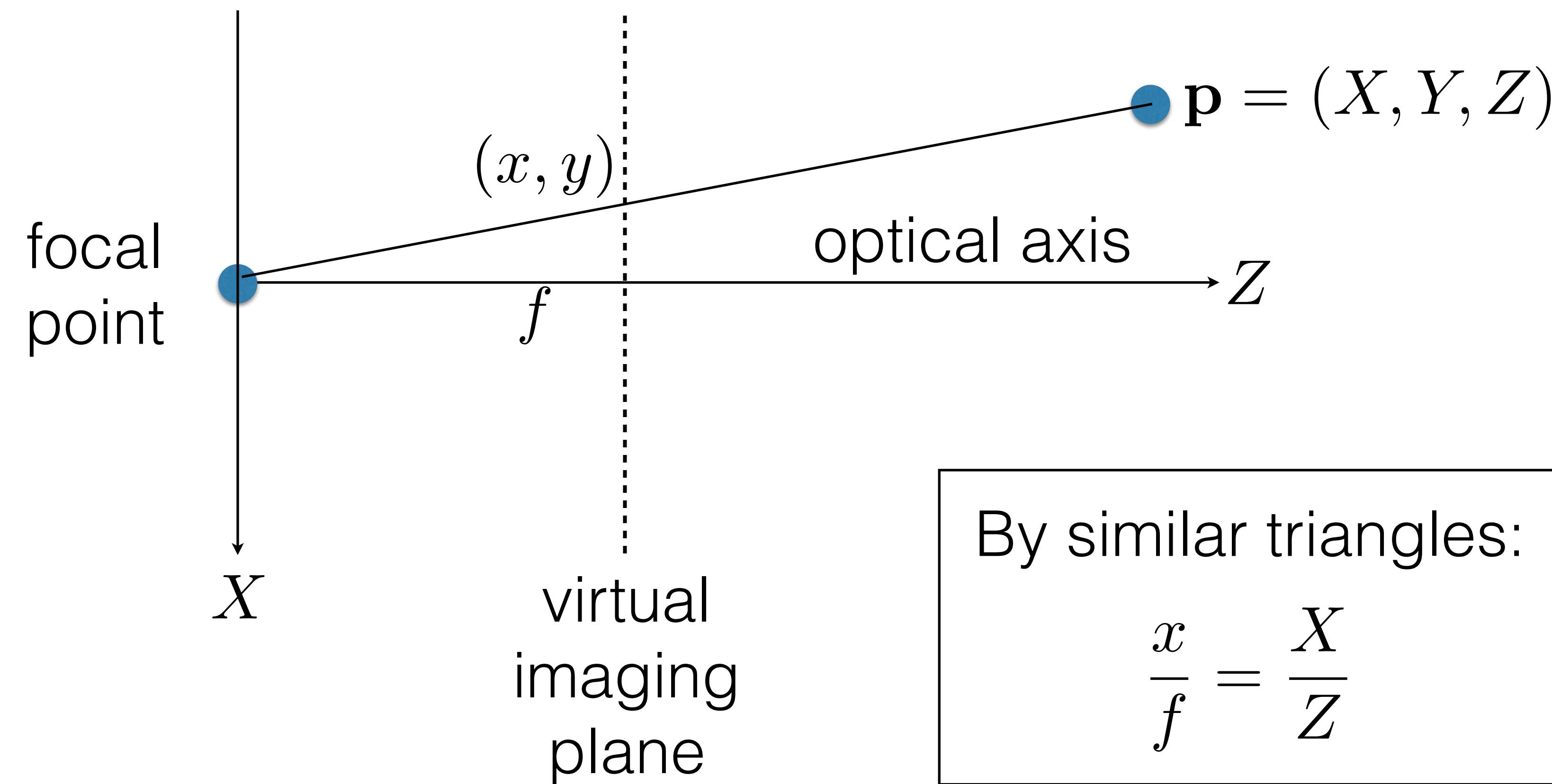
Geometric Model



Camera Coordinates



Projection



Projection

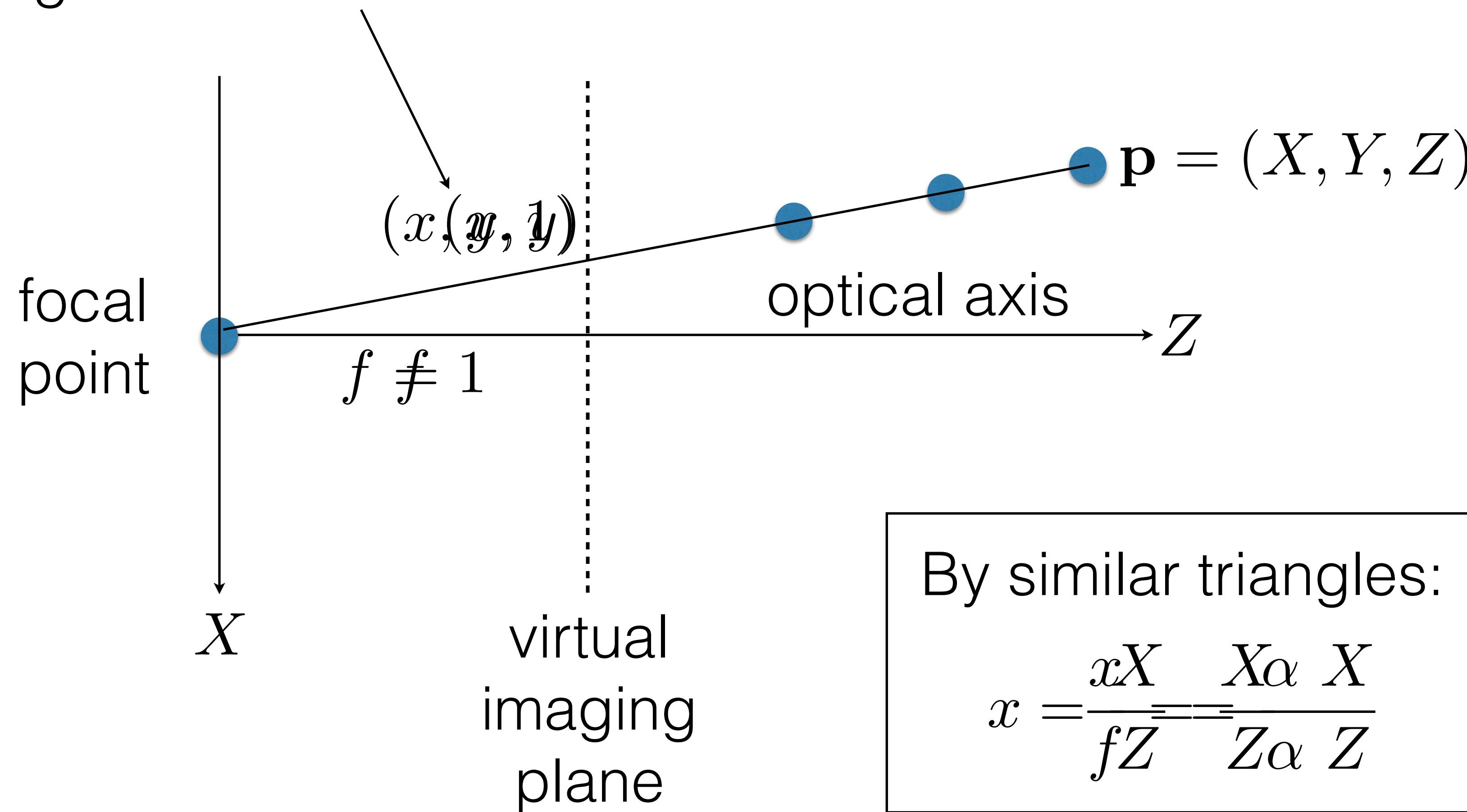
$$\frac{x}{f} = \frac{X}{Z} \qquad \qquad \frac{y}{f} = \frac{Y}{Z}$$

$$(x, y) = \left(\frac{fX}{Z}, \frac{fY}{Z} \right)$$

Note: this is the projected coordinate in real-world units.
To get actual pixel location, have to scale by pixel density
and apply offset to image origin (more on this later...)

Projection

homogeneous coordinate!

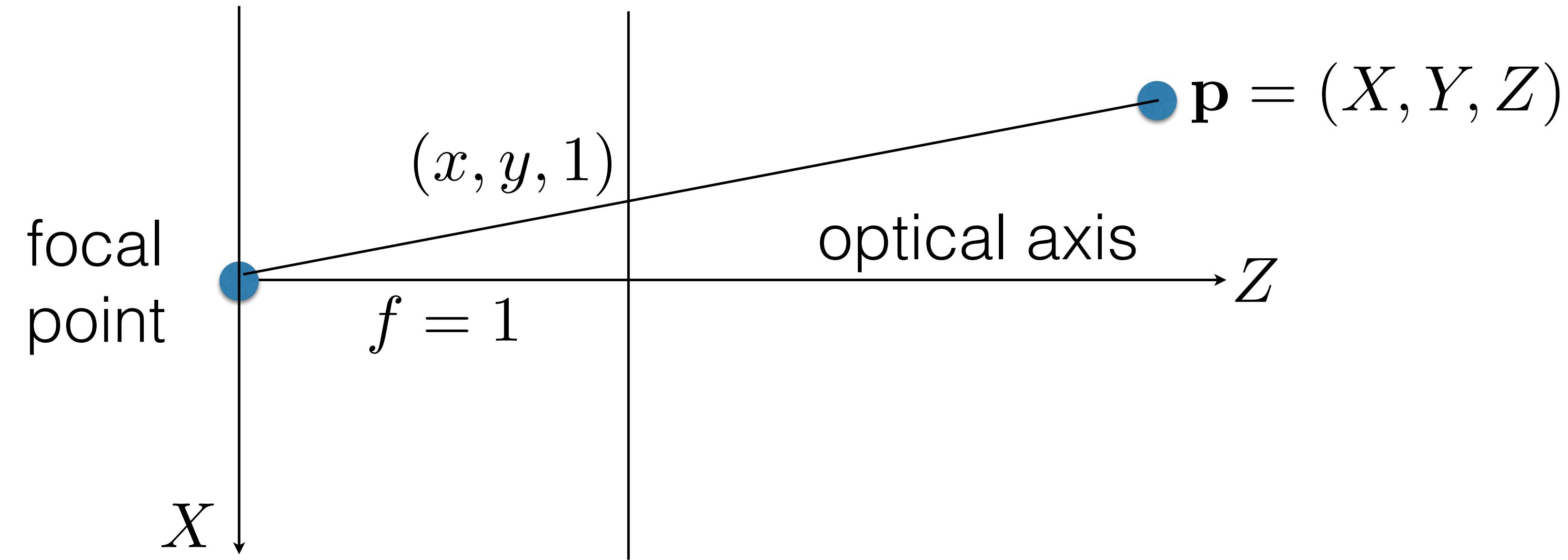


Homogenous Coordinates

- Homogeneous coordinates are used to represent all 3D points along the ray that falls on the same 2D projection:

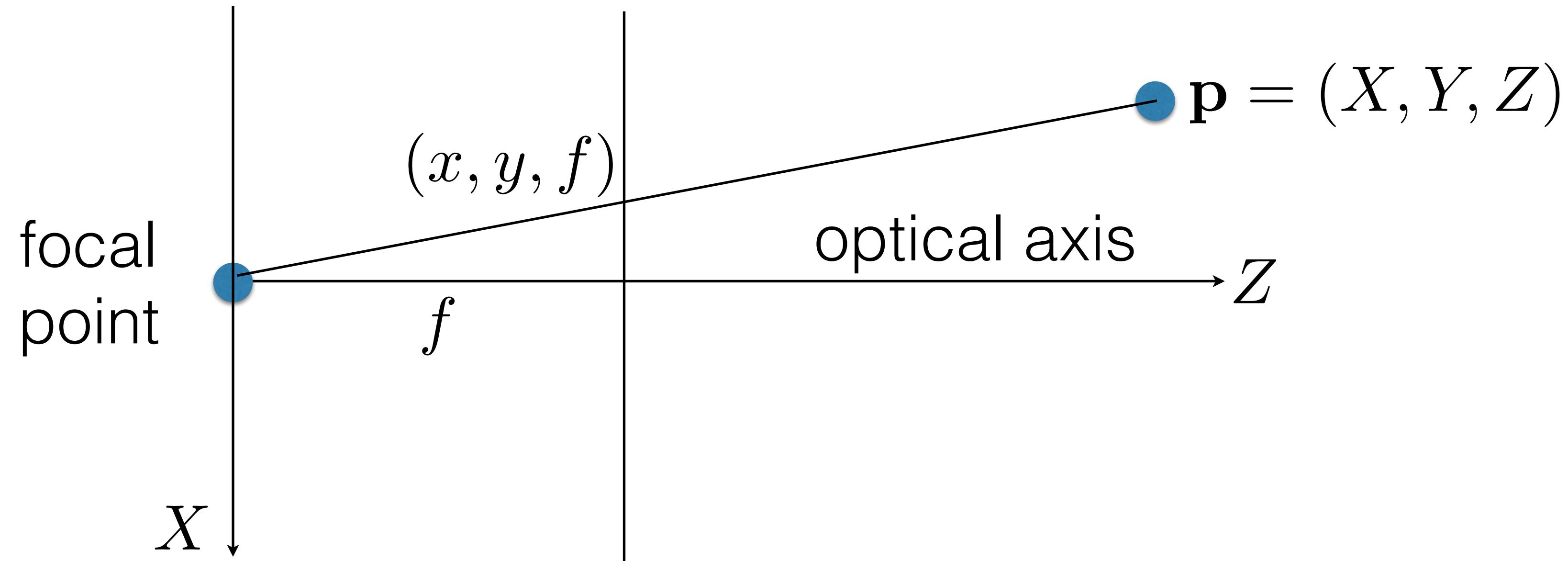
$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \sim \begin{bmatrix} \alpha x \\ \alpha y \\ \alpha \end{bmatrix}$$

Perspective Projection



$$\begin{bmatrix} x \\ y \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} X/Z \\ Y/Z \\ 1 \\ 1 \end{bmatrix} \sim \begin{bmatrix} X \\ Y \\ Z \\ Z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Perspective Projection



$$\begin{bmatrix} x \\ y \\ f \\ 1 \end{bmatrix} = \begin{bmatrix} fX/Z \\ fY/Z \\ f \\ 1 \end{bmatrix} \sim \begin{bmatrix} X \\ Y \\ Z \\ Z/f \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Alternative Form

One way (some implementation advantages):

$$\begin{bmatrix} x \\ y \\ \hline f \\ 1 \end{bmatrix} = \begin{bmatrix} fX/Z \\ fY/Z \\ \hline f \\ 1 \end{bmatrix} \sim \begin{bmatrix} X \\ Y \\ Z \\ Z/f \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Another way (some conceptual advantages):

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} fX/Z \\ fY/Z \\ 1 \end{bmatrix} \sim \begin{bmatrix} X \\ Y \\ Z/f \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Coming up...

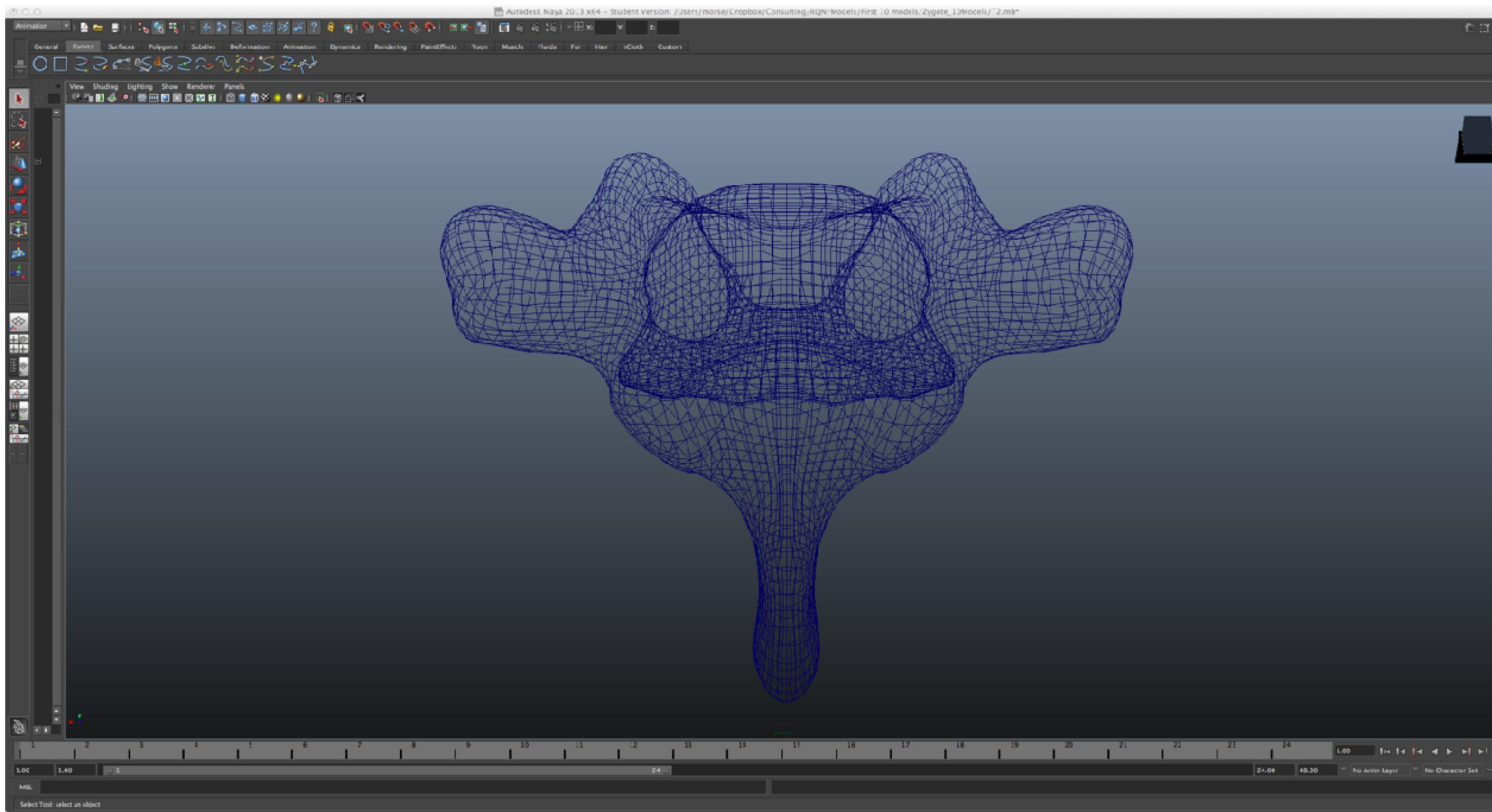
- Simple modeling primitives
(points, lines, polygons)
- 3D rendering geometry
- Introduction to OpenGL
- Hierarchical transformations
- Visibility
- Lighting



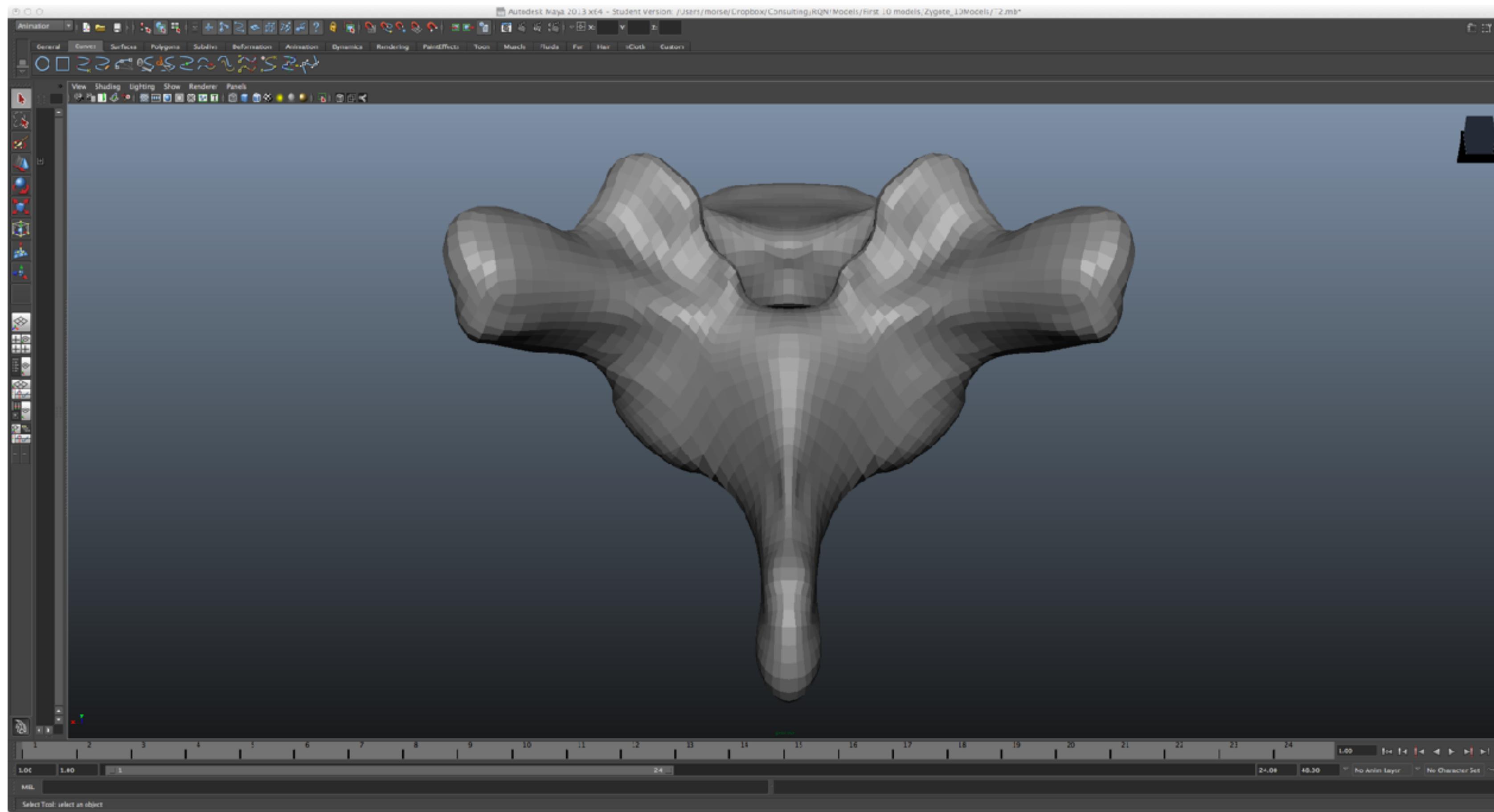
Points, Lines, and Polygons

CS 355: Introduction to Graphics and Image Processing

Wireframe Meshes

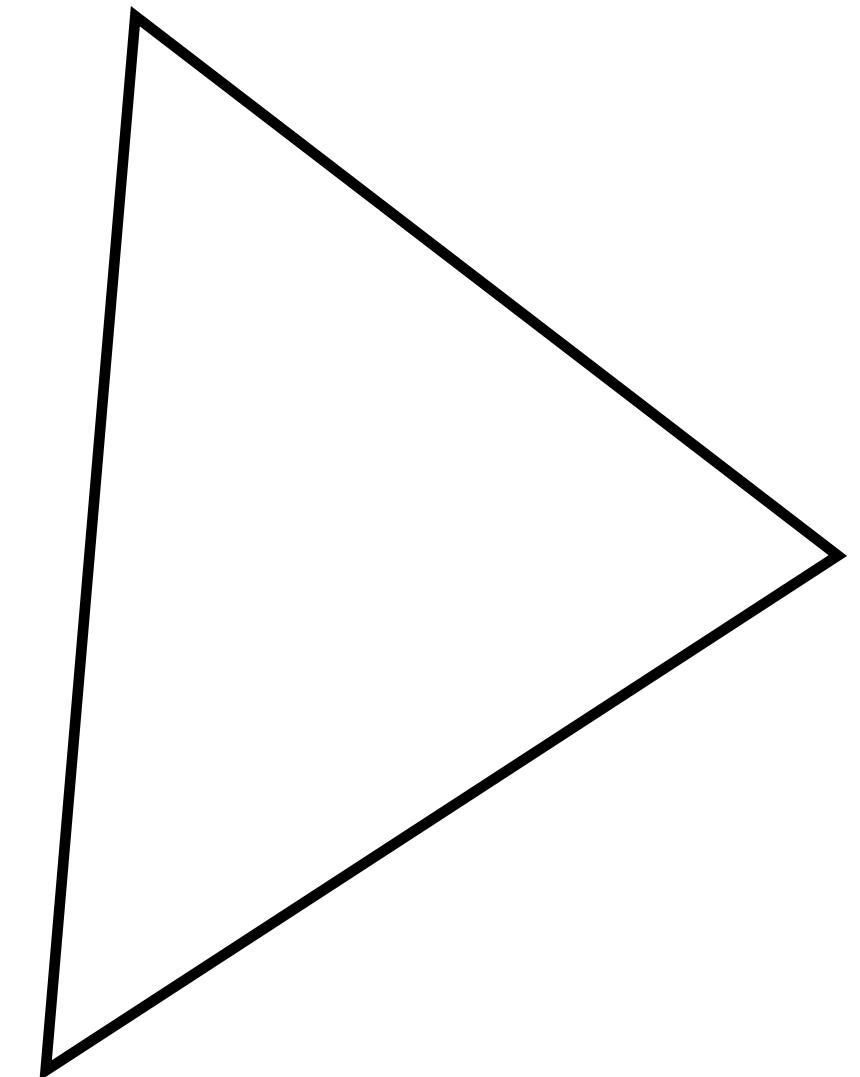


Polygonal Faces



Primitives

- Vertex:
A 3D point (X,Y,Z)
- Edge:
A line connecting two vertices
- Face:
A polygon defined by a set of “adjacent” (connected by edges) vertices



Storage

- Common way to store models:
 - List of vertices
 - List of faces bound by vertices (by index)
 - Other information about vertices or faces
- Avoids duplication of redundant data

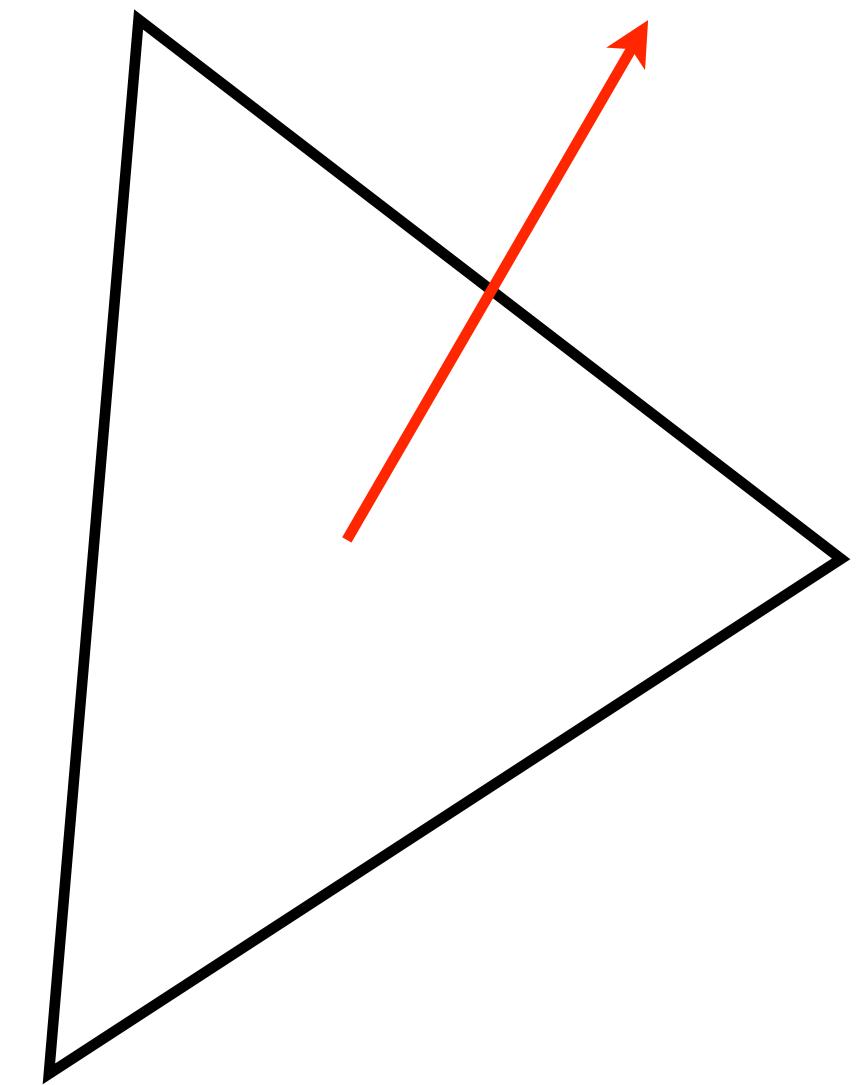
Storage Example

```
# List of Vertices, with (x,y,z[,w]) coordinates,  
# w is optional and defaults to 1.0.  
v 0.123 0.234 0.345 1.0  
v ...  
...
```

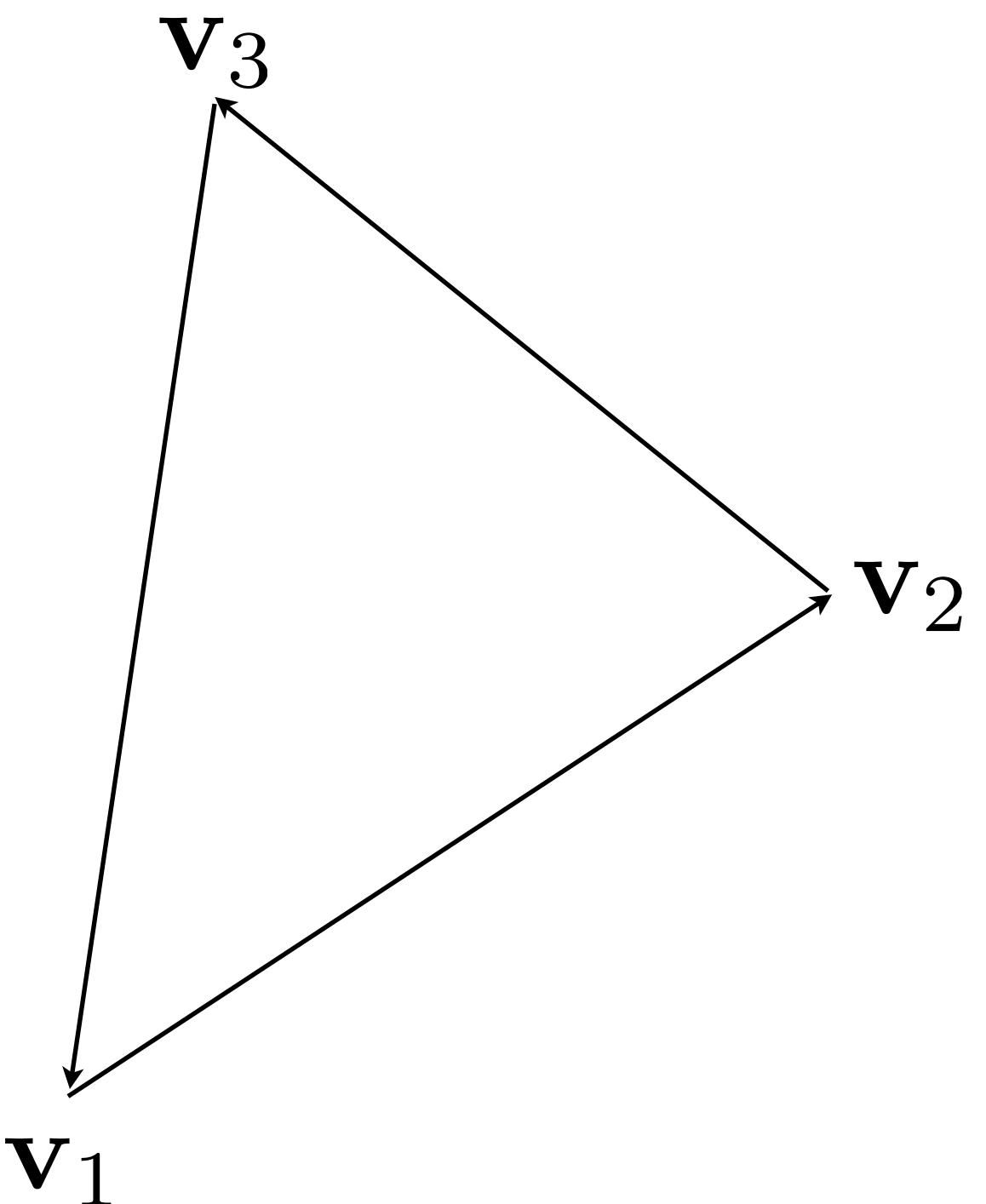
```
# Face Definitions (see below)  
f 1 2 3  
f 3 4 5  
f 6 3 7  
f ...  
...
```

Normals

- It's useful to determine the *normal* to the polygonal face
 - Visibility
 - Lighting
 - ...
- Be consistent—usually go with outward facing



Calculating Normals



$$\hat{\mathbf{n}} = \frac{(\mathbf{v}_2 - \mathbf{v}_1) \times (\mathbf{v}_3 - \mathbf{v}_2)}{\|(\mathbf{v}_2 - \mathbf{v}_1) \times (\mathbf{v}_3 - \mathbf{v}_2)\|}$$

Assumes a consistent
winding order

Coming up...

- 3D rendering geometry
- Introduction to OpenGL
- Hierarchical transformations
- Visibility
- Lighting

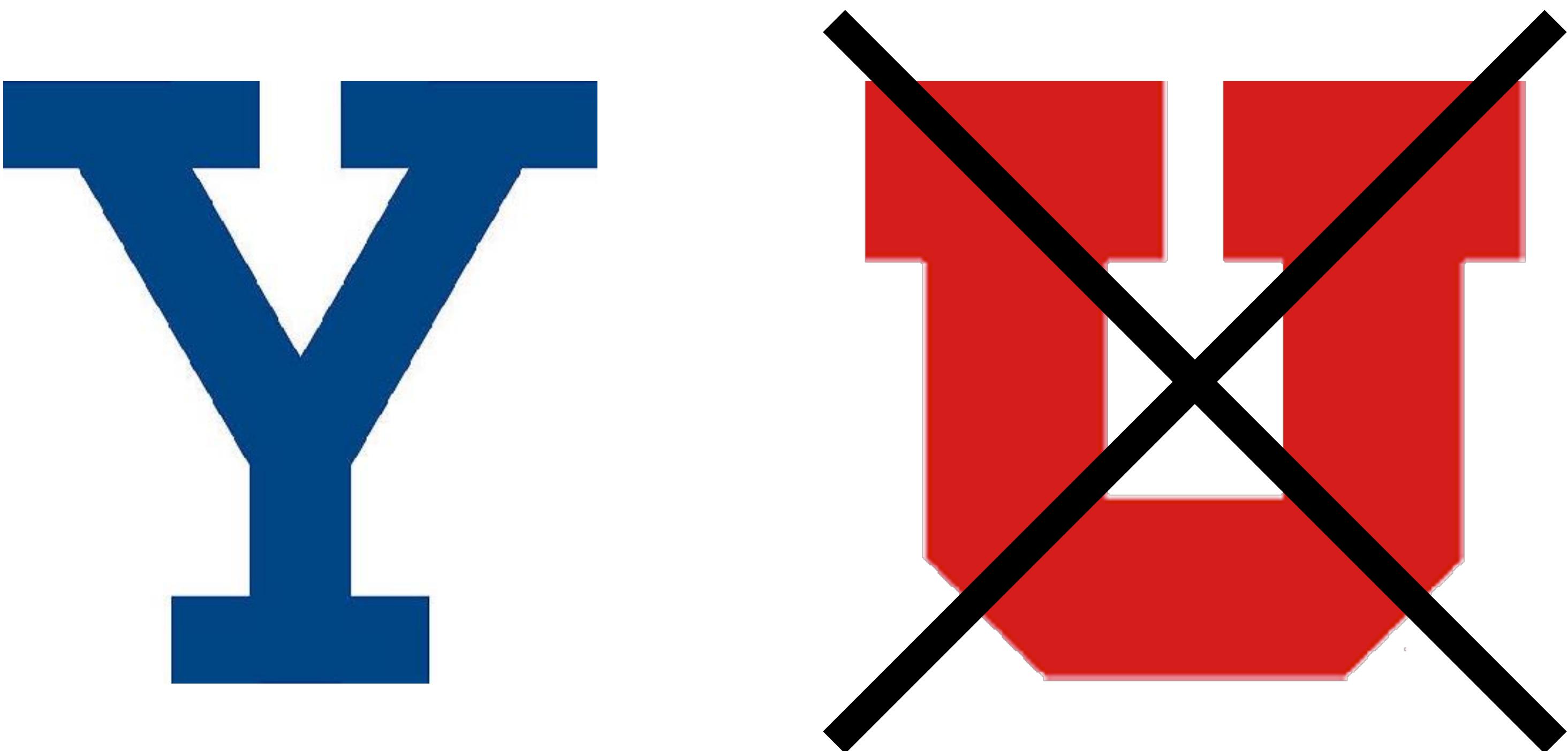


3D Rendering Geometry

CS 355: Introduction to Graphics and Image Processing

First, a detour on object modeling...

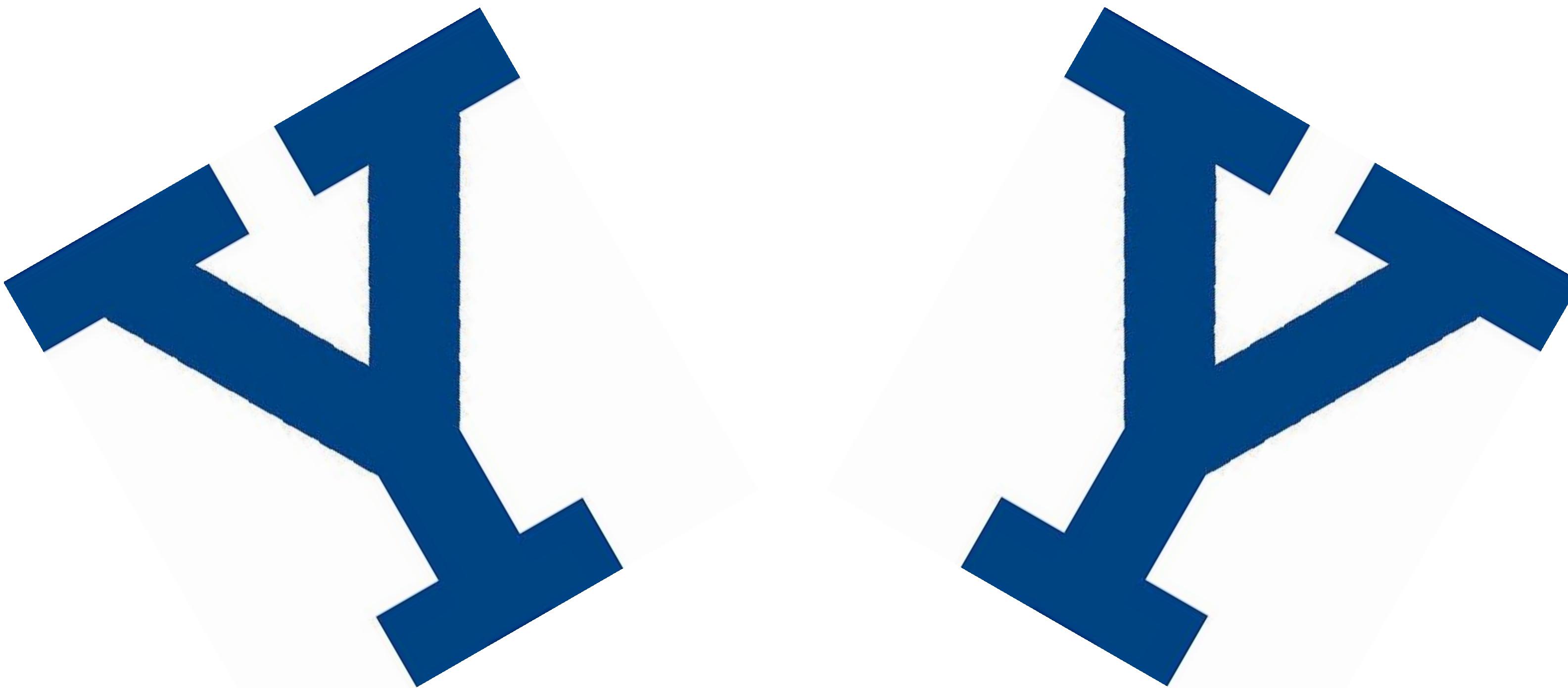
Objects



Objects



Objects



Objects

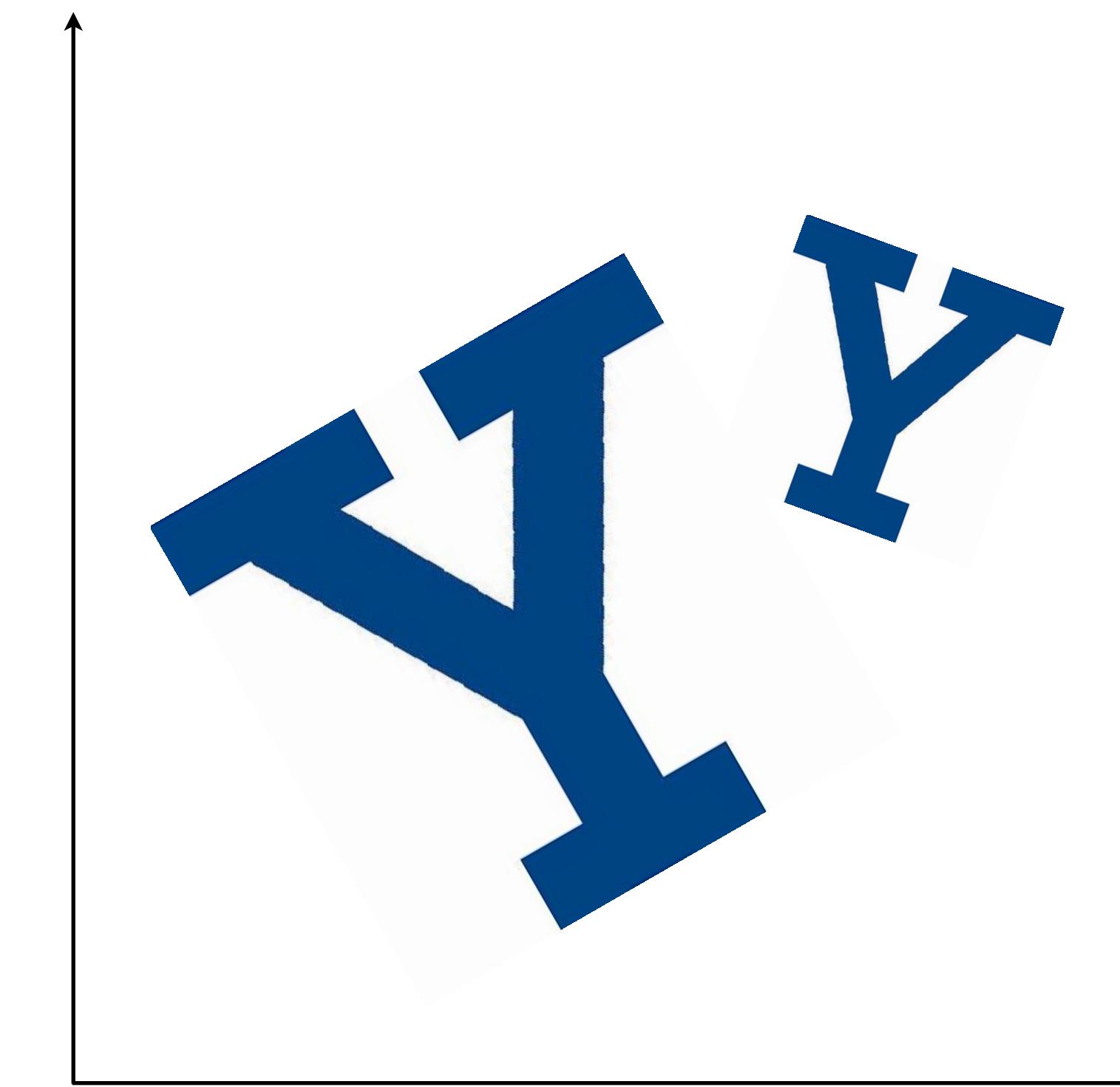


Objects



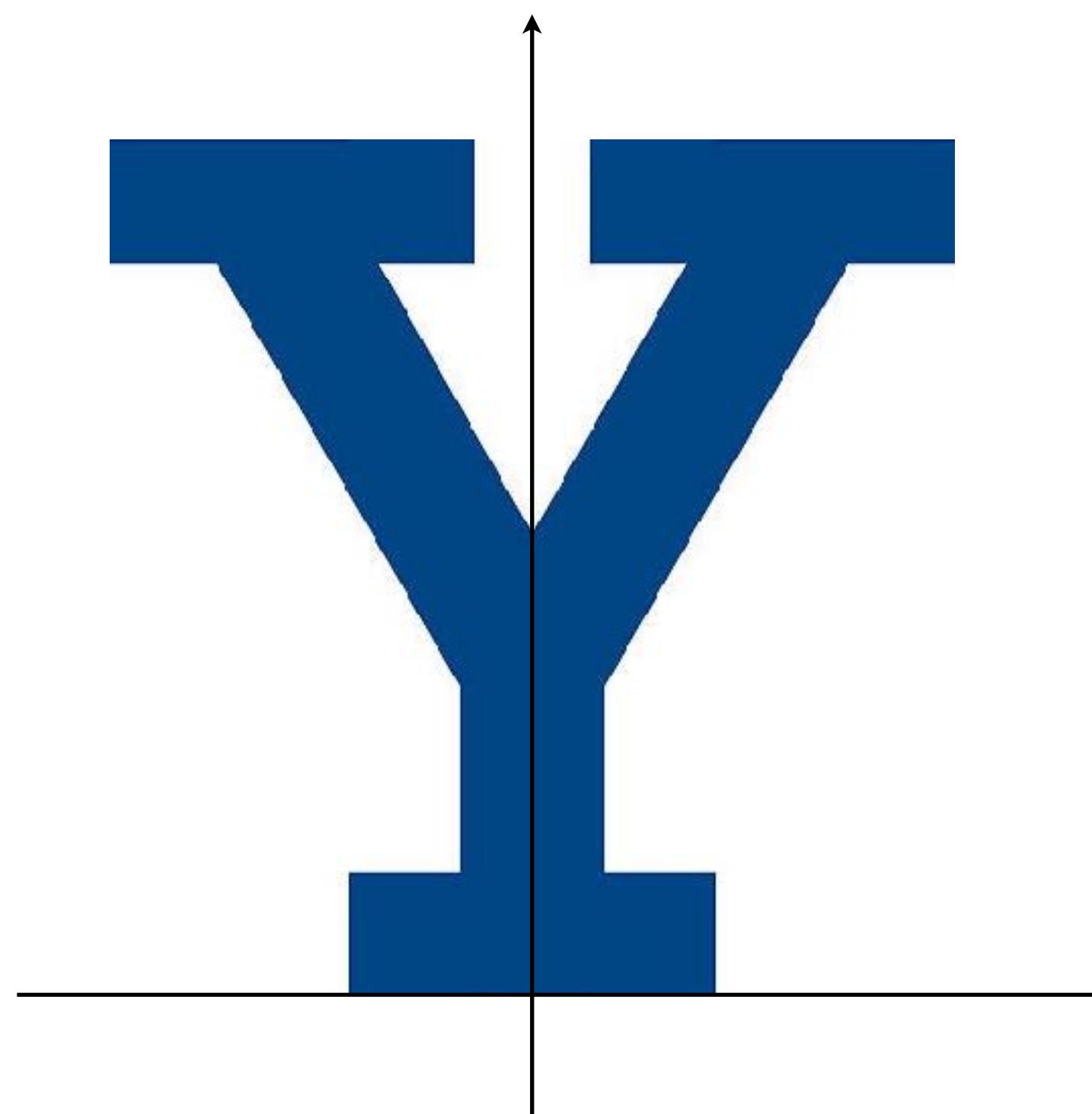
World Space

- The “world space” defines the space in which objects can live
- Choice of origin and coordinate system is arbitrary



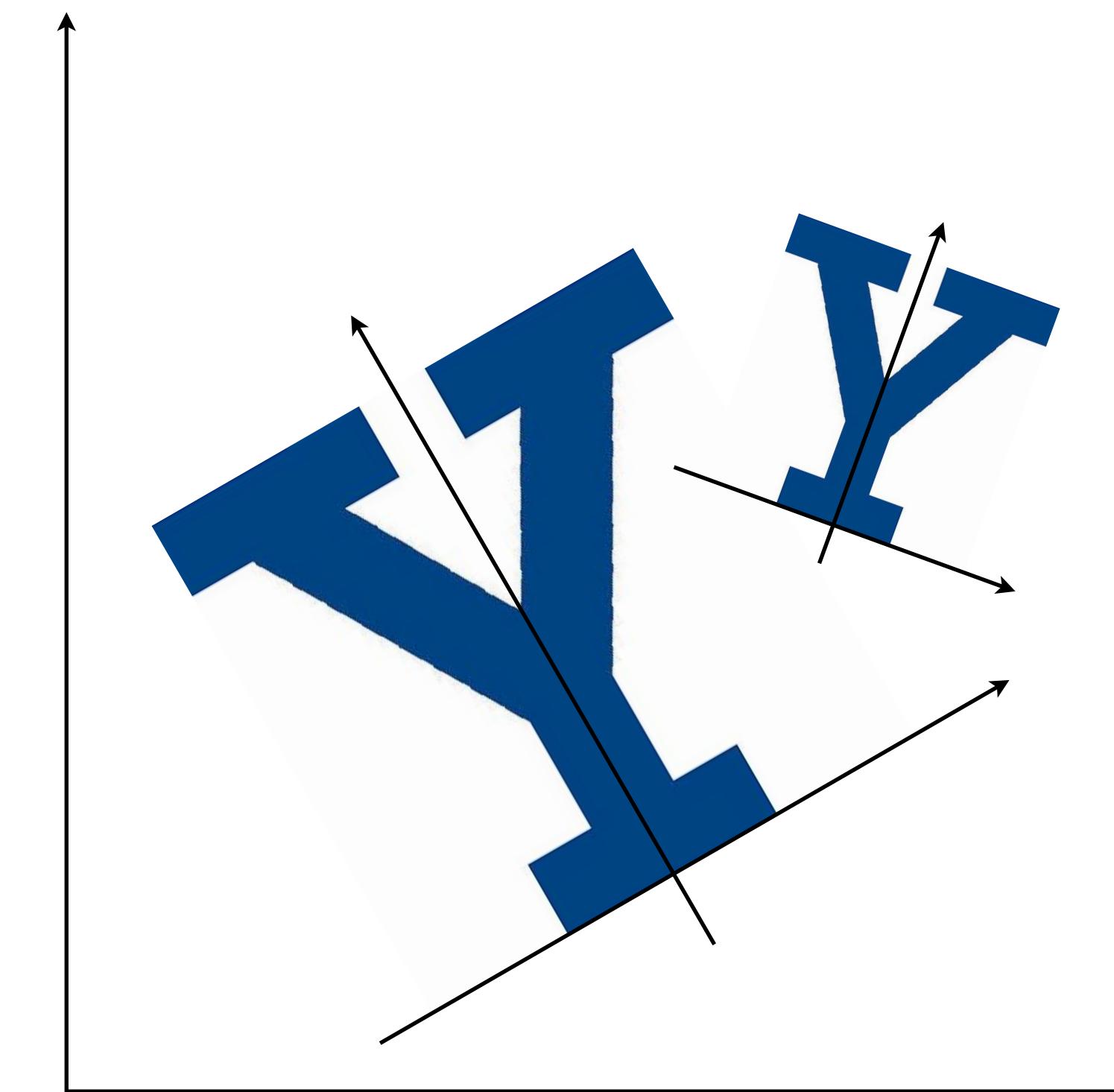
Object Space

- The coordinate system used to define an object
- Choice of origin and coordinate axes also arbitrary
- But usually chosen to make object definition the simplest



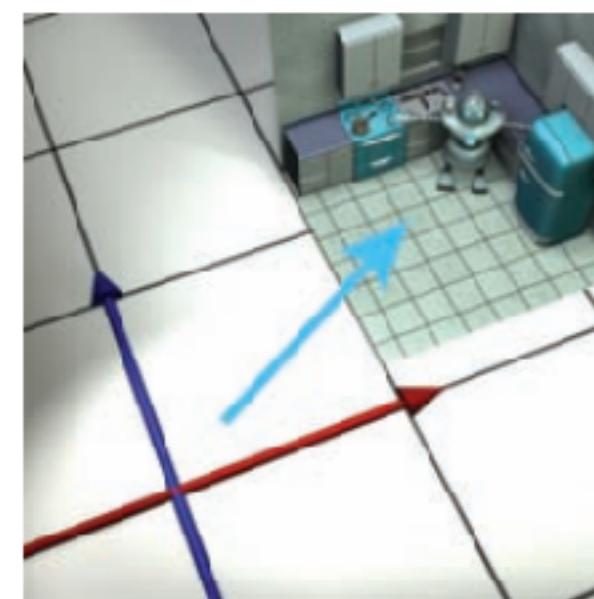
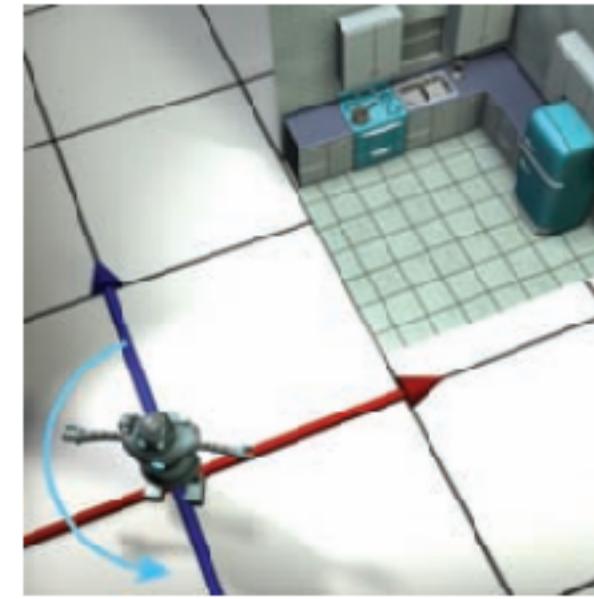
Objects in the World

- Placing an object in the world:
 - Size
 - Orientation
 - Location
- These define an *object-to-world transformation*



Object to World

- An object has a *position*, a *size*, and an *orientation*
 - First: **scale** in object space if needed (easiest if the coordinate axes are the natural directions for scaling)
 - Second: **rotate** in object space to desired world-space orientation
 - Third: **translate** (move) to the position in world space



Order matters!

Now back to rendering...

Rendering Geometry Pipeline

- Transform from object to world coordinates
- Transform from world to camera coordinates
- Preprocess to more efficiently handle things outside the field of view
(we're going to skip this for the moment)
- Perspective projection (to coordinates on the imaging plane)
- View transformation (to pixel coordinates on the screen)

Rendering Geometry Pipeline

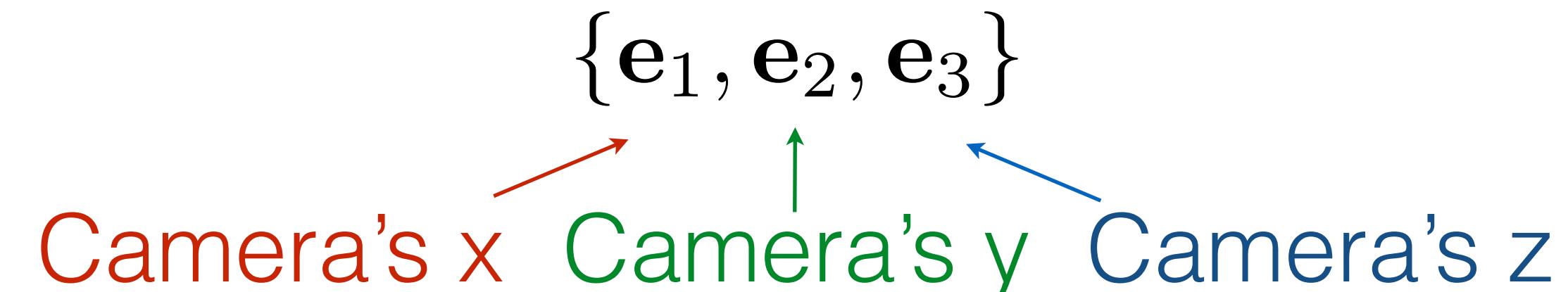
- ✓ Transform from object to world coordinates
 - Transform from world to camera coordinates
 - Preprocess to more efficiently handle things outside the field of view
(we're going to skip this for the moment)
- ✓ Perspective projection (to coordinates on the imaging plane)
 - View transformation (to pixel coordinates on the screen)

World to Camera

- Suppose that you know
 - Position of camera in world coordinates

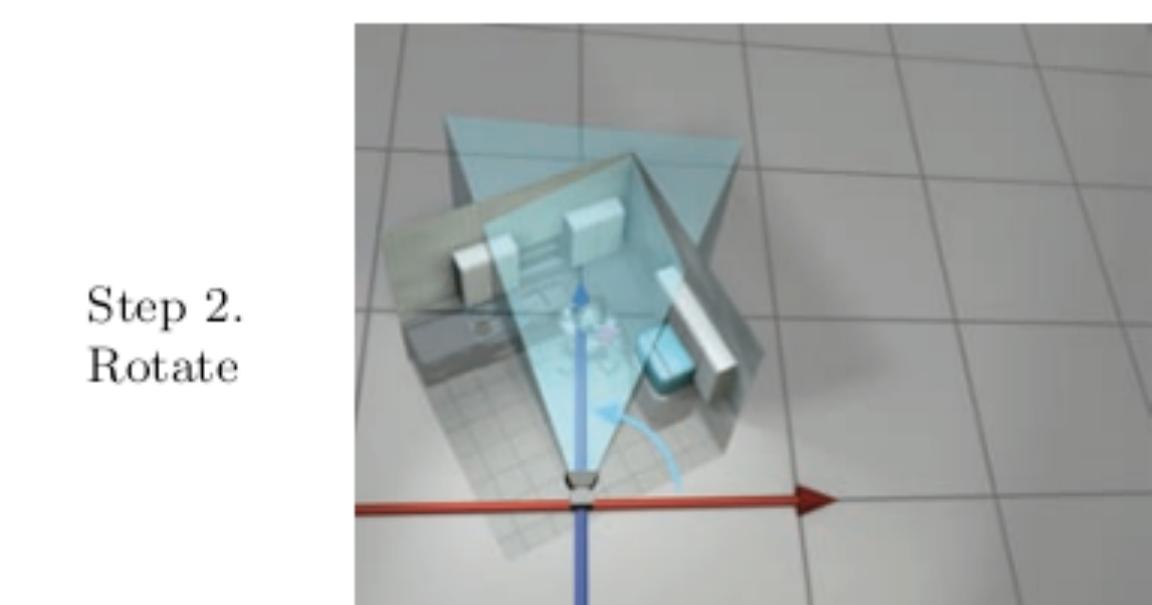
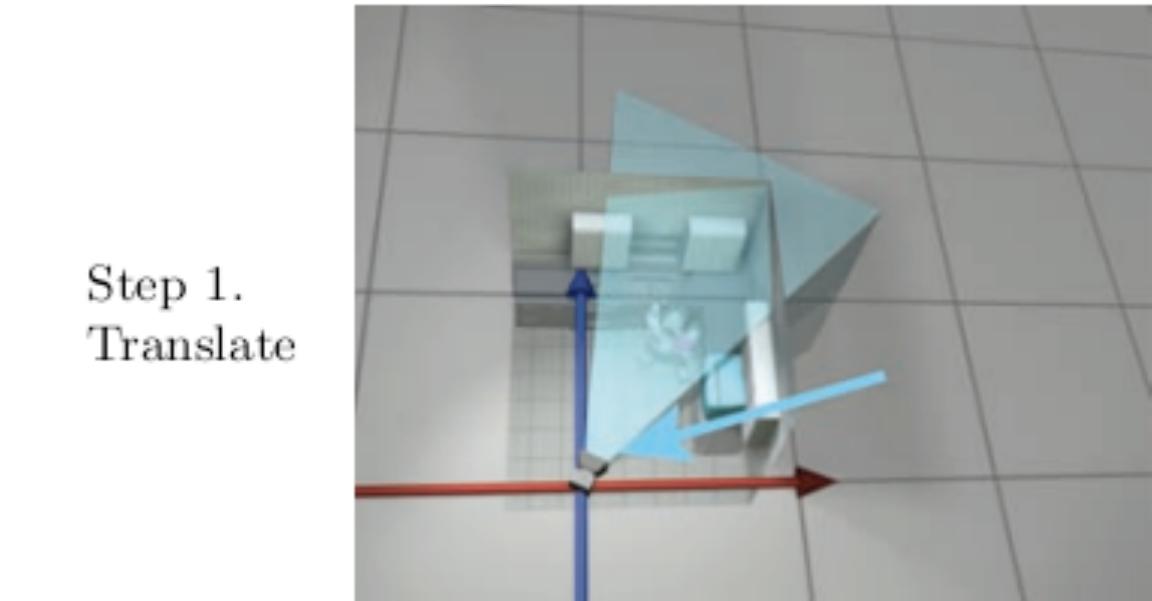
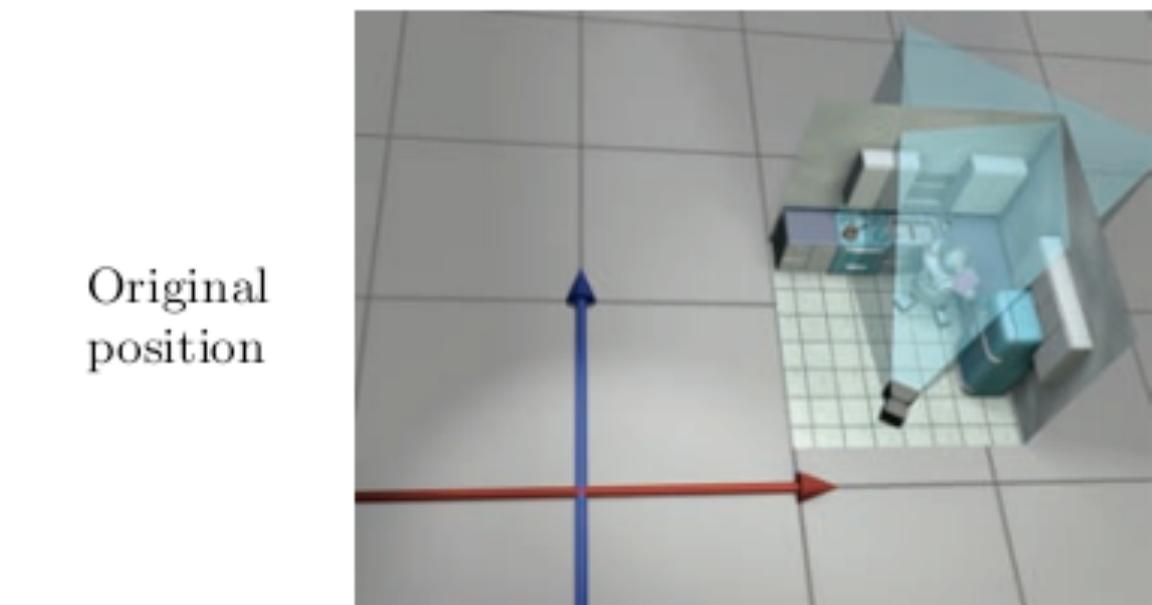
$$\mathbf{c} = (c_x, c_y, c_z)$$

- Orientation of camera as given by a set of basic vectors in world coordinates



World to Camera

- Two steps:
 - **Translate** everything to be relative to the camera position
 - **Rotate** into the camera's viewing orientation



World to Camera

- Two steps:
 - **Translate** everything to be relative to the camera position
 - **Rotate** into the camera's viewing orientation

$$\begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} e_{11} & e_{12} & e_{13} & 0 \\ e_{21} & e_{22} & e_{23} & 0 \\ e_{31} & e_{32} & e_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Camera's x
Camera's y
Camera's z

Putting It Together With Projection

World-to-camera transformation

$$\begin{bmatrix} x \\ y \\ f \\ 1 \end{bmatrix} \sim \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ Z_c/f \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix} \begin{bmatrix} e_{11} & e_{12} & e_{13} & 0 \\ e_{21} & e_{22} & e_{23} & 0 \\ e_{31} & e_{32} & e_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

Normalize Project Rotate Translate

Rendering Geometry

- ✓ Transform from object to world coordinates
- ✓ Transform from world to camera coordinates
- Preprocess to more efficiently handle things outside the field of view
(we're going to skip this for the moment)
- ✓ Perspective projection (to coordinates on the imaging plane)
- View transformation (to pixel coordinates on the screen)

To Screen Space

- Perspective projection gives you projected coordinates on the imaging plane
 - real-world units
 - centered at the focal point (intersection of the optical axis)
- We want actual on-screen pixel coordinates
- Simple transformation:
 - Multiply by the sampling density (pixels per real-world unit)
Need to specify the resolution of the image we're trying to render
 - Translate the origin to the upper left corner

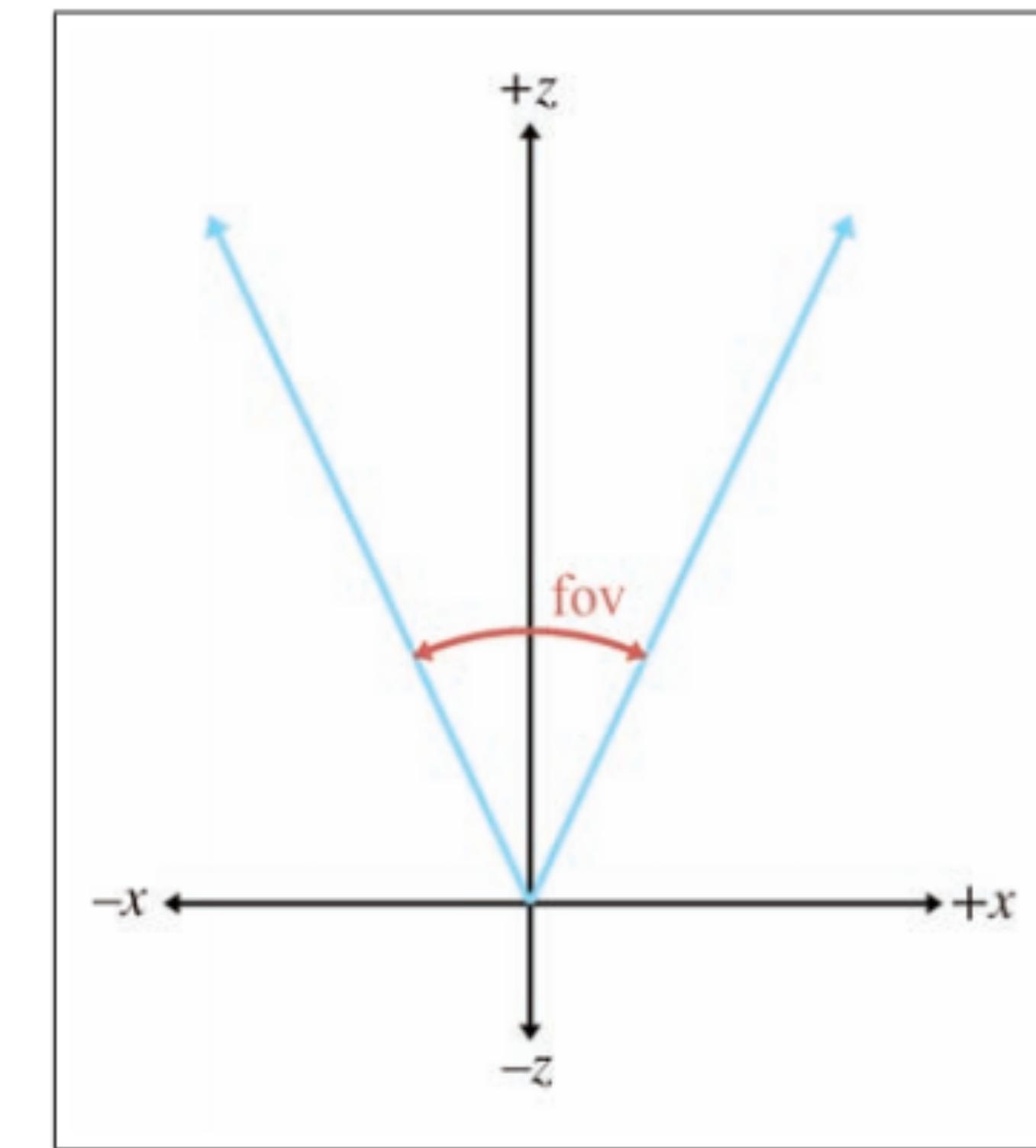
In efficient practice, there's a bit more — we'll come back to this in more detail later...

Rendering Geometry

- ✓ Transform from object to world coordinates
- ✓ Transform from world to camera coordinates
- Preprocess to more efficiently handle things outside the field of view
(we're going to skip this for the moment)
- ✓ Perspective projection (to coordinates on the imaging plane)
- ✓ View transformation (to pixel coordinates on the screen)

Field of View

- All cameras have a limited field of view
- Field of view depends on the focal length
 - Zoomed in - smaller
 - Zoomed out - larger

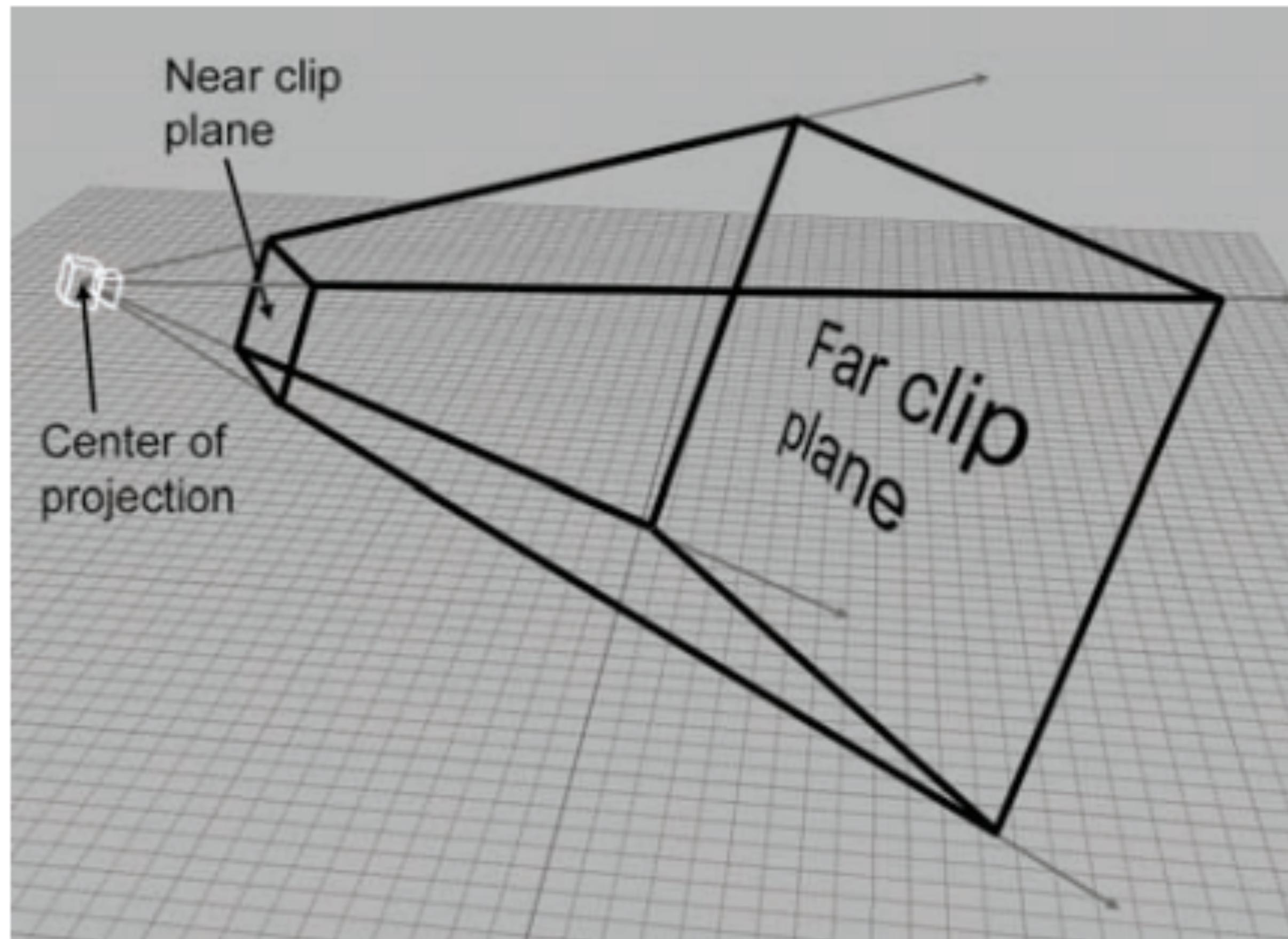


Idea: spend as little time as possible on things that are outside the field of view!

Near and Far Planes

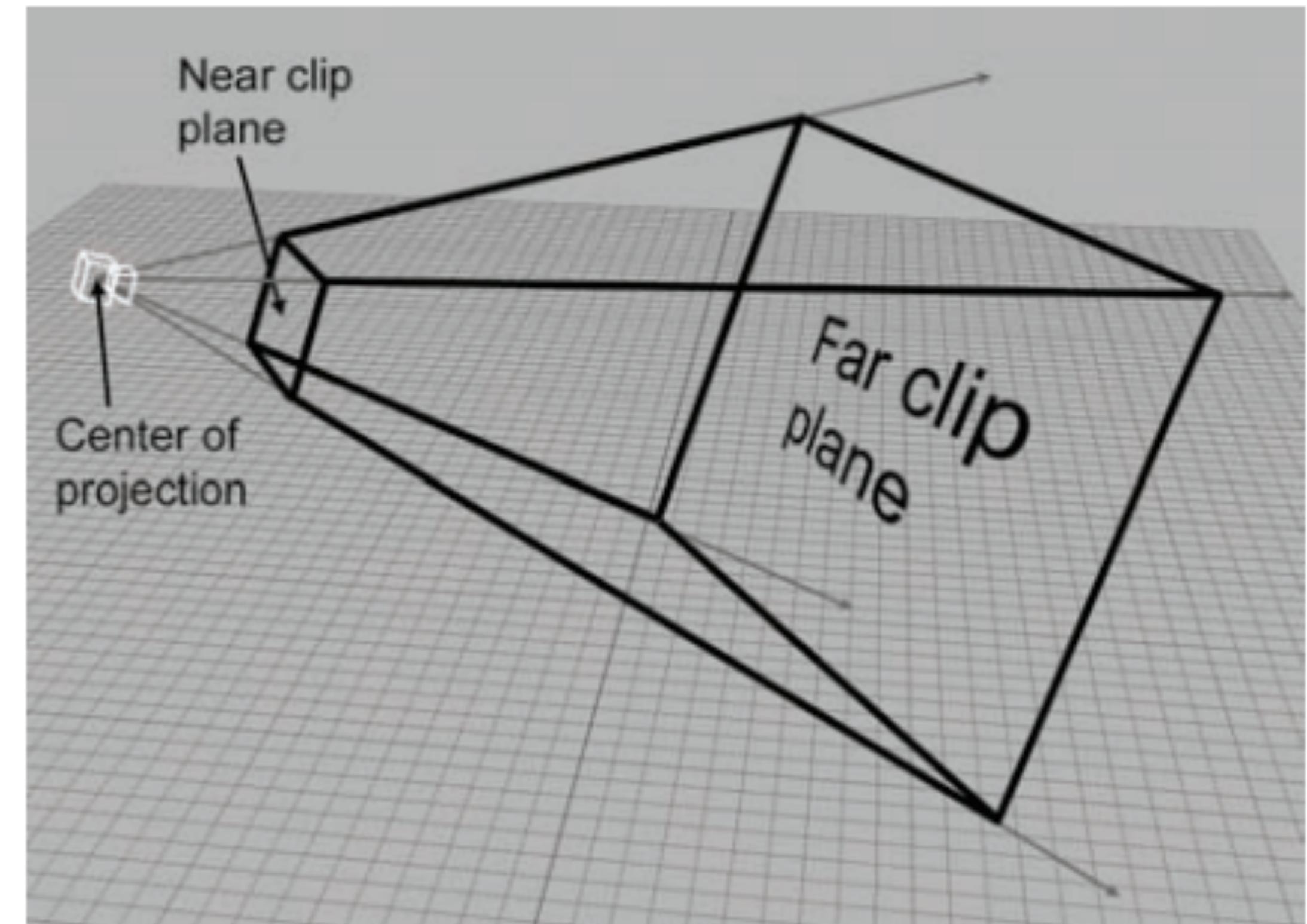
- We don't want to render things behind us, or perhaps even just barely in front of us
“Near plane”
- We don't care about things too far away to see well
“Far plane”

View Frustum

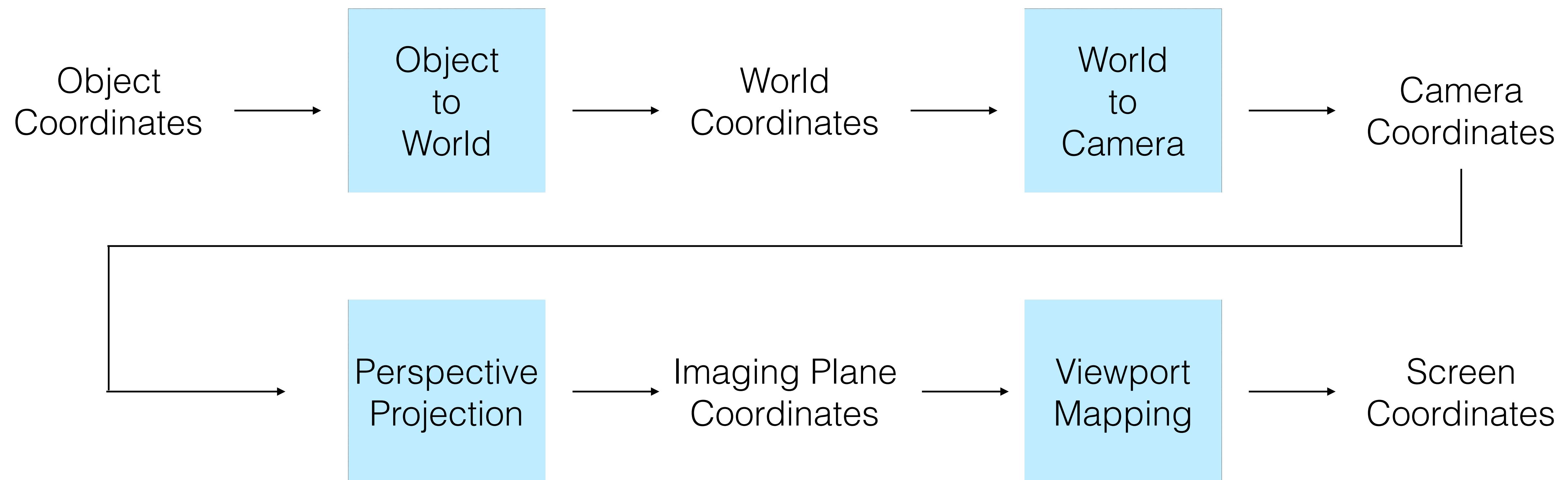


View Frustum Clipping

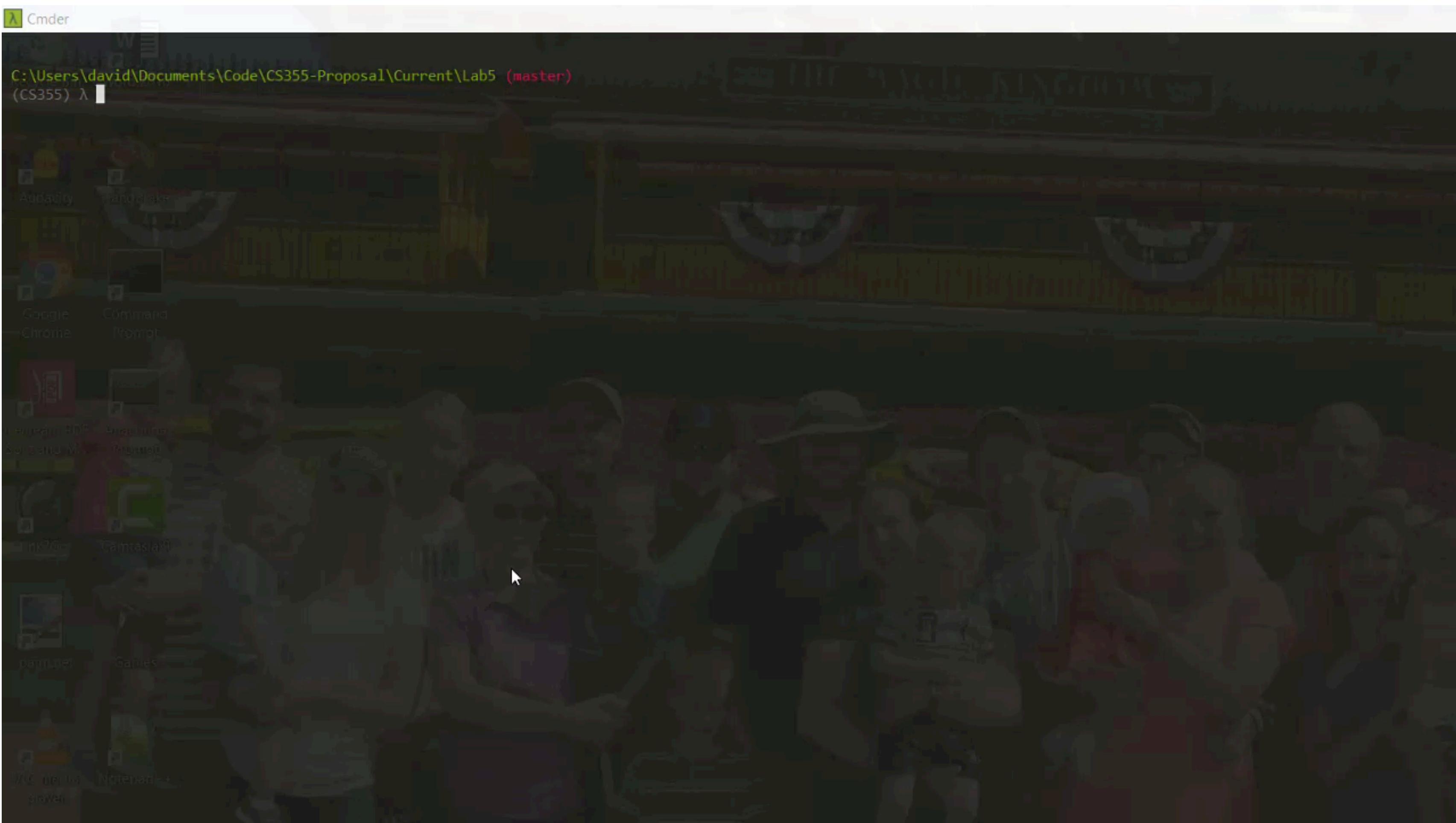
- Goal: Clip out as early as possible things that are outside the view frustum
- Idea: let's tweak our projection matrix to scale/shift things so that clipping tests are more efficient
- OpenGL will do this for you in Labs 5 and 6
- We'll go into more detail before you have to implement it yourself in Lab 7



3D Geometry Pipeline



Lab 5



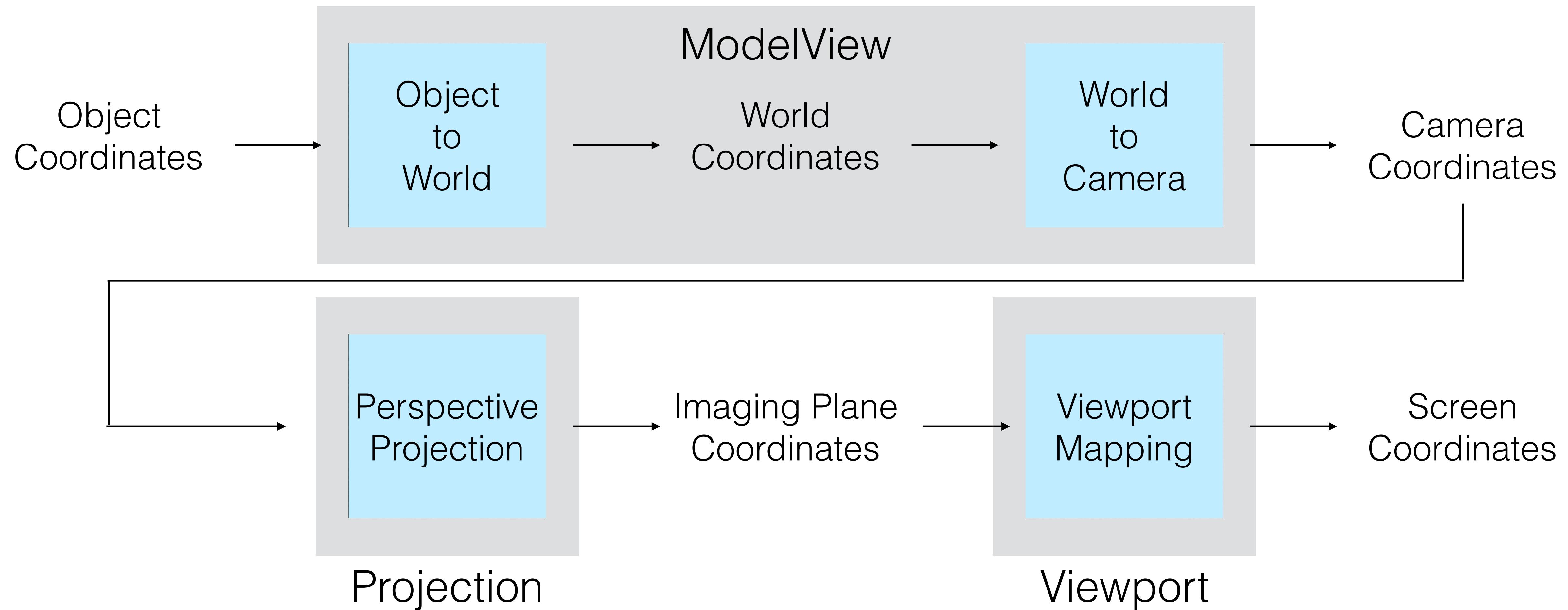
OpenGL

- OpenGL = Open Graphics Library
- Commonly used in many graphics applications
- Does the rendering for you
 - You set up the pipeline
 - You “draw” in 3D
 - It renders to the screen

OpenGL

- Key matrices in OpenGL:
 - ModelView (converts from model coordinates to camera ones)
 - Object to world
 - World to view
 - Projection
 - Orthographic
 - Perspective
 - Viewport

3D Geometry Pipeline



Lab 5: Key Functions

- glMatrixMode - changes which matrix you're manipulating
- glLoadIdentity - loads the identity matrix as the current one
- glRotated - concatenates a rotation matrix to the current one
- glTranslated - concatenates a translation matrix to the current one
- glOrtho - loads an orthographic projection matrix
- gluPerspective - loads a perspective projection matrix

Concatenating Matrices

- OpenGL concatenates new rotation/
translation operations to the **right** of the
current one $M = RT$
- Read **right-to-left** in order of **application** $M \leftarrow I$
- Build **left-to-right** in **construction** $M \leftarrow MR$

Coming up...

- Hierarchical transformations
- 3D rendering geometry
 - More details
 - Efficient implementation
- Visibility
- Lighting

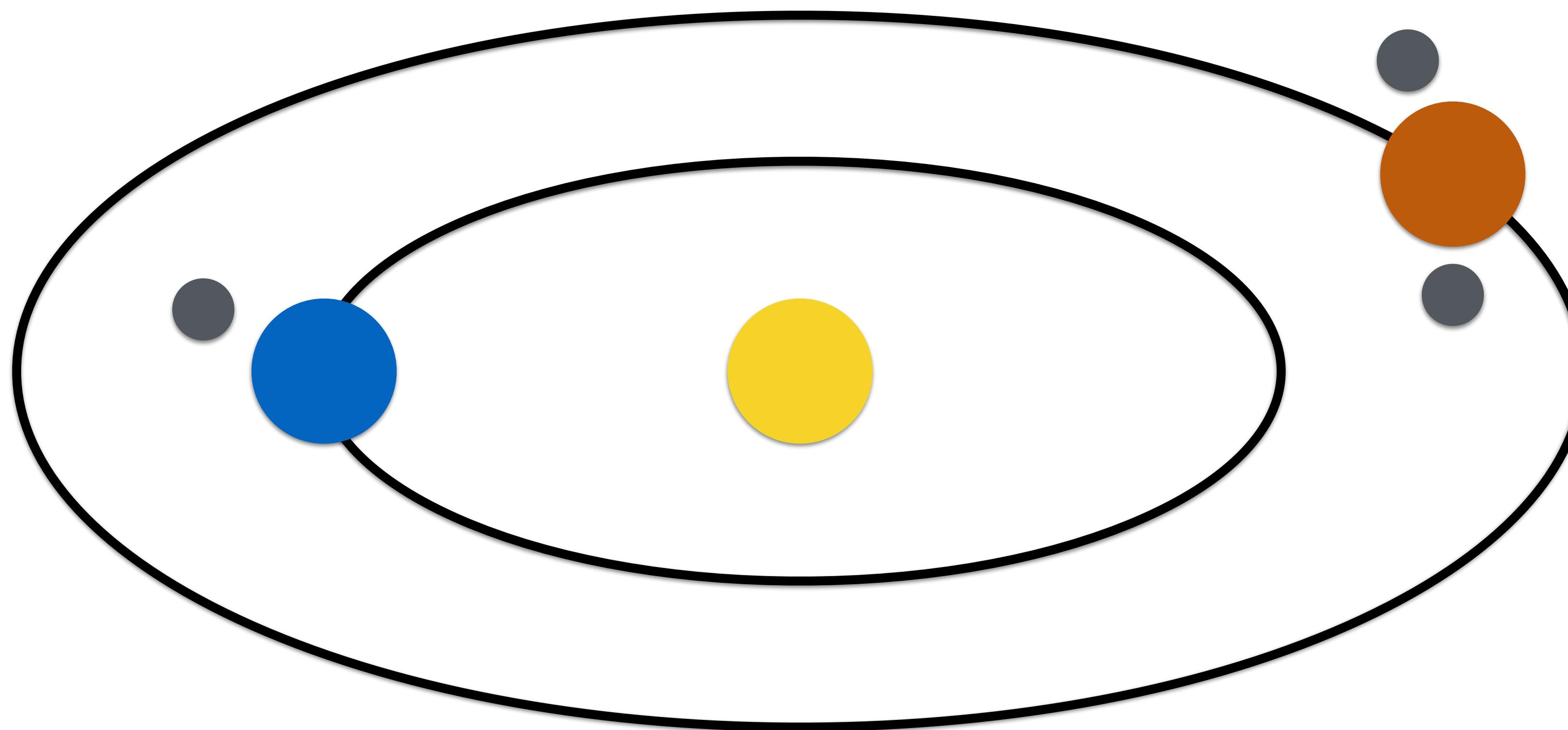


Transformation Hierarchies

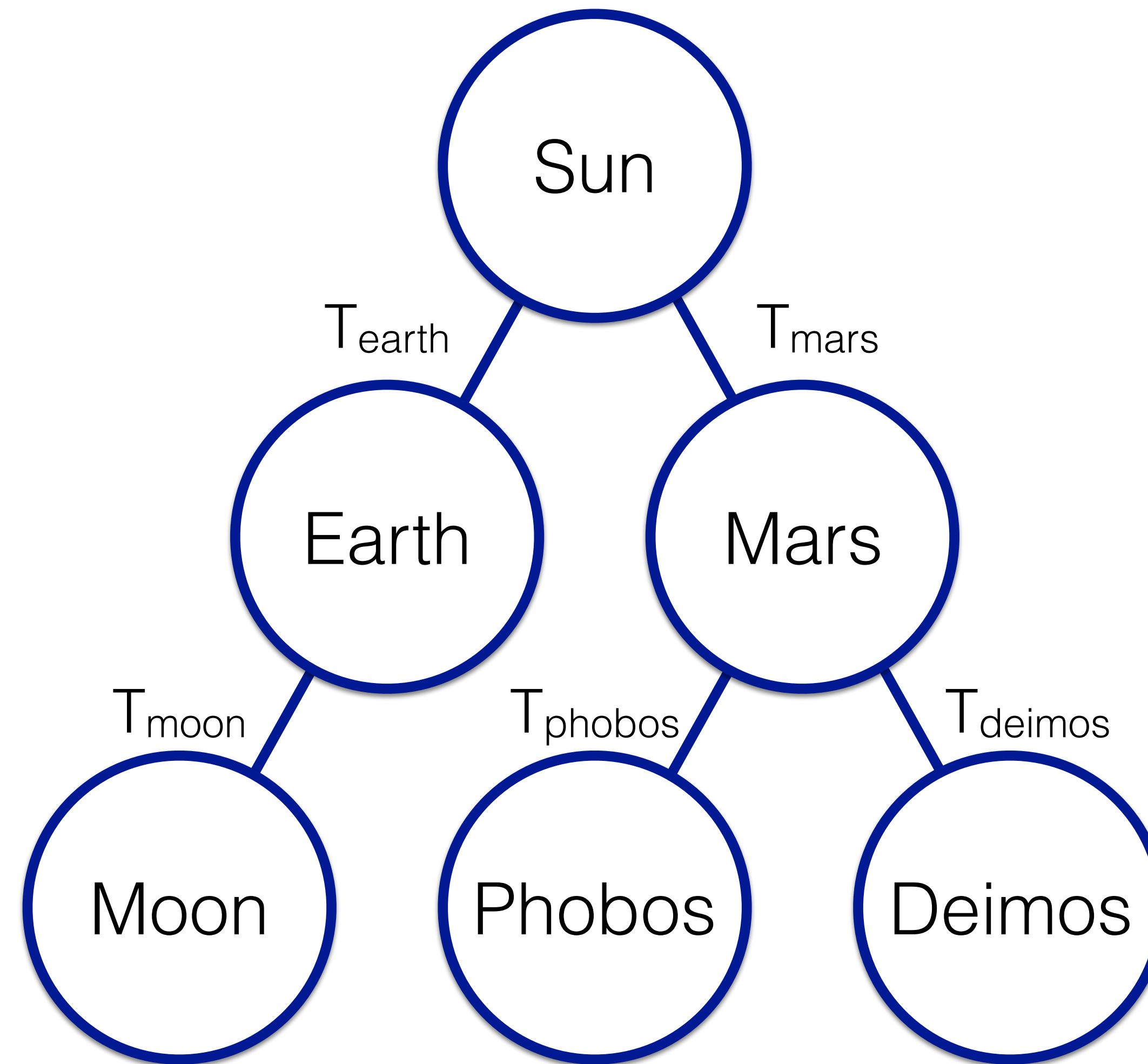
CS 355: Introduction to Graphics and Image Processing



Object Hierarchies



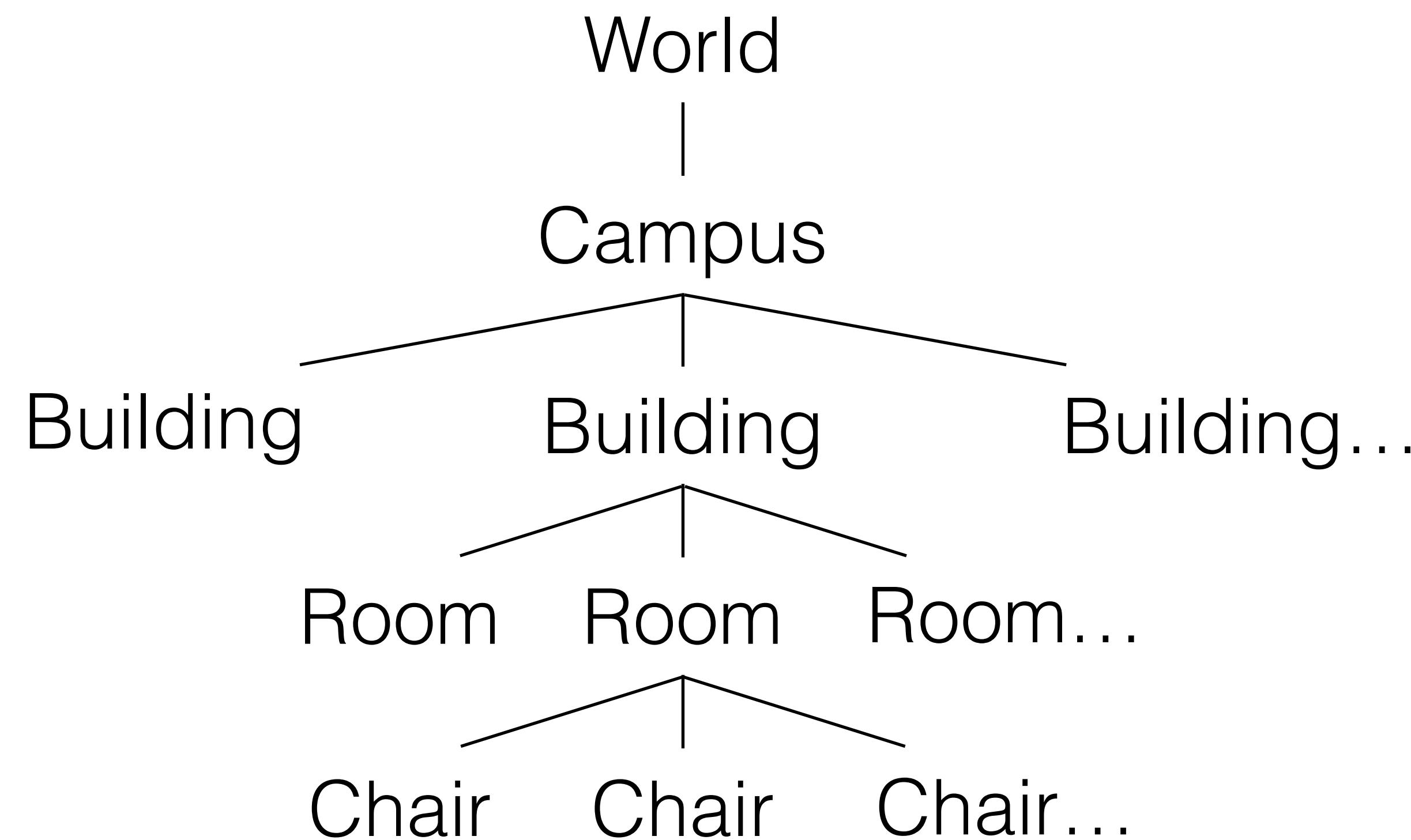
Transformation Hierarchies



Hierarchical Models

- Use same model for each chair in the classroom
 - each has a different orientation and position in the room
 - which is oriented and positioned in the building
 - which is oriented and positioned on campus
 - which is oriented and positioned in the world

Hierarchical Models



Order of Transformations

Chair 1: $\mathbf{p}_{\text{world}} = \mathbf{T}_{\text{campus}} \mathbf{T}_{\text{building}} \mathbf{T}_{\text{room}} \mathbf{T}_{\text{chair}_1} \mathbf{p}$

Chair 2: $\mathbf{p}_{\text{world}} = \mathbf{T}_{\text{campus}} \mathbf{T}_{\text{building}} \mathbf{T}_{\text{room}} \mathbf{T}_{\text{chair}_2} \mathbf{p}$

Chair 3: $\mathbf{p}_{\text{world}} = \mathbf{T}_{\text{campus}} \mathbf{T}_{\text{building}} \mathbf{T}_{\text{room}} \mathbf{T}_{\text{chair}_3} \mathbf{p}$

room-to-world

chair

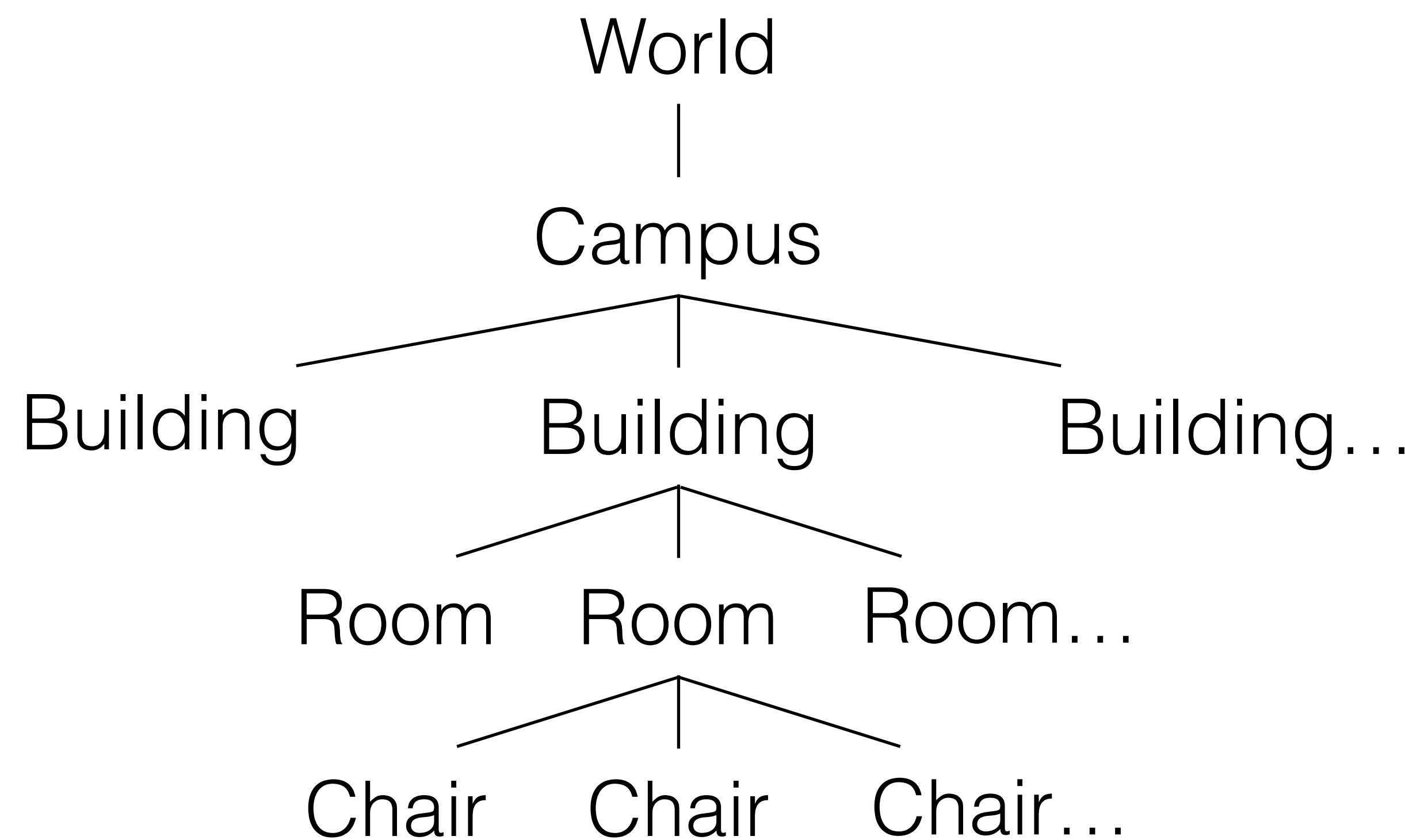
Leveraging Composition

Chair 1: $\mathbf{p}_{\text{world}} = (((\mathbf{T}_{\text{campus}} \mathbf{T}_{\text{building}}) \mathbf{T}_{\text{room}}) \mathbf{T}_{\text{chair}_1}) \mathbf{p}$

Chair 2: $\mathbf{p}_{\text{world}} = (((\mathbf{T}_{\text{campus}} \mathbf{T}_{\text{building}}) \mathbf{T}_{\text{room}}) \mathbf{T}_{\text{chair}_2}) \mathbf{p}$

Chair 3: $\mathbf{p}_{\text{world}} = (((\mathbf{T}_{\text{campus}} \mathbf{T}_{\text{building}}) \mathbf{T}_{\text{room}}) \mathbf{T}_{\text{chair}_3}) \mathbf{p}$

Hierarchical Models



Transformation Stacks



Transformation Stacks

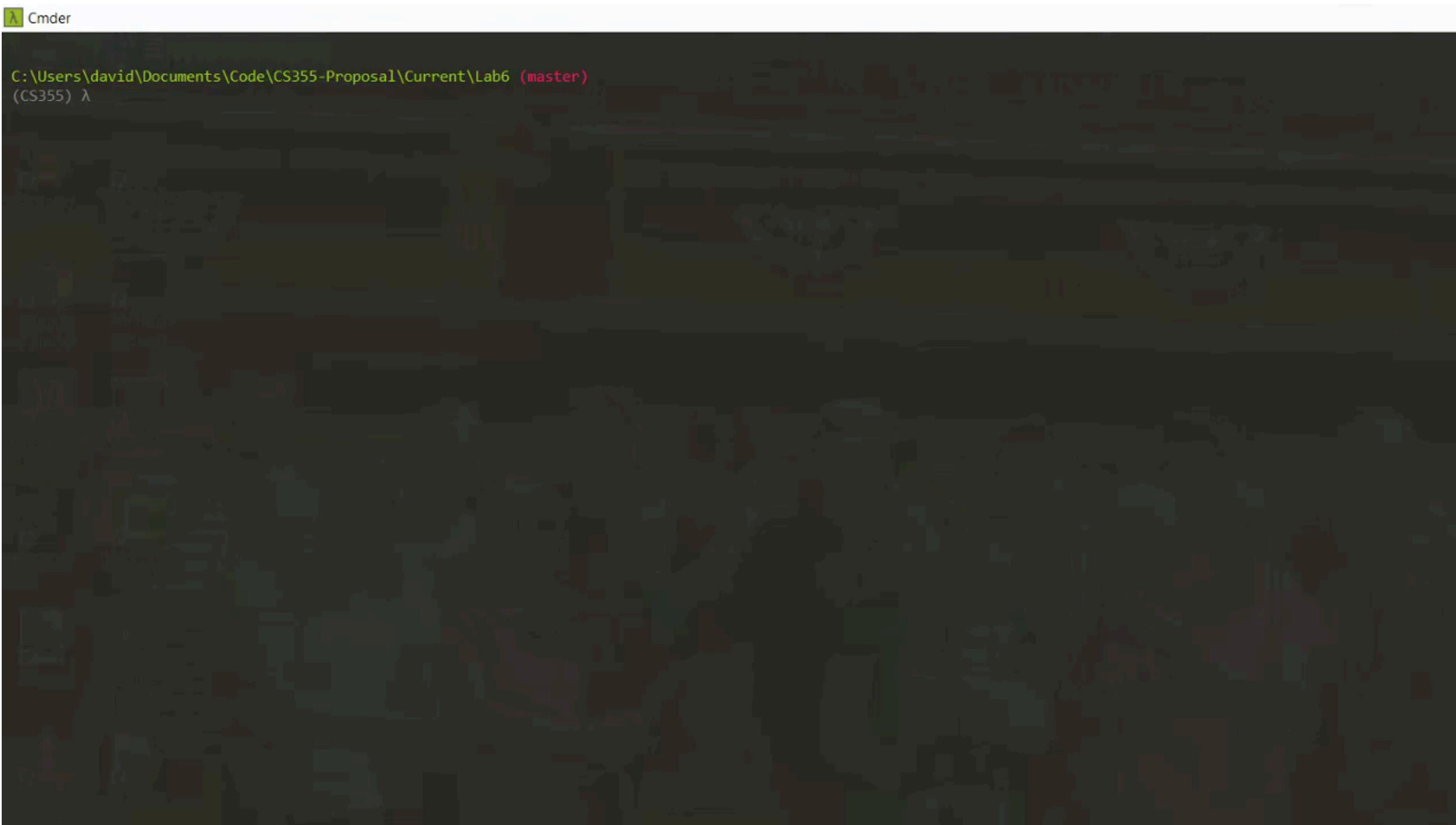
```
drawCars:  
    push()  
    For all cars i  
        drawCar(i)  
    pop()
```

```
drawCar(i):  
    push()  
    concatenate(carTransform[i])  
    drawCarBody()  
    for all tires j  
        drawTire(j)  
    pop()
```

```
drawTire(j):  
    push()  
    concatenate(tireTransform[j])  
    drawOneTire()  
    pop()
```

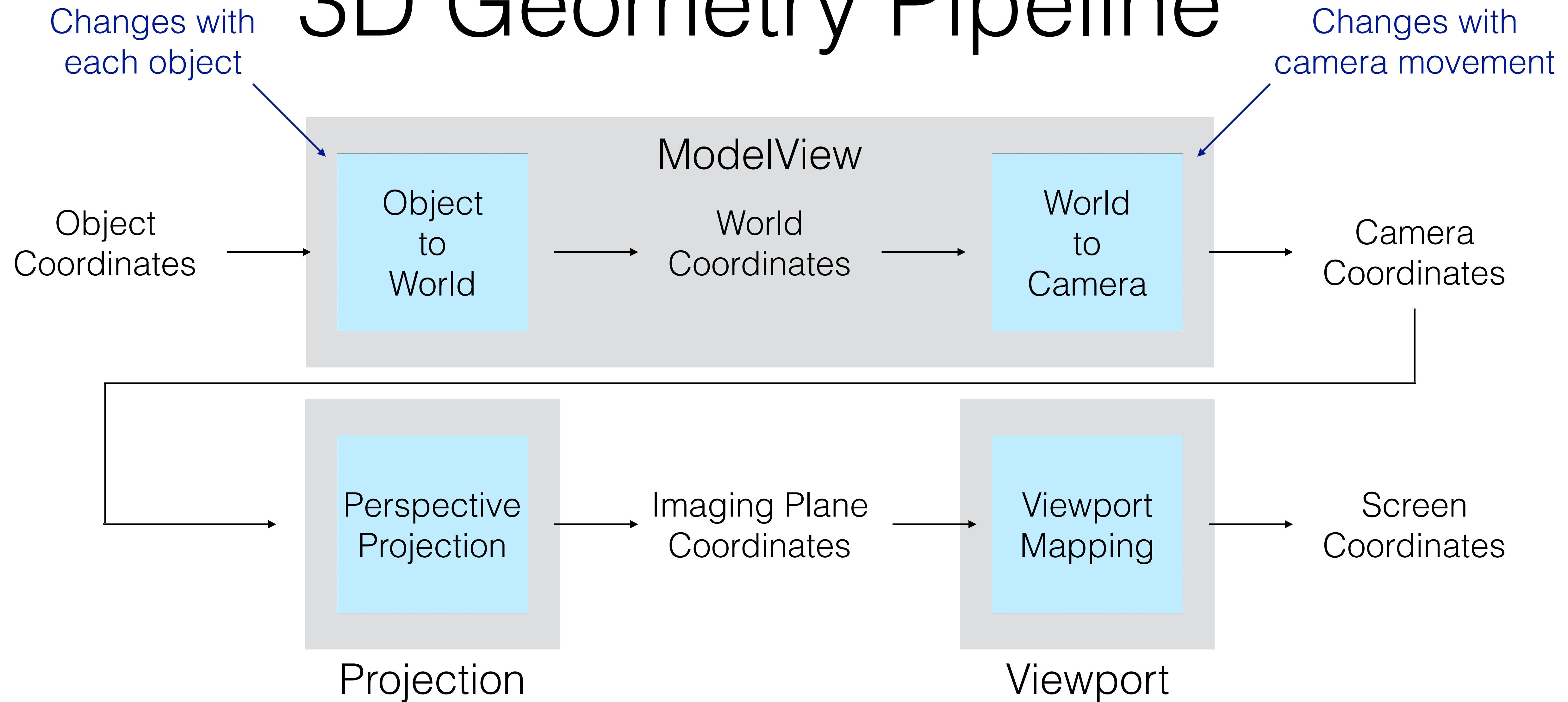


Lab 6



The image shows a terminal window titled "Cmder" with a dark background and light-colored text. At the top, it displays the path: "C:\Users\david\Documents\Code\CS355-Proposal\Current\Lab6 (master)". Below this, there is a large amount of text that is completely illegible due to heavy blurring.

3D Geometry Pipeline



Example

- Load the ModelView matrix with the desired world-to-view transformation
- Here's how you can draw one transformed object:

```
glPushMatrix()  
glTranslate(0,0,40)  
glRotatef(180,0,1,0)  
drawHouse()  
glPopMatrix()
```

- When you're done, the ModelView matrix is restored back
- Repeat for each different object
- Nest for hierarchical objects

Coming up...

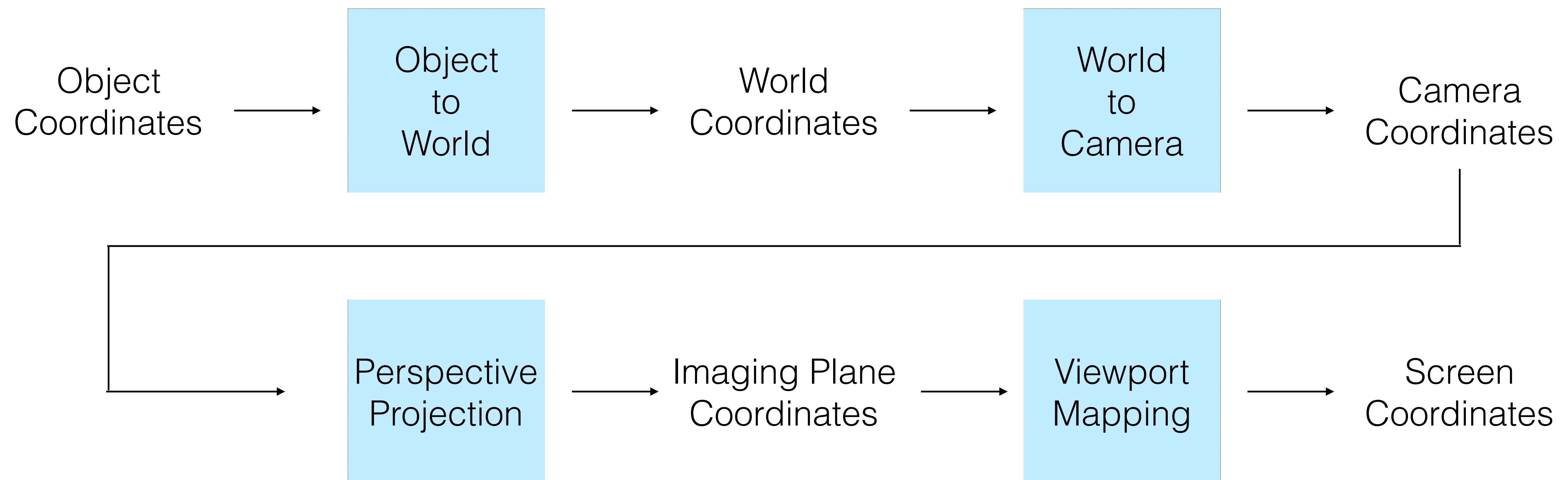
- 3D rendering geometry
 - More details
 - Efficient implementation
- Visibility
- Lighting



3D Rendering Geometry (cont'd)

CS 355: Introduction to Graphics and Image Processing

3D Geometry Pipeline



Let's revisit object placement.. .

3D Linear Transformations

- Scaling has the same form as in 2D
- Translation has the same form as in 2D
- Rotation has the same form as in 2D
if you begin with unit vectors for the new coordinate axes

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

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

$$\mathbf{R} = \begin{bmatrix} e_{11} & e_{12} & e_{13} & 0 \\ e_{21} & e_{22} & e_{23} & 0 \\ e_{31} & e_{32} & e_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3D Rotations

- Can construct rotation matrices directly from unit vectors for the new coordinate axes
 - All rows are orthogonal
 - Any matrix with orthogonal rows is a rotation!
- Can also construct from rotation angles (looks a lot like 2D rotation matrices)
 - Around x axis (in y-z plane)
 - Around y axis (in x-z plane)
 - Around z axis (in x-y plane)

$$\mathbf{R} = \begin{bmatrix} e_{11} & e_{12} & e_{13} & 0 \\ e_{21} & e_{22} & e_{23} & 0 \\ e_{31} & e_{32} & e_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

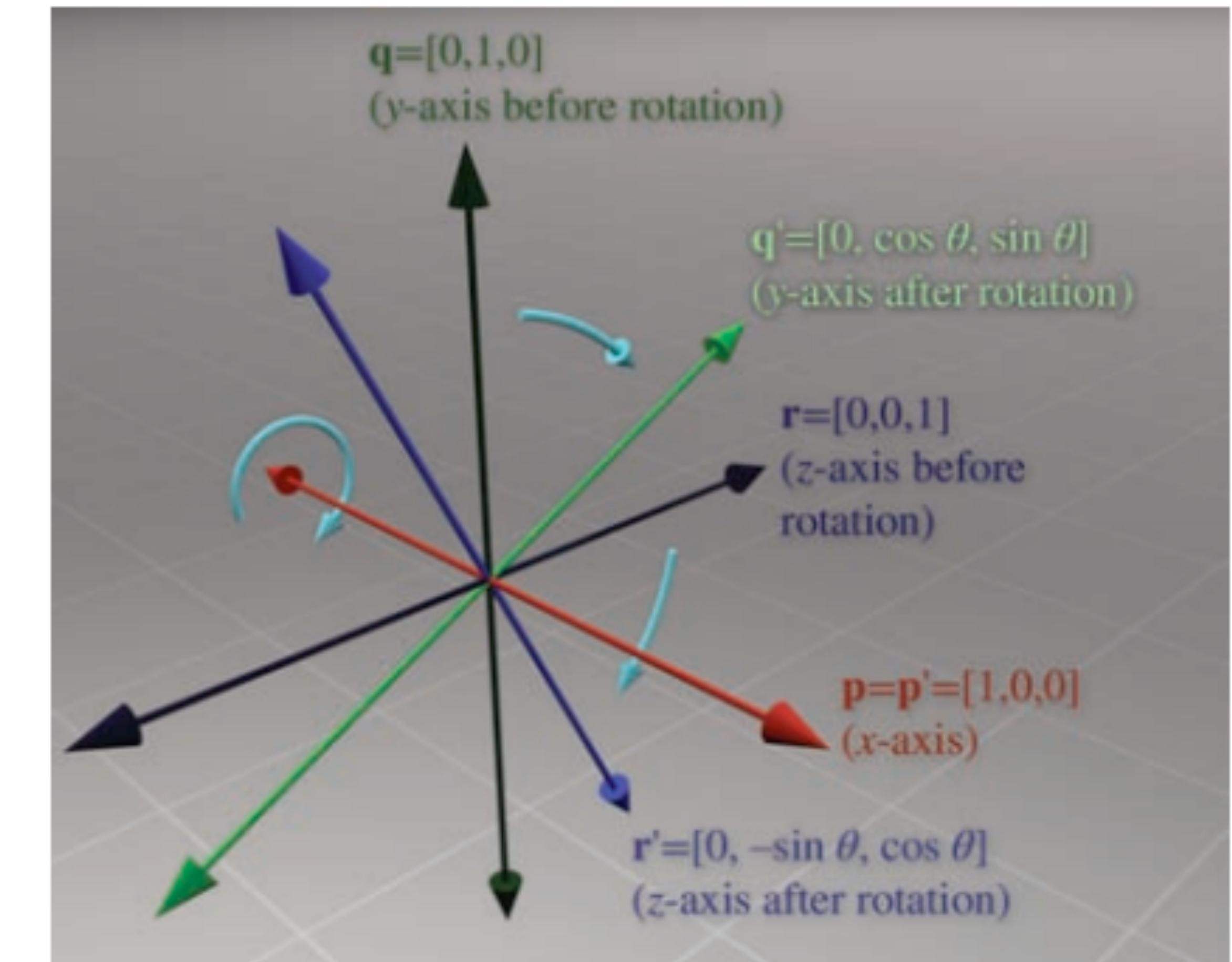
$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3D Rotations

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

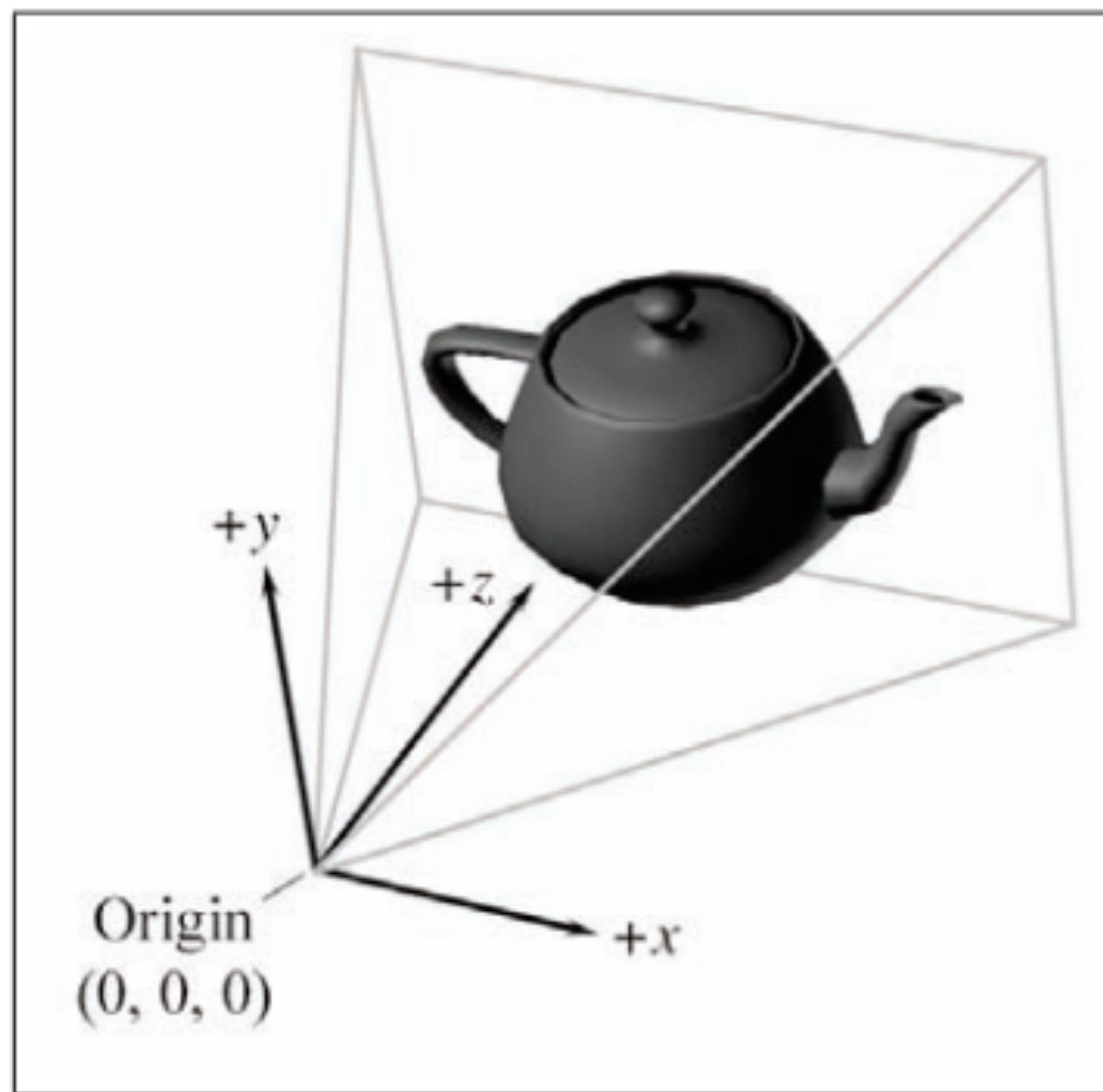
$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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



Let's revisit the camera space...

Camera Space



World to Camera

- You need to know
 - Position of camera in world coordinates $\mathbf{c} = (c_x, c_y, c_z)$
 - Orientation of camera as given by
 - a set of basic vectors in world coordinates, or
 - rotation angles

$\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$

Camera's x Camera's y Camera's z

Specifying the Camera

“Look from” point

\mathbf{p}_{from}

“Look at” point

\mathbf{p}_{at}

“Up” vector

\mathbf{v}_{up}



Roughly!

Building Coordinate System

Optical axis (Z) first:

$$\mathbf{e}_3 = \frac{\mathbf{p}_{\text{at}} - \mathbf{p}_{\text{from}}}{\|\mathbf{p}_{\text{at}} - \mathbf{p}_{\text{from}}\|}$$

Then side (X):

$$\mathbf{e}_1 = \frac{\mathbf{e}_3 \times \mathbf{v}_{\text{up}}}{\|\mathbf{e}_3 \times \mathbf{v}_{\text{up}}\|}$$

Then straighten “up” (Y):

$$\mathbf{e}_2 = \frac{\mathbf{e}_1 \times \mathbf{e}_3}{\|\mathbf{e}_1 \times \mathbf{e}_3\|}$$

“Gram - Schmidt” orthogonalization

World to Camera

- Two steps:

- **Translate**

everything to be relative to the camera position

$$\begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **Rotate**

into the camera's viewing orientation

$$\begin{bmatrix} e_{11} & e_{12} & e_{13} & 0 \\ e_{21} & e_{22} & e_{23} & 0 \\ e_{31} & e_{32} & e_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Let's revisit the pipeline...

Pipeline So Far

Idea: let's cull as much
as we can before dividing

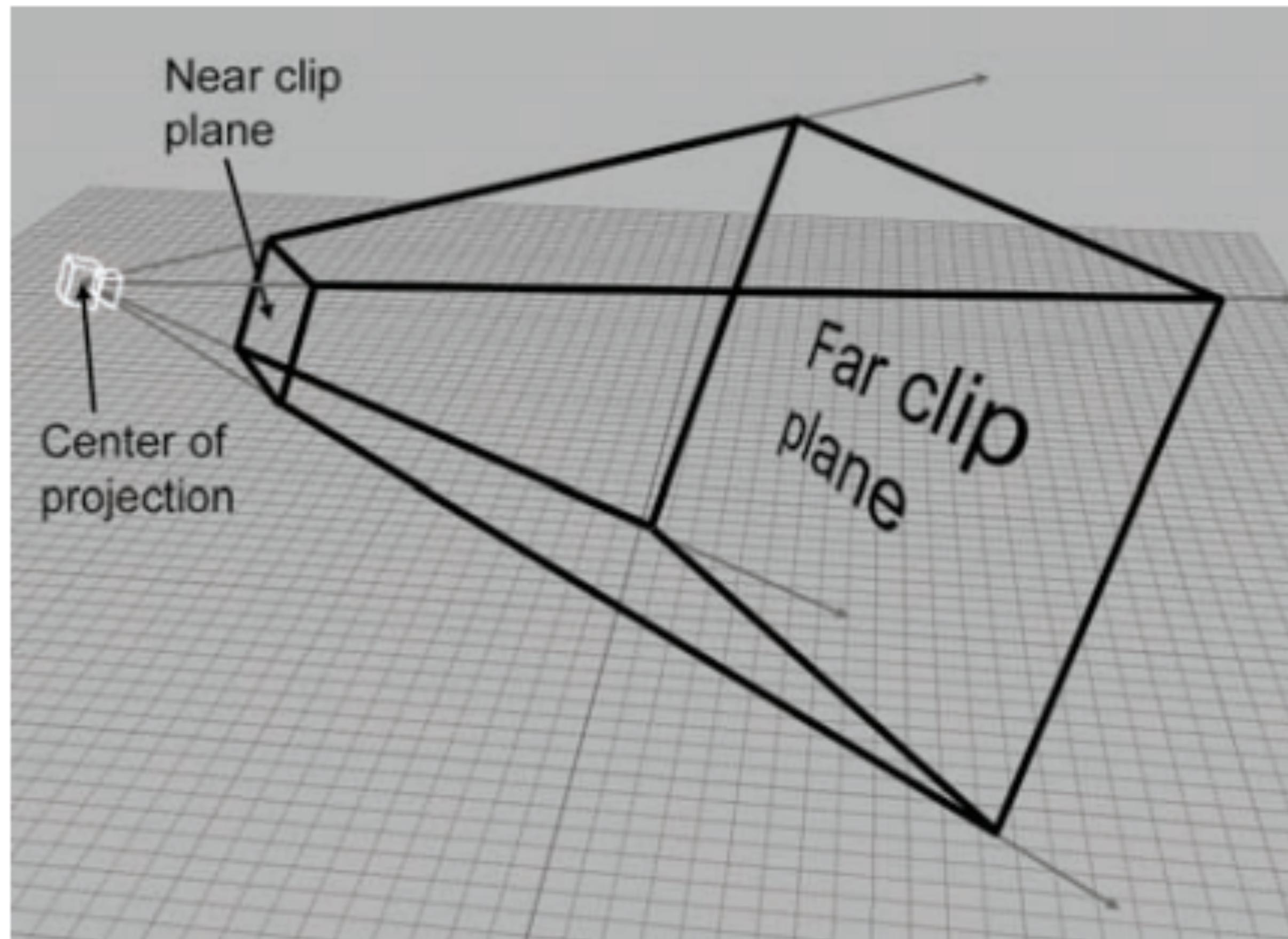
$$\begin{bmatrix} x \\ y \\ f \\ 1 \end{bmatrix} \sim \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ Z_c/f \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix} \begin{bmatrix} e_{11} & e_{12} & e_{13} & 0 \\ e_{21} & e_{22} & e_{23} & 0 \\ e_{31} & e_{32} & e_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

World-to-camera transformation

Normalize Project Rotate Translate

Big problem: lots of time spent on stuff you can't see!

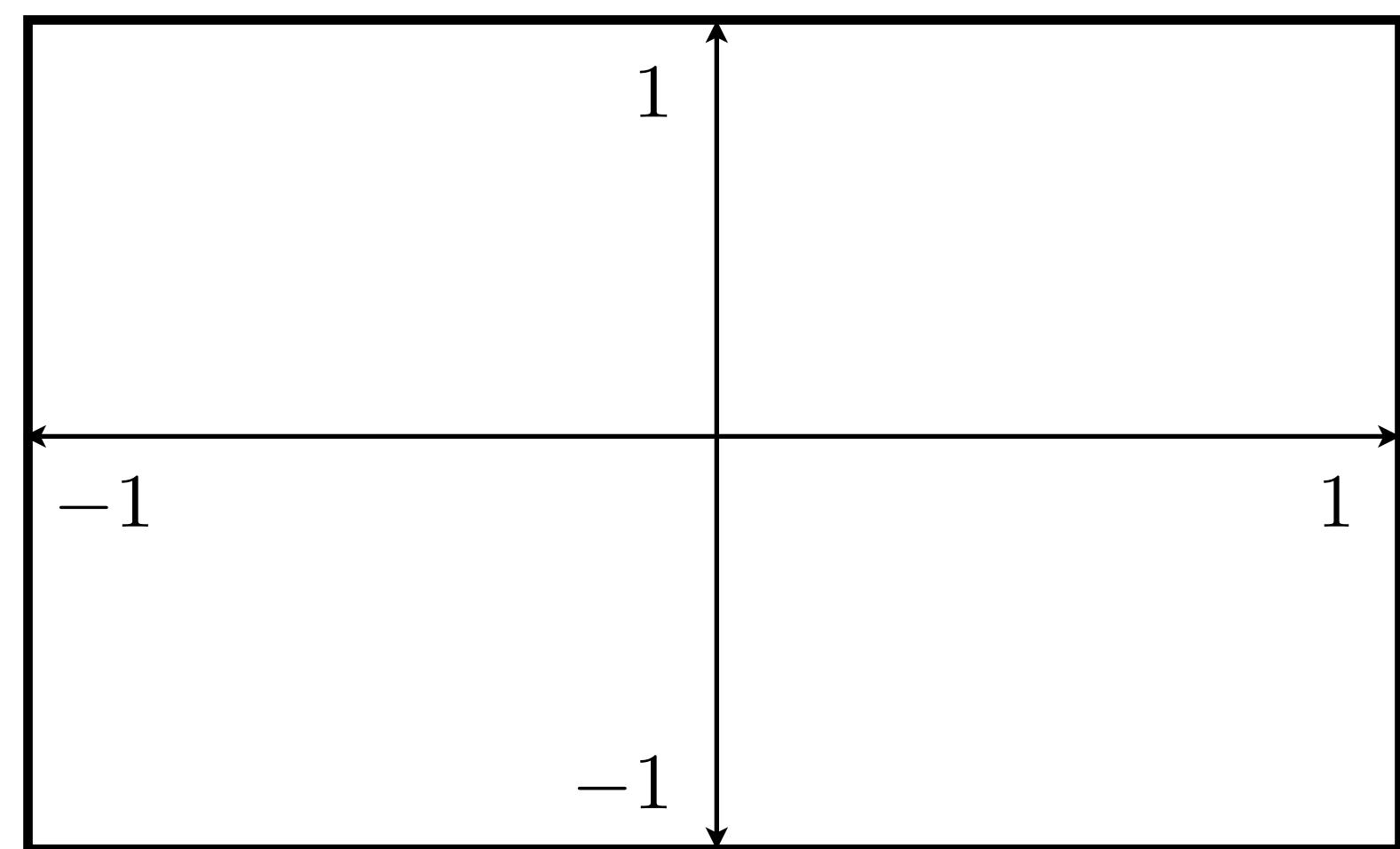
View Frustum



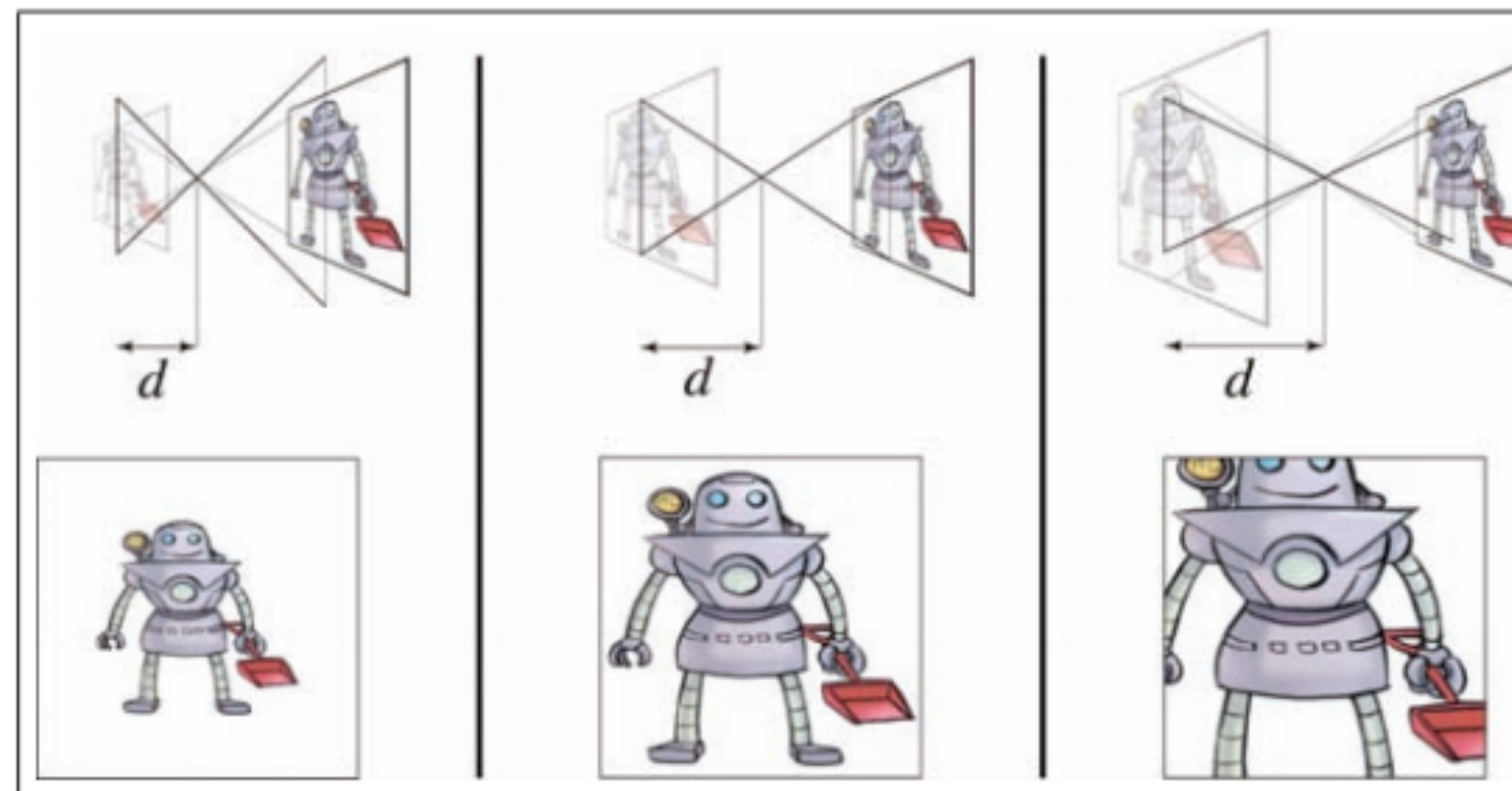
Can we clip things outside the view
frustum without doing a divide?

Canonical View

- To simplify, let's assume we map to $[-1, 1]$ in both x and y directions
- Also map [near, far] depth range to $[-1, 1]$
- Maps frustum to $[-1, 1]^3$ cube



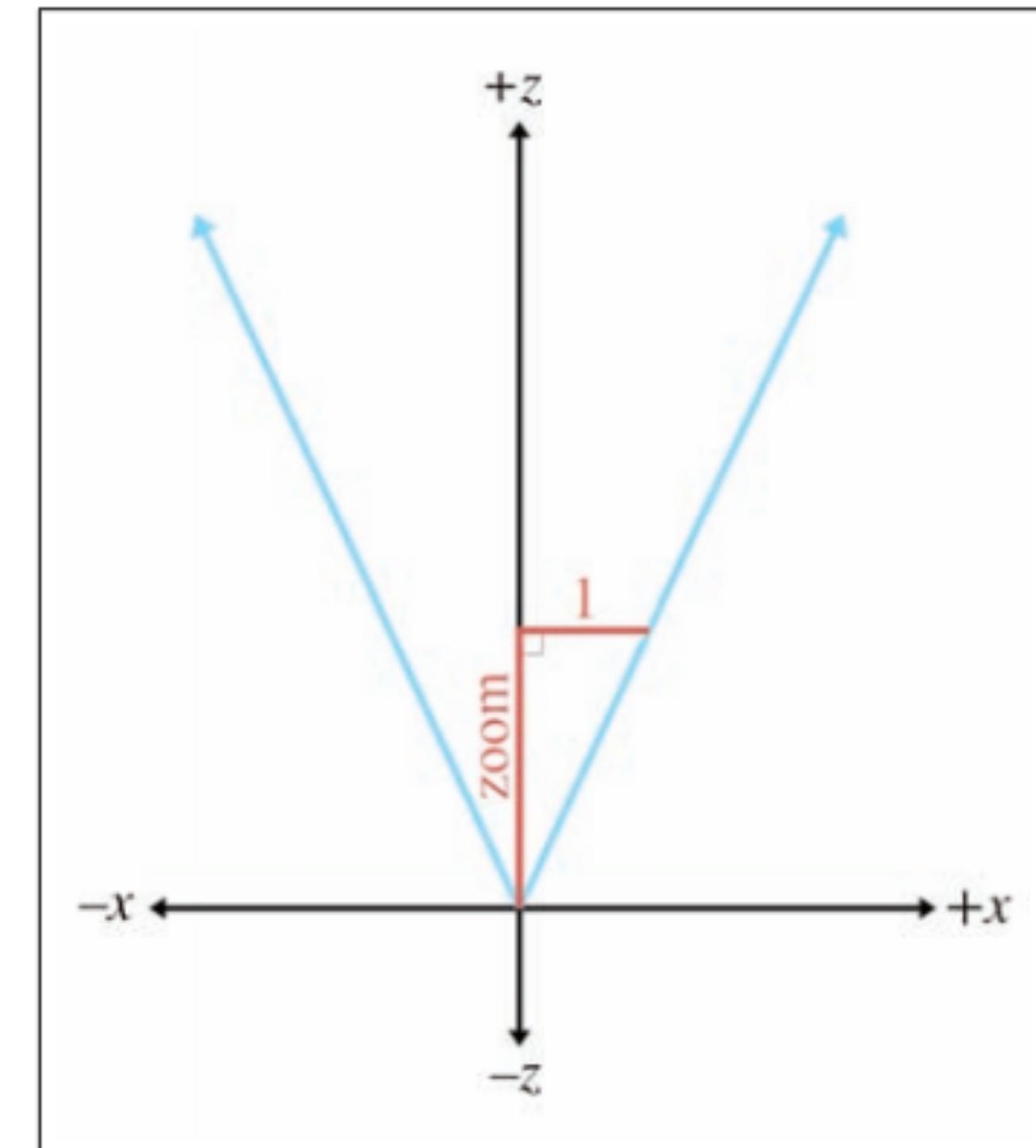
Changing Focal Length



Changing focal length changes overall zoom,
but also affects the shape of the view frustum

Zoom

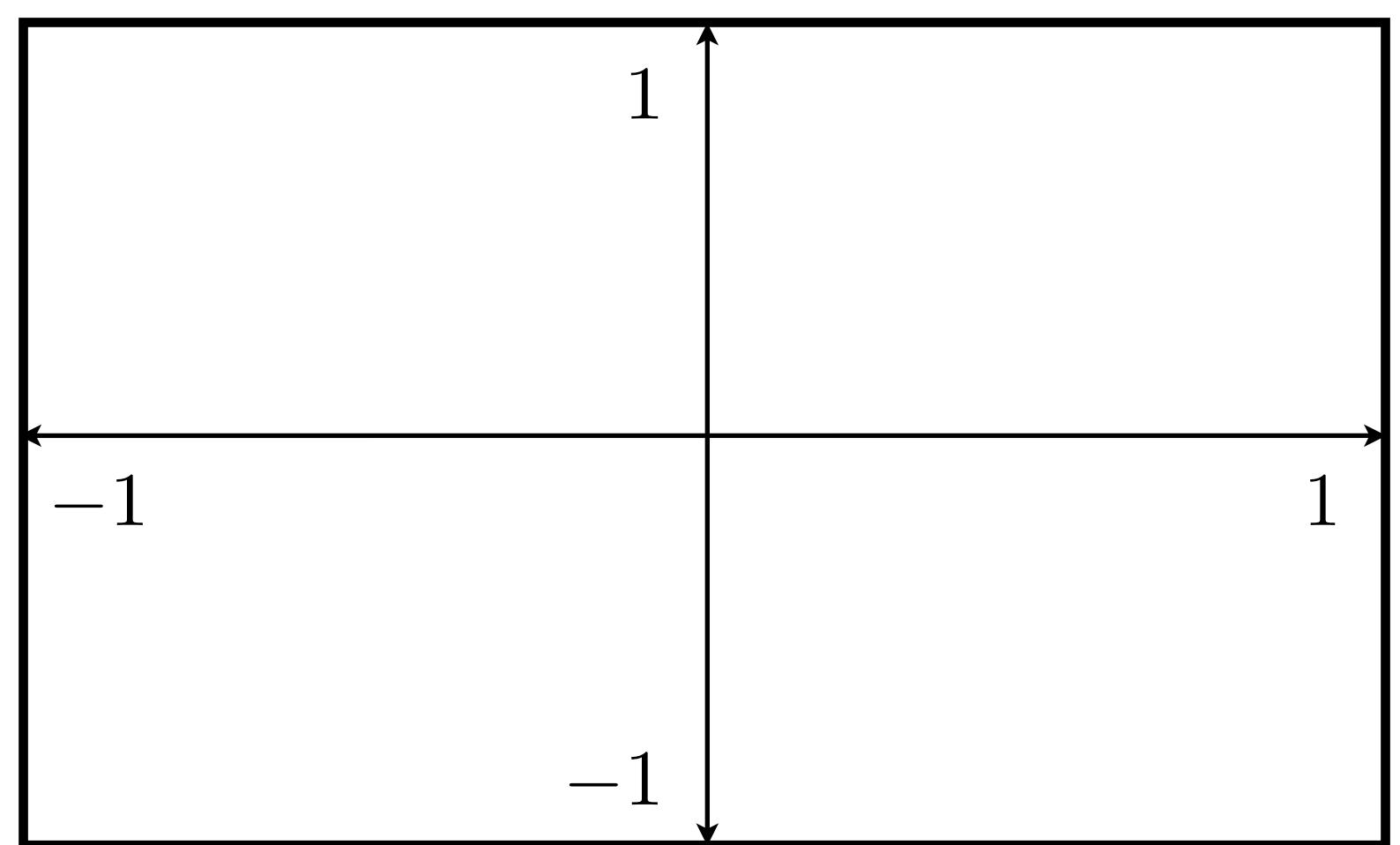
- Mapping to a canonical window loses
 - Real horizontal width
 - Real vertical width
- We'll need to fold this into our projection matrix
- Think in terms of different “zoom” levels for x and y



$$\text{zoom} = \frac{1}{\tan(\text{fov}/2)}$$

The Clip Matrix

- Let's build a new projection matrix that
 - Scales visible x to $[-1, 1]$
 - Scales visible y to $[-1, 1]$
 - Scales near to far z to $[-1, 1]$



The Clip Matrix

- Let's build a new projection matrix that

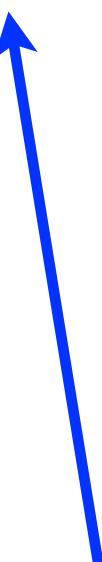
- Scales visible x to [-1,1]
- Scales visible y to [-1,1]
- Scales near to far z to [-1,1]

$$\begin{bmatrix} \text{zoom}_x & 0 & 0 & 0 \\ 0 & \text{zoom}_y & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{-2nf}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

n = near plane distance

f = far plane distance

The Clip Matrix

$$\begin{bmatrix} x/w \\ y/w \\ z/w \\ 1 \end{bmatrix} \sim \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} \text{zoom}_x & 0 & 0 & 0 \\ 0 & \text{zoom}_y & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{-2nf}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}$$


All of these are in the range [-1,1] for things in view

Clipping Tests

Left $x < -w$

Right $x > w$

Bottom $y < -w$

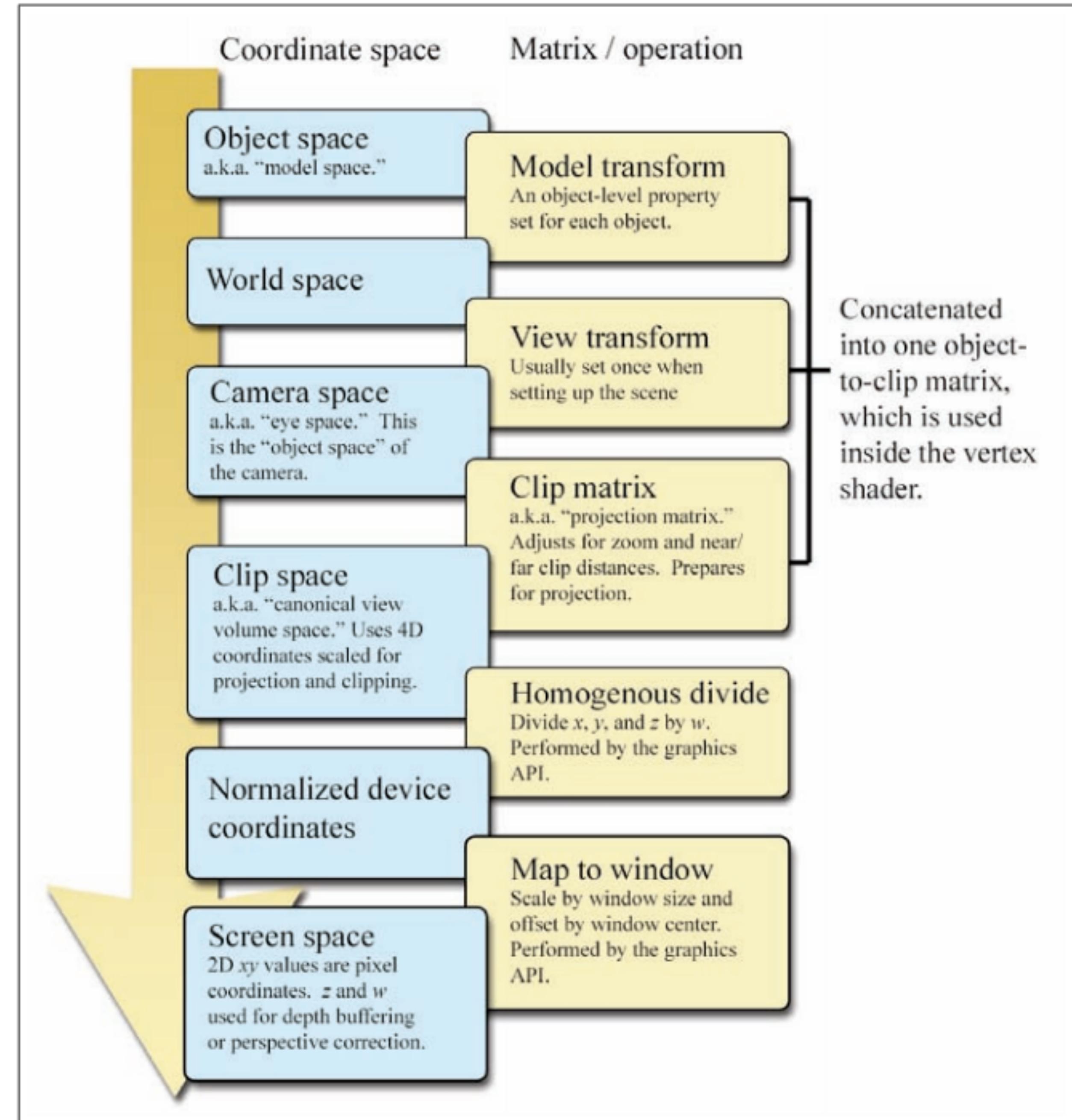
Top $y > w$

Near $z < -w$

Far $z > w$

Culling / Clipping

- If an entire primitive (line, polygon) fails the same clipping test, it is outside the field of view—if so, throw out
- If part fails and part passes, clip to the portion in view (create partial primitive) and process from there
- Clipping against multiple planes may not leave anything left — if so, throw out



To Screen Space

- Map $[-1, 1] \times [-1, 1]$ to screen
 - Scale x by half the width
 - Invert y and scale by half the height
 - Translate origin from center to upper left corner

$$\begin{bmatrix} x_{\text{screen}} \\ y_{\text{screen}} \\ 1 \end{bmatrix} = \begin{bmatrix} \text{width}/2 & 0 & \text{width}/2 \\ 0 & -\text{height}/2 & \text{height}/2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x/w \\ y/w \\ 1 \end{bmatrix}$$

Rendering Geometry

- ✓ Transform from object to world coordinates
- ✓ Transform from world to camera coordinates
- ✓ Clipping: near plane, far plane, field of view
- ✓ Perspective projection
- ✓ View transformation

Lab 7

- Repeat what you did for Lab 6 but without OpenGL
- Object placement: replace OpenGL rotate/translate calls by multiplying with your own transformation matrices
- World-to-camera: likewise replace OpenGL rotate/translate calls by multiplying with your own transformation matrices
- Projection: think about how the parameters to `gluPerspective` are used to construct a clip matrix
- Clip tests: implement your own clipping tests
 - For simplicity, clip all of a line if both endpoints fail the same clip test
 - Except clip all of a line if either endpoint fails the near-plane test
- Divide by the homogeneous element
- Map from canonical coordinates to screen coordinates
- Draw 2D lines (see the code we give you)

Coming up...

- Visibility
- Lighting



Visibility

CS 355: Introduction to Graphics and Image Processing

Two Parts of 3D Rendering

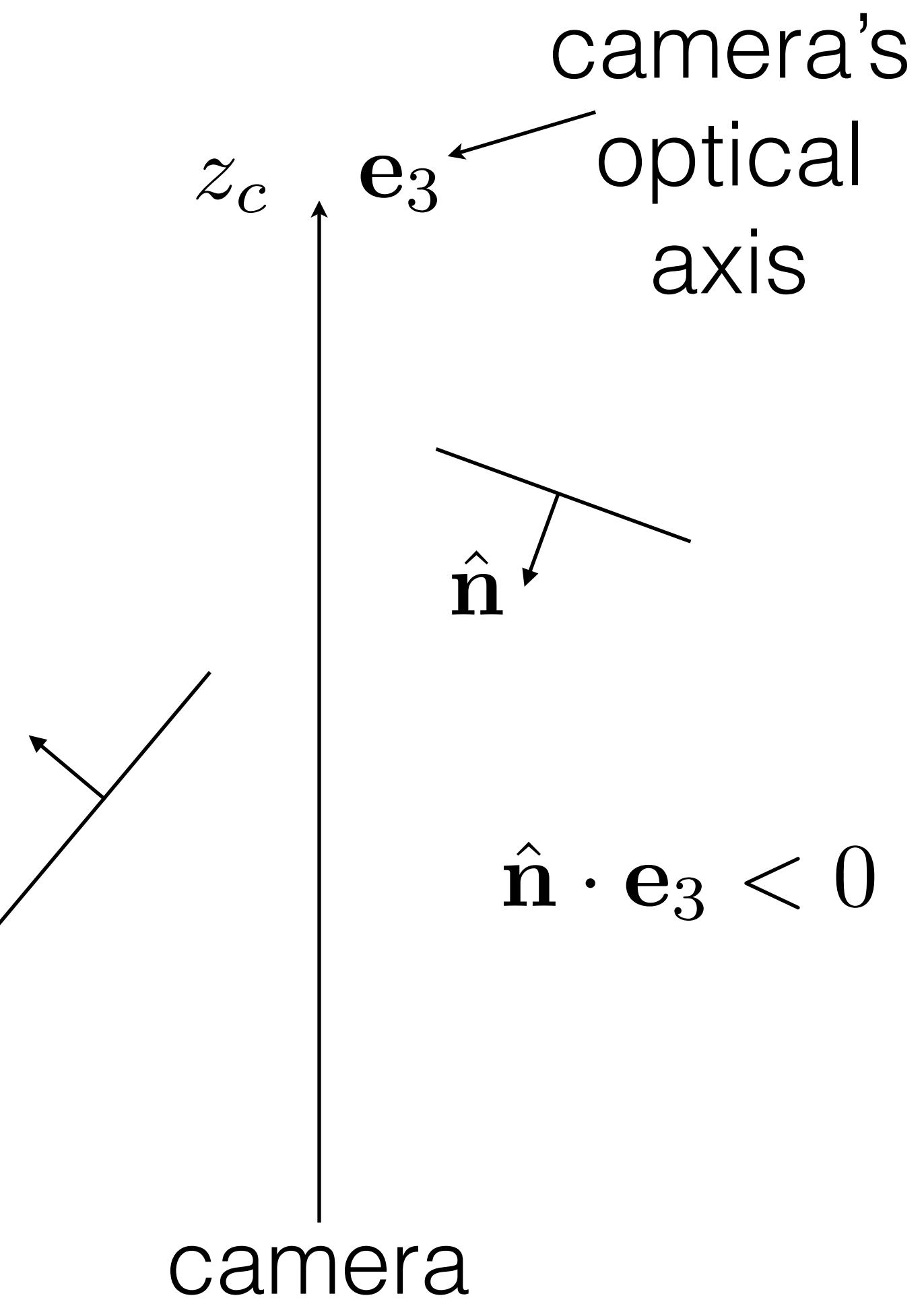
- Two key parts of 3D rendering:
 - What is visible where? (Visibility)
 - What light is coming from there? (Lighting)

Visibility

- Parts of visibility:
 - ✓ Where is everything relative to the camera?
(world-to-camera transformation)
 - ✓ What is within the field of view?
(clip matrix / clip tests)
- What is in front of what else?

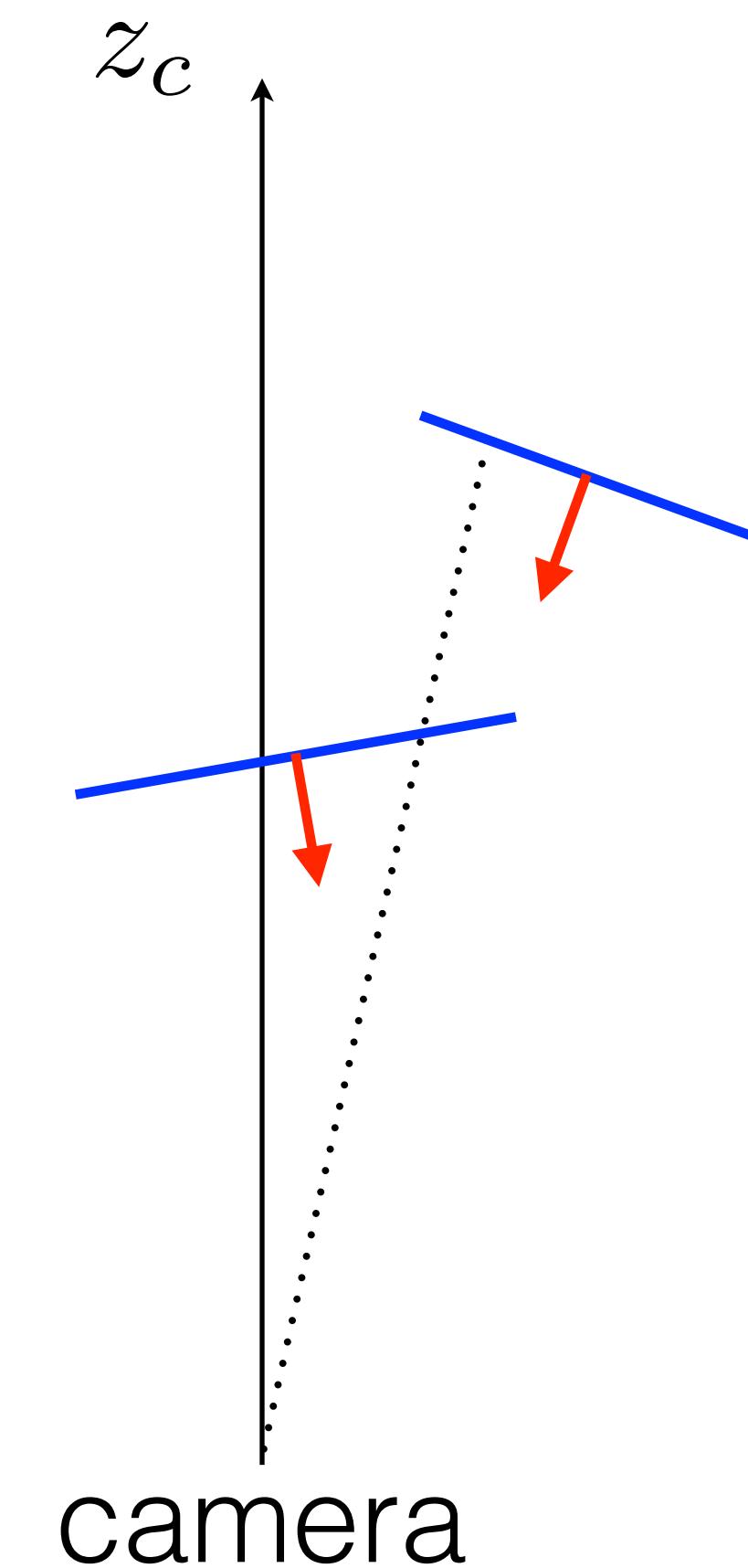
Back-Face Culling

- Simple idea:
 - Faces that point towards camera may be seen
 - Faces that point away from camera cannot be seen
 - Can do this test while still in world coordinates



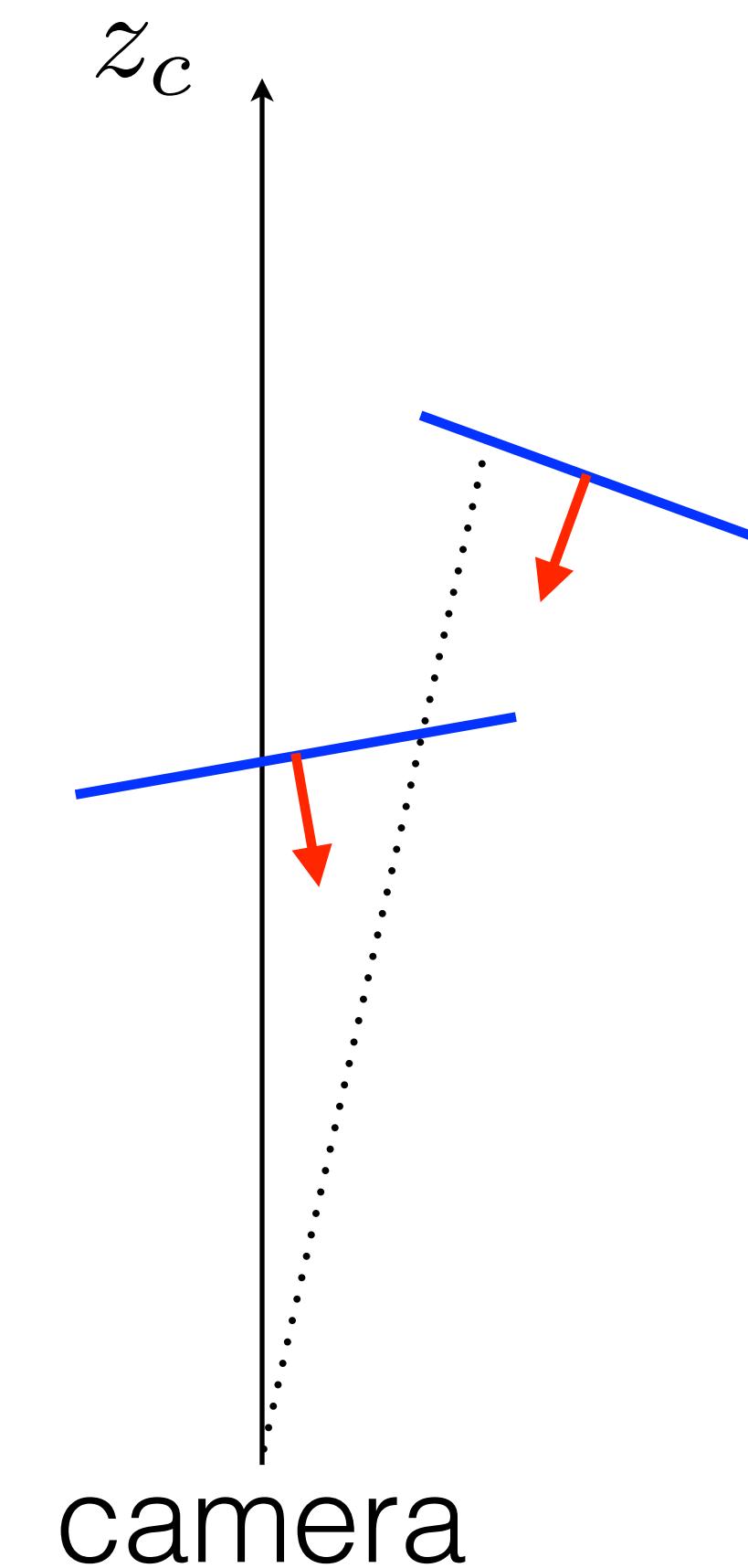
Occlusion Testing

- See what is in front of what else
- Test for things that fall on the same camera position
- If opaque, one object occludes the other



Occlusion Testing

- Three common ways:
 - Ordered rendering
(painter's algorithm)
 - Image space testing
(z-buffering)
 - Ray casting

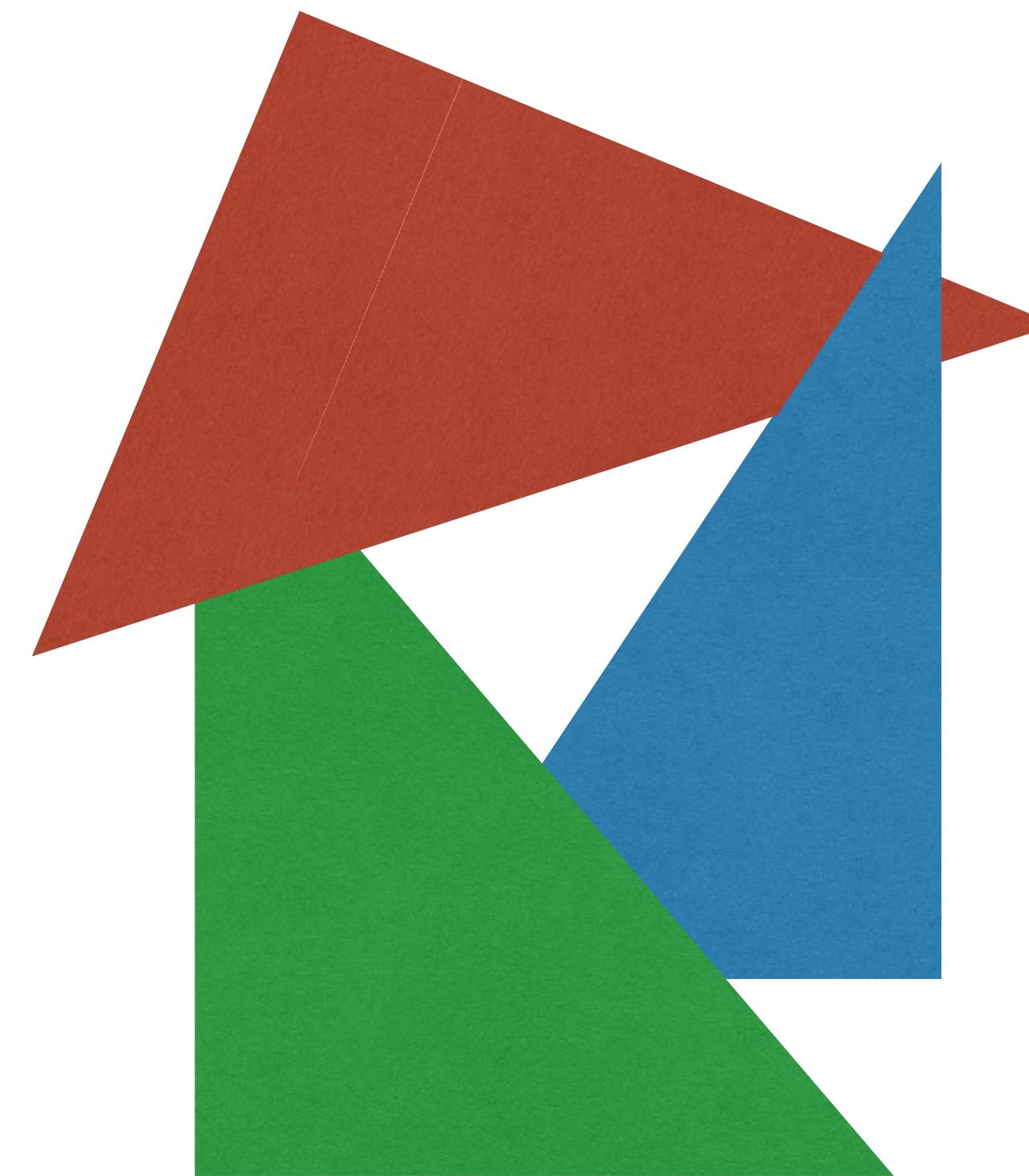


Painter's Algorithm

- Simple idea:
 - Render from back to front (decreasing z)
 - Draw things over top of others on screen
 - Last one drawn wins!
- Big problem:
 - Polygon depth isn't strictly ordered
 - Interpenetration
 - Mutually overlapping

This technique isn't used much anymore, but the idea shows up in *lots* of places

Painter's Problems



Z-Buffering

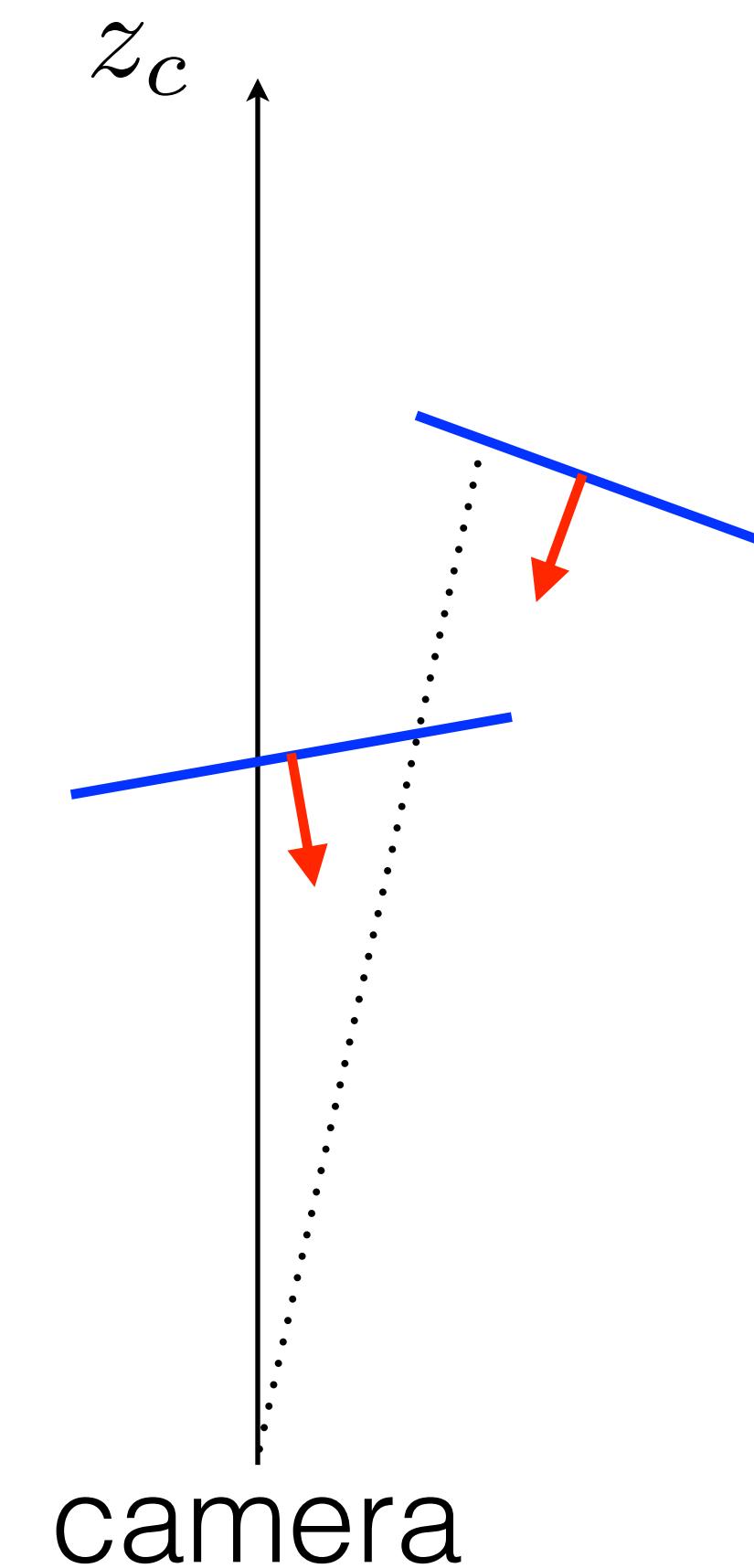
canonical screen coordinates

$$\begin{bmatrix} x/w \\ y/w \\ z/w \\ 1 \end{bmatrix} \sim \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} \text{zoom}_x & 0 & 0 & 0 \\ 0 & \text{zoom}_y & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{-2nf}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}$$

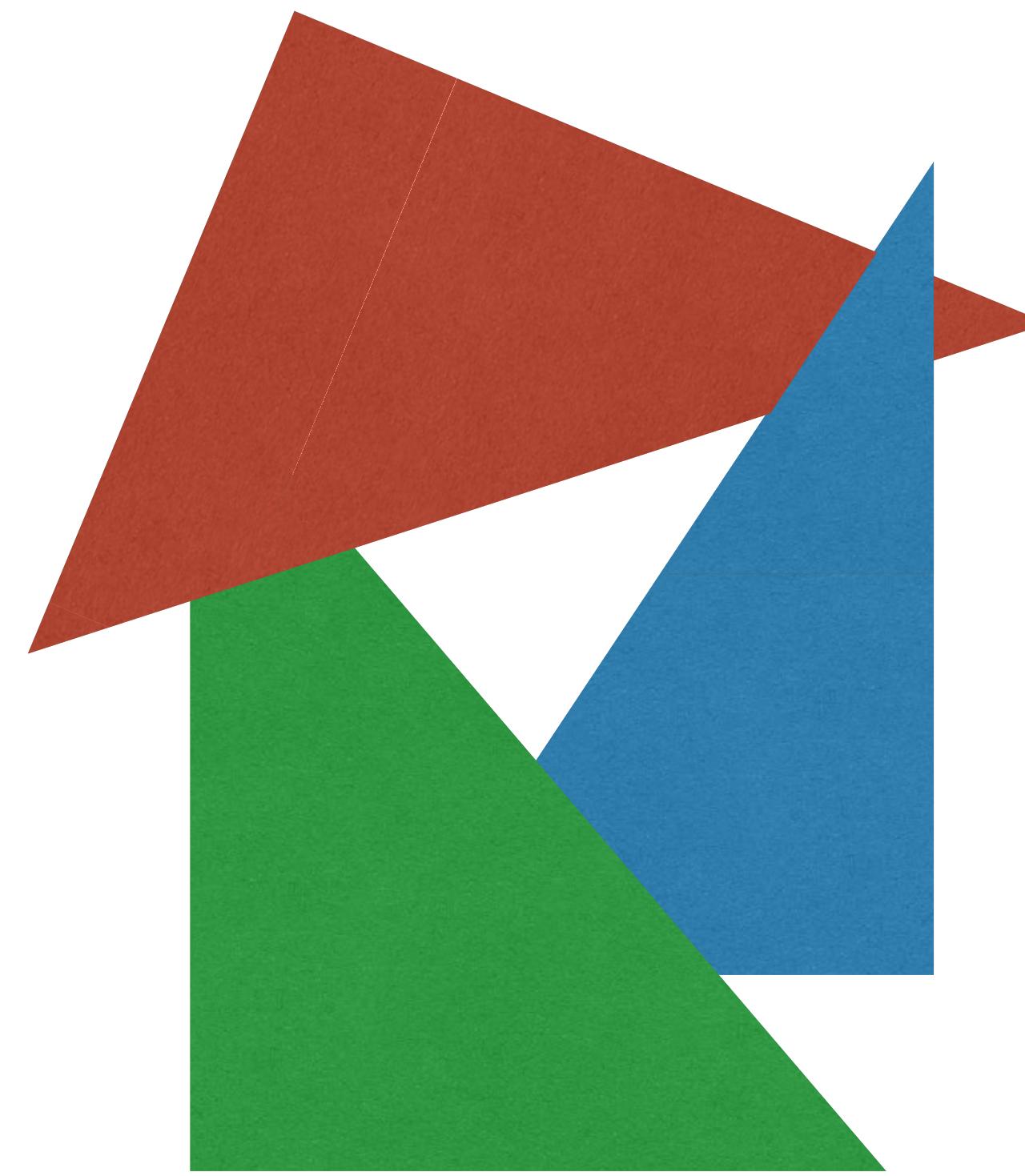
normalized depth

Z-Buffering

- Keep an image buffer that stores the depth of what is rendered at each pixel
- Render in any order
- Draw new stuff only if closer

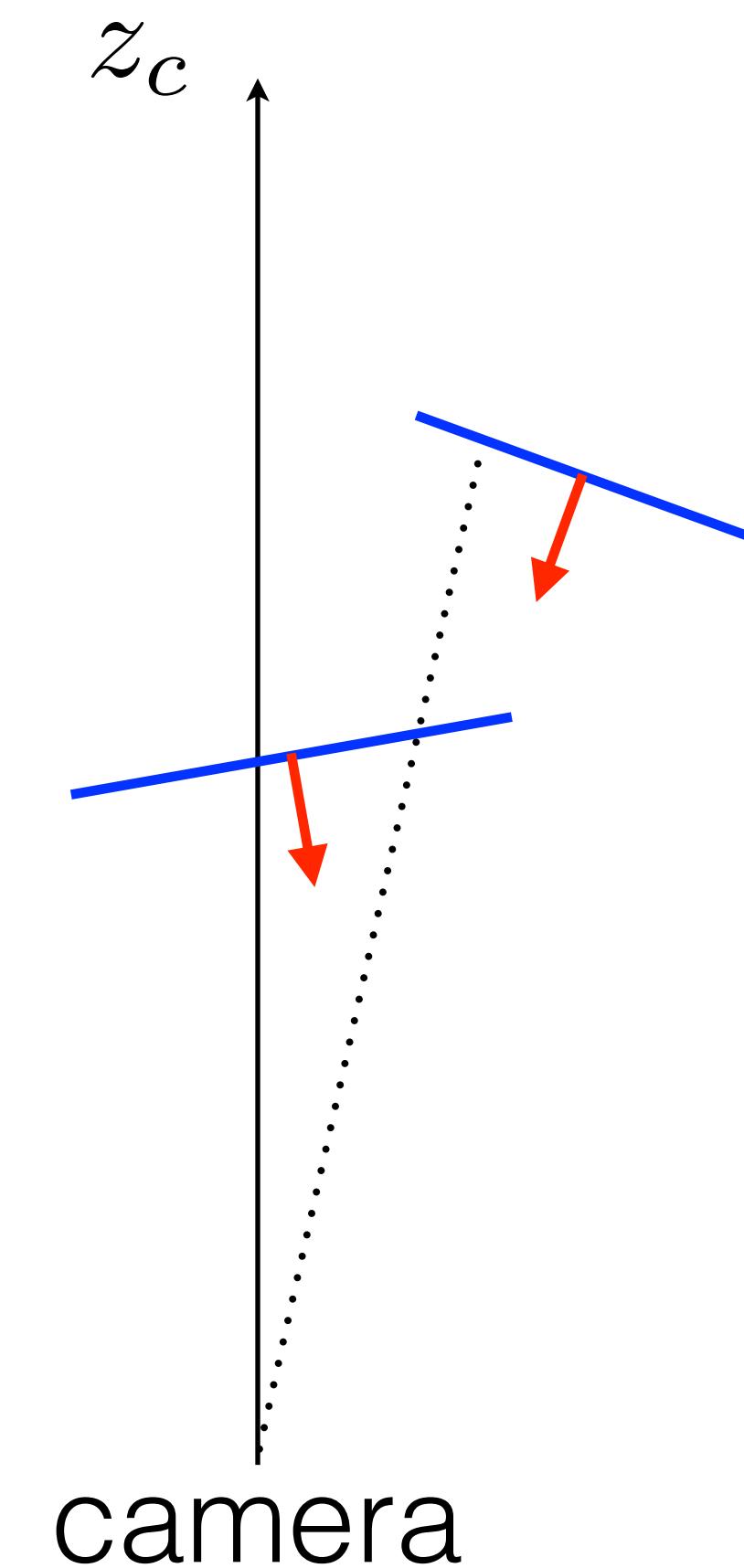


Z-Buffering



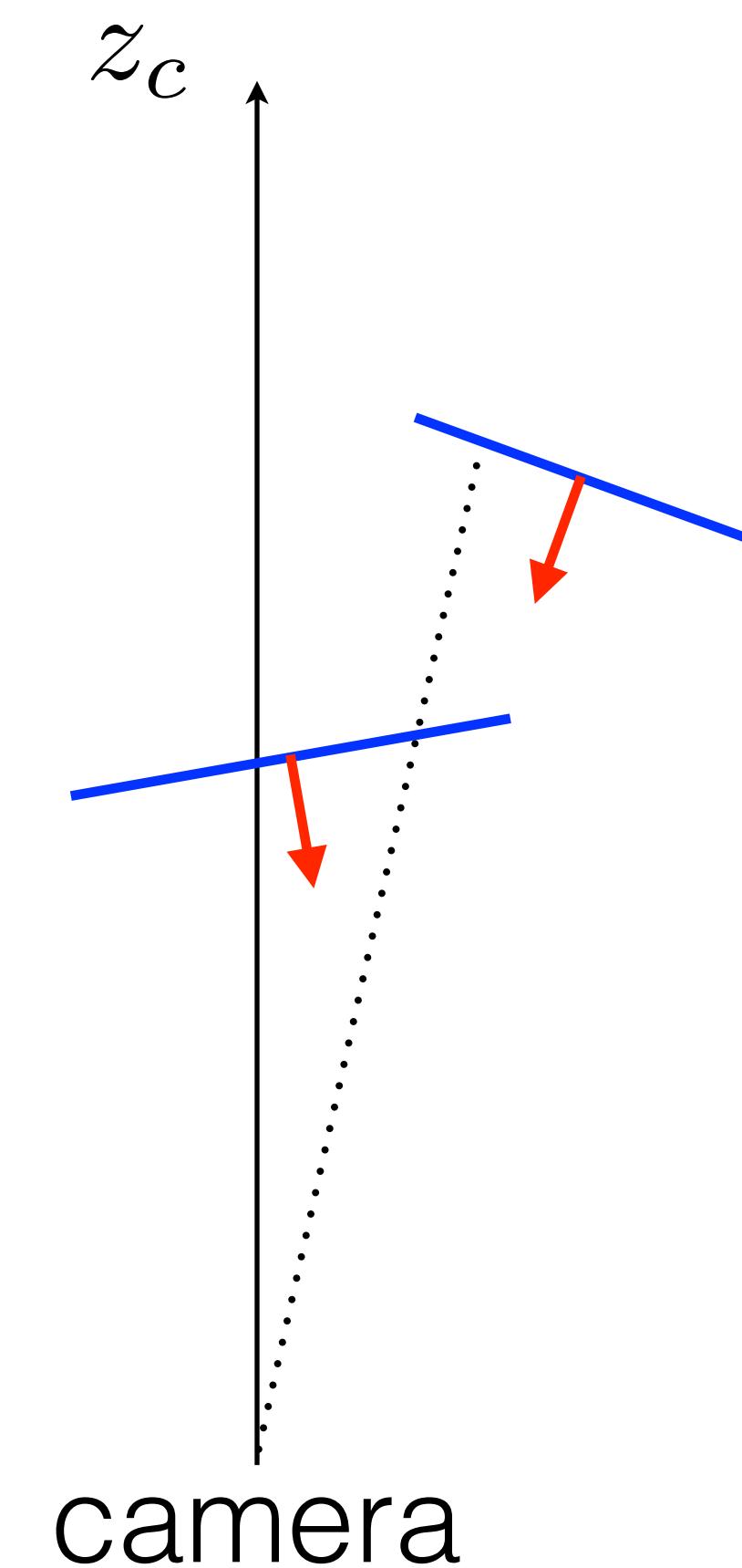
Z-Buffering

- Issue: quantization of the finite-precision z buffer
- Round-off error may be an issue
(most use floating point)
- Nonlinear by depth
(coarser farther away)



Z-Buffering

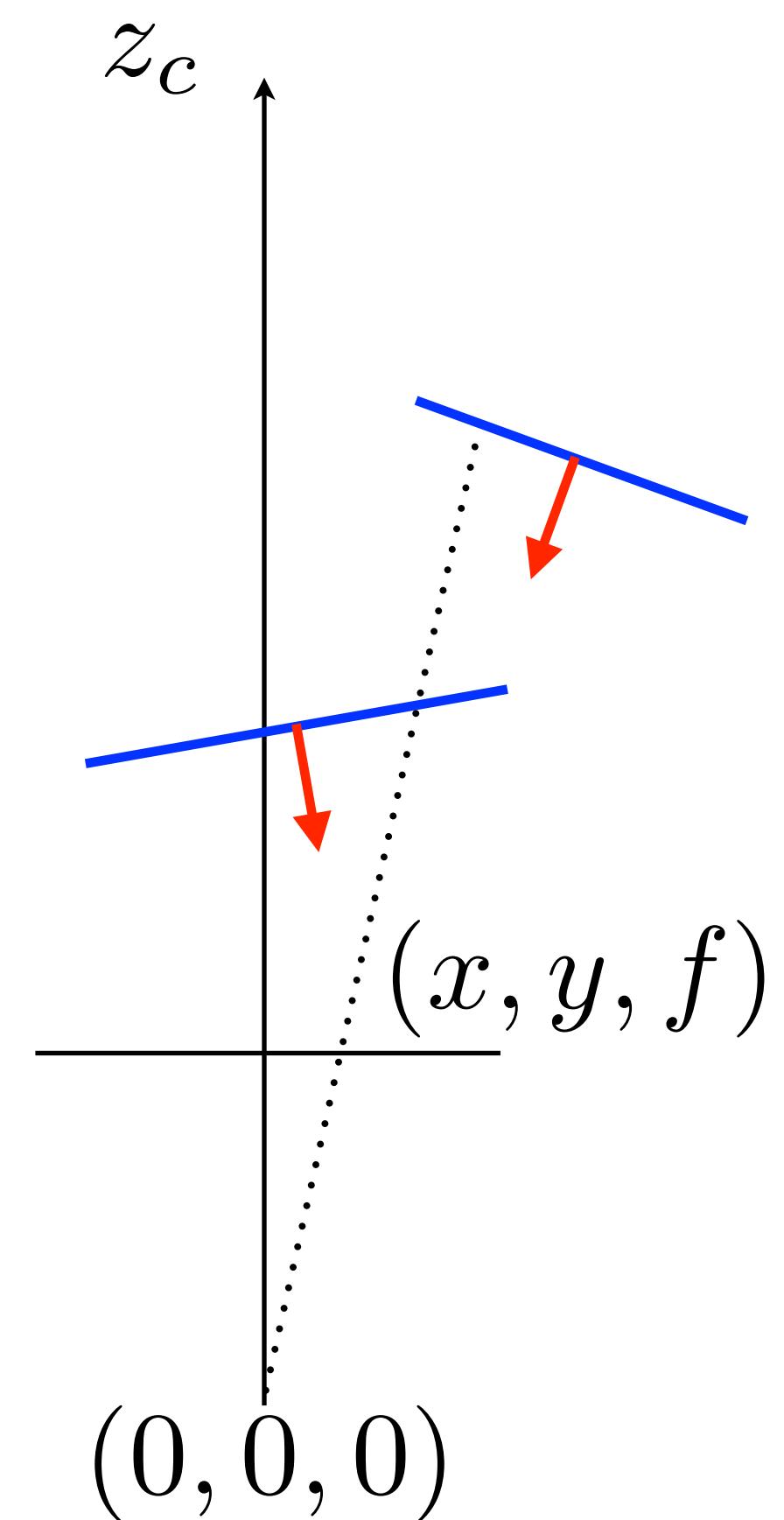
- Issue: discretization of the image z buffer
- Hard to do antialiasing (partial painting of pixels on boundaries)



Ray Casting

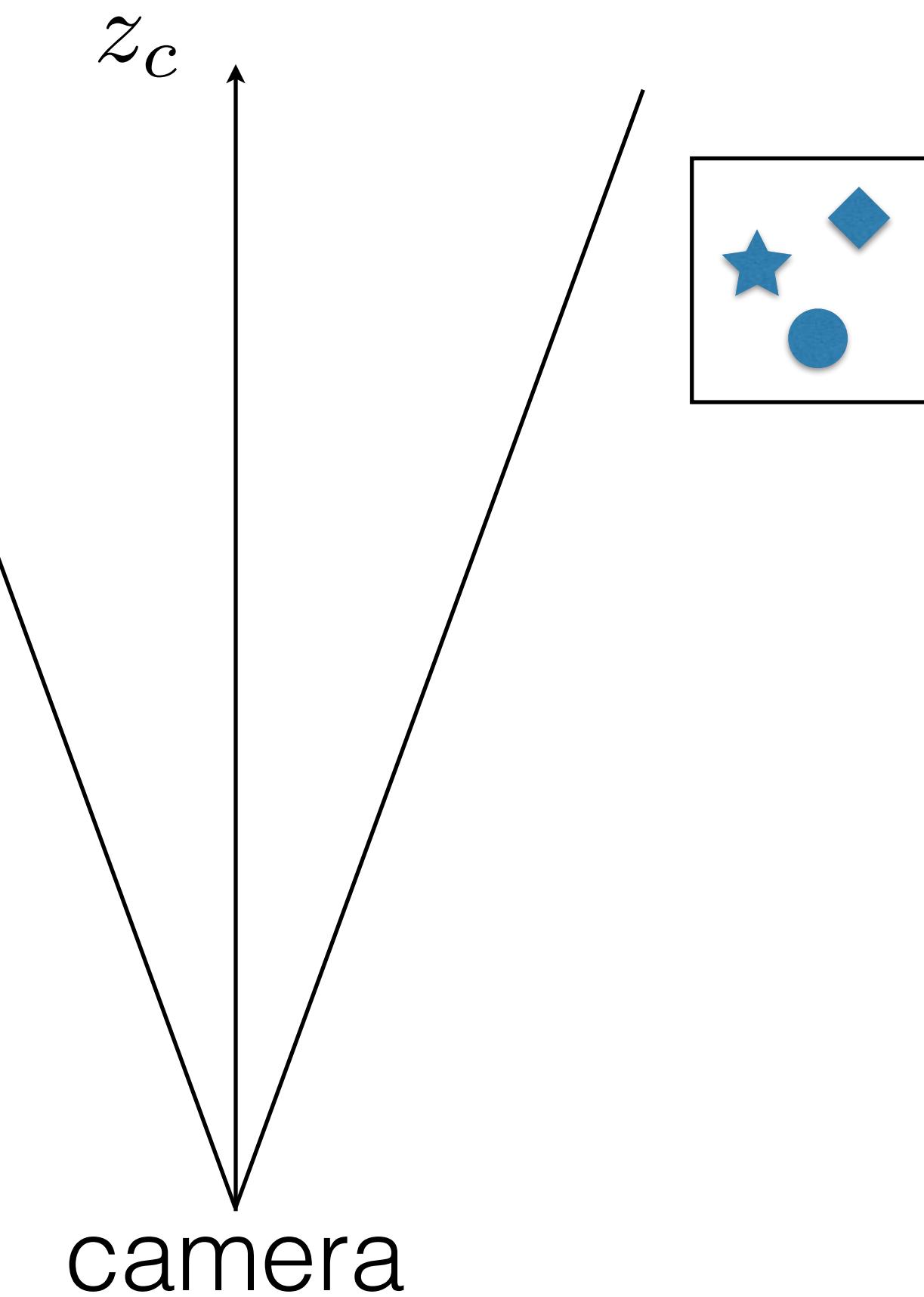
- Idea: shoot a ray out from the camera's focal point through the pixel location
- What does it hit first?
- Lots of ray-primitive intersection tests

(Arthur Appel, 1968)



Objects & Bounding Boxes

- All of these can be accelerated by grouping primitives and using **bounding boxes** (or other shapes)
- Often axis-aligned bounding boxes (AABBs) in original object space
- Transform to a general bounding box in camera space
- *Throw out if all corners are out of view, not visible, etc.*



Coming up...

- Visibility
- Lighting